論文97-34C-11-4

프로그램형 자동화기기를 위한 실시간 메카니즘 제어언어의 설계 및 구현기법

(Design and Implementation Technique of Real-Time Mechanism Control Language for Programmable Automation Equipment)

白 貞 鉉 *, 元 裕 憲 **

(Jeong-Hyun Baek, and Yoo-Hun Won)

요 약

산업 설비의 자동화 추세가 증가함에 따라 프로그램형제어기(PC), 수치제어기(NC), 분산제어시스템(DCS) 및 로봇(Robots) 제어기와 같은 프로그램형 자동화기기들의 사용이 급격히 증가하고 있다. 그러나 프로그램형 자동화기기기를 위한 프로그래밍 언어의 개발은 거의 이루어지지 않고 대부분 종래의 래더도(Ladder Chart)나 논리기호(Logic Symbol)과 같은 저급언어를 이용하는 수준에 머무르고 있다. 따라서 본 논문에서는 프로그래밍 언어의 조건문, 반복문, 선택문과 같은 구문 구조에 시간제한 구조와 실행시간 분석 구조를 추가한 실시간 메카니즘 제어언어(RTL/MCS)를 구현하고 프로그램형 레이저마킹기(programmable lazer marking machine) 제어환경 개발에 응용하였다. 또한, 실행시간 예측기법을 제안하여 언어의 번역시간에 실시간 응용프로그램의 시간제한 구문의 타당성과 전체 프로그램의 실행가능성을 예측할 수 있도록하였다.

Abstract

As the trend of the automation is increasing, the usage of the programmable automation equipments like programmable controller(PC), numerical controller(NC), distributed control systems(DCS) and robot controller is greatly expanding in the area of the industrial equipments. But the development of the programing language for the programmable automatic equipment is rarely accomplished. In this paper, we propose design and implementation technique of the real-time mechanism control language by adding time constraint constructs and timing analysis constructs to conditional statement and iteration statement of a programming language. Moreover, we made it possible to predict plausibility of time constraint constructs of a real time application program at compilation time and developing execution time analysiss technique.

* 正會員, 中京工業專門大學 電算科

(Department of computer science, Joongkyung Technical Junior College)

** 正會員, 弘益大學校 컴퓨터工學科

(Department of Computer Engineering, Hongik University)

※ 이 논문은 한국과학재단 산학협력 연구비지원에 의한 결과임(과제번호: 95-2-11-02-01-3).

接受日字:1997年4月18日, 수정완료일:1997年11月1日

I. 서 론

초기의 프로그램형 제어기(programmable controller)는 개폐기, 릴레이, 타이머, 카운터등과 같은 전기부품들을 배선하여 구성한 시이퀀스 제어반(sequence control panel)을 사용하였으며, 이는 제어공정이 바뀌거나 다른 공정에 응용할 경우 배선을 모두 해체 하여야하고, 또한 부피도 거대하고 설치비용도 증가하여 산업 설비의 자동화를 저해하는 요인이 되어 왔다. 그러나 마이크로 프로세서의 발달은 종전

의 하아드 와이어적(hard-wired)인 시이퀀스 제어반을 컴퓨터 소프트웨어화 하는데 크게 기여하게 되었다. 이는 전기 부품인 릴레이, 타이머, 카운터, 쉬프트 레지스터 등을 소프트웨어로 구현하고 이들의 제어를 프로그램화 함으로서 이루어졌다. 시이퀀 제어를 기본으로 했던 초기의 프로그램형 제어기는 마이크로 프로세서 제조기술의 발달로 소형 경량화되고, 고속처리가 실현되어 프로그래머블 콘트롤러(PC), 로봇(robot), 수치제어기(NC), 분산 제어 시스템(DCS)등에서 핵심 제어기기의 역활을 하게 되었다[7,8,9,12].

이러한 프로그램형 제어기의 실시간 메카니즘 제어는 실시간으로 동작하는 타이머와 카운터 및 각종 입출력 신호를 인식하여 처리하여야 하며, 프로그램 전체를 반복적으로 실행하여 프로그램문장의 위치에 관계없이 모든 문장들이 동시에 처리되는것과 같은 병행성을 가져야하며, 실시간제어를 위한 시간제한기능을 가져야한다[11,12].

따라서, 본 연구에서 제안하는 실시간 메카니즘 제어언어인 RTL/MCS(Real-Time Language for Mechanism Control Systems)는 실시간 표현능력을 고급 개념의 시간제한구조(time constraint construct)[3,4,5]를 갖는 프로그래밍 문장들을 설계하여 추가함으로서 실시간 제어프로그램 작성시 시간제한 표현이 자연스럽고, 시간제한과 관련된 실행과정을 가시적으로 나타내어 시간측정을 위한 디버깅 과정이 필요없이 효율적으로 실시간 응용 프로그램의 개발이 가능하도록 설계하였다. 또한 시간제한구문의 실행시간 분석 기법을 개발하여 실시간 프로그램의 시간제한 구문들이 프로그래머가 명시한 시간제한조건을 만족하는지를 분석하여 하드웨어적 실행과정 없이 컴파일시간에 프로그램의 타당성과 실행가능성을 예측할 수 있도록 하였다. 본 논문에서 제안한 RTL/MCS는 IBM/PC에서 구현하여 프로그램형 레이저마킹기의 제어환경 구축에 응용하여 언어의 실용성과 타당성을 평가하였다.

Ⅱ. 프로그램형 레이저 마킹기

자동차나 전자제품과 같은 제품 생산 라인에서 제품의 형식번호, 제조번호, 로고등 각종 심벌들의 마킹(marking)은 제품마다 인쇄내용이 다르고 종류와 형식이 다양하여 일괄적으로 제작하여 첨부할 수 없으므로 기본 유형에 수기하거나 타자기로 직접 타이핑하여 사용해야 하는 불편함으로 인하여 관리가 복잡하고 시간이 오래 걸려 생산성을 저하시키는 요인이 되어 왔다. 그러나 레이저 기술을 응용한 마킹기(Laser Marking Machine)는 다양하고 뛰어난 마킹 기능들이 컴퓨터에 의해서 제어되는 근 적외선 레이저 광속을 이용하여 마킹함으로 고속으로 섬세한 고품위의 인쇄 효과를 얻을 수있다. 그림 1은 본 연구에서 구현한 실시간 제어언어 RTL/MCS의 실행환경인 프로그램형 레이저 마킹기의 구조이다.

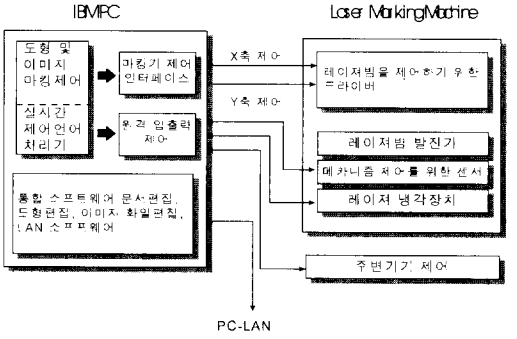


그림 1. 프로그램형 레이저마킹기
Fig. 1. Programmable Lazer Marking Machine.

레이저 마킹기의 원리는 도면의 입출력에 사용되는 X-Y 플로터와 같이 X축, Y축으로 자유롭게 움직이는 축에 펜을 끼워 원하는 도형을 그리듯이 레이저 마킹기는 펜대신 레이저 빔을 발사하여 피사체를 태워서 도형을표시한다. 그림 1에서 PC측의 마킹기 제어 인터페이스는 레이저 마킹기가 가공하려는 이미지에 따라서 X-Y축으로 레이저빔을 제어하기 위한 제어신호를 발생시킨다. 이는 전형적인 수치제어기(NC)의 원리와 같이 전압 혹은 펄스에 의해서 지령된 위치만큼 제어대상을 움직여 준다. 레이저 마킹기의 주요부는 레이저 발진기와 레이저 빔의 각도와 강약, 그리고 레이저빔의 ON/OFF를 제어하는 드라이버 부분이다. 드라이버는 PC측에서 보내진 위치이동 데이타에 따라서 레이저빔 반사경을 정확히 움직여 준다. 본 논문에서 제안한 실시간 제어언어(RTL/MCS) 처리기에 의해서 제어되는 주요 제어대상은 가공물의 반송과 모니터링 등의 주변기기 제어와, 마킹명령의 실행, 그리고 마킹기의 안전한 운전을 위한 켜짐과 꺼짐동작등을 제어한다. 따라서 실시간 제어언어처리기 부분이 프로그램형

레이저마킹기 시스템의 운영을위한 핵심 역할을 수행한다. 실시간제어언어 처리기는 인터프리터로 구현되며, 실행주기는 그림 2와같다.

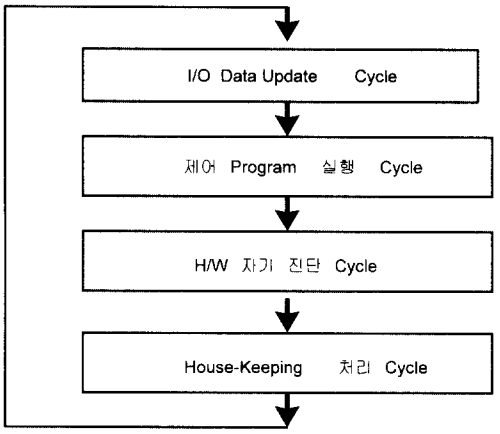


그림 2. 실시간제어언어 처리주기
Fig. 2. Scan cycle of real-time control language.

Ⅲ. 실시간 제어언어: RTL/MCS

본 논문에서 제안한 실시간 제어언어인 RTL/MCS의 프로그램은 제어문장들로 구성되어 반복실행되는 cycle블럭들과 함수로써 이루어 진다.

1. RTL/MCS의 특성

(1) Cycle Block

Cycle은 제어공정상 하나의 독립된 기능이나 상황과 관련된 제어문장들의 그룹으로 구성된다. 즉, 각 cycle들은 완전히 독립되어야 하며, 하나의 cycle 안의 제어문장들은 해당 cycle의 기능과 관련되어 의존성을 가질수 있다. 그러므로, 여러개의 cycle들이 모여서 하나의 RTL/MCS 프로그램을 이루며, 시스템의 제어 공정에 따라서 각 cycle들이 활성화(active)된다. 또한 활성화된 cycle들은 그림 3에서 보는바와 같이 cycle 리스트의 맨 처음에서 시작하여 끝까지 반복적으로 실행하며, cycle 리스트 전체를 한번 실행한 시간을 scan주기(scantime)라 한다[7,8,9].

(2) Cycle 상태

Cycle 블록은 운영체제의 프로세스와 같이 실행 상태를 전이한다. 대기상태에있는 cycle 리스트들은 프로그램의 start나 call 문장에 의하여 실행상태로 전이된다. 활성화된 cycle 리스트는 동시에 여러개가 존재하여 반복실행 되므로서 병행성을 갖게된다. 그러나 활성화된 cycle들 중에서 wait, delay, call, when, every와 같이 조건이나 시간제한을 기다리는 cycle들은 중단상태(suspend)로 전이하여 조건이 만족될 때까지 실행이 중지된다. 그러나 각 scan 주기마다 이들의 조건이 검사되어 조건이 만족되면 다시 활성화 상태로 전이되어 반복실행되고, cycle의 실행이 완료되거나 return, exit문에 의하여 반환이 이루어지면 cycle은 대기상태로 된다. 그러므로 RTL/MCS의 cycle들은 반드시 프로그램 문장에의해서 명시적으로 상태전이가 이루어지며, 이는 실시간언어의 기본 조건인 프로그램의 실행상태를 정확히 분석 및 예측할 수 있도록 해준다. 그림4는 cycle들의 상태전이 조건과 과정들을 나타내준다.

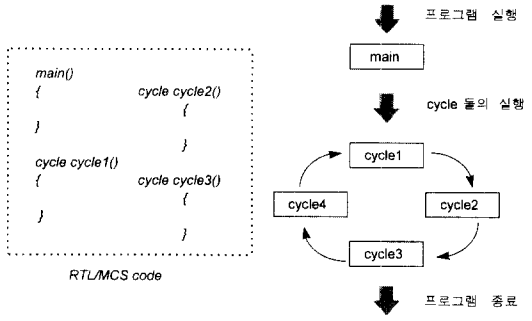


그림 3. RTL/MCS 프로그램 실행주기
Fig. 3. Program execution cycle of RTL/MCS.

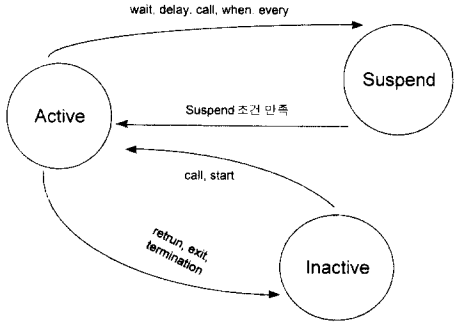


그림 4. Cycle의 상태전이
Fig. 4. State transition of Cycle.

2. 언어의 정의

실시간 메카니즘제어언어인 RTL/MCS는 C 언어의 기본 구조에 실시간 프로그래밍언어의 필수 조건인 시간제한 구조를 포함하고 있다. 이들 문장구조는 시간지연구조, 시간제한구조, 그리고 cycle을 제어하기 위한 문장들로 구성된다. 그림 5에 RTL/MCS의 주요 문장구조의 EBNF[1]를 나타내었다.

(1) 시간 제한 구문

① 시간지연 : delay문은 워드수식(word_expression)으로 지정한 시간동안 cycle의 실행을 중지시킨다. 또한 delay until문은 워드수식으로 주어진 시간(calendar clock)까지 cycle의 실행을 중지시킨다. 지연시간의 오차는 프로그램의 전체 scan 시간에 비례하여 커진다.

② 시간제한 : Within문은 문장들이 실행 되어야할 마감시간(deadline)을 지정한다. Timeout절은 문장들은 실행시간이 제한시간을 초과하게 되면 즉시 실행된다. 이때 within문안의 delay, call. when, every문들에 의해 실행되거나 중단된 cycle들의 실행도 종료된다.

③ 주기적 반복문 : every문은 지정된 주기에 따라서 문장들이 반복적으로 실행된다. 즉, 워드수식으로 지정된 시간만큼 주기적으로 문장들이 실행되며, 이때의 주기는 문장들의 실행시간과는 관계없이 절대적인 시간 간격을 의미한다.

(2) 메카니즘 동기화 구조

① 대기 : wait문은 비트수식(bit_expression)으로 주어진 상태가 참이될 때까지 cycle의 실행을 중지(suspend) 시킨다. 이때 중단된 cycle은 각각의 scan cycle마다 중단조건이 평가되어 결과가 참일 때 중단된 다음 문장부터 실행이 재개된다.

② 예외처리문 : condition문은 비트수식이 참일때 실행된다. 비트수식은 매 scan 주기마다 do 문의 문장들이 실행되기 전에 평가되고, 또한 cycle 실행을 중단시키는 어떠한 문장을 포함하고 있을때 평가된다.

③ 랑데부 : When문은 조건문으로서 비트수식이 참일 경우 대응되는 문장을 실행한후 when문을 벗어난다. 그러나 조건절의 값이 모두 거짓이면 해당 cycle은 실행이 중단되어 조건이 하나라도 만족되면 실행을 재개한다.

(3) Cycle 제어문

① 시작문 : Start문은 해당 cycle을 활성화(active)시켜 실행큐(queue)에 추가하므로서 다음 scan 주기부터 실행된다. 만일 cycle이 활성화 상태이면 start문은 아무런 영향도 주지 않는다. 활성화된 cycle 리스트들은 순차적으로 실행(scan)되기 때문에 활성화시킨 cycle은 다음 scan 주기때까지 대기하여 실제로는 다음 스캔주기에 실행된다. 또한 start문을 포함하는 기존의 cycle도 중단 되지 않는다.

② 호출문 : call문은 호출된 cycle을 활성화시킨다. 동시에 호출한 cycle은 호출 당한 cycle이 종료될 때까지 실행이 중지되며, 호출당한 cycle이 종료되어 호출한 cycle의 실행이 재개될 때에는 call 다음 문장부터 실행된다.

