테스트 시스템을 위한 프로그래밍 언어와 컴파일러 설계

(Design of a Programming Language and a Compiler for Test Systems)

고 훈 준 † 유 원 희 ††
(Hoon Joon Kouh) (Weon Hee Yoo)

요 약 테스트 시스템은 다양한 종류의 반도체 제품을 검사하고 분류한다. 따라서 테스트 시스템은 여러 가지 특수기능의 하드웨어 모듈과 각 제품을 테스트할 수 있는 프로그램이 필요하다. 프로그램은 엔지니어에 의해 수정되고 컴파일되어 실행될 수 있어야 한다. 따라서 테스트 시스템은 쉽고 편리하게 프로그래밍할 수 있는 프로그래밍 언어와 테스트 프로그램을 컴파일하고 실행할 수 있는 컴파일러가 필요하다.

본 논문에서는 기존의 국내 테스트 시스템에서 사용하고 있는 테스트 프로그래밍 언어와 컴파일러의 환경을 서술한다. 그리고 산업현장에서 엔지니어가 좀 더 쉽고 편리하게 사용할 수 있고 향상된 성능을 가지는 프로그래밍 언어와 컴파일러를 설계・구현하였다. 본 논문에서 설계한 프로그래밍 언어와 컴파일러를 사용하여 테스트 시스템에 적용해 본 결과 기존 시스템보다 제품을 검사하는 실행 속도 면에서 성능이 향상되었다.

키워드 : 테스트 시스템, 제어 명령어, 어셈블리언어 코드, 가상기계

Abstract Test systems verify and classify the various kinds of semiconductor products. So test systems need programs that can test the various special functions of hardware modules and products. Programs can be modified, compiled and executed by engineers. Consequently, the systems needs programming languages that can be easily programmed by engineers and their compilers that can compile and execute test programs.

In this paper we discuss the environment of programming languages and their compilers for the existing domestic test systems. We design a programming language and implement its compiler that can be conveniently used by the experienced engineers in the industry field. Experimental results show that a newly designed test system with our programming language and compiler can test products faster than the existing test system.

Key words : test System, control instruction, assembly language code, virtual machine

1. 서 론

테스트 시스템은 다양한 종류의 반도체 제품을 웨이퍼(wafer) 또는 완성된 제품으로 생산하여 전기적 특성과 성능을 검사하고 그 결과에 따라 제품을 구분하는 검사 장비이다. 테스트 시스템은 크게 하드웨어와 소프트웨어로 구성된다. 하드웨어는 제품의 전기적 특성을 검사하는 테스터와 제품을 이송하는 핸들러(handler)로 구성된다. 소프트웨어는 시스템을 제어하고 사용자 인터페이스 및 각종 자료를 처리하는 프로그램으로 구성된다[1]. 따라서 테스트 시스템은 여러 가지 특수 기능의 하드웨어 모듈(module)과 각 제품을 테스트할 수 있는 프로그램이 필요하다. 이 프로그램은 엔지니어에 의해 수정되고 컴파일되어 실행될 수 있어야 한다. 따라서 테스트 시스템은 쉽고 편리하게 프로그래밍할 수 있는 프로그래밍 언어와 테스트 프로그램을 컴파일하고 실행할 수 있는 컴파일러가 필요하다.

이러한 시스템은 고품질, 고신뢰성이 요구되는 기술

† 학생회원 : 인하대학교 전자계산공학과
hjkouh@hanmail.net
†† 종신회원 : 인하대학교 전자계산공학과 교수
whyoo@inha.ac.kr
논문접수 : 2001년 10월 23일
심사완료 : 2002년 2월 18일

집약적인 산업으로 고유의 기술적 장벽이 높아 몇몇 유명한 외국 업체가 세계 시장 및 국내 시장을 독점해 왔다. 따라서 국내의 반도체 생산 기업들은 시스템 LSI와 메모리 반도체 검사설비의 90%이상을 수입하여 사용하고 있다. 그 결과 테스트 비용이 웨이퍼 제조비용을 초과하는 현상을 나타내고 있다.

현재 가장 큰 시장을 점유하고 있는 TERADYNE INC.는 파스칼 언어를 기반으로 개발한 PASCAL/STEPS와 C++ 언어를 기반으로 개발한 언어를 테스트 프로그래밍 언어로 사용하고 있다. 이 두 언어는 TERADYNE 테스트 시스템을 사용하고 있는 수많은 테스트 엔지니어들에게 사용되어 왔고 그 편리성과 신뢰성을 인정받고 있다[2,3].

국내에서 테스트 시스템을 개발하고 있는 STATEC INC.는 고성능의 하드웨어 개발이 완료되어 국내외적으로 인정받고 있지만 테스트 프로그래밍 언어와 컴파일러는 자체적으로 개발하는 것이 어려운 실정이다. 따라서 지금까지 개발한 테스트 시스템의 프로그래밍 언어와 컴파일러는 C 언어와 상용 C 컴파일러를 사용해왔다[1].

본 논문에서는 엔지니어가 테스트 시스템을 편리하게 사용할 수 있는 프로그래밍 언어와 컴파일러를 설계한다. 프로그래밍 언어는 기존 시스템과 호환성을 유지하기 위해 C 언어를 기반으로 설계된다. 시스템 제어 명령어는 기존의 C 언어를 사용하고 있는 테스트 시스템의 엔지니어와 외국 시스템을 사용하던 엔지니어가 모두 쉽게 사용할 수 있도록 두 가지 타입의 언어 형태가 혼합되어 사용될 수 있는 테스트 프로그래밍 언어로 설계된다. 컴파일러는 테스트 관리 프로그램과 효율성을 고려하여 번역기(translator)와 가상기계(virtual machine)를 테스트 관리 프로그램 내에 내장할 수 있도록 설계된다. 그리고 테스트 관리 프로그램이 시스템을 제어, 관리하고 프로그램을 컴파일하고 실행할 수 있도록 한다. 이와 같은 시스템 구조는 테스트 관리 프로그램과 테스트 프로그램과의 효율성과 유연성의 문제를 해결할 수 있다. 그리고 테스트 시스템을 사용하는 엔지니어에게 편리성을 제공할 수 있다.

