Speedster7t Reference Designs User Guide (UG085)

Speedster FPGAs

Achronix®
Data Acceleration

Copyrights, Trademarks and Disclaimers

Achronix Semiconductor Corporation

2903 Bunker Hill Lane
Santa Clara, CA 95054
USA

Website: www.achronix.com
E-mail : info@achronix.com

Table of Contents

Chapter - 1: Introduction

The following designs are a collection of reference designs that demonstrate some of the advanced features of the Speedster7t FPGA family. They are primarily focused on the machine learning processor (MLP) block which is tightly coupled with a block memory (BRAM). This block allows for up to 32 simultaneous multiplications in a single cycle, as well as supporting a wide collection of number formats from floating point to 16 bit integer.

The designs have all been run through Synplify-Pro and ACE, and where applicable expected performance and resource utilization values are given. In addition, all of the designs include a simulation environment to allow the user to verify the correct operation.

The designs are provided "as-is", with no warranty with regard to fitness of purpose, or direct applicability to a users solution. Users are encouraged to test the designs and then to modify to suit their particular solution needs. With regard to detailed specifics as to how particular blocks function, the comments in the code can be consulted; alternatively simulation of the design will additionally show the functionality.

Downloading

All designs are available on the Achronix self-service FTP site, available at https://secure.achronix.com, located in the folder `/public/Achronix/Reference_Designs/Speedster7t`.

Packaging

The designs are all packaged in a zip file archive, using the following format:

```
<design_name>_<design_version>_<date_of_packaging>.zip
```

The archive contains all the source code, scripts to build the design, simulation script files, and optionally, GUI-based project files.

Versions

Design versions 1.x are intended to be used with ACE versions 7.x only. Design versions 2.x onward are intended to be used with ACE 8.x and upward versions. Please ensure you download the correct design release to match your ACE version

Operating System

All the designs have been build and tested using CentOS7. The Linux scripts provided are for the bash shell. In addition, Windows batch scripts are included for building the designs under Win7/10.

Tool Versions

All designs were tested with the following tools:

Software	Version
ACE	8.0
Synplify Pro	P-2019.03X-SP1-beta5
Modelsim Questa	10.6a
Synopsys VCS	O-2018.09-SP1-1

ACE_INSTALL_DIR

In order to support relocatable projects, the designs make use of an environment variable, `ACE_INSTALL_DIR`. This variable should be set to the ACE installation directory (where `ace.exe` or `ace` is installed). This variable is then used by both synthesis and simulation to correctly locate the ACE library files.

Building the Designs

The designs make use of a consistent build environment, using scripts to run both Synplify Pro and ACE in batch mode.

Batch flow

In the root directory, there is a `/scripts` directory, within which will there is a `build.sh` launch script. Before running this script the user has to have both Synplify Pro and ACE correctly installed, licensed and available in their path.

When `build.sh` is run (with the default options), it will create the following;

- `/results/syn` directory – Synplify Pro is executed in batch mode, synthesizing the design, which generates a netlist in `/results/syn/rev_1/<design_name.vm`. If synthesis is unsuccessful the user should consult `/results/syn/rev_1/<design_name>.srr` for details of any synthesis failure. Options to the generated Synplify Pro project file are controlled by `/src/constraints /synplify_options.tcl.`
- `/results/ace` directory – After synthesis, ACE is run in evaluation mode (meaning that no I/O pins need be specified). If any options are required for the ACE generated project, these are controlled by the `/src /constraints/ace_impl_options.tcl` file.

Other constraint files used in the flow (all located in `/src/constraints`) are:

- `ace_timing.sdc, <design_name>.sdc` – timing constraint files used by both synthesis and ACE.
- `<design_name>.fdc` – FPGA constraints used by Synplify Pro to set non-timing related directives and attributes, such as compile points.
- `<design_name>.pdc` – placement constraints used by ACE.

The full list of what files are used in the flow, and by which tool, can be determined by referring to the relevant `/src/filelist_xx.tcl` file.

GUI Flow

Although the initial builds are done using the batch flow, the flow will write out both the relevant project files. The user is then able to open both ACE and Synplify Pro in GUI mode and to interactively re-run or edit the builds.

Simulation

The designs have a consistent simulation environment, providing scripts for Mentor QuestaSim and Synopsys VCS simulators.

Location

All designs have a `/sim` directory located in the design root directory.

Auto File List Generation

The simulation file list is auto generated from the `../src/filelist.tcl` file. The script to create the simulation file list is `../scripts/create_sim_project.tcl`, and it uses a template file `../scripts /sim_template.f` to define the general simulation options. This create script is then called by the specific simulator script, as detailed below. The resultant file, `sim_filelist.f`, combines the template contents with the specific list of files in `../src/filelist.tcl`.

Files

In each `/sim` directory the following scripts are located:

- `qsim_<design_name>.do` – Unix shell launch script for QuestaSim. Auto creates `sim_filelist.f`., then compiles and opens the GUI. User can then execute the desired > run commands.
- `vcs_flow.sh` – Bash shell launch script for VCS. Auto creates the sim_`filelist.f`. Has options to enable or disable the compilation, simulation or view flow stages.
- `/wave_files/wave.do` – waveform file for QuestaSim.
- `/wave_files/session.sim_output_pluson.vpd.tcl` – session file for DVE waveform viewer.

Running the Simulation

QuestaSim

To launch a QuestaSim simulation:

```
code

$ vsim -do qsim_<design_name>.do
```

The script compiles the files and opens the wave display script file (`wave.do`). The user can then interactively run the simulation for the desired period.

VCS

To launch a VCS simulation:

code

```
$ ./vcs_flow.sh
```

When run with no options, the script compiles the files, runs the simulation, (writing the results to the `sim_output_pluson.vpd` waveform capture file), then opens the DVE waveform viewer with the generated waveform file.

