VHDL 을 이용한
기본 컴퓨터의 설계

(DIGCOM-A1.2)

2018-01-10

M.Morris Mano 의 컴퓨터 시스템 구조(Computer System Architecture) 에서는 기본 컴퓨터의 설계를 통해서 컴퓨터의 동작원리를 가장 이해하기 쉽게 설명하고 있다. DIGCOM-A1.2 는 일반적인 디지털 논리회로를 구현할 수 있는 실험 키트이며, 기본 컴퓨터가 실행될 수 있도록 개발되어 기본 컴퓨터에서 사용할 수 있는 입출력을 모두 가지고 있다. 따라서 DIGCOM-A1.2 는 기본 컴퓨터와 동일하게 동작하도록 구현된 컴퓨터 설계 실습 키트로도 사용될 수 있다. 컴퓨터 시스템 구조 과목의 목적은 컴퓨터 구조를 이해하고 컴퓨터가 프로그램 명령어를 실행시키는 과정을 이해하는 데 있다. 따라서 컴퓨터 구조를 이해하는데 가장 좋은 방법은 기본 컴퓨터 설계하는 것이다. 본 문서는 최종적으로 기본 컴퓨터를 설계하기 위한 과정으로 구성되어 있으며 설계를 시작하기 전에 기본 컴퓨터를 이해하기 위해 어셈블리 언어로 작성된 프로그램을 실행하고 디버깅하는 방법, 레지스터 전달과 산술논리 연산장치를 VHDL 로 구현하며, 이미 구현된 기본 컴퓨터에서 어셈블리 언어 프로그래밍 하고 실행하는 방법과 새로운 명령어를 추가하기 위해 제어부의 VHDL 코드를 수정해서 필요한 신호를 발생하는 실험으로 구성되었다. 마지막으로 기본 컴퓨터를 설계하는 과정을 설명한다. 본 문서에서 설명하는 모든 실습은 DIGCOM-A1.2 에 다운로드하여 기본컴퓨터의 동작을 확인할 수 있으나 키트가 없어도 시뮬레이션을 이용해서 동작을 확인할 수 있도록 하였다.

본 교재에서 나오는 예제를 실습하기 위해서는 Quartus Prime Lite 15.1 버전을 사용할 것을 권장한다. 15.1 버전까지는 소프트웨어에서 제공하는 University Program 에 의해 waveform 시뮬레이션을 지원하지만 그 이후의 버전에서는 내장된 시뮬레이션 기능이 없으며, 별도로 ModelSim 에서 시뮬레이션을 수행해야 한다.

목차

목차 **3**

1. 기본 컴퓨터의 실행 **1**

1.1. 어셈블리 언어 프로그래밍과 실행 2

1.1.1. 프로그램 작성 및 어셈블 2

1.1.2. BDF/Schematic 설계의 컴파일 3

1.2. 프로그램의 실행 4

1.2.1. 프로그램 모드(RUN Mode) 실행 방법 5

1.2.2. 명령어 모드(INSTruction Mode) 실행 방법 6

1.2.3. 클럭 모드(CLocK Mode) 실행 7

1.2.4. 입출력 명령 실행 8

1.2.5. 인터럽트 프로그램의 실행 10

1.2.6. 시뮬레이션에 의한 동작 확인 10

1.3. 예제 어셈블리 언어 프로그램 13

1.3.1. 두 수를 가산하는 프로그램(ex1.s) 13

1.3.2. 두 수를 감산하는 프로그램(ex2.s) 14

1.3.3. 10 개의 수를 연속적으로 더하는 프로그램(ex3.s) 15

1.3.4. 두 개의 배정밀도를 가산하는 프로그램(ex4.s) 16

1.3.5. 서브루틴 사용을 보이는 프로그램(ex5.s) 17

1.3.6. 파라미터 링키지를 보이는 프로그램(ex6.s) 18

1.3.7. 데이터 블록을 이동시키는 프로그램(ex7.s) 19

1.3.8. 한 문자를 입출력 시키는 프로그램(ex8.s) 21

1.3.9. 연속적으로 출력을 시키는 프로그램(ex9.s) 22

1.3.10. 간단한 인터럽트 프로그램(ex10.s) 23

2. 컴퓨터 구조 실습 **26**

2.1. 레지스터 전달 표현(Register Transfer Statement) 설계 26

2.2. Arithmetic Logic Shift Unit(ALSU)의 VHDL 설계 33

2.3. 마이크로 연산과 제어 함수 40

2.4. 프로그램, 명령어 및 마이크로 연산의 이해 50

2.5. 어셈블리 언어 프로그래밍 63

2.6. 새로운 명령어의 추가 67

3. 기본 컴퓨터의 설계 77

3.1. 리셋(reset) 회로 78

3.2. SC(Sequence Counter)부 79

3.2.1. 클럭의 사용 79

3.2.2. T 클럭 79

3.2.3. 메모리 클럭 81

3.3. 제어부 81

3.3.1. HLT 명령의 실행 82

3.3.2. 플립플롭 제어신호 발생 82

3.3.3. 레지스터 제어신호 발생 83

3.3.4. 레지스터 선택 신호 발생 84

3.3.5. 가산 논리 제어신호 85

3.3.6. 메모리 제어신호 86

3.4. 명령어 디코더 부 87

3.5. 산술연산부 87

3.5.1. E 플립플롭 값 87

3.5.2. 산술논리연산 88

3.6. 레지스터 부 89

3.6.1. 레지스터 값 변경 89

3.6.2. 메모리 읽기 동작 90

3.7. 입출력 부 91

3.7.1. 키 스위치 입력부 91

3.7.2. FGO 93

3.7.3. 7- 세그먼트 출력 96

3.7.4. 인터럽트 플립플롭, 인터럽트 인에이블 플립플롭 96

3.8. 메모리 97

3.8.1. 메모리의 생성 97

3.8.2. 메모리 읽기/쓰기 동작 101

3.9. 레지스터 출력 102

1. 기본 컴퓨터의 실행

컴퓨터의 동작을 이해하기 위해 가장 좋은 방법은 프로세서의 어셈블리 언어를 이해하는 것이다. 프로세서를 설계하는 가장 첫 단계는 프로세서가 명령어를 정의하는 것이며, 프로세서는 그 명령어를 실행할 수 있도록 설계된다. 따라서 사용자는 어셈블리 언어를 통해서 가장 가깝게 컴퓨터 프로세서의 동작을 확인할 수 있다. 어셈블리 언어 프로그램은 이 명령어들의 집합이며 이 명령어의 실행 결과를 이해 한다면 쉽게 프로세서가 동작하는 원리를 좀 더 쉽게 접근할 수 있다. 이 장에서는 기본 컴퓨터에서 어셈블리 언어로 프로그램을 작성하고 어셈블 한 후 DIGCOM-A1.2 에 다운로드하여 실행할 수 있는 방법을 설명하고, 실행할 수 있는 예제 프로그램에 대해서 설명을 하였다. 본 장에 나오는 예제 프로그램들은 이미 구현된 기본 컴퓨터와 함께 DIGCOM-A1.2 에 다운로드하여 실행시킬 수 있으며, 프로그램 단위, 명령어 또는 한 클럭에 실행되는 마이크로 연산 단위로 실행을 할 수 있다.

1.1. 어셈블리 언어 프로그래밍과 실행

기본 컴퓨터를 실행시키기 위해서는 어셈블리 언어로 프로그래밍을 한 후 어셈블러를 이용해서 어셈블을 하여 mif 파일을 생성한다. 이 mif 파일은 어셈블리 프로그램의 기계어 코드이며 기본컴퓨터의 메모리 초기화 데이터로 사용된다. mif 파일로 초기화된 메모리는 VHDL 과 schematic 으로 설계된 기본컴퓨터에 포함이 되며, 기본컴퓨터는 Quartus Prime Software 컴파일러에 의해 컴파일되어 DIGCOM-A1.2 에 다운로드되는 pof 파일을 생성한다. [그림 1-1]은 기본컴퓨터의 다운로드 파일인 pof 파일이 생성되는 과정을 보여준다.

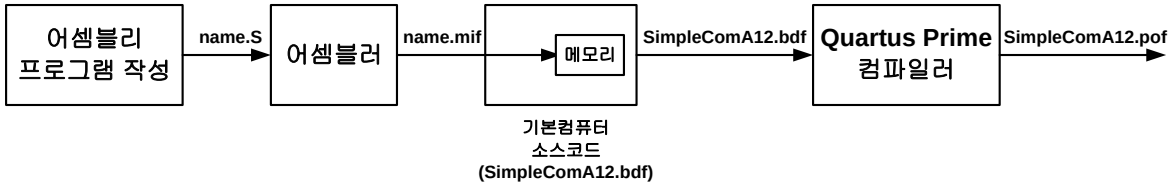


그림 1-1. 기본컴퓨터의 실행과정

1.1.1. 프로그램 작성 및 어셈블

어셈블러는 AsmCompiler.exe 는 Assembler 폴더에 저장되어 있다. AsmCompiler.exe 를 실행시키고 아래 [그림 1-2]와 같이 편집기로 프로그램을 작성한 후 확장자를 s 로 저장한 후, Assembler 메뉴의 Assembler ➔ Assemble 을 클릭하거나 또는 어셈블 아이콘( )을 클릭하여 어셈블 한 후 확장자 mif 로 저장한다.

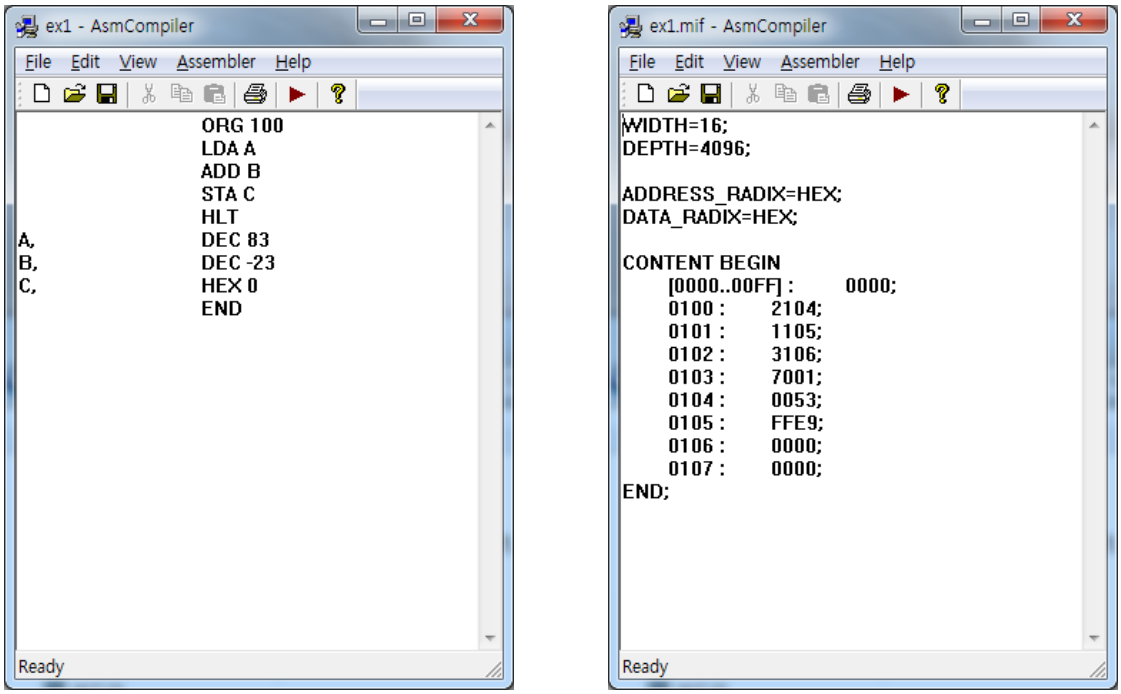


그림 1-2. 프로그램 작성 한 화면과 어셈블 한 결과

[그림 1-2]에서 왼쪽은 편집기로 작성한 어셈블리 언어 프로그램의 소스이고 오른쪽은 어셈블 한 결과 2 진 코드를 mif(Memory Initialization File) 파일 형식으로 저장한 결과이다. 소스 프로그램을 어셈블 하면 mif 파일이 생성되며, mif 파일은 ALTERA 디바이스에서 메모리에 프로그램을 입력하기 위한 파일 형식이다.

1.1.2. BDF/Schematic 설계의 컴파일

어셈블 된 mif 파일을 BDF/Schematic 설계가 있는 DIGCOM-A1.2₩VHDL₩SimpleComputer₩DIGCOM-A1.2-step 폴더로 이동한 후 simplecomA12 프로젝트를 [그림 1-3]과 같이 오픈한다.

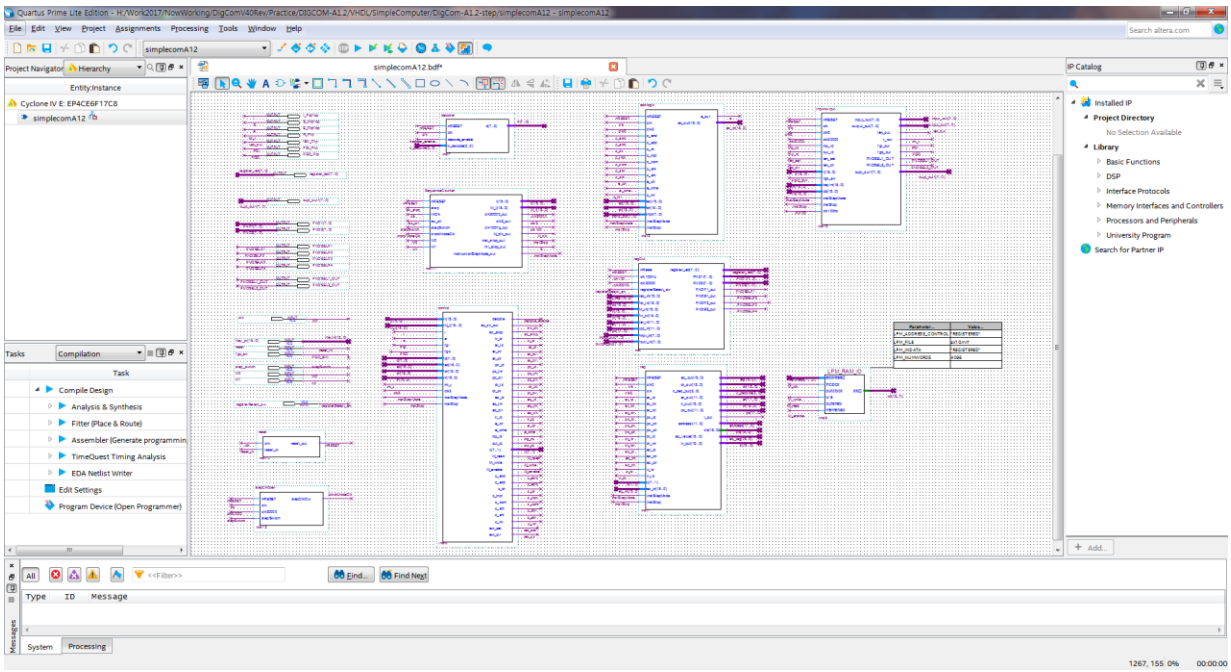


그림 1-3. SimpleComA12 의 BDF/Schematic 설계

[그림 1-3]에서 LPM_RAM_IO 의 Parameter Value 상자를 더블 클릭한 한 후 LPM_FILE 을 어셈블 된 mif 파일명으로 수정한다.

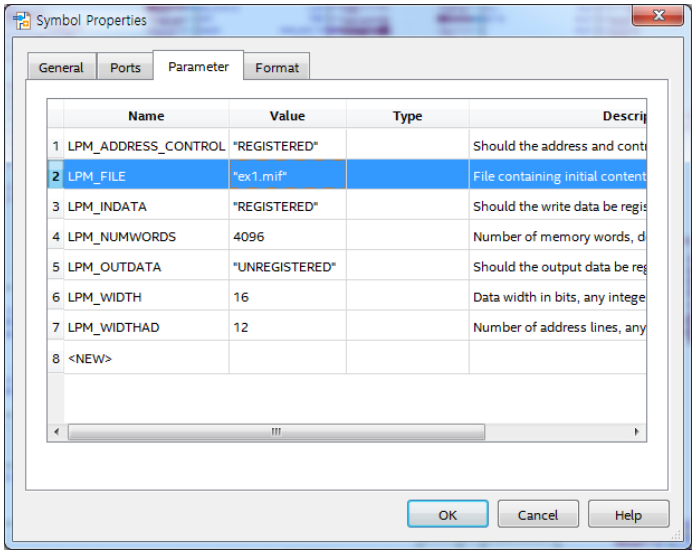


그림 1-4. BDF/Schematic 설계에서 메모리 초기화 파일의 지정

[그림 1-4]와 같이 초기화 파일을 지정한 후 다시 컴파일 한 후 DIGCOM-A1.2 에 다운로드하여 실행시킨다

1.2. 프로그램의 실행

앞에서 설명한 방법에 의해 생성된 pof 파일은 교재 "디지털시스템 설계 및 실습"의 "2.4 FPGA 에 다운로드"에서 설명한 방법에 의해 DIGCOM-A1.2 에 다운로드하여 실행해야 한다. pof 파일이 다운로드한 프로그램을 실행시키기 위해서는 먼저 실행 모드 선택 스위치를 이용하여 실행 모드를 결정한다. 현재 설정된 실행모드는 실행모드 스위치의 위치에 의해 확인할 수 있다. 프로그램을 처음부터 다시 실행시키기 위해서는 리셋스위치를 누르며 모든 프로그램은 100 번지부터 실행된다. 그러나 메모리의 내용을 변경하는 프로그램의 경우에는 리셋스위치를 눌러도 메모리의 값이 초기화가 되지 않으므로 전원을 껐다가 다시 켜야 한다. DIGCOM-A1.2 의 각 부분에 대한 설명은 "디지털시스템 설계 및 실습"의 "부록 C DIGCOM 키트"에 설명되어 있다. 다음은 실행모드를 정하고 DIGCOM-A1.2 에서 기본 컴퓨터를 실행시키는 방법에 대한 설명이다. 프로그램을 실행시키는 방법은 프로그램 모드, 명령어 모드 및 클럭모드 실행이 있으며 모드의 선택은 [그림 1-5]의 실행모드 스위치의 위치에 의해 결정된다. 실행모드는 실행모드 스위치가 "11"일 때 "RUN Mode", "10"일 때 "INSTruction Mode", "01"일 때 "CLocK Mode", 그리고 "00"일 때 "NOPeration"을 각각 나타낸다. [그림 1-5]는 기본 컴퓨터에서 사용하는 각 장치의 이름을 보여준다.

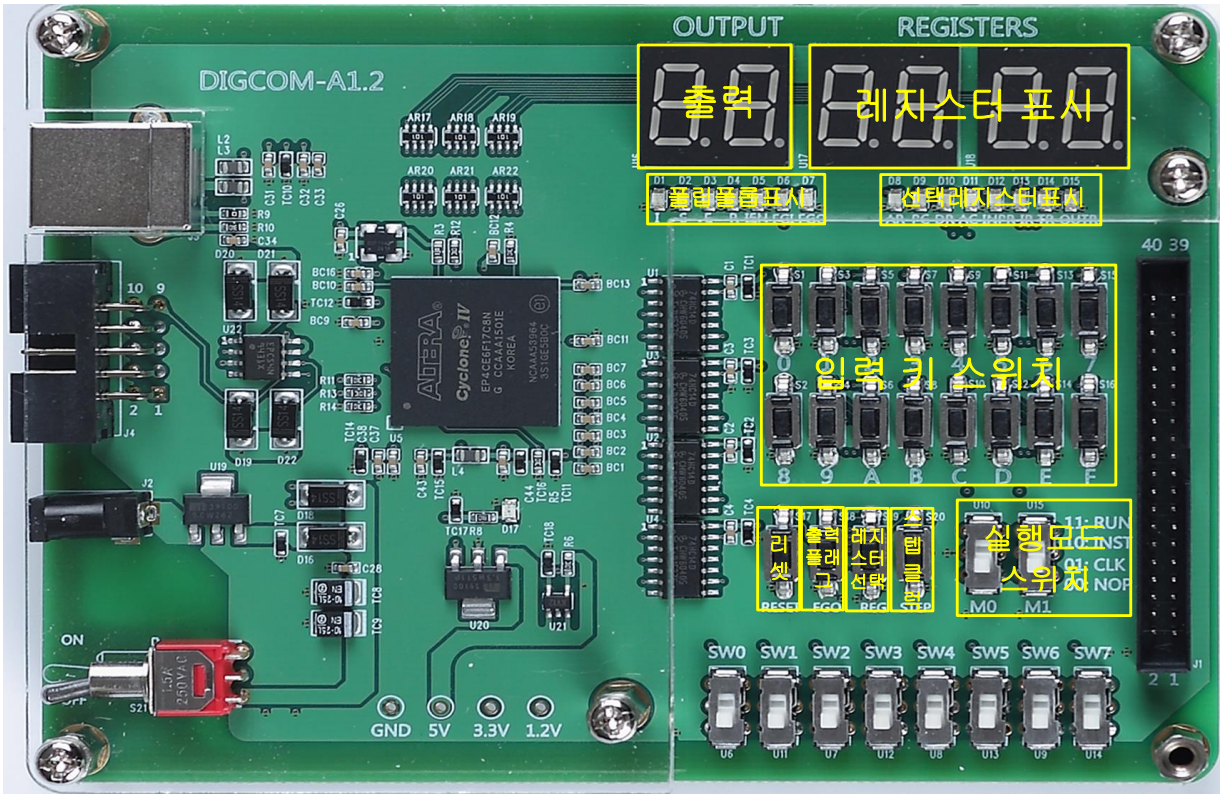


그림 1-5. 기본 컴퓨터에서 사용하는 각 장치의 이름

1.2.1. 프로그램 모드(RUN Mode) 실행 방법

프로그램 모드는 하나의 프로그램을 끝까지 실행하는 "RUN Mode"이다. 다음은 프로그램단위로 실행하는 방법이다.

① pof 파일을 다운로드한 후 [그림 1-5]의 실행모드 스위치를 "11 : RUN Mode"로 한다.

② 리셋 스위치를 누르거나 전원 스위치를 다시 껐다가 키면 프로그램이 처음부터 끝까지 실행된다. 실행결과 출력은 출력 7-세그먼트에서 확인한다.

③ 각 레지스터의 값을 확인하기 위해서는 레지스터선택 스위치를 한번씩 누를 때마다 선택레지스터표시 LED 가 왼쪽에서 오른쪽으로 하나씩 이동하면서 켜지며, 켜진 LED 에 해당하는 레지스터 값이 레지스터 표시 7-세그먼트에 출력된다.

④ 7 개의 플립플롭 값은 7 개의 플립플롭표시 LED 에 각각 출력된다.

실행모드가 프로그램 모드(RUN Mode)인 경우에는 전원이 켜지면 프로그램이 처음부터 끝까지 실행된다. 예제로 사용된 어셈블리 언어 프로그램의 마지막은 모두 HLT 명령어이므로 이 명령어가 실행된 후에 각 레지스터와 플립플롭 값을 유지하게 된다. 다운로드된 pof 파일은 configuration 디바이스(EPCS4)에 저장되므로 반드시 전원을 껐다가 다시 켜야한다. 이 때 EPCS4 의 데이터가 CycloneIV FPGA 로 configuration 되면서 프로그램이 실행된다. 프로그램이 다 실행된 후에 리셋

스위치를 누르면 프로그램이 다시 실행되지만 메모리의 값을 수정하는 프로그램의 경우에는 리셋 스위치에 의해 메모리가 초기화가 되지 않으므로 전원을 껐다가 다시 켜야 한다.

1.2.2. 명령어 모드(INSTruction Mode) 실행 방법

다음은 명령어 단위로 실행하는 방법이다.

① pof 파일을 다운로드한 후 [그림 1-5]의 실행모드 스위치를 "10 : INSTruction Mode"로 한다.

② 리셋 스위치를 누르거나 전원 스위치를 다시 껐다가 키면 프로그램이 실행대기 상태가 된다. 모든 레지스터와 플립플롭은 초기값을 출력한다.

③ 프로그램을 명령어 단위로 실행하기 위해서는 [그림 1-5]의 스텝클럭 스위치를 한번씩 누른다.

④ 스텝클럭 스위치를 한번씩 누를 때마다 하나의 명령어가 실행되며, 각 명령어가 실행된 후에 레지스터와 플립플롭 값을 확인하는 방법은 프로그램 모드에서 확인하는 방법과 같다.

[표 1-1]은 두 수를 더하는 프로그램을 어셈블 했을 때 2 진 코드를 보여주며, [표 1-2]에서는 이 프로그램을 명령어 단위로 실행 했을 때 각 명령이 실행된 다음의 레지스터 및 플립플롭 값을 보여준다. 두 수를 더하는 프로그램에 대한 설명은 "1.3.1 두 수를 가산하는 프로그램(ex1.s)"에서 상세하게 설명한다.

표 1-1. 예제(ex1.s) 프로그램과 어셈블된 2진 코드

프로그램 소스		주소	2 진 코드
	ORG 100		
	LDA A	0100	2104
	ADD B	0101	1105
	STA C	0102	3105
	HLT	0103	7001
A,	DEC 83	0104	0053
B,	DEC -23	0105	FFE9
C,	HEX 0	0106	0000
	END		

표 1-2. 예제(ex1.s) 프로그램의 명령어 단위 실행 결과

레지스터,FF / 명령어	레지스터(HEX)								Flip Flop	출력
	AR	PC	DR	AC	INPR	IR	TR	OUTR	I S E R IEN FGI FGO	
ORG 100	000	100	0000	0000	00	0000	0000	00	0 1 0 0 0 0 1	
LDA A	104	101	0053	0053	00	2104	0000	00	0 1 0 0 0 0 1	
ADD B	105	102	FFE9	003C	00	1105	0000	00	0 1 1 0 0 0 1	

STA C	106	103	FFE9	003C	00	3106	0000	00	0 1 1 0 0 0 1	
HLT	001	104	FFE9	003C	00	7001	0000	00	0 0 1 0 0 0 1	

첫번째 명령어인 "LDA A"는 A 번지의 데이터를 AC(Accumulator)에 로드하는 명령이므로 실행 후에는 AC 에 A 번지 데이터가 저장되어 있어야 한다. 그러므로 [표 1-1]의 첫번째 명령어 실행 후에는 AC 에 A 번지에 저장되어 있던 0x53 이 저장된다. [표 1-2]의 실행결과 플립플롭에서 S 는 HLT 명령어에 의해 '0'이 되나 명령어 단위로 실행할 때는 한 명령이 실행될 때마다 프로그램이 정지하므로 항상 '0' 값을 갖는다.

[표 1-2]는 "1.3 예제 어셈블리 언어 프로그램"에서 "두 수를 가산하는 프로그램(ex1.s)"에 대한 명령어 단위 실행 결과만 보여주고 있으나, 본 문서의 "부록 1. 예제 프로그램 명령어 단위 실행 결과"에서 "1.3 예제 어셈블리 언어 프로그램"에서 설명될 10 개의 예제 프로그램 ex1.s~ex10.s 에 대한 명령어 단위 실행결과를 모두 보여준다.

1.2.3. 클럭 모드(CLocK Mode) 실행

다음은 클럭 단위로 실행하는 방법이다.

① pof 파일을 다운로드한 후 [그림 1-5]의 실행모드 스위치를 "01 : CLK Mode"로 한다.

② 리셋 스위치를 누르거나 전원 스위치를 다시 껐다가 키면 프로그램이 실행대기 상태가 된다. 모든 레지스터와 플립플롭은 초기값을 출력한다.

③ 클럭 단위로 실행하기 위해서는 [그림 1-5]의 스텝클럭 스위치를 한번씩 누른다.

④ 스텝클럭 스위치를 한번씩 누를 때마다 하나의 클럭이 입력되며, 각 클럭 단위로 실행된 후에 레지스터와 플립플롭 값을 확인하는 방법은 프로그램 모드 및 명령어 모드에서 확인하는 방법과 같다.

앞에서 명령어 단위로 실행했던 프로그램의 마이크로 연산과 클럭단위로 실행됐을 때 레지스터와 플립플롭의 값은 다음과 같다.

표 1-3. 예제(ex1.s) 프로그램의 클럭 단위 실행 결과

Instruction	Micro operations	Register & Flip Flops								
		AR	PC	DR	AC	INPR	IR	TR	OUTR	I S E R IEN FGI FGO
ORG 100		0	100	0	0	0	0	0	0	0 1 0 0 0 0 1
LDA A	$R'T_0$: AR←PC	100	100	0	0	0	0	0	0	0 1 0 0 0 0 1
	$R'T_1$: IR←M[AR], PC←PC +1	100	101	0	0	0	2104	0	0	0 1 0 0 0 0 1
	$R'T_2$: D_0, ...,D_7←Decode IR (12-14) AR←IR(0-11), I←IR(15)	104	101	0	0	0	2104	0	0	0 1 0 0 0 0 1
	D'_7IT_3 : AR←M[AR]	104	101	0	0	0	2104	0	0	0 1 0 0 0 0 1
	D_2T_4 : DR←M[AR]	104	101	53	0	0	2104	0	0	0 1 0 0 0 0 1
	D_2T_5 : AC←DR, SC←0	104	101	53	53	0	2104	0	0	0 1 0 0 0 0 1

ADD B	$R'T_0$: AR←PC	101	101	53	53	0	2104	0	0	0 1 0 0 0 0 1
	$R'T_1$: IR←M[AR], PC←PC +1	101	102	53	53	0	1105	0	0	0 1 0 0 0 0 1
	$R'T_2$: D_0, ...,D_7←Decode IR(12-14) AR←IR(0-11), I←IR(15)	105	102	53	53	0	1105	0	0	0 1 0 0 0 0 1
	D'_7IT_3 : AR←M[AR]	105	102	53	53	0	1105	0	0	0 1 0 0 0 0 1
	D_1T_4 : DR←M[AR]	105	102	FFE9	53	0	1105	0	0	0 1 0 0 0 0 1
	D_1T_5 : AC←AC+DR, E←Cout, SC←0	105	102	FFE9	3C	0	1105	0	0	0 1 1 0 0 0 1
STA C	$R'T_0$: AR←PC	102	102	FFE9	3C	0	1105	0	0	0 1 1 0 0 0 1
	$R'T_1$: IR←M[AR], PC← C +1	102	103	FFE9	3C	0	3106	0	0	0 1 1 0 0 0 1
	$R'T_2$: D_0, ...,D_7←Decode IR (12-14) AR←IR(0-11), I←IR(15)	106	103	FFE9	3C	0	3106	0	0	0 1 1 0 0 0 1
	D'_7IT_3 : AR←M[AR]	106	103	FFE9	3C	0	3106	0	0	0 1 1 0 0 0 1
	D_3T_4 : M[AR]←AC, SC←0	106	103	FFE9	3C	0	3106	0	0	0 1 1 0 0 0 1
HLT	$R'T_0$: AR←PC	103	103	FFE9	3C	0	3106	0	0	0 1 1 0 0 0 1
	$R'T_1$: IR←M[AR], PC←PC +1	103	104	FFE9	3C	0	7001	0	0	0 1 1 0 0 0 1
	$R'T_2$: D_0, ...,D_7←DecodeIR(12-14) AR←IR(0-11), I←IR(15)	1	104	FFE9	3C	0	7001	0	0	0 1 1 0 0 0 1
	rB_0 : S←0	1	104	FFE9	3C	0	7001	0	0	0 0 1 0 0 0 1

예로 첫번째 명령어 "LDA A"의 가장 첫번째 마이크로 연산은 "AR ← PC"이며 PC 의 데이터가 AR 로 전송된다. 따라서 이 마이크로 연산이 실행된 후의 AR 에는 0x100 이 저장되며 이 값은 "AR ← PC"가 실행되기 전에 PC 에 저장되었던 값이다.

1.2.4. 입출력 명령 실행

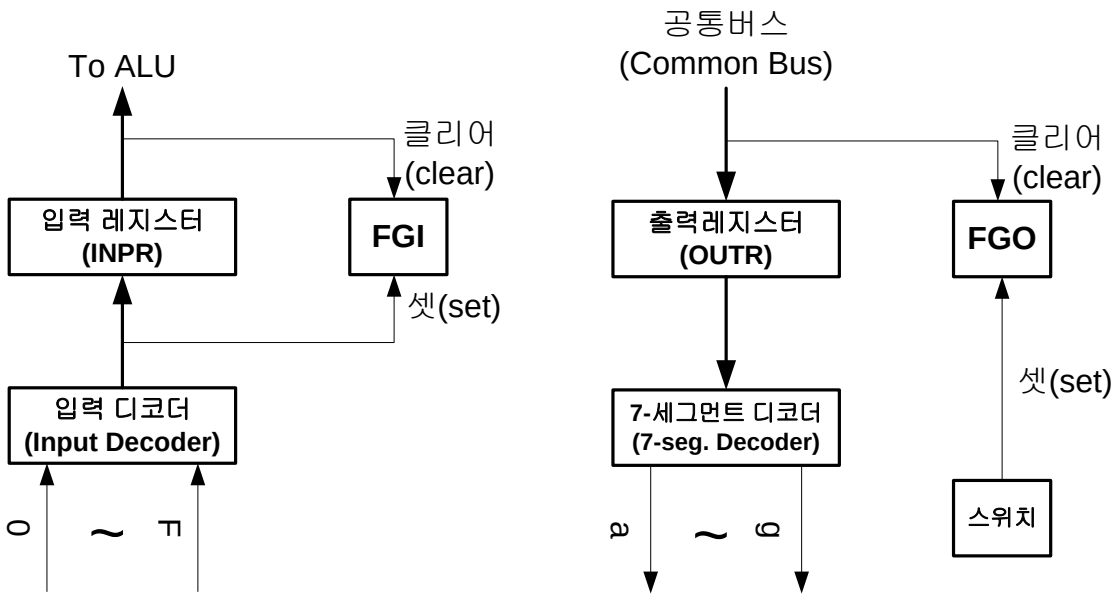


그림 1-6. 입출력 장치의 구조

입출력 동작에서도 실행모드를 정하고 레지스터와 플립 플롭 값을 모니터링하는 과정은 입출력 동작이 없는 경우와 동일하다. 입력을 할 때는 0x0~0xF 까지 값을 갖는 입력 키 스위치를 누르면 INPR(Input Register)에 값이 저장됨과 동시에 FGI 플립플롭이 '1'로 세트되어 INP 명령으로 INPR 에 저장된 값을 AC 로 전달함과 동시에 FGI 를 '0'으로 리셋한다. 출력의 경우에는 FGO 플립플롭을 검사해서 '1'이면 출력할 데이터를 OUTR(Output Register)에 전달함과 동시에 FGO 를 '0'으로 리셋한다. OUTR 에 저장된 값은 7-세그먼트에 출력이 되며 새로운 데이터를 출력하기 위해서는 외부에서 FGO 플립플롭을 '1'로 세트 시켜야 한다. FGO_SET 스위치를 누르면 FGO 플립플롭이 '1'로 세트되어 다음 데이터를 출력할 수 있다.

[표 1-4]는 한 문자를 입출력하는 프로그램을 명령어 단위로 실행했을 때 한 명령이 실행한 후에 각 레지스터와 플립플롭에 저장된 값을 나타낸다. 한 문자를 입출력하는 프로그램에 대한 설명은 "1.3.8 한 문자를 입출력 시키는 프로그램(ex8.s)"에서 상세하게 설명한다

표 1-4. 예제(ex8.s) 프로그램과 어셈블된 2진 코드

프로그램 소스		주소	2 진 코드
	ORG 100		
CIF,	SKI	0100	F200
	BUN CIF	0101	4100
	INP	0102	F800
COF,	SKO	0103	F100
	BUN COF	0104	4103
	OUT	0105	F400
	STA CHR	0106	3108
	HLT	0107	7001
CHR,		0108	0000
	END		

표 1-5. 예제(ex8.s) 프로그램의 명령어 단위 실행결과

레지스터,FF / 명령어	레지스터(HEX)								Flip Flop	출력	순서
	AR	PC	DR	AC	INPR	IR	TR	OUTR	I S E R IEN FGI FGO		
ORG 100	000	100	0000	0000	00	0000	0000	00	0 1 0 0 0 0 1	00	
CIF, SKI	200	101	0000	0000	00	F200	0000	00	1 1 0 0 0 0 1	00	1
BUN CIF	100	100	0000	0000	00	4100	0000	00	0 1 0 0 0 0 1	00	2
Input 'A' (Press 'A' Button)											
CIF, SKI	200	102	0000	0000	0A	F200	0000	00	1 1 0 0 0 1 1	00	3
BUN CIF											
INP	800	103	0000	000A	0A	F800	0000	00	1 1 0 0 0 0 1	00	4
COF, SKO	100	105	0000	000A	0A	F100	0000	00	1 1 0 0 0 0 1	00	5
BUN COF											

OUT	400	106	0000	000A	0A	F400	0000	0A	1 1 0 0 0 0 0 0	0A	6
STA CHR	108	107	0000	000A	0A	3108	0000	0A	0 1 0 0 0 0 0 0	0A	7
HLT	001	108	0000	000A	0A	7001	0000	0A	0 0 0 0 0 0 0 0	0A	8

위의 예제에서 "SKI" 명령을 실행하면서 입력 플래그를 계속해서 검사하며, 입력 스위치 'A'를 누름과 동시에 INPR 에 'A'가 저장되고 입력 플래그가 '1'로 세트되며, "INP" 명령을 실행하여 INPR 에서 AC 로 데이터를 전송한다. 입력된 값을 출력하기 위해 출력 플래그를 검사를 하며, 출력 플래그 FGO 의 초기값은 '1' 이므로 'A'가 OUTR 로 전송이 됨과 동시에 출력장치에 출력이 된다.

1.2.5. 인터럽트[1] 프로그램의 실행

1.3.10 에서 설명될 "간단한 인터럽트 프로그램(ex10.s)은 인터럽트를 사용하여 입출력을 하는 프로그램의 예이다. 인터럽트를 이용한 프로그램을 실행하기 위한 별도의 인터럽트 장치는 없으며, DIGCOM-A1.2 에서 인터럽트 장치는 입력 키 스위치 또는 FGO 스위치(출력 플래그)를 눌렀을 때 인터럽트가 발생한다. 당연히 인터럽트가 발생하기 위해서는 "ION" 명령으로 인터럽트를 인에이블 시켜야 한다. 입력 키 스위치를 눌렀을 때는 INPR 에 입력 데이터가 저장되었음을 알리는 것이며, FGO 스위치를 눌렀을 때는 OUPR 에 다음 데이터를 출력시킬 수 있음을 알린다.

1.2.6. 시뮬레이션에 의한 동작 확인

프로그램이 실행되는 것은 DIGCOM-A1.2 에서 확인할 수 있으나 시뮬레이션으로 동작을 확인할 수도 있다. 시뮬레이션은 디버깅을 할 때에도 유용하게 활용될 수 있다. 프로그램이 제대로 동작하지 않을 때에는 디버깅을 해야 하며 DIGCOM-A1.2 에서 다운로드하여 명령어 모드로 실행을 해서 한 명령을 마친 후에 각 레지스터에 저장된 값을 확인하거나 좀 더 세밀하게 디버깅하기 위해서는 클럭모드로 실행해서 하나의 마이크로연산이 실행된 후의 레지스터 또는 플립플롭을 모니터링함으로써 디버깅이 가능한다. 그러나 또 하나의 유용한 방법은 시뮬레이션으로 프로그램이 실행되는 과정을 추적하는 것으로 마이크로 연산대로 실행되는 과정을 쉽게 볼 수 있다.

앞에서 설명한 두 수를 더하는 프로그램(ex1.s)은 모두 4 개의 명령어도 구성되어 있으며 다시 각 명령은 하나의 클럭에 실행되는 몇 개의 마이크로 연산으로 구성되어 있다. 시뮬레이션 결과를 확인하기 위해서는 우선 하나의 명령어와 그 명령어를 구성하는 마이크로연산을 열거해야 한다. 예로 두수를 더하는 프로그램의 첫번째 명령어 "LDA A"에 대한 마이크로 연산은 [표 1-6]과 같다.

표 1-6. "LDA A"의 마이크로 연산

LDA A	$R'T_0$: AR←PC
	$R'T_1$: IR←M[AR], PC←PC +1

[1] 인터럽트(interrupt)란 프로세서가 프로그램을 실행하는 도중에 현재 실행중인 명령어를 중지하고 어떤 명령어들의 처리를 위해 외부 장치가 발생하는 신호.

	$R'T_2$: D_0, ...,D_7←Decode IR (12-14), AR←IR(0-11), I←IR(15)
	D'_7IT_3 : AR←M[AR]
	D_2T_4 : DR←M[AR]
	D_2T_5 : AC←DR, SC←0

기본 컴퓨터를 시뮬레이션 하기 위해 DIGCOM-A1.2/VHDL/SimpleComputer/DigCom-A1.2-NoStep 폴더의 digcom 프로젝트를 사용한다. 이 프로젝트를 오픈하고 메모리 블록의 LPM_RAM_IO 의 Parameter Value 상자를 더블클릭하여 LPM_FILE 을 ex1.mif 파일명으로 수정한다. 이 때 ex1.mif 파일이 프로젝트 폴더에 포함되어 있어야 한다.

