Chuck Richardson

ECE 563

Dr. Michela Becchi

2/21/2023

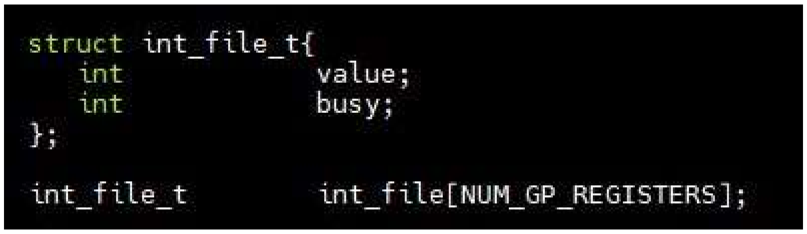
Project 1-Floating-Point Pipeline Report

1. Objective

The objective of this project was to implement a 5-stage MIPS floating-point pipelined processor in c++. This pipelined processor contains five stages including Fetch(IF), Decode(ID), Execute(EXE), Memory(MEM), and Writeback(WB). This design also includes 32 integer general-purpose registers(R0-R31) as well as 32 floating-pointer general-purpose registers (F0-F31). This MIPS processor needed to handle basic integer register arithmetic instructions such as ADD, SUB, XOR, as well as memory operations such as LOAD and STORE. For the floating-point operations, it needed to handle instructions such as basic floating point arithmetic operations such as ADDS, SUBS, MULTS, and DIVS and memory operations such as LWS and SWS. This pipelined processor also needed to handle control/branching instructions such as BEQZ, BNEZ, BLTZ, etc. This pipelined processor also had to handle hazards such as structural hazards pertaining to resource conflicts, data hazards pertaining to dependencies between instructions, and control hazards pertaining to branches and other control flow instructions. During this project the following metrics were kept track of such as the IPC(Instructions per Clock Cycle), Total Clock Cycles, Total Instructions, and Number of Stalls.

2. Data Structures

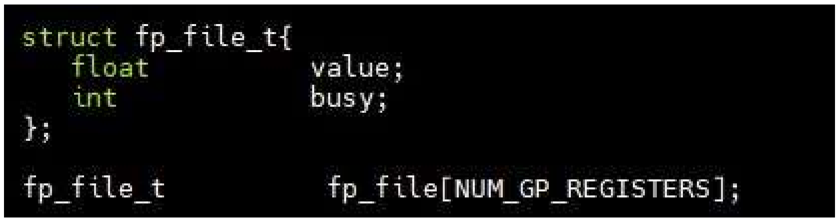
The MIPS pipelined processor needed special purpose pipelined registers, integer general-purpose registers, floating-point general-purpose registers, and an instruction pipelined register. For the integer general-purpose registers, I used an array with the size of the number of general-purpose registers. These general-purpose registers have a struct of type int_file_t, in which it contains two integer values to hold the value and to check if it is busy being written to. This is shown in Figure 1.



```
struct int_file_t{
   int            value;
   int            busy;
};

int_file_t        int_file[NUM_GP_REGISTERS];
```

Figure 1: Integer General-Purpose Register Implementation

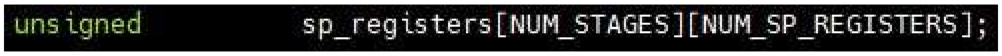
For the floating-point general-purpose registers, I used an array with the size of the number of general-purpose registers. These general-purpose registers have a struct of type fp_file_t, in which it contains one floating-point value to hold the value and an integer value to check if it is busy being written to. This is shown in Figure 2.



```
struct fp_file_t{
   float          value;
   int            busy;
};

fp_file_t          fp_file[NUM_GP_REGISTERS];
```

Figure 2: Floating-Point Register Implementation

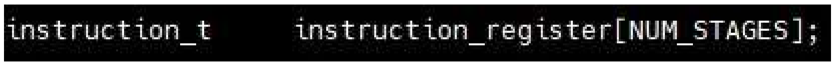
For the special purpose pipelined registers, I had to use a two-dimensional array in which the first index was the number of stages and 2nd index was the number of special purpose registers. This is shown in Figure 3.



```
unsigned          sp_registers[NUM_STAGES][NUM_SP_REGISTERS];
```

