GTM-IP Application note

AN020 – MCS Mutex implementation

Table of contents

1 Introduction

This application note describes methods to implement a mutual exclusion mechanism for concurrent running MCS channels which share a common resource. The exclusive usage of this shared resource is established by the implementation of a mutex, which has to be requested by an MCS channel, when he enters a code part (critical region), where he wants to access the shared resource.

The application note is applicable for GTM-IP starting from generation 1.

1.1 Problem description

The Multi Channel Sequencer (MCS) is a generic data processing module that is connected to the ARU. The MCS can do calculations on incoming signals and can generate complex output signals in a programmable manner. The MCS consists of several distinct channels, each with its own register set. The channels run in concurrent manner and share common resources like the ALU, the AEI-bus and MCS-memory. To access shared resources in a deterministic manner and to exclude race conditions, a semaphore mechanism is needed.

Actually, there is no dedicated HW-support to implement a semaphore mechanism for concurrent tasks within the MCS itself. Therefore, an alternative approach is needed to establish access to shared resources.

1.2 Outline

This application note describes an alternative approach for accessing a shared resource from different MCS channels. The solution uses a SW-mutex implemented with one dedicated MCS channel. This solution is described in section 2.

2 MCS Mutex implementation

2.1 Idea

Since there is no HW-support for mutual exclusion access to shared resources within MCS, other solutions have to be applied. For the implementation of a mutex mechanism, MCS internal resources like registers and memory can be used. The idea is to have one dedicated MCS channel which handles access to the shared resource. This dedicated channel is called *MCS_Gatekeeper* further on.
The other channels, called *MCS_Workers* further on, execute the application code and apply for access to the shared resource through a special register. They then monitor a memory cell for receiving access to the shared resource. For requesting the shared resource a dedicated bit in the MCS[i]_STRG register is used. For freeing the resource, the MCS[i]_CTRG register is used.

2.2 Resource consumption

Resource	Description
MCS Channel	One dedicated MCS channel is needed for administration of the shared resource. This should be the only task of this channel. The channel is called MCS_Gatekeeper.
STRG Register bit field	A bit field with adjacent bits in the global register MCS[i]_STRG is needed for requesting access to the shared resource by the MCS_Workers. There is one dedicated bit needed for each worker, which wants to access the shared resource. Table 2.2 shows an example for three concurrent MCS_Workers and their corresponding bits in MCS[i]_STRG register.
CTRG Register bit field	A bit field with adjacent bits in the global register MCS[i]_CTRG is needed for freeing up the shared resource for other MCS_Workers. There is one dedicated bit needed for each worker. The MCS_Worker writes a ‘1’ to its corresponding bit to free the resource. Table 2.3 shows an example for three concurrent MCS_Workers and their corresponding bits in MCS[i]_CTRG register.
MCS Memory cell	One dedicated MCS memory cell of 32 bit width is needed. This memory cell is used by the MCS_Gatekeeper to inform MCS_Workers if they can enter the critical region or not. Table 2.4 shows an example layout of the memory cell for three concurrent MCS_Workers.

Table 2.1: Resource consumption

2.2.1 MCS[i]_STRG Register organization

31	30	29	29	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								TRG23	TRG22	TRG21	TRG20	TRG19	TRG18	TRG17	TRG16	TRG15	TRG14	TRG13	TRG12	TRG11	TRG10	TRG9	TRG8	TRG7	TRG6	TRG5	TRG4	TRG3	TRG2	TRG1	TRG0
R								RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
0x00								0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 2.2: Example of MCS[i]_STRG resource consumption for three MCS_Workers

The bits TRG4, TRG5, and TRG6 of register MCS[i]_STRG are used in this example by the MCS_Workers to request access to a shared resource. Setting the corresponding bit to ‘1’ signals a request for accessing the resource. The MCS_Gatekeeper reads the MCS[i]_STRG register to determine channels, requesting access.
Ownership of the shared resource has to be requested by setting this bit and the shared resource must not be used by the MCS_Worker before the MCS_Gatekeeper granted the ownership.

