Armv8-M Synchronization

Course 4 Advanced Armv8-M Features

Learning objectives

At the end of this module you will be able to:

- Explain the rational for Exclusive Access instructions
- Determine how to use the Exclusive Access instructions
- Analyze the need for a Global Exclusives Monitor in a system design
- Determine the need for memory barriers when using Exclusives

Agenda

- **Introduction to synchronization and semaphores**
- Exclusive accesses
- Memory ordering
- Appendix

The need for atomicity

- **Most Thumb instructions can be categorized as either a memory access (load/store) or a data processing operation**
- **Programs typically combine these basic instructions together into compound operations, frequently creating some form of read-modify-write sequence**
- **Often, this read-modify-write sequence needs to be atomic for the algorithm to work correctly**
  - The code which needs to be atomic is highlighted in this example

```
Increment
        LDR     r0, =shared
        LDR     r1, [r0]
        ADD     r1, #1
        STR     r1, [r0]
```

The race for atomicity

With single-threaded data access, atomicity is implicit

In a multi-threaded situation, data shared between threads are vulnerable

- Once a thread has read the data, there is a race to write back the modified data before any other thread accesses it
- Losing this race violates atomicity

Thread 1

```
Increment
        LDR     r0, =shared
        LDR     r1, [r0]
        ADD     r1, #1
        STR     r1, [r0]
```

Thread 2

```
Increment
        LDR     r0, =shared
        LDR     r1, [r0]
        ADD     r1, #1
        STR     r1, [r0]
```

In a single core system disabling interrupts can prevent a new thread being scheduled

- This may not be possible due to interrupt latencies

Critical sections

Any set of compound operations that needs to be atomic can be considered a critical section of code

- This becomes important if multiple threads have access to the same data

By claiming a mutex before entering critical sections, programs can protect any resource that is shared between threads

- Mutex: a token with MUTually EXclusive ownership

```
Increment
        BL      claim_mutex
        LDR     r0, =shared
        LDR     r1, [r0]
        ADD     r1, #1
        STR     r1, [r0]
        BL      release_mutex
```

Simple lock implementation

A LOCK variable can be stored in memory

- Only one process can hold the lock

To gain the lock we:

- Read the **`LOCK`** variable
- If it is unlocked we can write the **`LOCKED`** value back
- We now own the lock

```
lock
        ; Is locked?
        LDR     r0, =LOCK
        LDR     r1, [r0]
        CMP     r1, #LOCKED
        BEQ     lock
        ; Attempt to lock
        MOV     r1, #LOCKED
        STR     r1, [r0]
        BX     lr
```

- **This implementation is vulnerable to the `LOCK` variable being changed by another process**
  - In-between the **`LDR`** and the **`STR`**
- **Arm has primitives that can detect if the memory has been written to since it was read**

Agenda

- Introduction to synchronization and semaphores
- **Exclusive accesses**
- Memory ordering
- Appendix

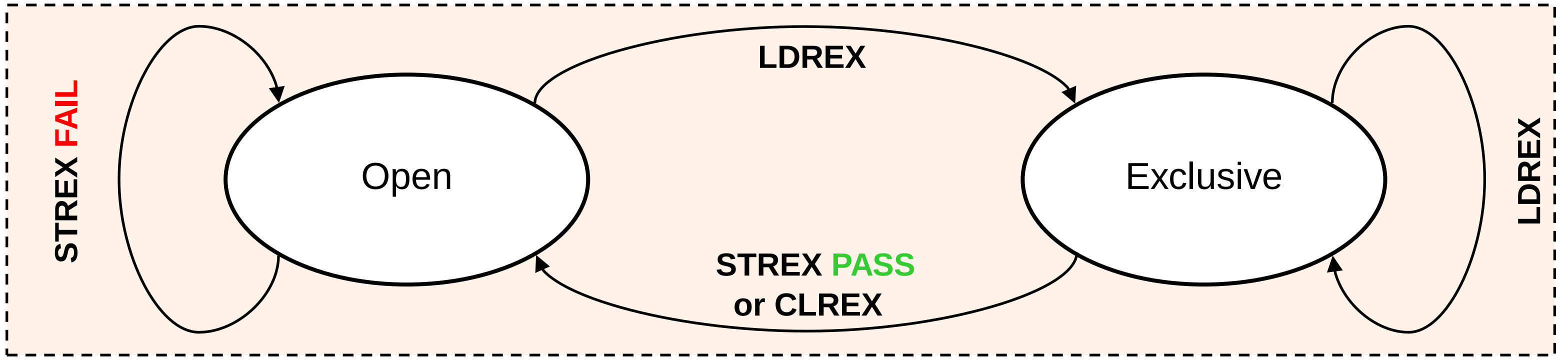
Effective atomicity

Armv8-M does not allow "enforced" atomicity

- No atomic compare and update instructions

Instead an Exclusive Monitor is used to monitor a location between the read and write