2. 테스트 시스템의 환경

테스트 시스템은 [그림 1]과 같이 테스트 관리 프로그램, 테스트 기계, 핸들러, 컴파일러로 구성된다.

테스트 시스템의 동작 구조는 다음과 같이 구성된다. 첫째, 테스트 관리 프로그램에서 테스트할 반도체 제품의 특성에 맞는 테스트 프로그램을 작성하고 컴파일한다. 원시프로그램을 컴파일하면 실행 파일(*.exe)이 생성된다. 둘째, 반도체 제품이 준비되면 테스트 관리 프로그램은 테스트 프로그램의 실행 파일을 실행하여 테스트 기계를 제어하고 핸들러를 사용하여 제품의 특성을 검사한다. 셋째, 측정된 제품의 특성에 따라 핸들러가 제품을 구별한다[1]. 그러나 이와 같은 기존의 방법은 다음과 같은 문제점이 있다. 첫째, 프로그래밍 언어로 C 언어를 사용하고 상용 컴파일러를 사용하는 경우는 테스트 실행 시 매번 프로세스를 생성해서 프로그램을 실행한다. 따라서, 프로그램 실행은 한 개의 반도체 제품이 검사될 때마다 프로세스가 생성되고 한 개의 제품 검사가 끝나면 프로세스가 소멸된다. 이로 인하여 매번 프로세스가 생성되고 소멸되는데 걸리는 시간이 전체적인 테스트 시간을 증가시키는데 영향을 준다. 둘째, 기존의 엔지니어는 PASCAL/STEPS 언어를 기반으로 작성하였다. 따라서 이러한 방법의 프로그램 작성은 엔지니어에게 불편한 시스템 제어 명령어를 제공한다. PASCAL/STEPS 형식의 제어 명령어의 예는 다음과 같다.

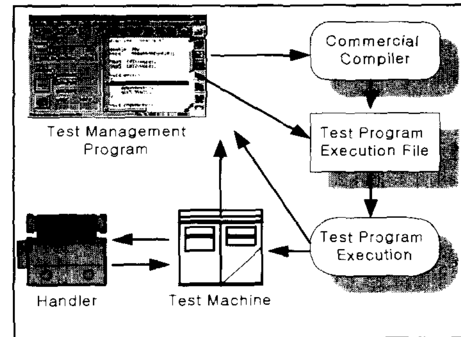


그림 1 테스트 시스템의 구성도

```
SET S1 V=10V I=10MA;
```

이 명령어는 C 언어에서 함수 형식으로 다음과 같이 구성하여 사용하였다.

```
set_source(1, 10*V, 10*MA);
```

그러나 외국 시스템의 시스템 제어 명령어에 익숙한 테스트 엔지니어에게는 C 언어 형식의 제어 명령어가 불편하다. 셋째, C 컴파일러는 상용 컴파일러를 사용하기 때문에 테스트 관리 프로그램과 분리되는 문제점이 있다. 또한 프로그램 디버깅도 상용컴파일러의 디버깅 시스템을 사용해야한다. 이는 엔지니어가 테스트 관리 프로그램을 사용하는데 불편하다. 테스트 시스템은 성능 향상이 중요하지만 이 시스템을 사용하는 엔지니어에게

```
set_source_v(int sno, double volt);
set_source_i(int sno, double current);
set_source_vr(int sno, int range);
set_source_ir(int sno, int range);
set_source_force(int sno, int force);
set_source_line(int sno, int cmd);
set_source_vvr(int sno double volt, int vrange);
set_source_iir(int sno, double current, int irange);
set_source_vi(int sno, double volt, double current);
set_source(int sno, double v, int vr, double i, int ir, int force);
set_source(int sno, double v, double I, int force);
set_hsource_v(double volt);
set_hsource_i(double current);
set_hsource_vr(int range);
set_hsource_ir(int range);
set_hsource_force(int force);
set_hsource_line(int cmd);
set_hsource_vvr(double volt, int vrange);
set_hsource_iir(double current, int irange);
set_hsource_vi(double volt, double current);
set_hsource(double v, int vr, double i, int ir, int force);
set_hsource(double v, double i, int force);
set_meter_input(int input);
set_meter_vrange(int vrange);
set_meter_filter(int flag);
set_meter_invr(int input, int vrange);
set_meter(int input, int vrange, int filter);
```

```
set_timer_mode(int mode);
set_timer_start(double v);
set_timer_stop(double v);
set_timer_v(double v1, double v2);
set_timer_bchannel(int chan);
set_timer_echannel(int chan);
set_timer_bslope(int slope);
set_timer_eslope(int slope);
set_timer_slope(int slope1, int slope2);
set_timer_trigger(int trig);
set_timer_prescaler(int pres);
set_timer_timeout(int tm);
set_timer_period(int slope, int ps);
set_timer_filter(int filter);
set_cbit(int wno, int bit);
set_ias_ih(double ih);
set_ias_il(double il);
set_ias_pulse(double width);
set_ias_start();
set_ias_clr();
wait(double time);
calc(char *str, double date);
reset(int sno);
reset(HVS);
reset(METER);
reset(TIMER);
reset(CBIT);
reset(IAS);
read(TIMER);
```

그림 2 C 언어 형태의 제어 명령어

편리성을 주는 것은 무엇보다도 중요하다.

3. 테스트 시스템의 프로그래밍 언어 설계

본 장에서는 테스트 시스템에서 엔지니어가 편리하게 사용할 수 있는 새로운 프로그래밍 언어 T를 설계한다. 테스트 시스템의 프로그래밍 언어 T는 두 가지 타입의 언어 형태가 혼합되어 사용되기 때문에 본 논문에서는 기본 구문과 제어 구문으로 구분하여 설명한다.

3.1 T 언어의 기본 구문

T 언어의 기본 구문은 Allen I. Holub[4]의 순수 C 언어를 기반으로 작성하였다. T 언어의 기본 구문은 일반적인 산술 연산자, 관계연산자, 논리연산자를 지원하고, 조건문, 반복문, 제어문, 함수 호출, 일차원 배열 등 거의 순수 C 언어 구문과 동일하다. 그러나 포인터, 구조체, 다중 배열, 공용체, 열거형 등은 지원하지 않는다. 또한 변수와 함수 타입은 void, int, char, double 형만을 지원한다. 그 외에 본 테스트 프로그램에 필요하지 않은 몇 가지 구문들은 제거한다.

