ECE 411: Computer Organization and Design

MP 3: A Pipelined Implementation of the LC-3b Processor

Version 2.0.2

1 Overview

This machine problem involves the design of a pipelined microprocessor. You are required to implement the LC-3b instruction set in its entirety (with the exception of RTI) using the pipelining techniques described in class. This handout is an incomplete specification to get you started with your design; a portion of this machine problem is left open ended for you to explore design options that interest you.

You will begin your design by creating a basic pipeline that can execute several basic LC-3b instructions. Then, as the project progresses, you will add support for the rest of the LC-3b ISA, a cache hierarchy consisting of split L1 caches and a unified L2 cache, hazard detection, and data forwarding. Toward the end of the project, you will have the opportunity to expand your design with advanced design options of your choosing. For groups which correctly implement an MP 3 design, a competition will be held to find which design can execute our benchmark program in the least amount of simulated time.

2 Getting started

2.1 Working as a group

For this assignment, you must work in a group of three people[1]. It will be your responsibility to work on the assignment as a team. Every member should be knowledgeable about all aspects of your design, so don't silo responsibilities and expect everything to work when you put the parts together. Good teams will communicate often and discuss issues (either with the design or with team members) that arise in a timely manner.

To aid collaboration, we provide a private GitLab[2] repository for class that you can use to share code within your team and with your TA. We will add you to our class account on GitLab, within which you will be able to create a repository to share with your group.

2.2 Mentor TAs

Given that every group's design will be different in MP 3, it is often difficult for a TA unfamiliar with your group to answer all of your questions. In order to make sure that each group has someone who is knowledgeable about their design, each group will be assigned a mentor TA. You will have regular meetings with your mentor TA so that they know how your project is doing and any major hurdles you have encountered along the way. You should meet with your mentor TA at least once every checkpoint. Scheduling these meetings is *your* responsibility. In the past some teams have (foolishly) blown these meetings off, and we've received course feedback asking that they be made mandatory.

Your first meeting with your mentor TA will be not only to review your paper design for your basic pipeline, but also to discuss your goals for the project. Before meeting with your mentor TA you should have discussed in detail with your team about what design options you plan to explore. Your mentor TA may advise against certain

[1] The only exception to form a group of two or an individual person is when the class is not a multiple of three.

[2] `https://gitlab-beta.engr.illinois.edu`

options, but you are free to work on whatever you like. As the project progresses, you may find that your goals and your design change. This is normal and you are free to take the project in a direction that interests you. However, you must keep your mentor TA up to date about any changes you have made or plan to make.

In order to get the most out of your relationship with your mentor TA, you should approach the relationship as if your group has been hired into a company and given the MP 3 design as a job assignment. A senior engineer has been assigned to help you stay on schedule and not get overwhelmed by tool problems or design problems. *Do not* think of the TA as an obstacle or hostile party. *Do not* try to "protect" your design from the TA by not allowing him or her to see defects or problem areas. *Do not* miss appointments or engage in any other unprofessional conduct. *Do not* hand in submissions past the deadline unless you have discussed extenuating circumstances with your TA. Your mentor TA should be a consulting member of your team, not an external bureaucrat.

2.3 Testing

Throughout the MP, you will need to generate your own test code. This is extremely important as untested components may lead to failing the final test code and competition benchmark altogether. Remember that in many of your components, such as the register bypassing unit, the order of the instructions as well as what operands are used is crucial. You cannot just test that your processor executes each of the instructions correctly in isolation. You should try to generate test code to test as many corner cases as you can think of.

3 The pipeline

3.1 Pipeline control

In this pipelined design, you will be using a simple control memory to generate control signals. No state diagrams will be allowed for pipeline control (state diagrams *are* still allowed for cache controllers). The control memory behaves similar to a ROM (read only memory). When an instruction is ready to be decoded, the opcode, (and possibly some other fields of the instruction word) are sent to a logic block, which then outputs a *control word* containing the signals necessary to control all the pipeline stages of the particular instruction. This control word (which is not necessarily 16 bits long) is then passed down the pipeline along with the other operands needed for the instruction. To assist in debugging, please pass the *entire* control word as well as the instruction's opcode down the pipeline. This allows the logic in each stage to be set correctly for that specific instruction.

One way to implement the control memory discussed above is using a SystemVerilog case statement. You can implement the control word as a `struct` containing the signals that you need. The following code blocks contain examples for the control word struct and control ROM.

