Computer Sciences Department
University of Wisconsin-Madison
CS/ECE 552 – Introduction to Computer Architecture
Project Description

This document combines all of the information about the project (except for the Final Report template) into a single document. Broadly it has four sections: Overview, Project Phases, Microarchitecture Description, and ISA Description.

Overview

1. Summary

The CS/ECE 552 term project is the complete functional design of a microprocessor called the WISC-F22. All components of your design will be written in Verilog. As with the course homework assignments, the CS/ECE 552 Verilog restrictions apply, and all final code is expected to pass the Vcheck program.

Phases 1 of the project will be completed individually. For phases 2 and 3 you may complete them either individually or with up to two partners. Update group information document.

The specifics of the microarchitecture and WISC-F22 architecture are found in separate documents and will also be posted on the course website.

The project will progress in several distinct stages. Some of these stages are enforced through grading deadlines; others are not. The deadlines are:

Date	**Project Component**
October 12th	Design Review (2% of total grade)
October 27th	Form Project Group for phase 2, update group information document
October 30th	Phase #1 Unpipelined (8% of total grade)
November 12th	Phase #2 Pipelined with Perfect Memory (10% of total grade)
N/A	Phase #2.1
N/A	Phase #2.2
November 26th	Phase #2.3 Caching (5% of total grade)
December 10th	Phase #3 (10% of total grade)
December 12th	Extra Credit

Each stage of the design makes the processor progressively more complicated. For your own benefit, it is strongly recommended that you not proceed to a new stage before you are confident the current stage is working to specification. Debugging errors in a complex design is much harder. It is almost always better to test smaller, simpler components first.

Many of the Verilog problems in the homework assignments were designed to be compatible with the project. Please feel free to reuse these modules (of course, fixing any errors first!).

In addition to the previous homework problems, you will be provided with several reusable modules that you can use in your design. Most of these are Verilog implementations of memory system components. Please note that these files do ***not*** follow the CS/ECE 552 Verilog restrictions, so don't include them when you run Vcheck (and do not worry if they don't pass vcheck). To get the complete collection of the files you will need for all stages of the project, download the project tarfile:
`/u/s/w/swamit/public/html/courses/cs552/fall2022/handouts/verilog_code/project.tgz`

An assembler for the WISC-F22 ISA is provided for your use. Sample test programs are also provided, although you are strongly encouraged to write custom tests to augment these. Be aware that these test programs were written for a slightly different ISA specification and therefore may not work as advertised. It will be your job as diligent designers to determine if unexpected behavior occurs due to a bug in your design or as the result of the change in ISA. See the course website for a description of how to use the assembler.

2. IMPORTANT NOTES

- Start early: This project is designed to take a considerable amount of time.
- Plan ahead: You may find that the instructor, TAs, and peer mentors will be very inaccessible the night before a deadline.
- Ask questions: If you are getting stuck on some problem ask for help. Ask me, the TAs, the peer mentors, or your classmates.
- Functionality: Getting a working design is of paramount importance. Optimizations, clock-speed and bonus questions come 2nd. First make sure your design works!

If you finish really early, you will get the opportunity to earn extra credit by adding extra features to your processor, synthesizing your design. Additional details are available in the Extra Credit document.

If you are able to complete this project without the unnecessary stress that procrastination imposes, it is our belief that you will find this to be a highly rewarding experience.

All of the files you will need are included in the project tar file.

Finally, a reminder of some important documents:

1. Follow the instructions on ModelSim Setup Tutorial to get your environment setup.

2. Read the Command-line Verilog Simulation Tutorial. Additional references are on course website.
3. Read the Verilog Cheat sheet and Verilog rules pages. Everything you need to know about Verilog are in these documents.
4. Read the Verilog file naming conventions and Verilog file naming convention checking webpages and adhere to those conventions. We will be checking this in your submissions to ensure you followed the rules.
5. Read the Verilog rules checking page on the course website and adhere to the conventions. This page also provides information on how you can check that your files conform to these rules.
6. Read the Handin Verification page on the course website. You will need to run this before submitting your answers.

Project Phases

This project has roughly six stages of development with several deadlines/demos along the way:

1. You will first build a single cycle non-pipelined processor with a highly idealized memory.
2. Your processor can then be pipelined into 5 distinct stages but while still using a highly idealized memory.
3. The memory will then be transitioned to using a more realistic banked memory module that cannot respond to requests in a single cycle.
4. A cache can then be implemented that can be used to improve the now degraded memory performance.
5. Once the cache has been fully verified it can be incorporated into the full processor.
6. Optimizations can then be added for additional processor performance.

More formally:

A. Design review (2% of project grade)

Each student should create a complete hand-drawn (or drawn with the aid of a graphing program like OpenOffice draw) schematic of an unpipelined WISC-F22 implementation. Each module, bus, and signal should be uniquely labeled. The schematic should be hierarchical so that the top-level design contains only empty shells for each planned submodule. In general, there will be a one-to-one mapping of modules in your schematic to the modules you will eventually write in Verilog. The textbook pipeline diagram(s) is a good starting point but there are many differences between it and the ISA for this project. You will need to look at the class ISA and make sure to adapt it for that.

Your schematic should explicitly include modules for the Fetch, Decode, Execute, Memory, and Writeback, and the code for these modules will go into the provided shell modules. While explicitly drawing the pipeline stages that correspond to these modules is not required in the schematic, you should still design with a pipeline in mind. It is a good idea to place modules near their final location in the pipelined design.

