Lab2 Report

Methodology

earlier branch resolving

Ports I added for implementation:

- `rv5s_eb_hazardunit.h`: the hazard unit

  The input ports are:

```
// ID.rs1 and ID.rs2 obtained from decode->r1_reg_idx and decode-
>r2_reg_idx
// Used to detect branch/jump hazard
INPUTPORT(id_reg1_idx, c_RVRegsBits);
INPUTPORT(id_reg2_idx, c_RVRegsBits);

// ID/EX.Rd, ID/EX.RegWrite, ID/EX.MemRead
INPUTPORT(ex_reg_wr_idx, c_RVRegsBits);
INPUTPORT(ex_do_mem_read_en, 1);
INPUTPORT(ex_do_reg_write, 1);

// EX/MEM.Rd, EX/MEM.MemRead, EX/MEM.RegWrite
INPUTPORT(mem_reg_wr_idx, c_RVRegsBits);
INPUTPORT(mem_do_mem_read_en, 1);
INPUTPORT(mem_do_reg_write, 1);

// MEM/WB.Rd, MEM/WB.RegWrite
INPUTPORT(wb_reg_wr_idx, c_RVRegsBits);
INPUTPORT(wb_do_reg_write, 1);

// ID/EX.opcode; decode->opcode in ID stage
INPUTPORT_ENUM(ex_opcode, RVInstr);
INPUTPORT_ENUM(id_opcode, RVInstr);
```

  The output ports are:

```
// Hazard Front End enable: Low when stalling due to a load-
use/branch/jump hazard
// Used to stall the PC reg and IF/ID regs
OUTPUTPORT(hazardFEEnable, 1);

// Hazard IDEX enable: Low when stalling due to an ECALL hazard
// Used to stall the IDEX regs
OUTPUTPORT(hazardIDEXEnable, 1);

// IDEX clear: High when a load-use/branch/jump hazard is detected
```

```
// Used to flush the ID/EX regs
OUTPUTPORT(hazardIDEXClear, 1);

// EXMEM clear: High when an ECALL hazard is detected
// Used to flush the EX/MEM regs
OUTPUTPORT(hazardEXMEMClear, 1);

// Stall Ecall Handling: High whenever we are about to handle an
ecall, but have outstanding writes in the pipeline which must be
comitted to the register file before handling the ecall.
OUTPUTPORT(stallEcallHandling, 1);

// controlflowHazard: High when there is a branch/jump hazard
OUTPUTPORT(controlflowHazard, 1);
```

The input ports are used to detect hazards.

The hazards are divided into 4 classes: LoadUseHazard, BranchHazard, JumpHazard, EcallHazard. There dectection laws are:

- LoadUseHazard:

  `!BranchOrJumpOp` && `ID/EX.MemRead` && `ID/EX.Rd`!=`x0` && (`ID/EX`==`IF/ID.rs1` || `ID/EX`==`IF/ID.rs2`)

- BranchHazard:

  `BranchOp` && {

  [`ID/EX.RegWrite` && `ID/EX.Rd`!=`x0` && (`ID/EX`==`IF/ID.rs1` || `ID/EX`==`IF/ID.rs2`)] ||

  [`EX/MEM.MemRead` && `ID/EX.Rd`!=`x0` && (`ID/EX`==`IF/ID.rs1` || `ID/EX`==`IF/ID.rs2`)] }

- JumpHazard:

  `JumpOp` && {

  [`ID/EX.RegWrite` && `ID/EX.Rd`!=`x0` && (`ID/EX`==`IF/ID.rs1`)] ||

  [`EX/MEM.MemRead` && `ID/EX.Rd`!=`x0` && (`ID/EX`==`IF/ID.rs1`)] }

EcallHazard is unchanged (same as that in `rv5s`).

