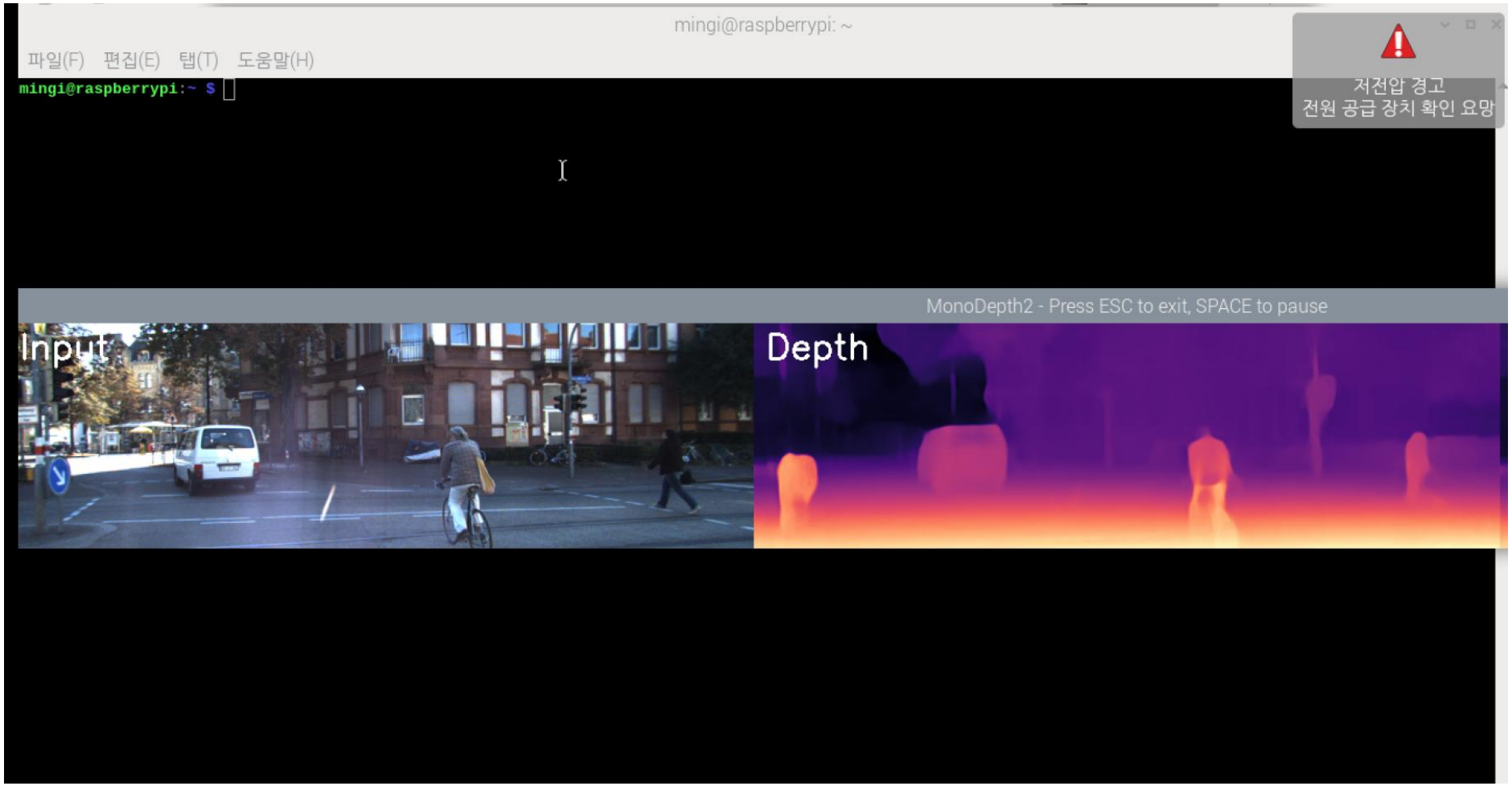
Converting Libtorch Operation to TVM Operation for Optimization

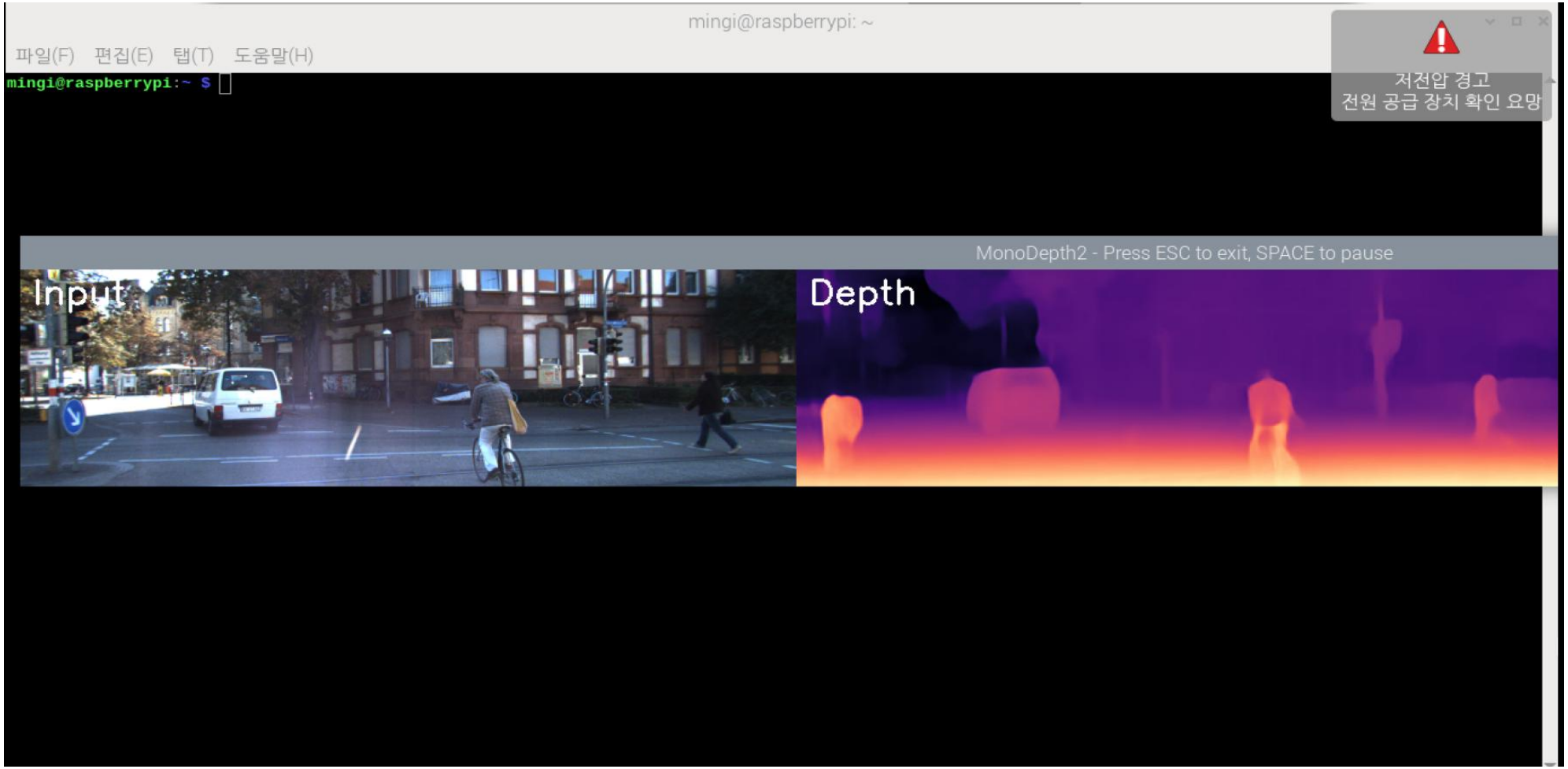
2025/12/14

김민기

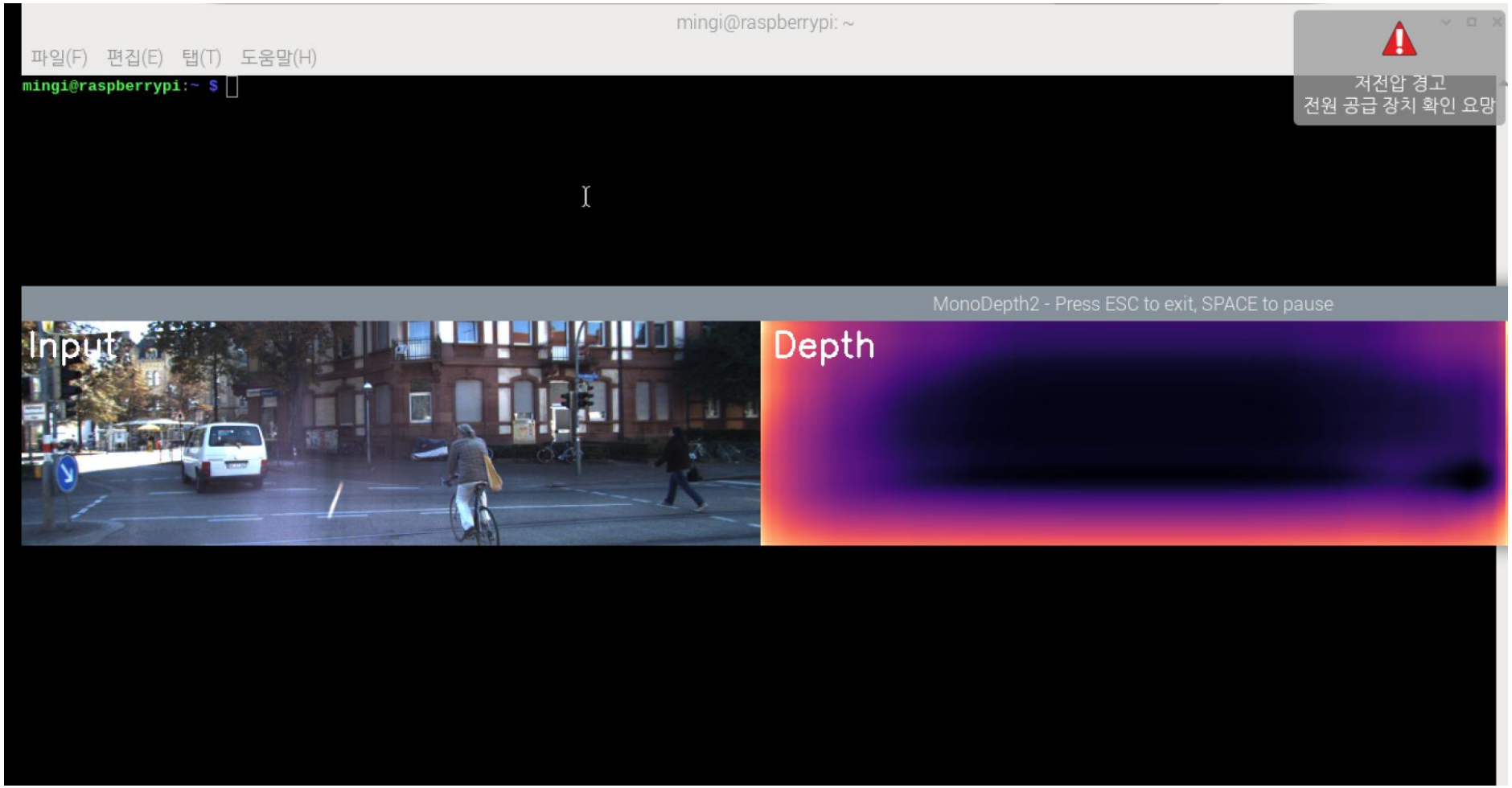
TEST NUMBER	CASE	INFERENCE TIME(ms)
1	OpenCV + Libtorch	2637
2	OpenCV + TVM(from_pytorch function)	488
3	OpenCV + TVM(program -> relay)	501

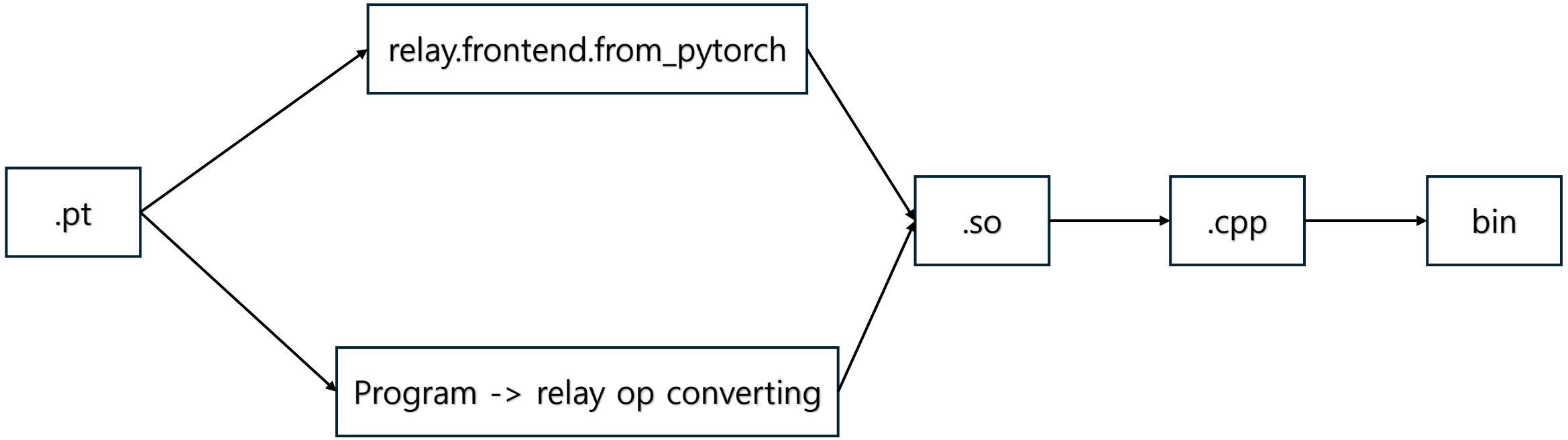


TEST NUMBER	CASE	INFERENCE TIME(ms)
1	OpenCV + Libtorch	2637
2	OpenCV + TVM(from_pytorch function)	488
3	OpenCV + TVM(program -> relay)	501



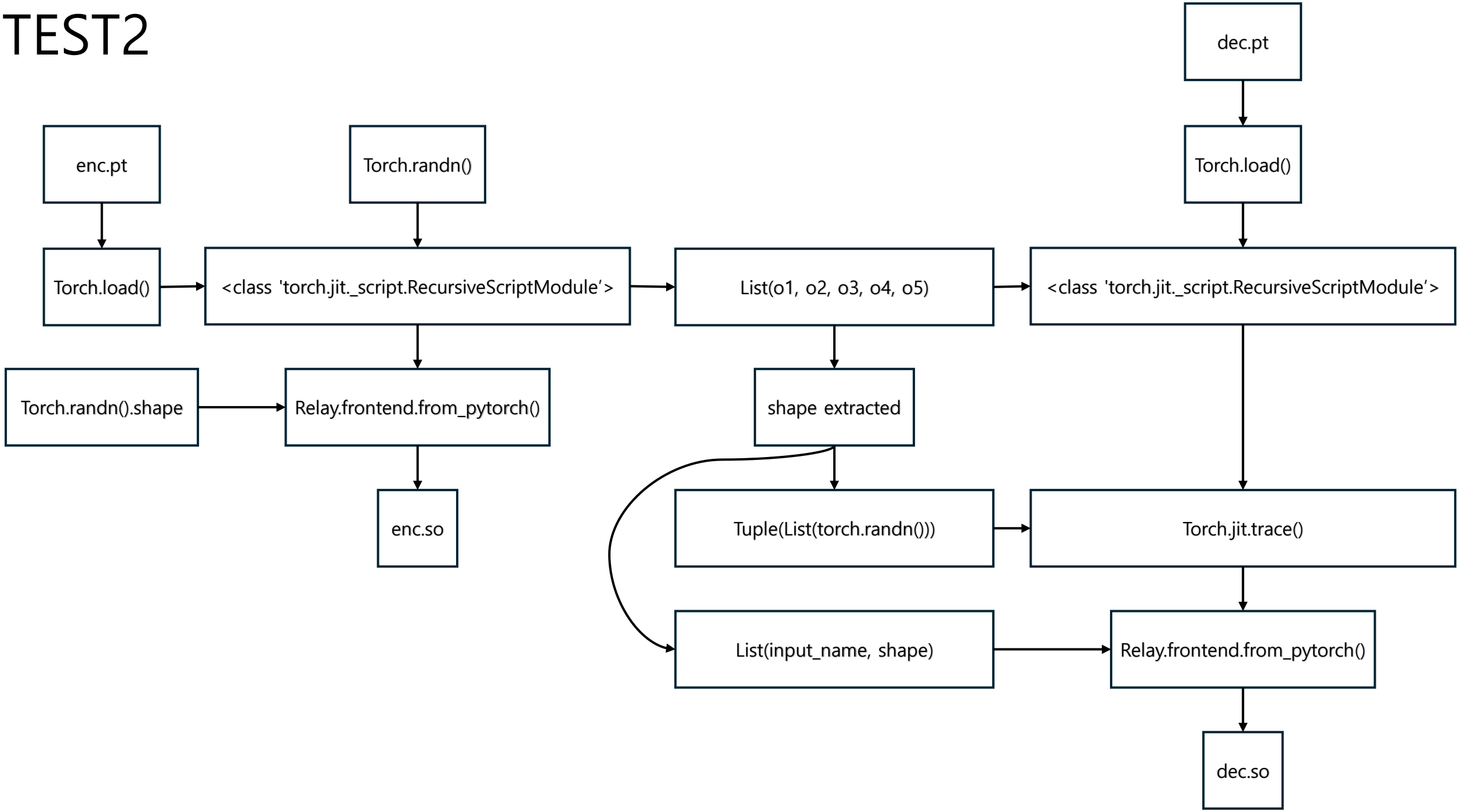
TEST NUMBER	CASE	INFERENCE TIME(ms)
1	OpenCV + Libtorch	2637
2	OpenCV + TVM(from_pytorch function)	488
3	OpenCV + TVM(program -> relay)	501



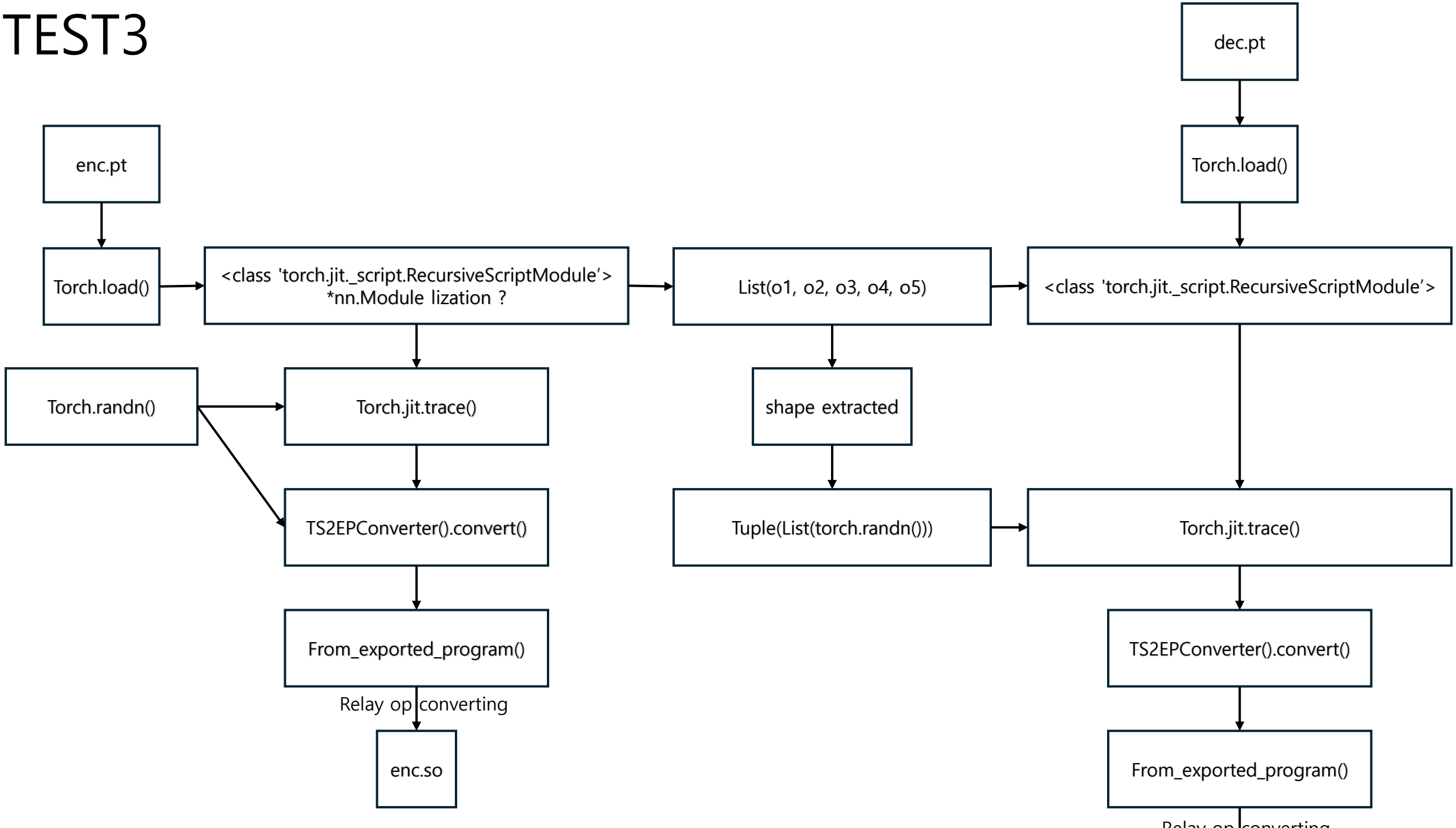


- TEST1, TEST2는 결과가 같은데, TEST3만 결과가 다르게(이상하게)나온다.
- TEST1, TEST2는 만들어 놓은 것을 가져다 쓰는 것이고 TEST3은 custom하는 작업이 들어간 것
- TEST2, TEST3에서의 변환/연산 과정을 톺아보자

TEST2

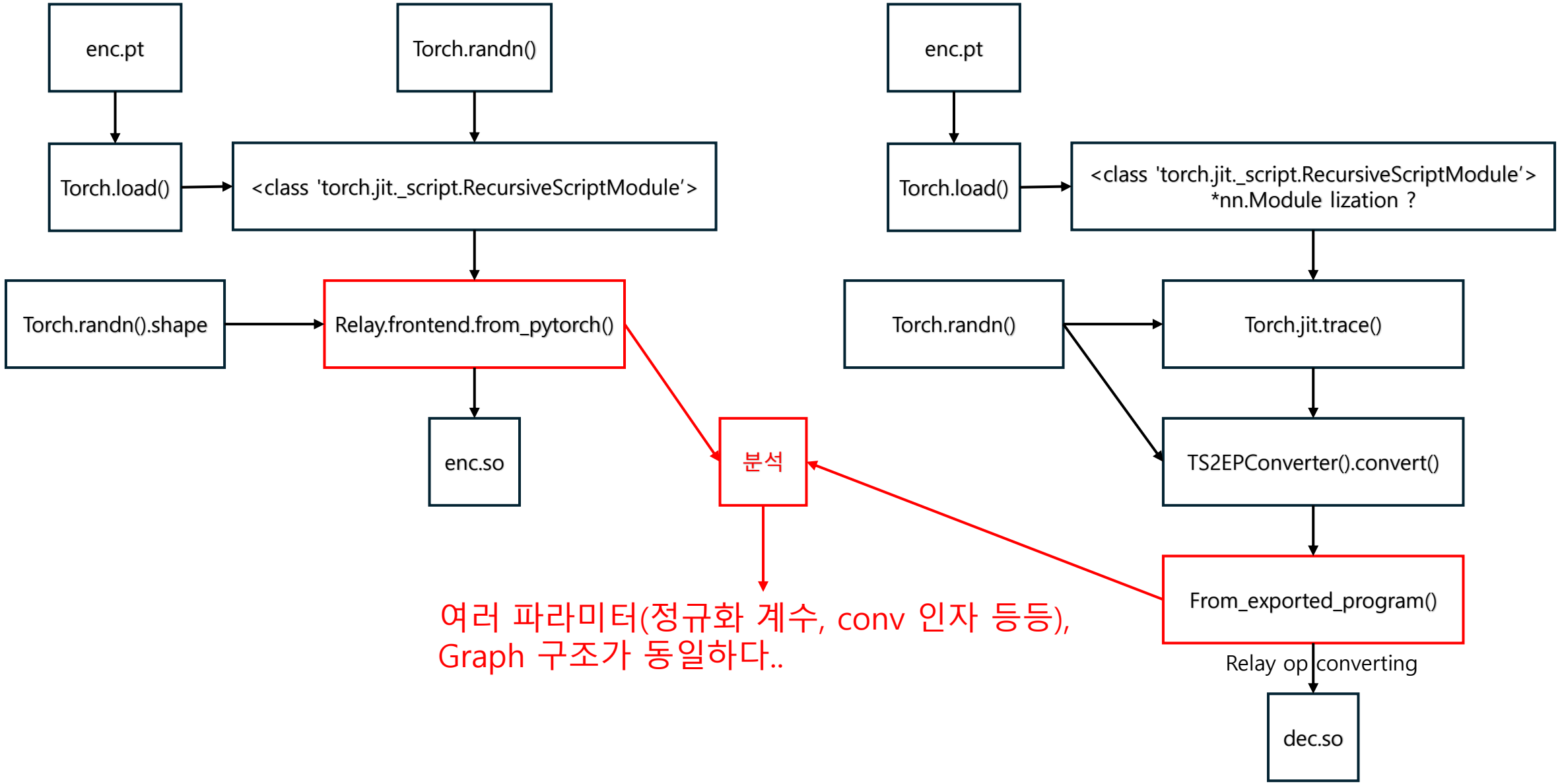

dec.pt
enc.pt
Torch.randn()
Torch.load()
Torch.load()
<class 'torch.jit._script.RecursiveScriptModule'>
List(o1, o2, o3, o4, o5)
<class 'torch.jit._script.RecursiveScriptModule'>
Torch.randn().shape
Relay.frontend.from_pytorch()
shape extracted
enc.so
Tuple(List(torch.randn()))
Torch.jit.trace()
List(input_name, shape)
Relay.frontend.from_pytorch()
dec.so

TEST3

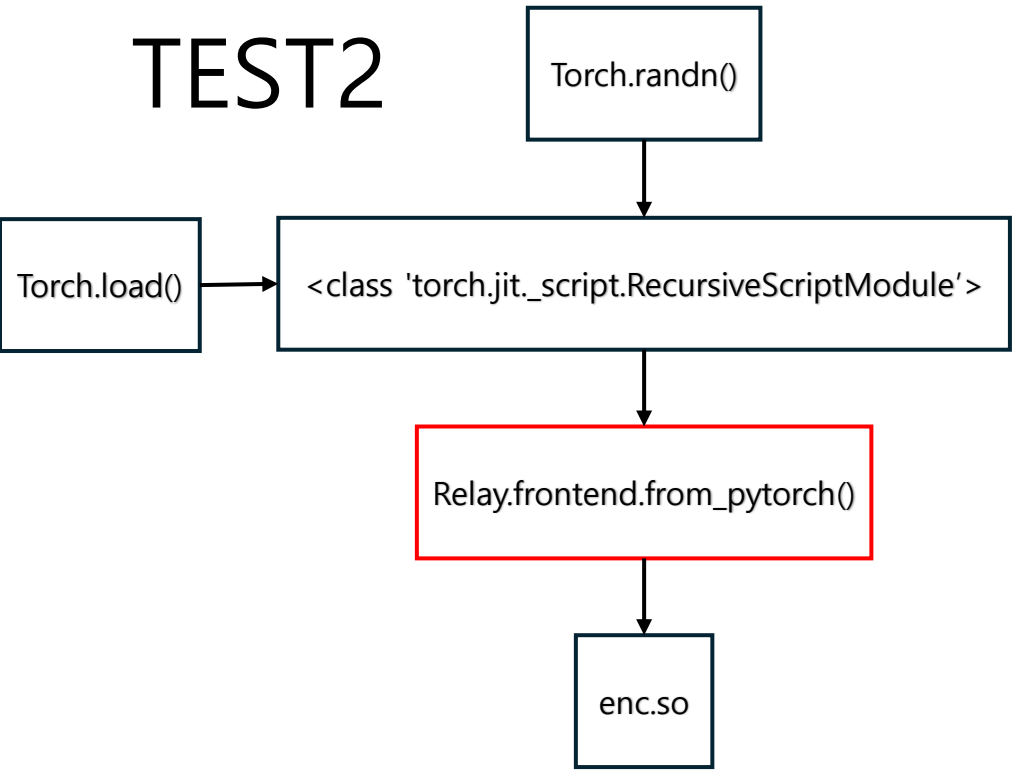

dec.pt
Torch.load()
enc.pt
Torch.load()
<class 'torch.jit._script.RecursiveScriptModule'>
*nn.Module lization ?
List(o1, o2, o3, o4, o5)
<class 'torch.jit._script.RecursiveScriptModule'>
Torch.randn()
Torch.jit.trace()
shape extracted
TS2EPConverter().convert()
Tuple(List(torch.randn()))
Torch.jit.trace()
From_exported_program()
TS2EPConverter().convert()
Relay op converting
enc.so
From_exported_program()

TEST2

TEST3


enc.pt
Torch.randn()
Torch.load()
<class 'torch.jit._script.RecursiveScriptModule'>
Torch.randn().shape
Relay.frontend.from_pytorch()
enc.so
분석
여러 파라미터(정규화 계수, conv 인자 등등),
Graph 구조가 동일하다..
enc.pt
Torch.load()
<class 'torch.jit._script.RecursiveScriptModule'>
*nn.Module lization ?
Torch.randn()
Torch.jit.trace()
TS2EPConverter().convert()
From_exported_program()
Relay op converting
dec.so

- TEST2, TEST3은 그래프 구조가 동일하지만, 그래프를 캡쳐하는 방식에 차이가 있어서 결과값이 다르게 도출되는 것이라고 생각됨.
- 그렇다면, 두 개의 방법에서 그래프를 캡쳐하는 방식이 어떻게 다를까 ?



- Torch.jit.ScriptModule은 이미 컴파일된 모델로 Torch jit ir 그래프를 내부에 보관
  - .graph: forwar의 torch jit ir
  - .inlined_graph: 함수호출, submodule을 전부 펼친 그래프
  - State_dict(): Parameter / Buffer
  - Prim::GetAttr: submodule/weight 접근

- Torch jit ir을 받아서 PytorchOpConverter를 통해 relay op 연산으로 변환
  - 이때, torch jit ir은 재실행을 수행하지 않는, 이미 정해진 정적 그래프이기 때문에 relay op 변환 과정에서 추가적인 weight folding, constant 승격이 일어나지 않음
  - 따라서 pytorch op -> relay op 변환을 수행할 때, 1:1로 정확하게 매핑 수행 가능

- torch.jit.ScriptModule에 대해 더 자세하게 알아보자
  - Torch.jit.ScriptModule은 이미 컴파일된 모델로 Torch jit ir 그래프를 내부에 보관
    - Python AST가 아니다
    - Eager execution을 수행하지 않는다. IR로 표현하여 구조로써 존재한다.
    - Control flow를 해석하지 않는다. IR로 표현하여 구조로써 존재한다.
    - SSA 기반의 중간 표현으로 lowering한 결과물이다.

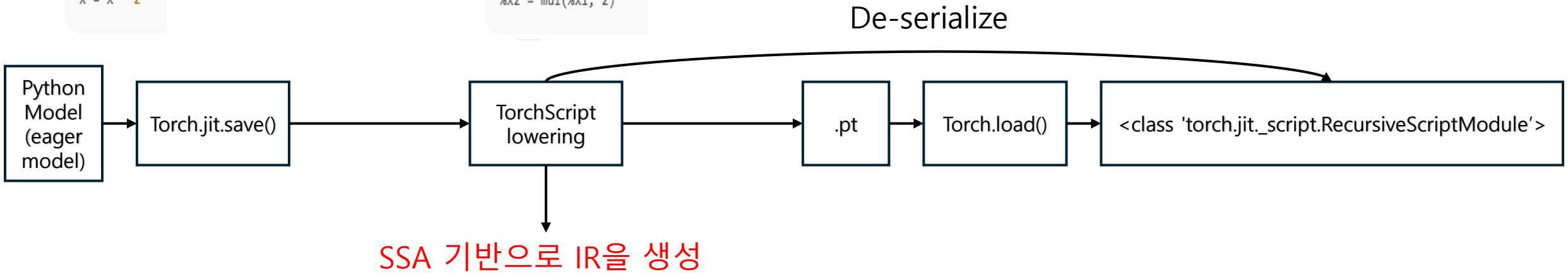
```python
def f(x):
    if x.sum() > 0:
        return x * 2
    else:
        return x / 2
```

- Eager Execution에 대해 더 자세하게 알아보자
  - Python의 기본 실행 모델은 eager execution
  - 오른쪽과 같은 코드가 있다고 했을 때, 해당 파일을 실행하는 시점에 변수가 할당 되고 즉시 실행된다.
  - 하지만, 이것은 컴파일러 관점에서는 안좋다.
    - 실행 전에는 연산을 결정할 수 없고, 입력 값에 따라 연산 구조가 바뀐다. 즉, 그래프가 입력에 의존적
  - 따라서 컴파일러 레벨에서 이러한 한계를 해결하기 위해 Script Module은 모든 연산의 구조를 실행하지 않고, 기술(description)한다.

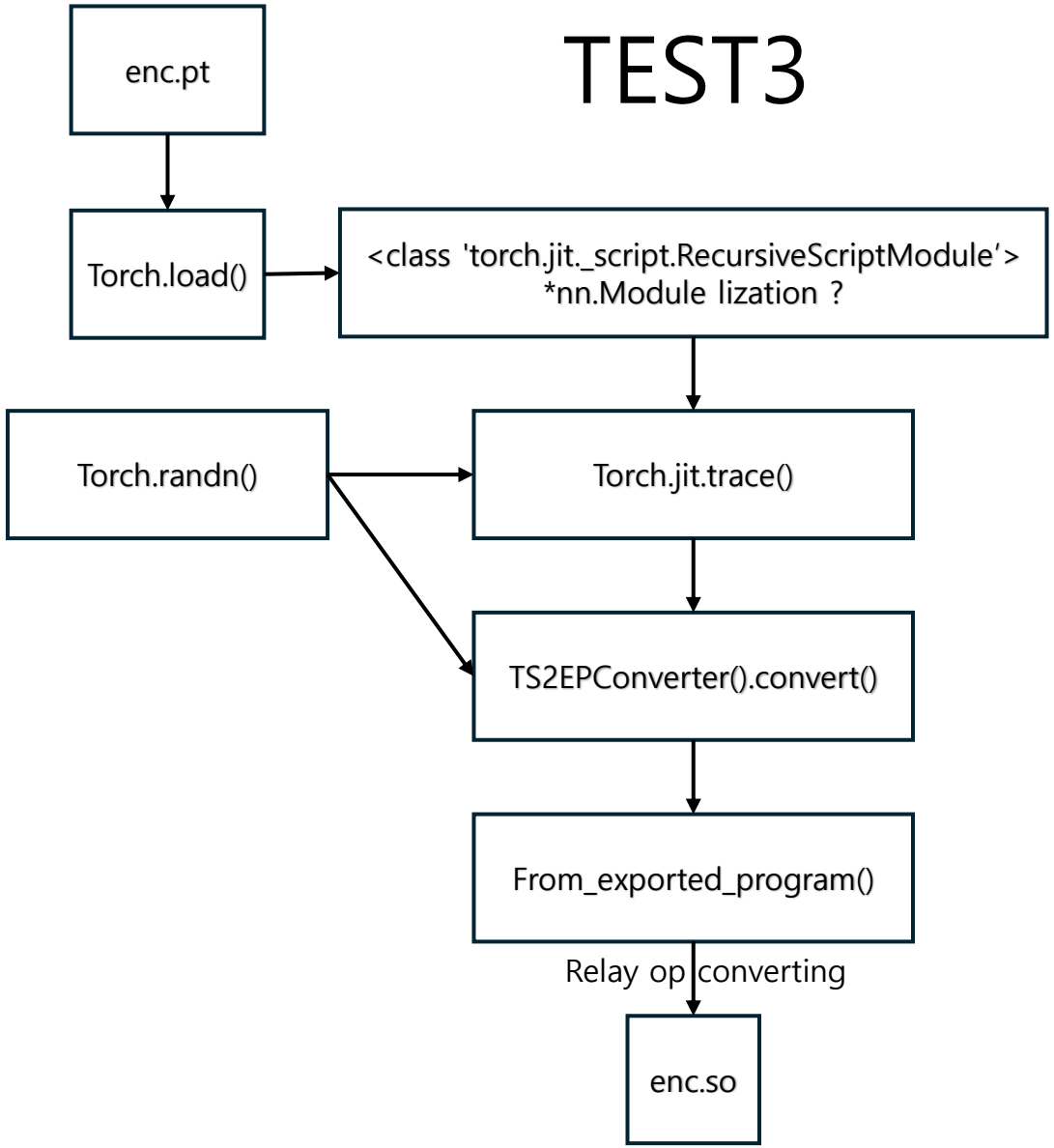
- SSA(Static Single Assignment)는 무엇인가?
  - 각 변수를 정확하게 한 번만 할당하게됨
  - 그렇기 때문에 데이터의 의존성을 분석할 수 있고, dead code elimination, operator fusion, 정적 분석 등이 가능해짐.

```python
x = x + 1
x = x * 2
```

```text
%x1 = add(%x0, 1)
%x2 = mul(%x1, 2)
```



TEST3



- Torch.jit.trace()함수는 module.forward(*inputs)를 실행하게됨
  - 이때, 실행 중 호출된 Aten operator를 순서대로 기록하여 새로운 jit graph를 생성
- 이미 생성된 정적인 그래프를 굳이 다시 실행시켜서 IR graph로 만들고 있었기 때문에 그래프 연산 구조가 바뀔 수 있음

- TS2EPConverter는 relay.frontend.from_pytorch() 함수와 다르게 1:1로 pytorch op -> relay op 매핑을 수행하지 않는다.
  - 여러가지 제약을 둠으로써 의미 차이를 명확하게 고착화 시킴
- (2), (3), (4)에서 기존 연산과 다르게 변형 가능성이 있음

```mathematica
(1) TorchScript Graph 추출
(2) Graph signature 재구성
(3) Parameter / Constant 분리
(4) ATen op graph로 lowering
(5) ExportedProgram 검증
```

Future Work

- TVM에서 최적화할 수 있는 방법들은 크게 다음과 같다.
  - Relay Pass(IR-Level 최적화)
  - TIR(Loop/Memory 최적화)
  - Codegen(Backend-specific Lowering)
- 위의 주제들 중 IR-Level 최적화를 먼저 해볼 예정