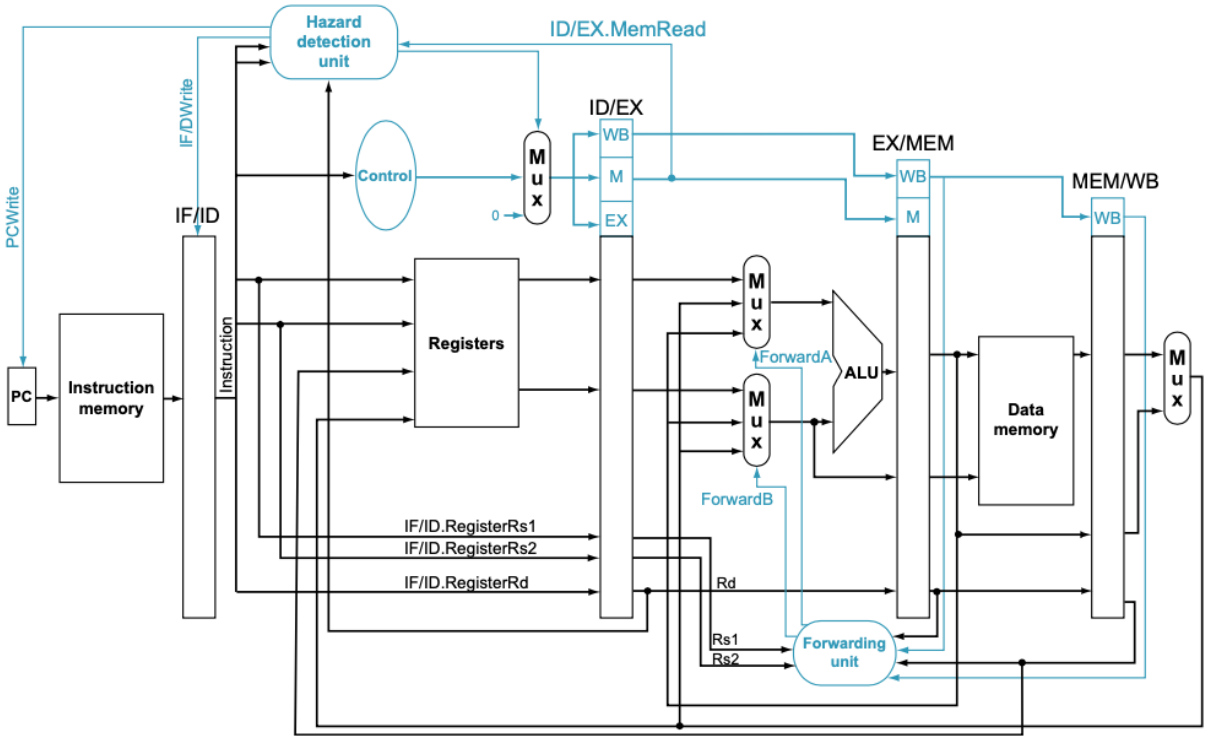
Milestone 2: [35 points] Full pipeline with hazard detection/resolving



In milestone 2, you need to improve your simulator (on top of milestone 1) to simulate a more realistic five-stage CPU pipeline, without considering the cache memory system. Your testcases will be more realistic: there are just real instructions, and no bubble instructions that were inserted on purpose as done in milestone 1. Specifically, as taught in Lecture 8-10, you will simulate all pipeline hazards (both data hazards and control hazards) detection and resolving (including stalling and data forwarding) in the CPU. As shown in the above pipeline diagram (Fig. 4.62 from the textbook, i.e., slide 26 of Lecture 9), we can see that data and control signals are being passed between the stages to account for pipeline hazard detection and resolving paths.