Listing 1: Example control word definition (place in *lc3b_types.sv*)

```
typedef struct packed {
    lc3b_opcode opcode;
    lc3b_aluop aluop;
    logic load_cc;
    logic load_regfile;
    /* ... other signals ... */
} lc3b_control_word;
```

Listing 2: Example control memory module

```
import lc3b_types::*;

module control_rom
(
    input lc3b_opcode opcode,
    output lc3b_control_word ctrl
);
```

```
always_comb
begin
    /* Default assignments */
    ctrl.opcode = opcode;
    ctrl.load_cc = 1'b0;
    /* ... other defaults ... */

    /* Assign control signals based on opcode */
    case(opcode)
        op_add: begin
            ctrl.aluop = alu_add;
        end

        op_and: begin
            ctrl.aluop = alu_and;
        end

        /* ... other opcodes ... */

        default: begin
            ctrl = 0;   /* Unknown opcode, set control word to zero */
        end
    endcase
end
endmodule : control_rom
```

You should only use the opcode concatenated with at most three other bits of the instruction as the inputs to this ROM (you should use only 4-7 bits to generate the control word). Also, you must not do any sort of computation in this ROM. The ROM is meant exclusively for generating simple control signals such as mux selects, etc. Do not use the control ROM to perform computations (such as addition, branch condition comparisons, etc).

3.2 Pipeline datapath

Regardless of your design, you will have to do things in a considerably different manner compared to your previous MP datapath designs. When you start entering your design into the software tools, start from scratch. Do not try to copy over your old MP 2 datapath design and then edit it.

Between each pipeline stage, you will need a set of registers to hold the values from the previous stage. Unlike the design shown in the textbook, you do not need to implement those registers as one colossal box. You are permitted to break the pipeline registers up into many smaller registers each containing one value (e.g., the ALU output, or a control word). Pick a style that fits your group.

3.3 Things to think about

A couple features of the LC-3b ISA present challenges for pipelining. The indirect memory instructions (LDI, STI) require two memory accesses and therefore require special attention in the pipeline. Also, conditional branches require special care to be resolved correctly. Your group should discuss these issues while working on your paper design.

4 Project milestones

MP 3 is divided into several submissions to help you manage your progress. The dates for submissions are provided in the class schedule. Late work *will not* be accepted for MP 3 unless you have discussed extenuating circumstances with your mentor TA *before* the deadline.

