Computation II - 5EIB0
mMIPS lab 1: adding a custom instruction

The luminance of an individual pixel in an image is represented by a single byte, thus a value between 0 and 255. However, when performing calculations on these pixels, the result might be outside the range of a single byte. This means that the output of our filter should be "clipped" to the [0,255] range; values larger than 255 should be stored as 255 and values smaller than 0 should be stored as 0. This is handled with an if-statement in the C code, resulting in many processor instructions.

In this lab, you will add an custom instruction to the mMIPS instruction set that handles clipping in hardware with a single instruction. You will use LCC to compile the algorithm and create a reference output for the filter. Then you will add an R-type instruction to the mMIPS instruction set and modify some modules, including the ALU. Afterwards you determine the performance gain due to the reduction in the number of cycles needed to complete the filter.

1 Creating a reference output

In order to check if your modifications result in an identical output image, you are going to create a reference output using the original mMIPS.

1. Open a terminal and use the command `cd` to change to the `mmips/memory` directory. Make sure that this directory contains the file `image.c`.

2. Use LCC to compile the algorithm and output the binary to `mips_mem.bin`. This can be done using the following command: `lcc -o mips_mem.bin image.c`

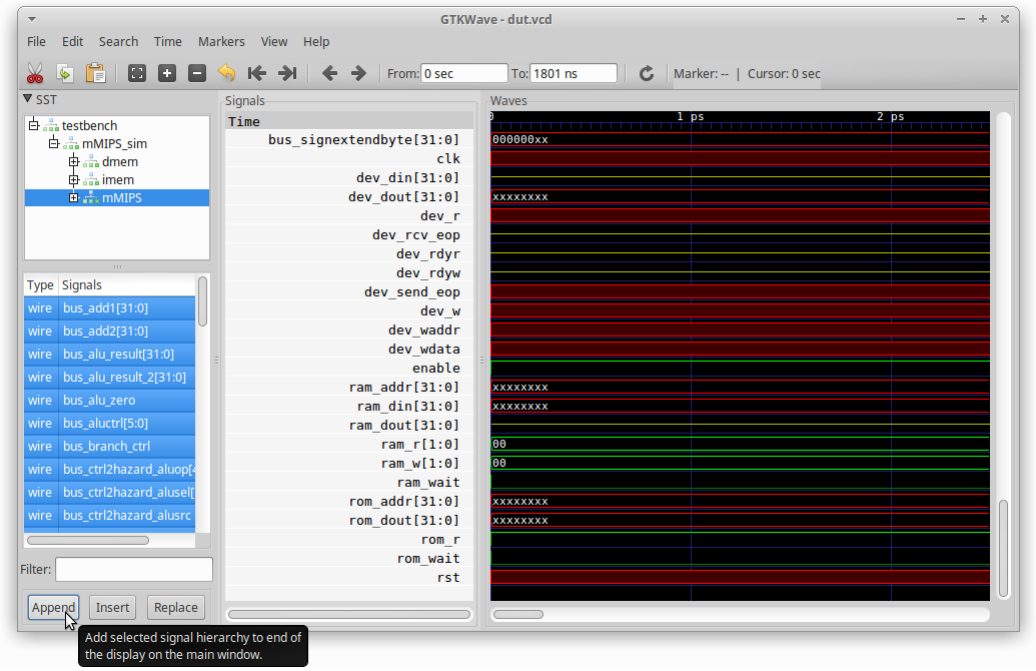
3. In order to move the binary into the ROM and RAM memory files, you first need to create those files. Run the `memory.py` script using the command "`./memory.py`" and then execute the command `create`. The script creates two new directories (ROM and RAM) inside the current directory. These directories contain the memory memory files each representing chunk of 2KB of memory.

4. Copy the content of the binary `mips_mem.bin` to the ROM (and global variables to the RAM) using the command `load lcc mips_mem.bin`. Verify that the script worked by opening the first ROM memory file. This file should not be filled with zeros at this point.

5. Next you are going to load a 32x32 pixels section of an image in the RAM at address 0x401000. Because the RAM module starts at 0x400000, this means that the address relative to the start of the RAM is 0x1000. In this lab, you will use the image `bicyle.y`, but the process is similar for the other images. To move the image into the RAM memory files, execute the command `load_img bicycle.y 0x1000` in the script `memory.py`.

6. Use the command `make dut` to create the Design Under Test (DUT) simulation executable. The command achieves this using the open source Icarus Verilog tool. Once the simulation executable has been successfully generated the simulation can be run using the command `make sim`. This might take a few minutes. The simulation is finished, when it halts on a line with `$finish`. This means the program has reached its end (the main function returned) and a dump of the RAM memory the way it was when the program ended, has been written to the `ram_mips.dump.hex` file.

7. When the simulation has finished, note/write down/store the number of cycles it took to finish the code as shown in the command line console (this is needed to calculate the performance gain of your optimizations).



8. Use the command `make view` to view the output signal waveforms from the simulation. This command uses the open source GTKWave tool to display the waveforms. All of the signals can be added to the waveform window by selecting mMIPS in the design hierarchy in the top left window, followed by selecting all of the signals in the lower left window (`Ctrl-A`) and then clicking on Append.

9. The wave window now shows what the signal values were at any time during the simulation. Explore the simulation output to find the point in time that the simulated mMIPS comes out of reset (the rst signal). After this you can close GTKWave.