3.2 T 언어의 제어 명령어 구문

T 언어의 제어 명령어 구문은 테스트 기계를 제어하는 명령어로 구성된다. 제어 명령어는 주요 하드웨어 기능에 따라 다음과 같이 여섯 가지로 나눌 수 있다[5].

① DCS(Direct Current Source) Commands
② HVS(High Voltage Source) Commands
③ VM(Volt Meter) Commands
④ Timer Commands
⑤ Cbit Commands
⑥ IAS Commands

여섯 가지의 제어 명령어를 기반으로 엔지니어에게 편리한 구문을 설계한다. [그림 2]는 T 언어에서 사용되는 C 언어 형태의 시스템 제어 명령어이다.

이와 같은 명령어는 외국 시스템에 익숙한 많은 엔지니어에게는 불편함을 주고 신뢰성을 주지 못한다. 따라서 [그림 3]과 같이 엔지니어에게 익숙한 제어 명령어를 설계한다. 형식은 PASCAL/STEPS의 제어명령어를 기반으로 하여 본 시스템에 알맞게 변형한다.

명령어의 종류는 SET S, SET HVS, SET METER, SET TIMER, SET CBIT, SET IAS, SET POWER, RESET, READ, WAIT으로 분류할 수 있고, 모두 알파벳 대문자를 사용한다. [그림 3]에서 기호 '~'는 사용 가능한 숫자의 범위를 나타낸다. '[ ]'는 그 안의 문자열 중 하나의 문자열만 사용 가능함을 나타내는 기호이다. 따라서 '[1~9]'의 의미는 1부터 9사이의 숫자 하나를 사용할 수 있음을 나타낸다. 각 제어 명령어는 세미콜론 ';'으로 마치고, 하나의 제어명령어에서 설정하는 여러 값들은 엔지니어가 설정하고자 하는 것만 필요에 따라 사용할 수 있다. 예를 들어, 명령어 SET HVS의 경우, 설정 가능한 값들은 V, VR, I, IR, FORCE, LINE이 있는데 SET HVS V=100V I=5A; 와 같이 설정에 필요

```
SET S[1~9]
     V=[-65V~65V]
     VR = [0V~1000V | AUTO]
     I = [0~5A]
     IR = [2UA~20UA | [2~200]MA | 2A | 5A | AUTO]
     FORCE=[V | I]
     LINE=[ON | OFF] ;
SET HVS
     V=[0V~1000V]
     VR=[100V~1000V | AUTO]
     I=[0~5A]
     IR= [2UA~20UA | [2~200]MA | 2A | 5A | AUTO]
     FORCE=[V | I]
     LINE=[ON | OFF] ;
SET METER
     INPUT=[VM1 | VM2 | S[1~9]]
     VR=[0V~1000V]
     FILTER=[ON | OFF]
     AVG=[1~100] ;
SET TIMER
     BCHAN=[A | B]
     ECHAN=[A | B]
     BSLOPE=[POS | NEG]
     ESLOPE=[POS | NEG]
     MODE=[INTERVAL | PERIOD]
     PRESCALE=[1~100000]
     TIMER=[1~100000]MS
     START=[-100~100]V
     STOP=[-100~100]V
     TRIGGER=[10V~100V]
     FILTER=[100KHZ | 1MHZ | 10MHZ | NONE] ;
SET CBIT W[1~4]=[0~07777] ;
SET IAS [ IH=[-50A~50A] IL=[-50~50]A PULSE=[1~255]US
     | START | CLR ] ;
SET POWER [ON | OFF];
RESET [ALL | S[1~9] | SOURCE | CBIT | TIMER | HVS |
        IAS | METER];
READ(TIMER | METER);
WAIT [1~650000]MS;
```

그림 3 PASCAL/STEPS 언어를 기반으로 한 제어 명령어

한 값들만 사용이 가능하다. 명령어 RESET의 경우는 RESET ALL;, RESET CBIT; 중 한가지를 설정할 수 있다. 이와 같은 형식의 제어 명령어들은 전압, 전류, 시간 등을 조절하여 시스템을 제어한다.

[그림 5]는 테스트 프로그래밍 언어 T의 예제 프로그램이다. 프로그램은 C언어의 구조를 가지면서 [그림 3]의 제어 명령어 구조가 혼합되어 있다. 이와 같은 프로그램을 컴파일하는 방법은 첫째, 두 언어 구조를 하나의 문법으로 구성하여 한번에 컴파일하는 것이다. 이 방법은 성격이 다른 언어 구조를 하나의 구조로 구성하기에 어렵다. 둘째, [그림 3]의 제어 명령어를 [그림 2]의 명령어로 변환하면 원시프로그램 전체가 순수 C언어 프로그램이 된다. 따라서 컴파일하기에 쉽다. 또한 엔지니어는 새롭게 제시한 제어 명령어와 기존 C언어의 함수로 구성된 제어 명령어를 모두 사용할 수 있다. 본 논문에서는 두 번째 방법을 적용한다. 제어 명령어는 선행 처리기에서 매크로 처리와 함께 실행해서 순수 C언어로 변환된다.

4. 번역기와 가상기계 설계

본 장에서는 [그림 4]와 같이 설계한 T언어를 목적코드로 변환하는 번역기와 가상기계를 기존의 테스트 관리 프로그램에 내장되도록 설계한다. 번역기는 선행처리기(preprocessor), 전위부(front-end), 후위부(back-end)로 구분하여 설명한다. 전위부는 원시프로그램을 읽어 어휘 분석, 구문 분석 단계로부터 중간코드인 어셈블리언어 코드를 생성하는 단계이다. 후위부는 어셈블리언어 코드를 읽어 목적코드를 생성하는 어셈블러 단계이다[6,7].

