Department of Computing and Information Systems
COMP20006 Programming the Machine
Semester 1, 2012

Project 1: Due Friday April 20th
Review due Friday May 4th

1 Aim

The main aim of this project is to give you experience with C programming, bit manipulation, and machine code. The secondary aim is for you is to gain experience with version control and program testing.

2 Deadline

The deadline for this project is 11pm on Friday April 20th.

This project is an individual project in which the submitted files must be your own individual work.

3 Task specification

Your task in this project is to write a C program named `analyse.c`, which is intended to be a simplified version of a tool that analyse a MIPS machine code program. It will print out information about the basic blocks of the machine code.

This is a staged project, complete the stages in order.

3.1 Stage A: Leader Determination

In the first stage you will print out all the *leaders* in a machine code. Leaders are addresses whose instruction is *not always* executed straight after the instruction before.

The input to this program will consist of two parts.

- The first part of the input is a sequence of 31 hexadecimal numbers specifying the contents of MIPS machine registers `$1` through `$31`. These contents are arbitrary 32 bit integers. Register `$0` always contains zero, which is why the program does not need to read in its value.
- The second part of the input is a sequence of zero or more hexadecimal numbers not including `0xffffffff`, followed by `0xffffffff` which is reserved for terminating the sequence. Each of the hexadecimal numbers, except the last one, is the encoding of a MIPS machine language instruction. You can assume there are no more than MAX-SIZE=1000 instructions in the second part of the input.

Your program should start by reading in the values of the registers, which you should then assume are fixed for the rest of the program. For each of the instructions that follow the register values, you program should determine whether it is

- one of the branch instructions `beq`, `bgez`, `bgtz`, `blez`, `bltz`, and `bne`; or
- one of the jump instructions `j`, `jal`, `jalr`, and `jr`; or
- the `syscall` instruction; or
- some other kind of instruction.

After determining the current instruction your program should:

- ignore the instruction if it falls into the last category;
- for `j` or `jr` instructions, determine what address it will jump to, and should output this address;
- for branch instructions, `jal` or `jalr` instruction it should find out what address the instruction may jump to and output this, and then output the address of the next instruction after the current one (since the conditional branch may fall through to there, or the jump-and-link instruction will probably jump back to there);
- for the `syscall` instruction it should output the address of the next instruction only (we could imagine also recording which system calls are made as well but we are interested in local addresses only);

The output should be one hex address (leader) per line as in the following *sample output*:

```
00400024
00400018
00400024
00400048
00400034
00400040
0040002c
00400018
```

To do all this, you will need to decode some aspects of MIPS instructions and keep track of the current address. Specifically, you will need to:

- keep track of the current location. You can assume the first instruction is at address `0x00400000` and each further instruction is four bytes later.
- locate the opcode field of each instruction and test it to see whether it is one the opcodes of interest;
- locate the number of the register used in the `jr` and `jalr` instructions (this may require some experimentation with SPIM); and

- locate the immediate field for each instruction `beq`, `bgez`, `bgtz`, `blez`, `bltz`, `bne`, `j`, `jal`, and do some address arithmetic for branch instructions.

Consider the small program from the slides, which when loaded into SPIM gives

```
[0x00400000]    0x8fa40000  lw     $a0, 0($sp)
[0x00400004]    0x27a50004  addiu  $a1, $sp, 4
[0x00400008]    0x24a60004  addiu  $a2 $a1 4
[0x0040000c]    0x00041080  sll    $v0 $a0 2
[0x00400010]    0x00c23021  addu   $a2 $a2 $v0
[0x00400014]    0x0c100009  jal    main
[0x00400018]    0x00000000  nop
[0x0040001c]    0x3402000a  li     $v0 10
[0x00400020]    0x0000000c  syscall
main:
[0x00400024]    0x34100001  li     $s0, 1        # i = 1
[0x00400028]    0x3411000a  li     $s1, 10       # limit = 10
test:
[0x0040002c]    0x0230402a  sgt    $t0, $s0, $s1 # is i > limit?
[0x00400030]    0x1d000006  bgtz   $t0, done     # if yes, goto "done"
[0x00400034]    0x34020001  li     $v0, 1        # 1 means print_int
[0x00400038]    0x72102002  mul    $a0, $s0, $s0 # square = i * i
[0x0040003c]    0x0000000c  syscall
[0x00400040]    0x22100001  addi   $s0, $s0, 1   # i = i + 1
[0x00400044]    0x0810000b  j      test
done:
[0x00400048]    0x03e00008  jr $ra
```

The input to your program for this example may look like

```
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00400018
8fa40000 27a50004 24a60004 00041080 00c23021 0c100009 00000000 3402000a
0000000c
34100001 3411000a 0230402a 1d000006 34020001 72102002 0000000c 22100001
0810000b 03e00008
ffffffff
```

where all registers are 0 except `$31` which is `00400018`. The *EXACT* expected output for this input is the *sample output* above.

