SCRATCH
User Manual

August 25, 2017

Contents

1 Introduction **3**

2 System Requirements **3**
2.1 SCRATCH 3
2.2 FPGA execution 3

3 Obtaining MIAOW 2.0 Design **3**

4 Trimming MIAOW FPGA Core **4**
4.1 CodeXL compilation 4
4.2 Core trimming 8

5 Synthesizing the Core **10**
5.1 Base system 10
5.1.1 MicroBlaze system - Xilinx Evaluation Platform 10
5.1.2 MicroBlaze system - Other Xilinx Platform 10
5.2 Instantiating MIAOW 11
5.3 Adding a second clock domain 11
5.4 Pre-fetch memory 11

6 Running a kernel on the Core **14**
6.1 Xilinx SDK 14
6.2 MIAOW2.0 Core initialization flow 14

7 Validating the Compute Unit Design **15**
7.1 ISA Validation – Verilog Simulator 15
7.2 ISA Validation on the FPGA 18
7.3 Validating the Benchmarks on the FPGA 19
7.4 Debugging strategy 19

A OpenCL compute unit initialization **20**

B Example MIAOW programming flow **23**

1 Introduction

SCRATCH trims a MIAOW FPGA core to match the requirements of a specific application. This manual describes the full flow necessary to run OpenCL kernels in a trimmed MIAOW FPGA core.

2 System Requirements

2.1 SCRATCH

To run ***SCRATCH*** the user must ensure the following conditions:

- The computing platform is a *Linux* based system with python 2.7 and its standard library installed.
- CodeXL (or an alternative compiler) is installed and can compile kernels for the Southern Islands architecture. This manual is based on version 2.4.39.

2.2 FPGA execution

To execute a kernel in the FPGA the user must ensure the following:

- The computing platform is a *Linux* based system with Vivado installed. This manual is based on version 2015.1, but a more recent version can be used.
- Xilinx SDK is installed alongside Vivado.

3 Obtaining MIAOW 2.0 Design

Please download the Verilog sources of MIAOW 2.0 here:
`https://github.com/scratch-gpu/MIAOW2`

Alternately, if you wish to use the original MIAOW GPU core developed by the Vertical Research Group at the University of Wisconsin-Madison[1] do the following:

- Obtain the sources for MIAOW at:
  `https://github.com/VerticalResearchGroup/miaow`
- In a terminal, navigate to the folder *src/verilog/rtl*.
- Execute the command:
  $ *make fpga*
- All the necessary files should now be in a folder called *fpga_core*

[1]Some differences between the original MIAOW and MAIOW2 can be found, depending on the status of each project.

4 Trimming MIAOW FPGA Core

The ***SCRATCH*** tool allows to prune the design in order to attain an area and power efficient architecture. This is an optional step that can be skipped should the user prefer to work with the untrimmed original architecture.

4.1 CodeXL compilation

Before using the ***SCRATCH*** tool the OpenCL kernel must be compiled through CodeXL targeting the Southern Islands architecture.

- Open CodeXL and switch to analyze mode as shown in figure 1.

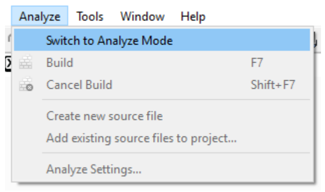


Figure 1: Switch to analyze mode.

- Under *File* select *New Project*, set the project name and leave everything else with the default settings as depicted in figures 2 and 3.

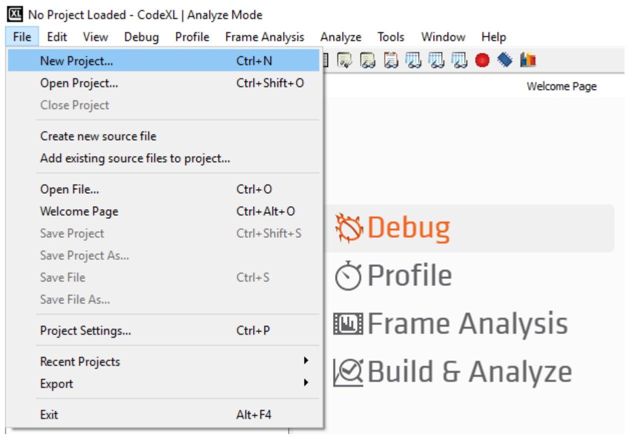
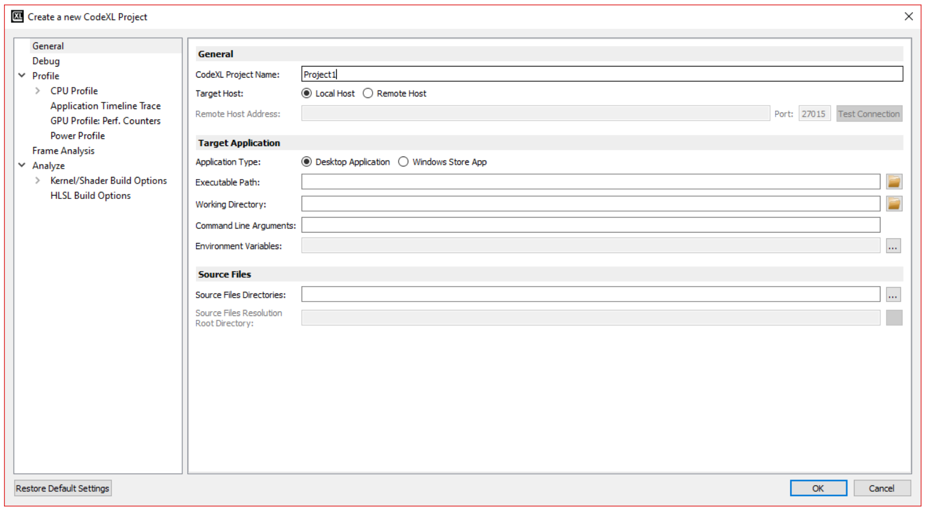


Figure 2: New project.


Create a new CodeXL Project

General

CodeXL Project Name: Project1

Target Host: [✓ Local Host] Remote Host

Remote Host Address: Port: 27015

Target Application

Application Type: [✓ Desktop Application] Windows Store App

Executable Path:

Working Directory:

Command Line Arguments:

Environment Variables:

Source Files

Source Files Directories:

Source Files Resolution Root Directory:

Figure 3: Set new project settings.

- Once a new project was created, the openCL source files may now be added to it. Do so by right-clicking the project name and select *Add existing source file*.

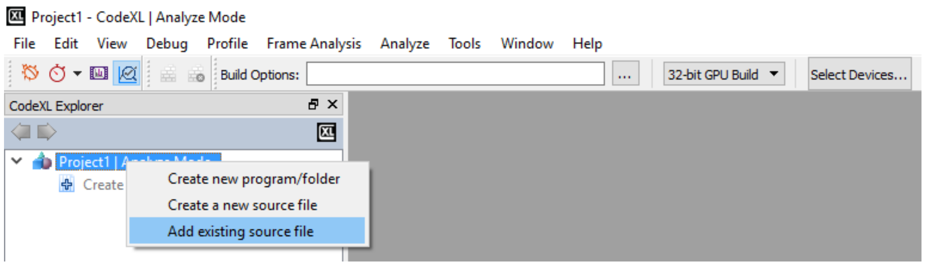


Figure 4: Adding source files.

- The devices to which the project will be built have to be specified. Follow the procedure as indicated in figures 5 and 6.

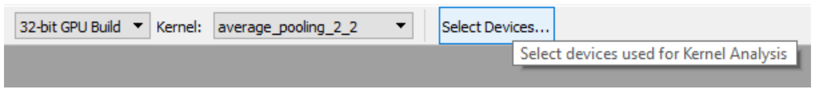
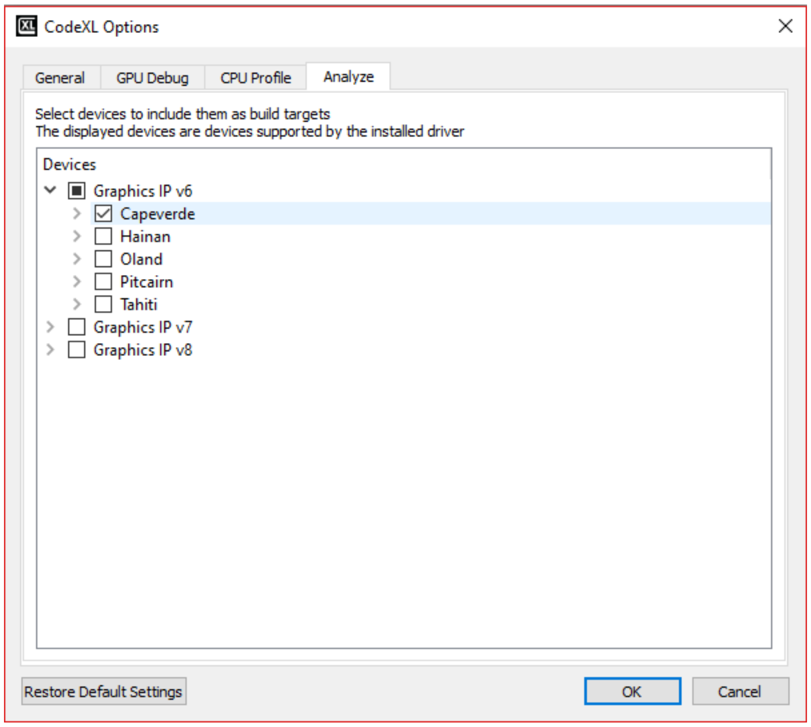


Figure 5: Select devices menu.


