韓國 OA學會 論文誌
第7卷 第2號, 2002. 6.

2002-7-2-1-3

VHDL을 이용한 Parwan CPU의 Modeling과 Design

박 두 열*

A study on the Modeling and design of Parwan CPU using a VHDL

Doo-Youl Park*

요 약

본 연구에서는 Parwan CPU를 VHDL을 이용하여 Behavioral Leve에서 기술하고 Dataflow LEVEL에서 상호 연결하여 기술하였고, Test-bench 방식을 이용하여 프로세서의 동작을 확인하기 위해 시뮬레이션하였다.

제시된 방식은 설계의 정보교환이 용이하고 동작의 표현이 정확하고 간결하였으며, 설계의 문서화가 용이하며, 구성된 프로세서의 동작을 확인하기가 용이하였다. VHDL의 Behavioral 기술은 설계자에게 설계된 시스템을 확인할 때 많은 도움을 주었으며. Dataflow 기술은 설계의 버스연결과 레지스터 구조를 확인할 때 유용하게 사용할 수 있었다.

Abstract

In this paper, we described the Parwan CPU using a VHDL at the behavioral level and then described by connecting CPU components at the dataflow level. Finally, we simulated to verify of execution of a CPU processor using a test-bench method.

A presented design method was to enable information exchange of design and representation of operation were very exact and simple. Also, a documentation of design was available and it was easy that verify a operation of designed processor. The behavioral description of VHDL aids designer as we verify our understanding of the designed system, while the dataflow description can be used to verify the bussing and register structure of the design.

* 동주대학 네트워크전자계열 교수
본 연구는 동주대학 학술연구비지원에 의한 논문임

논문접수 : 2002. 4. 2
심사완료 : 2002. 6. 7

Ⅰ. 서론

최근 수십 년 동안 반도체 기술의 발전으로 IC의 집적도나 성능이 급속도로 증가해 왔을 뿐만 아니라 이를 이용하는 시스템의 성능과 스케일이 급속도로 증가해왔다. 여기서 문제는 이러한 추세를 따라가기 위한 IC 칩 또는 디지털 시스템의 설계가 매우 어려워지게 된다는 것이다. 이런 문제와 부수적으로 설계과정에서 발생하는 오류의 검출 및 설계비용 등의 문제를 해결하기 위해 다양한 하드웨어 기술언어들을 사용하게 된다. 이 하드웨어 기술언어들로는 CDL[1], DDL[2], HDL[3], AHPL[4] 및 VHDL[5]들이 있다.

일반적으로, 점차 커져 가는 IC, 디지털 시스템 및 컴퓨터 설계를 Low-level에서 설계하기에는 무척 어렵게 되었으므로 High-level의 프로그래밍 언어를 이용하듯이 게이트 레벨에서 시스템을 기술하고 설계하는 것이 아니라, Algorithm 및 RTL 등과 같은 High-level에서 설계를 생성해내는 하향설계 방식을 이용하는 것이 요구되었고, 이렇게 함으로써 설계기간의 단축, 설계의 관리교환 및 재사용과 시뮬레이션이 쉽게 이루어지는 처리가 필수적으로 요구되었다[6~8].

따라서, 본 연구에서는 여러 가지의 하드웨어 기술언들 중에서 설계의 과정과 결과의 필수조건을 만족하는 VHDL을 이용하였다. 이 VHDL은 하드웨어의 기술과 표현을 위한 목적으로 만들어진 하드웨어 기술 전용 언어로서 하드웨어의 설계, 설계된 하드웨어의 검증 및 확인과 같은 일반적인 특징을 갖고 있을 뿐만 아니라 설계정보의 교환 및 보존 등의 여러 가지 용도로 사용될 수 있으며, 이것은 IEEE가 표준화함으로서 전자회로 또는 디지털 시스템 설계 관련 연구와 개발에 종사하는 개발자에게 매우 중요하게 사용되고 있다.

본 논문에서는 이 VHDL을 이용하여 Parwan CPU[7]를 구성요소 별로 기술하여 연결하고, 완성된 시스템의 동작을 확인하기 위해 Test-bench를 이용[8~10]]하여 시뮬레이션하였다. 여기서 Parwan으로 알려진 CPU를 사용한 이유는 간소화된 프로세서로서 논리를 설계하는 공학도에게 컴퓨터 하드웨어를 가르치는 데 사용되고 있고, 그것은 Simple Instruction Set를 채택하고 있으므로 아키텍처의 하드웨어를 상세하게 기술하기가 쉽고, 설계자가 논리소자로 CPU 내부의 구성을 쉽게 할 수 있다.

그에 따라 먼저 CPU의 데이터 경로와 그것의 상세한 구조가 설계되고, 설계정보는 제시된 Parwan CPU에 대한 Dataflow 기술을 개발하는 데 사용되도록 하며, 최종적으로는 Behavioral 모델이나 Dataflow 모델을 테스트하기 위해 Test-bench를 개발한다. 여기서 개발된 Test-bench를 이용한 시뮬레이션은 Parwan이 제대로 동작할 것인지를 시험하기 위해 컴퓨터에 설계된 것과 입력신호를 주고 출력이 원하는 대로 나오는지를 컴퓨터로 처리하는 과정으로 사용한다.

본 연구에서 제시된 이 방식[8~10]은 일반적인 다른 HDL들을 이용하여 하드웨어를 기술할 때의 설계 및 시뮬레이션 특징을 갖고있을 뿐만 아니라 정확하고 간단한 동작확인이 가능하다는 이점과 VHDL이 자연언어에 비해 간결하고 규정된 형식이 있어서 설계된 Parwan CPU의 설계와 동시에 문서화하기가 편리하다는 이점을 제공해 줄 수 있을 것으로 보았다.

Ⅱ. Parwan의 메모리와 명령어

Parwan CPU는 데이터 레지스터와 버스 크기가 8비트이기 때문에 8비트의 외부 데이터 버스와 12비트의 번지 버스를 가지며, ALU에서는 간단한 기본 연산만이 가능하고, 그것은 역시 몇 개의 점프와 분기 명령과 직접 및 간접 번지지정방식을 갖고 있으며, 역시 Parwan은 동작을 리세트하는 입력 인트럽트와 명령을 호출하는 간단한 서브루틴을 갖고 있다.

2.1 Parwan의 메모리 구성

Parwan CPU는 12-bit 번지 버스로 구성되기 때문에 4096바이트의 메모리 번지지정이 가능하고, 이 메모리는 256바이트로 구성된 16개의 페이지로 구분된다. 역시 〈Fig.1〉에서 보여주듯이 번지 버스의 상위 4비트는 페이지 번지를 구분하고, 하위 8비트는 옵세트를 지정한

다. 여기서 페이지와 옵세트는 컴마로 구분하고, 16페이지의 메모리 페이지에 4K로 처리되며, 페이지 교차는 자동적으로 수행된다. 메모리는 I/O 장치의 통신을 위해서도 사용되며, 페이지 교차는 자동적으로 처리된다.

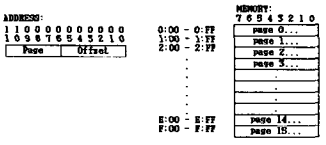

그림.1 Parwan번지의 옵세트부의 페이지
Fig.1 Page of offset parts of Parwan address

2.2 명령어 세트

Parwan은 〈Table.1〉에서 보여주듯이 3개의 명령형식을 갖는 17개의 명령으로 구성된다. Accumulator(AC)는 CPU의 데이터 레지스터로서 모든 명령과 처리를 하는 데 사용되며, V, C, Z와 N 등의 플래그를 갖는다. 이 플래그들은 주어진 플래그 명령이나 AC를 변경시키는 명령에 의해 그 값이 변형된다.

〈Table.1〉에서는 사용되는 각각의 명령들의 동작과 번지의 비트 수에 따라 각각 Full-address 명령, Page-address 및 무번지 명령을 설명하고, 간접 및 직접 번지지정 방식을 설명하고 있다. 또한 각각의 명령의 실행결과에 따라 플래그들의 변화를 설명하고 있다.

