Lab05. Multi-cycle CPU with Sequential Control

전기·정보공학부 김선우 (2019-16380)

1. Introduction

본 실습에서는, Multi-cycle 방식으로 동작하는 CPU를 설계하고 Verilog로 구현한다. State machine, ROM을 사용하는 sequential 방식의 Control module을 갖는 것이 설계의 특징이다. 설계할 microarchitecture는 TSC ISA의 총 23개 명령어를 실행해야 하며, CPU 내 1개의 ALU만 이용해 다음 Program Counter(PC) 값에 대해 다음 PC 값을 구하는 연산을 수행해야 한다.

표 1. TSC ISA 명령어 실행 단계(stage)

<table>
<tr><th>Format</th><th>Instruction</th><th>Stage①</th><th>Stage②</th><th>Stage③</th><th>Stage④</th><th>Stage⑤</th></tr>
<tr><td rowspan="12">R</td><td>ADD</td><td rowspan="25">IF</td><td rowspan="25">ID</td><td rowspan="8">EX</td><td rowspan="8">WB</td><td rowspan="20">-</td></tr>
<tr><td>SUB</td></tr>
<tr><td>AND</td></tr>
<tr><td>ORR</td></tr>
<tr><td>NOT</td></tr>
<tr><td>TCP</td></tr>
<tr><td>SHL</td></tr>
<tr><td>SHR</td></tr>
<tr><td>JPR</td><td colspan="2" rowspan="4">-</td></tr>
<tr><td>JRL</td></tr>
<tr><td>WWD</td></tr>
<tr><td>HLT</td></tr>
<tr><td rowspan="10">I</td><td>BNE</td><td rowspan="9">EX</td><td rowspan="3">-</td></tr>
<tr><td>BEQ</td></tr>
<tr><td>BGZ</td></tr>
<tr><td>BLZ</td><td rowspan="6">WB</td></tr>
<tr><td>ADI</td></tr>
<tr><td>ORI</td></tr>
<tr><td>LHI</td></tr>
<tr><td>LWD</td><td rowspan="2">MEM</td><td>WB</td></tr>
<tr><td>SWD</td><td rowspan="3">-</td></tr>
<tr><td rowspan="2">J</td><td>JMP</td><td colspan="2" rowspan="2">-</td></tr>
<tr><td>JAL</td></tr>
</table>

실행해야 할 명령어의 목록과, IF->ID->EX->MEM->WB로 실행 단계를 구분할 때 각 명령어를 최소 사이클로 실행하기 위해 어떤 단계를 진행해야 하는지를 표 1에 나타냈다. 예를 들어, IF, ID 단계는 모든 명령어의 실행에 포함되는 것이고, JPR 명령어는 "IF->ID"의 2 사이클만에 실행하므로 최소 사이클, LWD 명령어는 "IF->ID->EX->MEM->WB"의 5 사이클만에 실행하므로 최대 사이클이 소요된다. 명령어의 구체적인 형식과 동작 사항은 교부된 "tsc_ISA.pdf" 파일을 참고했다.

각 명령어를 명시된 최소 사이클 내에 실행하기 위한 microarchitecture를 설계하기 위해, CPU 전체 회로도에 대한 CPU 모듈과 각 Datapath 모듈들, 그리고 CPU의 매 단계를 인코딩해 각 단계에 따른 control signal을 출력하고 clock signal에 동기적으로 상태를 전이할 sequential한 Control 모듈을 설계해야 한다.

실습을 통해 multi-cycle, sequentially controlled CPU의 동작을 구체적으로 이해하고, sequentially-controlled microprocessor를 설계하고 디버깅하는 연습을 할 수 있다.

2. Design

1) 전체 회로도 설계

구현할 microarchitecture의 전체 회로는 다음 그림 1과 같다. 하나의 ALU로 명령어 내 여러 개의 연산을 실행하기 위해(hardware resource reuse) 각 단계의 연산 순서를 명시적으로 구분하고 값을 저장한다. 이를 위해 buffer(A, B, ALUOut, Memory Data Register; MDR)와 multiplexor(ALUSrcA, B 선택 등)를 사용했다. 또, 디버깅 및 출력을 위한 num_inst, halt signal과 Out 모듈을 포함했다.

