**Computer Architecture and Mobile Process**

**Single Cycle**

**Project2**

**April 9, 2022**

**Left day 5**

**Mobile System Engineering**

**32184939**

**Heo ChanYong**

싱글사이클이란?

싱글 사이클은 instruction 하나를 하나의 사이클에서 수행하는 것을 의미한다. 한 사이클이 끝나면 다음 instruction을 수행을 한다. Instruction의 format에 따라 작동해야 하는 하드웨어가 달라진다. 그래서 여러가지 instruction을 수행하려면 controller로 하드웨어를 조절해줘야 한다. 물론 단일 type의 instruction을 다룬다고 가정을 하면 따로 controller를 조절할 필요가 없는 경우가 생길 것이다. 물론 앞서 말한 가정은 instruction 이 복잡하지 않다는 가정이다. 하지만 type에 따라 하드웨어가 필요하면 비용이 많이 들것이다. 이와 같은 이유로 하나의 하드웨어에 controller를 두어 instruction의 opcode에 따라 controller로 하드웨어의 작동을 조절해줄 수 있다.

싱글 사이클을 만들기 위해 필요한 생각들

싱글 사이클을 구성할 때 가장 먼저 생각했던 것들은 어떻게 instruction을 읽어와 bit값들을 하드웨어 어떻게 전달을 하는 것 하나와 control을 어떻게 조절을 해서 instruction의 opcode나 func에 맞게 연산을 할 수 있을까? 였습니다. 그래서 저는 instruction을 메모리에서 가져와 어떤 instruction인지와는 상관없이 opcode, rt, rs, rd, shamt, funct, address를 나누어 주었습니다. 원래는 opcode에 따라 변수들이 정해지지만 Datapath를 보면 먼저 instruction을 사용하여 모든 변수들을 저장하는 것을 보고 아이디어를 얻었다. 위와 같이 decode를 하고 decode에서 얻은 opcode를 통해 control을 조절할 수 있다. Opcode를 통해 나온 control로 각각의 하드웨어들을 선택적으로 동작 시킨다. 예를 들어 add라고 가정을 하면 opcode는 0 이 되어 controller에서 RegWrite =1 RegDst=1 이 되고 나머지 controller는 0이 된다. 그래서 rd 레지스터에 reg[rt]+ reg[rs] 값을 저장을 한다. 물론 instruction의 format에 따라 크게 분류할 수 있겠지만 항상 예외들은 존재하기 마련이다. 그래서 DataPath를 그려보고 어떻게 controller를 조절하면 좋을까 생각을 많이 했다.

싱글 사이클 구현

싱글 사이클을 구현할 때 가장 먼저 하는 것은 bin파일에서 binary 형식으로 instruction을 읽어와 memory에 저장을 하는 것이다. 여기서 조심해야하는 점은 mips-binary 파일은 거꾸로 저장이 되기 때문에 이것을 우리가 조절을 해야 한다. 예를 들어 instruction이 원래는 0x27bd7ff0인 경우 27 bd 7f f0 순서대로 받아야 하지만 mips-binary의 특성상 f0 7f bd 27로 저장이 되어있다는 것이다. 그래서 bin파일을 받을 때 순서에 맞게 swap을 해줘야 한다. 본인 같은 경우는 아래 사진과 같이 swap을 해주었다.

텍스트, 실내이(가) 표시된 사진

자동 생성된 설명 figure 1

위 사진과 같이 instruction을 받으면 순서에 맡게 Memory에 저장을 해준다. bin파일을 다 읽고 우리가 만든 메모리 배열에 저장이 된다.

위 과정이 끝나면 진정한 single cycle의 구현이 시작된다. Single cycle의 구현을 위해서 필요한 모든 변수를 전역변수로 만들면 좋겠지만 예상치 못한 에러가 발생할 수도 있고 가시성에서 떨어질 수도 있다고 생각을 한다. 그래서 본인은 struct형태로 필요한 변수들을 묶었다. 물론 모든 변수들을 struct형식으로 만든 것은 아니다. 아래 사진과 같이 inst\_, Control\_, stat\_ 3개만 struct로 만들어 주었다.

