APEX-2 emulation library user guide

ABSTRACT:
This is the APEX-2 emulation library user guide document applicable for AMCU_SW_S32V234
KEYWORDS:
APEX-2, APU-C, ACF, emulation
APPROVED:

AUTHOR	SIGN-OFF SIGNATURE #1	SIGN-OFF SIGNATURE #2
Igor Aleksandrowicz		

Revision History

VERSION	DATE	AUTHOR	CHANGE DESCRIPTION
0.1	2014-03-13	Igor Aleksandrowicz	First draft
0.2	2014-03-17	Igor Aleksandrowicz	ACF differences section added
0.3	2014-04-02	Igor Aleksandrowicz	Header files names updated
0.4	2014-06-20	Igor Aleksandrowicz	Removed the data width limitation
0.5	2015-04-24	Anca Dima	Adapted to new product name

Table of Contents

LIST OF TABLES

1 Introduction

1.1 Purpose

The purpose of this document is to describe the APEX-2 emulation library user interface as well as the important design details of the library. It is intended to serve as a reference source for future APEX-2 emulation library development and maintenance. For the description of the emulated functionality itself refer to [2] and [3]. For the exact definitions and implementation details refer to [1].

1.2 Audience Description

This document is intended to be used by APEX-2 developers familiar with APU-C and ACF in scope of [2] and [3].

1.3 References

Id	*Title*	*Location*
[1]	*APEX2_emu source code documentation*	*APEX2_emu source code.*
[2]	*APU-C documentation*	*S32V234 SDK*
[3]	*ACF documentation*	*S32V234 SDK*

Table 1 References Table

1.4 Definitions, Acronyms, and Abbreviations

Term/Acronym	*Description*
ACF	*APEX Core Framework*
APU	*Array Processing Unit*
CMEM	*Computational (vector) memory*
CU	*Computational Unit*

Table 2 Acronyms Table

1.5 Document Location

<VISION_SDK >/s32v234_sdk/docs/

1.6 Problem Reporting Instructions

Problems with or corrections to this document should be reported by e-mail to Anca Dima Anca.Dima@freescale.com

2 Functional Description

The APEX-2 development library aims to provide a way of emulating the APEX-2 behaviour on the source code level. It doesn't simulate the real hardware in any way. The library is designed to be platform-agnostic requiring a C++ compiler only.
The library consists of two parts:

- APU-C language syntax library
  Brings the vector types and instructions present in APU-C to standard C/C++.
- ACF emulation library
  Emulates the ACF kernel, graph and process functionality.

The functionality support will be described in this section. For definitions of the APU-C and ACF constructs themselves refer to the APEX-2 documents [2] and [3].

2.1 APU Emulation Functionality

2.1.1 Scalar Types

The scalar types int08u, int08s, int16u, int16s, int32u, int32s used in APU-C are defined in apu_config.hpp using stdint types.

2.1.2 Vector Types

Vector types are implemented in the apu_vec.cpp and apu_vec.hpp files as APU_vec<T> template where T is a type of the vector element. The user can instantiate this template for variety of types but only vec08u, vec08s, vec16u, vec16s, vec32u and vec32s are defined in APU-C. These six types are also defined in apu_vec.hpp using the scalar types defined in apu_config.hpp as their elements' type. All vector operations are masked according to the state of the vector condition stack, i.e. they work correctly when used with the vif, velse, vendif statements. The number of CUs emulated is defined as VSIZE in apu_config.hpp. The user is free to change it.
Vector operations supported:

- binary arithmetic operators: +, -, *, /, %
- unary arithmetic operators: +, -
- binary bitwise operators: ^, |, &, <<, >>
- unary bitwise operators: ~
- relational operators: ==, !=, >, <, >=, <=
- assignment operator: =
- compound assignment operators: +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
- prefix and postfix operators: ++, --
- element access functions: vget, vput
- vector move shift functions: vml, vmsl, vmrl, vmr, vmsr, vmrr
- indirect memory access functions: vstore, vload
- arithmetic functions: vadd, vaddx, vsub, vsubx, vmul
- bitwise functions: vand, vor, vxor, vnot, vcomplement
- bit shift functions: vsl, vsr, vsll, vsra, vsrl
- comparison functions: vseq, vsne, vsge, vsgt, vsle, vslt
- combination functions: vselect, vswap