표.1 Parwan 명령의 설명
Table.1 Summary of Parwan instructions

Instruction Mnemonic	Brief Description	ADDRESSING			FLAGS	
		Bits	Scheme	Indirect	use	set
LDA loc	Load AC w/(loc)	12	FULL	YES	-	--zn
AND loc	AND AC w/(loc)	12	FULL	YES	-	--zn
ADD loc	Add (loc) to AC	12	FULL	YES	c	vczn
SUB loc	Sub (loc) from AC	12	FULL	YES	c	----
JMP adr	Jump to adr	12	FULL	YES	-	----
STA loc	Store AC in loc	12	FULL	YES	-	----
JSR tos	Subroutine to tos	12	PAGE	NO	-	----
BRA_V adr	Branch to adr if V	8	PAGE	NO	v	----
BRA_C adr	Branch to adr if C	8	PAGE	NO	c	----
BRA_Z adr	Branch to adr if Z	8	PAGE	NO	z	----
BRA_N adr	Branch to adr if N	8	PAGE	NO	n	----
NOP	No operation	8	NONE	NO	-	----
CLA	Clear AC	-	NONE	NO	-	----
CMA	Complement AC	-	NONE	NO	c	--zn
CMC	Complement carry	-	NONE	NO	-	-c--
ASL	Arith shift left	-	NONE	NO	-	vczn
ASR	Arith shift right	-	NONE	NO	-	--zn

2.3 명령어 서식

〈Table.1〉에서 보여주었듯이 명령은 3개의 그룹으로 분류된다. 2바이트의 Full-Address 명령은 메모리를 엑세스하며, 간접번지 지정방식을 갖고 있다. 2바이트를 갖는 페이지 번지지정 명령은 현재 페이지를 엑세스할 수 있으나, 간접번지 지정방식에서는 사용될 수는 없다. 세 번째 그룹은 무번지 명령이며, 메모리를 오퍼란드로 사용하지는 않는다. 〈Table.2〉에서는 Opcode와 3개 그룹 명령에 대한 서식을 보여준다.

표.2 Parwan 명령의 동작코드
Table.2 Parwan instruction opcodes

Instruction Mnemonic	Fields and Bits		
	Opcode 7 6 5	D/I 4	Bit7 3 2 1 0
LDA loc	0 0 0	0/1	page adr
AND loc	0 0 1	0/1	page adr
ADD loc	0 1 0	0/1	page adr
SUB loc	0 1 1	0/1	page adr
JMP adr	1 0 0	0/1	page adr
STA loc	1 0 1	0/1	page adr
JSR tos	1 1 0	-	- - - -
BRA_V adr	1 1 1	1	1 0 0 0
BRA_C adr	1 1 1	1	0 1 0 0
BRA_Z adr	1 1 1	1	0 0 1 0
BRA_N adr	1 1 1	1	0 0 0 1
NOP	1 1 1	0	0 0 0 0
CLA	1 1 1	0	0 0 0 1
CMA	1 1 1	0	0 0 1 0
CMC	1 1 1	0	0 1 0 0
ASL	1 1 1	0	1 0 0 0
ASR	1 1 1	0	1 0 0 1

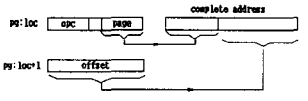

그림.2 Full-address명령의 번지지정
Fig.2 Addressing in full address instructions

2.3.1 Full-Address 명령

Full-Address 명령의 동작을 규정하는 Opcode는 첫 번째 바이트의 상위 3비트에 의해 서식화된다. 4번 비트는 직접 및 간접번지 지정방식을 규정한다. 그리고 나머지 4비트는 명령의 오퍼란드의 페이지 번호를 나타낸다. Full-Address 명령의 두 번째 바이트는 페이지 번지와 함께 오퍼란드의 번지를 표시하는 옵세트 번지를 규정한다. 〈Fig.2〉는 Parwan 명령의 완전한 12비트 번지의 명령어 서식을 보여준다.

2.3.2 페이지 번지 명령

〈Fig.3〉은 JSR과 분기명령에 대한 서식을 보여준다. 이 명령들은 명령들이 나타나는 페이지 내에서의 메모리를 참조한다. JSR의 Opcode는 '110'이고, 첫 번째 바이트의 나머지 5비트는 무시된다. 분기명령의 Opcode는 '111'이다. 비트 4는 항상 '1'이고, 최하위 비트는 분기조건을 규정한다. JSR과 분기의 두 번째 바이트는 현재의 페이지에서 점프할 번지를 지정한다.

JSR의 실행 후에 돌아올 번지(메모리에서 JSR의 다음에 오는 명령의 번지)는 서브루틴의 첫 번째 위치로 바뀐다. 간접 점프는 서브루틴 호출번지로 돌아오도록 프로

그램 흐름이 이루어지게 한다.

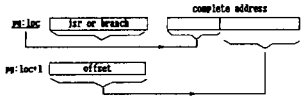

그림.3 page address 명령의 번지지정
Fig.3 Addressing in page address instructions

2.3.3 무번지 명령

무번지 명령들은 〈Table.2〉에서 보여주듯이 상위 4비트가 '1110'이고, 나머지 4비트는 NOP, CLS, CMA, CMC, ASL 및 ASR의 명령들을 구분한다.

2.3.4 Parwan의 간접번지 지정방식

Full-Address 명령에서 첫 4 비트가 '1'이면 이 명령에서의 번지는 오퍼란드의 간접번지 지정을 표시한다. 간접번지 지정방식은 메모리로부터 8비트의 옵세트를 받아들이기 위해 12비트의 번지를 사용한다. 간접번지의 페이지번호를 갖는 이 옵세트는 명령의 실제 오퍼란드의 완전한 번지를 만든다. 〈Fig.4〉는 Parwan의 간접번지의 예를 보여준다. 여기서 실제 오퍼란드는 6:1F의 18이다.

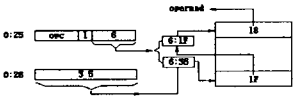

그림.4 Parwan의 간접번지의 예
Fig.4 An example for indirect addressing in Parwan

Ⅲ. Behavioral 기술

여기에서는 8비트의 시스템을 Behavioral로 기술하고 있으며, 전체적인 기술은 부록 A에 보여진다. 이 시스템의 상호연결을 위한 기술은 외부 제어신호와 메모리를 위한 비트들과 databus를 사용하고 있는 하드웨어 수준에서 이루어진다.

3.1 타이밍과 클럭킹

Behavioral 기술의 상호연결에는 사용되지 않는 클럭 신호들도 포함되어있고, 이 신호들은 실제적인 칩과 dataflow 모델에는 사용되나, Behavioral 모델의 타이밍은 클럭과는 무관하고, 실제적인 칩클럭과 정확히 일치하지 않을 수도 있다. 이런 타이밍은 메모리로나 메모리를 R/W할 때 Behavioral 모델에도 도입된다.

3.2 페키지

Behavioral 수준에서 Parwan을 기술할 때 basic_utility 페키지를 사용한다. 이 페키지는 CMOS 설계 라이브라리에 있으므로 필요에 따라 불러 사용한다. 또 하나의 par_library는 Parwan을 기술할 때 요구되는 유틸리티를 가지며 basic_utility에는 이것이 없다.

3.2.1 par_utility 페키지

par_library에 나타나는 2개의 페키지 중 첫 번째 것은 par_utility이고 그 페키지 선언은부록 A에 보여지며, 다양한 종류와 함수선언을 하고있는 이 페키지는 CMOS 라이브라리의 basic_utility 페키지를 사용한다.

여기에서 4, 8, 12 qit길이의 nibble, byte 및 twelve는 qit_vector이고 wired_nibble, wired_byte 및 wired_twelve는 basic_utility 페키지의 Wiring(Oring) 함수를 사용하여 qit_vector를 찾아낸다. par_utility 페키지내의 논리함수는 qit_vector 오퍼란드를 위해 대응하는 함수를 basic_utility 페키지내로 로드한다. basic_utility는 qit 형식의 오퍼란드를 위해 함수들을 갖는 기본적인 연산자를 로드한다고 가정하면 된다.

par_utility 페키지의 add_cv와 sub_cv 등의 함수는 덧셈과 뺄셈을 수행한다. 〈Fig.5〉는 par_utility 페키지의 몸체의 기술은 부록A에 보여진다.

3.2.2 par_parameter 페키지

par_library의 또 다른 페키지는 par_parameter이다. 여기서는 Opcode들의 비트 스트링과 명령의 그룹들을 정의한다.

Parwan Behavioral 기술의 상호연결은 CPU의 전체적인 구성에 따라 기술된다. CPU속으로 들어가는 입력들의 선언부는 형식이 qit, qit_vector 또는 그에 부응하는 신호들을 사용한다. 따라서 여기서는 Parwan의 상호연결 기술을 먼저 해야한다

이 기술은 CMOS 라이브라리의 basic_utility Package와 par_library 페키지의 par_utility와 par_parameter를 사용하고 있다. 여기서 규정하는 것은 메모리 R/W 타이밍

과 사이클 타임들이고, 또한 Parwan 입출력에 관련된 양방향 데이터 버스를 규정하고 있다. 이 데이터 버스는 basic_utilities 패키지의 출력함수를 연결할 때 사용하는 신호이다.

