2025 Spring

CSED311

컴퓨터구조

Lab 4-1 report

Team ID: 15

팀원 1: 20230345 이성재

팀원 2: 20230355 정지성

목차

1. 서론

• How does our pipelined CPU work?

1. 디자인

• Stall detection design

* When stalled?

• Forwarding design

* When forwarded?

1. 구현

• Stall detection implementation

• Forwarding implementation

1. 논의 사항
2. 결론

• Comparison of total cycles between the single cycle and pipelined CPU (Non-control flow input file)

1. **서론**

Single-cycle CPU와 multi-cycle CPU 디자인의 큰 차이점들은 다음과 같다.

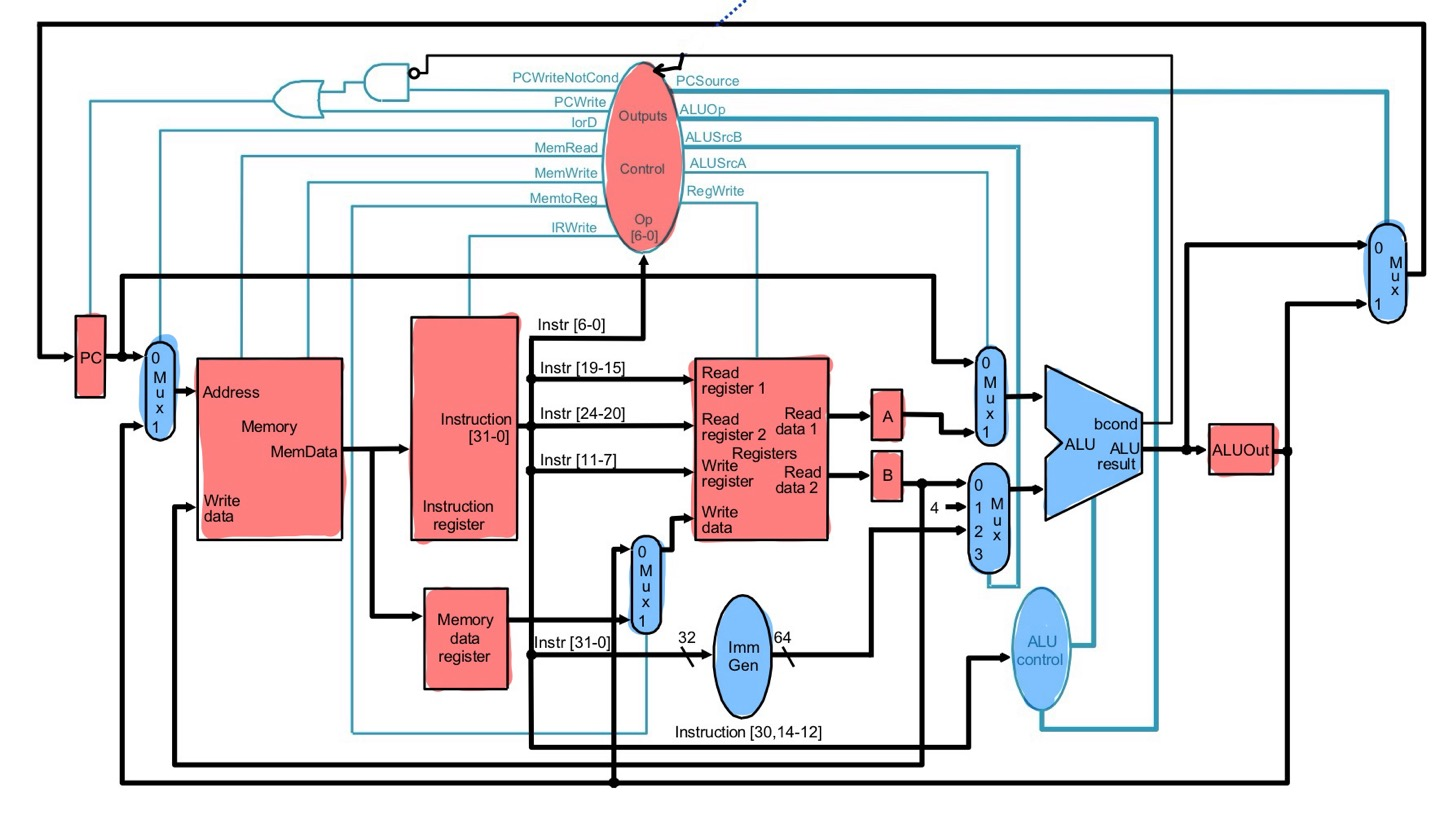
* Single-cycle CPU의 cycle는 가장 느린 instruction을 기준으로 맞추며, 한 instruction을 한 cycle에 처리한다. 그에 반해 multi-cycle CPU는 cycle의 시간을 감소하는 대신 하나의 instruction을 여러 cycle에 걸쳐 처리한다.
  + 이렇게 하면 빨리 끝나는 instruction에 대해서는 더 적은 cycle을, 늦게 끝나는 instruction에 대해서는 더 많은 cycle을 배정하여, 더 이상 소요 시간을 제일 느린 instruction에 맞추지 않아도 된다. 결론적으로 multi-cycle CPU의 instruction 당 평균 소요 시간이 더 작다.
* 한 instruction을 여러 cycle에 걸쳐 처리하므로, 하나의 모듈 (ALU, memory 등)을 한 instruction 안에서 여러 번 접근할 수 있다.
  + 즉 resource reuse가 가능해져, datapath의 길이를 줄일 수 있으며 공간 활용도도 높일 수 있다.
* Multi-cycle CPU는 하나의 instruction 안에서도 cycle에 따라 다른 행위를 해야 한다. 이 차이를 주기 위해서 control unit을 FSM을 이용하여 구현한다. 이는 control unit이 combinational logic으로만 구현되는 single-cycle CPU와 대조된다.

