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

Points: This project is worth up to 20 points (out of 200 points in 6.004).

Getting started: To create your initial Design Project repository, please visit the repository creation page at `https://6004.mit.edu/web/fall19/user/labs/project`. Once your repository is created, you can clone it on Athena by running:

```
git clone git@github.mit.edu:6004-fall19/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 (`https://6004.mit.edu`, Labs/Didit tab) to verify that your submission passes the tests for the points you think you should get.** If you finish the project on 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. Parts 1 and 2 are independent and you may do either or both. Part 3 builds on Parts 1 and 2, so you will need to complete them before attempting Part 3.

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

Introduction

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 their time sorting data, and are limited by the performance of this task. 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).

While working on the design project, keep in mind the following:

- You will earn more points by having your code or processor run faster. However, 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 (including from labs 2, 4, 5, 6, and 7). 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 give you hints on how to improve performance in each part and you should use what you learned in Lectures 19, 20, and 23, but you are allowed and encouraged to try other techniques!

Coding guidelines: Each part lists which files you can modify. In general, you should not modify files that are part of the test suite, as those files will be overwritten during Didit grading. You may add new files to your design, **but you must explicitly add, commit, and push these new files**.

Processor optimization guidelines: Before attempting Part 3 (and if you run into bottlenecks for Part 2), please read Appendix A, which describes how to optimize your processor by measuring and understanding the software, reducing the clock period, reducing CPI, and co-optimizing hardware and software.

Processor debugging guidelines: If your processor does not work as expected, please read Appendix B, which describes general debugging strategies and specific tips for pipelined processors.

1 Optimizing Software (4 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.

Optimize your quicksort implementation from Lab 2 to reduce the number of executed instructions. You may make any optimizations you wish as long as they would work for any input of the type described below (see the IMPORTANT note further down). You can even use a different sorting algorithm!

The data you will be sorting has the following properties, which may help you decide which sorting algorithm to use and what optimizations to apply:

- The numbers are signed 32-bit integers.
- The numbers are uniformly distributed over $[-2^{31}, 2^{31} - 1]$ (the full range for 32-bit signed integers).
- The numbers are in a random order.

All files for this part of the project are in the `part1/` directory. For this part, you may only modify `sort.S`. Start by changing into this directory (`cd project/part1`) and copying your code from Lab 2, like so: `cp ../../lab2/quicksort.S sort.S`.

Run `make` to compile your sort code. This will create two files:

- `sort_test.vmh` is the test program from Lab 2, which checks the correctness of your sort code.
- `sort_bench.vmh` is the benchmark program, 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 sort code, you can run the following commands:

- `rv_sim sort_test` to check the correctness of your sort assembly code.
- `rv_sim sort_bench` to measure the instruction count of your sort code.
- `rv_sim_gui` to use the graphical user interface for debugging.

Note that `rv_sim <program>` now gives other instruction-level statistics beyond instructions executed, such as branches taken, jumps, etc. These statistics will be useful to optimize your processor for Parts 2 and 3.

Score thresholds: Your score for Part 1 depends on the instruction count of your `sort_bench` program, as follows:

0 Points $\Leftarrow$ Instruction Count > 400000
1 Point $\Leftarrow$ Instruction Count $\in (325000, 400000]$
2 Points $\Leftarrow$ Instruction Count $\in (250000, 325000]$
3 Points $\Leftarrow$ Instruction Count $\in (210000, 250000]$
4 Points $\Leftarrow$ Instruction Count ≤ 210000

Your code must work correctly (i.e., pass `sort_test`) to earn any points. Earning all the points can be quite challenging, so if you don't see how to make further progress, we recommend that you attempt the other parts and come back to this if you have time.

You can run `../grade.sh 1` to see your current score for Part 1.

IMPORTANT: To make testing easy, we give you the arrays that your code will be benchmarked on, so you don't need to submit to Didit to find out your score. However, your optimizations must not use advance knowledge of the contents of the benchmark arrays to improve performance. An extreme example of this would be to store a sorted version of the benchmark arrays in your code, and copy them out instead of actually sorting the arrays. More subtle examples include precomputing pivot points from the benchmark arrays to minimize instructions executed, then hard-coding those pivots in your code.

In general, any optimization that relies on information derived from the benchmark arrays (other than the length of the arrays) is *strictly forbidden*. We will manually check for these violations at the end of the project, and code containing these tricks will be penalized or, in extreme cases, given zero points. (You should not be worried about this as long as you're acting in good faith.)

Hints

There are many ways to make your code faster. To get the most out of your changes, focus on optimizing the code that runs most frequently, *e.g.,* 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: Each time you make a call, you pay for both the call and return instructions 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. *Tail call elimination* is a particularly useful optimization of this type for quicksort.

Defer or avoid optimizations that increase code size: Some optimizations, like loop unrolling, yield some performance gain but they are tedious to do by hand and result in very large code that will be nearly impossible to optimize further. We recommend that you avoid these and focus on other avenues, such as exploring other algorithmic variants, inlining, and other optimizations that reduce the size of your code rather than increasing it. If you're a few instructions away, you can always apply some unrolling at the end to get past the point threshold, but leave this as a last resort.

There are two benefits to avoiding unrolling. First, unrolling is tedious, and the thresholds are set so you don't need it (*e.g.,* you can get full credit with a 35-instruction sort routine). Second, using a large number of instructions may increase instruction cache misses, which will be detrimental to performance on Part 3.

Further reading: For more suggestions, begin by reading https://en.wikipedia.org/wiki/Optimizing_compiler

2 Optimizing Hardware (6 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 in this system so that it performs sorting quickly. However, there are three constraints that you cannot control. First, Google does not trust your assembly code and insists on providing their own code. Second, the system *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—it cannot have a critical path longer than 550 ps. It may have a shorter critical path, but that will not change the clock period because other parts of the system require a 550 ps clock. Third, your processor should use a real memory system with instruction and data caches.