- The **`LDREX`** instruction causes the Exclusive Monitor to flag the accessed address as "exclusive"
- The **`STREX`** checks whether the Exclusive Monitor is still in the exclusive state
  - Store will only happen if exclusivity check passes



Simplified State Machine

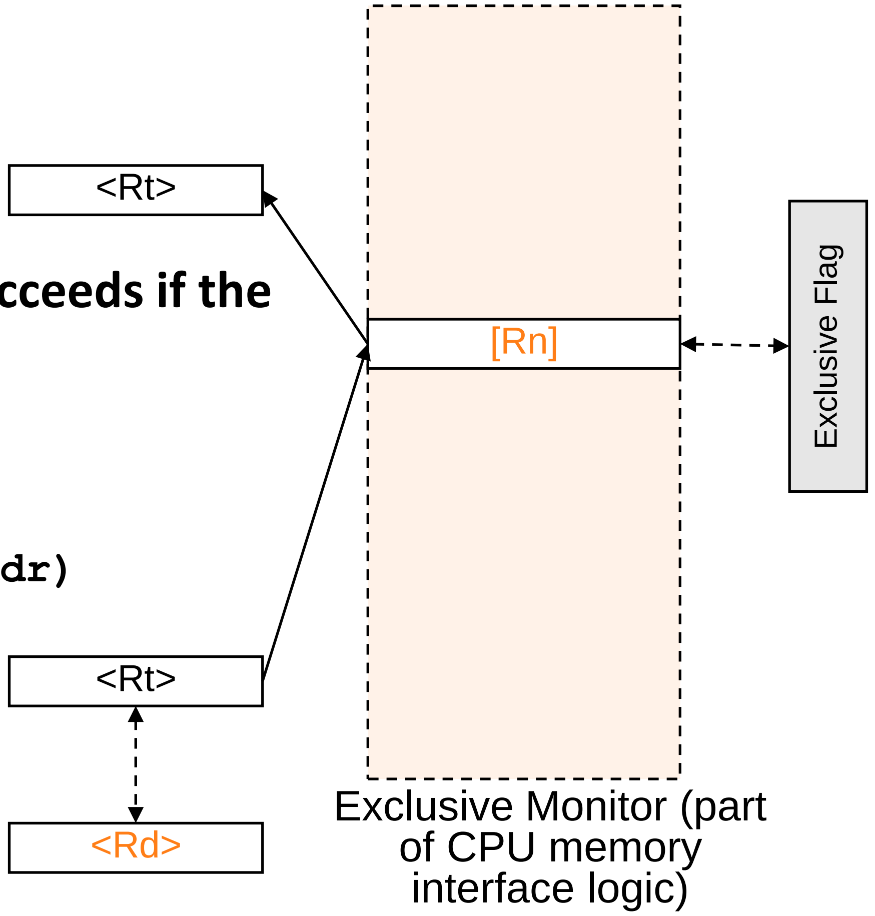
The Exclusive Monitor does NOT prevent another core/thread from reading or writing the monitored location

- It only monitors for whether the location has been written since the LDREX

The Exclusive Monitor is not a Mutex

LDREX and STREX instructions

- **LDREX (Load Exclusive) – performs a load and flags the physical address for exclusive access**
  - `LDREX <Rt>, [Rn]`
  - `uint32_t __LDREXW(volatile uint32_t *addr)`
- **STREX (Store Exclusive) – performs a conditional store which only succeeds if the address has previously been flagged for exclusive access**
  - `<Rd>` returns zero on success of the store operation
  - `STREX <Rd>, <Rt>, [Rn]`
  - `uint32_t __STREXW(uint32_t value, volatile uint32_t *addr)`
- **CLREX (Clear Exclusive) – clear exclusive flags associated with a processor**
  - `CLREX`
  - `__CLREX()`



