Atomic CISC Instructions

- atomic read-modify-write (RMW) instructions
  - test a certain condition σ referring to some memory addresses
  - if σ holds, modify (some of) the memory addresses
  - particular instances
  - atomic test-and-set instructions
  - atomic fetch-and-increment
  - atomic fetch-and-decrement

→ old fashioned, not in spirit of modern load/store architectures

Load/Store RMW Primitives

- how can instructions $I_1$ and $I_2$ be implemented?
  - consider MIPS instruction set
  - $I_1 \equiv \text{LL } rt,c(rs)$ is called load linked
    - load Mem[c + Reg[rs]] to Reg[rt]
    - store address c + Reg[rs] in a special register $L$
  - $I_2 \equiv \text{SC } rt,c(rs)$ is called store conditional
    - if $L \neq 0$, store Reg[rt] to memory address Mem[c + Reg[rs]]
    - and set Reg[rt] := 1
    - if $L = 0$, set Reg[rt] := 0

→ the cache controller helps $I_2$ to detect if there has been a write to Mem[L] between the execution of $I_1$ and $I_2$

Implementing Other Atomic Primitives

- example: exchange contents of Reg[4] and Mem[Reg[1]]
  
  LL R2,0(R1) ; load and protect Mem[Reg[1]]
  SC R3,0(R1) ; try Mem[Reg[1]] := Reg[3]
  BEQZ R3,try ; if not atomically executed, retry
  ADD R4,R2,R0 ; Reg[4] := Reg[5]

- example: fetch-and-increment memory location
  
  try: LL R2,0(R1) ; load and protect Mem[Reg[1]]
  ADDIU R3,R2,#1 ; increment Reg[2]
  SC R3,0(R1) ; try Mem[Reg[1]] := Reg[3]
  BEQZ R3,try ; if not atomically executed, retry

Semantics of Load/Store RMW Primitives

- $I_1 \equiv \text{LL } rt,c(rs)$ does the following
  
  next(Reg[rt]) := Mem[c + Reg[rs]]
  next(L) := c + Reg[rs];

- cache controller executes the following in each cycle
  
  if (WriteOnBus(L = AdrOnBus)) next(L) := 0;

- finally, $I_2 \equiv \text{SC } rt,c(rs)$ does the following
  
  if $(L \neq 0)$
    next(Reg[rs]) := Reg[rt];
  else next(Reg[rt]) := 0;
Implementing Other Atomic Primitives

- example: protect critical region by a lock variable Mem[Reg[1]]
  
  try: ADDI R3,R0,#1 ; Reg[3] := 1
  LL R2,(R1) ; load and protect Mem[Reg[1]]
  SC R3,(R1) ; try Mem[Reg[1]] := Reg[3]
  BEQZ R3,try ; if not atomically executed, retry
  BNEZ R2,try ; if region is locked, retry

- process can proceed if lock Mem[Reg[1]] is zero
- after critical code is executed, lock Mem[Reg[1]] must be reset
- note serialization of load/stores due to bus arbitration

Problem: Barrier Synchronization in a Loop

- the previous code must not be used in loop bodies
- the following could happen:
  - if all processes reached the barrier, release is set to 1
  - process P_i goes ahead and iterates the loop body, while the other processes do not proceed their execution
  - in the worst case, P_i reaches the barrier again, before the last process of the previous iteration has passed the barrier
  - a fast process can trap a slow process in the barrier by resetting release

Sense-Reversing Barrier Synchronization

- sense-reversing barrier synchronization solves the problem
- alternately, the processes wait either until release = 0 or release = 1 holds
- previous and new barrier synchronization due to a loop are no longer mixed up
- still possible that one process reaches barrier again before another has even left it
- however, this time the slow one can still leave the barrier

Bus Arbitration

- several processors may wish to access the bus
- arbitration required to manage access to the shared bus
- arbiters are implemented as hardware circuits (bus controller)
- simplest form: arbitration with static priorities
  - processors P_1, ... P_n have priorities 1, ... n
  - processor with highest priority yields the bus
  - problem: unfair arbitration P_1 may always use the bus
- fairness can be obtained by dynamic priorities
- simplest form: token ring
  - a token is sent from P_i to P_j mod n + 1
  - processor P_i with token yields the bus, if P_i wants to do so
  - problem: if only one P_i wants access to the bus, it may have to wait n cycles to receive the access, although the bus is available
  - combination of static and dynamic priorities recommended
Bus Arbitration

- combination of static and dynamic priorities
  - again, send a token from \( P_i \) to \( P_{(i \mod n)+1} \)
  - assume \( P_i \) currently has the token
  - \( P_j \) is granted access to the bus if:
    - either \( i \leq j \) and no \( P_k \) with \( i \leq k < j \) requests the bus
    - or neither a \( P_k \) with \( i \leq k \leq n \) nor a \( P_k \) with \( 1 \leq k < j \) requests the bus
  - in principle, priorities are changed by taken moves
- fair and efficient arbitration
- in the worst case, still \( n - 1 \) cycles have to be awaited for access

Memory Consistency Models

- memory consistency models determine the semantics of parallel execution
- several consistency models are used by multiprocessors
  - sequential consistency
  - weak consistency
  - release consistency
- depending on the consistency model, updates from one processor may not be immediately visible to all processors
- instead, other processors may notice the update only after some time or after some explicit synchronization operations

Sequential Consistency

- sequential consistency
  - load/stores of different processes may be arbitrarily interleaved
  - however, execution of a load/store can only be started if either load/store executions terminated and all updates are acknowledged
  - this holds trivially for snooping based protocols
  - however, directory based protocols require acknowledge messages to implement sequential consistency in order to know that a previous load/store is done
  - using sequential consistency, the previous program can not end with \( y_1 = y_2 = 2 \)

Weak Consistency

- sequential consistency is often too inefficient
  - since it requires to send a lot of acknowledge messages
  - weaker consistency model have been considered
    - several load/stores may be pending in messages of the network
    - explicit synchronization mechanisms guarantee that all load/stores are done
    - memory consistency is only given after such synchronization steps
- further references \([2, 49, 63]\)

Steinke-Nutt Hierarchy of Memory Models

- Steinke and Nutt \([96]\) present a categorization of weak memory models
  - to this end, a fixed number of processes \( P = \{ p_1, \ldots, p_m \} \) working on shared variables \( V = \{ x_1, \ldots, x_n \} \) are considered
  - each process \( p \in P \) performed a set of read/write actions:
    - \( \alpha_1 \preceq_p \ldots \preceq_p \alpha_n \)
    - note: each \( \preceq_p \) is transitive, but we only draw a few lines
  - different memory consistency models are defined on this basis
  - also, \([96]\) defined a lattice of memory consistency models by orthogonal consistency properties in a systematic way