.

Theia architecture specification

Version 0.1
Last update Saturday, July 07, 2012
Diego Valverde 2012

Revision & Review

Revision History

Version	Description	Author(s)	Date <yyyy-mm-dd>
0.1	Initial document	Diego Valverde Garro	2011-12-19

Table of contents

Table of tables

Table 1 Acronyms and Abbreviations

Acronym	Description
ALU	Arithmetic and logical unit
IIU	Instruction Issue Unit
RF	Register File
EA	Effective Address
RTL	Register transfer level
CBUS	Commit bus
IBUS	Issue bus
IMEM	Instruction memory
RS	Reservation Station
EU	Execution Unit
IM	Instruction Memory
SPR	Special Purpose Register
Qm.n	Fixed point number with n decimal bits a m integer bits
LUT	Lookup Table
ILP	Instruction level parallelism
TLP	Thread level parallelism
PC	Program counter
CP	Control Processor
VP	Vector Processor
CCB	Control Command Bus
OOO	Out of order

Table 2 Reference Documents

References	Description
TH-LS001	T Language Specification <URL>

THIS PAGE IS INTENTIONALLY LEFT BLANK

1. Introduction

THEIA is a multi-thread, multicore, vector graphic processing unit (GPU). The idea of the THEIA project is to provide an open source environment including functional RTL, test bench environment and an open source high level programming language/compiler called T-Language.

The present document is dedicated to describe and specify the hardware architecture of the THEIA GPU system and related hardware subsystems.

The THEIA hardware is described using RTL (register transfer level), written in Verilog 2001 HDL. In order to perform a full RTL simulation, the HDL model needs a series of input files which represent the various input parameters and the binary representation of the user code (written in T-Language or in THEIA-Assembly language).

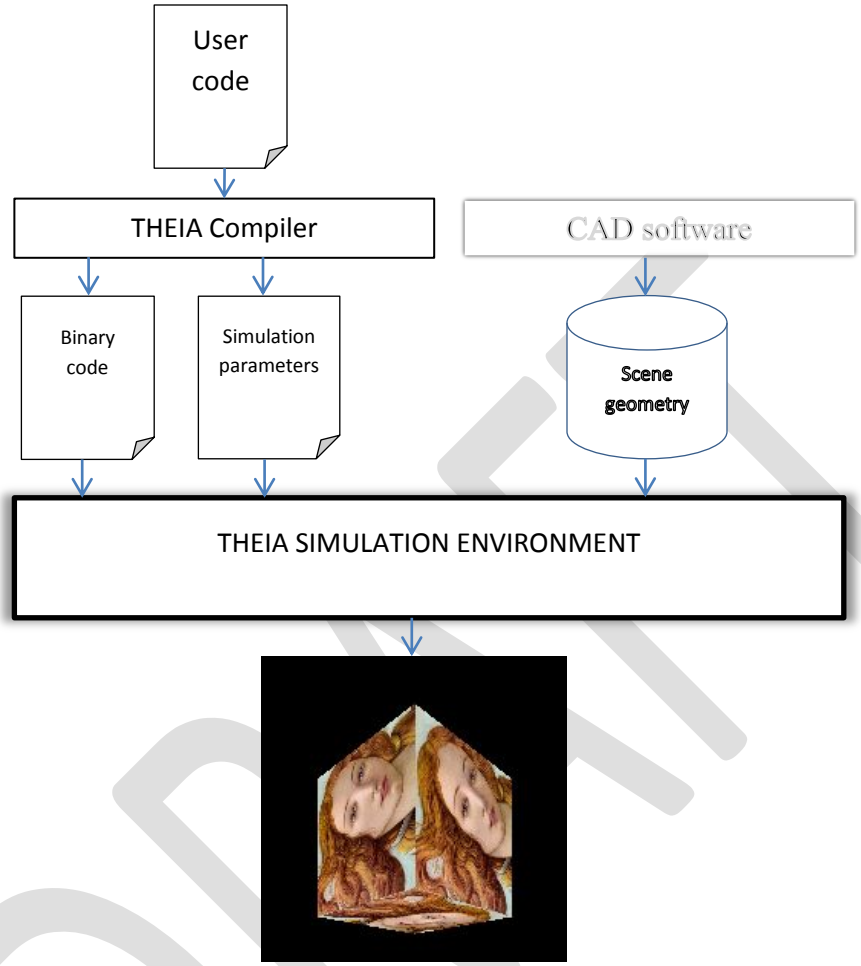

Figure 1 THEIA environment overview

The outputs from an RTL simulation are a series log files and the actual graphical representation of the rendered image in a format which can be opened using a standard image editor such as GNU Gimp.

Even if the hardware architecture of the THEIA GPU is designed to be efficient in 3D computer graphic related tasks, due to the flexibility of the system and the programming environment, a myriad of other applications that can benefit from vector processing and parallel processing are also possible.

1.1. Vector processing

One of the interesting features of the THEIA GPU is the ability to handle vector operations. Each single instruction can operate on vectors of data. Each element of the input data vector is fetched consecutively by the corresponding execution unit in a pipelined fashion as illustrated in the next figure.

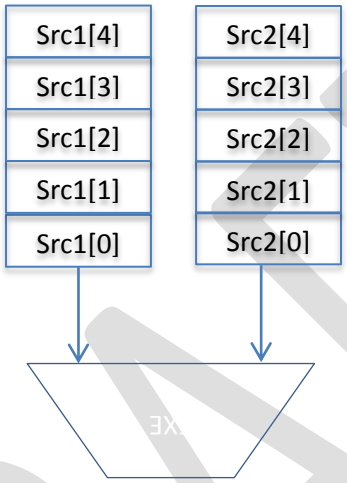

Figure 2 Data pipeline in an execution unit

Each vector functional unit is a separate and fully pipelined execution unit that and most execution units can start a new operation every clock cycle. Therefore, each vector functional unit is effectively a data pipeline. Furthermore each execution unit has 3 “data lanes” thus being able to simultaneously process 3 array elements every clock cycle.

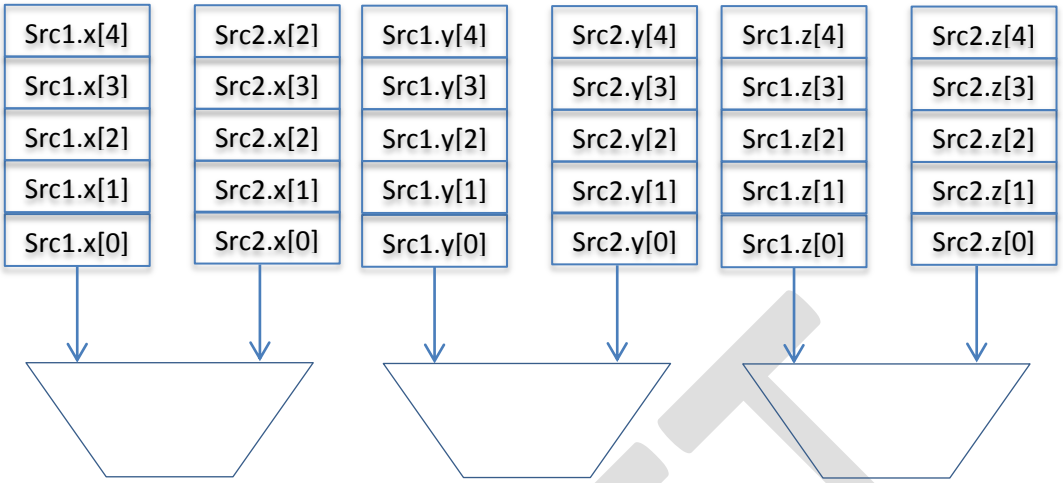

Figure 3 X, Y and X data vector data lanes

Also each GPU core has a large register file where the data is guaranteed to be located in consecutive memory positions[1].

1.2. Combining Vector processing and out-of-order execution.

The execution time of the vector operations primarily depends on the length of the vectors, but also depends on the structural hazards and data dependencies. In order to obtain more instruction level parallelism, the vector operations are combined with an out-out-of-order execution model. By executing the vector operations in an out-of-order fashion, the data dependencies can be minimized and a better performance is obtained.

The notion of "convoy" from [1] is defined as a series of vector instructions that can potentially execute together and the performance of a section of code can be estimated by counting the number of convoys. By introducing the OOO technique, these convoys are not limited to instructions that are sequential in the program flow therefore the performance of the program can be increased.

Vector "Chaining" allows the results from a vector functional unit to be forwarded to a second functional unit which has data dependency on the first one. By using chaining, a convoy which depends on the results from a previous convoy can be chained together into a single convoy.

[1] This is done by software, at the Control Processor (CP) level.

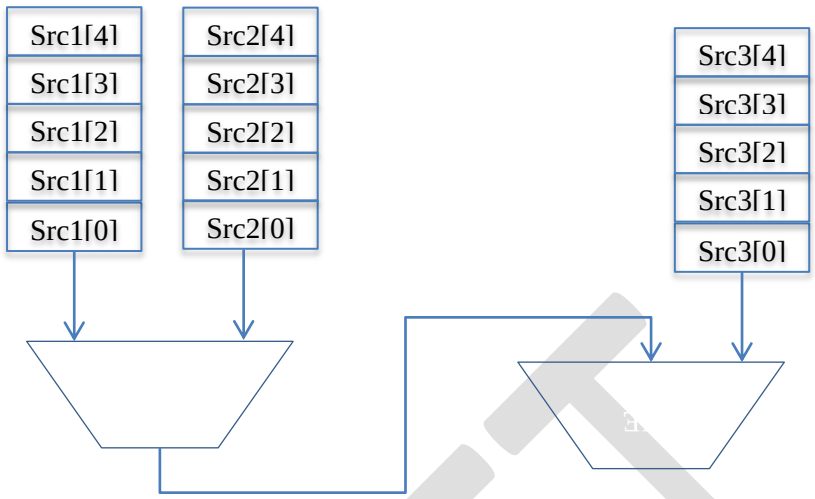

Figure 4 Convoy chaining

THEIA extends the data forward of execution results from the OOO model in order to implement "chaining" for the vector operations.

The details of the out-of-order engine are described later on this document.

2. System Overview

THEIA is a multi-thread, multicore, vector graphic processing unit (GPU). The THEIA GPU is comprised of different hardware blocks that interact with other in order to render an image frame.

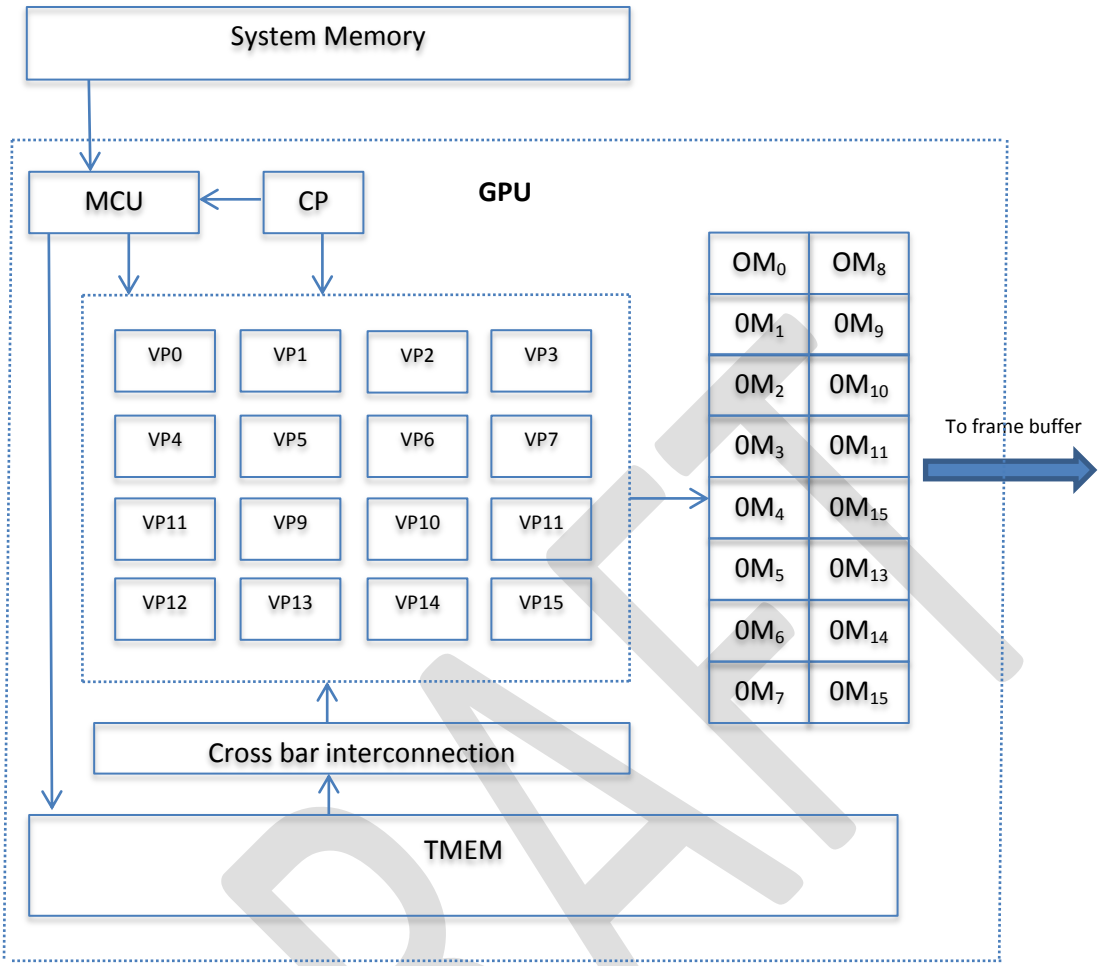

Figure 5 The GPU simplified block diagram

Figure 5 presents the GPU main functional blocks and also an external memory called “Main Memory” that is outside of the GPU. The Main memory is a large RAM that is used as a repository where the textures, code, geometry, etc. can be stored. The contents of the Main Memory are read only from the GPU stand point and can only be accessed through the internal Memory Control Unit (MCU). The MCU is controlled by the Control Processor Unit (CP).

The CP block is responsible to control and monitor the global execution of the GPU. The CP is a simple programmable unit which allows the user to programmatically control the resource allocation and the workload distribution of the GPU. The CP can command the MCU to copy execution code and data to one or more “Vector Processing Units” (VPs) at any given point in time. The CP can also request special actions from one or more VPs by sending special commands directly to the VPs using a dedicated

"Control Command Bus" (CCB). By using the MCU and the CCB, the software running on the CP effectively distributes the workload among the vector processing cores (VP).

The VPs, called V0 … VP15[2] in Figure 5, are the elements in charge of the actual processing of the data. Each VP is a multi-thread out-of-order vector processor with hardware architecture and instruction set that is specially optimized to operate on 3D vectors. Section 3 gives more detail regarding the Vector processor architecture.

The THEIA GPU topology follows a "CROW PRAM" model. PRAM stands for Parallel Random Access Machine, and is a common paradigm used to describe parallel machines [tbd]. CROW stands for Concurrent-Read Owned-Write, CROW PRAMs are described by [tbd] and offer series of advantages over other types of PRAM machines as analyzed by [tbd].
Following the CROW PRAM paradigm, some of the storage blocks from Figure 5 are read-only while other blocks are write-only. The OMEMs are write-only memories (from the VP's perspective) that are "owned" by each VP, this is, each VP can only access a single and unique OMEM block, and can only perform write operations to that OMEM block.
The TMEM, on the other hand, is a read-only block (from the VP's perspective). The TMEM can be concurrently accessed for reading operations by one or more VPs at any given point in time. Together the TMEM and the OMEM blocks allow the GPU to be modeled as a CROW PRAM machine.

The next sections will further describe the various functional blocks from Figure 5.

2.1. Control processor Overview

The main function of the Control Processor (CP) is to allow the user to programmatically control the resource allocation and the workload distribution of the GPU. In other words, instead of implementing complex dynamic hardware based scheduling algorithms, the CP allows for these algorithms to be implemented in software. This way the hardware complexity is reduced while the overall system becomes more flexible. This section will present an overview of the CP, to see a full description of the CP architecture and instruction set please see section 8.

[2] Figure 5presents a GPU configuration with 16 VPs but this number may vary depending on the specific GPU implementation.

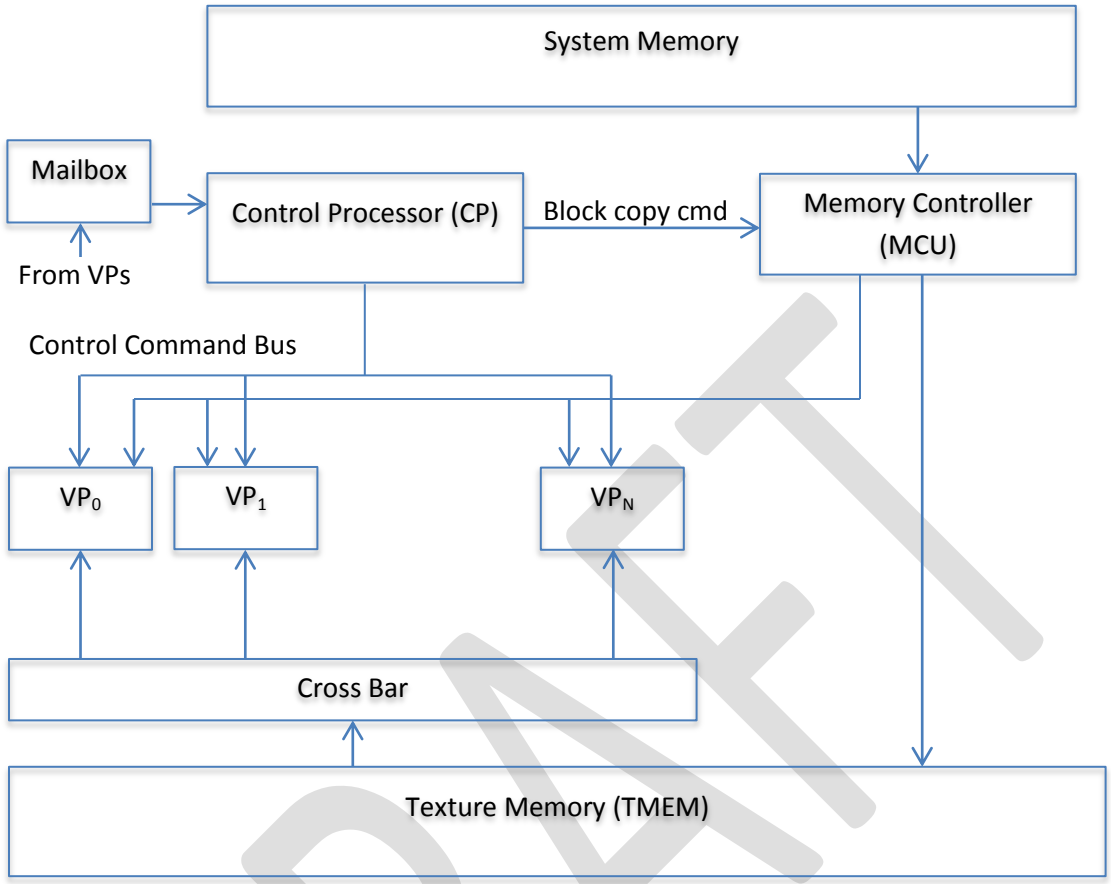

Figure 6 Control Processor within the system

The CP is a minimalistic processor. It is mainly in charge of controlling the dispatching of code, data and commands into the vector processors (VPs). There is a single control processor for the entire GPU and it is connected to the vector processors using a topology as the one depicted in Figure 6.

1.1.1. Data block copy operations

As depicted in Figure 6 the control processor (CP) interfaces with an internal memory controller (MCU).

The CP issues special instructions called “data block copy commands” to the MCU, telling the MCU to copy memory blocks from the main memory into the TMEM or into the VP’s internal memory locations. It is important to mention that the MCU can only copy data from the main memory and **not** into the main memory, in other words, the Main memory is read-only from the GPU perspective.

The format of the “data block copy commands” is illustrated in Figure 7

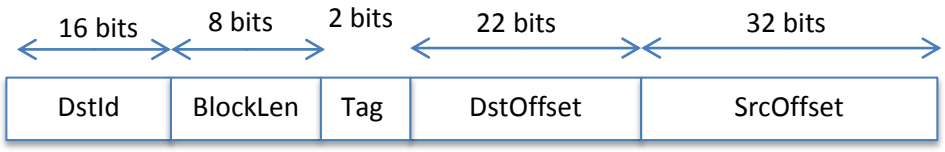

Figure 7 CP “data block copy command” format

The data block copy command is made of several fields as shown in the previous illustration. Table 3 describes the meaning of the various fields of the “data block copy command”.

Table 3 Data copy command fields

Field	Description
DstId	**0:** **Destination NULL**: No data blocks are copied and no copy commands are queued in the MCU. **1:** **Destination TMEM**: The data block is copied by the MCU from the main memory into the TMEM memory. **2 to N+2**: **Destination VP**: The data block is copied by the MCU from the main memory into the VP identified by the index *DstId-2*
BlockLen	How many blocks to copy from the Main memory into the destination resource identified by *DstId. Up to 1024 blocks can be copied.*
DstOffset	Offset where the MCU will copy the data at the destination resource identified by DstId. • When the target resource is the TMEM, the offset represents the linear address where the data will start to be copied. • When the target resource is one of the VPs, the offset is divided in address and tag: • DstOffset [20:0]: Linear address. • DstOff[22:21]: Destination Tag: Can have one of the following values: 10 -> Final destination is VP Instruction Address. 01 -> Final destination is VP Data Address .
SrcOffset	Offset into the main memory from where the MCU will start coping the data.

The data block copy commands are issued by the CP to the MCU in an asynchronous way. In other words, the CP issues a data block command and then the CP can continue with the control code execution even if the MCU has not yet finished copying the data blocks. If the CP issues another data copy command to the MCU while the previous copy command has not finished, then the copy command gets queued in the MCU. The MCU presents a signal with the number of pending copy operations to the CP. This signal is mapped into the CP internal register STATUS[MCU_OPERATION_PENDING][3]; it is the software responsibility to poll this register in order to know the number of pending memory copy operations in the MCU.

1.1.2. Control processor messages

The CP has the ability to send special messages called "control processor messages" to one or more VPs using the *Control Command Bus (CBC)* from Figure 6. The control processor messages have the following format:

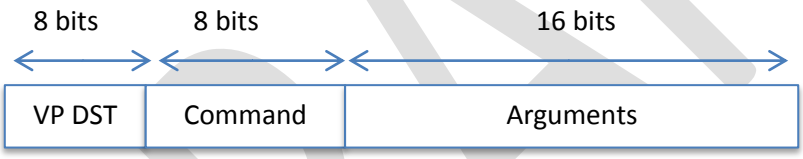

Figure 8 CP control processor message.

As depicted in Figure 8 the format of the "control processor messages" is very simple, it is made of a VP destination field, which specifies whether the CP message is targeted at a single VP or broadcasted to all the VPs, a command to specify the action that the VP has to perform and also an optional 32bit argument.

Table 4 Control processor messages

CP message field	Arguments
VPDST	The destination for the CP command. It has one of the following possible values: 0: NULL Message. The message has no target VPs. 1-127: Message is targeted to one of the possible VPs.[4] 128: Message is broadcasted to all the VPs.

[3] More information in section <TBD>

[4] The number of VPs depend on the version of the GPU implementation. Currently up to 16 VPs are supported.

Command	The command that the CP sends to one or more VPs to execute. It has one of the following possible values: 0: Start Execution 1: Stop Execution More to be defined
Argument	**Reserved for future use**

The most important use of the "control processor messages" is allowing the CP to start or to stop the VPs execution. This allows the user to program the CP in order to have full control of the resource and workload distribution.

1.1.3. Mail-boxing

Mail-boxing is a mechanism which allows passing messages between the VPs and the CP[5] during code execution. Each VP has a special 33 bit register called Mailbox. Each Mailbox has 32 bits of data and a 1 bit semaphore flag.

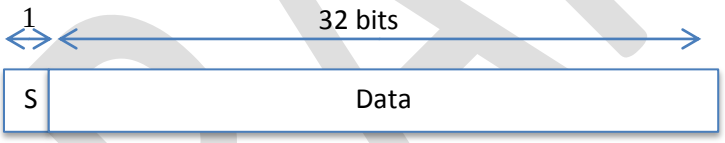

Figure 9 Mailboxing registers

The semaphore flag controls the write ownership of the mailbox register. If the semaphore bit is set then the CP has write ownership of the mailbox, otherwise the corresponding VP has write ownership of the mailbox. All mailboxes' semaphore bits are cleared after reset.

