creative commons

COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학석사학위논문

DRAM 테스트 패턴 프로그램의 기호 표현형

Symbolic Representation of DRAM Test-pattern Programs

2023년 2월

서울대학교 대학원

전기 정보 공학부

조 중 하

공학석사학위논문

DRAM 테스트 패턴 프로그램의 기호 표현형

Symbolic Representation of DRAM Test-pattern Programs

2023년 2월

서울대학교 대학원

전기 정보 공학부

조 중 하

DRAM 테스트 패턴 프로그램의 기호 표현형

Symbolic Representation of DRAM Test-pattern Programs

지도교수 문 수 묵

이 논문을 공학석사 학위논문으로 제출함

2023년 2월

서울대학교 대학원

전기 정보 공학부

조 중 하

조중하의 공학석사 학위 논문을 인준함

2023년 2월

위 원 장: _____

부위원장: _____

위　　원: _____

초록

DRAM 메모리의 테스트를 위한 테스트 패턴 프로그램은 ATE(Automatic Test Equipment)에 따라 프로그래밍 언어가 다르고 저수준이므로 가독성이 현저히 떨어진다. 본 논문에서는 테스트 패턴 프로그램의 분석을 위해 새로운 형태의 기호 표현형(symbolic representation)을 제시한다. 시뮬레이션 결과로 생성된 출력 기반의 중간 표현형(intermediate representation) 으로부터 각 사이클 핀 값을 분석하여 해당 사이클 정보를 추출해내고 이를 토대로 동일한 작업을 수행하는 사이클을 동일한 기호로 변환한다. 제품 수준의 테스트 패턴 프로그램 기반 21개의 벤치마크로 기호 표현형 변환을 진행하였고 의도한 결과가 도출됨을 확인하였다. 또한 기호 표현형을 활용한 분석 툴은 사용자가 패턴 프로그램의 구조와 작업 의도를 직관적으로 파악할 수 있도록 도와준다. 테스트 패턴 프로그램의 변환이 완료된 기호 표현형은 패턴 프로그램 변환 시 동일성 테스트 및 최적화, 그리고 결함 모델(fault model) 역추론 등에 활용될 수 있다.

주요어: DRAM, 테스트 패턴 프로그램, 기호 표현형, 중간 표현형

학 번: 2021-22071

목차

초록 **i**

제 1 장 서론 **5**

제 2 장 배경지식 **7**

2.1 테스트 패턴 프로그램의 구조 7

2.2 테스트 패턴 프로그램의 호환성 문제 9

제 3 장 Cycle Equivalency **12**

3.1 Cycle Equivalency 12

제 4 장 기호 변환 **14**

4.1 제안하는 기호 변환 방식 14

4.2 Equivalence Rule 15

4.2.1 Concrete command name 판단 15

4.2.2 세부 정보 판단 16

4.2.3 프로그램 context 고려 16

4.3 분석 도구 19

제 5 장 실험 결과 **21**

5.1 실험 결과 21

제 6 장 관련 연구 및 SR 활용 사례 **24**

6.1 관련 연구 24

6.1.1 패턴 프로그램 변환 플랫폼 24
6.2 SR 활용 사례 25
6.2.1 SR 기반 Visualization 툴 25

제 7 장 결론 및 향후 과제 29

ABSTRACT 33

표 목차

표 4.1 Register operation 예시 17
표 4.2 Register equivalecy 판단 예시 17

표 5.1 한 예제 패턴 프로그램의 파트 별 기호 수 23

그림 목차

그림 2.1 테스트 패턴 프로그램 코드 예시 8
그림 2.2 패턴 프로그램 변환에서의 IR 활용 11

그림 3.1 Cycle Equivalency 예시 패턴 13

그림 4.1 Symbol conversion flow chart 14
그림 4.2 Program context 고려 예시 18

그림 5.1 예제 프로그램의 C 기반 pseudo 코드 21
그림 5.2 예제 프로그램의 기호 변환 결과 22
그림 5.3 예제 프로그램의 symbolic representation tree 23

그림 6.1 변환 플랫폼 전체 구성 24
그림 6.2 Visualization tool horizontal ver 25
그림 6.3 Visualization tool vertical ver - specific command colored . . . 26
그림 6.4 Visualization tool vertical ver - comparison of two pattern programs 27

그림 7.1 loop nesting example 29

제 1 장 서론

DRAM 메모리 테스트를 위한 테스트 패턴 프로그램은 ATE에 따라 프로그래밍 언어가 다르고 대부분 저수준 언어이다. 또한 새로운 모델이 계속 개발됨에 따라 테스트 패턴 프로그램이 새로 작성되어서 축적된다. 이러한 특징 때문에 현업에 종사하는 엔지니어들도 과거의 legacy 패턴 프로그램이 무슨 의도를 가지고 쓰여졌는지 파악하는 것에 어려움이 있다. [5]에서 제시한 IR 또한 복잡한 데이터 구조를 가지고 있으며 각 사이클 마다 모든 output vector 정보를 포함하고 있어 직관적으로 이해하기에 용이하지 않다.

본 논문에서는 IR을 기반으로 모 패턴 프로그램이 어떤 커맨드를 어떠한 구조로 수행하는지 엔지니어가 이해하기 쉽게 분석할 수 있는 도구를 위한 기호 표현형 (symbolic representation, SR)을 고안하였다. IR에 기반하여 개발하였기 때문에 다양한 ATE 종류에 대해서 호환 가능하다는 장점이 있다. 또한 SR을 구성하는 각각의 symbol은 IR의 각 사이클을 이진 기호로 표현한 것으로 한번 더 추상화된 축약 형태이기 때문에 상대적으로 더 이해하기 쉽다. 동일한 의미를 가진 사이클은 동일한 기호로 표현되며 SR의 전체적인 구조는 루프 구조를 직관적으로 표현하기 위해 트리 구조로 설계하였다.

SR은 다양한 분석과 조작에 있어 활용성이 뛰어나다. 두 패턴 프로그램의 동일성 테스트를 나열된 기호들에 대한 스트링 비교 문제로 치환할 수 있다. 이는 나아가 주어진 패턴 프로그램이 어떤 fault model을 사용하는 것인지 판단하는 추적 도구 개발로 활용될 수 있으며, 패턴 프로그램의 메모리 제한이 있는 경우 패턴 프로그램의 크기를 최적화 하는데도 활용 가능하다.

SR을 만드는 과정에서 가장 중요한 단계는 동일한 의미를 가지는 사이클, 즉 equivalent cycle을 판단하는 문제이다. 단순히 같은 레지스터로 출력을 만든다고 해서 같은 symbol로 변환할 수 없고, 다른 레지스터로 출력을 만든다고 해도 같은

symbol로 변환해야 하는 경우가 있기 때문이다.

