CA2-lab1-Report

name: Yunxin Yang （杨昀昕）
2023233204
date:2023.11.18

Firstly, I implement a pipeline with no forwarding/passby logic but with branch prediction of not taken version for comparison and set up for scoreboard.

Pure stall version with not taken branch prediction implemention

difference with original simulator:

Data hazard stall implementation

1. Remove all the bypass logic
2. For exe-decode hazard: when detect hazard, make decode stage stall, and instruction next to execute bubble and make pc = pc-4 to stall the fetch stage. Meanwhile, tag to remind memory and writeback stage not to detect data hazard with decode stage anymore.
3. For memory-decode hazard: when detect hazard and no hazard in previous stage, make decode stage stall, and instruction next to execute bubble and make pc = pc-4 to stall the fetch stage. Meanwhile, tag to remind writeback stage not to detect data hazard with decode stage anymore.
4. For writeback-decode hazard: when detect hazard and no hazard in previous stage, make decode stage stall, and instruction next to execute bubble and make pc = pc-4 to stall the fetch stage.

Control hazard implementation

1. remove original branch prediction logic
2. In exe stage, when detected JUMP or BRANCH instruction(need to jump), make instruction of fetch and decode stage bubble to flush the pipeline . Meanwhile ,remind memory and writeback stage not to detect data hazard with decode stage anymore.

CDC scoreboard implementation

Inherit the simulator from the pure stall simulator first and Modify this version to scoreboard pipeline.

Scoreboard Table design

this design alike the design below

Instruction status

Instruction	j	k	Issue	Read operand	Executi complet	Write Result
LD F6	34+	R2	1	2	3	4
LD F2	45+	R3	5	6	7	8
MULT F0	F2	F4	6	9	19	20
SUBD F8	F6	F2	7	9	11	12
DIVD F10	F0	F6	8	21	61	62
ADDD F6	F8	F2	13	14	16	22

Functional unit status

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for j Qj	FU for k Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
0	Divide	No								

Register result status

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
62	FU								

FU status

1. types of component resumes as below

```
enum executeComponent {
    ALU,
    memCalc,
    dataMem,
    branchALU,
    iMul,
    iDiv,
    int2FP,
    fp2Int,
    fpDiv,
    fmaAdd, /* fused multiply-add unit for fp */
    fmaMul, /* fused multiply-add unit for fp */
    number_of_component_type,
    unknown = ALU,
  };
```

2. to implement multiple components of all type and to simply the implementation, I set the number of all different component the same and use a 2-dimension array(1st-dimension:type of component, 2nd-dimension: number of component) to store the flags of different FU components. For every FU component, it has a unique FUNumber(`type_number*number_component_per_type+the number of component`) to identify itself and for better record in scoreboard table(such as Qj).

   code to show the implement all the tag.

   ps:

   1. eRegFU is used to save execution result of completion to better support concurrent execution stage
   2. exeLeftTime: used to simulate the left time to complete of different FU component.

```
static const int number_component_per_type = 10;
bool busyFU[number_of_component_type][number_component_per_type];   // default: 0
int OpFU[number_of_component_type][number_component_per_type];          // default: -1, means none
int FiFU[number_of_component_type][number_component_per_type];      // default: -1, means none
```

```
int FjFU[number_of_component_type][number_component_per_type];      // default: -1, means none
int FkFU[number_of_component_type][number_component_per_type];      // default: -1, means none
int QjFU[number_of_component_type][number_component_per_type];      // default: -1, means none
int QkFU[number_of_component_type][number_component_per_type];      // default: -1, means none
bool RjFU[number_of_component_type][number_component_per_type];     // default: 0
bool RkFU[number_of_component_type][number_component_per_type];     // default: 0
EReg * eRegFU[number_of_component_type][number_component_per_type]; // default: nullptr; used to save execution result of completion
const u_int32_t EXEMAXLEFTTIME = INT32_MAX;
uint32_t exeLeftTime[number_of_component_type][number_component_per_type]; // default: 0; used to save execution left time
```

Register result status

```
int regFU[RISCV::REGNUM];
```

Scoreboard pipeline implementation

Stages Difference with original

1. split decode stage into issue stage and read Operand stage
2. merge memory stage into execute stage

New Stages actions

main logic as below

Detailed Scoreboard Pipeline Control

Instruction status	Wait until	Bookkeeping
Issue	Not busy (FU) and not result(D)	Busy(FU)← yes; Op(FU)← op; Fi(FU)← `D'; Fj(FU)← `S1'; Fk(FU)← `S2'; Qj← Result('S1'); Qk← Result(`S2'); Rj← not Qj; Rk← not Qk; Result('D')← FU;
Read operands	Rj and Rk	Rj← No; Rk← No
Execution complete	Functional unit done	
Write result	∀f((Fj(f)≠Fi(FU) or Rj(f)=No) & (Fk(f) ≠Fi(FU) or Rk(f)=No))	∀f(if Qj(f)=FU then Rj(f)← Yes); ∀f(if Qk(f)=FU then Rj(f)← Yes); Result(Fi(FU))← 0; Busy(FU)← No

