REPORT

제목: #project1 simple calculator

과목명: 고급 모바일실험1

담당 교수: 유시환 교수님

제출일: 2020년 4월 11일

학과/학번: 모바일시스템공학과 32182490

이름: 안소민

목차

Mobile processor: simple calculator

Somin an :32182490

1. Project introduction

간단한 계산기를 구현하는 것으로 작년 시스템 프로그래밍 과목에서 배운 MIPS code를 바탕으로 구현, 다음과 같은 연산자를 수행할 수 있다. 컴퓨터 구조 및 모바일 프로세서 과목을 수강함에 있어서 이 프로젝트를 통해 CPU가 instruction를 처리하는 구조를 공부한다.

- ✓ Binary arithmetic operations (+, -, /, *)
- ✓ Move operation (M – move some value to register ex) M R0 1 => R0 = 1)
- ✓ Jump operation (J – jump to some instruction ex) J 3 =>jump to third line)
- ✓ Compare operation (C – compare two value ex) C 3 4 => R0 =1)
- ✓ Branch or conditional jump (B – if R0=1, jump to some instruction)
- ✓ H (Hault, exit program)

2. Project goal and expectation effectiveness

컴퓨터는 지금까지 계속 발전하여 혁신적인 기술을 가지게 되었고 우리에게 꼭 필요한 기계가 되었다. 이에 따라 점점 컴퓨터와 관련하여 공부하는 사람들이 증가하고 코딩교육이 중요해진 오늘날이 되었다. 컴퓨터 전공자들 말고도 수준 높은 코딩실력을 가진 사람들이 많아지고 있다. 컴퓨터 전공자로서 이들과 맞서 경쟁력을 가질 필요가 있다. 성능이 더 좋은 프로그램을 만드는 것은 누구나 바라는 것이고 이 때 우리는 컴퓨터에 대한 이해를 갖추고 있으면 경쟁력을 갖출 수 있다. 성능이 좋은 프로그램을 만들기 위해서는 컴퓨터 내부에 대해서도 알아야 하는데 즉, 소프트웨어뿐만이 아니라 컴퓨터 내부도 알아야 할 필요가 있다. 메모리 용량 관리는 성능에 영향을 주고 메모리와 프로세서의 연결성을 이해해야 효율성이 좋은 프로그램을 만들 수 있다. 컴퓨터 구조를 이해함으로써 프로그램 개발 시 다른 사람과 다른 경쟁력을 가질 수 있게 된다. 따라서 우리는 컴퓨터 구조 전공 교과목을 배우는 것이다.

Project1은 컴퓨터 구조이론에서 배운 내용과 관련하여 계산기를 만드는 것으로, CPU가 instruction을 컴퓨터 내부에서 어떻게 처리하는지에 대한 이해도를 높이는 것이 핵심이다. 이 계산기는 CPU가 instruction을 처리하는 흐름과 비슷하게 설계함으로써 Data path을 이해하는

데에 도움이 된다. 작년 2학기에 배운 시스템 프로그래밍 과목에서 MIPS code 작성하는 것을 알고 있어야 하고 이를 적용하여 계산기를 구현한다. 첫 번째 프로젝트를 구현함으로써 앞으로 배울 컴퓨터가 프로그램을 처리하는 흐름, 구조들을 이해하는 데에 기초가 될 것이며 컴퓨터 관련 학과에서 기본적으로 배우는 이론들을 높은 이해도를 가지고 습득하는 것을 목표로 한다.