B. Phase #1 – Single-Cycle, Unpipelined Design (8% of total grade)

All of the files you will need for the project are in a project tar file (`/u/s/w/swamit/ /courses/cs552/Fall2022/handouts/verilog_code/project.tgz`).

To start, you should do a single-cycle, non-pipelined implementation. Figure 4.33 on page 299, Figure 4.35 on page 301, and Figure 4.36 on page 303 of the text are good places to start.

For this stage, you will use the single cycle perfect memory. Since you will need to fetch instructions as well as read or write data in the cycle, use two memories -- one for instruction memory and one for data.

Your design should be running the full WISC-F22 instruction set, except for the extra-credit instructions. It should use the single-cycle memory model. You should run vcheck and your files must all pass vcheck.

To verify your design works, you will run a set of programs on your processor using the wsrun.pl script (check the verification and simulation page for more info), show that your processor works on the test programs (full list in Test Programs page). You should run the tests under the following three categories:

1. Simple instruction tests
2. Complex tests for demo1
3. Random tests for demo1
    1. rand_simple
    2. rand_complex
    3. rand_ctrl
    4. rand_mem

Your grade for phase 1 will be scaled based on how many of these tests you pass, although some of the categories may be weighted more than others.

Use the *-list* file to run each of the categories of test. When you run wsrun.pl with the -list option, it will generate a file called *summary.log*, which looks like below:

```
add_0.asm  SUCCESS  CPI:1.3  CYCLES:12  ICOUNT:9  IHITRATE:  0
DHITRATE: 0
add_1.asm  SUCCESS  CPI:1.7  CYCLES:7  ICOUNT:4  IHITRATE:  0
DHITRATE: 0
add_2.asm  SUCCESS  CPI:1.7  CYCLES:7  ICOUNT:4  IHITRATE:  0
DHITRATE: 0
```

SUCCESS means the test passed. Run all the categories and rename the summary.log files as shown below:

1. Simple instruction tests: `instTests.summary.log`
2. Complex tests for demo1: `complex_demo1.summary.log`
3. Random tests for demo1
    1. rand_simple: `rand_simple.summary.log`
    2. rand_complex: `rand_complex.summary.log`
    3. rand_ctrl: `rand_ctrl.summary.log`
    4. rand_mem: `rand_mem.summary.log`

You can use the script `run-phase1-all.sh` to run all the required tests. A brief note of caution: if your all.list has > 1000 failures, wsrun will stop running tests from that list – please make sure to check your logs if you have many failures.

Running all the tests will take 10-15 minutes. So plan ahead!

The log files MUST have the exact name. These are the log files produced by running wsrun.pl -list with the all.list file for each of those sets of benchmarks. You will have to rename summary.log manually into these names. If your handed in code does not follow this convention, it will not be accepted and you will receive a zero for this demo. If in doubt about what to submit, as TAs *before* the deadline and double-check. Alternatively, you can/should use the verification script (`/u/s/w/swamit/public/html/courses/cs552/fall2022/handouts/scripts/project/phase1/verify_submission_format.sh`) to ensure that your files are in the correct place.

You should do rigorous testing and verification and should try to have zero failures on the other categories. It is ok to have a very small number of failures - but for every failure you must know the reason. You will submit your design electronically, which will be graded automatically. The instructor will then schedule one-on-one appointments with teams that have exhibited a large number of failures.

Everything due at 1159 PM Central Time on 30th October

Electronic submission instructions

Submit your *demo1.tgz* on Canvas containing the following directories [**tar -czvf demo1.tgz demo1**] where demo1 is the directory with the summary and verilog sub-directories. These sub-directories will already exist if you do your work in the demo1 directory from the original tar that was provided.

1. The sub-directories should contain the following files:
    1. **verilog/** containing all Verilog and Vcheck files. Please copy over ALL necessary files, your processor should be able compile and run

with files from this directory alone. It should also contain vcheck files for all of your Verilog files.

2. **summary/** containing the 6 summary.log files.

If the summary.log files are missing, you will automatically get **zero points**.

Please do not submit your __work directory as it will significantly inflate your submission size and make it hard for us to grade quickly.

Verifying Your Phase 1 Handin

You should also verify your submission has these files by running the verification script:

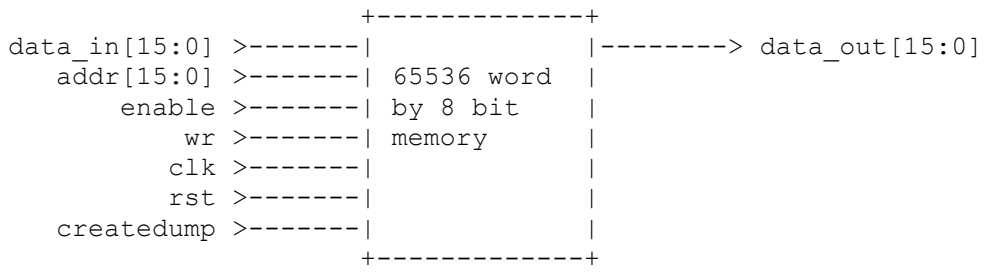
/u/s/w/swamit/public/html/courses/cs552/fall2022/handouts/scripts/project/phase1/verify_submission_format.sh demo1.tgz

Additional details, including a demo of this script are available here.

Single-Cycle Processor Memory Specification

Since your single-cycle design must fetch instructions as well as read or write data in the same cycle, you will want to use two instances of this memory -- one for data, and one for instructions.