텍스트이(가) 표시된 사진

자동 생성된 설명 텍스트이(가) 표시된 사진

자동 생성된 설명 텍스트이(가) 표시된 사진

자동 생성된 설명 figure 2,3,4

본인이 생각하기에는 해당 변수들을 struct로 나누어 준 이유는 관리가 편리하다고 생각을 하였기 때문이다. 해당 변수들을 초기화를 하고 싶으면 memset() 을 사용하여 원하는 값들로 코드 한 줄로 바꿀 수 있기 때문이다. 위 과정이 끝나면 instruction을 저장한 메모리를 읽기 시작한다. 초기 pc는 0이고 fetch를 통해Memory에서 instruction을 읽어온다. 읽어온 instruction을 decode과정을 해준다. Decode는 instruction을 받아 inst\_ 구조체의 변수들의 활용에 맞게 다 업데이트 해준다. 그렇게 decoding을 하고 나서 업데이트 된 inst\_ 구조체를 통해 control을 조절을 해준다. 여기서 Control은 매우 중요하다. Control은 figure 5와 같이 수행된다. 해당 Controle 함수는 opcode를 통해 control을 조절하는데 Control 구조체에 들어가 있는 controller들을 조건에 맞게 bool값으로 업데이트 해주었다. 교수님께서 업로드 하신 자료에는 jr\_chk나 Jal\_chk 와 같은 controller 가 없었으나 해당 single cycle을 구현할 때 필요하다고 생각이 되어 따로 controller를 control 구조체에 추가를 하였다. 물론 jump 한다는 점은 같을 수 있으나 사용해야하는 기능들이 조금씩 다르기 때문에 이와 같은 결정을 하였다. 해당 로직에서 조심해야할 점들은 교수님의 자료에 있는 DataPath나 Control의 조건에는 구현이 안된 instruction이 있기 때문에 구현하는 본인이 수정을 해야 하는 부분이 있다는 것을 인지를 해야 한다. 본인도 해당 부분을 인지를 하지 못하고 구현을 하다 무한루프에 빠져 하나씩 decoding을 통해 문제점을 확인할 수 있었다.

텍스트이(가) 표시된 사진

자동 생성된 설명figure 5

controller들을 업데이트 하고 나서 excute 함수로 이동을 한다. excute같은 경우에는 control 구조체를 통해 연산이 적용이 된다. 가장 먼저 ALUSrc가 false이며 opcode가 0인 경우 funct를 통해 연산이 이루어진다. 위와 같은 조건으로 나눈 이유는 beq나 bne ALUSrc가 False인데 I 타입이므로 연산과정이 달라지기 때문이다. 특히나 beq나 bne 같은 경우는 조건이 맞냐 틀리냐 에 따라 Br\_taken이 업데이트 되어야하기 때문에 조건을 나누어 주었다. 이제 ALUSrc가 true인 경우는 I type이기 때문에 opcode에 맞는 연산을 실행해준다. J 같은 경우 excute 가 필요 없기 때문에 해당 excute함수에는 포함되지 않는다. 실행이 되고 나서 memory에 저장하거나 레지스터에 저장을 해야 한다. 이를 수행하기 위해서 본인은 Memory\_and\_store라는 함수를 만들었다. Memory\_and\_store 함수에서 확인하는 controller는 Memwrite, MemRead, RegWrite, MemtoReg, Jump\_chk이다. 여기서 jump\_chk를 왜 확인을 하지라고 생각을 할 수 있다. 해당 부분은 본인의 실력이 부족하여 그럴 수도 있고 아니면 프로그래밍이다 보니 생기는 문제일 수 있다. 본인이 생각한 jump\_chk의 활용은 I type의 instruction을 다룰 때 문제가 생긴다. I type은 jump와 controller가 비슷하다는 문제점이 있다. RegWrite=1, ALUSrc=1, RegDest=0, MemtoReg=0, MemRead=0 이기 때문에 jump\_chk를 통해서 구분을 해줘야 한다. 만약 위와 같이 jump\_chk를 넣어주지 않으면 엉뚱한 register에 엉뚱한 값이 저장이 될 수 있기 때문이다. 위의 과정이 끝나면 pc를 업데이트 해주어야한다. Pc와 register는 opcode가 j, jal, jr에 따라서 업데이트가 다르게 작동이 된다. j같은 경우 jump\_chk라는 controller를 통해서 확인을 하고 jump address로 pc를 업데이트 시킨다. jal같은 경우는 jal\_chk라는 controller로 reg[31]를 pc값으로 업데이트 시키고 pc를 jump address값으로 업데이트 시킨다. jr같은 경우는 jr\_chk라는 controller로 확인을 하고 excute함수를 통해 얻은 ALU\_Result 값으로 pc를 업데이트 시킨다. 마지막으로 BNE나 BEQ 같은 경우 Br\_taken controller와 excute에서 얻은 ALU\_Result값이 동일한지를 통해 pc를 branch\_address로 업데이트 시켜준다. 위의 과정이 끝나면 한 사이클을 돌았다고 한다.

