



# Vision P6 DSP Training

**Version** 7.4

Estimated time: 2 days

# Course Agenda

## Day 1

- Vision P6 Deliverables
- Core Architecture
- ISA Highlights
- Estimating Performance
- Auto-Vectorization of C Code
  - Lab 1 - Auto-Vectorization
- Programming with Vector Types
- Programming with Intrinsics
  - Lab 2 - Intrinsics
- Assessing Performance
- ISA Deep Dive

## Day 2

- Gather/Scatter Engine
  - Lab 3 – Gather/Scatter
- Vector Floating Point Unit
- Histogram Package
- Introduction to the iDMA Engine
- Using the iDMA Library
  - Lab 4 – iDMA Library
- Using the DMA Tile Manager
  - Lab 5 - Tile Manager
- Using the xiLibrary
  - Lab 6 – Xi Library



# About This Course

**Module**

**1**

**Dependencies:** Xtensa Fundamentals Training

**Revision**

**1.0**

**Version**

**7.4**

**Estimated time:**

- **For the lecture**      10 minutes
- **For the lab**            N/A

# Course Prerequisites

Before taking this course, you need to

- Have taken Xtensa Fundamentals Training
- Have knowledge of
  - Embedded Programming
  - Some familiarity with DSP Processors
  - Have some experience with Image Processing Algorithms

# Course Objectives

In this course, you will

- Learn about basic Vision P6 architecture
- Get familiar with product deliverables
- Learn to estimate performance
- Use the auto-vectorizing compiler for simple imaging/vision algorithms
- Use vector types and operation intrinsics
- Assess and measure overall performance of your code
- Get an overview of P6 operations
- Use Gather/Scatter and VFPU operations in a practical examples
- Use the iDMA library DMA Tile Manager for data movement
- Apply xiLibrary functions to your code

# Vision P6 Deliverables

**Module** 2

**Revision** 1.0

**Version** 7.4

**Estimated Time:**

- For the lecture 15 minutes
- For the lab N/A

**Dependencies:**

- None

# Module Objectives

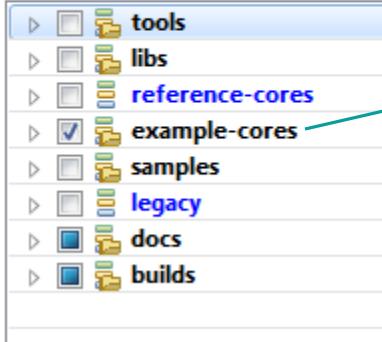
In this module, you

- Learn how to find the deliverables define the Vision P6 product
- Install the vision P6 configuration build
- Explore Online ISA HTML
- Open the User Guide
- View the Software Example Package
- Find the Xi Library
- Learn where to find the Tile Manager source code

# Finding the Vision P6 Deliverables in XPG

RG-2016.4

type filter text

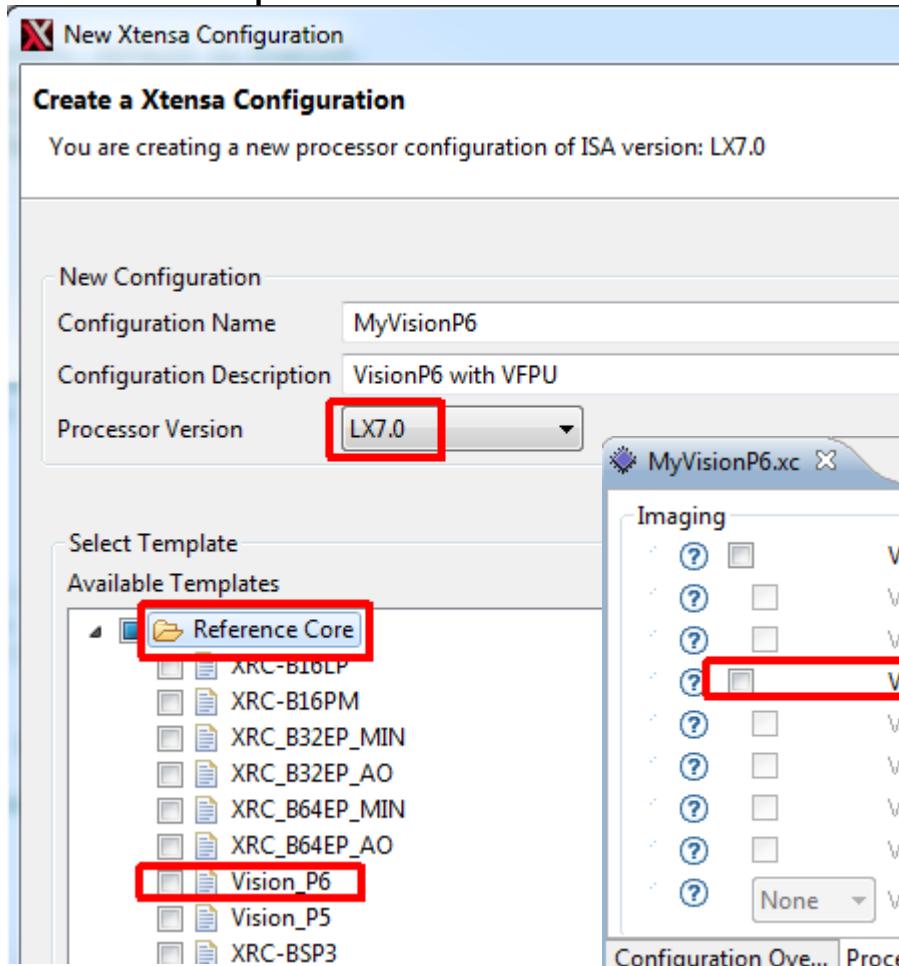


<input checked="" type="checkbox"/>	example-cores
<input type="checkbox"/>	Vision P5 example cores for Windows
<input checked="" type="checkbox"/>	Vision P5 example cores for Linux
<input type="checkbox"/>	Vision P6 example cores for Windows
<input checked="" type="checkbox"/>	Vision P6 example cores for Linux
<input type="checkbox"/>	HiFi example cores for Windows
<input type="checkbox"/>	HiFi example cores for Linux
<input type="checkbox"/>	HiFi3 example cores for Windows
<input type="checkbox"/>	HiFi3 example cores for Linux
<input type="checkbox"/>	TIE development example cores for Windows
<input type="checkbox"/>	TIE development example cores for Linux
<input type="checkbox"/>	Tutorial sample cores for Windows
<input type="checkbox"/>	Tutorial sample cores for Linux

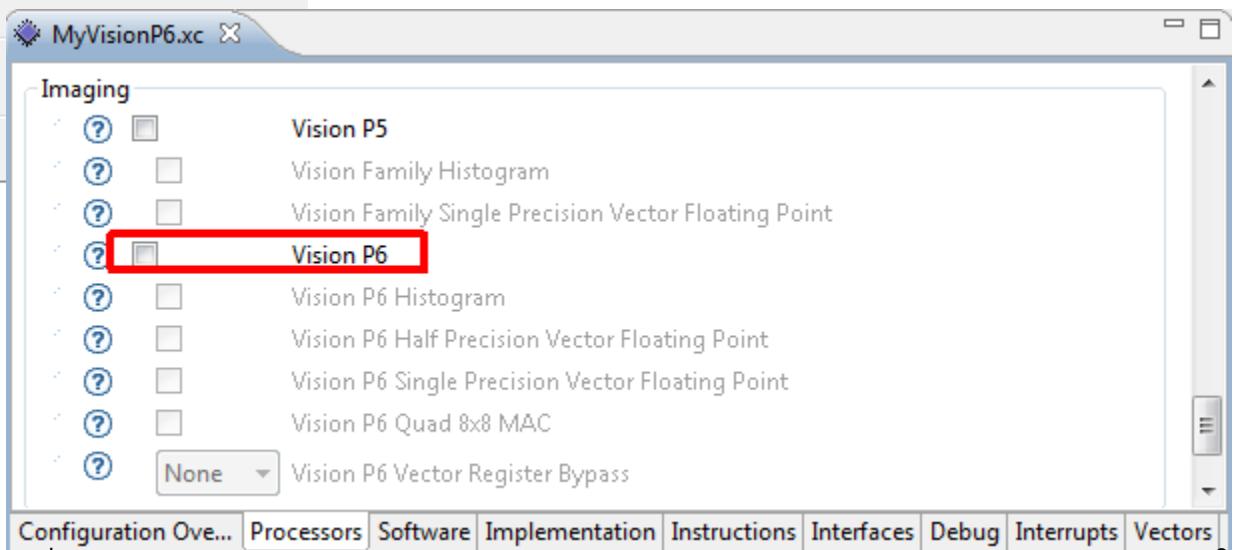
- The Vision P6 reference core is located in the example-core folder
- Vision P6 User Guide is located in the **docs** folder
- Software Package is located under **samples**

# How to build a Vision P6 configuration

- Need to choose LX7
- Select the Vision\_P6 reference template

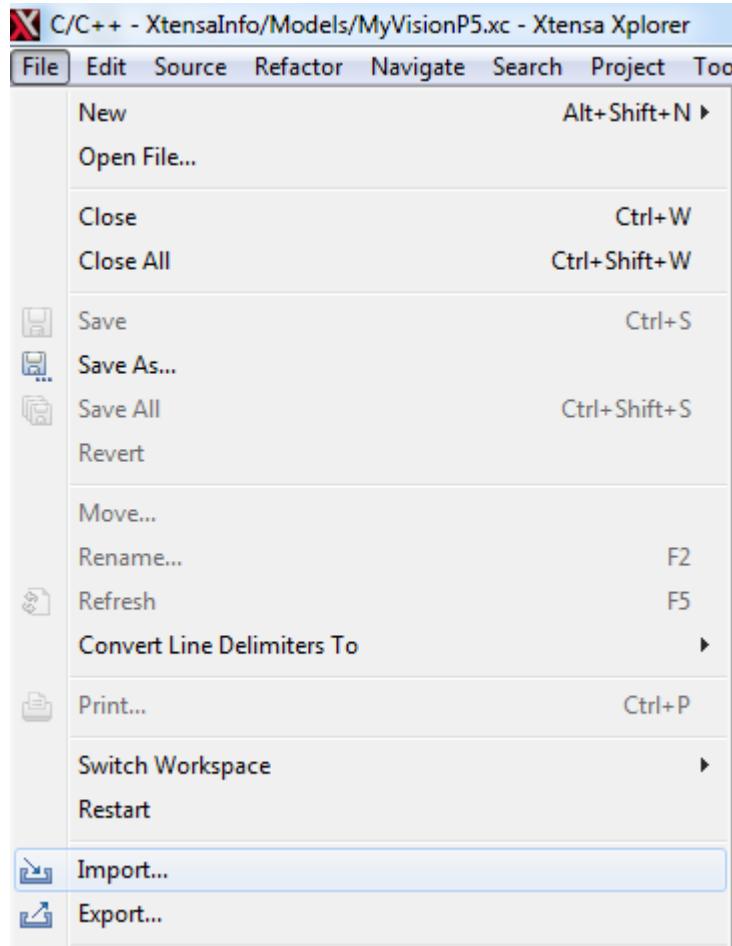


- In the **Configuration Overview** tab of the configuration editor you will find a block called **Imaging**
- Select **Vision P6 checkbox** for XPG to enforce configuration options consistency checks
- You can choose the **Histogram** and **VFP** at the same place

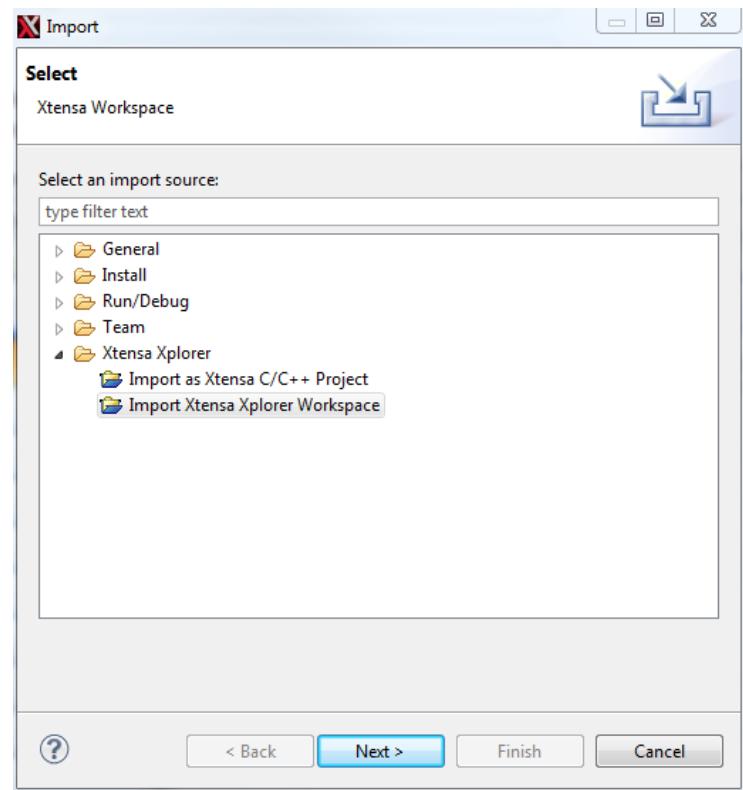


# Install the Vision P6 reference Configuration

From the File Menu select  
**Import ...**

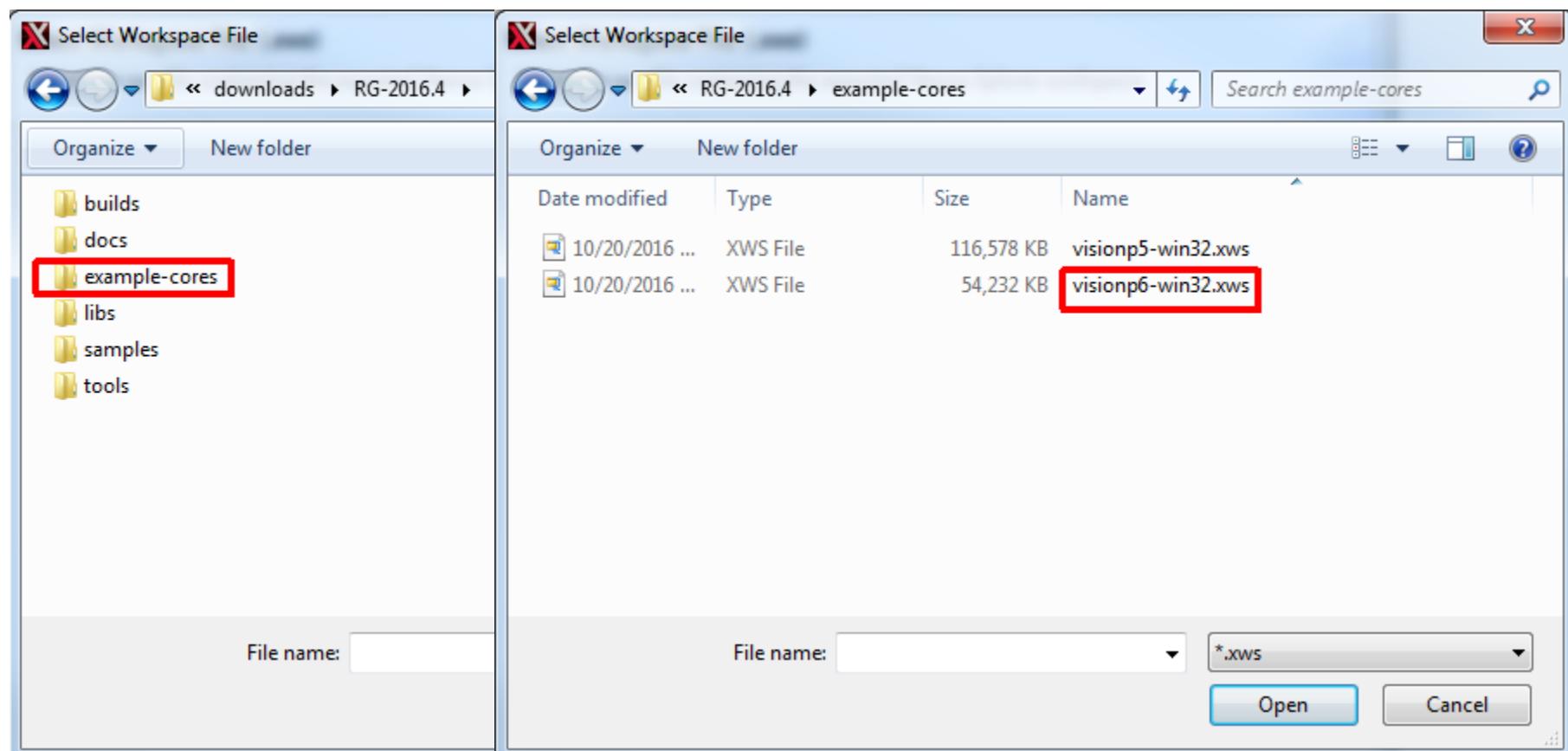


Select Import **Xtensa Xplorer**  
**Workspace**

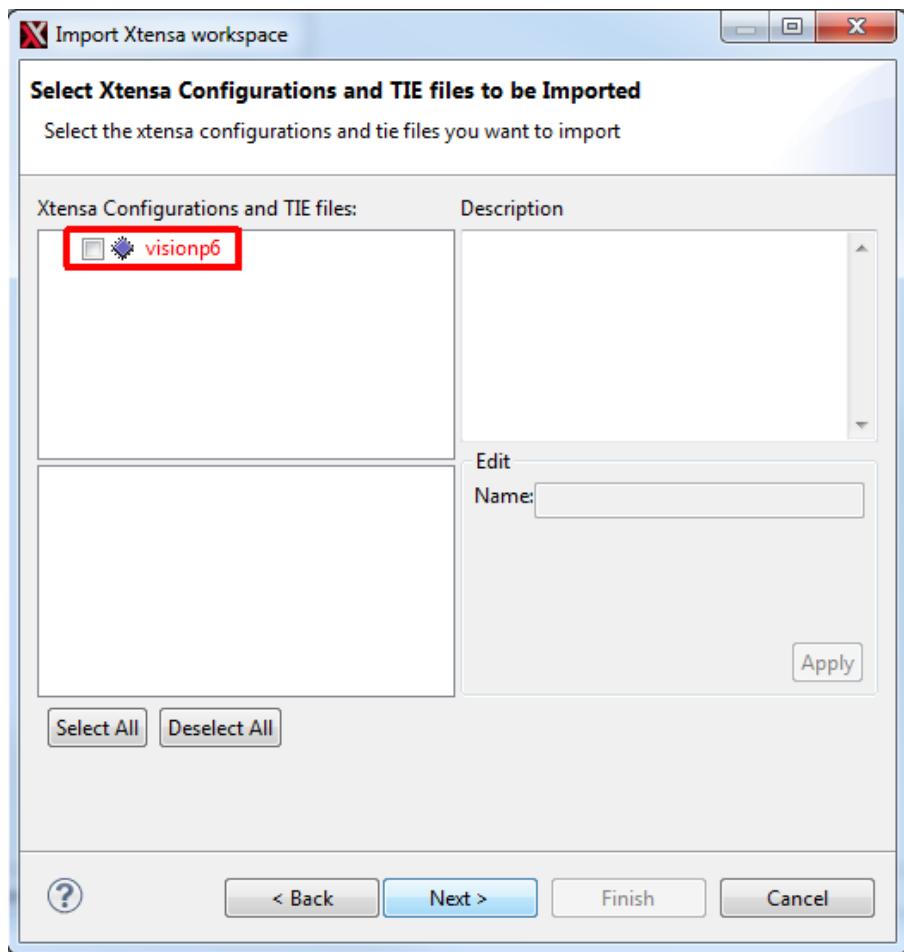


# Install the Vision P6 reference Configuration ... continued

- Locate the **downloads/RG-2015.1** Directory
- Choose **example-cores**
- Select the **visionP6-win32.xws** workspace



# Install the Vision P6 reference Configuration ... continued



- Click on both checkboxes `visionP6` and `visionP6_ao`
- `visionP6` is the reference configuration
- `visionP6_ao` (all options) is the same as the reference configuration except that it has the Histogram and Vector Floating Point Unit (VFPU) option selected
- Please keep in mind that Vision P6 is **feature controlled**. You may or may not be able to build Vision P6 depending on your **account settings**.

# How to find the ISA HTML

The screenshot shows a web browser window titled "ISA HTML" displaying configuration details and a list of instruction set architecture topics. The configuration details include a prefix of Xm\_visionp5, target release RG-2015.1, target hardware version LX7.0.1, and software version 12.0.1. A red oval highlights the "1.2 Instruction Descriptions" link in the list of topics.

Configuration Details

- Prefix: Xm\_visionp5
- Target Release: RG-2015.1
- Target Hardware Version: LX7.0.1
- Software Version: 12.0.1

1 Instruction Set Architecture

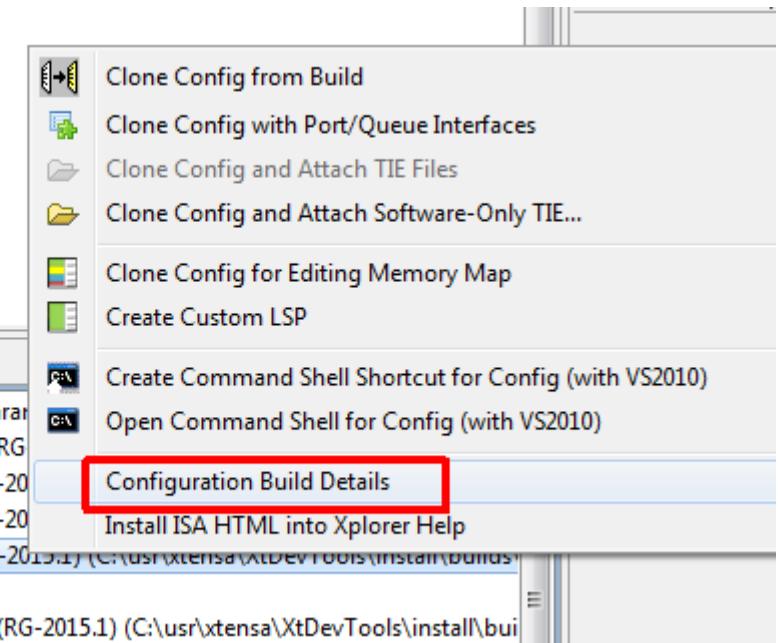
- 1.1 Instruction Formats
- 1.2 Instruction Descriptions
- 1.3 Instruction Opcodes
- 1.4 C-Types, Operators and Instruction Prototype
- 1.5 Instruction Pipelining
- 1.6 State List

2 Hardware Documentation

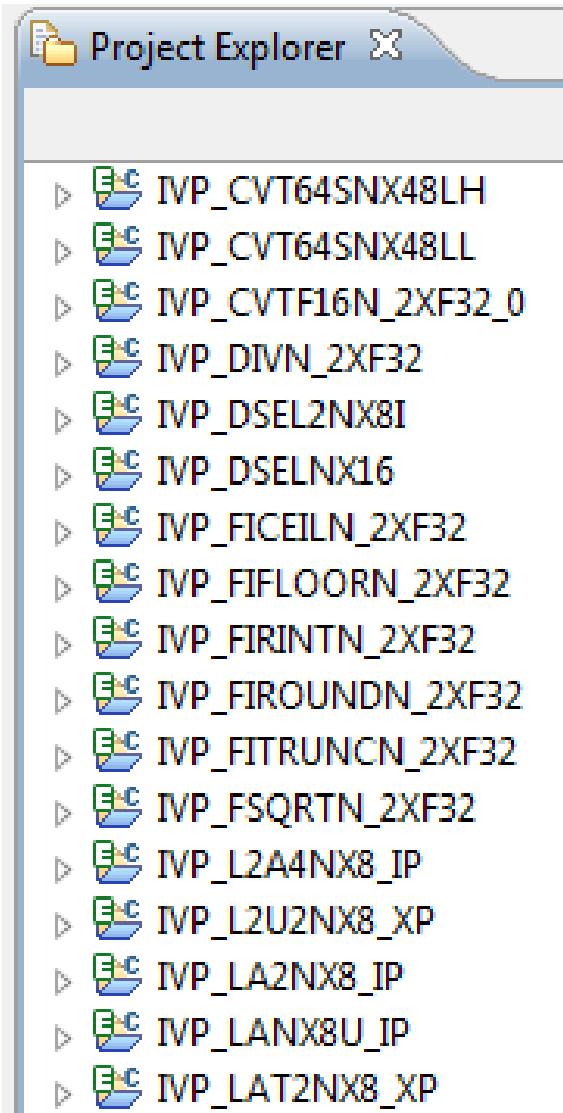
- 2.1 Port List

IVP\_DSEL    ^ v    Highlight All    Match Case

- Right click on the Vision P6 folder and choose **Configuration Build Details**
- This opens a web browser with ISA HTML help
- Instruction Descriptions** links to a list of Vision P6 operations



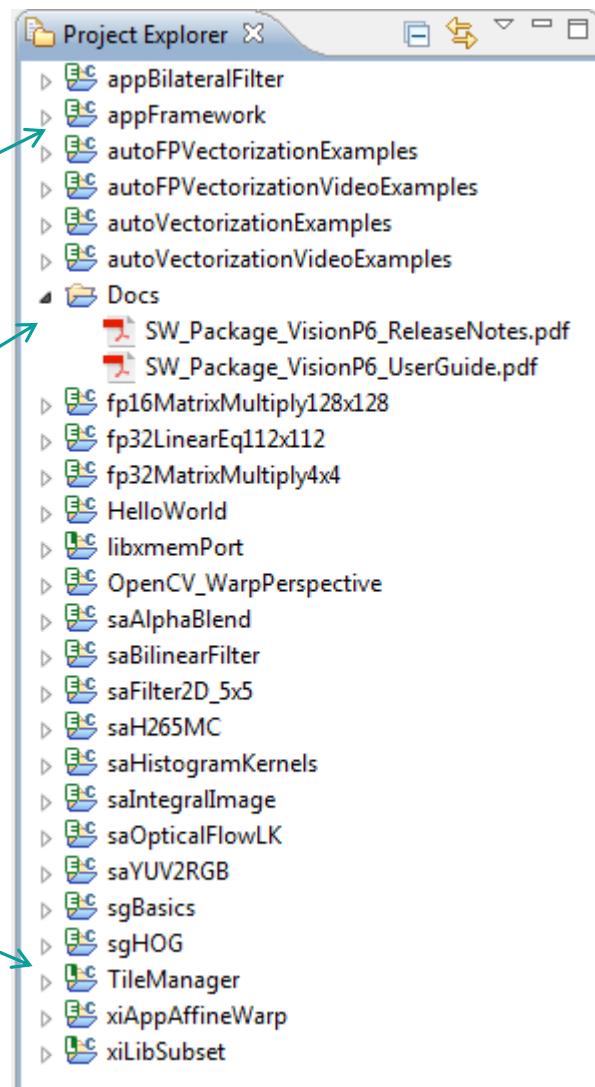
# Operation Examples Workspace



- Covers 80 vector operations
- Elementary example for operation
- Each project covers one operation
- Augments ISA HTML
- Workspace available on support site
- Search support site for
  - “Vision P6 Operation Examples”

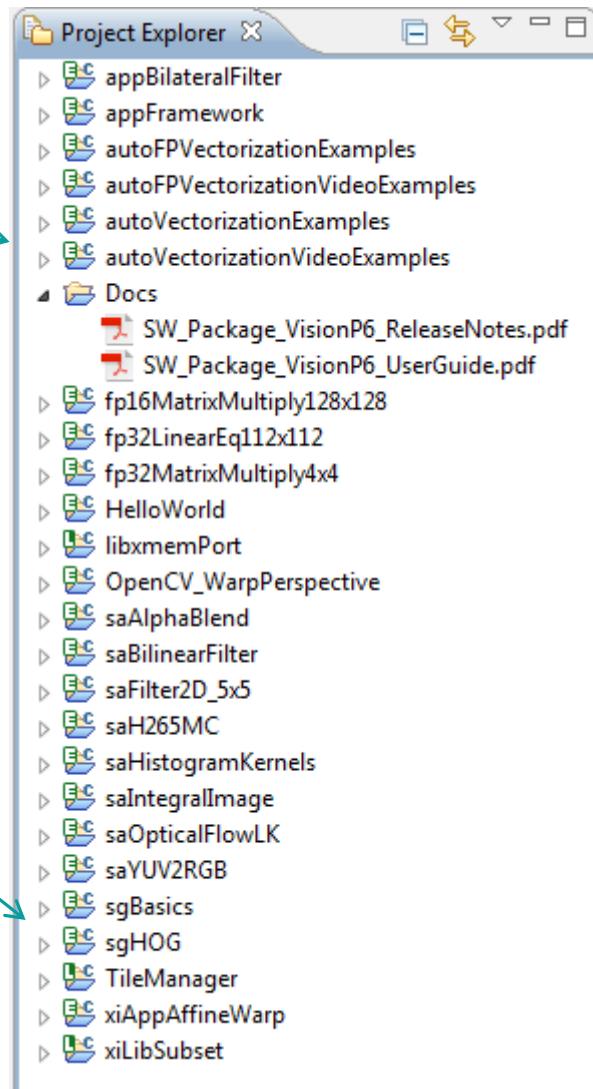
# Vision P6 Software Package

- Delivered as **XWS**
- Application Level projects include DMA code and start with the name **app**
- **Docs** folder contains PDF documents with Release notes and detailed UG that describes each example
- **Tile Manager** source code is provided as a project as well



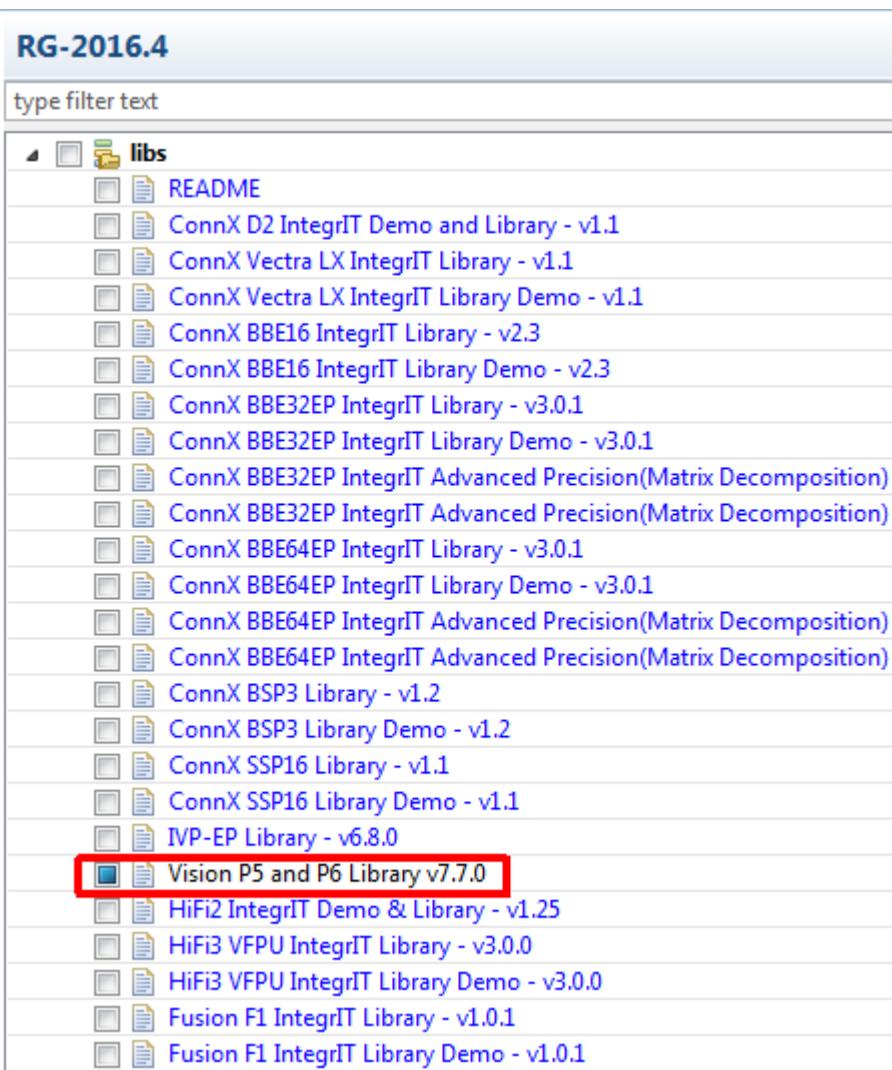
# Vision P6 Software Package

- Two examples target auto-vectorization
- Prefix **sa** (stand alone) indicates small examples without DMA
- Prefix **sg** indicates example using Scatter/Gather



# XI Library

- The XI Library (also referred to as OpenCV Library) is available in the XPG view
- It is located in the XPG “libs” directory
- The name is “Vision P5 andP6 Library”
- It is delivered as Xplorer workspace



# Summary

In this module, you learned to:

- Install the Vision P6 reference configuration inside Xtensa Xplorer
- Learned about the SW examples
- Saw how to access ISA HTML and UG
- Find the Xi Library and Tile Manager source code

The **User Guide** gives you **comprehensive overview** of the product.

The **examples** are a **reference** for developing your own Vision P6 kernels

The **Xi Lib** contains **imaging and vision functions** that **accelerate** your SW Application development

The **ISA HTML** provides operation **details**, which will come in handy when you program with intrinsics.

Programming Vision P6 is a **broad topic** that we will explore in the **3 separate modules**.

# Quiz

1. We refer to the Vision P6 core build as a configuration because
  - a) We provide a Vision P6 reference configuration.
  - b) Vision P6 is not configurable.
  - c) it is - like every Xtensa core build - defined by a choice of configuration options that were set when the core was build in XPG.
2. The ISA HTML
  - a) teaches the user good programming practices.
  - b) provides a comprehensive list of operation names and descriptions.
  - c) contains the User Guide.

Answers: 1c, 2b



# Vision P6 Architecture

<b>Module</b>	<b>3</b>	<b>Dependencies:</b>
<b>Revision</b>	<b>1.0</b>	<b>• None</b>
<b>Version</b>	<b>7.4</b>	
<b>Estimated Time:</b>		
• For the lecture	30 minutes	
• For the lab	N/A	

# Module Objectives

In this module, you get an overview of the Vision P6

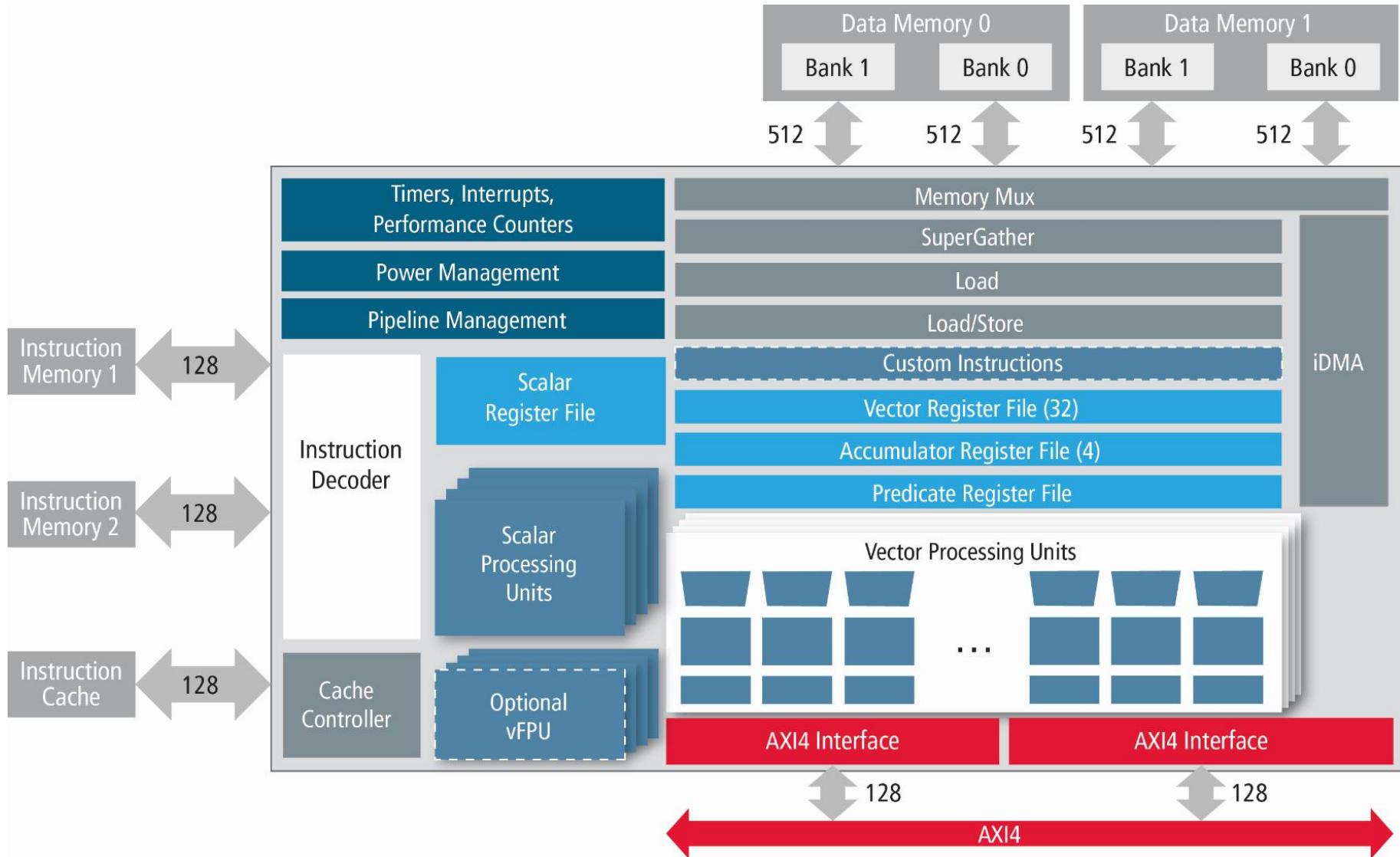
- Processor Components
- Reference Configuration
- Instruction and Data Paths
- Vision P6 Instruction/Data Parallelism
- Memory Architecture
- Configurability and Extensibility

# Vision P6 Highlights

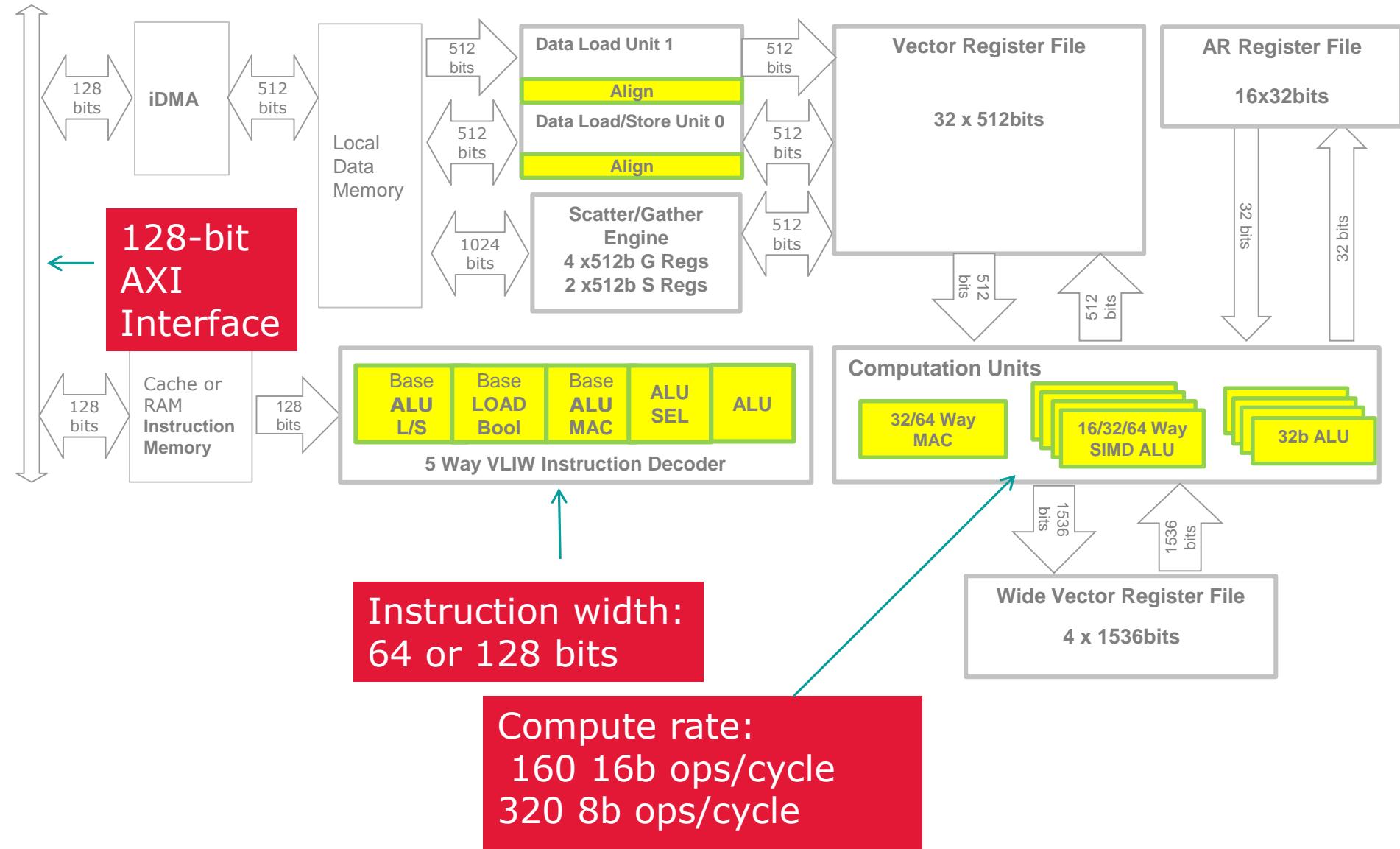
Vision P6 is a high performance SIMD-VLIW DSP optimized for computer vision and imaging applications

- **32/64-way SIMD, 1-5 issue slots per instruction**
- Support for **8/16/32-bit Integer** vector operations
- optional **16/32-bit half/single precision Vector Floating Point Unit(VFPU)**
- **7-stage processor pipeline**
  - one extra execution stage for complex DSP instructions (such as MAC)
- **128/64-bit Vector** and **16/24-bit Xtensa Base** instruction width
- **Dual load/store** capability with **512-bit wide interface** to local memory
  - capable of loading **1024 bits per cycle**
- **128-bit AXI4** interface to system memory and peripherals

# Vision P6 Block Diagram



# Vision P6 Processor Instruction/Data Paths



# Vision P6 Instruction/Data Parallelism

- 32-way, 16-bit wide vector SIMD ops in each VLIW slot
- 64-way, 8-bit wide vector SIMD ops in each VLIW slot
- 5-way VLIW instruction issue (**320 8-bit ops**)
- Number of **vector operations** per cycle
  - Up to **4 ALU**
  - **1 MUL/MAC,**
  - **1 Select**
    - Some SELI operations can issue in 3 additional slots
  - **2 loads or 1 load + 1 store**

One Instruction can issue up to 320(=64\*5) concurrent 8-bit ops:

64 LOAD + 64 PACK + 64 MUL + 64 SEL + 64 MOVE

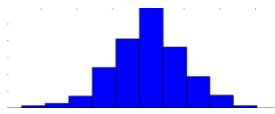
# Major Configuration Options and Reference Configuration

Option	Valid Range	Reference Configuration
Vision Histogram	On/Off	Off
single precision VFPU	On/Off	Off
half precision VFPU	On/Off	Off
Scalar FPU	On/Off	Off
I\$	0-128 kB, 2-4 way	64kB, 4 way
IRAM	0-4 MB	0
DATA RAM0	16kB - 4 MB, 4 or 8 sub-banks	128 kB, 8 sub-banks
DATA RAM1	16kB - 4 MB, 4 or 8 sub-banks	128 kB, 8 sub-banks

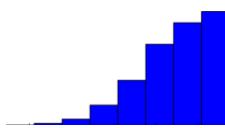
# Optional TIE Packages

## Histogram Package

- Accelerates computation of
  - distributive histograms



- cumulative histograms



- Up to 24x acceleration over base Vision P6 core

## Vector Floating Point Unit (VFPU)

- Support for single/half precision floating point
- IEEE 754 compliant
- 16/32-way SIMD

# Integrated DMA (iDMA) Engine

## Purpose:

- Allows hiding of system memory latency from the Vision P6 core
- Allows moving data between system and local data memory in the background while processor performs computation
- Also supports local to local Data RAM transfers
- Fully integrated with the processor and modelled in ISS

## Control

- Controlled by
  - DSP core reading/writing control registers (via WER/RER operations)
  - DMA descriptors stored in local data memory
- Descriptors processed in a linear order (single-channel DMA)
- Optional DMA completion interrupt

# iDMA Engine ... continued

## Descriptors

- Support **1D** and **2D** data transfers
- **JUMP** command enables
  - **circular** descriptor buffer or
  - “**non-linear**” processing of descriptors
- Source/Destination addresses can fall on **any byte boundary**
- Transfer size can be **any arbitrary byte size**

## Performance

- **128-bit** data throughput **per cycle** (over AXI)
- **Configurable**
  - AXI **request buffers** (1,2,4,8,16) for AXI read transactions
  - 2D transfer **row count** (2,4,8,16) which sets the maximal number rows that requests can pipeline along
- **Programmable** control over
  - maximum AXI **block size** (2,4,8,16)
  - maximum number of **outstanding AXI requests** (1 to 64)
  - AXI Quality of Service indication(QoS)

# Vision P6 Instruction Memory Interface

- 128-bit wide interface to
  - Instruction Cache (**I\$**)
  - Instruction RAM (**IRAM**)
- Optional **I\$** (0 up to 128 KB)
  - 2, 3 or 4 way associative with locking
- Optional **IRAM** (0 up to 4 MB)
  - for frequently used, performance critical code

# Vision P6 Local Data RAM (DRAM) Interface

- 512-bit interface to local data memory
- Dual load/store capability with 512-bit wide interface to local memory
  - capable of loading **1024 bits per cycle**
- Up to **2 local DRAMs** supported
  - enables conflict-free accesses of processor and iDMA
- Each DRAM size can be from **16 kB to 4 MB**
- Sub-banked memory interface supports scatter-gather operations
  - More on this on next slide ...

# What is Scatter / Gather?

- Ability to quickly and efficiently read/write **non-contiguous** locations from local data memory
- Enables the full use of SIMD capabilities for algorithms like
  - warping
  - lens distortion correction
  - canny edge tracing, etc.

## Data Memory Organization

- Each DRAM consists of **two banks** each 512 bits wide
- Each bank is divided into **sub-banks**
- **Number of sub-banks** is configurable from **4** to **8** (**8** for reference configuration)
- Scatter/Gather operations can
  - access sub-banks individually controlling the **row address per sub-bank**
  - Accessing **all sub-banks**, of each **memory bank**, of one **DRAM** in **parallel**
  - in the **same clock cycle**

# Scatter / Gather ... continued

## Scatter/Gather Performance

- Gathering 32 16-bit non-contiguous values
  - Requires between **1 and 32** consecutive accesses
  - depending on the **number of sub-banks** configured,
  - number of **sub-bank conflicts** (multiple **different rows** in the same **sub-bank**)
- The Gather Engine
  - pipelines request
  - loads across the two “oldest” gathers
  - for more efficient sub-bank utilization
- Gather requests (Gather-Address Operations)
  - are **non-blocking** for the processor pipeline
  - until the data is actually **needed** by subsequent instructions which is defined by
  - **issue of Gather-Data operation** with matching Gather/Scatter Register

# **Extensibility**

## **Extensibility with User TIE**

- Like every other Xtensa Processor Vision P6 is extensible by User TIE

## **Optional interfaces:**

- TIE port connections to external RTL blocks
- TIE queues for processor interconnect
- TIE lookups

# Quiz

1. The Vision P6 DSP can issue
  - a) two vector load and one vector store operations in parallel
  - b) two load operations or a load and a store operation in parallel
  - c) at most one load operation per cycle
2. The Vision P6 core is
  - a) extensible and configurable
  - b) extensible but not configurable
  - c) configurable but not extensible
  - d) neither configurable nor extensible

Answers: 1b, 2a

# Summary

In this module, you learned about the Vision P6

- reference configuration
- instruction and data path widths
- instruction and data parallelism
- configurability and extensibility

Vision P6 is not a **fixed** but **configurable** and **extensible**.

Information about data path widths and number of instruction slots gives you rough information about Vision P6's **compute throughput**.

We have **not told** you about the

- **kind of operations** that Vision P6 supports and
- how many of them can **execute in parallel**.

We will cover this in the next module: “**Module 4 – Vision P6 ISA Highlights**”.



# Vision P6 ISA Highlights

<b>Module</b>	<b>4</b>
<b>Revision</b>	<b>1.0</b>
<b>Version</b>	<b>7.4</b>
<b>Estimated Time:</b>	
• For the lecture	30 minutes
• For the lab	N/A

**Dependencies:**

- None

# Module Objectives

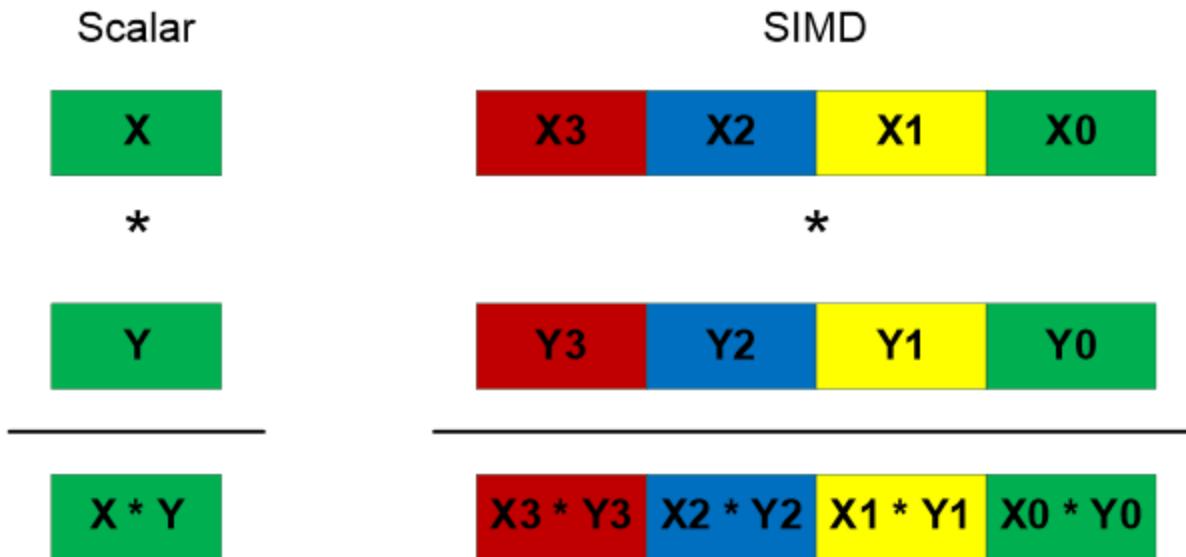
In this module, you will learn about Vision P6

- Data Parallelism (vector size)
- Instructions Parallelism (FLIX)
- Assignments of operation categories to instruction slots
- Instruction Formats
- Registers Types
- Operation Categories

# Data Parallelism: SIMD

- SIMD = **S**ingle **I**nstruction **Multiple **D**ata**
- Scalar operations compute a **single data element** at a time
- SIMD operations **replicate** an operation for **multiple** data elements in a **vector**
- In Vision P6 we refer to the SIMD width as the **vector length**

Example: Scalar and SIMD multiply operation



# Data Parallelism in Vision P6

The Vision P6 **vector length (SIMD-width)** is **512 bits = 64 bytes**.

The **number** of vector **elements** depends on the element size.

Element Width (bits)	Number of Vector Elements	Operation Name Modifier	Vector Size (bits)	Vector Elements
32	16	"N_2"		w15 w14 ... w2 w1 w0
16	32	"N"	512	s31 s32 ... s2 s1 s0
8	64	"2N"		b63 b62 ... b2 b1 b0

Examples of Vision P6 operation names will follow in the **next slide**.