본 논문의 구성은 다음과 같다. 1장의 서론에 이어 2장에서는 테스트 패턴 프로그램의 구조와 호환성 문제에 대해서 다룬다. 3장에서는 cycle equivalency 판단의 어려움에 대해서 설명하고 4장에서는 설정된 equivalence rule과 이에 기반한 symbol conversion의 전체 구조와 진행 그리고 분석 도구를 통해서 알아낼 수 있는 정보들에 대해 서술한다. 5장에서는 제품 수준의 테스트 패턴 프로그램에 대하여 symbol conversion을 수행한 결과와 분석 결과를 기술한다. 6장에서는 관련 연구와 symbolic representation의 활용 사례에 대해 다루고, 7장에서는 결론 및 향후 과제를 서술한다.

제 2 장 배경지식

2.1 테스트 패턴 프로그램의 구조

고속 메모리 칩을 제조하면 해당 제품의 결함을 검증하는 과정이 필요하다. 회로의 성능을 테스트하기 위해 사용되는 장비를 ATE(Automatic Test Equipment)라고 한다 [1]. ATE는 내부의 ALPG(Algorithmic Pattern Generator)라는 장비에서 데이터 읽기, 쓰기 등의 검증을 위한 instruction을 프로그래밍 할 수 있다. 이와 같이 검증에 사용되는 전기적 신호 벡터를 생성하는 프로그램을 테스트 패턴 프로그램이라고 한다.

고속 메모리 칩은 row와 column으로 이루어진 2차원 배열의 bank 여러 개로 구성된다. 테스트 패턴은 이러한 구조를 가로지르며 각각의 셀에 데이터를 읽고 쓰는 것을 반복하며 테스트를 진행하는데, 검증하고자 하는 결함의 종류에 따라 그에 맞는 알고리즘으로 row, column, bank와 읽고 쓸 데이터를 결정하게 된다. 이를 기반으로 comparator에서 테스트 패턴에 따른 DUT의 출력을 계측하여 예상 출력과 비교하고, 해당 과정을 반복함으로써 제조한 회로 결함을 검증하는 것이다.

반도체 칩이 초고속 메모리로 개발됨에 따라 신호 벡터 또한 고속으로 생성하고 전달해야 한다. ALPG의 동작 속도를 향상 시키는 것에는 한계가 있어 N개의 ALPG를 인터리빙하여 고속으로 신호를 생성, 전달한다. 그러나 인터리빙을 하게 되면 register operation 및 flow control에 제약이 발생하게 된다. 또한 반도체 메모리 테스트 패턴 프로그램이라는 특성상 DUT상의 제약 조건, 테스트 알고리즘 상의 제약 조건이 존재하기 때문에 테스트 패턴 프로그램의 제작은 전문 엔지니어를 통해 모든 제약 조건을 고려하여 완성된다.

아래의 코드는 테스트 패턴 프로그램을 간략화 한 예시 코드이다. 한 줄은 한개의 사이클을 의미하고 이를 마이너 사이클(minor cycle), 첫 사이클을 시작으로 8개의

```
1          LA1 :  NOP        :     B1=R1          ;
2                            :     Activate(1)    ;
3                            :     Activate(2)    ;
4                            :                    ;
5                            :                    ;
6                            :                    ;
7                            :                    ;
8                            :                    ;
9                 NOP        :     B2=R2          ;
10                           :     Write(1)       ;
11                           :     Write(2)       ;
12                           :                    ;
13                           :                    ;
14                           :                    ;
15                           :                    ;
16                           :                    ;
17                JNI LA1    :     Precharge      ;
18                           :     X = X+1        ;
19                           :                    ;
20                           :                    ;
21                           :                    ;
22                           :                    ;
23                           :                    ;
24                           :                    ;
```

그림 2.1: 테스트 패턴 프로그램 코드 예시

마이너 사이클 묶음을 메이저 사이클(major cycle)이라 정의한다. 메이저 사이클의 cycle 수는 인터리빙 등의 조건에 따라 달라질 수 있다.

1열의 LA1은 레이블의 의미이고 코드의 위치를 식별함으로써 점프 명령어의 사용이 용이하도록 한다. 2열은 flow control 파트로써 NOP 또는 JNI가 위치하는데 NOP의 경우 코드의 순서대로 명령이 정상 작동하며 JNI의 경우 해당 메이저 사이클을 다 수행한 다음에 뒤에 기록된 레이블이 있는 사이클로 점프하게 된다. 3열에는 뱅크에 register를 할당하거나 pre-charge, activate, read, write 등의 명령어가 위치한다.

테스트 패턴 프로그램에서 특정 명령어 간의 상관관계에 의해서 최소로 지켜야 하는 timing parameter 값이 발생한다. 예를 들어 셀에 접근하는 동안 그 셀이 속한 행은 액티브 상태로 유지되어야 하기 때문에 ACT 이후 유지해야하는 최소 사이클 수를 규정해야 한다. 이 때문에 ACT 명령어 이후 WR나 RD까지 지연이 발생한다. 또는 ACT 명령은 뱅크 내의 행을 여는데 사용되는데 뱅크 당 하나의 행을 활성 상태로 유지할 수 있다. 이에 따라 ACT to ACT 지연이 발생하고 동일 뱅크 그룹의 수행, 다른 뱅크 그룹에서의 수행에 따라 조건이 달라지기도 한다. 이와 같은 timing parameter 수치는 하드웨어 스펙에 따라 달라지기 때문에 각 모델은 고유한 타이밍 패러미터 수치를 가진다. 또한 점프 등 control flow에 해당하는 명령어의 경우 메이저 사이클의 시작 부분에 위치해야 하는 제약 조건이 있다. 이러한 제약 조건들을 만족시키기 위해 command cycle 이 후 아무것도 처리하지 않고 일정 시간 쉬어 갈 수 있도록 DES(Deselect) cycle을 의도적으로 삽입한다.

2.2 테스트 패턴 프로그램의 호환성 문제

현업에서의 테스트 패턴 프로그램의 사용에서 호환성 이슈가 존재한다. ATE는 여러 제조사에 의해 만들어지는데 각 제조사마다 프로그래밍 언어가 다르고, 같은 제조사의 경우에도 모델마다 레지스터의 개수나 ALPG의 개수 등 하드웨어 특성이 다르다.

따라서 동일한 목적을 가지고 동일한 알고리즘으로 bit vector를 출력하도록 기술한 테스트 패턴 프로그램이라도 매번 ATE 기종이나 테스트 환경에 맞게 프로그램을 새로 기술해야 한다.

이러한 이유로 이 기종 ATE 간의 테스트 패턴 프로그램 변환에서 호환성 이슈를 해결하고자 하는 연구들이 있어왔다. 첫번째로는 ATE 기종과는 무관하게 테스트 패턴 프로그램을 기술하는 인터페이스 형식을 고안하고, 이를 target ATE에 맞는 테스트 패턴 프로그램으로 변환하는 접근 방식을 제안하였다 [1, 2]. 그러나 현업에서는 새로운 인터페이스 형식의 적용과 ATE 제조사의 비협조적인 호환성 지원의 이유로 어려움이 있다.