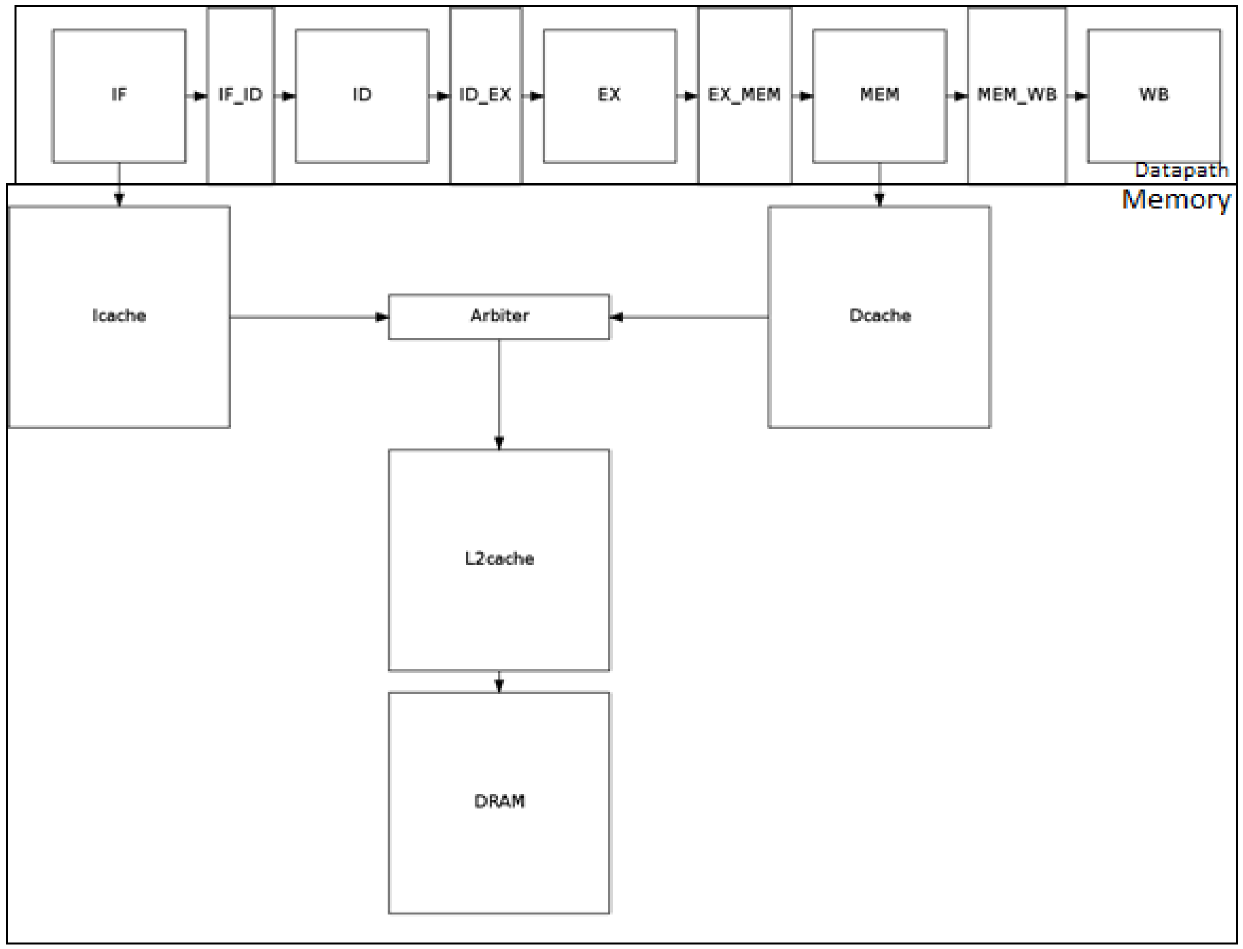

Figure 1: Overview of pipeline datapath and cache hierarchy. Note the location of pipeline stages, stage registers, and arbiter. Your paper designs should be much more detailed than this.

4.1 Basic pipeline paper design

The first submission for this project will be a paper design of your pipelined datapath. The design should be detailed enough for the TAs to trace the execution of all the LC-3b instructions through your datapath. The paper design should map out the entire pipeline, including components in all the stages (e.g., registers, muxes, ALU, register file), stage registers, and control signals. In other words, with the paper design in hand, you should be able to easily translate your design into code.

We will not require your design to handle data forwarding at this point, but you may still want to design for it to avoid having to change your design down the road. If completed, designs for advanced features such as branch prediction can also be included.

A good way to start the pipeline design is to first determine the number of stages and the function of each stage. Then, go through the LC-3b ISA, e.g., ADD, LDR, STR, BR, etc, to see what components need to be added to each stage for a given instruction. Use the book and lecture notes as references.

4.2 Checkpoints

There will be three checkpoints to keep you on track for this MP. For each checkpoint, you will be required to have implemented a certain amount of of the functionality for your pipelined LC-3b design. In addition, at each checkpoint, you should meet, as a team, with your mentor TA and provide him or her with the following information in writing:

- A brief report detailing progress made since the previous checkpoint. This should include what functionality you implemented and tested as well as how each member of the group contributed.
- A roadmap for what you will be implementing for the following checkpoint. The roadmap should include a breakdown of who will be responsible for what and paper designs for all design options that you are planning to implement for the next checkpoint.

Besides helping the TAs check your progress on the MP, the checkpoints are an opportunity for you to get answers to any questions that may have come up during the design process. You should use this time to get clarifications or advice from your mentor TA.

Note that the checkpoint requirements outline the minimum amount of work that should have been completed since the start of the project. You should work ahead where possible to have more time to complete advanced design options.

4.2.1 Checkpoint 1: Basic instructions

By checkpoint 1, you should have a basic pipeline that can handle ADD, AND, NOT, LDR, STR, and BR instructions. You *do not* need to handle any control hazards or data hazards. The test code will contain NOPs to allow the processor to work without hazard detection. For this checkpoint you can use a “magic” memory that always sets `mem_resp` high immediately so that you don’t have to handle cache misses or memory stalls.

Also, you should provide your mentor TA with an overview of your cache hierarchy (showing how your L1 caches, arbiter, and L2 cache interface with each other) as well as detailed designs for your arbiter and L2 cache.

