REPORT

이름: 안소민

단국대학교

제목: #project2 single cycle simulator

제출일: 2020 년 5 월 8 일

Freedays left: 4

Mobile processor: single cycle simulator

Somin an :32182490

I. Project Introduction

MIPS 시뮬레이터를 만드는 것으로, 프로젝트 1에서는 MIPS 의사 코드로 된 txt파일을 읽어서 간단한 계산기를 만드는 것이었지만 프로젝트 2에서는 MIPS ISA에 따라 컴퓨터가 실제 일을 수행할 때 하는 그 과정에 (Fetch, Decode, Execute, Memory Access, Write Back)맞춰 하나의 명령어를 single cycle로 처리하는 시뮬레이터를 구현한다. Binary 파일을 읽어서 실제 메모리가 있다고 가정하고 샘플 바이너리 파일을 읽어오는 것으로 프로그램을 구현한다. MIPS ISA에는 31개의 정수 명령어가 있으며 각 명령어 type은 R type, I type, J type으로 분류되며 32개의 register를 정의한다. MIPS Green sheet를 참고할 수 있으며 프로젝트에서 구현하는 명령어는 총 24개로 아래와 같다.

- ✓ add, addi, addiu, addu, sub, subu
- ✓ and, andi, nor, or, ori,
- ✓ beq, bne,
- ✓ j, jal, jr,
- ✓ lui,
- ✓ lw, sw,
- ✓ slt, slti, sltu,
- ✓ sll, srl

II. Project Goal

컴퓨터가 실제로 일을 수행할 때 하나의 cycle 에 하나의 명령어 수행처리 하는 single cycle 과정에 대해 수업시간에 배운 이론내용을 이해하고 프로젝트 2 에서 MIPS 시뮬레이터를 만듦으로써 single cycle datapath 를 더 정확하게 이해하는 것이 목표이다. 수행처리 단계 Fetch -> Decode-> Execute -> Memory Access -> Write Back 과정을 이해하고 구현하는 것을 통해 기본적인 ISA 역할과 그 수행과정을 배운다.

III. Concepts used in CPU simulation

- ISA: hardware software interface

  명령어 집합 구조로, 마이크로프로세서가 인식해서 기능을 이해하고 실행할 수 있는 기계어 명령어를 말하며 Hardware Software interface역할을 한다. 이 시뮬레이터에서는 바이너리 파일을 읽어서 명령어를 수행한다. ISA는 32개의 레지스터와 24개의 instructions이 있다. 이 명령어들은 모두 16진수 형식으로 구성되어 있으며 각 한 줄 당 하나의 명령어를 수행한다.

- 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 opcode와 function code가 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 I, J, R type 개념을 모두 사용한다.

- File operations:

  바이너리파일을 열어 읽고 각 명령어들을 처리한다. 이때 fopen, fclose를 파일을 열고 마지막에 닫을 때 필요하다. 파일을 열 때 여러 모드가 있는데 r, w, a read write append 등이 있고 이 프로젝트에는 이진형식 파일을 읽기 때문에 rb 모드를 사용한다. 파일 내용을 가져올 때에는 fread/fgets/fgetc 등이 있고 이 프로젝트에서는 fread를 사용한다.

- General Purpose Registers

  프로그램을 수행할 때 레지스터를 이용해서 메모리에 저장하기도 하고 연산을 하기도 한다. MIPS에서는 32개의 레지스터가 있고 여기서 29번 레지스터는 stack pointer 레지스터이고 31레지스터는 return address를 담고 있는 레지스터이다.

IV. Program Structure

프로그램은 main 함수에서 해당 샘플 바이너리 파일을 열고 fopen의 return 값을 인자로 하는 single_cycle(FILE* fp) 함수를 호출한다.

single_cycle 함수:

- While문을 이용하여 instuction을 메모리로부터 fectch
- Fectch한 명령어를 decode한다. (opcode, rs, rt, rd, immediate, function)
- Switch문을 이용하여 각 명령의 opcode를 검사
- 해당 opcode나 function code에 맞는 명령어를 execute
- 필요시 memory access
- 연산한 뒤 결과 값을 memory 나 register 나 pc value에 write back
- 만약 jump 명령어가 아니라면 pc = pc + 4
- 이 모든 것을 while문으로 반복하는 형태

main함수에서 각 샘플파일들을 읽고 fetch decode memory access write back 과정들을 반복적으로 수행하기 때문에 single_cycle이라는 함수를 따로 만들어 구현했다. 함수 호출하고 난 후 intel cpu는 16진수로 되어있는 명령어들을 거꾸로 읽어서 수행한다. 그래서 명령어 파일을 읽어들이면 거꾸로 명령어가 fetch하기 때문에 원래 명령어로 다시 정리할 수 있도록 while문을 이용해서 명령어를 정리하고 Memory[0x100000/4] 배열을 이용해서 저장한다.