The VCS flow script supports various options to disable each of the flow stages: These options can be displayed with the `-h` or `-help` options

code

```
$ ./vcs_flow.sh -help

Usage     : <script>.sh [-nocomp] [-nosim] [-noview] [-help] [-h]
  -nocomp : Do not compile a new simv executable
          : Note  : simulation execution and viewing rely on there being an existing simv
  -nosim  : Do not execute simv to run simulation
  -noview : Do not view the waveform
-help, -h : This help screen
```

Results Verification

All the designs make use of a self-checking test bench which compares the results generated from the RTL to a verified output. The verified output can come from a number of sources, either a math package, a software model, or an RTL behavioral model. The details of the applicable verification source is given in the detail of each individual design.

Chapter - 2: MLP_Conv2D

Introduction

The MLP_Conv2D design is a full-featured design that convolves a 2D input image with multiple kernels simultaneously. The design makes extensive use of the MLP and BRAM blocks, with each MLP performing 12 int8 multiplications in a single cycle. Further the MLP blocks are chained together using their cascade paths to efficiently pass image data up a column of MLPs. The BRAM blocks are equally cascaded — this arrangement then allowing multiple kernels to be processed simultaneously.

The design acquires data from the network on chip (NoC) using a NoC access point (NAP) for reading the data, and a further NAP for writing the data. The NoC is attached to the GDDR6 controllers in the Speedster7t device to connect to external memory. This design shows how accessing and writing data to these devices is greatly simplified by the use of the NAP.

The design includes Octave files to generate example test images, and kernels; and to generate check files that can be used to compare design results.

> **Note**
>
> Rather than be considered a full production solution for AlexNet processing, this reference design should be considered as a guide to how to effectively perform 2D convolution, and the principles used can be applied to other 2D convolution requirements.

AlexNet

Although the MLP_Conv2D design is initially configured for AlexNet image and kernel sizes, 2D convolution is a general process and so the design can be reconfigured and adapted to achieve many different 2D methods.

The general principle of 2D convolution is to pass a kernel (a 2D matrix) across an image (effectively another 2D matrix). For each calculation, the kernel is centered upon a pixel of the input image, and a multiplication is performed for each kernel value (known as a weight) and the pixel that it is currently aligned with. The sum of these multiplications gives a specific convolved result of the original image pixel. The kernel is then moved to the next pixel and the process repeated.

With fully trained kernels, the 2D convolution generates an output result image highlighting particular features of the input image, such as vertical lines, horizontal lines, diagonal lines of varying angles, and curves of varying radius. These feature highlights can then be fed into further processing layers (including further 2D convolutions) to combine the features sets into groups that can then be identified (usually in software) to be particular objects. Therefore, the 2D convolution process should not be considered as the full solution for image recognition, but as a single key component of a chain of processing operations.

Multiplication Density

The challenge with 2D convolution is the quantity of multiplications needed, which is where the MLP with its dedicated array of multipliers demonstrates its capabilities. For the AlexNet configuration, each kernel is 11 × 11 = 121 weights. However, the convolution is actually in 3D in that the input image has three layers (RGB) and so does the kernel. Therefore, there are 121 × 3 = 363 multiplications to produce a single output result. The AlexNet input image is 227 × 227; however, this image is processed with a stride of 4,(the kernel is moved by four pixels

between calculations). This process results in an output result matrix of 54 × 54 = 2916 results. Therefore, for one image there are 363 × 2916 = 1,058,508 multiplications required; i.e., more than one million multiply-accumulate operations are required to process a single image.

The MLP_Conv2D design is structured to process 60 kernels across a single image in one pass, performing more than 60 million multiply-accumulate operations in the single pass.

Performance

The MLP_Conv2D design is targeted to operate at a frequency of 750 MHz. In this configuration, a single MLP is able to convolve a single 227 × 227 RGB input image with an 11 × 11 kernel in 137 µs — equivalent to 7.7 tera operations per second (TOPS). However, the MLP_Conv2D design is structured as 60 MLPs operating in parallel, allowing 60 input images to be convolved at the same time — the equivalent of 463 TOPS. Finally the extensive performance of the Speedster7t fabric can be fully shown when up to 40 instances of the MLP_Conv2D design are instantiated into a single device, with each instance transferring data via it own NAPs to the GDDR6 memories. The efficient design of the MLP_Conv2D is such that data is only read from memory once and reused within each instance, allowing all 40 instances to run in parallel, achieving a combined performance of 18,500 TOPS — the equivalent to 28,2000 images being processed per second.

Utilization

The MLP_Conv2D is designed around the MLP and BRAM block capabilities and uses their respective cascade paths for large data flows. Equally the NAPs allow for data to be routed directly from the external memory interfaces to the design. These features result in minimal additional logic or routing requirements as illustrated by the utilization tables below:

Table 1: ***Resource Utilization for Single MLP_Conv2D Instance***

Resource	Quantity	Percentage of the Device
MLP	60	2.5%
BRAM	62	2.5%
LUT	1210	< 1%
DFF	4060	< 1%

Table 2: ***Resource Utilization for 40 MLP_Conv2D Instances***

Resource	Quantity	Percentage of the Device
MLP	2400	94%
BRAM	2480	97%
LUT	50K	8%
DFF	165K	12 %

As shown above, even for the 40 instance design, which can deliver 18,500 TOPS, the majority of the DFF and LUT logic is available within the device to perform other functions.

Accessing the Archive

See Downloading (see page 6) for how to get the archive. The required archive is named `mlp_conv2d_<version>_<date>.zip`.

Description

Parameters

A number of parameters can be configured for the MLP_Conv2d design, set in the design's top-level file, `../src/rtl/mlp_conv2d_top.sv`. The parameters follow the naming convention for the TensorFlow 2D convolution.

Table 3: *MLP_Conv2d Design Parameters*

Parameter	Default	Function
BATCH	4	Number of simultaneous kernels being convolved, which equates to the number of MLP blocks used. Range is 2-60. As the design uses the MLPs in fixed columns, the legal values are 2-12, 28, 44, 60. If the parameter is set to a value between the legal values, it will be rounded up to the next legal value.
IN_HEIGHT	227	Number of lines in the input image.
IN_WIDTH	227	Number of pixels in an input image line.
IN_CHANNELS	3	Number of layers of the input image; Default is 3 for RGB images.
FILTER_HEIGHT	11	Number of lines of each kernel.
FILTER_WIDTH	11	Number of weights in a kernel line.
OUT_CHANNELS	1	Number of output channels (not used).
INF_DATA_WIDTH	144	Data width of the input FIFO (2 × 72 bits).
BRAM_ADDR_WIDTH	10	Address width of MLP BRAM; each is 1K deep.

