CS 349, Summer 2015

Optimizing the Performance of a Pipelined Processor

Assigned: June 6, Due: June 21, 11:59PM

Harry Bovik (bovik@cs.cmu.edu) is the lead person for this assignment.

* Introduction

In this lab, you will learn about the design and implementation of a pipelined Y86-64 processor, optimizing both it and a benchmark program to maximize performance. You are allowed to make any semantics-preserving transformation to the benchmark program, or to make enhancements to the pipelined processor, or both. When you have completed the lab, you will have a keen appreciation for the interactions between code and hardware that affect the performance of your programs.

The lab is organized into three parts, each with its own handin. In Part A you will write some simple Y86-64 programs and become familiar with the Y86-64 tools. In Part B, you will extend the SEQ simulator with a new instruction. These two parts will prepare you for Part C, the heart of the lab, where you will optimize the Y86-64 benchmark program and the processor design.

在本实验中，您将学习流水线Y86-64处理器的设计和实现，优化它和基准程序以最大限度地提高性能。您可以对基准测试程序进行任何语义保留的转换，或者对流水线处理器进行增强，或者两者兼而有之。当你完成实验后，你会对影响程序性能的代码和硬件之间的交互产生强烈的兴趣。

实验室分为三个部分，每个部分都有自己的工作。在A部分中，您将编写一些简单的Y86-64程序，并熟悉Y86-64工具。在B部分中，您将用一个新的指令扩展SEQ模拟器。这两个部分将为C部分做好准备，C部分是实验室的核心部分，您将优化Y86-64基准测试程序和处理器设计。

* Logistics

You will work on this lab alone.

Any clarifications and revisions to the assignment will be posted on the course Web page.

你将独自在这个实验室工作。

对作业的任何澄清和修改都将张贴在课程网页上。

* Handout Instructions

SITE-SPECIFIC: Insert a paragraph here that explains how students should download the **archlab-handout.tar** file.

站点特定：在这里插入一段解释学生应该如何下载archlab-讲义.tar文件。

1. Start by copying the file archlab-handout.tar to a (protected) directory in which you plan to do your work.

1

1. Then give the command: tar xvf archlab-handout.tar. This will cause the following

files to be unpacked into the directory: README, Makefile, sim.tar, archlab.pdf, and simguide.pdf.

1. Next, give the command tar xvf sim.tar. This will create the directory sim, which contains your personal copy of the Y86-64 tools. You will be doing all of your work inside this directory.
2. Finally, change to the sim directory and build the Y86-64 tools:

1.从复制文件archlab开始-讲义.tar到一个（受保护的）目录中，您计划在其中执行工作。

2.然后发出命令：tarxvf archlab-讲义.tar. 这将导致以下情况

要解压到目录中的文件：自述文件，Makefile，模拟焦油, 建筑实验室.pdf，和simguide.pdf版.

3.接下来，给命令tar xvf模拟焦油. 这将创建目录sim，其中包含您个人的Y86-64工具副本。你将在这个目录中完成你所有的工作。

4.最后，切换到sim目录，构建Y86-64工具：

unix> cd sim

unix> make clean; make

* Part A

You will be working in directory sim/misc in this part.

Your task is to write and simulate the following three Y86-64 programs. The required behavior of these programs is defined by the example C functions in examples.c. Be sure to put your name and ID in a comment at the beginning of each program. You can test your programs by first assemblying them with the program YAS and then running them with the instruction set simulator YIS.

In all of your Y86-64 functions, you should follow the x86-64 conventions for passing function arguments, using registers, and using the stack. This includes saving and restoring any callee-save registers that you use.

在这一部分中，您将在sim/misc目录下工作。

您的任务是编写并模拟以下三个Y86-64程序。这些程序所需的行为是由examples.C中的example C函数定义的。请确保在每个程序开头的注释中添加您的名称和ID。你可以测试你的程序，首先用程序YAS组装它们，然后用指令集模拟器YIS运行它们。

在所有Y86-64函数中，传递函数参数、使用寄存器和使用堆栈都应该遵循x86-64约定。这包括保存和还原您使用的任何被调用方保存寄存器。

**sum.ys**: Iteratively sum linked list elements

迭代求和链表元素

Write a Y86-64 program sum.ys that iteratively sums the elements of a linked list. Your program should consist of some code that sets up the stack structure, invokes a function, and then halts. In this case, the function should be Y86-64 code for a function (sum list) that is functionally equivalent to the C sum list function in Figure 1. Test your program using the following three-element list:

编写Y86-64程序总和Y对链表的元素进行迭代求和。您的程序应该由一些代码组成，这些代码设置堆栈结构，调用函数，然后停止。在这种情况下，函数应该是Y86-64代码，用于函数（sum list），它在功能上等同于图1中的C sum list函数。使用以下三个元素列表测试程序：

* Sample linked list

.align 8 ele1:

.quad 0x00a

.quad ele2

ele2:

.quad 0x0b0

.quad ele3

ele3:

.quad 0xc00

.quad 0

2

1 /\* linked list element \*/

2 typedef struct ELE {

* long val;
* struct ELE \*next; 5 } \*list\_ptr;

6

7 /\* sum\_list - Sum the elements of a linked list \*/

8 long sum\_list(list\_ptr ls)

* {

1. long val = 0;
2. while (ls) {
3. val += ls->val;
4. ls = ls->next;
5. }
6. return val;
7. }

17

18 /\* rsum\_list - Recursive version of sum\_list \*/

19 long rsum\_list(list\_ptr ls)

1. {
2. if (!ls)
3. return 0;
4. else {
5. long val = ls->val;
6. long rest = rsum\_list(ls->next);
7. return val + rest;
8. }
9. }

29

30 /\* copy\_block - Copy src to dest and return xor checksum of src \*/

31 long copy\_block(long \*src, long \*dest, long len)

1. {
2. long result = 0;
3. while (len > 0) {
4. long val = \*src++;
5. \*dest++ = val;
6. result ˆ= val;
7. len--;
8. }
9. return result;
10. }

Figure 1: **C versions of the Y86-64 solution functions.** See sim/misc/examples.c

3

**rsum.ys**: Recursively sum linked list elements递归求和链表元素

Write a Y86-64 program rsum.ys that recursively sums the elements of a linked list. This code should be similar to the code in sum.ys, except that it should use a function rsum list that recursively sums a list of numbers, as shown with the C function rsum list in Figure 1. Test your program using the same three-element list you used for testing list.ys.

编写Y86-64程序相对湿度它递归地对链表的元素求和。此代码应与中的代码类似总和Y，但它应该使用一个函数rsum list递归地对一个数字列表求和，如图1中的C函数rsum list所示。使用与测试相同的三元素列表测试程序列表.ys.

**copy.ys**: Copy a source block to a destination block将源块复制到目标块

Write a program (copy.ys) that copies a block of words from one part of memory to another (non-overlapping area) area of memory, computing the checksum (Xor) of all the words copied.

Your program should consist of code that sets up a stack frame, invokes a function copy block, and then halts. The function should be functionally equivalent to the C function copy block shown in Figure Figure 1. Test your program using the following three-element source and destination blocks:

编写程序(copy.ys)它将一个字块从内存的一部分复制到另一个（非重叠区域）内存区域，计算复制的所有单词的校验和（Xor）。

您的程序应该由设置堆栈框架、调用函数复制块然后停止的代码组成。该函数应该在功能上等同于图1所示的C函数复制块。使用以下三个元素源和目标块测试程序：

.align 8

* Source block src:

.quad 0x00a

.quad 0x0b0

.quad 0xc00

* Destination block dest:

.quad 0x111

.quad 0x222

.quad 0x333

* Part B

You will be working in directory sim/seq in this part.

Your task in Part B is to extend the SEQ processor to support the iaddq, described in Homework problems 4.51 and 4.52. To add this instructions, you will modify the file seq-full.hcl, which implements the version of SEQ described in the CS:APP3e textbook. In addition, it contains declarations of some constants that you will need for your solution.

**在本部分中，您将在sim/seq目录下工作。**

**B部分的任务是扩展SEQ处理器以支持iaddq，如家庭作业问题4.51和4.52所述。要添加此说明，您将修改文件**seq-full.hcl**，它实现了CS:APP3e教科书中描述的SEQ版本。此外，它还包含解决方案所需的某些常量的声明。**

Your HCL file must begin with a header comment containing the following information:

Your name and ID.

A description of the computations required for the iaddq instruction. Use the descriptions of irmovq and OPq in Figure 4.18 in the CS:APP3e text as a guide.

