Speedster7t MLP Reference Design Guide (RD026)

Achronix
Data Acceleration

January 19, 2021 **Reference Design**

Introduction

This document describes a number of reference designs based on the Speedster®7t machine learning processor (MLP) . This design shows the flexible configuration of the MLP plus its ability to be combined with its associated BRAM and LRAM to create a powerful, multi-mode matrix and vector multiplication systems.

Designs

There are four designs available within this design suite:

- 2D Convolution (see page 1) – Performs an AlexNet 2D convolution on an image.
- Split MLP, Shared BRAM (see page 9) – Demonstrates how to share a BRAM between two MLPs, allowing for the remaining BRAM to be used for other tasks.
- Dot Product (see page 17) – Demonstrates how to produce the dot product from multiplying two vectors
- Matrix Vector Multiply (MVM) (see page 23) – Demonstrates how to multiply a matrix with a vector

2D Convolution

Introduction

The MLP 2D Convolution, (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 and the same number of additions 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 × 2 × 2916 = 2,117,106 calculations required; i.e., more than two 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 120 million multiply-accumulate operations in the single pass.

Performance

The MLP_Conv2D design is targeted to operate at a frequency of 600 MHz. In this configuration, a single MLP is able to convolve a single 227 × 227 RGB input image with an 11 × 11 kernel in 170 µs — equivalent to 12.4 giga operations per second (GOPS). 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 803 GOPS. 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 29,684 GOPS, or 30 tera operations per second (TOPS) — the equivalent to 235,000 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 32 TOPS, the majority of the DFF and LUT logic is available within the device to perform other functions.

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.

Parameter	Default	Function
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.

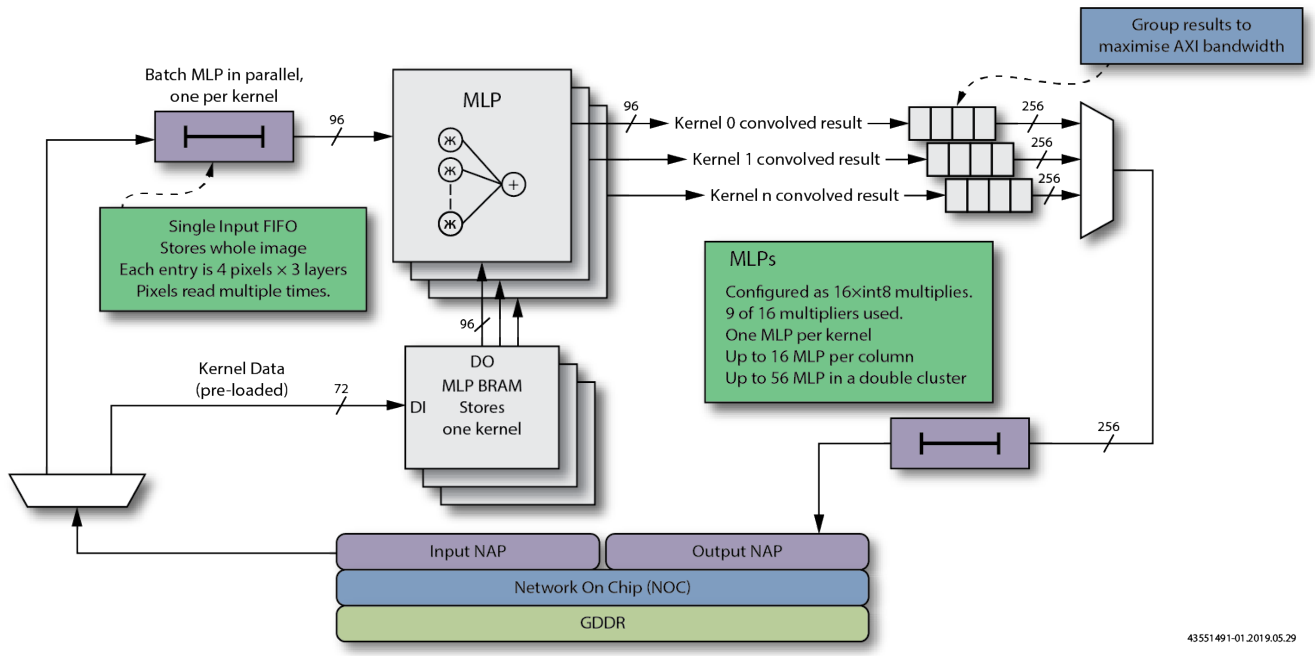
Group results to maximise AXI bandwidth
Batch MLP in parallel, one per kernel
96
MLP
96
Kernel 0 convolved result
Kernel 1 convolved result
Kernel n convolved result
256
256
256
Single Input FIFO
Stores whole image
Each entry is 4 pixels × 3 layers
Pixels read multiple times.
MLPs
Configured as 16×int8 multiplies.
9 of 16 multipliers used.
One MLP per kernel
Up to 16 MLP per column
Up to 56 MLP in a double cluster
96
DO
MLP BRAM
Stores
one kernel
DI
Kernel Data (pre-loaded)
72
256
Input NAP
Output NAP
Network On Chip (NOC)
GDDR
43551491-01.2019.05.29

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:

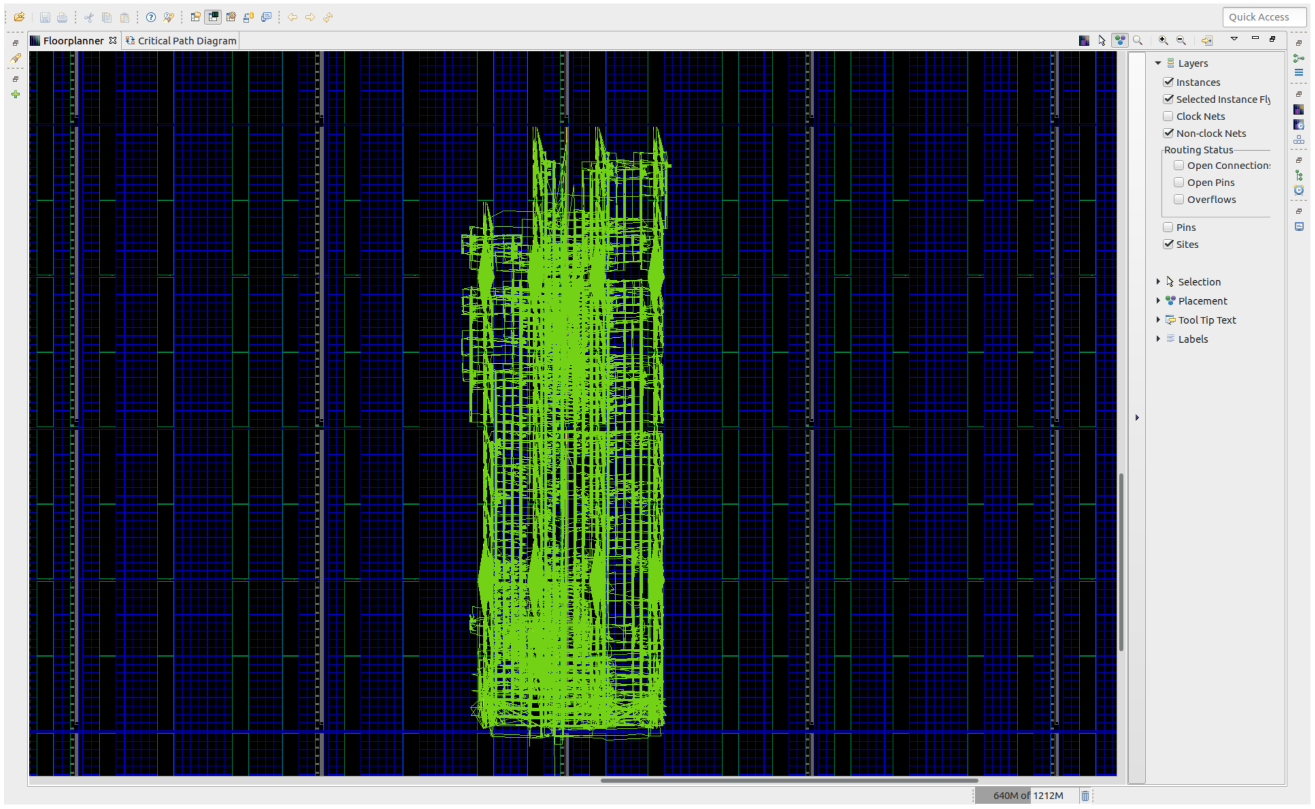

Figure 2: ***60 MLP Floorplan***

Building the Design

The design can be build using both a scripted flow using a makefile, and also using the tool GUI flows. For details of the scripted build flow, refer to Building the Reference Design (see page). If required, 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 makefile scripted 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/makefile`, by changing the `variant` value. The floorplan of the full b60x40 build is shown below

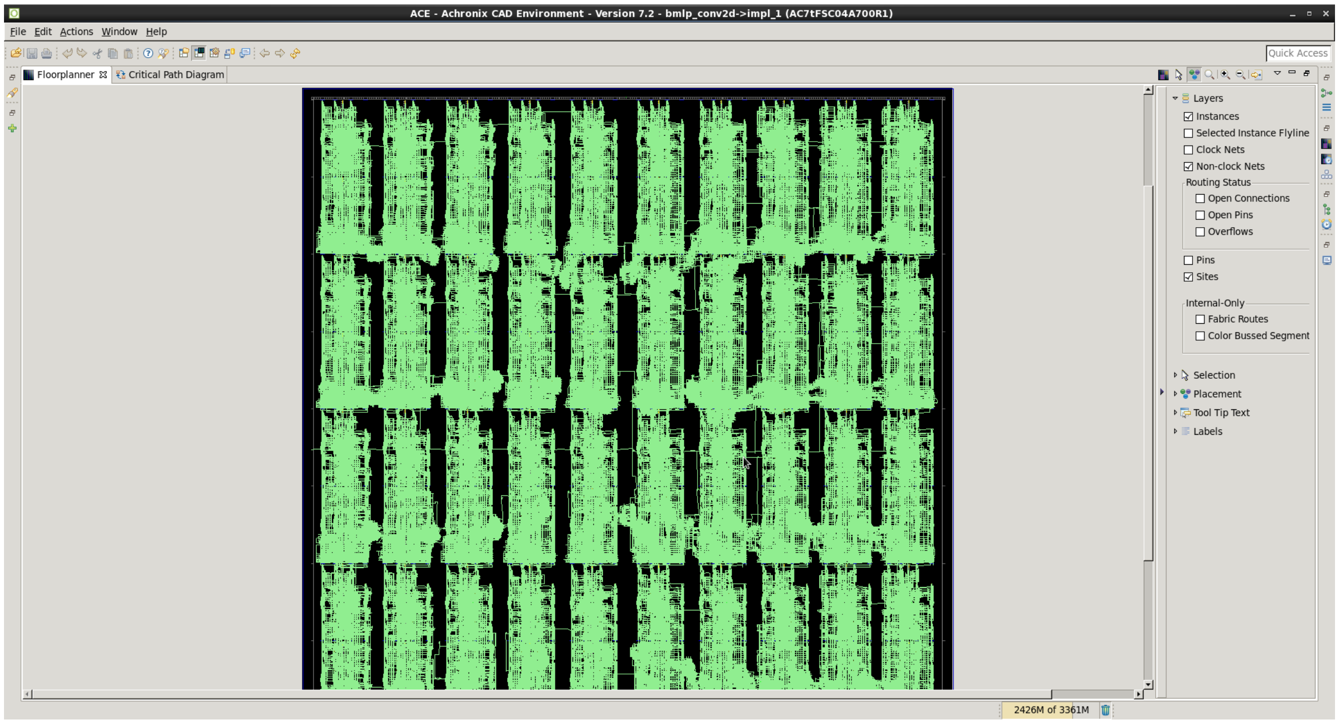

Figure 3: *2400 MLP Floorplan*

Simulation

Simulation supports two different flows. These flows are defined with the `FLOW` variable in the appropriate simulation makefile:

- STANDALONE – For the standalone simulation mode, 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.

- FULLCHIP_BFM – In this mode, the simulation uses the model of the ac7t1500 device complete with NoC and NAPs and behavioral models of the GDDR6 subsystems. The relevant GDDR6 subsystem is initialized, and data is read from it via the NoC using the NAP instantiated within the users design. This user logic NAP is bound in the testbench to a NAP within the model of the ac7t1500. There is a similar process for writing data where data is written to a user logic NAP, which uses the model of the ac7t1500 to transfer data to the model of the GDDR6 subsystem. The testbench performs on-the-fly data comparison for validation.

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`. Copies of both these files are then located in `/src/mem_init_files` where they are accessed by the simulation.

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 2).

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 Table: Input stimulus word format. (see page 3)

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 Simulating the Reference Design (see page)

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.

Split MLP Shared BRAM Stack

Introduction

In the Speedster7t AC7t1500, the MLP and BRAM sites are located side-by-side, with direct connections between them to save routing resources. This layout makes it efficient to store input data for an MLP in the associated BRAM, especially if the same data must be used multiple times. However, if each MLP uses its associated BRAM, then there are no BRAMs left for general storage and buffering. The split_mlp_shared_bram_stack design provides a possible solution: it creates a stack of MLPs with associated BRAMs, but shares each BRAM between two MLPs. This organization leaves (nearly) half a device's BRAMs free to be used for other purposes.

Overview

In this reference design, the MLPs are configured as a number of parallel dot-product (i.e., multiply-accumulate) engines, enabling full utilization of each MLP (16 parallel int8 × int8 multipliers per MLP). The BRAMs and MLPs are stacked and connected as in the figure (see page 10) below, where the number of MLPs can vary from 2 to 64. The screenshot on the right shows the placement in ACE; the yellow dots are additional logic (primarily registers).

A dot-product has the form A × B, where the A vector comes from the single 'A' BRAM at the bottom, with data passed through the stack's cascade connection. The B vector comes from the 'B' BRAMs which are shared among two MLPs each. The unused BRAM sites are free to be used for any BRAM with write_width and read_width ≤ 72. With the dot-product as the basic function, the design can be used to perform different operations. For instance, both matrix multiplication and multi-layer 2D convolution can be mapped to this design.

