Application Note

MMA Error Injection

Manojna Manojna

ABSTRACT

Fault injection and exception handling are critical for a safety system, this document explains how MMA faults are injected sequentially.

Table of Contents

Trademarks

All trademarks are the property of their respective owners.

1 Introduction

The Matrix Multiply Accelerator (MMA) is a tightly coupled matrix multiplication coprocessor that extends the scalar and vector facilities of the C7x processor architecture. Fed through the streaming engines (SEs), the MMA provides a large number of multiply-accumulate (MAC) operations, matrix-optimized storage, and matrix-optimized data movement that can be efficiently used for the dense linear algebra operations that dominate the computations found in vision (CNNs, SfM, filtering, and so forth), speech (xNNs), audio (convolutions), radar (FFTs) and controls (reinforcement learning, state updates) based applications.

This document covers the sequential process of fault injection, specifically focusing on Matrix Multiplier Accelerator (MMA)diagnostics.

This document explains about how these tests can be conducted sequentially, verifying a detailed understanding of fault handling and recovery.

This document does not discuss the recovery of the exception to execute the regular flow of the code after the sequence of error injection is done. Please refer to SDK 11.2 for integration example.

2 Instructions

2.1 Scope

This document focuses solely on the sequential fault injections tests for the MMA and provides details on how to trigger and conduct these faults.

This document does not discuss the procedure for resetting c7x after the sequential test is completed. Post-test recovery, system reset, and system state restorations are outside the scope of this document and must be conducted separately according to the system design and fault tolerance.

2.2 HWA Instruction Definition

The hardware accelerator (HWA) is tightly coupled with the C7x CPU. The HWA accepts commands and operates from the C7x CPU, performs specific computation tasks, and returns the result back to the C7x CPU.

- HWAOPEN: Sends the compute template to the HWA to communicate to the HWA the types of operands and computations to use for subsequent HWA operations.
- HWALDA, HWALDB, HWALDC, HWALDAB, HWALDBC: These various HWALD instructions are used to read values from the GRF, LRFL and SE0, SE1 registers and send the registers out to the HWA to initialize the HWA operand matrices.
- HWAOP: HWAOP instruction tells the HWA to proceed on the programmed computation.
- HWAXFER: HWAXFER instruction transfers the computed result from the HWA logic to the internal buffers.
- HWARCVS: HWARCVS instruction transfers the value stored in the HWA internal buffers to the C7x CPU general purpose registers.
- HWACLOSE: HWACLOSE closes the HWA operations. All intermediate values inside the HWA are discarded.

More details are available in MMA specifications document. See *C71x DSP CPU, Instruction Set, and Matrix Multiply Accelerator* for more details.

2.3 Fault Injection Procedure

Fault Injection Procedure

To evaluate MMA diagnostics, faults must be deliberately injected into the system while monitoring the resulting behavior. When the HWA_STATUS error code field contains non-zero values, executing the HWARCV_(0) instruction triggers a c7x exception.

After the exception occurs, the IERR and IESR registers must be read to determine the nature of the fault:

- IERR: Indicates the cause of the internal exception through specific flags.
- IESR: Provides additional details about fault type and subtype.

