Y86-Simulator Final Report

By 11300240001 蔡隆祺 rockyRocky
11300240007 陆盛谷 Lumig
11300240095 张雨薇 VV.

Catalog

1. TTY written by rockyRocky

1.1 Usage

cd $Y86_ROOT
python main.py

```
================================================
|                                              |
|        Y86-Simulator ics course pj 1.0.1     |
|        Copyright(c) 2013 CatMiaoMiaoMiao     |
|              Kernel version: 0.0.1           |
|                                              |
================================================
Error: missing input file
Usage: main.py [options] [y86 binary object]

Options:
  -h, --help            show this help message and exit
  -s, --second          use decoding rules in csapp 2nd edition. (default
                        using 1st editon rules)
  -m MAXCYCLE, --maxcycle=MAXCYCLE
                        set limit of running cycles. 0 means no limit.
                        (default is 32767)
  -g, --guimode         set log output syntax for GUI. (default is disabled)
  -n, --nologfile       print log to stdout instead of writing to file.
                        (default is disabled)
  -p, --print           print log on screen
  -l, --logfile=LOGFILE customize a new logfile. (default using input name
                        with .txt suffix)
  -d, --debug           begin debug mode (not done yet)
```

-g is not finished yet
-d is for simple debug, supports step, back, speed up and slow down.

For details please see help (enter 'help' or 'h').

```
================================================
|                                              |
|        Y86-Simulator ics course pj 1.0.1     |
|        Copyright(c) 2013 CatMiaoMiaoMiao     |
|              Kernel version: 0.0.1           |
|                                              |
================================================
Log file: y86-code/asum.txt
(ydb) h
[command]

Command:
    c, continue         run till the end/breakpoint
    s, step, n, next    step into next cycle
    b, back             back to last cycle
    h, help             show this help
    q, quit             quit
while running:
    p                   pause
    c                   continue
    +                   speed up
    -                   slow down
    q                   quit running mode

(ydb)
```

1.2 Implementation

- Python has a set of powerful option-parser functions, so it won't take much effort to master.
- Attend to run the simulator in the thread, or it could block the user interface. Python provides a simple thread class like:

```
thread.start_new_thread(simulator.run, ())
```

- To obtain the key knocked down, we need to consider different implementations for different platforms. (unix like/windows)

```
def getkey():
    # windows
    if (platform.system()=='Windows'):
        import msvcrt
        return msvcrt.getch()
    # unix like
    import termios
    term = open("/dev/tty", "r")
    fd = term.fileno()
    old = termios.tcgetattr(fd)
    new = termios.tcgetattr(fd)
    new[3] &= ~termios.ICANON & ~termios.ECHO
    termios.tcsetattr(fd, termios.TCSANOW, new)
    c = None
    try:
        c = os.read(fd, 1)
    finally:
        termios.tcsetattr(fd, termios.TCSAFLUSH, old)
        term.close()
    return c
```

2. GUI written by VV.

2.1 Overview

Despite the excellent command line interface, we still offer a friendly GUI interface to give our users a better experience. It will be easy for our users to get the expected information of register, program status, memory and cache, for special. For more details to install it and run it, refer to our readme file.

2.2 Design

The Graphic interface has an amazing hand-drawn background, which takes lots of efforts. It is a sketch drawing that presents all the information well. Sometimes it may be messy and unendurable for some person, especially when you run some extremely complicated programs. In such cases, you can consider to use our command line version. Our design servers to display more beautiful interface at the stand of art and gives the user a more comfortable enjoyment.

2.3 Realization Toolkit

Considering all the toolkits for python GUI designing, we choose PyQT4 to construct our Graphic interface due to its convenience and good compatibility to all platforms including windows, linux and MacOS. QT itself is such a powerful and widely applied library with an impeccable reference that can decrease our work to a great extent.