2.2.2 MCS[i]_CTRG Register organization

31	30	29	29	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								TRG23	TRG22	TRG21	TRG20	TRG19	TRG18	TRG17	TRG16	TRG15	TRG14	TRG13	TRG12	TRG11	TRG10	TRG9	TRG8	TRG7	TRG6	TRG5	TRG4	TRG3	TRG2	TRG1	TRG0
R								RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
0x00								0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 2.3: Example of MCS[i]_CTRG resource consumption for three MCS_Workers

The bits TRG4, TRG5, and TRG6 of register MCS[i]_CTRG are used in this example by the MCS_Workers to return the ownership of the shared resource to the MCS_Gatekeeper. By writing a ‘1’ to its corresponding trigger bit, the MCS_Worker can free the shared resource. The trigger bit is also cleared in the MCS[i]_STRG register, which informs the MCS_Gatekeeper, that he can give the shared resource to another MCS channel, requesting the resource.
The MCS[i]_CTRG register bit is to be set by a MCS_Worker, after leaving the critical region, where the shared resource was accessed. Otherwise, the shared resource will never be assigned to another MCS_Worker by the MCS_Gatekeeper.

2.2.3 MCS Memory cell layout

31	30	29	29	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Bit 31	Bit 30	Bit 29	Bit 28	Bit 27	Bit 26	Bit 25	Bit 24	Bit 23	Bit 22	Bit 21	Bit 20	Bit 18	Bit 18	Bit 17	Bit 16	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 2.4: MCS Memory cell for usage with three MCS_Workers

Each MCS_Worker competing for a shared resource owns one bit of the MCS memory cell. The MCS_Gatekeeper uses this memory location to inform the MCS_Workers that one of them is allowed to enter the critical region and access the shared resource. This is signalled by the MCS_Gatekeeper by writing a ‘1’ to the corresponding memory cell bit. The MCS_Workers have to poll this bit after setting the MCS[i]_STRG register and before they enter the critical region.

2.3 Implementation

2.3.1 MCS_Gatekeeper

The MCS_Gatekeeper is responsible for restricting the access to a shared resource for one and only one MCS_Worker at one point in time. He does this with the help of the MCS[i]_STRG register and a dedicated memory cell in MCS RAM. The MCS_Gatekeeper scans periodically the MCS[i]_STRG register for bits set. The MCS_Gatekeeper grants access to the shared resource by setting the corresponding bit in the MCS RAM memory cell. The complete algorithm is shown in Figure 2.1. The code is shown in Code 1.