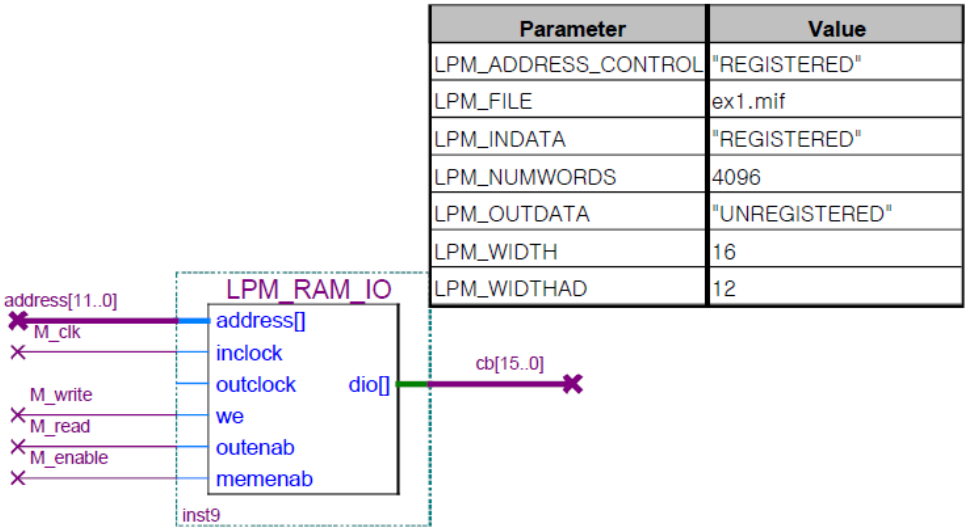


그림 1-7. 메모리 초기화 파일의 지정

그리고 DIGCOM-A1.2 프로젝트를 컴파일을 한 후에 시뮬레이션 파일(ex1.vwf)을 열고 신호와 레지스터의 값을 관찰한다. 시뮬레이션에서 신호 또는 레지스터 값을 보기 위해서는 해당 신호를 출력 포트로 지정한 후에 마이크로 연산에서 영향을 받은 신호 또는 레지스터 값을 관찰하기만 하면 된다. [그림 1-8]은 "LDA A"에 대한 시뮬레이션 결과이다.

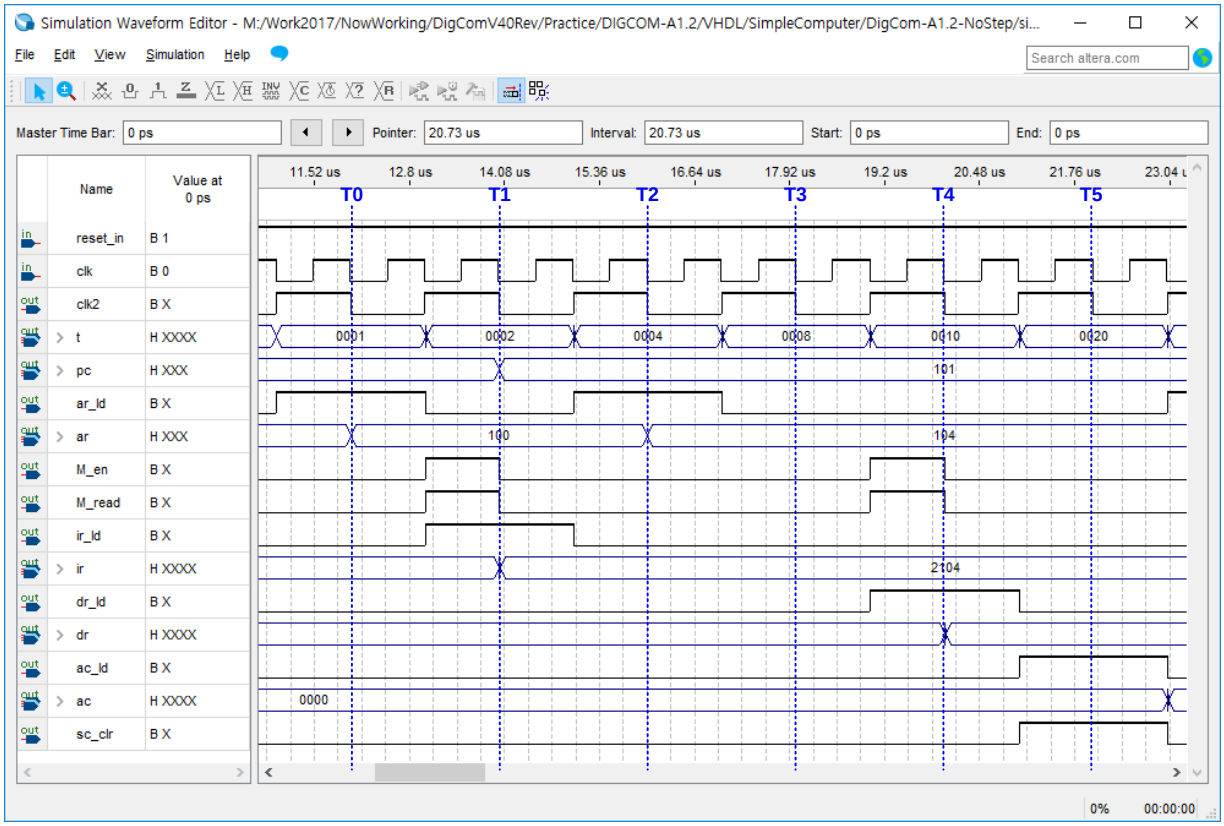


그림 1-8. 명령어 "LDA A"의 시뮬레이션 결과

[그림 1-8]에서 처음의 결과는 T_0 ~ T_2 클럭의 결과이고 두번째 는 T_3 ~ T_5 의 결과이다.

■ $R'T_0$: AR←PC

첫번째 마이크로 연산인 $R'T_0$: AR←PC 는 clk2 클럭의 하강에지에서 발생한다. R'는 인터럽트가 발생하지 않았을 때를 나타내며 시뮬레이션 파형에서 r 이 '0' 값을 유지하고 있고, PC 에 저장되어 있는 0x100 이 clk2 의 하강에지에서 AR 에 전달되는 것을 확인할 수 있다. 이 때 AR 레지스터에 값이 저장되기 위해 로드신호(ar_ld)가 인에이블되어 있는 것을 볼 수 있다.

■ $R'T_1$: IR←M[AR], PC←PC +1

두번째 마이크로 연산은 0x100 번지에 저장된 0x2104 가 IR 레지스터로 전송된다. 0x2104 는 "LDA A"의 2 진코드이며, 메모리의 데이터가 읽혀지기 위해 메모리 인에이블(M_en) 신호와 메모리 읽기(M_read)신호가 인에이블 되어있는 것을 볼 수 있다. 또한 동시에 PC 레지스터가 하나 증가되어 0x101 이 된다. IR 레지스터와 PC 레지스터의 값은 두번째 시뮬레이션 결과에 나타나 있다.

■ $R'T_2$: D_0, ...,D_7←Decode IR (12-14), AR←IR(0-11), I←IR(15)

세번째 마이크로 연산은 IR 레지스터의 12 ~ 14 번 비트는 opcode[2] 이며, LDA 의 opcode 는

[2] opcode 는 명령어의 종류를 결정하는 2 진 코드임. 여기에서는 16 비트 명령어에서 14 번 ~ 12 번의 3 비트에 해당 함.

"010"이므로 디코드 결과는 D_2 이다. 또한 IR 레지스터의 11 ~ 0 번 비트는 유효주소 0x104 를 나타내며 하강에지에서 AR 레지스터에 전달된 것을 볼 수 있다.(두번째 시뮬레이션 결과에서 볼 수 있음.) 그리고 IR 레지스터의 15 번 비트는 간접주소방식을 나타내며 "LDA A"는 직접주소이므로 I 플립플롭(i)에 '0'이 저장된다.

■ D'_7IT_3 : AR←M[AR]

이 마이크로 연산은 간접주소 명령어에서 유효주소를 읽어오기 위한 연산이며 "LDA A"는 직접주소 명령이므로 네번째 T 클럭에서는 아무런 동작도 하지 않는다.

■ D_2T_4 : DR←M[AR]

다섯번째 마이크로 연산은 메모리의 0x104 번지에 있는 피연산자 0x53 이 DR 레지스터에 저장된다.(시뮬레이션 결과에서 0x53 은 보이지 않으나 값이 변경되는 것을 확인할 수 있음.) 이 때 M_en, M_read 및 dr_ld 신호가 인에이블 된다.

■ D_2T_5 : AC←DR, SC←0

마지막으로 DR 에 저장된 0x53 이 AC 로 전달된다. 따라서 AC 레지스터에 저장하기 위해서 인에이블 신호인 ac_ld 가 인에이블 된 것을 볼 수 있다. 동시에 sc_clr 신호가 인에이블 되어 SC(Sequence Counter)가 초기화 되어 다음 명령어를 실행하게 된다.

이와 같은 방법으로 나머지 다른 명령어에 대해서도 보드에서 실행하지 않고 시뮬레이션 결과만으로 동작을 확인할 수 있다.

1.3. 예제 어셈블리 언어 프로그램

어셈블리 언어 프로그램은 크게 입출력이 포함된 경우와 포함되지 않은 경우, 그리고 서브루틴[3]을 사용하는 경우와 사용하지 않는 경우로 구분할 수 있다. 특히 입출력이 포함되어 있는 경우에는 인터럽트를 사용하는 경우와 사용하지 않는 경우로 구분된다. 다음에 나오는 10 개의 프로그램은 어셈블리 언어로 프로그래밍이 가능한 대부분의 경우를 포함하고 있다. DIGCOM-A1.2 에 다운로드하여 실행하는 방법은 교재 "디지털시스템 설계 및 실습"의 "2.2 Quartus Prime Design Software 사용" 설명하였다.

1.3.1. 두 수를 가산하는 프로그램(ex1.s)

■ 프로그램 소스 코드

```
ORG 100        : 프로그램이 0x100번지부터 실행된다.
```

[3] 서브루틴이란 프로그래밍에서 어떤 프로그램이 실행될 때 호출되거나 반복해서 사용되도록 만들어진 일련의 명령어 집합을 말 함. 하나의 서브루틴은 다수의 프로그램에서 사용될 수 있어 프로그래밍 과정에서 다시 작성하지 않아도 됨.

```
        LDA A       : A번지의 데이터(83)을 AC에 로드한다.
        ADD B       : B번지의 데이터(-23)을 AC 더한다.
        STA C       : AC의 값(60, 0x3C)을 C번지에 저장.
        HLT         : HLT 명령
A,      DEC 83
B,      DEC -23
C,      HEX 0
        END
```

■ 프로그램 설명

프로그램은 0x100 번지부터 시작된다. 위 프로그램은 A 번지와 저장된 십진수 83 과 B 번지에 저장된 -23 을 더해서 C 번지에 저장하는 프로그램이다. 프로그램 실행결과는 더한 결과 60(0x3C)이 C 번지에 저장된다. 프로그램 단위로 실행을 할 경우에 계산결과 50 이 C 번지에 저장되는 것을 확인할 수는 없으나 명령어 단위로 실행을 할 때 "ADD B" 명령을 실행한 후에 AC 에 저장된 결과 값을 모니터링을 하여 계산결과를 확인할 수 있다.

1.3.2. 두 수를 감산하는 프로그램(ex2.s)

■ 프로그램 소스 코드

```
        ORG 100     : 프로그램이 0x100번지부터 실행된다.
        LDA SUB     : SUB번지의 데이터(-23)을 AC에 로드한다.
        CMA         : AC의 값을 1의 보수를 취한다.
        INC         : AC의 값을 1 증가하여 2의 보수를 취한다.
        ADD MIN     : MIN번지의 83을 더한다.(83+(-23의 2의 보수) = 83+23)
        STA DIF: 더한 결과를 DIF번지에 저장한다.(106, 0x6A가 저장됨)
        HLT         : HLT 명령
MIN,    DEC 83
SUB,    DEC -23
DIF,HEX 0
        END
```

■ 프로그램 설명

위 프로그램은 MIN 에 저장된 83 에서 SUB 에 저장된 -23 을 빼는 프로그램이다. 그러나 명령어에는 더하는 명령어만 있고 빼는 명령어는 없으므로 -23 을 2 의 보수를 취한 후 십진수 83 에 더해줌으로써 뺄셈을 한다. 2 의 보수를 취하기 위해 "LDA SUB" 명령으로 SUB 번지에 저장된 값을 AC 로 읽어온 후 "CMA" 명령으로 보수를 취한다. 여기에 1 을 더하면 2 의 보수가 된다. 따라서 AC 에 있는 -23 에 대한 2 의 보수에 MIN 번지에 저장된 83 을 더하면 AC 에

106(0x6A)가 저장된다. 마지막으로 이 값을 DIF 번지에 저장한다. 이 프로그램에서도 프로그램 단위로 실행을 하면 결과값을 확인할 수 없으나 "ADD MIN" 명령을 실행한 후 AC 레지스터를 모니터링 하면 결과값을 볼 수 있다.

1.3.3. 10 개의 수를 연속적으로 더하는 프로그램(ex3.s)

■ 프로그램 소스 코드

```
        ORG 100
        LDA ADS     : 더할 수가 저장된 주소 0x150를 AC에 로드
        STA PTR     : PTR(Pointer)번지에 0x150를 저장
        LDA NBR     : 더할 수의 개수(-10)를 AC에 로드
        STA CTR     : CTR(Coujnter)번지에 -10을 저장
        CLA         : AC를 clear
LOP,    ADD PTR I   : PTR번지에서 포인트하는 주소에 있는 operand를 AC에 더함
        ISZ PTR: PTR번지의 수를 하나 증가시킴
        ISZ CTR: CTR번지의 수를 증가시키고 결과가 0이면 다음 명령어를 건너 뜀
        BUN LOP     : 레이블 LOP로 branch
        STA SUM     : SUM번지에 더한 결과를 저장
        HLT         : HLT 명령
ADS,    HEX 150
PTR,    HEX 0
NBR,    DEC -10
CTR,    HEX 0
SUM,    HEX 0

        ORG 150     : operand가 저장된 주소
        DEC 75      : 1st operand
        DEC 65      : 2nd operand
        DEC 23
        DEC 78
        DEC -13
        DEC 9
        DEC 14
        DEC -48
        DEC 24
        DEC 23
        END
```

■ 프로그램 설명

C 와 같은 고급 언어에서는 for 또는 while 과 같이 반복 실행을 쉽게 할 수 있는 방법이 있지만 대부분의 어셈블리 언어에서는 반복 실행을 하게 하는 명령어가 없다. 위 프로그램은 0x150 번지부터 저장되어 있는 10 개의 수를 더하는 프로그램으로 어셈블리 언어로 반복문을 실행하는 방법을 보여준다. ADS 번지에는 더해지는 10 개의 수가 저장된 주소 0x150 이 저장되어 있으며, "LDA ADS" 명령에 의해 AC 로 읽은 후에 "STA PTR" 명령에 의해 PTR 번지에 0x150 을 저장한다. 또한 NBR 번지에는 -10 이 저장되어 있으며 이 숫자는 10 개의 수를 더하기 위해 카운터로 사용되고, " LDA NBR"과 "STA CTR" 명령에 의해 CTR 번지에 저장된다. 덧셈을 하기 전에 먼저 AC 를 클리어 하고, "ADD PTR I" 명령을 실행한다. 이 명령은 간접주소 명령이므로 PTR 번지의 0x150 번지에 저장된 값이 피 연산자가 된다. 따라서 0x150 번지에 저장된 75 를 AC 에 더한다. 그리고 "ISZ PTR"에 의해 PTR 번지의 0x150 을 0x151 로 증가시키고, "ISZ CTR"에 의해 CTR 번지의 -10 을 -9 로 증가시킨 후 "ADD PTR I" 명령이 있는 LOP 으로 점프(BUN LOP)를 한다. 다시 같은 방법으로 0x151 번지의 피 연산자를 AC 에 더하며, 이와 같은 동작을 10 번 연속해서 실행을 하면 "ISZ CTR" 명령에 의해 카운터의 수가 0 이 된다. "ISZ" 명령은 "Increment and Skip if Zero"이므로 CTR 번지의 값이 0 이 되면 다음 명령인 "BUN LOP" 명령을 건너뛰고 다음 명령인 "STA SUM" 명령을 실행하여 AC 에 저장된 10 개의 덧셈결과를 SUM 번지에 저장한다. 결과는 프로그램 모드로 실행하면서 가장 마지막 피 연산자인 23 을 더한 후에 AC 에 저장된 값을 확인함으로써 알 수 있다.

1.3.4. 두 개의 배정밀도를 가산하는 프로그램(ex4.s)

■ 프로그램 소스 코드

```
        ORG 100
        LDA AL      : AL번지의 데이터(0xC186)를 AC에 로드
        ADD BL: BL번지의 데이터(0xA8B5)를 AC에 더함.
        STA CL      : CL번지에 더한 결과(=0x6A3B)를 저장
        CLA         : AC를 clear
        CIL         : AC를 왼쪽으로 한 비트 이동(E비트에 저장된 캐리를 AC에 로드)
        ADD AH      : AH번지의 데이터(0x34A7)를 AC에 로드
        ADD BH      : BH번지의 데이터(0x5CA2)를 AC에 더함.
        STA CH: 더한 결과(=0x914A)를 CH번지에 저장
        HLT
AL,     HEX C186
AH,     HEX 34A7
BL,     HEX A8B5
BH,     HEX 5CA2
CL,
```

```
CH,
        END
```

■ 프로그램 설명

기본 컴퓨터의 데이터는 16 비트이므로 한번에 16 비트를 넘어가는 연산을 할 수 없다. 그러나 16 비트만으로 모든 수를 표현할 수가 없으므로 피 연산자가 16 비트를 넘는 경우에는 몇번에 나누어서 연산을 해야 한다. 기본 컴퓨터에서 32 비트의 연산을 하기 위해서는 연산을 두 번 해야 한다. 위 프로그램에서는 32 비트 피 연산자를 더하는 프로그램이며, 16 비트 덧셈을 두 번해서 32 비트 덧셈을 한다. 이 때 중요한 것은 하위 16 비트 연산결과에서 발생하는 캐리를 상위 16 비트 덧셈에 반영을 해야 한다. 두 개의 피 연산자 A 와 B 는 모두 4 개의 주소 영역에 저장되어 있으며, 각각은 AL, AH, BL 및 BH 이다. 가장 먼저 피 연산자 A 의 하위 16 비트가 저장되어 있는 AL 번지의 0xC186 을 AC 로 읽은 후에("LDA AL") B 의 하위 16 비트가 저장된 BL 번지의 0xA8B5 를 더하고("ADD BL") 이 때 발생하는 캐리는 E 플립플롭에 저장된다. 더한 결과 0x6A3B 는 CL 번지에 저장하고("STA CL") AC 는 클리어 시킨다. "CIL" 명령을 실행하면 AC 레지스터의 각 비트를 왼쪽으로 한 비트씩 이동을 시키며, 이 때 AC(15) 비트는 E 플립플롭으로 이동을 하고 E 플립플롭 값은 AC(0) 비트에 이동된다.

따라서 AL 과 BL 의 덧셈에서 발생한 캐리가 AC 의 0 번 비트로 이동한다. 다음 상위 16 비트를 연산하기 AH 번지의 데이터 0x34A7 을 AC 에 더한다. 결과적으로 하위 16 비트에서 발생한 캐리와 AH 번지의 데이터가 더해진 결과 0x34A8 이 AC 에 저장되며, 다시 BH 번지의 데이터 0x5CA2 를 AC 에 더하면 상위 16 비트의 연산결과 0x914A 가 AC 에 저장된다. 마지막으로 최종 32 비트 덧셈 결과는 "STA CH" 명령에 의해 CH 번지에 저장된다. 결과를 확인하기 위해서는 프로그램 모드에서 실행하면서 "ADD BL" 명령을 실행하면 AC 레지스터 값에 하위 16 비트 결과가 저장되며, "ADD BH" 명령이 실행된 후에 AC 에 상위 16 비트 결과가 저장되므로 이 두값을 확인함으로써 알 수 있다.

1.3.5. 서브루틴 사용을 보이는 프로그램(ex5.s)

■ 프로그램 소스 코드

```
        ORG 100
        LDA X       : X번지의 데이터(0x1234)를 AC에 로드
        BSA SH4     : 서브루틴 SH4를 호출
        STA X       : X번지에 AC의 값을 저장
        LDA Y       : Y번지의 데이터(0x4321)를 AC에 로드
        BSA SH4     : 서브루틴 SH4를 호출
        STA Y       : Y번지에 AC의 값을 저장
        HLT         : HLT 명령어 실행
X,      HEX 1234
Y,      HEX 4321
```

```
SH4,    HEX 0       : 서브루틴 시작 번지
        CIL         : AC를 외쪽으로 4비트 이동
        CIL
        CIL
        CIL
        AND MSK     : AC를 MSK번지 데이터(0xFFF0)와 AND를 실행(상위 12비트만 남김)
        BUN SH4 I   : 서브루틴 리턴
MSK,    HEX FFF0
        END
```

■ 프로그램 설명

위 프로그램은 서브루틴을 사용하는 예를 보여준다. X 번지에 저장된 수 0x1234 를 AC 로 읽은 후에 서브루틴 "BSA SH4" 명령으로 서브루틴을 호출한다. 서브루틴을 호출하면 현재의 PC(Program Counter) 값인 0x102 를 SH4 번지에 저장을 하고 SH4 다음 번지부터 실행을 한다. SH4 번지에 저장된 주소는 서브루틴 실행을 마친 후에 실행이 될 리턴 주소이다. 서브루틴에서는 "CIL" 명령을 4 번 실행을 하고 E 플립플롭 값은 0 이므로 AC 에는 4 비트가 왼쪽으로 시프트된 0x2340 이 저장된다. 다음 MSK 번지에 저장된 HEX FFF0 값과 AND 논리연산("AND MSK")을 실행하므로 4 개의 16 진수 수 중에서 앞 3 자리수만 남고 마지막 16 진수 값은 0 이 된다. 따라서 AC 에는 0x2340 이 저장되고 "BUN SH4 I" 명령을 실행하여 서브루틴으로부터 리턴된다. "BUN SH4 I" 명령은 간접주소 지정방식이며, 이 명령을 실행하면 SH4 번지에 저장된 0x102 번지의 "STA X" 명령부터 실행한다. 이 명령으로 왼쪽으로 4 비트 이동된 0x2340 이 X 번지에 저장되고, 다시 Y 번지의 값 0x4321 을 AC 에 읽은 후에 같은 동작을 반복한다. 결과는 프로그램이 모두 실행된 후에 AC 에 저장된 값을 확인함으로써 알 수 있다.

1.3.6. 파라미터 링키지를 보이는 프로그램(ex6.s)

■ 프로그램 소스 코드

```
        ORG 100
        LDA X       : 0x7B95를 AC에 로드
        BSA OR: 서브루틴 OR를 호출
        HEX 3AF6    : 두번째 operand
        STA Y       :
        HLT         : HLT 명령어
X,      HEX 7B95
Y,      HEX 0
OR,     HEX 0       : 두 번째 피 연산자의 주소(0x102)가 저장 ➔ 리턴주소(0x103) 저장
        CMA         : AC 값을 보수를 취함
```

```
        STA TMP     : TMP번지에 저장
        LDA OR I    : OR번지에 저장된 번지(0x102)의 데이터(0x3AF6)를 AC에 로드
        CMA         : AC 값을 보수를 취함
        AND TMP     : TMP번지 데이터와 AND 실행
        CMA         : 보수를 취함(not((not A) AND (not B)) ➔ A OR B)
        ISZ OR      : OR에 저장된 리턴주소(0x102)를 1 증가 시킴.
        BUN OR I    : 0x103번지로 리턴
TMP,    HEX 0
        END
```

■ 프로그램 설명

서브루틴을 호출할 때는 메인 프로그램에서 서브루틴으로 파라미터를 전달해야 한다. C 언어에서는 호출할 함수의 괄호안에 전달할 파라미터를 넣기만 하므로 매우 간단하지만 어셈블리 언어로 프로그래밍할 때는 다른 방법을 사용해야 한다. ex5 의 예에서는 서브루틴을 호출할 때 전달할 파라미터가 없기 때문에 서브루틴을 호출하기만 하면 됐지만 위 프로그램에서는 파라미터를 전달하는 방법을 보여준다. 기본 컴퓨터의 명령에는 OR 명령이 없으며, 위 프로그램에서는 AND 논리연산과 CMA(Complement AC) 명령을 이용하여 OR 논리연산을 하는 방법을 보여준다. 드모르강 법칙에 의해 (x+y)′ = x′y′ 관계가 성립하므로 OR 연산을 할 두 피연산의 보수를 취한 다음에 AND 연산을 한 후 다시 보수를 취함으로써 OR 연산을 실행한다. 프로그램에서 X 번지에 있는 첫 번째 피 연산자 0x7B95 를 AC 에 읽은 다음에 서브루틴 OR 를 호출한다("BSA OR"). 서브루틴을 호출하면 리턴 주소 0x102 가 OR 번지에 저장되고 OR 다음 번지의 명령 "CMA"부터 실행한다. "CMA" 명령에 의해 AC 에 저장된 첫 번째 피 연산자의 보수가 취해지고(0x846A), TMP 번지에 저장된다("STA TMP"). 다음 명령인 "LDA OR I"는 간접 주소 지정방식이므로 OR 번지에 저장된 0x102 가 유효주소이며, 0x102 번지에 저장된 두 번째 피 연산자인 0x3AF6 을 AC 로 읽는다. 이 값도 첫 번째 피 연산자와 마찬가지로 보수를 취한 후에 TMP 번지의 데이터와 AND 연산을 한다. 연산결과는 (X′ AND Y′) 0x8408 이며, OR 연산의 보수이다. 다시 이 값의 보수를 취한 결과 AC 에는 0x7BF7 가 저장되고 OR 연산의 결과와 동일하다. OR 번지에는 0x102 가 저장되어 있으나 이 주소는 두 번째 피 연산자의 주소이므로 메인 프로그램으로 리턴하기 위해 "ISZ OR" 명령으로 OR 번지의 주소를 증가시키고, "BUN OR I" 명령에 의해 0x103 번지로 리턴하여 "STA Y" 명령을 실행해서 Y 번지에 OR 연산결과를 저장한다. 결과는 프로그램이 모두 실행된 후에 AC 에 저장된 값을 확인함으로써 알 수 있다.

1.3.7. 데이터 블록을 이동시키는 프로그램(ex7.s)

■ 프로그램 소스 코드

```
        ORG 100
        BSA MVE     : MVE번지 서브루틴 호출
        HEX 120     : 옮길 데이터가 저장된 주소
```

```
        HEX 140     : 데이터를 옮길 주소
        DEC -6      : 옮길 데이터 갯수
        HLT         : HLT 명령어
MVE,    HEX 0       : 서브루틴 시작 주소
        LDA MVE I   : MVE번지에 저장된 주소(0x101)에 저장된 값(0x120)을 AC에 로드
        STA PT1     : PT1번지에 0x120를 저장
        ISZ MVE     : MVE번지 데이터를 1 증가시킴(0x102)
        LDA MVE I   : MVE번지에 저장된 주소(0x102)에 저장된 값(0x140)을 AC에 로드
        STA PT2     : PT2번지에 0x140를 저장
        ISZ MVE     : MVE번지 데이터를 1 증가시킴(0x103)
        LDA MVE I   : MVE번지에 저장된 주소(0x103)에 저장된 값(-16)을 AC에 로드
        STA CTR     : CTR번지에 이동할 데이터의 개수(-16)를 저장
        ISZ MVE     : MVE번지 데이터를 1 증가시킴(리턴 주소, 0x104).
LOP,    LDA PT1 I   : PT1번지에 저장된 주소에 저장된 값을 AC에 로드
        STA PT2 I   : AC 값을 PT2에 저장된 주소의 번지에 저장
        ISZ PT1 : PT1 번지의 데이터를 1 증가시킴
        ISZ PT2 : PT2 번지의 데이터를 1 증가시킴
        ISZ CTR: CTR번지 데이터를 1 증가시키고 0이면 다음 명령을 건너뜀.
        BUN LOP     : LOP 번지로 branch
        BUN MVE I   : MVE번지에 저장된 주소(0x104)로 리턴
PT1,
PT2,
CTR,
        ORG 120
        DEC 11
        DEC 22
        DEC 33
        DEC 44
        DEC 55
        DEC 66

        ORG 140
        END
```

■ 프로그램 설명

위의 프로그램은 0x120 번지에 저장된 일련의 데이터를 0x140 번지로 이동하는 프로그램이다. 이를 위해서 이동시킬 데이터의 개수만큼 카운터를 증가시키면서 한 개씩 데이터를 이동시키고

카운터가 0 이 되면 프로그램을 종료한다. 이 프로그램에서도 데이터를 이동시키는 서브루틴을 호출하며 소스 데이터가 저장된 주소, 이동 목적지 주소 및 카운터 값을 파라미터로 전달하기 위해 ex6 에서 사용한 방법을 사용한다.

프로그램이 시작되면 MVE 서브루틴을 호출("BSA MVE")하면서, 현재의 PC 값 0x101 을 MVE 번지에 저장하고 MVE 다음번지의 명령부터 실행한다. 서브루틴에서는 가장 먼저 "LDA MVE I" 명령으로 MVE 번지에 저장된 이동시킬 데이터가 저장된 주소 0x120 를 AC 에 읽고 PT1 에 저장한다("STA PT1"). 그리고 MVE 번지에 저장된 값을 하나 증가(0x101 ➔ 0x102)시킨다. 같은 방법으로 데이터가 이동할 주소 0x140 를 PT2 에 저장하고 이동시킬 데이터의 개수 -6 을 CTR 에 저장한다. 카운터 값을 저장한 후 MVE 번지에 저장된 값을 0x104 로 증가시키며 이 주소가 서브루틴 리턴 주소가 된다. LOP 번지부터 시작되는 반복문이 시작되면 "LDA PT1 I"에 의해 PT1 에 저장된 0x120 번지의 첫 번째 데이터를 AC 로 읽고, "STA PT2 I" 명령으로 PT2 번지에 저장된 0x140 번지에 이 값을 저장한다. 그리고 PT1 과 PT2 에 저장된 0x120 와 0x140 를 각각 하나씩 증가시킴으로써 이동시킬 주소와 저장시킬 주소를 증가시킨다. 다음 카운터 -6 을 하나 증가시키면서 이동시킬 데이터 개수를 하나 감소시키고 LOP 부터 같은 동작을 반복한다. 이와 같은 방법으로 6 개의 데이터를 이동시킨 후에 "ISZ CTR"을 실행하면 CTR 데이터가 0 이 되므로 다음 명령("BUN LOP")을 생략하고 그 다음 명령 "BUN MVE I"를 실행하여 서브루틴에서 리턴한다. 이 때 리턴주소는 MVE 번지에 저장된 0x104 이다. 이 프로그램의 결과는 0x140 번지에 데이터를 다시 읽어서 확인을 해야 하나 메모리 값을 직접 볼 수 있는 방법이 없으므로 프로그램 모드로 실행을 하면서 반복 실행되는 "STA PT2 I" 명령 후에 AC 레지스터 값을 확인하면 0x140 번지에 저장되는 값을 알 수 있다.

1.3.8. 한 문자를 입출력 시키는 프로그램(ex8.s)

■ 프로그램 소스 코드

```
        ORG 100
CIF, SKI        : 입력 플래그를 check해서 '1'이면 INP 명령어 실행
        BUN CIF     : CIF 번지로 branch
        INP         : 문자를 읽어 들임.
COF,    SKO         : 출력 플래그를 check해서 '1이면 OUT 명령어 실행
        BUN COF     : COF 번지로 branch
        OUT         : 문자를 출력
        STA CHR     : CHR 번지에 문자를 저장
        HLT         : HLT 명령어 실행
CHR,
        END
```

■ 프로그램 설명

위의 프로그램은 입출력 동작을 이용하는 프로그램으로써 한 개의 수를 키 스위치에서 읽어서 출력장치에 출력한다. 키 스위치에서 입력된 값을 읽기 위해서는 입력 플래그(Input Flag)를 검사해야 한다. 키 스위치로부터 입력이 있으면 입력 플래그가 세트되고, "SKI" 명령에 의해 다음 명령인 "BUN CIF"을 생략한다. 그리고 다음 명령인 "INP" 명령을 실행하여 INPR 에 입력된 값을 AC 에 저장한다. 만일 입력 플래그가 '1'로 세트되지 않으면 "BUN CIF"에 의해 CIF 번지부터 입력 상태를 검사하는 명령을 반복한다. 다음 입력된 값을 출력하기 위해 "SKO" 명령으로 출력 플래그(Output Flag)를 검사한다. 출력 플래그는 출력장치가 출력할 준비가 되었을 때 '1'로 세트되며, "BUN COF" 명령을 생략하고 그 다음 명령인 "OUT" 명령으로 AC 에 저장된 데이터를 출력한다. 그러나 출력 플래그가 '0'일 때는 "BUN COF" 명령에 의해 COF 번지부터 반복한다. 입력한 데이터를 출력한 다음에 CHR 번지에 출력한 문자를 저장("STA CHR")하고 프로그램을 종료한다. 실행결과는 프로그램이 모두 실행된 후에 출력 7-세그머트에 입력한 값이 출력된다.

1.3.9. 연속적으로 출력을 시키는 프로그램(ex9.s)

■ 프로그램 소스 코드

```
        ORG 100
        LDA NBR     : 출력시킬 데이터 개수
        STA CTR     :
        LDA X       : X번지 데이터를 로드
LOP,    BSA SUB     : AC 값을 출력시키기 위한 서브루틴 호출
        INC         : AC 값을 1 증가시킴.
        ISZ CTR: CTR을 증가시키고 0이면 다음 명령을 건너뜀.
        BUN LOP
        HLT
X,      HEX 10
NBR,    DEC -2
CTR,    HEX 0

SUB,    HEX 0       : 리턴 주소를 저장
LOA,    SKO         : 출력 플래그를 check해서 '1'이면 OUT 명령어 실행
        BUN LOA     : LOA 번지로 branch
        OUT         : 문자를 출력
        BUN SUB I   : 서브루틴 리턴
        END
```

■ 프로그램 설명

위 프로그램은 연속된 두 개의 데이터를 출력시키는 프로그램이다. 먼저 출력한 데이터의 수가 저장된 NBR 번지에서 값 -2 를 읽어서("LDA NBR") CTR 번지에 저장한다("STA CTR"). 그 다음에

출력시킬 데이터가 저장된 X 번지 데이터를 읽은 후에("LDA X") 데이터를 출력시키는 서브루틴을 호출하면서("BSA SUB") 리턴 주소 0x104 를 SUB 번지에 저장한다. 서브루틴에서는 출력 플래그를 검사하고 출력 장치에 준비가 되면 ("BUN LOA") 명령을 생략하고 다음 명령인 "OUT"에 의해 AC 에 저장된 0x10 를 출력하고 "BUN SUB I" 명령으로 리턴한다. 서브루틴에서 리턴되면 0x104 번지 "INC" 명령으로 AC 값은 하나 증가해서 0x11 가 되고 "ISZ CTR" 명령으로 CTR 은 -1 로 증가한다. 그 다음 "BUN LOP"에 의해 LOP 번지부터 실행을 하여 다시 서브루틴을 호출한다. 이와 같이 두 번째 데이터 0x11 를 출력한 후에 CTR 을 증가시켜서 0 이 되면("ISZ CTR") 다음 명령 "BUN LOP"을 생략하고 프로그램을 종료한다. 실행결과는 출력 7-세그먼트에 출력된다.

1.3.10. 간단한 인터럽트 프로그램(ex10.s)

■ 프로그램 소스 코드

```
        ORG 0
ZRO,    HEX 0       : 인터럽트로부터의 리턴 주소를 저장
        BUN ISV     : 인터럽트 처리 루틴을 실행

        ORG 100
        CLA         : AC를 클리어
        OUT         : 출력 플래그를 클리어 시킴
        ION         : 인터럽트를 인에이블 시킴
LOP,    LDA X       : X 번지 데이터(0x10)를 AC에 로드
        ADD Y       : Y 번지 데이터(0x20)를 더함.(AC ← 0x30)
        STA Z       : AC 값(0x30)를 Z번지에 저장.
        BUN LOP     : LOP 번지부터 반복 실행.

ISV, STA SAC        : AC 값을 SAC 번지에 저장
        CIR         : AC를 한 비트 씩 오른쪽으로 이동, E 플립플롭은 AC(15)으로 이동
        STA SE      : AC를 SE 번지에 저장
        SKI         : 입력 플래그를 검사하여 '1'이면 다음 명령을 생략.
        BUN NXT     : NXT 번지부터 실행
        INP         : INPR에서 AC로 데이터를 읽음.
        STA X       : X 번지에 AC 값을 저장
NXT,    SKO         : 출력 플래그를 검사해서 '1'이면 다음 명령을 건너 뜀
        BUN EXT     : EXT 번지 명령부터 실행
        LDA Z       : Z 번지 데이터를 AC로 읽음.
        OUT         : OUTR 데이터를 출력함.
EXT,    LDA SE      : SE 번지 데이터를 AC로 읽음.
        CIL         : AC를 한 비트씩 왼쪽으로 이동, AC(15)는 E 플립플롭으로 이동
```

```
        LDA SAC     : SAC 번지 데이터를 AC로 읽음.
        ION         : 인터럽트를 인에이블 시킴.
        BUN ZRO I   : 서브루틴에서 리턴.

X,      HEX 10
Y,      HEX 20
Z,      HEX 0
SAC,    HEX 0
SE,     HEX 0
        END
```

■ 프로그램 설명

프로그램에서 입출력장치를 구동하는 방법은 인터럽트 방식과 폴링(polling) 방식이 있다. 앞의 예제 ex8 과 ex9 에서는 입출력 장치에서 데이터를 입력하거나 출력할 때는 계속해서 입력 플래그 또는 출력 플래그를 검사해서 입출력 장치가 준비되었을 때 입력 또는 출력을 실행한다. 그러나 이 방식의 단점은 입출력 플래그를 검사하는 동안에 프로그램에서 다른 동작 할 수 없다. 따라서 이런 단점을 보완하기 위해서 인터럽트 방식을 사용하는 것이 효율적이며, 이 방식에서는 입출력 장치에서 입력이 되었거나 또는 출력장치가 준비가 되었을 때만 입출력 동작을 하고 그 외에는 다른 동작을 할 수 있다.

인터럽트를 실행하는 기본 동작은 인터럽트를 인에이블 시킨 후에 프로그램을 실행하다가 인터럽트가 발생하면 인터럽트 리턴 주소를 저장하고 인터럽트 처리루틴으로 점프를 한다. 인터럽트 처리루틴에서는 자동으로 인터럽트가 디스에이블 되며, 현재 레지스터 값들을 대피시키고 입력 또는 출력 장치중 어느 장치에서 인터럽트가 발생했는지를 검사하고 해당 하는 인터럽트를 처리한 후 대피 시켰던 레지스터 값들을 복구 한 후 다시 인터럽트를 인에이블 시키고, 메인 프로그램으로 리턴한다.

메인 프로그램은 0x100 번지부터 실행이 되며 "ION" 명령이 실행된 후에 "LOP, LDA X" 부터 "BUN LOP" 명령어까지 계속 반복 실행된다. 이 부분을 반복 실행하면서 X 데이터를 읽어서 Y 데이터를 더한 후에 Z 에 저장한다. 인터럽트가 발생하기 전까지 이 동작을 반복하며, 인터럽트가 발생하면 현재의 PC 값을 0 번지에 저장하고 1 번지부터 실행을 한다. 인터럽트는 입력이 발생하거나 또는 출력장치가 준비가 되었을 때 입출력 플래그가 세트되고, 이 플래그에 의해 인터럽트 플립플롭이 '1'로 세트되면서 발생한다. 0 번지에 저장된 PC 값은 인터럽트 처리를 마친 후에 리턴 주소로 사용된다. 1 번지에는 "BUN ISV" 명령이 있으며, 인터럽트 처리 루틴에 있는 프로그램을 실행한다. ISV 번지에서는 가장 먼저 현재 사용중인 레지스터 값을 대피시키며, 위의 프로그램에서는 AC 값과 E 플립플롭을 메모리에 저장한다. "STA SAC"는 메인 프로그램에서 저장되었던 AC 값을 SAC 에 저장하며, "CIR"에 의해 AC 를 한 비트씩 오른 쪽으로 이동시킨다. 이 때 E 플립플롭은 AC 의 15 번 비트로 이동하며 "STA SE"에 의해 다시 AC 값이 저장되면서 AC 의 15 번 비트에 있는 E 플립플롭

값이 저장된다. 다음 인터럽트를 발생한 장치를 찾기 위해서 입력 플래그와 출력 플래그를 검사한다. 먼저 "SKI" 명령으로 입력 플래그를 검사하고 만일 이 플래그 값이 '0'이면 입력이 발생하지 않은 것이므로 다음 명령 "BUN NXT"를 실행한다. 그러나 입력 플래그가 '1'이면 "BUN NXT"를 생략하고 "INP" 명령과 "STA X" 명령으로 입력된 데이터를 X 번지에 저장한다. NXT 번지에서는 출력 플래그를 검사하며("SKO") 플래그가 '0'이면 EXT 번지로 점프하며("BUN EXT"), 플래그가 '1'이면 Z 번지에서 AC 에 읽은 후에 출력장치로 출력을 한다. 결과적으로 입력장치에서 읽은 값이 X 번지에 저장되며, 메인 프로그램에서 Y 번지의 0x20 와 더해져서 Z 번지에 저장되면 출력 인터럽트가 발생해서 Z 번지에 저장된 값이 출력된다. EXT 번지에서는 메인 프로그램으로 리턴하기 위해 메모리에 저장했던 AC 와 E 플립플롭 값의 복구 시킨다. 레지스터를 복구하는 순서는 메모리에 대피하는 순서와 반대이며 먼저 E 플립플롭을 복구하기 위해 SE 번지의 데이터를 AC 로 읽는다. AC 의 15 번에는 E 플립플롭 값이 저장되어 있으므로 이 값을 E 플립플롭으로 이동하기 위해 AC 를 한 비트씩 왼쪽으로 이동한다. 이 때 AC 의 15 번 비트는 E 플립플롭으로 이동하므로 E 플립플롭이 복구된다. 다음 SAC 번지에 저장된 값을 AC 로 읽으면 AC 가 복구가 되고, 인터럽트를 인에이블 시킨 후에("ION") "BUN ZRO I"에 의해 메인 프로그램으로 리턴한다. ZRO 번지에는 인터럽트가 발생했을 때 PC 값이 저장되어 있으므로 인터럽트가 발생했을 때 실행중이던 명령어의 다음 명령어부터 실행한다. 결과적으로 이 프로그램을 실행시키고 FGO 스위치를 누르면 인터럽트에 의해 X 번지와 Y 번지에 각각 저장된 0x10 과 0x20 를 더한 결과가 출력되고 다시 입력 키 스위치를 누르면(0xA)를 누르고 FGO 스위치를 누르면 입력된 키값(0xA)과 0x20 이 더해진 결과 0x2A 가 출력된다. 실행결과는 출력 7-세그먼트에 출력된다.

