THE UNIVERSITY OF WESTERN ONTARIO

Computer Science 2208b - Fall 2012
Fundamentals of Computer Organization

ASSIGNMENT #2 - WEIGHT: 5%

Due October 12, 2012, at 11:59 PM

Part I

Programming - Heart Rate Monitor (75 Marks)

You are working as an embedded systems programmer for Polar's sports products division. As part of a new elite training heart-rate monitor that Polar will be introducing to market shortly, you've been asked to write a small application to help users determine what their target heart rate should be while exercising.

To keep things simple, you'll be computing target heart rate based solely on the users age and using the formula:

$HR_{Max} = 205 - (age)/2$

The application will ask the user for their age, compute the maximum heart rate and then return 4 values to the user:

Maximum Heart Rate (HR_{max})
Target Anaerobic Heart Rate ($85\% \, of \, HR_{max}$)
Target Aerobic Heart Rate ($75\% \, of \, HR_{max}$)
Light Exercise Heart Rate ($55\% \, of \, HR_{max}$

Simple enough; There's only one difficulty:
the device you are writing the application for is very simple and has to run with very low current draw to prolong battery life. This means there is almost no operating system to speak of, and certainly no fancy compilers supporting high level languages. The device must be programmed directly in assembly language.

For the purposes of this assignment, we'll pretend you are using a special embedded SPARC that only allows INTEGER operations and NO branching. So, your task for this assignment is to write the appropriate SPARC assembly language program that performs the heart rate calculations. Again, this is trivial in a high-level language with floating-point variables, but not so in assembly language when you are allowed to use only integers and no branching instructions!

1 Learning Objectives

The purpose of this part of the assignment is to have you:

1. begin programming SPARC assembly language by writing a program that uses only integer arithmetic operations and simple I/O routines
2. use macros to define constants and to define symbolic names for registers
3. follow specifications
4. practice good programming style (including writing good comments)

2 Program Details

Your task for this part of the assignment is to write a SPARC assembly language program (subject to the aforementioned restrictions) that prompts the user for their age, performs the heart rate calculations, and outputs them in integer values.

Specifically, your program should:

- Input the users age as an integer
- Output the Maximum Heart Rate, rounded to the nearest bpm, followed by a new line character
- Output the Anaerobic Heart Rate, rounded to the nearest bpm, followed by a new line character
- Output the Aerobic Heart Rate, rounded to the nearest bpm, followed by a new line character
- Output the Light Exercise Heart Rate, rounded to the nearest bpm, followed by a new line character

3 Sample Run

We will be using an automated program to run tests on your code. This means that the output produced by your program must match *exactly* that shown below in order to receive full marks.

Note: we will still, of course, be examining your code, and marking for style, comments, etc. We are simply using a program to run tests on your code to speed up marking and get your assignment back to you sooner.

```
obelix[62]% ./asn2
>20
195
166
146
107
obelix[63]% ./asn2
>37
187
159
140
103
obelix[63]%
```

Observe that a new line IS printed at the end of the program.

4 Implementation Specifications

4.1 Restrictions

Because you are developing for a simplified SPARC-based microcontroller, the only instructions that you have available to you are:

- `mov`, `add`, `sub` (and the "synthetic instructions" `clr`, `inc`, `dec`)
- `call`, `nop`

Note that `save`, `ret`, `restore` must also be used as in the sample programs.

You are also restricted to calling only the following functions:

- `.mul`, `.div`, `.rem`
- `readInt`, `readChar`, `writeInt`, `writeChar`

4.2 Programming Conventions

1. Use macros to give symbolic names to constants. You should not be using "magic numbers". By convention, constant names are *capitalized*, and multiple words in a constant are separated by underscores (e.g. `NUM_CHARS`).
2. Use only local registers (`%l0` to `%l7`) for temporary storage, and registers `%o0` and `%o1` for function parameters as per convention. Efficiency note: you can leave a result that is returned in `%o0`, if you will then be using it as the input parameter in `%o0` for the next function call.
3. You may not use memory locations (variables).
4. Use macros to give symbolic names to the registers that you use, EXCEPT FOR the registers `%o0` and `%o1` (i.e. do **not** use symbolic names for registers `%o0` and `%o1`). By convention, symbolic register names are all in lowercase and end in `_r` (for example, `time_r`).
5. You are expected to use good programming style and include complete commenting (see the **Commenting Guidelines** on the CS 2208 web site).

