CMPX CHG

CS 520
Dept. of Computer Science
Univ. of New Hampshire
How is locking implemented?

lock l
load sum
add value
store sum
unlock l

only one thread can hold lock at a time
but how is this done?
What is a lock?

Object that contains:
- thread id for owner of lock
- queue of waiting threads

But it is also shared by the threads!

Need to lock before we update it so chicken-and-egg problem!

Need a lock to implement a lock
jmp

Operation

Unconditional jump.

Format

jmp addr

Encoding

jmp = 20 (0x14)

Description

pc += addr.

cmpxchg

Operation

Compare and exchange.

Format

cmpxchg reg1, reg2, addr

Encoding

cmpxchg = 21 (0x15)

Description

if reg1 == *(pc + addr) then *(pc + addr) = reg2 else reg1 = *(pc + addr).

Note: this is done atomically by locking the memory bus for the duration of the instruction. If the effective address is out of range of the available memory, then the executing processor halts with an error. The comparison is an integer comparison.
implementation of the parallel pi computation in vm520 assembler

number of intervals must be evenly divisible by number of processors

machine only supports single precision so cannot get very accurate value for pi.

# computational core

# figure out how much work (number of intervals) to do
getpn r0
load r2, intervals
divi r2, r0
store r2, chunk  # chunk = number of intervals to process

# figure out where to start on the x-axis
# this is a pain because there is no instruction to convert int to float
# so: start_x = (chunk * processor_id + 0.5) * width
# re-write to: start_x = (chunk * processor_id * width) + (0.5 * width)
# and implement (chunk * processor_id * width) by repeated addition
getpid r0
load r1, chunk
mul r0, r1  # r0 = chunk * processor_id
ldimm r1, 0  # r1 = 0
load r2, width  # r2 = width
ldimm r3, 0  # r3 is the loop bound
ldimm r4, 1  # r4 used to decrement loop index
lab1:
    beq r0, r3, lab2  #
    add r1, r2  # r1 += width
    sub r0, r4  # r0 -= 1
    jmp lab1  # end while
lab2:
    load r3, oneHalf  # r3 = 0.5
    mulf r2, r3  # r2 = (width * 0.5)
    addf r1, r2  # r1 = (chunk * processor_id * width) + (0.5 * width)

# iterate over the chunk of intervals summing f(x) = 4.0/(1.0 + x^2)
# r1 contains the initial x value
load r0, chunk  # r0 = chunk (ie loop index)
ldimm r2, 0  # r2 = 0 (loop bound)
ldimm r5, 0  # r5 = 0 (sum)
load r6, width  # r6 = width
ldimm r7, 1  # r7 = 1 (to decrement loop index)
lab3:
    beq r0, r2, lab4  # while (r0 > r2)
    ldimm r3, 0  # r3 = x
    addf r3, r1
    mulf r3, r1  # r3 = x * x
    load r4, one  # r4 = 1.0
    addf r4, r3  # r4 = 1.0 + (x * x)
    load r3, four  # r3 = 4.0
    divf r3, r4  # r3 = 4.0 / (1 + (x * x))
    addf r5, r3  # sum += 4.0 / (1 + (x * x))
    addf r1, r6  # x += width
    subi r0, r7  # r0 -= 1
    jmp lab3  # end while
lab4:

# multiply the local sum by width
load r0, width
mulf r5, r0
# sum local answer into the answer word
# use a lock to protect this update when multiple processors
# busy wait on the lock
1dimm r1, 1
tryAgain:
1dimm r2, 0
cmpeq r2, r1, lock
beg r2, r1, tryAgain
# want to set the lock to 1
# need to wait until lock is 0
# if lock is 0 then lock it
# else r2 will be set to 1, and if so repeat
load r0, answer
addf r0, r5
store r0, answer
# now have exclusive control
# so safe to add in the local answer
1dimm r2, 0
store r2, lock unlock

# all done!
halt

This approach uses "spin/busy waiting".
It repeatedly try to lock until it works.
CMPXCHG can be used to do locking because its two memory accesses are atomic.

Memory bus is locked for the duration of the instruction, i.e., no other processor can execute a memory operation until the CMPXCHG completes.
vm 520 implementation

memory treated as shared data

a processor must lock a mutex before it accesses memory
`CMPXCHG`  

`CMPXCHG r32, r/m32`  

Compare eax with r/m32.  

*If equal, set ZF and r32 is stored into r/m32.*  
*Else, clear ZF and load r/m32 into eax.*
LOCK prefix can be used with CMPXCHG to make it execute atomically.

Note that CMPXCHG r32, m32 has two memory accesses:

1. read m32
2. write m32 if eax == m32

To do locking, we need these two steps to be atomic.
Implementing locks on Intel

To lock:

```
tryAgain:
    movl $0, %eax
    movl $1, %edx
    lock
    cmpxchgs %edx, %edx
```

.branch if

```
set %eax
```

To unlock:

```
movl %eax, %eax
```

```
if %eax == 0
    set %edx
else
    clr %edx
```

```
%eax = %eax
```

yield: allow another thread to run
CMPXCHG is used to obtain internal lock only held for short amount of time in order to update lock object