CodeXL Options

General | GPU Debug | CPU Profile | Analyze

Select devices to include them as build targets
The displayed devices are devices supported by the installed driver

Devices
- ▣ Graphics IP v6
  - ☑ Capeverde
  - ☐ Hainan
  - ☐ Oland
  - ☐ Pitcairn
  - ☐ Tahiti
- ☐ Graphics IP v7
- ☐ Graphics IP v8

Restore Default Settings | OK | Cancel

Figure 6: Available devices.

- The kernels can now be built. For that right-click over the source file or the project name and select *Build <name_of_source>*.

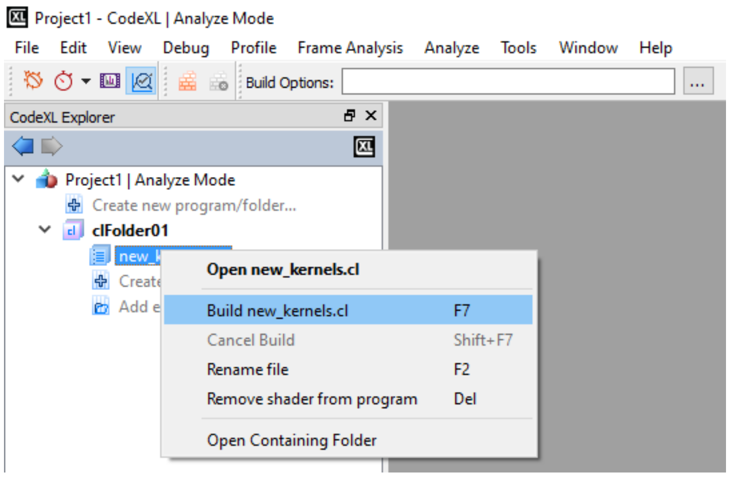
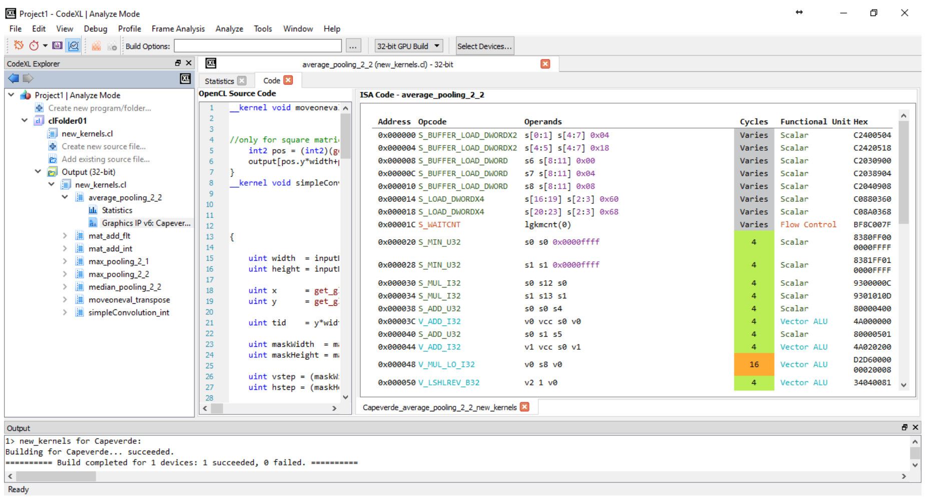


Figure 7: Building new kernels.

- When the compilation finishes, two outputs were produced: the ISA code and statistics.


Project1 - CodeXL | Analyze Mode
File Edit View Debug Profile Frame Analysis Analyze Tools Window Help
Build Options:
32-bit GPU Build
Select Devices...
CodeXL Explorer
Project1 | Analyze Mode
Create new program/folder...
clFolder01
new_kernels.cl
Create new source file...
Add existing source file...
Output (32-bit)
new_kernels.cl
average_pooling_2_2
Statistics
Graphics IP v6: Capever...
mat_add_flt
mat_add_int
max_pooling_2_1
max_pooling_2_2
median_pooling_2_2
moveoneval_transpose
simpleConvolution_int
average_pooling_2_2 (new_kernels.cl) - 32-bit
Statistics
Code
OpenCL Source Code
ISA Code - average_pooling_2_2
Capeverde_average_pooling_2_2_new_kernels
Output
Ready

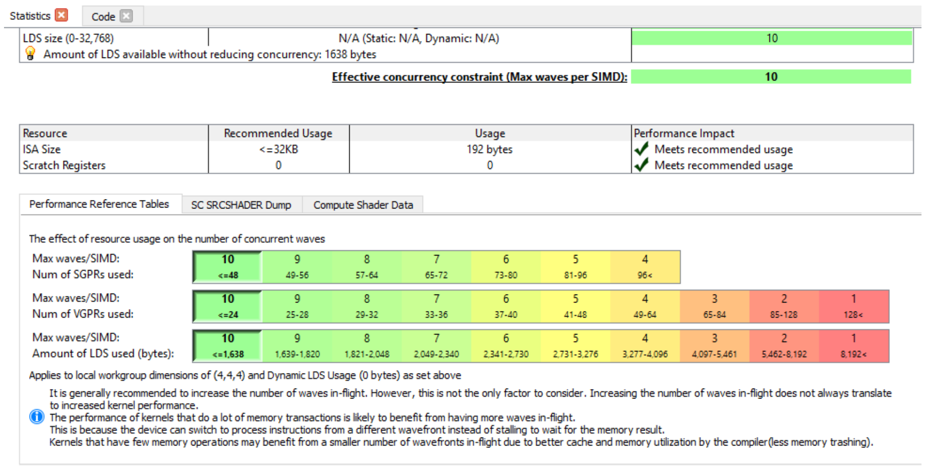
```
__kernel void moveoneva


//only for square matri
    int2 pos = (int2)(g
    output[pos.y*width+
}
__kernel void simpleCon




{

    uint width  = input
    uint height = input

    uint x      = get_g
    uint y      = get_g

    uint tid    = y*wid

    uint maskWidth  = m
    uint maskHeight = m

    uint vstep = (maskW
    uint hstep = (maskH
```

```
Address  Opcode                Operands                   Cycles  Functional Unit Hex
0x000000 S_BUFFER_LOAD_DWORDX2 s[0:1] s[4:7] 0x04         Varies  Scalar          C2400504
0x000004 S_BUFFER_LOAD_DWORDX2 s[4:5] s[4:7] 0x18         Varies  Scalar          C2420518
0x000008 S_BUFFER_LOAD_DWORD   s6 s[8:11] 0x00            Varies  Scalar          C2030900
0x00000C S_BUFFER_LOAD_DWORD   s7 s[8:11] 0x04            Varies  Scalar          C2038904
0x000010 S_BUFFER_LOAD_DWORD   s8 s[8:11] 0x08            Varies  Scalar          C2040908
0x000014 S_LOAD_DWORDX4        s[16:19] s[2:3] 0x60       Varies  Scalar          C0880360
0x000018 S_LOAD_DWORDX4        s[20:23] s[2:3] 0x68       Varies  Scalar          C08A0368
0x00001C S_WAITCNT             lgkmcnt(0)                 Varies  Flow Control    BF8C007F
                                                                                  8380FF00
0x000020 S_MIN_U32             s0 s0 0x0000ffff           4       Scalar          0000FFFF
                                                                                  8381FF01
0x000028 S_MIN_U32             s1 s1 0x0000ffff           4       Scalar          0000FFFF
0x000030 S_MUL_I32             s0 s12 s0                  4       Scalar          9300000C
0x000034 S_MUL_I32             s1 s13 s1                  4       Scalar          9301010D
0x000038 S_ADD_U32             s0 s0 s4                   4       Scalar          80000400
0x00003C V_ADD_I32             v0 vcc s0 v0               4       Vector ALU      4A000000
0x000040 S_ADD_U32             s0 s1 s5                   4       Scalar          80000501
0x000044 V_ADD_I32             v1 vcc s0 v1               4       Vector ALU      4A020200
                                                                                  D2D60000
0x000048 V_MUL_LO_I32          v0 s8 v0                   16      Vector ALU      00020008
0x000050 V_LSHLREV_B32         v2 1 v0                    4       Vector ALU      34040081
```

```
1> new_kernels for Capeverde:
Building for Capeverde... succeeded.
========== Build completed for 1 devices: 1 succeeded, 0 failed. ==========
```

Figure 8: ISA code produced by the compiler.

