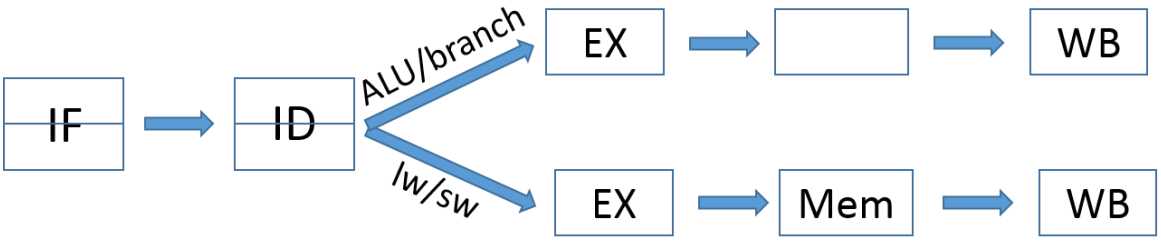
Project 1

Warning: Still under construction. Do not begin until formal release.

The goal of this project is to simulate a simplified superscalar architecture that uses static scheduling to avoid control hazards and dynamic scheduling to mitigate structural and control hazards.

Design Approach

The simplified superscalar pipeline that you will simulate has the following basic structure:



You will run your simulator on 4 short and 2 long trace files (sample1.tr, sample2.tr, sample3.tr, sample4.tr) and (sample_large1.tr, sample_large2.tr). These files are accessible at **/afs/cs.pitt.edu/courses/1541/long_traces** and **/afs/cs.pitt.edu/courses/1541/short_traces**. A small trace, *sample.tr*, can also be found in that directory.

Project Setup:

You are given a program, *five_stage.c*, which reads a trace file (a binary file containing a sequence of executed instructions) and simulates a 5 stage pipeline ignoring any control and data hazards. It outputs the total number of cycles needed to execute the instructions in the trace file and, if a switch *trace_view_on* is set, the instruction that exists the pipeline in each cycle.

Each trace file is a sequence of dynamic trace items, where each trace item represents one instruction executed in the program that has been traced. After *five_stage.c* reads a trace item, it stores it in a structure:

```
struct instruction {
    uint8_t type;       // holds the op-code - see below
    uint8_t sReg_a;     // 1st operand
    uint8_t sReg_b;     // 2nd operand
    uint8_t dReg;       // dest. operand
    uint32_t PC;        // program counter
    uint32_t Addr;      // mem. address
};
```

where

```
enum opcode_type {
    ti_NOP = 0,
```

```
        ti_RTYPE,
        ti_ITYPE,
        ti_LOAD,
        ti_STORE,
        ti_BRANCH,
        ti_JTYPE,
        ti_SPECIAL,
        ti_JRTYPE
    };
```

The “PC” (program counter) field is the address of the instruction itself. The “type” of an instruction provides the key information about the instruction. A detailed list of instructions is given below:

- NOP - it's a no-op. No further information is provided.
- RTYPE - An R-type instruction.
  - sReg_a: first register operand (register name)
  - sReg_b: second register operand (register name)
  - dReg: destination register name
  - PC: program counter of this instruction
  - Addr: not used
- ITYPE - An I-type instruction that is not LOAD, STORE, or BRANCH.
  - sReg_a: first register operand (register name)
  - sReg_b: not used
  - dReg: destination register name
  - PC: program counter of this instruction
  - Addr: immediate value
- LOAD - a load instruction (memory access)
  - sReg_a: first register operand (register name)
  - sReg_b: not used
  - dReg: destination register name
  - PC: program counter of this instruction
  - Addr: memory address
- STORE - a store instruction (memory access)
  - sReg_a: first register operand (register name)
  - sReg_b: second register operand (register name)
  - dReg: not used
  - PC: program counter of this instruction
  - Addr: memory address
- BRANCH - a branch instruction
  - sReg_a: first register operand (register name)
  - sReg_b: second register operand (register name)
  - dReg: not used
  - PC: program counter of this instruction
  - Addr: target address

JTYPE - a jump instruction
    sReg_a: not used
    sReg_b: not used
    dReg: not used
    PC: program counter of this instruction
    Addr: target address
SPECIAL - it's a special system call instruction
    For now, ignore other fields of this instruction.
JRTYPE - a jump register instruction (used for "return" in functions)
    sReg_a: source register (that keeps the target address)
    sReg_b: not used
    dReg: not used
    PC: program counter of this instruction
    Addr: target address

First, you should compile and run the program *five_stage.c* (which includes *CPU.h*) which takes two arguments; the name of the trace file and a switch value (0 or 1). Make sure that when you execute “five_stage sample.tr 1” you get the following output (if the second argument is 0 rather than 1, only the last line is printed):

```
[cycle 5] LOAD: (PC: 2097312)(sReg_a: 29)(dReg: 16)(addr: 2147450880)
[cycle 6] ITYPE: (PC: 2097316)(sReg_a: 255)(dReg: 28)(addr: 4097)
[cycle 7] ITYPE: (PC: 2097320)(sReg_a: 28)(dReg: 28)(addr: -16384)
[cycle 8] ITYPE: (PC: 2097324)(sReg_a: 29)(dReg: 17)(addr: 4)
[cycle 9] ITYPE: (PC: 2097328)(sReg_a: 17)(dReg: 3)(addr: 4)
[cycle 10] ITYPE: (PC: 2097332)(sReg_a: 255)(dReg: 2)(addr: 2)
[cycle 11] RTYPE: (PC: 2097336)(sReg_a: 3)(sReg_b: 2)(dReg: 3)
[cycle 12] RTYPE: (PC: 2097340)(sReg_a: 0)(sReg_b: 3)(dReg: 18)
[cycle 13] STORE: (PC: 2097344)(sReg_a: 28)(sReg_b: 18)(addr: 268454020)
[cycle 14] ITYPE: (PC: 2097348)(sReg_a: 29)(dReg: 29)(addr: -24)
[cycle 15] RTYPE: (PC: 2097352)(sReg_a: 0)(sReg_b: 16)(dReg: 4)
….
[cycle 26] BRANCH: (PC: 2149760)(sReg_a: 16)(sReg_b: 0)(addr: 2149800)
[cycle 27] LOAD: (PC: 2149764)(sReg_a: 16)(dReg: 4)(addr: 2147450887)
…
[cycle 36] BRANCH: (PC: 2140580)(sReg_a: 17)(sReg_b: 0)(addr: 2140596)
[cycle 37] RTYPE: (PC: 2140596)(sReg_a: 0)(sReg_b: 0)(dReg: 16)
....
 + Simulation terminates at cycle : 1004
```

