SE201 - 2016 - PR2 **Assignment**

General Instructions

- You can download all required configuration, make and source files, that you will need for the following exercises, from the course website.
- Each group should submit a report, written in French or English, in the PDF file format via the course website.
- Each student has to be member of a group, where groups should generally consist of 3 students.

1 Setup

For the following exercises you will need to download an archive from the course website containing configuration, as well as make and source files for the exercises. After extracting the archive, you should find the following files in the `PR2` directory:

```
brandner@kairon:~/PR2> find . -type f
./src/bsort.c
./cfg/bsort_bp_2bit_saturation.cfg
./cfg/bsort_bp_1bit.cfg
./bin/updatecfgFiles.sh
./Makefile
```

- **`src/bsort.c`**
  C source code of the program that should be analyzed in the exercises. The source code is compiled, using the MIPS C compiler, into binary program (`bsort.bin`) suitable for simulation by the `openDLX` simulator.
- **`cfg/bsort_bp_2bit_saturation.cfg`**
  Configuration files for running the compiled binary program `bsort.bin` using the `openDLX` simulator.

  This file specifies that a MIPS processor with a direct-mapped instruction cache, a set-associative data cache, and a branch predictor using a 2-bit saturation counter should be emulated by `openDLX`.
- **`cfg/bsort_bp_1bit.cfg`** Configuration files for running the compiled binary program `bsort.bin` using the `openDLX` simulator.

  This file specifies that a MIPS processor with a direct-mapped instruction cache, a set-associative data cache, and a simple 1-bit branch predictor should be emulated by `openDLX`.

- **bin/updatecfgFiles.sh**
  A helper script needed to update the `.cfg` files whenever the binary program (`bsort.bin`) changes.
- **Makefile**
  Makefile to compile `bsort.c` and launch simulations using `openDLX`.

To get started, simply launch a terminal, go into the directory containing the `Makefile`, and type `make run` (you can use `make all` to simply compile the program without launching `openDLX`). This should build all required files automatically and launch the `openDLX` simulator for both configurations. After completion you should find the following files:

```
brandner@kairon:~/PR2> find . -type f
./src/bsort.c
./cfg/bsort_bp_2bit_saturation.cfg
./cfg/bsort_bp_1bit.cfg
./bin/updatecfgFiles.sh
./bin/bsort.elf
./bin/bsort.bin
./Makefile
./openDLX.log
./bsort_bp_2bit_saturation.log
./bsort_bp_1bit.log
```

- **bin/bsort.elf**
  This is the binary produced by the MIPS compiler in the ELF format, which cannot be processed by `openDLX`. It is thus converted from the ELF format into a raw binary, which gives the final binary `bsort.bin`.
- **bin/bsort.bin**
  This is the binary program executed by `openDLX`.
- **bsort_bp_2bit_saturation.log**
- **bsort_bp_1bit.log**
  Log files containing information on various events (stalls, flushes, cache accesses, branch predictions, ...) that occurred during each cycle of the simulation

  At the very end of the file you will also find statistics, such as the cache miss-rate, the branch predictor's misprediction rate, et cetera.
- **openDLX.log**
  This file is crated by `openDLX`, but usually remains empty.

You can remove all the generated files using the usual `make clean` command.

1.1 Using `openDLX` Interactively

`openDLX` also provides a graphical user interface, which visualizes the execution of the program (similar to pipeline diagrams that we saw in the lecture), allows single-stepping (i.e., simulating the program cycle by cycle), and allows to inspect the register and memory values.

Using `make run`, the simulator merely generated the log files and did not launch the graphical user interface.

In order to start `openDLX` interactively, simply open a terminal, and type the following command: `openDLX`. This should open an `openDLX` window . . . you are good to go.

2 Warm Up

Aims: *Get familiar with the MIPS tools and the `openDLX` interface.*

Just follow the instructions below for the various exercises. Since the goal of these exercises is to get familiar with the MIPS tools and the `openDLX` simulator, you do not have to write anything in the report.

2.1 MIPS Tools

- Compile `bsort.c` using `make` and check which commands are executed.