1번과 2번에서 비교한 장점들에 의해, multi-cycle CPU가 single-cycle CPU보다 더 좋다고 볼 수 있다. 이번 lab에서는 multi-cycle CPU를 구현하였다.

1. **디자인**

**A. CPU의 디자인**

CPU 디자인은 Ch6, 14p에 나와있는 수업자료를 참고했다. 아래 디자인에서 언급할 만한 부분을 설명하겠다.



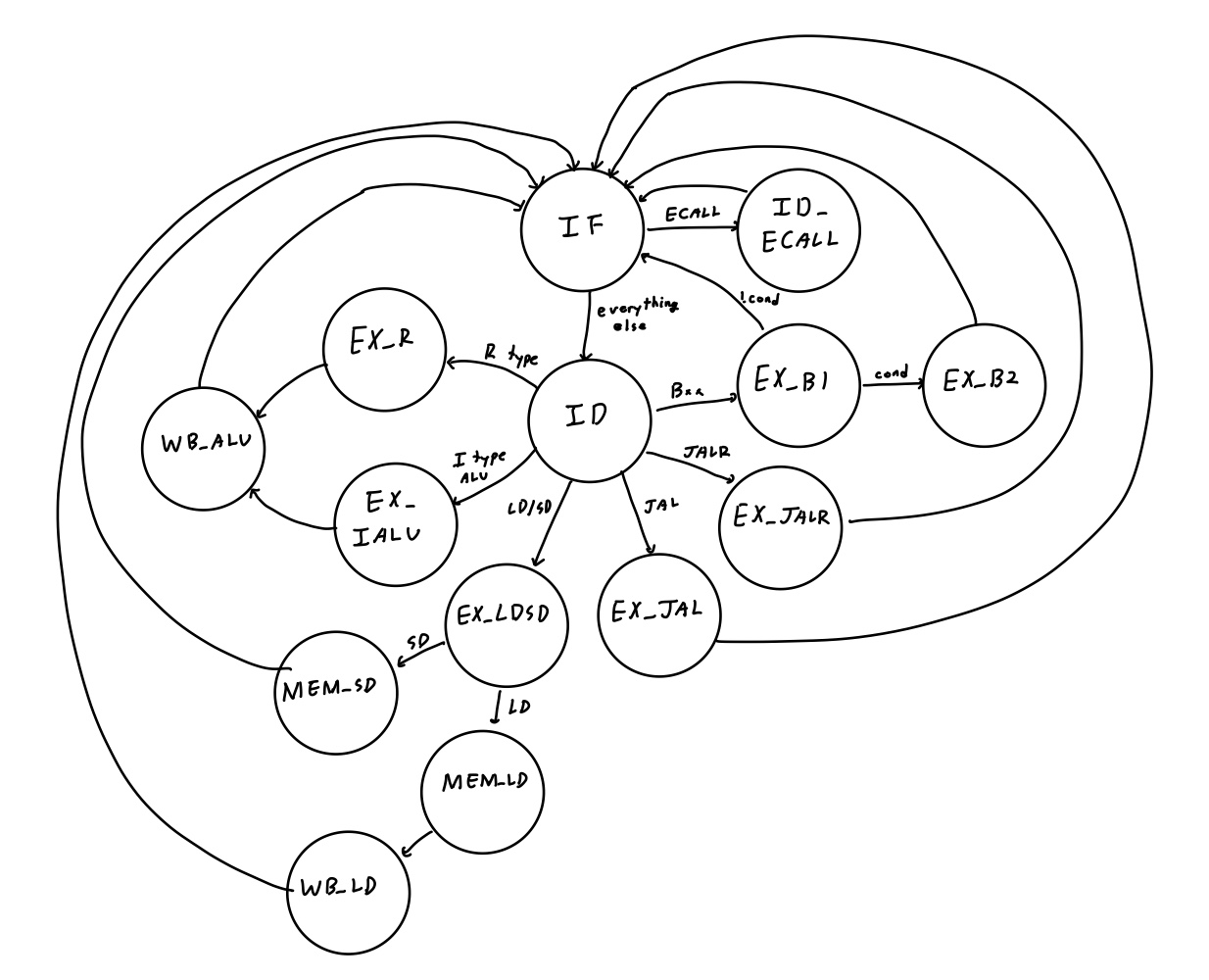
* 위 그림에서 clock synchronous한 module은 빨간색, asynchronous한 module은 파란색으로 표시하였다.
  + Memory와 register file은 데이터 저장 기능이 있으므로 synchronous해야 한다.
  + IR, MDR, A, B, ALUOut은 multi-cycle 구현을 위해 새로 도입한 latch이다.
  + IR레지스터는 control에 의해 enable이 조절되기 때문에, memory에서 출력되는 memdata값이 달라져도 IR값이 유지될 수 있다. 구현해야 하는 instruction들이 더 복잡해지면 MDR, A, B, ALUOut 등에도 control에 의한 enable을 주어야겠으나, 구현하는 instruction들이 그렇게 복잡하지 않아서 나머지 latch들은 “현재 사이클의 input 값을 다음 사이클로 넘겨주어 유지시켜주는” 형태 정도로 쓰이고 있다.
  + Control unit은 synchronous하도록 디자인되어 있다. 앞서 말했듯 multi-cycle CPU는 한 instruction이더라도 cycle에 따라 다른 행위를 해야 하므로 control unit은 clock synchronous해야 한다.