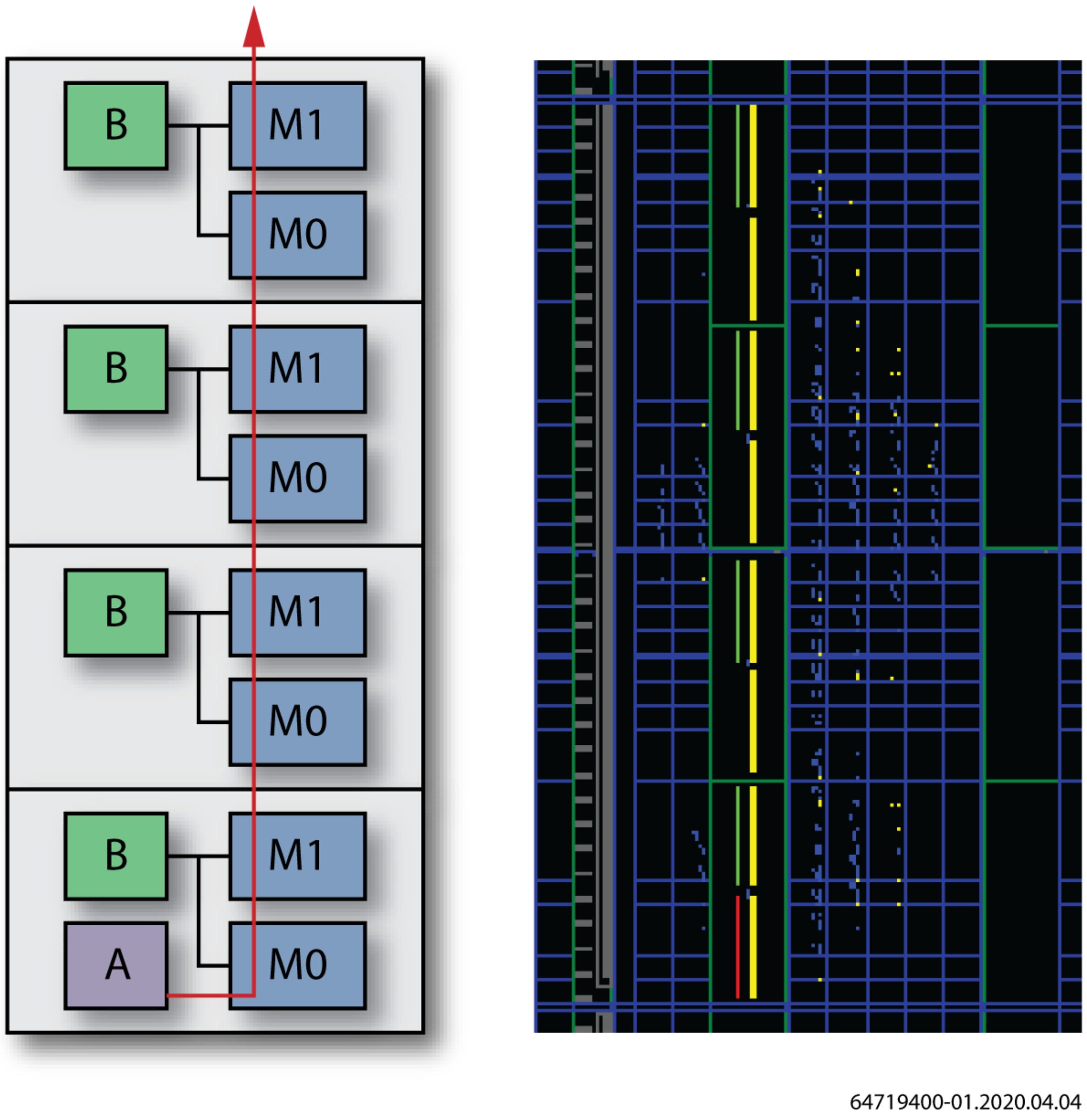

Figure 4: ***High-Level Block Diagram of split_mlp_shared_bram_stack Showing Connections and Placement***

Organization

Each MLP is split into two engines, each with eight parallel int8 × int8 multipliers, and each with its own accumulator. Each engine computes its own dot-product. To supply the eight parallel multipliers, the engine needs 64 bits of A input and 64 bits of B input each cycle. These inputs come from the A and B BRAMs.

The BRAMs are configured with read_width = 128. Each read value is then split into two 64-bit words, "h" for the upper bits, "l" for the lower bits. This way, a sequence of reads from the A BRAM produces two streams of 64-bit data items, Ah and Al. These streams form independent vectors, the only dependency being that they are stored at the same addresses. Likewise, each B BRAM produces streams Bh and Bl.

As shown in the diagram above, the design consists of groups with one B BRAM and two MLPs. Each MLP has two engines, E0 and E1, hence the group has four engines. The data streams, Ah, Al, Bh, and Bl, are distributed as shown in the table below:

Table 6: *Data Stream Distribution*

	Ah	Al	
Bl	M1E0	M1E1	→ result_1
Bh	M0E0	M0E1	→ result_0

While each stream is used by two engines, each engine computes a different dot-product. For example, M0E0 computes SUM(Ah × Bh) over some number of cycles.

When a dot-product operation is finished, the result is stored in an LRAM FIFO, which is internal to each MLP. E0 and E1 operate in parallel, but the E0 result precedes the E1 result in the FIFO. Results are 48 bits each, and the FIFO can store up to 32 results.

Memory Organization

The split_mlp_shared_bram_stack design has parallel write inputs (data and address) for all BRAMs. The BRAMs can be written to at any time, including while the engines are computing. For instance, a designer could choose a "ping-pong buffer" scheme, where half the BRAM stores the current data, while the other half is being written with data for the next computation.

Caution!

It is the user's responsibility to avoid read-write conflicts.

Read addresses are shared: there is one read address for A, `i_bram_a_rdaddr`, and one read address for B, `i_bram_b_rdaddr`. The A data is passed, in a systolic (pipelined) fashion to all MLPs. Meanwhile the B address is passed in the same way to all B BRAMs, so that the A and B data arrive simultaneously at each MLP. Since the user generates both the write and read addresses, the actual memory organization is up to the user. The same data can be used multiple times be re-issuing the same read address. This ability also enables multiple computations, such as matrix multiplication and 2D convolution.

There are two restrictions to the memory organization:

- The B BRAMs all get the same sequence of read addresses; therefore, they must store equivalent data at the same addresses.
- The "h" and "l" data items, which are otherwise independent, must be stored as a single 128-bit word.

The A BRAM has write_width = 64, with a 10-bit `wraddr` (for 1024 64-bit words). The Al data is stored at even addresses; the Ah data at odd addresses. When reading, Ah and Al are read as a single 128-bit wide word.

The B BRAM has write_width = 128, with a 9-bit `wraddr` (for 512 128-bit words). Bit `[63:0]` is Bl, bit `[127:64]` is Bh. The read is 128 bits wide.

Input Sequencing

On each cycle, an engine computes the SUM(A[i] × B[i]) over 8 (A, B) pairs. Often, vectors have more than 8 elements, hence multiple cycles are necessary to compute a full dot-product. The user design must generate the sequence of A and B read addresses for the two vectors, as well as two control signals

- The signal `i_first` must be set high for the first cycle of the dot-product, simultaneously with the first read address. This sequence has the effect of restarting the engine's accumulator.
- The signal `i_last` must be set high for the last cycle of the dot-product, simultaneously with the last read address. This sequence has the effect of storing the accumulated value in the LRAM FIFO.

Each vector must start with `i_first`, even if it immediately follows the previous vector (in which case `i_first` immediately follows `i_last`).

There is a third control signal, `i_pause,` that can be set high to indicate a pause of processing, causing the current read address (and the values of `i_first` and `i_last`) to be ignored. This function is implemented by not updating the accumulator register for one cycle.

The `i_pause` signal can be used if, during processing, there is no input data available, e.g., because it must be fetched from external memory. Since each dot-product is delineated by `i_first` and `i_last`, there is no need to assert `i_pause` between vectors.

The `i_first`, `i_pause`, and `i_last` control signal inputs are always aligned with the corresponding `i_bram_a_rdaddr` and `i_bram_b_rdaddr`; the design includes internal pipelining so that the control signals reach the accumulator at the right time, for example.

As the A data travels up the stack, each MLP causes one cycle of delay. The B data and the control signals are likewise delayed to match the A data. The net effect is that each MLP's computation occurs one cycle later than the computation of the MLP below it. This pipelining is shown by the small rectangles in the figure below.

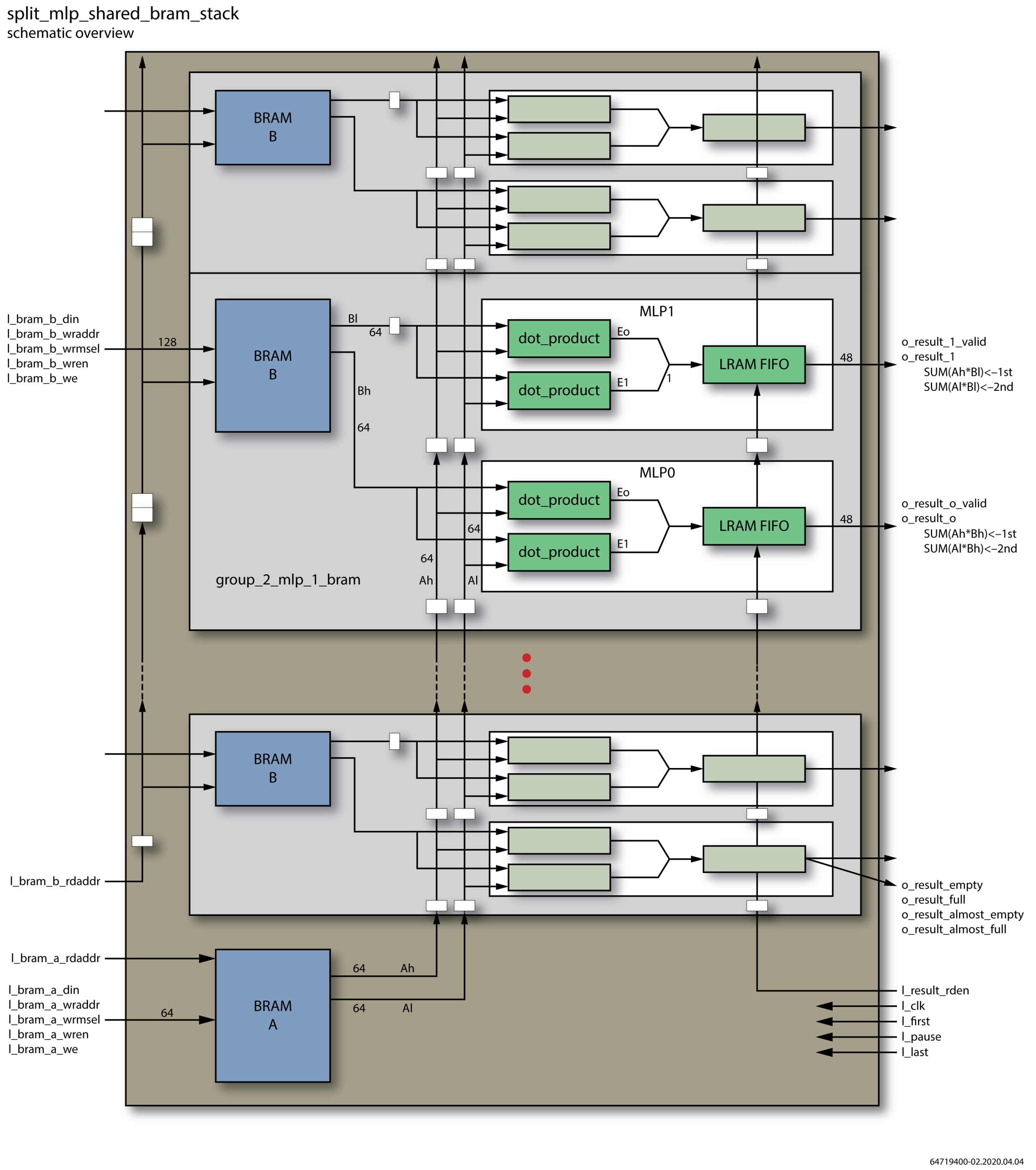

Figure 5: *Detailed Block Diagram of the split_mlp_shared_bram_stack design*

Result Output

Each MLP includes an LRAM FIFO to store the computed results. The FIFO contains alternating E0 and E1 results, starting with E0. Results are 48 bits wide, and each MLP can store up to 32 results. The split_mlp_shared_bram_stack design has parallel data outputs for all result FIFOs. However, there is a single control signal to initiate a FIFO read, `i_result_rden`. This signal is passed up the stack in the same way as input data, with a delay stage between MLPs. In addition to the delay up the stack, the FIFO itself has a cycle of delay. To make reading the parallel output easier, each output has an associated `result_valid` signal that is held high for one cycle when the `result` data is available following a read request.

Because there is only one `rden` signal, there is also only one set of FIFO status signals (empty, full, almost_empty, almost_full), corresponding to the bottom FIFO. The other FIFOs will have the same status when they are reached by the `rden` signal.

Since the results are stored in a FIFO, it is not necessary to read them immediately when they become available. In particular, for maximum performance, the second input vector should start before the result from the first input vector is available. The decision when to read the FIFO is further determined by one important restriction — asserting `i_result_rden` is automatically considered an assertion of `i_pause`, meaning that the current operation is ignored.

Note

This behavior is a result of the underlying implementation, which shares a bus between the FIFO and the B BRAM.

