Project 2: MIPS emulator

201811061 박동진

1. 소스 코드에 대한 간략한 설명

이 소스 코드는 C++로 작성되었습니다.
$./runfile [-m addr1:addr2] [-d] [-n num_instruction] input file 를 터미널에서 입력받은 후, addr1:addr2를 searchAddr에 저장하고, -d가 있는지 여부를 afterProcess로 있는 경우 true, 없는 경우 false로 설정하였습니다. 그리고 num_instruction을 performNum에 저장하였습니다.

searchAddr을 splitAddr함수로 ":"를 기준으로 나누어 mAddrs에 저장하였습니다.

ifstream으로 첫번째 줄과 두번째 줄을 읽고 그를 4로 나누어 인스트럭션 개수와 워드의 개수를 각각 textCount와 dataCount에 저장하였습니다.

ifstream으로 vector<unsigned int> lines에 인스트럭션 값과 워드 값을 unsigned int 형식으로 저장하였습니다.

map<unsigned int, unsigned int> address를 만들어 0x400000 부터 인스트럭션 주소와 값을 저장하고, 0x10000000 부터 워드의 주소와 값을 저장하였습니다.

map<unsigned int, unsigned int> regs를 만들어 R0 ~ R31의 value를 모두 0으로 초기화 해주었습니다.

endAddr = 4194304 + (textCount*4)로 while이 끝날 조건을 설정하였습니다.
이를 통해, -n 옵션이 없을 때, 모든 인스트럭션이 작동할 수 있도록 설정하였습니다.

bool isLimit과 unsigned int count를 이용해 while문이 끝날때마다 count를 1씩 증가시켜 count가 performNum과 같을 때 break하도록 하였습니다.
이를 통해, -n 옵션이 있을 때, num_instruction만큼 실행할 수 있도록 설정하였습니다.

while문 안에 isChange와 moveAddr을 이용해 branch나 jump 같은 주소를 이동하는 인스트럭션을 따로 관리 하였습니다.

branck나 jump가 아닌 보통 인스트럭션들은 currPc(=4194304)를 while문이 끝날 때마다 4씩 더해주어 다시 while문을 실행할 수 있도록 하였습니다.

unsigned int fullInst로 address에 있는 value를 가져오고 bit operation을 이용해 상위 6비트를 checkOp에 저장하였습니다. checkOp = 0일 때는 R format이므로 하위 6비트를 추가로 확인해 인스트럭션을 구분하였습니다. checkOp가 0이 아닐 때는 checkOp로 인스트럭션을 구분하였습니다. if문 옆에 주석으로 인스트럭션을 적어놓았습니다.

R format은 ctrRformat 함수를 이용해 fullInst를 rs, rt, rd 순서로 <unsigned int>vector에 저장하였습니다.

I format은 ctrIformat 함수를 이용해 fullInst를 rs, rt, offset(imm) 순서로 <unsigned int> vector에 저장하

였습니다.

sll, srl은 ctrShift 함수를 이용해 fullInst를 rt, rd, shamt 순서로 <unsigned int> vector에 저장하였습니다.

j, jal, jr은 따로 함수를 사용하지 않고 if 안에서 bit operation으로 실행하였습니다.

addu와 addiu에는 더한 값이 0xffffffff을 넘어갈 경우 0x100000000을 뺄셈을 해서 without overflow처리를 해주었습니다.

addiu와 sltiu는 imm을 sign-extend 해야 하는데 imm이 16비트인 것을 이용해 signed short 타입을 활용해 계산을 진행하였습니다.

리눅스는 little Endian 형식으로 데이터를 저장하기 때문에 lw, lb, sw, sb를 구현하기 위해 word 데이터를 big Endian으로 변환할 필요성을 느꼈습니다. 그래서 vector<unsigned int> originalData에 little Endian 형식으로 데이터를 저장하고 이를 포인터를 이용해 bigEndianData에 big Endian 형식으로 데이트를 옮겼습니다.

lw 설명:
originalData에서 rs에 해당하는 워드를 찾아 그 인덱스를 findIndex에 저장합니다. vector<unsigned char> lwChar에 findIndex*4 + offset에 해당하는 4바이트를 저장합니다. lwChar를 리눅스가 읽을 수 있도록 bit operation과 형변환을 통해 little Endian 형식으로 바꾸고 rt에 저장합니다.

lb 설명:
originalData에서 rs에 해당하는 워드를 찾아 그 인덱스를 findIndex에 저장합니다. char lbChar에 findIndex*4 + offset에 해당하는 1바이트를 저장합니다. char를 int로 변환하며 sign extension이 진행됩니다. 이를 rt에 저장합니다.

sw 설명:
originalData에서 rs에 해당하는 워드를 찾아 그 인덱스를 findIndex에 저장합니다. originalData를 bigEndianData로 변환하는 과정과 동일하게 rt에 저장된 4바이트 데이터를 bigEndian 형식으로 바꾸어 bigEndianData를 변경합니다. bigEndianData를 리눅스가 읽을 수 있도록 littleEndian으로 바꾸어주는 bigToLittle함수를 이용합니다. 그리고 이렇게 만든 littleEndian 데이터를 address에 업데이트하는 updateDataAddr함수를 이용해서 수정된 word data를 업데이합니다.

sb 설명:
originalData에서 rs에 해당하는 워드를 찾아 그 인덱스를 findIndex에 저장합니다. rt를 bigEndianData의 findIndex*4 + offset의 주소에 먼저 수정을 한뒤, bigToLittle 함수를 이용해 littleEndian Data로 바꿔줍니다. 그리고 updateDataAddr 함수로 address를 업데이트합니다.

다른 인스트럭션들은 regs[Arr[]] 형식을 활용해 계산을 진행하였습니다.

printOutput 함수를 이용해 PC, 레지스터, -m 옵션 주소를 출력합니다. -d 옵션이 없을 경우, 마지막 상태를 출력하도록 하였습니다.

performNum == 0 일 경우 printOutput을 실행시키고 return 0로 프로그램을 종료하여 인스트럭션이 하나도 실행되지 않은 상태를 출력하였습니다.

2. 컴파일/실행 환경, 방법

-컴파일/실행 환경

이 소스코드(main.cpp)는 Ubuntu 20.04.4 64bit 환경(VirtualBox)에서 작성되었습니다. 컴파일러는 g++ 9.4.0을 사용하였습니다.

-컴파일/실행

터미널에서 main.cpp와 input파일이 있는 디렉토리로 이동합니다. 터미널에 g++ .o runfile main.cpp를 입력합니다. 터미널에 $./runfile [-m addr1:addr2] [-d] [-n num_instruction] <input file> 을 입력합니다. 예를 들어 input파일이 sample.o 일경우, 터미널에 ./runfile -m 0x10000000:0x10000010 -d sample.o 를 입력합니다. 예를 들어, 인스트럭션을 5개 실행하고 싶을 경우 ./runfile -m 0x10000000:0x10000010 -d -n 5 sample.o를 입력하면 됩니다.