2. 컴퓨터 구조 실습

기본 컴퓨터를 VHDL 로 설계하기 위해서는 VHDL 을 잘 사용해야 할 뿐만 아니라 컴퓨터의 동작도 완벽하게 이해를 해야 한다. 그러나 동시에 두 가지를 안다는 것은 쉽지가 않다. 물론 본 교재 3 장에서는 기본 컴퓨터를 설계하는 과정을 가능한 상세하게 설명을 하지만 그 전에 기본 컴퓨터의 동작 원리를 충분히 이해 하고 있어야 한다. 그러나 동작원리를 이론적인 내용만으로 이해를 하는 것도 쉬운 것은 아닌 것으로 보인다. 따라서 2 장에서는 몇가지 실습을 통해서 컴퓨터 동작을 이해하는데 도움을 주고자 한다.

2.1. 레지스터 전달 표현(Register Transfer Statement) 설계

레지스터 전달 표현은 한 레지스터에서 다른 레지스터로 데이터를 전달하는 동작을 심볼로 표현한 것이며 일정한 조건을 만족할 때 동작이 발생한다. 컴퓨터의 실행은 많은 부분이 일정한 조건에서 한 레지스터에서 다른 레지스터로 데이터를 전달하거나 논리산술연산을 실행하면서 결과를 저장하는 동작으로 이루어 졌다. 본 실습에서는 심볼로 표현된 레지스터 전달 표현을 VHDL 로 구현하는 실습을 한다.

레지스터 전송은 아래와 같이 나타낼 수 있다.

if(P=1) then (R2 ← R1) 또는 P : R2 ← R1

위의 표현은 P 가 '1'이면 레지스터 R1 에 저장된 데이터를 레지스터 R2 로 전달하는 것을 나타낸 것이다. 위와 같은 레지스터 전송을 VHDL 로 DIGCOM-A1.2 에 구현하기 위해 R1 과 R2 를 8 비트의 크기를 갖는 레지스터로 선언하고, R1 레지스터에 저장될 데이터는 슬라이드 스위치로 입력할 수 있도록 하고 데이터가 전송될 R2 레지스터는 7-세그먼트로 출력하여 전송이 되는 것을 확인한다.

- 새로운 프로젝트를 생성
  ① Quartus Prime을 시작하고 **File ➔ New Project Wizard**를 선택한다. 새로운 윈도우에서 **Next**를 선택하고, 다음 윈도우에서 프로젝트 폴더(regTransfer)를 정하고 파일이름과 top-level design entity를 regTransfer으로 정하고 **Next**를 클릭한다.
  ② 기존의 파일이 없으므로 새로운 윈도우에서 **Next**를 클릭한다.
  ③ 다른 종류의 설계도구를 사용하지 않으므로 새로운 윈도우에서 **Next**를 클릭한다.
  ④ 다음 윈도우에서 **device famly**를 **Cyclone IV E**로 선택하고, **available device**에서 **EP4CE6F17C8**를 선택한 후 **Next**를 클릭하고 **Finish**를 누르면 새로운 프로젝트가 생성된다.

- [그림 2-1] 블록도는 Reg2 의 제어신호 load 가 '1'이면 Reg1 에 저장된 데이터가 Reg2 로 전송되는 회로의 블록도이다. 이 블록도를 VHDL 로 구현하라.

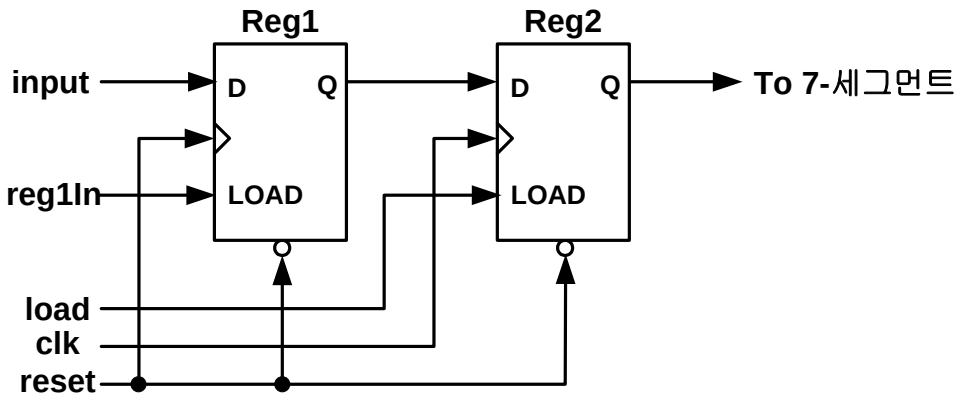


그림 2-1. 레지스터의 구현과 전송

① signal로 reg1과 reg2 두개의 레지스터를 선언하고 초기화 한다.

② reg1의 reg1In 신호가 인에이블이면 클럭의 상승에지에서 슬라이드 스위치에서 입력되는 데이터가 reg1 레지스터에 저장하고, load 신호가 인에이블되면 클럭의 상승에지에서 reg1의 데이터를 reg2에 전송한다.

```
if reset = '0' then
        reg1 <= "00000000";
        reg2 <= "00000000";
elsif falling_edge(clk) then
        if reg1In = '1' then
                reg1 <= input;
        elsif load = '1' then
                reg2 <= reg1;
        end if;
end if;
```

그림 2-2. 레지스터 전송 부분

③ reg2에 저장된 데이터를 7-세그먼트에 출력한다. 레지스터가 8비트 이므로 2개의 7-세그먼트에 16진수로 출력을 한다.

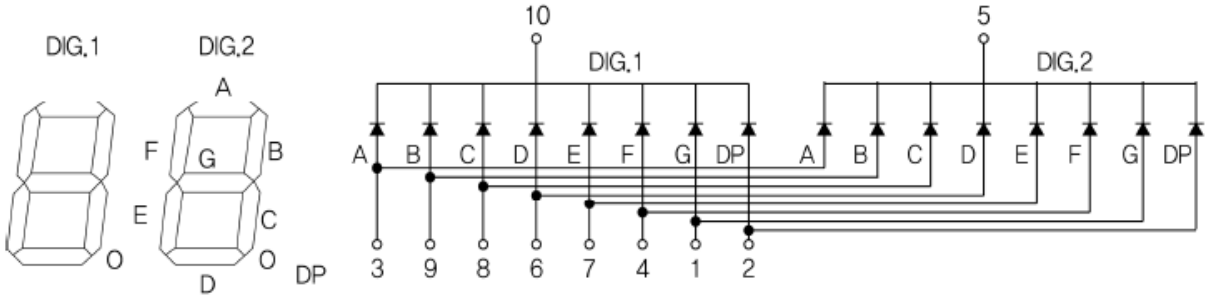


그림 2-3. 7-세그먼트 FND 의 LED 배치

7-세그먼트의 각 LED 는 위의 [그림 2-3]과 같으며, A~G 까지 출력단자에 차례로 연결되어 있다. Reg2 의 7 ~ 4 번 비트의 값을 7-세그먼트에 출력하는 VHDL 코드는 [그림 2-4]와 같다. (7-세그먼트 FND 디코딩 방법은 교재 "디지털 시스템 설계 및 실습"의 Lab. 4 "7-세그먼트 FND 디코더 설계"를 참고할 것.)

```
elsif rising_edge(clk) then
        case reg2(7 downto 4) is
--                                                  "abcdefg-"
                when "0000"    => seg1 <=    "1111110";
                when "0001"    => seg1 <=    "0110000";
                when "0010"    => seg1 <=    "1101101";
```

그림 2-4. 7-세그먼트 FND 출력

레지스터의 나머지 값에 대해서도 7-세그먼트 디코딩을 하고, 3~0 번 비트에 해당하는 값은 다른 7-세그먼트에 출력한다.

■ 컴파일과 시뮬레이션

① **Processing ➔ Start Compilation**을 클릭한다.
② 컴파일을 성공적으로 마치면 **Full Compilation was successful** 메시지를 볼 수 있다.
③ **File ➔ New ➔ Other Files ➔ Vector Waveform File**을 선택한 후 **OK**를 클릭한다.
④ 메인 메뉴에서 **Edit ➔ Insert ➔ Insert Node of Bus**를 선택한다.
⑤ **Node Finder**를 클릭.
⑥ **Filters**에서 **Pins:all**을 선택하고 **List**를 클릭한다.
⑦ **Nodes Found**에서 입출력 신호를 선택하고 화살표를 클릭하여 **Selected Nodes**로 이동시키고 **OK ➔ OK**를 클릭한다.
⑧ 신호를 입력하고 시뮬레이션을 실행하여 결과를 확인한다.

■ 핀 번호를 할당

① Quartus Prime 메인메뉴에서 **Assignments ➔ Pins**를 선택한다.

② **Top View – Wire Bond**에서 핀 번호를 더블클릭하고 할당할 입력 노드 이름을 입력한다.

③ 출력에 대해서도 같은 단계를 반복하여 출력에 핀번호를 할당한다.

④ 핀 번호를 할당하면 다시 컴파일 한다.

표 2-1. 디바이스 핀 할당과 이름

핀 이름	입출력 장치
reset	리셋 스위치
Clk	클럭
Input	슬라이드 스위치
reg1In	푸시버튼 스위치
Load	푸시버튼 스위치
FND	7-세그먼트

■ 컴파일된 pof 파일을 DIGCOM-A1.2 에 다운로드

① **USB Blaster** 케이블이 연결된 것을 확인하고 **Tools ➔ Programmer**를 실행한다.

② **Hardware Setup**이 'USB-Blaster[USB-0]'로 설정되었는지 확인하고, **Mode**를 '**Active Serial Programming**'으로 선택한다. 프로젝트 파일을 추가하기 위해 [**Add File...**] 버튼을 클릭하여 **'프로젝트 폴더₩output_files₩AlteraProject.pof'** 파일을 선택한 후 [**Open**] 버튼을 누른다. 파일이 추가되면 'Program/Configure'를 체크한다. 설정이 완료되면 [**Start**] 버튼을 클릭해서 FPGA에 다운로드한다.

■ DIGCOM-A1.2 에서 실행

① 첫번째 레지스터 Reg1에 저장할 데이터를 슬라이드 스위치(SW0~SW7)에 설정하고 푸시버튼 스위치 Key0(reg1In)를 눌러서 Reg1에 저장한다.

② 푸시버튼 스위치 Key1(load)을 누르면 Reg1에 있는 데이터가 Reg2로 전송되고, 값이 7-세그먼트에 출력된다.

③ 새로운 값을 입력해서 전송하기 위해서 리셋을 한 후 처음부터 반복한다.

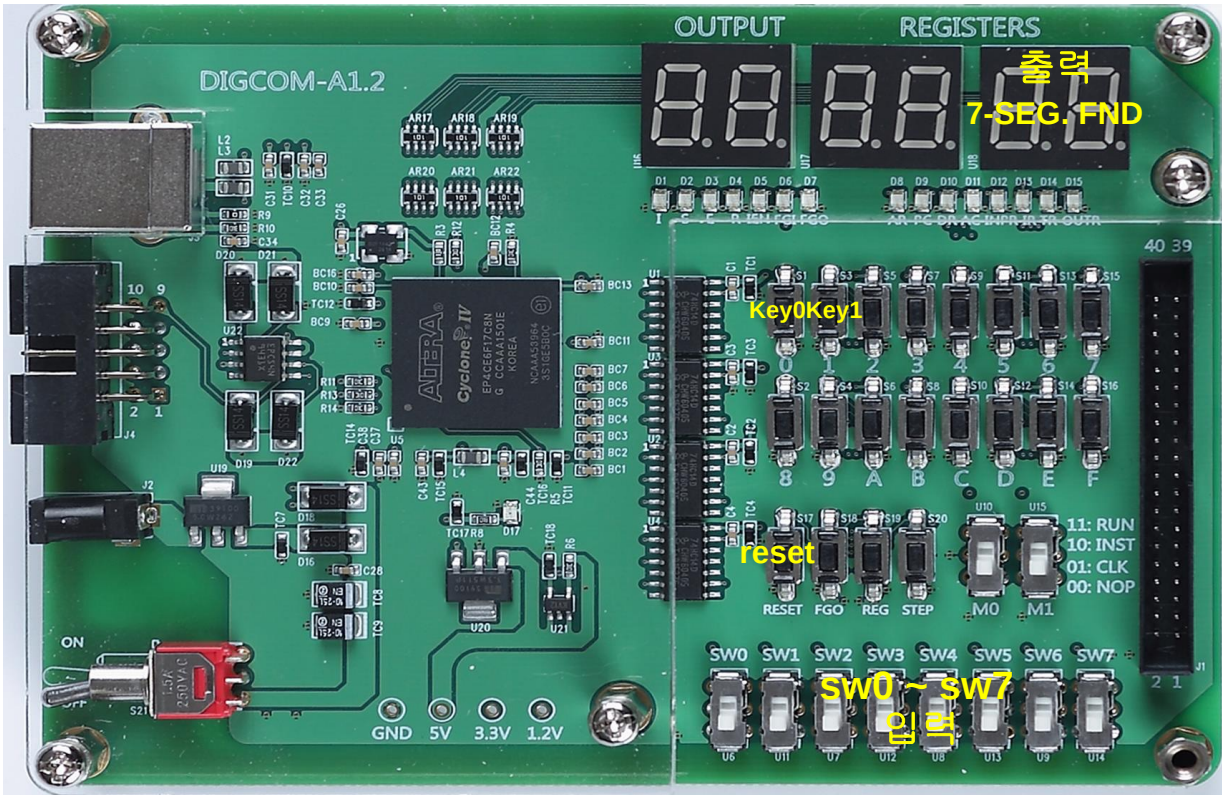
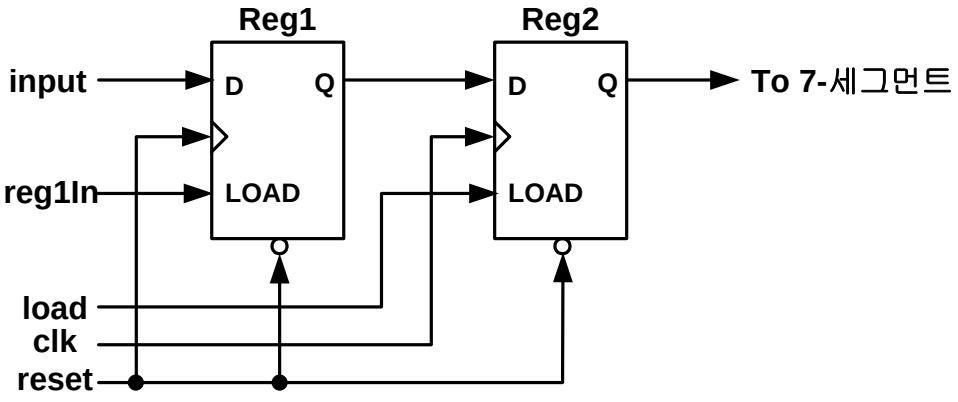


그림 2-5. 레지스터 전송을 위한 DIGCOM-A1.2**의 입출력 할당**

컴퓨터 구조 실습 결과보고서[ComLab-1]

일 시		**전 공**		**실험시간**	
학 번			**이 름**		
제 목	레지스터 전달 표현(Register Transfer Statement)의 VHDL 설계				
실습목적	레지스터 전달 표현은 한 레지스터에서 다른 레지스터로 데이터를 전달하는 동작을 심볼로 표현한 것이며 일정한 조건을 만족할 때 동작이 발생한다. 컴퓨터의 실행은 많은 부분이 일정한 조건에서 한 레지스터에서 다른 레지스터로 데이터를 전달하거나 논리산술연산을 실행하면서 결과를 저장하는 과정으로 이루어 졌으며, 본 실습을 통해 심볼로 표현된 레지스터 전달을 VHDL 로 구현함으로써 컴퓨터를 설계하는 방법을 습득한다.				
실습 내용	**실습결과**				
레지스터 전송블록의 VHDL 설계	아래 블록도는 제어신호에 의해 한 레지스터에서 입력을 받아 다른 레지스터로 데이터를 전송하는 회로이다. 이 블록도를 VHDL로 설계하라.  				

	VHDL 코드
실습키트 실행결과	컴파일된 pof 파일을 DIGCOM-A1.2에 다운로드하여 위 시뮬레이션과 같은 입력일 때 실행한 결과의 이미지를 아래 표에 넣어라. DIGCOM-A1.2 **결과 이미지**
실습결과 및 토의	

2.2. Arithmetic Logic Shift Unit(ALSU)의 VHDL 설계

ALU 는 컴퓨터에 있어서 산술논리 연산을 하는 가장 핵심적인 부분이다. 메모리에서 명령어를 패치한 후 명령어의 opcode 를 디코딩하고, 실행할 산술논리 연산의 종류를 결정한다. 본 실습에서는 디코딩된 opcode 를 대신하여 연산의 종류를 선택하는 select 신호를 입력하고, select 신호에 의해 실행할 연산동작을 VHDL 로 설계한다. 피 연산자는 2 개이므로 2 개의 8 비트 레지스터를 선언하고 레지스터 값은 DIGCOM-A1.2 의 슬라이드 스위치로 입력한다. 첫번째 레지스터의 load 신호에 할당된 푸시버튼 스위치를 누름으로써 슬라이드 스위치의 값이 레지스터에 입력되도록 하고, 또 다른 푸시버튼 스위치는 슬라이드 스위치의 값이 두번째 레지스터에 입력되도록 한다. 두 레지스터에 피 연산자가 입력이 되면 연산의 종류를 결정하는 select 를 슬라이드 스위치 설정하고 연산동작이 발생하도록 신호를 준다. 연산결과는 다른 레지스터에 저장이 되고 7-세그먼트에 출력함으로써 결과를 확인한다.

본 실습에서 설계할 연산의 종류는 다음과 같다.

표 2-2. Select 신호와 연산의 종류

Operation select					Operation	Function
S3	S2	S1	S0	Cin		
0	0	0	0	0	F = A	Transfer A
0	0	0	0	1	F = A + 1	Increment A
0	0	0	1	0	F = A + B	Addition
0	0	0	1	1	F = A + B + 1	Add with carry
0	0	1	0	0	F = A + B′	Subtract with borrow
0	0	1	0	1	F = A + B′ +1	Subtraction
0	0	1	1	0	F = A – 1	Decrement A
0	0	1	1	1	F = A	Transfer A
0	1	0	0	x	F = A ^ B	AND
0	1	0	1	x	F = A V B	OR
0	1	1	0	x	F = A xor B	XOR
0	1	1	1	x	F = A′	Complement A
1	0	x	x	x	F = shr A	Shift right A into F
1	1	x	x	x	F = shl A	Shift left A into F

- 새로운 프로젝트를 생성 : "2.1. 레지스터 전달 표현의 구현" 실습에서와 같은 방법으로 새로운 프로젝트를 생성한다. 이 프로젝트의 폴더와 프로젝트명을 alsUnit으로 정한다.

- [그림 2-6]은 [표 2-2]의 연산을 하는 회로의 블록도 이다. "2.1. 레지스터 전달 표현의 구현" 실습에서와 같은 절차로 VHDL을 설계한다.

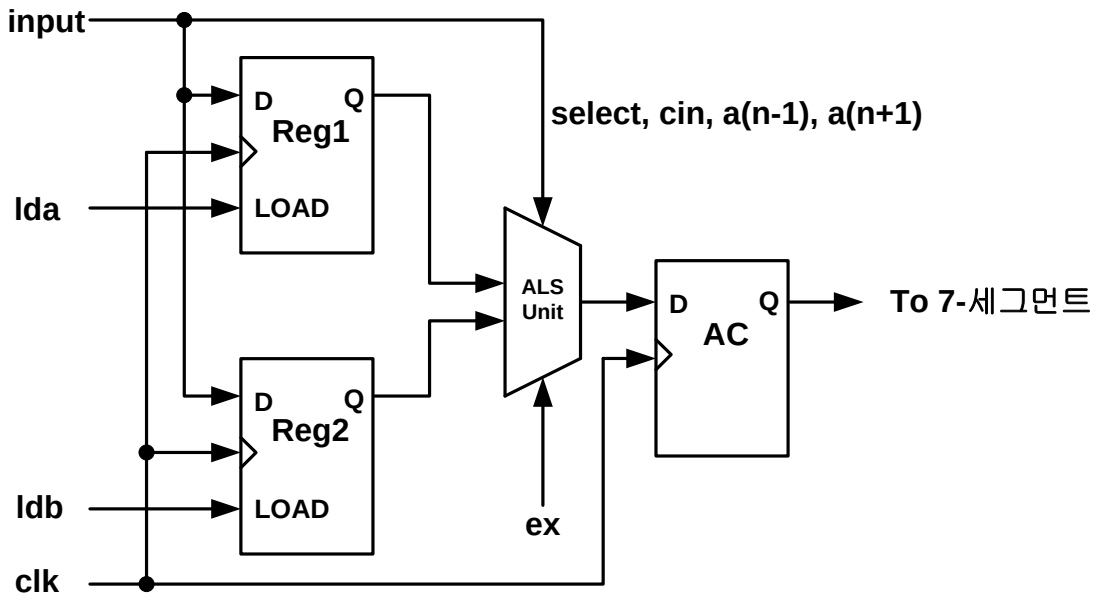


그림 2-6. Arithmetic Logic Shift Unit 블록도

① 입력 input은 슬라이드 스위치이며, lda는 input 입력을 reg1에, ldb는 입력을 reg2에 각각 저장하는 푸시버튼 스위치이다.

② 두 레지스터에 데이터를 입력하면 input은 연산의 종류를 결정하는 select 선택 신호와 시프트 동작에 사용될 a(n-1)과 a(n+1) 입력으로 사용된다.

③ ex 입력은 [그림 2-6]과 같이 두 레지스터에 저장된 데이터를 select 선택 신호에서 정한 연산을 하게 한다.

④ 연산 결과는 AC 레지스터에 저장되고, AC를 디코딩하여 두 개의 7-세그먼트에 출력한다. 디코딩 과정은 실습 2.1과 같다.

⑤ [그림 2-7]은 두 레지스터에 피 연산자를 입력하고 select 신호에 의해 정해지는 연산을 실행하는 VHDL 코드이다.

```
        if reset = '0' then
                reg1 <= "00000000";
                reg2 <= "00000000";
                ac <= "00000000";
        elsif rising_edge(clk) then
                if lda = '1' then
                        reg1 <= input;
                elsif ldb = '1' then
                        reg2 <= input;
                elsif ex = '1' then
```

```
case input(7 downto 4) is
        when "0000" =>
                if input(2) = '0' then ac <= reg1;
                else ac <= reg1 + 1;
                end if;
        when "0001" =>
                if input(2) = '0' then ac <= reg1 + reg2;
                else ac <= reg1 + reg2 + 1;
```

그림 2-7. 레지스터 데이터 입력과 연산동작

- 설계된 VHDL 코드를 컴파일 하고 시뮬레이션을 하여 결과를 확인한다. 과정은 실습 2.1의 과정을 참고하라.
- [표 2-3]과 같이 디바이스 핀 번호를 할당한다.

표 2-3. 디바이스 핀 할당과 이름

핀 이름	입출력 장치
Reset	리셋 스위치
Clk	클럭
Inp	슬라이드 스위치
lda, ldb, cin	푸시버튼 스위치
Ex	푸시버튼 스위치
FND	7-세그먼트

- 실습 2.1 과 같이 pof 파일을 DIGCOM-A1.2 에 다운로드하여 실행한다.

- DIGCOM-A1.2 에서 실행
  ① 첫번째 레지스터 reg1에 저장할 피연산자 값을 슬라이드 스위치(SW0 ~ SW7)에 설정하고 푸시버튼 스위치 Key0(lda)를 눌러서 Reg1에 저장한다.
  ② 같은 방법으로 두번째 레지스터 Reg2에 저장할 피연산자 값을 슬라이드 스위치(SW0~SW7)에 설정하고 푸시버튼 스위치 Key1(ldb)를 눌러서 Reg2에 저장한다.
  ③ 연산의 종류를 결정하는 select 신호를 input(7)~input(4)에 설정하고, cin(input(2))과 shift right in(input(1)) 및 shift left in(input(0))을 설정한 후에 푸시버튼 스위치 Key2(ex)를 누르면 정해진 연산이 실행되어 결과가 AC에 저장되며 그 값이 디코드 되어 7-세그먼트에 출력된다.

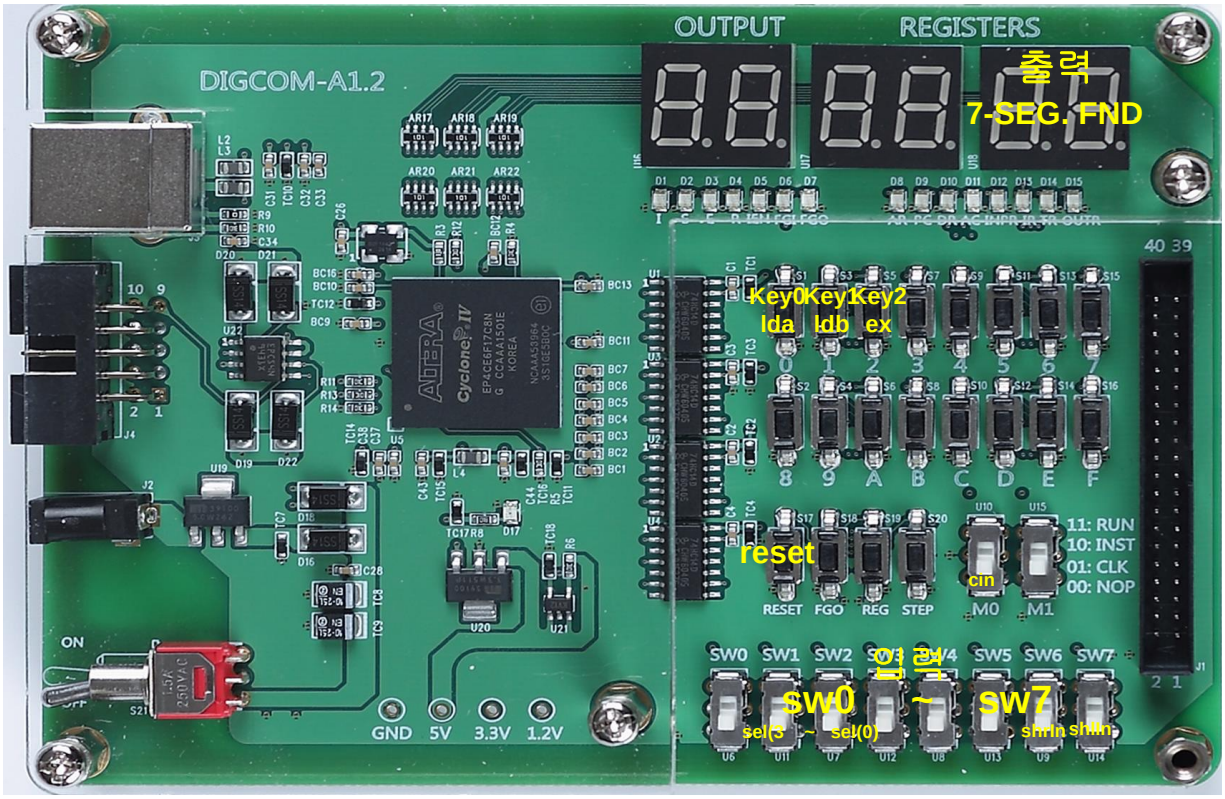
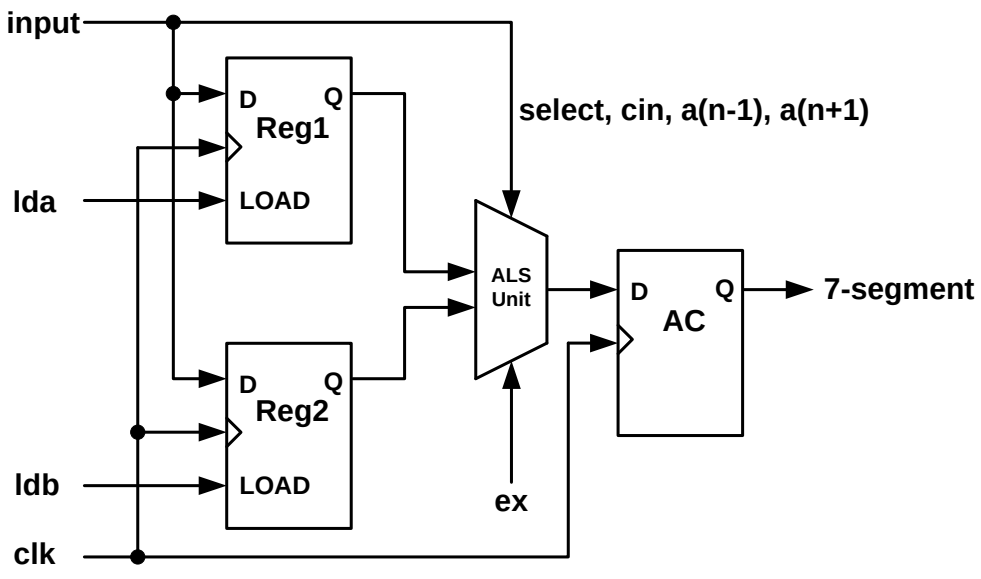


그림 2-8. ALSU를 위한 DIGCOM-A1.2의 입출력 할당

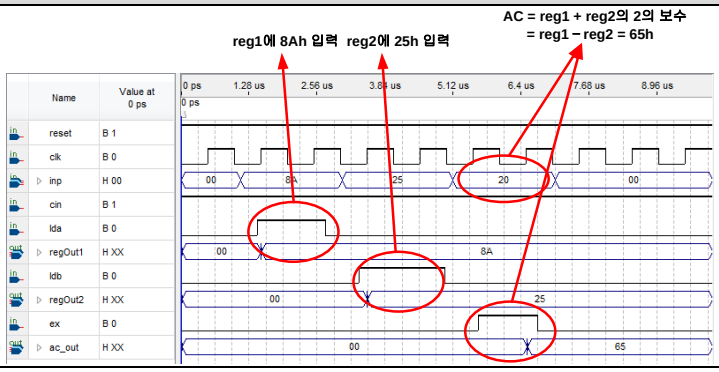
컴퓨터 구조 실습 결과보고서[ComLab-2]

일 시		**전 공**		**실험시간**	
학 번			**이 름**		
제 목	Arithmetic Logic and Shift(ALSU)의 VHDL 설계				
실습목적	ALSU 는 컴퓨터에 있어서 산술논리 연산을 하는 가장 핵심적인 부분이다. 메모리에서 명령어를 패치한 후 명령어의 opcode 를 디코딩하고, 실행할 산술논리 연산의 종류를 결정한 후, ALSU 에서 선택된 연산을 실행하고 결과를 AC 에 저장한다. 본 실습에서는 연산될 데이터를 입력하고 select 신호로 연산의 종류를 결정한 후 연산동작을 하는 ALSU 의 동작을 VHDL 로 설계한다.				
실습 내용	**실습결과**				
레지스터 전송블록의 VHDL 설계	아래 블록도는 제어신호에 의해 한 레지스터에서 입력을 받아 다른 레지스터로 데이터를 전송하는 회로이다. 이 블록도를 VHDL로 설계하라.  				

VHDL 코드

시뮬레이션 결과

설계한 ALSU를 시뮬레이션하여 두 입력에 대해서 select신호가 "0100"일 때 아래에서 보인 것과 같이 시뮬레이션 결과를 캡쳐하고 동작을 설명하라.

입력		select	c_{in}	시뮬레이션 결과
R1	R2			
0x8A	0x25	0010	1	
		0100	x	

<table>
<tr><td>실험키트
실행결과</td><td>pof 파일을 DigComV32에 다운로드하여 위 시뮬레이션과 같은 입력일 때 실행한 결과의 이미지를 아래 표에 넣어라.
<table><tr><th>DigComV32 실행 결과 이미지</th></tr><tr><td></td></tr></table></td></tr>
<tr><td>실습결과
및 토의</td><td></td></tr>
</table>

2.3. 마이크로 연산과 제어 함수

명령어는 몇 개의 마이크로 연산으로 구성되어 있으며, 이 마이크로 연산이 실행되기 위해서는 해당하는 연산이 실행될 수 있도록 제어신호가 발생해야 한다. 이 제어신호를 발생하는 논리함수를 제어함수라 한다. 예로 모든 명령에 해당하는 패치 디코드의 마이크로 연산은 아래와 같다.

Fetch $R'T_0$: AR←PC

$R'T_1$: IR←M[AR], PC←PC +1

Decode $R'T_2$: D_0, ...,D_7←Decode IR (12-14), AR←IR(0-11), I←IR(15)

즉 위의 패치 단계에서 PC 의 값이 AR 로 전달되기 위해 필요한 제어신호는 PC 의 값을 공통버스로 보내는 X(2)과 공통버스에 실린 데이터를 AR 에 저장하게 하는 load 신호(ar_ld)이다. 따라서 첫번 째 클럭에서 $R'T_0$ 에서 AR ← PC 마이크로 연산이 실행되기 위한 제어 신호와 함수는 다음과 같다.

$ar\_ld = R' \text{ AND } T_0$

$X(2) = R' \text{ AND } T_0$

이와 같이 각 T 클럭마다 제어신호를 발생시키고 이 제어신호에 의해 마이크로 연산이 실행된다.

- LDA 명령을 위한 제어 함수표 작성

① 기본 컴퓨터에서 사용되는 레지스터 또는 메모리와 이를 제어하는 신호는 [표 2-4]와 같다. 공통버스 제어는 각 레지스터에 저장된 값이 다른 레지스터로 전송되기 위해 공통버스에 실리게 하기 위한 제어신호이며, 레지스터 제어신호는 해당 레지스터에 값을 저장하거나 변경하기 위해 필요한 신호들이다.

표 2-4. 기본 컴퓨터에서 사용되는 레지스터와 제어신호

레지스터(메모리)	공통버스 제어	제어신호	레지스터 동작
AR(Address Register)	X(1)	ar_ld	공통버스 값을 AR 에 저장
		ar_inr	AR 을 하나 증가시킴
		ar_clr	AR 을 클리어 시킴
PC(Program Counter)	X(2)	pc_ld	공통버스 값을 PC 에 저장
		pc_inr	PC 을 하나 증가시킴
		pc_clr	PC 을 클리어 시킴
DR(Data Register)	X(3)	dr_ld	공통버스 값을 DR 에 저장
		dr_inr	DR 을 하나 증가시킴
		dr_clr	DR 을 클리어 시킴

AC(Accumulator)	X(4)	ac_ld	ALU 출력값을 AC 에 저장
		ac_inr	AC 을 하나 증가시킴
		ac_clr	AC 을 클리어 시킴
IR(Instruction Register)	X(5)	ir_ld	공통버스 값을 IR 에 저장
TR(Temporary Register)	X(6)	tr_ld	공통버스 값을 TR 에 저장
		tr_inr	TR 을 하나 증가시킴
		tr_clr	TR 을 클리어 시킴
OUTR(Output Register)		outr_ld	공통버스 값을 OUTR 에 저장
Memory Unit	X(7)	M_en	메모리에 읽기/쓰기를 할 때 발생
		M_write	메모리 쓰기를 제어
		M_read	메모리 읽기를 제어

[표 2-5]는 LDA가 실행되기 위한 마이크로 연산이다. T_1 클럭부터 각 마이크로 연산 단계에서 필요한 제어신호를 발생시키는 제어함수와 동작을 설명하라.

표 2-5. LDA를 위한 마이크로 연산, 제어 신호와 제어 함수

마이크로 연산	제어신호와 제어함수	동작 설명
$R'T_0$: AR←PC	X(2) = R′ AND T_0 ar_ld = R′ AND T_0	PC 값을 공통버스에 보냄 공통버스 데이터를 AR 에 저장
$R'T_1$: IR←M[AR], PC←PC +1		
$R'T_2$: D_0, …,D_7←Decode IR (12-14) AR←IR(0-11), I←IR(15)		
D'_7IT_3 : AR←M[AR]		
D_2T_4 : DR←M[AR]		
D_2T_5 : AC←DR, SC←0		

■ 하나의 명령어(LDA)를 갖는 어셈블리 언어 프로그램 작성

① Assembler 폴더의 AsmCompiler를 더블 클릭하여 실행하면 편집 화면이 나온다. 이 편집창에 [그림 2-9]와 같은 프로그램을 작성한다. 아래 프로그램은 하나의 명령어 LDA로 구성된 프로그램이다. 마지막에 HLT 명령어도 있지만 이 명령어는 프로그램 동작을 멈추기 위해 사용되었다.

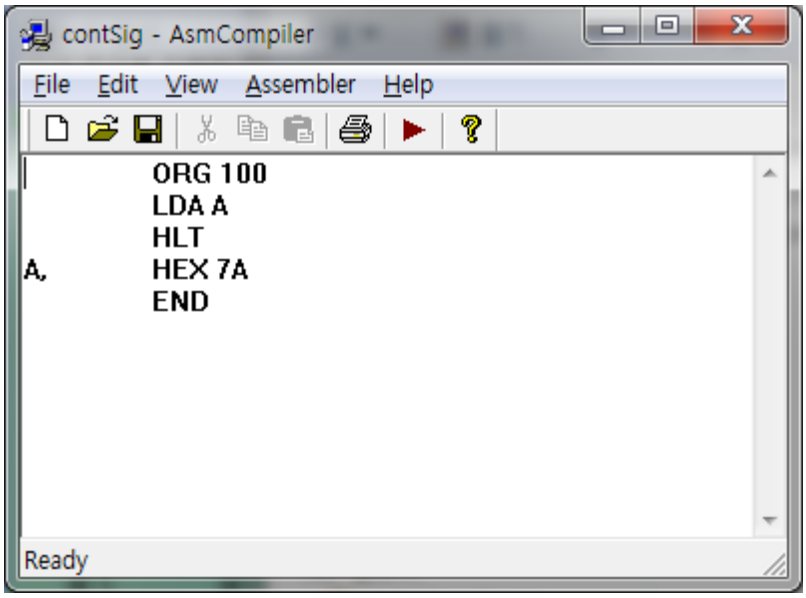


그림 2-9. 어셈블러 편집 창과 LDA 명령어

② 프로그램이 작성되면 메뉴에서 **File ➔ Save**를 선택하고 확장자 s(contSig.s)로 파일 이름을 저장한다.

③ 작성된 프로그램을 어셈블하기 위해 메뉴에서 **Assembler ➔ Assemble**를 선택하거나 **Assembler** 단축하이콘을 클릭하여 어셈블을 한 후 확장자 mif(contSig.mif)로 저장한다.

■ VHDL 코드를 수정하고 다시 컴파일 한다.

① 어셈블된 mif 파일을 DIGCOM-A1.2₩VHDL₩SimpleComputer₩DigCom-A1.2-NoStep 폴더로 복사한 후, simplecomA12 프로젝트를 오픈하고 메모리의 LPM_RAM_IO에서 LPM_FILE을 어셈블된 mif 파일명(contSig.mif)으로 수정한다.

