Application Note

Expanding AM13x AI Capability with External RAM: SW and HW Design Instructions

ABSTRACT

Texas Instruments supports Edge AI applications with a wide range of tool chains and MCU microcontrollers for a variety of end-equipment. In the world of fast-moving Edge AI, memory must not be the challenge when users try to deploy the relatively “complex” edge AI model in MCU. This application note leverages the AM13x external peripheral interface (EPI), offloading the AI model in external SDRAM, to help users achieve the balance between performance, cost, and power consumption at a system level. The approaches and step-by-step guidance are presented in this application note. The demo project is available in AM13E230x MCU_SDK at "*am13e230x_sdk_26_01_00_00/examples/driverlib/epi/epi_sdram_read_write*," supported by SDK version 26.01.00, and tested with SDRAM AS4C32M16S [14].

Table of Contents

Trademarks

1 Introduction

TI supports the comprehensive toolchain to help users to deploy AI applications in different TI MCU series, such as AM13x, C2000, AM26x, MSPM0, and so forth. At a system level, the AI model must either produce the results with a low latency with a good accuracy, or the latency can take a hit, but maintain higher accuracy. This is a trade-off system architects typically encounter.

In scenarios where speed matters, the AI models are “light” and those models only need tens of kilobytes of flash and SRAM. The NPU engine can accelerate the AI model process to output the result in real-time; in cases where model size matters, the model deployment is limited by MCU on-chip memories, and even CPU is good enough for model processing. With that limitation, users must choose the processor level device, which makes the application infeasible due to high cost and power consumption.

TI AM13x MCUs solve this issue by using external peripheral interface (EPI), which can interface external SDRAM, ASRAM, SRAM, NOR flash memories to offload the AI model in external SDRAM. This feature brings maximum 64MB external SDRAM for AI model execution; the system cost and power consumption is also significantly reduced with this structure.

Apart from the advantages, there are two drawbacks to this design. The first is size. 32 pins are needed for EPI connection. The minimum package of AM13E230x which supports the 32-pin EPI signal is LQFP100. The outline of this package is 15.5mm × 15.5mm, so the SRAM size must also be taken into consideration. This makes this design impractical or applications that require small size.

The second drawback is speed. The AM13E230x on-chip NPU is not designed to access to the external SRAM, so the model offloaded to external SDRAM must be executed by the ARM® Cortex®-M33, running at 200MHz. The instruction access speed is also limited by EPI bus, which makes this design impractical for high-speed applications, such as applications that need AI output in real time with very low latency.

The techniques discussed in this note are not limited to EdgeAI applications, but similar approaches can be expanded to store and execute any application on the AM13x MCUs.

1.1 TI AM13E230x Introduction

AM13E230x microcontrollers (MCUs) are part of the AM13x highly integrated, low-cost, 32-bit MCU family based on the Arm Cortex-M33 32-bit CPU operating at up to 200MHz frequency. These real-time control-optimized MCUs offer high-performance analog, control, and digital peripheral integration. The MCUs provide up to 512KB of embedded flash program memory (two banks of up to 256KB) with built-in error correction code (ECC) and up to 128KB SRAM with hardware parity. Smaller memory configuration variants are also offered.

Enhanced communication interfaces are supported through one MCAN and up to six UNICOMM peripherals to support a combination of UART/LIN, I2C/SMBUS, and SPI. For connecting to external devices or memory, the high-speed EPI can connect to SDRAM or asynchronous RAM devices such as FPGA or ASIC.

