

CAMP

Homework 1:

Report of HW1

<5 free-days left>

Name : SeokBeom Lim (임석범)

Department : Mobile System Engineering

Student ID : 32203743

Table of contents

1. 서론	2
2. 본론	
1) 전체적인 개요	2
2) 각 함수 별 대략적 설명	2
3) 주요 변수 및 세부 사항 설명	
4) input.txt에 들어가는 명령어 정리	3
5) 예외처리	3
6) GCD 알고리즘 및 chat gpt/ input.txt 관련	3
3. 결론	13

본 코드는 VS를 통해 build 하였습니다.

[화면 캡처 및 작업 증거]

The screenshot shows the Visual Studio 2019 interface with the assembly view open. The assembly view displays instructions such as 'jump 0x3', 'com R1, R2', 'beq 0xd', 'sub R0, R2, R1', 'mov R1, 0', 'jump 0x3', 'com R1, R2', 'beq 0xd', 'sub R0, R2, R1', 'mov R2, 0', 'jump 0x3', 'com R1, R2', 'beq 0xd', 'mov R0, 1'. The background window shows the corresponding assembly code for these instructions, including comments like 'branch fail: goto next inst', 'branch succeed: move to [PC:10]', and 'branch fail: goto next inst'.

서론

컴퓨터의 첫 시작은 Von Neumann 구조로부터 시작하였다고 봐도 무방하다. 이전부터 올바른 계산과 효율적인 처리가 제시 되지 않은 것은 아니지만 Von Neumann 이 제시한 컴퓨터 구조는 현재에도 쓰일 정도로 유용하게 사용된다. Von Neumann 구조는 2가지의 특징을 가지고 있다. 첫번째 특징은 “Stored Program”으로 데이터를 저장하고 사용하는 특징을 가진다. 두번째 특징은 “Sequential process”로 연산을 단계별로 진행하는 특징을 가진다. 본 과제물은 이러한 Von Neumann 컴퓨터 구조를 C언어를 통해 구현하여 컴퓨터 구조에 대한 이해를 높이고자 한다.

본론

I. 전체적인 개요

본 코드의 목적은 효율적인 계산이 아닌 Von Neumann 컴퓨터 구조에 알맞은 구조를 구현하는 것이 목적이다. 따라서, 현실과 비슷하게 만들기 위해 구현한 것이므로 코드나 계산은 비효율적인 부분이 있다.

사람이 현실에서 컴퓨터를 쓸 때 입력을 하고 출력을 받는다. 입력을 받은 컴퓨터는 입력 값을 어셈블리어(명령어)로 바꾸고 이를 실행한 후 출력 값을 준다. 이를 참고하여, 본 코드는 input.txt로부터 입력 값을 받고 이를 다시 명령어로 바꾼 후 실행(execute)한다. 이에 따라, 코드에 3개의 구조체가 존재하며 각각의 구조체는 다음과 같다.

구조체)

Input : input.txt에 있는 명령어(연산자와 피연산자)를 저장하는 구조체. 한 줄에 최대 4개의 단어를 저장 할 수 있다.

Instruction : 받은 Input 명령어를 Processor 구조체가 실제로 수행 할 명령어들로 바꾸어 준다. 예를 들어, + 0x3 0x4를 input.txt에서 받았다면 이를 실제로 수행하기 위해 2개의 movi 명령어를 통해 레지스터에 0x3과 0x4를 저장하고 add 명령어를 통해서 더할 수 있도록 해준다.

Processor : Von Neumann 컴퓨터 구조에서의 CPU역할을 맡는다. 가상의 CPU를 표현했으며, 10개의 레지스터 PC(program counter)를 가진다.