# Vision P6 Operation Name Examples

Operation Name	Explanation
IVP_ABSNX16	32-element <b>16-bit</b> signed absolute value, non-saturating
IVP_SEL2NX8	Select <b>64-element 8-bit</b> vector from two input vectors
IVP_ADDN_2X32	<b>16-element 32-bit</b> signed sum
ADD	<b>Xtensa Base Operation</b> , 32-bit signed sum
IVP_MULNX16	32-element <b>16-bit</b> signed multiply - producing (48-bit) results with sign-extension

- In the table above you can see how the **vector length** and element **bit-width** are embedded in the **operation name**
- Vision P6 **Vector operations** start with the prefix **IVP\_**
- Operations **without** that prefix are **Xtensa Base Operations**

# How to find the operations I need?

## ISA HTML

- Comprehensive, detailed documentation for vision P6 operations
- Contains detailed information about each operation
- In RG.4 we added hierarchical and functional grouping of operations
- It is also possible to search for input and output data types
- This makes it easier to find a particular operation

## Vision P6 UG

- Very useful for getting an overview of Vision P6
- The chapter “Standard DSP Operations By Type” gives a good introduction to the operation categories available in Vision P6

# Instruction Parallelism

- Instruction parallelism is often referred to as VLIW
- VLIW = **Very Long Instruction Word**
- A VLIW instruction can execute multiple operations in parallel (same cycle)
- The **number** of parallel operations performed depends on the **instruction format**
- Instructions with different formats and **bit widths** can be intermixed **without mode switching**
- In “Xtensa Language” this flexible VLIW concept is called **FLIX**

# Instruction Parallelism in Vision P6

- Vision P6's five issue slots
- Below is a table (simplified for clarity) of the categories of operations that can be issued in each

Slot 0	Slot 1	Slot 2	Slot 3	Slot 4
Base Ops Vector ALU Vector Load Vector Store GatherA Scatter	Base Ops Vector Load Vector Boolean WVEC Packs GatherD	Base Ops Vector ALU Vector MAC Vector Divide	Base Ops Vector ALU Vector Select Vector Shift Vector Reduction	Vector ALU

- Instruction formats (see next slide) will determine which of subset of operations from slot0 – slot4 can be issued by an instruction
- The **number** of operations Vision P6 can issue is **flexible**
  - Between **1** and **5** operations can execute in parallel
- The FLIX instruction can be **64** and **128**-bit wide
- The next slide shows a diagram with the Vision P6 instruction formats

# Why do we have instruction formats?

Allowing **concurrent issue of all operations every slot** would create

- A need for a high number of **register ports** which could negatively affect **area** and **max achievable frequency**
- **Unnecessary replication of HW**, for example
  - Allowing a concurrent **Vector Add** operation in all **5** instead of **3 operation slots** would likely not yield a performance benefit
  - There would still be a need to **load** and **store** data and we would likely **not be able to use** 5 Vector Add operations in parallel.
- Challenges in **encoding the instructions**
  - More operations and more operands per instructions take up more encoding bits

# Why do we have instruction formats? ... continued

Instead, we define instruction formats which

- allow a **constrained** but **common pairings** of operations
- Limit the operations the have high REG read/write bandwidth to a single a or a few slots
- **Balance** the register ports between slots
- Use “narrow” (N) formats to reduce **code size** when wide (F) formats can’t be fully utilized (an example will follow in a few slides)

Format	Slots	Usage
F0	4	Pairing DSEL, WVEC MUL, PACK and L/S
F3	5	Paring MUL, 2 ALU, PACK and L/S
F5	4	Rich pairing of Xtensa Base Operation in all slots
N1	2	“Narrow“ format with operations from Slot0 and Slot2

- For a full description please refer to section “Vision P6 Instruction Formats” in the **UG**

# Vision P6 FLIX Instruction Formats and Slots

## 4-Slot 128-bit Formats



*Format F0, F1, F2, F4, F5*

## 5-Slot 128-bit Formats



*Format F3 and F11*

## 2 Slot 64-bit Formats



*N0*



*N1*



*N2*

User Defined Formats can be added with TIE



*User Defined Format*

# Example of Vision P6 Assembly code

This assembly code is from the inner loop of the **Alpha Blend** project in the **Vision P6 SwP**

```
{ # format F0
    ivp_lv2nx8_ip    v0,a2,128          # [0*II+0]  id:44
    ivp_packvru2nx24 v2,wv0,a6          # [2*II+0]
    ivp_mulusp2n8xr16 wv0,v1,v0,a11   # [1*II+0]
    nop               #
}
{ # format N2
    ivp_sv2nx8_i    v5,a4,-64         # [3*II+1]  id:46
    ivp_lv2nx8_ip    v1,a3,128         # [0*II+1]  id:45
}
{ # format F0
    ivp_lv2nx8_i    v3,a2,-64         # [0*II+2]  id:44
    ivp_packvru2nx24 v5,wv1,a6          # [2*II+2]
    ivp_mulusp2n8xr16 wv1,v4,v3,a11   # [1*II+2]
    nop               #
}
{ # format N2
    ivp_sv2nx8_ip    v2,a4,128         # [2*II+3]  id:46
    ivp_lv2nx8_i    v4,a3,-64          # [0*II+3]  id:45
}
```

- On the left you see the assembly code for 4 (FLIX) instructions
- Instructions from formats F0 (128 bit) and narrow formats N2 (64 bit) are intermixed
- The compiler could have chosen instructions from format F0 for the 2'nd and 4'th instruction as well, by inserting 2 NOP into two slots
- Instead the compiler chooses instructions from format N2 to minimize code size

# Vision P6 Register Types

Register File Name	Bit Width	Number of Entries	Typical Usage
AR	32	32/64 (configurable)	Addresses and Scalar Data Types
BR	1	16	Boolean Values
VEC	512 (32 * 16b)	32	Pixel Data Values
WVEC	1536 (32 * 48b)	4	Intermediate Values
VBOOL	64 (64 * 1b)	8	Vector Boolean Predicates
ALIGN	512 (32 * 16b)	4	Pixel Stream Alignment Data

# Vision P6 Register Usage

Type	REG Size (bits)	Number of REGs	Element Width (bits)	Number of Elements	Typical Usage
VEC	512	32	8	64	input/output values
			16	32	
			32	16	
WVEC	1536	4	24	64	intermediate computation results
			48	32	
			64	16	
VBOOL	64	8	1	64	Boolean value typically generated by comparisons, used by predicated ops
			2	32	
			4	16	
VALIGN	512	4	16	32	Facilitate aligning Load and Store Operations
GSR	512	4	16	32	Gather Operations

# Vector Operation Categories

- **Arithmetic**
  - Examples are +,-,<, min/max
  - Usually VEC input and output
  - Comparison Operations usually have a VBOOL output
- **Multiply and Divide**
  - Many combinations of WVEC, VEC and AR types for input
  - Optional Versions operating on Half/Single Precision Float Data types
- **Shift**
  - Usually VEC input and output
  - Some versions fuse pack with shift operations to move from WVEC to VEC
- **Logical**
  - VEC input and output

# Vector Operation Categories

- **Select** (Data Reordering)
  - VEC input and output
  - allows **arbitrary permutation** of vector elements
  - Subcategories
    - **SEL**: General Select Operation with select control in VEC register
    - **SELI**: Constrained Select operation with select control via immediate value
    - **DSEL**: General Select Operation producing two (double) VEC register outputs

# Vector Operation Categories

- **Move/Convert**
  - move data between register types (WVEC, VEC, AR)
  - Conversion operations change the data format (integer, float, etc.)
- **Load and Store (L/S)**
  - **normal L/S** require address **alignment** to **size of VEC register**
  - **aligning L/S** allow **addresses** aligned to arbitrary **byte boundary**
- **Predicated** (more details will follow)
  - VEC register output
  - Operations controlled via VBOOL register
- **Reduction Operations** (more details will follow)
  - AR register Output, VEC register input

# Operation Variants: Predicated and Reduction

- Most vector operations
  - have input and output **vectors**
  - perform the **same operation** on every **data element**
  - Perform computations where different **elements** locations (**lanes**) do not cross.
- Exceptions are
  - **Reduction Operations**
  - **Predicated Operations**
- We will discuss those in the next slides

# Reduction Operations

Reduction operations operate on **vectors** and return a **scalar** value

Examples:

- IVP\_RADD – sum up all elements in a vector
  - An example will follow in a few slides
- IVP\_RMIN/MAX – find the smallest/largest element in a vector

# Predicated Operations

- Predicated Operations take a **VBOOL input register**
- Operations for each element are performed **conditionally**
- This is controlled by the **bit value** for each element
- These operations allow **vectorization** of code with **if/else** conditions
- The next slide will show an example

# Example using Predicated Reduction Operation IVP\_RADD2NX8T

- operation IVP\_RADD2NX8T adds up the  $2N$  8-bit elements of the input vector and writes the 16-bit output value into a scalar register
- For each element the operation is performed conditionally, depending on the bit value in a Boolean  $2N$ -bit mask
- Elements that have its corresponding bit value set to 0 will be skipped and don't contribute to the final output value

```
/* 64 bit mask used for vool2N c-type*/
uint64_t b_mask = 0xc000000000000002ULL;

xb_vec2Nx8 arg_in0; /* Input: Vector c-type holding 64-way, 8-bit signed integer */
vbool2N arg_in1; /* Input: Vector boolean c-type, holding 64-way 1-bit Boolean value*/
xb_int16 arg_out; /* Output: Vector c-type holding scalar 16-bit signed integer */

/* initialize arg_in0 to (2N-1,2N-2,...,1,0) */
arg_in0 = IVP_SEQ2NX8();

/* Load 64, 8-bit Signed integer into vector register */
arg_in1 = *((vbool2N *)&b_mask);

/* Add 64 8-bit signed values from input for which mask bit is 1
 * to produce signed 16-bit result. This operation does not produce
 * saturated output */
arg_out = IVP_RADD2NX8T(arg_in0,arg_in1);
```

# Summary

In this module, you

- Became familiar with the Vision P6 operation categories
- Learned about SIMD/FLIX and instruction formats

This should give you a basic understanding of

- Operations that can execute in parallel (FLIX)
- How many output results are produced per cycle (SIMD)

In the next module “**Module 5 – Estimating Performance**” you will **apply** that knowledge in order to generate algorithm performance **estimates**.

# Quiz

1. Vision P6 supports operations on
  - a) 8 bit data, but not 32 bit data
  - b) 16 bit data, but not 8 bit data
  - c) 8, 16 and 32 bit data
2. Reduction Operations
  - a) Reduce a scalar to a vector
  - b) Reduce a vector to a scalar
  - c) Subtract two vectors
3. Vision P6 can issue
  - a) 1 operation per cycle
  - b) 5 operations per cycle
  - c) 1,2,4 or 5 operations per cycle, depending on the instruction format

Answers: 1c, 2b, 3c

# Estimating Performance

**Module** 5

**Revision** 1.0

**Version** 7.4

**Estimated Time:**

• For the lecture 30 minutes

• For the lab N/A

**Dependencies:**

- None

# Module Objectives

In this module, you will

- Learn why performance estimation is an important first step when porting code to Vision P6
- Get familiar with common performance units we use with Vision P6
- Learn to estimate performance of a given algorithm by understanding the HW resource limitations

# Understanding the number of concurrent operations

Vision P6 has 5 operation slots,

- ALU Operations can be issued in 4 slots
- MULTIPLY only in one slot
- SELECT operations only in 1 slot

This implies that a Vision P6 instruction

- can issue at most 5 concurrent operations

Of these 5 operations only

- 2 can be a Vector Load
- 1 can be a Vector Store
- 4 can be Vector ALU Operations
- 1 can be a Vector Select Operation
- 1 can be a Multiply/Divide Operation

# Understanding data (SIMD) throughput

The number of computed vector elements depends on the data width:

Data Width (bit)	Operation Notation	Throughput (vector elements)
8	$2N$	64
16	$N$	32
32	$N_2$	16

- The throughput is inversely proportional to data width

# More on Performance for Select Operations

- We have two type of Select Operations
  - **general** SEL operations with select control comes from a VEC register
  - **SELI** operations with select control from an **immediate** field
- We only have one issue slot for **general** select Operations (slot 3)
- We have certain specialized SELI operations (IVP\_SEL2NX8I\_S{0,1,4}) can issue in other slots (slot0, slot1, slot4) as well
- We have a **general DSEL** operation (**D is for double**) with two vector outputs
  - They only exist for 16-bit data with (vector length N)
- Therefore the **general SEL/DSEL** throughput for both 8 and 16 bit data is: **2N (64)**
- **DSELI** operations throughput for 8-bit data is: **4N (128)**
  - Only a few immediate patterns exist (e.g., interleave/de-interleave)
- For SELI the theoretical throughput for 8-bit data is quite high:  $1 \text{ DSELI} + 3 \text{ SELI} = 4N + 3*2N = 10N$  (320)
- Likely it is not possible to reach that limit for any useful use case (select patterns are limited), but it is certainly possible to exceed that  $4N$  limit.

# More on Performance for Select Operations ... continued

The performance of SEL operations is summarized in the table below

- The “X” for DSEL with operand width 8-bit indicates that we don’t have a DSEL operation for that data type
- The “?” for the 16/32 bit types for DSELI/SELI operations indicate that it is not clear that useful select immediate pattern are available for your problem at hand.

SEL/cycle	Operand Bit-Width		
SEL Type	8	16	32
SEL	64	32	16
DSEL	X	64	32
DSELI	128	?	?
DSELI/SELI combo	320	?	?

# More on Performance for Multiply/Multiply-Accumulate (MAC) Operations

Vision P6 only has one issue slot for MAC Operations (slot 2) which can issue 3 different types of MAC operations

- General MAC

- Output WVEC, Input from WVEC, VEC, AR

- Paired MAC
  - Operates on a pair of input registers
  - doubles the MUL throughput
- Quad MAC
  - Operates on a quadruple of input registers
  - quadruples the MUL throughput

MUL/cycle	Operand Bit-Width			
MUL Type	8x8 integer	8x16 integer	16x16 integer	32x32 float
General MAC	64	64	32	16
Paired MAC	X	128	64	X
Quad MAC	256	X	X	X

# Simplifying Assumptions for Estimating Performance

## 1. Ignore Choice of Instruction and Instruction Scheduling

- The details of which operations can be issued concurrently is complicated. It depends on
  - Instruction Formats (operation slotting)
  - Register File Ports
  - Operation Schedules
  - Compiler

## 2. Ignore Vector Load/Store bandwidth

- Although we have seen L/S bound examples (MUL example) most realistic use cases fuse calculations, relieving L/S bandwidth

The purpose of the estimate is to come up with a **best case optimal performance** target.

Later on when we assess the performance of code we will compare against this target.

# Examples of Operation Throughputs

## ALU:

- 4 concurrent, 64 element 8-bit = 256 8-bit ops per cycle
- Or, 4 concurrent, 16 element 32-bit = 64 32-bit ops per cycle

## MUL:

- 64 8-bit MUL ops per cycle
- Or, 64 8-bit \* 16-bit MUL ops per cycle

## LOAD

- 2 16-bit 32-element LOAD ops = 128 Bytes per cycle

## General SELECT

- One (Double) Select Operation with 64 16-bit outputs per cycle

# Putting Performance Estimation to the Test

In the slides that follow we will look at three Examples

1. LookupTable
2. 4x4 Matrix Multiply
3. Integral Image

In these example we will see our estimation procedure working to different degrees of satisfaction

- In the LookupTable example estimation will **align with achieved results**
- For the Matrix Multiply achieved results will **fall short** of estimation due to challenges with **instruction scheduling**
- The Integral Image performance will fall short due to **data dependencies** that make vectorization more **challenging than expected**

# Example: Table Lookup (LUT)

```
void table128x8LookupRef(int8_t *output, int8_t *input)
{
    int offset, i;
    for (i=0; i<DATA_SIZE; i++)
    {
        offset=input[i];
        output[i]=table128x8[offset];
    }

void table128x8LookupOpt(int8_t *output, int8_t *input)
{
    int i;
    xb_vec2Nx8U offsetVec, lookupVec, tableVec, *tPtr;
    xb_vec2Nx8U ptL, ptH;
    xb_vec2Nx8U *inputVec = (xb_vec2Nx8U *) input;
    xb_vec2Nx8U *outputVec = (xb_vec2Nx8U *) output;

    ptL = * (xb_vec2Nx8U *) &table128x8[0];
    ptH = * (xb_vec2Nx8U *) &table128x8[64];

    for (i=0; i<DATA_SIZE/SIMD_2N; i++)
    {
        offsetVec = *(inputVec++);
        lookupVec = IVP_SEL2NX8(ptH, ptL, offsetVec);
        *(outputVec++) = lookupVec;
    }
}
```

Ref code performs table lookup of 128 entry table with 8-bit entries

## Our Performance Estimate:

**One 64 element 8-bit output vector per cycle** (1 SEL Slot)

## Achieved performance of optimized Vision P6 code:

- 4 cycles, with unroll=4
- **4 vectors** with 64 8-bit outputs are produced in **4 cycles**, or **1 output vector per cycle**

**The estimated and achieved performance match !!!**

# Example: 4x4 float Matrix Multiply

```
/*
 * The function Mat_Mul4x4 below implements a 4x4 floating point matrix multiplication C= A*B
 *
 * |a00  a01  a02  a03|   |b00  b01  b02  b03|
 * |a10  a11  a12  a13| * |b10  b11  b12  b13| =
 * |a20  a21  a22  a23|   |b20  b21  b22  b23|
 * |a30  a31  a32  a33|   |b30  b31  b32  b33|
 *
 * |a00*b00+a01*b10+a02*b20+a03*b30      a00*b01+a01*b11+a02*b21+a03*b31      a00*b02+a01*b12+a02*b22+a03*b32      a00*b03+a01*b13+a02*b23+a03*b33|
 * |a10*b00+a11*b10+a12*b20+a13*b30      a10*b01+a11*b11+a12*b21+a13*b31      a10*b02+a11*b12+a12*b22+a13*b32      a10*b03+a11*b13+a12*b23+a13*b33|
 * |a20*b00+a21*b10+a22*b20+a23*b30      a20*b01+a21*b11+a22*b21+a23*b31      a20*b02+a21*b12+a22*b22+a23*b32      a20*b03+a21*b13+a22*b23+a23*b33|
 * |a30*b00+a31*b10+a32*b20+a33*b30      a30*b01+a31*b11+a32*b21+a33*b31      a30*b02+a31*b12+a32*b22+a33*b32      a30*b03+a31*b13+a32*b23+a33*b33|
 *
```

- We will vectorize the matrix multiplication problem, by creating the 16 entries of the output matrix C in one VEC register, holding 16 (N\_2) floats
- Each entry is a sum of 4 products (since the Matrix C has 4 rows and cols)
- According to the notation above the first product (of 4) can be written as  
(a00,a00,a00,a00,a10,a10,a10,a20,a20,a20,a20,a30,a30,a30,a30)

X

(b00,b01,b02,b03, b00,b01,b02,b03, b00,b01,b02,b03, b00,b01,b02,b03,)

- This requires 1 VEC MUL and 2 VEC SEL (or 1 VEC DSEL) operations

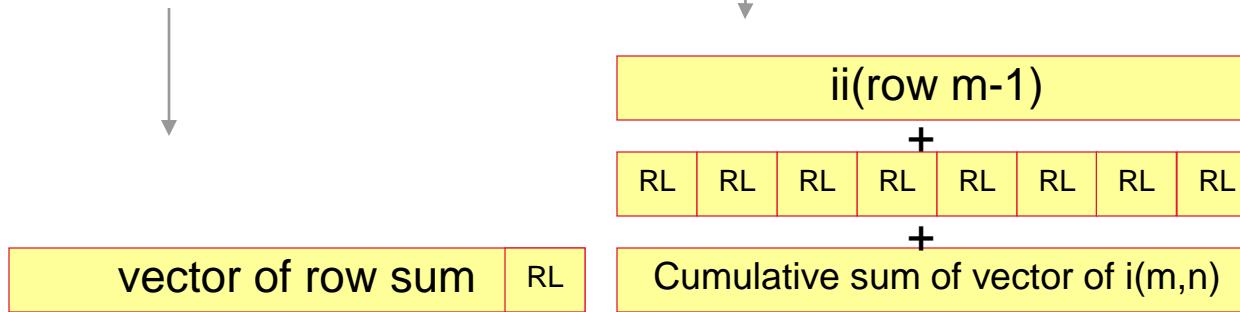
# Example: 4x4 float Matrix Multiply ... continued

- We have to create and add 4 of these vector products to compute the output matrix C.
- This will require 4 MUL, 4 DSEL and 3 ADD operation which we should be able to schedule in 4 cycles
- We predict that it takes **4 cycles** to create one output Matrix C
- Unfortunately our implementation did not achieve that.
- DSEL (format F0 and F4) and floating point MUL (format F1 and F2) operation did not schedule together as there is no format that contains them both.
- We achieved the best schedule by mixing DSEL and SEL operations and achieved a schedule of **6.25 cycles** (25 cycles for computing 4 Matrices)
- The code for the example is in the project fp32MatrixMultiply4x4 of the SwP

# Example: Integral Image Performance

$\text{ii}(m,n) = \sum_{k=0}^m \sum_{l=0}^n i(k,l)$  ii is sum over all pixels to the left and above

$\text{ii}(m,n) = \sum_{l=0}^n i(m,l) + \text{ii}(m-1,n)$  It can be decomposed like this



You can compute on vector of the integral image, by adding

1. The output vector of the integral image from the row above
2. The vector containing the N-times replicated last column, from the vector on the left
3. The vector containing the cumulative sum of the original image at the current vector location

# Example: Integral Image Performance ... continued

## Performance Estimation:

The picture in the previous slides implies the following operations:

- 2 ADD
- 1 ADD cumulative sum
- 1 Replication (vec of RL)
- 2 LD
- 1 ST

For a total of 4 ALU and 3 LD/ST which should fit into 2 cycles

The estimated performance is **2 cycles** per 32 16-bit element vector output.

## Actual Performance

The actual performance is **4.5 cycles** per output vector (9 cycles for two vectors, with unroll of 2).

The reason is that the cumulative sum calculation takes more than **1 cycle**.

There is a data dependency which requires **5-step ( $\log_2(32)=5$ ) iteration of SEL and ADD operations.**

In Vision P6 some of the SEL and SELI operations can overlap resulting in the cycle count of **4.5** instead of **5** cycles mentioned above.

# Understanding Performance Units

## Cycles Per Pixel (CPP)

- It is convenient to normalize the cycle count of an algorithm by the number of pixels
- Once you have this number it is easy to compute the cycles for different image sizes by multiplying CPP with the image dimensions
- Example: Let's assume it takes 2 Million Cycles to run a given filter on an the 3 color components (RGB) of an image of size 640 x 480 image:

$$\text{CPP} = 2,000,000 / (640 * 480) = 6.51$$

- The CPP per **color component** is one third of that:  $2.17 = 6.51/3$

## Frame Rate (unit: FPS = Frames Per Second)

- Often it is important to know whether Vision P6 can perform an algorithm at a certain frame rate.
- In this case you have to know the clock frequency for the processor:

$$\text{FPS} = \text{clockFreq} / (\text{CPP} * \text{ImageWidth} * \text{ImageHeight})$$

# Summary

In this module, you learned

- why performance estimation is an important first step when porting code to Vision P6
- became familiar with common performance units we use with Vision P6
- went through the performance estimation with 3 specific examples

Keep in mind that ...

- Performance estimation is very important in order to set a **performance target** for an algorithm
- In general your estimation will be **optimistic** (details of available ALU operations are ignored, e.g. Integral Image example).
- Getting **close** to the performance **target** is a good **stoppage point for optimization efforts**.

Now that you know how to **estimate**, we have to teach you how to **program Vision P6**. That will be the subject of the **next 3 modules**.

# Quiz

1. How many operation issue slots does Vision P6 have
  - a) 3
  - b) 4
  - c) 5
2. The Vision P6 core can simultaneously issue
  - a) 3 ALU operations
  - b) 1-4 ALU operations depending on the instruction format
  - c) 5 ALU operations
3. CPP stands for
  - a) Computed Performance Profile
  - b) Cycles Per Pixel
  - c) Calculated Performance Profile

Answers: 1c, 2b, 3b



# Auto-Vectorization of C-Code

**Module** 6

**Revision** 1.0

**Version** 7.4

**Estimated Time:**

- For the lecture 30 minutes
- For the lab 60 minutes

**Dependencies:**

- None

# Module Objectives

In this module, you will

- Work with C source code
- Use the auto-vectorizing Compiler
- Consult the Vectorization Assistant for feedback
- Find and analyze critical sections of assembly code
- Study the Software Pipelining Report
- Learn to use compiler hints (keywords) in your source code

# Code Vectorization

Code Vectorization is **key** to high performance on Vision P6!

Programmer can chose from 4 levels of abstraction:

- 1. Auto vectorization of C Code**
- 2. Auto vectorization of C Code using compiler hints**
- 3. Vectorization using vector types and operator overloading**
- 4. Vectorization using vector types and operation intrinsics**

The choice depends on the

- nature of the application
- experience of the programmer
- the performance goal

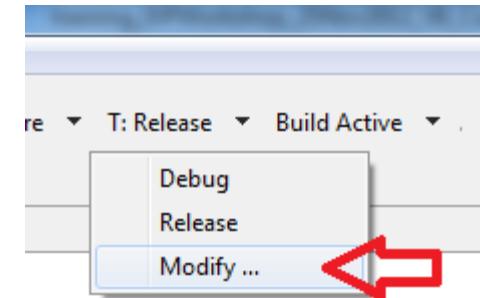
# Auto Vectorization of ANSI C-Code

- Compiler vectorizes standard ANSI C code
- No code modification is required
- No pragmas and compiler hints are added
- Might be appropriate for code that is not performance critical
- Requires **Optimization Level 3 (-O3)**
- Requires “**Enable Automatic Vectorization**” to be set (**-LNO:simd**)

# Compiler Options for Auto Vectorization

## Set Compiler Options

- “Enable Auto vectorization” and “Optimization Level O3”
- Enable options to keep **intermediate** compilation files and produce **w2c.c** file

A screenshot of the 'Build Properties for Example' dialog. The 'Target' is set to 'Release'. The 'Optimization' tab is selected, showing various compiler options. Several options are circled in red:

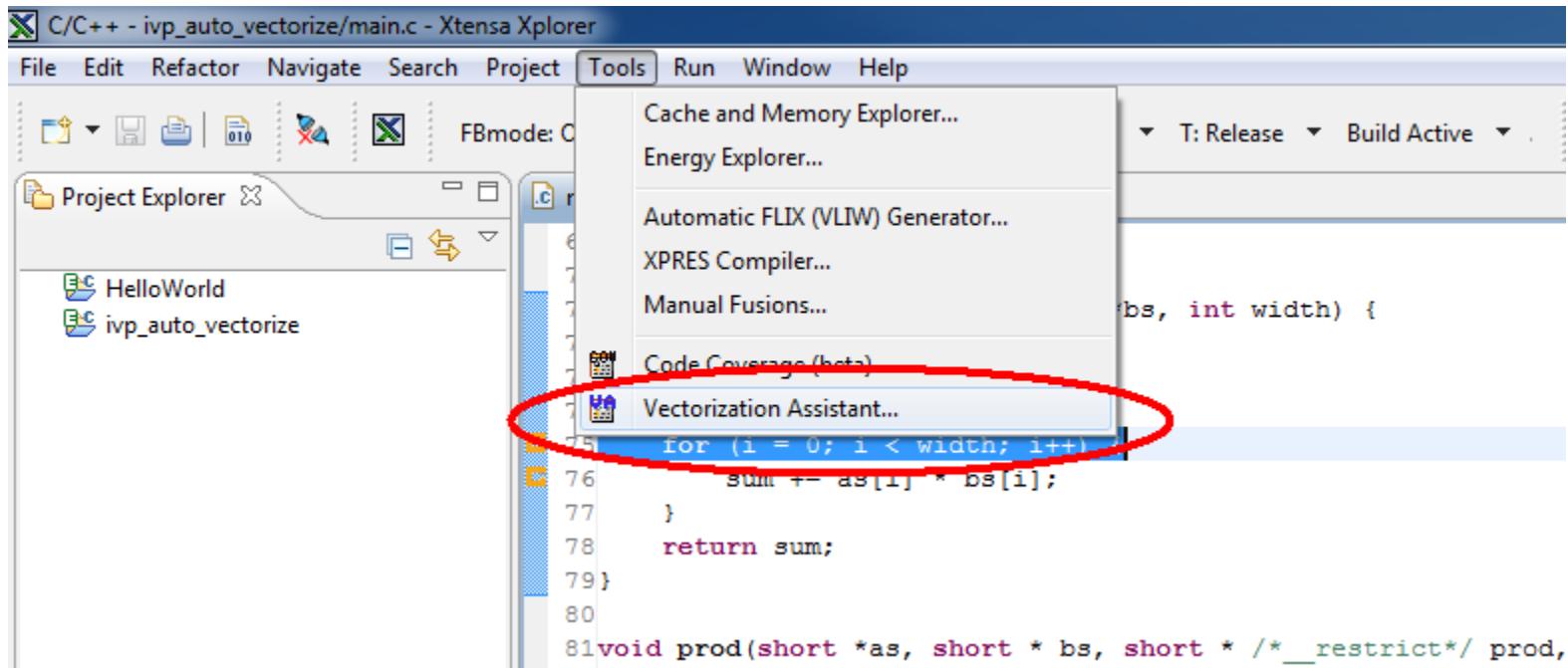
- 'Optimization Level' is set to 'O3'.
- 'Keep intermediate compilation files' is set to 'Yes'.
- 'Enable automatic vectorization' is set to 'Yes'.
- 'Produce a .w2c.c file' is set to 'Yes'.

The 'Template' section shows 'Apply...' and 'Save...'. Other tabs like 'Include Paths' and 'Symbols' are visible but not selected.

# Vectorization Assistant

Select the Vectorization Assistant from the Tools Menu

- opens a view with compiler **feedback** regarding the **vectorization** of loops



# Auto Vectorization Example 1: Dot Product

```
main.c X
63
64     short dot_prod(short *as, short *bs, int width) {
65         int i;
66
67         short sum = 0;
68         for (i = 0; i < width; i++) {
69             sum += as[i] * bs[i];
70         }
71         return sum;
72     }
73 }
```

The function ***dot\_prod*** vectorizes without any source code modifications

Vectorization Assistant reports that loop at line 68 vectorized

File/Line	Function	Cycles	Instructions	V...	MsgId	Message
main.c:59	init	0	0	n	SIMD_SCALAR_DEPENDENCE	Bad scalar dependences (variable 'j').
main.c:59	init	0	0	n	SIMD_SCALAR_DEPENDENCE	Bad scalar dependences (variable 'j').
main.c:59	init	0	0	n	SIMD_SCALAR_DEPENDENCE	Bad scalar dependences (variable 'j').
main.c:59	init	0	0	n	SIMD_SCALAR_DEPENDENCE	Bad scalar dependences (variable 'j').
main.c:59	init	0	0	n	SIMD_SCALAR_DEPENDENCE	Bad scalar dependences (variable 'j').
main.c:68	dot_prod	0	0	y	SIMD_LOOP_VECTORIZED	Loop 1 is vectorized by 32.
main.c:82	prod	0	0	n	SIMD_LOOP_NON_VECTORIZABLE	Loop is not vectorizable.
main.c:82	prod	0	0	n	SIMD_LOOP_VECTORIZATION_RETRY	Retrying loop vectorization by 16 (next priority vector length)
main.c:82	prod	0	0	n	SIMD_LOOP_VECTORIZATION_RETRY	Retrying loop vectorization by 64 (next priority vector length)
main.c:83	prod	0	0	n	SIMD_ARRAY_ALIAS	Array base 'prod' is aliased with array base 'as' at line 83.

# Auto Vectorization Example 1: Dot Product Profile Disassembly View:

The screenshot shows a development environment with three main windows:

- Code Editor (main.c):** Displays the C source code for a dot product function. The line `sum += as[i] \* bs[i];` is highlighted.
- Profile Disassembly:** A table showing assembly instructions for the `dot\_prod` function. The first few rows are:
 

Count	Address	Instruction
17	600005a2	{ { ivp_la2nx8_ip v4, u1, a3 nop nop }
15	600005aa	dot prod+0x7e { { ivp_la2nx8_ip v0, u0, a2 nop ivp_mulanx16packl v2, v1, v0 }
15	600005b2	{ { ivp_la2nx8_ip v1, u1, a3 nop nop }
15	600005ba	{ { ivp_la2nx8_ip v3, u0, a2 nop ivp_mulanx16packl v2, v4, v3 }
- Function Call Graph:** A table showing the total percentage of cycles for each function. The `dot\_prod` function is highlighted with a red border.

Function Name	Total (%)	Function	Children	Total	Called	Size (bytes)
prod	16.03	5,129	0	5,129	1	53
_ResetHandler	0.73	236	0	236	<spontaneous>	127
fir5	0.67	217	0	217	1	1,353
_start	0.50	161	31,561	31,722	1	140
dot_prod	0.29	93	0	93	1	292
main	0.14	46	31,153	31,199	1	166
exit	0.14	45	64	109	1	138
atexit	0.12	41	104	145	1	86
_clibrary_init	0.12	39	214	253	1	55
_Initlocks	0.10	32	19	51	1	111
_do_global_dtors_aux	0.09	31	0	31	1	75

# Auto Vectorization Example 1: Dot Product

## Disassembly File main.s

- Loop unrolled by 2
- 2 MUL and 4 LD in 4 cycles
- From a **HW resource** perspective 2 MUL and 4 LD could schedule in **2 cycles**
- The SWP report shows a **recurrence** limit of 4 cycles, and therefore the this auto-vectorized code only **schedules in 4 cycles**

```
#<swps> 4 cycles per pipeline stage in steady state with unroll=2
#<swps> 2 pipeline stages
#<swps> 6 real ops (excluding nop)
#<swps>
#<swps>          2 cycles lower bound required by resources
#<swps>  min   4 cycles required by recurrences
#<swps>  min   4 cycles required by resources/recurrence
#<swps>  min   6 cycles required for critical path
#<swps>          7 cycles non-loop schedule length

#<swps>  register file usage:
#<swps>    'a' total 2 out of 16 [2-3]
#<swps>    'v' total 5 out of 32 [0-4]
#<swps>    'u' total 2 out of 4 [0-1]
#<swps>

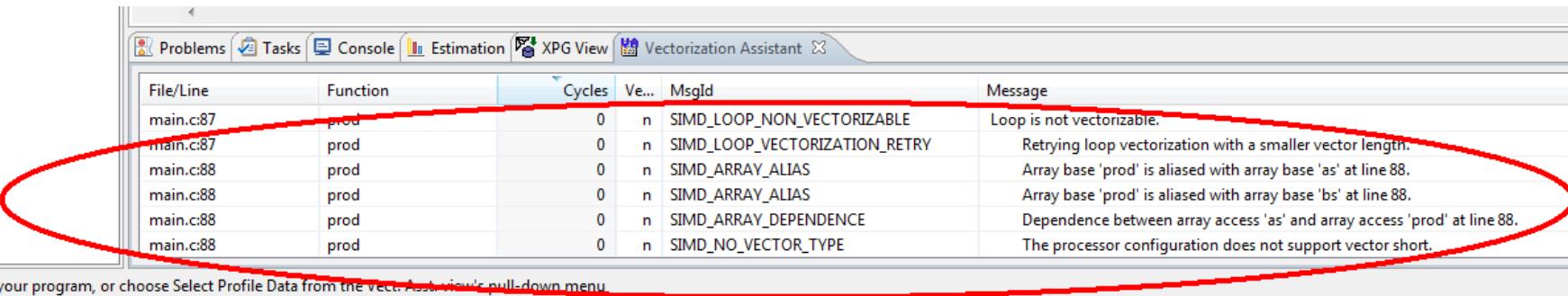
#<freq> BB:109 => BB:109 probability = 0.98000
#<freq> BB:109 => BB:119 probability = 0.02000
    .frequency 0.808 40.789
{  # format N1
    ivp_la2nx8_ip    v0,u0,a2          # [0*II+0]  id:23
    nop
    ivp_mulanx16packl  v2,v1,v0       # [1*II+0]
}
{  # format N1
    ivp_la2nx8_ip    v1,u1,a3          # [0*II+1]  id:22
    nop
    nop
}
{  # format N1
    ivp_la2nx8_ip    v3,u0,a2          # [0*II+2]  id:23
    nop
    ivp_mulanx16packl  v2,v4,v3       # [1*II+2]
}
{  # format N1
    ivp_la2nx8_ip    v4,u1,a3          # [0*II+3]  id:22
    nop
    nop
}
```

# Auto Vectorization Example 2: Product

```
main.c
82 }
83
84 void prod(short *as, short * bs, short * prod, int width) {
85     int i;
86
87     for (i = 0; i < width; i++) {
88         prod[i] = as[i] * bs[i];
89     }
90 }
91
```

The function ***prod*** does **not** vectorize without source code modifications

Vectorization Assistant reports array ***prod*** aliases with arrays ***as*** and ***bs*** on line 88.



File/Line	Function	Cycles	Ve...	MsgId	Message
main.c:87	prod	0	n	SIMD_LOOP_NON_VECTORIZABLE	Loop is not vectorizable.
main.c:87	prod	0	n	SIMD_LOOP_VECTORIZATION_RETRY	Retrying loop vectorization with a smaller vector length.
main.c:88	prod	0	n	SIMD_ARRAY_ALIAS	Array base 'prod' is aliased with array base 'as' at line 88.
main.c:88	prod	0	n	SIMD_ARRAY_ALIAS	Array base 'prod' is aliased with array base 'bs' at line 88.
main.c:88	prod	0	n	SIMD_ARRAY_DEPENDENCE	Dependence between array access 'as' and array access 'prod' at line 88.
main.c:88	prod	0	n	SIMD_NO_VECTOR_TYPE	The processor configuration does not support vector short.

your program, or choose Select Profile Data from the Vect. Ass't View's pull-down menu

# Auto Vectorization Example 2: Product Profile Disassembly View

Count	Address	Instruction
4	60000648	prod entry a1, 32
1	6000064b	nop
1	6000064e	nop.n
1	60000650	{ { loopgtz a5, 6000067b <prod+0x33> or a6, a4, a4 or a4, a2, a2 mov.n a2, a3 } }
1,025	60000660	{ { l16si a3, a2, 0 l16si a5, a4, 0 addi a2, a2, 2 addi a4, a4, 2 } }
3,072	60000670	mul16s a3, a5, a3
1,024	60000673	{ { s16i a3, a6, 0 nop addi a6, a6, 2 } }
1	6000067b	prod+0x33 retw.n

- Very large cycle count
- We see addi.n and mul16s operations
- These are **Xtensa Base** operations not **Vision P6 vector** operations

# Auto Vectorization of C Code Using Compiler Hints

- Programmer provides hints to compiler to help auto vectorization.
- This style is often sufficient to achieve high performance
- Requires minimal amount of code modification
- Requires that compiler can resolve data access patterns

Example:

*Keyword: **\_\_restrict***

- Identify which output arrays do not alias with inputs:

***#pragma aligned (v1, 64)***

- Identify when data is aligned to 64 bytes:

# Auto Vectorization with Compiler Hints

## Example 2: Product

The screenshot shows a code editor window titled "main.c". The code defines a function "prod" that takes pointers to short arrays "as" and "bs", a pointer to a short array "prod" marked with `__restrict`, and an integer "width". It uses `#pragma aligned` directives to align the arrays. A tooltip for the variable "short \*as" is visible, stating "Press 'F2' for focus".

```
73
74 void prod(short *as, short * bs, short * __restrict prod, int width) {
75     int i;
76
77     #pragma aligned (as, 64)
78     #pragma aligned (bs, 64)
79
80     for (i = 0; i < width; i++) {
81         prod[i] = as[i] * bs[i];
82     }
83 }
84
```

- With keyword `__restrict` the function `prod` vectorizes
- `#pragma aligned` informs the compiler that aligning loads are not needed

### Vectorization Assistant:

main.c:68	dot_prod	0	0	y	SIMD_LOOP_VECTORIZED	Loop 1 is vectorized by 32.
main.c:80	prod	0	0	y	SIMD_LOOP_VECTORIZED	Loop 2 is vectorized by 32.
main.c:99	fir5	0	0	y	SIMD_LOOP_VECTORIZED	Loop 3 is vectorized by 32.
main.c:148	gauss3x3	0	0	y	SIMD_LOOP_VECTORIZED	Loop 4 is vectorized by 32.
main.c:150	gauss3x3	0	0	n	SIMD_LOOP_NON_VECTORIZABLE	Loop 5 is not vectorizable.

# Auto Vectorization with Hints Example 2: Product

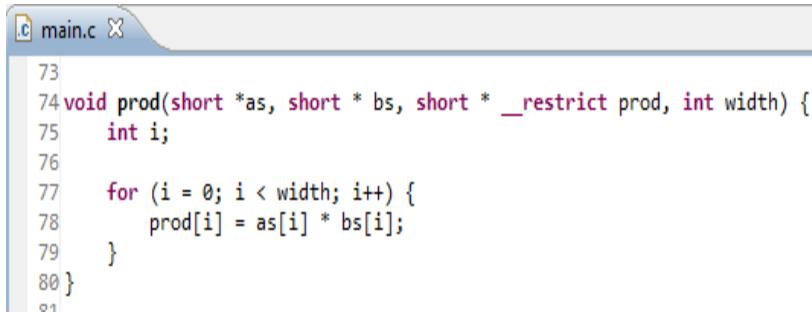
The screenshot shows the Xilinx Vivado IDE interface with the following details:

- Title Bar:** C/C++ - ivp\_auto\_vectorize/bin/VP6\_Alpha3/Release/main.s - Xtensa Xplorer
- Toolbar:** File Edit Source Refactor Navigate Search Project Tools Run Window Help
- Project Explorer:** Shows the project structure under "HelloWorld". The file "main.s" is selected and highlighted with a red box.
- Code Editor:** Displays the assembly code for "main.s". A specific line, "line 80", is highlighted with a red box. The code includes various vectorization hints and comments explaining the generated assembly.
- System Overview:** Shows configurations, subsystems, and TIE source.
- Console:** Displays the command line used to run the simulation and the path to the executable.
- Bottom Taskbar:** Shows system icons and the date/time (2:32 PM, 9/28/2016).

```
529 .LBB141_prod: # 0x2c7
530 #<loop> Loop body line 80, nesting depth: 1, estimated kernel iterations: 23
531 #<loop> unrolled 4 times
532 #<swps>
533 #<swps> 6 cycles per pipeline stage in steady state with unroll=4
534 #<swps> 3 pipeline stages
535 #<swps> 16 real ops (excluding nop)
536 #<swps>
537 #<swps> 6 cycles lower bound required by resources
538 #<swps> min 4 cycles required by recurrences
539 #<swps> min 6 cycles required by resources/recurrence
540 #<swps> min 10 cycles required for critical path
541 #<swps> 13 cycles non-loop schedule length
542
543 #<swps> register file usage:
544 #<swps> 'a' total 3 out of 16 [2-4]
545 #<swps> 'v' total 11 out of 32 [0-10]
546 #<swps> 'u' total 1 out of 4 [0]
547 #<swps>
548 #<freq> BB:141 => BB:141 probability = 0.96000
549 #<freq> BB:141 => BB:153 probability = 0.04000
550 .frequency 0.000 16.035
551 { # format F1
552     ivp_sa2nx8_ip v10,u0,a4      # [2*II+0] id:29
553     ivp_lv2nx8_ip v0,a2,192      # [0*II+0] id:28
554     ivp_mulnx16pack1 v4,v3,v2    # [1*II+0]
555     nop
556 }
557 { # format F1
558     ivp_sa2nx8_ip v1,u0,a4      # [1*II+1] id:29
```

# Auto Vectorization with Hints

## Example 2: Product



```
73
74 void prod(short *as, short * bs, short * __restrict prod, int width) {
75     int i;
76
77     for (i = 0; i < width; i++) {
78         prod[i] = as[i] * bs[i];
79     }
80 }
```

- Schedule length is **6 cycles** with **unroll of 4**
- Unroll of 4 means that one iteration of the assembly loop corresponds to 4 iterations in the C loop

### Compiler Output (inner loop):

```
#<loop> Loop body line 77, nesting depth: 1, estimated kernel iterations: 23
#<loop> unrolled 4 times
#<swps>
#<swps> 6 cycles per pipeline stage in steady state with unroll=4
#<swps> 3 pipeline stages
#<swps> 16 real ops (excluding nop)
#<swps>
#<swps> 6 cycles lower bound required by resources
#<swps> min 4 cycles required by recurrences
#<swps> min 6 cycles required by resources/recurrence
#<swps> min 10 cycles required for critical path
#<swps> 16 cycles non-loop schedule length

#<swps> register file usage:
#<swps> 'a' total 3 out of 16 [2,4,15]
#<swps> 'v' total 10 out of 32 [0-9]
#<swps> 'u' total 3 out of 4 [0-2]
#<swps>
#<freq> BB:133 => BB:133 probability = 0.96000
#<freq> BB:133 => BB:146 probability = 0.04000
```

# Auto Vectorization with Hints ... continued

## Example 2: Product

- On the left you see the 6 assembly instructions from the inner loop of function **prod**
- 4 multiply, 8 loads and 4 stores
- We have **2 load or store operations in every cycle**
- Loop is load/store bound**

Compiler Output (inner loop):

```
{# format F1
    ivp_sa2nx8_ip v5,u2,a4      # [2*ll+0] id:22
    ivp_la2nx8_ip v0,u0,a15     # [0*ll+0] id:21
    ivp_mulnx16packl v2,v1,v0   # [1*ll+0]
    nop                      #
}
{# format N2
    ivp_sa2nx8_ip v7,u2,a4      # [2*ll+1] id:22
    ivp_la2nx8_ip v7,u1,a2     # [1*ll+1] id:20
}
{# format F1
    ivp_la2nx8_ip v3,u0,a15     # [0*ll+2] id:21
    ivp_la2nx8_ip v4,u1,a2     # [1*ll+2] id:20
    ivp_mulnx16packl v5,v4,v3   # [1*ll+2]
    nop                      #
}
{# format N2
    ivp_sa2nx8_ip v9,u2,a4      # [2*ll+3] id:22
    ivp_la2nx8_ip v1,u1,a2     # [0*ll+3] id:20
}
{# format F1
    ivp_sa2nx8_ip v2,u2,a4      # [1*ll+4] id:22
    ivp_la2nx8_ip v6,u0,a15     # [0*ll+4] id:21
    ivp_mulnx16packl v7,v7,v6   # [1*ll+4]
    nop                      #
}
{# format F1
    ivp_la2nx8_ip v4,u1,a2     # [0*ll+5] id:20
    ivp_la2nx8_ip v8,u0,a15     # [0*ll+5] id:21
    ivp_mulnx16packl v9,v4,v8   # [1*ll+5]
    nop                      #
}
```

# Auto Vectorization with Hints Example 3: Floating Point 5 Tap Filter

The screenshot shows a code editor window with a tab labeled "vectorizationFunctionsAuto.c". The code contains several `#pragma aligned` and `#pragma vectorize` directives. The code itself is a function named `Filt5TapVectorAuto` that performs a 5-tap filter operation on a vector of floats.

```
140 void Filt5TapVectorAuto(float *pcoeff,
141                           float *pvecin,
142                           float *pvecout,
143                           int32_t veclen
144                         )
145 {
146     float(*__restrict pc) = pcoeff;
147     float(*__restrict pv) = pvecin;
148     float(*__restrict pf) = pvecout;
149     float temp;
150     int32_t indx;
151 #pragma aligned (pv, 64)
152 #pragma aligned (pf, 64)
153     for (indx = 0; indx < veclen; indx++)
154     {
155         temp    = (pv[indx - 2] * pc[0]);
156         temp   += (pv[indx - 1] * pc[1]);
157         temp   += (pv[indx] * pc[2]);
158         temp   += (pv[indx + 1] * pc[3]);
159         temp   += (pv[indx + 2] * pc[4]);
160         pf[indx] = temp;
161     }
162     return;
163 }
```

Compiler Output (inner loop):

```
#<loop> Loop body line 153, nesting depth: 1, estimated kernel iterations: 11
#<loop> unrolled 8 times
#<swps>
#<swps> 40 cycles per pipeline stage in steady state with unroll=8
#<swps> 2 pipeline stages
#<swps> 88 real ops (excluding nop)
#<swps>
#<swps> 40 cycles lower bound required by resources
#<swps> min 25 cycles required by recurrences
#<swps> min 40 cycles required by resources/recurrence
#<swps> min 35 cycles required for critical path
#<swps> 48 cycles non-loop schedule length

#<swps> register file usage:
#<swps> 'a' total 2 out of 16 [2-3]
#<swps> 'v' total 31 out of 32 [0-30]
#<swps> 'u' total 1 out of 4 [0]
#<swps>
#<freq> BB:130 => BB:130 probability = 0.91667
#<freq> BB:130 => BB:142 probability = 0.08333
```

The screenshot shows the Cadence XPG interface with a tab bar including "Problems", "Tasks", "Console", "Estimation - VisionP6\_AO\_Alpha", "XPG View", "Search", and "Vectorization Assistant". The "Vectorization Assistant" tab is active, displaying a table of analysis results.

File/Line	Function	Cycles	Instructions	Ve...	MsgId	Message
vectorizationFunctionsAuto.c:119	Filt3TapVectorAuto	0	0	y	SIMD_LOOP_VECTORIZED	Loop 3 is vectorized by 16.
vectorizationFunctionsAuto.c:153	Filt5TapVectorAuto	0	0	y	SIMD_LOOP_VECTORIZED	Loop 4 is vectorized by 16.
vectorizationFunctionsAuto.c:194	Filt3x3VectorAuto	0	0	y	SIMD_LOOP_VECTORIZED	Loop 5 is vectorized by 16.
vectorizationFunctionsAuto.c:206	Filt3x3VectorAuto	0	0	y	SIMD_LOOP_VECTORIZED	Loop 6 is vectorized by 16.

# Auto Vectorization with Hints Example 3: Floating Point 5 Tap Filter ... continued

- This example is for a vision P6 configuration **with VFPU unit**
- On the left we show 6 instructions from the 40 operations in the inner loop of the function **Fir5TapVectorAuto**
- Schedule length is 40 cycles, unroll of 8
- 40 MAC, 32 SEL, 8 LD and 8 ST
- We have a (float) MAC operation in every instruction
- Loop is **MAC bound!**

Compiler Output (inner loop):

```
{ # format N1
  ivp_la2nx8_ip    v0,u0,a2
  nop
  ivp_muln_2xf32  v2,v1,v22
}
{
# format N1
  ivp_la2nx8_ip    v6,u0,a2
  nop
  ivp_mulan_2xf32   v20,v16,v26
}
{
# format F2
  ivp_sv2nx8_ip    v14,a3,192
  ivp_la2nx8_ip    v10,u0,a2
  ivp_mulan_2xf32   v21,v3,v26
  nop
}
{
# format F1
  ivp_sv2nx8_i      v17,a3,-128
  ivp_la2nx8_ip    v13,u0,a2
  ivp_muln_2xf32  v7,v0,v22
  ivp_seln_2x32   v3,v0,v1,v30
}
{
# format F1
  ivp_sv2nx8_i      v19,a3,-64
  ivp_la2nx8_ip    v16,u0,a2
  ivp_muln_2xf32  v11,v6,v22
  ivp_seln_2x32   v5,v6,v0,v30
}
{
# format F1
  ivp_la2nx8_ip    v18,u0,a2
  nop
  ivp_muln_2xf32  v14,v10,v22
  ivp_seln_2x32   v4,v0,v1,v29
}
```

# Auto Vectorization and Optimization Summary

Compile for auto-vectorization:

- Enable Auto vectorization in Xplorer optimization options
- -O3 -LNO:simd
- Check messages in Vectorization Assistant

Keep intermediate compiler files:

- -keep (detailed compile information in bin directory, called <file>.s)

Look at operation schedules of inner loops

- Should see vector rather than scalar operations in disassembly
- Should see most slots in FLIX formats occupied
- Justify NOPs occurrence (slot conflicts, dependencies, etc.)

# Lab



## Lab 1: Auto-Vectorization

- Work with Vectorization Assistant
- Generate, locate and analyze assembly code
- Usage of keyword \_\_restrict

# Summary

In this module, you

- Used the auto-vectorizing Compiler
- Consulted the Vectorization Assistant for feedback
- Analyzed assembly code and studied the Software Pipelining Report
- You also learned about the keyword `__restrict`

Keep in mind that ...

- Auto-Vectorization is the **lowest level of effort** programming style.
- **Keyword insertion** is often necessary to get code to **vectorize**.

When achieved performance falls short of the estimate, you might want to consider the methods covered in the next **2 modules**

- **using vector types in your source code**
- **programming with intrinsics**

# Quiz

1. The Vectorization Assistant
  - a) Only vectorizes code when the Optimization level is set to 3
  - b) Is a tool that provides feedback on auto-vectorization of C-code
  - c) Only works if you use the keyword `__restrict`
2. Whether auto-vectorization succeeds or not strongly depends on
  - a) Whether your code has data dependencies
  - b) The editor that you use
3. The dot product example auto-vectorized without the keyword `__restrict`
  - a) We were lucky – it depends on your tools version
  - b) The compiler assumes that inputs and output don't alias
  - c) The output is a scalar so the compiler can safely assume that there is no input to output dependency that vector parallelism could break

Answers: 1b, 2a, 3c



# Programming with Vector Types

**Module** 7

**Revision** 1.0

**Version** 7.4

**Estimated Time:**

- For the lecture 30 minutes
- For the lab N/A

**Dependencies:**

- Module 6 – Auto-Vectorization of C-Code

# Module Objectives

In this module, you will

- Use vector types and operators to achieve vectorization
- Get familiar with vector register and data types
- Use the N-Way programming model

# Directly Working with Vectors – C Operators

- Use Vision P6 Vector Data Types
- C operators work for vector types:
  - +, -, \*, &, ^, |, <<, >>, ~, <, <=, >, >=, ==
- Pointers to vector types can be used
  - Indirection Operator \* on a vector pointer results in vector load
  - Compiler assumes that these addresses are aligned and uses normal load operations
  - Resulting code will only work with **aligned addresses**

# Example for using Vector Types: Vector ADD

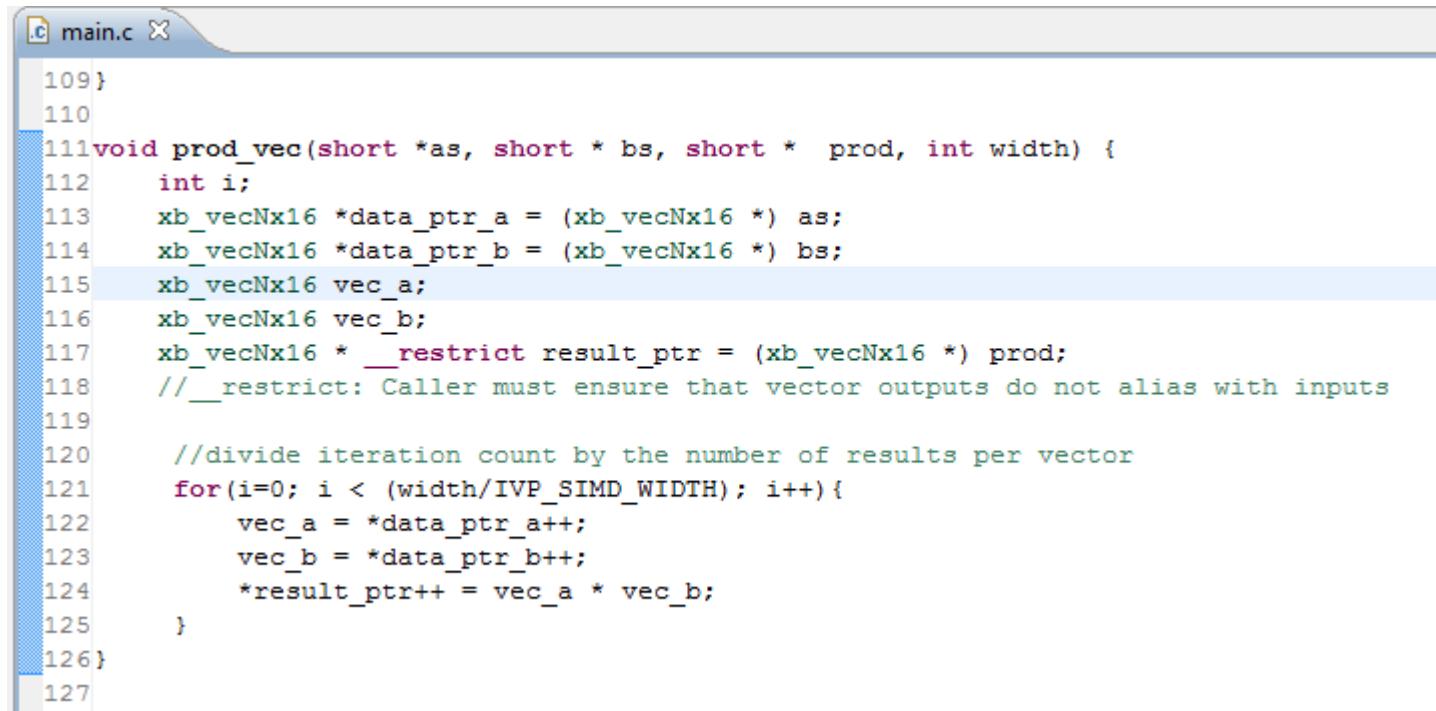
```
void vector_add (short a[], short b[], short c[], unsigned len)
{
    int i;
    int N=XCHAL_IVP SIMD_WIDTH;
    // Cast pointers to short into pointers to vectors
    xb_vecNx16 * vap = (xb_vecNx16*)a;
    xb_vecNx16 * vbp = (xb_vecNx16*)b;
    xb_vecNx16 * vcp = (xb_vecNx16*)c;
    // Change loop count to work on 32 elements at a time
    for (i = 0; i < len/N; i += 1) { // loop iterates on vectors of 32
        vcp[i] = vap[i] + vbp[i]; // store output back into 16 bit vector
    }
}
```

- Code uses pointer to a vector of type `xb_vecNx16`
- Iteration count is divided by vector length (SIMD width)
- `+` Operator is used on vectors

# Vision P6 Registers and Data Types

Register File Name	Data Type	Description	Typical Usage
VEC	xb_vecNx16	32 16-bit, signed	compute
VEC	xb_vecNx16U	32 16-bit, unsigned	compute
VEC	xb_vecN_2x32	16 32-bit, signed	compute
WVEC	xb_vec2Nx24	64 24-bit, signed	Intermediate Values
VBOOL	vbool	64 1-bit, Boolean	Vector Boolean Predicates
ALIGN	valign	512 bit data for load alignment	Byte Stream Alignment Data

# Example for using Vector Types: Vector Product



The screenshot shows a code editor window titled "main.c". The code is a C function named "prod\_vec" that performs a vector product. It uses SIMD types from the "xb" namespace. The function takes four parameters: pointers to short arrays "as" and "bs", a pointer to a short array "prod" for the result, and an integer "width". It iterates over "width/IVP SIMD\_WIDTH" elements, loading vectors "vec\_a" and "vec\_b" from "data\_ptr\_a" and "data\_ptr\_b" respectively, and calculating their dot product into "result\_ptr". The keyword `__restrict` is used to ensure that the result pointer does not alias with the input pointers.

```
109}
110
111void prod_vec(short *as, short * bs, short * prod, int width) {
112    int i;
113    xb_vecNx16 *data_ptr_a = (xb_vecNx16 *) as;
114    xb_vecNx16 *data_ptr_b = (xb_vecNx16 *) bs;
115    xb_vecNx16 vec_a;
116    xb_vecNx16 vec_b;
117    xb_vecNx16 * __restrict result_ptr = (xb_vecNx16 *) prod;
118    //__restrict: Caller must ensure that vector outputs do not alias with inputs
119
120    //divide iteration count by the number of results per vector
121    for(i=0; i < (width/IVP SIMD_WIDTH); i++) {
122        vec_a = *data_ptr_a++;
123        vec_b = *data_ptr_b++;
124        *result_ptr++ = vec_a * vec_b;
125    }
126}
127
```

- This is how the code Vector Product (see Auto-Vectorization Module) looks with vector types
- The keyword `__restrict` is still important for better code scheduling (more on this later)

# Vector-Length Independent Programming

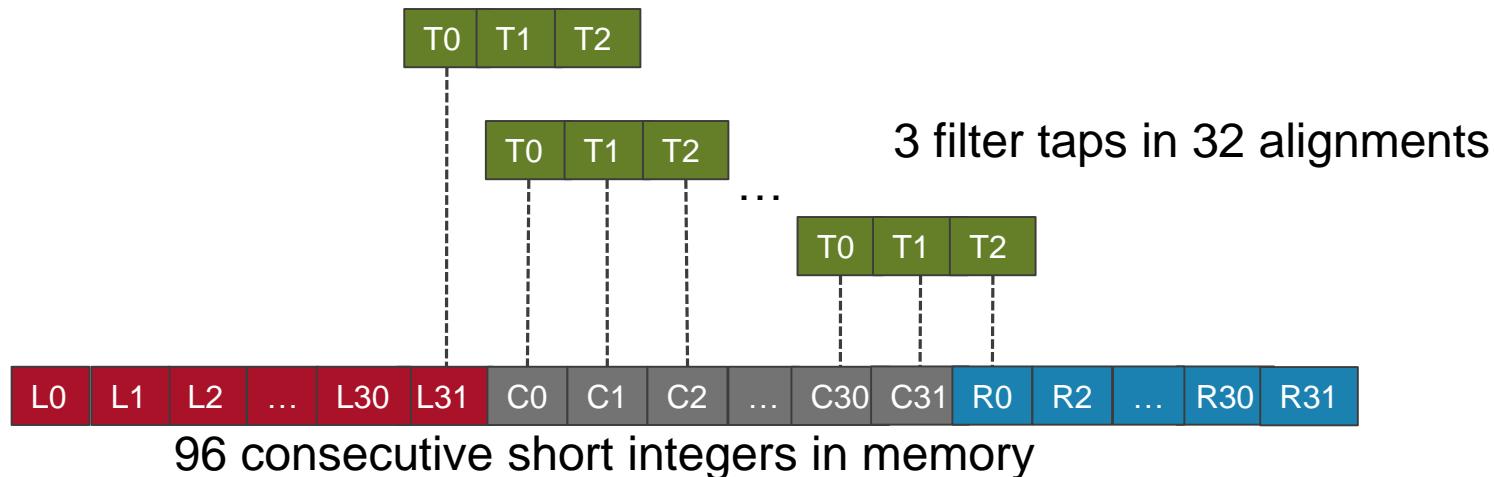
- For Vision P6 the SIMD width is **32** (16-bit elements)
- Future machines in the IVP processors family might have different SIMD widths
- We provide a **#define** constant to specify the SIMD width:  
**XCHAL\_IVP SIMD WIDTH**
- By avoiding to use the literal value 32 and using this constant for the vector length you can ensure compatibility with future versions of the Vision Processor

```
xb_vecNx16 vin, vout = 0;
for (i=0;i<N/XCHAL_IVP SIMD WIDTH;i++) {
    vout += vin[i] * vin[i];
}
*out_p = IVP_RADDNX16(vout);
```

# Example: Use of Multiply Operator in 3-tap Filter

- The following example demonstrates the use of the Multiply-Pack operations for a 1 dimensional 3-tap FIR filter
- This example also uses the a **select operation** which we will cover in detail later in this training
- FIR (Finite Impulse Response) filters are commonly used in image processing
- 2 dimensional FIR filters are more common in image processing, but those are often implemented as a sequence of 1D FIR filters as this sometimes reduces
  - The number of needed multiply operations (separable filters)
  - The number of operations per inner loop iterations, which often allows for a better loop schedule

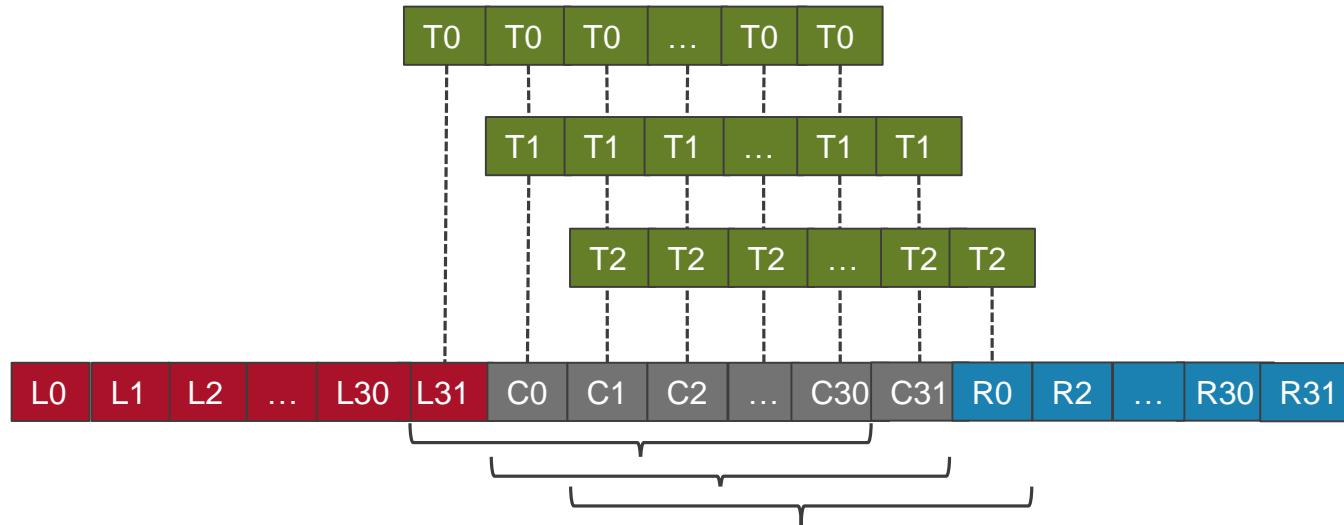
# Example: Use of Multiply Operator in 3-tap Filter



- Compute filter output by aligning and multiplying the filter mask (T0, T1, T2) with the input data
- Each of the 32 alignments will produce one output value

$$\begin{aligned} \text{Out}[0] &= T0 * L31 + T1 * C0 + T2 * C1 \\ \text{Out}[1] &= T0 * C0 + T1 * C1 + T2 * C2 \\ &\dots \\ \text{Out}[31] &= T0 * C30 + T1 * C31 + T2 * R0 \end{aligned}$$

# Example: Use of Multiply Operator in 3-tap Filter



- Replicate each filter tap 32 times
- Multiply with corresponding input vector
- The input data is needed in multiple alignments
- Select Operations can be used to create input vectors in those alignments

$$\begin{aligned} \text{Out}[0] &= T_0 * L_{31} + T_1 * C_0 + T_2 * C_1 \\ \text{Out}[1] &= T_0 * C_0 + T_1 * C_1 + T_2 * C_2 \\ &\dots \\ \text{Out}[31] &= T_0 * C_{30} + T_1 * C_{31} + T_2 * R_0 \end{aligned}$$

# Example: Use of Multiply Operator in 3-tap Filter

The pseudo C code for the filter looks like this:

```
//L, C, and R are vectors holding the input data  
//T0, T1, T2 are vectors with replicated filter tap values  
temp = IVP_SELNX16I(C, L, IVP_SELI_ROTATE_LEFT_1) // slide window left by 1 pixel  
out = T0*temp; // tap T0  
out += T1*C // center tap T1  
temp = IVP_SELNX16I(R, C, IVP_SELI_ROTATE_RIGHT_1) // slide window right by 1 pixel  
out += T2*temp; // tap T2
```

- This code produces 1 vector output of 32 elements \* 16 bit values in 3 cycles
- multiply operator \* will result in a IVP\_MULNX16PACKL operation, which
  - Takes 2 VEC register 16-bit inputs and creates 1 VEC register 16-bit output
  - Only the lower 16 bits of the multiply output are written to the output register
- This is appropriate when the products will not overflow 16 bit accumulation

# Summary

In this module, you

- Used vector types and operators to achieve vectorization
- Became familiar with vector register and data types
- Used the N-Way programming model

Programming with Vectors and Operators is a **mid level of effort** style.

