컴퓨터 구조

Processor Simulator Implementation
Term Final Report

과 목 명:	컴 퓨 터 구 조
학 과:	컴퓨터학과
학 번:	2000160184, 9648094
이 름:	이 진 무 , 조 재 현
제 출 일:	2021년 11월 25일 (목)
담당교수:	박 명 순 교수님

Contents

1. Abstract

이 문서는 2003학년도 2학기 컴퓨터구조의 텀 프로젝트의 결과보고를 위한 문서이다. 이번 프로젝트의 내용은 RISC processor의 simulator를 만드는 것이다. 구현과정에서는 구현의 간단화를 위해 32-bit RISC Instruction Set 중에서 일부만 사용할 수 있도록 한다. 이 구현을 통해 RISC processor의 동작 원리를 이해한다.

2. Team Members and Role

정보통신대학 컴퓨터학과 2000160184 이 진 무
- Datapath Design
- Control Hazard Resolution
- Processor Unit
- Documentation

정보통신대학 컴퓨터학과 9648094 조 재 현
- Input File Parser
- Graphical User Interface
- Datapath Viewer
- Documentation

3. Development Environment

Language	: C/C++
Tool	: Visual C++ 6.0 with Microsoft Foundation Class library
OS	: Windows 2000 Professional
Hardware	: Intel Pentium 3 800 MHz RAM 256M
Execution filename	: Procism.exe

4. Requirements Analysis

Processor Simulator는 어셈블리 언어로 작성된 소스 파일을 입력받고, 입력받은 소스파일에서 각 명령어 line이 레지스터의 값을 어떻게 변화시키는지,또 그 명령어가 데이터패스에서는 어떤 관계를 거쳐 실행이 되는지를 알아보기 위해 명령어 line 별로 레지스터의 값들을 화면으로 출력한다. 또한 3단계 Pipeline 구조의 동작 방식을 이해하기 위해 Pipeline 레지스터들의 값을 화면으로 출력해야 한다

4.1 Input

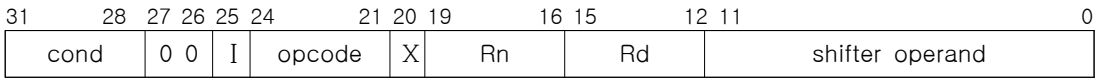
사용자가 메모장과 같은 Editor에서 어셈블리 언어로 된 소스파일을 작성.
소스파일 이름은 “Procsim.s"로 한다.
소스파일이 포함하는 명령어의 종류는 4.2 instruction set으로 한정한다.

4.2 사용하는 Instruction Set

RISC 프로세서의 명령어는 크게 Data 전송 명령어, Data 처리 명령어, 분기(Branch) 명령어로 나눌 수 있다.

4.2.1 Data Processing Instruction

데이터 처리 명령어란 프로그램 중에서 data에 산술연한 혹은 논리연산을 수행하는 명령들을 의미한다. 따라서 데이터 처리 명령을 수행하고 나면 데이터가 변하게 된다.

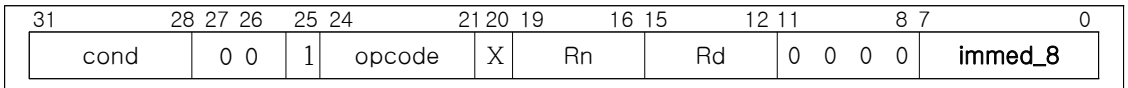


- **<cond>:** 사용하지 않음
- **<I>** : 해당 명령어의 주소 지정 방식을 나타냄
  - Immediate 방식이면 ‘1’
  - Register 방식이면 ‘0’
- **<opcode>** : 해당 명령어의 종류를 표기.
  즉, opcode[24:21]를 보고, 해당 명령어가 'ADD'명령어 인지, 'SUB'명령어 인지 등의 명령어 종류를 구분할 수 있다.
- **<X>** : 사용하지 않음
- **<Rn>** : 소스 레지스터
- **<Rd>** : 목적지 레지스터
- **<shifter operand>** : **주소 지정 방식.**
  주소 지정 방식의 종류에는 아래의 **4.1.1절**과 같은 **3가지 주소 방식**이 존재한다.

Opcode [24:21]	Mnemonic	Operation	명령어 표기법 및 설명	
0010	SUB	Subtract	표기법	**SUB** <Rd>, <Rn>, <shifter operand>
			설명	레지스터 <Rn>에서 <shifter operand>를 뺀 후, 그 결과 값을 레지스터 <Rd>에 저장
0100	ADD	Add	표기법	**ADD** <Rd>, <Rn>, <shifter operand>
			설명	레지스터 <Rn>과 <shifter operand>를 더한 후, 그 결과 값을 레지스터 <Rd>에 저장
1010	CMP	Compare	표기법	**CMP** <Rn>, <shifter operand>
			설명	레지스터 <Rn>의 값에서 <shifter operand>를 뺀 후, 결과값을 condition flag(N,Z,C,V)에 업데이트
1101	MOV	Move	표기법	**MOV** <Rd>, <shifter operand>
			설명	<shifter operand>의 값을 레지스터<Rd>로 옮김

Address mode of Data Processing Instruction

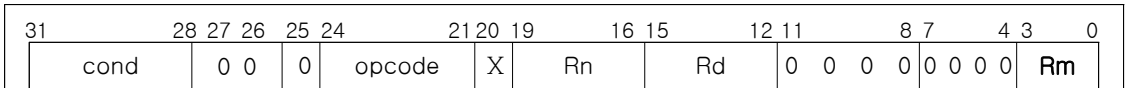
① **#<immed_8>** ; 즉시주소 지정방식



표기법	ADD Rd, Rn, **#<immed_8>**
사용예	ADD r1, r2, **#0xA**
설 명	레지스터 r2의 값에 16진수 값 'A'를 더한 후, 그 결과 값을 레지스터 r1에 저장

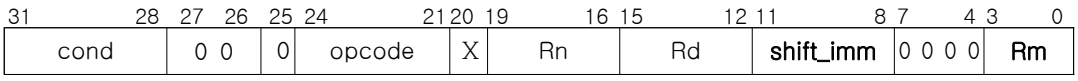
- **<immed_8>** : 값의 범위는 8비트 크기이며,
  값의 표현은 10진수와 16진수를 사용한다. '+', '-' 기호는 10진수만 사용 한다.
  즉, 양수의 경우는 (#+10), (#10) 둘 다 허용하며, 음수의 경우는 (#-10)과 같이 사용한다.
  그리고 16진수는 Signed number로 하며, 표현은 상수 앞에 "0x"기호를 붙인다.
- 연산 수행 시, 레지스터의 값은 32bit 인데 반해, <immed_8>의 값은 8bit 이므로,
  <immed_8>의 값을 32bit 값으로 확장하여 연산을 수행한다.

② **<Rm>** ; 레지스터(Register) 주소지정방식



표기법	ADD Rd, Rn, **Rm**
사용예	ADD r1, r2, **r3**
설 명	레지스터 r2와 레지스터 r3의 값을 더한 후, 그 결과 값을 레지스터 r1에 저장

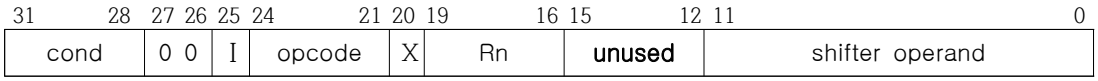
③ **<Rm>, LSL #<shift_imm>** ; Logical shift left by immediate



표기법	ADD Rd, Rn, **Rm, LSL #<shift_imm>**
사용예	ADD r1, r2, **r3, LSL #0xA**
설 명	레지스터 r3의 값을 왼쪽으로 16진수 값 'A'만큼 shift한 후, 그 값을 레지스터 r2에 더하여 그 결과 값을 레지스터 r1에 저장

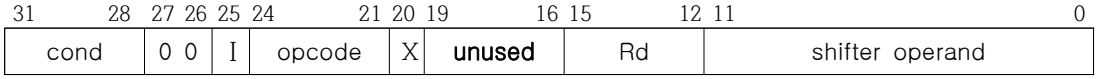
- **<Rm>** : shift될 레지스터.
- **<shift_imm>** : shift할 비트수를 명시**(양수)**. 범위는 4비트의 크기이다.

▶ CMP



- **<unused>** : 사용하지 않음
- **<shifter operand>** : p.6 <그림4>참조

▶ MOV



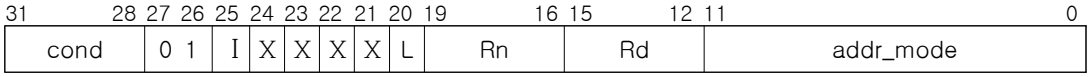
- **<unused>** : 사용하지 않음
- **<shifter operand>** : p.6 <그림4>참조

4.2.2 Data Transfer Instruction

Data 전송 명령어란, 데이터 값을 레지스터로부터 메모리로 저장하거나, 메모리로부터 레지스터에 데이터 값을 저장하는 작업을 수행하는 명령어를 말한다.

Opcode [27:26]	Mnemonic	Operation	명령어 표기법 및 설명	
01 (L bit=1)	LDR	Load Register	표기법	**LDR** <Rd>, <Rn>, **<addr_mode>**
			설명	레지스터 <Rn>에 담겨진 base address에서 메모리 주소를 <address_mode> 만큼 증가한 후, 해당 메모리 주소로부터 1word(=32bit) 만큼의 데이터를 Load하여 레지스터 <Rd>에 저장
01 (L bit=0)	STR	Store Register	표기법	**STR** <Rd>, <Rn>, **<addr_mode>**
			설명	레지스터 <Rn>에 담겨진 base address에서 메모리 주소를 <addr_mode>만큼 증가하여 얻어진 메모리 주소에 레지스터 <Rd>로부터 얻어진 1word(=32bit)의 데이터를 저장.

Data 전송 명령어에 대한 명령어 포맷 및 사용법에 대해서는 아래의 <그림5>를 참조한다.

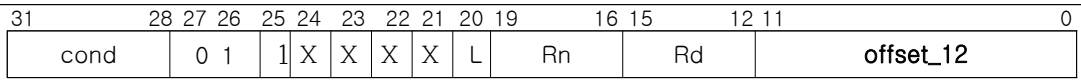


31 28	27 26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0 1	I	X	X	X	X	L	Rn	Rd	addr_mode

- **<cond>** : 사용하지 않음
- **<I>** : 해당 명령어의 주소 지정 방식을 나타냄
  - Immediate 방식이면 '1'
  - Register 방식이며 '0'
- **<X>** : "*Unused*" bit (사용하지 않음)
- **<L>** : 명령어가 'Load'이면 '1', 'Store'이면 '0'
- **<Rn>** : Base 레지스터
- **<Rd>** : 목적지 레지스터
- **<addr_mode>** : 주소 지정 방식.
  주소 지정 방식의 종류에는 아래의 **4.2.1절**과 같은 **2가지 주소 방식**이 존재한다.

Data 전송 명령어의 주소 지정 방식

① **<Rn>, #<offset_12>** ; Immediate Offset 방식



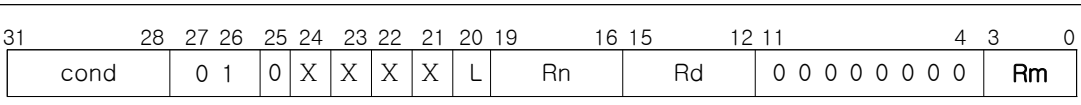
31 28	27 26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0 1	1	X	X	X	X	L	Rn	Rd	offset_12

표기법	LDR Rd, **Rn, #<offset_12>**
사용예	LDR r1, **r2, #10**
설 명	레지스터 r2에 담겨진 base address에서 메모리 address를 **'10'만큼 증가**한 후, 해당 메모리 주소로부터 1word(=32bit) 만큼의 데이터를 Load하여 레지스터 r1에 저장

▪ **<offset_12>** : 값의 범위는 12비트의 크기이며,
값의 표현은 10진수와 16진수를 사용한다. '+', '-' 기호는 10진수만 사용 한다.
즉, 양수의 경우는 (#+10), (#10) 둘 다 허용하며, 음수의 경우는 (#-10)과 같이 사용한다.
그리고 16진수는 Signed number로 하며, 표현은 상수 앞에 "0x"기호를 붙인다.