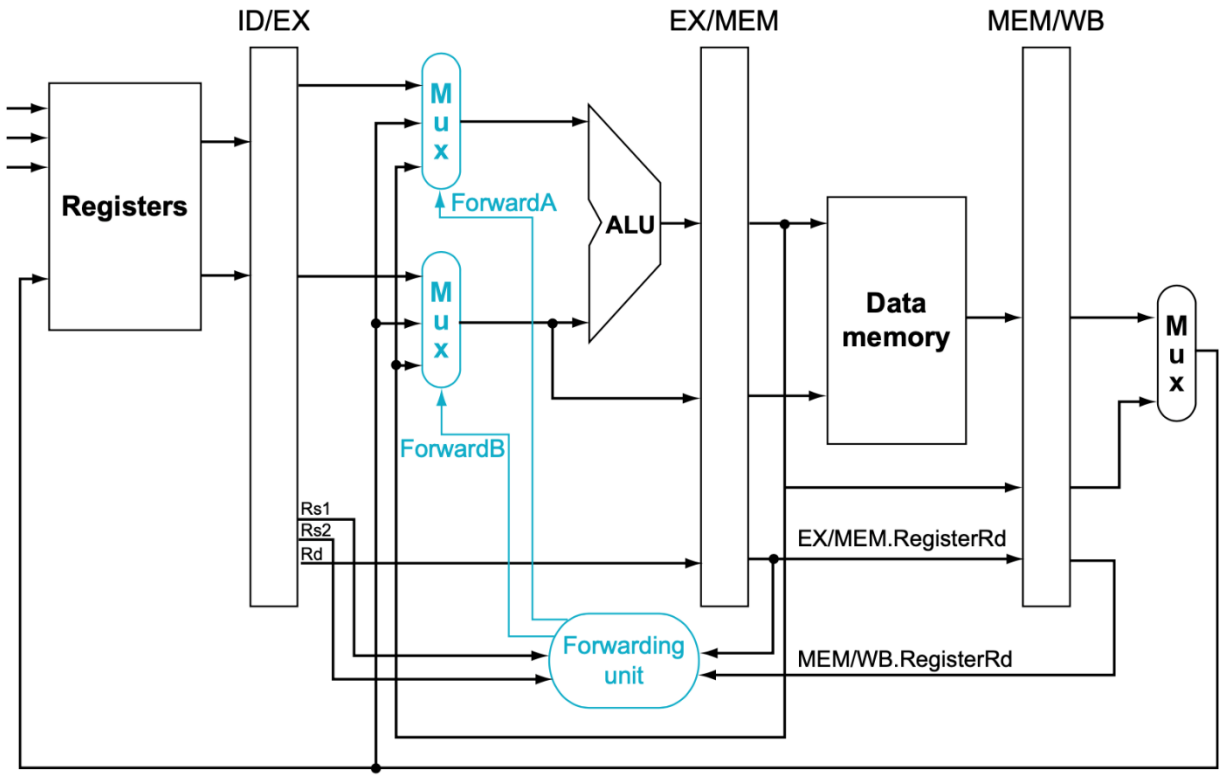
Note in milestone 2, we assume that there is no branch prediction component in the pipeline, and the branch outcome is made in the MEM stage by checking the ALU result stored in MEMWB pipeline registers. We do not handle exceptions in milestone 2. We also ignore the memory access latency in milestone 2, i.e., no extra cycles caused by accessing the cache or memory.

Register-Register Data Hazard Detection and Resolving

As taught in Lecture 9-10, for the register-register dependency (including double data hazards), there are two types of data hazards that need to be detected: the EX hazard and the MEM hazard. The condition of EX hazard is summarized in slide 14 and the condition of MEM hazard is summarized in slide 17. The hazard condition must be checked before the execution stage is processed. It will read IDEX, EXMEM, and MEMWB pipeline registers.

And the register-register data hazard can be resolved using data forwarding without any stalls: EX forwarding and MEM forwarding correspondingly. Please see the figure below (Fig. 4.56 from the

textbook, i.e., slide 11 of Lecture 9) for the architectural diagrams relating to the forwarding. The inputs to the ALU change based on which type of forwarding is required, e.g., take a look at the ALU inputs in the EX stage receiving the forwardA and forwardB control signals for the respective MUXes. Corresponding forwarding signals will be generated (slide 27). Please review Lecture 9 to get a more in-depth understanding.