3. Concepts used in CPU simulation

- ISA (Instruction Set Architecture):
  명령어 집합 구조로, 마이크로프로세서가 인식해서 기능을 이해하고 실행할 수 있는 기계어 명령어를 말하며 Hardware Software interface역할을 한다. 이 계산기 프로그램에서는 MIPS code를 작성해 놓은 input program(txt 파일)이 필요하다. ISA는 10개의 레지스터와 5개의 instructions이 있다. 이 명령어들은 모두 string형식으로 구성되어 있으며 각 한 줄 당 하나의 명령어이다.
- MIPS (Microprocessor without Interlocked Pipeline-Stages):
  MIPS Technologies에서 개발한 RISC기반의 명령어 집합 체계이다. R type, I type, J type 3가지 종류의 명령어로 구성되어 있으며 그 중 opcode는 명령어가 실행할 연산의 종류를 정의한다. R type은 2개의 레지스터 값을 이용하여 연산을 하고, 다른 레지스터 하나에 연산 결과 값을 기록한다. I type은 연산할 값 하나는 레지스터에서 하나는 지정된 임의의 값을 사용하여 그 둘을 연산한 다음 다른 하나의 레지스터에 값을 기록한다. 이 프로그램에서는 J, R type 개념을 사용한다.
- FILE operations:
  MIPS code가 있는 파일을 열어 읽고 명령어들을 처리한다. 이때 fopen, fclose를 파일을 열고 마지막에 닫을 때 필요하다.
  형식:
  FILE *포인터이름 = fopen(파일명, 파일모드); -> 성공하면 파일 포인터를 반환, 실패하면 NULL 를 반환

Fclose(파일 포인터); -> int fclose(FILE*_stream); -> 성공하면 0 을 반환, 실패하면 EOF(-1)를 반환

파일에서 명령어(문자열)들을 한 줄씩 읽어 올 때 fgets/getline 함수를 이용한다.

형식:

Char *fgets(char*_Buffer, int _MaxCount, FILE* _Stream); ->성공하면 읽은 문자열의 포인터를 반환, 실패하면 NULL를 반환.

ssize_t getline(char** lineptr, size_t*n, FILE* stream); -> 스트림으로부터 전체의 라인을 읽어서, 버퍼 안에 텍스트를 저장하고 *lineptr안에 버퍼의 주소를 저장한다. getline함수는 GNU 라이브러리에서 제공하는 함수로, 개발환경 putty에서는 c언어에서 getline함수가 정의되어 있다.

- String operations:

  strtok(); -> 하나의 문자열을 어떤 단위로 단어 하나씩 나누는 함수이다. Opcode, operand1,2를 나누는데 필요하다.

  형식:

  Char* strtok(char*_String, char const* _Dlimiter); -> 자른 문자열을 반환, 더 이상 자를 문자열이 없으면 NULL반환.

- Other operations:

  atoi(); -> 문자열을 정수로 바꾸어 주는 함수로 문자열로 된 명령어 속에서 연산자를 정수로 바꾸어 준다.

  형식:

  int atoi(char const* _String); -> 성공하면 변환된 정수를 반환, 실패하면 0을 반환.

  strtol(); -> 특정 진법으로 표기된 문자열을 정수로 변환해주는 함수이다. 16진수 연산을 위해 이 함수를 사용한다.

  형식:

  Long strtol(char const*_String, char** _Endptr, int_Radix); -> 성공하면 변환된 정수를 반환, 실패하면 0을 반환. -> 예시 strtol(s1, NULL, 16);

4. Program Structure

Main 함수에 while문 반복하여 파일을 읽어 들임.

File open

```
+-------------------------------------------
|  file을 한 줄씩 읽기-> Fetch
|  opcode, operand1,2 나누기 -> Decode
|  opcode_examine(op, opr1, opr2);
|  각 opcode에 맞는 함수 호출
|  연산 -> execution
|  출력
+-------------------------------------------
```

① 파일 읽기

getline()함수를 사용하여 while loop속에서 명령어를 한 줄씩 읽어온다. 읽어온 instruction을 화면에 출력함. -1을 return하면 break

② opcode 검사 함수호출

strtok을 이용하여 opcode, operand 2개를 나눈 뒤 opcode가 무엇인지 검사하는 함수를 호출한다. => op_examine(op, opr1, opr2);

③ 각 opcode에 해당하는 함수호출

opcode 종류는 +, / , -, * 와 M,J,C,B 로, switch문을 이용하여 해당 연산을 하는 함수를 호출한다.

```
switch (*op) {
        case 'H':
                return -1;
        case 'M':
                operatorisM(op, opr1, opr2); => 해당 소스코드.
                break;
                ……
```

④ 연산실행

각 연산을 수행하고 해당함수에서 화면으로 instruction과 결과 값 출력.