표기법	LDR Rd, **Rn, #<offset_12>**
사용예	LDR r1, **r2, #-10**
설 명	레지스터 r2에 담겨진 base address에서 메모리 address를 **'10'만큼 감소**시킨 후, 해당 메모리 주소로부터 1word(=32bit) 만큼의 데이터를 Load하여 레지스터 r1에 저장

② **<Rn>, <Rm>** ; Register Offset 방식



31 28	27 26	25	24	23	22	21	20	19 16	15 12	11 4	3 0
cond	0 1	0	X	X	X	X	L	Rn	Rd	0 0 0 0 0 0 0 0	Rm

표기법	LDR Rd, **Rn, Rm**
사용예	LDR r1, **r2, r3**
설 명	레지스터 r2에 담겨진 base address에서 레지스터 r3에 있는 offset만큼 메모리 address를 증가한 후, 해당 메모리 주소로부터 1word(=32bit) 만큼의 데이터를 Load하여 레지스터 r1에 저장

4.2.3 Branch Instruction

분기(Branch) 명령어란, jump나 subroutine call과 같이 프로그램 수행 중 PC(프로그램 카운터)를 변화시켜 프로그램의 흐름을 변화시키는 명령어를 말한다.

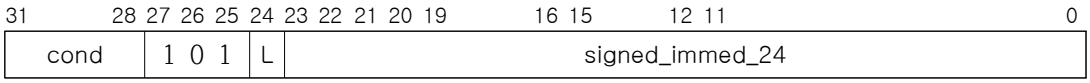
Opcode [27:25]	Mnemonic	Operation	명령어 표기법 및 설명	
101 (Lbit=0)	B	Branch	표기법	B <signed_immed_24>
			설명	<signed_immed_24>로 분기 (단순분기)
101 (Lbit=1)	BL	Branch and Link	표기법	BL <signed_immed_24>
			설명	return address를 r14에 저장하고, <signed_immed_24>로 분기 (서브루틴 콜)
101	B{L}EQ	Branch Equal	표기법	B{L}EQ <signed_immed_24>
			설명	만약 Z(zero) flag가 '1'이면 <signed_immed_24>로 분기 ("L" 옵션 사용시 return address를 r14에 저장)
101	B{L}NE	Branch Not Equal	표기법	B{L}NE <signed_immed_24>
			설명	만약 Z(zero) flag가 '0'이면 <signed_immed_24>로 분기 ("L" 옵션 사용시 return address를 r14에 저장)

Opcode [31:28]	Mnemonic Extention	의미	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not Equal	Z clear

(무조건 분기의 경우 Opcode 1110)

Branch 명령어에 대한 명령어 포맷 및 사용법에 대해서는 아래의 <그림6>을 참조한다.

Branch 명령어의 주소 지정 방식

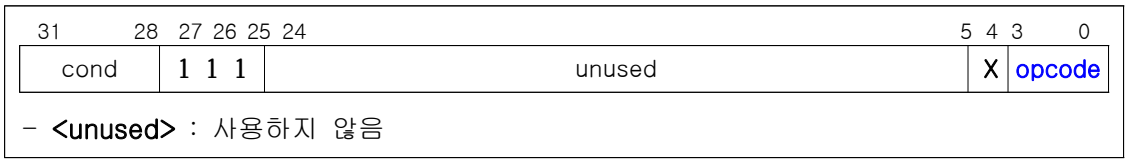


- **<cond>** : 컨디션 플래그(N,Z,V,C). **(p4. <표1> 참조)**
  실행시 분기여부를 결정하는 조건으로 사용된다
- **<L>** :
  - 'L'옵션을 사용하지 않으면(**L bit='0'**) : 단순분기
  - 'L'옵션을 사용하면(**L bit='1'**) : branch 명령어 수행 후의 return address를 link register(R14)에 저장하고 분기문을 수행.
- **<signed_immed_24>** : target address를 나타냄 **(label 사용을 허용 함)**.
  target address는 다음과 같이 얻어진다.
  ① 24bit address(signed)를 32bit address로 확장
  ② 2bit 만큼 Left shift
  ③ PC = PC + 8

표기법	B{L} {<cond>} **<signed_immed_24>**
사용예	B start ; 항상 start 로 분기 CMP r1,#0 ; r1 레지스터와 '0'을 비교 BEQ end ; 만약, condition flag 의 zero flag 가 '1'이면 end로 분기하고, 그렇지 않으면 다음 명령어를 수행
설 명	위의 예제에서 첫째 줄은 항상 start라는 주소로 분기하지만, BEQ는 앞의 CMP라는 인스트럭션의 결과에 따라서 Z 플래그가 '1'이면 end로 분기하고, 그렇지 않으면 다음 인스트럭션을 실행한다. - <cond>(조건)은 아래의 <표7>에 표기되어 있는"EQ","NE"만을 사용.

표기법	B{L} {<cond>} **<label>**
사용예	B start ; 항상 start 로 분기 start Add r1,r2,r3 ; r1=r2+r3

4.2.4 기타 명령어



- <unused> : 사용하지 않음

<그림 7. 기타 명령어의 명령어 형식>

◆ HALT

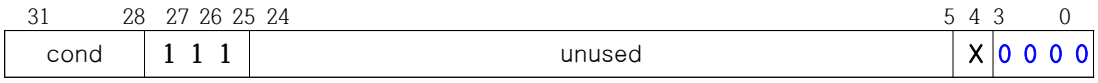


표 기 법	HALT
사 용 예	HALT
설 명	프로그램 실행을 중지하고 프로그램을 빠져나감

◆ No-op

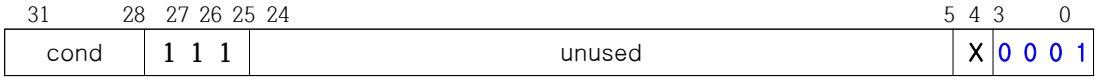


표 기 법	NOP
사 용 예	NOP
설 명	아무 작업도 수행하지 않음 (단순히 한 clock cycle만 차지 함.)

4.5 Pseudo Instruction

32bit 수를 Load하기에 편리한 pseudo 명령어를 정의한다.

◆ LDR Rd, =<const>

LDR Rd, =<const>

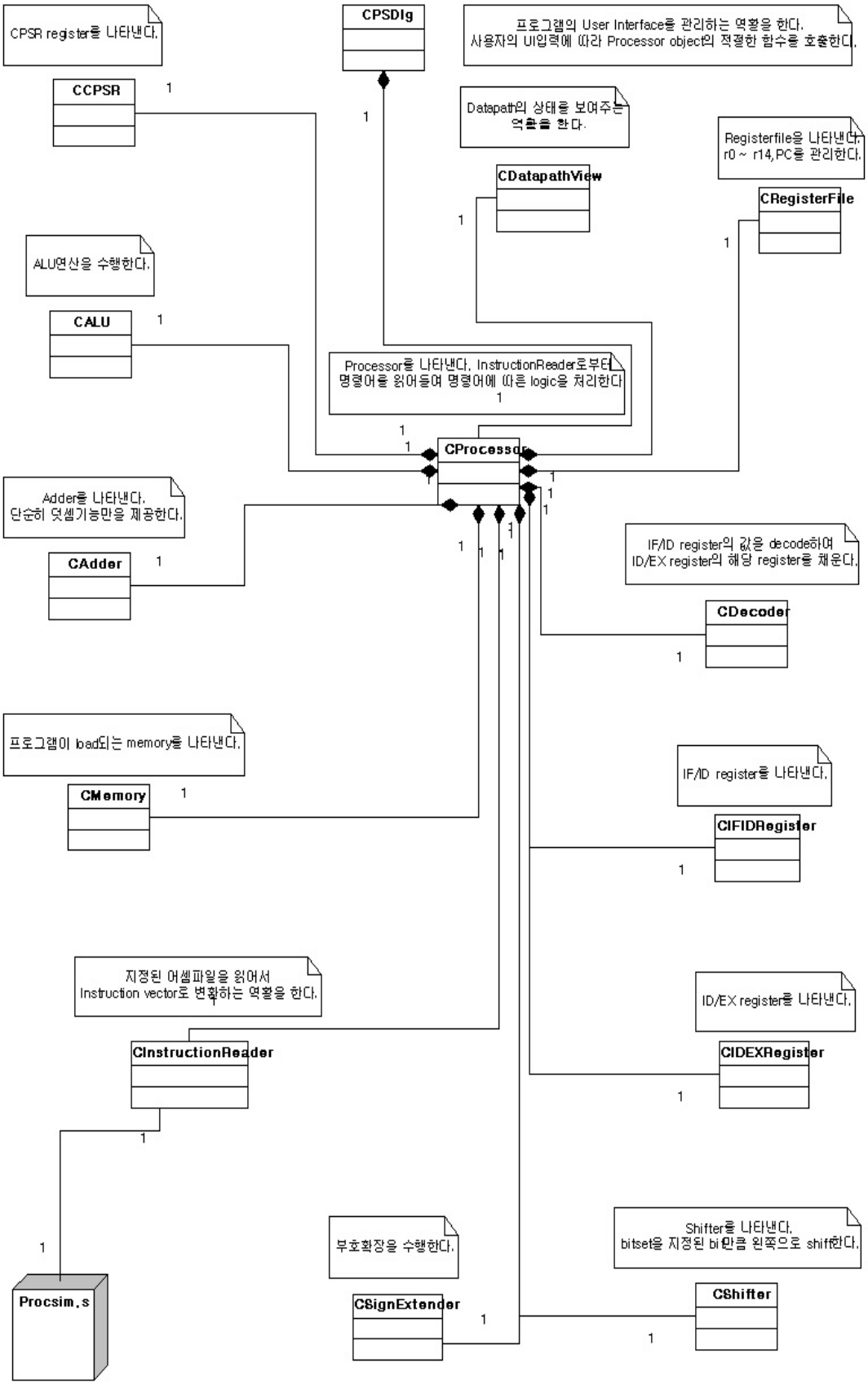
- <const> : 32bit 수 이며, signed number 이다.

4.6 Operation

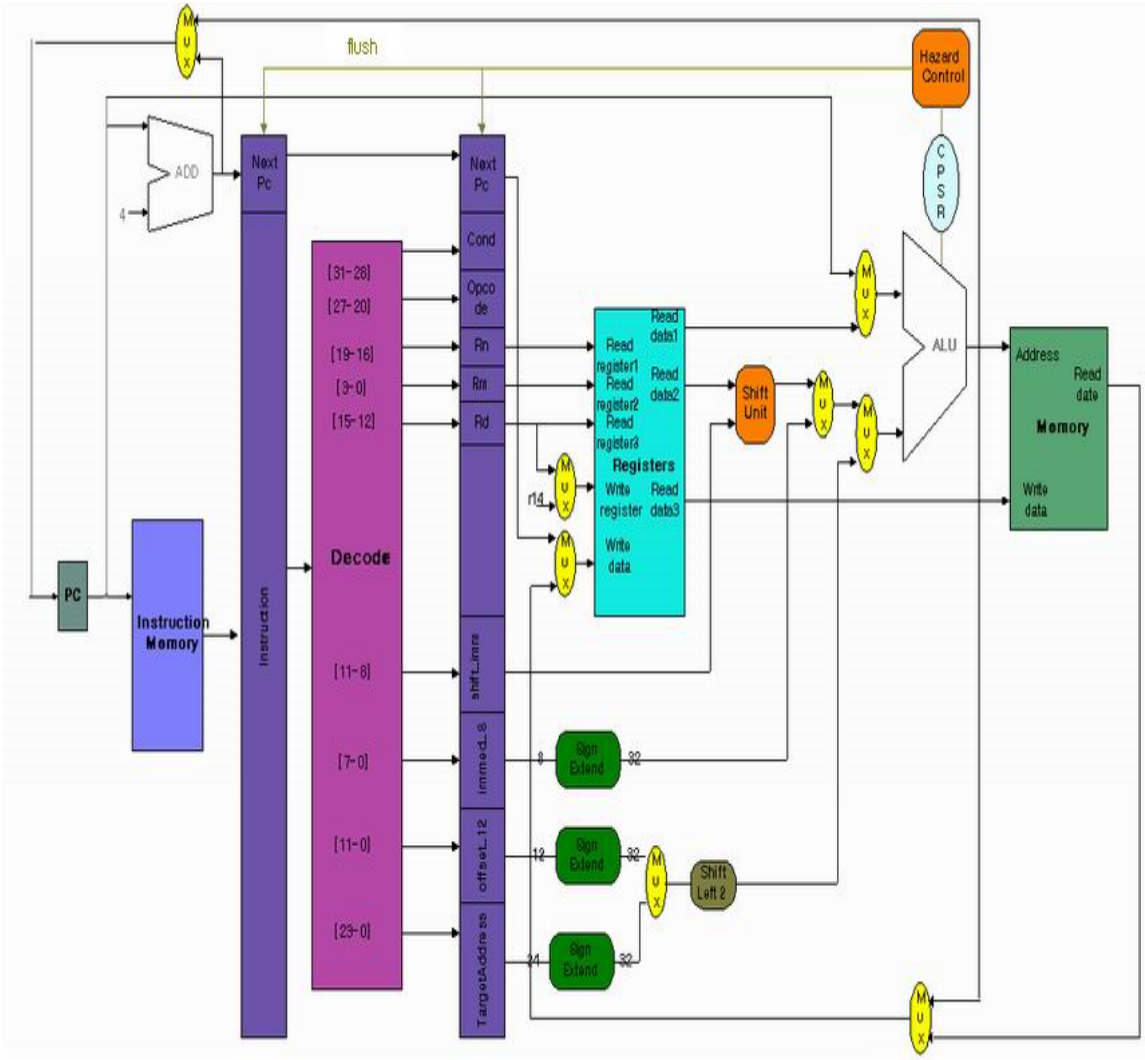
소스파일의 어셈블러 명령어를 clock cycle 단위로 수행하면서 이에 따른 범용 레지스터 값
들과 Pipeline 레지스터 값들을 화면에 출력하고, 이러한 명령어들이 거치는 Data Path를 출력한다.