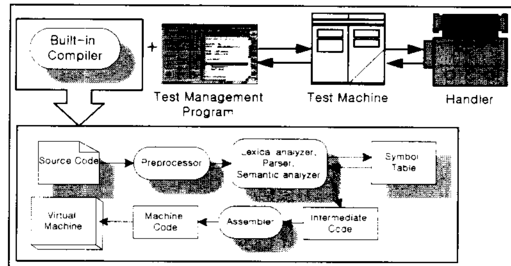


그림 4 제안한 시스템의 구성도

4.1 선행처리기

T 언어는 C 언어의 부분 집합과 특수 제어 명령어 구문이 혼합되는 프로그래밍 언어이기 때문에 동시에 어휘분석과 구문 분석이 어렵다. 따라서 본 논문에서는 선행 처리기에서 매크로 처리와 함께 먼저 특수 제어 명령어를 순수 C 언어의 함수로 변환한다. 그리고 순수 C 언어로만 구성된 프로그램을 컴파일한다. 제어 명령어를 변환하는 선행처리기 알고리즘을 요약하면 [알고리즘 1]과 같다.

[알고리즘 1] 선행처리기

1단계 :	원시프로그램을 입력으로 **#include**를 만나면 그 파일을 찾아서 열고 그 파일의 EOF(End of File)까지 읽는다. 그리고 **#define**을 만날 때마다 매크로 테이블에 값을 저장한다. 다시 **#include**를 만나면 1단계를 반복한다.
2단계 :	다시 원시프로그램을 입력으로 모든 제어 명령어 구문을 대응되는 C 언어 함수 구문으로 변환한다.
3단계 :	2단계에서 변환된 프로그램을 입력으로 모든 토큰을 매크로 테이블과 비교하여 일치하는 토큰을 해당하는 값으로 변환한다.

[알고리즘 1]에 의해 [그림 5]의 예제를 변환하면 [그림 6], [그림 7]을 거쳐 [그림 8]이 된다. [그림 6]은 [그림 5]에서 1단계에 의한 매크로 처리 결과이다. #define과 #include가 제거된다. [그림 7]과 같이 2단계

의 변환으로 제어 명령어들이 C 언어의 함수 구문으로 변환된다. b=a+NUM은 [그림 8]에서 b=a+10으로 변환된다. 또한 2단계에서 변환된 set_source_vi(2, 2.0*V, 20*MA);의 경우, V는 1로 MA는 1E*-3으로 변환된다. 그 결과는 [그림 8]과 같다.

```
#define NUM          10
#define V   1
#define MA           1E-3
#define CBIT         600
void main()
{
int a, b;
a = 1;
b = a + NUM;
   RESET CBIT;
   SET CBIT W1=0123 ;
   SET S2 V=2.0V I=20MA;
}
```

그림 5 원시프로그램

```
void main()
{
int a, b;
a = 1;
b = a + NUM;
   RESET CBIT;
   SET CBIT W1=0123 ;
   SET S2 V=2.0V I=20MA;
}
```

그림 6 1단계의 결과

```
void main()
{
int a, b;
a = 1;
b = a + NUM;
   reset(CBIT);
   set_cbit(1, 123);
   set_source_vi(2, 2.0*V,
     20*MA);
}
```

그림 7 선행처리기 2단계 결과

```
void main()
{
int a, b;
a = 1;
b = a + 10;
reset(600);
set_cbit(1,123);
set_source_vi(2, 2.0*1,
  20*1E-3);
}
```

그림 8 선행처리기 3단계 결과

선행처리기를 3단계로 설계하면 다음과 같은 장점이 있다. 첫째, 제어 구문이 특별한 형식이든, C 언어 형식이든 모두 사용이 가능하고, 둘째, #include 된 파일 내에 존재하는 제어 명령어가 #define의 매크로 이름을 포함한 제어 명령어라도 2단계에서 제어 명령어를 C 언어의 함수로 변환하고 3단계에서 매크로를 처리하기 때문에 이와 같은 방법이 가능하다. 만약, 2단계와 3단계의 순서를 바꾸어 처리하면 제어 함수내의 매크로 이름은 처리할 수 없다. 그리고 선행 처리를 하면 모든 형식이 T 언어의 기본 구문으로 변환되기 때문에 어휘분석과 구문분석을 쉽게 할 수 있다.

4.2 전위부

전위부는 선행 처리된 원시프로그램을 읽어 들여 어휘분석, 구문분석 단계로부터 AST(abstract syntax tree)를 어셈블리언어 코드로 생성하는 단계이다. 이는 WINDOWS용 Lex & Yacc[8] 자동화 도구를 이용하였다. Lex를 이용하여 어휘분석을 하고 Yacc을 이용하여 구문분석과 함께 AST를 생성한다. 이때 원시프로그램에서 어셈블리언어 코드가 생성되기까지 필요한 정보들은 심볼 테이블(symbol table)[6]에 저장된다[9]. 심볼 테이블의 중요 부분은 다음과 같다.

```
struct symbol {
        char *name;
        IDENT IDkind;
        union sym_value value;
        Exptype v_type;
        struct function_node *FNode;
        struct sym_struct *next;
        ...
}
```

IDENT은 Label, Variable, Array, Pointer, Function을 나타내고 Exptype은 Void, Char, Int, Double을 나타낸다. FNode는 심볼이 함수 이름이면 해당하는 함수 노드의 위치를 저장한다.

AST를 생성하기 위해서는 심볼 테이블과 함수 노드 테이블 그리고 트리 노드 테이블을 사용한다. 함수 노드 테이블의 중요 부분은 다음과 같다.

```
struct function_node {
        SymStruct *name;
        ExpType ReturnType;
        struct symbol *LocalSym;
        struct tree_node *treenode;
        ...
}
```

ReturnType은 함수의 리턴 타입을 나타내고 LocalSym은 각 함수 내에 지역 심볼 테이블을 나타낸다. 그리고 treenode는 각 문장들을 트리로 구성하기 위한 노드를 나타낸다.

트리 노드 테이블의 중요 부분은 다음과 같다.

```
struct tree_node {
        struct symbol SymPtr;
        struct treenode *sibling;
        struct treenode *child[4];
        ...
}
```

트리 노드는 원시프로그램의 각 문장들을 트리로 구성하기 위한 노드이다. SymPtr은 각 노드의 변수 정보

가 저장된 심볼 테이블 위치를 나타낸다. 이러한 테이블을 이용하여 [그림 8]로부터 생성된 AST의 중요 부분은 [그림 9]와 같다.

