Due date: Friday December 7th 11:59:59pm EST. This is a hard deadline: To comply with MIT rules, we cannot allow the use of late days.

Getting started: To create your initial Design Project repository, please visit the repository creation page at https://6004.mit.edu/web/fall18/user/project. Once your repository is created, you can clone it into your VM by running:

```
git clone git@github.mit.edu:6004-fall18/projects-project-{YourMITUsername}.git project
```

Turning in the project: To turn in this project, commit and push the changes you made to your git repository. After pushing, check the course website to verify that your submission passes the tests for the points you think you should get. If you finish the project in time but forget to push, you will not get credit for the lab. For this reason we recommend that you push early and push often.

There is no required portion for the project, it is up to you to decide how much you wish to complete.

Check-off meeting: There will not be a check-off meeting for the project.

Design Project

You’ve been hired as a systems engineer at a startup that designs RISC-V processors and low-level software for sensor networks. Their systems spend most of the time sorting data, and are limited by its performance. Therefore, your job is to improve the performance of sorting across different systems. This startup sells several systems that demand different optimizations. For this reason, you’ll have to optimize either software (part 1), hardware (part 2), or both software and hardware at the same time (part 3).

You will gain more points by having your code or processor run faster. But your code and processors need to behave correctly—a fast but incorrect implementation will earn no points.

This project is designed such that it is progressively harder to get more points. For each of the first two sections, it should be fairly straightforward to get 4 out of 7 points, but the final 3 points may require some serious effort.

This project lets you reuse code from previous labs. If you have done a good job optimizing your designs in previous labs, you will earn some points simply by importing them.

This project is open-ended. We will give you hints on how to improve performance in each section and in lectures 22 and 23, but you are allowed and encouraged to try other techniques!

Coding guidelines: You can change any existing bsv files, sw/lab4_opt/sort.S and sw/sort/sort.S. Modifications to other existing files will be overwritten during didit grading. However, you may add new files if you want.

Processor optimization guidelines: Before attempting each of the sections, please read Appendix A, which describes how to optimize your processor by measuring and understanding the software, reducing control and data hazards, and improving the clock period.

Processor debugging guidelines: If your processor does not work as expected, please read the Appendix B, which describes both general debugging strategies and shows how to use an optional pipeline visualization aid.
1 Optimizing Software (7 points)

The startup is currently selling a system that uses a simple, single-cycle RISC-V processor. Your first assignment is to improve the performance of a sorting program on this processor. Since all instructions take a single cycle, all you can do to accelerate sorting is reduce the number of instructions it takes.

You should start from your quicksort implementation from Lab 4. Please optimize it to reduce the number of executed instructions. There are no restrictions on the optimizations you can make—you could even use a different sorting algorithm. However, note that a well-optimized quicksort will earn all the points.

Please cd into sw/lab4_opt, which will be your working directory for Section 1. Start by copying your code from Lab 4 into the sw/lab4_opt/sort.S file. You will be using similar infrastructure as Lab 4.

Run make in sw/lab4_opt folder to compile your sorting software. This will create two software targets:

- sort_test.vmh: The Lab 4 testing code, which checks the correctness of your sort code.
- sort_bench.vmh: The benchmark software, which measures the performance of your sort code.

We use separate programs for testing and benchmarking because the testing program spends many instructions checking the correctness of the result. By contrast, the benchmark program is only calling the sort routine.

To evaluate your sorting software, you can run the following commands:

- python sim.py sort_test to check the correctness of your sort assembly code.
- python sim_gui.pyw to use the graphical user interface for debugging.
- python sim.py sort_bench to measure the instruction count of your sort code.

Please note that python sim.py <program> will also give other instruction-level statistics, which will be helpful for Sections 2 and 3.