1. issue stage:
    a. preserve the decode action of original decode but without read operand confirmation.
    b. add Structure hazard and WAW detection: to detect if there is a free FU(Function Unit) to use and if the reg to write is not being written, if yes, bookmark the effect of issue; If no, record the hazard and stall this issue and make pc = pc - 4 to stall fetch stage.
    c. **add Control Hazard Flush action without branch prediction:** when out of order execute, branch prediction is not avail because the instruction behind this instruction may be written previous this control instruction. Therefore, when decode a Branch or Jump prediction, just flush the pipieline of previous stages.
2. read Operand stage:
    a. check if can read Operand to read Operand.
    b. If can not read, incurs a RAW detection.
        i. needs to stall read Operand stage and make next cycle execution bubble
        ii. needs to stall issue stage
            1. if issue stage successfully run recover the issue stage's effect including the scoreboard table
            2. if issue stage detected structure hazard or WAW hazard to stall, needs to avoid repeat pc -=4, and reduce the hazard record.
    c. if this is a control flush instruction(branch or jump), flush the pipeline like what issue stage do the control flush.
3. execute stage:
    a. traverse all the busy FU to reduce the left time of themselves to simulate the latency of the FU.
    b. if this is a control flush instruction(branch or jump), flush the pipeline like what issue stage do the control flush.
    c. move original memory access logic to this stage
    d. use exeLeftTime array to record the left time of FU to simulate the latency of the FU.
    e. Because the completion of execute is out of Order, I simply design that every FU has a dynamic allocated eRegs to store the writeback registers.
4. Writeback stage:
    a. Because the completion of execute is out of Order, I simply design that it traverse all the busy FU to check if completes the execution and do the writeback logic.
    b. when writeback, check if there is a WAR hazard, if yes, delay a cycle to check
    c. If no WAR hazard, writeback the register and free the dynamic allocated eRegs.
5. simulate() function
    a. adjust it to do the new scoreboard pipeline.

Evaluation and analysis

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 just one compenent each

```
------------ STATISTICS -----------
Number of Instructions: 149
Number of Cycles: 492
Avg Cycles per Instrcution: 3.3020
Branch Perdiction Accuacy: -nan (Strategy: Always Not Taken)
Number of Control Hazards: 0
Number of Data Hazards: 43
Number of Memory Hazards: 0
Number of Structure Hazards: 177
Number of WAW Hazards: 12
Number of WAR Hazards: 0
Number of control flush counts: 113
-----------------------------------
```

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
-----------------------------------
```

analysis

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

original simulator(1.8054) > Scoreboard with adequate(10 in example) component each (2.4631) > pure stall simulator (2.6242) > Scoreboard with just one compenent each(3.3020)

Scoreboard with adequate(10 in example) component each almost elminate the sturcture hazard(in this test case , structure hazard is 0).

Original simulator outperforms most because it has branch prediction and data bypass most importantly, which performs well when limited instruction needs long cycles to run like in this test case(all simple instruction). Scoreboard can not show its better performance in simple test since it has no data bypass and branch prediction and detected structure hazard more when component is limited like Scoreboard with just one compenent each.

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. However, Scoreboard with just one compenent each performs worst since it detected structure hazard too early at issue time which will generates a lot of structure hazard since the FU is too limited, hence causing it stalls too many times.

special case for out-of-order scoreboard to outperform

When there is a lot of complicated instruction and instructions almost independent with each other, the scoreboard 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 just one compenent each

```
------------ STATISTICS -----------
Number of Instructions: 269
Number of Cycles: 1254
Avg Cycles per Instrcution: 4.6617
Branch Perdiction Accuacy: -nan (Strategy: Always Not Taken)
Number of Control Hazards: 0
Number of Data Hazards: 41
Number of Memory Hazards: 0
Number of Structure Hazards: 836
Number of WAW Hazards: 12
Number of WAR Hazards: 0
Number of control flush counts: 98
----------------------------------
```

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
----------------------------------
```

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

Scoreboard with adequate(10 in example) component each(1.7212) > original simulator(2.3941)> pure stall simulator (2.7881) > Scoreboard with just one compenent each(4.6617)

Scoreboard with adequate(10 in example) component each(1.7212) outperforms most 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. Scoreboard with just one compenent each performs worst since the heavy structure hazard(836 times) and long latency of MUL

PS

1. I only implement the scoreboard with Integer Instruction for simplicity but enough to show the effect of scoreboard with specified testcase.