SNURISC5 Design Document

Computer Architecture Project #4

2021-16988 Jaewan Park

Part 1 : Push & Pop Instructions

Push and pop instructions both require accessing the register file and the data memory. They also utilize the ALU to calculate updated values for the stack register. Push acts with an input for the `rs2` register, assumes `sp` as the `rs1` register, and sets the `op2` value as 4. The ALU subtracts 4 from the value from `sp`, and then data is read from `rs2` and is written to the data memory with address equal to the decreased `sp` value. Pop acts with an input for the `rd` register, assumes `sp` as the rs1 register, and sets the `op2` value as 4. Data is first read from the data memory at the current `sp` address, and then is loaded to `rd` at the register file, and finally `sp` is increased by 4 by the ALU.

At the WB stage, loading the updated `sp` value to the register file was required, and for this the ALU output needed to be sent to the WB stage, and writing to the register file needed to be executed regardless of the `rd` update. Therefore two extra registers were added to the pipeline registers. One sends the ALU output to the end stage, which may not be equivalent to the original writeback data. Another sends a signal if the WB stage is handling a push or pop instruction, to detect whether it should additionally update the `sp` register after the original `rd` update.

The overall signals and behavior of push, pop are like the following.

	`rs1 = op1`	`rs2`	`op2`	`rd`	ALUop	Register	Memory
Push	`sp`	input	4	-	subtract	Read, Write	Write
Pop	`sp`	-	4	input	add	Read, Write	Read

Also, push and pop instructions cause various new pipeline hazards. There are two main hazards, both which can be solved via forwarding.

The first case is when an instruction called shortly after push or pop requires the access of the `sp` register. Since push or pop updates the register value by increment or decrement of 4, forwarding is required. We can get the value from the EX or MM stage, from the ALU output and forward it to the ID stage of the next instruction.

```
self.fwd_op1 = FWD_EX     if (EX.reg_rd == Pipe.ID.rs1) and rs1_oen and   \
                             (EX.reg_rd != 0) and EX.reg_c_rf_wen or      \
                             EX.reg_wb_sp and rs1_oen and                 \
                             (Pipe.ID.rs1 == STACK) else                  \
               FWD_MM     if (MM.reg_rd == Pipe.ID.rs1) and rs1_oen and   \
                             (MM.reg_rd != 0) and Pipe.MM.c_rf_wen or     \
                             MM.reg_wb_sp and rs1_oen and                 \
                             not Pipe.MM.c_rf_wen and                     \
                             (Pipe.ID.rs1 == STACK) else                  \
               FWD_MM_ALU if MM.reg_wb_sp and rs1_oen and                 \
                             Pipe.MM.c_rf_wen and                         \
                             (Pipe.ID.rs1 == STACK) else                  \
               FWD_WB     if (WB.reg_rd == Pipe.ID.rs1) and rs1_oen and   \
                             (WB.reg_rd != 0) and WB.reg_c_rf_wen or      \
                             WB.reg_wb_sp and rs1_oen and                 \
                             not WB.reg_c_rf_wen and                      \
                             (Pipe.ID.rs1 == STACK) else                  \
               FWD_WB_ALU if WB.reg_wb_sp and rs1_oen and                  \
                             WB.reg_c_rf_wen and                           \
                             (Pipe.ID.rs1 == STACK) else                   \
               FWD_NONE
```

This new hazard changes the control signals for forwarding between `rs1`, `rs2`, `op2` values in EX or MM to `op1`, `op2`, `rs2` data in ID. The above code is an example for forwarding the `rs1` value to `op1` data. Since the regular write back data and ALU output differ in push and pop instructions, there is an extra signal(`FWD_MM/WB_ALU`) to forward ALU outputs. On detecting when to forward, we check whether the EX, MM, WB stage is handling a push or pop instruction, and check if the requiring data needs access of the stack.

Another case occurs only after the call of pop. Popping from the stack fetches data from the stack at the MM stage, and updates the destination register at the WB stage. Therefore if an instruction shortly after pop requires access to that destination register, forwarding is needed. If the instruction is called one step after, we should stall one stage and then forward the value from MM, while if it is called two stages after we could simply forward the value from MM.

```
EX_load_inst    = (EX.reg_c_dmem_en or EX.reg_wb_sp) and EX.reg_c_dmem_rw == M_XRD
load_use_hazard = (EX_load_inst and EX.reg_rd != 0) and        \
                  ((EX.reg_rd == Pipe.ID.rs1 and rs1_oen) or   \
                   (EX.reg_rd == Pipe.ID.rs2 and rs2_oen))
```

This hazard is controlled like a load-use hazard, so changing the control signals for the load-use hazard is sufficient, like shown above. `EX_load_inst` detects whether EX has the output data to write to the `rd` register. Therefore we add a signal showing that EX is handling a pop instruction, then forwarding the output data to ID is done easily. Stalling is also done automatically with the previous codes.