Load-Use Hazard Detection and Resolving

As taught in Lecture 9-10, the condition of load-use hazard is summarized in slide 20, it's checked when the use instruction is decoded in ID stage.

To resolve this hazard, we stall the pipeline for one cycle, and then detect the MEM hazard and apply data forwarding. Note there will be one-cycle stall (bubble) for load-use hazard even with data forwarding. To insert this bubble, the hazard unit reads the IDEX pipeline register, and rewrites the PC and the IFID register with their previous values, while clearing out all the control fields in the IDEX register. The hazard unit must generate a signal for the MUX after the control module, which is used to either clear out the control fields in the IDEX register, or forward the correct values generated by the control module. Please review Lecture 9 to get a more in-depth understanding.

Produce-Store Hazard Detection and Resolving

As taught in Lecture 9-10, produce-store data hazard can be captured by either EX or MEM hazard as discussed in Register-Register Data Hazard Detection and Resolving.

Control Hazard Detection and Resolving

In milestone 2, we assume that the branch outcome is made in the MEM stage by checking the ALU result stored in MEMWB pipeline registers. And we assume that there is no branch prediction component in the pipeline, i.e., it always assumes that it would fetch the next instruction at PC+4. If the branch outcome is taken, it's a hazard, and we need to flush the pipeline and fetch instructions from the target address. Please review Lecture 9 to get a more in-depth understanding.

Framework Code Modifications

We provide you with modified framework code (project_ms2.zip) to help you keep one unified source code to support both milestone 1 and milestone 2. Please edit the following files according to the given instructions to update your framework.

1. Makefile: It helps you clean output files in both your milestones. Please **replace** your current Makefile with this new Makefile.
2. riscv.c: We have updated riscv.c to print new simulation statistics that you are supposed to print after milestone 2 is completed. Please **replace** your current riscv.c with this new riscv.c.
3. config.h: As you have seen in milestone 1, we are using C++ macros to control different prints that we require during different simulation configurations. From now on, this config.h file will be used to control what macros should be enabled (only for the current milestone) and what macros should be commented out (for all other milestones). Please **copy** this new file to your working directory (top project folder.
4. test_simulator_ms2.sh & test_simulator_ms2_extended.sh: These two files contain the commands required for testing in milestone 2 as described below. Please **copy** these new scripts to your working directory.
5. pipeline.h: We have removed C++ macros (which control prints and other functionalities) from the pipeline.h file and placed them in the config.h. Also, we have added new variables to be used to track some statistics like number of branches and stalls. Please **merge** these new changes to the pipeline.h from your milestone 1. Please watch the TA tutorial for more details.
6. pipeline.c: We have added new variables to be used to track some statistics like the number of branches and stalls. Please **merge** these new changes to the pipeline.c from your milestone 1. Please watch the TA tutorial for more details.
7. code/ms2 folder: Like milestone 1, this folder contains modified testcases, references and output directories. Please **copy** this new 'ms2' folder into your 'code' folder, i.e.,
   cp project_ms2/code/ms2 <your_project_folder>/code/ -rf

Your Implementation

In Milestone 2, we will overhaul our pipeline to manage the four hazards described earlier.

Register-Register Data Hazard

In the skeleton code in stage_helpers.h, a function named *gen_forward* is provided to implement this. There are two types of register-register data hazards:

1. EX Hazard: When resolving an EX hazard (which will require a forwarding from EXMEM register to the EX stage), the simulator should print the following line: "[FWD]: Resolving EX hazard on ***RS***: ***xREG***"
2. MEM Hazard: When resolving a MEM hazard (which will require a forwarding from MEMWB register to the EX stage), the simulator should print the following line: "[FWD]: Resolving MEM hazard on ***RS***: ***xREG***"

Here, ***RS*** can be either "rs1" or "rs2". ***xREG*** can be any of the "x[0-31]" registers. As an example, resolving a MEM hazard on x6 being used as the rs2 field for an instruction should print:

"[FWD]: Resolving MEM hazard on rs2: x6"

Produce-Store Hazard

The produce-store hazard can be captured by the above Register-Register Data Hazard. It doesn't need extra handling.