4.2.2 Checkpoint 2: Complete LC-3b ISA and cache arbiter

By checkpoint 2, your pipeline should be able to handle all of the LC-3b instructions. You should also have your cache arbiter implemented and integrated such that the datapath connects to your split L1 caches, which connect to the arbiter, which connects to physical memory (or the L2 cache if that is completed). You must implement multi-cycle L2 accesses at this time. This can be done by adding something like the MAR and MDR registers from your MP0 CPU to your cache datapath, or by adding some registers and control logic to the arbiter. On an L1 miss, the read should not be seen by L2 until the next cycle. This should mean that your critical path does not span all the way from your memory pipeline stage to the L2 cache. At this point, if you have not done so already, you should provide your mentor TA with paper designs for data forwarding

At this point, if you have not done so already, you should provide your mentor TA with paper designs for data forwarding.

4.2.3 Checkpoint 3: Data forwarding and L2 cache

By checkpoint 3, your pipeline should be able to do full (control,data,structural) hazard detection and data forwarding. Furthermore, your L2 cache should be completed and integrated into your cache hierarchy.

4.2.4 Checkpoint 4: Advanced design part 1

By checkpoint 4, you should have a fully functional LC3 pipeline with a 2-level split cache hierarchy with hazard detection and data forwarding. Although your design has come a long way since MP0, all groups have a fairly similar architecture at this point. The remainder of the project will be focused on developing advanced features and optimizing your architecture for performance. CP4 will provide you with base set of advanced features chosen which you will build upon for the final design competition. These features are: software-visible performance counters, an eviction write buffer and static branch prediction.

In order to make intelligent decisions for your advanced features, you will need information about the performance of your processor on different codes. This information is provided through **software visible performance counters**. You should at least have counters for hits and misses in each of the three caches, for mispredictions and total branches in the branch predictor, and for stalls in the pipeline (one for each class of pipeline stages that get stalled together). Once you have added a few counters, adding more will be easy, so you should add counters for any part of your design that you want to measure and use this information to make the design better. The counters should be accessed through dedicated memory addresses (memory mapped input and output). Reading from the dedicated memory address should store the value of the counter in the desired register, and storing to the memory address should clear the counter.

In most test codes, writes are much more infrequent than reads. Additionally, in uniprocessor systems writes are generally not consumed immediately. These two observations motivate the existence of an **eviction write buffer**. Eviction write buffers sit either between two cache levels or between the last level cache and memory. Upon a writeback, rather than writing to to the next level of memory, your cache instead adds the writeback request to the eviction write buffer. The eviction write buffer then services the writeback at the next available opportunity when the next level cache is free. Loads should only stall if there is a writeback performed while the eviction write buffer is full. For CP4, it is only necessary to implement one single-entry eviction write buffer; however, you may find it useful to make a larger buffer, or place several eviction write buffers in your hierarchy.

You will need to implement a **static branch prediction** scheme. You may choose between static taken, static not taken, backwards not taken/forward taken, or backwards taken/forward not taken. Keep in mind that overall performance is graded for the final turn in. Your branch predictor must decide within the instruction fetch stage whether to use the branch PC or the standard PC+2, such that there will be no stalls to resolve the PC. If your prediction is correct, you will not need to disrupt the pipeline. If your prediction is incorrect, you must flush the erroneous stages of the pipeline and begin fetching from the correct PC. On an incorrect prediction, you must be sure to not modify the architectural state.

To get points for the eviction write buffer, performance counters, and static branch predictor in checkpoint 4, you must write test code that demonstrates these features. **Note that having correct register values is not sufficient for verifying correctness of these features.**

After checkpoint 4, you will be able to focus on completing the remaining advanced design options of your choosing. You should discuss your roadmap with your mentor TA to get advice and make sure you will be able to complete the project.

Note: While the features in CP4 are important for your final design, correctness is infinitely more important than performance. In general, you should not move on to CP5 until your code works completely on all of the provided test codes. See CP5 / Design Competition for further details on grading and consult your mentor TA if you become concerned about your progress.

4.2.5 CP5 / Design Competition

CP5 is where your team can really differentiate your design. A list of advanced features which you can choose to implement is provided below, along with their point values. This is not an exhaustive list; feel free to propose to your TA any feature which you think may improve performance, who will add it to the list and assign it a point value. The features in the provided list are designed to improve performance on most test codes based on real-world designs. In order to design, implement, and test them, you will need to do background research and consult your mentor TA. In order to decide on exact feature specifications and tune design parameters (e.g. branch history table size, size of victim cache), you will need to profile the provided test codes on your design using the performance counters from CP4 (and hopefully more!).