You have a fair **amount of code to change** to move from integer to vector data types. You also have to explicitly find the parallelism yourself instead of the compiler to find it for you.

Occasionally you might come across an algorithm where operations that you want to target can not be **easily inferred** by the compiler.

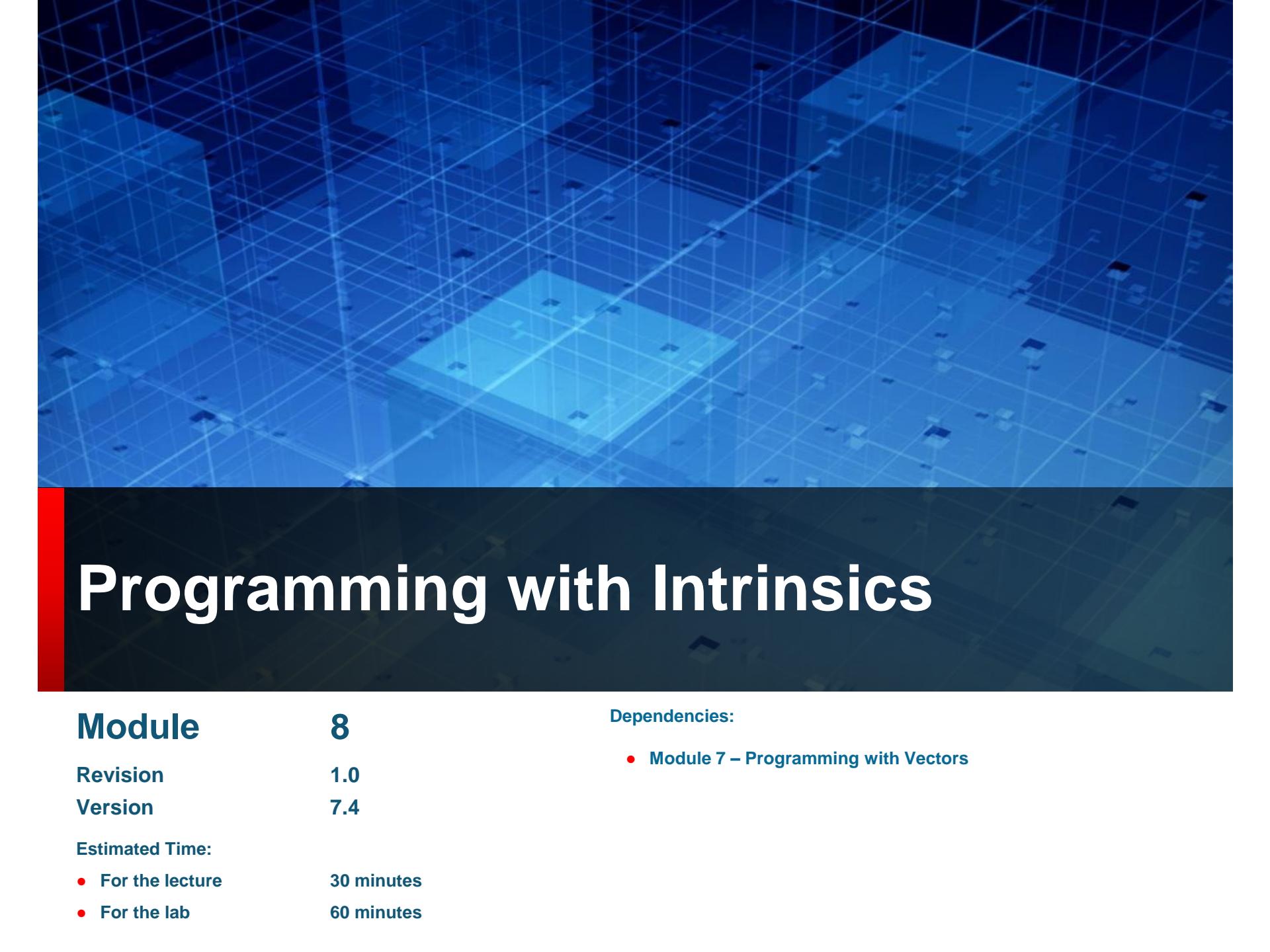
In those cases you will have to program with **intrinsics**.

This programming style is covered in the next module.

# Quiz

1. The vector type xb\_vecNx16
  - a) describes a vector with N elements that are each 16-bit signed integers.
  - b) describes a vector with N elements that are each 16-bit unsigned integers.
  - c) describes a vector with 16 elements that are each N-bit signed integers.
2. The expression &v+1 (where v is declared as xb\_vecNx16 v;)
  - a) is invalid.
  - b) is the address of the element 1 in the vector 0 (as opposed to element 0 which is the first one).
  - c) Is the address of a vector that is offset 64 bytes (32 X 16bit) from the element 0 of vector v.
3. The N-way programming model is encouraged
  - a) because N is faster to type than 64, 32, 16.
  - b) because it makes your code forward compatible with future versions of Vision that might have a different vector length.
  - c) Because it helps the compiler to produce faster performing code.

Answers: 1a, 2c, 3b



# Programming with Intrinsics

**Module** 8

**Revision** 1.0

**Version** 7.4

**Estimated Time:**

- For the lecture 30 minutes
- For the lab 60 minutes

**Dependencies:**

- [Module 7 – Programming with Vectors](#)

# Module Objectives

In this module, you will

- Learn about the difference between programming with intrinsics and in assembly
- Use intrinsics in your C code to target specific operations
- Apply operation intrinsics to achieve vectorization where neither auto-vectorization nor programming with vectors could

# Vectorization Using C Intrinsics

**Intrinsics** are C language constructs that map to machine operations

- Intrinsics are **not assembly** language
- Syntax is like C/C++ functions, arguments are C or vector data types
- Compiler still handles type checking and data movement
- Compiler still schedules operations into VLIW instructions
- Lowest level style of programming for maximal flexibility
- Can also give access to a sequence of operations
- Special intrinsics called protos can abstract from data types

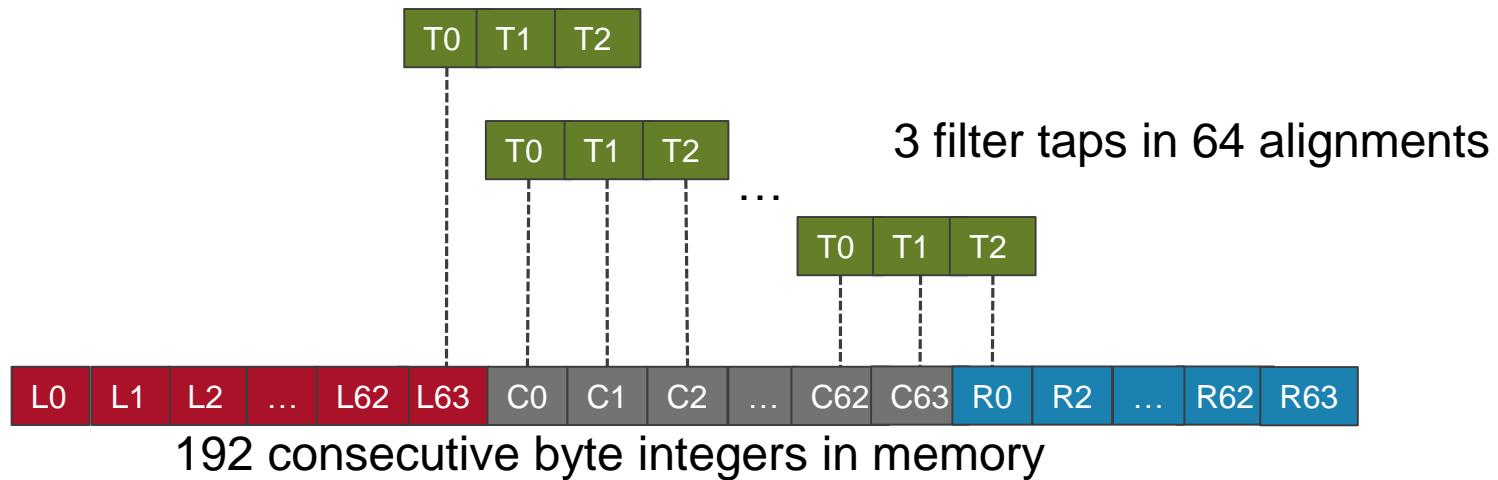
# Why use Intrinsics?

- To target specific instructions that allow for better instruction scheduling
  - A given algorithm might require a pair of operations that don't have a common format. User might be able to find similar operations that share a common format and allow for a better instruction schedule
- To facilitate use of operations that the compiler can't infer (e.g., SELECT operations for a Look-Up-Table)
- To use an interface to common set of operations:
  - In order to perform a divide operation a sequence of Vision Operations are needed
  - This sequence is combined into one divide proto

```
proto IVP_DIV16U { out xb_int16U quo, out xb_int16U rem, in xb_int16U dvdnd, in xb_int16U dvsr }
    IVP_DIVNX16U_4STEP0 tmp_quo, tmp_pr, dvdnd, dvsr, 0;
    IVP_DIVNX16U_4STEP tmp_quo, tmp_pr, dvsr, 0;
    IVP_DIVNX16U_4STEP tmp_quo, tmp_pr, dvsr, 0;
    IVP_DIVNX16U_4STEPN tmp_quo, tmp_pr, dvsr, 0;
    IVP_SELSNX16 quo, tmp_quo, 0;
    IVP_SELSNX16 rem, tmp_pr, 0;
```

}

## Example: Use of Multiply and Select Operations in 3-tap Filter



- We will re-visit the 3-tap filter example that we saw earlier
- Now we assume that input data is 8 bit and filter coefficients are 16 bit
- We also assume 8 bit output values
- We will use following operations
  - **IVP\_SEL2NX8I – 64-element \* 8 bit select from immediate**
  - **IVP\_MULUS2N8XR16 - 64-element \* 8 bit vector by 16 bit scalar multiply**
  - **IVP\_MULUSPA2N8XR16 - 64-element \* 8 bit vector by 16 bit scalar paired multiply**

## Example: Use of Multiply and Select Operations in 3-tap Filter

The pseudo C code for the filter looks like this:

```
T2T1 = (T2 << 16) | T1; //Combine T2T1 for using paired MAC(IVP_MULUSPA2NXR16)
//L, C, and R are vectors holding input data, T0, T2T1 are filter tap values
temp = IVP_SEL2NX8I(C, L, IVP_SELI_8B_ROTATE_LEFT_1);
outW = IVP_MULUS2N8XR16(temp, T0); //no accumulation on first multiply
temp = IVP_SEL2NX8I(R, C, IVP_SELI_8B_ROTATE_RIGHT_1);
//paired MAC: outW += (temp * T2 + C * T1)
IVP_MULUSPA2N8XR16(outW, temp, C, T2T1);
//pack outputs from WVEC into a VEC with shift amount in AR register
out = IVP_PACKVRU2NX24(outW, sh);
```

- Code produces 1 vector output of **64** elements \* **8** bit values in **2 cycles**
- Intermediate results stored in a WVEC register **64 elements \* 24 bits**
- Pack operation moves the values in the WVEC register **outW** into a VEC register **out**
- The AR register **sh** applies a shift amount during the pack operation

# Summary

In this module, you

- Saw the difference between programming with intrinsics and in assembly
- Used intrinsics in your C code to target specific operations
- Applied operation intrinsics to achieve vectorization with higher throughput than previous methods.

Programming with intrinsics is a **high level of effort** style. You have to

- move from integer to vector data types
- find the parallelism
- Identify the needed operation intrinsics to insert into your code

Programming with intrinsics is appropriate for inner loops where **vectorization** or an efficient **instruction schedule** can not be achieved with the methods in the previous module.

Now that you are versed in these 3 programming styles we will teach you how to assess the performance of your code in “Module 9 – Assessing Performance”



## Lab 2: Intrinsics

- Use operation intrinsics in your code
- Compare performance of auto-vectorized code and code with intrinsics.
- Usage of keyword `__restrict`

# Quiz

## 1. Intrinsic map

- a) always to one operation
- b) map to one or a sequence of multiple operations

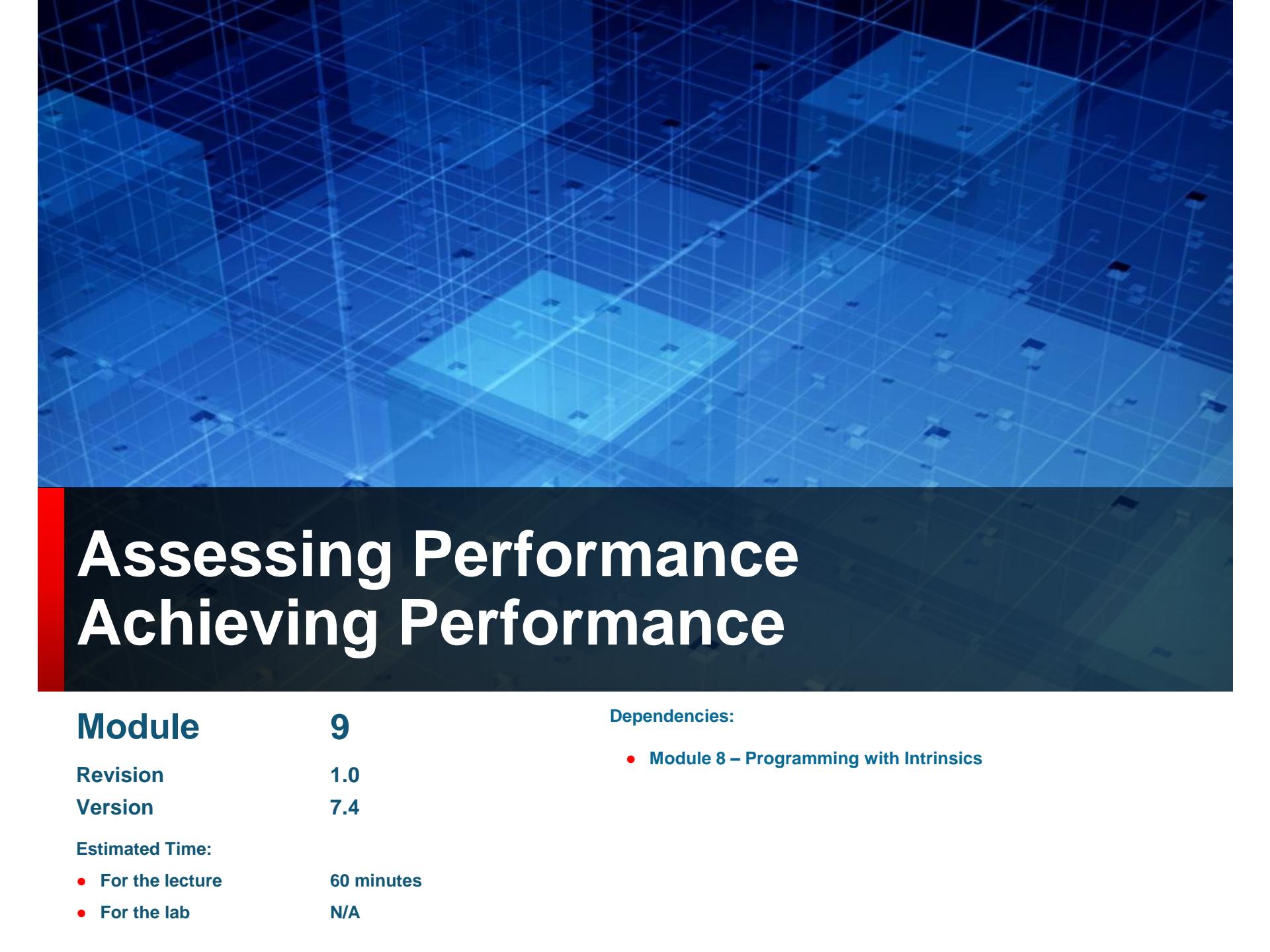
## 2. When programming with vector types - the keyword \_\_restrict in front of output pointers is

- a) not necessary when you program with intrinsics since you already use vector types explicitly
- b) still necessary to ensure that your code can software pipeline
- c) illegal in code using vector types

## 3. Programming with vector operation intrinsics

- a) is just as much work as programming in assembly language.
- b) is less work than programming in assembly since you don't have to manually schedule registers and assign operation to operations slots and come up with an operation schedule.
- c) is discouraged since it does not make your code forward compatible

Answers: 1b, 2b, 3b



# Assessing Performance Achieving Performance

**Module** 9

**Revision** 1.0

**Version** 7.4

**Estimated Time:**

- For the lecture 60 minutes
- For the lab N/A

**Dependencies:**

- Module 8 – Programming with Intrinsics

# Module Objectives

In this module, you will learn how to assess performance of your code. You will

- Learn about software pipelining (SWP)
- Read the SWP report in the assembly code
- See how the compiler treats code and inner versus outer loops
- Learn to use the instruction set simulator (ISS) with and without memory modelling
- Interpret profiler results

# Alpha Blend Example

Alpha Blending two images means to calculate a weighted average based on a blending factor:

$$out(x, y) = \alpha in1(x, y) + (1 - \alpha)in2(x, y), \quad 0 < \alpha < 1$$

The optimized code implements this equation with a MULP and a PACK operation to get the results back into a vector register.

```
void xvAlphaBlend(uint8_t *pIn1, uint8_t *pIn2, uint8_t *pOut, int16_t alpha)
{
    int32_t i, alpha1Malpha;
    xb_vec2Nx8U * __restrict pvecIn1 = (xb_vec2Nx8U *) pIn1;
    xb_vec2Nx8U * __restrict pvecIn2 = (xb_vec2Nx8U *) pIn2;
    xb_vec2Nx8U * __restrict pvecOut = (xb_vec2Nx8U *) pOut;
    xb_vec2Nx8U vecIn1, vecIn2, vecOut;
    xb_vec2Nx24 wvec0;
    alpha1Malpha = ((0x3fff - alpha) << 16) + alpha;

    for (i = 0; i < (WIDTH * HEIGHT) / 2 / XCHAL_IVPN SIMD_WIDTH; i++)
    {
        vecIn1     = *pvecIn1++;
        vecIn2     = *pvecIn2++;
        wvec0     = IVP_MULUSP2N8XR16(vecIn2, vecIn1, alpha1Malpha);
        vecOut     = IVP_PACKVRU2NX24(wvec0, 14);
        *pvecOut++ = vecOut;
    }
}
```

# Assembly Code of Alpha Blend Example: How to read the software pipelining report?

```
#<loop> Loop body line 53, nesting depth: 1, kernel iterations: 61
#<loop> unrolled 2 times
#<swps>
#<swps> 4 cycles per pipeline stage in steady state with unroll=2
#<swps> 4 pipeline stages
#<swps> 10 real ops (excluding nop)
#<swps>
#<swps>          4 cycles lower bound required by resources
#<swps>          min 3 cycles required by recurrences
#<swps>          min 4 cycles required by resources/recurrence
#<swps>          min 10 cycles required for critical path
#<swps>          14 cycles non-loop schedule length

#<swps> register file usage:
#<swps>   'a' total 5 out of 16 [2-4,6,11]
#<swps>   'v' total 6 out of 32 [0-5]
#<swps>   'wv' total 2 out of 4 [0-1]
#<swps>
#<freq> BB:69 => BB:69 probability = 0.98438
#<freq> BB:69 => BB:78 probability = 0.01563
    .frequency 1.000 63.492
{
  # format F0
  ivp_lv2nx8_ip    v0,a2,128      # [0*III+0]  id:44
  ivp_packvru2nx24  v2,wv0,a6      # [2*III+0]
  ivp_mulusp2n8xr16 wv0,v1,v0,a11 # [1*III+0]
  nop               #
}
  # format N2
  ivp_sv2nx8_i     v5,a4,-64      # [3*III+1]  id:46
  ivp_lv2nx8_ip    v1,a3,128      # [0*III+1]  id:45
}
  # format F0
  ivp_lv2nx8_i     v3,a2,-64      # [0*III+2]  id:44
  ivp_packvru2nx24  v6,wv1,a6      # [2*III+2]
  ivp_mulusp2n8xr16 wv1,v4,v3,a11 # [1*III+2]
  nop               #
}
  # format N2
  ivp_sv2nx8_ip    v2,a4,128      # [2*III+3]  id:46
  ivp_lv2nx8_i     v4,a3,-64      # [0*III+3]  id:45
}
```

- Schedule length 4 cycles
- Unroll = 2 – 2 iterations of C loop in one assembly loop
- 4 pipeline stages – Operations from 4 loop iteration execute in parallel (e.g. [0\*II+0] corresponds to stage0 ,[2\*II+0] corresponds to stage2)
- 10 real operations in 4 cycles, or 2.5 ops per cycle
- HW resource limit is 4 cycles as 4 Load, 2 Store and 2 Pack ops can only go in slot0 and slot1.
- 8 ops in 2 slots require at least 4 cycles.
- This loop schedules at the HW resource limit of 4 cycles.

# Understanding software pipelining

If we removed the `__restrict` keyword we would see the following schedule:

Cycle	Op
0	LOAD/LOAD
1	Bubble
2	Bubble
3	MULP
4	Bubble
5	PACK
6	Bubble
7	Bubble
8	ST

If we unroll by a factor of 2

Cycle	Op
0	LOAD/LOAD
1	LOAD/LOAD
2	Bubble
3	MULP
4	MULP
5	PACK
6	PACK
7	Bubble
8	ST
9	ST

# Understanding software pipelining

The color coding suggest the operations in this loop can go into different stages.

The software pipelined schedule now looks at follows:

LOAD	PACK	MULP	NOP
ST	LOAD		
LOAD	PACK	MULP	NOP
ST	LOAD		

Since this loop has 4 pipeline stages it takes 4 iterations of this loop to load, process and store first result.

Since the compiled code is unrolled by 2, work for 2 iterations of the C loop is handled in one assembly loop.

In every stage 2 result vectors (unroll 2) are produced.

# Prologue, Epilogue and Steady State Loop

Stages	Code density	loop state	Called how often?
0	lowest	prologue	1
0+1	low		
0+1+2	higher		
0+1+2+3	max	steady state	?
1+2+3	higher		
2+3	low	epilogue	1
3	lowest		

- During loop execution data flows through the processing pipeline from one stage to the next.
- In **steady state**, every stage of the loop feeds data to the consecutive stage which will consume it during the next iteration
- Before steady state loop can execute Stages 0,1,2 and 3 (in parallel), it needs data produced by stages 0,1,2 run earlier.
- In turn that code needs to be preceded by code running stages 0 and 1 and so on.
- The code that needs to run prior to the steady state loop is called **prologue** the code after is called **epilogue**

# What is the iteration count for the steady state loop?

Stages	Code density	loop state	Called how often?
0	lowest	prologue	1
0+1	low		
0+1+2	higher		
0+1+2+3	max	steady state	N-3
1+2+3	higher	epilogue	1
2+3	low		
3	lowest		

- Now we need to figure out how frequently the code in the steady state loop needs to be executed.
- If we add up the number of stages in the prologue and epilogue we see that is accounts for exactly 3 iterations of the 4 stage loop.
- This means that steady state loop of a 4-stage loop only needs to execute **N-3** times
  - For a 3-stage loop the number is **N-2**
  - For a 2-stage loop the number is **N-1**

# Cycle Profile the Alpha Blend Example

```
void xvAlphaBlend(uint8_t *pIn1, uint8_t *pIn2, uint8_t *pOut, int16_t alpha)
4 {
    int32_t i, alpha1Malpha;
    xb_vec2Nx8U * __restrict pvecIn1 = (xb_vec2Nx8U *) pIn1;
    xb_vec2Nx8U * __restrict pvecIn2 = (xb_vec2Nx8U *) pIn2;
    xb_vec2Nx8U * __restrict pvecOut = (xb_vec2Nx8U *) pOut;
    xb_vec2Nx8U vecIn1, vecIn2, vecOut;
    xb_vec2Nx24 wvec0;
    alpha1Malpha = ((0xffff - alpha) << 16) + alpha;

4 for (i = 0; i < (WIDTH * HEIGHT) / 2 / XCHAL_IVPN SIMD_WIDTH; i++)
{
    vecIn1      = *pvecIn1++;
    vecIn2      = *pvecIn2++;
    wvec0       = IVP_MULUSP2N8XR16(vecIn2, vecIn1, alpha1Malpha);
    vecOut      = IVP_PACKVRU2NX24(wvec0, 14);
    *pvecOut++ = vecOut;
}
1 }
```

- 281 cycles spent in xvAlphaBlend
- 276 are spent in the inner loop
- 5 cycles are spent in the outer loop/function call

What is the loop iteration count?

WIDTH = 256, HEIGHT=32,  
2\*SIMD\_WIDTH= 64

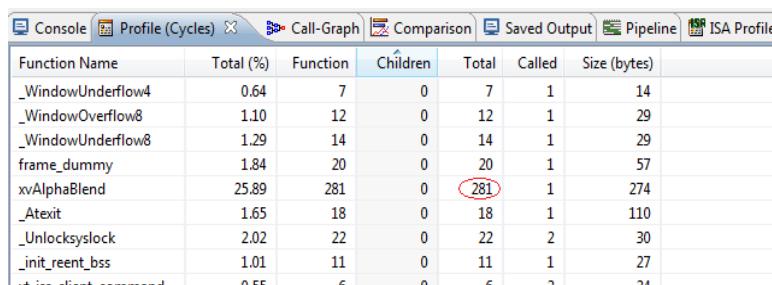
C Loop iteration count: **128**

Assembly loop iteration count  
**(unroll=2): 64**

Nominal cycle count =

schedule length \* iteration count =  
**4 \* 64 = 256**

Measured inner loop cycle count  
(profiler): **276 (20 cycles overhead)**



Function Name	Total (%)	Function	Children	Total	Called	Size (bytes)
_WindowUnderflow4	0.64		7	7	1	14
_WindowOverflow8	1.10		12	12	1	29
_WindowUnderflow8	1.29		14	14	1	29
frame_dummy	1.84		20	20	1	57
xvAlphaBlend	25.89		281	281	1	274
_Atexit	1.65		18	18	1	110
_Unlocksyslock	2.02		22	22	2	30
_init_reent_bss	1.01		11	11	1	27
__stack_chk_fail	0.00		0	0	0	0

# Instruction Count Profile (Disassembly View)

- The iteration count for the assembly loop is 64
- On slide 4 we saw that this code has 4 pipeline stages so we expect the iteration count of the steady state loop to be  $N-3 = 64-3 = 61$
- We can confirm that count when looking at the **instruction count** profiler output

Count	Address	Instruction
1	60002193	ivp_packvru2nx24 v5, wv0, a6 ivp_mulusp2n8xr16 wv0, v6, v7, a11 nop } { { nop nop nop }
1	6000219b	nop
1	6000219e	nop.n
1	600021a0	{ { loopgtz a7, 600021e0 <xvAlphaBlend+0xc0> ivp_lv2nx8_i v4, a3, -64 ivp_mulusp2n8xr16 wv1, v2, v4, a11 nop }
61	600021b0	{ { ivp_lv2nx8_ip v0, a2, 128 ivp_packvru2nx24 v2, wv0, a6 ivp_mulusp2n8xr16 wv0, v1, v0, a11 nop }
61	600021c0	{ { ivp_sv2nx8_i v5, a4, -64 ivp_lv2nx8_ip v1, a3, 128 }
61	600021c8	{ { ivp_lv2nx8_i v3, a2, -64 ivp_packvru2nx24 v5, wv1, a6 ivp_mulusp2n8xr16 wv1, v4, v3, a11 nop }
61	600021d8	{ { ivp_sv2nx8_ip v2, a4, 128 ivp_lv2nx8_i v4, a3, -64 }
1	600021e0	{ { nop ivp_packvru2nx24 v16, wv0, a6 ivp_mulusp2n8xr16 wv3, v1, v0, a11 nop }
1	600021f0	{ { nop ivp_packvru2nx24 v15, wv1, a6 ivp_mulusp2n8xr16 wv2, v4, v3, a11 nop }
1	60002200	1111 0000 0000 0000 0000 0000 0000 0000

# Cycle Count Profile (Disassembly View)

- When we look at the cycle count profile we see that it does not exactly match the instruction count.
- The first instructions shows 64 cycles, the next 3 instructions show 61 cycles (for executing 61 instructions).
- In other words for the first instructions we see 3 overhead cycles
- For the other instructions we see 2 overhead cycles
- These cycles are overheads incurred by entering and exiting the zero overhead loop.

Count	Address	Instruction
		ivp_mulusp2n8xr16 wv0, v6, v7, a11 nop }
1	60002193	{ { nop nop nop } }
1	6000219b	nop
1	6000219e	nop.n
1	600021a0	{ { loopgtz a7, 600021e0 <xvAlphaBlend+0xc0> ivp_lv2nx8_i v4, a3, -64 ivp_mulusp2n8xr16 wv1, v2, v4, a11 nop }
64	600021b0	{ { ivp_lv2nx8_ip v0, a2, 128 ivp_packvru2nx24 v2, wv0, a6 ivp_mulusp2n8xr16 wv0, v1, v0, a11 nop }
63	600021c0	{ { ivp_sv2nx8_i v5, a4, -64 ivp_lv2nx8_ip v1, a3, 128 }
63	600021c8	{ { ivp_lv2nx8_i v3, a2, -64 ivp_packvru2nx24 v5, wv1, a6 ivp_mulusp2n8xr16 wv1, v4, v3, a11 nop }
63	600021d8	{ { ivp_sv2nx8_ip v2, a4, 128 ivp_lv2nx8_i v4, a3, -64 }
1	600021e0	{ { nop ivp_packvru2nx24 v16, wv0, a6 ivp_mulusp2n8xr16 wv3, v1, v0, a11 nop }
2	600021f0	{ { nop ivp_packvru2nx24 v15, wv1, a6 ivp_mulusp2n8xr16 wv2, v4, v3, a11 nop }
1	60002200	{ { ivp_sv2nx8_i v5, a4, -64 ivp_packvru2nx24 v14, wv2, a6 }

# Overall Assessment of Alpha Blend Performance

Nominal Cycle count:  $4 * 64 = \mathbf{256}$  cycles

Measured cycle count for function xvAlphaBlend **281**

Overhead =  $(281 - 256) / 256 \sim 9\%$

## Summary

- Code only has one (inner) loop that is running with a high iteration count (64)
- Although there are noticeable overheads before and after the steady state loop and for the function call, they have a relative low cost due the large numbers of cycles spent in the steady state loop.

# SWP in inner and outer loops

It is important to understand that the compiler only applies software pipelining to inner loops.

## Inner Loops

- are zero overhead loops (no cycles are spent on loop counter incrementing and checking and code branching).
- Will be software pipelined by compiler if directed (keyword `__restrict`)

Outer loops are neither zero overhead and nor software pipelined.

For that reason it is best to put as few operations in an outer loop as you can.

If there is work to do in an outer loop consider

- if results from the inner loop can be buffered and
- final processing be done in a subsequent inner loop.

# Example: Gauss3x3 Filter

In the following slides we look at a Gaussian 3x3 filter:

1	2	1
2	4	2
1	2	1

- A 3x3 filter mask (coefficients shown to the right) is overlaid at each pixel location
- The code below implements this filter with 9 multiplications for every output pixel
- code for this filter is shown in the following slide
- 2 nested for-loops iterate over the image height and image width
- In the following slides we will show the performance numbers for different combinations of image width and height

# C-Code for Gauss3x3 Filter

```
void gauss3x3(short *p_img, short* p_img_out, int img_width, int img_stride, int img_height){  
/* This function implements a 2D gaussian filter with a 3x3 filter mask below.  
* The actual filter coefficients can be modified under user control  
*  
*  1 2 1  
*  2 4 2  
*  1 2 1  
*****  
int i,j;      //i rows, j cols  
short *data; //generic pointer to input data  
short coefs[9] = COEFS_3X3; //holds coeffs for filter mask  
short * __restrict cur_row = p_img;  
short (* __restrict cur_out) = p_img_out;  
#pragma aligned(cur_row,64)  
#pragma aligned(cur_out,64)  
  
for(i=0; i < img_height; i++) {  
    short * prev_data = cur_row-img_stride;  
    short * next_data = cur_row+img_stride;  
    data = cur_row;  
    for(j=0; j < img_width; j++) {  
        cur_out[j] = coefs[0]*prev_data[j-1] + coefs[1]*prev_data[j] + coefs[2]*prev_data[j+1];  
        cur_out[j] += coefs[3]*data[j-1]      + coefs[4]*data[j]      + coefs[5]*data[j+1];  
        cur_out[j] += coefs[6]*next_data[j-1] + coefs[7]*next_data[j] + coefs[8]*next_data[j+1];  
    }  
    cur_row += img_stride;  
    cur_out += img_width;  
}  
}
```

# SWP Report for Gauss3x3 filter

```
#<loop> Loop body line 138, nesting depth: 2, estimated kernel iterations: 49
#<loop> unrolled 2 times
#<swps>
#<swps> 18 cycles per pipeline stage in steady state with unroll=2
#<swps> 2 pipeline stages
#<swps> 42 real ops (excluding nop)
#<swps>
#<swps>          18 cycles lower bound required by resources
#<swps>          min 16 cycles required by recurrences
#<swps>          min 18 cycles required by resources/recurrence
#<swps>          min 26 cycles required for critical path
#<swps>          34 cycles non-loop schedule length

#<swps>  register file usage:
#<swps>    'a' total 4 out of 16 [2-5]
#<swps>    'v' total 32 out of 32 [0-31]
#<swps>    'u' total 3 out of 4 [0-2]
#<swps>
```

- This code auto-vectorizes. By inspecting the assembly code which we (omit for brevity) we can see that the code produces an output vector 32 (16-bit) pixels.
- The code is unrolled by 2, which means that one loop iteration produces 2 (horizontally adjacent) output vectors in 18 cycles
- The code issues 18 MUL operations in 18 cycles and therefore reaches that MUL HW resource limits (we only have one MUL slot)
- There are other MUL operations that use WVEC register that could issues significantly more MUL operations per cycle but this is the fastest performance for MUL ops that use VEC register inputs and outputs

# Profiling the Gauss3x3 Code

We will now run the code with an image size of 10 K pixels.

- Nominally this should take  $18 * 10 \text{ K} / (2 * 32) = 2,980$  cycles
- The 18 cycles per 64( $=2*32$ ) pixels come from the SWP report in the previous slide

Let's look at the performance results for different image widths and heights

Width	Output Height	Cycles	% of Nominal
10*1024	1	2981	100%
1024	10	3,557	119%
256	40	5,477	184%
64	160	13,477	452%

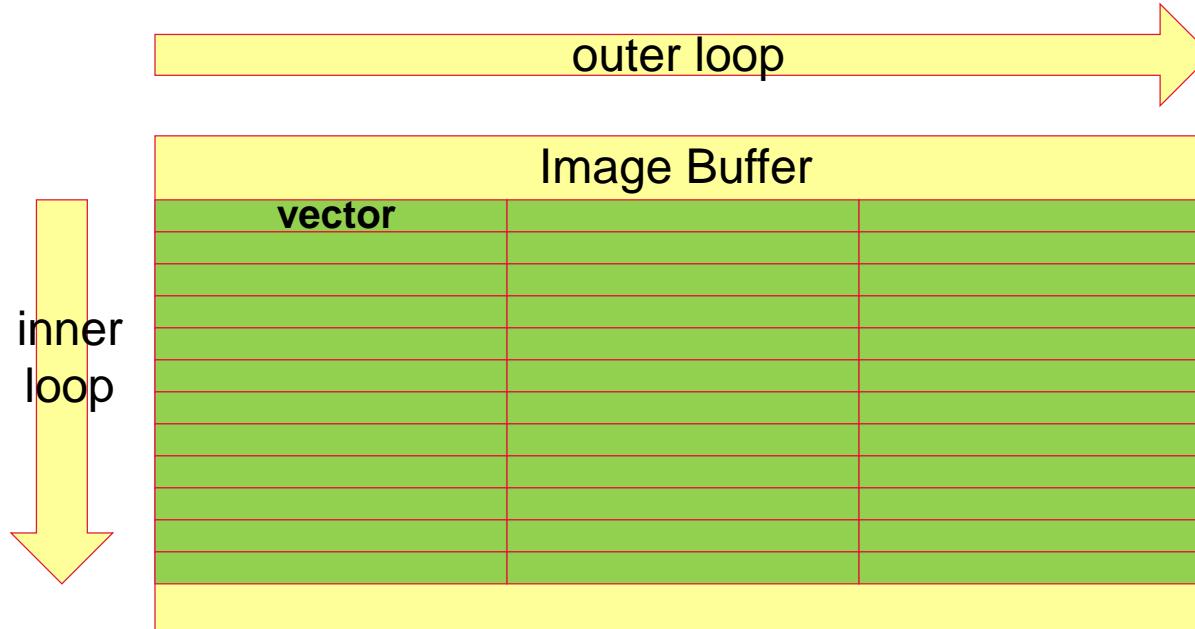
# Explanation of Gauss3x3 Profiling Results

- All cycles correspond to computing 10 K output pixels.
- The first row shows almost exactly the expected cycle counts
- As we move down in the table rows the outer loop (vertical) counts are getting higher and the inner (horizontal) loop counts are getting lower.
- For the last row the steady state inner loop is not even executed a single time
  - just prologue and epilogue code is executed

Width	Output Height	Cycles	% of Nominal
10*1024	1	2981	100%
1024	10	3,557	119%
256	40	5,477	184%
64	160	13,477	452%

# Conclusions from Gauss3x3 Profiling Results

- In order to achieve good overall performance, loop iteration counts must be high
- Image buffer widths often too small to allow large horizontal iteration counts since our vectors are very wide (64 or 32 pixels).
- In order to allow **large iteration counts** it is often best to make the vertical loop the inner loop (next output vector is below the current vector not to the right of it).

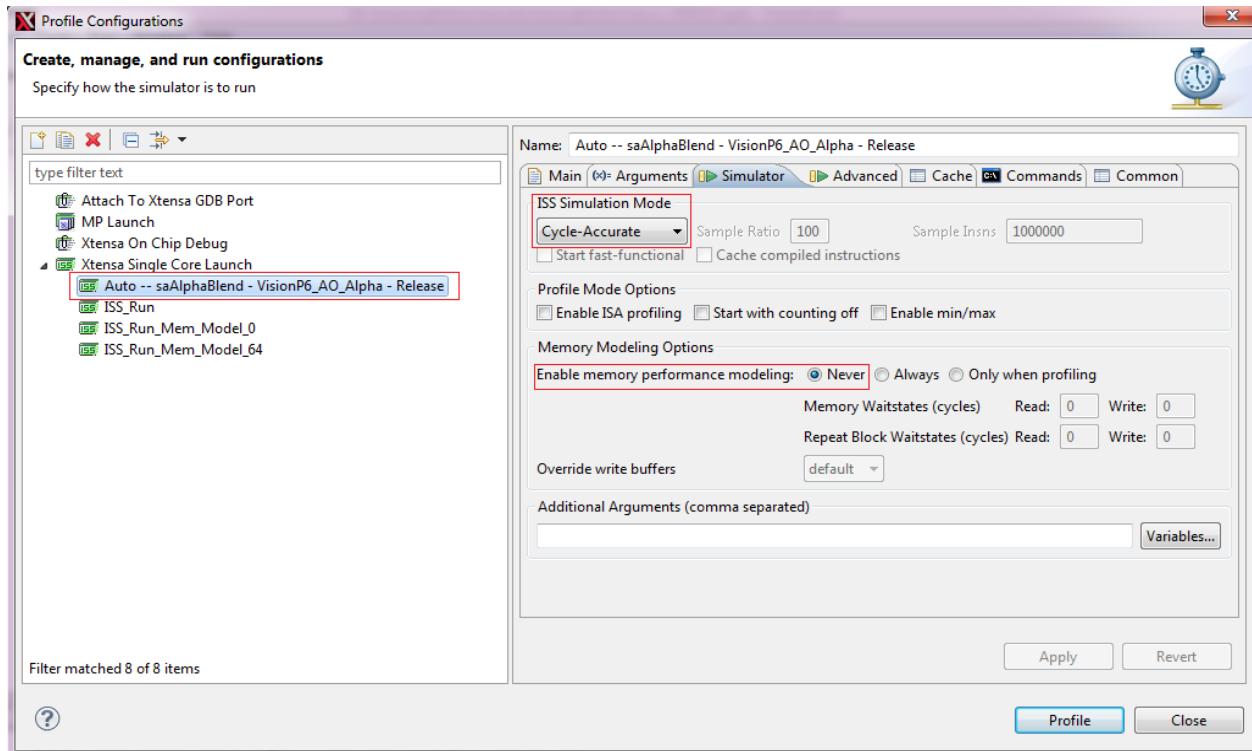


# Using the Profiler without Memory Modelling

- At early stages of code development you might be more concerned with vectorizing and measuring performance of inner loops
- You may not to worry about data flow, placement of data and buffers
- You could run the profiler **without memory modelling**.
- In this case the Instruction Set Simulator (ISS) will make the assumption that memory accesses have no contention and 0 cycle latency.
- This might allow you to get a rough idea about performance quickly, yet it might yields unrealistically optimistic results, when using gather/scatter operations.
- Dual Load/Store conflicts are not modelled either

# Profiling without Memory Modelling

- Below you see how to disable Memory Modelling in your Profile Configuration
- ISS Simulation Mode to Fast-Functional or Program-Directed can accelerate simulation speed, but can easily create confusion.
- If you are new to using the Profiler please set the ISS Simulation mode to **Cycle accurate** when profiling code

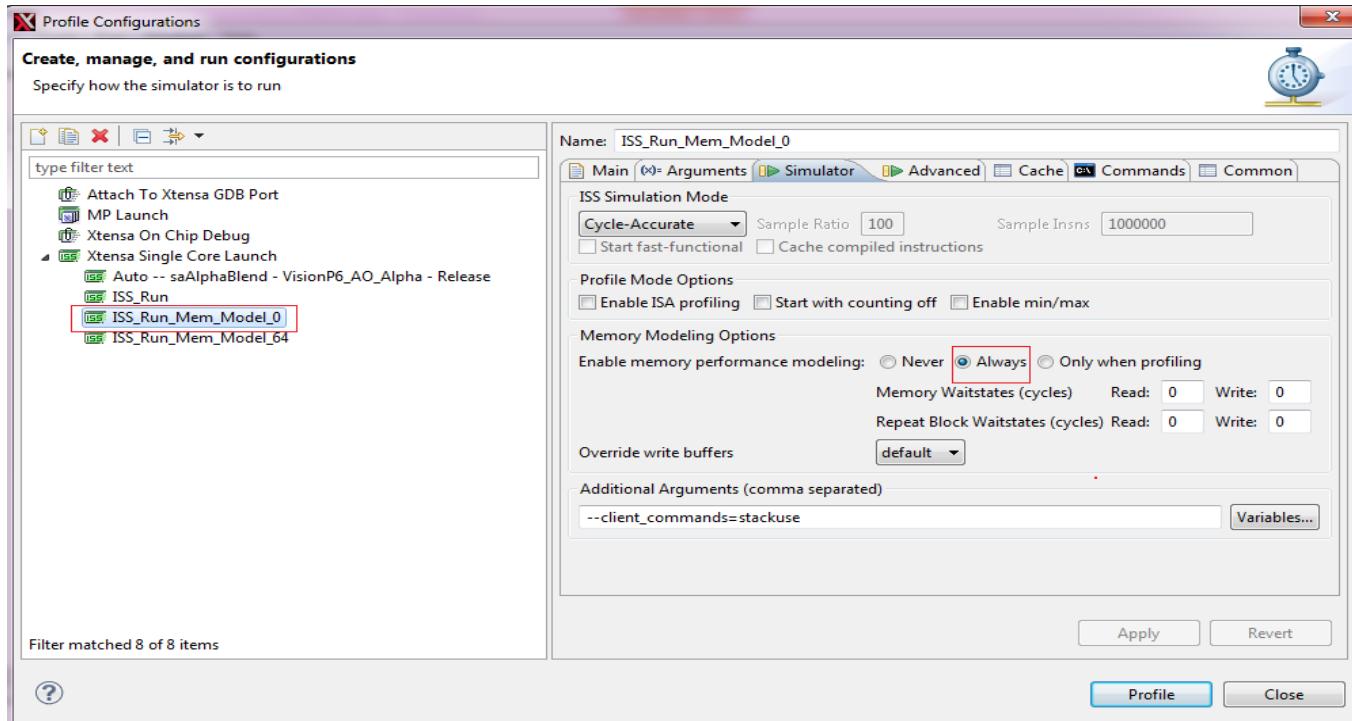


# Profiling **with** Memory Modelling

- At later stages of development you might be concerned with more realistic modelling of memory accesses and latencies.
- This modelling is more line with the performance that you would see on HW (e.g. FPGA emulation Board or SoC).
- As soon as you start profiling with **memory modelling** you have to make sure that your **working buffers, variables and stack are in local DRAM**
- Otherwise you will likely see **large cycle count degradations** compared to profiling without memory modelling.
- It is easy to forget about some **variables** that you are using in your code but you could not fit in your **local DRAM**
- In order to find such problems it is convenient to toggle between different profile configurations and compare cycle count differences.
- The profiler annotates lines of source code which is helpful to narrow down exactly where cycles are spent.
- In the following slide we will show you how you can set up multiple **profile configurations** for that purpose.
- Even setting the **memory latency to 0** produces different results from running ISS **without memory modelling**. A load to SYSRAM will at least cause a **pipeline replay**.

# Custom Profile Configuration-I

“ISS\_Run\_Mem\_Model\_0” has memory performance modeling enabled with “0” wait states for read and write.

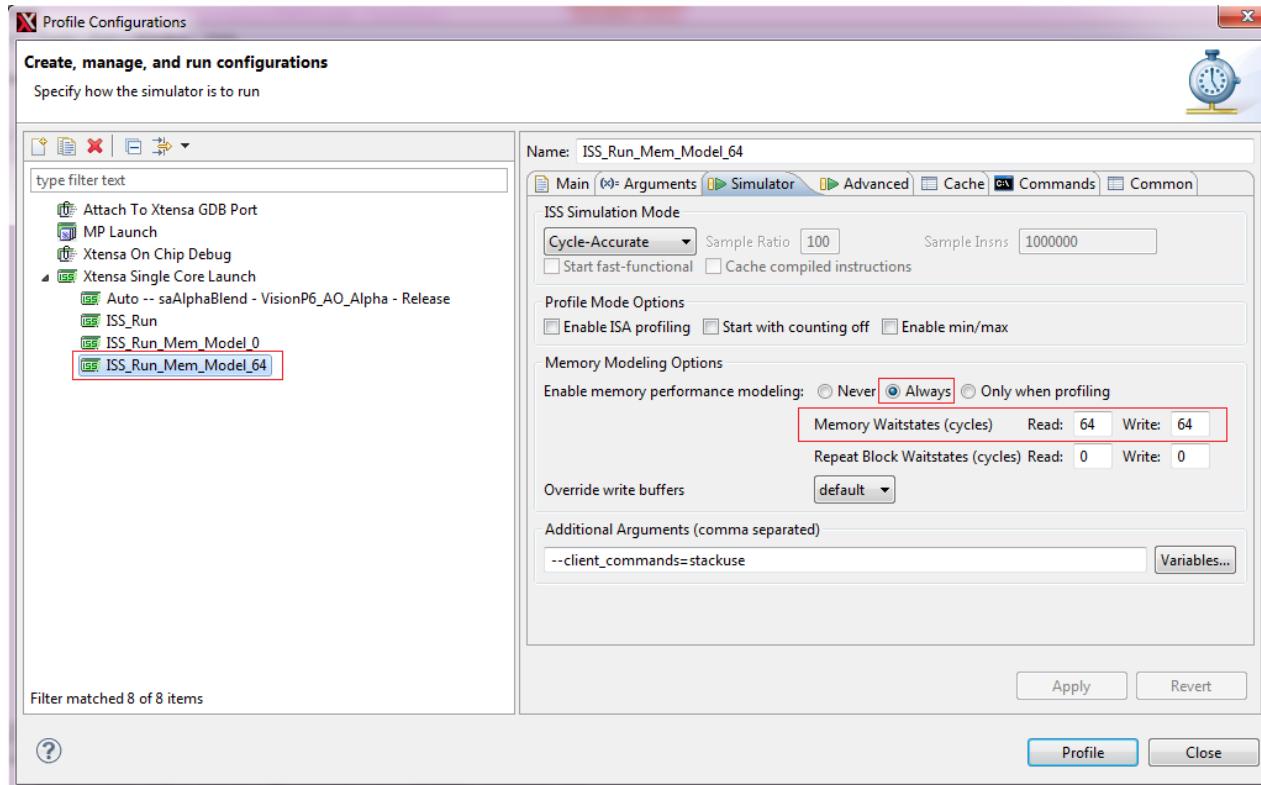


Function Name	Total (%)	Function	Children	Total	Called	Size (bytes)
_WindowUnderflow8	0.32	14	0	14	1	29
frame_dummy	0.82	35	0	35	1	57
xvAlphaBlend	65.29	2,775	0	2,775	1	274
_Atexit	0.96	41	0	41	1	110

Data Buffers  
in SYSRAM!!!

# Custom Profile Configuration-II

“ISS\_Run\_Mem\_Model\_0” has memory performance modeling enabled with “64” wait states for read and write.



Function Name	Total (%)	Function	Children	Total	Called	Size (bytes)
_WindowUnderflow8	0.05	14	0	14	1	29
frame_dummy	0.62	166	0	166	1	57
xvAlphaBlend	74.16	19,708	0	19,708	1	274
_Atexit	0.88	235	0	235	1	110

Data Buffers  
in SYSRAM!!!