Score thresholds: Your score for Section 1 depends on the instruction count of the sort_bench programs, as follows:

0 Points \iff Instruction Count \gt 145000
1 Point \iff Instruction Count \in (130000, 145000]
2 Points \iff Instruction Count \in (115000, 130000]
3 Points \iff Instruction Count \in (105000, 115000]
4 Points \iff Instruction Count \in (90000, 105000]
5 Points \iff Instruction Count \in (82000, 90000]
6 Points \iff Instruction Count \in (70000, 82000]
7 Points \iff Instruction Count \leq 70000

Your code must work correctly to earn any points. Earning all the points can be very challenging, so if you don’t see how to make further progress, we recommend that you attempt to earn points from the other sections.

You can run ../../didit_grade.sh 1 to see your current score for Section 1.

Hints

There are many ways to make your code faster. To get the most out of your changes, you would want to optimize the code that runs most frequently, e.g., the code inside a loop body. Here are a few potential avenues:

Iterate on addresses, not array indices: Your code may be keeping array indices in registers, then using the indices to compute the address of the element to be accessed at every iteration of the loop. It is more efficient to iterate on the addresses directly, without keeping track of the indices into the array.

Perform loop-invariant code motion: Your code may be calculating the same value on each iteration of a loop. Instead you can calculate the value before the loop. For more details, see https://en.wikipedia.org/wiki/Loop-invariant_code_motion.

Optimize call sites: Remember that each time you make a call you pay for both the call and return instruction as well as dealing with the stack. Avoid saving and restoring values from the stack unnecessarily.
(however, your code must obey the calling convention). You may want to inline the bodies of frequently-called functions into their callers to avoid function call overheads.

If you want to go beyond these suggestions, begin by reading https://en.wikipedia.org/wiki/Optimizing_compiler

2 Optimizing Hardware (7 points)

The startup has partnered with Google to design the hardware for one of Google’s products. You job is to optimize the RISC-V processor that will run on this system so that it performs sorting well. However, there are a couple of constraints that you cannot control. First, Google insists on using a standard quicksort implementation compiled with gcc -O2—they do not trust your assembly code. Second, the system is using memories that must be clocked at 1.82 GHz (i.e., a 550 ps clock period). Therefore, your processor must also use a 550 ps clock period—your processor cannot have a critical path longer than 550 ps. It may have have a shorter critical path, but that will not improve its performance.

These limitations leave you with one option: improving performance requires designing a processor that minimizes CPI (cycles per instruction), while keeping the critical path at or below 550 ps. (This is equivalent to maximizing IPC, or instructions per cycle.)

Please optimize your Lab 7 processor design. Your processor will execute a baseline sorting software, sort_base, which is comes from a quicksort implementation in C compiled with gcc -O2. The goal of this section is to reduce the cycle count of the processor when running sort_base.

ProcessorIPC.bsv is a three-stage pipelined processor with the same issues as in Exercise 3 of Lab 7.

Please first fix ProcessorIPC.bsv as you did in Lab 7. (You can copy most of your Lab 7’s design as-is, but please do not the overwrite ProcessorIPC.bsv with ThreeStage.bsv of Lab 7, which will delete the $display messages of ProcessorIPC.bsv that are used for auto-grading.)

Then, improve your processor to reduce the cycle count. Remember that you need to keep your clock period below 550ps.

Compile your processor by running make ProcessorIPC.

You can run a suite of tests on the processor by running ./test.sh. Please choose ProcessorIPC for this Section.

Passing fullasmtests (Option 6 in test.sh) is required for getting any grades in this Section.

Running sort_gcc_baseline (Option 13 in test.sh) will give you the cycle count and clock period of your processor running sort_base.

There are some test.sh flags you might find useful:

- You can silence the $display messages and only see PASSED or FAILED, cycle counts and processor clock period by running ./test.sh -s or ./test.sh --summary.
- You can skip running synth for showing the clock period by running ./test.sh -q or ./test.sh --quick.
- To look at other flags of test.sh, run ./test.sh -h. The flags of test.sh are additive.