두번째 해결 방식은 각각의 ATE에 적용할 공통 라이브러리를 제작하여 단일 소스 테스트 패턴 프로그램을 만드는 것이다 [4]. 하지만 이 접근은 현업에 사용되고 있는 대부분의 패턴 프로그램이 저수준(low-level) 언어임에 따라 활용이 제한된다.

실제로 현장에서 채택하는 방식 중 하나는 이종 ATE 테스트 패턴 프로그램 간의 직접 변환이다. 그러나 이러한 방식도 레지스터 구조나 인터리빙 하드웨어가 다른 ATE 간의 변환에서는 적용이 어려우며, 적용이 용이하다고 하더라도 n개의 서로 다른 프로그래밍 언어로 쓰여진 테스트 패턴 프로그램이 존재한다고 할 때, n^2개의 translators가 필요하다.

이러한 문제들을 보완하기 위한 컴파일러의 관점에서의 접근으로 패턴 프로그램 실행 결과 자체를 중간표현(Intermediate Representation, IR)으로 표현하고 이를 이용하여 프로그램을 변환하는 기법이 제시되었다 [5].

IR은 소스 코드를 표현하기 위해 컴파일러나 가상 머신에 의해 내부적으로 사용되는 데이터 구조 또는 코드를 말한다. 그림 2.2와 같이, n개의 서로 다른 프로그램 변환 과정에서 IR의 활용은 n^2개의 translators 대신 n개의 simulators 와 n개의 translators만 사용해도 된다는 장점이 있다. 이때 ATE 환경에서 다양한 ATE의 하드웨어를 추상화하고 테스트 패턴 프로그램의 디테일을 표현하는 것은 어려운 일이다. [5]에서는 테스트 패턴 프로그램의 출력 결과인 simulation result 자체를 IR로 사용

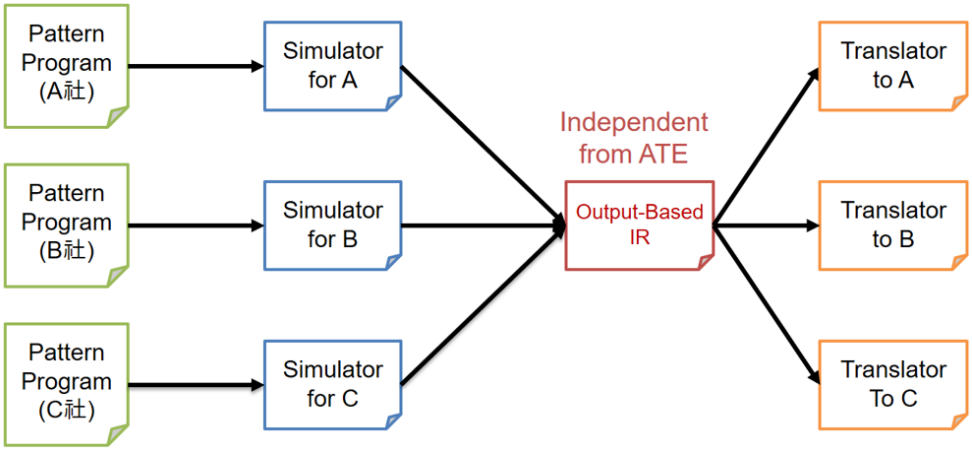

그림 2.2: 패턴 프로그램 변환에서의 IR 활용

하는 output-based IR 접근방식을 사용하였다. 같은 제품에 대해 같은 결함을 테스트하기 위해서 생성된 테스트 패턴 프로그램은 발생시키는 bit vector 가 동일하고 이로부터의 출력 또한 같기 때문에 IR 로 활용하기 효과적이다.

한 가지 문제점은 제품 수준의 패턴 프로그램 출력의 크기가 너무 크다는 것에 있는데, [5]에서는 반복되는 루프 구조를 축약하고 이에 대한 정보를 데이터 구조로 저장하는 Folded IR 의 고안으로 이를 해결하였다.

제 3 장 Cycle Equivalency

3.1 Cycle Equivalency

테스트 패턴 프로그램은 ATE의 레지스터를 사용하여 매 클럭마다 비트 벡터를 DRAM의 핀에 보낸다. 따라서 커맨드와 레지스터 및 간단한 연산을 루프 형태로 표현한다. DRAM은 bank로 나누어져 있고 bank group도 정의되어 있다.

테스트 패턴 프로그램의 symbolic representation 변환에서 핵심적인 부분은 각 사이클의 output bit-vector와 프로그램 구조를 기반으로 동일한 작업을 수행하는 사이클을 동일한 기호로 변환하는 작업이다. 다시 말해 cycle equivalency를 판단하는 것이 SR 변환의 핵심이다. 그림 3.1은 cycle equivalency를 판단하는 문제가 쉽지 않은 이유를 설명하는 예시이다.

그림 3.1의 예시에서 기호 a, b, c의 세 비트 벡터를 가지고 패턴 1은 abc로 표현할 수 있는 루프를 1000번 반복하고 있으며 패턴 2는 abcabc로 표현할 수 있는 루프를 500번 반복하고 있다. 두 패턴은 사실 동일한 패턴이므로 각 패턴에서 Activate 명령을 수행하는 사이클은 사용하고 있는 레지스터의 종류가 x_1, x_2로 다르다고 하더라도 equivalent cycle로 판단하여 a라는 동일한 symbol로 변환하여야 한다. 하지만 패턴 3을 보면 패턴 2에서 아래 주황색으로 표시된 x_2의 변화량이 8에서 64로 바뀌었으므로 x_2라는 같은 레지스터를 사용하였다 하더라도 레지스터 내용의 차이가 있으므로 equivalent cycle로 판단하면 안 된다. 이와 같이 각 사이클의 output vector 값만 비교하여 cycle equivalency를 판단할 수 없고 코드의 context를 고려하여 기호 변환을 진행해야 SR 변환의 결과를 제대로 활용할 수 있는 것이다.

```
// Pattern 1: read for 1000 times, increase x_1 by 8. (abc)^1000
// High-level style pseudocode
x_1 = 0;
Loop(1000){
 Activate(register x_1);  // symbolized as "a"
  ...          // perform read and precharge. Read as "b", Precharge as "c"
  x_1 += 8;
}
```

다른 이름의 레지스터를 사용하지만
Equivalent cycle

```
// Pattern 2: same output with Pattern 1. (abcabc)^500
// High-level style pseudocode
x_2 = 0;
Loop(500){
 Activate(register x_2); // symbolized as "a". x_2 here is 0, 16, 32, …
  ...          // perform read and precharge. Read as "b", Precharge as "c"
  x_2 += 8;
 Activate(register x_2); // this must be "a" too. x_2 here is 8, 24, 40, …
  ...          // perform read and precharge
  x_2 += 8;
}
```

같은 Register 를 쓰는
같은 Command 지만
equivalent cycle 이 아님

```
// Pattern 3: looks similar with 2 but cannot be optimized (abcdbc)^500
// High-level style pseudocode
x_2 = 0;
Loop(500){
 Activate(register x_2);  // symbolized as "a". x_2 here is 0, 72, 144, …
  ...          // perform read and precharge. Read as "b", Precharge as "c"
  x_2 += 8;
 Activate(register x_2);  // this must be "d". x_2 here is 8, 80, 152, …
  ...          // perform read and precharge
  x_2 += 64;
}
```