The hazard stall/flush law is: (- means enable = 1, clear = 0)

	PC	IF/ID	ID/EX	EX/MEM
load-use	enable = 0	enable = 0	clear = 1	-
branch/jump	enable = 0	enable = 0	clear = 1	-
syscall	enable = 0	enable = 0	enable = 0	clear = 1

The syscall exit stall/flush law is:

	PC	IF/ID	ID/EX	EX/MEM
syscallExit	-	clear = 1	clear = 1	-

The controlflow change (without branch/jump hazard) stall/flush law is:

	PC	IF/ID	ID/EX	EX/MEM
controlflow (no br/jmp hz)	-	clear = 1	-	-

Hence the controls `enable` and `clear` of registers are determined by :

```
hazardFEEnable << [=] {return !(hasLoadUseHazard() ||
hasBranchHazard() || hasJumpHazard() || hasEcallHazard()); };

hazardIDEXEnable << [=] {return !hasEcallHazard(); };

hazardIDEXClear << [=] {return (hasLoadUseHazard() ||hasBranchHazard()
|| hasJumpHazard()); };

hazardEXMEMClear << [=] {return hasEcallHazard(); };

stallEcallHandling << [=] {return hasEcallHazard(); };

controlflowHazard << [=] {return hasBranchHazard() || hasJumpHazard();
};
```

- `rv5s_eb_forwardingunit.h`:

  The input ports are:

```
// ID/EX.rs1, ID/EX.rs2
INPUTPORT(id_reg1_idx, c_RVRegsBits);
INPUTPORT(id_reg2_idx, c_RVRegsBits);

// ID.rs1 and ID.rs2 obtained from decode->r1_reg_idx and decode-
>r2_reg_idx
INPUTPORT(br_id_reg1_idx, c_RVRegsBits);
INPUTPORT(br_id_reg2_idx, c_RVRegsBits);

// EX/MEM.Rd, EX/MEM.RegWrite, EX/MEM.MemRead
INPUTPORT(mem_reg_wr_idx, c_RVRegsBits);
INPUTPORT(mem_reg_wr_en, 1);
INPUTPORT(mem_mem_do_read, 1);

// MEM/WB.Rd, MEM/WB.RegWrite
INPUTPORT(wb_reg_wr_idx, c_RVRegsBits);
INPUTPORT(wb_reg_wr_en, 1);

// decode->opcode, ID/EX.opcode
```

```
INPUTPORT_ENUM(id_opcode, RVInstr);
INPUTPORT_ENUM(ex_opcode, RVInstr);
```

The output ports are:

```
// Forwarding source selector
OUTPUTPORT_ENUM(alu_reg1_forwarding_ctrl, ForwardingSrc);
OUTPUTPORT_ENUM(alu_reg2_forwarding_ctrl, ForwardingSrc);
OUTPUTPORT_ENUM(br_reg1_forwarding_ctrl, ForwardingSrc);
OUTPUTPORT_ENUM(br_reg2_forwarding_ctrl, ForwardingSrc);
```

There are three data sources for `pc_br` and `alu` input ports: `IdStage`, `MemStage`, `WbStage`. The forwarding law is:

- `alu_reg1_forwarding_ctrl`:
  - `MemStage`: !`BranchOrJumpOp` && `EX/MEM.RegWrite` && `EX/MEM.Rd`!=`x0` && `EX/MEM.Rd`==`ID/EX.rs1`
  - `WbStage`: !`BranchOrJumpOp` && `MEM/WB.RegWrite` && `MEM/WB.Rd`!=`x0` && `MEM/WB.Rd`==`ID/EX.rs1`
  - `IdStage`: Otherwise
- `alu_reg2_forwarding_ctrl`:
  - `MemStage`: !`BranchOrJumpOp` && `EX/MEM.RegWrite` && `EX/MEM.Rd`!=`x0` && `EX/MEM.Rd`==`ID/EX.rs2`
  - `WbStage`: !`BranchOrJumpOp` && `MEM/WB.RegWrite` && `MEM/WB.Rd`!=`x0` && `MEM/WB.Rd`==`ID/EX.rs2`
  - `IdStage`: Otherwise
- `br_reg1_forwarding_ctrl`:
  - `MemStage`: `BranchOrJumpOp` && `EX/MEM.RegWrite` && !`EX/MEM.MemRead` && `EX/MEM.Rd`!=`x0` && `EX/MEM.Rd`==`decode->rs1`
  - `WbStage`: !`BranchOrJumpOp` && `MEM/WB.RegWrite` && `MEM/WB.Rd`!=`x0` && `MEM/WB.Rd`==`decode->rs1s`
  - `IdStage`: Otherwise
- `br_reg2_forwarding_ctrl`:
  - `MemStage`: `BranchOp` &&4 `EX/MEM.RegWrite` && !`EX/MEM.MemRead` && `EX/MEM.Rd`!=`x0` && `EX/MEM.Rd`==`decode->rs2`
  - `WbStage`: `BranchOp` && `MEM/WB.RegWrite` && `MEM/WB.Rd`!=`x0` && `MEM/WB.Rd`==`decode->rs2`
  - `IdStage`: Otherwise

