CS520 Computer Architecture

Project 2 – Spring 2020

Due: 5/5/2020, 11:59 pm

1. RULES

(1) All students must work alone. Cooperation is not allowed.
(2) Sharing of code between students is considered cheating and will receive appropriate action in accordance with University policy. The TA will scan source code through various tools available to us for detecting cheating. Source code that is flagged by these tools will be dealt with severely.
(3) You must do all your work in the C/C++.
(4) Your code must be compiled on remote.cs.binghamton.edu or the machines in the EB-G7 and EB-Q22. This is the platform where the TAs will compile and test your simulator. As I know, they all have the same software environment.
(5) There will be no extension in the due date.

2. Project Description

In this project, you will construct a superscalar pipeline.

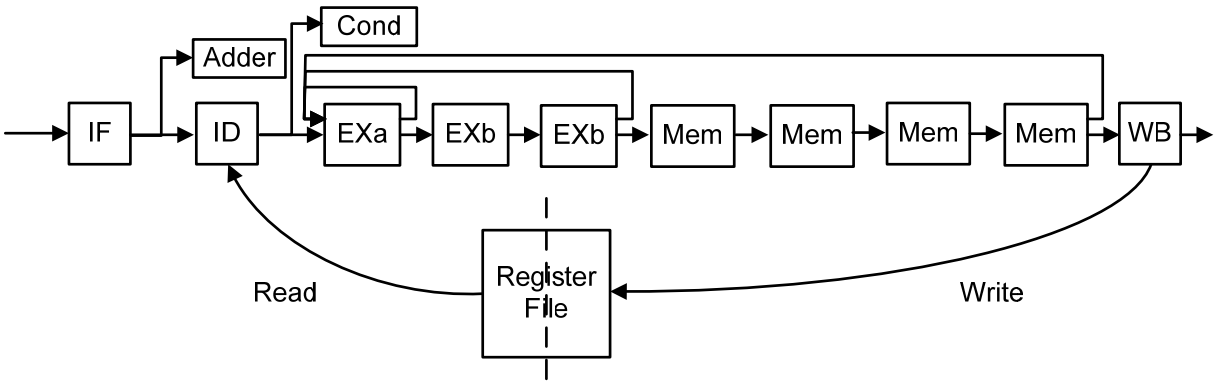
3. Baseline Pipeline

Model simple pipeline with the following stages.

- 1 stage for fetch (IF)
- 1 stage for decode (ID)
- 1 stages for addition and subtraction (Execution unit A: EXa)
- 2 stages for multiplication and division (Execution unit B: EXb)
- 4 stages for memory operation (Memory unit: MEM)
- 1 stage for write back (WB)
- 16 x 4B registers
- 64KB memory (code for 0-999, data for 1000 – 65535)
  memory.map is the memory map file for this project.
  Each memory entry has 4B memory space (A space is in between memory entries)
- Instruction formats (4B fixed instruction)

```
set Rx, #Imm   // set value to register Rx
ld  Rx, #Addr      // load into register Rx the data stored in the address #Addr. E.g.) ld R1, 0x0010
ld  Rx, Ry         // load into register Rx the data stored in the address at Ry
st  Ry, #Addr      // store the content of register Rx into the address #Addr. E.g.) st R1, 0x0010
st  RX, Ry         // store the content of register Rx into the address at Ry
add  Rx, Ry, Rz    // Compute Rx = Ry + Rz
add  Rx, Ry, #Imm   // Compute Rx = Ry + #Imm (an immediate valve)
sub  Rx, Ry, Rz       // Compute Rx = Ry – Rz
```

```
sub  Rx, Ry, #Imm   // Compute Rx = Ry - #Imm (an immediate valve)
mul  Rx, Ry, Rz     // Compute Rx = Ry * Rz
mul  Rx, Ry, #Imm   // Compute Rx = Ry * #Imm (an immediate valve)
div  Rx, Ry, Rz     // Compute Rx = Ry / Rz
div  Rx, Ry, #Imm   // Compute Rx = Ry / #Imm (an immediate valve)
bez Rx, #Imm     // branch to #Imm if Rx==0
bgez Rx, #Imm    // branch to #imm if Rx >= 0
blez Rx, #Imm    // branch to #imm if Rx <= 0
bgtz Rx, #Imm    // branch to #imm if Rx > 0
bltz Rx, #Imm    // branch to #imm if Rx < 0
ret              // exit the current program
```