그림 3.1: Cycle Equivalency 예시 패턴

제 4 장 기호 변환

4.1 제안하는 기호 변환 방식

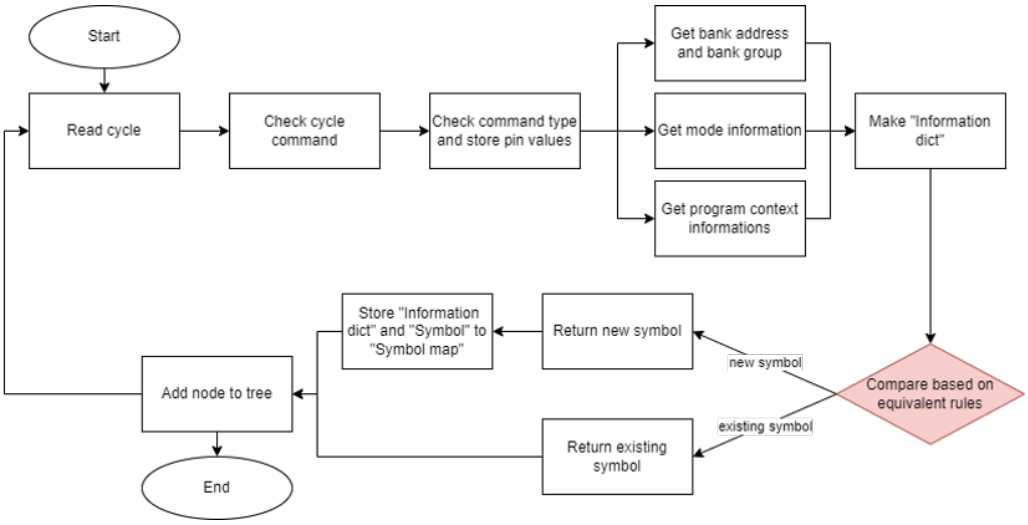

그림 4.1: Symbol conversion flow chart

그림 4.1은 symbol conversion의 전체 flow chart를 보여주고 있다. 먼저 사이클 핀 값을 분석하여 해당 사이클의 커맨드 타입(1-cycle-command, 2-cycle-command)을 판단한다. 2-cycle-command의 경우 pair-command 핀 값도 저장한다. 커맨드 종류와 핀 값을 통해서 해당 사이클 정보인 bank address, bank group, 모드, program context등의 정보를 추출해내게 된다. 추출해 낸 사이클 정보들을 기존에 저장해 둔 사이클 symbol들의 사이클 정보와 equivalent rule을 기반으로 비교하여 해당 사이클의 symbol을 부여한다. 부여된 symbol과 사이클 정보는 다시 symbol 맵에 저장하고 해당 symbol을 트리에 삽입한다. 해당 과정을 반복함으로써 IR을 symbolic representation으로 변환할 수 있다.

4.2 Equivalence Rule

3.1에서 기술하였듯이 symbol conversion의 핵심은 cycle equivalency를 판단하는 작업이다. 사이클의 output vector 정보와 program context를 종합적으로 고려하여 두 사이클이 동일한 작업을 수행한다 라고 판단할 수 있는 기준을 만들었다. 이를 Equivalence Rule 이라고 정의하였고 이에 기반하여 3단계에 걸쳐 symbol 변환이 이루어진다.

4.2.1 Concrete command name 판단

첫번째 단계에서는 output bit vector를 분석해서 커맨드의 종류와 bank 정보, 커맨드 특성에 따라 설정 가능한 mode가 일치하는지 확인한다. 커맨드 종류 판단은 먼저 해당 command가 1-cycle-command인지 2-cycle-command 인지를 판단하고 다음으로 activate, write, pre-charge 등의 커맨드 종류를 판단하게 된다. 2-cycle-command의 경우에는 앞 또는 뒤의 커맨드 위치 정보를 포함한다. 다음으로 커맨드에 따라 사용되는 bank address, bank group이 일치하는지 비교한다. 그리고 auto-precharge-mode 등 커맨드 특성에 따라 설정 가능한 mode가 일치하는지 확인한다. 일반적으로 패턴 프로그램 상에서 커맨드의 종류와 bank address, bank group, mode 정보를 함축한 형태로 표현하는데, 본 논문에서는 이러한 형태의 표현을 concrete-command-name 이라 정의하였다. 예를 들어 커맨드 종류가 Write이고 2-cycle-command의 앞 위치에 해당하고, bank address, bank group이 각각 A, 0, auto-precharge-mode가 on되어있는 경우의 concrete-command-name은 A0WRA1으로 표현할 수 있다. 이러한 표현 방식은 현재 개발 진행중인 분석 tool의 visualization 단계에서 활용되며, 사용자가 cycle 정보를 직관적으로 이해하는데 도움이 된다.

4.2.2 세부 정보 판단

다음 단계는 커맨드의 종류에 따라 row address, column address 등의 비교가 필요한 특정 pin 값들을 다룬다. 특정 필요 핀 값이 0 또는 1의 지정값이 아닌 register 할당일 경우 마지막 단계인 프로그램 context 고려 단계에서 확인한다. Command 종류가 write 또는 read 일 때는 추가적으로 사용하는 data 값을 비교한다. Data 비교 시에는 burst mode에 따른 데이터 량을 고려하여 분석해야 한다.

4.2.3 프로그램 context 고려

마지막 단계는 프로그램 context 를 고려하는 단계로 cycle equivalency 판단의 어려움이자 핵심이다. 다시 말해 해당 cycle command 가 사용하는 register의 register equivalency를 판단하는 것이다. Register equivalency 판단은 cycle command에 영향을 주고 있는 모든 register operation 과 커맨드 사이클의 register state 를 종합적으로 분석하는 것으로 진행된다. Register operation의 구성으로는 operator, operation loop type, init value, delta value, specified value가 있다. Register state 는 type, init value, delta value 로 구성된다.

표 4.1은 2개의 서로 다른 예시 register operation "$x_1 < x_1 + 1$"과 "$x_1 < 0$"의 구성 내용에 대해 보여준다. "$x_1 < x_1 + 1$" 처럼 매 루프마다 일정 변화량을 증가 또는 감소시켜주는 경우 operator 를 +, "$x_1 < 0$"와 같이 기존 값과는 무관하게 지정값으로 변경하는 경우 = 로 표기한다. 즉, + operator 에서는 init value와 delta value가 중요한 값일 것이고, = operator 에서는 specified value가 중요한 값일 것이다. Operation loop type은 해당 register operation이 일어나는 루프의 위치를 표현한다. 특정 커맨드에 영향을 주는 register operation을 종합하여 register operation의 구성 내용과 해당 커맨드 사이클의 register state 를 종합 분석함으로써 program context를 고려할 수 있다.