Example: lock()

```
    ; void lock(lock_t* pAddr)
lock
    MOV     r3, #LOCKED
lockloop
    ; Is locked?
    LDREX   r1, [r0]            ; Check if locked
    CMP     r1, r3              ; Compare with "locked"
    BEQ     lockloop            ; If LOCKED, try again

    ; Attempt to lock
    STREX   r2, r3, [r0]        ; Attempt to lock
    CMP     r2, #0x0            ; Check whether store completed
    BNE     lockloop            ; If store failed, try again
    DMB
    BX      lr
```

Example: unlock()

```
 ; void unlock (lock_t* pAddr)
unlock
  DMB                                ; To ensure accesses to protected
                                     ; resource have completed
  MOV      r1, #UNLOCKED             ; Write "unlocked" into lock field
  STR      r1, [r0]

  BX       lr
```

Programs still have to be smart

A mutex is really just a flag

- Exclusive access allows atomic access to the flag
- Any thread or program that accesses the flag can know that it was set correctly

The actual resource that the mutex is protecting can still be accessed

- The mutex, and any exclusive accesses, are only related to the resource by what the code says
- If a program ignores the mutex, there’s nothing in the architecture stopping it from accessing that resource

Or, if a program doesn’t use the exclusive access instructions...

- There’s nothing remarkable or special about the memory used for mutexes
- Once the exclusive access is over, it’s just another piece of data in memory

Example: Multi-thread Mutex

Assume mutex is not locked (value of 0) when thread A attempts to reserve exclusive access

```
        lock = lock(adr);
    lock
        LDREX   r1, [r0]
        CMP     r1, #0
        <<INTERRUPT>>








        <<RESUME>>
        MOV     r1, #LOCKED
        ITTE    EQ
        STREXEQ r2, r1, [r0]
        CMPEQ   r2, #0
        BNE     lock
        DMB
        BX      lr
```

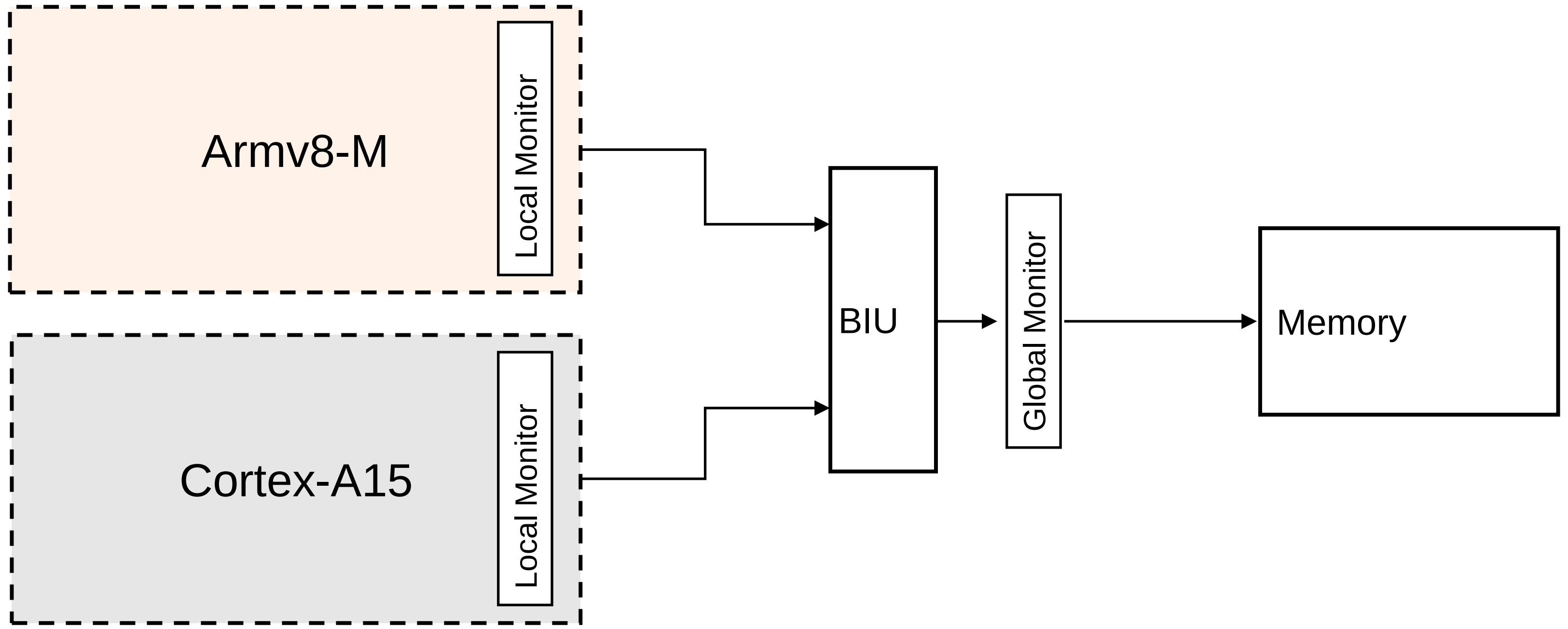
Get mutex value for thread A
Exclusive load of mutex location
Check if mutex is available
Interrupt occurs; thread A suspended
After ISR, thread B scheduled
Get mutex value for thread B
Exclusive load of mutex location
Check if mutex is available
Exclusive store of mutex value
Check if exclusive store successful
Store should be successful; no branch
Return and suspend thread B
Resume thread A
Exclusive store of mutex value
Check if exclusive store successful
Branch to retry loop (since store will not be successful)

```
        lock = lock(adr);
    lock
        LDREX   r1, [r0]
        CMP     r1, #0
        MOV     r1, #LOCKED
        ITTE    EQ
        STREXEQ r2, r1, [r0]
        CMPEQ   r2, #0
        BNE     lock
        DMB
        BX      lr
```