3.4 Parwan Behavioral 아키텍처

Parwan의 Behavioral 기술은 명령실행의 처리와 과정으로부터 시스템을 모델화하여 제시했다. 이 모델의 기능은 Dataflow와 게이트 수준이어서 실제적인 하드웨어로 기술된다.

par_central_processing_unit의 Behavioral 아키텍쳐의 전반적인 기술한 다음에는임시 변수와 버퍼를 선언하며, 인터럽트 입력이 '1'이면 실행되는 코드를 기술한다. 바이트 1 변수가 이 바이트에 어떻게 저장되는지를 기술하고 메모리의 첫 번째 바이트가 읽혀져 데이터 버스에 연결된 후, 다음 명령을 읽을 준비를 하는 동작을 기술한다.

```
ARCHITECTURE behavioral OF par_central_processing_unit IS
BEGIN
  PROCESS
    -- Declare nessary variable;
  BEGIN
    IF interrupt = '1' THEN
      -- Handle interrupt;
    ELSE
      -- Read first byte into byte1, increment pc;
        IF byte1 (7 DOWNTO 4) = single_byte_instructions THEN
          -- Execute Single-byte instructions;
        ELSE
          -- Read second byte into byte2, increment pc;
            IF byte1 (7 DOWNTO 5) = jsr THEN
              -- Execute jsr instruction, byte2 has address;
            ELSEIF byte1 (7 DOWNTO 4) = bra THEN
              -- Execute bra instruction, address in byte2;
            ELSE
              IF byte1 (4) = indirect THEN
                -- Use byte1 and byte2 to get address;
              END IF
              IF byte1 (7 DOWNTO 5) = jmp THEN
                -- Execute jmp instruction;
              ELSEIF byte1 (7 DOWNTO 5) = sta THEN
                -- Execute sta instruction,write ac;
              ELSE
                -- Read memory onto databus;
                -- Execute lda, and, add, and sub;
                -- Remove memory from databus;
              END IF
            END IF
        END IF
    END IF
  END PROCESS;
END behavioral;
```

그림.5 전반적인 Parwan behavioral 기술
Fig.5 Outline of Parwan behavioral description

〈Fig.5〉의 Behavioral 기술에서는 명령의 첫 번째 바이트를 읽은 후, 1개의 메모리 바이트만을 사용하는 명령들을 찾아낸다. 이 명령들을 위해 Single 바이트 명령의 코드가 실행된다. 이 코드는 CLA, CMA, CMC, ASL 및 ASR의 명령 바이트의 (3~0)비트를 검사하여 적합한 동작을 수행한다. 2개의 메모리 바이트를 사용하는 명령에서 다른 1바이트는 메모리로부터 읽혀져서 2바이트 변수로 치환된다.

JSR 명령의 실행 코드는 PC의 옵세트부가 서브루틴의 Top에 먼저 기록된다는 것을 보여준다. 그것의 옵세트는 명령의 두 번째 바이트로부터 얻어진 메모리 위치이며, PC의 내용을 이 위치로 전달하면, 서브루틴의 돌아올 번지가 얻어진다.

분기코드 JSR의 실행은 byte 2의 내용을 갖고서 PC의 옵세트부의 조건에 따라 로드하도록 해준다. 여기서 분기동작은 현재 페이지에 대해선 만 이루어진다.

읽어온 2바이트 명령이 JSR 혹은 분기 명령이 아니면, par_control_processing_unit의 Behavioral 아키텍쳐는 간접번지 지정을 검사하게 된다.

간접번지 지정의 기술에서 페이지 번호가 byte 1의 최하위 nibble에서 만들어지고 그것의 옵세트 byte 2내의 12비트의 번지는 메모리를 지정하기 위해 사용되고, 실제적인 옵세트를 읽는데 사용된다. 여기서 byte2의 이전 내용은 새로 읽은 바이트로 대체된다.

Full-Address 명령을 위한 코드는 〈Fig.5〉에서 보여준 간접번지 지정에 따라 나온 JMP이고, 그 명령의 실행을 위한 기술에서 보여주는 것처럼 그것은 byte 1과 byte 2로부터 만들어진 PC의 번지를 로드하면 Full-Address 명령 STA는 byte 1과 byte 2의 최하위 nibble에 의해 만들어진 메모리 번지에 AC를 저장한다. 그 동작은 부록A의 sta 명령을 위한 기술에서 보여진다.

Full-Address 명령의 마지막 그룹은 LDA, ADD 및 SUB로서, 실제 오퍼란드는 메모리로부터 읽혀져 databus에 연결된다.

Parwan의 Behavioral 기술에서 명령의 실행을 위해 적어도 한 사이클 타임이 사용되며, 각 타임에서 하나의 명령이 완전히 실행되고, 3개의 사이클이 요구된다. Parwn의 완전한 Behavioral 기술은 모두 보록A에 보여진다. 그리고 이 기술들은 〈Fig.5〉의 지정된 위치에 삽입된다.

Ⅳ. 버스 연결 구조

CPU의 하드웨어 설계단계는 그것의 레지스터와 논리장치의 버스연결 구조를 기술하는 것이다. 〈Fig.6〉은

Parwan의 버스연결 구조를 보여준다. 이 다이어그램은 Parwan의 Dataflow를 기술할 때 사용될 수 있다.

4.1 구성요소들의 상호연결

Parwan의 주요 구성요소는 AC, IR, PC, MAR, SR, ALU 및 SHU이다. 데이터는 버스들을 하드와이어적으로 연결하여 구성요소들 사이에서 전송된다. 〈Fig.6〉은 버스에 레지스터와 논리장치의 출력 및 제어신호의 상호연결과 데이터 전송을 위한 고정된 연결을 보여준다. 구성된 Parwan의 VLSI 구현은 다중소스 버스에서의 데이터의 선택을 위해 전송 게이트를 사용한다.

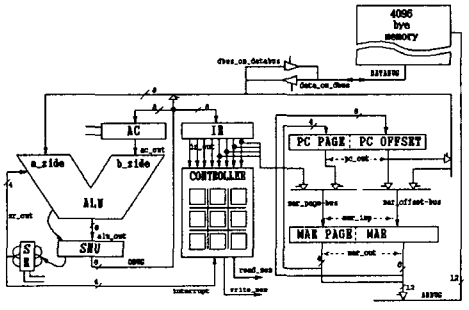

그림.6 Parwan의 버스연결 구조
Fig.6 Parwan bussing structure

4.2 Parwan 구성요소의 개요

Parwan의 구성요소 중 AC, IR, PC 및 SR은 레지스터이고, ALU와 SLU는 조합논리 회로로서 이들은 입출력 세트와 여러 개의 제어라인을 갖는다.

AC는 ALU의 오퍼란드를 제공하는 8비트 레지스터이고, IR은 ALU를 통해 dbus에 연결되며, 제어기에 의해 명령비트를 제공하고, 번지버스에 페이지 번지를 제공한다. 12비트의 PC는 2진 Upcounter이고, MAR을 통해 명령번지를 번지버스(adbus)에 제공해 준다. MAR과 PC는 페이지와 옵세트부를 갖기도 한다.

ALU는 2개의 8비트 입력과 4개의 플래그 입력과 3개의 제어입력을 갖는 조합논리회로로서, 이 출력은 8비트 오퍼란드를 갖고서 좌우측 시프트하는 SHU 장치의 입력에 연결되며, 이 ALU와 SHU를 통해 만들어진 상태는 SR의 입력이 된다.

4.3 명령의 실행

〈Fig.6〉의 버스연결구조는 Parwan 명령을 실행하기 위해 필요한 레지스터와 데이터 경로를 보여주고 있다. 제어기는 주어진 명령에 적합한 실행을 시키기 위한 제어신호를 만든다.

이 버스연결구조의 설명을 위해 LDA 명령의 실행과 처리과정을 기술해 보자. 초기에 인출할 명령의 번지를 갖는 PC를 mar_bus에 실어 MAR로 보내고, 하나 증가시킴으로써 시작한다. 이것이 수행되면 제어기는 MAR을 abus에 싣는 read_mem 신호를 만들어 읽기를 준비한다. 이 동작은 바이트가 메모리로부터 databus에 실리도록 해주며, 제어기는 databus_ondbus 제어신호를 활성화하여 메모리 출력을 a_side로 빼내 dbus에 실어 ALU에 보내고, SHU에게 시프트없이 그대로 출력하도록 하여 obus를 통해 IR로드한다. 이 것을 수행한 후에 IR의 비트에 따라 제어기가 제어신호를 만들어 그에 맞는 동작을 결정하고 지시하여 실행을 시켜준다. 간접 번지지정 방식에서, 제어기는 연산을 수행하기 전에 메모리로부터 번지를 읽도록 한다.