✓ 사칙연산 – calculate()함수 호출 간단한 연산 실행(정수, 16진수,레지스터)

✓ M – operatorisM() 함수를 호출하여 opcode가 M일 때의 연산을 수행

✓ C – Condition()함수를 호출하여 operand 1,2 를 비교하여 연산 수행

✓ J - opcode가 J 가 되었을 때 file을 다시 열어 for loop을 사용해서 해당 줄로 pointer를 두고 함수호출을 하여 jump를 하는 방법을 사용함. Main 함수가 file을 읽는 loop문이기 때문에 겹쳐지는 걸 피하기 위해 main 함수의 loop를 break하고 jump함수를 이용하여 종료될 때까지 instruction을 읽어오게 함.

✓ B – opcode가 B가 되었을 때 R0 값이 1일때, jump() 호출.

☆ J 나 B 일 때는 operand2가 없기 때문에 함수 호출 시 “0” 값을 넣어 호출.

```
case 'J':
        jump(atoi(opr1));
        return -1;
case 'B':
if(Reg[0]== 1){
        jump(atoi(opr1));
        return -1;                    =>해당 소스코드
```

```
for (int i = 0; i < n; i++)
   { fgets(line, 1024, fp);}// 해당 코드 줄 pointer 이동
```

=> jump함수 해당 소스코드 부분

⑤ 출력 – operand가 레지스터일때, 값의 변화를 확인하기 위해 원래 전의 value 값 출력도 같이 출력.

- 추가구현: GCD

  MIPS code input txt파일에 작년 시스템프로그래밍 과목에서 배운 것을 바탕으로 직접 구현한 J, B, C를 이용해서 코드를 작성하여 실행. => 두 수를 큰 수에서 작은 수를 빼는 식으로 반복하여 숫자가 같아지는 때 그 수가 최대 공약수임을 알 수 있다.

5. Problems and Solutions

① 처음에는 사칙연산부터 코드를 작성하기 시작했습니다. 코드를 작성하기 전에 미리 어떻게 구현할지 구상을 하고 짜기 시작을 했고 약간의 문제가 있었던 것은 너무 코드가 길어졌습니다. 길어지면 오류를 찾을 때도 힘들고, 복잡해집니다. Opcode, operand1,2를 함수 인자로 넣어 연산하는 함수를 호출해서 연산을 수행시키려 하려고 했습니다. 하지만 opcode는 여러 개이고 각 opcode마다 다른 연산을 수행하기 때문에 코드가 길어질 수밖에 없었습니다. 따라서 저는 함수를 여러 개 만들기로 했습니다. opcode, operand1를 검사하는 함수, opcode에 따라 각각 다른 함수를 호출하게 코드를 설계했습니다.

② 사칙연산구현을 다 하고 난 뒤 jump 구현을 시작했습니다. Jump 구현을 어떻게 할지 구상을 하다가 파일에 있는 MIPS code들이 어딘 가에 저장되어 가져와야 할 것 같다는 생각이 들어 연결리스트 Queue를 이용해서 구현하기로 생각을 했습니다. 연결리스트로 구현하는 것은 쉽지 않았고 구조체 선언, 큐 관련 함수 때문에 코드가 길어져서 오류를 찾는 것도 쉽지 않았습니다. => 당시 다른 과목 과제들도 있었고 해결을 빨리 못하면 할 일을 끝내지 못하는 상태가 될 거 같아 다른 방법을 고안해 보다 다시 파일을 열어 for 반복문을 사용하여 해당 줄로 점프할 수 있도록 jump함수를 만들어서 구현에 성공했습니다. 결론은 처음에 구현을 시작하기 전에, 하나의 방법만을 생각하지 말고 최소 2개 이상을 방법을 생각하고 가장 효율적이고 접근이 쉬운 방법으로 구현을 했다면 더 좋을 것 같습니다.

