Concurrent Systems

*Nebenläufige Systeme*

V. Elementary Operations

Wolfgang Schröder-Preikschat

December 3, 2020
Agenda

Preface

Primitive Instructions
  Atomic Operations

Memory Models
  Properties

Summary
Subject Matter

- discussion on **abstract concepts** as to elementary operations at instruction structure set architecture level
  - atomic load/store of a naturally aligned machine word
  - atomic read-modify-write of complex machine instructions
Subject Matter

- discussion on **abstract concepts** as to elementary operations at instruction structure set architecture level

- impartation of knowledge on memory models that are relevant to multi-threading on multi/many-core (multi-) processors
  - atomicity, visibility, and ordering of memory operations against the background of UMA, NUMA, and (partly) COMA architectures
  - ordering enforcing hardware such as memory barriers or fences, resp., allowing one to pattern sequential, relaxed, and weak **data consistency**
Subject Matter

- discussion on **abstract concepts** as to elementary operations at instruction structure set architecture level

- impartation of knowledge on memory models that are relevant to multi-threading on multi/many-core (multi-) processors

- excursion into practice of **hardware features** that are of importance for the implementation of any synchronisation algorithm
Outline

Preface

Primitive Instructions
  Atomic Operations

Memory Models
  Properties

Summary
of particular interest (at this point) are shared-memory operations
- commonality is the opportunity, at least, for indivisible execution
of particular interest (at this point) are **shared-memory operations**

note, all memory operations are also divisible in the following respect:

**sub-operation**

- processors are word-oriented, but memory is byte-oriented
- with *word size* as a multiple of *byte size*, e.g. $4 \times 8$ bits
- thus, loads/stores will operate on a **sequence of bytes**
Memory-Operation Semantics

- Of particular interest (at this point) are *shared-memory operations*.
- Note, all memory operations are also divisible in the following respect:
  - **sub-operation**: Processors are word-oriented, but memory is byte-oriented.
  - **sub-step**: Processors perform a *fetch-execute-cycle* to run programs. *n*-address machines mean *n*-operand instructions, *n* ≥ 2 \(^1\)
  - Thus, execution requires a sequence of *loads/stores*.

\(^1\)In general *n* ≥ 0, but only for *n* ≥ 2 becomes the problem apparent.
#include <stdint.h>

static int64_t label;

int64_t get_label() {
    return label;
}

void set_label(int64_t value) {
    label = value;
}
```c
#include <stdint.h>

static int64_t label;

int64_t get_label() {
    return label;
}

void set_label(int64_t value) {
    label = value;
}
```

in logical respect any of these single statements is indivisible, atomic
- lines 6 conceals a load and line 10 conceals a store operation
- each case forms an ELOP of the abstract processor “C”
```c
#include <stdint.h>

static int64_t label;

int64_t get_label() {
    return label;
}

void set_label(int64_t value) {
    label = value;
}

■ in logical respect any of these single statements is indivisible, atomic

■ in physical respect these statements are conditionally atomic, only
  ■ a matter of optimisation options, the CPU, and alignment restrictions
```
```c
#include <stdint.h>

static int64_t label;

int64_t get_label() {
    return label;
}

void set_label(int64_t value) {
    label = value;
}

Beware of devirtualisation!

- in logical respect any of these single statements is indivisible, atomic

- in physical respect these statements are conditionally atomic, only
```
gcc -m32...

get_label:
1. movl label, %eax
2. movl label+4, %edx
3. ret

set_label:
4. movl 4(%esp), %eax
5. movl 8(%esp), %ecx
6. movl %ecx, label+4
7. movl %eax, label
8. ret
Load/Store II

ASM—Level 4

gcc -m64...

12 get_label:
13 \texttt{movq \texttt{label}(\%rip), \%rax}
14 \texttt{ret}

16 set_label:
17 \texttt{movq \%rdi, label(\%rip)}
18 \texttt{ret}

actions 2-3 and 9-10 are divisible
any of these 8
\texttt{mov} instructions is
conditionally indivisible

beware of the processor architecture or the data alignment, resp.
usually, memory-word loads/stores are indivisible if “word” corresponds
to the smallest addressable unit of main memory: \texttt{bytes}, nowadays
on some architectures (e.g., x86) they are indivisible too if the address of
the memory operand is
naturally aligned
Load/Store II

gcc -m32...

get_label:
1. movl label, %eax
2. movl label+4, %edx
3. ret

set_label:
4. movl 4(%esp), %eax
5. movl 8(%esp), %ecx
6. movl %ecx, label+4
7. movl %eax, label
8. ret

gcc -m64...

get_label:
12. movq label(%rip), %rax
13. ret

set_label:
16. movq %rdi, label(%rip)
17. ret

- actions 2-3 and 9-10 are divisible
- any of these 8 mov instructions is conditionally indivisible
Load/Store II

gcc -m32...

1  get_label:
2     movl label, %eax
3     movl label+4, %edx
4     ret

5
6  set_label:
7     movl 4(%esp), %eax
8     movl 8(%esp), %ecx
9     movl %ecx, label+4
10    movl %eax, label
11    ret

gcc -m64...

12  get_label:
13     movq label(%rip), %rax
14     ret
15
16  set_label:
17     movq %rdi, label(%rip)
18     ret

- beware of the processor architecture or the data alignment, resp.
  - usually, memory-word loads/stores are indivisible if “word” corresponds to the smallest addressable unit of main memory: byte, nowadays
  - on some architectures (e.g., x86) they are indivisible too if the address of the memory operand is naturally aligned

© wosch, thoenig CS (WS 2020/21, LEC 5) Primitive Instructions–Atomic Operations


- execution cycle of a machine instruction that involves the ALU


\(^2\text{arithmetic-logic unit, the operation unit of the CPU.}\)
**Read-Modify-Write (RMW)**

- **execution cycle** of a machine instruction that involves the ALU\(^2\)
  - consists of the following individual operation steps:
    1. load input operands (acc. operation code or addressing mode, resp.)
    2. compute result (acc. operation code)
    3. store output operand (acc. operation code or addressing mode, resp.)

\(^2\)arithmetic-logic unit, the operation unit of the CPU.
Read-Modify-Write

- **execution cycle** of a machine instruction that involves the ALU\(^2\)
  - consists of the following individual operation steps:
    - i) load input operands (acc. operation code or addressing mode, resp.)
    - iii) store output operand (acc. operation code or addressing mode, resp.)
  - steps (i) and (iii) require the **bus** in case of memory-sensitive operations
    - reusable hardware resource, shareable, allocated per (load/store) step

\(^2\)arithmetic-logic unit, the operation unit of the CPU.
**execution cycle** of a machine instruction that involves the ALU\(^2\) consists of the following individual operation steps:

- **i** load input operands (acc. operation code or addressing mode, resp.)
- **ii** compute result (acc. operation code)
- **iii** store output operand (acc. operation code or addressing mode, resp.)

