Project 1 report

作者

周延儒

computer arhchitecture

[教師姓名]

1: Project Description:

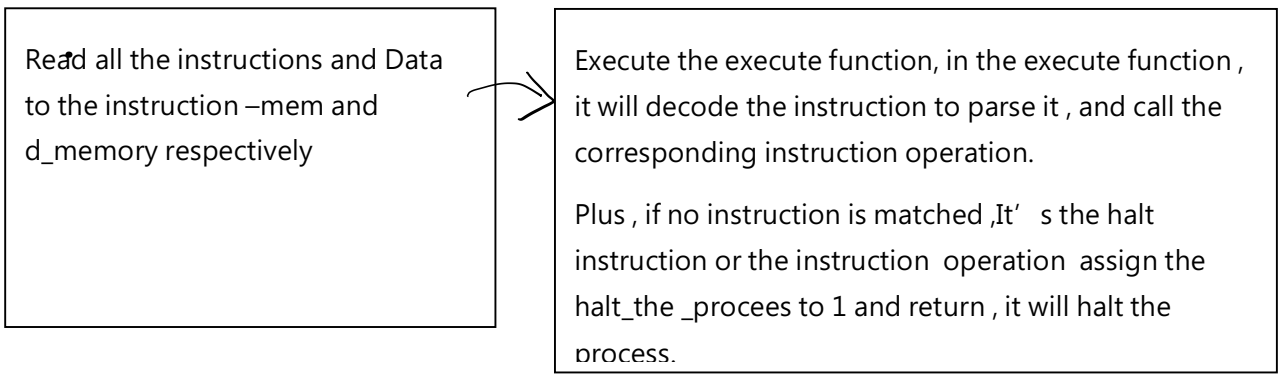
In this project , we are required to make a simulator to simulate a processing unit with single cycle. The designs are as follows:

a. program flow

To achieve the entire design,we have to analyze the simulator in the very beginning. After careful contemplate, we can divide them into several parts. Main.c instruction memory , data_memory (D-memory), instruction operation, reg_file(register_file), printreg, and executive.

Additionally , we declare extract variable , and they are sp,pc,and halt_the_process.

For an instruction in the instruction-memory is executed. It can perceived as these flow:



In the instruction operation, depending on the instruction , it will operate the proper arithmetic operation or load or store data, in the d_memory and reg_file with rs rt td c_shame,and c_immediate, Moreover, if it detect any error in each operation , such as write 0 error , number overflow , address overflow, memory misalignment ,it can also record them and determine whether to assign the halt_the_instruction to 1 and return . Then execute function will halt the process.

After operation instruction in instruction operation . The executive will call printreg() function to print the register-file and error to Snapshot.rpt

In the printreg() function, it will print all the 32 registers in hexadecimal form in the regfile.

In the end, close FILE pointer and accomplish the single_cycle CPU.

After program flow , let' s explore all the details. First of all, we declare 3 structures we need . They are reg, instruction_mem_element, data_mem.

- Reg is aimed to store the value of the 32 registers in the regfile . In the reg, we store the value in character form as well as the interger , (char alt_name, int val).Plus, we declare then as reg_file[32].
- Instruct_mem_element is for the purpose of the data of each instruction. Inside this struct , we store the opcode , rs, rt,rd,c_shame,funct, and C_immediate in character array and interger form . Besides, depending on the opcode , we store additional c_immediate_signed to help us to operate properly.

  In the end, we store instruction in string to help debug,and

  There are 256 instruction memories ,for the memory unit we design is 1 word .
- Data_mem is to store the data in the D_memory . Of course we store the value in char array and integer form. Still , we store 1 word in each memory , so the size of the entire memories is 256 .

Next , we start to explain the instruction operation. As far as we know, there are 3 types of instructions,R ,I, and J type.

The execution function will decode the instructions and operate them depending on the op code ,rs,rt ,rd, immediate c_shame, and the function code. Besides, if there is no opcode and function matched, we think of it as "no instruction matched" and skip this numateched instruction . Althought in this case, it is impossible or it is the illegal testcase, we have to prevent this sort of hazard.

- R type. The common factor is their opcode are exactly the same , so we analyze them in terms of funct . Something we have to notice is overflow detection of add ,and sub. The overflow detection of add is (a>0 && b>0 && (a+b)<=0) ll (a<0 && b<0 &&(a+b)>=0), where a, b are the same signs and however the result is on the opposite.

  Then , sub overflow detection is (a<0==(-b<0))&&(a<0!=(a-b)<0) is the same idea .

- I type. The common factor is they have the immediate value

  So we analyze the op,rs,rt,c_immediate(unsigned or signed), and operate the instruction. Still , there exists some hazards, the address may overflow . For lw,sw , the last memory position is 1021 , sb,lb the one is 1024, and lh sh the one 1022 .

  Notice that because we store the data in per unit of word, to access the half word or bytes ,we have to take particular approach. Here we take lb and sb for instance , the sb lb is the same method to some degree.

  For lb , we the value the rs ,and the immediate as the offset. And calculate the address . If it doesn't exceed the memory address, we shift the the address for 2 , that is , it it divide by 4. Then we can get word memory the bytes, which we desire, belongs to .

  Because the word is composed of 4 Bytes, we have to know the which bytes we want . Fortunately , we can find it with modul0 operator . We can calculate the value of (rs + immediate)%4 ,for the word is 4 Bytes. Once we know it , we can load the word value and use shift operation to "trim" the word value as we want. For example if (immediate+reg_file[rs].val)%4==2) , we know we need to

loard the third bytes of the word. As a result , the value is val=val<<16; then val=val>>24; As for the sb, the way is very similar . we just need to change it a little bit. Take the previous example ,

We have to trim the data in the D-memory first , data should be data=data & 0xffff00ff; insert_byte=insert_byte<<8; for we want to store them in the third place . Ultimately, add them to store them back in the D_meory . Something interesting is that , for beq, bne , bgtz, we don't have multiply immediate by 4 , for we store the instruction per unit of exact one word.

- J type: There are actually 2 columns we need parse, opcode and address. The pc value will "jump" to the corresponding address.

Next, in each of the instruction operation , we must detect all the errors . In fact, in this project , there are 4 types of error, `Write $0 Error, Number Overflow, Address Overflow, Misalignment Error respectively.` Notice that after the address overflow and misalignment errot occurs , we must halt the simulator, As a conqequenc, we declare a variable called halt_the_instruction. Whenever the variable is assigned to be 1 , the simulator will halt after the operation instead of immediately for we have to detect all the error in each instruction. So these are the the way we detect the errors.

- Write 0 error: if we detect any instruction operation which will assign value to the zero register , we will fprint the write 0 error except the operation is NOP 0x00000000.
- Number overflow: if we detect any instruction operation which involves the arithmetic operation such as add or sub, we must know whether the result will overflow or not. If it overflow , we still frprint the error. The way we detect is similar to the way we do

with the instruction add and sub . Fortunately , the way we detect overflow require the add overflow detection the most . And it is the easier way to implement .

- Address overflow : in my opinion , this part may be the most difficult on , compare with the rest . To determine whether the operation will cause address overflow, we have to deal with it separately . First of all , we know this kind of error happens only if we load or store data in the D-memory . So , we have to care about lw lh lhu lb lbu and the inverse of them . For the loading or storing word, the address boundary is between 0 to 1020 . so if the rs value + immediate is over this range, we print the address overflow and assign the halt_the_instruction to be 1 .
  Similarly , for half word instruction, the range is between 0 to 1022 , and for the byte instruction is between 0 to 1023.
- Misalignment Error: continue the loading and storing word instruction . there is one more thing we have to pay attention to : Will the location be the very right location we can access ?
  For the word instruction , we can only access the D-memory whose address is the multiple of 4 . That is , if the rs_value +immediate is multiple of 4 , we can access it to read or write. Otherwise, we can not access it and we should print the Misalignment Error and make the halt_the_instruction to 1.
  Likewise , the half word, the address should be multiple of 2 . And for the byte , there is not misalignment error unless the address is not a interger which is ridiculous .

After detection of the all the error , if the instruction is correct , operate the instruction , Othewise , skip the operation and return, The execution function will judge whether to halt the instruction or not .

Then in the end of the execution function , it will call the printreg() function to fprint the register file data to the snapshot.rpt. There is something that we have to be cautious about , for the 0 to 31 registers and pc , the value is exactly we want except the pc value . As what we have mentioned above , the value of pc is word dependent . So the pc value ranges from 0 to 255 . Compared with the really pc value which is byte dependent, we have take a appropriate approach . Luckily, we just have to shift the pc value left by 2 . Moreover, because , pc value has something to do with I-memory , there is supposed to be no address misalignment or address overflow , or it will be illegal input file . As a result , this method ought to be correct.

After all the instruction is executed properly , the single_cycle processor has done the duty . Then ,we fclose the the file pointer and generate the very correct output file if there is no bug.

Next part is the case design :

To design the test case , we need to design testcase with all the possible cases. In the first submission , I didn' design the extreme testcase. It' s in the seconde submission that I did design the extreme testcase. In this report , we need to explain the testcase in term of the first submission regardless of the testcase after the first submission .

This is the assembly I design

```
addi $4, $1, 400
addi $29, $29, -12
add  $4, $5 , $3
addi $8, $0,32767
addi $0, $1 , 6
sw   $6, 200($0)
sw   $6, 20($5)
sra  $4,$1,5
srl  $5, $25, 2
srl  $0, $4,20
xor  $1 ,$5 ,$20

halt
```

In the very beginning, I design the testcase manually . As a result , it' s not productive to generate the testcases. In submission , I tried to design the cases that it wasn't involved in the TA testcase instruction .

Makefile:

In addition to the single_cycle we have to design our own makefile.

```
TARGET=make

$(TARGET):  main.o
    gcc simulator.o -lm  -o single_cycle


main.o:
    gcc -c  simulator.c -lm

clean:
```

```
rm -f *.o single_cycle
```

In designing the makfile , we encounter something weird with the gcc ,the pow function in <math.h> will cause conflict , so we have to add the parameter –lm so solve the conflict .
In this makefile , it will generate the single_cycle excutable file .

Reviews:

In this project , we try to implement the single_cyle CPU. To implement the design , we have to analyze the structure of the CPU. In my opinion , this could be the most difficult part, for structure have a to do with the following development . To design to the CPU is extremely difficult. It took me almost 70 hours to develope it . In the beginning, it was quite frustrating for there are numerous bugs and error. However , after finishing the simulator , I felt a sense of achievement . Although there is something wrong with my first submission because there is external file with the tar file so I got 0 with the submission , I hope I get the full scores in the second submission .
Stiil , hope we can overcome the all the difficulties in the following days!!