① 소스파일에서의 각 명령어 라인에 대해 “TRACE" 버튼을 누를 때 마다 하나의 clock cycle씩 증가시키며, 그 명령어가 Data Path에서는 어떠한 관계를 거쳐 실행이 되는지를 출
력.
② 각 명령어 라인에 대한 15개의 범용 레지스터들(r0~r14)의 값과 CPSR(Current Program Status Register), PC 값을 출력.
③clock-cycle 증가에 따른 Pipeline 레지스터의 값을 화면에 출력
④소스파일이 변경되었거나 처음부터 다시 시작하려면 ①번 과정부터 다시 수행.

5. Class Diagram



6. Datapath Design



1단계는 fetch단계이다. instruction memory로부터 현재 PC값에 해당하는 instruction을 가져온다. 이 가져온 instruction을 IF/ID pipeline register의 instruction register에 저장한다. 분기 명령어의 link option사용시에 다음 PC값을 r14 register에 저장할 필요가 있는데 이를 위해서 IF/ID register의 NextPC register에 현재 PC +4한 값을 저장한다.

2단계는 decode단계이다. execution단계에서 처리를 간단히 하기 위해서 IF/ID register의 instruction을 가져와서 decoder를 통하여 decode한후 다음과 같이 세분화 하여 ID/EX register에 저장한다.

nextPC - 다음 PC값 link시에 사용
cond - [31-28]

OPCode - [27-20]
Rn - [19-16]
Rd - [15-12]
Rm - [3-0]
immed_8 - [7-0]
shift_imm - [11-8]
offset_12 - [11-0]
targetAddress - [23-0]

3단계는 execution단계이다. ID/EX register로부터 intruction을 가져와서 명령어에 따른 처리를 수행한다. 이때 분기가 발생시에 flush line을 활성화 하여 IF/ID register와 ID/EX register를 NOP와 같은 값을 가지도록 설정한다.

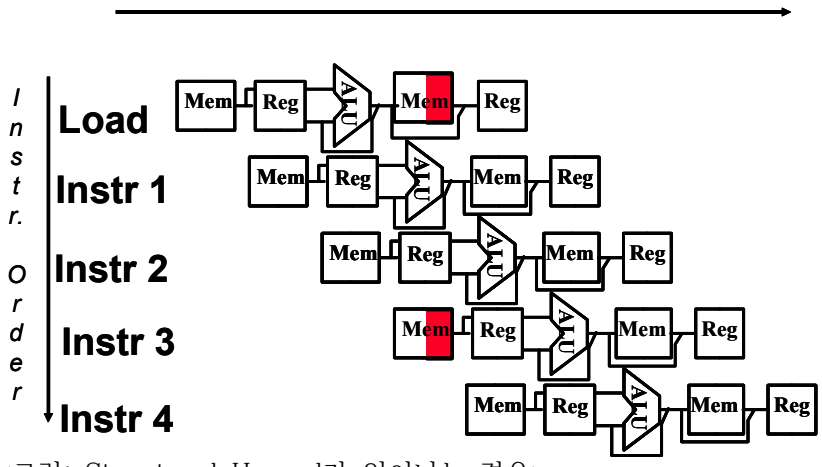
7. Hazard의 처리방안

Pipelining을 사용할 때, 바로 다음 clock cycle에 다음 instruction을 실행시킬 수 없는 경우가 생기게 되는데, 이런 상황을 Hazard라고 한다. Hazard에는 3가지 종류가 있는데, 여기서 각각을 설명하고 해결방안에 대해서 생각해 보자.

7.1. Structural Hazards

한 clock cycle에 동시에 수행하고 싶은 instruction들의 조합을 hardware자체가 지원하지 못하는 경우를 Structural Hazard라 한다. 두 개의 instruction이 동시에 같은 resource를 access해야 할 때 일어나는 hazard이다.

예를 들어, 아래와 같이 5단계 pipelining을 이용한 data path에 instruction들이 수행되고 있다고 하자. 이때, 4번째 cycle에서 Load instruction과 Instr3가 동시에 Memory를 접근하게 된다. 이런 경우는, instruction과 data를 위한 memory를 개별적으로 만듦으로써 해결할 수 있다.



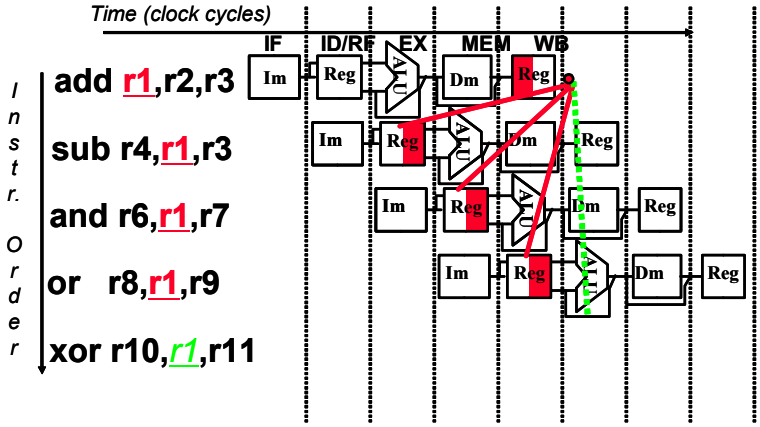
<그림: Structural Hazard가 일어나는 경우>

이번 프로젝트에서는 hardware가 완벽하게 지원되고 instruction memory와 data memory를 개별적으로 두는 경우이기 때문에, 이 종류의 hazard는 일어나지 않는다.

7.2. Data Hazards

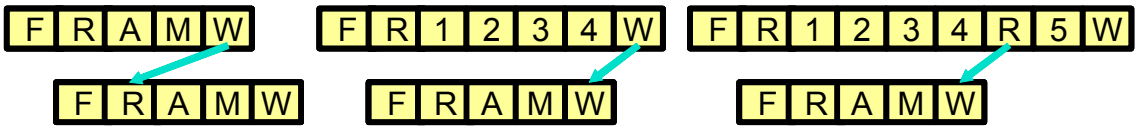
한 instruction의 결과가 다음 instruction의 입력으로 쓰이는 경우가 있다. 이때 pipelining을 사용하면 이전 instruction의 결과가 아직 register나 memory에 기록되기 전에 다음 instruction이 실행되는 경우에 Data Hazard가 일어났다고 한다.

예를 들어, 다음 그림과 같은 instruction들이 수행되고 있다고 하자. 이때 add instruction이 완전히 끝나서 register r1에 써지기 전까지는 sub instruction을 수행해서는 안 된다.



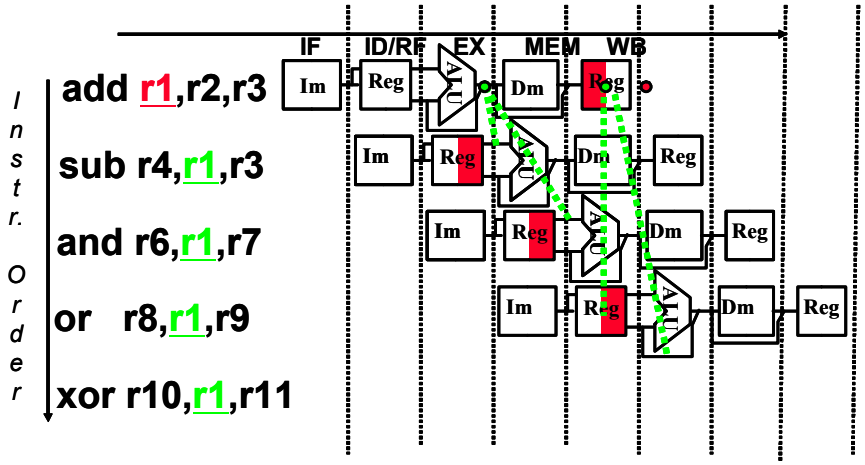
<그림: Data Hazard가 일어나는 경우>

Data Hazard의 종류로는 RAW(Read After Write), WAW(Write After Write), 그리고 WAR(Write After Read)가 있다. 다음 그림이 각각의 경우들을 설명하고 있다.



<그림: Data Hazard의 종류들>

Data hazard를 해결하기 위한 두 가지 방법이 있다. 첫 번째는 그냥 위의 instruction이 끝나서 다음 instruction이 수행 가능할 때까지 기다리는 것(Stall)이다. 두 번째 방법은 "Forwarding(Bypassing)"이라는 기법을 사용한다. 위의 그림에서 두 번째 instruction sub가 실행되기 위해서는 register r1의 값이 필요하다. 이때, 첫 번째 명령어의 결과가 Register File에 저장되기까지 기다리는 것이 아니라, EX cycle에서의 ALU결과를 직접 가져와서 연산을 하면 아무런 delay가 없이 정상적인 instruction을 수행할 수 있다.



<그림: Data Hazard를 해결하기 위한 Forwarding 기법>

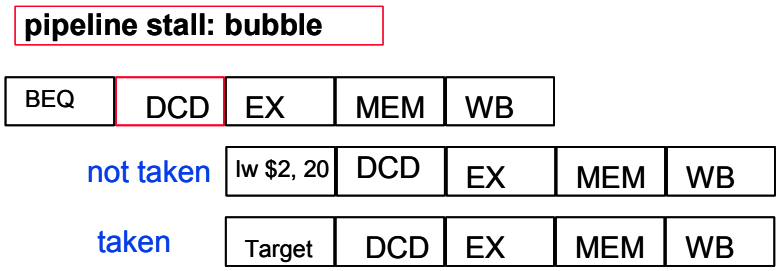
이번 프로젝트에서는 3단계 파이프라인 중 마지막 cycle에서 모든 수행 결과가 atomically 처리되기 때문에 data hazard가 일어나지 않는다.

7.3. Control Hazards

한 instruction의 수행결과로 어떤 data flow에 결정을 내리는 경우가 있다. 이때, 이후 instruction들이 앞으로의 결과를 모르는 경우에 Control Hazard가 일어난다. 예를 들면, 분기명령어가 들어왔을 때에, 분기의 여부에 따라 바로 다음에 오는 instruction이 수행될 수도 있고, jump가 일어난 후의 instruction이 수행되어야 하는 경우도 생긴다.

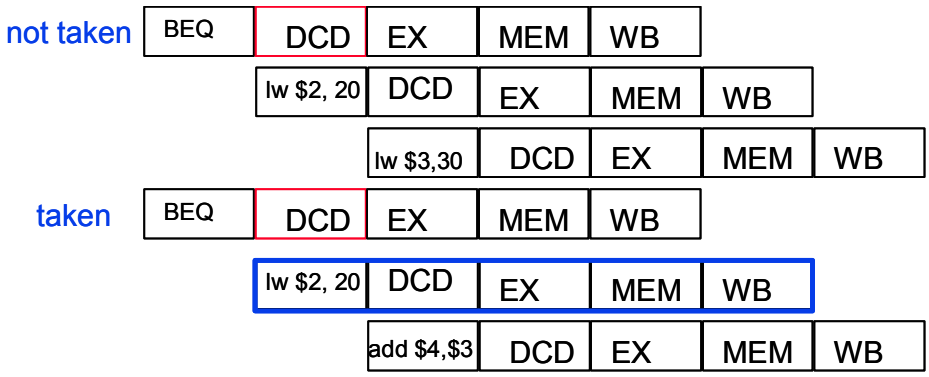
이번 프로젝트에서 생길 수 있는 hazard가 이 control hazard이다. 그러므로 이것을 해결하기 위한 방안이 필요하다. 크게 3가지의 해결책을 생각할 수 있다.