- typical **compound action** at instruction set architecture (ISA) level
  - is memory-sensitive only for a *complex instruction set computer* (CISC)

\(^2\) *arithmetic-logic unit*, the operation unit of the CPU.
**execution cycle** of a machine instruction that involves the ALU^2

- consists of the following individual operation steps:
  1. load input operands (acc. operation code or addressing mode, resp.)
  2. compute result (acc. operation code)
  3. store output operand (acc. operation code or addressing mode, resp.)

In a **multiprocessor case**, the whole cycle is divisible (non-atomic)

- merely the individual sub-steps may form indivisible actions (cf. p. 8)
- while the loads/stores may be in sync, the compound action is not

---

^2 *arithmetic-logic unit*, the operation unit of the CPU.
execution cycle of a machine instruction that involves the ALU\(^2\) consists of the following individual operation steps:

- load input operands (acc. operation code or addressing mode, resp.)
- compute result (acc. operation code)
- store output operand (acc. operation code or addressing mode, resp.)

in a multiprocessor case, the whole cycle is divisible (non-atomic)

indivisibility requires a bus lock for the duration of the whole cycle:

- an atomic RMW instruction that implicitly performs the lock
- a lock prefix that makes the adjacent normal RMW instruction atomic

\(^2\) arithmetic-logic unit, the operation unit of the CPU.
Definition (TS, acc. IBM System/370)

The leftmost bit (bit position 0) of the byte located at the second-operand address is used to set the condition code, and then the entire addressed byte is set to all ones. [8, p. 144]
Definition (TS, acc. IBM System/370)

The leftmost bit (bit position 0) of the byte located at the second-operand address is used to set the condition code, and then the entire addressed byte is set to all ones. [8, p.144]

The operation effectively does an **unconditional store** in main memory.

- The byte in storage is set to all ones as it is fetched for the testing of bit position 0. [8, p.144] \(^3\)

\(^3\)A similar effect has ldstub of SPARC V9.
Definition (TS, acc. IBM System/370)

The leftmost bit (bit position 0) of the byte located at the second-operand address is used to set the condition code, and then the entire addressed byte is set to all ones. [8, p. 144]

The operation effectively does an unconditional store in main memory.

In terms of main memory significance, this translates into the following:
- with effective address \( \text{ref} \) being the second-operand address \( D_2(B_2) \)

```c
bool tas(byte *ref) {
    atomic { bool cc = *ref & 0x1; *ref = 0x11111111; }
    return cc;
}
```
Test & Set I

Definition (TS, acc. IBM System/370)

The leftmost bit (bit position 0) of the byte located at the second-operand address is used to set the condition code, and then the entire addressed byte is set to all ones. [8, p. 144]

- the operation effectively does an **unconditional store** in main memory

```plaintext
bool tas(byte *ref) {
    atomic { bool cc = *ref & 0x1; *ref = 0x11111111; }
    return cc;
}
```

- note that TS interlocks against simultaneous main memory accesses

© wosch, thoenig  
CS (WS 2020/21, LEC 5)  
Primitive Instructions – Atomic Operations
the original copy (IBM System/370) has **swapping characteristic**

- \(\text{swap}(x, y)\), with \(x = \ast ref_0\) and \(y = 11111111_2[0]\)
the original copy (IBM System/370) has **swapping characteristic**

for a contemporary processor (x86), this translates into the following:

```c
bool tas(any_t *ref) {
    return TAS(ref);
}

tas:
    movl 4(%esp), %ecx
    movl $1, %eax
    xchgl %eax, (%ecx)
    ret
```

whereby (using GCC atomic built-in functions):

```c
#define TAS(ref) __sync_lock_test_and_set(ref, 1)
```

note that **xchg** interlocks against simultaneous main memory accesses.

beware of the unconditional store carried out by both **TS** and **xchg**.

this semantic has a deleterious effect for cache-coherent processors.

the cache line holding the main memory operand is always invalidated,
→ dedicated hardware implementation (p.35) or mapping to **CAS** (p.12)

Same holds for **TAS** of the M68000 family and **ldstub** of the SPARC family.
the original copy (IBM System/370) has **swapping characteristic**

- for a contemporary processor (x86), this translates into the following:

```c
bool tas(any_t *ref) {
    return TAS(ref);
}
```

```assembly
tas:
    movl 4(%esp), %ecx
    movl $1, %eax
    xchg %eax, (%ecx)
    ret
```

- whereby (using GCC atomic built-in functions):

```c
#define TAS(ref) __sync_lock_test_and_set(ref, 1)
```

- note that `xchg` interlocks against simultaneous main memory accesses. Beware of the unconditional store carried out by both `TS` and `xchg`.

4 This semantic has a deleterious effect for cache-coherent processors; the cache line holding the main memory operand is always invalidated, → dedicated hardware implementation (p.35) or mapping to CAS (p.12)

- Same holds for `TAS` of the M68000 family and `ldstub` of the SPARC family.
the original copy (IBM System/370) has **swapping characteristic**

for a contemporary processor (x86), this translates into the following:

```c
bool tas(any_t *ref) {
    return TAS(ref);
}
```

whereby (using GCC atomic built-in functions):

```c
#define TAS(ref) __sync_lock_test_and_set(ref, 1)
```

note that `xchg` interlocks against simultaneous main memory accesses
the original copy (IBM System/370) has **swapping characteristic**

- for a contemporary processor (x86), this translates into the following:

```c
bool tas(any_t *ref) {
    return TAS(ref);
}

tas:
    movl 4(%esp), %ecx
    movl $1, %eax
    xchgl %eax, (% ecx)
    ret
```

- whereby (using GCC atomic built-in functions):

```c
#define TAS(ref) __sync_lock_test_and_set(ref, 1)
```

- beware of the unconditional store carried out by both TS and xchg

- this semantic has a **deleterious effect** for cache-coherent processors

- the cache line holding the main memory operand is always invalidated

- dedicated hardware implementation (p. 35) or mapping to CAS (p. 12)

---

4Same holds for TAS of the M68000 family and ldstub of the SPARC family.
Definition (CS, acc. IBM System/370)

*The first and second operands are compared. If they are equal, the third operand is stored in the second-operand location. If they are unequal, the second operand is loaded into the first-operand location.* [8, p. 123]
The first and second operands are compared. If they are equal, the third operand is stored in the second-operand location. If they are unequal, the second operand is loaded into the first-operand location. [8, p. 123]

The operation effectively performs a **conditional store** in main memory.