- Under the tab *Statistics*, the usage report can be found alongside the initialization parameters.



LDS size (0-32,768)	N/A (Static: N/A, Dynamic: N/A)	10
Amount of LDS available without reducing concurrency: 1638 bytes		
	Effective concurrency constraint (Max waves per SIMD):	10

Resource	Recommended Usage	Usage	Performance Impact
ISA Size	<=32KB	192 bytes	✓ Meets recommended usage
Scratch Registers	0	0	✓ Meets recommended usage

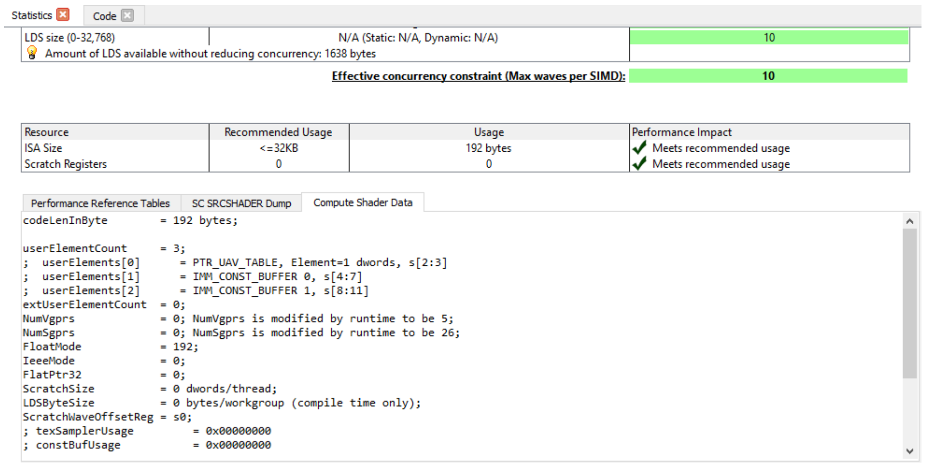
The effect of resource usage on the number of concurrent waves

Max waves/SIMD:	10	9	8	7	6	5	4			
Num of SGPRs used:	<=48	49-56	57-64	65-72	73-80	81-96	96<			
Max waves/SIMD:	10	9	8	7	6	5	4	3	2	1
Num of VGPRs used:	<=24	25-28	29-32	33-36	37-40	41-48	49-64	65-84	85-128	128<
Max waves/SIMD:	10	9	8	7	6	5	4	3	2	1
Amount of LDS used (bytes):	<=1,638	1,639-1,820	1,821-2,048	2,049-2,340	2,341-2,730	2,731-3,276	3,277-4,096	4,097-5,461	5,462-8,192	8,192<

Applies to local workgroup dimensions of (4,4,4) and Dynamic LDS Usage (0 bytes) as set above

It is generally recommended to increase the number of waves in-flight. However, this is not the only factor to consider. Increasing the number of waves in-flight does not always translate to increased kernel performance.
The performance of kernels that do a lot of memory transactions is likely to benefit from having more waves in-flight.
This is because the device can switch to process instructions from a different wavefront instead of stalling to wait for the memory result.
Kernels that have few memory operations may benefit from a smaller number of wavefronts in-flight due to better cache and memory utilization by the compiler(less memory trashing).

Figure 9: Usage report.



LDS size (0-32,768)	N/A (Static: N/A, Dynamic: N/A)	10
Amount of LDS available without reducing concurrency: 1638 bytes		

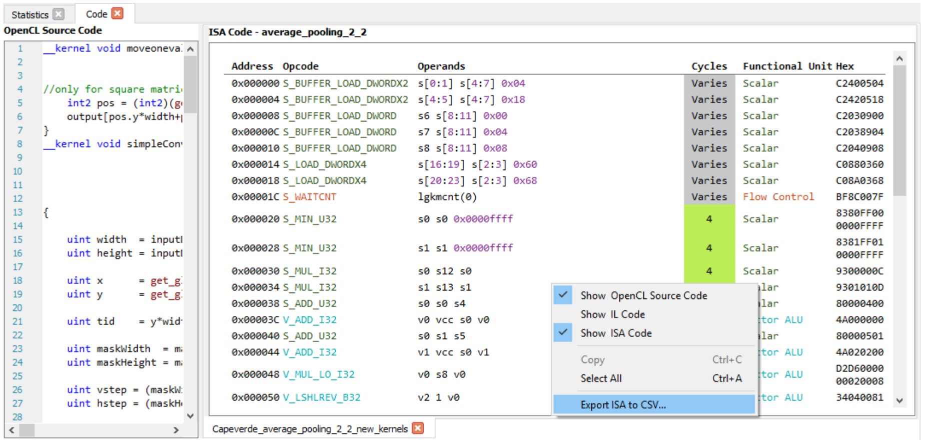
Effective concurrency constraint (Max waves per SIMD): 10

Resource	Recommended Usage	Usage	Performance Impact
ISA Size	<=32KB	192 bytes	✓ Meets recommended usage
Scratch Registers	0	0	✓ Meets recommended usage

```
codeLenInByte        = 192 bytes;

userElementCount     = 3;
;  userElements[0]      = PTR_UAV_TABLE, Element=1 dwords, s[2:3]
;  userElements[1]      = IMM_CONST_BUFFER 0, s[4:7]
;  userElements[2]      = IMM_CONST_BUFFER 1, s[8:11]
extUserElementCount  = 0;
NumVgprs             = 0; NumVgprs is modified by runtime to be 5;
NumSgprs             = 0; NumSgprs is modified by runtime to be 26;
FloatMode            = 192;
IeeeMode             = 0;
FlatPtr32            = 0;
ScratchSize          = 0 dwords/thread;
LDSByteSize          = 0 bytes/workgroup (compile time only);
ScratchWaveOffsetReg = s0;
; texSamplerUsage        = 0x00000000
; constBufUsage          = 0x00000000
```

Figure 10: Initialization parameters.

- Finally, the output product of the compilation that will be used by the ***SCRATCH*** is the Assembly code. It has to be exported as a `.csv` file to be fed to the tool. For exporting the Assembly code, right-click anywhere in the code and select *Export ISA to CSV*.



```
__kernel void moveoneva

//only for square matri
    int2 pos = (int2)(g
    output[pos.y*width+
}
__kernel void simpleCon




{

    uint width  = input
    uint height = input

    uint x      = get_g
    uint y      = get_g

    uint tid    = y*wid

    uint maskWidth  = m
    uint maskHeight = m

    uint vstep = (maskW
    uint hstep = (maskH
```

```
Address  Opcode                Operands                  Cycles  Functional Unit Hex
0x000000 S_BUFFER_LOAD_DWORDX2 s[0:1] s[4:7] 0x04        Varies  Scalar          C2400504
0x000004 S_BUFFER_LOAD_DWORDX2 s[4:5] s[4:7] 0x18        Varies  Scalar          C2420518
0x000008 S_BUFFER_LOAD_DWORD   s6 s[8:11] 0x00           Varies  Scalar          C2030900
0x00000C S_BUFFER_LOAD_DWORD   s7 s[8:11] 0x04           Varies  Scalar          C2038904
0x000010 S_BUFFER_LOAD_DWORD   s8 s[8:11] 0x08           Varies  Scalar          C2040908
0x000014 S_LOAD_DWORDX4        s[16:19] s[2:3] 0x60      Varies  Scalar          C0880360
0x000018 S_LOAD_DWORDX4        s[20:23] s[2:3] 0x68      Varies  Scalar          C08A0368
0x00001C S_WAITCNT             lgkmcnt(0)                Varies  Flow Control    BF8C007F
0x000020 S_MIN_U32             s0 s0 0x0000ffff          4       Scalar          8380FF00
                                                                                 0000FFFF
0x000028 S_MIN_U32             s1 s1 0x0000ffff          4       Scalar          8381FF01
                                                                                 0000FFFF
0x000030 S_MUL_I32             s0 s12 s0                 4       Scalar          9300000C
0x000034 S_MUL_I32             s1 s13 s1                          lar            9301010D
0x000038 S_ADD_U32             s0 s0 s4                           lar            80000400
0x00003C V_ADD_I32             v0 vcc s0 v0                       tor ALU        4A000000
0x000040 S_ADD_U32             s0 s1 s5                           lar            80000501
0x000044 V_ADD_I32             v1 vcc s0 v1                       tor ALU        4A020200
0x000048 V_MUL_LO_I32          v0 s8 v0                           tor ALU        D2D60000
                                                                                 00020008
0x000050 V_LSHLREV_B32         v2 1 v0                            tor ALU        34040081
```

Figure 11: Exporting ISA code to `.csv` file to be fed to the tool.