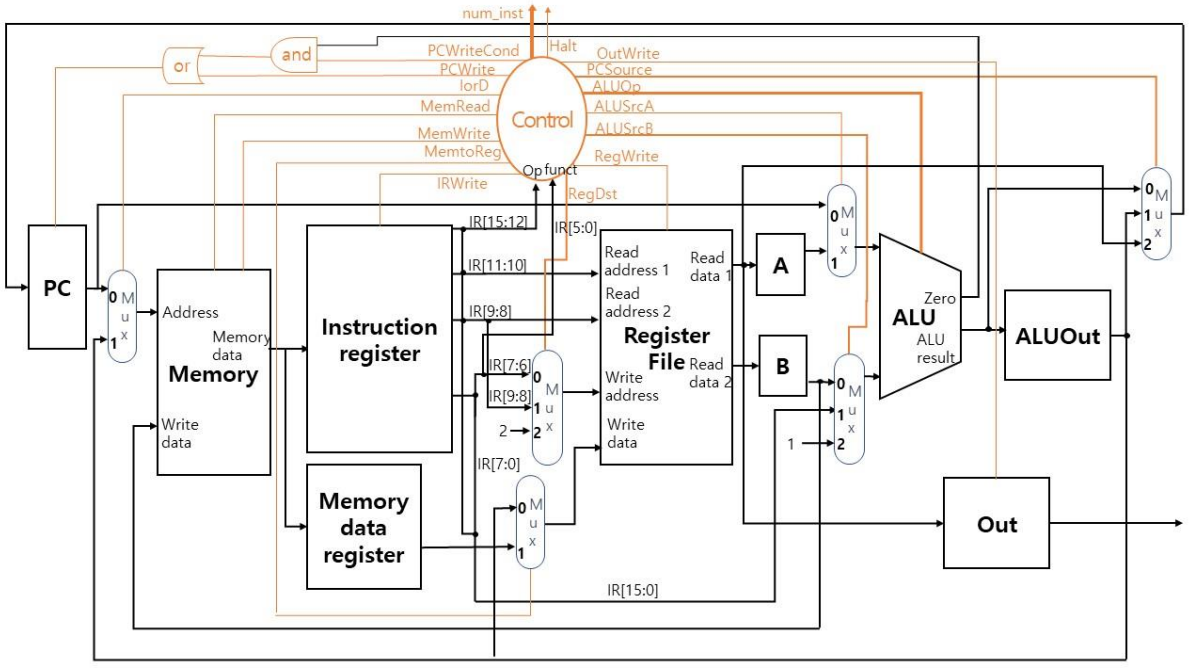

그림 1 CPU 전체 회로도(clock, reset 신호 제외)

2) 명령어 분류, Register transfer 분석 및 control signal 지정

전체 23개 명령어는 R, I-type arithmetic 명령어 11개(ADD, SUB, AND, ORR, NOT, TCP, SHL, SHR, ADI, ORI, LHI), Branch 명령어 4개(BNE, BEQ, BGZ, BLZ), 점프 명령어 4개(JPR, JRL, JMP, JAL), 메모리 명령어 2개(LWD, SWD), 기타 디버깅 명령어 2개(WWD, HLT)로 분류할 수 있다. 이렇게 명령어들을 종류에 따라 분류해, 각각의 실행 단계와 각 단계에서 필요한 register transfer 및 대응하는 control signal을 정확히 구할 수 있다. 이를 다음 표 2에 나타냈다. 이름을 볼드체로 표시한 명령어는 해 단계에서 실행이 완료되는 명령어이다. IR은 Instruction Register을 의미한다.