For 16-bit vectors only:

- specialized shifts: vsllx, vsrax, vsrlx
- specialized multiplications: vmul_ulul, vmul_uluh, vmul_ ulsh… vmul_shsh

2.1.3 Vector Boolean Type

The vector boolean type vbool is implemented in the apu_vbool.cpp and apu_vbool.hpp files. All vector boolean operations are masked according to the state of the vector condition stack, i.e. they work correctly when used with the vif, velse, vendif statements. The number of CUs emulated is defined as VSIZE in apu_config.hpp. The user is free to change it.
Vector boolean operations supported:

- binary logical operators: &&, ||
- unary logical operators: !
- relational operators: ==, !=
- assignment operator: =
- element access functions: vget, vput
- vector move shift functions: vmsl, vmrl, vmsr, vmrr
- gathering functions: vall, vany

2.1.4 Vector Conditional Execution

The APU-C simulation library supports APU vector conditional execution. The mechanism is implemented in the apu_cond.cpp and apu_cond.hpp files as VectorConditionController singleton class. Internally, it's implemented as a stack of conditions where the top of the stack influences all the vector operations and the vector boolean operations. Users shouldn't use methods of VectorConditionController directly but should use the APU-C constructs vif, velse and vendif. These constructs are implemented as macros performing following operations:

- vif pushes a condition onto the stack (which is limited to 7 conditions as in the APU)
- velse negates the element on top of the stack (the top of the stack is prevented from being negated multiple times so as not to allow multiple velse between a vif vendif pair)
- vendif pops the conditions from the stack (each vif is required to be matched by vendif)

2.1.5 Data Layout Functions

ACF kernels expect the image data in CMEM to be in a specific format. The image is sliced vertically into horizontal slices that can fit into CMEM. Each slice is then tiled into CU chunks and each CU can only see its chunk data. See figure 1.

Raw input image:

1	2	3	4	5	6	7	8	9	10	11	12
13	14	15	16	17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32	33	34	35	36
37	38	39	40	41	42	43	44	45	46	47	48

Tiled image:

CU0			CU1			CU2			CU3		
1	2	3	4	5	6	7	8	9	10	11	12
13	14	15	16	17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32	33	34	35	36
37	38	39	40	41	42	43	44	45	46	47	48

CMEM:

CU0	CU1	CU2	CU3
1	4	7	10
2	5	8	11
3	6	9	12
13	16	19	22
14	17	20	23
15	18	21	24
25	28	31	34
26	29	32	35
27	30	33	36
37	40	43	46
38	41	44	47
39	42	45	48

Figure 1: Example CMEM layout for an image slice assuming 4 CUs

Many kernels require the CUs to sometimes access data not present in their respective chunks (e.g. 3x3 kernel needs to access the whole 3x3 pixel neighborhood for each pixel processed). While it's possible to solve that by using vector move shifts, the ACF supports data padding, i.e. including external pixels in the CU chunks themselves. These pixels are generally not processed by the CU but they are read. See figure 2.