If the semaphore bit is set, then the CP gets notified of an incoming message delivered by the VP into the corresponding mailbox. The CP can now post a reply and then clear the semaphore flag to notify the VP that a reply has been delivered.

1.2. Vector processors (VP)

[5] In the current version of the RTL, the communication can only be initiated by the VP.

The VPs are a series of multithreaded out-of-order vector processors featuring fixed point arithmetic units and special purpose hardware to accelerate the most common 3D graphic operations. Each VP is divided into 5 main block called IO, EXE, MEM and CTRL. This is illustrated in Figure 10.

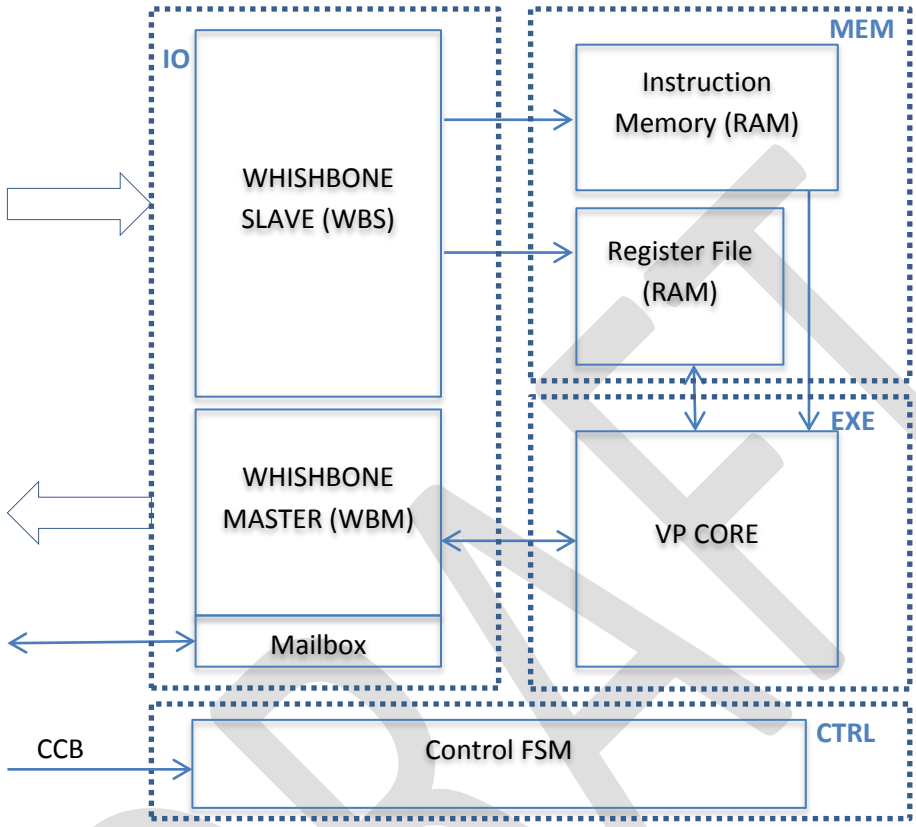

Figure 10 The main blocks of a CORE

The main building blocks shown in Figure 10 are further described later on this document.

The "VP CORE" block is where most of the complexity resides, thus section 3 of this document is dedicated exclusively to the VP CORE block.

2. Control FSM

This block has the main control FSM that is in charge of orchestrating the VP operation. Each VP has a single Control FSM. The Control FSM is mainly responsible of handling commands coming from the CP through the CCB (control command bus) and guarantying that the VP Core reacts accordingly.

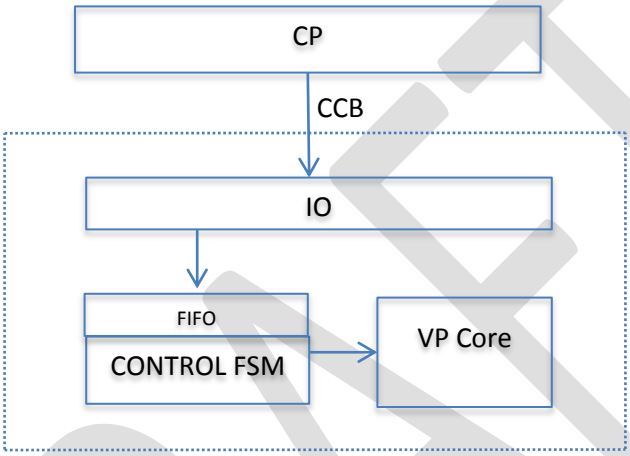

Figure 11 The CONTROL FSM, the CP and the VP Core

When the VP first comes out of reset no code in the VP gets executed by default. Instead of this, the VP first reaches a state called WAIT_FOR_CP where it will remain until one or more CP commands get queued into the CP command FIFO. Once a CP command becomes queued, the FSM will transition into a specific state which will take care of the CP request. If the CP command requires starting the main execution thread then the FSM transitions into the START_MAIN_THREAD state and then back into the WAIT_FOR_CP state. This means that the control FSM is not required to wait until the VP code is finished in order the handle more CP commands.

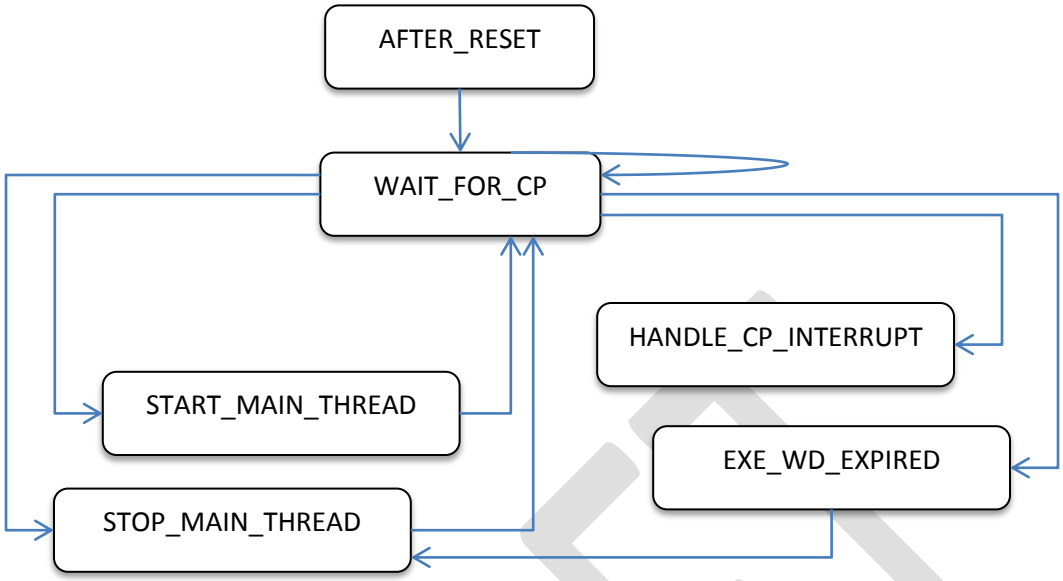

Figure 12 Control FSM

3. Vector Processor CORE (VP CORE)

The current section provides the architecture specification for the GPU's VPs. A detailed explanation of all the VP data structures is reviewed in the subsequent sections.

3.1. Introduction

Each THEIA VP combines the features of a vector processor and a multithreaded out-of-order machine. This means that the VP is capable of sustaining various levels of instruction level parallelism (ILP) and data level parallelism.

Instruction level parallelism is achieved by means of an in-order pipelined issue unit and several out-of-order execution units. The Tomasulo's algorithm [citation] is used to implement the out-of-order machine using the register renaming technique.

In order to cover for long stalls that the ILP from the out-of-execution can no longer prevent, a multithreading technique is used.

There are three main approaches to multithreading as mentioned by [1]: Fined grained multithreading, coarse-grained multithreading and simultaneous multithreading (SMT). The THEIA VP implements a simple version of SMT where up to 4 threads[6] can share the resources of a single VP unit.

As with most SMT implementations, all of the issues at a given point in time come from the same thread, but instructions from different threads can start executing on the same clock cycle (when dependencies are resolved at the reservation stations and so on). Since THEIA VP builds SMT on top on of an out-of-order machine, separate per-thread PC and renaming tables are maintained.

Data level parallelism is achieved by having 3 separate data lanes on each execution unit and also by means of vector processing techniques. Each THEIA VP has 256 x 32 bit registers which are divided logically across the 3 data lanes. These registers are implemented using a simple RAM memory structure divided into banks in order to provide the sufficient bandwidth for the vector operations. Each Thread is limited to access up to ¼ of the total register address[7] and there is no means of data sharing among the threads.

3.1.1. Single Thread execution example

This section will briefly describe the flow of the execution for a program running on a single thread. The next short code snippet will help clarify some of the concepts related to the VP execution and capabilities.[8] This code is written in a high level language called T-Language which is designed specifically to write code for the THEIA GPU. The language itself is described on separate document. Given that the T-Language closely resembles C/C++ it is assumed that an average reader can understand it.

[6] This number may vary depending on the release of the RTL

[7] When multithreading is disabled, the single thread has access to the entire register address space, this is in fact controlled by the software

[8] The code snippet is written in "T-Language". For more information on the "T-Language" please refer to the <TBD> documentation.

```
//Declare some variables. These variables will get stored in the internal VP register file
vector V1 = (1,2,3), V2=(4,5,6), V3=(7,8,9);
vector V4 = (10,11,12), V5=(13,14,15), V6=(16,17,18);
vector A[10], B[10], C[10];

//Some code to initialize the arrays goes in here

//Divide 2 variables. The division can take up to 32 clock cycles to complete
V1 = V2 / V3;

//Multiply two "arrays" in order to use the VP "vector" processing capabilities.
//Thanks to the out of order nature of the VP this multiplications can be executed in
//parallel with the previous division. Since each array has 10 element it will take the VP
//10 clock cycles to complete the multiplications (each multiplication takes 1 clock cycle)
C = B * A;

//Now do a subtraction. Once more because of the out of order nature of the VP,
//this will happen in parallel with the previous two operations. Also play around with
//the "destination channel selector" and "swizzling" features

V4.y = V5.yxz – V6.zyy;
```

Figure 13 A sample code (single thread)

The above code is meant to give the machine the opportunity to execute several instructions in parallel as we are about to see. The code starts by declaring several variables. Each variable is called a "vector"

and consist of a single word which is divided among a 32 bit “X block”, a 32 bit “Y block” and a 32 bit “Z block”. Consider the following statements from the above code:

vector V1 = (1,2,3), V2=(4,5,6), V3=(7,8,9);

This code declares 3 variables: V1, V2 and V3. It also specifies that those variables should be initialized to the constant values (1,2,3), (4,5,6) and (7,8,9).

Each VP operation executes simultaneously on the X, Y and Z blocks of the data, meaning that the VP has 3 data lanes. In order words, each VP has 3 adders, 3 dividers, 3 multipliers and so on. Consider the division operation from Figure 13, in a single clock cycle the 3 dividers will trigger in order to start calculating the division for the X, Y and Z blocks from V2 and V3.

Since the VP runs in an out of order fashion, it doesn’t need to wait until the division is complete in order to issue the next instruction. The next instruction is a multiplication; it multiplies two vector “arrays” together. A vector “array” is an array consisting of two or more vector elements (each element is a vector with an X block, a Y block and a Z block as before).

Each element from a *vector array* is internally allocated in consecutive memory positions so that the VP can perform a type of “data pipeline” using “convoys” of data[9]. This type of data level parallelism is typical in vector processor architectures.

Figure 5 shows how the multiplications are executed in parallel with the division. Each clock cycle the VP’s multipliers serially calculate the result of each vector array element in the data convoy. Next, the code specifies to execute a subtraction, this happens in parallel with the previous operations. Since the Additions/Subtractions take 1 clock cycle, then the subtraction is going to end before the array multiplication and the divisions are done.

[9] See Hennesy and Paterson…

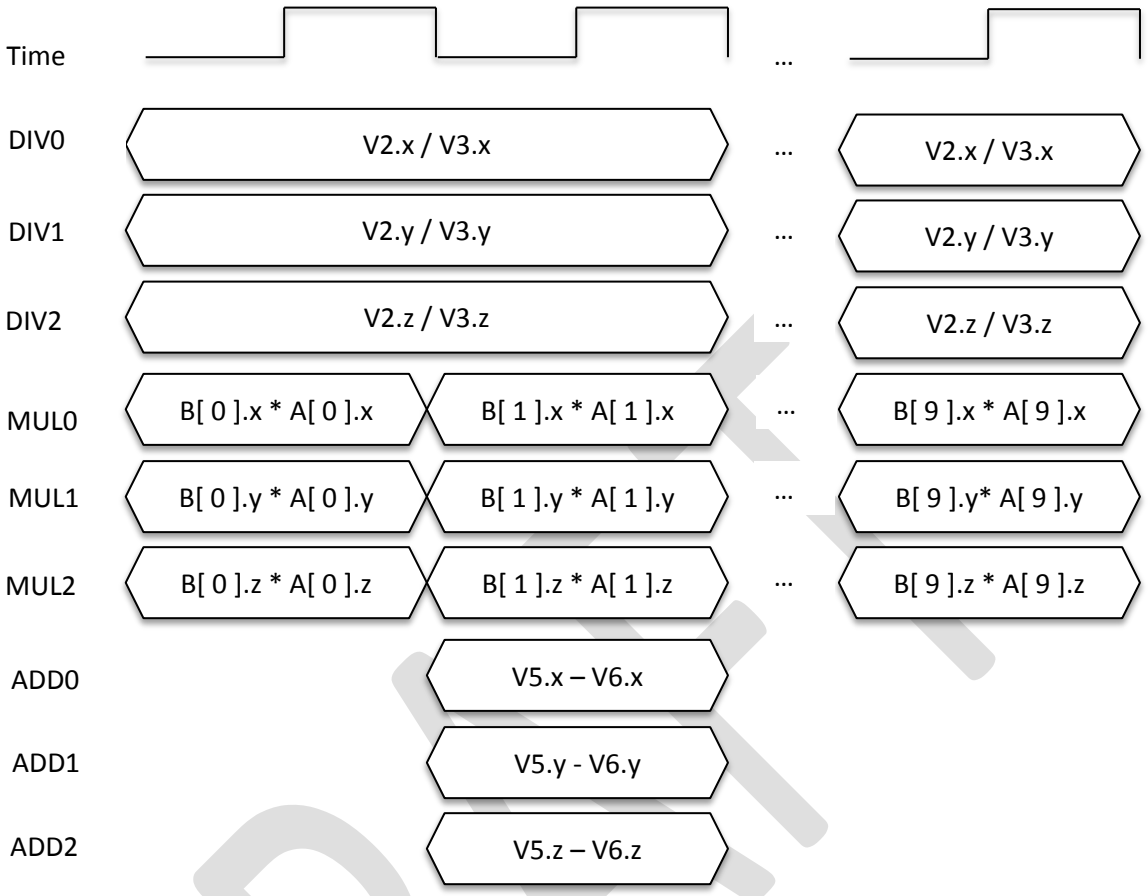

Figure 14 Behaviour of the execution units over time for the example from Figure 13

Figure 14 shows the overlapping execution of the various execution units involved in the code from Figure 13. As it was mentioned before, the Division algorithm takes 32 clock cycles to complete (worst case). It is also interesting to observe how the multiplying units are kept busy working on the array elements with no need of a new instruction being issued. Also, since the addition units are free, those can handle additional operation over time as shown in Figure 14. The Architectural features allowing this kind of parallelism are described in the next sections.

3.2. VP Architecture

The VP is the logic block responsible to perform the Arithmetic and Logic operations within each CORE. The VP can operate on vectors of data, each vector consisting of 3 32-bits words as explained in section 3.3. The VP features an in-order fetch and out-of-order execution following the classic Tomasulo's

[10]algorithm for register renaming.

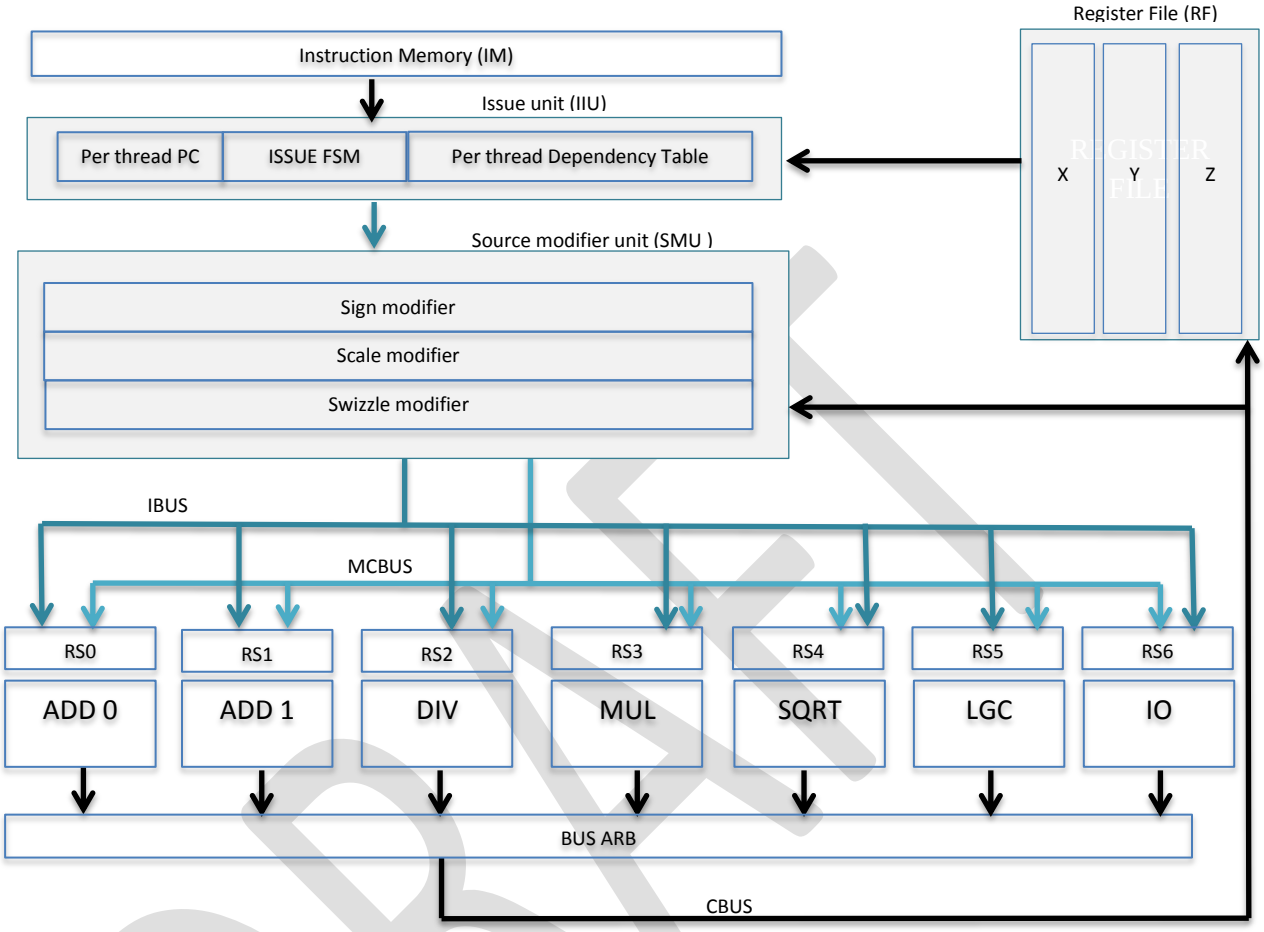

Figure 15 VP architecture

Figure 15 shows the main building blocks of the VP. The instructions are initially fetched from the instruction memory (IM) by the IIU. Each instruction is a 64 bit word and the layout of the instruction is discussed in section 3.5.

Once the instruction is fetched, the IIU chooses a free reservation station (RS1 to RS6) to issue the instruction, according to the instruction OPCODE. If there are no reservation stations available to execute the instruction, then there is a structural hazard condition and the IIU stalls until an appropriate reservation station becomes available.

[10] Note: with some modifications as that are specified in s subsequent sections.

If there are reservation stations available to execute the instruction, then the IIU determines the reservation station number and data dependencies that will be added into the *Issue packet* using the dependency table from Figure 1. The *issue packet* is a special data structure that is broadcasted to all the reservation stations connected to the Issue Bus. The format of the issue packet is discussed in section 4.1.2.1.

While the dependencies are established, the IIU also reads the instruction operand values from the register file RF. Each instruction operand value is a 96 bit word, and the layout of these words is discussed in section 3.3.

Once the operand values are retrieved from the RF, the operand manipulators (section 3.5.3) are applied in the following sequence:

First the sign control does a 2 complement on the individual X, Y and Z components of each operand source, as described in section 3.14.

Next the source operands are scaled according to the rules in section 3.13 and swizzled according to the rules in section 3.15.

Finally the source operands are presented to the issue bus, together with reservation station number and the dependency fields. The instruction is finally issued to the reservation stations, the dependency table gets updated with the current instruction and the Issue packet is broadcasted to the Reservation stations.

3.3. Word size and Endianness

THEIA words are little endian, meaning that the least significant bit goes into the lowest address. Each word is 96 bits long and usually represents a 3D vector[11]; thus it is divided among three 32 bits value slots called X, Y and Z as depicted in Figure 16. Depending on the VP operation, the X, Y and Z components of the word can be individually accessed or the entire 96 bits can be accessed simultaneously.

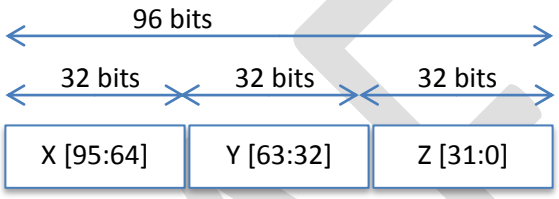

Figure 16 Vector word layout

3.4. Fixed point arithmetic.

The VP has the ability to work with integer arithmetic or with fixed point arithmetic.
When working with integer arithmetic, the entire 32 bits from the X, Y or Z blocks of a word are used to store the integer value.
When working with fixed point arithmetic (Qm.n), the 'n' least significant bits of the X, Y or Z blocks are used to store the decimal part of the number , while the 'm' more significant bits of the X, Y or Z blocks are used to store the integer part of the number.

[11] 4D/Homogenous-coordinates are not natively supported at the hardware level.

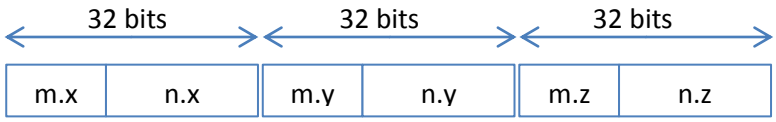

Figure 17 Storing Fixed point numbers in a 96 bit word

The length of the 'n' bits of a Qm.n number is called the fixed point SCALE. The SCALE is used to transform numbers from fixed point to integer and vice versa and is also used as part of the fixed point multiplication and division operations[12]. Table 5 shows the way in which the arithmetic operations are performed when using integer arithmetic and when using fixed point arithmetic.

Table 5 Scaling arithmetic operation for fixed point

Operation	Integer	Fixed Point
Addition	R = A + B	R = A + B
Subtraction	R = A – B	R = A – B
Multiplication	R = A * B	R = (A * B) >> SCALE
Division	R = A / B	R = (A << SCALE) / B
Logic operation	See section <>	n/a

It is important to mention that it is the compiler's responsibility to appropriately manage the SCALE in the operations to either use fixed point or integer arithmetic. In other words, the VP has no knowledge if a particular instruction should use fixed point or integer arithmetic; the VP only executes the operation after applying the SCALE to the input arguments according to Table 22. The logic operations are not affected by the SCALE.

3.5. Instruction overview

THEIA instructions are 64 bits wide. Each instruction is divided into various *"sections"* as depicted in Figure 18: operation section, destination section, source 1 and source 0 sections or immediate value section. The source 0 and source 1 sections are mutually exclusive with the immediate value section.

[12] The square root operation is a special case which always assumes fixed point input arguments. See section 3.9 for details.

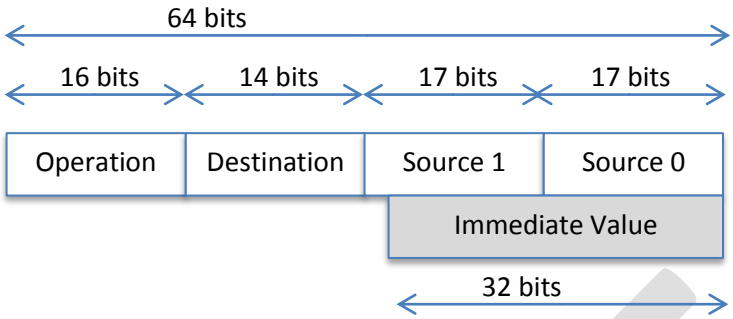

Figure 18 Instruction Layout

Each instruction section has special *fields* that modify the VP behavior in various ways. A very important field is the IMM field. The IMM field tells the VP whether it has to interpret the lowest 32 bits of the instruction as an immediate (literal) value, called IMMV, or as part of the register source sections. The IMM field also takes part in the addressing mode as discussed in section 3.6. Figure 19 illustrates how the VP interprets the instruction depending on the IMM bit.