Stride

For the current design, the STRIDE (number of steps the kernel is moved between convolutions) is fixed at 4. Future versions of this design will support more stride values.

Block Diagram

The MLP_Conv2D is architected as per the figure below.

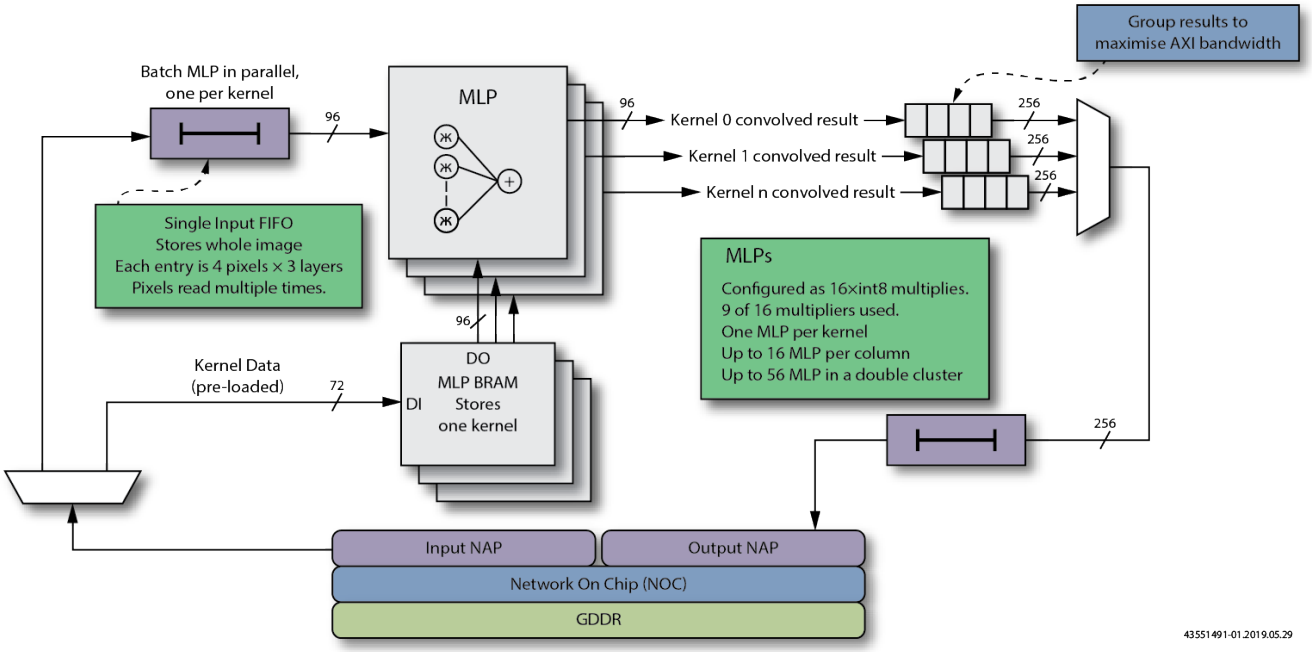

Figure 1: ***MLP_Conv2D Block Diagram***

Data Flow

Single MLP

Each MLP has a tightly coupled BRAM. For this design, this BRAM is used to store the kernel and to play it out multiple times to the MLP. On initialization the kernel for each BRAM is read from the input NAP and written to the appropriate BRAM. The BRAM is configured as 72 bits on the write side, with reading set to 144 bits. During operation, only 96 bits are used as the kernel weights are read as 4 weights × 3 layers × 8 bits.

Initial image data is read from the NAP into the input FIFO, which is used to store the image as a series of lines. Although listed as a FIFO, this input memory operates as a re-readable FIFO, in that lines can be read from it multiple times. The memory is configured as 144 bits wide, being composed of two BRAM72K. However, only 96 bits are used, with each word consisting of 4 pixels × 3 layers × 8 bits. On initialization enough lines are read in to match the number of lines in the kernel plus the number of lines needed for a vertical stride. i.e.

```
Initial Lines read = FILTER_HEIGHT + STRIDE
```

Once the initial data and kernels are loaded, the computation starts.

From the input FIFO, the first image line is read, reading an amount of image data pixels that match the horizontal size of the kernel. As these pixels are read, the matching kernel weights are read. The MLP multiplies each of these 96-bit streams as 12 int8 values and accumulates the result.

The input FIFO is advanced to the second line, and the start of line pixels are read and multiplied against the matching kernel weights from the second line of the kernel. This process is repeated until all lines of the kernel have been multiplied against the appropriate pixels in the top, left-hand corner of the input image. During this process the MLP has accumulated the result; this result is now the 2D convolution of the top left-hand corner of the image convolved with the kernel. This result is output from the MLP as a 16-bit result.

The process is repeated with the input FIFO being advanced across the line by the number of pixels set by the STRIDE parameter (for the current design, STRIDE is fixed at 4). At the conclusion of each process cycle, another result is generated until the appropriate number of results horizontally have been obtained.

The input FIFO is then advanced down by a STRIDE number of lines, and the process repeated to generate the convolution results for the next set of lines in the input image. As the input FIFO is advanced down, the initial lines in the FIFO are no longer required, and so in parallel with the MLP computation, the next set of STRIDE lines for the input image is loaded.

When considering bandwidth requirements from the external memory source, it can be seen that the image and kernels are only read from memory once. They are then reused from their respective BRAM, reducing the overall load on external memory bandwidth.

Multiple MLPs

A compelling feature of the MLP is the ability to cascade data and results from one MLP, or BRAM, to another directly above it in the same column. The MLP_Conv2D makes use of these cascade paths by placing the MLPs and their associated BRAM in groups of columns.

When loading the BRAM with the kernel, the cascade path is used to pipeline the data to each BRAM, and the BRAM block address mode is used to select which BRAM to write the kernel to.

During calculation, the input image data is cascaded up the column of MLPs, so that each MLP receives the image data one cycle after its neighbor below. At the same time, the BRAM read address, which controls the kernel reading, is cascaded up the BRAM column, with a one-cycle delay. In this way each MLP receives the same image data and same kernel read address, one cycle after its preceding MLP. The difference in calculation for each MLP is that its associated BRAM will have different kernel data. The result is that the one image is convolved with multiple kernels in parallel. The number of parallel convolutions is referred to as the BATCH.

Results

As previously detailed, each MLP produces a 16-bit result for each convolution of kernel and section of image. The MLPs are arranged in columns of 16, so from this column a 256-bit word is generated composed of the results from each MLP in the column. This 256-bit word is then written to the output NAP. This arrangement results in the convolved results being stored in memory as layers from the same image; so matching the input word arrangement when the three layers or RGB are stored in a single input word.

This arrangement then allows for parallel processing of the convolved results into a activation layer, as the activation function can be performed in 16 parallel instances on the full 256-bit result. Equally once the 256-bit result is written back to memory via the output NAP, then the results could be read back into a further 2D convolution circuit.

Floorplan

The design of the MLP_Conv2D is architected to match the structure of the Speedster7t fabric. In the Speedster7t architecture, for each NAP there are 32 MLPs. The design is optimized to use two NAPs, one for reading, one for writing, and hence 64 MLPs.

However, the input and output FIFOs require two BRAM 72K memory blocks each to create a combined memory which is 256 bits wide. These memories will, therefore, consume four of the available 64 sites for data I/O.

The floorplan of the design is arranged to use the four columns of MLPs associated with the two NAPs; however, both the first and last columns use 14 MLP sites each, leaving two MLP sites free for the input and output FIFOs respectively. The middle two columns use all 16 available MLP sites. In the floorplan, the columns are arranged such that the first column, which has the input FIFO memory at the bottom, is located adjacent to the NAP in order to improve timing.

An example floorplan of the design (with routes highlighted) using the maximum available 60 MLP processing blocks is shown below:

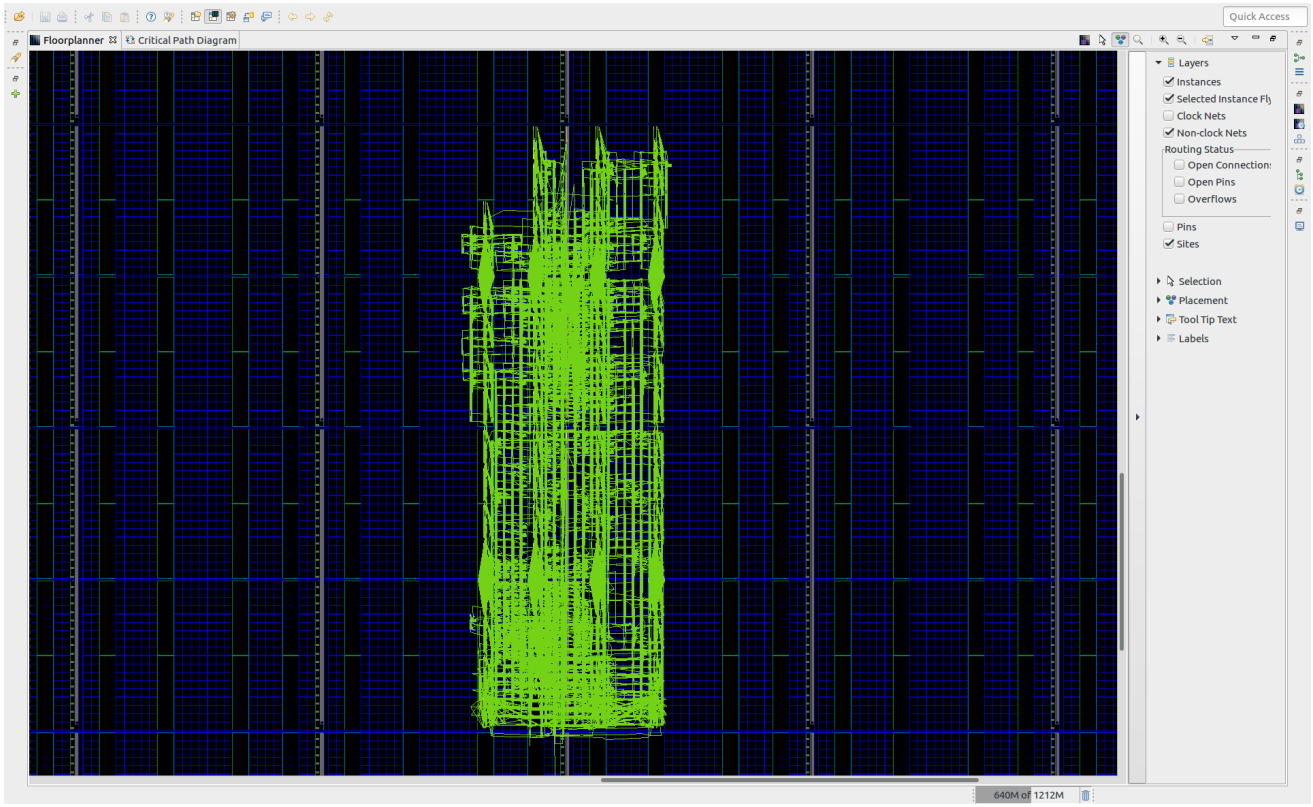

Figure 2: ***60 MLP Floorplan***

Building the Design

The design can be build using both a scripted flow, and also using the tool GUI flows.

For details of the scripted build flow refer to Building the Designs (see page 7). In addition to supporting a scripted flow, the user is recommended to modify and adapted the scripted build flow to fit within their own scripted build environment.

To perform a build flow using the tool GUI flow, within the `/src/syn` and `/src/ace` directories, GUI project files are provided for Synplify Pro and ACE respectively. The user should first synthesise the design using the Synplify Pro GUI, and then Place and Route using the ACE GUI.

The script flow supports two build options:

- A single instance, using two NAPs and 60 MLPs, named mlp_conv2d_b60 (batch 60). This build option is the default scripted build flow option, and is also supported by the GUI build flow.
- A full-chip build, using 40 instances of the b60 design. This build demonstrates the unique features of a Speedster7t device by using the built-in NoC, with each instance communicating with the memory using the NAPs. Therefore, each instance is separate from other instances. As a result, the performance in the form of F_{MAX} does not degrade as multiple instances are used. This full-chip build is named mlp_conv2d_b60x40 (40 instances of the single instance) and uses all 80 NAPs in the device, as well as 94% of the MLP and BRAM72K.

> **Note**
>
> Building this device can take between 4 and 8 hours dependent upon the build environment. A minimum memory of 24 GB is also recommended.

Build selection is performed in `/build_scripts/build.sh`. The floorplan of the full b60x40 build is shown below

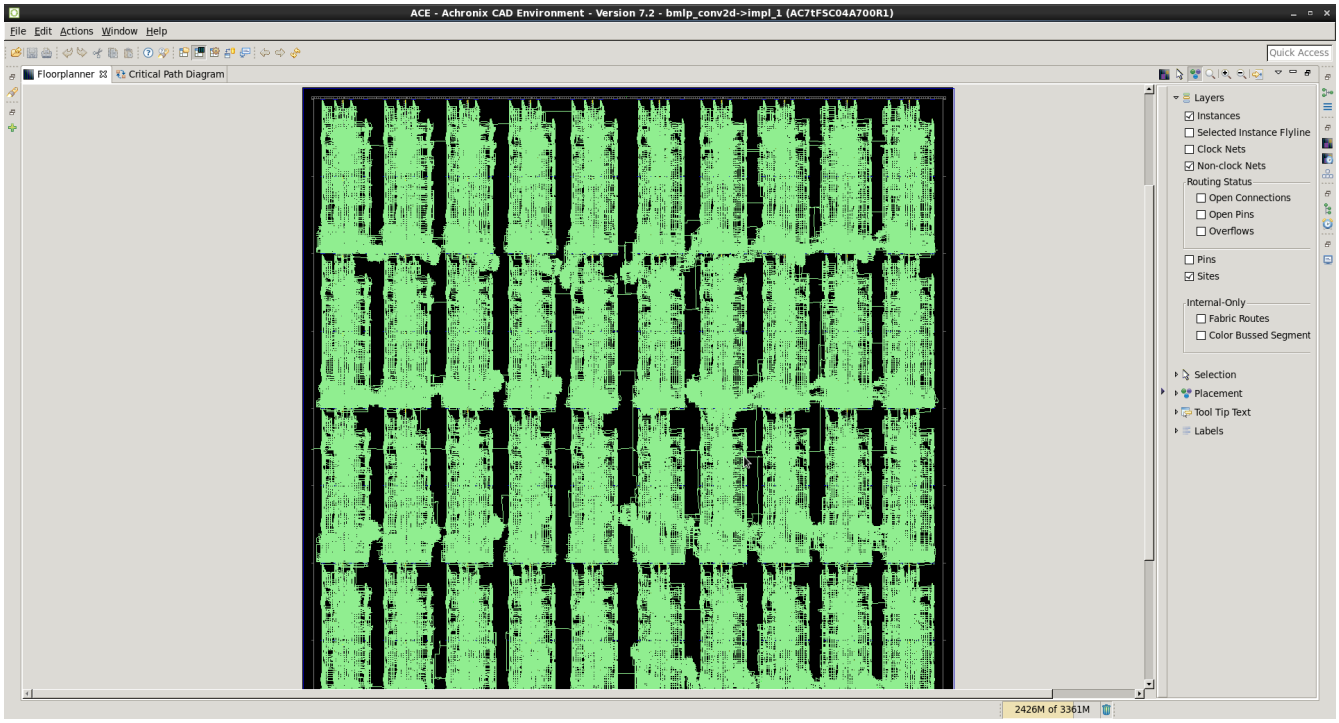

Figure 3: ***2400 MLP Floorplan***

Simulation

For simulation the design makes use of behavioral models of the NAPs, These models contain behavioral memory models which mimic the external storage connected to the NoC. The input NAP memory is programmed with a stimulus file containing the input image and the kernels. The output NAP model is programmed with an expected results file When the simulation is executed, input data is read from the input NAP, processed, and written to the output NAP, where it is compared to the expected results on the fly.

Stimulus Files

The stimulus and result files are generated using `.m` files which can be processed by either of the popular math packages Octave or Matlab.

The input image, in PNG format, is read and converted to the correct arrangement of int8 values. Multiple kernels are generated, formatted, and written to the input stimulus file along with the input image. The input stimulus file is written to `/src/oct/nap_in.txt`. In the same process, each kernel is convolved with the input image, and the result written to the output NAP file, `/src/oct/nap_out.txt`.

Caution!

If the input stimulus file is modified to use different size images or kernels than those provided in the reference design, then the parameters configuring the MLP_Conv2D design must be modified accordingly.

Memory Map

The memory map used to generate the input stimulus file must match the memory map encoded within the MLP_Conv2D design. The reference design memory map is detailed in Input stimulus memory map (see page 11).

Table 4: *Input Stimulus Memory Map*

Base Address	Content
0x0000_0000	Kernel 0
0x0001_0000	Kernel 1
0x0002_0000	Kernel 2
0x003b_0000	Kernel 60
0x0040_0000	Input image

If the user wishes to modify the memory map, then it is necessary to modify both the stimulus source file, and the RTL accordingly.

The stimulus file is written by `/oct/convolve_2d_debug.matrix.m`. The address map is defined at the top of the file.

The reading of the input NAP is controlled by `/rtl/dataflow_control.sv`. The address map is controlled by parameters at the top of the file.

Memory Format

The NAP presents the external memory data as a 256 bit word. For the kernels and the input image, 96 bits of each memory word is used, formatted as shown in Input stimulus word format (see page 11)

Table 5: *Input Stimulus Word Format*

	[255:128]	[95:88]	[87:80]	[79:72]	[71:64]	[63:56]	[55:48]	[47:40]	[39:32]	[31:24]	[23:16]	[15:0]	[7:0]
Pixel / Weight	reserved	3			2			1			0		
Layer		2	1	0	2	1	0	2	1	0	2	1	0

Running the Simulation

The simulation is executed from the `/sim` directory, within which are Unix bash shell script files to run Mentor QuestaSim, and Synopsys VCS simulators. Before running the simulation, the user needs to ensure that the appropriate simulator is available in their path.

For additional details of how to configure and run reference design simulations, refer to Simulation (see page 8).

With Questa

To launch the Questa simulation, enter:

```
$ vsim -do qsim_mlp_conv2d.do
```

With VCS

To launch the VCS simulation, enter:

```
$ ./vcs_mlp_conv2d.sh
```

For the provided reference design, once the appropriate simulator has opened and compiled the design, a run of 200 µs is sufficient to load the kernels, process the input image and write the results to memory. If the simulation has run successfully, the message "`TEST PASSED`" will be written to the simulation console.

Chapter - 3: Dot Product

The dot product suite of reference designs demonstrate how to perform vector multiplication, with the result being the sum of each of the elements of the vectors multiplied together, commonly known as the dot product. This vector multiplication is shown below:

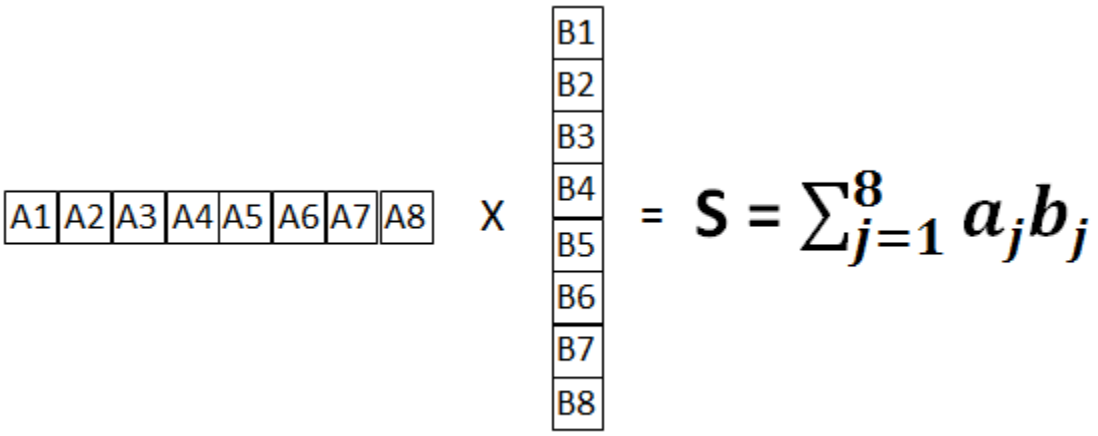

Figure 4: ***Dot Product Operation***

The sum *S* is composed of the sum of each of the vector elements multiplied together, hence $S = a_1b_1+a_2b_2 + a_3b_3$ ....

> **Note**
>
> In the dot product reference designs, with the notable exception of the the dot_product_10bfp_fp10_6x6 design, i_a and i_b inputs can be considered interchangeable, as the final result is the sum of each vector multiplied together. For the dot_product_10bfp_fp10_6x6 design, as the i_b input is firstly stored in BRAM, the user may wish to use these inputs for vectors that are reused multiple times.

Design Configurations

The dot product suite makes use of the MLP72 in a number of different configurations, demonstrating the range of numerical types and multipliers supported by the MLP72. For additional details on each design refer to the `README.txt` in the relevant `/src` directory.

dot_product_N_8x8

Description

This design demonstrates the dot product of a sequence of int8 values. On each cycle *N* int8 inputs are multiplied and summed. The result is then accumulated and available two cycles after the input. The

accumulation is controlled by the `i_first` and `i_last` inputs. The `i_first` input signal indicates the first set of inputs to sum and to zero the accumulation. The `i_last` signal indicated the last set of inputs to sum and add to the accumulation. The final value is output with `o_valid`.

Configuration

Table 6: *dot_product_N_8x8 Configuration*

Input Format	Output Format	Parallel Multiplications	Number of MLP72
int8	int48	N	1

Ports

Table 7: *dot_product_N_8x8 Ports*

Port	Direction	Description
i_a	Input	"a" input to multiplication. Array of N x int8 (Nx8 bits).
i_b	Input	"b" input to multiplication. Array of N x int8 (Nx8 bits).
i_first	Input	Indicates first group of inputs to sum and start accumulation. Sets internal accumulator to 0.
i_last	Input	Indicates last group of inputs to sum and accumulate.
o_sum	Output	48b integer accumulated output.
o_valid	Output	Validates o_sum output.

Table Note

`i_first` and `i_last` cannot be coincident. However, they can be on adjacent cycles.

Timing Diagram

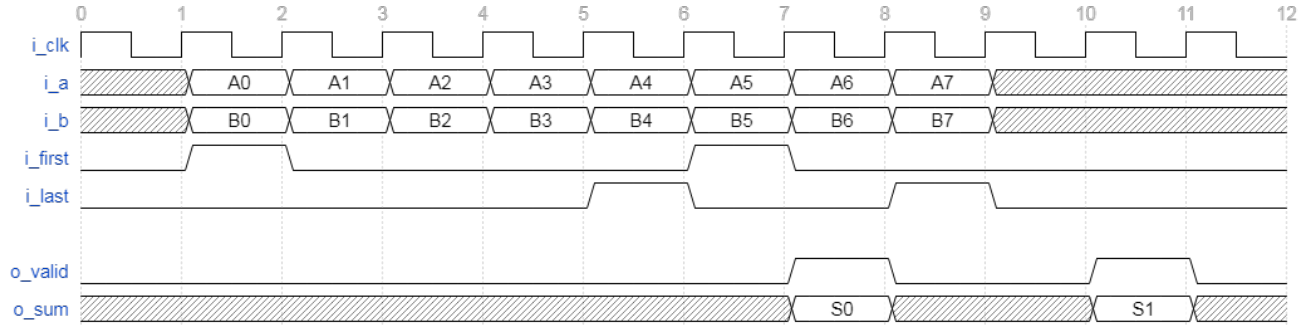

Figure 5: *dot_product_N_8x8 timing diagram*

Where:

- A0 = Array of N x a inputs, i.e. {a_1, a_2 , a_3 .. a_n }[0]
- B0 = Array of N x b inputs, i.e. {b_1, b_2 , b_3 .. b_n }[0]
- S0 = Sum of array of multiplications, = {a_1*b_1 + a_2*b_2 + a_3*b_3 + ... a_n*b_n }[0] + {a_1*b_1 + a_2*b_2 + a_3*b_3 + ... a_n*b_n }[1] + {a_1*b_1 + a_2*b_2 + a_3*b_3 + ... a_n*b_n }[2] + ...

dot_product_2bf16_doutcasc4

Description

This design demonstrates the dot product of a number of bfloat16 inputs. The design consists of four MLP72, connected using their cascade paths. Each MLP72 sums the result of two parallel multiplications, with each multiplication being the result of an `i_a` input multiplied by an `i_b` input (both being bfloat16 numbers). The sum from each MLP72 is cascaded up the column of MLP72 to the next block above. In the last MLP72, on each cycle, the sum of the eight parallel bf16 multiplications is calculated.

The final result is the accumulated sum across a number of input cycles. The accumulation is controlled by the `i_first` and `i_last` inputs. The `i_first` input signal indicates the first set of inputs to sum and to zero the accumulation. The `i_last` signal indicated the last set of inputs to sum and add to the accumulation. The final value is available six cycles after `i_last`, and is qualified with `o_valid`.

Configuration

Table 8: *dot_product_2bf16_doutcasc4*

Input Format	Output Format	Parallel Multiplications	Number of MLP72
bfloat16	bfloat16	8	4

Ports

Table 9: *dot_product_2bf16_doutcasc4 Ports*

Port	Direction	Description
i_a	Input	"a" input to multiplication. Array of 4 x 2 x bfloat16 (128 bits).
i_b	Input	"b" input to multiplication. Array of 4 x 2 x bfloat16 (128 bits).
i_first	Input	Indicates first group of inputs to sum and start accumulation. Sets internal accumulator to 0.
i_last	Input	Indicates last group of inputs to sum and accumulate.
o_sum	Output	bfloat16 accumulated output.
o_valid	Output	Validates o_sum output..

Timing Diagram

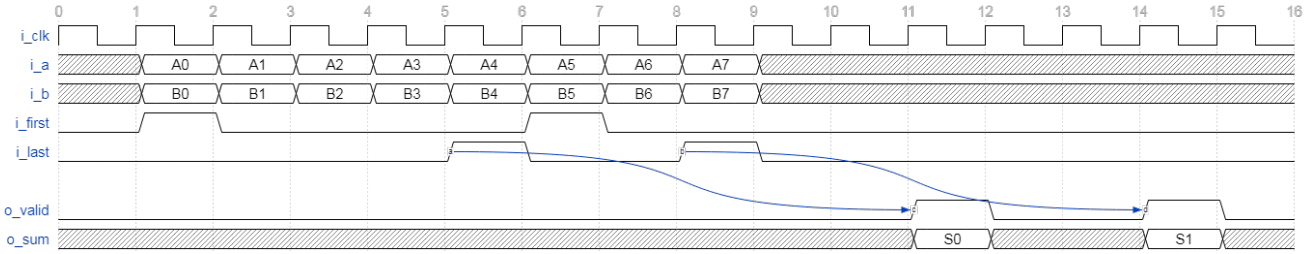

Figure 6: ***dot_product_2bf16_doutcasc4 Timing Diagram***

Where:

- A0 = Array of N × a inputs, i.e., $\{a_1, a_2, a_3 .. a_n\}[0]$
- B0 = Array of N × b inputs, i.e., $\{b_1, b_2, b_3 .. b_n\}[0]$
- S0 = Sum of array of multiplications, = $\{a_1*b_1 + a_2*b_2 + a_3*b_3 + ... a_n*b_n\}[0] + \{a_1*b_1 + a_2*b_2 + a_3*b_3 + ... a_n*b_n\}[1] + \{a_1*b_1 + a_2*b_2 + a_3*b_3 + ... a_n*b_n\}[2] + ...$

dot_product_10bfp_fp10_6x6

Description

This design computes the accumulated sum of a block of 10 floating-point number pairs. Each pair of numbers is multiplied and the result added to the sum. The inputs are in fp10 format (5 bits mantissa + 5 bits exponent). This design uses a single MLP72, and its tightly coupled BRAM72K, to demonstrate how the combination of blocks can process a total input bit width that is larger than any of the individual inputs bit widths. This capability is achieved by the design operating in two phases; in phase one the `i_b` inputs are written to the BRAM. In phase two the `i_a` inputs are applied to the MLP72 as the `i_b` values are read from the BRAM72K.

An advantage of this structure is that if the `i_b` values need to be used more than once, such as for a kernel or weights, then they only need be written the once and can be re-read multiple times as different sets of `i_a` inputs are applied.

During phase two, the final result is the accumulated sum across the number of input cycles. The number of cycles of i_a must match the number of i_b values written. The accumulation is controlled by the `i_first` and `i_last` inputs. The `i_first` input signal indicates the first set of `i_a` inputs to sum and to zero the accumulation. The `i_last` signal indicated the last set of `i_a` inputs to sum and add to the accumulation. The final value is available four cycles after `i_last`, and is qualified with `o_valid` .

