# Advanced Computer Architecture: Lab 0 Report

Due on September 23rd at 11:59am

Name: **Zhenghong Yu**
Mailbox: **yuzhh1@shanghaitech.edu.cn**
Student ID: 2020533156

# Contents

# Realize LR and SC in simulator

## Requirement of LR and SC

### Premise

The content below based on the **"A" Standard Extention for Atomic Instructions, Version 2.1** which is the newest version on the wiki.riscv.org.

### Details

· LR and SC are both 32-bits instructions with opcode $0x2F$, funct5 (which means the last 5 bits number of the instruction) is $0x3$ for 8 bytes data and $0x2$ for 4 bytes data. Additionally, we dont need to consider aligned data, number limitation between LR and SC.

· LR loads a data from the address in $rs1$, places the **sign-extended** value in $rd$, and registers **a reservation set**—a set of bytes that subsumes the bytes in the addressed word.

· SC conditionally writes a data in $rs2$ to the address in $rs1$: the SC succeeds only if the reservation is still **valid** and the reservation set contains the bytes being written. If the SC succeeds, the instruction writes the data in $rs2$ to memory, and it writes **zero** to $rd$. If the SC fails, the instruction does **not** write to memory, and it writes a **nonzero** value to $rd$. Regardless of success or failure, executing an SC instruction invalidates **any** reservation held by this hart.

· An SC can only **pair with** the **most recent** LR in program order. An SC may succeed only if **no store** from another hart to the reservation set can be observed to have occurred between the LR and the SC, and if there is **no other** SC between the LR and itself in program order. An SC may succeed only if no write from a device other than a hart to the bytes accessed by the LR instruction can be observed to have occurred between the LR and SC. Note this LR might have had a different effective address and data size, but reserved the SC's address as part of the reservation set.

· The SC must **fail** if the address is not within the reservation set of the **most recent** LR in program order. The SC must **fail** if a store to the reservation set from another hart can be observed to occur between the LR and SC. The SC must **fail** if a write from some other device to the bytes accessed by the LR can be observed to occur between the LR and SC. An SC must **fail** if there is another SC (to any address) between the LR and the SC in program order.

· The failure code with value **1** is reserved to encode an unspecified failure. Other failure codes are reserved at this time, and portable software should only assume the failure code will be non-zero.

· To other access memory operations (**Load** or **Store**), we only concern about store instruction which access to these addresses that have been already reserved, remove these addresses from the reservation set.

## Idea

· I use **registry_addr** to record the first address, **registry_whe** to record whether the reservation set is still valid, **registry_num** to record the length of the reservation set. These is declared in the *Simulator.h*.

· I almost ues the datapath of **Load** instruction and **Store** instruction, as they have the same hazard

which can handle samely.

· Specially, despite the instruction fetch and datapath set, I handle the most important part (reserve and check) in the *classmemoryAccess*, which I think it might be more easier to be realize on hardware.

## Implementation

New variable or modification statement in *Simulator.h*

```
1   ......
2   namespace RISCV
3   {
4       ......
5       enum Inst
6       {
7           ......
8           LRW = 54,    /* add encode lr.w instruction as 54 */
9           LRD = 55,    /* add encode lr.d instruction as 55 */
10          SCW = 56,    /* add encode sc.w instruction as 56 */
11          SCD = 57,    /* add encode sc.d instruction as 57 */
12          ......
13      };
14      ......
15      /* add LR SC instruction with opcode 0X2F */
16      const int OP_CAS = 0x2F;
17      ......
18      inline bool isReadMem(Inst inst)
19      {
20          /* As LR and SC both need memory access and write back to
21             reg, so we need to handle with harzard in pipline */
22          if (...... || inst == LRW || inst == LRD
23              || inst == SCW || inst == SCD)
24          {
25              return true;
26          }
27          return false;
28      }
29  }
30
31  class Simulator
32  {
33  public:
34      ......
35  private:
36      ......
37      struct EReg
38      {
39          ......
```

```
            bool _cas;
        } eReg, eRegNew;
        ......
        uint32_t registry_addr; /* Registry hold the first address */
        uint32_t registry_num;  /* Registry hold the number */
        bool registry_whe;      /* Registry hold the valid bit */
    }
```