2.4 Functions

The GUI version supports both .yo and .ys files, which is different to the command line. Once a .ys file is imported, the y86 assembler will be called automatically to assemble the codes to binary ones that can be executed by our simulator. Once an invalid .ys file is loaded, the program will automatically call the GUI version of the assembler to help the user fix the problems. The basic functions of GUI are to be listed as following:

- display all the information you need including register, program status, cycles, source code, memory and cache
- support run, pause, step ,back and reset instructions
- change the frequency of the simulator dynamically at any time
- debug your program in our assembler
- a good handle of exceptions
- a special function to express our honor to D.Jin

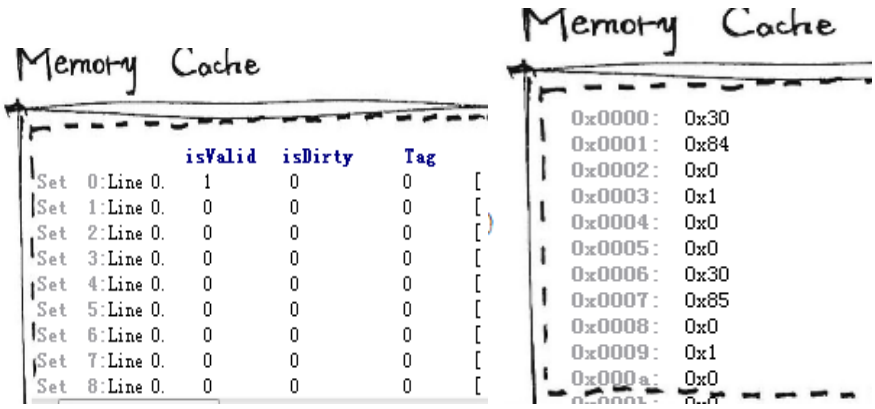
2.5 Technical implementation

Due to our Maximize Cohesion and Minimize Coupling designing schema, the GUI part is quiet separate from our simulator and assembler/disassembler, which is quiet a great advantage for use to implement all of our functions.

The GUI get all the details of every cycle and every stage by creating a Simulator class as its own. Then it calls function simulator.load() to convey the input file and output route to the simulator class and then calls simulator.run() to start the simulation. All the information it needs to present can be fetched in the private data of the simulator class. In GUI designing, we have no need at all to consider the details of the kernel design.

To compile the .ys input file, the GUI import assemble.py and simply use the assemble function. The assembly code are stored in an output file with a same prefix-name. Then our GUI import this output file and run it normally.

The highlight of our codes uses the QSyntaxHighLighter class in highlighter.py proposed by official docs of QT, which is quiet cool. The memory window only displays all changed memory address since the program runs. Our cache window displays all the cache.



We also use qss layout tools to help beautify our GUI.

3. Kernel written by rockyRocky

3.1 Structure

y86-simulator/kernel/

```
|----Sim.py          // status of simulator
|----Simulator.py    //  extends Sim; implements main functions
|----Memory.py       // manages the memory system
|----Cache.py        // under management of Memory,
|                    // and manages the Cache
|----History.py      // records history for stepping back
|----constants.py    // constants for y86 instruction set
|----utils.py        // useful binary converting functions
```

3.2 Usage

Only requires the class Simulator

3.2.1 new an instance

```
from kernel import *
simulator = Simulator()
```

3.2.2 load a .yo

```
simulator.load(fin, [fout])
```

If fout is missing, a default logfile would be created with the same name of input. Any input error would throw an IO exception.

3.2.3 step, back, run

```
simulator.step()
simulator.back()
simulator.run()
```

3.2.4 get status

Since python doesn't have public and private, I didn't write setters and getters. READ ONLY. Be CAREFUL.

```
// to highlight where the program is going
simulator.cycle          // get the current cycle
simulator.F_currentPC    // get the current PC of stage F
```

simulator.D_currentPC // get the current PC of stage D
simulator.E_currentPC // get the current PC of stage E
simulator.M_currentPC // get the current PC of stage M
simulator.W_currentPC // get the current PC of stage W
// to debug/display
simulator.getMemory(addr) // get memory at addr
simulator.getCache(si, li) // get cache at set si, line li.
simulator.isTerminated // .yo file finished execution or not
simulator.isGoing // whether simulator is in running mode
// others are the same as the figure at CSAPP2