Figure 3: Special Purpose Registers Implementation

For the instruction pipelined register, I used an array with the size of the number of stages to hold the instructions for each collective stage. This instruction register array has struct of type instruction_t to hold the opcode, src1, src2, etc. This is shown in Figure 4.



```
instruction_t     instruction_register[NUM_STAGES];
```

Figure 4: Instruction Pipelined Register Implementation

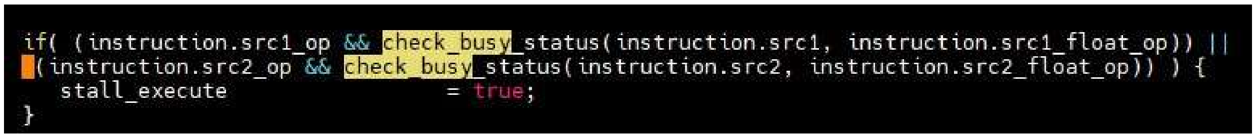
3. Hazard Handling

There are three types of hazards to be handled in the MIPS pipeline including structural hazards, data hazards, and control hazards.

3.2. Data Hazards

3.2.1. RAW Hazards

In the MIPS Pipeline there exists data dependencies between instructions. For RAW(Read-After-Write) Hazards we must stall the decoding of the instruction until the value for a specified register that is needed to be written back has completed in the writeback stage. We check if the general-purpose registers are busy being written to and if they are then we stall. Once the value is written back, the instruction from decode stage can then be decoded and propagated to the execute stage. This is shown in Figure 5.

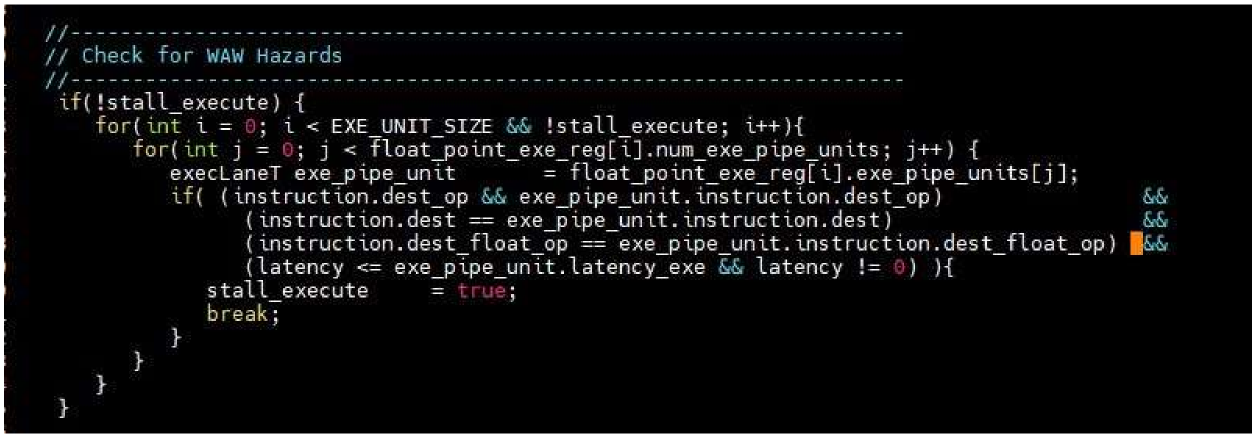


```
if( (instruction.src1_op && check_busy_status(instruction.src1, instruction.src1_float_op)) ||
 (instruction.src2_op && check_busy_status(instruction.src2, instruction.src2_float_op)) ) {
   stall_execute              = true;
}
```

Figure 5: Data Hazard Checking

3.2.2. WAW Hazards

For WAW(Write-After-Write) Hazards, I first check if the destination register in the decode stage and the destination register in the execute stage are being used or/are going to be presently in use. Then following that I check if the two instruction destination registers are equal. Then I check if the latency of the current instruction will be less than or equal to the instruction inside the execute units' current latency, thus determining if the instruction in the decode stage will finish before the instruction being executed. All these factors combined I determine if there will be a WAW Hazard. This is shown in Figure 6.