싱글 사이클 코드 실행 결과

Simple.bin 실행결과

텍스트이(가) 표시된 사진

자동 생성된 설명

figure 6

Simple2.bin 실행결과

텍스트이(가) 표시된 사진

자동 생성된 설명 figure 7

Simple3.bin 실행결과

텍스트이(가) 표시된 사진

자동 생성된 설명 figure 8

Simple4 실행결과

텍스트이(가) 표시된 사진

자동 생성된 설명 figure 9

gcd.bin 실행결과

텍스트이(가) 표시된 사진

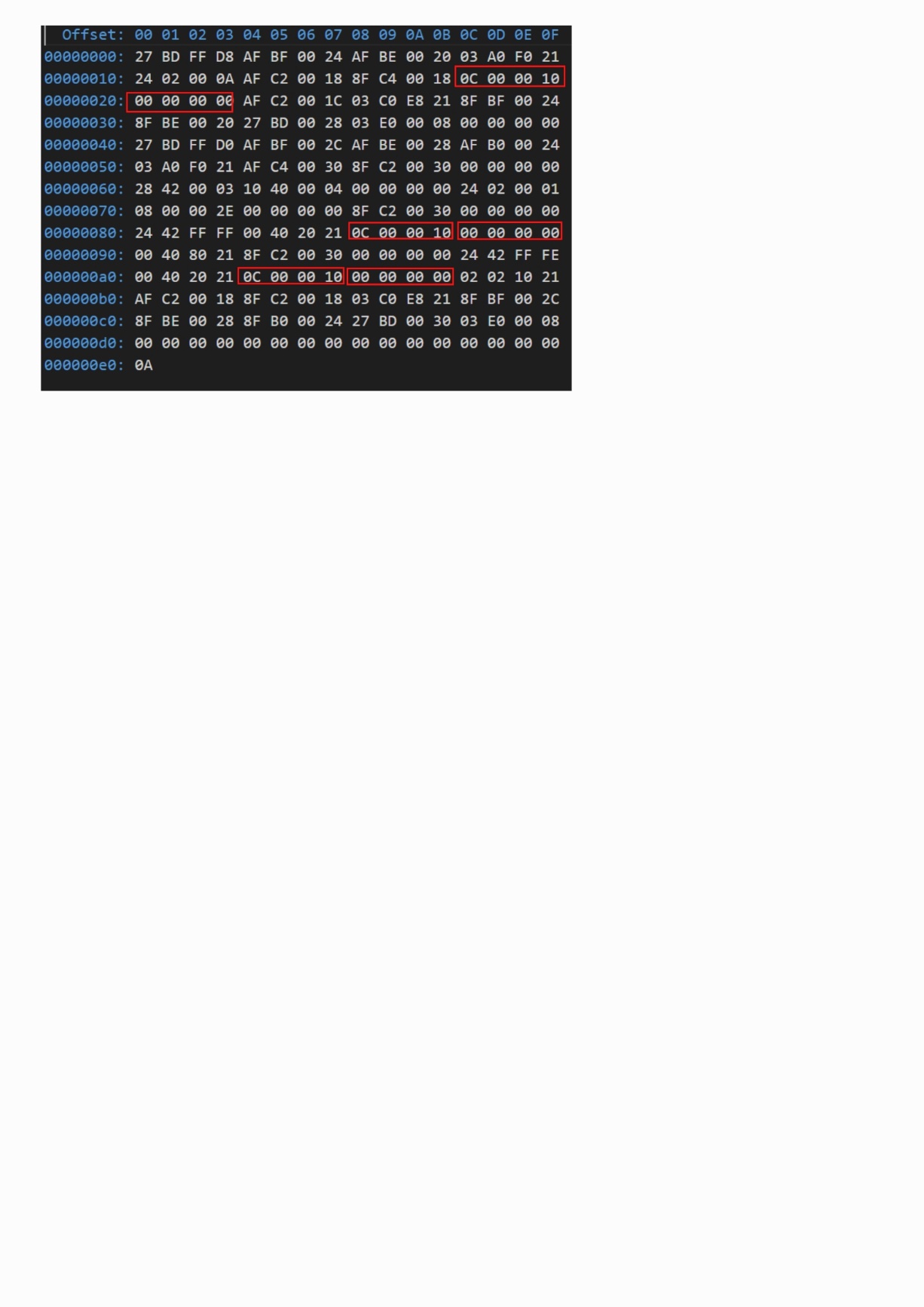
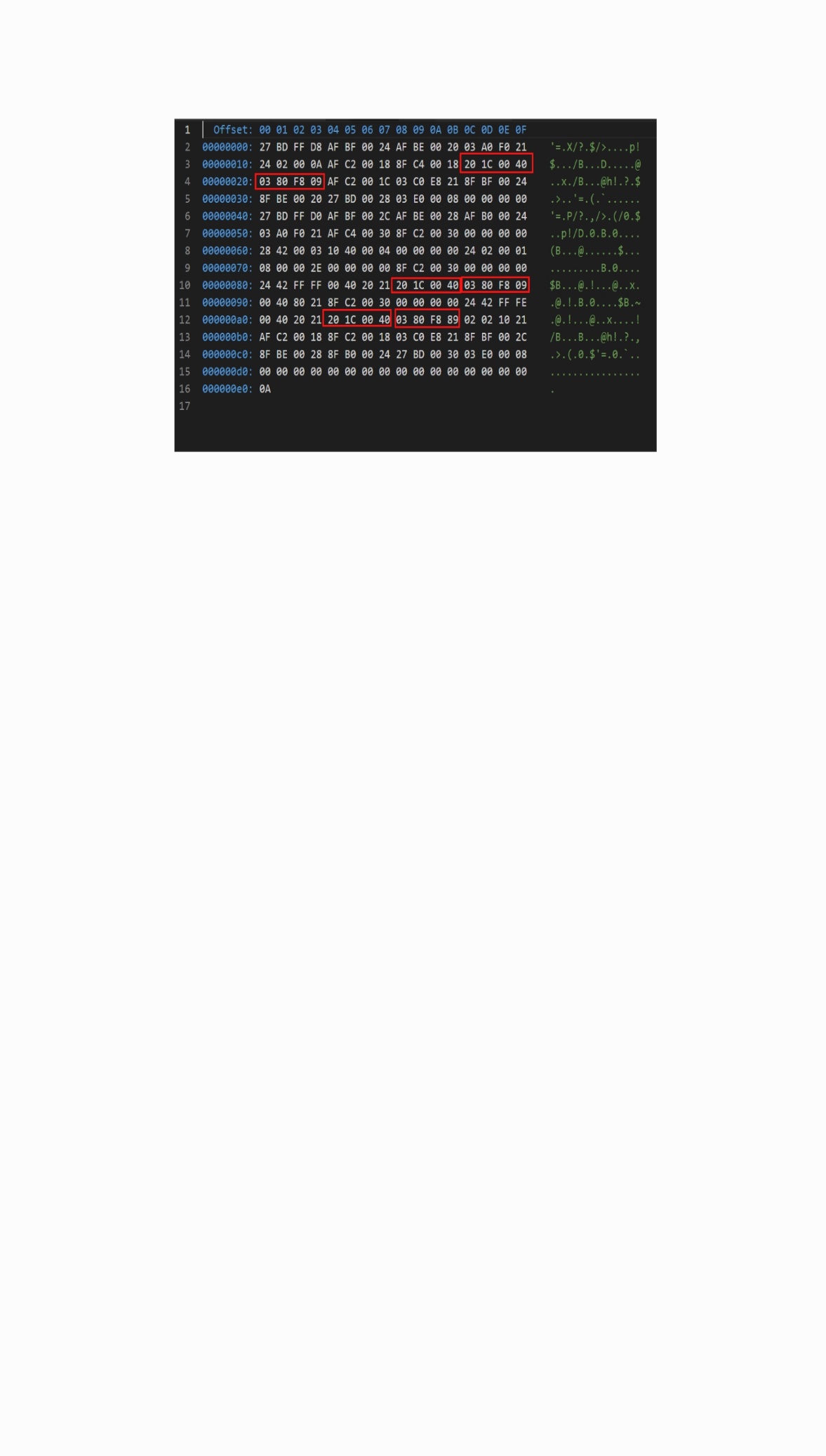
자동 생성된 설명 figure 10

