# CA2-lab4-report

Author: Yunxin Yang

studentID: 2023233204

## Implementation

### Data structure

#### ReOrder buffer entry

ROB entry design:

| Ready | Busy | Instruction | Destination | Value | Address |
|---|---|---|---|---|---|
| Is the result ready to use? | Is the instruction under execution? | Instruction type | The destination of the instruction (e.g. register index or ROB entry index) | The result value | Holds information for the memory address calculation |

```
struct ReorderBufferEntry {
  bool ready;
  bool busy;
  RISCV::Inst instType;
  int destination; // register index or ROB entry index
  int value;
  int address;
  uint32_t memLen; // for store
};
```

#### ReOrder buffer

use `kReorderBufferSize` ReorderBufferEntry to consist a Reorder buffer

```
const static int kReorderBufferSize = 32; // set ROB size
ReorderBufferEntry reorderBuffer[kReorderBufferSize];
```

#### ReOrder buffer index

use a class `ReorderBufferIndex` to simulate a queue for pushing and getting a ROB entry

push() : get a ReorderBufferIndex

top(): return the head ReorderBufferIndex

pop(): remove a ReorderBufferIndex

```
class ReorderBufferIndex {
public:
  int head;
  int tail;
  bool full;
  int push(); // -1 : full; > 0 new buffer index; if push, must be used
  int top(); // -1 : blank; > 0 head index
  void pop();
  ReorderBufferIndex();
};

int ReorderBufferIndex::push(){
  if (full)
    return NONE;
  int ret = tail;
  tail = (tail+1)%kReorderBufferSize;
  if (tail == head)
    full = true;
  return ret;
}

int ReorderBufferIndex::top(){
  if (!full && tail == head) // blank
    return -1;
  return head;
}

void ReorderBufferIndex::pop(){
  if (!full && tail == head) {
    // blank
    fprintf(stderr, "ROB blank but still pop\n");
    exit(-1);
  }
  head = (head+1)%kReorderBufferSize;
  full = false;
}
```

## Register Status Data Structure

| Field | Meaning | x1 | x2 | ... | x32 |
|---|---|---|---|---|---|
| Reorder | The ROB entry index | | | | |
| Busy | Whether the register is busy | | | | |

```
struct RegisterStatus {
    int reorderEntryIndex;
    bool busy;
  } registerStatus[RISCV::REGNUM];
```

## Reservation Station Structure

| Busy | Op | Vj | Vk | Qj | Qk | Dest | A |
|---|---|---|---|---|---|---|---|
| Whether the FU is busy | Instruction type | Value of one source operand | Value of the other source operand | The index of the ROB entry produces `Vj` | The index of the ROB entry produces `Vk` | The destination of the instruction (e.g. register index or ROB entry index) | Holds information for the memory address calculation |

additional tag and variable:

- start：indicates starting execute
- addressDone: indicates load step 1 done
- load2Start: indicates load step 2 can start

- exeLeftTime: execution left cycle to complete
- result：temporarily to store the result
- pc: store the RS's pc for B/I type instruction
- memLen, readSignExt : parameters that load needs

using a 2-D array (1st: the function unit type; 2nd: how many function unit for each type? each function unit has one reservation station) to store the reservation stations:

```
const static int kFUNumberPerComponent = 5;
  struct ReservationStationEntry {
    // int functionUnit; // TODO
    bool busy;
    RISCV::Inst op; // Instruction type
    int64_t Vj; // value
    int64_t Vk;
    int Qj; // index
    int Qk;
    int destination; // rob_index
    int64_t address;

    bool start;
    bool addressDone;// for load
    bool load2Start;// for load
    int exeLeftTime;
    uint64_t pc; // for branch
    int64_t result;
    uint32_t memLen; // for load
    bool readSignExt; // for load
  } reservationStations[number_of_component][kFUNumberPerComponent];
```

## algorithm design

### simulate()

- initialize ROB, registerStatus, RS first
- make the 4 stage: issue, execute, writeback, commit

### Issue()

1. fetch the instruction according to pc first, simulating the step that get instruction from instruction queue
2. Judge if currently under control hazard(B/J type), if so, make pc = pc -4 and stall issue
3. decode the instruction
4. Judge if there is empty entry in corresponding RS(reservation station) and ROB, if no then do nothing but wait for next cycle
5. IF so, judge if the instruction is B/J type for control hazard, if so, mark the pipeline is under control hazard
6. mark the issued instruction in corresponding rs
    a. finding if the source register is available
        i. available, get the value from registerFile directly
        ii. not available but depend on other ROB entry, than mark the Qj/k to track the depended ROB entry
    b. mark rs busy, and rs's corresponding ROB

7. mark the corresponding ROB entry, destination register and instruction type etc.
8. update the dest register's newest ROB to for following instruction to track the RAW dependency for the newest.
9. record other variables such as the immediate number for load/store, branch, jump for use.

