5-STAGE MIPS-LIKE FLOATING-POINT PIPELINED PROCESSOR SIMULATOR

ECE-563 Project-1 Report

- Vijayalakshmi Parthiyoor Sundar (200201162)

DATA STRUCTURES

Floating-point Units:

- The data structure to handle the multiple FP units at the execute stage is handled by implementing a structure for each of the execution unit, with each of that configured to having the number of instances with different latencies.
- To implement Floating-point registers, a structure **"fpFileT"** is declared with its members as value and busy (of type float). The value stores the value of the respective register, while the busy is used to deal with RAW and WAW dependences. The general-purpose register file was then implemented as an array of 32 as **fpFile[NUM_GP_REGISTERS]** (of type fpFileT).

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

struct execLaneT{
   instructT      instruct;
   int            ttl;
   unsigned       b;
   unsigned       exNpc;

   execLaneT(){
      ttl            = 0;
   }

};

struct execUnitT{
   execLaneT      *lanes;
   int            numLanes;
   int            latency;

   execUnitT(){
      lanes          = NULL;
      numLanes       = 0;
      latency        = 0;
   }

   void init(int numLanes, int latency){
      ASSERT( latency > 0, "Impractical latency found (=%d)", latency );
      ASSERT( numLanes > 0, "Unsupported number of lanes (=%d)", numLanes );
      this->numLanes += numLanes;
      this->latency   = latency;
      lanes           = (execLaneT*)realloc(lanes, this->numLanes * sizeof(execLaneT));
   }
```

Instruction Memory:

To implement this, structure **"instructT"** is declared with its primary members being the opcode (of type opcode_t), destination register, source-1 register, source-2 registers, and an immediate field (of type uint32_t).
Using this structure, the instruction memory is declared as **"instMemory"** of type instructT**, to help accommodate the instruction as it is into what could be perceived as an array of instructions. The **instMemory** is populated in the parser (load program) with the instructions.

```
typedef struct instructT* instructPT;

struct instructT{
   opcode_t             opcode;
   uint32_t             dst;
   uint32_t             src1;
   uint32_t             src2;
   uint32_t             imm;
};
instructPT         *instMemory;
```

General Purpose Registers:

To implement these registers, a structure **"gprFileT"** is declared with its members as value and busy (of type int). The value stores the value of the respective register, while the busy is used to deal with RAW and WAW dependences. The general-purpose register file was then implemented as an array of 32 as **gprFile[NUM_GP_REGISTERS]** (of type gprFileT).

```
78
79 class sim_pipe{
80
81 public:
82    struct gprFileT{
83       int            value;
84       int            busy;
85    };
86
87    int               cycleCount;
88    int               instCount;
89    int               latCount;
90    int               numStalls;
91    bool              latency;
92
93    gprFileT          gprFile[NUM_GP_REGISTERS];
```

Declaration of general purpose registers

Pipeline Registers:

To implement the pipeline registers, a two-dimensional array is declared for each stage of the MIPS architecture, consisting of all the pipeline registers. It is declared as **pipeReg[NUM_STAGES][NUM_SP_REGISTERS]** (of type uint32_t). This will keep track of values of the special purpose registers for each of the stages.

```
95    uint32_t          pipeReg[NUM_STAGES][NUM_SP_REGISTERS];
96
```

Declaration of pipeline registers

HAZARD HANDLING

Data Hazards and Structural Hazards:

RAW hazards:

To handle the RAW hazards, we will require to stall the decode stage until the instruction moves to WB, there by introducing two stalls into EX and MEM stages. The Decode Stage checks for dependency by checking the corresponding **"busy"** of the source registers of both gp registers and fp registers, and if either of them are set, the decode stage stalls the EX stage (also making its pipeline registers UNDEFINED). This is resolved once the WB stage unlocks busy, there by propagating the instruction to the EX stage, and the pipeline registers for the stages are updated accordingly.

WAW hazards:

To handle the WAW hazards, firstly, it is checked if the destinations of the instruction in decode stage and the instructions in the execute stage are valid, then the destination registers are checked if they are equal, then the latency of the current instruction (according to execution unit) is checked against the "time to live" of the instruction inside execute stage. The decode is stalled until the hazard is cleared up.

Contention at Memory:

If two instructions at execute stage end at the same time, there will be a structural hazard at the memory stage. So before pushing an instruction into any execution unit, it is checked if there is any contention, that is if the current instruction's latency is equal to the "time to live" of any instruction in the execution stage. If that is the case, the instruction is pushed only after the contentions are cleared.