다음 while문에서 명령어를 하나씩 읽어서 fetch decode를 하고 switch 문을 이용해서 opcode를 이용해 각 24개의 명령어가 수행되도록 구현했다. MIPS Green Sheet를 참고했는데, 같은 opcode를 가진 명령어들이 있었기 때문에 switch문 안에서 겹치는 몇 개의 명령어만 function code를 이용해서 if, else if 문을 만들어서 execute를 구현했다. switch문 안에 execute, memory access, write back을 모두 구현하고 jump instruction jump, jal, jr 와

bne, beq 와 같이 점프할 수 있는 명령어 일때는 pc value 값이 pc = pc + 4가 아니므로 따로 신호 변수를 두어 pc 가 제대로 명령어 위치를 따라 갈 수 있도록 구현했다.

※추가구현 - jalr (jump and link register)

추가 구현으로 jalr instruction 명령어를 이용하여 jump and link을 수행한다. 샘플파일은 기존 파일인 fib.bin 파일을 수정하여 fib2.bin을 만들어서 구현했다. 어떻게 구현할지 고민하다가 jal을 대신해서 수행하기 때문에 mips 코드로 fib.bin을 출력하여 jal을 수행하는 명령어를 찾아서 fib.bin에 들어가서 해당 명령어를 수정했다.

< Jump and Lind Register Format>

	opcode	operation
jalr	001001	reg[31] = pc + 4, pc = reg[rs] * 4

Pc 값에 rs인덱스 레지스터 값을 할당해주는 것이 문제였는데, jal 명령어 앞 부분에 addi 명령어 연산을 이용하여 rs인덱스 레지스터에 jal일 때의 jump_adress(0x10) 값을 만들어 주고 jal 명령어 부분에 jalr 명령어로 수정했다. rs인덱스의 값은 이 fib.bin 프로그램에서 32개의 레지스터 중에서 사용하지 않는 레지스터 중에 28을 인덱스로 하여 address 값을 대입하는 방식으로 fib2.bin 파일을 만들었다. fib.bin 파일에서1c, 88, a4 번째 명령어(0c000010)를 03800009(opcode가 9일 때 jalr), 그 앞에 21bc0010(jump address를 register[rs]/reg[28]에 할당) 명령어를 추가했다.

V. Problems and solutions

프로젝트1 할 때 실수가 많아서 디버깅 하는 데 많은 시간이 걸려서 힘들었다. 그 이유는 if else문을 많이 사용하다 보니 복사 붙여넣기를 자주 하다가 코드가 틀려서 오류 부분을 찾기가 어려웠던 것이다. 이번에는 그 실수를 줄이기 위해서 그리고 쉽게 알아보기 위해서 switch문을 사용하여 execute를 했다.

MIPS Green Sheet를 보면서 코드를 구현했는데 모든 명령어 구현을 하고 실행을 시켜보니 몇 가지 오류가 발생했다. 계속 무한으로 프로그램이 출력되어서 또 출력속도가 빠르기 때문에 sleep함수를 이용해서 천천히 원인을 살펴봤다. Bne 구현에서 Branch address라고 되어 있어서 simm이 아닌 imm라고 생각하고 실행시켰기 때문에 오류가 발생한 거였고 imm로 수정했다. 또한 ori 명령어 부분에서 green sheet에 zeroEximm 부분이 imm 값을 의미하는 줄 모르고 simm을 사용해서 오류가 났었다. 구현하기 전에 이렇게 데이터가 필요한 것들 중에서는 정확히 모른다면 구체적으로 의미 등 무엇인지 먼저 찾아보는 것이 더 효율적일 것 같다고 생각했다.

그 밖에도 바이너리 파일을 수정할 때에도 조금 어려웠던 것 같다. 수정은 했는데 그대로 16진수 파일로 저장할 수 없기 때문에 다시 bin 파일로 바꿔야 하는데 2진수 파일을 16진수 파일로 볼 수 있는 %!xxd 처럼 그 반대 명령어가 있는지 여러 번 구글링을 해보고 친구의 도움으로 해당 명령어가 %!xxd -r 인 것을 알아냈다.