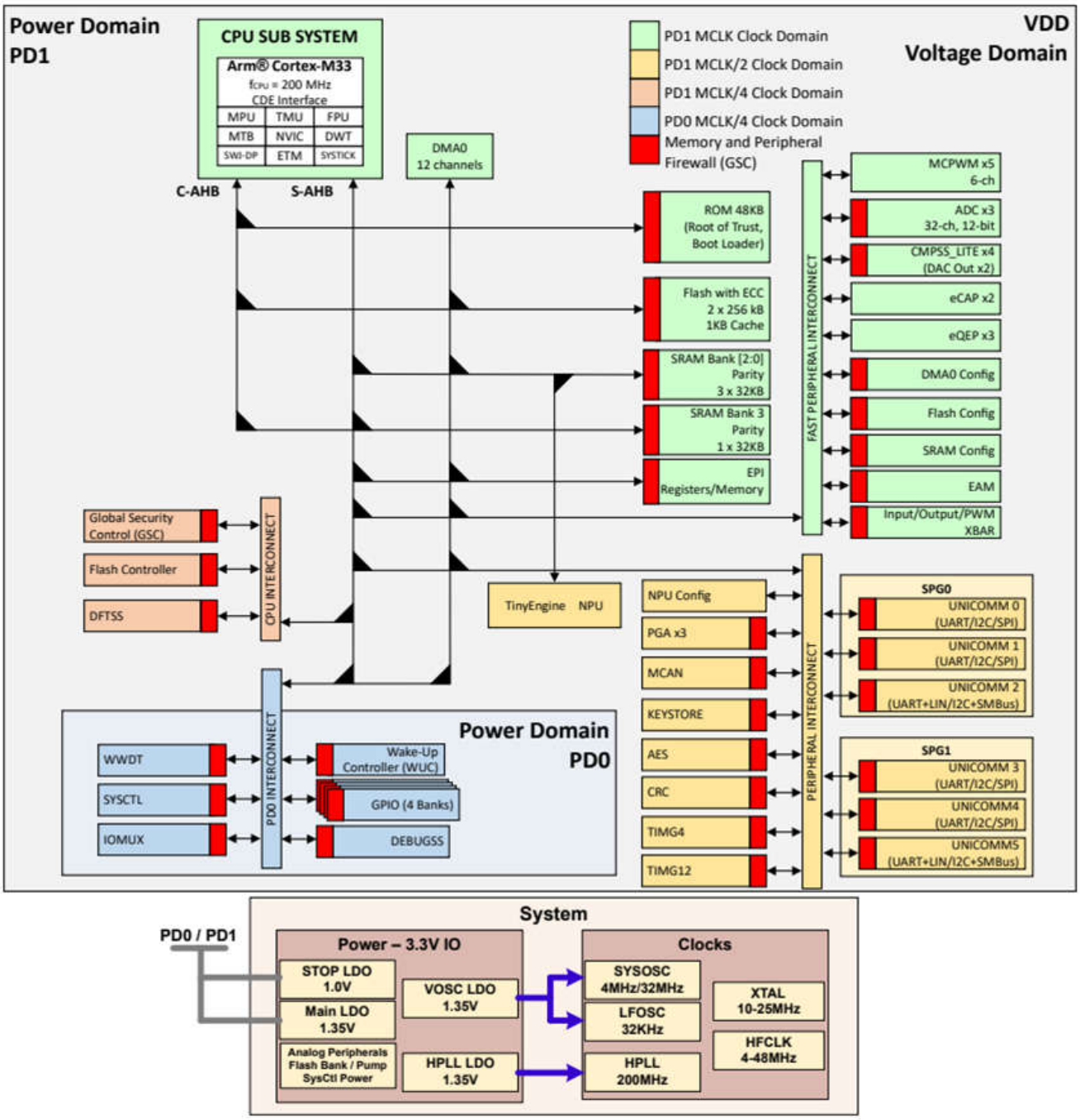

Figure 1-1. AM13E230x Functional Block Diagram

1.2 AI Toolchain for TI MCU Introduction

TI provides strong toolchain support for EdgeAI for MCUs. Customers with different experience levels can leverage TI Toolchain to build and deploy AI models on MCUs. Right from data acquisition to data labeling to training the AI model and compiling and deployment, the TI toolchain manages the entire process.