  + ALU control은 control unit이 주는 ALUOp 신호를 통해 cycle마다 다른 control을 출력하는 형태이며, ALU control 자체는 asynchronous하게 디자인하였다.
  + 나머지 모듈들은 그냥 주어진 input들을 combinational하게 처리하는 asynchronous 디자인으로 구현하였다.
* Resource reuse: Single-cycle CPU에 있던 세 개의 ALU를 하나로 합쳤으며, instruction memory와 data memory도 한 모듈로 합쳤다.
  + 이것이 가능한 이유는 한 instruction을 여러 cycle에 나누어 처리하기 때문이다.
  + 어느 cycle에 어떤 연산을 해야 하는지는 control unit이 결정해줄 것이다.
  + Single-cycle CPU 기준으로 어느 memory나 ALU를 ‘선택’하는지는 memory와 ALU 앞의 mux들을 통해 결정해 준다. (PC vs ALUOut), (PC vs A), (B vs 4 vs immediate)

**B. Microcode controller의 디자인**

Control unit은 자그마한 FSM으로 구현하였다. 구현의 편의를 위해 Moore machine으로 구현했다. Instruction들에 맞추어 state들을 잘 정의하고, 각 state마다 CPU가 해야 하는 계산이나 처리를 정의해 주면 된다. 여기서 주의해야 할 점은, 우리의 구현에서 resource reuse를 하고 있으므로 한 state에서 같은 resource를 두 번 이상 쓰지 않도록 주의해야 한다는 점이다.