Non-coherent Multiprocessor

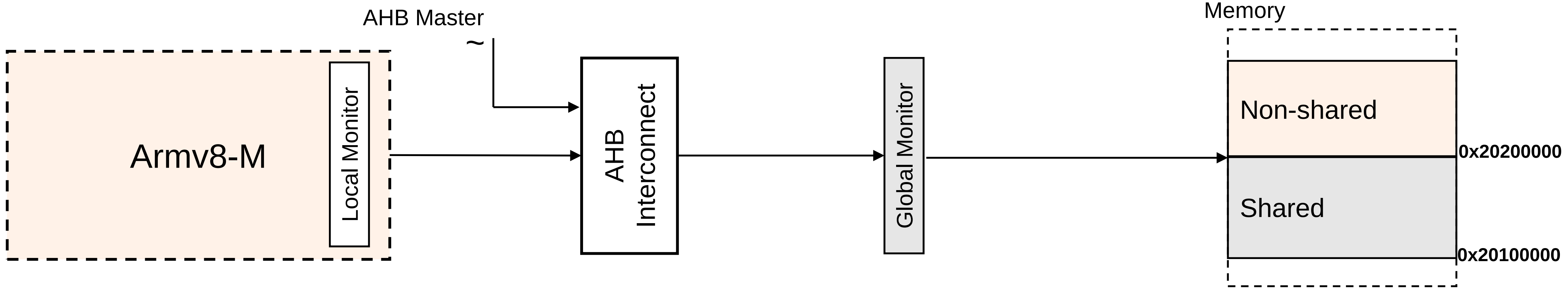
Some systems require both a Local and a Global monitor

- The Local monitor is used for resources shared by threads on one processor
- The Global and Local monitors are used for resources shared between multiple non-coherent processors



Memory attributes determine whether the Local and/or Global monitor observe the exclusive access

Memory attributes



The Local monitor observes memory accesses to address regions marked as non-shared and shared

- Example: `LDREX <Rt>, [0x20200000]`
- Local monitor observes the access and enters exclusive state
- Global monitor is unaffected by exclusive accesses to non-shared regions

The Global monitor will observe memory accesses to shared memory regions

- Example: `LDREX <Rt>, [0x20100000]`
- Global and Local monitor observe the access and enter exclusive state
- The outcome of `STREX` depends on both the Local and Global monitor

Configuring Sharable memory

Processors with an MPU can use an active region to control shareability

- Configured in the MPU Region Base Address Register (MPU_RBAR)

If the MPU is disabled or not present the default memory map is used

- All the default Normal memory regions are Non-sharable
- Would not usually use the Global Monitor

Cortex-M33 and Cortex-M23 can force Non-sharable memory accesses to use the Global Monitor