③ Segmentation fault 문제: 프로그램은 실행이 되는 데, 맨 마지막에 이 오류가 출력되어 해결하고자 gdb를 이용해서 오류가 생긴 위치를 찾아보았지만 알아볼 수 없는 형식의 글이 나와 => 구글링을 하여 segmentation fault가 어떤 건지 어떤 문제로 인해 발생하는 문제인지 분석을 하고 다시 소스 코드를 보고 op_examine함수를 호출할 때 J나 B는 opr2가 없으니 opr2변수는 Null값인채로 계속 인자로 존재했었고 대신 "0" 값을 임의로 대입해서 문제를 해결할 수 있었습니다.

④ 출력 문제: operand가 될 수 있는 것이 16진수, 정수, 레지스터를 구분하기 위해 if else구문을 사용했습니다. 한번은 결과 값들이 출력이 잘 되는데 한 번은 다른 값이 출력되는 등 문제가 생겼습니다. => 해당 명령어를 처리하는 함수로 돌아가서 보니 if else를 너무 많이 사용하다 보니 copy+paste 자주해서 다른 경우에 같은 조건문이 붙어서 그랬던 것이었습니다. 따라서 저는 리팩토링 능력이 갖출 필요가 있다고 생각했습니다.

⑤ J 오류 발생: J를 구현에 성공하고 다른 함수 구현도 마쳐서 GCD 구현을 해봤는데 제 GCD 코드에는 J, B 두 개의 J가 있는데 실행이 계속 반복되었습니다. 그래서 다른 간단한 코드를 만들어 J를 두 번 실행하도록 했더니 이번에도 실행이 멈추질 않았습니다. => J구현을 할 때 J를 한번 사용만 해서 실행을 해보고 두 개이상일때는 구현해보지 않았기에 오류를 알았고, 제 프로그램의 실행 흐름을 천천히 따라가 보니 jump함수는 계속 호출이 되는데 그 안의 while loop가 남아있음에도 return 종료를 시키지 않아서 함수가 계속 종료되지 않고 실행이 되던 것임을 알고 jump을 할 때 -1 값을 반환하여 종료하도록 하니 실행이 정상종료가 되었습니다. 결론적으로 저는 함수 구현을 할 때 발생할 문제를 예측을 하지 못했으니 여러가지 경우를 실행해보는 연습을 필요하다고 생각했습니다.

⑥ GCD 구현 오류: gcd 구현을 하기 위해 gcd MIPS 코드를 텍스트 파일에 작성하여 실행을 하는데 아무리 실행을 해도 무한 실행이 되었습니다. 제 MIPS

code 구현이 잘못되었나 생각해서 직접 몇 번을 다시 써보고 실행했지만 코드는 틀리지 않았습니다. 출력이 너무 빨리 진행되다 보니 앞에 어떤 값이 잘못 들어갔는데 출력화면이 초반 부분이 잘려 볼 수 가없어서 오류를 못 찾았습니다. => 리눅스 환경에서 쓸 수 있는 sleep함수를 사용해서 2초에 한번씩 결과 값을 출력하도록 만들고 다시 오류를 찾아보니 제가 -1 음수를 연산할 때 숫자임을 확인하는 if 조건문을 보니 양수만 알아볼 수 있도록 조건을 넣어 놔서 atoi함수를 써도 값이 다른 값이 들어가 무한 반복이 만들어진 것이었습니다. 8시간동안 다른 문제가 있는지 찾다가 조건문에 특정 부분만 지우면 되는 문제였고 허무했습니다. 저는 디버깅 하는 능력을 키우기 위해 여러 경험이 필요하다고 생각했습니다.

6. Build Environment

Compilation: Linux Assam, with GCC

To compile, please type:

gcc calculator_32182490.c -o calculator_32182490

To run, please type:

./calculator_32182490

You should have input.txt in the same directory.