4.2 Core trimming

Upon obtaining the CSV file generated from CodeXL containing the assembly and binary instruction information, this file must be passed to the tool, alongside the MIAOW FPGA core generated above. The tool will, in its turn, output newly generated verilog files to replace the original ones.

- Obtain the tool from:
  `https://github.com/scratch-gpu/Trimming-Tool`
- In a terminal navigate to the folder containing the tool's main python executable (*scratch.py*).
- Run the tool providing two arguments:

  1. MIAOW Folder – - -*MIAOW_folder* or -*m* – This specifies the folder containing MIAOW's FPGA core.
     By default this folder is set to <script_dir>/fpga_core.
     The folder path provided must be relative to the script directory.
  2. Instruction CSV – - -*instruction_csv* or -*f* – This specifies the file outputted by CodeXL for the desired application.

  The command will then look as follows:
  $*./scratch.py -m fpga_core -f mmult_int.csv*
- Upon running the tool the generated verilog files (**.v*) will be created in the script's directory. These files can then be used to overwrite the original ones in the "fpga_core" folder.
- After overwriting the original files with the generated ones the core can be instantiated in a Vivado project and synthesized as described below.

5 Synthesizing the Core

To create a usable system in the FPGA a few elements other that the actual MIAOW2.0 FPGA core need to be instantiated. The following process will assist you in this, although it assumes that the user is fairly familiar with creating a MicroBlaze based design with a MIG module.
If you have a Xilinx Evaluation Platform use the following steps, otherwise skip to the next sub-section.

5.1 Base system

5.1.1 MicroBlaze system - Xilinx Evaluation Platform

- Open Vivado and create a new project.
- After setting the name and project location hit Next.
- In *Project Type* select *Example Project* and hit Next.
- Select *Base MicroBlaze* and hit Next.
- Select your evaluation platform and continue.
- Click finish.
- In the newly created system open the block diagram design.
- Add a MIG controller to your design.
- Run block automation to configure the MIG controller.
- Run connection automation to connect MIG to the remainder of the system.

5.1.2 MicroBlaze system - Other Xilinx Platform

- Open Vivado and create a new project.
- In *Project Type* select *RTL Project.*
- Continue without adding sources or existing IP or constraints.
- Select your part and finish the project creation.
- Under *IP Integrator* click *Create Block Design.*
- In the newly created design add a MicroBlaze, a MicroBlaze debug module, a MIG controller and, if desired, some form of IO (GPIO, UART, etc).
- If your system does not feature a UART interface please enable the *Enable JTAG UART* option in MicroBlaze debug module.
- Run block automation to configure your blocks.
- Run connection automation to connect all blocks together.

5.2 Instantiating MIAOW

Start by obtaining the sources of MIAOW2.0 from `https://github.com/scratch-gpu/MIAOW2` and adding the core sources by clicking "Add Sources". Then:

- Select "Add or create design sources" and press "Next".
- Select "Add Directories..." and navigate to the fpga_core folder.
- Continue until you exit the "Add Sources" dialog box.

To connect MIAOW2.0 to the MicroBlaze system created above the user needs to create a special interconnect block developed by the original MIAOW team. Details on how to create this peripheral and use it to connect MIAOW's FPGA core to the MicroBlaze system can be found in `https://github.com/VerticalResearchGroup/miaow/wiki/Neko` under the subtitle *AXI Peripheral*.

Once the procedures above are concluded the user should have a MIAOW system ready to be synthesized.

5.3 Adding a second clock domain

Change MIG *ui_clock* frequency By default the frequency of the *ui_clock* outputted by MIG will be 100MHz. This value must be changed to 200MHz.

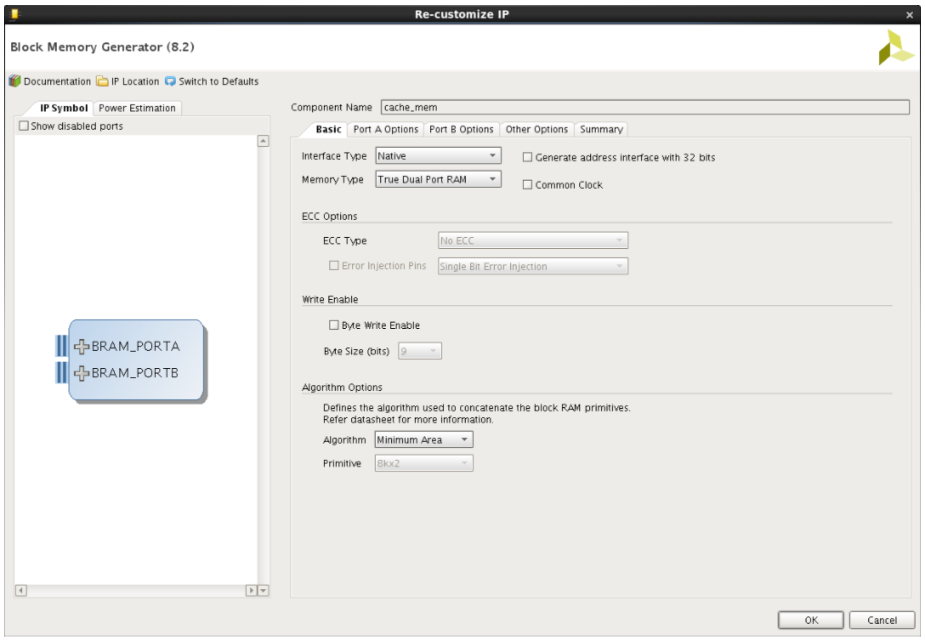
- Double-click on the MIG block.
- Once the MIG configuration window opens click *Next* until the *Controller Options* page is reached.
- Change the *PHY to Controller Clock Ratio* from *4:1* to *2:1*.
- Proceed until you can can regenerate the MIG controller.
- Instantiate a new clock wizard that should receive a custom clock as input.
- Configure the new clock wizard have a 200MHz input frequency and a 50MHz output frequency.
- Use the new clock wizard to feed MIAOW's interconnect peripheral and also create an output port from this signal (to feed MIAOW's FPGA Core).
- Use the MIG's *ui_clock* to drive all other blocks.

5.4 Pre-fetch memory

The pre-fetch memory module is a small memory block instantiated within the *fpga_memory* module.

- Under *Project Manager* open the *IP Catalog*.

- Search for *Block Memory Generator*.
- Set the *Basic* options according to Figure 12:



Block Memory Generator (8.2)

Show disabled ports

Component Name: cache_mem

Interface Type: Native
Memory Type: True Dual Port RAM
☐ Generate address interface with 32 bits
☐ Common Clock

ECC Options
ECC Type: No ECC
☐ Error Injection Pins: Single Bit Error Injection

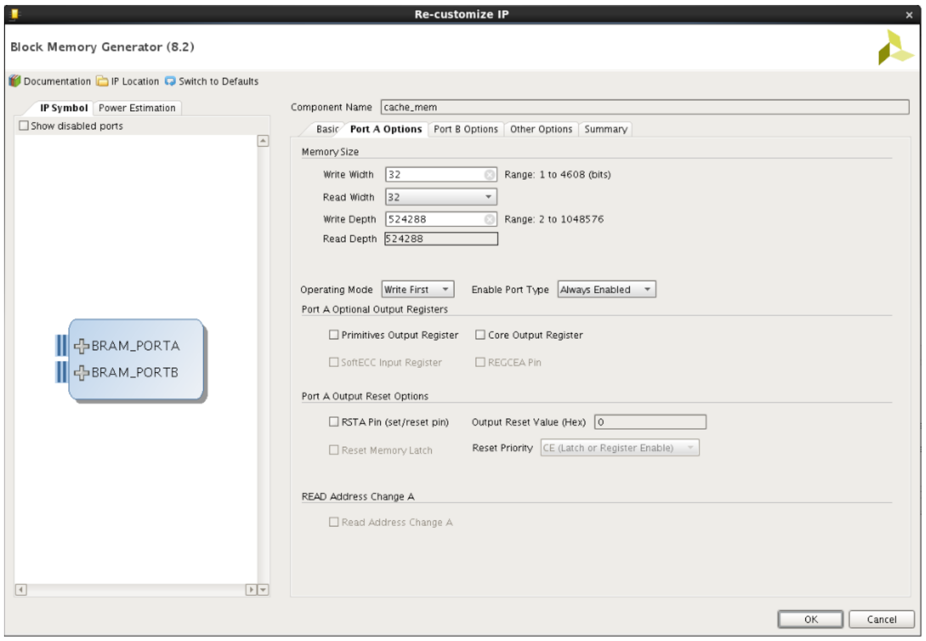
Write Enable
☐ Byte Write Enable
Byte Size (bits): 9

Algorithm Options
Defines the algorithm used to concatenate the block RAM primitives.
Refer datasheet for more information.
Algorithm: Minimum Area
Primitive: 8kx2

Figure 12: Block Memory Generator Basic

.