3.2.5 set params

simulator.isSecond // default as 1st edition
simulator.setCacheParams(S, E, B, m) // set params for cache
// attend to set cache params BEFORE use,
// because doing so would reset the cache

3.3 Implementation

3.3.1 Simulator

- Class Simulator supports basic functions such as step, back and run.
- It recognizes yo-codes in both 1st and 2nd edtion of CSAPP.
- Below are listed the instructions supported:

nop, halt, rrmovl, irmovl, rmmovl, mrmovl, opl (i.e. addl, subl, andl, xorl), jxx (i.e. jmp, jle, jl, je, jne, jge, jg), call, ret, pushl, popl, iaddl, leave)

3.3.1.1 Load

// Attend to filter the instruction stream using Regular Expression

```
str_valid = re.sub(r'\/.*','',str_valid)
str_valid = str_valid.strip(' \n\r\t')
if str_valid == '':
    continue
```

// Separate addr and instr

```
tokens = str_valid.split(':')
if tokens[1] == '':
    continue
addr = int(tokens[0].strip(' '), 16)
content_str = tokens[1].strip(' ')
content_bytes = str_to_bytes(content_str)
self.memory.setBytesThrough( addr, content_bytes)
```

3.3.1.2 Step

/*
Each time it steps, first call the control logic to produce bubbles and stalls, second update the stage registers (the sequential logic), third update the combination logic and finally record the history.
*/

/*
Take care of the sequence of pipe-control, write, and stage. In write and stage, make sure to do in the reverse order (i.e. Write Back, Memory, Excecute, Decode, Fetch), considering we don't get all variables in intermediate registers (i.e. F, D, E, M, B)
*/

```
def step(self):
    if self.maxCycle != 0 and self.cycle > self.maxCycle:
        self.simLog('Reach Max Cycle')
        self.cpustat = 'HLT'
        return
    self.showStatTitle()
    self.pipeCtr()

    self.writeW() # take care of orders if you wanna save some ir
    self.writeM()
    self.writeE()
    self.writeD()
    self.writeF()

    self.stageW()
    self.stageM()
    self.stageE()
    self.stageD()
    self.stageF()
    self.showStat()
    self.history.record(self)
    self.cycle += 1

    if self.cpustat != 'AOK' and self.cpustat != 'BUB':
        self.isTerminated = True
        return False
    else:
        return True
```

/*
For details see the code. Simply translate the hcl at webaside, CSAPP, some parts referencing the source by Linus Yang.
*/

3.3.1.3 Back

```
def back(self):
    tmp = self.history.back(self.cycle-1)
    if tmp==False or tmp==None:
        self.simLog('out of history\'s reach')
        return
    else:
        self.isTerminated = False
        self.copy(tmp)
        self.showStatTitle()
        self.showStat()
        self.cycle += 1
```

3.3.1.4 run

// throw exception wherever it goes wrong

```
def run(self):
    try:
        while True:
            if self.isGoing:
                if not self.step():
                    break
                time.sleep(self.interval)
        self.logfile.close()
    except:
        self.simLog('Error: bad input binary file')
        self.logfile.close()
        raise
```

3.3.2 Memory

Main memory is stored as list (in python). Index is address and value is its value.

Each time try fetching data at cache first. If missing, handle cache miss (i.e. to fetch a new block) and try again.

Conflict policy is set a dirty flag, and only update the cache at conflict. When the line is to be replaced, update the memory.

Main functions:

handleCacheMiss(addr)
getByte(addr)
getWord(addr)
setByte(addr)
setWord(addr, word)
setByteThrough(addr, byte)// that is not to use cache

3.3.3 Cache

- Cache is stored as list and index is setIndex*entrySize+lineIndex
- Placement policy is addr mod E
- Main functions:

  getByte(addr)
  setByte(addr, byte)
  setBlock(addr, block)

3.3.4 History

- History is stored as list of Sim. Default size is 100.
- Since instance of History is owned by Simulator, however history takes down everything of simulator, that would cause a recursive referencing. To avoid this, I make a Sim out of Simulator, taking all the status, and make Simulator extend Sim. Finally record Sims in History.