1) Stalling: wait until decision is clear! 즉, delay가 될지 여부를 확인할 때가지 data flow를 잠시 멈추는 것이다. 이 방법을 쓰는 경우는 branch명령어가 나올 때마다 2 clock을 낭비하는 결과를 가져온다.



<그림: Delay Hazard를 해결하기 위한 Stalling 방법>

2) Prediction: delay 여부를 미리 예측해서 data flow를 계속 해나가는 것이다.
2.1) 무조건 분기가 일어나지 않는다고 가정을 한다.
2.2) 무조건 분기가 일어난다고 가정한다.



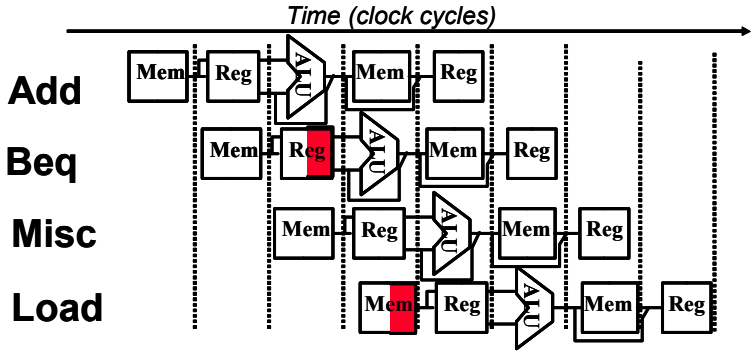
<그림: Delay Hazard를 해결하기 위한 Prediction 방법>

이 방법을 사용하는 경우는 예상이 맞았다면, 낭비되는 cycle이 없고, 예상이 틀렸다면, pipeline안에 들어와 있는 정보들을 flush시켜야 한다.

2.3) Dynamic Branch Prediction: 이전에 분기가 일어났는지 여부를 저장하여 앞으로도 같은 분기결과가 나올 것이라고 예상한다. 이 방법을 사용하기 위해서는 이전에 해당 branch instruction이 최근에 Taken했는지 Not Taken했는지 기록하는 Branch Prediction Buffer또는 Branch History Table 과 같은 저장 공간이 필요하다.

<그림: Delay Hazard를 해결하기 위한 Dynamic Branch Prediction>

3) Delayed Branch: 소스코드를 compile할 때에 data의 dependancy를 보고 분기의 여부와 관계가 없는 instruction을 분기명령어 바로 다음에 수행하도록 한다. 그리고 나서 분기의 결과가 나온 후에 분기결과와 관계가 있는 instruction을 다소 늦게 실행한다. 이 경우는 compiler에 많은 complexity를 요하게 된다.



<그림: Branch Hazard를 해결하기 위한 Delayed Branch 방법>

이번 프로젝트에서 유일하게 일어날 수 있는 Hazard가 이 Control Hazard이다. 우리는 이 hazard의 해결방안으로 Taken/Not Taken여부를 먼저 예상하는 Prediction 방법을 사용하였다. 프로젝트에서는 3-way pipelining을 사용하였다. 이때, Execution Cycle이 끝난 후에나 branch할 address를 계산할 수 있으므로, 분기를 Taken한다고 가정하든지, 안한다고 가정하든지 마찬가지로 두 번의 NO-OP operation들이 필수적이다.

8. Implementation

8.1 구현 아이디어

데이터 패스에 등장하는 모든 unit들(Mux제외)을 class로 만들어 내부 로직까지 시뮬레이션의 의미에 맞게 처리하고자 하였다. 따라서 모든 Unit class들은 입력과 출력 모두를 실제와 같이 bit로 한다.

따라서 모든 Unit class들의 입출력 함수는 다음과 같이 argument와 return value로 bitset을 사용하는 형식을 가진다.
bitset<n> Read(cpnst bitset<n>& addressBIts);
읽기 함수
Write(const bitset<n>& dataBits);
쓰기 함수

이번 프로젝트에서는 bit단위로 설정하고 값을 얻어야 하는 경우가 많은데 비트 연산은 C++ 표준 라이브러리에 속한 bitset type을 사용하였다. bitset<32>로 선언하면 32bit로 이루어진 bit배열을 얻게 된다. 특정 비트를 읽고 쓰는 연산이 많은데 이러한 것은 bit의 string(예:“1010”) 으로 설정하고 읽어올 수 있도록 처리 하였다.

void SetBits(const bitset<32> & bits,size_t idx,const string& strBits);
해당 bitset의 idx번부터 strBit에 해당하는 문자열 bit string을 설정한다.
사용예) SetBits(bits,27,"101");
bit의 27번 bit부터 101로 설절한다.

string GetBits(const bitset<32>& bits,size_t idx,int num);
해당 bitset의 idx번부터 num개의 bit를 문자열 bit string으로 얻는다.
사용예) GetBits(bits,27,3) bit의 27번부터 3개의 bit를 “101”로 얻는다.

8.2 Module Design

8.2.1. Key Class 설명
(key class의 구현에 관한 세부 사항은 뒤에서 설명)

CInstructionReader

- 지정된 어셈파일을 읽어서 machine code로 변환하는 역할을 한다.
인터페이스로 다음의 함수를 제공한다.
int GetInstruction(LPCSTR strFilePath,vector<CInstruction>& insVec)
source파일명을 주고 Instruction vector를 구한다. return 값은 instruction의 시작 index 이다.

CProcessor

-이번 프로젝트에서 설계한 processor를 나타낸다.

프로그램을 load하고 fetch,decode,execute의 파이프라인 각 단계를 처리한다. 외부 UI와 interface를 수행하는 역할을 하므로 UI module에서 버튼을 눌렀을 때 수행 될 load(), trace(), reset()함수를 제공한다. 실제 프로세서가 하는 것처럼 회로 unit의 역할을 하는 class의 instance를 사용하여 명령어 처리 logic을 구현하고 CDatapathView instance를 통하여 결과 datapath그림을 출력한다.

trace버튼을 눌렀을 때 각 파이프라인의 단계가 모두 동시에 실행되어야 하는 것을 시뮬레이션 해야 하기 때문에 trace()함수에서 execute - decode - fetch의 순으로 순차적으로 각 단계의 함수를 모두 실행한다. 하지만 이와 같은 순차적인 처리에서 분기예측이 실패시에 NOP가 decode와 execute단계에 각각 하나씩 두개가 들어가야 하기 때문에 execute 단계에서 분기가 되었을 때 분기가 되었음을 알리는 bJump flag를 true로 설정하고 targetAddress를 분기가 될 instruction주소로 설정한다. 그 후에 fetch단계에서 현재 pc값으로 명령어를 가져오고 bJump flag가 true이면 그 때 pc값을 targetAddress로 설정한다.

void LoadProgram(LPCSTR strFilePath);
: Program을 load한다.

void GetSource(string& strSource);
: source code와 변환된 machine code를 구한다.

void Trace();
: Trace button입력시 처리를 한다.

void Reset();
: Reset button입력시 처리를 한다.

void Draw(CDC* pDC);
: UI로부터 dc를 받아서 processor의register들의 state를 구한다.

void GetGRegState(string& strState);
: register file의 상태를 구한다.

void GetIFIDRegState(string& strState);
: IF/ID pipeline register의 상태를 구한다.

void GetIDEXRegState(string& strState);
: ID/EX pipeline register의 상태를 구한다.

void GetCPSRRegState(string& strState);
: CPSR register의 상태를 구한다.

CDatapathView

- 설정된 datapath 비트맵을 화면에 보여주고 선과 문자열을 그리는 함수를 제공한다. 더블 버퍼링 기법을 사용하여 그림을 다시 그릴 때 화면이 깜빡거리는 현상을 해결 하였다.

Datapath위에 그려줄 line을 CLine이라는 class로 정의하였다. CLine class는 선을 그릴 수 있는 (x1,y1),(x2,y2)의 좌표를 저장하는 역할을 한다. 명령어에 따라 발생할 수 있는 datapath의 의미 있는 라인의 집합을 path로 정의하고 CLine의 vector로 구현하였다. 이 path는 20가지 경우로 나누어지는데 명령어를 처리하면서 어떠한 path를 그려야 하는지를 결정해 나가면서 그릴 path의 번호들을 path number vector에 저장하게 된다. 나중에 명령어에 따라서 저장된 path number vector의 element인 path를 순차적으로 그리면 명령어에 따른 데이터 패스 라인을 그릴 수 있다.

다음과 같은 인터페이스를 제공한다.
void SetBitmap(UINT nBitmapID);
: bitmap을 설정한다.

void Draw(CDC* pDC,int x,int y);
: (x,y)의 위치에 데이터패스를 그린다.

void TextOut(int x,int y,const string& strText,COLORREF crColor);
void TextOut(int x,int y,const DisplayInfo& displayInfo);
: x,y의 위치에 crColor의 색으로 문자열을 출력한다.

void DrawLines(const vector<int>& pathNumVec,COLORREF crColor);
: path num에 해당하는 path를 해당색으로 그린다.

void Clear();
: 화면을 clear한다.

8.2.2 Unit class
(datapath에서 사용되는 unit의 class)설명

CMemory

- 프로그램이 load되는 memory를 나타낸다. C++표준 라이브러리인 map으로부터 상속 받아 구현하였다. Map은 int형의 주소를 나타내는 key값을 가지고 명령어를 나타내는 bitset<32>를 value로 가진다. (map<int,bitset<32> >)
프로그램을 load하는 memory의 역할을 하며 memory에 다음과 같이 읽고 쓰는 기능을 제공한다.

void WriteData(const bitset<32>& addressBits ,const bitset<32>& bits);
: addressBits에 해당하는 주소에 해당 bits를 쓴다.

bitset<32> ReadData(const bitset<32>& addressBits);
: addressBit에 해당하는 memory에 주소에서 data를 읽는다.

CALU

- ALU를 나타낸다. 두개의 bitset을 받아서 Add 또는 Sub의 연산을 수행하고 그에 따라 CPSR register의 값을 설정한다.
다음과 같은 인터페이스를 제공한다.

void Operate(OPType opType,const bitset<32>& bits1,const bitset<32>& bits2,bitset<32>& retBits);
: OPType에 지정된 연산을 수행한다. 두개의 bitset을 받아서 OPType에 지정된 연산을 수행하고 결과 bitset을 돌려준다.
(OPType은 덧셈일때 ALU_ADD,뺄셈일때 ALU_SUB를 지정한다.)

CAdder

- Adder를 나타낸다. 단순히 덧셈기능만을 제공한다.
다음과 같은 인터페이스를 제공한다.

void Add(const bitset<32> bits1,const bitset<32>& bits2,bitset<32>& retBits);
두개의 bitset을 받아서 덧셈을 수행하고 결과 bitset을 돌려준다.

CShifter

- Shifter를 나타낸다. bitset을 지정된 bit만큼 왼쪽으로 shift한다.
다음과 같은 인터페이스를 제공한다.

void ShiftLeft(bitset<32>& bits,bitset<4> shiftBits)
: 지정된 bitset을 shiftBits만큼 왼쪽으로 shift한다.

CSignExtender

- 부호 확장을 수행하는 역할을 한다. 8,12,24bit에서 32bit로 부호 확장하는 함수를 제공한다.

다음과 같은 인터페이스를 제공한다.

void SignExtend(const bitset<n>& bits1,bitset<32>& bits2)
: bits1을 32bit인 bits2로 부호확장한다.

CCPSR

- CPSR register를 나타낸다.

다음과 같은 인터페이스를 제공한다.

void GetState(string& strState);
: CPSR의 현재 상태를 얻는다.

void Clear();
: CPSR의 모든 flag bit들을 0으로 초기화 한다.

레지스터의 현재 설정된 값을 얻어오는 GetState()함수와 모든 bit들을 reset하는 Clear()함수를 제공한다.

CRegisterFile

- 범용 register file을 나타낸다. 범용 레지스터의 값을 설정할 수 있는 함수와 PC값을 쉽게 설정 할 수 있는 함수를 제공한다.

void WriteRegister(const bitset<4>& regNum,bitset<32>& data)
: 지정된 register에 bitset값을 설정한다.

bitset<32> ReadRegister(const bitset<4>& regNum)
: 지정된 register에서 bitset값을 읽는다.