  Hint: if make tells you `make: Nothing to be done for 'all'`, either use `make -B` or `make clean all` – try both to see what the difference is.

- Try to find out what the compiler options `-nostdlib` and `-nostartfiles` do.

  Hint: Use `man gcc`.

- Look at the assembly code of the resulting ELF binary using the following command: `mips-linux-gnu-objdump -d bin/bsort.elf`

  The output of this command consists of three columns the addresses of the various instructions (on the left), the instructions' binary representation (as hexadecimal numbers), and the human readable assembly code.

  Hint: Using `mips-linux-gnu-objdump -d bin/bsort.elf | less` might be even more convenient.

- Look at the assembly code of the ELF binary using the command: `mips-linux-gnu-objdump -S bin/bsort.elf`, which in addition to the assembly code also shows the original C code. Find the assembly code corresponding to the first **`for`**-loop in function `main`.

- Look at the symbols defined in the ELF binary (functions and global variables) using the command: `mips-linux-gnu-objdump -t -j .data -j .text bin/bsort.elf`

  The output of this command consists of several columns (not all of them are interesting for us). The leftmost column shows the address of the symbols, while the rightmost shows the symbols' names.

  Try to find the address of symbol `_fdata` (probably it will be `0x111d0`). This symbol represents the variable `input` from the C code.

  Bonus Question: Can you find the instructions that load the address of `_fdata` into a register? Probably it will be loaded into register `v0` within the first **`for`**-loop of `main`.

2.2 openDLX

- Launch `openDLX` and load the configuration `bsort_bp_1bit.cfg`.

  Hint: Make sure the program is compiled (`make`), then load the configuration file via the menu *File · Run from configuration file*.

- Open the *memory* window using the mouse or the menu *Window · Display Memory*. Type the address of the symbol `_fdata` (`0x111d0`) into the field *start addr* and inspect the values of the array. Verify that the values match those from the C code.

- The values of the array `input` are copied by the first **`for`**-loop of `main` into another array `buf`, which is stored on the stack. Run the program up to the address of the store instruction within this loop (probably `0x1098`) using the menu *Simulator · Run to address X*.

  Try to determine the address to which the store instruction is writing. This address should be the address of array `buf` (probably `0xeffffd8`).

  Hint: Check the value of the register operand (*Window · Display Register Set*) used to compute the address of the store (probably `v0`). Note that the value of that register is not yet written to the register file, it is computed a few cycles later. Use single-stepping (*Simulator · Do cycle*) until the register value changes.

- Open the *memory* window again and watch how the values of array `buf` get sorted using the menu *Simulator · Run program slowly*. Are the values really sorted at the end?

- Reset the simulator, i.e., reload the configuration, using the menu *Simulator · Restart program* and execute the first `9` cycles using *Simulator · Do X Cycles*.

  Check the *log* window (*Window · Display Log*), where branch misprediction was signaled at the beginning of cycle `8`. Find out which instruction caused the misprediction in the *cycles and pipeline* window (*Window · Display Clock Cycle Diagram)*.

- Now open the *statistics* window (*Window · Display statistics*) and see how many instructions were fetched and how many were executed. Also, check the current statistics of the branch predictor and the caches.

- Finally, run the program to completion *Simulator · Run Program*. And check the statistics again.

3 Pipelining

Aims: *Deepen the understanding of the operation of a realistic pipelined processor.*

You now saw pretty much everything `openDLX` can do. In the next exercises you will use the various functions to explore the operation of the processor pipeline. Please include the answers to the following questions in the report.

- Reset or reload the configuration for the 1-bit branch predictor, execute the program for about 30 cycles. Can you tell from the pipeline diagram whether the simulated processor supports forwarding? Justify your answer.

  Hint: If it helps the explanation, you may include a screen-shot of the pipeline diagram (only the relevant parts, not the entire screen).

- When you look at the pipeline diagram, you will see quite a few instructions that do not complete their execution, i.e., the instructions do not reach the `WB` stage. Find examples in the pipeline diagram illustrating this situation (include pictures in the report). Distinguish at least two different scenarios and explain what happens for each of them. For instance, explain why the respective instruction did not finish its execution, i.e., was *aborted*.