- Using the Auxiliary Control Register

When ACTLR.EXTEXCLALL=1 accesses related to exclusive loads and stores will use the Global Monitor

ACTLR is banked when the Security Extension is implemented

Context switching

It is necessary to ensure that the exclusive monitor is in the Open Access state after a context switch

- In Armv8-M, the local monitor is changed to Open Access state automatically as part of an exception entry or exit sequence

An instruction is also provided to clear the Exclusive Monitor

- **`CLREX`** (Clear Exclusive)
- Forces the local monitor into Open Access state
- It is implementation-defined whether the global monitor moves to Open Access state

Keep the `LDREX/STREX` pair close together for best performance

- Minimizes the likelihood of the exclusive monitor state being cleared between the **`LDREX`** instruction and the **`STREX`** instruction

Exclusives Reservation Granule

The Exclusives Reservation Granule (ERG) is the granularity of the Exclusive Monitor

- Minimum spacing between addresses for the monitor to distinguish between them
- Usually only a consideration on multiprocessor systems

ERG is implementation defined

- Valid global monitor ERGs are between 1 and 512 words in powers of two
- Local monitors do not store any address bits
  - Any exclusive access will match a previous Load-Exclusive

Placing two mutexes within one ERG can lead to false *negatives*

- An **`STREX`** to either mutex clears the exclusivity of both
- Architecturally-correct software will still function correctly – but may be less efficient

Example 1: Multiprocessor Mutex

r0 = 0x20080028

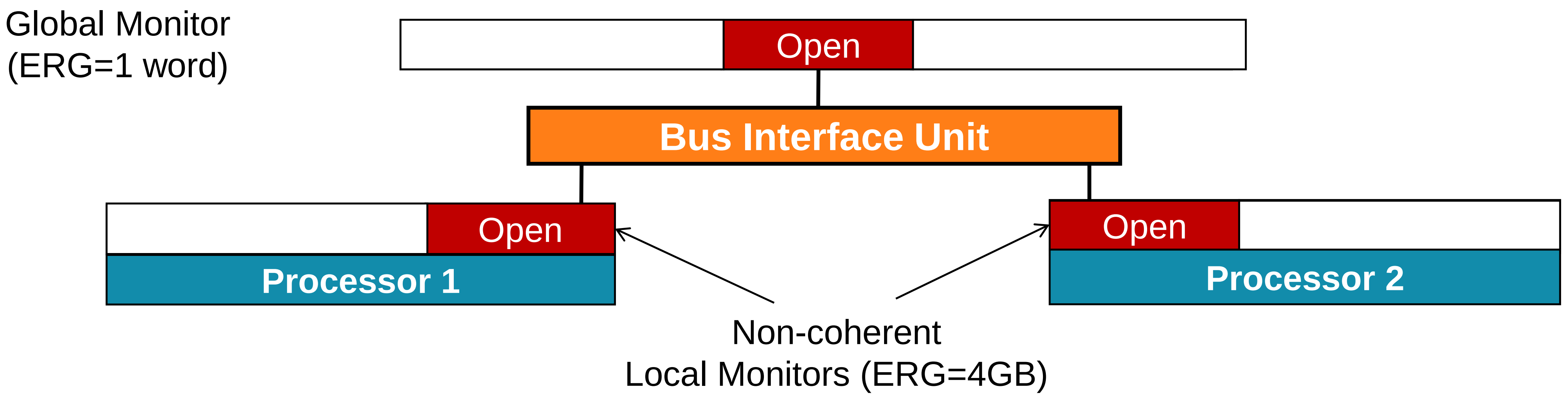
```
         MOV     r1, #1
loop:    LDREX   r2, [r0]
         CMP     r2, #0
         ITTE    EQ
         STREXEQ r2, r1, [r0]
         CMPEQ   r2, #0
         WFENE
         DMB
         BNE     loop
```

Success

r0 = 0x20080028

```
         MOV     r1, #1
loop:    LDREX   r2, [r0]
         CMP     r2, #0
         ITTE    EQ
         STREXEQ r2, r1, [r0]
         CMPEQ   r2, #0
         WFENE
         DMB
         BNE     loop
```

Failure


Global Monitor (ERG=1 word)
Open
Bus Interface Unit
Open
Processor 1
Open
Processor 2
Non-coherent Local Monitors (ERG=4GB)