The user's guides for different tools are attached to the References section:

- tinyml-lensorlab release in git hub [1] and edge AI studio GUI release in TI.com official website [2] contains the models and the whole toolchain for AI model compiling validated by TI. Users can directly integrate these models into applications.. In git hub release, users can read the .py file directly to know how the toolchain works. Those models are all developed for speed, and no need for external SRAM, users can directly train

and validate the model with TI's launchpad. The MCU deployment example for those models are included in the individual MCU device SDKs.

- If users want to do some DIY AI model with the commonly used AI model developer such as PyTorch, TI Neural Network Compiler for MCUs [3] can support compiling the .onnx file into MCU .a library file. In this case, if the MCU on chip NPU must be used as accelerator, the model must be built with the NPU supported operation [4], and the TI-NPU QAT [5] must be applied to .onnx file, to make the model fit into NPU. If the model needs to be executed in CPU core, then the DQD is good enough for quantization. Users must validate the model carefully by themselves. The compiler also outputs the lib.c file for users to debug the input and output of each layer. Those highly customized models with different AI training structures can easily occupy large amounts of memories. One case is AI models using transformer structures, in which case users need external SDRAM for memory extending.

With both approaches, the final generated AI model that needs to be embedded into MCU project is a ".a" file (code and weights library file), and a ".h" header file to provide the interface of the library.

2 Detailed Description

This section guides users step-by-step to load the AI model file in MCU on chip flash and offload the model execution to external SDRAM.

2.1 Sysconfig Settings for EPI

This section is to generate the initialization code of EPI peripheral using sysconfig tool. Detailed specs for EPI IP can be found in the AM13E230 TRM [6] EPI section. The code generated can be found in file <ti_sdk_dl_config.c> and called in SYSCFG_DL_init function in main function. To read more about EPI in detail, please refer to the AM13x academy chapter for EPI attached in the References section.

There are 2 key specs to be configured: refresh count and EPI clock divider.

2.1.1 Refresh Count

The refresh count is based on the external clock speed and the number of rows per bank as well as the refresh period. The RFSH field represents how many external clock cycles remain before an AUTO-REFRESH is required. The normal formula is:

$$\text{RFSH} \leq (\text{t\_Refresh\_us} \ / \ \text{number\_rows}) \ / \ \text{ext\_clock\_period\_us} \quad (1)$$

A refresh period is normally 64ms, or 64000 us. The number of rows is typically 4096 or 8192. The ext_clock_period is a value expressed in us and is derived by dividing 1000 by the clock speed expressed in MHz. So, 50MHz is 1000 / 50 = 20ns, or 0.02us. A typical SDRAM is 4096 rows per bank if the system clock is running at 50MHz with an EPIBAUD register value of 0:

$$\text{RFSH} = (64000 \ / \ 4096) \ / \ 0.02 = \ 15.625\text{us} \ / \ 0.02\text{us} = 781.25 \quad (2)$$

The default value in the RFSH field is 750 decimal, or 0x2EE, to allow for a margin of safety and providing 15us per refresh. Note that this number must always be smaller or equal to what is required by Equation 2. If a number larger than allowed is used, the SDRAM is not refreshed often enough, and data is lost.

2.1.2 EPI Clock Divider

The EPI Controller SDRAM interface can operate up to 50MHz. The COUNT0 field in the EPIBAUD register configures the speed of the EPI clock. For main clock (MCLK) speeds up to 50MHz, the COUNT0 field can be 0x0000, and the SDRAM interface can run at the same speed as MCLK. However, if MCLK is running at higher speeds, the bus interface can run only as fast as half speed, and the COUNT0 field must be configured to at least 0x0001, configurations refer to Figure 2-1.