② 시뮬레이션을 위해서 VHDL 소스코드에서 reset 블록과 scounter 블록을 수정한다. 프로젝트에서 digcom.bdf를 오픈하고 리셋 블록을 클릭한 후 [그림 2-10]과 같이 수정한다.

```
                    if rising_edge(clk) then
--                          if cnt < 1000000 then
                            if cnt < 100 then
                                    cnt := cnt + 1;
                                    reset_node <= '0';
--                          elsif cnt >= 1000000 then
                            elsif cnt >= 100 then
                                    cnt := cnt + 0;
```

```
                    reset_node <= '1';
                end if;
            end if;
```

그림 2-10. 리셋 블록의 수정

위의 수정내용은 전원이 공급이 되면 약 1초후에 리셋신호를 해제하는 부분으로 외부에서 공급되는 클럭이 1MHz이므로 카운터가 백만번을 카운트해서 1초가 경과한 후에 리셋 신호를 해제하는 것이다. 그러나 시뮬레이션에서는 백만번까지 카운트할 필요가 없기 때문에 위와 같이 수정해서 100번만 카운트 하도록 한다. 또한 scounter 블록에서 [그림 2-11]과 같이 수정한다.

```
            elsif rising_edge(inClk) then
--              if cnt20000 >= 20000 then
                if cnt20000 >= 10 then
                    cnt20000 := 0;
                    clk20000 <= not clk20000;
                else
                    cnt20000 := cnt20000 + 1;
                end if;
            end if;
```

그림 2-11. scounter 블록 수정

[그림 2-11]의 수정 내용은 푸시 버튼 스위치를 누를 때 채터링(chattering)[4]을 제거하기 위해서 입력 클럭을 20,000 분주한 후 분주된 클럭에 의해 스위치 값을 입력받는 부분으로 이 부분도 20,000번까지 카운트 할 필요가 없기 때문에 10번으로 줄인다. 이와 같이 VHDL 코드를 수정하고 다시 컴파일 한다.

■ 기본컴퓨터를 시뮬레이션을 한다.

① 메인 메뉴에서 **Processing ➔ Simulation Tool**을 선택한 후 **Open** 버튼을 클릭하면 시뮬레이션 화면이 오픈된다.

② 메인 메뉴에서 **Processing ➔ Start Simulation** 또는 단축 아이콘을 클릭하고, 시뮬레이션이 성공하면 종료 다이얼로그 박스가 보여지고, **OK**를 클릭한다. [그림 2-12]는 시뮬레이션 결과가 패치의 T_0 클럭에서 발생하는 제어신호와 데이터의 이동을 보여준다.

[4] 채터링이란 기계적인 스위치가 열림 → 닫힘, 닫힘 → 열림으로 바뀌는 순간에 스위치에 흐르는 전류가 일정하지 않고 불안정하게 됨으로써 발생하는 잡음을 나타냄.

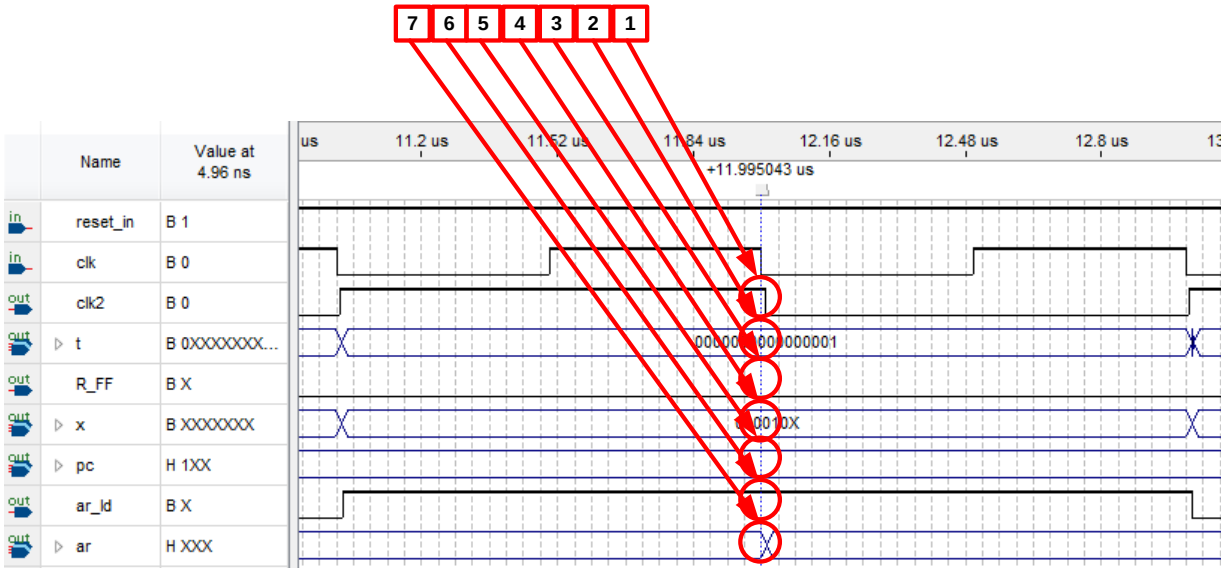


그림 2-12. LDA 명령어의 T_0 클럭에서의 시뮬레이션 결과

T_0 클럭에서 $R'T_0$: AR←PC 마이크로 연산의 각 신호에 대한 설명은 [표 2-6]과 같다.

표 2-6. $R'T_0$: AR←PC 마이크로 연산의 제어신호

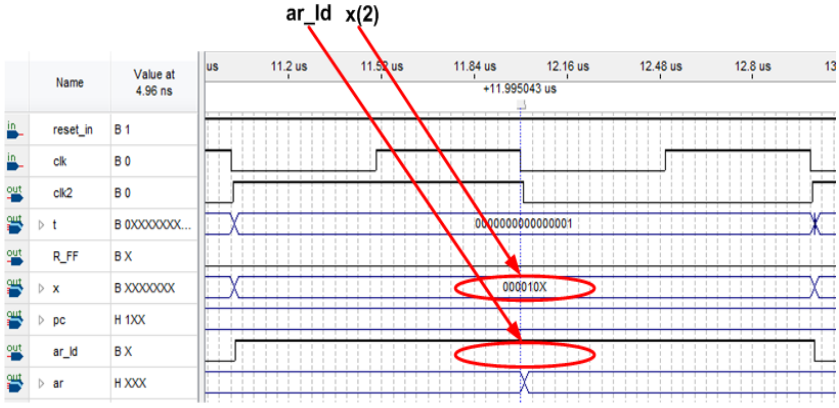
1	클럭의 하강 에지에서 레지스터 전송이 일어난다.
2	T 클럭의 값은 가장 마지막 비트만 1이므로 T_0 클럭이다. $R'T_0$에서 T_0에 해당.
3	$R'T_0$에서 R'에 해당하는 신호로 인터럽트 사이클이 아니므로 R의 값은 0이다.
4	X(2) = '1' 이므로 PC의 값이 공통버스에 전달된다.
5	ar_ld 신호가 '1'이므로 $R'T_0$ 에서 공통버스의 데이터를 저장한다.
6	PC는 ORG 100에 의해 초기 값이 100이다. ([그림 2-12]에는 전체 시간 구간의 일부만 보여주었기 때문에 표시되지 않았음.)
7	AR 레지스터에는 $R'T_0$ 에서 PC의 0x100가 공통버스를 통해 저장된다. ([그림 2-12]에는 전체 시간 구간의 일부만 보여주었기 때문에 표시되지 않았음.)

따라서 위 시뮬레이션 결과에서 X(2)와 ar_ld 신호가 발생되어 PC에 저장된 0x100 이 공통버스를 통해서 AR에 저장되는 것을 알 수 있다.

■ T_1 ~ T_5 클럭에서의 제어신호 발생

T_1 ~ T_5 클럭의 시뮬레이션 결과를 [그림 2-12]와 같이 클럭 단위로 이미지를 캡쳐하고 [표 2-7]에서 작성된 각 클럭에서 발생하는 제어신호를 시뮬레이션 결과에 아래와 같이 표시하라.

표 2-7. LDA의 시뮬레이션 결과와 제어신호

T 클럭	필요한 제어 신호와 함수	시뮬레이션 결과
T_0	X(2) = R′ AND T_0 ar_ld = R′ AND T_0	
T_1		
T_2		
T_3		
T_4		
T_5		

컴퓨터 구조 실습 결과보고서[ComLab-3]

일 시		전 공		실험시간	
학 번		이 름			
제 목	마이크로 연산과 제어함수				
실습목적	명령어는 몇 개의 마이크로 연산으로 구성되어 있으며, 이 마이크로 연산이 실행되기 위해서는 해당하는 연산이 실행될 수 있도록 제어신호가 발생해야 한다. 이 제어신호를 발생하는 논리함수를 제어함수라 한다. 본 실습을 통해 마이크로 연산이 실행되기 위해 필요한 제어 신호와 이 제어 신호를 발생하는 제어함수 및 마이크로 연산과의 관계를 이해한다.				
실습 내용	**실습결과**				

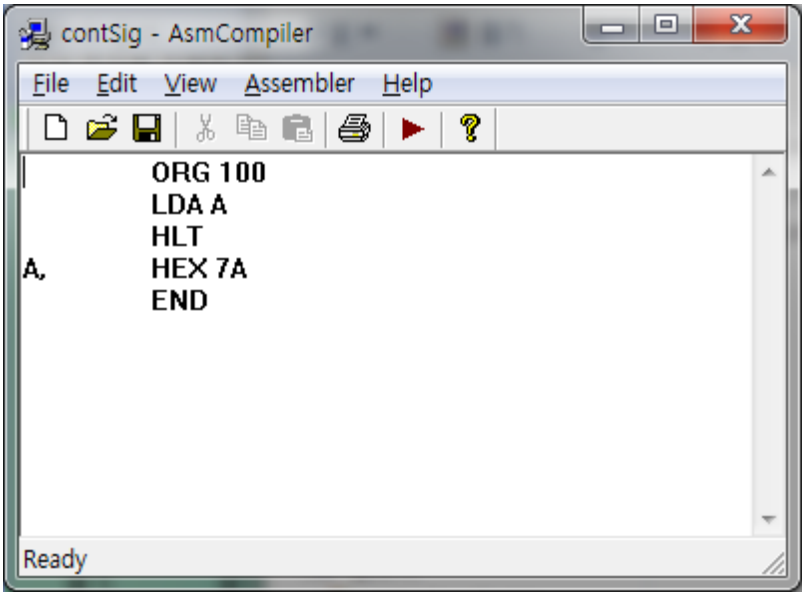
LDA 명령어의 제어신호와 동작

아래 표에 LDA 명령어의 마이크로 연산에서 레지스터 및 메모리 제어에 필요한 제어신호를 [표 2-4]에서 찾고 이의 제어함수를 채워라. 또한 각 마이크로 연산의 동작을 설명하라.

마이크로 연산	제어신호와 제어함수	동작 설명
$R'T_0$: AR←PC	X(2) = R′ AND T_0 ar_ld = R′ AND T_0	PC 값을 공통버스에 보냄 공통버스 데이터를 AR 에 저장
$R'T_1$: IR←M[AR], PC←PC +1		
$R'T_2$: D_0, …,D_7←Decode IR (12-14) AR←IR(0-11), I←IR(15)		
D'_7IT_3 : AR←M[AR]		
D_2T_4 : DR←M[AR]		
D_2T_5 : AC←DR, SC←0		

프로그램 작성 및 어셈블

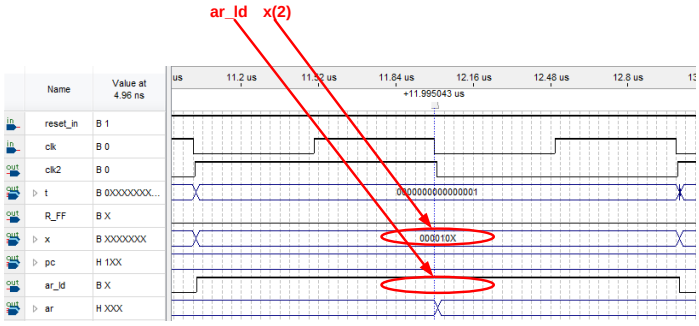
LDA 명령으로만 작성된 어셈블리 언어 프로그램을 아래 그림과 같이 작성하고 어셈블러를 이용하여 어셈블 한 후 mif 파일을 캡쳐하여라.



contSig.mif 파일 캡쳐 화면

실험키트 실행결과

DIGCOM-A1.2₩VHDL₩SimpleComputer₩DigCom-A1.2-NoStep 폴더의 simplecomA12 프로젝트를 오픈하고 시뮬레이션을 위해 reset 블록과 scounter 블록을 수정한 후, 메모리의 LPM_RAM_IO에서 LPM_FILE을 어셈블된 mif 파일명(contSig.mif)으로 수정하고 다시 컴파일 한다. 시뮬레이션을 실행했을 때 아래 그림 T_0 클럭에서과 같이 레지스터 값을 변경시키기 위한 제어신호와 제어함수를 나열하고, 그리고 시뮬레이션 결과를 캡쳐하라.

T 클럭	필요한 제어 신호와 함수	시뮬레이션 결과
T_0	X(2) = R' AND T_0 ar_ld = R' AND T_0	
T_1		
T_2		
T_3		
T_4		
T_5		

실습결과 및 토의	

2.4. 프로그램, 명령어 및 마이크로 연산의 이해

컴퓨터 프로그램은 일련의 명령어의 집합으로 구성되어 있다. 또한 각 명령어는 여러 개의 마이크로 연산으로 구성되어 있다. 하나의 마이크로 연산은 한 개의 클럭에 실행이 되므로 가장 작은 실행단위이다. 따라서 하나의 명령어가 실행되기 위해서는 메모리에 저장된 명령어를 IR(Instruction Register)로 읽어오는 패치단계, IR 에 저장된 명령어의 종류를 해독하는 디코드과정과 명령어를 실행하는 단계로 구분된다.

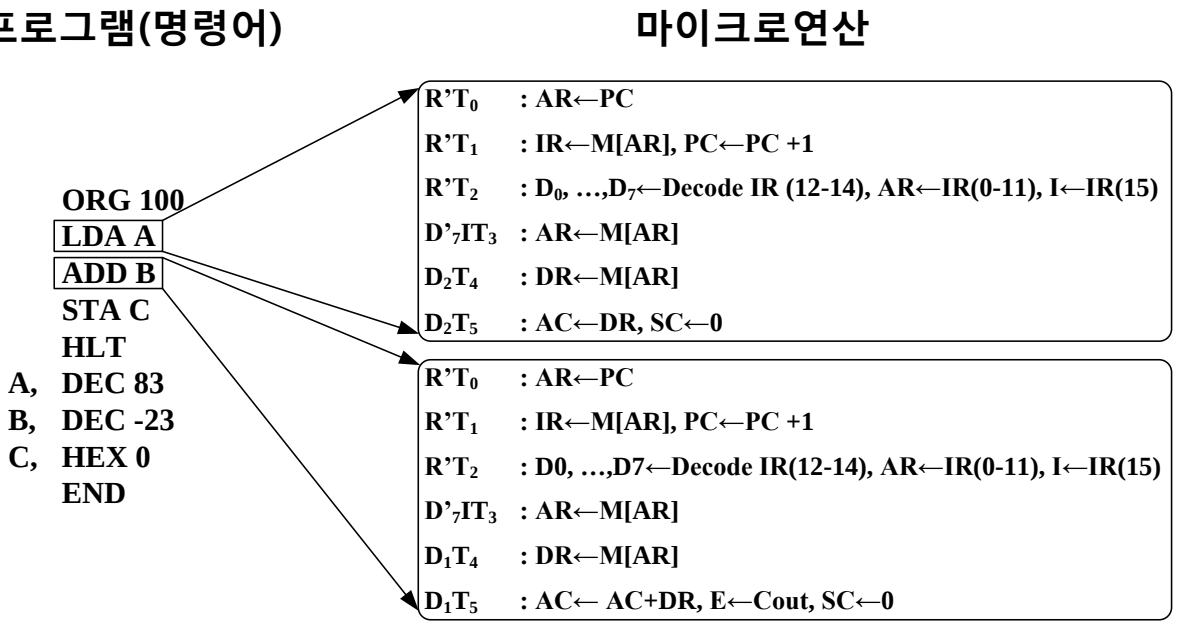


그림 2-13. 명령어와 마이크로 연산

[그림 2-13]은 두 수를 더하는 간단한 프로그램이다. 이 프로그램은 LDA, ADD, STA 및 HLT 의 4 개 명령으로 구성되어 있다. 또한 각 명령어는 명령어 종류에 따라 몇 개의 마이크로 연산으로 구성된다. 예로 LDA 명령어는 패치와 디코드 과정에 3 개의 연산이 필요하고, 실행하는 데 3 개의 연산이 필요하므로 모두 6 개의 마이크로 연산이 필요하다. 패치와 디코드는 모든 명령어에서 모두 동일하기 때문에 실제로 달라지는 부분은 명령어를 디코드한 후에 실행되는 과정이다. 이 실습에서는 주어진 프로그램의 명령어가 실행되는 과정을 마이크로 연산단위로 이해함으로써 컴퓨터 동작 원리를 이해할 수 있도록 한다.

■ 어셈블러의 이해 : 핸드 어셈블

아래 프로그램은 두 수를 감산하는 프로그램이다. 두 수를 감산하는 과정은 빼는 수에 대해서 2의 보수를 취한 다음에 더한다. 프로그램에 대한 자세한 설명은 "1.3.2. 두수를 감산하는 프로그램(ex2.s)" 부분을 참고하기 바란다.

```
ORG 100         : 프로그램이 0x100번지부터 실행된다.
```

```
        LDA SUB     : SUB번지의 데이터(-23)을 AC에 로드한다.
        CMA         : AC의 값을 1의 보수를 취한다.
        INC         : AC의 값을 1 증가하여 2의 보수를 취한다.
        ADD MIN     : MIN번지의 83을 더한다.(83+(-23의 2의 보수) = 83+23)
        STA DIF     : 더한 결과를 DIF번지에 저장한다.(0x106이 저장됨)
        HLT         : HLT 명령
MIN,    DEC 83
SUB,    DEC -23
DIF,    HEX 0
        END
```

그림 2-14. 두 수를 감산하는 프로그램

① 위의 프로그램을 어셈블러 AsmCompiler를 사용하지 않고 손으로 어셈블(hand assemble)하여 [표 2-8]를 채워라. 명령어 심볼에 대한 2진 코드는 [표 2-9]와 같다.

표 2-8. 감산 프로그램과 어셈블된 기계어 코드

프로그램 소스	주소	기계어 코드 (2 진 코드)
ORG 100		
LDA SUB		
CMA		
INC		
ADD MIN		
STA DIF		
HLT		
MIN, DEC 83		
SUB, DEC -23		
DIF, HEX 0		
END		

표 2-9. 명령어와 16진수로 표현된 기계어 코드(2진 코드)

심볼	16 진수 코드	심볼	16 진수 코드
AND	0 or 8	CLA	7800
ADD	1 or 9	CLE	7400
LDA	2 or A	CMA	7200
STA	3 or B	CME	7100

BUN	4 or C	CIR	7080
BSA	5 of D	CIL	7040
ISZ	6 of E	INC	7020
INP	F800	SPA	7010
OUT	F400	SNA	7008
SKI	F200	SZA	7004
SKO	F100	SZE	7002
INO	F080	HLT	7001
IOF	F040		

② Assembler 폴더의 AsmCompiler를 더블 클릭하여 실행하면 편집 화면이 나온다. 이 편집창에 감산하는 프로그램을 작성한다.

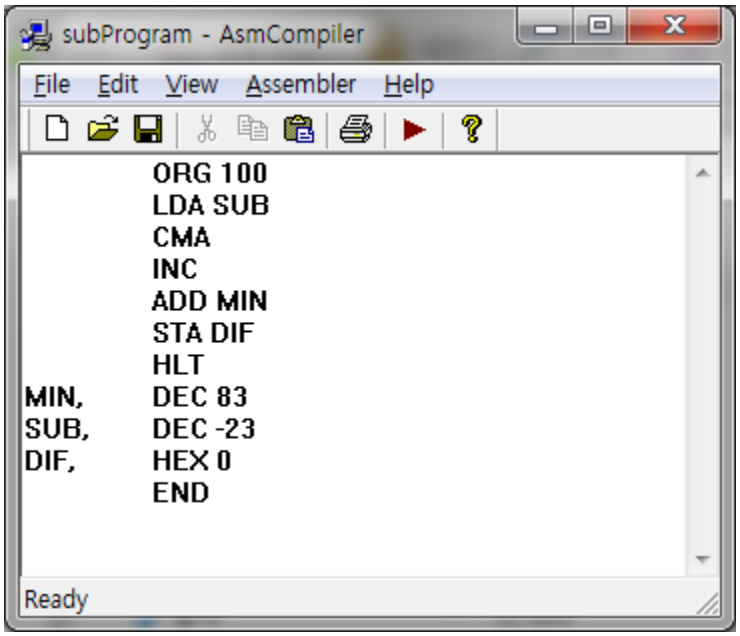


그림 2-15. 어셈블러 편집 창

③ 프로그램이 작성되면 메뉴에서 **File ➔ Save**를 선택하고 확장자 s로 파일 이름을 저장한다.

④ 작성된 프로그램을 어셈블하기 위해 메뉴에서 **Assembler ➔ Assembler**를 선택하거나 **Assembler** 단축하이콘을 클릭하여 어셈블을 한 후 확장자 mif로 저장한다. 어셈블된 2진 코드의 mif 파일 형식은 아래와 같다.

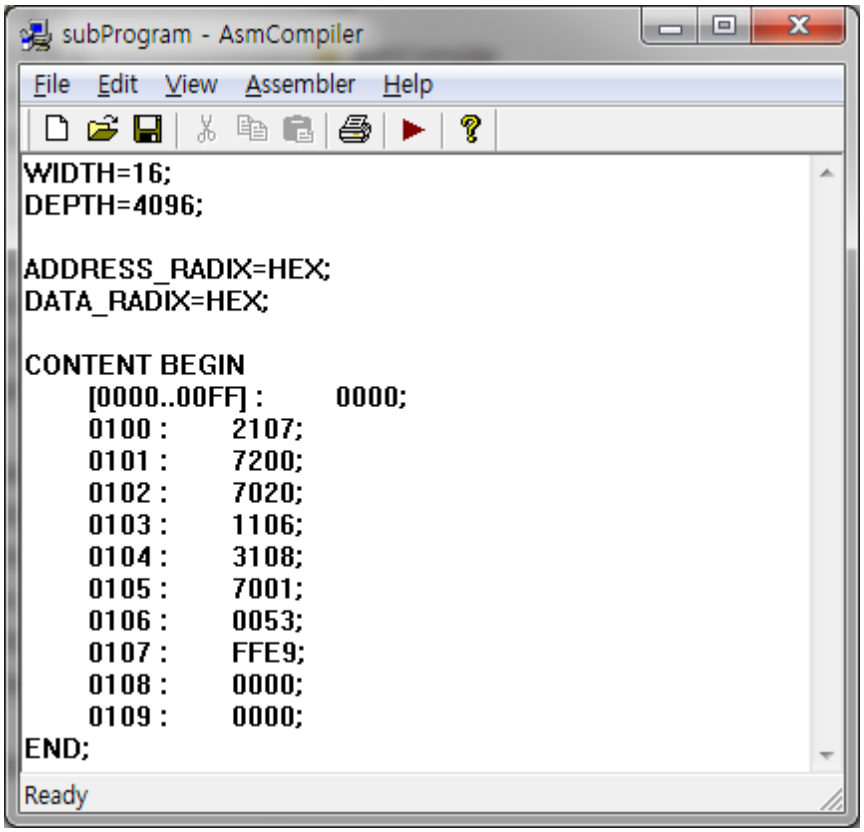


그림 2-16. 어셈블된 2진 코드의 mif 파일 형식

⑤ mif 파일의 2진 코드와 [표 2-8]에서 핸드 어셈블한 결과와 비교를 하고 틀린 부분을 찾아서 수정한다.

■ 명령어 실행결과

① [표 2-10]은 감산 프로그램의 각 명령어가 실행된 후에 레지스터와 플립플롭에 저장되는 값을 채우는 표이다. 첫번째 LDA와 두번째 CMA 명령어의 칸에 채워진 것과 같이 DR, AC 및 IR 레지스터에 저장될 값을 채워라. 이 때 앞에서 어셈블 한 2진 코드를 이용한다.

표 2-10. 감산 프로그램의 명령어 단위 실행

레지스터,FF / 명령어	레지스터(HEX)								Flip Flop	출력
	AR	PC	DR	AC	INPR	IR	TR	OUTR	I S E R IEN FGI FGO	
ORG 100	000	100	0000	0000	00	0000	0000	00	0 1 0 0 0 0 1	
LDA SUB	107	101	FFE9	FFE9	00	2107	0000	00	0 1 0 0 0 0 1	
CMA	200	102	FFE9	0016	00	7200	0000	00	0 1 0 0 0 0 1	
INC	020	103			00		0000	00	0 1 0 0 0 0 1	

ADD MIN	106	104			00		0000	00	0 1 0 0 0 0 1	
STA DIF	108	105			00		0000	00	0 1 0 0 0 0 1	
HLT	001	106			00		0000	00	0 0 0 0 0 0 1	

② 어셈블된 mif 파일을 DIGCOM-A1.2₩VHDL₩SompleComputer₩DigCom-A1.2-step 폴더로 복사한 후, DIGCOM-A1.2 프로젝트를 오픈하고 메모리의 LPM_RAM_IO에서 LPM_FILE을 어셈블된 mif 파일명으로 수정한다.

③ 기본컴퓨터를 다시 컴파일 한 후 DIGCOM-A1.2에 다운로드한다. 다운 로드를 한 후 실행모드 스위치를 명령어 모드(M1, M0=10)로 전환하고 스텝 클럭(step clock)을 한 번 누른다. 각 레지스터에 저장된 값을 보기 위해서는 레지스터 선택 스위치를 누르면 현재 출력이 되는 레지스터가 레지스터 선택 LED에 표시되고 값이 레지스터 출력 7-세그먼트에 출력되고 플립플롭의 값은 플립플롭 LED에 표시된다.

④ 스텝 클럭 스위치를 누르면서 7-세그먼트에 표시된 레지스터의 값과 LED에 표시된 값을 앞에서 작성한 [표 2-10]와 비교하라. 틀린 부분이 있으면 잘못된 부분을 찾아서 표에서 수정한다.

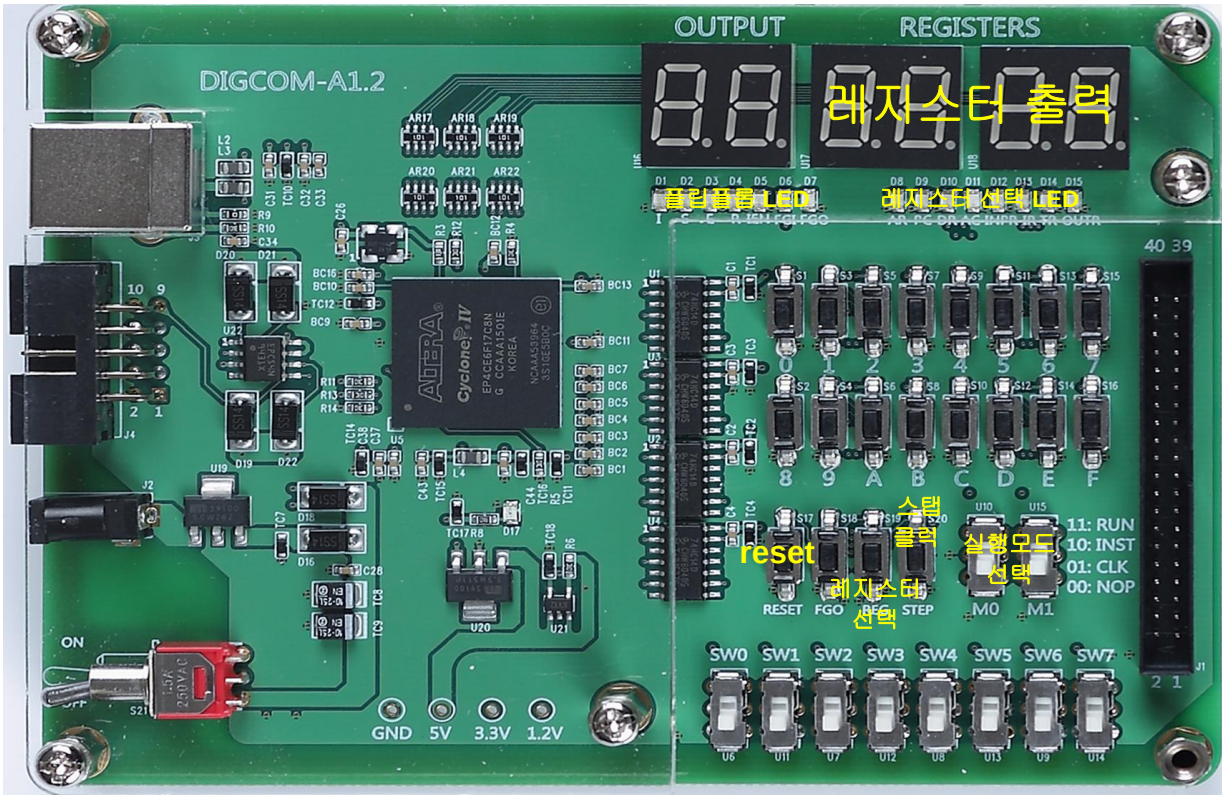


그림 2-17. DIGCOM-A1.2에서 명령어 단위 실행

■ 마이크로 연산 실행

① [표 2-11]은 감산 프로그램의 각 명령어에 대한 마이크로 연산을 나열한 표이다. 각 명령어에 대한 마이크로 연산을 첫번째 LDA와 두번째 CMA 명령어와 같이 나열하고, 각 마이크로 연산이 실행된 후에 각 레지스터와 플립플롭에 저장되는 값을 채워라.

표 2-11. 감산 프로그램의 마이크로 연산 단위 실행

nstruction	Micro operations	Register & Flip Flops								
		AR	PC	DR	AC	INPR	IR	TR	OUTR	S E R IEN FGI FGO
ORG 100		0	100	0000	0000	00	0000	0000	00	0 1 0 0 0 0 1
LDA SUB	$R'T_0$: AR←PC	100	100	0000	0000	00	0000	0000	00	0 1 0 0 0 0 1
	$R'T_1$: IR←M[AR], PC←PC +1	100	101	0000	0000	00	2107	0000	00	0 1 0 0 0 0 1
	$R'T_2$: D_0, …,D_7←Decode IR (12-14) AR←IR(0-11), I←IR(15)	107	101	0000	0000	00	2107	0000	00	0 1 0 0 0 0 1
	D'_7IT_3 : AR←M[AR]	107	101	0000	0000	00	2107	0000	00	0 1 0 0 0 0 1
	D_2T_4 : DR←M[AR]	107	101	FFE9	0000	00	2107	0000	00	0 1 0 0 0 0 1
	D_2T_5 : AC←DR, SC←0	107	101	FFE9	FFE9	00	2107	0000	00	0 1 0 0 0 0 1
CMA	$R'T_0$: AR←PC									
	$R'T_1$: IR←M[AR], PC←PC +1									
	$R'T_2$: D_0, …,D_7←Decode IR (12-14) AR←IR(0-11), I←IR(15)									
	rB_9 : AC ← /AC									
INC										
ADD MIN										

STA DIF										
HLT										

② 실행모드 스위치를 클럭 모드(M1, M0=01)로 전환하고 스텝 클럭을 한 번 누른다. 스위치를 한번 누를 때마다 마이크로 연산이 하나씩 실행된다. 앞에서와 같은 방법으로 각 레지스터에 저장된 값과 LED에 표시된 값을 작성한 [표 2-11]과 비교하라. 틀린 부분이 있으면 잘못된 부분을 찾아서 표에서 수정한다.

컴퓨터 구조 실습 결과보고서[ComLab-4]

일 시		전 공		실험시간	
학 번			이 름		
제 목	프로그램, 명령어 및 마이크로 연산의 이해				
실습목적	하나의 프로그램은 여러 컴퓨터 명령어로 이루어 졌으며, 하나의 컴퓨터 명령어는 몇 개의 마이크로 연산으로 이루어 졌다. 본 실습에서는 하나의 마이크로 연산이 실행됨으로서 하나의 명령어가 실행되는 과정을 실습을 통해 이해하고, 여러 명령어가 실행됨으로써 하나의 프로그램이 실행되는 과정을 이해한다.				
실습 내용	**실습결과**				

핸드 어셈블

아래 어셈블리 언어로 작성된 프로그램을 핸드 어셈블한 후 어셈블러로 어셈블 한 결과와 비교하라.

프로그램 소스	주소	핸드어셈블	어셈블러 어셈블 결과
ORG 100 LDA SUB CMA INC ADD MIN STA DIF HLT MIN, DEC 83 SUB, DEC -23 DIF, HEX 0 END			

마이크로 연산과 동작 설명

다음 프로그램의 각 명령어에 대한 마이크로 연산을 나열하고 동작을 설명하라.

Instruction	Micro operations	Operation Description
ORG 100		
LDA SUB	$R'T_0$: AR←PC	PC 의 초기주소 0x100 가 AR 에 저장
	$R'T_1$: IR←M[AR], PC←PC +1	0x100 번지 명령어 LDA SUB(0x2107)가 IR 에 저장
	$R'T_2$: D_0, …,D_7←Decode IR (12-14) AR←IR(0-11), I←IR(15)	Opcode 가 디코드된 결과 D_2 가 출력됨. 주소 SUB(0x107)가 AR 에 저장, I 에 '0'이 저장됨.
	D'_7IT_3 : AR←M[AR]	직접주소 명령이므로 no operation
	D_2T_4 : DR←M[AR]	0x107 번지 데이터 -23 이 DR 로 전송.
	D_2T_5 : AC←DR, SC←0	DR 의 83 이 AC 로 전송, SC 가 초기화 됨.
CMA	$R'T_0$: AR←PC	
	$R'T_1$: IR←M[AR], PC←PC +1	
	$R'T_2$: D_0, …,D_7←Decode IR (12-14) AR←IR(0-11), I←IR(15)	
	rB_9 : AC ← /AC	
INC		
ADD MIN		
STA DIF		
HLT		

명령어 단위실행 결과

아래표는 명령어 단위로 프로그램을 실행했을 때 각 레지스터에 저장되는 값이다. INC 명령어 이후 DR, AC 및 IR 레지스터에 저장되는 값을 채워 넣어라.

레지스터,FF / 명령어	레지스터(HEX)								Flip Flop
	AR	PC	DR	AC	INPR	IR	TR	OUTR	I S E R IEN FGI FGO
ORG 100	000	100	0000	0000	00	0000	0000	00	0 1 0 0 0 0 1
LDA SUB	107	101	FFE9	FFE9	00	2107	0000	00	0 1 0 0 0 0 1
CMA	200	102	FFE9	0016	00	7200	0000	00	0 1 0 0 0 0 1
INC	020	103			00		0000	00	0 1 0 0 0 0 1
ADD MIN	106	104			00		0000	00	0 1 0 0 0 0 1
STA DIF	108	105			00		0000	00	0 1 0 0 0 0 1
HLT	001	106			00		0000	00	0 0 0 0 0 0 1

DIGCOM-A1.2 에 위의 프로그램을 어셈블하여 다운로드 한 후 명령어모드로 실행시켰을 때 위에서 채워 넣은 레지스터의 값과 비교하고, 비교한 결과 위의 표와 다른 부분을 찾고 차이가 생긴 원인에 대해서 설명하라.

차이점	원 인

마이크로 연산 단위 실행

아래표는 마이크로 연산 단위로 프로그램을 실행했을 때 각 레지스터에 저장되는 값이다. CMA 명령어 이후 DR, AC 및 IR 레지스터에 저장되는 값을 채워 넣어라

Instruction	Micro operations	Register & Flip Flops								
		AR	PC	DR	AC	INPR	IR	TR	OUTR	I S E R IEN FGI FGO
ORG 100		0	100	0000	0000	00	0000	0000	00	0 1 0 0 0 0 1
LDA SUB	$R'T_0$: AR←PC	100	100	0000	0000	00	0000	0000	00	0 1 0 0 0 0 1
	$R'T_1$: IR←M[AR], PC←PC +1	100	101	0000	0000	00	2107	0000	00	0 1 0 0 0 0 1
	$R'T_2$: D_0, ...,D_7←Decode IR (12-14) AR←IR(0-11), I←IR(15)	107	101	0000	0000	00	2107	0000	00	0 1 0 0 0 0 1
	D'_7IT_3 : AR←M[AR]	107	101	0000	0000	00	2107	0000	00	0 1 0 0 0 0 1

	D_2T_4 : DR←M[AR]	107	101	FFE9	0000	00	2107	0000	00	0 1 0 0 0 0 1
	D_2T_5 : AC←DR, SC←0	107	101	FFE9	FFE9	00	2107	0000	00	0 1 0 0 0 0 1
CMA	$R'T_0$: AR←PC									
	$R'T_1$: IR←M[AR], PC←PC +1									
	$R'T_2$: D_0, …,D_7←Decode IR (12-14) AR←IR(0-11), I←IR(15)									
	rB_9 : AC ← /AC									
INC										
ADD MIN										
STA DIF										
HLT										

<table>
<tr><td></td><td>DIGCOM-A1.2 를 클럭모드로 실행시켰을 때 위에서 채워 넣은 레지스터의 값과 비교하고, 비교한 결과 위의 표와 다른 부분을 찾고 차이가 생긴 원인에 대해서 설명하라.

<table><tr><th>차이점</th><th>원 인</th></tr><tr><td></td><td></td></tr></table></td></tr>
<tr><td>실습결과
및 토의</td><td></td></tr>
</table>

2.5. 어셈블리 언어 프로그래밍

컴퓨터 시스템은 하드웨어와 소프트웨어로 구성되며, 하드웨어는 컴퓨터의 장치들을 의미하고 소프트웨어는 컴퓨터에서 실행되어야 할 프로그램을 말한다. 프로그램을 작성하는 일은 일련의 기계 명령어를 표현하는 것인데 컴퓨터 내부의 기계어는 이진 형태로 되어 있으므로 보고 이해하기가 어렵기 때문에 영문자와 숫자로 표현되는 기호로 프로그램을 작성하는 것이 더욱 편리하다. 이러한 기호를 어셈블리 언어라 하고 어셈블리 언어(assembly language)로 작성된 프로그램을 어셈블리 언어 프로그램(assembly language program)이라 한다. 따라서 작성된 프로그램이 실행되기 위해서는 기호로 표시된 프로그램이 하드웨어가 인식할 수 있는 이진 프로그램으로 번역을 해야 하며 이러한 번역 프로그램을 어셈블러(assembler)라 한다. 이 실습에서는 어셈블리 언어로 작성된 프로그램을 작성하고 어셈블러를 이용해서 기계어로 번역한 후 하드웨어 메모리에 입력하여 DIGCOM-A1.2 에 다운로드해서 실행하는 과정을 실습한다. 본 실습에서는 반복 가산 방법에 의하여 두 개의 양수를 곱하는 프로그램을 작성한다. 예을 들어 5 x 4 곱셈 연산을 위하여 5 를 네번 더해서 곱을 만든다. 즉, 5 + 5 + 5 + 5 를 실행하는 프로그램을 작성한다.

- 프로그램은 100번지부터 시작하며 5를 4번 더하는 과정이므로 카운터와 피연산자를 메모리에 저장한 후 카운터 수만큼 피 연산자를 더하는 과정으로 곱셈 연산을 하는 프로그램을 작성한다.

  ① Assembler 폴더의 AsmCompiler를 더블 클릭하여 "2.4. 프로그램, 명령어 및 마이크로 연산의 이해"실습에서와 같은 방법으로 곱셈 프로그램을 작성한다.
  ② 프로그램이 작성되면 확장자 s로 파일 이름을 저장한다.
  ③ 프로그램을 어셈블을 한 후 확장자 mif로 저장한다.
  ④ 어셈블된 mif 파일을 DIGCOM-A1.2₩VHDL₩SompleComputer₩DigCom-A1.2-step 폴더로 복사한 후, DIGCOM-A1.2 프로젝트를 오픈하고 메모리의 LPM_RAM_IO에서 LPM_FILE을 어셈블된 mif 파일명으로 수정한다.
  ⑤ 기본컴퓨터를 다시 컴파일 한 후 DIGCOM-A1.2에 다운로드하여 실행한 후 7-세그먼트에 나온 결과를 확인한다.

- 앞에서 작성한 프로그램을 서브루틴을 사용할 수 있도록 수정한다. 더하는 부분을 서브루틴으로 작성하고 앞에서와 같이 카운터 회수만큼 서브루틴을 호출한다.

  ① 앞에서 프로그램을 작성한 방법과 동일하게 어셈블러를 실행하고 편집창에 프로그램을 작성한다. 서브루틴을 호출하기 위해서는 피연산자를 전달해야 한다. 피연산자는 "1.3.6. 파라미터 링키지를 보이는 프로그램(ex6.s)"에서와 같이 서브루틴을 호출하는 명령어 다음에 두어 서브루틴으로 전달한다. 서브루틴은 더하는 횟수만큼 호출되어 결과는 7-세그먼트에 출력된다.
  ② 작성된 프로그램을 어셈블한 후, 어셈블된 mif 파일을 앞에서 했던 것과 같이 프로젝트 폴더로 옮긴 후 컴파일하여 DIGCOM-A1.2에 다운로드 하여 실행하여 결과를 확인한다.
  ③ 4 x 5 연산외에 카운터 값과 피연산자를 바꿔서 다른 곱셈을 했을 때 결과가 맞는지를

확인한다.