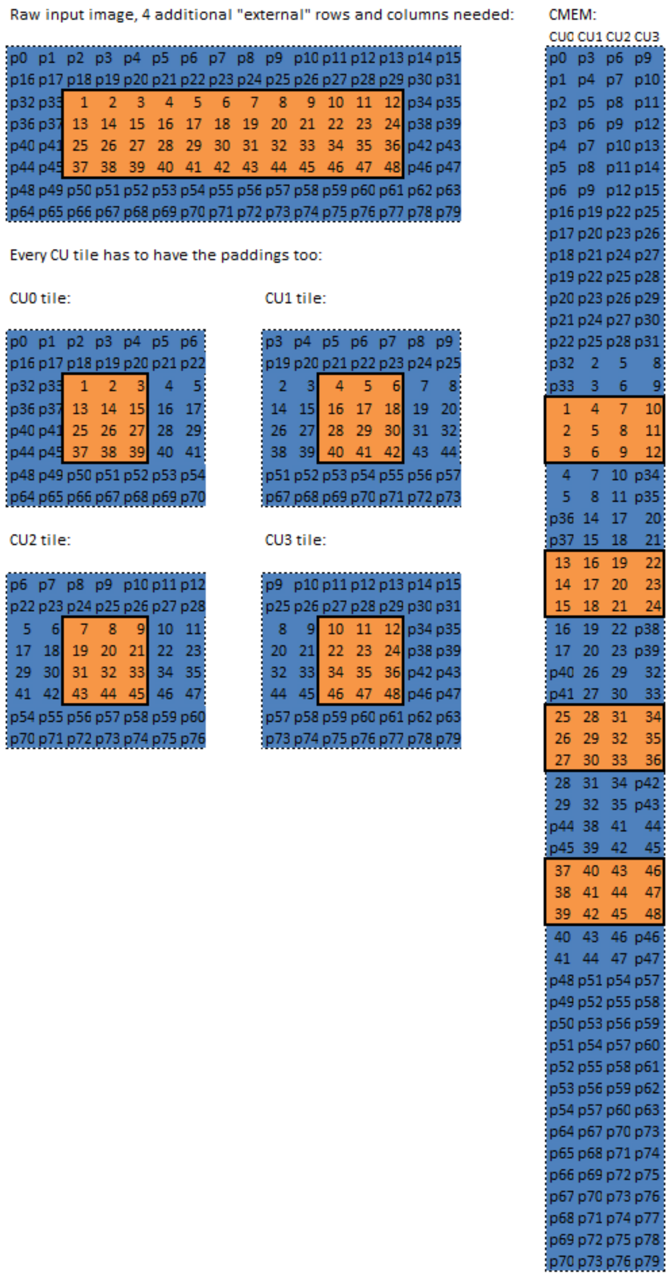

Figure 2: Example CMEM layout with padding for an image slice assuming 4 CUs

The APU-C simulation library includes several data layout transformation function templates in the `apu_extras.hpp` file. The functions are not aware of the vector types and they see the data as arrays of scalar values. Users should use them with the `int08u`, `int08s`, `int16u`, `int16s`, `int32u`, `int32s` types and cast to the appropriate vector types. All pointers passed to these functions should point to the first data array element (i.e. they shouldn't point to the first non-padding pixel as is often the case in the kernel functions).

- **`ArrayToCMEMDataSym, ArrayToCMEMData`** transform the data from the plain array format to the tiled CMEM format with optional padding
- **`ArrayToCMEMDataIndirect`** transforms the data from the plain array format to the tiled CMEM format, the CU chunks are offset by the user-provided offsets
- **`CMEMDataToArray`** transforms the data from the plain array format to the tiled CMEM format with optional padding
- **`SrcImageArraySize, DstImageArraySize`** compute the size of an array needed to store the plain image input/output data based on the CU chunk configuration
- **`CMEMArraySize`** computes the size of an array needed to store the data in the tiled CMEM format
- **`AddPaddingCMEM, RemovePaddingCMEM, ChangePaddingCMEM`** change the padding of data in the tiled CMEM format

2.1.6 Vector Operations Cycle Counting

APU-C emulation library supports cycle count estimation. It's implemented on the vector function level, the real result may be different because it depends on an assembly code produced by the APU-C compiler and ignores scalar instructions.
Vector operations, vector boolean operations and vector condition statements increase the cycle counter based on the information provided in the `apu_cycle_database.hpp` file. The counter value can be obtained by calling the `GetVectorCyclesElapsed` function declared in the `apu_cycle_database.hpp` file. Alternatively, one can use the `GetVectorOperationsElapsed` function to obtain the number of vector operations performed. The cycle counting functionality can be disabled by setting the `COUNT_CYCLES` define to 0 in the `apu_config.hpp` file which results in significantly faster code execution.

2.1.7 APU-C Simulation Library And APU-C Differences

There are three differences between the library and the APU-C compiler that users should be aware of:

- 16-bit and 32-bit vector values are organized byte-by-byte in CMEM:

  0_LO 1_LO 2_LO 3_LO …
  0_HI 1_HI 2_HI 3_HI …

  These values are organized element-by-element when using the library:

  0_LO 0_HI 1_LO 1_HI 2_LO 2_HI 3_LO 3_HI …

  This difference has no effect on the code except when using x-bit wide type pointers for traversing y-bit wide type data when x, y are different.
- Users should always use `vendif` for ending a vector conditional block. The APU-C compiler seems to ignore `vendif` statements.
- One should always be explicit about the types when writing expressions mixing different vector types together, e.g. this code will work differently in APU-C and in the emulation library:

```
vec08u five = 5;
vec08u nine = 9;
vec16s dif = five - nine;
```

The type of the `five - nine` subexpression is `vec16s` in APU_C but it's `vec08u` in the simulation library.