Note: You should instantiate this memory module twice. One instance will serve as the instruction memory while the other will serve as the data memory. Note that the program binary should be loaded into both instances. This will automatically be done (without any additional effort on your part) if you use the same module definition for both instances



During each cycle, the "enable" and "wr" inputs determine what function the memory will perform:

Enable	Wr	Function	data_out
0	X	No operation	0
1	0	Read	M[addr]

1	1	Write data_in	0

During a read cycle, the data output will immediately reflect the contents of the address input and will change in a flow-through fashion if the address changes. For writes, the "wr", "addr", and "data_in" signals must be stable at the rising edge of the clock ("clk").

The memory is initialized from a file. The file name is "loadfile_all.img", but you may change that in the Verilog source to any file name you prefer. The file is loaded at the first rising edge of the clock during reset. The simulator will look for the file in the same location as your .v files (or the directory from which you run `wsrun.pl`. The file format is:

@0
12
12
12
12

where "*@0*" specifies a starting address of zero, and "*12*" represents any 2-digit hex number. Any number of lines may be specified, up to the size of the memory. The assembler will produce files in this format.

At the end of the simulation, the memory can produce a dumpfile so that you may determine what has been written to the memory. When "createdump" is asserted at the rising edge of the clock, the memory will create a file named "dumpfile" in the mentor directory. You may want to use the decode of the "halt" instruction to assert "createdump" for a single cycle.

When a dumpfile is created, it will contain locations zero through the highest address that has been modified with a write cycle (not the highest address loaded from the loadfile). The format is:

0000 1234
0001 1234
0002 1234

Examining the source file *memory2c.v*, several possible changes should be obvious. The names of the files may be changed. The format of the dumpfile may be changed by modifying the $fdisplay statement; the syntax is very similar to C's fprintf statement. The starting and ending addresses to dump may be modified in the "for" statement. The only thing that cannot be modified is the format of the loadfile; that is built-in.

When you have two copies of the memory, for instructions and data, you may want to let both memories load the same loadfile, but only have the data memory generate a dumpfile.

The way to load programs for your processor is to use the assembler, create the memory dump. Name the memory dump, loadfile_all.img and copy this into the directory where memory2c.v is present.

C. Phase #2 – Pipelined Design with Perfect Memory (10% of total grade)

At this point, the pipelined version of your design needs to be running correctly, but no optimizations are needed yet. Correctly means that it must detect and do the right thing on pipeline hazards (e.g., stall). You will still use the single-cycle memory model. We will follow similar protocol as demo1. We will run your tests and ask teams with any failures to sign up for a demo with us.

You must write at least two additional hand tests to test pipelining. Writing more will help simplify debugging. If you write additional tests, include them in verification/mytests/.

You must create and submit a document which should explain the behavior of your processor for the perf-test-dep-ldst.asm test. Please use the following format:

Cycle	*Instruction Retired*	*Reason*
1		
2		
Etc.		

The instruction retired (we will discuss what retired means in class, but essentially for our 5-stage in-order processor it means what is in Write Back that cycle) would either be one of the instructions from the test program or a "NOP" if dependencies necessitate any stall cycles. The reason column would explain why a stall was needed in that instance. Please include this information in a PDF file titled instruction_timeline.pdf.

Everything due at 11:59 PM Central time on November 12th.

Electronic submission instructions

Submit your *demo2.tgz* file on Canvas containing the following directories [**tar -czvf demo2.tgz demo2**] where demo2 is the directory containing the verification and Verilog sub-directories. These sub-directories will already exist if you do your work in the *demo2* directory from the original tar that was provided.

1. verilog/ containing all Verilog and Vcheck files. Please copy over ALL necessary files, your processor should be able compile and run with files from this directory alone.
2. verification/mytests/ The assembly (.asm) files that you have written.
3. verification/results/ Run all the categories and rename the summary.log files as shown below:

1. Simple instruction tests: `inst_tests.summary.log`
2. Complex tests from demo1: `complex_demo1.summary.log`
3. Random tests for demo1
    1. rand_simple: `rand_simple.summary.log`
    2. rand_complex: `rand_complex.summary.log`
    3. rand_ctrl: `rand_ctrl.summary.log`
    4. rand_mem: `rand_mem.summary.log`
4. Random tests for demo2: `complex_demo2.summary.log`
5. Your code results: `mytests.summary.log`

4. verification/instruction_timeline.pdf - The timeline you have created for the retiring instructions of perf-test-dep-ldst.asm.

You can use the script `run-phase2-almostAll.sh` to run almost all the required tests. It will create all these summary.log files **except for the mytests.summary.log** summary file (hence "almost all"). A brief note of caution: if your all.list has > 1000 failures, wsrun will stop running tests from that list – please make sure to check your logs if you have many failures.

Running all the tests will take 10-15 minutes. So plan ahead!

The log files MUST have the exact name. These are the log files produced by running wsrun.pl -list with the all.list file for each of those sets of benchmarks. You will have to rename summary.log manually into these names (unless you use the above script). If your handed in code does not follow this convention, it will not be accepted and you will receive a zero for this demo. If in doubt about what to submit, email the TAs *before* the deadline and double-check.

Verifying Your Phase 2 Handin

You should also verify your submission has the correct files by running the verification script:

/u/s/w/swamit/public/html/courses/cs552/Fall2022/handouts/scripts/project/phase2/verify_submission_format.sh demo2.tgz

Additional details, including a demo of this script are available here.

The next few deadlines are minor changes to your processor and you should plan on doing them very quickly. **They are optional. No print or electronic submissions required.** The due dates are simply suggestions. Make sure all demo2 tests pass at these phases.