HCL文件必须以包含以下信息的标题注释开头：

你的名字和身份证。

对iaddq指令所需计算的描述。使用CS:APP3e文本中图4.18中irmovq和OPq的描述作为指南。

4

Building and Testing Your Solution

Once you have finished modifying the seq-full.hcl file, then you will need to build a new instance of the SEQ simulator (ssim) based on this HCL file, and then test it:

构建和测试解决方案

一旦你完成了seq的修改seq-full.hcl，则需要基于此HCL文件构建SEQ simulator（ssim）的新实例，然后对其进行测试：

Building a new simulator. You can use make to build a new SEQ

建造一个新的模拟器。你可以使用make来构建一个新的SEQ simulator: unix> make VERSION=full

This builds a version of ssim that uses the control logic you specified in seq-full.hcl. To save typing, you can assign VERSION=full in the Makefile.

Testing your solution on a simple Y86-64 program. For your initial testing, we recommend running simple programs such as asumi.yo (testing iaddq) in TTY mode, comparing the results against the ISA simulation:

这将构建使用seq中指定的控制逻辑的ssim版本-全盐酸. 要保存键入，可以在生成文件中指定VERSION=full。

在一个简单的Y86-64程序上测试您的解决方案。对于初始测试，我们建议运行一些简单的程序，例如asumi.yo（测试iaddq）在TTY模式下，将结果与ISA模拟进行比较：

unix> ./ssim -t ../y86-code/asumi.yo

If the ISA test fails, then you should debug your implementation by single stepping the simulator in GUI mode:

如果ISA测试失败，则应在GUI模式下通过单步执行模拟器来调试实现：

unix> ./ssim -g ../y86-code/asumi.yo

Retesting your solution using the benchmark programs. Once your simulator is able to correctly execute small programs, then you can automatically test it on the Y86-64 benchmark programs in

../y86-code:

使用基准测试程序重新测试解决方案。一旦您的模拟器能够正确执行小程序，那么您就可以在中的Y86-64基准测试程序上自动测试它

../y86代码：

unix> (cd ../y86-code; make testssim)

This will run ssim on the benchmark programs and check for correctness by comparing the resulting processor state with the state from a high-level ISA simulation. Note that none of these programs test the added instructions. You are simply making sure that your solution did not inject errors for the original instructions. See file ../y86-code/README file for more details.

这将在基准程序上运行ssim，并通过将生成的处理器状态与高级ISA模拟的状态进行比较来检查其正确性。请注意，这些程序没有测试添加的指令。您只需确保您的解决方案不会为原始指令插入错误。有关详细信息，请参阅文件../y86代码/自述文件。

报错

拼音 双语对照

Performing regression tests. Once you can execute the benchmark programs correctly, then you should run the extensive set of regression tests in ../ptest. To test everything except iaddq and leave:

进行回归测试。一旦您能够正确地执行基准测试程序，那么您应该在../ptest中运行大量的回归测试。要测试除iaddq之外的所有内容并离开：

unix> (cd ../ptest; make SIM=../seq/ssim)

To test your implementation of iaddq:

要测试iaddq的实现：

unix> (cd ../ptest; make SIM=../seq/ssim TFLAGS=-i)

For more information on the SEQ simulator refer to the handout CS:APP3e Guide to Y86-64 Processor Simulators (simguide.pdf).

有关SEQ模拟器的更多信息，请参阅讲义CS:APP3e指南Y86-64处理器模拟器(simguide.pdf版).

5

* /\*
* \* ncopy - copy src to dst, returning number of positive ints

3 \* contained in src array.

* \*/

5 word\_t ncopy(word\_t \*src, word\_t \*dst, word\_t len)

* {
* word\_t count = 0;

8word\_t val;

9

1. while (len > 0) {
2. val = \*src++;
3. \*dst++ = val;
4. if (val > 0)

14 count++;

1. len--;
2. }
3. return count;
4. }

Figure 2: **C version of the** **ncopy** **function.** See sim/pipe/ncopy.c.

* Part C

You will be working in directory sim/pipe in this part.

The ncopy function in Figure 2 copies a len-element integer array src to a non-overlapping dst, re-turning a count of the number of positive integers contained in src. Figure 3 shows the baseline Y86-64 version of ncopy. The file pipe-full.hcl contains a copy of the HCL code for PIPE, along with a declaration of the constant value IIADDQ.

您将在sim/pipe目录下工作。

图2中的ncopy函数将len element integer array src复制到不重叠的dst中，重新返回src中包含的正整数的计数。图3显示了ncopy的基线Y86-64版本。文件管道-全盐酸包含管道的HCL代码副本，以及常量值IIADDQ的声明。

Your task in Part C is to modify ncopy.ys and pipe-full.hcl with the goal of making ncopy.ys run as fast as possible.

