POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Actuation control for Flight Control Computer application by using parallel computation via AMD Xilinx Versal AI Engine

Supervisor:
Prof. Mario Roberto Casu
Co-supervisors:
Eng. Antonio Bruscino
Eng. Riccardo Sticca

Candidate
Paolo Molino

Academic Year 2024-2025

Abstract

The advent of high-performance computing on aircraft computer platforms introduces several design challenges. In particular, digital fly-by-wire Flight Control Computers rely on sophisticated actuation control loop algorithms, which demand significant hardware resources for efficient execution.

To address these challenges, the AMD Xilinx Versal ACAP (Adaptive Compute Acceleration Platform) provides a cutting-edge and promising architecture. This FPGA is the industry's first compute platform that integrates dedicated acceleration engines (AI Engines) for scalar and vector processing, tightly coupled with programmable logic and configurable on-chip connectivity. Such a heterogeneous architecture enables the design of customized, high-performance hardware solutions.

This thesis, carried out in collaboration with Leonardo Electronics, investigates the development of digital microarchitectures for airborne electronic modules on the Versal platform, and evaluates their hardware implementation both in standard DSP logic blocks and in AI Engines according to the required operation schedules. In particular, using Vitis Model Composer—a high-level development environment—an actuator control loop has been analyzed and mapped across AI Engines and Programmable Logic, interconnected through the AXI4 protocol. C++ code for AI Engines has been developed, while Programmable Logic VHDL code has been generated via graphical block design. Subsequently, the VCK190 Evaluation Board has been programmed accordingly, and performance has been assessed.

The results confirm the feasibility of the proposed implementation and highlight the potential to further generalize the algorithms in order to fully exploit the computational capabilities of the hardware.

Acknowledgements

I would like to express my gratitude to Antonio and Riccardo for their help and for the opportunity and trust given to me. I would also like to thank my colleagues at Leonardo for making me feel part of the team.

Contents

List of Tables

List of Figures

Chapter 1

Introduction and thesis organization

In modern aircraft, traditional direct connections between pilots' input and mechanical actuators are replaced by a *Fly-by-wire* system. This means that, before reaching the actuators in the real world, the input commands have to be processed by a *Digital Flight Control Computer* (DFCC). This process is capable of integrating autopilot and stabilization functions, and is usually performed by a CPU, which has naturally large overheads and is very inefficient for this kind of specialized computations.

An FCC realized with custom hardware is the best solution, especially given the growing complexity of control algorithms, and the performances can increase even by two orders of magnitude [21].

This electronic system is, however, *Safety-critical*: this means that a malfunction may cause important injuries to people, the environment, or important damage to property. For this reason, the custom hardware has to pass very severe certification processes, such as DO-254, which is the assurance guidance specific for airborne electronic hardware.

The safety-critical applications, such as FCCs, apply for the highest assurance level (Level A). For this reason, at the moment, the most popular choice for hardware implementation is the FPGA, which can provide both control and determinism.

This thesis, conducted in collaboration with Leonardo Electronics, focuses on the analysis of the choice of a recent Versal System on Chip (SoC) - Versal ACAP - in order to realize an actuator control loop, which can become a starting point to be taken into consideration for FCCs design in years to come.

1.1 Organization

The work begins with a description of the Versal ACAPs and their main characteristics, with a particular focus on the AI Engines. These engines represent one of AMD's most recent architectural innovations, capable of delivering high throughput through algorithm parallelization, especially in the context of DFCCs (Chapter 2).

In Chapter 3, following a brief overview of the design flow and a powerful tool that

significantly accelerates development—*AMD Vitis Model Composer*, which is based on Simulink and supports both simulation and code generation [9]—an analysis of the AXI4-Stream handshake protocol is provided. This analysis demonstrates how the protocol can facilitate the mapping of complex data flow graphs (DFGs) between various parts of the chip. The chapter concludes with a short introduction to the evaluation board used for testing the designs.

Chapter 4 details the steps taken to implement an example of Actuator Control Electronic (ACE) on the VCK190 evaluation board, using the aforementioned tool. This chapter also includes a simpler exercise implemented on the same board, whose results and performance are compared with those of the complete design.

Finally, Chapter 5 summarizes the results obtained and offers reflections on the potential future applications of the device in the avionics domain.

Chapter 2

Preliminary background

2.1 Versal ACAP

The computational demand in AI applications, real-time processing, Wireless 5G applications, modern Radars, and many other fields is continuously growing. One possibility to satisfy the need for performance are Versal adaptive System on Chips (SoCs). They are Heterogeneous devices in 7 nm FinFET technology that allow the user to exploit the advantages of both scalar and parallel computation, with a slice of the device reserved for programmable logic (PL) to provide greater flexibility.

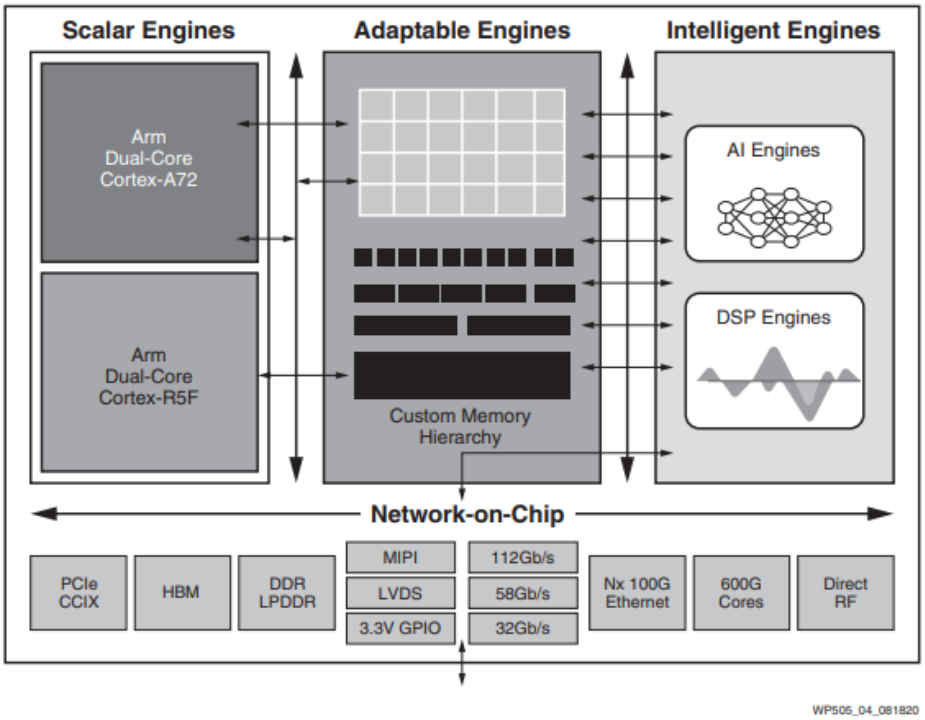


Figure 2.1. ACAP scheme [1].

Versal ACAPs (Adaptive Compute Acceleration Platform) contains (as shown in Figure 2.1) scalar processors (Arm Cortex-A72 and Arm Cortex-R5F), useful for complex algorithms; programmable logic ("Adaptable Engines" in the figure), which can be used also to obtain a custom hierarchical memory; and "Intelligent Engines": a portion of the silicon is occupied by DSP Engines, which usually are DSP58, optimized for 27 x 24 bits accumulation [2]), and AI Engines, where *AI* stands for "Adaptive Intelligent", and are basically computational tiles with both a scalar and a vector processor units (their details are better explained in the Section 2.1.1). All three parts are interconnected by a high-bandwidth Versal Network on Chip (NoC, Section 2.1.2), which also provides connections with the external world by means of a series of I/O elements and interfaces.

It is clear that this kind of device, properly programmed, is capable of solving basically any kind of computational problem. Furthermore, the user can approach them at any level, from a low-level HDL description to a higher level description by means of C/C++ codes or with the help of tools like Vitis Model Composer, providing graphical solutions (this tool will be better described in Section 3.2).

2.1.1 The AI Engines

The AI Engines are VLIW (Very Long Instruction Word) SIMD (Single Instruction Multiple Data) computational units. All the information related to AI Engines structure and features in this Section refers to [3].

The array

They are usually organized in a 2-dimensional array (Figure 2.2), in which the first row is composed of interface tiles, which allow communication from/to the NoC, communication from/to the Programmable Logic (PL), and a single configuration tile. An AI Engine array can contain 30-400 tiles.

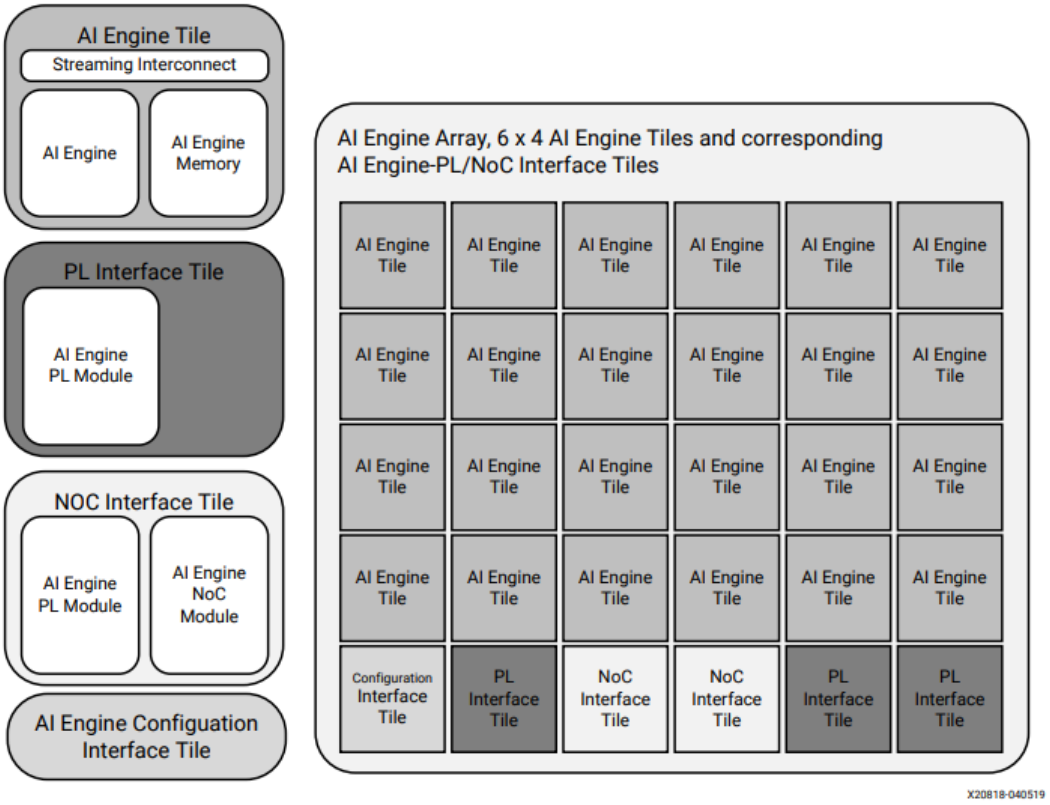


Figure 2.2. AI Engines array scheme [3].

The array interface

There are two kinds of array interface tiles:

- The NoC interface tiles are basically AXI4-Stream switches plus level shifters in AIE-NoC interface tiles (the NoC components are in different power domains).
- The PL interface tiles possess 8 AXI Stream ports in upstream and 6 in downstream, which offer a privileged path for PL-AIE communication (see Figure 2.3).

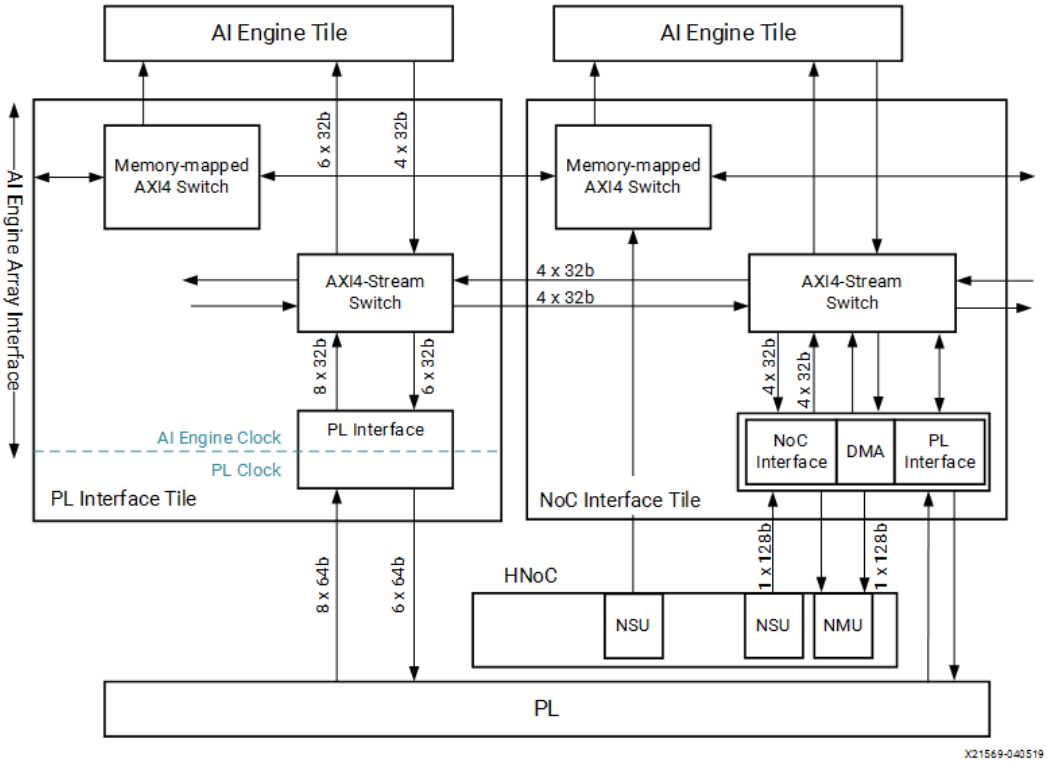


Figure 2.3. Interface tiles [3].

As concerns the configuration tile, it contains the PLL for the AI Engine's clock, the POR (Power On Reset) unit, and registers for global features. It also contains the interrupt generation unit (interrupts can be generated from events, useful for debug or performance simulations).

The AI Engine tile

Each tile (Figure 2.4) contains:

- an AXI4 interconnect block
- a memory module
- the AI engine

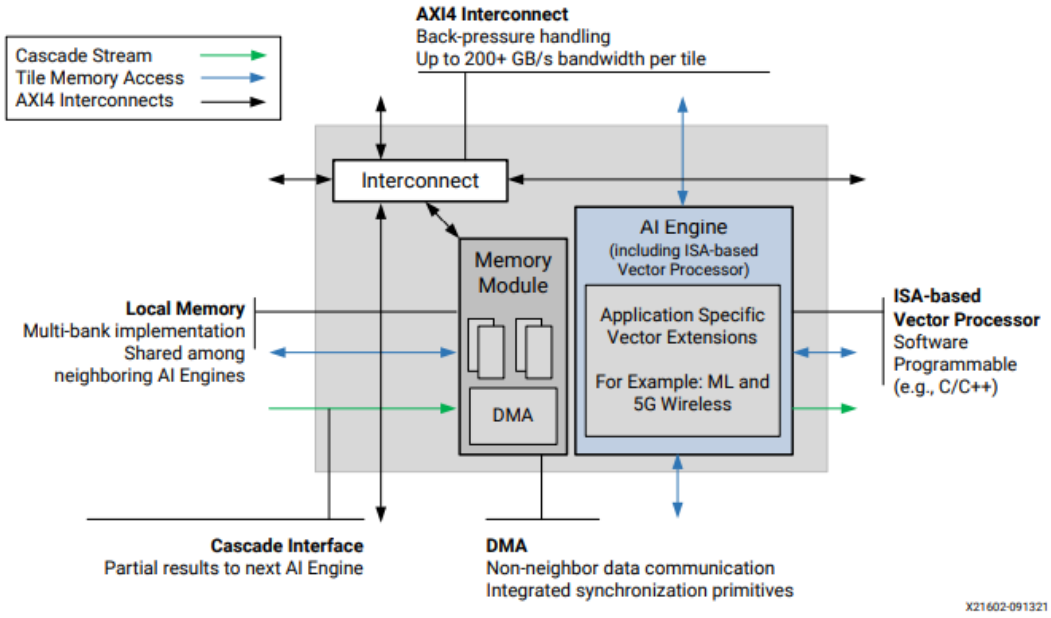


Figure 2.4. AI Engines tile scheme [3].

The interconnections

The mapping through AXI4 protocol provides the possibility to read/write the tile registers or their memory from an external master through the NoC. Also, an AXI4-Stream switch, which can be configured both for circuit-switched or packet-switched streams and is basically a programmable FIFO (4-deep FIFO with 2 cycles latency) with 6 programmable arbiters for the packet-switched stream mode.

The packet-switched streaming data transfer's latency is not deterministic, since resources can be contested, while circuit-switched streaming guarantees deterministic latency but not the best overall performance.

A cascade streaming interface allows a tile to write the results directly in the "next" tile's accumulator (a particular type of 384-bit register). Starting from the bottom-left tile, the "next" one is the one that follows a "snake" which moves in the right direction until the end of the row, then moves up of a tile and continues to the left until the end of the row, and so on...

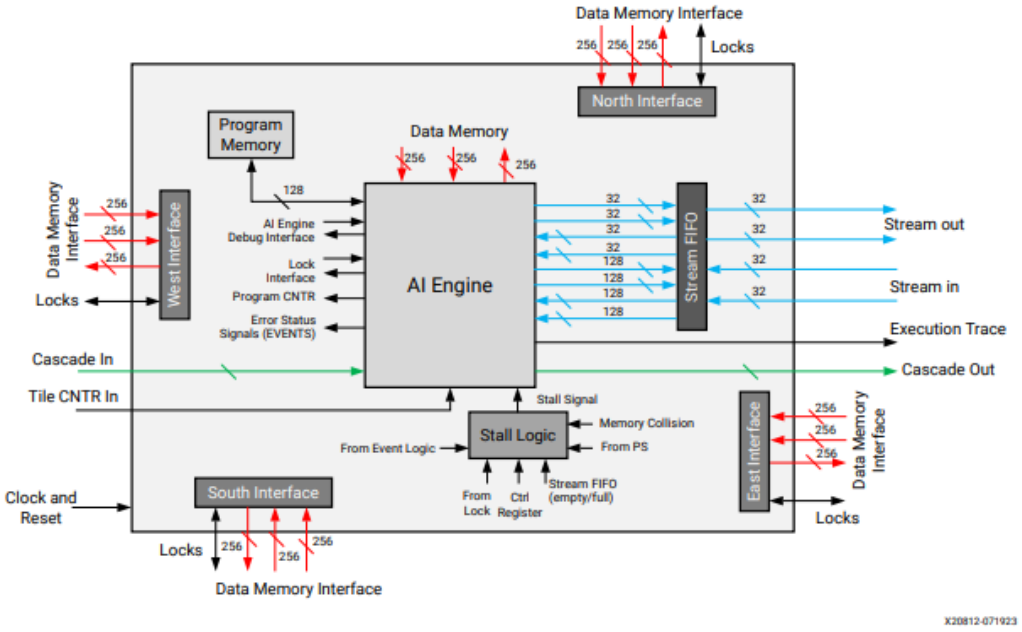


Figure 2.5. AI Engines tile interconnections [3].

The memory modules

The memory module can contain 32 kB of data. They are divided into 8 memory banks, in which the first two have 7-bit ECC for each 32-bit word, while the last two have a parity bit for each 32-bit word.

An additional 16 kB of program memory is present, with 7-bit ECC for each 128-bit instruction (it actually contains in fact 1024 x 128-bit instructions). The program memory can be accessed both from the memory-mapped AXI4 and from the AIE interface.

Despite the other ways, the main method for a couple of neighboring AIE tiles to communicate is the shared memory: each tile can access its 32 kB memory, but also the South tile's one, the North tile's ones and the Right or the Left tiles' one (depending on the position in the array), for a total of 128 kB of accessible memory.

To communicate with non-neighboring tiles, 2 x STMM (Stream to Memory Module) DMAs and 2 x MMTS (Memory Module to Streaming) DMAs are present. 16 buffer descriptors (buffers that contain the information regarding a DMA transfer) can be accessed, and a hardware synchronization (Locks) module behaves as an arbiter between simultaneous requests.

The engine

The AI Engine architecture and features are what make Versal ACAPs unique. The scheme of the computational unit is reported in Figure 2.6.

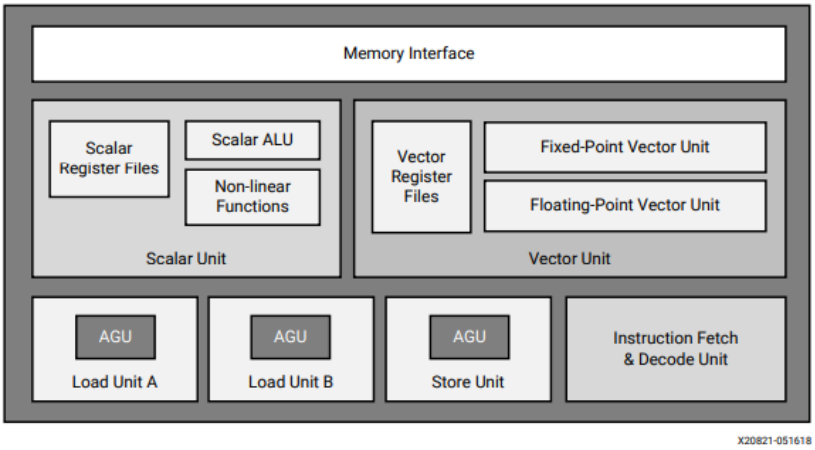


Figure 2.6. AI Engines scheme [3].

Up to 7 instructions can be issued simultaneously: 2 loads, 1 store, 1 vector operation, 1 scalar operation, 2 move operations. Who writes the kernel, which has to be in c/c++, must take this into account, and can take advantage of the AI Engine API [7], which is a set of functions and pragmas to program the engines.

In order to use these functions in an efficient way to reach the best performances possible, a good understanding of the architecture is required.

The AI Engine contains both a scalar and a vector unit.

Scalar unit

It can perform standard scalar operations such as sum, subtraction, multiplication, comparisons, bit-wise operations, but also more complex non-linear functions such as sin/cos, absolute value, count leading zeros, square root, inverse and inverse square root, or conversion float2fix and fix2float.

All these operations are performed with a throughput of 1 cycle, but the latency depends on the function; for example, multiplication is performed in 3 cycles, while non-linear functions require up to 4 cycles.

The scalar unit DOES NOT contain a floating point (FP) unit: FP operations are performed through emulation. Furthermore, the floating point arithmetic used in the AIEs is not fully compliant with the IEEE one, for 2 reasons:

- in case of saturation, 0 is returned instead of max/min values;
- if float2fix returns -2^{-31}, which is still a value within the valid range, the OVFL exception is raised.

Vector unit

The vector unit contains three different data paths:

- Multiply Accumulator (MAC) Path
- Upshift Path
- Shift-round Saturate (SRS) Path

The last one is needed since the accumulator registers are 48-bit or 80-bit, so this operation is needed in order to store the results in normal registers, which contain locations with power of two number of bits.

The engine contains scalar registers, special registers, and vector registers. These last in particular are 16 x 128-bit wide registers, which can be combined to create 256-bit, 512-bit, or 1024-bit vector registers.

The number of MACs depends on the data type and is strictly linked to the vector dimension: for what concerns fixed-point data, 128 8-bit multipliers can be post-added and combined into 16 or 8 accumulator lanes of 48 bits: the 384-bit accumulator register can be seen as 8 lanes of 48 bits. So it can perform up to 128 x 8-bit MAC, but also, for example, 8 x 32-real MACs or 2 x 32-complex MACs with a single instruction.
As concerns FP arithmetic, 8 lanes of SPFP MACs are provided, with a 3-cycle latency and 1-cycle Initialization Interval.

The vector operations allowed include comparisons, min/max (element-wise), and fix2float conversions (for float2fix, the scalar unit is used instead).
The AI Engines do not support Half Precision Floating Point (HPFP) numbers or other custom formats of floating point numbers.

Furthermore, 8 exception bits can be converted into events and raise interrupts.

Boot sequence

Here are reported the main steps of the boot sequence, which involves the PMC (Platform Management Controller), which is independent from the PS (Processing System), and is responsible for the boot and configuration of the device.

1. Power-on and power-on-reset
2. The PMC provides the AIE array configuration using the NPI interface (data comes from a flash device)
3. PLL is enabled
4. Reset is released
5. Programming of the AIE array over the memory-mapped AXI4 (from the NoC)

2.1.2 The NoC

Even if the distances across various parts of the chip are reducing, the delays are increasing due to scaling. This leads to the need to design a proper way to manage data transfer, capable of sustaining large bandwidths and programmable. The information in this Section can be found in [22].

Versal *Network on Chip* (NoC) globally interconnects all the blocks through a single memory mapping: the master connects to an entry point, called *Network Master Unit* (NMU), and the slave connects to an exit point called *Network Slave Unit* (NSU). Routes are programmable and re-programmable, and the data width is (32-512 bits).

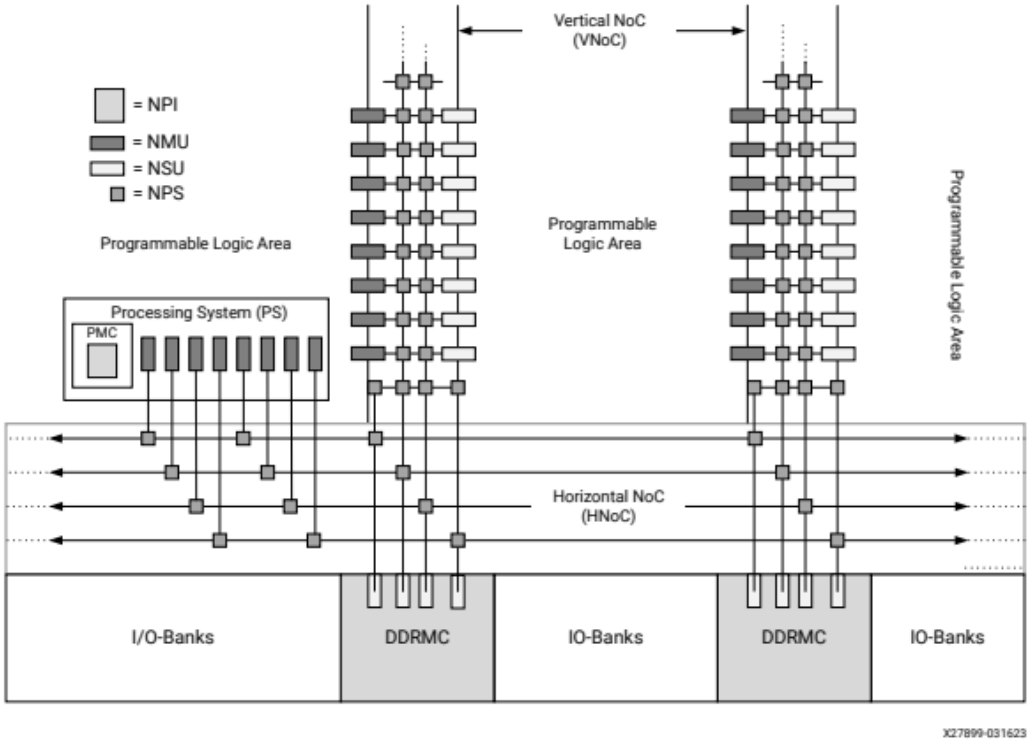


Figure 2.7. Scheme of the NoC [11]: NPI = Network Peripheral Interconnect, NMU = Network Master Unit, NSU = Network Slave Unit, NPS = Network Packet Switch.

The basic elements (NoC Switch) have 4 inputs and 4 outputs, with a minimum latency of 2 cycles, and are arranged into horizontal (HNoC) and vertical (VNoC) sub-blocks.

The network employs *Wormhole Routing* [19]: a packed switched technique in which the packet is divided into *flits*. The first one is the header, and the last one is the tail. The header contains information about the path that is to be followed, all the next ones follow it like a "worm", and the channel is set free when the tail is passed.

There is no need to wait for the entire packet to arrive in a node to start the next movement; on the contrary, this form of routing is like a simple FIFO.

Thanks to the routing scheme, the maximum number of input ports is 4096 and the latency can be deterministic.

In order to satisfy the bandwidth request and the determinism, no link has to be oversubscribed, and the routing graph has to be acyclic, such that deadlocks never occur. In Versal NoC, this is obtained by means of virtual channels and constrained routing (for example, allocating the horizontal portion of the flow always before the vertical).

Furthermore, the traffic from the starting node to the destination can easily be divided into different flows to satisfy the request more easily.

A compiler receives all the user information and constraints about data movements across the chip, and provides a report with the performance of the solution found, making this as user-friendly as efficient.

2.1.3 AIE-ML

The AIE-ML is a different and optimized version of the initial AIE. It almost doubles the computational capability, improves connectivity between different tiles, and in the array organization itself. Furthermore, the memory contained in each tile is doubled, some types of data are not supported anymore, while some others are introduced, and enhanced DMAs are introduced. For any details about them, [5] can be referred.

Since the AIE features have already been described, the remaining part of this Section will be focused on the main differences between the two versions of the engines.

New features with respect to AIE

IN AIE-ML, there is no native INT32 support: INT32 multiplications are obtained through decomposition into 16 x 16-bit simpler multiplications. Also, FP32 is supported through emulation of bfloat16 operations.

Bfloat16 and INT4 types of data are supported by this version of the engine.

The spatial disposition of the matrix is different, as shown in Figure 2.8.

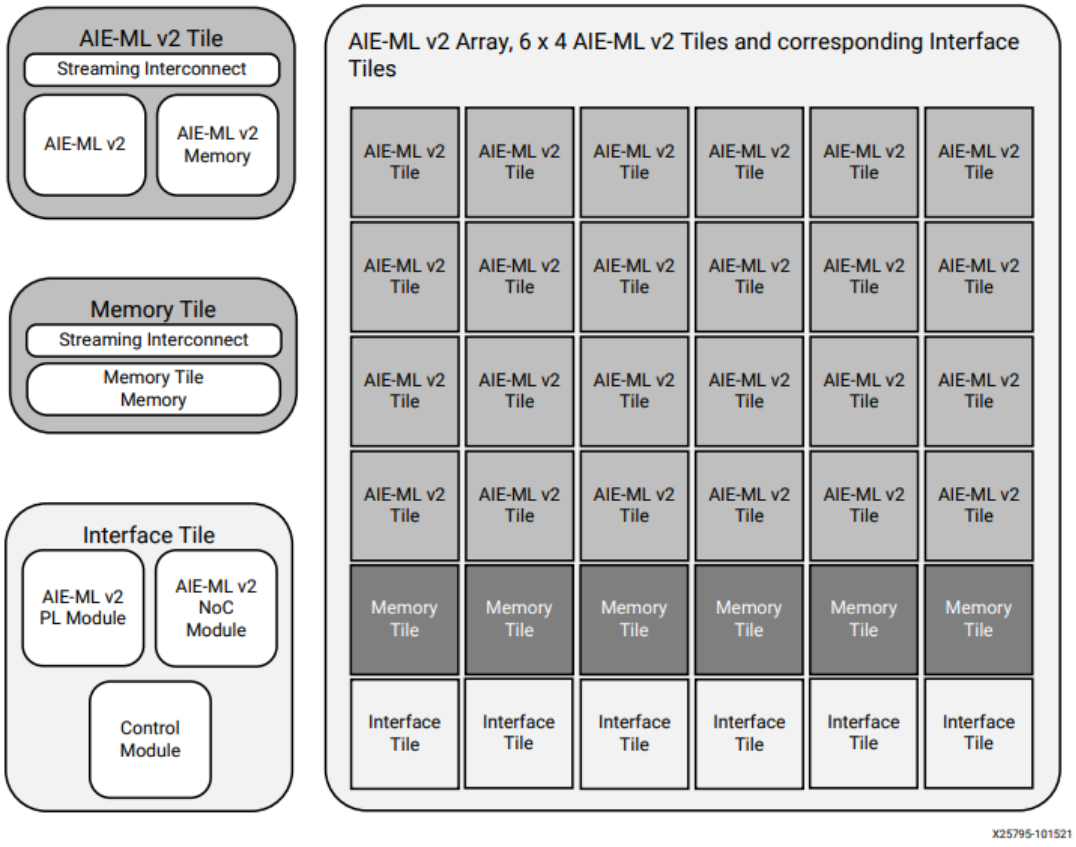


Figure 2.8. AIE-ML array scheme [5].