D. Phase #2.1 – Pipelined Design with Aligned Memory (0% of project grade)

No Submission required. **This is optional.**

At this step, replace the original single-cycle memory with the Aligned single cycle memory. This is a very similar module, but it has an "err" output that is generated on unaligned memory accesses. Your processor should halt when an error occurs. Verify your design. If you want to test, you can test with:

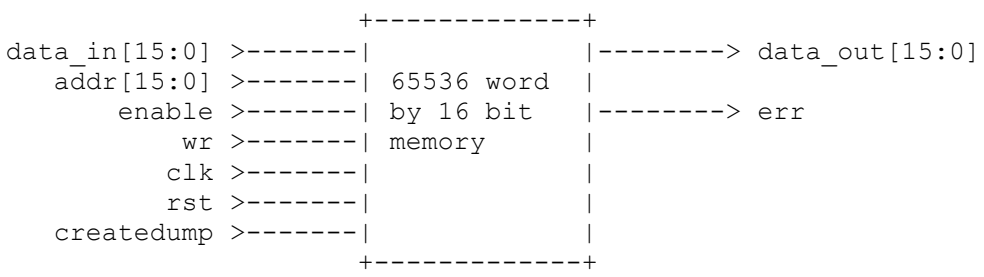
```
/u/s/w/swamit/public/html/courses/cs552/Fall2022/handouts/testprograms/public/unaligned.list
```

Aligned Single-Cycle Memory Specification

Before building your cache, you should use this memory to update and test your processor's interface to properly handle unaligned accesses. Many processors (e.g., MIPS) are byte addressable, but require that all accesses be aligned to their natural size (i.e., byte loads and stores can access any individual byte, but word loads and stores must access aligned words). Since your processor only has word loads and stores, this is pretty simple (to support byte stores, the memory would need byte write enable signals; to support byte loads, either the memory or the processor needs a mux to select the right byte). Notice that the memory always returns aligned data even on a misaligned load, and memory does not store anything on a misaligned store.

The Verilog source (memory2c_align.v) was included in the project tar.

Since your single-cycle design must fetch instructions as well as read or write data in the same cycle, you will want to use two instances of this memory -- one for data, and one for instructions.



During each cycle, the "enable" and "wr" inputs determine what function the memory will perform. On an unaligned access err is set.

enable	Wr	Function	data_out	err
0	X	No operation	0	0
1	0	Read	M[addr]	0
1	1	Write	Write data_in	0
1	X	X	if (addr[0]) set	1

E. Phase # 2.2 – Pipelined Design with Stalling Memory: 1 week after Phase #2 (0% of project grade)

No Submission required. **This is optional.**

At this step, replace the single cycle memory with the Stalling memory. This is a very similar module, but has stall and done signals similar to the cache you will build. Your pipeline will need to stall to handle these conditions. Verify your design.

- **Instruction memory:** First replace your instruction memory module with this stalling memory, keep your data memory module the same (i.e. aligned perfect memory from previous step). Verify your design. This will be easier to debug, as only module's behavior has changed.
- **Data memory:** Now, replace your data memory module alone with this stalling memory, revert your instruction memory module back to the aligned perfect memory. Verify your design. This will be easier to debug, as only module's behavior has changed.
- **Instruction and Data memory:** Now change both instruction and data memories to the stalling memory design. Verify your design.

Stalling Memory Specification

This module has an interface identical to the cache interface in `mem_system_hier.v`.

Examining the source file `stallmem.v`, you will see "rand_pat", a shift register which controls the "ready" output. This is a random 32-bit number. You can change its value by changing the seed used for random number of generation. You can do this by passing in "-seed" to wsrun.pl. For example:

```
wsrun.pl -pipe -seed 45 -prog foo.asm proc_hier_pbench *.v
```

If you are executing from inside ModelSim with *run -all* or using a testbench of your own for preliminary testing, you can pass in the seed, by adding the string "+seed=<value>" to the *vsim* command. Or simply edit *stallmem.v* and set the seed to a different value.

F. Cache Demo - Working two-way set-associative cache (5% of total grade)

All information on the cache design, including submission instructions is posted on the cache design page.

Due November 26th at 11:59 PM on Canvas, no exceptions (except for late days).

G. Phase #3 (final demo) - Pipelined Multi-cycle Memory with Optimizations (30% of project grade)

Due December 10th at 11:59 PM on Canvas, no exceptions (except for late days).

At this final demo teams are expected to demonstrate the complete design to all specifications. This includes the following required items:

- Two-way set-associative caches with multi-cycle memory
- Register file bypassing
- Forwarding from beginning of the MEM stage to beginning of EX stage (EX → EX forwarding)
- Forwarding from beginning of the WB stage to the beginning of the EX stage (MEM → EX forwarding)
- Branches predicted not taken

When implementing your forwarding remember to not implement it in such a way that the work of multiple pipeline stages occurs along a single combinational path or you will lose credit for violating the pipelining. The IPC of your processor will also account for a small portion of this demo grade so you should try to eliminate excess stall cycles where ever present.

Hand in format will be similar to demo 1 and 2.

Electronic submission instructions Submit a single demo3.tgz file containing the following directories [**tar -czvf demo3.tgz demo3**] where demo3 is the directory that contains the verification and verilog sub-directories. These sub-directories will already exist if you do your work in the demo3 directory from the original tar file that was provided.