As we saw in Lecture 19, the time spent executing a program can be expressed as the product of three components: $Instrs \cdot CPI \cdot t_{CLK}$, where $Instrs$ is the number of instructions executed, CPI is the average cycles per instruction, and t_{CLK} is the clock period. Your goal is to maximize performance, *i.e.,* minimize execution time. In this case, $Instrs$ and t_{CLK} are fixed, so your goal is to *minimize* CPI while staying at or below the target t_{CLK} of 550 ps.

To achieve the target t_{CLK} while keeping CPI reasonably low, you'll need to implement a pipelined processor. This part and its score thresholds are structured to guide you towards a good implementation. Achieving full points in this part doesn't require applying many optimizations—the goal of this part is to give you a reasonable processor that you can use as a baseline for Part 3, which is more open-ended. Note that doing a good job here will automatically give you some points in Part 3.

Starting from your single-cycle processor from Lab 6, build a processor that uses a realistic memory system and optimize it to run the provided sorting code as quickly as possible. To get all the points in this part, you will need to pipeline your processor, integrate your cache from Lab 7, and add some mechanisms to reduce the impact of data and control hazards.

Your processor will execute a baseline sorting program, `sort_base`, that comes from a quicksort implemented in C and compiled with `gcc -O2`. The goal of this part is to reduce the cycle count of the processor when running `sort_base`.

All files for this part of the project are in the `part2/` directory. Start by changing into this directory (`cd project/part2`) and copying the following files from prior labs:

- From Lab 6, copy `Decode.ms`, `Execute.ms`, and `ALU.ms`
- From Lab 7, copy `CacheHelpers.ms`, `DirectMappedCache.ms`, and `TwoWayCache.ms`

The `part2/` directory already has a skeleton for the pipelined processor in `Processor.ms`, and also includes the testing infrastructure from labs 6 and 7, so you can test your pipelined processor and caches.

You can modify most Minispec files (including `ProcTypes.ms` and other files that were off-limits before) but do not modify any other types of files since they will be overwritten during Didit grading. The only Minispec files that you cannot modify are `MainMemory.ms` and `SRAM.ms`.

Compile your processor by running `make Processor`. You can use `./test.py` to test the processor. Running `sort_base` (Option `s1` in test.py) will give you the score for this part.

Score thresholds: Your score for Part 2 depends on the critical path of your processor and its CPI on the `sort_base` program, as follows:

0 Points $\Leftarrow t_{CLK} > 550$ps
2 Points $\Leftarrow$ Processor with single-cycle instruction/data memories and $CPI < 1.5$
4 Points $\Leftarrow$ Processor with instruction/data caches and $CPI < 2.5$
5 Points $\Leftarrow$ Processor with instruction/data caches and $CPI < 2.0$
6 Points $\Leftarrow$ Processor with instruction/data caches and $CPI < 1.5$

Your code must work correctly (*i.e.,* pass the tests) to earn any points. These thresholds are designed to help you get partial credit along the way if you follow our recommended implementation strategy, detailed below. However, you need not follow this strategy.

You can run `../grade.sh 2` to see your current score for Part 2. We also recommend that you push your code to Didit occasionally, to make sure that you haven't modified a file that breaks Didit grading.

Recommended Implementation Strategy

We recommend that you complete this design exercise by following these milestones:

1. Implement a 4-stage pipelined processor with *single-cycle instruction and data memories* and no bypassing.
2. Substitute the single-cycle memories with one of your caches from Lab 7, achieving a processor with *instruction and data caches*. This will require adding stall and control logic to your processor to account for the fact that caches cannot take in a request on every cycle.
3. Pipeline your cache so that it can sustain one load per cycle if the loads hit. This is particularly important for the instruction cache, since we want to fetch one instruction every cycle. However, your cache from Lab 7 only supports one load request every two cycles, even when the loads hit. This will improve CPI.
4. Add bypassing to your pipeline to further improve CPI.

Completing each of these milestones should get you additional points given the thresholds above. This implementation strategy and the target clock period assumes you have a reasonably fast implementation of `decode()` and `execute()` and of the logic in `Processor.ms`. If your processor becomes too slow at some point (*e.g.,* after adding caches or bypasses), refer to Appendix A for optimization tips on reducing clock cycle time. Conversely, if you have a processor implementation with a really low critical path, you may be able to use fewer pipeline stages (but unless you've spent significant effort optimizing the critical path on Lab 6, it's very unlikely that your critical path is low enough to support 3 stages).

We heavily recommend that you **commit and push your code after each milestone** (at least), and that you commit intermediate working versions as well. If you don't do this, you may break a working design and be unable to go back to it. If you commit your code, you can always revert a broken pipeline to a previously working one.

The rest of this section describes each of the milestones in more detail.

Milestone 1: 4-stage pipeline with single-cycle memories

Code overview: `Processor.ms` includes skeleton code for a 4-cycle pipeline. This skeleton code has three main differences from your single-cycle processor:

First, the instruction and data memories, `iMem` and `dMem`, are single-cycle memories rather than magic memories. Whereas `MagicMemory` had combinational reads (*i.e.,* data was returned on the same cycle), `SingleCycleMemory` works the same way as the SRAMs you used in Lab 7: it has a single `req` input through which you can issue requests, and read data is available *on the following cycle*. However, unlike the SRAMs, `SingleCycleMemory` takes requests of type `MemReq`, the same type of requests that your caches took. `SingleCycleMemory` can take a new request every cycle (*i.e.,* it does not stall), and all requests are satisfied in a single cycle. Because they are simpler than caches, they are a nice starting point for your pipelined processor.