Each column is equipped with an interface tile, which communicates with both PL and Noc. Furthermore, additional memory tiles are added, which are capable of storing 512 kB of data with ECC and contain 12 DMAs, and support up to 30 GB/s read and write

in parallel.

Also, the data memory inside a normal tile is doubled, passing from 32 kB to 64 kB.

This new array structure supports an improved cascade mechanism: 512-bit cascade from left to right and 384-bit cascade from top to bottom (there is no more the "snake" configuration).

The operations' parallelism is doubled: it passes from 128 8b x 8b operations to 256, and introducing the INT4 data type, 512 4b x 8b MAC are supported in parallel. For floating point, the MAC parallelism is 128 bfloat16 x bfloat16 with FP32 accumulation. FP32 ops/tile grows from 16 to 42 [1], but are obtained through emulation of FP16 operations: this means that the user can choose whether to have FP32 standard accuracy or to be content with less precision in favor of faster calculation.

Sparsity support is added, while some other features are removed, such as scalar non-linear functions and not aligned memory accesses.

Table 2.1 summarizes the main differences between AIE and AIE-ML.

	AIE	AIE-ML
Array structure	Checkerboard	All lines identical
Cascade interface	384-bits wide Horizontal direction	512-bits wide Horizontal and vertical directions
Tile stream interface	2 x 32-bit in and 2 x 32-bit out	1 x 32-bit in and 1 x 32-bit out
Memory load/store per cycle	512/256 bits	512/256 bits
Advanced DSP functionality	Yes	No
INT4 operations/tile	256	1024
INT8 operations/tile	256	512
INT16 operations/tile	64	128
INT32 operations/tile	16	32
Bfloat16 operations/tile	-	256
FP32 operations/tile	16	42
Data memory/tile	32 kB	64 kB
Program memory/tile	16 kB	16 kB
Memory tiles	-	512 kB
Programmable logic (PL) to AIE array bandwidth	1X	1X
Tile local memory DMA	-	3D addressing mode, S2MM finish on TLAST, out of order packets, compression/decompression
Local memory locks	Boolean	Semaphore

Table 2.1. Summary of main differences between AIE and AIE-ML.

2.1.4 Single Event Latch-up (SEL) and Single Event Upset (SEU)

In radiation-prone environments, such as aerospace or high-altitude avionics, electronic components are subject to Single Event Effects (SEE), which occur when a high-energy particle strikes a sensitive region of the device. Among these effects, Single Event Upset

[1] Recall that 1 MAC = 2 operations

(SEU) and Single Event Latch-up (SEL) are two of the most critical phenomena to consider during system design and verification.

An SEU is a non-destructive event in which a charged particle alters the logical state of a memory cell, flip-flop, or register, typically causing a bit-flip (i.e., from '0' to '1' or vice versa). The underlying hardware remains physically intact, but the corrupted data may propagate and affect system behavior or software execution.

An SEL, by contrast, is a potentially destructive condition resulting from the triggering of a parasitic structure within the silicon substrate. This creates a low-impedance path between the power supply and ground, leading to an abnormally high current draw. If not promptly interrupted by power cycling, the overcurrent condition can cause permanent damage to the device.

Trying to test SEL and SEU countermeasure proposed by Versal, a study [17] reports: "No SEL events were observed in Xilinx 7 nm XCVC1902 at VCCmax and Tj up to $120°C$ in for both neutron & 64 MeV proton testing for an equivalent of $>$ 10 million years of neutron irradiation at NYC sea level for a total fluence of 2x1012 protons/cm^2 and 1x1012 neutrons/cm^2". And "7 nm SEU innovations combined with 7 nm process lead to approximately 10X reduction in CRAM SEU cross Section relative to Xilinx 16nm Kintex UltraScale+™ CRAM SEU cross section. In addition, zero uncorrectable events were observed in 7nm UltraScale CRAM and URAM at sea level with built-in interleaving scheme. And 0.2% of events are uncorrectable in BRAM".

Xilinx also provides a pre-verified Single Event Mitigation firmware (XilSEM). XilSEM on Versal™ ACAP performs CRAM error detection and correction (via integrated support for ECC and CRC). Single-bit errors are correctable thanks to ECCs, while multi-bit ones are only detectable. During another experiment, using NYC sea level neutron flux of 12.9 $neutron/cm^2/hr$, "no uncorrectable CRAM multiple-bit upset (MBU) events (i.e., within the same word) were observed and reported by the XilSEM tool. This result confirms that the built-in interleaving solution is more than adequate for this generation" [14].

2.2 Fly by wire control system

In fly-by-wire control systems, the input signal is transmitted to the actuator electronically rather than mechanically. These kinds of systems overcame traditional mechanical ones for several improvements concerning their weight, reliability, and maintenance (since there are fewer moving parts), precision of the response, safety, and, above all, reduction of pilot workload.

As explained in [20], the pilot has basically 3 functions:

- sensing
- regulation
- decision making.

The first two functions can be defined to be "high bandwidth", because there is little to think about, while decision making is "low bandwidth" since it requires more concentration.

FCC can allow moving the high bandwidth functions to the autopilot, since they are just like reflexes, in such a way that humans' jobs become easier. Of course, computers are not intelligent and do not possess emotions, but, reducing their domain to just some algorithmic functionality, they can achieve operative equivalence.

Basically, in this kind of system, the electronic signal is transmitted to an actuator, for example, a Direct Drive Valve (DDV), controlling the fluid flux and allowing to move the RAM (the aerodynamic surface).

In order to monitor the position of the aerodynamic surfaces across the aircraft, several sensors are placed, for example, LVDT (Linear Variable Differential Transformer) sensors [23], depicted in Figure 2.9.

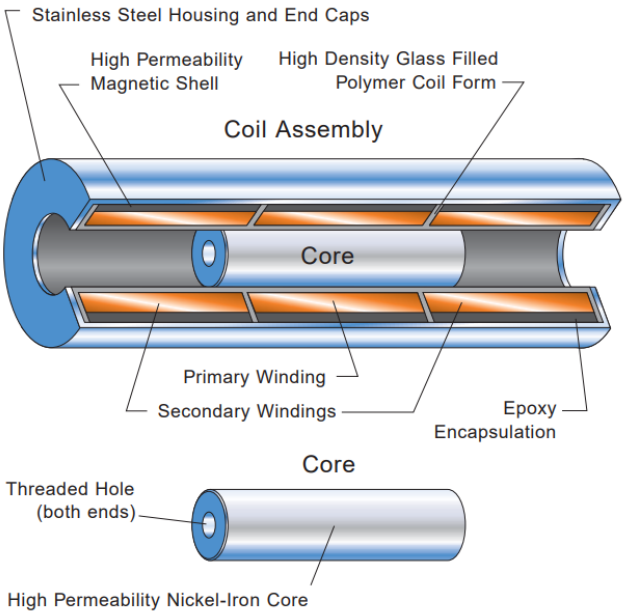


Figure 2.9. LVDT sensor: "the primary winding is illustrated in the center of the LVDT. Two secondary coils are wound symmetrically on each side of the primary coil as shown for “short stroke” LVDTs or on top of the primary coil for “long stroke” LVDTs. The two secondary windings are typically connected in “series opposing” (Differential)", [23]

The primary winding is excited by a sinusoidal signal. The secondary coils provide a differential output, whose value depends on the position of the core (which strongly affect the coupling), allowing a very high precision of position detection.

In particular, for the FCC considered in the following sections, these sensors provides a ratiometric feedback[2], necessary to stabilize the control loop. For each actuator, LVDT sensors are used to monitor both DDV and RAM position, and the DDV valve is controlled through an H-bridge, that allows the current to flow in both directions through the actuator coil.

The flight control computer is divided in a series of electronic boards with different purposes. In the next paragraphs the various components of the FCC are quickly described, and a schematic drawing is also reported in Figure 2.10 in order to provide a better understanding.

[2]Ratiometric feedback means

$$V_{fb} = \frac{V_1 - V_2}{V_1 + V_2} \tag{2.1}$$

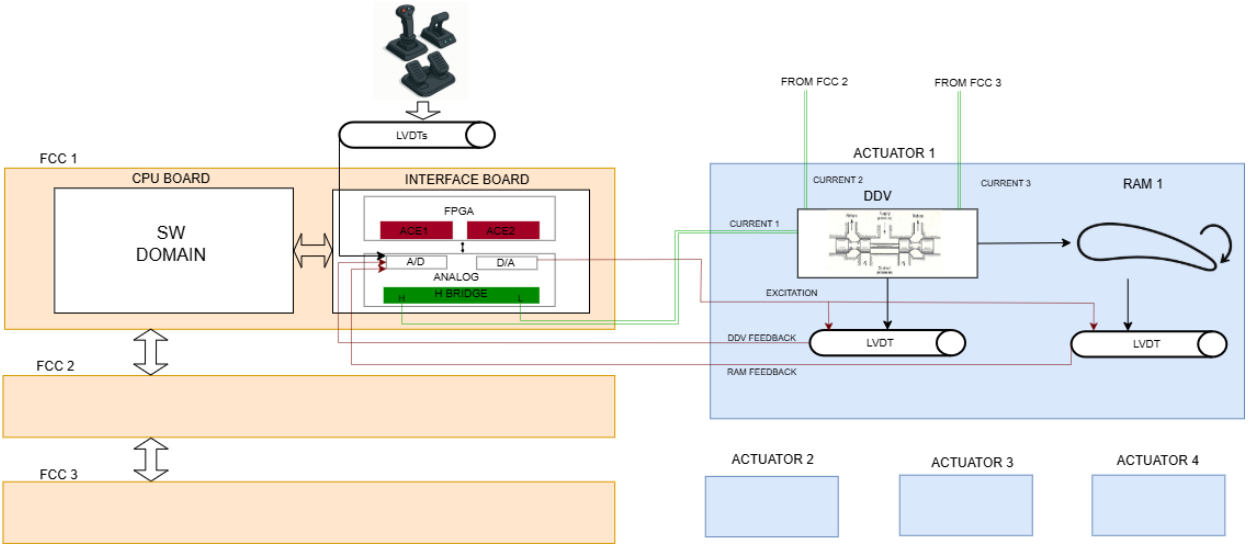


Figure 2.10. Scheme of FCC's role in airplane flight. Actuator control electronic (red rectangle) is located in the FPGA Section of the interface board.

2.2.1 The FCC

The FCC receives an external command every 10 ms, but reads the sensors and updates the PWM value to control an hydraulic valve every 100 μs.

This kind of safety-critical apparatus are usually redundant: that means that the same FCC's hardware has 3, 4 or more copies that operates receiving input from the same number of copies of sensors. If some fault is detected from the software, the failed one can be turned off, giving more weight to the other ones. In the actuator, more than one coupled coil is present, one for each copy of the FCC, and their contributes are summed.

2.2.2 The actuator control electronic (ACE)

The case study algorithm taken as example is reported in Figure 2.11.

The ACE can be divided into 3 main steps: the first one that receives the `SETPOINT` command and, given the difference between the desired RAM position and the actual one (feedback), performs a filtering through the first Proportional Integral Derivative (PID) module (that will be called "controller block" later on).

The filtered output is the translation in DDV domain of the needed RAM movement, and so this time is the difference between that signal and the DDV feedback to be filtered and to proceed into the third section of the graph.

The last step is in the current domain, and after another difference with the `ACTUATOR CURRENT` signal, the execution proceeds through the last PID module, in order to be ready for the PWM generator.

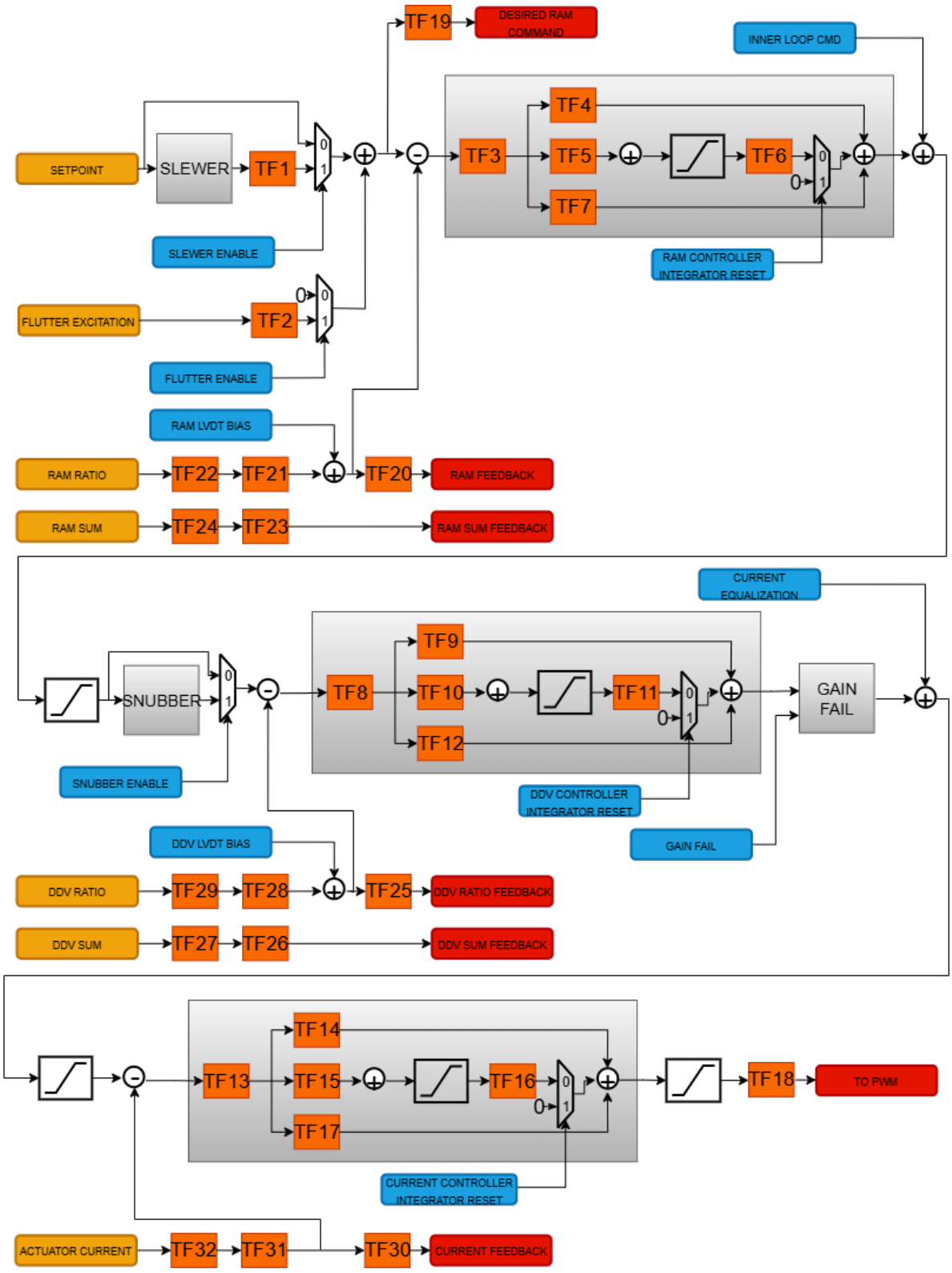


Figure 2.11. DFG of the algorithm (Simulink model). Each small square (TFx) is an IIR filter of second order. In yellow are represented inputs, in red outputs, while blue labels correspond to tunable parameters.

This system has 7 main inputs:

- SETPOINT, which is the position toward which the RAM has to move. This command is generated by the human input and is elaborated by the software domain;
- FLUTTER EXCITATION;
- RAM RATIO, which contains information about the RAM position;
- RAM SUM, that corresponds to E1+E2, and should be constant. If it is not constant, probably something is malfunctioning;
- DDV RATIO;
- DDV SUM;
- ACTUATOR CURRENT, which is the measured current value inside the DDV actuator.

Furthermore, many other control parameters can be used in order to activate/deactivate some functionality or to add some biases. Given this explanation, here is a list of the tunable parameters (they can be tuned by the software domain):

- SLEWER ENABLE: the slewer is a block that operates directly on the input, since its sample frequency is 100 times smaller and its variation may be seen from the system as a step, the slewer can help to mitigate this effect;
- SNUBBER ENABLE;
- FLUTTER ENABLE;
- GAIN FAIL: the sum of the gain fails in the various copies of the FCCs is usually 1, and is equally distributed between the working ones;
- RAM LVDT BIAS, that can be helpful to neutralize bias differences between the copies of the same sensor;
- DDV LVDT BIAS;
- RAM CONTROLLER INTEGRATOR RESET, to reset the RAM integrator loop (see Figure 2.11);
- DDV CONTROLLER INTEGRATOR RESET;
- CURRENT CONTROLLER INTEGRATOR RESET;
- INNER LOOP CMD;
- CURRENT EQUALIZATION.

As concerns the outputs, they are:

- DESIRED RAM COMMAND, which is basically the setpoint passed through the slewer block and filtered;
- RAM POSITION FEEDBACK SUM, which should be 0 and goes to the SW domain;
- RAM POSITION FEEDBACK, which gives the SW info about RAM position;
- DDV POSITION FEEDBACK SUM, which should be 0 and goes to the SW domain;
- DDV POSITION FEEDBACK, which gives the SW info about DDV position;
- CURRENT FEEDBACK AMP, which gives the SW info about H-bridge current in Ampére;
- TO PWM, which is the digital signal that controls the PWM duty cycle.

2.2.3 Parallelization for AI Engines

As previously described, AI Engines are basically vector processors, and the maximum arithmetic intensity (OP/Byte) is obtained through massive parallelization. The most efficient examples suitable for these architectures are vector (or matrix) multiplications, and this is the reason why they are so widely used in machine learning (e.g., CNNs

(Convolutional Neural Networks)).

In the case of an algorithm such as the one described in Section 2.2.2, there is no possibility of parallelization on the input data (for example, providing multiple input samples and elaborating all of them together), since the whole DFG has to be traversed before reading the next input set.

Furthermore, there is a long series of IIR-order 2 filters (13 filters), over which applying some sort of pipeline is not possible since many feedback inputs are present, and it is not possible to apply a pipeline to feedback loops.

Considering this factor, this specific may not be the best kind of algorithm to run on these devices. Anyway, the focus of the thesis is to evaluate its feasibility and, given the astonishing performance of the hardware, suggest some modifications that can lead to an increase in performance or flexibility.

Chapter 3

Design and toolchain overview

3.1 The design flow

In UG1504 [10], AMD advises following a method in order to help streamline the design on Versal ACAPs.

3.1.1 Application Mapping

The prerequisite is having a good understanding of the requirements of the system, and it can be achieved with a functional behavioral model of the algorithm. The most relevant pieces of information needed about the algorithm are:

- Sampling rates
- Data types
- Storage requirements
- Compute requirements
- Graph form (branches or no, pipeline, feedback,..)

All of them have to be decided on the basis of throughput and latency requirements, maximum bandwidth, power requirements, and precision of the output (this concerns mainly the data type, but more complex data types may lead to more computational time, especially with AI Engines).

After that, the next thing to do is the so-called *Application Mapping*: it is necessary to associate each part of the algorithm to some part of the SoC.

Usually, the more compute-intensive parts are mapped to the accelerators (AI Engines, DSP Engines,...), having in mind what kind of operations they allow, and the data flows and manipulations are left to the programmable logic (PL).

For example, since AI Engines are SIMD vector processors, mapping a kernel on them is efficient only if the algorithm is parallelizable in some way. If it is not the case, the overhead in terms of clock cycles due to the data movements may neatly overcome the resource saving.

Data movement

Also, the data movement across the chip has to be properly planned: the AI Engines, for example, have a direct connection with the external DDR memory, but for large bandwidth it is more efficient to pass through the PL.

Furthermore, the PL contains Ultra RAM and Block RAM, which can be employed to build a custom memory hierarchy (that is especially effective in large matrix multiplication, for example, in Convolutional Neural Networks applying tiling techniques[16]).

Data re-usage is the key to bandwidth savings, especially considering the shared memory mechanism of the AI Engines array. Also, the cascade streaming can be exploited to save multiple reading/writing operations, if the data flow consents.

3.1.2 Performance Modeling and Simulation

In order to monitor the traffic flow performances, it is very useful to perform simulations of the NoC dataflow (for example, with AMD generator IP). In this way, the designer can understand if the throughput and latency requirements of the systems are met. Some tutorials are accessible on GitHub from the User Guide.

3.1.3 HW only vs embedded systems

This phase is different depending on the basis of the system type.

For hardware-only systems, embedded or external processors are not used, but only the PMC (*Platform Management Controller*) is used to program and control the design.

As concerns embedded systems, the starting focus should be the type of accelerators that are to be used (PL or AIEs). Generally speaking, both PS, PL, and AIEs can work together, according to the application mapping, which has to be performed manually.

Both AIEs and PL will be controlled by the software stack. The PMC boots the device, and different boot modes are supported. The Arm Cortex-R5F works in a low power domain (LPD), is also ASIL-C safety compliant, and is a good choice if looking for determinism and real-time applications. For more complex applications, the Linux OS can be the best solution, but it requires the full-power domain (FPD).

3.1.4 Power and Thermal Planning

Since restarting from zero is very time and money consuming, inserting power and thermal considerations in the board planning phase is generally a good idea.

AMD provides the Versal adaptive SoC Power Design Manager (PDM) tool, in order to run simulations and obtain power reports, which may help to understand if some constraints are not met.

Thermal simulations allow to get θ_{JA}, and in addition to the maximum ambient temperature, that information allows to evaluate the worst-case power consumption.

3.2 Tools: Vitis Model Composer

"AMD Vitis™ Model Composer offers a high-level graphical entry environment based on Simulink® from MathWorks for simulation and code generation of designs that include AI Engine, HLS, and RTL components. For more information, see the Vitis Model Composer User Guide (UG1483 [9])" (UG1076 [6]).

This tool takes on another level of abstraction the development of a complex system. It is still needed to write by hand the kernel C++ code for the AI Engine, but it is capable to generate automatically the graph description, mapping the kernels on the array and organizing the data movement between them.

As concerns the Programmable Logic (PL), a whole library of basic blocks is already included in the tool; however, it is possible to insert custom HDL components (even if they should be a single entity) or generate HDL code starting from MATLAB functions.

It also supports HLS, so if the user wants to further raise the abstraction level, it is possible to include custom HLS blocks.

3.2.1 HDL - AIE connections

To exploit both AIEs and PL in the same project, some considerations have to be made:

- all the PL blocks have to be inserted in a subsystem, which has to contain only blocks of the specific HDL library;
- all the AIE kernels have to be grouped in another subsystem, containing only blocks of the AIE library;
- the link between the two subsystems has to be managed passing through "AIE to HDL" or "HDL to AIE" blocks, which are capable of managing AXI4 interconnections by means of a handshake protocol.

These are summarized in Figure 3.1

Figure 3.1. Interconnection between AIE and HDL blocks, [9]

All the simulations are bit-accurate, which means that the results obtained are not approximated but are exactly the ones generated by the hardware. The only problem is that AIE kernel simulations are not cycle accurate: the latency of the kernels and of data transfer between AIE and PL is not considered by the simulator. This does not allow the user to estimate easily latency performances, and some measurements have to be performed on real HW.

3.2.2 Features of the tool

Model Composer does not stop at a simple behavioral simulation, but is capable of also performing a series of steps that normally have to be performed by hand, guiding the user all the way through the flow up to the bitstream generation.

The tool contains a constraints editor, in which it is possible to control which tiles have to be employed to run kernels, FIFO depths in blocks at AIE or PL input, and kernel Runtime Ratios (fraction of available execution cycles that are used by the kernel during operation).

DMAs

To test the system on hardware with real samples, an input and a golden output files are added to the bitstream, with HLS DMAs responsible for the data movement from DDR memory to input ports and for checking the real output against the Simulink golden reference (it should not happen, but it is possible that on real HW the actual behavior does not correspond to simulations). These two DMAs should be found in the Vivado project's block diagram (as in Figure 4.36).

HLS codes for the DMAs are located in the `datamover/src/` directory, and the system behavior is controlled by PS.

By default, in automatically generated PS code (`host.cpp`), DMAs are programmed and only then the AIE graph is initialized and started (an example is reported in appendix B.1), but to measure latencies (maybe modifying the block diagram on Vivado), the user can customize the C++ code as please. However, in this case, it is necessary to repeat the `.elf` creation with makefile commands (not relying on automatic code generation offered by the tool, since the `host.cpp` will be overwritten).

Table 3.1 summarizes how the tool helps.

	User (by hand)	Model Composer
Kernels (.cpp and .h)	x	
Block diagram (PL & AIE)	x	
HDL description & IP creation (PL)		x
AI subsystem graph (.cpp and .h)		x
libdaf.a		x
connectivity file (.cfg)		x
input.txt / reference_output.txt (from Simulink)		x
datamovers (s2mm and mm2s) for in/out		x
platform (.cpp and .h)		x
host.cpp		x
lscript.ld		x
sd_card.img / BOOT.bin		x

Table 3.1. Table summarizing main activities from design to bitstream generation. After the first generation, every source can be customized (but output files have to be generated via bash).

3.3 The AXI4-Stream handshake

The communication between the AIE and the PL, in both directions, uses the AXI4-Stream protocol. For each 32-bit data transfer, the TVALID and TREADY signals are used to manage the handshake between the transmitter and the receiver.

UG1483 [9] reports the following:

"The TREADY/TVALID handshake is a fundamental concept in AXI to control how data is exchanged between the master and slave allowing for bidirectional flow control. TDATA, and all the other AXI4-Stream signals (TSTRB, TUSER, TLAST, TID, and TDEST) are all qualified by the TREADY/TVALID handshake. The master indicates a valid beat of data by the assertion of TVALID and must hold the data beat until TREADY is asserted. TVALID once asserted cannot be de-asserted until TREADY is asserted in response (this behavior is referred to as a “sticky” TVALID). AXI also adds the rule that TREADY can depend on TVALID, but the assertion of TVALID cannot depend on TREADY. This rule prevents circular timing loops."

The timing diagram in Figure 3.2 provides an example of the TREADY/TVALID handshake.

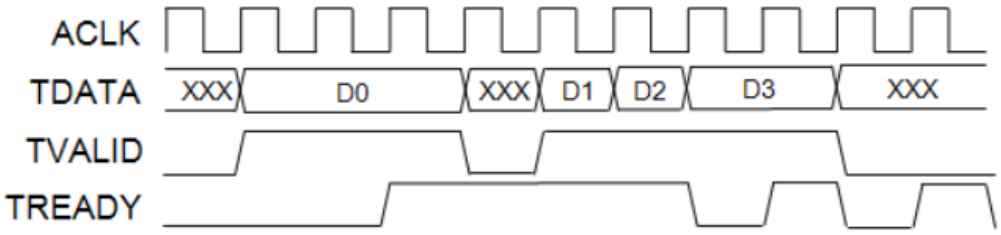


Figure 3.2. Handshake mechanism example [9].

3.3.1 AXI4 connections in complex DFG

Usually, data coming from a PL kernel is processed by a single AI Engine (AIE) kernel. In such cases, the simple sequential nature of the dataflow can be implemented using straightforward AXI4-Stream interfaces.

However, when the graph becomes more complex—for example, when the output of multiple AIE kernels must be combined in the PL—it is necessary to ensure that all input data remain valid until all required data are available. To achieve this, the TREADY signal generated in the PL should be the logical AND of all TREADY signals from the PL output interfaces, as well as all TVALID signals from the input interfaces of that Section of the algorithm.

Some hints on how to manage this kind of MISO handshake protocols can be found in [13], where the same concepts have been applied in the context of latency-insensitive protocols (even if in the context of this thesis, the complexity of the suggested adaptive protocols is completely exaggerated given the requirements of the application).

In Figure 3.3 is reported a simple example is reported, in which two input vectors are sent to PL with an AXI handshake protocol just to be summed together.

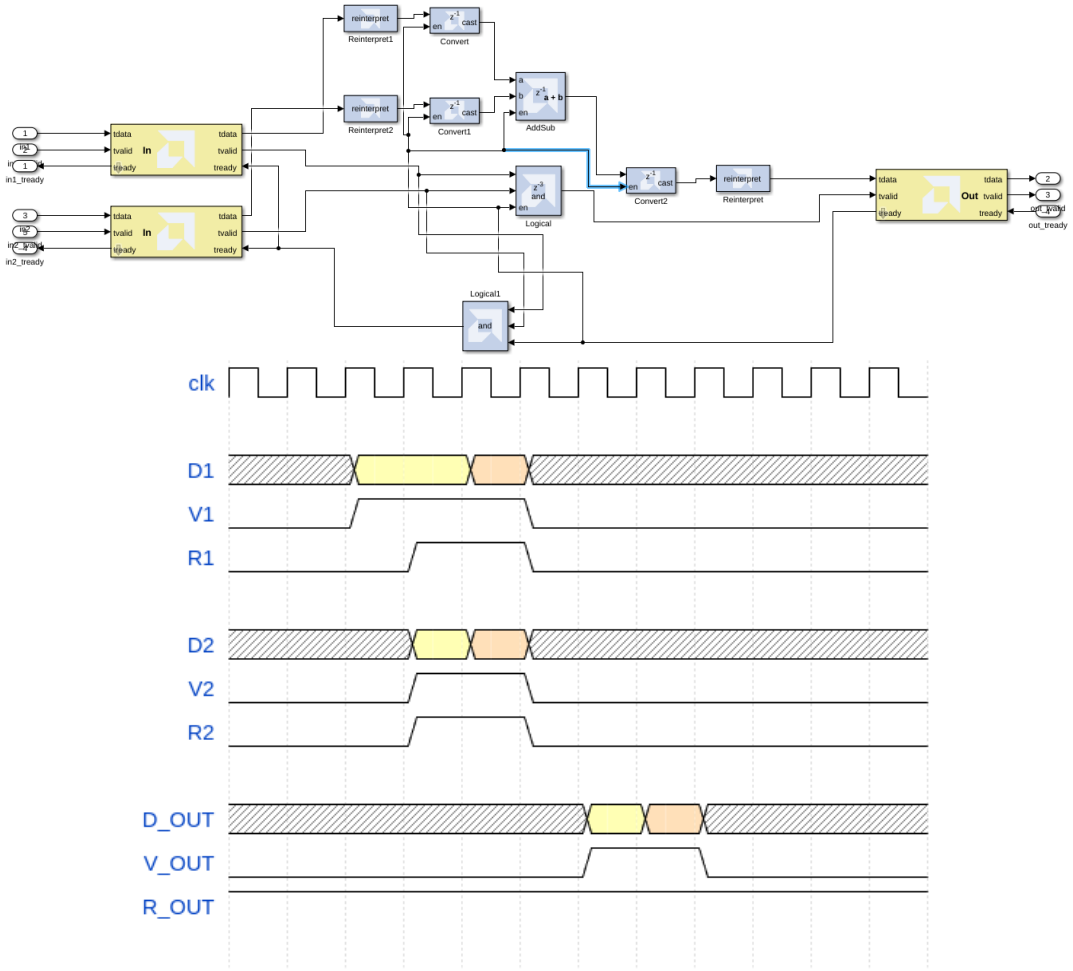


Figure 3.3. Simple PL subsystem with two AXI inputs and one output and its waveforms. Each color represents a group of data processed in the same temporal window or computational epoch.

A first important thing that needs to be underlined concerning complex DFGs is the TVALID pipeline: TVALID latency should last exactly as the main data pipeline (in Figure 3.3 it's 3 cycles).