모든 instruction들의 PVS(PC, memory, register) 업데이트는 instruction의 마지막 사이클에서 이루어질 수 있도록 구현했다. 첫 번째 그림은 control unit의 FSM state transition diagram, 두 번째 사진은 각 state별로 CPU에서 해야 하는 계산/처리들을 정리했다.

(is\_halted의 구현은 control unit의 밖에서 이루어져, ECALL의 control unit 내부 구현은 단순하다.)



텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

1. **구현**

**A. 전반적 CPU의 구현**

* cpu.v의 코드는 위의 “II-A. CPU의 디자인”을 그대로 코드로 옮긴 것이다. 추가적인 로직엔 다음과 같은 것들이 있다.
  + **is\_halted** 판별 로직: memory에서 꺼내온 **ecall\_reg\_cond** (x17 == 10?)와 control unit에서 꺼내온 **is\_ecall**을 and로 결합하여 판별한다.
  + PCWriteNotCond와 PCWrite control signal을 pc의 enable에 연결하는 간단한 combinational logic은 pc.v에서 구현했다.
  + Control unit의 input으로 bcond를 추가했다.
* Immediate generator, ALU, memory, register은 single-cycle의 것과 거의 동일하게 구현했다.
* mux들은 ternary operator을 이용해 구현했다.
* Latch들(A, B, ALUOut, IR, MDR)은 추가적 모듈 없이 cpu.v에서 reg로 정의하고 always @(posedge clk)으로 업데이트했다. 이 중 IR은 IRWrite == 1인 경우에만 업데이트했다.

**B. Control Unit과 Microcode Controller**

우선 control unit의 코드를 차례대로 살펴보겠다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 모듈의 Input과 output이다. 교안의 것에서 **is\_ecall**만 하나 추가해 주었다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 우선 state에 필요한 wire들을 정의하고, **is\_ecall**을 빠르게 처리해 주었다. **is\_halted**를 분별하는 로직은 메인 cpu 모듈에 구현되어 있어, control unit에서는 이 정도로만 해 주면 된다.

텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

Control unit의 메인 기능이 담긴 코드이다. 이 모듈의 기능을 크게 세 가지로 나누면 다음과 같다.

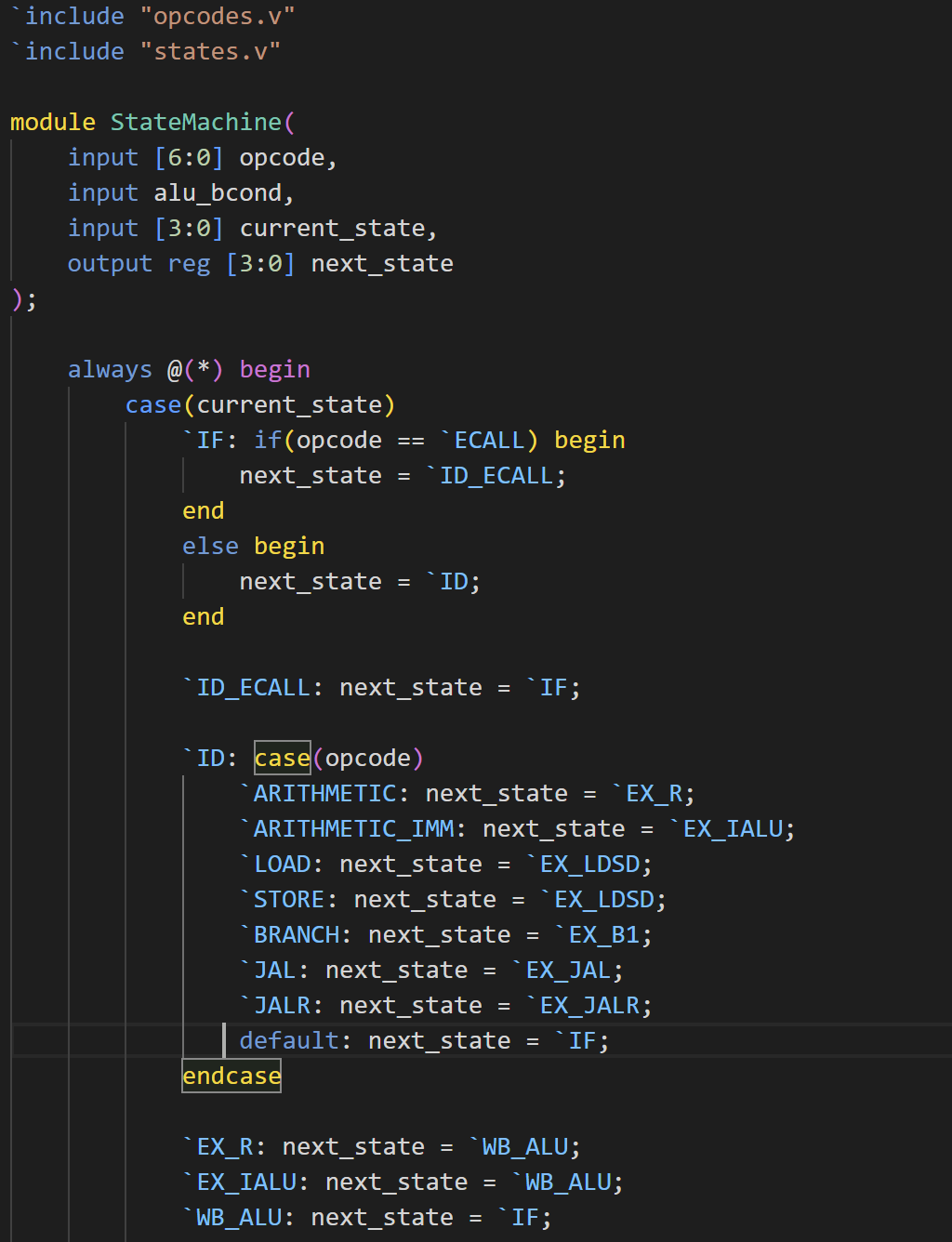
* 현재 state에 따라 control 시그널 결정하기
* 현재 state와 opcode에 따라 다음 state 결정하기
* Latch를 이용해 state를 저장하고 업데이트하기