Second, in addition to the previous state elements (PC, register file, etc.), the `Processor` module has three *pipeline registers*: `f2d`, `d2e`, and `e2w`. This is because we recommend that you implement a pipeline with the following four stages:

1. Fetch should initiate an instruction fetch from `iMem`.
2. Decode should read the instruction returned by `iMem`, decode it, and read its source registers.
3. Execute should produce the outputs of the instruction (by calling `execute()`) and initiate a request to `dMem` for loads and stores.
4. Writeback should write back the relevant data to the register file. For loads, this data should be the output of `dMem`.

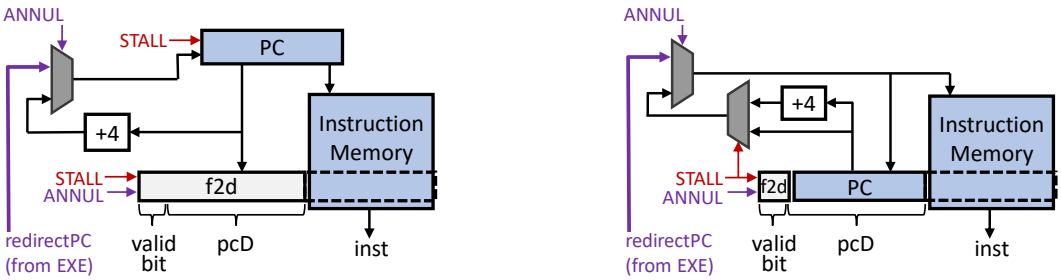
Note that this pipeline is slightly different from the 5-stage pipeline we saw in lecture: there is no Memory stage. This is because the data memory (and later, the data cache) return data fast enough to meet our target clock cycle. If in Part 3 you choose to optimize for a lower clock period, at some point you may need to add a Memory stage.

Each pipeline register is named after the stages it separates (*e.g.,* `d2e` is the pipeline register between Decode and Execute). Because the processor can stall and the pipeline is initially empty, each pipeline register is a Maybe type; it is Invalid when the downstream stage has a NOP, and has valid contents otherwise. Each pipeline register holds the contents needed by downstream stages; for example, `d2e` includes the instruction's PC, the decoded instruction, and its source register values. Note that the PC field in each stage is used to keep track of the PC that the instruction in that stage originally came from, not necessarily the current value of the PC register. We already give you register definitions that include all necessary data, but you can freely change the contents of each pipeline register, or add/remove pipeline registers.

Third, the processor rule is structured differently from the single-cycle rule: whereas in the single-cycle processor the code followed the steps required to execute the instruction (first fetch, then decode, then execute, etc.), in the rule of the pipelined processor, *pipeline stages are laid out in reverse order!*. In other words, the rule begins with writeback, then execute, then decode, and finally fetch.

This reversed structure stems from the fact that *later pipeline stages can affect earlier ones*, but *earlier pipeline stages cannot affect later ones*. For example, Writeback or Execute can stall Decode, but Decode cannot stall Execute or Writeback. By writing the stages in reverse order, we can write the signals that cross pipeline stages (such as values used to stall, bypass, or annul) as simple variables that a later stage (*i.e.,* code towards the beginning of the rule) sets and an earlier stage (*i.e.,* code towards the end of the rule) reads.

Structure of the Fetch stage: We recommend that you implement the Fetch stage somewhat differently from what we have seen in Lecture. We give you the detailed structure of Fetch for two reasons: *(i)* it is the one part of the pipeline different from Lecture, and *(ii) following this structure will make adding caches much easier*.



(a) Standard Fetch stage

(b) Recommended Fetch stage

Figure 1: Comparison of the standard Fetch stage, with separate Fetch and Decode PC registers, and the recommended Fetch stage implementation, where Fetch and Decode share a single PC register.

Figure 1 compares a standard Fetch stage with the Fetch stage we recommend. In the standard implementation from Lecture, shown in Figure 1a, the Fetch and Decode stages use separate PC registers: PC is in the Fetch and holds the PC of the instruction we need to fetch this cycle, and Decode's pipeline register, `f2d`, has the PC of the instruction currently in the Decode stage (*i.e.*, at the output of the memory).

By contrast, in our recommended implementation, shown in Figure 1b, the Fetch and Decode stages share the same PC register. The PC should hold the address of the instruction *in the Decode stage*, and the Fetch stage uses this PC to compute the address to request to the instruction memory (which can be either PC+4, the `redirectPC` from the Execute stage on an annulment, or PC if stalled).

Our recommended design has two advantages over the standard one. First, *there is less state to manage*, a single PC instead of two. This doesn't matter much for now, but when adding caches (which may stall), keeping Fetch and Decode PCs in sync becomes cumbersome. Second, this design reduces the misprediction penalty by one cycle: on an annulment, instead of updating the PC register and then starting a fetch to the right PC on the next cycle, we start an instruction fetch to the right PC on the cycle of the annulment.

Implementation steps: For this first milestone, you only need to fill in the `Processor` rule to implement a working 4-stage pipeline. You can test your implementation using `test.py` just like in Lab 6. The following steps are a reasonable approach to get a working pipeline:

1. Fill in the code for every stage based on your processor code from Lab 6, but do not implement stall logic yet. In fetch, always increment `pc` by 4. This will fail most microtests, but will pass the few that have no dependences (up to `02_addi`).
2. Add in stall logic by passing the destination registers currently in Execute and Writeback to Decode, and stall Decode + Fetch if the instruction in Decode reads any of the destination registers that are still in flight. This will pass quite a few microtests (`00-03`, `07`, `08`, `10`, `11`, and `13`), but will still fail any test that has branches or jumps that are taken.
3. Add in annulment logic to handle all instructions where Execute determines that nextPc does not match the predicted pc (pc+4). This should pass all tests.