③ 종료 : exit문은 현재 실행중인 cycle을 종료시킨다. 만일 cycle이 call이나 start문에 의하여 다시 활성화되면 cycle은 exit 다음 문장부터 실행을 재개한다.

```
delay_statement ::= DELAY '(' word_expression ')' ( MSEC | SEC | MIN )
                  | DELAY UNTIL word_expression ':' word_expression
within ::= WITHIN word_expression ( MSEC | SEC | MIN )
                  [ ON TIMEOUT statement_list ] DO statement_list
every ::= EVERY word_expression ( MSEC | SEC | MIN ) statement_list
        | EVERY word_expression ( MSEC SEC | MIN )
          WHILE '(' bit_expression ')' statement_list
wait_statement ::= WAIT '(' bit_expression ')'
                  [ TIMEOUT number (MSEC | SEC | MIN) ]
condition_do ::= CONDITION '(' bit_expression ')'
                  ON EXCEPTION statement_list DO statement_list
when_statement ::= WHEN bit_expression ':' statement_list
      { OR bit_expression ':' statement_list } TIMEOUT word_expression;
start_statement ::= START cycle_name
exit_statement ::= EXIT
return_statement ::= RETURN
if_statement ::= IF '(' bit_expression ')' statement_list
                  { elsie '(' bit_expression ')' statement_list }
while_statement ::= WHILE '(' bit_expression ')'
            [TIMEOUT number | CONSTRUCT number] statement_list
for_state ::= FOR '(' word_expression ';' bit_expression ';' word_expression ')'
            [ COUNTOUT number ]
forever ::= FOREVER statement_list
set_statement ::= SET '(' bit_term { ',' bit_term } ')' ( ON | OFF | IRV )
                | SET ')' bit_term { ',' bit_term } ')' TO bit_expression
let_statement ::= [ LET ] word_term '=' word_expression
word_expression ::= word_expression ( '+' | '-' | '*' | '/' | '%' )
                  word_expression | word_term
word_term ::= word_name | number | word_function
bit_expression ::= bit_expression ( '&' | '|' | '!=' | '^' ) bit_expression
                | bit_term
bit_term ::= bit_name | bit_relation | bit_selector | cycle_name
            | bit_vector | bit_function
bit_relation ::= word_expression ('<' | '>' | '<=' | '>=' | '==' | '<>' )
                word_expression
statement_list ::= '{' { statement ';' } '}'
cycle ::= cycle_name '(' ')' var_decl statement_list
var_decl ::= { ( WORD | BIT ) identifier { ',' identifier } ';' }
program ::= main_function { cycle }
```

그림 5. RTL/MCS 구문의 EBNF
Fig. 5. EBNF of RTL/MCS.

Ⅳ. RTL/MCS의 의미분석

이 장에서는 RTL/MCS의 주요 구문들에 관한 의미분석과 중간코드로의 번역규칙을 기술한다. 특히 cycle과 시간제한 구조와 관련된 구문들의 번역과정을

중심으로 설명한다.

1. Cycle

Cycle 선언에 관련된 중간코드는 StartCycle과 EndCycle이 있다. 그림 6에서 StartCycle은 cycle이 실행되기위한 매개변수 저장과 초기화를 실행한다. 또한 EndCycle은 현재의 cyle을 실행큐에서 제거하고 call문에 의해 호출되었을 경우 상태레지스터를 참조하여 호출한 cycle을 실행큐에 삽입한다.

```
cycle  monitor()                    StartCycle  27
{                                   Load  1,17
    a1 = dev17;                     Store  3,45
    if (a1*60 > 1000) {             Pop
        .....                           ......
        .....                           ......
    }
}                                   EndCycle
```

그림 6. Cycle 의 코드생성 규칙
Fig. 6. Code generation rule of "Cycle" statement.

2. within

Within문의 구현은 그림 7과 같이 실시간 타이머를 이용한다. RTL/MCS의 중간코드에는 타이머를 설정하고 제거하기 위한 SetTimer와 DeleteTimer가 있다. SetTimer는 제한시간과 timeout handler를 설정할 수 있다. within문을 중간코드로 표현할 때는 구문을 시작하면서 타이머를 설정하고 구문을 마치면서 타이머를 소거함으로써 구현할 수 있다. 그림 7은 within문에 대한 중간코드의 번역과정이다.

```
within  100 ms                              SetTimer  T002,100,LT2
on timeout                          LT2:    Jump  L17
{  TimeOutCmd  = DefaultTimeOutCmd  ;       Load  3, 57
   .....                                    Store  2,67
   .....                                        .....
}                                           .....
do {                                        .....
   B1Status =  DefaultData  ;               Delete  T002
   .....                                    Jump  L18
   .....                            L17:    Load  4, 56
}                                           Store  2, 45
                                            .....
                                            Delete T002
                                    L18:    .....
```

그림 7. within의 코드생성 규칙
Fig. 7. Code generation rule of "within".

3. every

Every문은 설정된 시간의 주기를 가지고 반복적으로 cycle블럭을 실행한다. 따라서, every문도 그림 8과 같이 Timer를 사용하는 중간코드를 이용하여 구현한다. 다음은 every문에 대한 중간코드 번역 과정이다. 중간코드 번역에서 ①의 SetTimer는 제한시간을 every문의 실행주기로 설정한다. 실행주기 내에 every문의 실행을 마치지 못하면 자동적으로 defualt timeout handler가 호출되어 프로그램의 실행이 정지된다. ②의 SetTimer는 every문이 설정한 주기의 간격으로 수행될 수 있도록 현재 실시간 타이머에 남아있는 값을 재설정(reload) 설정하고 timeout handler에서 cycle을 기억된 위치에서부터 다시 실행할 수 있도록한다. 그리고, ③의 DeleteRun을 이용하여 every문이 속한 cycle을 Ready 리스트에서 제거한다. ④의 ExitCycle은 다음번에 cycle을 실행할 때 시작위치를 every문의 시작점으로 설정하게 된다.

②의 timeout handler(ET05)는 중지된(suspend) cycle을 설정주기가 경과한후 활성화큐에 추가하여 cycle이 다시 실행될 수 있도록 한다.

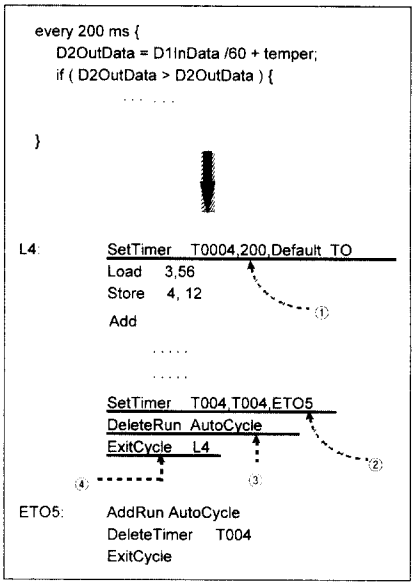


그림 8. every문의 코드생성
Fig. 8. Code generation rule of "every" statement.

V. RTL/MCS의 실행시간 분석

1. RTL/MCS의 실행시간 분석기법

RTL/MCS의 각 문장 요소들에 대한 실행시간을 MET(Maximum Execution Time)라고 정의할때 RTL/MCS의 프로그램에 대한 최악 실행시간

WCET(Worset Case Execution Time)의 분석은 표 1에서 기술한 방법으로 계산할 수 있다.

표 1. RTL/MCS의 실행시간 계산기법
Table 1. Execution time computation technique of RTL/MCS.

구문	최악실행시간(WCET)
순차문장	Σ MET(문장)i
if문	MET(조건절) + max(MET(then)i, MET(elif)i)
while문	MET(조건절)+LoopBound*(MET(body)-MET(조건절))
within	if(설정시간 > MET(body)) then WCET = MET(body) else WCET = 설정시간
every	if(설정주기 > MET(body)) then WCET = MET(body) else WCET = 설정주기
when	if(max(MET(조건절)i) > timeout) then WCET= timeout else WCET = max(MET(조건절)i)
condition	MET(조건절)+max(MET(on exception),MET(do의 body))

표 1과 같이 단순한 문장들의 나열은 각 문장들의 실행시간의 합이 된다. if문의 경우에는 조건을 만족하여 실행하는 then부분과 만족하지 않아 실행하는 else 부분의 실행시간중 큰 것과 조건 검사하는데 걸리는 시간의 합이 if문의 최악 실행시간이 된다. while문은 조건 검사시간과 몸체(body)의 실행시간의 합에 프로그래머가 지정한 경계(bound) 값을 곱한것과 while문을 빠져나올 때의 조건 검사시간을 더한 것이 while 문의 실행시간이 된다.

2. if문의 실행시간분석

그림 9은 if문의 실행시간을 분석하기위한 시간스택 생성 규칙을 yacc를 이용하여 정의한 내용이다.

```
if_statement : if_prefix { tsp++; } statement
               { timeStack[tsp -2] = timeStack[tsp -1]
                                   + timeStack[tsp]; tsp -= 2; };
             | if_prefix { tsp++; } statement
               ELSE { tsp++; } statement
               { if( timeStack[tsp] + timeStack[tsp-2]
                        > timeStack[tsp-1] + timeStack[tsp -2] )
                 {  timeStack[tsp-3] = timeStack[tsp]
                                     + timeStack[tsp -2];
                 } else { timeStack[tsp - 3] =
                            timeStack[tsp -1] + timeStack[tsp -2];
                          tsp -= 3;
                          }
               };
if_prefix  :  IF '(' { tsp++; }  bit_expression ')' ;
```

그림 9. if문의 시간스택생성
Fig. 9. Timing stack generation of "if" statement.

시간스택포인터인 tsp의 증가는 그림10에서 template을 스택에 push하는것과 같다.

if문의 실행시간 분석은 문장을 parsing하기 전에 stack에 시간을 저장하기 위한 template을 그림 10의 ①과 같이 push한다. 이 template에 조건검사를 위해 필요한 문장들의 실행시간을 저장한다. 그리고, then부분을 parsing하기 전에 그림 10의 ②처럼 template을 push한다. 이 template에 then부분 문장들의 실행시간을 저장한다. 마찬가지로 else부분을 parsing하기 전에 그림 10의 ③과 같이 template을 push하고 이 template에 else부분의 실행시간을 기록한다.

if문의 parsing이 끝나는 부분에서 그림 10의 ④와 같이 then부분의 template과 else부분의 template을 꺼내 큰 곳을 조건검사의 template과 더해서 기존의 맨 위에 쌓여있던 template에 더한다.

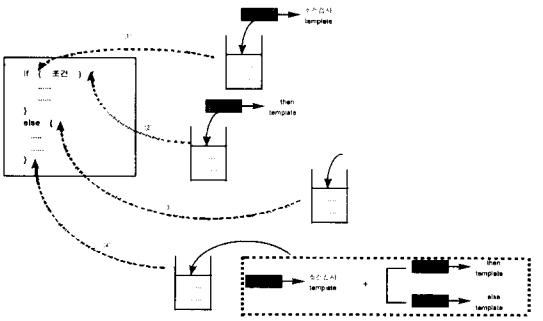


그림 10. if문의 실행시간 분석과정
Fig. 10. Execution timing analysis of "if".

3. 스캔시간(Scantime) 분석

RTL/MCS로 작성된 실시간제어 프로그램의 실행가능성 분석은 작성된 응용프로그램이 허용하는 최대 시간오차인 scantime을 프로그래머가 지정한 제한 시간내에서 실행가능한지 분석하여 예측할 수 있다. 이는 각 scan주기에서 동시에 활성화(active)된 cycle들의 실행시간을 모두 더한것과 같다. 그러나 scantime의 분석은 각 cycle문장안에 every, wait, delay, when등의 문장들이 cycle을 실행중단(suspend) 상태로 전이시키기 때문에 scantime에 영향을주는 각 cycle의 실행시간은 cycle내에 있는 모든 문장들의 실행시간이 아니라 어느 한 scan주기에 활성화되어 실행되는 문장들만 계산하기 때문에 scantime분석을 위한 cycle의 실행시간은 이 문장들을 경계로 나뉘어진다. 그러므로 나뉘어진 경계들의 실행시간 중에서 가장 큰 시간을 각 cycle별로 계산하여 더하면 전체 프

로그램의 최악 scantime이 된다. 따라서 이 최악 scantime이 프로그래머가 명시한 scantime 값보다 작으면 작성된 응용프로그램은 해당제어기에서 실행가능하게 된다.

```
wait_statement :  WAIT '(' bit_expression ')'  TIMEOUT NUMBER
                  {  if( maxScanTime < timeStack[tsp] )
                        { maxScanTime = timeStack[tsp] };
                  };
delay_statement :  DELAY { tsp++; } '(' word_expression ')' MSEC
                  {  if( maxScanTime < timeStack[tsp] )
                        { maxScanTime = timeStack[tsp] };
                  };
```

그림 11. 스캔시간 분석기법
Fig. 11. Scan time analysis technique.

그림 11은 cycle들의 scan 시간을 분석하기위한 시간스택 생성과정을 wait와 delay문에대하여 yacc를 이용하여 정의하였다.

VI. RTL/MCS의 구현

본 논문에서는 RTL/MCS를 Unix환경에서 gnu c++, yacc, lex를 이용하여 구현 하였다. Unix환경에서 구현한 RTL/MCS는 크게 세가지 부분으로 이루어졌다. 그림 12은 Unix환경에서 구현한 RTL/MCS의 언어처리기 구성을 나타내었다.

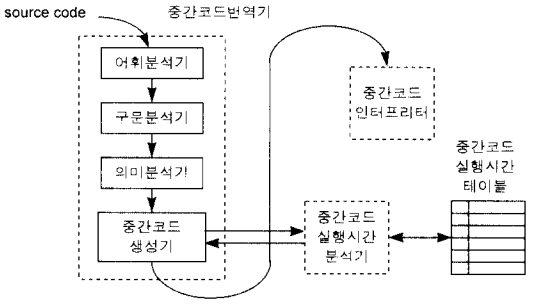


그림 12. 언어 처리기의 구성
Fig. 12. Structure of language processor.

1. 커널(kernel)의 구조

RTL/MCS의 cycle은 제어문장들의 그룹으로서 병행적으로 실행되어야 한다. 따라서 cycle의 구현은 커널의 과부하(overhead)가 적고 효율적으로 실행할 수 있는 다중 스레드(multi-thread)를 기반으로 설계되었다. 즉, RTL/MCS의 프로그램은 여러개의 다중스레드로 이루어진 단일 프로세스이다. 따라서 각각의 스레드는 공유데이타와 커널의 자료구조들을 공유하는 반면, 각 cycle의 텍스트 코드와 프로그램 카운터, 스택포인터등이 저장된 상태레지스터와 스택을 가지고 있다. 그림 13은 이러한 다중스레드형 프로세스의 구조를 기반으로 하는 RTL/MCS의 커널구조를 나타내었다. 주함수(main)와 cycle들로 이루어진 RTL/MCS응용프로그램은 그림 13에서와 같은 구조로서 메모리에 적재된다.

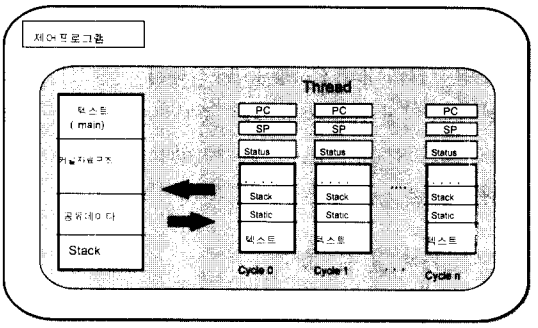


그림 13. RTL/MCS 커널의 구조
Fig. 13. Kernel structure of RTL/MCS.

```
Class CycleBlock {
  char CycleName[MAXIDSIZE];
  int no;
  int pc, sp;
  int *status;
  int *stack;
  CodeBlock* IB;
public:
 CycleBlock(char *name);
 void AddInstruction(Code* code);
 void Exec();
};
class Cycle {
 CycleBlock* CBs[MAXCYCLE];
  int cycleno;
public:
  Cycle();
  void LoadCycleName(char* );
  void AddInstruction(Code* );
  void ExecCycleBlocks();
  void PrintCycle();
};
```

그림 14. Cycle 블록의 자료구조
Fig. 14. Data structure of Cycle blocks.

여기서 커널의 자료구조는 cycle들을 제어하기위한

시간테이블이나 상태(status) 데이터등을 저장하며, 공유데이타 영역에는 제어기의 입력, 출력 및 내부 릴레이와 데이터레지스터들을 저장하여 모든 cycle들이 공유한다.

