論文94-31A-3-7

파이프라인 데이타패스 자동 생성을 위한 상위수준 합성 시스템의 설계

(Design of a High-Level Synthesis System for Automatic Generation of Pipelined Datapath)

李 海 東*, 黃 善 泳*

(Hae Dong Lee and Sun Young Hwang)

要 約

본 논문에서는 파이프라인 방식으로 연산과정을 처리하는 하드웨어를 자동적으로 생성하는 상위수준 합성 시스템인 SODAS-VP에 대하여 기술한다. 파이프라인 데이타패스의 타겟 아키텍춰와 클럭위상을 결정하였고, 파이프라인의 성능을 감소시킬 수 있는 파이프라인 hazard를 처리하였다. 파이프라인 데이타패스를 생성하기 위하여 하나의 연산과정은 적재, 연산, 저장의 부연산으로 분할되고, 각 부연산은 분할된 제어구간에서 수행된다. 분할 과정에서 파이프라인 hazard를 internal forwarding 또는 지연삽입 방식을 사용하여 처리하고, similarity measure를 우선순위 함수로 사용하는 모듈할당 과정에서 분할된 제어구간 사이의 자원충돌을 제거하여 파이프라인 방식의 하드웨어를 합성한다.

실험결과를 통해 SODAS-VP가 HAL, ALPS 시스템보다 빠른 수행속도의 하드웨어를 생성함을 보였다. SODAS-VP가 MCNC 벤치마크 프로그램인 5차 엘립틱 웨이브 필터에 대해 생성한 하드웨어의 수행속도가 HAL과 ALPS 시스템에 비해 각각 17.1%와 7.4% 정도의 속도향상을 보였다.

Abstract

This paper describes the design of a high-level synthesis system, SODAS-VP, which automatically generates hardwares executing operation sequences in pipelined fashion. Target architecture and clocking schemes to drive pipelined datapath are determined, and the handling of pipeline hazards which degrade the performance of pipeline is considered. Partitioning of an operation into load, operation, and store stages, each of which is executed in partitioned control step, is performed. Pipelinecl hardware is generated by handling pipeline hazards with internal forwarding or delay insertion techniques in partitioning process and resolving resource conflicts among the partitioned control steps with similarity measure as a priority function in module allocation process.

Experimental results show that SODAS-VP generates hardwares that execute faster than those generated by HAL and ALPS systems. SODAS-VP brings improvement in execution speed by 17.1% and 7.4% comparing with HAL and ALPS systems for a MCNC benchmark program, 5th order elliptical wave filter, respectively.

Ⅰ. 서 론

파이프라인 방식은 throughput을 높이기 위해 하나의 작업을 여러개의 부작업으로 분할하여 수행하는

* 正會員, 西江大學校 電子工學科
(Dept. of Elec. Eng., Sogang Univ.)
接受日字 : 1993年 2月 8日

방식으로, 이상적인 파이프라인은 스테이지의 갯수에 비례하여 throughput이 증가한다는 장점이 있다. 대표적인 예로 RISC[1]에서 사용하는 인스트럭션 파이프라인은 하나의 인스트럭션을 여러 스테이지로 나누어 수행하여 평균 한 사이클당 한 인스트럭션의 수행속도를 높이고 있다. 데이타패스 상에서 수행되는 연산과정은 레지스터에서 데이타를 읽는 과정인 적재과정, 실제 연산을 수행하는 연산과정, 연산결과를 레지스터에 저장하는 저장과정을 통해 순차적으로 수행된다.

상위수준 합성 (High-Level Synthesis)은 top-down 설계 방식을 지원하기 위해 알고리듬 수준의 하드웨어 기술 언어 (Hardware Description Language)를 입력으로 받아들여 RT (Register Transfer) 수준의 데이타패스를 자동적으로 생성하는 과정이다. 상위수준 합성은 1960년대에 처음으로 소개되기 시작하여, 지금까지 많은 연구가 진행되어 왔고 현재에도 많은 시스템이 발표되면서 활발한 연구가 진행되고 있다.[2] [3] [4]

상위수준 합성 과정에 파이프라인 방식을 지원하기 위해서는 클럭위상과 같은 타이밍, 타겟 아키텍춰 등에 대한 고려가 선행되어야 하고, 파이프라인 데이타패스에서 발생할 수 있는 여러가지 hazard의 처리도 고려되어야 한다. 파이프라인 방식은 이미 범용 컴퓨터 아키텍춰부터 DSP (Digital Signal Processing) 전용 칩에 이르기까지 여러 분야에 걸쳐 사용되고 있다.[5] [6] 최근에는 생성하고자 하는 하드웨어의 타겟 아키텍춰를 설정한 상태에서 상위수준 합성을 수행하므로써 합성범위를 축소시키는 방향으로 연구가 진행되고 있으며 대표적인 예로 DSP 전용 실리콘 컴파일러가 있다. 현재 개발된 DSP 상위수준 합성 시스템으로는 Sehwa[5], CATHEDRAL[7] 시스템 등이 있고, 성능향상을 위해 파이프라인 방식을 사용하고 있다. 이들 파이프라인 합성 시스템은 입력 행위기술로부터 전체 데이타/콘트롤 흐름에 대한 정보를 가지는 중간형태를 생성하고, 주어진 DII (Data Initiation Interval)에 의하여 중간형태를 파이프라인 방식으로 중첩하여 합성하므로써 수행시간을 향상시킨다. 그러나, 스케쥴링 과정에 의하여 각 제어구간에 할당된 연산과정을 파이프라인 방식으로 수행하는 하드웨어를 합성하는 시스템은 아직 발표되고 있지 않다.

기존의 파이프라인/비파이프라인 상위수준 합성 시스템은 데이타의 적재, 연산, 결과의 저장 과정이 하나의 제어구간에서 모두 이루어지는 하드웨어를 합성하였다. 이러한 경우 한 연산과정의 수행시간은 세 과정의 지연시간의 합으로 주어진다. 그러나, 상위수준 합성에서 연산과정을 파이프라인 방식으로 수행할 경우, 하나의 연산과정은 세 개의 스테이지로 분할되고 클럭의 주기는 스테이지의 지연시간으로 줄어들며, 한 주기 내에서 하나의 연산과정이 수행될 수 있어 수행시간 측면에서 유리한 결과를 가져온다. 연산과정을 파이프라인 방식으로 처리할 경우 제어의 분기에 의한 콘트롤 hazard, 데이타 의존성에 의한 데이타 hazard와 같은 파이프라인 hazard가 발생할 수 있다. 본 논문에서 제안한 시스템 SODAS-VP는 콘트롤 hazard의 발생시 지연소자를 삽입하므로써 해결하며, 데이타 hazard는 internal forwarding 또는 지연소자 삽입 방식으로 처리한다.

본 논문에서는 주어진 동작 기술로부터 연산과정을 적재, 연산, 그리고 저장의 부작업으로 분할하여 파이프라인 방식으로 처리하는 하드웨어를 자동 생성하는 상위수준 합성 시스템 SODAS-VP에 대하여 기술하며 전체 구성은 다음과 같다. 2 장에서는 파이프라인의 개요와 발생할 수 있는 hazard와 이들의 처리에 대해 기술하고, 클럭위상과 타겟 아키텍춰의 결정을 3 장에서 설명하였다. 4 장에서 파이프라인을 지원하는 상위수준 합성 과정과 중간형태에 대하여 기술하고, 5 장에 실험결과를 보이며 끝으로 6 장에서 결론을 맺는 순서로 기술한다.

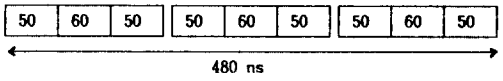
Ⅱ. 파이프라인

1 개요

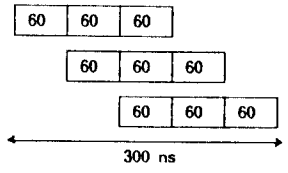
파이프라인은 처리할 작업을 동일한 지연시간을 가지는 2개 이상의 스테이지로 분할하고, 분할된 스테이지내에서 여러 작업을 중첩시켜 처리하는 방식으로서, 이미 범용 컴퓨터 시스템에서 적용되어 수행시간 측면에서의 효율성이 입증되었다. 그림 1에 수행시간 측면에서 파이프라인과 비파이프라인 방식의 비교를 보였다. 그림 1 (a)는 비파이프라인 방식의 예로서 50, 60, 50 ns의 부작업으로 구성되는 하나의 작업이 비파이프라인 방식으로 세번 수행되는 경우를 보인 것으로, 전체 수행시간은 480 ns가 소요된다. 그림 1 (b)는 그림 1 (a)의 작업을 파이프라인 방식으로 수행한 경우로서 전체 작업의 수행시간이 300 ns가 되어 비파이프라인 방식보다 효율적인 결과를 얻을 수 있다. 여기서 파이프라인의 각 스테이지는 동일한 지연시간을 가져야 하므로 최대 수행시간을 가지는 60 ns로 주어진다. 일반적으로 파이프라인 스테이지의 지연시간은 아래식으로 주어진다.