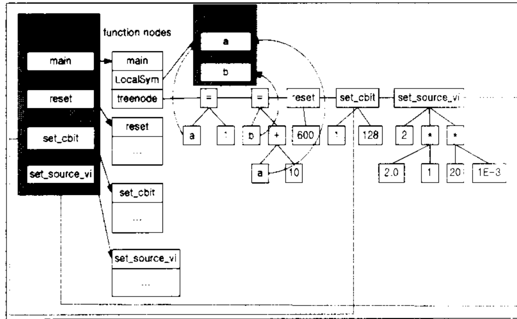


그림 9 그림 8에 대한 AST 생성 결과

원시프로그램에서 전역변수는 전역 심볼 테이블(global symbol table)에 정보가 저장된다. 함수는 전역 심볼 테이블에 정보가 저장되고 함수 노드 테이블을 생성하여 심볼 테이블과 연결한다. 지역 변수는 각 지역 심볼 테이블(local symbol table)에 저장된다. 각 함수 내의 문장들은 트리 노드 테이블로 트리를 구성하여 함수 노드 테이블과 연결한다.

어셈블리언어는 스택 가상기계에서 수행하기 위한 명령어로 구성된다. 명령어는 기본 명령어와 하드웨어 제어 명령어로 구분된다. 기본 명령어는 가상기계 수행 명령어이고 제어 명령어는 C 언어 함수의 제어 명령어를 처리하기 위한 명령어이다.

[표 1]은 본 논문에서 사용한 어셈블리 언어의 기본 명령어이다.

IADD, DADD에서 I는 정수형을 뜻하고, D는 실수형을 뜻한다. PUSHL은 현재의 FramePtr와 offset을 더해서 그 값을 스택에 PUSH하는 명령어이다. SETSP는 스택에 [length]만큼 공간을 확보하고 스택 포인터를 바꾸고, SETP는 현재의 스택 포인터를 지역변수의 시작주소로 하는 명령어이다. CONTENTS는 스택에서 POP한 값을 주소로 하여 그 주소의 값을 스택에 저장하는 명령어이다. 또한 I2D는 스택에 있는 값을 POP해서 그 값을 실수 값으로 타입 변환 후 스택에 저장하는 명령어이고 D2I는 반대로 스택에 있는 값을 POP해서 그 값을 정수 값으로 타입 변환 후 스택에 저장하는 명령어이다. 번호는 목적 코드의 명령어 번호와 일치한다.

[표 2]는 시스템 하드웨어의 제어 명령어이다. 명령어 형식은 '명령어 [인자 개수] [타입, …]'이다. 만약 하드웨어 제어 명령어들을 직접 프로그램 소스에 넣어 컴파일하고 실행할 경우, 프로그램의 크기가 매우 커져서 컴파일과 실행 속도에 영향을 미치게 된다. 따라서, 하드웨어 제어 명령어는 C언어 형식의 함수로 만들어서 라이브러리로 만들고 [표 2]의 명령어들로 라이브러리의 함수를 호출하도록 하였다. 따라서 가상기계가 각 명령어들을 수행하다가 제어 명령어를 만나면 라이브러리를 호출함으로서 제어 명령어를 처리한다.

[그림 10]은 산술문, 조건문, 반복문의 원시프로그램 언어에 대한 어셈블리 코드의 표현이다.

표 1 기본 명령어

No	명령어	No	명령어	No	명령어	No	명령어
1	IADD	16	ILSSEQL	31	IPUSH variable	46	SETSP length
2	ISUB	17	INOT	32	IPOPC variable	47	SETP
3	IMUL	18	DEQUAL	33	IPOP	48	RDCHAR
4	IDIV	19	DNOTEQL	34	DPUSHC double value	49	RDINT
5	INEG	20	DGREATER	35	DPUSH variable	50	RDDOUBLE
6	DADD	21	DLESS	36	DPOPC variable	51	WRCHAR
7	DSUB	22	DGTREQL	37	DPOP	52	WRINT
8	DMUL	23	DLSSEQL	38	AREF size	53	WRDOUBLE
9	DDIV	24	DNOT	39	BRANCH address	54	CONTENTS
10	DNEG	25	PUSHC char value	40	JUMP	55	ICONTENTS
11	IEQUAL	26	PUSH variable	41	BREQL address	56	DCONTENTS
12	INOTEQL	27	PUSHL offset	42	BRLSS address	57	HALT
13	IGREATER	28	POPC variable	43	BRGTR address	143	I2D
14	ILESS	29	CPOP	44	CALL address	144	D2I
15	IGTREQL	30	IPUSHC integer value	45	RETURN		

