1 Overview

The purpose of assignment series 4 is to exercise your understanding of architecture and assembly to optimize the performance of a benchmark program: rendering an animated movie to the screen.

1.1 Instructions

- For this assignment, you can work in groups of 2.
- Read this entire document before you start.
- Your must submit a compressed tarball\(^1\), named after your last name and student ID, containing the files you have produced during the assignment, including your write ups to open questions.

- Your submission must be sent by e-mail before the deadline, at the e-mail address given by the assistants. Do not send your submission to the mailing list!

\(^1\)A compressed tarball is created with `tar -czf xxxx.tgz ....`
1.2 Prerequisites

You will need the following:

- the Alpha an MIPS cross-utilities and cross-compilers;
- the MGSim simulator compiled for Alpha, as per assignment 1, and optionally the MGSim simulator compiled for MIPS using your ISA implementation from assignment 2;

**Note**

You need a version of MGSim that was updated to support graphical output with 1 bit per pixel. You can obtain this from https://github.com/knz/mgsim/tree/ca2014 or, if you already have a working source tree, pull the latest patch from this tree:

```
git pull https://github.com/knz/mgsim.git ca2014
```

**Tip**

Preferably your MGSim build should be configured with SDL enabled so that you can view the graphical output. **If SDL is not enabled, it is still possible to complete the assignment, simply the graphical output will not be visible.**

- a copy of the following files, which should accompany this document:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>test.c</td>
<td>Example/test program.</td>
</tr>
<tr>
<td>view.c</td>
<td>Example/test program.</td>
</tr>
<tr>
<td>view-fast.c</td>
<td>Example/test program.</td>
</tr>
<tr>
<td>animate.c</td>
<td>Example/test program.</td>
</tr>
<tr>
<td>assignment.c</td>
<td>Skeleton for your assignment.</td>
</tr>
<tr>
<td>example.c</td>
<td>Reference program.</td>
</tr>
<tr>
<td>Makefile</td>
<td>Build rules.</td>
</tr>
<tr>
<td>alpha-cc</td>
<td>Script to invoke the Alpha cross-compiler.</td>
</tr>
<tr>
<td>mipsel-cc</td>
<td>Script to invoke the MIPS cross-compiler.</td>
</tr>
<tr>
<td>minisim.ini</td>
<td>Configuration file for MGSim.</td>
</tr>
<tr>
<td>include/framework.h</td>
<td>API header: benchmarking utilities</td>
</tr>
<tr>
<td>include/pbm.h</td>
<td>API header: PBM image format manipulation</td>
</tr>
<tr>
<td>include/gfx.h</td>
<td>API header: graphical output</td>
</tr>
<tr>
<td>include/output.h</td>
<td>API header: text output</td>
</tr>
<tr>
<td>include/abort.h</td>
<td>API header: simulation control</td>
</tr>
<tr>
<td>include/mtconf.h</td>
<td>API header: machine configuration</td>
</tr>
</tbody>
</table>

... continued on next page
1.3 Machine configuration

The architecture configured in \texttt{minisim.ini} has the following characteristics:

- 1 core running at 2GHz;
- 1KiB 2-way associative I-cache;
- 4KiB 2-way associative D-cache;
- shared memory bus clocked at 800MHz;
- 1 DDR module connected to the shared memory bus, clocked at 800MHz;
- all cores connected using dedicated I/O to a shared I/O bus clocked at 200MHz;
- ROMs and graphical output connected to the shared I/O bus.

Note how the ROMs and I/O devices are 10 times slower than the cores, and the DDR memory is 2.5x slower.

1.4 First steps - getting to know the environment

For this part you should not dedicate more than 1-2 hours of work.

1. run \texttt{make alpha}.
   
   This should build the example/reference programs using the Alpha-cross compiler.