Input4.bin 파일이 엄청 길다는 것은 알았지만 수행시간이 이렇게 오래 걸릴 줄은 모르고 아무리 기다려도 끝이 나지 않아서 오류가 있는지 한 시간을 들여다봤다. 친구에게 물어봐서 일반적인 실행을 한다면 20분 정도가 걸린다는 말을 듣고 실행파일을 이용하면 빨리 결과 값이 출력된다는 동료의 이야기를 듣고 실행파일을 만들어서 1분안으로 결과가 출력되는 것을 확인했다. 오류가 아니라 시간이 오래 걸렸던 것이다. 이렇게 실행 파일을 이용하면 훨씬 빠른 프로그램 수행을 할 수 있다는 것을 알게 되었다.

VI. Build environment

Compilation: Linux Assam, with GCC

To compile, please type:

gcc single_cyclesomin.c -o single_cyclesomin

To run, please type:

./single_cyclesomin

* main 함수에 각 bin 파일을 fopen하는 코드를 simple.bin 만 남겨두고 주석처리를 했습니다. 다른 파일을 실행시켜 보실 때에는 수정 만하시거나, 주석처리를 제거하시면 됩니다.

VII. Screen Capture

```
--------------------------------------------------------------------
[Fetch] 0x03e00008
[Decode] opcode(0) rs: 31 rt: 0 rd: 0 func(0x8) simm(8) <jr>
pc : 0x1c
pc = reg[31]
pc : 0x1c ----> pc : 0xffffffff

******************result*********************
reg[2] (final return)value: 0
number of instructions: 8
number of R type instructions: 4
number of I type instructions: 4
number of J type instructions: 0
number of memory access instructions: 2
number of taken branches instructions: 0
*********************************************
```

<캡처1: input file- simple.bin 결과 출력>

```
--------------------------------------------------------------------
[Fetch] 0x03e00008
[Decode] opcode(0) rs: 31 rt: 0 rd: 0 func(0x8) simm(8) <jr>
pc : 0x24
pc = reg[31]
pc : 0x24 ----> pc : 0xffffffff

******************result*********************
reg[2] (final return)value: 100
number of instructions: 10
number of R type instructions: 3
number of I type instructions: 7
number of J type instructions: 0
number of memory access instructions: 4
number of taken branches instructions: 0
*********************************************
```

<캡처2: input file-simple2.bin 결과 출력>

```
--------------------------------------------------------------------
[Fetch] 0x03e00008
[Decode] opcode(0) rs: 31 rt: 0 rd: 0 func(0x8) simm(8) <jr>
pc : 0x68
pc = reg[31]
pc : 0x68 ----> pc : 0xffffffff

******************result*********************
reg[2] (final return)value: 5050
number of instructions: 1330
number of R type instructions: 409
number of I type instructions: 920
number of J type instructions: 1
number of memory access instructions: 613
number of taken branches instructions: 102
*********************************************
```

<캡처3: input file-simple3.bin 결과 출력>

```
--------------------------------------------------------------------
[Fetch] 0x03e00008
[Decode] opcode(0) rs: 31 rt: 0 rd: 0 func(0x8) simm(8) <jr>
pc : 0x2c
pc = reg[31]
pc : 0x2c ----> pc : 0xffffffff

******************result*********************
reg[2] (final return)value: 55
number of instructions: 253
number of R type instructions: 89
number of I type instructions: 153
number of J type instructions: 11
number of memory access instructions: 100
number of taken branches instructions: 10
*********************************************
```

<캡처4: inputfile-simple4.bin 결과 출력>

```
163634473 ------------------------------------------------------------------------
163634474 [Fetch] 0x03e00008
163634475 [Decode] opcode(0) rs: 31 rt: 0 rd: 0 func(0x8) simm(8) <jr>
163634476 pc : 0x18ef0
163634477 pc = reg[31]
163634478 pc : 0x18ef0 ----> pc : 0xffffffff
163634479
163634480 ******************result*********************
163634481 reg[2] (final return)value: 85
163634482 number of instructions: 23372706
163634483 number of R type instructions: 10152862
163634484 number of I type instructions: 13219741
163634485 number of J type instructions: 103
163634486 number of memory access instructions: 7116606
163634487 number of taken branches instructions: 2029699
163634488 ****************************************
```

<캡처5: inputfile-input4.bin 결과 출력(실행파일 이용)>

```
------------------------------------------------------------------------
[Fetch] 0x03e00008
[Decode] opcode(0) rs: 31 rt: 0 rd: 0 func(0x8) simm(8) <jr>
pc : 0x38
pc = reg[31]
pc : 0x38 ----> pc : 0xffffffff

******************result*********************
reg[2] (final return)value: 55
number of instructions: 2788
number of R type instructions: 927
number of I type instructions: 1697
number of J type instructions: 164
number of memory access instructions: 1095
number of taken branches instructions: 109
****************************************
```