표 4.2에서 특정 사이클 커맨드에 영향을 주는 서로 다른 4가지 register operation set이 있을 때의 register equivalency 판단에 대한 간단한 예시를 확인할 수 있다.

Register operation example	”x_1 <x_1 + 1”	”x_1 <0”
Operator	+	=
Operation loop type	Inner	Outer
Init value	0	-
Delta value	1	-
Specified value	-	0

표 4.1: Register operation 예시

		Used register operation	
		Inner loop	Outer loop
Cycle command	A	”x_1 <x_1 + 1”	”y_1 <y_1 + 8”
	B	”x_1 <x_1 + 4”	”y_1 <y_1 + 8”
	C	”z_1 <z_1 + 1”	”y_1 <y_1 + 8”
	D		”x_1 <x_1 + 1”, ”y_1 <y_1 + 8”

표 4.2: Register equivalecy 판단 예시

편의상 inner loop과 outer loop의 iteration 횟수 x_1, y_1, z_1의 init value = 0 으로 같고 register operation은 loop의 마지막 사이클에 위치해 있다고 가정한다. A와 B는 inner loop에서의 x_1에 대한 register operation에서 변화량이 1과 4로 다르기 때문에 register equivalency 하지 않으며 A와 D는 register operation은 일치하지만 x_1에 대한 register operation이 1에서는 inner, 4에서는 outer loop에 위치해있어 register equivalency 하지 않다. A과 C는 register 이름은 다르나 register operation 각각의 operator, operation loop type, init value, delta value 가 다 일치하므로 register equivalency 관계에 있다고 판단할 수 있다.

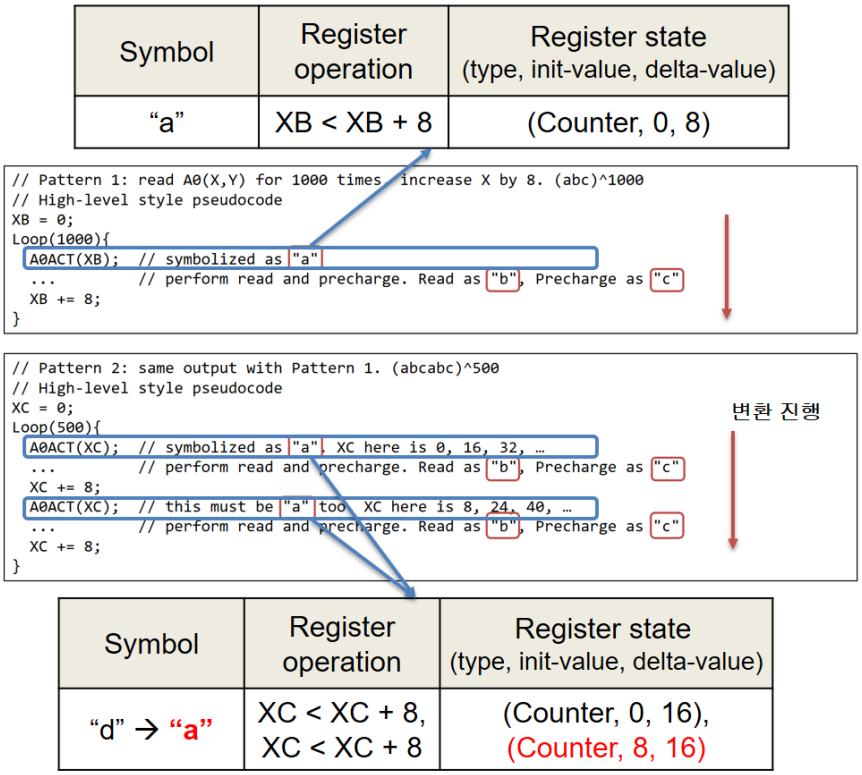

그림 4.2: Program context 고려 예시

그림 4.2는 그림 3.1 의 cycle equivalency 예시 패턴 중 패턴 1, 2 에서의 program context 고려를 설명한다. 패턴 1의 activate 명령 사이클은 매 루프 레지스터 xb가 8씩 증가하는 1개의 register operation의 영향을 받고 있으며 counter 타입, init value 0, delta value 8 의 register state을 가진다. type은 레지스터가 매 루프 변화할 때는 counter, 지정 값으로 정해질 때는 buffer 라고 정의한 parameter 로써 register operation 구성 요소 중 operator 와 의미가 비슷하다. 패턴 2의 ACT 명령 사이클의 경우 2개의 register operation 에 영향을 받는데, 변환이 진행됨에 따라 처음에는 패턴 1의 register operation과 register state 모두 다르기 때문에 'd' 의 새로운 symbol로 변환된다. 변환이 계속 진행됨에 따라, 두번째 ACT 명령 사이클 변환을 하게 되면 앞선 A0ACT 사이클과 cycle equivalency 1, 2 단계가 일치하기 때문에 현재 'd'로 변환된 심볼의 심볼 맵에 접근하게 된다. 이 때 register operation이 $XC < XC + 8$로 2개가 동일하게 있으며 기존에 있던 register state 중 마지막 state의 init value와 현재 변환이 진행되는 사이클의 register state, init value의 차이가 register operation 상의 변화량 8로 일치한다. 이는 루프를 접었다고 생각하면 패턴 1과 같은 구조가 된다는 것을 의미하기 때문에 패턴 1의 심볼 "a"에 해당하는 레지스터 내용과 일치하다고 판단할 수 있고 모든 "d" symbol은 "a" symbol로 update 된다.

이렇게 register operation 정보와 변환이 진행됨에 따라 변화하는 register state 정보를 종합하여 분석하는 것으로 프로그램 context를 고려할 수 있다. 해당 예시의 경우 register name 이 XB, XC 로 다르지만 register context 를 고려해 동일한 심볼로 변환한 것이다.

4.3 분석 도구

패턴 프로그램을 symbolic representation으로 변환하는 과정에서 분석도구를 통해 정보들을 추출한다. 이를 통해 패턴의 루프 구조와 커맨드 실행 순서를 알 수 있다. 그리고 패턴 실행 시 루프에 따른 register operation, 변화하는 register 값이 사용되는

커맨드 종류 정보를 가져올 수 있다. 추출한 정보들을 통해 엔지니어들이 line by line 으로 패턴을 읽지 않아도 해당 패턴의 작업 의도를 쉽게 파악할 수 있다.

