CSC3050_ASS2_Zhouliang_Yu_120040077

Overview:

In this project we first use Phase1 and Phase2 to assemble each instructions in the .asm file to get the LabelTable, and write the assembled result of the instructions in file with the name of arg[2].

Then we store each instruction and its valid line number in a map to assistant our future works on simulating each instructions. Next, we put the information in .data and .text region in the allocated memory. In the end, we simulate each instructions.

How Do We Map the Virtual Memory to the Real Memory

We have a pointer named "real_mem" storing the real address of the block of memory allocated so the value of "real_mem" is mapping to 400000_hex. Then we can do a 1-to-1 mapping relationship to the simulated address. So For a given simulated address let's say sim_addr, the real address mapping to it would be

```
real_addr = real_mem + sim_addr - 0x400000
```

Then for each line components in the map, we first classify the data type, there are 5 data types. ascii, asciiz, word, half, and byte

byte: 8 bits

half: 2 bytes

word: 4 bytes

ascii: 8 bits

asciiz: 8 bits with one "\0" to end with

How to extract valid data

The data line in .asm file look like this:

```
str1: .asciiz "Testing lb,sb,read/print_char, etc\n"        #at 0x00500000
str2: .asciiz "Please enter a char:\n"              #at 0x00500024
str3: .asciiz "The char you entered is:"            #at 0x0050003C
str4: .asciiz "\nTests for .ascii\n"                #at 0x00500058
str5: .ascii "aaa\n"                          #at 0x0050006C
str6: .ascii "bbbbbbb\n"                      #at 0x00500070
str7: .asciiz "ccc\n"                         #at 0x00500078
```

The Problem is How to extract the information from the asm file so that we can use a vector to load the line information like

```
vector<string> result;
result[0] = "str1:"
result[1] = ".asciiz"
result[3] = "Testing lb,sb,read/print_char, etc\n"
```

and for half and byte data line

How to Put the Data

We first scan the .data region in .asm file, then we use the following map to store the information of valid data line(not a blank, not a comment), we take valid data line number as the key and a vector of string to contain each component in the line as value.

```
map<int, vector<string>> data_map;
```

How to Put the text

Say we have translate the instruction into binary form machine code in phase1 and phase2. The next task in our project is to put those binary form string into 32-bits binary number and put them into the text region in the memory.

Then we came across a problem how to transfer a binary-like string into a 32 bits representation. we find that reinterpret_cast and bitset can perfectly solve this problem.

So the work flow of this process is: 1. we use getline to extract a machine code binary string from the .asm file. 2. we transform the string line into a int32_t integer number with the help of bitset. 3. we use memMap to map to the virtual memory, and put the 32-bits integer in the real memory.

How to do Branch

In operations i.e. beq, bgez. we need to deal with the labels. So we remember in the Proj1, We record the Label's name and its corresponding addresses in the LabelTabel, and the last last 16-bits of the relevant machine code is the offset.

So for operations like beq, bgez, we extract the last 16-bits as offset, then we do the corresponding operations.

How to Load and Store Data

For load operation like lb:

```
lb  $t0, 1($s3)
```

This loads a byte from a location in memory into the register $t0. The memory address is given by 1($s3), which means the address $s3+1. This would be the 0+1=1st byte in memory.

So to load the data we first do sign extension (or zero extension for unsigned operations). Secondly, we do reinterpret_cast to cast the data to the required int types. Finally we store the data into the registers.

Sign Extension and Zero Extension

- An integer register on the MIPS is 32 bits. When a value is loaded from memory with fewer than 32 bits, the remaining bits must be assigned.
- Sign extension is used for signed loads of bytes (8 bits using the lb instruction) and halfwords (16 bits using the lh instruction). Sign extension replicates the most significant bit loaded into

the remaining bits.
- Zero extension is used for unsigned loads of bytes (lbu) and halfwords (lhu). Zeroes are filled in the remaining bits.