SD (Stage Delay) : 스테이지의 지연시간
ST (Sub-Task) : 부작업의 지연시간
N : 부작업의 갯수

$SD = Max (ST_1, ST_2, ST_3, \cdots, ST_N)$



(a)



(b)

그림 1. 파이프라인과 비파이프라인의 비교
(a) 비파이프라인
(b) 파이프라인
Fig. 1. Comparison of pipelined & non-pipelined execution.
(a) non-pipelined execution.
(b) pipelined execution.

2. Pipeline Hazards

파이프라인 방식은 수행시간 측면에서의 잇점이 있지만, 연속적인 작업간의 데이타 의존성 및 제어 의존성에 의해 발생되는 pipeline hazard가 문제점으로 생길 수 있다. Pipeline hazard는 스테이지 간의 자원충돌에 의해 발생하는 structural hazard, 데이타 의존성에 의해 발생하는 data hazard, 제어의 분기에 의해 발생하는 control hazard 등이 있다. [8]

그림 2에 data hazard의 예를 보였다. Data hazard는 데이타 의존성에 의해 발생하는 경우로서, ST1의 변수 a는 ST2의 입력 변수로서 사용되어 데이타 의존성의 관계가 있다. 그림 2-(b)는 이러한 경우의 파이프라인 스테이지를 보인 것으로, S3에서 변수 a에 새로운 결과가 저장되기 전에 S4의 적재(load) 연산을 수행하게 되어 결과적으로 잘못된 결과가 변수 d에 저장된다. 이러한 data hazard를 해결하기 위한 방법으로 지연삽입 (delay insertion)과 internal forwarding (이하 IF) 기법이 있다 [1] . 지연 삽입 방법은 S3의 수행이 완료될 때까지 지연소자 (delay slot)를 삽입하여 ST2의 수행을 지연시키는 방법으로 구현이 간단하다는 장점이 있는 반면에 수행시간 측면에서 단점이 있다. 그림 2-(c)에 지연 삽입 방법을 보였다. IF 방식은 지연소자를 사용하지 않고 직접 연산자의 입출력 사이를 선으로 연결하는 방법으로 그림 2-(d)에 보였다.

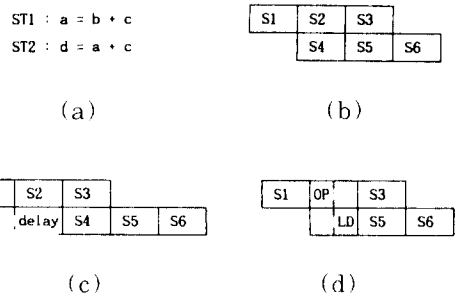


그림 2. Data hazard의 예
Fig. 2. An example of data hazard.

Control hazard는 순차적인 콘트롤의 흐름에서 if, loop 문과 같은 분기가 발생할 경우 나타나는 현상으로 그림 3에 예를 보였다. 조건문에 의해 분기되는 연산과정의 주소을 결정할 때까지 지연소자를 삽입하므로써 데이타의 충돌을 방지한다. Structural hazard는 자원 충돌에 의해 발생하며 모듈할당 과정에서 이를 고려하여 합성하여야 한다.

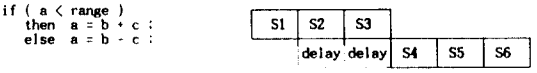


그림 3. Control hazard의 예
Fig. 3. An example of control hazard.

Ⅲ. 타겟 아키텍춰

본 절에서는 SODAS-VP가 지원하는 연산과정의 수행 순서를 설명하고, 연산과정을 파이프라인 파이프라인 방식으로 구동하기 위한 타이밍, 스테이지의 설계 및 합성기의 타겟 아키텍춰에 대해 기술한다.

1. 클럭위상 (Clock Phase)의 결정

레지스터 전송 수준에서 수행되는 연산과정은 주로 레지스터를 통해 이루어진다. 예를 들어 a = b + c와

같은 연산과정이 있을 때 피연산자 b와 c의 적재와 연산, 그리고 연산 결과의 저장의 순서로 진행된다. 변수 b와 c가 레지스터 화일 내에 존재한다고 가정하면 이들 변수에 대한 디코딩이 수행되고 동시에 bus에 대한 pre-charge를 수행한 후 데이타를 랫치에 저장한다. 이러한 적재 작업은 보통 bus 또는 mux를 통해 이루어진다. Bus를 통한 데이타의 전달을 위해서는 TSB (Tri-State Buffer)의 구동에 대한 제어신호가 필요하고, mux의 경우에는 입력선택에 대한 제어신호가 필요하다. 다음 과정으로 연산이 수행되고, 연산 결과의 저장도 적재 작업과 동일하게 수행된다.

전술한 연산과정의 수행 순서를 만족시키고 파이프라인 방식으로 연산과정을 수행하기 위해 2-phase와 4-phase 클럭위상을 설계하였다. 그림 4에 각 클럭위상에서 수행되는 작업을 보였다. 2-phase 클럭을 사용할 경우, 레지스터는 ϕ1에서 저장 과정을, ϕ2에서 적재 과정을 수행한다. 연산모듈(FU)은 IF 방식을 사용할 경우 연산의 결과를 직접 적재하기 위하여 ϕ1에서 연산이 수행되고, 지연소자의 삽입 방법을 사용할 경우에는 ϕ1과 ϕ2에서 구동된다. 4-phase 클럭을 사용할 경우, 레지스터의 저장, 적재 과정이 ϕ2와 ϕ4에서 각각 수행되며, 연산모듈은 IF 방식의 경우 ϕ1에서 ϕ3내에서 구동되고, 지연소자 삽입의 경우 ϕ1에서 ϕ4내에서 구동된다. 래치, TSB, BUS, 그리고 MUX는 각각 적재와 저장과정에 해당되는 위상에서 구동된다. 설계된 각 클럭위상을 사용하여 합성과정을 수행할 때 고려해야할 사항으로써, 2-phase 클럭위상을 사용할 경우에는 적재와 저장 과정에서 bus의 충돌로 인한 structural hazard가 발생할 가능성이 있다. 이러한 경우 4 phase 클럭위상을 사용하면 읽기/쓰기 bus에 대한 pre-charge를 따로 수행하므로 structural hazard를 방지할 수 있다.

RD : Read decoding　　WD : Write decoding
S_w : MUX selection for write　　S_r : MUX selection for read

	2-phase 클럭위상		4-phase 클럭위상			
	ϕ1	ϕ2	ϕ1	ϕ2	ϕ3	ϕ4
RF, Reg.	RD, Write	WD, Read	WD	Write	RD	Read
FU	OP		Operation			
latch	enable	enable		enable		enable
TSB	enable	enable		enable		enable
BUS	PC, prop.	PC, prop.	PC	prop.	PC	prop.
MUX	S_w	S_r		S_w		S_r

그림 4. 각 클럭위상 내에서 수행되는 작업
Fig. 4. Tasks executed in each clocking scheme.

2. 타겟 아키텍춰

본 시스템이 지원하는 타겟 아키텍춰는 파이프라인을 지원하는 레지스터 전송 수준의 데이타패스이다. 레지스터 전송 수준의 데이타패스는 여러가지 모듈의 집합과 그들 사이의 통신경로로 표현된다. 하나의 디지탈 시스템은 데이타패스와 콘트롤러로 구성되고, 데이타패스는 기억소자, 연산모듈, 연결구조로 이루어진다.