3.2 Stage B: Basic Blocks

Each address which is The real aim of finding the leaders of a bit of machine code is to find those sets of instructions that must be executed in sequence, in order to optimize them. The sequences of instructions which will always be executed in sequence are called *basic blocks* and go from one leader to the instruction before the next leader. Note that `0x00400000` is also always a leader although we won't print it out above.

The output for stage B, the basic blocks, should be printed out as follows for the small example above *Hint: you dont need to sort the leaders, there is an easier way.*

```

00400000 - 00400014
00400018 - 00400020
00400024 - 00400028
0040002c - 00400030
00400034 - 0040003c
00400040 - 00400044
00400048 - 00400048
```

Note that there must be a blank line at the top, separating them from the output of stage A. You can ignore leaders determined in the previous stage that dont occur in the address space of the program read in. For example if `$31` was initialized to `00000018` instead then the stage A output would change (last line `00000018`) but not stage B.

3.3 Extension Stage C

Note that this stage is much more work than the previous two put together, but is worth less marks. Now we have to model which registers are used in each basic block. For this part we now have to examine all the commands, not just those related to branching. You will notice there are really two major forms of machine code instructions using registers

•

0	rs	rt	rd	?	?
6	5	5	5	5	6

•

?	rs	ru	imm
6	5	5	16

For each basic block you need to create a 32 bit hexadecimal number recording which registers are *read* (used as rs, rt or ru in a branch instruction) within the basic block and another 32 bit number to record which registers are *written* (used as rd or ru in a non branch instruction) within the basic block. You can ignore the `$0` register.

For example in the first basic block registers `$4` (`$a0`), `$5` (`$a1`), `$6` (`$a2`), and `$2` (`$v0`) are written to. *BUT* there is also a write to `$31` (`$ra`) hard coded in the `jal` instruction! This is encoded as the 32 bits `10000000000000000000000001110100` = `0x80000074`. In the first basic block the first 4 registers above are also read from as well as register `$29` (`$sp`), this is encoded as the 32 bits `00100000000000000000000001110100` = `0x20000074`.

The output for stage C must follow the output of stage B (separated by a blank line), giving the basic block and two 32 bit hexadecimal numbers encoding the registers read and written in the block. For the example it should be exactly:

```

00400000 - 00400014 : 20000074 80000074
00400018 - 00400020 : 00000000 00000004
00400024 - 00400028 : 00000000 00030000
```

```
0040002c - 00400030 : 00030100 00000100
00400034 - 0040003c : 00010000 00000014
00400040 - 00400044 : 00010000 00010000
00400048 - 00400048 : 80000000 00000000
```

You dont need to handle all MIPS instructions but at least: `addiu`, `addi`, `addu`, `add`, `andi`, `and`, `beq`, `bgez`, `bgtz`, `blez`, `bltz`, `bne`, `jalr`, `jal`, `jr`, `j`, `lb`, `lh`, `lui`, `lw`, `mul`, `nop`, `nor`, `ori`, `or`, `sb`, `sh`, `slt`, `sltu`, `slti`, `sltiu`, `subu`, `sub`, `sw`, `syscall`, `xori`, `xor`.

4 Using subversion

You must use **subversion** to record your progress during the development of your code for the project. See Section 3 of the notes for more information on subversion. We have created a personal subversion repository directory for you to use, and checked it out as the directory `comp20006` in your home directory. This directory has a subdirectory named `comp20006/project1`. You should do your work for the project in this directory. You should execute `svn add analyse.c` when you start work on `analyse.c`. When you complete a significant change, you should check it in, with a descriptive log message.

5 Submission

When you are ready to submit a version of your `analyse.c`, copy it to your `comp20006/project1/submit` directory, execute `svn add analyse.c` in that directory if you have not already done so, and check it in (`svn ci`). Basically, your `comp20006/project1` directory is for your work-in-progress, and your `comp20006/project1/submit` directory is for submitted work.

That is all you have to do to submit your work. At the deadline, we will make a copy of the relevant part of your subversion repository, and find the snapshot of `analyse.c` that you most recently committed in `comp20006/project1/submit`. The markers will mark this version of `analyse.c`. If you do not copy `analyse.c` to this directory, or if you forget to execute `svn add` or `svn ci` on it, there will be nothing to mark, and you will get a zero.