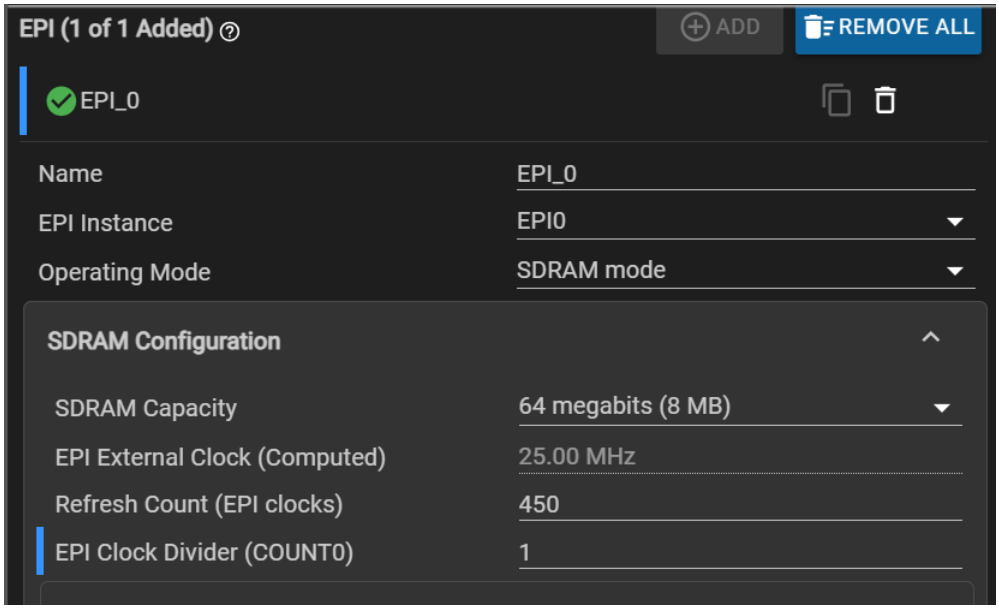

Figure 2-1. EPI Configuration Example #1

The EPI can assign the address for external RAM data and instructions, since there is no peripheral connected, there is no base address configuration needed. See Figure 2-2 for details. In SDRAM mode there is no need to configure the interrupt, the H/W pin mux is assigned automatically by sysconfig tool.

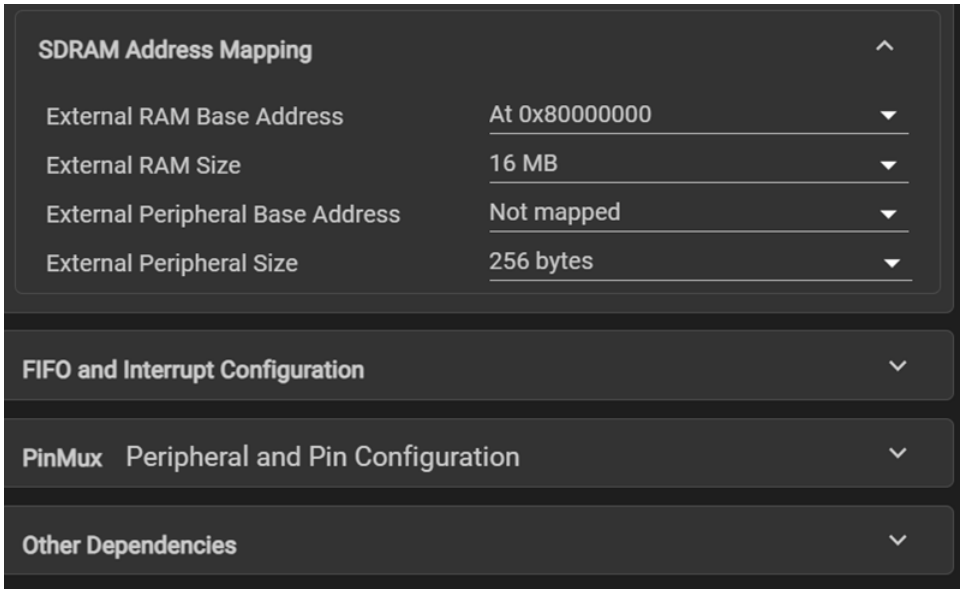

Figure 2-2. EPI Configuration Example #2

2.2 AI Model Library Deployment for External SRAM Execution

The attention matrices, weight matrices, and so forth in AI applications occupy most of the SRAM in MCUs. The matrices only live when the AI model start to run, and die after the application is ended. With this feature, the flash needs for AI model are much smaller than the RAM needs. By storing the AI model in MCU on chip flash and executing the AI model with external SDRAM, the AI model memory requirement can be met. This section guides users to use the compiler key words to realize this memory allocation.

