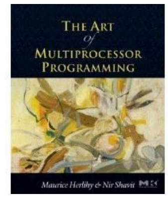
Department of Computer Science

COS 226

Chapter 4
Foundations of Shared Memory

Acknowledgement

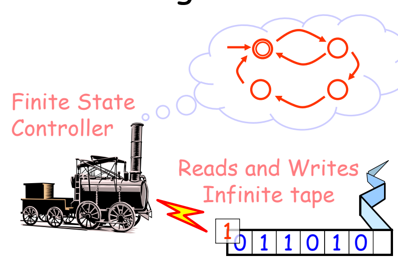


- Some of the slides are taken from the companion slides for "The Art of Multiprocessor Programming" by Maurice Herlihy & Nir Shavit

Church-Turing Thesis

- Anything that can be computed, can be computed by a Turing Machine.
- Foundations of sequential computing

Turing Machine



Concurrent shared-memory computing

- Consists of multiple threads – each a sequential program
- That communicate by calling methods of objects in shared memory

Threads

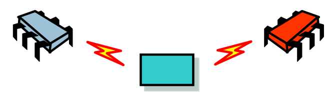
- Threads are asynchronous
  - They run at different speeds and can be halted for an unpredictable duration at any time

Shared-Memory Computability?

- Mathematical model of concurrent computation
- What is (and is not) concurrently computable
- Efficiency (mostly) irrelevant

Foundations of Shared Memory

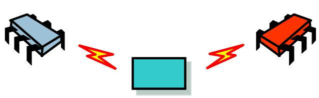
To understand modern multiprocessors we need to ask some basic questions …



8

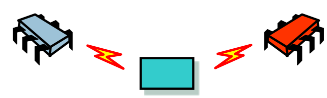
Foundations of Shared Memory

What is the weakest useful form of shared memory?



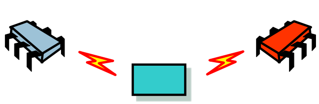
Foundations of Shared Memory

What can it do?



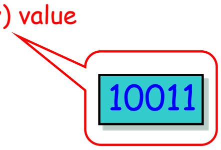
Foundations of Shared Memory

What can't it do?



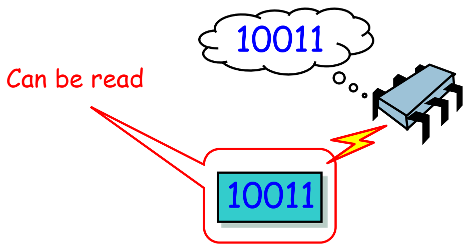
Register *

Holds a (binary) value

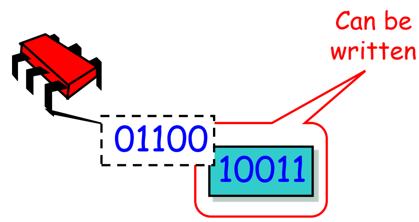


* A memory location: name is historical

Register



Register



Registers

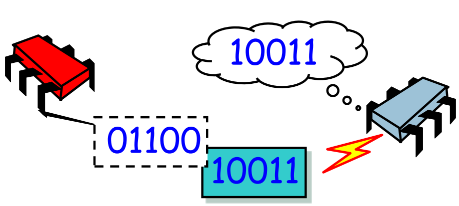
```
public interface Register<T> {
  public T read();
  public void write(T v);
}
```

Registers

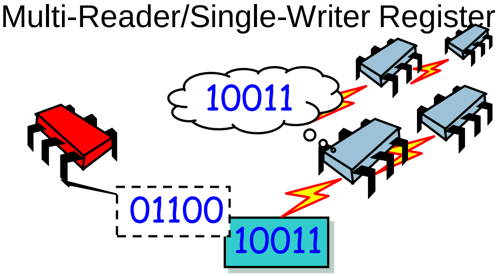
```
public interface Register<T> {
  public T read();
  public void write(T v);
}
```

Type of register (usually Boolean or m-bit Integer)

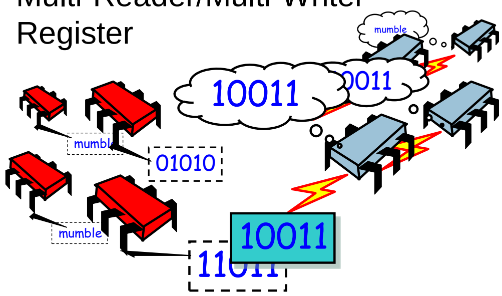
Single-Reader/Single-Writer Register



Multi-Reader/Single-Writer Register



Multi-Reader/Multi-Writer Register



Jargon Watch

- SRSW
  - Single-reader single-writer
- MRSW
  - Multi-reader single-writer
- MRMW
  - Multi-reader multi-writer

Concurrent registers

- On a multiprocessor, we expect reads and writes to overlap
- How do we specify what a concurrent method call mean?

One approach

- Rely on mutual exclusion:
  - Protect each register with a mutex lock acquired by each `read()` and `write()` call
  - Possible problems?