fib.bin 실행결과

텍스트이(가) 표시된 사진

자동 생성된 설명 figure 11

Fib2.bin



텍스트이(가) 표시된 사진

자동 생성된 설명

figure 12 13 14

Optional JAL을 JALR로 고쳐서 하는 추가 문제가 있었다. JAL을 어떻게 JALR로 고칠 수 있을까 생각을 하다가 가장 먼저 생각했던 것은 어떤 레지스터에 다음 주소값을 집어넣어야 할까였다. 만약에 사용하는 레지스터에 값을 집어넣게 되면 연산이 문제가 생길 수 있다고 판단을 하였기 때문이다. 그래서 mips 코드를 보았고 28번 레지스터는 사용을 하지 않는다고 판단을 하였다. 그러면 레지스터에 주소 값을 어떻게 집어넣어야 할까? 를 다음으로 생각을 하였다. Addiu를 사용해서 0번 zero 레지스터를 이용해서 주소 값을 넣어야 겠다고 판단을 하였다. 그렇게 28번 레지스터에 0x40이라는 값을 집어넣었다. 0x40이라는 값은 괜히 나온 값이 아닌 fib 함수의 첫 주소를 의미하였고 다음 jalr 이 가야하는 target address를 의미하기 때문이다. Addiu를 통해 r28 에 주소값을 삽입을 하고 다음 0x0380f889를 통해서 다음 값으로 넘겨주었다. Opcode는 0이고 rs=28, rt= 0, rd =31 func는 9이다. 여기서 jalr은 28번 레지스터 값을 31번 레지스터에 저장을 해준다. 그렇게 JAL 역할을 2가지 instruction으로 나누어 수행을 하도록 하였다. 실제 JALR은 점프이기 때문에 pc를 업데이트 하는 부분에서 조절을 해주었다. 해당 코드에서 조심해야하는 부분은 reg[ins->rd]=pc라고 하였는데 해당 pc는 현재 pc에 4를 더한 값이다. 원래라면 jump를 하거나 JAL을 할 때는 원래 pc에 +8을 하는 것이 정상적이나 다음 수행해야하는 pc를 오직 +4만 한 것은 원래 jal뒤에는 nop이라는 0x0000 0000 이라는 명령어가 있어 뛰어넘어야 했지만 본인은 nop자리에 jalr을 넣었기 때문에 현재 pc에 +4만 하였다. Jalr은 jal 명령어를 두개로 나누다 보니 cycle을 더 많이 돈 것을 확인할 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명 figure 15

input4.bin 실행결과

텍스트이(가) 표시된 사진

자동 생성된 설명 figure 16

해당 파일들을 전부 실행을 하였을 때 R, I, J instruction을 수행했을 때 나오는 횟수의 합과 총 cycle의 횟수는 같아야한다. 현재 코드는 nop이라는 instruction을 R 타입 instruction이라고 가정을 하였다. 그래서 위와 같은 결과들이 나왔고 합도 cycle의 횟수와 같게 나오고 결과값도 예상값과 똑같게 나와서 single cycle을 알맞게 구현했다고 생각을 한다. 여기서 주목해야하는 점은 만약에 single cycle을 최적화 시킨다면 어떨까라는 생각을 하였다. 물론 본인이 잘못 생각을 하고 있을수도 있지만 한 사이클을 돌 때 강의 자료에서 처럼 600ps가 걸린다고 가정을 해보자. Gcd.bin을 실행 결과를 참고하여 GCD.bin 의 실행시간은 636600 ps가 걸리고 만약 아래 그림처럼 필요한 하드웨어만 들어갔다고 가정을 해보자.