II. 각 함수 별 대략적 설명

init_processor(Processor p)** : 프로세서를 초기화 한다. 모든 레지스터를 1로 설정하고, 프로그램 카운터를 0으로 설정한다. 초기화되지 않은 값에 접근할 우려가 있으며 초기값을 0으로 설정해 놓으면 zero-division error가 나올 수 있으므로 레지스터의 초기값은 1로 설정한다.

sub, add, mult, divi, com, jump, beq, movi, mov () : 각각 뺄셈, 덧셈, 곱셈, 나눗셈, branch, 레지스터간의 이동 및 비교 등의 기능을 수행한다. 연산 결과는 주로 R0(register 0번)에 저장되며 jump는 pc 값을 바꾸며 beq 같은 경우, 조건에 따라 pc값을 바꾼다.

save_char(Instruction i, char oper, char* op1, char* op2, char* op3, int* a)** : 분석된 명령어와 피연산자들을 Instruction 구조체에 저장하는 역할을 수행한다. 매개변수는 차례대로, 명령어, 연산자, 피연산자 3개, 마지막으로 포인터 a는 Instruction의 개수이며 함수의 마지막 부분에서 1 증가를 시켜주므로써 함수를 끝마친다.

is_oper_valid(const char op)* 및 *is_operand_valid(const char operand)** : 주어진 명령어 또는 피연산자가 유효한 형식인지 검사하는 역할을 한다.

input2inst(Input input, Instruction inst, int ip_cnt)** : Input 구조체 배열에서 읽은 명령어를 실제 수행 가능한 Instruction 구조체 배열로 변환한다. 또한, 위의 is_valid 계열 함수를 통해 대부분의 잘못된 명령어와 피연산자를 걸러내며 input2inst 내부의 int 변수(logic)를 이용하여 저장되는 숫자가 레지스터 관련된 숫자인지 16진수의 숫자인지 판별을 한다. 판별 이후, save_char 함수를 통해 instruction구조체에 조건에 맞게 저장한다.

parse(char a)** : 문자열 형태의 피연산자를 적절한 형태의 데이터(레지스터 번호 또는 16진수)로 변환한다. 변환한 후 int값을 리턴하는 함수이다.

execute(Instruction i, Processor p, int inst_cnt)** : 변환된 명령어를 순차적으로 실행한다. pc값을 통해 해당 명령어를 수행하며 명령어를 수행하기전 parse함수를 통해 char배열을 숫자로 변환한 후 명령어에 따라 레지스터 값을 변경하거나 pc를 조정하여 실행한 후, 다음 실행할 명령어를 결정하는 역할을 가진다.

III. 주요 변수 및 세부 사항 설명

위 함수들을 통한 자세한 코드 실행 흐름은 다음과 같다.

1) file io

먼저, 구조체를 선언해준다. Input의 경우 index를 0x400(1024)개까지 받을 수 있게 설정한다. 각 input에 비해 나올 수 있는 Instruction은 평균 2개정도 이므로 Instruction (변수 이름 : inst_reg) 구조체의 경우 그 2배인 0x800(2024)개의 index를 준비한다. 가상의 cpu인 Processor . .파일을 main함수에서 읽어준 후 input.txt.파일을 열어 파일을 한 줄씩 읽어준다. 파일을 한줄씩 읽어주며 받은 input의 개수가 0x400을 넘어갈 시 0x400개 이후의 input은 받지 못하게 한다. 각 줄을 받아주면서, 한 줄에 받는 input 단어 개수가 5개를 넘어가면, 저장되지 않게 한다. 각각의 input구조체에 들어가는 인덱스의 개수를 input구조체의 inst_cnt 변수에 저장한다.

2) input2inst

이후 input2inst를 통해 input을 instruction 에 넣어주게 한다. Input2inst 의 역할을 3가지로 첫번째는 유효하지 않은 연산자를 걸러주는 역할을 수행하며 두번째는 int logic 변수들을 통해 피연산자들이 레지스터 번호인지 0x관련된 번호인지 확인해준다. logic을 표로 정리하면 다음과 같다. 위 logic 값과 같지 않다면 잘못된 부분이므로 다음 input으로 넘어간다.

예시)) M R3 2, MI 3 0x3

logic (op1, op2)	logic1	logic2	logic3	logic4
0x, 0x	1	0	1	0
R, 0x	0	1	1	0
0x, R	1	0	0	1
R, R	0	1	0	1

또한, 잘못된 input을 받았다면 is_oper_valid와 is_operand_valid 함수를 통해서 유효성을 검사한 후 다음으로 넘어간다. 알맞은 input값을 받았다면, 조건에 맞게 input을 inst로 바꿔준다.

logic_temp 변수들의 경우, 받은 instruction 개수가 몇 개인지 파악해준다. logic_temp의 변수를 통해 명령어의 알맞은 피연산자의 개수를 판별해주기도 한다. 예를 들어, J 0x300 0x100의 경우 의도한 입력값보다 피연산자가 더 많은데 이를 logic_temp1의 조건으로 오류로 판별해주고 입력 값을 명령어로 바꾸지 못하게 만든다. 이와 다르게, 조건에 맞는 입력값의 경우 save_char 함수를 통해 instruction으로 알맞게 저장된다.

