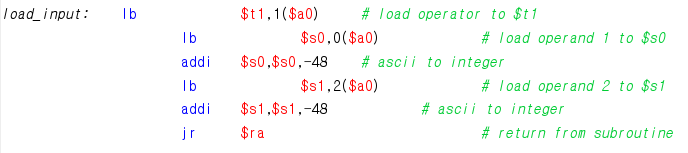
**컴퓨터 구조론 Project1 보고서**

**전자공학과 12181539 조권호**

1.



위의 부분에서 입력이 배열에 들어가기 때문에 바이트 단위로 두번째 들어온 문자를 $t1에 넣는것이다. 그런데 주어진 입력이 3+2와 같이 두번째 연산자가 연산자이면 3과 2를 더하는 코드는 문제없이 작동한다. 위의 코드는 말 그대로 문자로 인식해서 입력 받아 바이트 단위로 나눠서 48을 빼주면 아스키 코드에서 숫자가 되는 방법을 사용한 것이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

한 자리수 끼리의 연산은 위의 식을 이용하면 된다. 하지만 두 자리수 이상의 연산을 하면 위의 식은 작동하지 않는다. 따라서 입력을 받아 배열을 한칸씩 살펴보며 연산자가 있을 경우 beq를 사용하여 아래와 같이 분기를 나눠 이동하도록 하였다.

텍스트, 테이블이(가) 표시된 사진

자동 생성된 설명

이렇게 하면 마찬가지로 한자리수 끼리의 연산은 잘 작동한다.

위 함수는 사용자에게 입력 받은 string을 처리하는 함수이다. 동작을 살펴보면, **($a0)+4**의 메모리주소에 저장된 값을 **$t1**에 저장한다. 그리고 **$a0**의 메모리 주소에 저장된 값을 **$s0**에, ($a0)+8의 메모리 주소에 저장된 값을 $t2에 저장하고, 두 레지스터의 값에서 ‘0’의 ASCII 코드 값인 48을 빼서 사용자가 입력한 값을 저장하고 main함수로 돌아간다.

위에서 설명한 load\_input: 함수는 단순히 입력 string의 첫번째 자리에 있는 것을 operand1, 두번째 자리에 있는 것이 operator, 세번째 자리에 있는 것이 operand2 라는 것을 가정한 상황이다. 위의 테스트 입력에서 “21+4”는 error가 발생하지만 “8\*12”에서는 두번째 operand가 1까지만 읽히므로 프로그램이 계속 실행되어 논리적으로 오류가 있는 결과를 출력하는 것이다. 그래서 분기를 나누면 되겠다고 생각을 했다.

세자리 수 까지의 연산을 생각하여 작성하여 보면 아래와 같다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이렇게 나눈 후 add\_op2에서 다시 beq를 이용해 뒤의 숫자들의 자리수를 나누어 mul을 사용해 10,100을 자리수대로 곱하고 모두 더해서 완성하려 했지만 실행이 되지 않았다.

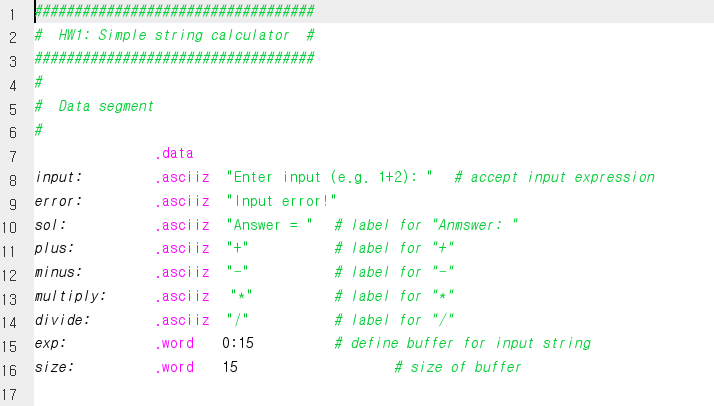
텍스트이(가) 표시된 사진

자동 생성된 설명

따라서 위의 방법 말고 다른 방법이 필요할 것 같다.

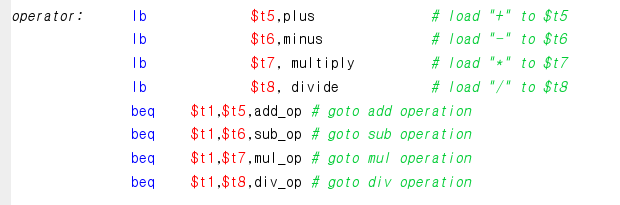
2.

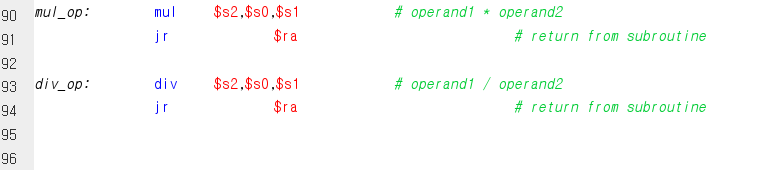
다시 곱셈과 나눗셈 기능을 추가하기 위해 data segment에 곱셈과 나눗셈 기호를 추가한다.



