ECE #563

# 5-Stage MIPS Floating-Point Pipeline Processor

Zhiping Wang | Unity ID: zwang79 | Student ID: 200265085

03.17.2019

## Objective

The goal of this project is to design and implement a C++ cycle-accurate simulator of a 5-stage MIPS-like pipelined processor.

## Result

Both the integer testcases and floating-point testcases are matched accurately.

## Data Structure

*Data memory:*

```
unsigned char *data_memory;
```

Data memory is where 32-bit data is stored with two parameters, mem_size and mem_latency, that was defined by the constructor. Pipelined processor read out or write in data by LW/SW instruction. Also, data stored in the memory is initialized to 0xFF by the constructor at the beginning of every simulation.

*Instruction memory:*

```
instruction_t instr_memory[PROGRAM_SIZE];
```

Instruction memory is defined as instruction_t type. The load_program function translates and decomposes the assembly code into instruction_t type variables like opcode, src1, etc. and store them into instruction memory for IF stage to fetch specific instruction.

*General purpose registers:*

General purpose registers are the simulation of 32 register files of microprocessor, which stores the value of operands like R1 and F1.

```
unsigned gp_int_register[NUM_GP_REGISTERS];
unsigned gp_fp_register[NUM_GP_REGISTERS];
```

*Special purpose registers:*

```
unsigned sp_register[NUM_STAGES][NUM_SP_REGISTERS];
```

Special purpose registers are the 5 registers that link different stages of the processor, like IF_ID. Each of entry of this 2-d array has specific registers that stores the intermedia values that propagates between stages in each clock cycle, including PC, NPC, IR, A, B, IMM, COND, ALU_OUTPUT, LMD. What we basically need to do is to update these values every clock cycle.