Then, here is a list of logic functions explained to produce output TVALID, input TREADY, and the ENABLE of the logic blocks:

- Since the pipeline has to be propagated only if all the necessary inputs are valid,
$$TVALID_{out} = \prod_{i=1}^{n} TVALID_i$$
- As stated in [9], it is just the TVALID that cannot be generated from the TREADY, but not vice versa, so as TREADY is used
$$TREADY_i = (\prod_{i=1}^{n} TVALID_i) \cdot TREADY_{out}$$
In this way, data are sampled when the output is ready to receive them.
- The problem arises when the output is no longer ready to receive data: since in the pipeline there are already three valid samples that need to be stopped to not lose

information, all the registers need an ENABLE, which should be

$$ENABLE = TREADY_{out}$$

The waveforms registered on the VCK190 evaluation board with ILA are reported and explained in Section 4.3.2.

Unfortunately, on Model Composer, external IPs inclusion in the design is not so easy, anyway on Vivado there are some Xilinx IPs that may help to fork or combine more AXI4 Streams, for example the AXI4-Stream Broadcaster and the AXI4-Stream Combiner (this one reproduces the same mechanism explained in the previous paragraphs)[4], while the broadcaster is extremely useful when transmitting the same AXI Stream to more destinations.

3.3.2 AXI4 connections in very complex DFG

What happens if the mapping of the algorithm leads to an even more complex form of AXI-based PL subsystem?

An example representative of the algorithm analyzed in this thesis work, even if much simpler, is reported in Figure 3.4. Here, data entering the PL are both sent to an AIE kernel and kept inside the PL, and the TVALID should be kept high at least until the output of the AIE is ready, to maintain data coherency.

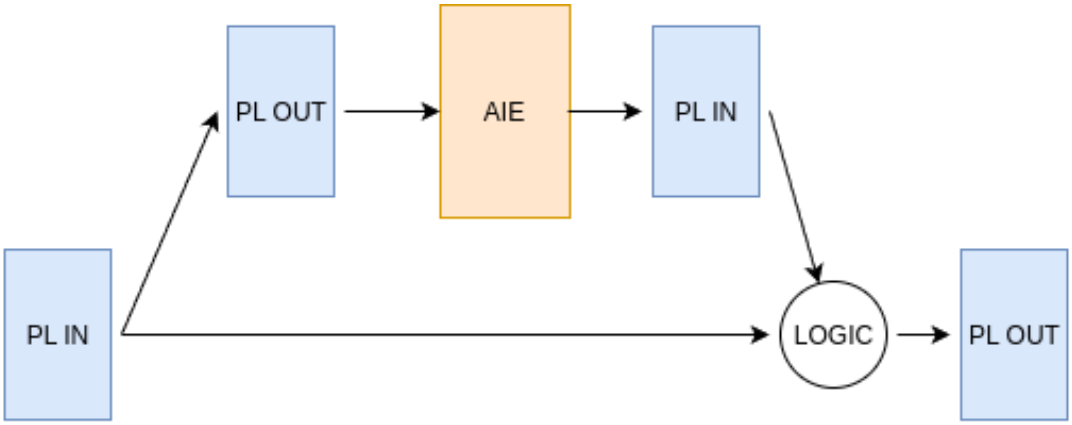


Figure 3.4. Example of complex AXI handshake protocol.

Sampling the input when both TVALID and TREADY are high is no longer sufficient, as the execution dependency on the upper branch must also be considered.

The canonical and most correct approach to this problem involves the use of Vivado IPs: in such a case, a broadcaster distributes the input to both branches, while a combiner merges the output of the upper and lower branches.

However, this solution comes with increased complexity in both the Vivado design and the overall system architecture, and it cannot be simulated within Model Composer.

A simpler and original alternative, designed to stay entirely within Model Composer, can be applied to the specific case in which the complete graph—possibly much more complex—must process one input sample at a time.

Since the case study algorithm is structured to start processing the next sample only after the previous one has fully propagated through the entire graph, the chosen strategy is to generate a single TREADY signal. This signal is asserted only when the full computation has completed.

As a result, an *edge detector* block (Figure 3.5) is required, because the TVALID signal remains high throughout the entire graph execution, while the data must be sampled only once.

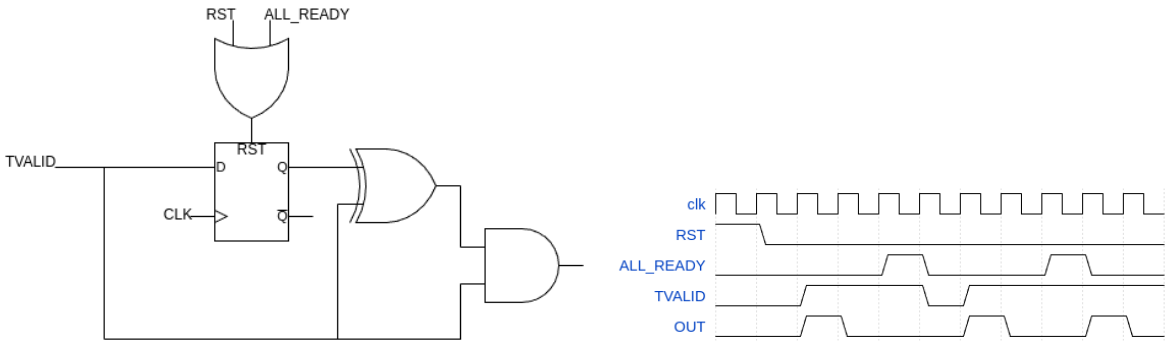


Figure 3.5. Edge detector block: it becomes high only when the input (TVALID) has a rising edge. The ALL_READY signal resets the flip-flop, allowing it to detect also situations in which the TVALID does not fall (the new data has already arrived).

3.4 The board

The chosen board in order to experiment with the adaptive engine's effectiveness on the FCC algorithm is the *VCK190 Evaluation Board* [8] developed by *AMD*, which mounts a VC1902 device.

3.4.1 The Versal Device

In particular, the complete part number of the chip is `XCVC1902-2MSEVSVA2197`, where:

- `XC` is the commercial Xilinx prefix;
- `VC` identifies the family (Versal Core);
- `1902` identifies the devices (dimension, resources,...);
- `2` is the speed grade (medium);
- `M` identifies the voltage, which is medium;
- `S` identifies the static screen (standard);
- `E` identifies the temperature grade (0 to 110°);
- `VSVA2197` identifies the package: Ball Grid Array (BGA) with 2197 balls;

The device contains 400 x AIE engines. That is precious information to be taken into account in order to properly think about how to efficiently exploit them. They are arranged as a 50x8 tile matrix, with 39 PL interface tiles [3]. With a simple calculation, the total bandwidth for Pl-AIE communication is

$$BW = f_{ck} \cdot 39 \cdot n \cdot 8 \, Bytes/s \tag{3.1}$$

where f_{ck} is 0.5 GHz (maximum PL frequency), and n is the number of streaming ports (8 for input, 6 for output).

So the actual AIE-PL bandwidth for this specific device is 1.560 TB/s for input streaming and 1.170 TB/s for output streaming.

The device contains 34 Mb of Block RAM, 130 Mb of Ultra RAM, 4xDDR Memory Controller (DDRMC) with a DDR Bus Width of 256 bits. The overall DDR4 bandwidth is 102.4 GB/s, and the overall SRAM Bandwidth is 188 TB/s.

In table 3.2 are summarized other important features of the Versal device.

AI Engines	400
DSP Engines	1968
System Logic Cells (k)	1968
LUTs	899840
Application Processing Unit	Dual-Core Arm® Cortex®-A72
Real-Time Processing Unit	Dual-Core Arm® Cortex®-R5F
Maximum I/O Pins	770
Programmable NoC Ports	28
Integrated Memory Controllers	4

Table 3.2. Most important features of the VC1902.

3.4.2 The board

In Figure 3.6, the board is reported.

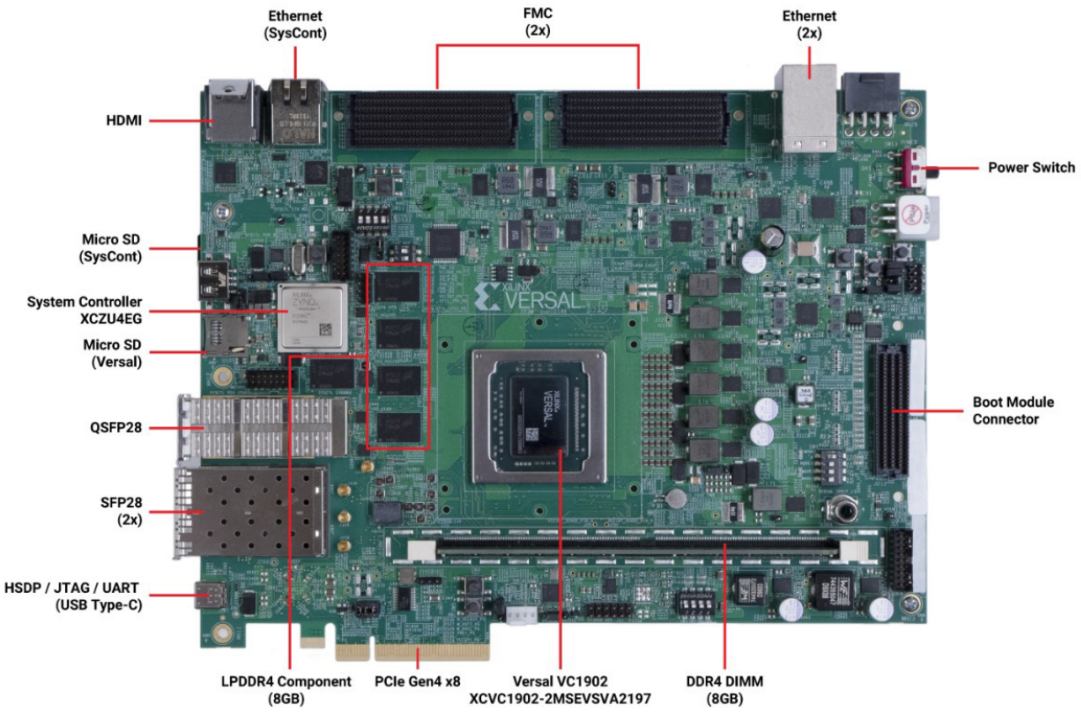


Figure 3.6. VCK190 Evaluation Board.

As concerns the configuration of the Versal device, both JTAG, QSPI32, and SD1_3.0 modes are supported, and DIP switch SW1 is responsible for the choice of the boot configuration (as shown in [8]). In particular, to boot from an SD card, it is sufficient to:

- copy a valid SD image in the Versal SD (see Figure 3.6). This can be done with the following command (on a terminal inside the folder containing the sd_card.img:

  ```
  sudo dd if=sd_card.img of=/dev/sdc bs=4M
  ```

  where "sdc" is the name of the SD. It is strongly suggested to check the right name of the SD card with the command

  ```
  lsblk
  ```

- Set SW1 to SD1_3.0 configuration (Mode Pins [3:0] = 1110).
- Power the board (SW2) or press the POR pushbutton.

Then, if the serial output has to be displayed on a terminal, it is sufficient to install *Screen* (Linux environment) and type the command

```
sudo screen /dev/ttyUSB1 115200
```

In order to connect the board to the PC through JTAG, it can be used the USB-C input port. It is just necessary to tune the SW1 in JTAG configuration (Mode Pins [3:0] = 0000). This is very useful, especially because, with a JTAG connection, it is possible to debug the device with the ILA (Integrated Logic Analyzer).

Chapter 4

Implementation

4.1 AI Engines employment for the ACE

The first step in implementing the algorithm on this SoC is mapping: it is important to identify which parts of the graph are suitable for execution on the AI Engines.

Based on the considerations discussed in sub Section 2.2.3, and intending to explore the technology as well as evaluate latency for this class of algorithms, the approach is to focus on repetitive patterns within the algorithm. These patterns are typically the computationally intensive parts of the graph, preferably those that can be parallelized.

The first obvious choice, looking at Figure 2.11, is the IIR filter order 2, which appears to be repeated 32 times in the computational graph. An alternative could be a group, for example, 8 IIR filters together in a single kernel: in this way, the parallelism would have been much better exploited. The only problem with this idea is that the longest chain of filters in sequence is 13 filters, so it is not possible to group 8 filters at a time because data dependencies would not be respected.

A first attempt was therefore made with a single second-order IIR filter.

4.1.1 AI Engine kernel - IIR order 2 filter, stream

In Figure 4.1, the Simulink model of the filter is represented, which has been described in C++ using the proper APIs.

As shown in the Figure, the already implemented model adopts a signed fixed-point data type format. This format ensures no precision loss during intermediate computations, while the output is finally floored to a 32-bit word with the binary point positioned after 16 bits.

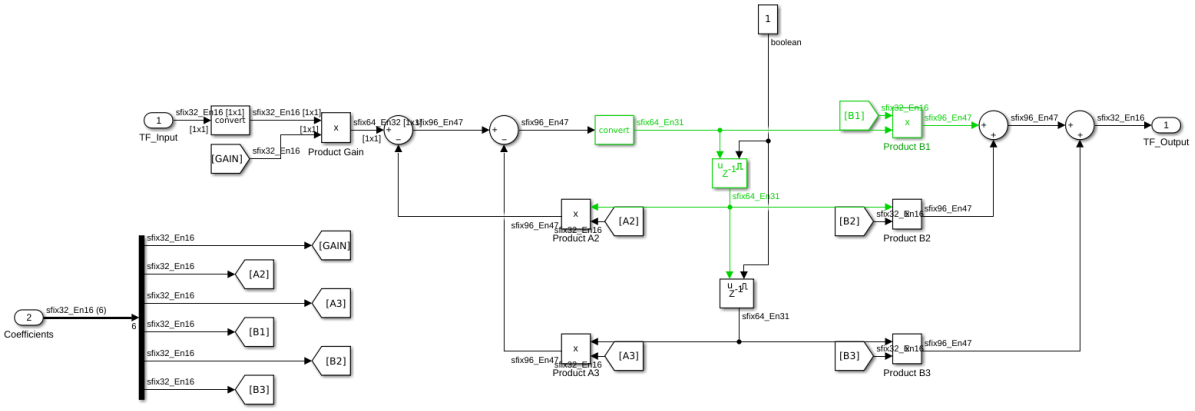


Figure 4.1. IIR filter order 2 - Simulink model.

Floating point data format for AIE

With AIE, a variable-length fixed-point representation is not possible. For this reason, Single Precision Floating Point (SPFP) or FP32 has been adopted. SPFP, according to IEEE 754, has

- 1 x sign bit
- 8 x exponent bits (biased form)
- 23 x mantissa bits (implicit 1)

Let's evaluate the pros and cons of this choice:

- For what concerns the quantization error, with floating point numbers it is not constant in absolute value, but considering that the first bit of the mantissa is always 1, the relative error is always 2^{-23}, which is worse than the fixed point one (32-bit) only if the whole dynamic is exploited efficiently in the whole algorithm.
- Numbers dynamics increase a lot in terms of orders of magnitude.
- To work in PL, it is more efficient to use fixed-point numerical formats, so a cast operation is required.
- IEEE standards impose an unbiased rounding, so no bias is integrated along the algorithm graph (as it would be, for example, if the rounding technique adopted was biased).

The kernel

On the web, several IIR filter parallelizations are available to be used on AIEs [25]. However, none of them can be used in this case, since the examples suppose that parallelism can be exploited by operating on an already given input sequence. As concerns the ACE, only one sample can be elaborated at a time.

The code has been developed from zero, and is reported in the appendix section. Some different versions of it have been studied in order to analyze performances.

Also, the assembly code generated has been compared to understand what could be the best implementation of the single filter.

The experiments on this first simple example enlighten a couple of key points to be taken into account concerning kernel development and feeding of AIE:

- simple scalar mathematical operations on floating point numbers are translated into a call of the FP32 specific function (FPADD or FPMUL), causing large overheads (50-70 cycles). If the same operation is transformed into a vectorial one, even if just the first position of the vector contains meaningful data, the jump to the FP function is avoided, and the performances increase (let's recall that no scalar floating point unit is present in AI Engines).
- if buffer data type is chosen, the AXI4 handshake protocol has to be handled manually. For this reason, it has been chosen to save coefficients as constants inside the internal AIE's memory, and partial products as a static variable (initialized to 0). In this way, just one input and one output are present, and it is possible to adopt streaming data transport (with handshake protocol automatically managed). Both implementations have been analyzed, but only the stream has a future.

To exploit internal parallelism, the IIR filter has been re-conducted as a simple vectorial operation:

$$y_n = \begin{pmatrix} x_{n-1} & x_{n-2} & y_{n-1} & y_{n-2} & x_n & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} b_2 \\ b_3 \\ -a_2 \\ -a_3 \\ b1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = x_{n-1} \cdot b_2 + x_{n-2} \cdot b_3 - y_{n-1} \cdot a_2 - y_{n-2} \cdot a_3 + x_n \cdot b_1 \tag{4.1}$$

With subsequent update of the row vector [1].

Assembly code generation and simulation

In order to proceed with code generation, the hardware selection inside the Model Composer Hub block is fundamental. If the target is just a simple AIE compilation, it is important to know at least if the chosen hardware contains AIE or AIE-ML, especially if working with floating-point data types.

The reason is that the architectures differ so much that the assembly code can be completely different. For example, AIE-ML does not support FP32 data operations naturally, and those have to be decomposed into FP16 operations "emulating" FP32. There are different kinds of emulation accuracy:

- *safe*: this is the highest level of accuracy, and the result of the emulated operation perfectly matches the normal one. To emulate an FP32 operations in this way 9 cycles of MACs of `bfloat16` are needed;
- *fast* (default one): slightly worse accuracy, but 6 cycles of MACs required;
- *low*: worst accuracy (however, it's better than `bfloat16`), but only 3 cycles of MACs.

[1] the value x_n is the input value multiplied by g. In this way, when the row value is updated, x_{n-1} is already the one multiplied by g.

The floating-point accuracy mode can be set through the following pre-processor directives:

```
-DAIE2_FP32_EMULATION_ACCURACY_FAST
-DAIE2_FP32_EMULATION_ACCURACY_SAFE
-DAIE2_FP32_EMULATION_ACCURACY_LOW
```

and has to be set on a single kernel basis.

The AIE code verification is composed of three phases:

1. Compiling the AI Engine graph design.
2. Running simulation using the AI Engine simulator.
3. Verifying the simulation results by comparing the output with the golden reference output (generated by Simulink).

If the directive is not properly set, the third phase of the code verification compares AIE results obtained with the default *fast* accuracy, with normal FP32 Simulink's result, and this leads to a result mismatch (even if probably in proximity of mantissa's LSBs) [2].

The difference between a fast accuracy mode IIR filter order 2 and the same filter generated for a standard AIE tile can be 155 instructions vs only 80. In order to work with SPFP numbers, given the obtained result, it would probably be more efficient to use normal AIE.

Test and comparison

The code has been simulated with Simulink, and SPFP results compared with the ones obtained by the original model. The testbench prepared is reported in Figure 4.2.

[2]This behavior was not well documented. After opening a ticket with Xilinx support, their response was: "We are also discussing with the development team to document this behavior for cases where simulation mismatches occur on AIE-ML devices, so that users can be aware and apply the pre-processor flag themselves."

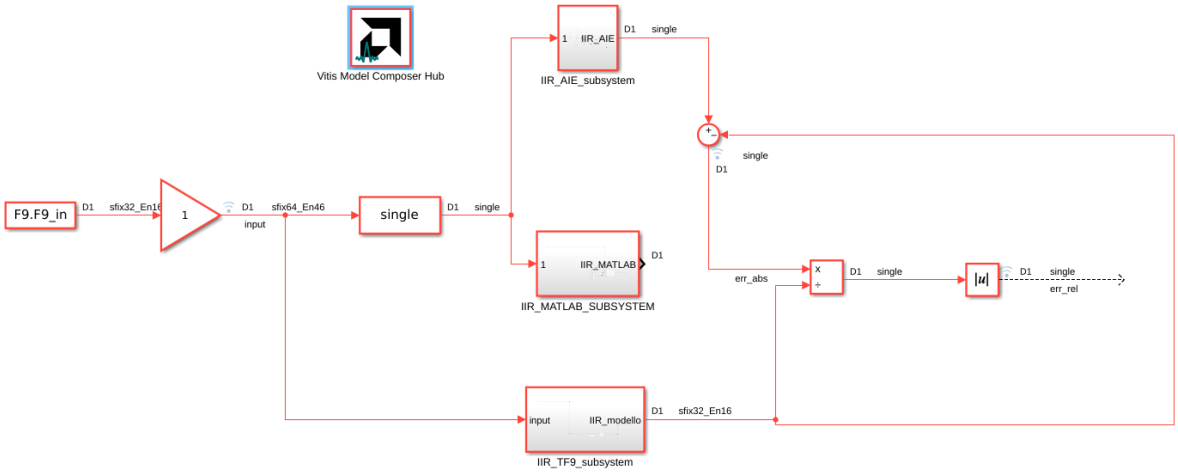


Figure 4.2. Simulink model for the testbench of the single kernel.

As a device, initially the `ve2302` was selected, containing a 17x2 matrix of AIE-ML [3].

The filter TF09 of the Actuator Control Loop has been selected as a subject for the test. The same input has been fed to both the AIE kernel, the MATLAB standard filter block, and the original fixed-point filter. Then, absolute and relative error between AIE and original solutions has been evaluated.

The MATLAB block and the AIE kernel did not present any differences, while, given the dynamics of the input, no noticeable absolute error was detected between SPFP and fixed point models, as shown in Figure 4.3.

[3]Initial tests were performed supposing to use the Trenz TE0950 evaluation board [24]; however, due to bureaucratic issues, the hardware that eventually arrived for testing was the VCK190 evaluation board. As a result, the necessary kernels were also generated for the standard AIE architecture (see Table 4.1).

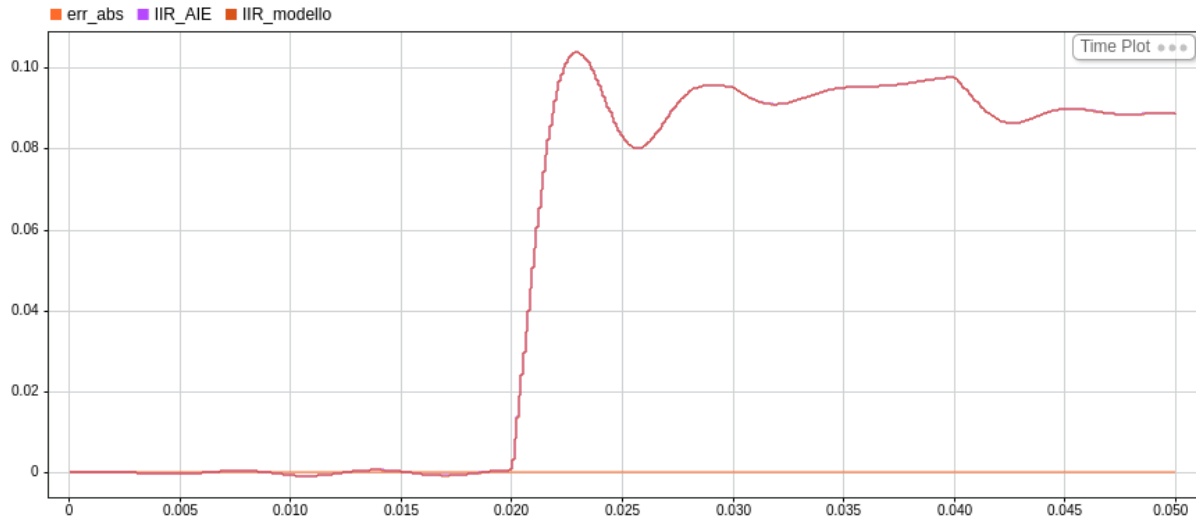


Figure 4.3. Simulink testbench results.

Analysis

Thanks to Model Composer, it is possible to have a look at the generated assembly code.

The obtained latency for the final kernel (code in C.3) is 155 cycles (which means 155 ns). The main problem of this kernel, looking at the assembly code, is that, even if some parts are quite dense (let's recall that the AIE are VLIW processors, able to issue up to 7 instructions per clock cycle), when it comes to sum together the 5 components of the vector data dependencies has to be respected. For this reason, several NOPs are allocated since the processor architecture is pipelined, and at least 4 cycles are needed until the results can be re-utilized for another calculation. The portion of the assembly code dedicated to that operation (Figure 4.4) is extremely inefficient.

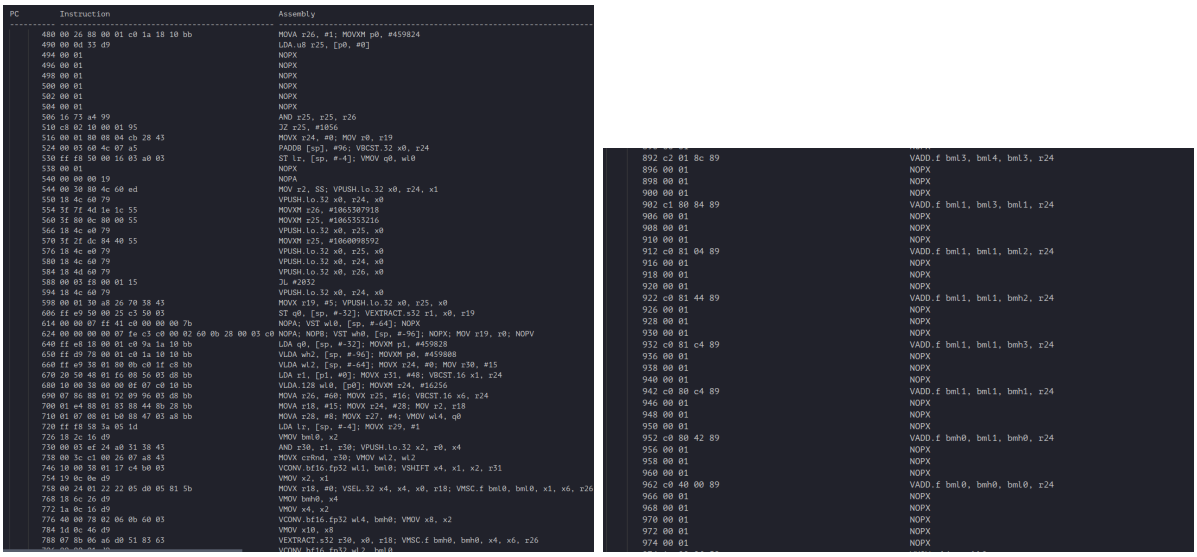


Figure 4.4. Assembly code for AIE-ML architecture. Left: dense snippet, right: sparse snippet.

It is also possible to look at the kernel location inside the array (which can be modified with constraints), as shown in Figure 4.5. Moving the cursor on the green lines, the latency from the interface tile to the destination appears: in the picture is 12 ns.

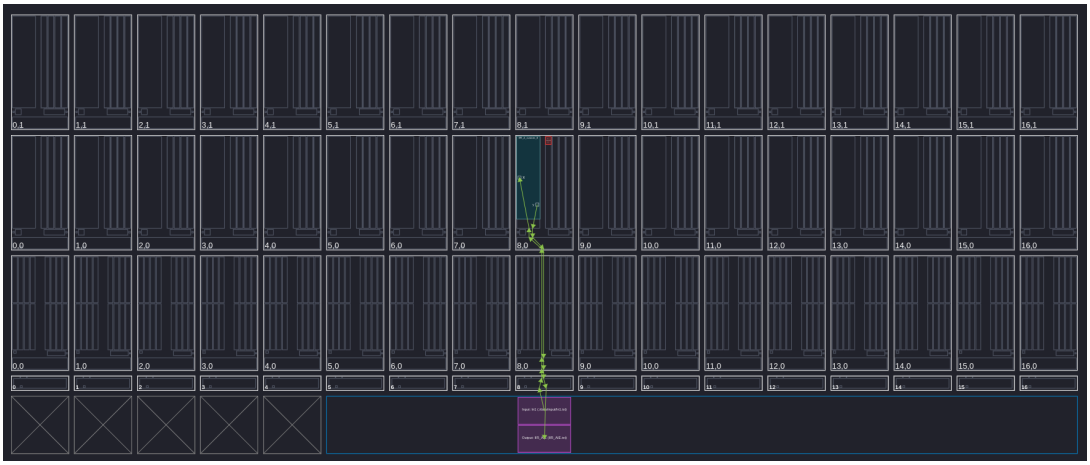


Figure 4.5. AIE-ML array. The device considered has a 17x2 AIE matrix (the lower one is the interface row).

With the trace analyzer (Figure 4.6), it is possible to have a look inside the AIE array, and to know in each clock cycle what is happening in every tile.

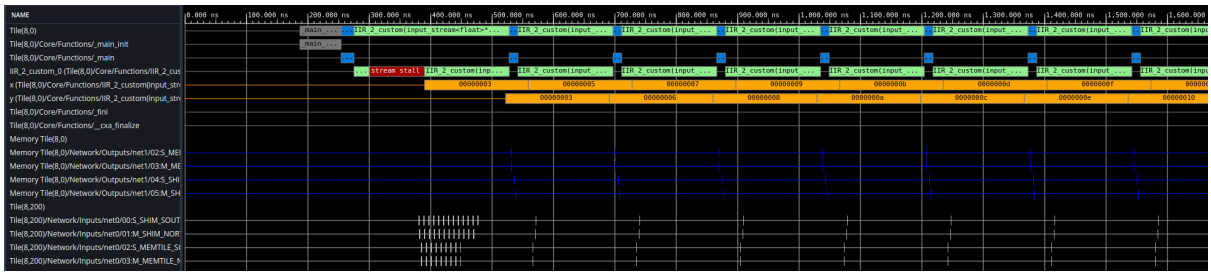


Figure 4.6. Trace results of the IIR order 2 custom function on an AI-ML tile.

After a first stall due to missing input, from the second call on, the latency is periodic. The blue segment in the first row is the main function, called every time before the IIR2 function, that has a latency of 155 cycles.

4.1.2 Enlarging the kernels

Since moving continuously from the AIE array to PL and vice versa may not be optimal and causes a non-negligible overhead, it is useful to think about how to enlarge the kernel in some way. Furthermore, if the kernel is bigger but the added operations can be executed in parallel, the overall latency may not increase linearly but in a slower way.

Enlarging the kernel merging filters

According to appendix A.1, two consecutive IIR filters can be considered as a single IIR filter, whose degree is the sum of the initial two. This idea can be used both to increase the flexibility of the system, for example, enlarging the degree of each filter, or maybe to reduce their number, since every time two (or more) of them are connected in series, they can be merged into a larger one.

This has been investigated: starting from 2 IIR filters of order 2, a larger IIR filter of order 4 has been created and, adopting a similar strategy as before, 169 cycles latency has been obtained. This value is very similar to the order 2 one, making this new approach very promising.

Enlarging the kernel in parallel

Since in the DFG a group of three filters to be executed in parallel appears many times, the next trial has been to create a kernel enlarged in that direction. In particular, since a sub-block of the graph is repeated three times, it seemed a good idea to try to fit inside an AIE as a kernel the whole block, from now on named "controller block" (Figure 4.7).

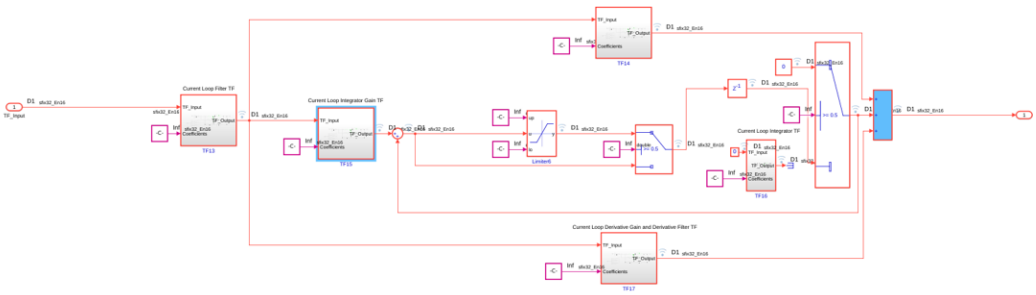


Figure 4.7. Kernel computational structure of the "controller block".