표 2. 명령어 실행 단계(stage)마다 수행할(necessary) register transfers, 필요한 control signals

Stage / Instruction		Register Transfers (necessary transfers)	Control Signals (not mentioned: set to low)
IF		IR<-MEM[PC], PC<-PC+1	IorD=0("I"), MemRead=1, IRWrite=1, ALUSrcA=0("PC"), ALUSrcB=2("1"), ALUOp="A+B", PCSource=0("ALUresult"), PCWrite=1
ID	R, I-type arithmetic	A<-RF[IR[11:10]], B<-RF[IR[9:8]]	-
	Branch	ALUOut<-PC+sign-extend(IR[7:0])	ALUSrcA=0("PC"), ALUSrcB=1("IR"), ALUOp="A+sign-extend(B[7:0])"
	JMP	PC<-{(PC-1)[15:12], IR[11:0]}	ALUSrcA=0("PC"), ALUSrcB=1("IR"), ALUOp="concat((A-1)[15:12], B[11:0])", PCSource=0("ALUresult"), PCWrite=1
	JAL	RF[2]<-ALUOut , PC<-{(PC-1)[15:12], IR[11:0]}	RegDst=2("2"), MemtoReg=0, RegWrite=1, ALUSrcA=0("PC"), ALUSrcB=1("IR"), ALUOp="concat((A-1)[15:12], B[11:0])", PCSource=0("ALUresult"), PCWrite=1
	JPR	PC<-RF[IR[11:10]]	PCSource=2("RFread_data1"), PCWrite=1
	JRL	RF[2]<-ALUOut, PC<-RF[IR[11:10]]	RegDst=2("2"), MemtoReg=0, RegWrite=1, PCSource=2("RFread_data1"), PCWrite=1
	WWD	(none)	OutWrite=1
	HLT	(none)	Halt=1
	LWD, SWD	A<-RF[IR[11:10]], B<-RF[IR[9:8]] (SWD)	-

EX	R-type arithmetic	ALUOut<-alu_op(A, B) (e.g. ALUOut<-A+B)	ALUSrcA=1("A"), ALUSrcB=0("B"), ALUOp="ALUoperation"
	I-type arithmetic	ALUOut<-alu_op(A, imm) (e.g. ALUOut<-A+sign-extend(IR[7:0]))	ALUSrcA=1("A"), ALUSrcB=1("IR"), ALUOp="ALUoperation"
	Branch	PC<-ALUOut (iff branch taken, else none)	ALUSrcA=1("A"), ALUSrcB=0("B"), ALUOp="Branch_operation", PCWriteCond=1, PCSource=1("ALUOut")
	LWD, SWD	ALUOut<-A+sign-extend(IR[7:0])	ALUSrcA=1("A"), ALUSrcB=1("IR"), ALUOp="A+sign-extend(B[7:0])"
MEM	LWD	MDR<-MEM[ALUOut]	IorD=1("D"), MemRead=1
	SWD	MEM[ALUOut]<-B	IorD=1("D"), MemWrite=1
WB	**R-type arithmetic**	RF[IR[7:6]]<-ALUOut	MemtoReg=0, RegDst=0("IR[7:6]"), RegWrite=1
	I-type arithmetic	RF[IR[9:8]]<-ALUOut	MemtoReg=0, RegDst=1("IR[9:8]"), RegWrite=1
	LWD	RF[IR[9:8]]<-MDR	MemtoReg=1, RegDst=1("IR[9:8]"), RegWrite=1

Register transfer 열에 표기된 것은 register 이름으로, 각 단계에서는 당시 register의 값을 기준으로 연산한다. 예를 들어, Branch 명령어가 ID 단계에서 "ALUOut<-PC+sign-extend(IR[7:0])"할 때의 PC 값은, 앞 IF 단계에서 "PC<-PC+1"을 하고 난 결과의 PC 값으로 연산한다.

이렇게 실행할 때, 각 명령어에서 그 다음 PC 값이 다음과 같이 맞게 연산됨을 확인할 수 있다.

R, I-type arithmetic: next PC <= PC + 1

Branch: next PC <= PC + 1 + sign-extend (immediate)

JMP, JAL: next PC <= {PC[15:12], target}

JPR, JRL: next PC <= $rs

표 2와 같이 명령어별로 단계를 지정하면 Introduction에서 명시한 실행 사이클 제한을 만족하고, ISA에 명시된 것처럼 PC 및 Register file, Memory의 업데이트가 이루어지므로, 이와 같이 상태를 인코딩하고 Control 모듈을 설계한다.

3) Datapath 설계

그림 1과 같이 datapath를 구성하는 각 모듈들을 설계한다. 메모리의 동작을 모델링한 파일은 따로 주어지므로 PC, IR, MDR, Register File, A, B, ALU, ALUOut, Out 모듈을 설계하면 된다. 여기서 A, B, ALUOut은 control signal 없이 동기적으로 latch하도록 설계하고, PC와 Register File은 control signal에 따라 동작하도록 설계한다(지난 실습 참고).