2. run the program \texttt{test.alpha-bin} using MGSim for Alpha:
   
   \texttt{mtalpha-mgsim -c minisim.ini test.alpha-bin}

   As a result you should see an output like the following:

   `### random seed: 1416786914  
   Maximum supported output size: 1280x800  
   connected to I/O, 16 devices, 8 notification channels. [+1560]  
   smc at 0x7e000000, enumerates 15 devices. [+605]  
   I/O enumeration data at 0x242e0. [+1970]  
   uart at 0x74000000. [+1714]  
   rpc interface at 0x75000000. [+855]  
   lcd at 0x76000000, size 80x25. [+571]  
   lcd at 0x77000000, size 16x2. [+391]  
   rtc at 0x78000000. [+391]  
   gfxctl at 0x79000000, framebuffer at 0x7a000000. [+774]  
   extra ROM 0 at 0x7b000000. [+894]  
   argument ROM at 0x7c000000. [+658]  
   configuration ROM at 0x7d000000. [+700]`

   3
gfx: warning: resolution changed to 100x100
gfx: warning: resolution changed to 100x100
gfx: warning: resolution changed to 100x100
gfx: warning: resolution changed to 100x100
gfx: warning: resolution changed to 100x100
cpu0.action: Program requested simulator to exit with code 42.

Additionally, while the program is running, you may see a graphical screen opening and displaying a white diagonal line on a black background, five times.

Using the MGSim interface, count the number of core cycles and number of instructions needed to run this program. Note these numbers in your report.

3. Read the program test.c; you may want to open the files include/gfx.h and include/time.h which contain comments that document the APIs. Then using alpha-linux-gnu-objdump, analyze the assembly code for test.alpha-bin.

Then in your report:

- relate each instruction (or group of instructions) in the function do_test to the corresponding line(s) of source code in test.c;
- relate each instruction (or group of instructions) in the function gfx_putpixel to the corresponding line(s) of source code in lib/mtgfx.c.

4. The function do_test() is relatively inefficient, because gfx_putpixel() will test its arguments every time it is called.

Create a new file test-simple.c containing the following main function:

```c
int main(void)
{
    sys_detect_devs();
gfx_init();
gfx_resize(100, 100, 1, 0);
gfx_clear();

    // ... YOUR CODE HERE ...
    for (i = 0; i < 100; ++i)
        // ... YOUR CODE HERE ...

    return 42;
}
```

Then fill in the blanks to draw a white diagonal line from top left to bottom right, like do_test(1) did, but without all the checks performed by gfx_putpixel. For this you should use gfx_get_fb_address() (see gfx.h).

Finally, add the name of the file into the Makefile (on the line with PROGRAMS = ...). Then run make test-simple.alpha-bin.
5. Using the MGSim interface, count the number of core cycles and number of instructions needed to run your improved program.

**Note**
MGSim simulates multiple architectural clocks: one for the cores, one for memory, one for I/O, etc. The "master cycle counter" in the final simulation output is a pseudo-clock used to synchronize all the architectural clocks. Do not consider this in your work; instead focus on the "core cycle counter" which reports the number of clock cycles from the perspective of the processor cores.

6. Draw a block diagram of the architecture in your report. You can aid yourself by looking at MGSim’s topology output (mgsim -T arch.dot ...), however your diagram should be well-organized.

### 1.5 Graded assignment part A - predictions & measurements

For this part you should not spend more than 1-2 hours work.

1. Read the source code for the programs `view.c` and `view-fast.c`.
   For both programs, make an estimate of:
   - how many memory loads are performed in the main display loop;
   - how many memory stores are performed in the main display loop.
   Write your estimates in your report. Based on your estimates, predict how much faster the program `view-fast` should be compared to `view`.

2. Run the programs `view` and `view-fast` as follows:

   ```
   mtalpha-mgsim -c minisim.ini view.alpha-bin -L 1 images/avatar.pbm
   mtalpha-mgsim -c minisim.ini view-fast.alpha-bin -L 1 images/avatar.pbm
   ```

   **Tip**
   If SDL output is enabled, you can make MGSim run slightly faster by pressing the down arrow on your keyboard multiple times while the focus is on the graphical display).