기억 소자에는 레지스터와 메모리가 있으며, 래지스터는 임시 저장 기억장소로서 프로그램 수행 중의 데이타를 저장한다. 레지스터의 enable 단자는 콘트롤러에 연결되고, 콘트롤러는 특정 제어 구간에서 사용되는 레지스터에 대한 콘트롤 신호를 전송한다. 메모리는 데이타의 행렬을 저장하며 데이타의 입출력 포트와 주소 포트로 구성된다. 메모리의 enable 단자는 읽기/쓰기의 형태로 콘트롤러와 연결된다. 연산모듈은 기억 소자의 데이타를 이용하여 연산을 수행하는 단위로서 ALU, 곱셈기, 뺄셈기등이 여기에 해당된다. 연산 모듈에는 하나의 연산만을 수행하는 단순 연산 모듈 (Single Functional Unit)과 여러개의 연산을 수행하는 다중 연산 모듈 (Multiple Functional Unit)으로 구성되며, 다중 연산 모듈에는 연산의 선택을 위한 선택 단자가 콘트롤러에 연결된다. 연결 구조는 MUX와 BUS로서 구성되며, 이들은 기억 소자와 연산 모듈 사이의 데이타 전송을 위해 사용된다. MUX는 여러개의 입력 중 하나가 선택되어 하나의 출력으로 전송되고, MUX의 선택 단자는 콘트롤러에 연결된다. BUS는 여러개의 입력 중 하나가 선택되어 여러개의 출력으로 전송된다. BUS의 입력에는 TSB (Tri-State Buffer)가 연결되고 TSB의 선택 단자는 콘트롤러가 구동한다.

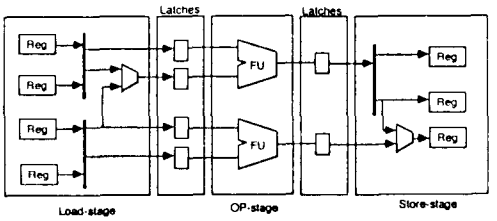


그림 5. RTL 파이프라인 데이타패스
Fig. 5. RTL pipelined datapath.

그림 5에 RTL 파이프라인 데이타패스를 보였다. RTL 데이타패스가 파이프라인을 지원하기 위해서는 연산모듈의 입출력단에 랫치의 삽입이 필요하다. 랫치가 삽입되면 파이프라인 스테이지는 적재단, 연산단, 저장단의 세 스테이지로 나뉘게 되고, 연산모듈

의 입출력 랫치를 경계로 스테이지가 구분된다. 적재단에서는 레지스터의 값을 연산모듈의 입력 랫치에 저장하는 기능을 하며, 레지스터와 랫치 사이의 데이타 전달은 bus 또는 mux와 같은 연결구조를 통하여 수행된다. 연산단에서는 입력 랫치의 데이타를 이용하여 연산을 수행한 후 결과를 출력 랫치에 전달하며 별도의 통신경로는 존재하지 않는다. 저장단에서는 연산모듈의 출력 랫치의 값을 레지스터에 저장하며, 적재단의 경우와 같이 연결구조가 존재한다.

Ⅳ. 상위수준 합성

1. 시스템 개관

본 논문에서 제시한 시스템 SODAS-VP의 전체 구성도를 그림 6에 보였다. 입력 동작기술 언어로는 1988년에 IEEE에 의해 이미 표준화되어 사용되고 있는 VHDL [9] 을 사용하며, 파싱과 동작변환 (behavioral transformation)을 통하여 콘트롤과 데이타 흐름의 정보를 가지는 C/DFG를 생성한다. 시뮬레이션과 합성기는 모두 C/DFG를 이용하여 수행되고, 합성과정의 수행을 위해 설계 제약조건, 라이브러리 등이 제공된다. 상위수준 합성이 완료되면 합성된 데이타패스를 구동하기 위한 콘트롤러의 합성이 필요하며, SODAS-VP는 FSM 및 마이크로 코드 형태의 콘트롤러를 지원한다.

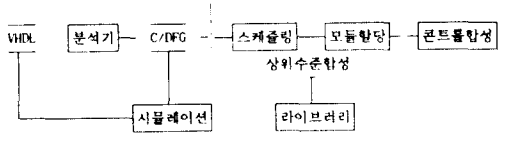


그림 6. 시스템의 전체 구성도
Fig. 6. Overall system flow.

2. 스케쥴링

스케쥴링은 연산을 특정 제어구간에 할당하는 과정으로 시간과 면적을 최소화시키는 스케쥴링이 요구된다. 같은 종류의 연산을 서로 다른 제어구간에 할당할 경우 입력을 muxing하므로써 연산기를 공유할 수 있으므로 면적을 줄일 수 있다. 간단한 스케쥴링 방법으로 ASAP, ALAP가 있고 [2] urgency, freedom, force 등 우선순위 함수에 따라 여러 알고리듬이 제안되었다. [3] [10]

데이타패스의 합성시 if, switch 문에서 발생하는 분기가 있을 경우 상호배제의 성질을 가지며, 그러한 경우에는 서로 같은 제어구간에 할당된 연산도 실제 한 방향의 연산과정이 수행되므로 공유가 가능하다. [11] 그리고 시스템 클럭이 결정되어 있을 경우 각 연산의 지연시간에 따라 하나의 제어구간에 할당되지 못할 때 여러구간에 걸쳐 연산을 수행하거나 (멀티싸이클링), 하나의 제어구간에 여러개의 연산이 연결되어 수행할 수도 있다 (체이닝). 스케쥴링 과정의 목적함수에 사용되는 파라미터로는 제어구간의 수, 지연시간, 사용되는 모듈의 갯수 등이 될 수 있으며, 기존의 시스템들은 면적과 시간 제약조건 하에서 목적함수를 정의하여 사용하고 있다. SODAS-VP 시스템에서는 HAL 시스템의 force-directed 스케쥴링에 연산을 수행하는 연산모듈의 면적을 고려하여 보다 효율적인 스케쥴링 알고리듬인 entropy-based 스케쥴링 [12] 을 사용하였다.

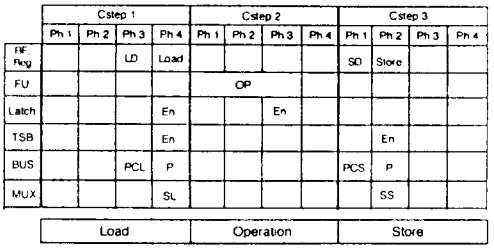
3. 파이프라인 스테이지의 구성 및 hazard의 처리

스케쥴링 과정이 완료되면 C/DFG상의 각 연산자가 수행되는 제어구간이 결정되고, 각 연산의 수행시간은 피연산자의 적재, 연산, 결과의 저장 시간의 합으로 주어진다. 그러나 파이프라인 데이타패스를 지원하기 위해 적재, 연산, 저장의 부작업이 스테이지로 분할되어 수행되며, 그림 7에 2-phase와 4-phase 클럭을 사용한 경우의 파이프라인 스테이지를 보였다.

파이프라인 스테이지는 크게 적재-스테이지, 연산-스테이지, 저장-스테이지로 구성된다. 적재-스테이지에서는 레지스터의 데이타가 연결구조를 통하여 연산기의 입력래치로 전달되고, 데이타의 전달을 위해 필요한 하드웨어가 클럭에 의해 구동된다. 연산-스테이지에서는 연산기의 입력래치에 저장된 데이타를 이용하여 연산을 수행하고 그 결과를 출력래치에 저장하는 과정이 수행된다. 연산-스테이지에서는 연결구조의 구동이 필요없으며, 연산기와 이에 연결된 입출력래치가 구동된다. 저장-스테이지에서는 연산기의 출력래치의 데이타를 연결구조를 통하여 레지스터에 저장하는 과정이 수행된다. 적재-스테이지의 경우와 같이 레지스터와 데이타의 전달을 위해 필요한 연결구조가 구동된다. 스케쥴링의 결과 생성되는 모든 연산은 그림 7의 스테이지로 분할되고, 모듈할당 과정에

	Cstep 1		Cstep 2		Cstep 3	
	phase 1	phase 2	phase 1	phase 2	phase 1	phase 2
RF_Reg	LD	Load		SD	Store	
FU			OP			
Latch		enable	enable			
TSB		enable			enable	
BUS	PCL	P		PCS	P	
MUX		SL			SS	
	Load		Operation		Store	

(a)



(b)

그림 7. 파이프라인 스테이지의 구성
(a) 2-phase 클럭 (b) 4-phase 클럭

Fig. 7. Construction of pipeline stage.
(a) With 2-phase clocking scheme.
(b) With 4-phase clocking scheme.