As described in Introduction, the file needed to be integrated in the project for AI model is a ".a" library, containing tow sections: a .text section for AI functions and a .rodata section for weight and table. Users can use the linker section to store those sections in Flash as a load image and copied to SDRAM at boot.

All function addresses are assigned from the SDRAM address space (`0x80000000+`). Table 2-1 shows the memory allocation:

Table 2-1. Memory Allocation for AI Model Load in Flash and Run in External SDRAM

Flash (load image stored)		SDRAM (code executes from here)
0x000xxxxx	-> copyCode() ->	0x80000000 (set by syscfg)
.text (AI functions)		.text (PC fetches from here)
.rodata (weights and tables)		.rodata

The following sections are the step-by-step setups to realize the target memory allocation. The example uses “customer_ai_model.a” as the “.a” file name, which users can change accordingly. The basic AI embedded project refers to “ ..\am13e230x_sdk_<version>\examples\ai\generic_timeseries_classification” [11].

For the compiler symbol details and cmd file details, refer to <TI ARM CLANG COMPILER TOOLs USER’s GUIDE> [8] and <TI Linker Command File Primer> [9] in the References section. The demo project is available in AM13E230x MCU_SDK at "*am13e230x_sdk_26_01_00_00/examples/driverlib/epi/epi_sdram_read_write*."

2.2.1 Step 1: Add the AI “.a” File to the CCS Project

Copy `customer_ai_model.a` into the project folder to replace “libCMSISDSP_m33_ti_arm_clang.a” file.

In CCS: **Project Properties -> Build -> Arm Linker -> File Search Path**

- Add library filename: 'customer_ai_model.a'
- Add the folder path containing the '.a' file to the library search path

2.2.2 Step 2: Linker Script Change (`linker_m33_ti_arm_clang.cmd`)

Add MEMORY section 'EPI_CS2_EXEC_1' to the project. Adjust the 'length' to accommodate the AI model size of the customer. CMD file changes can be referred below:

```
MEMORY
{
    RAM_S         : ORIGIN = 0x20000000, LENGTH = 0x18000   // 96KB internal SRAM
    RAM_C         : ORIGIN = 0x00C18000, LENGTH = 0x8000    // 32KB code SRAM
    FLASH_BANK0   : ORIGIN = 0x00000000, LENGTH = 0x40000   // 256KB Flash
    FLASH_BANK1   : ORIGIN = 0x00040000, LENGTH = 0x40000   // 256KB Flash

    // External SDRAM through EPI – AI model code + weights execute from here
    // Customer with 300KB AI model: increase to 0x0080000 (512KB) or 0x0100000 (1MB)
    EPI_CS2_EXEC_1(RWX) : origin = 0x80000000, length = 0x0010000  // 64KB (demo size)
}
```

Add .CS1_EXECUTE section and replace `libCMSISDSP_m33_ti_arm_clang.a` with “customer_ai_model.a” filename. CMD file changes can be referred below:

```
.CS1_EXECUTE : LOAD = FLASH_BANK0, palign(8),
               RUN  = EPI_CS2_EXEC_1,
               LOAD_START(emif1_cs2_sec1_loadstart),
               LOAD_SIZE(emif1_cs2_sec1_loadsize),
               RUN_START(emif1_cs2_sec1_runstart),
               RUN_SIZE(emif1_cs2_sec1_runsize)
{
    // Replace with the customer's AI .a filename.
    // The <*> wildcard grabs ALL objects in the archive automatically.
    customer_ai_model.a<*>(.text .text.* .rodata .rodata*)
}
```