2.2 ACF Emulation Functionality

2.2.1 ACF Kernels

The ACF emulation library implements both the metadata specification and the kernel entry point specification. Both can be defined after including the `acf_kernel.hpp` file. Kernel entry point functions support up to 16 I/O ports. Kernels themselves have to be registered by using the `REGISTER_ACF_KERNEL(METADATA, FUNCTION)` macro from the `acf_lib.hpp` file before they can be used in a graph.

2.2.2 ACF Graphs

The library supports graph creation via the `ACF_Graph` base class from the `acf_graph.hpp` file. Graphs can be created by inheriting from this class and implementing the Create method the same way as in the ACF.

2.2.3 ACF Process Descriptors

A process descriptor can be created by inheriting from the `ACF_Process_Desc_APU` class from the `acf_process_desc_apu.hpp` file and implementing the Create method the same way as in the ACF.

2.2.4 ACF Data Descriptors

Data descriptor class named `DataDescriptor` which can be used for connecting process inputs and outputs can be found in the `acf_data_descriptor.hpp` file. It always uses the standard virtual memory allocation whereas its ACF counterpart can allocate physically contiguous memory regions for the process inputs and outputs.

2.2.5 ACF Processes

A process type should be created by using the `REGISTER_PROCESS_TYPE(NAME, DESC)` macro from the `acf_lib.hpp` file. A process can be created as an instance of the registered type. The following ACF process functions are supported: `Initialize`, `ConnectIO`, `ConnectIndirectInput`, `Start`, `Wait`. In addition, two error reporting functions are available and ACF kernel, ACF graph and ACF process errors can be detected:

- **`ErrorOccured`** tells whether any error occurred.
- **`ErrorMessage`** returns the error message. Only the first error encountered is reported.

Two other functions can be used for debugging:

- **`GetExecutionPlanDescription`** returns a description of the steps the library takes when executing the process.
- **`GetDataPlacePtr`** returns a pointer to an `ACF_Graph::DataPlace` structure containing an intermediate result corresponding to a kernel/graph input or output.

2.2.6 ACF Simulation Library and ACF Differences

There are differences between the library and the ACF that users should be aware of:

- Regions of interest (ROI) are supported in the library, but they are treated as standalone images, i.e. padding at the roi borders is performed by mirroring the roi-border pixels. No check of alignment to the chunk size is performed.
- Indirect inputs’ chunk sizes have not to be defined by the SetInputChunkSize function in the emulation library.
- Padding doesn’t work correctly when using indirect inputs at the moment (it replicates the pixels of the offset image instead of the source image), use larger chunk sizes instead.

3 File Structure

3.1 APU-C Emulation Files (`emu/apu/src`):

File name	Description
`apu_cond.cpp` `apu_cond.hpp`	Vector conditional execution (vif, velse, vendif) implementation.
`apu_config.hpp`	Library configuration (CU count, cycle counting, typedefs for the APU-C scalar types).
`apu_cycle_database.hpp`	Cycle count database.
`apu_cycles.cpp` `apu_cycles.hpp`	Cycle counting implementation.
`apu_extras.hpp`	Extra functions, mainly data layout transform functions.
`apu_includes.hpp`	
`apu_lib.hpp`	Main library header, users should include this file.
`apu_vbool.cpp` `apu_vbool.hpp`	Vector Boolean type implementation.
`apu_vec.cpp` `apu_vec.hpp`	Vector data types implementation (vec08u, vec08s, vec16u, vec16s, vec32u, vec32s).