그림 14는 RTL/MCS의 cycle의 저장과 실행을 위한 자료구조로서 CycleBlock과 Cycle 클래스를 정의하였다. CycleBlock은 각각의 cycle을 저장, 실행하기 위한 클래스이고, Cycle은 CycleBlock형태로 표현된 각 cycle들을 관리 및 실행시키는 역할을한다. CycleBlock의 멤버함수인 Exec는 Cycle이 자기자신을 실행하기 위한 함수이다. 그리고, Cycle의 멤버함수인 ExecCycleBlocks는 Cycle이 저장하고 있는 Cycle 리스트인 CBs 내의 Cycle들을 실행하기 위한 함수이다. 현재 구현된 RTL/MCS의 중간코드 인터프리터는 라운드-로빈(Round Robin) 처리방법을 기본으로 반복실행(cyclic execution) 되도록 구현되었다.

2. 시간분석 및 실행결과

본 논문에서 구현한 실행시간 분석기로 응용 프로그램의 실행시간을 분석한결과 문장단위의 예측 성공비율은 매우 높다. 그러나 프로그램 전체의 실행시간과 실행가능성의 예측은 프로그래머가 제공한 경험적인 시간정보에 크게 의존하는 결과를 보였다. 그림 16는 RTL/MCS로 작성된 프로그램의 스캔시간 분석과정을 보여준다. 또한 그림 15은 cycle내의 문장단위의 실행시간 분석 과정을 나타내었다.

```
Statement Name -> (within 100 ms) in Acycle
Predicted Execution Time:: 59.120 ms
Execution Time :=> 64.354 ms

Statement Name -> (within 120 ms) in Acycle
Predicted Execution Time:: 49.814 ms
Execution Time :=> 58.983 ms

Statement Name -> (every 500 ms) in Bcycle
Predicted Execution Time:: 38.473 ms
Execution Time :=> 43.215 ms
```

그림 15. 문장의 실행시간 분석
Fig. 15. Execution time analysis of statements.

RTL/MCS로 작성된 예제프로그램을 본 연구에서 구현한 목적(target) 시스템에서 실행할 경우 예측한 시간과 비교할 때 약 10% 내외의 오차를 보였다. 이는 Linux가 시분할 시스템으로 구현되었기 때문으로 분석된다. 또한 운영체제의 정확한 부하(load)를 계산할 수 없었기 때문에 오차가 가변적이었다.

```
Cycle Name -> Acycle
Predicted Execution Time for ScanTime
:=> 67.427 ms
Execution Time for Acycle
:=> 71.418 ms

Cycle Name -> Bcycle
Predicted Execution Time for ScanTime
:=> 149.381 ms
Execution Time for Acycle
:=> 165.808 ms
```

그림 16. 프로그램 Scan시간 분석
Fig. 16. Scan time analysis of programs.

3. 레이저 마킹기 제어환경의 구현

실시간 제어언어가 내장된 프로그램형 레이저 마킹기 제어 소프트웨어의 구조는 그림 17과 같다.

① 편집기 : 레이저 마킹기 운영소프트웨어에 내장된 편집기는 문자를 편집할 수 있는 문서편집기와 도형과 그래픽이미지를 편집하기 위한 도형편집기가 있다.

② 마킹기 S/W : 마킹기소프트웨어는 도형, 이미지, 바코드, 텍스트를 마킹하기위한 모듈들을 포함하고 있다. 도형의 마킹은 직선, 다각형, 원같은 도형을 좌표에 따라서 레이저 빔을 연속적으로 제어함으로서 이루어진다. 그러나 이미지나 텍스트 문서 등은 이미지를 스캔하면서 레이저빔을 ON/OFF하면서 마킹을 수행하게된다.

③ 모니터링 S/W : 운영소프트웨어의 모니터링 기능은 크게 마킹상황의 모니터링과 주변제어기기의 모니터링으로 나누어진다. 마킹상황의 모니터링은 현재 레이저빔에의하여 마킹되고있는 도형이나 이미지의 상태를 모니터상에 나타내주는 것이다. 이는 현재 실행되고 있는 제어 언어를 추적하여 그 결과를 화면상에 디스플레이 함으로서 이루어진다. 또한, 주변 제어기기의 모니터링은 레이저 마킹기와 연결된 제어기기들의 센서나 액추에이터의 위치를 감지하여 현재 작동되고있는 마킹시스템의 상황을 화면에 표시해주며, 상황에 따라서는 주변기기들을 직접 제어할 수 있다. 모니터링 소프트웨어에 포함된 그래픽편집기는 마킹기와 주변제어기기들의 구성을 그래픽으로 작성하여 모니터링과 제어를 할 수 있는 그래픽 데이터를 생성해 준다.

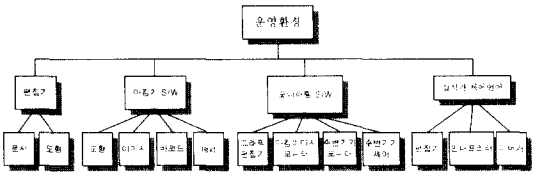


그림 17. 운영소프트웨어의 구조
Fig. 17. Structure of operating software.

그림 18. 프로그램형 레이저마킹기
Fig. 18. Programmable Lazer Marking Machine.

표 2. 실시간언어의 비교
Table 2. Comparsion of Real-Time Languages.

기능	Ada	Modula-2	Occam 2	RTL/MCS
concurrent prog.	yes	yes	yes	yes
timing constraint	clock, delay	delay	clock, delay	시간제한구조
timing analysis	no	no	no	yes
communication	rezdezvous	monitor	channel	hared memory
synchronization	rezdezvous	transfer	condition	rezdezvous
process creation	dynamic	dynamic	dynamic	static
process topology	dynamic	dynamic	dynamic	static
abstraction	+++	++	+	-
exceptions	yes	no	no	yes
proof support	-	-	-	+
large programming	+++	++	+	+
distributed environ.	no	no	yes	no
device handing	공유메모리	공유메모리	메세지전달	공유메모리
efficient implement	-	++	++	+++
application area	general	embedded systems	parallel programming	mechanism control

④ 실시간 제어언어 RTL/MCS의 처리기 소프트웨어는 프로그램 편집기와 중간 코드를 생성하기 위한 컴파일러, 인터프리터, 디버거로 구성되어있다. 프로그램 편집기는 프로그램작성시 언어의 문법을 직접 점검할 수있도록 Syntax-Directed-Editor로 설계하였다. 또한 인터프리터와 연결하여 실행시키면서 프로그램을 디버깅할 수 있도록 트레이스 기능과 스텝핑 기능을 추가하였다. 그림 18은 실시간제어언어가 내장된 프로그램형 레이저마킹기의 시험장면이다. 그림의 왼쪽부분이 레이저마킹의 레이저빔 발진장치이며, 중앙은 제어소프트웨어를 운영하기위한 컴퓨터이고, 오른쪽부분이 냉각장치 및 전원공급 장치이다.

Ⅶ. 결 론

본 연구에서는 수치제어기(NC), 로봇제어기, 프로그램형제어기(PLC)와같은 프로그램가능한 자동화기기의 제어기에 적합한 실시간 메카니즘 제어언어(RTL/MCS)를 설계하고 구현기법을 제안하였다.

또한, 프로그램형 레이저마킹기(programmable lazer marking machine) 제어환경을 구현하여 언어처리기를 내장하여 평가함으로서 실용성을 입증하였다.

RTL/MCS는 실시간 프로그램 작성시 시간제한을 자유롭게 표현할 수 있고, 병행성을 가지고 있어 자동화기기들의 실시간 메카니즘 제어 프로그램을 효율적으로 작성할 수 있다. RTL/MCS의 실행시간분석기는 RTL/MCS의 시간제한 구문들의 실행시간을 분석하여 실시간 프로그램의 컴파일시간에 프로그램의 타당성과 실행가능성을 예측할 수 있으므로 반복적인 하드웨어적 시뮬레이션을 거쳐 프로그램의 정확성을 검증하던 기존의 실시간 제어언어들에 비하여 시간과 비용을 크게 절약할 수 있다.

표 3. 제어언어의 비교
Table 3. Comparison of Control Languages.

기능	LS	LD	SL	FBD	STL	SFC	RTL/MCS
언어수준	low	low	middle	middle	high	high	high
병행성	명령어	명령어	문장	명령어	태스크	태스크	태스크
시간제어	clock	clock	clock	clock	delay	delay	제한구조
시간분석	H/W	H/W	H/W	H/W	H/W	H/W	S/W
언어유형	텍스트	그래픽	텍스트	그래픽	텍스트	그래픽	텍스트
현장적응성	-++	+++	++	++-	++	-	+
프로그래밍	++	-++	++	++-	++	++	-+
예외처리	-	-	-	-	+	+	++
모듈화	-	-	-	-	++	++	++-
효율성	+++	+--	++	++	-	+	++

LS - Logic Symbol LD - Ladder Diagram, SL - Statement List
FBD - Function Block Diagram, STL - Structured Textual Language
SFC - Sequential Function Chart

표 2는 기존의 범용 실시간언어들과 RTL/MCS를 비교평가하였다. 또한 표 3은 현재 가장 많이 사용되고있는 제어언어들과 RTL/MCS의 특징을 비교평가하였다. 따라서 본 연구는 현재 폭넓게 연구되고있는 실시간 시스템의 스케쥴링 기법의 연구와 실시간 프로그래밍언어 및 컴파일러 연구에 도움이될것으로 기대하며, 특히 선진국에 비하여 크게 낙후되어 있는 국내의 메카트로닉스(mechatronics) 기술발전에 도움이 될것으로 기대된다.

참 고 문 헌

[1] Jeffrey D. Ullman, Ravi Sethi and Alfred V. Aho, Compilers: principles, techniques, and tools, Addison-Wesley,1986.

[2] Burns A. and Wellings A.J., Real-time Systems and their Programming Languages, Addison-Wesley, 1990.

[3] Seongsoo Hong, Compiler-Assisted Scheduling for Real-Time Application: A Static Alternative to Low-Level Tuning, Computer Science, University of Maryland, PhD. thesis, 1994.

[4] M. Saksena, Prarmetric Scheduling for Hard Real-Time Systems, Computer Science, University of Maryland,PhD. thesis, 1994.

[5] T.Chung, H.Dietz, Language construction and transformation for hard real-time systems, Second ACM SIGPLAN Workshop on Language, Compilers, ans Tools for Real-Time Systems, 1995.

[6] Roderick Chapman, Static Timing Analysis and Program Proof, Computer Science, University of York, PhD. thesis, 1995.

[7] G. Michel, Programmable Logic Controllers-Architecture and Application, John Wiley & Sons, 1990.

[8] A2NCPU Programming Manual, MITSUBISH ELECTRIC, 1991.

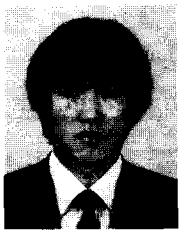
[9] PLC-5/15 Programmable Controller processor manual, ALLEN-BRADLEY,1990.

[10] 원유헌, 프로그래밍 언어론, 1997, 정익사

[11] 백정현, 원유헌, 실시간 제어언어의 설계 및 실행시간 분석 기법, 한국정보과학회 프로그래밍 언어 연구회지 제 10권 1호, 한국정보과학회, 1996, 12

[12] 백정현, 원유헌, 프로그램형 제어기를 위한 실시간 제어언어의 구현 기법, 한국정보과학회 '97 봄 학술발표 논문집제 24권 1호, 1997

저 자 소 개



白 貞 鉉(正會員)

1985년 홍익대학교 전자계산학과(이학사). 1987년 홍익대학교 전자계산학과(이학석사). 1995년 홍익대학교 전자계산학과 박사과정 수료. 1987년 현대중공업(주) 중전기연구소 연구원. 1992년 ~ 현재 중경공업전문대학 전산과 조교수. 주관심분야는 프로그래밍언어, 실시간시스템, 멀티미디어시스템, 컴퓨터교육

元 裕 憲(正會員)

1972년 성균관대학교 수학과졸업(B.S). 1975년 한국과학원 전자계산학과 1회 졸업(M.S). 1975년 한국과학기술연구소 근무. 1985년 고려대학교 대학원 졸업(Ph.D). 1986년 미국 RPI대학 교환 교수. 1976년 ~ 현재 홍익대학교 컴퓨터공학과 교수. 주관심분야는 프로그래밍언어, 병행언어, 실시간언어, 멀티미디어 시스템

論文97-34C-11-5

회로 크기면에서 효율적인 디지털 VCR 용 리드-솔로몬 디코더/인코더 구조

(An Area-Efficient Reed-Solomon Decoder/Encoder Architecture for Digital VCRs)

權 成 勳 *, 朴 東 昱 *, 申 鉉 哲 *, 李 載 先 **, 盧 世 龍 **, 林 龍 澤 **

(Sunghoon Kwon, Dongwook Park, Hyunchul Shin, Jaesun Lee, Seyong Ro, and Yongtaik Kim)

요 약

본 논문에서는 회로 크기면에서 효율적인 구조의 디지털 VCR 용 리드-솔로몬 디코더/인코더를 제안한다. 회로의 크기와 지연 시간을 줄일 수 있는 새로운 구조의 디코더/인코더는 다음과 같은 두 가지 특징을 갖는다. 첫째, 인코딩 (encoding), 변형 신드롬 (modified syndrome), 이레이저 위치 (erasure locator) 다항식을 계산하는 세 가지의 기능들이 하나의 기능 블럭을 공유하도록 구현하여 면적을 효율적으로 사용하도록 하였다. 둘째, 새로운 회로 구조로 변형 유클리드 알고리듬 (modified Euclid's algorithm) 을 구현하였다. 실험 결과 제안한 방법으로 설계된 디코더/인코더는 기존의 방법 [1] 에 기초한 직관적인 설계 결과보다 약 25 % 정도 작은 크기로 구현되었으며 디코딩 지연 시간도 줄어들었다.

Abstract

In this paper, we propose an area-efficient architecture of a Reed-Solomon (RS) decoder/encoder for digital VCRs. The new architecture of the decoder/encoder targeted to reduce the circuit size and decoding latency has the following two features. First, area-efficiency has been significantly improved by sharing a functional block for encoding, modified syndrome computation, and erasure locator polynomial evaluation. Second, modified Euclid's algorithm has been implemented by using a new architecture. Experimental results have showed that the decoder/encoder designed by using the proposed method has been implemented with 25 % smaller size over straight forward implementation based on the conventional method [1] and the decoding latency has been reduced.

I. 서 론

리드-솔로몬 (RS) 코드는 집단 에러 (burst errors) 정정 능력이 우수하기 때문에, 디지털 전송 시스템과 디지털 VCR, CD, DAT와 같은 디지털 비디오/오디오 장치에 널리 사용된다[2, 3, 4, 5, 6]. 낮은 복잡도 (low complexity) 및 high-bit-rate 의 리드-솔로몬 디코더 설계 문제는 많은 연구가 필요한 분야이다[2].

RS 코드는 시간 영역 또는 주파수 영역에서 디코딩과 인코딩이 가능하다[7]. 대부분의 시간 영역 기술에서는 에러/이레이저 (error/erasure) 위치 (locator)

* 正會員, 漢陽大學校 電子工學科
(Dept of Electronics Eng. Hanyang University)
** 正會員, LG 電子 비디오 硏究所
(Video Research Lab., LG Electronics Co.)
接受日字:1997年5月29日, 수정완료일:1997年10月15日

및 값 (evaluator) 다항식을 구할 때, 벌리캠프-메세이 (Berlekamp-Massey) 알고리듬[8], 유클리드 (Euclid) 알고리듬[1, 9, 10], 행렬 (Matrix) 계산[3] 등을 사용하였다.

[1]에서는 VLSI 설계가 용이하도록 시스톨릭 배열 (systolic array) 구조를 이용한 파이프라인 RS 디코더 (pipeline RS decoder)를 제안하였다. 특히, 멀티플렉싱 (multiplexing)과 반복적인 기술 (recursive technique)을 사용함으로써, 변형 유클리드 알고리듬 구현 회로의 크기를 감소시켰다. [10]에서는 세 가지 (outer/inner/subcode)의 코드를 유연하게 디코딩할 수 있는 RS 인코더/디코더를 제안하였다. 하드웨어 크기를 줄이기 위하여 신드롬 계산 블럭과 인코더를 공유한 점이 특징이다. [11]에서는 RS 디코더가 인코더 회로로도 사용될 수 있음을 보여 주었다.