Here, after a first filter executed normally, 3 filters are computed in parallel, with a similar scheme as the one already described. Before summing all the results together, in the second lane, an integrator with a limiter is added in the code. Since the limiter implies an `if...else` statement (and so the possibility of taking jumps in the code), determinism has been momentarily put aside to investigate possible performances of this block, which resulted in 567 cycles of Initialization Interval.

Since `if` statements are not ideal when aiming for deterministic execution time, an optimal version of this kernel should be modified to always last the same, but anyway, the first version is reported in the appendix (code in C.5).

Feedback input

Another repeated block can be found in the group of 4 filters and a sum concerning the elaboration of the input, the "input block" (Figure 4.8).

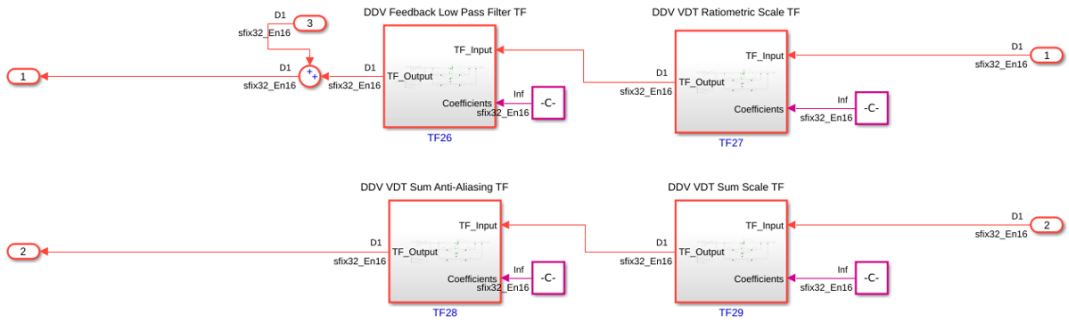


Figure 4.8. Kernel computational structure of the "input block".

This block has two inputs and two outputs, which is not a problem on standard AIE tiles: they can receive two streams as input.

As concerns AIE-ML, only one streaming input is present.

Since the first tests have been performed on AIE-ML tiles, the two 32-bit numbers have been joined to a single 64-bit word input.

This has been implemented, again exploiting a larger parallelism, obtaining an Initialization Interval of 350 cycles.

	Average latency (# assembly instructions), AIE-ML	Average latency (# assembly instructions) AIE
IIR filter order 2 - buffer input	210	NN*
IIR filter order 2 - streaming input (no code optimizations)	220	NN*
IIR filter order 2 - streaming input	155	74
IIR filter order 4	169	NN*
Controller block	567	254
Input block	350	197

Table 4.1. Average latency of implemented custom functions. NN*: Not Needed; for the normal AIE, only the useful kernels for the Actuator Control Electronic have been tested.

4.2 The whole ACE on AIE + PL

Now that the key kernels have been identified, here is a first approach to recreate the whole system. It contains 13 computational kernels:

- 3 x controller blocks,
- 3 x input blocks,
- 7 x second-order IIR filter blocks.

Data enters PL, then moves towards AIEs and comes back to PL. All the passages between PL and AIE have been arranged employing AXI4 handshakes, with the help of the custom block "edge detector" described in sub Section 3.3.1.

Figure 4.9 represents a map of the algorithm that shows how the graph is distributed across the AIE array and PL.

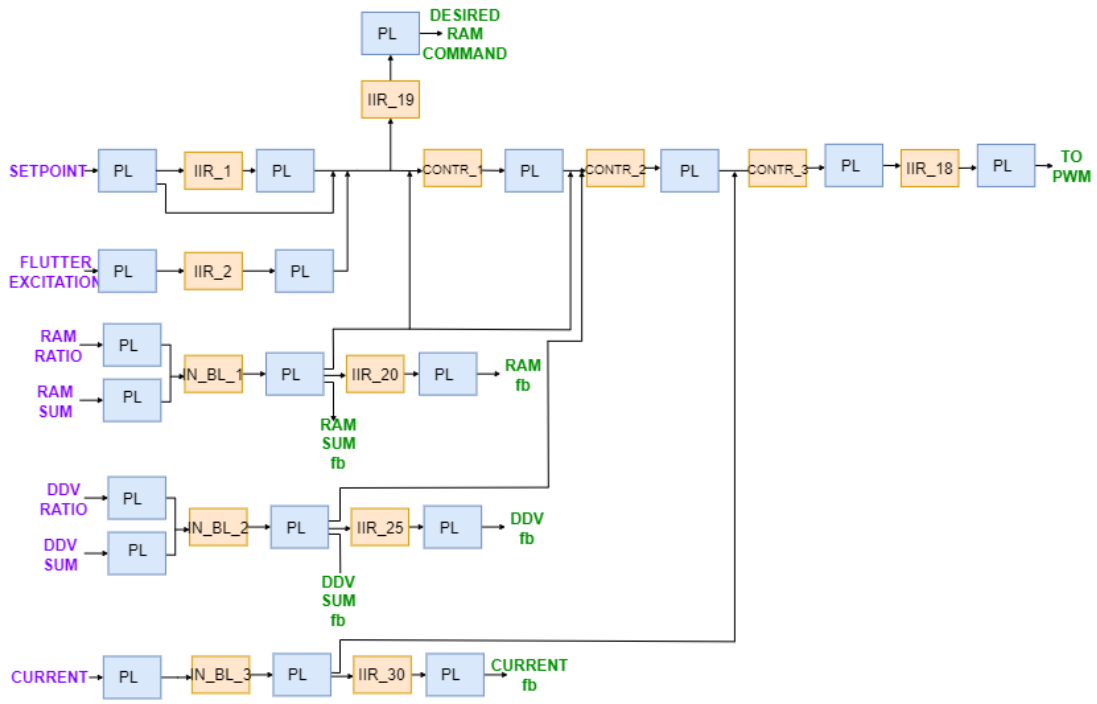


Figure 4.9. ACE functional map. Legend: purple = input; green = output; blue block = input of the PL; orange block = input of an AIE kernel.

This system is translated into two subsystems: the AIE subsystem, where the 13 kernels are instantiated, and the PL subsystem, which contains everything not implemented within the AIE.

4.2.1 AIE and PL interconnection

The overall system's block diagram is depicted in Figure 4.10.

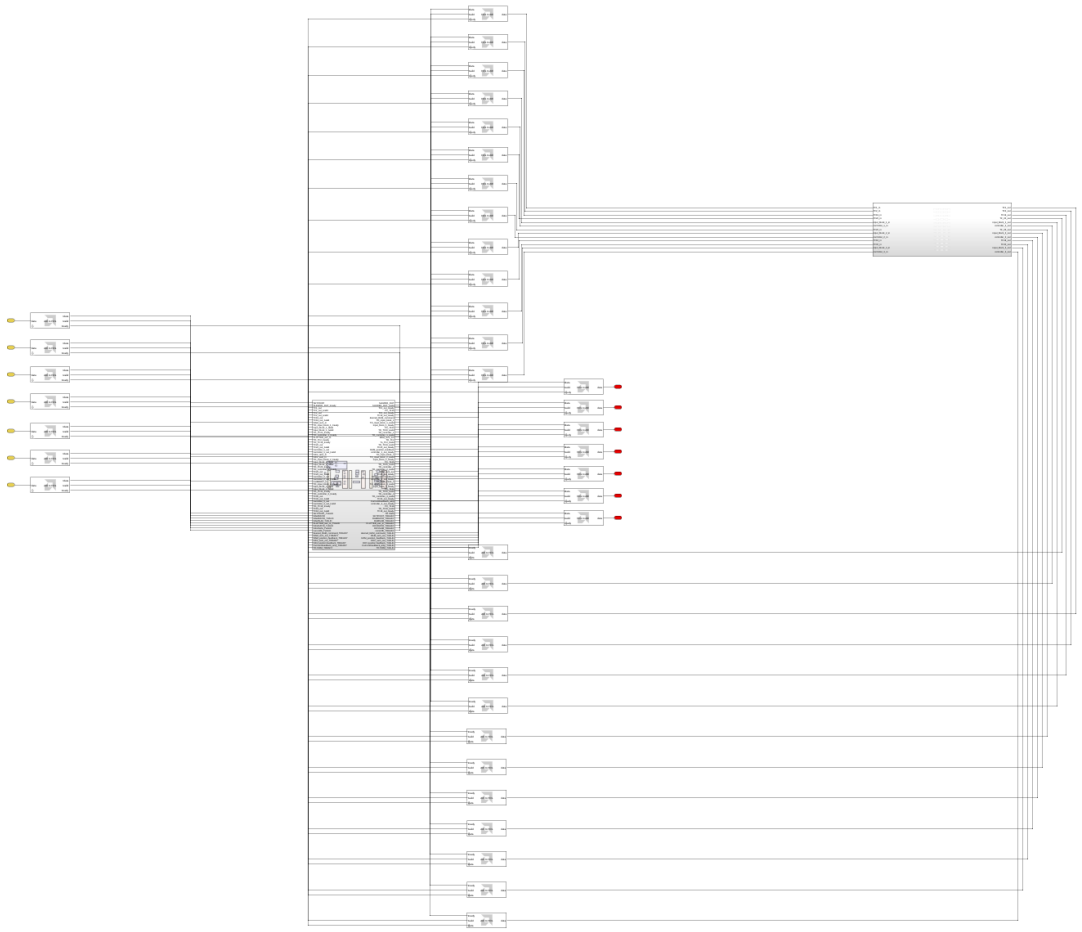


Figure 4.10. Simulink block diagram of the ACE, top level. Yellow ovals are the inputs, while red ones are the outputs. All the interconnections between AIE and PL subsystems are managed through "AIE to HDL" and "HDL to AIE" blocks, which represent AXI4-Stream handshakes.

Since on the AIE array 13 kernels are instantiated, 26 AXI-Stream interconnections are used to connect the AIE array and the PL. The problem is that in this system, only one sample will be sent to AIE for elaboration at a time, but as shown in Figure 2.3, AIE-PL streaming communication interfaces are 64-bit wide. If just one 32-bit sample is sent, the interface waits for the second one to arrive, and so the whole graph is paused. To overcome this problem, the AIE kernels have been modified in order to work on 64-bit input words, of which just the lowest 32 are the real input. The first thing performed in the kernel is an unmasking of the input, while the last is the composition of a 64-bit output word, of which just the lowest 32 are valid.

This may seem a large number of interconnections, but considering that in the `xcvc1902`'s AIE array there are 39 PL interface tiles, with a total of 234 32-bit AXI4-Stream input and 312 32-bit output, it is less than 6% of the overall resources.

4.2.2 AIE subsystem

The AIE subsystem is pretty simple: the graph is composed of just 13 kernels in parallel, each of which contains the C++ code reported in the appendix (properly modified in

order to host the correct filter(s)).

Each kernel input and output pass through a PLIO block, in which it is possible to set the interface frequency.

A picture of the AIE subsystem is reported in Figure 4.11, which in the AIE array is translated into the representation in Figure 4.12.

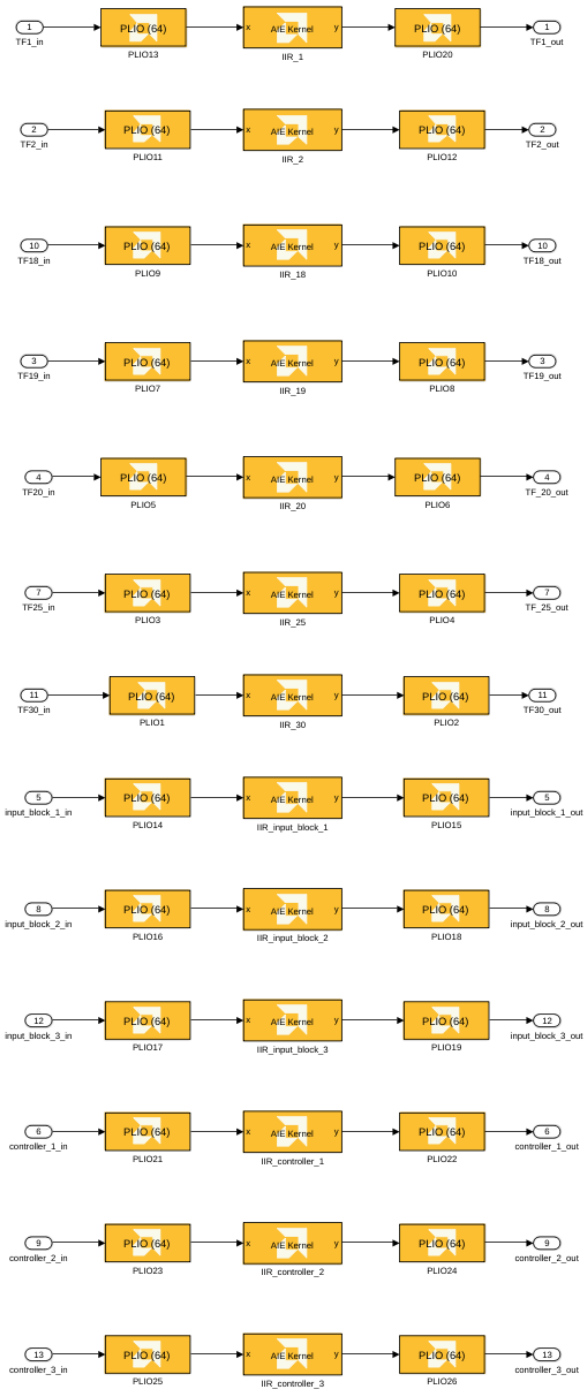


Figure 4.11. Simulink block diagram of the AIE subsystem.

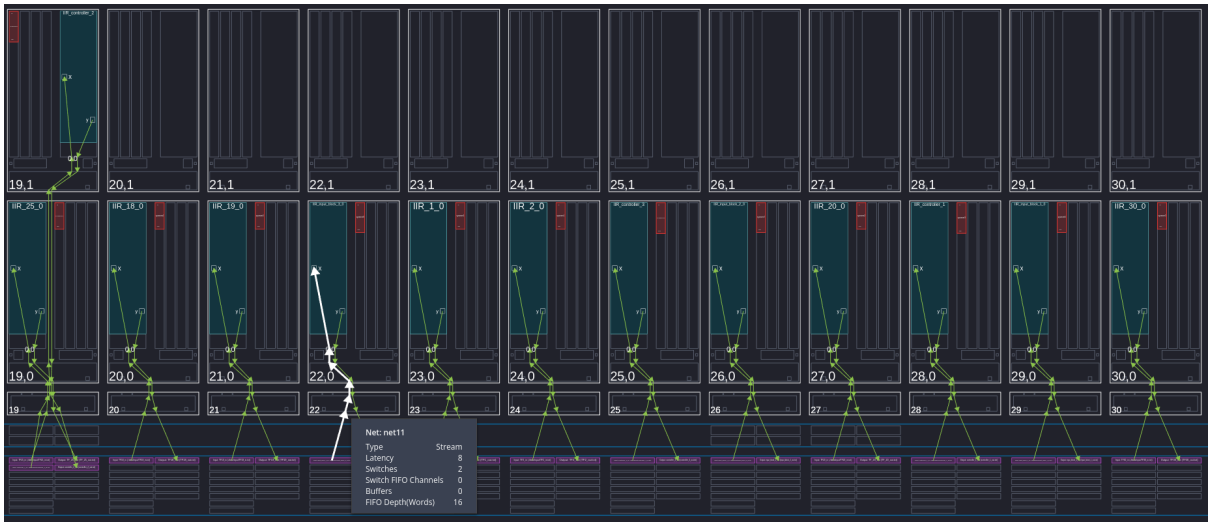


Figure 4.12. AIE graph allocation on the array. Red rectangles are the used portions of the memory, while green arrows represent the data transfer.

As shown in Figure 4.12, the graph is mainly allocated in the central columns of the first row of the array, and data transfer from/to the interface tiles lasts 8 ns for what concerns the first row kernels and 12 ns for the ones located in the second row.

An AIE simulation has been performed, and the trace analyzer results are reported in Figure 4.13. However, since this simulation does not take into account the PL latencies but feeds the array as if all the inputs are furnished in a stream way (Figure 4.14), this is not what happens during the real execution, where only one sample is elaborated at a time.

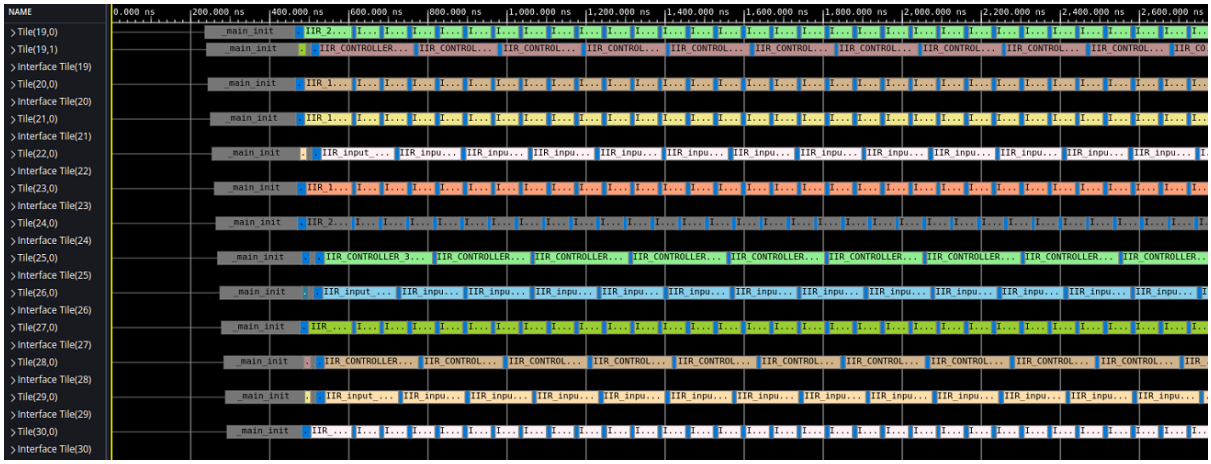


Figure 4.13. Trace analyzer results for the AIE array.

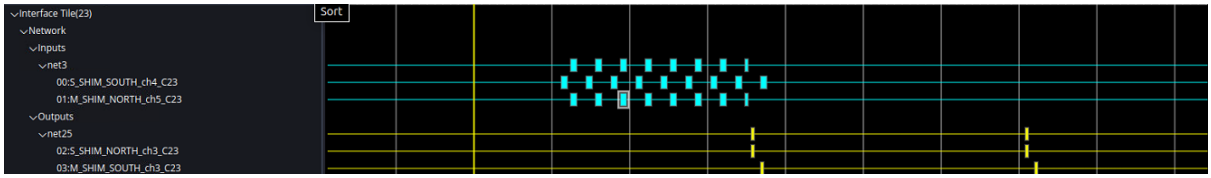


Figure 4.14. Feeding of array tile 23.

As concerns kernel latencies, they are the ones reported in table 4.1 (the table ones do not consider the initial setup). The small "main" calls (blue) between kernel executions are given by data transfers.

4.2.3 PL subsystem

The most challenging part of the design, once all the kernels were functioning as expected, has been the PL subsystem — particularly the complex mechanism chosen for managing handshakes.

The subsystem, which is reported in Figure 4.15, contains 20 AXI-Stream inputs (7 from the external world and 13 from AIEs).

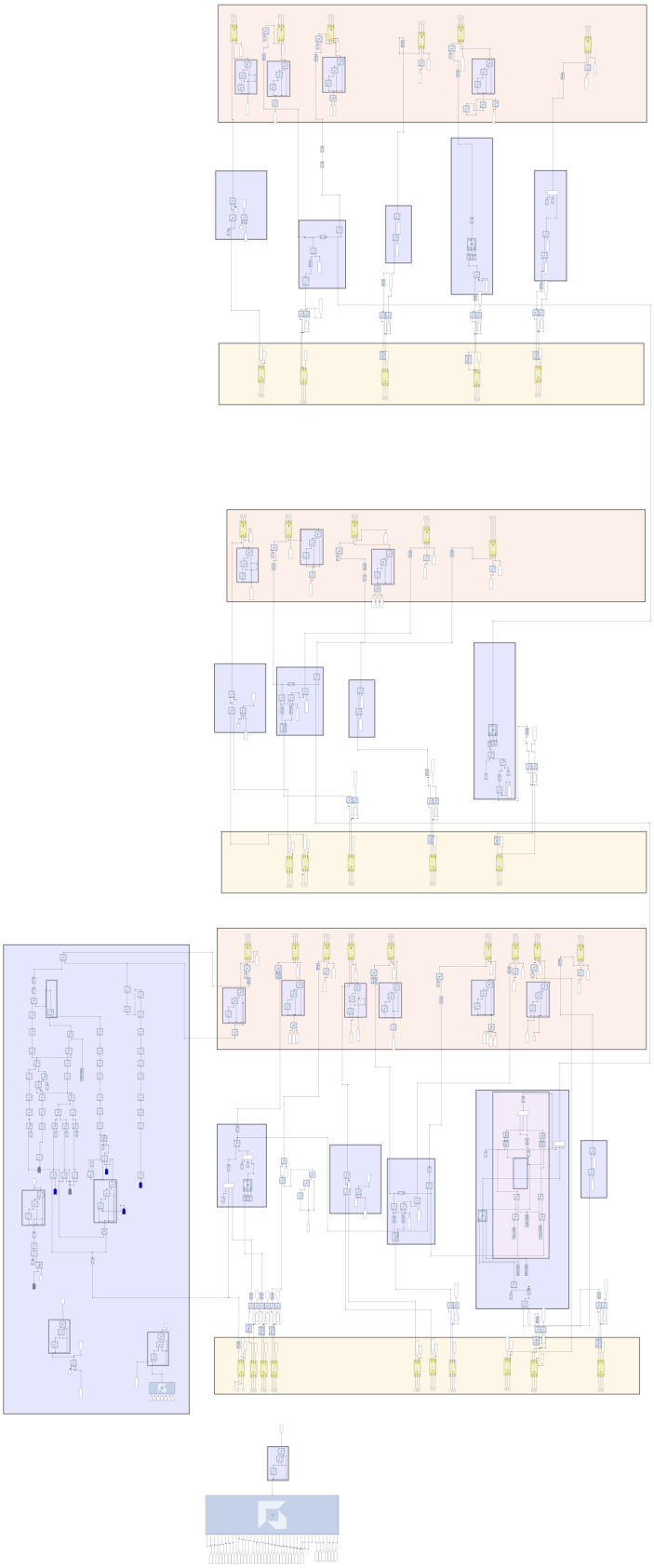


Figure 4.15. Simulink block diagram of the PL subsystem. Blue rectangles contain the logic instantiation, yellow rectangles contain the input logic, and red rectangles contain the output logic.

Data arrives from the external world and AIEs as floating point, but since it is more efficient to work with fixed point data types inside the PL, it has been chosen to cast all the input into signed fixed point of 32 bits (16 bits of decimal part). Then, data width is managed in order not to lose any information until the final re-conversion to the floating world.

TVALID-TREADY management

In order to speed up the clock frequency, a pipeline approach has been introduced. This approach complicated a lot the TVALID management in handshakes: since at each PL output port TVALID should be raised for just 1 cycle, the scheme for its generation should be the one reported in Figure 4.16: a logical AND between all the input from which the output depends, delayed by the same number of cycles as the longest combinatorial path, to be fed to an edge detector block.

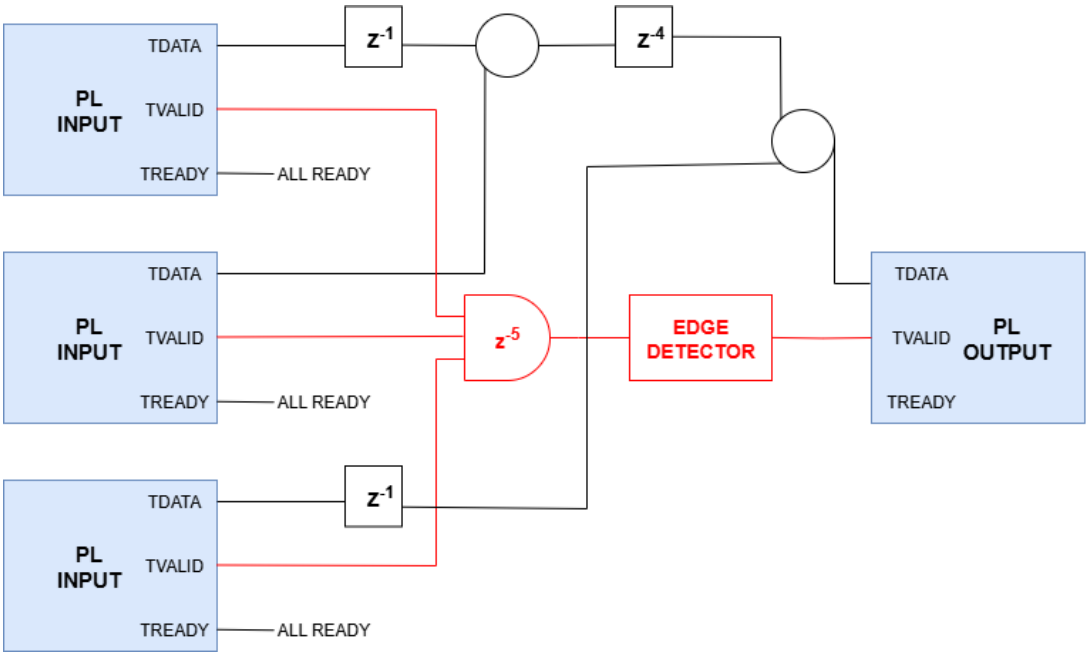


Figure 4.16. Block diagram example for TVALID management in PL: in this case, the output TVALID is generated starting from an AND between all the inputs, delayed as the longest path.

Figure 4.17 shows the TREADY generation: since it is a TREADY whose assertion allows the new set of input to enter the graph, it is generated as a logical AND of all the TREADY (all outputs must be accepted) and all the TVALID (all the graph must have finished execution).

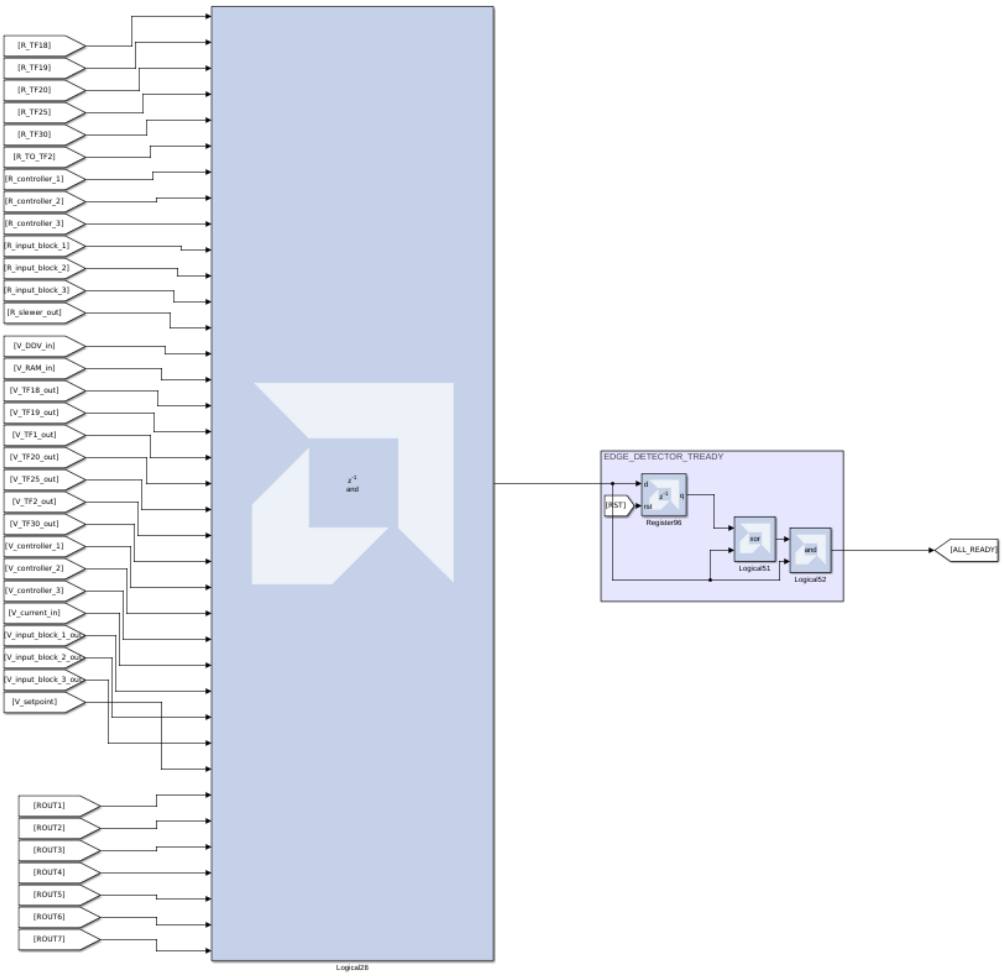


Figure 4.17. TREADY generation for all the external world input blocks of the PL subsystem.

This TREADY signal is used just for the 7 input interfaces coming from the external world, while for the 13 outputs of AIE, it is fixed to a logical '1', since if just one sample is elaborated at a time, the PL is always ready to accept a new value. Since the input TVALID should stay high at least until the data is no longer necessary, the ALL_READY signal is used to reset a register for TVALID coming from each AIE output, as shown in Figure 4.18.

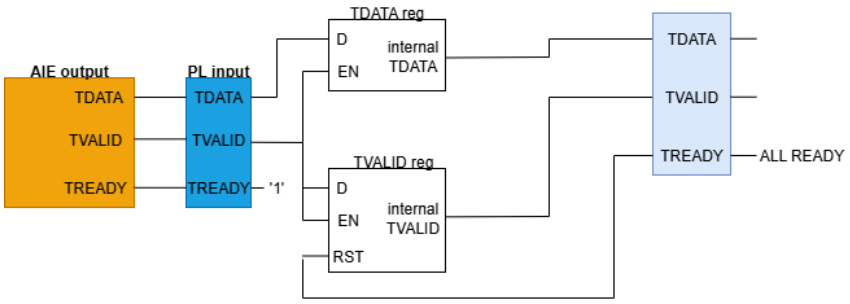


Figure 4.18. Block used to store the output of AIE inside the PL. It introduces 1 extra cycle of latency. For the 7 external inputs, these registers are not necessary.

In Figure 4.19 is reported a simple example waveform of TVALID/TREADY management.

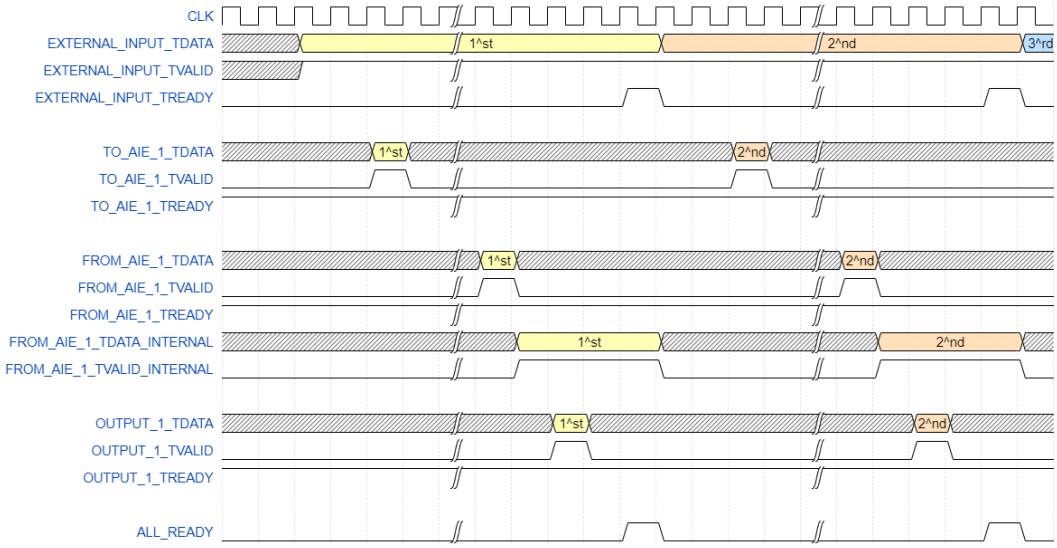


Figure 4.19. Waveform for TVALID-TREADY internal management. It is supposed that external inputs are always valid, while inputs coming from AIE need the TREADY always high, so to keep the validity high, two internal signals are required (as shown in Figure 4.18). ALL_READY signal allows the reset of these registers and a new set of input data to enter the graph.