1. verilog/ containing all Verilog and Vcheck files. Please copy over ALL necessary files, your processor should be able compile and run with files from this directory alone.
2. verification/mytests/ The assembly (.asm) files that you have written. (at least two tests)
3. verification/results/ Run all test programs and rename the summary.log files as listed below:
    1. perf.summary.log
    2. complex_demofinal.summary.log
    3. rand_final.summary.log
    4. rand_ldst.summary.log
    5. rand_idcache.summary.log
    6. rand_icache.summary.log
    7. rand_dcache.summary.log
    8. complex_demo1.summary.log
    9. complex_demo2.summary.log
    10. rand_complex.summary.log
    11. rand_ctrl.summary.log
    12. inst_tests.summary.log

You can use the script `run-final-all.sh` to run all the required tests. It will create all these summary.log files. A brief note of caution: if your all.list has > 1000 failures, wsrun

will stop running tests from that list – please make sure to check your logs if you have many failures.

Running all the tests will take about 40 minutes. So plan ahead!

We will electronically grade this submission, similar to the prior phases.

As mentioned previously, you can submit each phase up to 2 days late with a 10% penalty per day, along with having 5 "free" late days for the project across the semester.

- If your design has known failures, then bring to the demo a written short explanation for as many failures as you can track down. This will exponentially increase the points you will get, compared to simply showing up and saying we don't know the reason for the failures.
- If your entire design does not work, then you may show us a demo of a partially complete processor. So, in your best interest, snapshot working parts of your design as you add more functionality (or, if you are using git, I strongly recommend make commits and/or branches for each phase). For example, you may show us any one of the following, if your full pipeline+cache does not work.
    - Stalling instruction memory alone
    - Stalling data memory alone
    - Stalling inst+data memory
    - Direct-mapped instruction memory alone
    - Direct-mapped data memory alone
    - Direct-mapped inst+data memory
    - 2-way instruction memory alone
    - 2-way data memory alone
    - 2-way inst+data memory

You must be present and are expected to explain and answer questions about the whole design. Answering a question with: "I have no idea" is a failing answer. You must (at least) be able to answer: "I think it works in the following way....".

Verifying Your Phase 3 Handin

You should also verify your submission has the correct files by running the verification script:

```
/u/s/w/swamit/public/html/courses/cs552/fall2022/handouts/scripts/project/phase3/verify_submission_format.sh demo3.tgz
```

Additional details, including a demo of this script are available here.

WISC-F22 Microarchitecture Specification

Next we describe the microarchitecture, including register file specifications, memory system organization, etc. you will use for your CS/ECE 552 project. The WISC-F22 architecture that you will design for the final project shares many resemblances to the MIPS R2000 described in the text. The major differences are a smaller instruction set and 16-bit words for the WISC-F22. Similarities include a load/store architecture and three fixed-length instruction formats.

1. Registers

There are eight user registers, R_0-R_7. Unlike the MIPS R2000, R_0 is ***not*** always zero. Register R_7 is used as the link register for JAL or JALR instructions. The program counter is separate from the user register file. If you chose to implement exceptions for extra credit, a special register named EPC is used to save the current PC upon an exception or interrupt invocation.

2. Memory System

The WISC-F22 is a Harvard architecture, meaning instructions and data are located in different physical memories. It is byte-addressable, word aligned (where a word is 16 bits long – note that this is different from some of the examples in class), and big-endian. The final version of the WISC-F22 will include a multi-cycle memory and one level of cache. However, initial versions of the machine will contain a single cycle memory. See the project deadlines for more details.

The WISC-F22 cache replacement policy is deterministic. See the cache module description for an outline of the algorithm you must use.

NOTE: For phase1 and phase2, you will work with a simplified memory model which supports un-aligned accesses.

3. Pipeline

The final version of the WISC-F22 contains a five-stage pipeline identical to the MIPS R2000. The stages are:

1. Instruction Fetch (IF)
2. Instruction Decode/Register Fetch (ID)
3. Execute/Address Calculation (EX)
4. Memory Access (MEM)
5. Write Back (WB)

See Figure 4.33 on page 299, Figure 4.35 on page 301, or Figure 4.36 on page 303 of the text for good starting points.

4. Optimizations

Your goal in optimizations is to reduce the CPI of the processor or the total cycles taken to execute a program. While the primary concern of the WISC-F22 is correct functionality, the architecture must still have a reasonable clock period. Therefore, you may not have more than one of the following in series during any stage:

- register file
- memory or cache
- 16-bit full adder
- barrel shifter

You may implement any type of optimization to reduce the CPI (as long as it's a valid optimization). The required optimizations are:

- Register file bypassing
- There are two register forwarding paths in the WISC-F22:
    - Forwarding from beginning of the MEM stage to beginning of EX stage (EX → EX forwarding)
    - Forwarding from beginning of the WB stage to the beginning of the EX stage (MEM → EX forwarding)
- All branches should be predicted not-taken. This means that the pipeline should continue to execute sequentially until the branch resolves, and then squash instructions after the branch if the branch was actually taken.

5. Exceptions: extra credit

Exception handling is extra credit. If you choose not to implement exception handling, an illegal instruction should be treated as a NOP.

`IllegalOp` is the only defined exception in the WISC-F22 architecture – i.e., if you run the wiscalculator it will show the siic instruction as an "IllegalOp." If you implement exception support, you will find that the exception handler is invoked when the opcode of the currently executing instruction is not a recognized member of the ISA (e.g., a siic instruction). Upon finding an illegal opcode, your processor should save the current PC into the reserved register EPC and then load address 0x02, which is the location of the IllegalOp exception handler. Note that if you choose to implement exceptions, address 0x00 must be a jump to the start of the main program.

The exception handler itself need not be complex. At a minimum it should load the value 0xBADD into R_7 and then use/call the RTI instruction to return to the address specified by the EPC. Several provided tests do exactly this.