3.3.5 Utils

utils.py contains some common but useful binary converting functions like lab1. Attend the function *bytes_to_int*. Since python doesn't has strict types, it would regard 0xffffffff as 4294967295 rather than -1. Hence, converting it manually is required at the end of function.

4. Assembler/Dessembler

written by Lumig

4.1 To Get Start

The assembling/disassembling tools follow the rules as described in CSAPP 1st editon and both have command line version and GUI version as well. More details are also offered in the readme file

```
E:\github\y86-simulator\compiler>assemble.py
Y86 compiler
Usage: inputfile [outputfile]
```

4.2 Assembler

The assembler supports all the y86 instruction in CSAPP book and some even more features as follows:

- All the y86 instructions even
- .pos, .long, .word, .byte instruction
- Labels for jump and call
- Jump to a direct address
- Support widely used comment format # , /**/ , //
- Retain all the original lines to help comprehension
- Show different error messages and the line position after assembling

We use regular expression, a recently widely applied technic to speed up the analysis as follows.

```
for reline in INFILE:
    origline.append(reline)    #save origin lines
    linepos += 1
    reline = re.sub(r'#.*$', '', reline)       #comment like     #...
    reline = re.sub(r'/\*.*\*/', '', reline)   #comment like     /*...*/
    reline = re.sub(r'//.*$', '', reline)      #comment like     //...
    reline = re.sub(r'\s*,\s*', ',', reline)   #explcit blanks
    if reline.find(':') != -1:
        lab = re.compile('([^\s]+):')
        labmatch = lab.search(reline)
        reline = lab.sub('', reline)
```

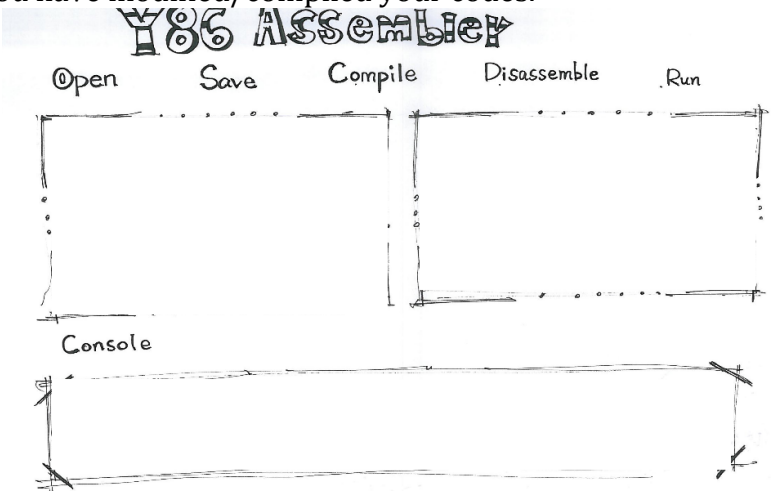
4.3 Dissembler

The disassembling tool does not support .long or .byte instruction. Every time it finds the address of an instruction not match its length, the .pos instruction will be applied to fix it. In the GUI mode, hit run to test your code in our simulator easily.

Apart from what is mentioned above, the disassembler enjoys as more functions and features as the assembler.

4.4 GUI Mode (Text Editor)

Another hand-drawn graphic interface with two code display window and a console. You can run immediately in our GUI simulator after you have modified/compiled your codes.



5. Test

For details see the source listed below.
$Y86_ROOT/y86-code/genTestYo.sh
$Y86_ROOT/y86-code/test.sh
Manually compare the results with those run by lab4.
Mainly compare the result of *asum.yo* with that given by TA.

That's all. Thank you!