It is supposed that in the real application, a new set of input arrives every 100 us, but for simulation purposes, the new elaboration will start just as the previous one has finished, and so the sampling time is given by a logical AND of the validity of the 7 inputs.

4.2.4 Testbench & Simulink simulation

As a first approach to debug the system, a Simulink testbench has been created, which contains both the original model and the Model Composer one. Both the models receive the same setpoint, but each one has its own actuator model to generate the feedback signals (Figure 4.20).

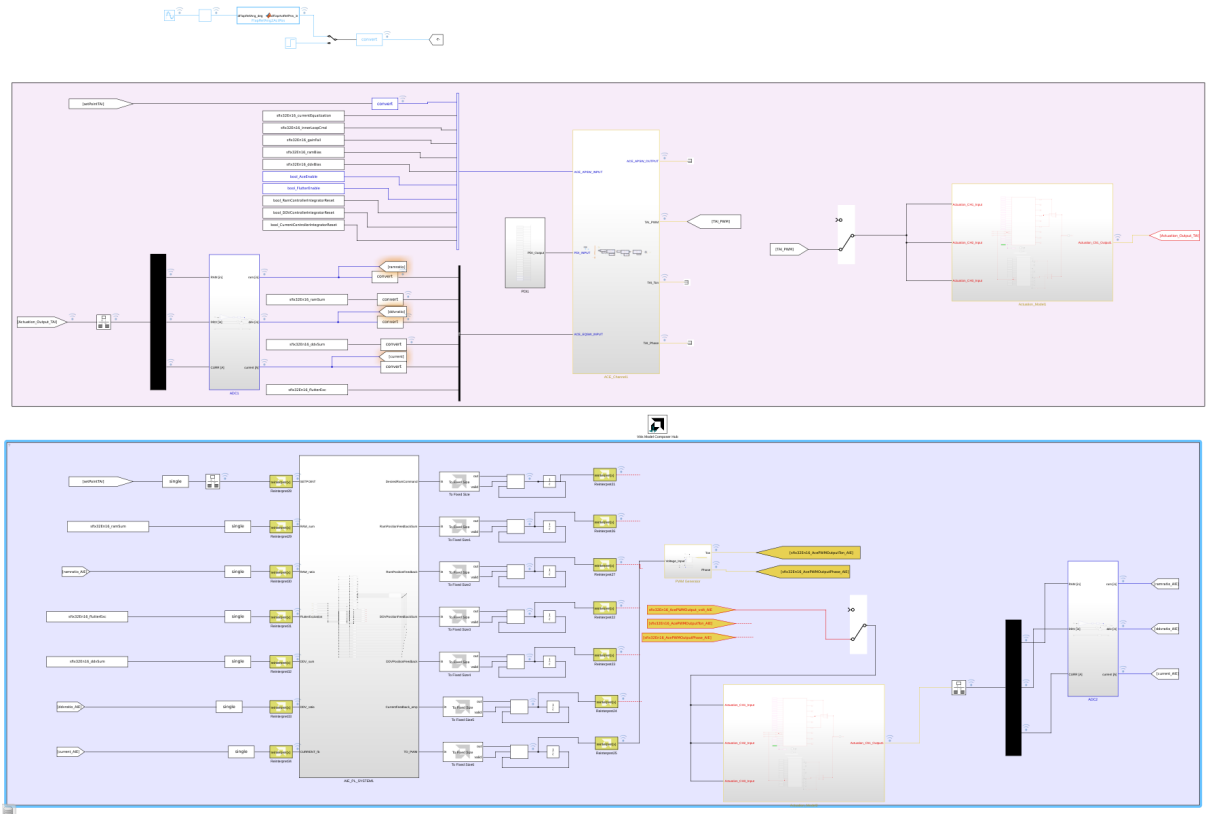


Figure 4.20. Testbench of the ACE.

The first simulation run connected the feedback generated by the original model to the input of both models, such that all 7 inputs were the same, and the result of this simulation is shown in Figure 4.21.

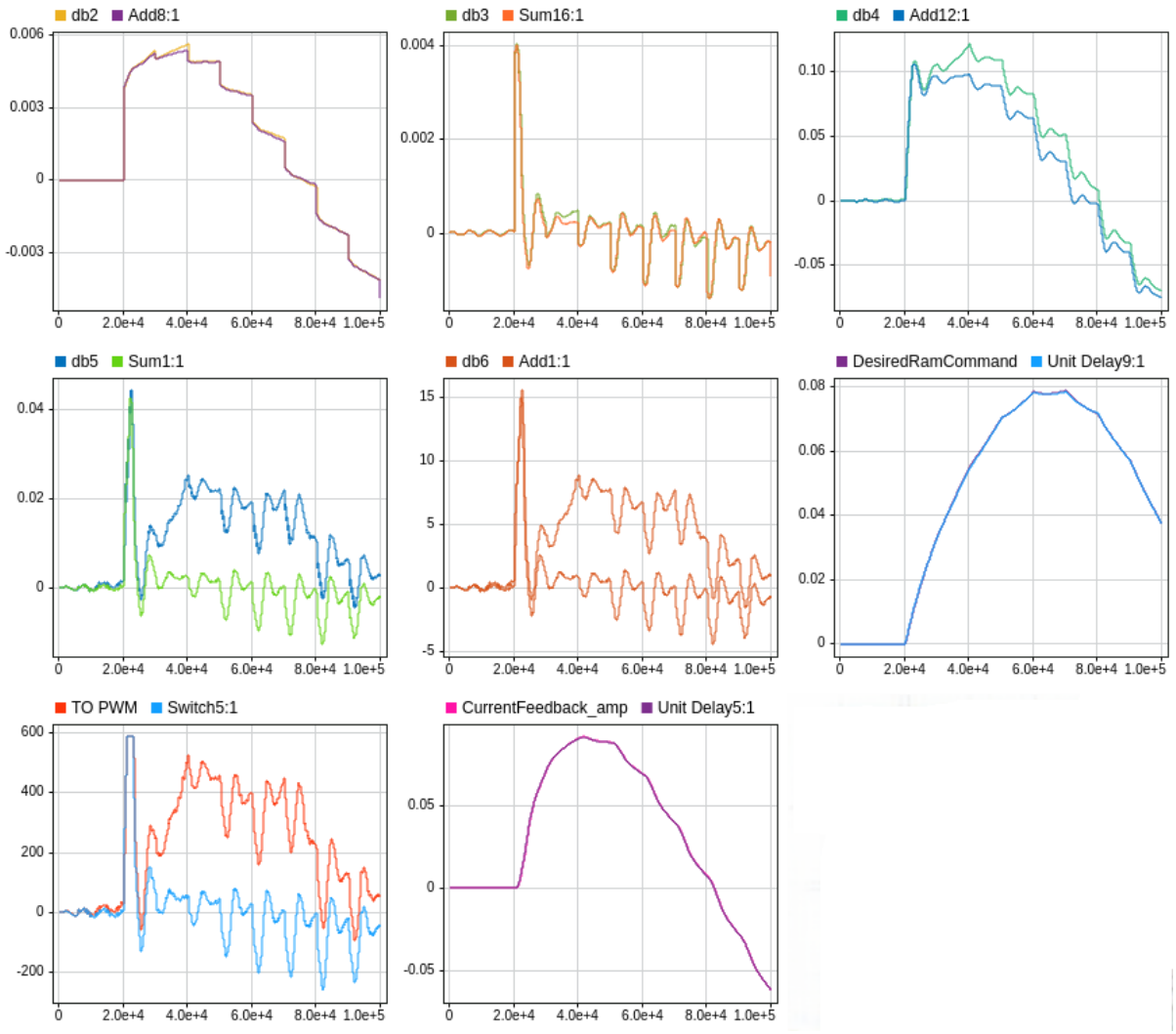


Figure 4.21. Results of the testbench where both models receive the same input. x-axis is in μs. In the legend of each graph, the first signal is the one coming from the PL subsystem, the second one comes from the reference model.

It is quite evident that the results are quite divergent, so some debug probes have been inserted in the PL design in order to understand this behavior (Figure 4.22).

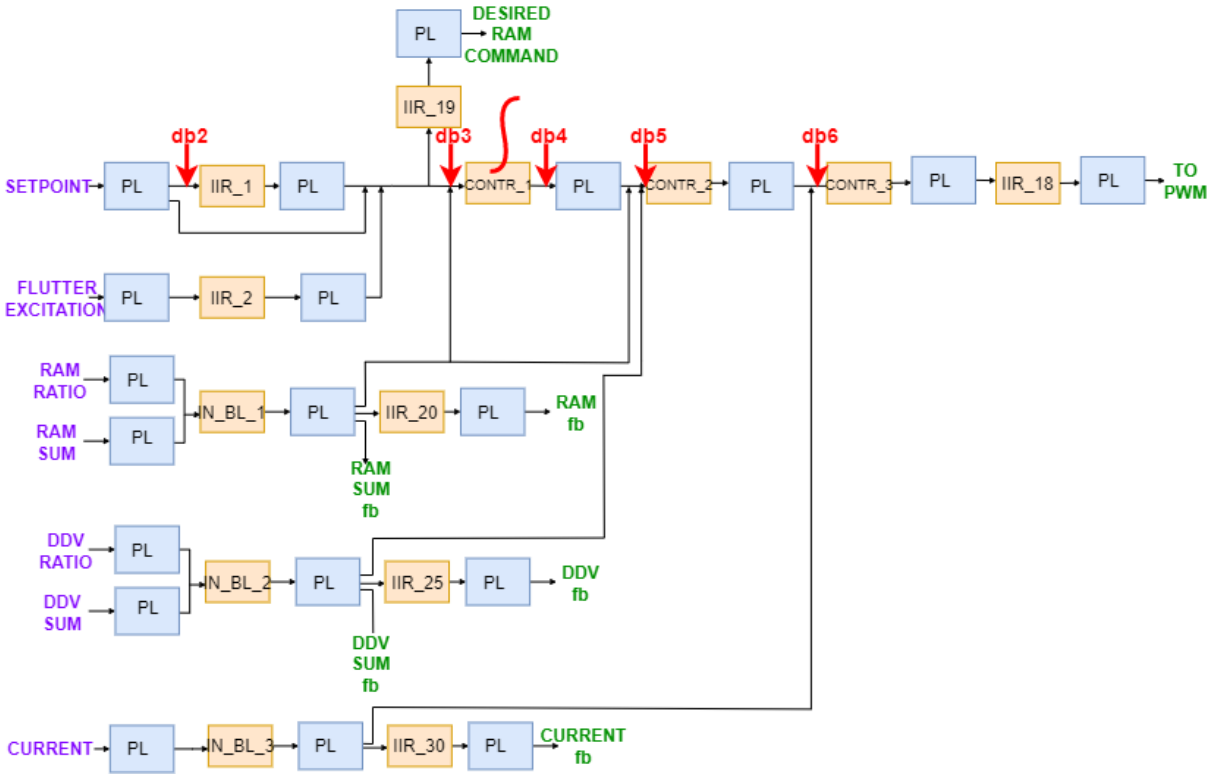


Figure 4.22. Debug probes position in the graph. The block CONTROLLER_1 is the one responsible for the integration.

A little error, introduced by the slewer logic, is integrated in db4; then, after a subtraction, the relative amplitude of the error increases in db5, and is translated to TO_PWM output, which can be considered as the ultimate output of the ACE since, to generate it, the entire graph must have been traversed.

The original cause of the divergence is the slewer, but why? The slewer is a block needed in order to mitigate step-like behavior given by the fact that the setpoint changes once every 100 executions of the graph. In Figure 4.23 are reported the block diagrams of the slewer model and implementation on the PL.

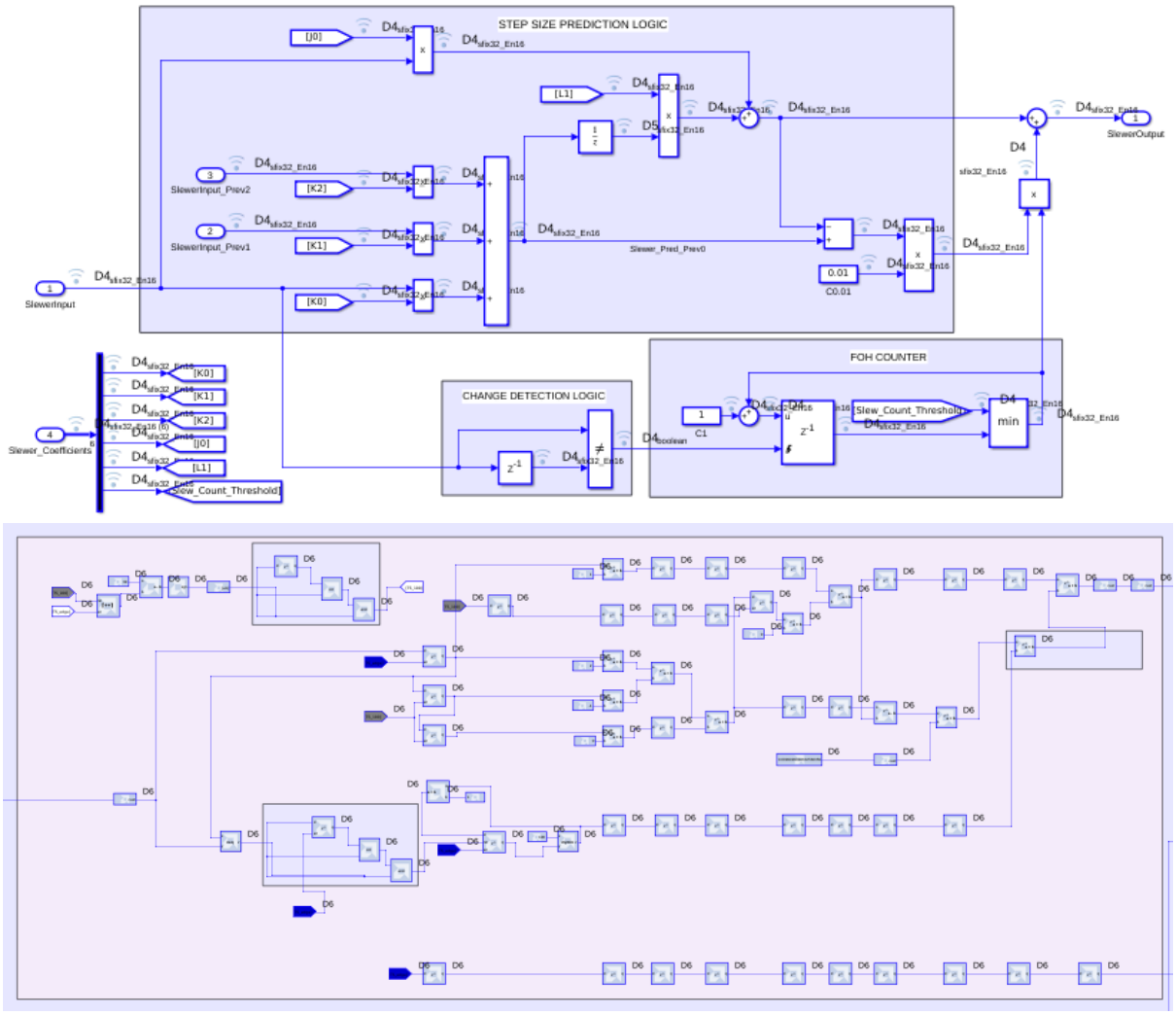


Figure 4.23. Slewer block diagram. Above: original model, below: PL one.

The difference between the two systems is the data type: in the original model, all the operations are performed on `sfixdt(1,32,16)`, and even after multiplication, the output is floored to the nearest representable value (only 16 bits of decimal part). The represented behavior is not correct, since values are typically small numbers, which means that always keeping 16 bits of decimal part may cause significant relative errors. Since the result is rounded towards zero, when the value is positive, the rounding error is always negative, while when the output is negative, the rounding error is positive. This also explains why the gap between the two curves gets narrower when the input becomes negative.

This behavior is not acceptable because such different values for the PWM will cause the system to behave unpredictably.

Things change when the feedback of the Model Composer system is connected to the one generated by its actuator. Such a simulation produces the output shown in Figure 4.24. As shown there, the bias error has disappeared because of the negative feedback

effect, and now the numerical behavior is the same.

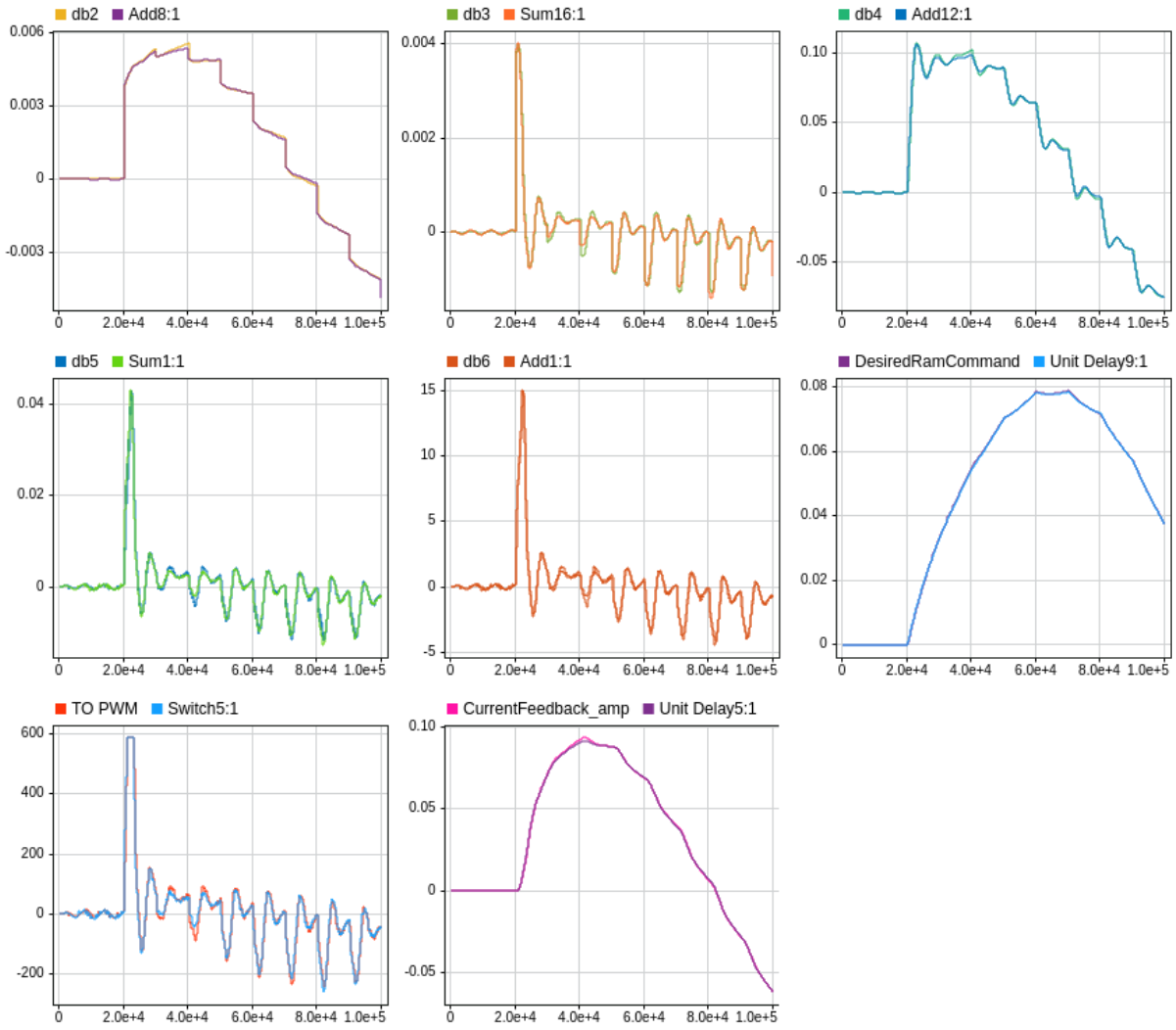


Figure 4.24. Results of the testbench where each model receives its own feedback. x-axis is in μs. In the legend of each graph, the first signal is the one coming from the PL subsystem, the second one comes from the reference model.

At this point, since the behavior of the system is correct, the only remaining thing is to generate a `BOOT.BIN` to be loaded into the device and check whether the behavior is the same also on the real hardware, and then check its performance against the normal FPGA design.

4.3 Tests on the board - simple example

Since Model Composer's simulations are only bit-accurate concerning the AI Engines (AIEs), some tests were performed on the VCK190 board to estimate actual latencies and verify result correctness.

Manually managing correct AXI-Stream handshakes is not trivial, but it allows for greater control over the synthesized circuit. As the initial hardware results did not match those from the Simulink simulations, a simpler system was chosen to ease the debugging process.

This simplified system integrates both the Programmable Logic (PL) and the AI Engine (AIE), effectively serving as a scaled-down representation of the complete project architecture. It enables early validation of core functionalities and design choices.

4.3.1 Procedure

Model Composer automatically generates software for the Processing System and creates two HLS DMAs in order to move input from DDR toward the IP under test and to compare them with the golden output.

The problem is that with this procedure, it can be established only the correctness of the output, but not the latency of the computation. Furthermore, in case the system does not work correctly, debugging is extremely difficult.

For this reason, it is useful to open the Vivado project generated by Model Composer and integrate an ILA (Integrated Logic Analyzer) in order to take a look at the real waveforms. If simulation results are different from hardware simulation, probably the error lies in the AIE-to-PL interfaces, since Simulink does not take into account the AXIS interface's latency or AIE's latency.

Here is reported just an example of serial test results generated according to Model Composer's PS software:

```
******************************** Test Results ********************************

Mismatch found for output signal gateway_out_axis in sample 3.Hardware output:
↪  3d60e000,Model Composer output:3c610000
Mismatch found for output signal gateway_out_axis in sample 5.Hardware output:
↪  3dfcd000,Model Composer output:3d60e000
Mismatch found for output signal gateway_out_axis in sample 6.Hardware output:
↪  3e608000,Model Composer output:3dfcd000
Mismatch found for output signal gateway_out_axis in sample 7.Hardware output:
↪  3e608000,Model Composer output:3dfcd000
Mismatch found for output signal gateway_out_axis in sample 8.Hardware output:
↪  3eaf3800,Model Composer output:3e608000
Mismatch found for output signal gateway_out_axis in sample 9.Hardware output:
↪  3efc1000,Model Composer output:3e608000
Mismatch found for output signal gateway_out_axis in sample 10.Hardware
↪  output: 3efc1000,Model Composer output:3eaf3800
```

```
Mismatch found for output signal gateway_out_axis in sample 11.Hardware
↪  output: 3f2b6000,Model Composer output:3eaf3800
Mismatch found for output signal gateway_out_axis in sample 12.Hardware
↪  output: 3f5f9a00,Model Composer output:3efc1000
Mismatch found for output signal gateway_out_axis in sample 13.Hardware
↪  output: 3f5f9a00,Model Composer output:3efc1000
Mismatch found for output signal gateway_out_axis in sample 14.Hardware
↪  output: 3f8d5b00,Model Composer output:3f2b6000
...
```

As shown there, it seems like some hardware result is repeated, but in order to understand what is happening, ILA is necessary.

Here are the steps to be followed to introduce the ILA block inside the design:

1. Open the Vivado project generated by model composer (the project is located in `code/platform/work/_x/link/vivado/vpl/prj/prj.xpr`).
2. Add the ILA IP from the catalog inside the block diagram.
3. Double-click the ILA and set it as necessary. To monitor PL interfaces, it should be set as "interface monitor" for axis RTL.
4. Connect the clock, the reset, and the input signals.
5. Generate a new device image (this may take a while).
6. Copy the just-created `vitis_design_wrapper.pdi` file in the `/code/hw` folder.
7. Create a copy of `boot_img.bif` and rename it (for example `boot_img_custom.bif`).
8. Inside the `boot_img_custom.bif`, change the `.bin` file name, substituting '.' with '_', also removing the path. Then, locate it and copy it inside the `/code/hw` folder.
9. Repeat the procedure for the `.pdi` file, but in this case it has to be substituted with the just created `vitis_design_wrapper.pdi`.
10. Run the Petalinux `settings.sh` script, to use it from the shell.
11. Launch

```
bootgen -arch versal -image <path to /code/hw>/boot_image_custom.bif
↪  -o <path to /code/hw>/BOOT.BIN -w on
```

Following these steps, thanks to the ILA, the system can be easily debugged to find problems.

Trigger at startup

If the whole execution of the system is very short, and the system is just prepared to run once, the ILA has to be triggered at startup.

To do so, as shown in UG908 [12], the following passages have to be added, to be performed on the TCL console in Vivado and the VCK190 programmed via JTAG with the previously produced BOOT.bin:

1. open the ILA dashboard and set the trigger condition
2. export the register map of the ILA core with

```
run_hw_ila -file ila_trig_1.tas [get_hw_ilas hw_ila_1] -force
```

3. open the implemented design and apply the trigger design settings with

   ```
   apply_hw_ila_trigger ila_trig_1.tas
   ```

4. save the modified constraints file
5. write the device image from the TCL console with

   ```
   write_device_image vitis_design_wrapper.pdi -force
   ```

6. re-perform the steps 6-11 of the previous section.

4.3.2 Debug of AXI handshake in PL

This procedure has been applied to a simple not-working example of Model Composer project, reported in Figure 4.25.

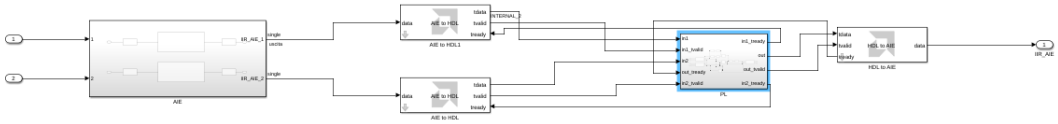


Figure 4.25. Block diagram of the simple system.

Two identical FP32 inputs are provided to two identical AIE kernels. Those kernels are second-order IIR filters, whose outputs are sent to PL. Here, they are summed together after a conversion to fixed-point numbers. The result is again converted into SPFP and then sent to an AXI4 interface. The adder is implemented with 1 cycle of latency, to break the critical path, and as clock frequency 100 MHz was chosen. The PL subsystem is reported in Figure 4.26.

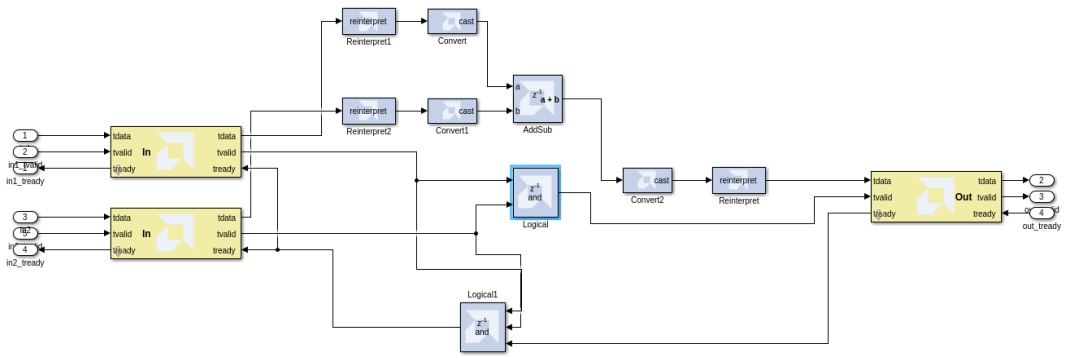


Figure 4.26. Initial block diagram of the PL region of the simple system.

ILA tests in this case helped to discover a problem in TREADY/TVALID management: every two inputs, three outputs were produced instead of two (as shown in Figure 4.27).

The solution to this problem is to sample the output TVALID as a logic AND of the input block's TVALID and TREADY. In this example, 1 cycle latency has been introduced, which is, of course, wrong, but it is also a good example for showing the testbench reaction to errors and ILA debugging of wrong waveforms.

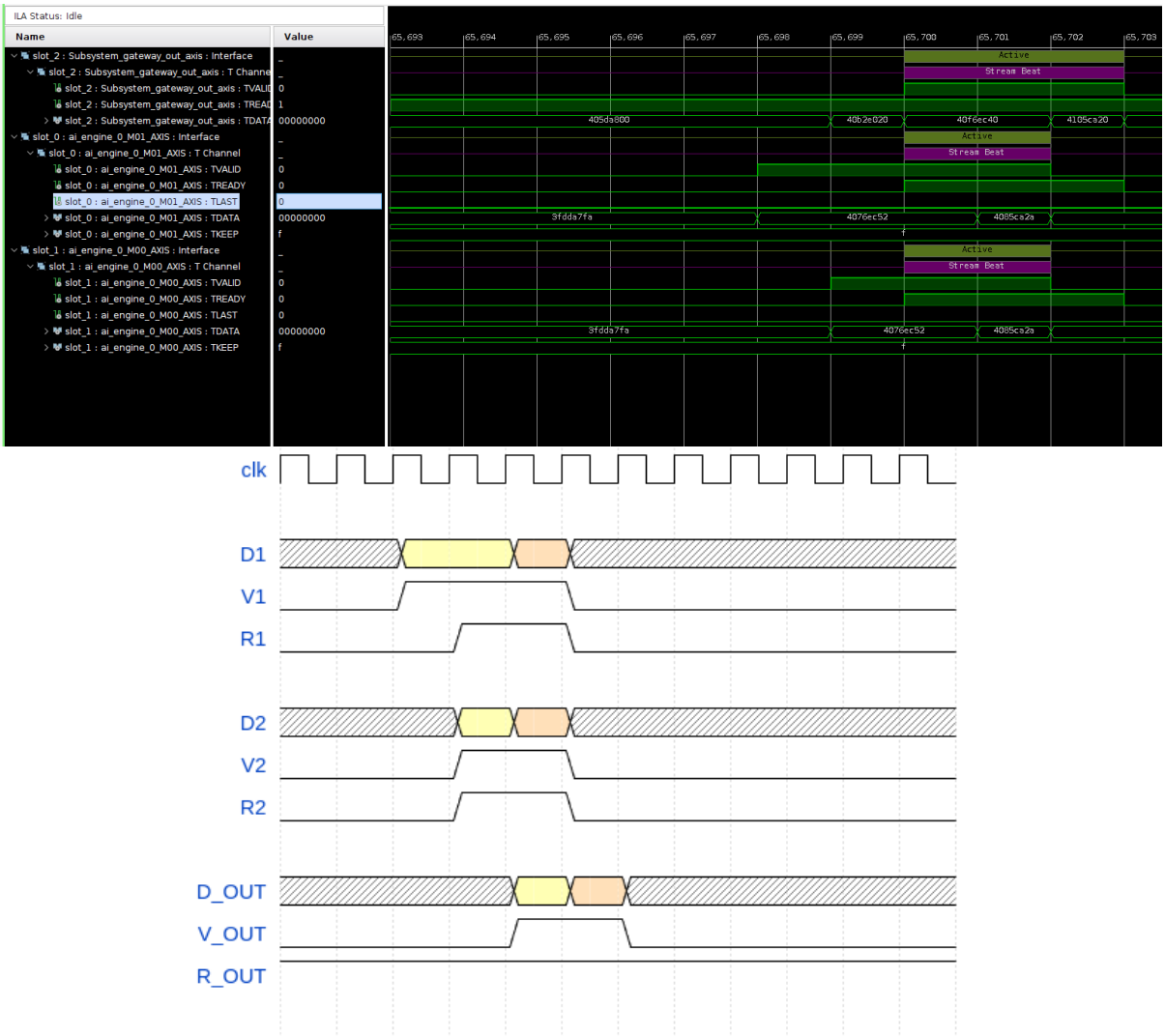


Figure 4.27. ABOVE: wrong initial ILA results. SLOTs 0 and 1 are the input, SLOT 2 is the output. BELOW: theoretical right ones. Each color represents a group of data processed in the same temporal window or computational epoch.