New variable or modification statement in *Simulator.cpp*

```
    ......
    namespace RISCV
    {
        ......
        const char *INSTNAME[]{ /* add instruction name */
            ......,"lrw","lrd","scw","scd"};
    }
    ......
    void Simulator::decode()
    {
        ......
        if (this->fReg.len == 4) // 32 bit instruction
        {
            ......
            switch (opcode)
            {
                ......
                case OP_CAS: /* add OP_CAS handler */
                {
                    /* read data in reg[rs1] which store the address */
                    op1 = this->reg[rs1];
                    /* read data in reg[rs2] which is no mean in LR,
                        store the old_value in SC */
                    op2 = this->reg[rs1];
                    reg1 = rs1;
                    reg2 = rs2;
                    dest = rd;  /* The reg which should be write back */

                    /* Get the funct5 */
                    uint32_t  temp = (inst >> 27) & 0x1F;
                    switch (temp)
                    {
                    case 0x2:   /* LR instruction */
                        if (funct3 == 3)    /* LR.D instruction */
                        {
                            instname = "lrd";
                            insttype = LRD;
                        }
                        else if (funct3 == 2)   /* LR.W instruction */
                        {
```

```
41                              instname = "lrw";
42                              instyype = LRW;
43                          }
44                          else
45                          {
46                              this->panic("Unknown 32bit funct3
47                                  0x%x\n", funct3);
48                          }
49                          break;
50                      case 0x3:   /* SC instruction */
51                          if (funct3 == 3)    /* SC.D instruction */
52                          {
53                              instname = "scd";
54                              insttype = SCD;
55                          }
56                          else if (funct3 == 2)   /* SC.W instruction */
57                          {
58                              instname = "scw";
59                              instyype = SCW;
60                          }
61                          else
62                          {
63                              this->panic("Unknown 32bit funct3
64                                  0x%x\n", funct3);
65                          }
66                          break;
67                      default:
68                          this->panic("Unknown 32bit funct5
69                              0x%x\n", temp);
70                      }
71                  }
72                  ......
73              }
74          }
75          ......
76      }
77
78      void Simulator::excecute()
79      {
80          ......
81          bool _cas = false;  /* To identify whether it is a LR or SC */
82
83          switch (inst)
84          {
85              ......
86              case LRW:   /* Datapath of LR.W instruction */
87                  readMem = true;
88                  writeReg = true;
89                  memLen = 4;
```

```
90                  out = op1;
91                  readSignExt = true;
92                  _cas = true;
93                  break;
94              case LRD:   /* Datapath of LR.D instruction */
95                  readMem = true;
96                  writeReg = true;
97                  memLen = 8;
98                  out = op1;
99                  readSignExt = true;
100                 _cas = true;
101                 break;
102             case SCW:   /* Datapath of SC.W instruction */
103                 writeReg = true;
104                 writeMem = true;
105                 memLen = 4;
106                 out = op1;
107                 _cas = true;
108                 op2 = op2 & 0xFFFFFFFF;
109                 break;
110             case SCD:   /* Datapath of SC.D instruction */
111                 writeReg = true;
112                 writeMem = true;
113                 memLen = 8;
114                 out = op1;
115                 op2 = op2 & 0xFFFFFFFF;
116                 _cas = true;
117                 break;
118             ......
119         }
120         ......
121         /* Pass the information to the next stage */
122         this->eRegNew._cas = _cas;
123     }
124
125     void Simulator::memoryAccess()
126     {
127         ......
128         bool _cas = this->eReg._cas;
129         ......
130         if (writeMem)
131         {
132             switch (memLen)
133             {
134             case 1:
135                 good = this->memory->setByte(out, op2, &cycles);
136                 if ((out >= this->registry_addr) && (out <=
137                     (this->registry_addr + this->registry_num)))
138                 {   /* If there is any byte write to the reservation set */
```

```
139                     this->registry_whe = false;
140                 }
141                 break;
142             case 2:
143                 good = this->memory->setShort(out, op2, &cycles);
144                 for (uint32_t i = 0; i < 2; i++)
145                 {
146                     if (((out + i) >= this->registry_addr) && ((out + i)
147                         <= (this->registry_addr + this->registry_num)))
148                     {   /* Any byte write to the reservation set */
149                         this->registry_whe = false;
150                     }
151                 }
152                 break;
153             case 4:
154                 if (_cas)
155                 {   /* If this is SC instruction */
156                     if (this->registry_whe && (this->registry_addr
157                         == out) && (this->registry_num == 4))
158                     {   /* Reservation set is valid
159                             the first address same
160                             the access length same */
161                         good = this->memory->setInt(out, op2, &cycles);
162                         out = 0;
163                     }
164                     else
165                     {   /* If failed return code 1 */
166                         out = 1;
167                     }
168                     /* SC eliminate all reservation */
169                     this->registry_whe = false;
170                 }
171                 else
172                 {   /* If not, do basic store operation */
173                     for (uint32_t i = 0; i < 4; i++)
174                     {   /* For all bytes */
175                         if (((out + i) >= this->registry_addr) &&
176                             ((out + i) <= (this->registry_addr +
177                             this->registry_num)))
178                         {   /* Any access to the bytes */
179                             this->registry_whe = false;
180                         }
181                     }
182                     good = this->memory->setInt(out, op2, &cycles);
183                 }
184                 break;
185             case 8:
186                 if (_cas)
187                 {
```

```
188                     if (this->registry_whe && (this->registry_addr
189                         == out) && (this->registry_num == 8))
190                     {   /* Reservation set is valid
191                             the first address same
192                             the access length same */
193                         good = this->memory->setLong(out, op2, &cycles);
194                         out = 0;
195                     }
196                     else
197                     {
198                         out = 1;
199                     }
200                     tthis->registry_whe = false;
201                 }
202                 else
203                 {
204                     for (uint32_t i = 0; i < 8; i++)
205                     {
206                         if (((out + i) >= this->registry_addr)
207                             && ((out + i) <= (this->registry_addr
208                             + this->registry_num)))
209                         {
210                             this->registry_whe = false;
211                         }
212                     }
213                     good = this->memory->setLong(out, op2, &cycles);
214                 }
215             }
216         }
217         ......
218         if (readMem)
219         {   /* Handle with LR instruction */
220             switch (memLen)
221             {
222                 ......
223             case 4:
224                 if(readSignExt)
225                 {
226                     if(_cas)
227                     {   /* LR.W instruction */
228                         this->registry_whe = true;  /* Valid */
229                         this->registry_addr = out;  /* Store */
230                         this->registry_num = 4; /* Length */
231                     }
232                     out = (int64_t)this->memory->getInt(out, &cycles);
233                 }
234                 ......
235             case 8:
236                 if(readSignExt)
```

```
237                 {
238                     if(_cas)
239                     {   /* LR.W instruction */
240                         this->registry_whe = true;
241                         this->registry_addr = out;
242                         this->registry_num = 8;
243                     }
244                     out = (int64_t)this->memory->getInt(out, &cycles);
245                 }
246                 ......
247 
248             }
249 
250         }
251     }
```