- Set the *Port A Options* according to Figure 13:



Component Name: cache_mem

Memory Size
Write Width: 32 — Range: 1 to 4608 (bits)
Read Width: 32
Write Depth: 524288 — Range: 2 to 1048576
Read Depth: 524288

Operating Mode: Write First
Enable Port Type: Always Enabled

Port A Optional Output Registers
☐ Primitives Output Register
☐ Core Output Register
☐ SoftECC Input Register
☐ REGCEA Pin

Port A Output Reset Options
☐ RSTA Pin (set/reset pin)
Output Reset Value (Hex): 0
☐ Reset Memory Latch
Reset Priority: CE (Latch or Register Enable)

READ Address Change A
☐ Read Address Change A

Figure 13: Block Memory Generator Port A Options

.

- Set the *Port B Options* to match the ones of Port A.

  In this case the memory module will replace the first 524288 32-bit words in memory. Please feel free to implement other buffer schemes as desired.

- Once the changes above are made the user can synthesize the new system and generate the bitstream by pressing *Generate Bitstream* under *Program and Debug*.

6 Running a kernel on the Core

6.1 Xilinx SDK

After synthesis please start Xilinx's SDK with the following steps:

- In Vivado go to *File*, *Export Export Hardware...*
- Select *Include bitstream* and click *OK*.
- Launch the SDK:
  1. In Vivado go to *File*.
  2. Press *Launch SDK* and then press *OK*.
- Create a new project by following this tutorial: `https://www.xilinx.com/video/soc/hello-world-in-5-minutes.html`
- Note that you should choose the MicroBlaze system and not the ZC702 as presented in the video.
- With the HelloWorld application you should be able to verify the correctness of your system by accessing peripherals, like MIAOW's interconnect, and the memory.
- Once you have the MicroBlaze system running you can start using the MIAOW Core 2.0.

6.2 MIAOW2.0 Core initialization flow

To run a program in the MIAOW2.0 Core a few preparation steps have to be performed. To start execution the following flow must be performed:

- Initialize the memory area where the core will fetch data (as specified in the kernel inputs).
- Send a reset pulse.
- Populate the scalar registers with initial data.
- Populate the vector registers with initial data.
- Load the instruction buffer (program memory)
- Send the *Start Execution* signal.

Appendix A gives a more detailed explanation on which registers should be initialized prior to execution to match AMD's OpenCL expected behavior.

Once execution has started MicroBlaze is responsible for performing memory accesses on behalf of the compute unit. This requires MicroBlaze to constantly read the *MEM_OP* signal while the end of execution is not reached. Appendix B presents a real example of this flow.

7 Validating the Compute Unit Design

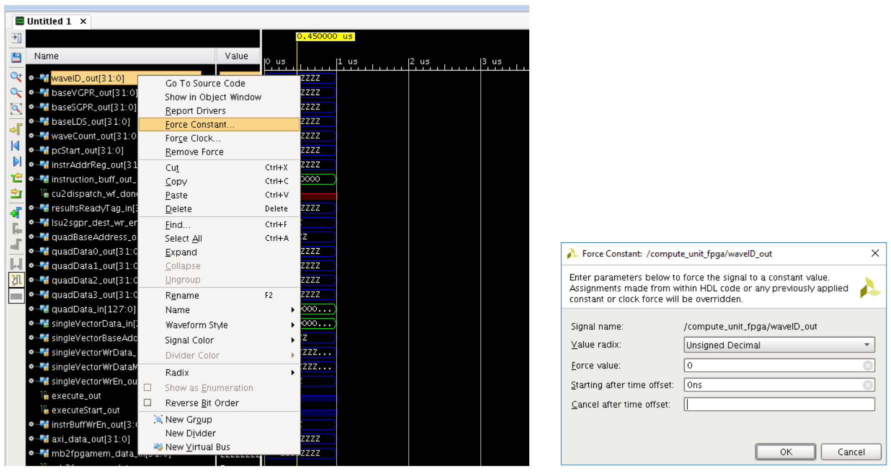
Naturally, as an open-source project, the user is allowed to make any modification to the RTL in order to improve performance, energy, or to adapt to some particular application needs. However, once a modification to the RTL is made, the user should carefully inspect the design in order to guarantee that modified units behave in the expected way. In this sense, it is highly suggested to start by validating (in simulation mode) the affected modules (e.g., decode, issue, scoreboard, scheduling, functional units) individually to assure correct functioning. Once this step is validated, a set of scripts and examples are provided to allow for the validation of the complete MIAOW2.0 design, by relying on the following strategy:

1. ISA validation at the Verilog Simulator (see sub-section 7.1 for details on how to simulate the MIAOW2.0 CU architecture), which includes:
   - Low-level debugging of the architecture on a signal basis (for instance testing a small localized change in a single unit).
   - Instruction level debugging (testing the effect of the unit's change in the overall architecture).
2. Single instruction testing on the FPGA. This consists in testing the correctness of all supported instructions by running a micro-kernel that sets the execution mask, running the instruction and finishing execution. The scripts provided in the `validation_scripts` MIAOW2.0 folder allow calculating the "golden" reference value using the MicroBlaze and then comparing the two values to check for instruction correctness. Sub-section 7.2 provides further details on using such scripts.
3. Kernel/benchmark level testing on the FPGA. Once we have the overview of all supported instructions the user can proceed to test an entire kernel/benchmark. This tests that the interaction between instructions, allowing to asses whether a correct behavior is attained. In this case, the output data (stored by the CU in memory) should compared against a golden reference (again generated by the MicroBlaze). Sections 7.3 and 7.4 provide additional information on how to attain this goal.

7.1 ISA Validation – Verilog Simulator

A procedure is herein described to allow the user to validate the correct execution of kernels at a simulation level (although only the steps for performing behavioral simulations are described, they can be easily replicated for post-synthesis or post-implementation cases). For this, a simulation using Xilinx Vivado Simulator (alternative verilog simulators could also be used) is first made:

- Open the Vivado project. If you haven't created one already, follow the steps described in section 5.



Force Constant: /compute_unit_fpga/waveID_out

Enter parameters below to force the signal to a constant value. Assignments made from within HDL code or any previously applied constant or clock force will be overridden.

Signal name: /compute_unit_fpga/waveID_out
Value radix: Unsigned Decimal
Force value: 0
Starting after time offset: 0ns
Cancel after time offset:

Figure 14: Forcing signal `waveID_out` to constant value 0.

- In simulation sources, confirm that the compute_unit_fpga component is the top module. If it is not, set it as top module.
- Start the simulation by selecting "Run behavioral simulation" under "Run simulation".

Once the simulation starts the user can exercise a certain unit (or the entire compute unit) by forcing its input signals to a given value in a cycle and observing the produced output signals. A user can force a signal either via a TCL command[2] or by using the Vivado Simulator GUI.

In the GUI the user should select the waveform window, right click the desired signal and press "Force constant...". The value of the signal can then be specified in the pop-up window – see also Figure 14.

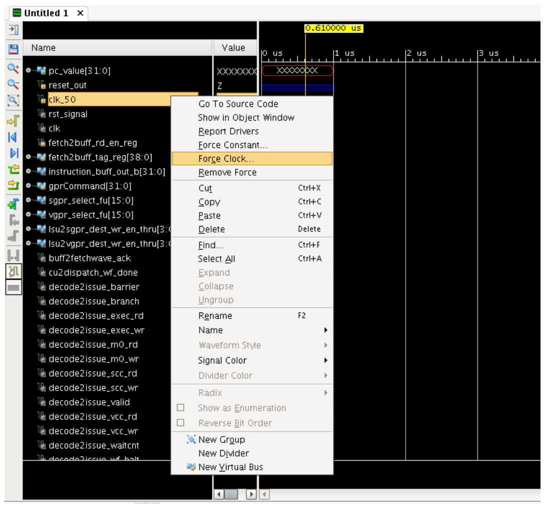
After setting all the desired signals the user can advance the simulation in time by using the run commands provided by Vivado. The clock signal can also be set by doing "Force clock" instead of "Force constant..." (see Figure 15).

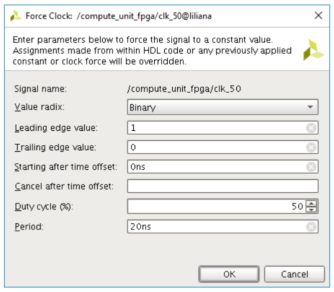
Alternatively a validation script can be made using TCL language, by following the Vivado Design Suite Tcl Command Reference Guide.

Example of TCL scripts that able to load and run micro-kernels in the CU can be found in the `validation_scripts` MIAOW2.0 folder. In this script the user can see the initial reset of signals, the setting of some initial data in the registers (scalar/vector), the initialization of instructions[3] in the instruction buffer and the initiation procedure to start execution in the CU. The user can

[2]The comprehensive list of TCL commands can be found at the Vivado Design Suite Tcl Command Reference Guide, available online: `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/ug835-vivado-tcl-commands.pdf`

[3]The definition of the AMD Southern Islands ISA can be found at `https://developer.amd.com/wordpress/media/2012/12/AMD_Southern_Islands_Instruction_Set_Architecture.pdf`




Force Clock: /compute_unit_fpga/clk_50@liliana

Enter parameters below to force the signal to a constant value. Assignments made from within HDL code or any previously applied constant or clock force will be overridden.