Score thresholds: Your score for Section 2 depends on the cycle count that sort_gcc_baseline (Option 13 of test.sh) achieves on your processor, as follows:

- 0 Points \( \iff \) Cycle Count \( > 221354 \)
- 1 Point \( \iff \) Cycle Count \( \in (209704, 221354] \)
- 2 Points \( \iff \) Cycle Count \( \in (198054, 209704] \)
- 3 Points \( \iff \) Cycle Count \( \in (186404, 198054] \)
- 4 Points \( \iff \) Cycle Count \( \in (174753, 186404] \)
- 5 Points \( \iff \) Cycle Count \( \in (168928, 174753] \)
- 6 Points \( \iff \) Cycle Count \( \in (163103, 168928] \)
- 7 Points \( \iff \) Cycle Count \( \leq 163103 \)
These thresholds correspond to fractional improvements of 5%, 10%, 15%, 20%, 25%, 27.5%, and 30% over a reasonable Lab 7 baseline. Remember that earning any points requires passing fullasmtests and having a critical-path delay at or below 550 ps. Once again, getting all the the points can be quite challenging.

You can run ./didit_grade.sh 2 to see to see your current score for Section 2.

Hints

Please read Appendix A for guidelines on optimizing your processor. Note that some of the optimizations that you can apply, like branch prediction and bypass paths, will be explained in depth in Lectures 22 and 23, so you may want to wait to implement them.

3 Optimizing Hardware and Software (6 points)

Finally, the startup has chosen you to lead the design of their next-generation RISC-V processor and software stack. Your job is to optimize both the sorting program and the processor for the combination of both to be as fast as possible. Unlike in Section 2, you have no limitations: you can use your own sorting program and the memories can run at any clock period, so the frequency of the system will be determined by the critical path of your processor.

Your goal is to minimize the amount of time it takes your processor to run your sorting code, that is, the cycles to execute your code times the clock period of your design. How to achieve this is up to you—reducing the number of executed instructions, the processor’s CPI, and the processor’s cycle time will all help. However, you’ll probably find you’ll need to trade one metric for another to get the best overall performance.

The software code for Section 3 is sw/sort/sort.S.

The hardware code for Section 3 is ProcessorRuntime.bsv.

Please start by copying sort.S from sw/lab4_opt folder to the sw/sort folder and copying ProcessorIPC.bsv to ProcessorRuntime.bsv.

Compile your processor by running make ProcessorRuntime.

You can run a suite of tests on the processor by running ./test.sh. Please choose the ProcessorRuntime option for this Section.

Passing fullasmtests and sort_test (Option 6 and 12 in test.sh) is required for earning any points in this Section.

Running sort_benchmark (Option 14 in test.sh) will give you the run time (cycle count × clock period) of your processor on sort_bench.

Please use the flags in test.sh as described in Section 2 to test your design.

There will no points for failing fullasmtests or sort_test. Otherwise, your score of Section 3 will depends on the runtime of sort_benchmark running on ProcessorRuntime as the following.

Score thresholds: Your score for Section 3 depends on the run time of sort_gcc_baseline (Option 13 of test.sh) on your processor, as follows:

0 Points ⇐ Runtime > 70000 ns
1 Point ⇐ Runtime ∈ (65000 ns, 70000 ns]
2 Points ⇐ Runtime ∈ (60000 ns, 65000 ns]
3 Points ⇐ Runtime ∈ (55000 ns, 60000 ns]
4 Points ⇐ Runtime ∈ (50000 ns, 55000 ns]
5 Points ⇐ Runtime ∈ (45000 ns, 50000 ns]
6 Points ⇐ Runtime ≤ 45000 ns

To earn any points, your design must pass fullasmtests and sort_test. Earning all the points may be challenging.
We recommend that you work on the previous two sections first. It will be easier to get a good score in this section if you start from reasonably optimized designs (4 or more points in each of the first two parts).

You can run ./didit_grade.sh 3 to see your current score for Section 3.