적합한 중간형태로 변환된다. 중간형태에 대한 자세한 내용은 다음절에서 다룬다.

전술하였듯이 파이프라인 방식으로 작업을 수행할 경우 hazard가 발생할 수 있다. SODAS-VP는 데이타 의존성에 의해 발생하는 data hazard를 처리하기 위해 IF와 지연소자 삽입 방식을 지원한다. 그림 8에 IF 방식을 사용할 경우의 파이프라인 스테이지를 보였다. I1의 연산-스테이지의 결과는 직접 I2의 적재-스테이지로 전달되고, 이를 위해 연산기의 입출력 래치 사이에 bypass wire가 추가된 데이타패스를 그림 8 (c)에 보였다. IF 방식을 사용할 경우의 클럭 주기는 그림 8 (b)의 Cstep 2에서 연산과 적재 시간의 합으로 주어진다.

그림 9에 지연소자 삽입 방식을 사용한 경우의 파이프라인 스테이지를 보였다. I1의 연산결과가 레지스터에 저장되는 Cstep 3까지 지연소자를 삽입하여 data hazard를 제거한다. 지연소자 삽입 방식을 사용할 경

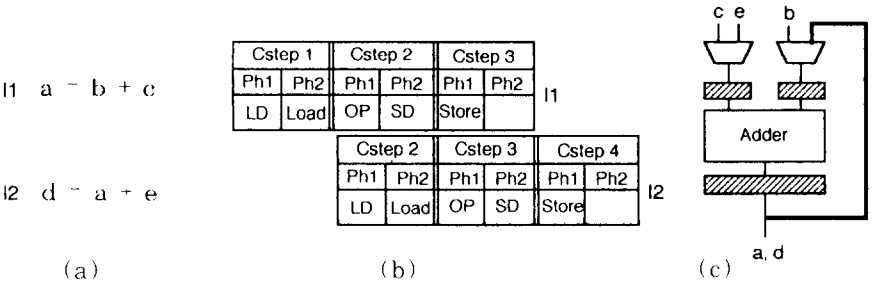


그림 8. Data hazard의 처리를 위한 IF 방식
(a) 데이타 의존성이 있는 연산의 예 (b) IF를 위한 파이프라인 스테이지
(c) IF를 위한 데이타패스

Fig. 8. Internal forwarding for handling data hazard.
(a) An example of operations with data dependency.
(b) Pipeline stages for internal forwarding.
(c) Datapath supporting internal forwarding.

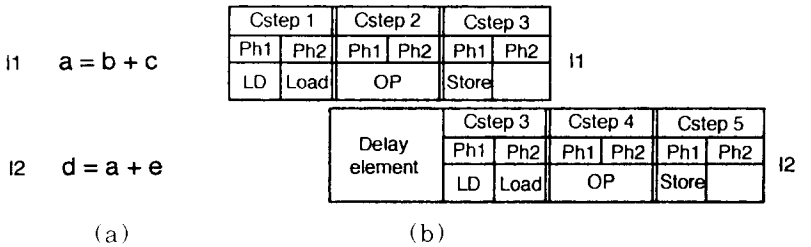
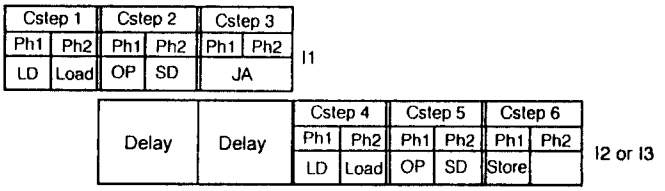


그림 9. Data hazard의 처리를 위한 지연삽입 방식
(a) 데이타 의존성이 있는 연산의 예 (b) 지연삽입을 위한 파이프라인 스테이지

Fig. 9. Delay insertion for handling data hazard.
(a) An example of operations with data dependency.
(b) Pipeline stages for delay insertion.

```
I1   If ( a > b )
I2      then    c = a - b;
I3      else    c = b - a;
```

(a)



(b)

그림 10. Control hazard의 처리
(a) Control hazard가 있는 연산의 예
(b) 지연삽입을 이용한 파이프라인 스테이지

Fig. 10. Handling of control hazard.
(a) An example of operations with control hazards.
(b) Pipeline stages with delay insertion.

우의 클럭 주기는 Cstep 2의 연산시간과 Cstep 3의 적재와 저장 시간의 합 중의 최대값으로 주어진다.

Control hazard는 제어의 분기가 발생하는 조건문 또는 반복문에 의하여 발생한다. SODAS-VP 시스템은 control hazard의 처리를 위하여 지연소자 삽입 방식을 지원한다. 그림 10에 control hazard의 예와 이에 대한 파이프라인 스테이지를 보였다. If 문에 의해 조건 분기가 발생하게 되고, 다음에 수행되는 연산이 결정되는 Cstep 3까지 지연소자를 삽입하여 control hazard를 처리한다.

4. 중간형태

스케쥴링 과정을 통하여 연산자의 제어구간이 결정되고, 파이프라인 스테이지의 구성으로 연산과정의 분할이 완료되면 SAIP (Scheduling Allocation Intermediate- format for Pipeline)의 중간형태로 모듈할당 과정의 입력으로 주어진다. SAIP에서 제어구간은 하나의 스테이지에 해당하며 각 스테이지는 적재, 연산, 저장의 세가지 타입이 있다.

SAIP는 기존의 SAIF[11]를 스테이지로 분할한 형태로서 표 1의 명령어로 표현되고, 전체 구성은 크게 선언부와 동작부로 이루어진다. 선언부에서는 회로 동작의 수행에 필요한 변수 또는 포트가 선언되며 표

표 1. SAIP 명령어
Table 1. SAIP instructions.

연산명령어	add sub mult div and or pmul mdiv and or ge le equ ne gt lt
적재명령어	lod loc
저장명령어	str

1의 SAIP 명령어로 구성된 수행순서가 동작부에 기술된다. 적재 명령은 lod (load data)와 loc (load constant)의 두 종류가 있고 각각 변수와 상수를 랫치에 적재하는 기능을 수행한다. SAIP에서 사용되는 변수, 상수, 포트는 선언부에 기술되고 랫치의 고유번호는 랫치가 연결된 연산자 번호와 연산자의 입출

```
lod a, l#(2.1)      add o#(2)      str l#(4.3), a
loc 5, l#(3.2)      pmul o#(3.2.1)
lod l#(4.3),l#(5.1)
```

(a) 적재 명령 (b) 연산 명령 (c) 저장 명령

그림 11. SAIP 명령의 예
Fig. 11. Examples of SAIP instructions.

력 포트 번호로 구성되고 그림 11-(a)에 예를 보였다. lod 명령은 변수 a를 연산 2의 입력 포트 1의 랫치에 적재하고, loc 명령은 상수 5를 연산 3의 입력 포트 2의 랫치에 적재하는 기능을 한다. IF를 지원하기 위해 랫치간의 적재 예도 보였다. 4번 연산자의 출력 랫치의 값을 5번 연산자의 입력포트 1의 랫치에 적재한다.

연산 명령은 크게 operand가 하나인 단일 연산과 둘이 필요한 이진 연산의 두종류로 구분되며, 적재 명령의 다음 스테이지에서 수행된다. 연산 명령은 연산의 타입과 연산자 번호로 구성되고 그림 11-(b)에 연산명령의 예를 보였다. 파이프라인 연산의 경우는 스테이지의 전체 갯수와 수행되는 스테이지의 번호가 추가된다. pmul o#(3.2.1)은 연산자의 고유번호가 3이고 2 스테이지 파이프라인 곱셈기의 첫번째 스테이지를 수행함을 의미한다. 저장 명령으로는 str 명령이 사용되며 연산자의 출력 랫치의 값을 레지스터에 저장하는 기능을 한다. 그림 11-(c)의 명령은 4번 연산자의 출력 랫치의 값을 변수 a에 저장한다.

5. 모듈할당

모듈할당은 스케쥴링의 결과 생성된 명령어의 집합인 SAIP를 수행하기 위한 파이프라인 데이타패스를 구성하는 과정으로 주어진 동작기술을 정확히 수행하면서 사용되는 하드웨어의 수를 최소화하는데 목적이 있다. RTL 하드웨어에는 기본적으로 연산모듈, 메모리 소자, 연결구조가 있으며, 모듈할당 과정에서 변수는 메모리 또는 레지스터에, 연산자는 연산모듈에 할당되고 bus와 multiplexor를 통하여 연결구조가 결정된다. SODAS-VP 시스템은 similarity measure를 사용하여 모듈할당 과정을 수행한다.