- `rv5s_eb.h`

The branch destination computing component is moving to `ID` stage, using `pc_br`. Note that we do not remove `alu_reg1_forwarding_ctrl` for `auipc` instruction. Except for hazard unit, forwarding unit and branch destination computing unit, the other parts are similar to `rv5s` implementation.

branch prediction

- branch predictor `rv_branch_predictor.h`

  Attach a 2-bit saturate counter to each branch instruction address (as introduced in the lecture). Use a BHT to matain these counters. Since in this lab the BHT size is not constrained, I used a brute-force method to directly construct a $2^{16}\times 2^{16}$ 2-D array in my first try:

  ```
  int BHT[1<<16][1<<16]
  ```

  The first dimension indices the higher 16 bits of an instruction address, and the second indices the lower bits. The entries store the counters.Instead of recording counters for only branch instructions, I covered the whole address space. This resulted in slow initialization and reset in Ripes simulator (since every time it should set all 2^{32} entries 0).

- branch target buffer `rv_branch_target_buffer.h`

  I also used a $2^{16}\times 2^{16}$ 2-D array as the BTB (still, my first try):

  ```
  VSRTL_VT_U BTB[1<<16][1<<16];
  ```

  The entries store the target address of the branch instructions.

- branch prediction policy

  Every time the branch predictor reads the instruction address `branch_address`. If the instruction **is** a branch/jump instruction, it looks up the corresponding saturate counter in `BHT[branch_address[31:16]][branch_address[15:0]]`. The branch predictor predicts taken iff the counter is `0b10` or `0b11` and the instruction is a branch/jump instruction. Otherwise the counter predicts not taken.

  At the same time, the `target_address` will returned from `BTB[branch_address[31:16]][branch_address[15:0]]`.

  Then, it is `pc_src_if`'s responsibility to choose the predicted instruction address between `pc+4` to `target_address`. It will choose the latter iff the branch predictor returns taken. At this moment we get the predicted instruction address.

  Finally, `pc_src` need to choose between `PREDICTED` and `ACTUAL` adresses. This is controlled by the `branch` component that computes the actual branch behavior and check the predicted address at ID stage. `PREDICTED` addresses will be chosen iff the prediction is correct.

- BHT/BTB update policy

  - Update BHT: BHT should be updated as long as there is a branch/jump instruction. If a branch is taken, its counter will increase 1 until it reach `0b11`. If a branch is not taken, its counter will decrease one until it reach `0b00`. It is `branch` component's job to check whether the branch is taken or not.

    - Update BTB: BTB should be updated as long as there is a branch/jump instruction. The corresponding entry in BTB is the target address returned by the `branch` component.

- Prediction check policy

  The prediction is wrong iff: (1) the current instruction (at ID stage) is a branch/jump instruction (2) the predicted instruction (at IF stage) and the branch result have different PC values.

- Tradeoff:

  Directly allocating a large 2-D array may reduce conflicts between two branch addresses, but the overheads to initialize it is also high (set 2^{32} entries 0). So I choose to use a 2^{16}-entry 1-D array for BTB and BHT finally. The only change I need to consider is the mapping rule: branch instruction at address `addr[31:0]` is mapped to the entry at index `addr[15:0]`. It turns out that the number of cycles it takes is still under the bar.

Results

Part 3: Earlier branch resolving

- Number of cycles

	rv5s_eb	rv5s
test_branch_0.S	102	101
test_branch_1.S	108	110
test_branch_2.S	933	1095
test_branch_3.S	172	189
performance_1_matrix.S	295507	300947
performance_2_ranpi.S	345422	362146

  Analysis:

  For early branch resolving, it reduces the number of nullifications when there are controlflow changes, and that's why `rv5s_eb` has fewer cycles than `rv5s`; it also increases the number of stalls if branch/jump need to read after a preceding register write instruction, and considering the fact that a branch/jump instruction always follows immediately after writing to the register it need, that explains the relatively small gap between `rv5s_eb` and `rv5_s`'s cycle number.