Hints

Please read Appendix A for guidelines on optimizing your processor. Note that some of the optimizations that you can apply, like branch prediction and bypass paths, will be explained in depth in Lectures 22 and 23, so you may want to wait to implement them.

This is very dependent on your implementation, but one common thing is to look at how you are handling loads. If your processor takes multiple cycles to complete each load instruction, eliminating loads from your program can help.

Even if you have a highly-optimized sorting routine for Section 1, you may get better performance by reordering instructions to reduce or eliminate stalls due to data hazards. This may also affect which bypass paths are beneficial to add to the processor.

Finally, if you have some common tasks that take multiple instructions and you want to combine them, in this part you can add additional instructions to your processor to improve performance. This is not necessary to earn full credit, but it is an alternative approach to designing a highly-optimized processor.
Appendix

A Processor Optimization Guide

A.1 Profiling your software

Before implementing different optimizations, it is worthwhile to measure the sorting program and estimate how much benefit you could get for each processor optimization.

We have enhanced sim.py to produce statistics of various types of instructions:
- Total executed instructions
- Load instructions (lw)
- Jump instructions (jal jalr)
- Branch instructions (beq bne blt bge bltu bgeu)
- Branches taken

To measure your software, please go to the sw/sort directory, where the sorting code for Sections 2 and 3 resides.

Run make first, then run:
- python sim.py sort_base for Section 2.
- python sim.py sort_bench for Section 3.

As you optimize the code for Section 3, you may see that the breakdown of instructions changes, and this may affect the potential of different optimizations.

A.2 Dealing with Control Hazards

- Penalties of miss prediction for jump instructions (jal or jalr) and branches can be reduced by resolving them earlier, e.g., in the decode stage instead of in the execute stage. Depending on your critical path, you may want to resolve some instructions at decode (e.g., jumps) and some at execute (e.g., branches).
- The number of mispredictions can be reduced by implementing a branch target buffer (BTB). Specifically, you can use an n-entry BTB as a next-address predictor for your processor. Replace the pc + 4 prediction with a prediction from the BTB, and make sure to train the BTB in case of jumps and mispredicted branch instructions.

Hint: Your BTB should be made out of normal registers, not SRAM. To implement an array of normal registers, you can use a Vector of registers as seen in mkRFile2R1W in RFile.bsv and as you did on Lab 6 (for the LRU bits). The first parameter of Vector is the size of the vector. Experiment with the size of BTB to find the optimal number of entries to reduce mispredictions without hurting critical-path delay.

A.3 Dealing with Data Hazards

- Data hazards can be resolved by adding bypass paths from later pipeline stages to earlier stages. You can use the BypassFIFO module from Bluespec’s SpecialFIFOs package for data bypassing. An example instantiation of such a FIFO is as follows:

```python
import SpecialFIFOs::*;
...
module ...
   FIFO#(Word) bypassFifo <- mkBypassFIFO;
...
```

When bypassing a register file, a Scoreboard module of either mkScoreboard or mkBypassingScoreboard is also needed to remove data hazards appropriately. Please read the rule schedules of those modules.
in the comment of Scoreboard.bsv.

- For lw instructions, the three-stage pipelined processor needs an extra cycle in the execute stage, which stalls the processor for a cycle. To avoid this stall, you can make a 4-stage pipeline, and read the data loaded from memory in this fourth stage (which is typically called Writeback). Since the register file only has one write port and instructions must write to it in order, the register file should always be written from a single stage (the Writeback stage in this case). You may need to add one or more bypass paths for your 4-stage pipelined processor to achieve good performance.

A.4 Improving your clock period

Estimate where your critical path of your processor by running `synth Processor<IPC|Runtime>.bsv mkProcessor -l multisize -n`. Please note the `-n` flag will give you more information about the critical path by recovering the Bluespec variable names on the critical path. However, this flag may not always work. Please drop `-n` in case of error.