V. Dataflow 기술

3장에서 설명한 Behavioral 기술은 Parwan의 정확한 동작을 표현하지는 못하였다. 즉, 이 시스템의 하드웨어 구현을 위해서는 〈Fig.5〉의 기술로부터는 명확하지 않으므로, 실제 하드웨어에 더 접근한 Parwan의 기술을 해야한다. 전반적인 기술은 데이터 레지스터와 논리장치의 구조적 상호연결을 하는 것이며, 레지스터들과 버스들을 통하는 데이터의 흐름의 제어를 처리하고 있기 때문에 여기서는 그것을 부록B에서 Parwan의 Dataflow 기술로 표현하고 있다.

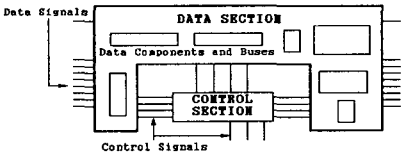

그림.7 Parwan CPU의 데이터와 제어부
Fig.7 Data and control sections of Parwan CPU

5.1 데이터와 제어의 분리동작

〈Fig.7〉은 Parwan의 Dataflow 기술을 위해 사용하는 데이터와 제어를 구분하여 설명한다. 데이터부는 CPU 구성요소의 연결규칙을 보여주며, 구성요소들과 적합한 버스에 출력을 실어주는 동작을 하게 된다.

제어부는 외부의 제어신호와 데이터부로부터의 신호를 사용하며, 레지스터 속으로 데이터의 조건적 할당과 데이터부의 버스 속으로 데이터의 할당을 제어하는 신호를 만들어 낸다.

〈Table.3〉은 데이터부 내의 데이터 이동을 제어하는 제어부에 의해 만들어진 제어신호들을 보여준다. 이 명칭들은 신호에 의해 제어되는 연산에 따라 선택된다.

표.3 제어부의 입출력
Table.3 Inputs and outputs of Parwan control section.

RELATED TO:	SIGNAL CATEGORY AND NAME
Register control signals	
AC IR PC MAR SR	load_ac, zero_ac load_ir increment_pc, load_page_pc, load_offset_pc, reset_pc load_page_mar, load_offset_mar load_sr, cm_carry_sr
Bus connection control signals	
MAR_BUS DBUS ADBUS DATABUS	pc_on_mar_page_bus, ir_on_mar_page_bus pc_on_mar_offset_bus, dbus_on_mar_offset_bus pc_offset_on_bus, obus_on_dbus, databus_on_dbus mar_on_abus dbus_on_databus
Logig unit function control signals	
SHU ALU	arith_shift_left, arith_shift_right alu_code
Memory control and other external signals	
Etc	read_mem, write_mem, interrupt

5.2 데이터의 타이밍과 제어의 처리

데이터부와 제어부는 동일한 클럭신호에 의해 구동되고, 이 클럭 주기 동안 제어부는 상태변이를 일으키고, 데이터부의 레지스터들이 로드된다. 〈Fig.8〉은 회로의 클럭에 관련된 제어신호의 타이밍을 보여준다.

하나의 제어신호는 클럭펄스의 falling-edge에서 활성화되어, 다음 negative-edge 때까지 활성화를 유지한다. 제어신호가 활성화되어 있는 동안 데이터부의 논리장치는 규정된 동작을 수행하며, 그 결과는 목적지 레지스터의 입력에 사용될 수 있게된다.

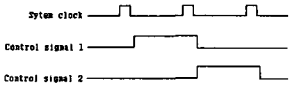

그림.8 제어신호의 타이밍
Fig.8 Timing of control signals

5.3 일반적인 기술의 근거와 순서

Parwan의 기술은 〈Fig.7〉의 데이터부와 제어부의 구분에 따라 기술되고 〈Fig.6〉에서 보여진 데이터부의 구성요소들은 Behavioral이나 Dataflow 수준에서 독립적으로 기술된다. 여기서 버스구조에 따라 그것의 구성요소들을 연결하고 구성요소출력을 버스에 연결함으로써 데이터부를 기술한다. 데이터부의 완성 후에 상태 머신 기술형식이 Parwan의 제어부의 기술을 위해 사용된다. 전체적인 기술은 데이터부와 제어부를 묶어줌으로써 완성된다.

5.4 Parwan의 구성요소의 기술

ALU, SHU, SR, IR, AC, PC 및 MAR과 같은 데이터부 구성요소들은 주어진 순서를 갖고서 기술된다. CMOS 라이브라리 내의 basic_utility와 par_library내의 par_utilities가 이 구성요소들을 기술하는 데 사용된다. 여기서, 이 구성요소들은 par_dataflow VHDL 설계 라이브라리로 모아지며, 그것의 기술들은 부록B에 모두 보여진다.

5.4.1 산술연산 논리장치(ALU)

ALU는 2개의 8비트 오퍼란드와 3개의 선택선과 4개의 플래그 입출력을 갖는다. 3개의 선택선은 〈Table.4〉에 근거하여 ALU의 동작을 선택하며, ALU의 연산에 의한 플래그의 변화를 보여준다.

표.4 ALU의 동작과 플래그
Table.4 Operations and flags of alu.

S2 S1 S0	OERATION	FLAGS
0 0 0	a AND b	zn
0 0 1	NOT b	zn
1 0 0	a	zn
1 0 1	b PLUS a	vczn
1 1 0	b	zn
1 1 1	b MINUS a	vczn

〈Fig.9〉는 Parwan의 ALU에 대한 논리심블을 보여준다. 이 ALU를 구현하기 위해 먼저 ALU의 동작의 명칭을 정의하고, ALU 코드를 쉽게 해독하기 위해 이 기술에서는 alu_operations 페키지가 basic_utility와 par_utilities에 사용된다.

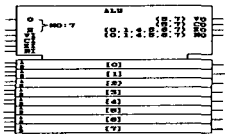

그림.9 Parwan alu.Library cmos의 논리심블
Fig.9 Logic symbol for Parwan alu. Library cmos;

Parwan ALU에 대한 VHDL기술은 부록B에 보여준다. 이 arithmetic_logic_unit의 Behavioral 아키텍쳐에서는 3비트 코드에 근거하여 ALU의 동작이 선택된다.

5.4.2 Shifter 장치

〈Fig.10〉은 시프트 장치에 대한 논리심블을 보여주며, 캐리비트 및 오버플로비트 등을 고려하여 좌우 시프트동작을 하여 출력으로 내보낸다. 이 SHU의 기술에서 시프트 연산이 없을 때는 입력 데이터와 플래그가 SHU의 출력으로 바로 나간다.

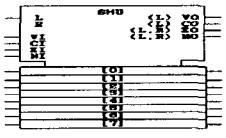

그림.10 Parwan shu의 논리심블
Fig.10 Logic symbol of Parwan shu

5.4.3 Status Register(SR)

상태 레지스터는 4비트로서 Negative-edge 트리거와 동기되어 있다. 이것은 플래그로 로드되는 데이터는 load 입력과 cm_carry 입력에 의해 동기적으로 제어된다. 이 SR의 VHDL 기술은 부록B에 보여진다.

5.4.4 Accumulator(AC)

Parwan의 AC는 연속적인 로딩과 '0'으로 만드는 입력을 갖는 8비트 레지스터이다. AC는 외부데이터를 레지스터에 로드하는 동작은 load 입력이 활성화되고, zero 입력이 비활성화될 때 클럭의 falling-edge에서 수행된다. AC의 기술은 부록B에 보여진다.

5.4.5 Instruction register(IR)

명령 레지스터 IR은 load 입력 신호에 동기된 8비트 레지스터이다. load 입력은 클럭을 활성화하고 클럭 입력의 falling-edge에서 레지스터가 로드되도록 하여 IR의 동작이 기술된다.

5.4.6 Program counter(PC)

PC는 1개의 레지스터와 2개의 load 입력을 갖는 12비트 동기식 카운터이다. load_page 입력은 최상위 4개 비트로 입력 데이터를 동기적으로 로드하고 load_offset은 레지스터의 최하위 8개 비트로 입력 데이터를 로드한다. 그것에 대한 VHDL 기술은 부록B에 보여진다.