*Instruction registers:*

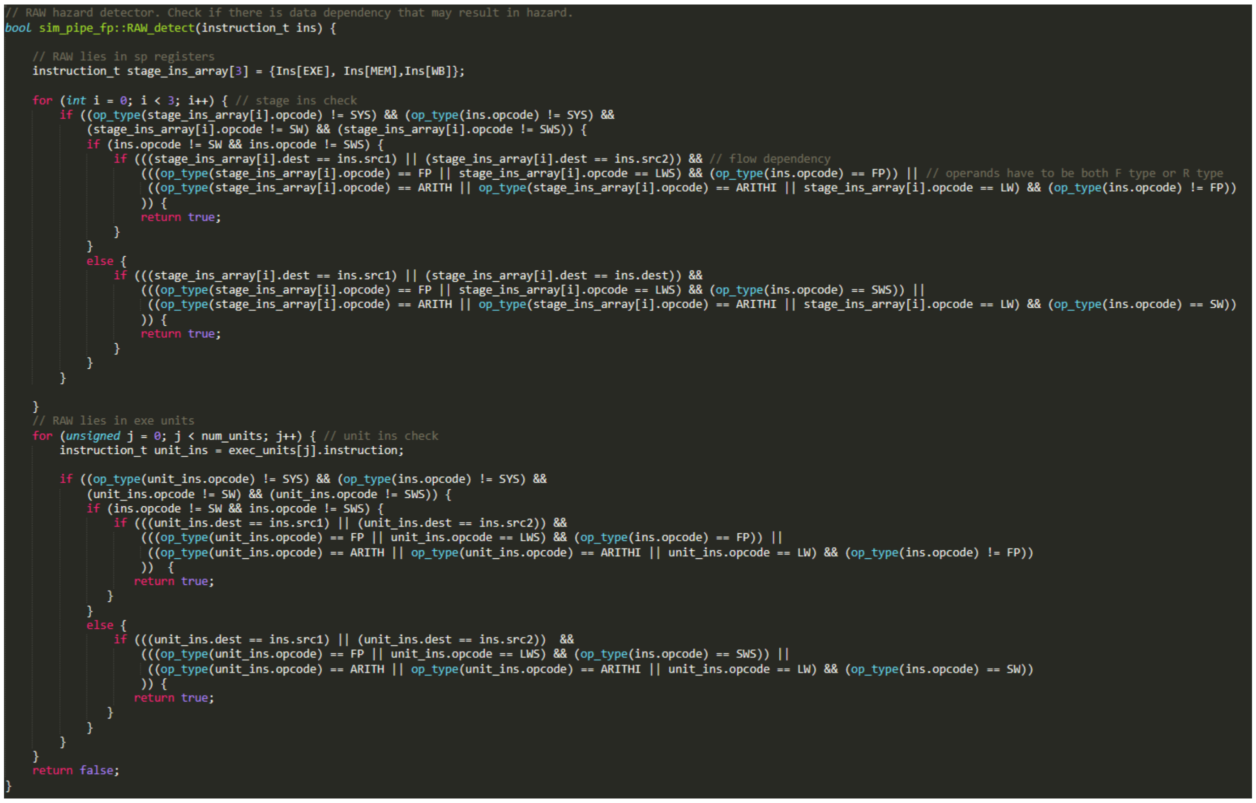
```
instruction_t Ins[NUM_STAGES];
```

While the instruction memory is defined as instruction_t type, the sp registers cannot store the instructions directly. I simply define a new instruction_t type array to propagate the instructions, so that the special values stored in the instruction can be easily extracted: Ins[ID].opcode, Ins[ID].src1, etc..

## Data Hazard Handling

### *RAW Hazard*

RAW hazards are detected on ID stage and eliminated once the dependent instruction arrives at WB stage. Once detected, ID will insert a NOP to ID/EXE sp registers. The detection of RAW is kind of tricky. Not only is flow dependency required, but the operands also need to be both F type or R type. Moreover, RAW hazard lies in both sp registers and execution units. Because, once the instruction goes into the execution units, the EXE sp registers are flushed.

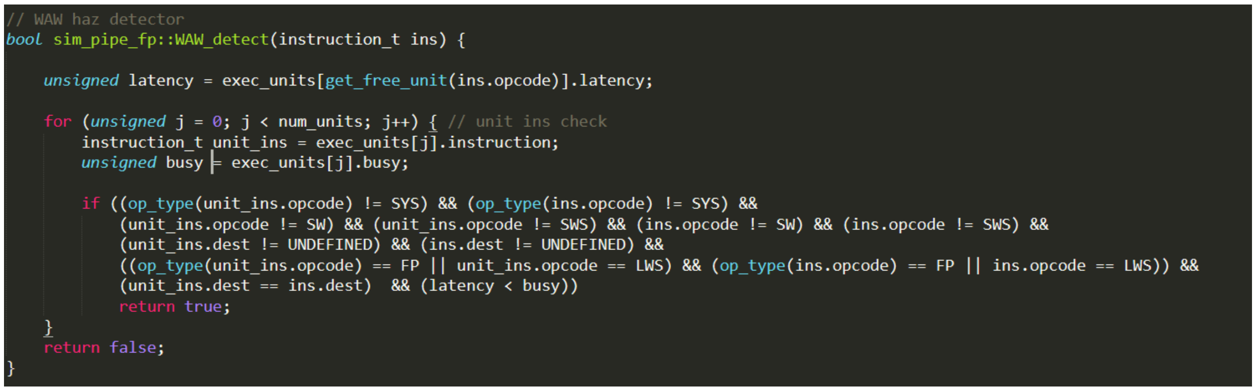


```
// RAW hazard detector. Check if there is data dependency that may result in hazard.
bool sim_pipe_fp::RAW_detect(instruction_t ins) {

    // RAW lies in sp registers
    instruction_t stage_ins_array[3] = {Ins[EXE], Ins[MEM],Ins[WB]};

    for (int i = 0; i < 3; i++) { // stage ins check
        if ((op_type(stage_ins_array[i].opcode) != SYS) && (op_type(ins.opcode) != SYS) &&
            (stage_ins_array[i].opcode != SW) && (stage_ins_array[i].opcode != SWS)) {
            if (ins.opcode != SW && ins.opcode != SWS) {
                if (((stage_ins_array[i].dest == ins.src1) || (stage_ins_array[i].dest == ins.src2)) && // flow dependency
                    (((op_type(stage_ins_array[i].opcode) == FP || stage_ins_array[i].opcode == LWS) && (op_type(ins.opcode) == FP)) || // operands have to be both F type or R type
                     ((op_type(stage_ins_array[i].opcode) == ARITH || op_type(stage_ins_array[i].opcode) == ARITHI || stage_ins_array[i].opcode == LW) && (op_type(ins.opcode) != FP))
                    )) {
                    return true;
                }
            }
            else {
                if (((stage_ins_array[i].dest == ins.src1) || (stage_ins_array[i].dest == ins.dest)) &&
                    (((op_type(stage_ins_array[i].opcode) == FP || stage_ins_array[i].opcode == LWS) && (op_type(ins.opcode) == SWS)) ||
                     ((op_type(stage_ins_array[i].opcode) == ARITH || op_type(stage_ins_array[i].opcode) == ARITHI || stage_ins_array[i].opcode == LW) && (op_type(ins.opcode) == SW))
                    )) {
                    return true;
                }
            }
        }

    }
    // RAW lies in exe units
    for (unsigned j = 0; j < num_units; j++) { // unit ins check
        instruction_t unit_ins = exec_units[j].instruction;

        if ((op_type(unit_ins.opcode) != SYS) && (op_type(ins.opcode) != SYS) &&
            (unit_ins.opcode != SW) && (unit_ins.opcode != SWS)) {
            if (ins.opcode != SW && ins.opcode != SWS) {
                if (((unit_ins.dest == ins.src1) || (unit_ins.dest == ins.src2)) &&
                    (((op_type(unit_ins.opcode) == FP || unit_ins.opcode == LWS) && (op_type(ins.opcode) == FP)) ||
                     ((op_type(unit_ins.opcode) == ARITH || op_type(unit_ins.opcode) == ARITHI || unit_ins.opcode == LW) && (op_type(ins.opcode) != FP))
                    ))  {
                   return true;
               }
            }
            else {
                if (((unit_ins.dest == ins.src1) || (unit_ins.dest == ins.src2))  &&
                    (((op_type(unit_ins.opcode) == FP || unit_ins.opcode == LWS) && (op_type(ins.opcode) == SWS)) ||
                     ((op_type(unit_ins.opcode) == ARITH || op_type(unit_ins.opcode) == ARITHI || unit_ins.opcode == LW) && (op_type(ins.opcode) == SW))
                    )) {
                   return true;
               }
            }
        }
    }
    return false;
}
```