제 5 장 실험 결과

5.1 실험 결과

```
initialize register values
for(i=1;i<=64;i++):
    x_1 <- 0
    y_1 <- y_1 + 16
    for(j=1;j<=4096;j++):
        x_1 <- x_1 + 1
        ...
        Activate(1) bank address 0 and bank group 0
        Activate(2) bank address 0 and bank group 0
        Activate(1) bank address 0 and bank group 1
        Activate(2) bank address 0 and bank group 1
        ...
        Write(1) bank address 0 and bank group 0
        Write(2) bank address 0 and bank group 0
        Write(1) bank address 0 and bank group 1
        Write(2) bank address 0 and bank group 1
        ...
module end
```

그림 5.1: 예제 프로그램의 C 기반 pseudo 코드

제품 수준의 3개 테스트 패턴 프로그램 예제에 대해서 결과 분석의 용이함을 위하여 루프 구조를 기반으로 총 21개의 벤치마크로 나누어 SR 생성 작업을 실험하였다. 그림 5.1은 실험에 사용된 한 벤치마크를 추상화하여 표현한 한 예제 패턴 프로그램의 C-like pseudo 코드이다. 패턴 프로그램 특성상 커맨드 사이 간격 제약 조건들이 발생하는데 이를 DES(De-select command) 사이클의 삽입으로 조절한다 [5]. 그림 5.1의 pseudo 코드에서는 편의상 DES 사이클을 생략하였다.

Cycle number	Symbol	Command
0	0x1	DES
1	0x1	DES
2	0x1	DES
3	0x1	DES
4	0x2	Activate(1) BA 0 and BG 0
4	0x3	Activate(2) BA 0 and BG 0
5	0x1	DES
6	0x1	DES
7	0x4	Activate(1) BA 0 and BG 1
7	0x5	Activate(2) BA 0 and BG 2
8	0x1	DES
9	0x1	DES
10	0x1	DES
11	0x6	Write-a(1) BA 0 and BG 0
11	0x7	Write-a(2) BA 0 and BG 0
12	0x1	DES
13	0x1	DES
14	0x8	Write-a(1) BA 0 and BG 1
15	0x9	Write-a(2) BA 0 and BG 1
16	0x1	DES
17	0x1	DES
18	0x1	DES
19	0x1	DES
20	0x1	DES
21	0x1	DES
22	0x1	DES
23	0x1	DES

그림 5.2: 예제 프로그램의 기호 변환 결과

그림 5.2는 예제 프로그램에 대하여 기호 변환을 수행한 결과이다. 편의상 bank address는 BA, bank group은 BS로 표현하였다. Bank group의 차이, 커맨드 종류 등의 차이로 서로 다른 기호가 부여되었다. 기호 변환이 완료된 트리 형태 SR은 그림 5.3과 같은 모습이다. Outer loop이 64번, inner loop이 4096번 iteration하며 inner loop에는 다음과 같은 커맨드들이 순차적으로 실행됨을 보기 쉽게 알 수 있다.

모든 벤치마크에 대해서 기호 변환이 올바로 적용됨을 확인할 수 있었다. 표 5.1에는 벤치마크들 중 14개의 파트로 분할된 한 프로그램의 파트 별 기호 개수가 기술되어 있다. 해당 파트 패턴의 경우 2개의 파트가 pair로 (ex, part1, part2) 같은 작업을 수행하고 있는데 변환된 symbol 개수도 pair인 파트가 같음을 확인할 수 있고 기호 변환이 올바로 되었음을 확인할 수 있었다.

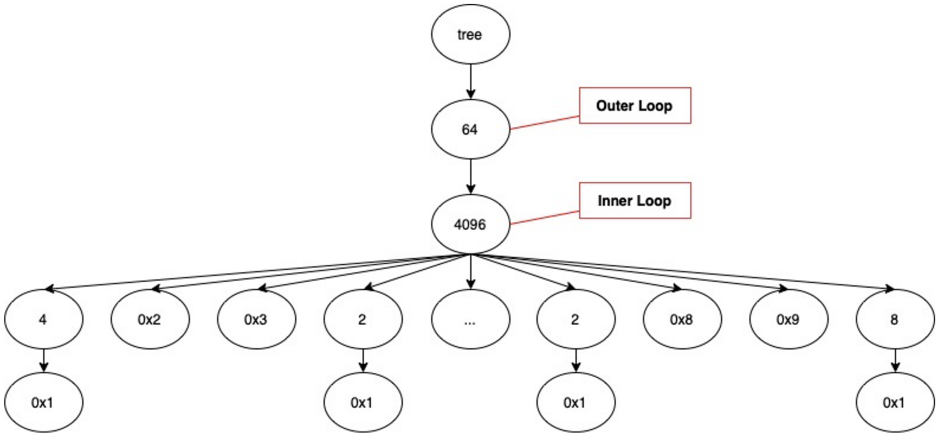

그림 5.3: 예제 프로그램의 symbolic representation tree

패턴 프로그램							
파트	1	2	3	4	5	6	7
Symbol 수	129	129	129	129	129	129	193
파트	8	9	10	11	12	13	14
Symbol 수	193	161	161	129	129	193	193

표 5.1: 한 예제 패턴 프로그램의 파트 별 기호 수

제 6 장 관련 연구 및 SR 활용 사례

6.1 관련 연구

6.1.1 패턴 프로그램 변환 플랫폼

[5]에서는 출력 기반 중간 형태를 활용하여 테스트 패턴 프로그램 변환 플랫폼인 PaTran을 구축하였다. PaTran은 사용자가 별도의 클라이언트 설치 없이 웹 브라우저에서 활용 가능하다.

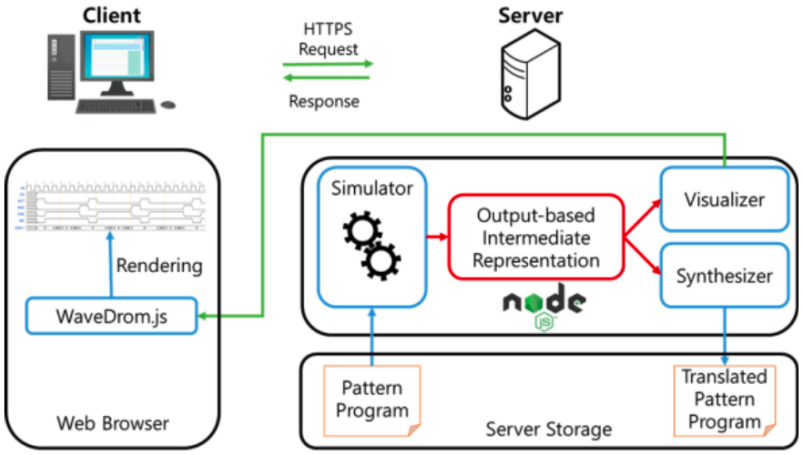

그림 6.1: 변환 플랫폼 전체 구성

그림 6.1에서 테스트 패턴 프로그램의 변환 과정을 살펴볼 수 있다. 첫 단계로 simulator 모듈에서 대상 패턴 프로그램을 출력 기반 중간 표현 형태로 변환하고, synthesizer 모듈에서 출력 기반 중간 표현 형태를 target ATE 환경에 맞게 재구축한다. 이러한 과정은 Node.js[6]를 기반으로 구축한 서버에서 진행된다.

해당 변환 플랫폼의 visualizer 모듈은 출력 기반 중간 표현 형태를 사용자가 구동

하는 웹페이지에서 렌더링할 수 있는 데이터로 변환해준다. 사용자는 WaveDrom.js API[8]를 통해 IR의 출력 신호에 해당하는 결과를 waveform 형태로 확인할 수 있다.

