LECTURE 17: IMPLEMENTATION OF SPIN LOCK

Motivation- Mutual Exclusion

```
Initially: A = 0;
```

T_0

```
A = A+1;
```

T_1

```
A=A+1;
```

T_0

```
ld R1, A
daddi R1, R1, #1
sd R1, A
```

T_1

```
ld R2, A
daddi R2, R1, #1
sd R2, A
```

Possible execution order

```
ld R1,A
ld R2, A
daddi R2, R1, #1
sd R2, A
daddi R1, R1, #1
sd R1, A
```

Basic Synchronization Primitive - Lock

- Lock: primitive to protect critical section
- 2 values
  - 0 : lock is free; thread can proceed
  - 1 : lock is set; thread must wait
- 2 operations
  - acquire: set lock and wait if set
  - release: free lock

Wrong Implementation of Spin Lock

```
acquire:
          LD       R2, lock          ; R2 = M[lock]
          BNEZ     R2, acquire       ; if lock set repeat
          DADDI    R1, R0, #1        ; R1 = 1
          SD       R1, lock          ; M[lock] =1
          JR       R31               ; return
```

```
release:
          SD       R0, lock          ; M[lock] =0
          JR       R31               ; return
```

Wrong Implementation of Spin Lock

T_0

```
acquire:
         LD      R2, lock
         BNEZ    R2, acquire
         DADDI   R1, R0, #1
         SD      R1, lock
         JR      R31
```

T_1

```
acquire:
         LD      R2, lock
         BNEZ    R2, acquire
         DADDI   R1, R0, #1
         SD      R1, lock
         JR      R31
```

- Both threads think they acquired lock and enter critical section
- Need atomic read-modify-write instruction

Test-and-Set

Atomic RMW Instruction – Test-and-Set

- RMW – Read Modify Write instructions
  - read location and set it to another value (atomically)
  - return success or failure

- Simplest one: Test-and-Set
  - T&S R1, lock ; R1 = Mem[lock] $\oplus$ Mem[lock] = 1
  - **success** if R1 == 0; **failure** if R1!=0

Spin Lock with Test-and-Set

```
acquire:
        T&S     R2, lock        ; R2 = Mem[lock]   ⊕       Mem[lock]=1
        BNEZ    R2, acquire     ; if (R2!=0) goto acquire (lock was set)
        JR      R31             ; return
```

```
release:
        SD      R0, lock        ; M[lock] =0
        JR      R31             ; return
```

Other RMW Instructions

- Atomic exchange
  - exchange value in register for value in memory

```
EXCH      R1, lock  ;         T=Mem[lock]     ⊕       Mem[lock] = R1   ⊕       R1=T
```

- Fetch-and-Increment
  - fetch value from memory location and atomically increment it

```
F&I       R1, lock  ;         R1= Mem[lock]   ⊕       Mem[lock]++
```

- Load Linked and Store Conditional

Load Linked and Store Conditional

- Implementing atomic memory operations challenging
  - requires read and write in single, uninterruptable instruction
- Separate read and write + way of deducing pair was atomic
  - Load Linked or Load Locked

```
LL          R1, lock                ; R1=Mem[lock]
```

  - Store Conditional

```
SC          R2, lock                ; if (no other thread has written lock since LL)
                                    ;          Mem[lock] = R2  ⊕  R2 =1
                                    ; else R2=0
```

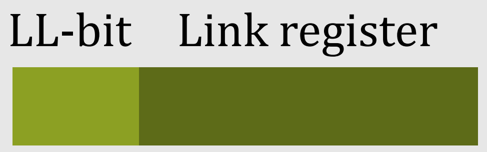
Other Sync Primitives Using LL and SC

- F&I implementation using LL and SC:

```
F&I:
            LL      R2, lock            ; R2=Mem[lock]
            DADDI   R3,R2,#1            ; R3=R2+1
            SC      R3, lock            ; store conditional
            BEQZ    R3, F&I             ; if SC failed (R3==0) retry
```

Implementation of LL and SC

- Bus interface supports LL-bit and link register
- LL executed – set LL-bit and copy address to link register
- Bus interface snoops update or invalidate signals
  - address == address in link register -> LL-bit cleared
- SC executed-> test LL-bit; if cleared, return failure
- Clear LL-bit also when lock variable is ejected from cache and on context switches
- Can have multiple LL-bits and link registers for different addresses



Thank You