Each keyword meaning is explained in Table 2-2:

Table 2-2. Keyword Meanings in Link Command File Modification

Keyword	Meaning
LOAD = FLASH_BANK0`	AI model binary image stored in Flash at link time
RUN = EPI_CS2_EXEC_1	All AI function addresses assigned from SDRAM (`0x80000000+`)

Table 2-2. Keyword Meanings in Link Command File Modification (continued)

Keyword	Meaning
LOAD_START (emif1_cs2_sec1_loadstart)	Linker symbol = Flash address where image starts
LOAD_SIZE (emif1_cs2_sec1_loadsize)	Linker symbol = number of bytes to copy
RUN_START (emif1_cs2_sec1_runstart)	Linker symbol = SDRAM destination (`0x80000000`)
<*> (.text .text.* .rodata .rodata*)	Wildcard: captures ALL code and weights from the archive

2.2.3 'main.c' Changes

This section explains the main.c file changes to deploy the project.

2.2.3.1 Include Headers

The below header files must be included in the project:

```
#include "device.h"
#include "log.h"
#include "stdint.h"
#include "dl_epi.h"
#include "dl_epi.c"           /* EPI driver – included directly              */
#include "basic_math_functions.h"  /* CMSIS DSP – defines float32_t           */
#include "ti_sdk_dl_config.h"
```

2.2.3.2 Declare Linker-Generated Symbols

These four symbols are generated automatically by the linker from the section attributes. In C, '&symbol' gives the value the linker assigned.

```
extern unsigned int emif1_cs2_sec1_loadstart;  // Flash address of load image start
extern unsigned int emif1_cs2_sec1_loadsize;   // Size of load image in bytes
extern unsigned int emif1_cs2_sec1_runstart;   // SDRAM destination (0x80000000)
extern unsigned int emif1_cs2_sec1_runsize;    // Size of section at run time
```

2.2.3.3 memcpy Function

This function copies the section from Flash to SDRAM using the TI linker symbol convention. '&sdram_code_loadstart' gives the value the linker assigned.

```
memcpy(&sdram_code_runstart,&sdram_code_loadstart,(size_t)&sdram_code_loadsize);
```

2.2.3.4 Place the Entry Function in the SDRAM Section

In the demo project, "ExecuteFromSDRAM()" is the function that executes from SDRAM an dcall s"arm_dot_prod_f32()." The function is placed in ".CS1_EXECUTE through "__attribute__."

```
/* This function executes from SDRAM. memcpy() must be called first. */
__attribute__((section(".CS1_EXECUTE"), __noinline__))
void ExecuteFromSDRAM(void)
{
    LOG("Executing from SDRAM!\r\n");

    /* Call arm_dot_prod_f32 – this function is also in .sdram_code (SDRAM).
     * Input data lives in RAM_S and is accessed through the normal data bus.  */
    arm_dot_prod_f32(App_DotProd_SrcA,
                     App_DotProd_SrcB,
                     APP_DOT_PROD_SIZE,
                     &App_DotProd_Result);
}
```

2.2.3.5 main() Boot Sequence

This is the main function sequence to properly boot the MCU and start to run the AI model with external SDRAM.

```
int main(void)
{
    /* Step 1: Hardware init – Device_Init() sets up clocks and cache.
     * SYSCFG_DL_init() configures the EPI peripheral, making SDRAM
     * accessible at 0x80000000. Both must run before any SDRAM access. */
    Device_Init();
    SYSCFG_DL_init();

    /* Step 2: Copy .CS1_EXECUTE section from Flash to SDRAM.
     * Transfers all AI model functions and weights from their Flash
     * load addresses to their SDRAM run addresses (0x80000000+).
     * After this call, all AI functions execute from SDRAM. */
    memcpy(&sdram_code_runstart,&sdram_code_loadstart,(size_t)&sdram_code_loadsize);


    /* Step 3: Call AI inference – PC moves to 0x80000000+ region.
     * In the demo project this is executeFromCs0().
     * For customer: replace with the AI model inference entry point. */
    ExecuteFromSDRAM();/* customer replaces this with their AI inference call */


    /*Verify dot product result. */
    LOG_ASSERT((App_DotProd_Result == APP_DOT_PROD_EXPECTED),
               "arm_dot_prod_f32 result mismatch: expected 20.0\r\n");
    LOG("PASS: arm_dot_prod_f32 = %.1f – executed from SDRAM\r\n",
        App_DotProd_Result);

    LOG("Example PASS\r\n");

    return 0;
}
```