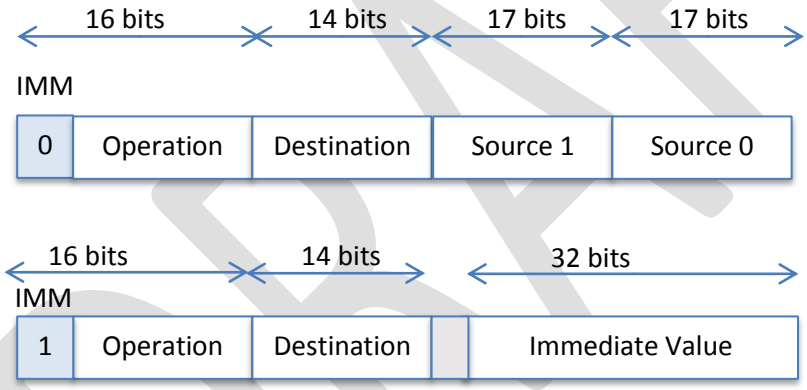

Figure 19 Immediate bit and the way the instruction is interpreted by the IIU

Other instruction fields specify different behaviors such as which blocks of the resulting word to write back into the RF, how to handle the sign of the input operands, how to handle branches, etc. Table 13, Table 14, Table 18 and Table 19 show the various Instruction section fields and their meaning.

3.5.1. Instruction operation codes

The instruction operation codes or OPCODEs are the set of all the possible arithmetic or logic operations that the VP is capable of doing. The VP is actually able to do a small number of different OPCODES:

addition, multiplication, division, square root, IO and logic operations. This may seem as a small set of possible operations at first, but when combined with the instruction source modifiers from section 3.5.3, it becomes capable of doing a wide variety of operations.

Also, each of the possible OPCODES is executed simultaneously on the x, y and z blocks of the instruction sources. In other words, the VP has 3 adders, 3 multipliers, 3 dividers and so on[13]. Table 6 lists the possible arithmetic operations the VP can do.

Table 6 VP operations

Operation	Description
Addition	$\begin{pmatrix} Rx \\ Ry \\ Rz \end{pmatrix} = \begin{pmatrix} Ax + Bx \\ Ay + By \\ Az + Bz \end{pmatrix}$
Multiplication	$\begin{pmatrix} Rx \\ Ry \\ Rz \end{pmatrix} = \begin{pmatrix} Ax * Bx \\ Ay * By \\ Az * Bz \end{pmatrix}$
Division	$\begin{pmatrix} Rx \\ Ry \\ Rz \end{pmatrix} = \begin{pmatrix} Ax/Bx \\ Ay/By \\ Az/Bz \end{pmatrix}$
Square root	$\begin{pmatrix} Rx \\ Ry \\ Rz \end{pmatrix} = \begin{pmatrix} \sqrt{Ax} \\ \sqrt{Ax} \\ \sqrt{Ax} \end{pmatrix}$
Bitwise AND	$\begin{pmatrix} Rx \\ Ry \\ Rz \end{pmatrix} = \begin{pmatrix} Ax\ AND\ Bx \\ Ay\ AND\ By \\ Az\ AND\ Bz \end{pmatrix}$
Bitwise OR	$\begin{pmatrix} Rx \\ Ry \\ Rz \end{pmatrix} = \begin{pmatrix} Ax\ OR\ Bx \\ Ay\ OR\ By \\ Az\ OR\ Bz \end{pmatrix}$
Bitwise NOT	$\begin{pmatrix} Rx \\ Ry \\ Rz \end{pmatrix} = \begin{pmatrix} NOT\ Ax \\ NOT\ Ay \\ NOT\ Az \end{pmatrix}$
SHIFT LEFT	

[13] The exception to this is the SQRT which has a s ingle execution unit

	$\begin{pmatrix} Rx \\ Ry \\ Rz \end{pmatrix} = \begin{pmatrix} Ax \ll Bx \\ Ay \ll By \\ Az \ll Bz \end{pmatrix}$
SHIFT RIGHT	$\begin{pmatrix} Rx \\ Ry \\ Rz \end{pmatrix} = \begin{pmatrix} Ax \gg Bx \\ Ay \gg By \\ Az \gg Bz \end{pmatrix}$

Note from Table 6 how each operation is simultaneously executed on the x, y and z blocks of the source data. Again, it is important to mention that the VP makes no distinction between fixed point numbers and integer numbers; it is the compiler that needs to apply the corresponding SCALE using the techniques from the next sections in order to obtain the proper result for integer numbers or fixed point numbers.

3.5.2. Instruction destination block selector

As mentioned in section 3.3, each THEIA word has 3 32 bit blocks called x, y and z. Each instruction has the ability to write the results simultaneously into the three destination blocks, or it can also choose to store the results into only some of the x, y and z blocks leaving the other blocks un-changed.

Table 7 Example of destination selection

Operation	Description
R = A + B	$\begin{pmatrix} Rx \\ Ry \\ Rz \end{pmatrix} = \begin{pmatrix} Ax + Bx \\ Ay + By \\ Az + Bz \end{pmatrix}$
R.y = A + B	$\begin{pmatrix} Ry \end{pmatrix} = \begin{pmatrix} Ay + By \end{pmatrix}$
R.xnz = A + B	$\begin{pmatrix} Rx \\ \\ Rz \end{pmatrix} = \begin{pmatrix} Ax + Bx \\ \\ Az + Bz \end{pmatrix}$

Note from Table 7 that the ‘n’ symbol stands for no-write, so for example R1.xnz means to write the results into the x and z blocks but not into the ‘y’ block. Section 3.13 will list all of the possible combinations of destination blocks; from Table 7 it becomes clear that destination can be all of the

blocks, one single block or any two blocks.

3.5.3. Instruction source modifiers

Instruction source modifiers are special ways in which the VP can modify the input source data (Source 0 (SRC0) and Source 1 (SRC1)) before they reach the execution units (EU).

There are 3 ways in which the SRC0 and SRC1 data can be modified before they reach the EU: modifying the signs, modifying the scale or "*swizzling*" the data blocks. Each of these three modifications can be individually applied into the x, y or z blocks of SRC1 or SRC0. The next series of figures represent examples of possible source modifications.

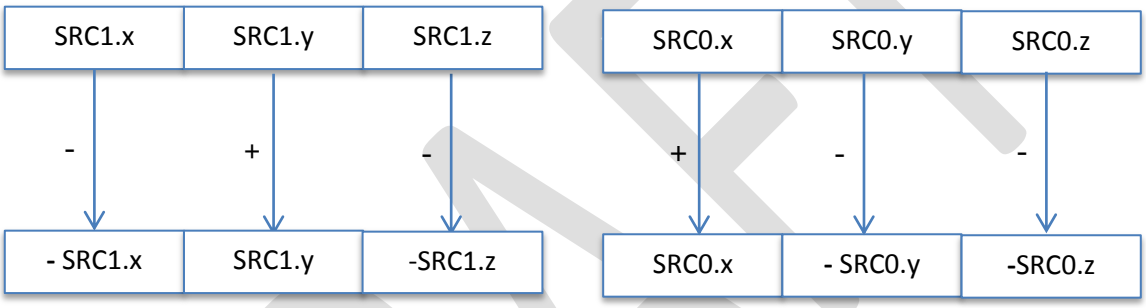

Figure 20 Modifying the individual signs of the instruction sources

Figure 20 shows an example of how the signs of the individual x, y and z blocks of the data sources can be modified. The sign modification can be used for vector operations such as cross products. Furthermore, the VP does not have a subtraction operation *per se*, but instead the compiler is required to negate the SRC0 from an addition in order to perform a subtraction.

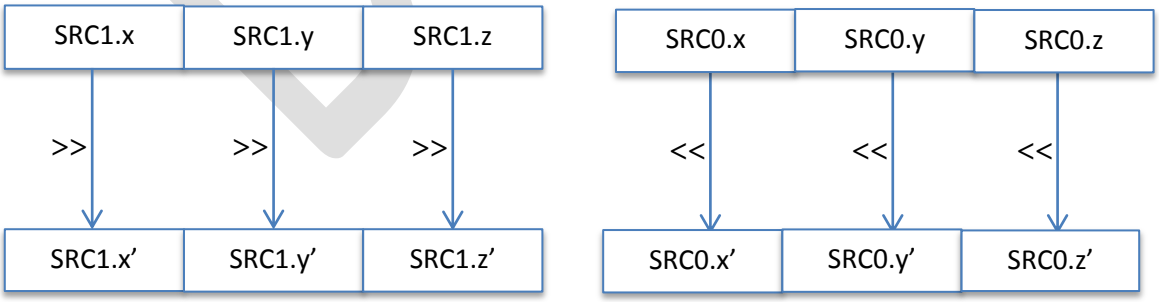

Figure 21 Modifying the scale of the instruction sources

Figure 21 illustrates a source scaling. Each x, y and z block can be shifted left or shifted right SCALE number of bits. The value of SCALE can be controlled by the setting the appropriate value in the configuration registers as will be detailed later on this document. The scale operations are used to transform numbers between the integer and fixed point numerical representations, and are also used to perform fixed point multiplications and divisions as it is specified on Table 5.

The last of the three possible source modifications is what is called “swizzling”. Swizzling allows replacing the x, y or z blocks of a source register by any other x, y or z block from that same source register. This concept is illustrated in the next figure.

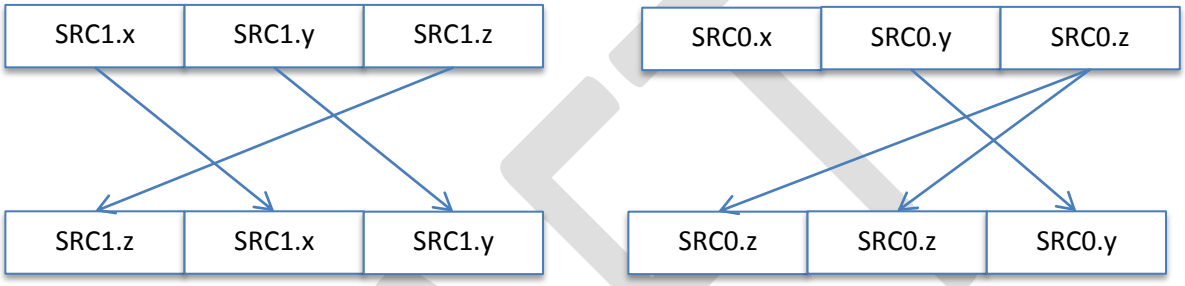

Figure 22 Swizzling instruction sources

Register swizzle is a very powerful technique which allows the VP to perform a variety of operations. An example of the usefulness of operand swizzling is matrix multiplication. Let’s take for instance the following 3x3 matrix multiplication:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} * \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} 1a + 2b + 3c \\ 4a + 5b + 6c \\ 7a + 8b + 9c \end{pmatrix} \quad (1)$$

Let’s assume that R1 has been loaded with the value (1,4,7), that R2 has been loaded with the value (2,5,8), that R3 has been loaded with the value (3,6,9), and that R4 has been loaded with the value (a,b,c). Equation (1) can be represented as series of swizzled operations in *T-language* like this:

```
R7 = R1 * R4.xxx;

R8 = R2 * R4.yyy;

R9 = R3 * R4.zzz;

R1 = R7 + R8;

R1 = R1 + R9;
```

The previous code shows that it would take the VP 5 operations to complete the 3x3 matrix

multiplication.

Finally it also possible to simultaneously combine the 3 types of source modifiers in a single instruction as illustrated in the next figure.

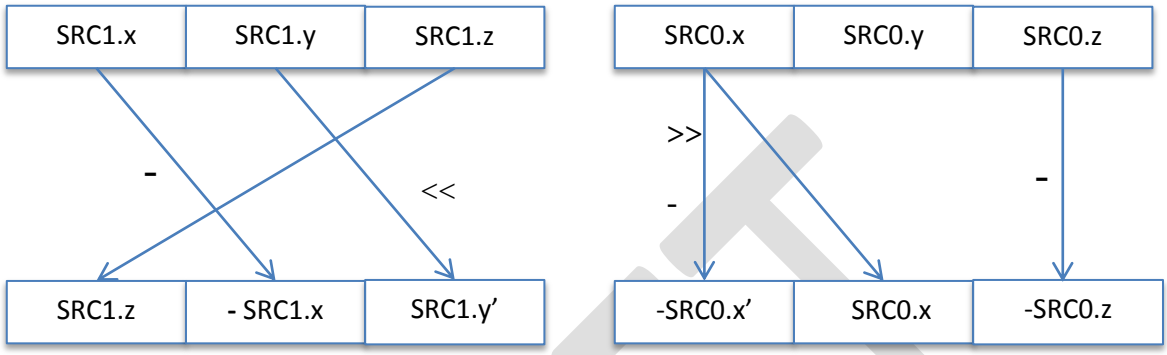

Figure 23 Combining several source modifiers in a single instruction

Figure 23 shows how it is possible to combine sign modifications, scaling and swizzling in the same instruction. To illustrate this, let's take for example a cross product vector operation. The cross product can be written as the following column of vectors:

$$\begin{pmatrix} Ax \\ Ay \\ Az \end{pmatrix} * \begin{pmatrix} Bx \\ By \\ Bz \end{pmatrix} = \begin{pmatrix} AyBz - AzBy \\ AzBx - AxBz \\ AxBy - AyBx \end{pmatrix} \quad (2)$$

You can see from (2) that the cross product needs to perform a series of subtractions (the VP uses sign control to negate the second argument for subtractions) and also needs to organize the sources in a special way in order to obtain the desired result. Let's assume that R1 has been loaded with the value (Ax, Ay, Az) and R2 has been loaded with the value (Bx, By, Bz). Equation (2) can be represented as series of swizzled operations in T-language[14] like this:

```
R3 = R1.yzx * R2.zxy;

R4 = R1.zxy * R2.yzx;

R1 = R3 – R4;
```

So the previous code shows that the VP can perform a cross product using only three instructions.

It is important to mention that the source modifiers are implemented as pure combinatorial blocks, this means that they add no extract latency to the operations.

[14] This a special 'middle level' language specially designed for the THEIA GPU, more details on section <TBD>

Table 8 summarizes the three possible instruction source modifiers and the document section where more information can be found.

Table 8 instruction operand manipulators

Instruction source modifier	Document section	Description
Swizzle control	3.15	Input operand Swizzle control logic.
Sign control	3.14	Input operand Sign control logic.
Scale control	3.13	Input operand Scale control logic.

3.5.4. Data dependencies and source modifiers

Looking back at Figure 15, it must be noted how the Source modifier unit (SMU) is connected to the issue unit (IIU) and is also connected to the commit bus (CBUS). This is because the SMU needs to apply the source modifiers to the data sources coming from the issue stage and it may also potentially need to apply source modifiers to the results from the execution units (EU) when the data dependencies get resolved. Let's illustrate this concept with an example.

```
R2    = (10,20,30);

R3      = (2,0,0);


R1 = R2 / R3.xxx;

R2.y = R1.zzz + R1;
```

Figure 24 Example of data dependencies when using source modifiers

In the example from Figure 24, the addition operation depends upon the result from the division operation. Because of the out-of-order execution, the addition instruction will be issued into the reservation stations regardless of the division result not being yet committed into the RF. Once the divider EU finishes calculating the result of the division, this result is written back into the RF and also forwarded into the SMU. The SMU needs to apply to proper modifiers to the result from the division EU and then present this modified result into the reservation stations so that the data dependencies can be properly resolved. This concept is illustrated in the next series of figures.

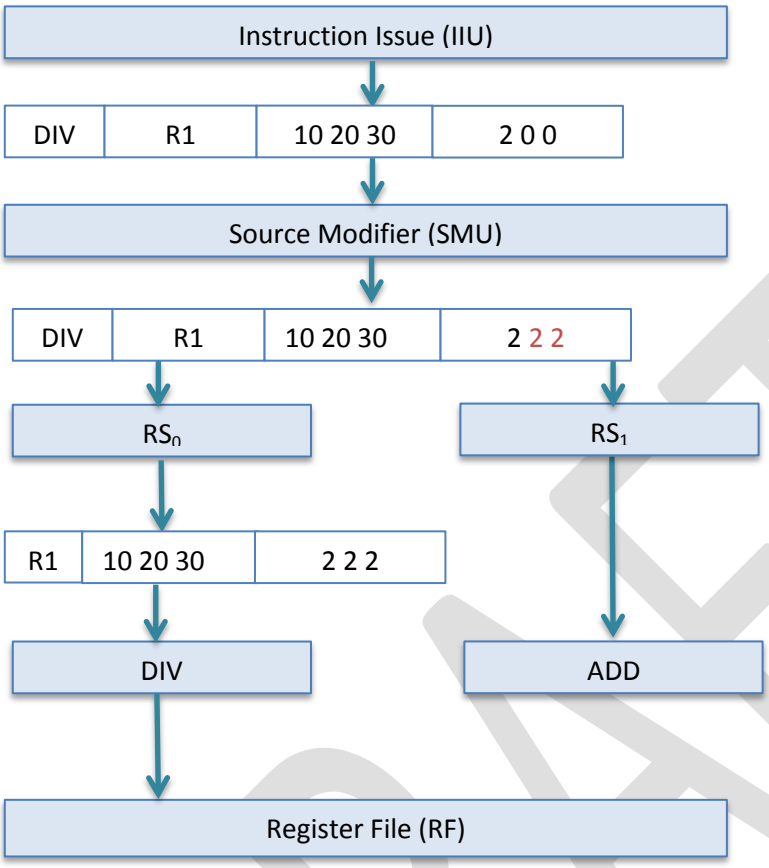

Figure 25 IIU issues a division

Figure 25 depicts the IIU is issuing the division instruction from Figure 24. The IIU retrieves the value of R2 (10, 20, 30) and the value of R3 (2, 0, 0) from the RF and then sends these vectors to the SMU. The SMU swizzles the values of R2 so that it becomes R2.xxx (2, 2, 2) and then broadcasts these values to the reservation stations along with the reservation station index (RSID)[15]. The reservation station whose index matches with the RSID broadcasted by the SMU latches the value, since no data dependencies where found, the RS triggers the corresponding EU (the divider for this example) using these input values from the SMU.

[15] Not shown in the picture.

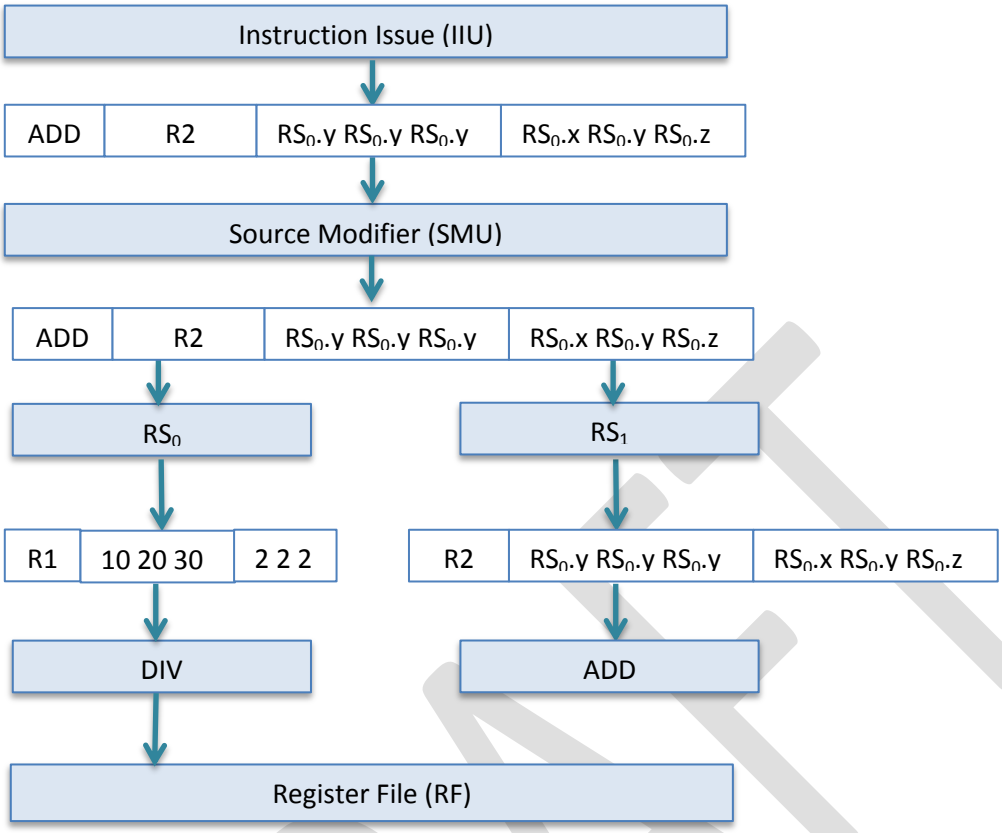

Figure 26 The IIU issues an addition operation

While the Divider EU is busy calculating the result of DIV operation, the IIU issues the next instruction which is an addition. Since the addition source operands depends on the result from the division instruction, the IIU uses the register renaming technique to specify the reservation station that will resolve the data dependencies, this is illustrated in Figure 26. The RS_1 will not start the addition EU until it receives the result from RS_0.

A number of clock cycles after issuing the DIV instruction, the divider EU is finally done calculating the result; this is shown in Figure 27. Figure 27 depicts the DIV EU committing the division results into the shared commit bus (CBUS). These results are also forwarded into the SMU, the SMU has a series of internal registers that store the various data dependencies that need to be scaled[16], signed changed or swizzled. In this example, the SMU knows that it needs to propagate two result values from RS_0 back to the reservation stations. One value would be the division result using the "yyy" swizzle combination and the other value would be the division result with no modifiers. Figure 27 shows when the SMU issues the "yyy" swizzled value (15, 15, 15) back into the reservation stations[17].

[16] More detail on this on section 4.1.2

[17] The order in which the dependencies values are presented by the SMU into the RSs is not deterministic. See section xxx for details.

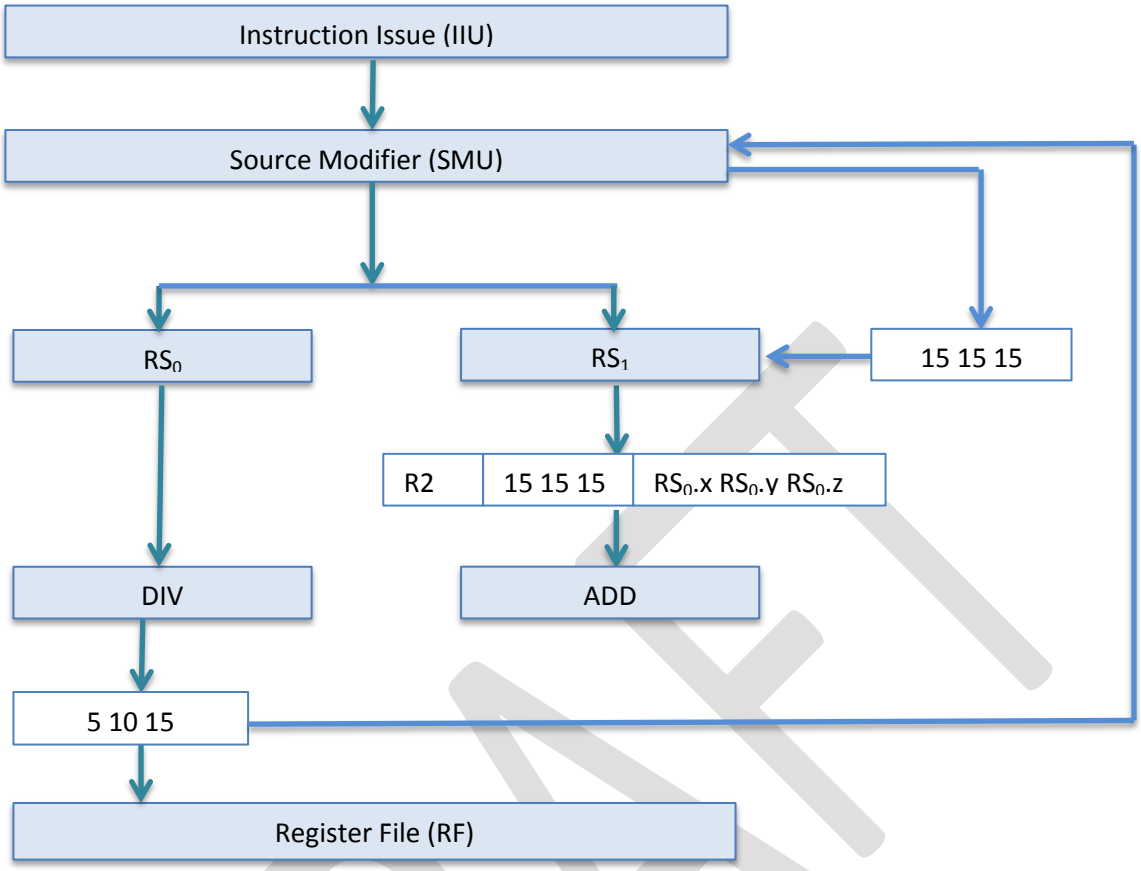

Figure 27 The DIV UE commits the results to the CBUS and the SMU. The SMU presents the first data dependency to the reservation stations.