표 2 제어 명령어

No	명령어	No	명령어	No	명령어
59	RESET_CBIT	87	SET_IAS_START	115	SET_SFT_ACTIVE
60	SET_CBIT	88	SET_IAS_VGON	116	RESET_TIMER
61	READ_CBIT	89	READ_IAS_DATA	117	READ_TIMER
62	SET_CBIT_MASK	90	GET_IAS_IH	118	SET_TIMER_START
63	READ	91	GET_IAS_IL	119	SET_TIMER_STOP
64	RESET	92	SET_IAS_CLR	120	SET_TIMER_TRIGGER
65	RESETALL	93	SET_IAS_PULSE	121	SET_TIMER_FILTER
66	WAITTIME	94	RESET_MUX	122	SET_TIMER_MODE
67	SETPOWER	95	PROGRAM	123	SET_TIMER_TIMEOUT
68	DELAY	96	LIMIT	124	SET_TIMER_PRESCALER
69	INITSYSTEM	97	SORT_BIN	125	SET_TIMER_BCHANNEL
70	AVGREADMETER	98	CALC	126	SET_TIMER_ECHANNEL
71	LIBERROR	99	PRINT	127	SET_TIMER_BSLOPE
72	RESET_HSOURCE	100	HW_STATUS	128	SET_TIMER_ESLOPE
73	RESET_HVS	101	TRAP	129	SET_TIMER_PSLOPE
74	SET_HSOURCE_V	102	RESET_SOURCE	130	SET_TIMER_V
75	SET_HSOURCE_I	103	SET_SOURCE_VR	131	SET_TIMER_CHANNEL
76	SET_HSOURCE_VR	104	SET_SOURCE_IR	132	SET_TIMER_SLOPE
77	SET_HSOURCE_IR	105	SET_SOURCE_FORCE	133	SET_TIMER_PERIOD
78	SET_HSOURCE_FORCE	106	SET_SOURCE_LINE	134	RESET_METER
79	SET_HSOURCE_LINE	107	SET_SOURCE_V	135	READ_METER
80	SET_HSOURCE_VI	108	SET_SOURCE_I	136	SET_TIMER_INPUT
81	SET_HSOURCE_VVI	109	SET_SOURCE_VI	137	SET_TIMER_VRANGE
82	SET_HSOURCE_IIR	110	SET_SOURCE_VVR	138	SET_TIMER_FILTER
83	SET_HSOURCE	111	SET_SOURCE_IIR	139	SET_TIMER_INVR
84	RESET_IAS	112	SET_SOURCE	140	SET_TIMER
85	SET_IAS_IH	113	SET_FTIME	141	SET_TIMER_SADI
86	SET_IAS_IL	114	SET_POWER_MODE	142	SET_TIMER_SADV

```
a = a + 1;
PUSHL  -4
ICONTENTS
IPUSHC  1
IADD
PUSHL  -4
IPOP
```

```
if(a > 2) stmt else stmt
IPUSHC  2
IPUSHC  a
ICONTENTS
IGREATER
BREQL  L1
(code for stmt)
BRANCH L2
L1:
(code for stmt1)
L2:
```

```
while(a < 100)  stmt
L1:
IPUSHC  100
IPUSHC  a
ICONTENTS
ILESS
BREQL  L2
(code for stmt)
BRANCH L1
L2:
```

```
for(i=0;i<100;i=i+1) stmt
IPUSHC  0
IPUSHC  i
IPOP
L1:
IPUSHC  100
IPUSHC  i
ICONTENTS
ILESS
BREQL  L2
(code for stmt)
IPUSHC  i
ICONTENTS
IPUSHC  1
IADD
IPUSHC  i
IPOP
BRANCH  L1
L2:
```

그림 10 기본 구문에 대한 어셈블리 코드

[그림 10]에서 정의된 어셈블리언어 코드를 기반으로 [그림 9]의 AST를 변환하면 [그림 11]의 어셈블리언어 코드가 생성된다.

```
CALL    _main
HALT
_main:
SETSP   8
IPUSHC  1
PUSHL   -4
IPOP
PUSHL  -4
ICONTENTS
IPUSHC  10
IADD
PUSHL   -8
IPOP
IPUSHC  600
RESET 1 2
IPUSHC  1
IPUSHC  123
SET_CBIT  2 2 2
IPUSHC  2
DPUSHC  2.0
IPUSHC  1
I2D
DMUL
IPUSHC  20
I2D
DPUSHC  1E-3
DMUL
SET_SOURCE_VI  3 4 4 2
RETURN 0
```

그림 11 그림 9로부터 생성된 어셈블리언어 코드

2	139	44	6	0	0	0	57	46	8	0	0	0	30	1
0	0	0	27	252	255	255	255	33	27	252	255	255	255	55
30	10	0	0	0	1	27	248	255	255	255	33	47	30	88
2	0	0	[illegible]	1	0	0	0	2	0	0	0	47	30	0
0	0	0	30	123	0	0	0	[illegible]	2	0	0	0	2	0
0	0	2	0	0	0	47	30	2	0	0	0	34	0	0
0	0	0	0	0	64	30	1	0	0	0	143	8	30	20
0	0	0	143	34	252	169	241	210	77	98	80	63	8	[illegible]
3	0	0	0	4	0	0	0	4	0	0	0	2	0	0
0	45	0	0	0	0									

그림 12 그림 11의 목적 코드

[그림 11]의 프로그램은 CALL 명령어로 시작하여 _main을 호출하고 RETURN을 만나면 다시 돌아와 다음 줄의 HALT 명령을 실행함으로써 종료된다. RESET, SET_CBIT, SET_SOURCE_VI는 테스트 시스템 제어 명령어의 어셈블리 코드이다. SET_SOURCE_VI의 경우 첫 번째 인자 '3'은 인자의 개수를 나타내고, '4 4 2' 각 인자의 타입을 나타낸다. 2는 정수, 4는 실수를 나타낸다. 일반적인 사용자 정의 함수는 CALL 명령어를 사용하지만 제어 명령어는 사용자 정의 함수와 구분하여 처리하기 위해 사용하지 않는다.

4.3 후위부

목적코드는 프로그램을 가상기계에서 실행하기 위한 코드로 [표 1], [표 2]의 No를 코드 번호로 구성하여 나타낸다. 목적코드를 생성하는 어셈블러는 일반적인 two pass로 구현한다. one pass에서는 변수와 레이블의 주소 값을 레이블 테이블에 저장하고 two pass에서는 레이블 테이블을 참조하여 어셈블리언어 코드를 목적코드로 변환한다. [그림 12]는 [그림 11]에 대한 목적 코드이다. 가상기계는 [그림 12]의 코드를 메모리로 로드하고 스택을 사용하여 한번에 하나의 명령어를 수행한다.

목적 코드의 처음 숫자 2는 시작주소를 나타내고 두 번째 숫자 139는 목적 코드의 개수를 나타낸다. 나머지 숫자들은 목적 코드이며 명령어와 인자로 구성되어 있다. 예를 들면, 세 번째 숫자 44는 CALL 명령어이므로 다음의 6은 다음 명령어의 주소를 지시한다. 목적 코드에서 스택에 저장되는 정수는 4 바이트로 저장된다. 예를 들어 정수 2이면 '2 0 0 0'으로 저장된다. 그리고 색칠이 된 코드는 시스템 제어 명령어를 나타낸다. 64, 60, 109는 각각 RESET, SET_CBIT, SET_SOURCE_ VI를 나타내고 이 코드의 실행은 라이브러리를 호출하여 실행한다.

4.4 가상기계

가상기계는 Jonathan Amsterdam[10]이 제시한 가상기계를 기반으로 스택 기반 가상기계로 설계하였으며 다섯 가지 기본적인 전제조건을 가지고 설계하였다[9,11].