3.2 ACF Emulation Files (`emu/acf/src`):

File name	Description
`acf_data_descriptor.cpp` `acf_data_descriptor.hpp`	ACF data descriptor implementation.
`acf_graph.cpp` `acf_graph.hpp`	ACF graph creation implementation.
`acf_kernel.cpp` `acf_kernel.hpp`	ACF kernel creation implementation.
`acf_lib.hpp`	Main library header, users should include this file.
`acf_process_desc_apu.cpp` `acf_process_desc_apu.hpp`	ACF process creation and execution implementation.
`acf_template_apu_process_desc.hpp`	

4 Usage

Both the emulation libraries use the APEX2 namespace, it's encouraged to use the `using` directive with the namespace or the individual namespace members used to maximize the source-level compatibility with the ACF code. There are several header files which have to be included in source files when writing APU-C or ACF code. These are located in the @VSDK/tools/emu directory:

From the APU library under the @VSDK/tools/emu/apu/src :

- **`apu_lib.hpp`** when writing the code in the APU-C language intended to run on the APU.
- **`apu_extras.hpp`** when using the data layout transformation functions.
- **`apu_cycles.hpp`** when using the cycle counting estimation functionality.

From the ACF library under the @VSDK/tools/emu/acf/src :

- **`acf_kernel.hpp`** when implementing the kernel including its metadata.
- **`acf_graph.hpp`** when implementing an ACF graph.
- **`acf_process_desc_apu.hpp`** when implementing an ACF process description.
- **`acf_lib.hpp`** when writing the host code using ACF processes.

4.1 ACF Kernel Example:

```
#ifdef APEX2_EMULATE

#include "apu_lib.hpp"
#include "acf_kernel.hpp"

using namespace APEX2;
#endif

#ifdef ACF_KERNEL_METADATA

KERNEL_INFO _kernel_info_rgb_to_grayscale
(
   "apu_rgb_to_grayscale",
   2,
   __port(__index(0),
          __identifier("INPUT"),
          __attributes(ACF_ATTR_VEC_IN),
          __spatial_dep(0,0,0,0),
          __e0_data_type(d08u),
          __e0_size(3, 1),
          __ek_size(1, 1)),
   __port(__index(1),
          __identifier("OUTPUT"),
          __attributes(ACF_ATTR_VEC_OUT),
          __spatial_dep(0,0,0,0),
          __e0_data_type(d08u),
          __e0_size(1, 1),
          __ek_size(1, 1))
);

#endif //#ifdef ACF_KERNEL_METADATA

#ifdef ACF_KERNEL_IMPLEMENTATION


void rgb_to_grayscale(vec08u* apDest, const vec08u* apcSrc,
                      int aBlockWidth, int aBlockHeight, int aInputSpan);
```

```
void apu_rgb_to_grayscale(kernel_io_desc lIn0, kernel_io_desc lOut0)
{
   vec08u* lpvIn0  = (vec08u*)lIn0.pMem;
   vec08u* lpvOut0 = (vec08u*)lOut0.pMem;

   rgb_to_grayscale(lpvOut0, lpvIn0, lIn0.chunkWidth, lIn0.chunkHeight, lIn0.chunkSpan);
}

void rgb_to_grayscale(vec08u* apDest, const vec08u* apcSrc,
      int aBlockWidth, int aBlockHeight, int aInputSpan)
{
  for (int16s y = 0; y < aBlockHeight; ++y)
  {
    int16s x3 = 0;
    for (int16s x = 0; x < aBlockWidth; ++x)
    {
      vec16u accum = 27 * vec16u(apcSrc[x3]) + 92 * vec16u(apcSrc[x3+1]) + 9 * vec16u(apcSrc[x3+2]);
      apDest[x] = vec16u(accum>>7);
      x3 += 3;
    }

    apDest += aBlockWidth;
    apcSrc += aInputSpan;
  }
}

#endif //#ifdef ACF_KERNEL_IMPLEMENTATION
```

4.2 ACF Graph Example:

```
#ifdef APEX2_EMULATE
#include "acf_graph.hpp"
using namespace APEX2;
#endif


class apu_rgb_to_grayscale_graph : public ACF_Graph
{
public:

   void Create()
   {
      //set identifier for graph
      SetIdentifier("apu_rgb_to_grayscale_graph");

      //add kernel instances
      AddKernel("rgb_to_grayscale_0", "apu_rgb_to_grayscale");

      //add graph ports
      AddInputPort("INPUT");
      AddOutputPort("OUTPUT");

      //specify connections
      Connect(GraphPort("INPUT"), KernelPort("rgb_to_grayscale_0", "INPUT"));
      Connect(KernelPort("rgb_to_grayscale_0", "OUTPUT"), GraphPort("OUTPUT"));
   }
};
```