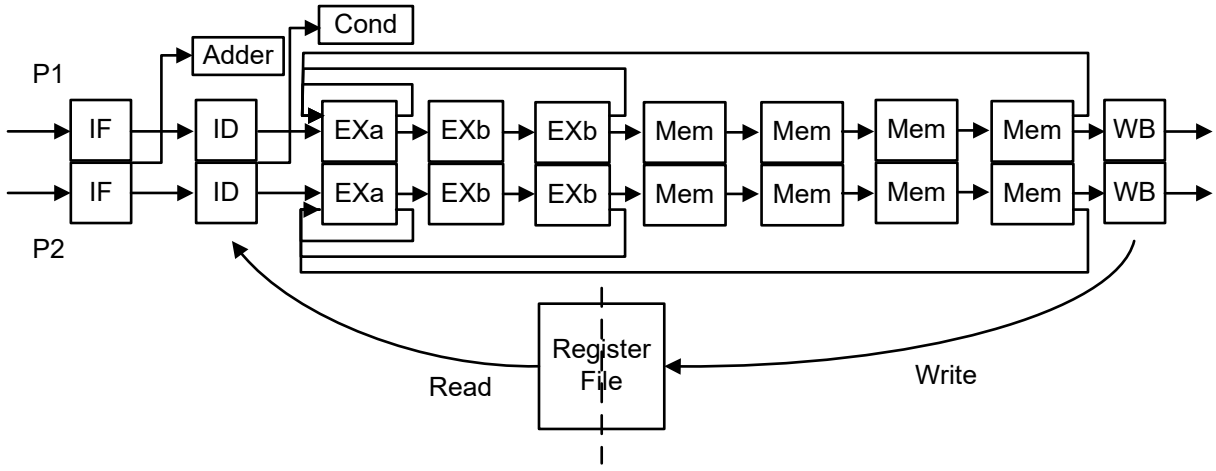
For this project, you need your previous project codes (Pipeline update #2). As described in the previous project, the dependency check and register read are done at the ID stage. Your register file has two read ports and one write port. The register file allows the write operation in the first half of the clock cycle and the read operation in the second half of the cycle. We have an additional adder so that the branch target address is computed at the ID stage, and the branch condition is checked at the EXa stage. Additionally, the pipeline has bypassing/forwarding. The result of each instruction becomes ready after the last pipeline stage of the corresponding execution unit (e.g., if MEM unit needs 4 cycles to complete an instruction, your result is available after the 4th cycle in the Mem unit).

3.1. (25 points) Pipeline update #1: Parallel Pipeline

Your pipeline width is 2, meaning that all functional units are doubled in the pipeline. Here, we call one pipeline as pipeline P1 and another as pipeline P2. The pipeline supports the bandwidth to extend the wire width in the pipeline twice. Your register file also has four read ports and two write ports. Each register only allows one operation at a cycle. The register file allows the write operation in the first half of the clock cycle and the read operation in the second half of the cycle.

Two instructions are fetched simultaneously. However, the ordering is maintained in the way that always the P1 fetch unit fetches an earlier instruction while the P2 fetch unit fetches the next instruction. This pipeline doesn't support out-of-order processing. Also, the pipeline buffers(registers) do not support reordering, meaning that two instructions at the IF stage must advance simultaneously. An instruction at the ID stage can advance alone **in-order**, meaning that only the instruction at the P1 ID stage can advance alone. The freed space cannot be used by a subsequent instruction until both instructions advance.

The number of an extra adder for the branch target address is still one. If you need to compute two branch instructions at the same cycle, one must wait until the adder becomes available. An earlier instruction in the program order is always executed first. The dependency check and register read are performed at the ID stage. Additionally, the pipeline has bypassing/forwarding. The result of each instruction becomes ready after the last pipeline stage of the corresponding execution unit (e.g., if MEM unit needs 4 cycles to complete an instruction, your result is available after the 4th cycle in the Mem unit).



Print the following simulation results.