Signal name:	/compute_unit_fpga/clk_50
Value radix:	Binary
Leading edge value:	1
Trailing edge value:	0
Starting after time offset:	0ns
Cancel after time offset:	
Duty cycle (%):	50
Period:	20ns

Figure 15: Forcing signal `clk_50` to a clock with a 50 MHz operating frequency.

then advance the simulation time to see how the instruction is decoded, issued and executed across the entire CU.

In the example TCL scripts, the `S_ENDPGM` instruction (end of program) is used to mark the end of execution, which can be found in simulation by identifying when the signal `cu2dispatch_wf_done_in` is set to logic value '1' (see figure 16). To validate that the micro-kernel was correctly executed the user should verify the value of the scalar and logic registers, or memory.

To verify the values in the CU's internal registers the use must stop the execution (by setting the `execute_out` and `execute_start_out` signals are set to logic value '0'. Once this is done the address of the register must be provided.

Vector register To read the value of a given vector register the user must specify the desired vector (0, 1, 2, ...) by forcing the `singleVectorBaseAddress_out` signal to the corresponding value (0 for VGPR0, 1 for VGPR1, ...). Once the value is set the user must run the simulation for 2 clock cycles (40ns) to see the correct value being displayed in `singleVectorData_in`.

Scalar register To read the value of a given scalar register the user must specify the the desired register in the `quadBaseAddress_out` signal. In the scalar register unit four registers are displayed at once and therefore the user is setting the base register and will also be able to immediately see the next 3 registers. Once the value is set the user must run the simulation for 2 clock cycles (40ns) to see the correct value being displayed in `quadData_in`.

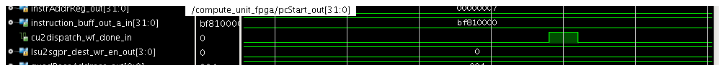

instrAddrReg_out[31:0]
/compute_unit_fpga/pcStart_out[31:0]
00000007
instruction_buff_out_a_in[31:0]
bf81000
bf810000
cu2dispatch_wf_done_in
0
lsu2sgpr_dest_wr_en_out[3:0]
0
0

Figure 16: The activation of signal `cu2dispatch_wf_done_in` marks the ends of execution.

7.2 ISA Validation on the FPGA

After verifying via simulation that the changes made to the RTL are correct the user should synthesize the design and implement it, targeting its board. The first type of validation should be made on an individual instruction basis. To do so, our team developed automated scripts in C that run on the MicroBlaze. These scripts issue an instruction in the CU and also the equivalent operation in the MicroBlaze, in order to validate execution correctness. At the end, each script prints a listing of supported instructions that can be used to identify which applications can be run on the CU.

To the run the aforementioned scripts the user must create a bitstream from the RTL design and create the MicroBlaze program that will control execution on the board. To do so, after generating the bitstream, start Xilinx's SDK by using the following procedure in the Vivado window:

- In the File menu select Export > Export Hardware.
- In the pop-up window select "Include bitstream" and press "OK".
- In the File menu select Launch SDK and press "OK" in the pop-up window.

Once the SDK launches you can create a new project by doing the following:

- Program the board with the bitstream by selecting "Program FPGA" under the "Xilinx Tools" menu and clicking "Program" in the pop-up window.
- Right-click in the hardware platform in the Project Explorer and select "New" > "Project".
- In the pop-up window expand "Xilinx" and select "Application Project".
- Choose a project name, make sure that the processor is set to microblaze_0 and that the "Create new" option is selected in "Board Support Package".
- Select "Hello World" and press "Finish".
- Once the project creation finishes you can try and run the "Hello World" application by right-clicking the project in "Project Explorer" and selecting "Debug As" > "Launch on Hardware (System Debugger)".

The base Hello World program can then be replaced with one of the C scripts found in the `validation_scripts` MIAOW2.0 folder. This will exercise a set of instructions (scalar, vector or memory) and print the supported ones. The user can then compare the supported instructions with what was expected and therefore check if the changes made did not break other instructions unintendedly.

7.3 Validating the Benchmarks on the FPGA

For final validation of a modified MIAOW2.0 design, a set of benchmarks is provided. These benchmarks, available in the `example_apps` MIAOW2.0 folder, are based on standard OpenCL benchmarks (e.g., Rodinia and AMD SDK Benchmark Suite). In particular, since in the MIAOW2.0 design, the MicroBlaze soft processor is responsible for controlling execution of OpenCL applications, working both as host processor and also acting as the Ultra-threaded Dispatcher, it is responsible for implementing all non-accelerated application code and also for controlling the execution of the OpenCL kernels (including by initializing all CU registers when launching a new workgroup[4]). Hence, the original OpenCL code was carefully modified in order to run on the MIAOW 2.0 design.

If you desire to obtain your own benchmarks please follow the steps in subsection 4.1 that explain how to obtain the hexadecimal encoding of the instructions of the application. The hexadecimal instruction can then be placed in the "inst_mem" array that can be found in any of the example applications. The Microblaze code must match the code that would run in the OpenCL host by performing the buffer allocation and the initialization described in Appendix A.

7.4 Debugging strategy

Given the three stage approach for validation of the RTL and applications (see the beginning of this section), the debugging methodology should depend on where the error was encountered. In particular, if an error is encountered in the initial stage (simulation of the unit) the user should revisit the written RTL code for bugs, identifying and correcting any bugs that creates the undesired behavior.

If the error is encountered while running an instruction that exercises the unit then the user needs to think on how the changes to the RTL reflect in the overall functioning of the unit (do they match the registers read/write behaviors of other instructions in the unit, etc.).

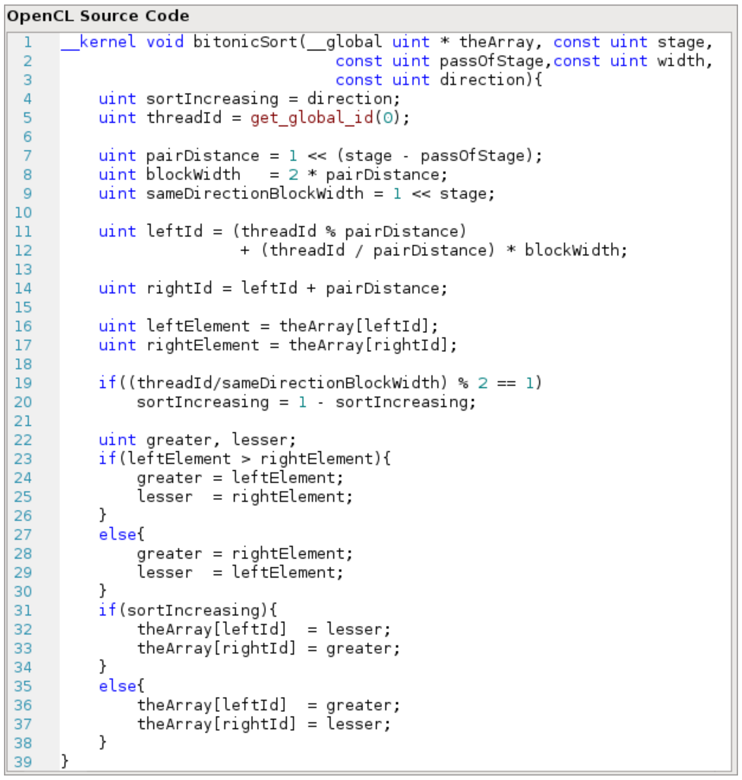
Finally, if the error is encountered at the application-level then the user needs to think on whether any resource that is now used in the unit might be conflicting with other instructions in the same unit or in different units.

[4]See Appendix A for information on how to achieve initialize the CU when launching a new workgroup.