- Average CPI

	rv5s_eb	rv5s
test_branch_0.S	1.79	1.77
test_branch_1.S	1.38	1.41
test_branch_2.S	1.76	2.06

	rv5s_eb	rv5s
test_branch_3.S	1.38	1.51
performance_1_matrix.S	1.26	1.29
performance_2_ranpi.S	1.2	1.26

Analysis:

CPI is determined by %branch/jump, branch_misprediction_penalty, and %stall. In these two cases, it is always predicted that the next instruction is pc+4 (i.e., branch always not taken), hence the branch_misprediction_penalty for `rv5s_eb` and `rv5s` is 1 and 2 cycles, resp. However, %stall of `rv5s_eb` is higher than that of `rv5s`, and %branch/jump is relatively low, so 0.5CPI(rv5s) < CPI(rv5s_eb) < CPI(rv5s)

Part 4: Branch prediction

- Statistics:
  - `rv5s_hz` after implementing branch prediction:

	CPI	#cyles	#retired instr
performance_1_matrix.S	2.05	740128	360729
performance_2_ranpi.S	1.61	718895	446381

  - `rv5s_hz` before implementing branch prediction (the original one, always predict not taken)

	CPI	#cyles	#retired instr
performance_1_matrix.S	2.11	749015	355609
performance_2_ranpi.S	1.72	741673	432350

  - Analysis:

    Comparing the above two cases, we can see that the number of cycles reduced by 8887 and 22778 for two programs, resp, after implementing branch prediction. We can approximately assume that a branch misprediction cause one cyle stall (actually, if a branch misprediction is avoided, it may still be stalled for data hazard later for even more cycles). Hence the improvement by a more sophisicated branch prediction is quite considerable.

Part 5: BPU attack

- Codes
  - `attacker.S`

```
.text
main:
```

```
    addi    sp, sp, -32
    addi    s0, s0, 2
    nop
check:
    blt     s0, zero, exit
    bne     zero, zero, exit  # prime / probe, 0x10
    addi    s0, s0, -1
    j       check
exit:
    addi    sp, sp, 32
```

- `victim.S`

```
.text
main:
    addi    sp, sp, -32
    addi    t0, t0, 0       # secret data t0: 0 or 1
    addi    t1, t1, 2
check:
    blt     t1, zero, exit
    beq     t0, zero, loop  # victim, 0x10
    j       exit
loop:
    addi    t1, t1, -1
    j       check
exit:
    addi    sp, sp, 32
```

- Organize execution

1. Revise the code in `rv_branch_predictor.h` and `rv_branch_target_buffer`: remove `reset()` function to prevent the BHT and the BTB from reseting every time we load a program.
2. Reload the processor in Ripes to initialize BHT, BTB.
3. Set `t0` to be 0 in `victim. S`. Load and run `attacker.S`, `victim.S`, `attacker.S` in order. Do not click the reset button after loading a program.
4. Reload the processor in Ripes to initialize BHT, BTB.
5. Set `t0` to be 1 in `victim. S`. Load and run `attacker.S`, `victim.S`, `attacker.S` in order. Do not click the reset button after loading a program.

- Results

`t0`	attcker primes (#cyles)	victim runs (#cyles)	attacker probes (#cyles)
0	30	33	32
1	30	15	30

Analysis:

I chose to attack the branch predictor unit.

Initially, the BHT and the BTB entries were all set to 0 when creating the processor. Then, when `attacker.S` was loaded and run for the first time for priming, the `bne zero, zero, exit` at address `0x10` was executed for 3 times, and the branch was never taken. Hence, after priming, the state of the saturate counter of address `0x10` was SN (strongly not taken, `0b00`).

Then, `victim.S` was loaded and run. Since `reset()` was not set, the states of the BTB and BHT didn't change. There were two cases:

- If `t0` was set 0 in `victim.S`: branch `beq t0, zero, loop` at address `0x10` was executed for 3 times, and the branch was always taken. Hence the state of the saturate counter of address `0x10` was updated to ST (strongly taken, `0b11`).
- If `t0` was set 1 in `victim.S`: branch `beq t0, zero, loop` at address `0x10` was never executed. Hence the state of the saturate counter of address `0x10` was unchanged (still SN).

Also notice that the state of the staturate counters at the other branch instructions'addresses in `attacker.S` were not affected by `victime. S`, hence would not affect the cylce number when re-executing `attacker.S`.

Finally, `attacker.S` was reloaded and run. The `bne zero, zero, exit` at address `0x10` was executed for 3 times, and the branch was always actually not taken. There were two cases:

- If `t0` was set 0 in `victim.S`: The state of the saturate counter at address `0x10` changed from ST to WT to WN. So the branch predictor state was MISS, MISS, HIT. This resulted in 2 more cycles to flush the mispredicted instruction. Hence the `attacker.S` took 2 more cycles to run at the second time.
- If `t0` was set 1 in `victim.S`: The state of the saturate counter at address `0x10` was SN, same as before. Hence the state of the saturate counter of address `0x10` was unchanged (still SN), and the number of cyles remained the same as before.

Thus, by comparing the number of cycles of the two execution of `attacker.S`, we could know the secret bit of `victim.S`.

Question Answering

1. *How are stalls implemented in Ripes ?*

   Answer. Use `enable` in pc and stage registers to prevent them from being updated: if `enable` is set 0, then the registers won't be updated, thus the current instruction will be re-executed at the next cycle at the same stage. Use `clear` in stage registers to add a `nop`: if `clear` is set 0, the stage registers will all be 0, which looks as if the current instruction at this stage is `nop`. By combining the two, stalls can be implemented.

2. *How should we choose the source of PC ? When can the value from the prediction modules be used ? When should the PC be recovered by the branch module ?*

   Answer.

   - Source of PC:

instruction	PC src
`branch`	PC + offset, PC + 4
`jump`	PC + offset, reg + offset

- The values from the prediction modules can be used when: (1) `pc_src` need the next instruction address (values from `branch_target_buffer` or `branch`) (2) `branch` component need to check the correctness of the prediction (values from `branch_target_buffer` or `branch_predictor`)
- The PC should be recovered by the branch module when: (1) ID.instr (the instruction at ID stage) is a branch/jump instruction, and (2) IF.instr's address is different from the result of `pc_src_id` (the actual target address)

3. *How are mispredicted branches handled ?*

   Answer. The `branch` component detects the misprediction by `wrongPredictPc()`. If the prediction is wrong, the output signal from port `wrong_predict_pc` will be sent to the OR gate `wrong_pc_hazard_or`, and this gate will set IF/EX.clear to 0. At that time, the wrongly predicted instruction (currently at IF stage) is flushed. Meanwhile, the right instruction PC will chosen by `pc_src`. So in the next cycle, the right instruction will be executed, while the wrong instruction will have been replaced by a `nop`.

4. *What are the pros and cons of resolving branches earlier at the ID stage ?*

   Answer.

   pros: Reduce instruction nullification caused by controlflow change: if compute branch/jump at EX stage, if the control flow changes, instructions at IF and ID stage should be flushed, while if compute branch/jump at ID stage, only instructions at IF stage should be flushed. (Here we assume always not taking branch.)

   cons: More possible hazards and stalls, since `rs1` and `rs2`'s values are needed at ID stage instead of EX stage. Also, the clock cycle may be longer for longer critical path.

5. *How to defend against branch-prediction-based attack ?*

   Answer.

   - Software: algorithmically eliminate the dependicies of branches on secret data; reduce conditional branch by compiler
   - Hardware: periodically randomize the mapping from instruction addresses to BHT indices; let software developer tell the hardware to remove the prediction for sensitive branches; partition the branch prediction unit so that the victims and the attackers do not use the same BPU; add noise to performance counters, etc.

   I referred to `asplos18.pdf` provided in `lab2` folder.

6. *Can we implement an out-of-order processor using this design ? If we can, how should the lab framework be modified ? If not, what are the major reasons ?*

Answer. We can add some stages based on the framework: renaming stage to resolve depencies, issue stage to schedule instruction executions, and retire stage to commit results. Also we need several copies of each stage modules to support OoO execution. Ealier branch resolving and branch prediction can be remained so that we can fetch more instructions at IF stage, which increase parallelism.