Milestone 2: 4-stage pipeline with instruction and data caches

While single-cycle memories are simple, they are impractically expensive. When implemented as SRAMs, `iMem` and `dMem` take an area of about 850000 μm^2, about 50× the size of your processor! (you can synthesize your processor to compare). In addition, it's unrealistic for memories this large to take a single cycle. This is why you should use small caches instead, connected to a large (and cheap and slow) main memory.

Thus, the next step is to substitute these single-cycle memories with one of your caches from Lab 7. `Processor.ms` already has some of the logic to help you do this—search for the comments that mention "Milestone 2".

First, at the top of `Processor.ms`, import either your `DirectMappedCache` or your `TwoWayCache`. Which one you use is up to you—we suggest you start with `TwoWayCache` since it'll have fewer misses and thus better CPI, but `DirectMappedCache` has a shorter critical path, so it may give you a lower t_{CLK}. You can even mix and match (*e.g.*, a direct-mapped instruction cache + a 2-way data cache).

Second, comment out the `SingleCycleMemory iMem` and `dMem` submodules, and uncomment the code below that instantiates the caches and main memory ports. You will end up with cache modules `iCache` and `dCache`, and main memory port modules `iMem` and `dMem`, which the caches take as inputs and which you should not use directly. *It is important that you follow these naming conventions*, especially for `iMem` and `dMem`: the grading script checks that your processor has main memory ports named that way.

Third, change your `Processor` code to use `iCache` and `dCache` instead of `iMem` and `dMem`. The pipeline needs to stall when the cache's `reqEnabled` output is `False`.

We recommend you change your caches one by one: first substitute the `iMem` for an `iCache` (still using a `SingleCycleMemory dMem`), then swap in `dMem` for a `dCache`. Both caches have slightly different needs and constraints on the pipeline, so here are some hints:

- Because caches have a variable latency, you will need to stall whenever the pipeline register has a Valid instruction that needs data but the cache's `data` output is Invalid. For example, for the `iCache`, Decode must stall whenever `f2d` is valid but `iCache.data` is invalid.
- Unlike other multi-cycle modules, caches do not have an output register and, for each load request, they give valid data for only a single cycle. In the `iCache`, this is a bit tricky because if *(i)* Fetch issues `iCache` requests only when it is not stalled, and *(ii)* Decode stalls (due to a data hazard) when the `iCache` replies, then the fetched instruction will be lost.
  There is a simple solution to this problem: in Fetch, fetch an instruction whenever the `iCache` lets you (*i.e.,* whenever `iCache.reqEnabled == True`), even if Fetch is stalled. If Decode stalls, then Fetch will also stall and retain the current PC, so the same instruction will be fetched multiple times until Decode can make progress.
  This issue does not arise in the `dCache`, because the only thing that stalls Writeback is a `dCache` miss (so it's never the case that `dCache` returns data but Writeback is stalled).
- Watch out for the interaction between annulments and stalls caused by `iCache` misses. Suppose that Fetch and Decode are currently stalled on an `iCache` miss, but Execute has a taken branch, so Execute sets the annul signal. What should happen in Decode and Fetch? Clearly, even though it's stalled, Fetch should take Execute's `redirectPC`, because the `pc` register is simply a wrong value. But Decode cannot simply mark the `f2d` register invalid, because the `iCache` is in the middle of a miss and will reply with an incorrect instruction! Instead, when Decode is annulled while on an `iCache` miss, you need to have an extra bit of state that remembers that the `iCache` is going to return an instruction that should be ignored.
  Again, this issue does not arise in the `dCache`, because instructions are not annulled after the Decode stage in this processor (and so loads in Writeback are never annulled).

Once you've addressed these issues, you should have a pipelined RISC-V processor with caches that passes all the tests. This is a great achievement, and you should feel proud! However, due to cache stalls, your processor will have a significantly higher CPI than your processor from Milestone 1. Fortunately, the next two milestones address this with two simple enhancements.

Milestone 3: 4-stage pipeline with *pipelined* instruction and data caches

The caches you implemented in Lab 7 have an obvious limitation: they only support one request every two cycles. This is not a major problem for the `dCache`, because few instructions are loads or stores. But it is nearly doubling the CPI of your pipeline, because the `iCache` cannot take a new request every cycle!

To solve this problem, you can modify your cache implementation to *pipeline load hits*. Note how, in your cache from Lab 7, in the Lookup state, when there is a load hit the cache does not issue any accesses to the data/tag/status SRAM arrays. This makes pipelining load hits quite simple: if the cache is in the Lookup state and there is a load hit, the cache should *(i)* output `reqEnabled == True` to allow a new request, and *(ii)* if `req` contains a new request, initiate it (while in the Lookup state) by issuing reads to the data/tag/status SRAM arrays. Figure 2 shows this graphically by highlighting the modifications to the cache FSM from Lab 7 to pipeline load hits.

With this change, your `iCache` will be able to fetch one instruction per cycle, improving CPI substantially.

Note: There are more sophisticated ways to pipeline the cache, such as pipelining both load and store hits. While commercial processors often do this, we explicitly recommend that you do *not* attempt this—pipelining

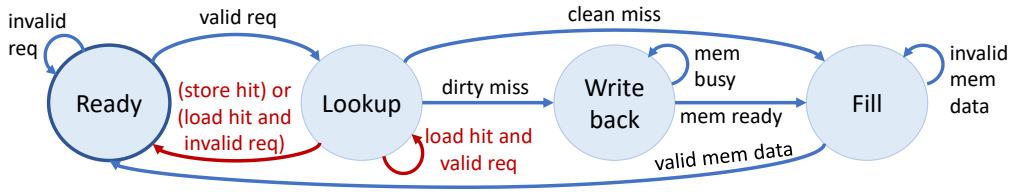


Figure 2: State-transition diagram for handling a cache request with pipelined load hits. Changes from the Lab 7 state-transition diagram are highlighted in red.