Once the RS_1 gets the value (15, 15, 15) from the SMU it stores this value inside a set of internal registers. This resolves the first dependency, but the second dependency (RS_0.x RS_0.y RS_0.z) is still pending. One extra clock cycle needs to pass before RS_1 gets the second dependency from SMU; this is shown in Figure 28.

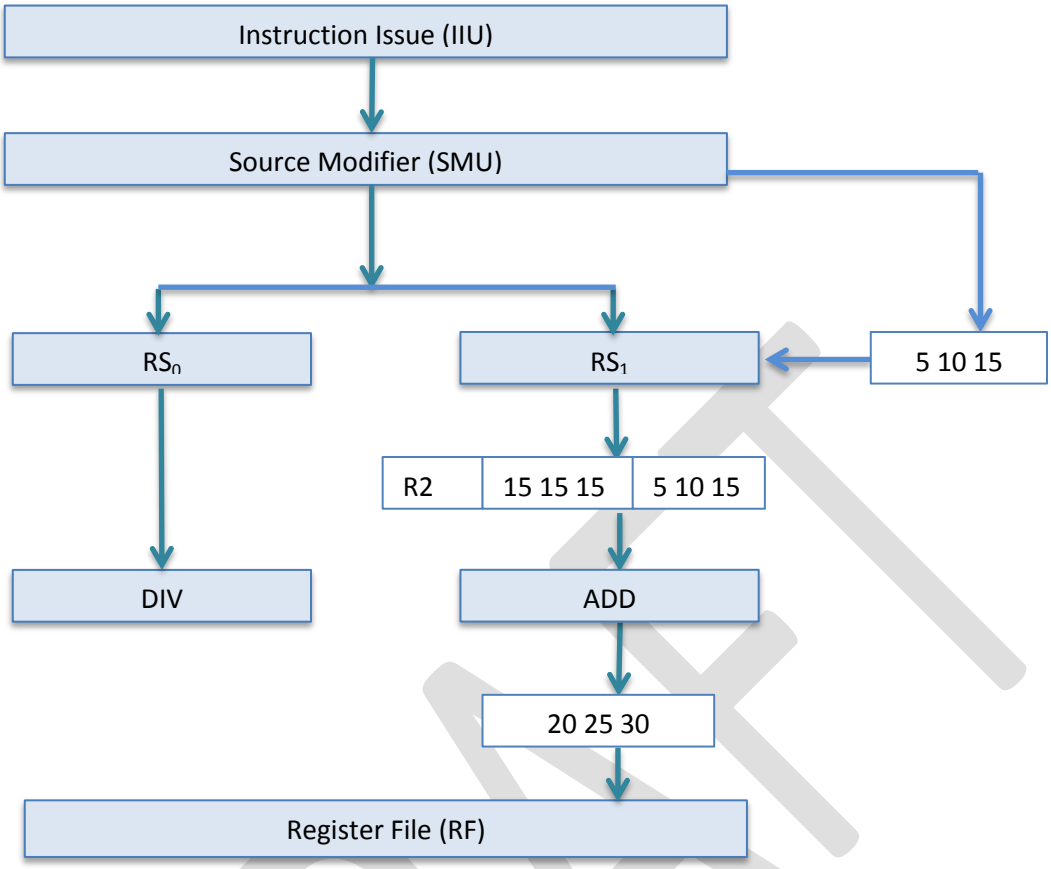

Figure 28 The SMU presents the second data dependency to the reservation stations. The ADD EU commits the result to the RF.

Figure 28 shows the last step needed to execute the code from Figure 24. In this illustration, the RS_1 is presented with the second data dependency (5 10 15) coming from the SMU. Now that RS_1 has the two necessary data dependencies it can finally send the operands to the ADD EU so that 1 clock cycle later, the result is calculated and presented to the CBUS so that it can finally be written back into the register file.

3.5.4.1. VP Flags

The IIU receives two input flags from the EUs. These two flags are called the ZFLAG and the SFLAG. The ZFLAG is a 3 bit wide signal that indicates that the current result in the CBUS is a zero. The SFLAG is a 3 bit signal that indicates that the current result in the CBUS is a negative number[18].

[18] Using 2's complement.

Table 9 Execution SFLAG values

SFLAG.x SFLAG.y SFLAG.z	Description
000	Result X block, Y block and Z block in the CBUS are non-negative.
001	Result X block in the CBUS is negative.
010	Result Y block in the CBUS is negative.
011	Result Y block and Z block in the CBUS is negative.
100	Result Z block in the CBUS is negative.
101	Result X block and Z block in the CBUS are negative.
110	Result X block and Y block in the CBUS are negative.
111	Result X block, Y block and Z block in the CBUS are negative.

Table 10 Execution ZFLAG values

ZFLAG.x ZFLAG.y ZFLAG.z	Description
000	Result X block, Y block and Z block in the CBUS are non-zero.
001	Result X block in the CBUS is zero.
010	Result Y block in the CBUS is zero.
011	Result Y block and Z block in the CBUS is zero.
100	Result Z block in the CBUS is zero.
101	Result X block and Z block in the CBUS are zero.
110	Result X block and Y block in the CBUS are zero.
111	Result X block, Y block and Z block in the CBUS are zero.

The ZFLAG and the SFLAG are mainly used in the branch decision logic as explained in section 3.16.

3.5.5. Execution units and reservation stations

The VP has 6 reservation stations (RS). Each reservation station controls a single execution unit (EU). The reservation stations are responsible of triggering the execution units when the operands are ready or stalling the execution units while waiting for the data dependencies to arrive in the commit bus. Table 11 lists the available reservation stations.

Table 11 VP Reservation Stations

Reservation station	Latency (clock cycles)	Description

RS_ADD0	1	Integer unsigned addition/subtraction
RS_ADD1	1	Integer unsigned addition/subtraction
RS_DIV	Variable	Integer signed division
RS_MUL	1	Integer signed multiplication
RS_SQRT	1	Integer square root.[19]
RS_LOGIC	1	Bitwise logic operations. See section <> for more details.
RS_IO	Variable	Input/Output operations

It is important to note from Table 11 that there are 2 reservation stations dedicated to do additions, RS_ADD0 and RS_ADD1. The reason to have 2 separate reservation stations dedicated to add is that the addition is the most issued instruction[20]. If a reservation station becomes busy waiting for a data dependency, it is most likely that this RS was one of the adders, and it is also likely the next instruction that will get fetched from IM is another addition.

The additions, subtractions, branches, register to register assignments, constant to register assignments, etc. all these constructs can be achieved using simple additions. In order to illustrate this concept, consider the snippet of code written in T-language[21] presented in Figure 29.

The code from Figure 29 is basically assigning constant values to variables (variables are always stored in registers), then it enters a function (called main), evaluates an "if" statement and calls a second function from within the first function.

```
...


CameraPosition.x        = 0;

CameraPosition.y        = 0x00040000;

CameraPosition.z        = 0x00020000;

//-------------------------------------------------------------

function main()

 {
```

[19] Note: This is **not** generic square root algorithm; it approximates the square root integer number within a range of 0 and 512. See section 3.10 for details.
[20] See appendix TBD for a quantitative proof.
[21] This a special 'middle level' language specially designed for the THEIA GPU, see "T-Languague Document specification" for more details.

```
        if ( PrimitiveCount != MaxPrimitives )
        {
              GenerateRay();
              Hit = 0;
              PrimitiveCount = 0;
        }
        CalculateBaricentricIntersection();
        exit;
}

//------------------------------------------------------------

...
```

Figure 29 An example code written in T-Language.

The code from Figure 29 then is compiled into a series of ADD operations as it is shown in the next figure.

```
...

8:      c001 200b 0 0                                   //ADD R11.x__ 0
9:      a001 200b 2 0                                   //ADD R11._y_ 40000
10:     9001 200b 1 0                                   //ADD R11.__z 20000
//__main
11:     241 10 f 1c010                                  //ADD <BRANCH.ZERO> @16.___ R15.xyz R16.-x-y-z
12:     f001 201f 0 e                                   //ADD R31.xyz e
13:     201 @__GenerateRay 0 0                          //ADD <BRANCH.ALWAYS> @__GenerateRay.___ R0.xyz R0.xyz
14:     f001 201a 0 0                                   //ADD R26.xyz 0
15:     f001 200f 0 0                                   //ADD R15.xyz 0
16:     f001 201f 0 12                                  //ADD R31.xyz 12
17:     201 @__CalculateBaricentricIntersection 0 0     //ADD <BRANCH.ALWAYS> @__CalculateBaricentricIntersection.___ R0.xyz R0.xyz
18:     401 0 0 0                                       //ADD R0.___ R0.xyz R0.xyz

...
```

Figure 30 The code from Figure 29 translated into assembly language

Although the specific syntax of the assembly language from Figure 30 will not be covered in this document, it becomes clear from this figure that the generated code is simply a series of ADD operations.

Section 3.5 will provide more information regarding how the majority of the operations are really just additions combined with some other fields from the instruction in Table 13.

3.5.6. VP Stall conditions

The VP can get into a stall state under certain scenarios; these scenarios are specified in Table 12 . When the VP reaches a stall condition, the IIU stops fetching instructions from the IM and stops issuing instructions to the reservation stations.

Table 12 IIU Stall conditions

Stall condition	Description	Un-stall condition

Structural hazard detected	The IIU detected that there are no reservation stations available to execute the current instruction.	Once an appropriate RS becomes available to execute the current instruction
Data dependency and special operand modifiers.	The current instruction has a data dependency on one of the operands and the SMU has no free slots to handle the dependency.[22]	The data dependencies are resolved and corresponding result vector are written back into the RF.

3.6. Instruction addressing modes

The VP has four addressing modes: direct, direct with displacement, indirect and indirect with displacement. The addressing modes depend on the IMM bit and the MODE field as described in section 3.7.

In direct addressing mode the instruction destination is simply the index of the general purpose register specified by the **literal** DSTINDEX field from Table 14. This mode does not depend on the IMM instruction bit.

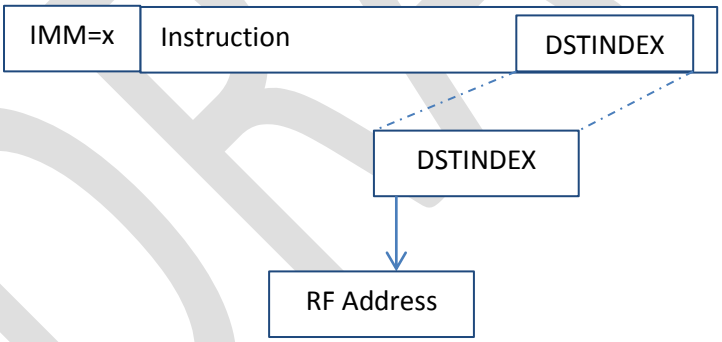

Figure 31 Direct addressing mode

In direct addressing with displacement the instruction destination is the index of the general purpose register specified by the **literal** DSTINDEX field from Table 14, plus the SPR field OFFSET[23].

Figure 32 depicts the logic that is used to calculate the RF address when using direct addressing with displacement. It is important to note that the direct addressing mode can only be used to address

[22] For more information regarding the SMU dependency slot mechanism see section TBD.
[23] The value of OFFSET is zero by default, but this needs to be set by the software.

memory locations in the internal GPU register file (RF). The direct addressing mode with displacement does not depend on the IMM bit. Figure xxx shows an example of direct addressing.

```
ADD R1 R2 R3          //R1 = R2 + R3
```

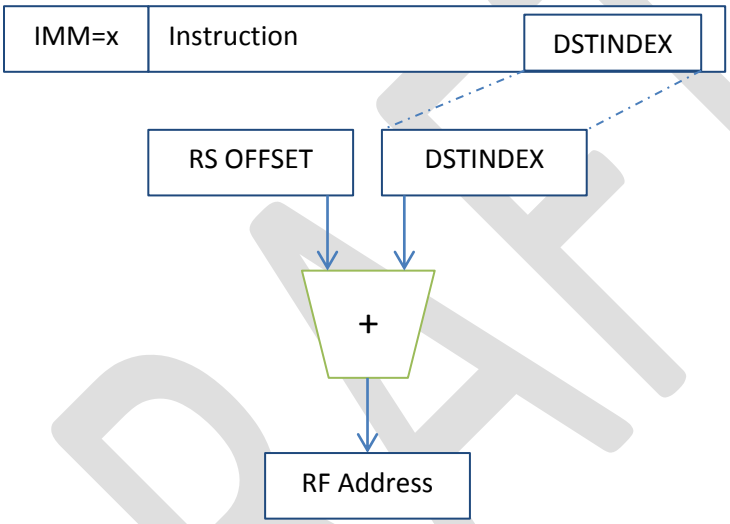

Figure 32 Direct Addressing with displacement

Figure 33 shows an example of an instruction using direct addressing. The register in red (R1) is used as the index into the RF, in other words the index is simply "1". This index is added the value of OFFSET. Since the OFFSET field is part of a special purpose register (SPR), it is not explicitly used in the instruction. The only way to change the value of the OFFSET SPR is by writing directly into this special purpose register[24].

```
//Assume that the OFFSET register has been set to a value

ADD R[1 + offset] R2 R3          //R[1+offset] = R2 + R3
```

Figure 33 direct addressing with displacement

Additionally, the direct mode with displacement can have an "index" that is added to the offset. This illustrated in the next figure.

[24] See section TBD for more information.

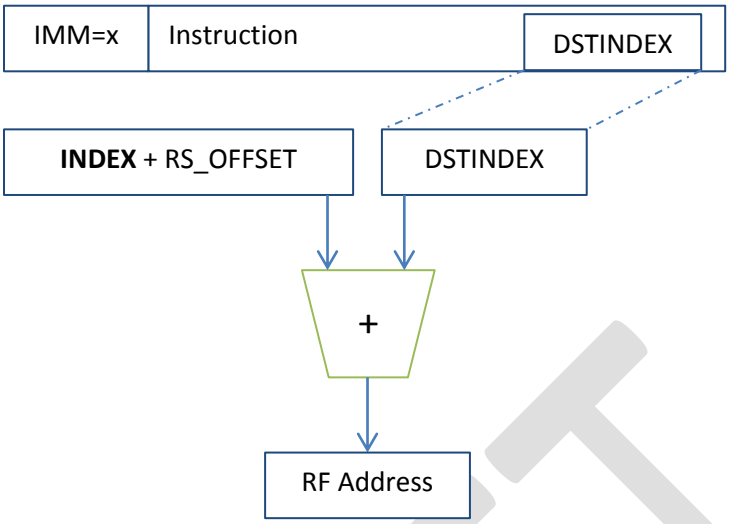

Figure 34 Displacement and Index

The additional index from Figure 34 is used to de-reference arrays.

In indirect addressing mode the instruction destination is the **content** of the register file location pointed by the DSTINDEX field from Table 14. In other words, the index of DSTINDEX is used as a *pointer*, pointing to a memory location in the RF where the actual index will reside.

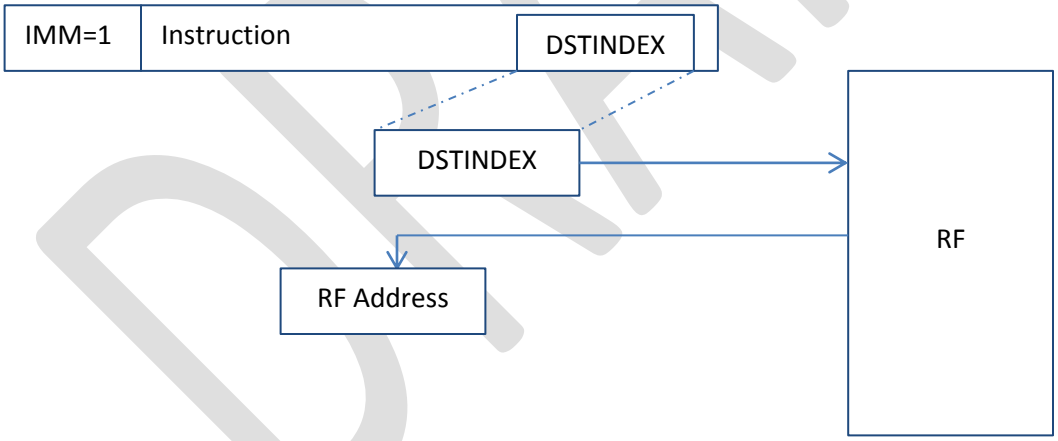

Figure 35 Indirect addressing mode

Once again, the value pointed by DSTINDEX is added the OFFSET SPR. This concept is illustrated in Figure 36.

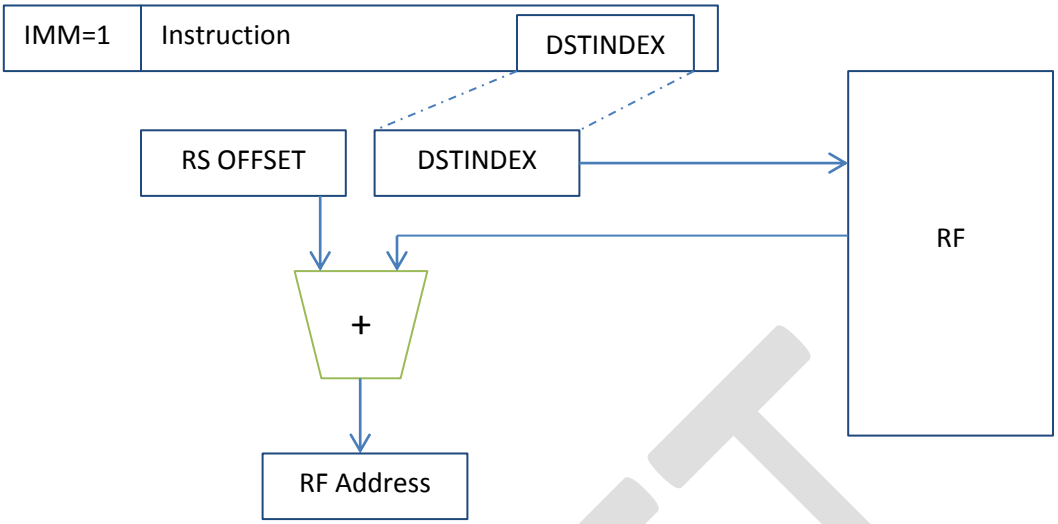

Figure 36 Indirect addressing with displacement

One important thing about the VP architecture is that only some instructions support the indirect addressing mode while other instructions (most of them in fact) only support direct addressing mode. This means that the instruction set is not *orthogonal*. This decision was made in order to remove complexity from the decoding logic. Figure 37 shows an example of an instruction using indirect addressing.

```
//Assume that the OFFSET register is zero

ADD R1.x 0x7                          //R1 = 0x7

ADD <BRANCH_ALWAYS> *R1.x 0x1         //Jump to content of R
```

Figure 37 indirect addressing example

In the example from Figure 37, the first addition stores the immediate value 0x7 into the register file location R1 (using direct addressing). Then, the second ADD operation executes a branch (see section 3.16 for more details on branching); the destination content of the register R1 (0x7) is used as the branch destination.

Finally, Table 17 in the following sections presents the internal encoding of the addressing mode inside the instruction word. This table may seem long, but is just an expanded version of the direct/indirect addressing modes plus offset as it has been described. Table 17 also shows that it is possible to apply the addressing modes to the SRC0 and SRC1 operands to use them as pointers under some configurations.

3.7. Instruction word fields

Section 3.5 presented a brief overview of the VP instructions; it showed how the VP instruction words are broken down into “sections”. Each of these sections is broken down into fields.

This chapter is dedicated to specify all of the fields in each instruction section and describe its functionality.

The next tables summarize the various instruction fields for the Operation, Destination, Source1 and Source0 sections.

The first section to summarize is the *Operation* Section. The Operation section contains information regarding which arithmetic operation will be performed, the type of instruction (using immediate value or not using immediate value), branch information etc. The next table summarizes these concepts.

Table 13 Instruction Operation section fields

Field name	Range	Description
IMM	63	Immediate operation bit. If this bit is set to 1, then the 32 least significant bits of the instruction will be interpreted as the literal value IMMV. See Figure 19.
SCOP/LOP	62:59	Scale modifier. This determines how the scale modifier is to be applied. See section 3.13 for more details. For logic operations it chooses the logic operation to perform see section <TBD> for more details.
EOF	58	End of flow bit.
BBIT	57	Branch bit. See section 3.16for details.
BOP	56:54	Branch operation. See section 3.16 for details.
RESERVED	53:51	Reserved for future use.
OPCODE	50:48	Operation code. See section 3.9 for details.

The next section is the destination section. The destination section has to do with how the destination address is resolved.

Table 14 Instruction Destination section fields

Field name	Range	Description
MODE	47:45	Addressing mode, see Table 17.
WEX	44	Destination write enable X. If this bit is set to 1, then the channel X result from the VP will be stored at the channel X slot of the destination register DSTADDR, in the register file RF.

WEY	43	Destination write enable Y. If this bit is set to 1, then the channel Y result from the VP will be stored at the channel Y slot of the destination register DSTADDR, in the register file RF.
WEZ	42	Destination write enable Z. If this bit is set to 1, then the channel Z result from the VP will be stored at the channel Z slot of the destination register DSTADDR, in the register file RF.
DSTINDEX	41:34	

3.8. Addressing mode encoding

The next two tables specify the values of the SRC1, SRC0 and DSTADDR for the various addressing mode encodings. There are two main encodings: one when IMM = 0 and one when IMM = 1.

Table 15 Addressing mode encoding IMM = 0.

z/off1/off0 (IMM=0)	SRC1	SRC0	DSTADDR
000	R[SRC1INDEX]	R[SRC0INDEX]	DSTINDEX
001	R[SRC1INDEX]	R[SRC0INDEX + OFFSET]	DSTINDEX + OFFSET
010	R[SRC1INDEX +OFFSET]	R[SRC0INDEX]	DSTINDEX
011	R[SRC1INDEX + OFFSET]	R[SRC0INDEX + OFFSET]	DSTINDEX + OFFSET
100	R[SRC1INDEX]	R[SRC0INDEX]	DSTINDEX
101	R[SRC1INDEX]	R[SRC0INDEX + OFFSET]	DSTINDEX + OFFSET
110	R[SRC1INDEX + OFFSET]	R[SRC0INDEX]	DSTINDEX
111	R[SRC1INDEX + OFFSET]	R[SRC0INDEX + OFFSET]	DSTINDEX + OFFSET

Table 16 Addressing mode encoding IMM = 1.

z/off1/off0 (IMM=1)	SRC1	SRC0	DSTADDR
000 *branch	IMMV	R[**DSTINDEX**]	DSTINDEX
001 *	IMMV	R[DSTINDEX]	DSTINDEX + OFFSET
010	R[SRC1INDEX]	R[SRC0INDEX+OFFSET + **RINDEX**]	DSTINDEX + OFFSET
011*array	0	R[SRC0INDEX + OFFSET]	DSTINDEX + OFFSET + **SRC1[X]**
100*assign	IMMV	0	DSTINDEX
101*assign	IMMV	0	DSTINDEX + OFFSET

110	R[SRC1INDEX + OFFSET+ RINDEX]	0	DSTINDEX+OFFSET
111 * array	R[SRC1INDEX + OFFSET+ RINDEX]	R[SR0INDEX+OFFSET]	DSTINDEX + OFFSET

The detail on how the addressing mode word is specified in Table 17. This table assumes the convention:

R[DSTADDR] = SRC1 OPCODE SRC0

This may seem as a rather large list but is simply a set of possible 'flavors' of the direct or indirect addressing described in section 3.6.

Table 17 Addressing mode encoding.