- 프로그램의 디버깅

  작성된 프로그램은 DIGCOM-A1.2에 다운로드한 후 한번에 실행되어 결과를 확인할 수 있으나 원하는 결과가 나오지 않을 때는 실행 모드 스위치를 명령어 또는 클럭 모드로 전환해서 프로그램 중간의 레지스터 값을 확인해서 프로그램의 어느 부분에서 잘못되었는지를 찾을 수 있다. 다른 방법으로는 실습키트에서 실행하기 전에 시뮬레이션으로 결과값을 확인하는 것인데, 시뮬레이션 결과 타이밍 도를 보는 것이 복잡하기 때문에 앞의 방법이 더 편리하다. 그러나 스텝모드로도 디버깅이 안 될 때는 시뮬레이션 결과 타이밍 도를 분석해서 원인을 찾아야 한다.

컴퓨터 구조 실습 결과보고서[ComLab-5]

일 시		**전 공**		**실험시간**	
학 번			**이 름**		
제 목	어셈블리 언어 프로그래밍				
실습목적	컴퓨터 시스템은 하드웨어와 소프트웨어로 구성되며, 하드웨어는 컴퓨터의 장치들을 의미하고 소프트웨어는 컴퓨터에서 실행되어야 할 프로그램을 말한다. 프로그램을 작성한다는 것은 기계 명령어를 표현하는 것인데 컴퓨터 내부의 기계어는 이진 형태로 되어 있으므로 이해하기 어려우므로 이를 영문자와 숫자로 표현한 것을 어셈블리 언어라 한다. 어셈블리 언어(assembly language)로 작성된 프로그램을 어셈블리 언어 프로그램(assembly language program)이라 하며, 이를 컴퓨터가 인식할 수 있는 2 진 코드로 번역하는 프로그램을 어셈블러(assembler)라 한다. 이 실습에서는 어셈블리 언어로 작성된 프로그램을 작성하고 어셈블러를 이용해서 기계어로 번역한 후 기본 컴퓨터의 메모리에 포함하여 컴파일해서 DIGCOM-A1.2 에 다운로드하여 실행하는 과정을 실습함으로써 어셈블리 프로그래밍 방법을 익힌다.					
실습 내용	**실습결과**				
프로그래밍과 어셈블 I	프로그램은 100번지부터 시작하며 5를 4번 더하는 과정이므로 카운터와 피연산자를 메모리에 저장한 후 카운터 수만큼 피 연산자를 더하는 과정으로 곱셈 연산을 하는 프로그램을 작성하라. 이 프로그램에서는 서브루틴을 사용하지 않고 더하는 횟수만큼 반복실행하는 방법으로 작성하라. 아래 표에 소스코드와 어셈블된 mif 파일을 보여라.				

프로그램 소스 코드	mif 파일

<table>
<tr><td>프로그래밍과 어셈블 II</td><td>앞에서 작성한 프로그램과 같은 동작을 하는 프로그램을 작성하되 더하는 부분을 서브루틴을 호출하도록 하라. 아래 표에 소스코드와 어셈블된 mif 파일을 보여라.
<table><tr><th>프로그램 소스 코드</th><th>mif 파일</th></tr><tr><td></td><td></td></tr></table></td></tr>
<tr><td>프로그램 실행</td><td>DIGCOM-A1.2₩VHDL₩SompleComputer₩DigCom-A1.2-step 폴더의 DIGCOM-A1.2 프로젝트를 오픈하고 메모리의 LPM_RAM_IO에서 LPM_FILE을 어셈블된 mif 파일명으로 수정한 후 다시 컴파일한다. 생성된 pof 파일을 DIGCOM-A1.2에 다운로드하여 실행한 후 7-세그먼트에 나온 결과를 확인하고, 아래 표에 DIGCOM-A1.2 결과 이미지를 보여라.
<table><tr><th>DIGCOM-A1.2 실행 결과 이미지</th></tr><tr><td></td></tr></table></td></tr>
</table>

실습결과 및 토의	

2.6. 새로운 명령어의 추가

기본 컴퓨터는 7 개의 메모리 참조명령, 12 개의 레지스터 명령과 6 개의 입출력 명령을 사용한다. opcode 는 모두 4 비트가 할당되므로 명령어의 개수에 제한이 있다. 따라서 기존의 명령이 충분하지 않기 때문에 때로 다양한 기능을 제공하는 새로운 명령을 추가해야 할 필요성을 느끼게 된다. 본 실습에서는 기존의 명령에 새로운 명령을 추가하고 이를 기존에 이미 설계한 기본컴퓨터의 코드를 수정해서 구현하는 과정을 설명한다. 새로운 명령을 추가하는 과정은 추가할 명령을 정한 후에 기존의 명령에서 대체할 명령어를 선택하고 새로운 명령어의 2 진 코드 값을 정한 후에 이 명령어를 구성하는 마이크로 연산을 정한다. 다음 기본 컴퓨터 프로젝트의 제어부에서 기존 명령에 해당하는 제어신호를 삭제하고 대신에 새로운 명령어에 해당하는 제어신호를 발생하도록 VHDL 코드를 추가한다. 기본 컴퓨터 VHDL 코드를 수정한 후에 새로운 명령을 테스트 하기 위해 이 명령어를 포함하는 프로그램을 작성하고 어셈블을 해야 한다. 그러나 어셈블러는 기존의 명령어만 어셈블 하므로 기존의 명령어를 사용해서 프로그램 한 후 mif 파일에서 추가된 명령어의 2 진 코드로 수정하고 기본 컴퓨터의 메모리에 포함해서 컴파일한 후 실습키트에 다운로드하여 실행한다.

기존의 명령어를 대신해서 추가할 수 있는 메모리 참조명령은 [표 2-12]와 같다.

표 2-12. 새로운 메모리 참조 명령어

Symbol	Opcode	Symbolic desination	Description in words
XOR	000	AC ← AC XOR M[EA]	Exclusive-OR to AC
ADM	001	M[EA] ← M[EA] + AC	Add AC to memory
SUB	010	AC ← AC – M[EA]	Subtract memory from AC
XCH	011	AC ← M[EA], M[EA] ← AC	Exchange AC and memory
SEQ	100	If(M[EA = AC] then (PC ← PC +1)	Skip on equal
BPA	101	If(AC > 0) then (PC ← EA)	Branch if AC positive and non-zero

위 명령에서 XCH 명령어를 추가하는 과정을 실습하는 과정을 설명한다.

- 추가할 명령어 XCH에 대한 opcode와 마이크로 연산을 정한다.

  ① [표 2-13]에서 XCH의 opcode는 "010" 이며 이 opcode는 기존의 명령어 STA에 해당한다. 따라서 STA 명명을 XCH 명령어로 대체한다.

  ② XCH에 해당하는 마이크로 연산을 [표 2-13]과 같이 정한다.

표 2-13. XCH 명령어의 마이크로 연산

명령어	마이크로 연산
XCH	$R'T_0$: AR ← PC
	$R'T_1$: IR ← M[AR], PC ← PC + 1
	$R'T_2$: D_0, …,D_7 ← Decode IR(12-14), AR ← IR(0-11), I ← IR(15)
	D'_7IT_3 : AR ← M[AR]
	D_3T_4 : DR ← M[AR]
	D_3T_5 : M[AR] ← AC, AC ← DR, SC ← 0

[표 2-13]의 마이크로 연산에서 T_3 클럭까지는 일반적인 패치, 디코드와 간접주소 지정방식에서 유효 주소를 읽어오는 과정이므로 동일하고, T_4에서는 AC에 저장된 값을 메모리에서 DR로 읽어온 후에 T_5에서 AC의 값을 메모리에 저장하고, DR의 값을 AC로 전달한다. 결과적으로 AC에 있던 값과 메모리에 있는 값이 교환된다.

- STA 명령이 XCH 명령으로 대체되므로 기본 컴퓨터의 제어부에서 STA 명령어의 제어신호를 발생하는 코드를 삭제해야 한다. 패치 디코드는 모든 명령어에 공통적으로 포함이 되는 부분이며, STA 명령의 T_4 클럭에서 마이크로 연산은 "D_3T_4 : M[AR] ← AC, SC ← 0"이므로 이 마이크로 연산을 실행하는 부분을 삭제한다.

  ① Quartus Prime을 실행하고 DIGCOM-A1.2₩VHDL₩SimpleComputer₩ComLab_Lecturer₩DigiCom-A1.2_NewInst 폴더의 simplecomA12 프로젝트를 오픈한다.

  ② [그림 2-18]의 BDF 화면에서 control block을 더블 클릭하여 control.vhd 파일을 오픈한다.

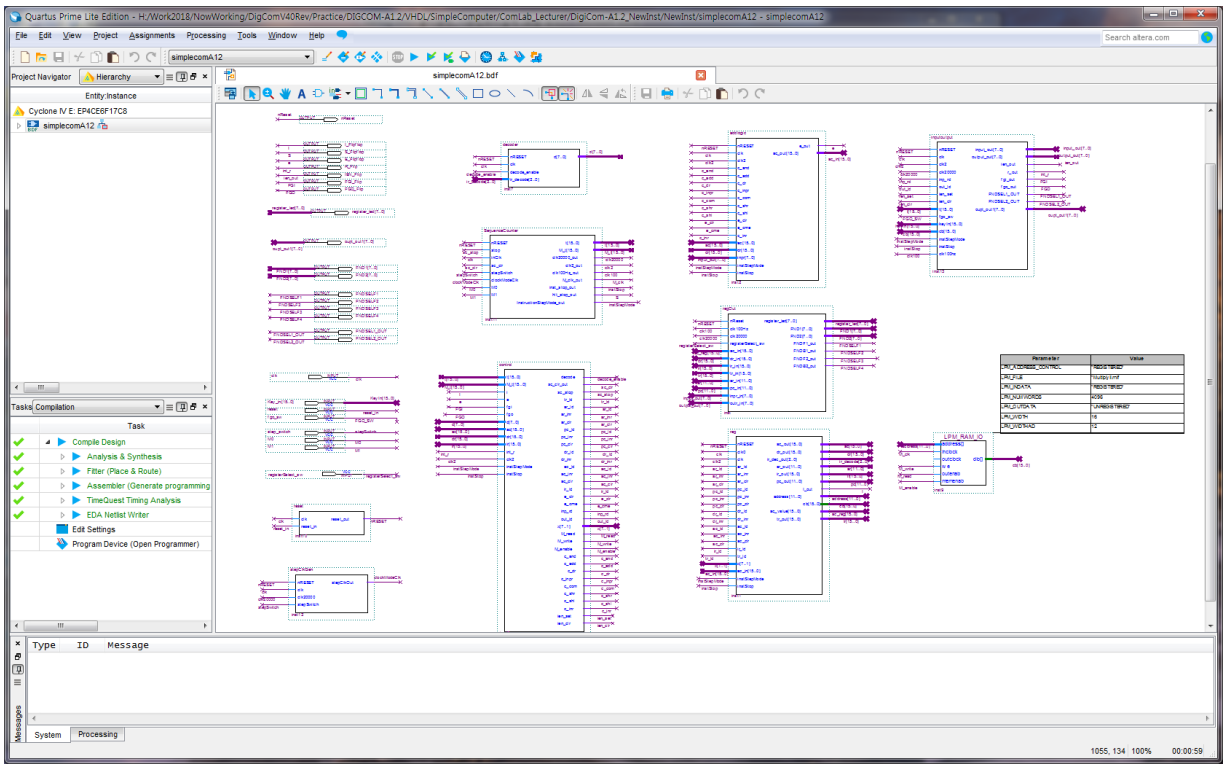


그림 2-18. simplecomA12 프로젝트 화면

③ D_3T_4에서 D_3는 STA를 디코드한 결과이며, T_4는 5번째 T클럭을 나타내며(T_0부터 시작함.) 이 조건에서 AC의 값이 메모리에서 저장된다. 따라서 [그림 2-19]와 같이 D_3T_4에서 메모리에 저장하기 위한 제어신호 M_write와 M_enable을 발생하는 부분을 삭제한다.

```
211         M_write <= (int_R and M_t(1) and clk2)                -- interrupt cycle
212 --                 or (d(3) and M_t(4) and clk2)                -- memory STA
213                    or (d(5) and M_t(4) and clk2)                -- memory BSA
214                    or (d(6) and M_t(6) and clk2);               -- memory ISZ
215
216         M_enable <= (M_t(1) and clk2)                           -- interrupt cycle
217 --                 or (d(3) and M_t(4) and clk2)                -- memory STA
218                    or (d(5) and M_t(4) and clk2)                -- memory BSA
219                    or (d(6) and M_t(6) and clk2)                -- memory ISZ
220                    or (not d(7) and i and M_t(3) and clk2)          -- decode
221                    or (d(0) and M_t(4) and clk2)                -- memory AND
222                    or (d(1) and M_t(4) and clk2)                -- memory ADD
223                    or (d(2) and M_t(4) and clk2)                -- memory LDA
224                    or (d(6) and M_t(4) and clk2);               -- memory ISZ
```

그림 2-19. STA에서 메모리에 저장하는 부분 삭제

④ 또한 D_3T_4에서 AC의 값이 공통버스를 통해 메모리에 전달이 되며, 공통버스에서 AC 값을 선택하기 위해서는 D_3T_4에서 x(4) 제어신호를 발생시켜야 한다. 따라서 [그림 2-20] 과 같이 이 신호를 발생하는 부분을 삭제한다.

```
154
155 --          x(4) <= (d(3) and t(4))                -- memory STA, M[AR] <- AC
156 --                  or (p_buf and ir(10));         -- I/O OUT, OUTR <- AC(0-7)
157            x(4) <=  (p_buf and ir(10));   -- I/O OUT, OUTR <- AC(0-7)
158
```

그림 2-20. AC를 공통버스로 전달하는 부분 삭제

⑤ D_3T_4에서 [그림 2-21]과 같이 시퀀스 카운터(sequence counter)를 초기화 하는 부분을 삭제한다.

```
192          sc_clr <= p_buf              -- I/O reference instruction
193               or r_buf                -- register reference instruction
194               or int_s_clr            -- interrupt cycle
195               or (d(0) and t(5))      -- memory AND
196               or (d(1) and t(5))      -- memory ADD
197               or (d(2) and t(5))      -- memory LDA
198     --        or (d(3) and t(4))      -- memory STA
199               or (d(4) and t(4))      -- memory BUN
200               or (d(5) and t(5))      -- memory BSA
201               or (d(6) and t(6))      -- memory ISZ
202               or (r_buf and t(2));
203          sc_clr_out <= sc_clr;
```

그림 2-21. 시퀀스 카운터를 초기화 하는 부분을 삭제

■ 기본 컴퓨터의 제어부에서 XCH 명령어의 제어신호를 발생하는 코드를 추가한다. XCH를 실행하는 과정은 서로 교환할 두 데이터가 하나는 AC에 있고 다른 하나는 메모리에 있는 상태에서, 첫번째 클럭에서 메모리 데이터를 DR(Data Register)로 읽어오고, 다음 클럭에서 AC와 교환한다. 명령어 패치와 디코드를 마친 후에 명령어가 실행되므로 T_4, T_5 클럭에서 두 마이크로 연산이 실행된다.

① D_3T_4에서 DR의 load(dr_ld) 신호를 발생하도록 [그림 2-22]와 같이 추가한다.

```
117          dr_ld <= (d(0) and t(4))          -- memory AND
118             or (d(1) and t(4))             -- memory ADD
119             or (d(2) and t(4))             -- memory LDA
120             or (d(3) and t(4))             -- modified memory XCH
121             or (d(6) and t(4));            -- memory ISZ
122
```

그림 2-22. DR의 load 신호 발생 추가

② 메모리 데이터는 공통버스를 통해서 DR에 전달되므로 D_3T_4에서 메모리 데이터를 공통버스로 보내는 신호 x(7)를 발생하도록 [그림 2-23]과 같이 추가한다.

```
164          x(7) <= (not int_R and t(1))      -- Fetch, IR <- M[AR]
165             or (not d(7) and i and t(3))   -- indirect, AR <- M[AR]
166             or (d(0) and t(4))             -- memory AND, DR <- M[AR]
167             or (d(1) and t(4))             -- memory ADD, DR <- M[AR]
168             or (d(2) and t(4))             -- memory LDA, DR <- M[AR]
169             or (d(3) and t(4))             -- modified memory XCH
170             or (d(6) and t(4));            -- memory ISZ, DR <- M[AR]
```

그림 2-23. 메모리 데이터를 공통버스로 보내는 x(7) 신호 발생 추가

③ 메모리에서 데이터를 읽기 위해 D_3T_4에서 M_read와 M_enable 신호를 발생하는 부분을 추가한다.

```
217       M_enable <= (M_t(1) and clk2)                          -- interrupt cycle
218   --              or (d(3) and M_t(4) and clk2)              -- memory STA
219                   or (d(3) and M_t(4) and clk2)              -- modified XCH
220                   or (d(5) and M_t(4)and clk2)              -- memory BSA
221                   or (d(6) and M_t(6) and clk2)              -- memory ISZ
222                   or (not d(7) and i and M_t(3) and clk2)       -- decode
223                   or (d(0) and M_t(4) and clk2)              -- memory AND
224                   or (d(1) and M_t(4) and clk2)              -- memory ADD
225                   or (d(2) and M_t(4) and clk2)              -- memory LDA
226                   or (d(6) and M_t(4) and clk2);                -- memory ISZ
227
228       M_read <= (not int_R and M_t(1) and clk2)                  -- Fetch cycle
229                   or (not d(7) and i and M_t(3) and clk2)       -- decode
230                   or (d(0) and M_t(4) and clk2)              -- memory AND
231                   or (d(1) and M_t(4) and clk2)              -- memory ADD
232                   or (d(2) and M_t(4) and clk2)              -- memory LDA
233                   or (d(3) and M_t(4) and clk2)              -- modified XCH
234                   or (d(6) and M_t(4) and clk2);                -- memory ISZ
```

그림 2-24. 메모리 데이터를 읽는 M_read, M_enable 신호 발생 부분을 추가

④ D_3T_5에서 AC 값을 메모리로 전송하기 위해서 M_write와 M_enable 신호가 발생하도록 [그림 2-25]와 같이 추가한다.

```
211       M_write <= (int_R and M_t(1) and clk2)                  -- interrupt cycle
212   --              or (d(3) and M_t(4) and clk2)              -- memory STA
213                   or (d(3) and M_t(5) and clk2)              -- modified XCH
214                   or (d(5) and M_t(4) and clk2)              -- memory BSA
215                   or (d(6) and M_t(6) and clk2);                -- memory ISZ
216
217       M_enable <= (M_t(1) and clk2)                          -- interrupt cycle
218   --              or (d(3) and M_t(4) and clk2)              -- memory STA
219                   or (d(3) and M_t(5) and clk2)              -- modified XCH
220                   or (d(3) and M_t(4) and clk2)              -- modified XCH
221                   or (d(5) and M_t(4)and clk2)              -- memory BSA
222                   or (d(6) and M_t(6) and clk2)              -- memory ISZ
223                   or (not d(7) and i and M_t(3) and clk2)       -- decode
224                   or (d(0) and M_t(4) and clk2)              -- memory AND
225                   or (d(1) and M_t(4) and clk2)              -- memory ADD
226                   or (d(2) and M_t(4) and clk2)              -- memory LDA
227                   or (d(6) and M_t(4) and clk2);                -- memory ISZ
```

그림 2-25. 메모리에 쓰기 위해 M_write, M_enable 신호 발생 추가

⑤ AC 값은 공통버스를 통해 메모리에 저장되므로 D_3T_5에서 x(4) 신호를 발생하는 부분을 추가한다.

```
155   --        x(4) <= (d(3) and t(4))          -- memory STA, M[AR] <- AC
156             x(4) <= (d(3) and t(5))       -- modified XCH
157                   or (p_buf and ir(10));   -- I/O OUT, OUTR <- AC(0-7)
158
```

그림 2-26. 공통버스에 AC를 전달하는 x(4) 제어신호 발생을 추가

⑥ DR이 AC로 전달되는 과정은 공통버스를 통하지 않고 직접 전달되며, 이 때 ALU(Adder and Logic Unit)에서는 DR에서 입력되는 데이터를 AC로 전달한다. 따라서 D_3T_5에서 AC의 load 신호를 발생하는 부분을 추가한다.

```
125         ac_ld <= (d(2) and t(5))       -- memory LDA
126            or (d(0) and t(5))          -- memory AND
127            or (d(1) and t(5))             -- memory ADD
128            or (d(3) and t(5))          -- modified XCH
129            or (r_buf and ir(9))           -- register CMA
130            or (r_buf and ir(7))           -- register CIR
131            or (r_buf and ir(6))        -- register CIL
132            or (p_buf and ir(11));         -- I/O INP
```

그림 2-27. ac_ld 신호를 발생하는 부분을 추가

⑦ ALU(Adder and Logic Unit)에서는 DR에서 입력되는 데이터를 AC로 전달하기 위해 ALU에 c_dr 신호를 발생하는 부분을 추가한다. c_dr은 명령어를 디코드한 결과 ALU에서 DR을 아무런 연산없이 AC로 전달할 때 발생하는 신호이다.

```
172         c_and <= d(0) and t(5);           -- AND micro operation
173         c_add <= d(1) and t(5);           -- ADD
174         c_dr <= (d(2) and t(5))           -- LDR
175                 or (d(3) and t(5));       -- modified XCH
176         c_inpr <= p_buf and ir(11);          -- INP
177         c_com <= r_buf and ir(9);         -- CMA
178         c_shr <= r_buf and ir(7);         -- SHR
179         c_shl <= r_buf and ir(6);         -- SHL
180         c_inr <= r_buf and ir(5);         -- INC
```

그림 2-28. c_dr 신호를 발생하는 부분을 추가

⑧ 마지막으로 D_3T_5에서시퀀스 카운터를 초기화시키는 부분을 추가한다.

```
192         sc_clr <= p_buf               -- I/O reference instruction
193              or r_buf                 -- register reference instruction
194              or int_s_clr             -- interrupt cycle
195              or (d(0) and t(5))       -- memory AND
196              or (d(1) and t(5))       -- memory ADD
197              or (d(2) and t(5))       -- memory LDA
198     --       or (d(3) and t(4))       -- memory STA
199               or (d(3) and t(5))      -- modified XCH
200              or (d(4) and t(4))       -- memory BUN
201              or (d(5) and t(5))       -- memory BSA
202              or (d(6) and t(6))       -- memory ISZ
203              or (r_buf and t(2));
204         sc_clr_out <= sc_clr;
205         --p <= p_buf;
```

그림 2-29. 시퀀스 카운터를 초기화 하는 부분 추가

■ 기본 컴퓨터의 명령어에 XCH 명령을 추가했으면 추가된 명령어를 포함하는 프로그램을 작성하고 실행시켜서 동작을 확인해야 한다. 그러나 어셈블러는 XCH 명령어를 어셈블 할 수 없기 때문에 XCH 명령이 들어가는 부분에 원래의 명령어인 STA를 넣고 어셈블을 한다. STA와 XCH는 같은 2진코드를 사용하므로 어셈블된 2진 코드를 그대로 사용한다.

① [그림 2-30]과 같이 프로그램을 작성한다. 프로그램은 0xA번지의 54를 AC에 저장하고 0xB번지에 저장되어 있는 27과 교환한 후 AC에 저장된 27을 출력하는 프로그램이다. 프로그램에서 "STA B"는 추가된 "XCH B"에 해당하는 부분으로 XCH를 어셈블 할 수 없으므로 STA 명령어를 그대로 사용했다.

```
        ORG 100
        LDA A
        STA B
        OUT
        HLT
A,      DEC 54
B,      DEC 27
        END
```

그림 2-30. XCH 테스트 프로그램

② 실습 2.5에서 했던 방법대로 어셈블러를 이용해서 작성된 프로그램을 어셈블 하여 mif 파일을 생성한다. STA와 XCH는 메모리 참조명령이고 2진 코드가 같으므로 생성된 mif 파일을 수정할 필요가 없다.

■ mif 파일을 메모리의 초기화 데이터에 포함하여 컴파일 한 후 다운로드하여 실행한다.

① 생성된 mif 파일을 프로젝트 폴더 DIGCOM-A1.2₩VHDL₩SimpleComputer₩DigiCom-A1.2-NewInst에 복사하고 메모리의 LPM_RAM_IO에서 LPM_FILE을 어셈블된 mif 파일명으로 수정하고 기본컴퓨터를 다시컴파일한다.

② 컴파일된 pof 파일을 DIGCOM-A1.2에 다운로드한다.

③ 실습키트를 실행시켜서 출력장치에 27(0x1B)가 출력되는 것을 확인한다.

컴퓨터 구조 실습 결과보고서[ComLab-6]

일 시		**전 공**		**실험시간**	
학 번			**이 름**		
제 목	새로운 명령어의 추가				
실습목적	기본 컴퓨터는 7 개의 메모리 참조명령, 12 개의 레지스터 명령과 6 개의 입출력 명령을 사용한다. opcode 는 모두 4 비트가 할당되므로 명령어의 개수에 제한이 있다. 따라서 기존의 명령이 충분하지 않기 때문에 때로 다양한 기능을 제공하는 새로운 명령을 추가해야 할 필요가 있다. 본 실습에서는 기존의 명령에 새로운 명령을 추가하고 이를 기존에 이미 구현된 기본컴퓨터의 VHDL 코드를 수정해서 구현함으로써 명령어 구현을 하기 위해 필요한 제어신호를 추가함으로써 컴퓨터의 제어 역할을 이해한다.				
실습 내용	**실습결과**				
추가할 명령어의 마이크로 연산	기본 컴퓨터에 새로 추가할 명령어는 ADM(ADd to Memory)이며 ADD 명령어는 AC와 메모리의 데이터를 더해서 AC에 저장하나, ADM에서는 더한 결과를 메모리에 저장하고 AC의 값은 변하지 않는다. ADM의 동작을 심볼로 표현하면 다음과 같다. M[EA] ← M[EA] + AC ADM은 기존의 명령어 ADD를 대체하며 opcode는 "001"이다. 아래 표에 ADM에 대한 마이크로 연산을 T_5부터 나열하라.				

명령어	마이크로 연산
ADM	$R'T_0$: AR ← PC $R'T_1$: IR ← M[AR], PC ← PC + 1 $R'T_2$: D_0, …,D_7 ← Decode IR(12-14), AR ← IR(0-11), I ← IR(15) D'_7IT_3 : AR ← M[AR] D_1T_4 : DR ← M[AR] D_1T_5 : D_1T_6 :

ADD 제어신호 발생하는 코드 삭제

아래 표의 D_1T_4 사이클에서와 같이 ADD 명령이 실행되기 위해 사용되는 제어신호를 D_1T_5 에 대해서도 나열하라. 그리고 DIGCOM-A1.2₩VHDL₩SimpleComputer₩DigiCom-A1.2-NewInst 폴더의 simplecomA12 프로젝트를 오픈하고 제어부의 VHDL에서 이 신호들을 발생하는 부분을 삭제하거나 주석처리하라.(아래 마이크로연산에서 E ← C_{OUT}은 산술논리 연산부에서 이루어지며, 제어부에는 영향을 미치지 않으므로 수정할 부분이 없다.)

마이크로 연산	제어 신호	VHDL 코드에서 삭제할 부분	
D_1T_4 : DR ← M[AR]	M_enable	(d(1) and M_t(4) and clk2)	-- memory ADD
	M_read	(d(1) and M_t(4) and clk2)	-- memory ADD
	x(7)	(d(1) and t(4))	-- memory ADD
	dr_ld	(d(1) and t(4))	-- memory ADD
D_1T_5 : AC ← AC + DR, E ← C_{OUT}, SC ← 0			

ADM 명령 제어신호 발생하는 코드 추가

ADD에 필요한 제어신호를 삭제했으면 ADM에 필요한 제어신호를 추가해야 한다. 아래 표에 T_4 이후의 마이크로 연산과 각 마이크로 연산에 필요한 제어신호를 나열하고, 이 신호를 발생하기 위해 추가되어야 할 코드를 아래와 같이 표시하라. 아래 표에서 T_4에서는 ADD와 ADM의 마이크로연산이 같기 때문에 삭제한 코드와 추가한 코드가 동일하다.

마이크로 연산	제어 신호	VHDL 코드에서 추가할 부분	
D_1T_4 : DR ← M[AR]	M_enable	(d(1) and M_t(4) and clk2)	-- memory ADM
	M_read	(d(1) and M_t(4) and clk2)	-- memory ADM
	x(7)	(d(1) and t(4))	-- memory ADM
	dr_ld	(d(1) and t(4))	-- memory ADM
D_1T_5 :			
D_1T_6 :			

새로운 명령을 사용하는 프로그램 작성

VHDL 을 수정했으면 아래와 같은 프로그램을 작성한다. 어셈블러가 ADM 명령을 어셈블하지 못하므로 ADM 대신에 ADD 를 사용한다. 아래 프로그램은 0xA 에 저장된 83 과 0xB 에 저장된 -23 을 더해서 0xB 에 저장한 후 다시 AC 로 읽어서 출력하는 프로그램이다. 프로그램을 어셈블한 후 오른쪽에 mif 파일을 보여라.

프로그램		mif 파일
	ORG 100	
	LDA A	
	ADD B	
	LDA B	
	OUT	
	HLT	
A,	DEC 83	
B,	DEC -23	
	END	

테스트 프로그램 실행

생성된 mif 파일을 메모리에 포함시켜 컴파일한 후 DIGCOM-A1.2 실습 키트에 다운로드하여 실행한다. 실행한 결과 출력 7-세그먼트에 어떤 결과가 나와야 하는가? 아래에 실행한 결과 캡쳐 화면을 보여라.

실행한 결과 캡쳐 화면

실습결과 및 토의

3. 기본 컴퓨터의 설계

기본 컴퓨터를 이해하는 가장 좋은 방법은 기본컴퓨터를 설계하는 것이다. 본 장에서는 기본 컴퓨터를 설계하는 과정을 기능별로 설계하는 과정을 설명한다. [그림 3-]은 기본 컴퓨터의 블록도이다. 이 블록도에서 각 블록은 기능별로 구성되어 있으며 VHDL 로 설계한 후 BSF(Block Schematic File)로 생성되고, 상위 계층 설계에서 BDF/Schematic 로 설계된다. 설계된 기본 컴퓨터는 DIGCOM-A1.2 에 어셈블된 프로그램과 함께 다운로드되어 실행된다. 본 장에서 설명되는 모든 소스 코드는 DIGCOM-A1.2₩VHDL₩SimpleComputer₩DigCom-A1.2-NoStep 폴더의 simplecomA12 프로젝트에 포함되어 있다.

각 블록에서 하는 중요한 기능은 다음과 같다.

- RESET 블록 : 전원을 공급할 때 power-on 리셋과 리셋 스위치에 의한 리셋을 제공한다. 출력은 기본 컴퓨터에서 리셋으로 사용한다.
- SC(Sequence Counter) 블록 : 가장 기본적인 기능으로 T 클럭을 발생시킨다. 이 외에도 외부에서 입력되는 1MHz 클럭을 2 분주하여 기본 컴퓨터에서 메인 클럭으로 사용하며, HLT 명령어가 실행되면 카운트 동작을 멈추어 컴퓨터를 정지시킨다.
- CONTROL 블록 : 가장 중요한 부분으로 각 레지스터와 메모리가 데이터를 전송할 때 필요한 제어신호와 ALU 에서 필요한 연산을 제어하는 신호를 발생시킨다.
- ALU 블록 : control 블록에서 연산 제어신호와 레지스터 블록에서 입력 데이터를 받아 필요한 연산을 실행한다. 연산결과는 다시 레지스터 블록의 AC 에 저장된다.
- INPUT/OUTPUT 블록 : 키 버튼으로부터 입력을 받아 인코딩하여 INPR(Input Register)에 저장하고 OUTR(Output Register)의 값을 디코딩 하여 7-세그먼트에 출력한다.
- REGISTER 블록 : 각 레지스터가 선언되어 있고 control 블록에서 제어신호를 받아 다른 레지스터 또는 메모리로부터 데이터를 입력 받는다. 또한 control 블록에서 공통버스를 제어하는 x 신호를 입력 받아 해당하는 레지스터에 저장된 데이터를 공통버스로 출력한다.
- MEMORY 블록 : 메모리는 ALTERA 에서 제공하는 라이브러리 megafunction 을 이용한다. 이 메모리에 프로그램을 입력하여 실행시킬 수 있다.
- REGISTER OUT 블록 : 각 레지스터의 값을 모니터링 할 수 있도록 한다. 모니터링하고자 하는 레지스터를 레지스터 선택 스위치를 이용해서 선택하면 레지스터에 저장된 데이터가 7-세그먼트에 출력된다. 스텝 모드로 실행될 때 레지스터의 중간 결과값을 7-레지스터에서 볼 수 있다.

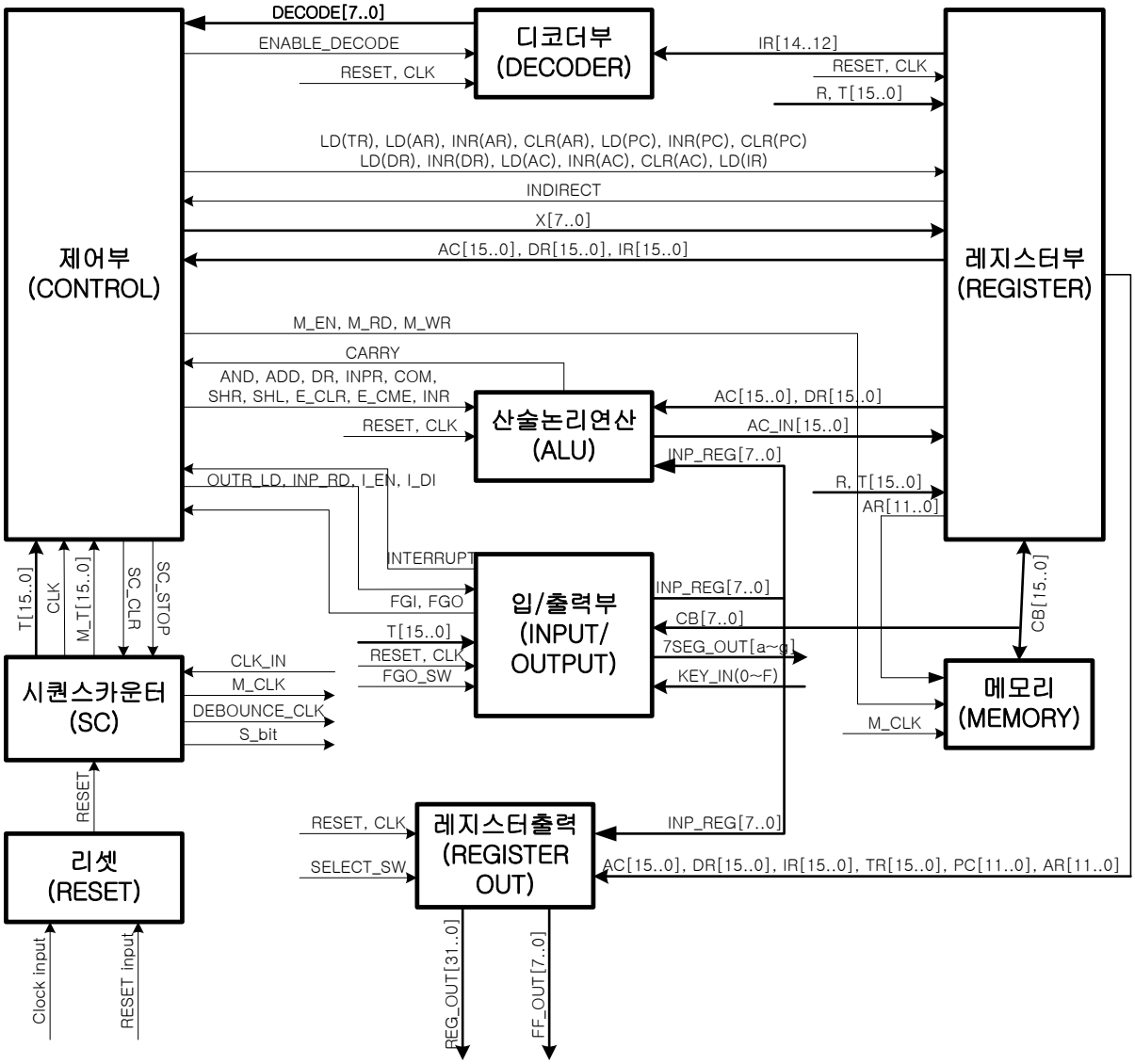


그림 3-1. 기본 컴퓨터의 블록도

3.1. 리셋(reset) 회로

리셋은 전원 스위치에 의한 리셋과 리셋 스위치에 의한 리셋으로 구분된다. 전원이 켜지면 백만번의 시스템 클럭 동안 리셋 상태를 유지하다가 해제가 되며, 리셋 스위치 신호의 값과 'AND'를 하여 다른 회로에 리셋을 제공한다.

코드 3-1

```
BEGIN
PROCESS (clk)
        variable cnt : integer range 0 to 1000000;
```

```
BEGIN
        if rising_edge(clk) then
                if cnt < 1000000 then
                        cnt := cnt + 1;
                        reset_node <= '0';
                elsif cnt >= 1000000 then
                        cnt := cnt + 0;
                        reset_node <= '1';
                end if;
        end if;
END PROCESS;
reset_out <= reset_node and reset_in;
```

3.2. SC(Sequence Counter)부

3.2.1. 클럭의 사용

SC 에는 FPGA 외부에서 공급되는 1MHz 오실레이터 클럭을 2 분주하여 사용한다. 따라서 사용되는 클럭의 속도는 500MHz 이며, 이 클럭의 상승에지(rising edge)에 동기가 되도록 설계하였다.

3.2.2. T 클럭

코드 3-2

```
PROCESS (sc_clk, nRESET)
        VARIABLE        cnt                     : INTEGER RANGE 0 TO 16;
BEGIN
        if nRESET = '0' then
                cnt := 0;

        elsif rising_edge(sc_clk) then
                if sc_clr_delay = '1' then
                        cnt := 1;
                else
                        cnt := cnt + 1;
                end if;
        END IF;

        case cnt is
```

```
                when 0      =>    t_node <= "0000000000000000";
                when 1      =>    t_node <= "0000000000000001";
                when 2      =>    t_node <= "0000000000000010";
                when 3      =>    t_node <= "0000000000000100";
                when 4      =>    t_node <= "0000000000001000";
                when 5      =>    t_node <= "0000000000010000";
                when 6      =>    t_node <= "0000000000100000";
                when 7      =>    t_node <= "0000000001000000";
                when 8      =>    t_node <= "0000000010000000";
                when 9      =>    t_node <= "0000000100000000";
                when 10     =>    t_node <= "0000001000000000";
                when 11     =>    t_node <= "0000010000000000";
                when 12     =>    t_node <= "0000100000000000";
                when 13     =>    t_node <= "0001000000000000";
                when 14     =>    t_node <= "0010000000000000";
                when 15     =>    t_node <= "0100000000000000";
                when 16     =>    t_node <= "1000000000000000";
        end case;

        if cnt = 16 then
                cnt := 0;
        end if;
END PROCESS;
```

T 클럭은 16 비트로 되어 있으며, 0 에서 최대 15 까지 카운트를 할 수 있는 카운터가 하나씩 증가함에 따라 16 비트중의 한 비트가 1 로 출력되어 T_0 ~ T_{15} 까지 신호를 발생한다.

cnt 는 0 ~ 16 사이의 값을 갖는 정수형 변수이다. cnt 는 0 으로 초기화 된 후에 하나의 명령어가 실행되는 동안에는 클럭의 상승 에지에서 하나씩 증가한다. 그러나 한 명령어의 마지막 마이크로 연산이 실행되면서 제어부로부터 sc_clr_delay 신호가 발생되며 이 신호가 인에이블되면 cnt 가 초기화되어 T1 부터 다시 발생한다. 제어부에서는 하나의 명령어를 마칠 때마다 sc_clr 신호를 발생시키나 이 신호가 마지막 하나의 T 클럭동안만 인에이블 상태를 유지하므로 그 다음 클럭의 상승에지에서 디스에이블(disable)되기 때문에 cnt 를 초기화 시키지 못한다. 따라서 클럭의 상승 에지에서 인에이블 상태를 유지하기 위해 오실레이터의 반 클럭 만큼 지연시켜 인에이블 상태를 유지하도록 하였다. cnt 는 최대값이 16 이므로 case 문장을 이용하여 cnt 의 각 값에 해당하는 비트를 T 클럭에 발생시킨다.

3.2.3. 메모리 클럭

기본 컴퓨터에서는 메모리가 동기식 입력을 가지며 메모리에서 사용될 클럭을 SC 부에서 발생시킨다. 또한 하나의 T 사이클 동안 메모리 읽기 또는 쓰기 동작이 이루어지도록 메모리 제어신호가 만들어져야 하며, 이를 위해서 SC 부에서 T 클럭과 다른 위상을 갖는 메모리 클럭을 발생시킨다.

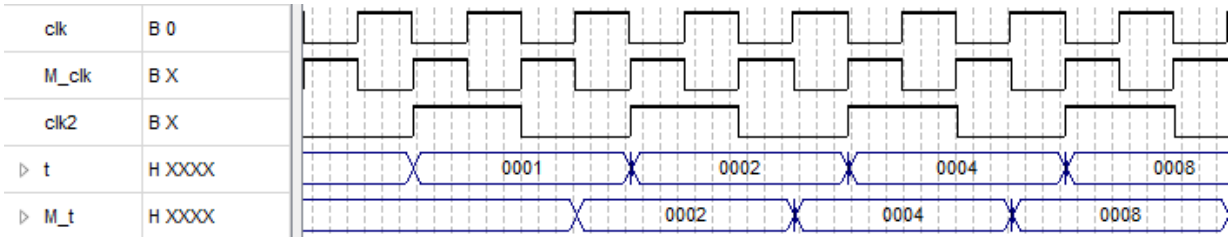


그림 3-2. 메모리 클럭

[그림 3-2]에서 메모리 클럭(M_clk)은 주 클럭(clk)과 반대 위상을 가지며, T 클럭은 2 분주된 clk2 의 상승에지에서 한 클럭동안 유지되고, 메모리 클럭은 T 클럭보다 1/2clk 빠르다. T 클럭과 메모리 클럭(M_t)은 16 비트를 16 진수로 표시한 것이므로, 0x0001 은 M_t(0)가 출력된 것이고, 0x0002 는 M_t(1)이 출력된 것이다.