Load-Use Hazard

To resolve this hazard, a bubble is inserted between the IFID and the IDEX registers by zeroing out the control fields in the IDEX register. In the skeleton code for stage_helpers.h, a function named *detect_hazard* is provided to implement this.

When resolving a load-use hazard, the same instruction is fetched again by the IF stage. As an example, if this re-fetched address of the PC is 0x00002000, the simulator should print:

"[HZD]: Stalling and rewriting PC: 0x00002000"

Control Hazard

To resolve a Control Hazard, the pipeline must be flushed, clearing out the IFID, IDEX, and EXMEM registers. However, to prevent the decode_instruction from throwing errors, please set the *instr* field in the cleared registers to 0x00000013 (NOP) instead of 0x0. While resolving a control hazard arising out of taking a branch, the simulator should print:

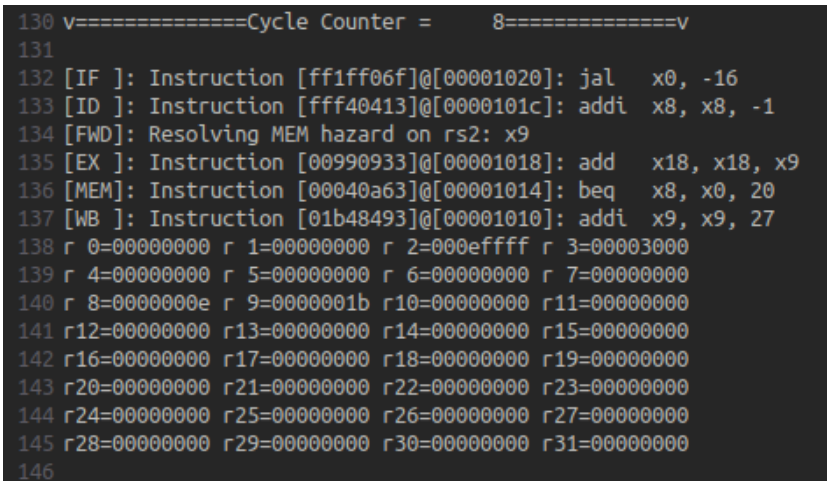
"[CPL]: Pipeline Flushed"

Remember, you will have to call these required helper functions (*gen_forward, detect_hazard*, etc.) in the correct stages to support hazard detection and resolving. The framework code that we provide is a helpful guideline, but there is no singular way to write a simulator. Actual functionality is open to implementation. You are free to define any additional helper functions per your discretion, limited to either the pipeline.c or the stage_helpers.h files.