사칙연산 명령어의 (+-*/) 경우 movi (move imm)를 통해 두 피연산자를 레지스터 1번과 2번에 저장한 후 addi, mult, divi, sub를 통해 레지스터 연산을 할 수 있게 명령어를 저장해준다. 사칙연산을 통해 입력값을 넣을 경우 피연산자 2개는 레지스터 및 16진수의 값을 가질 수 있다. 이는 4가지 경우가 존재하는데, 0x 0x, 0x R, R 0x, R R이 있다. 해당 경우들은 logic변수를 통해 분류 해준다. 위 함수들을 반복한 후, input이 instruction구조체로 바뀐 횟수만큼 inst_cnt변수에 저장되고 저장된 변수는 총 주소(명령어 주소)의 개수로 사용하기 위해 반환해준다.

3) execute

execute 함수의 경우 3가지 특징이 있다. 첫번째는 instruction구조체의 oper(연산자)를 읽어낸 후 적합한 함수에 들어가게 도와준다. 두번째는 char배열 형태로 저장된 피연산자들을 int 형태로 반환하여 사용할 수 있게 해준다. execute함수에서 사용된 parse 함수의 경우가 int 형태로 반환해주는 함수이다. 마지막으로 pc의 값을 더해주어 다음 명령어를 실행 할 수 있도록 도와준다. 실행할 때 호출되는 함수들과 명령어는 다음과 같다.

- mov rt rs

rt (target register)값을 rs (source register) 값으로 바꾸어준다.

- divi rd rt rs

rt (target register)값을 rs (source register) 값으로 나누어준다.

이후, 나누어진 값은 rd에 저장한다.

- add rd rt rs

rt (target register)값과 rs (source register) 값을 더해준다.

이후, 더해진 값은 rd에 저장한다.

- **jump mem_add**

mem_add 로 pc값을 바꾸어준다

- **beq mem_add**

R0 값이 0과 같다면 pc값이 mem_add로 바뀐다.

- **com rt rs**

rt (target register) 값이 rs (source register)값과 같다면 pc값이 mem_add로 바뀐다.

- **movi rt imm**

rt (target register)값을 imm(immediate)값으로 바꾸어준다.

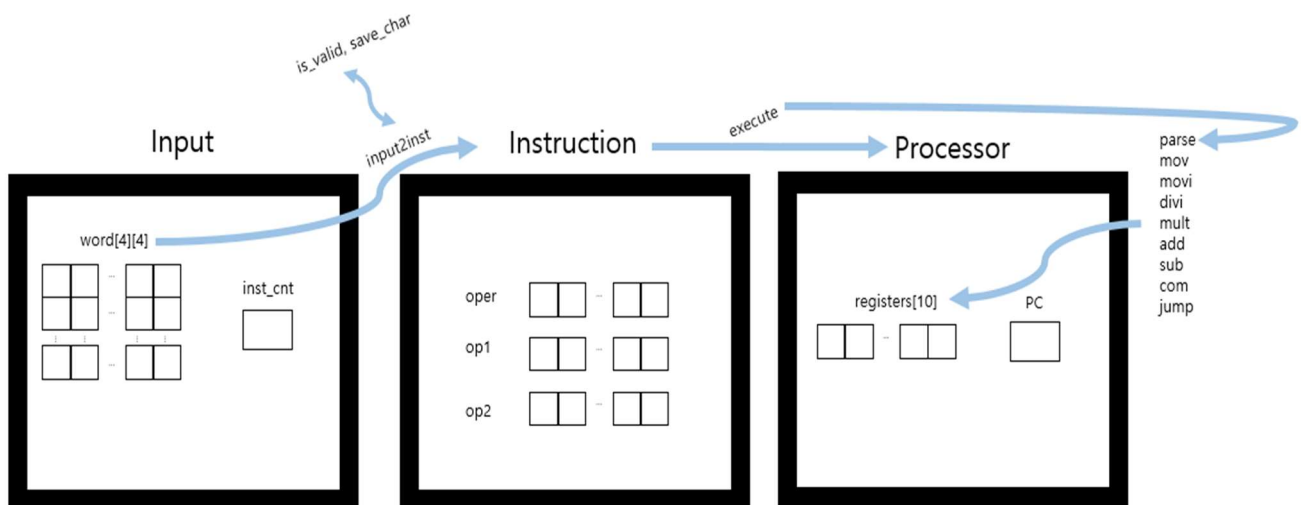
- **sub rd rt rs**

rt (target register)값을 rs (source register) 값으로 빼준다.
이후, 빼진 값은 rd에 저장한다.