### *WAW Hazard*

WAW hazards are detected and refreshed on ID stage every clock cycle. The output dependency is checked among all execution units. Like RAW, the operands need to be both F type or R type as well. Meanwhile, the remaining execution time of the instruction already inside the execution unit needs to be longer than the required execution time of the following instruction waiting at the ID stage. This can be simplified as:

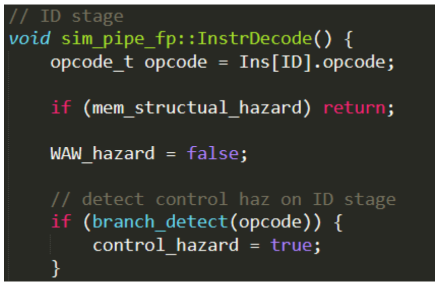
second_instruction.latency < first_instruction.buzy



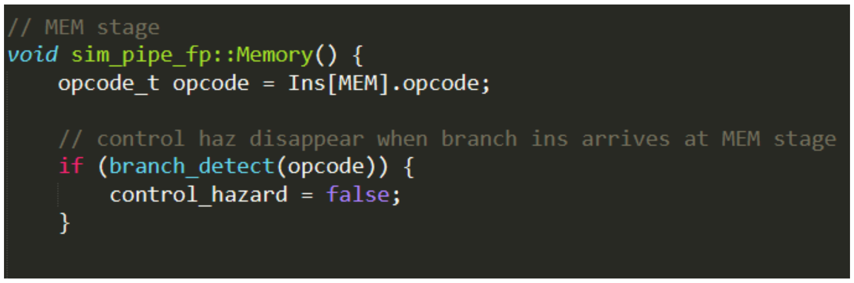
```
// WAW haz detector
bool sim_pipe_fp::WAW_detect(instruction_t ins) {

    unsigned latency = exec_units[get_free_unit(ins.opcode)].latency;

    for (unsigned j = 0; j < num_units; j++) { // unit ins check
        instruction_t unit_ins = exec_units[j].instruction;
        unsigned busy = exec_units[j].busy;

        if ((op_type(unit_ins.opcode) != SYS) && (op_type(ins.opcode) != SYS) &&
            (unit_ins.opcode != SW) && (unit_ins.opcode != SWS) && (ins.opcode != SW) && (ins.opcode != SWS) &&
            (unit_ins.dest != UNDEFINED) && (ins.dest != UNDEFINED) &&
            ((op_type(unit_ins.opcode) == FP || unit_ins.opcode == LWS) && (op_type(ins.opcode) == FP || ins.opcode == LWS)) &&
            (unit_ins.dest == ins.dest)  && (latency < busy))
            return true;
    }
    return false;
}
```