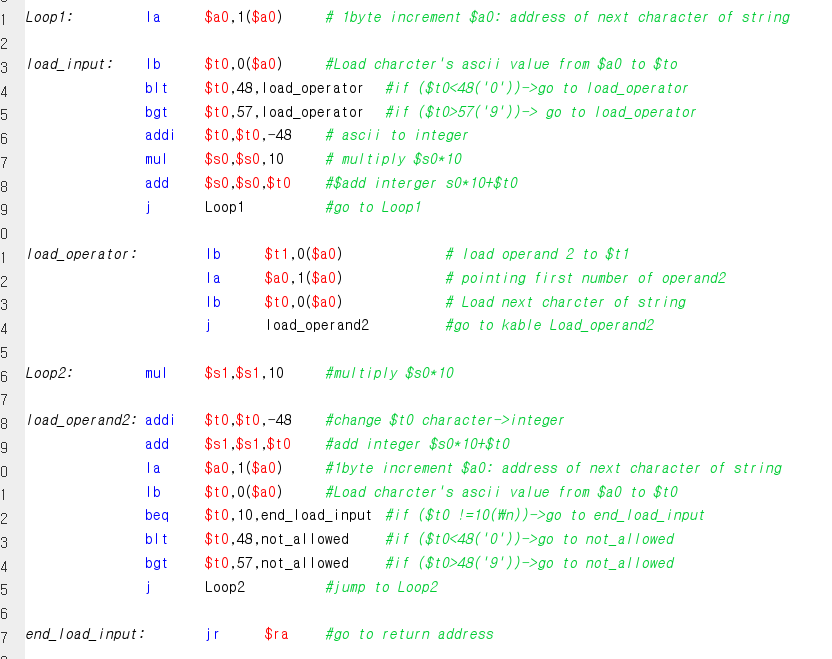
그 다음으로 연산자를 확인하는 함수 operator: 에 곱셈과 나눗셈 기호를 $t7, $t8에 저장하는 lb 명령어를 사용하여 추가한다. 그리고 $t1이 “\*” 또는 “/”인지 확인하고 각각 mul\_op, div\_op label로 점프하는 branch 명령어를 삽입한다.

각 연산을 구현한 부분에 mul\_op 함수와 div\_op 함수를 작성한다. 이 때 operator1,2는 각각 레지스터 $s0, $s1에 저장되어 있는 상태이다. 계산 결과는 $s2에 저장되어 print함수를 통해 출력된다.





여기까지는 1의 방법과 같다. 하지만 문제는 입력 스트링을 처리하는 input\_load 함수 때문에 두자리 수 이상의 연산이 안되는 점이다. 이러한 문제를 해결하기 위해, 첫번째 operand는 연산기호가 오기 전까지, 두번째 operand는 enter키의 입력이 들어오기 전까지 읽어 들여야 한다는 사실을 알 수 있고 이에 따라 load\_input: routine을 수정할 계획을 수립할 수 있다.

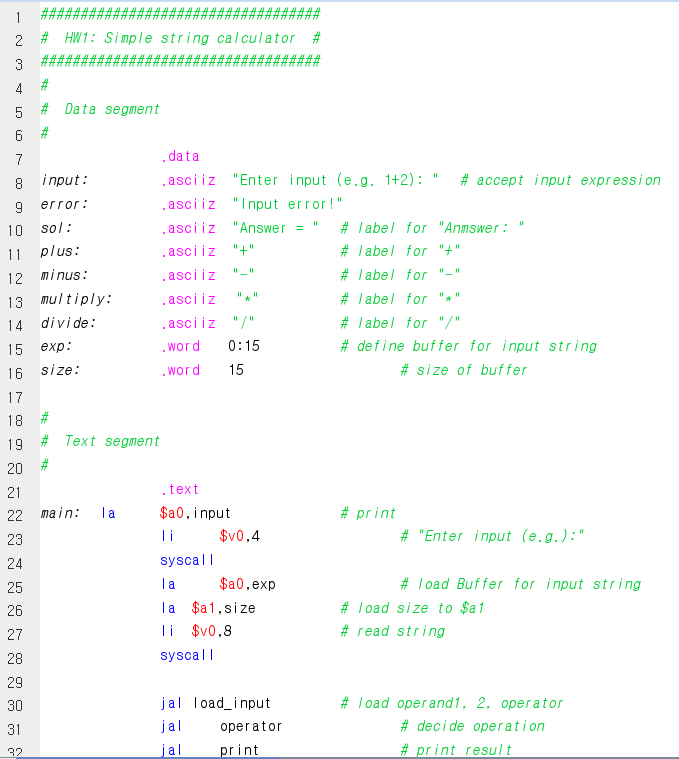


첫번째 operand를 입력 받을 때, 우선 $a0의 주소에서 가장 먼저 읽은 숫자를 $s0에 load한다. 그리고 $a0의 다음 주소가 가리키는 문자가 숫자인지 체크하고, 숫자라면 $s0의 값에 10을 곱하고 $a0가 가리키는 값의 숫자를 $s0에 더한다. 숫자가 아니라면 (예외상황이 아닐 경우) operand이므로 이러한 반복을 중지한다