DWORD GetPCValue()
: PC값을 구한다.

void SetPCValue(const bitset<32>& PC)
: PC값을 설정한다.

void GetState(string& strState)
: Register의 상태를 보여주는 string을 return한다.

CDecoder

- 2단계 decode단계에서의 처리를 수행한다.IF/ID register의 값을 decode하여 ID/EX register의 해당 register를 채운다.

void Decode(const bitset<32>& instruction,CIDEXRegister& reg)
명령어를 decode하여 CIDEXRegister의 해당 register를 채운다.

CIFIDRegister

- IF/ID register를 나타낸다. BL등의 분기 명령어에서 link시에 사용할 다음 PC값을 저장하는 nextPC register와 fetch해온 명령어를 저장할 instruction register를 가진다.

void GetState(string& strState);
: 레지스터의 값을 문자열로 얻는다.

void Clear();
: 레지스터를 초기화한다.

void SetNOP();
: instruction을 NOP로 설정한다. flush line이 설정되었을 때 호출된다.

CIDEXRegister

- IDEX register를 나타낸다. decode한 결과로 만들어지는 다음의 register를 저장한다.

```
bitset<32> nextPC; //다음 PC값 link시에 사용
bitset<4> cond;     //cond [31-28]
bitset<8> OPCode; //OPCode [27-20]
bitset<4> Rn;       //Rn[19-16]
bitset<4> Rd;       //Rd[15-12]
bitset<4> Rm;       //Rm[3-0]
bitset<8> immed_8;  //immed_8 [7-0]
bitset<4> shift_imm; //shift_imm[11-8]
bitset<12> offset_12; //offset_12[11-0]
bitset<24> targetAddress; //target address [23-0]
```

다음의 인터페이스를 가진다.
void GetState(string& strState);
: 레지스터의 값을 문자열로 얻는다.
void Clear();
: 레지스터를 초기화한다.
void SetNOP();
: register들을 NOP에 적절하게 설정한다. flush line이 설정되었을 때 호출된다.

8.2.3 User Interface Related Class

CPSDlg

- UI를 처리하는 dialog를 나타낸다. CProcessor class의 instance를 소유하여 버튼을 눌렀을 때 processor의 적절한 함수를 호출한다. Edit창에 보여줄 문자들의 가독성을 높이기 위해서 고정길이의 font를 만들어서 edit의 font로 지정하였다.

afx_msg void OnLoadFile();
: 파일 열기 버튼을 눌렀을 때의 처리를 한다.

afx_msg void OnTrace();
: Trace 버튼을 눌렀을 때의 처리를 한다.

afx_msg void OnReset();
: Reset 버튼을 눌렀을 때의 처리를 한다.

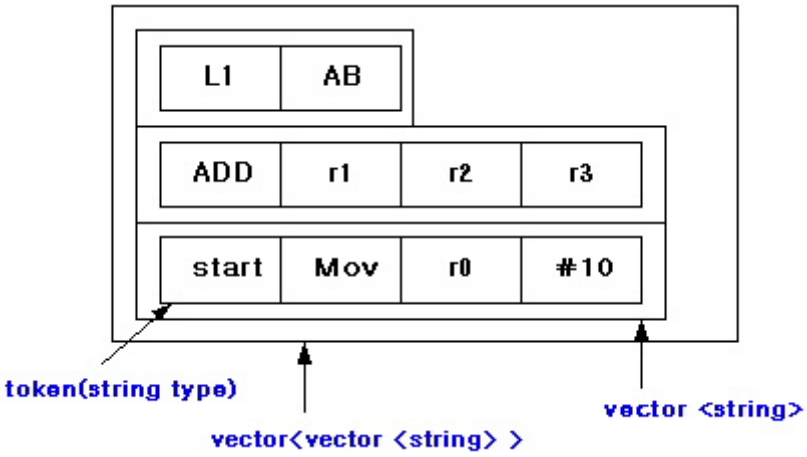
afx_msg void OnExit();
: Exit 버튼을 눌렀을 때의 처리를 한다.

8.3 Logic Explanation in detail

8.3.1 어셈블리 코드를 machine code로 변환하는 방식

어셈블리 code의 machine code로의 변환은 CInstructionReader class에서 담당하고 아래에서 기술한 처리 방식으로 변환을 수행한다.

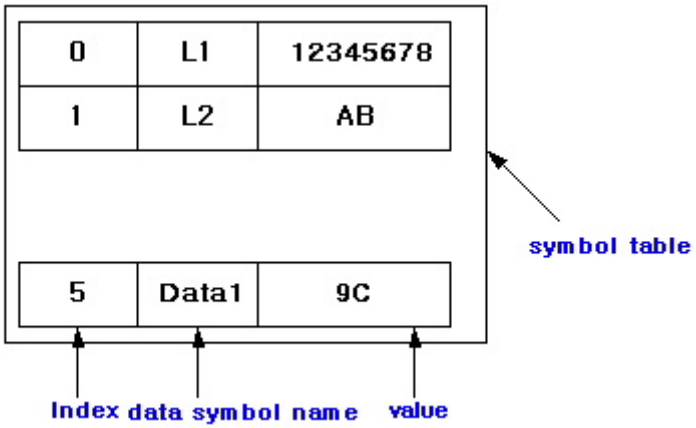
1. 파싱을 쉽게 하기 위하여 명령어의 token으로 이루어진 token vector로 구성한다. 이때 공백라인과 주석을 삭제한다.(By GetFileContent(),GetTokenVector())
token vector는 아래의 그림처럼 이루어 진다.



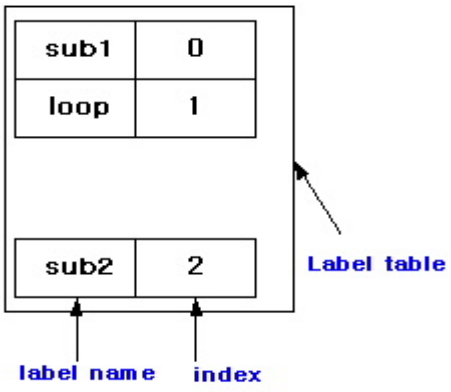
2. 소스 코드의 'ENTRY'와 'start'사이의 전방 data symbol을 symbol table에 추가한다.
라인별로 두개의 token을 읽어서 레이블과 값으로 symbol table에 저장한다.
data symbol구성후 'ENTRY'로부터 'statrt'이전의 라인들을 삭제한다.
(By FrontDataSymbolToSymbolVec(tokenVec);)

3. token이 두개로 이루어져 있고 두번째 token이 '0'으로 시작하면 후방 data symbol의 시작 위치임을 알수 있다. 이로부터 'END'가 나올때 까지 라인 별로 두개의 토큰을 읽어서 레이블과 값으로 symbol table에 저장한다.
data symbol구성후 'END'위치부터 vector의 끝까지 라인들을 삭제한다.
(By RearDataSymbolToSymbolVec(tokenVec);)
2,3의 과정을 거친후 아래 그림과 같은 symbol table이 구성된다.

4. 3,4의 과정에서 symbol table구성후 필요없는 라인을 삭제 하였기 때문에 이과정에서는 [Label] +명령어로 구성된 소스들로만 구성된다.
레이블을 찾아서 label table을 구성한다.
label은 첫번째 token이 keyword가 아니고 두번째 token이 keyword인 것으로 구별 해낼 수 있다. (keyword는 'MOV', 'ADD'등의 명령어의 집합을 말한다.)
첫 번째 토큰을 레이블 명으로 하고 현재 라인수를 값으로 label table을 구성한다.

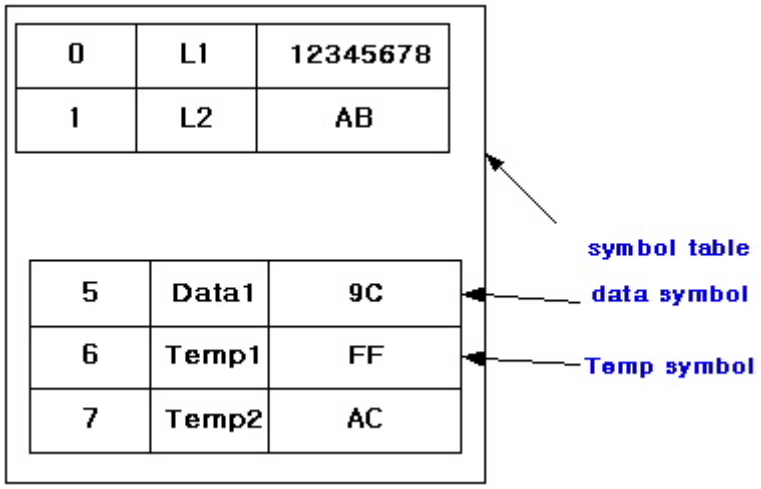


처리한 label토큰을 삭제한다.
(By GetLabelAddress(tokenVec);)

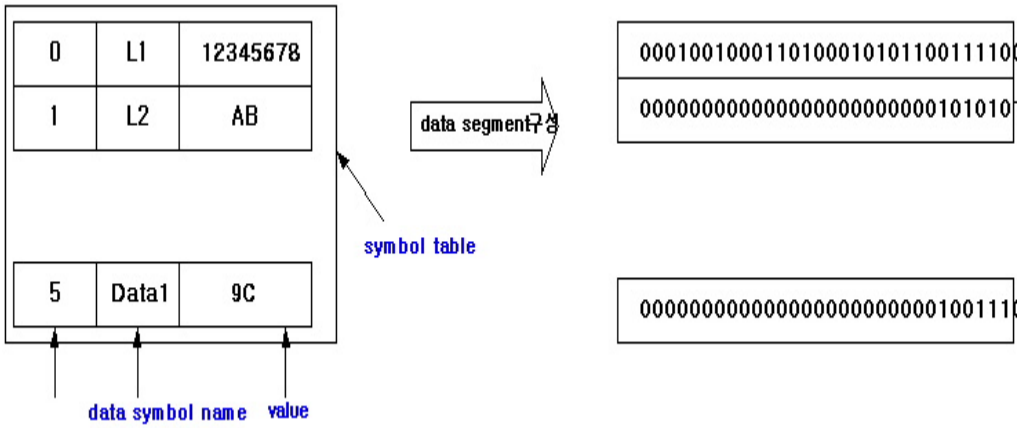


위의 그림에서 index는 code segment의 상대 위치를 나타낸다.

5. 4번 과정을 거치면서 순수 명령어로만 이루어진 소스만 남게 된다. token의 갯수가 3개이고 3번째 token이 '='로 시작하는 명령어를 찾는다.(예:LDR Rd =12345678의 형식)
세 번 째 token에 대해서 temp symbol을 만들고 만들어진 temp symbol에 세 번째 토큰의 값을 할당하고 세 번째 토큰을 temp symbol명으로 치환한다.
(예: LDR r1,=12345678 -> LDR r1,=TEMP1)
temp symbol을 symbol vector에 추가한다.
(By VoluntarySymbolToSymbolVec(tokenVec);)
위의 과정을 거치면 아래의 그림과 같이 symbol table에 temp symbol이 할당된다.



6. symbol table로부터 data segment를 구성한다.