To eliminate timing violations problems (and to find the highest frequency at which the PL can work) from possible errors, a timing analysis has been performed.

Performing a timing analysis with a 3.2 ns PL period results in a timing violation, even introducing other registers in the cast blocks to further break the critical path (now there is a 3-cycle latency), but with 6.4 ns, all timing requirements are met, as shown in Figure 4.28.

The last modification was the introduction of an enable block for the internal PL logic. Without it, if TREADY was de-asserted, the three samples in the pipeline would have continued propagating to the output, which would not have been ready to accept them.

Violation type : setup

	Slack (ns)	Delay (ns)	Logic Delay (ns)	Routing Delay (ns)	Levels of Logic	Source	Destination	Source Clock	Destination Clock	Path Constraints
1	-2.2720	5.1450	2.1350	3.0100	23	base_kernel_finale/AIE_PL/PL/AddSub	base_kernel_finale/AIE_PL/PL/Convert2	clk	clk	create_clock -name clk -period 3.2 [get_ports clk]
2	1.3440	1.5190	0.7390	0.7800	8	base_kernel_finale/AIE_PL/PL/Convert1	base_kernel_finale/AIE_PL/PL/AddSub	clk	clk	create_clock -name clk -period 3.2 [get_ports clk]
3	2.2150	0.6170	0.3870	0.2300	0	base_kernel_finale/AIE_PL/PL/Logical	base_kernel_finale/AIE_PL/PL/Logical	clk	clk	create_clock -name clk -period 3.2 [get_ports clk]
4	2.3700	0.5200	0.1410	0.3790	1	base_kernel_finale/AIE_PL/PL/Logical1	base_kernel_finale/AIE_PL/PL/Logical	clk	clk	create_clock -name clk -period 3.2 [get_ports clk]
5	2.5110	0.3620	0.0900	0.2720	0	base_kernel_finale/AIE_PL/PL/AddSub	base_kernel_finale/AIE_PL/PL/Convert2	clk	clk	create_clock -name clk -period 3.2 [get_ports clk]

Violation type : setup

	Slack (ns)	Delay (ns)	Logic Delay (ns)	Routing Delay (ns)	Levels of Logic	Source	Destination	Source Clock	Destination Clo...	Path Constraints
1	0.9280	5.1450	2.1350	3.0100	23	base_kernel_finale/AIE_PL/PL/AddSub	base_kernel_finale/AIE_PL/PL/Convert2	clk	clk	create_clock -name clk -period 6.4 [get_ports clk]
2	4.5440	1.5190	0.7390	0.7800	8	base_kernel_finale/AIE_PL/PL/Convert1	base_kernel_finale/AIE_PL/PL/AddSub	clk	clk	create_clock -name clk -period 6.4 [get_ports clk]
3	5.4150	0.6170	0.3870	0.2300	0	base_kernel_finale/AIE_PL/PL/Logical	base_kernel_finale/AIE_PL/PL/Logical	clk	clk	create_clock -name clk -period 6.4 [get_ports clk]
4	5.5700	0.5200	0.1410	0.3790	1	base_kernel_finale/AIE_PL/PL/Logical1	base_kernel_finale/AIE_PL/PL/Logical	clk	clk	create_clock -name clk -period 6.4 [get_ports clk]
5	5.7110	0.3620	0.0900	0.2720	0	base_kernel_finale/AIE_PL/PL/AddSub	base_kernel_finale/AIE_PL/PL/Convert2	clk	clk	create_clock -name clk -period 6.4 [get_ports clk]

Figure 4.28. Timing analysis results of the simple subsystem. Above the requirements are not met ($T_{ck} = 3.2$ ns), bottom the requirements are met ($T_{ck} = 6.4$ ns).

With these couple of modifications, with the PL subsystem reported in Figure 4.29, the serial test result of the VCK190 reports:

```
******************************** Test Results ********************************

****** Model Composer and Hardware output match for all 47 samples for output
↪  signal gateway_out_axis

****** TEST PASSED

****** VMC_TEST_DONE
```

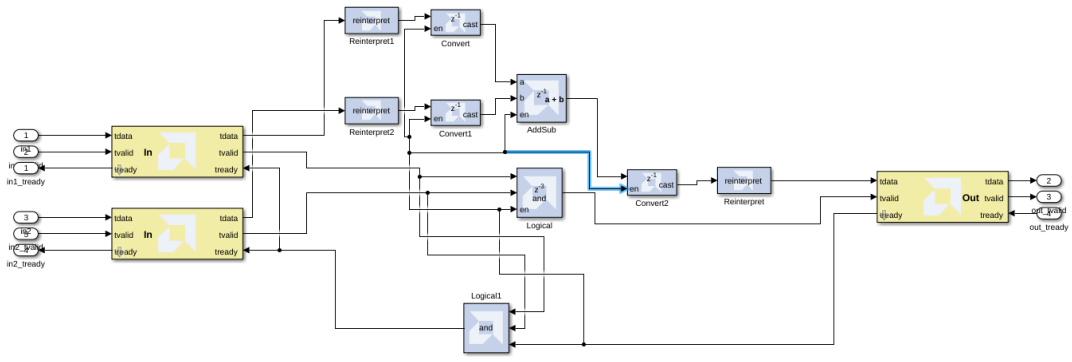


Figure 4.29. Correct block diagram of the PL region of the simple system.

As concerns the ILA in this working experiment, in Figure 4.30 is reported the correct AXIS handshake waveform.

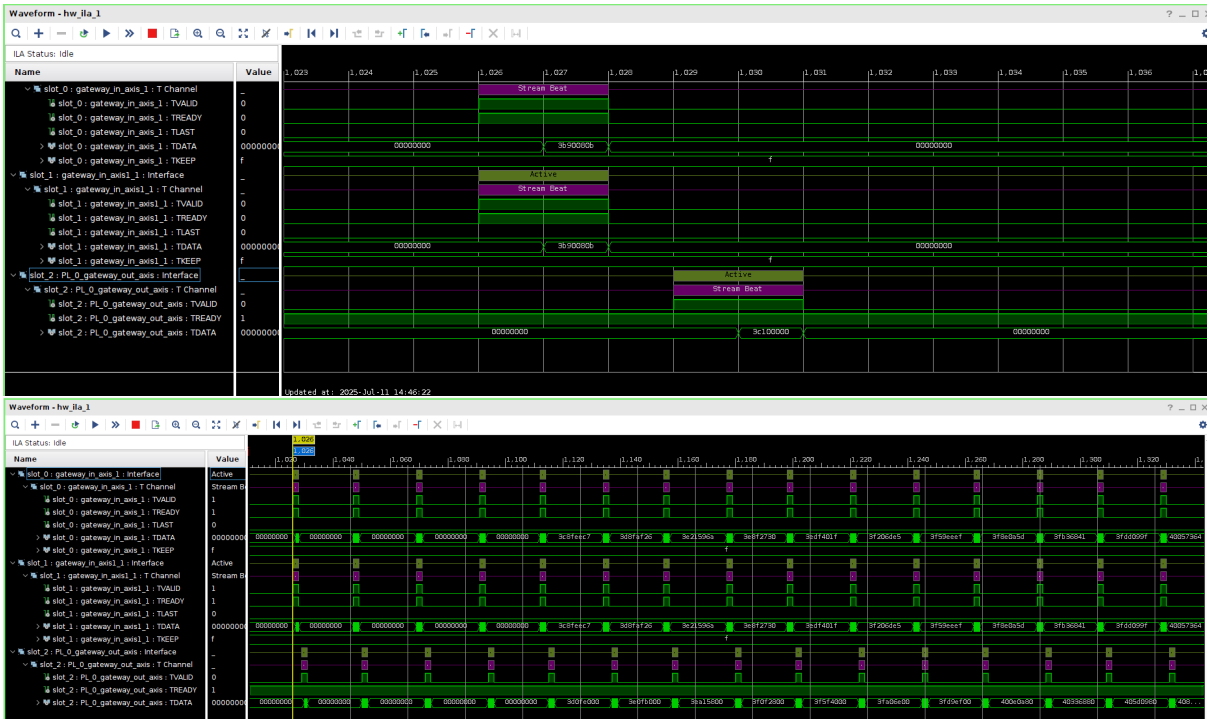


Figure 4.30. Correct behavior registered with ILA. Above: zoom on a single transition, below: subsequent transmissions.

A couple of observations deserve to be pointed out about these waveforms:

- the latency now is 3 cycles, since two more stages have been introduced;
- stream beats are composed of 2 samples. This is because the AIE-PL streaming interface is 64-bit wide, so 2 is the minimum number of 32-bit words that can be packed together;
- after the first four beats, in idle condition, the bus is not empty anymore, but hosts an old value: it is as if the compiler used 8 different paths to stream the data cyclically, and so when it is time for the ninth data, before the actual transmissions, the bus still contains the first data;
- trying to measure throughput, the 2-sample beat is received every 21 or 22 clock cycles. This may be due to the different frequencies: the AIE interface is working at 312.5 MHz, while PL is at 156.25 MHz. The sampling may happen sometimes before and sometimes after the arrival time from the different clock domains.

Throughput and latency

In order to estimate the throughput of the system, it is known that $f_{ck} = 156{,}25\,MHz$, so $T_{ck} = 6.4\,ns$. Two outputs are ready in around 21.5 cycles, this means that the Iteration Interval $II = \frac{6.4ns \cdot 21.5}{2} = 68.8\,ns$. This should also be the latency of AIE computational kernels, since for a new execution to start, the previous must end.

Looking at the assembly code of an IIR filter order 2, the number of assembly instructions is 80, the clock frequency is $f_{ck} = 1.250\,GHz$, and so the esteemed execution time inside the AIE is 64 ns, and since that should be the main contribution of the II, things

make sense.

Now, it is interesting to take a look at the overall latency from input to output introduced by the whole system. The components of the latency are:

- Memory to AIE data transfer (NoC).
- Intra-AIE NoC's latency.
- AIE kernel computational latency.
- Intra-AIE NoC's latency.
- AIE to PL data transfer latency.
- PL latency.

As concerns intra-AIE NoC latency, its contribution can be found thanks to Vitis Analyzer, and is 8 x 0.8 ns = 6.4 ns, while PL latency is 3 cycles (19.2 ns). The NoC and the AIE to PL data latency cannot be measured, but their value is probably negligible with respect to the rest.

4.3.3 Direct measure

A direct measure of latency given by the AIE array (2 x Intra-AIE streaming + kernel execution) can be obtained by inserting two ILAs: one at the input of AIE and one at the output. The first one can be used to trigger the second one.

The problem is that the trigger should be the same for both the ILAs to measure how many cycles of intercourse occur between the input and output of the AIE array.
To do so, the ILAs should be configured as shown in Figure 4.31.

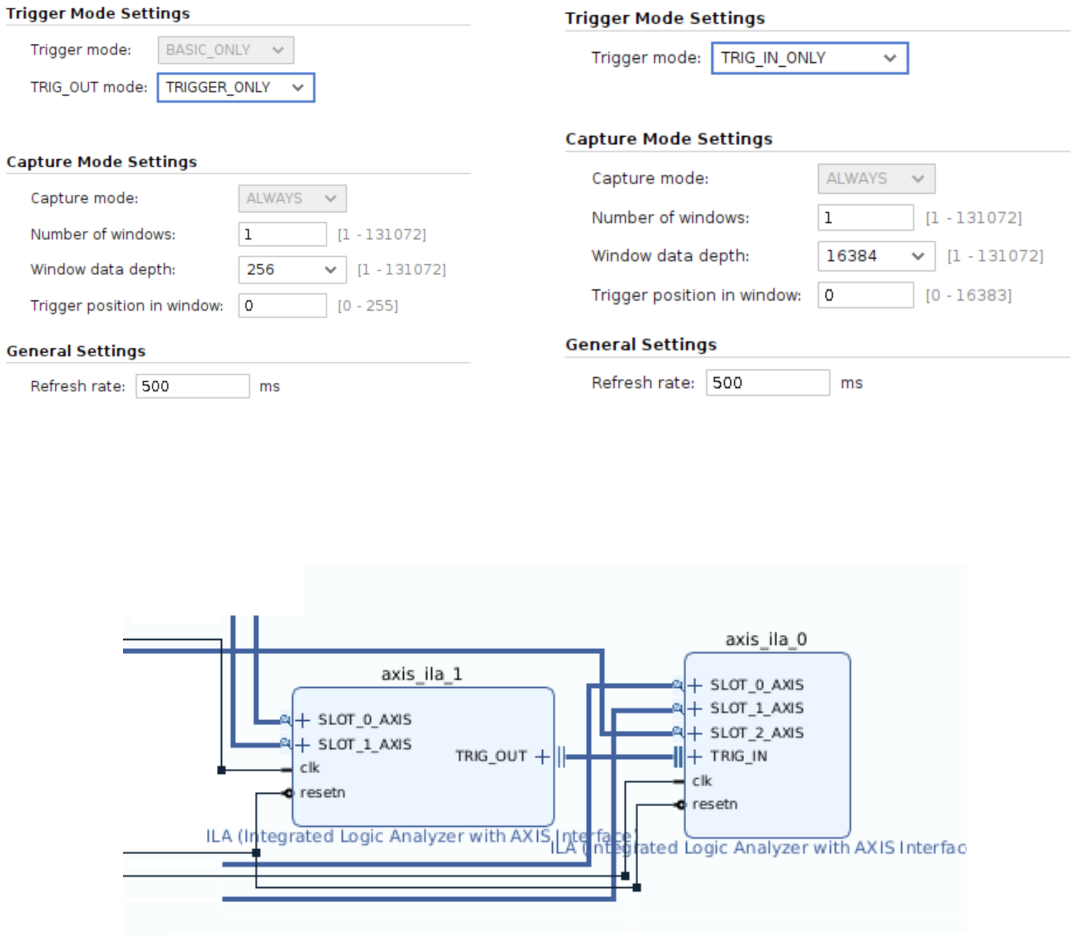


Figure 4.31. Settings of ILAs in order to measure with the same trigger. On the left, the first one, on the right, the second one, below the block diagram (axis_ila_1 is the first, while axis_ila_0 is the second).

The trigger condition can be set as $TDATA! = 0$. Then, the same procedure applied for the trigger at startup has to be repeated, doubling each command to configure the registers of both ILAs, and the constraints file is automatically updated, configuring the ILAs to arm the trigger at startup.

Measuring the input of the array with the ILA, the waveform in Figure 4.32 is measured.

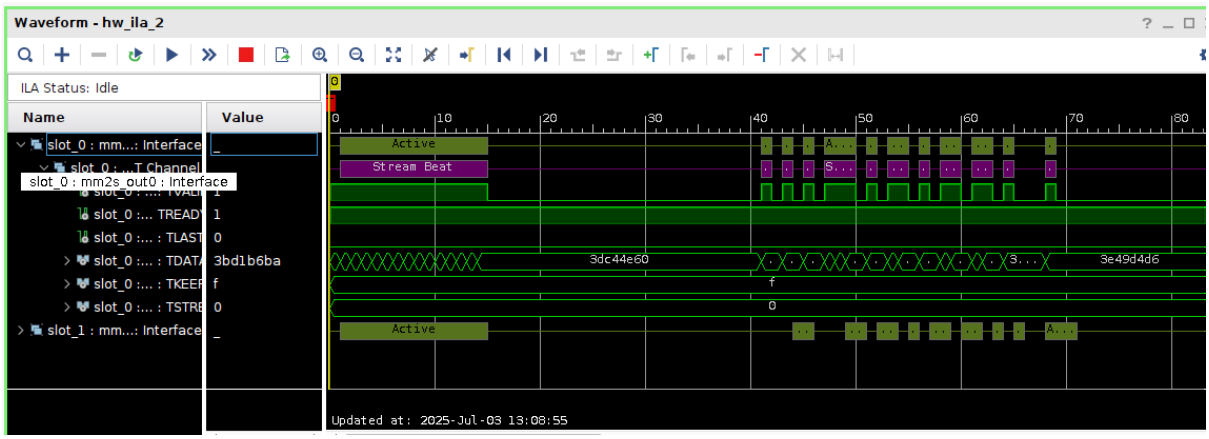


Figure 4.32. Waveforms of the AIE array's input.

The data transfer registered here is between the DDR and the AIE array, performed with the Model Composer's automatically generated HLS DMA *mm2s*. Probably the compiler of the NoC retained the most efficient start with a 16-sample beat (that probably is filling an input buffer of the AIE array), to wait a little and start again with smaller beats later.

The second ILA was set up to trigger at the same time as the first one, but looking at the waveforms, it is clear that the AIE elaboration's start is not registered (no rising edge of TVALID detected at the output of the AIE, at least in the 0.85 ms after the trigger condition).

Looking at the code `host.cpp` running on the PS (generated by Model Composer), the reason for that is clear: Model Composer automatically programs the DMAs before initializing and starting the graph, as shown here:

```
...
   Xil_Out32(mm2s_base + M_IN0_OFFSET, (uint32_t) mem_in0Addr);
   Xil_Out32(mm2s_base + M_IN0_OFFSET + 4, 0);
   Xil_Out32(mm2s_base + M_IN0_SIZE_OFFSET, input0_size);

   Xil_Out32(mm2s_base + M_IN1_OFFSET, (uint32_t) mem_in1Addr);
   Xil_Out32(mm2s_base + M_IN1_OFFSET + 4, 0);
   Xil_Out32(mm2s_base + M_IN1_SIZE_OFFSET, input1_size);

   Xil_Out32(s2mm_base + S_OUT0_OFFSET, (uint32_t) mem_out0Addr);
   Xil_Out32(s2mm_base + S_OUT0_OFFSET + 4, 0);
   Xil_Out32(s2mm_base + S_OUT0_SIZE_OFFSET, output0_size);

   Xil_Out32(mm2s_base + M_PULSE_OFFSET, 2);
   Xil_Out32(mm2s_base + M_CTRL_OFFSET, 1);
   Xil_Out32(s2mm_base + S_CTRL_OFFSET, 1);

   printf("AI Engine graph init\n");
   mygraph.init();
```

```
    printf("AI Engine graph run\n");
    mygraph.run();
...
```

and the delay given by graph initialization is greater than the maximum ILA window at 156.25 MHz.

Before measuring the result, another modification was made: instead of inserting two ILAs with a cascade trigger, just one ILA was inserted with a clock conversion and AXI broadcasters, to visualize all the waves on the same window.

In Figure 4.36, the block diagram set up for this measurement is reported, while in Figure 4.33, Figure 4.34, and Figure 4.35 they are displayed the waveforms of the final modified system.

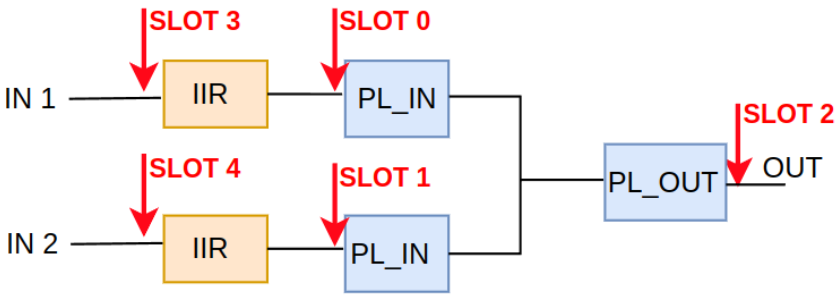


Figure 4.33. ILA slot positions in the system. SLOT 0-1: PL input; SLOT 2: PL output; SLOT 3-4: AIE input.

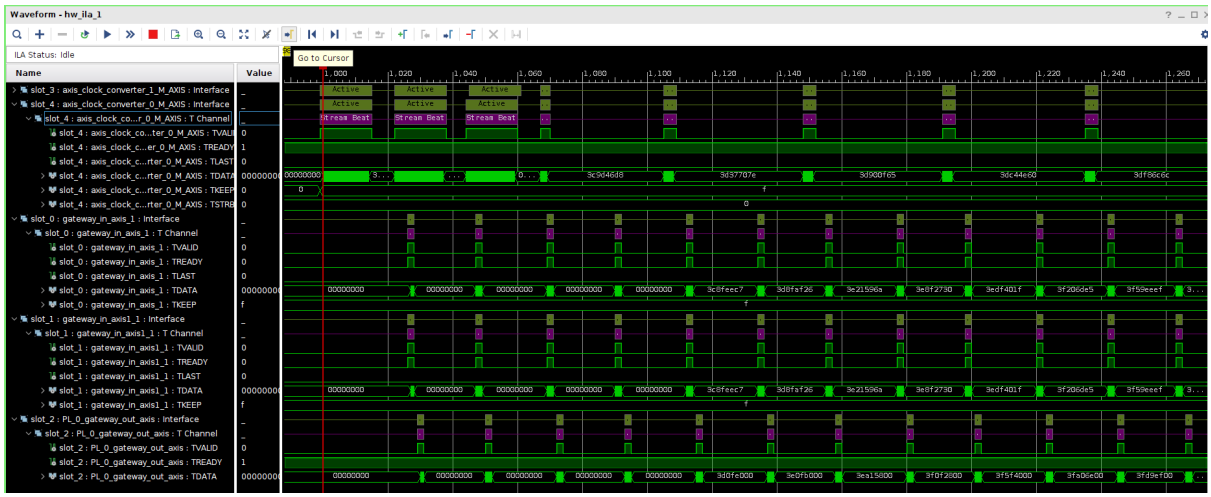


Figure 4.34. Expanded waveforms of the system. Trigger condition: TDATA (slot 3) != 0.

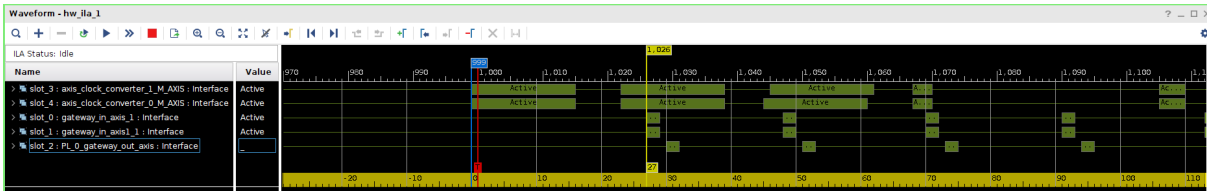


Figure 4.35. Measurement of the latency on the waveforms of the system. Trigger condition: TDATA (slot 3) != 0.

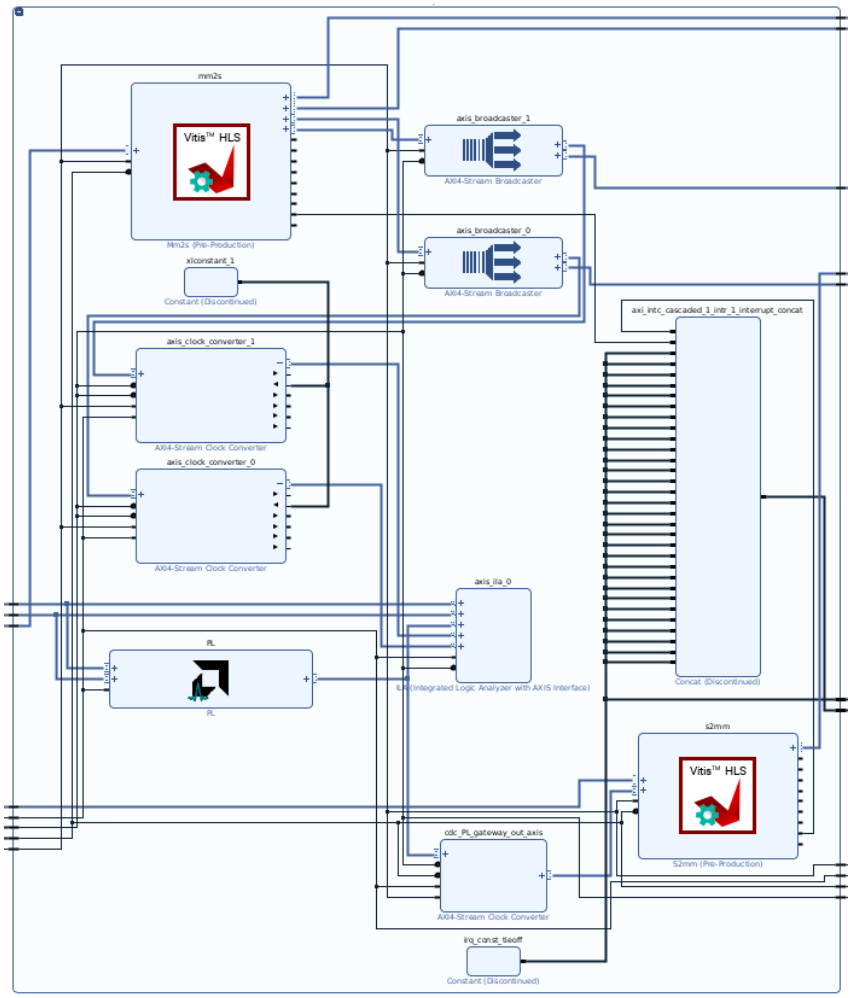


Figure 4.36. Block diagram to conFigure 1 ILA with AXI signal from different clock domains (zoom on Vitis region).

The latency between the input of the PL and the input of AIE is 27 cycles at 156.25 MHz, which corresponds to 172.8 ns, and in this period, the first two outputs are provided. These 172.8 ns are probably given by 59.2 ns for the first kernels' execution, 59.2 ns for the second, and the remaining 54.4 ns for the data transport.

Given this measure, the latency of AIE-PL transport can be estimated to be in the (35 ÷ 50) ns range, since two data are transferred, but the 54.4 ns include both AIE-PL data

transfer of the two samples and intra-AIE data transfer, and also considering that the sampling frequency is not the same frequency of the AIE array interface.

It is important to remember that ILA data are registered in PL, so also the delay due to their movement to PL has to be considered; however, these values are precious estimations for what concerns the graph mapping on the SoC.

4.3.4 Other considerations

On Vivado it is possible to take a look at many interesting report, such as the chip planner (Figure 4.37), that shows which portion of the chip are used, a NoC utilization scheme (Figure 4.38) and power reports (Figure 4.39). These power consumptions will be compared with the ACE implementation's one in subsection 4.4.2.

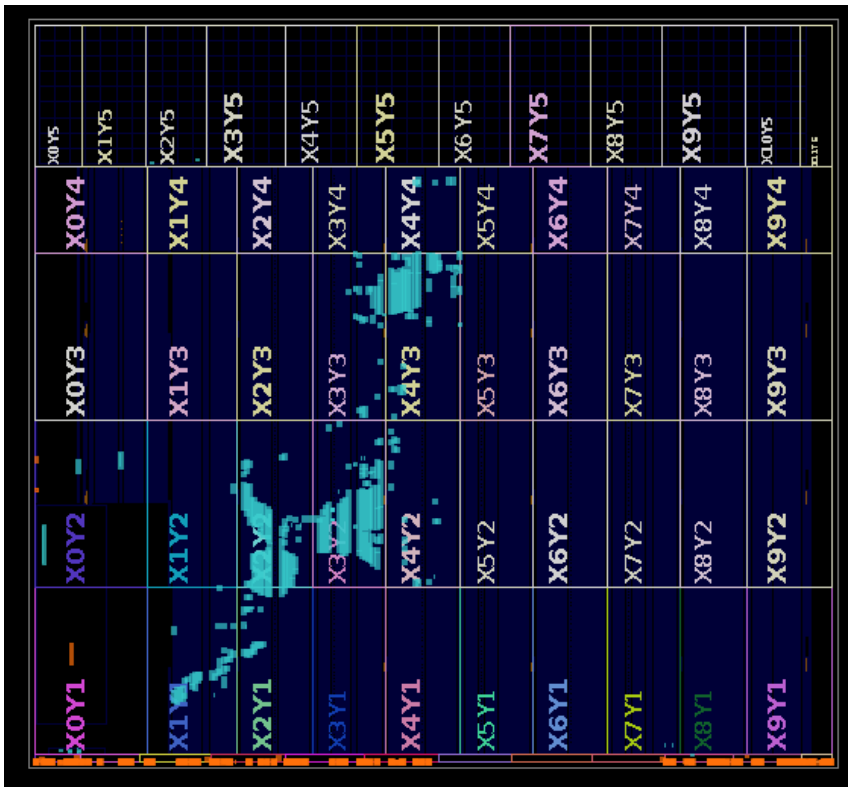


Figure 4.37. Chip plan for the simple example. The AIE region corresponds to the Y5 row of the chip, but on Vivado, even if the tiles are used, just the interface tiles are highlighted.

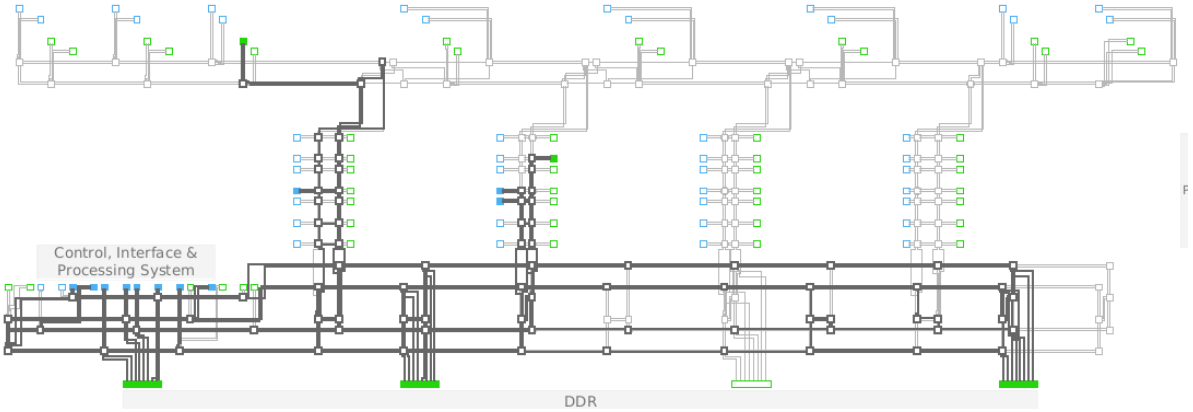


Figure 4.38. NoC utilization for the simple example.