본 논문에서는 [12]의 표준화된 디지털 VCR 의 설계 사양을 만족시키면서 회로 크기면에서 효율적인 RS 디코더/인코더에 대하여 기술한다. 회로의 크기를 줄이기 위하여 세 가지의 서로 다른 기능을 수행하는 기능 블럭을 공유하여 하나의 기능 블럭 내에 구현하였다. 또한 새로운 하드웨어 구조를 개발하여 변형 유클리드 알고리듬을 구현하였다.

제 II 장에서는 새로운 구조의 RS 디코더/인코더의 특징에 대하여 설명한다. 제 III 장에서는 디코더/인코더 내의 세부 블럭과 회로 전체의 동작을 설명한다. 제 IV, V 장에서는 합성할 때 필요한 회로의 크기 비교 결과 및 결론을 제시한다.

II. 새로운 유연한 구조의 RS 디코더/인코더의 특징

디지털 VCR 의 에러 정정 단계에서는 프러덕트 코드 (product code) 형식의 RS 코드가 사용된다.

프러덕트 코드는 내부 코드 (inner 또는 row code)와 외부 코드 (outer 또는 column code) 의 2-D 배열 형태로 구성된다[12]. 디코딩은 내부 디코딩 및 외부 디코딩으로 수행된다[13]. 내부 디코딩은 에러 정정과 검출을 수행한다. 디코딩이 실패한 경우, 2-D 배열 내의 열 (row) 에 있는 모든 심볼들을 이레이져로 표시한다. 외부 디코딩에서는 에러/이레이져를 검출하고 정정한다.

제안한 디코더/인코더는 Galois Fields 인 GF(256) 내에서 세 가지 (n, k) RS 코드 - (85, 77) 내부 코드, (14, 9), (149, 138) 외부 코드- 를 유연하게 디코딩할 수 있다 ([12]의 표준화된 디지털 VCR 설계 사양임). GF (q) 내의 (n, k) RS 코드에서 n 은 코드 길이, k 는 데이터 길이, $2t = (n - k)$에서 $t = \lceil (n - k) / 2 \rceil$ 는 에러 정정 능력을 의미한다. 에러 수를 e, 이레이져 수를 f 라고 하면 이레이져 디코딩은 $2e + f \leq n - k$을 만족하는 e 와 f 갯수까지 정정이 가능하다.

그림 1 은 본 논문에서 제안한 RS 디코더/인코더의 전체 구조를 보여 준다. RS 디코더/인코더는 파이프라인 방식으로 실시간 동작되며, 크게 다섯 블럭 (Syndrome Computation, Encoder and Polynomial Expansion, Modified Euclid's Algorithm, Chien Search, Error Correction and Verification)과 외부 FIFO 메모리로 구성된다. 그림 1 내부에는 각 블럭의 입·출력 신호와 비트 수가 표시되어 있다.

디코딩 방법에 대하여 간단히 설명한다. 세부적인 디코딩 방법은 [1, 2, 6]을 참조한다. 먼저, 수신 데이터로부터 신드롬 다항식 $S(x)$을 계산한 후 이레이져 위치 (erasure locator) $\Lambda(x)$와 변형 신드롬 (modified syndrome) $T(x)$ 다항식을 계산한다. 이

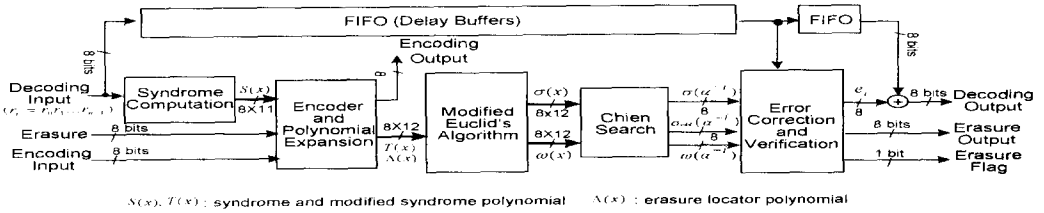


그림 1. 에러와 이레이져를 정정하기 위한 RS 인코더/리코더 구조
Fig. 1. RS encoder/decoder architecture for both error and erasure correction.

들 다항식을 이용하여 변형 유클리드 알고리듬을 수행하면 에러 위치 $\sigma(x)$와 에러 값 $\omega(x)$다항식을 구할 수 있다. 마지막으로 실제 에러 값을 계산한 후 에러를 정정하고 검증하여 출력한다. 일반적으로 인코더는 디코더와 별도의 회로에서 구현된다.

본 논문에서 제안하는 RS 디코더/인코더 구조는 기존의 회로 구조에 비하여 회로의 크기를 줄일 수 있는 다음과 같은 두 가지의 특징을 갖는다.

1. 세 가지의 서로 다른 기능 - 인코더, 이레이져 위치 다항식 $\Lambda(x)$, 변형 신드롬 다항식 $T(x)$ 계산 - 을 하나의 기능 블럭 (Encoder and Polynomial Expansion) 에 공유하여 수행한다. 이들 세 기능들은 서로 다른 시간에 동작하고 회로 구조가 비슷하기 때문에 하나의 하드웨어 블럭을 공유할 수 있다. 두 다항식 계산 블럭을 공유하기 위하여 일반적으로 1 비트의 이레이져 플래그를 사용하는 대신에 8 비트의 이레이져 주소 (address) 를 사용한다.
2. 변형 유클리드 알고리듬 (Modified Euclid's Algorithm) 을 하드웨어로 구현할 때에는 알고리듬에서 소요되는 지연 시간 때문에 회로의 크기가 증가할 수 있다. 따라서, 다항식 계수 계산 과정에서 상위 및 하위 계수를 별도의 회로에서 병렬 계산하여 지연 시간을 1/2로 줄였다.

[1] 에서는 인코더와 별도로 $\Lambda(x)$, $T(x)$를 구하는 두 개의 Polynomial Expansion (Fig. 2) 회로를 사용하였다. 또한, [1] 의 회로 구조로는 실시간 처리에서의 지연 시간 문제 때문에 두 개의 변형 유클리드 알고리듬 처리 회로 (Fig. 3) 가 필요하다. 따라서, 새로운 구조를 이용하여 회로를 설계하면 기존의 방법 [1] 에서보다 회로 크기가 감소되고 디코딩 지연 시간도 줄어든다.

Ⅲ. 새로운 RS 디코더/인코더 회로의 동작

1. RS 디코더/인코더 전체 블럭 동작

그림 1 의 디코더/인코더는 GF(256) 내에서 (n, k) 코드로 표현되는 (85, 77) 내부 코드 및 (14, 9), (149, 138) 외부 코드를 디코딩 한다. 먼저 "Syndrome Computation" 블럭에서는 매 클럭 마다 입력되는 8 비트 r_i를 이용하여 n 클럭 후 신드롬 다항식 $S(x)$을 계산한다. $S(x)$를 계산할 때 "Encoder and Polynomial Expansion" 블럭에서는 8 비트의 입력 이레이져들을 이용하여 이레이져 위치 다항식 $\Lambda(x)$를 계산한다. 본 논문에서는 1 비트의 이레이져 플래그 대신 8 비트의 이레이져 주소를 사용하여 $\Lambda(x)$를 계산한다. 따라서, n 개의 코드에 대한 이레이져가 모두 입력되지 않더라도 $\Lambda(x)$를 미리 계산할 수 있다.

$S(x)$ 다항식의 계수들이 "Encoder and Polynomial Expansion" 블럭 내의 레지스터에 저장된 후에 이레이져 값들이 입력되면, 변형 신드롬 다항식 $T(x) = S(x)\Lambda(x) \mod x^{2t}$ 가 2t 클럭 시간 동안 계산된다. "Modified Euclid's Algorithm" 블럭에서는 $\Lambda(x)$, $T(x)$ 를 이용하여 에러/이레이져 위치 및 값 다항식 $\sigma(x)$, $\omega(x)$를 출력하며, IV 장에서 자세히 설명한다. "Chien Search" 블럭에서는 에러 위치 및 에러 값을 구하기 위하여 사용되는 $\sigma(\alpha^{-i})$, $\sigma_{odd}(\alpha^{-i})$, $\omega(\alpha^{-i})$ 값을 매 클럭마다 계산한다.

마지막으로, "Error Correction and Verification" 블럭에서는 에러/이레이져를 정정하고 그 값이 올바른 값인지를 검증한다. 입력 데이터 r_i와 에러 값 e_i를 더하고, 더한 값을 이용하여 다시 한번 $S(x)$를 구한다. $S(x) = 0$이면 정정된 값이 정확하므로 더한 값을 그대로 출력한다. 그러나, $S(x) \neq 0$일 경우 (정정된 값이 부정확할 경우), 내부 디코딩 과정이면 이레이져 플래그를 출력하고 외부 디코딩 과정이면 정정되지 않은 FIFO 메모리에 저장된 디코더 입력 데이터를 출력한다.

2. RS 디코더/인코더 세부 블럭 동작

RS 디코더/인코더 전체 블럭 중에서 새롭게 제안하는 두 가지 블럭 (인코더와 다항식 확장, 변형 유클리드 알고리듬) 의 하드웨어 구조 및 동작 원리에 대하여 기술한다.

1) 인코더와 다항식 확장 (polynomial expansion) 블럭

그림 2 는 인코더 $\Lambda(x)$, $T(x)$를 하나의 블럭에 구현한 "Encoder and Polynomial Expansion" 블럭의 내부 구조를 보여 준다.

(85, 77) 코드일 때는 ($R_3 \sim R_{11}$), (14, 9) 코드일 때는 ($R_6 \sim R_{11}$) 사이의 레지스터에 저장된 데이터를 사용하여 반복적으로 계산한다. (149, 138) 비디오 코드를 인코딩하는 방법에 대하여 알아보자.

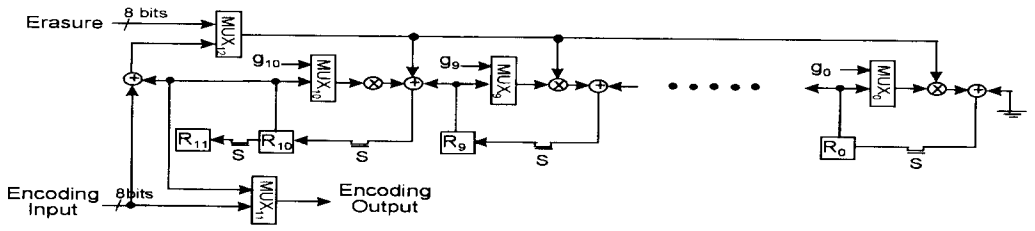


그림 2. 인코더와 다항식 확장 계산 블록

Fig. 2. Encoder and polynomial expansion computation block.

그림 2 에서 스위치 S 는 신호를 통과 (on) 시키거나 또는 차단 (off) 시키고자 할 때 사용되는 스위치이다. 초기에 모든 스위치는 on 이고, 레지스터 값들은 0 으로 세팅된다. MUX_0 ~MUX_{10} 에서는 곱셈기의 계수로써 생성 다항식 (generator polynomial) 의 상수인 g_0 ~ g_{10} 이 선택된다. 'Encoding Input' 에서 입력된 데이터는 초기 k 번의 클럭 동안 MUX_{11} 에서 선택되어 'Encoding Output' 으로 출력된다. 동시에 MUX_{12} 에서도 선택되어 인코딩이 수행된다 (systematic encoding). (k + 1) 번째부터 n 번째 클럭 동안 레지스터 R_0 ~R_{10} 에 저장된 연산 결과인 패리티 심볼들은 MUX_{11} 에서 선택되어 'Encoding Output' 으로 출력됨으로써 인코딩이 완료된다.

이레이져 위치 다항식 $\Lambda(x)$를 구하는 과정을 알아본다. 입력 이레이져들은 MUX_{12} 에서 선택되어 다항식 확장 계산에 사용된다. 초기에 R_0 = 1 이고 나머지 R_1 ~R_{11} 값들은 0 이다. R_i 레지스터에는 i 차수의 계수 값이 저장된다. MUX_0 ~MUX_{10} 에서는 곱셈기의 계수로써 11 개의 레지스터의 출력 값이 선택된다. 8 비트의 이레이져 값이 입력될 때마다 다항식 $\Lambda(x)$의 차수가 1 씩 증가되고, 동시에 연산 회로에서 계산된 다항식의 계수 값들이 레지스터에 저장된다. 이레이져가 없을 경우 스위치 S 는 off 되어 신호가 차단되기 때문에 다항식 확장 계산이 수행되지 않는다.

변형 신드롬 다항식 $T(x)$는 초기에 R_0~R_{10} 레지스터에 $S(x)$의 계수 값들이 저장된다는 것을 제외하고는 $\Lambda(x)$계산 과정과 비슷하다. 이레이져들은 $\Lambda(x)$와 $T(x)$를 계산할 때 한번씩 사용되어야 하므로, "Encoder and Polynomial Expansion" 에 두 번 입력된다.

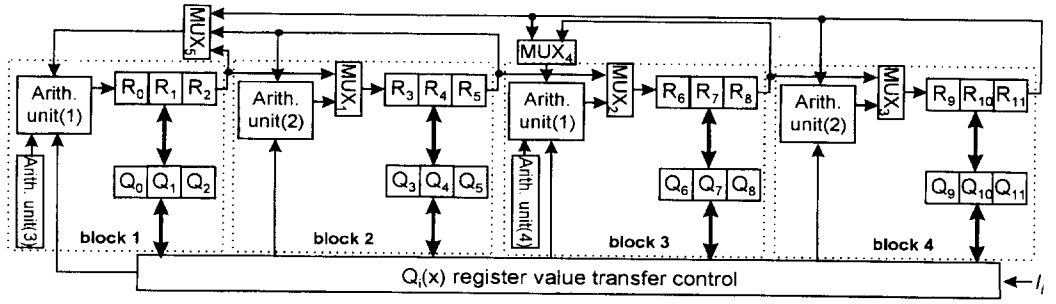
2) 변형 유클리드 (Modified Euclid) 알고리듬 구현 블럭

변형 유클리드 알고리듬은 두 다항식 $\Lambda(x)$, $T(x)$를 사용하여 에러/이레이져 위치와 값 다항식 $\sigma(x)$, $\omega(x)$를 구한다. 그림 3 의 (a), (b) 는 변형 유클리드 알고리듬 수행 블럭을 하드웨어로 구현하였을 때의 내부 구조이다.

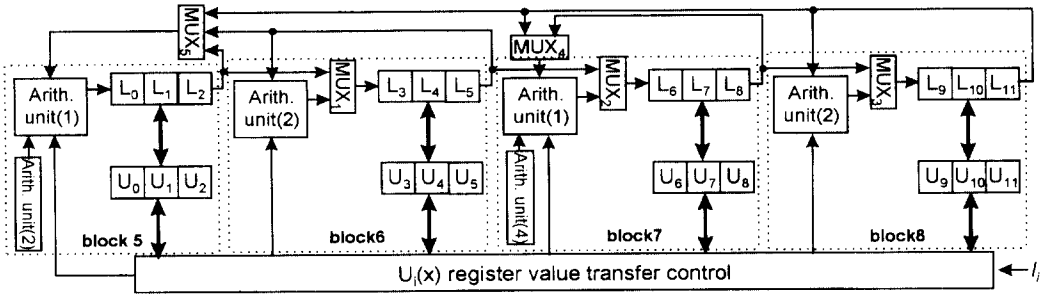
초기에 R_i, Q_i, L_i, U_i 레지스터 내부에 저장되어야 할 4 가지의 다항식과 l_i 및 (a_i, b_i) 값은 다음과 같다 ([1] 의 변형 유클리드 알고리듬 참조).

$R_i(x) = x^{2t}$, $Q_i(x)$, $L_i(x) = 0$, $U_i(x) = \Lambda(x)$

$l_i = \deg(R_i(x)) - \deg(Q_i(x))$, (a_i, b_i) = leading coefficient of $R_i(x), Q_i(x)$



(a)



(b)

그림 3. Modified Euclid 알고리듬 블록(에러/이레이져 디코딩)
(a) $R_i(x)$, $Q_i(x)$ 계산 블록 (b) $L_i(x)$, $U_i(x)$ 계산 블럭

Fig. 3. Modified Euclid's Algorithm Block(error/erasure decoding).
(a) $R_i(x)$, $Q_i(x)$ Computation Block (b) $L_i(x)$, $U_i(x)$ Computation Block

위의 다항식 표현에서 (예를 들면, $R_i(x)$) 첨자 i 는 반복적으로 수행되는 변형 유클리드 알고리듬의 반복 횟수를 나타낸다. 초기에는 i = 0 이며 매 반복마다 i 값이 1 씩 증가한다. 그림 3 의 (a)와 (b)는 $R_i(x), Q_i(x)$ 다항식과 $L_i(x), U_i(x)$ 다항식을 반복적으로 계산하는 블록이다. 유클리드 알고리듬 수행이 완료되면 (a), (b) 블럭에서 $\omega(x) = R_i(x)$와 $\sigma(x) = L_i(x)$다항식이 출력된다.