Example 2: Multiprocessor Mutex

r0 = 0x200800fc

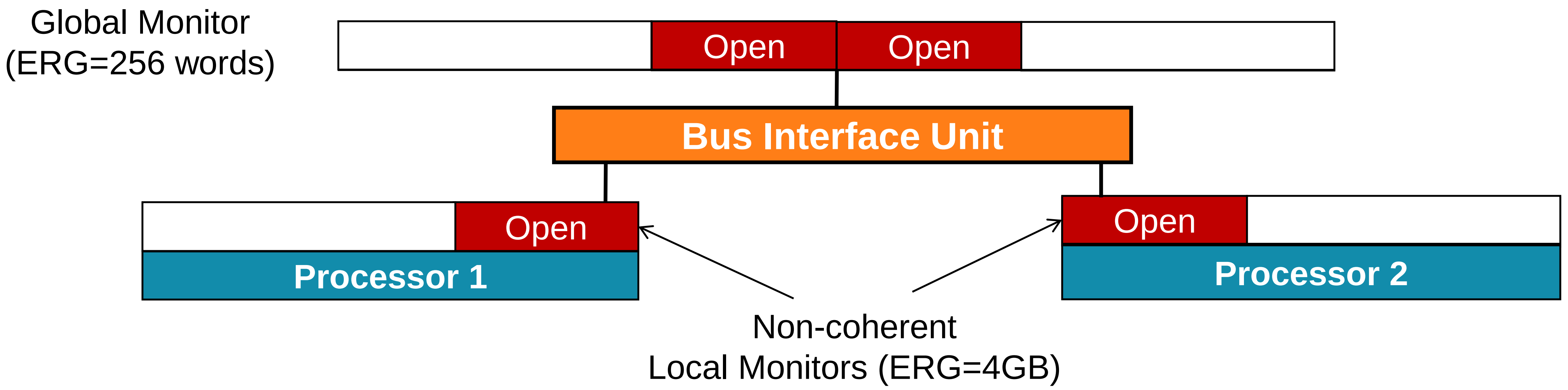
```
        MOV     r1, #1
loop:   LDREX   r2, [r0]
        CMP     r2, #0
        ITTE    EQ
        STREXEQ r2, r1, [r0]
        CMPEQ   r2, #0
        WFENE
        DMB
        BNE     loop
```

Success

r0 = 0x20080100

```
        MOV     r1, #1
loop:   LDREX   r2, [r0]
        CMP     r2, #0
        ITTE    EQ
        STREXEQ r2, r1, [r0]
        CMPEQ   r2, #0
        WFENE
        DMB
        BNE     loop
```

Success



Agenda

- Introduction to synchronization and semaphores
- Exclusive accesses
- **Memory ordering**
- Appendix

Weakly ordered memory and mutual exclusion

The Armv8-M architecture defines a weakly ordered memory model

- Memory accesses to normal memory can be seen out of order

When using mutexes to protect shared resources access order is important

- Claiming the mutex is a memory access!

```
/* Claim ownership of shared
memory buffer */

mutexWait(buffer_id);

/* Write data into buffer */

dataWrite(&buffer);

/*Release buffer */
mutexRelease(buffer_id);
```

