Due date: Thursday May 16th 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/spring19/user/labs/project. Once your repository is created, you can clone it into your VM by running:

git clone git@github.mit.edu:6004-spring19/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 project. 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).

The goal of this project is to optimize both software and hardware to speedup a sorting program using various optimization techniques. Some important things to keep in mind while working on the design project:

— 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. Getting full points will be a very challenging, but hopefully rewarding experience.

— 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 23 and 24, but you are allowed and encouraged to try other techniques!

Coding guidelines: You can change any the following bsv files: ALU.bsv, BTB.bsv, Cache.bsv, CacheTypes.bsv, CAU.bsv, Decode.bsv, Execute.bsv, ProcessorIPC.bsv, ProcessorRuntime.bsv, RFile.bsv, Scoreboard.bsv. You can also edit sw/sort_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, but you must remember to add, commit, and push these new files.

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 Appendix B, which describes both general debugging strategies and shows how to use an optional pipeline visualization aid.

1 Optimizing Software (5 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 2. 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.

The data you will be sorting does have some known properties that can help you decide on which sorting algorithm to use and which optimizations to apply.

- the numbers are signed 32 bit integers
- the numbers are randomly distributed between -2,147,483,648 and 2,147,483,647, the full range for 32-bit signed integers
- the numbers are in a random order

Please cd into sw/sort_opt, which will be your working directory for Section 1. Start by copying your code from Lab 2 into the sw/sort_opt/sort.S file.

Run make in sw/sort_opt folder to compile your sorting software. This will create sort.vmh which will print out the number of executed instructions as well as if you passed the three tests. Instructions are not counted for the checking code.

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

- python sim.py sort to measure the instruction count of your sort code on example input.

Please note that python sim.py <program> will also give other instruction-level statistics, which will be helpful for Sections 2 and 3. Note: As is usually the case in the real world, you are given some example input to test on but you don’t have access to the exact input arrays that will be used for scoring. When you push your code to Didit, it will be run on different input arrays so you may see slightly different cycle counts.

For debugging help continue to use the infrastructure from Lab 2 and copy your code over after it’s working.

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

- 0 Points $\Leftarrow$ Instruction Count $> 400000$
- 1 Point $\Leftarrow$ Instruction Count $\in (350000, 400000]$
- 2 Points $\Leftarrow$ Instruction Count $\in (300000, 350000]$
- 3 Points $\Leftarrow$ Instruction Count $\in (240000, 300000]$
- 4 Points $\Leftarrow$ Instruction Count $\in (180000, 240000]$
- 5 Points $\Leftarrow$ Instruction Count $\leq 180000$

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 ../../grade.sh 1 to see an estimate of your current score for Section 1. You will need to check your Didit results to see your final score (see the note above).
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](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 externally). 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](https://en.wikipedia.org/wiki/Optimizing_compiler).

2 Optimizing Hardware (10 points)

The startup has partnered with Google to design the hardware for one of Google’s new products. Your 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 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 6 processor design. Your processor will execute a baseline sorting software, **sort_base**, which 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**.

Start by copying your Multicycle processor code from Lab 6 into **ProcessorIPC.bsv**. (You can copy most of your Lab 6 design as-is, but please do not overwrite **ProcessorIPC.bsv** with **MultiCycle.bsv** of Lab 6, as this will delete the $display messages of **ProcessorIPC.bsv** that are used for auto-grading.)

There are a few changes that will cause you to have some compiler errors. The first is the division of memory into 2 parts, one for instructions with a variable named `iMem`, and one for data with a variable named `dMem`. You should update your processor to replace all accesses to the old unified memory variable `mem` to one of these new variables.

The second set of compiler errors will be caused by a change in the types of the source and destination registers in the DecodedInst and ExecInst types. You will need to fix these by updating your processor to use the `Maybe#(RIndx)` type for the source and destination registers.

**Maybe Types:** The `Maybe` type is very similar to the struct we created for the `RDst` in Lab 6, in that it’s value is either valid or invalid. In order to check if it currently has a valid value, where in Lab 6 we would check `RDst.valid`, for a `Maybe` we call the built-in Bluespec function `isValid()` and pass it the `Maybe`
object. For example, `isValid(dInst.dst)` will tell you if a decoded instruction has a valid destination register. In order to extract the actual value of the `Maybe` object, where in Lab 6 we would’ve used `RDst.data`, we use the built-in Bluespec function `fromMaybe()`. This function takes 2 arguments, the first is the value that the variable will be if the `Maybe` object doesn’t have a valid value, and the second is the `Maybe` object we are getting the value of. For example, if we have `dInst.src1` that has a valid value, then `fromMaybe(0, dInst.src)` will return that value. However, if it does not have a valid value, then it would return the default value we gave it of 0.

There are two reasons that we are using this new type. First, we are giving you a fairly good decoder that depends on `Maybe`s (since optimizing a decoder can be unintuitive and is not the best use of your time). You are free to optimize it further if you wish. Second, the new types will be helpful when dealing with hazards when you pipeline your processor.