You can also estimate the critical path by (i) looking at the starting and ending points in the critical-path report, and (ii) synthesizing individual functions (e.g., decode and exec) by running `synth <BSVfile> <function name> -l multisize`.

Based on where your critical path is, you can then choose optimizations to reduce it. For example, you could use a faster adder, optimizing the ALU (you can use your implementation from Lab 2), optimizing the decoder, reducing unnecessary multiplexers, etc.

For example, if your design has an adder in its critical path, consider using a faster adder. The `+` operator in Bluespec produces a 32-bit adder that has a latency of about 250 ps—much better than a ripple-carry adder, but we can do better still. A Kogge-Stone adder ([https://en.wikipedia.org/wiki/Kogge-Stone_adder](https://en.wikipedia.org/wiki/Kogge-Stone_adder)) has a latency of about 150 ps and can be implemented in about 10 lines of Bluespec.

B Debugging Help

If your processor does not work as expected, there are some simple strategies you can follow to debug it. This appendix first discusses a general strategy to debug Bluespec circuits, then discusses a tool that’s specific to pipelined designs.

B.1 General Guidelines

If things don’t work as expected, start by adding `$display` statements to see what rules are being invoked and at which cycles. It helps to be systematic: we recommend that you first add `$display("[%d] <ruleName>", cycles);` at the top of each rule. Many times this is sufficient to understand what’s going wrong (e.g., if you forget to enqueue to a FIFO that’s read by a rule, you’ll see that the rule doesn’t fire at all or stops firing). Then, refine by adding more `$display` statements or more output to each statement.

B.2 Pipeline Visualization with ScheduleMonitor

This lab contains a ScheduleMonitor module that you can optionally use to obtain a visual representation of which pipeline rules are firing each cycle. This module is simulation-only and produces no actual hardware. For a fully pipelined processor with no data or control hazards or load instruction, this module may produce an output similar to the one below:

```
fetch
decode
  execute
F__
```
The names at the top are the names of each of the columns. These correspond to pipeline stages. The rows below correspond to what is happening in each clock cycle. The first row `F_` means in the first clock cycle only fetch fired. The fifth row `FDE` means in the fifth clock cycle all 3 pipeline stages fired concurrently.

There are four other letters that may appear as output from the ScheduleMonitor integrated with the provided initial code:

- `L` - An instruction is in LoadWait state of execute stage
- `x` - An instruction was killed in-place in the specified stage.
- `s` - An instruction stalled in the decode stage due to a data hazard.
- `R` - The execute stage fired and redirected the fetch stage due to a mispredicted next pc.

You can also use any other letters of your choice.

**Using ScheduleMonitor**

Using ScheduleMonitor is optional and requires adding some code to your processor.

The module constructor for ScheduleMonitor (mkScheduleMonitor) takes in a File object (either `stdout`, `stderr`, or an opened text file) and a vector of pipeline stage names. The order of names in this vector determines the order of the columns in the output. The code changes outlined below instantiate a ScheduleMonitor for a 3-stage pipeline that prints to `stdout`.

```plaintext
ScheduleMonitor monitor <- mkScheduleMonitor(stdout, vec("fetch", "decode", "execute"));
rule doFetch;
    // do rest of fetch
    monitor.record("fetch", "F");
endrule
rule doDecode;
    // do rest of decode
    if (...) 
        // not stalling
        monitor.record("decode", "D");
    else 
        // stalling
        monitor.record("decode", "s");
endrule
rule doExecute;
    // do rest of execute
    if (...) 
        // not redirecting
        monitor.record("execute", "E");
    else 
        // killed
        monitor.record("execute", "x");
end
endrule
rule doLoadWait;
    // do rest of loadwait
    monitor.record("execute", "L");
```

8
rule doRedirection;
    // do rest of redirection
    monitor.record("execute", "R");
endrule

The record method of ScheduleMonitor writes a character in the specified column of the pipeline schedule diagram. Typically the first letter of the pipeline stage is written in the column when the stage fires normally, but the above code uses some other letters to show special conditions.