2.2.4 Step 4: CSS Project Validation

To validate the approach before applying to the customer's AI `.a` file, the CMSIS DSP library (`libCMSISDSP_m33_ti_arm_clang.a`) is used as a substitute. This library contains `arm_dot_prod_f32(),' which is a multiply-accumulate function that is part of `BasicMathFunctions.c.obj` inside the archive.

2.3 H/W Design Best Practices

This section explains the general guidelines for EPI H/W design and how to better realize the requirements with Altium. Users can refer to the TM4C EPI reference design document [12] for the project files.

The AM13E230 hardware design guide [13] lists the full guidance for MCU H/W design. Here are the EPI guidelines to follow:

- Place the target device (memory IC or FPGA) close to the MCU to keep trace lengths short
- Use 50Ω single-ended traces to control impedance
- Minimize reflections and crosstalk by routing signals groups on the same layer and minimize vias and layer transitions
- Match the following trace lengths to reduce timing skew:
  - Data bus: match within ±25mils
  - Address and command signals: match within ±25mils
  - Control signals: match within ±25mils to ±50mils
  - Clock signals: match within ±5mils to ±10mils to the longest data trace
- Route address and command signals in parallel
- Avoid stubs and keep routes as direct as possible
- Consider series termination resistors (22Ω to 33Ω) for data and clock lines, placed close to the transmitter.

Note there is a recommendation for single trace from EPI pin to address/data, then to other pin to minimize the signals reflections as shown in Figure 2-3.