IMM/MODE	Description
0 000	**Direct**: The Indexes from SCR1, SRC0 and DST are directly used to calculate the corresponding addresses in the RF. DSTADDR = DSTINDEX SRC1 = R[SRC1INDEX] SRC0 = R[SRC0INDEX]
0 001	**Direct with displacement**: SRC0INDEX is added OFFSET and then used to calculate SRC0ADDR in RF. DSTADDR = DSTINDEX SRC1 = R[SCR1INDEX] SRC0 = R[SRC0INDEX + OFFSET]
0 010	**Direct with displacement**: SRC1INDEX is added OFFSET and then used to calculate SRC1ADDR in RF. DSTADDR = DSTINDEX SRC1 = R[SCR1INDEX + OFFSET] SRC0 = R[SRC0INDEX]
0 011	**Direct with displacement**: SRC1INDEX is added OFFSET and then used to calculate SRC1ADDR in RF. SRC0INDEX is added OFFSET and then used to calculate SRC0ADDR in RF. DSTADDR = DSTINDEX SRC1 = R[SCR1INDEX + OFFSET] SRC0 = R[SRC0INDEX + OFFSET]
0 100	**Direct with displacement**: DSTINDEX is added OFFSET and then used to calculate DSTADDR in RF. DSTADDR = DSTINDEX + OFFSET SRC1 = R[SCR1INDEX] SRC0 = R[SRC0INDEX]

0 101	**Direct with displacement**: DSTINDEX is added OFFSET and then used to calculate DSTADDR in RF. SRC0INDEX is added OFFSET and then used to calculate SRC0ADDR in RF. DSTADDR = DSTINDEX + OFFSET SRC1 = R[SCR1INDEX] SRC0 = R[SRC0INDEX + OFFSET]
0 110	**Direct with displacement**: DSTINDEX is added OFFSET and then used to calculate DSTADDR in RF. SRC1INDEX is added OFFSET and then used to calculate SRC1ADDR in RF. DSTADDR = DSTINDEX + OFFSET SRC1 = R[SCR1INDEX + OFFSET] SRC0 = R[SRC0INDEX]
0 111	**Direct with displacement**: All the indexes from SRC1, SRC0 and DST are displaced by the OFFSET. DSTADDR = DSTINDEX + OFFSET SRC1 = R[SCR1INDEX + OFFSET] SRC0 = R[SRC0INDEX + OFFSET]
1 000*	**Direct with IMMV**: The 32-bit immediate (literal) value IMMV is used as SRC1, the value of the register pointed by DSTINDEX is used as SRC0.[25] DSTADDR = DSTINDEX SRC1.x = IMMV SRC1.y = IMMV SRC1.z = IMMV SRC0 = R[DSTINDEX]
1 001*	**Direct with IMMV and displacement**: Combines displacement and direct addressing. DSTADDR = DSTINDEX + OFFSET SRC1.x = IMMV SRC1.y = IMMV SRC1.z = IMMV SRC0 = R[DSTINDEX + OFFSET]
1 010	**Indirect with non-immediate** DSTADDR = R[DSTINDEX + **SRC1[7:0]**] SRC1 = R[SRCINDEX1] SRC0 = R[SRCINDEX1]
1 011	**Indirect with non-immediate and offset:** This is used to store the results of the instruction directly into array elements. (there is a traversal algorithm (see section TBD) which makes heavy use of an array (working as a stack) this is why is

[25] In other words what this does is: R[DSTINDEX] = IMMV **OPERATION** R[DSTINDEX], where operation is one of the operations from Table 6.

	necessary for the instruction set to support storing directly into array elements) DSTADDR = R[DSTINDEX + OFFSET + **SRC1[7:0]**] SRC1 = R[SRCINDEX1 + OFFSET] SRC0 = R[SRCINDEX0 + OFFSET] R[DSTADDR + **SRC1[X_RNG]** + OFFSET] = SRC0
1 100*	**Indirect with IMMV and Zero**: DSTADDR = DSTINDEX SRC1.x = IMMV SRC1.y = IMMV SRC1.z = IMMV SRC0.x = 0 SRC0.y = 0 SRC0.z = 0
1 101*	**Indirect with IMMV, displacement and clear SRC0**: Combines displacement, indirect addressing and zeroing of SRC0. DSTADDR = DSTINDEX + OFFSET SRC1.x = IMMV SRC1.y = IMMV SRC1.z = IMMV SRC0.x = 0 SRC0.y = 0 SRC0.z = 0
1 110	
1 111	**Indirect with non-immediate and offset:** This is used to store the results of the instruction directly into array elements. (there is a traversal algorithm (see section TBD) which makes heavy use of an array (working as a stack) this is why is necessary for the instruction set to support storing directly into array elements) DSTADDR = R[DSTINDEX + OFFSET + SRC1] SRC1 = R[SRCINDEX1] SRC0 = R[SRCINDEX0] **R[DSTADDR + SRC1 + OFFSET] = SRC0**

The next 2 tables are the fields from the instruction source sections. Both SRC1 and SRC0 sections have a similar layout. The values from these next two tables are especially important for the SMU in order to do the source modifications.

Table 18 Instruction Source 1 section fields

Field name	Range	Description

SIGN1X	33	Source 1 sign X bit.
SIGN1Y	32	Source 1 sign Y bit.
SIGN1Z	31	Source 1 sign Z bit.
SWZZ1X	30:29	Source 1 swizzle X. See section 3.15 for details.
SWZZ1Y	28:27	Source 1 swizzle Y. See section 3.15 for details.
SWZZ1Z	26:25	Source 1 swizzle Z. See section 3.15 for details.
SRC1ADDR	17:24	Source 1 Address in RF.

Table 19 Instruction Source 0 section fields

Field name	Range	Description
SIGN0X	16	Source 0 sign X bit.
SIGN0Y	15	Source 0 sign Y bit.
SIGN0Z	14	Source 0 sign Z bit.
SWZZ0X	13:12	Source 0 swizzle X. See section 3.15 for details.
SWZZ0Y	11:10	Source 0 swizzle Y. See section 3.15 for details.
SWZZ0Z	9:8	Source 0 swizzle Z. See section 3.15 for details.
SRC0ADDR	7:0	Source 0 Address in RF.

3.9. Selecting the Arithmetic operation

The arithmetic operations were briefly introduced in section 3.5.1. This section will provide more details on how the instruction determines the arithmetic operations.

The arithmetic operation within the instruction is controlled by the OPCODE field from Table 20. After the IIU fetches an instruction it decodes the OPCODE in order to select the appropriate reservation station to execute the OPCODE.

Table 20 Instruction OPCODE field values

OPCODE	Name	Description
000	NOP	A NOP operation is issued by IIU.[26]
001	ADD	Integer Addition. [27]
010	DIV	Integer division.
011	MUL	Integer multiplication.

[26] **Note**: The NOP is actually sent into the IBUS with an RSID equal to zero. Since no reservation station has the number zero as RSID then the NOP issue will be ignored by all the reservation stations and no operation will be performed.

[27] **Note**: In order to perform a subtraction, the sign of one of the operands must be set to negative. See section 3.14 for details.

100	SQRT	Integer square root. See section 3.10 for details.
101	LOGIC	Bitwise logic operations. The specific logic operation is chosen by setting the appropriate value into the SCOP/LOP field under the operation instruction section. See section <> for details.
110	IO	Input/Output operations see section 6 for details.
111	RSVR2	RESERVED.

The NOP, ADD, DIV and MUL operations from Table 20 are very straight forward. The square root operation is a special case that is briefly explained in the next section.

3.10. Fixed point Square Root unit

As shown in Table 20, THEIA features an execution unit called SQRT which is dedicated to calculate square roots. The SQRT unit has been designed to be very fast, but in return for that speed the SQRT unit has a number of limitations. These limitations are related to the fact that the SQRT has been implemented using a LUT, so SQRT can only calculate square roots for values that are constrained within a certain range of numbers and only for fixed point numbers.

The SQRT can only calculate square roots for numbers that are between 0 and 127. This may seem like a small range at first, but consider the following property of square roots illustrated with this example:

$$\sqrt{x} = \sqrt{64\frac{x}{64}} = 8\sqrt{\frac{x}{64}} \qquad (3)$$

So, if the number X from (3) is not between 0 and 127, then the number X can be divided by a power of 2 until it results in a number which is can be found within the range of numbers stored in the LUT. Then the result from the LUT is multiplied back in order to get the result as in (3)[28].

As it was mentioned earlier, SQRT only operates on fixed point numbers. The fixed point numbers have an associated SCALE as described in section 3.4. The SQRT unit uses an LUT (ROM) to store the square roots using a fixed point representation; this means that the SCALE is fixed for the SQRT unit. Since the SQRT scale is fixed, then is the compiler's responsibility to apply the appropriate scale operation (see section 3.4) to the input arguments when issuing instructions into the SQRT unit.

The next table summarizes the limitations and special conditions of the SQRT unit.

Condition	Description
Fixed point Scale	The fixed point scale used by the SQRT unit is 17.
Numeric Range	The range of number is between 0 and 64*127. If the SQRT attempts to calculate a value outside if this range, then an arithmetic error condition is generated. See section <TBD> for details.

[28] This multiplications and division by powers of two are implemented as shift operations.

Decimal truncation	The when the fixed point number is between the range, then it is truncated to the closest value on the LUT in order to calculate the square root.

3.11. Bitwise logic operations

The VP can perform bitwise logic operations by setting the value 3'b101 in the OPCODE field from table <> and then choosing among one of the following possible bitwise operations from table <>.

SCOP/LOP	Name	Description
000	AND	Bitwise AND
001	OR	Bitwise OR
010	NOT	Bitwise NOT
011	SHL	Shift left
100	SHR	Shift right

Add more description here

3.12. Destination write channel control

Each VP instruction has the ability to specify the individual 32-bit destination blocks where the result will be written back into the RF. This was briefly introduced in section 3.5.2. A VP instruction can choose to alter the 3 32-bit destination blocks (X, Y and Z) or to selectively write only to some blocks, for example storing the results into the X block only, or storing the results into the Z and the Y but not altering the X block.

The way to control where to store the results is by using the WEX, WEY and WEZ Instruction bits from the instruction destination section in Table 13. Table 21 lists all the possible WEX, WEY and WEZ values.

Table 21 Write channel control bit values

BBIT WEX WEY WEZ	Description
0 000	No values are written to DSTADDR.[29]
0 001	The result Z value is written to the DSTADDR Z block.
0 010	The result Y value is written to the DSTADDR Y block.
0 011	The result Z value is written to the DSTADDR Z block AND the result Y value is written to the DSTADDR Y

[29] This is especially useful for branches. It is in general not desired that a branch operation writes values to the RF.

	block.
0 100	The result X value is written to the DSTADDR X block.
0 101	The result X value is written to the DSTADDR X block AND the result Z value is written to the DSTADDR Z block.
0 110	The result X value is written to the DSTADDR X block AND the result Y value is written to the DSTADDR Y block.
0 111	The result X value is written to the DSTADDR X block AND the result Y value is written to the DSTADDR Y block AND the result Z value is written to the DSTADDR Z block.
1 xxx	No values are written to DSTADDR. The branch logic is activated.[30]

It is important to note from Table 21 that if a given X, Y or Z value is not written by the instruction, then the previous (old) value will remain in the RF.

3.13. Operand Scale control

Each VP instruction has the ability to specify the optional scale operation for the input arguments SCR1 and SCR0. The scale operation shifts the x, y and z blocks of the specific source register by SCALE number of bits. The scale operation is controlled by the SCOP instruction field from Table 13. The input operands can be scaled to the left or can be scaled to the right depending on the value of the SCOP field as specified in Table 22.

Table 22 input operand scale control

SCOP	Description
000	No scale changes are applied to SRC1 or SCR0.
001	SRC1 << SCALE
010	SRC0 << SCALE
011	SRC1 << SCALE *AND* SCR0 << SCALE
100	Reserved
101	SRC1 >> SCALE
110	SRC0 >> SCALE
111	SRC1 >> SCALE *AND* SCR0 >> SCALE

It is important to remember that the scale operations from table<> modifies the individual x, y and z blocks of the corresponding register, for example doing SCR0 << SCALE is really doing:

[30] See section 3.16

(SRC0.x << SCALE, SCR0.y << SCALE, SCR0.z << SCALE)

So each x, y and z block is scaled individually.

The Scale operation is used to perform the operand scaling necessary for the fixed point arithmetic operations[31]. The default value for the SCALE is 17 as defined in section **Error! Reference source not ound.**, and can be changed in the control register CNTREG.

3.14. Operand Sign control

Each VP instruction has the ability to change to sign of any given X, Y or Z block from any of the two operand values. The sign change is applied by performing a 2 complement of the selected X, Y or Z value. The sign control is very important since the VP doesn't actually have a subtraction OPCODE (see Table 20), therefore the sign control allows the SRC1 to be complemented in order to execute a subtraction. Also note that the individual X, Y or Z blocks can be negated, this combined with the operand "swizzling" allows for more complex operations as we will see in the next sections. The next tables define how the sign is controlled using the SIGN* bits from the instruction SRC1 and SRC0 fields.

Table 23 SRC1 Sign control

SIGN1X SIGN1Y SIGN1Z	Description
000	No sign changes are applied to SRC1.
001	SRC1 Z sign is inverted.
010	SRC1 Y sign is inverted.
011	SRC1 Y sign is inverted AND SRC1 Z sign is inverted.
100	SRC1 X sign is inverted.
101	SRC1 X sign is inverted AND SRC1 Z sign is inverted.
110	SRC1 X sign is inverted AND SRC1 Y sign is inverted.
111	SRC1 X sign is inverted AND SRC1 Y sign is inverted AND SRC1 Z sign is inverted.

Table 24 SRC0 Sign control

SIGN0X SIGN0Y SIGN0Z	Description
000	No sign changes are applied to SRC1.
001	SRC0 Z sign is inverted.
010	SRC0 Y sign is inverted.
011	SRC0 Y sign is inverted AND SRC0 Z sign is inverted.

[31] see section 3.4

100	SRC0 X sign is inverted.
101	SRC0 X sign is inverted AND SRC0 Z sign is inverted.
110	SRC0 X sign is inverted AND SRC0 Y sign is inverted.
111	SRC0 X sign is inverted AND SRC0 Y sign is inverted AND SRC0 Z sign is inverted.

3.15. Operand swizzle control

Operand swizzle consists of re-ordering the x, y and z blocks of the instruction input operands. Each individual x, y or z operand block can be replaced by one of the x, y or z blocks in the same operand. This is done by means of the SWZZL* fields from Table 18 and Table 19. The next tables define all the possible input operand swizzle combinations.

Table 25 SRC1 Swizzle control X

SWZZ1X	Description
00	Operand1.x is not modified.
01	Operand1.x is replaced by Operand1.z
10	Operand1.x is replaced by Operand1.y
11	Reserved

Table 26 SRC1 Swizzle control Y

SWZZ1Y	Description
00	Operand1.y is not modified.
01	Operand1.y is replaced by Operand1.z
10	Operand1.y is replaced by Operand1.x
11	Replaced

Table 27 SRC1 Swizzle control Z

SWZZ1Z	Description
00	Operand1.z is not modified.
01	Operand1.z is replaced by Operand1.y
10	Operand1.z is replaced by Operand1.x
11	Reserved

Table 28 SRC0 Swizzle control X

SWZZ0X	Description
00	Operand1.x is not modified.
01	Operand1.x is replaced by Operand1.z
10	Operand1.x is replaced by Operand1.y
11	Reserved

Table 29 SRC0 Swizzle control Y

SWZZ0Y	Description
00	Operand1.y is not modified.
01	Operand1.y is replaced by Operand1.z
10	Operand1.y is replaced by Operand1.x
11	Reserved

Table 30 SRC0 Swizzle control Z

SWZZ0Z	Description
00	Operand1.z is not modified.
01	Operand1.z is replaced by Operand1.y
10	Operand1.z is replaced by Operand1.z
11	Reserved

There are 2 separate combinatorial blocks in the SMU dedicated to do the input operand swizzle[32], one for each of the two possible instruction operands. This is shown in Figure 38.

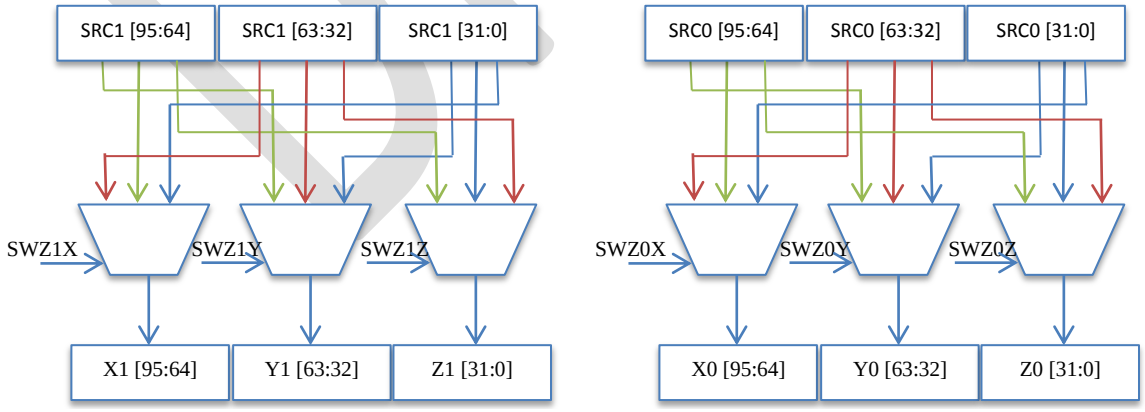

Figure 38 Operand swizzle logic

[32] Note: The instruction result **cannot** be swizzled.

3.16. Branching operations

Branching is done by setting to 1 the BBIT in the instruction's operation field. After the instruction's result has been committed by the execution units, the IIU will check the values from the ZFLAG and the SFLAG against Table 31 to decide if the branch was taken or not taken.

Table 31 Branch operation BOP values

BBIT/ BOP	Description
1 000	Unconditional Branch.
1 001	Branch if ZFLAG is 1
1 010	Branch if ZFLAG is 0
1 011	Branch if SFLAG is 1
1 100	Branch if SFLAG is 0
1 101	Branch if ZFLAG is 1 OR SFLAG is 1
1 110	Branch if ZFLAG is 1 OR SFLAG is 0
1 111	Reserved
0 xxx	No branch is performed

The branch decisions from Table 31 are further predicated by Table 32. Since the both the SFLAG and ZFLAG have x, y and z values corresponding to the individual x, y and z result blocks, Table 32 describes which x, y and z values from the ZFLAG and SFLAG are used by Table 31 in order to make the final branch decision.

Table 32 Branch operation predicates.

BBIT / WE	Description
1 000	Reserved
1 001	Use only z values of ZFLAG and SFLAG to make the branch decision.
1 010	Use only y values of ZFLAG and SFLAG to make the branch decision.
1 011	Use only y and z values of ZFLAG and SFLAG to make the branch decision.
1 100	Use only x values of ZFLAG and SFLAG to make the branch decision.
1 101	Use only x and z values of ZFLAG and SFLAG to make the branch decision.
1 110	Use only x and y values of ZFLAG and SFLAG to make the branch decision.
1 111	Use x, y and z values of ZFLAG and SFLAG to make the branch decision

0 xxx	The WE action is controlled by Table 21

Branching can usually be achieved by configuring the VP to perform a subtraction (this is an addition with SRC0 sign bits set to 1) and then checking the SFLAG and ZFLAG to see if the source registers were equal, greater, etc. according to Table 31 and Table 32**.**

It is important to note that nothing prevents the compiler from choosing to execute an operation different from a subtraction and then checking the results of this operation against Table 31 to determine the branch.

3.16.1. Unconditional branches

Unconditional branches are branches which are always taken. In order to set a branch as unconditional, the BOP field has to be set to zero as specified in Table 31. Unconditional branches can either branch into an effective address (EA) specified as an immediate value or can branch into an effective address specified as the content of a register. Table 33 and Table 34 illustrate the previous concepts.

Table 33 Unconditional branch with branch destination as immediate value

Field name	Range	Value
IMM	63	0
WEX	62	0
WEY	61	0
WEZ	60	0
BP	59	0
EOF	58	0
BBIT	57	1
BOP	56:54	000
OPCODE	48:53	Any operation but NOP. See Table 20
DSTINDEX	41:34	Literal represent the EA to branch into.

Table 34 Unconditional branch with branch destination stored in a register

Field name	Range	Value
IMM	63	1
WEX	62	0
WEY	61	0
WEZ	60	0
BP	59	0
EOF	58	0
BBIT	57	1
BOP	56:54	000

OPCODE	48:53	Any operation but NOP. See Table 20
DSTINDEX	41:34	Literal represent the register index where the EA to branch will be read.

3.16.2. Conditional Branches

For conditional branches, the IMM bit has to be clear to zero. THEIA does not allow using immediate values as part of the sources to determine a conditional branch. The source values for branches shall always be stored in registers.

Table 35, Table 36 and Table 37 show a possible scenario where the compiler would configure the VP to perform a conditional branch by checking the ZFLAG and SFLAG after a subtraction.

Table 35 Example of Instruction operation for a conditional branch instruction

Field name	Range	Value
IMM	63	0
WEX	62	0
WEY	61	0
WEZ	60	0
BP	59	0
EOF	58	0
BBIT	57	1
BOP	56:54	B2 B1 B0. See Table 31.
OPCODE	48:53	ADDITION. See Table 20.

Table 36 Example of Instruction Destination for conditional branch instruction.

Field name	Range	Value
DSTZERO	47	DON'T CARE
RESERVED	46:34	Branch address.

Table 37 Example of Instruction Sources for a conditional branch instruction

Field name	Range	Value
SOURCE1	33:17	Any valid combination as described in <>
SIGN0X	16	1
SIGN0Y	15	1
SIGN0Z	14	1

SRC0ADDR	7:0	Source 0 address in RF.

4. VP Data path

Now that the various instruction fields have been described in the previous sections, it is time to give a brief walk-through of the VP data path. The VP data path follows the path of the instruction and data from the IM all to way down to the RF. There are several data structures and special values that get added or removed along the way; this is illustrated in Figure 39. Figure 39 uses a series of acronyms such as DSTADDR, SC, WE, etc. These acronyms come from the previous sections.

Table 38 Data path fields.

Field name	Section	Description
RSID		Reservation station ID determined by the IIU
DSTADDR		Destination address determined by the IIU
SC	Table 13	Scale control
WE	Table 14	The WE.x WE.y and WE.z from the Destination section.
RSID1		The ID of the reservation station which is currently calculating the data dependency for SRC1. (Zero means no dependency)
RSID0		The ID of the reservation station which is currently calculating the data dependency for SRC0. (Zero means no dependency)
SRC1		The 96 bit value (32 * 3) representing the instruction Source 1.
SRC0		The 96 bit value (32 * 3) representing the instruction Source 1.