# How to avoid Degradation from Memory Accesses?

- Place input buffers and variables in DRAM using \_\_attribute\_\_((section(".dram0.data"))))
- Make sure Stack is located in DRAM
- Use iDMA to move data from SDRAM to DRAM

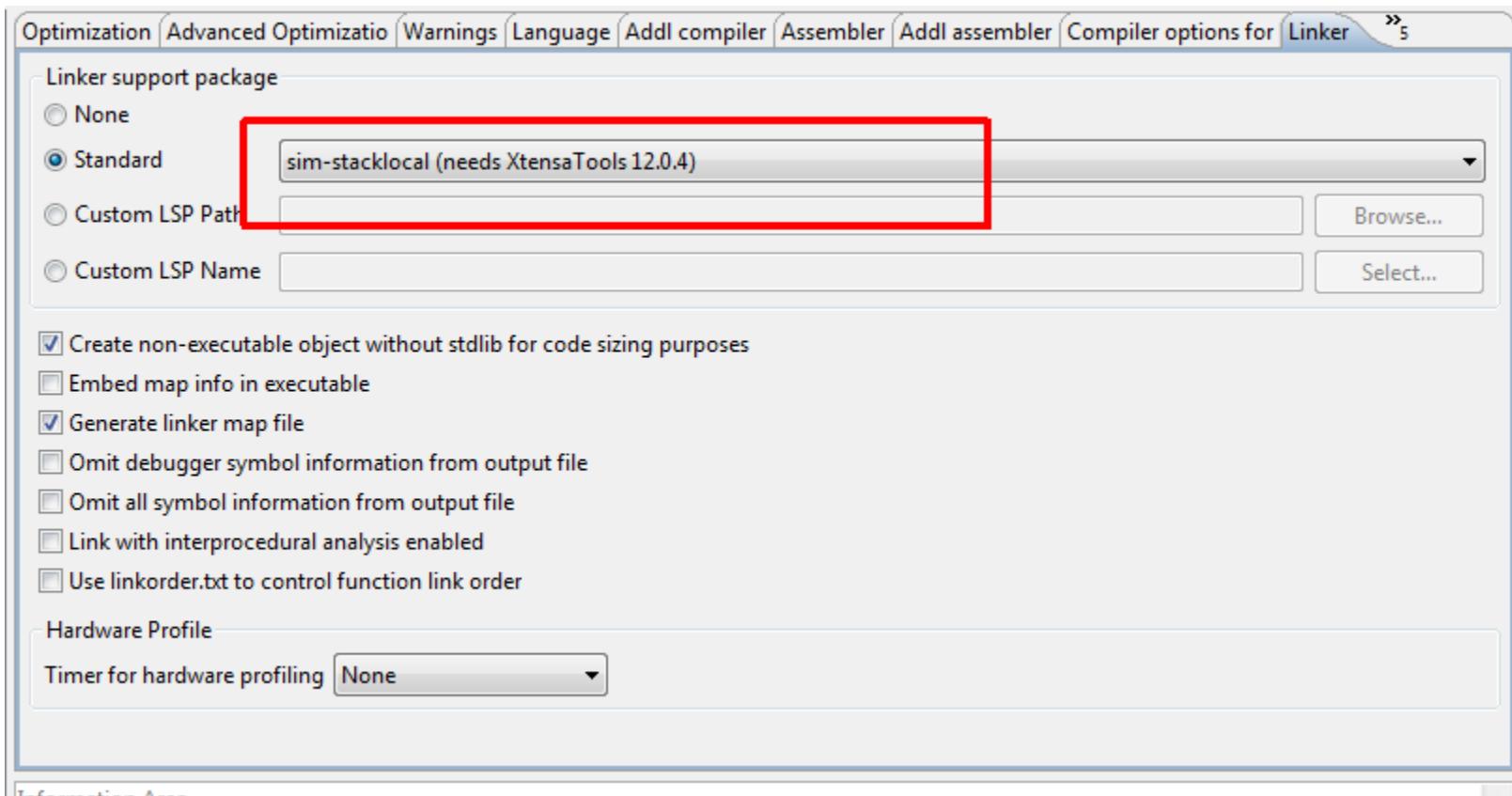
Cycles with “ISS\_Run\_Mem\_Model\_0” when buffers are in local DRAM”

Function Name	Total (%)	Function	Children	Total	Called	Size (bytes)
_WindowOverflow8	0.68	12	0	12	1	29
_WindowUnderflow8	0.79	14	0	14	1	29
frame_dummy	1.99	35	0	35	1	57
xvAlphaBlend	16.41	288	0	288	1	274
_Atexit	2.73	48	0	48	1	110

Data Buffers  
in DRAM!!!

# How do I place my Stack in local DRAM?

- Use a standard LSP that places the Stack in local DRAM: **sim-stacklocal**
- **sim-stacklocal** is a new standard LSP introduced in **RG.4**
- You can also create your own custom LSP (Please see Linker Support Packages (LSPs) Reference Manual for details)



# Performance Optimization check list

## End to End:

1. Are your profile cycles close to your high level estimate?

## Loop Schedules:

1. Do your inner loops SWP?
2. Does your loop hit expected HW bounds (MUL/ALU/L/S)?
3. Do you see up to 4 or 5 operations per cycle/instruction in your inner loops
4. Are the profile cycles close to loop schedule length \* iteration count?
5. Try to eliminate as many operations from outer loops since outer loop don't SWP.

# Performance Optimization check list **continued ...**

## Loop Iteration Counts

1. Is your loop count (after unrolling by compiler) large?
  - 16, 32, 64 will probably give you low overheads
  - 8,4,2 might start to create significant overheads
  - Shorter loop lengths will expose overheads sooner
2. Did you iterate over vectors vertically?

## Memory Accesses

1. Did you place your all your data in local DRAM or do you have L/S to SYSRAM?
2. Run ISS with and without memory modelling to eliminate data placement problems.
3. Make sure all your code is placed in IRAM if you don't have an I\$.

# Advanced and Optimization check list

1. Can you identify what is limiting your current loop schedule length?
  - You might be able to reduce the number of operations by finding one that fuses 2 existing operations
  - You might be able to break a long loop into two short ones that schedule better
2. Did you fully exploit the bit range of input data?
  - E.g. you might be able to replace a 32-way 16b\*16b MUL with a 64-way 8b\*16b MUL
3. Did you remove redundant operations?
  - Is there a way to re-use already computed results
4. Can you reduce precision of your algorithm (not bit exact) without degrading functional performance?
  - This often increases performance

# Summary

In this module, you learned how to assess performance of your code. You

- Learned about software pipelining (SWP)
- Saw how the compiler treats code and inner versus outer loops
- Used the profiler and learned to interpret its outputs

In earlier sections

- we taught you how to estimate performance
- taught you different styles to program Vision P6.

This should give you a rough idea about achievable performance on Vision P6.

Going beyond ...

- requires a deeper understanding of the details of available operations.
- Operations that you counted earlier as two separate operations might be available as a single fused operation
  - e.g. add and shift: an average operation might combine add and shift

The next 3 modules take a deep dive into the available operations on Vision P6.

# Quiz

- 1.** We say that a loop is software pipelined (SWP)
  - a) if portions of multiple loop iterations are performed in parallel by the assembly loop.
  - b) if it is compiled for a processor that has 5 or 7 stage pipeline.
  - c) if it is vectorized.
- 2.** Data placement attributes ...
  - a) are not that important because all your data will usually fit on local Data RAM
  - b) are not that important because access to System RAM is usually as fast as access to Data RAM
  - c) are very important for data structures that are read by the processor as System RAM access often has large latencies and results in processor stalls.
- 3.** Running ISS without memory modelling
  - a) is encouraged since ISS does not model system memory correctly anyway
  - b) is preferred since it makes your result look better
  - c) is often convenient at the early exploration stage of your implementation where you are mostly concerned with achieving good vectorization and scheduling but have not yet spent the effort to plan data flow and memory buffer sizes carefully.

Answers: 1a, 2c, 3c



# ISA Deep Dive

**Module** **10**

**Revision** **1.0**

**Version** **7.4**

**Estimated Time:**

- **For the lecture** **2.5 hours**
- **For the lab**

**Dependencies:**

**Module 8 – Programming with Intrinsics**

# Module Objectives

In this module, you will get an overview of the Vision P6 operations details

- Learn about Load/Store Operations
- Learn about the Multiply Operations
- Learn about Select Operations
- Learn about Reduction Operations



# Vector Load and Store Operations

**Submodule**    **10-1**

**Revision**              **1.0**

**Version**              **7.4**

**Estimate Time**

- **Lecture**              **0.5 hour**
- **Lab**

# Load and Store Operations

In this section we will cover

- Loading and storing data with 64 Byte alignment
- Loading and storing data with arbitrary Byte alignment
- Predicated loading and storing that conditionally load and store data elements
- Loading and storing a variable number of elements

# Normal and Aligning Loads

Memory Interface to local DRAM is **64 byte** wide

**Normal** vector loads **assume** that data is aligned to a

- 32 byte boundary for 32-element \* 8-bit loads
- 64 byte boundary for
  - 32-element \* 16-bit loads
  - 64-element \* 8-bit loads

**Aligning** vector loads allow the use of **any**

- 1 byte boundary memory data loading

# 32-element Load Operation IVP\_LVNX16\_XP

## C Syntax:

```
IVP_LVNX16_XP(xb_vecNx16 a /*out*/, xb_vecNx16 * b /*inout*/, int c);
```

- **Output:**
  - 1 VEC register (32 elements \* 16 bits)
- **Inputs:**
  - 1 AR register (32 bits) for address
  - 1 AR register (32 bits) for address offset post-increment

## Description:

- Load 32 elements \* 16 bits from 64-Byte aligned memory location
- Post-increment address value b by the value in c

# Load/Store Addressing Modes

Vector load store instructions have four addressing modes

I: immediate offset

- Load address is base AR register + immediate offset value
- Immediate offset is a multiple of 64

IP: immediate offset + post-update

- Load address is base AR register value
- Base address is post-incremented

X: register offset

- Load address is base AR + AR register offset value

XP: register offset+ post-update

- Load address is base AR register value
- Base address is post-incremented by AR register value

# Aligning Loads

- First load primes an **alignment register** (1 cycle penalty)
- Subsequent loads merge elements from alignment register and current load
- No additional penalty for contiguous loads (stride = vector length)
- **Other strides** will require re-priming (1 cycle penalty)
- Compiler will default to using aligning load operations when **auto-vectorizing** C-Code

# Alignment Register Prime Operation IVP\_LA\_PP

## C Syntax:

```
valign IVP_LA_PP(const xb_vecNx16 * b)
```

- **Output:**
  - 1 VALIGN register (64 Bytes)
- **Inputs:**
  - 1 AR register (32 bits) for (unaligned) load address

## Description:

- The 6 bits [5:0] from address value in AR register c are set to 0 to create an address with 64 Byte alignment
- The 64Bytes stored at that memory address are loaded into the VALIGN register

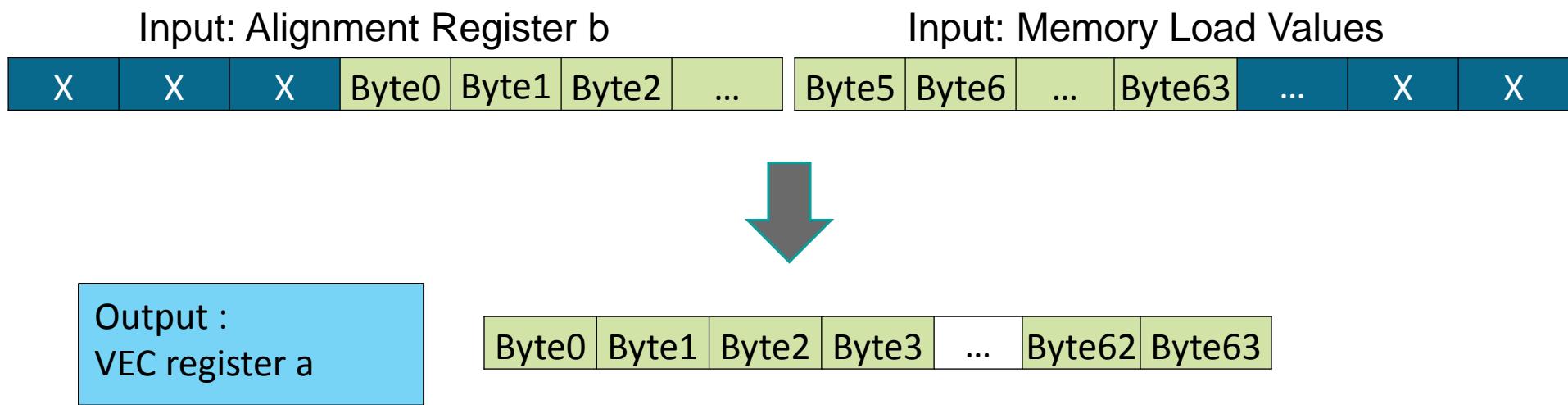
# 32-element Load Operation IVP\_LANX16\_IP

## C Syntax:

```
IVP_LANX16_IP(xb_vecNx16 a /*out*/, valign b /*inout*/, const  
xb_vecNx16 * c /*inout*/)
```

- **Output:**
  - 1 VEC register (32 elements \* 16 bits)
- **Inputs:**
  - 1 ALIGN register (32 elements \* 16 bits)
  - 1 AR register (32 bits) for address

# Diagram of Load Operation: IVP\_LANX16\_IP



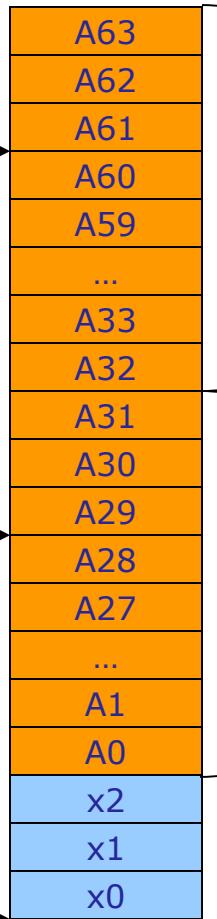
# 32-element Load Operation IVP\_LANX16\_IP

## Description:

- The 6 bits [5:0] from address c are used as a **byte shift** amount
- The address value of  $cplus63 = c + 63$  is calculated and the lower 6 bits [5:0] of  $cplus63$  address are set to zero to make it 64-byte aligned
- 64 byte-elements are loaded from that aligned memory location
- Elements from the loaded data and the alignment register are merged
  - The memory data forms the upper and the alignment register the lower part of a 1024 bit word
  - The output elements are chosen by applying the byte shift above
  - The lower 512 bits of that **byte shifted** word form the 64 output elements

# Aligning Load Example: offset by 3 elements

Data Memory:  
16-bit elements

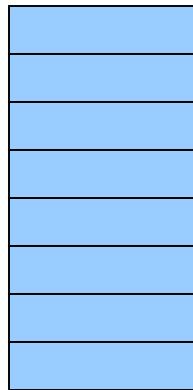


desired  
vector 1

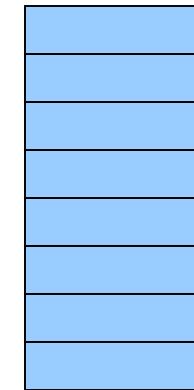
desired  
vector 0

64-byte boundaries

Alignment Register (16-bit elements)

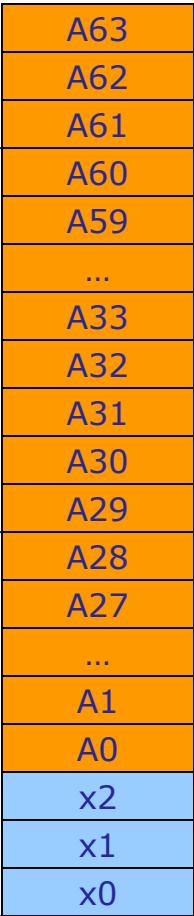


Destination Vector  
(16-bit elements)



# Aligning Load Example: offset by 3 elements

Data Memory:  
16-bit elements



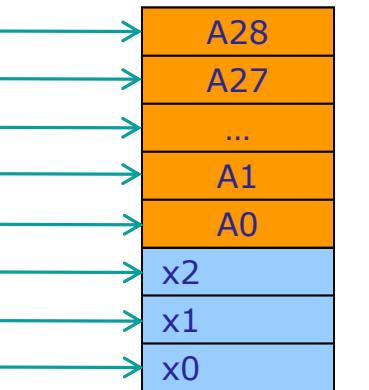
Step 1: IVP\_LA.PP: load & prime alignment register

Step 2: IVP\_LANX16.IP: aligning Load

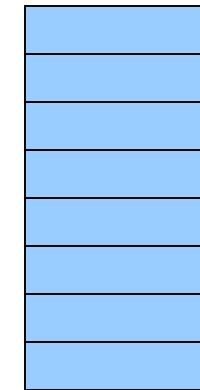
loads output vector by merging data from alignment register and  
memory load data

(same cycle) reloads alignment register

Step 3....: Repeat Step 2



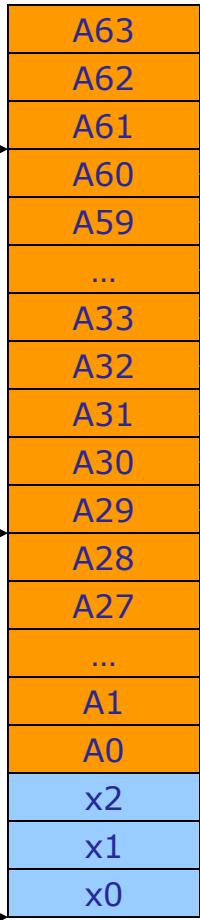
Alignment Register (16-bit elements)



Destination Vector (16-bit elements)

# Aligning Load Example: offset by 3 elements

Data Memory:  
16-bit elements



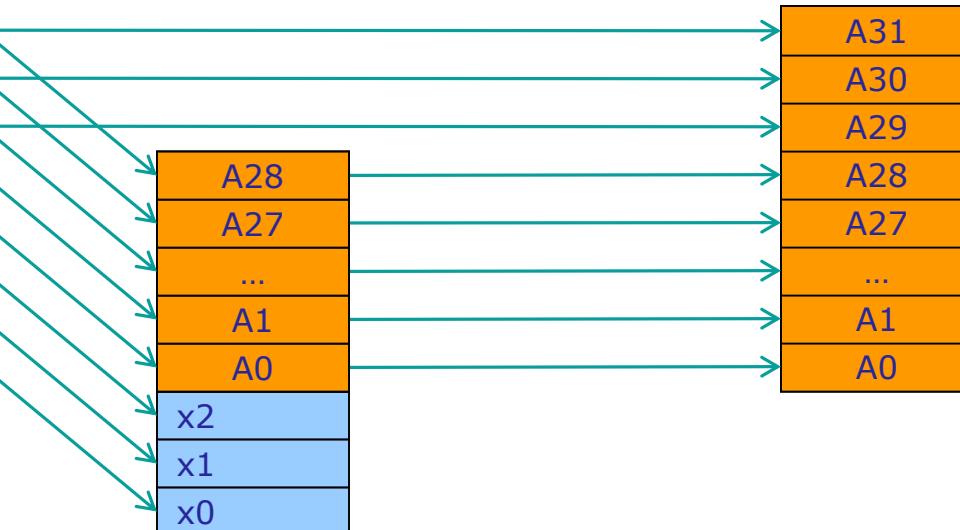
Step 1: IVP\_LA.PP: load & prime alignment register

Step 2: IVP\_LANX16.IP: aligning Load

loads output vector by merging data from alignment register and  
memory load data

(same cycle) reloads alignment register

Step 3....: Repeat Step 2

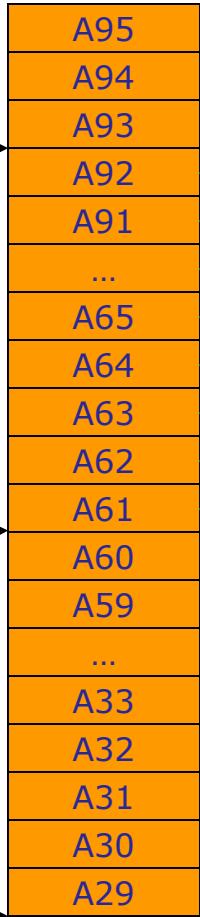


Alignment Register (16-bit elements)

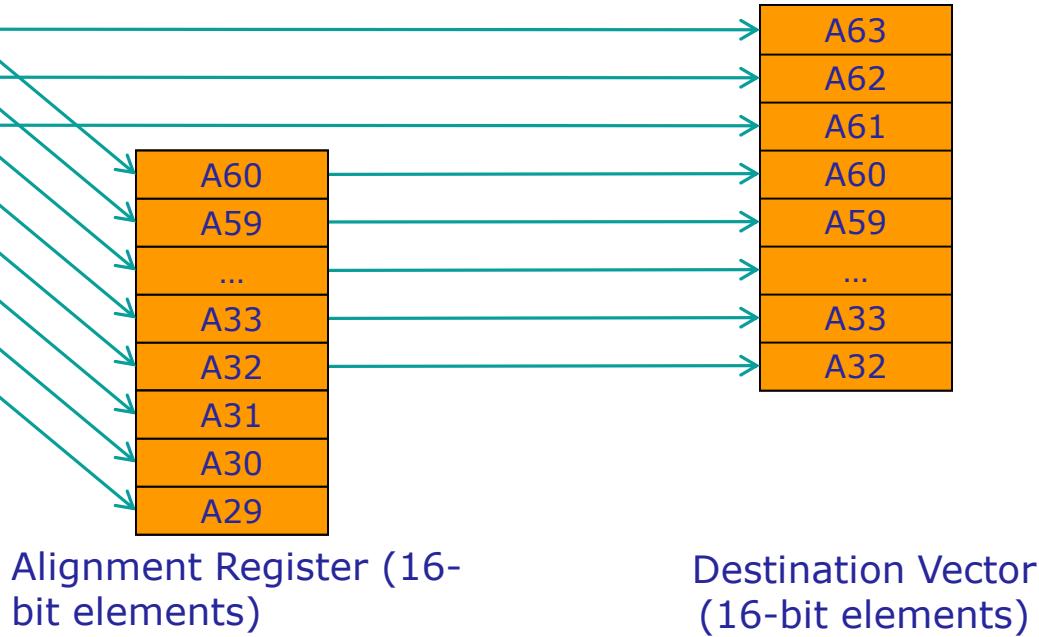
Destination Vector (16-bit elements)

# Aligning Load Example: offset by 3 elements

Data Memory:  
16-bit elements



Step 3: Repeat aligning load IVP\_LANX16.IP  
– Loads adjacent 32 elements



# Example: Using Aligning Load Operations

```
int32_t SumAbsDiff(uint16_t *dataIn0/*unaligned*/,
                    uint16_t *dataIn1/*aligned*/, int32_t N){

    int32_t i;
    valign va0;
    xb_vecNx16 *pv0 = (xb_vecNx16 *) dataIn0;
    xb_vecNx16 *pv1 = (xb_vecNx16 *) dataIn1;
    xb_vecNx16 vDiff, vAbs, vIn0, vIn1;
    xb_int16 sum=0;

    va0 = IVP_LA_PP(pv0); <-- bracketed block

    for(i=0;i<N;i++){
        IVP_LANX16_IP(vIn0, va0, pv0); <-- bracketed block
        vIn1 = *pv1++; <-- bracketed block

        vDiff = vIn0 - vIn1;
        vAbs = IVP_ABSNX16(vDiff);
        sum += IVP_RADDNX16(vAbs);
    }
    return sum;
}
```

- The priming Alignment Register operations IVP\_LA\_PP is required at the beginning of the loop to initialize the ALIGN register va0
- Subsequent loads will not require re-priming as long as all the data elements pointed to by **pv0** are contiguous
- The aligned data elements pointed to by **pv1** can be accessed via pointer de-referencing

# Aligning Stores: Overview

There are 3 steps to handling aligning stores

- zero the alignment register
- issue subsequent store operations
- flush and store the final bytes from the alignment register

Aligning stores operations will store to memory:

- portion of alignment register bytes
- a portion of current vector data

Subsequent stores will have no additional penalty when storing consecutive 32/64-byte addresses.

# Aligning Store Operation IVP\_SANX16\_IP

## C Syntax:

### IVP\_SANX16\_IP

```
(xb_vecNx16 a, valign b /*inout*/, xb_vecNx16 * c /*inout*/)
```

- **Output:**
  - In general 32 elements \* 16 bits are stored to memory
  - 1 ALGIN register (32 elements \* 16 bits)
  - 1 AR register (32 bits) with post-incremented memory address
- **Inputs:**
  - 1 VEC register (32 elements \* 16 bits)
  - 1 ALGIN register (32 elements \* 16 bits)
  - 1 AR register (32 bits) for memory address

# Diagram: Aligning Store Operation IVP\_SANX16\_IP

Input: AR register c

31:6	5:0
------	-----

The bits 5:0 are used as a *byte offset*. Assume this offset is 3 Bytes in this example

X	X	X	B0	B1	...	B60
---	---	---	----	----	-----	-----

Output: Stored values in Memory

Input: VEC register a

B0	B1	B2	B3	...	B62	B63
----	----	----	----	-----	-----	-----

The most significant bit of the alignment register is set by this operation if *byte offset* is non-zero. This bit will be checked by the flush operation

B61	B62	B63	X	...	X	X
-----	-----	-----	---	-----	---	---

Output: Alignment Register b

# Align Register Flush Operation: IVP\_SAPOS\_FP

C Syntax: **IVP\_SAPOS\_FP**(valign a /\*inout\*/, xb\_vecNx16 \* b)

- **Inputs:**

- 1 ALGIN register (64 elements \* 8 bits)
- 1 AR register (32 bits) holding memory address

- **Output:**

- A partial number of bytes are stored starting from the memory address in AR register b

## Description:

- The 6 address bits b[5:0] define the **number of Bytes** to store
- A 64-Byte aligned address is formed by setting **bits b[5:0]** to **0**
- Byte[0] to Byte[number of Bytes -1] are stored at this aligned memory location
- This operation is **conditional** on bit 511 of the ALIGN register **a** being set to **1**
- At the end of this operation bit 511 of the ALIGN register **a** is **cleared**

# Zero Align Register Operation: IVP\_ZALIGN

**C Syntax:** extern valign IVP\_ZALIGN(void)

Description:

- This operation clears a register of type ALIGN
- Usage: valign va0 = IVP\_ZALIGN();
- This operation should be used to initialize an alignment register before use in an aligning store operation
- It ensures that Bytes are not erroneously stored by the subsequent flush operation

# Example: Using aligning store operations

```
xb_vecNx16 *pv;  
xb_vecNx16 vVec0, vVec1, vVec2;  
  
valign va0 = IVP_ZALIGN(); ←  
  
IVP_SANX16_IP(vVec0, va0, pvOut); ←  
vVec = IVP_LVNX16_I(pv, 64);  
IVP_SANX16_IP(vVec1, va0, pvOut); ←  
vVec = IVP_LVNX16_I(pv, 128);  
IVP_SANX16_IP(vVec2, va0, pvOut); ←  
  
IVP_SAPOS_FP(va0, pvOut); ←
```

- clear the alignment register before usage
- Multiple aligning store operations can be issued in sequence when storing to contiguous vector locations
- A flush operation is required once to store the bytes left over in the alignment register

# Loading and Storing a variable number of elements

**IVP\_LAVNX16\_XP**(xb\_vecNx16 a /\*out\*/, valign b /\*inout\*/, const xb\_vecNx16 \* c /\*inout\*/, int d)

- This operation loads a variable number of Bytes defined by the AR register d
- This operation is supposed to be used in conjunction with IVP\_LA\_PP

**IVP\_SAVNX16\_XP**(xb\_vecNx16 a, valign b /\*inout\*/, xb\_vecNx16 \* c /\*inout\*/, int d)

- This operation stores a variable number of Bytes defined by the AR register d
- This operation is supposed to be used in conjunction with IVP\_ZALIGN and IVP\_SAPOS\_FP

**Please refer to the online ISA HTML for details of these operations!**

# Load Operations using 2 Load/Store Units

- pdvec is an **unaligned** address in an **AR** register
- vec0, vec1 **VEC** Registers holding the result of the load operations
- va is an **ALIGN** register

Operation	Description
IVP_L2AU2NX8_IP(vec0, va, pdvec)	<b>Prime and Load</b> loads vec0 and primes va for next load
IVP_L2A4NX8_IP(vec1, vec0, va, pdvec)	<b>Dual Load</b> load 2 adjacent vectors into v0 and v1 Requires primed alignment register
IVP_L2U2NX8_XP(vec0, pdvec, 64)	<b>Single cycle unaligned load</b> load to vec0 (no priming needed)



# Vector Multiply Operations

**Submodule**    **10-2**

**Revision**              **1.0**

**Version**              **7.4**

**Estimate Time**

- **Lecture**              **1 hour**
- **Lab**

# Vector Multiply Operations in Vision P6

- There is a large number (**over 100**) multiply operations in Vision P6
- Trying to find the exact one that meets your need for a particular implementation can be difficult
- In this sub-module we want to help the user to **find his way to through the forest** of multiply operations
- We will do this by **classifying** the Multiply Operations into different **groups**
- We will start by first describing the most **common MUL operations**
- Then we will describe the different **flavors** of the MUL operations that will allow us to **classify** the MUL operations into different **categories**.
- After covering the MUL operations we will also talk about **Pack operations** which are often used in conjunction with MUL operations

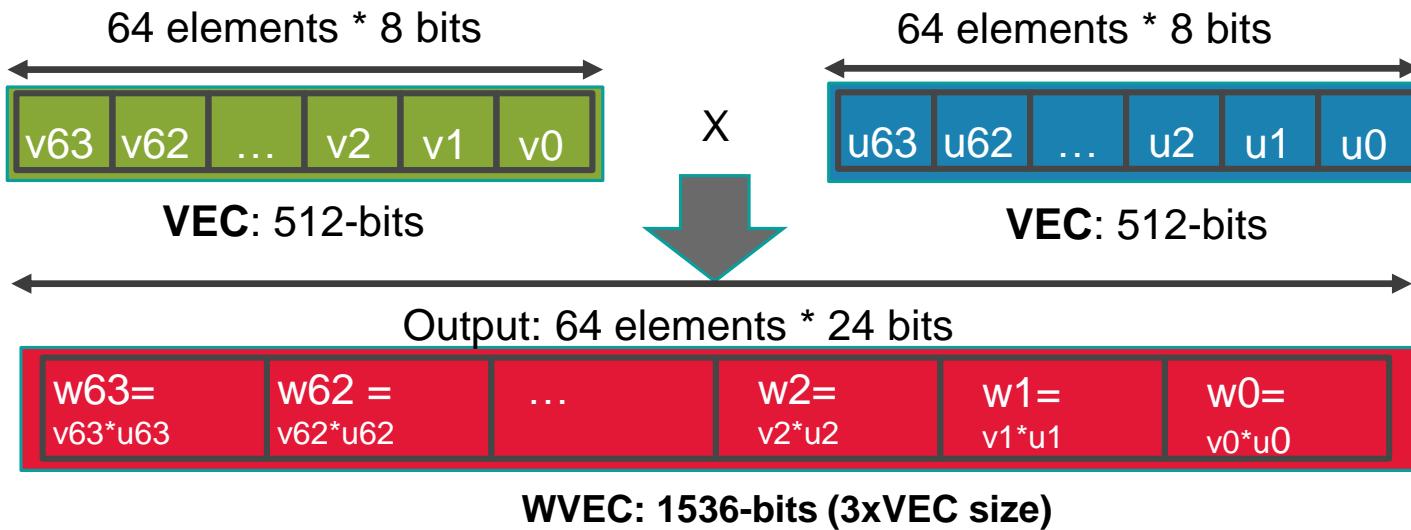
# Vision P6 Vector Multiplication Basics

- The vector multiplication unit supports
  - Multiply (**MUL**),
  - Multiply-Accumulate (**MULA**) and
  - Multiply-Subtract (**MULS**) operations
- Up to 64 parallel (SIMD) outputs can be processed per cycle
- **Input Operand** is usually narrow vector (**VEC**) register (64 Bytes)
- **Output Operand** is
  - narrow vector register (64 Bytes) or
  - wide-vector register(**WVEC**) (192 Bytes = 3 \* 64)
- Later in this module we will explore many MUL operation variations with different input types
- An example of a common MUL operation will follow on the next slide

# 64-way 8bit MUL Operation with 64-way 24 bit output

xb\_vec2Nx24 w =

IVP\_MUL2NX8(xb\_vec2Nx8 u, xb\_vec2Nx8 v);



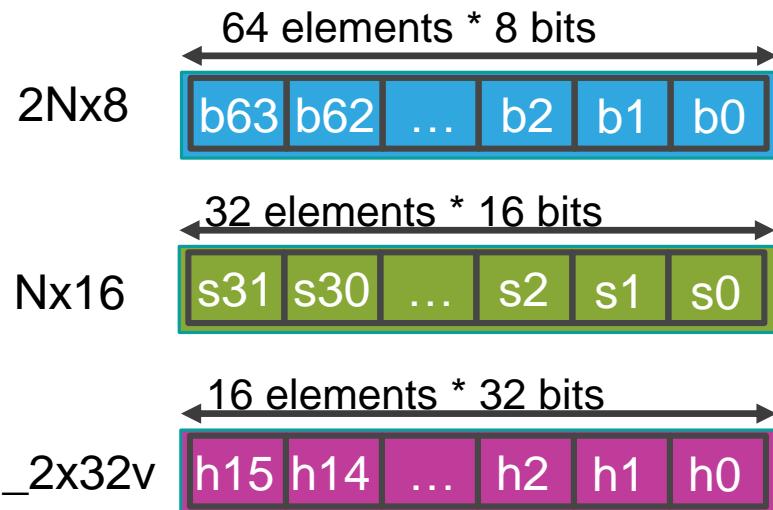
Only 16 of the 24 output bits are used (8 guard bits)

# Multiply Category: Element Bit Width

- 3 element bit-widths are supported 8, 16 and 32 bit
- Bit widths imply the number of elements (ways): 64, 32 or 16
- The 3 bit-widths are implied data types are shown below

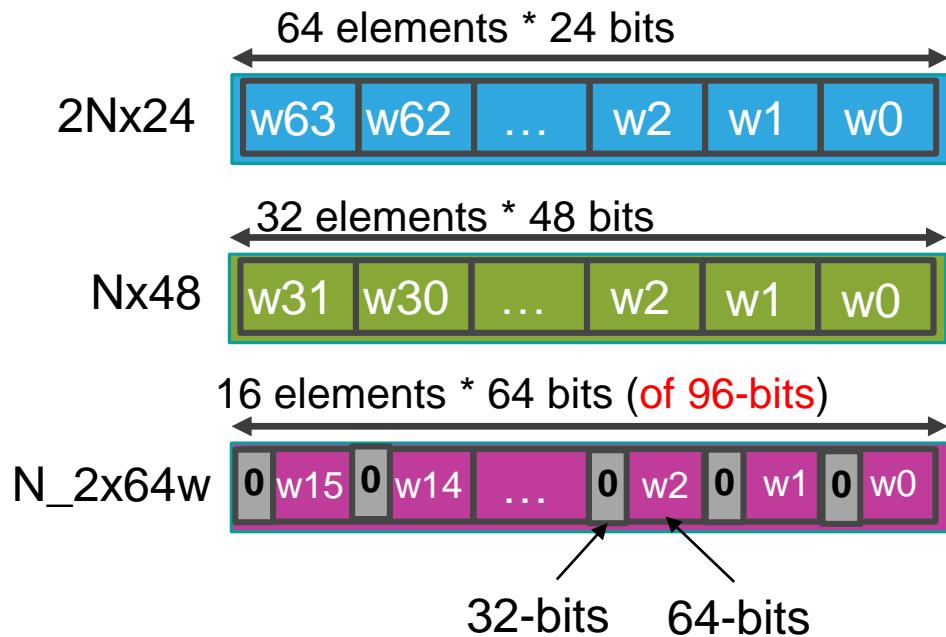
**Input:**

**VEC:** 512-bits



**Output:**

**WVEC** 1536-bits



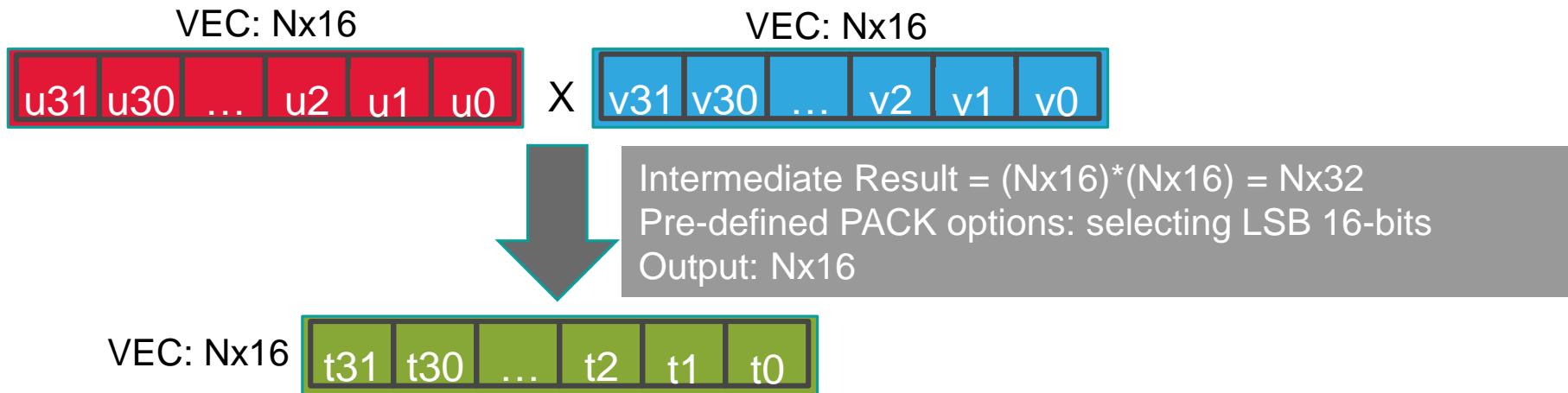
# Multiplication Category: Packing of Results

- Multiplication of **two n-bit** Operands produces a **2\*n-bit** result
- Operations that **change** the output **bit-width** - discarding extra bits using right shift or masking MSB bits - are called **PACK** operation
- A category of operations pre-define **PACK** methods which store result directly into narrow-vector
- We call this operation category (narrow-vector) **MUL PACK Operations**
- MUL operations with WVEC outputs are called **Wide-vector MUL Operations**
- These later provide **guard bits** for **accumulation** of results, e.g. 16 guard bits to store Nx32 result in WVEC of type Nx48
- These MUL Operations are usually followed by **pack operations** the we will cover later
- The next slide shows an **example of a MUL Pack Operation**

# Example of narrow-vector Multiply-pack Operations

xb\_vecNx16 t =

IVP\_MULNX16PACKL(xb\_vecNx16 u, xb\_vecNx16 v);



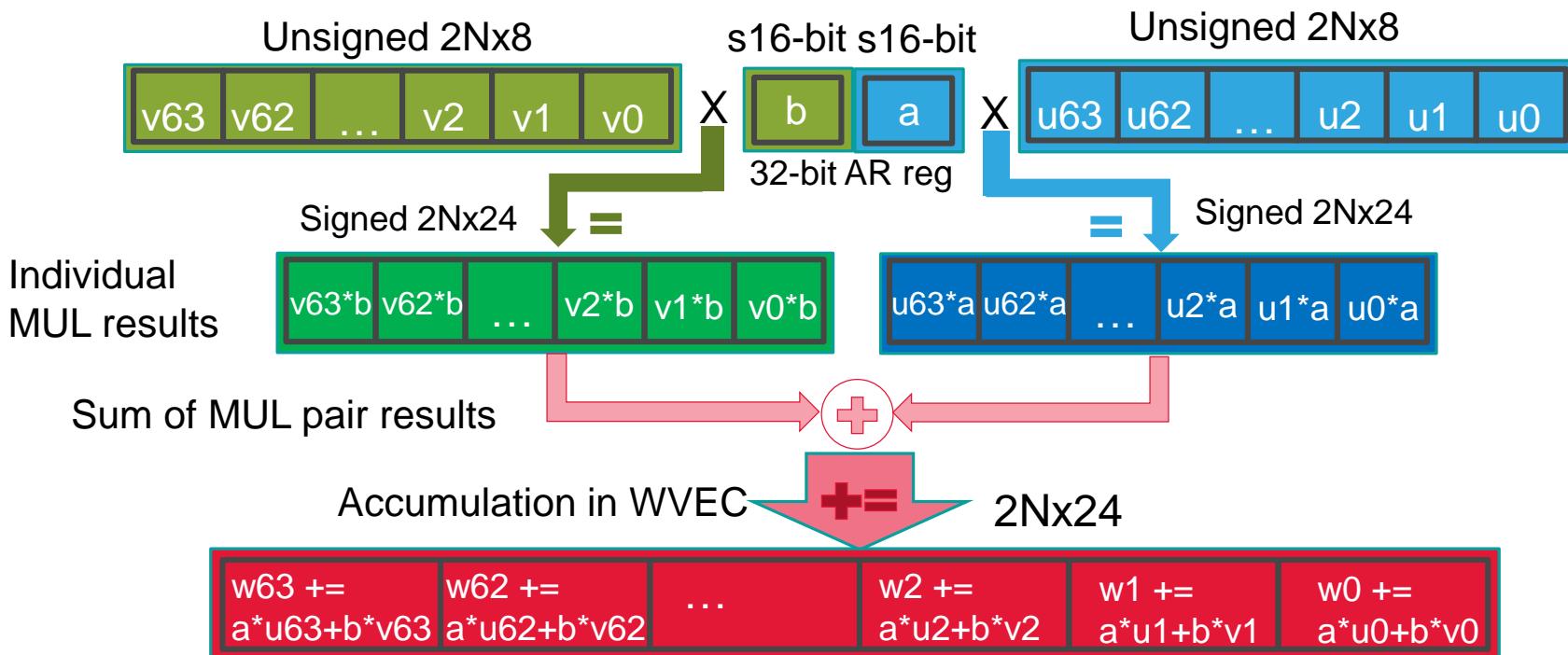
# Many more Multiplication Categories

- **Paired/Quad:** More than 1 MUL per VEC-way
  - Results from 2 or 4 MULs accumulated per way
  - Output SIMD width remains same as single MUL
- **Mixed Operand Sizes:**
  - E.g.,  $2N \times 8 * N \times 16$
- **VEC \* scalar (AR)**
  - Reduces VEC register pressure
- **Mixed Signed/Unsigned**
  - Output is sign extended
- **Accumulating/Subtracting**
- **Interleaved:** interleave Operand elements before MUL

The next slide shows and example of a paired MUL which mixes many of the above flavors

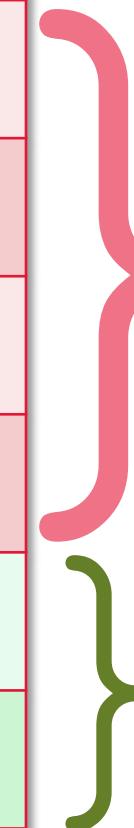
# 8 x 16 bit Unsigned Signed Paired Multiply Accumulate Operation

xb\_vec2Nx24 w = IVP\_MULUSP2N8XR16(xb\_vec2Nx8U v, xb\_vec2Nx8U u, int ab);

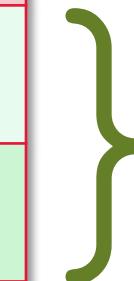


# MUL Operation Classification by Operand bit-width

Multiplication Variants				
Input 1		Input 2		Result
8b	x	8b	=	24b
8b	x	16b	=	24b
16b	x	16b	=	48b
16b	x	32b	=	64b
16b	x	16b	=	16b
32b	x	32b	=	32b

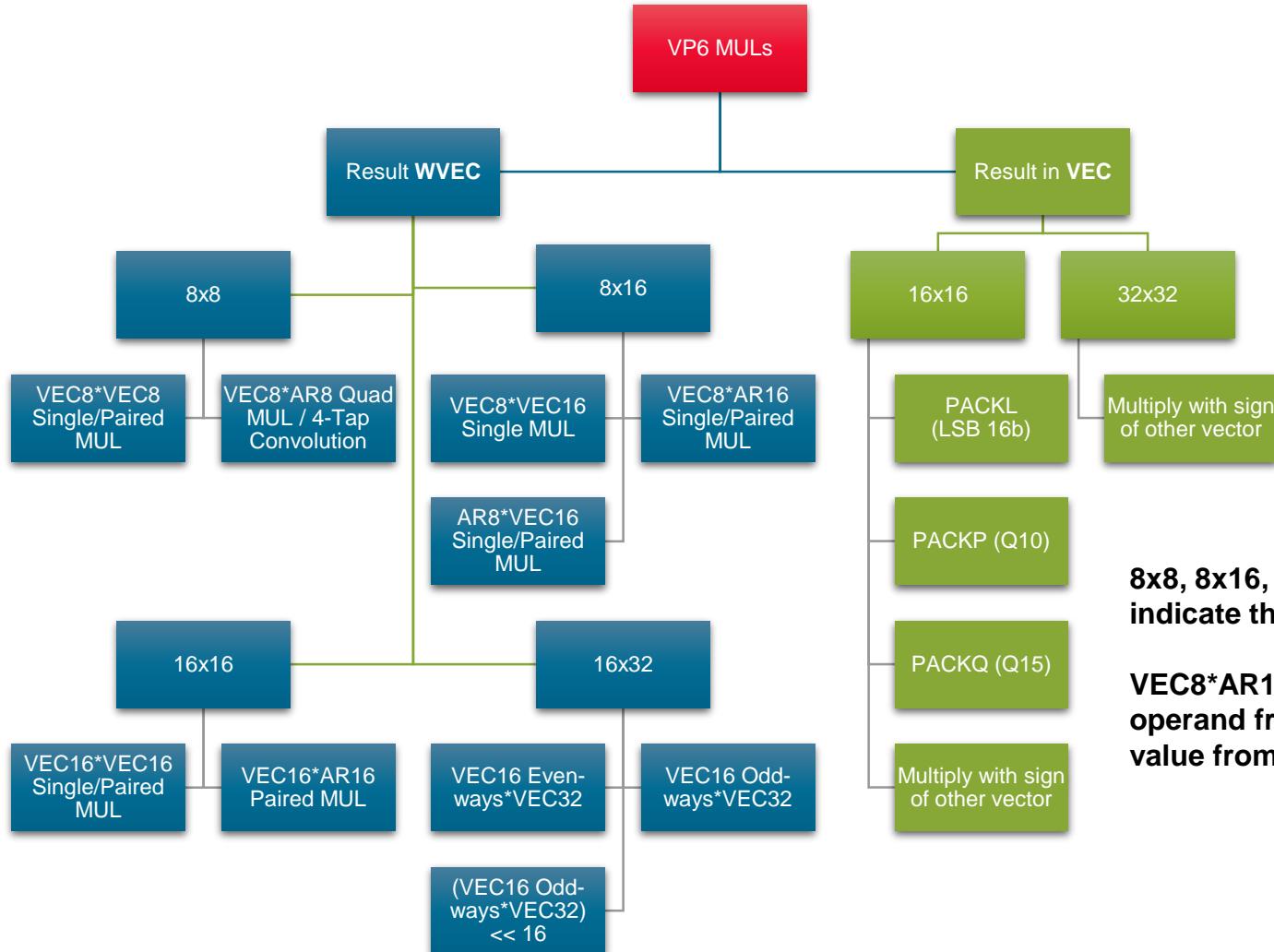


Output  
in  
Wide-  
vector



Output  
in  
Narrow-  
vector

# Vector Multiplication Classification Chart



**8x8, 8x16, 16x16, 16x32 and 32x32 indicate the operand bit-widths**

**VEC8\*AR16 indicates an 8-bit operand from a VEC and a 16-bit value from an AR register**

# Explore MUL operations by element bit-width and narrow and wide output Type

- In the slides that follow we will now explore the wide MUL operations by input operand bit-width
  - 8x8
  - 8x16
  - 16x16
  - 16x32
- After that will then go through the narrow MUL Pack operations

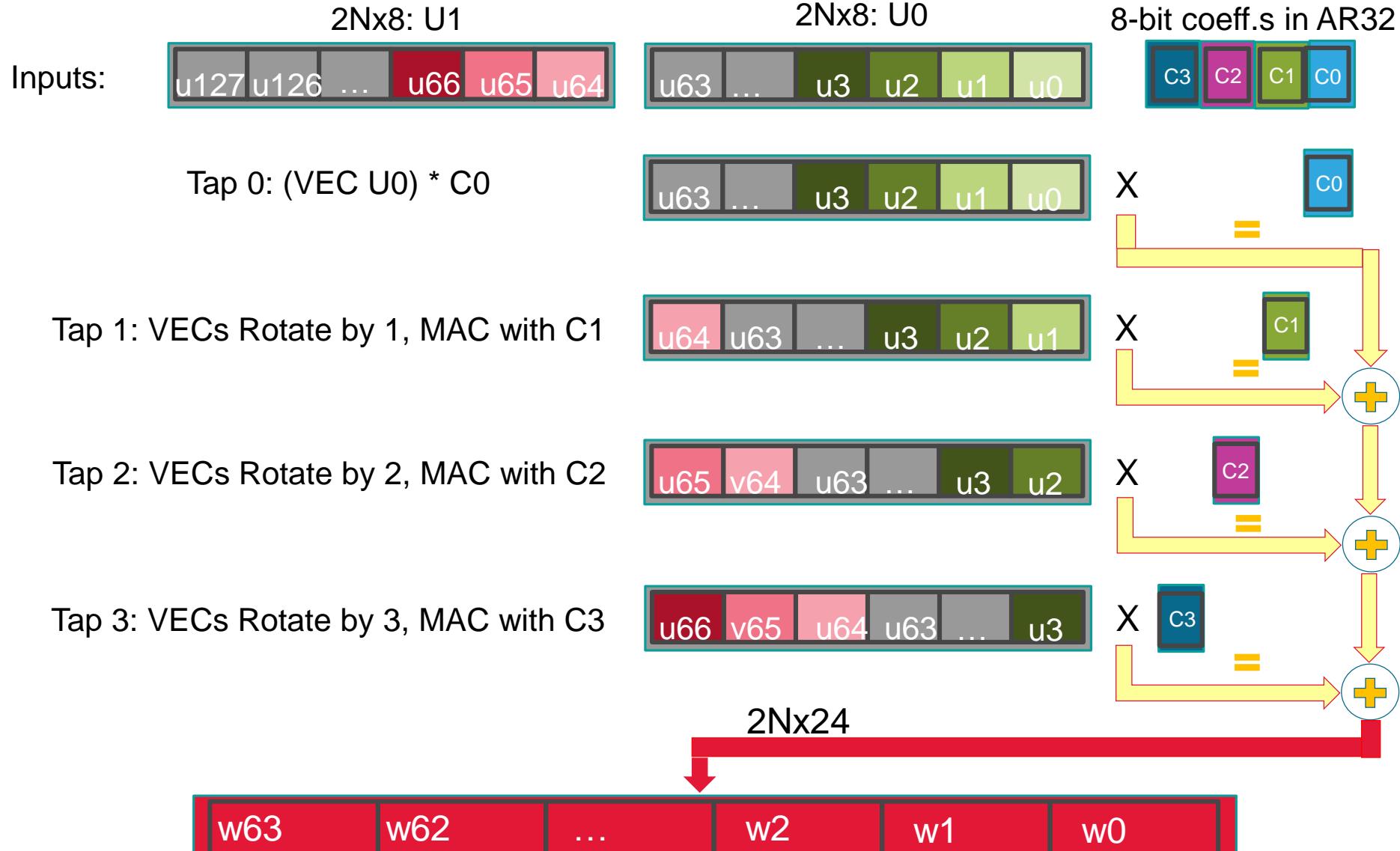
# **8bx8b = 24b WVEC MUL Ops**

- Single,
- Paired,
- Quad and
- 4-tap convolution ops are available

Multiply Type	Operation	Input Operand 1	Input Operand 2
Single MUL	IVP_MUL2NX8	VEC 2Nx8	VEC 2Nx8
Paired MUL	IVP_MULP2NX8	2 VEC 2Nx8	2 VEC 2Nx8
Quad MUL	IVP_MULQ2N8XR8	4 VECs 2Nx8	(4 8-bit scalars) In AR 32-bit
4-Tap Convolution	IVP_MUL4T2N8XR8	2 VEC 2Nx8	(4 8-bit scalars) In AR 32-bit

Ops Naming: IVP\_MUL[US|UU][P|Q|4T][A]2N[X8|8XR8]

# 4-Tap Convolution: IVP\_MUL4T2N8XR8



# 8bx16b = 24b WVEC MUL Ops

- Single and Paired Multiplications are available
- Nx16 vectors are Interleaved (**not concatenated**) to form 2Nx16 vector

Multiply Type	Operation	Input Operand 1	Input Operand 2
Single MUL	IVP_MUL <del>I</del> <b>2</b> NX8X16	VEC 2Nx8	2 Nx16 VECs <b>Interleaved</b>
Single MUL	IVP_MUL <del>I</del> <b>2</b> NR8X16	AR 8-bit	2 Nx16 VECs <b>Interleaved</b>
Single MUL	IVP_MUL2N8XR16	VEC 2Nx8	AR 16-bit
Paired MUL	IVP_MUL <del>P</del> <b>2</b> NR8X16	2 8-bit Scalars (16b in AR)	2 (2 Nx16 VECs) <b>Interleaved</b>
Paired MUL	IVP_MUL <del>P</del> <b>2</b> N8XR16	2 VECs 2Nx8	2 16-bit Scalars (32b in AR)

Ops Naming: IVP\_MUL[US][P][A][I]2N[X8|R8|8]X[R]16

# Example: 64-element Multiply Interleaved Operation

xb\_vec2Nx24 w =

IVP\_MULI2NX8X16(xb\_vec2Nx8 t, xb\_vecNx16 v, xb\_vecNx16 u)

Input: 32 elements \* 16 bits



interleave

Input: 32 elements \* 16 bits



64 elements \* 16 bits



X



Input: 64 elements \* 8 bits



Output: 64 elements \* 24 bits



# **16bx16b = 48b wvec MUL Ops**

- Single and Paired Multiplications are available

Multiply Type	Operation	Input Operand 1	Input Operand 2
Single MUL	IVP_MULNX16	VEC Nx16	VEC Nx16
Paired MUL	IVP_MULPNX16	2 VEC Nx16	2 VEC Nx16
Paired MUL	IVP_MULPN16XR16	2 VEC Nx16	2 16-bit Scalars (32b in AR)

Ops Naming: IVP\_MUL[US|UU][P][A|S]N[X16|16XR16]

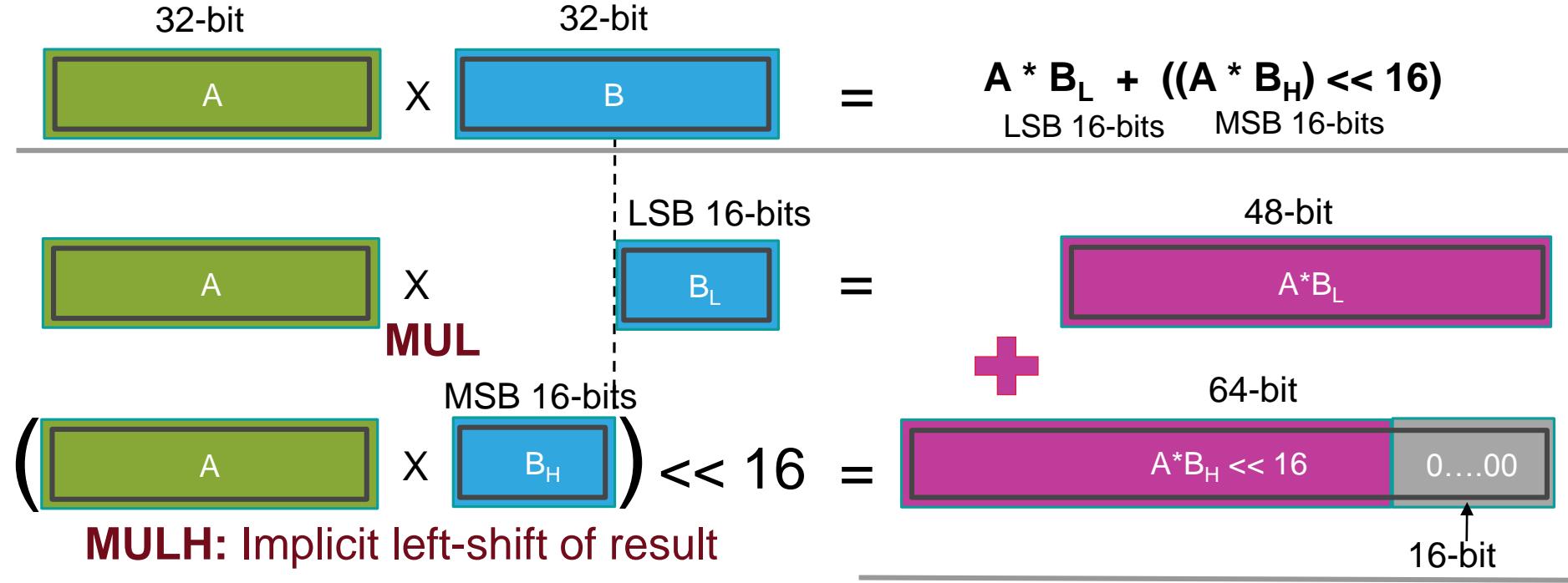
# 16bx32b = 64b wvec MUL Ops

- Only Single Multiplications
- “H” high 16-bit x 32-bit multiply operations shift the result by 16-bits to the left
- These operations facilitate 32x32 multiplications
- An example of this will follow on the next slide