6.2 SR 활용 사례

6.2.1 SR 기반 Visualization 툴

Symbolic representation tree를 기반으로 하여 Analysis 결과를 쉽게 확인할 수 있는 Visualization 툴 개발이 진행되고 있다. 그림 6.2, 6.3, 6.4는 Prototype을 보기 쉽게 형상화 한 모습이다.

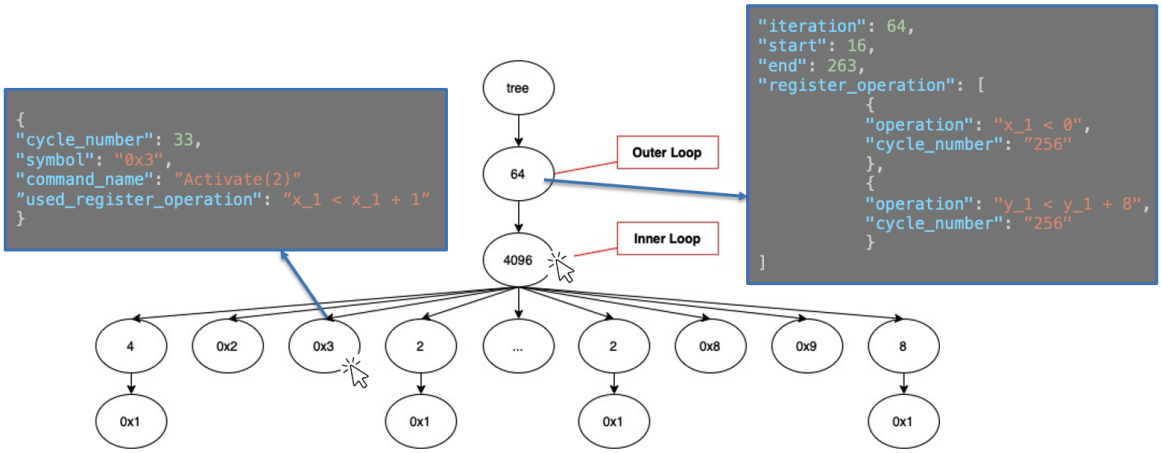

그림 6.2: Visualization tool horizontal ver

사용자는 loop head에 해당하는 node를 선택했을 때, 해당 루프에 대한 정보를 제공받거나 그림 6.2의 왼쪽처럼 inner loop 내부의 0x3 symbol에 해당하는 node를 선택했을 때 해당 symbol의 사이클 넘버, 커맨드 이름, used-register-operation과 같은 정보를 제공받을 수 있다. Node 값은 symbol 이외에도 앞서 정의한 concrete-command-name의 형태로 변환 가능하여 직관적으로 구조를 살펴볼 수 있다.

Symbolic representation으로의 변환 과정에서 각 사이클에서 추출한 정보들은 해당 사이클이 변환된 symbol과 함께 symbol map에 저장되는데, 이를 활용하면 좀 더 다양한 기능들을 visualization tool에 구현할 수 있다.

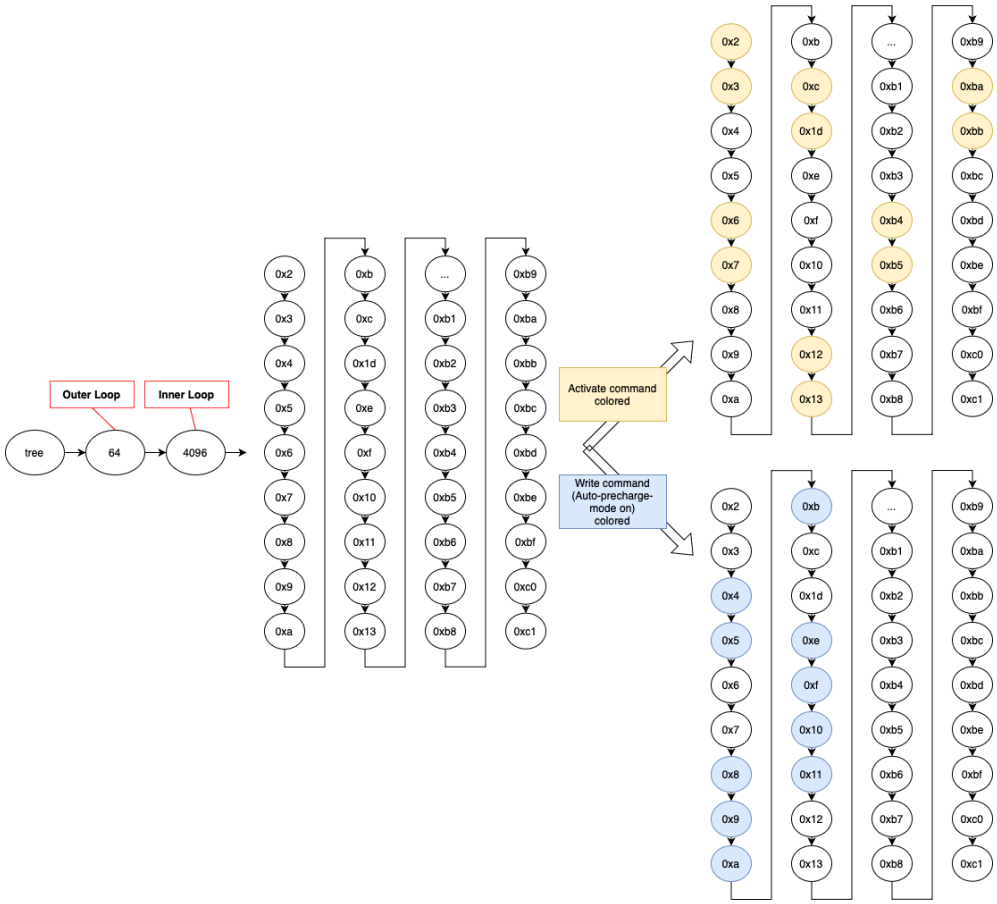

그림 6.3: Visualization tool vertical ver - specific command colored

예를 들어 그림 6.3과 같이 Activate command를 수행하는 사이클 또는 Write command 중 Auto-precharge mode가 on된 사이클은 테스트 패턴 프로그램 상에서 어떠한 분포로 위치해있는지 쉽게 시각화하여 확인할 수 있다. 또는 특정 bank ad-

dress, bank group에서 작동하는 사이클의 분포를 확인하여 해당 패턴 프로그램이 activate, write, pre-charge 커맨드가 순서대로 이루어지는지, activate와 write가 서로 교차되면서 파이프라인 형태를 가지며 이루어지는지 등의 구조 파악이 가능하며, 특정 명령어 사이의 사이클 딜레이가 얼마나 되는지 등의 정보를 추출함에 있어서도 접근하기 용이해진다.