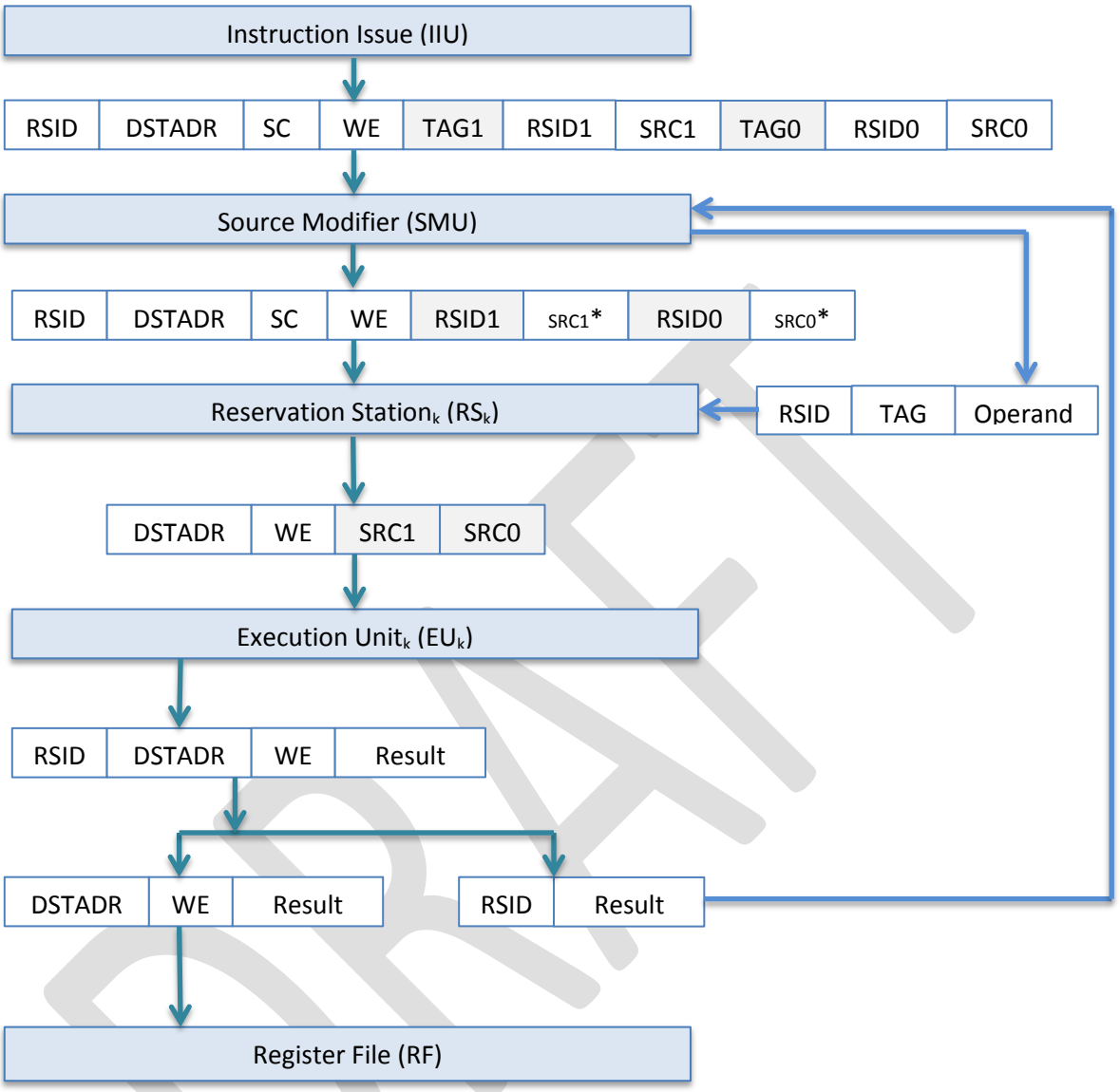

Figure 39 VP data path Walk Through

The walkthrough starts with the instruction reaching the IIU. The IIU is in charge of decoding the instruction and generating a decoded packet. This decoded packet is composed of various fields as it is shown in Figure 39. The decoded packet is reprinted in Figure 40 for clarity.

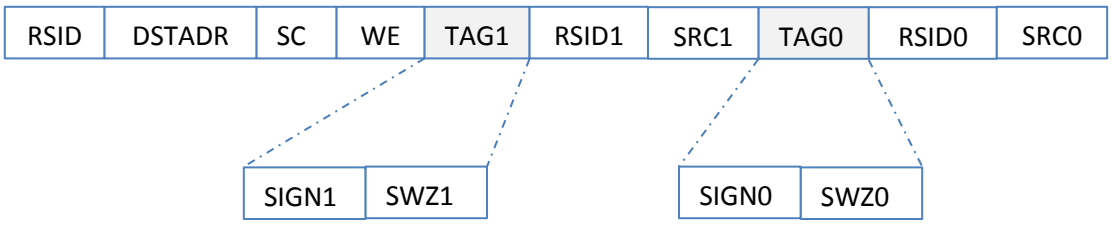

Figure 40 The decoded instruction presented by the IIU to the SMU

The first field from Figure 40 is the RSID which is simply the numerical index of the reservation station that is required to handle this issue, this calculated using Table 20. The next field is the DSTADR, this is the effective address calculated using Table 17. Next is the SC field, this is the scale field taken from the Table 13. Next is the WE (Write Enable) field which is taken from Table 14. Next is the TAG1. The TAGs are simply the SIGN+SWIZZLE for each SRC0 or SRC1 operands. The RSID1 field is the reservation station index of the RS that is resolving the data dependency of SRC1 (zero in case there are no dependencies), similarly RSID0 is the index of the RS resolving the data dependency of SRC0. SRC1 and SCR0 are the 96 bit wide values of the source operands taken from the RF.

This decoded packet from Figure 40 is presented to the SMU. The SMU looks at the TAG1 and TAG0 fields. If any of these fields is non-zero, then the SMU applies the corresponding data modifications according to Table 22, Table 23, Table 24, Table 25, Table 26, Table 27, Table 28, Table 29 and Table 30.

The output from the SMU is modified packet that is reprinted from Figure 39 by Figure 41.

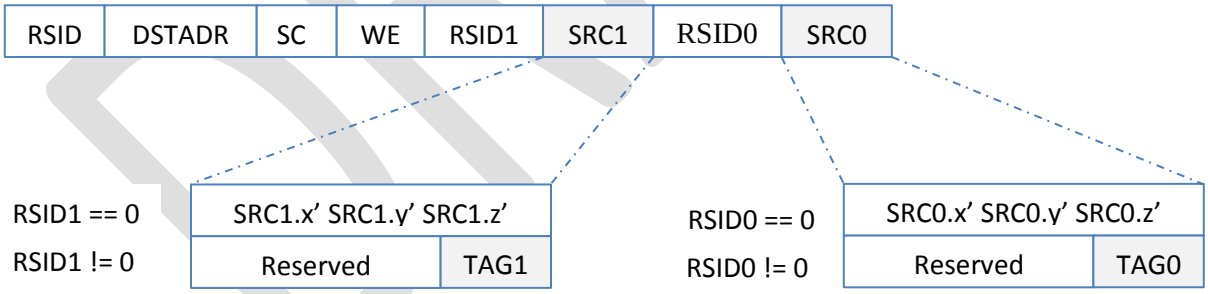

Figure 41 The packet presented by the SMU to the reservation stations (RS).

All of the fields from Figure 41 have been already mentioned. There are a couple of important things to see from Figure 41. The first thing is that if the RSID1 is zero (meaning that there are no data dependencies), then the SMU simply applies the scaling, swizzling and sign modifications to (SRC1.x, SRC1.y, SRC1.z) so that is becomes (SRC1.x', SRC1.y', SRC1.z'). If RSID1 is not zero, then SMU cannot apply the source modifications, instead the SMU inserts theTAG1 into the least significant bits of SRC1.

The same thing happens with SRC0.

Following Figure 39, the output packet from the SMU is broadcasted to the reservation stations. Only the RS who's RSID matches the RSID field from Figure 41 will handle the issue request. It is important to mention that an issue request is targeted to a single RS, in other words it is not possible for two or more RSs to handle the same issue request at any given point in time. Once the packet from Figure 41 reaches an RS two things can happen: either there are no data dependencies (RSDI1 and RSID0 are both zero) and the instruction is passed directly to the corresponding EU, or there is at least one data dependency and then the RS will wait until the dependency becomes available from the SMU.

In the scenario where there are no data dependencies, the RS will trigger the EU so that the arithmetic or logic operation starts executing. A number of clock cycles after the RS triggers the EU, the results from the operation are obtained. These results need to follow several paths as illustrated in Figure 39. One of these paths connects the results directly with the RF. The RF only needs information regarding where to write the result values (DSTADDR) and which of the X, Y or Z channels to update (WE), and of course the actual data to write. Another possible path for the results is to be connected from the EU back into the SMU. This is used in order to resolve data dependencies which are pending a swizzle, scale or sign change (for an example see section 3.5.4)

4.1.1. Instruction issue unit (IIU)

The instruction issue unit (IIU) is responsible of fetching the next instruction from the IM, decoding the instruction, selecting the appropriate reservation stations, issuing the instruction into the IBUS or stalling the machine when the stalling conditions from Table 12 are met.

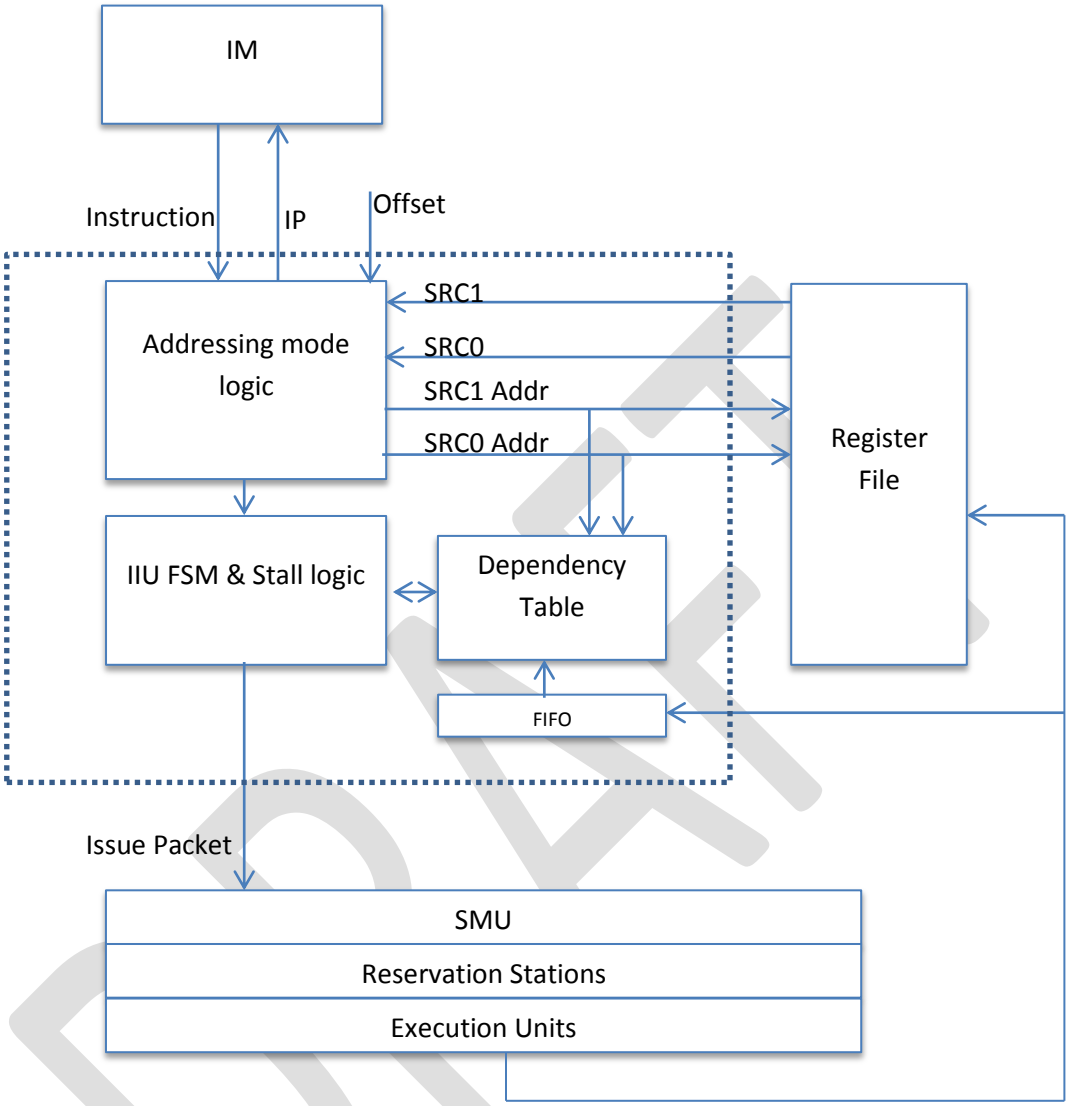

Figure 42 Block diagram of the IIU

Figure 42 shows a block diagram of the IIU. The IIU interfaces with the instruction memory (IM), the SMU and the Register File (RF). The main inputs to the IIU are the Instruction from the IM and the SRC0 and SRC1 data from the RF and the main output is the Issue-Packet send to the SMU; please refer to Figure 39 for a detailed description of these packets.

The IIU is responsible of generating the next instruction pointer (IP). This IP is send to the IM and after 1 clock cycle the requested instruction reaches the IIU. Once the instruction arrives at the IIU, the SRC0 address and SRC1 address are decoded from the Instruction and send to the RF. One clock cycle after the SRC0 address and SRC1 address are send to the RF, the corresponding data (SRC1 and SRC0) arrive at

the IIU.

In the same clock cycle when the SRC0 and SRC1 are requested from RF, these same addresses are checked for dependencies on the *Dependency-Table*, this also takes 1 clock cycle but as mentioned earlier it happens in parallel with the SRC1 and SRC0 request from the RF. So in the next clock cycle the IIU knows the values of SRC0 and SRC1 and also knows if those two values are valid and can be used as part of the *Issue-Packet*. Depending on the instruction codification, either the SRC1 and SRC0 or the IMMV and the SRC1 are used to build the *Issue-Packet*. Also depending on the addressing mode, the DST, SRC0 or SRC1 may need to be added the *Offset* value and/or *Index* value.

A FSM takes care of the special cases during the IIU execution: branch stall conditions, dependency resolution, etc. For example, it may happen that SRC1 for the current instruction has a dependency on RS_k marked in the *Dependency-Table*. When this happens, the FSM first checks to see if the dependency is currently waiting to be updated (there is an input FIFO in the IIU to serialize incoming dependency resolutions from the EUs), if the dependency resolution is not currently pending on the FIFO then the FSM uses the dependency index RS_k value from the Dependency-Table to mark the corresponding SRC1RS section of the Issue-Packet as a dependency of RS_k.

The FSM also takes cares of stalling the IIU. The IIU can be stalled under the conditions described in Table 12. For example, if there are no free reservations stations available to handle the current instruction, then the FSM will stall until a suitable RS becomes available, also if the SMU runs out of free slots then the FSM will also stall the IIU.

Finally when all the necessary information to create the Issue-Packet has been obtained (this usually takes two clock cycles unless there is a stall) the FSM makes sure that the Dependency-Table gets updated for the current instruction and finally issues the decoded instruction to the SMU.

It is important to mention that even if most of the EU can execute in 1 clock cycle, the IIU can only issue an instruction every 2 clock cycles. This limitation is solved by the use of multithreading as we will see later on this document.

4.1.2. Source Modification unit (SMU)

The Source modification unit is a hardware block dedicated to apply the scale, swizzle and sign modifications to the data coming from IIU and to data result forwarded from the EUs. A behavioral explanation of what the SMU does is available in section 3.5.3. The next figure illustrates the structure of the SMU[33].

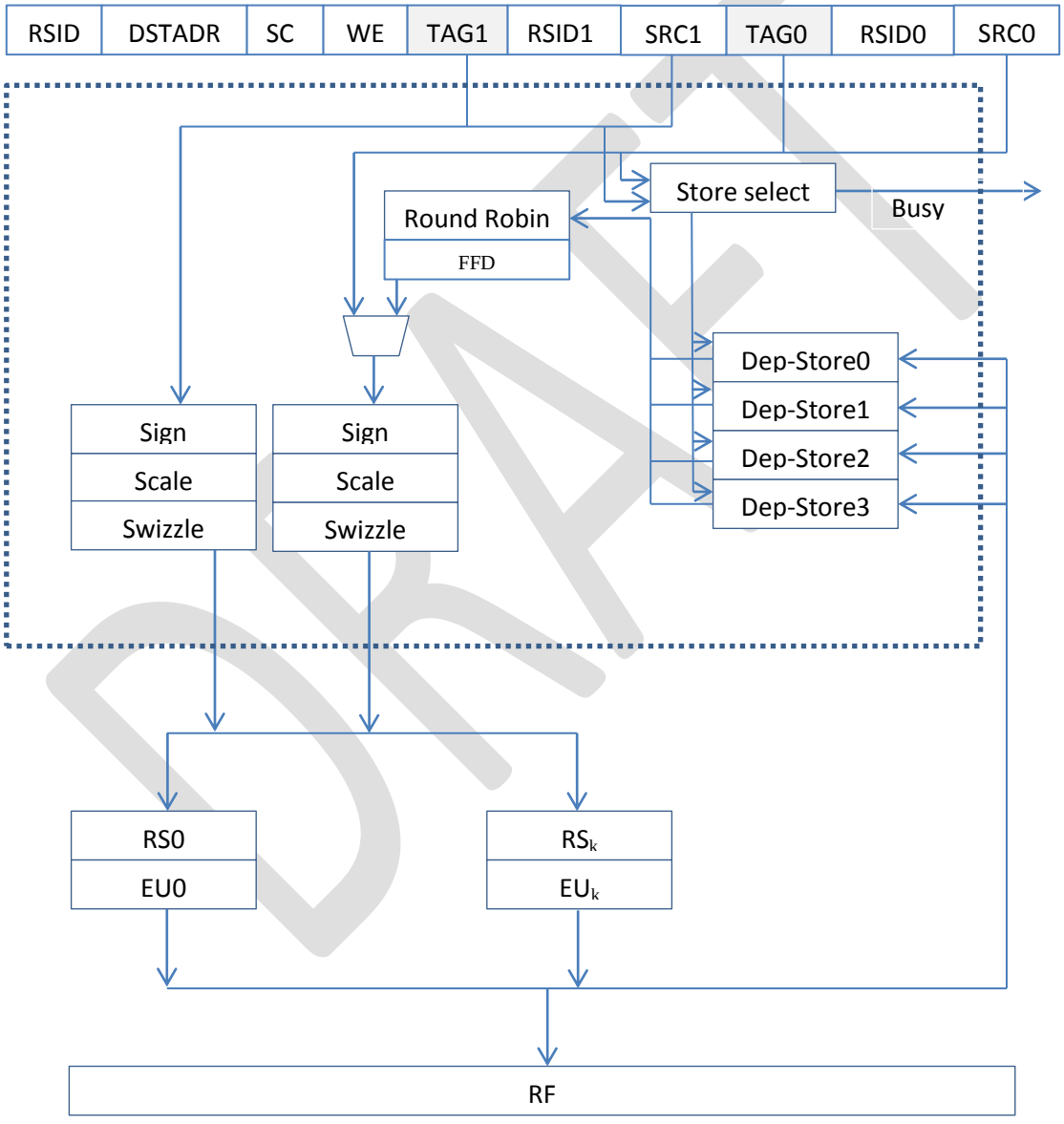

Figure 43 SMU simplified diagram

[33] This is a simplification for the sake of the discussion

The main inputs to the SMU are the Issue-Packet from the IIU and the result forwarded from the EU, and the main output is the modified-issue packet that is send to the reservation stations; please refer to Figure 39 for a detailed description of these packets.

When a packet arrives from the IIU, the SMU looks at the packet fields to see if the SRC1, SRC0 or the Result needs to be modified. If either the SRC1 or the SRC0 (or both) need a modification, then the SMU uses the two combinatorial blocks dedicated to do Scale, Swizzle and Sign modifications. If the Result needs to be modified, then the SMU updates a special field on the packet and uses the "Dep-Store" blocks to store the dependencies so that when the results are forwarded back from the EUs the modifications can be applied.

The SMU has 4 "Dep-Store" blocks. Each Dep-Store block keeps track of a single result dependency by storing a single TAG/Register pair. Every time a packet arrives from the IIU, a free Dep-Store block is used to store the dependencies for SRC1 and SRC0 (if any). If there are no free Dep-Store blocks to store the dependencies then the SMU stalls and sends a busy signal back to the IIU, indicating that it can handle no more requests.

When a result is forwarded from the EUs, the SMU broadcasts this result to the "Dep-Stores". If the result is not pending a modification on any of the Dep-Stores, then no changes are applied to it and the result is forwarded verbatim back to the reservation stations. If one or more Dep-Store has the result marked as pending for modification, then the modifications are applied and the modified results are serially forwarded to the RS, using a round robin algorithm.

4.1.2.1. Issue Bus (IBUS)

The issue bus or IBUS is a 216 bit wide shared bus which connects the IIU with the Reservation stations. The next table shows the structure of the IBUS.

Table 39 Issue bus fields

Field name	Range	Description
SCOP	218:216	The scale operation bits (see section 3.13)
DEST_ZERO	215	
RSID	214:211	The reservation station ID.
WE	210:208	Write enable bits. (see section 3.12)
DST	207:200	The destination address in RF.
SRC1RS	199:196	The SRC1 *renamed* register index according to Table 11. The value 0 means that there are no data dependencies for SRC1.
SRC0RS	195:192	The SRC0 *renamed* register index according to Table 11. The value 0 means that there are no data dependencies for SRC0.

SRC1	191:96	The 96 bit value (32x3) of SRC1.
SRC0	95:0	The 96 bit value (32x3) of SRC0.

All the reservation stations (RS) are connected to the IBUS as depicted in **Error! Reference source not ound.**. When a RSID field in the IBUS matches the RS ID, the RS reads in the issue data from the IBUS. The WE and DST fields are directly forwarded by the execution units into the CBUS.

The SRC1RS and SRC0RS are the instruction operand dependencies (aka. Renamed registers). These registers are the indexes of the RSs which are currently operating on SRC0 and SRC1 respectively (according to the Tomasulo's algorithm). A value of zero on SRC*RS means that there are no data dependencies.

4.1.2.2. Commit Bus (CBUS)

The commit bus or CBUS is a 111 bit wide shared bus which connects the execution units with the RF. The CBUS also retro-feeds into the reservation stations and reaches back into the IIU to allow for data forwarding as shown in **Error! Reference source not found.**.

Table 40 Commit bus fields

Field name	Range	Description
RSID	110:107	The ID of the reservation station currently owning the CBUS.
WE	106:104	The write enable x, y and z values (see section 3.12)
DST	103:96	The destination address in RF.
COMMIT_X	95:64	The X block of the result
COMMIT_Y	63:32	The Y block of the result
COMMIT_Z	31:0	The Z block of the result

The CBUS is a shared bus. The RF and all the Reservation stations can concurrently read from the CBUS, but only one execution unit is allowed to have write ownership of the CBUS at any given point in time. The write arbitration of the CBUS is performed by a fair round robin arbiter as shown in **Error! eference source not found.**. If only a single EU is requesting write ownership of the CBUS then the arbiter grants the ownership one clock cycle after the request. If there are multiple EUs requesting write ownership of the CBUS, then it may take up to (# of requesting EUs) clock cycles for a given EU to be granted ownership of the CBUS.

5. VP SMT (simultaneous multithreading)

As previously mentioned each VP can execute multiple HW threads in an SMT fashion. There is a separate issue unit for each thread with independent instruction pointers and dependency tables. Only one of the issue units can issue an instruction to the reservation stations at any given point in time, since many the instructions can take more than one clock cycle to complete, the instruction execution of different threads overlaps in time. Each thread has a separate variable space in the register file; this register file thread partition is done by the software[34]. These concepts are illustrated in the following figure.

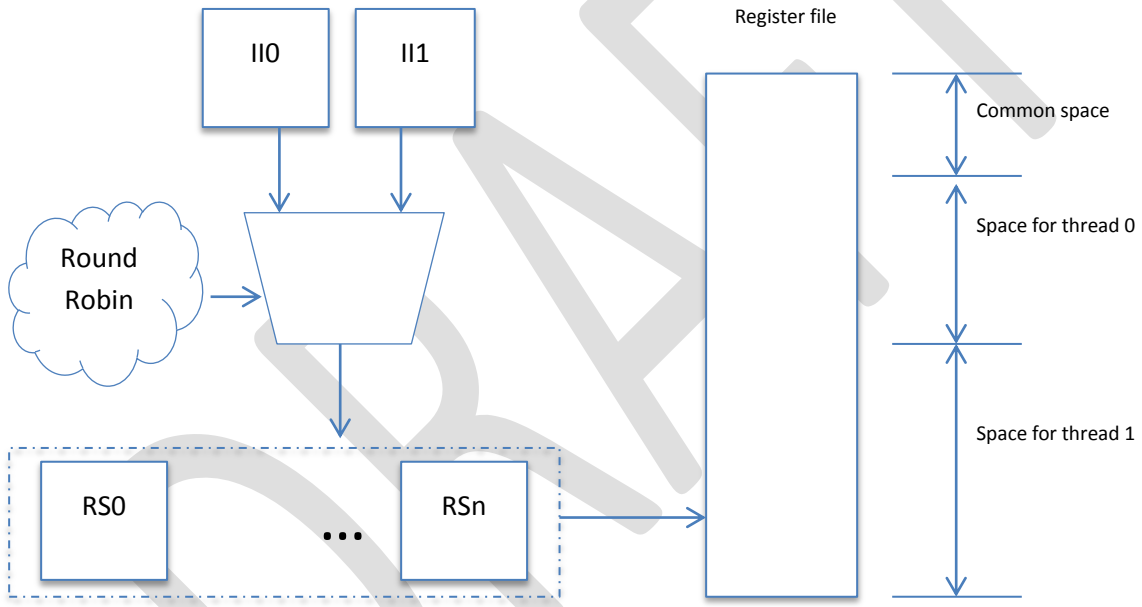

Figure 44 multithreading

The previous figure also shows that there is a common variable space in the register file for all the threads. This common variable space contains special control variables such as R0, R1, R2 and R3 which can be safely used by each thread during its own time slot.