10. A file `mips_ram.dump.hex` has been created in the directory `./memory/ram`. Copy this file to `./memory` for future reference (and maybe rename it to something memorable like `mips_ram.ref.hex`).

11. You can extract and view the result by running the `memory.py` tool and executing the command `store_img out.y bicycle.y`. Using this command an area of 32x32 pixels in the upper-left corner of the image `bicycle.y` has been replaced with the filtered output. You can view the image by converting it to PNG using the `ImProc` program.

2 Adding a custom instruction to image.c

Every pixel in the (grayscale) image is represented by a single byte. This means that every pixel has a value between 0 (black) and 255 (white). Because the filter can output values outside of this range, we need to make sure the values are clipped to their boundaries. That is, a value larger than 255 should be represented as 255 and a value smaller than 0 should be represented as 0. In `image.c` the following lines take care of this clipping:

```
if(result<0) buf_o[a * WIDTH + b] = 0;
else if (result > 255) buf_o[a * WIDTH + b] = (char)255;
else buf_o[a * WIDTH + b] = result;
```

When compiled using LCC, this C code is converted to many MIPS processor instructions. Hence, it will take the mMIPS numerous cycles to execute this program. We want to speed up the mMIPS by adding a special instruction that performs the clipping operation in hardware.

1. Disassemble the binary code by executing the command `disas mips_mem.bin` in a terminal (in the memory directory). You can prevent that all data is output on the terminal using the command `disas mips_mem.bin | less` Alternatively you can output the disassembled program to a text file using the command: `disas mips_mem.bin > mips_mem.asm`

2. Look at the assembly and try to find the instructions used for the clipping. How many instructions are needed? If we had used a single dedicated instruction, how much (in cycles) do you think would be the performance gain?

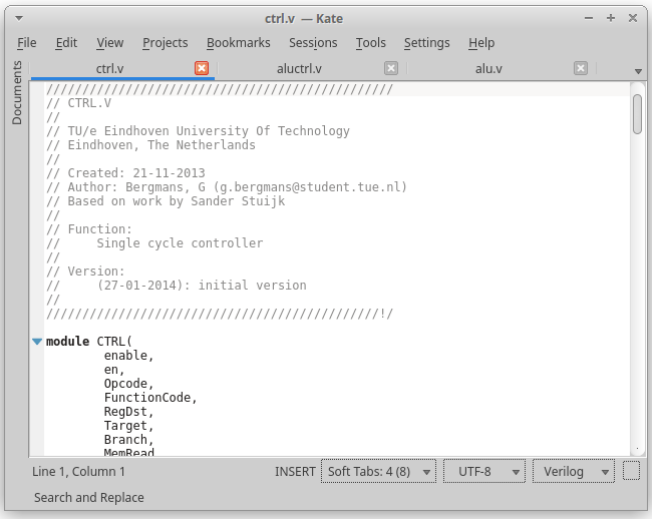
3. As can be read in the introduction pdf, LCC already comes with 4 constructs that map to custom operations in LCC. We are going to use the construct `((a) + ((b) + *(int *) 0x12344321))`, which maps on an R-type instruction with function code 0x30. Replace the clipping code in `image.c` with this construct, using `result` for register a and a constant of 255 for register b. (Note: this constant 255 has to be an integer variable that has 255 assigned to it, otherwise the magic construct will not be recognised by LCC!)

4. Compile the new program and disassemble to check if the new output has been used. You should find an instruction with function code 0x30 at about the same lines where the clipping code was before. This assembly instruction should have a name that is not part of the usual mMIPS instruction set.

5. Load the file in the ROM by using the `memory.py` command `load lcc mips_clip.bin` (or whatever filename you stored the output of LCC). The image we already loaded should still be there.

3 Adding the clipping instruction to the mMIPS

In this exercise you are going to change the hardware in various modules of the mMIPS to make it compatible with the clipping instruction. To do this, you will add clipping hardware to the `alu` module.



1. Use the `kate` text editor to open the files `ctrl.v`, `aluctrl.v` and `alu.v`.
2. Look through the `ctrl.v` file and try to understand what happens. What does the `ctrl` module do? Is additional hardware really needed? Use the mMIPS schematic to see where the signals are going. (Hint: Does the clipping function fundamentally differ from other R-type instructions like add, and, or, etc.?)
3. Look at the `aluctrl.v` and `alu.v` files. The `alu` module performs a certain task (like subtracting) based on the `ctrl` value that it gets from the `aluctrl` module. The `aluctrl` module selects the `alu` function based on the type of instruction (e.g. Both the R-type and the immediate add use the same ALU add hardware). Because you are going to add a new function to the ALU, select an unused `aluctrl` code for our clipping function.
4. Change the `aluctrl` module in such a way, that it gives the clipping control code you chose to the ALU in case of your clipping instruction (R-type with function code `0x30`).
5. Add the clipping hardware to the `alu` module. Match the `ctrl` number with the number you used in the `aluctrl` module.

4 Verify the results

Try to simulate your new code and hardware (`make sim`). Resolve any errors you might get. If your hardware does not work, use the waveform viewer (`make view`) to observe what your design is doing incorrectly. To check whether the output is the same, use the following `memory.py` command:
`compare ./mips_ram.ref.hex ./ram/mips_ram.dump.ref`.

In how many cycles does your program finish? How much is the performance gain from the dedicated clipping instruction, expressed in the reduction of the number of cycles needed to complete the program?