2.3.1.1 Algorithm for MCS_Gatekeeper implementation

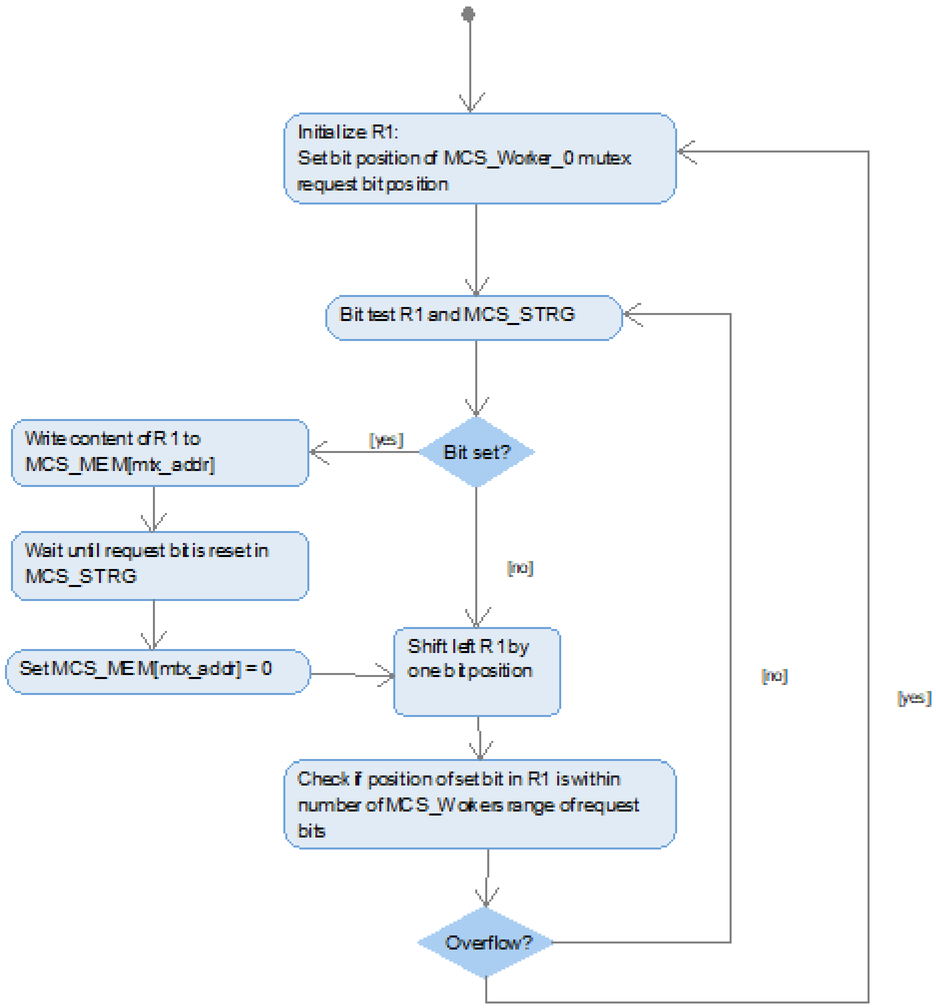

Figure 2.1: MCS_Gatekeeper algorithm

2.3.1.2 Source code for MCS_Gatekeeper implementation

```
.define mtx_ofs_c     4                  # bit offset for mtxbit field
.define mtx_len_c     3                  # three mcsworkers
.define mtx_mask_c    (((2**mtx_len_c)-1)*(2**mtx_ofs_c))

mtx_sta_v: .var       (2**mtx_ofs_c)     # MCS RAM position to assign mutex

mcs_gatekeeper:
     movl R2, 0x0
mtx_sched_start:
     movl R1, (2**mtx_ofs_c)             # mtx_ofs_c= 1. Bitposition
mtx_sched_test:
     bt   R1, STRG                       # test for mutex request
     jbs  STA, Z, mtx_sched_shift        # current bit == requested bit
     mov  R6, R1                         # prepare mask for wurmx
     mwr  R1, mtx_sta_v                  # assign mutex
     wurmx R2, STRG                      # wait until mutex release
     mwr  R2, mtx_sta_v                  # remove mutex assign bit
mtx_sched_shift:
     shl  R1, 1                          # try next request bit
     btl  R1, mtx_mask_c                 # check for overflow
     jbc  STA, Z, mtx_sched_test
     jmp  mtx_sched_start
```

Code 1: MCS_Gatekeeper sample source code

For the algorithm, the two variables `mtx_ofs_c` and `mtx_len_c` have to be initialized. `mtx_ofs_c` defines the offset of the bit field used to implement the mutex mechanism (in the example shown in Table 2.2, Table 2.3, Table 2.4, the first MCS_Worker resides at bit position four). `mtx_len_c` defines the number of workers, taking part in the competition for the shared resource (in the example, there are three workers). By this, the bit field is defined, where the MCS_Gatekeeper and MCS_Workers communicate with each other. It is important for the MCS_Gatekeeper algorithm, that the bit positions in the registers and memory correspond to each other for each MCS_Worker.

2.3.2 Implementation pattern for MCS_Workers

The MCS_Workers implement the application code. For accessing a shared resource, the MCS_Workers have to follow a strict communication scheme with the MCS_Gatekeeper for requesting ownership of a shared resource and freeing the shared resource up afterwards.
If this communication scheme is not followed by all MCS_Workers, the SW-mutex implementation described within this application note does not work at all. Figure 2.2 shows the communication scheme and Code 2 depicts the corresponding implementation pattern of the MCS_Worker.