(1) Similarity measure의 정의

Similarity measure(이하 SM)는 두 연산자를 공유함에 있어서 효율성을 나타내는 척도이다. 예를 들어, 곱셈기와 덧셈기의 경우에는 두 연산자의 수행시간 및 면적의 차이가 크고 하드웨어의 구조도 다르기 때문에 하나의 연산모듈로 할당되기에 어려움이 많다. 반면 덧셈기와 뺄셈기의 경우에는 수행시간이나 면적의 차이가 크지 않고, 하드웨어의 구조 자체도 유사하여 하나의 연산모듈로 공유가 쉽다. Similarity measure 공유가능한 두 연산자 사이에 존재하며, 그림 12에 SM의 정의를 보였다.

SM은 크게 6가지의 경우로 정의된다. SM이 6인 경우는 두 연산자가 단일 연산자에 의해 구현될 경우로서 시간과 면적측면에서 가장 효율적이다. 두 연산자의 종류가 다른 경우에 대하여 SM은 5에서 1 사이의 값을 가지며 DR과 AR에 의하여 결정된다. SM은 레지스터의 할당과 연산모듈의 할당 과정에서 우선순위 함수로서 사용된다.

(2) 레지스터의 할당

레지스터의 할당 과정에서는 연결구조를 최소화하면서 레지스터의 수를 줄이는 것이 목적이다. 레지스터의 할당 과정에서 필요한 첫번째 단계는 생존구간 분석 (lifetime analysis)을 통한 공유가능 변수의

공유가능한 두 연산자 OP_i, OP_j가 주어졌을 때,
(단, $i \neq j$이고 $1 \leq i,j \leq N$; N은 전체 연산자의 수)

$$DR\ (Dealy\ Ratio) = \frac{|\ OP_i.dealy - OP_j.delay\ |}{\max(OP_i.dealy,\ OP_j.delay)}$$

$$AR\ (Area\ Ratio) = \frac{|\ OP_i.area - OP_j.area\ |}{\max(OP_i.area\ ,\ OP_j.area)}$$

$SM = 6$: $OP_i.type = OP_j.type$
$SM = 5$: $0.0 < DR\ ,\ AR < 0.2$ 단, $OP_i.type \neq OP_j.type$
$SM = 4$: $0.2 \leq DR\ ,\ AR < 0.4$ 단, $OP_i.type \neq OP_j.type$
$SM = 3$: $0.4 \leq DR\ ,\ AR < 0.6$ 단, $OP_i.type \neq OP_j.type$
$SM = 2$: $0.6 \leq DR\ ,\ AR < 0.8$ 단, $OP_i.type \neq OP_j.type$
$SM = 1$: $0.8 \leq DR\ ,\ AR < 1.0$ 단, $OP_i.type \neq OP_j.type$

그림 12. Similarity measure의 정의
Fig. 12. Definition for similarity measure.

추출이다. 변수의 생존구간은 live와 dead의 두가지 상태로 표현되며, 변수의 live 구간은 그 변수가 처음으로 정의된 제어구간에서 마지막으로 사용된 제어구간까지의 구간이고, dead 구간은 마지막으로 사용된 제어구간에서 다음으로 정의되는 제어구간까지의 구간이다.[13] 두 변수가 공유할 수 있는 조건은 전체 제어구간에 대하여 두 개의 변수의 생존구간이 중첩되는 구간이 없어야한다는 것이다.

비파이프라인 데이타패스에서는 피연산자의 적재와 연산, 결과의 저장이 하나의 제어구간에서 수행되므로 정확한 생존구간 분석을 수행할 수 없다. 파이프라인 구조의 데이타패스에서는 피연산자의 적재, 연산, 저장이 각각의 스테이지에서 수행되므로, 생존구간 분석을 스테이지 단위로 조사해야 한다. 데이타 의존성에 의한 data hazard가 발생할 경우, IF 방식으로 처리함을 가정할 때 생존구간 분석의 예를 그림 13에 보였다. 그림 13-(a)는 비파이프라인 방식의 경우로서 ST1의 저장변수 a가 ST2의 피연산자 a에, ST2의 저장변수 d가 ST3의 피연산자 d에 데이타 의존성을 가지고 있다. 그림 13-(b)는 그림 13-(a)에 대한 파이프라인 스테이지를 보인 것으로서 이러한 데이타 의존성은 SAIP에서 IF 방식으로 처리되고, 그림 13-(b)에서 ST2와 ST3의 첫번째 적재 명령으로 표현된다. ST2의 IF 적재 명령은 연산자 $+_1$의 출력 랫치의 값을 직접 연산자 $-_2$의 입력 랫치에 전달하는 기능을 하며, ST3의 IF 적재명령은 연산자 -의 출력랫치의 값을 연산자 +의 입력랫치에 직접 전달한다.

비파이프라인과 파이프라인 방식의 생존구간을 비교할 때, 비파이프라인에 비하여 파이프라인 방식의 경우가 공유가능성이 높다. 예로서 변수 a와 b의 경우 비파이프라인 방식에서는 ST2에서 공유가 불가능하지만, 파이프라인 방식에서는 ST2의 변수 a의 적재를 래치를 통한 IF 방식으로 처리하므로 공유가 가능하다. 이러한 공유 가능성은 생존구간 분석을 통해 CG (Compatibility Graph)로 표현된다. 레지스터의 할당 과정에서 CG의 노드는 변수를 나타내고, 노드간에 존재하는 간선은 공유가능성을 나타낸다. 전체 노드를 탐색하여 공유가 가능한 노드 사이에 간선을 연결하여 CG를 구성한다.

레지스터 할당 과정에서 사용되는 우선순위를 그림 14에 보였다. 우선순위는 SM을 참조하여 CG 상의 간선에 주어지며, 7 가지의 경우로 분류된다. 가장 높은 우선순위는 할당문(assign statement)에서 변수쌍이 각각 source와 destination으로 사용되는 경우로서, 공유시 레지스터 간의 데이타 전송과 연결선이 필요없다. 할당문이 아닌 경우에는 각 변수를 피연산자로 가지는 연산자를 모두 탐색하여 SM의 최대값을 우선순위로 가지며, 1 ~ 6 사이의 값을 가진다. 레지스터의 할당 과정에서 SM을 고려하는 이유

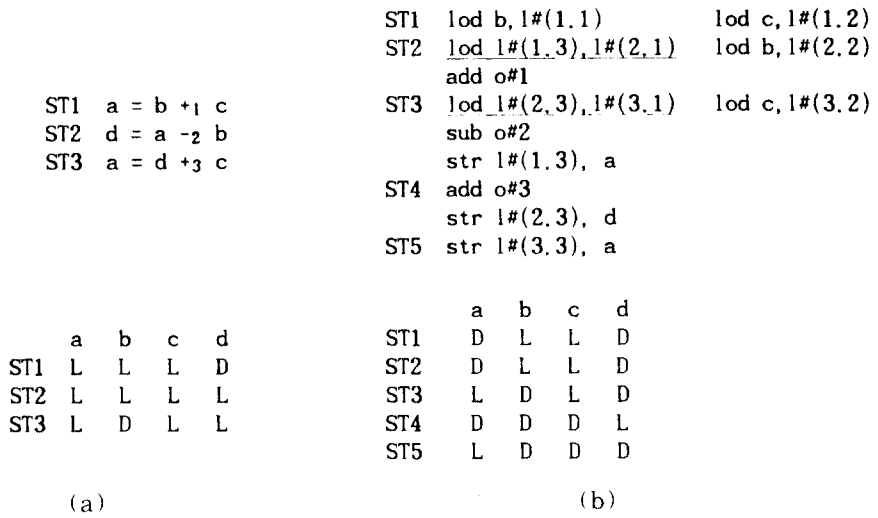


그림 13. 생존구간 분석의 예
(a) 비파이프라인 방식 (b) 파이프라인 방식

Fig. 13. An example of lifetime analysis
(a) Non-pipeline method. (b) Pipeline method.

