Palmer Robins

Programming #2 – Design Document

Overview: This program receives a file containing MIPS instructions, formatted in binary or assembly, and checks that file to ensure syntactic correctness, relative to whether it is a ".asm" or ".mach" input file. It then stores a set of the instructions the file contains. Then, it checks the inputted code for read after write dependences, and simulates an ideal pipeline, a pipeline which stalls for RAW dependences, and a pipeline which utilizes data forwarding. The classes used in this implementation include Pipeline, DependencyChecker, ASMParser, BinaryParser, Opcode Table, Register Table, Instruction. Finally, a file named PipelineSim.cpp executes pipeline simulation. Once the program has run, the results of the pipeline sim, and the list of dependences, are printed to the terminal for the user.

Classes:

1. Pipeline: This class simulates the pipeline process. It includes an "ideal pipeline" base class, and stalling and forwarding pipeline subclasses. It contains a list of the instructions and is responsible for using the dependency checker to determine any dependences the second two pipelines need to account for. The two subclasses override the 'runPipeline' method, slightly altering the fundamentals of the pipeline simulation. Finally, it constructs the output that will go out to the command line.
2. DependencyChecker: This class constructs a list of dependences present within the instruction file. Using a list of instructions, it is capable of finding any RAW, WAR, or WAW dependences, although in this implementation we only require RAW. It stores needed register information, used to find the dependences, in a map.
3. BinaryParser: If the file contains binary MIPS instructions, this class opens the input file and checks for syntax correctness. Parsing each binary string, it stores its opcode, function field and any register/immediate values in its list of instructions. Then, it uses the instruction, which contains all the data that is needed, into its corresponding assembly string. An iterator is used to help handle the construction of ITYPE memory instruction assembly strings.
4. ASMParser: If the file contains MIPS assembly instructions, this class opens the input file and checks for syntax correctness. Parsing each line, it stores the opcode, registers used, the assembly string, and any immediate values in its list of instructions.
5. Instruction: As mentioned in BinaryParser, this is the place we can store information about all of the individual instructions included in the input file. Along with all the information listed above, it includes the final assembly string we will eventually print to the command line.
6. Register Table: This class stores data corresponding to the MIPS register system. If a register exists in MIPS, both its name and number are listed in a specific entry.
7. Opcode Table: All of the MIPS instructions that the program supports for translation are listed here. We can store information that is needed concerning each instruction, like its name and any registers that will have to be present within the binary string. Any info

needed about an instruction we can translate will be accessible through its corresponding entry in Opcode Table.

Organization: PipelineSim.cpp reads the filename from the command line, which it then passes to either a BinaryParser or an ASMParser, depending on the file format (binary or assembly). In the Parser that was needed, we open the file, assure that it opened correctly, and then check the file's syntax. Iterating through the file instruction by instruction, we check to make sure all opcode values are contained within our Opcode Table, along with assuring that all register values are contained within our Register Table. If each is accounted for, we create a new Instruction instance for that line and add it to our instruction list. Along the way, the dependency checker, within the Pipeline class, adds any RAW dependences to our dependency list, which we'll need when the three pipelines are simulated. After the instruction list is finalized and all syntax was correct, we can begin to simulate the pipeline. The first step is to set the instruction in each stage to null, and the instruction in the first stage to the first stage of the pipeline. In the ideal pipeline, we do not worry about the dependency list, so we just process instructions through each of the five pipeline stages (fetch, decode, execute, memory, write back) one by one, counting cycles as we go. When an instruction leaves the write back stage, we construct a string to add to our output, listing the instruction number, number of cycles executed, and the assembly string. In the stalling and data forwarding pipelines, check to see if the instruction leaving write back was dependent on a RAW by using our dependency list. If it was, calculate the number of stall cycles needed and then construct the print line. Similar to the ideal, we construct strings when an instruction leaves the write back stage, and finally print the list of strings to the command line.