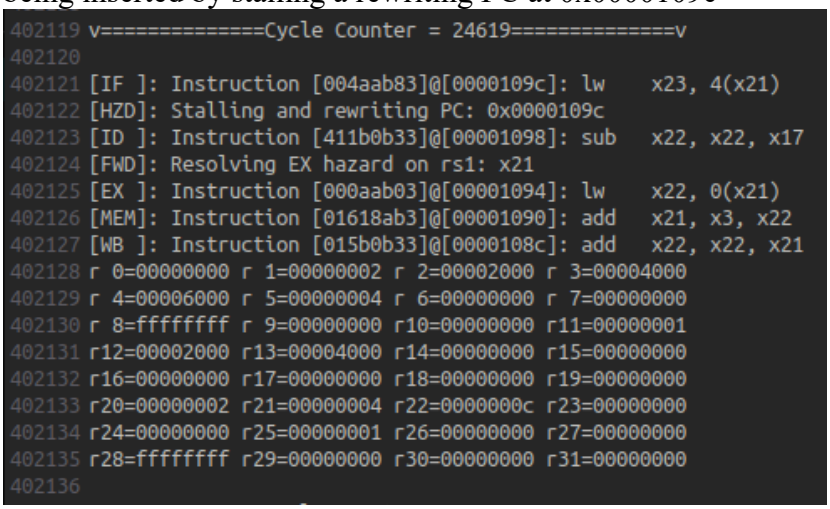
Milestone 2 Testing

The testing for milestone 2 is very similar to milestone 1 testing. However, the input trace files no longer have the bubbles that were intentionally added to prevent control/data hazards. In addition to the milestone 1 content, the following additional content will be compared.

1. Detailed prints of hazard detection/resolving as described above.
   For example, line 134 from code/ms2/ref/multiply.trace shows MEM hazard being resolved for x9



2. Insertion of a bubble (in the pipeline, NOT in the input trace).
   For example, line 402,122 from code/ms2/ref/vec_xprod_tiny.trace shows the bubble being inserted by stalling a rewriting PC at 0x0000109c



3. Flushing the pipeline.
   For example, line 186 from code/ms2/ref/multiply.trace shows that the pipeline needs to be flushed after recognizing the jal instruction in the MEM stage at clock-cycle 11. In the next clock cycle, ID, EX, and MEM stages show addi x0,x0,0 (line 199, 200, 201) since the corresponding pipeline registers are all cleared to NOPs.

```
179 v==============Cycle Counter =    11==============v
180
181 [IF ]: Instruction [01312023]@[0000102c]: sw    x19, 0(x2)
182 [ID ]: Instruction [ffc10113]@[00001028]: addi  x2, x2, -4
183 [EX ]: Instruction [00100993]@[00001024]: addi  x19, x0, 1
184 [MEM]: Instruction [ff1ff06f]@[00001020]: jal   x0, -16
185 [WB ]: Instruction [fff40413]@[0000101c]: addi  x8, x8, -1
186 [CPL]: Pipeline Flushed
187 r 0=00000000 r 1=00000000 r 2=000effff r 3=00003000
188 r 4=00000000 r 5=00000000 r 6=00000000 r 7=00000000
189 r 8=0000000d r 9=0000001b r10=00000000 r11=00000000
190 r12=00000000 r13=00000000 r14=00000000 r15=00000000
191 r16=00000000 r17=00000000 r18=0000001b r19=00000000
192 r20=0000000e r21=00000000 r22=00000000 r23=00000000
193 r24=00000000 r25=00000000 r26=00000000 r27=00000000
194 r28=00000000 r29=00000000 r30=00000000 r31=00000000
195
196 v==============Cycle Counter =    12==============v
197
198 [IF ]: Instruction [01b48493]@[00001010]: addi  x9, x9, 27
199 [ID ]: Instruction [00000013]@[0000102c]: addi  x0, x0, 0
200 [EX ]: Instruction [00000013]@[00001028]: addi  x0, x0, 0
201 [MEM]: Instruction [00000013]@[00001024]: addi  x0, x0, 0
202 [WB ]: Instruction [ff1ff06f]@[00001020]: jal   x0, -16
203 r 0=00000000 r 1=00000000 r 2=000effff r 3=00003000
204 r 4=00000000 r 5=00000000 r 6=00000000 r 7=00000000
205 r 8=0000000d r 9=0000001b r10=00000000 r11=00000000
206 r12=00000000 r13=00000000 r14=00000000 r15=00000000
207 r16=00000000 r17=00000000 r18=0000001b r19=00000000
208 r20=0000000e r21=00000000 r22=00000000 r23=00000000
209 r24=00000000 r25=00000000 r26=00000000 r27=00000000
210 r28=00000000 r29=00000000 r30=00000000 r31=00000000
```

Important: You have to **ENABLE all macros for relevant milestone 2 test and disable all other macros in config.h.**

Milestone 2 consists of 7 tests in total, which are divided into 2 sets:

1. [18 points] 6 tests which print out instruction trace, register trace, and statistics, invoked using ./test_emulator_ms2.sh
2. [6 points] 1 huge test which prints out only the statistics (see config.h for detailed macro configuration), invoked using ./test_emulator_ms2_extended.sh

It is strongly recommended that you make sure Set 1 is passed before launching Set 2. Before running the tests, make sure that the appropriate sections (macros) in config.h are commented or uncommented, so that the prints are correctly taken care of.

Commands to run (Set 1 – test_simulator_ms2.sh):

1. R-type: 3 points
   ./riscv -s -f ./code/ms2/input/R/R.input > ./code/ms2/out/R/R.trace
   diff ./code/ms2/ref/R/R.trace ./code/ms2/out/R/R.trace
2. I-type: 3 points
   ./riscv -s -f ./code/ms2/input/I/I.input > ./code/ms2/out/I/I.trace
   diff ./code/ms2/ref/I/I.trace ./code/ms2/out/I/I.trace

3. Mixed L-type and S-type: 3 points
   ./riscv -s -f ./code/ms2/input/LS/LS.input > ./code/ms2/out/LS/LS.trace
   diff ./code/ms2/ref/LS/LS.trace ./code/ms2/out/LS/LS.trace

4. Random test: 3 points
   ./riscv -s -e -f ./code/ms2/input/random.input > ./code/ms2/out/random.trace
   diff ./code/ms2/ref/random.trace ./code/ms2/out/random.trace

5. Multiply test: 3 points
   ./riscv -s -e -f ./code/ms2/input/multiply.input > ./code/ms2/out/multiply.trace
   diff ./code/ms2/ref/multiply.trace ./code/ms2/out/multiply.trace

6. Reduced vector cross-product test: 3 points
   ./riscv -s -e -f ./code/ms2/input/vec_xprod_tiny.input
   > ./code/ms2/out/vec_xprod_tiny.trace
   diff ./code/ms2/ref/vec_xprod_tiny.trace ./code/ms2/out/vec_xprod_tiny.trace

Commands to run (Set 2 – Blind Test - test_simulator_ms2_extended.sh):

1. Full vector cross product test: 6 points
   ./riscv -s -e -f ./code/ms2/input/vec_xprod.input > ./code/ms2/out/vec_xprod.trace
   diff ./code/ms2/ref/vec_xprod.trace ./code/ms2/out/vec_xprod.trace

Milestone 2 Marking (35 points)

- Milestone 2 contains 35 points of the project.
- 18 points are given for the successful completion of the above tests in Set 1. There are 6 tests in Set 1, each worth 3 points. For each test, if the entire test passes (i.e., match with the reference output), you will earn all 3 points; otherwise, you will get 0 points for the test. No partial points will be given.
- 6 points are given for the successful completion of the blind test (Set 2). The reference file for the blind test will not be provided. If the entire test passes (i.e., match with the final generated statistics), you will earn all 6 points; otherwise, you will get 0 points for the test. No partial points will be given.
- 6 points are given for the demo of this part. Earning these points depends on how confident you will explain and answer questions about your work related to this part during the demo.

- 5 points are given for your report. You must include a section clearly describing how you have implemented milestone 2 with reference to your project code including the following details. If the section is missing, or the description is vague or incomplete, no points will be awarded.
    - What is the condition in your code that determines an EX-hazard? Explain your code with how it determines the EX-hazard. Include details about how you implemented forwarding to resolve this hazard.
    - What is the condition in your code that determines a MEM hazard? Explain your code with how it determines the MEM hazard. Include details about how you implemented forwarding to resolving this hazard.
    - What is the condition in your code that determines a load-use hazard? Explain this with how your code determines the load-use hazard.
    - Explain how you implement the pipeline flush (inserting a NOP) in your code.
    - Explain how you insert a bubble in your (simulator) code to implement a stall.
    - Document any other changes you had to make in pipeline stage functions to accommodate the above requirements. Make sure to include reasons for those changes.