4.3 ACF Process Descriptor Example:

```
#ifdef APEX2_EMULATE
#include "acf_process_desc_apu.hpp"
using namespace APEX2;
#endif

#include "apu_rgb_to_grayscale_graph.hpp"

class apu_rgb_to_grayscale_apu_process_desc : public ACF_Process_Desc_APU
{
public:

   void Create()
   {
      Initialize(mGraph, "APU_RGB_TO_GRAYSCALE");

      SetInputChunkSize("INPUT", 8, 8);
   }

   apu_rgb_to_grayscale_graph mGraph;
};
```

4.4 ACF Host Program Example:

```
#include <opencv2/opencv.hpp>
#include <iostream>

#ifdef APEX2_EMULATE
#include "apu_lib.hpp"
#include "acf_lib.hpp"
using namespace APEX2;
#endif

#include "apu_rgb_to_grayscale_apu_process_desc.hpp"

using namespace cv;
using namespace std;

//for kernel metadata and entry function visibility
extern KERNEL_INFO _kernel_info_rgb_to_grayscale;
void apu_rgb_to_grayscale(kernel_io_desc lIn0, kernel_io_desc lOut0);

int main(int argc, _TCHAR* argv[])
{
  //Register the kernel and the process
  REGISTER_ACF_KERNEL_kernel_info_rgb_to_grayscale, apu_rgb_to_grayscale)
  REGISTER_PROCESS_TYPE(APU_RGB_TO_GRAYSCALE, apu_rgb_to_grayscale_apu_process_desc)

  Mat inColor = imread("in.png", CV_LOAD_IMAGE_COLOR);
  imshow("input", inColor);

  if (in.empty() || inColor.empty())
  {
    cout << "could not open the input files " << endl;
    return -1;
  }

  cout << "rgb to grayscale: " << endl << endl;
  Mat out(in.rows, in.cols, CV_8UC1);

  //Prepare input and output data
  DataDescriptor dataIn(inColor.cols*3, inColor.rows, DATATYPE_08U);
```

```
  DataDescriptor dataOut(in.cols, in.rows, DATATYPE_08U);
  memcpy(dataIn.GetDataPtr(), inColor.data, inColor.cols * inColor.rows * 3);

  APU_RGB_TO_GRAYSCALE process;

  int lRetVal = 0;
  //Set up the project
  lRetVal |= process.Initialize();
  lRetVal |= process.ConnectIO("INPUT", dataIn);
  lRetVal |= process.ConnectIO("OUTPUT", dataOut);

  //Execute
  lRetVal |= process.Start();
  lRetVal |= process.Wait();

  //Could also use if (process.ErrorOccured())
  if (lRetVal)
  {
    std::cout << "process error!" << endl;
    std::cout << process.ErrorMessage() << endl;
  }

  //Get the output
  memcpy(out.data, dataOut.GetDataPtr(), in.cols * in.rows);
  imshow("output ", out);

  waitKey();

  return lRetVal;
}
```

5 SDK examples

There are several Visual Studio 2013 projects inside the SDK which use the APEX-2 emulation library. They use the same source files as the target platform makefiles.

- `s32v234_sdk\demos\apex_emulation_test\build-deskwin32\mvc\apex_emulation_test.sln`
  A project demonstrating the usage of the library on several ACF kernels.
- `C:\vsdk\s32v234_sdk\demos\apex_orb_cv\build-deskwin32\mvc\orb_apex2_emu.sln`
  Homography based on the ORB algorithm.
- `s32v234_sdk\demos\apex_face_detection_cv\build-deskwin32\mvc\face_detection_lbp_apex2_emu.sln`
  Face detection using local binary pattern features (on demand there is also a face detection application basing on Haar-like features).
- `s32v234_sdk\demos\apex_add\build-deskwin32\mvc\apex_add.sln`
  Sample project that can be used for creating new projects using ACF.