GCD 실행하실 때는, GCD.txt 파일을 사용하시면 됩니다. (input.txt로 code복사)

7. Screen capture

✓ 사칙연산, M

```
[1] + 1 2
opcode: + operand1: 1 operand2: 2
result: R0 = 3

[2] * 2 -2
opcode: * operand1: 2 operand2: -2
result: R0 = -4

[3] - 3 4
opcode: - operand1: 3 operand2: 4
result: R0 = -1

[4] / 10 5
opcode: / operand1: 10 operand2: 5
result: R0 = 2

[5] * 0x3 0x4
opcode: * operand1: 0x3 operand2: 0x4
result: R0 = 12

[6] H
somin18@assam:~/project1$
```

```
[1] + 1 4
opcode: + operand1: 1 operand2: 4
result: R0 = 5

[2] M R2 R0
opcode: M operand1: R2(value: 0) operand2: R0(value: 5)
result: R2 = 5

[3] M R3 10
opcode: M operand1: R3(value: 0) operand2: 10
result: R3 = 10

[4] M R1 0x12
opcode: M operand1: R1(value: 0) operand2: 18
result: R1 = 18

[5] H
somin18@assam:~/project1$
```

✓ J (jump)왼쪽: J가 무한 반복이 아닐 때 오른쪽: 무한반복(코드상 메모리 지정->용량 초과)

```
[1] + 1 4
opcode: + operand1: 1 operand2: 4
result: R0 = 5

[2] M R2 R0
opcode: M operand1: R2(value: 0) operand2: R0(value: 5)
result: R2 = 5

[3] * 4 5
opcode: * operand1: 4 operand2: 5
result: R0 = 20

[4] J 6

[6] + 10 34
opcode: + operand1: 10 operand2: 34
result: R0 = 44


[7] / 100 10
opcode: / operand1: 100 operand2: 10
result: R0 = 10


[8] H
somin18@assam:~/project1$
```

```
[2] J 1

[1] + 1 3
opcode: + operand1: 1 operand2: 3
result: R0 = 4


[2] J 1

[1] + 1 3
opcode: + operand1: 1 operand2: 3
result: R0 = 4


[2] J 1
Segmentation fault (core dumped)
```

✓ C (compare)

```
[1] + 1 4
opcode: + operand1: 1 operand2: 4
result: R0 = 5

[2] M R2 R0
opcode: M operand1: R2(value: 0) operand2: R0(value: 5)
result: R2 = 5

[3] * 4 5
opcode: * operand1: 4 operand2: 5
result: R0 = 20

[4] C 4 10
opcode: C operand1: 4 < operand2: 10
result: R0 = 1

[5] C 30 10
opcode: C operand1: 30 < operand2: 10
result: R0 = 0

[6] C 3 R2
opcode: C operand1: 3 < operand2: R2 (value: 5)
result: R0 = 1

[7] C R2 R0
opcode: C operand1: R2 (value: 5) < operand2: R0 (value: 0)
result: R0 = 0

[8] H
somin18@assam:~/project1$
```

=>C (compare)

✓ B (Branch)

```
[1] + 1 4
opcode: + operand1: 1 operand2: 4
result: R0 = 5

[2] M R2 R0
opcode: M operand1: R2(value: 0) operand2: R0(value: 5)
result: R2 = 5

[3] * 4 5
opcode: * operand1: 4 operand2: 5
result: R0 = 20

[4] B 6
[5] - 3 2
opcode: - operand1: 3 operand2: 2
result: R0 = 1

[6] M R1 R0
opcode: M operand1: R1(value: 0) operand2: R0(value: 1)
result: R1 = 1

[7] B 9

[9] C R1 R0
opcode: C operand1: R1 (value: 1) < operand2: R0 (value: 1)
result: R0 = 1


[10] H
```