- **mult rd rt rs**

rt (target register)값을 rs (source register) 값과 곱해준다.
이후, 빼진 값은 rd에 저장한다.

위 소개의 중요한 부분을 도표로 정리하면 다음과 같다.



[사칙연산 관련 input.txt 명령어]

- R#(0x#) R#(0x#)

첫번째 피연산자를 두번째 피연산자로 뺀다. 결과는 R0을 통해 알 수 있다.

+ R#(0x#) R#(0x#)

첫번째 피연산자와 두번째 피연산자를 더한다. 결과는 R0을 통해 알 수 있다.

*** R#(0x#) R#(0x#)**

첫번째 피연산자와 두번째 피연산자로 곱한다. 결과는 R0을 통해 알 수 있다.

/ R#(0x#) R#(0x#)

첫번째 피연산자를 두번째 피연산자로 나눈다. 결과는 R0을 통해 알 수 있다.

[그외 input.txt 명령어]

COM R# R#

com instruction과 동일

J 0x#

jump instruction과 동일

M R# R#

mov instruction 과 동일

MI R# 0x#

movi instruction 과 동일

BEQ 0x#

beq instruction 과 동일

V. 예외처리 : 코드 주석에 예외 처리에 관하여 기입 되어있다.

- 1) input.txt.를 받았을 때, 일정 개수를 넘어가서 Input 구조체 배열을 index를 넘어가면 input을 그만 받도록 처리
- 2) 한 줄에 5가지 이상 받는 instruction은 못 받게 하였다.
- 3) input2inst 함수에서 is_oper_valid를 통해 연산자가 의도한 입력값과 다르게 입력된다면, instruction에 저장하지 못하게 처리하였다. is_operand_valid 함수는 input.txt 숫자 앞에 R이나 0x가 붙어 있지않으면 input2inst의 조건문이 continue문을 통해 instruction 구조체에 들어가지 못하도록 처리하였다.
- 4) logic 변수들을 통해 R과 16진수가 연산자와 적합하지 않으면 instruction으로 저장되는 것을 막는다. 예를 들어, MI R0 R3의 경우 두번째 피연산자는 16진수 값이어야 하므로 이때, input이 instruction으로 저장되는 것을 막아준다.

logic_temp 변수들을 통해 피연산자의 개수가 맞는지 확인한다. J 0x300 0x700 같은 경우가 걸러진다.
- 5) 나누기를 할 때 0으로 나누어지는 것을 방지해준다.

VI. GCD 알고리즘 및 chat gpt/ input.txt 관련

GCD를 구현하기 위해 채택한 알고리즘은 유클리드 알고리즘이다. 두 수를 비교하며 큰 숫자를 점점 빼 간다. 계속 반복해서 뺀다면, 두 숫자가 같아지는데 같아진 숫자가 GCD에 해당한다.

[gcd 알고리즘]

```
int gcd(int a, int b) {  
    if (a == b) { return a; }  
    else if (a > b) { return gcd(a - b, b); }  
    else { return gcd(a, b - a); }  
}
```

다음 코드를 input.txt의 형식으로 바꾼다면 다음과 같으며 옆의 그림은 실행 결과이다.