store hits is substantially more complicated and will yield negligible benefit: in this part consecutive stores are not common, and in Part 3 you can change the software to avoid back-to-back stores altogether.

Milestone 4: 4-stage pipeline with bypasses

To decrease CPI further, you can add bypass paths to your pipeline: in Decode, instead of always stalling when one of the instruction's source operands is the destination of an in-flight instruction, you can route the data from the later stage to decode (as long as the instruction has produced the data already).

As discussed in Lectures 19 and 20, watch out for multiple matches (you need to route the data from the correct stage). Also, beware that bypassing from both Execute and Writeback may add too much delay and push t_{CLK} past its limit. To avoid this, you can either bypass from only one of the stages, or make your other components faster (which will also help you with Part 3).

3 Optimizing Hardware and Software (10 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 Part 2, you have no limitations: you can use your own sorting program and the frequency of the system will be determined by the critical path of your processor.

> Building on your code from parts 1 and 2, design a processor and sorting algorithm that minimize the amount of time it takes your processor to run your sorting code, that is, the cycles needed to execute your code times the clock period of your design.

This part of the project is open-ended. How to improve performance 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 will likely have to trade one metric for another to get the best overall performance.

This part of the project requires the use of a realistic memory system with caches. Do not begin this part until you have finished Part 2 until at least Milestone 2 (though we recommend finishing Part 2 completely). Conversely, *simply importing well-performing implementations from parts 1 and 2 will automatically give you some points on this part of the project.*

Appendix A contains detailed guidance on how to analyze and optimize performance. We recommend that you read the guidance on analyzing performance. You can follow some of the many optimization tips in Appendix A, or explore other options on your own. The score thresholds are set so that getting all the points will require substantial optimization effort, but you can try these optimizations incrementally, and not all of them are necessary. Finally, some optimizations are more effective than others, and some are in direct opposition (e.g., they will improve CPI but may hurt t_{CLK}), so you should combine them judiciously.

For example, one strategy is to aggressively minimize t_{CLK} at the expense of a somewhat higher CPI. (this style of processor is known as a *speed demon*). In this case, applying some optimizations to improve CPI will be fine as long as t_{CLK} isn't dramatically worse, but a single CPI optimization that tanks t_{CLK} will be bad tradeoff overall. Alternatively, you could aggressively improve CPI without worrying too much about minimizing t_{CLK} (this style of processor is known as a *brainiac*). In that case, optimizations that improve t_{CLK} at the expense of CPI may help if you have a single particularly long path on your design, but if you have many long paths, they are unlikely to help. In short, don't put tractor wheels on your F1 car.

All files for this part of the project are in the `part3/` directory. Start by changing into this directory (`cd project/part3`) and copying the following files from prior parts:

- From Part 1, copy `sort.S` into the `sw/sort/` subfolder, like so: `cp ../part1/sort.S sw/sort/`
- From Part 2, copy `Processor.ms`, `Decode.ms`, `Execute.ms`, `ALU.ms`, `CacheHelpers.ms`, `DirectMappedCache.ms`, `TwoWayCache.ms`, and any other Minispec files you have changed.

The `part3/` directory includes the testing infrastructure from previous parts. Do not modify `MainMemory.ms`, `SRAM.ms`, or any of the files that are part of the test suite. Like in Part 2, you can modify any other Minispec file.

Like before, compile your processor by running `make Processor`. You can use `./test.py` to test the processor. Running `sort_bench` (Option `s3` in test.py) will give you the runtime (cycle count $\times$ clock period) of your processor on `sort_bench`.

Score thresholds: Your score for Part 3 depends on the runtime of `sort_bench` (Option `s3` of `test.py`) on your processor in nanoseconds, as follows:

0 Points $\Leftarrow$ Runtime $> 180000ns$
1 Point $\Leftarrow$ Runtime $\in (165000ns, 180000ns]$
2 Points $\Leftarrow$ Runtime $\in (150000ns, 165000ns]$
3 Points $\Leftarrow$ Runtime $\in (135000ns, 150000ns]$
4 Points $\Leftarrow$ Runtime $\in (125000ns, 135000ns]$
5 Points $\Leftarrow$ Runtime $\in (115000ns, 125000ns]$
6 Points $\Leftarrow$ Runtime $\in (105000ns, 115000ns]$
7 Points $\Leftarrow$ Runtime $\in (95000ns, 105000ns]$
8 Points $\Leftarrow$ Runtime $\in (90000ns, 95000ns]$
9 Points $\Leftarrow$ Runtime $\in (85000ns, 90000ns]$
10 Points $\Leftarrow$ Runtime $\leq 85000ns$

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 parts first. It will be easier to get a good score in this part if you start from reasonably optimized designs (3 or 4 points in Part 1 and 5 or 6 points in Part 2).

You can run `../grade.sh 3` to see to see your current score for Part 3.

Appendix

A Processor Optimization Guide

A.1 Profiling your software

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

We have enhanced `rv_sim` 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, simply run the different sort benchmarks in each part of the project. Each of these stats is relevant for data and control hazards, which you can then use to estimate the number of cycles lost to stalls and mispredictions in your processor pipeline.

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

A.2 Understanding the performance of your processor

For a program with a fixed number of instructions, the time taken by a processor depends on the product of two factors: the CPI (cycles per instruction), and the t_{CLK} (clock cycle time).