5.4.7 Memory address register(MAR)

MAR은 2개의 동기된 load입력을 갖는 12비트 레지스터이다. load_page 입력은 레지스터의 최상위 nibble로 병렬로 로드하고, load_offset은 그것의 최하위 바이트로 로드하며, 그것의 동작에 따라 MAR이 기술된다.

5.5 Parwan의 데이터부

Parwan의 데이터부는 여러 구성요소들의 상호연결을 규정하고 Parwan의 버스연결구조를 정의한다. 이 장치의 입력들은 데이터버스, 클럭신호 및 제어부로부터의 신호들이다. 제어신호들은 데이터부 내의 구성요소들의 동작을 규정하고 그들의 클럭을 제어하여 입력을 데이터버스에 신도록 해준다. 데이터부의 출력들은 데이터버스와 번지버스에 연결된 것으로 IR의 비트들과 4개의 상태 플래그들이다.

par_data_path의 structural 아키텍쳐의 선언부의 기술은 구성요소들의 규격과 데이터부의 구성요소를 선언하고 있다. 이 선언부는 데이터부의 구성요소들을 연결하기 위한 신호와 버스도 선언하고 있다.

par_data_path의 structural 아키텍쳐의 기술은 구성요소의 초기상태, 신호할당과 〈Fig.6〉에서의 연결을 규정하는 기술들이다. 역시 레지스터들의 연결을 기술한 것은 Parwan의 레지스터 동작의 준비와 연결을 정하며, 마지막으로 ALU와 SHU의 초기동작을 지시하고 있으며, 〈Fig.6〉의 구성에 따라 이 두 장치들의 연결을 규정하고 있다.

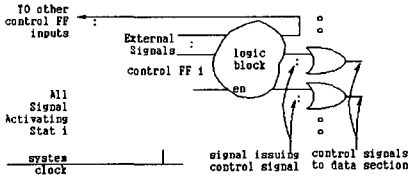

그림.11 제어 FF의 하드웨어
Fig.11 Typical hardware surrounding a control FF

5.6 Parwan의 제어부

Parwan의 제어부의 기술은 정확히 1-클럭주기 동안 활성화되는 선택된 제어신호를 갖는 상태의 변이로 이루어진다. 이 상태머신의 기술에서는 multiple_state_

machine의 것과 비슷한 형식을 사용한다. 이 Parwan 제어기에서, 클럭의 edge 검출은 매 상태에서 반복된다. 여기서 기술한 형식의 하드웨어 구성은 FF이 각각의 상태에 대응하여 이루어진다. 〈Fig.11〉은 실제적인 제어 FF 하드웨어를 보여준다.

여기서 논리블럭의 입력들은 다른 상태FF과 제어부에서 상태변이에 영향을 주는 외부 입력들로부터 나온 것이며, 논리블럭의 출력들은 제어신호가 되고 데이터부에 주어지는 것들이다.

Parwan의 제어부의 입력선언은 메모리 읽기와 쓰기 신호들을 위한 지연값을 갖고있고, 데이터부의 ir_lines에 대한 출력과 데이터부로부터의 상태입력과 외부 메모리 처리와 인터럽트 라인들을 기술한다.

par_control_unit의 Dataflow 아키텍쳐의 선언부의 기술은 제어부의 각각의 제어출력을 위한 예정된 신호들을 선언하고 basic_utilities Package에 있는 Oring함수를 사용한다.

par_control_unit의 Dataflow 아키텍쳐의 기술에서 신호들은 par_control_unit의 실제적인 출력에 들어가고, 이것은 OR 게이트들의 출력을 제어부의 실제 출력으로 연결해준다.

상태 1에서 인출동작은 PC를 mar 버스에 실어줌으로써 시작하며, 인터럽트 입력이 활성화되면, PC가 리세트되어 제어는 상태 1로 되돌아온다. CPU가 인터럽트되지 못하면, 상태 2가 클럭에서 활성화된다.

그 다음에 data_bus_on_dbus를 사용하여 데이터 버스의 내용을 IR로 전송하도록 ALU의 a_input로 연결한다. 클럭의 edge에서 제어상태 3이 활성화되고, IR은 새로운 값을 갖게 되며, 다음 클럭에서 PC가 증가하도록 해준다.

상태 2에 있을 때 MAR 버스는 상태 1에 의해 만들어진 새로운 값을 받아들이며, 이 기술에서 보여주듯이 adbus에 MAR을 싣고, read_mem을 발생시킴으로써 인출동작을 완료한다.

상태 3이 활성화될 때 이 부분의 기술에서 알 수 있듯이 새롭게 읽은 명령은 IR에 오게된다. 상태 3은 메모리로부터 다음 바이트를 읽기 시작한다. 그와 동시에 상태 3은 현재의 명령의 바이트 수를 검사하여 2 바이트 명령이면, 아래 바이트가 번지가 되고, 제어상태 4가 2바이트 명령의 실행을 계속하기 위해 활성화된다. 반면에 현재의 명령이 무번지 명령이고 두 번째 바이트가 필요없다면, 상태 3은 명령실행을 한 후에, 다음 명령을 인출하기 위해 상태 2를 활성화하게 된다.

상태 4는 Full-address나 Page-address 명령이 실행되고 있을 때 활성화된다. 상태 3에서, 번지 바이트(명령의 두 번째 바이트)를 읽을 준비가 되면, 상태 4의 기술에서 이 읽기 동작을 수행하고 mar의 옵세트부의 입력에서 새로운 바이트를 읽는다.

상태 4는 Full-address 명령의 직접 및 간접번지 지정방식을 처리하는 상태 5나 6을 구동한 다음 각각 JSR이나 BRA 명령에 대한 상태 7이나 9를 구동한다. 상태 5를 구동하는 클럭에서 번지가 MAR에 로드된다.

상태 5의 기술은 간접번지 지정방식을 처리하며, 직접번지 지정 모드가 사용될 때 상태 4에 의해 구동되는 것과 같이 상태 6을 구동한다.

상태 6은 JMP, STA, LDA, AND, ADD 및 SUB명령이 실행될 때 구동된다. 이 상태 동안 MAR은 완전한 오퍼란드의 번지를 기억하고 있다. jm, st 및 rd 명령들을 수행하기 위한 상태의 기술에서는 load_pc가 MAR의 내용이 PC로 로드되도록 활성화된다. 이것은 PC에 의해 지정된 메모리 위치로부터 새로운 명령을 인출하기 위한 상태 2의 동작에 의해 나온 것이다.

st 블록의 표현은 STA명령의 Opcode가 ir_lines의 MSB에서 검사되면 활성화되며, rd 블록은 상태 6으로 연결되어 LDA, AND, ADD 및 SUB를 처리한다. 여기에서 Full-address 명령의 수행에 따라 제어는 다음 명령 인출을 위해 상태 1로 분기한다.

상태 7의 기술은 sr의 실행을 계속시켜 PC의 내용을 MAR에 의해 지정된 서브루틴의 Top으로 기억시키고, 그 시간에 PC를 위해 서버루틴의 Top의 번지(MAR의 12비트)를 저장 목적지로 지정한다. 클럭에서 상태 7의 다음에 나오는 상태 8의 기술은 sr의 실행을 완료하기 위해 활성화된다.

실제적인 서브루틴의 코드는 top_of_subroutine후의 위치에서 시작하기 때문에 상태 8은 increment_pc 신호를 만들어내고, 서브루틴의 첫 번째 명령을 인출하기 위해 상태 1을 구동한다.

이렇게 되면 제어는 상태 9에 도달하고, 그때 상태 4가 구동되고, 분기명령이 수행된다. 상태 9가 구동될 때

분기번지는 MAR 레지스터에 있다. 상태 9는 분기방향(3~0비트)과 상태 레지스터 비트(v,z,s,n 플래그) 사이에서 조건이 만족된다면 MAR을 PC로 로드한다. 분기조건이 만족되지 않으면, PC는 본래 값을 유지한다. PC 번지는 분기명령 다음에 나오는 메모리 위치를 지정하므로 제어는 다음 명령을 인출하기 위해 상태 1로 되돌아간다. 모든 수행의 완료 후 par_control_unit의 Dataflow 아키텍처에서는 모든 상태에 '0'을 할당한다.

지금까지 Parwan 제어기의 Dataflow 기술을 완성했고 그것은 부록에 모두 보여진다. 여기서 제시된 기술에 대응하는 회로의 전반적인 도시는 〈Fig.12〉에 보여지며, 이 회로 다이어그램은 〈Fig.11〉에서 사용한 것과 동일한 표기를 갖는다.