Multiply Type	Operation	Input Operand 1	Input Operand 2
Single MUL	IVP_MULN_2X16X32_0	Even 16-ways (0, 2,.., 30)	VEC N_2x32v
Single MUL	IVP_MULN_2X16X32_1	Odd 16-ways (1, 3,.., 31)	VEC N_2x32v
Single MUL	IVP_MULHN_2X16X32_1 (H->High)	Odd 16-ways (1, 3,.., 31)	VEC N_2x32v

Ops Naming: IVP\_MUL[SU|US|UU][A|S][H]N\_2X16X32\_[0|1]

# MULH (High) Usage in 32x32 MUL



**MULH with accumulation (MULAH) is used to get 32\*32 output in 2 cycles**

- Operand Signs vary for MUL and MULH
- A can be signed or unsigned
- $B_L$  is always considered as unsigned (irrespective of sign of B) i.e.  $A * (\text{unsigned}) B_L$  but  $B_H$  assumes sign of B i.e.  $(A * (\text{sign of } B) B_H) << 16$

# Multiplication Throughput Table

Feature	MUL Type			
	8x8 (20 Ops)	8x16 (20 Ops)	16x16 (19 Ops)	16x32 (36 Ops)
MULs/vector-way (per cycle)	Up to 4	Up to 2	Up to 2	1
Vector SIMD Width	64	64	32	16
Accumulator width (per vector way)	24b	24b	16b or 48b	64b
Max MUL rate (Total MULs/cycle)	256	128	64	16
MUL & Subtract	No	No	Yes (limited)	Yes (full)
Special Features	4-Tap convolution			32x32 support by using MUL 16x32

# Narrow Multiply-Pack Operations

- For the rest of this sub-module we will focus on Multiply-Pack Operations
- Multiply-Pack Operations have
  - Input: 1 VEC register, 32 elements \* 16 bit
  - Output: 1 VEC register, 32 elements \* 16 bit
- 16 x 16 bit multiply produces a 32-bit result, but only 16 bits of it are stored in the output VEC register
- Operations versions for **signed** and **unsigned** inputs exist
- **accumulating (A)** and **subtracting (S)** versions of the operation exist
- **Predicated True** versions are notated with **(T)**
- In the next slides we will focus on these operations
  - **IVP\_MUL(A/S)NX16PACKL(T)** - producing lower 16 bit results
  - **IVP\_MUL(A/S)NX16PACKP(T)** - producing 16-bit fractional Q5.10 results
  - **IVP\_MUL(A/S)NX16PACKQ(T)** - producing 16-bit fractional Q15 results

## 32-element Multiply-Pack Operation

### ***IVP\_MULNx16PACKL***

C Syntax:

```
xb_vecNx16 IVP_MULNX16PACKL(  
xb_vecNx16 u, xb_vecNx16 v)
```

- **Output:**
  - VEC register (32 elements \* 16 bits)
- **Inputs:**
  - 2 VEC registers (32 elements \* 16 bits)

# 32-element Multiply-Pack Operation *IVP\_MULNx16PACKL*

Input: 32 elements \* 16 bits



Input: 32 elements \* 16 bits



X

$\text{temp}[31:0] = \text{u1}[15:0] \times \text{v1}[15:0]$   
 $t1 = \text{temp}[15:0]$



Output: 32 elements \* 16 bits

# 32-element Multiply-Pack Operation

## *IVP\_MULNx16PACKL*

### Description:

- The elements of VEC registers **u** and **v** are multiplied to create a 32 bit product **temp**
- The lower **16 bits** of **temp** are stored in each element of the output VEC register **t**

# 32-element Multiply-Pack Predicated Operation

## ***IVP\_MULANx16PACKLT***

C Syntax:

```
IVP_MULANX16PACKLT(xb_vecNx16 t /*inout*/,  
xb_vecNx16 u, xb_vecNx16 v, vboolN m)
```

- **Output:**
  - VEC register (32 elements \* 16 bits)
- **Inputs:**
  - 2 VEC registers (32 elements \* 16 bits)
  - 1 VBOOL register (32 elements \* 1 bit)

# 32-element Multiply-Pack Predicated Operation

## *IVP\_MULANx16PACKLT*

Input: 32 elements \* 16 bits

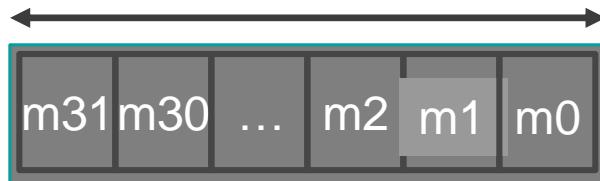


Input: 32 elements \* 16 bits



X

Input: 32 elements \* 1 bit



temp[31:0] = u1[15:0] x v1[15:0]  
t1 += temp[15:0], if m1 == 1  
t1 is not updated, if m1==0



Output: 32 elements \* 16 bits

# 32-element Multiply-Pack Predicated Operation

## *IVP\_MULANx16PACKLT*

### Description:

- The elements of VEC registers **u** and **v** are multiplied to create a 32 bit product **temp**
- A **sum** is computed from the
  - **current value** of the element of output register **t**
  - **lower 16 bits** of the element in **temp**
- This **sum** is stored in each output element of VEC register **t**,
  - if the corresponding bit element of VBOOL vector **m** is 1 (true)
  - otherwise (the corresponding bit element is set to 0 (false)) the corresponding element of vector **t** is not changed

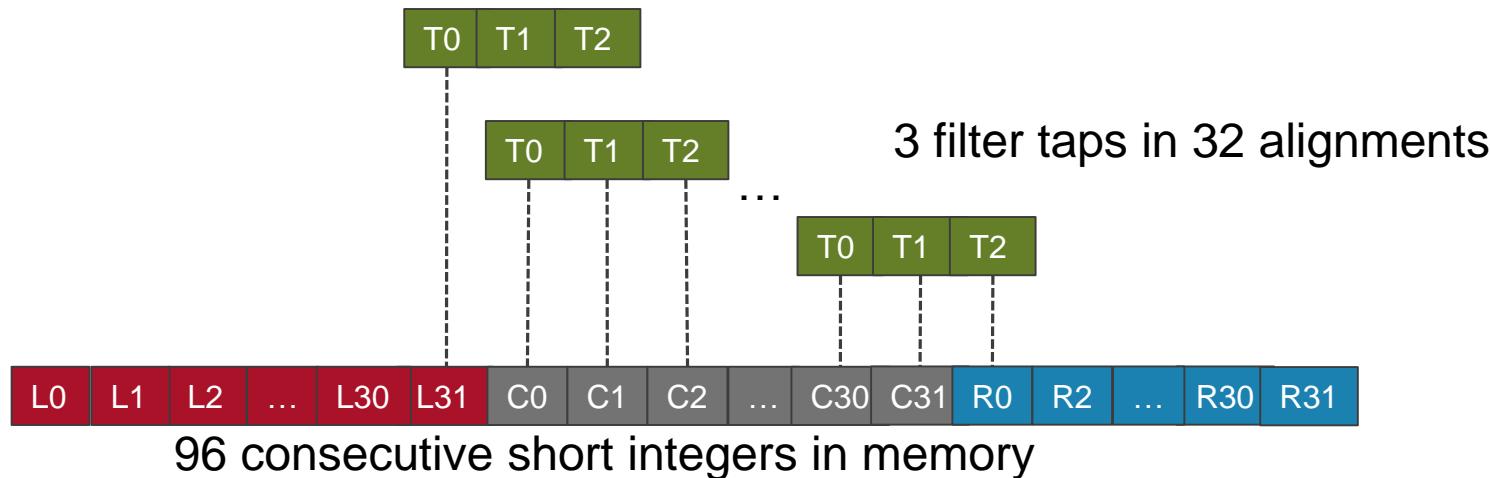
# Usage of the 32-element Multiply-Pack Operations

- IVP\_MULNx16PACKL – outputs the lower 16-bits [15:0]
  - Used when the inputs are **small enough** to not overflow 16 outputs bits
- IVP\_MULNx16PACKQ - outputs 16 bits [30:15]
  - 32-bit output gets shifted **15 bits to the right**
  - Typically used when one or both inputs are in **Q15** format
  - **Q15** format is a fixed point format with **1 sign bit** and **15 bits** to the right of the decimal point
- IVP\_MULNx16PACKP – outputs bits [25:10]
  - 32-bit output gets shifted **10 bits to the right**
  - Typically used when inputs are in **Q5.10** format
  - **Q5.10** format is a fixed point format with **1 sign bit** and **5 bits** to the left and **10 bits** to the right of the decimal point

# Example: Use of Multiply-Pack Operations in 3-tap Filter

- The following example demonstrates the use of the Multiply-Pack operations for a 1 dimensional 3-tap FIR filter
- This example also uses the a **select operation** which we will cover in detail later in this training
- FIR (Finite Impulse Response) filters are commonly used in image processing
- 2 dimensional FIR filters are more common in image processing, but those are often implemented as a sequence of 1D FIR filters as this sometimes reduces
  - The number of needed multiply operations (separable filters)
  - The number of operations per inner loop iterations, which often allows for a better loop schedule

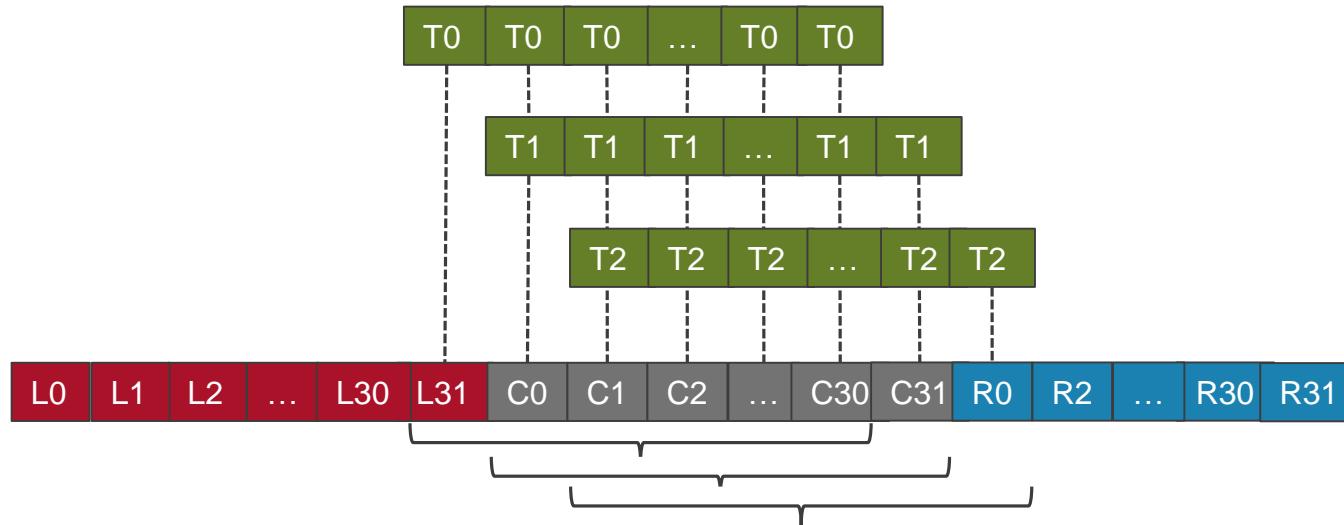
# Example: Use of Multiply-Pack Operations in 3-tap Filter



- Compute filter output by aligning and multiplying the filter mask (T0, T1, T2) with the input data
- Each of the 32 alignments will produce one output value

$$\begin{aligned} \text{Out}[0] &= T0 * L31 + T1 * C0 + T2 * C1 \\ \text{Out}[1] &= T0 * C0 + T1 * C1 + T2 * C2 \\ &\dots \\ \text{Out}[31] &= T0 * C30 + T1 * C31 + T2 * R0 \end{aligned}$$

# Example: Use of Multiply-Pack Operations in 3-tap Filter



- Replicate each filter tap 32 times
- Multiply with corresponding input vector
- The input data is needed in multiple alignments
- Select Operations can be used to create input vectors in those alignments

$$\begin{aligned} \text{Out}[0] &= T0 * L31 + T1 * C0 + T2 * C1 \\ \text{Out}[1] &= T0 * C0 + T1 * C1 + T2 * C2 \\ &\dots \\ \text{Out}[31] &= T0 * C30 + T1 * C31 + T2 * R0 \end{aligned}$$

# Example: Use of Multiply-Pack Operations in 3-tap Filter

The pseudo C code for the filter looks like this:

```
//L, C, and R are vectors holding the input data  
//T0, T1, T2 are vectors with replicated filter tap values  
temp = IVP_SELNX16I(C, L, IVP_SELI_ROTATE_LEFT_1) // slide window left by 1 pixel  
out = T0*temp; // tap T0  
out += T1*C // center tap T1  
temp = IVP_SELNX16I(R, C, IVP_SELI_ROTATE_RIGHT_1) // slide window right by 1 pixel  
out += T2*temp; // tap T2
```

- ◆ This code produces 1 vector output of 32 elements \* 16 bit values in 3 cycles
- ◆ Use of the multiply operator \* will result in a IVP\_MULNX16PACKL operation
- ◆ This is appropriate when the products will not overflow 16 bit accumulation
- ◆ An Alternative is to avoid the use of the multiply operator \* and explicitly use the IVP\_MULNx16PACKP or IVP\_MULNx16PACKQ operation intrinsic
- ◆ The later will result in an implicit shift by 10 or 15 bits before results are accumulated into the output vector

# Pack Operation Categories

- **Wide** Multiply operations are usually followed by a pack operation that **pack** elements from a WVEC register into a VEC register
- **Packing** means moving data from a larger element data type to a smaller one by **discarding** some of the bits

Pack Operations have the following categories

- **Truncate 8 or 16 bits (LSB)** of 1 WVEC register into 1 VEC register
- Pack with **variable shift amount** with from 1 WVEC into 1 VEC register
  - include rounding and saturation
  - have signed and unsigned versions
- Pack **16 bits (LSB) odd/even** elements of 1 WVEC register into 1 VEC register

# Pack Operations truncating to LSB(s) or MSB(s)

The Pack Operations in this category are

- IVP\_PACKMNX48
  - Input: 1 WVEC register (32-elements \* 48 bit)
  - Output: 1 VEC register (32-elements \* 16 bit)
  - output contains **middle (M)** bits [31:16] of the 48 bit input
- IVP\_PACKL2NX24
  - Input: 1 WVEC register (64-elements \* 24 bit)
  - Output: 1 VEC register (64-elements \* 8 bit)
  - output contains the **LSB (L)** bits [7:0] of the 24 bit input
- IVP\_PACKL2NX24\_(0/1)
  - Input: 1 WVEC register (64-elements \* 24 bit)
  - Output: 1 VEC register (32-elements \* 16 bit)
  - output contains the **2 LSBs (L)** bits [15:0] of the 24 bit input of the **even/odd (0/1)** input elements

# Pack Operations with variable shift amounts

The Pack Operations in this category are

- IVP\_PACKVRNX48
  - Input: 1 WVEC register (32-elements \* 48 bit), AR register (24 bit)
  - Output: 1 VEC register (32-elements \* 16 bit)
  - output is rounded and right shifted (by 0 to 23) and saturated to signed 16 bit value
- IVP\_PACKVR(U)2NX24
  - Input: 1 WVEC register (64-elements \* 24 bit), AR register (24 bit)
  - Output: 1 VEC register (64-elements \* 8 bit)
  - output is rounded, right shifted (by 0 to 23) and saturated to signed or unsigned 8 bit value

# Pack Operations with variable shift amounts

The Pack Operations in this category continued ...

- IVP\_PACKVR(**U**)2NX24\_(**0/1**)
  - Input: 1 WVEC register (64-elements \* 24 bit), AR register (24 bit)
  - Output: 1 VEC register (32-elements \* 16 bit) containing **even/odd (0/1)** elements
  - output is rounded, right shifted (by 0 to 23) and saturated to signed or unsigned 16 bit value
- IVP\_PACKVRNR2NX24\_(**0/1**)
  - Input: 1 WVEC register (64-elements \* 24 bit), AR register (24 bit)
  - Output: 1 VEC register (32-elements \* 16 bit) containing **even/odd (0/1)** elements
  - output right shifted (by 0 to 23), but **no rounding (NR)** and **no saturation** is applied

# Multiply Example: Alpha Blend

This example demonstrates the use of the following operations

- **Multiply of AR with and VEC**
- **Pack operation from WVEC to VEC**

Alpha blending creates an output image by computing the

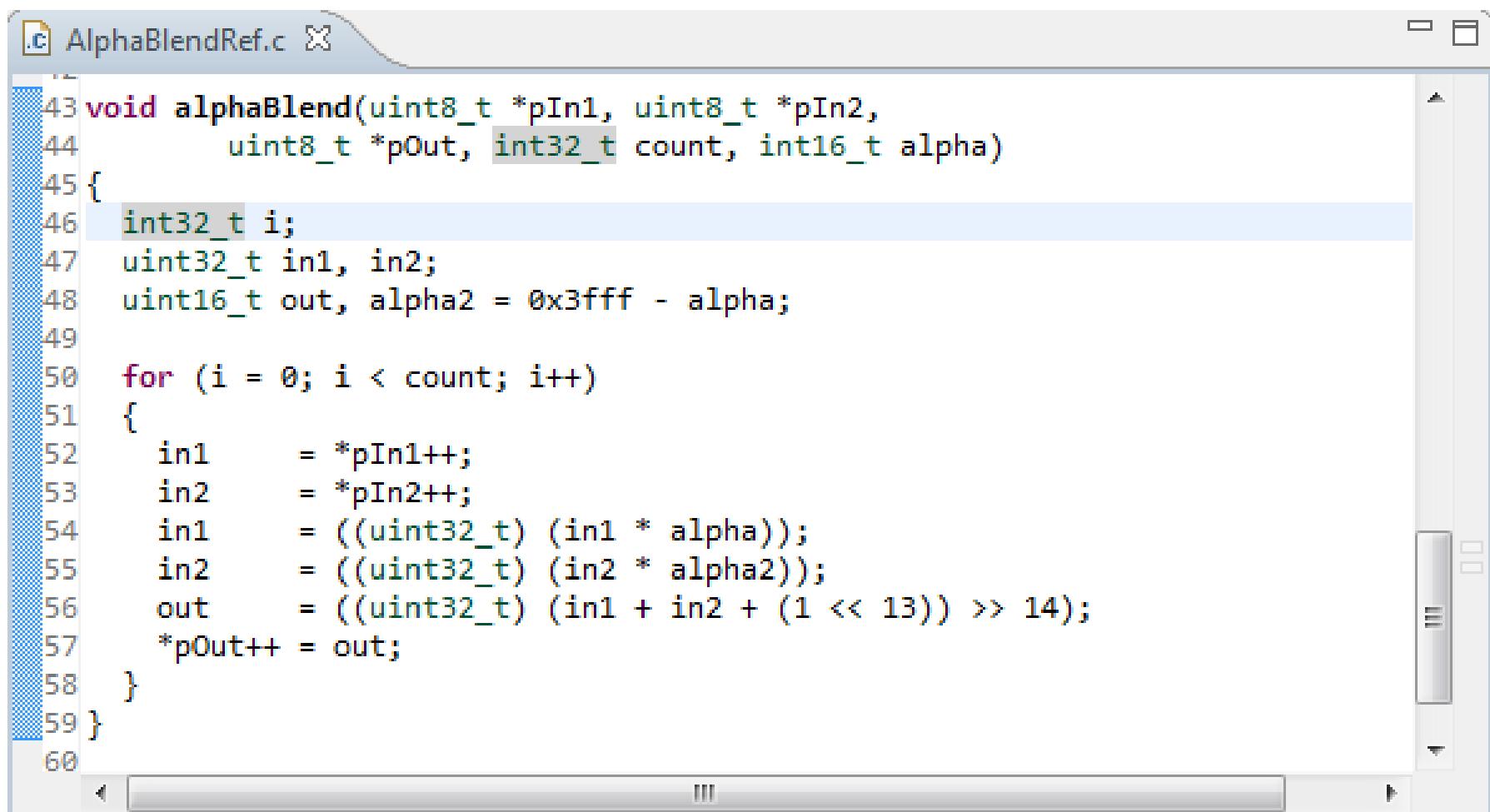
- linear interpolation of two input pixel
- with a blending factor  $\alpha$  between 0 and 1
- independent of the image coordinate  $x$  and  $y$

The formula can be written as:

$$\text{Out}(x,y) = \alpha * \text{in0}(x,y) + (1 - \alpha) * \text{in1}(x,y)$$

# Alpha Blend C Reference Code

- Alpha is encoded with 14 bit
- Alpha = 1 corresponds to 0x3ff
- $1 \ll 13$  correspond to 0.5 rounding value



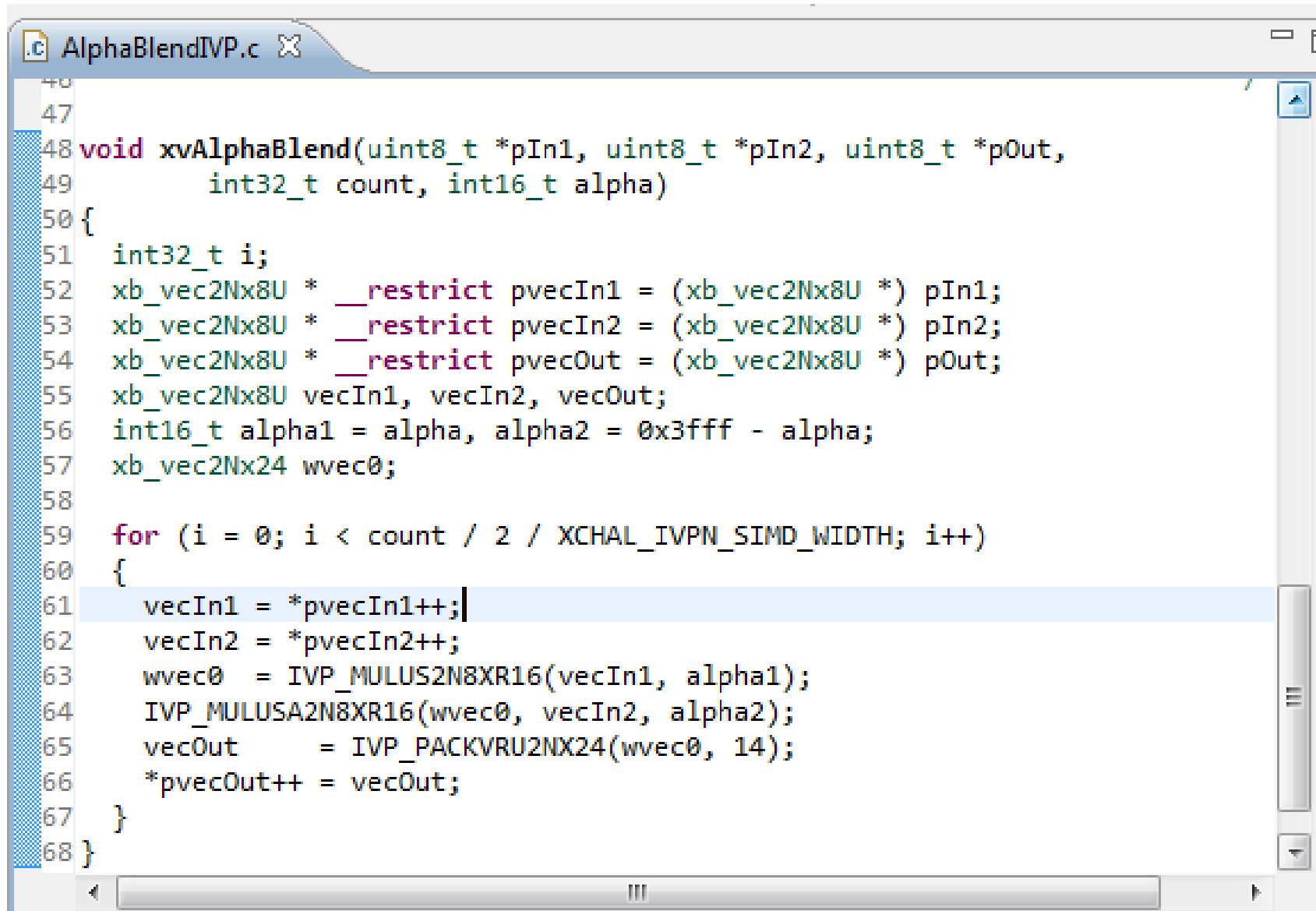
The screenshot shows a code editor window with the title "AlphaBlendRef.c". The code is written in C and defines a function for alpha blending. The code uses uint8\_t for pixel components and int32\_t for intermediate calculations. It calculates the weighted sum of two pixels, taking into account the alpha value and rounding it to 0.5.

```
43 void alphaBlend(uint8_t *pIn1, uint8_t *pIn2,
44                 uint8_t *pOut, int32_t count, int16_t alpha)
45 {
46     int32_t i;
47     uint32_t in1, in2;
48     uint16_t out, alpha2 = 0x3fff - alpha;
49
50     for (i = 0; i < count; i++)
51     {
52         in1      = *pIn1++;
53         in2      = *pIn2++;
54         in1      = ((uint32_t) (in1 * alpha));
55         in2      = ((uint32_t) (in2 * alpha2));
56         out      = ((uint32_t) (in1 + in2 + (1 << 13)) >> 14);
57         *pOut++ = out;
58     }
59 }
```

# Multiply and Pack Operations used in Alpha Blend

- IVP\_MULUSA2N8XR16
  - 64-way 8-bit unsigned vector x 16-bit signed address register (AR)
  - scalar scalar-vector multiply(-**accumulate**)
  - produces wide (24-bit) results
  
- IVP\_PACKVRU2NX24
  - 64-way wide (24-bit) element arithmetic shift
  - Shift amount 0..23 from AR
  - round and saturate to unsigned 8-bit

# Alpha Blend IVP Code



The screenshot shows a software development environment with a code editor window titled "AlphaBlendIVP.c". The code is written in C and implements an alpha blending operation. It uses SIMD instructions from the IVP library to process pairs of vectors from two input buffers and produce a single output buffer.

```
+0
47
48 void xvAlphaBlend(uint8_t *pIn1, uint8_t *pIn2, uint8_t *pOut,
49                     int32_t count, int16_t alpha)
50 {
51     int32_t i;
52     xb_vec2Nx8U * __restrict pvecIn1 = (xb_vec2Nx8U *) pIn1;
53     xb_vec2Nx8U * __restrict pvecIn2 = (xb_vec2Nx8U *) pIn2;
54     xb_vec2Nx8U * __restrict pvecOut = (xb_vec2Nx8U *) pOut;
55     xb_vec2Nx8U vecIn1, vecIn2, vecOut;
56     int16_t alpha1 = alpha, alpha2 = 0x3fff - alpha;
57     xb_vec2Nx24 wvec0;
58
59     for (i = 0; i < count / 2 / XCHAL_IVPN_SIMD_WIDTH; i++)
60     {
61         vecIn1 = *pvecIn1++;
62         vecIn2 = *pvecIn2++;
63         wvec0 = IVP_MULUS2N8XR16(vecIn1, alpha1);
64         IVP_MULUSA2N8XR16(wvec0, vecIn2, alpha2);
65         vecOut = IVP_PACKVRU2NX24(wvec0, 14);
66         *pvecOut++ = vecOut;
67     }
68 }
```

# Alpha Blend IVP Code

- First IVP\_MULUS2N8XR16 sets WVEC to in1\*alpha
- Second IVP\_MULUSA2N8XR16 computes in1\*alpha + in2\*alpha2
- IVP\_PACKVRU2NX24 shifts and rounds the output to an 8 bit value

# Summary of Multiply and Pack Operations

Let's look at the advantages and disadvantages of Wide and Narrow Multiply Operations

- The Multiply-Pack operations
  - are 32-element operations
  - combine the pack with the multiply operation
  - accumulate into a 16 bit register
- Advantages of Wide Multiply Operations
  - They often allow **64**-elements operations
  - They allow accumulation with more **guard bits**
    - **24** bits for 64-element operations
    - **48** bits for 32-element operations
- Disadvantage of Wide Multiply Operations
  - They require a **separate pack operation** to move the result from a WVEC to a VEC register before the result can be stored to memory

# Summary of Multiply and Pack Operations

- In general we expect Wide Multiply Operations to have **higher performance** and **higher precision**
- However, using Wide Multiply Operations requires
  - separate Pack Operation(s)
  - in some cases even Select Operations for data interleaving
- This is a tradeoff and the **choice** of multiply operation might be **different** for each application kernel



# Vector Select Operations

**Submodule**    **10-3**

**Revision**              **1.0**

**Version**              **7.4**

**Estimate Time**

- **Lecture**              **1 hour**
- **Lab**

# Data Reorganization

- Many **image processing tasks** require accesses to image data with **arbitrary alignment** (pointer address is not a multiple of vector size)
- **2D Filters** require access to the image data with many different pixel **offsets** in the **vertical** and **horizontal** directions
- **Image Interpolation** often requires **de-interleaving** of image data into odd and even pixels
- **Look Up Tables** require **random access** to the values in a table depending on the **pixel value**
- In this module we will introduce **operations that facilitate** these types of accesses to image data

# Shuffle and Select Operations

- **Shuffle Operations** have **1 VEC** register input for table values, **1 VEC** register input for select indices and **1 VEC** register output
- **Select Operations** have **2 VEC** register inputs for table values, **1 VEC** register input for select indices and **1 VEC** register output.
- **Double Select Operations** have **2 VEC** register inputs for table values, **1 VEC** register input for select indices and **2 VEC** register outputs.
- **Immediate** versions of the above operations do **not have a VEC** register input for select indices. Instead they take an **immediate** operand which indicates which hard coded **access pattern** to use.
- **Repeat** Operations allow replication of specific input vector elements into the elements of an output vector
- In this module we will focus in **detail** on all these types of operations

# Shuffle Operations

- We will focus on the Shuffle Operations first
- Shuffle Operations have **1 VEC** register input
  - Select Operations have **2 VEC** register inputs
- Shuffle Operations are **special cases** of the Select Operations
  - They use **fewer registers** which helps to
    - reduce register pressure
    - save power
- We will now focus on different **versions** of the Shuffle Operation

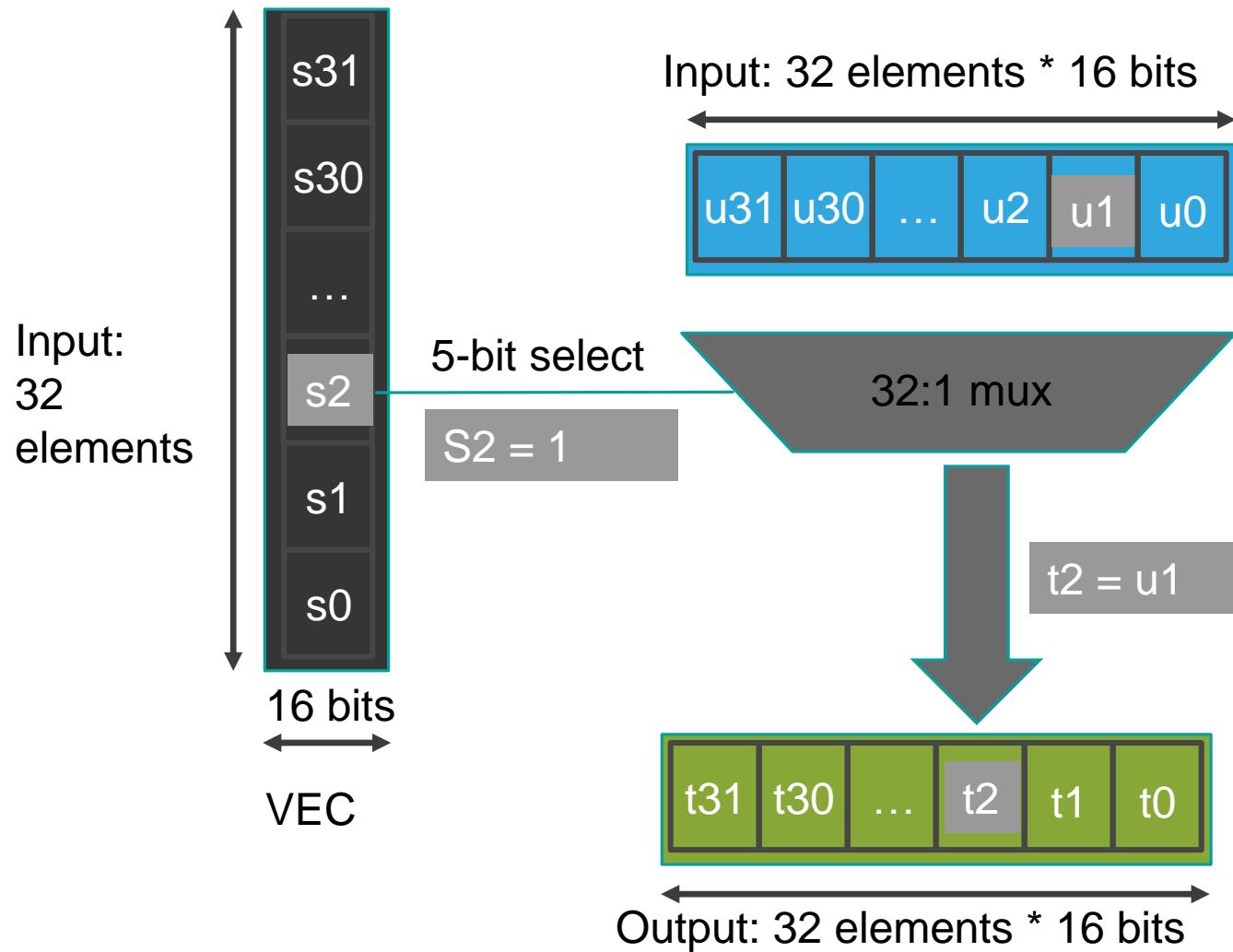
# 32-element Shuffle Operation IVP\_SHFLNX16

C Syntax:

```
xb_vecNx16 IVP_SHFLNX16(  
    xb_vecNx16 u, xb_vecNx16 s)
```

- **Output:**
  - 1 VEC register (32 elements \* 16 bits)
- **Inputs:**
  - 1 VEC registers (32 elements \* 16 bits) of table values
  - 1 VEC registers (32 elements \* 16 bits) of select indices

# 32-element Shuffle Operation IVP\_SHFLNX16



# 32-element Shuffle Operation IVP\_SHFLNX16

## Description:

- Each element of VEC register **s** contains a 5-bit **index**
  - bits 5 to 15 are ignored
- Each **index** selects one of the 32 elements from the input vector **u** as the output element of vector **t**
- Each output element is generated using the select element at the corresponding position in the select vector.

# Examples for Shuffle Operation IVP\_SHFLNX16

## Example: Repeat Element 0

- Input:
  - VEC register s {s31, s30, s2, s1, s0} = {0, 0, ..., 0, 0}
  - VEC register u {u31, u30, ..., u2, u1, u0}
- Output: {t31, t30, ..., t2, t1, t0} = {u0, u0, ..., u0, u0, u0}

## Example: Duplicate Even Elements

- Input:
  - VEC register s {s31, s30, s2, s1, s0} = {30, 30, ..., 2, 2, 0, 0}
  - VEC register u {u31, u30, ..., u2, u1, u0}
- Output: {t31, t30, ..., t2, t1, t0} = {u30, u30, ..., u2, u2, u0, u0}

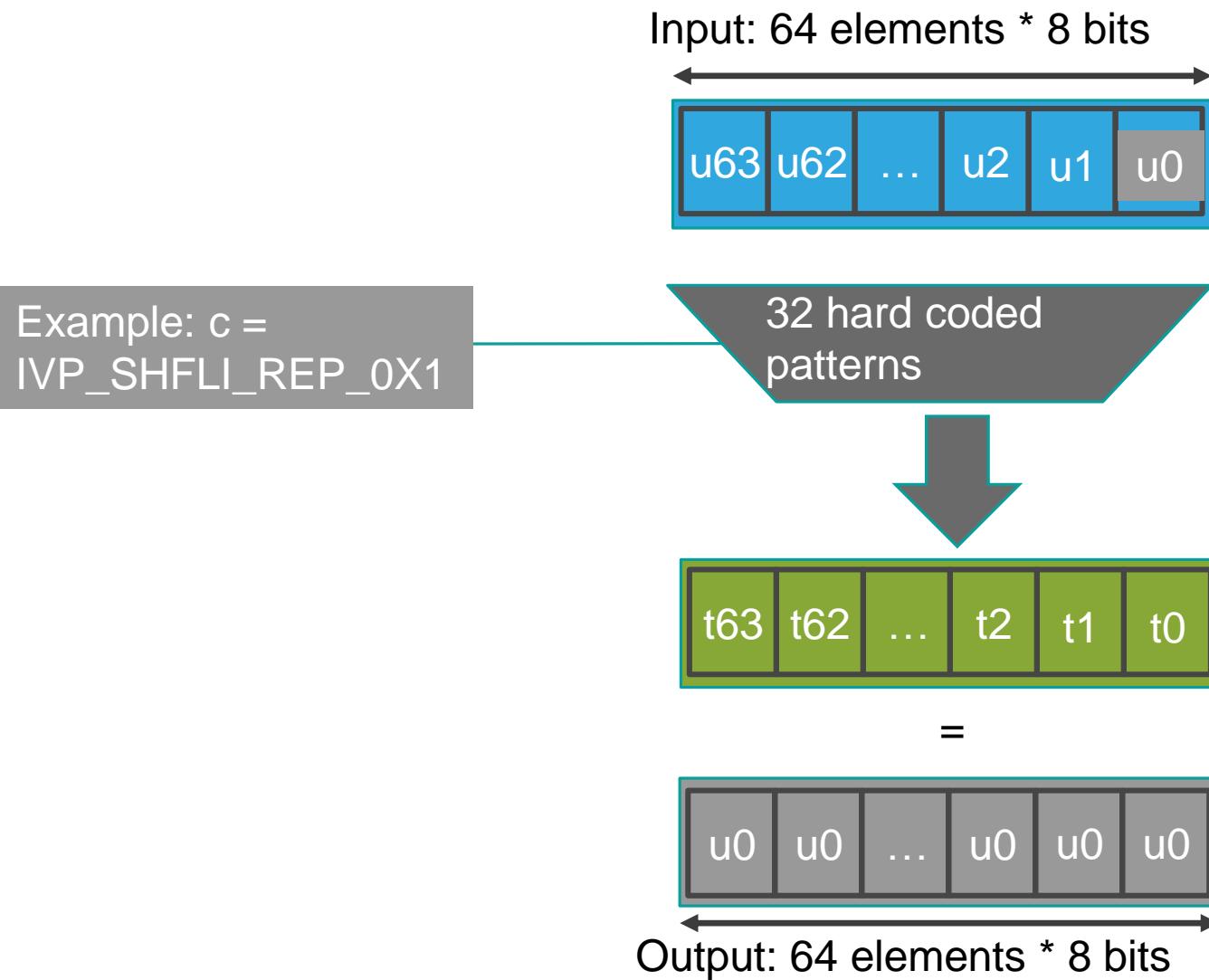
# **64-element Shuffle immediate Operation IVP\_SHFL2NX8I**

**C Syntax:**

**xb\_vec2Nx8 IVP\_SHFL2NX8I(xb\_vec2Nx8 b,  
immediate c) Output:**

- 1 VEC register (64 elements \* 8 bits)
- **Inputs:**
  - 1 VEC register (64 elements \* 8 bits) of table indices
  - 1 Immediate value c to chose a select pattern

# 64-element Shuffle immediate Operation IVP\_SHFL2NX8I



# 64-element Shuffle immediate Operation

## IVP\_SHFL2NX8I

### Description:

- Choose from **32 hard coded access patterns** with immediate value **c**
- If **c** is out of the number range of **0-31** a value of **0 is used instead**
- A 64-elements output vector **t** is chosen from a combination of elements in the 64-element input vector **u**
- Examples of hard coded access patterns are
  - **replicate** specific elements
  - **reverse element order**
  - **duplicate even or odd elements**
- A table of supported immediate values c and a description of their meaning can be found in the ISA-HTML of this operation

# Select Operations

- We will now focus on Select Operations
- Select Operations have
  - **2 VEC** input registers for a table elements
  - **1 VEC** input register for select indices
  - **1 VEC** output register for **normal** Select Operations
  - **2 VEC** output register for **Double** Select Operations
- We also support a **64-element Double Select Immediate** Operation
- We will now focus on different **versions** of the Select Operation

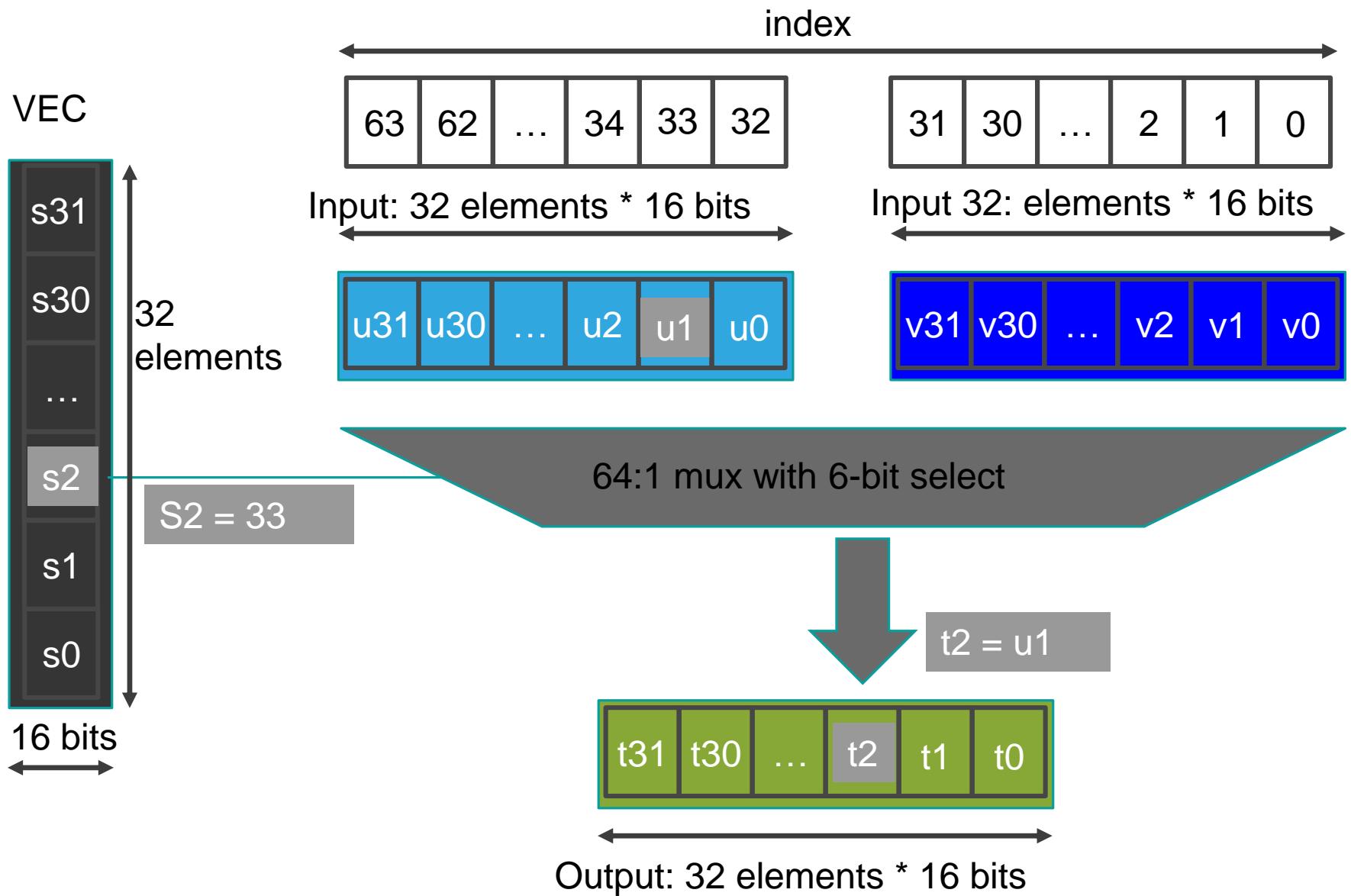
# 32-element Select Operation IVP\_SELNX16

C Syntax:

```
xb_vecNx16 IVP_SELNX16(  
    xb_vecNx16 u, xb_vecNx16 v, xb_vecNx16 s)
```

- **Output:**
  - 1 VEC register (32 elements \* 16 bits)
- **Inputs:**
  - 2 VEC registers (32 elements \* 16 bits each) of table values
  - 1 VEC register (32 elements \* 16 bits) of select indices

# 32-element Select Operation IVP\_SELNX16



# 32-element Select Operation IVP\_SELNX16

## Description:

- Only Lower **6** bits of the VEC register s are used to as an **index**
  - bits 6 to 15 are ignored
- The **index** selects one of the 64 input elements
- The elements of the input VEC registers u and v are combined by putting them side by side
  - The elements of VEC register **u** correspond to indices **32 to 63**
  - The elements of VEC register **v** correspond to indices **0 to 31**
- The 32 elements of the VEC register s contain 32 indices
  - one for each output element
- Each output element is generated using the select element at the corresponding position in the select vector.

# 32-element Double Select Operation

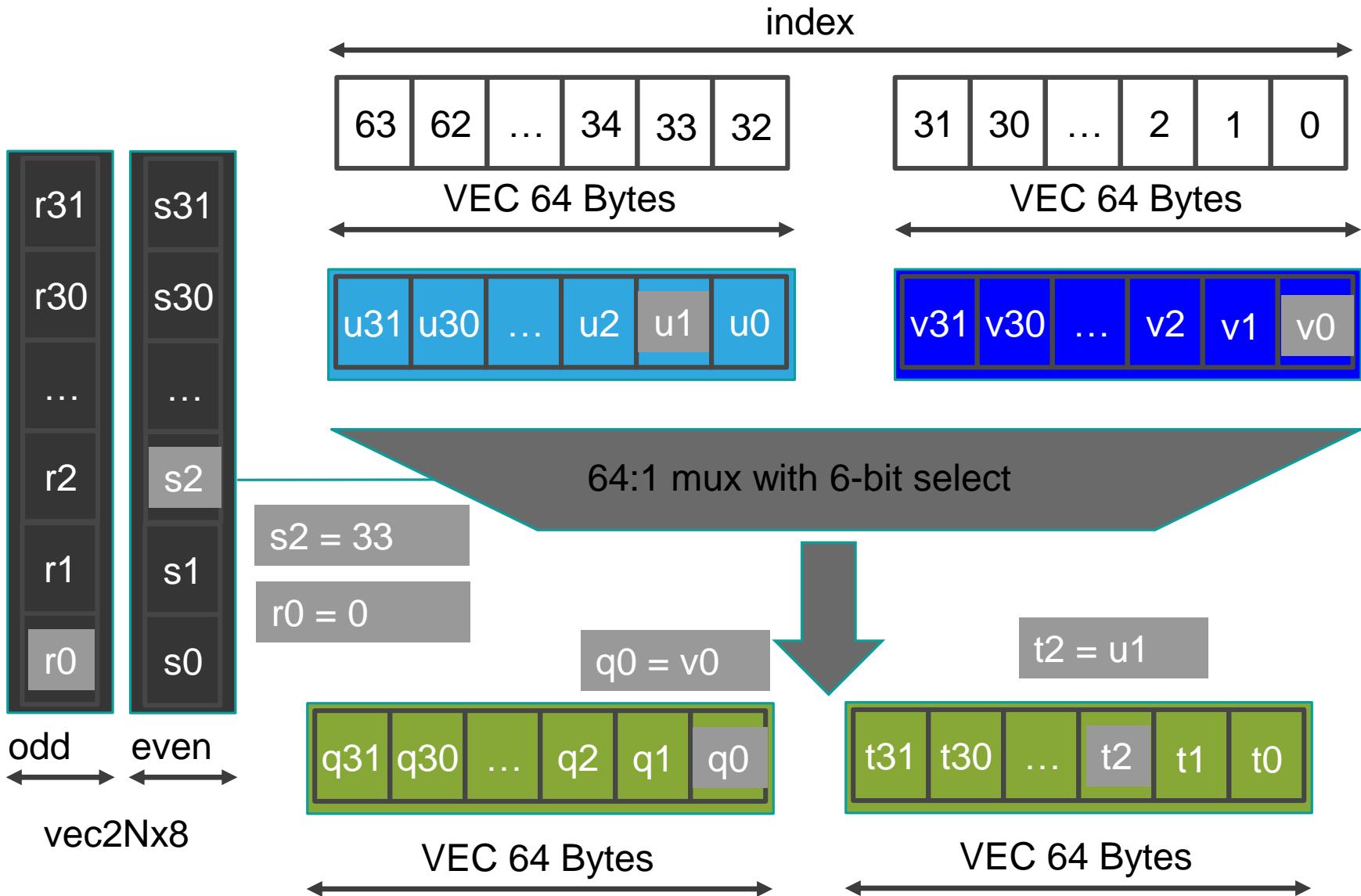
## IVP\_DSELNX16

C Syntax:

```
IVP_DSELNX16(  
    xb_vecNx16 q /*out*/, xb_vecNx16 t /*out*/,  
    xb_vecNx16 u, xb_vecNx16 v, xb_vec2Nx8 r)
```

- **Output:**
  - 2 VEC registers (32 elements \* 16 bits each)
- **Inputs:**
  - 2 VEC registers (32 elements \* 16 bits each) of table values
  - 1 VEC register (64 elements \* 8 bits) of select indices

# 32-element Double Select Operation IVP\_DSELNX16



# 32-element Double Select Operation

## IVP\_DSELNX16

### Description:

- Only Lower **6** bits of the VEC register **s** are used to create an **index**
  - bits 6 and 7 are ignored
- The **index** selects one of the 64 input elements
- The elements of the input VEC registers **u** and **v** are combined by putting them side by side
  - The elements of VEC register **u** correspond to index **32 to 63**
  - The elements of VEC register **v** correspond to index **0 to 31**

# 32-element Double Select Operation

## IVP\_DSELNX16

Description continued:

- The output elements of VEC register **q** are controlled by **odd** indices of VEC register r
- The output elements of VEC register **t** are controlled by **even** indices of VEC register r

# Repeat Operations

**Repeat** Operations allow replication of specific input vector elements into the elements of an output vector

- **IVP\_REPNX16:** Replicate one of 32 input vector elements into all elements of the output vector register
- **IVP\_REP2NX8:** Replicate one of 64 input vector elements into all elements of the output vector register
- **IVP\_SELSNX16:** Select one of 32 input vector elements into element 0 of the output vector register

Please refer to the online ISA HTML for details on these operations

# Usage of Select Operations for Lookup Tables

The Select operation is useful for table lookups

- 64-8b lookups from a 128 entry table (SEL2NX8)
- 64-16b lookups from a 64 entry table (DSELNX16)

# Shuffle and Select Immediate Operations in Slots 0,2 and 4

- IVP\_SHFL2NX8I supports **32** immediate values (shuffle patterns) and can only be issued in **slot3**
  - IVP\_SHFL2NX8I\_S{0,2,4} are specialized versions that can be issued in slot0, slot2 or slot4
  - This specialized version only supports 2 immediate values (select patterns)
  - See **ISA HTML** for details
- IVP\_SEL2NX8I supports **73** immediate values (select patterns) and can only be issued in **slot3**
  - IVP\_SEL2NX8I \_S{0,2,4} are specialized versions that can be issued in slot0, slot2 or slot4
  - This specialized version only supports 27 immediate values (select patterns)
  - See **ISA HTML** for details
- These operations can **accelerate select/shuffle bound kernels**
- The **user** is encouraged to just use the **general** version of these two operations
- Compiler will **opportunistically** insert the special operations based on the user provided **immediate** pattern

# Summary of Data Reorganization Operations

- We have discussed a variety of operations for data reorganization
  - **Shuffle Operations**
  - **Selects and Double Select Operations**
  - **Operation variants with immediate inputs**
- A **Shuffle** is preferred if only a single input vector register is needed
- A **Double Select** might be preferred if **multiple output vectors** are needed
- Check whether the **needed select pattern** is already covered by an **immediate** operation version

# Summary

In this module, you got an overview of the Vision P6 operations details. You learned about

- Load/Store,
- Multiply,
- Select
- Reduction Operations

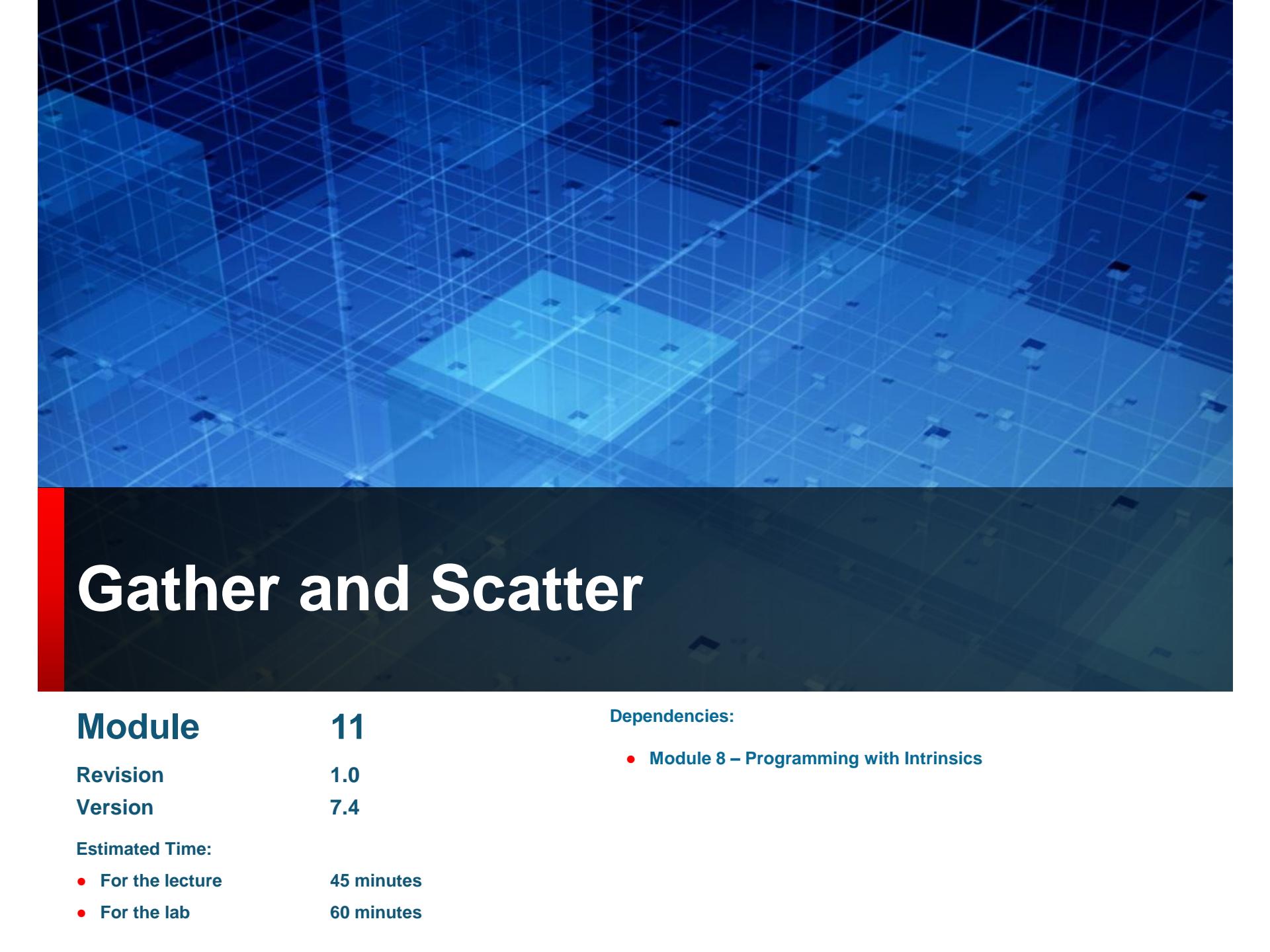
Knowledge about the details of these type of operations will enable you to look at inner loops of code and identify alternative operations sequences that might accomplish the same task with a smaller number of operations.