The register R2.z controls whether the multithreading is enabled or disabled and also stores the offset of each thread variable area in the register file. It is important to note that the more active threads at a

[34] Currently this is done by the high level compiler.

given point in time the smaller is the variable space allocated for each thread in the register file. Furthermore, if a single thread is executing then this thread has access to the entire RF address space.

It is also important to note that any thread can write into the VP's OMEM resource, it is up to the programmer to keep track of how each thread access the OMEM in order to avoid data corruption or inconsistencies.

What happens if the code attempts to issue a thread and there are no more Issue units available?

6. VP IO

Each vector processor has a special reservation station dedicated to perform IO operations. The IO reservation station can handle one OMEM write operation or one TMEM read operation. The OMEM write operation takes 1 clock cycle, whereas the TMEM read operations can take multiple clock cycles depending on the traffic congestion in the TMEM cross bar.

6.1. Output memory OMEM

The OMEM is a 32-bits x TBD memory where the CORE writes its result data. These results are usually colors in RGB "true color" format, this is 8 bit per color channel plus 8 bit alpha transparency[35] = 32 bits per pixel color.

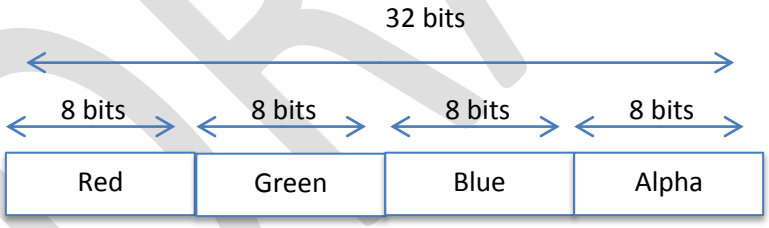

Figure 45 A typical pixel color stored as a 32 bit value in VP's the OMEM

The VP IO module in charge of the OMEM logic is called the Output Memory Interface (OMI).

[35] Alpha channel is not mandatory and sometimes is simply ignored all zero filled.

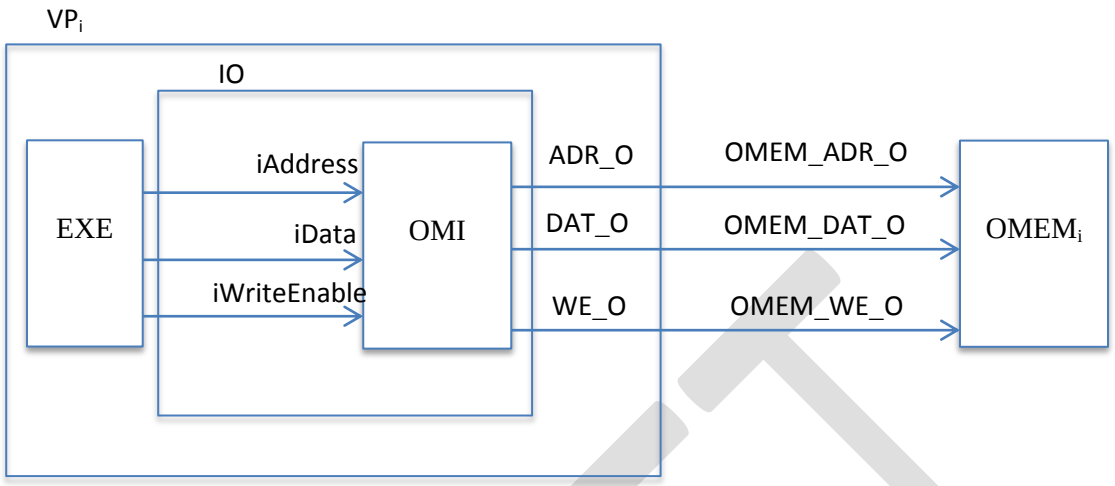

Figure 46 The OMI inside the IO unit

Figure 46 shows the signals entering and exiting the OMI and how these signals reach into the OMEM. The OMI is directly connected to the VP's EXE block, the IO Reservation station inside of the EXE provides the OMI with the 3 main input signals: iAddress, iData and iWriteEnable. The following figure illustrates how the OMI handles these inputs signals in order to write the data into the OMEM.

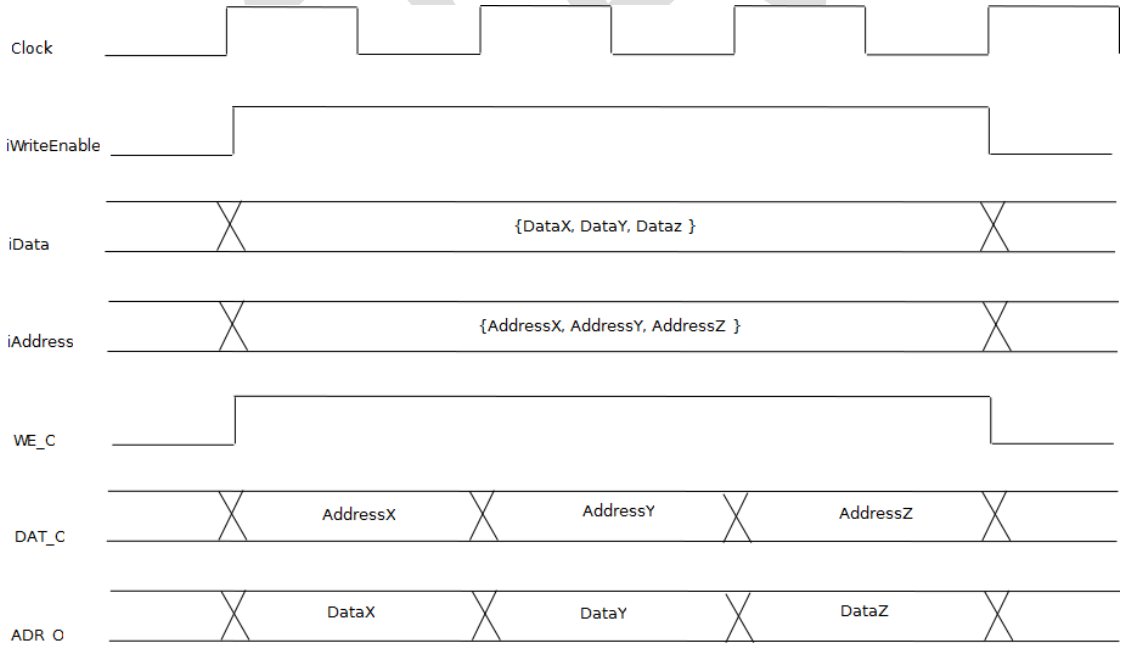

Figure 47 EXE and OMI signals

The signals iData and iAddress from Figure 47 are 96 bit wide OMI input ports. The iData signal represents a 32 bit triplet of data that the OMI will write on each of the 3 32 bit addresses presented by the EXE on the iAddress OMI input port. It is important to note that the iWriteEnable input port shall be asserted for at least 3 clock cycles otherwise the data triplet will not be effectively written into the OMEM.

As mentioned earlier, each VP is assigned to a single OMEM. The OMEM is write-only from the VP's perspective. Table 41 lists the relevant signals to communicate the VP with its corresponding OMEM unit. Since there is no risk of contention, the bus cycles to write into the OMEM do not follow the Wish-Bone protocol.

Table 41 presents the signals involved in the communication between the VP and the OMEM unit.

Table 41 – CORE signals for OMEM write bus cycles.

Signal name	Type	Size	Description
OMEM_WE_O	Output	1	Output memory Write Enable. The VP-n puts this signal in 1 to write into the write-only memory OMEM-n.
OMEM_ADR_O	Output	1	Output memory Write Address. The VP-n uses this signal to specify the write address into the write-only memory OMEM-n.
OMEM_DAT_O	Output	1	Output memory Write Data. The VP-n uses this signal to specify the data to write into the write-only memory VP-n.

The following figure illustrates some of the concepts from Table 41.

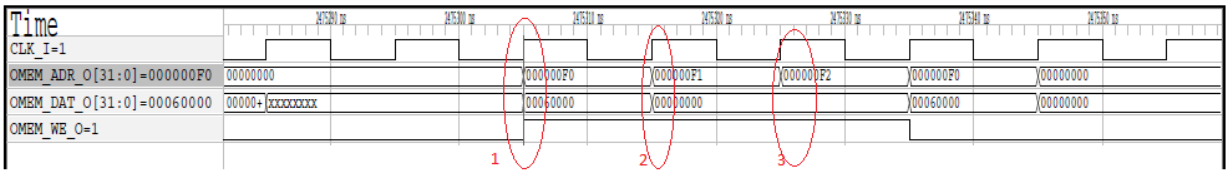

Figure 48 - VP writing data to an OMEM.

Marker 1 from Figure 48 shows when the VP is setting the OMEM_WE_O signal to 1. One clock cycle after the OMEM_WE signal is set to 1 by the VP, the data on OMEM_DAT_O is written into the memory address specified by OMEM_ADR_O.

6.2. Texture memory TMEM

The TMEM is an external memory from where the VP reads the texture information. The TMEM is read-only from the VP's perspective. All the VPs can access the TMEM through a cross bar interconnection in order to perform read operations.

Figure 49 show a conceptual representation of the cross-bar bus. Each cross point from Figure 49 is implemented as a simple switch. The TMEM is an interleaved RAM divided upon a number of memory banks, called TM0 … TM3 in Figure 49. Also each memory bank has its own simple bus arbiter (not shown in the picture). If two or more VPs want to read from the same memory bank at any given point in time, then a bus contention scenario occurs and the corresponding bus arbiter will handle the read requests in a round-robin fashion.

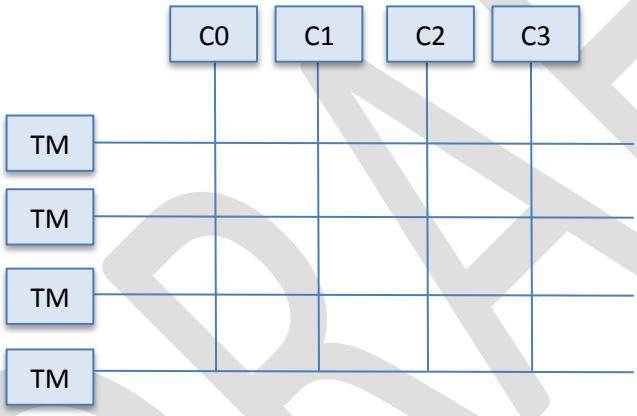

Figure 49 - Cross bar bus example

Table 42 presents the signals involved in the communication between the VP and the TMEM unit.

Table 42 – CORE signals for TMEM write bus cycles.

Signal name	Type	Size	Description
TMEM_DAT_I	Input	32	TMEM read data. Data read from TMEM.
TMEM_ADR_O	Output	32	TMEM read address. The VP specifies the address in TMEM from which to read.

TMEM_CYC_O	Output	1	Wishbone output cycle signal. The VP puts this signal in one in order to request ownership of the crossbar bus for a bus read cycle. The corresponding memory bank arbiter will grant the petition by asserting the GNT_I input signal.
TMEM_GNT_I	Input	1	Cross bar bank read access granted. The memory bank arbiter sets this signal to 1 when a bus read ownership petition is granted for this CORE instance.

The following figure illustrates some of the concepts from Table 42.

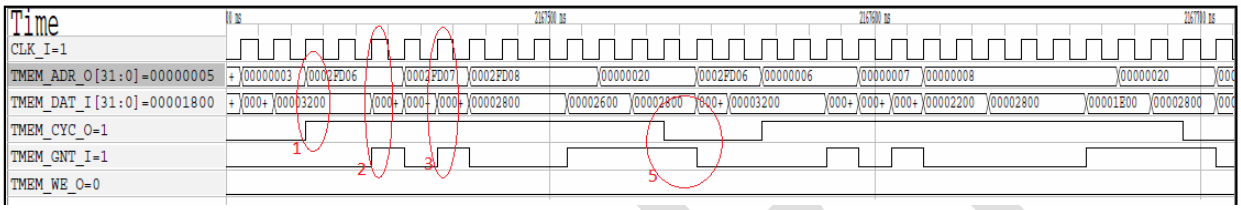

Figure 50 - CORE reading data from TMEM.

Figure 50 shows a read bus cycle where a VP is reading from the TMEM. Since the VPs and the TMEM are connected through a cross-bar bus, concurrent read access from different cores is guaranteed as long as no two VP are attempting to read from the same memory of TMEM at the same time. If two or more VPs are attempting to concurrently read from the same TMEM memory bank then the corresponding arbiter will grant ownership of the bus to each VP in a round-robin fashion.

The marker 1 from Figure 50 shows a VP setting the TMEM_CYC_O to 1. By setting the TMEM_CYC_O signal to one, the VP is requesting a read bus cycle from the address specified by TMEM_ADR_O. If no other VP is trying to read from that same memory bank then the bus arbiter immediately grants the bus ownership to the VP by asserting the TMEM_GNT_I signal to one, otherwise the VP has to wait until the bus ownership is granted by the bus arbiter.

Marker 2 from Figure 50 shows the arbiter setting the TMEM_GNT_I signal to 1. This means that the VP has been assigned exactly 1 clock cycle to read in the data from the TMEM_DAT_I signal. Note that 1 clock cycle after the data is read in by the VP, the TMEM_ADR_O signal changes values, since the TMEM_CYC_I signal is still high, the arbiter understands that this VP wants to perform another read bus cycle, the data corresponding to this new read bus cycle arrives when the VP is granted the bus in marker 3.

Marker 5 from Figure 50 shows the VP setting the TMEM_CYC_O signal back to cycle. This marks the end of the read bus cycle, and the bus arbiter assumes that no more read petitions will come from this VP.

7. VP Register specification

The register files (RF) hosts up to 64[36] 96bit general purpose registers. The VP also has a set of 8 special purpose registers (SPR) which hold special values. The following sections summarize the register specification.

7.1. General purpose registers (GPRs)

The general purpose registers are a set of 64 * 96 bit registers[37]. Each register has the structure described in section 3.3. The general purpose registers are readable and writable by the ALU.

Although the Hardware makes no distinction on the usage of each general purpose registers, the software compiler has special uses for some of the general purpose registers; this is summarized in Table 43.

Table 43 Special purpose registers.

Register	Size (bits)	Name	Description
R0.x	32	Zero Register	This is intended to have the value 0x0.[38]
R0.y	32	One Register	This is intended to store the value 0x1.
R0.z	32	Two Register.	This is intended to store the value 0x2.[31]
R1	96	Return Value	The software shall store the return value from a function here. [31]
R2.x	32	Return Address	The software shall store the return address in this register. [31]
R2.y	32		The scale used for fixed point arithmetic. [31]
R2.z	32	Multi thread Control	Control Register. See table <TBD> for details. 0: -> Multithread enabled. 8:1 -> Thread 1 Code Offset
R3.x*	32	OFFSET register	The OFFSET used for the direct addressing mode with displacement and the indirect addressing mode with displacement. [31]
R3.y*	32	Previous OFFSET	The previous value of the OFFSET. [31]
R3.z	32	Index Register SCALE	Index Register used by the software to dereference arrays.
R4 – R9	96	Function parameters	The software shall store up to the first 6 function input parameters in the registers R24 – R29. [38]

[36] This number might change, depending on the performance analysis.
[37] This is about 3 kilobytes, perhaps we can bring down this number to 128 register which is around 1.5kB.
[38] This is a software convention; there is no hardware which enforces this convention.

R10 – R63	96	General purpose	Used by the compiler to store program variables and arrays.

The Registers marked with an '*' in Table 43 are shadowed. See next sections for details on this.

7.1.1. Zero register – R0.

As mentioned earlier, the compiler assumes that the register R0 has the value (0, 1, 2). This is a software convention, but it is very important for the compiler. Consider the following example:

```
//Assign a value
R7 = R8;

//Do a simple increment
R1.y++;
```

```
//Assign a value
ADD R7 R8, R0.xxx

//Do a simple increment
ADD R1.y R1 R0.yyy
```

Figure 51 Using the R0 register

In the previous code, the compiler first uses the register R0 in order to copy the value from R8 into R7. Since the VP does not have a "COPY" opcode, the compilers achieves the copy operation simply by using an addition in the form R7 = R8 + 0. This is done by using the swizzled register R0.xxx which is assumed by the compiler to have the value (0, 0, 0).

After doing the operation R = R8, the code from Figure 51 does a unitary increment R1.y++. To do this increment, the compiler uses the swizzled register R0.yyy, which is assumed to have the value (1, 1, 1), so is effectively doing R1.y = R1.xyz + (1, 1, 1);

It is important to mention that R0 (and in fact all of the general purpose registers) are readable and writable by the user. This means that nothing prevents the user from changing the values of R0, and this is fine for programs written in the THEIA assembly language, but for the high level language, unpredictable behavior may happen when the values of R0 are changed.

7.1.2. Return address register – R2.x

The R2.x register is used by the compiler to store the return address before making function calls. When the called function returns, the value in R2.x is used as an indirect address to return to the caller function. This illustrated in the next code.

```
//main calls MyFunction

function main                    //    GenerateRay();

{                                //store return address

                                 24:    8001 2702 0 1c          //ADD R2.xyz I(1c) 0
  GenerateRay();
                                 //store current frame offset

}                                25:    1 203 a03 a00           //ADD R3._y_ R3.xxx R0.xxx

                                 //displace next frame offset by the number of auto variables in current frame

                                 26:    8001 403 0 2            //ADD R3.x__ I(2) R[DST]

                                 //call the function

                                 27:    201 @GenerateRay 0      //ADD <BRANCH.ALWAYS> @GenerateRay.___ R0.xyz R0.xyz
```

Figure 52 Using the R2 register

7.1.3. Offset registers – R3.x, R2.y

The Offset registers are used by the compiler to implement the function "stack frame". The function stack frame is used to allocate space for the *automatic*[39] variables. Since the VP has no direct access to external memory locations, the space for auto variables is simply allocated by providing an offset into the register file (RF). The SPR R3.x is used as a pointer to the first memory location of the current stack frame[40]. Each time a function gets called, the R3.x register is updated by adding the number of local auto variables in the current frame. Also the previous frame offset is stored in R2.y; this is used so that when the subroutine returns, the previous function frame is restored. The next figure illustrated these concepts.

[39] See T-Language specification document for more details.
[40] Each memory location is a word, this is 96 bits.

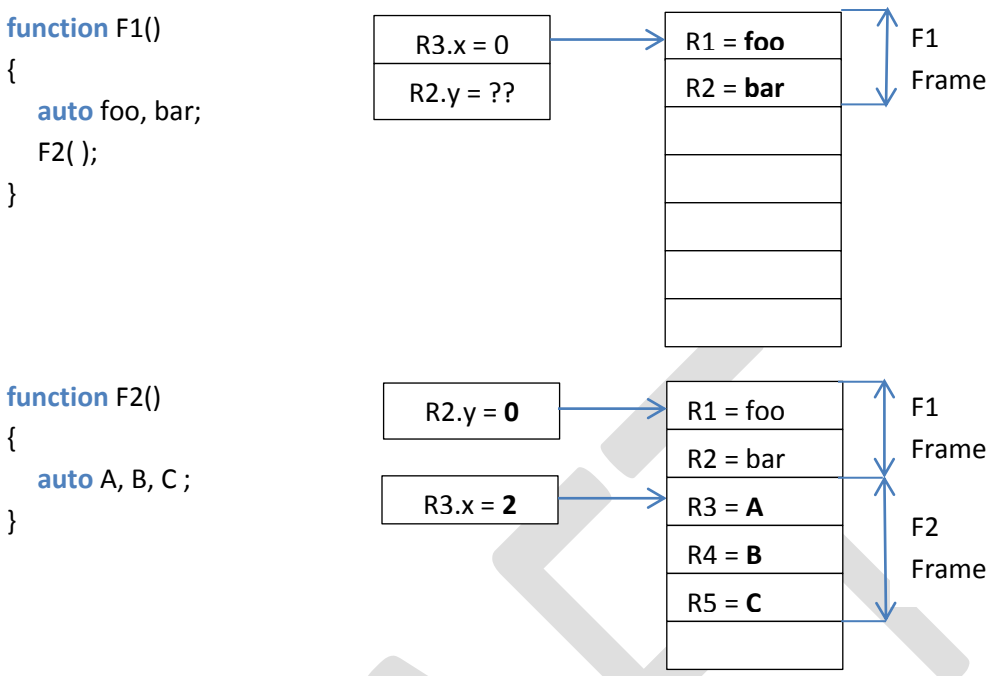

Figure 53 Example of using the offset register R30 to allocate memory for automatic variables.

7.2. Shadowed GPRs

As previously mentioned each VP has a set of general purpose registers (GPRs). Some of these GPRs have special meanings for the compiler. Also, some of the GPRs have special behaviors and under certain scenarios, certain VP blocks may have a need to read from a GPR "without having to access the RF directly".

Let's illustrate this with an example, let's suppose that the IIU is decoding an instruction that has direct addressing mode with displacement. Since the displacement is used the IIU would need to read the Offset register (R3.x) from the RF, but it would also needs to read the SRC0 and SRC1 values from the RF. The RF is a dual read channel RAM, meaning that the IIU can simultaneously read from 2 memory locations in the RF, but for this particular example it would need to read from 3 RF locations in the same clock cycle (which is not possible). In order to be able to read from certain special GPRs without using one of the two data address lines from the RF, a special "shadow register" topology is used for a small number of the GPRs such as R3.

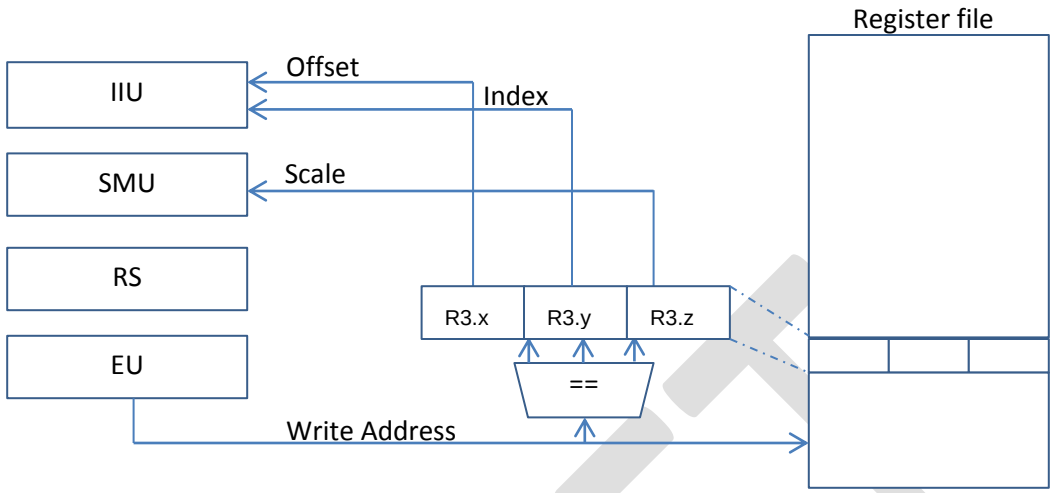

Figure 54 Example of an SPR shadowing R30

Figure 54 shows how R3 is present in the RF but it is also replicated outside of the RF, in a separate set of flops. When the EUs write into R3, the values are written to both the R3 location in the RF and also to the copy of R3 in the external flops. This allows the IIU to read the value of R3.x from the external flops instead using one of the two address lines to read from the RF, so that it can simultaneously read R3.x together with other two values from the RF during the same clock cycle.

7.3. Special purpose registers (SPRs)

These are special registers outside of the GPR space. **This section is TBD.**

Table 44 List of special purpose registers

Name	Position in RF	Size	Description
CONFIG	TBD	TBD	
ALUERR	TBD	TBD	The VP error registers. See table <>.
WDT	TBD	TBD	The watch dog timer. When the specified bit of the WDT is set, then an interrupt is generated.

The next table provides a description of the Control register.

Table 45 Control register (CNTREG)

Field	Range	Description

RESERVED	23:0	~~The scale used for the input operand scaling.~~[41] ~~See section 3.13.~~
EXCEN	24	Enable exception handling.
WDTEN	25	Watch dog timer enabled
WDTSEL	30:26	WDT select bit.

The next table provides a description of the Error register.

Table 46 Arithmetic error register

Field	Range	Description
XYZ	2:0	This field indicates if the current error is related to the x, y or z block. **000:** Unknown: the machine has no information regarding the x, y or blocks which generated the error. **001:** Current error generated by the operation z block. **010:** Last operation had division by zero on the Y block. **011:** Current error generated by the operation's Y block and the Z block. **100:** Current error generated by the operations X block. **101:** Current error generated by the operations X block and the Z block. **110:** Current error generated by the operations X block and the Y block. **111:** Current error generated by the operations X block, Y block and the Z block.
Division by zero[42]	3	Division by zero. The block specified by the XYZ field generated the error.
Arithmetic overflow	7:4	RSID of the RS causing the arithmetic overflow. The block specified by the XYZ field generated the error.