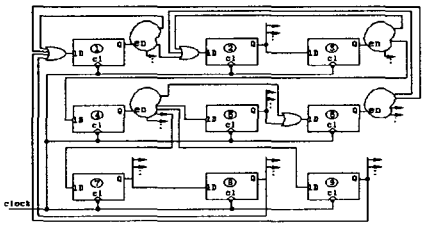

그림.12 Parwan 제어기의 다이어그램
Fig.12 General outlin of the Parwan controller

5.7 Parwan의 연결 데이터와 제어부

Parwan 프로세서의 완전한 Dataflow 기술은 par_data_path와 par_control_unit로 구성되어 있다. 이 CPU의 입력선언은 〈Fig.13〉에 보여지며, 이 선언은 인터럽트 처리 기술에서 보여진 것과 비슷하다.

```
ENTITY par_central_processing_unit IS
  PORT (clk: IN qit; interrupt: IN qit;
        read_mem, write_mem: OUT qit;
        databus: INOUT wired_byte BUS :=
                 "ZZZZZZZZ"; adbus: OUT
twelve );
END par_central_processing_unit;
```

그림.13 Dataflow기술을 위한 입력선언

Fig.13 Entity declaration of the Parwan CPU for it's dataflow description

par_centrall_processing_unit의 Dataflow 아키텍처의 이 기술은 par_data_path의 Structrural 아키텍처와 par_control_unit의 Dataflow 아키텍처의 상호연결을 규정하여 전체적인 윤곽을 보여준다.

Ⅵ. Test-bench를 이용한 시뮬레이션

〈Fig.14〉는 Parwan CPU의 간단한 Test Bench를 보여준다. 이 기술은 par_central_processing_unit를 시동하고, 그것의 인터럽트와 클럭신호의 파형을 발생시키고, SRAM의 구성을 모델링한다.

```
ARCHITECUTRE input_output OF parwan_tester IS
  COMPONENT parwan PORT (clk: IN qit; interrupt: IN qit;
                         read_mem, write_mem: OUT qit;
          databus: INOUT wired_byte; adbus: OUT twelve);
  END COMPONENT;
  SIGNAL clock, interrupt, read, write: qit;
  SIGNAL data:wired_byte := "ZZZZZZZZ"; address: twelve;
  TYPE byte_memory IS ARRAY (INTEGER RANGE <> ) OF byte;
BEGIN
  int: interrupt <= '1','0' AFTER 4500 NS;
  clk: clock <= NOT clock AFTER 1 US WHEN NOW <= 140 US ELSE
clock;
  cpu: parwan PORT MAP (clock, interrupt, read,
                        write, data, address);
  mem: PROCESS  -- <Fig 14-b>
    VARIABLE memory: byte_memory (0 DOWNTO 63) :=
      -- ( (LDA 24, STA 25) (AND 26, ADD 27)
      -- (CAC, ASR, SUB 28) (LDA i 29, JSR 36)
      -- (ASL, NOP, JMP 32) (4(H8'0')
      -- (24, 25, 26, 27) (28, 29, 30, 31)
      -- (JMP 18) ( , CMA, JMP i 36) (24(H8'0')));
    VARIABLE ia: INTEGER;
  BEGIN
    WAIT ON read, write; qit2int(address, ia);
    IF read = '1' THEN
      IF ia >= 64 THEN data <= "ZZZZZZZZ";
      ELSE data <= wired_byte (memory(ia));
      END IF;
      WAIT UNTIL read = '0';
      data <= "ZZZZZZZZ";
    ELSE write = '1' THEN
      IF ia < 64 THEN memory(ia) := byte(data);
      END IF;
      WAIT UNTIL write ='0';
    END IF;
  END PROCESS mem;
END input_output;
```

(a) Assembly for Test-bench

```
VARIABLE memory: byte_memory (0 DOWNTO 63) :=
  ("00000000", "00011000", "10100000", "00011001",
   "00100000", "00011010", "01000000", "00011011",
   "11100010", "11101001", "01100000", "00011100",
   "00010000", "00011101", "11000000", "00100100",
   "11101000", "11100000", "10000000", "00100000",
   "00000000", "00000000", "00000000", "00000000",
   "00001100", "00011111", "00000000", "00000000",
   "00001100", "00011111", "00000000", "01011010",
   "10000000", "00010010", "00000000", "00000000",
   "00000000", "11100010", "10010000", "00100100",
   "00000000", "00000000", "00000000", "00000000",
   "00000000", "00000000", "00000000", "00000000",
   "00000000", "00000000", "00000000", "00000000",
   "00000000", "00000000", "00000000", "00000000",
   "00000000", "00000000", "00000000", "00000000",
   "00000000", "00000000", "00000000", "00000000");
```

(b) Binary code for Assembly

그림.14 간단한 Test-bench
Fig.14 A simple test bench for Parwan behavioral and dataflow descriptions.

parwan_tester의 input_output 아키텍처내에 있는 mem은 Parwan의 여러 명령들로 초기화되어 있는 64바이트 배열을 처리하고 있다. 그 처리는 메모리 동작을 수행하기 위해 R/W 신호가 '1'이 될 때를 기다린다. 한

바이트를 읽기 위해 그 처리는 메모리를 인덱스하는 번지 값을 사용하여 메모리로부터 만든 바이트를 data버스에 실어준다.

이 데이터는 read가 '0'이 될 때까지 머무르고, mem 처리는 'z'를 갖고서 데이터선을 구동할 때까지 머무른다. 메모리에 데이터를 저장한 후, write가 활성화될 때 그 처리는 다른 R/W요청을 검사하기에 앞서 write 제어신호의 제거를 기다리게 된다.

〈Fig.15〉는 Parwan CPU의 Behavioral 혹은 Dataflow 기술에 대한 〈Fig.14〉의 Test bench에서 par_central_processing_unit의 CPU 동작을 묶어주는 선언을 보여준다. 여기서는 par_central_processing_unit의 두 버전을 테스트하기 위해 동일한 아키텍쳐를 사용하고 있다.

〈Fig.15〉의 Behavioral과 Dataflow 구성의 시뮬레이션은 Parwan 프로세서의 2개의 기술이 함수적으로 동일하다는 것을 표현하고 있다.

```
CONFIGURATION behavior OF parwan_tester IS
  FOR input_output
    FOR cpu: parwan
      USE ENTITY
behavioral.par_central_processing_unit(behavioral);
    END FOR;
  END FOR;
END behavior;
```

(a)

```
CONFIGURATION dataflow OF parwan_tester IS
  FOR input_output
    FOR cpu: parwan
      USE ENTITY
par_dataflow.par_central_processing_unit(dataflow);
    END FOR;
  END FOR;
END datafow;
```

(b)

그림.15 Parwan Tester의 입출력 구성
(a) par_centeral_processing_unit의 behavioral 테스트 (b) Dataflow의 테스트
Fig.15 Configuring input_output architecture of the Parwan tester.
(a) for testing behavioral architecture of the par_central_processing_unit, (b) for testing dataflow architecture of the par_central_processing_unit

여기서는 〈Fig.6〉에서 보았듯이 이 버스연결 구조에서 Dataflow 기술에 근거를 두었다.

〈Fig.14〉의 테스트 프로그램의 실행을 위한 시뮬레이션 수행은 Parwan CPU의 Behavioral 모델에 대해서 0.7초 걸렸고, Dataflow 모델에 대해서는 4.4초 걸렸다. 특히 정확한 시뮬레이션 결과를 얻기 위한 비용은 CPU의 사이클에 달려 있었다.

Ⅶ. 결 론

본 연구에서는 Parwan 프로세서의 CPU를 VHDL을 이용하여 Behavioral Level에서 기술하고, Dataflow Level에서 상호 연결하였으며, 구성된 프로세서의 동작을 확인하기 위하여 Test-bench를 이용하여 시뮬레이션하였다.

이 과정에서, 시스템이 설계되기 전에 Behavioral Level에서 시스템을 기술할 때 VHDL이 어떻게 사용되는 지와 주 설계 판단이 결정된 후 Dataflow Level에서 시스템의 버스연결과 레지스터 구조를 기술할 때 VHDL이 어떻게 사용되는 지를 보여 주었으며, 특히 Parwan 설계를 완성하기 위해서 Parwan Dataflow 모델에서는 플립플롭과 게이트들의 상호연결은 구성요소 별 기술로 표현하였다.

구현된 프로세서를 시뮬레이션하기 위해서 여기서는 프로세서에서 수행되는 명령어를 제시된 시험체제의 환경하에서 Test-vector를 프로세서에 입력하여 수행하였다. 이렇게 함으로써 동작을 확인할 때 시스템의 수행과정과 특성을 쉽게 파악할 수 있었고, 역시 클럭의 순서에 따른 동작확인이 가능하였다.