## Control Hazard handle

Control Hazard is handled similarly to integer pipeline. Once decode a branch instruction on ID stage, a control hazard is introduced. NOPs will keep being inserted at IF stage until the branch instruction arrives at MEM stage.



```
// ID stage
void sim_pipe_fp::InstrDecode() {
    opcode_t opcode = Ins[ID].opcode;

    if (mem_structual_hazard) return;

    WAW_hazard = false;

    // detect control haz on ID stage
    if (branch_detect(opcode)) {
        control_hazard = true;
    }
```



```
// MEM stage
void sim_pipe_fp::Memory() {
    opcode_t opcode = Ins[MEM].opcode;

    // control haz disappear when branch ins arrives at MEM stage
    if (branch_detect(opcode)) {
        control_hazard = false;
    }
```

## Structural Hazard handle

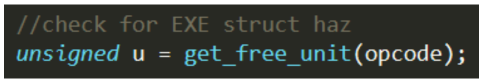
*Memory Structural Hazard*

This part is the same as the integer pipeline. Hazard is introduced due to memory latency. IF, ID, EXE stages are stuck until the memory structural hazard disappears.

*Execution Units Structural Hazard*

EXE structural hazard is detected on ID. Similarly, IF, ID stages would be stuck until the hazard is eliminated. There are two scenarios that would cause this kind of structural hazard.

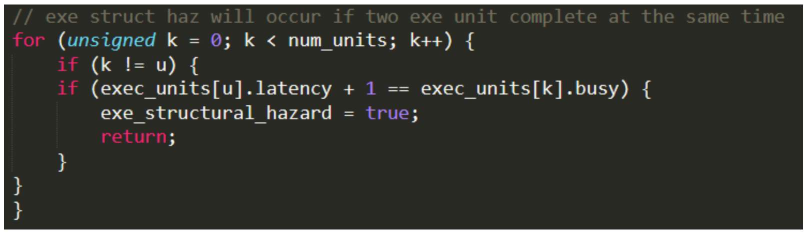
First is no execution unit being available for instruction pending on ID. This scenario can be covered by the given function:



```
//check for EXE struct haz
unsigned u = get_free_unit(opcode);
```

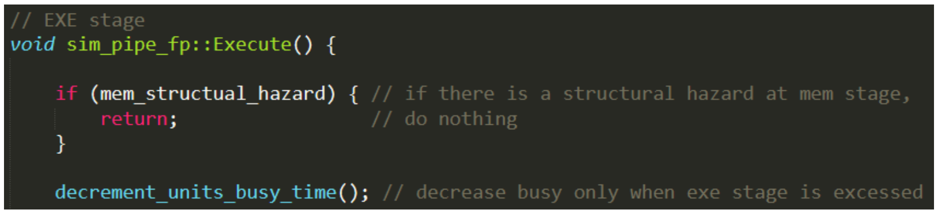
If the return value is UNDEFINED and the instruction is not NOP or EOP, EXE structural hazard is detected.

Second circumstance is that when two execution units complete at the same time, only one can propagate. This can be handled before the instruction goes into execution stage.



```
// exe struct haz will occur if two exe unit complete at the same time
for (unsigned k = 0; k < num_units; k++) {
    if (k != u) {
    if (exec_units[u].latency + 1 == exec_units[k].busy) {
        exe_structural_hazard = true;
        return;
    }
}
}
```

Decreasing of the unit busy time can only implement when the EXE stage is working. In other words, busy time will not decrease when there is memory structural hazard.



```
// EXE stage
void sim_pipe_fp::Execute() {

    if (mem_structural_hazard) { // if there is a structural hazard at mem stage,
        return;                  // do nothing
    }

    decrement_units_busy_time(); // decrease busy only when exe stage is excessed
```

## Hazard Priority

Memory structural hazard > Execution unit structural hazard > RAW > (WAW = Control hazard) (Because WAW and Control hazard are independent).

## Summary

I though I could use my integer pipeline structure as a good start. I was wrong.