[41] Even if this scale gets changed, the SQRT always expects SCALE = 17. See section 3.10

[42] Fixed point arithmetic allows infinity divisions.

Scale overflow	11:8	RSID of the RS causing the scale overflow. The block specified by the XYZ field generated the error.
Scale underflow	12	A scale underflow occurred in the IIU.
Unknown square root	13	The value send into the SQRT unit was not found in the LUT. See section 3.10 for details.

8. Control Processor architecture (CP)

This section is WIP

The control processor (CP) is an in-order processor with a simple 3 stage pipeline.

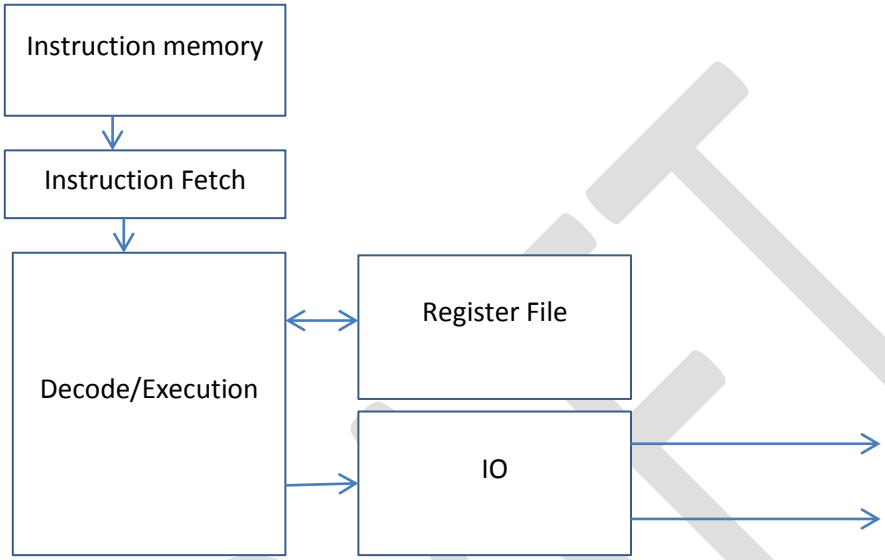

Figure 55 Control processor CP

8.1. CP Instruction set

Insert table with instruction set

Table 47 CP Instruction set

Name	Offset	Description
CP_SPR_STATUS	2	Bit 0: MCU pending operations: . Zero means that there are no operations pending in the MCU.
CP_SPR_BLOCK_DST	3	Bits 15:0 This register stores the destination ID that subsequent block copy operations will use. For example the next high level statement that issues a block copy command

8.2. CP Special purpose registers (SPRs)

Table 48 CP Special purpose registers

Name	Offset	Description
CP_SPR_STATUS	2	Bit 0: MCU pending operations: . Zero means that there are no operations pending in the MCU.
CP_SPR_BLOCK_DST	3	Bits 15:0 This register stores the destination ID that subsequent block copy operations will use. For example the next high level statement that issues a block copy command

CP_SPR_STATUS 2
Bit 0: MCU pending operations: . Zero means that there are no operations pending in the MCU. The CP can check for pending block copy operations using a code like the one in the following example.

```
//wait until queued block transfers are complete

    while ( block_transfer_in_progress ) {}
```

Figure 56 CP block transfer high level syntax

The reserved keyword “block_transfer_in_progress” returns 1 if CP_SPR_STATUS[0] is zero meaning there a no pending block copy operations, otherwise returns 0.

CP_SPR_BLOCK_DST 3 [15:0]

This register stores the destination ID that subsequent block copy operations will use.
For example the next high level statement that issues a block copy command

```
copy_data_block< CoredId, DstOffsetAndLen, SrcOffset>;
```

Figure 57 CP copy data block high level syntax

Translates into the next sequence of instructions:

```
//Setting destination ID SPR for Copy data block
14:     2030a00             //ADD R3 R10 R0
//Copy data block
15:     e000b0c             //COPYBLOCK DstId: R0 SrcOffset: R11
```

Figure 58

Notice how the R3 (CP_SPR_BLOCK_DST) is written and then COPYBLOCK command is issued.

8.3. CP Branching

The branch operation takes 1 extra clock cycle to decide the next instruction to fetch.
The compiler automatically inserts a NOP operation after each branch operation as shown in the following control processor code listing.

```
17:   d890001          //ASSIGN R137 I(1 )
18:   7150289          //BEQ R21 R2 R137
//branch delay
19:   110000           //NOP R0 R0 R0
//while loop goto re-eval boolean
20:   6110000          //BRANCH R17 R0 R0
//branch delay
21:   110000           //NOP R0 R0 R0
//  start <2>;
//Start
22:   1020000          //DELIVERCOMMAND R2 R0 R0
```

9. Internal Memory Controller (MCU) Architecture

10. Appendix A: VP Issue unit encoding table

Op	C3	C2	C1	C0	BUSY6	BUSY5	BUSY4	BUSY3	BUSY2	BUSY1	BUSY0	RS3	RS2	RS1	RS0
NOP	0	0	0	0	0	0	0	0	0	0	0				
NOP	0	0	0	0	0	0	0	0	0	0	1				
NOP	0	0	0	0	0	0	0	0	0	1	0				
NOP	0	0	0	0	0	0	0	0	0	1	1				
NOP	0	0	0	0	0	0	0	0	1	0	0				
NOP	0	0	0	0	0	0	0	0	1	0	1				
NOP	0	0	0	0	0	0	0	0	1	1	0				
NOP	0	0	0	0	0	0	0	0	1	1	1				
NOP	0	0	0	0	0	0	0	1	0	0	0				
NOP	0	0	0	0	0	0	0	1	0	0	1				
NOP	0	0	0	0	0	0	0	1	0	1	0				
NOP	0	0	0	0	0	0	0	1	0	1	1				
NOP	0	0	0	0	0	0	0	1	1	0	0				
NOP	0	0	0	0	0	0	0	1	1	0	1				
NOP	0	0	0	0	0	0	0	1	1	1	0				
NOP	0	0	0	0	0	0	0	1	1	1	1				
Operation	C3	C2	C1	C0	BUSY6	BUSY5	BUSY4	BUSY3	BUSY2	BUSY1	BUSY0	RS3	RS2	RS1	RS0
ADD	0	0	0	1	0	0	1	0	0	0	0				**1**
ADD	0	0	0	1	0	0	1	0	0	0	1			**1**	
ADD	0	0	0	1	0	0	1	0	0	1	0				**1**
ADD	0	0	0	1	0	0	1	0	0	1	1				
ADD	0	0	0	1	0	0	1	0	1	0	0				**1**
ADD	0	0	0	1	0	0	1	0	1	0	1			**1**	
ADD	0	0	0	1	0	0	1	0	1	1	0				**1**
ADD	0	0	0	1	0	0	1	0	1	1	1				
ADD	0	0	0	1	0	0	1	1	0	0	0				**1**
ADD	0	0	0	1	0	0	1	1	0	0	1			**1**	
ADD	0	0	0	1	0	0	1	1	0	1	0				**1**
ADD	0	0	0	1	0	0	1	1	0	1	1				
ADD	0	0	0	1	0	0	1	1	1	0	0				**1**
ADD	0	0	0	1	0	0	1	1	1	0	1			**1**	

ADD	0	0	0	1	0	0	1	1	1	1	0				**1**
ADD	0	0	0	1	0	0	1	1	1	1	1				
Operation	C3	C2	C1	C0	BUSY6	BUSY5	BUSY4	BUSY3	BUSY2	BUSY1	BUSY0	RS3	RS2	RS1	RS0
DIV	0	0	1	0	0	1	0	0	**0**	0	0			**1**	**1**
DIV	0	0	1	0	0	1	0	0	**0**	0	1			**1**	**1**
DIV	0	0	1	0	0	1	0	0	**0**	1	0			**1**	**1**
DIV	0	0	1	0	0	1	0	0	**0**	1	1			**1**	**1**
DIV	0	0	1	0	0	1	0	0	1	0	0				
DIV	0	0	1	0	0	1	0	0	1	0	1				
DIV	0	0	1	0	0	1	0	0	1	1	0				
DIV	0	0	1	0	0	1	0	0	1	1	1				
DIV	0	0	1	0	0	1	0	1	**0**	0	0			**1**	**1**
DIV	0	0	1	0	0	1	0	1	**0**	0	1			**1**	**1**
DIV	0	0	1	0	0	1	0	1	**0**	1	0			**1**	**1**
DIV	0	0	1	0	0	1	0	1	**0**	1	1			**1**	**1**
DIV	0	0	1	0	0	1	0	1	1	0	0				
DIV	0	0	1	0	0	1	0	1	1	0	1				
DIV	0	0	1	0	0	1	0	1	1	1	0				
DIV	0	0	1	0	0	1	0	1	1	1	1				
Operation	C3	C2	C1	C0	BUSY6	BUSY5	BUSY4	BUSY3	BUSY2	BUSY1	BUSY0	RS3	RS2	RS1	RS0
MUL	0	0	1	1	0	1	1	**0**	0	0	0		**1**		
MUL	0	0	1	1	0	1	1	**0**	0	0	1		**1**		
MUL	0	0	1	1	0	1	1	**0**	0	1	0		**1**		
MUL	0	0	1	1	0	1	1	**0**	0	1	1		**1**		
MUL	0	0	1	1	0	1	1	**0**	1	0	0		**1**		
MUL	0	0	1	1	0	1	1	**0**	1	0	1		**1**		
MUL	0	0	1	1	0	1	1	**0**	1	1	0		**1**		
MUL	0	0	1	1	0	1	1	**0**	1	1	1		**1**		
MUL	0	0	1	1	0	1	1	1	0	0	0				
MUL	0	0	1	1	0	1	1	1	0	0	1				

MUL	0	0	1	1	0	1	1	1	0	1	0				
MUL	0	0	1	1	0	1	1	1	0	1	1				
MUL	0	0	1	1	0	1	1	1	1	0	0				
MUL	0	0	1	1	0	1	1	1	1	0	1				
MUL	0	0	1	1	0	1	1	1	1	1	0				
MUL	0	0	1	1	0	1	1	1	1	1	1				
Operation	C3	C2	C1	C0	BUSY6	BUSY5	BUSY4	BUSY3	BUSY2	BUSY1	BUSY0	RS3	RS2	RS1	RS0
SQRT	0	1	0	0	1	0	**0**	0	0	0	0		**1**		**1**
SQRT	0	1	0	0	1	0	**0**	0	0	0	1		**1**		**1**
SQRT	0	1	0	0	1	0	**0**	0	0	1	0		**1**		**1**
SQRT	0	1	0	0	1	0	**0**	0	0	1	1		**1**		**1**
SQRT	0	1	0	0	1	0	**0**	0	1	0	0		**1**		**1**
SQRT	0	1	0	0	1	0	**0**	0	1	0	1		**1**		**1**
SQRT	0	1	0	0	1	0	**0**	0	1	1	0		**1**		**1**
SQRT	0	1	0	0	1	0	**0**	0	1	1	1		**1**		**1**
SQRT	0	1	0	0	1	0	**0**	1	0	0	0		**1**		**1**
SQRT	0	1	0	0	1	0	**0**	1	0	0	1		**1**		**1**
SQRT	0	1	0	0	1	0	**0**	1	0	1	0		**1**		**1**
SQRT	0	1	0	0	1	0	**0**	1	0	1	1		**1**		**1**
SQRT	0	1	0	0	1	0	**0**	1	1	0	0		**1**		**1**
SQRT	0	1	0	0	1	0	**0**	1	1	0	1		**1**		**1**
SQRT	0	1	0	0	1	0	**0**	1	1	1	0		**1**		**1**
SQRT	0	1	0	0	1	0	**0**	1	1	1	1		**1**		**1**
Operation	C3	C2	C1	C0	BUSY6	BUSY5	BUSY4	BUSY3	BUSY2	BUSY1	BUSY0	RS3	RS2	RS1	RS0
LOGIC	0	1	0	1	1	**0**	1	0	0	0	0		**1**	**1**	
LOGIC	0	1	0	1	1	**0**	1	0	0	0	1		**1**	**1**	
LOGIC	0	1	0	1	1	**0**	1	0	0	1	0		**1**	**1**	
LOGIC	0	1	0	1	1	**0**	1	0	0	1	1		**1**	**1**	
LOGIC	0	1	0	1	1	**0**	1	0	1	0	0		**1**	**1**	
LOGIC	0	1	0	1	1	**0**	1	0	1	0	1		**1**	**1**	

LOGIC	0	1	0	1	1	**0**	1	0	1	1	0		**1**	**1**	
LOGIC	0	1	0	1	1	**0**	1	0	1	1	1		**1**	**1**	
LOGIC	0	1	0	1	1	**0**	1	1	0	0	0		**1**	**1**	
LOGIC	0	1	0	1	1	**0**	1	1	0	0	1		**1**	**1**	
LOGIC	0	1	0	1	1	**0**	1	1	0	1	0		**1**	**1**	
LOGIC	0	1	0	1	1	**0**	1	1	0	1	1		**1**	**1**	
LOGIC	0	1	0	1	1	**0**	1	1	1	0	0		**1**	**1**	
LOGIC	0	1	0	1	1	**0**	1	1	1	0	1		**1**	**1**	
LOGIC	0	1	0	1	1	**0**	1	1	1	1	0		**1**	**1**	
LOGIC	0	1	0	1	1	**0**	1	1	1	1	1		**1**	**1**	
Operation	C3	C2	C1	C0	BUSY6	BUSY5	BUSY4	BUSY3	BUSY2	BUSY1	BUSY0	RS3	RS2	RS1	RS0
IO	0	1	1	1	1	1	0	0	0	0	0				
IO	0	1	1	1	1	1	0	0	0	0	1				
IO	0	1	1	1	1	1	0	0	0	1	0				
IO	0	1	1	1	1	1	0	0	0	1	1				
IO	0	1	1	1	1	1	0	0	1	0	0				
IO	0	1	1	1	1	1	0	0	1	0	1				
IO	0	1	1	1	1	1	0	0	1	1	0				
IO	0	1	1	1	1	1	0	0	1	1	1				
IO	0	1	1	1	1	1	0	1	0	0	0				
IO	0	1	1	1	1	1	0	1	0	0	1				
IO	0	1	1	1	1	1	0	1	0	1	0				
IO	0	1	1	1	1	1	0	1	0	1	1				
IO	0	1	1	1	1	1	0	1	1	0	0				
IO	0	1	1	1	1	1	0	1	1	0	1				
IO	0	1	1	1	1	1	0	1	1	1	0				
IO	0	1	1	1	1	1	0	1	1	1	1				
IO	0	1	1	1	1	1	1	0	0	0	0				
IO	0	1	1	1	1	1	1	0	0	0	1				

IO	0	1	1	1	1	1	1	0	0	1	0			
IO	0	1	1	1	1	1	1	0	0	1	1			
IO	0	1	1	1	1	1	1	0	1	0	0			
IO	0	1	1	1	1	1	1	0	1	0	1			
IO	0	1	1	1	1	1	1	0	1	1	0			
IO	0	1	1	1	1	1	1	0	1	1	1			
IO	0	1	1	1	1	1	1	1	0	0	0			
IO	0	1	1	1	1	1	1	1	0	0	1			
IO	0	1	1	1	1	1	1	1	0	1	0			
IO	0	1	1	1	1	1	1	1	0	1	1			
IO	0	1	1	1	1	1	1	1	1	0	0			
IO	0	1	1	1	1	1	1	1	1	0	1			
IO	0	1	1	1	1	1	1	1	1	1	0			
IO	0	1	1	1	1	1	1	1	1	1	1			
IO	0	1	1	1	**0**	0	0	0	0	0	0	**1**	**1**	**1**
IO	0	1	1	1	**0**	0	0	0	0	0	1	**1**	**1**	**1**
IO	0	1	1	1	**0**	0	0	0	0	1	0	**1**	**1**	**1**
IO	0	1	1	1	**0**	0	0	0	0	1	1	**1**	**1**	**1**
IO	0	1	1	1	**0**	0	0	0	1	0	0	**1**	**1**	**1**
IO	0	1	1	1	**0**	0	0	0	1	0	1	**1**	**1**	**1**
IO	0	1	1	1	**0**	0	0	0	1	1	0	**1**	**1**	**1**
IO	0	1	1	1	**0**	0	0	0	1	1	1	**1**	**1**	**1**
IO	0	1	1	1	**0**	0	0	1	0	0	0	**1**	**1**	**1**
IO	0	1	1	1	**0**	0	0	1	0	0	1	**1**	**1**	**1**
IO	0	1	1	1	**0**	0	0	1	0	1	0	**1**	**1**	**1**
IO	0	1	1	1	**0**	0	0	1	0	1	1	**1**	**1**	**1**
IO	0	1	1	1	**0**	0	0	1	1	0	0	**1**	**1**	**1**
IO	0	1	1	1	**0**	0	0	1	1	0	1	**1**	**1**	**1**
IO	0	1	1	1	**0**	0	0	1	1	1	0	**1**	**1**	**1**
IO	0	1	1	1	**0**	0	0	1	1	1	1	**1**	**1**	**1**

dsdsdsds

11. Appendix B: VP addressing mode examples

This section gives several examples and use cases of the VP addressing modes from Table 17. The examples are provided from a software/compiler perspective, so knowledge of the T-Language and GPU assembly language is assumed.

Direct (0 000): The Indexes from SCR1, SRC0 and DST are directly used to calculate the corresponding addresses in the RF.

```
DSTADDR   =   DSTINDEX
SRC1      =   R[ SRC1INDEX ]
SRC0      =   R[ SRC0INDEX ]
```

Example:

```
//Simple addition

R1 = R2 + R3;
```

Becomes:

ADD R1.xyz R2.xyz R3.xyz	DSTADDR	1
	SRC1	R[2]
	SRC0	R[3]

Direct with displacement (0 001): SRC0INDEX is added OFFSET and then used to calculate SRC0ADDR in RF.

```
DSTADDR   =   DSTINDEX
SRC1      =   R[ SCR1INDEX ]
SRC0      =   R[ SRC0INDEX + OFFSET ]
```

Example:

```
//Simple addition using offset for index0

function foo()

{

   vector LocalVec = (1,2,3);

   R1 = R2 + LocalVec;

}
```

Becomes:

ADD R1.xyz R2.xyz R[**8**+offset].xyz[43]	DSTADDR	1
	SRC1	R[2]
	SRC0	R[8+offset]

---

0 010 Direct with displacement**: SRC1INDEX is added OFFSET and then used to calculate SRC1ADDR in RF.**

DSTADDR = DSTINDEX
SRC1 = R[SCR1INDEX + OFFSET]
SRC0 = R[SRC0INDEX]

Example:

```
//Simple addition using offset for index0

function foo()

{

   vector LocalVec = (1,2,3);

   R1 = LocalVec + R2;
```

---

[43] 8 is the RF address where the local variables for the current function frame are allocated.

```
}
```

Becomes:

ADD R1.xyz R2.xyz R[**8**+offset].xyz[44]

DSTADDR	1
SRC1	R[8+offset]
SRC0	R[2]

0 011 **Direct with displacement**: SRC1INDEX is added OFFSET and then used to calculate SRC1ADDR in RF. SRC0INDEX is added OFFSET and then used to calculate SRC0ADDR in RF.

DSTADDR = DSTINDEX
SRC1 = R[SCR1INDEX + OFFSET]
SRC0 = R[SRC0INDEX + OFFSET]

Example:

```
//Simple addition using offset for index0

function foo()

{

    vector A = (1,2,3),B=(4,5,6);

    R1 = LocalVec + B;

}
```

Becomes:

[44] 8 is the RF address where the local variables for the current function frame are allocated.

ADD R1.xyz R[9+offset].xyz R[**8**+offset].xyz[45]	DSTADDR	1
	SRC1	R[8+offset]
	SRC0	R[9+offset]

0 100 **Direct with displacement**: DSTINDEX is added OFFSET and then used to calculate DSTADDR in RF.

DSTADDR = DSTINDEX + OFFSET
SRC1 = R[SCR1INDEX]
SRC0 = R[SRC0INDEX]
Example:

```
//Simple addition using offset for index0

function foo()

{

   vector   Result;

   Result = R1 + R2;

}
```

Becomes:

ADD R[8+offset].xyz R1.xyz R2.xyz[46]	DSTADDR	8+offset
	SRC1	R[1]
	SRC0	R[2]

0 101 **Direct with displacement**: DSTINDEX is added OFFSET and then used to calculate DSTADDR in RF. SRC0INDEX is added OFFSET and then used to calculate SRC0ADDR in RF.

[45] 8 is the RF address where the local variables for the current function frame are allocated.
[46] 8 is the RF address where the local variables for the current function frame are allocated.

DSTADDR = DSTINDEX + OFFSET
SRC1 = R[SCR1INDEX]
SRC0 = R[SRC0INDEX + OFFSET]
Example:

```
//Simple addition using offset for index0

function func()

{

    vector    Result, foo = (1,2,3);

    Result = R1 + foo;

}
```

Becomes:

ADD R[8+offset].xyz R1.xyz R[9+offset].xyz[47]	DSTADDR	8+offset
	SRC1	R[1]
	SRC0	9+offset

0 110 **Direct with displacement**: DSTINDEX is added OFFSET and then used to calculate DSTADDR in RF. SRC1INDEX is added OFFSET and then used to calculate SRC1ADDR in RF.

DSTADDR = DSTINDEX + OFFSET
SRC1 = R[SCR1INDEX + OFFSET]
SRC0 = R[SRC0INDEX]
Example:

```
//Simple addition using offset for index0

function func()

{

    vector    Result, foo = (1,2,3);
```

[47] 8 is the RF address where the local variables for the current function frame are allocated.

```
    Result = foo + R1;

}
```

Becomes:

ADD R[8+offset].xyz R[9+offset].xyz R1.xyz [48]	DSTADDR	8+offset
	SRC1	9+offset
	SRC0	R[1]

0 111 **Direct with displacement**: All the indexes from SRC1, SRC0 and DST are displaced by the OFFSET.

DSTADDR = DSTINDEX + OFFSET
SRC1 = R[SCR1INDEX + OFFSET]
SRC0 = R[SRC0INDEX + OFFSET]

Example:

```
//Simple addition using offset for index0

function func()

{

    vector    Result, foo = (1,2,3), bar = (4,5,6);

    Result = foo + bar;

}
```

Becomes:

ADD R[8+offset].xyz R[9+offset].xyz R[a+offset].xy [49]	DSTADDR	8+offset
	SRC1	R[9+offset]

[48] 8 is the RF address where the local variables for the current function frame are allocated.

SRC0	R[a+offset]

1 000 **Direct with IMMV**: The 32-bit immediate (literal) value IMMV is used as SRC1, the value of the register pointed by DSTINDEX is used as SRC0.

DSTADDR = DSTINDEX
SRC1.x = IMMV
SRC1.y = IMMV
SRC1.z = IMMV
SRC0 = R[DSTINDEX]

Example:

```
//Cummulative addition

R1 += 5;
```

Becomes:

ADD R1 IMM(5) R1	DSTADDR	1
	SRC1	(5,5,5)
	SRC0	R[1]

1 001 **Direct with IMMV and offset:**

DSTADDR = DSTINDEX
SRC1.x = IMMV
SRC1.y = IMMV
SRC1.z = IMMV
SRC0.x = 0
SRC0.y = 0
SRC0.z = 0

Example:

```
//Literal increment
```

[49] 8 is the RF address where the local variables for the current function frame are allocated.

```
vector foo;

foo += 0xcafe;
```

Becomes:

ADD R[8+offset] IMM(0xcafe) R[8+offset]	DSTADDR	8+offset
	SRC1	(0xcafe,0xcafe,0xcafe)
	SRC0	R[8+offset]

1 100 **Direct with IMMV and clear SRC0**: Similar to the previous case except that SRC0 always takes the value of zero instead of the value of R[DSTINDEX]

DSTADDR = DSTINDEX
SRC1.x = IMMV
SRC1.y = IMMV
SRC1.z = IMMV
SRC0.x = 0
SRC0.y = 0
SRC0.z = 0

Example:

```
//Literal Assignment

R1 = 0xcafe;
```

Becomes:

ADD R1.0 IMM(0xcafe) **0x0**	DSTADDR	1
	SRC1	(0xcafe,0xcafe,0xcafe)
	SRC0	(0,0,0)

Works Cited

[1] D. P. John Hennessy, Computer Architecture: A Quantitative Approach, Morgan Kaufmann, 5 edition (September 30, 2011).