아래는 메모리 클럭을 발생시키는 부분이다.

코드 3-3

```
process(clk2, t_node, clk)
begin
        if clk2 = '0' and rising_edge(clk) then
                M_t_node(15 downto 1) <= t_node(14 downto 0);
                M_t_node(0) <= '0';
        end if;
end process;
```

메모리 T 클럭은 clk 의 상승에지에서 T 클럭 값이 복사된다. 메모리 클럭과 T 클럭을 이용해서 메모리에서 읽기/쓰기는 동작은 제어부와 메모리부에서 설명된다. 발생된 M_clk 은 메모리의 입력클럭으로 사용되며, M_t 는 제어부에 입력되어 메모리 제어신호를 발생하는데 사용된다.

3.3. 제어부

기본 컴퓨터에서 제어부는 가장 중요한 기능을 제공하며, 기본 컴퓨터가 실행되는데 필요한 모든 제어신호를 발생 시킨다. 제어부에서는 SC 로부터 T 클럭, AC 레지스터 값, DR 레지스터 값, 디코드 된 결과, 연산 결과 및 플립플롭 값 등을 입력 받아 다음과 같은 신호를 발생하기 위한 동작을 한다.

- HLT 명령어에 의해 시퀀스 카운터를 정지 시키는 신호를 발생
- 플립플롭을 세트 또는 클리어 시키는 신호를 발생
- 레지스터의 제어신호를 발생
- 양방향 공통버스에서 레지스터를 선택하기 위한 신호를 발생
- AC 에 대해 가산 논리회로를 제어하기 위한 신호를 발생
- 메모리에서 읽기 쓰기를 제어하기 위한 신호를 발생

3.3.1. HLT 명령의 실행

HLT 명령이 실행되기 위해서 시퀀스 카운터에서 클럭 발생을 정지 시킨다. 시퀀스 카운터를 정지시키기 위해서 sc_stop 신호를 발생시키며, 이 신호는 SC 부로 전달되어 시퀀스 카운터에서 T 클럭을 중지 시킨다.

코드 3-4

```
process(d(7), i, t(3), ir(0))
begin
        sc_stop <= d(7) and not i and t(3) and ir(0);     -- register reference HLT
end process;
```

VHDL 코드 3-4 는 sc_stop 신호를 발생시키는 부분이다. HLT 명령어는 레지스터 참조명령으로 제어부에서는 명령어의 opcode 를 디코드한 결과가 D_7(d(7)='1')이고, 간접비트가 '0'(i='0')이며, 명령어 레지스터의 가장 마지막 비트가 '1'(ir(0)='1') 일 때 T_3(t(3)='1') 사이클에서 sc_stop 신호를 발생시킨다.

3.3.2. 플립플롭 제어신호 발생

기본 컴퓨터에는 모두 7 개의 플립플립이 있다. 이 플립플롭은 현재 기본 컴퓨터의 동작 상태를 저장하거나 프로그램에 의해 제어되는 상태를 저장한다. 플립플롭은 세트, 클리어 또는 보수를 취함으로써 제어를 하며, CLE(Clear E), CME(Complement E), HLT(Halt computer), ION(Interrupt on) 및 IOF(Interrupt off) 명령어와 같이 명령어의 실행에 직접 플립플롭을 제어하거나, 명령어의 실행과정에서 마이크로 연산과정에 의해 제어될 수 있다.

제어부에서는 명령어에 의해 플립플롭을 제어하는 신호를 발생시킨다. 플립플롭를 제어하는 명령어는 레지스터 명령과 입출력 명령에 포함되며 이 들 명령과 이를 실행하는 마이크로 연산은 [그림 3-3]과 같다.

```
D7I'T3 = r(Common to all register-reference instructions)
CLE     rB10    :    E ← 0
CME     rB8     :    E ← /E
```

D_7IT_3 = p(Common to all input-output instructions)			
ION	pB_7	:	IEN ← 1
IOF	pB_6	:	IEN ← 0

그림 3-3. 플립플롭을 제어하는 마이크로 연산들

E 플립플롭을 제어하는 명령은 E 플립플롭을 클리어 시키거나 또는 보수를 취하는 마이크로 연산을 나타낸 것이다. r 은 명령어가 레지스터 참조명령이고 T_3 사이클임을 알리는 신호($r=D_7I'T_3$)이고, B_{10} 과 B_9 는 각각 명령어 레지스터의 10 번 비트와 9 번 비트가 세트되어 CLE 명령과 CME 명령임을 알려준다. IEN 플립플롭을 제어하는 명령은 IEN 플립플롭을 세트하거나 클리어 시키는 마이크로 연산을 나타낸다. p 는 명령어가 입출력 명령이고 T_3 사이클임을 알리는 신호($p=D_7IT_3$)이고, B_7 과 B_6 는 각각 명령어 레지스터의 7 번 비트와 6 번 비트가 세트되어 ION 명령과 IOF 명령임을 알려준다.

VHDL 에서 위 마이크로 연산을 실행하는 제어신호를 발생하기 위한 부분은 다음과 같다.

코드 3-5

```
p_buf <= d(7) and i and t(3);
r_buf <= d(7) and (not i) and t(3);
ien_clr <= (int_r and t(2)) or (p_buf and ir(6));
ien_set <= (p_buf and ir(7));
e_clr <= r_buf and ir(10);                    -- register CLE
e_cme <= r_buf and ir(8);                     -- register CME
```

VHDL 코드 3-5 에서 IEN 플립플롭이 클리어 되는 경우는 IOF 명령어에 의한 경우 외에 입출력 장치에 의해 인터럽트 처리 루틴이 실행되기 전에 인터럽트 사이클에서 IEN 을 클리어 시킴으로써 입출력 인터럽트 처리 중에는 다른 인터럽트가 발생하지 못하도록 한다.

3.3.3. 레지스터 제어신호 발생

기본 컴퓨터에는 모두 8 개의 레지스터가 있다. 이 레지스터를 제어하기 위한 동작은 로드(load), 인크리먼트(increment) 및 클리어(clear) 등이 있다. AR 레지스터의 제어신호를 발생시키기 위해서는 기본 컴퓨터 명령어의 마이크로 연산 테이블에서 AR 의 내용을 변경시키는 모든 마이크로 연산을 찾아야 한다.

$R'T_0$	:	AR ← PC
$R'T_2$	:	AR ← IR(0-11)
$D_7'IT_3$	:	AR ← M[AR]
RT_0	:	AR ← 0
D_5T_4	:	AR ← AR + 1

그림 3-4. AR 레지스터의 값을 변경하는 마이크로 연산들

처음 3 개의 마이크로 연산은 다른 레지스터의 값을 AR 로 전송하는 것이고, 나머지 두 연산은 각각 AR 을 클리어하고 값을 하나 증가시키는 동작이다.

위 마이크로 연산을 VHDL 로 구현하면 다음과 같다.

코드 3-6

```
ar_ld <= (not int_R and t(2))
    or (not int_R and t(0))
    or ((not d(7)) and i and t(3));          -- indirect
ar_clr <= int_R and t(0);                    -- interrupt cycle
ar_inr <= (d(5) and t(4));                   -- memory BSA
```

이와 같이 나머지 7 개 레지스터에 대한 제어신호도 VHDL 로 구현할 수 있다. 제어부에서 발생되는 레지스터 제어신호는 레지스터부의 각 레지스터 제어입력으로 전달되어 레지스터를 제어한다.

3.3.4. 레지스터 선택 신호 발생

기본 컴퓨터는 레지스터간 또는 메모리에서 읽거나 쓸 때 공통버스를 사용하므로 공통버스를 사용할 레지스터를 선택하는 신호가 필요하다. 따라서 입출력 레지스터를 제외한 6 개의 레지스터와 1 개의 메모리를 선택하기 위해서 7 개의 선택신호 x(1) ~ x(7)가 필요하다. 7 개 신호와 이 신호가 발생될 때 선택되는 레지스터는 다음과 같다.

표 3-1. 레지스터 선택 신호

X(1)	X(2)	X(3)	X(4)	X(5)	X(6)	X(7)
AR	PC	DR	AC	IR	TR	Memory

[표 3-1]에서 X(7) 신호는 메모리를 선택하기 위한 신호이며, 명령어 마이크로 연산 테이블에서 register ← M[AR] 과 같은 형태의 마이크로 연산은 다음과 같다.

```
R'T1      :   IR ← M[AR]
D7'IT3    :   AR ← M[AR]
D0T4      :   DR ← M[AR]
D1T4      :   DR ← M[AR]
D2T4      :   DR ← M[AR]
D6T4      :   DR ← M[AR]
```

그림 3-5. 메모리 데이터를 전송하기 위한 마이크로 연산들

위 마이크로 연산을 VHDL 로 구현하면 다음과 같다. 생성된 레지스터 선택신호는 레지스터부로 전달되어 레지스터 값을 읽을 때 해당 레지스터를 선택하는데 사용된다.

코드 3-7

```
x(7) <= (not int_R and t(1))              -- Fetch, IR <- M[AR]
    or (not d(7) and i and t(3))      -- indirect, AR <- M[AR]
    or (d(0) and t(4))                -- memory AND, DR <- M[AR]
    or (d(1) and t(4))                -- memory ADD, DR <- M[AR]
    or (d(2) and t(4))                -- memory LDA, DR <- M[AR]
    or (d(6) and t(4));               -- memory ISZ, DR <- M[AR]
```

3.3.5. 가산 논리 제어신호

산술논리 제어신호는 AC(Accumulator)에서 사용된다. AC 의 동작은 모두 8 가지이며 AC 의 동작을 결정하는 레지스터 전송문을 명령어 마이크로 연산 표에서 찾으면 [그림 3-6]과 같다.

D_0T_5 : AC ← AC ∧ DR
D_1T_5 : AC ← AC + DR
D_2T_5 : AC ← AC ← DR
pB_{11} : AC(0-7) ← INPR
rB_9 : AC ← /AC
rB_7 : AC ← shr AC, AC(15) ← E
rB_6 : AC ← shl AC, AC(0) ← E

그림 3-6. AC 레지스터의 값을 변경하는 마이크로 연산들

위와 같은 레지스터 전송문에 따라 VHDL 를 구현하며, ALU 에서 산술 또는 논리연산을 하도록 아래와 같이 제어신호를 발생한다. 따라서 아래 연산의 결과가 AC 로 전달되어 저장된다.

코드 3-8

```
c_and <= d(0) and t(5);                -- AND micro operation
c_add <= d(1) and t(5);                -- ADD
c_dr  <= d(2) and t(5);                -- LDR
c_inpr <= p_buf and ir(11);            -- INP
c_com <= r_buf and ir(9);              -- CMA
c_shr <= r_buf and  ir(7);             -- SHR
c_shl <= r_buf and  ir(6);             -- SHL
```

3.3.6. 메모리 제어신호

메모리 제어신호도 다른 제어신호와 마찬가지로 마이크로 연산 테이블에서 메모리 관련 레지스터 전송문에 의해서 발생된다. 읽기 신호는 register ← M[AR]와 같은 레지스터 전송문에서 발생되고, 쓰기 신호는 M[AR] ← register 와 같은 경우에 발생한다. M_enable 신호는 두 경우 모두 발생한다. 그러나 다른 레지스터 제어신호는 하나의 T 클럭동안에 발생하지만 메모리 제어신호는 메모리 T 클럭과 clk2 을 사용한다. 다음 VHDL 은 메모리 제어신호를 발생시키는 부분이다.

코드 3-9

```
process(M_t, int_R, d, i, clk2)
begin
M_write <= (int_R and M_t(1) and clk2)          -- interrupt cycle
        or (d(3) and M_t(4) and clk2)           -- memory STA
        or (d(5) and M_t(4) and clk2)           -- memory BSA
        or (d(6) and M_t(6) and clk2);          -- memory ISZ

M_enable <= (M_t(1) and clk2)                   -- interrupt cycle
        or (not int_R and M_t(1) and clk2)      -- Fetch cycle
        or (d(3) and M_t(4) and clk2)           -- memory STA
        or (d(5) and M_t(4)and clk2)            -- memory BSA
        or (d(6) and M_t(6) and clk2)           -- memory ISZ
        or (not d(7) and i and M_t(3) and clk2) -- decode
        or (d(0) and M_t(4) and clk2)           -- memory AND
        or (d(1) and M_t(4) and clk2)           -- memory ADD
        or (d(2) and M_t(4) and clk2)           -- memory LDA
        or (d(6) and M_t(4) and clk2);          -- memory ISZ

M_read <= (not int_R and M_t(1) and clk2)       -- Fetch cycle
        or (not d(7) and i and M_t(3) and clk2) -- decode
        or (d(0) and M_t(4) and clk2)           -- memory AND
        or (d(1) and M_t(4) and clk2)           -- memory ADD
        or (d(2) and M_t(4) and clk2)           -- memory LDA
```

```
                or (d(6) and M_t(4) and clk2);          -- memory ISZ
        end process;
```

3.4. 명령어 디코더 부

명령어 디코더는 메모리 참조명령에 대하여 명령어 레지스터(IR)의 12~14 번 비트를 디코드해서 명령어의 종류를 알아낸다.

코드 3-10

```
    case ir_sig is
        when "000" => d <= "00000001";      -- 0
        when "001" => d <= "00000010";      -- 1
        when "010" => d <= "00000100";      -- 2
        when "011" => d <= "00001000";      -- 3
        when "100" => d <= "00010000";      -- 4
        when "101" => d <= "00100000";      -- 5
        when "110" => d <= "01000000";      -- 6
        when "111" => d <= "10000000";      -- 7
    end case;
```

코드 3-10 에서 7 개 메모리 참조명령의 종류에 따라서 디코드된 결과가 d(0)에서 d(6)로 각각 출력되고 d(7)은 레지스터 또는 입출력 참조명령일 때 출력된다. 디코드 결과는 제어부로 전달되어 마이크로 연산에 필요한 제어신호 발생하는데 사용된다.

3.5. 산술연산부

3.5.1. E 플립플롭 값

E 플립플롭은 연산결과를 반영하거나 명령어에서 직접 값을 변경한다. 더하기 명령의 결과가 16 비트를 초과하면 E 플립플롭이 '1'로 세트되고, 시프트 동작에서 AC 의 가장 마지막 또는 처음 비트값이 E 플립플롭으로 이동된다. 또한 CLE(Clear E) 또는 CME(Complement E)와 같이 직접 값을 변경하기도 한다.

코드 3-11

```
    case k is
        when "000100000" =>         -- add
                e <= conv_std_logic_vector(conv_integer(ac) + conv_integer(dr), 17)(16);
```

```
        when "000001000" =>          -- shr
                e <= ac(0);
        when "000000100" =>          -- shl
                e <= ac(15);
        when "000000010" =>          -- cle
                e <= '0';
        when "000000001" =>          -- cme
                e <= not e2;
        when others => null;
    end case;
```

VHDL 에서 vector k 는 다음과 같이 정의된다.

```
k <= c_dr & c_com & c_and & c_add & c_inpr & c_shr & c_shl & e_clr & e_cme;
```

따라서 E 플립플롭이 변경될 수 있는 명령어인 ADD, SHR, SHL, CLE, CME 에서 case ~ when 을 이용하여 구현하였으며, ADD 명령에서는 AC 와 DR 을 정수형으로 더한 결과에서 17 번째 비트를 E 플립플롭으로 저장한다. CME 명령의 경우에 자신의 값에 보수를 취하는 것이므로 항상 E 와 같은 값을 갖는 복사본을 유지하는 것이 필요하다. 일반적으로 마이크로 연산의 레지스터 전송은 클럭의 하강에지에서 발생하도록 하였으나 E 의 복사본을 유지하기 위해서 1/2 클럭 뒤에 상승에지에서 아래 VHDL 와 같이 E 를 복사한다.

코드 3-12

```
    elsif rising_edge(clk) then      -- register instance
        e2 <= e;
    end if;
```

3.5.2. 산술논리연산

산술논리 연산된 결과는 AC 에 저장되기 위하여 레지스터부로 전달된다. 이 부분에서도 연산의 종류를 알기 위하여 위에서 정의된 vector k 를 이용한다.

코드 3-13

```
    case k is
        when "100000000" =>                  -- c_dr
                ac_out <= dr;
        when "010000000" =>                  -- c_com
                ac_out <= not ac;
        when "001000000" =>                  -- c_and
```

```
            ac_out <= ac and dr;
        when "000100000" =>                         -- c_add
            ac_out <= conv_std_logic_vector(conv_integer(ac) +
                        conv_integer(dr), 17)(15 downto 0);
        when "000010000" =>                         -- c_inpr
            ac_out(7 downto 0) <= inpr;
            ac_out(15 downto 8) <= "00000000";
        when "000001000" => --shr
            ac_out(14 downto 0) <= ac(15 downto 1);
            ac_out(15) <= e2;
        when "000000100" => --shl
            ac_out(15 downto 1) <= ac(14 downto 0);
            ac_out(0) <= e2;
        when others =>
            ac_out <= "0000000000000000";
    end case;
```

3.6. 레지스터 부

3.6.1. 레지스터 값 변경

레지스터은 새로운 값을 입력받아 저장하거나, 하나 증가하거나 또는 클리어되는 경우에 값이 변경된다. 제어부에서 전달받은 제어신호에 의해 세가지 중의 한 동작을 실행한다. 아래 VHDL 은 AR ← PC 레지스터 전달을 구현한 예의 일부분이다.

코드 3-14

```
    when "0000010" =>                               -- PC output
        cb_node(11 downto 0) <= pc;
        cb_node(15 downto 12) <= "0000";
        cb <= cb_node;
```

```
    if ar_ld ='1' then
        ar <= cb(11 downto 0);
    elsif ar_inr = '1' then
        ar <= ar + "000000000001";
    elsif ar_clr = '1' then
        ar <= "000000000000";
    end if;
```

VHDL 코드 3-14 에서 첫째 부분은 PC 가 공통버스로 전달되는 과정을 나타낸다. 레지스터 선택신호 x(="0000010")가 PC 를 선택하면 PC 에 저장된 값이 공통버스를 나타내는 CB(Common Bus)로 전달된다. 그리고 동시에 AR Load 신호가 제어부로부터 입력되어 공통버스에 있는 값이 AR 에 저장된다. 레지스터 블록에서 공통버스는 signal 로 구현되며, 메모리는 다른 블록으로 구현되므로 메모리에서 읽기/쓰기 동작을 위해서는 양방향 버스를 통해서 데이터를 주고 받아야 한다.

연산결과에 의해 AC 가 변경되는 경우에는 AC 가 연산의 입력으로 사용되는 동시에 결과를 저장한다. 따라서 AC 의 복사 값을 유지하는 것이 필요하다. AC 가 입력으로 사용될 때는 AC 의 복사 값을 이용하고 AC 가 변경된 후에는 복사도 함께 변경한다.

코드 3-15

```
if nRESET = '0' then
    ac2 <= "0000000000000000";
elsif rising_edge(clk) then              -- register instance
    if ac_ld ='1' or ac_inr = '1' or ac_clr = '1' then
            ac2 <= ac;
    end if;
end if;
```

3.6.2. 메모리 읽기 동작

메모리에서 데이터를 주고 받기 위해서는 양방향 버스를 제어해야 한다. 다음 VHDL 은 메모리에서 데이터를 읽는 동작인 DR ←M[AR]를 구현한 부분이다.

코드 3-16

```
when "1000000" =>                                   -- Memory input
    cb <= "ZZZZZZZZZZZZZZZZ";
    cb_node <= cb;
```

위 부분은 공통버스에서 메모리 데이터를 읽는 부분이다. 외부로 나가는 공통버스(cb)는 양방향이므로 출력은 3-상태(tri-state)를 출력하고 입력을 받는다. 이 때 제어부에서는 메모리 인에이블(M_en) 신호와 메모리 읽기(M_read) 신호를 출력하므로 메모리에 저장된 데이터가 공통버스를 통해 입력된다. 아래 부분은 공통버스에서 입력되는 데이터를 DR 에 저장하는 부분으로 dr_ld 신호일 때 하강 에지(falling edge)에서 저장된다.

코드 3-17

```
if dr_ld = '1' then
    dr <= cb (15 downto 0);
elsif dr_inr = '1' then
```

```
        dr <= dr + "0000000000000001";
    end if;
```

3.7. 입출력 부

3.7.1. 키 스위치 입력부

입력을 할 때는 0x0 ~ 0xF 까지 값을 갖는 입력 키 스위치를 누르면 INPR(Input Register)에 값이 저장됨과 동시에 FGI 플립플롭이 '1'로 세트되어 INP 명령으로 INPR 에 저장된 값을 AC 로 전달하게 함과 동시에 FGI 를 '0'으로 리셋한다. [그림 3-7]은 FGI 와 입력동작을 보여준다.

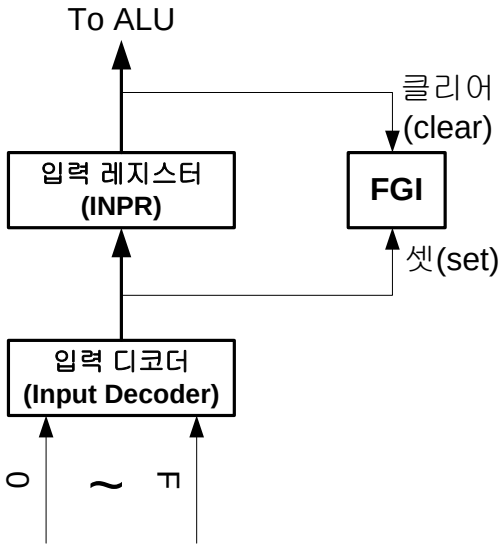


그림 3-7. FGI 플립플롭과 입력 동작

16 개 푸시버튼 스위치가 입력 스위치로 사용된다. 각 키는 0x0 ~ 0xF 의 키 값을 갖는다. 그러나 외부 푸시버튼 스위치에서 값을 입력할 때는 기계적인 스위치에 의해 발생하는 채터링을 제거해야 한다. 물론 스위치 주변에 디바운스(Debounce) 회로를 추가하였지만 VHDL 코드에서도 채터링을 제거하였다.

코드 3-18

```
    process(nRESET, clk20000)
    begin
        if nRESET = '0' then
                keyIn_node(15 downto 0) <= "0000000000000000";
        elsif rising_edge(clk20000) then
                keyIn_node(15 downto 0) <= keyIn(15 downto 0);
        end if;
    end process;
```

```
keyNode <= keyIn_node;

key_clk <=      keyIn_node(0)  or keyIn_node(1)  or keyIn_node(2)
                or keyIn_node(3)  or keyIn_node(4) or keyIn_node(5)
                or keyIn_node(6)  or keyIn_node(7)  or keyIn_node(8)
                or keyIn_node(9)  or keyIn_node(10) or keyIn_node(11)
                or keyIn_node(12) or keyIn_node(13) or keyIn_node(14)
                or keyIn_node(15);
```

VHDL 에서는 클럭을 20000 분주하여 이 클럭(clk20000)의 상승에지에서 푸시버튼 스위치를 입력하여 signal vector(keyIn_node)에 저장한다. 이 때 푸시버튼 스위치를 누르지 않은 상태에서는 '1'이 입력되고 누르면 '0'이 입력되므로 반대논리 값을 저장한다. 그리고 입력된 모든 키 값을 'OR'를 취함으로서 키 값의 변화가 있을 때 이 신호(key_clk)를 키 입력을 위한 클럭으로 사용한다.

코드 3-19

```
elsif rising_edge(key_clk) then
    case keyIn(15 downto 0) is
            when "0000000000000001" => input <= "00000000"; fgi <= '1';
            when "0000000000000010" => input <= "00000001"; fgi <= '1';
            when "0000000000000100" => input <= "00000010"; fgi <= '1';
            when "0000000000001000" => input <= "00000011"; fgi <= '1';
            when "0000000000010000" => input <= "00000100"; fgi <= '1';
            when "0000000000100000" => input <= "00000101"; fgi <= '1';
            when "0000000001000000" => input <= "00000110"; fgi <= '1';
            when "0000000010000000" => input <= "00000111"; fgi <= '1';
            when "0000000100000000" => input <= "00001000"; fgi <= '1';
            when "0000001000000000" => input <= "00001001"; fgi <= '1';
            when "0000010000000000" => input <= "00001010"; fgi <= '1';
            when "0000100000000000" => input <= "00001011"; fgi <= '1';
            when "0001000000000000" => input <= "00001100"; fgi <= '1';
            when "0010000000000000" => input <= "00001101"; fgi <= '1';
            when "0100000000000000" => input <= "00001110"; fgi <= '1';
            when "1000000000000000" => input <= "00001111"; fgi <= '1';
            when others => null;
    end case;
end if;
```

key_clk 의 상승에지에서 signal 에 입력된 키 값에 따라 INPR 에 0x0 ~ 0xF 값을 저장한다. 그리고 동시에 FGI 를 세트 함으로써 현재 INPR 에 유효한 데이터가 입력되어 있음을 나타낸다.

3.7.2. FGO

출력의 경우에는 FGO 를 검사해서 '1'이면 출력할 데이터를 OUTR(Output Register)에 전달함과 동시에 FGO 를 '0'으로 리셋한다. OUTR 에 저장된 값은 7-세그먼트에 출력이 되며 새로운 데이터를 출력하기 위해서는 외부에서 FGO 를 '1'로 세트시켜야 한다. FGO_SET 스위치를 누르면 FGO 플립플롭이 '1'로 세트되어 다음 데이터를 출력할 수 있다. [그림 3-8]은 FGO 에 의해 데이터를 출력시키는 동작을 보여준다.

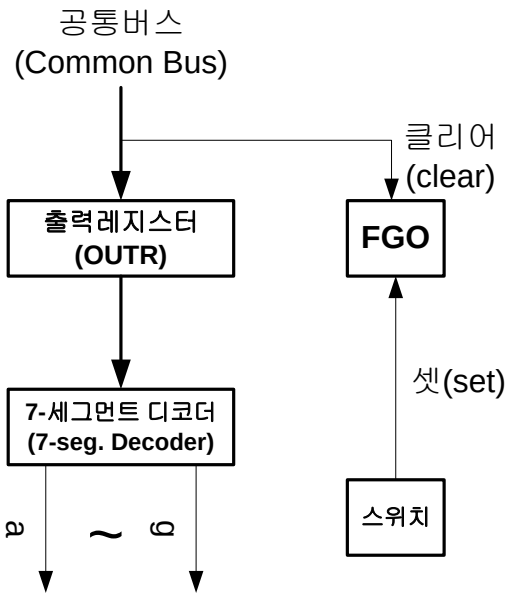


그림 3-8. FGO 플립플롭과 출력

연속되는 출력이 있을 때에는 한 데이터를 출력한 후에 다시 FGO 가 '1'로 세트될 때까지 기다리며, 외부에서 '1'로 세트가 되면 다음 데이터를 OUTR 에 전송을 해서 출력을 한다. 그러나 기본 컴퓨터에서 FGO 를 검사해서 데이터를 출력하는 시간이 푸시버튼 스위치를 눌렀다가 떼는 시간보다 매우 짧기 때문에 스위치를 누른 상태에서 FGO 가 계속해서 '1'인 것으로 간주해서 7-세그먼트의 출력 데이터를 확인할 시간도 없이 다음 데이터를 연속해서 출력한다.

이를 방지하기 위해서 FGO 스위치가 한 번 눌릴 때마다 한 개의 클럭이 발생하게 하고 이 클럭을 이용하여 FGO 스위치가 한 번 눌릴 때마다 한 개의 데이터만이 출력될 수 있게 설계하였다. 코드 3-20 은 FGO 스위치가 한번 눌릴 때 한 개의 클럭을 발생하게 하는 부분이다.

코드 3-20

```
process(nRESET, clk20000, fgo_sw)
```

```
begin
    if nRESET = '0' then
            fgo_set <= '0';
    elsif rising_edge(clk20000) then
            fgo_set <= fgo_sw;
    end if;
end process;

-- generating fgo clock with duration of clk
process(nRESET, clk, fgo_set)
begin
    if nRESET = '0' then
            state_fgo <= s0;
            fgo_clk <= '1';
    elsif rising_edge(clk) then
            case state_fgo is
                    when s0 =>
                            if fgo_set = '1' then
                                    state_fgo <= s0;
                                    fgo_clk <= '1';
                            else
                                    state_fgo <= s1;
                                    fgo_clk <= '0';
                            end if;

                    when s1 =>
                            if fgo_set = '1' then
                                    state_fgo <= s0;
                                    fgo_clk <= '1';
                            else
                                    state_fgo <= s1;
                                    fgo_clk <= '1';
                            end if;
            end case;
    end if;
end process;
```

위에서 처음 process 문에서는 디바운스를 위해 20000 분주한 클럭(clk20000)에 동기시켜 스위치 값을 signal fgo_set 에 저장한다. 그 다음에는 state machine 을 이용하여 스위치가 한번 눌렸을 때 하나의 클럭이 발생되도록 한다. 클럭(fgo_clk)의 초기 상태는 '1'이며 스위치가 한번 눌려지면 한 시스템 클럭동안 '0' 상태를 유지했다가 다시 '1'로 되돌아 간다.

VHDL 코드 3-20 에서 FGO 스위치가 눌려지면 clk20000 의 상승에지에 동기 되어 fgo_set signal(fgo_set)이 '1'로 세트된다. 다음 state machine 에 의해 클럭(clk)의 상승에지에 동기 되어 FGO 클럭(fgo_clk)이 '1'에서 '0'으로 되면서 S1 상태로 천이한다. 그리고 다음 클럭에서 fgo_set 의 값과 상관없이(FGO 스위치를 누른 상태이던 누르지 않은 상태이던 상관없이) 다시 '1'로 되어 1 개의 fgo_clk 이 발생함으로써 1 개의 FGO 클럭이 발생한다. 이 때 스위치를 누른 상태이면 S1 상태를 계속 유지하고 누르지 않은 상태이면 S0 상태로 천이를 한다.

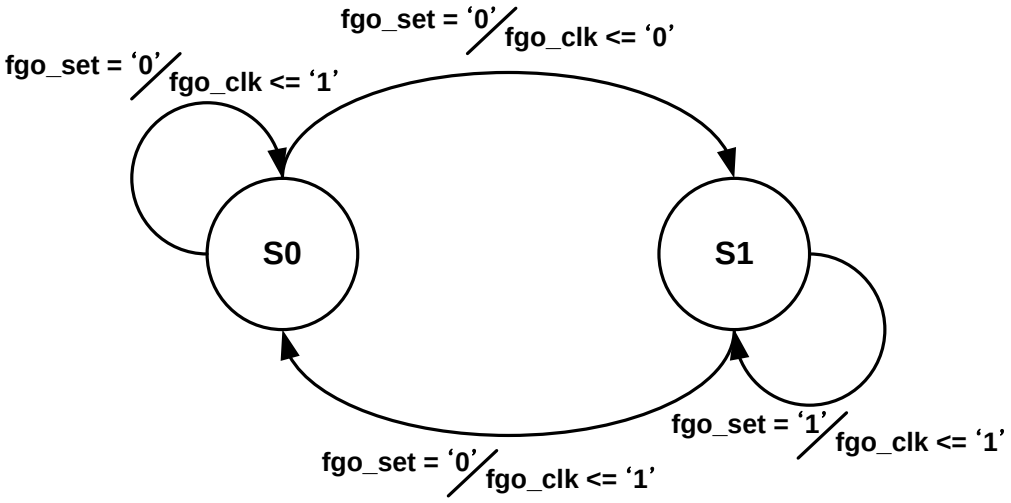


그림 3-9. FGO 플립플롭의 제어신호 상태도

위에서 발생시킨 한 개의 FGO 클럭은 OUTR 의 load 신호와 함께 FGO 플립플롭을 제어하는데 사용된다. 코드 3-21 은 FGO 플립플롭을 제어하는 부분이다.

코드 3-21

```
PROCESS (clk, nRESET, fgo, fgo_clk, out_ld, output)
        variable k                          : std_logic_vector(1 downto 0);
BEGIN
        k := out_ld & fgo_clk;
        if nRESET = '0' then
                output <= "00000000";
                fgo <= '1';
        elsif falling_edge(clk) then
```

```
            if out_ld = '1' and fgo = '1' then
                output <= cb(7 downto 0);
            end if;
            case k is
                when "00" => fgo <= '1';        -- fgo switch is pressed
                when "01" => fgo <= NULL; -- nothing happend
                when "10" => fgo <= NULL; --out_ld is enabled and fgo switch is pressed
                when "11" => fgo <= '0';        -- out_ld is enabled
                when others => NULL;
            end case;
        end if;
```

코드 3-21 에서 FGO 의 초기 값은 '1'이므로 값을 출력할 수 있다. OUTR 의 ld 신호(out_ld)가 '1'이고 FGO 가 '1'이면 데이터가 출력된다. 다음 case 문에 의해 out_ld 와 fgo_clk 이 "00"이면 FGO 스위치가 눌렸다는 것을 의미하므로 다음 데이터 출력을 위해 FGO 플립플롭을 '1'로 세트하고, "01"이면 출력과 관련된 아무 일도 발생하지 않은 경우이며, "10"이면 출력신호 out_ld 가 인에이블되고(out_ld ='1'), 스위치가 눌려진 상태이므로 FGO 플립플롭은 아무 동작도 하지 않는다. 마지막으로 "11"인 경우에는 출력신호만 인에이블 된 상태이므로 FGO 플립플롭을 '0'으로 리셋한다.

3.7.3. 7- 세그먼트 출력

7- 세그먼트 는 OUTR 에 직접 연결되어 있어 레지스터 데이터를 출력한다. 레지스터는 8 비트 크기이므로 2 개의 7- 세그먼트 가 사용되며, 각 7-세그먼트 에는 0x0 ~0xF 의 값을 출력할 수 있다.

3.7.4. 인터럽트 플립플롭, 인터럽트 인에이블 플립플롭

인터럽트와 관련된 플립플롭은 IEN(Interrupt Enable)과 R 플립플롭 2 개가 있다. IEN 은 명령어에 의해 세트되거나 클리어되며, IEN 이 세트되었을 때만 인터럽트가 발생할 수 있다. R 플립플롭은 인터럽트가 발생한 상태를 저장하며, R 이 '0'이면 명령어 사이클을 실행한다. 명령어 사이클에서 IEN 을 검사하고 모두 '0'이면 입력이나 OUTR 에 전송할 데이터가 없는 것이므로 다음 명령어 사이클을 계속하고, IEN 이 '1'이고 FGI 또는 FGO 두 플래그 중의 하나라도 '1'이면 R 이 '1'로 세트된다. R 값은 명령어 실행의 마지막 단계에서 검사되어 '1'이면 인터럽트 사이클을 실행한다.

코드 3-22

```
-- process for ien flip-flop
    PROCESS (clk2, nRESET, ien_set, ien_clr)
    BEGIN
        if nRESET = '0' then
```

```
                ien <= '0';
            elsif falling_edge(clk2) then
                if ien_set = '1' then
                    ien <= '1';
                elsif ien_clr = '1' then
                    ien <= '0';
                end if;
            end if;
        end process;

-- process for interrupt flip-flop(R flip-flop)
        process(clk2, nRESET, t, ien, fgi, fgo)
        begin
            if nRESET = '0' then
                r <= '0';
            elsif falling_edge(clk2) then
                if (not t(0) and not t(1) and not t(2) and ien and (fgi or fgo)) = '1' then
                    r <= '1';
                end if;
                if (r and t(2)) = '1' then
                    r <= '0';
                end if;
            end if;
        end process;
```

IEN 은 명령어에 의해 제어되므로 제어부에서 발생하는 ien_set 또는 ien_clr 신호에 의해 세트 또는 클리어 된다. R 은 T_2 클럭이후에 IEN 이 세트되어 있고, FGI 또는 FGO 가 '1'이면 '1'로 세트된다. 그리고 인터럽트 사이클의 T_2 클럭에서 클리어된다.

3.8. 메모리

메모리는 Quartus Prime 라이브러리에서 제공하는 megafunction 을 이용한다. 다음은 메모리를 생성하는 과정을 설명한다.

3.8.1. 메모리의 생성

다음은 Quartus Prime 에서 메모리를 생성하는 과정을 설명한다.

■ 외쪽 마우스 버튼을 두번 클릭하고, "Library"에서 megafunctions ➔ storage ➔ lpm_ram_io 를

선택한다.

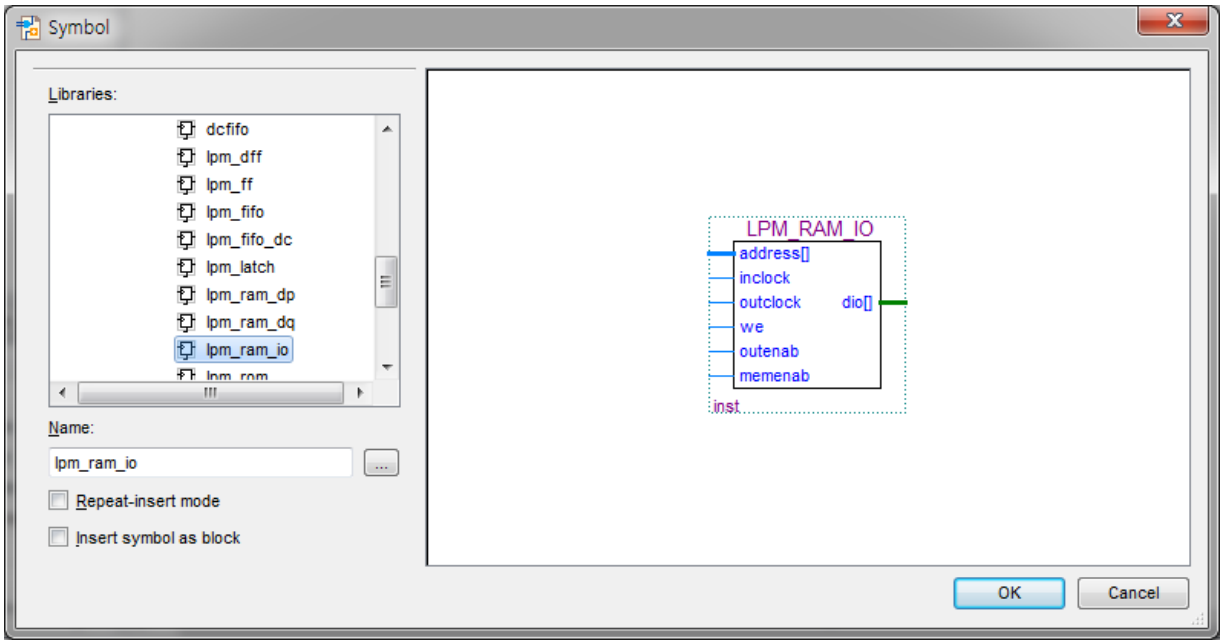
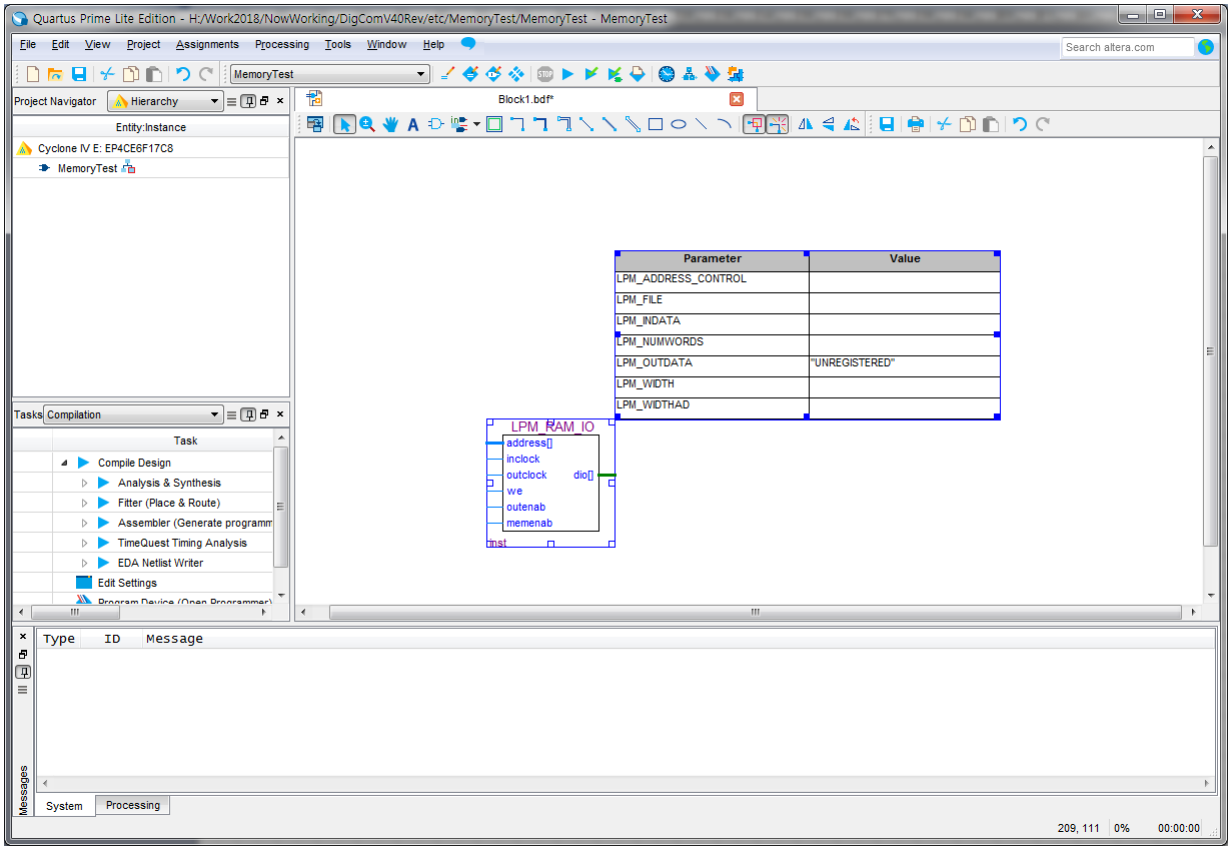


그림 3-10. LPM_RAM_IO 선택

■ OK 를 클릭하여 선택해서 설계에 포함시킨다.