   Using the MGSim statistics at the end of execution, measure how many core cycles have elapsed to run the programs and how many memory operations were performed.

   Are the measurements compatible with your predictions? Why?

3. Run the programs again with a larger input image, for example `powerdeer.pbm`.
   Again, measure how many core cycles have elapsed to run the programs and how many memory operations were performed.

   Is the prediction better or worse than for the run at step #2 above? Why?
1.6 Graded assignment part B - performance optimization

1. Read the code for animate.c. Notice how the program is structured: some initialization, then an “infinite” loop which displays all the frames of the input file then terminates.

Then read the code for example.c, and compare it with animate.c. Notice how this new example only displays the first 10 frames of the input, in a loop so that each frame is displayed 10 times in total.

Based on your measurements at step #3 in the previous section, make a prediction of how many cycles and instructions are needed to print each step in the animation powerdeer.pbm.

2. Try the example program out:

```
mtalpha-mgsim -c minisim.ini animate.alpha-bin \
  -L 1 images/powerdeer-small.pbm
mtalpha-mgsim -c minisim.ini example.alpha-bin \
  -L 1 images/powerdeer-small.pbm
```

**Note**
The parameter -L 1 configures a new ROM in the architecture, fills it with the contents of the file specified on the right, then connect the ROM to the core.

Since the program may take a long time to run, simply wait for a couple of frames to be displayed then press Ctrl+C to interrupt the execution. The statistics will still be printed.

Are the measurements compatible with your prediction? Why?

3. Open the file assignment.c. Inside this file, write your own implementation that successfully renders a 100-frame animation using less core cycles on average than the implementation in example.c.

For every optimization you choose to implement, describe in your report what is its main principle and how much performance is gained in this way.

If you implement multiple optimizations, try to isolate each optimization and quantify the performance improvement separately for each optimization.
Tip
There are multiple directions you can explore:

- you can write some code in assembly in a separate file and modify the Makefile to include it in libXXX.a.
- it is OK perform some extra work during the display of the first frame if it gains you time on average over the 99 remaining frames.
- for example, try copying the data to RAM before displaying.
- for example, try "packing" the data so that pbm_read_header does not need to be called for every frame.
- for example, try "compressing" the input data so that you need to read less data from DDR.
- for example, try "compressing" the rendering so that you do not need to update all pixels on the screen for each successive frame.

You can make the following assumptions in your work:

- the animations fit a screen of 640x480 pixels, ie. a buffer of 40KiB.
- the animation widths are multiple of 8 pixels.

Tip
There are several example animations in the sub-directory images. To generate your own animation input file, proceed as follows:

1. download an imaged GIF with your desired animation.
2. compute a reasonable size (width, height) in pixels that fits the assumption and so that your simulation will not be too slow.
3. using the program convert from ImageMagick, convert your GIF animation to PBM:

   ```
   convert yourfile.gif \
   -coalesce -monochrome \
   -resize WxH! yourfile.pbm
   echo "end" >>yourfile.pbm
   ```

1.7 “Expert” extra assignments

For extra credits you can investigate the following, in no particular order:

A. ensure and demonstrate that your optimized program also works using your MIPS ISA simulator from assignment 2.

B. extend your solution to support color animations.
1.8 Grading

- 3 points if you have submitted an archive in the right format and your report demonstrates that you have performed the steps in the section “First steps” above successfully.

- +2 points if you have completed part A successfully;

- +1 points if you have achieved a faster implementation in part B than the reference example program;

- +1 if your optimizations for part B are explained in your report with the measurements that prove they were useful.

- +0 to +1.5 points attributed depending on the quality of your discussion responses for part A and B in your report.

- +0 to +1.5 points attributed if you have achieved either or both of the expert assignments.

- +1 point if the performance of your implementation falls in the top-10 of the entire class.

1.9 Related articles

The following articles may provide you with additional inspiration:
