APU-2 C Programmer Guide

UG-10301-00-06

Revision History

Version	Details of Change	Author	Date
01	Initial release	W. Hulme	Mar. 31 2013
02	Update for APU2 Tools R2.1	W. Hulme	April 17 2013
03	Update for APU2 Tools R2.2	W. Hulme	July 3, 2013
04	Update for APU2 Tools 13R1.2	W. Hulme	Nov. 18, 2013
05	Update for APU2 Tools J-2014.09	Lee, Ki-ju	Feb. 11, 2015
06	Updated text and graphics for clarification and typos	S. Francois	Sept. 27, 2016

Table of Contents

Table of Figures

Table of Tables

1 General Document Information

1.1.1 Overview

This document provides an overview of the APU-2 processor architecture, and the programming language for software development on the APU-2 architecture.

This document describes the C/C++ language extensions that allow a software developer to access the vector processor of APU-2. It explains compiler restrictions and unsupported features. It also describes the intrinsics that a software developer can use to get the maximum performance from the APU.

This document is not just a language description document, it also tries to map constructs to the actual hardware platform to help the reader understand what is being abstracted.

1.1.2 Acronyms

The following acronyms are used in this document, or in documents related to this release.

Acronym	Definition
CU	Computational Unit
CMEM	Computational Memory, local memory for each CU
DMEM	Data Memory
IMEM	Instruction Memory
I/O	Input/Output
SIMD	Single Instruction Multiple Data

Table 1: Acronyms

1.1.3 Audience

This document is intended for system or application programmers who wants to write APU programs on APU-2 technology.

1.1.4 References

The following APU-2 compiler documents are referenced or included in this document: “Chess Compiler User Manual”.

2 Architecture Overview

This section presents an overview of the array processor architecture. The processor architecture used in APU-2 is a reconfigurable-multi-core vector architecture. The main components of the design are illustrated in the following diagram (see

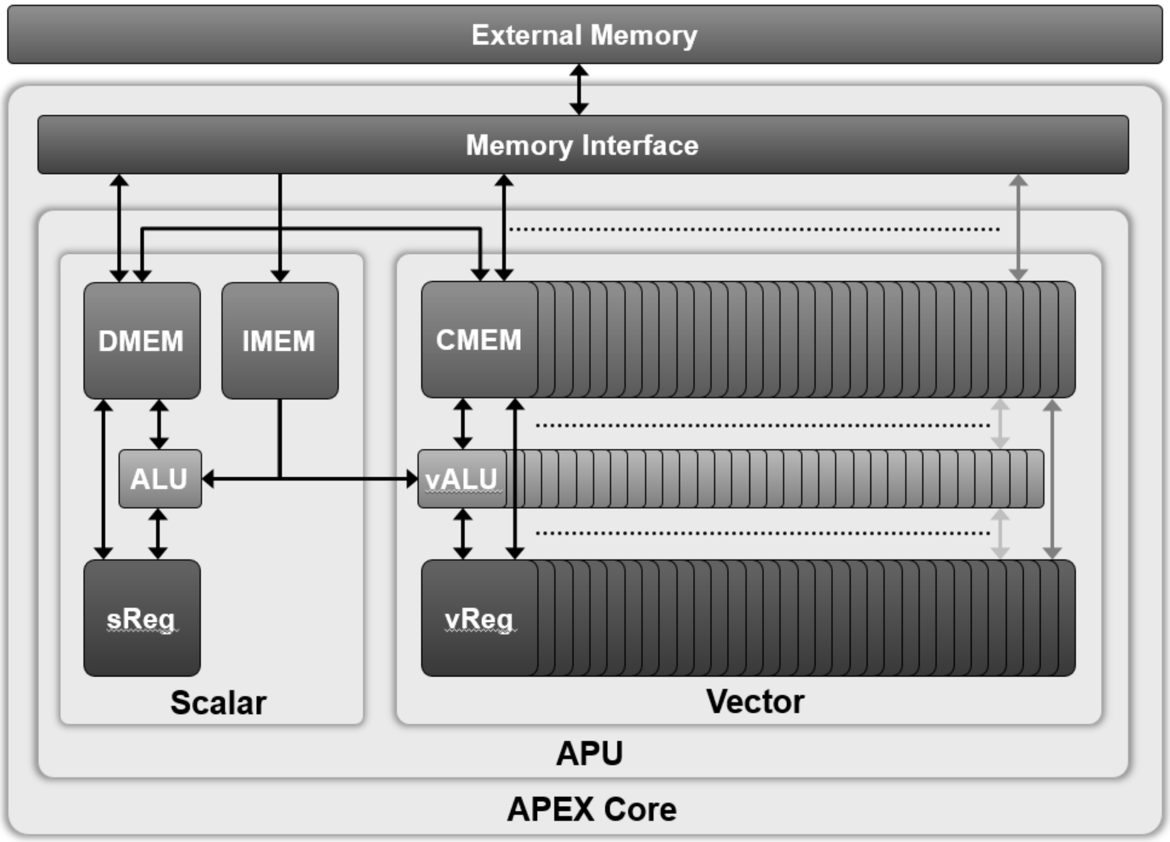

Figure 1). The purpose of this figure is to show all the resources:

- Memories
- Registers
- Processors (only ALU represented here)

This includes:

1. Scalar Processor (ACP);
    a. Instruction Memory (IMEM)
    b. Data Memory (DMEM);
    c. Scalar ALU (ALU)
    d. Scalar Register (SREG)
2. Vector Processor
    a. CU Vector Memory (CMEM)
    b. Vector ALU (vALU)
    c. Vector Register (vREG)

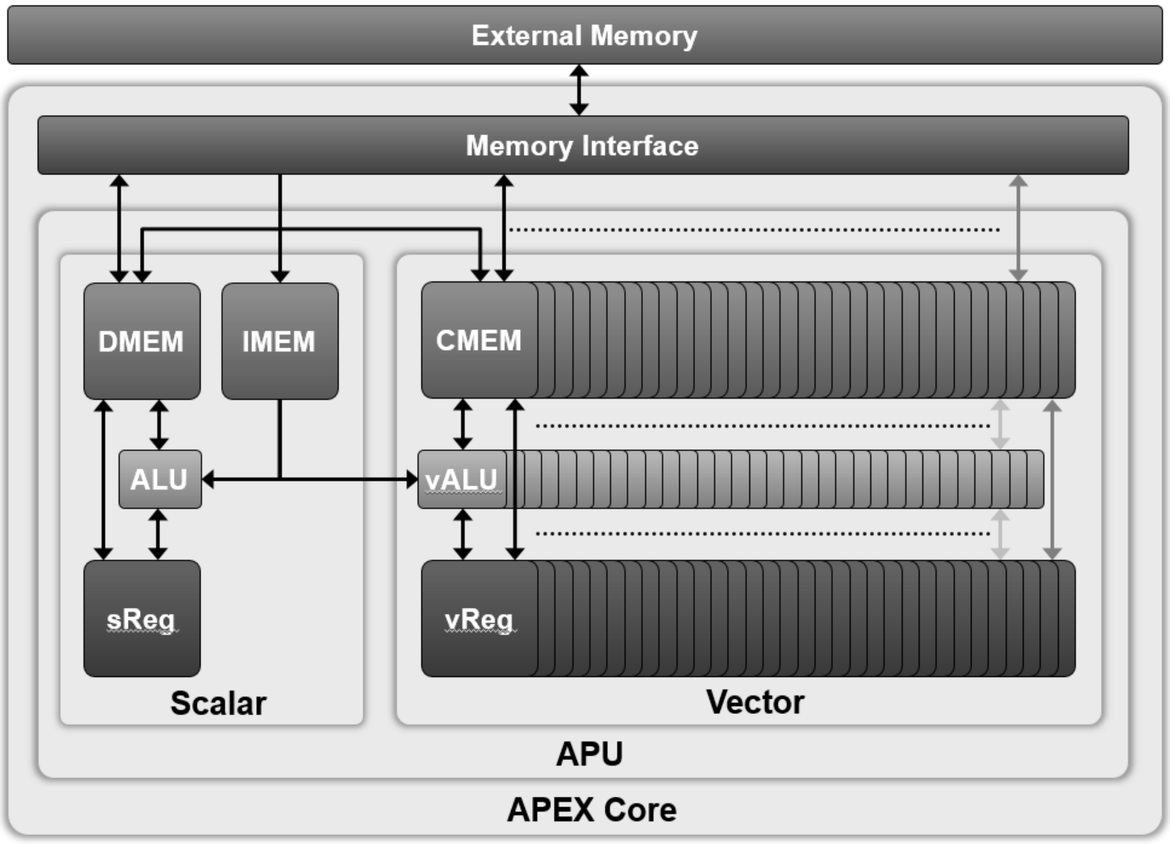

Figure 1: APU-2 - processor architecture diagram.

2.1.1 Storage of different data types

Scalars (data of scalar types) are stored into DMEM by default; like illustrated in the left hand side of Figure 2, in which an 8-bit unsigned scalar and a 16-bit unsigned scalar are stored into a given DMEM location.

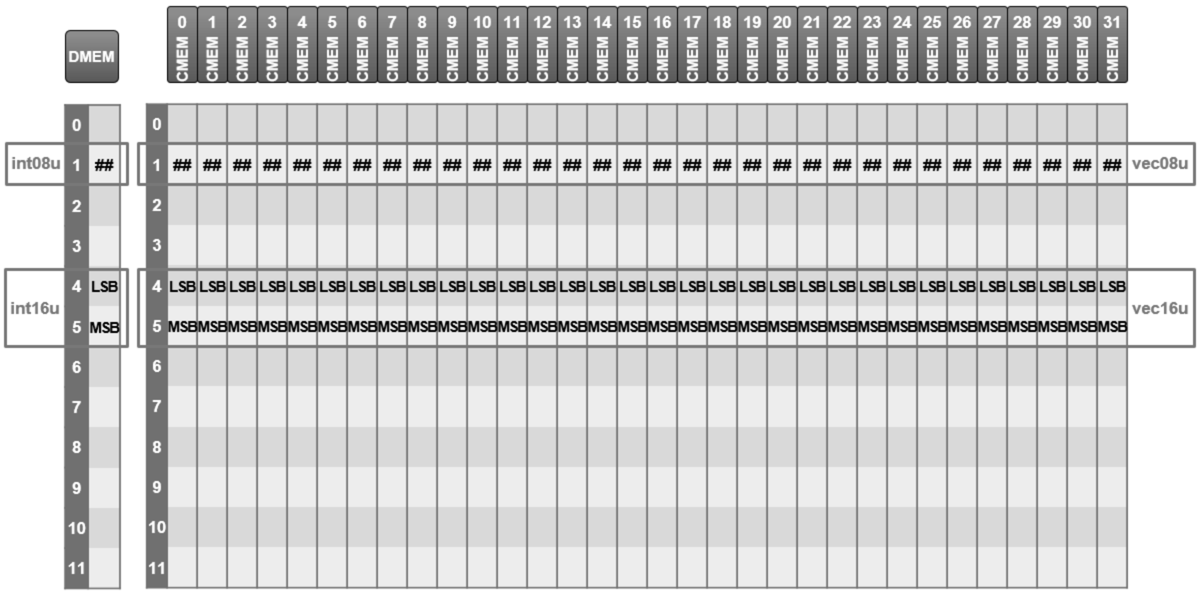

Figure 2: Local storage of scalar and vector data

Similarly, vectors (data of vector types) default memory storage location is the CMEM. The right hand side of Figure 2 shows how a vector of 8-bit unsigned (vec08u) and a vector of 16-bit unsigned are stored into CMEM.

The SIMD Vector Processor implements vector operations across an array of Computational Units, or CUs. A CMEM memory address or CU register contains a single independent value for each CU in the vector.

2.1.2 Scalar memory (DMEM)

Scalar memory (DMEM) is 8-bit addressable.

2.1.3 Vector memory (CMEM)

CU Vector memory (CMEM) is 8-bit addressable. The memory address of each element of the vector is identical, that is to say that vector memory locations are not specific to a given CU.

The maximum amount of memory per CU is dependent on the specific hardware version and is managed by APEX Core Framework.

3 Language and Compiler

The APU-2 specific language supported by the APU-2 tools is designed to be an extended subset of C (C99), with some C++ (C++98) (*ISO/IEC 14882:1998)* support. Language extensions exist to facilitate programming of the APU-2 SIMD array processor.

The language described in this document exposes the HW architecture of the APU to the programmer to allow the development of efficient and optimized processing kernels.

3.1 Nomenclature

3.1.1 Vector notation

Throughout the document we will use the {} brackets to index elements within vectors. If v is a vector of a given type v{0} is the first element (i.e. the value for CU 0), v{1} the second element (i.e. the value for CU 1), etc.

3.2 Basic Data Types

The basic data types are similar and derived from <stdint.h> introduced in C99, with extensions to express vector types. We are introducing them early in the document since they are used in the next section.

Scalar Type Name	Vector Type Name	C99 Equivalent (stdint.h)	ElementSize (bits)	Notes
bool	vbool	bool	8	false is 0 : true is not 0
int08u	vec08u	uint8_t	8	unsigned 8-bit element
int08s	vec08s	int8_t	8	signed 8-bit element
int16u	**vec16u**	uint16t	16	unsigned 16-bit element
int16s	**vec16s**	int16_t	16	signed 16-bit element
int32u unsigned int	vec32u	uint32_t	32	unsigned 32-bit element
int32s int	vec32s	int32_t	32	Signed 32-bit element
float	-	-	32	IEEE float, single precision
void	-	void	-	Used for function definition
void*	-	void*	32	Pointers are 32-bit Scalars

Table 2: Basic data types

Types in **Bold** are the native types: 32-bit for Scalars and 16-bit for Vectors.

3.2.1 true and false

The C99 true and false keywords are of type bool.

3.3 Data Storage Location

Scalar variables are stored in DMEM by default and vector variables are stored in CMEM by default.

Memory	Apply to
DMEM	all scalar
CMEM	all vector types

Table 3: Storage qualifiers

Example:

```
void func_00(vec16u vector1, vec16u vector2, int32_t n)
{
…
}
```

where func_00 is a function that takes 3 parameters, 2 of type vec16u (vector with 16-bit unsigned element) which are vector types (CMEM storage). The third parameter is a scalar 32-bit signed integer (DMEM storage).

3.3.1 Type Casting

The APU2-Tools compiler does not allow implicit casting for all operations and combinations of operands.

Relying on implicit casts may cause problems due to native data sizes. The scalar processor is a 32-bit processor. The vector processor is a 16-bit processor. Casting scalar types of int, int32s or int32u implicitly to vector will result in a vec16s type.

Pure vector operations:

- Vectors of the same type and size may rely on implicit casting.
- Vectors of same size but different sign require explicit casting.
- Implicit casting may be used when mixing vec16s and any vec08 operands. In this case, all operands and results will be implicitly cast to vec16s.
- Excepting the above, all operations containing operands of different sizes require an explicit cast.

Mixed vector/scalar operations:

- No implicit casts from scalar to 32-bit vector types occur. (see above)
- Implicit casting may be used if both scalar and vector operands have the same size and sign, for 8- and 16-bit types. In this case, they will be converted to vector types.
- Implicit casting may also be used if the scalar and vector operands are of the same sign, but the scalar operand is smaller than the vector type.
- In all other cases, an explicit cast needed.

Other:

The compiler will never require an explicit cast for operations between two or more scalar operands.

Logical and relational vector operations implicitly return vbool.

Relying on implicit casting will, where appropriate, result in an output type which is:

- Vector, if at least one operand was vector.
- Unsigned, if all operands were of the same size but both signed and unsigned operands were present.

- The size is set to be the same of the largest size of the operand types.

3.3.1.1 Scalar to scalar

The APU-2 compiler supports all implicit type conversions from one basic scalar data type to any basic scalar data type of the same size or bigger. Explicit cast is needed when going to a type of small size.

For example:

```
int8u b;
int16s a = (int16s)b;
```

The compiler imposes the following casting rules:

When the size of the destination type is less than the source type, the least significant bits are copied to the destination without any sign conversion.

For example:

```
int08s a;
int16u b = 0x008F; // b = 143
a = b;             // a = 0x8F = -113
```

3.3.1.2 Scalar to vector

When a scalar assignment is applied to a vector, the scalar value is implicitly casted to all elements of the vector.

For example,

```
int16u a = 5;
vec16u b = (vec16u)a; // all elements of vector b are equal to a – equal to 5
```

3.3.2 Vector to scalar

Vector to scalar cast is prohibited and results in a compiler error.

3.4 Pointers

3.4.1 Pointer to scalar

The scalar pointer to a scalar is like any other pointer on a generic CPU. It represents the address where a variable of a given type is located.

For example:

```
int32u s0;
int32u *p_s0 = &s0;
```

declares p_s0 as a scalar pointer pointing to a scalar unsigned 32-bit integer stored in DMEM.

3.4.2 Pointers to vectors and array of vectors

The same way we can declare scalar pointers to scalars we can declare scalar pointers to vector. For example, first we define an array of 32-bit unsigned vectors that we call v0_ar. Each element of this array is then a 32-bit unsigned vector. Then we define a pointer to a 32-bit unsigned vector and assign to it the address of the first element of our array of vector.

We would write,

```
vec16s v0_ar[16];
vec16s *pv_v0 = &(v0_ar[0]); // Declares a pointer to a vector in CMEM
pv_v0++;                     // Now point to v0_ar[1];
```

The next line

```
pv_v0++;
```

increments the pointer by the vector data type size (here, the vec16u is 2 vecbytes*, so the pointer address increments by 2), and now pv_v0 points to the next element in the array, which is v0_ar[1].

(*where a vecbyte is the smallest addressable unit in CMEM)

3.5 Size of basic data types

The number of elements per vector can be read through the special global variable __vecsize. This is a constant unsigned integer 32-bit set-up at runtime and holding the number of computation units (CU) controlled by the ACP.

```
extern volatile int32u __vecsize;
```

The APU-2 compiler supports the *sizeof*(type) operator for scalar and vector datatypes. For scalar datatypes, it behaves similarly to code for any conventional processors.

When the *sizeof* operator is applied to a vector data type, the size returned is the size of the of a single element of the vector, so in other words, it corresponds to the scalar type of the element.

All pointers, to scalar or vector data, are scalars themselves and stored on 4 bytes using a 32-bit address. The *sizeof* operator applied to any pointer returns 4.

3.5.1 Data structures

Standard C structure are supported by the compiler. Structures are either scalar or vector data structures. Structures containing a mix of scalar and vector members are not allowed.

3.5.2 Scalar Data Structures

Scalar structures are identical the standard C structures.

These structures are stored in DMEM (Data Memory).

For example:

```
struct aScalarStructure
{
   int08u mV0;
   int32s mV1;
   bool   mV2;
};
```

The members are accessed using the standard C/C++ notation.

```
aScalarStructure ss0;

ss0.mV0 = 0;
```

3.5.3 Vector Data Structures

The vector structures are not really different than scalar structures. As previously mentioned, vector structures only contain vector types, and are therefore stored in the vector memory (CMEM).

For example:

```
struct aVectorStructure

{

   vec08u mV0;

   vec16s mV1;

   vbool  mV2;

};
```

The members are accessed using the standard C/C++ notation.

```
aVectorStructure vs0;

vs0.mV1 = (vec16s)0;
```

3.5.4 Pointers to structures

Pointers to structures are allowed and their usage respects the standard C/C++ syntax. For example, using the previously defined structures

```
aVectorStructure vs0;

aVectorStructure *p_vs0 = &(vs0);


p_vs0->mv0 = (vec08u)0;
```

Like in C/C++, the -> operator is used to dereference members from a pointer to a structure.

This way it is possible to combine scalar and vector element into one structure by using pointers to structures; pointers being always stored in DMEM. It then becomes a scalar structure containing pointers to either scalar or vector elements.

For example:

```
// Structures definition and creation
struct aScalarStructure
{
   int32u i1;
   int32u i2;
} mya;

struct aVectorStructure
{
   vec08u input[2];
   vec16s output;
} myv;

struct aSVStructure
{
   aScalarStructure *pa;
   aVectorStructure *pv;
} myav;

// Structure initialization
myav.pa = &mya;
myav.pv = &myv;

// Using the structures
Myav.pa->input[0] = 0; // The elements of the first vector of the vector array
                         ‘input’ is set to 0
```

3.5.5 Function Calls

The APU compiler supports two separate stacks:

- Scalar Stack
- Vector Stack (non-standard)

So, it is possible for a function to accept multiple parameters of either scalar or vector types. It also means that return values can either be vectors or scalars.

3.5.6 Vector return value

Function can return a vector variable. For example,

```
vec16s function(…)
{
   vec16s rVal = 0;


   return rVal;
}
```

returns a vec16s variable.

3.6 Intrinsics

3.6.1 Access to vector elements

Access to a vector element form the scalar can be done through the use of the 2 following intrinsics:

- vput
- vget

See the help files in the /doc/html directory, accompanying this document, for more information on vput and vget.

3.6.2 Access to neighboring CUs data

Neighbour CUs accesses are done through 2 sets of specific intrinsics

- Vector Move Shifts
- Vector Move Rotate

See the help files in the /doc/html directory, accompanying this document, for more information on these operations.

3.7 Vector control flow statements

Vector processors control flow statements are different than the scalar ones. Because of its nature, the vector processor will implement control flow using predicated instructions. Predicated instruction is a computer design strategy to mitigate the cost associated with conditional branches. In our case, since all CU must execute the same instruction (SIMD) at the same time, all execution paths must be executed and instructions must have the capability of executing conditionally. For this reason, vector control flow statements are transformed differently than the standard scalar control flow, which implement the traditional conditional branches.

Nested vector conditions are implemented via a vector condition stack. The size of the vector condition stack is limited to handling seven nested vector conditions.

Warning: keep nested vif statement to a maximum of seven levels.

The compiler cannot detect if the vector condition stack overflows when more than seven nested vif are being used. The simulator will provide an indication at runtime when the vector condition stack overflows. When the maximum number of nesting levels is exceeded, the existing values are overwritten. This will result in incorrect or unexpected behavior at runtime.

3.7.1 Vector "*if*" statement

As mentioned, the vector control flow is done using predicated branch. A vector if statement follows the following notation:

```
vif (vexpression)
{
   statement true
}
velse
{
   statement false
}
vendif
```

Where "vector condition" can be any condition comparing 2 vector expressions and will execute following what is described by Figure 3.

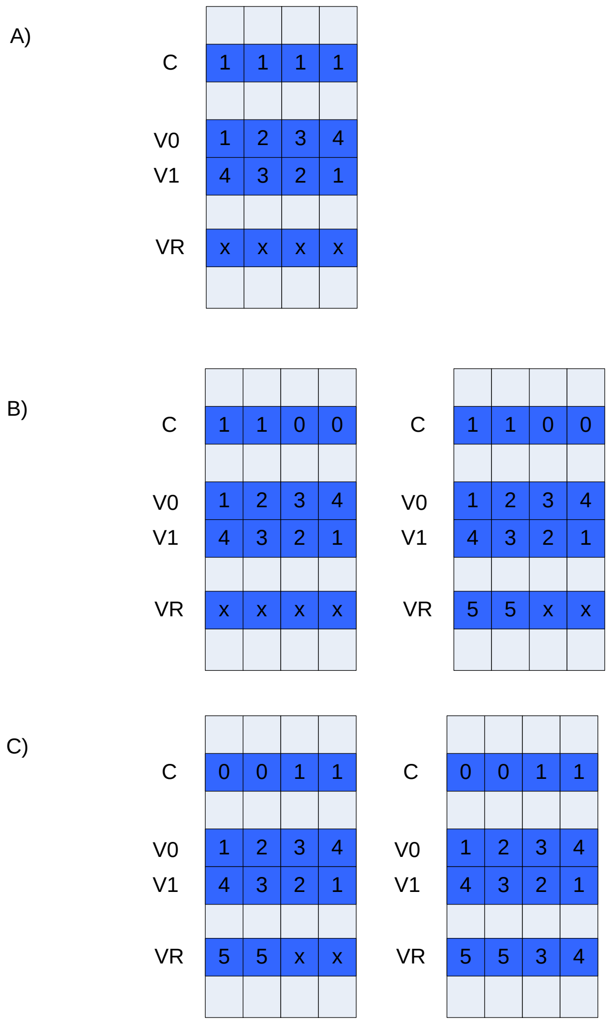

Figure 3: CMEM content during predicated if

An example code,

```
vec16s V0,V1,VR;


vif (V0 < V1)
{
   VR = V0 + V1;
}
velse
{
   VR = V0;
}
vendif
```

A) shows the content of the CMEM prior to executing the predicated instructions shown in the example code. After the vif statement is executed the C condition (which is a temporary vbool variable) is set. Note that the condition doesn’t have to be stored in CMEM. In fact, the value will most likely stay in a register. Though, it is easier to visualize the example this way.

B) shows the two state the CMEM goes through when first executing the vif statement and the first predicated instruction.

C) shows the last two state after executing the velse statement and the second predicated instruction.

3.7.2 Nested vifs

APU-2 supports a maximum of up to 7 nested vector conditions.

4 APU2 Instructions and Intrinsics

Consult the help content of instructionset_APU2-L-2016.09-2_NXP.html file in the /doc/apex/apu directory for more information.

5 APU-2 Compiler – C/C++ language and library support

See Chapter 4 of the Chess Compiler User Manual in the APU Tools /doc/manuals directory.

6 Standard Compliance of APU-2 C Compiler

See Appendix A of the Chess Compiler User Manual in the APU Tools /doc/manuals directory.