- The first and third operands [each are] occupying a general register. The second operand is a word in main storage. [8, p. 123]

```c
atomic bool cas (register old, word *ref, register new) {
    bool cc;
    return cc = (*ref == old) ? (*ref = new, 0) : (old = *ref, 1);
}
```
Compare & Swap I  

**Definition (CS, acc. IBM System/370)**

*The first and second operands are compared. If they are equal, the third operand is stored in the second-operand location. If they are unequal, the second operand is loaded into the first-operand location.* [8, p. 123]

the operation effectively performs a **conditional store** in main memory

- in terms of **main memory significance**, this translates into the following:

```c
atomic bool cas(register old, word *ref, register new) {
  bool cc;
  return cc = (*ref == old) ? (*ref = new, 0) : (old = *ref, 1);
}
```
Definition (CS, acc. IBM System/370)

The first and second operands are compared. If they are equal, the third operand is stored in the second-operand location. If they are unequal, the second operand is loaded into the first-operand location. [8, p. 123]

The operation effectively performs a **conditional store** in main memory

```c
atomic bool cas(register old, word *ref, register new) {
    bool cc;
    return cc = (*ref == old) ? (*ref = new, 0) : (old = *ref, 1);
}
```

Note that CS interlocks against simultaneous main memory accesses
“textbook semantics” of TAS has a deleterious effect for the cache:

```c
bool tas(word *ref) {
    atomic {
        word aux = *ref; *ref = 1;
    }
    return aux;
}
```

Same is true when using the GCC atomic built-in function (x86, cf. p11):

```c
#define TAS(ref) __sync_lock_test_and_set(ref, 1)
```

Worst-case overhead of five instructions (cf. p11) pays off, depending on processor and cache architecture.
use of CAS, with \texttt{define} \texttt{CAS \_\_sync\_bool\_compare\_and\_swap}:

```
bool tas(long *ref) {
    return CAS(ref, 0, 1);
}
```

- worst-case overhead of five instructions (cf. p 11)
- pays off, depending on processor and cache architecture

```
tax:    xorl %eax, %eax
        movl $1, %ecx
        movl 4(%esp), %edx
        lock
        cmpxchgl %ecx, (%edx)
        testl %eax, %eax
        sete %al
        movzbl %al, %eax
        ret
```
The ABA problem is a **false positive** execution of a CAS-based speculation on a shared location $L_i$. [2, p. 186]
**Definition (ABA, also A-B-A)**

The ABA problem is a **false positive** execution of a CAS-based speculation on a shared location $L_i$. [2, p. 186]

when the successful execution of a CAS instruction indicates:

- that the two operands subject to comparison are equal and, thus, purport the presence of a certain global condition (*positive*),
- but this global condition is not in fact present (*false*)
Definition (ABA, also A-B-A)

The ABA problem is a false positive execution of a CAS-based speculation on a shared location $L_i$. [2, p. 186]

- assuming that processes $P_1$ and $P_2$ simultaneously access location $L_i$
- value $A$ read by $P_1$ from $L_i$ be a sign of a dedicated global state $S_1$, but $P_1$ will be delayed before being able to commit a new value to $L_i$
- meanwhile $P_2$ changes the value of $L_i$ to $B$ and then back to $A$, defining a new global state $S_2 \neq S_1$
- $P_1$ resumes, observes that the value of $L_i$ equals $A$ and, thus, acts on the assumption that the global state must be $S_1$—which is no longer true
Definition (ABA, also A-B-A)

The ABA problem is a **false positive** execution of a CAS-based speculation on a shared location $L_i$. [2, p. 186]

- assuming that processes $P_1$ and $P_2$ simultaneously access location $L_i$

- severity of false positive execution depends on the problem (cf. p. 36)
Load-Linked/Store-Conditional I

**Definition**

Paired instructions to form a flow of actions without any guarantee of indivisibility but that it succeeds only in case of indivisible operation.

- originated in the MIPS II or R6000, resp., RISC architecture [9]

LL/SC
Load-Linked/Store-Conditional I

**Definition**

Paired instructions to form a flow of actions without any guarantee of indivisibility but that it succeeds only in case of indivisible operation.

originated in the MIPS II or R6000, resp., RISC architecture [9]:

- **LL**
  - loads a word from the specified effective memory address
  - makes a reservation on that very address (range)\(^5\)

- **SC**
  - checks for a reservation on the specified effective memory address\(^5\)
  - if the reservation persists, stores the specified word at that address
  - delivers the result of the reservation check

---

\(^5\)The dimension of the reservation depends on the hardware implementation. It may be exact the effective address or a larger address range around.
Definition

Paired instructions to form a flow of actions without any guarantee of indivisibility but that it succeeds only in case of indivisible operation.

- originated in the MIPS II or R6000, resp., RISC architecture [9]:

- reasons for **cancellation** of a persisting address (range) reservation:
  1. successful execution of SC—hoped for, normally
  2. execution of LL by another processor applying the same address (range)
  3. an exception (trap/interrupt) on the processor holding the reservation
Load-Linked/Store-Conditional I

**Definition**

Paired instructions to form a flow of actions without any guarantee of indivisibility but that it succeeds only in case of indivisible operation.

- originated in the MIPS II or R6000, resp., RISC architecture [9]:
  - LL and SC interlock against simultaneous main memory accesses
use of LL/SC to recreate TAS and CAS:
  - in case of TAS, a boolean variable is conditionally set true

```c
bool tas(long *ref) {
    return (LL(ref) == 0) && SC(ref, 1);
}
```

- in case of CAS, a memory word is conditionally overwritten

```c
bool cas(long *ref, long old, long new) {
    return (LL(ref) == old) && SC(ref, new);
}
```
Load-Linked/Store-Conditional II

use of LL/SC to recreate TAS and CAS:

- in case of TAS, a boolean variable is conditionally set true

```c
bool tas(long *ref) {
    return (LL(ref) == 0) && SC(ref, 1);
}
```

- in case of CAS, a memory word is conditionally overwritten

```c
bool cas(long *ref, long old, long new) {
    return (LL(ref) == old) && SC(ref, new);
}
```

note that this implementation of CAS is free from the ABA problem:

- \(P_1\) shares location \(ref\) with \(P_2\), established reservation \(ref_{P_1}\) by LL
  - gets delayed for some reason, thus has not yet executed SC
- \(P_2\) overlaps \(P_1\), establishes reservation \(ref_{P_2}\) and, thus, cancels \(ref_{P_1}\)
  - successfully executes SC \(\Rightarrow\) CAS succeeds
- \(P_1\) resumes \(\Rightarrow\) SC will fail because reservation \(ref_{P_1}\) is invalid
  - returns failure of CAS \(\Rightarrow\) rolls back, backs up, and retries CAS...
Load-Linked/Store-Conditional II

- use of LL/SC to recreate TAS and CAS:
  - in case of TAS, a boolean variable is conditionally set true

```c
bool tas(long *ref) {
    return (LL(ref) == 0) && SC(ref, 1);
}
```

- in case of CAS, a memory word is conditionally overwritten

```c
bool cas(long *ref, long old, long new) {
    return (LL(ref) == old) && SC(ref, new);
}
```