주어진 메모리는 읽기 상황에서 readM(그림 1의 MemRead를 인가) signal이 High일 때 일정 read_delay(<half clock cycle) 이후 입력한 주소에 저장된 값을 data 포트로 출력하고, 쓰기 상황에서 writeM(그림 1 MemWrite 인가) signal이 High일 때 일정 write_delay(<half clock cycle) 이후 입력한 주소에 data 포트로 입력한 값을 저장한다. 이때, 값을 주고받는 것이 끝난 다음에는 readM, writeM signal을 다시 Low로 지정해야 다음 번에도 값을 읽거나 쓸 수 있음에 유의해야 한다.

Single-cycle implementation과 달리 multi-cycle implementation에서는 메모리에 접근하는 단계가 명시적으로 구분되므로, IF와 MEM 단계에서만 readM, writeM에 High를 인가하고, 그 다음 단계에 다시 Low를 인가해 메모리 접근을 제어한다. 메모리가 별도의 stable time 없이 동작하므로, High를 인가한 IF, MEM 단계 동안은 delay time 이후 한 사이클 내내 메모리와 값을 주고받는 것이 유효하고, 사이클이 끝난 다음 단계에는 메모리와 값을 주고받는 것이 없다고 생각할 수 있다.

따라서 메모리와 값을 주고받는 것이 실행 단계(CPU의 상태)에 따라 완전히 결정된다. Control signal 또한 상태에 따라 완전히 결정되므로, IR과 MDR에 값을 쓰는 것은 control signal을 이용해 제어한다. IR은 IRWrite이 High일 때, 내부 reg 변수에 입력 값을 대입한다. MDR은 inputReady signal을 이용하는데, 메모리 접근 명령어의 MEM 단계에서만 값을 업데이트하도록 IRWrite는 Low, inputReady는 High일 때 입력 값을 대입한다.

ALU 모듈은, 표 2에서 필요한 연산을 모두 지원하도록 연산 종류를 정한다. 다음 PC 값을 정할 때 필요한 모든 연산을 지원해야 하므로, single-cycle로 구현했을 때보다 연산 종류가 증가한다.

Out 모듈은, OutWrite signal이 High일 때 입력 값을 곧바로 출력, Low라면 z 값을 출력한다.

3. Implementation

1) Control 모듈 구현

Design 단계에서 구한 표 2와 같이, 가능한 CPU 상태를 인코딩하고 Finite state machine 형식으로 상태 전이를 나타낸다. 또, 각 상태에 따라 register transfer에 대응하는 output signal을 출력한다. 가능한 상태를 5-bit로 인코딩하고 출력 신호를 20-bit로 인코딩한다면, 2^5 * 20 = 80B 크기의 ROM으로 구현할 수 있다. 가독성을 위해 case 문을 이용한다. 상태에 따라 출력 신호를 지정할 때, 사용하지 않는 경우 memory signal과 PC, RF write signal은 Low로 지정하는 것에 유의한다.

상태 인코딩은 IDLE 1개, HALT 1개, IF 1개, ID 1개, R, I-type arithmetic instruction EX 11개, Branch EX 4개, Load EX 1개, Store EX 1개, Load MEM 1개, Store MEM 1개, R-type arithmetic instruction WB 1개, I-type arithmetic instruction WB 1개, Load WB 1개로 총 26개, 5-bit로 인코딩했다. 표 2의 register transfer과 다음 전이할 상태에 따라 분류했고, "states.v" 파일에 매크로로 선언했다.

이를 바탕으로 "control.v" 모듈을 작성했다. Clk, reset_n, opcode, function_code를 입력으로 받고 Halt, PCSource, PCWrite, PCWriteCond, IorD, MemRead, MemWrite, IRWrite, RegDst, MemtoReg, RegWrite, ALUSrcA, ALUSrcB, ALUOp, OutWrite, num_inst의 control 및 디버깅 신호를 출력한다. 모듈 내부적으로 present_state, next_state로 상태를 나타내는 reg 변수를 사용하고, 출력 신호의 값을 always 문 내부에서 대입하기 위해 대응하는 reg 변수를 각각 선언해 할당하고, 사용했다. Initial 상태는 present_state에 IDLE, 모든 출력 신호에 0을 대입했다. 다음 상태의 값과, 상태에 따른 control 신호를 구하는 부분을 combinational circuit으로 작성했고, 동기적으로 상태를 전이하는 부분을 sequential circuit으로 작성했다. Num_inst는 명령어가 새로 fetch되는 IF 단계에서 1씩 증가하도록 했고, ID 단계에서 명령어가 HLT일 경우 계속 HLT 상태에 멈춰 있도록 다음 상태에 현재 상태를 대입했다.