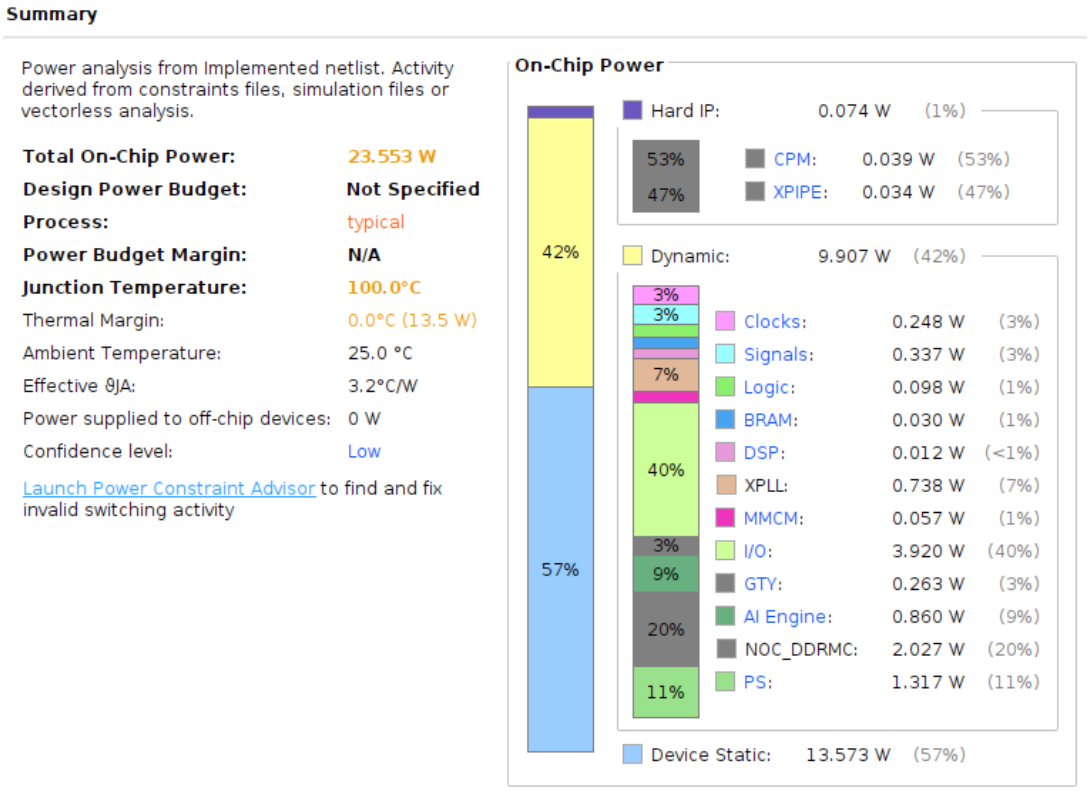
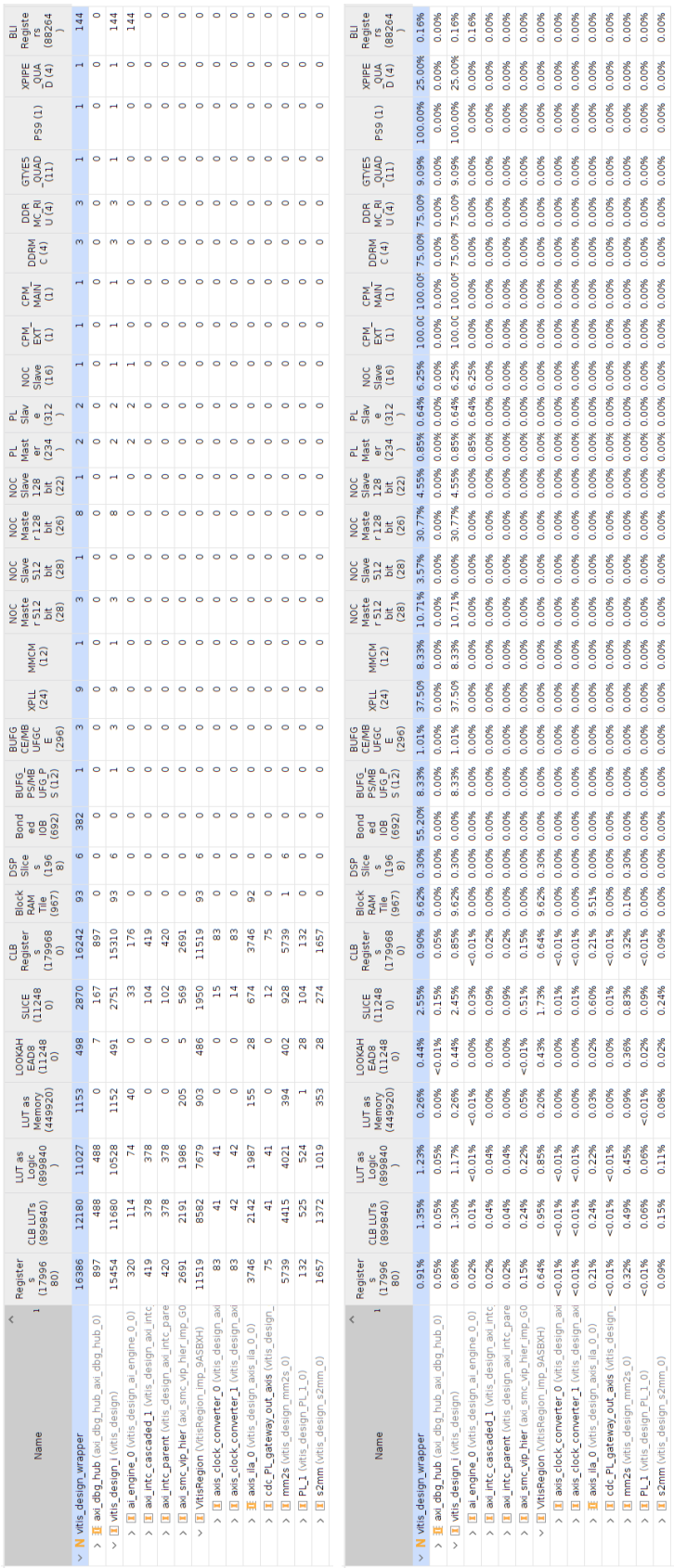


Figure 4.39. Power consumption for the simple example.

Figure 4.40 shows the resource utilization of this simple project.



Name	Register s (179968 0)	CLB LUTs (899840)	LUT as Logic (899840)	LUT as Memory (449920)	LOOKAHEAD8 (112480)	SLICE (112480)	CLB Registers (179968 0)	Block RAM Tile (967)	DSP Slices (1968)	Bonded IOB (692)	BUFG_PS/MBUFG_PS (12)	BUFG_CE/MBUFG_CE (296)	XPLL (24)	MMCM (12)	NOC Master 512 bit (28)	NOC Slave 512 bit (28)	NOC Master 128 bit (26)	NOC Slave 128 bit (22)	PL Master (234)	PL Slave (312)	NOC Slave (16)	CPM_EXT (1)	CPM_MAIN (1)	DDRMC (4)	DDR MC_RIU (4)	GTYE5_QUAD (11)	PS9 (1)	XPIPE_QUAD (4)	BLI Registers (88264)
N vitis_design_wrapper	16386	12180	11027	1153	498	2870	16242	93	6	382	1	3	9	1	3	1	8	1	2	2	1	1	1	3	3	1	1	1	144
> axi_dbg_hub (axi_dbg_hub_axi_dbg_hub_0)	897	488	488	0	7	167	897	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
> vitis_design_i (vitis_design)	15454	11680	10528	1152	491	2751	15310	93	6	0	1	3	9	1	3	0	0	0	1	2	1	0	1	3	3	1	1	1	144
> ai_engine_0 (vitis_design_ai_engine_0_0)	320	114	74	40	0	33	176	0	0	0	0	0	0	0	0	0	0	0	2	2	1	0	0	0	0	0	0	0	144
> axi_intc_cascaded_1 (vitis_design_axi_intc_	419	378	378	0	0	104	419	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
> axi_intc_parent (vitis_design_axi_intc_pare	420	378	378	0	0	102	420	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
> axi_smc_vip_hier (axi_smc_vip_hier_imp_G0	2691	2191	1986	205	5	569	2691	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
v VitisRegion (VitisRegion_imp_9ASBXH)	11519	8582	7679	903	486	1950	11519	93	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
> axis_clock_converter_0 (vitis_design_axi	83	41	41	0	0	15	83	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
> axis_clock_converter_1 (vitis_design_axi	83	42	42	0	0	14	83	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
> axis_ila_0 (vitis_design_axis_ila_0_0)	3746	2142	1987	155	28	674	3746	92	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
> cdc_PL_gateway_out_axis (vitis_design_	75	41	41	0	0	12	75	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
> mm2s (vitis_design_mm2s_0)	5739	4415	4021	394	402	928	5739	1	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
> PL_1 (vitis_design_PL_1_0)	132	525	524	1	28	104	132	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
> s2mm (vitis_design_s2mm_0)	1657	1372	1019	353	28	274	1657	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Name	Register s (179968 0)	CLB LUTs (899840)	LUT as Logic (899840)	LUT as Memory (449920)	LOOKAHEAD8 (112480)	SLICE (112480)	CLB Registers (179968 0)	Block RAM Tile (967)	DSP Slices (1968)	Bonded IOB (692)	BUFG_PS/MBUFG_PS (12)	BUFG_CE/MBUFG_CE (296)	XPLL (24)	MMCM (12)	NOC Master 512 bit (28)	NOC Slave 512 bit (28)	NOC Master 128 bit (26)	NOC Slave 128 bit (22)	PL Master (234)	PL Slave (312)	NOC Slave (16)	CPM_EXT (1)	CPM_MAIN (1)	DDRMC (4)	DDR MC_RIU (4)	GTYE5_QUAD (11)	PS9 (1)	XPIPE_QUAD (4)	BLI Registers (88264)
N vitis_design_wrapper	0.91%	1.35%	1.23%	0.26%	0.44%	2.55%	0.90%	9.62%	0.30%	55.20%	8.33%	1.01%	37.50%	8.33%	10.71%	3.57%	30.77%	4.55%	0.85%	0.64%	6.25%	100.00%	100.00%	75.00%	75.00%	9.09%	100.00%	25.00%	0.16%
> axi_dbg_hub (axi_dbg_hub_axi_dbg_hub_0)	0.05%	0.05%	0.05%	0.00%	<0.01%	0.15%	0.05%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
> vitis_design_i (vitis_design)	0.86%	1.30%	1.17%	0.26%	0.44%	2.45%	0.85%	9.62%	0.30%	0.00%	8.33%	1.01%	37.50%	8.33%	10.71%	0.00%	30.77%	4.55%	0.85%	0.64%	6.25%	100.00%	100.00%	75.00%	75.00%	9.09%	100.00%	25.00%	0.16%
> ai_engine_0 (vitis_design_ai_engine_0_0)	0.02%	0.01%	<0.01%	<0.01%	0.00%	0.03%	<0.01%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.85%	0.64%	6.25%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.16%
> axi_intc_cascaded_1 (vitis_design_axi_intc_	0.02%	0.04%	0.04%	0.00%	0.00%	0.09%	0.02%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
> axi_intc_parent (vitis_design_axi_intc_pare	0.02%	0.04%	0.04%	0.00%	0.00%	0.09%	0.02%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
> axi_smc_vip_hier (axi_smc_vip_hier_imp_G0	0.15%	0.24%	0.22%	0.05%	<0.01%	0.51%	0.15%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
v VitisRegion (VitisRegion_imp_9ASBXH)	0.64%	0.95%	0.85%	0.20%	0.43%	1.73%	0.64%	9.62%	0.30%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
> axis_clock_converter_0 (vitis_design_axi	<0.01%	<0.01%	<0.01%	0.00%	0.00%	0.01%	<0.01%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
> axis_clock_converter_1 (vitis_design_axi	<0.01%	<0.01%	<0.01%	0.00%	0.00%	0.01%	<0.01%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
> axis_ila_0 (vitis_design_axis_ila_0_0)	0.21%	0.24%	0.22%	0.03%	0.02%	0.60%	0.21%	9.51%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
> cdc_PL_gateway_out_axis (vitis_design_	<0.01%	<0.01%	<0.01%	0.00%	0.00%	0.01%	<0.01%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
> mm2s (vitis_design_mm2s_0)	0.32%	0.49%	0.45%	0.09%	0.36%	0.83%	0.32%	0.10%	0.30%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
> PL_1 (vitis_design_PL_1_0)	<0.01%	0.06%	0.06%	<0.01%	0.02%	0.09%	<0.01%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
> s2mm (vitis_design_s2mm_0)	0.09%	0.15%	0.11%	0.08%	0.02%	0.24%	0.09%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%

Figure 4.40. Resources utilization of the final implementation (absolute values on the left and % on the right).

4.3.5 Estimation of complete system behavior

Based on measurements of this simple example, a possible latency estimation of the whole ACE can be made.

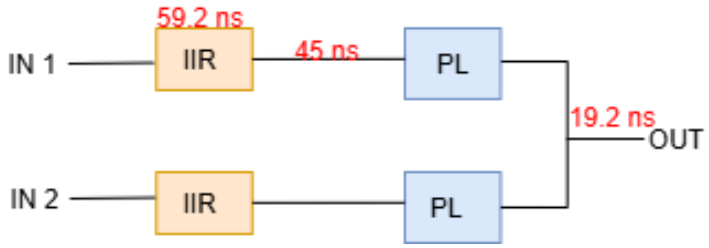


Figure 4.41. Latency measured on the simple example.

In Figure 4.41 are reported the latency measurements performed on the simple system. The kernel latency can be evaluated as $\#assembly\ instructions \cdot 0.8ns$, and their values can be found in table 4.1.

As concerns the PL latency, it is known to be exactly $T_{ck} \cdot \#pipeline\ stages$.

For intra-AIE + AIE-PL data transfer, it is reasonable to assume 45 ns as an estimation, since probably around 10 ns are spent for intra-AIE transport and $(35 \div 50)$ for data streaming from AIE to PL or vice-versa (for simplicity, also the latency from the AIE array interface tile to the kernel tile is included in the 45 ns).

The graphical calculation of the overall latency is reported in Figure 4.42.

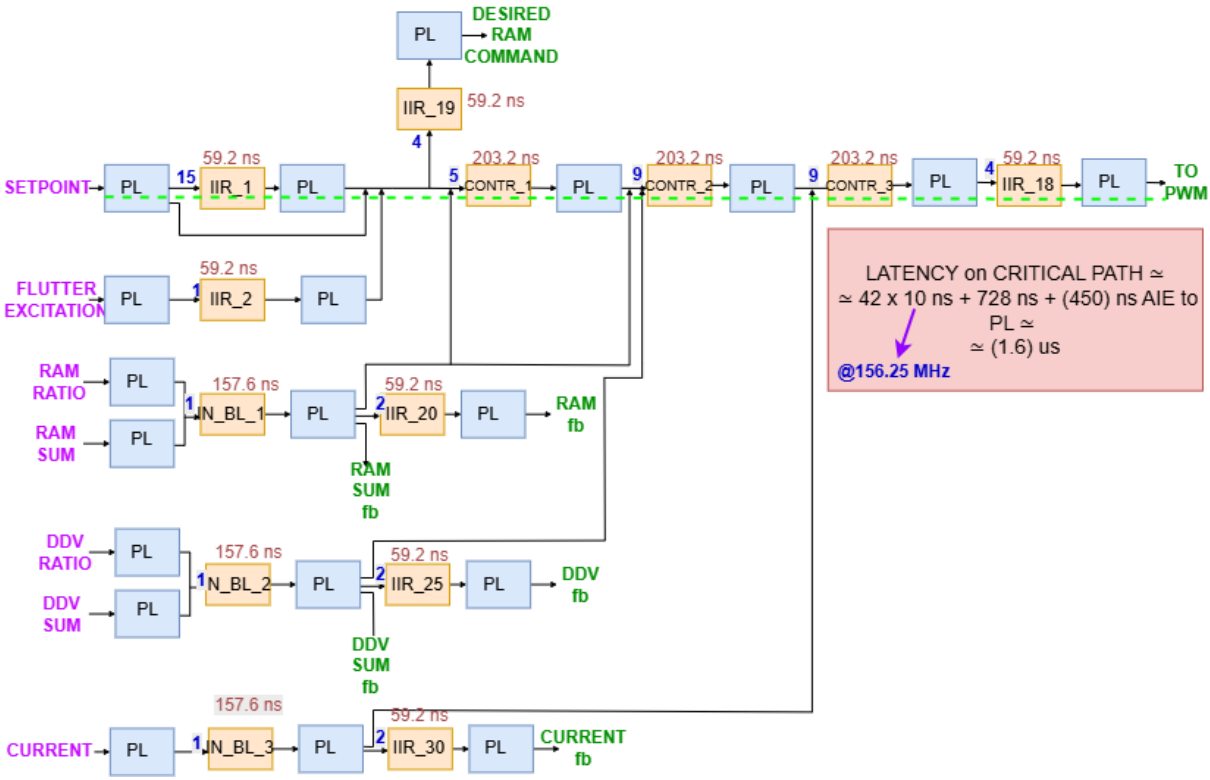


Figure 4.42. Latency estimation for the whole ACE. Blue number identifies the number of clock cycles to reach the destination (considering the input that generates it, as in Figure 4.16), while the green dotted line identifies the longest path.

Even if this algorithm is not much parallelizable, its implementation on Versal Adaptive SoC makes easily possible to obtain a minimum latency of the whole ACE of less than $2\mu s$, which is less than 2% of the requirement.

4.4 Code generation and test of the ACE

In order to proceed with a test of the ACE model described in Section 4.2, some modification to the Simulink testbench has to be performed: on Model Composer 2025.1, even if not directly synthesized, testbench block such as buses or discrete time integrator breaks the timing analysis and so the whole code generation itself.

Since neither the PL nor the AIE subsystem is a problem itself, once the numerical behavior of the system has been approved, they have been transferred into a new blank project, and the input has been generated randomly, just to produce the golden reference data to populate the testbench for the simulation.

The new Simulink system is reported in Figure 4.43.

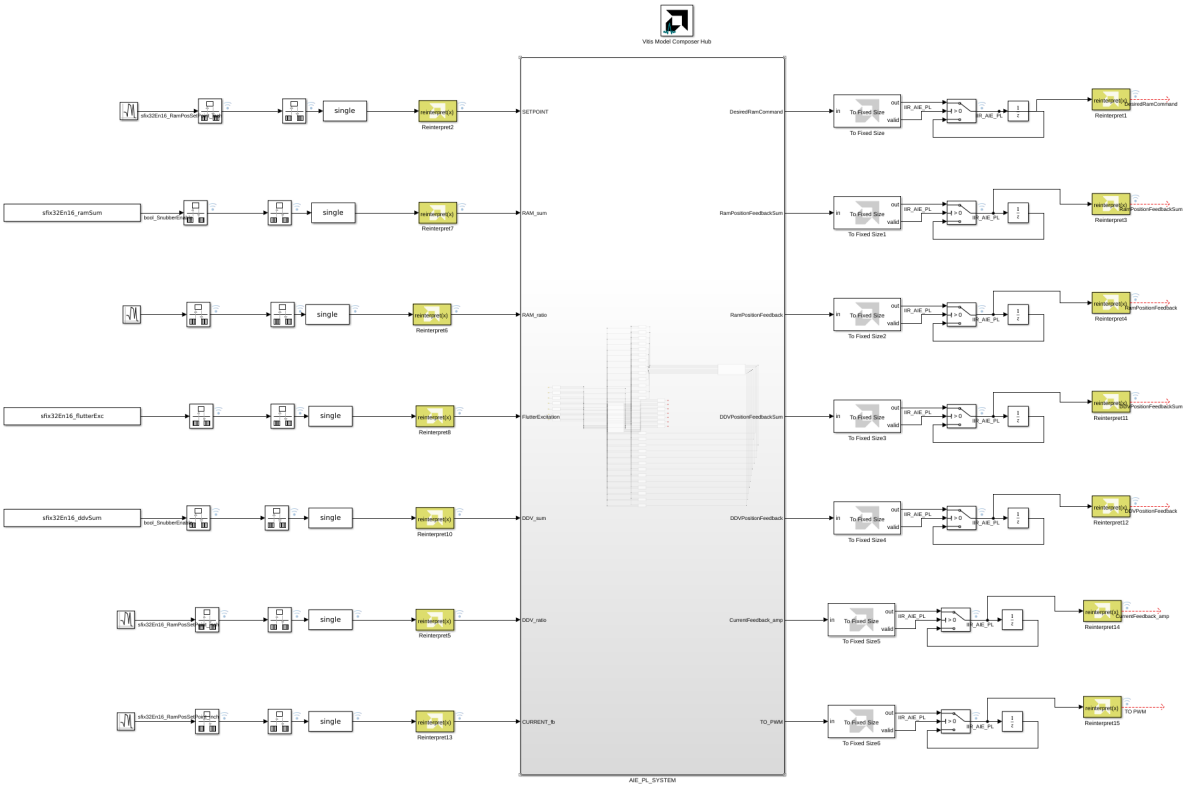


Figure 4.43. Simulink system to generate the `.bin` file.

The complete serial output generated by the host code (appendix B.1) is:

```
[0.012]****************************************
[0.045]Xilinx Versal Platform Loader and Manager
[0.080]Release 2025.1   Jul 25 2025  -  09:27:57
[0.116]Platform Version: v2.0 PMC: v2.0, PS: v2.0
[0.158]BOOTMODE: 0x0, MULTIBOOT: 0x0
[0.188]****************************************
[0.407]Non Secure Boot
[3.638]PLM Initialization Time
```

```
[3.667]***********Boot PDI Load: Started***********
[3.725]Loading PDI from SBI
[3.751]Monolithic/Master Device
[4.009]0.299 ms: PDI initialization time
[4.045]+++Loading Image#: 0x1, Name: lpd, Id: 0x04210002
[4.091]---Loading Partition#: 0x1, Id: 0xC
[58.415] 54.280 ms for Partition#: 0x1, Size: 10544 Bytes
[63.423]---Loading Partition#: 0x2, Id: 0xB
[104.556] 37.257 ms for Partition#: 0x2, Size: 63008 Bytes
[107.065]+++Loading Image#: 0x2, Name: pl_cfi, Id: 0x18700000
[112.295]---Loading Partition#: 0x3, Id: 0x3
[2114.011] 1997.753 ms for Partition#: 0x3, Size: 3189232 Bytes
[2116.767]---Loading Partition#: 0x4, Id: 0x5
[3040.845] 920.029 ms for Partition#: 0x4, Size: 1438112 Bytes
[3043.527]+++Loading Image#: 0x3, Name: cpm, Id: 0x04218007
[3048.760]---Loading Partition#: 0x5, Id: 0x6
[3052.994] 0.190 ms for Partition#: 0x5, Size: 2224 Bytes
[3057.886]+++Loading Image#: 0x4, Name: aie_subsys, Id: 0x0421C005
[3063.718]---Loading Partition#: 0x6, Id: 0x7
[3070.677] 2.912 ms for Partition#: 0x6, Size: 1936 Bytes
[3072.930]+++Loading Image#: 0x5, Name: fpd, Id: 0x0420C003
[3078.160]---Loading Partition#: 0x7, Id: 0x8
[3084.568] 2.362 ms for Partition#: 0x7, Size: 4544 Bytes
[3087.286]+++Loading Image#: 0x6, Name: aie_dev_part, Id: 0x18800000
[3093.291]---Loading Partition#: 0x8, Id: 0x0
[3099.165] 1.826 ms for Partition#: 0x8, Size: 7712 Bytes
[3102.421]+++Loading Image#: 0x7, Name: aie_image, Id: 0x18800000
[3108.167]---Loading Partition#: 0x9, Id: 0x0
[3160.557] 48.344 ms for Partition#: 0x9, Size: 82400 Bytes
[3163.264]+++Loading Image#: 0x8, Name: default_subsys, Id: 0x1C000000
[3169.161]---Loading Partition#: 0xA, Id: 0x0
[4433.118] 1259.907 ms for Partition#: 0xA, Size: 2011344 Bytes
[4435.931]***********Boot PDI LInitializing AIE driver...
Initializing ADF API...
XAIEFAL: INFO: RePLMrce group Avail is created.
XAIEFAL: INFO: Resource group Static is created.
XAIEFAL: INFO: Resource group Generic is created.
Beginning test
in0=188748
in1=188648
in2=1ec790
in3=1ecb90
in4=188e48
in5=1eca90
in6=188d48
out0=244720
out1=244830
out2=244940
out3=244a50
out4=244b60
out5=244c70
out6=244d80
Starting test w/ cu
```

```
AI Engine graph init
Initializing graph mygraph...
AI Engine graph run
Calling adf::graph::run() without specifying number of iterations ... It will
↪  either run AIE cores indefinitely, or run AIE cores for t.
Enabling core(s) of graph mygraph
50 out of 64 samples have been processed for new_desired_ram_cmd...
64 out of 64 samples have been processed for new_ram_sum_out...
64 out of 64 samples have been processed for new_ram_pos_fb...
64 out of 64 samples have been processed for new_ddv_sum_out...
64 out of 64 samples have been processed for new_ddv_position_feedback...
64 out of 64 samples have been processed for new_currentfeedback_amp...
64 out of 64 samples have been processed for new_to_pwm...

******************************** Test Results ********************************

****** Model Composer and Hardware outputs match for all 64 samples for output
↪  signal new_desired_ram_cmd
****** Model Composer and Hardware outputs match for all 64 samples for output
↪  signal new_ram_sum_out
****** Model Composer and Hardware outputs match for all 64 samples for output
↪  signal new_ram_pos_fb
****** Model Composer and Hardware outputs match for all 64 samples for output
↪  signal new_ddv_sum_out
****** Model Composer and Hardware outputs match for all 64 samples for output
↪  signal new_ddv_position_feedback
****** Model Composer and Hardware outputs match for all 64 samples for output
↪  signal new_currentfeedback_amp
****** Model Composer and Hardware outputs match for all 64 samples for output
↪  signal new_to_pwm

****** TEST PASSED

****** VMC_TEST_DONE
```

4.4.1 Results & performances

In order to measure the real latencies on the board, the generated Vivado project has been modified, adding an ILA triggered at startup as shown in Section 4.3.

The new Vivado block diagram is reported in Figure 4.44.

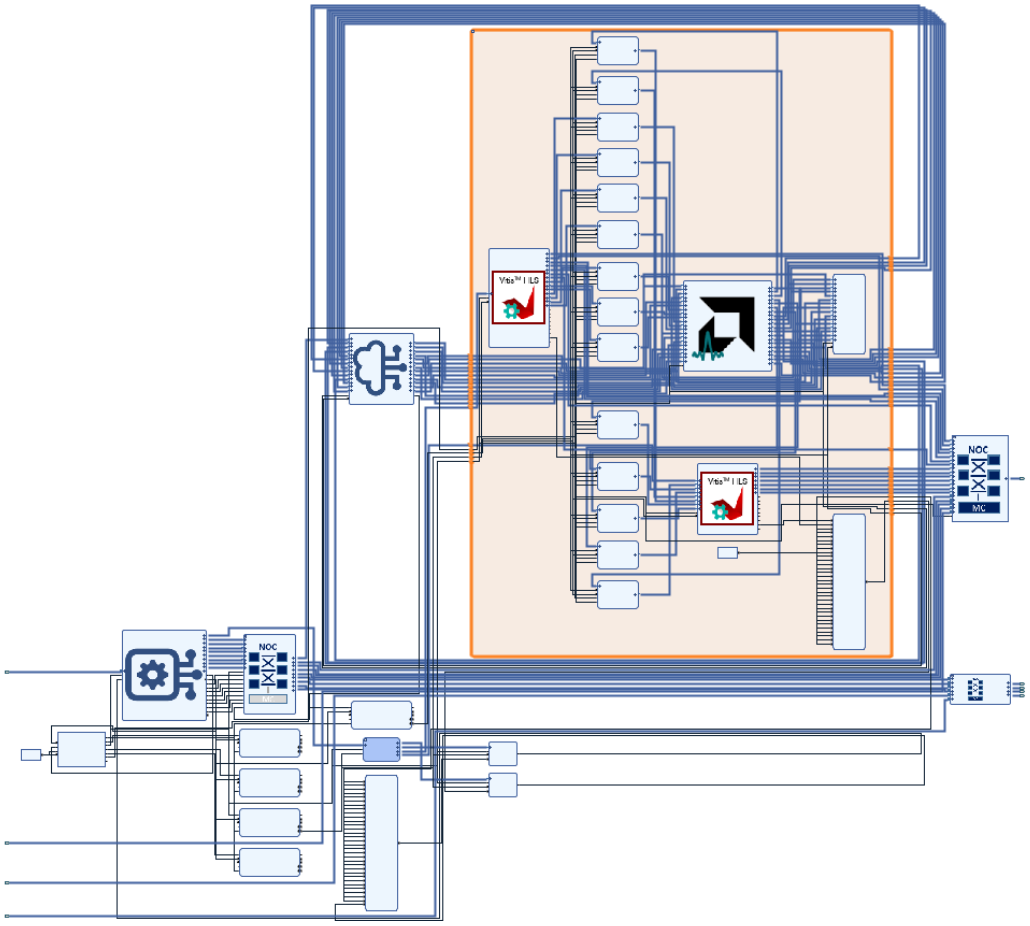


Figure 4.44. ACE project's block diagram.

The base blocks are the same as the simple example one, but the PL subsystem (central block in the orange square) is much more complex.

Latency measurement

The ILA is connected to PL-relevant input/output, and therefore, differently from Section 4.3, there is no extra latency due to measurements made outside the PL.

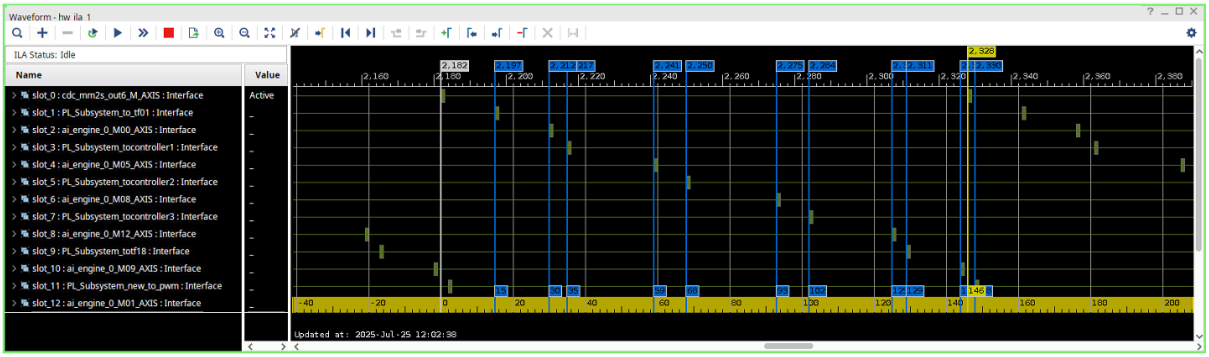


Figure 4.45. Measurement of the latency with the ILA. The scale is in terms of clock cycles (1 cycle = 10 ns). The white checkpoint corresponds to the start of the elaboration, while the yellow line corresponds to the end of the elaboration.

In Figure 4.45, the latency measurement is reported. This is a list of the ILA slots (the ones on the longest path of Figure 4.42) and their latencies from the setpoint ingress inside the PL in terms of PL clock cycles (the frequency is 100 MHz):

- SLOT_0: SETPOINT input -> 0 cycles;
- SLOT_1: IIR_1 input -> 15 cycles;
- SLOT_2: IIR_1 output -> 30 cycles;
- SLOT_3: CONTR_1 input -> 35 cycles;
- SLOT_4: CONTR_1 output -> 59 cycles;
- SLOT_5: CONTR_2 input -> 68 cycles;
- SLOT_6: CONTR_2 output -> 93 cycles;
- SLOT_7: CONTR_3 input -> 102 cycles;
- SLOT_8: CONTR_3 output -> 125 cycles;
- SLOT_9: IIR_18 input -> 129 cycles;
- SLOT_10: IIR_18 output -> 144 cycles;
- SLOT_11: TO_PWM output -> 148 cycles.

All the latencies are cyclically repeated until all the testbench input data are elaborated, as shown in Figure 4.46.

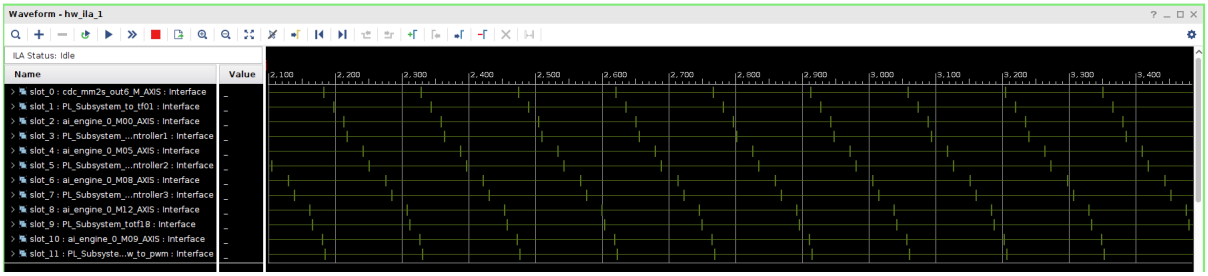


Figure 4.46. Periodicity of the system registered with the ILA.