After this, copy the code from Lab 7 into `CAU.bsv`, `Cache.bsv` and `CacheTypes.bsv`. You can choose which cache to use. When copying your caches over, there are a few important changes you’ll need to make to your code. The first is in the `SRAM` module. It has a new interface, with two different request methods `rdReq` and `wrReq`. This change allows the `SRAM` to handle both a read request and a write request at the same time. You should check out these new methods and what arguments they take in `SRAM6004.bsv` and update your usage of the `SRAM` module accordingly.

The second change is that the helper functions for extracting the tag, index, and line offset from a memory address now reside in `CacheTypes.bsv` instead of `CacheHelpers.bsv`, so make sure you copy your implementations for them there.

Once you have your cache ported, improve your processor to reduce the cycle count. Remember that you must keep your clock period below 550 ps.

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 points 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`. If you see an error like: `(standard_in) 1: syntax error`, it may be caused by running out of memory in your VM. Try shutting down your VM and increasing the “Base memory” allocation in your VM settings (under the “System” tab).

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:

<table>
<thead>
<tr>
<th>Points</th>
<th>Cycle Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&gt; 719996</td>
</tr>
<tr>
<td>1</td>
<td>∈ (695552, 719996]</td>
</tr>
<tr>
<td>2</td>
<td>∈ (671108, 695552]</td>
</tr>
<tr>
<td>3</td>
<td>∈ (646664, 671108]</td>
</tr>
<tr>
<td>4</td>
<td>∈ (622220, 646664]</td>
</tr>
<tr>
<td>5</td>
<td>∈ (597776, 622220]</td>
</tr>
<tr>
<td>6</td>
<td>∈ (573332, 597776]</td>
</tr>
<tr>
<td>7</td>
<td>∈ (548888, 573332]</td>
</tr>
<tr>
<td>8</td>
<td>∈ (524444, 548888]</td>
</tr>
</tbody>
</table>
9 Points $\Leftarrow$ Cycle Count $\in (500000, 524444]$

10 Points $\Leftarrow$ Cycle Count $\leq 500000$

Remember that earning any points requires passing fullasmtests and having a critical-path delay at or below 550 ps. Once again, getting all of the points will be quite challenging.

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

The overall method we recommend to decrease your cycle count is to alternate between improving cycle count and critical path. First, decrease your cycle count as much as you can until your critical path goes over 550. Then switch your focus and try and reduce your critical path.

Hints

— The first step you should do is to pipeline your processor. Doing this will allow multiple stages to operate in parallel, dramatically reducing your cycle count. How to pipeline your processor is explained in Lectures 19 thru 21.

— We recommend that you try to make as small a number of changes as possible and then retest your your processor to make it is still functioning correctly. Do not try and make big changes; the more changes you have made for an improvement the harder it will be to debug if anything goes wrong. If you immediately try and jump from a multicycle processor to a 4- or 5-stage pipelined processor the staff will not be able to help you debug. When you add pipelining you should add only one stage at a time.

— Since you are starting with a multicycle processor, you should start by creating the two-stage processor as shown in Lecture 21. This should dramatically reduce your cycle count. Now you can check its critical path. If you find that it’s under 550 ps then you are good, if not you’ll need to decrease it.

— The first version of the cache for this processor should be the one you ported from Lab 7. However, you are free to change the cache however you want. A good place to start on this is to tweak the parameters of the cache in CacheTypes.bsv to change either the number of lines or the number of words per line. These numbers can be optimized with respect to the characteristics of the sorting program’s inputs to improve performance.

— Whenever you need to decrease your critical path there are two ways to do it. The first is to try to optimize the gates along the critical path, for example if the critical path goes through your adder you can speed it up by implementing a faster adder\(^1\). If the synthesis tool is hard to understand an easy way to check if something is along your critical path is to simply comment it out and synthesize again, this will temporarily make your code incorrect, but can help debugging your critical path.

— The second way to cut down your critical path is to add more pipeline stages. You should determine what the longest stage is and try to cut it as much in half as you can.

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 20 and 21, so you may want to wait to implement them.

3 Optimizing Hardware and Software (5 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 to make the combination of both 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.

\(^1\)The built in adder, using ‘+’, is pretty good so you probably will not be able to get that much of an improvement from this, but it’s certainly possible to get some.
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/sort_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.

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

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.

Score thresholds: Your score for Section 3 depends on the run time of sort_benchmark (Option 14 of test.sh) on your processor, as follows:

0 Points ⇐ Runtime > 180000
1 Point ⇐ Runtime ∈ (165000, 180000]
2 Points ⇐ Runtime ∈ (150000, 165000]
3 Points ⇐ Runtime ∈ (135000, 150000]
4 Points ⇐ Runtime ∈ (120000, 135000]
6 Points ⇐ Runtime ≤ 120000

To earn any points, your design must pass fullasmtests and sort_test. Earning all the points may be challenging, so 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 ./grade.sh 3 to see 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 20 and 21, 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.
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.
• 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 is 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 instructions, this module may produce an output similar to the one below:

```plaintext
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");
```

endrule
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.