2.3.2.1 Implementation pattern of MCS_Worker for requesting shared resources

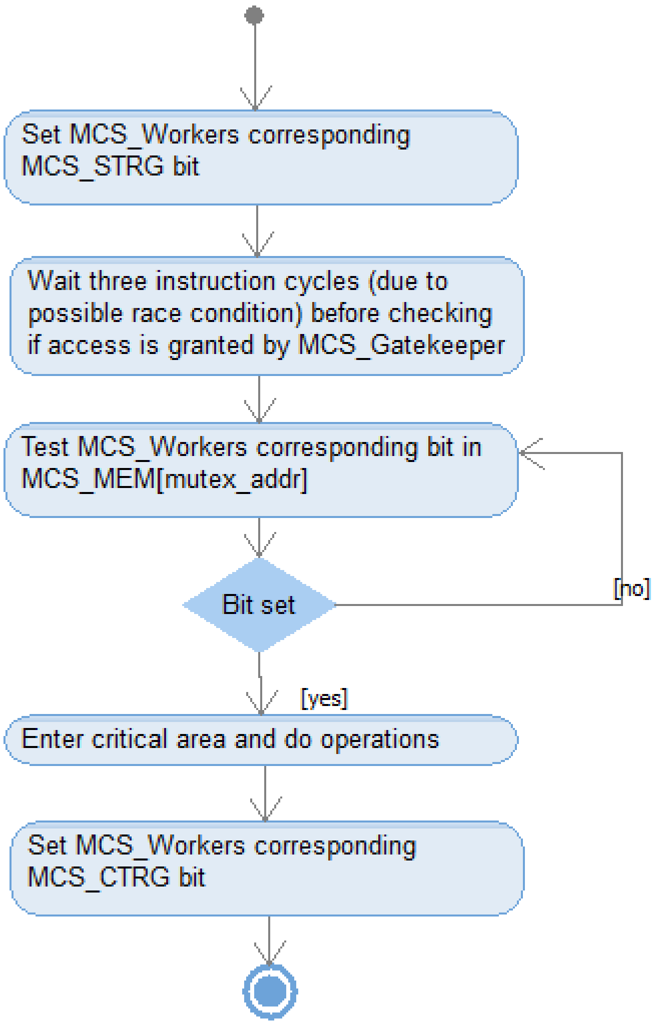

Figure 2.2: Requesting a shared resource by MCS_Worker

2.3.2.2 Source code example for MCS_Worker

```
mcs_worker_0:
      ## request mutex(set TRIGGER-Bit)
      movl STRG, TRIG_BIT_MCS_WORKER_0
      ## wait three instruction cycles before check for mutex
      nop
      nop
      nop
l_wait_locked_mtx_w0:
      ## check for successful mutexaccess
      mrd  R2, sem_sta_v
      andl R2, TRIG_BIT_MCS_WORKER_0
      jbs  STA, Z, l_wait_locked_mtx_w0
      ## do operation in critical region
      ## free mutex(clear TRIGGER-Bit)
      movl CTRG, TRIG_BIT_MCS_WORKER_0
```

Code 2: Example for MCS_Worker implementation

In a first step, the MCS_Worker sets its corresponding TRGx bit in the MCS[i]_STRG register, to inform the MCS_Gatekeeper, that he wants to enter the critical region. After the request, it is important for the requesting MCS_Worker to wait some time, before checking if the access is granted by the MCS_Gatekeeper. The amount of time depends on the algorithm used by the MCS_Gatekeeper to scan through and process pending requests. In this example, there are three NOPs which implement the wait time. For a detailed discussion of this timing constraint, please refer to section 2.3.3.

After this wait time, the MCS_Worker checks repeatedly if the access is granted to him by checking the corresponding bit in the memory location. If the bit was set by the MCS_Gatekeeper, the MCS_Worker is allowed to enter the critical region and access the shared resource.
When the MCS_Worker no longer requires the shared resource, he has to clear his dedicated corresponding trigger bit in MCS[i]_STRG, by writing a '1' to the associated bit in MCS[i]_CTRG. This frees the resource and allows the MCS_Gatekeeper to continue scanning the MCS[i]_STRG register for new/pending requests.