- note that this implementation of CAS is free from the ABA problem:
  - \( P_1 \)
    - shares location \( ref \) with \( P_2 \), established reservation \( ref_{P_1} \) by LL
    - gets delayed for some reason, thus has not yet executed SC
  - \( P_2 \)
    - overlaps \( P_1 \), establishes reservation \( ref_{P_2} \) and, thus, cancels \( ref_{P_1} \)
    - successfully executes SC \( \Rightarrow \) CAS succeeds
  - \( P_1 \)
    - resumes \( \Rightarrow \) SC will fail because reservation \( ref_{P_1} \) is invalid
    - returns failure of CAS \( \Rightarrow \) rolls back, backs up, and retries CAS...
Semantics-aware emulation of TAS and CAS

- TAS and CAS provide the result of a comparison, not of a link check
- that SC can fail is irrelevant to the result
- only if the comparison fails or SC succeeds the action (TAS, CAS) is done

```c
inline bool LCS (long *ref, long old, long new) {
    do if (LL(ref) != old)
        return false;
    while (! SC(ref, new));
    return true;
}
```

- but this makes a wait-free operation lock-free [7]:
Semantics-aware emulation of TAS and CAS

use of an LL/SC-based auxiliary function:

```c
inline bool LCS(long *ref, long old, long new) {
    do if (LL(ref) != old) return false;
    while (!SC(ref, new));
    return true;
}
```

- load, compare, and then store, if the link still exists, or retry
Semantics-aware emulation of TAS and CAS

use of an LL/SC-based auxiliary function:

```c
inline bool LCS(long *ref, long old, long new) {
    do if (LL(ref) != old) return false;
    while (!SC(ref, new)); return true;
}
```

now the implementation of TAS and CAS, again, is child’s play:

```c
bool tas(long *ref) {
    return LCS(ref, 0, 1);
}
bool cas(long *ref, long old, long new) {
    return LCS(ref, old, new);
}
```
use of an LL/SC-based auxiliary function:

```cpp
inline bool LCS(long *ref, long old, long new) {
    do if (LL(ref) != old) return false;
    while (!SC(ref, new)); return true;
}
```

now the implementation of TAS and CAS, again, is child’s play:

```cpp
bool tas(long *ref) {
    return LCS(ref, 0, 1);
}

bool cas(long *ref, long old, long new) {
    return LCS(ref, old, new);
}
```

but this makes a wait-free operation lock-free [7]: a process can starve
Fetch & Add

Definition (acc. [6, p. 17])

A value-returning instruction that operates on a global (i.e., shared) variable $G$ and a local variable $L$. 

= FAA($G$, $L$) ≡ AAF($G$, $L$) − $L$

whereby (cf. p.39):

FAA($G$, $L$) ≡ FAA($G$, $L$) + $L$
**Fetch & Add**

<table>
<thead>
<tr>
<th>Definition (acc. [6, p. 17])</th>
</tr>
</thead>
<tbody>
<tr>
<td>A value-returning instruction that operates on a global (i.e., shared) variable $G$ and a local variable $L$.</td>
</tr>
</tbody>
</table>

- an atomic RMW instruction, inspired by “Replace Add” [3, p. 6]
  - prefix (FAA) or postfix (AAF) form, as to when fetch becomes effective
    - **prefix** – save the old value of $G$ for return, then add $L$ to $G$
    - **postfix** – add $L$ to $G$, then return the new value of $G$
Fetch & Add

**Definition (acc. [6, p. 17])**

A value-returning instruction that operates on a global (i.e., shared) variable $G$ and a local variable $L$.

- an atomic RMW instruction, inspired by “Replace Add” [3, p. 6]
  - prefix (FAA) or postfix (AAF) form, as to when fetch becomes effective

- whereby (cf. p. 39):

\[
\begin{align*}
\text{FAA}(G, L) & \equiv \text{AAF}(G, L) - L \\
\text{AAF}(G, L) & \equiv \text{FAA}(G, L) + L
\end{align*}
\]
**Fetch & Add**

**Definition (acc. [6, p. 17])**

A value-returning instruction that operates on a global (i.e., shared) variable $G$ and a local variable $L$.

- an atomic RMW instruction, inspired by “Replace Add” [3, p. 6]
  - prefix (FAA) or postfix (AAF) form, as to when fetch becomes effective

- whereby (cf. p. 39):
  
  $$\text{FAA}(G, L) \equiv AAF(G, L) - L$$  
  $$\text{AAF}(G, L) \equiv \text{FAA}(G, L) + L$$

- transferable to any associative binary operation *fetch-and-$\Phi$*  
  - but for noninvertible operations the prefix form is considered more general 
  - be $\Phi = \text{max}$ (i.e., $X$): only $XAF(G, L) \equiv \text{max}(FAX(G, L), L)$ (cf. p. 40)
Properties Relevant to Multi-Threading

fundamental characteristics that are of particular importance for the implementation of any synchronisation algorithm
fundamental characteristics that are of particular importance for the implementation of any synchronisation algorithm:

**atomicity**
- as to *how* certain machine instructions are executed
- differentiates in RISC and CISC machines
- specific to each ELOP that was discussed before (pp. 7–18)
Properties Relevant to Multi-Threading

- fundamental characteristics that are of particular importance for the implementation of any synchronisation algorithm:
  - **atomicity**  ■ as to *how* certain machine instructions are executed
  - **visibility**  ■ as to *when* memory-cell changes are observable
    ■ concerns delays in sensing the most recent memory-word write
    ■ introduces time factors on the availability of written data
Properties Relevant to Multi-Threading

fundamental characteristics that are of particular importance for the implementation of any synchronisation algorithm:

- **atomicity** as to *how* certain machine instructions are executed

- **visibility** as to *when* memory-cell changes are observable

- **ordering** as to *how* memory operations appear to be performed
  - stands for a variant of *out-of-order* execution
  - reflects on (sequential, relaxed, or weak) consistency models

atomicity applies to all other—and to a single machine instruction, only
visibility depends on the memory architecture, may cause "jitter"
ordering comprises multiple machine instructions, may cause "fencing"
as to the level of abstraction, they must all be considered together
this is especially true for the operating-system machine level (i.e., level 3)
Properties Relevant to Multi-Threading

- fundamental characteristics that are of particular importance for the implementation of any synchronisation algorithm:
  - **atomicity**: as to how certain machine instructions are executed
  - **visibility**: as to when memory-cell changes are observable
  - **ordering**: as to how memory operations appear to be performed

- these properties are linked with each other, are mutual prerequisites:
  - atomicity applies to all other—and to a single machine instruction, only
  - visibility depends on the memory architecture, may cause “jitter”
  - ordering comprises multiple machine instructions, may cause “fencing”
Properties Relevant to Multi-Threading