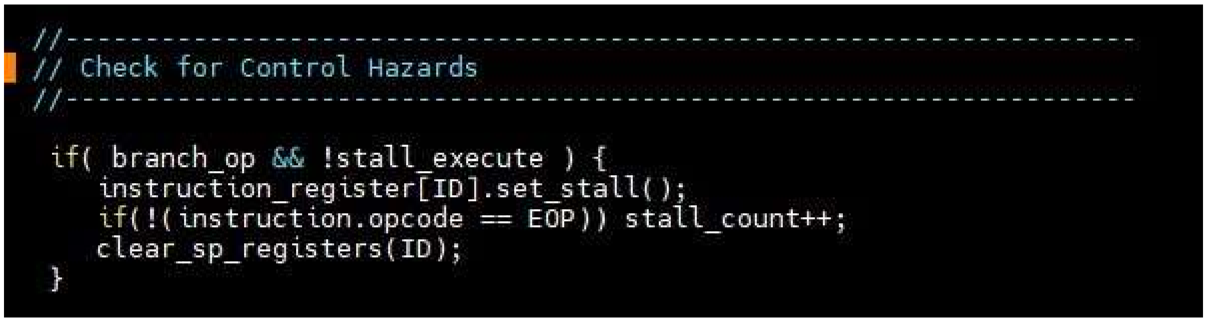


```
//--------------------------------------------------------------
// Check for WAW Hazards
//--------------------------------------------------------------
 if(!stall_execute) {
    for(int i = 0; i < EXE_UNIT_SIZE && !stall_execute; i++){
       for(int j = 0; j < float_point_exe_reg[i].num_exe_pipe_units; j++) {
          execLaneT exe_pipe_unit       = float_point_exe_reg[i].exe_pipe_units[j];
          if( (instruction.dest_op && exe_pipe_unit.instruction.dest_op)            &&
                (instruction.dest == exe_pipe_unit.instruction.dest)                &&
                (instruction.dest_float_op == exe_pipe_unit.instruction.dest_float_op) &&
                (latency <= exe_pipe_unit.latency_exe && latency != 0) ){
             stall_execute     = true;
             break;
          }
       }
    }
 }
```

Figure 6: WAW Hazard Checking

3.3. Control Hazards

Control Hazards are caused by conditional and unconditional branching such as BEQZ,BLTZ, JUMP, etc. This detection happens in the decode stage. This will also change what instruction is being fetched, whether the branch condition is true or false. At the decode stage we will check if the instruction being executed at the execute stage is a branch if it is then we stall the decode stage as well as the fetch stage. Then once the target of the branch is calculated we will then execute the next instruction whether it be the instruction at the target of the branch or the next instruction in program order. This is checking is shown in Figure 7.



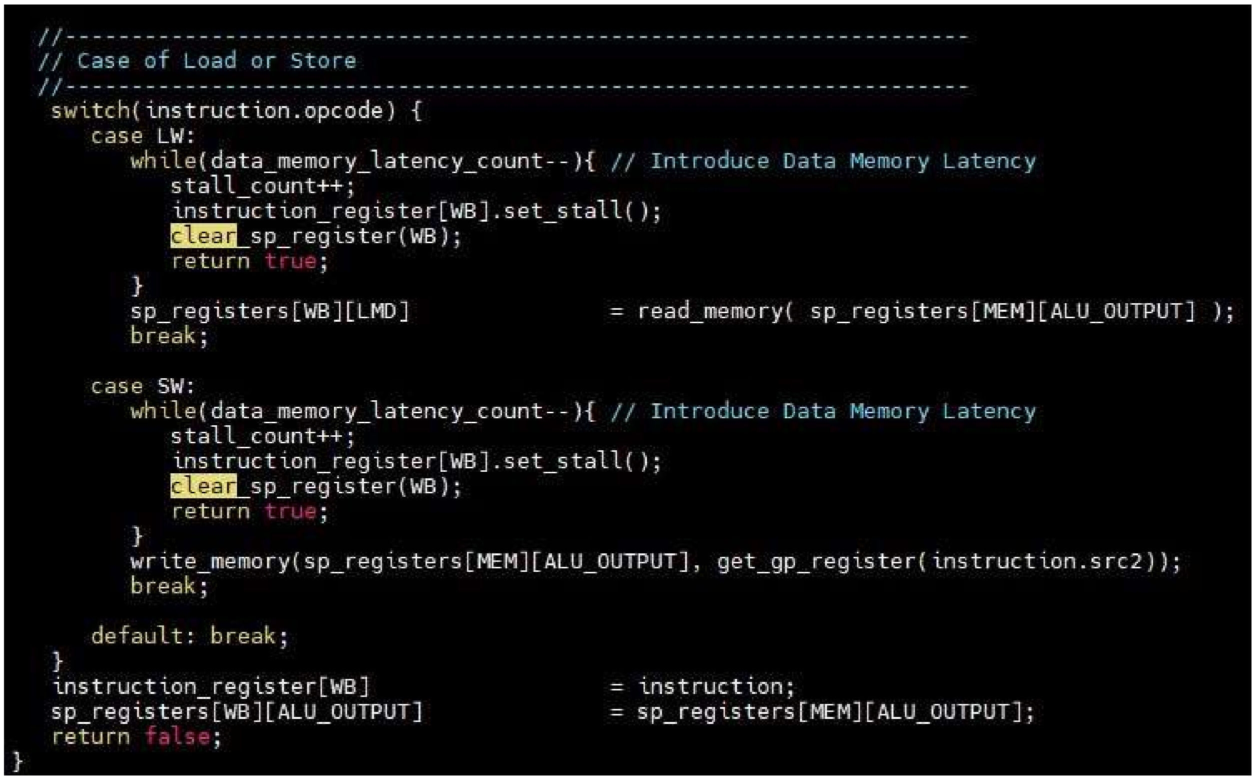
```
//--------------------------------------------------------------
// Check for Control Hazards
//--------------------------------------------------------------

 if( branch_op && !stall_execute ) {
    instruction_register[ID].set_stall();
    if(!(instruction.opcode == EOP)) stall_count++;
    clear_sp_registers(ID);
 }
```