> Parallel Pipeline <

R0:
R1:
...
R15:

(A) Pipeline stalls due to data hazard:
(B) Wasted cycles due to control hazard:
(C) Pipeline stalls due to structural hazard:
(A + B + C) Total pipeline stalls:

Total cycles:
IPC:

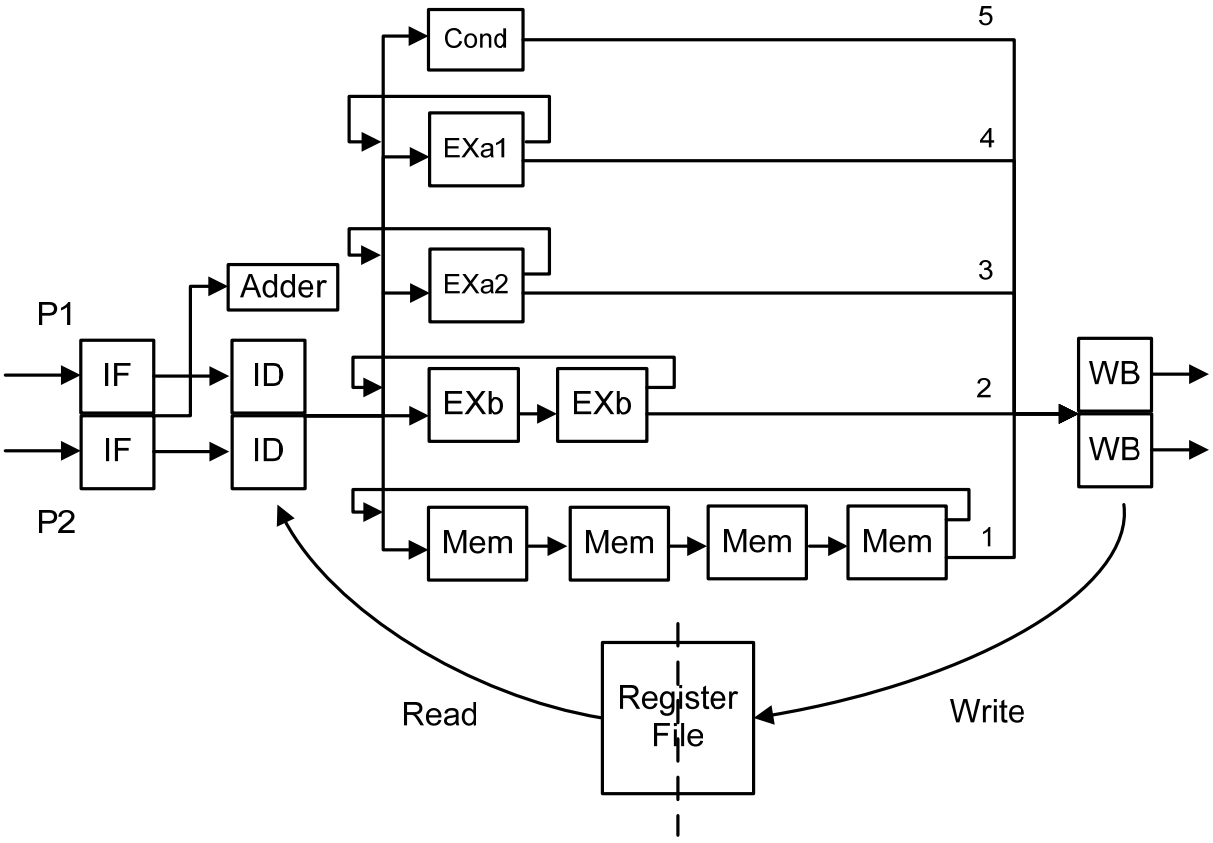
If an instruction stalls due to both a data hazard and a structural hazard, it is counted as a stall due to a data hazard.

3.2. (25 points) Pipeline update #2: Diversified Pipeline

Update the parallel pipeline into a diversified pipeline as follows. We have 2x EXa units (EXa1 and EXa2), 1x EXb unit, 1x Mem unit, and 1x branch condition check unit (Cond). The Cond unit processes both conditional branch instructions (bez, bgez, blez, bgtz, and bltz) and unconditional branch instructions (ret). The set instruction only sets a register at the WB stage, but we define that it takes the path to the EXa. All other instructions take paths to the corresponding function units at the ID stage. The pipeline has two IF, ID, and WB units, processing two instructions per cycle. If there is a branch instruction, the PC is updated at Cond unit. After the branch instruction is executed, all wrongly fetched instructions must be squashed.