WISC-F22 ISA Specification

1. Instruction Summary

(**KEY**: sss = rs, ddd = rd, ttt = rt, iii* = immediate)

Instruction Format	Syntax	Semantics

00000 xxxxxxxxxxx	HALT	Cease instruction issue, dump memory state to file
00001 xxxxxxxxxxx	NOP	None
01000 sss ddd iiiii	ADDI Rd, Rs, immediate	Rd <- Rs + I(sign ext.)
01001 sss ddd iiiii	SUBI Rd, Rs, immediate	Rd <- I(sign ext.) - Rs
01010 sss ddd iiiii	XORI Rd, Rs, immediate	Rd <- Rs XOR I(zero ext.)
01011 sss ddd iiiii	ANDNI Rd, Rs, immediate	Rd <- Rs AND ~I(zero ext.)
10100 sss ddd iiiii	ROLI Rd, Rs, immediate	Rd <- Rs <<(rotate) I(lowest 4 bits)
10101 sss ddd iiiii	SLLI Rd, Rs, immediate	Rd <- Rs << I(lowest 4 bits)
10110 sss ddd iiiii	RORI Rd, Rs, immediate	Rd <- Rs >>(rotate) I(lowest 4 bits)
10111 sss ddd iiiii	SRLI Rd, Rs, immediate	Rd <- Rs >> I(lowest 4 bits)
10000 sss ddd iiiii	ST Rd, Rs, immediate	Mem[Rs + I(sign ext.)] <- Rd
10001 sss ddd iiiii	LD Rd, Rs, immediate	Rd <- Mem[Rs + I(sign ext.)]
10011 sss ddd iiiii	STU Rd, Rs, immediate	Mem[Rs + I(sign ext.)] <- Rd Rs <- Rs + I(sign ext.)
11001 sss xxx ddd xx	BTR Rd, Rs	Rd[bit i] <- Rs[bit 15-i] for i=0..15
11011 sss ttt ddd 00	ADD Rd, Rs, Rt	Rd <- Rs + Rt
11011 sss ttt ddd 01	SUB Rd, Rs, Rt	Rd <- Rt - Rs
11011 sss ttt ddd 10	XOR Rd, Rs, Rt	Rd <- Rs XOR Rt
11011 sss ttt ddd 11	ANDN Rd, Rs, Rt	Rd <- Rs AND ~Rt
11010 sss ttt ddd 00	ROL Rd, Rs, Rt	Rd <- Rs << (rotate) Rt (lowest 4 bits)
11010 sss ttt ddd 01	SLL Rd, Rs, Rt	Rd <- Rs << Rt (lowest 4 bits)
11010 sss ttt ddd 10	ROR Rd, Rs, Rt	Rd <- Rs >> (rotate) Rt (lowest 4 bits)
11010 sss ttt ddd 11	SRL Rd, Rs, Rt	Rd <- Rs >> Rt (lowest 4 bits)
11100 sss ttt ddd xx	SEQ Rd, Rs, Rt	if (Rs == Rt) then Rd <- 1 else Rd <- 0
11101 sss ttt ddd xx	SLT Rd, Rs, Rt	if (Rs < Rt) then Rd <- 1 else Rd <- 0
11110 sss ttt ddd xx	SLE Rd, Rs, Rt	if (Rs <= Rt) then Rd <- 1 else Rd <- 0
11111 sss ttt ddd xx	SCO Rd, Rs, Rt	if (Rs + Rt) generates carry out then Rd <- 1 else Rd <- 0
01100 sss iiiiiiii	BEQZ Rs, immediate	if (Rs == 0) then PC <- PC + 2 + I(sign ext.)
01101 sss iiiiiiii	BNEZ Rs, immediate	if (Rs != 0) then PC <- PC + 2 + I(sign ext.)

01110 sss iiiiiiii	BLTZ Rs, immediate	if (Rs < 0) then PC <- PC + 2 + I(sign ext.)
01111 sss iiiiiiii	BGEZ Rs, immediate	if (Rs >= 0) then PC <- PC + 2 + I(sign ext.)
11000 sss iiiiiiii	LBI Rs, immediate	Rs <- I(sign ext.)
10010 sss iiiiiiii	SLBI Rs, immediate	Rs <- (Rs << 8) | I(zero ext.)
00100 ddddddddddd	J displacement	PC <- PC + 2 + D(sign ext.)
00101 sss iiiiiiii	JR Rs, immediate	PC <- Rs + I(sign ext.)
00110 ddddddddddd	JAL displacement	R7 <- PC + 2 PC <- PC + 2 + D(sign ext.)
00111 sss iiiiiiii	JALR Rs, immediate	R7 <- PC + 2 PC <- Rs + I(sign ext.)
00010	siic Rs	produce IllegalOp exception. Must provide one source register.
00011 xxxxxxxxxxx	NOP / RTI	PC <- EPC

2. Formats

WISC-F22 supports instructions in four different formats: J-format, 2 I-formats, and the R-format. These are described below.

2.1 J-format

The J-format is used for jump instructions that need a large displacement.

J-Format

5 bits	11 bits
Op Code	Displacement

Jump Instructions

The Jump instruction loads the PC with the value found by adding the PC of the next instruction (PC+2, not PC+4 as in MIPS) to the **sign-extended** displacement.

The Jump-And-Link instruction loads the PC with the same value and also saves the address of the next sequential instruction (i.e., PC+2) in the link register R_7.

The syntax of the jump instructions is:

- `J displacement`

- `JAL displacement`

2.2 I-format

I-format instructions use either a destination register, a source register, and a 5-bit immediate value; or a destination register and an 8-bit immediate value. The two types of I-format instructions are described below.

I-format 1 Instructions

I-format 1

5 bits	3 bits	3 bits	5 bits
Op Code	R_s	R_d	Immediate

The I-format 1 instructions include XOR-Immediate, ANDN-Immediate, Add-Immediate, Subtract-Immediate, Rotate-Left-Immediate, Shift-Left-Logical-Immediate, Rotate-Right-Immediate, Shift-Right-Logical-Immediate, Load, Store, and Store with Update.

The **ANDNI** instruction loads register R_d with the value of the register R_s AND-ed with the **one's complement** of the zero-extended immediate value. (It may be thought of as a bit-clear instruction.) **ADDI** loads register R_d with the sum of the value of the register R_s plus the **sign-extended** immediate value. **SUBI** loads register R_d with the result of subtracting register R_s from the **sign-extended** immediate value. (That is, immed - R_s, **not** R_s - immed.) Similar instructions have similar semantics, i.e. the logical instructions have zero-extended values and the arithmetic instructions have sign-extended values.

For Load and Store instructions, the effective address of the operand to be read or written is calculated by adding the value in register R_s with the **sign-extended** immediate value. The value is loaded to or stored from register R_d. The **STU** instruction, Store with Update, acts like Store but also writes R_s with the effective address.

The syntax of the I-format 1 instructions is:

- `ADDI` R_d, R_s, immediate
- `SUBI` R_d, R_s, immediate
- `XORI` R_d, R_s, immediate
- `ANDNI` R_d, R_s, immediate
- `ROLI` R_d, R_s, immediate
- `SLLI` R_d, R_s, immediate
- `RORI` R_d, R_s, immediate
- `SRLI` R_d, R_s, immediate
- `ST` R_d, R_s, immediate
- `LD` R_d, R_s, immediate

- `STU Rd, Rs, immediate`

I-format 2 Instructions

I-format 2

5 bits	3 bits	8 bits
Op Code	Rs	Immediate

The Load Byte Immediate instruction loads R_s with a sign-extended 8-bit immediate value.

The Shift-and-Load-Byte-Immediate instruction shifts R_s 8 bits to the left and replaces the lower 8 bits with the immediate value.

The format of these instructions is:

- `LBI Rs, signed immediate`
- `SLBI Rs, unsigned immediate`

The Jump-Register instruction loads the PC with the value of register R_s + signed immediate. The Jump-And-Link-Register instruction does the same and also saves the return address (i.e., the address of the JALR instruction plus one) in the link register R_7. The format of these instructions is

- `JR Rs, immediate`
- `JALR Rs, immediate`

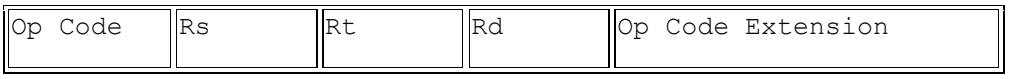
The branch instructions test a general-purpose register for some condition. The available conditions are: equal to zero, not equal to zero, less than zero, and greater than or equal to zero. If the condition holds, the signed immediate is added to the address of the next sequential instruction and loaded into the PC. The format of the branch instructions is

- `BEQZ Rs, signed immediate`
- `BNEZ Rs, signed immediate`
- `BLTZ Rs, signed immediate`
- `BGEZ Rs, signed immediate`

2.3 R-format

R-format instructions use only registers for operands.
R-format

5 bits	3 bits	3 bits	3 bits	2 bits



ALU and Shift Instructions

The ALU and shift R-format instructions are similar to I-format 1 instructions, but do not require an immediate value. In each case, the value of R_t is used in place of the immediate. No extension of its value is required. **In the case of shift instructions, all but the 4 least-significant bits of R_t are ignored**.

The ADD instruction performs signed addition. The SUB instruction subtracts R_s from R_t. (*Not* R_s - R_t.) The set instructions SEQ, SLT, SLE instructions compare the values in R_s and R_t and set the destination register R_d to 0x1 if the comparison is true, and 0x0 if the comparison is false. SLT checks for R_s less than R_t, and SLE checks for R_s less than or equal to R_t. (R_s and R_t are two's complement numbers.) The set instruction SCO will set R_d to 0x1 if R_s plus R_t would generate a carry-out from the most significant bit; otherwise it sets R_d to 0x0. The Bit-Reverse instruction, BTR, takes a single operand R_s and copies it to R_d, but with a left-right reversal of each bit; i.e. bit 0 goes to bit 15, bit 1 goes to bit 14, etc.

The syntax of the R-format ALU and shift instructions is:

- `ADD` R_d, R_s, R_t
- `SUB` R_d, R_s, R_t
- `XOR` R_d, R_s, R_t
- `ANDN` R_d, R_s, R_t
- `ROL` R_d, R_s, R_t
- `SLL` R_d, R_s, R_t
- `ROR` R_d, R_s, R_t
- `SRL` R_d, R_s, R_t
- `SEQ` R_d, R_s, R_t
- `SLT` R_d, R_s, R_t
- `SLE` R_d, R_s, R_t
- `SCO` R_d, R_s, R_t
- `BTR` R_d, R_s

3. Special Instructions

Special instructions use the R-format. The HALT instruction halts the processor. The HALT instruction and all older instructions execute normally, but the instruction after the halt will never execute. The PC is left pointing to the instruction directly after the halt.

The No-operation instruction occupies a position in the pipeline but does nothing.

The syntax of these instructions is:

- `HALT`
- `NOP`

The SIIC and RTI instructions are extra credit and can be deferred for later. They will be not tested until the final demo.

The SIIC instruction is an illegal instruction and should trigger the exception handler. EPC should be set to PC + 2, and control should be transferred to the exception handler which is at PC 0x02.