결론적으로 본 연구에서 VHDL을 이용한 시스템의 설계와 시뮬레이션 방법은 다음과 같은 장점을 얻을 수 있었다.

- 기술한 시스템의 문서를 통해 설계자들 사이 및 설계자와 툴 사이의 정보교환이 용이하였고,
- 설계하고자 하는 하드웨어가 어떻게 동작하는 지의 표현이 정확하고 간결하였으며,
- 기술방식이 형식화되어 있어서 설계와 동시에 문서화하는 것이 가능하였고,
- 특히 시뮬레이션이 쉽고 간결하였다.

따라서 위에서 설명한 것을 근거로 볼 때 본 연구는 산업체에서의 칩 개발과 연구기관에서의 교육과 연구에 많은 도움이 될 것으로 보며, 복잡한 하드웨어 설계의 문서를 간결화 함으로써 설계과정의 다양한 결정을 용이하게 할 수 있도록 해줄 것으로 본다.

참고문헌

[1] Y.Chu, "Structure of CDL programs", Dep. computer science. Univ. Maryland, Tech. no.1 pp.74-58, May 1974.

[2] J.R.Duley and D.L.Dietmeyer, "Translation of a DDL digital system specification to boolean equation", IEEE Trans. on computers vol. C-18, pp.305-313, 1970.

[3] S.J.Piats, "HDL: A comprehensive H/W design language:, Proc. of the 17th Hawaii International conference on system science 1984, vol. 2, Jan. 1984.

[4] R.E.Swanson, Z.Navabi and F.J.Hill, "An AHPL compiler/simulator system", In Proc. sixth Texas conf. computer system., pp.1-11, 1977.

[5] R.D.Acosta, S.P.Smoth and J.Larson, "Mixed mode simulation of compiled VHDL programs", In Proc. Int. Conf. Computer-Aided Design, pp.180-183, Santa Clara, Nov. 1989.

[6] R.Lipsett, C.Schaefer and C.Ussery, "VHDL: Hardware Description and Design", Kluwer Academic Pub., Norwel, Massachusetts, 1989

[7] Z.Navabi, "VHDL: Analysis and modeling of digital systems", McGraw-Hill, 1994.

[8] Y.C.Hsu, K.F.Tsai, J.T.Liu and E.S.Lin, "VHDL modeling for digital design synthesis", Toppan Kluwer.

[9] 박두열, "APL을 이용한 소형명령 컴퓨터의 설계와 시뮬레이션에 관한 연구", 동아대학교 박사학위논문, 1989.

[10] 박두열, "HDL을 이용한 파이프라인 프로세서의 테스트 벡터구현에 의한 시뮬레이션", 한국OA학회 논문지, 제5권 제3호, pp.16-28, 2000, 9.

부록A (Behavioral Description)

```
LIBRARY cmos;
Use cmos.basic_utilities
PACKAGE par_utilities IS
  FUNCTION "XOR"(a, b : qit) RETURN qit;
  FUNCTION "AND"(a, b : qit_vector) RETURN qit_vector;
:
:
  FUCTION sub_cv (a, b : qit_vector; cin : qit) RETURN qit_vector;
  FUCTION set_if_zero (a : qit_vector; cin : qit) RETURN qit
END par_utilities

PACKAGE BODY par_utilities IS
  FUNCTION "XOR"(a, b : qit) RETURN qit IS
    CONSTANT qit_or_table : qit_2d := (('0','1','1','X'),
      ('1','0','0','X'), ('1','0','0','X'), ('X','X','X1','X'));
  BEGIN RETURN qit_or_table (a, b);
  END "XOR"
  FUNCTION "AND"(a, b : qit_vector) RETURN qit_vector IS
:
:
  FUNCTION "NOT"(a : qit_vector) RETURN qit_vector IS
    VARIABLE r : qit_vector (a'RANGE);
  BEGIN
:
:
     ELSE r(a'LEFT+2) := '0'; END IF;
    REURN r;
  END add_cv;
  FUNCTION "sub_cv"(a, b : qit_vector; cin, qit) RETURN qit_vector
:
:
:
      RETURN zero;
  END set_if_zero;
END par_utilities;

LIBRARY cmos;
USE comos.basic_utilities.ALL;
PACKAGE par_parameters IS
  CONSTANT single_byte_instructions : qir_vector (3 DOWNTO 0) :=
"1110"
:
:
  CONSTANT add : qit_vector (2 DOWNTO 0) := "010";
  CONSTANT sub : qit_vector (2 DOWNTO 0) := "011";
END par_parameters;

LIBRARY cmos;
USE comos.basic_utilities.ALL;
LIBRARY par_library;
:
:
      read_mem, write_mem : OUT qit;
      databus : INOUT wired_byte BUS := "ZZZZZZZZ"; adbus :
OUT welve
      );
END par_central_processing_unit;

VARIABLE pc : twelve;
VARIABLE ac byte1, byte2 : byte;
VARIABLE v, c, z n : qit;
VARIABLE temp : qit_vector (9 DOWNTO 0);
```

```
pc := zero_12;
Wait FOR cycle_time;

adbus <= pc;
read_mem <= '1' ; WAIT FOR read_high_time;
byte1 := byte(databus);
read_mem <= '0'; WAIT FOR read_low_time;c := zero_12;
WAIT FOR cycle_ime;

CASE byte1 (3 DOWNTO 0) IS
  WHEN cla =>
    ac := zero_8;
  WHEN cma =>
:
:
    IF ac = zero_8 THEN z := '1' END IF;
    n := ac(7);
  WHEN OTHERS => NULL;
END CASE;

adbus <= pc;
read_mem <= '1'; WAIT FOR read_high_time;
byte2 := byte(databus);
read_mem <= '1'; WAIT FOR read_low_time;
pc := inc(pc);

databus <= wired_byte (pc (7 DOWNTO 0) );
adbus (7 DOWNTO 0) <= byte2;
write_mem <= '1'; WAIT FOR write_high_time;
write_mem <= '0'; WAIT FOR write_low_time;
databus <= "ZZZZZZZZ";
pc (7 DOWNTO 0) := inc(byte2);

IF
  (byte1(3) = '1' AND v = '1') OR
  (byte1(2) = '1' AND c = '1') OR
  (byte1(1) = '1' AND z = '1') OR
  (byte1(0) = '1' AND n = '1') OR
THEN
  pc (7 DOWNTO 0) := byte2
END IF;

adbus (11 DOWNTO 8) <= byte1 (3 DOWNTO 0);
adbus (7 DOWNTO 0) <= byte2;
read_mem <= '1' WAIT FOR read_high_time;
byte2 := byte(databus;
read_mem ,= '0'; WAIT FOR read_low_tume;

pc := byte1(3 DOWNTO 0) & byte2;

adbus <= byte1 (3 DOWNTO 0) & byte2;
databus <= wired_byte (ac);
write_mem <= '1' WAIT FOR write_high_time;
write_mem ,= '0'; WAIT FOR write_low_tume;
databus <= 'ZZZZZZZZ';

adbus (11 DOWNTO 8) <= byte1 (3 DOWNTO 0);
adbus (7 DOWNTO 0) <= byte2;
read_mem <= WAIT FOR read_high_time;
CASE byte1 (7 DOWNTO 5) IS
:
:
IF ac = zero_8 THEN z := '1' END IF;
n := ac(7);
read_mem <= '1' WAIT FOR read_low_time;
```