변형 유클리드 알고리듬 수행 블럭의 동작 원리를 요약하여 기술한다. 초기 i = 0 일 때 초기값들이 $R_i(x)$, $Q_i(x)$, $L_i(x)$, $U_i(x)$레지스터에 저장된다. Stop 조건이 만족되지 않을 경우에는 매 클럭 마다 계산된 출력 값을 레지스터에 저장하고 시프트하는 과정을 반복한다. Stop 조건이 만족되면 정확한 $\omega(x)$와 $\sigma(x)$를 구한 후, "Chien Search" 블럭에서 사용될 초기 값을 계산한다.

기존의 유클리드 알고리듬 [1]을 실제 하드웨어로 구현할 때, stop 조건 (if $d(L_i(x)) > d(R_i(x))$, stop) 이 만족될 때까지 최대한 (2t + 1)*(2t) 개의 클럭 시간이 필요하다. 알고리듬 수행 시간이 코드 길이 n 보다 길어지는 경우, 현재 코드의 계산이 완료되지 않은 상태에서 다음 코드가 입력될 수 있다. 예를 들면, (14, 9) 코드를 디코딩하는 과정에서는 수행 시간이 (2t + 1)*(2t) = 30 이므로 [30/14] = 3 이 되어 변형 유클리드 알고리듬 수행 회로가 최소한 3 개 필요하다.

본 논문에서는 그림 3 의 변형 유클리드 알고리듬 회로를 수행할 때, 다항식의 차수를 상위 (upper) 차수와 하위 (lower) 차수로 나누어 계산한다. 그러므로, 변형 유클리드 알고리듬을 수행하는 데에 걸리는 지연 시간은 ((2t + 1)/2*2t) 이며, 기존의 방법 [1]에 비해 1/2 정도로 감소된다.

그림 3 (a)의 $R_i(x)$, $Q_i(x)$ 계산 블럭에서 (149, 138) 외부 코드가 입력될 때의 회로 동작은 다음과 같다. 초기 $R_i(x)$는 2t = 11 이므로 11 차수의 다항식이다 (다항식 계수는 12 개임에 주의). 다항식의 상위 차수의 계수 6 개를 점선으로 표시된 block 3, 4 의 ($R_6 \sim R_{11}$) 에, 그리고 하위 차수의 계수 6 개를 block 1, 2 의 ($R_0 \sim R_5$) 에 나누어 저장하고 반복적으로 계산한다. 이 때 각각의 MUX들은 다음과 같이 동작한다.

(MUX_1, MUX_3, MUX_4, MUX_5) => (R_2, R_8, R_{11}, R_5) 레지스터 값 선택.

MUX_2 => block 3 의 Arith. unit (1) 의 계산 결과 선택.

(85, 77) 코드인 경우에는 (149, 138) 코드와 같은 방법으로 데이터가 처리되지만, 상위 3 개의 레지스터 (R_9, R_{10}, R_{11}) 값이 사용되지 않기 때문에 0으로 세팅된다. (14, 9) 코드일 때는 block 1, 2 에서 하나의 (14, 9) 코드를 처리함과 동시에 block 3, 4 에서도 다른 하나의 (14, 9) 코드를 처리한다. 이와 같이 데이터를 처리하는 이유는 14 클럭 시간 내에 (14, 9) 코드를 처리할 수 없기 때문이다.

3. RS 디코더/인코더 회로의 타이밍 설명

먼저, n 클럭 동안 신드롬 다항식이 계산되고, 동시에 2t 클럭 동안 이레이져 위치 다항식도 계산된다. 2t

클럭 동안 변형 신드롬 다항식이 계산된 후, ([(2t + 1) / 2] + 1) * 2t + 12 클럭 동안 유클리드 알고리듬이 수행된다. [(2t + 1) / 2] 는 다항식 계수들을 반으로 나누어 계산하기 때문에 필요한 클럭 시간이고, 두 다항식을 교환할 때 1 클럭, Chien search 알고리듬의 초기값을 계산할 때에 12 클럭 이 필요하다. Chien search 와 n + 3 클럭 시간이 소요되는 에러 정정과 검증 과정이 완료되면 하나의 코드에 대한 디코딩이 완료된다.

Ⅳ. 실험 결과

본 논문에서 제안한 새로운 RS 디코더/인코더 구조와 [1] 논문에 기초한 회로 구조를 이용하여 실제로 회로를 구현할 때에 필요한 회로 크기를 비교·실험하였다. 비디오/오디오 데이터에 랜덤 (random) 하게 에러/이레이져를 발생시켜 첨가한 후, 에러/이레이져 들이 100 % 정정됨을 소프트웨어 시뮬레이션으로 확인하였다.

표 1. 리드-솔로몬 디코더/인코더의 회로 크기 비교

Table 1. Comparisons on circuit sizes of RS decoder/encoder.

Hardware Blocks \ Design Methods	Straight Forward Implementation ([1])	Implementation Results (Proposed Architecture)
Encoding	2,000	0
Syndrome Computation	2,000	2,000 (1,750)
Encoder and Polynomial Expansion	10,000	5,000 (6046)
Modified Euclid's Algorithm	22,000	17,000 (18,885)
Chien Search	4,000	4,000 (4,760)
Error Correction and Verification	9,000	9,000 (9,726)
Number of Total Gates (%)	49,000 (100%)	37,000(41,167) 75%)

(주의 : 제안한 방법에서 괄호 안의 게이트 수는 'Synopsys' 툴로 합성한 결과임)

표 1은 서로 다른 두 가지 설계 방법으로 구현할 때에 필요한 회로 크기를 추정 및 합성 (괄호 안의 숫자는 지연 시간을 고려하여 실제 설계 툴을 이용하여 합성한 결과임) 한 결과이다. 새로운 구조의 디코더/인코더는 VHDL로 기술되었으며 'Synopsys' 사에서 제공하는 'class.db' 라이브러리를 사용하여 합성되었다. 논문 [1] 에서 제안한 회로 구조와 비교한 이유는 몇몇 논문들 ([3, 10]) 과는 다르게 회로의 내부 구조를 알 수 있어서 게이트 수 추정이 가능하기 때문이다. 회로 크기를 추정한 이유는 [1] 방법에 기초한 디지털 VCR 용 회로 설계 결과가 발표되지 않았기 때문이다.

표 2는 디코딩 지연 시간 동안 필요한FIFO 메모리 크기를 비교한 결과이다.

표 2. FIFO 메모리 크기 비교

Table 2. Comparisons on FIFO memory sizes.

Design Methods	Straight Forward Implementation ([1])	Implementation Results (Proposed Architecture)
FIFO Memory Size	8 bits * 456	8 bits * 401

하나의 게이트는 대략 2 입력 NAND 게이트의 크기이다. 전체 회로에서 제어 부분의 회로 크기를 정확히 계산하는 것이 어렵기 때문에, 회로 구현에 필요한 최소한의 게이트 수를 표 1 에 추정하여 기술하였다. 따라서, 실제 회로 합성 결과는 추정 결과에 비해 약 10 % 정도 증가할 수 있다.

본 논문에서 제안한 구조로 RS 디코더/인코더를 설계하면 37,000 게이트 정도 필요하고, [1] 구조에 기초하여 직관적인 방법으로 설계하면 49,000 게이트 정도 필요하다. 결과적으로, 약 12,000 게이트 정도로 회로 크기가 줄어들고 FIFO 메모리의 크기도 감소됨을 알 수 있다.

Ⅴ. 결 론

본 논문에서는 회로 크기 면에서 효율적인 구조의 RS 디코더/인코더를 제안하였다. 기존의 디코더 회로 구조에 비해 제안한 구조는 다음과 같은 두 가지 특징을 갖는다. 첫째, 공유 가능한 하드웨어 블럭을 최대한 공유하여 회로 크기를 줄였다. 둘째, 새로운 회로 구조를 이용하여 변형 유클리드 알고리듬 구현 블럭의 크기를 감소시켰다. RS 디코더/인코더 회로의 크기를 비교 실험한 결과, 회로 설계에 필요한 게이트 수는 기존 방법에 기초하여 직관적으로 설계한 경우보다 25 % 정도 줄어들고 디코딩 지연 시간도 감소되었다. 추

후에, 더욱 최적화된 회로 구조를 사용하고 동시에 일반적인 (n, k) RS 코드 처리가 가능한 디코더/인코더가 하드웨어로 구현 될 수 있을 것으로 판단된다.

참 고 문 헌

[1] H. M. Shao and I. S. Reed, "On the VLSI Design of a Pipeline Reed-Solomon Decoder Using Systolic Arrays," *IEEE Trans. on Computers*, vol. 37, no. 10, pp. 1273-1280, Oct. 1988.

[2] S. B. Wicker and V. K. Bhargava, *Reed-Solomon Codes and Their Applications*, New York, IEEE Press, pp. 1-107, 1994.

[3] T. Iwaki, T. Tanaka, E. Yamada, T. Okuda, and T. Sasada, "Architecture of A High Speed Reed-Solomon Decoder," *IEEE Trans. on Consumer Electronics*, vol. 40, no. 1, pp. 75-81, Feb. 1994.

[4] K. Sato, M. Hattori, N. Ohya, M. Sasano, and N. Shirota, "ARSDES: An Automated Reed-Solomon Decoder and Encoder Synthesis System," *CICC*, pp. 28.2.1-28.2.4, 1995.

[5] M. H. Lee, S. B. Choi, and J. S. Chang, "A High Speed Reed-Solomon Decoder," *IEEE Trans. on Consumer Electronics*, vol. 41, no. 4, pp. 1142-1149, Nov. 1995.

[6] S. B. Wicker, *Error Control Systems for Digital Communication and Storage*, Prentice Hall, pp. 175-237, 1995.

[7] Y. Jeong and W. Burleson, "High-Level Estimation of High-Performance Architectures for Reed-Solomon Decoding," *ISCAS*, vol. 1, pp. 720-723, May, 1995.

[8] J. M. Hsu and C. L. Wang, "An Area-Efficient VLSI Architecture for Decoding of Reed-Solomon Codes," *ICASSP*, vol. 6, pp. 3291-3294, May 1996.

[9] H. M. Shao, W. Hills, T. K. Truong, I. S. Hsu, both of Pasadena, L. J. Deutsch, Sepulveda, all of Calif., "Architecture for Time or Transform Domain Decoding of Reed-Solomon Codes," *U. S. Patent 4,868,828*, Sep. 19, 1989.

[10] G. Y. Lee, B. H. Kwuan, S. W. Lee, J. W. Jung, S. H. Nam, Y. S. Chun, D. I. Han, K. S. Park, Y. D. Choi, D. I. Cho, J. K. Lee, "A VLSI Design of RS CODEC for Digital Data Recorder," *ASIC Design Workshop*, pp. 115-124, 1996.

[11] C. C. Hsu, I. S. Reed, and T. K. Truong, "Use of the RS Decoder as an RS Encoder for Two-Way Digital Communications and Storage Systems," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 4. no. 1, pp. 91-92, Feb. 1994.

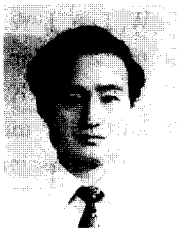
[12] *Specifications of Digital VCR for Consumer-Use*, SD; Standard Definition, NTSC, PAL, SECOM, HD-Digital VCR Conference, DRAFT IEC document, Confidential, June 1994.

[13] S. H. Kim and S. W. Kim, "An Error-Control Coding Scheme for Multispeed Play of Digital VCR", *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 5. no. 3. pp. 243-247, June 1995.

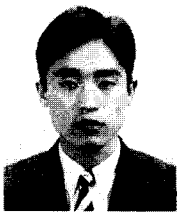
저 자 소 개

權 成 勳(正會員) 第 32卷 A編 第 1號 參照
현재 한양대학교 대학원 박사과정

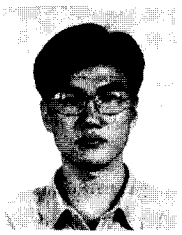
申 鉉 哲(正會員) 第 32卷 A編 第 1號 參照
현재 한양대학교 대학원 박사과정



盧 世 龍(正會員)
1983년 한양대학교 전자공학과. 1989년 와세다 대학원 전자공학과. 현재 LG전자 Multimedia 사업본부 평택연구소 책임연구원. 1990년 Color Video Printer 개발. 1993년 Double-Deck VCR개발, Video-CD Player개발. 현재 ¼" DVC Camcorder 개발. 주관심분야는 DTV, DBS, Internet application 등 Digital 상 압축/신장 기술, 채널코딩 기술



朴 東 昱(正會員)
1974년 1월 18일생. 1996년 2월 한양대학교 전자공학과 졸업. 1997년 현재 한양대학교 전자공학과 석사과정. 주관심분야는 디지털 회로 설계 및 CAD



李 載 先(正會員)
1991년 한양대 전자공학과. 1993년 한양대 전자공학과 대학원. 1993년 ~ 현재 LG전자 Multi-media 사업본부 평택연구소 주임연구원. 1994년 ¼" DVC logical confirm. 1996년 DVC용 RS En/Dec Engine G/A 개발. 1997년 ~ 현재 MPEGII decoder 차기 Version 개발. 주관심분야는 Error Control Coding, DTV, DBS, 채널코딩

林 龍 澤(正會員)
1976년 한양대학교 전자공학과. 1979년 금성사 중앙 연구소. 1995년 ~ 현재 LG전자 Multi-midia 사업본부 평택연구소장(이사). 1980년 국내최초 전자식 VCR 개발. 1985년 8mm 캠코더 개발. 1995년 ~ 현재 DBS 수신기/PCTV 개발. 주관심분야는 DBS, Internet Application, PCTV 등 차세대 Digital Multimedia 분야

論文97-34C-11-6

IEEE 반올림과 덧셈을 동시에 수행하는 부동 소수점 곱셈 연산기 설계

(Design of the Floating Point Multiplier Performing IEEE Rounding and Addition in Parallel)

朴 祐 贊 *, 鄭 喆 浩 *, 梁 眞 冀 *, 韓 鐸 敦 *

(Woo-Chan Park, Cheol-ho Jeong, Jin-Ki Yang, and Tack-Don Han)

요 약

일반적인 부동 소수점 곱셈 연산에서 분수부의 처리는 곱셈, 캐리와 합의 덧셈, 반올림, 정규화로 이루어지는 부류가 있고 곱셈, 캐리와 합의 덧셈, 정규화, 반올림으로 이루어지는 것이 있다. 그런데, 반올림 단계에서는 별도의 고속 가산기를 필요로하여 많은 처리 시간과 칩 면적을 차지한다. 본 논문에서는 부동 소수점 곱셈 연산기에서 덧셈 연산시 사용되는 고속 덧셈기인 올림수 선택 덧셈기를 사용하고 부동 소수점 곱셈의 특성을 분석함으로써 덧셈과 IEEE에서 지정한 반올림을 동시에 수행하는 부동 소수점 곱셈 연산기를 제시하였다. 제안된 부동 소수점 곱셈 연산기는 반올림 처리를 위한 별도의 시간을 요하지 않고, 반올림 단계를 위한 가산기나 증가기를 필요로 하지 않는다. 따라서, 제안하는 부동 소수점 곱셈 연산기는 성능면이나 차지 면적 면에서 모두 효율적이다.

Abstract

In general, processing flow of the conventional floating-point multiplication consists of either multiplication, addition, normalization, and rounding stages, or multiplication, addition, rounding, and normalization stages. Then, the rounding stage of the conventional floating-point multiplier requires a high speed adder for increment, increasing the overall execution time and occupying a large amount of chip area. A floating-point multiplier performing addition and IEEE rounding in parallel is designed by using the carry select adder used in the addition stage and optimizing the operational flow based on the characteristics of floating point multiplication operation. A hardware model for the floating point multiplier is proposed and its operational model is algebraically analyzed in this paper. The proposed floating point multiplier does not require any additional execution time nor any high speed adder for rounding operation. Thus, performance improvement and cost-effective design can be achieved by this suggested approach.