CG = (V, E) 이고 변수 V_i와 V_j가 주어졌을 때, (V : 노드의 집합, E : 간선의 집합)

V.op = 변수 V를 피연산자로 가지는 연산자
V.op.step = 변수 V를 피연산자로 가지는 연산자가 수행되는 제어구간
V.N = 변수 V를 피연산자로 가지는 연산자의 갯수

$$E_{ij}.weight = \begin{cases} 7, & V_i(V_j)\text{가 source 이고 } V_j(V_i)\text{가 destination인 경우 } (V_i\text{와 } V_j\text{는 할당문의 변수쌍}) \\ Max[SM(V_i.op, V_j.op)] & \text{for } 1 \le i \le V_i.N,\ 1 \le j \le V_j.N,\ \text{otherwise} \\ & (\text{단, } V_i.op.step \ne V_j.op.step) \end{cases}$$

그림 14. 레지스터 할당의 우선순위
Fig. 14. Priority function in register allocation.

```
Operator_Allocation ()
{
    Insert latches into input/output ports of an operator;
    Extract compatibility graph using function ExtractCG();
    Calculate priority function for each edge of CG;
    Perform weight-driven clique partitioning for allocation;
}

ExtractCG()
{
    for (i = 1 to N - 1) { /* N : 전체 제어구간의 수 */
        if (Instr[i].type == LOAD_STAGE || STORE_STAGE)  continue;
        for (j = i + 1 to N) {
            if (Instr[j].type == LOAD_STAGE || STORE_STAGE)  continue;
            if (no conflicts btn i-th and j-th instruction)
                construct an edge in CG btn node i and j;
        }
    }
}
```

그림 15. 연산모듈 할당 과정
Fig. 15. Functional unit allocation process.

는 연산모듈의 할당 과정에서 SM이 큰 연산자 쌍에 우선순위를 두므로 이를 레지스터 할당 과정에서 미리 고려하므로써 연결구조를 최소화하기 때문이다. CG의 우선순위가 결정되면 weight-driven clique partitioning 알고리듬[14]을 적용하여 레지스터의 할당이 완료된다.

(3) 연산모듈의 할당

변수에 대한 레지스터의 할당이 완료되면 다음 과정으로 연산자의 할당을 수행한다. 연산모듈의 할당은 제어구간의 조사를 통해 수행되므로 비파이프라인 방식과 동일하며, 파이프라인 스테이지의 구성을 위해 연산자의 입출력단에 랫치의 삽입 과정이 필요하다. 연산모듈 할당은 SM을 우선순위로 하여 수행하며, 전체 수행 과정을 그림 15에 나타냈다. 할당 과정은 크게 각 연산자의 입출력단에 랫치 삽입, 공유가능 그래프인 CG의 추출, 우선순위의 계산, 클리크 분할 알고리듬의 적용의 네 단계로 나뉜다. 랫치의 삽입은 연산-스테이지와 적재/저장-스테이지의 분할을 위해 수행되고, CG의 추출은 SAIP 상의 명령어 집합을 탐색하면서 적재-스테이지와 저장-스테이지를 제외한 연산-스테이지의 제어구간을 조사하여 CG를 구성한다.

연산모듈의 할당 과정에서 사용되는 우선순위 함수를 그림 16에 보였다. 우선순위는 SM을 참조하여 연결구조 비용을 최소화시키도록 결정하였다. CG 상의 노드는 하나의 연산자를 의미하고, 공유가능한 노드 사이의 간선에 우선순위가 주어진다. 두 연산자가 하나의 연산모듈을 공유할 때 부가적으로 추가되는 연결구조 비용은 I_{ij}로 표현된다. 연산자의 입출력 포트에 연결된 레지스터를 탐색하여 공통적으로 사용되는 레지스터의 갯수가 많을수록 연결구조 비용(MUX의 수, 연결선의 수)을 줄일 수 있다. 따라서 I_{ij}의 값이

CG = (V, E) 이고 V_i와 V_j가 주어졌을 때, (V : 노드의 집합, E : 간선의 집합)

$V.pid$ = 연산자 V의 입출력 포트 ID
$V.pid.reg$ = 연산자 V의 해당 포트 pid에 연결된 레지스터
I_{ij} = 연산자 V_i와 V_j에 공통으로 연결된 레지스터의 갯수 = $\sum_{\substack{M \in V_i.pid \\ N \in V_j.pid}} X_{MN}$

$$\text{where, } X_{MN} = \begin{cases} 1, & M.reg = N.reg \quad (M, N\text{은 입출력 포트 ID}) \\ 0, & \text{otherwise} \end{cases}$$

$$W_{ij} = \alpha \cdot SM(V_i, V_j) + \beta \cdot I_{ij} \quad (\text{단, } i \neq j)$$

그림 16. 연산모듈 할당의 우선순위
Fig. 16. Priority function in operator allocation.

클수록 연결구조의 비용은 감소하며, 우선순위는 SM과 I_{ij}의 합으로 주어진다. CG에 대한 우선순위가 결정되면 weight-driven clique partitioning을 적용하여 연산모듈 할당 과정이 완료된다.

(4) 연결구조의 할당

레지스터와 연산모듈의 할당이 완료되면 데이타패스 합성의 마지막 과정으로 연결구조의 할당 과정이 수행된다. 연결구조는 레지스터와 연산모듈 간의 상호통신을 위하여 bus 및 mux를 통하여 합성되고, SAIP의 적재/저장 스테이지에 대하여 수행된다. 전체적인 연결구조의 할당 과정은 통신경로의 추출, CG의 추출, weight-driven clique partitioning 알고리듬 적용의 순서로 진행된다.

통신경로의 추출은 레지스터와 연산모듈의 합성 결과를 참조하여 데이타의 전달을 위해 필요한 연결선을 추출하는 과정이다. 통신경로는 크게 적재경로와 저장경로의 두가지의 형태가 있으며, 합성 결과와 SAIP를 탐색하여 추출한다. 적재경로는 SAIP의 적재명령에서 레지스터의 출력과 연산자의 입력을 연결하고, 저장경로는 연산자의 출력과 레지스터의 입력을 연결한다. 이들 두 경로외에 IF를 처리하기 위한 IF 경로가 존재할 수 있다. IF 경로는 연산자의 출력이 직접 연산자의 입력으로 연결되는 경로로서 데이타 의존성이 발생할 때 존재하고 적재경로에 포함된다. 통신경로의 추출이 완료되면 적재/저장-스테이지에 대한 자원충돌 여부를 조사하여 공유가능 그래프인 CG를 구성한다.

CG를 구성할 때, 2 phase 클럭을 사용할 경우에는 적재경로와 저장경로간에 자원충돌이 발생할 경우가 있다. 그림 17에 2 phase와 4 phase 클럭을 사용할 경우 bus의 동작을 보였다. 2 phase의 경우에는 phase 1에서 버스에 대한 pre-charge를 수행하고 phase 2에서 데이타를 전달하므로 적재와 저장 과정에서 자원충돌이 발생한다. 이를 처리하기 위해서는 적재버스와 저장버스를 독립적으로 합성해야 하는데, 이는 면적 측면에서 큰 부담으로 작용한다. 반면 4 phase 클럭을 사용하면 버스의 pre-charge와 데이타의 전달을 적재와 저장에 대해 각각 수행할 수 있으므로 자원충돌이 발생하지 않고 면적측면에서도 유리하다.

	2 phase		4 phase			
	$\phi 1$	$\phi 2$	$\phi 1$	$\phi 2$	$\phi 3$	$\phi 4$
bus	pre-charge	전달	pre-charge	전달	pre-charge	전달

그림 17. 2 phase와 4 phase 클럭의 비교
Fig. 17. Comparison of 2 phase and 4 phase clock.

CG = (V, E) 이고 V_i와 V_j가 주어졌을 때, (V : 노드의 집합, E : 간선의 집합)

$V.src$ = 연결선 V의 source
$V.dst$ = 연결선 V의 destination

$$E_{ij} = \begin{cases} 3, & V_i.src \neq V_j.src \\ 2, & V_i.dst \neq V_j.dst \\ 1, & \text{otherwise} \end{cases}$$

그림 18. 연결구조 할당 과정의 우선순위
Fig. 18. Priority in interconnection allocation.

연결구조의 할당 과정에서 사용되는 우선순위를 그림 18에 보였다. CG 상의 노드는 하나의 연결선을 나타낸다. 우선순위가 가장 큰 경우는 연결선 쌍의 source가 같은 경우로서, 동일한 제어구간에서 사용되어도 자원충돌이 발생하지 않는다. 우선순위가 2인 경우는 연결선 쌍의 destination이 같은 경우이고, source와 destination이 모두 다른 경우의 우선순위는 1로서 최소값을 가진다. 우선순위가 결정된 CG에 weight-driven clique partitioning 알고리듬을 적용하면 BUS 구조의 통신경로가 합성되고, 레지스터와 연산모듈의 입력 포트를 조사하여 다중입력이 존재할 경우 MUX를 삽입하므로써 데이타패스의 연결구조가 완료된다.

