<pa2 - 보고서>

1. How you process the MIPS instructions

MIPS instruction이 hex형태로 command창에 입력되면, 이는 process_instruction을 call하여 unsigned int 형태로 들어오게 된다. 이때 각 instruction들은 opcode를 기준으로 어떤 instruction인지 구분할 수 있으므로 opcode가 0인 r type, 2, 3인 j type, 나머지는 I type으로 나눠줄 수 있다.

이때 각 instruction의 struct type을 선언하고, instruction의 opcode, rs, rt, rd, shamt, funct, address, immediate value는 instruction의 shift와 masking을 이용해서 각각 구현해준다.

이후 각 type에 따라서 process_r, process_i, process_j함수 호출을 통하면 registers배열에 우리가 원하는 각 instruction에 맞는 기능을 구현할 수 있다.

이때 각 타입에 맞게 add, sub, and, or, andi, ori, jal,j 등등을 구현해 주었는데 각 instruction의 정의에 맞게 구현해주었다. 이때 헷갈릴 만한 기능의 구현을 살펴보면, slt instruction은 rs와 rt를 교해야 하는데 이때 단순 비교를 할 시에 registers가 unsigned int형이므로 값이 알맞게 비교가 되지 않을 수 있다. 따라서 signed bit가 1인 음수인 경우의 비교를 위해서 signed int로 register을 casting해 주었다. sra역시 casting해주었다. 또한 process_j에서 j instruction의 경우에 Pseudo-direct jump addressing을 적용하여 pc의 4bit만 가져온 후 이를 address를 2만큼 shift한것과 bit or 해 주었다. 또한 jal의 경우도 마찬가지 이지만, pc값을 ra에 넣어줘야 한다.

따라서 이 경우 run_program을 한다면 run_program에서 pc를 4로 증가시키고, process_instruction을 실행하기 때문에 바뀌진 pc를 ra에 넣는 것이다. (따라서 jal에서 pc+4를 하면 안된다.)

또한 immediate value를 사용하는 I format의 경우 andi, ori는 ZeroExtImm을 사용하고, 나머지는 SignExtImm을 쓴다. 따라서 16bit immediate value를 위해 short로 casting하고 각각 ZeroExtImm인지, SignExtImm인지에 따라 unsigned short와 short로 casting 해준다. 또한 beq, bne는 Pc-relative addressing에 따라서 $pc + imm*4를 적용해준다.

마지막으로 lw, sw시에 메모리의 값을 레지스터에 담아야 하는데 이때 Big-endian을 사용하기 위해서(mips는 big endian을 사용하므로) Little-endian 메모리에 반대로 써져 있는 값의 가장 작은 메모리 주소 값을 추출하기 위해서 왼쪽으로 shift하는 연산을 해야 한다. 이렇게 되면 가장 작은 주소가 가장 큰 주소로 바뀌게 된다. 따라서 shift를 하고 그 값과 or하는 연산을 반복함으로써 Little/Big-endian 문제를 해결하였다. (sw는 반대의 과정이기 때문에 오른쪽으로 shift한다.)

2. How you load the program into the memory

load_program는 Initial_PC가 가리키는 메모리에 파일에 있는 instruction들을 하나하나 로드 하는 것이다.

따라서 이때는 파일을 열고, 파일에 있는 instruction 명령들을 하나하나 수행하는데 이때 strtoimax함수를 이용하여 instructon을 unsigned int로 바꿔주고 이때 메모리에 로드하는 과정에서 instruction을 메모리에 그대로 로드 할 경우 반대로 로드 되기 때문에 Big-endian을 고려하여 instruction의 제일 왼쪽 부분을 메모리의 제일 작은곳에 쓰는 방식의 Big-endian의 shift연산을 수행하였다. 이후 메모리의 제일 끝에 halt명령인

0xffffffff을 추가해주었다.

3. How you run the program

run_program은 메모리에 로드 된 program을 running하는 것이다. 이때에 메모리에는 Big-endian형태로 값이 들어가 있지만, unsigned int parameter을 갖는 process_instruction을 수행하기 위해서는 메모리에 들어간 instruction 명령의 값을 다시 unsigned int 형식의 반대로 뒤집어진 값으로 만들어야 한다.
따라서 memory에서 pc가 가리키는 것의 값을 8bit씩 &연산을 통해서 뽑아내고 이를 shift하는데 이때 가장 왼쪽은 가장 오른쪽으로, 두번째 왼쪽은 하나만 오른쪽으로, 제일 오른쪽에 있는 것은 가장 왼쪽으로, 오른쪽에서 두번째는 하나만 왼쪽으로 이동하면 된다. 따라서 <<와 >>를 각각 두 개씩 섞어서 구현해주었다.
이렇게 되면 process_instruction에 들어갈 값을 만들어 준 것이 되고, 이는 “fetch an instruction at pc”와 같으므로 이후에 pc를 4증가시킨 후 “execute the instruction” 즉 process_instruction을 하면 된다. 이때 process_instruction의 리턴값이 성공적이면 1, 'halt' 또는 모르는 instructions가 오면 0이기 때문에 이를 이용하여 load된 program을 반복해서 수행한다.

4. Lesson learned

이번 과제를 통해 실제 MIPS machine이 어떻게 돌아가는지 하나하나 구현해보며 알 수 있었던 것 같다. 또한 개인적으로 중간고사 이후에 머릿속에서 까먹은 사소한것들때문에 구현할 때 디버깅에서 매우 애를 먹었다. 하지만 이를 통해 ZeroExtImm와 SignExtImm를 사용하는 instruction은 무엇인지 알게 되었고, 그리고 pc값이 증가하는 메커니즘을 다시 한번 알게 되었다. 또한 jal명령에서 pc가 증가해야 한다고 생각했지만 교수님께서 말씀하셨듯이 “PC를 조정하는 부분은 run_program에서 처리해야 할 일”이라는 것을 알게 되었다.
또한 endian을 수업시간에 간단히 짚고 넘어간 지라 정확히 무엇인지 몰랐는데 이번 과제를 하면서 endian문제를 해결하기 위해서 endian에 대해 찾아보며 MIPS는 Big-endian을 사용하고 있다는 것을 알게 되었다. 또한 Big-endian이 사람이 읽고 메모리에 쓰는 방식이 같아 훨씬 이해하기 편한 것 같다고 생각하였다.
또한 Fibonacci testcase에서 addi를 제대로 구현하지 않아서 push pop이 제대로 동작하지 않아서 segmentation fault가 났는데 이를 통해서 addi가 SignExtImm를 사용한다는것을 알게 되었다. 또한 slt명령의 경우 값을 그냥 비교하면 안된다는 것을 깨달았다. 이를 통해 컴퓨터에서 signed인지 unsigned인지가 매우 중요하다는 것을 알게 되었다.