* 正會員, 延世大學校 컴퓨터科學科 竝列處理시스템硏究室

(Parallel Processing Systems Laboratory, Dept of Computer Science Yonsei University)

※ 본 연구는 통상산업부와 과기처에서 지원하는 IDEC 주관 사업인 ASIC 기반기술과제의 지원으로 수행되었음.

接受日字:1997年5月16日, 수정완료일:1997年10月15日

Ⅰ. 서 론

부동 소수점 연산기(Floating Point Unit)는 그래픽 가속기(Graphic Accelerator), 디지탈 신호 처리기(Digital Signal Processor) 및 고성능을 요구하는 컴퓨터에 필수적으로 사용되고 있다. 부동 소수점 연산기(Floating Point Unit)는 반도체 분야의 발전으로 칩(Chip)의 집적도가 증가함에 따라서 중앙처리 장치(Central Processing Unit)와 함께 한 칩에 내장할

수 있게 됨으로써, 주 연산기의 중요한 요소로 등장하고 있다[1,2,3,4,5]. 중앙처리 장치에 내장되는 경우 차지하는 면적으로 인하여 보통 덧셈/뺄셈, 곱셈 등의 기본적인 연산기만 내장된다. 따라서, 부동 소수점 곱셈 연산기는 전체 부동 소수점 연산에 큰 영향을 미친다.

일반적인 부동 소수점 곱셈 연산에서 분수부의 처리는 곱셈(Multiplication), 곱셈 과정에서 생성된 캐리(carry)와 합(sum)의 덧셈(Addition), 정규화(Normalization), 반올림(Rounding)의 순서로 이루어지는 것이 있고[6,7], 곱셈, 덧셈, 반올림, 정규화의 4 단계의 순서로 이루어지는 것이 있다[8,9,10]. 첫번째와 두번째의 경우 반올림 처리를 위해 별도의 고속 증가기나 덧셈기를 사용한다. 또한, 첫번째의 경우 반올림시 오버플로우(overflow)로 인하여 발생하는 재정규화(Renormalization)를 위한 별도의 하드웨어가 필요하고, 두번째의 경우 정규화 전에 반올림 위치에서 반올림을 수행하는 별도의 하드웨어가 필요하다. 따라서, 차지하는 면적이 커지고 처리 시간이 길어진다.

한편, 부동 소수점 연산기에서 덧셈과 반올림을 동시에 수행하는 연구가 [2,11,12]에 발표되었다. [11,12]에서는 덧셈과 반올림을 병렬적으로 수행할수 있는 하드웨어 모델을 제시 하였다. [11]에서는 IEEE 표준안에서 요구하는 4가지 반올림 방법중 round-to-neatest 방법만 지원을 한다. [2]에서는 [11]에서 제시한 하드웨어 모델을 수정하여 IEEE 표준안에 준하는 4가지 반올림 방법을 모두 지원하며, 반올림과 덧셈을 동시에 하는 부동 소수점 곱셈 연산기를 구현하였다. 그러나, [2]에서는 처리 알고리즘이 복잡하여 덧셈과 반올림을 동시에 하기 위해서 두 단계의 파이프라인을 필요로 하며 여러개의 부가적인 로직을 필요로 한다. [12]에서는 4가지 반올림 방법을 모두 지원하며, 반올림과 덧셈을 동시에 수행하기 위한 요소들을 선정하여 반올림과 덧셈을 동시에 수행할수 있는 일반적인 하드웨어 모델을 제시하였으나, 구체적인 게이트레벨(gate level)의 구현에 대한 언급은 없다.

본 논문에서는 부동 소수점 곱셈 연산기에서 덧셈과 반올림을 동시에 수행할 때의 특성을 수식 적으로 분석하고, 이를 바탕으로 [12]에서 제시한 하드웨어 모델을 수정하여 최적화된 하드웨어 모델을 제시하고 이를 게이트레벨로 구현하였다. 본 부동 소수점 곱셈 연산기는 반올림이 덧셈과 병렬로 수행되기 때문에 기존의 부동 소수점 곱셈 연산시 반올림 단계에서 필요한 덧셈기를 위한 별도의 하드웨어가 필요하지 않는다. 본 논문에서 제시한 부동 소수점 곱셈 연산기는 덧셈과 반올림을 한개의 파이프라인에서 동시에 수행할 수 있다. 따라서, 처리 속도가 빨라지고 소요되는 게이트(gate)의 수도 줄였다.

IEEE 표준안[13]이 발표된 이후의 거의 모든 부동 소수점 연산기는 IEEE 표준안을 따르기 때문에, 본 논문에서 반올림으로는 IEEE 표준안인 4가지 모드(mode)를 모두 지원하도록 하였다. 부동 소수점 곱셈 연산은 분수부 처리와 지수부 처리로 나뉜다. 지수부의 처리는 간단한 로직으로 구성이 되있어서 본 논문에서는 분수부의 처리만을 주로 다룬다. 또한, 곱셈과 정규화는 일반적인 부동 소수점 곱셈 연산기[8,9,10]와 동일함으로 본 논문에서는 다루지 않는다.

2장에서는 IEEE 표준안을 따르는 부동 소수점 분수부의 표현과 반올림 방법에 대해 살펴보고, 덧셈 연산과 반올림을 동시에 수행할때 덧셈 연산과 반올림과의 관계를 살펴본다. 3장에서는 덧셈 연산과 반올림을 동시에 수행할수 있는 하드웨어 모델과 각각의 반올림 모드에 대한 효율적인 구현에 대해 살펴본다.

Ⅱ. IEEE 반올림과 덧셈을 동시에 수행할 때의 특성

이번 장에서는 IEEE 반올림 방법에 대해서 논하고, 부동 소수점 곱셈 연산기에 IEEE 반올림 방법이 적용되는 것을 수식적으로 분석한다. 또한, 부동 소수점 곱셈 처리 과정에서 곱셈, 덧셈, 반올림과의 관계를 살펴보고, 덧셈과 반올림을 병렬로 수행할 때 어떤 특성이 있는지를 살펴보겠다. 이번 장에서 살펴본 결과는 다음 장의 하드웨어 모델을 디자인할때 사용된다.

1. IEEE 반올림

IEEE 표준안에서는 부동 소수점 수를 표현하는데 있어서 32 비트인 단정밀도(single precision) 형식과 64 비트인 배정밀도(double precision) 형식이 있다. 단정밀도의 형식은 1 비트의 부호 비트, 8 비트의 지수부, 23 비트의 분수부로 되어 있다. 배정밀도의 형식은 1 비트의 부호 비트, 11 비트의 지수부, 52 비트의 분수부로 되어 있다. IEEE 표준안에 따르는 정규화된 피연산자 A는 $(-1)^s \times 1.f \times 2^{(e-bias)}$으로 표현

된다. 여기서, s는 부호비트의 값이고 f는 분수부의 값이고, e는 지수부의 값에 해당한다. s는 분수부에 대한 부호 비트이며, f는 절대치 형태의 분수부이며, e는 바이어스(bias)형태의 지수부이다. 분수부의 정규화된 형태는 MSB(Most Significant bit)가 1인 상태이며 부동 소수점 표현에서는 MSB가 생략된다. 이 MSB를 히든 비트(hidden bit)라고 한다. 본 논문에서의 분수부는 히든 비트를 포함한 분수부를 말한다.

$$F = \overbrace{f_{2n-1}f_{2n-2}\cdots f_{n-1}}^{F_I} \cdot \overbrace{f_{n-2}\cdots f_1 f_0}^{F_T}$$

그림 1. F_I과 F_T 의 정의
Fig. 1. The definitions of F_I and F_T.

두 부동 소수점 수의 분수부인 n 비트의 A, B를 곱하면 2n 비트의 합(sum)인 $S = s_{2n-1}s_{2n-2}\ldots s_0$와 캐리(carry)인 $C = c_{2n-1}c_{2n-2}\ldots c_0$ 가 생성된다[14]. 그 후, C와 S를 더한 결과인 2n 비트의 F가 생성된다. 그런데, 2n 비트인 F는 부동 소수점의 분수부인 상위 n+1 비트와 버려지는 부분인 하위 n-1 비트로 나눌 수 있다. 여기서, F의 하위 n-1 비트는 반올림의 판단 근거가 된다. 그리고, 이 정보는 라운드 비트(R : Round bit)와 스티키 비트(Sy : Sticky bit)로 표현될 수 있으며, 이것을 가지고 IEEE에서 지정한 반올림을 수행할 수 있다[11,12]. R은 F에서 하위 n-1 비트의 MSB이고, Sy는 F에서 하위 n-2 비트를 전부 논리합 연산을 한 값이다. 앞으로 나올 수식들을 간단히 하기 위해 본 논문에서는 2n 비트의 변수에서 상위 n+1 비트를 정수로 보고, R과 Sy 비트 위치인 하위 n-1 비트를 소수로 본다. 그리고, 정수부분은 밑첨자 I가 붙고, 소수부분은 밑첨자 T가 붙는다. 그림1에서 보는 바와 같이 F에서 상위 n+1 비트인 $f_{2n-1}f_{2n-2}\ldots f_{n-1}$를 정수부분이라 보고, 그 하위 비트인 $f_{n-2}\ldots f_0$을 소수로 본다. 그리고, F에서 정수부분을 F_I 라 하고, 소수부분을 F_T라 한다. 따라서, F는 다음 식으로 나타낼 수 있다.

$$\begin{aligned} F &= A\times B = C+S \\ &= (c_{2n-1}c_{2n-2}\cdots c_{n-1}.c_{n-2}\cdots c_0)+(s_{2n-1}s_{2n-2}\cdots s_{n-1}.s_{n-2}\cdots s_0) \\ &= f_{2n-1}f_{2n-2}\cdots f_{n-1}.f_{n-2}\cdots f_0 \end{aligned}$$

반올림을 위해서는 R와 Sy를 생성해야 되는데 이는 $f_{n-2}\cdots f_0$에 해당한다. 따라서, 위의 식은 다음 식(1)로 나타낼 수 있다.

$$F = f_{2n-1}f_{2n-2}\cdots f_{n-1}.RSy \quad (1)$$

IEEE 표준안에는 round-to-nearest, round-to-zero, round-to-positive-infinity, round-to-negative-infinity인 4가지 반올림 방식을 규정하고 있다. IEEE 4가지 반올림 방식을 부호에 따라 나누면 round-to-nearest, round-to-zero, round-to-infinity로 나눌 수 있다. 다음은 부동 소수점 곱셈 연산에서 곱셈, 덧셈, 정규화를 거친후에 생성된 분수부의 LSB(Least Significand Bit), R, Sy 비트에 대한 round-to-nearest, round-to-zero, round-to-infinity의 결과이다. 여기서 return 0는 반올림의 결과 버림을 뜻하며, return 1는 반올림의 결과 올림을 뜻한다.

Algorithm 1 : $Rounding_{Nearest}(LSB, R, Sy)$

```
if (R=0) return 0
else if (Sy=1) return 1
     else if (LSB=0) return 0
          else return 1
```

Algorithm 2 : $Rounding_{Zero}(LSB, R, Sy)$

```
return 0
```

Algorithm 3 : $Rounding_{Infinity}(LSB, R, Sy)$

```
if ((R=1) or (Sy=1)) return 1
else return 0
```

2. 곱셈, 덧셈, 반올림과의 관계 분석

식 (1)에서 $f_{2n-1}=0$이면 정규화 과정에서 쉬프트가 필요하지 않으며 이때를 NS라고 하고, $f_{2n-1}=1$이면 정규화 과정에서 1 비트 오른쪽 쉬프트가 필요하며 이때를 RS라고 하겠다. X는 don't care condition이다. C_k^{in}는 하위 k-1 비트에서 k 비트로의 올림수이다. overflow(Z)는 Z의 연산의 결과 오버플로우(overflow)가 생기면 1을 return하고 아니면 0을 return한다. "∧"는 논리곱 연산이고, "∨"는 논리합 연산이고, "⊕"은 exclusive-or 연산이다. 식 (1)는 [11,12]에 의해 다음 식 (2)로 나타낼수 있다.

$$\begin{aligned} F &= C+S \\ &= (c_{2n-1}c_{2n-2}\cdots c_{n-1}.c_{n-2})+(s_{2n-1}s_{2n-2}\cdots s_{n-1}.s_{n-2}) \\ &\quad +(0.0c_{n-3}\cdots c_0)+(0.0s_{n-3}\cdots s_0) \\ &= (c_{2n-1}c_{2n-2}\cdots c_{n-1}.c_{n-2})+(s_{2n-1}s_{2n-2}\cdots s_{n-1}.s_{n-2})+0.C_{n-2}^{in}+0.0Sy \\ C_{n-2}^{in} &= overflow((c_{n-3}\cdots c_0)+(s_{n-3}\cdots s_0)) \end{aligned} \quad (2)$$

여기서, Sy는 A와 B의 trailing-zero의 합이 n-2보다 크면 0이고, n-2보다 작으면 1이다[11]. 그리고, Sy는 A와 B를 곱하여 C와 S를 생성할때와 병렬적으로 구할수 있다.

$D_I=1\,C_I+S_I=d_nd_{n-1}\cdots d_0$이고 $D_T=C_T+S_T$라고 하면 식 (2)는 다음과 같이 나타낼수 있다.

$$\begin{aligned} F &= (c_{2n-1}c_{2n-2}\cdots c_{n-1},c_{n-2})+(s_{2n-1}s_{2n-2}\cdots s_{n-1},s_{n-2})+0,C^{in}_{n-2}+0,0Sy \\ &= C_I+S_I+0,RSy+C^{in}_{n-1} \\ &= D_I+C^{in}_{n-1}+0,RSy \\ C^{in}_{n-1} &= overflow(C^{in}_{n-2}+s_{n-2}+c_{n-2}) \\ R &= C^{in}_{n-2}\oplus s_{n-2}\oplus c_{n-2} \end{aligned} \tag{3}$$

따라서, 식 (3)에 의해 $D_T=C_T+S_T$는 다음 식 (4)과 같다.

$$D_I=C_T+S_T=C^{in}_{n-1},RSy \tag{4}$$

또한, 식 (3)에 의해 F_I는 다음과 같다.

$$F_I=C_I+S_I+C^{in}_{n-1}=D_I+C^{in}_{n-1} \tag{5}$$

정규화 과정은 NS와 RS로 나눌수 있다. NS후에 F의 값을 P^{NS}라고 하고 RS후에 F의 값을 P^{RS}라고 하면 P^{NS}는 다음 식 (6)로 나타낼 수 있다.

$$\begin{aligned} P^{NS}_I &= f_{2n-2}\cdots f_{n-1} \\ P^{NS}_T &= RSy \end{aligned} \tag{6}$$

P^{RS}는 다음 식 (7)로 나타낼수 있다.

$$\begin{aligned} P^{RS}_I &= f_{2n-1}\cdots f_n \\ P^{NS}_T &= f_{n-1}(R\vee Sy) \end{aligned} \tag{7}$$

Q는 F에 대해서 정규화 전에 반올림 한 값 이라면, NS시 Q의 값은 Q^{NS}이고 RS시 Q의 값은 Q^{RS}이다. 또한, $Rounding_{mode}(f_{n-1},R,Sy)$는 해당 반올림 모드에서 반올림의 결과를 나타낸 것이고, mode는 세가지 반올림 방법에서 어떤 한가지를 의미한다. 이때, NS시 Q^{NS}는 식 (5)과 (6)를 적용하면 다음 식 (8)와 같다.

$$\begin{aligned} Q^{NS} &= P^{NS}_I+Rounding_{mode}(f_{n-1},R,Sy) \\ &= F_I+Rounding_{mode}(f_{n-1},R,Sy) \\ &= D_I+C^{in}_{n-1}+Rounding_{mode}(f_{n-1},R,Sy) \end{aligned} \tag{8}$$

또한, RS시 Q^{RS}는 식 (5)과 (7)을 적용하면 다음과 같다.

$$\begin{aligned} Q^{RS} &= (P^{RS}_I+Rounding_{mode}(f_n,f_{n-1},R\vee Sy))\times 2 \\ &= ((f_{2n-1}f_{2n-2}\cdots f_n)+Rounding_{mode}(f_n,f_{n-1},R\vee Sy))\times 2 \\ &= (f_{2n-1}f_{2n-2}\cdots f_nf_{n-1})+2\times Rounding_{mode}(f_n,f_{n-1},R\vee Sy) \\ &= F_I+2\times Rounding_{mode}(f_n,f_{n-1},R\vee Sy) \\ &= D_I+C^{in}_{n-1}+2\times Rounding_{mode}(f_n,f_{n-1},R\vee Sy) \end{aligned}$$