117/5000

C部分中的任务是修改ncopy.ys和pipe-full.hcl，以使ncopy.ys尽可能快地运行。

You will be handing in two files: pipe-full.hcl and ncopy.ys. Each file should begin with a header comment with the following information:

您将提交两个文件：pipe-full.hcl和ncopy.ys。 每个文件应以带有以下信息的标题注释开头：

Your name and ID.

A high-level description of your code. In each case, describe how and why you modified your code.

您的姓名和身份证。

您的代码的高级描述。 在每种情况下，请描述修改代码的方式和原因。

Coding Rules

You are free to make any modifications you wish, with the following constraints:

编码规则

您可以根据以下约束自由进行所需的任何修改：

Your ncopy.ys function must work for arbitrary array sizes. You might be tempted to hardwire your solution for 64-element arrays by simply coding 64 copy instructions, but this would be a bad idea because we will be grading your solution based on its performance on arbitrary arrays.

您的ncopy.ys函数必须适用于任意数组大小。 您可能会想通过简单地编写64个复制指令来为64个元素的阵列硬连接解决方案，但这将是一个坏主意，因为我们将根据其在任意阵列上的性能来对您的解决方案进行分级。

6

* ##################################################################

2 # ncopy.ys - Copy a src block of len words to dst.

3 # Return the number of positive words (>0) contained in src.

* #

5 # Include your name and ID here.

* #

7 # Describe how and why you modified the baseline code.

* #＃ncopy.ys-将len个单词的src块复制到dst。
* 3＃返回src中包含的正数（> 0）。
* 5＃在此处输入您的姓名和ID。
* 7＃描述如何以及为何修改基线代码。
* ##################################################################

10 # Do not modify this portion

11 # Function prologue.

12 # %rdi = src, %rsi = dst, %rdx = len

13 ncopy:

14

1. ##################################################################
2. # You can modify this portion
3. # Loop header

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 18 |  | xorq %rax,%rax | # count = 0; | |
| 19 |  | andq %rdx,%rdx | # len <= | 0? |
| 20 |  | jle Done | # if so, | goto Done: |
| 21 |  |  |  |  |
| 22 | Loop: | mrmovq (%rdi), %r10 | # read val from src... | |
| 23 |  | rmmovq %r10, (%rsi) | # ...and | store it to dst |
| 24 |  | andq %r10, %r10 | # val <= | 0? |
| 25 |  | jle Npos | # if so, | goto Npos: |

1. irmovq $1, %r10

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 27 |  | addq %r10, %rax | | # count++ |
| 28 | Npos: | irmovq $1, | %r10 |  |
| 29 |  | subq %r10, | %rdx | # len-- |

1. irmovq $8, %r10

|  |  |  |
| --- | --- | --- |
| 31 | addq %r10, %rdi | # src++ |
| 32 | addq %r10, %rsi | # dst++ |
| 33 | andq %rdx,%rdx | # len > 0? |
| 34 | jg Loop | # if so, goto Loop: |

1. ##################################################################
2. # Do not modify the following section of code
3. # Function epilogue.
4. Done:
5. ret
6. ##################################################################
7. # Keep the following label at the end of your function
8. End:

Figure 3: **Baseline Y86-64 version of the** **ncopy** **function.** See sim/pipe/ncopy.ys.

7

Your ncopy.ys function must run correctly with YIS. By correctly, we mean that it must correctly copy the src block and return (in %rax) the correct number of positive integers.

The assembled version of your ncopy file must not be more than 1000 bytes long. You can check the length of any program with the ncopy function embedded using the provided script check-len.pl:

unix> ./check-len.pl < ncopy.yo

Your pipe-full.hcl implementation must pass the regression tests in ../y86-code and ../ptest (without the -i flag that tests iaddq).

Other than that, you are free to implement the iaddq instruction if you think that will help. You may make any semantics preserving transformations to the ncopy.ys function, such as reordering instruc-tions, replacing groups of instructions with single instructions, deleting some instructions, and adding other instructions. You may find it useful to read about loop unrolling in Section 5.8 of CS:APP3e.

Building and Running Your Solution

In order to test your solution, you will need to build a driver program that calls your ncopy function. We have provided you with the gen-driver.pl program that generates a driver program for arbitrary sized input arrays. For example, typing

unix> make drivers

will construct the following two useful driver programs:

sdriver.yo: A small driver program that tests an ncopy function on small arrays with 4 elements. If your solution is correct, then this program will halt with a value of 2 in register %rax after copying the src array.

ldriver.yo: A large driver program that tests an ncopy function on larger arrays with 63 ele-ments. If your solution is correct, then this program will halt with a value of 31 (0x1f) in register %rax after copying the src array.