7. 각 명령어에 따라 code segment를 채운다.
(By TranslateMachineCode(tokenVec);)

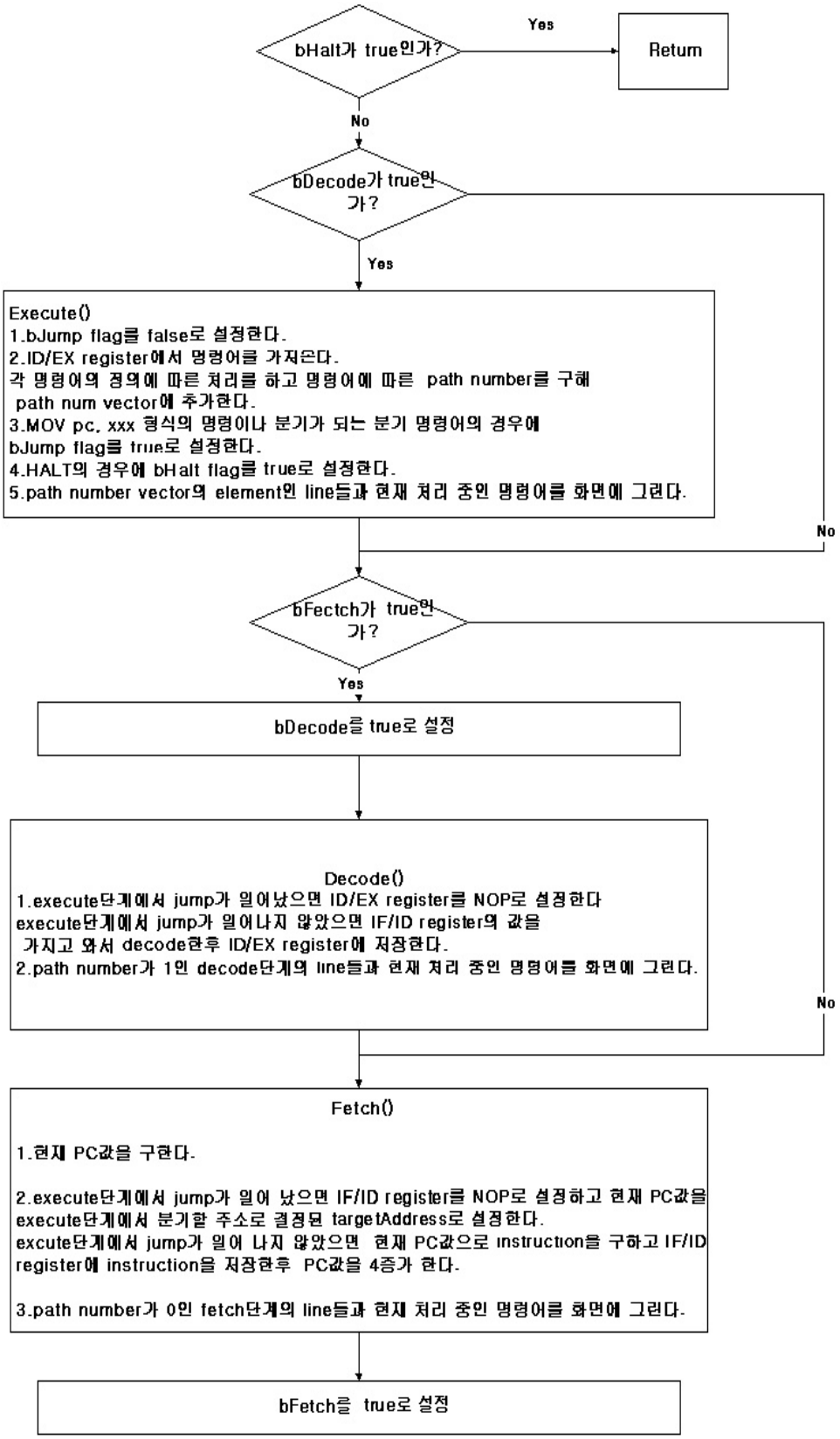
각 명령어세서 PC상대 주소는 다음과 같이 계산한다.
1) LDR rd,label형식의 명령어는 LDR rd,pc,offset형식으로 바꾼다.
offset은 symbol vector에서 lable의 위치 - (symbol vector의 size + 현재 명령어의 code segment상의 index + 2)로 구해진다.

LDR rd,Data1의 경우 symbol table에서 Data1의 인덱스는 5이고 symbol vector의 size는 6이다. 그리고 현재 명령어가 code segment에서 7번째에 위치 하고 있다면
offset = 5 - (6 +7 +2) = -10이다.

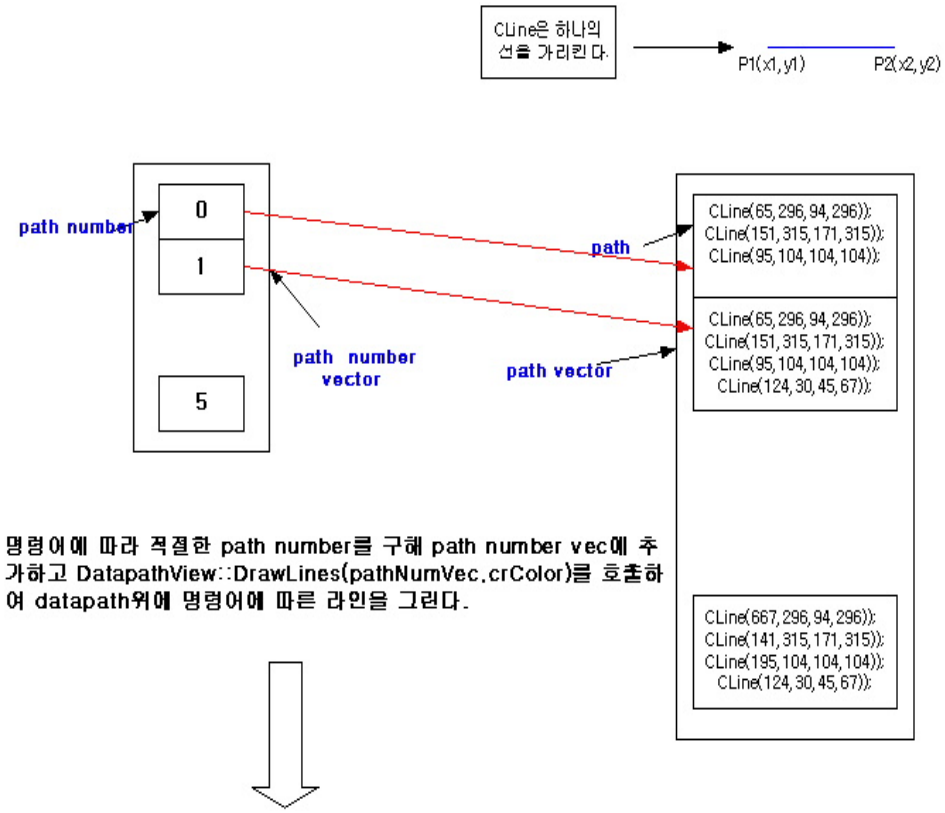
2) 분기 명령어의 label형식의 targetAddress주소를 구하는 방식

1.label을 label table에서 찾아서 있다면 target addresss = lable의 label table에서의 위치 - (code segment에서 현재 명령어의 인덱스 + 2)이다.
(1의 과정에서 target address를 구하면 2의 과정은 하지 실행하지 않는다.)
2.label을 symbol table에서 찾아서 있다면 target addresss = lable의 symbol table에서의 위치 - (code segment에서 현재 명령어의 인덱스 + 2)이다.

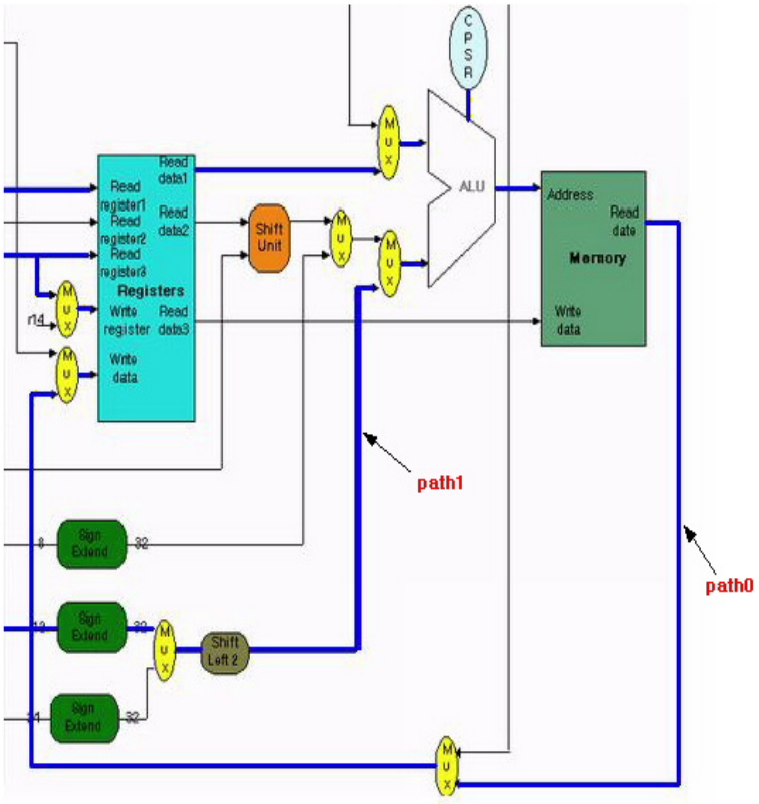
8.3.2 CProcessor Class내에서 trace 처리 방식



8.3.3 CDatapathView에서 데이터 패스위에 명령어에 따른 선 그리는 방식



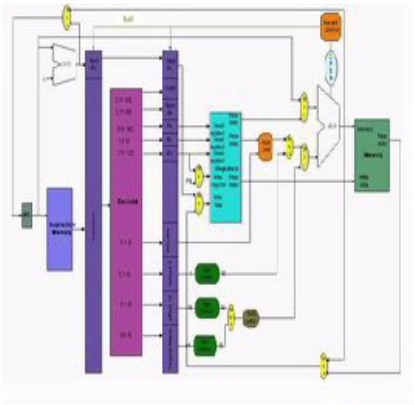
명령어에 따라 적절한 path number를 구해 path number vec에 추가하고 DatapathView::DrawLines(pathNumVec, crColor)를 호출하여 datapath위에 명령어에 따른 라인을 그린다.



8.3.4 CDatapathView에서 double buffering처리 방식

윈도우에 무엇인가를 반복적으로 출력하려면 깜박임(Flickering)이 발생한다. 이번 프로젝트에서도 datapath bitmap과 그 위에 line을 여러게 그려서 이것을 다시 화면으로 뿌려줘야 하기 때문에 더블 버퍼링 기법을 사용하여 깜박임을 없애야 한다. 따라서 datapath는 두개의 memory dc를 가지게 된다.

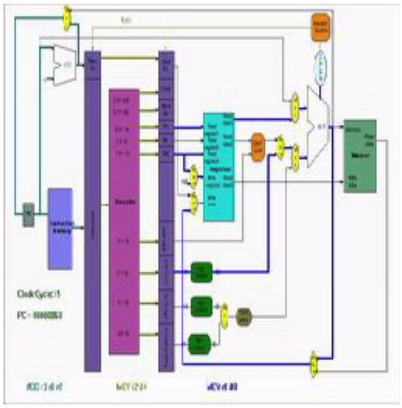
Background DC
:datapath bitmap만을 가지고 있다.memory DC를 지우는데 사용한다.



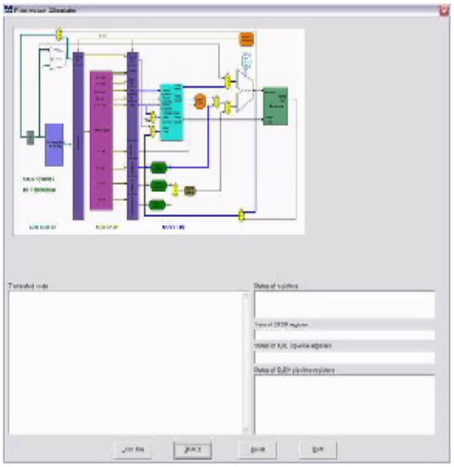
BitBlt()함수를 사용하여 복사한다.

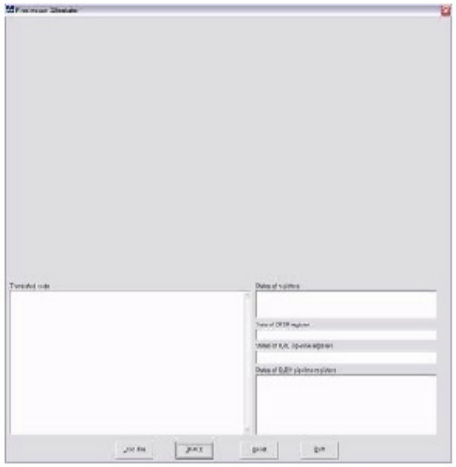
위의 memory dc간의 복사 과정을 flip이라고 하는데 Draw()함수 호출 후 첫번째 linedraw()함수가 호출 되었을 때에 한번만 filp이 일어나게 된다.

memory DC
:Backgrond DC를 복사한 후 그 위에 명령어에 따른 line을 그리는데 사용한다.



필요한 모든 라인을 그린후에 BitBlt()함수를 사용하여 Dialog의 DC(device context)에 그린다.