4.3 Input and Output Specifications

1. When prompting the user for a time value, use the single character prompt `>`.
2. Once again, please ensure that your output matches exactly the output shown in the **Sample Run** section.
3. Do not attempt to display any messages; don't use any string I/O routines in this assignment, even if you know about them.
4. In this assignment, do not worry about arithmetic overflow.
5. We will assume that the user will enter valid data.

5 Implementation Hints

- Remember that you can only deal with integers, not floating-point numbers!
- Look up **fixed-point representation**. If you're too lazy to do that, look at this example: If I'm given the floating-point value 0.85, how can a represent this without a decimal point? Well, I could just multiply it by 100, like this: 0.85*100=85 and then make a mental note that, at some point, I'm going to have to divide that 100 back out to get the correct answer.
- When we multiply a floating-point value by some integer to make it a fixed point value, we call that integer the scale factor. In the example above, the scale factor was 100.
- To get the actual integer value of your fixed-point number, you just have to divide by the scale factor. Think carefully about how you do your fixed-point representation! Many choices will work, but if you are clever you can make things much faster.
- Pay attention to scale factors! You can only add or subtract values that have been scaled by the same scale factor. What about multiplication?
- Careful choices in constant values required can save time in calculations also.

6 Efficiency Challenge

Part of the task is to make your assembly language program SMALL and EFFICIENT. If you can write your program in 70 or fewer executable instructions, you will get a bonus of 5 marks.

Since filling delay slots with instructions other than `nop` is NOT a requirement in your assignments, you need to count the executable instructions as they would be WITHOUT filling delay slots with instructions other than `nop`, even if you did so. Note that `nop` IS an executable instruction.

So, the code sequence

```
call  writeChar
mov   10, %o0            ! filled delay slot
```

counts as THREE executable instructions, i.e. the same as using a `nop` instruction like this:

```
mov   10, %o0
call  writeChar
nop
```

To get the bonus, your program must work and follow all the implementation specifications.

Important: Please put the number of executable instructions in the header comment at the top of your program if you want it to be considered for the bonus marks.

7 What to Submit for Part 1

1. Submit your code in a file `asn2.m` within your `asn2` directory
2. Just like the directory name, the file should be named `asn2.m` and **NOT** `Asn2.m`, `Assign2.m`, `Assignment2.m`, or any other variation. We used automated scripts to assemble your programs and they will be expecting a file named `asn2.m`. Failing to adhere to the naming convention dictated here will result in a deduction of a few marks.

Part II

Using the Debugger: gdb (25 Marks)

It can be very difficult to find bugs in an assembly language program. The debugger is a tool that allows you to step through your program, to stop and start execution of the program, and to examine the contents of registers. The debugger gdb is not a very user-friendly tool, but it should show you exactly what the computer is doing when it runs your program.

This part of the assignment consists of three sessions, to give you practice in single-stepping through code and in setting breakpoints. You may be able to see what the programs are doing without using gdb, but learning how to use the debugger can help you locate bugs in the more complex programs that you will write later in this course.

You should have a copy of the gdb lab notes handy when you do this part of the assignment. Also, gdb will be discussed in Lab 4.

8 The Mystery Program (10 marks)

The purpose of this exercise is to have you single-step through a program, displaying the contents of significant registers at significant places in the program, in order to find out what the program is doing. Note that running this program does NOT produce any output on the screen! To see what it does, you'll have to use gdb to examine the contents of registers as it executes.

1. Download the file `mystery.s` from the CS 2208 Assignment 2 page. Copy it to your `asn2` directory within your Git repository.

2. Open a script file to capture your gdb session, and then start up gdb, as described below.

3. Use the following commands at the start of your gdb run (unless you already have them in your `.gdbinit` file, as explained on page 10 of the gdb notes):

   ```
   display/i $pc
   break main
   run
   ```

4. When gdb stops at the breakpoint (note that it is actually at `main+4` rather than at `main`), start single stepping through the code in `mystery.s`. Use the gdb command `si` or `ni` to single-step through the instructions. To make single-stepping easier, hit Enter to repeat the previous command to gdb.

   Note that the `set` instruction in the code actually assembles to **two** basic instructions, `sethi` and `or`. See the lecture notes, Topic 3C for an explanation.

5. As you single step, display the contents of registers `%l0`, `%l1` and/or `%l2` as often as you wish, but AT LEAST at the points in the program commented by `checkpoint 1` to `checkpoint 5`.