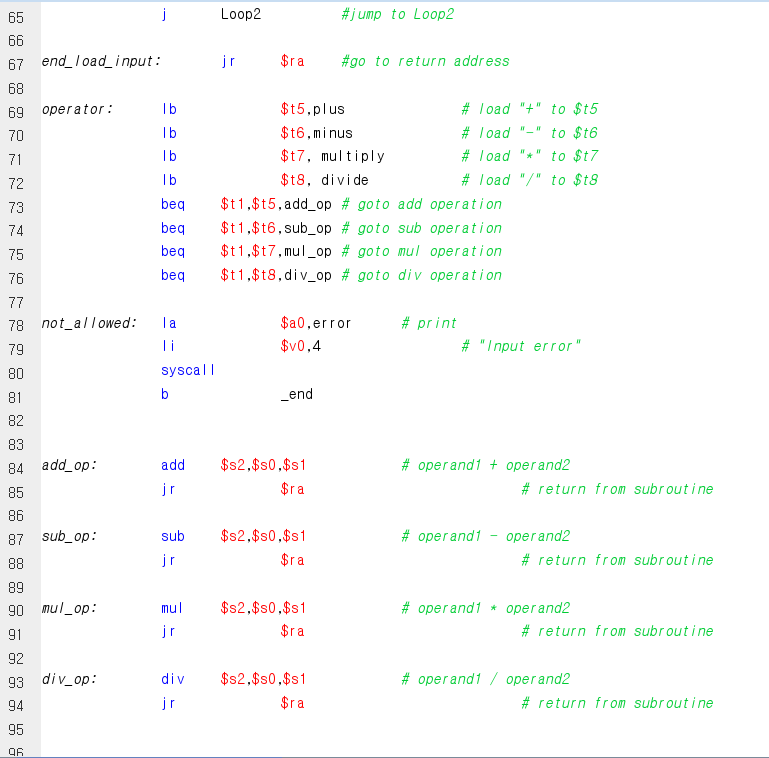
위의 원리는 두번째 operand를 입력 받을 때도 비슷하게 적용할 수 있다. 하지만 입력 받은 숫자를 모두 읽어낼 때까지 반복해야 하는 부분이 있어서 반복되는 부분을 위해 따로 label을 생성하여 반복문을 구현해야 할 것이다.

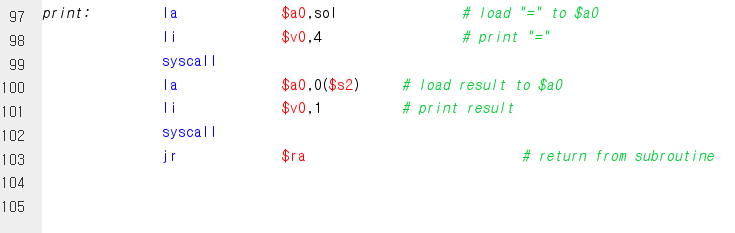
Load\_input의 기능을 하는 기존의 procedure를 load\_operand1과 load\_operand2로 나누었고, string의 중간에서 operator를 입력받으면 레지스터 $t1에 저장하여 나중에 따로 처리할 수 있도록 하였다. 이렇게 string의 처리가 끝난 후, jr 명령어에 의해 load\_input을 호출한 main 함수의 다음 코드로 이동한다. operator 함수에 의해 $t1에 저장된 operator의 종류에 따라서 각각의 연산을 수행하는 procedure로 branch되고, 계산된 값이 $s2에 저장되어 print함수가 출력하도록 한다.

마지막으로 아스키코드를 이용해서 -48을 하면 문자형식의 숫자가 숫자값을 갖는 원리를 이용했기 때문에 다른 문자가 들어와도 숫자로 인식하고 계산을 하는 문제가 있다. 따라서 문제를 해결하기 위해 blt(branch if less than) 명령어 다음에 bgt(branch if greater than)명령어를 통해 숫자를 벗어나지 않게 된다. 숫자나 연산자가 아닌 다른 입력이 올 경우 모두 load\_operator에서 처리되어 error메시지를 출력하게 되는 것이다. $t0가 읽어온 값이 엔터키(아스키코드의 10)이면 end\_load\_inpur으로 브랜치 하여 $ra의 주소로 return 한다. 그 후 $t0가 읽은 값이 숫자인지 체크하여 아니라면 not\_allowed procedure로 이동, 숫자가 맞으면 Loop2로 이동하여 동작반복을 계속 한다. 이 때 not\_allowed는 operator의 일부분을 분리한 것으로, operator 함수가 끝나면 자동으로 not\_allowed가 실행되므로 첫번째 operand와 operator입력에 대한 에러 메시지는 여전히 출력할 수 있다.









와 같이 코드를 구현하였고 실행결과는 아래와 같다.