Configuration

Table 10: ***dot_product_10bfp_fp10_6x6***

Input format	Output format	Total multiplications	Number of MLP72
fp10	fp16	10	1

Ports

Table 11: *dot_product_10bfp_fp10_6x6 Ports*

Port	Direction	Description
i_a	Input	"a" input to multiplication. Array of 10 × fp10 (100 bits)
i_b	Input	"b" input to multiplication. Array of 10 × fp10 (100 bits)
i_b_addr [9:0]	Input	Address in BRAM to write the i_b values
i_wren	Input	Write enable to BRAM for i_b values
i_first	Input	Indicates first group of inputs to sum and start accumulation. Sets internal accumulator to 0
i_last	Input	Indicates last group of inputs to sum and accumulate.
o_sum	Output	fp16 accumulated output
o_valid	Output	Validates o_sum output.

Timing Diagram

The timing diagram is divided into two, showing each of the two phases:

Phase 1

Write i_b values to BRAM72K.

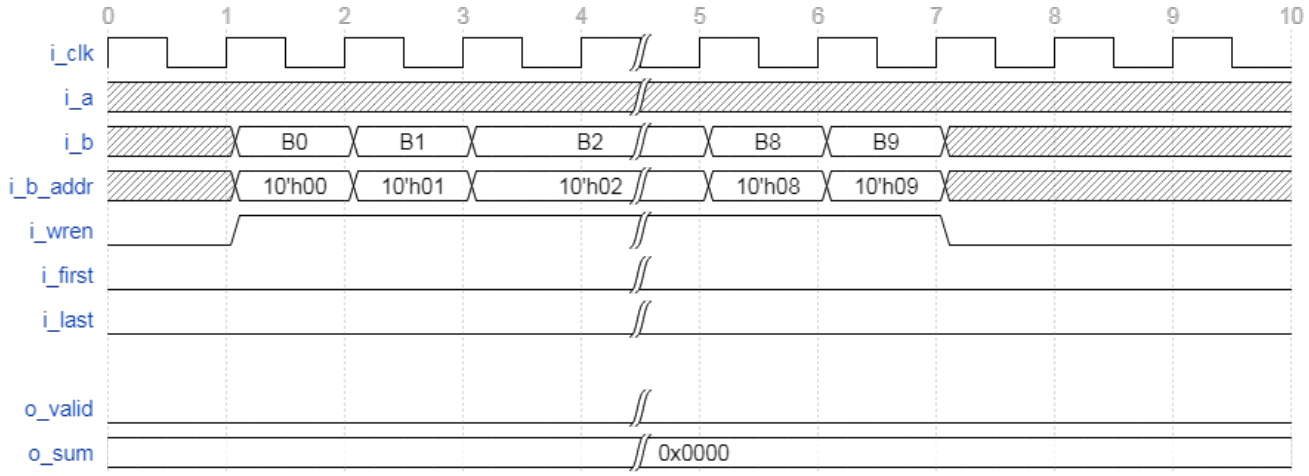

Figure 7: *dot_product_10bfp_fp10_6x6 Phase 1 Timing Diagram*

Phase 2

Perform dot product.

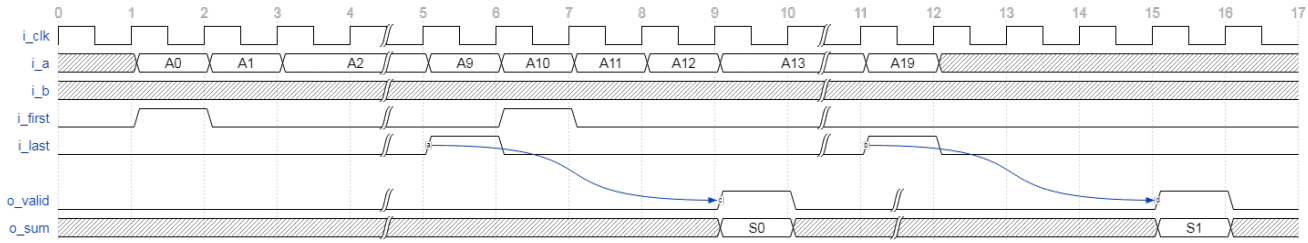

Figure 8: *dot_product_10bfp_fp10_6x6 Phase 2 Timing Diagram*

Builds

All the dot product reference designs make use of a common build environment. The builds are all targeted for the AC7tFSC04A700R1 device. For full details of this build environment, please refer to Building the Designs (see page 7).

For each of the dot product designs, a top chip-level file is provided, `/src/rtl/<design_name>_top.sv`. These top-level files incorporate local registers for the signal I/O, which creates timing paths between the I/O and the MLP. Therefore with these builds the user can determine the timing closure and performance of the dot product design when incorporated within surrounding logic.

To select between different builds, edit the `/build_scripts/build.sh` file.

As the dot product reference designs are small in nature, each of them will build relatively quickly. All of the designs are targeted to close timing at 750 MHz.

Simulation

All of the dot product reference designs have a common simulation methodology, with the same structure of simulation scripts, as detailed in Simulation (see page 8).

Each of the dot product reference designs compares the MLP calculated result against a SystemVerilog behavioral model of the math operation. For the dot_product_N_8x8 reference designs, this verification model, located in `/src/tb/test_sequence.vh` implements integer math. The results from `test_sequence.vh` are then compared against the output of the dot_product design.

For the floating-point dot_product designs, a library file, `/src/tb/acx_fp_sim.sv` is used to correctly perform any floating-point calculations, including any common exponent changes, rounding and truncation. These library functions are called by the relevant `test_sequence.vh` in order to generate floating-point verification results that are compared on the fly to the output of the MLP. These library functions also print useful intermediate results and debug, which may assist users when debugging or designing other floating-point operations with the MLP.

For all of the dot product reference designs, the verification is done on the fly; and the result of each calculation is displayed as either correct or in error.

Revision History

Version	Date	Description
1.0	19 Jun 2019	• Initial Achronix release.
1.1	16 Sep 2019	• Update tools to ACE 8.0, and Synplify Pro P-2019.03X-beta5 • Break reference design releases into 1.x and 2.x, to be used with ACE 7.x and ACE 8.x onwards respectively. • Simulation flow now uses auto-generated file list. • Update VCS simulation flow to use generic flow script