   Recall that when an instruction is displayed by gdb, it has not yet been executed – gdb pauses right before the instruction is executed. To help you see what is happening in the registers, it is more helpful to display their contents in hex rather than in decimal. To do this, use the gdb command `print/x` or `p/x`. Note that you must use the `$` rather than `%` in giving the register name (e.g. `$l2` rather than `%l2`).

6. Keep single stepping until you reach the instruction `ret`. At this point, use the gdb command `cont` (or `c`) which will cause the program to exit. You can also use `ni` until the program exits; however, do not use `si`, since this will single-step into system exit routines and mass confusion will ensue.

7. Quit gdb by using the gdb command `quit`.

8.1 What to submit for Mystery Session 1

1. For this part of the assignment, create a script file `mystery1.script` that shows a screen capture of your gdb session. Type the following commands:

   ```
   script mystery1.out
   pwd                              # show the path to your assignment
   date                             # show the date/time of your run
   whoami                           # show your user name
   ls -lt                           # list your assignment directory before assembly
   gcc -o mystery mystery.s         # assemble mystery.s to the executable mystery
   ls -lt                           # list your assignment directory after assembly
   gdb mystery                      # start gdb
    ...   {enter your gdb commands here}
   exit

   scriptfix mystery1.out > mystery1.script   # Clean up your script file
   rm mystery1.out
   ```

   Don't forget to run `scriptfix` on your script, as shown above. Please delete `mystery1.out` before committing your code (also shown above).

   Be sure to include the clean version of `mystery1.script` in your `asn2` directory.

2. Create a text file `mystery1.txt` in your `asn2` directory that describes **in one sentence** what you think the program accomplishes in general. Note that the *input* to the program is just a SAMPLE bit pattern; it is NOT meant to be anything specific. Be concise and to the point in describing the overall functionality of the program. Do NOT just say what each instruction is doing!

9 The Mystery Program with Breakpoints (5 marks)

To get some practice with using breakpoints rather than single-stepping through a program, run `mystery.s` in gdb again just as you did in Session 1, but, this time, when gdb stops at the breakpoint at `main+4`, set breakpoints at the labels `ckpt1`, `ckpt2`, etc. and give the gdb command ccrun.

When gdb stops at a breakpoint, print the contents of the registers as before. Then continue execution using the gdb command `cont` (or `c`).

There is no sleuthing involved in Mystery Session 2: you're just practicing using breakpoints. Breakpoints are very useful when debugging as they allow us to stop execution of the program at specific points, allowing us to narrow down the location of a bug. Using breakpoints is typically much faster than stepping line-by-line throughout an entire program.

9.1 What to submit for Mystery Session 2

1. A clean (`scriptfix`'d) script file `mystery2.script` should be included in your `asn2` directory.

10 Gdb Session 3: Program with Bug (10 marks)

The purpose of this exercise is to have you see the bug in a program by using gdb to observe what is happening in the program.

1. Download the file `bug.m` from the Assignment 2 webpage. The program is intended to the sum of squares for 0,1,2,3,4,5. It will print out each square, and then the sum of all 6 squares. Assemble and run it (remember to link it with `iofunc.o`). What happens?
2. Clearly, `bug.m` has a bug! You must identify the bug in it using gdb (even if you already recognize what it is!). Start up the program in gdb, and step through it to observe what is happening.
3. Identify the bug in the program and create a script of a gdb session which shows the bug. You will submit a brief explanation of what the bug is, what is happening in the program, and why. This should only be no more than one paragraph in length.

10.1 What to submit for the Bug Session

1. A clean (`script fix`'d) script file `bug.script` should be in your `asn2` directory.
2. A text file `bug.txt` should be in your `asn2` directory, containing your explanation of the bug.

Part III
Submission (0 to -10 Marks)

Submitting your assignment correctly the first time saves all of us time. Failure to following correct procedure and naming guidelines will result in up to 10 marks being deducted from your grade. Your "asn2" directory should contain the following files, as described above:

```
asn2.m
mystery1.script
mystery1.txt
mystery2.script
bug.script
bug.txt
```

You will be submitting your assignment as described in Lab 3 (Part 3 - Section 7). You **MUST** have completed the repository setup steps required in Part 1 before you can submit. Once you have completed the above files to your satisfaction, and you are ready to submit, run the following commands required to submit your assignment:

```
cd ~/your/CS2208assignment/directory/asn2
git add .
git commit -m "Assignment 2 final commit before submission"
git push
git tag -a asn2 -m "Assignment 2 submission"
git push --tags
```

If you need to resubmit, or would like more detail on the submission process see Lab 3 (Part 3). Resubmitting before the Assignment deadline will not result in lost marks.