A OpenCL compute unit initialization

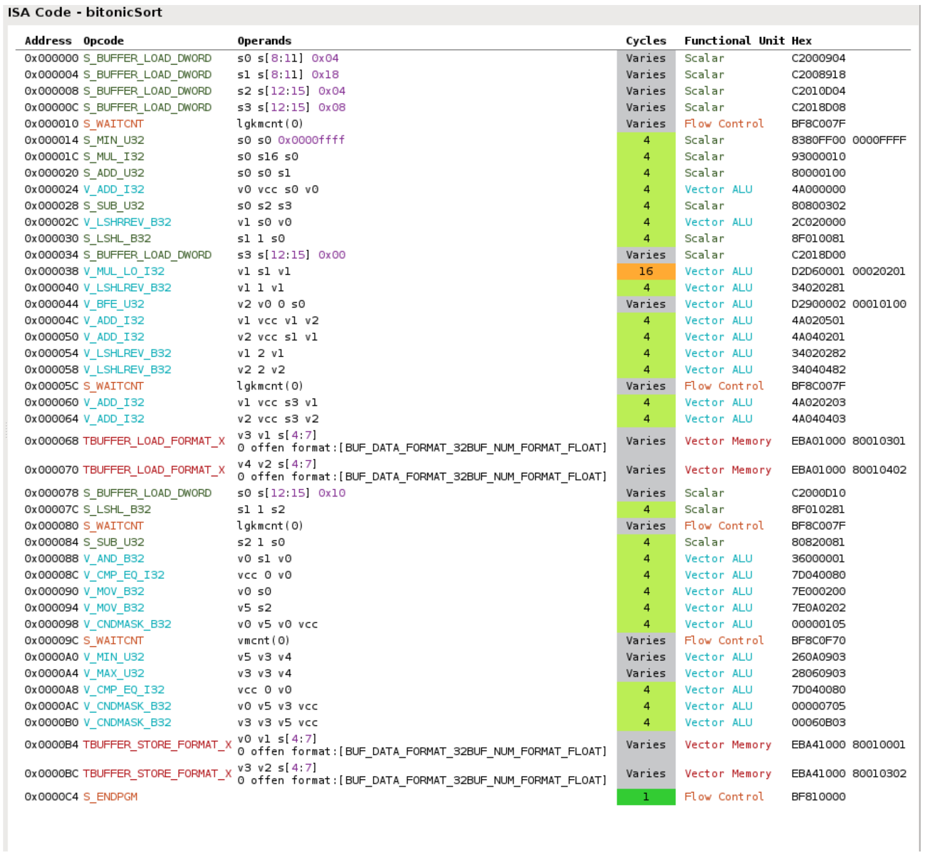
Prior to execution, the compute unit needs to be initialized with state data registers. This initialization is performed by the MicroBlaze, acting as an ultra-threaded dispatcher. Figure 18 clearly illustrates the use of pre-initialized registers. For instance, the first instruction loads a value on to scalar register zero (s0), using the value that was preset in scalar registers eight to eleven (s[8:11]) as the base address. Each application may set the register state by using specific system calls. This causes the number of pre-initialized register to vary on a per-application basis. Upon performing the compilation of a given kernel, CodeXL provides detailed information over the initial register state (see Figure 19).

On the selected applications, there are four major sets of scalar registers used, and three major vector registers used.



```
__kernel void bitonicSort(__global uint * theArray, const uint stage,
                          const uint passOfStage,const uint width,
                          const uint direction){
    uint sortIncreasing = direction;
    uint threadId = get_global_id(0);

    uint pairDistance = 1 << (stage - passOfStage);
    uint blockWidth   = 2 * pairDistance;
    uint sameDirectionBlockWidth = 1 << stage;

    uint leftId = (threadId % pairDistance)
                   + (threadId / pairDistance) * blockWidth;

    uint rightId = leftId + pairDistance;

    uint leftElement = theArray[leftId];
    uint rightElement = theArray[rightId];

    if((threadId/sameDirectionBlockWidth) % 2 == 1)
        sortIncreasing = 1 - sortIncreasing;

    uint greater, lesser;
    if(leftElement > rightElement){
        greater = leftElement;
        lesser  = rightElement;
    }
    else{
        greater = rightElement;
        lesser  = leftElement;
    }
    if(sortIncreasing){
        theArray[leftId]  = lesser;
        theArray[rightId] = greater;
    }
    else{
        theArray[leftId]  = greater;
        theArray[rightId] = lesser;
    }
}
```

Figure 17: OpenCL Code Example. This code is used as part of the bitonic sort benchmark, from the suite Multi2Sim.



ISA Code - bitonicSort

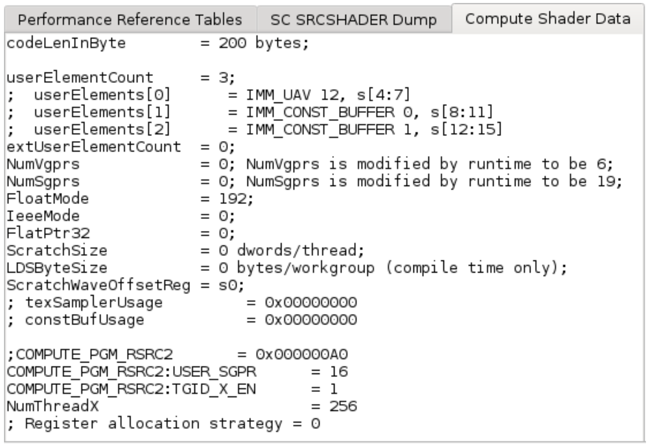
Address	Opcode	Operands	Cycles	Functional Unit	Hex
0x000000	S_BUFFER_LOAD_DWORD	s0 s[8:11] 0x04	Varies	Scalar	C2000904
0x000004	S_BUFFER_LOAD_DWORD	s1 s[8:11] 0x18	Varies	Scalar	C2008918
0x000008	S_BUFFER_LOAD_DWORD	s2 s[12:15] 0x04	Varies	Scalar	C2010D04
0x00000C	S_BUFFER_LOAD_DWORD	s3 s[12:15] 0x08	Varies	Scalar	C2018D08
0x000010	S_WAITCNT	lgkmcnt(0)	Varies	Flow Control	BF8C007F
0x000014	S_MIN_U32	s0 s0 0x0000ffff	4	Scalar	8380FF00 0000FFFF
0x00001C	S_MUL_I32	s0 s16 s0	4	Scalar	93000010
0x000020	S_ADD_U32	s0 s0 s1	4	Scalar	80000100
0x000024	V_ADD_I32	v0 vcc s0 v0	4	Vector ALU	4A000000
0x000028	S_SUB_U32	s0 s2 s3	4	Scalar	80800302
0x00002C	V_LSHRREV_B32	v1 s0 v0	4	Vector ALU	2C020000
0x000030	S_LSHL_B32	s1 1 s0	4	Scalar	8F010081
0x000034	S_BUFFER_LOAD_DWORD	s3 s[12:15] 0x00	Varies	Scalar	C2018D00
0x000038	V_MUL_LO_I32	v1 s1 v1	16	Vector ALU	D2D60001 00020201
0x000040	V_LSHLREV_B32	v1 1 v1	4	Vector ALU	34020281
0x000044	V_BFE_U32	v2 v0 0 s0	Varies	Vector ALU	D2900002 00010100
0x00004C	V_ADD_I32	v1 vcc v1 v2	4	Vector ALU	4A020501
0x000050	V_ADD_I32	v2 vcc s1 v1	4	Vector ALU	4A040201
0x000054	V_LSHLREV_B32	v1 2 v1	4	Vector ALU	34020282
0x000058	V_LSHLREV_B32	v2 2 v2	4	Vector ALU	34040482
0x00005C	S_WAITCNT	lgkmcnt(0)	Varies	Flow Control	BF8C007F
0x000060	V_ADD_I32	v1 vcc s3 v1	4	Vector ALU	4A020203
0x000064	V_ADD_I32	v2 vcc s3 v2	4	Vector ALU	4A040403
0x000068	TBUFFER_LOAD_FORMAT_X	v3 v1 s[4:7] 0 offen format:[BUF_DATA_FORMAT_32BUF_NUM_FORMAT_FLOAT]	Varies	Vector Memory	EBA01000 80010301
0x000070	TBUFFER_LOAD_FORMAT_X	v4 v2 s[4:7] 0 offen format:[BUF_DATA_FORMAT_32BUF_NUM_FORMAT_FLOAT]	Varies	Vector Memory	EBA01000 80010402
0x000078	S_BUFFER_LOAD_DWORD	s0 s[12:15] 0x10	Varies	Scalar	C2000D10
0x00007C	S_LSHL_B32	s1 1 s2	4	Scalar	8F010281
0x000080	S_WAITCNT	lgkmcnt(0)	Varies	Flow Control	BF8C007F
0x000084	S_SUB_U32	s2 1 s0	4	Scalar	80820081
0x000088	V_AND_B32	v0 s1 v0	4	Vector ALU	36000001
0x00008C	V_CMP_EQ_I32	vcc 0 v0	4	Vector ALU	7D040080
0x000090	V_MOV_B32	v0 s0	4	Vector ALU	7E000200
0x000094	V_MOV_B32	v5 s2	4	Vector ALU	7E0A0202
0x000098	V_CNDMASK_B32	v0 v5 v0 vcc	4	Vector ALU	00000105
0x00009C	S_WAITCNT	vmcnt(0)	Varies	Flow Control	BF8C0F70
0x0000A0	V_MIN_U32	v5 v3 v4	Varies	Vector ALU	260A0903
0x0000A4	V_MAX_U32	v3 v3 v4	Varies	Vector ALU	28060903
0x0000A8	V_CMP_EQ_I32	vcc 0 v0	4	Vector ALU	7D040080
0x0000AC	V_CNDMASK_B32	v0 v5 v3 vcc	4	Vector ALU	00000705
0x0000B0	V_CNDMASK_B32	v3 v3 v5 vcc	4	Vector ALU	00060B03
0x0000B4	TBUFFER_STORE_FORMAT_X	v0 v1 s[4:7] 0 offen format:[BUF_DATA_FORMAT_32BUF_NUM_FORMAT_FLOAT]	Varies	Vector Memory	EBA41000 80010001
0x0000BC	TBUFFER_STORE_FORMAT_X	v3 v2 s[4:7] 0 offen format:[BUF_DATA_FORMAT_32BUF_NUM_FORMAT_FLOAT]	Varies	Vector Memory	EBA41000 80010302
0x0000C4	S_ENDPGM		1	Flow Control	BF810000

Figure 18: AMD CodeXL Assembly result of the compilation of the bitonic sort OpenCL code from Multi2Sim's benchmark, depicted in Figure 17. The figure shows the ISA instruction opcode and operands, as well as the resulting hexadecimal code.