✓ GCD (10,5) -> 5가 결과값이 나옴.

```
[1] M R2 10
opcode: M operand1: R2(value: 0) operand2: 10
result: R2 = 10

[2] M R1 5
opcode: M operand1: R1(value: 0) operand2: 5
result: R1 = 5

[3] - R2 R1
opcode: - operand1: R2(value: 10) operand2: R1(value: 5)
result: R0 = 5

[4] + R0 1
opcode: + operand1: R0(value: 5) operand2: 1
result: R0 = 6

[5] B 15
[6] - R0 1
opcode: - operand1: R0(value: 6) operand2: 1
result: R0 = 5

[7] M R3 R0
opcode: M operand1: R3(value: 0) operand2: R0(value: 5)
result: R3 = 5

[8] C 0 R3
opcode: C operand1: 0 < operand2: R3 (value: 5)
result: R0 = 1

[9] B 13

[13] M R2 R3
opcode: M operand1: R2(value: 10) operand2: R3(value: 5)
result: R2 = 5


[14] J 3
```

```
[14] J 3

[3] - R2 R1
opcode: - operand1: R2(value: 5) operand2: R1(value: 5)
result: R0 = 0

[4] + R0 1
opcode: + operand1: R0(value: 0) operand2: 1
result: R0 = 1


[5] B 15

[15] H
somin18@assam:~/project1$
```

✓ 사칙연산, C, B, J 가 모두 들어간 연산

```
[1] + 1 3
opcode: + operand1: 1 operand2: 3
result: R0 = 4

[2] * 2 -3
opcode: * operand1: 2 operand2: -3
result: R0 = -6

[3] C 6 5
opcode: C operand1: 6 < operand2: 5
result: R0 = 0

[4] B 3
[5] - 4 5
opcode: - operand1: 4 operand2: 5
result: R0 = -1

[6] M R2 R0
opcode: M operand1: R2(value: 0) operand2: R0(value: -1)
result: R2 = -1

[7] M R1 R2
opcode: M operand1: R1(value: 0) operand2: R2(value: -1)
result: R1 = -1

[8] J 10

[10] M R3 8
opcode: M operand1: R3(value: 0) operand2: 8
result: R3 = 8

[11] H
```

=> 제출한 input.txt에 해당합니다.

8. Personal feelings

저번 학기 때에는 다른 학기와 비교해서 코딩을 많이 하지 않아서 코딩 공부 덜 된 상태였기 때문에 이번 학기 프로젝트1을 하면서 프로그래밍 실력이 많이 줄었을 것 같다는 생각에 프로그램 구현에 있어서 많은 걱정이 먼저 앞섰습니다. 확실히 조금 쉬다 보니 문법들이 조금 틀리거나 오류가 생성되었을 때 대처도 효율적이지 못했던 것 같습니다. 어려워도 이틀이면 어느 정도는 다 구현할 수 있었을 거라고 생각했지만 전혀 아니었습니다. 중간 중간에 오류가 생성되고 디버깅 하는 시간도 많이 걸렸었습니다. 그 시간이 많이 차지해서 생각보다 오랜 시간 동안 이 프로젝트를 만들었던 것 같습니다. 오류의 원인을 찾아보고 수정해본 결과 거의 대부분은 작은 실수, 오타에서 나온 것이었습니다. 그 때마다 너무 허무하고 스트레스가 심해서 힘들었던 것 같습니다. 앞으로 이런 작은 실수를 줄이기 위해서는 미리 했던 실수들을 기록해 두거나 잘 되지 않는다고 계속 앉아서 컴퓨터만 바라본다고 해결되지 않기 때문에 조금씩은 휴식이 필요하다는 것을 알게 되었습니다. 이렇게 화면에 무엇인가를 출력해야 하는 프로그램을 구현할 때에는 sleep함수를 이용해서 어디부분에서 출력이 의도한 바와 다르게 잘못되었고 오류를 찾는데 도움이 된다는 것을 알게 되었습니다. 오류가 자주 생성되었다고 너무 힘들고 급하게 고치려 들지 말고 차분히 생각해서 문제를 해결하는 것이 더 도움이 되었던 것 같습니다.

첫번째 프로젝트는 쉽지 않았지만 파일에서 읽어와서 단어별로 나누고 해석하여 연산처리 하는 등 cpu가 instruction을 처리하는 것과 유사하는 수업시간에 배웠던 컴퓨터 구조 흐름을 이해하는 데에 도움이 되었던 프로젝트였던 것 같습니다. 이번 프로젝트를 계기로 실수를 좀 더 줄이고 문제해결 능력이 향상되었으면 좋겠습니다.