Part 2 : Branch Prediction with Branch Target Buffer (BTB)

```
class BTB(object):
   WORD_SIZE_BIT = WORD_SIZE * 8
   VALID_BIT     = 0
   TAG           = 1
   TAG_ADDRESS   = 2
   NOT_VALID     = 0
   VALID         = 1

   def __init__(self, k):
      self.k = k
      self.n = 1 << k
      self.cache = [[0, 0, 0]] * self.n

   def lookup(self, pc):
      lookup_index, lookup_tag = self._parse(pc)
      valid_bit, tag, target_address = self.cache[lookup_index]
      if valid_bit and (lookup_tag == tag) : return target_address, True
      else : return 0, False

   def add(self, pc, target):
      index, tag = self._parse(pc)
      self.cache[index] = [BTB.VALID, tag, target]

   def remove(self, pc):
      index, tag = self._parse(pc)
      self.cache[index][BTB.VALID_BIT] = BTB.NOT_VALID

   def _parse(self, pc):
      index = (pc >> 2) % self.n
      tag = pc // (self.n << 2)
      return index, tag
```

The implementation of the BTB is shown above. The buffer is stored in a cache memory which is saved as a Python list, and looking up data or adding, removing data is available.

Detecting whether a branch is taken or not is done in EX. First a control signal is generated at ID, detecting whether the branch is taken or not, i.e. the whether the branch result is true or not. Each is saved as `br_correct` and `br_wrong` and is accessed at EX. If the branch is taken, we add the branch information to the BTB and if it is not, we remove the information. Also `jal` is considered as an always correct branch. This is implemented as the following.

```
# in EX.update()
if Pipe.CTL.br_correct : Pipe.cpu.btb.add(self.pc, self.brjmp_target)
if Pipe.CTL.br_wrong   : Pipe.cpu.btb.remove(self.pc)

# in Control.gen()
self.br_correct = (EX.reg_c_br_type == BR_NE  and (not Pipe.EX.alu_out)) or   \
                  (EX.reg_c_br_type == BR_EQ  and Pipe.EX.alu_out)       or   \
                  (EX.reg_c_br_type == BR_GE  and (not Pipe.EX.alu_out)) or   \
                  (EX.reg_c_br_type == BR_GEU and (not Pipe.EX.alu_out)) or   \
                  (EX.reg_c_br_type == BR_LT  and Pipe.EX.alu_out)       or   \
                  (EX.reg_c_br_type == BR_LTU and Pipe.EX.alu_out)       or   \
                   EX.reg_c_br_type == BR_J
self.br_wrong   = (EX.reg_c_br_type == BR_NE  and Pipe.EX.alu_out)       or   \
                  (EX.reg_c_br_type == BR_EQ  and (not Pipe.EX.alu_out)) or   \
                  (EX.reg_c_br_type == BR_GE  and Pipe.EX.alu_out)       or   \
                  (EX.reg_c_br_type == BR_GEU and Pipe.EX.alu_out)       or   \
                  (EX.reg_c_br_type == BR_LT  and (not Pipe.EX.alu_out)) or   \
                  (EX.reg_c_br_type == BR_LTU and (not Pipe.EX.alu_out))
```

Detecting misprediction is done when generating control signals. We check two cases, when the branch is correct and looking up the BTB fails, or when the branch is wrong and looking up the BTB succeeds. Meanwhile, `jalr` should not be affected and should be always-not-taken, so we handle them independently, so that the BTB is not searched for those operations. One modification is made in the datapath, where the result of looking up the BTB (`btb_status`) is passed through the pipeline register from the IF stage to the EX stage. This is implemented as the following.

```
EX_brjmp = self.br_correct and not Pipe.EX.btb_status or         \
           self.br_wrong and Pipe.EX.btb_status or               \
           EX.reg_c_br_type == BR_JR and                         \
           self.pc_sel != PC_4
```

After the detection, searching the BTB and choosing the next operation is done in the IF stage. First lookup the BTB and get results, and if we succeed in searching we set the next PC value as the searched target address. However this should be decided after checking all signals from the branches or jumps. Checking whether a mispredicted branch is fixed from above, or whether a jump signal is coming from above should be done first, and then we can move the PC to the address stored in BTB. This is implemented as the following.

```
# in IF.execute()
target_address, self.btb_status = Pipe.cpu.btb.lookup(self.pc)
```

```
self.pc_next = self.ex_pcplus4         if Pipe.CTL.pc_sel == PC_PR_WRONG else \
               Pipe.EX.brjmp_target    if Pipe.CTL.pc_sel == PC_BRJMP    else \
               Pipe.EX.jump_reg_target if Pipe.CTL.pc_sel == PC_JALR     else \
               target_address          if self.btb_status                else \
               self.pcplus4            if Pipe.CTL.pc_sel == PC_4        else \
               WORD(0)

# in Control.gen()
self.pc_sel = PC_JALR     if  EX.reg_c_br_type == BR_JR                  else \
              PC_BRJMP    if self.br_correct and not Pipe.EX.btb_status  else \
              PC_PR_WRONG if self.br_wrong and Pipe.EX.btb_status        else \
              PC_4
```