Each time you modify your ncopy.ys program, you can rebuild the driver programs by typing

unix> make drivers

Each time you modify your pipe-full.hcl file, you can rebuild the simulator by typing

unix> make psim VERSION=full

If you want to rebuild the simulator and the driver programs, type

unix> make VERSION=full

8

To test your solution in GUI mode on a small 4-element array, type

unix> ./psim -g sdriver.yo

To test your solution on a larger 63-element array, type

unix> ./psim -g ldriver.yo

Once your simulator correctly runs your version of ncopy.ys on these two block lengths, you will want to perform the following additional tests:

Testing your driver files on the ISA simulator. Make sure that your ncopy.ys function works prop-erly with YIS:

unix> make drivers

unix> ../misc/yis sdriver.yo

Testing your code on a range of block lengths with the ISA simulator. The Perl script correctness.pl generates driver files with block lengths from 0 up to some limit (default 65), plus some larger sizes.

It simulates them (by default with YIS), and checks the results. It generates a report showing the status for each block length:

unix> ./correctness.pl

This script generates test programs where the result count varies randomly from one run to another, and so it provides a more stringent test than the standard drivers.

If you get incorrect results for some length K, you can generate a driver file for that length that includes checking code, and where the result varies randomly:

unix> ./gen-driver.pl -f ncopy.ys -n K -rc > driver.ys unix> make driver.yo

unix> ../misc/yis driver.yo

The program will end with register %rax having the following value:

**0xaaaa** : All tests pass.

**0xbbbb** : Incorrect count

**0xcccc** : Function ncopy is more than 1000 bytes long.

**0xdddd** : Some of the source data was not copied to its destination.

**0xeeee** : Some word just before or just after the destination region was corrupted.

Testing your pipeline simulator on the benchmark programs. Once your simulator is able to cor-rectly execute sdriver.ys and ldriver.ys, you should test it against the Y86-64 benchmark programs in ../y86-code:

unix> (cd ../y86-code; make testpsim)

9

This will run psim on the benchmark programs and compare results with YIS.

Testing your pipeline simulator with extensive regression tests. Once you can execute the benchmark programs correctly, then you should check it with the regression tests in ../ptest. For example, if your solution implements the iaddq instruction, then

unix> (cd ../ptest; make SIM=../pipe/psim TFLAGS=-i)

Testing your code on a range of block lengths with the pipeline simulator. Finally, you can run the same code tests on the pipeline simulator that you did earlier with the ISA simulator

unix> ./correctness.pl -p

* Evaluation

The lab is worth 190 points: 30 points for Part A, 60 points for Part B, and 100 points for Part C.

Part A

Part A is worth 30 points, 10 points for each Y86-64 solution program. Each solution program will be eval-uated for correctness, including proper handling of the stack and registers, as well as functional equivalence with the example C functions in examples.c.

The programs sum.ys and rsum.ys will be considered correct if the graders do not spot any errors in them, and their respective sum list and rsum list functions return the sum 0xcba in register %rax.

The program copy.ys will be considered correct if the graders do not spot any errors in them, and the copy block function returns the sum 0xcba in register %rax, copies the three 64-bit values 0x00a, 0x0b, and 0xc to the 24 bytes beginning at address dest, and does not corrupt other memory locations.

Part B

This part of the lab is worth 35 points:

10 points for your description of the computations required for the iaddq instruction.

10 points for passing the benchmark regression tests in y86-code, to verify that your simulator still correctly executes the benchmark suite.

15 points for passing the regression tests in ptest for iaddq.

Part C

This part of the Lab is worth 100 points: You will not receive any credit if either your code for ncopy.ys or your modified simulator fails any of the tests described earlier.

10

20 points each for your descriptions in the headers of ncopy.ys and pipe-full.hcl and the quality of these implementations.

60 points for performance. To receive credit here, your solution must be correct, as defined earlier. That is, ncopy runs correctly with YIS, and pipe-full.hcl passes all tests in y86-code and ptest.

We will express the performance of your function in units of cycles per element (CPE). That is, if the simulated code requires C cycles to copy a block of N elements, then the CPE is C=N. The PIPE simulator displays the total number of cycles required to complete the program. The baseline version of the ncopy function running on the standard PIPE simulator with a large 63-element array requires 897 cycles to copy 63 elements, for a CPE of 897=63 = 14:24.