- fundamental characteristics that are of particular importance for the implementation of any synchronisation algorithm:
  - atomicity: as to how certain machine instructions are executed
  - visibility: as to when memory-cell changes are observable
  - ordering: as to how memory operations appear to be performed

- these properties are linked with each other, are mutual prerequisites

- as to the level of abstraction, they must all be considered together
  - this is especially true for the operating-system machine level (i.e., level 3)
common are two classes of memory-sensitive operations (cf. p. 25):

**L/S**
- atomic load (L) or store (S), resp., as single action
- granularity is the **machine word**, i.e., a multiple of a byte
- with **word-alignment** constraint on the operand address, usually
  - only word-aligned accesses will be carried out indivisibly

**RMW**
- atomic read (R), modify (M), and write (W) as single action
- common for CISC and, there, for **two-address machines**
  - uncommon for RISC, which is characteristic of load/store principle
- single- or double-word cycles for 32- or 64-bit architectures, resp.
  - “double” means “physically consecutive” or “logically interrelated”
  - i.e.: CDS or cmpxchg8b/cmpxchg16b compared to DCAS or CAS2
Atomicity

- common are two classes of memory-sensitive operations (cf. p. 25):
  - **L/S** atomic load (L) or store (S), resp., as single action

- **RMW** atomic read (R), modify (M), and write (W) as single action

- processes cannot observe any intermediate steps and partial effects
  - here, only in matters of a single (L/S or RMW) machine instruction
  - that is to say, the ISA-level action appears *indivisible* and *irreducible*
  - as a consequence, the instruction will be performed entirely or not all
    - with the latter meaning *failure indication* (TAS, CAS, SC)
Visibility

Hegemony of ccNUMA—still

When other interacting processes will notice the changes made by the current process, and whether they will notice them at all.

- depends on the **memory architecture** and behaviour of read or write operations to the same memory location.
When other interacting processes will notice the changes made by the current process, and whether they will notice them at all, depends on the memory architecture and behaviour of read or write operations to the same memory location.

**UMA**
- uniform memory architecture $\sim$ the same access time
- each address is assigned a fixed home in the global address space
- no processor uses private (local) memory besides shared memory
Visibility

Hegemony of ccNUMA—still

When other interacting processes will notice the changes made by the current process, and whether they will notice them at all, depends on the memory architecture and behaviour of read or write operations to the same memory location.

NUMA

- non-uniform memory architecture $\Rightarrow$ different access times
- each address is assigned a fixed home in the global address space
- each processor ("NUMA node") uses private (local) memory, too
Visibility

Hegemony of ccNUMA—still

When other interacting processes will notice the changes made by the current process, and whether they will notice them at all.

- depends on the **memory architecture** and behaviour of read or write operations to the same memory location

**COMA**
- **cache-only memory architecture** $\Rightarrow$ different access times
- no address is assigned a fixed home in the global address space
- each processor uses private (local) memory, only
Visibility

Hegemony of ccNUMA—still

When other interacting processes will notice the changes made by the current process, and whether they will notice them at all.

- depends on the **memory architecture** and behaviour of read or write operations to the same memory location
  - **UMA**  ■ *uniform memory architecture*
  - **NUMA**  ■ *non-uniform memory architecture*
  - **COMA**  ■ *cache-only memory architecture*

- orthogonal with it is the **consistency** aspect as to shared information stored in multiple local **caches**
  - *cache-coherent (cc) v. non-cache-coherent (ncc) memory architecture*
UMA (symmetric multiprocessing, SMP)

- M (memory)
- P (processor)

Bus interconnect

NUMA node (N) zone of uniform memory characteristic
distance number of (network) hops to distant memory

UMA/NUMA combination
Memory Architectures at a Glance

**Simplified**

### NUMA

- **NUMA node (N)**
  - zone of uniform memory characteristic
- **NUMA distance**
  - number of (network) hops to distant memory
Memory Architectures at a Glance

Simplified

- **UMA** (symmetric multiprocessing, SMP) bus interconnect
- **NUMA** scalable interconnect
- **COMA** scalable interconnect

- **COMA distance**
  - number of (network) hops to distant memory
**Memory Architectures at a Glance**

**Simplified**

**UMA** (*symmetric multiprocessing*, SMP)
- **M**
- **P**
  - bus interconnect

**NUMA**
- **M**
  - scalable interconnect
  - **N**
  - **P**

**COMA**
- **M**
  - scalable interconnect
  - **P**

**Key Terms:***
- **NUMA node (N)**
- zone of uniform memory characteristic
- **NUMA/COMA distance**
- number of (network) hops to distant memory
- **UMA/NUMA combination**
Ordering

What memory re-orderings are possible for a process, relatively to the order as specified by its program.
Ordering

What memory re-orderings are possible for a process, relatively to the order as specified by its program.

To improve performance, memory-sensitive machine instructions are not executed in the order originally specified by the program.

- On the one hand, the compiler reorders ($L_3$) instructions before run-time.
- On the other hand, the CPU reorders ($L_2$) instructions at run-time.
  - It is this aspect of **dynamic ordering** that is of relevance in the following...

---

$^6$According to the actual level of abstraction: operating-system machine ($L_3$) or instruction set architecture ($L_2$) level. See also [10] or [17, p. 34].
Ordering

What memory re-orderings are possible for a process, relatively to the order as specified by its program.

- to improve performance, memory-sensitive machine instructions are not executed in the order originally specified by the program.

- mainly, dynamic ordering is an issue of non-blocking synchronisation.
  - as blocking synchronisation implicitly can take care of “fencing” proper
    - depending on the kind of critical section and type of data dependency.
  - but, critical section per se is no guarantee for memory ordering (cf. p. 25).

© wosch, thoenig  CS (WS 2020/21, LEC 5)  Memory Models – Properties 24
Ordering

What memory re-orderings are possible for a process, relatively to the order as specified by its program.

- to improve performance, memory-sensitive machine instructions are not executed in the order originally specified by the program.

- mainly, dynamic ordering is an issue of non-blocking synchronisation.

- ordering ensuring needs special instructions: memory barrier/fence.
Dynamic Ordering

assuming that the following function is executed by a single processor, but the global variables are then read by at least one more processor:

```c
int a = 1, b = 2;  // what values of a and b do other processors see once line 6 has been reached by one processor?

void ab_set() {
    a = 3;
    b = 4;
}
```

what values of `a` and `b` do other processors see once line 6 has been reached by one processor?

depending on processor and memory architecture writes are not necessarily seen by other processors in the order as specified by the program!

assuming that the next function is executed directly afterwards to the former one just discussed, but by a different processor:

```c
void ab_get (int ab [2]) {
    ab [0] = b;
    ab [1] = a;
}
```

what values of `a` and `b` are delivered?

line 8 may read the new value of `b` while line 9 may read the old value of `a` although the assignment to `a` (line 4) was instructed previous to the one of `b`.
Dynamic Ordering

assuming that the following function is executed by a single processor, but the global variables are then read by at least one more processor:

```c
int a = 1, b = 2;

void ab_set () {
    a = 3;
    b = 4;
}
```

what values of \(a\) and \(b\) do other processors see once line 6 has been reached by one processor?