Two instructions are fetched simultaneously. However, the ordering is maintained in the way that always the P1 fetch unit fetches an earlier instruction while the P2 fetch unit fetches the next instruction. This pipeline doesn't support out-of-order processing. Also, the pipeline buffers(registers) do not support reordering, meaning that two instructions at the IF stage must advance simultaneously. An instruction at the ID stage can advance alone **in-order**, meaning that only the instruction at the P1 ID stage can advance alone. The freed space cannot be used by a subsequent instruction until both instructions advance.

If both instructions at the ID stage advance to the EXa units, the earlier instruction is always processed at EXa1, and the next instruction is processed at EXa2 in the program order. The forwarded data is delivered to the beginning of the execution stages. To implement a correct pipeline, you need to monitor WAW dependency. The instruction must stall at the **ID stage** until the dependency is solved.



Since up to two instructions can advance to the WB stage, the functional units have a priority that determines which instruction proceeds to the WB stage first when multiple instructions conflict to use the bandwidth. The priority is numbered above the wires in the below figure. As numbered, Cond unit has the highest priority (5), and Mem unit has the lowest priority (1). For example, if three instructions are in the EXa1, the EXb, and the 4th Mem stage, the instructions in the EXa1 and EXb advance to the WB stage while the memory instruction in the 4th Mem stage must wait its turn. This stall is a structural hazard, but you do not need to count it for the results because it is counted at the ID stage when the stall is backpropagated.

Note. The return instruction may not be the last completed instruction. For example, the return instruction is at the Cond stage while an earlier instruction is still at the 3rd Mem stage. However, your simulator must measure the pipeline performance until the time when all the instructions complete.

Print the following simulation results.

> Diversified pipeline <

R0:
R1:
...
R15:

(A) Pipeline stalls due to data hazard:
(B) Wasted cycles due to control hazard:
(C) Pipeline stalls due to structural hazard:
(A + B + C) Total pipeline stalls:

Total cycles:
IPC:

If an instruction stalls due to both a data hazard and a structural hazard, it is counted as a stall due to a data hazard.

3.3. (45 points) Pipeline update #3: Dynamic Pipeline

Update the diversified pipeline into a dynamic pipeline as follows. Now the pipeline supports out-of-order processing with renaming. The renaming is performed at the ID (dispatch) stage. The pipeline also has an additional adder for the branch instruction. Now, the pipeline can calculate multiple target addresses at the ID stage. The pipeline has three new buffers, a fetch queue, a reservation station, and a reorder buffer. The size of each buffer is as follows.

Fetch queue: 4 entries
Reservation Station: 6 entries
Reorder Buffer: 10 entries

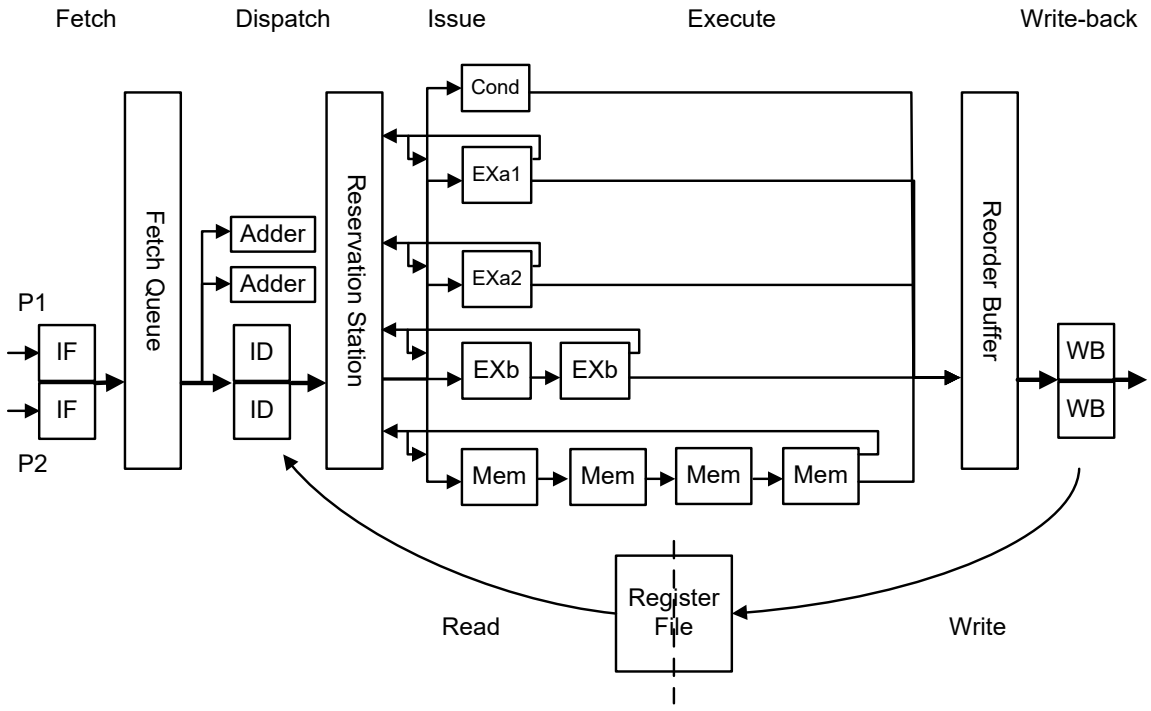
All three buffers support reordering, meaning that one instruction in the previous stage can allocate an entry when only one entry is available in the buffer. Once an instruction is fetched, the instruction is placed in the fetch queue. If an instruction is dispatched, an entry in the reservation station and an entry in the reorder buffer are allocated for the instruction. When the instruction is issued, the entry in the reservation station is freed. Once the instruction is committed (retire), the entry in the reorder buffer is freed. The instructions stay in the reservation station until they become ready.