Different approach: Wait-Free Implementation

Definition: An object implementation is ***wait-free*** if every method call completes in a finite number of steps

- *No mutual exclusion*
- *Guarantees independent progress*

- *We require register implementations to be wait-free*

Different kinds of registers

- According to:
  - Range of values
    - Boolean or Integer (M-valued)
  - Number of readers and writers
  - Degree of consistency

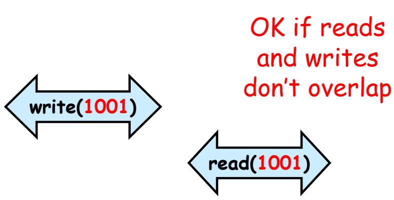
Degree of consistency

- Safe
- Regular
- Atomic

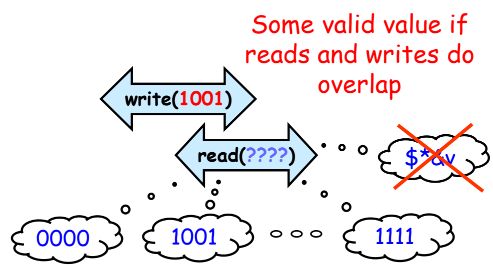
Safe Register

- A single-writer, multi-reader register is safe if:
  - A read() that does not overlap a write() return the last value
  - If a read() overlaps a write() it can return any value within the register's range

Safe Register



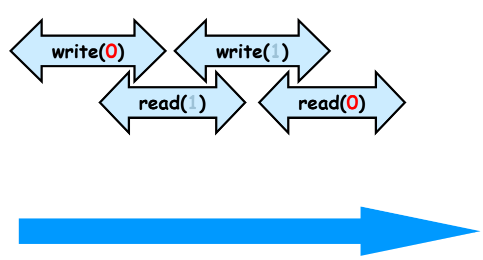
Safe Register



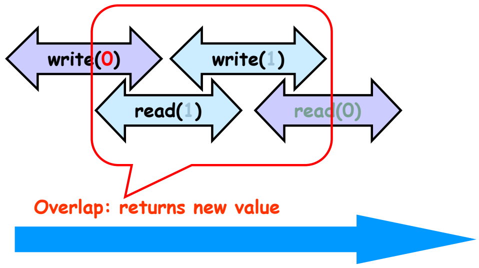
Regular register

- A single-writer, multi-reader register is regular if:
  - A read() that does not overlap a write() returns the last value
  - If a read() overlaps a write() it returns either the old value or the new value
    - Value being read may "flicker" between the old and new value before finally changing to the new value

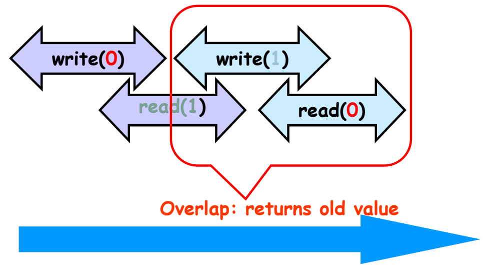
Regular or Not?



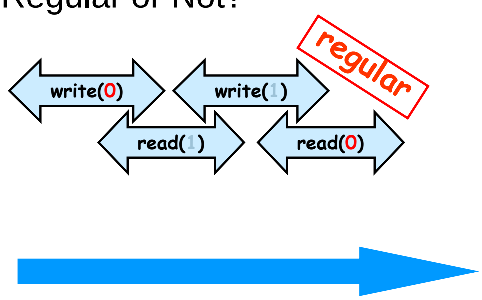
Regular or Not?



Regular or Not?



Regular or Not?

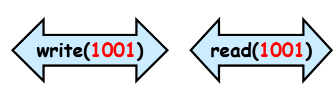


Regular ≠ Linearizable

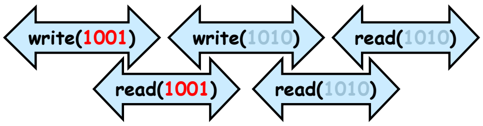
Atomic register

- Linearizable implementation of sequential register
- A single-writer, multi-reader register is atomic if:
  - Each read() returns the last value written

Sequential Register

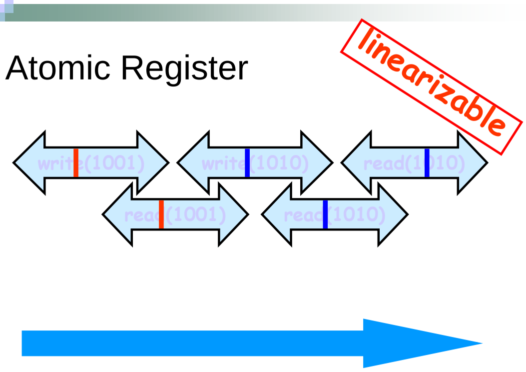


Atomic Register

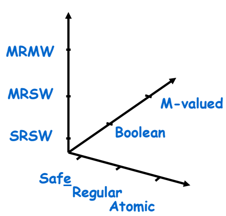


Linearizable?