\((1, 2), (1, 4), (3, 2), (3, 4)\)

depending on processor and memory architecture
assuming that the following function is executed by a single processor, but the global variables are then read by at least one more processor:

```c
int a = 1, b = 2;

void ab_set () {
    a = 3;
    b = 4;
}
```

what values of \( a \) and \( b \) do other processors see once line 6 has been reached by one processor?

- \((1, 2), (1, 4), (3, 2), (3, 4)\)
- depending on processor and memory architecture

writes are not necessarily seen by other processors in the order as specified by the program!
assuming that the following function is executed by a single processor, but the global variables are then read by at least one more processor:

```c
int a = 1, b = 2;

void ab_set() {
    a = 3;
    b = 4;
}
```

what values of \( a \) and \( b \) do other processors see once line 6 has been reached by one processor?

\((1, 2), (1, 4), (3, 2), (3, 4)\)
depending on processor and memory architecture.

writes are not necessarily seen by other processors in the order as specified by the program!

assuming that the next function is executed directly afterwards to the former one just discussed, but by a different processor:

```c
void ab_get(int ab[2]) {
    ab[0] = b;
    ab[1] = a;
}
```

what values of \( a \) and \( b \) are delivered?

line 8 may read the new value of \( b \) while line 9 may read the old value of \( a \) although the assignment to \( a \) (line 4) was instructed previous to the one of \( b \).
assuming that the following function is executed by a single processor, but the global variables are then read by at least one more processor:

```c
int a = 1, b = 2;

void ab_set() {
    a = 3;
    b = 4;
}
```

what values of \(a\) and \(b\) do other processors see once line 6 has been reached by one processor?

\((1, 2), (1, 4), (3, 2), (3, 4)\) depending on processor and memory architecture

writes are not necessarily seen by other processors in the order as specified by the program!

assuming that the next function is executed directly afterwards to the former one just discussed, but by a different processor:

```c
void ab_get(int ab[2]) {
    ab[0] = b;
    ab[1] = a;
}
```

what values of \(a\) and \(b\) are delivered?

- line 8 may read the \textit{new} value of \(b\) while line 9 may read the \textit{old} value of \(a\)

- although the assignment to \(a\) (line 4) was instructed previous to the one of \(b\)
Memory barrier instructions directly control only the interaction of a CPU with its cache, with its write-buffer that holds stores waiting to be flushed to memory, and/or its buffer of waiting loads or speculatively executed instructions. [12]
Memory barrier instructions directly control only the interaction of a CPU with its cache, with its write-buffer that holds stores waiting to be flushed to memory, and/or its buffer of waiting loads or speculatively executed instructions. [12]

- \( \text{ld}_a \text{ LoadLoad} \text{ ld}_b \): ensures that \( a \) is read before \( b \) is accessed\(^7\)
  - speculative loads, out-of-order processing
- \( \text{st}_a \text{ StoreStore} \text{ st}_b \): ensures that \( a \) is visible before \( b \) is flushed\(^7\)
  - disordered flushes from write buffers
- \( \text{ld}_a \text{ LoadStore} \text{ st}_b \): ensures that \( a \) is read before \( b \) is flushed\(^7\)
  - out-of-order processors that can bypass loads
- \( \text{st}_a \text{ StoreLoad} \text{ ld}_b \): ensures that \( a \) is visible before \( b \) is accessed\(^7\)
  - write to same location by another processor

\(^7\)Including the execution of all subsequent loads or stores, resp.
Memory barriers instructions directly control only the interaction of a CPU with its cache, with its write-buffer that holds stores waiting to be flushed to memory, and/or its buffer of waiting loads or speculatively executed instructions. [12]

- \texttt{ld}_a \texttt{LoadLoad} \texttt{ld}_b \quad \text{ensures that } a \text{ is read before } b \text{ is accessed}^7

- \texttt{st}_a \texttt{StoreStore} \texttt{st}_b \quad \text{ensures that } a \text{ is visible before } b \text{ is flushed}^7

- \texttt{ld}_a \texttt{LoadStore} \texttt{st}_b \quad \text{ensures that } a \text{ is read before } b \text{ is flushed}^7

- \texttt{st}_a \texttt{StoreLoad} \texttt{ld}_b \quad \text{ensures that } a \text{ is visible before } b \text{ is accessed}^7

- CAS and LL/SC typically include a \texttt{StoreLoad} barrier on the target
  - i.e., not only a general-purpose but also the most expensive fence

^7 Including the execution of all subsequent loads or stores, resp.
Consistency Models

- **data consistency** as close as possible to sequential processes or with optimisation margins for high-latency memory.
data consistency as close as possible to sequential processes or with optimisation margins for high-latency memory

**Sequential**
- Processors see writes on the same target in the same order.
- But the order may appear different for an “external observer”\(^8\).
- Two requirements: **program order** and **write atomicity** [11].

---

\(^8\)Weaker than “strict consistency” that requires a read from a memory location to return the value of the most recent write.
data consistency as close as possible to sequential processes or with optimisation margins for high-latency memory

relaxed
- in terms of the constraints defined by sequential consistency
- as to (i) program order, (ii) write atomicity, or (iii) both:
  - i write to read, write to write, read to read and read to write
  - ii read other’s, write early
  - iii read own, write early
- pertaining to (i) different or (ii) same memory locations
Consistency Models

Relevant Excerpt (cf. [13])

- **data consistency** as close as possible to sequential processes or with optimisation margins for high-latency memory

---

**weak**

- “limited to hardware-recognized synchronizing variables” [4]
  - implemented by operating system machine level programs
  - usually not provided by the instruction set architecture level

---
**Consistency Models**

- **data consistency** as close as possible to sequential processes or with optimisation margins for high-latency memory

**relaxed**
- in terms of the constraints defined by sequential consistency

**weak**
- “limited to hardware-recognized synchronizing variables” [4]

**state of the art processors** provide relaxed or weak consistency models
Outline

Preface

Primitive Instructions
  Atomic Operations

Memory Models
  Properties

Summary
Résumé

- Elementary operations at instruction architecture level
- Atomic load/store of a naturally aligned machine (double-) word
- Atomic read-modify-write of complex machine instructions
- TAS, CAS and FAA or FA* for CISC and LL/SC for RISC
- Equality of atomic operations as to their consensus number (cf. p.41)

- Memory-access properties that are relevant to multi-threading
- Atomicity, visibility, and ordering of memory operations
- Memory architectures of type UMA, NUMA, and COMA
- Dynamic ordering at instruction set architecture level
- Memory barriers or fences, resp., to enforce ordering properly
- Sequential, relaxed, and weak data consistency