While implementing advanced features is required to earn design points, you must always design with performance in mind. In order to motivate performance-centric thinking, part of your CP5 grade will be determined by your design's best execution time on the competition test codes which we provide. Two separate scales will determine your grade:

- The first scale is a straight linear scale ranking all of the teams in the design competition. First place will receive full points and last place will receive zero points.
- The second scale is a linear scale between the average of the best 3 execution times and a baseline MP3 CP3 design. We will announce the execution time of the baseline closer to the contest date
- Your grade will be determined by the higher of these two scales. This ensures that very high performing designs in a competitive class are not penalized unfairly.
- Ensure that your code works correctly. **Designs which cannot 100% correctly execute the competition code will receive 0 points for performance.**
- You may use a separate design for advanced feature grading and for the competition.

4.3 Final submission

For the final demo, your design should have all proposed features working correctly. You should be able to demonstrate with your own test code any advanced features that you expect to get design points for. You should also know how each feature impacts the performance of your pipeline.

4.4 Presentation and report

At the conclusion of the project, you will give a short presentation to the course staff (and fellow students) about your design. In addition, you will need to collect your checkpoint progress reports and paper designs together as a final report that documents your accomplishments. More information about both the presentation and report will be released at a later date.

5 Grading

MP 3 will be graded out of 140 points. Out of the 140 points, 70 points are allocated for regularly meeting with your TA, for submitting paper designs of various parts of your design, for a final presentation given to the course staff, and for documenting your design within a final report.

A breakdown of points for MP 3 is given in Table 1. Points are organized into two categories across five submissions. Note that the number of points you can attain depends on what advanced design options you wish to pursue.

5.1 Advanced design options

Of the 70 implementation points, 35 will come from the implementation of the basic pipeline and required advanced features. 20 points will be given for the implementation of advanced design options. Up to 20 points will come from your group's position in the design contest, meaning that groups can earn up to 5 extra credit points for high performing designs. For each advanced design option, points will be awarded based on the three criteria below:

- Design and implementation: Your group has a clear understanding of what is to be built and how to go about building it, and is able to produce a working implementation.

	Implementation [70+]	**Documentation [70]**
PD [5]		TA meeting [0] Basic LC-3b pipelined datapath design [5]
CP 1 [15]	LC-3bα pipelined datapath [5]	TA meeting [1] Progress report [2] Roadmap [2] Cache design [5]
CP 2 [20]	Basic LC-3b pipelined datapath [5] Split L1 caches [2] Cache arbiter [3]	TA meeting [1] Progress report [2] Roadmap [2] Hazard detection & forwarding design [5]
CP 3 [20]	Hazard detection & forwarding [5] Unified L2 cache [5]	TA meeting [1] Progress report [2] Roadmap [2] Branch prediction, eviction write buffer, performance counters design[5]
CP 4 [15]	Static branch prediction [5] Eviction write buffer [3] Performance counters [2]	TA meeting [1] Progress report [2] Roadmap [2]
Final [65+]	Advanced design options [20] Design contest [20]	Presentation [20] Report [10]

Table 1: MP 3 points breakdown. Points for each item are enclosed in brackets.

- Testing strategy: The design is thoroughly tested with test code and/or test benches that you have written. Corner cases are considered and accounted for and you can prove that your design works as expected.
- Performance analysis: A summary of how the advanced design impacts the performance of your pipelined processor. Does it improve or degrade performance? How is the performance impact vary across different workloads? Why does the design improve or degrade performance?

A list of advanced design options along with their point values are provided in Section 6.2.

5.2 Group evaluations

At the end of the project, each group member will submit feedback on how well the group worked together and how each member contributed to the project. The evaluation, along with feedback provided at TA meetings throughout the semester, will be used to judge individual contribution to the project. Up to 30 points may be deducted from a group member's score if it is evident that he or she did not contribute to the project.

Although the group evaluation occurs at the end of the project, this should *not* be the first time your mentor TA hears about problems that might be occurring. If there are major problems with collaboration, the problems should be reflected in your TA meetings and progress reports. The responses on the group evaluation should not come as a surprise to anyone.

6 Design guidelines

6.1 Basic design

Every group must complete the basic pipelined LC-3b design which consists of the following:

- **Datapath**

  - 5-stage pipeline which implements the full LC-3b ISA (except RTI) [10]
  - Hazard detection and data forwarding (EX→EX, MEM→EX, MEM→MEM, transparent register file, memory stalling) [5]

- **Cache**
  - Split L1 caches [2]
  - L1 cache request arbiter [3]
  - Unified L2 cache [5]

- **Basic Advanced Features**
  - Static branch prediction [5]
  - Eviction write buffer [3]
  - Software visible performance counters [2]

6.2 Advanced design options

Points for each design option are shown in brackets. To obtain full points for a design option, you must satisfy all the requirements given in Section 5.1. If you would like to add a feature to this list, you may work with your mentor TA to assign it a point value.