[15와 96의 GCD값 구하는 알고리즘]

MI R1 0xf

MI R2 0x60

MI R3 0x1

COM R1 R2

BEQ 0xd

COM R0 R3

BEQ 0xa

$$-R_2 R_1$$

M R2 R0

J 0x3

- R1 R2

M R1 R0

```
[Input to Instruction]

[Execution]
inst count : 14
PC : 0      movi    R1    f
PC : 1      movi    R2    60
PC : 2      movi    R3    1
PC : 3      com     R1    R2
PC : 4      beq     0xd
PC : 5      com     R0    R3
PC : 6      beq     0xa
PC : 7      sub     R0    R2      R1
PC : 8      mov     R2    0
PC : 9      jump    0x3
PC : 0      R1      R1
PC : 4      beq     0xd
PC : 5      com     R0    R3
PC : 6      beq     0xa
PC : 7      sub     R0    R2      R1
PC : 8      mov     R2    0
PC : 9      jump    0x3
PC : 3      com     R1    R2
PC : 4      beq     0xd
PC : 5      com     R0    R3
PC : 6      beq     0xa
PC : 7      sub     R0    R2      R1
PC : 8      mov     R2    0
PC : 9      jump    0x3
PC : 3      com     R1    R2
PC : 4      beq     0xd
PC : 5      com     R0    R3
PC : 6      beq     0xa
PC : 7      sub     R0    R2      R1
PC : 8      mov     R2    0
PC : 9      jump    0x3
PC : 3      com     R1    R2
PC : 4      beq     0xd
PC : 5      com     R0    R3
PC : 6      beq     0xa
PC : 7      sub     R0    R2      R1
PC : 8      mov     R2    0
PC : 9      jump    0x3

R1 : 15
R2 : 96
R3 : 1
R0 : -1      ( 15 < 96 )
branch fail : goto next inst
R0 : -1      ( -1 < 1 )
branch fail : goto next inst
R0 : 81 = 96 - 15
R0 to R2 : 81
jump from 9 to 3
R0 : -1      ( 15 < 81 )
branch fail : goto next inst
R0 : -1      ( -1 < 1 )
branch fail : goto next inst
R0 : 66 = 81 - 15
R0 to R2 : 66
jump from 9 to 3
R0 : -1      ( 15 < 66 )
branch fail : goto next inst
R0 : -1      ( -1 < 1 )
branch fail : goto next inst
R0 : 51 = 66 - 15
R0 to R2 : 51
jump from 9 to 3
R0 : -1      ( 15 < 51 )
branch fail : goto next inst
R0 : -1      ( -1 < 1 )
branch fail : goto next inst
R0 : 36 = 51 - 15
R0 to R2 : 36
jump from 9 to 3
R0 : -1      ( 15 < 36 )
branch fail : goto next inst
R0 : -1      ( -1 < 1 )
branch fail : goto next inst
R0 : 21 = 36 - 15
R0 to R2 : 21
jump from 9 to 3
R0 : -1      ( 15 < 21 )
branch fail : goto next inst
R0 : -1      ( -1 < 1 )
branch fail : goto next inst
R0 : 6 = 21 - 15
R0 to R2 : 6
jump from 9 to 3
```

J 0x3

M R0 R1

PC : 3	com	R1	R2	R0 : 1	(15 > 6)
PC : 4	beq	0xd		branch fail : goto next inst	
PC : 5	com	R0	R3	R0 : 0	(0 = 1)
PC : 6	beq	0xa		branch succeed : move to [PC:10]	
PC : 10	sub	R0	R1	R0 : 9 = 15 - 6	
PC : 11	mov	R1	0	R0 to R1 : 9	
PC : 12	jump	0x3		jump from 12 to 3	
PC : 3	com	R1	R2	R0 : 1	(9 > 6)
PC : 4	beq	0xd		branch fail : goto next inst	
PC : 5	com	R0	R3	R0 : 0	(0 = 1)
PC : 6	beq	0xa		branch succeed : move to [PC:10]	
PC : 10	sub	R0	R1	R0 : 3 = 9 - 6	
PC : 11	mov	R1	0	R0 to R1 : 3	
PC : 12	jump	0x3		jump from 12 to 3	
PC : 3	com	R1	R2	R0 : -1	(3 < 6)
PC : 4	beq	0xd		branch fail : goto next inst	
PC : 5	com	R0	R3	R0 : -1	(-1 < 1)
PC : 6	beq	0xa		branch fail : goto next inst	
PC : 7	sub	R0	R2	R0 : 3 = 6 - 3	
PC : 8	mov	R2	0	R0 to R2 : 3	
PC : 9	jump	0x3		jump from 9 to 3	
PC : 3	com	R1	R2	R0 : 0	(3 = 3)
PC : 4	beq	0xd		branch succeed : move to [PC:13]	
PC : 13	mov	R0	1	R1 to R0 : 3	