- Hardware features that are of importance for the implementation of any synchronisation algorithm including but not limited to non-blocking synchronisation, especially...
Résumé

- **elementary operations** at instruction structure set architecture level
  - atomic load/store of a naturally aligned machine (double-) word
  - atomic read-modify-write of complex machine instructions
    - TAS, CAS and FAA or FΑΦ, resp., for CISC and LL/SC for RISC
  - equality of atomic operations as to their *consensus number* (cf. p. 41)
memory-access properties that are relevant to multi-threading

- atomicity, visibility, and ordering of memory operations
- memory architectures of type UMA, NUMA, and COMA
- dynamic ordering at instruction set architecture level
- memory barriers or fences, resp., to enforce ordering proper
- sequential, relaxed, and weak data consistency
Résumé

- **hardware features** that are of importance for the implementation of any synchronisation algorithm
  - including but not limited to non-blocking synchronisation, especially
Résumé

- **elementary operations** at instruction structure set architecture level

- **memory-access properties** that are relevant to multi-threading

- **hardware features**


In: ACM Transactions on Programming Languages and Systems 11 (1991), Jan., Nr. 1, S. 124–149


[18] **Wikipedia:**  
*Test-and-Set.*  
Definition (Dual-Ported RAM)

A kind of random access memory (RAM) that supports simultaneous load and store operations from two directions.
Definition (Dual-Ported RAM)

A kind of random access memory (RAM) that supports simultaneous load and store operations from two directions.

The interlock is conducted by a “DPRAM monitor” that, e.g. [18]:

- records the processor that issued the TAS and acquired access
- notifies processors that, at a time, issue a TAS simultaneously
  - signalling BUSY interrupt, forcing the receiving processor into busy waiting
- performs the test and then, if and only if the test succeeds:
  i. sets the memory location to the value given by the owning processor and
  ii. releases access to that memory location
Definition (Dual-Ported RAM)

A kind of random access memory (RAM) that supports simultaneous load and store operations from two directions.

- the interlock is conducted by a “DPRAM monitor”

- this scheme translates into a conditional store as follows:

```c
word tas(word *ref) {
    word aux;
    atomic { if ((aux = *ref) == 0) *ref = 1; }
    return aux;
}
```
given a LIFO list (i.e., stack) of following structure: head ⇓ A ⇓ B ⇓ C
with head stored at location \( L_i \) shared by processes \( P_1 \) and \( P_2 \)
- push (cf. [16, p.11]) and pull adding or removing, resp., list items:

```c
chain_t *cas_pull(stack_t *this) {
    chain_t *node;
    do if ((node = this->head.link) == 0) break;
    while (!CAS(&this->head.link, node, node->link));
    return node;
}
```

assuming that the following sequence of actions will take place:
- \( P_1 \) reads head item \( A \) followed by \( B \) on the list, gets delayed at line 4
- remembers \( \text{node} = A \), but has not yet done CAS:
- \( \text{head} = A \ B \ C \)
- \( P_2 \) pulls head item \( A \) from the list:
- \( \text{head} = B \ C \)
- pulls head item \( B \) from the list:
- \( \text{head} = C \)
- pushes item \( A \) back to the list, now followed by \( C \):
- \( \text{head} = A \ C \)
- \( P_1 \) resumes, CAS realises \( \text{head} = A \) (followed by \( B \)):
- \( \text{list state} \:\text{head} = A \ C \) as left behind by \( P_2 \) is lost...

© wosch, thoenig  
CS (WS 2020/21, LEC 5)  
Addendum – Compare & Swap
given a LIFO list (i.e., stack) of following structure: \( \text{head} \triangleright A \triangleright B \triangleright C \)
- with \( \text{head} \) stored at location \( L \), shared by processes \( P_1 \) and \( P_2 \)
- \( \text{push} \) (cf. [16, p. 11]) and \( \text{pull} \) adding or removing, resp., list items:

```c
chain_t * cas_pull (stack_t * this) {
    chain_t * node;
    do if ((node = this->head.link) == 0) break;
    while (!CAS(&this->head.link, node, node->link));
    return node;
}
```

assuming that the following sequence of actions will take place:
- \( P_1 \) reads head item \( A \) followed by \( B \) on the list, gets delayed at line 4
  - remembers \( node = A \), but has not yet done CAS: \( \text{head} \triangleright A \triangleright B \triangleright C \)
- \( P_2 \) pulls head item \( A \) from the list:
  - \( \text{head} \triangleright B \triangleright C \)
  - pulls head item \( B \) from the list:
  - \( \text{head} \triangleright C \)
  - pushes item \( A \) back to the list, now followed by \( C \): \( \text{head} \triangleright A \triangleright C \)
- \( P_1 \) resumes, \( \text{CAS} \) realises \( \text{head} = A \) (followed by \( B \)): \( \text{head} \triangleright B \triangleright \) ☹️
  - list state \( \text{head} \triangleright A \triangleright C \) as left behind by \( P_2 \) is lost...
prevalent approach is to add a **change number** to the “control word” [8, p.125], i.e., to practice some kind of **versioning**

- this number increments at each CAS attempt on the control word
- appropriate techniques depend on the change-number parameters
prevalent approach is to add a **change number** to the “control word” [8, p. 125], i.e., to practice some kind of **versioning**

- this number increments at each CAS attempt on the control word

appropriate techniques depend on the change-number parameters

a. the values margin has a whole word size available
   - both the control and change-number word must be updated, indivisibly
   - *compare double and swap* (CDS, [8, p. 124]) of two consecutive words
   - *double compare and swap* (DCAS, also CAS2 [14, p. 4-66]) of any two words

---

*See also cmpxchg8b or cmpxchg16b, in case of x86.*
prevalent approach is to add a **change number** to the “control word” [8, p. 125], i.e., to practice some kind of **versioning**

- this number increments at each CAS attempt on the control word

appropriate techniques depend on the change-number parameters

a. the values margin has a whole word size available
   - both the control and change-number word must be updated, indivisibly
   - *compare double and swap* (CDS, [8, p. 124]) of two consecutive words
   - *double compare and swap* (DCAS, also CAS2 [14, p. 4-66]) of any two words

b. the values margin utilizes fully unused bits in the control word itself
   - CAS facilitates indivisible updates of control word including change number
   - workaround, especially suitable for handling aligned data-structure **pointers**
   - gimmick is in data-structure padding for an object size of a power of two

---

9See also *cmpxchg8b* or *cmpxchg16b*, in case of x86.
prevalent approach is to add a **change number** to the “control word” [8, p.125], i.e., to practice some kind of **versioning**