2) Datapath 모듈 구현

Design 단계에서 주장한 것과 같이, A, B, ALUOut은 control signal 없이 동기적으로만 latch하므로, Buffer 모듈을 정의해 그 인스턴스로 각 모듈을 구현했다.

PC, RF, IR, MDR은 control signal에 따라 동기적으로 입력 값을 내부 reg 변수에 쓰고, 비동기적으로 값을 출력하게 구현했다. IR의 출력 포트는 그림 1과 같이 IR[15:12], IR[11:10], IR[9:8], IR[7:0]을 각각 출력하고, 필요에 따라 각 신호를 묶어서 사용하도록 구성했다. IR, MDR에 값을 쓰는 시점은 clock이 하강하는 edge로 설정했는데, 이는 "constants.v" 파일에 명시된 것과 같이 read_delay가 clock cycle의 절반보다 작다(30<50)는 사실을 이용한 것이다. MemRead 등의 signal을 통해 현재의 단계가 메모리에서 값을 읽어오는 단계임을 안다면, clock의 하강 edge에서는 유효한 값이 준비되었음을 알 수 있기 때문이다. 만약 delay가 길어질 경우, inputReady signal을 사용할 수 있다.

ALU 모듈은 지원하는 연산을 모두 구해 내부 wire에 할당한 뒤, alucode 입력에 따라 대응하는 연산 결과를 출력한다. Branch 명령어가 아닌 경우 zero=0을 출력하고, Branch 명령어에서 ALU result는 z(high-impedance)를 출력한다.

Out 모듈은 tri-state buffer로, OutWrite signal이 High일 때만 $rs값이 출력되도록 구현했다.

모듈들을 "datapath.v" 파일에 정의했고, "CPU.v" 파일에서 그림 1과 같이 연결했다. 생략했던 clock, reset 신호는 모든 모듈에 연결했다. Wire들은 "w_"를 붙여 연결을 위한 wire임을 나타냈고, 모듈과 값에 따라 "o_pc", 혹은 "w_inst"과 같이 이름했다. Control signal들은 앞에 "c_"를 붙여 "c_iord"와 같이 이름을 붙였다. 전체 wire 이름을 명시한 전체 회로도를 그림 2에 나타냈다.