```
//------------------------------ RAW BEGIN ------------------------------------
if( (instruct.src1Valid && regBusy(instruct.src1, instruct.src1F)) ||
      (instruct.src2Valid && regBusy(instruct.src2, instruct.src2F)) ) {
   stallEx                = true;
}
//------------------------------ RAW ENDS -------------------------------------
//---------------------- WAW & EX CONTENTION BEGINS ---------------------------
for(int i = 0; i < EXEC_UNIT_TOTAL && !stallEx; i++){
   for(int j = 0; j < execFp[i].numLanes; j++){
      execLaneT lane = execFp[i].lanes[j];
      if(latency == lane.ttl && latency != 0) {
         stallEx           = true;
         break;
      }
   }
}

if(!stallEx) {
   for(int j = 0; j < execFp[opcodeToExUnit(instruct.opcode)].numLanes; j++) {
      execLaneT lane       = execFp[opcodeToExUnit(instruct.opcode)].lanes[j];
      if( (instruct.dstValid && lane.instruct.dstValid) && // Destinations should be valid
            (instruct.dst == lane.instruct.dst) && // Reg numbers should match
            (instruct.dstF == lane.instruct.dstF) && // Both should either be R or F
            (latency <= lane.ttl && latency != 0) ){ // Latency must be LTE ttl
            stallEx     = true;
            break;
         }
      }
   }
//------------------------ WAW & EX CONTENTION ENDS ----------------------------
```

Checking for free lanes:

When an instruction is in the decode stage, you will have to check if there is any lane in the respective function unit that is free for the instruction to go through, and if the execution unit is full, there is a structural hazard incurred and the decode stage should be stalled. To handle this, the “time to live” of these lanes are checked, and if it is zero, then there is a free lane, and the instruction can go through.

```
//--------------------------- CHECKING FOR LANES BEGIN -------------------------
if(!stallEx) {
   bool isFreeLane    = false;
   for(int j = 0; j < execFp[opcodeToExUnit(instruct.opcode)].numLanes; j++){
      if(execFp[opcodeToExUnit(instruct.opcode)].lanes[j].ttl == 0) {
         isFreeLane   = true;
         break;
      }
   }
   stallEx            = !isFreeLane;
}
//--------------------------- CHECKING FOR LANES ENDS --------------------------
```

When there is a memory latency to fetch the data, the instructions following it, each in EX stage, ID stage and IF stage are stopped from proceeding, while the WB stays idle.

```
pipeReg[WB][LMD]                    = UNDEFINED;
switch(instruct.opcode) {
   case LW:
      //To introduce latency into the memory stage
      while(memFlag--){
         this->instrArray[WB].stall();
         numStalls++;
         for(int i = 0; i < NUM_SP_REGISTERS; i++) {
            this->pipeReg[WB][i]          = UNDEFINED;
         }
         return true;
      }
      pipeReg[WB][LMD]              = read_memory( pipeReg[MEM][ALU_OUTPUT] );
      break;
```

Control Hazards:

To handle this hazard, in the Decode stage, it is checked if the instruction is a branch and if the instruction in the execute stage is a branch as well. If so, the instruction is propagated into EX stage, while the decode stage is stalled. The fetch is stopped until the instruction moves out of the EX stage after the computation of PC using the current NPC and the offset, the value of which is stored in exe/mem.alu_output. This is loaded back into the fetch stage. The sp register COND is updated as required, and the PC is chosen as per COND, which will fetch the necessary instruction.

```
//-------------------------- BRANCH CONTROL BEGINS ------------------------------
bool stallBranch                    = instruct.is_branch || instrArray[EX].is_branch || intBranch();
if( stallBranch && !stallEx ) {
   this->instrArray[ID].stall();
   if(!(instruct.opcode == EOP)) numStalls++;
   for(int i = 0; i < NUM_SP_REGISTERS; i++) {
      this->pipeReg[ID][i]          = UNDEFINED;
   }
}
//-------------------------- BRANCH CONTROL ENDS --------------------------------
```

- The code works for the testcase provided, the integer testcases that were provided, and with some additional testcases that were developed for testing.
- A loopback was added to emulate the Flip Flop behaviour because of the pre-sampling of ALU-OUTPUT from the EXE stage, so the instruction receives the correct PC before the clock-cycle.

- -std=c++11 was added in the Makefile to set the standard as c++11