9. User Interface and Execution Result

9.1 User Interface

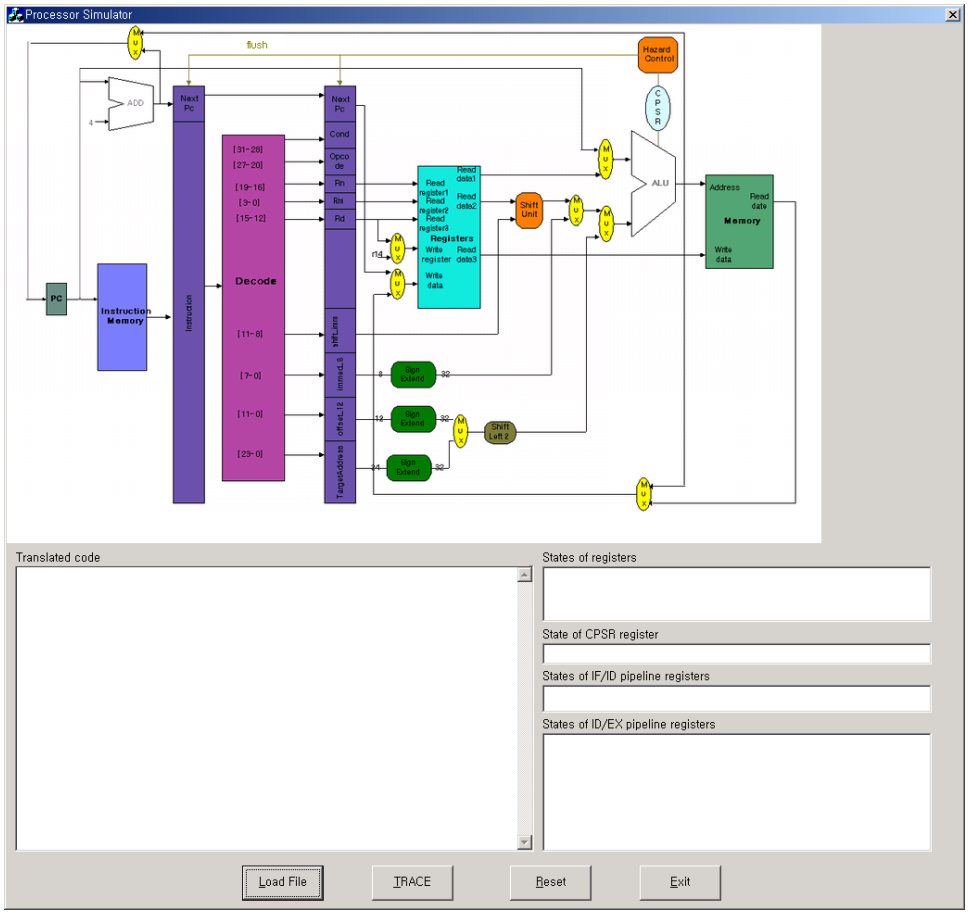
이번 프로젝트에서 만든 program의 user interface에 대해서 설명해보자.

9.1.1. 프로그램 실행 환경

- Operating System: Windows 2000 Professional or its Compatibles
- Monitor Resolution: 1280X1024 이상에 최적화 되어있음.

9.1.2. 화면 구성

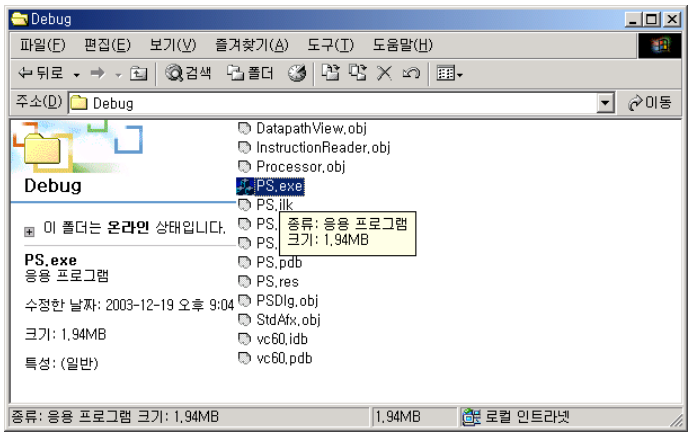
프로그램을 수행시키면 다음과 같은 화면이 출력된다.



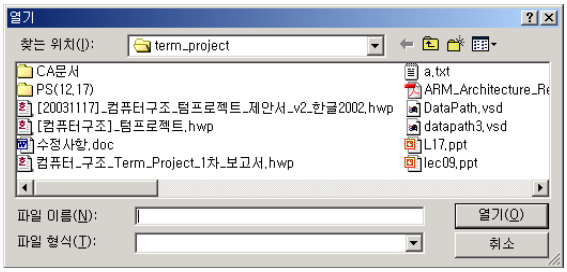
1) 화면의 상단에는 설계한 Data Path가 나타난다.
2) 화면의 좌측하단에는 Source Assembly Program을 기계어로 바꾼 결과가 나타난다.
3) 화면의 우측하단에는 현재 Register들의 상태가 나타난다.
4) 그리고 제일 아래에는 프로그램의 제어를 할 수 있는 [Load], [TRACE], [Reset], [Exit] Button들이 있다

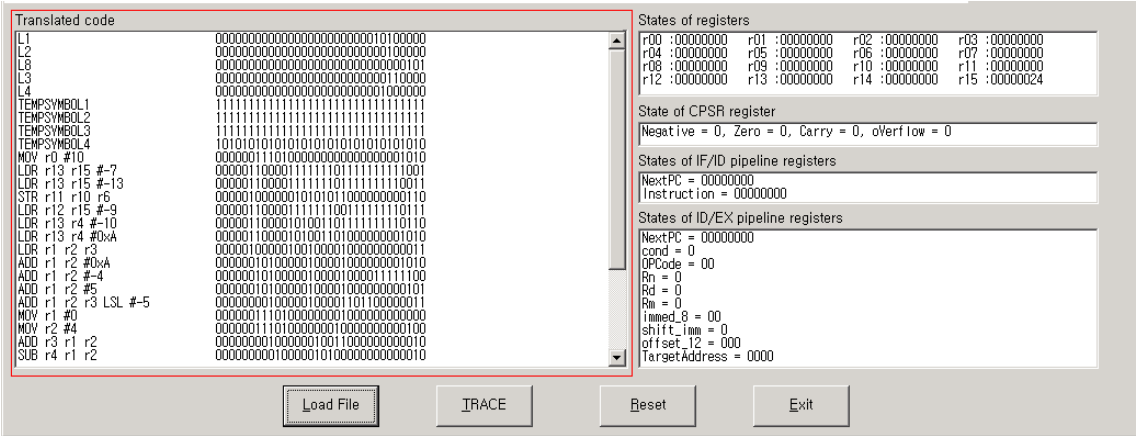
9.1.3. 프로그램 실행방법

1) 실행파일 "PS.exe"을 더블 클릭 하여 실행시킨다.

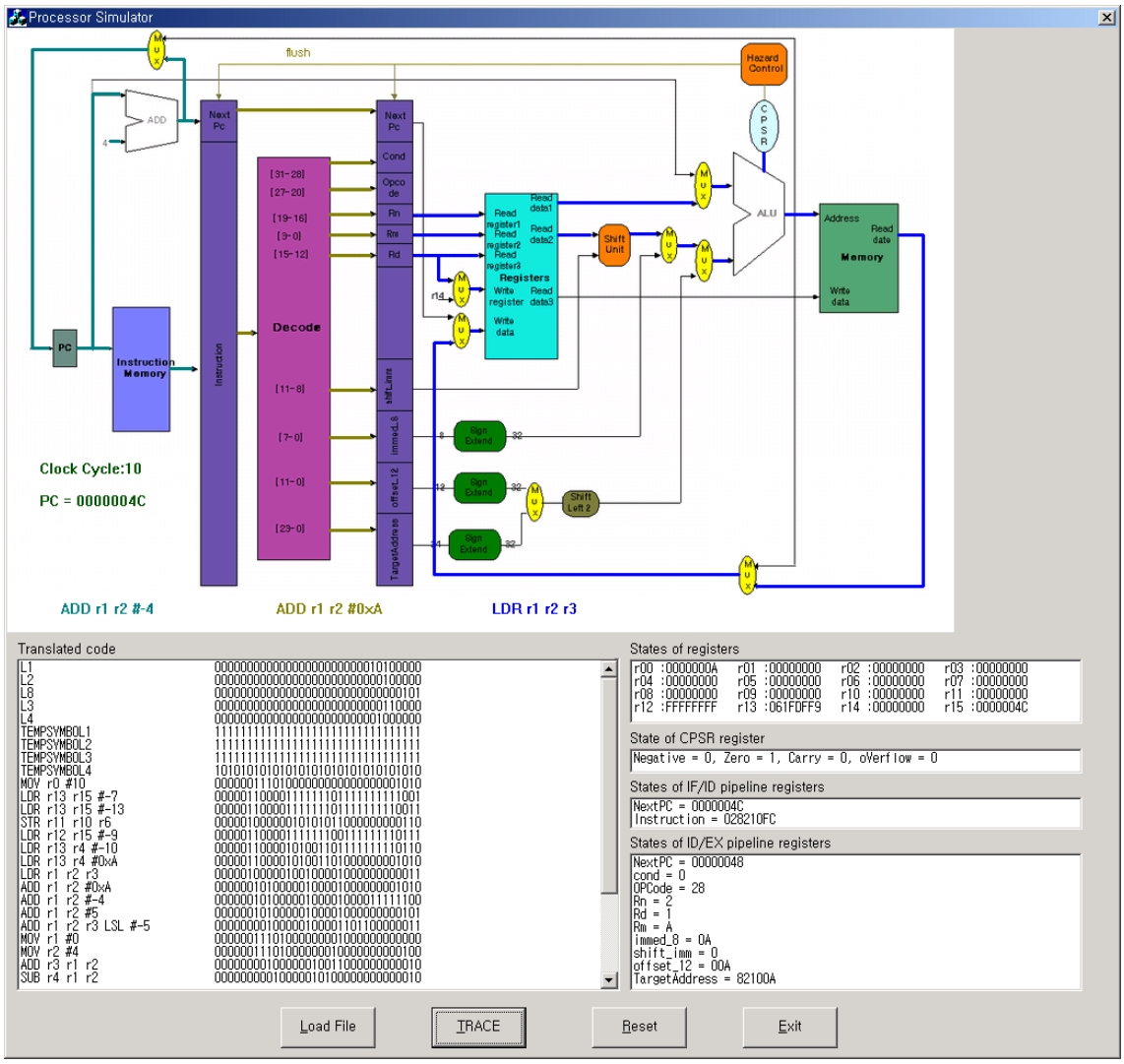


2) 마우스로 [Load] Button을 누르거나 keyboard의 "L"을 누름으로써 simulation에서 실행시킬 assembly언어로 이루어진 source file을 지정한다. Source file을 입력 받아서 Simulator가 수행할 수 있는 Machine Code로 변환시키게 된다.





3) 마우스로 [Trace] Button을 누르거나 keyboard의 "T"를 누름으로써 Simulation을 수행한다. 계속 Tracing을 시키면 한 clock씩 Simulator가 machine code를 수행하게 된다.