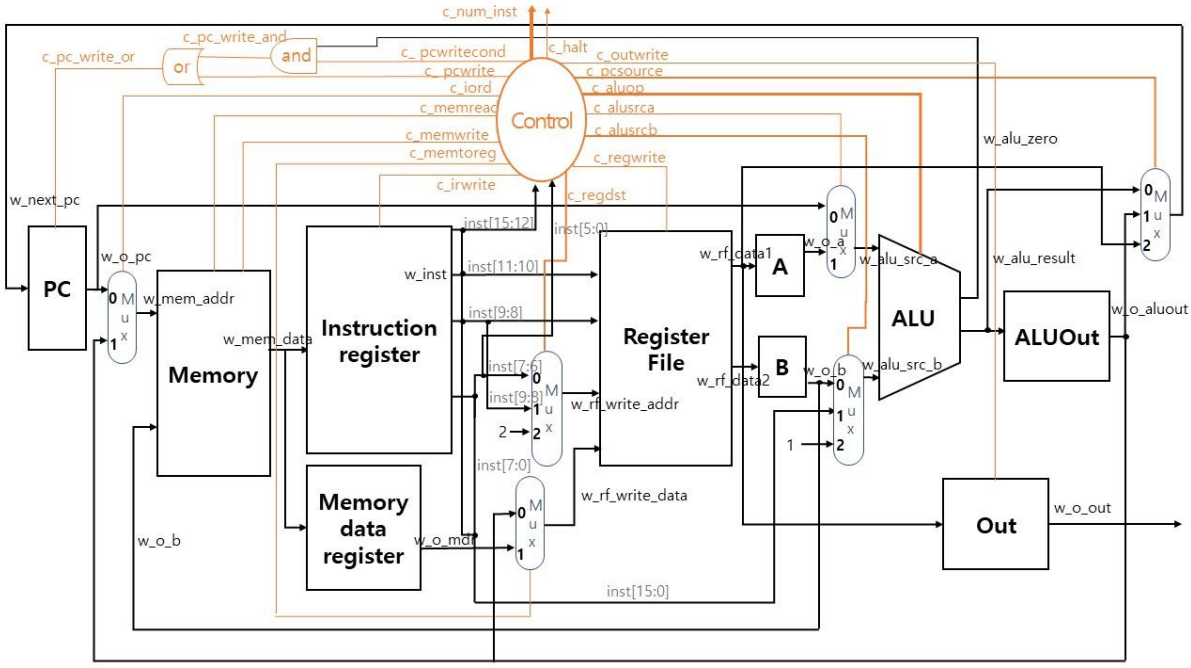

그림 2 CPU 모듈 wire 이름

4. Discussion

LWD 명령어의 WB 단계에서, 메모리의 값이 MEM 단계의 앞쪽 절반 사이클 내에 MDR에 대입된다. MDR 값의 업데이트를 clock이 하강하는 edge에서 한다면, 뒤이어 연속되는 WB 단계의 앞쪽 절반 동안은 읽어온 메모리 값이 MDR에 유지되므로 register write이 WB의 앞쪽 절반 동안에만 이루어진다면 MDR에 별도의 control signal을 인가하지 않고 clock signal만 이용해 MDR의 동작을 정상적으로 구현할 수 있다.

그러나, 실제로 실험했을 때는 WB 결과 register file에 Z(High-impedance) 값이 인가되었다. register write이 일어나는 시점이 WB 단계에 진입하는 상승 edge가 아니라 다음 단계로 전이하는 상승 edge에서 일어나기 때문에, 해당 시점에서는 이미 하강 edge가 한번 더 거쳐져 메모리와 연결이 끊어진, Z값이 MDR에 들어와 있기 때문이다. 이 문제를 해결하기 위해 register write을 clock cycle의 앞쪽 절반에서 하강 edge에 실행, read를 뒤쪽 절반에서 실행하는 방식을 고려해 보았지만(이후 internal forwarding도 염두에 두었다), 주어진 테스트벤치의 test 시점이 상승 edge이므로 output port의 값을 비교하는 시점이 적절치 않아 반려했다. MDR에 control signal을 인가하는 방법으로 문제를 해결했고, SWD 명령어의 MEM 단계에서만 MDR 값이 업데이트되도록 inputReady=High, IRWrite=Low에서 값을 업데이트하도록 했다. 이렇게 하면 원하는 register write 시점에 MDR 값이 유효하게 유지된다. Pipelined CPU를 구현하면서는 internal forwarding을 구현하기 위해, register write과 read을 clock의 앞, 뒤 절반에 수행하는 것을 다시 시도해 보겠다.

5. Conclusion

주어진 테스트벤치에서 "All Pass!"를 얻었다. 그림 1, 2의 회로도에 명시했듯이, 하나의 ALU로 필요한 연산 전체를 구현했다. (Instruction signal slicing은 ALU 없이도 간단히 가능하므로 제외)

마지막 1082번째 명령어인 HLT를 2 사이클만에 종료했을 때, 처음 IDLE부터 최종 HLT까지 총 3489 사이클이 소요되었다. 따라서, 본 테스트벤치로 구한 CPU의 IPC는 1082/3489 = 0.31이다.

복잡한 회로를 설계할 때, 강의 시간에 배운 것처럼 register transfer로 각 단계를 모델링하고 대응하는 control signal의 종류로 각 단계를 물리적으로 구분할 수 있음을 배웠다. 이 같은 순서를 통해, 체계적으로 시스템을 설계하고 분석하는 연습이 되었다.

Num_inst에 따라 test하는 시점의 문제로 test가 각각 2회씩 수행되어, 2번째 결과가 참이도록 타이밍을 정해 summary에서는 옳은 결과가 나오도록 구현했다. 이 같이 debugging/test logic을 회로에 도입할 때, 타이밍과 포트를 잘 맞춰 좋은 test를 만드는 방법론이 있는지 궁금하다.

Pipelined CPU 및 이후 실습을 통해, 다양한 방식으로 현재의 성능을 개선할 것이 기대된다.

*조교님께: Q&A 게시판 8번 질문에 대한 장하민 조교님의 답변에 따라, 마지막 결과가 옳게 나오므로 이번 실습에서는 test 횟수에 대해 더 이상의 조작을 하지 않았습니다. 감사합니다.