마지막으로 chat gpt 이용 여부와 input.txt에 대한 추가적인 기록물이 있어서 이에 대해 설명하고자 한다. 코드 구현에 있어서 Chat GPT를 사용하지 않았지만, 코드 검증을 위한 input.txt의 랜덤 연산자와 피연산자를 얻기 위해 Chat GPT의 도움을 받았다. 또한, 예외 상황을 처리하기 위해 오류를 유발하는 랜덤 코드 예시를 위해 Chat GPT의 도움을 받았다. 아래는 Chat GPT의 도움을 받은 부분이다.

SBL 당신

input.txt에 쓸 값이 필요한데

형식은 operator op1 op2형식이고

operator에 들어갈 수 있는 것들은 + - * / M J BEQ MI COM이야

1) M: op1과 op2에 R과 함께 0부터 9까지의 숫자를 써줘야됨(숫자가 들어가서는 안 됨

3) MI는 op1에 R과 함께 0부터 9까지의 숫자를 써주고 op2에는 0x0부터 0xffff까지의 숫자를 넣어줘

4) BEQ와 J의 경우는 input.txt에 넣지마

5) COM의 경우 op1과 op2 안에 모두 레지스터 값이 들어가야함

6) 나머지 사칙연산은 다음과 같아.

op1,op2는 2가지로 이루어져있는데

첫번째 방법은 숫자를 표기하는거야. 16진수로 써주고 범위는 0x0부터 0xffff까지

두번째 방법은 R과 함께 0부터 9까지의 숫자를 써주는거고

위 조건들이 맞는 것들 100개 써줘

[코드 검증을 위한 랜덤 연산자와 피연산자 추출]



ChatGPT

1. ``/ R2 0xE4D``
2. ``M R7 R2``
3. ``COM R2 R6``
4. ``- R1 R3``
5. ``* R9 R9``
6. ``COM R7 R7``
7. ``* R2 0x6AD``
8. ``M R5 R5``
9. ``MI R5 0xFB46``
10. ``- 0xAD5 0x725``
11. ``+ R0 0x35A``
12. ``* 0xB3F R6``
13. ``/ R4 0xF5C``
14. ``M R3 R4``
15. ``COM R1 R2``
16. ``+ R2 0x6AB``
17. ``* 0xC9D 0x8F2``
18. ``M R6 R0``
19. ``MI R3 0xEAF8``
20. ``+ 0x4D2 R3``
21. ``* R8 R7``

[입력값 오류 해결능력 검증을 위한 오류 유발 연산자/피연산자 추출]



당신

input.txt에 오류가 될만한 코드만 집어서 30개만 줘



ChatGPT

! 코드 해석 중 오류 발생 ▾

입력 파일에 오류를 유발할 수 있는 30개의 예시를 만들었습니다. 이들은 잘못된 형식의 연산자, 레지스터 이름, 16진수 값 등을 포함하고 있어서 처리 과정에서 예외 상황이 발생할 수 있습니다:

1. `MII R6 R8`
2. `* RR1 RR0`
3. `JJ 0x81 R5`
4. `MM 0xx12 RR0`
5. `/ 0x1a RR1`
6. `MM x6c 0xb2`
7. `JJ x6c R7`
8. `MII R6 0xx12`
9. `M 0xe4 0xx12`
10. `J R5 0xe4`
11. `M 0xG1 x6c`
12. `/ 0xx12 RR2`
13. `BEQQ 0xx12 R0`
14. `MII RR2 RR1`

결론

과제를 통해 Von Neumann 구조에 기반한 컴퓨터 시스템을 구현을 하면서, 컴퓨터 구조의 원리와 프로그래밍 언어가 실제 하드웨어와 어떻게 상호 작용하는지 알 수 있었다. 이 과정에서 Input, Instruction, Processor 등의 구조체를 활용하여 실제 컴퓨터 구조와 유사하게 구현하니 사칙연산과 다양한 연산 처리에 대한 이해를 심화 할 수 있었다. 명령어만 실행 가능한 간단한 구조라서, 복잡한 연산을 처리하지 못하지만, Logic Gate와 Micro Processor의 구조를 추가한다면 복잡한 연산도 처리 할 수 있을 것으로 생각된다.