2.3.1 Block Diagram

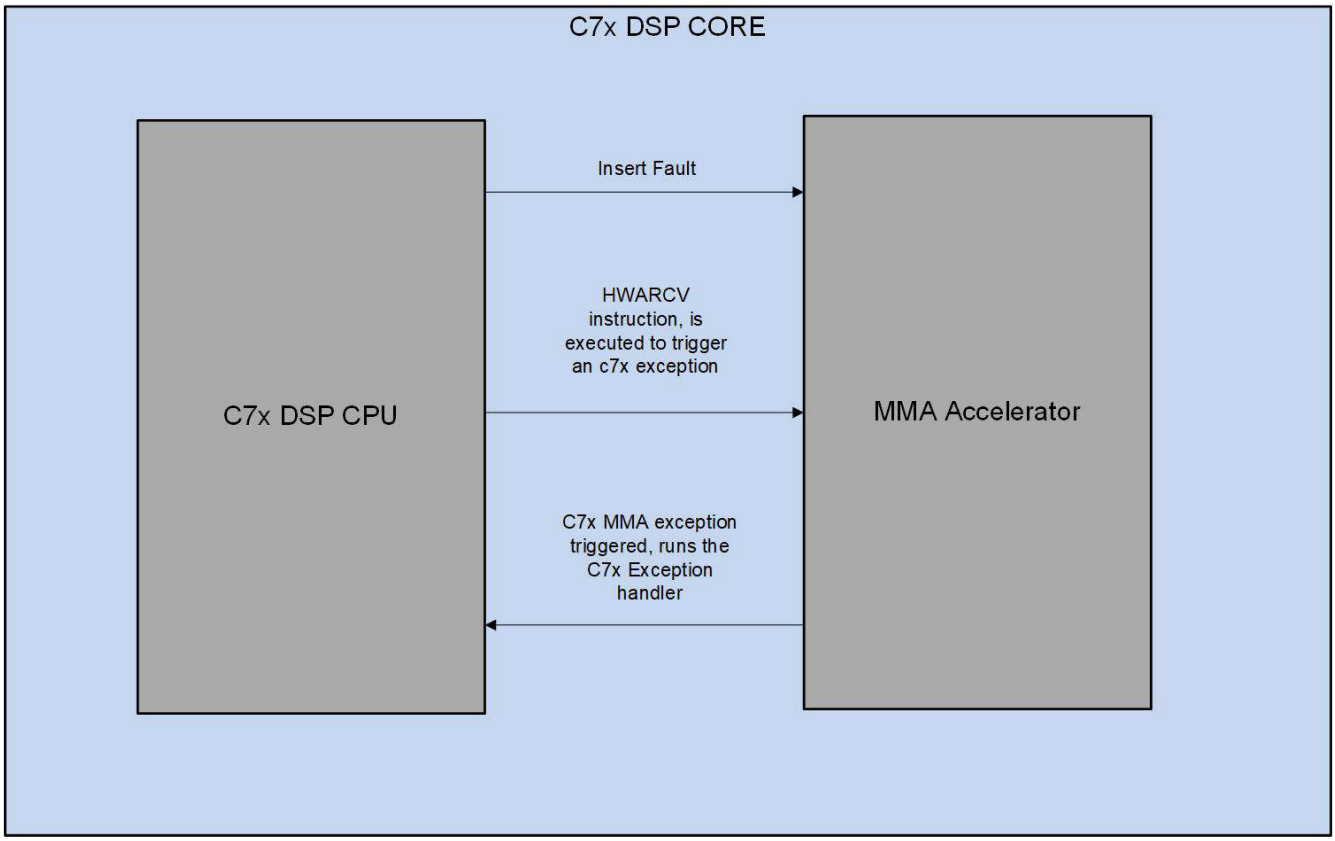

Figure 2-1. C7x -MMA Error Injection Block Diagram

2.3.2 Fault Injection: Steps

2.3.2.1 Underflow Error

Underflow error is caused when Transfer buffer read was attempted but there was no valid data in the FIFO (programming error),given below are the steps to inject underflow error.

1. The HWAOPEN instruction is used to write HWA_CONFIG and HWA_OFFSET.
2. Execute HWAXFER to transfer the content of HWA_STATUS, to internal buffer.
3. Execute HWARCV_(0) instruction to transfers the value stored in the HWA internal buffers to C7x CPU general purpose registers.
4. Repeat Step 3 until there is no valid data in the internal buffer to c7x CPU which triggers exception.

```
void underflow_exception(void)
{
  __HWA_CONFIG_REG_v1 mma_config_reg;
  mma_config_reg = __gen_HWA_CONFIG_REG_v1();
  __HWA_OFFSET_REG offset_reg;
  offset_reg = __gen_HWA_OFFSET_REG();
  __HWAOPEN(mma_config_reg,offset_reg,__MMA_OPEN_FSM_RESET);
  __HWAADV();
  __HWAADV();
  __HWAXFER(__MMA_XFER_SRC_HWA_STATUS);
  __HWAADV();
  __HWAADV();
  __HWAADV();
  __HWAADV();
  __HWARCV(0);
  __HWARCV(0);
}
```

2.3.2.2 Overflow Error

An overflow error is caused when a transfer buffer read is attempted but an earlier HWA instruction caused the FIFO to overflow (programming error). Follow the steps listed to inject the error.

1. The HWAOPEN instruction is used to write HWA_CONFIG and HWA_OFFSET.
2. Execute HWAXFER to transfer the content of HWA_STATUS, to internal buffer.
3. Repeat Step 2 until buffer is overflowed.
4. The HWARCV instruction is executed to observe the error.

```
void overflow_exception(void)
{
  __HWA_CONFIG_REG_v1 mma_config_reg;
  mma_config_reg = __gen_HWA_CONFIG_REG_v1();
  __HWA_OFFSET_REG offset_reg;
  offset_reg = __gen_HWA_OFFSET_REG();
  __HWAOPEN(mma_config_reg,offset_reg,__MMA_OPEN_FSM_RESET);
  int loop_begin = 1;
  int loop_end  = 30;
  for(;loop_begin<=loop_end;loop_begin++)
  {
    __HWAXFER(__MMA_XFER_SRC_HWA_STATUS);
    __HWAADV();
    __HWAADV();
    __HWAADV();
    __HWAADV();
  }
  __HWARCV(0);
}
```

2.3.2.3 Offset Parity Error

An offset parity error occurs when a parity mismatch is detected in the HWA_OFFSET register. Use the following steps to inject the HWA_OFFSET parity error.

1. The HWAOPEN instruction is used to write HWA_CONFIG and HWA_OFFSET with parity computation enabled and parity checking enabled (PARITYCTRL = NORMAL).
2. The values in the vector registers are corrupted by the software in the specific field to be tested.
3. The HWAOPEN instruction is again used to write HWA_CONFIG and HWA_OFFSET using the corrupted data from that C7x vector register with the parity computation disabled and parity checking enabled (PARITYCTRL = PNCMCK). In this mode, the originally computed and saved parity value is checked against the parity computed on use in the affected register.
4. Execute HWAXFER to transfer the content of HWA_CONFIG/HWA_OFFSET and so on to transfer the buffer.
5. The HWARCV instruction is executed to observe the error.

```
 __HWA_OFFSET_REG get_corrupted_offset(void)
{
__HWA_OFFSET_REG mma_offset_reg = __gen_HWA_OFFSET_REG();
mma_offset_reg.A_LUT_VAL_1 = 0x01;
return mma_offset_reg; //return the corrupted offset
}
#pragma FUNC_CANNOT_INLINE(offset_parity_test)
#pragma FUNCTION_OPTIONS(offset_parity_test,"--opt_level=0")
void offset_parity_test(void)
{

__HWA_CONFIG_REG_v1 mma_config_reg;
mma_config_reg = __gen_HWA_CONFIG_REG_v1();
mma_config_reg.A_ALUTEN = __MMA_A_CONFIG_LUT2;
mma_config_reg.PARITYCTRL = __MMA_NORMAL;
__HWA_OFFSET_REG offset_reg;
offset_reg = __gen_HWA_OFFSET_REG();
__HWAOPEN(mma_config_reg,offset_reg,__MMA_OPEN_FSM_RESET);
__HWAADV();
__HWAADV();
__HWAADV();
__HWAADV();
__HWA_CONFIG_REG_v1 mma_config_1;
mma_config_1 = __gen_HWA_CONFIG_REG_v1();
mma_config_1.A_ALUTEN = __MMA_A_CONFIG_LUT2;
//Parity computation disabled and parity checking enabled.
mma_config_1.PARITYCTRL = __MMA_PNCM_CK;
__HWA_OFFSET_REG offset_reg2 = get_corrupted_offset(); //This gets the corrupted offset
__HWAADV();
__HWAADV();
__HWAADV();
__HWAADV();
__HWAOPEN(mma_config_1,offset_reg2,__MMA_OPEN_FSM_RESET);
__HWAADV();
__HWAADV();
__HWAADV();
__HWAADV();
__HWAXFER(__MMA_XFER_SRC_HWA_OFFSET);
__HWAADV();
__HWAADV();
__HWAADV();
__HWARCV(0);
__asm(" NOP");
__asm(" NOP");


}
```

2.3.2.4 Config Parity Error

An config parity error occurs when a parity mismatch is detected in the HWA_CONFIG register. The procedure on how to inject the HWA_CONFIG parity error are listed in the following steps. All the FSM configurations (A,B,C,X FSM Configurations) are protected by the parity.

1. The HWAOPEN instruction is used to write HWA_CONFIG and HWA_OFFSET with parity computation enabled and parity checking enabled (PARITYCTRL = NORMAL).
2. The values in the vector registers are corrupted by software in the specific field to be tested.
3. The HWAOPEN instruction is again used to write HWA_CONFIG and HWA_OFFSET using the corrupted data from that C7x vector register with the parity computation disabled and parity checking enabled (PARITYCTRL = PNCMCK). In this mode, the originally computed and saved parity value is checked against the parity computed on use in the affected register.
4. Execute HWAXFER to transfer the content of HWA_CONFIG/HWA_OFFSET and so on to transfer the buffer.
5. The HWARCV instruction is executed to observe the error

2.3.2.4.1 A FSM Config Parity Error

An A FSM config parity error occurs when a parity mismatch is detected in the HWA_CONFIG register. The procedure on how to inject the A FSM Config parity error are listed in the following steps.

1. The HWAOPEN instruction is used to write HWA_CONFIG and HWA_OFFSET with parity computation enabled and parity checking enabled (PARITYCTRL = NORMAL) also configure HWA_CONFIG with the parameters mentioned in Get_MMAconfig().
2. The values in the vector registers are corrupted by software in the specific field to be tested(A FSM).
3. The HWAOPEN instruction is again used to write HWA_CONFIG and HWA_OFFSET using the corrupted data from that C7x vector register with the parity computation disabled and parity checking enabled (PARITYCTRL = PNCMCK). In this mode, the originally computed and saved parity value is checked against the parity computed on use in the affected register.
4. Execute HWALDA /HWAXFER to inject the error. The error is updated in HWA_STATUS register.
5. The HWARCV instruction is executed to observe the error

```
#pragma FUNCTION_OPTIONS(Get_AFSM_Corrupt_config,"--opt_level=off")
__HWA_CONFIG_REG_v1 Get_AFSM_Corrupt_config(void)
{
  __HWA_CONFIG_REG_v1 corrupted_config = __gen_HWA_CONFIG_REG_v1();
  corrupted_config.A_ATYPE = __MMA_A_CONFIG_ATYPE_UINT32;
  corrupted_config.A_ALUTEN = __MMA_A_LUTEN_LAST;
  return corrupted_config;
}
#pragma FUNCTION_OPTIONS(Get_MMAconfig,"--opt_level=off")
__HWA_CONFIG_REG_v1 Get_MMAconfig(void)
{
  __HWA_CONFIG_REG_v1 mma_config_reg;
  mma_config_reg = __gen_HWA_CONFIG_REG_v1();
  mma_config_reg.A_ATYPE = __MMA_A_CONFIG_ATYPE_UINT16;
  mma_config_reg.A_ALUTEN = __MMA_A_LUTEN_LAST;
  mma_config_reg.B_BSWPER = 1000;
  mma_config_reg.B_BTYPE = __MMA_B_CONFIG_SIZE16;
  mma_config_reg.B_BSTART = 1;
  mma_config_reg.C_ATYPE = __MMA_C_CONFIG_ATYPE_SA;
  mma_config_reg.C_BTYPE = __MMA_C_CONFIG_BTYPE_UINT16;
  mma_config_reg.PARITYCTRL = __MMA_NORMAL;
  mma_config_reg.C_BSTART = 1; /* Initial B bank selection for reading B matrix data for the matrix
computations */
  mma_config_reg.C_CRSTART = 1; /* Initial C bank selection for reading operands */
  mma_config_reg.C_CWSTART = 1; /* Initial C bank selection for writing computation results */
  mma_config_reg.C_CLSTART = 1;
  return mma_config_reg;
}
#pragma FUNCTION_OPTIONS(MMA_AFSM_configParityError_injection,"--opt_level=off")
void MMA_AFSM_configParityError_injection(void)
{
  __HWA_CONFIG_REG_v1 mma_config_reg = Get_MMAconfig();
  __HWA_OFFSET_REG offset_reg;
  offset_reg = __gen_HWA_OFFSET_REG();
```

```
  __HWAOPEN(mma_config_reg,offset_reg,__MMA_OPEN_FSM_RESET);
  __HWAADV();
  __HWAADV();
  __HWAADV();
  __asm(" HWALDA .L2 VB1");
  __HWAXFER(__MMA_XFER_SRC_C);
  __HWA_CONFIG_REG_v1 mma_config_corrupt = Get_AFSM_Corrupt_config();
  mma_config_corrupt.PARITYCTRL = __MMA_PNCM_CK;
  __HWA_OFFSET_REG offset_reg2 = __gen_HWA_OFFSET_REG();
  __HWAADV();
  __HWAADV();
  __HWAADV();
  __HWAOPEN(mma_config_corrupt,offset_reg2,__MMA_OPEN_FSM_RESET);
  __HWAADV();
  __HWAADV();
  __HWAADV();
  __asm(" HWALDA .L2 VB0"); //Load to A matrix
  __HWAXFER(__MMA_XFER_SRC_HWA_CONFIG);
  __HWAADV();
  __HWAADV();
  __HWAADV();
  __HWARCV(0);
  __asm(" NOP");
  __asm(" NOP");
}
```

2.3.2.4.2 B FSM Config Parity Error

An B FSM config parity error occurs when a parity mismatch is detected in the HWA_CONFIG register. The procedure on how to inject the B FSM Config parity error are listed in the following steps.

1. The HWAOPEN instruction is used to write HWA_CONFIG and HWA_OFFSET with parity computation enabled and parity checking enabled (PARITYCTRL = NORMAL) also configure HWA_CONFIG with the parameters mentioned in Get_MMAconfig().
2. The values in the vector registers are corrupted by software in the specific field to be tested(B FSM).
3. The HWAOPEN instruction is again used to write HWA_CONFIG and HWA_OFFSET using the corrupted data from that C7x vector register with the parity computation disabled and parity checking enabled (PARITYCTRL = PNCMCK). In this mode, the originally computed and saved parity value is checked against the parity computed on use in the affected register.
4. Execute HWALDB /HWAXFER to inject the error. The error is updated in HWA_STATUS register.
5. The HWARCV instruction is executed to observe the error

```
#pragma FUNCTION_OPTIONS(Get_BFSM_Corrupt_config,"--opt_level=off")
__HWA_CONFIG_REG_v1 Get_BFSM_Corrupt_config(void)
{
  __HWA_CONFIG_REG_v1 corrupted_config = __gen_HWA_CONFIG_REG_v1();
  corrupted_config.B_BSWPER = 100;
  corrupted_config.B_BTYPE = __MMA_B_CONFIG_SIZE32;
  return corrupted_config;
}
#pragma FUNCTION_OPTIONS(Get_MMAconfig,"--opt_level=off")
__HWA_CONFIG_REG_v1 Get_MMAconfig(void)
{
  __HWA_CONFIG_REG_v1 mma_config_reg;
  mma_config_reg = __gen_HWA_CONFIG_REG_v1();
  mma_config_reg.A_ATYPE = __MMA_A_CONFIG_ATYPE_UINT16;
  mma_config_reg.A_ALUTEN = __MMA_A_LUTEN_LAST;
  mma_config_reg.B_BSWPER = 1000;
  mma_config_reg.B_BTYPE = __MMA_B_CONFIG_SIZE16;
  mma_config_reg.B_BSTART = 1;
  mma_config_reg.C_ATYPE = __MMA_C_CONFIG_ATYPE_SA;
  mma_config_reg.C_BTYPE = __MMA_C_CONFIG_BTYPE_UINT16;
  mma_config_reg.PARITYCTRL = __MMA_NORMAL;
  mma_config_reg.C_BSTART = 1; /* Initial B bank selection for reading B matrix data for the matrix
computations */
  mma_config_reg.C_CRSTART = 1; /* Initial C bank selection for reading operands */
  mma_config_reg.C_CWSTART = 1; /* Initial C bank selection for writing computation results */
  mma_config_reg.C_CLSTART = 1;
  return mma_config_reg;
}
#pragma FUNCTION_OPTIONS(MMA_BFSM_configParityError_injection,"--opt_level=off")
```

```
void MMA_BFSM_configParityError_injection(void)
{
__HWA_CONFIG_REG_v1 mma_config_reg = Get_MMAconfig();
  __HWA_OFFSET_REG offset_reg;
  offset_reg = __gen_HWA_OFFSET_REG();
  __HWAOPEN(mma_config_reg,offset_reg,__MMA_OPEN_FSM_RESET);
  __HWAADV();
  __HWAADV();
  __HWAADV();
  __asm(" HWALDB .L2 VB1");
  __HWAXFER(__MMA_XFER_SRC_C);
  __HWA_CONFIG_REG_v1 mma_config_corrupt = Get_BFSM_Corrupt_config();
  mma_config_corrupt.PARITYCTRL = __MMA_PNCM_CK;
  __HWA_OFFSET_REG offset_reg2 = __gen_HWA_OFFSET_REG();
  __HWAADV();
  __HWAADV();
  __HWAADV();
  __HWAOPEN(mma_config_corrupt,offset_reg2,__MMA_OPEN_FSM_RESET);
  __HWAADV();
  __HWAADV();
  __HWAADV();
  __asm(" HWALDB .L2 VB0"); //Load to B matrix
  __HWAXFER(__MMA_XFER_SRC_HWA_CONFIG);
  __HWAADV();
  __HWAADV();
  __HWAADV();
  __HWARCV(0);
  __asm(" NOP");
  __asm(" NOP");

}
```

2.3.2.4.3 C FSM Config Parity Error

An C FSM config parity error occurs when a parity mismatch is detected in the HWA_CONFIG register. The procedure on how to inject the C FSM Config parity error are listed in the following steps.

1. The HWAOPEN instruction is used to w rite HWA_CONFIG and HWA_OFFSET with parity computation enabled and parity checking enabled (PARITYCTRL = NORMAL) also configure HWA_CONFIG with the parameters mentioned in Get_MMAconfig().
2. The values in the vector registers are corrupted by software in the specific field to be tested(C FSM).
3. The HWAOPEN instruction is again used to write HWA_CONFIG and HWA_OFFSET using the corrupted data from that C7x vector register with the parity computation disabled and parity checking enabled (PARITYCTRL = PNCMCK). In this mode, the originally computed and saved parity value is checked against the parity computed on use in the affected register.
4. Execute HWALDC /HWAXFER to inject the error. The error is updated in HWA_STATUS register.
5. The HWARCV instruction is executed to observe the error.

```
#pragma FUNCTION_OPTIONS(Get_CFSM_Corrupt_config,"--opt_level=off")
__HWA_CONFIG_REG_v1 Get_CFSM_Corrupt_config(void)
{
  __HWA_CONFIG_REG_v1 corrupted_config = __gen_HWA_CONFIG_REG_v1();
  corrupted_config.C_ATYPE = __MMA_C_CONFIG_ATYPE_UA;
  corrupted_config.C_BTYPE = __MMA_C_CONFIG_BTYPE_UINT16;
  return corrupted_config;
}
#pragma FUNCTION_OPTIONS(Get_MMAconfig,"--opt_level=off")
__HWA_CONFIG_REG_v1 Get_MMAconfig(void)
{
  __HWA_CONFIG_REG_v1 mma_config_reg;
  mma_config_reg = __gen_HWA_CONFIG_REG_v1();
  mma_config_reg.A_ATYPE = __MMA_A_CONFIG_ATYPE_UINT16;
  mma_config_reg.A_ALUTEN = __MMA_A_LUTEN_LAST;
  mma_config_reg.B_BSWPER = 1000;
  mma_config_reg.B_BTYPE = __MMA_B_CONFIG_SIZE16;
  mma_config_reg.B_BSTART = 1;
  mma_config_reg.C_ATYPE = __MMA_C_CONFIG_ATYPE_SA;
  mma_config_reg.C_BTYPE = __MMA_C_CONFIG_BTYPE_UINT16;
  mma_config_reg.PARITYCTRL = __MMA_NORMAL;
  mma_config_reg.C_BSTART = 1; /* Initial B bank selection for reading B matrix data for the matrix
computations */
  mma_config_reg.C_CRSTART = 1; /* Initial C bank selection for reading operands */
```

```
  mma_config_reg.C_CWSTART = 1; /* Initial C bank selection for writing computation results */
  mma_config_reg.C_CLSTART = 1;
  return mma_config_reg;
}

#pragma FUNCTION_OPTIONS(MMA_CFSM_configParityError_injection,"--opt_level=off")
void MMA_CFSM_configParityError_injection(void)
{
  __HWA_CONFIG_REG_v1 mma_config_reg = Get_MMAconfig();
  __HWA_OFFSET_REG offset_reg;
  offset_reg = __gen_HWA_OFFSET_REG();
  __HWAOPEN(mma_config_reg,offset_reg,__MMA_OPEN_FSM_RESET);
 __HWAADV();
  __HWAADV();
  __HWAADV();
  __asm(" HWALDC .L2 VB1");
  __HWAXFER(__MMA_XFER_SRC_C);
  __HWA_CONFIG_REG_v1 mma_config_corrupt = Get_CFSM_Corrupt_config();
  mma_config_corrupt.PARITYCTRL = __MMA_PNCM_CK;
  __HWA_OFFSET_REG offset_reg2 = __gen_HWA_OFFSET_REG();
  __HWAADV();
  __HWAADV();
  __HWAADV();
  __HWAOPEN(mma_config_corrupt,offset_reg2,__MMA_OPEN_FSM_RESET);
   __HWAADV();
  __HWAADV();
  __HWAADV();
  __asm(" HWALDC .L2 VB0"); //Load to C Matrix
  __HWAXFER(__MMA_XFER_SRC_HWA_CONFIG);
 __HWAADV();
  __HWAADV();
  __HWAADV();
  __HWARCV(0);
  __asm(" NOP");
  __asm(" NOP");
}
```

2.3.2.4.4 X FSM Config Parity Error

An X FSM config parity error occurs when a parity mismatch is detected in the HWA_CONFIG register. The procedure on how to inject the X FSM Config parity error are listed in the following steps.

1. The HWAOPEN instruction is used to write HWA_CONFIG and HWA_OFFSET with parity computation enabled and parity checking enabled (PARITYCTRL = NORMAL) also configure HWA_CONFIG with the parameters mentioned in Get_MMAconfig().
2. The values in the vector registers are corrupted by software in the specific field to be tested(X FSM).
3. The HWAOPEN instruction is again used to write HWA_CONFIG and HWA_OFFSET using the corrupted data from that C7x vector register with the parity computation disabled and parity checking enabled (PARITYCTRL = PNCMCK). In this mode, the originally computed and saved parity value is checked against the parity computed on use in the affected register.
4. Execute HWAXFER to inject the error. The error is updated in HWA_STATUS register.
5. The HWARCV instruction is executed to observe the error.

```
#pragma FUNCTION_OPTIONS(Get_XFSM_Corrupt_config,"--opt_level=off")
__HWA_CONFIG_REG_v1 Get_XFSM_Corrupt_config(void)
{
  __HWA_CONFIG_REG_v1 corrupted_config = __gen_HWA_CONFIG_REG_v1();
  corrupted_config.X_ReLU = 1;
  return corrupted_config;
}
#pragma FUNCTION_OPTIONS(Get_MMAconfig,"--opt_level=off")
__HWA_CONFIG_REG_v1 Get_MMAconfig(void)
{
  __HWA_CONFIG_REG_v1 mma_config_reg;
  mma_config_reg = __gen_HWA_CONFIG_REG_v1();
  mma_config_reg.A_ATYPE = __MMA_A_CONFIG_ATYPE_UINT16;
  mma_config_reg.A_ALUTEN = __MMA_A_LUTEN_LAST;
  mma_config_reg.B_BSWPER = 1000;
  mma_config_reg.B_BTYPE = __MMA_B_CONFIG_SIZE16;
  mma_config_reg.B_BSTART = 1;
  mma_config_reg.C_ATYPE = __MMA_C_CONFIG_ATYPE_SA;
```

```
  mma_config_reg.C_BTYPE = __MMA_C_CONFIG_BTYPE_UINT16;
  mma_config_reg.PARITYCTRL = __MMA_NORMAL;
  mma_config_reg.C_BSTART = 1; /* Initial B bank selection for reading B matrix data for the matrix
computations */
  mma_config_reg.C_CRSTART = 1; /* Initial C bank selection for reading operands */
  mma_config_reg.C_CWSTART = 1; /* Initial C bank selection for writing computation results */
  mma_config_reg.C_CLSTART = 1;
  return mma_config_reg;
}
#pragma FUNCTION_OPTIONS(MMA_XFSM_configParityError_injection,"--opt_level=off")
void MMA_XFSM_configParityError_injection(void)
{
  __HWA_CONFIG_REG_v1 mma_config_reg = Get_MMAconfig();
  __HWA_OFFSET_REG offset_reg;
  offset_reg = __gen_HWA_OFFSET_REG();
  __HWAOPEN(mma_config_reg,offset_reg,__MMA_OPEN_FSM_RESET); //Initialises Config,offset
 __HWAXFER(__MMA_XFER_SRC_HWA_CONFIG);
  __HWA_CONFIG_REG_v1 mma_config_corrupt = Get_XFSM_Corrupt_config(); //Receives X FSM corrupt
configuration
  mma_config_corrupt.PARITYCTRL = __MMA_PNCM_CK; //Parity checking enabled calculation disabled
  __HWA_OFFSET_REG offset_reg2 = __gen_HWA_OFFSET_REG();
  __HWAADV();
  __HWAADV();
  __HWAADV();
  __HWAOPEN(mma_config_corrupt,offset_reg2,__MMA_OPEN_FSM_RESET);
 __HWAADV();
  __HWAADV();
  __HWAADV();
  __HWAXFER(__MMA_XFER_SRC_HWA_CONFIG);
  __HWAADV();
  __HWAADV();
  __HWAADV();
  __HWARCV(0);
  __asm(" NOP");
  __asm(" NOP");
}
```

2.3.2.5 C READ Error

A C read error occurs when there are multiple simultaneous read accesses to the same C matrix bank, which leads to a conflict. Use the following steps to inject this error.

1. The HWAOPEN instruction is used to write HWA_CONFIG and HWA_OFFSET by configuring the operation as MULPLUS.
2. Execute HWAOP and HWAXFER operation in parallel.
3. Short and frequent read and writes to c matrix bank triggers error.
4. The HWARCV instruction is executed to observe the error.

```
void c_read_error(void)
{
    int switch_period=0;
    int repeat_op_count = 0;
    for( ;switch_period<5;switch_period++)
    {
        __HWA_CONFIG_REG_v1 config_reg = __gen_HWA_CONFIG_REG_v1();
        config_reg.C_OPERATION0 = __MMA_C_CONFIG_MULPLUS;
        config_reg.C_OPERATION1 = __MMA_C_CONFIG_MULPLUS;
        config_reg.C_CRSWPER = switch_period; //different switching period
        config_reg.C_CWSWPER  = switch_period+10000;
        __HWA_OFFSET_REG offset_reg;
        offset_reg = __gen_HWA_OFFSET_REG(); //This generates the offset register.
        __HWAOPEN(config_reg,offset_reg,__MMA_OPEN_FSM_RESET); //This opens theHWAOPEN
        repeat_op_count = 0;
        for(;repeat_op_count<5;repeat_op_count++)
        {
        __asm(" HWAOP.S1 SA0");
        __asm(" ||HWAXFER .L1 CMAT");
        __HWAADV();
        __HWAADV();
        __HWAADV();
        __HWAADV();
        }
    }
  __HWARCV(0); //calls the error
  __asm(" NOP");
__asm(" NOP");
}
```

2.3.2.6 C Write Error

A C write error occurs when there are multiple simultaneous write access to the same C matrix bank, which leads to a conflict. Use the following steps to inject this error.

1. The HWAOPEN instruction is used to write HWA_CONFIG and HWA_OFFSET by configuring the operation as MULPLUS.
2. Execute HWAOP and HWALDC operation in parallel.
3. Short and frequent read and writes to c matrix bank triggers an error.
4. The HWARCV instruction is executed to observe the error.

```
void c_write_error(void)
{
   int switch_period=0;
    int repeat_op_count = 0;
    for( ;switch_period<2;switch_period++)
    {
        __HWA_CONFIG_REG_v1 config_reg = __gen_HWA_CONFIG_REG_v1();
        config_reg.C_OPERATION0 = __MMA_C_CONFIG_MULPLUS;
        config_reg.C_OPERATION1 = __MMA_C_CONFIG_MULPLUS;
        config_reg.C_CRSWPER = switch_period; //different switching period
        config_reg.C_CWSWPER  = switch_period+10000;
        __HWA_OFFSET_REG offset_reg;
        offset_reg = __gen_HWA_OFFSET_REG(); //This generates the offset register.
        __HWAOPEN(config_reg,offset_reg,__MMA_OPEN_FSM_RESET); //This opens theHWAOPEN
        repeat_op_count = 0;
        for(;repeat_op_count<10;repeat_op_count++)
        {
        __asm(" HWALDC .L2 VB0"); //Earlier SE0++
        __asm(" ||HWAOP.S1 SA0"); //This should run them in parallel
        __HWAADV();
        __HWAADV();
        __HWAADV();
        __HWAADV();
        }
    }
    __HWARCV(0);  //Calls the error
    __asm(" NOP");
    __asm(" NOP");
}
```

3 Flow Diagram

The state diagram represents the sequential execution of fault injection tests of MMA. The image outlines the structured approach to introduce faults and recover from the fault and inject the next set of faults.

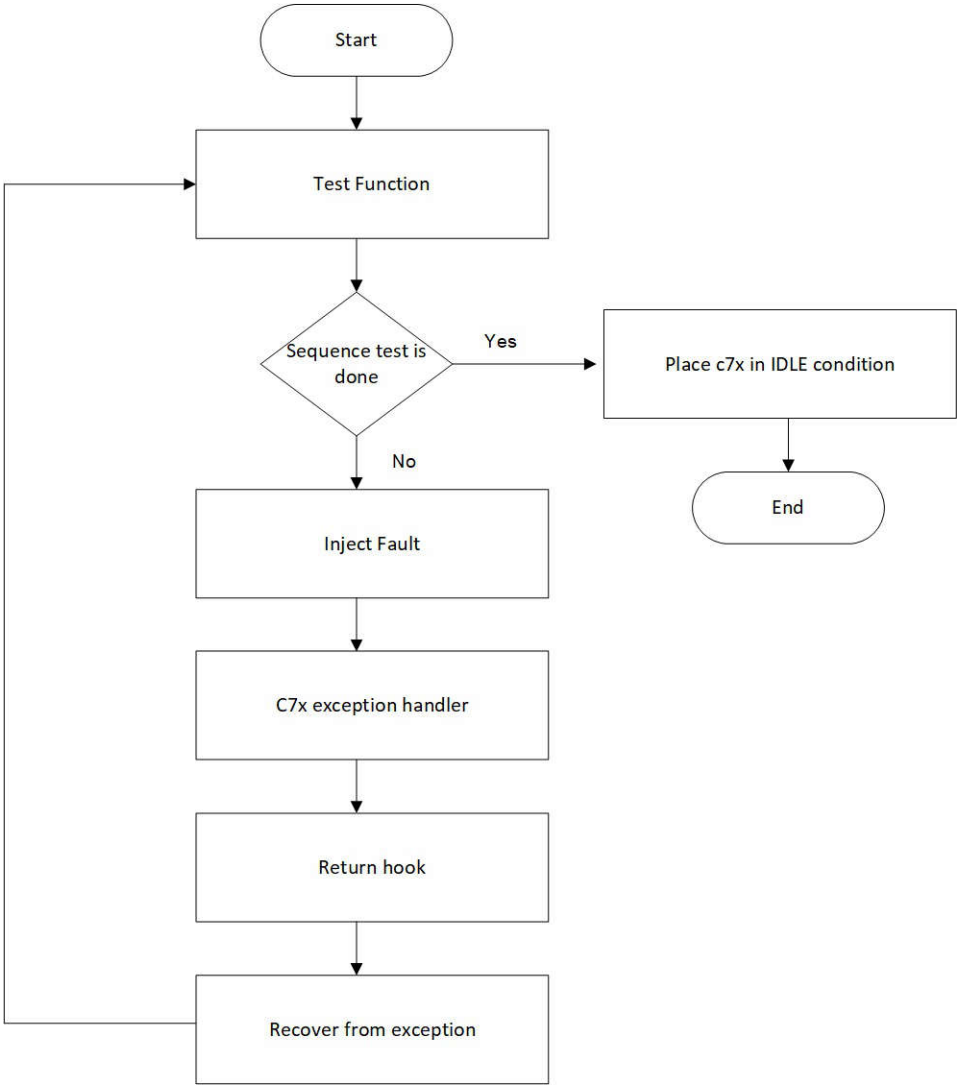

Figure 3-1. Test Sequence Flow Diagram

3.1 Code Changes

Before getting into implementation and code changes, there are certain prerequisites that must be followed.

Return hook must be registered. If the return hook is not registered, then return hook function is not executed after the exception handler.

3.1.1 Return Hook Definition

The registered return hook function must contain the below assembly code, to revive from the exception handler, but this is just for sequence testing. For regular flow, this must not be executed, as this generates a sequence of exception.

```
void reg_return_hook (void)
{
    __asm(" addkpc.d1 $PCR_OFFSET(test_fun), a0"); //Transfers the //test function
//address to a0.
    __asm(" ldd.d1 *ECSP[1], a1");//Status is saved in a1
    __asm(" rete.s1 a0,a1"); //rete, returns from the exception //to the beginning of the test
function, as //a0 contains the test //function address
}
```

3.1.2 Clear MMA Function

This function clears the MMA internal states so that another round of testing can be prepared.

```
void clear_mma(void)
{
     __HWA_CONFIG_REG_v1 mma_config_reg;
     mma_config_reg = __gen_HWA_CONFIG_REG_v1();
     mma_config_reg.PARITYCTRL = __MMA_NORMAL;
     __HWA_OFFSET_REG offset_reg;
     offset_reg = __gen_HWA_OFFSET_REG();
     __HWAOPEN(mma_config_reg,offset_reg,__MMA_OPEN_FSM_RESET);
     __HWAADV();
     __HWAADV();
     __HWAADV();
     __HWAADV();
     __HWACLOSE(0); //Clearing the first error code
}
```

3.1.3 Sequence Test

The following function verifies that there is a sequence of MMA error injection test. The following code is a sample of the end implementation is completely dependent by the user.

```
volatile unit8_t track_exception = 0;
void test_fun(void)
{
    switch(track_exception)   /*track exception is a global variable ,        initialized to zero*/
    {
        case 0:
            clear_mma(); /*this cleares the previous mma error code, definition is given above*/
            track_exception = track_exception+1;//increment track //exception
            overflow_exception();//generate exception
            break;
        case 1:
            clear_mma();
            track_exception = track_exception+1;
            appLogPrintf("Under flow exception\n"); /*Only for debug, remove*/
            underflow_exception(); // generate exception
            break;
        case 2:
            clear_mma();
            track_exception = track_exception+1;
            appLogPrintf("offset parity exception\n"); /*only for debug remove*/
            offset_parity_test();//generate excpetion
            break;
        case 3:
            clear_mma();
```

```
            appLogPrintf("config parity exception\n"); /*Only for debug*/
            track_exception = track_exception+1;
            config_parity_test();
            break;
        case 4:
            clear_mma();
            track_exception = track_exception+1;
            appLogPrintf("offset parity exception\n");/*Only for debug*/
            offset_parity_test();
            break;
        case 5:
            appLogPrintf("Signal ESM\n");
            appLogPrintf("Waiting for ESM event\n");
            while(1) /*it needs to wait, if not the program doesnt go , and it would generate
multiple exceptions*/
            {

            }
            break;
        default:
    //Handle any unlikely scenario
        break;
}
    return;
}
```

4 Summary

This document focuses on sequence of MMA fault injection and exception handling on the c7x DSP core.

This document does not discuss the recovery of the exception to execute the regular flow of the code after the sequence of error injection is done.

5 References

- Texas Instruments, *C71x DSP CPU, Instruction Set, and Matrix Multiply Accelerator*, user's guide.
- Texas Instruments, *J721S2, TDA4AL, TDA4VL, TDA4VE, AM68A Technical Reference Manual,* technical reference manual.

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265