In the next two modules we will continue our deep dive into the Vision P6 ISA, but will focus on operations with a specific purpose.

# Quiz

1. Vision P6 can issue the following operations in one instruction
  - a) 2 normal vector load operations
  - b) 2 aligning vector load operations
  - c) 1 aligning vector load operation that does not require an aligning register (no priming)
  - d) All of the above ...
2. A double select operation
  - a) Has one vector output operand
  - b) Has two vector output operands
  - c) Has two scalar output operands
3. The operation IVP\_MUL2N8XR16
  - a) Multiplies 1 vector of 2N 8-bit elements with 16 bit value from an AR register
  - b) Multiplies 2 vectors of N 8-bit elements and produces a vector of N 16-bit outputs
  - c) Multiplies 2 vectors of 2N 8-bit elements with each other, produces a reduction sum of the output vector and stores the 16 bit result into an AR register

Answers: 1d, 2b, 3a



# Gather and Scatter

**Module** **11**

**Revision** **1.0**

**Version** **7.4**

**Estimated Time:**

- For the lecture **45 minutes**
- For the lab **60 minutes**

**Dependencies:**

- **Module 8 – Programming with Intrinsics**

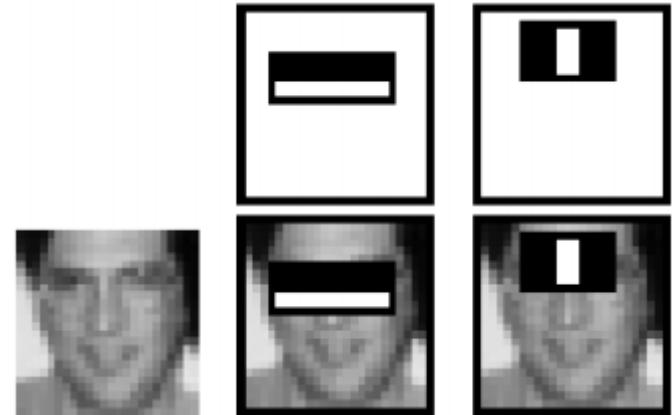
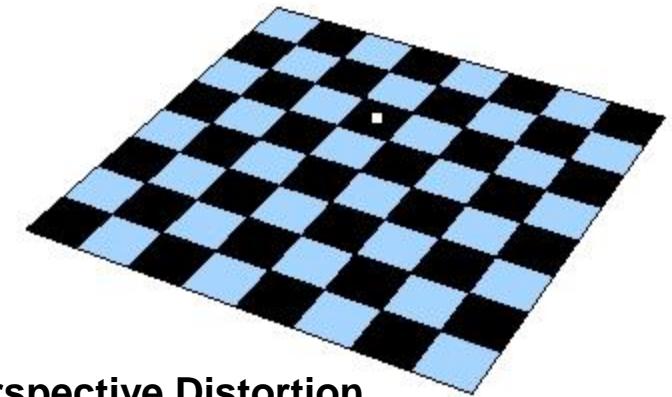
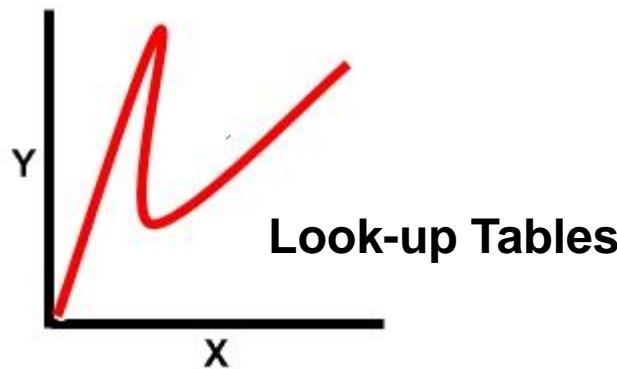
# Module Objectives

In this module, you will get an overview of the Vision P6 Gather/Scatter Engine.

You will learn about

- The gather/scatter engine and registers
- How the throughput of gather/scatter operations depends on access patterns
- Application examples that are accelerated by this engine

# Data for image and vision algorithms is often not SIMD-friendly



# Why do we need Gather and Scatter Operations?

- Vision P6 is a **wide vector** machine with a vector size of **64** elements(8-bit).
- **Load** operations are suitable when the same operation is applied along a vector of **adjacent** pixels.
- Computer Vision Algorithms often process **feature points** around **small image blocks** that are not multiples of the P6 **vector length**.

In order to exploit SIMD parallelism for these algorithms

- load operations are needed that can pick up data elements from these **arbitrarily sized blocks** and
- fill **all** vector elements with useful data to process.

Algorithms that benefit from these operations are

- **Haar-Cascade** for Face Detection
- Calculation of **Feature Descriptors**
- **Image Warping and Perspective Transformation**
- Look Up Table Operations (**LUT**)

# Introduction to Gather/Scatter Operations

Gather/Scatter Operations are **generalized** load and store operations

## Gather Operations

- **Load** data elements from memory into a VEC register
- The data elements do not have to be **contiguous** in memory
- They can be at arbitrary locations (**scattered**) in local memory

## Scatter Operations

- **Store** data elements from a VEC register into local memory
- The data elements do not have to be **contiguous** in memory
- They can be at arbitrary locations (**scattered**) in local memory

# How does Gather/Scatter work Conceptually?

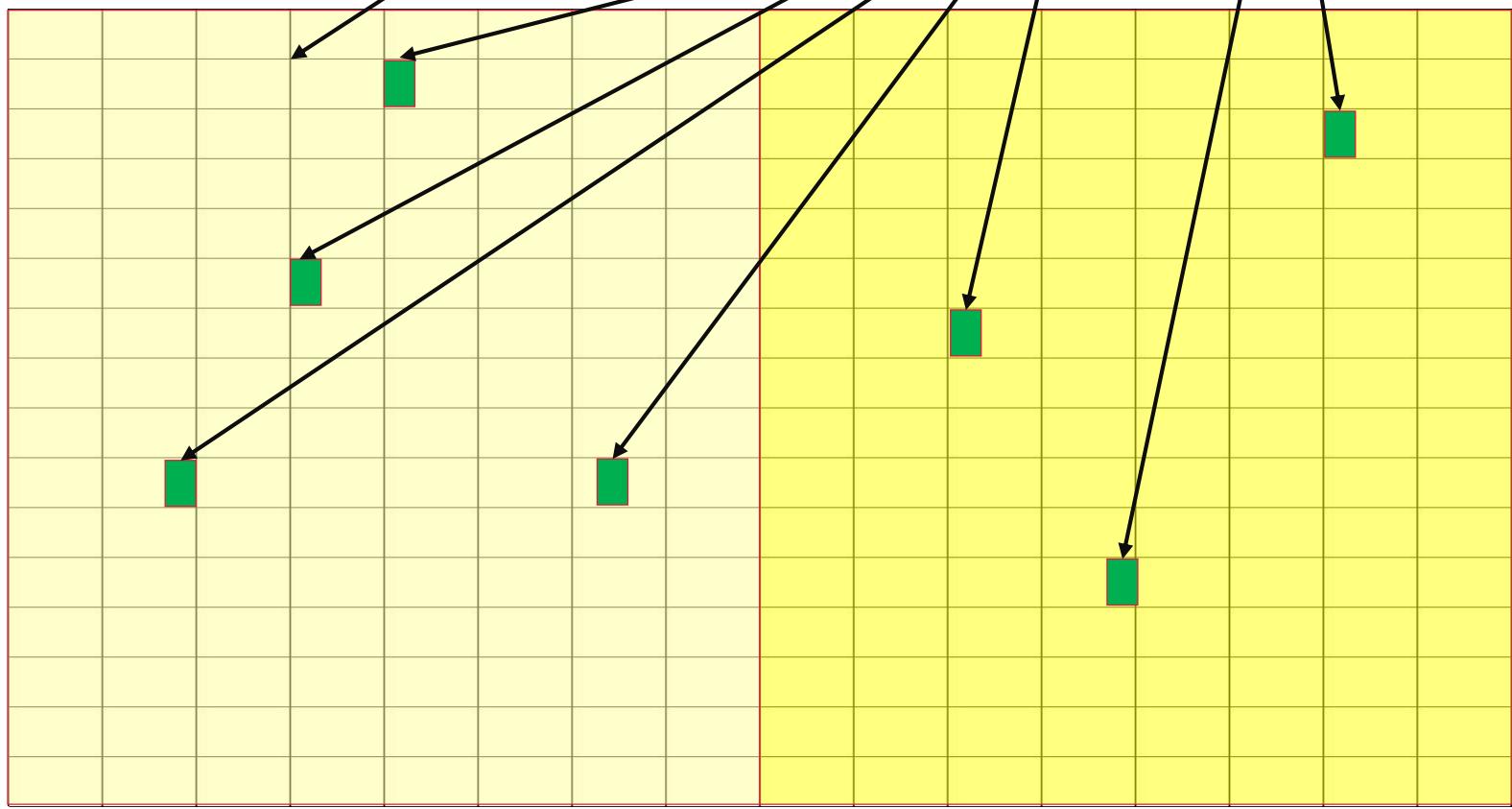
AR with Base Address:

0x3ffc1000

VEC Register with Address Offsets:



Gather/Scatter Register:



# Overview of Gather Operations

Gather Operations are composed of two related operations

## 1. Gather Address

- Inputs: AR register with base address, VEC register with address offsets
- Output: Gather/Scatter (GSR) register with loaded data
- This operation will start the data gathering stage of loading the data from local memory into the GSR register

## 2. Gather Data

- Input: GSR register with loaded data
- Output: VEC register with loaded data
- This operation moves data from a GSR register to a VEC register
- It will **stall** the processor pipeline until all data elements are loaded from memory into the GSR register

# Why are Gather Operations partitioned into GatherA / GatherD stages?

Gather Operations complete with a **very long latency**

- It takes a minimum of **6** cycles to complete a Gather Operation
- In addition there is a variable added latency from **1 to 32** cycles for a total latency of **7 to 38** cycles (could even be longer when you account for conflicts with load/store unit).
- Elements distributed across **different sub-banks** can be read in **parallel**
- Elements distributed across **multiple rows of the same sub-bank** have to be read **sequentially** (examples follow later)
- The GatherD/GatherA pairs allow the compiler to schedule the data **load** and **use** many cycles apart from each other
- Often programmers have **knowledge of access pattern** during code execution. These might result in expected latencies that can be passed to the compiler as **hints** (more on this later)
- Vision P6 has **4** Gather Registers, that allow pipelining of multiple Gather Operations

# 16-bit Gather **Address** Operation: IVP\_GATHERANX16

This Gather operation initiates the gathering/loading) of **32 16-bit** data elements

**Operation:** extern xb\_gsr IVP\_GATHERANX16(const short \* b,  
xb\_vecNx16U c);

- In: AR register with base address **b**
- In: VEC register **c** with 32 elements of 16-bit unsigned address offsets
- Out: GSR register with 32 loaded 16-bit data values

# 16-bit Gather **Data** Operation: IVP\_GATHERDNX16

Moves 32 16-bit elements from a GSR register to a VEC register.

It **stalls** the pipeline if loads to some elements of the GSR are still in flight.

**Operation:** extern xb\_vecNx16 IVP\_GATHERDNX16(xb\_gsr b);

- In: GSR register **b** with 32 loaded 16-bit data values
- Out: VEC register with 32 loaded 16-bit data values

Note, there is a **proto IVP\_GATHERNX16** that bundles the operations

1. IVP\_GATHERANX16;
2. IVP\_GATHERDNX16;

# Variable Delay Proto: IVP\_GATHERANX16\_V

**Proto:** extern xb\_gsr IVP\_GATHERANX16\_V(const short \* b, xb\_vecNx16U c, immed delay);

- In: AR register with base address **b**
- In: VEC register **c** with 32 elements of 16-bit unsigned address offsets
- Out: GSR register with 32 loaded 16-bit data values
- Delay 1-32 corresponding to the estimating additional delay due to sub-bank conflicts (more on this in later slides)

We mentioned earlier that the required latency between GatherA/GatherD is generally quite long:

- At least 6(fixed)+1(variable up to 32) cycles
- Compiler will schedule GatherA/GatherD operations 7 cycles apart
- The variable delay proto allows you to insert a delay different from that default and the compiler will attempt to schedule GatherA/GatherD 6+delay cycles apart.
- Proto exists for many operation flavors: IVP\_GATHERAN\_2X32T\_V, IVP\_GATHERANX8U\_V, etc.

# Simple 16-bit Gather Example

This simple example

- loads contiguous block of 16-bit data elements
- could be implemented with vector load and store operations
- illustrates how to use gather/scatter and calculate the offset vector

```
int xvGather_16bit(int16_t *a, int16_t *d)
{
    int i = 0;
    int start, stop;
    xb_vecNx16 vecIdx = IVP_SEQNX16() << 1;   ← creates 32 address offsets 0, 2, 4, ..., 60, 62
    xb_vecNx16 vec_0;
    xb_vecNx16 *pd = (xb_vecNx16 *) d;

    printf("----- Initial offsets ----- \n");
    vec_print16(vecIdx);

    TIME_STAMP(start);
    for (i = 0; i < 512/32; i++)
    {
        vec_0      = IVP_GATHERNX16(a, vecIdx);
        pd[i]     = vec_0;
        vecIdx += (xb_vecNx16) 64;
    }
    TIME_STAMP(stop);
```

The Loop iteration count is 512/32  
← since we gather 32 elements at a time  
← combines GatherA/GatherD operations  
← creates address offsets 64, 66, 68, ..., 124, 126  
that are needed for next iteration

# Gather Operations for 8 and 32 bit data

## Gather Operations for 8 bit data

- 16 bit address offsets are provided in the GatherA type operations
- This limits the number of gather elements to 32 (not 64)
- Loading the 64 elements of an 8-bit vector requires 2 gather operations

## Gather Operations for 32 bit data

- The width of the GSR and VEC register limits the number of gather elements to 16

# Simple 8-bit Gather Example

This simple example

- loads contiguous block of 8-bit data elements
- could be implemented with vector load and store operations
- illustrates how to use gather/scatter and calculate the offset vector

```
int xvGather_8bit(int8_t *a, int8_t *d)
{
    int i = 0;
    xb_vecNx16 vecIdx;
    xb_vec2Nx8 dvec_0;
    xb_vec2Nx8 *pd = (xb_vec2Nx8 *) d;
    xb_gsr _gs0, _gs1;

    vecIdx = IVP_SEQNX16(); ← creates 32 address offsets 0, 1, 2, ..., 30, 31
    for (i = 0; i < 512/64; i++) ← The Loop iteration count is 512/64
    {
        _gs0      = IVP_GATHERANX8U(a, vecIdx); ← since we gather 32 elements at a time
        vecIdx += (xb_vecNx16) 32;
        _gs1      = IVP_GATHERANX8U(a, vecIdx); } ← Two separate GatherA operations are issued
                                                    to each gather 32 8-bit data elements.
        dvec_0   = IVP_GATHERD2NX8_L(_gs0); } ← Two separate GatherD operations are issued
        IVP_GATHERD2NX8_H(dvec_0, _gs1); } ← to merge the results from the 2 GSR
                                                registers into one 64-element 8-bit vector
        pd[i] = dvec_0;
        vecIdx += (xb_vecNx16) 32;
    }
```

# Overview of Scatter Operations

## Scatter Operation

- Input: VEC register with data elements to store
- Input: AR register with base address, VEC register with address offsets
- Output: Written memory locations in local memory
- The scatter operation is conceptually simpler than the gather -not broken up into stages (like GatherA/GatherD)
- 2 Scatter registers help to pipeline scatter operations
- If memory **synchronization** is required for subsequent gather or load operations a **scatter wait** (`void IVP_SCATTERW(void)`) can be issued

# Simple 32-bit Scatter Example

This simple example

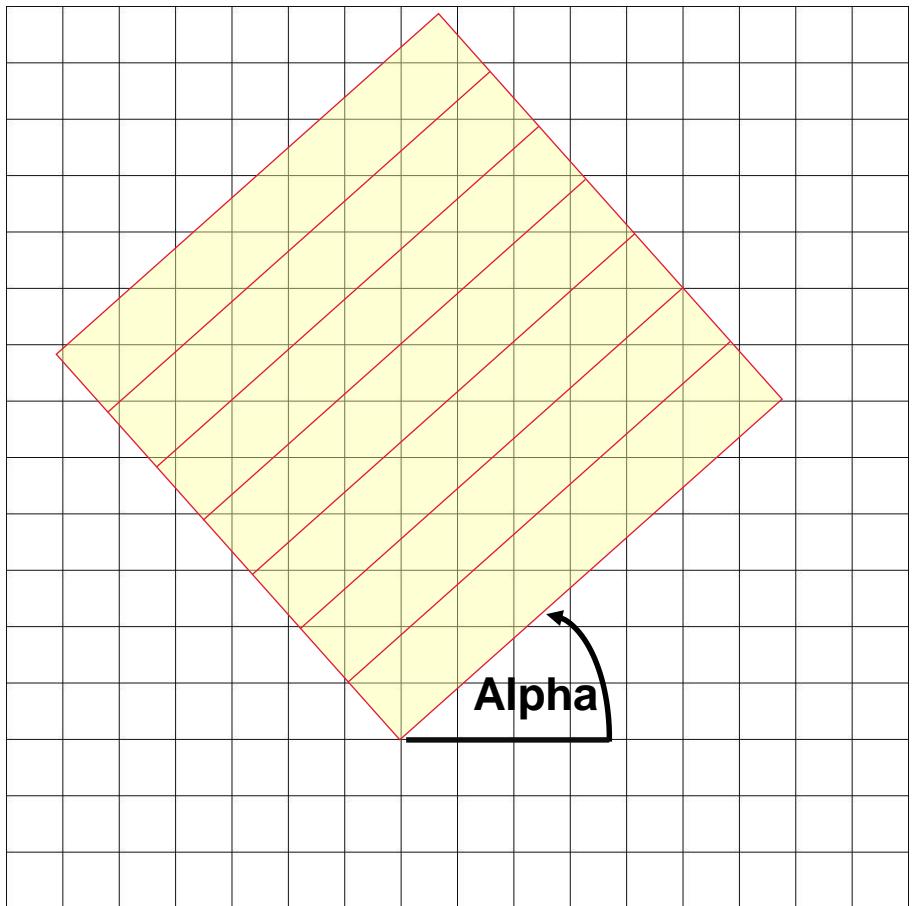
- stores contiguous 32-bit data elements
- could be implemented with vector load and store operations
- illustrates how to use gather/scatter and calculate the offset vector

```
int xvScatter_32bit(int32_t *a, int32_t *d)
{
    int i = 0;
    int start, stop;
    xb_vecN_2x32v vecIdx;
    xb_vecN_2x32v dvec_0;
    xb_vecN_2x32v *pvecDst = (xb_vecN_2x32v *) d;
    xb_vecN_2x32v *pvecSrc = (xb_vecN_2x32v *) a;

    vecIdx = IVP_SEQN_2X32() << 2;           ← creates 16 address offsets 0, 4, 8, ..., 56, 60
    vec_print32(vecIdx);

    for (i = 0; i < 512/16; i++)           ← The Loop iteration count is 512/16
    {                                       since we scatter 16 elements at a time
        dvec_0 = pvecSrc[i];
        IVP_SCATTERN_2X32(dvec_0, pvecDst, vecIdx); ← Scatters the 16 32-bit elements in
        vecIdx += (xb_vecN_2x32v) 64;                  dvec_0 to memory
    }
}
```

# Example: Load rotated rectangle from memory



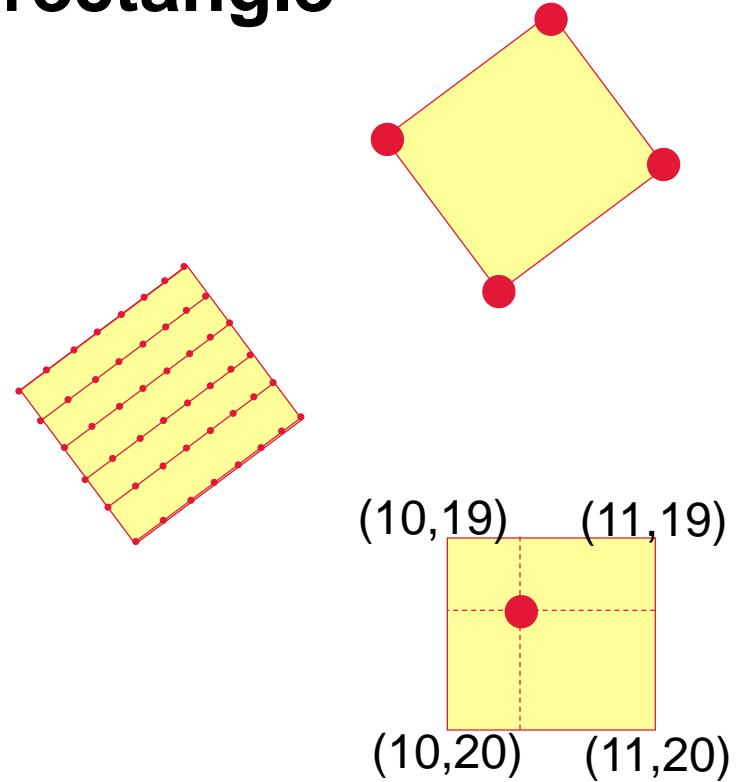
Assume a rectangle rotated by the angle **Alpha**. Assume that

- Rectangle is in local memory
- You want to load data along the **red lines**
- Grid lines correspond to memory location

This would be difficult with normal vector load operations, but a gather operation is well suited.

# Procedure for loading rotated rectangle

1. Calculate the image coordinates of the four rectangle **corners**.
2. Interpolate the interior coordinates by dividing into a number of desired **rows** and **columns** for the un-rotated rectangle (e.g., 32 x 32)
3. Convert the above fractional coordinates to **integer** coordinates by **rounding** to nearest integers
4. Convert integer coordinates to address of set address
5. Use the image base address and offset (row) vector for the gather operation



**offset = y-coord \* image\_stride + x-coord**

`vec_0 = IVP_GATHERNX16(a, vecIdx);`

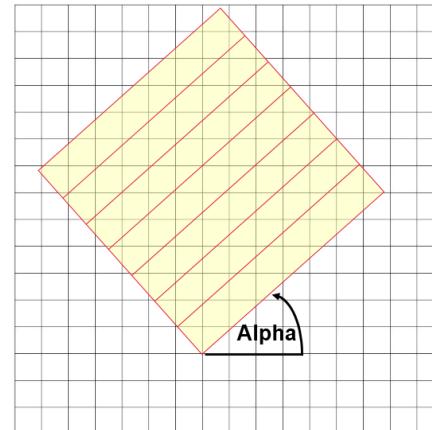
one row of rectangle      start of image      vector of offsets

# Performance of rotated rectangle example

Angle Alpha (Deg.)	Cycles	CPP	Throughput in elements per cycle
0	98	0.10	10.45
10	106	0.10	9.66
20	120	0.12	8.53
30	132	0.13	7.76
40	159	0.16	6.44
50	206	0.20	4.97
60	268	0.26	3.82
70	349	0.34	2.93
80	450	0.44	2.28
90	518	0.51	1.98

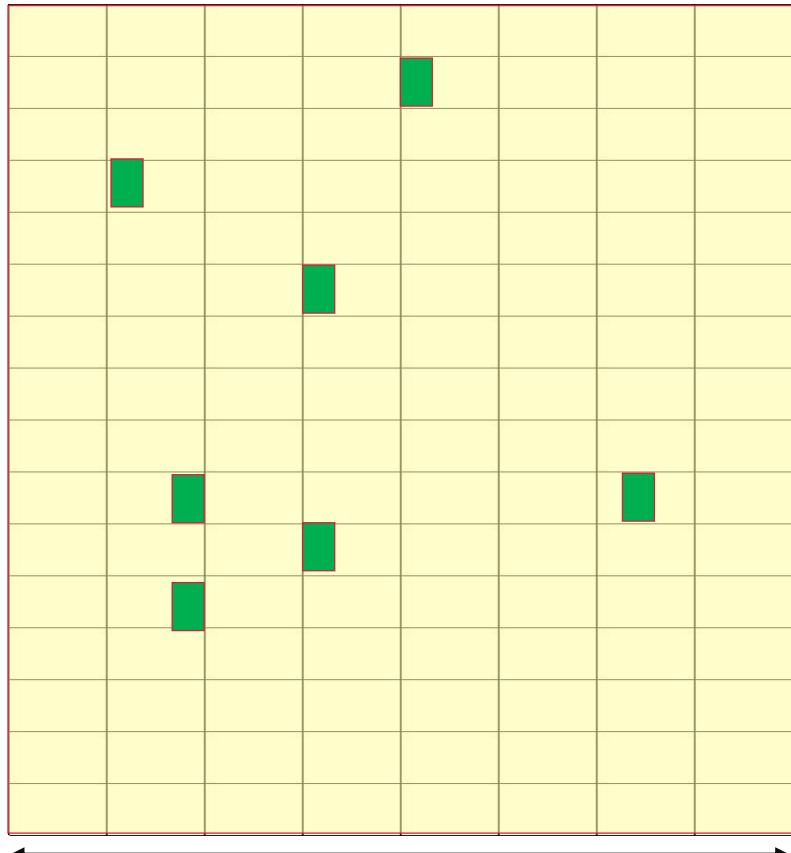
The cycle count depends on the **angle** by which the rectangle is rotated.

In the next slide we will explain why performance of gather/scatter operations is **dependent** on the memory **locations** of the loaded/stored data elements



# Performance Considerations for Gather/Scatter

## One local memory bank and sub-banks



8 sub-banks ( $8 * 64b = 512b$ )

- Gather/Scatter Operations loads/store up to 32 data elements
- elements in different memory banks or sub-banks can be accessed in **parallel**
- elements in **different** rows of the same sub-bank are accessed **sequentially**
- It can take from **1-32 cycles** to process one gather/scatter operation
- Only **one** Gather/Scatter operation can complete every **2 cycles**
- This means that gather/scatter operations can give you a best case throughput of **16 elements per cycle**
- The worst case throughput is 1 element every two cycles

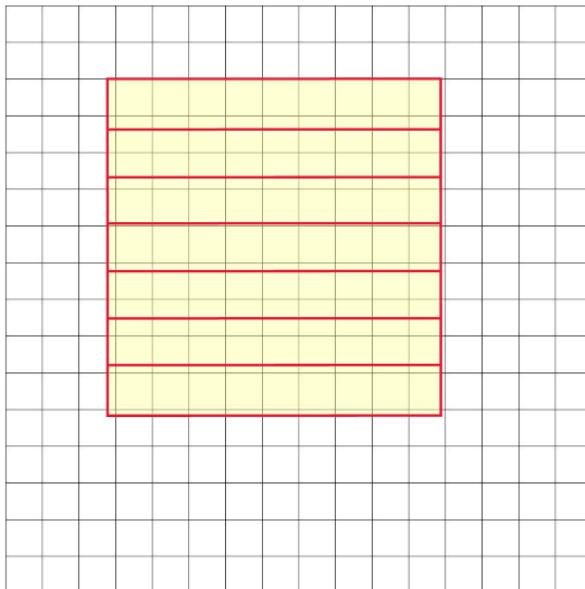
# Performance for Gather/Scatter continued ...

In order to get optimal data throughput out of your gather/scatter operations you want to make sure

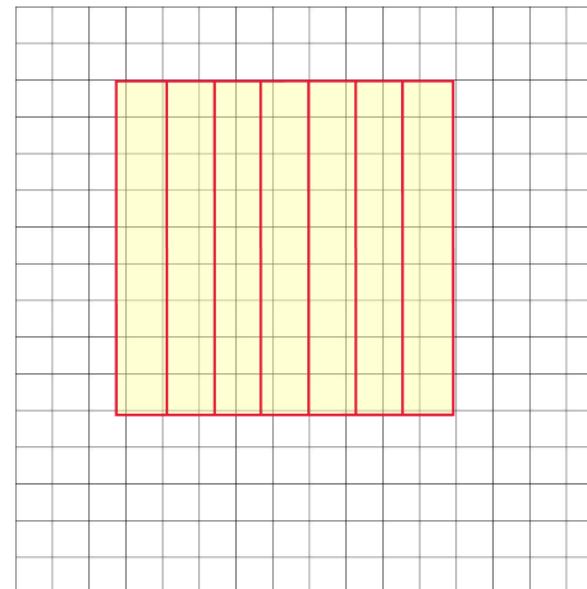
- That the data you are loading/storing with one gather/scatter operation is spread **across as many sub-banks** as possible
- This is only possible if you (the programmer) know about the **width** of the image buffer and have a good understanding of the **addresses** of the data elements you are loading.

# Performance for Gather/Scatter continued ...

- In the “rotated rectangle” example
  - We are gathering the data across a rectangle row
  - The image buffer width is a multiple of 64B (width of one memory bank)
  - With a rotation angle of **0** degrees the accesses are across **all the sub-banks**
  - With a rotation angle of **90** degrees the accesses are all within **one memory sub-bank**



0 degrees



90 degrees

# Gather Scatter Exceptions

- Address not aligned to Gather/Scatter Element Size
- Base address + offset
  - Not in local memory
  - Not contained within one local DRAM
  - Address offset is larger than 16 bits
  - Outside the valid MPU region

# Summary

In this module, you learn about of the Vision P6 Gather/Scatter Engine's

- Architecture
- Performance
- Application to vision examples

It takes time to master the use of Gather/Scatter operations due to their  
**address offset dependent completion time**

Please keep in mind that although Gather/Scatter operations are well suited to load data that is **randomly scattered** in memory they are **slower** than vector load and store operations when loading **contiguous** data from memory

The tutorial example in the module immediately store out data to memory after it was gathered. In **realistic** use cases you will likely want to **overlap** processing of the gathered data with processing of that data inside a loop to achieve best performance.

# Lab



## Lab 3: Gather / Scatter

- Use gather and scatter operations
- Understand gather scatter performance

# Quiz

## 1. The Gather/Scatter Engine

- a) Can load and store vector elements to/from a vector register to/from local DRAM
- b) Can load and store vector elements to/from a vector register to/from SYSRAM
- c) Can load and store vector elements to/from a vector register to/from either local DRAM or SYSRAM

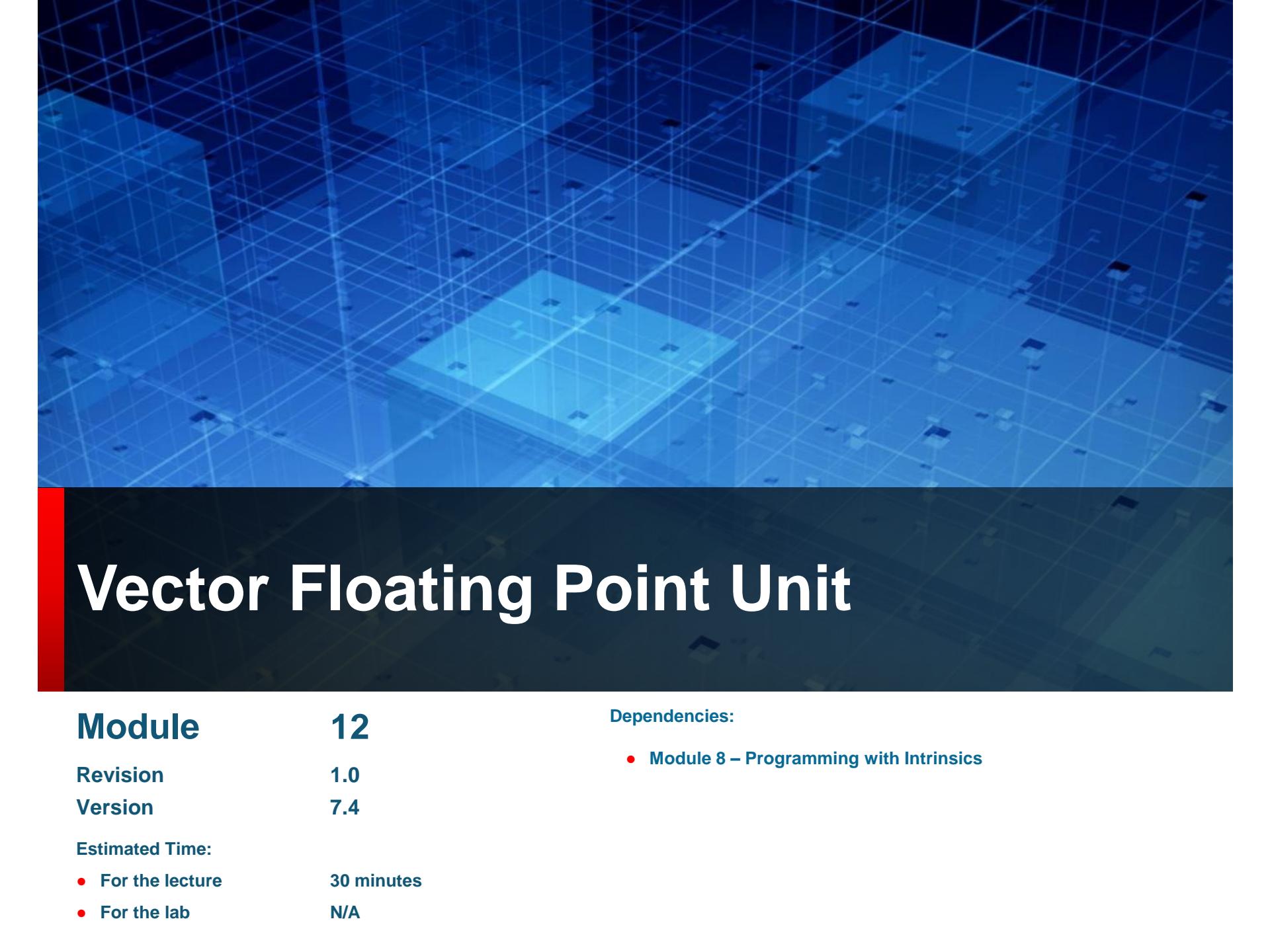
## 2. The Gather/Scatter Engine

- a) Is able to pipeline multiple operations to achieve better throughput
- b) Can only process one operation at a time and holds the next operation in the R-Stage until the current operation has completed

## 3. The processing time for a gather data operation

- a) Depends on the configuration options for the Gather/Scatter engine, but not the index vector.
- b) Depends on both the configuration options for the number of memory sub-banks and the index vector.
- c) is 7 cycles, since Vision P6 has a 7 stage pipeline.

Answers: 1a, 2a, 3b



# Vector Floating Point Unit

**Module** **12**

**Revision** **1.0**

**Version** **7.4**

**Estimated Time:**

- For the lecture **30 minutes**
- For the lab **N/A**

**Dependencies:**

- **Module 8 – Programming with Intrinsics**

# Module Objectives

In this module, you will learn about the Vision P6 Vector Floating Point Unit (VFPU). You will

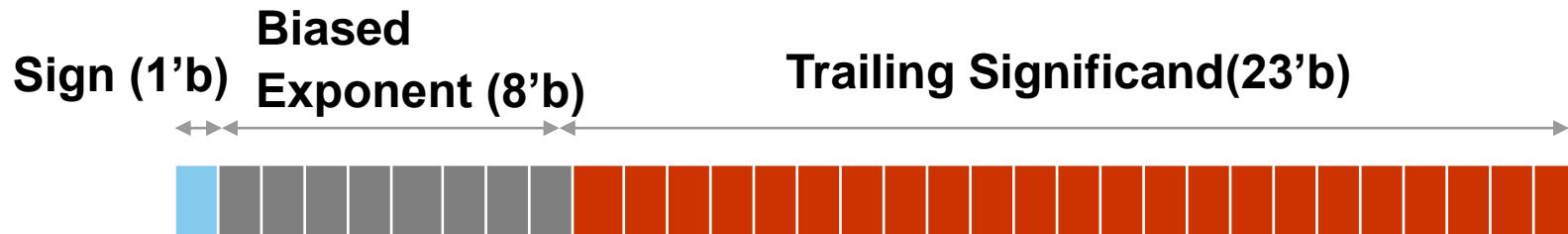
- Learn about half and single precision floating point data types
- Get an idea about the units performance and data throughput
- See a few usage examples

# Single/Half Precision VFPU Checkbox Options

The Single and/or Half Precision Vector FPU is a checkbox option in the configuration editor

- **Single precision Option:**
  - 16-way, 32-bit vector floating point operations
- **Half precision Option:**
  - 32-way, 16-bit vector floating point operations
- **Scalar floating** point checkbox option is **not compatible** with the options above
- However, the full set of scalar floating point operations are supported by using only **element0** of the 16 or 32 vector elements.

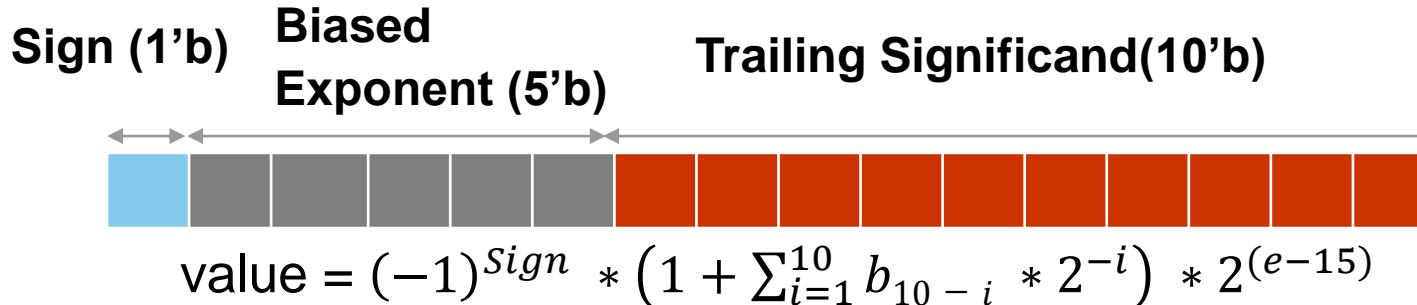
# Single Precision(32-bit) Floating Point Format



$$\text{value} = (-1)^{\text{Sign}} * \left(1 + \sum_{i=1}^{23} b_{23-i} * 2^{-i}\right) * 2^{(e-127)}$$

- IEEE 754, binary32 format
- Bias value is 127
- The significand consists of 1 implicit leading bit to the left of an implied binary point, and 23 explicit trailing bits to the right of the binary point.
- MAX representable value:  $3.4 \times 10^{38}$
- Advantages
  - Higher dynamic data range and precision
  - No need to convert to fixed if algorithm originated in floating point

# Half Precision(16-bit) Floating Point Format



- IEEE 754, binary16 format
- Bias value is 15
- The significand consists of 1 implicit leading bit to the left of an implied binary point, and 10 explicit trailing bits to the right of the binary point.
- MAX representable value: 65504
- Advantages of **half precision** VFPV
  - **Half the storage requirement and double the throughput** compared to single precision
  - Half precision data range often sufficient for many imaging and computer vision algorithms.

# Features of the VFPU

- Shares VEC register file, L/S Unit and other operations (SEL) with fixed-point operations
- Supports predicated operations

Signals 5 type of exceptions:

1. Invalid: if no usefully defined result.
2. Div by 0: if exact infinite result is generated for finite operands
3. Overflow: if destination largest finite number is exceeded
4. Underflow: signaled after rounding when a non-zero result is computed as though the exponent range were unbounded would lie strictly between a positive minimum normal number and a negative minimum normal number.
5. Inexact: if the rounded result differs from what would have been computed if both the exponent range and the precision were unbounded

# Floating point Control and Status Registers

## Floating-point Control Register(FCR):

4 rounding modes available in

0-> round to nearest

1-> round towards 0

2-> round towards +Infinity

3-> round towards -Infinity

Exceptions can be enabled by setting bit in **FCR**

## Status register:

- Exception Status Flags can be interrogated with this register

# VFPU Data Types

## Scalar Float Data type

- xtfloat (float): 32-bit single precision value
  - stored in **element0** of 16-way, 32-bit VEC register
- xb\_f16(half precision): 16-bit half precision value
  - stored in **element0** of 32-way, 16-bit VEC register

## Vector Float Data types

- xb\_vecN\_2xf32: 32-bit single precision
  - 16 values stored in VEC register
- xb\_vecNx16: 16-bit half precision
  - 32 values stored in VEC register

# VFPU Programming

## Auto-vectorization

- Use **float / xb\_f16** type and usual standard C operators
- Auto-Vectorizing Compiler will try to utilize 16/32-way SIMD operations

## Intrinsic programming

- Direct use of 16/32 way VFPU operations
- Use **xb\_vecN\_2xf32 / xb\_vecNxf16** 16/32 way vector data type

## Floating-point operations have longer latencies than integer ops.

- That makes instruction scheduling more challenging for compiler
- Loop unrolling is a key technique for optimization for VFPU operations

# VFPU Operation Categories

## MAC: multiply, multiply/add, multiply/subtract

- IVP\_MULN\_2XF32/ IVP\_MULNXF16
- IVP\_MULAN\_2XF32/ IVP\_MULANXF16
- IVP\_MULSN\_2XF32/ IVP\_MULSNXF16

## ALU: add, subtract, negate, abs

- IVP\_ADDN\_2XF32/ IVP\_ADDNXF16
- IVP\_SUBN\_2XF32/ IVP\_SUBNXF16
- IVP\_NEGN\_2XF32/ IVP\_NEGNXF16
- IVP\_ABSN\_2XF32/ IVP\_ABSNXF16

## Conversions: float to/from int

- IVP\_TRUNCN\_2F32/ IVP\_TRUNCNF16
- IVP\_FLOATN\_2F32/ IVP\_FLOATNF16

# VFP Operation Categories continued ...

## Round/Truncate/Ceiling/Floor

- IVP\_FIROUNDN\_2XF32/IVP\_FIROUNDNXF16
- IVP\_FITRUNCN\_2XF32/IVP\_FITRUNCNXF16
- IVP\_FICEILN\_2XF32/IVP\_FICEILNXF16
- IVP\_FIFLOORN\_2XF32/IVP\_FIFLOORNXF16
- IVP\_FIRINTN\_2XF32/IVP\_FIRINTNXF16

## Min/Max

- IVP\_MINN\_2XF32/ IVP\_MINNXF16
- IVP\_MAXN\_2XF32/ IVP\_MAXNXF16

## Ordered compare

- IVP\_O[EQ/LE/LT]N\_2XF32/IVP\_O[EQ/LE/LT]NXF16

## Un-Ordered compare

- IVP\_U[EQ/NEQ/LE/LT]N\_2XF32/IVP\_U[EQ/NEQ/LE/LT]NXF16

# VFP Operation Categories continued ...

## Conversion between half and single precision

- Single to half: IVP\_CVTF16N\_2XF32\_0
- Half to single: IVP\_CVTF32NXF16\_0/1

## Division and Square root

- DIV0/N, SQRT0/N

## Reciprocal and reciprocal square-root

- RECIP0/N, RSQRT0/N

## Special operations

- Fused Multiplication Addition
  - No rounding on multiplication.
  - Only final result is rounded.
  - It improves speed.
  - Can disable using -mno-fused-madd.

# Example 1: 3-tap (single-precision) FIR- Auto-Vectorization

```
void Filt3TapVectorAuto(float *pcoeff, float *pvecin, float *pvecout, int32_t veclen)
{
    float(*__restrict pc) = pcoeff;
    float(*__restrict pv) = pvecin;
    float(*__restrict pf) = pvecout;
    float temp;
    int32_t indx;
#pragma aligned (pv, 64)      // this will improve compiler's auto vectorization.
#pragma aligned (pf, 64)      // see section 4.7.2 of Xtensa C/C++ Compiler User's Guide
    for (indx = 0; indx < veclen; indx++)
    {
        temp      = (pv[indx] * pc[0]);
        temp     += (pv[indx + 1] * pc[1]);
        temp     += (pv[indx + 2] * pc[2]);
        pf[indx] = temp;
    }
    return;
}
```

- Schedules in 24 cycles for unroll of 8
- 3 cycles per iteration
- Schedule reaches **100%** of MUL bound

## Example 2: 128X128 (half-precision) Matrix Multiplication

$$C = A \times B$$

$$\begin{bmatrix} a_{0,0} & \cdots & a_{0,127} \\ \vdots & \ddots & \vdots \\ a_{127,0} & \cdots & a_{127,127} \end{bmatrix} \times \begin{bmatrix} b_{0,0} & \cdots & b_{0,127} \\ \vdots & \ddots & \vdots \\ b_{127,0} & \cdots & b_{127,127} \end{bmatrix} = \begin{bmatrix} c_{0,0} & \cdots & c_{0,127} \\ \vdots & \ddots & \vdots \\ c_{127,0} & \cdots & c_{127,127} \end{bmatrix}$$

Implementation:

$$C = \begin{bmatrix} a_{0,0} \\ \vdots \\ a_{127,0} \end{bmatrix} \cdot \begin{bmatrix} b_{0,0} & \cdots & b_{0,127} \\ \vdots & \ddots & \vdots \\ b_{0,0} & \cdots & b_{0,127} \end{bmatrix} + \begin{bmatrix} a_{0,1} \\ \vdots \\ a_{127,1} \end{bmatrix} \cdot \begin{bmatrix} b_{1,0} & \cdots & b_{1,127} \\ \vdots & \ddots & \vdots \\ b_{1,0} & \cdots & b_{1,127} \end{bmatrix} + \dots + \begin{bmatrix} a_{0,127} \\ \vdots \\ a_{127,127} \end{bmatrix} \cdot \begin{bmatrix} b_{127,0} & \cdots & b_{127,127} \\ \vdots & \ddots & \vdots \\ b_{127,0} & \cdots & b_{127,127} \end{bmatrix}$$

Here [X].[Y] implies point-wise multiplication

# Example 2: 128X128 (half-precision) Matrix Multiplication

```
//Initialize "Out" with Zero for MAC operations.  
for (col = 0; col < 128; col++){  
    f16vecRowB0 = *((xb_vecNxf16 *) pf16vecB++);  
    f16vecRowB1 = *((xb_vecNxf16 *) (pf16vecB++));  
    f16vecRowB2 = *((xb_vecNxf16 *) (pf16vecB++));  
    f16vecRowB3 = *((xb_vecNxf16 *) (pf16vecB++));  
    pf16vecC = (xb_vecNxf16 *) Out;  
    pf16vecC2 = (xb_vecNxf16 *) Out;  
  
    for (row = 0; row < 128; row++){  
        xb_f16_loadxp(xTemp0, f16ptrA, 2 * STRIDE);  
        f16vecTempA0 = xTemp0;  
        IVP_LVNXF16_XP(f16vecTempSum0, pf16vecC, 64);  
        IVP_MULANXF16(f16vecTempSum0, f16vecTempA0, f16vecRowB0);  
        IVP_SVNXF16_XP(f16vecTempSum0, pf16vecC2, 64);  
        //Above block is repeated for f16vecRowB1/2/3.  
    }  
    f16ptrA -= (ROW * STRIDE);  
    f16ptrA++;  
}
```

- Example uses Intrinsics
- 32-way, 16 bit SIMD floating operations
- Inner loop produces 128 products with 4, 32-way 16-bit MUL operations
- Schedule is 9 cycles for unroll of 2 (89% of MUL bound)
- This code is part of the fp16MatrixMultiply128x128 project in the **SwP**

# Summary

In this module you learned about

- the Vision P6 VFPU
- performance and data throughput rates

The VFPU is a configuration option.

- If you require high performance throughput of float operations the VFPU might be a good choice for you but adds a noticeable amount of area.
- If float operation only consume a small portion of the overall budget omitting it or enabling a **scalar FPU** might be a better choice.

# Quiz

## 1. The VFPU

- a) Supports double precision floating point operations on a vector of 16 32-bit floating point numbers
- b) Supports single precision floating point operations on a vector of 16 32-bit floating point numbers
- c) Supports half precision floating point operations on a vector of 32 16-bit floating point numbers
- d) b or/and c

## 2. Floating point math operations like sine, cosine and arctan

- a) Are supported natively as an operation
- b) Require support via a Library
- c) Are not supported, but can easily be implemented in fixed point arithmetic with a table lookup

Answers: 1d, 2b

# Histogram Package

**Module** **13**

**Revision** **1.0**

**Version** **7.4**

**Estimated Time:**

- **For the lecture** **30 minutes**
- **For the lab** **N/A**

**Dependencies:**

- **Module 8 – Programming with Intrinsics**

# Module Objectives

In this module, you will learn about the Histogram Package.

We will cover

- An Introduction to the Histogram package
- Describe Histogram Terminology
- Cover the most important histogram operations
- Demonstrate how these operations can be used to calculate an image histogram
- Go over a code example computing an 8-bit 256-bin histogram

# Introduction to the Histogram Package

- The histogram package is an optional TIE package for the Vision P6 configuration
- It accelerates histogram calculations by up to 40 times compared to a P6 configuration without that package
- The package is most efficient for 8-bit values for and a small number of bin counts (32 to 256 bins)
- It progressively gets less efficient for higher bit values (10-bit and 12-bit) and larger bin counts
- For bin counts between 2000 – 4000 bins there is almost no noticeable acceleration
- If histogram calculations have a significant cycle contribution in your application and if bin counts are moderate you might want to consider this package for your Vision P6 configuration

# Introduction to the Histogram Package continued ...

- The histogram package adds operations that accelerate histogram calculation for 8-bit input data
- However, it can also be used when input data exceeds 8-bit
- In this case the histogram operations need to be called iteratively, while holding the upper bit counts constant and making use of predicated versions of the histogram operations.
- This is demonstrated in multiple examples of the project **saHistogramKernels** in the **SwP**
- In this module we will only cover the case of 8-bit input data with a bin count of 256, but encourage the user to study the other use cases in the project above.

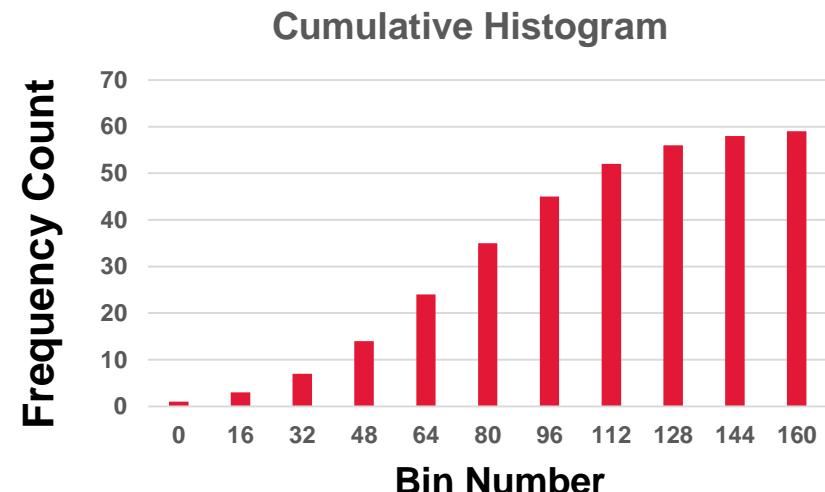
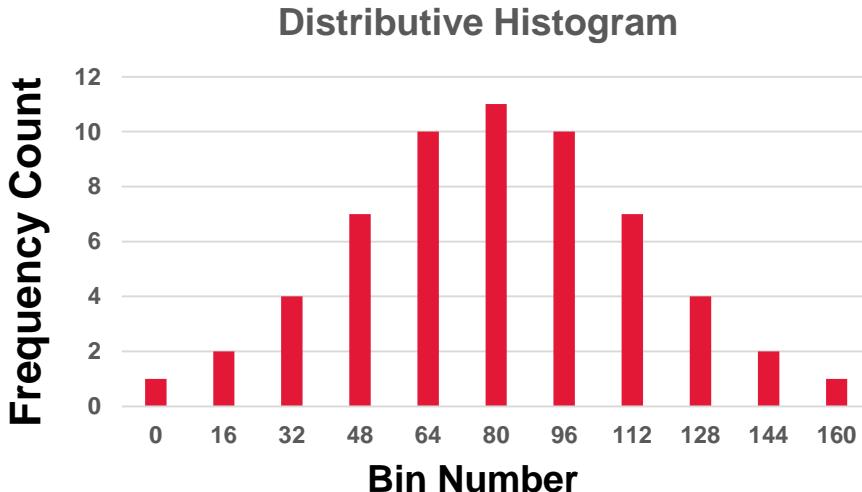
# Histogram Terminology

An image histogram

- Contains **bins** of frequency counts for every image value
- **Bins** can be single value or multiple values wide

We consider 2 types of histograms

- **Distributive** – every bin is a frequency count
- **Cumulative** – integral (sum) over the bins of a distributive histogram



# Histogram Package Operation: COUNT

## The COUNT Operation

- Consumes 2 input VEC registers (128 8-bit elements)
- Produces 1 output VEC register (32 16-bit elements) in one cycle

## Flavors of COUNT operations:

- Distribution Histogram: **COUNTEQ** (count equal)
- Cumulative Histogram: **COUNTLE** (count less than)
- Masked versions **COUNT(EQ/LE)M** take 2 additional vbool operands and count only elements with corresponding mask value set to 1
- Initially a **zeroing** version of this operation is used (e.g., **COUNTEQZ**) to reset the bin range to bins 0,1,...,31
- Normal versions increase the bin range (e.g., **COUNTEQ**) and are called repeatedly until the final bin range (bins 224, 225, ..., 254,255) is reached.