Since some cycles are used to set up the call to ncopy and to set up the loop within ncopy, you will find that you will get different values of the CPE for different block lengths (generally the CPE will drop as N increases). We will therefore evaluate the performance of your function by computing the average of the CPEs for blocks ranging from 1 to 64 elements. You can use the Perl script benchmark.pl in the pipe directory to run simulations of your ncopy.ys code over a range of block lengths and compute the average CPE. Simply run the command

本部分实验的总分是100分：如果您的ncopy.ys代码或修改后的模拟器未通过前面所述的任何测试，您将不会获得任何积分。

ncopy.ys和pipe-full.hcl的标头中的描述各占20分，以及这些实现的质量。

性能得分60分。要在这里获得信誉，您的解决方案必须是正确的（如前所述）。也就是说，ncopy可以在YIS上正确运行，并且pipe-full.hcl通过y86代码和ptest中的所有测试。

我们将以每元素周期数（CPE）表示您的功能性能。也就是说，如果仿真代码需要C个周期来复制N个元素的块，则CPE为C = N。 PIPE仿真器显示完成程序所需的总循环数。在具有63个元素的大型数组的标准PIPE仿真器上运行的ncopy函数的基准版本需要897个周期来复制63个元素，CPE为897 = 63 = 14:24。

由于使用了一些周期来建立对ncopy的调用并在ncopy中建立循环，因此您会发现对于不同的块长度，您将获得不同的CPE值（通常，CPE将随着N的增加而下降）。因此，我们将通过计算1到64个元素的块的CPE平均值来评估您的功能性能。您可以在管道目录中使用Perl脚本Benchmark.pl在一定范围的块长度上运行ncopy.ys代码的模拟并计算平均CPE。只需运行命令

unix> ./benchmark.pl

to see what happens. For example, the baseline version of the ncopy function has CPE values ranging between 29:00 and 14:27, with an average of 15:18. Note that this Perl script does not check for the correctness of the answer. Use the script correctness.pl for this.

You should be able to achieve an average CPE of less than 9:00. Our best version averages 7:48. If your average CPE is c, then your score S for this portion of the lab will be:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| S = | 8 | 20 |  | (10:5 |  | c) ; 7:50 c |  | 10:50 |  |
|  | > | 0 ; |  |  |  | c > 10:5 |  |  |  |
|  |  |  |  |  |  |
|  | < | 60 ; | |  |  | c < 7:50 |  |  |  |
|  | > |  |  |  |  |  |  |  |  |

:

By default, benchmark.pl and correctness.pl compile and test ncopy.ys. Use the -f argument to specify a different file name. The -h flag gives a complete list of the command line arguments.

* Handin Instructions

SITE-SPECIFIC: Insert a description that explains how students should hand in the three parts of the lab. Here is the description we use at CMU.

You will be handing in three sets of files:

– Part A: sum.ys, rsum.ys, and copy.ys.

– Part B: seq-full.hcl.

11

– Part C: ncopy.ys and pipe-full.hcl.

Make sure you have included your name and ID in a comment at the top of each of your handin files. To handin your files for part X, go to your archlab-handout directory and type:

unix> make handin-partX TEAM=teamname

where X is a, b, or c, and where teamname is your ID. For example, to handin Part A:

unix> make handin-parta TEAM=teamname

After the handin, if you discover a mistake and want to submit a revised copy, type unix make handin-partX TEAM=teamname VERSION=2

Keep incrementing the version number with each submission.

You can verify your handin by looking in

CLASSDIR/archlab/handin-partX

You have list and insert permissions in this directory, but no read or write permissions.

* Hints

By design, both sdriver.yo and ldriver.yo are small enough to debug with in GUI mode. We find it easiest to debug in GUI mode, and suggest that you use it.

If you running in GUI mode on a Unix server, make sure that you have initialized the DISPLAY environment variable:

unix> setenv DISPLAY myhost.edu:0

With some X servers, the “Program Code” window begins life as a closed icon when you run psim or ssim in GUI mode. Simply click on the icon to expand the window.

With some Microsoft Windows-based X servers, the “Memory Contents” window will not automati-cally resize itself. You’ll need to resize the window by hand.

The psim and ssim simulators terminate with a segmentation fault if you ask them to execute a file that is not a valid Y86-64 object file.

12