This behavior means that if `i_result_rden` is asserted while a vector is being processed (between `i_first` and `i_last`), the current read address (and control signals) are ignored and must be repeated on the next cycle. It may be easier to avoid this restriction by only asserting `i_result_rden` between vectors (after `i_last` and before `i_first`). Once the request has been issued, processing can immediately continue, even if the actual data has not been received yet from the FIFO. Since an MLP computes two results in parallel, the overhead for reading the results is two cycles per input `(i_first` .. `i_last)` sequence.

Note

This restriction only applies to the read *request*, `i_result_rden`.

Parameters

Table 7: *Split MLP Shared BRAM Stack Parameters*

Parameter	Supported Values	Default	Description
num_groups	1..32	4	Number of groups in the stack, where each group has 2 MLPs.
bram_a_wr_width	64, 72	64	Write width for the "A" BRAM (72 may be useful if the design is converted to block floating point).
bram_a_wraddr_width	1..10	10	Address width for the "A" BRAM. Use 10 to access all 1024 words.

Parameter	Supported Values	Default	Description
bram_b_wr_width	64, 72, 128, 144	128	Write width for the "B" BRAM. Can be changed to 64, which would require twice as many writes (72 and 144 may be useful if the design is converted to block floating point).
bram_b_wraddr_width	1..10	9	Address width for the "B" BRAM. Use 9 to access all 512 words of 128 bits.
mlp_dout_width	2..48	48	Result width. Internal integer results are 48 bit; smaller values cause truncation.
result_afull_threshold	0..15	1	Almost full threshold. See the LRAM2K_FIFO documentation (write_width is 144).
result_aempty_threshold	0..31	2	Almost empty threshold. See the LRAM2K_FIFO documentation (read_width is 72).
bottom_mlp_location	MLP site name	""	Specify an MLP site name to fix the placement of the bottom MLP. This parameter also determines the placement of the other MLPs and the BRAMs.

Ports

Port names starting with i_ indicate input ports; those starting with o_ are output ports. Some ports, such as `i_bram_b_din`, are arrays of vectors, with one array element per group. Where a port, such as `i_bram_b_wren`, would be an array of 1-bit values, it has instead been declared as a vector of the same dimension, as that is sometimes easier in Verilog.

Table 8: ***Split MLP Shared BRAM Stack Ports***

Name	Vector Width	Array Size	Description
i_clk	1		Clock
i_bram_a_din	bram_a_wr_width		Write data for "A" BRAM
i_bram_a_wraddr	bram_a_wraddr_width		Write address for "A" BRAM
i_bram_a_wrmsel	1		Select for write "remap" mode; see BRAM72K_SDP documentation
i_bram_a_wren	1		Write enable for "A" BRAM
i_bram_a_we	bram_a_wr_width/8		Byte enables for write to "A" BRAM
i_bram_b_din	bram_b_wr_width	num_groups	Write data for "B" BRAM, per group
i_bram_b_wraddr	bram_b_wraddr_width	num_groups	Write address for "B" BRAM, per group
i_bram_b_wrmsel	num_groups		Select for write "remap" mode; see BRAM72K_SDP documentation

Name	Vector Width	Array Size	Description
i_bram_b_wren	num_groups		Write enable for "B" BRAM, per group
i_bram_b_we	bram_b_wr_width/8	num_groups	Byte enables for write to "B" BRAM, per group
i_bram_a_rdaddr	9		Read address for "A" BRAM (128-bit words)
i_bram_b_rdaddr	9		Read address for "B" BRAM (128-bit words), passed to each group
i_first	1		Set high to indicate first input for dot-product
i_pause	1		When high, read address and i_first/i_last are ignored
i_last	1		Set high to indicate last input for dot-product
i_result_rden	1		Read one result from the FIFO; passed to each FIFO in pipelined fashion
i_result_rstn	1		Resets the FIFO pointers, discarding any stored values. At power-up, the FIFO is already in its reset state, but this signal can be used to interrupt operation later. To ensure flushing of pending operations, i_result_rstn should be asserted for 7 cycles.
o_result_empty	1		Indicates the FIFO is empty and cannot be read
o_result_full	1		Indicates the FIFO is full, and must be read before new results are written
o_result_almost_empty	1		High if the FIFO has ≤ result_aempty_threshold values
o_result_almost_full	1		High if the FIFO has ≤ result_afull_threshold empty slots
o_result_1	mlp_dout_width	num_groups	Output of the FIFO of MLP1 of each group
o_result_1_valid	num_groups		High when o_result_1 is valid (per group), following i_result_rden
o_result_0	mlp_dout_width	num_groups	Output of the FIFO of MLP0 of each group
o_result_0_valid	num_groups		High when o_result_0 is valid (per group), following i_result_rden

Example Design

The user design is split_mlp_shared_bram_stack. However, to make a design that can be run through ACE, and for easier simulation, a simple wrapper has been added around the design. All inputs and outputs are registered to enable ACE to report timing for them. To reduce the number of I/O pins, the parallel inputs and outputs have been reduced:

- Instead of the parallel write inputs for the B BRAMs, there is a single input that fans out to all B BRAMs with an index (`i_bram_b_group`) to indicate which BRAM should be written.
- The parallel result outputs feed into a pipeline to output the results in a single sequence.

Note

Both of these methods were only used to simplify the test design: they are not recommended for real designs. In particular, serializing the results means that more time is spent collecting results than computing them. The design should be run with "evaluation" flow_mode, so that ACE automatically places all I/O pins.

As explained above, the user chooses the memory organization, as well as the sequence of read addresses, meaning the user controls the actual function that is computed. In the provided testbench, a matrix is written to the A BRAM, and another matrix is distributed over the B BRAMs. Then a sequence of read addresses is generated that computes the product of the two matrices. The testbench verifies that the computed results are correct.

For the purposes of demonstration, results are being read while the computation is ongoing. As explained in Result Output (see page 14), reading during computation causes an automatic pause state which may require the testbench to issue the same read address twice. As mentioned, in practice it may be easier to read results in between processing steps.

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:

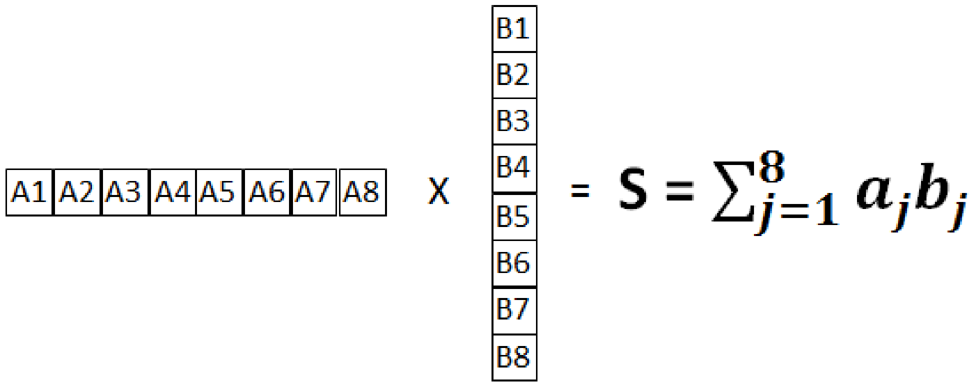

Figure 6: *Dot Product Operation*

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 `i_a` and `i_b` inputs can be considered interchangeable, as the final result is the sum of each vector multiplied together.

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 9: *dot_product_N_8x8 Configuration*

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

Ports

Table 10: *dot_product_N_8x8 Ports*

Port	Direction	Description
i_clk	Input	Clock input
i_a	Input	"a" input to multiplication. Array of N × int8 (N × 8 bits).
i_b	Input	"b" input to multiplication. Array of N × int8 (N × 8 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.

Port	Direction	Description
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

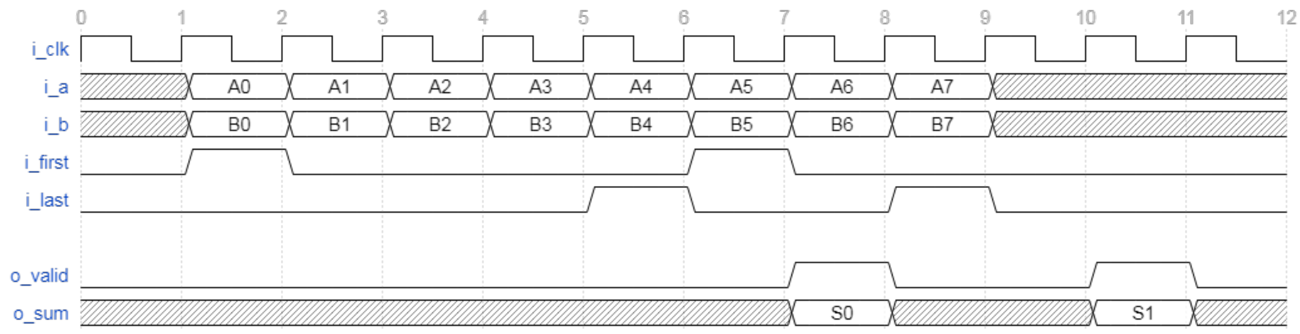

Figure 7: ***dot_product_N_8x8 timing diagram***

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_bfloat16_4mlp

Description

This design demonstrates the dot product of a number of "brain float 16", (`bfloat16`) inputs (see below for an explanation of the difference between brain float and block floating point). The design consists of four MLP72s 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 `fp16e8` values). The sum from each MLP72 is cascaded up the column of MLP72 to the next block above. On each cycle, the sum of the eight parallel `fp16e8` multiplications is calculated In the last MLP72.

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.`

Brain Float versus Block Floating Point

The term `bfloat16` stands for "brain float" and is a 16-bit format introduced by the Tensorflow package. It has 16 bits: 1 sign bit, 8 exponent bits, and 7 mantissa bits. The precision is 8 bits, composed of 7 mantissa bits plus the "hidden 1". (All mantissa's are assumed to start with a '1'.)

In addition Achronix also supports a computational method known as "block floating point". Block floating point (`blockfp`) and `bfloat16` are not related in any way. To avoid confusion, Achronix uses the term `fp16e8` instead of `bfloat16` when referring to the number formats.

Configuration

Table 11: ***dot_product_bfloat16_4mlp***

Input Format	Output Format	Parallel Multiplications	Number of MLP72
fp16e8	fp16e8	8	4

Latency

Output latency is measured from the clock cycle on which `i_last` is asserted. The output is then valid 12 cycles after the cycle `i_last` was asserted on. The output is validated with the `o_valid` signal.

Ports

Table 12: ***dot_product_bfloat16_4mlp Ports***

Port	Direction	Description
i_clk	Input	Clock input
i_a	Input	"a" input to multiplication. Array of 4 × 2 × fp16_e8 (128 bits).
i_b	Input	"b" input to multiplication. Array of 4 × 2 × fp16e8 (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	fp16e8 accumulated output. Output latency is 12 cycles from `i_last`being asserted
o_valid	Output	Validates o_sum output..

Timing Diagram

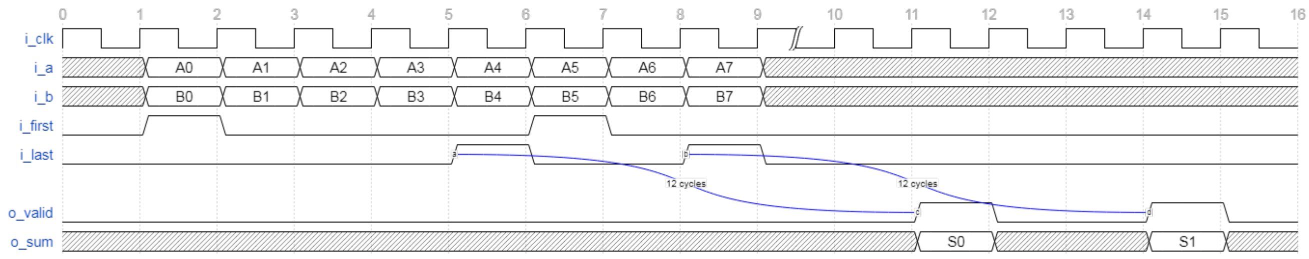

Figure 8: ***dot_product_bfloat16_4mlp Timing Diagram***

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_fp16_4mlp

Description

This design demonstrates the dot product of a series of 16-bit floating values. The design consists of four MLP72s, 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 `fp16` values). The sum from each MLP72 is cascaded up the column of MLP72s to the next block above. On each cycle, the sum of the eight parallel `fp16` multiplications is calculated in the last MLP72.

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.`

Note

fp16 format is the same as IEEE-754 "binary16" format, and is often referred to as "half-precision". It consists of 16 bits: 1 sign bit, 5 exponent bits, and 10 mantissa bits. The precision is 11 bits, 10 mantissa bits and the "hidden 1", (all mantissa's are assumed to start with '1').

Within the Achronix floating point library, this format is also referred to as `fp16e5`.

Configuration

Table 13: ***dot_product_fp16_4mlp***

Input Format	Output Format	Parallel Multiplications	Number of MLP72
fp16e5	fp16e5	8	4

Latency

Output latency is measured from the clock cycle on which `i_last` is asserted. The output is then valid 12 cycles after the cycle `i_last` was asserted on. The output is validated with the `o_valid` signal.

Ports

Table 14: *dot_product_fp16_4mlp Ports*

Port	Direction	Description
i_clk	Input	Clock input
i_a	Input	"a" input to multiplication. Array of 4 × 2 × fp16_e5 (128 bits).
i_b	Input	"b" input to multiplication. Array of 4 × 2 × fp16e5 (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	fp16e5 accumulated output. Output latency is 12 cycles from `i_last` being asserted.
o_valid	Output	Validates o_sum output..