- **Cache organization design Options**
  - 4-way set-associative or higher L2 cache with LRU Policy (Pseudo or True) [5]
  - Greater than 3 cycle multi-cycle L2 cache hit [5]

- **Advanced cache options**
  - Victim cache [8]
  - Non-blocking L1 cache[8]
  - Banked L1/L2 cache[5]

- **Branch prediction options**
  - Local branch history table [4]
  - Global 2-level branch history table [7]
  - Tournament Branch Predictor [10]
  - Branch target buffer, support for unconditional + indirect branches [5]
  - 4-way set-associative or higher BTB [8]

- **Prefetch design options**
  - Software prefetching [8]
  - Hardware prefetching (Basic) [8]
  - Hardware prefetching (Advanced) [10]

- **Difficult design options**
  - Memory stage leapfrogging [12]
  - Early branch resolution in EX stage [5]

- **Superscalar design options**
  - Multiple Issue [15]

    – Register Renaming[5]
    – Scoreboarding [20]
    – Tomasulo [20]

- **Design penalties**
    – Handling LDI and STI in software [-15]

6.3 Advanced design options explained

- **Cache organization design Options**
- **Multi-cycle L2 cache** Depending on the size of your L2 cache, you may find that you can achieve a higher clock frequency by designing your L2 cache to take several cycles to complete L1 cache requests. This design option requires you to make modifications to your L2 cache controller and to constrain your design properly for timing analysis. It is possible to tell the Timing Analyzer to allow 2 cycles to complete the corresponding register-register transfer. To set up a multicycle path , follow this tutorial: https://www.altera.com/support/support-resources/design- examples/design-software/timequest/tq-multicycle-path.html To understand how to modify your cache control logic, this forum entry would be useful : http://www.alteraforum.com/forum/showthread.php?t=5576
- **Advanced cache options**
    – **Victim cache**
      This is a version of the eviction write buffer on steroids. The buffer is expanded to be fully associative with multiple entries (typically 8-16), is filled with data even on clean evictions, and is not necessarily written back to DRAM immediately. This enables a direct-mapped cache to appear to have higher associativity by using the victim buffer only when conflict misses occur. This is only recommended for groups who love cache.
    – **Non-blocking L1 cache** While a blocking cache services a miss, no other cache accesses can be serviced, even if there is a hit. A non-blocking cache instead has the ability to queue misses in MSHRs (miss status holding registers) while continuing to service hits. In order for this ability to be useful, the processor must be able to support either out-of-order execution or memory-stage leapfrogging.
    – **Banked L1/L2 cache** A banked cache further divides each cache way into banks, which hold separate chunks of addresses. Each bank can be accessed in parallel, so that multiple memory accesses can begin service at once if there is no 'bank conflict'; that is, each request is directed to a different bank. This option is useful for L1 for groups with a multiple issue processor, and for L2 in the case of having both an i-cache and d-cache miss.
- **Branch prediction options**
- **Branch target buffer**
  This component stores the destination of recent branch instructions so that when they are fetched again, the IF stage can immediately fetch their target if the branch is predicted taken. For unconditional branches that are always taken, you can store information to ensure that those instructions always jump. For these other control instructions, the destination can change depending on a register or memory value. However, you can still store a destination and later check to see if this address has changed.
- **Local branch history table**
  This is conceptually the simplest dynamic branch prediction scheme that earns points. It contains a table of 2-bit predictors indexed by a a combination of the PC values and the history of conditional branches at those PC values.
- **Global 2-level branch history table**
  A global branch history register records the outcomes of the last N branches, which it then combines with (some bits of) the PC to form a history table index. From there it works the same as the local BHT. By recording the past few branches, this scheme is able to to take advantage of correlations between branches in order to boost the prediction rate.

- **Tournament branch predictor**

  A tournament branch predictor chooses between two different branch prediction schemes based on which is more likely to be correct. You must maintain two different branch predictors (e.g. both a local and a global predictor) and then add the tournament predictor to select between which of the two is the best predictor to use for a branch. This predictor should use the two bit counter method to make its selection, and should update on a per-branch basis.

Prefetch design options Prefetching operation is a technique that helps us avoid cache misses. Rather than waiting for a cache miss to perform a memory fetch, prefetching anticipates such misses and issues a fetch to the memory system in advance of the actual memory reference. This prefetch proceeds in parallel with normal instructionsâĂŹ execution, allowing the memory system time to transfer the desired data to cache. Here are several options of implementing prefetching.