그런데, RS시 $f_{n-1}=0$ 이고 $Round_{mode}(f_n,f_{n-1},R\vee Sy)=1$ 이면, $Q^{RS}=(f_{2n-1}f_{2n-2}\cdots f_{n-1})+2$ 이다. 그러나, $f_{n-1}=1$ 이고 $Round_{mode}(f_n,f_{n-1},R\vee Sy)=1$ 이면, $Q^{RS}=(f_{2n-1}f_{2n-2}\cdots f_{n-1})+1$ 혹은 $Q^{RS}=(f_{2n-1}f_{2n-2}\cdots f_{n-1})+2$ 이다. 왜냐하면, RS시에는 Q^{RS}의 상위 n 비트가 부동 소수점 곱셈 연산기의 실질적인 결과 값이 되는데, $(f_{2n-1}f_{2n-2}\cdots f_{n-1})+1$과 $(f_{2n-1}f_{2n-2}\cdots f_{n-1})+2$결과의 상위 n 비트는 동일하기 때문이다. 따라서, 위의 식은 f_{n-1}의 값에 따라 다음의 두 가지로 나눌수 있다.

$$\begin{aligned} &\text{If } f_{n-1}=0 \\ &\quad Q^{RS} = (f_{2n-1}f_{2n-2}\cdots f_nf_{n-1})+2\times Rounding_{mode}(f_n,f_{n-1},R\vee Sy) \\ &\quad = F_I+2\times Rounding_{mode}(f_n,f_{n-1},R\vee Sy) \\ &\quad = D_I+C^{in}_{n-1}+2\times Rounding_{mode}(f_n,f_{n-1},R\vee Sy) \\ &\text{If } f_{n-1}=1 \\ &\quad Q^{RS} = (f_{2n-1}f_{2n-2}\cdots f_nf_{n-1})+Rounding_{mode}(f_n,f_{n-1},R\vee Sy) \\ &\quad = F_I+Rounding_{mode}(f_n,f_{n-1},R\vee Sy) \\ &\quad = D_I+C^{in}_{n-1}+Rounding_{mode}(f_n,f_{n-1},R\vee Sy) \end{aligned}$$

그런데, round-to-nearest 모드에서는 반올림의 결과가 올림이 될려면 f_{n-1}는 1이 되어야 된다. 한편, round-to-infinity 모드에서는 $R\vee Sy=1$ 이면 f_{n-1}의 값에 상관없이 반올림의 결과가 올림이다. 따라서, 위의 식은 식 (9)와 같이 나타낼수 있다.

$$\begin{aligned} &\text{If } f_{n-1}=1 \\ &\quad Q^{RS} = F_I+Rounding_{mode}(f_n,f_{n-1},R\vee Sy) \\ &\quad = D_I+C^{in}_{n-1}+Rounding_{mode}(f_n,f_{n-1},R\vee Sy) \\ &\text{If } f_{n-1}=X \\ &\quad Q^{RS} = F_I+2\times Rounding_{mode}(f_n,f_{n-1},R\vee Sy) \\ &\quad = D_I+C^{in}_{n-1}+2\times Rounding_{mode}(f_n,f_{n-1},R\vee Sy) \end{aligned} \tag{9}$$

위의 식 (8)과 (9)을 살펴보면 반올림과 덧셈을 동시에 수행하기 위해 D_I, D_I+1, D_I+2, D_I+3 의 결과값을 적절히 생성해내야 된다는 것을 알수 있다. 다음 장에서는 이런 결과값을 생성해 낼 수 있는 하드웨어 모델을 제안하고 이에 대한 게이트 레벨(gate level)의 구현에 대하여 논한다.

Ⅲ. IEEE 반올림과 덧셈을 동시에 수행하는하드웨어 모델 제시 및 구현

이번 장에서는 덧셈과 반올림을 병렬적으로 수행할 수 있는 하드웨어 모델을 살펴본다. 그리고, 이것을 이용하여 round-to-nearest, round-to-zero, round-

to-infinity 대해 구현하는 것을 살펴보고, NS시와 RS시와 f_{2n-1}과의 관계를 살펴보겠다.

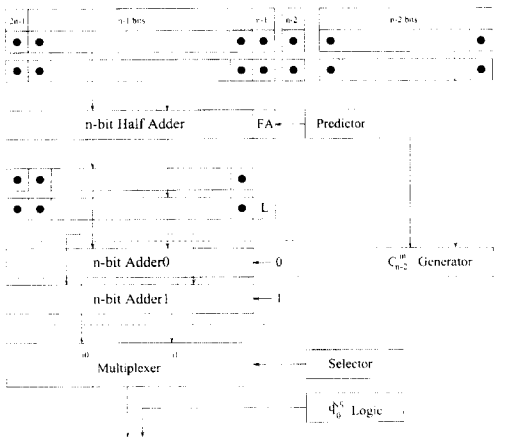


그림 2. IEEE 반올림과 덧셈을 동시에 수행하는 하드웨어 모델

Fig. 2. Hardware model performing IEEE rounding and addition in parallel.

1. IEEE 반올림과 덧셈을 동시에 수행하는 하드웨어 모델

그림 2는 덧셈과 IEEE에서 지정한 반올림을 병렬적으로 수행할 수 있는 하드웨어 모델이다. 이는 n 비트의 반가산기(Half adder), 1 비트의 전가산기(Full Adder), predictor, 캐리 선택 덧셈기(Carry Select Adder), C_{n-2}^{in} generator, q_0^{NS} 생성 로직으로 구성이 되어 있다. 먼저, 2n 비트의 C와 S에서 C_I와 S_I를 n 비트의 반가산기와 1 비트의 전가산기를 가지고 더한다. 이때, predictor값이 전가산기에 입력된다. 반가산기들과 전가산기를 거치면 n 비트의 캐리(carry)와 n+1 비트의 합(sum)이 발생한다. 이때, n+1 비트의 합(sum)의 LSB(Least Significanf Bit)를 그림 2에서 L이라고 한다. 반가산기들과 전가산기에서 생성되는 캐리와 합의 상위 n 비트 값은 n 비트의 올림수 선택 덧셈기(Carry Select Adder)[7,15,16]인 Adder0와 Adder1로 더해지고 결과 값이 멀티플렉서(Multiplexer)에 입력된다. Adder0와 Adder1는 두개의 덧셈기가 아니고, 고성능 덧셈기의 일종인 올림수 선택 덧셈기 한개로 구현이 되며, 이는 일반적인 부동소수점 덧셈/뺄셈기와 곱셈기에 필수적으로 쓰인다. selector는 멀티플렉서의 두 입력값 중 덧셈과 반올림이 수행된 값을 선택하는 것이며, 멀티플렉서의 제어신호인 selector = 0 이면 i0를 선택하고 selector = 1이면 i1을 선택한다. C_{n-2}^{in} Generator는 식 (2)에서의 C_{n-2}^{in}을 생성하는 로직이고, q_0^{NS}는 NS시 반올림이 수행된 값의 LSB를 생성하는 로직이다.

그림 2에서 살펴보면 멀티플렉서의 출력값으로는 selector의 값에 따라서 $D_I+predictor$와 $D_I+predictor+2$이다. 식 (8)과 (9)에 의하여, 반올림과 덧셈을 동시에 수행하기 위해 D_I, D_I+1, D_I+2 and D_I+3의 결과값을 적절히 생성해야 한다. 따라서, 그림 2에서의 predictor와 selector를 각각의 경우에 맞게 선택을 하면 덧셈과 반올림을 동시에 수행한 값인 Q를 생성해 낼수 있다. 이때, predictor의 로직을 구성하는데 다음의 두가지 요소를 고려해야 한다. 첫번째로, predictor로 입력되는 신호들은 올림수 선택 덧셈기가 덧셈을 시작하기 전에 이미 결정이 되어져야 한다. 두번째는, selector의 지연 시간이 거이 무시될 수 있게 되도록 predictor를 선정해야 한다. selector가 최적화되고 무시될 수 있는 지연시간을 가지게 할려면, predictor는 매우 신중하게 선택이 되어져야 한다. 다음 절들에는 predictor의 선택과 그에 따른 selector의 구성에 대해서 논한다.

멀티플렉서에서 i0의 입력값을 $E=e_ne_{n-1}e_{n-2}\cdots e_1$라 하자. 그런데, E의 LSB는 C와 S에서 n 번째 비트에 해당함으로, E의 정수값은 $E_I=E\times 2$이다. 그리고, n+1 비트의 정수부 E를 E^*라 하면, $E_I^*=E_I+L$이다. 따라서, E_I^*는 다음과 같이 나타낼수 있다.

$$E_I^*=E_I+L=C_I+S_I+predictor=f_{2n-1}e_{n-1}\cdots e_1L \quad (10)$$

다음의 절들에서는 각각의 반올림 모드에 따라 제시된 하드웨어 모델에 따르는 predictor와 selector의 선정에 대해서 논한다.

2. round-to-nearest

round-to-nearest일때는 RS시 반올림의 결과가 올림이 되기 위해서는 f_{n-1}이 Algorithm 1에 의해서 1이 되야 한다. 따라서, 식 (8)과 (9)에 의해 반올림과 덧셈을 동시에 수행하기 위해서 D_I, D_I+1, D_I+2의 생성이 필요하다. 그런데, predictor = 0이면 멀티플렉서의 입력으로는 D_I와 D_I+2가 된다. D_I 혹은 D_I+2의 상위 n 비트는 D_I+1의 상위 n 비트와 동일하다. 따라서, predictor = 0일때 D_I, D_I+1,

D_I+2가 모두 생성될수 있다. 그러나, predictor를 결정할때, c_{n-2}와 s_{n-2}로 부터의 올림수를 고려해야 한다. 따라서, 효율적인 selector를 생성하기 위하여, predictor를 c_{n-2}와 s_{n-2}의 올림수 신호로 선정하였다. 이는 overflow $(c_{n-2}+s_{n-2})$과 동일하다. 따라서, predictor는 다음 식 (11)과 같다.

$$predictor = c_{n-2} \wedge s_{n-2} \tag{11}$$

식 (11)의 predictor는 overflow $(c_{n-2}+s_{n-2})$과 동일하기 때문에, 식(2)는 다음 식(12)로 나타낼 수 있다. 여기서 C^{in*}_{n-1}과 C^{in*}_{n}은 이전 식들에서의 C^{in}_{n-1}과 C^{in}_{n}을 구분하기 위한 것이다.

$$\begin{aligned}
F &= (c_{2n-1}c_{2n-2}\cdots c_{n-1}.c_{n-2}) + (s_{2n-1}s_{2n-2}\cdots s_{n-1}.s_{n-2}) \\
&+ (0.0c_{n-3}\cdots c_0) + (0.0s_{n-3}\cdots s_0) \\
&= C_I + S_I + 0.c_{n-2} + 0.s_{n-2} + 0.C^{in}_{n-2} + 0.0Sy \\
&= C_I + S_I + overflow(c_{n-2}+s_{n-2}) + 0.(c_{n-2}\oplus s_{n-2}) + 0.C^{in}_{n-2} + 0.0Sy \\
&= C_I + S_I + predictor + 0.(c_{n-2}\oplus s_{n-2}) + 0.C^{in}_{n-2} + 0.0Sy \\
&= E_I + L + 0.(c_{n-2}\oplus s_{n-2}) + 0.C^{in}_{n-2} + 0.0Sy \\
&= E_I + L + C^{in*}_{n-1} + 0.R + 0.0Sy \\
&= E_I + C^{in*}_{n} + f_{n-1} + 0.R + 0.0Sy
\end{aligned} \tag{12}$$

$$\begin{aligned}
R &= c_{n-2}\oplus s_{n-2}\oplus C^{in}_{n-2} \\
C^{in*}_{n-1} &= overflow((c_{n-2}\oplus s_{n-2}) + C^{in}_{n-2}) = (c_{n-2}\oplus s_{n-2}) \wedge C^{in}_{n-2} \\
C^{in*}_{n} &= overflow(L + C^{in*}_{n-1}) = L \wedge C^{in*}_{n-1} \\
f_{n-1} &= L \oplus C^{in*}_{n-1} \\
f_n &= e_1 \oplus C^{in*}_{n}
\end{aligned}$$

그림 1에 의해서, (n-2) 번째 위치는 소수점 아래로 부터 첫번째 비트이고, (n-1) 비트는 소수점 위로 부터 첫번째 비트이며 정수부분의 첫번째 비트이고, n번째 비트는 정수부분의 두번째 비트이다. 따라서, 위치정보에 입각하여 식 (12)에서 C^{in}_{n-2}은 $0.C^{in}_{n-2}$으로 정렬이 되며, C^{in*}_{n-1}은 그대로 C^{in*}_{n-1}이 되며, C^{in*}_{n}는 $2\times C^{in*}_{n}$으로 표현될수 있다.

$C^{in*}_{n}=1$일때는 식 (12)에 의해 L과 C^{in*}_{n-1}가 1이 되며, 따라서 $c_{n-2}\oplus s_{n-2}=1$, $C^{in}_{n-2}=1$이 된다. 따라서, 이 경우 식 (12)에 따라 $f_{n-1}=R=0$ 임으로 Algorithm 1에 의해 NS시와 RS시 반올림의 결과가 버림이다. 즉, $C^{in*}_{n}=1$이고 반올림의 결과가 올림인 경우는 없다. 따라서, 식 (8)과 (12)에 의해 Q^{NS}는 다음과 같다.

$$\begin{aligned}
Q^{NS} &= F_I + Rounding_{Nearest}(f_{n-1}, R, Sy) \\
&= E_I + L + C^{in*}_{n-1} + Rounding_{Nearest}(f_{n-1}, R, Sy) \\
&= E_I + C^{in*}_{n} + f_{n-1} + Rounding_{Nearest}(f_{n-1}, R, Sy)
\end{aligned} \tag{13}$$

따라서, 식 (13)에 의해 NS시 selector와 q_0^{NS}는 식 (14)과 같다.

$$\begin{aligned}
selector &= C^{in*}_{n} \vee (f_{n-1} \wedge Rounding_{Nearest}(f_{n-1}, R, Sy)) \\
q_0^{NS} &= f_{n-1} \oplus Rounding_{Nearest}(f_{n-1}, R, Sy)
\end{aligned} \tag{14}$$

또한, $Round_{Nearest}(f_n, f_{n-1}, R \vee Sy)=1$이 될려면 $f_{n-1}=1$이 되야하기 때문에, 식 (9)과 (12)에 의해 Q^{RS}는 다음과 같다.

$$\begin{aligned}
Q^{RS} &= F_I + Rounding_{Nearest}(f_n, f_{n-1}, R \vee Sy) \\
&= E_I + L + C^{in*}_{n-1} + Rounding_{Nearest}(f_n, f_{n-1}, R \vee Sy) \\
&= E_I + C^{in*}_{n} + f_{n-1} + Rounding_{Nearest}(f_n, f_{n-1}, R \vee Sy)
\end{aligned} \tag{15}$$

따라서, 식 (15)에 의해 RS시 selector는 식 (16)과 같다.

$$selector = C^{in*}_{n} \vee Rounding_{Nearest}(f_n, f_{n-1}, R \vee Sy) \tag{16}$$

3. round-to-zero

round-to-zero의 predictor는 round-to-nearest의 predictor와 동일하게 놓는다. 그런데, $Rounding_{Zero}(X, X, X)=0$임으로 NS시와 RS시 selector와 q_0^{NS}는 식 (14)과 (16)의 $Rounding_{Nearest}$를 0으로 대치하여 적용하면 식 (17)과 같다.

$$\begin{aligned}
selector &= C^{in*}_{n} \\
q_0^{NS} &= f_{n-1}
\end{aligned} \tag{17}$$

4. round-to-infinity

$C^{in}_{n-2}=L=predictor=Sy=1$의 상황에서 round-to-nearest와 같은 predictor를 쓰는 경우, $C^{in*}_{n}=1$, $f_{n-1}=0$, $R=0$, Sy = 1이 된다. 따라서, Algorithm 1에 의하여 round-to-nearest의 경우 반올림의 결과가 버림이다. 그런데, round-to-infinity시에는 Sy = 1 임으로 Algorithm 3에 의하여 반올림의 결과가 올림이 된다. 따라서, round-to-infinity시에는 round-to-nearest의 predictor를 쓸 수가 없다.

round-to-infinity일때 predictor를 다음 식 (18)로 결정하였다.

$$predictor = (c_{n-2} \vee s_{n-2}) \tag{18}$$

표 1은 C^{in}_{n-1}, R, predictor의 값이 c_{n-2}, s_{n-2}, C^{in}_{n-2}에 대하여 제시되어 있다. 이는, $c_{n-2}\oplus s_{n-2}$와 C^{in}_{n-2}의 값에 따라 다음의 두가지 경우로 나누어 볼 수 있다. 첫째로 $C^{in}_{n-2}=0$이고 $c_{n-2}\oplus s_{n-2}=1$일때이다. 이경우는 $C^{in}_{n-1}=0$이 되고 R과 predictor는 1이 된다. 두번째의 경우는, 첫번째 경우를 제외한 모든

경우이며, 이때는 C^{in}_{n-1}과 predictor의 값이 동일하게 된다. 첫번째의 경우에서는, R=1이므로 Algorithm 3에 의해서 반올림의 결과가 올림이다. 이때, $C^{in}_{n-1}=0$이 되고 predictor=1이 된다. 따라서, 식 (8)과 (10)에 의해 Q^{NS}는 다음과 같다.