The syntax of this instruction is:

- `SIIC Rs`

The source register name must be ignored. The syntax is specified this way with a dummy source register, to reuse some components from our existing assembler. The RTI instruction should remain equivalent to NOP until the rest of the design has been completed and thoroughly tested.

RTI returns from an exception by loading the PC from the value in the EPC register.

The syntax of this instruction is:

- `RTI`

Note that if you do not implement exception support, then you should treat RTI as a NOP. But, once you add this support, RTI will always load the PC from the value in the EPC register – i.e., it is no longer a NOP.

See the Part 4 in the Microarchitecture description for more information on optimizations.

Extra Credit

The extra credit components should be implemented atop a working 5-stage pipeline design with caches (Phase 3). Without a working baseline for the mandatory project requirement(s), no extra credit points will be awarded.

Extra Credit Components

Up to a total of 7 points (out of a total of 35 points for the course project) can be potentially obtained by successfully implementing different components of extra credit. The description of different components and their corresponding points are listed below. You will be capped at 7 points of extra credit though.

Note: for the extra credit portion, you must still abide by the 552 Verilog Rules and Filenaming Conventions. Moreover, you should implement these optimizations in a separate folder from your Phase 3 submission – i.e., keep a working copy of your Phase 3 submission separate from this.

Due Date: Your extra credit optimizations should be turned in by 11:59 PM on December 12th on Canvas.

Submission Requirements

1. You are required to submit a pdf document summarizing your extra-credit design and its differences from the baseline in terms of features, benefits and overheads.
2. You are required to develop one or more test cases, which clearly highlight the benefits of your optimized design, and the results should be shown in the submitted document.
3. You are also required to submit a tar'd or zipped file containing: all the Verilog files of your design, all testbenches used and any other support files.

Since there are many different extra credit optimizations, please explain in your accompanying PDF what your directory structure is.

Extra Credit Optimizations

1. **Branch Decisions in Decode (0 – 1 point)**: As discussed in class, doing your decisions for branches in decode will reduce the number of instructions that need to be flushed and likely improve CPI. You will need to update your forwarding, stall, and flush logic accordingly when you make this optimization. After making this optimization, you should see that performance improves for the tests that use mispredicted branches.

2. **Additional Forwarding Paths (0 – 1 point)**: For Phase 3 you are only required to implement forwarding from beginning of the MEM stage to beginning of EX stage (i.e., X → X forwarding) and forwarding from beginning of the WB stage to the beginning of the EX stage (i.e., M → X forwarding). Thus, there are additional forwarding paths you may implement to improve you CPI:

- Forwarding from the beginning of the WB stage to the beginning of the MEM stage (i.e., M → M forwarding)
- (if you do branch decisions in decode) Forwarding from the beginning of the MEM stage to the beginning of the ID stage (i.e., X → D forwarding) and the beginning of the WB stage to the beginning of the ID stage (i.e., M → D forwarding, assuming you are implementing branches in ID)

Implementing each of these forwarding paths is worth ½ point of extra credit, with a max of 1 point.

3. **Cache Replacement Policy (0-1 points for instruction cache, 0-2 points for data cache)**: Increase the performance of your cache design by using LRU (or another) superior replacement policy. To test the improvement, run all of the tests from Phase 3 and compare the CPI to that of your design before this optimization (all of the tests that are expected to pass for Phase 3 should still pass after this optimization).

4. **Employing "critical word first" data reads from memory to cache (0-1 point for instruction cache, 0-2 points for data cache)**: to service waiting requests early, add support for servicing the requested (critical) word first, before the other words on the same cache line. This will require "hit under miss" behavior so that while your FSM is still filling a previous miss, subsequent instruction fetches, loads, and stores can continue as long as they are to different cache blocks.

You must check to see if they are to the same cache block and stall in those cases until the cache block is filled.

To test the improvement, run all of the tests from Phase 3 and compare the CPI to that of your design before this optimization (all of the tests that are excepted to pass for Phase 3 should still pass after this optimization).

5. **Dynamic branch prediction (0-2 points):** implement dynamic branch prediction to reduce the misprediction rate of your processor. To test the improvement, run all of the tests from Phase 3 and compare the CPI to that of your design before this optimization (all of the tests that are excepted to pass for Phase 3 should still pass after this optimization).

6. **Exception Handling (0–2 points)**: As discussed in the Microarchitecture Specification, exception handling is extra credit. If you choose not to implement exception handling, an illegal instruction should be treated as a NOP.

`IllegalOp` is the only defined exception in the WISC-F22 architecture. It is invoked when the opcode of the currently executing instruction is not a recognized member of the ISA. Upon finding an illegal opcode, the computer shall save the current PC into the reserved register EPC and then load address 0x02, which is the location of the IllegalOp exception handler. Note that if you choose to implement exceptions, address 0x00 must be a jump to the start of the main program.

After implementing exception handling, you should see that the exception handling tests now pass. To verify the correctness of your exception handling, you should run the tests in `/u/s/w/swamit/public/html/courses/cs552/fall2022/handouts/testprograms/public/exceptions.list`. The exception handler routines in these programs vary in complexity.

You should also write and submit at least one additional test that demonstrates the handling of exceptions. In this program, your exception handler routine does not need to be complex. At a minimum, it should load 0xBADD into R_7 and then call the RTI instruction.

Other optimizations (please discuss with instructor/TAs before proceeding) (0-2 points)

Note: Please talk to the TAs or Swamit before embarking on any of these extra-credit components so as to clearly establish requirements of the extra-credit component.