Figure 7: Control Hazard Checking

3.4. Structural Hazards

3.4.1. Memory Latency

One source of a Structural Hazards is memory latency. This issue is caused whenever a load or store instruction reaches the memory stage. If the instruction at the memory stage is a load or a store the other stages will be stalled such as IF, ID, and EXE until the memory stage has been completed its operation. Note we also must stall the Writeback Stage in this case due to the ordering of our stages in the run function. The implementation of data memory latency and stalls are shown in Figure 8.



```
  //------------------------------------------------------------------
  // Case of Load or Store
  //------------------------------------------------------------------
   switch(instruction.opcode) {
      case LW:
         while(data_memory_latency_count--){ // Introduce Data Memory Latency
            stall_count++;
            instruction_register[WB].set_stall();
            clear_sp_register(WB);
            return true;
         }
         sp_registers[WB][LMD]               = read_memory( sp_registers[MEM][ALU_OUTPUT] );
         break;

      case SW:
         while(data_memory_latency_count--){ // Introduce Data Memory Latency
            stall_count++;
            instruction_register[WB].set_stall();
            clear_sp_register(WB);
            return true;
         }
         write_memory(sp_registers[MEM][ALU_OUTPUT], get_gp_register(instruction.src2));
         break;

      default: break;
   }
   instruction_register[WB]              = instruction;
   sp_registers[WB][ALU_OUTPUT]           = sp_registers[MEM][ALU_OUTPUT];
   return false;
}
```

Figure 8: Structural Hazard Implementation

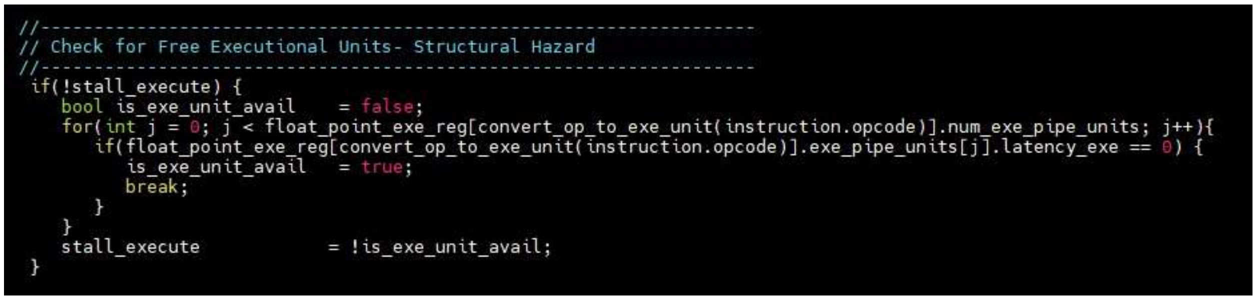
3.4.2. Floating-Point Execute Structural Hazard

Another source of a structural hazard might occur due to execute unit contention. For example, say let's say we have a series of instructions shown below:

DIV F1 F2 F3

DIV F3 F4 F5

These series of instructions might cause contention for the floating-point divider since they both need to use the divider during the execute stage, so therefore we will have to stall the execution of the second instruction until the divider finishes executing the first instruction. This checking is shown in Figure 9.