6. 콘트롤러의 합성

합성된 데이타패스는 RTL 구성요소와 이들 사이의 연결구조로 표현된다. 데이타패스는 단순히 구조적인 기술로 표현되므로 이를 구동시키기 위한 순차적인 콘트롤러가 필요하다. SODAS-VP 시스템에서는 FSM 형태 및 마이크로코드 형태의 콘트롤러가 지원된다.

상위수준 합성 과정이 완료되면 RTL 데이타패스의 구조가 추출된다. RTL 데이타패스에서 콘트롤이 필요한 하드웨어로는 레지스터, 연산모듈 (다중연산모듈인 경우에는 선택단자가 필요), MUX, TSB 등이 있다. 각 제어구간에서 구동되는 하드웨어의 추출이 이루어지면 AC (Active Component) 테이블의 형태로 콘트롤러 합성기의 입력으로 주어진다. AC 테이블은 각 제어구간에서 구동되는 RTL 하드웨어의 정보를 표현한 것으로, 각 하드웨어의 구동에 필요한 제어신호의 bit 수, bit pattern 등의 정보를 가진다. 제어신호는 4 클럭위상의 해당 phase와 AND된 형태로 데이타패스에 직접 연결된다. 그림 19에 콘트롤러의 합성과정을 보였다.

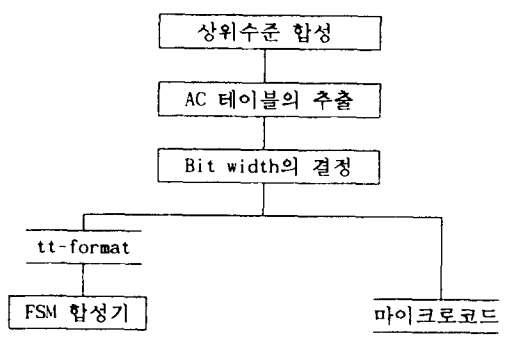


그림 19. 콘트롤러의 합성 과정
Fig. 19. Controller synthesis process.

콘트롤러의 bit width는 상위수준 합성 결과 생성된 데이타패스 상의 RTL 하드웨어를 탐색하므로써 결정되고, 아래와 같이 주어진다. 여기서 R_F는 합성된 다중 연산모듈의 연산 갯수이고 M_i는 MUX의 입력 갯수이다.

SR_i (Signal for Register) =
1 : i-th레지스터의 구동에 필요한 제어신호의 수
SF_i (Signal for FU) = $\lceil \log_2 R_F \rceil$:
i-th FU의 연산 선택에 필요한 제어신호의 수
SM_i (Signal for Mux) = $\lceil \log_2 M_i \rceil$:
i-th mux의 입력선택을 위한 제어신호의 수
ST_i (Signal for TSB) = 1 :
i-th TSB의 구동에 필요한 제어신호의 수

$$\text{bit width} = \Sigma SR_i + \Sigma SF_i + \Sigma SM_i + \Sigma ST_i$$

FSM 콘트롤러의 합성은 tt-format [15]을 입력으로 하여 FSM 생성기에 의해 최적화된 회로로 구현된다. FSM 생성기는 콘트롤러에 대한 상태표로부터 입력, 출력, 현재상태, 다음상태 등의 정보를 통하여 FSM 합성을 수행한다. FSM 콘트롤러를 합성하기 위해서는 먼저 AC 테이블로 부터 콘트롤러의 bit width를 결정하고 tt-format을 생성해야 한다. TT-format은 입출력 핀의 갯수와 하나의 상태에 대한 출력신호를 정의한 상태천이표로서 출력신호는 데이타패스 상에 존재하는 각 RTL 하드웨어의 enable 핀과 다중 연산모듈의 경우에는 수행연산의 선택을 위하여, 그리고 mux의 경우에는 입력의 선택을 위한 제어신호가 출력된다.

마이크로코드 콘트롤러의 합성 과정은 tt-format의 추출 과정과 유사하다. AC 테이블을 통해 데이타패스의 RTL 하드웨어 전체의 bit width를 결정하고, 각 제어구간에서 구동되는 하드웨어의 출력신호를 저장하게 된다. 콘트롤러에서 출력하는 제어신호는 수평 병렬 (horizontal parallel) 신호로서 데이타패스에 디코딩이 필요없이 직접연결된다.

V. 실험결과

SODAS-VP 시스템은 Sun 워크스테이션의 UNIX 프로그래밍 환경 하에서 C 언어로 구현되었다. 시스템의 성능 평가를 위해 상위수준 합성 분야에서 벤치마크 입력으로 사용되는 5차 엘립틱 웨이브 필터 (EWF)에 대한 합성결과를 HAL 시스템 [3] 및 ALPS 시스템 [17]과 비교하여 표 2에 보였고,

표 2. EWF에 대한 합성 결과의 비교
Table 2. Comparison of synthesis results for EWF.

Latency를 17로 한 경우

	#regs.	#FUs	#MIs	#BUSes	#latches	#CSs	#PCSs	P(ns)	T(ns)	Δ_T(%)	Δ_A(%)
HAL	12+ROM	3	-	6	-	19	-	80	1520	-	n/a
ALPS	n/a	3	n/a	n/a	n/a	26*	-	80	1360	-10.5	n/a
SODAS-VP_I	16	3	14	5	12	19	21	60	1260	-17.1	+8.3
SODAS-VP_D	17	3	11	5	12	19	32	40	1320	-13.2	+8.1

MIs : MUX 입력의 갯수 CS : 제어구간의 수 PCS : 분할된 제어구간의 수
$\triangle_T$(%): HAL에 비교한 수행시간의 감소분 $\triangle_A$(%): 랫치의 증가로 인한 면적의 증가분

SODAS-VP에 의해 생성된 파이프라인 데이타패스를 그림 20에 보였다. 이 합성결과는 40 ns의 스테이지 지연시간을 가지는 2 단 파이프라인 곱셈기를 사용하고, 자원 제약조건으로 2개의 덧셈기와 1개의 곱셈기를 사용한 경우이다. 적재-스테이지와 저장-스테이지의 지연시간이 모두 20 ns로 주어졌을 때, HAL과 ALPS의 클럭주기는 20 + 40 + 20 = 80 ns 이다. SODAS-VP의 클럭주기는 IF 방식을 사용할 경우 연산모듈의 주기와 적재 지연시간의 합인 40 + 20 = 60 ns로 주어지고, 지연삽입 방식을 사용할 경우에는 적재와 저장 지연시간의 합과 연산 지연시간의 최대값인 40 ns로 결정된다.

면적 측면에서 결과를 비교할 때, 연산모듈의 갯수는 3개로 모든 시스템이 동일한 결과를 보이고, SODAS-VP의 경우 파이프라인 스테이지의 구성을 위해 12개의 랫치가 추가되었다. 추가된 랫치로 인한 전체면적의 증가분은 IF 방식의 경우 8.3 %, 지연소자 삽입 방식의 경우 8.1 %의 증가를 보였다. 면적 데이타는 게이트 수를 단위로 하여 산출하였고, 데이타의 bit 수는 32 bit이다 (덧셈기:594, 곱셈기: 16042, 2-input MUX: 256, 레지스터: 288, 래치:160). 그러나, 면적의 증가분은 배치/배선 과정을 수행한 후에 알 수 있는 BUS 및 wire의 면적은 고려하지 않았으므로, 이들 면적을 고려할 경우 면적 증가분은 더 감소할 수 있을 것으로 보인다. HAL 시스템의 경우 필터의 계수를 ROM에 저장하여 합성한다. 그림 20은 지연삽입 사용하여 합성한 데이타패스로서 계수로 사용되는 R1~R6을 제외한 나머지 레지스터의 갯수는 11개이며, IF 방식을 사용할 경우 10 개로 합성되어 HAL 시스템보다 유리한 결과를 보인다. 연결구조 측면에서는 BUS의 갯수는 HAL이 6 개를 사용하고, SODAS-VP가 IF와 지연소자 삽입 방식으로 합성한 경우 모두 5개의 결과를 보인다. IF 방식으로 합성할 때 추가되는 bypass wire의 영향으로 MUX 입력의 갯수가 증가함을 알 수 있다. 생성된 데이타패스의 전체 수행시간을 비교할 때, 비파이프라인 합성기인 HAL의 경우가 1520 ns로서 가장 큰 결과를 보였다. 파이프라인 합성기인 ALPS 시스템은 2단 파이프라인 곱셈기를 사용할 경우 latency를 EWF의 임계경로인 17로 주었을 때 1360 ns로서 HAL에 비하여 10.5% 감소된 수행속도를 보인다. SODAS-VP가 IF 방식을 사용하여 합성한