  Hint: There is a cases where the aborted instruction restarts on the next cycle. So, in fact the instruction is not aborted, but something else happens.

- Have a close look at the way branches and jumps are handled. Can you find a difference with regard to the way these instructions are handled in comparison to the lecture?

  Hint: Recall that the simple processor (without branch prediction) discussed in the lecture did flush both instructions in the `IF` and `ID` stages.

4 Branch Prediction

Aims: *Understand the impact of branch prediction.*

The following exercises will focus on issues related to branch prediction. Please include the answers to the following questions in the report.

- Simulate both configurations (with the 1-bit and the 2-bit saturation counting branch predictors respectively), e.g., using `make run`. Then have a close look at the statistics regarding branch prediction in the log file. Which branch predictor is doing better? Justify your answer.

  Hint: Try to find the misprediction rates as well as the number of executed/fetched instructions in the statistics and use these numbers as arguments.

- Have an even closer look at the statistics (to be precise, at the lines immediately following the misprediction rate in the statistics window starting with `bpc:` – *branch program counter*). Using this information you can see how the branch predictors perform for individual branches. What are the statistics telling you? Is the winning predictor from the previous question always better than the other?

- Have a look at the assembly (and C) code corresponding to the first two branches shown in the statistics (look again for the lines starting with `bpc:`). Can you characterize the behavior of these two branches, i.e., what is the typical branch direction for both of them? Based on this, explain the difference in the misprediction rates of the two branch predictors.

  Hint: Take the branch addresses (`bpc`) and search for that address in the output of `mips-linux-gnu-objdump` (use the `-S`).

- Restart the `openDLX` simulator using either of the two branch predictor configurations. Using the pipeline diagram and or the information in the log file, determine the penalties of taken/untaken branches when predicted correctly/mispredicted. Explain the difference to the way branch prediction was explained in the lecture.

  Hint: Search for lines starting with `INFO␣␣[BP_MODULE]:` in the log file.

5 Data Caches

Aims: *Understand the performance of data caches.*

The following exercises will focus on data caching (note that `openDLX` also simulates the instruction cache, but we will ignore it). Please include the answers to the following questions in the report.

- Simulate the configuration with the 1-bit branch predictor, e.g., using `make run`. Then, summarize the behavior of the data cache, e.g., give numbers related to the hit/miss rate, number of access, et cetera. What is the total size of the cache?

  Hint: The information is in the *statistics* window or at the end of the log file.

- Open the file `cfg/bsort_bp_1bit.cfg` in your favorite editor and edit the data cache configuration (starting at line 20). Rerun the simulation for increasing numbers of cache lines (`dcache_line_number`): 4, 8, 16, 32, and 64. How does the size of the cache develop and how does the cache hit/miss rate change? Which configuration performs best, with regard to the cache size.

- Open the file `cfg/bsort_bp_1bit.cfg` in your favorite editor and edit the data cache configuration (starting at line 20). Rerun the simulation for increasing associativity (`dcache_associativity`) 2, 4, and 8, using the replacement policies (`dcache_replacement_policy`) `FIFO` and `LRU`. Set the number of lines to 8. How does the hit/miss rate compare to the direct-mapped cache with the same total size from before?

- Open again the file `cfg/bsort_bp_1bit.cfg` in your favorite editor and edit the data cache configuration. Rerun the simulation for increasing line size (`dcache_line_size`) 4, 8, 16, and 32, using a 4-way set-associative cache with LRU replacement. Set the number of lines to 8. How does the hit/miss rate compare to the caches with the same total size from above?

- Have a look at the C code of the program. How is data accessed by the program? Is data reused frequently? Which kind of cache miss (compulsory, capacity, conflict) do you believe dominates the cache's performance for this (small) program? How much data is actually accessed by the program?

  Hint: Closely look at the array accesses and the value of `SIZE` defined at the top of the source code.

- Which of the explored cache configurations performed the best? Which cache configuration gave the best trade-off between hit/miss rate and total cache size? Justify your answers using the numbers you collected before. Which parameter appears to be the most relevant? Relate this observation to the way data is accessed – as discussed in the previous exercise regarding the C code.