3-11. 설계에 포함된 메모리

■ Parameter, Value 상자를 클릭하여 "Symbol Properties"를 설정한다. 이 때 설정할 값들은 아래와 같다.

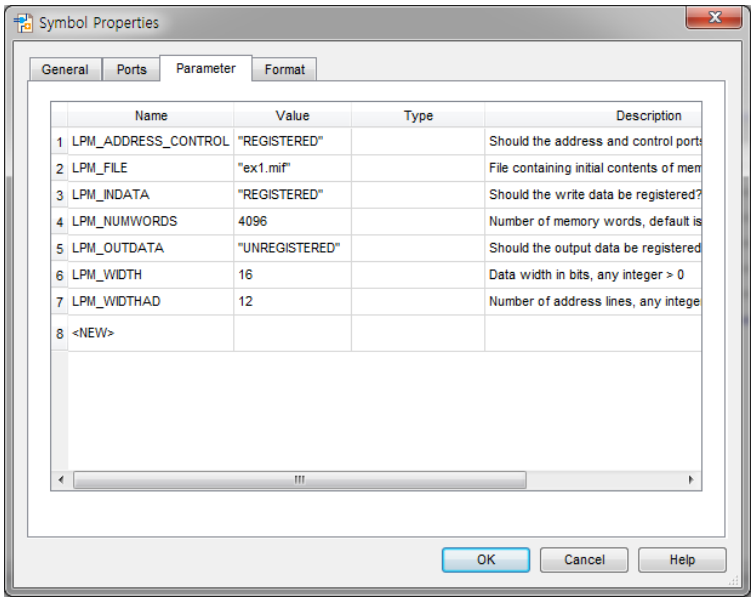


그림 3-12. 파라미터 값 설정

- LPM_ADDRESS_CONTROL : "REGISTERED"
- LPM_FILE : 메모리에 입력할 *.mif 프로그램 파일명을 입력("ex1.mif")
- LPM_INDATA : "REGISTERED"
- LPM_NUMWORDS : 4096(주소선이 12 개이므로 전체 4,096 word)
- LPM_OUTDATA : "UNREGISTERED"
- LMP_WIDTH : 16(데이터 길이)
- LPM_WIDTHAD : 12(주소선 개수)

■ Address, data 버스와 inclock 을 제외한 나머지 신호선들을 연결한다.

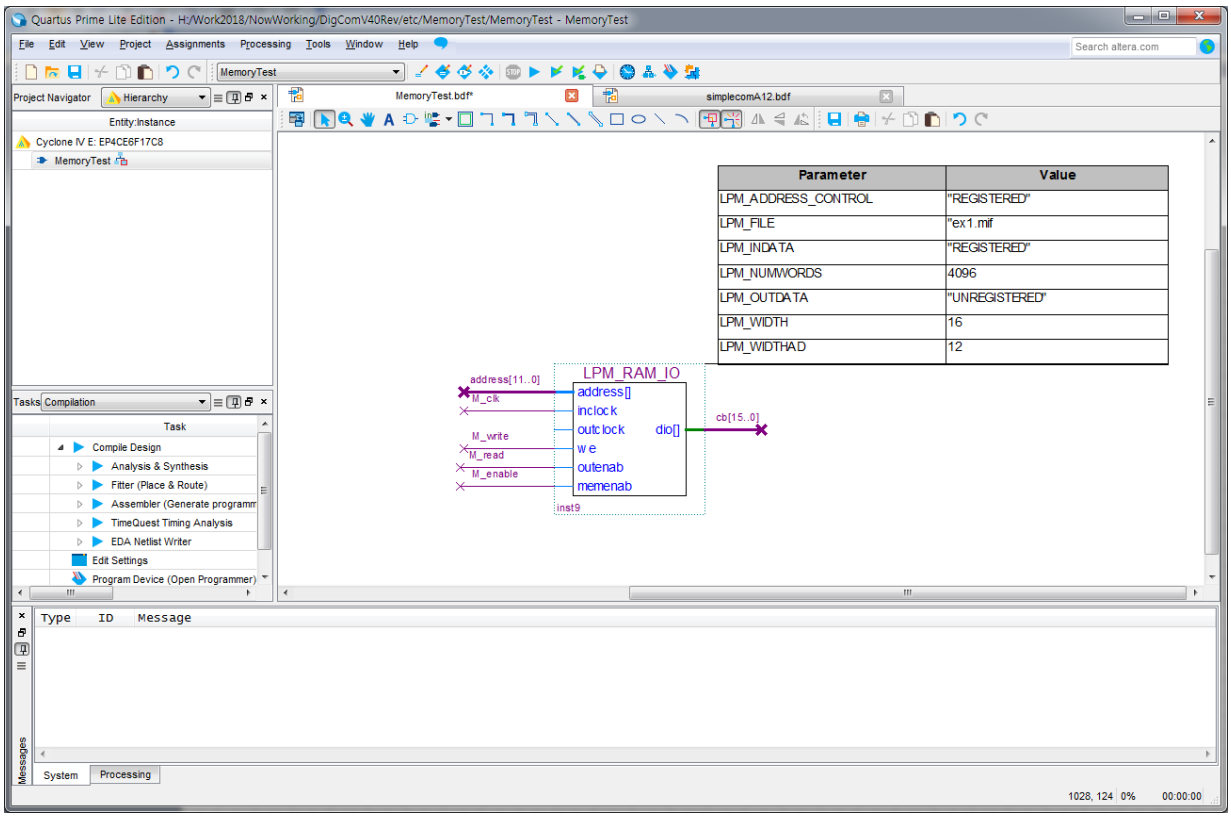


그림 3-13. 메모리의 데이터와 신호선 연결

3.8.2. 메모리 읽기/쓰기 동작

메모리 읽기는 메모리 클럭에 상관없이 비 동기식으로 읽히며, 쓰기는 메모리 클럭의 하강에지에서 이루어진다.

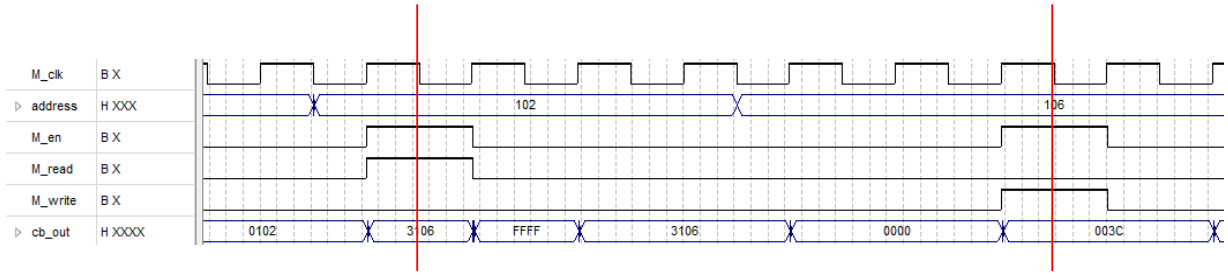


그림 3-14. 메모리에서 읽기/쓰기 타이밍

[그림 3-10]에서 처음 메모리 엑세스는 0x102 번지에서 0x3106 을 읽는 타이밍이며, 두 번째 엑세스는 0x106 번지에 0x003C 를 쓰는 타이밍이다.

3.9. 레지스터 출력

기본 컴퓨터에는 모두 8 개의 레지스터가 있다. 컴퓨터의 동작을 정확하게 이해하고, 프로그램의 디버깅을 위해서 내부 레지스터 값을 모니터링을 하는 것이 매우 효과적이다. 그러나 8 개의 16 비트 레지스터를 모두 출력하기 위해서는 매우 많은 출력장치가 있어야 하지만, 4 개의 7- 세그먼트를 이용하여 푸시버튼 스위치를 누를 때마다 순차적으로 각 레지스터 값이 하나씩 출력되도록 하였다.

코드 3-23

```
process(nReset, select_node, reg_node)
begin
        if nReset = '0' then
                reg_select <= "000";
        elsif rising_edge(select_node) then
                reg_select <= reg_select + '1';
        end if;
end process;
```

코드 3-23 에서 select_node 는 스위치를 한번 누를 때마다 발생하는 클럭이다. reg_select 를 초기화 한 다음 select_node 클럭이 발생할 때마다 3 비트 reg_select 를 하나씩 증가시키며, 이 값에 따라서 8 개중의 한 레지스터를 출력한다.

VHDL 을 이용한
기본 컴퓨터의 설계

(DIGCOM-A1.2)

부 록

2018-01-10

목차_Toc493512853

부록 1. 예제 프로그램 명령어 단위 실행 결과 1

1. 두수를 가산하는 프로그램 (ex1.s) 1

2. 두 수를 감산하는 프로그램(ex2.s) 2

3. 10 개의 수를 연속적으로 더하는 프로그램(ex3.s) 3

4. 두 개의 배정밀도를 가산하는 프로그램(ex4.s) 5

5. 서브루틴 사용을 보이는 프로그램(ex5.s) 7

6. 파라미터 링키지를 보이는 프로그램(ex6.s) 9

7. 데이터 블록을 이동시키는 프로그램(ex7.s) 11

8. 한 문자를 입출력시키는 프로그램(ex8.s) 14

9. 연속적으로 출력을 시키는 프로그램(ex9.s) 15

10. 간단한 인터럽트 프로그램(ex10.s) 17

부록 2. 어셈블러 프로그래밍 21

1. 개요

1.1. 기계어 21

1.2. 2 진 코드로 번역 22

1.3. 주소심볼 테이블(address symbol table) 23

1.4. 1st 패스 23

1.5. 2nd 패스 23

2. 어셈블러 프로그래밍 24

2.1. 명령어 데이터 구조 24

2.2. 슈도 명령어 테이블 25

2.3. MRI 테이블 25

2.4. Non-MRI 테이블 26

2.5. 주소 심볼 테이블 27

2.6. Main 함수 29

2.7. 1st 패스 함수 30

2.8. 2nd 패스 함수 31

2.9. ParseLineCode() 함수 32

2.10. 컴퓨터 명령어 부 함수(Computer instruction sub functions) 33

부록 1. 예제 프로그램 명령어 단위 실행 결과

1. 두수를 가산하는 프로그램 (ex1.s)

■ ex1.s 프로그램과 어셈블된 기계어 코드

프로그램 소스	주소	기계어 코드
ORG 100		
LDA A	0100	2104
ADD B	0101	1105
STA C	0102	3105
HLT	0103	7001
A, DEC 83	0104	0053
B, DEC -23	0105	FFE9
C, HEX 0	0106	0000
END		

■ 명령어 단위 프로그램 실행결과

레지스터,FF / 명령어	레지스터(HEX)								Flip Flop	출력
	AR	PC	DR	AC	INPR	IR	TR	OUTR	I S E R IEN FGI FGO	
ORG 100	000	100	0000	0000	00	0000	0000	00	0 1 0 0 0 0 1	
LDA A	104	101	0053	0053	00	2104	0000	00	0 1 0 0 0 0 1	
ADD B	105	102	FFE9	003C	00	1105	0000	00	0 1 1 0 0 0 1	
STA C	106	103	FFE9	003C	00	3106	0000	00	0 1 1 0 0 0 1	
HLT	001	104	FFE9	003C	00	7001	0000	00	0 0 1 0 0 0 1	
A, DEC 83										
B, DEC -23										
C, HEX 0 → 3C										

두 수를 더하는 프로그램은 명령어 순서대로 실행이 되므로 실행모드 스위치를 가운데에 두고 스텝 스위치를 누를 때마다 하나의 명령어가 실행된다. 실행결과에서 아래에 있는 C 번지 데이터는 초기값은 0 이지만 덧셈결과가 0x3C 로 되는 것을 나타낸다.

2. 두 수를 감산하는 프로그램(ex2.s)

■ ex2.s 프로그램과 어셈블된 기계어 코드

프로그램 소스		주소	기계어 코드
	ORG 100		
	LDA SUB	0100	2107;
	CMA	0101	7200;
	INC	0102	7020;
	ADD MIN	0103	1106;
	STA DIF	0104	3108;
	HLT	0105	7001;
MIN,	DEC 83	0106	0053;
SUB,	DEC -23	0107	FFE9;
DIF,	HEX 0	0108	0000;
	END	0109	0000;

■ 명령어 단위 프로그램 실행결과

레지스터,FF / 명령어	레지스터(HEX)								Flip Flop	출력
	AR	PC	DR	AC	INPR	IR	TR	OUTR	I S E R IEN FGI FGO	
ORG 100	000	100	0000	0000	00	0000	0000	00	0 1 0 0 0 0 0 1	
LDA SUB	107	101	FFE9	FFE9	00	2107	0000	00	0 1 0 0 0 0 0 1	
CMA	200	102	FFE9	0016	00	7200	0000	00	0 1 0 0 0 0 0 1	
INC	020	103	FFE9	0017	00	7020	0000	00	0 1 0 0 0 0 0 1	
ADD MIN	106	104	0053	006A	00	1106	0000	00	0 1 0 0 0 0 0 1	
STA DIF	108	105	0053	006A	00	3108	0000	00	0 1 0 0 0 0 0 1	
HLT	001	106	0053	006A	00	7001	0000	00	0 0 0 0 0 0 0 1	
MIN, DEC 83										
SUB, DEC -23										
DIF, HEX 0 ➔ 6A										

두 수를 빼는 프로그램도 더하는 프로그램과 마찬가지로 명령어 순서대로 실행이 된다. 실행결과 DIF 번지의 값은 초기에는 0 이었으나 뺄셈결과 0x6A 가 저장되는 것을 나타낸다.

3. 10 개의 수를 연속적으로 더하는 프로그램(ex3.s)

■ ex3.s 프로그램과 어셈블된 기계어 코드

프로그램 소스		주소	기계어 코드
	ORG 100		
	LDA ADS	0100	210B
	STA PTR	0101	310C
	LDA NBR	0102	210D
	STA CTR	0103	310E
	CLA	0104	7800
LOP,	ADD PTR I	0105	910C
	ISZ PTR	0106	610C
	ISZ CTR	0107	610E
	BUN LOP	0108	4105
	STA SUM	0109	310F
	HLT	010A	7001
ADS,	HEX 150	010B	0150
PTR,	HEX 0	010C	0000
NBR,	DEC -10	010D	FFF6
CTR,	HEX 0	010E	0000
SUM,	HEX 0	010F	0000
	ORG 150	0150	004B
	DEC 75	0151	0041
	DEC 65	0152	0017
	DEC 23	0153	004E
	DEC 78	0154	FFF3
	DEC -13	0155	0009
	DEC 9	0156	000E
	DEC 14	0157	FFD0
	DEC -48	0158	0018
	DEC 24	0159	0017
	DEC 23	015A	0000
	END		

■ 명령어 단위 프로그램 실행결과

레지스터,FF / 명령어	레지스터(HEX)								Flip Flop	출력
	AR	PC	DR	AC	INPR	IR	TR	OUTR	I S E R IEN FGI FGO	
ORG 100	000	100	0000	0000	00	0000	0000	00	0 1 0 0 0 0 1	
LDA ADS	10B	101	0150	0150	00	210B	0000	00	0 1 0 0 0 0 1	
STA PTR	10C	102	0150	0150	00	310C	0000	00	0 1 0 0 0 0 1	
LDA NBR	10D	103	FFF6	FFF6	00	210D	0000	00	0 1 0 0 0 0 1	
STA CTR	10E	104	FFF6	FFF6	00	310E	0000	00	0 1 0 0 0 0 1	

CLA	800	105	FFF6	0000	00	7800	0000	00	0 1 0 0 0 0 1	
LOP, ADD PTR I	150	106	004B	004B	00	910C	0000	00	1 1 0 0 0 0 1	
ISZ PTR	10C	107	0151	004B	00	610C	0000	00	0 1 0 0 0 0 1	
ISZ CTR	10E	108	FFF7	004B	00	610E	0000	00	0 1 0 0 0 0 1	
BUN LOP	105	105	FFF7	004B	00	4105	0000	00	0 1 0 0 0 0 1	
가)										
LOP, ADD PTR I	151	106	0041	008C	00	910C	0000	00	1 1 0 0 0 0 1	
ISZ PTR	10C	107	0152	008C	00	610C	0000	00	0 1 0 0 0 0 1	
ISZ CTR	10E	108	FFF8	008C	00	610E	0000	00	0 1 0 0 0 0 1	
BUN LOP	105	105	FFF8	008C	00	4105	0000	00	0 1 0 0 0 0 1	
나)										
LOP, ADD PTR I	159	106	0017	00FA	00	910C	0000	00	1 1 0 0 0 0 1	
ISZ PTR	10C	107	005A	00FA	00	610C	0000	00	0 1 0 0 0 0 1	
ISZ CTR	10E	109	0000	00FA	00	610E	0000	00	0 1 0 0 0 0 1	
BUN LOP										
STA SUM	10F	10A	0000	00FA	00	310F	0000	00	0 1 0 0 0 0 1	
HLT	001	10B	0000	00FA	00	7001	0000	00	0 0 0 0 0 0 1	
ADS, HEX 150										
PTR, HEX 0 ➔ 0x150 ➔ 0x151 ➔ 0x152 ➔ 0x153 ➔ 0x154 ➔ 0x155 ➔ 0x156 ➔ 0x157 ➔ 0x158 ➔ 0x159 ➔ 0x15A										
NBR, DEC -10										
CTR, HEX 0 ➔ 0xFFF6 ➔ 0xFFF7 ➔ 0xFFF8 ➔ 0xFFF9 ➔ 0xFFFA ➔ 0xFFFB ➔ 0xFFFC ➔ 0xFFFD ➔ 0xFFFE ➔ 0xFFFF ➔ 0x0000										
SUM, HEX 0 ➔ 4B ➔ 8C ➔ A3 ➔ F1 ➔ E4 ➔ ED ➔ FB ➔ CB ➔ E3 ➔ FA										

위 프로그램은 10 개의 수를 더하는 프로그램이며 피연산자는 0x150 번지부터 저장되어 있다. 프로그램 실행은 10 개의 수를 더하기 위해 LOP 레이블을 반복실행한다. 위 실행결과에서 **가)** 위치부터는 LOP 레이블로 Branch 해서 반복 실행된 결과를 보여주며, 같은 방법으로 9 개의 피연산자를 반복실행을 하면서 더하며, **나)** 위치부터는 마지막 피연산자인 0x159 번지의 데이터를 더하는 명령어 실행의 결과를 보여준다. 이 부분에서 "ISZ CTR" 명령에서 CTR 번지의 카운터를 증가시키면 0 이 되므로 "BUN LOP"을 실행하지 않고 최종 결과를 SUM 번지에 저장하고 프로그램을 종료한다. 프로그램 마지막 부분의 PTR, CTR 및 SUM 에는 프로그램이 실행되면서 바뀌는 값을 보여준다.

4. 두 개의 배정밀도를 가산하는 프로그램(ex4.s)

■ ex4.s 프로그램과 어셈블된 기계어 코드

프로그램 소스	주소	기계어 코드
ORG 100		
LDA AL	0100	2109
ADD BL	0101	110B
STA CL	0102	310D
CLA	0103	7800
CIL	0104	7040
ADD AH	0105	110A
ADD BH	0106	110C
STA CH	0107	310E
HLT	0108	7001
AL, HEX C186	0109	C186
AH, HEX 34A7	010A	34A7
BL, HEX A8B5	010B	A8B5
BH, HEX 5CA2	010C	5CA2
CL,		
CH,		
END		

■ 명령어 단위 프로그램 실행결과

레지스터,FF / 명령어	레지스터(HEX)								Flip Flop	출력
	AR	PC	DR	AC	INPR	IR	TR	OUTR	I S E R IEN FGI FGO	
ORG 100	000	100	0000	0000	00	0000	0000	00	0 1 0 0 0 0 1	
LDA AL	109	101	C186	C186	00	2109	0000	00	0 1 0 0 0 0 1	
ADD BL	10B	102	A8B5	6A3B	00	110B	0000	00	0 1 1 0 0 0 1	
STA CL	10D	103	A8B5	6A3B	00	310D	0000	00	0 1 1 0 0 0 1	
CLA	800	104	A8B5	0000	00	7800	0000	00	0 1 1 0 0 0 1	
CIL	040	105	A8B5	0001	00	7040	0000	00	0 1 0 0 0 0 1	
ADD AH	10A	106	34A7	34A8	00	110A	0000	00	0 1 0 0 0 0 1	
ADD BH	10C	107	5CA2	914A	00	110C	0000	00	0 1 0 0 0 0 1	
STA CH	10E	108	5CA2	914A	00	310E	0000	00	0 1 0 0 0 0 1	
HLT	001	109	5CA2	914A	00	7001	0000	00	0 0 0 0 0 0 1	
AL, HEX C186										
AH, HEX 34A7										
BL, HEX A8B5										

BH, HEX 5CA2
CL, HEX 0 ➔ 0x6A3B
CH, HEX 0 ➔ 0x914A

기본 컴퓨터의 데이터는 16 비트이므로 32 비트 데이터를 더하기 위해서는 16 비트씩 더해야 한다. 위 프로그램에는 Branch 명령이나 서브루틴이 없으므로 명령어 순서대로 실행되고 더한 결과는 CL 과 CH 번지에 저장된다.

5. 서브루틴 사용을 보이는 프로그램(ex5.s)

■ ex5.s 프로그램과 어셈블된 기계어 코드

프로그램 소스	주소	기계어 코드
ORG 100		
LDA X	0100	2107
BSA SH4	0101	5109
STA X	0102	3107
LDA Y	0103	2108
BSA SH4	0104	5109
STA Y	0105	3108
HLT	0106	7001
X, HEX 1234	0107	1234
Y, HEX 4321	0108	4321
	0109	0000
SH4, HEX 0	010A	7040
CIL	010B	7040
CIL	010C	7040
CIL	010D	7040
CIL	010E	0110
AND MSK	010F	C109
BUN SH4 I	0110	FFF0
MSK, HEX FFF0	0111	0000
END		

명령어 단위 프로그램 실행결과

레지스터,FF / 명령어	레지스터(HEX)								Flip Flop	출력	순서
	AR	PC	DR	AC	INPR	IR	TR	OUTR	I S E R IEN FGI FGO		
ORG 100	000	100	0000	0000	00	0000	0000	00	0 1 0 0 0 0 1		
LDA X	107	101	1234	1234	00	2107	0000	00	0 1 0 0 0 0 1		1
BSA SH4	10A	10A	1234	1234	00	5109	0000	00	0 1 0 0 0 0 1		2
STA X	107	103	FFF0	2340	00	3107	0000	00	0 1 1 0 0 0 1		9
LDA Y	108	104	4321	4321	00	2108	0000	00	0 1 1 0 0 0 1		10
BSA SH4	10A	10A	4321	4321	00	5109	0000	00	0 1 1 0 0 0 1		11
STA Y	108	106	FFF0	3210	00	3108	0000	00	0 1 0 0 0 0 1		18
HLT	001	107	FFF0	3210	00	7001	0000	00	0 0 0 0 0 0 1		19
SH4, HEX 0 ➔ 0x102											
CIL	040	10B	1234	2468	00	7040	0000	00	0 1 0 0 0 0 1		3
CIL	040	10C	1234	48D0	00	7040	0000	00	0 1 0 0 0 0 1		4

CIL	040	10D	1234	91A0	00	7040	0000	00	0 1 0 0 0 0 1		5
CIL	040	10E	1234	2340	00	7040	0000	00	0 1 1 0 0 0 1		6
AND MSK	110	10F	FFF0	2340	00	0110	0000	00	0 1 1 0 0 0 1		7
BUN SH4 I	102	102	FFF0	2340	00	C109	0000	00	1 1 1 0 0 0 1		8
SH4, HEX 0 ➔ 0x105											
CIL	040	10B	4321	8643	00	7040	0000	00	0 1 0 0 0 0 1		12
CIL	040	10C	4321	0C86	00	7040	0000	00	0 1 1 0 0 0 1		13
CIL	040	10D	4321	190D	00	7040	0000	00	0 1 0 0 0 0 1		14
CIL	040	10E	4321	321A	00	7040	0000	00	0 1 0 0 0 0 1		15
AND MSK	110	10F	FFF0	3210	00	0110	0000	00	0 1 0 0 0 0 1		16
BUN SH4 I	105	105	FFF0	3210	00	C109	0000	00	1 1 0 0 0 0 1		17
MSK HEX FFF0											

서브루틴을 호출하는 프로그램을 명령어 단위로 추적하는 것은 약간 복잡하다. “BSA SH4”는 SH4 서브루틴을 호출하며 위 프로그램에서는 서브루틴을 두 번 호출한다. 프로그램이 실행되는 순서를 보여주기 위해 오른쪽 칸에 순서를 번호로 표시하였다. 즉, 순서 2 의 “BSA SH4” 명령어가 실행되면 SH4 번지에 리턴되는 주소 0x102 가 저장되고 그 다음에 “CIL” 명령이 실행된다. 서브루틴의 마지막 명령어이 “BUN SH4 I” 다음에는 다시 메인 프로그램의 명령어 “STA X”가 실행된다. 이와 같은 방법으로 서브루틴이 한번 더 실행되고 프로그램이 종료된다.

6. 파라미터 링키지를 보이는 프로그램(ex6.s)

■ ex6.s 프로그램과 어셈블된 기계어 코드

프로그램 소스	주소	기계어 코드
ORG 100	0100	2105
LDA X	0101	5107
BSA OR	0102	3AF6
HEX 3AF6	0103	3106
STA Y	0104	7001
HLT	0105	7B95
X, HEX 7B95	0106	0000
Y, HEX 0	0107	0000
OR, HEX 0	0108	7200
CMA	0109	3110
STA TMP	010A	A107
LDA OR I	010B	7200
CMA	010C	0110
AND TMP	010D	7200
CMA	010E	6107
ISZ OR	010F	C107
BUN OR I	0110	0000
TMP, HEX 0	0111	0000
END		

■ 명령어 단위 프로그램 실행결과

레지스터,FF / 명령어	레지스터(HEX)								Flip Flop	출력	순서
	AR	PC	DR	AC	INPR	IR	TR	OUTR	I S E R IEN FGI FGO		
ORG 100	000	100	0000	0000	00	0000	0000	00	0 1 0 0 0 0 1		
LDA X	105	101	7B95	7B95	00	2105	0000	00	0 1 0 0 0 0 1		1
BSA OR	108	108	7B95	7B95	00	5107	0000	00	0 1 0 0 0 0 1		2
HEX 3AF6											
STA Y	106	104	0103	7BF7	00	3106	0000	00	0 1 0 0 0 0 1		11
HLT	001	105	0103	7BF7	00	7001	0000	00	0 0 0 0 0 0 1		12
X, HEX 7B95											
Y, HEX 0 ➔ 0x7BF7											
OR, HEX ➔ 0x102 ➔ 0x103											
CMA	200	109	7B95	846A	00	7200	0000	00	0 1 0 0 0 0 1		3
STA TMP	110	10A	7B95	846A	00	3110	0000	00	0 1 0 0 0 0 1		4

LDA OR I	102	10B	3AF6	3AF6	00	A107	0000	00	1 1 0 0 0 0 1		5
CMA	200	10C	3AF6	C509	00	7200	0000	00	0 1 0 0 0 0 1		6
AND TMP	110	10D	846A	8408	00	0110	0000	00	0 1 0 0 0 0 1		7
CMA	200	10E	846A	7BF7	00	7200	0000	00	0 1 0 0 0 0 1		8
ISZ OR	107	10F	0103	7BF7	00	6107	0000	00	0 1 0 0 0 0 1		9
BUN OR I	103	103	0103	7BF7	00	C107	0000	00	1 1 0 0 0 0 1		10
TMP, HEX 0 ➔ 0x846A											

위 프로그램도 서브루틴을 호출하는 프로그램이지만 파라미터를 전달하는 방법이 ex5 와 다르다. 순서 2 “BSA OR”이 실행되면 OR 번지에 리턴주소 0x102 가 저장되지만 0x102 번지에는 서브루틴으로 전달할 파라미터 0x3AF6 이 저장되어 있고 실제로 리턴되는 주소는 0x103 이다. 3 번째 실행되는 명령어는 서버루틴이 시작되는 CMA 명령이며 서브루틴이 끝나기 전에 “ISZ OR”에 의해 OR 번지에 저장된 리턴주소를 0x103 으로 하나 증가시킨다.

7. 데이터 블록을 이동시키는 프로그램(ex7.s)

■ ex7.s 프로그램과 어셈블된 기계어 코드

프로그램 소스		주소	기계어 코드
	ORG 100		
	BSA MVE	0100	5105
	HEX 120	0101	0120
	HEX 140	0102	0140
	DEC -6	0103	FFFA
	HLT	0104	7001
MVE,	HEX 0	0105	0000
	LDA MVE I	0106	A105
	STA PT1	0107	3116
	ISZ MVE	0108	6105
	LDA MVE I	0109	A105
	STA PT2	010A	3117
	ISZ MVE	010B	6105
	LDA MVE I	010C	A105
	STA CTR	010D	3118
	ISZ MVE	010E	6105
LOP,	LDA PT1 I	010F	A116
	STA PT2 I	0110	B117
	ISZ PT1	0111	6116
	ISZ PT2	0112	6117
	ISZ CTR.	0113	6118
	BUN LOP	0114	410F
	BUN MVE I	0115	C105
PT1,			
PT2,			
CTR,			
	ORG 120		
	DEC 11	0120	000B
	DEC 22	0121	0016
	DEC 33	0122	0021
	DEC 44	0123	002C
	DEC 55	0124	0037
	DEC 66	0125	0042
	ORG 140		
	END		

■ 명령어 단위 프로그램 실행결과

레지스터,FF / 명령어	레지스터(HEX)								Flip Flop	출력	순서
	AR	PC	DR	AC	INPR	IR	TR	OUTR	I S E R IEN FGI FGO		
ORG 100	000	100	0000	0000	00	0000	0000	00	0 1 0 0 0 0 1		
BSA MVE	106	106	0000	0000	00	5105	0000	00	0 1 0 0 0 0 1		1
HEX 120											
HEX 140											
DEC -6											
HLT	001	105	0000	0042	00	7001	0000	00	0 0 0 0 0 0 1		41
MVE, HEX 0 ➔ 101 ➔ 102 ➔ 103 ➔ 104											
LDA MVE I	101	107	0120	0120	00	A105	0000	00	1 1 0 0 0 0 1		2
STA PT1	116	108	0120	0120	00	3116	0000	00	0 1 0 0 0 0 1		3
ISZ MVE	105	109	0102	0120	00	6105	0000	00	0 1 0 0 0 0 1		4
LDA MVE I	102	10A	0140	0140	00	A105	0000	00	1 1 0 0 0 0 1		5
STA PT2	117	10B	0140	0140	00	3117	0000	00	0 1 0 0 0 0 1		6
ISZ MVE	105	10C	0103	0140	00	6105	0000	00	0 1 0 0 0 0 1		7
LDA MVE I	103	10D	FFFA	FFFA	00	A105	0000	00	1 1 0 0 0 0 1		8
STA CTR	118	10E	FFFA	FFFA	00	3118	0000	00	0 1 0 0 0 0 1		9
ISZ MVE	105	10F	0104	FFFA	00	6105	0000	00	0 1 0 0 0 0 1		10
LOP, LDA PT1 I	120	110	000B	000B	00	A116	0000	00	1 1 0 0 0 0 1		11
STA PT2 I	140	111	000B	000B	00	B117	0000	00	1 1 0 0 0 0 1		12
ISZ PT1	116	112	0121	000B	00	6116	0000	00	0 1 0 0 0 0 1		13
ISZ PT2	117	113	0141	000B	00	6117	0000	00	0 1 0 0 0 0 1		14
ISZ CTR	118	114	FFFB	000B	00	6118	0000	00	0 1 0 0 0 0 1		15
BUN LOP	10F	10F	FFFB	000B	00	410F	0000	00	0 1 0 0 0 0 1		16
LOP, LDA PT1 I	121	110	0016	0016		A116			1 1 0 0 0 0 1		17
STA PT2 I	141	111	0016	0016		B117			1 1 0 0 0 0 1		18
ISZ PT1	116	112	0122	0016		6116			0 1 0 0 0 0 1		19
ISZ PT2	117	113	0142	0016		6117			0 1 0 0 0 0 1		20
ISZ CTR	118	114	FFFC	0016		6118			0 1 0 0 0 0 1		21

BUN LOP	10F	10F	FFFC	0016		410F			0 1 0 0 0 0 1		22
LOP, LDA PT1 I	125	110	0042	0042		A116			1 1 0 0 0 0 1		35
STA PT2 I	145	111	0042	0042		B117			1 1 0 0 0 0 1		36
ISZ PT1	116	112	0126	0042		6116			0 1 0 0 0 0 1		37
ISZ PT2	117	113	0146	0042		6117			0 1 0 0 0 0 1		38
ISZ CTR	118	115	0000	0042		6118			0 1 0 0 0 0 1		39
BUN LOP											
BUN MVE I	104	104	0000	0042		C105			1 1 0 0 0 0 1		40
PT1, HEX 0 ➔ 0x121 ➔ 0x122 ➔ 0x123 ➔ 0x124 ➔ 0x125											
PT2, HEX 0 ➔ 0x141 ➔ 0x142 ➔ 0x143 ➔ 0x144 ➔ 0x145											
CTR, HEX 0 ➔ 0xFFFA ➔ 0xFFFB ➔ 0xFFFC ➔ 0xFFFD ➔0xFFFE ➔0xFFFF ➔ 0x0000											
ORG 120											
DEC 11											
DEC 22											
DEC 33											
DEC 44											
DEC 55											
DEC 66											

위 프로그램은 0x120 번지에 있는 6 개의 데이터를 0x140 번지로 이동하는 프로그램이다. 시작하자마자 서브루틴을 호출하며 서브루틴에서 반복문을 실행하면서 데이터를 하나씩 이동한다. 순서번호 16 까지 실행하면 하나의 데이터가 이동하고 다시 22 번까지 실행하면 다음 데이터가 이동한다. 마지막 데이터를 이동하는 반복문은 35 번부터이고 0x125 번지 데이터 66 이 0x145 번지로 이동된다.

8. 한 문자를 입출력시키는 프로그램(ex8.s)

■ ex8.s 프로그램과 어셈블된 기계어 코드

프로그램 소스	주소	기계어 코드
ORG 100		
CIF, SKI	0100	F200
BUN CIF	0101	4100
INP	0102	F800
COF, SKO	0103	F100
BUN COF	0104	4103
OUT	0105	F400
STA CHR	0106	3108
HLT	0107	7001
CHR,	0108	0000
END		

■ 명령어 단위 프로그램 실행결과

레지스터,FF / 명령어	레지스터(HEX)								Flip Flop	출력	순서
	AR	PC	DR	AC	INPR	IR	TR	OUTR	I S E R IEN FGI FGO		
ORG 100	000	100	0000	0000	00	0000	0000	00	0 1 0 0 0 0 1	00	
CIF, SKI	200	101	0000	0000	00	F200	0000	00	1 1 0 0 0 0 1	00	1
BUN CIF	100	100	0000	0000	00	4100	0000	00	0 1 0 0 0 0 1	00	2
Input 'A' (Press 'A' Button)											
CIF, SKI	200	102	0000	0000	0A	F200	0000	00	1 1 0 0 0 1 1	00	3
BUN CIF											
INP	800	103	0000	000A	0A	F800	0000	00	1 1 0 0 0 0 1	00	4
COF, SKO	100	105	0000	000A	0A	F100	0000	00	1 1 0 0 0 0 1	00	5
BUN COF											
OUT	400	106	0000	000A	0A	F400	0000	0A	1 1 0 0 0 0 0	0A	6
STA CHR	108	107	0000	000A	0A	3108	0000	0A	0 1 0 0 0 0 0	0A	7
HLT	001	108	0000	000A	0A	7001	0000	0A	0 0 0 0 0 0 0	0A	8

위 프로그램은 하나의 입력을 받아서 출력하는 프로그램이다. 입력 스위치를 누르지 않으면 입력 플래그가 세트되지 않으므로 1 과 2 를 반복 실행한다. 입력 키 스위치(0xA)를 누르면 "SKI" 명령어에 의해 다음 명령을 생략하고 4 번의 "INP"명령을 실행한다. 그리고 출력 플래그의 초기값은 '1'이므로 6 번의 "OUT" 명령에 의해 입력한 값이 그대로 출력된다.

9. 연속적으로 출력을 시키는 프로그램(ex9.s)

■ ex9.s 프로그램과 어셈블된 기계어 코드

프로그램 소스	주소	기계어 코드
ORG 100		
LDA NBR	0100	2109
STA CTR	0101	310A
LDA X	0102	2108
LOP, BSA SUB	0103	510B
INC	0104	7020
ISZ CTR	0105	610A
BUN LOP	0106	4103
HLT	0107	7001
X, HEX 10	0108	0010
NBR, DEC -2	0109	FFFE
CTR, HEX 0	010A	0000
SUB, HEX 0	010B	0000
LOA, SKO	010C	F100
BUN LOA	010D	410C
OUT	010E	F400
BUN SUB I	010F	C10B
END		

■ 명령어 단위 프로그램 실행결과

레지스터,FF / 명령어	레지스터(HEX)								Flip Flop	출력	순서
	AR	PC	DR	AC	INPR	IR	TR	OUTR	I S E R IEN FGI FGO		
ORG 100	000	100	0000	0000	00	0000	0000	00	0 1 0 0 0 0 1		
LDA NBR	109	101	FFFE	FFFE	00	2109	0000	00	0 1 0 0 0 0 1	00	1
STA CTR	10A	102	FFFE	FFFE	00	310A	0000	00	0 1 0 0 0 0 1	00	2
LDA X	108	103	0010	0010	00	2108	0000	00	0 1 0 0 0 0 1	00	3
LOP, BSA SUB	10C	10C	0010	0010	00	510B	0000	00	0 1 0 0 0 0 1	00	4
INC	020	105	0010	0011	00	7020	0000	10	0 1 0 0 0 0 0	10	8
ISZ CTR	10A	106	FFFF	0011	00	610A	0000	10	0 1 0 0 0 0 0	10	9
BUN LOP	103	103	FFFF	0011	00	4103	0000	10	0 1 0 0 0 0 0	10	10
HLT	001	108	0000	0012	00	7001	0000	11	0 0 0 0 0 0 0	11	19
X, HEX 10											
NBR, DEC -2											
CTR, HEX 0 ➔ FFFE ➔ FFFF ➔ 0000											

SUB, HEX 0 ➔104											
LOA, SKO	100	10E	0010	0010	00	F100	0000	00	1 1 0 0 0 0 1		5
BUN LOA											
OUT	400	10F	0010	0010	00	F400	0000	10	1 1 0 0 0 0 0	10	6
BUN SUB I	104	104	0010	0010	00	C10B	0000	10	1 1 0 0 0 0 0	10	7
LOP, BSA SUB	10C	10C	FFFF	0011	00	510B	0000	10	0 1 0 0 0 0 0	10	11
INC	020	105	FFFF	0012	00	7020	0000	11	0 1 0 0 0 0 0	11	17
ISZ CTR	10A	107	0000	0012	00	610A	0000	11	0 1 0 0 0 0 0	11	18
LOA, SKO	100	10D	FFFF	0011	00	F100	0000	10	1 1 0 0 0 0 0	10	12
BUN LOA	10C	10C	FFFF	0011	00	410C	0000	10	0 1 0 0 0 0 0	10	13
FGO KEY PRESS											
LOA, SKO	100	10E	FFFF	0011	00	F100	0000	10	1 1 0 0 0 0 1	10	14
BUN LOA											
OUT	400	10F	FFFF	0011	00	F400	0000	11	1 1 0 0 0 0 0	11	15
BUN SUB I	104	104	FFFF	0011	00	C10B	0000	11	1 1 0 0 0 0 0	11	16

위 프로그램에서는 X 번지에 저장된 0x10 을 출력시킨 후에 1 을 증가시켜서 다시 0x11 을 출력한다. 4 번 위치에서 “BSA SUB”에 의해 서브루틴을 호출하면 FGO 의 초기값은 ‘1’이므로 AC 에 저장된 0x10 이 출력된다. 다시 메일 프로그램으로 리턴되면 8 번위치부터 실행되고 AC 의 값을 ‘1’증가 시켜 0x11 로 만든 후에 11 번 위치에서 서브루틴을 또 호출한다. 그러나 FGO 가 ‘0’이기 때문에 서브루틴에서 12 번과 13 번 위치의 명령어를 반복실행하다가 FGO 키를 눌러서 플립플롭을 ‘1’로 세트하면 0x11 이 출력되고 17 번 위치에서 메인 프로그램으로 리턴한 후에 프로그램을 종료한다.