Atomic Register



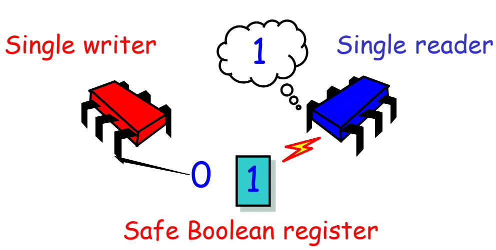
Register Space



Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic

Weakest Register



Register construction

- We will now build a range of registers from single-reader, single-writer Boolean safe registers

Road Map

- SRSW safe Boolean
- MRSW safe Boolean ← **Next**
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic

Register Names

```
public class SafeBoolMRSWRegister
 implements Register<Boolean> {
  public boolean read() { … }
  public void write(boolean x) { … }
}
```

Register Names

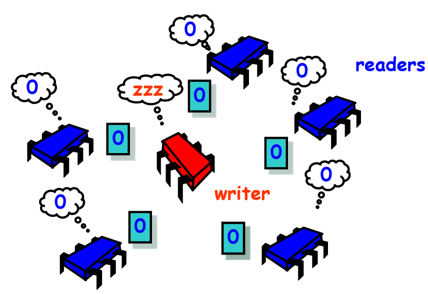
```
public class SafeBoolMRSWRegister
 implements Register<Boolean> {
  public boolean read() { … }
  public void write(boolean x) { … }
}
```

property

type

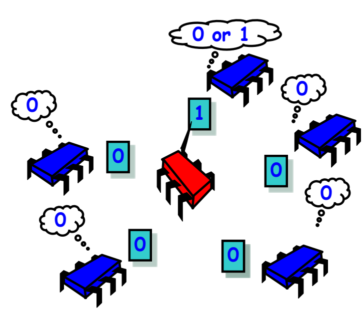
How many readers & writers?

Safe Boolean MRSW

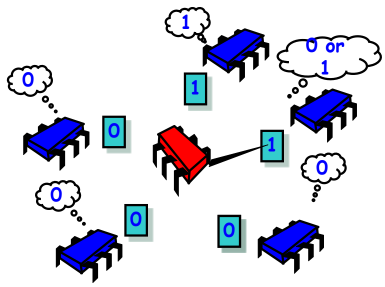


Safe Boolean MRSW

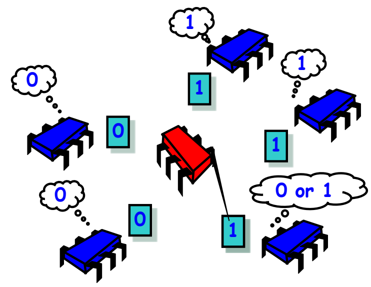
Safe Boolean MRSW



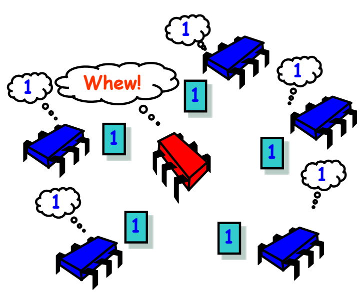
Safe Boolean MRSW



Safe Boolean MRSW



Safe Boolean MRSW



Safe Boolean MRSW

```
public class SafeBoolMRSWRegister implements
  Register<boolean> {

boolean[] s_table; //array of SRSW registers

  public SafeBoolMRSWRegister(int capacity) {
     s_table = new boolean[capacity];
  }
  public boolean read() {
     return s_table[ThreadID.get()];
  }
  public void write(boolean x) {
     for (int i = 0; I < s_table.length; i++)
          s_table[i] = x;
  }
}
```

Each thread has own safe SRSW register

Safe Boolean MRSW

```
public class SafeBoolMRSWRegister implements
  Register<boolean> {

boolean[] s_table; //array of SRSW registers

  public SafeBoolMRSWRegister(int capacity) {
     s_table = new boolean[capacity];
  }
  public boolean read() {
     return s_table[ThreadID.get()];
  }
  public void write(boolean x) {
     for (int i = 0; I < s_table.length; i++)
          s_table[i] = x;
  }
}
```

Write each thread's register one at a time

Safe Boolean MRSW

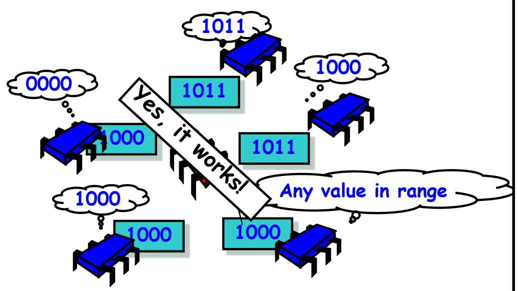
```
public class SafeBoolMRSWRegister implements
  Register<boolean> {

boolean[] s_table; //array of SRSW registers

  public SafeBoolMRSWRegister(int capacity) {
     s_table = new boolean[capacity];
  }
  public boolean read() {
     return s_table[ThreadID.get()];
  }
  public void write(boolean x) {
     for (int i = 0; I < s_table.length; i++)
          s_table[i] = x;
  }
}
```

Each thread reads own register

Safe Multi-Valued MRSW?



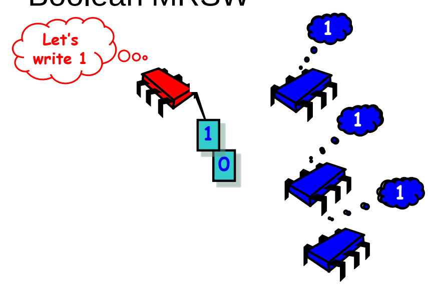
Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean ← Next
- MRSW regular
- MRSW atomic
- MRMW atomic

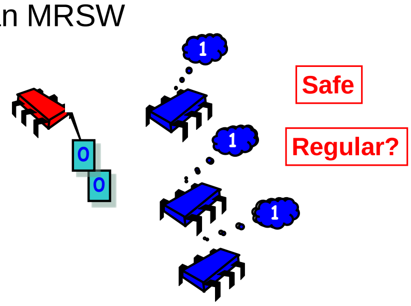
Safe BooleanMRSW vs Regular BooleanMRSW

- Only difference is when newly written value is same as old value:
  - Safe register can return either Boolean value
  - Regular register can return either new value or old value – if both new and old is x, then regular can only return x
  - So… write value only if distinct from previous written value

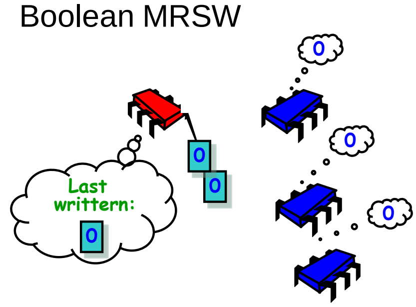
Safe Boolean MRSW → Regular Boolean MRSW



Safe Boolean MRSW → Regular Boolean MRSW



Safe Boolean MRSW → Regular Boolean MRSW



Safe Boolean MRSW → Regular Boolean MRSW

```
public class RegBoolMRSWRegister
 implements Register<Boolean> {
  private boolean old;
  private SafeBoolMRSWRegister value;
  public void write(boolean x) {
   if (old != x) {
    value.write(x);
    old = x;
   }}
  public boolean read() {
   return value.read();
  }}
```

Safe Boolean MRSW → Regular Boolean MRSW

```
public class RegBoolMRSWRegister
 implements Register<Boolean> {
  threadLocal boolean old;
  private SafeBoolMRSWRegister value;
  public void write(boolean x) {
   if (old != x) {
    value.write(x);
    old = x;
   }}
  public boolean read() {
   return value.read();
  }}
```

Last bit this thread wrote

(made-up syntax)

Safe Boolean MRSW →Regular Boolean MRSW

```
public class RegBoolMRSWRegister
 implements Register<Boolean> {
  threadLocal boolean old;
  private SafeBoolMRSWRegister value;
  public void write(boolean x) {
   if (old != x) {
    value.write(x);
    old = x;
   }}
  public boolean read() {
   return value.read();
  }}
```

Actual value

Safe Boolean MRSW → Regular Boolean MRSW

```
public class RegBoolMRSWRegister
 implements Register<Boolean> {
  threadLocal boolean old;
  private SafeBoolMRSWRegister value;
  public void write(boolean x) {
   if (old != x) {
    value.write(x);
    old = x;
   }}
  public boolean read() {
   return value.read();
  }}
```

Is new value different from last value I wrote?

Safe Boolean MRSW → Regular Boolean MRSW

```
public class RegBoolMRSWRegister
 implements Register<Boolean> {
  threadLocal boolean old;
  private SafeBoolMRSWRegister value;
  public void write(boolean x) {
   if (old != x) {
    value.write(x);
    old = x;
   }}
  public boolean read() {
   return value.read();
  }}
```

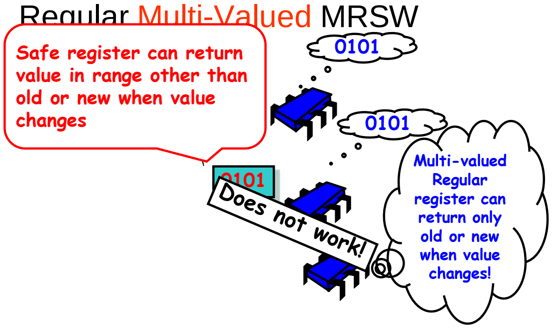
If so, change it (otherwise don't!)

Safe Boolean MRSW → Regular Boolean MRSW

```
public class RegBoolMRSWRegister
 implements Register<Boolean>{
  threadLocal boolean old;
  private SafeBoolMRSWRegister value;
  public void write(boolean x) {
   if (old != x) {
    value.write(x);
    old = x;
   }}
  public boolean read() {
   return value.read();
  }}
```

- Overlap? No Overlap?
  - No problem
  - either Boolean value works

Safe Multi-Valued MRSW → Regular Multi-Valued MRSW



Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular ⟵ Next
- MRSW atomic
- MRMW atomic

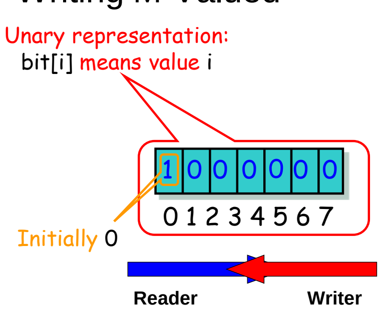
Regular M-Valued MRSW Register

- Values are represented using unary notation
- An M-valued register is implemented as an array of *m* regular MRSW Boolean registers
- Initially the register is set to 0

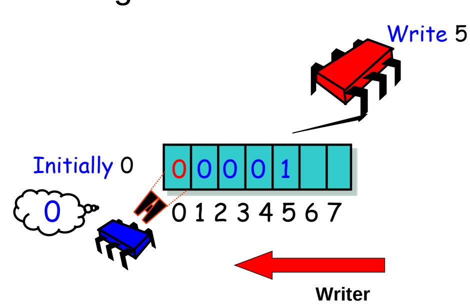
Regular M-Valued MRSW Register

- write():
  - A write() of value x, writes true to location x – which is a Regular Boolean MRSW register
  - It then sets all the lower locations to false
- read():
  - Reads the locations from lower to higher values until it reaches a value that is true

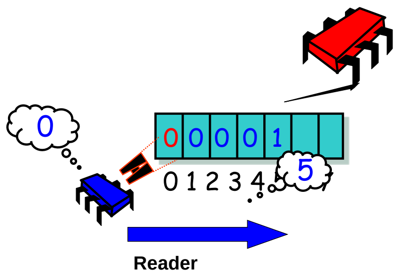
Writing M-Valued



Writing M-Valued



Writing M-Valued



MRSW Regular Boolean → MRSW Regular M-valued

```
public class RegMRSWRegister implements Register{
  RegBoolMRSWRegister[M] bit;

  public void write(int x) {
    this.bit[x].write(true);
    for (int i=x-1; i>=0; i--)
      this.bit[i].write(false);
  }

  public int read() {
    for (int i=0; i < M; i++)
      if (this.bit[i].read())
        return i;
   }}
```

MRSW Regular Boolean → MRSW Regular M-valued

```
public class RegMRSWRegister implements Register{

 RegBoolMRSWRegister[M] bit;

  public void write(int x) {
    this.bit[x].write(true);
    for (int i=x-1; i>=0; i--)
      this.bit[i].write(false);
  }

  public int read() {
    for (int i=0; i < M; i++)
      if (this.bit[i].read())
        return i;
   }}
```

Unary representation: bit[i] means value i

MRSW Regular Boolean → MRSW Regular M-valued

```
public class RegMRSWRegisterimplements Register {

RegBoolMRSWRegister[m] bit;

  public void write(int x) {
    this.bit[x].write(true);
    for (int i=x-1; i>=0; i--)
      this.bit[i].write(false);
  }

  public int read() {
    for (int i=0; i < M; i++)
      if (this.bit[i].read())
        return i;
   }}
```

Set bit x

MRSW Regular Boolean → MRSW Regular M-valued

```
public class RegMRSWRegisterimplements Register {

RegBoolMRSWRegister[m] bit;

  public void write(int x) {
    this.bit[x].write(true);
    for (int i=x-1; i>=0; i--)
      this.bit[i].write(false);
  }

  public int read() {
    for (int i=0; i < M; i++)
      if (this.bit[i].read())
        return i;
   }}
```

Clear bits from higher to lower

MRSW Regular Boolean → MRSW Regular M-valued

```
public class RegMRSWRegisterimplements Register {

RegBoolMRSWRegister[m] bit;

  public void write(int x) {
    this.bit[x].write(true);
    for (int i=x-1; i>=0; i--)
      this.bit[i].write(false);
  }

  public int read() {
    for (int i=0; i < M; i++)
      if (this.bit[i].read())
        return i;
   }}
```

Scan from lower to higher & return first bit set

Regular Register Conditions

- Further conditions for a register to be regular:
  - No read() call should return a value from the future
  - No read() call should return a value from the distant past – only the most recently written non-overlapping value must be returned

Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic ← Next
- MRMW atomic

Road Map (Slight Detour)

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular ← SRSW Atomic
- MRSW atomic
- MRMW atomic

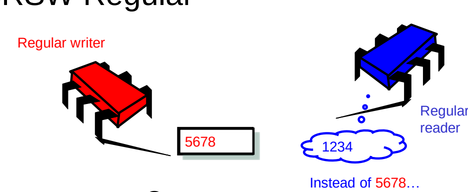
Atomic Register Conditions

- Together with the conditions for a regular register, an additional condition for an atomic register is:
  - An earlier read() cannot return a value later that that returned by a later read()
  - In other words, values read() should be in the correct order

SRSW register

- Since a SRSW register has no concurrent reads, the only way that the condition for an atomic register can be violated is when two reads that overlap the same write read values out of order

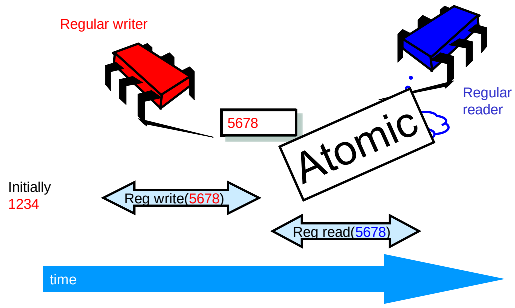
SRSW Regular



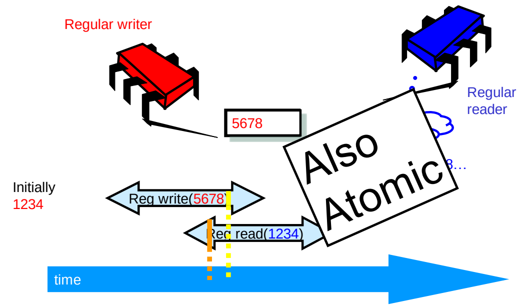
Concurrent Reading

When could this happen?

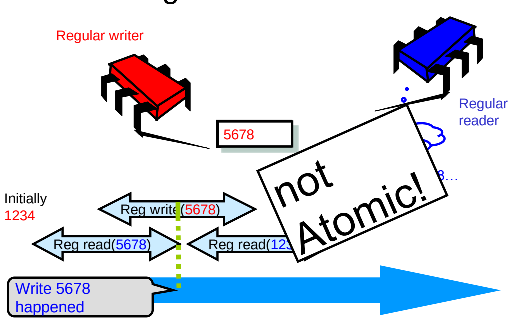
SRSW Regular



SRSW Regular



SRSW Regular



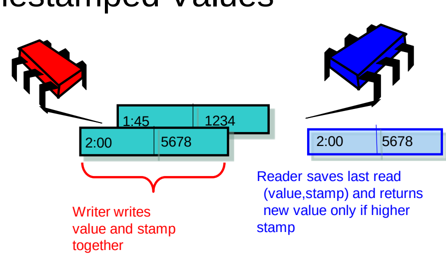
Timestamps

- Solution is to for each value to have an added tag – a timestamp
- Timestamps are used to order concurrent calls

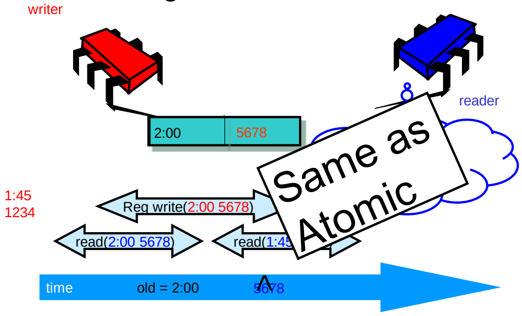
Timestamps

- The writer writes a timestamp to a value
- Each reader remembers the latest timestamp/value pair ever read
- If a later read() then returns an earlier value the value is discarded and the reader uses the last value

Timestamped Values



SRSW Regular → SRSW Atomic



Atomic SRSW

```
public class StampedValue<T> {
   public long stamp;
   public T value;
   public StampedValue (T init) {
        stamp = 0;
        value = init;
   }
   public StampedValue max (StampedValue x,StampedValue y)
   {
        if (x.stamp > y.stamp)
                return x;
        else return y;
   }
}
```

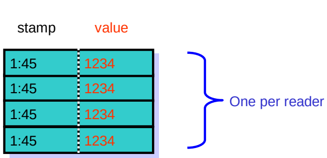
Atomic SRSW

```
public class AtomicSRSWRegister<T> implements Register<T> {
   long lastStamp;
   StampedValue<T> lastRead;
   StampedValue<T> value;
   public T read()  {
        StampedValue<T> result = StampedValue.max(value,
   lastRead);
        lastRead = result;
        return result.value;
   }
   public void write(T v) {
        long stamp = lastStamp + 1;
        value  = new StampedValue(stamp, v);
        lastStamp = stamp;
   }
}
```

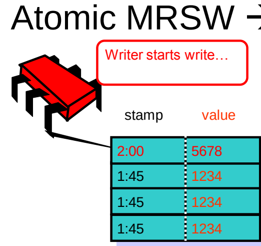
Atomic SRSW → Atomic MRSW

- Can the atomic SRSW be used to built an atomic MRSW?
- Solution of Safe MRSW Registers:
  - Every thread in array
  - Write starts at the beginning of the array and iterates through array
  - Read reads only its own array location

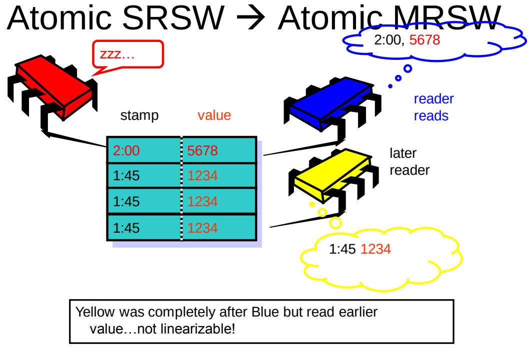
Atomic Single Reader → Atomic Multi-Reader



Atomic MRSW → Atomic SRSW



Atomic SRSW → Atomic MRSW

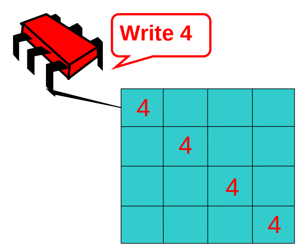


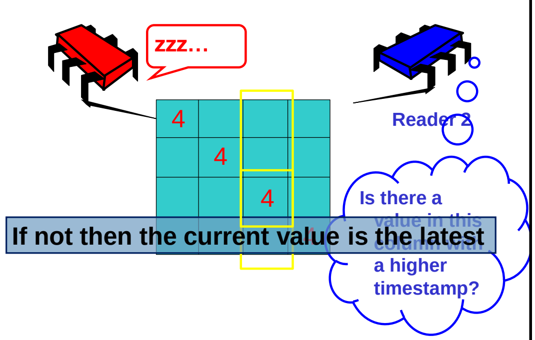
Atomic MRSW

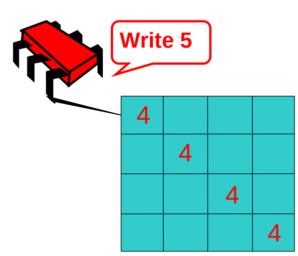
- We address this problem by having earlier reader threads help out later threads, by telling them which value they read

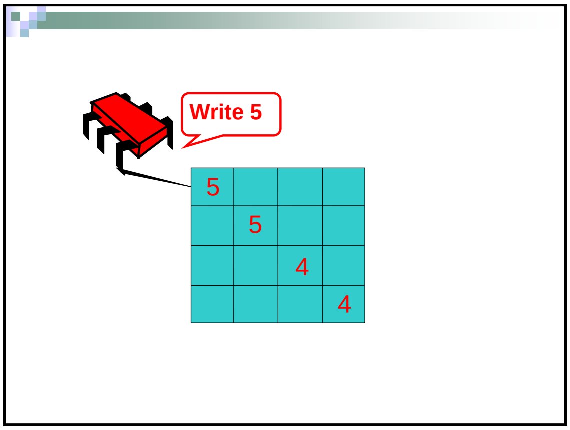
Atomic MRSW

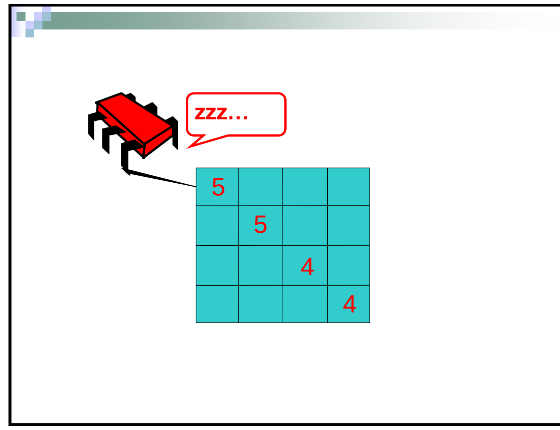
- n-threads share a n-by-n array of stamped values
- Read() calls determine latest threads by timestamps
- Similar to the Safe MRSW Register implementation, the writer writes the new values to the array, but only on the diagonals

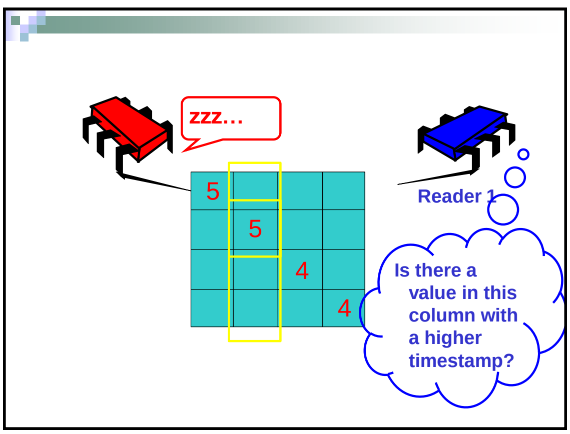


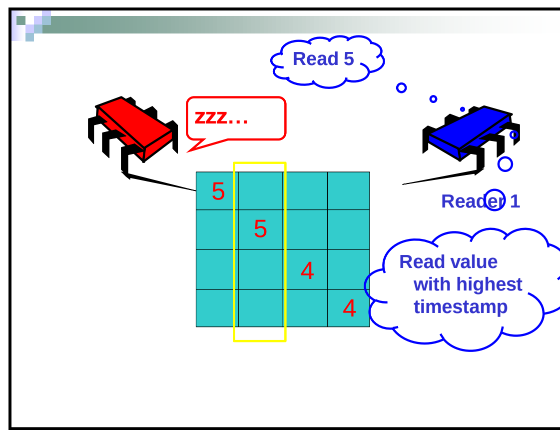


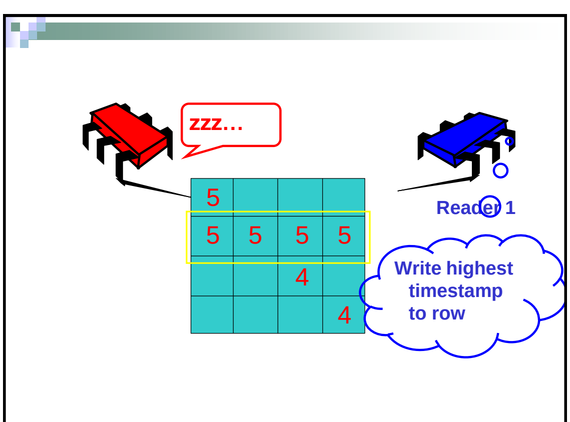


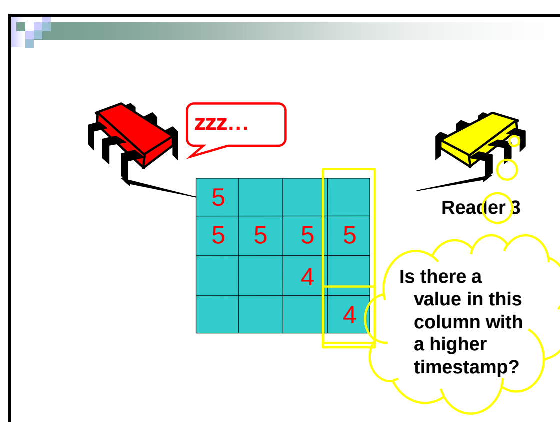

Write 5
5
5
4
4

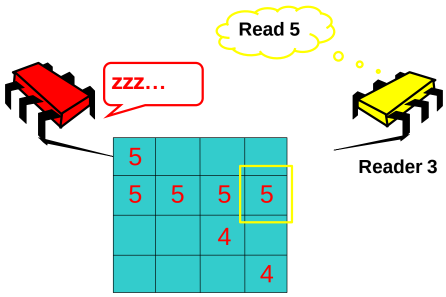

zzz…
5
5
4
4


zzz…
5
5
4
4
Reader 1
Is there a value in this column with a higher timestamp?