- this number increments at each CAS attempt on the control word

appropriate techniques depend on the change-number parameters

a. the values margin has a whole word size available
   - both the control and change-number word must be updated, indivisibly
   - *compare double and swap* (CDS, [8, p.124]) of two consecutive words\(^9\)
   - *double compare and swap* (DCAS, also CAS2 [14, p.4-66]) of any two words

b. the values margin utilizes fully unused bits in the control word itself
   - CAS facilitates indivisible updates of control word including change number
   - workaround, especially suitable for handling aligned data-structure **pointers**
   - gimmick is in data-structure padding for an object size of a power of two

\(\Rightarrow\) an object size of \(2^n\) bytes then gives \(n - 1\) low-order bits always 0

\(\Rightarrow\) these \(n - 1\) low-order bits then will be used as a **change-number tag**

\(\Rightarrow\) for **pointer operations**, the change-number tag is temporary neutralised

\(^9\) See also cmpxchg8b or cmpxchg16b, in case of x86.
prevalent approach is to add a **change number** to the “control word” [8, p. 125], i.e., to practice some kind of **versioning**
  - this number increments at each CAS attempt on the control word
  
appropriate techniques depend on the change-number parameters
  a. the values margin has a whole word size available
    - both the control and change-number word must be updated, indivisibly
    - *compare double and swap* (CDS, [8, p. 124]) of two consecutive words
    - *double compare and swap* (DCAS, also CAS2 [14, p. 4-66]) of any two words
  b. the values margin utilizes fully unused bits in the control word itself
    - CAS facilitates indivisible updates of control word including change number
    - workaround, especially suitable for handling aligned data-structure **pointers**
    - gimmick is in data-structure padding for an object size of a power of two
      
        ↦ an object size of $2^n$ bytes then gives $n - 1$ low-order bits always 0
        ↦ these $n - 1$ low-order bits then will be used as a **change-number tag**
        ↦ for **pointer operations**, the change-number tag is temporary neutralised

but the ABA problem never disappears, it only gets more improbable

---

9 See also *cmpxchg8b* or *cmpxchg16b*, in case of x86.
as GCC does not provide atomic built-in functions for this case:

```c
INLINE
long LL(long *ref) {
    long aux;

    asm volatile(
        "lwarx %0, 0, %1" : "=r" (aux) : "r" (ref));

    return aux;
}

INLINE
int SC(long *ref, long val) {
    long ccr;

    asm volatile(
        "stwcx. %2, 0, %1\n\t" : "=r" (ccr) : "r" (ref), "r" (val) : "cc", "memory");

    return ccr & 0x2;
}
```

with "\#define INLINE extern inline" for GCC to ensure that stand-alone object code is never emitted for in-line functions\(^{10}\)

\(^{10}\)Use "\#define INLINE inline" for C99, for the same reason.
#define FAA __sync_fetch_and_add

```c
int faa(int *p, int v) {
    return FAA(p, v);
}
```

```
faa:
    movl 4(%esp), %ecx
    movl 8(%esp), %eax
    lock
    xaddl %eax, (%ecx)
    ret
```
```c
#define FAA __sync_fetch_and_add

int faa(int *p, int v) {
    return FAA(p, v);
}

#define AAF __sync_add_and_fetch

int aaf(int *p, int v) {
    return AAF(p, v);
}
```

© wosch, thoenig  CS (WS 2020/21, LEC 5)  Addendum – Fetch & Add
Noninvertible Operation

*fetch-and-Φ*, with \( \Phi = \text{max} \)

Safe-load of global variable \( G \) and conditional-store of \( \text{max}(G, L) \) at \( G \)

```c
word fax(word *ref, word val) {
    word aux;
    atomic { if ((aux = *ref) < val) *ref = val; }
    return aux;
}
```

Assuming that \( G = 42 \) and \( L = 4711 \):

\[
\text{XAF}(G, L) \equiv \text{max}(\text{FAX}(G, L), L)
\]
both terms result in 4711

\[
\text{FAX}(G, L) \equiv \text{max}(\text{XAF}(G, L), L)
\]
\( \text{FAX} \) may result in 42
Noninvertible Operation \textit{\textit{fetch-and-}}\Phi, with \Phi = \text{\textit{max}}

- safe-load of global variable $G$ and conditional-store of $\text{max}(G, L)$ at $G$

```c
word fax(word *ref, word val) {
    word aux;
    atomic { if ((aux = *ref) < val) *ref = val; }
    return aux;
}
```

- conditional-store of $\text{max}(G, L)$ at $G$ and return of $\text{max}(G, L)$

```c
word xaf(word *ref, word val) {
    atomic { word aux = (*ref > val) ? *ref : *ref = val; }
    return aux;
}
```
Noninvertible Operation

fetch-and-Φ, with Φ = max

- safe-load of global variable G and conditional-store of \( \max(G, L) \) at G

```c
word fax(word *ref, word val) {
    word aux;
    atomic { if (((aux = *ref) < val) *ref = val; } 
    return aux;
}
```

- conditional-store of \( \max(G, L) \) at G and return of \( \max(G, L) \)

```c
word xaf(word *ref, word val) {
    atomic { word aux = (*ref > val) ? *ref : *ref = val; } 
    return aux;
}
```

- assuming that \( G = 42 \) and \( L = 4711 \):
  - \( XAF(G, L) \equiv \max(FAX(G, L), L) \): both terms result in 4711
  - \( FAX(G, L) \not\equiv \max(XAF(G, L), L) \): FAX may result in \( 42 < 4711 \)
operations that need consensus number $n$ cannot have a semantically equivalent implementation by operations of consensus number $m < n$

**Definition (Consensus Number)**

\[
\text{The consensus number for } X \text{ is the largest } n \text{ for which } X \text{ solves } n\text{-process consensus. If no largest } n \text{ exists, the consensus number is said to be infinite.} \ [7, \text{p. 130}]
\]

- $n$ processes need to interact to achieve agreement on a single data value
- note that only 1-process consensus requires no interaction
Equality of Atomic Operations

- operations that need consensus number $n$ cannot have a semantically equivalent implementation by operations of consensus number $m < n$

**Definition (Consensus Number)**

*The consensus number for $X$ is the largest $n$ for which $X$ solves $n$-process consensus. If no largest $n$ exists, the consensus number is said to be infinite.* [7, p. 130]

- consensus numbers of the elementary operations considered:
  - $\infty$ ■ compare-and-swap, load-linked/store-conditional
  - 2 ■ test-and-set, swap, fetch-and-add
  - 1 ■ atomic read, atomic write
Equality of Atomic Operations

- operations that need consensus number $n$ cannot have a semantically equivalent implementation by operations of consensus number $m < n$

**Definition (Consensus Number)**

*The consensus number for $X$ is the largest $n$ for which $X$ solves $n$-process consensus. If no largest $n$ exists, the consensus number is said to be infinite.* [7, p. 130]

- key point is the **progress guarantee** a certain operation has to give
  - for wait-freedom [7], the operation must have consensus number $n = \infty$
  - in that case, every action has guarantee to complete in finite steps/time