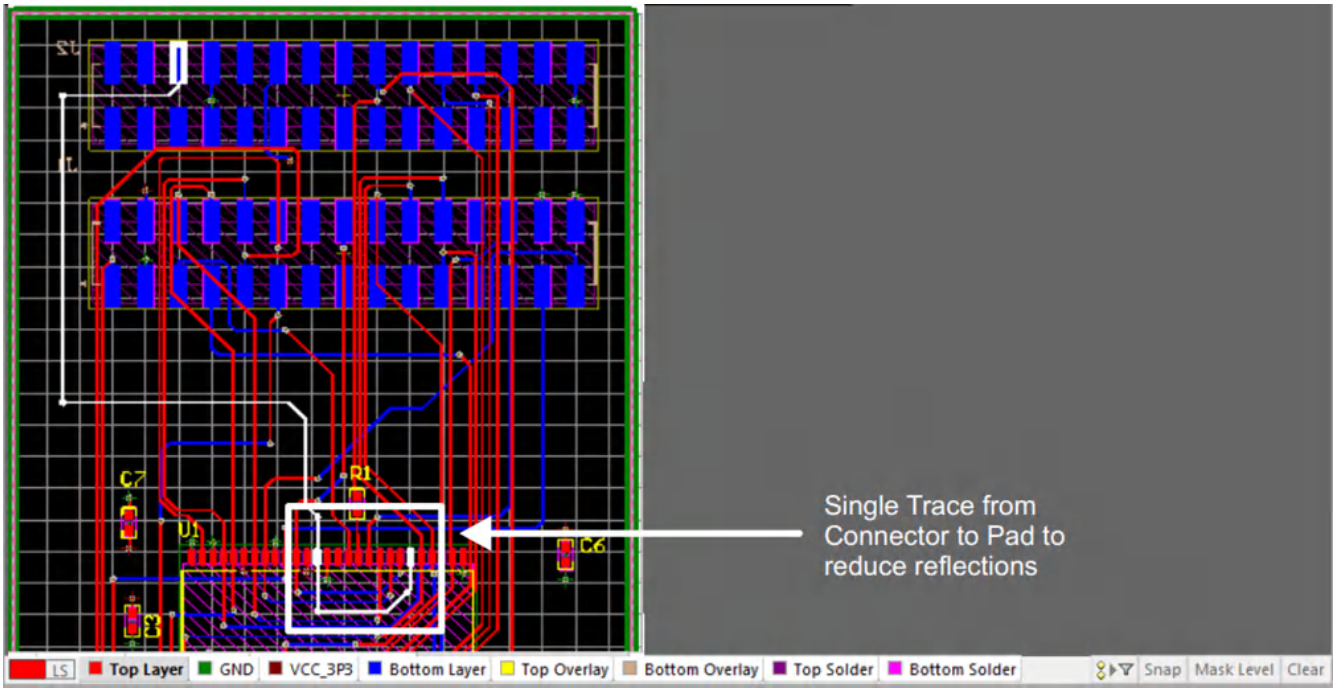

Figure 2-3. Single Trace from EPI Pin to Address/Data

To realize the H/W design easier with Altium:

- The address and data traces must use two different net names so user is able to utilize length tuning to match the trace lengths. If the net name is currently the same between both (after the resistor), then this becomes difficult to set up in Altium. Utilizing different trace names between both is recommended and when the traces meet at the EPI pin of the AM13E23019, the user can either use a resistor to short them together or use a net tie component to short the nets together at the pin.

Users can utilize trace matching by using the following steps in Altium:

- Go to Design > Rules from the top menu to open the *PCB Rules and Constraints* editor.
- Expand the *High-Speed* category and select *Matched Lengths*.
- Right-click and select *New Rule* to name and configure the rules.
- Under the *Where the Object Matches* scope, select the specific net, xSignal, or Net Class to match.
- Under the *Constraints* section, set up your tolerance limit. This defines the maximum allowable difference between the shortest and longest trace in the group.
- The user can then combine all the nets together in this tolerance group and set the tolerance limit to 25mil.

Additionally, placing the resistors adjacent to corresponding SDRAM pins and keeping the distance common between all pins and the corresponding resistors is the recommended operation, so users do not need to do trace matching for that portion of the trace.

3 Summary

This application note shares a method to offload the AI model to external SRAM with AM13E230 EPI. This method solves the memory shortage issue on AI model deploying in MCU, helping users to achieve the balance between performance, cost, and power consumption at a system level. The approaches and step-by-step guidance are presented in this note, and the demo project is available in AM13E230x MCU_SDK at "*am13e230x_sdk_26_01_00_00/examples/driverlib/epi/epi_sdram_read_write*."

4 References

1. Texas Instruments, Tiny ML Tensorlab User Guide, user guide.
2. Texas Instruments, EDGE-AI-STUDIO.
3. Texas Instruments, TI Neural Network Compiler for MCUs User's Guide - 2.1.2 LTS, user's guide.
4. Texas instruments, Layer Configurations Supported on the NPU.
5. Texas Instruments, TI-NPU QAT.
6. Texas Instruments, AM13E230x Microcontrollers, technical reference manual.
7. Varahmihir, Arpit. External Peripheral Interface (EPI).
8. Texas Instruments, The SECTIONS Directive.
9. Texas Instruments, TI Linker Command File Primer.
10. Texas Instruments, AM13E23019: EPI Code Execution Demo.
11. Texas Instruments, AM13E2X-SDK.
12. Texas Instruments, Interfacing SDRAM on High-Performance Microcontrollers Design Guide, design guide.
13. Texas Instruments, AM13E230x Hardware Design Guidelines, application note.
14. Alliance Memory, AS4C32M16SB Datasheet, datasheet.

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), 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 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, TI’s General Quality Guidelines, 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. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

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

Last updated 10/2025