2.3.3 Timing considerations

It is important to note, that the MCS_Workers should not immediately monitor the memory cell for their bit, because the MCS_Gatekeeper needs some time to process the information inside of the MCS[i]_STRG register and set/clear bits in the memory cell.
This is because the resource ownership bit in the MCS memory stays set until it is cleared by the MCS_Gatekeeper after he received the clearing of the MCS[i]_STRG trigger bit. Since this clearing takes some time, the MCS_Worker has to wait before reading the bit field in the MCS memory.

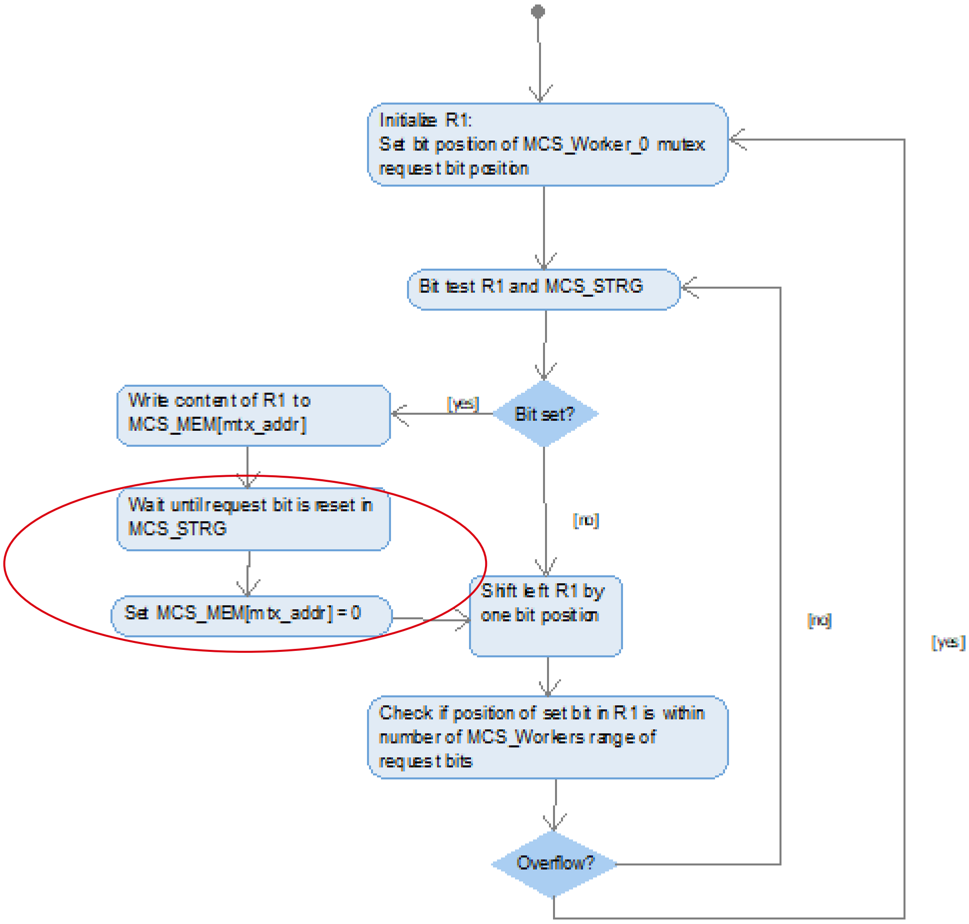

Figure 2.3: Non-atomic administration of MCS memory cell

Figure 2.3 shows the critical section, where the bit in the MCS memory can be set, while the MCS_Gatekeeper is modifying the MCS memory content. The instructions behind these two activities of the algorithm are the two `wurmx R2, STRG` and `mwr R2, mtx_sta_v` instructions.
The time for `wurmx` is one instruction cycle, while `mwr` takes two instruction cycles. Thus, a total of three instruction cycles are necessary to update the MCS memory and clear the bit of the old requesting MCS_Worker.
Those three instruction cycles are inserted by three NOPs in the MCS_Worker code.

3 References

4 General Information about this document

4.1 LEGAL NOTICE

4.2 Revision History

Version	Date	Description
0.1	25.07.2018	Initial version
1.0	27.07.2018	Released

Robert Bosch GmbH
AE/EID5
Postfach 13 42
72703 Reutlingen
Germany

bosch.semiconductors@de.bosch.com