The instructions are dispatched in-order and are executed out-of-order. They retire (commit) in-order (i.e., the pipeline supports precise exception). To maintain the in-order processing, the fetch queue and the reorder buffer have the first-in, first-out (FIFO) policy. The issue bandwidth is unlimited so that instructions can be issued as long as corresponding functional units are available. The reorder buffer also has unlimited bandwidth so that instruction can update the reorder buffer when its execution completes. However, the WB stage still has a bandwidth limitation, allowing up to two instructions every cycle.

If two ready instructions need the same functional unit at the same cycle, an earlier instruction in the program order is always issued first, while the next instruction must wait its turn at the next cycle. Although this stall is a structural hazard, you do not need to count it for the results. In this pipeline, you only need to count structural hazards when fetched instructions at the IF stage cannot advance due to the full fetch queue. Additionally, you need to count the number of the ID stage stalls due to the full reservation station and the full reorder buffer. If the instruction is stalled due to both full buffers, it is counted as a stall due to the full reservation station.

The data is forwarded to both the reservation station and the beginning of the issue stage. Due to the out-of-order execution, the pipeline now must monitor WAR hazard. Both WAW and WAR hazards are solved by the renaming.

Note. You should refer to the course slides for further detail on the buffers and out-of-order execution.



Print the following simulation results.

> Dynamic pipeline <

R0:
R1:
...
R15:

Number of pipeline stalls due to the full fetch queue:
Number of ID stage stalls due to the full reservation station:
Number of ID stage stalls due to the full reorder buffer:

Total cycles:
IPC:

If an instruction stalls due to both the full reservation station and the full reorder buffer, it is counted as a stall due to the full reservation station.

3.5. Experiments and report

Programs: test program #1 and test program #2

(5 points) Plot two bar graphs of speedup over the baseline pipeline on the y-axis versus each pipeline mode (parallel, diversified, and dynamic) on the x-axis for every test program (test1 and test2). You don't need to include the baseline pipeline (because its speedup is always 1). Thereby, each graph contains 3 bars.

4. Validation and Other Requirements

4.1. Validation requirements

You implement the advanced pipeline described in this project description. Your first reference is this project description and course slides. Additionally, sample simulation outputs will be provided on the website. You should run your simulator and debug it until it matches the simulation outputs. Your simulator must print outputs to the console (i.e., to the screen).

Note. The sample solution is programmed by TA who is a student like you. You should understand that his program can have bugs. Please let TA know immediately once you find a bug in the solution.

Your output must match both numerically and in terms of formatting, because the TAs will literally "diff" your output with the correct output. You must confirm correctness of your simulator by following these two steps for each program:

1) Redirect the console output of your simulator to a temporary file. This can be achieved by placing "> your_output_file" after the simulator command.