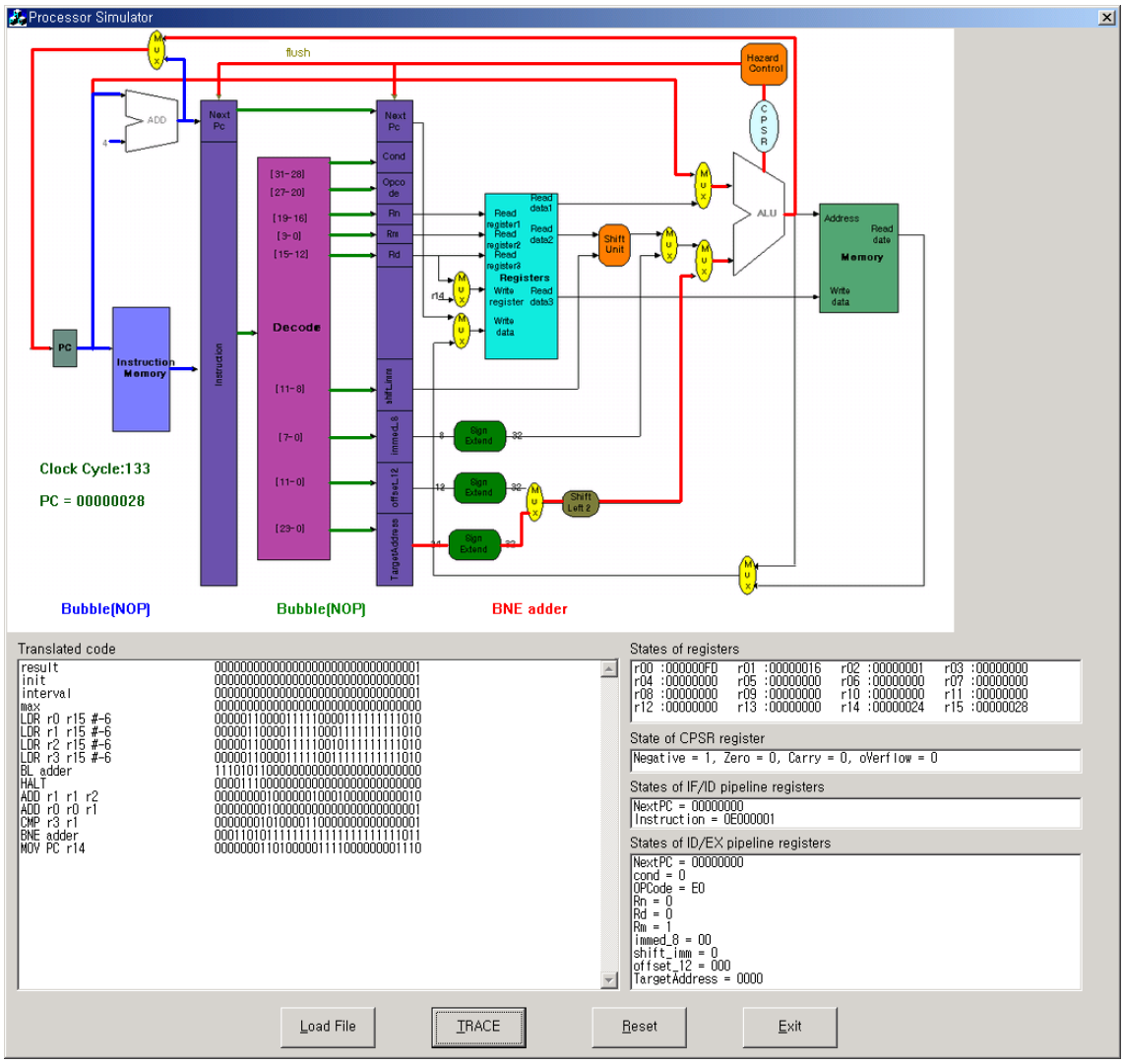
4) 만약, 처음부터 다시 시작하고 싶으면, [Reset] Button을 누르거나, keyboard의 "R"를 누름으로써, 프로그램을 초기화시킬 수 있다.

5) 모든 프로그램을 종료하고 싶으면, [Exit] Button을 누르거나, keyboard의 "E"를 누름으로써 종료시킨다.

9.2. 프로그램 수행결과

9.2.1 상황

Branch Instruction이 Not Taken한다고 가정했는데, 실제로는 Taken하여서 Hazard Detection Unit이 앞의 두 pipeline register들에 있는 값들을 모두 flush시키고, NOP operation을 수행, 즉, Stalling을 두 cycle을 수행한다.

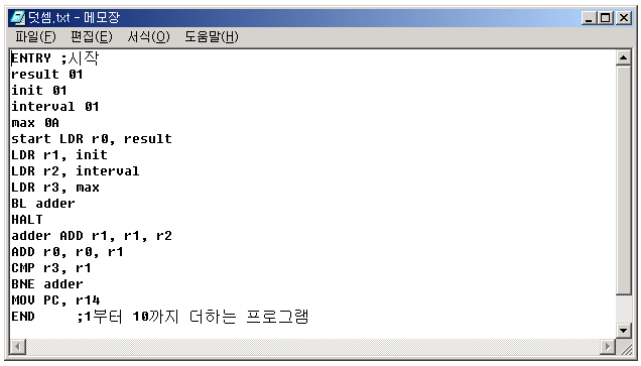


<그림: Control Hazard가 일어나 Hazard Control Unit이 NOP를 setting하는 경우>

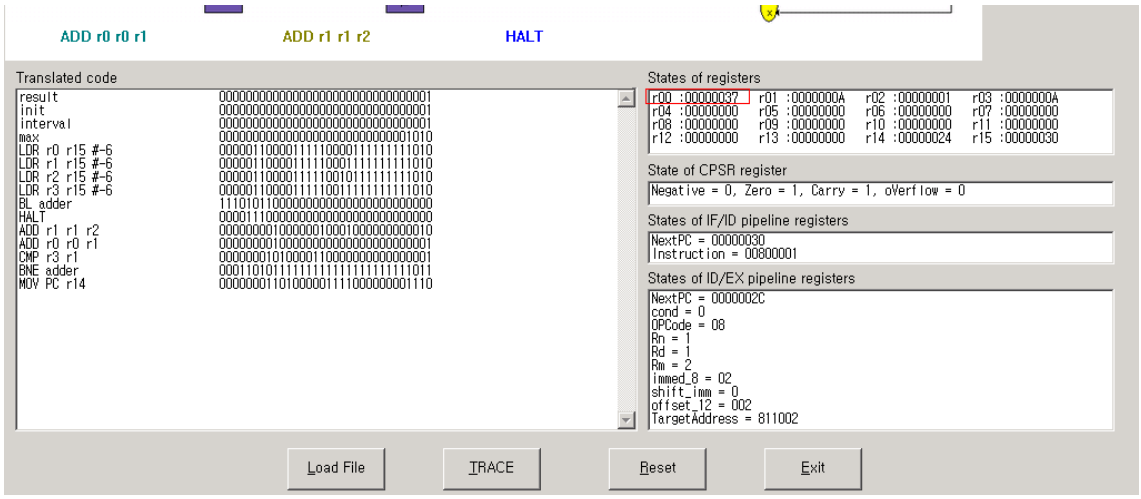
=> Hazard Control Unit에서 Flush선이 setting 되고 BNE adder 이후의 두 개 instruction들은 Bubble(NOP)로 치환된다.

9.2.2. 예제1

1부터 10까지의 값을 더해서 Register 00에 저장하는 프로그램.



<그림: 1에서 10까지의 합을 구하는 source assembly 프로그램>



<그림: assembly program 수행의 최종결과>

=>

1+2+3+4+5+6+7+8+9+10 = 55 = 0x36 (Hexadecimal)

위의 Register r00의 상태를 보면, 000000000037 로 되어서 올바른 결과가 나타남을 알 수 있다.

9.2.3. 예제2

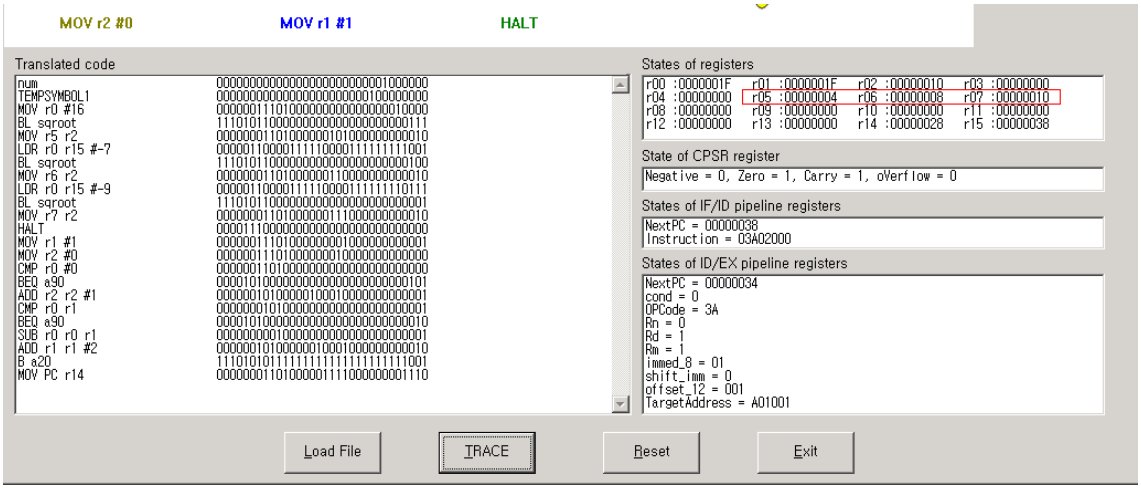
16, 0x40(64), 0x100(256)의 square root값을 구해서 Register r05, r06, r07에 저장하는 프로그램

ROOT.txt - 메모장

파일(F) 편집(E) 서식(O) 도움말(H)

```
;               evaluate root()
ENTRY
num             040
start           MOV     r0, #16                 ; r0 = square root를 구하고자하는 값
                BL      sqroot
                MOV     r5, r2                  ; 16의 root값을 r5에 저장
                LDR     r0, num                 ; 0x40의 square root값을 구한다.
                BL      sqroot
                MOV     r6, r2                  ; 0x40의 square root값을 r6에 저장해 놓는다.
                LDR     r0, =100                ; 0x100의 square root값을 구한다
                BL      sqroot
                MOV     r7, r2                  ; 0x100의 square root값을 r7에 저장한다.
                HALT
sqroot          MOV     r1, #1                  ; square root를 구하는 함수, r0에 square root를 구하
                MOV     r2, #0                  ; initialize
                CMP     r0, #0
                BEQ     a90                     ; 입력값이 0이면 그냥 리턴.
a20             ADD     r2, r2, #1
                CMP     r0, r1                  ; end of evaluation
                BEQ     a90
                SUB     r0, r0, r1
                ADD     r1, r1, #2                      ; next odd number
                B       a20
a90             MOV     PC, r14                 ; 함수 return
END
; end of file
```

<그림: Square root를 구하는 source Assembly Program>



<그림: 위의 source program을 수행한 결과>

=>

Register r05에 sqrt(16) = 4, r06에 sqrt(0x40) = 8, 그리고 r07에 sqrt(0x100) = 10 이 저장되어서 올바른 결과가 나타남을 알 수 있다.

10. Conclusion and Future Work

10.1 애로사항

모든 프로젝트가 원활히 진행되기 위해서는 정확한 요구명세를 파악하는 것이 가장 먼저 이루어져야 한다. 잘못된 요구분석을 통해 만든 프로그램은 나중에 가서 전혀 쓸모가 없게 될지도 모르기 때문이다. 이번 프로젝트에서 가장 어려웠던 점은 요구사항이 자주 바뀌었고 텀 설명서의 문서에 없는 내용이 많아서 요구분석을 제대로 하기 힘들었다. 그래서 초반에 설계를 제대로 하지 못해서 아무런 진척 없이 시간낭비를 많이 했다. 나중에 요구명세를 명확히 파악한 후에는 쉽게 진행되었던 거 같다. 소프트웨어 공학시간에서 들었던 요구분석의 중요성을 새삼 느끼게 해준 프로젝트였다.

10.2 성능 개선 방안

이 프로젝트에서 우리의 창의성을 나타낼 수 있는 부분이 Hazard 처리에 관한 것이다. 이번에는 3단계 pipelining을 구현하였는데, 분기의 여부를 알아내는 것이나, 분기할 주소를 알아내는 시점이 마지막 3단계에 가서나 가능하였다. 그러므로 분기 예측 시에 Taken을 가정하든 Not Taken을 가정하든지, 또 Delayed Branch방법을 사용하든지 최악에는 2 cycle을 Stalling해야 한다. 만약, 2단계 decode 단계에서 분기할 주소를 알아낸다면, Not Taken Prediction을 하면 개선된 결과를 얻을 수 있다. 분기명령어는 loop을 구현하는 경우가 많은데 loop은 대부분의 경우에 branch를 taken하고 마지막에 한번만 Not Taken하게 된다. 이러한 점을 생각했을 때, branch를 taken한다고 prediction한다면, 이번에 구현한 Processor보다 나은 throughput을 얻을 수 있을 것이다.

10.3 Epilogue

이 진 무
이번 학기를 마지막으로 졸업하게 되는 시점이다. 그리고 또 이번 프로젝트가 마지막 프로젝트이다. 시작이 반이듯이 유종의 미 또한 중요한 것 같다. 마지막 학기를 잘 마무리할 수 있어서 좋았다. 재현이형 메리크리스마스.

조 재 현
학생으로서 마지막 프로젝트라 아쉬움이 많이 남는다. 4학년 마지막학기라 편하게 보낼 줄 알았는데 CA 텀 프로젝트 덕분에 긴장감을 놓치지 않고 생활할 수 있었다. 진무는 좋은 학교에서 admission받고 거기서 공부 열심히 해서 훌륭한 사람 되고... 난 회사에서 아마 코딩하며 밤을 지새워야 하지 않을까? 모두들 메리 크리스마스~~