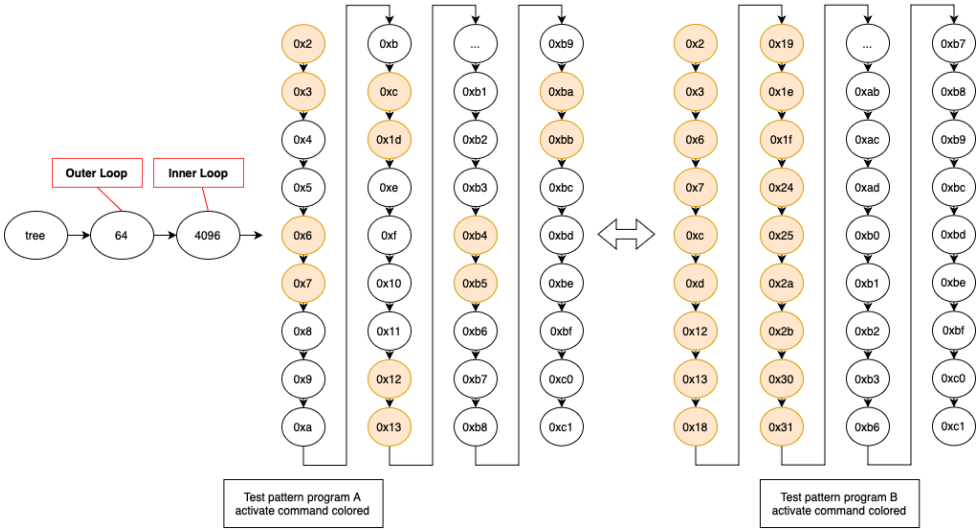

그림 6.4: Visualization tool vertical ver - comparison of two pattern programs

추가적으로 서로 다른 패턴 프로그램 A, B에 대해서 symbol map을 공유한 상태로 symbolic representation 변환을 하고, 앞선 기능을 적용한다면 그림 6.4에서 볼 수 있듯이 패턴 프로그램 A에서의 특정 심볼 그룹의 분포와 패턴 프로그램 B에서 동일 심볼 그룹의 분포 차이를 확인할 수 있고, 이를 통해 두 패턴 프로그램 간의 유사도나 구조 차이를 직관적으로 살펴볼 수가 있다. 진행 상황으로는, Symbolic Representation과 Analysis Data가 담긴 json 파일을 기반으로 d3.js library와 VS Code의 Live Server extension을 통해 local 환경에서 배포하여 visualize 할 수 있다. Visualization tool은 6.1.1의 PaTran 패턴 프로그램 변환 플랫폼과 연동하여 활용할 수 있도록 계

획하고 있다.

제 7 장 결론 및 향후 과제

본 논문에서는 테스트 패턴 프로그램의 변환에 사용되었던 output based IR을 활용하여 각 사이클을 binary symbol로 변환한 symbolic representation을 고안하였다. 각 사이클의 output bit vector에서 추출할 수 있는 정보들과 프로그램 구조를 분석해서 동일한 작업을 수행하는 사이클은 같은 symbol로 변환한다. Cycle equivalency를 판단하는 것은 SR 변환의 핵심이며 정교하게 설정한 equivalence rule을 기반으로 진행된다. 또한 분석 도구를 통해 엔지니어들의 패턴 프로그램 이해에 도움을 줄 수 있는 정보를 추출한다.

$$\mathtt{abcbcbcdabcbcbcdabcbcbcd}$$
$$\downarrow$$
$$(a(bc)^3d)^2a(bc)^3d$$
$$\downarrow$$
$$(a(bc)^3d)^3$$

그림 7.1: loop nesting example

IR의 symbolic representation 변환은 패턴을 symbol이 나열된 string으로 변환하는 것과 같다. 이는 패턴 프로그램의 크기를 최적화하는데 많은 도움이 된다. 간단한 예시로 그림 7.1과 같이 symbolic representation으로 변환된 패턴은 가장 loop nesting이 많이 되는 형태를 알아내기 용이하다. ATE에는 패턴 프로그램을 업로드하는 메모리 크기에 제한이 있기 때문에 패턴 프로그램의 줄 수를 더 짧게 만드는 것이 의미가 있다.

그리고 서로 다른 두 패턴 프로그램이 있을 때 두 패턴이 동일한지, 유사하다면 어떤 부분이 얼마나 유사한지를 string comparison 문제로 치환하여 해결할 수 있다. 이러한 동일성 비교 모델링은 test coverage 체크를 통한 효율적인 테스트 진행을 가

능하게 할 수 있다. 또한 나아가서 패턴 프로그램의 fault model을 역추론하는 도구 개발로 활용될 수 있다.

참고 문헌

[1] Comfort, J.J., Hayter, P.A., Kargathra, D., Mason, B.R., Turner, G.N., Fisher, I.R. and Bailey, J.W.,“Automatic Test Equipment,” 1987, U.S. Patent No. 4,807,161. 1987.

[2] IEEE Computer Society,“IEEE Standard Test Interface Language (STIL) for Digital Test Vector Data — Core Test Language,”Vol. 1999.

[3] S. McGinty, D. Hadad, C. Nappi, and B. Caquelin,“Developing a modern platform for test engineering - Introducing the Origen semiconductor developer’s kit,”2015 IEEE International Test Conference (ITC), pp. 1-10, Anaheim, CA, 2015.

[4] 박성철, 김동윤, 김동건, 박세호, 이상훈, 박제영, “이종 ATE 간 공용 라이브러리를 활용한 TEST 방법,” 한국반도체테스트학회 2019.

[5] Park, Jung-Geun, et al. ”Patran: Translation platform for test pattern program.” 2019 IEEE European Test Symposium (ETS). IEEE, 2019.

[6] Node.js Foundation, “Node.js,” Node.js, 2020. [Online]. Available: https://nodejs.org, [Accessed: Oct. 15, 2020].

[7] A. Chapyzhenka and J. Probell. “WaveDrom: rendering beautiful waveforms from plain text,” in Synopsys User Group (SNUG) Silicon Valley 2016 Proceedings. Accessed on: Oct. 15, 2020. [Online]. Available:https://wavedrom.com/images/SNUG2016-WaveDrom.pdf

[8] M. Kim et. al, “Output-based Intermediate Reprentation for Translation of Test-pattern Program,” ACM Trans. Embed. Comput. Syst., Vol.18, no.5s, Article 55, October 2019.

ABSTRACT

Test pattern programs for testing DRAM memory have different programming languages depending on Automatic Test Equipment (ATE) and are low in readability. In this paper, a new type of symbolic representation for the analysis of test pattern programs is presented. Each cycle pin value is analyzed from the output-based intermediate representation, which is the simulation result, to extract the corresponding cycle information, and based on this, the cycle performing the same task is converted into the same symbol. Symbol phenotype conversion was performed with 21 benchmarks based on commercial test pattern programs, and it was confirmed that the intended result was derived. In addition, the analysis tool using the symbolic representation helps users to intuitively understand the structure and work intention of the pattern program. The symbolic representation of the test pattern program can be used for optimization of the pattern program, equality test when converting the pattern program, and fault model detection.

주요어: DRAM, Test Pattern Program, Symbolic Representation, Intermediate Representation

학번: 2021-22071