첫째, 메모리는 부호 없는 문자형(unsigned char) 일차원 배열로 선언하고 코드 블록과 스택으로 구분하였다.

둘째, 변수 타입은 문자형(char), 정수형(int), 실수형(double)만을 허용한다.

셋째, 실행코드는 메모리 0번지부터 저장하고, 스택을 사용할 때는 메모리의 끝에서부터 사용한다.

넷째, 가상 레지스터는 프로그램 카운터(program counter), 스택포인터(stack pointer), 프레임 포인터(frame pointer), 지역변수 시작 포인터(local variable start pointer)가 있고 함수를 복귀할 때 값을 저장할 버퍼(buffer)가 있다.

다섯째, 명령어는 1 바이트, 주소는 4 바이트, 정수는 4 바이트, 실수는 8 바이트로 구성한다.

기존의 테스트 관리 프로그램에 내장된 스택 기반 가상기계의 실행 구조는 [그림 13]과 같다.

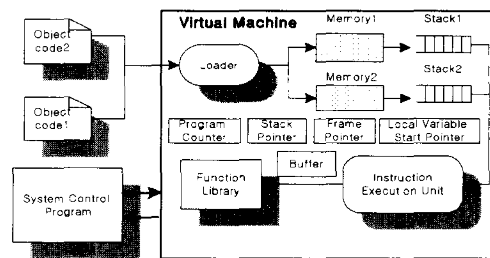


그림 13 가상기계의 실행 구조

테스트 관리 프로그램이 프로그램을 실행하면 로더가 코드를 읽어 메모리로 옮기고 가상기계는 메모리로부터 코드를 스택에서 실행한다. 테스트 시스템은 핸들러가 기본적으로 두 개이기 때문에 프로그램이 번갈아 실행할 수 있도록 메모리와 스택은 각각 두 개이다. 그리고

프로그램 실행에 대한 제어는 시스템 제어 프로그램이 제어한다. 즉, 목적 코드1이 먼저 실행되고 종료되면 핸들러1이 실행 결과에 따라 동작하는 동안 테스트 관리 프로그램이 목적코드 2를 실행한다. 목적 코드2가 종료되면 핸들러2가 실행 결과에 따라 동작된다. 그리고 엔지니어가 새로운 프로그램을 컴파일하기 전까지는 메모리1과 메모리2에 기존의 목적코드가 그대로 존재하기 때문에 이 코드를 가지고 번갈아 가며 프로그램이 실행될 수 있다. 명령어가 시스템 제어 명령어의 함수 호출 명령어 일 경우에는 함수 라이브러리의 함수를 호출하여 실행된다.

5. 실험 및 고찰

본 장에서는 테스트 시스템 AZ400에서 시스템 관리 프로그램 내에 내장한 스택기반 가상기계의 성능을 검사한다. 그리고 기존 시스템과 본 논문에서 새롭게 설계한 시스템을 비교한다.

테스트 관리 프로그램은 윈도우즈 98 환경에서 개발된 프로그램이고, 기존 시스템의 컴파일러는 Turbo C 3.0을 사용하였다. 본 실험에 사용된 예제 프로그램은 디바이스 7805 반도체 제품을 테스트하는 프로그램을 사용하였다.

실험은 세 가지 모델을 가지고 진행하였다. 첫 번째 모델은 기존 시스템 모델(TSystem1)이고, 두 번째 모델은 매번 테스트 시 목적 코드를 메모리에 적재하여 실행하는 방법 모델(TSystem2)이다. 그리고 세 번째 모델은 목적 코드를 메모리에 적재하여 계속 실행하는 모델(TSystem3)이다.

[그림 14]는 프로그램이 실행되어 제품을 검사하기 전에 테스트 시스템이 초기화하는데 걸리는 시간과 디바이스 7805 한 개를 검사하는데 걸리는 시간을 보여주고 있다.

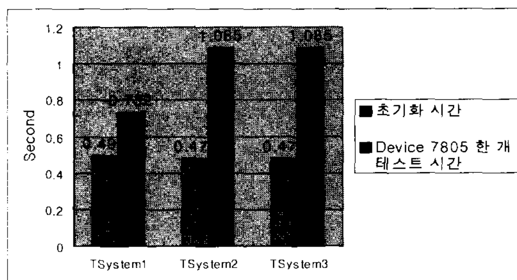


그림 14 시스템 초기화 시간과 디바이스 7805 제품 한 개를 테스트한 시간

TSystem1이 TSystem2와 TSystem3 보다 초기화하는 속도가 느린 것을 알 수 있다. TSystem1은 실행 파일을 실행하면 프로세스를 생성하고 파일을 메모리로 적재하여 실행한다. 그러나 TSystem2와 TSystem3은 가상기계가 테스트 관리 프로그램과 같이 실행된다. 따라서 프로그램을 실행하면 가상기계는 머신 코드를 메모리로 적재하고 실행하기 때문에 프로세스를 생성하는 시간이 절약된다. 그러나 1 분 동안 디바이스 7805를 테스트 한 결과 매번 목적 코드를 읽어 실행하는 TSystem2는 TSystem1에 비해 상당한 성능 저하를 나타냈다. 그러나 TSystem3의 경우는 디바이스를 검사하는 속도가 TSystem1보다 더욱 빠른 것을 알 수 있다.

[그림 15]의 실험은 핸들러가 한 개일 경우와 두 개일 경우, 각 핸들러에서 디바이스 7805 제품을 1분 동안 검사한 실험 결과이다. TSystem1은 1분 동안 TSystem2보다 더 많은 양의 제품을 검사했으나 TSystem3보다는 적은 양의 제품을 검사했다.