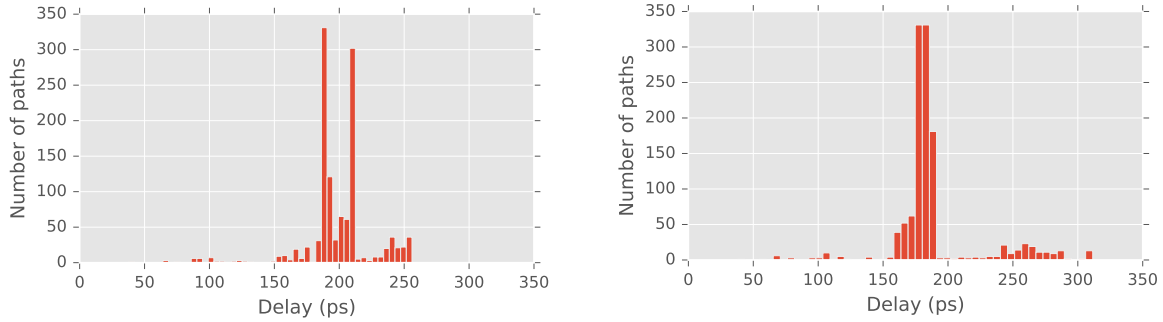
Measuring and analyzing CPI is simple: `rv_sim` counts instructions, and the processor skeleton code we give you already counts the number of cycles and reports it at the end of execution. You could add additional counters to see how many cycles are lost to different events (data hazards, control hazards, etc.), but the event counts from `rv_sim` alone should be enough to give you a good idea about stalls.

Measuring and analyzing clock cycle time is a bit more involved. When you synthesize the processor with `synth`, by default only the critical path is shown. This is useful to understand what path sets t_{CLK}. But to optimize your processor, it's useful to understand the delay of other paths. For example, if you're shifting logic across pipeline stages to reduce the critical-path length, you'd like some assurance that the path you're moving logic into won't become the critical path.

We have updated `synth` to help you with this analysis. The new `-p <N>` option reports the N longest paths in the design (to see the delay of all paths, use $N = -1$). For example,

```
synth Processor.ms Processor -l multisize -p 100
```

shows the 100 paths with the longest propagation delays. For each path, `synth` will show the start and end points, and the In addition, `synth` now understands Minispec types and their internal layout, and will print the particular struct member or vector element that each bit corresponds to.



(a) With multisize (critical path: 255 ps) (b) With extended (critical path: 311 ps)

Figure 3: Path delay histograms for a processor synthesized with the (a) multisize and (b) extended libraries.

If you are optimizing for clock cycle time aggressively, we recommend that you analyze your designs with both the `multisize` and `extended` libraries. The reason is that, with `multisize`, `synth` uses larger gates on the critical/near-critical paths and smaller gates on non-critical paths. This causes the few longest paths to be clustered near the critical path, which may make it look like there are few optimization opportunities because there are many near-critical paths. By contrast, `extended` only has gates of a single size, so long paths will be less clustered, revealing optimization opportunities.

Figure 3 shows this effect by comparing the distributions of path lengths for the same processor, when synthesized with `multisize` and `extended`. With `multisize` the critical path is shorter, 255 ps, but there's a substantial number of paths near 250 ps because `synth` decides to save area on them by using smaller gates. With `extended` the critical path is longer, 311 ps, but long paths are far more scattered, and we can see there's only a small group of (related) paths over 300 ps. Optimizing those paths would let us reduce the critical path by about 20 ps on `extended`, and by about 10 ps on `multisize`. (This processor is already well optimized, so these gains are small; in your design you'll see larger spreads).

Finally, sometimes the path analysis may not be fine-grain enough to tell you where the critical path is. For example, if you have a critical path in the Execute stage that goes through the ALU, it's not immediately clear which logic block is causing it.

To find the culprit, a simple option is to *synthesize smaller functions*, such as the main functions within the ALU (like you did in Lab 4). You can then pinpoint which component sets the critical path.

When synthesizing individual components is hard (e.g., if the critical path is in a stall signal that crosses pipeline stages), you can *remove the path from your design* and synthesize an (incorrect) design to see how performance improves. For example, if the critical path is between Execute's `nextPc` and the input of the `pc` register, you can comment out the logic that sets `nextPc`, and see how much cycle time would improve.

As a final alternative, `synth` has an experimental option called `--names` that will try to recover the names of the wires along the critical path. But this is a very complex process, and `synth` will sometimes crash when attempting it. Moreover, if you reuse the same variable name constantly (e.g., "data"), these names won't be very informative. So treat the `--names` option as a convenience that may be useful, but use the above alternatives as your main strategy.

Now that you have the tools needed to understand performance, the rest of the appendix describes techniques to *(i)* reduce clock cycle time, *(ii)* reduce CPI, and *(iii)* improve performance by changing hardware and software at the same time (for Part 3 only).

A.3 Optimizing clock cycle time

Begin by identifying your processor's critical path. Then, consider some of the options below. We classify options by difficulty (easy, medium, or advanced).

Use a fast adder (Easy): If your design has an adder in its critical path, consider using a faster adder. The `+` operator in Minispec produces a 32-bit adder that has a latency of about 210 ps—much better than a ripple-carry adder, but we can do better still. A Kogge-Stone carry-lookahead adder (`https://en.wikipedia.org/wiki/Kogge-Stone_adder`) has a latency of about 140 ps if implemented well and should only take about 10 lines of Minispec.

Reduce the gates along the critical path (Easy-Medium): In some cases, you will find a long critical path that arises from unnecessary logic within it. For example, if in your decoder you're not setting `src1` and `src2` to constant values (e.g., setting them to zero for instructions that do not read registers), the reads of the register file will start later. Similar optimizations include reducing the number of multiplexers by reusing components (e.g., adders in your execute stage), optimizing the decoder, etc.

Add pipeline stages (Medium): You will probably need to add a pipeline stage between your `iCache` and the Decode stage (creating a second Fetch stage), and between the `dCache` output and the Writeback stage (creating a Memory stage). Add stages only when it's clear you will get significant benefits, because more stages generally increase CPI.