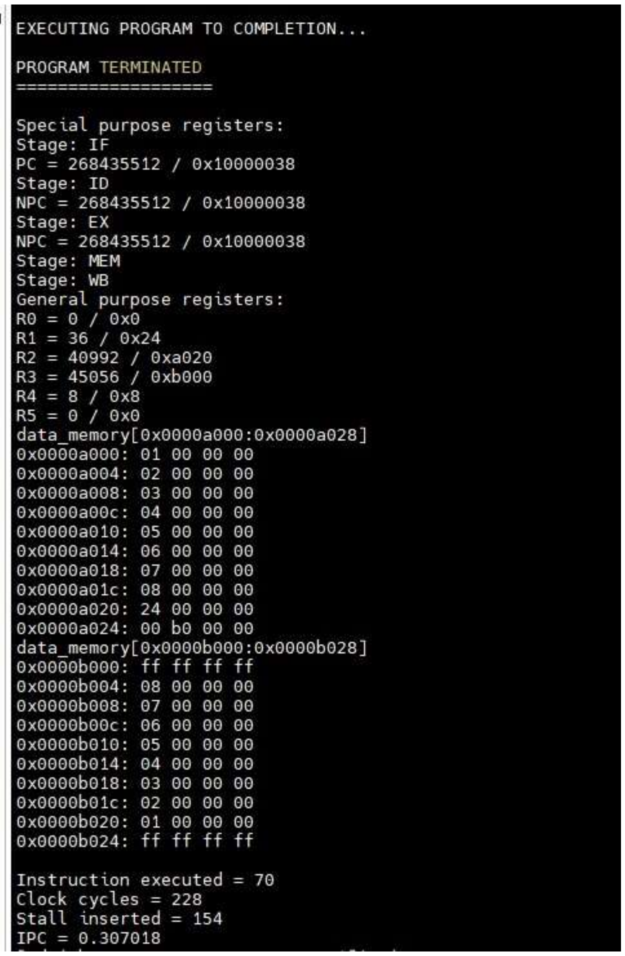


```
//------------------------------------------------------------------
// Check for Free Executional Units- Structural Hazard
//------------------------------------------------------------------
 if(!stall_execute) {
    bool is_exe_unit_avail    = false;
    for(int j = 0; j < float_point_exe_reg[convert_op_to_exe_unit(instruction.opcode)].num_exe_pipe_units; j++){
       if(float_point_exe_reg[convert_op_to_exe_unit(instruction.opcode)].exe_pipe_units[j].latency_exe == 0) {
          is_exe_unit_avail    = true;
          break;
       }
    }
    stall_execute            = !is_exe_unit_avail;
 }
```