Your assignment is to use *five_stage.c* as a guide to write a simulator for the superscalar pipeline (*superscalar.c*). You will test your simulator using your own generated traces and then evaluate the architecture on the traces provided. You will submit your code, test results, results of comparing different design techniques and the output of your simulations. Following are the main tasks in that project:

Task 1:

Modify *five_stage.c* to simulate a superscalar with two pipelines, the first for ALU, branch/jump and SPECIAL instructions and the other for load/store instructions. The IF stage fetches (and buffers) two consecutive instructions every cycle and the ID stage decodes and reads the registers for two consecutive instructions every cycle. A dynamic scheduler will then send up to two instructions every cycle from the ID stage to the EX stages (when an instruction is moved from ID to EX, we say that the instruction is *issued)*. Note that *five_stage.c* does not deal with potential hazards (and hence does not simulate correct execution). Your superscalar simulator should correctly deal with hazards as follows:

Structural hazards: If during a given cycle the two instructions in the ID stage are to be issued to the same pipeline (structural hazard), then only the first instruction is issued and the second one remains in the ID stage. Consequently, only one new instruction is fetched and one instruction is decoded during this cycle. Note that you need to keep track of the order of the two instructions that are in the ID stage so that you do not issue instructions out of order.

Data hazards: Assuming that the architecture supports hardware forwarding, data hazards will be avoided by a dynamic scheduler which will not issue an instruction (move it from ID to EX) if it detects a data hazard. Specifically:

1. The scheduler will not issue two instructions in the same cycle if one of them will use the result of the other (two data dependent instructions). Forwarding will not be possible in this situation.
2. The scheduler will not issue an instruction that read data from a specific register, R at a given cycle, If during that cycle, the EX stage was processing a *load* instruction that will write into R. This means that one of the pipelines (or possibly both) will stall (by inserting a no-op) until the *load* instruction fetches the data from memory.

Note that if only one instruction is issued in a given cycle, then only one instruction will be fetched and one will be decoded in that cycle. If no instruction is issued in a given cycle, then no instructions will be fetched and decoded in that cycle. Again, you need to keep track of the order of the two instructions that are in the ID stage so that you do not issue instructions out of order.

Control hazards: Assuming that branches and jumps are resolved in the ID stage, you will simulate static avoidance of control hazards by the compiler. That is, you will assume that the compiler adds a no-op after each branch or jump instruction. Note that the traces you are given are dynamic traces that do not include such no-ops. Hence, while reading instructions from the trace file, you should add a no-op after each branch and jump instruction.

Task 2:

You need to verify the correctness of your simulator. To help you with this task, the following program, *trace_generator.c*, may be used to build your own test trace files (very short sequence of instructions). This program takes the name of the trace file that you want to create as a command line argument and will prompt you for the 5 fields of the “instruction” struct for each instruction that you want to include in the trace. You may verify the correctness of your simulation by testing it on multiple test traces that you specifically create to test specific features/scenarios.

Task 3:

Run your simulator on the given long and short traces.

What your group should upload to the box folder specified to you in the email that the TA sent you:

1) Your source code for *superscalar.c* that take 2 arguments; the input trace file and the *trace_view_on* switch (in that order). In case the last parameter is not specified, its default values should be 0. When *trace_view_on* = 1 *superscalar.c*, should produce two lines for every cycle (each line having the same format as in *five_stage.c*).

2) A pdf file which contains
   a. The test traces that you used to test your simulator. For each test trace, you should provide the result of running it and an explanation of the functionality that you were testing.
   b. The result of running the short traces that you used in task 2 on both *five_stage.c* and *superscalar.c* with *trace_view_on* = 1. Note that the TA will run her own short traces, in addition to your own test traces, to test the correctness of your program.
   c. The result of running your simulators with *trace_view_on* = 0 for each short and long trace file.

Note: Prior to submission, make sure your code compiles and runs on the CS Linux cluster (ssh linux.cs.pitt.edu) which will be used for grading. You should also generate all submitted results on this cluster.

Grading Criteria: The maximum grade for the project will be 80 points distributed as follows:

Grade Percentage	*Criteria*
15	a basic superscalar simulator which compiles and runs
15	Correctness testing of the simulator (your own verification effort)
10	Results of running on the long and short traces
10	Handling control hazards correctly (passing the TA tests)
15	Handling structural hazards correctly (passing the TA tests)
15	Handling data hazards correctly (passing the TA tests)

Normally, all members of a team will receive the same grade for the project. However, if any member of the team complains about the lack of contribution from other members, then I will set up an oral interview with all the members of the team to discuss the contribution of each member and determine if I should assign different grades to different members.

NOTE: The files *five_stage.c, trace_generator.c* and *CPU.h* can be copied from /afs/cs.pitt.edu/courses/1541.