The marking process will start by checking

- whether your program has been properly checked in,
- whether it compiles without any errors and warnings, and
- whether its executable generates the right outputs for a small set of test cases.

This will all be done by a simple software tool. We will make this tool available to you in `/home/subjects/comp20006/local/project1`, together with the expected output generator and some test cases.

While we will normally look at only the last version of the program in your submit directory, we reserve the right to look at the program's development history in cases of disputed authorship, and we will be actively looking for submissions that are too similar. Your submission is supposed to reflect your own unaided work.

Marks will be out of 10 with a possibility of a bonus mark for complete handling of all MIPS instructions (excluding floating point instructions) in part C. You will be assessed on your understanding of the project, as evidenced by the design of your solution; your programming style with respect to clarity, logical structure, and modularity; the robustness of your program; and your standard of documentation, as well as correctness of the program.

6 Extensions

You may request an extension for one of two reasons.

The first reason is that there is some factor outside your control that is affecting your ability to work on this project. In such cases, the department typically asks for whatever documentation is practicable under the circumstances (such as a note from a medical practitioner), but if the extension is granted, there is no penalty. You need to apply for such extensions *before* the deadline.

The other reason is that you simply haven't got far enough in your work, and think you can do significantly better given a bit of extra time. Late submissions of this kind will incur a penalty of 1 mark per day (or part thereof) for the first two days after the deadline and a penalty of 2 marks per day (or part thereof) after that. Note if you do not request this extension *before the deadline* we will use the version in the submit directory at the due date.

Note that machine breakdowns (within reason) and projects in other subjects are part of life, and you must learn to deal with them. One way you can do this is by starting work on your projects as soon as possible. Therefore they will not be acceptable as an excuse for not having a project completed by the due date.

7 Code review

Reading other people's code is educational, particularly if it is code for a task you have already attempted yourself. If the other person's code is better in some way, e.g. it has fewer bugs or is clearer or better commented, you can learn from that; if the other person's code is worse in some way, you can learn from that too. There will therefore be a code review step as part of this project.

The CIS department normally asks students to put a comment giving their name at the top of every file submitted for assessment. To preserve your privacy, we ask that the first few lines of your program should look like this:

```
// COMP20006 Project 1
// HIDE_START
// Author: John Doe
// HIDE_END
```

The only place where you should put any text that identifies you as the author of this code is between the comment lines containing `HIDE_START` and `HIDE_END`. When we make submitted programs available for review, we will strip out any lines between lines containing the words `HIDE_START` and `HIDE_END`, thus protecting your privacy.

After the project deadline, each project will be forwarded to two other students for review. The reviews are due May 4th 2011 at 11pm. Code reviews should:

- Comment on the layout, commenting, and coding style of the submission.

- Point out strong and weak points in the implementation, if any.
- Make suggestions for improving the implementation, if any.

Code reviews will be returned to the code author for their consideration. Both code and reviews will be anonymous.

Each student will receive two submissions to review. The submissions will be inserted into their subversion repository as `comp20006/reviews/XXXX-disasm.c` and `comp20006/reviews/YYYY-disasm.c`. It will also insert two blank text files `comp20006/reviews/XXXX-review.txt` and `comp20006/reviews/YYYY-review.txt`. You will be informed by email when your review submissions are ready; you will then have to execute `svn update` to receive them in your working directory.

The reviews should be written as plain text files by editing the empty files that have been checked out. To submit reviews, do `svn commit`.

Reviewers will be marked on the quality of their reviews, but the reviews will have no impact on the marks received by the original authors of the code.

Feel free to include small fragments of the code you are reviewing in the review, in order to point out things, but do not just include the whole thing.

8 Marking

This project is worth 10% of the marks for the subject.

The marking scheme will give marks for

- programming style, including good variable names, appropriate and useful comments, and consistent indentation that reflects the structure of the program; and for use of subversion. [2 marks]
- correctness of the code that does input and output. [1 mark]
- correctness of the code that detects leaders (stage A). [2 marks]
- correctness of the code that outputs basic blocks (stage B [2 marks]
- Correctness of the code stage C [1 mark + 1 bonus mark]).
- quality of the code review. [2 marks]

The maximum mark for the project is 11/10!

Since this marking scheme is only a draft, we reserve the right to change it, but we don't expect that to be necessary.

This marking scheme is fairly typical for subjects offered by the CIS department. For us, a working program, while important, is *not* the only criterion for assessment. You may also be assessed on your understanding of the project, as evidenced by the design of your solution; your programming style with respect to clarity, logical structure, and modularity; the robustness of your program; and your standard of documentation. The last is especially important: it is your responsibility to help us understand your code.