Change cache geometry (Easy): You can switch from a two-way cache to a direct-mapped one to reduce clock cycle time. Alternatively, you can change the number of cache sets and the number of words per line by changing the relevant variables in `CacheTypes.ms`. These may improve clock cycle time at the expense of some increase in CPI. Note that main memory expects `wordsPerLine` to be 4, 8, or 16 (which is the default value), and that having more words per line will improve main memory bandwidth.

Balance logic across pipeline stages (Medium): Sometimes you'll end up with pipeline stages that have imbalanced delays. For example, Decode may be shorter than Execute. In that case, you may want to shift some of the computation in Execute to Decode.

Change the data representation to reduce delay (Medium): Sometimes you can encode data in way that reduces computation at the expense of adding storage. For example, suppose that your critical path is in the ALU's adder, which you are using for both addition and subtraction as in Lab 4. To figure out whether to add or subtract, the ALU first needs to compute whether `aluFunc == Sub`, which takes a non-trivial delay. Instead, you could spend an extra bit in the `d2e` register to directly encode whether `aluFunc == Sub`, so that the right operation can start at the beginning of the cycle. Similar optimizations may apply to fields in `DecodedInst` and `ExecInst`.

Avoid high-fanout paths (Medium-Advanced): As a variant of the above optimization, sometimes a single signal is driving a many gates, and this causes significant delay (remember the multiplexer `synth` exercise from Lab 4). You can reduce fanout across pipeline stages by replicating the same value over several registers, then using each value on a smaller number of gates.

A.4 Optimizing cycles per instruction

First, analyze the CPI bottlenecks in your code as explained above. Then, consider the options below.

Add data bypasses (Easy): Route results from instructions in Execute and Writeback to Decode, reducing stalls due to data hazards.

Change cache geometry (Easy): Just like you can tweak the caches to reduce the clock cycle (Section A.3), you can change their parameters (number of sets, ways, and words per line) to reduce misses and thus CPI, perhaps at the expense of a higher clock cycle.

Improve nextPC predictions (Medium-Advanced): Always predicting nextPC as PC+4 is inefficient. Jumps and taken branches are always mispredicted, adding misprediction penalties that can add up quickly, since sorting code is often branch-heavy. A *branch target buffer* (BTB) is a simple next-PC predictor that remembers past behavior (specifically, past taken branches and jumps) to better predict nextPC. At a high level, a BTB is a tiny cache for branch outcomes that sits on the fetch stage: the "tag" is the PC of a taken branch or jump, and the "data" is its next PC. On each instruction fetch, the Fetch stage looks up the BTB using the current PC. On a BTB hit, Fetch uses the output of the BTB as the predicted next PC. Otherwise, it predicts PC+4 as the next PC. The BTB trains through mispredictions: when Execute signals a misprediction, Fetch inserts the correct (pc, nextPC) pair in the BTB. We will see BTBs in more detail in Lecture 23, but you can explore read further on your own and implement it in advance (see the reference materials for Lecture 23 for additional information).

Hints: Your BTB should use normal registers rather than SRAM. To implement an array of normal registers, you can use a `Vector` of registers as in Lab 5 and in the register file. Experiment with the size of BTB to find the optimal number of entries to reduce mispredictions without hurting critical-path delay.

Make it superscalar (Very Advanced): With every optimization so far, the best you can achieve is a CPI that approaches 1.0. To reduce CPI even further, your processor would need to fetch, decode, and execute *more than one instruction per cycle*. This is what is called a *superscalar* design.

While most commercial processors are superscalar and they routinely execute 2-4 instructions per cycle, this is quite a complex optimization and we normally would not be giving it as a suggestion. But there's a saving grace: while it is extremely hard to build a processor that executes *any* pair of consecutive instructions in a single cycle (e.g., two loads, two stores, etc.), building a processor that can execute pairs of *some types* of instructions in parallel (e.g., a load and an ALU instruction) is comparatively much easier (but still quite hard). Because in Part 3 you have full control of the software, you can build a 2-wide superscalar pipeline with lots of restrictions on which pairs of instructions can execute on the same cycle, and yet, by carefully scheduling instructions into appropriate pairs, achieve a CPI reasonably close to 0.5.

For example, you can treat your current pipeline as the *primary* pipeline, which can run any type of instruction, and have a simpler *secondary* pipeline that can run the next instruction if several restrictions are met. First, to fetch two instructions per cycle, note that your `iCache` has 16-word cache lines. Unless the PC points to the last word in the line, your cache can easily return the next word in the line (i.e., the instruction at PC+4) through an additional method similar to `data` (e.g., `Maybe#(Word) nextData`). Then, you can add another decoder to your Decode stage, extend the register file to have 4 read ports and 2 write ports, and restrict the secondary pipeline to run only OP or OPIMM instructions, so that Execute can implement this with only an extra ALU. Finally, in Fetch you will need to decide whether to fetch at PC+8, PC+4, or PC, depending on whether 2, 1, or no instructions will proceed from Decode to Execute.

While this can be a very rewarding optimization to get right and it is doable with the new infrastructure in the course, note that this is by far the hardest of our suggestions, it is not needed to get full credit, and as far as we know, no student in 6.004 has completed it before.

A.5 Co-optimizing hardware and software

The optimizations below apply to Part 3 only, where you have control of both hardware and software. They can be very effective, but they require you to have a deep understanding of how code and processor interact.

Change your code to minimize stalls in your pipeline (Easy): Many stalls due to data hazards can be avoided by leaving enough space between dependent instructions. Even when you use bypassing, you can avoid implementing a fully bypassed pipeline by placing dependent instructions at the right distances. For example, if you bypass from Execute but not from Writeback, you can have back-to-back dependent instructions in your code but avoid dependences that are two instructions away.