10. 간단한 인터럽트 프로그램(ex10.s)

■ ex10.s 프로그램과 어셈블된 기계어 코드

프로그램 소스		주소	기계어 코드
	ORG 0		
ZRO,	HEX 0	0000	0000
	BUN ISV	0001	4107
	ORG 100		
	CLA	0100	7800
	OUT	0101	F400
	ION	0102	F080
LOP,	LDA X	0103	2117
	ADD Y	0104	1118
	STA Z	0105	3119
	BUN LOP	0106	4103
ISV,	STA SAC	0107	311A
	CIR	0108	7080
	STA SE	0109	311B
	SKI	010A	F200
	BUN NXT	010B	410E
	INP	010C	F800
	STA X	010D	3117
NXT,	SKO	010E	F100
	BUN EXT	010F	4112
	LDA Z	0110	2119
	OUT	0111	F400
EXT,	LDA SE	0112	211B
	CIL	0113	7040
	LDA SAC	0114	211A
	ION	0115	F080
	BUN ZRO I	0116	C000
X,	HEX 10	0117	0010
Y,	HEX 20	0118	0020
Z,	HEX 0	0119	0000
SAC,	HEX 0	011A	0000
SE,	HEX 0	011B	0000
	END		

■ 명령어 단위 프로그램 실행결과

레지스터,FF / 명령어	레지스터(HEX)								Flip Flop	출력	순서
	AR	PC	DR	AC	INPR	IR	TR	OUTR	I S E R IEN FGI FGO		
ORG 0	000	000	0000	0000	00	0000	0000	00	0 0 0 0 0 0 0 1	00	
ZRO, HEX 0 ➔ 0x103											
BUN ISV	107	107	0020	0030	0A	4107	103	00	0 1 0 0 0 1 0	00	9
BUN ISV	107	107	0020	002A	0A	4107	103	00	0 1 0 0 0 0 1	00	28
ORG 100	000	100	0000	0000	00	0000	0000	00	0 1 0 0 0 0 1	00	
CLA	800	101	0000	0000	00	7800	0000	00	0 1 0 0 0 0 1	00	1
OUT	400	102	0000	0000	00	F400	0000	00	1 1 0 0 0 0 0	00	2
ION	080	103	0000	0000	00	F080	0000	00	1 1 0 0 1 0 0	00	3
LOP, LDA X	117	104	0010	0010	00	2117	0000	00	0 1 0 0 1 0 0	00	4
ADD Y	118	105	0020	0030	00	1118	0000	00	0 1 0 0 1 0 0	00	5
STA Z	119	106	0020	0030	00	3119	0000	00	0 1 0 0 1 0 0	00	6
Input 'A' Interrupt(Press 'A' Button)											
BUN LOP	103	103	0020	0030	0A	4103	0000	00	0 1 0 1 1 1 0	00	7
INT(FGI SET)	000	001	0020	0030	0A	4103	0103	00	0 1 0 0 0 1 0	00	8
LOP, LDA X	117	104	000A	000A	0A	2117	0103	00	0 1 0 0 1 0 0	00	23
ADD Y	008	105	0020	002A	0A	1118	0103	00	0 1 0 0 1 0 0	00	24
STA Z	119	106	0020	002A	0A	3119	0103	00	0 1 0 0 1 0 0	00	25
Output Interrupt(Press FGO key)											
BUN LOP	103	103	0020	002A	0A	4103	0103	00	0 1 0 1 1 0 1	00	26
INT(FGO SET)	000	001	0020	002A	0A	4103	0103	00	0 1 0 0 0 0 1	00	27
ISV, STA SAC	11A	108	0020	0030	0A	311A	0103	00	0 1 0 0 0 1 0	00	10
CIR	080	109	0020	0018	0A	7080	0103	00	0 1 0 0 0 1 0	00	11
STA SE	11B	10A	0020	0018	0A	311B	0103	00	0 1 0 0 0 1 0	00	12
SKI	200	10C	0020	0018	0A	F200	0103	00	0 1 0 0 0 1 0	00	13
BUN NXT											
INP	800	10D	0020	000A	0A	F800	0103	00	1 1 0 0 0 0 0	00	14

STA X	117	10E	0020	000A	0A	3117	0103	00	0 1 0 0 0 0 0	00	15
NXT, SKO	100	10F	0020	000A	0A	F100	0103	00	1 1 0 0 0 0 0	00	16
BUN EXT	112	112	0020	000A	0A	4112	0103	00	0 0 0 0 0 0 1	00	17
LDA Z											
OUT											
EXT, LDA SE	11B	113	0018	0018	0A	211B	0103	00	0 1 0 0 0 0 0	00	18
CIL	040	114	0018	0030	0A	7040	0103	00	0 1 0 0 0 0 0	00	19
LDA SAC	11A	115	0030	0030	0A	211A	0103	00	0 1 0 0 0 0 0	00	20
ION	080	116	0030	0030	0A	F080	0103	00	1 1 0 0 1 0 0	00	21
BUN ZRO I	103	103	0030	0030	0A	C000	0103	00	1 1 0 0 1 0 0	00	22
ISV, STA SAC	11A	108	0020	002A	0A	311A	0103	00	0 1 0 0 0 0 1	00	29
CIL	080	109	0020	0015	0A	7080	0103	00	0 1 0 0 0 0 1	00	30
STA SE	11B	10A	0020	0015	0A	311B	0103	00	0 1 0 0 0 0 1	00	31
SKI	200	10B	0020	0015	0A	F200	0103	00	1 1 0 0 0 0 1	00	32
BUN NXT	10E	10E	0020	0015	0A	410E	0103	00	0 1 0 0 0 0 1	00	33
INP											
STA X											
NXT, SKO	100	110	0020	0015	0A	F100	0103	00	1 1 0 0 0 0 1	00	34
BUN EXT											
LDA Z	119	111	002A	002A	0A	2119	0103	00	0 1 0 0 0 0 1	00	35
OUT	400	112	002A	002A	0A	F400	0103	2A	1 1 0 0 0 0 0	2A	36
EXT, LDA SE	11B	113	0015	0015	0A	211B	0103	2A	0 1 0 0 0 0 0	2A	37
CIL	040	114	0015	002A	0A	7040	0103	2A	0 1 0 0 0 0 0	2A	38
LDA SAC	11A	115	002A	002A	0A	211A	0103	2A	0 1 0 0 0 0 0	2A	39
ION	080	116	002A	002A	0A	F080	0103	2A	1 1 0 0 1 0 0	2A	40
BUN ZRO I	103	103	002A	002A	0A	C000	0103	2A	1 1 0 0 1 0 0	2A	41
X, HEX 10 ➔ 0xA											
Y, HEX 20											
Z, HEX 0 ➔ 0x30, ➔ 2A											
SAC,HEX 0 ➔ 0x30 ➔ 0x 2A											

SE,HEX 0 ➔ 0x18 ➔ 0x15

위 프로그램은 간단한 인터럽트 프로그램이지만 입력과 출력을 인터럽트로 처리하기 때문에 명령어 단위로 실행하는 과정은 간단하지 않다. 메인 프로그램은 0x100번지부터 시작하지만 인터럽트 처리 루틴(ISR : Interrtup Service Routine)은 0x1번지의 "BUN ISV"이다. 메인 프로그램이 시작되면 "OUT" 명령에 의해 FGO를 클리어시킨다. 이 것은 FGO가 세트되어 있으면 인터럽트를 인에이블시킴과 동시에 인터럽트가 걸리기 때문이다. 그리고 인터럽트를 인에이블시킨 후에 4번에서 7번까지의 명령어를 반복수행하면서 인터럽트를 기다린다. 이때 입력 스위치(=0xA)를 누르면 8번 위치에서 FGI 플래그가 '1'로 세트가 되면서 인터럽트가 발생한다(8번은 명령어에 의해 실행되는 것이 아니며 인터럽트가 발생되는 것을 보이기 위해 포함시킨 것임). 입력 스위치가 0x106번지의 명령("BUN LOP")이 실행되는 동안에 눌려졌다고 가정하면 0x0번지에 인터럽트의 리턴 주소에는 0x103이 저장된다. 그리고 9번위치의 ISR에 의해 ISV 레이블이 있는 위치로 branch를 한다. ISV 번지부터는 인터럽트를 발생한 장치를 검색하고 해당하는 프로그램을 실행을 해 주며, 첫 번째 인터럽트는 입력장치에서 발생했으므로 13번 위치에서 FGI가 세트된 것을 인식하고 그 다음 명령인 "INP" 명령을 실행하여 AC에 0xA를 읽어 들이고, X번지에 저장한다. 메인 프로그램으로 리턴을 하면 0x103 번지 "LDA X" 명령부터 시작되고(23번 위치), 역시 23번에서 26번 사이를 반복수행한다. 이때 AC에는 입력받은 0xA와 Y번지의 데이터 0x20이 더해진 0x2A가 저장되어 있다. 이 값을 출력하기 위해 FGO 키를 누르면 인터럽트가 걸리고 인터럽트 처리 루틴(28번 위치)에서 ISV로 branch를 해서 29번부터 실행이 된다. FGO에 의해 인터럽트가 발생했으므로 AC에 저장한 0x2A를 출력한 후 메인 프로그램으로 리턴하여 프로그램을 종료한다.

부록 2. 어셈블러 프로그래밍

1. 개요

어셈블러는 심볼로 표시된 명령어를 2 진 코드의 기계어로 번역해 주는 프로그램이다. 어셈블러의 입력은 어셈블리 언어로 구성된 원시 프로그램이며, 출력은 2 진 코드로 구성된 기계어 코드이다. 따라서 어셈블러는 문자열 프로그램을 동등한 2 진 코드로 변환한다.

1.1. 기계어

프로그램은 컴퓨터가 데이터를 처리해서 필요한 작업을 수행하기 위한 명령어 또는 서술문의 연속이다. 프로그램 언어에는 가장 많이 사용되는 C 언어 등 다양한 종류가 있으나 컴퓨터는 2 진 형태로 표현될 때만 수행이 가능하다. 모든 언어로 쓰여진 프로그램은 컴퓨터에서 실행되기 전에 명령어를 나타내는 2 진 코드로 번역이 되어야 한다. 이때 C 언어와 같이 고급 프로그래밍 언어를 2 진 기계어를 번역하는 과정을 컴파일이라 한다. 표 A4-1 은 기본컴퓨터에서 사용하는 명령어 세트와 이에 해당하는 2 진 코드를 보여준다.

표 A4-1. 기본 컴퓨터에서 사용하는 명령어 세트를 보여준다.

Symbol	Hexadecimal Code	Description
AND	0 or 8	AND M to AC
ADD	1 or 9	Add M to AC, carry to E
LDA	2 or A	Load AC from M
STA	3 or B	Store AC in M
BUN	4 or C	Branch unconditionally to m
BSA	5 or D	Save return address in m and branch to m+ 1
ISZ	6 or E	Increment M and and skip if zero
CLA	7800	Clear AC
CLE	7400	Clear E
CMA	7200	Complement AC
CME	7100	Complement E
CIR	7080	Circulate right E and AC
CIL	7040	Circulate left E and AC
INC	7020	Increment AC
SPA	7010	Skip if AC is positive
SNA	7008	Skip if AC is negative
SZA	7004	Skip if AC is zero
SZE	7002	Skip if E is zero
HLT	7001	Halt computer
INP	F800	Input information and clear flag
OUT	F400	Output information and clear flag
SKI	F200	Skip if input flag is on
SKO	F100	Skip if output flag is on
ION	F080	Turn interrupt on

IOF	F040	Turn interrupt off

1.2. 2 진 코드로 번역

일반적으로 어셈블러 입력은 ASCII 문자로 작성된 심볼 프로그램이다. 이 입력은 몇 라인의 코드로 구성되었으며, 각 라인은 문자, 숫자로 구성되어 있다. 각 라인 코드는 레이블, opcode, 주소 및 주석으로 구성되며, 한 예는 다음과 같다.

```
LOOP, ADD SUB I      // comment here
```

위 라인 코드에서 레이블은 LOOP, opcode 는 ADD, 주소는 SUB 그리고 슬래시(//)는 주석의 시작을 나타낸다. 문자 I 는 주소가 간접주소(indirect address) 임을 나타낸다. 어셈블러는 입력 프로그램의 각 라인 코드를 읽어서 각 심볼에 해당하는 기계어 코드로 교체한다. 표 A4-2 는 심볼 언어 프로그램의 예와 해당하는 기계어 2 진 코드를 보여준다. 이 예에서 첫 번째 라인의 ORG 는 슈도 명령어(Pseudo-instruction)이며 2 진 프로그램이 16 진수 주소 100 번지부터 시작한다는 것을 알려준다. 두 번째 라인에는 opcode LDA 및 주소 심볼 SUB 등 2 개의 심볼이 있다. 주소 심볼의 2 진 값은 주소 심볼 테이블에서 찾을 수 있으며 9 번째 라인의 레이블 필드에서 찾을 수 있다. 두 번째 라인이 16 진수 100 번지에 해당하는 명령어를 포함하고 있으며, 각 라인은 메모리에서 연속적으로 저장되는 명령어 또는 피 연산자 데이터를 나타내고 있다. 따라서 레이블 SUB 는 메모리 위치 107 번지에 해당하는 것을 알 수 있다. 그러므로 명령어 LDA 의 16 진수 주소는 107 이 된다. 명령어의 두 부분이 어셈블되면 100 번지의 16 진수 코드는 2107 를 얻을 수 있다. 프로그램의 다른 라인들도 같은 방법으로 번역과정을 통해 2 진 코드 또는 16 진 코드를 얻을 수 있다.

그러나 슈도 명령어는 번역과정을 거치지 않는다. 8 번째 라인에는 슈도 명령어 DEC 83 을 포함한다. 10 진수 83 이 2 진수로 번역되어 저장되며, 이 예에서는 106 번지 16 진수 53 이 저장된다. 슈도 명령어 HEX 도 유사한 방법으로 번역되며, 슈도 명령어 END 는 어셈블러에게 더 이상 번역할 라인이 없다는 것을 알린다.

표 A4-2. 어셈블리 프로그램(심볼릭 프로그램), 슈도명령어 및 기계어 코드

Hexadecimal code			
Location	Content	Symbolic program	
			ORG 100
100	2017		LDA SUB
101	7200		CMA
102	7020		INC
103	1106		ADD MIN
104	3108		STA DIF
105	7001		HLT
106	0053	MIN,	DEC 83
107	FFE9	SUB,	DEC -23
108	0000	DIF,	HEX 0

```
END
```

1.3. 주소심볼 테이블(address symbol table)

위에서 설명한 바와 같이 매번 주소의 16 진수 값을 결정하기 위해서 레이블 필드[1]를 검색해서 해당하는 위치의 주소를 찾아야 한다. 그러나 이 절차는 프로그램의 길이가 길어지게 되면 더욱 어려워지고 시간 소모적이다. 그러나 어셈블 하려는 프로그램을 2 번 스캔하면 어셈블 과정이 간단해진다. 첫 번째 스캔에서는 번역을 하지 않으며, 주소 심볼과 그에 해당하는 16 진수 값을 포함하는 테이블을 작성한다. 이 예에서 주소 심볼 테이블은 표 A4-3 과 같다.

표 A4-3. 주소 심볼 테이블

Address symbol	Hexadecimal addre
MIN	106
SUB	107
DIF	108

두 번째 스캔에서 주소 심볼 테이블을 참고해서 메모리 참조 명령어의 주소 값을 얻는다.

1.4. 1st 패스

2-패스 어셈블러는 전체 프로그램을 두 번 스캔한다. 첫 번째 스캔에서는 프로그래머가 정의한 심볼 주소를 그에 해당하는 2 진 값을 서로 관련시키는 테이블을 작성한다. 1st 패스에서는 명령어 위치를 계속 추적하기 위해 LC(Location Counter)를 사용한다. ORG 슈도 명령어는 LC 를 최초 값으로 초기화하고 각 라인 코드를 처리한 후에 1 씩 증가한다. 심볼 코드에서는 컴마에 의해서 레이블이 있는지를 검사한다. 라인 코드에 레이블이 없으면 어셈블러는 명령어 opcode 필드를 검사하고, 필드 값이 ORG 이면 LC 를 ORG 다음의 주소 값으로 초기화한 후 다음 라인을 처리한다. 다음 라인이 END 슈도 명령어이면 어셈블러는 1st 패스를 끝내고 2nd 패스로 넘어간다. 만일 라인 코드에 레이블을 포함하면 심볼 주소와 LC 의 값이 주소 심볼 테이블에 함께 저장되며, 레이블이 없으면 아무 것도 저장되지 않는다. 다음 LC 는 1 이 증가하여 새로운 라인을 처리한다.

1.5. 2nd 패스

기계어 명령어는 2nd 패스과정에서 테이블 룩업 과정(table lookup process)에 의해 변환된다. 테이블 룩업 과정은 특정한 항목이 테이블에 저장된 것과 일치를 하는지를 찾는 과정이며, 어셈블러에서는 4 개의 테이블을 사용한다. 프로그램에서 검색되는 어떠한 심볼도 4 개 중의 한 테이블의 항목과

[1] 레이블은 어셈블리어 프로그램에서 주소를 심볼로 나타낸 것을 의미함. 레이블 필드란 프로그램에서 레이블이 차지하는 프로그램 영역을 말함.

일치해야 하며 그렇지 않으면 심볼은 번역되지 않는다. 2nd 패스에서 사용되는 4 개의 테이블은 다음과 같다.

- 슈도 명령어 테이블 : 이 테이블의 항목은 ORG, END, DEC 및 HEX 의 4 개 슈도 명령어이다.
- MRI 테이블 : MRI 테이블은 메모리 참조 명령인 7 개의 심볼과 2 진 코드를 포함한다.
- Non-MRI 테이블 : non-MRI 테이블은 레지스터 참조 명령과 입출력 명령에 대한 심볼과 2 진 코드를 포함한다.
- 주소 심볼 테이블 : 1st 패스에서 만들어 지며, 주소 심볼과 이에 해당하는 주소를 포함한다.

어셈블러는 심볼 코드에 해당하는 2 진 값을 찾기 위해 현재 처리 중인 심볼을 테이블에서 찾아서 2 진 코드를 만들어서 출력파일에 저장한다.

2. 어셈블러 프로그래밍

어셈블러 프로그램은 심볼 프로그램을 2 진 코드로 변환하기 위해 사용된다. 이 절에서는 어셈블러 프로그램을 C 언어로 설계하고 구현하는 과정을 설명한다.

2.1. 명령어 데이터 구조

앞에서 언급된 것과 같이 어셈블러 프로그램은 4 개의 테이블을 사용한다. 주소 심볼 테이블은 1st 패스에서 생성되며, 나머지 3 개의 테이블은 명령어 테이블에 정의되었다. 이 테이블들을 구성하기 위해 코드 A4-1 과 같이 명령어 세트에 대한 데이터 구조체를 정의하였다. 이 구조체는 다음과 같은 두 개의 필드를 포함한다.

- Command name : 컴퓨터 명령어의 심볼로 구성된 문자열
- Pointer : 명령어를 2 진 코드로 변환하는 함수를 가리키는 포인터. 2nd 패스 동안 현재 처리중인 심볼이 Command name 과 일치하면 어셈블러는 동등한 2 진 코드를 발생시키기 위해 이 함수를 호출한다.

코드 A4-1. 명령어 세트의 구조체

```
typedef struct tagCmd
{
    Char            szCommandName[10];
    pCallBackfn     pExecute;
} COMMAND_SET, *PCOMMAND_SET;
```

2.2. 슈도 명령어 테이블

슈도 명령어 테이블은 코드 A4-2 와 같이 정의된다.

코드 A4-2. 슈도 명령어 테이블

```
// Pseudo-instruction table
COMMAND_SET PsedoInstrTB[] =
{
    {"ORG", &CCommand::fntAsmORG},
    {"END", &CCommand::fntAsmEND},
    {"DEC", &CCommand::fntAsmDEC},
    {"HEX", &CCommand::fntAsmHEX}
};
```

Listign ??은 어셈블러가 슈도 명령어 테이블을 검색하는 방법을 보여준다. 어셈블러는 처리 중인 라인 코드의 심볼과 일치하는 명령어 이름을 포함하는 구조체 요소를 찾을 때가지 슈도 명령어 테이블의 각 구조체 요소를 순차적으로 검색한다. 그러나 이 조건을 만족하는 구조체 요소가 없으면 어셈블러는 다른 테이블을 검색한다. 다른 테이블에도 일치하는 구조체 요소가 없으면 처리 중인 라인을 오류처리 한다. 어셈블러가 테이블에서 일치하는 구조체 요소를 찾으면 이 요소의 포인터 필드를 참조하여 해당하는 함수를 호출한다. 이 함수에서는 처리 중인 라인의 코드를 분석하여 2 진 코드로 변환하는 동작을 한다. 다음 어셈블러는 반복해서 다음 라인의 코드를 처리한다.

코드 A4-3. 슈도 명령어 테이블 검색

```
// Check Pseudo-instruction
while(intIndex < 4)
{
        pCommand = &PsedoInstrTB[intIndex];
        if(strLineCode.Find(pCommand->szCommandName) >= 0)
        {
                if(!CheckError(strLineCode, pCommand->szCommandName))
                        return FALSE;
                return(this->*(pCommand->pExecute))(strLineCode);
                //return TRUE;
        }
        intIndex++;
}
```

2.3. MRI 테이블

MRI 테이블은 메모리 참조명령의 7 개 심볼로 구성된다. MRI 테이블은 다음과 같이 정의된다.

코드 A4-4. MRI 테이블

```
COMMAND_SET cmdMriTB[] =
{
```

```
    {"ADD", &CCommand::fntAsmADD},
    {"AND", &CCommand::fntAsmAND},
    {"LDA", &CCommand::fntAsmLDA},
    {"STA", &CCommand::fntAsmSTA},
    {"BUN", &CCommand::fntAsmBUN},
    {"BSA", &CCommand::fntAsmBSA},
    {"ISZ", &CCommand::fntAsmISZ}
};
```

코드 A4-5 는 MRI 테이블을 검색하는 절차를 나타낸다.

코드 A4-5. MRI 테이블 검색

```
intIndex = 0;
    // Check MRI instruction
    while(intIndex < MIR_NUMBER)
    {
            pCommand = &cmdMriTB[intIndex];
            if(strLineCode.Find(pCommand->szCommandName) >= 0)
            {
                    if(!CheckError(strLineCode, pCommand->szCommandName))
                            return FALSE;
                    return (this->*(pCommand->pExecute))(strLineCode);
                    //return TRUE;
            }
            intIndex++;
    }
```

2.4. Non-MRI 테이블

Non-MRI 테이블은 18 개의 레지스터, 입출력 명령어의 심볼을 포함한다. Non-MRI 테이블은 코드 A4-6 와 같이 정의되며, A4-7 는 Non-MRI 명령어를 찾는 절차를 보여준다.

코드 A4-6. non_MRI 테이블

```
COMMAND_SET cmdNonMriTB[] =
{
    {"CLA", &CCommand::fntAsmCLA},
    {"CLE", &CCommand::fntAsmCLE},
    {"CMA", &CCommand::fntAsmCMA},
    {"CME", &CCommand::fntAsmCME},
    {"CIR", &CCommand::fntAsmCIR},
    {"CIL", &CCommand::fntAsmCIL},
    {"INC", &CCommand::fntAsmINC},
    {"SPA", &CCommand::fntAsmSPA},
    {"SNA", &CCommand::fntAsmSNA},
    {"SZA", &CCommand::fntAsmSZA},
    {"SZE", &CCommand::fntAsmSZE},
    {"HLT", &CCommand::fntAsmHLT},
    {"INP", &CCommand::fntAsmINP},
    {"OUT", &CCommand::fntAsmOUT},
    {"SKI", &CCommand::fntAsmSKI},
    {"SKO", &CCommand::fntAsmSKO},
```

```
        {"ION", &CCommand::fntAsmION},
        {"IOF", &CCommand::fntAsmIOF}
    };
```

코드 A4-7. non_MRI 테이블 검색

```
    intIndex = 0;
    while(intIndex < NON_MIR_NUMBER)
    {
        pCommand = &cmdNonMriTB[intIndex];
        if(strLineCode.Find(pCommand->szCommandName) >= 0)
        {
            if(!CheckError(strLineCode, pCommand->szCommandName))
                return FALSE;
            return (this->*(pCommand->pExecute))(strLineCode);
            //return TRUE;
        }
        intIndex++;
    }
```

2.5. 주소 심볼 테이블

다른 테이블과는 달리 주소 심볼 테이블은 미리 정의될 수 없으며 1st 패스 동안에 생성된다.

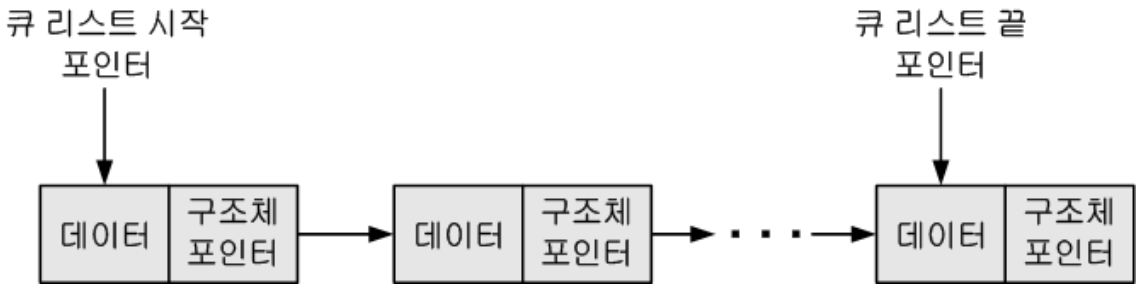


그림 A4-1. 주소 심볼 테이블의 큐 리스트

주소 심볼 테이블은 그림 A4-1 과 같이 큐 리스트에 저장된다. 큐 리스트의 각 리스트는 코드 A4-8 의 구조체와 같이 정의된다.

코드 A4-8. 주소 심볼 테이블 구조체

```
typedef struct tagSymbol
{
    DWORD       dwLc;
    char        strSymBol[10];
    struct      tagSymbol *ptsNext;
}SYMBOL,*PSYMBOL;
```

위 구조체의 각 필드는 다음과 같다.

- *Location counter(dwLc)* : 주소 심볼의 주소를 저장.

- *String(strSymBol)* : 주소 심볼의 이름을 저장.
- *Pointer (tagSymbol * ptsNext)*: 큐 리스트에서 다음 항목을 포인트

큐잉 리스트 ***First*** 구조체로 시작하고 ***Last*** 구조체로 끝나는 다수의 구조체로 구성된다. 1st 패스 동안 코드 라인이 컴마로 구분되는 레이블을 포함하면 어셈블러는 새로운 구조체를 선언하고 *Location counter* 필드에 LC 값을 할당하며, *String* 필드에 라인의 주소 심볼을 할당한다. 그리고 이 구조체를 큐잉 리스트에 추가한 후 리스트의 ***Last*** 구조체로 한다. 코드 A4-9 는 새로운 주소 심볼 테이블을 생성하는지를 보여준다.

코드 A4-9. 새로운 주소 심볼 테이블 생성

```
while(i < strTemp.GetLength())
{
        if(strTemp.GetAt(i) == ',')
        {
                strTemp = strTemp.Left(i);
                strTemp.TrimRight();
                if(m_ptrFirstSymbol == NULL)
                {
                        // Create Symbol symol table
                        PSYMBOL pSymbol;
                        pSymbol = (SYMBOL*)malloc(sizeof(SYMBOL));
                        pSymbol->dwLc = m_dwLC;
                        //pSymbol->dwOrg = m_dwOrg;
                        strcpy(pSymbol->strSymBol, strTemp);
                        pSymbol->ptsNext = NULL;
                        m_ptrFirstSymbol = pSymbol;
                        m_ptrLastSymbol = pSymbol;
                }
                else{
                        // Add new Symbol symbol to symbol Symbol
                        PSYMBOL pSymbol;
                        pSymbol = (SYMBOL*)malloc(sizeof(SYMBOL));
                        pSymbol->dwLc = m_dwLC;
                        //pSymbol->dwOrg = m_dwOrg;
                        strcpy(pSymbol->strSymBol, strTemp);
                        pSymbol->ptsNext = NULL;
                        m_ptrLastSymbol->ptsNext = pSymbol;
                        m_ptrLastSymbol = pSymbol;
                }
                return TRUE;
        }
        i++;
}
```

어셈블러가 주소 심볼의 16 진수 주소를 찾을 때, 큐잉 리스트의 ***First*** 구조체에서부터 String 필드가 코드의 레이블과 일치하는 구조체를 찾을 때까지 연속적으로 검색한다. 이 구조체의 location 필드는 주소 심볼과 동일한 16 진수 값을 계산하는 데 사용된다. 코드 A4-10 는 큐잉 리스트로부터 주소 심볼의 16 진수 값을 계산하는 과정으로 보여준다.

코드 A4-10. 주소 심볼 테이블을 검색

```
ptrSymbol = m_ptrFirstSymbol;
while(ptrSymbol != NULL)
{
        if(strSymbol == ptrSymbol->strSymBol)
                return ptrSymbol->dwLc;
        ptrSymbol = ptrSymbol->ptsNext;
}
return SYMBOL_ERROR;
```

2.6. Main 함수

프로그램을 번역할 때 메인 함수가 호출된다. 그림 A4-2 는 메인 함수의 동작을 보여준다.

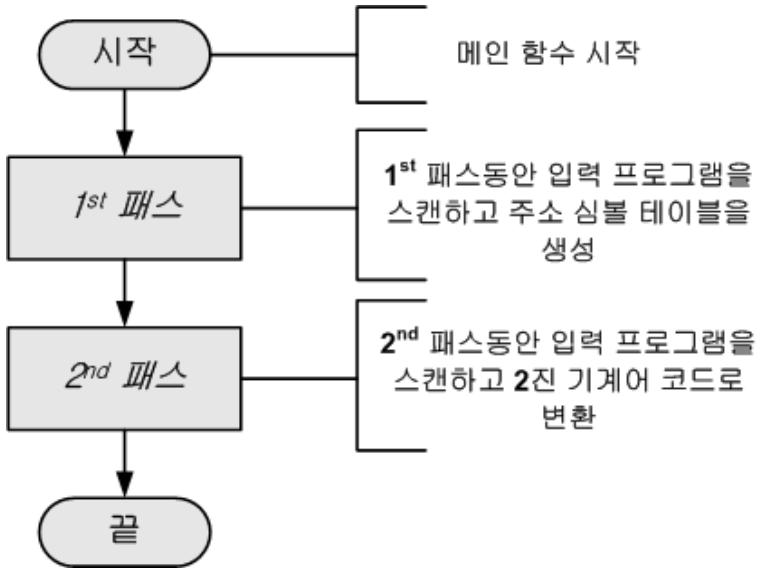


그림 A4-2. 메일 함수의 흐름도

코드 A4-11 은 메인함수의 동작을 구현한 코드를 보여준다. 메인함수의 입력 파라미터는 심볼 프로그램에 대한 파일 경로이다. 메인함수는 두 개의 부 함수(sub function)을 호출하며, 첫 번째는 1st 패스이고, 두 번째는 2nd 패스이다. 메인함수가 실행된 후에는 출력인 2 진 프로그램이 생성된다.

코드 A4-11. 메인함수

```
BOOL CCommand::fntGenerateBnCode(CString strFileName)
{
        BOOL bResult;
        if(strFileName.GetLength() <= 0)
                return FALSE;

        m_strInPutFile = strFileName;
```

```
	m_strOutPutFile = CreateTempFile();

	CreateOuputFile(m_strOutPutFile);


	// Fist Pass
	m_ptrFirstSymbol = NULL;
	m_ptrLastSymbol = NULL;
	fntFirsftPass(strFileName);

	// Second pass
	bResult = fntSecondPass(strFileName);

	End();

	// Free memory of Symbol symbol tabel
	if(m_ptrFirstSymbol != NULL)
			FreeSymbolTable(m_ptrFirstSymbol);

	return bResult;
}
```

2.7. 1st 패스 함수

이 함수는 메인함수에 의해 호출되며, 어셈블러의 1st 패스를 구현하였다. 코드 A4-12 는 1st 패스를 구현한 코드 부분을 보여준다.

코드 A4-12. 1st 패스 함수

```
void CCommand::fntFirsftPass(CString strFileName)
{
	char szLineCode[256];
	FILE* pFile;

	pFile = fopen(strFileName, "rb");

	if(pFile == NULL)
		return;

	m_dwLC = 0;
	m_dwOrg = 0;

	while(!feof(pFile))
	{
		// Scans line of code
		fgets(szLineCode, 256, pFile);

		if(!IsLineCode(szLineCode))
			continue;

		// Check if is lable, add address symbol to address symbols table
		if(!CheckSymbol(szLineCode))
		{
			// Check if content ORG instruction, set value of location counter LC
```

```
                        if(!CheckORG(szLineCode))
                        {
                                if(CheckEnd(szLineCode))
                                {
                                        // Go to second pass
                                        fclose(pFile);
                                        return;
                                }
                        }
                }
                m_dwLC++;

        }
        fclose(pFile);
}
```

2.8. 2nd 패스 함수

이 함수는 메인 함수의 2nd 패스를 처리하기 위해 호출되며, 입력으로 심볼 프로그램을 포함하는 입력의 경로가 사용된다. 이 함수는 순차적으로 각 라인을 읽어서 ParseLineCode() 함수를 호출한다. 코드 A4-13 은 2nd 패스가 구현된 코드를 보여준다.

코드 A4-13. 2nd 패스 함수

```
BOOL CCommand::fntSecondPass(CString strFineName)
{
        FILE* pFile;
        char buffer[256];
        m_dwLC = 0;
        m_dwOrg = 0;
        m_dwErrorLine = 0;
        m_lgLineOuput = -1;

        pFile = fopen(strFineName, "rb");
        if(pFile == NULL)
                return FALSE;

        while(!feof(pFile))
        {
                // Scand line of code
                fgets(buffer, 256, pFile);
                m_dwErrorLine++;
                if(!IsLineCode(buffer))
                        continue;
                // Process the line of code processing
                if(!ParserLineCode(buffer))
                {
                        fclose(pFile);
                        pFile = NULL;
                        return FALSE;
```

```
                }
                m_dwLC++;
        }
        fclose(pFile);
        pFile = NULL;
        return TRUE;
}
```

2.9. ParseLineCode() 함수

이 함수는 2nd 패스 함수에 의해 호출된다. 이 함수는 코드 라인을 구문 분석하기 위해 사용되며 그 다음 2 진 코드로 변환한다. 이 함수는 처리하고 있는 코드 라인의 문자열을 하나의 파라미터로 입력받는다. 그림 A4-3 은 ParseLineCode() 함수의 흐름도를 보여준다.

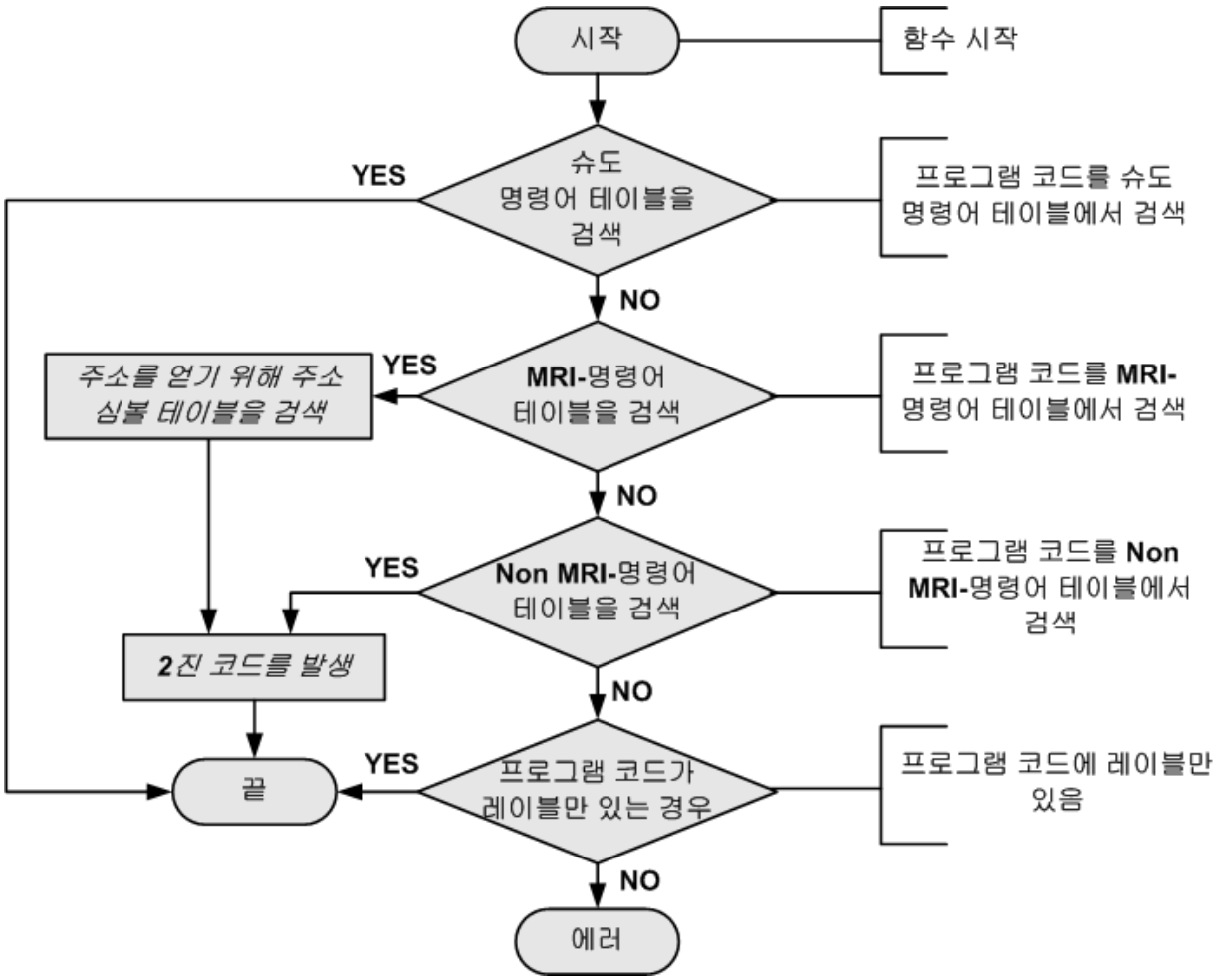


그림 **A4-3.** ParseLineCode() 함수의 흐름도

2.10. 컴퓨터 명령어 부 함수(Computer instruction sub functions)

명령어 라인의 심볼에는 슈도 명령어, 7 개의 MRI 명령어와 18 개의 Non-MRI 명령어를 포함하고 있으므로 컴퓨터 명령어 부 함수(Computer instruction sub functions)는 각 라인의 동작 코드에 해당하는 심볼을 2 진 코드로 변환하도록 설계되었다. 컴퓨터 명령어 부 함수는 ParseLineCode() 함수에 의해 호출되며 코드 A4-14 는 이 함수들에 대한 정의이다.

코드 A4-14. 컴퓨터 부 함수의 정의

```
// Pseudo-instruction
BOOL   fntAsmORG(CString strLineCode);
BOOL   fntAsmEND(CString strLineCode);
BOOL   fntAsmDEC(CString strLineCode);
BOOL   fntAsmHEX(CString strLineCode);

// MRI instruction
BOOL   fntAsmADD(CString strLineCode);
BOOL   fntAsmAND(CString strLineCode);
BOOL   fntAsmLDA(CString strLineCode);
BOOL   fntAsmSTA(CString strLineCode);
BOOL   fntAsmBUN(CString strLineCode);
BOOL   fntAsmBSA(CString strLineCode);
BOOL   fntAsmISZ(CString strLineCode);

// None-MRI instruction
BOOL   fntAsmCLA(CString strLineCode);
BOOL   fntAsmCLE(CString strLineCode);
BOOL   fntAsmCMA(CString strLineCode);
BOOL   fntAsmCME(CString strLineCode);
BOOL   fntAsmCIR(CString strLineCode);
BOOL   fntAsmCIL(CString strLineCode);
BOOL   fntAsmINC(CString strLineCode);
BOOL   fntAsmSPA(CString strLineCode);
BOOL   fntAsmSNA(CString strLineCode);
BOOL   fntAsmSZA(CString strLineCode);
BOOL   fntAsmSZE(CString strLineCode);
BOOL   fntAsmHLT(CString strLineCode);
BOOL   fntAsmINP(CString strLineCode);
BOOL   fntAsmOUT(CString strLineCode);
BOOL   fntAsmSKI(CString strLineCode);
BOOL   fntAsmSKO(CString strLineCode);
BOOL   fntAsmION(CString strLineCode);
BOOL   fntAsmIOF(CString strLineCode);
```