How to implement lwl, lwr

To discuss the implementation of lwl and lwr, we take lwl for an example:

This instruction reads the left portion of a register (hign-order bytes). Conceptually, LWL starts at the specified byte in memory and loads that byte into the high-order (left most) byte of the destination register. Then, it proceeds to the low-order byte of the word in memory. And the low-order byte of the register, loading bytes from memory into the register until it reaches the low-order byte of the memory word. So, start at some specified address, and start copying bytes from memory and **increment** the memory byte address until you reach the next highest memory address that is word aligned, not including the byte at this last word aligned address. The least significant (right-most) byte(s) of the register will not be changed. Consider the following instruction, memory contents for two words, and register 4's values before and after executing a LWL instruction.

For instruction below:

```
lwl $4, 2($0)
```

the memory and register is like

```
            Memory                              Register 4
            byte 0, byte 1, byte 2, byte 3      byte 0, byte 1, byte 2, byte 3
address 4:    4       5       6       7           A       B       C       D
before
address 0:    0       1       2       3           2       3       C       D
after
```

So in my implementation:

we first calculate the target address, which is:

```
int32_t tar_adadr = reg_mem[rt] + offset_32;
```

then we can get the word from the memory,

finally, we check the lowest two bits of rs register to decide the bytes in memory should be taken.

How to implement swl and swr

To discuss the implementation of swl and swr, we take swl for an example:

This instruction stores the left portion of a register (high-order bytes). Conceptually, SWL starts at the specified byte in memory and stores the high-order (left most) byte of the source register to that memory location. Then, it proceeds to the low-order byte of the word in memory and the low-order byte of the register, storing bytes from the register into memory until it reaches the low-order byte of the memory word. So, start at some specified address, and start copying bytes from the register to memory and **increment** the memory byte address until you reach the next highest memory address that is word aligned, not including the byte at this last address. The most

significant (left-most) byte(s) of the memory location will not be changed. Consider the following instruction, memory contents for two words, and register 4's values before and after executing a SWL instruction.

For instruction:

```
swl $4, 5($0)
```

the memory and register is like

```
            Memory              Register 4
            byte 0, byte 1, byte 2, byte 3  byte 0, byte 1, byte 2, byte 3
address 4:    4       5       6       7       A       B       C   D     before
address 0:    0       1       2       3       A       B       C   D     before

address 4:    4       A       B       C       A       B       C   D     after
address 0:    0       1       2       3       A       B       C   D     after
```

Notice, that we are starting at address 0+5, that's byte 1 of word at memory address 4. The byte at that memory location (value 5) is replaced by the left-most (most significant) byte of the source register. Bytes are copied toward the right until the the last byte of this memory word is reached

So in my implementation:

Elaboration of Selected Syscall

sbrk

we store the address of the start of dynamic data, then we set the offset address which is the content stored in v0. In the end, we update the address of where dynamic data start by:

```
dynamic_data_start += offset;
```

exit

We simply use the system call with parameter set to be 0.

read_char

in our project the way we open the file is through the file stream. So to read the char, we simply use ifstream.get()

open

the file descripter in stored in the reg_mem[4], the flag is stored in reg_mem[5], and the mode is stored in reg_mem[6]. so to open a particular file we just call open() in system call by the above arguments.

How to Test the Project

The project strongly follows the piplines of project guide and checkpoints.
so the command for running my program is by:

```
./simulator test.asm test.txt test_checkpoints.txt test.in test.out
```

Some Releted Referance

[1] bitset: https://www.cnblogs.com/magisk/p/8809922.html

[2] https://www.cplusplus.com/reference/bitset/bitset/

[3] reinterpret_cast https://www.youtube.com/watch?v=wcKXGvi1JRk

[4] lb: https://www.youtube.com/watch?v=p_cv0npmGok

[5] binary operater: (24条消息) C++位运算符（&、｜、^）_Wonder-King的博客-CSDN博客c++^

[6] zero extention & sign extention: https://www.youtube.com/watch?v=UshbQEn2H-k

[7] lwl, lwr https://stackoverflow.com/questions/57522055/what-do-the-mips-load-word-left-lwl-and-load-word-right-lwr-instructions-do

[8] fstream: https://stackoverflow.com/questions/7877553/very-surprising-perfs-of-fprintf-vs-stdofstream-fprintf-is-very-slow#:~:text=There%20is%20a%20file%20buffer,()%20to%20have%20a%20test.

[9] lwl, lwr, swl, swr: https://courses.cs.duke.edu//fall02/cps104/homework/lwswlr.html