Latch를 제외한 기능들은 각각 모듈화해서 구현하였다. 이제 각각의 모듈 코드도 하나씩 살펴보겠다.

**StateMachine** 모듈은 microcode controller에 해당하는 부분이라고 볼 수 있다. 사실은 엄밀히 말하면 microcode라기보다는 그냥 FSM state transition module이다. 이렇게 구현한 이유는 **IV. 논의사항** 에서 다루도록 하겠다.

**StateMachine** 모듈의 코드는 다음과 같다. 디자인 파트에서 첨부한 state transition diagram을 그대로 코드로 옮긴 것임을 확인할 수 있다.

추가로, branch에 따라서 state이 달라져야 하는 경우가 있으므로 input에 **alu\_bcond**이 포함된다.



텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

**StateToControl** 모듈은 microcode storage에 해당하는 부분이라고 볼 수 있다. (여기서도 엄밀하게 말하면 microcode storage는 아니지만 그와 동일한 기능을 하고 있다고 볼 수 있다.) 우리는 FSM으로 구현했으므로 각 state에 맞는 control output을 세팅해주면 된다. 이것에 해당하는 코드를 모두 첨부하겠다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 메뉴, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

**C. ALUControl**

ALUControl은 자체적으로 synchronous한 모듈로 구현하지 않으며, control unit을 통해 받은 정보를 이용해서 ALU에게 적절한 신호를 준다. Single-cycle cpu에서 구현한 ALU에서, control unit으로 ALUOp만 추가적으로 받아주면 된다. ALUOp에 따라 ALU가 해야 하는 행위는 다음과 같다.

* ALUOp == 2’b00: 덧셈
* ALUOp == 2’b01: branch
* ALUOp == 2’b10: funct3에 써있는 ALU operation

이를 반영한 코드를 아래에 첨부한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

ALU 모듈은 single-cycle CPU의 구현과 동일하기 때문에 다루지 않겠다.

1. **논의 사항**

앞서 말했듯 우리의 구현은 엄밀한 의미의 microcode 대신 그냥 FSM transition으로 control unit을 구현하였다. 이 부분에서는 1) FSM적인 구현과 microcode적인 구현의 차이 및 장단점, 2) FSM으로 구현한 이유에 대해서 서술하고자 한다.