부록B (Dataflow Description)

```
USE cmos.basic_utilities.ALL;
PACKAGE alu_operations IS
  CONSTANT a_and_b : qit_vector (2 DOWNTO 0) := "000";
  CONSTANT b_compl : qit_vector (2 DOWNTO 0) := "001";
  CONSTANT a_input : qit_vector (2 DOWNTO 0) := "100";
  CONSTANT a_add_b : qit_vector (2 DOWNTO 0) := "101";
  CONSTANT b_input : qit_vector (2 DOWNTO 0) := "110";
  CONSTANT a_sub_b : qit_vector (2 DOWNTO 0) := "111";
END alu_operations;

Entity arithmetic_logic_unit IS
  PORT (a_side, b_side : IN byte; code: IN qit_vector (2 DOWNTO
0);
      in_flags: IN nibble; z_out: OUT byte; out_flags: OUT nibble);
END airthmetic_logic_unit;
--
ARCHITECTURE behavioral OF arithmetic_logic_unit IS
BEGIN
  coding: PROCESS(a_side, b_side, code)
    VARIABLE t: qit_vector(9 DOWNTO 0);
    VARIABLE v, c, z, n: qit;
    ALIAS n_flag_in: qit IS in_flags(0);
    ALIAS z_flag_in: qit IS in_flags(1);
    ALIAS c_flag_in: qit IS in_flags(2);
    ALIAS v_flag_in: qit IS in_flags(3);
  BEGIN
    CASE code IS
      WHEN a_add_b =>
        t:= add_cv(b_side, a_side, c_flag_in;
        c:= t(8); v:= t(9);
:
:
     END CASE;
    n:= t(7);
    z:= set_if_zero(t);
    z_out <= t(7 DOWNTO 0);
    out_flags <= v & c & z & n;
  END PROCESS coding;
END behavioral;

Entity shifter_unit IS
  PORT (alu_side: IN byte; arith_shift_left,
        arith_shift_right: IN qit; in_flags: IN nibble;
        obus\kern1pt_side: OUT byte; out\kern1pt_flags: OUT nibble);
END shifter\kern1pt_unit;
--
ARCHITECTURE behavioral OF shifter_unit IS
BEGIN
:
:
   ALIAS v_flag_in: qit IS in_flags(3);
 BEGIN
    IF arith_shift_right = '1' AND arith_shift_left = '0' Then
      t:= alu_side (7 DOWNTO 0);
:
:
      v:= v_flag_in;
    ENDIF;
    obus_side <= t;
    out_flags <= v & c & z & n;
  END PROCESS coding;
END behavioral;

ENTITY status_register_unit IS
  PORT (in_flags: IN nibble; out_status: OUT nibble; load,
  cm_carry, ck: IN qit);
END status_register_unit;

ARCHTECTURE behavioral OF status_register_unit IS
BEGIN
  PROCESS (ck)
    VARIABLE internal_state: nibble:= "0000";
    ALIAS internal_c: qit IS internal_state(2);
  BEGIN
    IF (ck = '0') THEN
```

```
      END BLOCK clocking;
    END BLOCK enable;
  END dataflow;

  ENTITY program_counter_unit IS
    PORT (i12: IN byte; o12: OUT twelve; increment, load_page,
load_offset, reset, ck: IN qit);
  END program_counter_unit;
  ARCHTECTURE behavioral OF program_counter_unit IS
  BEGIN
    PROCESS (ck)
      VARIABLE internal_state: twelve := zero_12;
  :
  :
       o12 <= internal_state;
      END F;
    END PROCESS;

  ENTITY program_counter_unit IS
    PORT (i12: IN byte; o12: OUT twelve; increment, load_page,
load_offset, reset, ck: IN qit);
  END program_counter_unit;
  ARCHTECTURE behavioral OF program_counter_unit IS
  BEGIN
    PROCESS (ck)
  :
  :
       o12 <= internal_state;
      END IF;
    END PROCESS;
  END behavioral;

  ENTITY par_data_path IS
    PORT (databus: INOUT wired_byte BUS:= "ZZZZZZZZ";
          adbus: OUT twelve; clk: IN qit; load_ac, zero_ac,
          load_ir, increment_pc, load_page_pc, reset_pc,
  :
  :
          ir_lines: OUT byte; status: OUT nibble;
          );
  END par_data_path;

  ARCHITECTURE structural OF par_data_path IS
    COMPONENT ac
      PORT (i8: IN byte; o8: OUT byte; load, zero ck: IN qit);
    END COMPONENT;
    FOR r1: ac USE ENTITY WORK.accumulator_unit (dataflow);
    COMPONENT ir
      PORT (i8: IN byte; o8: OUT byte; load, zero ck: IN qit);
  :
  :
     SIGNAL shu_glags, ar_out: nibble; mar_bus:wired_twelve BUS;
          mar_inp: twelve

  BEGIN
    -- bus connection
       dbus1: alu_a_inp <= qit_vector (dbus);
  :
  :
          mar_bus (11 DOWNTO 8) <= GUARDED wired_qit_vector
    (ir_out (3 DOWNTO 0));
          END BLOCK ir2;
  r3: pc PORT MAP (mar_out, pc_out, increment_pc, load_offset_pc,
reset_pc, clk);
        pc1: BLOCK (pc_on_mar_page_bus = '1')
  :
  :
     12: shu PORT MAP (alu_out, arith_shift_left,
    arith_shift_right, alu_flags, obus, shu_flags);
  END structural;

  BEGIN
    -- implied or assignments to output signals.
    load_ac <= load_ac_oi;
  :
  :
    read_mem <= read_mem_oi;
    write_mem <= write_mem_oi;
    alu_code <= qit_vector (alu_code_oi);

  s1: BLOCK (s(1) = '1')
  BEGIN  -- start of fetch.
    -- pc to mar
    pc_on_mar_page_bus_oi <= GUARDED '1';
  :
  :
      s(2) <= GUARDED '1'' WHEN interrupt /= '1' ELSE '0';
    END BLOCK ck;
  END BLOCK s1;

  s2: BLOCK (s(2) = '1')
  BEGIN  -- fetchin continue
    -- read memory into ir
  :
  :
    ck: BLOCK ((clk = '0' AND NOT clk'STABLE) AND GUARD)
    BEGIN
      s(3) <= GUARDED '1'
    END BLOCK ck;
  END BLOCK s2;

  s3: BLOCK (s(3) = '1')
  BEGIN
    -- pc to mar, for next read
    pc_on_mar_page_bus_oi <= GUARDED '1';
    pc_on_mar_offset_bus_oi <= GUARDED '1';
  :
  :
      BEGIN
        s(2) <= GUARDED '1';
      END BLOCK ck;
    END BLOCK sb;
  END BLOCK s3;

  s4: BLOCK (s(4) = '1')
  BEGIN -- page from ir, offset from next memory makeup 12 -- bit
address
    -- read memory into mar offset
    mar_on_adbus_oi <= GUARDED '1';
    read_mem_oi <= GUARD '1' AFTER ready_delay;
  :
  :
      END BLOCK ck;
    END BLOCK sp;
    -- increment pc
    increment_pc_oi <= GUARDED '1';
  END BLOCK s4;
    END BLOCK sb;
  END BLOCK s3;

  s5: BLOCK (s(5) = '1')
  BEGIN -- indirect addressing
    -- read actual operand from memory into mar offset
    mar_on_adbus_oi <= GUARDED '1';
  :
  :
      -- goto 1
      ck: BLOCK ((clk='0' AND NOT clk'STABLE) AND GUARD)
      BEGIN
        s(1) <= GUARDED '1';
      END BLOCK ck;
    END BLOCK st;
  rd: BLOCK (ir_lines(7)= '0') AND GUARD)
    BEGIN
      -- mar on dbus, read memory for operand, perform  -- operation
      mar_on_adbus_oi <= GUARDED '1';
      read_mem_oi <= GUARDED '1' AFTER ready_delay;
  :
  :
      ck: BLOCK ((clk='0' AND NOT clk'STABLE) AND GUARD)
      BEGIN
        s(1) <= GUARDED '1';
      END BLOCK ck;
    END BLOCK rd;
```

```
END BLOCK s6;

s7: BLOCK (s(7) = '1')
BEGIN -- jsr
  -- write pc offset to top of subroutine
  mar_on_adbus_oi <= GUARDED '1';
  pc_offset_on_dbus_oi <= GUARDED '1';read_delay;
:
:
    s(8) <= GUARDED '1';
  END BLOCK ck;
END BLOCK s7;

s8: BLOCK (s(8) = '1')
BEGIN
  -- increment pc
  increment_pc_oi <= GUARDED '1';
  -- goto 1
  ck: BLOCK ((clk= '0' AND NOT clk'STABLE) AND GUARD)
:
:
  BEGIN
    s(1) <= GUARDED '1'
  END BLOCK ck;
END BLOCK s9;

ARCHITECUTRE dataflow OF par_central_processing_unit IS
  COMPONENT par_data_path
  PORT (databus: INOUT wired_byte; adbus: OUTtwelve;
        clk: IN qit; load_ac, zero_ac, ...
:
:
         ir_lines, status);
  ctrl: par_control_unit PORT MAP
        (clk, load_ac, zero_ac, ... alu_code, ir_lines,
         status, read_mem, write_mem, interrupt);
END dataflow;
```

저 자 소 개

박 두 열

1980.2.: 동아대학교 공과대학 전자공학과 졸업

1982.2.: 동아대학교 대학원 공학석사

1989.8.: 동아대학교 대학원 공학박사

1985.3~ : 동주대학 교수 재직

관심분야: · Computer H/W Descriptions.
· Computer Visions.
· Multimedia contents.