표 1. c_{n-2}, s_{n-2}, C^{in}_{n-2}의 값에 따른 C^{in}_{n-1}, R, predictor의 결과

Table 1. The result values of C^{in}_{n-1}, R, predictor according to c_{n-2}, s_{n-2}, C^{in}_{n-2}.

c_{n-2}	s_{n-2}	C^{in}_{n-2}	$c_{n-2}\oplus s_{n-2}$	C^{in}_{n-1}	R	Predictor
0	0	0	0	0	0	0
0	0	1	1	0	1	1
0	1	0	1	0	1	1
0	1	1	0	1	0	1
1	0	0	0	0	1	0
1	0	1	1	1	0	1
1	1	0	1	1	0	1
1	1	1	0	1	1	1

$$\begin{aligned} Q^{NS} &= C_I+S_I+C^{in}_{n-1}+Rounding_{Infinity}(f_{n-1},R,Sy)\\ &= C_I+S_I+1\\ &= C_I+S_I+predictor\\ &= E_I+L \end{aligned}$$

따라서 NS시 q_0^{NS}와 selector는 식 (19)과 같다.

$$\begin{aligned} selector &= 0\\ q_0^{NS} &= L \end{aligned} \tag{19}$$

RS시에는 식 (9)과 (10)에 의해 Q^{RS}는 다음과 같다.

$$\begin{aligned} Q^{RS} &= C_I+S_I+C^{in}_{n-1}+2\times Rounding_{Infinity}(f_n,f_{n-1},R\vee Sy)\\ &= C_I+S_I+2\\ &= C_I+S_I+predictor+1\\ &= E_I+L+1 \end{aligned}$$

따라서 RS시 selector는 식 (20)과 같다.

$$selector=L \tag{20}$$

두번째의 경우에서는, C^{in}_{n-1}과 predictor의 값이 동일하다. 따라서, NS시 Q^{NS}는 다음과 같다.

$$\begin{aligned} Q^{NS} &= C_I+S_I+C^{in}_{n-1}+Rounding_{Infinity}(f_{n-1},R,Sy)\\ &= C_I+S_I+predictor+Rounding_{Infinity}(f_{n-1},R,Sy)\\ &= E_I+L+Rounding_{Infinity}(f_{n-1},R,Sy) \end{aligned}$$

따라서, selector와 q_0^{NS}는 다음과 같이 나타낼수 있다.

$$\begin{aligned} selector &= L\wedge Rounding_{Infinity}(f_{n-1},R,Sy)\\ q_0^{NS} &= L\oplus Rounding_{Infinity}(f_{n-1},R,Sy) \end{aligned} \tag{21}$$

RS시 Q^{RS}는 다음과 같다.

$$\begin{aligned} Q^{RS} &= C_I+S_I+C^{in}_{n-1}+2\times Rounding_{Infinity}(f_n,f_{n-1},R\vee Sy)\\ &= C_I+S_I+predictor+2\times Rounding_{Infinity}(f_n,f_{n-1},R\vee Sy)\\ &= E_I+L+2\times Rounding_{Infinity}(f_n,f_{n-1},R\vee Sy) \end{aligned}$$

따라서, selector는 다음과 같다.

$$selector=Rounding_{Infinity}(f_n,f_{n-1},R\vee Sy) \tag{22}$$

5. f_{2n-1}과 e_n의 관계

이번 절에서는 NS와 RS를 결정하는 신호인 f_{2n-1}과 e_n과의 관계를 각각의 반올림 모드에 대해 살펴보겠다. 식 (5), (10), (11)에 의해 round-to-nearest시 F_I와 E_I^*는 다음 식 (23)과 같다.

$$\begin{aligned} E_I^* &= C_I+S_I+predictor=D_I+predictor\\ C^{in}_{n-1} &= overflow(C^{in}_{n-2}+c_{n-2}+s_{n-2})\\ F_I &= C_I+S_I+C^{in}_{n-1}=D_I+C^{in}_{n-1}\\ predictor &= overflow(c_{n-2}+s_{n-2}) \end{aligned} \tag{23}$$

이때, $e_n=1$이면 식 (23)에서 보는바와 같이 $F_I\geq E_I^*$임으로 $f_{2n-1}=1$이다. 따라서, $e_n=1$일때는 RS의 경우가 된다.

$e_n=0$일때는 다음 3가지 경우가 발생한다. 첫째로, predictor = 1일 경우는 식 (23)에 의해 $C^{in}_{n-1}=1$이다. 따라서, $F_I=E_I^*$이며 $f_{2n-1}=0$이다. 둘째로, predictor = 0이고 $C^{in}_{n-1}=0$일 경우는 식(23)에 의해 $F_I=E_I^*$이다. 따라서, $f_{2n-1}=0$이다. 셋째로, predictor = 0이고 $C^{in}_{n-1}=1$일 경우는 식 (23)에 의해 $C^{in}_{n-2}=1$이고 $c_{n-2}\oplus s_{n-2}=1$이다. 이때, 식(23)의 $D_I=d_nd_{n-1}\cdots d_0$에서 모든 k에 대해 $d_k=1(0\leq k\leq n-1)$이고 $d_n=0$일 경우 $F_I=100\cdots000$이고 $E_I^*=011\cdots111$이며, 식(12)에 의해 $f_{n-1}=0$이고 R = 0이 된다. 즉, $e_n=0$이고, $f_{2n-1}=1$인 경우가 발생한다. 그런데, 이 경우 NS시와 RS시 반올림시 버림이기 때문에 반올림을 고려하지 않아도 됨으로, $f_{2n-1}=1$임으로 RS시 이지만 E_I^*에 대해 NS의 경우로 처리해도 무방하다.

위의 round-to-nearest의 경우들을 종합해 보면 round-to-nearest시 그림 2에서 식(11)의 predictor를 사용하면 $e_n=1$일때는 RS의 경우이며 $e_n=0$일때는 NS의 경우로 처리해도 무방하다. round-to-zero시 반올림시 모든 경우에 대해 버림임으로 $e_n=1$일때는 RS의 경우이며 $e_n=0$일때는 NS의 경우로 처리해도 무방하다.

Round-to-infinity 일때는 $c_{n-2} \oplus s_{n-2}$와 C^{in}_{n-2}의 값에 따라 두가지 경우로 나누어 짐을 전 절에서 살펴 보았다. 첫째로 $C^{in}_{n-2}=0$이고 $c_{n-2} \oplus s_{n-2}=1$일때 이다. 이경우는 $C^{in}_{n-1}=0$이 되고 R과 predictor는 1이 된다. 이때, F_1과 E^*_1는 식(5),(10),(18)에 의해 다음과 같이 나타낼수 있다.

$$\begin{aligned} F_1 &= C_1+S_1+C^{in}_{n-1}=D_1+0 \\ E^*_1 &= C_1+S_1+predictor=D_1+1 \\ C^{in}_{n-1} &= overflow(C^{in}_{n-2}+c_{n-2}+s_{n-2}) \end{aligned} \quad (24)$$

이때, $f_{2n-1}=0$과 $e_n=1$의 경우가 발생 할수 있다. 즉, 모든 k에 대해 $d_k=1(0 \le k \le n-1)$이고 $d_n=0$일 경우 $F_1=011\cdots111$이고 $E^*_1=100\cdots000$이다. 이 경우 식(12)에 의해 R = 1이 되며, 반올림의 결과가 올림이 된다. 따라서, F에 대해 반올림의 후의 상위 n 비트의 결과값은 $Q^F=100\cdots000$이다. 그런데, $E^*_1=100\cdots000$이고 $C^{in}_{n-1}=0$임으로 식(20)에 의해 반올림 후의 상위 n 비트의 결과값 $Q^E=100\cdots000$이 된다. 따라서, 결국 $Q^F=Q^E$이다. 두번째의 경우는 C^{in}_{n-1}과 predictor의 값이 동일함으로, e_n과 f_{2n-1}은 동일한 값을 가진다.

결론적으로, 각각의 반올림 모드에 대해 그림2의 Adder0에서 발생하는 f_{2n-1}=0이면 NS시 이고 f_{2n-1}=1이면 RS시 이다. 이 결과를 식(14), (16), (17), (19), (20), (21), (22)에 적용하면 각각의 반올림 모드에 대해 그림2의 selector의 로직을 구할 수 있다.

VI. 결 론

일반적인 부동 소수점 곱셈 연산은 반올림 단계에서 별도의 고속 가산기나 증가기를 필요로하여 많은 면적과 처리 시간을 차지한다. 또한, 재정규화 혹은 정규화 전에 반올림을 하기 위한 별도의 하드웨어가 필요하다. 본 논문은 부동 소수점 곱셈 연산기에서 덧셈과 반올림을 동시에 함으로써 일반적인 4 단계의 연산 단계를 곱셈, 덧셈과 반올림, 정규화의 3 단계로 줄였다. 덧셈 연산시 사용되는 고속 덧셈기인 올림수 선택 덧셈기를 이용하여 덧셈과 IEEE에서 지정한 반올림을 동시에 처리 하였다.

본 논문에서 제안하는 부동 소수점 곱셈 연산기는 반올림을 위한 별도의 하드웨어를 필요치 않으나, n 비트의 반가산기와 1 비트의 전가산기와 selector와 q_0^{NS} 로직이 필요하다. 그런데, 기존의 부동 소수점 곱셈 연산기에서의 반올림을 위한 고속 덧셈기와 그 외의 부가적인 로직은 본 논문에서 추가된 하드웨에 비해 대단히 큰 면적과 처리 시간을 차지 한다. 본 논문의 부동 소수점 곱셈 연산기는 일반적인 부동 소수점 곱셈 연산기에 비해 대략 한개의 고속 덧셈기 만큼의 면적과 시간면에서 잇점이 있다. 또한, 제안하는 부동 소수점 연산기는 [11]과 비교하면 IEEE에 준하는 모든 반올림 수행이 가능하고, [2]에 비해서는 성능과 차지면적에서 우수하고, [12]와 비교해서는 수식적인 분석을 통하여 최적화된 게이트레벨의 구현을 가능하게 했다.

그림 2의 selector와 q_0^{NS}는 C^{in}_{n-2} generator에서 발생하는 C^{in}_{n-2}와 덧셈 연산후에 발생하는 f_{2n-1}과 덧셈 연산전이나 도중에 발생하는 신호인 L, e_0, c_{n-2}, s_{n-2}, Sy을 식(14), (16), (17), (19), (20) (21), (22)에 적용하여 구할수 있다. 이때, f_{2n-1}과 C^{in}_{n-2}가 결정된 후 selector의 신호가 나올때 까지 지연 시간이 생기는데, selector와 q_0^{NS}의 로직을 최소화(minimize)하면 지연시간이 무시할 수 있을 정도로 된다. 본 논문에서 사용된 내용들은 division과 multiply-add-fused instruction의 구현시 반올림 처리에 응용할 수 있다. 앞으로 이에 대한 연구를 할 예정이다.

참 고 문 헌

[1] Kenneth C. Yeager, "The Mips R10000 Superscalar Microprocessor," *IEEE Micro*, vol. 16, no. 2, pp. 28-40, Apr. 1996.

[2] Marc Tremblay and J. Michael O'Connor, "Ultra Sparc I : A Four-Issue Processor Supporting Multimedia," *IEEE Micro*, vol. 16, no. 2, pp. 42-50, Apr. 1996.

[3] D. Alpert and D. Avinon, "Architecture of the pentium Microprocessor," *IEEE Micro*, vol. 13, no. 3, pp. 11-21, June 1993.

[4] E. McLellan, "The Alpha Architecture and 21064 Processor," *IEEE Micro*, vol. 13, no. 3, pp. 36-47, June 1993.

[5] M.C. Becker et al, "The PowerPC 601

Microprocessor," *IEEE Micro*, vol. 13, no. 5, pp. 54-67, Oct. 1993.

[6] N. Ide et al., "A 320-MFLOPS CMOS Floating-Point Processing Unit for Superscalar Processors," *IEEE Journal of Solid-state Circuit*, vol. 28, no. 3, pp. 352-360, March 1993.

[7] J.L. Hennessy and D.A. Patterson, "Computer Architecture A Quantitative Approach," *Morgan Kaufmann Publishers Inc*, 1990.

[8] M. Awaga er al., "The muVP 64-Bit Vector Coprocessor:A New Implementation of High-Performance Numerical Computation," *IEEE Micro*, vol. 13, no. 5, pp. 24-36, Oct. 1993.

[9] M. Darley et al., "The TMS390C602A Floating-Point Coprocessor for Sparc Systems," *IEEE Micro*, pp. 36-47, June 1990.

[10] M. Birman et al., "Developing the WTL3170/3171 Sparc Floating-Point Coprocessors," *IEEE Micro*, vol. 8, no. 1, pp. 55-64, Feb. 1990.

[11] M. R. Santoro et al., "Rounding Algorithms for IEEE Multiplier," *Proceedings of the 9th Symposium on Computer Arithmetic*, pp. 176-183, June 1989.

[12] N. Quach et al., "On Fast IEEE Rounding," Techcal Report CSL-TR-91-459, Stanford University, Jan. 1991.

[13] IEEE Std 754-1985, "IEEE Standard for Binary Floating-Point Arithmetic," *IEEE*, 1985.

[14] C. S. Wallace, "A Suggestion for Fast Multipliers", *IEEE Transactions on Electronic Computers*, vol. EC-13, pp. 14-17, Feb. 1964.

[15] O. J. Bedrij, "Carry-Select Adder," *IEEE Transactions on Electronic Computers*, vol. EC-11, pp. 340-346, June 1962.

[16] M. Uya et al., "A CMOS Floating Point Multiplier," *IEEE Journal of Solid-state Circuits*, vol. SC-19, pp. 697-702, Oct. 1984.

저 자 소 개

朴 祐 贊(正會員)

1993년 연세대학교 이과대학 전산과학과 학사. 1995년 연세대학교 이과대학 전산과학과 석사. 1995년 ~ 현재 연세대학교 공과대학 컴퓨터과학과 박사 재학중. 관심분야는 ASIC 설계, 3차원 그래픽 가속기, Computer arithmetic, 고성능 컴퓨터구조

梁 眞 冀(正會員)

1997년 연세대학교 공과대학 컴퓨터과학과 학사. 1997년 ~ 현재 연세대학교 공과대학 컴퓨터과학과 석사 재학중. 관심분야는 ASIC 설계, 3차원 그래픽 가속기, Computer arithmetic, 병렬처리 컴퓨터구조

鄭 喆 浩(正會員)

1996년 숭실대학교 정보과학대학 컴퓨터학부 학사. 1997년 ~ 현재 연세대학교 공과대학 컴퓨터과학과 석사 재학중. 관심분야는 ASIC 설계, 3차원 그래픽 가속기, Computer arithmetic, 병렬처리 컴퓨터구조

韓 鐸 敦(正會員)

1978년 연세대학교 공과대학 전자공학과 학사. 1983년 Wayne State University 컴퓨터공학 석사. 1987년 University of Massachusetts 컴퓨터공학 박사. 1981년 금성전기 연구원. 1987년 ~ 1989년 Cleveland 주립대학 조교수. 1989년 ~ 현재 연세대학교 공과대학 컴퓨터과학과 부교수. 관심분야는 병렬처리 컴퓨터구조, ASIC 설계, 고성능 컴퓨터구조, Memmory-processor integration, Web computing