Add simple instructions (Medium): If you had other simple instructions in your processor, such as min, max, or conditional moves, you could reduce the number of instructions and/or cycles spent in your code's most frequent loops. You can implement additional instructions similarly to how we have seen in recitation problems and quizzes, by *(i)* picking an instruction encoding that is currently unused, *(ii)* extending your decoder to decode this new instruction, and *(iii)* extending your Execute stage to implement this new function.

If you choose this path, you can encode each instruction in your program directly, using assembly `.word` directives. For example, to insert a new instruction with encoding `0x5CA1AB1E` between two existing instructions, your code would look like this:

```
add a1, a1, a0
.word 0x5CA1AB1E  // new instruction, encoded directly in assembly as a 32-bit data word
xor a3, a2, a1
```

Add branches with delay slots (Medium): This is a variant of the above technique meant to reduce or eliminate the impact of branch mispredictions. Branch delay slots (https://en.wikipedia.org/wiki/Delay_slot) allow delaying the effect of a branch for a fixed number of instructions, so that the 1–2 instructions following the branch *always execute, regardless of whether the branch is taken.* This way, on a misprediction, you do not need to annul the instructions following the branch (however, you need to be able to put some useful work after the branch).

Branch delay slots were invented in the 80's and were featured in some of the very first RISC processors (specifically, the Stanford MIPS). Today, they are considered outmodded and processors instead implement

sophisticated branch predictors to reduce mispredicitons. They also have the problem of making the code hard to understand (as you need to think of the branch as taking effect later). But delay slots, hacky as they are, work pretty well for the specific processor you're building.

Since RISC-V does *not* have delay slots, you cannot change existing branch instructions to have a delayed effect. But you could add variants of branch instructions with delay slots. As explained above, you'd then have to insert these branches in your code using `.word` macros. There's no need to change all the branches in your code—a few will get you most of the benefit.

Reuse your sorting network (Medium-Advanced): In Part 1, you probably found that your code spends a significant amount of time on the "base case", i.e., sorting small arrays are then merged into larger ones. The pipelined bitonic sorting network from Lab 5 could do this part much faster. Since the bitonic sorting network you built takes 8 words at a time and the data cache has 16-word cache lines, a single cache read could feed the pipeline for two cycles, leaving some cycles to store back the sorted results.

To use the sorting network in your processor, you could have a sort instruction for each 8-word chunk, but it will likely be easier to have a single instruction that gives the entire range of memory that the sorting network should process (e.g., the whole array), and stalls the processor pipeline until the sorting network finishes processing its assigned data. Note that, even though the sorting network will do most of the work, you still need to have a processor capable of running normal RISC-V instructions, and should initiate the bitonic sorting network operation by adding a new instruction.

For a more advanced variant of this approach, you could use your sorting network to build a specialized circuit that sorts the entire array. This will yield much higher speedup than only sorting groups of 8 words.

Finally, note that your Lab 5 sorting network performs *unsigned* comparisons, but for this project you should sort arrays of *signed* elements. This will require some simple changes to your sorting network.

B Debugging Help

If your processor does not work as expected, there are some simple strategies you can follow to debug it. Building your debugging skills is especially important for the project, because the project is more open-ended than the labs. Thus, you are more likely to find problems that are unique to your design.

General guidelines: `$display` statements are your main debugging tool. When things do not work, add them to the rules in your processor and/or cache. In the processor, you can display the current cycle (in the `cycle` register). You can display the contents of complex types with `fshow()`. For example, `$display(fshow(e2w))` will display the contents of each member of the `e2w` pipeline register. Section 7 of the Minispec sequential logic tutorial gives more information on `$display` and debugging of modules.

Debugging pipelined processors: In `Processor.ms`, we have included code at the top of the rule that prints the state of the pipeline on each cycle. For each stage, it prints whether each stage has a valid instruction, and if so, the instruction's PC. This is often sufficient to pinpoint some bugs, but for others you will want to print more information, such as the stall and annul signals, where Decode is bypassing from when a bypass path is exercised, or the contents of registers written in the Writeback stage.

Debugging pipelined processors is also somewhat different from debugging modules in prior labs in that, when something goes wrong, the error manifests a few cycles later rather than immediately. Thus, you may have to look at the trace of events over multiple cycles to figure out what went wrong.

Let's go over a concrete example. Suppose that your pipelined processor is failing a microtest because register `x3` does not match the expected value. A good starting point is to add a `$display` statement to the Writeback stage to print the contents of all writes to register `x3`. Suppose you find that, indeed, the instruction you expected (from looking at the microtest source), an `addi`, is writing to `x3`, but it is producing the wrong value. The next step is to find where that value is coming from—what is the instruction that produced the value `addi` is using as a source register? In this case, you can add `$display` statements to show the destination registers of all instructions in the pipeline. By doing this, you'd finally pinpoint the bug, e.g., Decode is not bypassing from the correct instruction.

Finally, an effective debugging strategy is to *use your single-cycle processor*, which already works correctly, as a reference for correct operation. You can print the PCs and register values written by each instruction in both your single-cycle and pipelined processors, then compare both traces. This will *directly show you where execution starts going awry* in your pipelined processor.

test.py and debugging: The test harness, `test.py`, works by capturing the output of the simulation, so you will not see `$display` statements when using it. To see your `$display` output, you can run `./Processor`. This will simulate the processor on the last program that `test.py` tested, showing you its full output.

If you want to understand `test.py` in more depth or want to run other programs that `test.py` does not run, the simulation works as follows: at the beginning of the simulation, the simulator initializes `MainMemory` with the contents of file `mem.vmh`. This is how the processor simulates different programs. `test.py` simply copies the `.vmh` file from each test (under the `sw/` directory) into `mem.vmh`, then calls `./Processor` to run the simulation. `test.py` leaves `mem.vmh` to the last test executed.