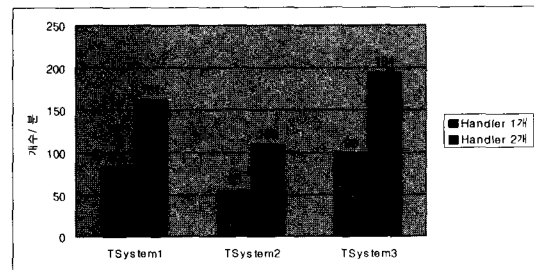


그림 15 디바이스 7805 제품을 1분 동안 테스트한 결과

핸들러를 한 개 사용한 결과와 두 개를 사용한 결과를 비교하였을 때 세 가지 시스템의 경우 거의 모두 두 배의 제품을 검사하였다. 이는 하나의 핸들러가 한 개의 제품을 검사하고 결과에 따라 핸들러가 제품을 분류하는 동안 다른 하나의 핸들러가 또 다른 제품을 검사하기 때문에 핸들러를 한 개 사용하였을 경우에 비해 두 개 사용하였을 경우 두 배의 제품을 검사하게 된다.

그 결과 기존 시스템은 프로세스 생성 시간과 목적 코드를 메모리로 로드하는 시간을 소비하기 때문에 TSystem2보다 느린 것을 알 수 있다. 실험 결과, TSystem1에 비해 TSystem3의 경우 12%의 성능이 향상되었다.

6. 결 론

테스트 시스템은 반도체 제품의 전기적 특성과 성능

을 검사하고 그 결과를 산출해내는 검사장치로 여러 가지 특수기능의 하드웨어 모듈과 각 제품을 테스트 할 수 있는 프로그램이 필요하다. 그러나 국내 테스트 시스템의 경우 하드웨어 개발은 잘 이루어지고 있으나 소프트웨어 개발 즉, 시스템에 알맞은 프로그래밍 언어와 컴파일러를 자체적으로 개발하는 것이 어려운 실정이다.

본 논문에서는 기존의 국내 기업에서 개발한 테스트 시스템의 문제점을 해결하기 위해 새로운 프로그래밍 언어와 컴파일러를 설계하였다. 첫째, 기존의 엔지니어에게 불편했던 제어 명령어의 구문을 PASCAL/STEPS 형식의 구문과 C 언어 형식의 구문 모두를 사용할 수 있도록 하였다. 따라서 엔지니어는 같은 제어 명령어지만 자신에게 편리한 구문을 사용할 수 있게 되었다. 둘째, 설계한 프로그래밍 언어를 처리하기 위해 번역기와 스택 기반 가상기계를 설계하였다. 그리고 번역기와 가상기계를 테스트 관리 프로그램에 내장시킴으로써 별도의 에디터와 컴파일러를 사용하지 않고 테스트 관리 프로그램 내에서 프로그래밍을 하고 컴파일을 할 수 있게 하였다. 그래서 테스트 관리 프로그램과 테스트 프로그램과의 유연성 문제를 해결하였고 엔지니어가 테스트 시스템을 좀 더 편리하게 사용할 수 있도록 하였다. 또한 실험 결과 실행 속도도 기존 시스템보다 좋은 성능을 보였다. 일반적으로 컴파일러 시스템에 비해 가상기계를 사용하는 인터프리터 방식의 시스템은 실행 속도가 느리다. 하지만 이 시스템은 테스트 관리 프로그램과 번역기 그리고 가상기계가 하나의 프로그램으로 만들어져 실행되기 때문에 테스트 프로그램의 컴파일과 실행 속도에서 성능이 향상되었다. 또한 테스트 시스템은 일반적으로 한가지 종류의 제품을 연속적으로 검사한다. 따라서 가상기계에서 수행되는 목적 코드를 메모리로 한번만 옮기고 연속적으로 실행하면 매번 실행할 때마다 목적 코드를 메모리에 옮기는 시간을 줄일 수 있어 실행 속도에서 성능이 향상되었다. 이와 같은 테스트 시스템의 기술 개발은 반도체 시장의 원가 절감과 경쟁력을 향상시킬 수 있다.

향후 효율적인 실행을 위해서는 어셈블리 코드와 가상기계의 최적화 연구가 필요하고 엔지니어가 편리하게 프로그램을 디버깅하는 방법에 관한 연구가 필요하다.

참 고 문 헌

[1] STATEC Inc. AZ400 Manual, Korea, 2000.

[2] Teradyne Inc. PASCAL/STEPS Reference Manual, Boston, 1995.

[3] Teradyne Inc. PASCAL/STEPS & Catalyst Digital, http://www.teradyne.com

[4] Allen I. Holub, *Compiler Design in C*, Prentice-Hall International, Inc. 1990.

[5] 고훈준, 류진수, 김기태, 유원희, "Test System 용 번역기 설계", '2001춘계 학술발표논문집 제5권 1호, 한국정보과학회, 2001.

[6] Christopher W. Fraser, David R. Hanson, "A Retargetable Compiler for ANSI C," ACM SIGPLAN Notices 26(10): 29-43, October 1991.

[7] Forest Baskett. The best simple code generation technique for while, for and do loops. ACM SIGPLAN Notices, 13(4):31-32, April 1978.

[8] John R. Levine, Tony Mason, Doug Brown, *LEX & YACC*, O'Reilly & Associates, Inc.,Suite A Sebastopol, CA, 1995.

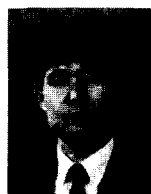
[9] 고훈준, 유원희, "컴파일러와 디버거 프로그램 개발", 최종연구 결과 보고서, 인하대학교, 2001.

[10] Jonathan Amsterdam, *A SIMPL Compiler*, BYTE, Vol.110, No10,11, 1985.

[11] 고훈준, 안용균, 조선문, 유원희, "Test System용 가상기계 설계", '2001 춘계 학술발표논문집 제8권 제1호, 한국정보처리학회, 2001.

고 훈 준

2000년 2월 인하대학교 전자계산공학과 졸업(공학석사). 2000년 3월 ~ 현재 인하대학교 전자계산공학과 박사과정. 관심분야는 컴파일러, 디버거, 보안, 병렬처리, 프로그래밍 언어 등



유 원 희

1975년 서울대학교 공과대학 응용수학과 졸업. 1978년 서울대학교 대학원 계산학 전공(이학석사). 1985년 서울대학교 대학원 계산학 전공(이학박사). 1992년 ~ 1993년 University of California, Irvine 객원연구원. 1979년 ~ 현재 인하대학교 전자계산공학과 교수. 관심분야는 프로그래밍 언어, 컴파일러, 실시간 시스템, 병렬 시스템 등