**텍스트, 도표, 평면도, 기술 도면이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

1. FSM vs microcode

Microcode는 CPU 내부에서 명령어 해석 및 제어 신호 생성을 담당하는 하위 레벨 코드이다. 위의 그림처럼, opcode를 받으면 내부적 로직을 이용해 microprogram counter을 조종하고, 이 microprogram counter이 가리키는 microcode storage에서 control output을 결정시킨다. 이렇게 하면 일단적으로 FSM으로 구현했을 때보다 공간을 더 아낄 수 있고, 새로운 instruction을 implement하고 싶을 때의 확장성이 더 뛰어나다.

1. FSM으로 구현한 이유

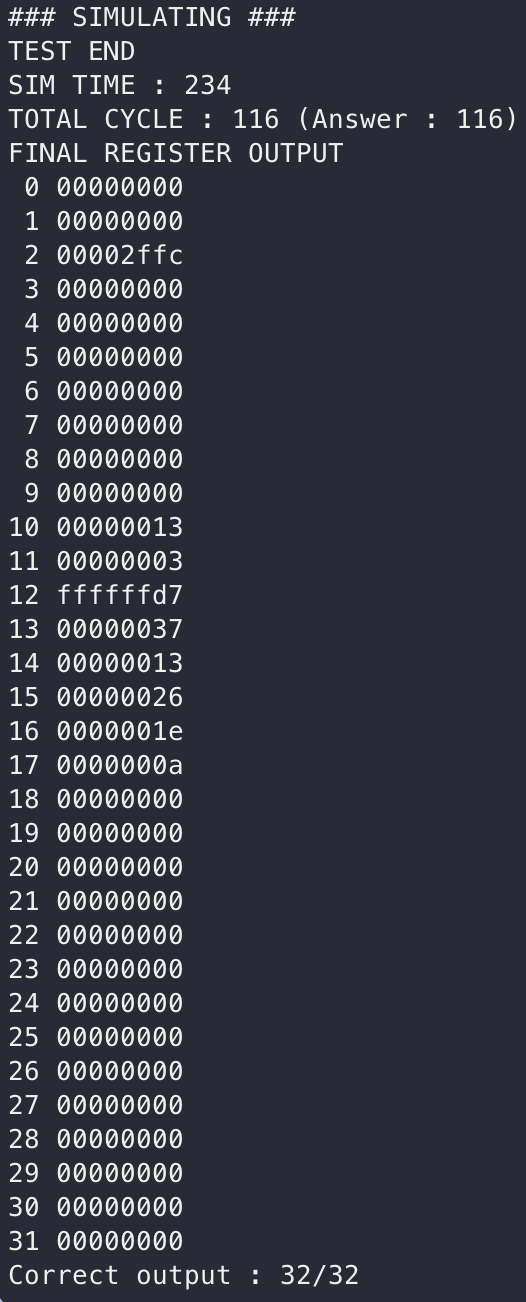
텍스트, 스크린샷, 번호, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

우리의 구현에서 state가 그렇게 많지 않고, state transition이 opcode에 dependent한 경우가 그렇지 않은 경우보다 많다. 따라서 대부분의 state transition이 microcode에서는 branch 형태일 것이다. 구체적으로, 위의 그림에서 “ - ”에 해당하는 cell이 우리 implementation에서는 거의 없다. 따라서 그냥 FSM으로 구현하는 것이 더 자연스럽다고 생각했다.

1. **결론**

5개의 example에 대한 testbench를 실행해본 결과는 아래와 같다. 순서대로 basic, non-controlflow, loop, ifelse, recursive example에 대해 testing을 해 본 결과이다.

 텍스트, 스크린샷, 디자인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다. 텍스트, 스크린샷, 디자인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 디자인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다. 텍스트, 스크린샷, 책, 디자인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

구체적으로, basic example에서 116 cycles, loop example에서 977 cycles을 사용하였다. Final register outputs도 answer의 것과 같은 것으로 보아 우리의 multi-cycle CPU 구현은 정확하다는 결론을 내릴 수 있다.

처음에는 multi-cycle CPU가 조금 낮설고 어렵게 느껴졌었다. 특히 control flow를 state들을 이용해서 구현하는 부분이 헷갈렸는데, 직접 구현해 보면서 익숙해지게 되어 좋았다. 이해가 더 잘 된 느낌이다.