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

This project has roughly six stages of development with several 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. Form team (1% of project grade)

The project is done in groups of two. These groups should be formed no later than February 14th.

B. Design review (4% of project grade)

Each group needs to create a complete hand-drawn (or drawn with the aid of a graphing program like OpenOffice draw) schematic of an unpipelined WISC-SP20 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.

During the review, individual team members should be able to describe the datapath of any legal WISC-SP20 instruction using the schematic as a reference. Teams will also be expected to defend the design decisions that they make. You need to have thought through the control path and decode logic. It is not necessary to have done a complete table of signals, but if you have such a table with the control signal values for every instruction, that would be great.

Every group will sign up for a 10 minute design review slot on this Google Doc – first come, first served. Write each partner's last name (Column D) and your group number (Column C) in your timeslot. If none works, discuss with your classmates about a possible swap. If you still cannot find a timeslot that works, email both the TAs and Matt. All meetings will be in Matt’s office, 6369 CS.

Both partners are required to be present and are expected to explain and answer questions about the whole design. Answering a question with: "I have no idea, my partner did that" is a failing answer. You must (at least) be able to answer: "My partner implemented that, but it works in the following way....".

C. Phase #1 – Single-Cycle, Unpipelined Design (15% of project grade)

All of the files you will need for the project are in a project tar file (`/u/s/i/sinclair/public/html/courses/cs552/spring2020/project/project.tgz`) or on Github Classroom. You should download and untar this (or clone from Github) while getting started.

To start, you should do a single-cycle, non-pipelined implementation. Figure 4.35 on page 289 and Figure 4.36 on page 291 of the text are a good place 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-SP20 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.

You do not need to do a demo for phase 1, unless your design has a significant number of problems (then we will schedule a demo to give you some partial credit). 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`

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, email the TAs *before* the deadline and double-check. Alternatively, you can/should use the verification script (`/u/s/i/sinclair/public/html/courses/cs552/spring2020/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 March 13th.

Electronic submission instructions

Submit a single *demo1.tgz* per group 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 (or Github Classroom) that was provided.

1. The sub-directories should contain the following files:
    1. **verilog/** containing all Verilog 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/i/sinclair/public/html/courses/cs552/spring2020/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 indeed be done (without any additional effort on your part) if you use the same module definition for both instances

```
                          +-------------+
 data_in[15:0] >--------|             |--------> data_out[15:0]
    addr[15:0] >--------| 65536 word  |
        enable >--------| by 8 bit    |
            wr >--------| memory      |
           clk >--------|             |
           rst >--------|             |
    createdump >--------|             |
                          +-------------+
```

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.

D. Phase #2 – Pipelined Design with Perfect Memory (30% of project 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 April 3rd.

Electronic submission instructions

Submit a single *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 (or Github) 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: `instTests.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

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, 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/i/sinclair/public/html/courses/cs552/spring2020/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.

E. 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.

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 and on Github.

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.

```
                      +-------------+
data_in[15:0] >-------|             |--------> data_out[15:0]
   addr[15:0] >-------| 65536 word  |
       enable >-------| by 16 bit   |--------> err
           wr >-------| memory      |
          clk >-------|             |
          rst >-------|             |
   createdump >-------|             |
                      +-------------+
```

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

F. 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. With the same semantics.

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 -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.

G. Cache Demo - Working two-way set-associative cache (10% of project grade)

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

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

Due May 1st at 11:59 PM on Canvas, no exceptions.

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 (or Github Classroom) that was provided.

1. verilog/ containing all verilog 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.

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

We will electronically grade this submission, similar to the prior phases. TAs will be holding extra office hours on May 2nd (time TBA) for any teams that have failures and need to meet about partial credit. If this time does not work for a team, please send us an email in advance to set up an alternative time.

As mentioned previously, you can submit each phase up to 2 days late with a 10% penalty per day, along with having 3 "free" late days for the project across the semester. If we meet you for a demo, we will use files that you submitted at or before 1159PM on the 1st of May (or the appropriate day if you are using late days).

- 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 Github, 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

Both partners are required to be present and both are expected to explain and answer questions about the whole design. Answering a question with: "I have no idea, my partner did that" is a failing answer. You must (at least) be able to answer: "My partner implemented that, but 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/i/sinclair/public/html/courses/cs552/spring2020/handouts/scripts/project/phase3/verify_submission_format.sh demo3.tgz
```

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

H. Final Project Report: May 5th (10% of project grade)

Due by 1159pm on May 5th

Each team should turn in one final report that is typed, well written, and well organized. Semantic, spelling, or grammatical errors will be penalized.

- Please check the template for details on what is required.

For writing the final report use this template if you use Word, or follow the format in this pdf. The templates are linked in the “Final Report” assignment on Canvas and on the Project course webpage page.