# **8-bit 128-element input count operation**

## **IVP\_COUNTLE4NX8**

### **C Syntax:**

```
IVP_COUNTLE4NX8(xb_vecNx16U vt /*out*/, int arr /*inout*/,
xb_vec2Nx8U vs, xb_vec2Nx8U vr);
```

### **Inputs:**

- 2 VEC input registers containing 128 8-bit values
- 1 AR register
  - bits [0:1] (shift = 0,1,2,3) specify the input right shift amount
  - bits [18:16] (j = 0,1,...,6,7) specify the output bin range j, Bin  $32^*j$  to Bin  $32^*(j+1)$

### **Output:**

- 1 VEC register (32 elements \* 16 bits), containing 32 output bin values
- 1 AR register
  - Incremented value of bits [18:16] for next output bin range

# 128 8-bit element input count operations IVP\_COUNTLE4NX8 ... continued

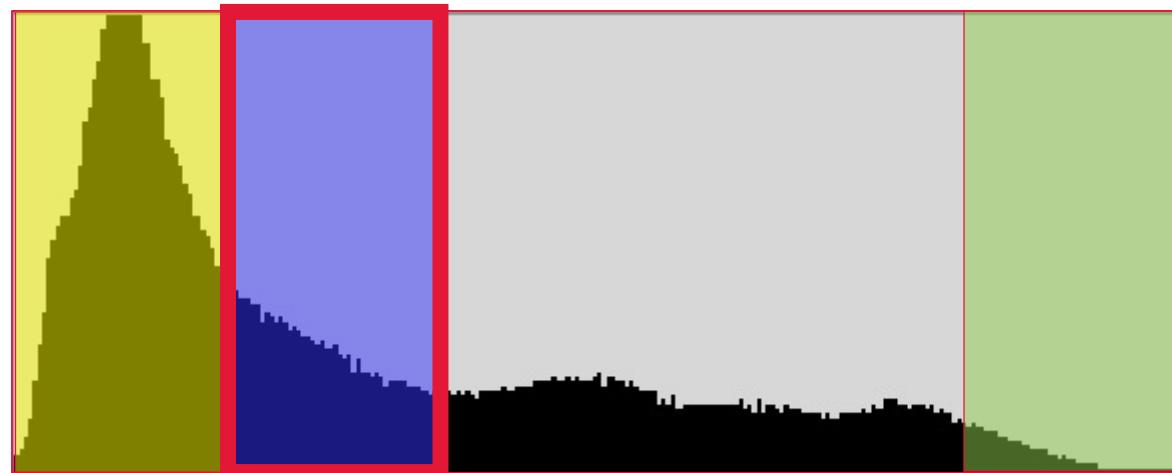
## Description:

- Each of the 128 input values is checked against the current bin range  $(32*j) \leq value < 32*(j+1)$
  - For each value that falls into the range the corresponding bin is incremented by 1
  - If the value is out of that range none of the bins are incremented for that value
  - A shift value from 0 to 3 can be used, to scale (right shift) the input values. This effectively creates a bin width of 1, 2, 4 or 8 values.
  - At the end of the operation that bin range is incremented to prepare bin range for the next operation

# 128 8-bit element input count operations

## IVP\_COUNTLE4NX8 ... continued

256-bin histogram broken into 32-bin ranges



0 - 31

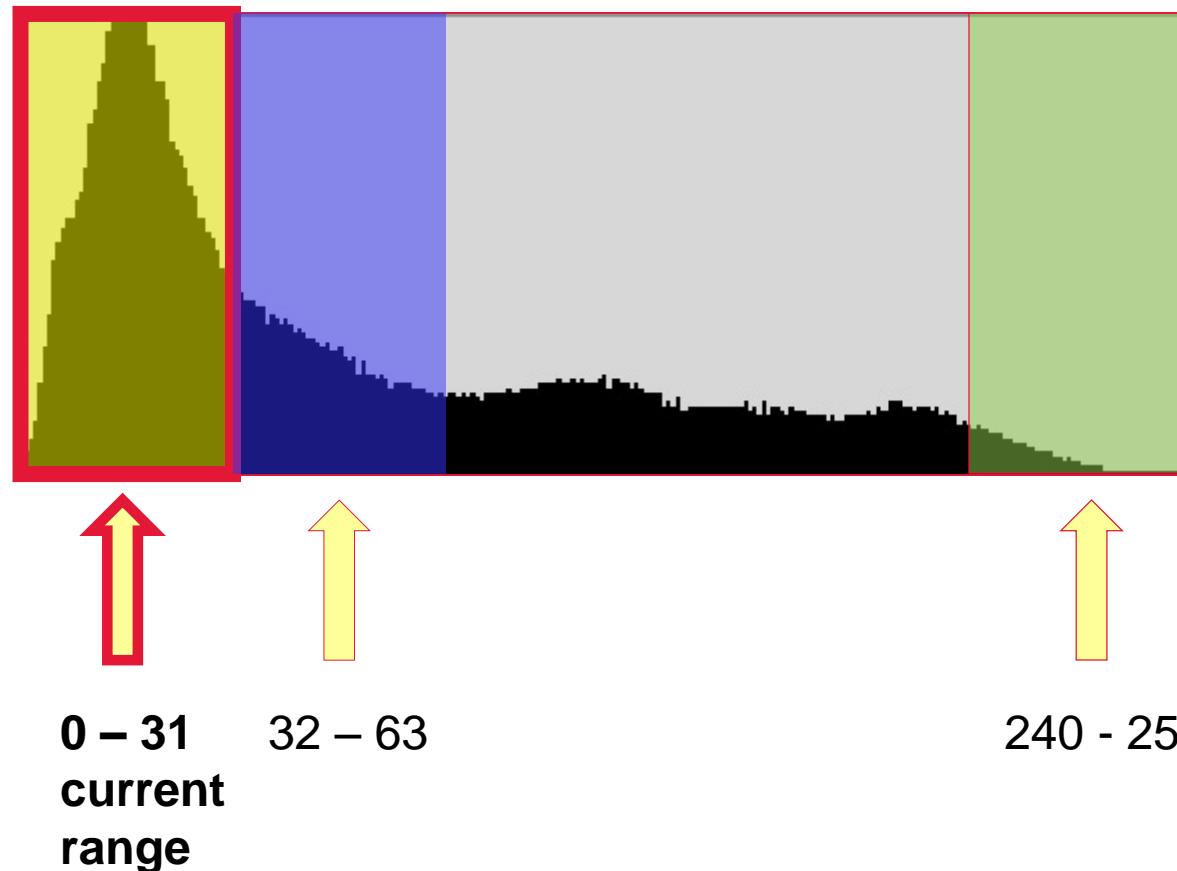
32 – 63  
current  
range

224 - 255

# 128 8-bit element input **zeroing** count operations

## IVP\_COUNTLEZ4NX8 ... continued

256-bin histogram broken into 32-bin ranges



# Code Example For Histogram Calculation

- The next slide contains a code example from the project **saHistogramKernels** in the **SwP**
- The functions `xvHistDistributive8bit256bin_H` calculates a 256-bin histogram of 8 bit input values.
- It takes as inputs arguments
  - a pointer to the input data buffer (`uint8_t*`)
  - a pointer to the histogram buffers(`uint16_t`)
  - `binCnt` (256)
  - Input data shift amount (0)

# Code Example: xvHistDistributive8bit256bin\_H

```
void xvHistDistributive8bit256bin_H(uint8_t * __restrict datain,
                                    uint16_t * __restrict pHist, int32_t binCnt, int32_t dataBitsShiftRight)
{
    xb_vecNx16U * __restrict pvecHist = (xb_vecNx16U *) pHist;
    xb_vec2Nx8U * __restrict pdvecData = (xb_vec2Nx8U *) datain;
    int32_t i;
    uint32_t sr = dataBitsShiftRight;
    xb_vecNx16U vecHist;
    xb_vec2Nx8U dvecData0, dvecData1;

    // zero histogram
    for (i = 0; i < (binCnt) / IVP SIMD WIDTH; i++)
    {
        IVP SVN16_X(0, (xb_vecNx16 *) pvecHist, i * 2 * IVP SIMD WIDTH);
    }

    for (i = 0; i < (DATA_SIZE / INPUT_LANES / 2); i++)
    {
        dvecData0 = *pdvecData++;
        dvecData1 = *pdvecData++;
        IVP_COUNTEQZ4NX8(vecHist, sr, dvecData0, dvecData1);
        pvecHist[0] += vecHist;
        IVP_COUNTEQ4NX8(vecHist, sr, dvecData0, dvecData1);
        pvecHist[1] += vecHist;
        IVP_COUNTEQ4NX8(vecHist, sr, dvecData0, dvecData1);
        pvecHist[2] += vecHist;
        IVP_COUNTEQ4NX8(vecHist, sr, dvecData0, dvecData1);
        pvecHist[3] += vecHist;
        IVP_COUNTEQ4NX8(vecHist, sr, dvecData0, dvecData1);
        pvecHist[4] += vecHist;
        IVP_COUNTEQ4NX8(vecHist, sr, dvecData0, dvecData1);
        pvecHist[5] += vecHist;
        IVP_COUNTEQ4NX8(vecHist, sr, dvecData0, dvecData1);
        pvecHist[6] += vecHist;
        IVP_COUNTEQ4NX8(vecHist, sr, dvecData0, dvecData1);
        pvecHist[7] += vecHist;
    }
}
```

pvecHist pointer to current Histogram bin  
dvecData0, dvecData1

- 2 vectors holding 128 pixels
- Load 128 pixels
- Call **zeroing** version of **count**
- Call count 7 more times
- pvecHist[0] holds bins 0,...,31
- pvecHist[7] holds bins 224,...,255
- With each **iteration** we **accumulate** into histogram
- We stop after input data buffer is consumed

# Summary

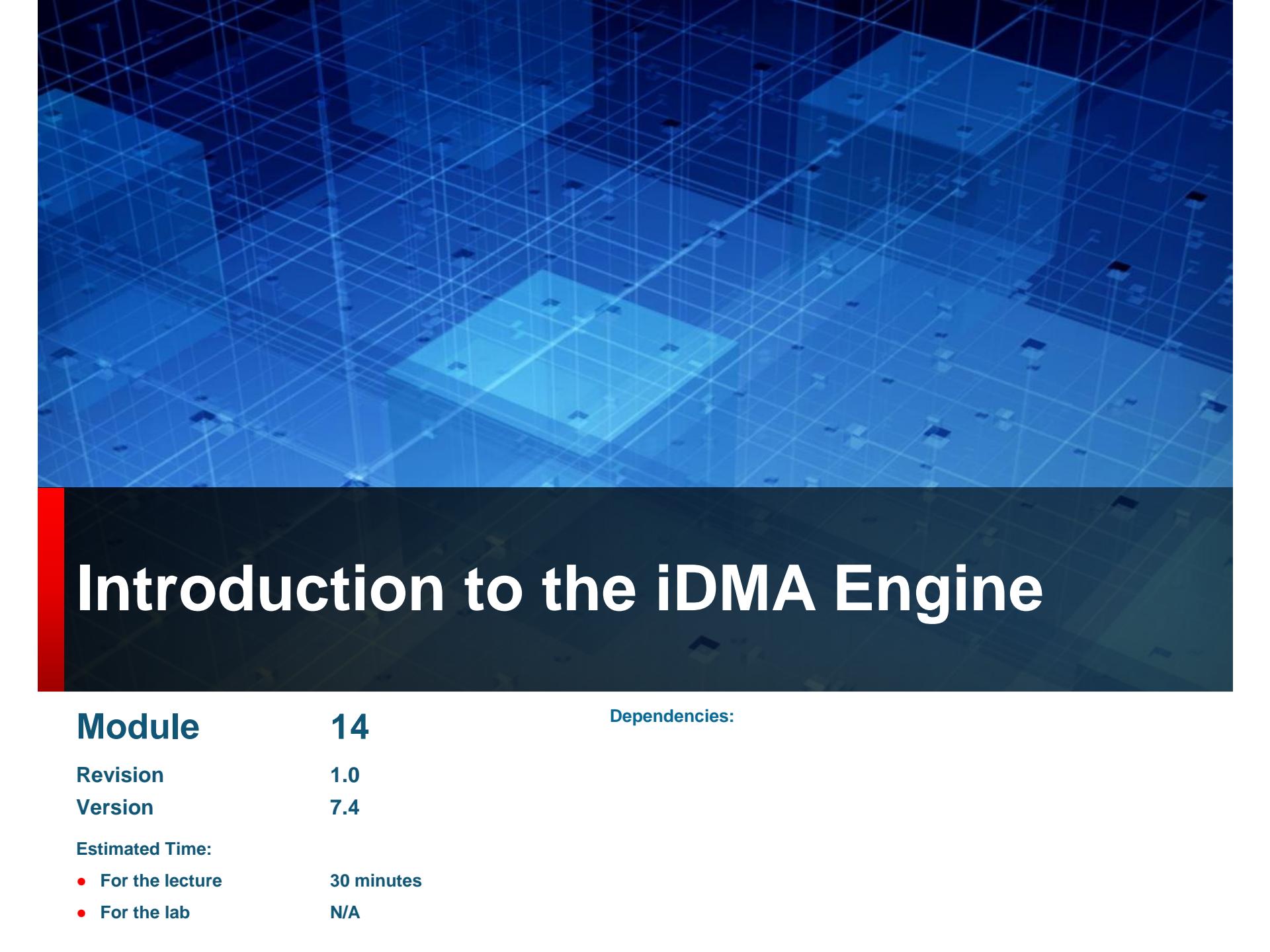
In this module, you got an overview of the Histogram Package. You learned about

- Distributive and Cumulative Histograms
- Learned under what circumstance the histogram package might be of interest to you
- Learned that the Histogram Package can also be used for input data wider than 8-bit (maybe 10 or 12-bit)
- Learned Important Histogram Operations
- Saw how the Operations can be used to calculate an 8-bit 256-bin histogram

# Quiz

1. Which of the following statements is true about the histogram package
  - a) It can only be used for 8-bit input data
  - b) It can be used for 8-bit or wider input data, any number of bin-counts and always accelerates histogram calculations by a factor of 40X over a configuration without
  - c) Whether the histogram package accelerates this histogram calculations strongly depends on the histogram's bin count
2. The Operations IVP\_COUNTLEZ4NX8
  - a) Sets the Histogram to 0 and calculates 32 bin values
  - b) Sets bin range to 0-31 and calculates 32 bin values

Answers: 1c, 2b



# Introduction to the iDMA Engine

**Module** **14**

**Dependencies:**

**Revision** **1.0**

**Version** **7.4**

**Estimated Time:**

- **For the lecture** **30 minutes**
- **For the lab** **N/A**

# Module Objectives

In this module, you will learn about the iDMA engine of Vision processor

- Why DMA engine is necessary for Vision processor
- DMA transaction life cycle and latencies
- How latency affects DMA throughput
- The design and features of the iDMA engine
- DMA descriptors and scheduling
- Configuration options and parameters to reduce latency
- Simulation of an iDMA application

# Vision Applications Have High Data Throughput

- Vision processing has two key challenges: **data movement** and **data processing**
- Imaging/vision applications process a huge amount of multi-dimensional data

Frame Size	30 fps	60 fps	120 fps
1280x720x3 (HD)	83 MB/s	166 MB/s	332 MB/s
1920x1080x3 (FHD)	187 MB/s	373 MB/s	746 MB/s
3840x2160x3 (4K)	746 MB/s	1493 MB/s	2986 MB/s

- The numbers are even higher for
  - Multiple video channels: multi-spectrum, stereo vision, multi-view vision
  - Multi-pass processing
- Imaging/vision application often requires complex memory layout and access patterns

# Vision DSP's Memory Hierarchy

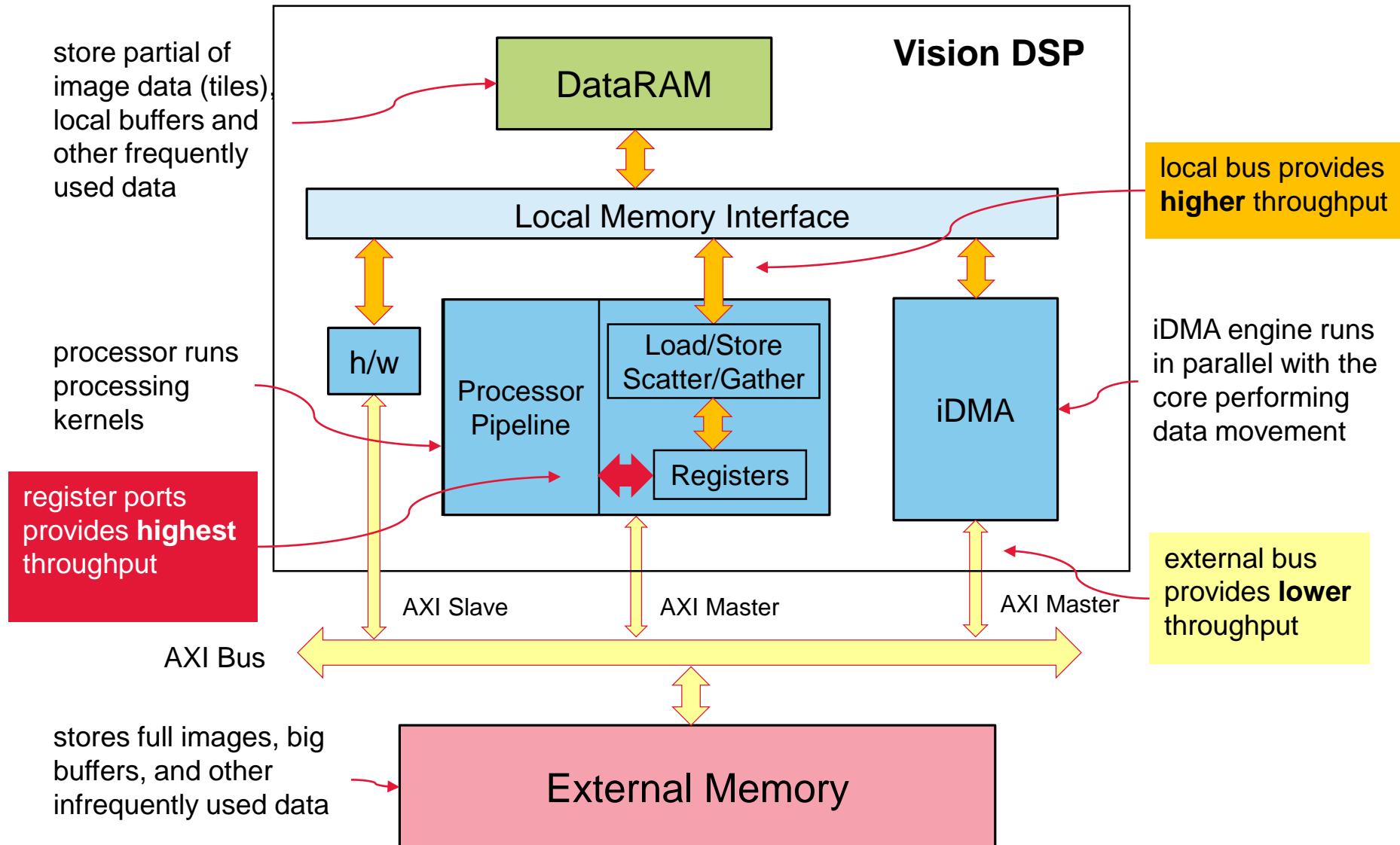
- Most vision processing tasks are on localized data
- Vision DSP's memory hierarchy is designed to support high throughput on localized data
  - Registers: small set, directly coupled into processor pipeline, direct access
  - DataRAM: small size, local bus access, low latency
  - External memory: large size, external bus access, high latency

Memory Type	Size	Connection	Bandwidth	Latency
Register	~3.5 KB	Register port	>1K B/cycle	Direct access
DataRAM	32 KB ~ 1 MB	Local bus	128 B/cycle	Low
External Memory	Giga Bytes	AXI bus	16 B/cycle	High

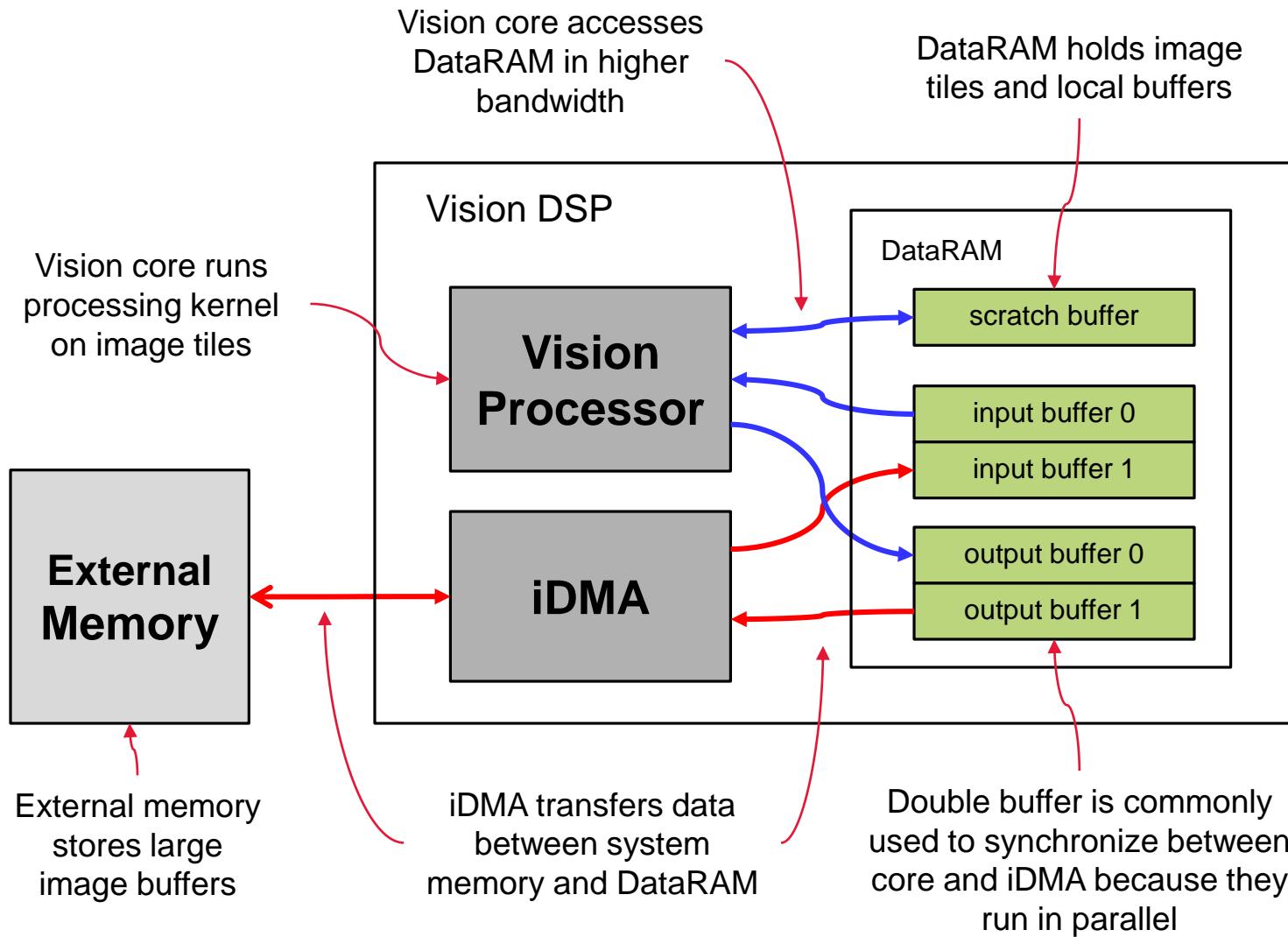
# The Function of the DMA Engine

- In Vision DSP's memory hierarchy
  - External memory holds large images and buffers
  - DataRAM holds portions (tiles) of these images and buffers
- Processor needs to access DataRAM with high throughput
- iDMA engine offloads data movement from processor so that it is not disturbed from processing data at this high rate
- iDMA is designed to be tightly-coupled with processor to avoid processor idling
- Data movement can be between external memory (SYSRAM) and DataRAM, or between DataRAM regions

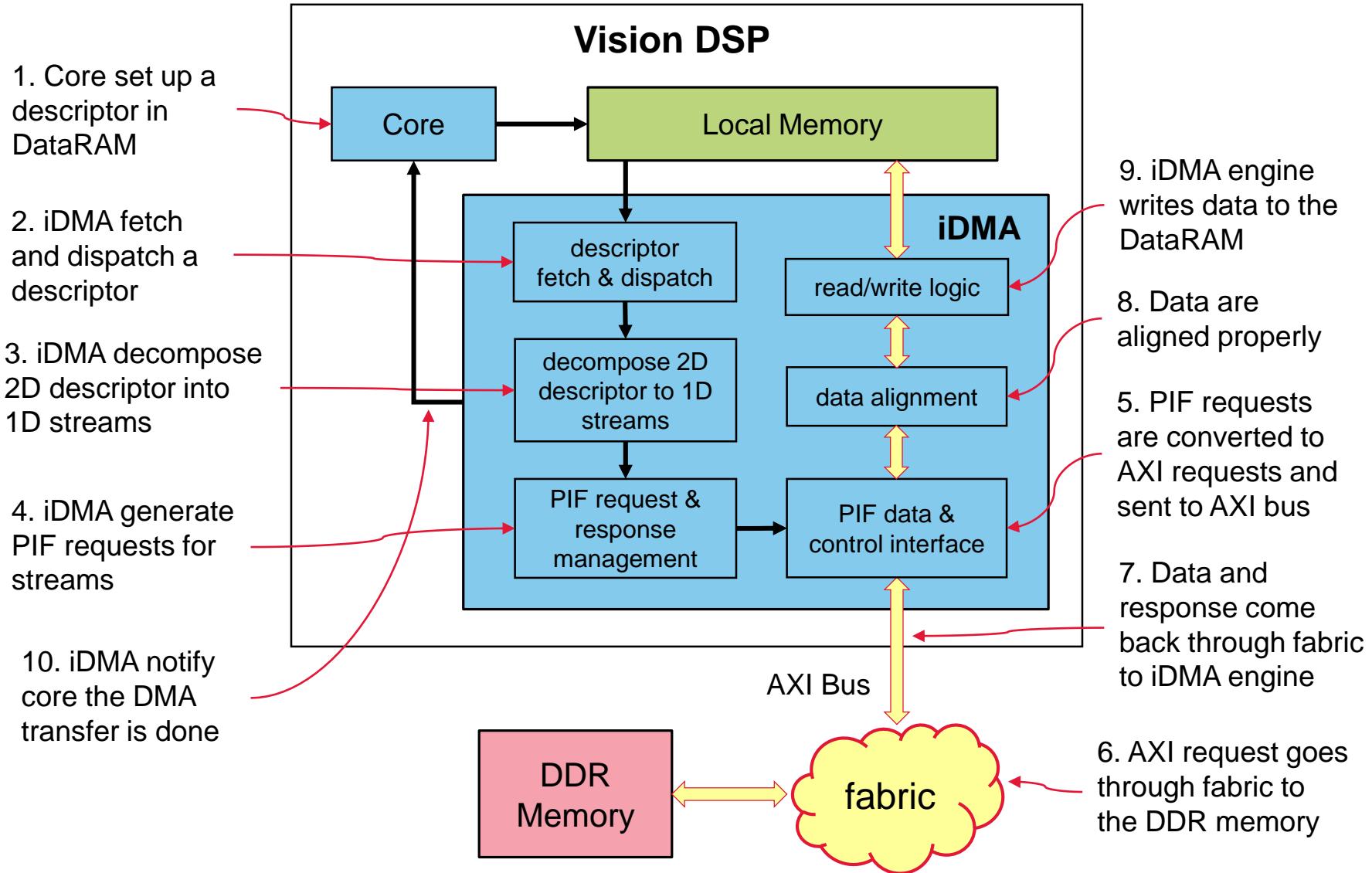
# Vision DSP Block Diagram



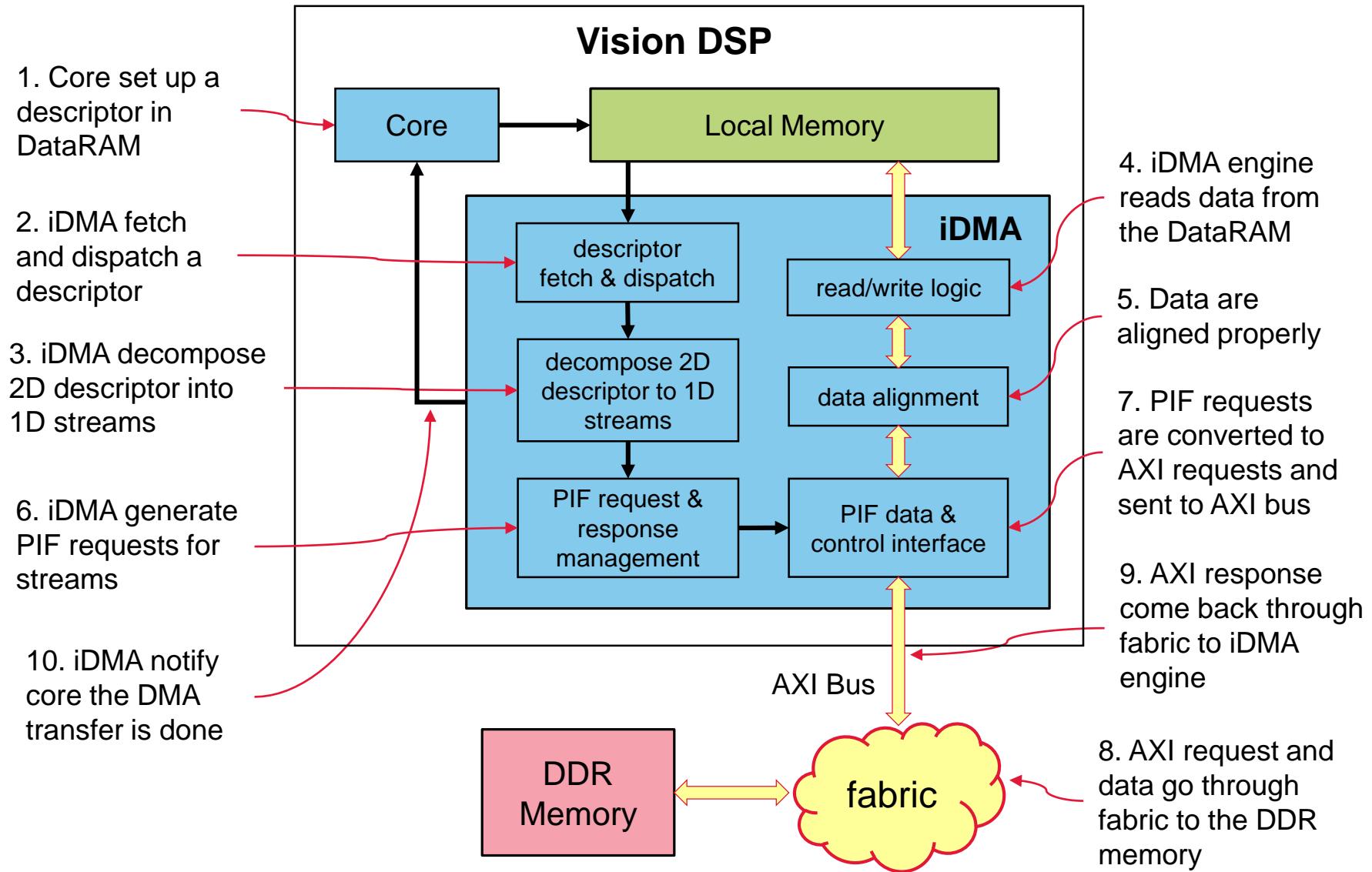
# A Typical Vision Application with DMA Transfers



# Life Cycle of a DMA Read Transfer



# Life Cycle of a DMA Write Transfer



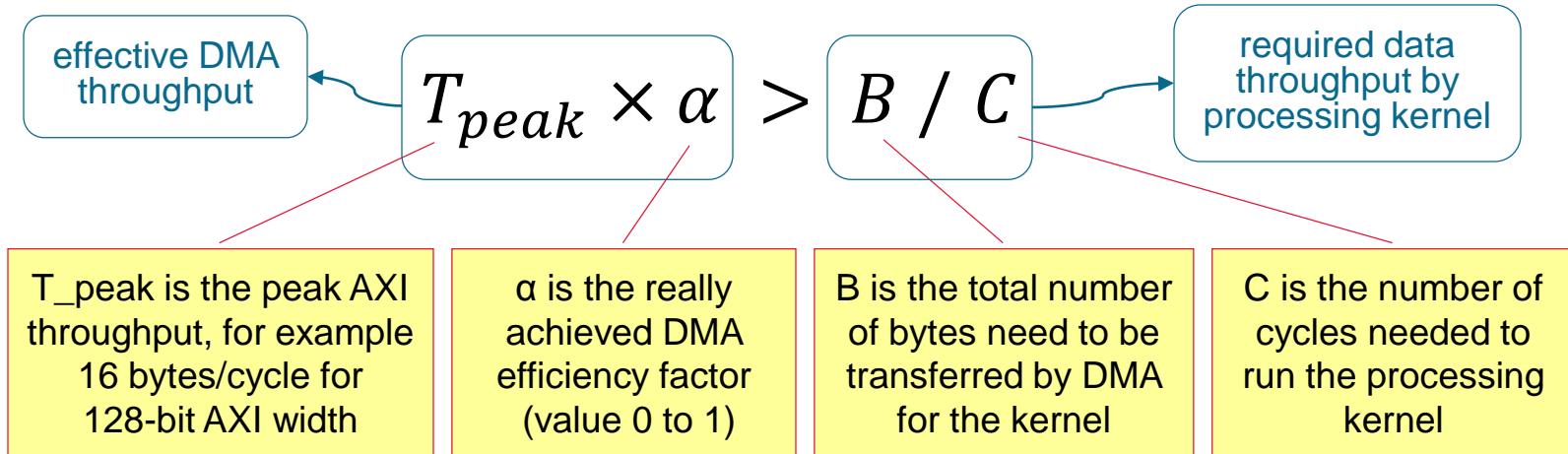
# Latencies Affect DMA Efficiency

- Latency is the major cause that lowers the DMA efficiency factor  $\alpha$
- There are different latencies throughout a DMA transfer

Sources of Latency	Location	Percentage
Schedule a descriptor	Core	Small
Fetch and parse a descriptor	iDMA	Small
Decompose 2D descriptor to 1D	iDMA	Small to medium
System bus	System bus	Medium to large
DDR memory access	DDR	Medium to large
DataRAM access	DataRAM bus	Small to medium
Data alignment	iDMA	Small

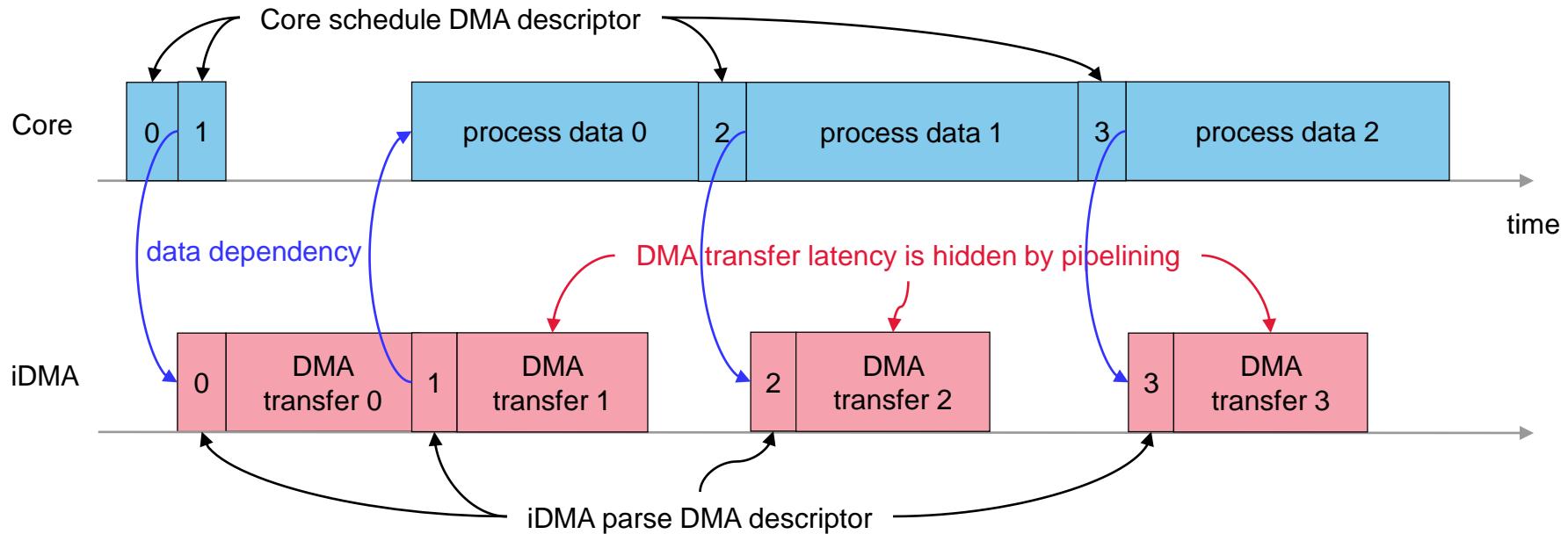
# Effective DMA Throughput

- Ideally the effective DMA throughput (**T**) should meet the computation throughput requirement (**B/C** where **B** is number of Bytes moved and **C** is the cycle count):



- To meet the above requirement, we need to
  - Optimize data flow design to minimize  $B$
  - Maximize efficiency factor  $\alpha$ , which will be explained further

# Use Double Buffering to Hide DMA Latency with Computation



- Processor core and iDMA engine run in parallel
- Core issues iDMA requests ahead of time so that iDMA engine processes them before the processor needs them
- Double buffering is common scheme that allows processor and DMA engine to work in parallel
- Unless DMA throughput is the bottleneck, double buffering hides the DMA transfer latency from the processor

# How can you improve DMA Efficiency?

- When DMA throughput is the bottleneck, double buffering cannot hide the DMA latency completely and cause the processor to idle
- Avoid that DMA becomes the bottleneck by
  - merging kernels to decrease the amount of data the DMA engine has to move
  - improving DMA efficiency factor (see below)
- DMA efficiency factor is affected by
  - 2D tile size shape (**the wider the better**), and address alignment (**256 Byte boundaries** allows highest block size)
  - PIF settings (**max block size** and **number of outstanding requests**)
  - Long latency and big latency variation from system bus/memory
  - DRAM access conflicts between processor (**L/S**) and **DMA** engine
- How to improve a low DMA efficiency factor? When possible,
  - Choose larger tile size, larger row size, and aligned addresses
  - Choose largest max PIF block size
  - Pipeline more PIF requests ahead of time, and set number of outstanding PIF requests to a larger value
  - Consider having iDMA and processor access separate DataRAM to avoid conflicts

# DMA Efficiency Example: PIF block size and number of outstanding requests

- Test condition
  - Image size = 1920x1080, tile size = **256x16** (256 width, 16 height)
  - **B = max PIF block size, N = number of outstanding PIF requests**
  - **L(v1, v2) = memory latency cycles** (initial access v1, repeat access v2)
  - The values in the table below are **DMA efficiency factor alpha**

alpha	L = (10, 1)	L = (100, 1)	L = (500, 1)
B = 2, N = 4	0.53	0.08	0.02
B = 2, N = 64	0.73	0.62	0.21
B = 16, N = 4	0.60	0.14	0.03
B = 16, N = 64	0.71	0.60	0.31

- Conclusion
  - DMA throughput (alpha) decreases with increased latency
  - Larger block sizes and a higher number of outstanding PIF requests improve the DMA throughput since it allows these requests to pipeline deeper
  - Be careful that you choose proper values for max PIF block size and number of outstanding PIF requests and memory latency when studying performance of code with DMA

# DMA Efficiency Example: Tile Size and Shape

- Test condition
  - Image size = 1920x1080
  - **Max PIF block size = 16, number of outstanding PIF requests = 64**
  - $L(v1, v2)$  = **memory latency cycles** (initial access v1, repeat access v2)
  - The values in the table below are DMA efficiency factor

Tile Size	$L = (10, 1)$	$L = (100, 1)$	$L = (500, 1)$
256x16	0.71	0.60	0.31
128x32	0.69	0.57	0.20
64x64	0.61	0.45	0.12
32x32	0.45	0.23	0.05

- Conclusion
  - Wider tiles improve the DMA throughput for larger memory latencies (As they allow block requests with larger block sizes).
  - Choosing larger tile size also leads to a higher DMA throughput (PIF requests can not pipeline across tiles). They also reduce processor cycles for DMA scheduling since less transfers have to be scheduled per image.
  - When possible, choose larger tile size and larger tile width to generate PIF requests with maximal block size to better hide long memory latency

# iDMA Engine Overview

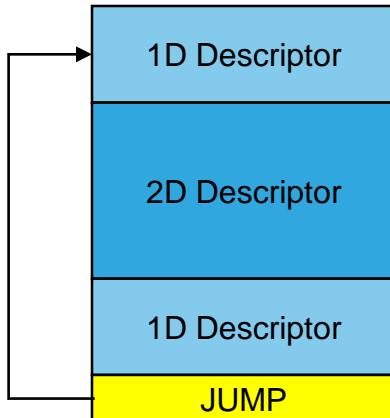
- iDMA engine is a single-channel DMA engine, multiple DMA requests (tiles) are processed sequentially
- Details of iDMA transfer
  - Support 1D and 2D transfers (tiles)
  - Flexible scheduling using registers and JUMP command (chaining)
  - Support arbitrary data alignment to byte address
  - iDMA supports data transfers between system memory and local Data RAM, and between local Data RAM regions
  - iDMA does NOT support data transfer between system memory regions
  - iDMA performs memory protection by MPU lookup
  - Data transfers cannot cross the Data RAM boundary
  - Source and destination region should not overlap
  - Status report by interrupt or register polling
  - H/W trigger input and output to synchronize with external logic

# iDMA Registers

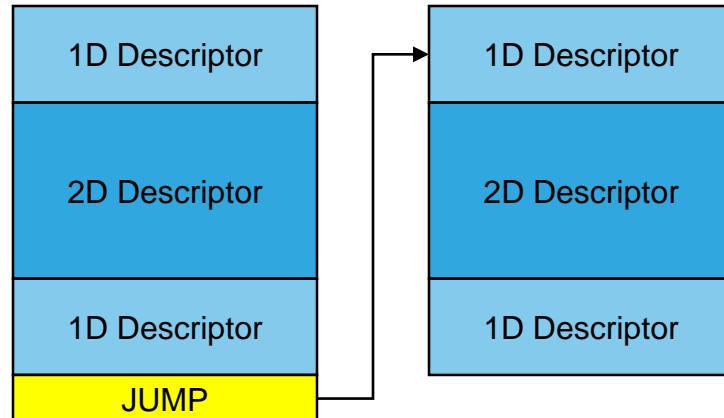
- Processor controls iDMA by writing iDMA registers using WER instruction
- Processor reads iDMA status and internal states by reading iDMA registers using RER instruction
- iDMA registers have user and privilege mode for access protection
- For more details, please refer to Xtensa LX7 Microprocessor Data Book, section 19.3, iDMA Registers

# iDMA Descriptor

- Processor schedules DMA transfers through registers and descriptors
- A descriptor is a command that defines parameters of a DMA transfer. It is stored in DataRAM (16 or 32 Bytes per descriptor)
- Two types of descriptors for either 1D or 2D transfers
- Processor prepares descriptors in DataRAM and informs iDMA of the starting address and the number of descriptors by writing to iDMA registers
- iDMA engine reads and parses descriptors and performs DMA transfers
- Special JUMP command (4 Bytes) directs iDMA engine to the next descriptor



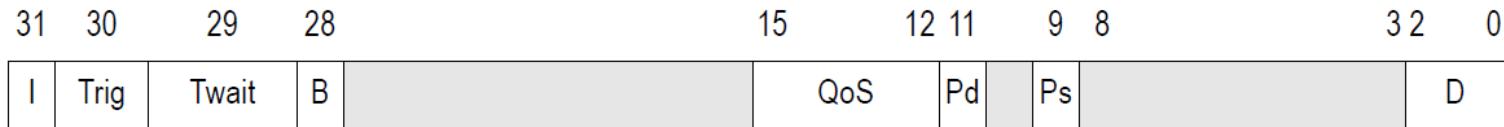
Use JUMP command to form a circular buffer



Use JUMP command to form a linked list

# iDMA 1D Descriptor

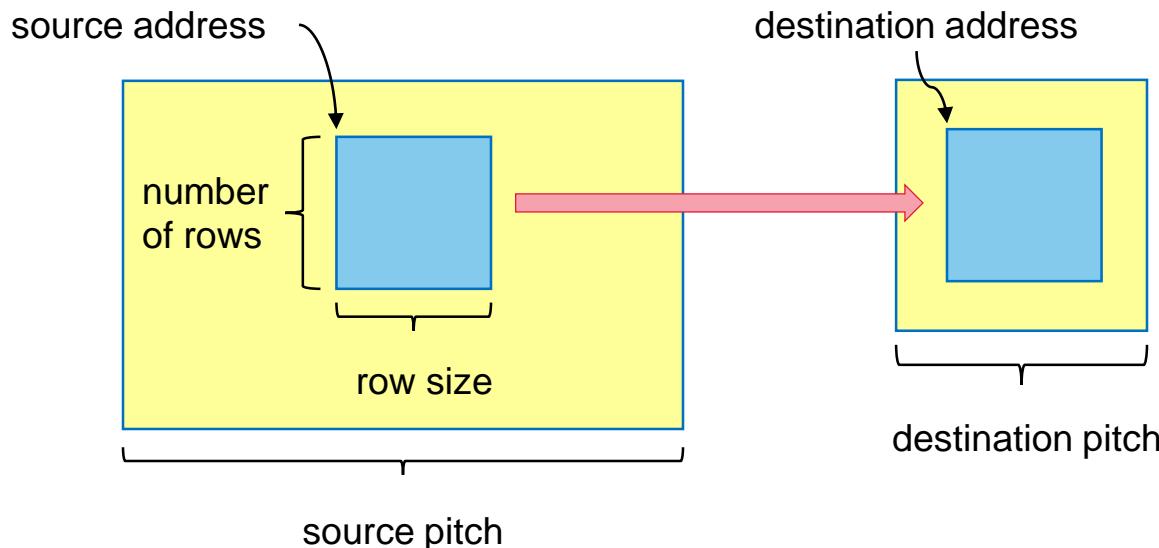
- 1D descriptor: contains four 32-bit fields, defines one 1D memory transfer
  - Word 0: control word
  - Word 1: source address
  - Word 2: destination address
  - Word 3: number of bytes to transfer
- Control word



- D [2:0]: type of descriptor (1D, 2D or JUMP)
- Ps [9]: MPU privilege for source access
- Pd [11]: MPU privilege for destination access
- QoS [15:12]: PIF request priority
- Twait [29]: wait for hardware trigger input to start the transfer
- Trig [30]: enable hardware trigger signal on transfer completion
- I [31]: enable interrupt on transfer completion

# iDMA 2D Descriptor

- 2D descriptor: contains eight 32-bit fields, defines one 2D memory transfer
  - Word 0 ~ 2 (same as 1D): control word, source address, destination address
  - Word 3: row size in bytes
  - Word 4: source pitch
  - Word 5: destination pitch
  - Word 6: number of rows
  - Word 7: reserved



# iDMA Configurations and Parameters

- H/W configurability (tradeoff between performance and area)
  - **Bus buffer depth (1/2/4/8/16)**
    - This buffer buffers incoming AXI data when iDMA cannot gain the write access to DataRAM.
  - **Number of outstanding 2D rows (2/4/8/16)**
    - This buffer allows iDMA engine to pre-decode a 2D request into 1D row requests, and pipeline the AXI bus requests ahead of the time.
    - Larger value helps when row size is small and memory latency is long.
- S/W parameters
  - **QoS:** request priority on AXI bus
  - **Maximum PIF block size (2/4/8/16)**
    - This parameter sets the maximum PIF/AXI block size. Larger value allows larger PIF/AXI block transfer to reduce latency, but occupies more bus bandwidth.
  - **Maximum number of outstanding PIF requests allowed (1~64)**
    - Larger value allows more outstanding PIF/AXI requests to hide bus latency.

# Simulation of an iDMA Application

- DMA behavior needs to be simulated with memory modeling enabled
  - Without memory modeling, simulator assumes immediate access of memory without latency, which gives unrealistic result
- ISS provides simple memory modeling
  - Simple parameters defining memory access latency
  - Fast way to simulate an application with DMA transfers
- XTSC supports more complex memory modeling
  - User defines more accurate model of memory and bus behaviors
  - Runs slower than ISS
  - To save simulation time, use Xtensa simulator directives to enable cycle-accurate simulation only for the code of interest

# Summary

In this module, you have learned the following

- Data movement is as important as compute kernel processing to achieve high performance
- Achieving the best performance requires a good data flow design to balance computation and DMA throughput
- Latency affects throughput if it cannot be hidden by behind processor computation
- You learned about features to hide or reduce the DMA latency
- ISS simulates DMA behavior with simple memory model
- XTSC allows simulating bus and memory behavior with more complex models

In the two next modules we will focus on SW APIs for this DMA engine.

# Quiz

1. iDMA can move data
  - a) Between local DRAM and local DRAM
  - b) Between SYSRAM and local DRAM
  - c) Both of the above
  - d) Between SYSRAM and SYSRAM
2. iDMA requires transfer start addresses that are
  - a) Aligned to the AXI/PIF bus width (16 Bytes)
  - b) Aligned to any byte boundary
  - c) Aligned to any bit boundary
3. What's the size of an iDMA 2D descriptor
  - a) 8 bytes
  - b) 16 bytes
  - c) 32 bytes
  - d) 64 bytes

Answers: 1c, 2b, 3c



# Introduction to iDMA Library

**Module** **15**

**Dependencies:** Module 13 – Introduction to iDMA

**Revision** **1.0**

**Version** **7.4**

**Estimated Time:**

- **For the lecture** **30 minutes**
- **For the lab** **N/A**

# Module Objectives

In this module, you will get an overview of the iDMA library.

- Understand the purpose of iDMA library
- Understand the API of iDMA library
- Learn how to use the API by going over a usage examples

# The Purpose of iDMAlib

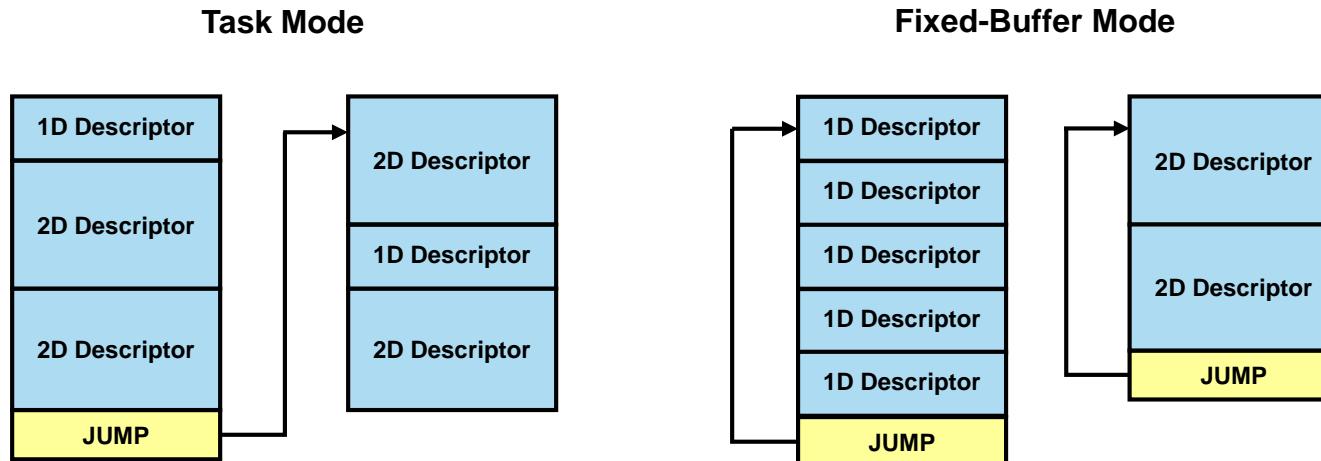
- Programming the iDMA engine requires deep understanding of the hardware, such as descriptor scheduling, registers, control flow, etc.
- It also requires careful design, especially in multi-task environments
- iDMAlib provides user a simple API for iDMA programming, such as parameter setting, descriptor scheduling, interrupt handling, status and error reporting, etc.
- iDMAlib provides a software abstraction layer for that shields the user from future hardware changes

# iDMAlib Overview

- iDMAlib is a library of low-level DMA control functions which help user to schedule and manage DMA transfers
- iDMAlib provides two modes to serve user DMA requests
  - **Task-Execution mode:** organize DMA requests as a linked list of tasks
  - **Fixed-Buffer mode:** organize DMA requests as a circular buffer
- iDMAlib supports interrupt or polling for status and errors
- iDMAlib manages DMA descriptor buffer internally
- iDMAlib API contains 3 major portions: common API, task mode API, and fixed buffer mode API
- Where to find the library
  - iDMAlib is prebuilt and included in the Vision core package
  - Library header file: {XtensaToolPath}/XtensaTools/xtensa-elf/include/xtensa
  - Library source files: {XtensaToolPath}/XtensaTools/xtensa-elf/src/libidma
  - Library binary files: {XtensaCorePath}/xtensa-elf/lib

# iDMAlib Operation Modes

- Task-Execution mode
  - Execute independent DMA requests from different threads or applications
  - One task contains one or more DMA requests (can be mix of 1D and 2D)
  - Multiple tasks can be linked into a task list
  - One request failure does not affect other requests
- Fixed-Buffer mode
  - Descriptors (of the same type) are arranged in a single circular buffer
  - Execute DMA requests sequentially through the descriptor buffer



# Common API Functions

API Function	Description
<b>IDMA_BUFFER_DEFINE</b>	The macro which defines a buffer containing descriptors.
<b>idma_init</b>	Initialize iDMAlib and iDMA hardware engine.
<b>idma_stop</b>	Stop iDMA engine.
<b>idma_pause</b>	Pause iDMA hardware.
<b>idma_resume</b>	Resume iDMA hardware.
<b>idma_sleep</b>	Put processor in sleep until an iDMA interrupt happens.
<b>idma_add_desc</b>	Add 1D descriptor to the next location.
<b>idma_add_2d_desc</b>	Add 2D descriptor to the next location.
<b>idma_get_state</b>	Check iDMA hardware state.
<b>idma_log_handler</b>	Setup logging function.
<b>idma_register_interrupts</b>	Register interrupt handlers.
<b>idma_error_details</b>	Obtain iDMA hardware error details.
<b>idma_buffer_error_details</b>	Obtain iDMA buffer error details.

# Task Mode API Functions

API Function	Description
<b>idma_init_task</b>	Initialize an iDMA task.
<b>idma_task_status</b>	Get the task execution status.
<b>idma_copy_task</b>	Create and schedule 1D copy request to a task.
<b>idma_copy_2d_task</b>	Create and schedule 2D copy request to a task.
<b>idma_schedule_task</b>	Schedule a task for execution.
<b>idma_process_tasks</b>	Trigger internal processing of completed tasks in polling mode.
<b>idma_buffer_check_errors</b>	Check if iDMA hardware is in error.
<b>idma_error_details</b>	Get error details of a failed task.
<b>idma_abort_tasks</b>	Abort all pending tasks and stop iDMA hardware after current processing descriptor.

# Example of Using Task Mode

```
IDMA_BUFFER_DEFINE(task1, 2, IDMA_1D_DESC);  
IDMA_BUFFER_DEFINE(task2, 2, IDMA_2D_DESC);
```

Allocate memory for task1 with 2 1D descriptors.  
Allocate memory for task2 with 2 2D descriptors.

```
idma_log_handler(xlog);  
idma_init(0, MAX_BLOCK_2, 16, TICK_CYCLES_8,  
100000, err_cb_func);
```

Initialize iDMA engine and set log function.  
Max PIF block size is 2, number of outstanding PIF requests is 16.

```
idma_init_task(task1, IDMA_1D_DESC, 2,  
idma_cb_func, task1);  
idma_init_task(task2, IDMA_2D_DESC, 2,  
idma_cb_func, task2);
```

Initialize task1 and task2 with callback functions.

```
idma_add_desc(task1, dst1, src1, rowSize, 0);  
idma_add_desc(task1, dst2, src2, rowSize, DESC_NOTIFY_W_INT);
```

Add 2 1D descriptors to task1. The last descriptor will trigger interrupt.

```
idma_add_2d_desc(task2, dst3, src3, rowSize,  
DESC_NOTIFY_W_INT, rows, srcPitch, dstPitch);  
idma_add_2d_desc(task2, dst4, src4, rowSize,  
DESC_NOTIFY_W_INT, rows, srcPitch, dstPitch);
```

Add 2 2D descriptors to task2. Each descriptor will trigger interrupt.

```
idma_schedule_task(task1);  
idma_schedule_task(task2);
```

Schedule task 1 and task2. task2 will be linked after task1.

```
while (idma_task_status(task2) > 0)  
    idma_sleep();
```

Wait for task2 to finish. task1 will finish before task2. If interrupt is enabled, idma\_sleep() put core into sleep and wait for interrupt. If interrupt is not enabled, idma\_sleep() returns immediately, and this will be a polling loop.

# Fixed-Buffer Mode API Functions

API Function	Description
<code>idma_init_loop</code>	Initialize fixed-buffer allocated by IDMA_BUFFER_DEFINE.
<code>idma_schedule_desc</code>	Schedule a number of consecutive descriptors to execute.
<code>idma_schedule_desc_fast</code>	Schedule descriptors without managing the circular buffer, assuming no wrap-around of the descriptor buffer.
<code>idma_copy_desc</code>	Add a 1D descriptor to the fixed buffer and schedule it.
<code>idma_copy_2d_desc</code>	Add a 2D descriptor to the fixed buffer and schedule it.
<code>idma_desc_done</code>	Check if the descriptor of the specified index is finished.
<code>idma_update_desc_dst</code>	Update the destination field of the next descriptor.
<code>idma_update_desc_src</code>	Update the source field of the next descriptor.
<code>idma_update_desc_size</code>	Update the transfer size field of the next descriptor.
<code>idma_buffer_status</code>	Get buffer execution status.

# Fixed-Buffer Example

```
IDMA_BUFFER_DEFINE(dmaBuffer, 4, IDMA_2D_DESC);
idma_init(0, MAX_BLOCK_2, 16,
TICK_CYCLES_2, 100000, err_cb_func);
idma_log_handler(xlog);

idma_init_loop(dmaBuffer, IDMA_2D_DESC,
2, &ctx, idma_cb_func);
idma_add_desc(dmaBuffer, dst[0], src[0],
rowSize, DESC_NOTIFY_W_INT);
idma_add_desc(dmaBuffer, dst[1], src[1],
rowSize, DESC_NOTIFY_W_INT);

for (i = 0; i < numBlocks; i++) {
    bufIndex = i & 1;
    idma_schedule_desc(1);
    while (idma_buffer_status() > 0)
        idma_sleep();
    ConsumeBuffer(&dst[bufIndex]);
}
```

Allocate memory for the fixed-buffer named dmaBuffer, with depth of 4 and type of 2D descriptor.

Initialize iDMA, with maximum PIF block size of 2, number of outstanding PIF request buffer of 16.

Setup iDMA logging function.

Initialize fixed-buffer dmaBuffer, with depth of 2 and type of 2D descriptor. It also set idma\_cb\_func as the callback function, and ctx as the callback data.

Add 2 descriptors to the fixed buffer. Source and destination point to the double buffers.

Loop for all data blocks.

Double buffer index is toggled for every loop.

Schedule the next descriptor in the fixed buffer.

Enter sleep mode to wait for DMA transfer done. Core will be waked up by iDMA interrupt.

Consume the data.

# Summary

This module gave you

- An overview of the iDMAlib and the API functions
- Examples of using iDMAlib API functions in different modes

In the next module we introduce you to a high level DMA SW API (Tile Manager) which is implemented on top of the iDMA Library.

# Quiz

1. Using the iDMA Library API is
  - a) encouraged as it shields the programmer from future changes to the HW engine and register field assignments
  - b) discouraged, since the API has high SW overheads and better performance can be achieved by programming the iDMA HW registers directly
2. The `idma_init` call is required
  - a) The first time your application is calling the API to initialize internal buffer states
  - b) Each time before you issue a DMA request
  - c) Can be used to ensure that all outstanding DMA requests have been processed
3. The fixed-buffer mode organizes iDMA descriptors
  - a) as a linked-list
  - b) as a circular buffer
  - c) for random access

Answers: 1a, 2a, 3b



# Introduction to DMA Tile Manager

**Module** **16**

**Dependencies:** Module 14 – Introduction to iDMA Library

**Revision** **1.0**

**Version** **7.4**

**Estimated Time:**

- **For the lecture** **30 minutes**
- **For the lab** **30 minutes**

# Module Objectives

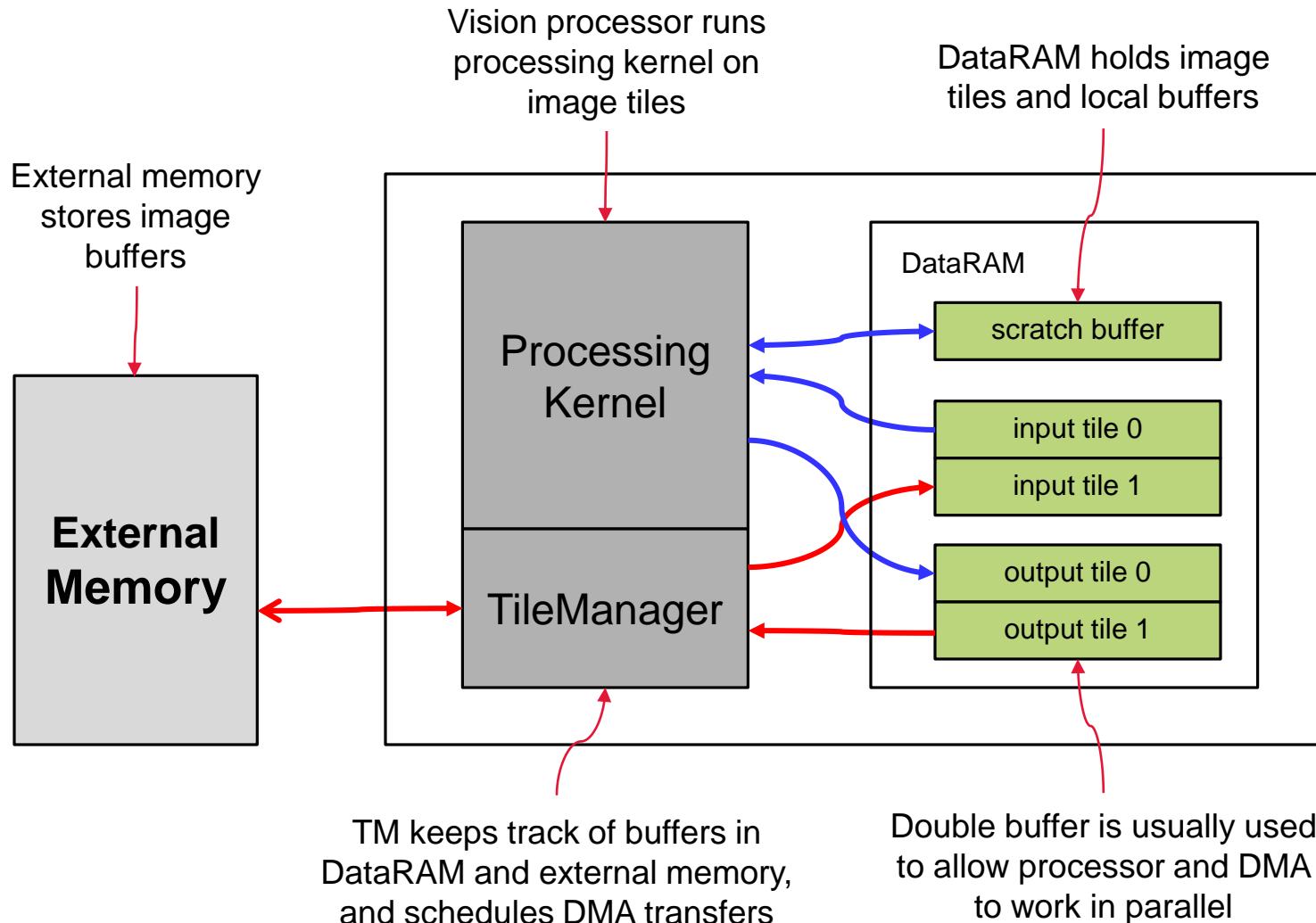
In this module, you will learn the following about the Tile Manager (TM)

- Why we need the Tile Manager library for vision applications
- Basic concepts about the Tile Manager design
- Overview of the Tile Manager API
- Usage examples for the Tile Manager API

# What issues does the Tile Manager address?

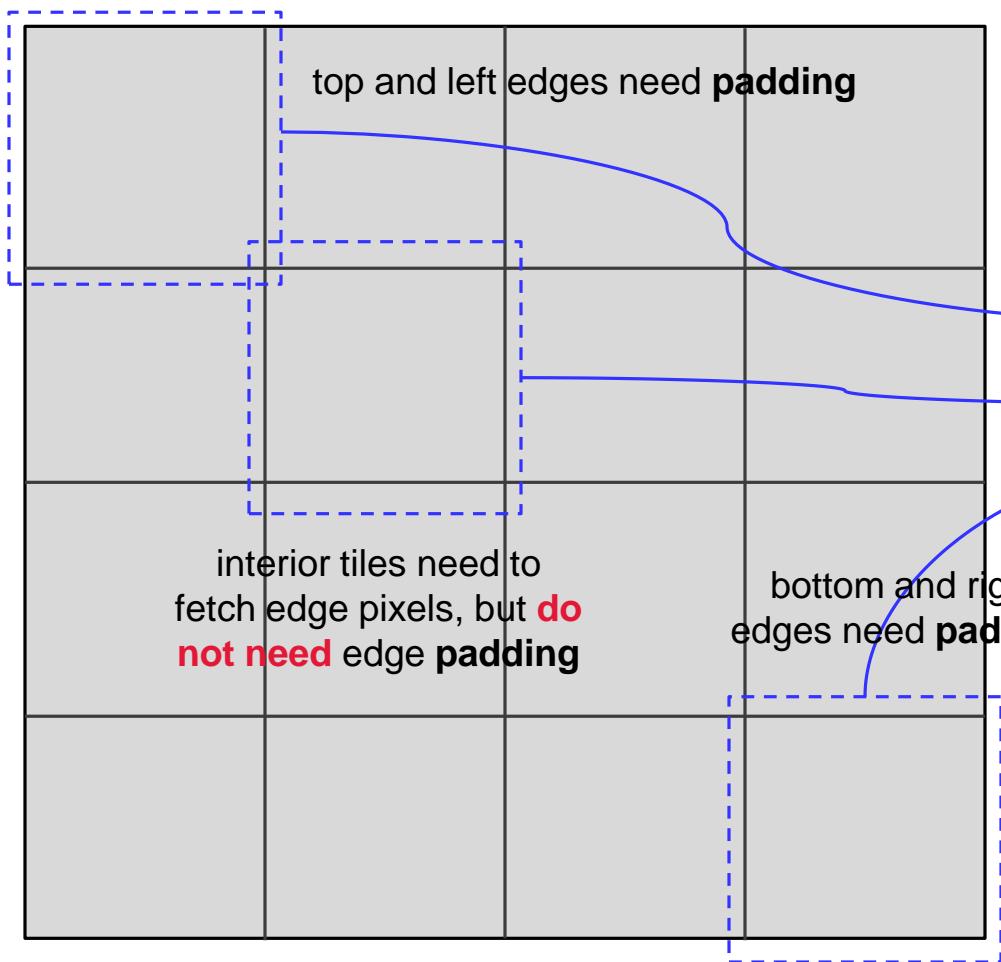
- In most vision applications the entire image is too large to fit into local DataRAM
- A common solution is to divide the image into sub-regions (**Tiles**) that can be loaded into DataRAM and processed by vision processor
- Many processing kernels require extra pixels around the edge of effective region, so need special **edge handling**
- Code is required to manage data buffers in DataRAM and external memory
- Code is required to manage and schedule DMA transfers
- Code is required to maintain data synchronization between Vision core and iDMA engine
- The TileManager is a library that assists the user with those issues

# A Typical Vision Application with DMA Transfers

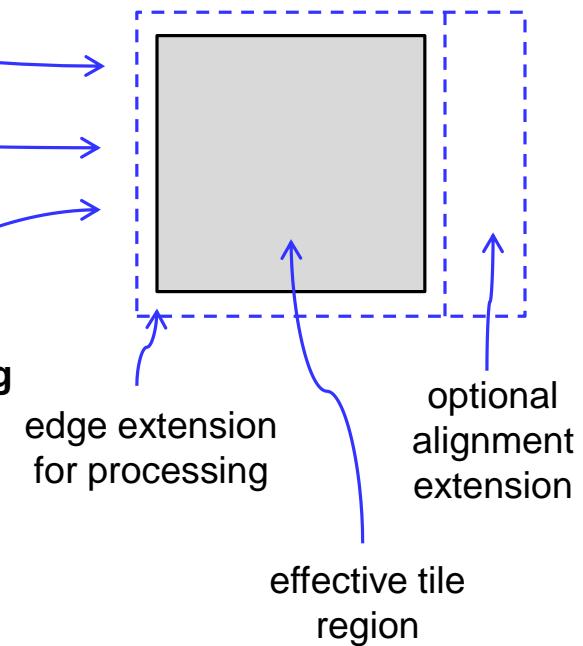


# Image and Tile Layout

Image frame layout in external memory



Tile layout  
in DataRAM



# Tile Manager Overview

- Tile Manager is part of the Vision DSP Software Package
- The major components of Tile Manager
  - **iDMA object (pdmaObj)**: it's the iDMAlib buffer object which is used to schedule iDMA transfers using iDMAlib
  - **DataRAM memory manager (memBankMgr[])**: used to manage memory pools in DataRAM. One memory manager corresponds to one DataRAM. The memory manager itself is implemented in libxmem.
  - **DataRAM pools (pMemBankStart[])**: physical memory pools in DataRAM. One pool corresponds to one DataRAM.
  - **Frame buffers (frameArray[])**: an array of frame buffers, which map the frame buffers in external memory
  - **Tile buffers (tileArray[])**: an array of tile buffers, which map the tile buffers in DataRAM
  - **Processing tile queue (tileProcQueue[])**: the queue of tiles that are scheduled to be processed

# Image Frame Structure

- xvFrame is defined to manage image buffers in external memory

```
typedef struct xvFrameStruct {  
    void      *pFrameBuff;      // pointer of frame buffer  
    uint32_t   frameBuffSize;  // frame buffer size in bytes  
    void      *pFrameData;     // pointer of active data  
    uint16_t   frameWidth;    // frame width in pixels  
    uint16_t   frameHeight;   // frame height in pixels  
    uint16_t   framePitch;    // frame row pitch in bytes  
    uint8_t    pixelRes;      // pixel resolution in bytes  
    uint8_t    numChannels;   // number of data channels  
    uint8_t    leftEdgePadWidth; // left edge width in pixels  
    uint8_t    topEdgePadHeight; // top edge height in pixels  
    uint8_t    rightEdgePadWidth; // right edge width in pixels  
    uint8_t    bottomEdgePadHeight; // bottom edge height in pixels  
    uint8_t    paddingType;    // edge padding type, 0=zero, 1=edge  
    uint8_t    paddingVal;     // edge padding value  
} xvFrame, *xvpFrame;
```

# Array Structure

- TM defines xvArray to manage local buffers in DataRAM
- It can be a local memory unit that a processing kernel runs on
- It is also the building block of tile data structure

```
#define XV_ARRAY_FIELDS \
    void      *pBuffer;      \      => pointer of array buffer
    uint32_t bufferSize;    \      => size of array buffer in bytes
    void      *pData;        \      => pointer of effective data
    int32_t   width;         \      => array width
    int32_t   pitch;         \      => array pitch
    uint32_t  status;        \      => status flag
    uint16_t  type;          \      => data type
    uint16_t  height;        \      => array height

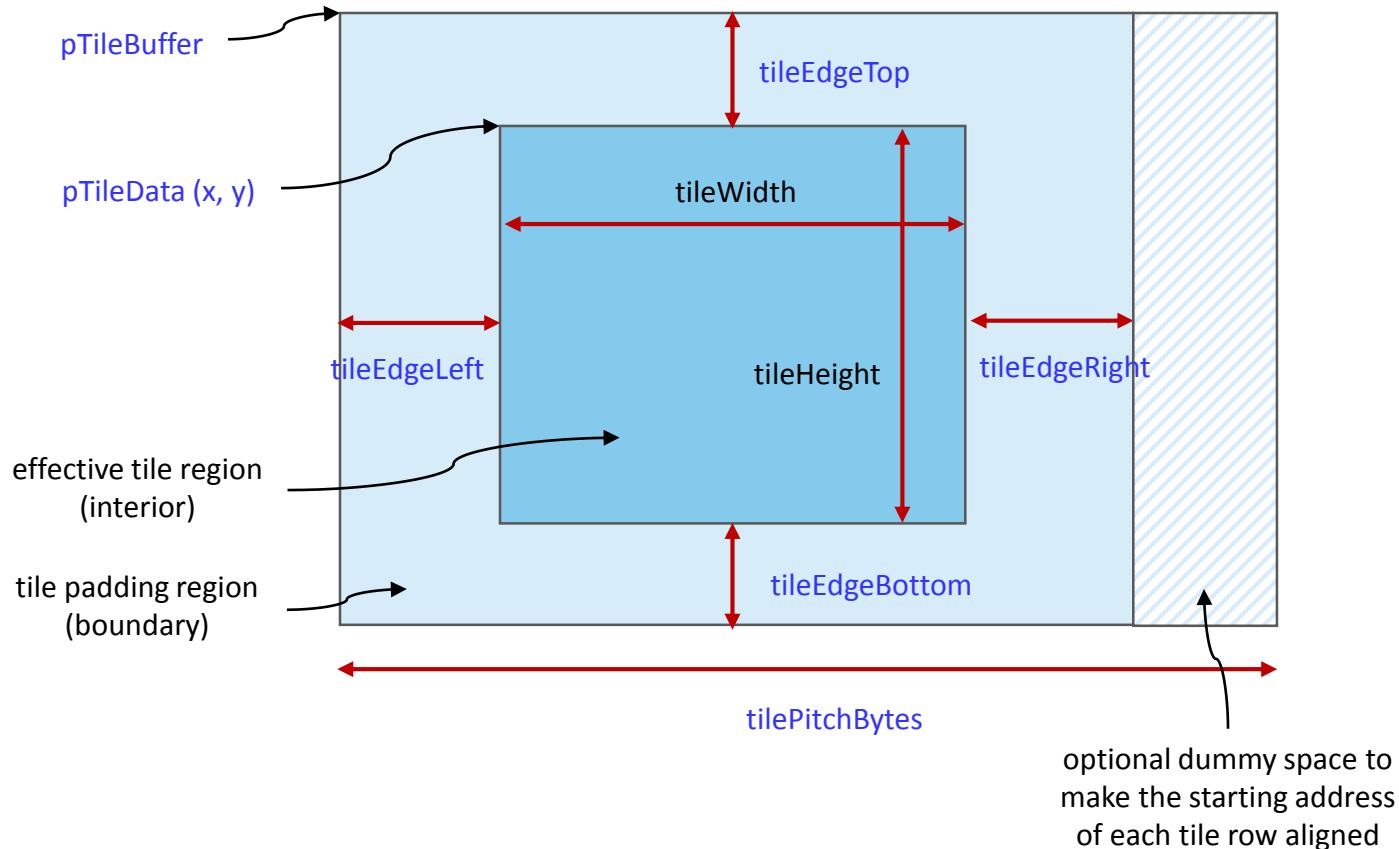
typedef struct xvArrayStruct {
    XV_ARRAY_FIELDS
} xvArray, *xvpArray;
```

# Tile Structure

- xvTile is defined to manage image tiles in DataRAM
- It is the main memory unit of DMA transfer
- It is the main memory unit that a processing kernel runs on

```
typedef struct xvTileStruct {  
    XV_ARRAY_FIELDS          // data fields from array structure  
    xvFrame     *pFrame;      // pointer of corresponding frame structure  
    int32_t      x;           // tile's x coordinate in frame  
    int32_t      y;           // tile's y coordinate in frame  
    uint16_t     tileEdgeLeft; // left edge width in pixels  
    uint16_t     tileEdgeTop;  // top edge height in pixels  
    uint16_t     tileEdgeRight; // right edge width in pixels  
    uint16_t     tileEdgeBottom; // bottom edge height in pixels  
    int32_t      dmaIndex;    // index used to track DMA descriptor  
    int32_t      reuseCount;  // reuse count used to track tile's life cycle  
    struct xvTileStruct *pPrevTile; // pointer to previous tile in a linked-list  
} xvTile, *xvpTile;
```

# Memory Layout of 2D Tile in DataRAM



# Tile Edge Handling

- In many cases, the processing function requires input pixels that are beyond the tile boundary
  - E. g., a 5x5 Gaussian filter requires 2 extra input pixels to the left, right, bottom and top of each pixel being filtered
- For **internal tiles** (tiles do not touch image boundary), tile edge extensions are from image data, edge padding is not required
- For **boundary tiles** (tiles touch image boundary), tile edge extensions require padding
- Tile Manager supports three modes of edge padding: zero extension, constant value extension, and edge pixel replication
- Two adjacent tiles' edge extension share overlapped areas. Tile Manager supports local Data RAM fetching from overlapped area of neighbor tile, instead of fetching from external memory.

# Tile Manager API Overview

- Tile Manager provides following API categories
- Initialization API
  - iDMA initialization
  - Tile Manager initialization
  - DataRAM memory manager initialization
- Buffer management API
  - Image buffer allocation/de-allocation in external memory
  - Array/tile buffer allocation/de-allocation in DataRAM
- DMA scheduling and synchronization API
  - Input/output tile DMA scheduling
  - Direct DMA transfer scheduling
  - Wait/check DMA transfer status
- Error handling API

# TileManager API Functions

Category	API Functions	Function Behavior
Initialization	<b>xvInitIdma</b>	Initialize iDMA engine
	<b>xvInitTileManager</b>	Initialize tile manager
	<b>xvResetTileManager</b>	Free all the buffers and reset TM data structure
	<b>xvInitMemAllocator</b>	Initialize DataRAM memory management
Buffer management	<b>xvAllocateBuffer</b>	Allocate a buffer in local DataRAM for tile
	<b>xvFreeBuffer</b>	Free a buffer allocated by <b>xvAllocateBuffer</b>
	<b>xvFreeAllBuffers</b>	Free all local buffers in the TileManager
Frame management	<b>xvAllocateFrame</b>	Allocate a frame from TileManager
	<b>xvFreeFrame</b>	Free a frame and return it to TileManager
	<b>xvFreeAllFrames</b>	Free all frames and return them to TileManager
Tile management	<b>xvAllocateTile</b>	Allocate a tile from TileManager
	<b>xvFreeTile</b>	Free a tile and return it to TileManager
	<b>xvFreeAllTiles</b>	Free all tiles and return them to TileManager

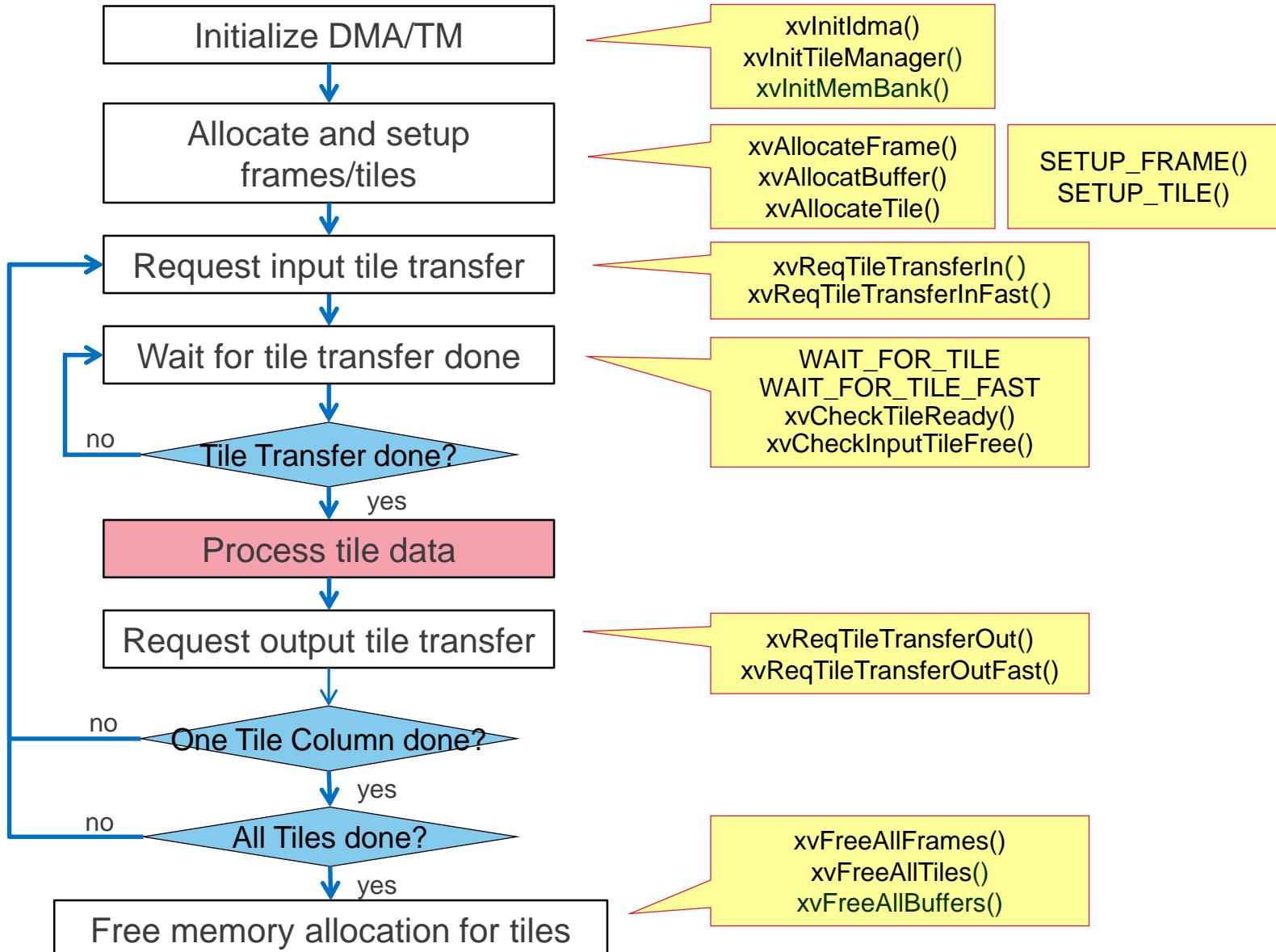
# TileManager API Functions (continue)

Category	API Functions	Function Behavior
Transfer request	<b>xvAddIdmaRequest</b>	Add an iDMA transfer request
	<b>xvReqTileTransferIn</b>	Request an input tile transfer
	<b>xvReqTileTransferInFast</b>	Fast version of input tile transfer request
	<b>xvReqTileTransferOut</b>	Request an output tile transfer
	<b>xvReqTileTransferOutFast</b>	Fast version of output tile transfer request
Check status	<b>xvCheckForIdmaIndex</b>	Check if a DMA transfer is done
	<b>xvCheckTileReady</b>	Check if a tile transfer is done
	<b>xvCheckInputTileFree</b>	Check if an input tile is free
Other	<b>xvPadEdges</b>	Pad tile edges
	<b>xvGetErrorInfo</b>	Get detailed error information

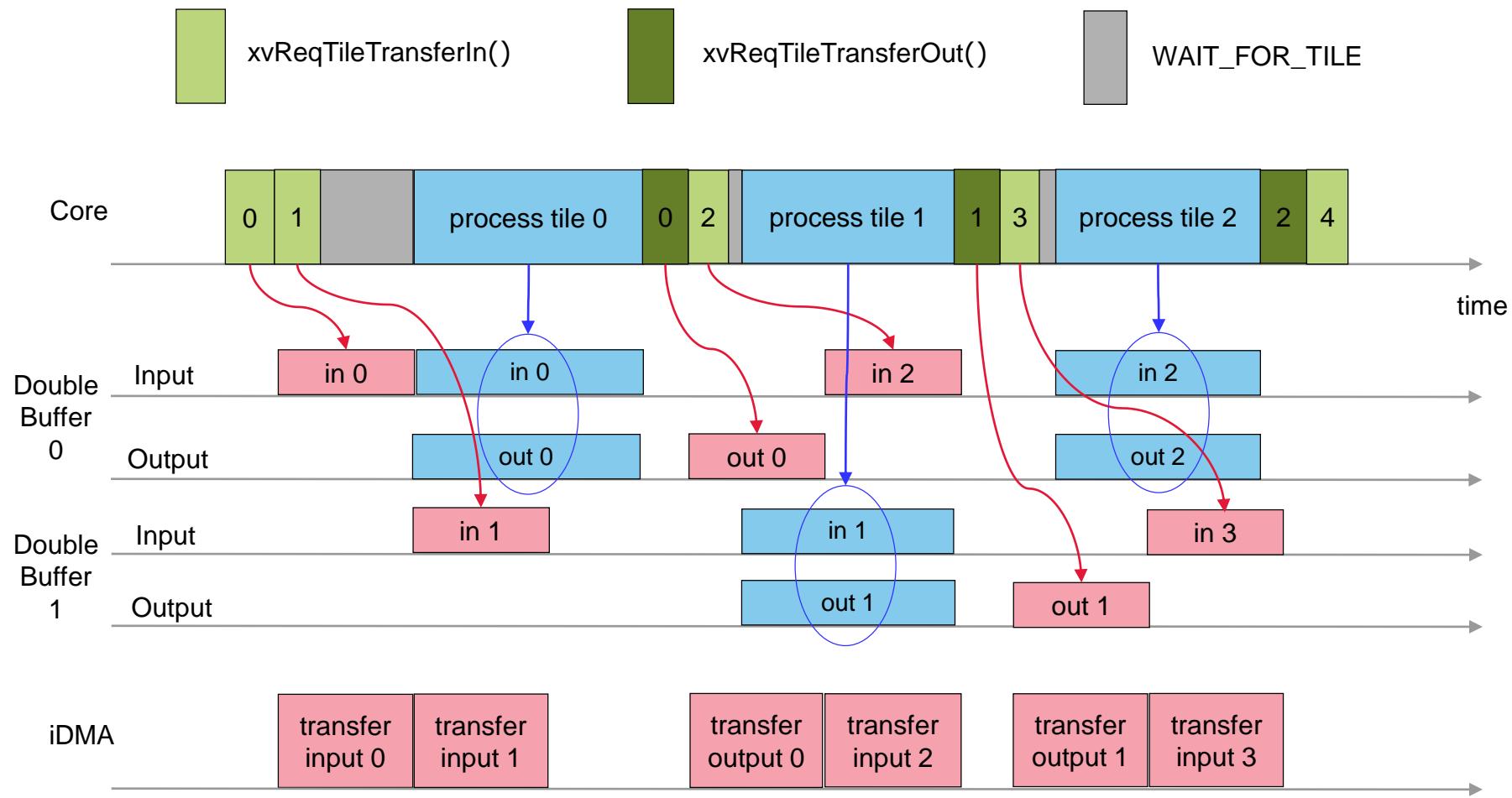
# TileManager API Macros

API Macro	Function Behavior
<b>SETUP_TILE</b>	Setup a tile with tile parameters
<b>SETUP_FRAME</b>	Setup a frame with frame parameters
<b>WAIT_FOR_TILE</b>	Wait for a tile transfer is done
<b>WAIT_FOR_DMA</b>	Wait for a DMA transfer is done
<b>WAIT_FOR_TILE_FAST</b>	Fast version of waiting for a tile transfer done
<b>XV_TILE_UPDATE_EDGE_HEIGHT</b>	Update edge height of a tile
<b>XV_TILE_UPDATE_EDGE_WIDTH</b>	Update edge width of a tile
<b>XV_TILE_UPDATE_DIMENSIONS</b>	Update dimensions of a tile
<b>XV_IS_TILE_OK</b>	Check parameter consistency of a tile

# TM Function Flowchart and API Functions



# Double Buffering Timing Diagram



# Summary

In this module, you learned the following about Tile Manger (TM)

- How the TM can help you to develop applications using DMA
- Memory layout of frame and tile
- Main data structures and API functions of TM
- Example of using TM APIs

This concludes the training portion about DMA.

In the last module you will we introduce you to our imaging library XILib.

# Lab



## Lab 4: Tile Manger

- **Understand Tile Structure**
- **Understand Tile Manger API**
- **Build and Profile Tile Manger Application**

# Quiz

## 1. The TM API

- a) should address all of your data movement needs and we don't expect that you would ever need the iDMA Lib API once you understand how to use the TM
- b) should address most of your data movement needs and we expect that you would need the iDMA Lib occasionally for special cases
- c) Is build on top of the iDMA Lib API
- d) b) and c)

## 2. When defining input tiles for a 7x7 Gaussian filter, the tileEdge{Left,Right,Top,Bottom} need to be set to at least

- a) 1
- b) 2
- c) 3
- d) 4

Answers: 1d, 2c



# Introduction to XI Library

**Module** **17**

**Dependencies:** Module 13 – Using DMA in your Application

**Revision** **1.0**

**Version** **7.4**

**Estimated Time:**

- For the lecture **30 minutes**
- For the lab **60 minutes**

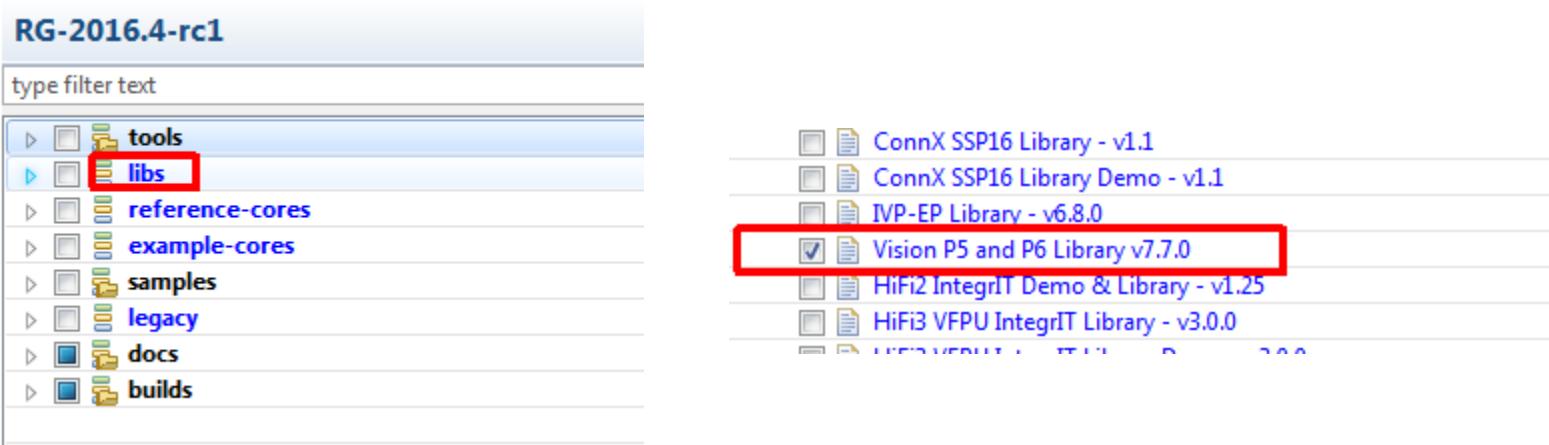
# Module Objectives

In this module, you will get an overview of XI Library. You will

- Get an overview of the categories of supported functions
- Become familiar with the tile concept and data structure
- Use the XI Library in an application example
- Use the XI Library performance spreadsheet to plan the performance of your application
- Get and idea about achieved performance when using the XI Library versus programming the kernels on your own

# What is XI Library

- XI Library contains a wide range of kernels of image and video processing, computer vision etc.
- XI Library kernels are optimized for Vision Processor family
- XI Library allows user to focus more on algorithm development, without spending much efforts on optimization details
- User can build big applications quickly with XI Library kernels
- XI Library is released through XPG(RG-2016.4/libs/Vision P5 and P6 Library v7.7.0)



# XI Library Deliverables

XILib is delivered as a workspace (current version 7.7.0)

- **libxi:** IVP optimized source code for imaging/vision kernels.
- **libxi\_ref:** Source code for reference kernels.

These are base routines for IVP optimized kernels.

**Most of the IVP kernels** are bit exact with definition of XI\_ENABLE\_BIT\_EXACT\_CREF enabled for this project.

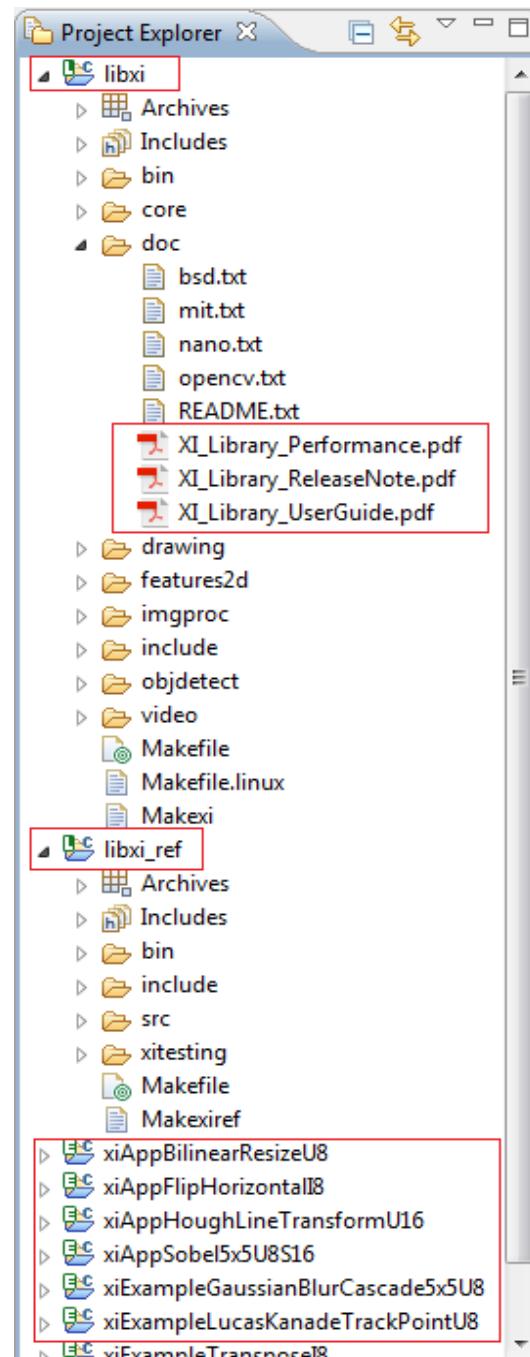
- **Release Notes** (XI\_Library\_ReleaseNote.pdf)
- **Performance Summary** (XI\_Library\_Performance.pdf)

Performance table for all the kernels with various test cases.

- **User Guide** (XI\_Library\_UserGuide.pdf)

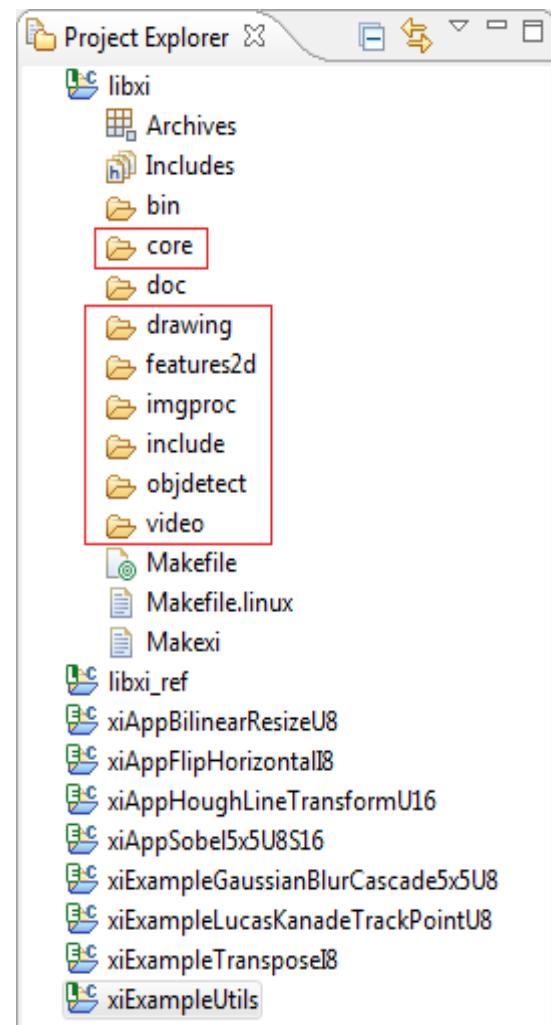
Information about usage(prototypes, temporary buffer requirement if any, description about i/o parameters) of XI lib kernels.

- **Example Projects** for using XI lib kernels



# XI Library Modules

- **core:**
  - low-level support functions
  - library-specific data types
- **imgproc:** Most commonly used image processing kernels
- **video:** video processing kernels
- **features2d:** Harris corner/Shi-Tomasi features etc.
- **objdetect:** Haar, HoG etc.
- **drawing:** utilities to draw objects on image



# XI Library Core Module

- Primitive types
  - Error type: XI\_ERR\_TYPE
  - Fixed point 16 bit formats: XI\_Q15, XI\_Q5\_10, XI\_Q8\_7 etc.
  - Fixed point 32 bit formats: XI\_Q1\_30, XI\_Q13\_18, XI\_Q15\_16 etc.
- Data structures
  - Definitions of special data structures used in XI lib  
For ex: xi\_point, xi\_affine, xi\_perspective etc.
- Error constants
  - XI\_ERR\_OK, XI\_ERR\_IALIGNMENT, XI\_ERR\_OALIGNMENT etc.
- Utility functions
  - xiExtendEdges, xiCopyTile, xiFillTile etc.
- Unary element-wise operations
  - XiAbs, xiBitwiseNot, xiClip, xiNeg etc.
- Binary element-wise operations with scalar value
  - xiAbsdiffScalar, xiAddScalar, xiBitwiseAndScalar etc.

# XI Library Core Module (2)

- Binary element-wise operations
  - xiAbsdiff, xiAdd, xiAddWeighted, xiBitwiseAnd etc.
- Reduction operations
  - xiCountNonZero, xiCountEqual, xiCountGreater, xiMeanStdDev etc.
- Vector operations
  - xiCartToPolar, xiLog, xiMagnitude, xiPhase etc.
- Miscellaneous operations
  - xiExtractChannel(interleaved to planar), xiLUT, xiMerge(multiple single channels merged into single), xiSort etc.

# XI Library Image Processing Functions

- Image convolution functions
  - xiBoxFilter: Box filter with ksize
  - xiCanny: Find edges in tile using Canny algorithm
  - xiSobel: First order x and y derivatives using 3x3/5x5 Sobel Operator
  - xiDilate/xiErode: Dilation and Erosion
  - xiGaussianBlur: Blur using Gaussian Kernel
  - xiMedianBlur: Blur using 3x3 median filter
  - xiPyrUp/xiPyrDown: Upsampling and downsampling by 5x5 Gaussian Kernel
- Geometric transformations
  - xiFlipHorizontal: Flip tile in horizontal direction
  - xiFlipVertical: Flip in vertical direction
  - xiTranspose: transpose of a tile
  - xiWarpAffineBilinear: Affine warp using bilinear interpolation
  - xiWarpPerspectiveNearest: Perspective warp using Nearest neighbor interpolation

# XI Library Image Processing Functions (2)

- Structural analysis
  - `xiConnectedComponents`: Finds and labels connected components in a tile
- Miscellaneous transformations
  - `xiCalcHist`: Histogram calculation
  - `xiCvtColor`: Color space conversion
  - `xiEqualizeHist`: Histogram equalization of tile
  - `xiHoughCirclesTransform` : Circles' center in a tile using Hough transform
  - `xiIntegral`: Integral of an array
  - `xiThreshold`: Fixed level threshold to every element

# XI Library Feature2D Functions

- Feature detection
  - xiCornerHarris : Harris edge detector algorithm
  - xiCornerShiTomasi: Shi-Tomasi response map for corner detection algorithm
  - xiExtractKeyPoints: Extracts +ve elements of tile
  - xiExtractPoints: Extracts elements of tile with values bigger than a given threshold
  - xiFAST: Detects corners using FAST algorithm
- Descriptor extraction
  - xiBRIEF : Computes BRIEF descriptor
- Descriptor Matching
  - xiKnnSearch: Assuming every line of query and train tiles are feature descriptor, searches one or two nearest neighbors among lines of train trail for each line of query in the sense of L1, L2 or Hamming norm correspondingly.

# XI Library Video Processing Functions

- Motion analysis
  - xiAccumulate:  $dst(i, j) = dst(i, j) + src(i, j)$  if  $mask(i, j) \neq 0$
  - xiAccumulateImage:  $dst(i, j) = dst(i, j) + src(i, j)$ , with saturation
  - xiAccumulateProduct:  $dst(i, j) = dst(i, j) + src1(i, j) * src2(i, j)$  if  $mask(i, j) \neq 0$
  - xiAccumulateScalar:  $dst(i, j) = dst(i, j) + val$
  - xiAccumulateSquare:  $dst(i, j) = dst(i, j) + src(i, j)^2$  if  $mask(i, j) \neq 0$
  - xiAccumulateSquareImage:  $dst(i, j) = dst(i, j) + (src(i, j)^2 \gg shift)$
  - xiAccumulateWeighted:  $dst(i, j) = (1 - \alpha) * dst(i, j) + \alpha * src(i, j)$  if  $mask(i, j) \neq 0$
  - xiAccumulateWeightedImage:  $dst(i, j) = (1 - \alpha) * dst(i, j) + \alpha * src(i, j)$
  - xiCalcOpticalFlowBM: Optical flow using block matching.
- Background subtraction
  - xiBS\_MOG: Implements Gaussian Mixture-based Background/Foreground Segmentation algorithm

# XI Library Object Detection Functions

- xiHaarStump: Detects objects using trained cascade classifier based on Haar features (weak classifiers are stumps).
- xiHOGBinPlanes, xiHOGGradient, xiHOGNormalize, xiHOGWeighted: For Histogram of oriented gradients
- xiRANSAC\_line: Attempts to estimate parameters of a line model from a set of input data using the RANSAC algorithm:  $a * x + b * y + c = 0$
- xiRANSAC\_affine: Estimate parameters of affine transform between two sets of points using RANSAC algorithm

$$x_{to} = a_{11} * x_{from} + a_{12} * y_{from} + xt,$$

$$y_{to} = a_{21} * x_{from} + a_{22} * y_{from} + yt$$

- xiSVMPredict: Predicts the support vector machines (SVM) response for input sample.

# XI Library Function naming convention

- Why do we need to understand naming convention?
  - XI lib has many optimized kernels(non generic)which are written to support various i/o data types, buffer alignment etc.
  - Since there are large number of kernels in XI lib supported. The number of kernels optimized for various data types/buffer alignment are also high.
  - In order to identify the best kernel based on given input conditions it is important to understand naming convention for XI lib kernel.
  - For example: xiAdd has following versions
    - xiAdd\_I8
    - xiAdd\_I8A2
    - xiAdd\_U8S16
    - xiAdd\_U8S16A
    - xiAdd\_U8S16S16
    - xiAdd\_U8S16S16A
    - xiAdd\_I16
    - xiAdd\_I16A

# XI Library Function naming convention

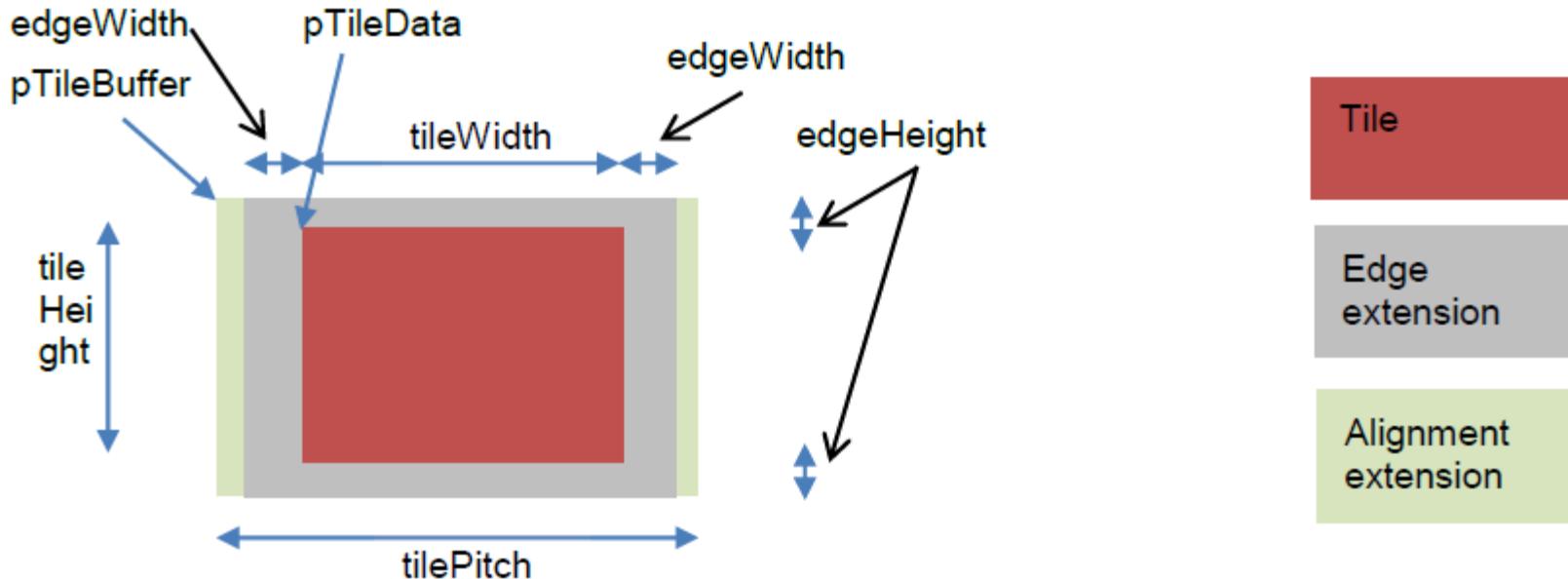
- Function Name
- S: Saturated arithmetic
- N: Output is Normalized to occupy all bits of o/p and normalization shift amount is produced
- R: Output is shifted right arithmetically by some number
- D: Output is divided by some number before storing destination
- Q: Output is multiplied by some number before storing to destination
- Input data type:
  - U8: unsigned 8 bit
  - S16: signed 16 bit
  - I32: unsigned/signed 32 bit
- Output data type: Needed only when data type is different from i/p

# XI Library Function naming convention

- Data alignment
  - <empty> (default) - arbitrary aligned inputs and outputs are supported;
  - A indicates that function requires minimal alignment for all inputs and outputs;
  - A2 indicates doubled alignment requirement for all inputs and outputs;
  - Ia indicates that function requires minimal alignment for inputs;
  - Ia2 indicates doubled alignment requirement for inputs;
  - Oa indicates that function requires minimal alignment for outputs;
  - Oa2 indicates doubled alignment requirements for outputs;
  - Sa indicates that the function requires stride-aligned inputs and outputs;
  - Ma indicates that the function requires same modulo alignment of input/output tile pointers.
- Fixed point format: Q\_15/Q\_5\_10 etc.
- Color format: YUV/RGB etc.
- Chroma Sampling grid: 4:4:4, 4:2:2, 4:2:0
- Function Specific Suffix
- Examples
  - xiBoxFilter\_3x3\_U8Oa
  - xiFlipHorizontal\_I8A2

# Tile Processing

- All XI Library kernels are **tile-based**, they process tiles in local memory
- For kernels which need the data to be processed on full image instead of tile additional steps might be required to get required results from tile o/p's.
- Tile is a sub-block of an image with necessary edge extensions
- Tile data can be transferred by TileManager



# xi\_tile Data Structure

```
typedef struct {
    void *pBuffer;
    uint32_t bufferSize;
    void *pData;
    int32_t width;
    int32_t pitch;
    uint32_t status;
    uint16_t type;
    uint16_t height;
    xi_frame *pFrame;
    int32_t x;
    int32_t y;
    uint16_t edgeWidth;
    uint16_t edgeHeight;
} xi_tile, *xi_pTile;
```

- Pointer to start of tile buffer(including edge border and alignment extension)
- Total size of tile buffer(bytes)
- Pointer to start of active tile data(excluding edge borders and alignment extension)
- Width of tile(active data in pixel)
- Pitch of tile(pixel)
- Status of DMA transfer
- Details of element size
- Tile height
- Pointer to source image
- X/Y location of top-left active tile pixel in frame
- Number of edge pixels on each side along width boundary
- Number of edge pixels on each side along height boundary

# Xi library Example Source code

```
xi_tile src_t;

//Set src tile
SETUP_TILE(&src_t,           /* tile          */
            memSrcTile,      /* tile buff ptr */
            MAX_SRCTILE_BYTES,
            pFrameDummy,     /* frame pointer */
            inTileW,          /* tile width    */
            inTileH,          /* tile height   */
            (inTileW + 2*W_EXT),
            XI_TILE_U8,       /* tile data type */
            W_EXT,            /* tile edge width */
            H_EXT,            /* tile edge height */
            0,
            0,
            alignType
);
```

# Xi library Example Source code(2)

```
xi_tile dst_t;

SETUP_TILE(&dst_t,           /* tile          */
           memDstTile,      /* tile buff ptr */
           MAX_DSTTILE_BYTES,
           pFrameDummy,     /* tile data ptr */
           outTileW,         /* tile width    */
           outTileH,         /* tile height   */
           outTileW,
           XI_TILE_U8,       /* tile data type */
           0,                /* tile edge width */
           0,                /* tile edge height */
           alignType
);
```

```

for(indy=0; indy<srcHeight; indy+=TILE_H)
{
    for(indx=0; indx<srcWidth; indx+=TILE_W)
    {
        extractTile(
            gsrc,                                /* src      */
            (uint8_t *)(XI_TILE_GET_BUFF_PTR(&src_t)), /* tile     */
            srcWidth,                            /* src pitch */
            srcHeight,                            /* src height */
            XI_TILE_GET_PITCH(&src_t),           /* tile pitch */
            XI_TILE_GET_HEIGHT(&src_t) + 2*H_EXT, /* tile height */
            indx-XI_TILE_GET_EDGE_WIDTH(&src_t), /* origin X */
            indy-XI_TILE_GET_EDGE_HEIGHT(&src_t)); /* origin Y */

        //process

        xiErr = xiFlipHorizontal_I8((xi_pArray)&src_t, (xi_pArray)&dst_t);

        moveTile8(
            (uint8_t *)(XI_TILE_GET_BUFF_PTR(&dst_t)), /* src      */
            gdst+srcWidth-indx-TILE_W+indy*srcWidth, /* dst      */
            XI_TILE_GET_WIDTH(&dst_t),             /* tile width */
            XI_TILE_GET_HEIGHT(&dst_t),            /* tile height */
            XI_TILE_GET_PITCH(&dst_t),             /* tile pitch */
            srcWidth
        );
    }
}

```

# Xi library Example Source code(3)

# Sample pseudo-code for using functions from XI lib in an application

- Set up source and destination tiles
  - Set up parameters like buffer pointer, data pointer, width , pitch, height, type, edge extensions etc.
  - XI\_TILE\_SET\_BUFF\_PTR
  - XI\_TILE\_SET\_DATA\_PTR
  - XI\_TILE\_SET\_BUFF\_SIZE
  - XI\_TILE\_SET\_WIDTH
  - XI\_TILE\_SET\_HEIGHT
  - XI\_TILE\_SET\_PITCH
  - XI\_TILE\_SET\_TYPE
  - XI\_TILE\_SET\_EDGE\_WIDTH
  - XI\_TILE\_SET\_EDGE\_HEIGHT

# Sample pseudo-code for using functions from XI lib in an application(2)

- Set up temporary tile/memory (required if any) for some APIs
- Copy the data from input buffer(s) to source tile(s) by using memcpy/iDMA/tile manager APIs
  - memcpy
  - idma\_copy\_desc/idma\_copy\_2d\_desc/idma\_desc\_done
  - xvReqTileTransferIn/xvCheckTileReady
- Call XI lib function with appropriate parameters(src tile/dst tile/temporary tile/ function specific parameters)
  - xiErr = xiSobel\_5x5\_U8S16(&src\_t, (xi\_pArray)&dst\_dx\_t, (xi\_pArray)&dst\_dy\_t, 1);
- Copy the data from o/p tile to o/p buffer using memcpy/iDMA/tile manager APIs
  - memcpy
  - idma\_copy\_desc/idma\_copy\_2d\_desc/idma\_desc\_done
  - xvReqTileTransferOut

# How to read the XI Library Performance Summary

- Below we show and example for interpreting the performance numbers of the XiLib function XiSepFilter2D\_D\_S16
- Reported cycles are for separable 2D filter on an output tile of 64 x 64 16 bit pixels
- CPPI is cycles per input pixel CPPO is cycles per output pixel
- Filter size is N = 5, 15 or 37 coefficients
- Input size is  $(64+N-1) \times (64+N-1)$ , but output size is always 64x64
- The results is divided with the divisor 3
- No bias is applied before dividing

Function Name	Perf	Metric	Config
xiSepFilter2D_D_S16	0.6108	CPPO	(kSize: 5, TileSize: { 64, 64}, divisor: 3, bias: 0)
xiSepFilter2D_D_S16	1.4517	CPPO	(kSize: 15, TileSize: { 64, 64}, divisor: 3, bias: 0)
xiSepFilter2D_D_S16	3.5605	CPPO	(kSize: 37, TileSize: { 64, 64}, divisor: 3, bias: 0)

# Important notes for using performance numbers

- The XI Library had to make tradeoffs between generality and performance for specific use cases.
- Functions in this library are highly parameterized and a higher degree of optimization is possible when optimizing for a specific case.
- All performance numbers are generated using reference VP5/6 reference configuration (no histogram or VFPU package).
- Performance numbers indicated are generated using test framework. Numbers may be different when used in an application for following reasons
  - Error checking may be enabled
  - Different compile options/Different compiler version
  - Difference in overhead between application and test framework.
  - Difference in function parameters/tile size etc.

# Important notes for using performance numbers

- It gives range of performance numbers measured for a small set of test parameters but the range is not exhaustive of all possible conditions
- Purpose of this table is to generate **quickly** estimate, performance of kernel and application not to display overall **best** and **worst** cases
- For **accurate** estimation, kernels must be tested with realistic use cases expected in that application.
- Numbers with \* are measured by enabling memory performance modeling because they use Gather-Scatter.

# Summary

In this module, you got an overview of the XI Library. You learn about

- the categories of supported functions,
- became familiar with the tile concept,
- used the XI Library in an application example
- You also used the XI Library performance spreadsheet to plan the performance of your application

You also learned about the tradeoffs between generality and performance that the library had to make.

We expect you to be able to get good usage out of the XI Library.

However, if you have functions that are very performance critical you might be better off writing your own optimized routine that only cover the specific use case that you are interested in with loss in generality, but hopefully a performance upside.

# Lab



## Lab 5: XI Library

- Understand XI Library API
- Understand Memory Manger
- Understand Application Code using the XI Library

# Quiz

- 1. Which of the following statements about the XI Library is true**
  - a) It is highly optimized by expert programmers. If you find a function that you could use in your application than you are better off using it than writing your own.
  - b) The XI Library had to make tradeoffs between generality and performance for specific use cases. Functions in this library are highly parameterized and a higher degree of optimization is possible when optimizing for a specific case.
  - c) The availability of a large number of XI functions should accelerate your application development. At a second optimization stage you can replace highly performance critical functions with your own implementation.
  - d) b) and c)
- 2. The XI Library works with Image Tiles**
  - a) It is the programmers responsibility to ensure the input tiles in local DRAM contain the necessary data inputs. The user is encouraged to use the TM API for that purpose.
  - b) The XI library handles DMA internally and just requires pointers to input images located in SYSRAM

Answers: 1d, 2a