<캡처6: inputfile-fib.bin 결과 출력>

```
--------------------------------------------------------------------
[Fetch] 0x03e00008
[Decode] opcode(0) rs: 31 rt: 0 rd: 0 func(0x8) simm(8) <jr>
pc : 0x44
pc = reg[31]
pc : 0x44 ----> pc : 0xffffffff

******************result********************
reg[2] (final return)value: 1
number of instructions: 1098
number of R type instructions: 396
number of I type instructions: 637
number of J type instructions: 65
number of memory access instructions: 486
number of taken branches instructions: 73
****************************************
```

<캡처7: inputfile - gcd.bin 결과 출력>

```
 1 00000000: 27bd ffd8 afbf 0024 afbe 0020 03a0 f021  '......$... ...!
 2 00000010: 2402 000a afc2 0018 8fc4 0018 21bc 0010  $...........!...
 3 00000020: 0380 0009 afc2 001c 03c0 e821 8fbf 0024  ...........!...$
 4 00000030: 8fbe 0020 27bd 0028 03e0 0008 0000 0000  ... '..(........
 5 00000040: 27bd ffd0 afbf 002c afbe 0028 afb0 0024  '......,...(...$
 6 00000050: 03a0 f021 afc4 0030 8fc2 0030 0000 0000  ...!...0...0....
 7 00000060: 2842 0003 1040 0004 0000 0000 2402 0001  (B...@......$...
 8 00000070: 0800 002e 0000 0000 8fc2 0030 0000 0000  ...........0....
 9 00000080: 2442 ffff 0040 2021 0380 0009 0000 0000  $B...@ !........
10 00000090: 0040 8021 8fc2 0030 0000 0000 2442 fffe  .@.!...0....$B..
11 000000a0: 0040 2021 0380 0009 0000 0000 0202 1021  .@ !...........!
12 000000b0: afc2 0018 8fc2 0018 03c0 e821 8fbf 002c  ...........!...,
13 000000c0: 8fbe 0028 8fb0 0024 27bd 0030 03e0 0008  ...(...$'..0....
14 000000d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
15 000000e0: 0a                                       .
```

<캡처8: 추가구현 jalr – fib2.bin>

```
--------------------------------------------------------------------
[Fetch] 0x03e00008
[Decode] opcode(0) rs: 31 rt: 0 rd: 0 func(0x8) simm(8) <jr>
pc : 0x38
pc = reg[31]
pc : 0x38 ----> pc : 0xffffffff

******************result********************
reg[2] (final return)value: 55
number of instructions: 2788
number of R type instructions: 926
number of I type instructions: 1698
number of J type instructions: 164
number of memory access instructions: 1095
number of taken branches instructions: 109
****************************************
```

<캡처9: 추가구현 jalr - fib2.bin 결과 출력>

VIII. Personal feelings

저번 프로젝트1을 할 때에는 txt 파일에 직접 MIPS 코드를 작성해서 코드를 읽어와서 프로그램을 실행했는데 이번 프로젝트2에서는 진짜 프로그램 코드 bin 파일을 통해 16진수로 된 명령어를 읽어 들이는 등 좀 더 컴퓨터가 명령어를 수행하는 과정에 대해 깊이 있게 배울 수 있는 기회였던 것 같다. 싱글 사이클 프로그램을 만드는 것이 이번 과제의 목표였는데 각 명령어들이 하는 역할, 연산을 컴퓨터 구조 이론 수업에서 배웠지만 pc 값에 주소가 들어가는 형식, jump 할 때 target이 되는 구체적인 주소 값들 등 정확하게 기억은 하지 못하고 어느 정도만 이해를 하던 중이었기 때문에 과제를 하면서 이론 공부도 하고 코드를 구현하면서 single cycle 개념에 대한 이해를 높일 수 있었다.

그저 코딩 문제를 풀거나 코드를 구현하는 것이 아닌 배운 이론 개념을 가지고 코드 구현을 하는 것은 이번 학기가 처음이고 쉽지 않지만 나중에 많은 도움이 될 것 같다고 생각했다. 다른 전공과목에서도 프로젝트 과제가 있는데 이번에 너무 기간이 겹쳐서 힘들었지만 single cycle simulator 과제 기간이 길어서 정말 다행이었다. 이론 공부도 해야 하는데 코드를 구현하는 과제를 많이 하다 보니 코딩 하는게 더 재밌다는 생각이 들었다.

3학년 1학기 때가 가장 바쁜 시기 일 것 같고 어려운 것도 구현하는 등 많은 것들을 배우는 시기가 될 것 같다. 이번학기 무사히 수업 잘 듣고 건강히 마칠 수 있으면 좋겠다.