2) Test whether or not your outputs match properly, by running this unix command:
"diff –iw <your_output_file> <posted_output_file>"
The –iw flags tell "diff" to treat upper-case and lower-case as equivalent and to ignore the amount of whitespace between words. Therefore, you do not need to worry about the exact number of spaces or tabs as long as there is some whitespace where the sample outputs have whitespace. Both your outputs and final memory.map must be the same as the solution.

3) Your simulator must run correctly not only with the given programs. Note that TA will validate your simulator with hidden programs.

4.2. Compiling and running simulator

We will provide you the instruction parsing code as a library. We recommend you use the provided libraries for your simulator.

You will hand in source code and the TAs will compile and run your simulator. As such, you must meet the following strict requirements. Failure to meet these requirements will result in point deductions (see section "Penalties").

1. You must be able to compile and run your simulator on machines in EB-G7 and EB-Q22. This is required so that the TAs can compile and run your simulator. You also can access the machine with the same environment remotely at remote.cs.binghamton.edu via SSH.

2. Along with your source code, you must provide a Makefile that automatically compiles the simulator. This Makefile must create a simulator named "sim". The TAs should be able to type only "make" and the simulator will successfully compile. The TAs should be able to type only "make clean" to automatically

remove object files and the simulator executable. An example Makefile will be posted on the web page, which you can copy and modify for your needs.

3. Your simulator must accept command-line arguments as follows:

The pipeline must be able to receive 2 parameters, pipeline mode (4: parallel pipeline, 5: diversified pipeline, 6: dynamic pipeline) and a program name.

e.g. sim 4 test_01.asm

4. Your simulator must print outputs to the console (i.e., to the screen) except the final memory map. This way, when a TA runs your simulator, he/she can simply redirect the output of your simulator to a filename of his/her choosing for validating the results. Your simulator must leave the final memory map file as follows (parallel pipeline: memory_4.map, diversified pipeline: memory_5.map, dynamic pipeline: memory_6.map).

4.3. Run time of simulator

Correctness of your simulator is of paramount importance. That said, making your simulator efficient is also important for a couple of reasons.

First, the TAs need to test every student's simulator. Therefore, we are placing the constraint that your simulator must finish a single run in 2 minutes or less. If your simulator takes longer than 2 minutes to finish a single run, please see the TAs as he may be able to help you speed up your simulator.

Second, you will be running many experiments during your implementation. Therefore, you will benefit from implementing a simulator that is reasonably fast.

One simple thing you can do to make your simulator run faster is to compile it with a high optimization level. The example Makefile posted on the web page includes the –O3 optimization flag.

Note that, when you are debugging your simulator in a debugger (such as gdb), it is recommended that you compile without –O3 and with –g. Optimization includes register allocation. Often, register-allocated variables are not displayed properly in debuggers, which is why you want to disable optimization when using a debugger. The –g flag tells the compiler to include symbols (variable names, etc.) in the compiled binary. The debugger needs this information to recognize variable names, function names, line numbers in the source code, etc. When you are done debugging, recompile with –O3 and without –g, to get the most efficient simulator again.

5. What to submit

You must hand in a single zip file called project2.zip. Below is an example showing how to create project2.zip from a Linux machine. Suppose you have a bunch of source code files (*.cc, *.h), and the Makefile, and your project report (report.doc).

zip project2 *.cc *.h Makefile

project2.zip must contain the following (any deviation from the following requirements may delay grading your project and may result in point deductions, late penalties, etc.):

(a) Project report. This must be a single PDF document named report.pdf. (No Word documents please.) The report must include the following:

- A cover page with the project title, the Honor Pledge, and your full name as electronic signature of the Honor Pledge. A cover page will be posted on the project website.

(b) Source code. You must include the commented source code for the simulator program itself. You may use any number of .cc/.h files, .c/.h files, etc.

(c) Makefile. See Section 4.2 for strict requirements. If you fail to meet these requirements, it may delay grading your project and may result in point deductions.

6. Penalties

Various deductions (out of 100 points):
-10 points per date late for the first 4 days.
-20 points per date late after the first 4 days.
Note. It is the end of the semester. There cannot be any extension.

Up to -10 points for not complying with specific procedures. Follow all procedures very carefully to avoid penalties.

Cheating: Source code that is flagged by tools available to us will be dealt with according to University Policy. This includes a 0 for the project and other disciplinary actions.