Memory access!
Must occur before accessing the shared buffer

Memory access!
Must occur after getting the mutex but before releasing it

Memory access!
Must happen after accessing the buffer

Ordering with DMB

To enforce ordering mutex implementations usually contain memory barriers

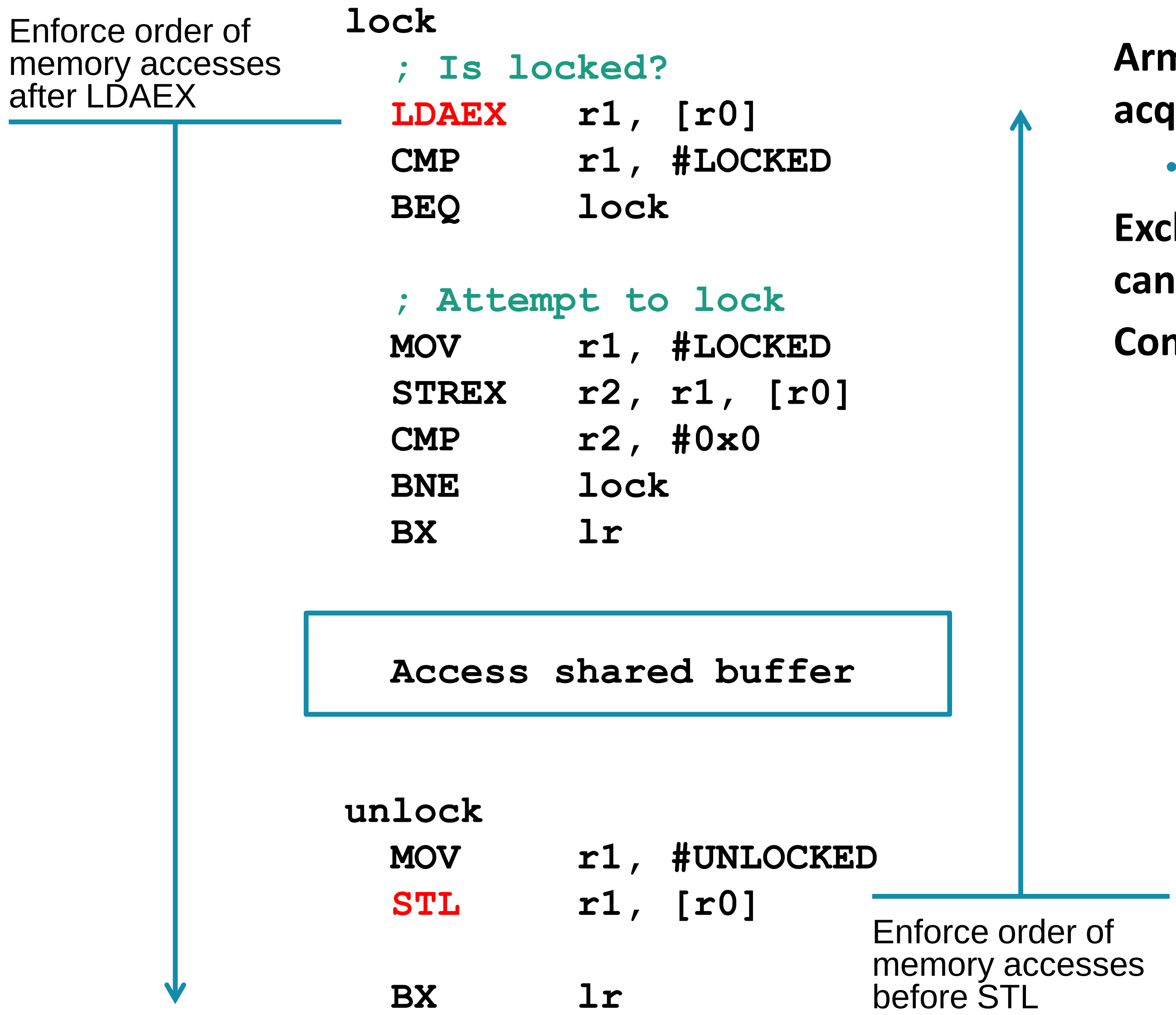
```
lock
  ; Is locked?
  LDREX    r1, [r0]
  CMP      r1, #LOCKED
  BEQ      lock
  ; Attempt to lock
  MOV      r1, #LOCKED
  STREX    r2, r1, [r0]
  CMP      r2, #0x0
  BNE      lock
  DMB
  BX       lr
```

```
unlock
  DMB

  MOV      r1, #UNLOCKED
  STR      r1, [r0]

  BX       lr
```

DMBs enforce ordering of explicit memory accesses before the DMB and explicit memory accesses after the DMB

- This might be more ordering than necessary

Exclusive access with LDAEX/STLEX


Enforce order of memory accesses after LDAEX
Enforce order of memory accesses before STL

```
lock
  ; Is locked?
  LDAEX    r1, [r0]
  CMP      r1, #LOCKED
  BEQ      lock

  ; Attempt to lock
  MOV      r1, #LOCKED
  STREX    r2, r1, [r0]
  CMP      r2, #0x0
  BNE      lock
  BX       lr

  Access shared buffer

unlock
  MOV      r1, #UNLOCKED
  STL      r1, [r0]

  BX       lr
```

Armv8-M introduces memory accesses with acquire/release semantics

- ”One way” barriers

Exclusive loads and stores with acquire/release semantics can be used instead of DMBs

Comply with C11/C++11 atomics

Thank You
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Merci
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה