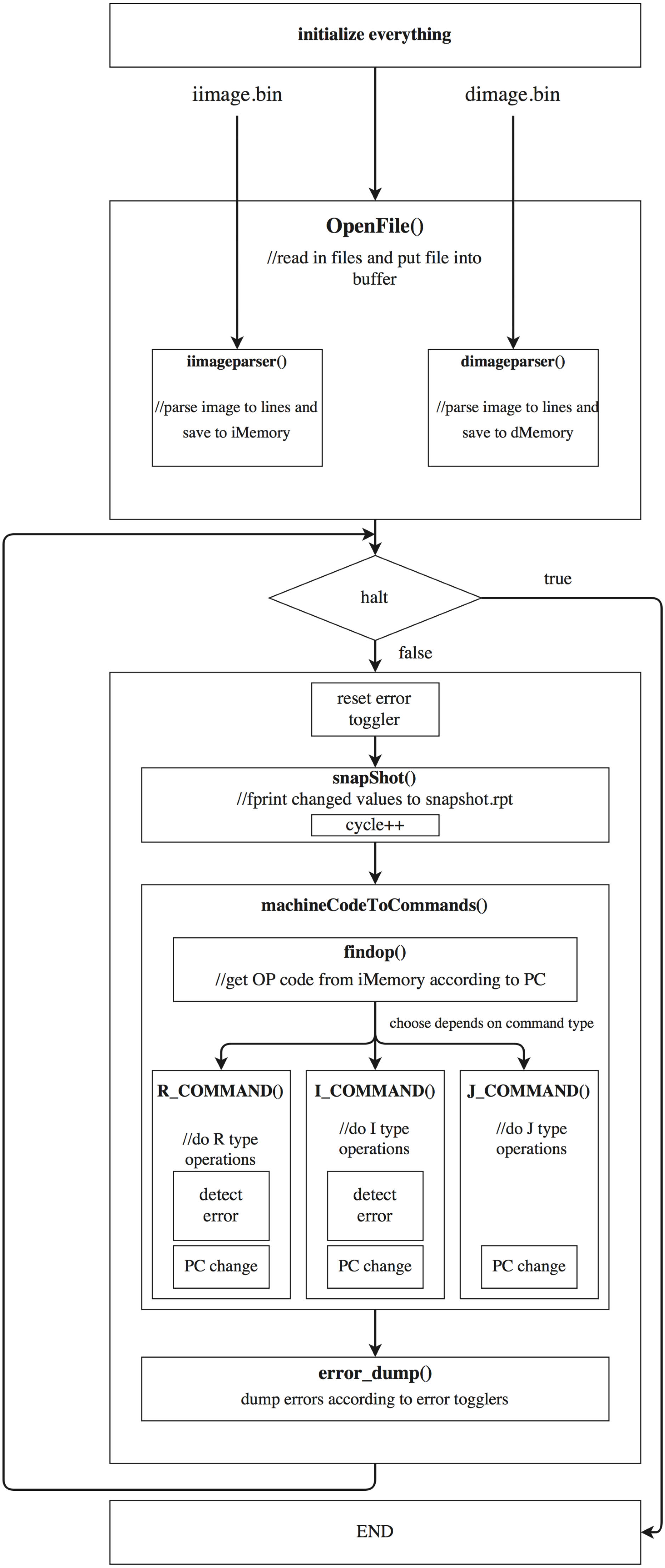
計算機結構
Computer Architecture

PROJECT ONE REPORT

104062206

蔡鎮宇

Content


initialize everything
iimage.bin
dimage.bin
OpenFile()
//read in files and put file into buffer
iimageparser()
//parse image to lines and save to iMemory
dimageparser()
//parse image to lines and save to dMemory
halt
true
false
reset error toggler
snapShot()
//fprint changed values to snapshot.rpt
cycle++
machineCodeToCommands()
findop()
//get OP code from iMemory according to PC
choose depends on command type
R_COMMAND()
//do R type operations
detect error
PC change
I_COMMAND()
//do I type operations
detect error
PC change
J_COMMAND()
//do J type operations
PC change
error_dump()
dump errors according to error togglers
END

Detail Description

- **iimageparser() and dimageparser() :**

  According to iimage.bin , we can initial PC value and fetch instructions , storing them into **iMemory[ ]** . Note that the fetched instructions haven't been converted into human-readable instructions (such as op,st,st…etc) . We'll do it in other stage.

  According to dimage.bin , we can initial $sp and fetch data , storing them into **dMemory [ ]** .

- **The main process is a simple while loop which only stops when halt is true , indicating all instructions are done or error happens.**

- **snapShot() :**

  This function fprintf() all the values at the first time when **cycle==0** . At other cycles it puts only values that are different from last cycle . We made this happen by setting up temporary registers that store last cycle's values , and only outputs when it differs from the value this cycle . Last but not least, we make **cycle++** here since it's the first function in the loop.

- **machineCodeToCommands :**

  First , we find the instruction we'd like to execute from iMemory based on **PC** .

  Then , we use **findOP**() to get the OP code and call the corresponding function that does the instruction . A simple switch statement favors us. If **OP is 0** (**R type**) , we need to use **findFUNC**() to know the function to perform.

  findOP() : OP = iMemory[PC] right shift 2 bits (unsigned).
  findFUNC() : func = iMemory[PC+3]'s last six bits;

- **R_COMMAND() :**

  In this function , we implement R type commands . But first we call **findRSRTRD() ,** **findSHAMT**() to find all the codes we need .

  findRSRTRD() : RS = iMemory[PC]'s last two bits | iMemory[PC+1]'s first three bits . RT = iMemory[PC+1]'s last five bits . RD = iMemory[PC+2]'s first five bits.

  findSHMAT() : shame = iMemory[PC+2]'s last three bits | iMemory[PC+3]'s first two bits.

After this , it's easy to perform the designated instruction . Since most instructions are intuitive and simple to implement , here I only list things that worth mentioning.

In add($d,$s,$t) : the OVERFLOW detection is needed . A detectOverflow(a,b,c) function is called (the order of a,b,c means a+b=c). It detects overflow by looking at the sign bit of a,b,c . If a's and b's are the same and c's is different , that's a OVERFLOW. We then toggle numOverflow on. What's tricky is in C language , a int means signed int and all values stored in register is unsigned , so we need to be careful while manipulating.

In sub($d,$s,$t) : the OVERFLOW detection is needed . We call the detectOverflow(a,b,c) as we do in add($d,$s,$t) . The only difference is we should take b = 2's complement of $t since 2's complement means negative of a value . That means we treat it as same as a+(-b) = c.

In jr($s) : directly set PC as the $s value . Need to return instantly since we'll let PC+=4 at the end of each instruction .

In mult(),multu(),mfhi(),mflo() : multiply two 32 bits value and obtain a 64 bits value , then store the higher 32 bits in HI , lower 32 bits in LO . The tip is that we need a 64 bits (long long) thingy to hold the value first , then separate them into HI and LO . mfhi() and mflo() are commands that copy HI or LO into a destined reg . Note a WriteHILO detection should be performed as well.
WriteHILO detection : toggle on HILO when mflo(),mfhi() . If mult(),multu(),then consider if already toggled on MULT . If it's on already , we should turn HILO off in case of duplication . If a mult() or multu() is performed when HILO isn't on , then it's a HILOoverwrite.

- ## I_COMMAND() :

In this function , we implement R type commands . Note we also perform **findRSRTRD()** but we don't call **findIMMEDIATE()** here since some instructions use signed immediate while some use unsigned immediate . After this , it's easy to perform the designated instruction . Since most instructions are intuitive and simple to implement , here I only list things that worth mentioning.

Note : When finding immediate , if we need a signed one we should do sign extension .

In lw,lh,lhu,lb,lbu : the memOverflow and dataMisaligned are needed . When loading data from memory , signed extension is needed for lw , lh , lb .

memOverflow detection : if the data address you ought to load is over 1K , it's memovf.

dataMisaligned detection : for different size of data to load , if the start address isn't size*n for n = 0,1,2,3… , then it's misaligned.

In sw,sh,sb : the memOverflow and dataMisaligned are needed too.

In beq,bne,bgtz : change PC based on the condition , note PC should +4 before + the relative address and return right after .

- ## J_COMMAND() :
  In this function , we implement J type commands . **j() and jal()** are basically the same just notice we should **change $31 to PC+4** . Note : the jump destination address is absolute compared to relative one of branch instructions.
- ## writeRegZero :
  Every instruction involving writing to $0 is forbidden , thus we need to consider the case in those instructions . Note : sll $0,$0,$0 is not the case though , because it means NOP.

- ## error_dump() :
  Dump errors when the corresponding toggle has been turned on after this cycle . Note the **cycle is still the same** while PC has been changed . As mentioned above , we change cycle value in next snapShot() call . If dataMisaligned or memOverflow is on,halt will be changed to true , indicating the whole process has been terminated .

Test Data Description

Most of my test cases are designed to test out error handling such as number overflow , write to zero . The test process halts due to data misaligned and memory overflow , testing if it halts on these two errors correctly.

dMemory arrangement

0x00 : 0x7FFFFFFF (MAX of 32 bit signed int = 2^31 -1)
0x04 : 0x00F0F0F0 (no special meaning)
0x08 : 0x80000000 (MIN of 32 bit signed int = -2^31)
0x0c : 0x00000210 (another no special meaning)

Overflow Detection

```
|#test largest positive - second smallest negative OVF
|	lw $1,0x0000($0)
|	lw $2,0x0010($0)
|	sub $1,$1,$2
|#test addi largest ovf
|	lw $1,0x0000($0)
|	addi $1,$1,0x7FFF
|#test addi smallest ovf
|	lw $1,0x0008($0)
|	addi $1,$1,0x8000
|#test largest positive sub smallest negative overflow
|	lw $1,0x0000($0)
|	lw $2,0x0008($0)
|	sub $1,$1,$2
|#test largest positive sub overflow
|	lw $1,0x0000($0)
|	lw $2,0x0000($0)
|	sub $1,$1,$2
|#test smallest negative sub overflow
|	lw $1,0x0008($0)
|	lw $2,0x0008($0)
|	sub $1,$1,$2
|#test smallest negative add overflow
|	lw $1,0x0008($0)
|	lw $2,0x0008($0)
|	add $1,$1,$2
|#test biggest positive add overflow
|	lw $1,0($0)
|	lw $2,0($0)
|	add $1,$1,$2
|#test smallest negative sub largest positive overflow
|	lui $1, 0x8000
|	sub $2,$1,$2
|#test addi overflow
|	addi $2,$1,0x8000
```

Test Load/Save of dMemory

```
#test load half word
    lh $3,4($0)
    lhu $3,4($0)
#test load byte
    lb $3,3($0)
    lbu $3,3($0)
#test save word
    lw $4,0($0)
    sw $0,0xff01($3)
    lw $4,0($0)
#test save halfword
    sh $1,0xff01($3)
    lw $4,0($0)
#test save byte
    sb $0,0xff02($3)
    lw $4,0($0)
```

Test some signed/unsigned operations

```
#test shift right arithmetic
    sra $1,$1,31
#test signed comparison
    srl $2,$2,1
    slt $3,$1,$2
#test unsigned I-command
    andi $2,$1,0xffff
    ori $2,$2,0xffff
    nori $2,$2,0xffff
#test signed comparison
    slti $2,$1,0xffff
    slti $2,$1,0x8000
    slti $2,$0,0xffff
    slti $2,$0,0x1fff
```

Test every write to $0

```
|#test every writeto0
|        add $0,$0,$0
|        addu $0,$0,$0
|        sub $0,$0,$0
|        and $0,$0,$0
|        or $0,$0,$0
|        xor $0,$0,$0
|        nor $0,$0,$0
|        nand $0,$0,$0
|        slt $0,$0,$0
|        srl $0,$0,0X0000
|        sra $0,$0,0X0000
|        mult $1,$1
|        multu $1,$1
|        mfhi $0
|#test only no write $0 error
|        sll $0,$0,0
|        sll $0,$1,0
|        mult $1,$1
|        addi  $0,$1,0xffff
|        addiu  $0,$1,0xffff
|        lw   $0,0x0000($0)
|        lh    $0,0x0000($0)
|        lhu   $0,0x0000($0)
|        lb    $0,0x0000($0)
|        lbu  $0,0x0000($0)
|        lui   $0,0x0000
|        andi   $0,$1,0xffff
|        ori    $0,$1,0xffff
|        nori   $0,$1,0xffff
|        slti    $0,$1,0xffff
```

Test branch and jump

```
|   #jump to boundary then jump back to execute next instruction
|---jal 0x100                    #jump to PC=0x3fc          #pc=0x200
|   lw $4,0x0010($0)             #load 0x0210               #pc=0x204
|   jr $4                        #jump to 0x210             #pc=0x208
|   halt                                                    #pc=0x20C
|   halt                                                    #pc=0x210
| ..... TEST EVERY WRITE TO ZERO HAPPENS HERE!!!
|#test branch
|   addi $2,$0,0x0000                                       #pc=0x280
||<-beq  $2,$0,0x0001                                       #pc=0x284
||        halt                                              #pc=0x288
||<->bne  $1,$0,0x0001                                      #pc=0x28C
||        halt                                              #pc=0x290
||->bgtz $1,0xFFFF                                          #pc=0x294
||<-bgtz $4,0X0009                                          #pc=0x298
||        halt                                              #pc=0x29C
|||>lw $4,0x8($0)                                           #pc=0x2A0
|||       lh $3,0x(-1)($4) #####test misalign,numovf,addovf  END#####  #pc=0x2A4
|||       halt                                              #pc=0x2A8
|||       halt                                              #pc=0x2AC
||<->bgtz $4,0Xfffb                                         #pc=0x2B0
|||       halt                                              #pc=0x2B4
||<->bne $1,$0,0xfffd                                       #pc=0x2B8
|||       halt                                              #pc=0x2BC
||<->beq $2,$0,0xfffd                                       #pc=0x2C0
|
.
.
.
|
|------ j 0x200                     jump to PC=0x200        #pc=0x3FC
```

NOTE : YELLOW PARTS ARE TESTING BRANCHING BACK

REPORT ENDS HERE~