The first three sets of scalar registers contain memory descriptors. These are defined in AMD's Southern Islands ISA as a combination between a 48 bit address and state data for the memory access. The first set is identified, in Figure 19, as IMM_UAV, and is present in scalar registers four to seven. This memory descriptor contains an offset for data gathering accesses. The second set of registers, IMM_CONST_BUFFER 0 (s[8:11]), contains the base address of OpenCL call values. For instance, if a thread inquires its global ID this memory area is accessed, using a specific offset, to retrieve the required value.

The third set of registers, IMM_CONST_BUFFER 1 (s[12:15]), holds a pointer to the space in memory where the kernel arguments are kept.

The fourth—and final—set of scalar registers contains the thread group ID across the three possible dimensions (X, Y, and Z). In Figure 19, the flag *TGID_X_EN* is enabled ('1'), meaning that the register following the third set is initialized with the workgroup ID for dimension X. Since the flags for dimensions Y and Z are not set, only the first register of this set is initialized.



```
codeLenInByte        = 200 bytes;

userElementCount     = 3;
;  userElements[0]      = IMM_UAV 12, s[4:7]
;  userElements[1]      = IMM_CONST_BUFFER 0, s[8:11]
;  userElements[2]      = IMM_CONST_BUFFER 1, s[12:15]
extUserElementCount  = 0;
NumVgprs             = 0; NumVgprs is modified by runtime to be 6;
NumSgprs             = 0; NumSgprs is modified by runtime to be 19;
FloatMode            = 192;
IeeeMode             = 0;
FlatPtr32            = 0;
ScratchSize          = 0 dwords/thread;
LDSByteSize          = 0 bytes/workgroup (compile time only);
ScratchWaveOffsetReg = s0;
; texSamplerUsage         = 0x00000000
; constBufUsage           = 0x00000000

;COMPUTE_PGM_RSRC2       = 0x000000A0
COMPUTE_PGM_RSRC2:USER_SGPR      = 16
COMPUTE_PGM_RSRC2:TGID_X_EN      = 1
NumThreadX                       = 256
; Register allocation strategy = 0
```

Figure 19: AMD CodeXL Pre-Initialized Registers. CodeXL details the pre-initialized registers and their usage. The three user elements present consist of three memory descriptors that have to be initialized. Furthermore, a few usage statistics are given, like the number of used VGPR. To the end, there is indication that the first sixteen registers are initialized. Afterwards, there is a list of enabled flags, in this case only TGID_X_EN is set, which means that register number 16 has the workgroup ID in dimension X.

The vector registers are pre-initialized to contain the thread IDs on the different dimensions (X, Y, or Z). The dimensions depend on the type of application. A program whose data consists of one dimensional arrays only operates on the X dimension. If working on a two, or three, dimensional matrix then the second, or third, dimensions—Y and Z, respectively—, are also operated upon. The first vector register (v0) contains the thread IDs on dimension X and should always be defined. The subsequent registers are only initialized if more than one dimension is used.

B Example MIAOW programming flow

```
int32_t run_vop_program_neko(uint32_t insts[], int32_t num_insts,
    int32_t inst_scalar_data[], int32_t num_scalar_data,
    int32_t inst_vect_data[], int32_t num_vect_data, int32_t
        max_clocks) {
  /*
   * Execution Flow:
   * Resets Neko
   * Populates NEKO's instruction buffer, the scalar registers and
       the vector
registers (all 64 words of a register are initialized with the
    same value)
   * Send "start execution" command and waits for program
       completion or until
the timeout is reached
   * If the program reaches the end of execution before the
       timeout the data in
the registers is read and success(1) is returned
   * Otherwise returns 0 (unsuccessful)
   */
  int32_t index, address, data;
  int32_t vgpr, vgpr_word;
  int32_t * vgpr_data_pointer = (int32_t*) VGPR_DATA;

  //NEKO's reset pulse
  XIo_Out32(NEKO_RESET, 0);
  XIo_Out32(NEKO_RESET, 1);
  XIo_Out32(NEKO_RESET, 0);

  XIo_Out32(NEKO_BASE_LDS, XPAR_MIG_7SERIES_0_BASEADDR);

  //Load scalar registers with data
  for (index = 0; index < num_scalar_data; index += 4) {
    XIo_Out32(NEKO_SGRP_ADDR, index);
    XIo_Out32(NEKO_SGRP_QUAD_0, inst_scalar_data[index]);
    XIo_Out32(NEKO_SGRP_QUAD_1, inst_scalar_data[index + 1]);
    XIo_Out32(NEKO_SGRP_QUAD_2, inst_scalar_data[index + 2]);
    XIo_Out32(NEKO_SGRP_QUAD_3, inst_scalar_data[index + 3]);
    XIo_Out32(NEKO_GPR_CMD, 1);
  }
  //Load vector registers with data (replicating the data for
      every word of the
  //register)
  for (vgpr = 0; vgpr < 64; vgpr++) {
```

```
  XIo_Out32(VGPR_ADDR, vgpr);
  XIo_Out32(VGPR_WR_CLEAN, 1);
  XIo_Out32(VGPR_WR_CMD, 1);
}

for (vgpr = 0; vgpr < num_vect_data; vgpr++) {
  XIo_Out32(VGPR_ADDR, vgpr);
  for (vgpr_word = 0; vgpr_word < 64; vgpr_word++) {
    vgpr_data_pointer[vgpr_word] =
      inst_vect_data[vgpr * 64 + vgpr_word];
  }
  XIo_Out32(VGPR_WR_CMD, 1);

}

//Load the instruction buffer
for (index = 0; index < num_insts; index++) {
  XIo_Out32(NEKO_INSTR_ADDR, index);
  XIo_Out32(NEKO_INSTR_VALUE, insts[index]);
}

//Start execution
XIo_Out32(NEKO_CMD_ADDR, 1);

//Wait for the end of execution
while (XIo_In32(NEKO_CMD_ADDR) != 1) {

  //Check for memory accesses
  data = XIo_In32(NEKO_MEM_OP);
  if (data != 0) {

    int nextValue = MEM_RD_RDY_WAIT;

    if (data == MEM_RD_ACK_WAIT) {
      nextValue = MEM_RD_RDY_WAIT;
    } else if (data == MEM_WR_ACK_WAIT) {
      nextValue = MEM_WR_RDY_WAIT;

    } else if (data == MEM_WR_LSU_WAIT || data ==
        MEM_RD_LSU_WAIT)
      continue;//last instruction is not finished yet

    XIo_Out32(NEKO_MEM_ACK, 0);
    XIo_Out32(NEKO_MEM_ACK, 1);
    XIo_Out32(NEKO_MEM_ACK, 0);
```

```
      do {
        data = XIo_In32(NEKO_MEM_OP);
      } while (data != nextValue);

      address = XIo_In32(NEKO_MEM_ADDR);
      if (nextValue == MEM_RD_RDY_WAIT) {
        data = XIo_In32(address);
        XIo_Out32(NEKO_MEM_WR_DATA, data);
        nextValue = MEM_RD_LSU_WAIT;
      } else {
        data = XIo_In32(NEKO_MEM_RD_DATA);
        XIo_Out32(address, data);
        nextValue = MEM_WR_LSU_WAIT;
      }

      XIo_Out32(NEKO_MEM_DONE, 0);
      XIo_Out32(NEKO_MEM_DONE, 1);
      XIo_Out32(NEKO_MEM_DONE, 0);
      do {
        data = XIo_In32(NEKO_MEM_OP);
      } while (data != 0 && data != nextValue && data !=
          MEM_RD_ACK_WAIT
          && data != MEM_WR_ACK_WAIT);
    }
  }
  int32_t num_cycles = XIo_In32(NEKO_CYCLE_COUNTER);
  //NEKO's reset pulse
  XIo_Out32(NEKO_RESET, 0);
  XIo_Out32(NEKO_RESET, 1);
  XIo_Out32(NEKO_RESET, 0);

  return (num_cycles);
}
```