테이블이(가) 표시된 사진

자동 생성된 설명

figure 17

R은 359\*400-> 143600, I는 400\*78-> 31200, Lw는 333\*600 -> 199800, Sw 550\*153 -> 84150, br 73\*350 ->25550 총 484300ps가 걸린다. 이렇게 최적화를 하면 약 30% 정도 cost를 덜 사용할 수 있지 않을까를 생각했다. 물론 아직 배움이 짧다보니 해결책은 더 생각을 해보아야겠다.

해당 프로젝트를 하고 느낀점

프로젝트를 진행하기 전에는 겁이 많이 났다. 내가 올바르게 single cycle이라는 개념을 이해하고 Data Path에 대하여 올바르게 이해를 하였는지 확신도 안 섰기 때문이다. 해당 프로젝트를 Data Path도 안 그려보고 시작을 하였으나 바로 장애물에 부딪히게 되었다. Data Path를 보면 우리가 어떻게 하드웨어를 조절할지 하드웨어가 어떤 기능을 하는지에 대해서 알 수가 있다. 이러한 부분들을 하나씩 이해하고 공부를 하니 어떤 방식으로 접근을 해야 하고 어떤 함수들을 만들어 싱글사이클을 구현해야 할지 이해하였다.

해당 프로젝트를 하면서 막혀서 힘들었던 부분들을 몇 가지 서술하겠다.

첫번째는 구조체에 대해서 잘못 이해하고 있었던 점이다. 구조체를 선언을 하면 포인터로 관리를 하겠지 했으나 값들이 제대로 업데이트가 되지 않았고 그에 따라 pc나 다른 구조체나 구조체의 변수들이 제대로 업데이트가 되지 않았다. 디버깅하면서 문제점들을 찾아 잘못 알고 있던 부분들을 수정하였다.

두번째는 Memory\_and\_Store() 함수의 조건이었다. Instruction에 맞게 Control이 업데이가 되고 그에 맞게 Memory\_and\_Store()이라는 함수가 작성되었어야 했다. 그래서 Memory\_and\_store 함수에 들어가는 조건들을 조절을 했어야 했다. 하지만 수업 자료에 있던 Data Path를 맹신하여 코드를 짜서 문제가 발생하였다. 분명 교수님이 수정하셔야 합니다. 라고 말씀을 하셨으나 해당 내용들을 잊고 코드를 짰었다. 문제가 생겼던 부분은 I type 부분이었다. Jump를 할 때도 해당 부분이 실행이 되어 건드릴 필요 없는 register에 이상한 값들이 저장이 되어 무한루프가 도는 경우가 생겼었다. 이를 찾기 위해서 디버깅을 끊임없이 했어야 했다. 하지만 시작 전에 교수님의 말씀을 기억했다고 해도 아마 디버깅을 하면서 열심히 찾지 않았을까 생각이 든다.

해당 프로젝트를 하면서 많은 것들을 느꼈다. 그래도 코드는 어느정도 짠다고 생각했던 내 자신이 정말 부끄러웠다. 물론 실력이 많이 부족하다고 생각은 하였으나 평균정도는 될 것이라고 생각을 하였으나 누가 보기에는 간단한 코드도 며칠을 투자하여 끝낼 정도로 실력이 형편없다고 느꼈다. 뿐만 아니라 코드를 짜기 전에 어떻게 짤 것인지 구상을 하는 것도 무척 중요하다고 느꼈다. 무턱대고 생각나는 대로 작성하다 보면 많은 예상치도 못한 문제들이 발생하여 디버깅을 하더라도 찾지 못하는 경우가 생길 수 있다는 것을 느꼈다.