### Execute()

1. Traverse all RS entry
2. If the RS entry flag start = 0, busy = 1, indicating it is issued but need to handle the RAW hazard to really dive into execution
    a. judge if there is data hazard According to RS.Qj, Qk , which indicates if it has dependency on other ROB. If RS.Qj, Qk = -1, indicating the source value is ready and can start Execution, otherwise do nothing but wait for next cycle.
    b. do the operation execution
    c. judge if it is the B/J instruction, if so, remove the mark that it is undergoing control hazard and update the fetch pc
    d. save the result in RS.result (normal instruction), RS.address(for later load execution), ROB.address(for later store in commit stage)
        i. ps: I use a different strategy from the guide book for store since I think it needn't to wait for reaching the head of queue and it can send the calculated address to its corresponding ROB entry and affects nothing.
    e. although in simulation it completes the execution, it still needs to simulate the FU latency, so set RS.exeLeftTime to simulate the FU latency for every cycle to reduce a latency cycle(detailed in 3.).
3. If the RS entry flag start = 1, indicating the instruction execution starts
    a. for normal instructions(other than load), just reduce the exeLeftTime to simulate the execution latency (result calculated before but need to simulate the FU latency)
    b. for load instruction:
        i. if load step 1 done, mark it can start load step 2 judge
        ii. if can start load step2 judge, traverse the ROB entry before the load's corresponding, if there is store ROB entry not ready(address is not calculated yet) or store address is conflicted with the load address(to avoid Memory hazard), mark it can not start load step2, otherwise mark it can start load step2, and do the read memory operation; send the result to corresponding ROB
        iii. if load step2 start, just reduce the exeLeftTime to simulate the load latency

### Writeback()

1. for simplicity, I enable multiple writeback in one cycle to ROB and RS's pending source register value
2. Traverse every reservation station entry, if the RS.start = 1 (indicating execution start) and RS.exeLeftTime = 0 (indicating execution done), it can writeback
3. for normal instruction(other than store), fill the RS.result to corresponding ROB, and other RS if source register depends on this ROB.
4. For store instruction, wait for RS.Qk ready, if not ready do nothing but wait in next cycle; If ready send the Vk to corresponding ROB

### Commit()

1. I design to commit at most one instruction one cycle
2. Judge if commitStallTime > 0(means the in-order-commit needs to wait for store latency)
3. get the top ROB entry to see if it is ready (result calculation completed)
4. if ready and instruction is store, store the result to memory, and record the store latency to commitStallTime to simulate the latency
5. IF normal instruction, store the result to regFile

6. pop ROB entry
7. mark RegisterStatus busy = 0 if its corresponding newest ROB entry is this ROB entry

# Discussion

- How to solve WAW and WAR: since the write result first written to local ROB, and at last in order commit, hencing solving WAW, WAR
- Since no middle register designed for register transfer between stages and all depends on the global ROB, RS, RegisterStatus structure, I add 3 execution clock latency to simulate the stage transferring latency.

# Evaluation and analysis

I compare this ROB version to previous: Original simulator, pure stall simulator(I made in Lab1), scoreboard simulator(lab1)

## Normal case comparison

takes code below to run:

```
int main() {
        print_s("Hello, World!\n");
    exit_proc();
}
```

### original simulator

```
------------ STATISTICS -----------
Number of Instructions: 149
Number of Cycles: 269
Avg Cycles per Instrcution: 1.8054
Branch Perdiction Accuacy: 0.5000 (Strategy: Always Not Taken)
Number of Control Hazards: 28
Number of Data Hazards: 66
Number of Memory Hazards: 2
-----------------------------------
```

### pure stall simulator

```
------------ STATISTICS -----------
Number of Instructions: 149
Number of Cycles: 391
Avg Cycles per Instrcution: 2.6242
Branch Perdiction Accuacy: -nan (Strategy: Always Not Taken)
Number of Control Hazards: 28
Number of Data Hazards: 111
Number of Memory Hazards: 2
-----------------------------------
```

### Scoreboard with adequate(10 in example) component each

```
------------ STATISTICS -----------
Number of Instructions: 149
Number of Cycles: 367
Avg Cycles per Instrcution: 2.4631
Branch Perdiction Accuacy: -nan (Strategy: Always Not Taken)
Number of Control Hazards: 0
Number of Data Hazards: 77
Number of Memory Hazards: 0
Number of Structure Hazards: 0
Number of WAW Hazards: 28
Number of WAR Hazards: 6
Number of control flush counts: 125
-----------------------------------
```

### Reorder-buffer-version with 5 FU components for each FU type

```
------------ STATISTICS -----------
Number of Instructions: 152
Number of Cycles: 242
Avg Cycles per Instrcution: 1.5921
Branch Perdiction Accuacy: -nan (Strategy: Always Not Taken)
Number of Control Hazards: 164
Number of Data Hazards: 62
Number of Memory Hazards: 0
Number of Structure Hazards: 1
-----------------------------------
```

### analysis

From `Avg Cycles per Instrcution`, we can see that in this case, performance:

Reorder-buffer-version with 5 FU components for each FU type(1.5921) > original simulator(1.8054) > Scoreboard with adequate(10 in example) component each (2.4631) > pure stall simulator (2.6242)

We can see that although without branch prediction,  Reorder version outperforms most since it uses adequate FU to eliminate structure hazard and it supports out-or-order issue and it eliminate the WAW and WAR hazard, which enable more instructions to execute at the same time.