# Realize CAS in lab0.c

## Requirement of CAS

### Details

· Input of function is three variables: address which we want to store, old value that we think it should be, new value that we want to store in the address.

· We first load the value from the address, compare to the old value, if they are same (means until now, this address is not modified), store the new value into it and return code 1(true), if they are not same (means it has already been modified), stop store and return code 0(false).

· To make sure the CAS operation would not been interrupted, we must use **LR** instruction to load and **SC** instruction to store, in order to make sure the content of the perticular address would not be modified while doing CAS operation. If **SC** failed, just try again until success.

## Idea

· I use inline assembler to make sure that it will use *LR* and *SC* instruction. Also use keyword **___volatile___** to make sure that the assembler will not optimize my code.

## Implementation

```
1 int CAS(long *dest, long new_value, long old_value)
2 {
3     int ret = 0;
4 
5     // TODO: write your code here
6     __asm__ __volatile__(
7         "retry: lr.w %0, %[output2]\n" /* Load value of addr */
```

```
8           "       bne   %0, %[input1], fail\n" /* Compare and check */
9           "       sc.w  %0, %[input2], %[output2]\n" /* Store new value */
10          "       bnez  %0, retry\n"  /* Whether atom operation succcess */
11          "       li    %[output1], 1\n"  /* Return code 1 */
12          "       j     success\n"    /* Go on following */
13          "fail:  li    %[output1], 0\n"  /* Return code 0 */
14          "success: \n"
15          : [output1] "=&r"(ret), [output2] "+A"(*dest)
16          : [input1] "rJ"(old_value), [input2] "rJ"(new_value)
17          : "memory");
18
19      return ret;
20  }
```

## Result

I compile *lab*0.*c* and run it, the following result is correct(some debug output, just ignore it), the first circulation should be failed, the second circulation should be success and store new value into the address.

```
ubuntu@VM-4-7-ubuntu:~/RISCV-Simulator/build$ ./Simulator ../riscv-elf/lab0.riscv
Dst address is: 71608
CAS FAIL
0
1
Dst address is: 71608
CAS SUCCESS
1
211
Program exit from an exit() system call
------------ STATISTICS -----------
Number of Instructions: 418
Number of Cycles: 625
Avg Cycles per Instrcution: 1.4952
Branch Perdiction Accuacy: 0.5000 (Strategy: Always Not Taken)
Number of Control Hazards: 65
Number of Data Hazards: 209
Number of Memory Hazards: 20
-----------------------------------
ubuntu@VM-4-7-ubuntu:~/RISCV-Simulator/build$
```

## Verification

I check the whole process and write some simple other test to my implementation, I believe it is right.