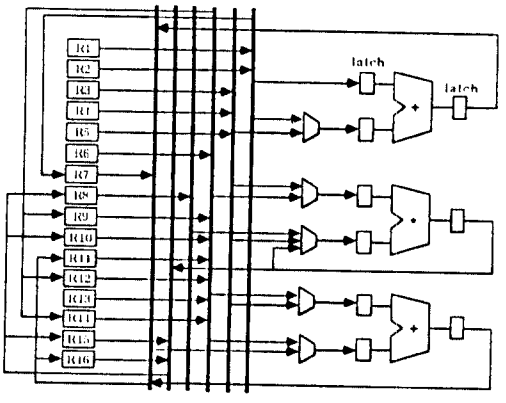


그림 20. SODAS-VP가 생성한 EWF의 데이타패스
Fig. 20. Datapath for EWF generated by SODAS-VP.

데이타패스의 수행속도는 1260 ns로서 HAL에 비하여 17.1%가 감소된 가장 좋은 결과를 보이며, 지연소자 삽입 방식을 사용하여 합성한 데이타패스의 경우 13.2% 감소된 결과를 보였다. 데이타 의존성이 많은 EWF의 경우 지연소자의 삽입 방식보다 IF 방식이 더 유리함을 알 수 있다.

Ⅵ. 결론

본 논문은 파이프라인 방식을 지원하는 하드웨어를 합성하기 위한 상위수준 합성 과정에 대해 기술하였다. 스케줄링의 결과 생성된 각 제어구간의 연산과정은 적재, 연산, 저장의 세 스테이지로 분할되어 SAIP의 형태로서 모듈할당 과정의 입력으로 주어진다. 스테이지의 분할 과정에서는 파이프라인에서 발생할 수 있는 hazard를 고려하였고, 분할된 연산과정의 수행과 hazard의 처리를 위한 클럭위상을 설계하였다. 파이프라인을 지원하기 위해 모듈할당 과정에서 첨가된 작업으로, 레지스터의 할당 과정에서 IF를 고려하여 공유가능한 변수의 집합을 증가시키므로써 전체 레지스터의 수를 줄일 수 있었다. 연간모듈의 할당에서는 랫치의 삽입 과정이 첨가되었고, 연결구조의 할당 과정에서는 IF의 처리를 위한 bypass wire의 추가 과정이 첨가되었다. 모듈할당 과정은 similarity measure를 우선순위로 사용하고, 각 할당 과정에서 전체적인 연결구조를 최소화시키는 방향으로 우선순위 함수를 설정하여 합성 과정을 수행하였다.

수행시간 측면에서 실험결과를 비교할 때, 본 연구에서 설계구현한 시스템 SODAS-VP가 생성하는 하드웨어에서의 수행속도가 HAL 시스템에 의해 생성된 하드웨어에 비해 17.1% (IF 방식), 13.2% (지연소자 삽입 방식)의 속도향상을 보였다. ALPS 시스템과의 비교에서는 IF 방식을 사용한 SODAS-VP 시스템이 ALPS보다 7.4%의 속도향상을 보였다. 이는 하나의 명령을 파이프라인 방식으로 수행하여 얻은 결과로서 제어구간의 수는 증가하지만 클럭주기의 감소로 전체 수행시간은 향상됨을 알 수 있다. 면적 측면에서는 스테이지의 분할을 위해 랫치가 부가적으로 첨가되었지만, 연산모듈과 연결구조는 타시스템과 거의 비교할 만한 결과가 나왔고 레지스터의 갯수 측면에서는 SODAS-VP가 더 유리한 결과를 생성하였다.

參 考 文 獻

[1] Manolis G. H. Katevenis, Reduced Instruction Set Computer Architecture for VLSI, The MIT Press, 1984, pp. 108-112.

[2] A. C. Parker, M. C. McFarland, R. Camposano, "Tutorial on High-Level Synthesis," in Proc. ACM/IEEE Design Automation Conference, June 1988, pp. 330-336.

[3] P. G. Paulin, "HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis," in Proc. ACM/IEEE Design Automation Conference, June 1986, pp. 263-270.

[4] G. De Micheli, D. C. Ku, "HERCULES: A System for High-Level Synthesis," in Proc. ACM/IEEE Design Automation Conference, June 1988, pp. 483-488.

[5] N. Park, "Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications," *IEEE Trans. CAD*, vol. 7, no. 3, March 1988, pp. 356-370.

[6] K. S. Hwang, "Scheduling and Hardware Sharing in Pipelined Data Paths," in Proc. ICCAD, Nov. 1989, pp. 24-27.

[7] F. Catthoor, H. J. De Man, "Application Specific Architectural Methodologies for High Throughput Digital Signal and Image Processing," *IEEE Trans. ASSP.*, vol. 38, no. 2, Feb. 1990, pp. 339-349.

[8] J. Hennessy, D. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann Publishers Inc., 1990, pp. 257-284.

[9] "IEEE Standard VHDL Language Reference Manual," IEEE Std 1076-1987, IEEE, N. Y., March 1988.

[10] P. G. Paulin, "Force Directed Scheduling for the Behavioral Synthesis of ASIC's," *IEEE Trans. CAD*, vol. 8, no. 6, June 1989, pp. 661-679.

[11] H. D. Lee, H. S. Jun, S. Y. Hwang, "Datapath Synthesis in Sogang Silicon Compiler," in Proc. JTC/CSCC, Dec. 1990, pp. 430-435.

[12] 전홍신, 황선영, "엔트로피에 바탕을 둔 새로운 스케쥴링 알고리듬," 한국정보과학회 봄 학술발표 논문집, vol. 18, no. 1, 1991년 4월, pp. 427-430.
[13] A. V. Aho, R. Sethi, J. D. Ullman, Compilers, Addison Wesley Pub. : Reading, Mass., 1988, pp. 320-329.
[14] C. Tseng, D. Siewiorek, "Automated Synthesis of Data Paths in Digital Systems." *IEEE Trans. CAD*, vol. 5, no. 3, July 1986, pp. 379-395.
[15] R. K. Brayton, G. D. Hachtel, C. T. McMullen, A. L. Sangiovanni - Vincentelli, "Logic Minimization Algorithm for VLSI Synthesis," Kluwer Academic Pub. : Reading, Mass., 1984, pp. 5-14.
[16] B. M. Pangrle, "Splicer: A Heuristic Approach to Connectivity Binding," in Proc. ACM/IEEE Design Automation Conference, June 1988, pp. 536-541.
[17] C. T. Hwang, J. H. Lee, Y. C. Hsu, "A Formal Approach to the Scheduling Problem in High Level Synthesis," *IEEE Trans. CAD*, vol. 10, no. 4, April 1991, pp. 464-475.

著者紹介

李 海 東(正會員)

1990年 2月 서강대학교 전자공학과 졸업. 1992年 2月 서강대학교 전자공학과 공학석사 취득. 1992年 2月 ~ 현재 서강대학교 전자공학과 박사과정 재학중. 주관심 분야는 High-level Synthesis, Design for Testability, VLSI 시스템 등임.

黃 善 泳(正會員)

1976年 2月 서울대학교 전자공학과 졸업. 1978年 2月 한국 과학원 전기 및 전자 공학과 공학 석사 취득. 1986年 10月 미국 Stanford대학 공학 박사 학위 취득. 1976年 ~ 1981年 삼성 반도체 주식회사 연구원. 1986年 ~ 1989年 Stanford대학 Center for Integrated Systems 연구소 연구원, Fairchild Semiconductor Palo Alto Research Center 기술자문. 1989年 3月 ~ 현재 서강 대학교 전자 공학과 교수. 주관심 분야는 CAD 시스템, Computer Architecture 및 Systems Design, VLSI 설계 등임.