Figure 9: Structural Hazard at Execute Checking

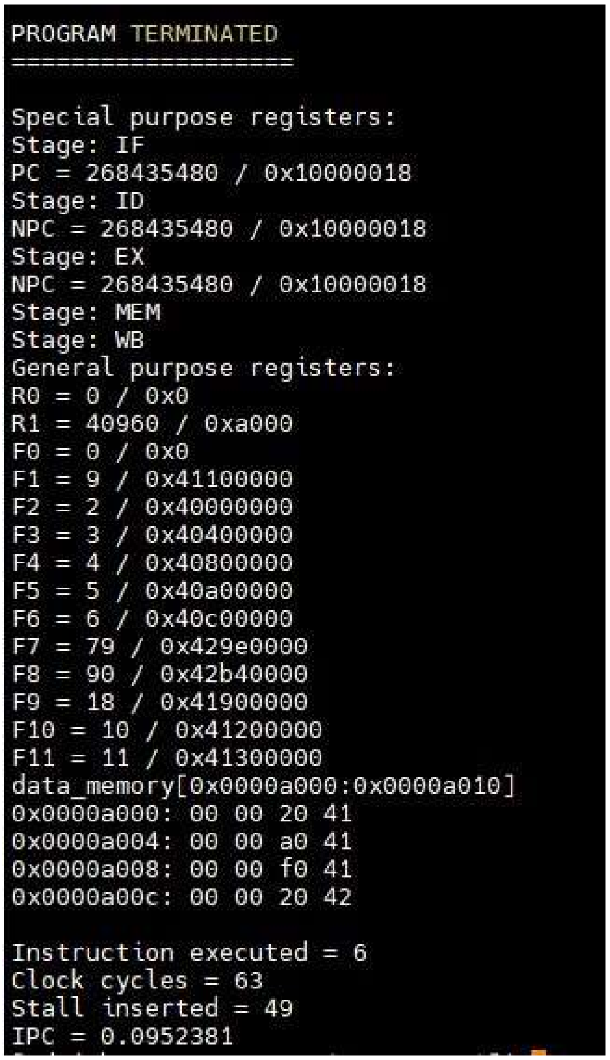
4. Conclusion

In conclusion, my MIPS Pipeline works for all testcases. I used Clion as well as valgrind to debug possible issues such as incorrect values for each respective stage and segmentation faults. My implementation for checking hazards works extremely well as well and passes all testcases. Shown below in Figure 10, is a sample of my output from testcase6 for the integer pipeline. Shown in Figure 11 is my output from my floating-point pipeline for testcase_fp5.



```
EXECUTING PROGRAM TO COMPLETION...

PROGRAM TERMINATED
===================

Special purpose registers:
Stage: IF
PC = 268435512 / 0x10000038
Stage: ID
NPC = 268435512 / 0x10000038
Stage: EX
NPC = 268435512 / 0x10000038
Stage: MEM
Stage: WB
General purpose registers:
R0 = 0 / 0x0
R1 = 36 / 0x24
R2 = 40992 / 0xa020
R3 = 45056 / 0xb000
R4 = 8 / 0x8
R5 = 0 / 0x0
data_memory[0x0000a000:0x0000a028]
0x0000a000: 01 00 00 00
0x0000a004: 02 00 00 00
0x0000a008: 03 00 00 00
0x0000a00c: 04 00 00 00
0x0000a010: 05 00 00 00
0x0000a014: 06 00 00 00
0x0000a018: 07 00 00 00
0x0000a01c: 08 00 00 00
0x0000a020: 24 00 00 00
0x0000a024: 00 b0 00 00
data_memory[0x0000b000:0x0000b028]
0x0000b000: ff ff ff ff
0x0000b004: 08 00 00 00
0x0000b008: 07 00 00 00
0x0000b00c: 06 00 00 00
0x0000b010: 05 00 00 00
0x0000b014: 04 00 00 00
0x0000b018: 03 00 00 00
0x0000b01c: 02 00 00 00
0x0000b020: 01 00 00 00
0x0000b024: ff ff ff ff

Instruction executed = 70
Clock cycles = 228
Stall inserted = 154
IPC = 0.307018
```

Figure 10. Testcase 6 Output



```
PROGRAM TERMINATED
===================

Special purpose registers:
Stage: IF
PC = 268435480 / 0x10000018
Stage: ID
NPC = 268435480 / 0x10000018
Stage: EX
NPC = 268435480 / 0x10000018
Stage: MEM
Stage: WB
General purpose registers:
R0 = 0 / 0x0
R1 = 40960 / 0xa000
F0 = 0 / 0x0
F1 = 9 / 0x41100000
F2 = 2 / 0x40000000
F3 = 3 / 0x40400000
F4 = 4 / 0x40800000
F5 = 5 / 0x40a00000
F6 = 6 / 0x40c00000
F7 = 79 / 0x429e0000
F8 = 90 / 0x42b40000
F9 = 18 / 0x41900000
F10 = 10 / 0x41200000
F11 = 11 / 0x41300000
data_memory[0x0000a000:0x0000a010]
0x0000a000: 00 00 20 41
0x0000a004: 00 00 a0 41
0x0000a008: 00 00 f0 41
0x0000a00c: 00 00 20 42

Instruction executed = 6
Clock cycles = 63
Stall inserted = 49
IPC = 0.0952381
```

Figure 11: Testcase_fp5 Output