- **Software prefetching** Implement a prefetching instruction that fetches the line of data from memory that contains the byte specified with the source operand. When this prefetch instruction is executed, the address calculated from the source operand is sent to memory without forcing the processor to wait for a response. Exp: PREFETCH R1, OFFSET ; cache <- line(mem[R1 + OFFSET]) Implement a compiler to compile the assembly code to machine code (you may exploit the unsed RTI instruction). Profile the test code and insert the prefetch instruction to where it improves performance best. Extra points will be awarded if you implement a program analyzer that automatically insert prefetch instruction to test code and deliver performance improvement.

- **Hardware prefetching (Basic)** One block lookahead (OBL) prefetch, one of the sequential prefetching scheme that takes advantage of spatial locality and is easy to implement. This approach initiates a prefetch for line i + 1 whenever line i is accessed and results in a cache miss. If i + 1 is already cached, no memory access is initiated.

- **Hardware prefetching (Advanced)** PC based strided prefetching. This prefetching scheme is based on following idea: *Record the distance between the memory addresses referenced by a load instruction (i.e. stride of the load) as well as the last address referenced by the load. *Next time the same load instruction is fetched, prefetch last address + stride. For more detail, refer to Baer and Chen, âĂIJAn effective on-chip preloading scheme to reduce data access penalty,âĂİ SC 1991.

Difficult design options

- **Memory stage leapfrogging**

  This allows independent instructions to “jump past” the memory stage if/when there is a data cache miss. Beware! This requires extra special care to make sure that the register file values and condition codes are set correctly when the stalled instruction finally completes.

- **Early branch resolution in EX stage**

  Sometimes branches can be resolved early, sometimes they can’t. If your design can resolve branches early when they can, you get points. Beware! This is very tricky to get right, and correctness depends on how your pipeline is laid out. In other words, nobody can help you on this, since your design will surely be different than any other design. It is also possible to do branch resolution even earlier, and would be worth more points, but we do not recommend trying it, since it is *very* tricky.

Superscalar design options

- **Multiple Issue**

  A multiple issue processor is capable of dispatching and committing multiple instructions in a single cycle. This requires modifications to several major structures in your pipeline. First, you must be capable of fetching multiple instructions from your icache in a single cycle. You also must expand your register file ports to accommodate operand fetching and simultaneous writes. Your forwarding and hazard detection logic will need to detect dependencies between in-flight instructions in the same as well as different pipeline stages. In order to see the most performance improvement for this option, implement in conjunction with banked caches.

- **Scoreboarding, Tomasulo**

  These options are for designs that support parallel execution of multiple instructions and cannot be combined with bonus points for memory stage leapfrogging. In general, we do not recommend these options for any groups, but some groups insist on doing them, and some even succeed. For documentation, see the textbook.

  For full scoreboarding points, you may implement an out-of-order processor based on the scoreboarding structure discussed in class. This option requires that you also implement register renaming. Discuss with your mentor TA for more details.

- **Register Renaming** Just like forwarding is used to fix read after write hazards in your pipeline, register renaming can fix write after write dependencies. WAW dependencies are not an issue in standard MP3 pipelines, but can arise if you implement memory stage leapfrogging or multiple issue, which means you may only get points for register renaming if you implement one of these two features. For scoreboarding and Tomasulo, register renaming is required and the points for register renaming are included in the points for those options.

Design penalties

- **Handling LDI, STI, or other ISA instructions in software**

  All LC-3b ISA instructions must be handled in your pipeline. You may not convert, for example, an LDI into a series of LDRs.

7 FAQs

- **Can we use state machines for our MP 3 design?**

  Only in the cache hierarchy, nowhere else. A non-pipelined cache may use a state machine as its controller.

- **Can we change the bandwidth of physical memory?**

  By default, physical memory has a bandwidth of 128-bits (per direction) and a delay of 200 ns. You may widen this to 256-bits (per direction), but you must set the delay to be 250 ns.

- **What does "no artificial stalls" mean?**

  Note: This question is only relevant if you are pursuing the memory stage leapfrogging design option.

  A better phrasing would probably be "no unnecessary stalls." It means that non-dependent, non-memory instructions which follow a memory operation must not be stalled by the memory operation. This is true even if the memory instruction encounters a cache miss. (Note: For the purposes of this requirement, WAW (write after write) dependencies are considered valid dependencies.) Some examples might help clarify.