Both without data bypass, Scoreboard with adequate(10 in example) component each (2.4631) > pure stall simulator (2.6242) since Scoreboard with adequate(10 in example) component each still has some library code that can run out of order to out perform.

From control hazard, we can see that Reorder version has the most since it accelerate the process but the control hazard just stall the issue stage and do not affect the out-of-order and out-of-order execution in later stage, so the Reorder version is not much affected.

The reorder version also has least Data Hazards count since it has the fastest process.

## special case for Reorder buffer version to outperform

When there is a lot of complicated instruction and instructions almost independent with each other, the Reorder-buffer-version with adequate component can perform very well. The real-life application is matrix multiple and so on. I will use code(consecutively complicated instruction: mul) below to show the scoreboard potential.

```
int main()
{
    asm volatile(
        "addi t0, x0, 11;"
        "addi t1, x0, 11;"
        "addi t2, x0, 11;"
        "addi t3, x0, 11;"
        "addi t4, x0, 11;"
        "addi t5, x0, 11;"
        "addi t6, x0, 11;"
                "mul a0, t0, t0;"
                "mul a1, t1, t1;"
                "mul a2, t2, t2;"
                "mul a3, t3, t3;"
                // omit 100 lines...
                "mul a1, t1, t1;"
                "mul a2, t2, t2;"
                "mul a3, t3, t3;"
                "mul a4, t4, t4;"
                "mul a5, t5, t5;"
                "mul a6, t6, t6;"
            );
            exit_proc();
}
```

### original simulator

```
------------ STATISTICS -----------
Number of Instructions: 269
Number of Cycles: 644
Avg Cycles per Instrcution: 2.3941
Branch Perdiction Accuacy: 0.4706 (Strategy: Always Not Taken)
Number of Control Hazards: 24
Number of Data Hazards: 189
Number of Memory Hazards: 2
```

### pure stall simulator

```
------------ STATISTICS -----------
Number of Instructions: 269
Number of Cycles: 750
Avg Cycles per Instrcution: 2.7881
Branch Perdiction Accuacy: -nan (Strategy: Always Not Taken)
Number of Control Hazards: 24
Number of Data Hazards: 95
Number of Memory Hazards: 2
-----------------------------------
```

### Scoreboard with adequate(10 in example) component each

```
------------ STATISTICS -----------
Number of Instructions: 269
Number of Cycles: 463
Avg Cycles per Instrcution: 1.7212
Branch Perdiction Accuacy: -nan (Strategy: Always Not Taken)
Number of Control Hazards: 0
Number of Data Hazards: 66
Number of Memory Hazards: 0
Number of Structure Hazards: 0
Number of WAW Hazards: 30
Number of WAR Hazards: 6
Number of control flush counts: 108
-----------------------------------
```

### Reorder-buffer-version with 5 FU components for each FU type

```
------------ STATISTICS -----------
Number of Instructions: 271
Number of Cycles: 383
Avg Cycles per Instrcution: 1.4133
Branch Perdiction Accuacy: -nan (Strategy: Always Not Taken)
Number of Control Hazards: 149
Number of Data Hazards: 53
Number of Memory Hazards: 0
Number of Structure Hazards: 27
-----------------------------------
```

From `Avg Cycles per Instrcution`, we can see that in this case, performance:

Reorder-buffer-version with 5 FU components for each FU type(1.4133) > Scoreboard with adequate(10 in example) component each(1.7212) > original simulator(2.3941)> pure stall simulator (2.7881)

Reorder-buffer-version with 5 FU components for each FU type(1.4133) and Scoreboard with adequate(10 in example) component each(1.7212) outperforms original simulator and  pure stall simulator as expected because there are a lot of time-comsuming instructions(mul) which will make original simulator and pure stall simulator  stall a lot due to latency of MUL. However Reorder-buffer-version and Scoreboard will fully utilize their multiple FU to allow concurrent MUL operations.

Reorder-buffer-version still outperforms even the Scoreboard has more FU than Reorder-buffer-version (10>5 each type) most since it uses adequate FU to eliminate structure hazard and it supports out-or-order issue and it eliminate the WAW and WAR hazard, which enable more instructions to execute at the same time.

Reorder-buffer-version still gains least Data hazard. Reorder-buffer-version ‘s control hazard count is more than others but don’t affect much with the same reason given in last section.

## Answer Questions

- Is this implementation optimal?
  - NO, it didn’t consider how to deal with when multiple execution done and how to choose instruction to writeback.
- How can you further optimize this design?
  - make more complex consideration in how to deal with when multiple execution done and how to choose instruction to writeback.