Read 5
zzz…
5
5
4
4
Reader 1
Read value with highest timestamp


zzz…
5
5 5 5 5
4
4
Reader 1
Write highest timestamp to row


zzz…
5
5 5 5 5
4
4
Reader 3
Is there a value in this column with a higher timestamp?



MRSW Atomic

```
public class AtomicMRSWRegister implements Register{
   long lastStamp;
   StampedValue<T>[][] a_table;

   public T read()  {
       int me = ThreadID.get();
       StampedValue<T> value = a_table[me][me];
       for (int i = 0; i < n; i++)
              value = StampedValue.max(value,
              a_table[i][me];
       for (int i = 0; i < n; i++)
              a_table[me][i] = value;
   }
```

Matrix of StampedValues

MRSW Atomic

```
public class AtomicMRSWRegister implements Register{
   long lastStamp;
   StampedValue<T>[][] a_table;

   public T read()  {
       int me = ThreadID.get();
       StampedValue<T> value = a_table[me][me];
       for (int i = 0; i < n; i++)
              value = StampedValue.max(value,
              a_table[i][me];
       for (int i = 0; i < n; i++)
              a_table[me][i] = value;
   }
```

Check column for maximum

MRSW Atomic

```
public class AtomicMRSWRegister implements Register{
   long lastStamp;
   StampedValue<T>[][] a_table;

   public T read()  {
       int me = ThreadID.get();
       StampedValue<T> value = a_table[me][me];
       for (int i = 0; i < n; i++)
              value = StampedValue.max(value,
              a_table[i][me];
       for (int i = 0; i < n; i++)
              a_table[me][i] = value;
   }
```

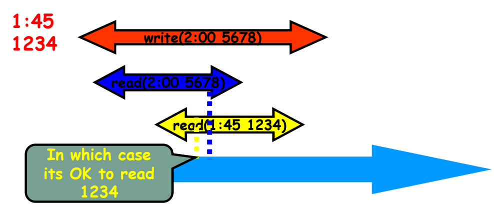
Write maximum to row

MRSW Atomic

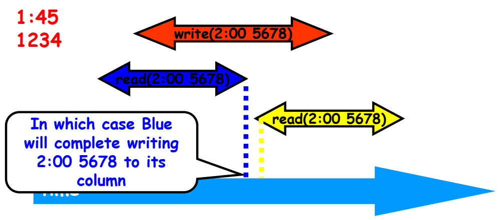
```
public void write(T v) {
   long stamp = lastStamp + 1;
   lastStamp = stamp;
   StampedValue<T> value = new StampedValue<T>(stamp,
   v);
 for (int i = 0; i < n; i++)
       a_table[i][i] = value;
}
```

Write to diagonal

Can’t Yellow Miss Blue’s Update? … Only if Readers Overlap…



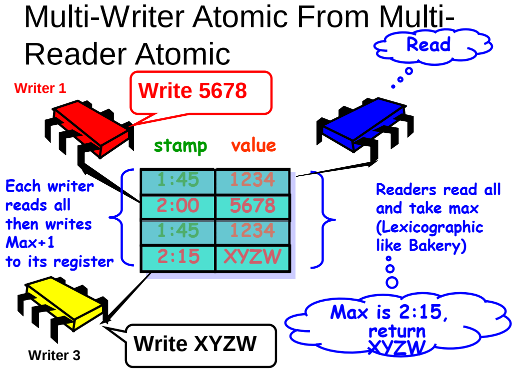
Bad Case Only When Readers Don't Overlap



Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic ⟵ Next

Multi-Writer Atomic From Multi-Reader Atomic



MRMW Atomic

```
public class AtomicMRMWRegister implements Register{
  StampedValue<T>[] a_table;

  public void write (T value)  {
      int me = ThreadID.get();
      StampedValue<T> max = StampedValue.MIN;
      for (int i = 0; i < n; i++)
          max = StampedValue.max(max,
          a_table[i]);
      a_table[me] = new StampedValue(max.stamp + 1,
      value);
  }
```

Array of StampedValues

MRMW Atomic

```
public class AtomicMRMWRegister implements Register{
  StampedValue<T>[] a_table;

  public void write (T value)  {
      int me = ThreadID.get();
      StampedValue<T> max = StampedValue.MIN;
      for (int i = 0; i < n; i++)
          max = StampedValue.max(max,
          a_table[i]);
      a_table[me] = new StampedValue(max.stamp + 1,
      value);
  }
```

Find highest timestamp

MRMW Atomic

```
public class AtomicMRMWRegister implements Register{
  StampedValue<T>[] a_table;

  public void write (T value)  {
      int me = ThreadID.get();
      StampedValue<T> max = StampedValue.MIN;
      for (int i = 0; i < n; i++)
          max = StampedValue.max(max,
          a_table[i]);
      a_table[me] = new StampedValue(max.stamp + 1,
      value);
  }
```

Write new value to array

MRMW Atomic

```
public T read() {
  StampedValue<T> max = StampedValue.MIN;
  for (int i = 0; i < n; i++)
      max = StampedValue.max(max, a_table[i]);
  return max.value;
}
```

Find highest timestamp

Conclusion

- One can construct a wait-free MRMW atomic register from SRSW Safe Boolean registers