  Example 1:

  ```
  LDI R1, R0, 5       ; A
  ADD R4, R4, R3      ; B
  ```

  In this example, instruction B is not dependent on instruction A. It should not be stalled by the fact that instruction A will be in the MEM stage for at least 2 cycles. Instruction B should "go around" the MEM stage and proceed down the pipeline.

  Example 2:

  ```
  LDI R1, R0, 5       ; A
  ADD R3, R1, R5      ; B
  ```

  Here instruction B must stall because it is dependent on instruction A.

  Example 3:

```
LDI R1, R0, 5        ; A
LDR R4, R2, label    ; B
```

Instruction B must stall because it is a memory instruction.

Example 4:

```
LDB R1, R0, 7        ; A
ADD R4, R4, R5       ; B
AND R3, R1, R6       ; C
LSHF R2, R2, 2       ; D
```

Instruction B should not stall (independent). Instruction C must stall. Instruction D is independent, but may stall because the instruction before it is stalling. This illustrates that you can stop letting instructions "go around" the MEM stage once you encounter a dependent instruction.

Example 5:

```
LDI R1, R0, 10       ; A
BRz label            ; B
```

Instruction B is dependent on instruction A. (It needs the condition codes that will be set by A.) Instruction B must stall.

Example 6:

```
LDB R4, R2, 10       ; A
ADD R5, R1, 4        ; B
BRz label            ; C
```

Assume A misses in the cache. Instruction B is independent of A, and should continue on to writeback. C depends only on the condition codes set by B, and should also continue on to writeback.

8 Advice from past students

- On starting early:
  - "Start early. Have everything that you have implemented also in a diagram, updating while you go."
  - "START EARLY. take the design submission for next checkpoint during TA meetings seriously. it will save you a lot of time. Front-load your advanced design work or sufferrrrr"
  - "start early and ask your TA for help."
  - "Finish 3 days before it's due. You will need those 3 days (at least) to debug, which should involve the creation and execution of your own tests!"
  - "Make the work you do in the early checkpoints bulletproof and it will make your life WAY easier in the later stages of MP3."
  - Don't let a passed checkpoint stop you from working ahead. The checkpoints aren't exactly a perfect balance of work. There's *always* more you can add for extra credit.
  - (In an end-of-semester survey, most students responded that they spent 10-20 hours per week working on ECE 411 assignments.)
- Implementation tips:
  - "Don't trust the TA provided hazard test code, just because it works doesn't mean your code can handle all data and control hazards."

  - "Also, it was very good to test the cache interface with the MP 2 cache, and test the bigger cache you do (L2 cache, more ways, 8-way pseudo LRU) on the MP 2 datapath. This just makes it easier to stay out of each other's hair."
  - "Run timing analyses along the way so you're not trying to meet the 100 MHz requirement on the last night."
  - "Write your own test code for every case. Check for regressions."
  - "Don't pass the control bits down the pipeline separately, pass the *entire* control word down the pipeline. Also, pass the opcode and PC down. These are essential when debugging."
  - "Check your sensitivity lists!!"

- Possible difficulties:
  - "Implement forwarding from the start, half of our bugs were in this. Take the paper design seriously, we eliminated a lot of bugs before we started. Use the test-o-matic."
  - "Integration is by far the most difficult part of this MP. Just because components work on their own does not mean they will work together."
  - "The hard part about mp3 is 1) integrating components of your design together and 2) edge cases. Really try to think of all edge cases/bugs before you starting coding. Also, be patient when debugging."

- On teamwork:
  - "Try to split up the work into areas you like – cache vs datapath, etc. You will be in the lab a lot, so you might as well be doing a part of the project you enjoy more than other parts"
  - "Don't get overwhelmed, it is a lot of work but not as much as it seems actually. As long as you start at least a paper design ASAP, you should finish each checkpoint with no problems."
  - "Come up with a naming convention and *stick to it*. Don't just name signals `opcode1`, `opcode2`, etc. For example, prepend every signal for a specific stage with a tag to specify where that signal originates from (`EX_Opcode`, `MEM_Opcode`)."
  - "Label all your components and signals as specific as possible, your team will thank you and you will thank yourself when you move into the debugging stages!"
  - "If you know how to use any version control, use it. If you don't, it's probably a good thing to learn it anyway. We used git and it was super useful."
  - "Learn how to use Github well! It is very difficult to get through MP3 without this knowledge."
  - "If you put in the work, you'll get results. All the tools you need for debugging are at your disposal, nothing is impossible to figure out."