Timing Diagram

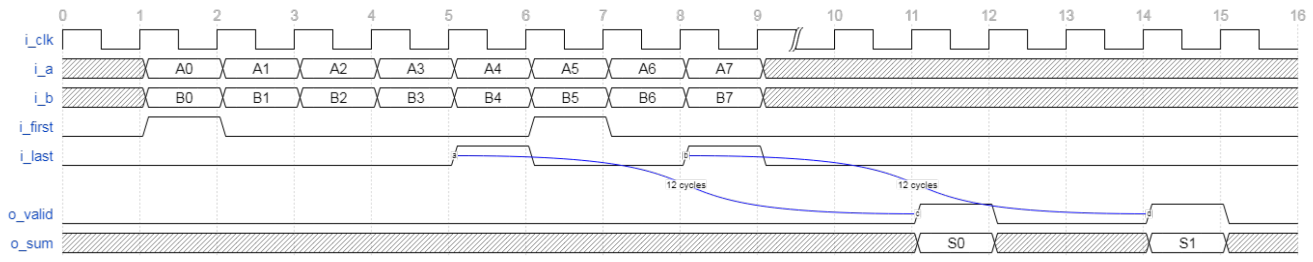

Figure 9: *dot_product_fp16_4mlp Timing Diagram*

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] + ...

Builds

All the dot product reference designs make use of a common build environment. The builds are all targeted for the AC7t1500 device. For full details of this build environment, refer to Building the Reference Design (see page 51).

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 VARIANT parameter in the common `/build/Makefile` 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 700 MHz.

Simulation

All of the dot product reference designs have a common simulation methodology, with the same structure of simulation scripts, as detailed in Simulating the Reference Design (see page 46).

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.

Matrix Vector Multiply

The matrix vector multiply (MVM) design, mvm_8mlp_16int8_earlyout, multiplies a 256 × 256 matrix with a 256 × 1 vector, using int8 input values. The design uses 8 MLP72 and 8 BRAM72K instances. The 256 × 256 matrix is stored in the BRAMs, and can be used multiple times. No extra storage is needed for the 256 × 1 input vector: it is stored in the LRAMs that are closely coupled to the MLP72.

Configuration

Overview

The design consists of a stack of 8 BRAMs and 8 MLPs. Input of data is at the bottom of the stack; the output of results is at the top. The design uses both the internal cascade connections and the direct connection between BRAMs and MLPs. The use of these internal connections greatly reduces the use of fabric routing resources, and simplifies timing as well.

- BRAM write data (the weight matrix), write address, and read address are distributed through the BRAM cascade connection
- Likewise, the vector data is distributed to the LRAMs via the MLP input cascade (the LRAMs are physically contained within the MLPs)
- Data from the BRAMs is passed to the MLPs via their direct connection
- The MLP output cascade is used to collect and accumulate the output of each stage

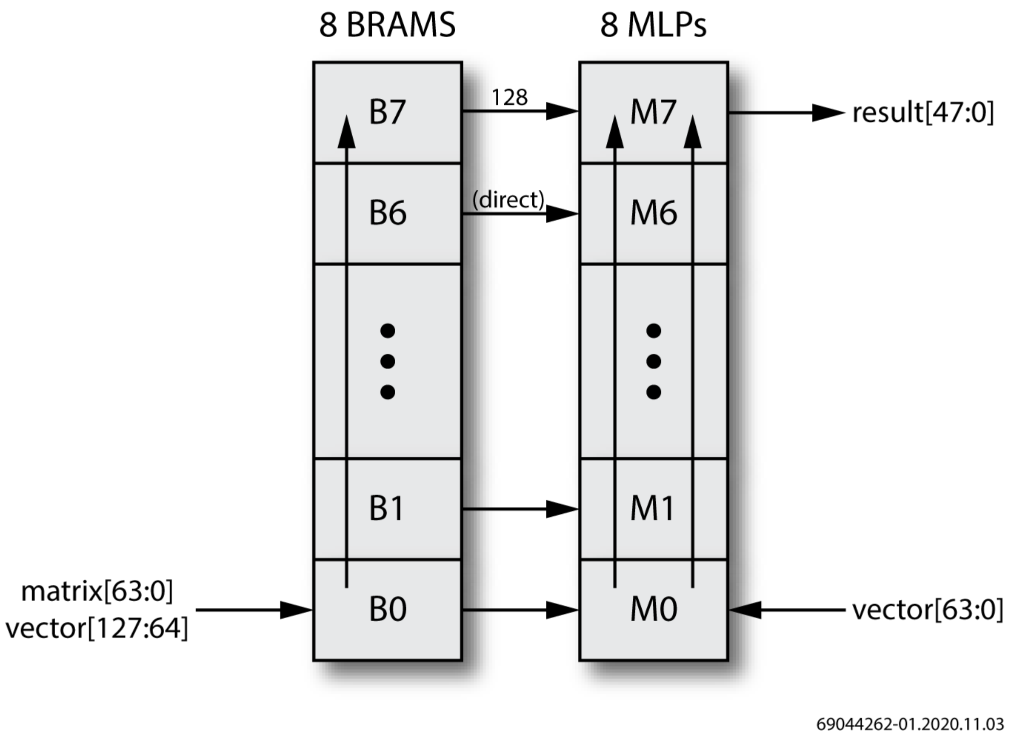

Figure 10: ***BRAM and MLP interconnections***

MLP

Each MLP uses 16 int8 × int8 multipliers, which are summed using the full integer adder tree. The latency through each MLP is set to three cycles.

The 'a' input of the MLP, representing a row of the matrix, is the output of the adjacent BRAM. The 'b' input, representing the vector, is the output of the (internal) LRAM. The vector and the matrix rows each have 256 elements, which are divided in 16 blocks of 16 elements each. Per cycle, an MLP computes the product of a vector block and a row block (using the parallel multipliers and adder tree). To compute a full vector × row dot-product, 16 such products must be added. In this design, that addition is distributed over the MLPs, as illustrated in the block diagram (see page 25) below. In this diagram, `v[i]` and `r[i]` refer to blocks of the vector and current row, respectively. As the result passes up through the MLP stack, 8 products are added. Since 16 products must be added, the extra accumulator in the top MLP is used to add two such sums, producing one result value every other cycle.

Looking at the bottom MLP, once the two products v0 × r0 and v8 × r8 have been computed, the next cycle again computes v0 × r0, but now r0 is a block of the next matrix row. The figure below shows that the bottom MLP's LRAM only needs to store two blocks of the vector, v0 and v8 (32 values total). The same is true for the other MLPs, and each needs to store just two blocks of values. The BRAM is organized so that the row blocks line up with the LRAM.

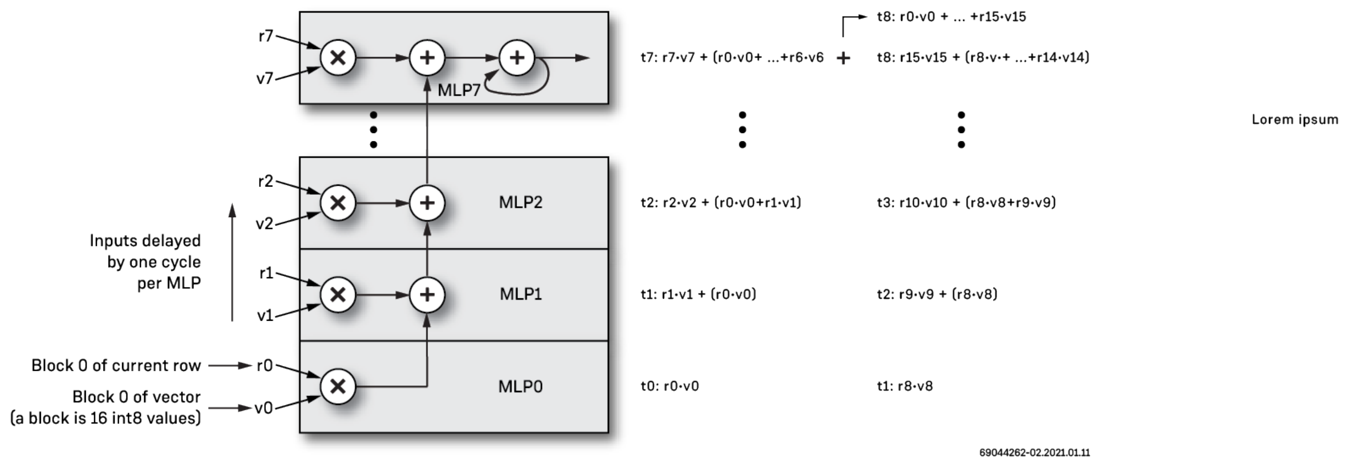
r7
v7
MLP7
r2
v2
MLP2
r1
v1
MLP1
r0
v0
MLP0
Inputs delayed by one cycle per MLP
Block 0 of current row
Block 0 of vector (a block is 16 int8 values)
t8: r0·v0 + ... +r15·v15
t7: r7·v7 + (r0·v0+ ...+r6·v6
t8: r15·v15 + (r8·v·+ ...+r14·v14)
Lorem ipsum
t2: r2·v2 + (r0·v0+r1·v1)
t3: r10·v10 + (r8·v8+r9·v9)
t1: r1·v1 + (r0·v0)
t2: r9·v9 + (r8·v8)
t0: r0·v0
t1: r8·v8
69044262-02.2021.01.11

Figure 11: *Matrix Vector Multiply Block Diagram*

Calculation Flow

Referring to the block diagram (see page 25) above, at the output of the first adder within MLP7, the first computed value is:

`r0 × v0 + r1 × v1 + ... + r7 × v7`

On the following cycle, the value is:

`r8 × v8 + r9 × v9 + ... + r15v15`

Therefore the accumulator function in MLP7 adds these two values, to give a result every two cycles of:

`r × v = r0 × v0 + ... + r15 × v15`

LRAM

The LRAM takes 128-bit wide input from the `fwdi_multa_h/l` cascade, and from the fabric for `LRAM0`. Each LRAM has its own address counter in the fabric. Since only two blocks per LRAM are stored, the address just toggles between 0 and 1. The LRAM output register is disabled (the LRAM read is combinational). Instead, the LRAM output drives the MLP stage1 register.

BRAM

The BRAMs are configured for deep writes, and wide reads: `wrdata`, `wraddr`, `wren`, and `rdaddr` all come in from the fabric to BRAM0, then are passed along the forward (`fwdi`) cascades. Read data goes directly to the MLP.

The matrix is input to the macro in regular row-major order, but internally the write address is computed to distribute the data over the BRAMs to match the distribution of the vector in the LRAMs. Reordering the blocks as they come in is easily achieved in deep memory mode (using the block address). If it is desired to write all BRAMs in parallel instead (which would be much faster), reordering on the fly will be challenging and may have to be done in advance in the off-chip memory.

Interface

Loading the Matrix

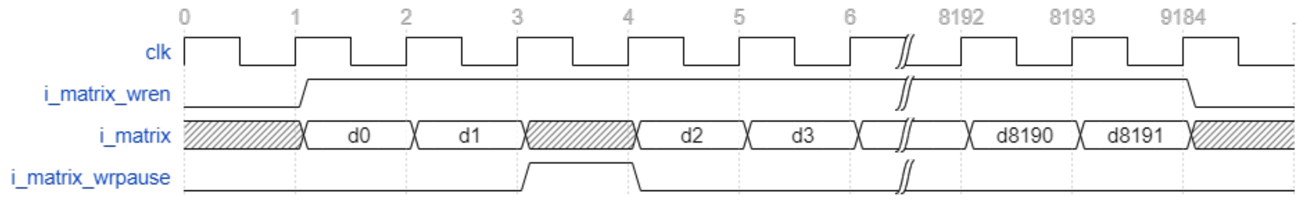

Figure 12: ***Loading the Matix Timing Diagram***

Matrix data input is signaled by `i_matrix_wren`. All values should be input in row-major order, 8 values per cycle. Hence, writing 256 × 256 values takes 8192 cycles. The signal `i_matrix_wrpause` can be asserted to pause writing temporarily, when no data is available. When `i_matrix_wrpause` is high, `i_matrix` is ignored.

Note

The signal `i_matrix_wren` must remain high until the entire matrix has been written.

If `i_matrix_wren` is de-asserted, (= 1'b0), then `i_matrix_wrpause` is ignored. This feature allows `i_matrix_wrpause` to be shared with other modules — `i_matrix_wren` then acts as a module select.

Performing Matrix-Vector Multiply

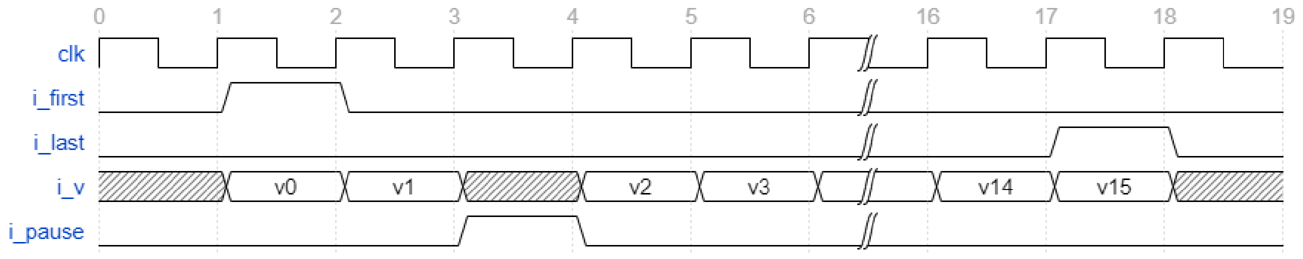

Figure 13: ***Matrix-Vector Input Timing Diagram***

The vector must be input in consecutive blocks of 16 values each (16 blocks). The first block must be marked by `i_first`, the last block by `i_last`. If data it temporarily unavailable, `i_pause` can be asserted to pause writing. When `i_pause` is asserted, `i_v` is ignored. The actual matrix-vector multiplication starts automatically following `i_last`.

Note

The signal `i_pause` must only be asserted between `i_first` and `i_last` (not simultaneously with `i_first` or `i_last`).

Result Output

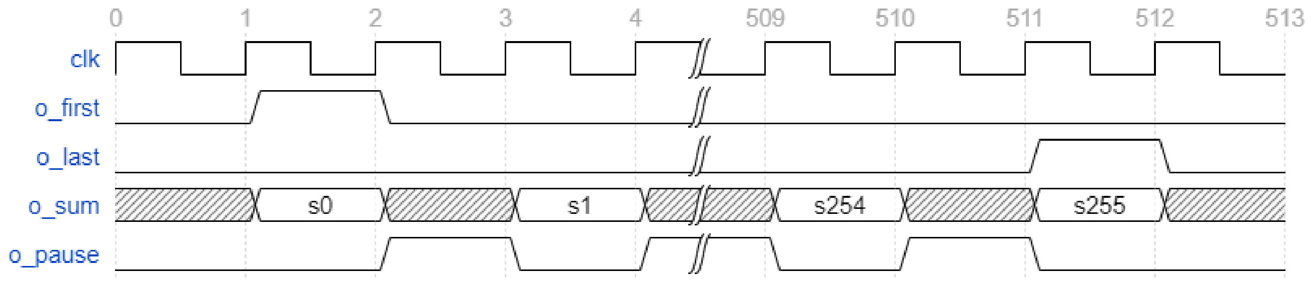

Figure 14: *Matrix-Vector Output Timing Diagram*

There are 256 results, s0..s255. The protocol is the same as for the vector input — the first `o_sum` output occurs when `o_first` is high, the last `o_sum` output occurs when `o_last` is high. In between, `o_sum` is valid when `o_pause` is low. The next input vector (indicated by `i_first`) can start after `o_last`.

Performance

The matrix has 256 × 256 = 64k int8 values. When configured as 64 bits wide, each BRAM is 8k deep, hence the matrix exactly fills the 8 BRAMs. The BRAMs are written 64 bits per cycle, hence writing the entire matrix takes 64k/8 = 8192 cycles. Alternatively, the design could easily be modified to write the BRAM 128 bits at a time, reducing the total write time by half. It is assumed that the BRAM contents are used multiple times; if this assumption is not valid, then it may make sense to write tothe BRAMs in parallel instead, reducing the write time by a factor 8.

The vector has 256 values, and is written 16 values per cycle, requiring 256/16 = 16 cycles. A result is produced every two cycles, hence it takes 512 cycles to produce the 256-value result vector.

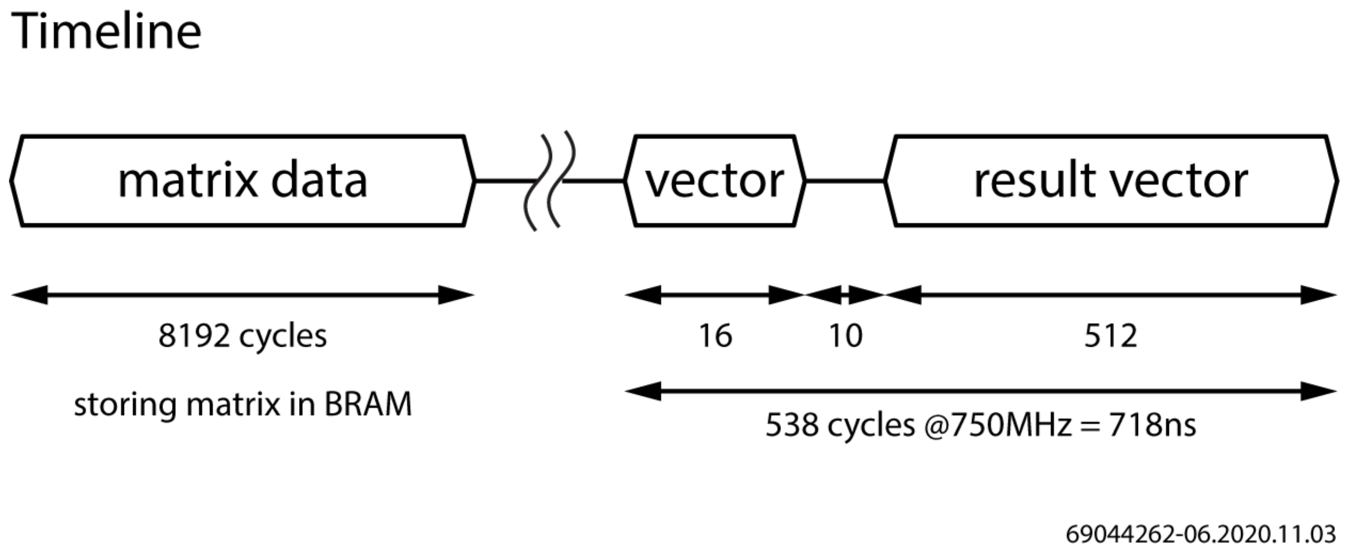

Figure 15: *Matrix-Vector Overall Timing Diagram*

There are a few cycles between the input of the vector and the first output. As a result, the total time to input a vector and compute the product is 538 cycles:

Theoretical Performance of MLP

16 multiplications and 16 additions per cycle:

32 × 8 MLPs × 538 cycles = 137728 operations

Mathematically

A row × vector dot-product has 256 multiplications and 255 adds, hence:

256 × (256 + 255) = 130816 operations

Operational Efficiency

130816/137728 = 95%

130816/538 = 243 effective operations/cycle

@750 MHz = 243 × 750 = 182 effective rate in giga operations per second (GOPS) for the mvm design

The AC7t1500 Speedster7t device has 2560 MLPs. If all are used for instances of the mvm design, the performance is:

182 × 2560/8 = 58 TOPS

Note

With minor modifications to the limits used by cycle_count, the exact same circuit could use a quarter-sized matrix (128 × 128) in a quarter of the time. Other sizes can be mapped to the same design, but not as efficiently — time is saved by computing fewer results, but the MLPs are used less efficiently because the blocks do not distribute equally over the MLPs, or the rows do not divide equally into blocks. In those cases, a similar configuration with a different number of BRAMs/MLPs may work better.

Internal Control Signals

The diagrams below show the timing of the internal control signals for reading and writing the vector data. In particular, the `cycle_count` register is used to determine when `o_first` and `o_last` must be raised, and when the computation is finished.

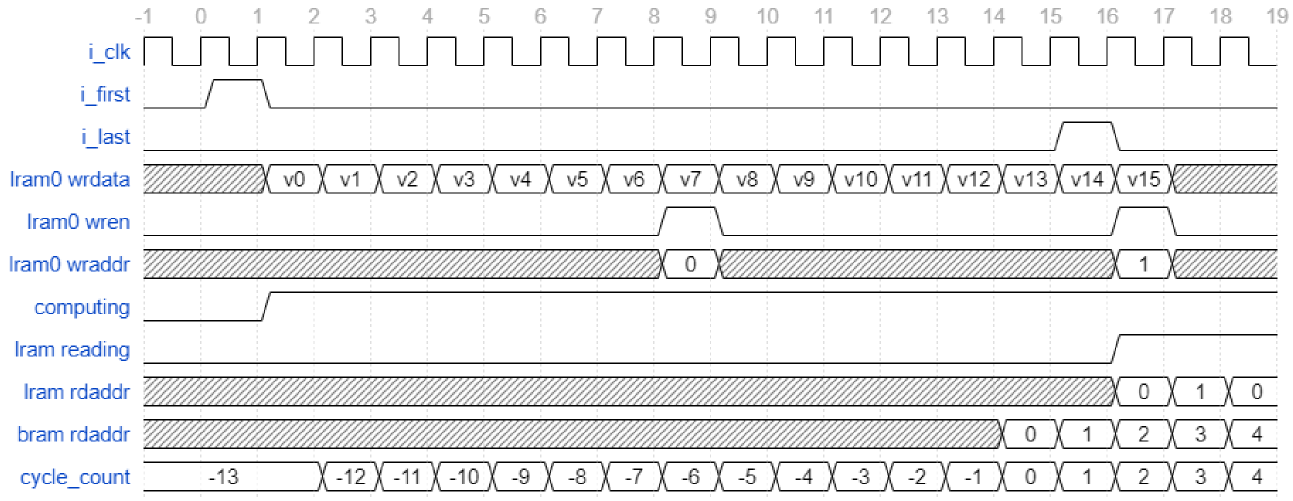

Figure 16: *Matrix-Vector Internal Write Control Signals Timing Diagram*

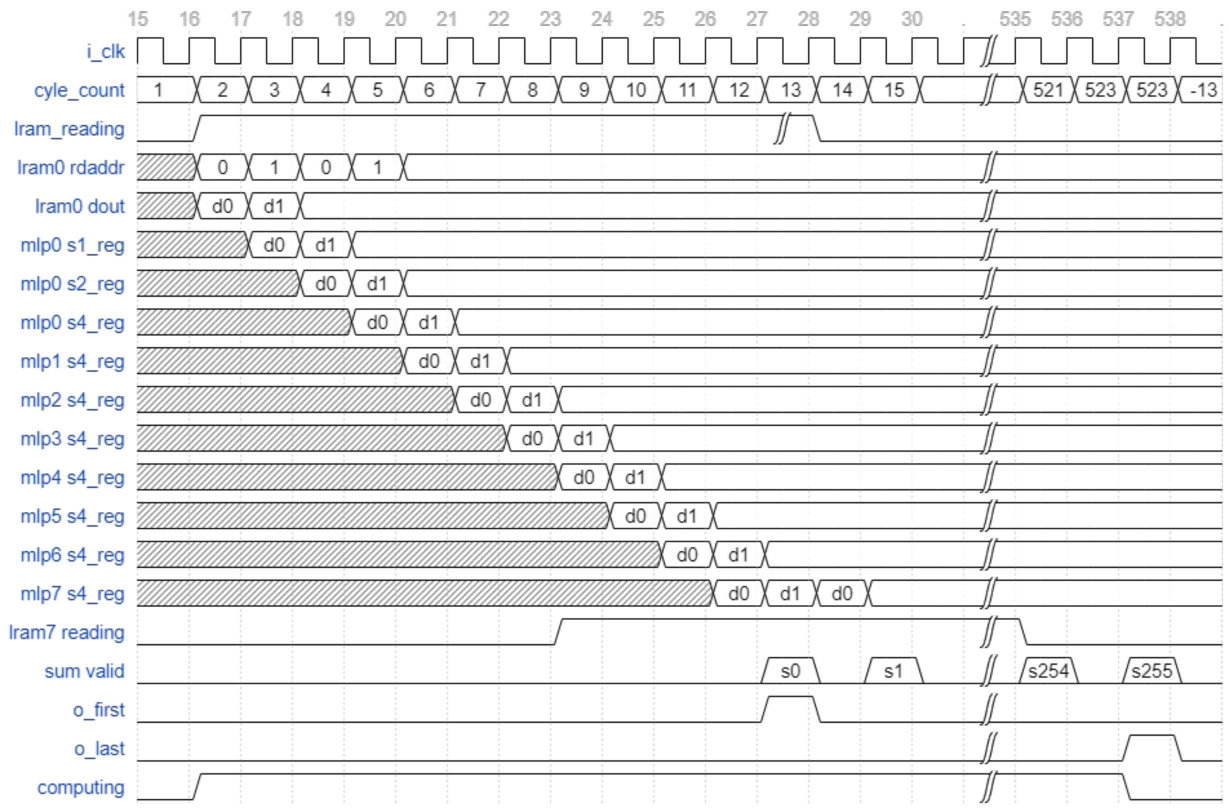

Figure 17: *Matrix-Vector Internal Read Control Signals Timing Diagram*

Instructions

Installing the Reference Design

Downloading

The design is available for download on the Achronix self-service FTP site (https://secure.achronix.com), located in the folder `/public/Achronix/Reference_Designs/Speedster7t`.

Packaging

The design is 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.

In addition the root of the archive contains release notes identifying the change history of the design.

Operating System

Linux

The design scripts and build flow are natively designed for Linux and have been built and tested using CentOS 7 and Ubuntu 16.04LTS. The flows use Makefiles, so are Linux shell agnostic.

Windows

To enable the scripted simulation or implementation flows to operate under Windows 10, the user must install one of the following;

- A Linux environment installed under Windows, such as www.cygwin.com The installation should include a Tcl interpreter and the `make` executable.
- A Tcl Interpreter and a `make` executable for Windows. There are many variants available, including both no cost and licensed versions.

If the user is unable to install any of the options above, it is still possible to run some of the flows under Windows:

- Implementation – GUI projects are provided for both Synplify and ACE, enabling builds to be done using the tools directly in GUI mode.
- Simulation – A Tcl script is provided which can be executed in the QuestaSim Tcl console. This script enables simulation using QuestaSim under Windows. For further details, refer to the Simulation section.

Note

For correct operation of any script flow, ACE should be installed in a directory without spaces in the path name, e.g., `C:\achronix\8.3\Achronix_CAD_Environment`. Additionally the environment variable `ACE_INSTALL_DIR` must use "/" as path separators rather than "\", for example:

```
ACE_INSTALL_DIR = C:/achronix/8.3/Achronix_CAD_Environment/Achronix
```

Tool Versions

All designs were tested with the following tools:

Table 15: *Minimum Tool Versions*

Software	Version
ACE	8.3
Device Simulation Model	8.2.1
Synplify Pro	Q-2020.03X
Mentor Questa	10.7c-1
Synopsys VCS	O-2018.09-SP1-1

Environment Variables

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 point 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.

Note

For correct operation of any script flow, ACE must be installed in a directory without spaces in the path name. Examples of suitable paths are:

- Windows – `C:\achronix\8.3\Achronix_CAD_Environment`
- Linux – `/opt/achronix/ace/8.3`

Additionally when installed under Windows, the environment variable `ACE_INSTALL_DIR` must use "/" as path separators rather than "\", for example:

```
ACE_INSTALL_DIR = C:/achronix/8.3/Achronix_CAD_Environment/Achronix
```

Directory Structure

The design has a directory structure that allows for easy navigation and separation of source and generated files. The directory structure can be easily modified to suit a users preferred layout; however, if the structure is modified, then the necessary makefiles and build scripts will need to be modified to suit. To support portability, relative paths are used, as opposed to absolute paths; with the use of environment variables to select the root directory.

Table 16: ***Design Directory Structure***

Directory			Decription
<design_name>			Root directory. Contains release notes.
	/build		Synthesis and place-and-route building
	/doc		Documentation and user guide
	/scripts		Scripts used for building and simulation
	/sim		Simulation area
		/vcs	Synopsys VCS simulation files
		/questa	Mentor QuestaSim simulation files
	/src		Source code
		/ace	ACE GUI project
		/acxip	ACE .acxip configuration files

Directory			Decription
		/constraints	Placement and timing constraint files
		/include	RTL include files
		/ioring	ACE generated ioring files
		/rtl	RTL source files
		/syn	Synplify Pro GUI project
		/tb	RTL testbench files
		filelist.tcl	Filelist used for building and simulation

Language Support

The reference designs support both Verilog, SystemVerilog and VHDL RTL languages. These can be used for both building, and standalone simulation. If the full-chip BFM simulation is used, that environment requires that the top-level testbench is Verilog or SystemVerilog. However, the design under test (DUT) may be written in VHDL.

Constraint Files

The design is supplied with a full set of constraint files, located under `/src/constraints`. These files demonstrate how various constraints and directives may be applied to the design. The constraint files, and their usage is detailed below.

Table 17: *Constraint File Details*

File Name	Usage
`ace_constraints.sdc`	Timing constraints used by ACE. More than one SDC file can be included in an ACE project.
`ace_options.sdc`	Control ACE settings, such as flow mode, speed grade, reporting of unconstrained paths.
`ace_placements.pdc`	Fix locations of elements within the ACE fabric, and creation of placement regions. **Note** Pin placements between the fabric and the I/O ring are automatically created by the I/O ring designer, and provided in the ioring PDC files.
`synplify_constraints.fdc`	Synplify FPGA design constraints. Set attributes such as compile points, or default memory styles.
`synplify_constraints.sdc`	Synplify timing constraints. Clock and timing constraints. Should match those set in `ace_constraints.sdc`.

File Name	Usage
`synplify_options.tcl`	Control Synplify settings, such as top module. Create synthesis specific parameters, generics and defines.

I/O Ring Constraint Files

In addition to the constraint files listed above, the I/O ring generates constraint files specific to the interface between the fabric core and the I/O ring containing the interface subsystems. These constraint files can be auto-generated by ACE from the respective `.acxip` files; however, to aid the build flow, pre-generated files are provided for projects that require configuration of the I/O ring interface subsystems. These files are located under `/src/ioring`. The purpose of each file is detailed below.

Table 18: *IO Ring Constraint File Details*

File Name	Usage
`<design_name>_ioring.sdc`	I/O timing constraints for direct connection interfaces, between the fabric and the I/O ring.
`<design_name>_ioring.pdc`	Placement of the fabric I/O pins, to assign them to the direct connection interfaces in the I/O ring.
`<design_name>_ioring_util.xml`	Used by ACE to generate a combined utilization report, combining the fabric and I/O ring resources.

Device Simulation Model

Many designs require a simulation overlay named the Device Simulation Model, (DSM). This package combines the full RTL of the network on chip (NoC) with bus functional models (BFMs) of the interface subsystems that surround the NoC and FPGA fabric. This combination of true RTL for the NoC and models for the interface subsystems allows users to develop their designs within a fast responsive simulation environment, while achieving cycle-accurate interfaces from the NoC, and representative cycle responses from the hard interface subsystems. This simulation environment allows a designer to iterate rapidly to develop and debug their design.

Description

The DSM provides full RTL code for the NoC and BFM models of the surrounding interface subsystems. This structure is wrapped within a SystemVerilog module named per device, i.e., ac7t1500. The user needs to instantiate one instance of this module within their top-level testbench.

In addition, the DSM provides binding macros such that the user can bind between elements of their design and the same elements within the device. For example, the design may instantiate a NoC access point (NAP). It is then necessary to bind this NAP instance to the NAP in the correct location within the NoC by using the `ACX_BIND_NAP_SLAVE, `ACX_BIND_NAP_MASTER, `ACX_BIND_NAP_HORIZONTAL, or `ACX_BIND_NAP_VERTICAL macro, whichever is appropriate for the design.

Similarly it is necessary to bind between the ports on the design and the direct-connection interface (DCI) for the interface subsystem. Each DCI within the device is connected to a SystemVerilog interface. This interface can then be directly accessed from the top-level testbench, and signals assigned between the SystemVerilog interface and the ports on the design.

Version Control

The DSM is version controlled. Within a release, new functions may be added and older functions may be deprecated or replaced. The release is indicated both in the package name (`ACE_<major>.<minor>.<patch>_IO_BFM_sim_<update>.zip/tgz`) and in the `readme` file placed in the root directory of the package.

To ensure that the correct version of the I/O ring simulation package is used, a task must be included within the design testbench to confirm the version compatibility. This function should be instantiated as detailed below:

Code

```
// For this example the DSM instance is ac7t1500
initial begin
    // Ensure correct version of DSM is being used
    // This design requires 8.3.1 as a minimum
    ac7t1500.require_version(8, 3, 1, 0);
end
```

require_version() Task

The require_version task has four arguments. In order

- Major Version – Will match the major version of the release
- Minor Version – Will match the minor version of the release
- Patch – Will match the patch version of the release (optional)
- Update – Will match the update number of the release (optional)

If either of patch or update is not specified, then these should be set to 0; for example, for the 8.3 release, the arguments would be set as 8,3,0,0.

Note

The values can be expressed either a numbers (0-9) or as strings ("0" – "9") or as letters ("a/A", "b/B"), with the letters "a" and "b" represent alpha or beta releases. When deciding on the priority of a release, a number represents a more recent release than a letter; therefore, 8.3.alpha (defined as as 8,3,"a",0) will precede the full 8.3 release (designated as 8,3,0,0).

Example Design

An example structure of a users testbench, instantiating both the DSM and the users design under test is shown in the diagram (see page 35) below. This example shows the macros required for the slave NAPs, and the DCIs for two instances of the GDDR6 subsystem. For other forms of NAPs, or for other DCI types, such as DDR, consult the Bind Macros (see page 48) and Direct-Connection Interfaces (see page 39) tables.

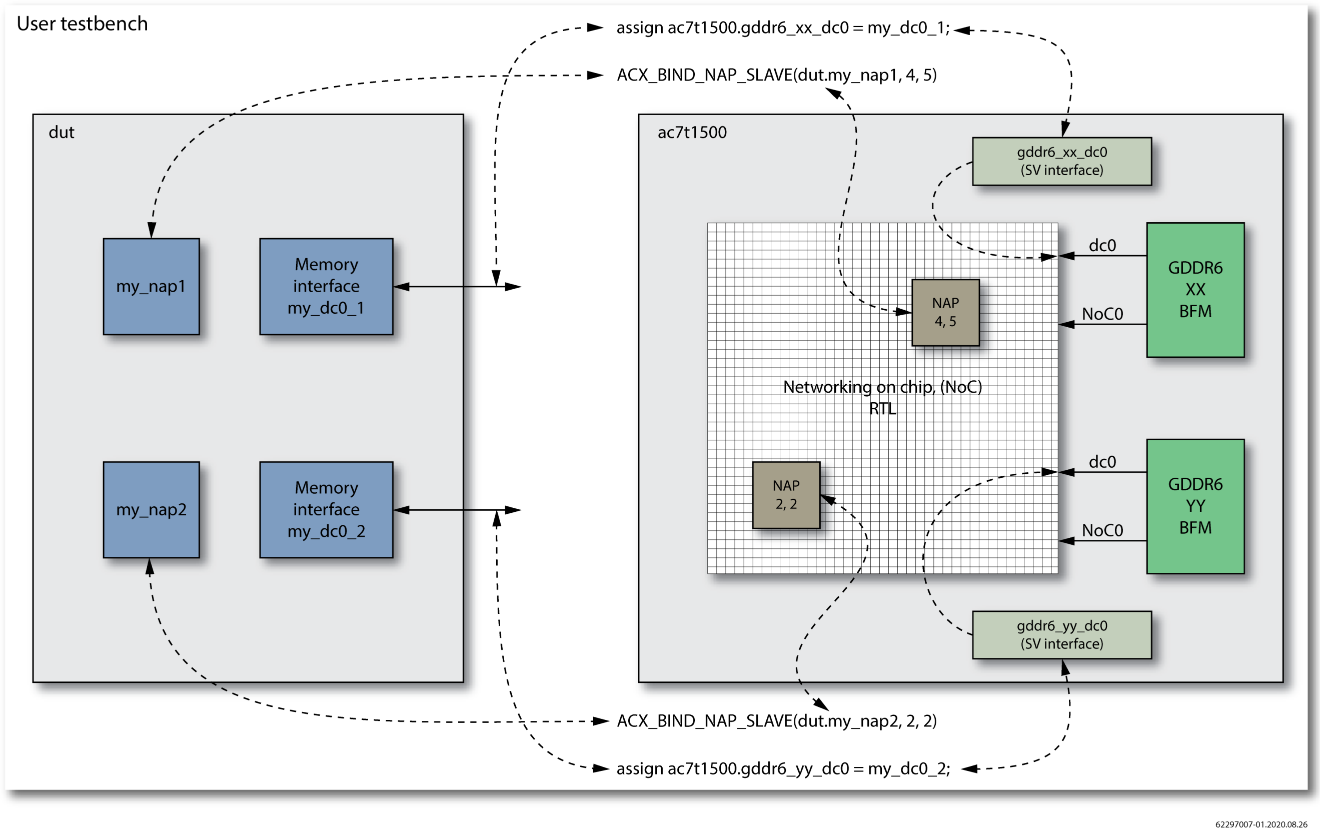
User testbench
assign ac7t1500.gddr6_xx_dc0 = my_dc0_1;
ACX_BIND_NAP_SLAVE(dut.my_nap1, 4, 5)
dut
my_nap1
Memory interface my_dc0_1
my_nap2
Memory interface my_dc0_2
ac7t1500
gddr6_xx_dc0 (SV interface)
dc0
NoC0
GDDR6 XX BFM
NAP 4, 5
Networking on chip, (NoC) RTL
NAP 2, 2
dc0
NoC0
GDDR6 YY BFM
gddr6_yy_dc0 (SV interface)
ACX_BIND_NAP_SLAVE(dut.my_nap2, 2, 2)
assign ac7t1500.gddr6_yy_dc0 = my_dc0_2;
62297007-01.2020.08.26

Figure 18: *Example I/O Ring Simulation Structure*

In the example above, there are two NAPs, `my_nap1` and `my_nap2`. In addition there are two direct-connect interfaces, `my_dc0_1` and `my_dc0_2`. In the top-level testbench bindings are made between the NAPs in the design and the NAPs within the device using the ACX_BIND_NAP_SLAVE macro. This macro supports inserting the coordinates of the NAP within the NoC in order that the simulation is aligned with physical placement of the NAP on silicon.

The DCIs are ports on the user design; these ports are then assigned to the appropriate signals within the device direct-connect SystemVerilog interface.

The Verilog code to instantiate the above is shown below. This example is based on using the ac7t1500 device

```
    // ----------------------------------------------
    // Instantiate Speedster7t1500
    // ----------------------------------------------
    // Connect the chip ready port
    // Note : All ac7t1500 ports are defined, so can be directly connected if required
    ac7t1500 ac7t1500( .FCU_CONFIG_USER_MODE (chip_ready ) );

    // Set the verbosity options on the messages
    // Use the inbuilt set_verbosity() task.
    initial begin
        ac7t1500.set_verbosity(3);
    end

    // ----------------------------------------------
```

```
// Bind NAPs
// ----------------------------------------------
// Bind my_nap1 to location 4,5
`ACX_BIND_NAP_AXI_SLAVE(dut.my_nap1,4,5);
// Bind my_nap2 to location 2,2
`ACX_BIND_NAP_AXI_SLAVE(dut.my_nap2,2,2);

// ----------------------------------------------
// Connect to DC interfaces
// ----------------------------------------------
// Create signals to attach to direct-connect interface
logic                         my_dc0_1_clk;
logic                         my_dc0_1_awvalid;
logic                         my_dc0_1_awaddr;
logic                         my_dc0_1_awready;
.....
logic                         my_dc0_2_clk;
logic                         my_dc0_2_awvalid;
logic                         my_dc0_2_awaddr;
logic                         my_dc0_2_awready;
.....

// Connect signals to gddr6_xx_dc0 interface within ac7t1500 device
// Inputs to device
assign ac7t1500.gddr6_xx_dc0.awvalid  = my_dc0_1_awvalid;
assign ac7t1500.gddr6_xx_dc0.awaddr   = my_dc0_1_awaddr;
....
// Outputs from device
assign my_dc0_1_awready = ac7t1500.gddr6_xx_dc0.awready;
....

// Connect signals to gddr6_xx_dc0 interface within ac7t1500 device
// Inputs to device
assign ac7t1500.gddr6_yy_dc0.awvalid  = my_dc0_2_awvalid;
assign ac7t1500.gddr6_yy_dc0.awaddr   = my_dc0_2_awaddr;
....
// Outputs from device
assign my_dc0_2_awready = ac7t1500.gddr6_yy_dc0.awready;
....

// ----------------------------------------------
// Remember to connect the clock!
// ----------------------------------------------
assign my_dc0_1_clk = ac7t1500.gddr6_xx_dc0.clk;
assign my_dc0_2_clk = ac7t1500.gddr6_yy_dc0.clk;
```

Note

When using bind macros, the user is able to specify the column and row coordinates of the target NAP. To ensure consistency between simulation and silicon, the user should add matching placement constraints to the ACE placement `.pdc` file, for example:

In simulation

```
`ACX_BIND_NAP_AXI_SLAVE(dut.my_nap1,4,5);
```

In place and route

```
set_placement -fixed {i:my_nap} {s:x_core.NOC[4][5].logic.noc.nap_s}
```

set_verbosity() Task

Alongside specifying the required simulation package version and instantiating the device, the user can control the verbosity of the messages that are output from the device simulation model. These levels are controlled by the set_verbosity task. An example of how to call this function is given in the code examples above.

The verbosity levels are defined in the following table.

Table 19: ***Verbosity Levels***

Verbosity Level	Description
0	Print no messages
1	Print messages from master and slave interfaces only
2	Print messages from level 1 and from each NoC data transfer
3	Print messages from level 2 and NoC performance statistics

Chip Status Output

From initial simulation start, the device operates similarly to its silicon equivalent with an initialization period when the device is in reset. In hardware this occurs during configuration as the bitstream is loaded. After this initialization period, the device asserts the `FCU_CONFIG_USER_MODE` signal to indicate that it has entered user mode, whereby the design starts to operate.

It is suggested that the top-level testbench monitor `FCU_CONFIG_USER_MODE` and only starts to drive stimulus into the device once this signal is asserted (shown in the example above by use of a testbench `chip_ready` signal).

Bind Macros

The following bind statements are available.

Table 20: ***Bind Macros***

Macro	Arguments	Description
ACX_BIND_NAP_HORIZONTAL	user_nap_instance, noc_column, noc_row	To bind a horizontal streaming NAP, instance ACX_NAP_HORIZONTAL.
ACX_BIND_NAP_VERTICAL	user_nap_instance, noc_column, noc_row	To bind a vertical streaming NAP, instance ACX_NAP_VERTICAL.
ACX_BIND_NAP_AXI_MASTER	user_nap_instance, noc_column, noc_row	To bind an AXI master NAP, instance ACX_NAP_AXI_MASTER.

Macro	Arguments	Description
ACX_BIND_NAP_AXI_SLAVE	user_nap_instance, noc_colunm, noc_row	To bind an AXI slave NAP, instance ACX_NAP_AXI_SLAVE.

Direct-Connect Interfaces

The following direct-connect interfaces are available.

Table 21: *Direct-Connect Interfaces*

Subsystem	Interface Name	Physical Location	GDDR6 Channel	SystemVerilog Interface Type	Data Width	Address Width
GDDR6	gddr6_1_dc0	West 1	0	t_ACX_AXI4	512	33
GDDR6	gddr6_1_dc1	West 1	1	t_ACX_AXI4	512	33
GDDR6	gddr6_2_dc0	West 2	0	t_ACX_AXI4	512	33
GDDR6	gddr6_2_dc1	West 2	1	t_ACX_AXI4	512	33
GDDR6	gddr6_5_dc0	East 1	0	t_ACX_AXI4	512	33
GDDR6	gddr6_5_dc1	East 1	1	t_ACX_AXI4	512	33
GDDR6	gddr6_6_dc0	East 2	0	t_ACX_AXI4	512	33
GDDR6	gddr6_6_dc1	East 2	1	t_ACX_AXI4	512	33
DDR	ddr4_dc0	South	–	t_ACX_AXI4	512	40

Note

Not all interfaces are available in all devices. Please consult the appropriate device datasheet to understand which interfaces are present in the selected device.

Clock Frequencies

In addition to binding to the interfaces, it is possible to control the frequencies of the clocks generated by these interfaces. For design integrity the clock frequencies set within simulation should match the desired design operating frequencies. For design implementation, the frequencies are configured within the ACE IO Designer tool. For simulation, the `set_clock_period` function is provided.

The example below shows setting the GDDR6 East 1 controller to an operating frequency of 1 GHz (suitable for 16 Gbps operation). As the DC interface operates at half the controller frequency, it is then configured for 500 MHz.

Using this method the user can first ensure that the simulation operates at the correct frequencies. Second, they are able to operate each subsystem at a different frequency if required.

```
// Set default GDDR6 clock frequency to 1000 ps = 1GHz
localparam GDDR6_CONTROLLER_CLOCK_PERIOD = 1000;
```

```
// Configure the NoC interface of GDDR6 E1 to 1GHz
ac7t1500.clocks.set_clock_period("gddr6_5_noc0_clk", GDDR6_CONTROLLER_CLOCK_PERIOD);

// Configure the DC interface of GDDR6 E1 to 500MHz, (double the period of the NoC interface)
ac7t1500.clocks.set_clock_period("gddr6_5_dc0_clk", GDDR6_CONTROLLER_CLOCK_PERIOD*2);
```

Note

The `set_clock_period` function is within the ac7t1500 model. This model has a default timescale value of 1 ps; therefore, the specified clock period will be applied in picoseconds, irrespective of the timescale value of the calling module.

The following clock frequency interfaces are available

Table 22: *Clock Frequency Interfaces*

Subsystem	Interface Name	Physical Location	GDDR6 Channel
GDDR6	gddr6_0_noc0_clk	West 0 NoC	0
GDDR6	gddr6_0_noc1_clk	West 0 NoC	1
GDDR6	gddr6_1_noc0_clk	West 1 NoC	0
GDDR6	gddr6_1_noc1_clk	West 1 NoC	1
GDDR6	gddr6_2_noc0_clk	West 2 NoC	0
GDDR6	gddr6_2_noc1_clk	West 2 NoC	1
GDDR6	gddr6_3_noc0_clk	West 3 NoC	0
GDDR6	gddr6_3_noc1_clk	West 3 NoC	1
GDDR6	gddr6_4_noc0_clk	East 0 NoC	0
GDDR6	gddr6_4_noc1_clk	East 0 NoC	1
GDDR6	gddr6_5_noc0_clk	East 1 NoC	0
GDDR6	gddr6_5_noc1_clk	East 1 NoC	1
GDDR6	gddr6_6_noc0_clk	East 2 NoC	0
GDDR6	gddr6_6_noc1_clk	East 2 NoC	1
GDDR6	gddr6_7_noc0_clk	East 3 NoC	0
GDDR6	gddr6_7_noc1_clk	East 3 NoC	1
GDDR6	gddr6_1_dc0_clk	West 1 DCI	0

Subsystem	Interface Name	Physical Location	GDDR6 Channel
GDDR6	gddr6_1_dc1_clk	West 1 DCI	0
GDDR6	gddr6_2_dc0_clk	West 2 DCI	0
GDDR6	gddr6_2_dc1_clk	West 2 DCI	1
GDDR6	gddr6_5_dc0_clk	East 1 DCI	0
GDDR6	gddr6_5_dc1_clk	East 1 DCI	1
GDDR6	gddr6_6_dc0_clk	East 2 DCI	0
GDDR6	gddr6_6_dc1_clk	East 2 DCI	1
DDR	ddr4_noc0_clk	South NoC	–
DDR	ddr4_dc0_clk	South DCI	–
PCIe	pciex16_clk	Gen5 PCIe ×16	–
PCIe	pciex16_dc_clk	Gen5 PCIe ×16 DCI	–
PCIe	pciex8_clk	Gen5 PCIe ×8	–
Configuration	cfg_clk	System wide configuration clock	–

Configuration

A number of the interface subsystems require configuration at power-up. In the physical device, this configuration would be performed by the bitstream pre-programming the relevant configuration registers. Within the simulation environment, there are tasks that can read configuration files and apply those files to the relevant interface subsystem. An example of applying a configuration is shown in the code snippet below

```
// -------------------------
// Configuration
// -------------------------

// Call function within device to configure the registers
// By using fork-join, the two configurations will be run in parallel, configuring both
// Ethernet blocks.  This saves overall simulation time.
// Both blocks are configured the same, hence the use the same file
initial
begin
    fork
        ac7t1500.fcu.configure( "ethernet_cfg.txt", "ethernet0" );
        ac7t1500.fcu.configure( "ethernet_cfg.txt", "ethernet1" );
    join
end
```

Startup Sequence

While the task `fcu.configure()` is processing the configuration (including waiting for any polling to return a valid value), the Chip Status Output (see page 37) is not asserted. This behavior mirrors that of the device where the device will only enter user mode once configuration is completed.

The simulation testbench can issue configuration processes as shown above, and once the Chip Status Output (see page 37) is asserted, the testbench will know the device is correctly configured. The testbench can then proceed to apply the necessary tests.

fcu.configure() Task

The task `fcu.configure` has the following arguments:

```
fcu.configure ( <configuration filename>, <interface subsystem name> );
```

The following interface subsystem names are supported:

Table 23: *Configuration Subsystem Names*

Subsystem	Interface Subsystem Name (1)	Physical Location
GDDR6	gddr6_0	West 0
GDDR6	gddr6_1	West 1
GDDR6	gddr6_2	West 2
GDDR6	gddr6_3	West 3
GDDR6	gddr6_4	East 0
GDDR6	gddr6_5	East 1
GDDR6	gddr6_6	East 2
GDDR6	gddr6_7	East 3
DDR	ddr4	South
Ethernet	ethernet0	North
Ethernet	ethernet1	North
GPIO North	gpio_n	North
GPIO South	gpio_s	South
PCIe ×8	pcie_0	North
PCIe ×16	pcie_1	North
All subsystems	full (2)	–

Subsystem	Interface Subsystem Name [1]	Physical Location

Table Notes

1. The interface subsystem name is case insensitive.
2. When using the `full` subsystem name, the full 42-bit address is required in the configuration file. When selecting an individual subsystem, only the 28-bit address is required. See Configuration File Format (see page 42) below for details.

Configuration File Format

The configuration file has the following format:

```
# ------------------------------------------
# Config file
# Supports both # and // comments
# ------------------------------------------

# A comment line
// Another comment line

# Format is <cmd> <addr> <data>

# Commands are
 "w" - write
 "r" - read
 "v" - read and verify

# Address is either 28-bit, (7 hex characters), or 42-bit, (11 hex characters).
# 28-bits supports the configuration memory space of an single interface subsystem
# 42-bits supports the full configuration memory space

# Data is 32-bit, (8 hex characters).

# For reads, put 0x0 for the data
# For verify put the expected data value

# Examples

# Writes
w 00005c0 76543210
w 0000014 00004064

# Reads
r 00005c0 00000000
r 0000014 00000000

# Verify
v 00005c0 76543210
```

```
v 0000014 00004064
```

Address Width

The address width varies according to the requirements of the file:

- When addressing an individual subsystem, only the lower 28 bits of the address field are used. The higher 14 bits are derived from the subsystem name.
- When addressing the full configuration memory space (interface subsystem name is set to `full`), then 42 bits of the address space are required. In this mode, the FCU confirms that bits [41:34] of the address field are set to 8'h20, which aligns with the NoC global memory map plus control and status register (CSR) memory area. In this mode, the one configuration file can address multiple interface subsystems. See the *Speedster7t Network on Chip User Guid*e (UG089) for more details.

Parallel Configuration

The `fcu.configure()` task is defined as a SystemVerilog automatic task to allow it to be re-entrant and run in parallel. Therefore, it is possible to program multiple interface subsystems in parallel using a `fork - join` construct. Refer to the reference design testbench for examples of this parallel programming.

SystemVerilog Interfaces

The following SystemVerilog interfaces are defined, and are used for DCI assignments.

Note

The interface below is only available in the simulation environment. For code that must be synthesized, users need to define their own SystemVerilog interfaces, or use one of the interfaces predefined within the reference designs.

```
interface t_ACX_AXI4
    #(DATA_WIDTH = 0,
      ADDR_WIDTH = 0,
      LEN_WIDTH  = 0);

   logic                       clk;       // Clock reference
   logic                       awvalid;   // AXI Interface
   logic                       awready;
   logic [ADDR_WIDTH -1:0]     awaddr;
   logic [LEN_WIDTH -1:0]      awlen;
   logic [8 -1:0]              awid;
   logic [4 -1:0]              awqos;
   logic [2 -1:0]              awburst;
   logic                       awlock;
   logic [3 -1:0]              awsize;
   logic [3 -1:0]              awregion;
   logic [3:0]                 awcache;
   logic [2:0]                 awprot;
   logic                       wvalid;
   logic                       wready;
   logic [DATA_WIDTH -1:0]     wdata;
   logic [(DATA_WIDTH/8) -1:0] wstrb;
   logic                      wlast;
   logic                      arready;
   logic [DATA_WIDTH -1:0]    rdata;
   logic                      rlast;
   logic [2 -1:0]             rresp;
   logic                      rvalid;
   logic [8 -1:0]             rid;
   logic [ADDR_WIDTH -1:0]    araddr;
   logic [LEN_WIDTH -1:0]     arlen;
   logic [8 -1:0]             arid;
   logic [4 -1:0]             arqos;
   logic [2 -1:0]             arburst;
   logic                      arlock;
   logic [3 -1:0]             arsize;
   logic                      arvalid;
   logic [3 -1:0]             arregion;
   logic [3:0]                arcache;
   logic [2:0]                arprot;
   logic                      aresetn;
   logic                      rready;
   logic                      bvalid;
   logic                      bready;
   logic [2 -1:0]             bresp;
   logic [8 -1:0]             bid;

   modport master (input  awready, bresp, bvalid, bid, wready, arready, rdata, rlast, rresp,
rvalid, rid,
                   output awaddr, awlen, awid, awqos, awburst, awlock, awsize, awvalid, awregion,
                          bready, wdata, wlast, rready, wstrb, wvalid,
                          araddr, arlen, arid, arqos, arburst, arlock, arsize, arvalid, arregion);

   modport slave  (output awready, bresp, bvalid, bid, wready, arready, rdata, rlast, rresp,
rvalid, rid,
                   input  awaddr, awlen, awid, awqos, awburst, awlock, awsize, awvalid, awregion,
                          bready, wdata, wlast, rready, wstrb, wvalid,
                          araddr, arlen, arid, arqos, arburst, arlock, arsize, arvalid, arregion);

endinterface : t_ACX_AXI4
```

Installation

There is a DSM package per device, available for both Linux and Windows. The packages are named `<device>_IO_BFM_Sim_<version>.tgz` for Linux and `<device>_IO_BFM_Sim_<version>.zip` for Windows. The packages are available on the Achronix self-service FTP site at secure.achronix.com, located in the `/Achronix/ACE/Speedster7t` folder. As each device package is independent, it is possible to either download only the device the user is targeting, or all device packages.

Any package is only required to be installed once — it is common for all designs targeting the selected device.

ACE Integration

Upgrading an Existing Installation

If a version of the DSM package was previously installed into ACE, it is recommended to first delete the existing DSM package before upgrading to ensure the integrity of the new installation. To delete an existing package, navigate to `<ACE_INSTALL_DIR/system/data/yuma-alpha-rev0` and remove the `/sim` directory. Then return to the root of the ACE installation and proceed with the instructions for First Installation (see page 45) below.

First Installation

The recommended installation method is to merge the contents of the package into the current ACE installation. The package contains a root directory `/system`. The contents of this folder should be merged with the selected ACE installation `/system` folder.

Warning!

The contents of the simulation package consists of files that are not present in the base ACE installation. These files should not replace or overwrite any existing files. However, if the user has already downloaded an earlier version of the simulation package, then they should select "overwrite" to ensure the latest version of the simulation files are written to the ACE installation.

Standalone

In certain instances it may not be possible for a user to modify their existing ACE installation. In these cases it is possible to install the package separately and to simulate using files from both this simulation package and the existing simulation files within ACE.

To install as standalone, simply uncompress the package to a suitable location.

Note

All reference designs are configured for the simulation package to be integrated within ACE. If the standalone method is selected, the user must edit the necessary environment variables in the reference design makefiles.

Environment Variables

The locations of both ACE and the simulation package are controlled by two environment variables. For all reference designs these two variables must be set before simulating.

ACE_INSTALL_DIR

The environment variable ACE_INSTALL_DIR must be set to the directory location of the `ace`, or `ace.exe` executable. This variable is used by both simulation and synthesis to locate the correct device library files.

ACX_DEVICE_INSTALL_DIR

The environment variable ACX_DEVICE_INSTALL_DIR is used to select the DSM files. It should be set to the base directory of the device files within the DSM package. For example if the ac7t1500 device is selected, then the device base directory is `yuma-alpha-rev0`.

When the installation is done as ACE integration mode, then the following setting should be used:

```
ACX_DEVICE_INSTALL_DIR = $ACE_INSTALL_DIR/system/data/yuma-alpha-rev0
```

When the installation is done as standalone, the the following setting should be used:

```
ACX_DEVICE_INSTALL_DIR = <location of standalone package>/system/data/yuma-alpha-rev0
```

Simulating the Reference Design

Supported Simulators

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. Within this directory there are `/vcs` and `/questa` directories for each of the simulators.

Simulation Flows

Where applicable the simulation supports a number of flow options, which offer a balance between speed and accuracy. Not all flow options are available with all reference designs; the relevant makefiles will list what flow options can be set.

Standalone

Any NAP in the design uses a standalone model bound to the NAP, modelling memory behavior. This mode is the quickest simulation to run, but is the least cycle accurate. The NAP will only interact with its own memory model; therefore, this mode does not support multiple NAPs designed to access a common memory. To enable this mode in the supplied reference design, set the `FLOW` variable in the simulation makefile to `STANDALONE`

Full-Chip BFM

This uses a model of the full chip, with cycle-accurate NoC. There are then bus functional models (BFMs) for all the hardened interfaces around the NoC. These BFMs have representative delays, allowing this mode to offer near cycle-accurate simulations. This mode does not require the interface subsystems to perform initialization and calibration steps, offering a quicker iterative time compared to a full cycle-accurate simulation.

The full-chip BFM simulations require the I/O Ring Simulation Package to have been downloaded and installed. To enable this mode in the supplied reference design, set the `FLOW` variable in the simulation makefile to `FULLCHIP_BFM`.

Full-Chip RTL

This mode uses the full RTL of the subsystem combined. if necessary, with a cycle-accurate model of any necessary external component (such as a memory). This configuration gives a fully cycle-accurate simulation representing the final silicon operation. For most of these simulations it will be necessary to configure the relevant subsystems using the provided configuration files. As these simulations are using the full RTL of the subsystem, they run slower than the BFM equivalent simulations, while offering complete timing accuracy.

Note

To obtain the encrypted RTL of the GDDR/DDR or PCIe subsystems, a second licensed simulation package is required. Please contact Achronix Support to arrange licensing and access to this package.

To enable this mode in the supplied reference design, set the `FLOW` variable in the simulation makefile to `FULLCHIP_RTL`.

Build Options

Within each simulator directory is a makefile. This makefile can be edited by the user to configure the simulation to their needs. The following variables need to be set:

- `FLOW` – To match the selected simulation flow. The options (detailed in the makefile) are `STANDALONE, FULLCHIP_BFM or FULLCHIP_RTL.`
- `TOP_LEVEL_MODULE` – Preset for the supplied design. However, if the user ports the design to their own testbench, this variable must be updated.
- `ACX_DEVICE_INSTALL_DIR` – Points to the directory (normally under ACE) where the target device files are stored, for example, `$ACE_INSTALL_DIR/system/data/yuma-alpha-rev0`. For further information consult the I/O ring simulation installation instructions.

Prerequisites

Before running any installation, the user must ensure the following are configured:

- `ACE_INSTALL_DIR` environment variable: This should point to the ACE installation directory, where the ACE executable is located.
- Path to the required simulator. For VCS this should also include the VCS_HOME environment variable.

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 or ../../scripts/sim_template_bfm.f` to define the general simulation options. The create script is called by the specific simulator makefile 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/vendor` directory the following scripts are located:

- `makefile` – Makefile supporting the various simulation flows. Default target is to compile and run the simulation.
- `system_files_bfm.f` – (Full-chip BFM flow only) List of system files used by the full-chip BFM flow. Any user defines can also be added to this file.
- `system_files_rtl.f` – (Full-chip RTL flow only) List of system files used by the full-chip RTL flow. Also any defines required to specify which subsystems should be cycle-accurate RTL rather than BFM. The defines are named as <subsystem name>_FULL, i.e., GDDR6_2_FULL. Any user defines can also be added to this file.

VCS Only

- `fullchip_bfm_vcs_waiver.cfg` – (Full-chip BFM flow only) Waiver file to remove benign warnings.

- `session.sim_output_pluson.vpd.tcl` – Session file for DVE waveform viewer.

QuestaSim Only

- `wave.do` – Waveform file.
- `qsim_<design_name>.do` – Non-makefile GUI flow. OS independent.

RTL Simulation Defines

In order to compile and run the Full-Chip RTL flow, the user will need to enable SystemVerilog defines for the simulation compilation. When the simulation makefile `FLOW` variable is set to `FULLCHIP_RTL`, the makefile will include the `system_files_rtl.f` file in the compilation command line.

Any required defines to toggle simulation blocks from BFM model to full RTL are enabled within the `system_files_rtl.f` file. For example, if using the GDDR6 reference design, and the user decides they need to simulate GDDR6 memory controller 1 with the full RTL, but leave all other GDDR6 controllers using the BFM model, they need to edit the `system_files_rtl.f` file as shown below:

```
# Define GDDR Data Rate
+define+GDDR6_DATA_RATE_16     # <----- For GDDR6 RTL simulation the user must
                                        enable one of the data rates.
//+define+GDDR6_DATA_RATE_14
//+define+GDDR6_DATA_RATE_12

# Turn on GDDR RTL
# Enable the desired GDDR memory controllers below
# Any undefined controllers will use their BFM model
//+define+ACX_GDDR6_0_FULL
+define+ACX_GDDR6_1_FULL        # <--- User enables this define
//+define+ACX_GDDR6_2_FULL
//+define+ACX_GDDR6_3_FULL
//+define+ACX_GDDR6_4_FULL
//+define+ACX_GDDR6_5_FULL
//+define+ACX_GDDR6_6_FULL
//+define+ACX_GDDR6_7_FULL

<-------- User does not need to change any options below here  ---->

# ACE libraries must be defined first as they are referenced
# by the fullchip files that follow
+incdir+$ACE_INSTALL_DIR/libraries/
$ACE_INSTALL_DIR/libraries/device_models/7t_simmodels.v

# Fullchip include filelist
# This must be placed before the user filelist as it defines
# the binding macros and utilities used by the user testbench.
$ACX_DEVICE_INSTALL_DIR/sim/ac7t1500_include_bfm.v
```

The table of available defines to enable each module to use full RTL rather than BFM model is listed in the table below (see page 48)

Table 24: *Simulation RTL defines*

Module	Define
GDDR6 controller 0	ACX_GDDR6_0_FULL
GDDR6 controller 1	ACX_GDDR6_1_FULL
GDDR6 controller 2	ACX_GDDR6_2_FULL
GDDR6 controller 3	ACX_GDDR6_3_FULL
GDDR6 controller 4	ACX_GDDR6_4_FULL
GDDR6 controller 5	ACX_GDDR6_5_FULL
GDDR6 controller 6	ACX_GDDR6_6_FULL
GDDR6 controller 7	ACX_GDDR6_7_FULL
DDR4 controller	ACX_DDR4_FULL
Ethernet subsystems (both)	ACX_ETHERNET_FULL
PCIe Controller 0 (×8)	ACX_PCIE_0_FULL
PCIe controller 1 (×16)	ACX_PCIE_1_FULL
GPIO North block	ACX_GPIO_N_FULL
GPIO South block	ACX_GPIO_S_FULL
All SerDes lanes	ACX_SERDES_FULL

Table Note

Locations and names of each of the interface subsystems above is visible using the ACE IP Configuration perspective, and selecting the I/O layout diagram

Warning!

Enabling full RTL simulation for modules will increase simulation time. If several modules are enabled, the simulation time could be extended by significant amounts.

Running the Simulation

For all simulator flows there is a `makefile` located in the simulation directory. The makefiles support the following build options

VCS

code

```
$ make            : Default flow. Compile with no debug options, run in batch mode writing the
output to a VPD file.
$ make run        : Same as "make".
$ make debug      : Build with debug options (increased visibility of lower level variables).
                    Run in batch mode writing to a VPD file.
$ make open_dve   : Open the DVE waveform viewer and load the VPD file and viewing session file.
$ make clean      : Delete all generated and temporary files.
```

QuestaSim

code

```
$ make            : Default flow. Compile with no debug options, run in batch mode writing the
output to a WLF file.
$ make run        : Default flow. Same as "make".
$ make debug      : Build with debug options (increased visibility of lower level variables).
                    Open GUI with wave.do and allow user to run interactively.
$ make open_wave : Open the GUI waveform viewer. Load the generated WLF file and viewing wave.do f
ile.
$ make clean      : Delete all generated and temporary files.
```

QuestaSim Non-Makefile Flow

To support environments that do not natively support makefiles (such as Windows), there is an additional QuestaSim non-makefile flow using QuestaSim `.do` files. The following steps are required to use this flow:

1. Navigate to `sim/questa`
2. Launch QuestaSim GUI. Normally:

   ```
   $ vsim
   ```

   Within the QuestaSim GUI launch the script

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

Note

The QuestaSim script uses the ACE_INSTALL_DIR environment variable. For correct operation of this, (or any other), script flow, ACE should be installed in a directory without spaces in the path name, e.g., `C:\achronix\8.2.1\Achronix_CAD_Environment`. Additionally the environment variable ACE_INSTALL_DIR must use "/" as path separators rather than "\", for example:

`ACE_INSTALL_DIR = C:/achronix/8.2.1/Achronix_CAD_Environment/Achronix`

Note

The do script is configured for the `FULLCHIP_BFM` flow. It can be modified to match the `STANDALONE` flow by selecting the appropriate options commented within the script.

Results Verification

All the designs make use of a self-checking testbench 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.

Building the Reference Design

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

Prerequisites

Before running any installation, the user must ensure the following are configured:

- `ACE_INSTALL_DIR` environment variable. This variable must point to the ACE installation directory where the ACE executable is located.
- ACE should be in the environment path. The easiest method is to add `$ACE_INSTALL_DIR` to the path variable.
- Synplify Pro should be in the environment path.

Batch Flow

In the root directory, there is a `/build` directory, within which there is a `makefile`. Before running the makefile, ensure the prerequisites above have been met. The relative paths within the makefile are intended to be run from the `/build` directory. If the makefile is moved to a new location or called outside of this directory, then the paths will require amending accordingly.

When the makefile is run (with the default options), it will create the following;

- `/build/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.`
- `/build/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.

Makefile Options

The makefile supports multiple build flow options

code

```
$ make              : Default flow. Synthesize and build a single implementation with ACE
$ make run        : Same as "make"
$ make syn_only    : Synthesis only
$ make pnr_only : Run ACE place and route only. This requires synthesis to have previously been
run.
$ make run_mp   : Run multiprocess. Synthesize and build multiple implementations with ACE
multiprocess.
$ make clean      : Delete all generated and temporary files
```

Constraint Files

In addition to the constraint files listed above, additional files may be used in any build flow. All constraint files are located in `/src/constraints:`

- `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

The design has pre-generated GUI project files for both ACE and Synplify Pro. These files are located in `/src/ace` and `/src/syn` directories respectively. The user can open these to interactively edit or run builds.

Note

When using the GUI projects, any generated files will be placed beneath the GUI project file directory.

- For Synplify Pro, the revision directory, rev_1 etc. will be generated in `/src/syn/rev_1.`
- For ACE, any implementation directory will be generated as `/src/ace/impl_<name>.`
- For ACE, if I/O Designer is used to generate new constraint files, the default directory for those files is `/src/ace/ioring_design.`

When builds are done using the batch flow, the flow writes out both the relevant project files under the `/build/results` directory. Within this directory are the generated project files for both ACE and Synplify Pro. The user can open both of these project files in GUI mode and interactively re-run or edit the builds.

Revision History

Version	Date	Description
1.0	09 Apr 2020	• Initial release.
2.0	11 Jan 2021	• Add MVM and dot product designs. • Convert ASCII text timing waveforms to graphical diagrams.
2.1	15 Jan 2021	• Updated links.
2.2	19 Jan 2021	• Set Conv"D frequency to 600 MHz. Updated performance details accordingly.

Achronix®
Data Acceleration

Achronix Semiconductor Corporation

2903 Bunker Hill Lane
Santa Clara, CA 95054
USA

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

Notice of Disclaimer