The ALL_READY signal rises before the validity of the output (this event corresponds to the ingress of a new series of input, and in Figure 4.45 corresponds to the yellow line with 146 clock cycles of latency), because all is still needed is for the last branch to flush, and the system is ready to read other inputs.

Since the Initialization Interval is not exactly 100 μs, both for simplicity and to reduce occupation of the ILA waveforms, the set of 7 outputs is sent outside the PL once every time a new input is received (a logical AND of the 7 inputs TVALID is used as a temporization signal, in absence of an external one).
In Figure 4.47, the synchronicity of the output validity is shown.

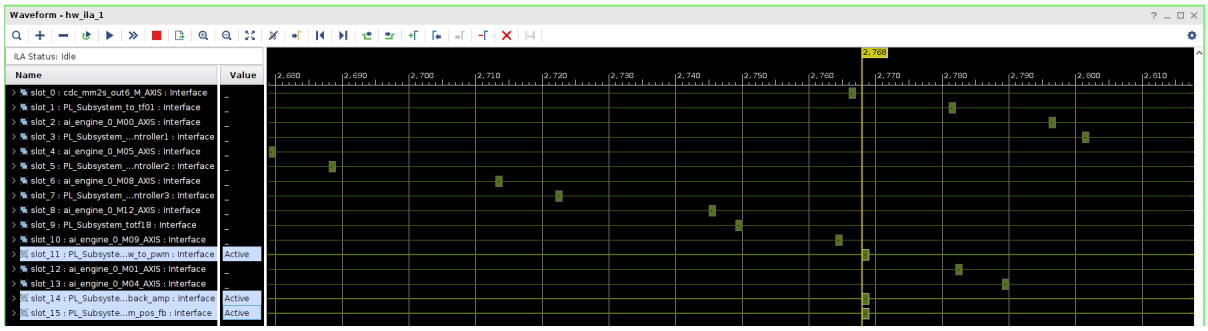


Figure 4.47. Synchronicity of the output validity registered with the ILA. Selected slots correspond to 3 of the 7 outputs (just because it is not possible to register more than 16 signals with ILA).

The overall latency to traverse all the graph is therefore 1.48 μs, which is very similar to the estimation made in Section 4.3.5.

4.4.2 Other reports

In Figure 4.48 the chip plan is reported, in Figure 4.49 the NoC utilization, and in Figure 4.50 the power report.

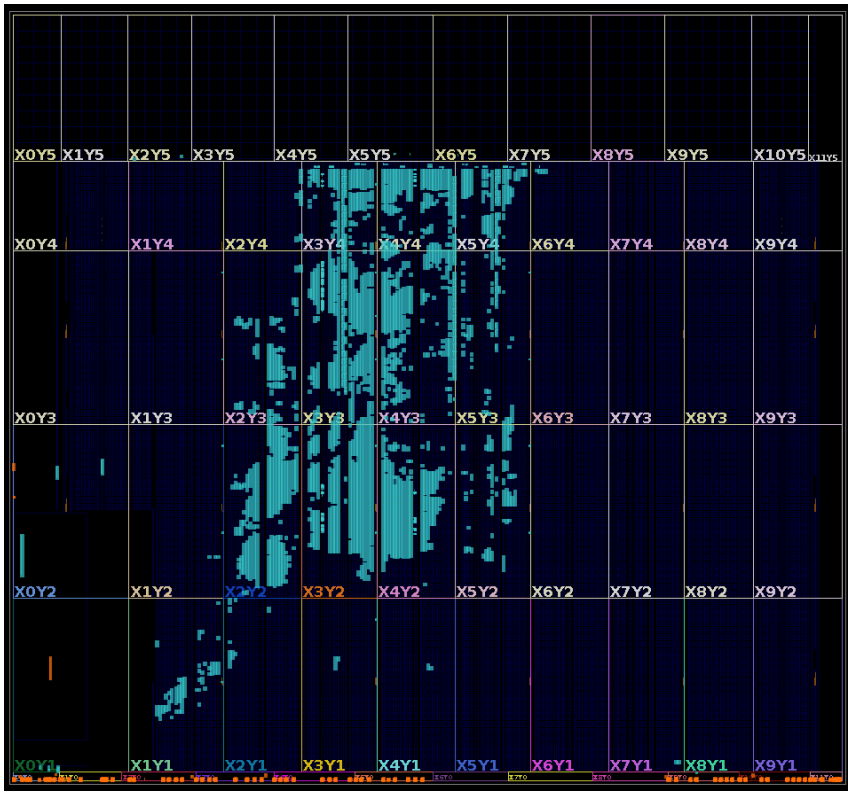


Figure 4.48. Chip plan of the final implementation. The AIE region corresponds to the Y5 row of the chip, but on Vivado, even if the tiles are used, just the interface ones are highlighted.

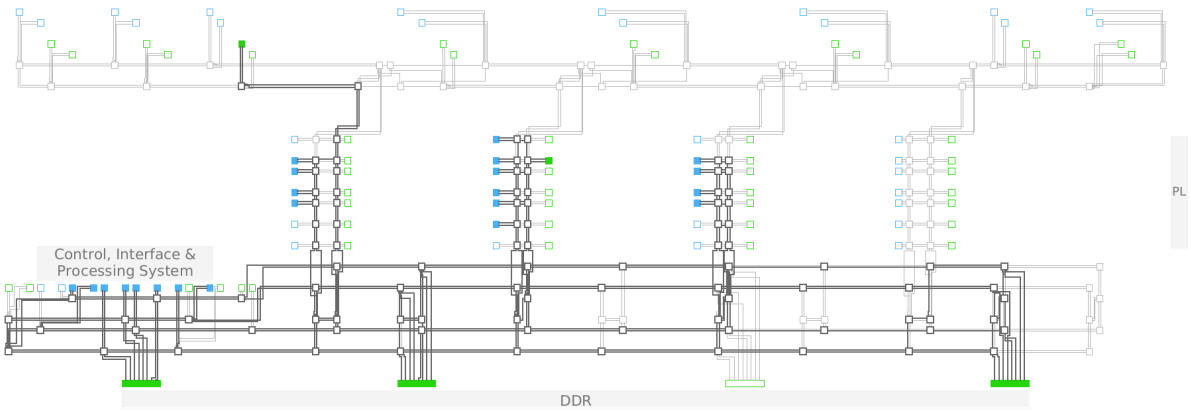


Figure 4.49. NoC utilization of the final implementation. The 14 middle blue squares correspond to PL input/output of the system, while AIE-PL communication does not employ the NoC, and so is not present in the diagram.

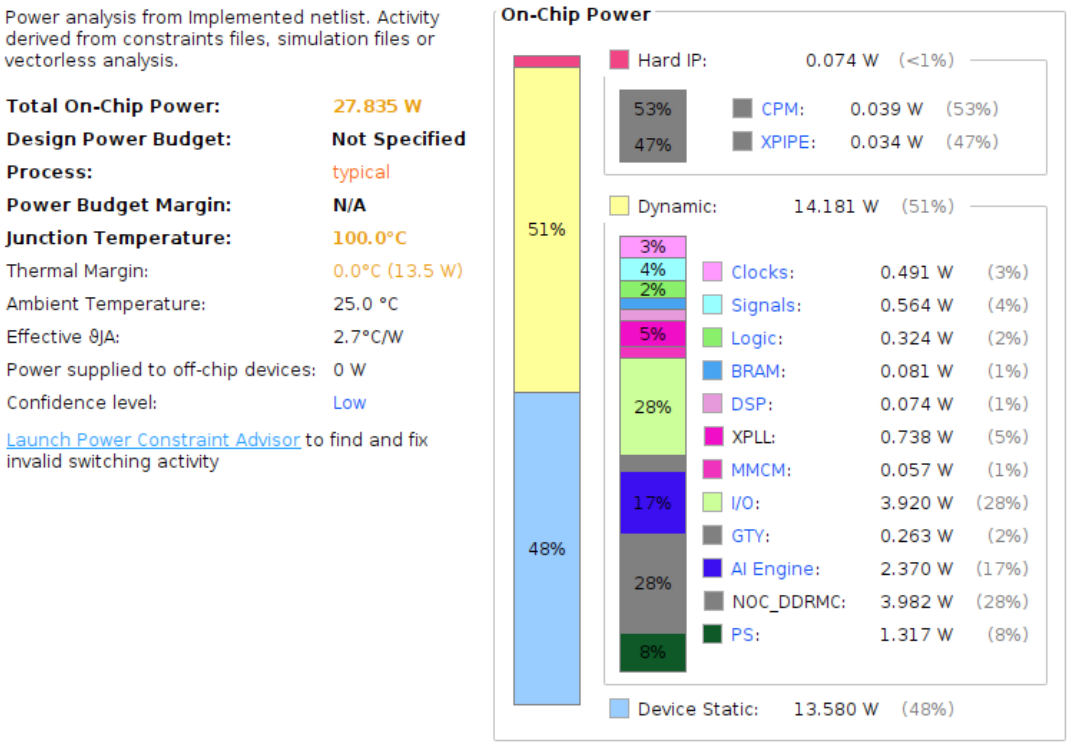


Figure 4.50. Power consumption estimation of the final implementation.

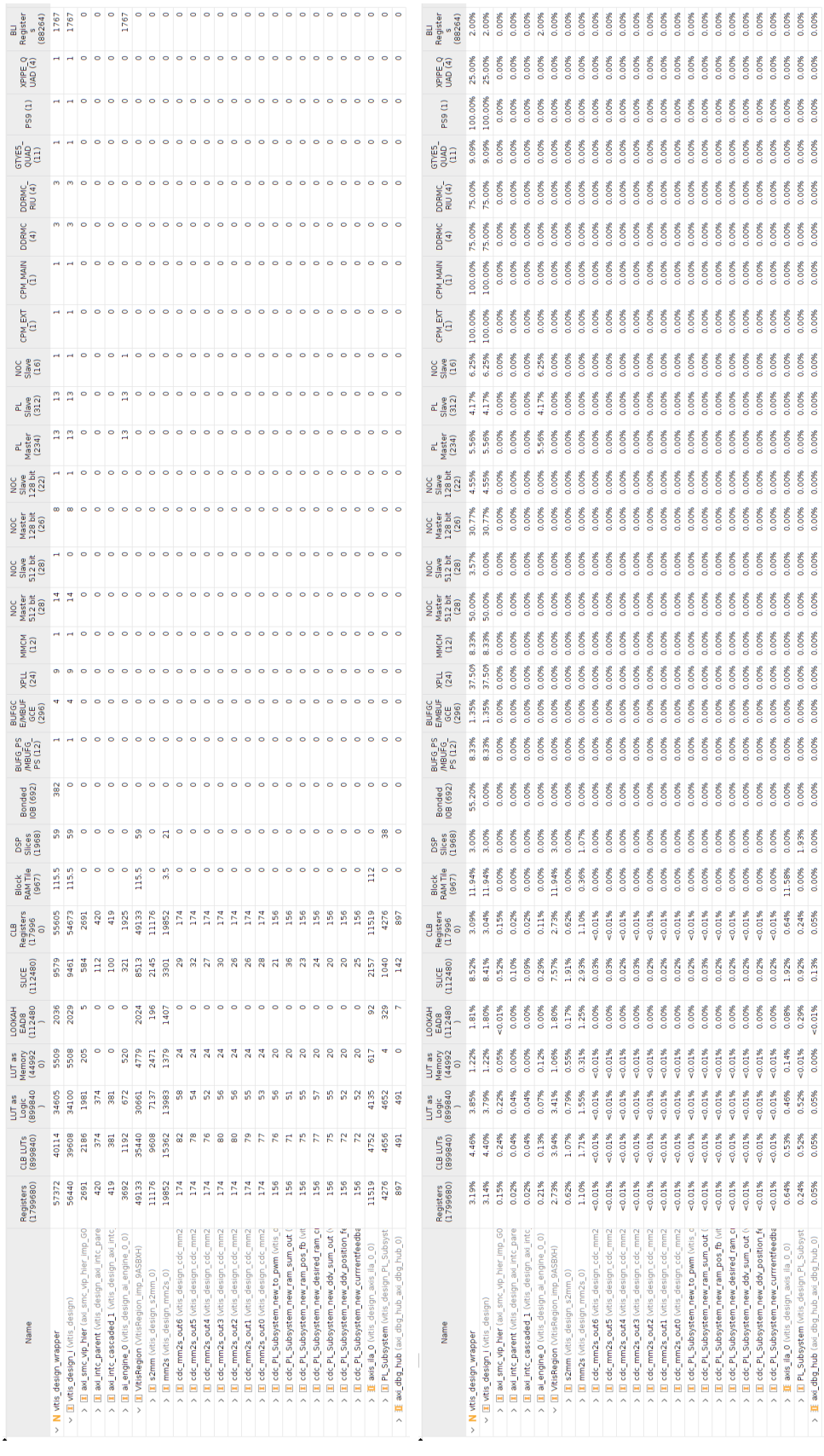
It is interesting to compare the result of the complete ACE implementation with the simple example one (subsection 4.3.4): the chip plan is much more colored, and in the NoC, much more data is traveling (from external DDR to PL input).

As concerns power utilization, the most relevant increases are DSP power (6.2x), logic power (3.3x), AIE (2.8x), BRAM (2.8x), clock (2x), NoC DDRMC (2x), and signals power (17x2). All other contributions remain more or less the same, and for this reason, the overall power consumption switching from the simple design to the complete ACE is just 1.2x.

The power consumption is much greater than the one obtained by implementing the ACE on a traditional FPGA (around 4 W), but by increasing the complexity of the algorithm, Versal ACAPs can achieve far better power consumptions, especially if a higher level of parallelism allows a better exploitation of the AI Engines. The same conclusions have been reached by [15].

However, due to the highly concentrated computation on a single chip, its power consumption may also reach 100 W. Such a delivery, with currents that may exceed 100 A, can be challenging to minimize voltage drop, losses, and temperature rises, as explained in [18].

The overall resource utilization is reported in Figure 4.51.



Name	Registers (1799680)	CLB LUTs (899840)	LUT as Logic (899840)	LUT as Memory (44992 0)	LOOKAH EAD8 (112480)	SLICE (112480)	CLB Registers (17996 0)	Block RAM Tile (967)	DSP Slices (1968)	Bonded IOB (692)	BUFG_PS /MBUFG_PS (12)	BUFGC E/MBUF GCE (296)	XPLL (24)	MMCM (12)	NOC Master 512 bit (28)	NOC Slave 512 bit (28)	NOC Master 128 bit (26)	NOC Slave 128 bit (22)	PL Master (234)	PL Slave (312)	NOC Slave (16)	CPM_EXT (1)	CPM_MAIN (1)	DDRMC (4)	DDRMC_ RIU (4)	GTYE5_ QUAD (11)	PS9 (1)	XPIPE_Q UAD (4)	BLI Register s (88264)
N vtis_design_wrapper	57372	40114	34605	5509	2036	9579	55605	115.5	59	382	1	4	9	1	14	1	8	1	13	13	1	1	1	3	3	1	1	1	1767
vtis_design_i (vtis_design)	56440	39608	34100	5508	2029	9461	54673	115.5	59	0	1	4	9	1	14	0	8	1	13	13	1	1	1	3	3	1	1	1	1767
axi_smc_vip_hier (axi_smc_vip_hier_imp_G0	2691	2186	1981	205	5	584	2691	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
axi_intc_parent (vtis_design_axi_intc_pare	420	374	374	0	0	112	420	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
axi_intc_cascaded_1 (vtis_design_axi_intc	419	381	381	0	0	100	419	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ai_engine_0 (vtis_design_ai_engine_0_0)	3692	1192	672	520	0	321	1925	0	0	0	0	0	0	0	0	0	0	0	13	13	1	0	0	0	0	0	0	0	1767
VitisRegion (VitisRegion_imp_9AS8XH)	49133	35440	30661	4779	2024	8513	49133	115.5	59	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
s2mm (vtis_design_s2mm_0)	11176	9608	7137	2471	196	2145	11176	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
mm2s (vtis_design_mm2s_0)	19852	15362	13983	1379	1407	3301	19852	3.5	21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
cdc_mm2s_out6 (vtis_design_cdc_mm2	174	82	58	24	0	29	174	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
cdc_mm2s_out5 (vtis_design_cdc_mm2	174	78	54	24	0	32	174	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
cdc_mm2s_out4 (vtis_design_cdc_mm2	174	76	52	24	0	27	174	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
cdc_mm2s_out3 (vtis_design_cdc_mm2	174	80	56	24	0	30	174	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
cdc_mm2s_out2 (vtis_design_cdc_mm2	174	80	56	24	0	26	174	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
cdc_mm2s_out1 (vtis_design_cdc_mm2	174	79	55	24	0	26	174	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
cdc_mm2s_out0 (vtis_design_cdc_mm2	174	77	53	24	0	28	174	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
cdc_PL_Subsystem_new_to_pwm (vtis_d	156	76	56	20	0	21	156	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
cdc_PL_Subsystem_new_ram_sum_out (v	156	71	51	20	0	36	156	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
cdc_PL_Subsystem_new_ram_pos_fb (vt	156	75	55	20	0	23	156	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
cdc_PL_Subsystem_new_desired_ram_ci	156	77	57	20	0	24	156	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
cdc_PL_Subsystem_new_ddv_sum_out (v	156	75	55	20	0	20	156	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
cdc_PL_Subsystem_new_ddv_position_fe	156	72	52	20	0	20	156	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
cdc_PL_Subsystem_new_currentfeedba	156	72	52	20	0	25	156	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
axis_ila_0 (vtis_design_axis_ila_0_0)	11519	4752	4135	617	92	2157	11519	112	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
PL_Subsystem (vtis_design_PL_Subsyst	4276	4656	4652	4	329	1040	4276	0	38	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
axi_dbg_hub (axi_dbg_hub_axi_dbg_hub_0)	897	491	491	0	7	142	897	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Name	Registers (1799680)	CLB LUTs (899840)	LUT as Logic (899840)	LUT as Memory (44992 0)	LOOKAH EAD8 (112480)	SLICE (112480)	CLB Registers (17996 0)	Block RAM Tile (967)	DSP Slices (1968)	Bonded IOB (692)	BUFG_PS /MBUFG_PS (12)	BUFGC E/MBUF GCE (296)	XPLL (24)	MMCM (12)	NOC Master 512 bit (28)	NOC Slave 512 bit (28)	NOC Master 128 bit (26)	NOC Slave 128 bit (22)	PL Master (234)	PL Slave (312)	NOC Slave (16)	CPM_EXT (1)	CPM_MAIN (1)	DDRMC (4)	DDRMC_ RIU (4)	GTYE5_ QUAD (11)	PS9 (1)	XPIPE_Q UAD (4)	BLI Register s (88264)
N vtis_design_wrapper	3.19%	4.46%	3.85%	1.22%	1.81%	8.52%	3.09%	11.94%	3.00%	55.20%	8.33%	1.35%	37.50%	8.33%	50.00%	3.57%	30.77%	4.55%	5.56%	4.17%	6.25%	100.00%	100.00%	75.00%	75.00%	9.09%	100.00%	25.00%	2.00%
vtis_design_i (vtis_design)	3.14%	4.40%	3.79%	1.22%	1.80%	8.41%	3.04%	11.94%	3.00%	0.00%	8.33%	1.35%	37.50%	8.33%	50.00%	0.00%	30.77%	4.55%	5.56%	4.17%	6.25%	100.00%	100.00%	75.00%	75.00%	9.09%	100.00%	25.00%	2.00%
axi_smc_vip_hier (axi_smc_vip_hier_imp_G0	0.15%	0.24%	0.22%	0.05%	<0.01%	0.52%	0.15%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
axi_intc_parent (vtis_design_axi_intc_pare	0.02%	0.04%	0.04%	0.00%	0.00%	0.10%	0.02%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
axi_intc_cascaded_1 (vtis_design_axi_intc	0.02%	0.04%	0.04%	0.00%	0.00%	0.09%	0.02%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
ai_engine_0 (vtis_design_ai_engine_0_0)	0.21%	0.13%	0.07%	0.12%	0.00%	0.29%	0.11%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
VitisRegion (VitisRegion_imp_9AS8XH)	2.73%	3.94%	3.41%	1.06%	1.80%	7.57%	2.73%	11.94%	3.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
s2mm (vtis_design_s2mm_0)	0.62%	1.07%	0.79%	0.55%	0.17%	1.91%	0.62%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
mm2s (vtis_design_mm2s_0)	1.10%	1.71%	1.55%	0.31%	1.25%	2.93%	1.10%	0.36%	1.07%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
cdc_mm2s_out6 (vtis_design_cdc_mm2	<0.01%	<0.01%	<0.01%	<0.01%	0.00%	0.03%	<0.01%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
cdc_mm2s_out5 (vtis_design_cdc_mm2	<0.01%	<0.01%	<0.01%	<0.01%	0.00%	0.03%	<0.01%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
cdc_mm2s_out4 (vtis_design_cdc_mm2	<0.01%	<0.01%	<0.01%	<0.01%	0.00%	0.02%	<0.01%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
cdc_mm2s_out3 (vtis_design_cdc_mm2	<0.01%	<0.01%	<0.01%	<0.01%	0.00%	0.03%	<0.01%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
cdc_mm2s_out2 (vtis_design_cdc_mm2	<0.01%	<0.01%	<0.01%	<0.01%	0.00%	0.02%	<0.01%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
cdc_mm2s_out1 (vtis_design_cdc_mm2	<0.01%	<0.01%	<0.01%	<0.01%	0.00%	0.02%	<0.01%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
cdc_mm2s_out0 (vtis_design_cdc_mm2	<0.01%	<0.01%	<0.01%	<0.01%	0.00%	0.02%	<0.01%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
cdc_PL_Subsystem_new_to_pwm (vtis_d	<0.01%	<0.01%	<0.01%	<0.01%	0.00%	0.02%	<0.01%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
cdc_PL_Subsystem_new_ram_sum_out (v	<0.01%	<0.01%	<0.01%	<0.01%	0.00%	0.03%	<0.01%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
cdc_PL_Subsystem_new_ram_pos_fb (vt	<0.01%	<0.01%	<0.01%	<0.01%	0.00%	0.02%	<0.01%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
cdc_PL_Subsystem_new_desired_ram_ci	<0.01%	<0.01%	<0.01%	<0.01%	0.00%	0.02%	<0.01%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
cdc_PL_Subsystem_new_ddv_sum_out (v	<0.01%	<0.01%	<0.01%	<0.01%	0.00%	0.02%	<0.01%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
cdc_PL_Subsystem_new_ddv_position_fe	<0.01%	<0.01%	<0.01%	<0.01%	0.00%	0.02%	<0.01%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
cdc_PL_Subsystem_new_currentfeedba	<0.01%	<0.01%	<0.01%	<0.01%	0.00%	0.02%	<0.01%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
axis_ila_0 (vtis_design_axis_ila_0_0)	0.64%	0.53%	0.46%	0.14%	0.08%	1.92%	0.64%	11.58%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
PL_Subsystem (vtis_design_PL_Subsyst	0.24%	0.24%	0.52%	<0.01%	0.29%	0.92%	0.24%	0.00%	1.93%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
axi_dbg_hub (axi_dbg_hub_axi_dbg_hub_0)	0.05%	0.05%	0.05%	0.00%	<0.01%	0.13%	0.05%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%

Figure 4.51. Resources utilization of the final implementation (absolute values on the left and % on the right).

It is interesting to look at the 13 PL Master and PL Slave, which are the interfaces between AIE and PL, and at the BRAM tiles, in which the ILA data is stored.

From the low % values, it is clear that the chip is largely unused, and even if it may seem that the NoC master ports are almost fully employed, many of them have a low bandwidth occupation.

Chapter 5

Conclusions

5.1 Considerations on technology and tools

It is important to highlight that the toolchain used to program this technology is still relatively immature and not yet fully refined.

During the development of this thesis, several issues and errors were encountered—many of which have already been discussed in the previous chapters. The following are some of the most significant problems that led to major delays in the development process:

- `"HDDMProto::readMessage failed: bad length"` — This error occurred when pressing the "Analyze" button to launch an AIE simulation in Model Composer. It was resolved by uninstalling and reinstalling the tool.
- Limited documentation on AIE-ML floating-point accuracy — This caused several days of delay and required the opening of a support ticket with AMD. The issue was clarified once a more detailed version of the manual was provided.
- Timing analysis in Model Composer — This was not possible due to the presence of blocks external to the AMD subsystem. According to available information, the issue may be resolved in version 2025.2 of the tool.
- Simulation freeze — Even on a correctly configured machine (correct tool versions, valid licenses, and all required libraries), simulations sometimes froze indefinitely without displaying error messages. Although they appeared to be running, no actual CPU usage was observed. This issue remains unresolved.
- Hardware emulation in Vivado — This failed with the error: `"Neither AIE_WORK_DIR (AIE work directory) is set nor AIESIM_CONFIG. AIE simulation won't run without setting any one of these. Exiting simulation"`. No useful documentation was found, and a support ticket was opened.

While these issues highlight the current limitations of the toolchain, it is reasonable to expect that programming SoCs with these tools will become increasingly streamlined soon, as the ecosystem matures and documentation improves. However, at present, the development process can still be quite challenging and, at times, frustrating.

5.2 Summary and Conclusions

This research aimed to explore the capabilities of a new and versatile technology: the ACAP (Adaptive Compute Acceleration Platform).

These devices integrate, on a single chip, two scalar processors, programmable logic (PL), which can be used for data management, specific computations, or as a custom hierarchical memory, and AI Engines—VLIW (Very Long Instruction Word) SIMD (Single Instruction Multiple Data) computational units.

These AI Engines are arranged in a two-dimensional array, where each unit corresponds to one tile, located in a dedicated region of the device. Each tile operates at 1 GHz and, depending on data precision, can simultaneously perform from 8 single-precision floating-point (SPFP) MACs to 128 INT8 MACs.

The number of tiles can range from 30 to 300, and data transfer among them is managed through AXI4 protocols, a dedicated cascade mechanism, or shared memory—allowing direct communication between tiles without occupying the NoC (Network-on-Chip) bandwidth unnecessarily.

The NoC interconnects all chip components, enabling flexible partitioning of computation. In addition, a dedicated connection between the AI Engines and the PL is available.

The design process for exploiting this technology involves several steps, greatly simplified by AMD Vitis Model Composer, a high-level graphical development environment based on Simulink:

- Mapping the algorithm across the chip components (preferably using AI Engines for highly parallelizable computations);
- Programming AI Engine kernels in C/C++;
- Describing the PL hardware (Model Composer offers a library of common building blocks);
- Connecting the PL and AIE subsystems via AXI interfaces (again, simplified through dedicated Model Composer blocks);
- Synthesizing and analyzing the design to meet performance and resource constraints.

The selected case study was the Actuator Control Electronics (ACE): the actuator control loop of an FCC, composed of 34 second-order IIR filters connected through a complex data flow graph (DFG).

Unfortunately, due to data dependencies and the need for single-sample processing, this application is not ideal for a massively parallel architecture like the AIE array, which is capable of delivering several TFLOPs.

Nonetheless, its implementation is feasible—albeit underutilized—especially considering the flexibility of selectively activating only the required tiles, thus conserving power and resources.

Initial experiments also revealed that the number and order of filters processed in parallel had minimal impact on performance when using AI Engines as the primary computational units.

Moreover, multiple kernels can be deployed within a single tile, and switching from fixed-point to floating-point arithmetic may offer significant benefits in terms of dynamic

range and precision—without incurring a noticeable latency penalty.

From a performance standpoint, as shown in Chapter 4, the entire ACE can be traversed in less than $2 : \mu s$, which is more than 50 times faster than the original requirement.

Regarding reliability, studies in [17] and [14] highlight the effectiveness of built-in mitigation techniques and the XilSEM firmware in preventing uncorrectable errors, ensuring high dependability in safety-critical applications such as avionics and automotive.

For these reasons, this technology presents a compelling option for enhancing hardware flexibility—e.g., increasing filter order or count. However, it must be noted that other applications with much higher intrinsic parallelism—such as radar signal processing or convolutional neural networks (CNNs)—stand to benefit far more from the computational acceleration offered by AI Engines.

Future research in such domains is certainly recommended.

Appendix A

Math demonstrations

A.1 Two IIR order 2 filter compressed in a single IIR order 4

Supposing to have two IIR filters of order 2 in series, the overall transfer function is given by the product of the standalone transfer functions of each filter.

Given

$$F_1(z) = \frac{B_1 + B_2 \cdot z^{-1} + B_3 \cdot z^{-2}}{1 + A_2 \cdot z^{-1} + A_3 \cdot z^{-2}} \tag{A.1}$$

and

$$F_2(z) = \frac{C_1 + C_2 \cdot z^{-1} + C_3 \cdot z^{-2}}{1 + D_2 \cdot z^{-1} + D_3 \cdot z^{-2}} \tag{A.2}$$

so the overall $H(z)$ will be:

$$H(z) = F_1(z) \cdot F_2(z) = \tag{A.3}$$

$$= \frac{B_1 + B_2 \cdot z^{-1} + B_3 \cdot z^{-2}}{1 + A_2 \cdot z^{-1} + A_3 \cdot z^{-2}} \cdot \frac{C_1 + C_2 \cdot z^{-1} + C_3 \cdot z^{-2}}{1 + D_2 \cdot z^{-1} + D_3 \cdot z^{-2}} = \tag{A.4}$$

$$= \frac{B_1C_1 + (B_1C_2 + B_2C_1) \cdot z^{-1} + (B_1C_3 + B_3C_1 + B_2C_2) \cdot z^{-2} + (B_2C_3 + B_3C_2) \cdot z^{-3} + B_3C_3 \cdot z^{-4}}{1 + (A_2 + D_2) \cdot z^{-1} + (A_3 + D_3 + A_2D_2) \cdot z^{-2} + (A_2D_3 + A_3D_2) \cdot z^{-3} + A_3D_3 \cdot z^{-4}} \tag{A.5}$$

that can be considered as an IIR filter of order 4.

Appendix B

Model composer generated code

Codes in this Section are relative to the ACE project.

B.1 ACE host.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <unistd.h>
#include "platform.h"
#include "xparameters.h"
#include "xil_io.h"
#include "xil_cache.h"
#include "PL_Subsystem_new_currentfb.h"
#include "PL_Subsystem_new_ddv_ratio.h"
#include "PL_Subsystem_new_ddv_sum.h"
#include "PL_Subsystem_new_flutter_exc.h"
#include "PL_Subsystem_new_ram_ratio.h"
#include "PL_Subsystem_new_ram_sum.h"
#include "PL_Subsystem_new_setpoint.h"
#include "PL_Subsystem_new_desired_ram_cmd.h"
#include "PL_Subsystem_new_ram_sum_out.h"
#include "PL_Subsystem_new_ram_pos_fb.h"
#include "PL_Subsystem_new_ddv_sum_out.h"
#include "PL_Subsystem_new_ddv_position_feedback.h"
#include "PL_Subsystem_new_currrentfeedback_amp.h"
#include "PL_Subsystem_new_to_pwm.h"
#include "xmc_ps_main.h"
#include "../../ip/AIE_Subsystem/src/AIE_Subsystem.cpp"


/*****************************************************************************
*/
void InitData(uint32_t** out, int size)
{
    int i;
```

```
31     *out = (uint32_t*)malloc(sizeof(uint32_t) * size);
32
33     if(*out == NULL) {
34         printf("Memory Allocation failed."
35                "Please reduce the Sample Time in your Model Composer design"
36                "to reduce the number of samples in theoutput(s) of the design
       .\n");
37         exit(-1);
38     }
39
40     for(i = 0; i < size; i++) {
41         (*out)[i] = 0xABCDEF00;
42     }
43 }
44
45 /************************************************************************
       */
46
47 int RunTest(uint64_t mm2s_base,
48             uint64_t s2mm_base,
49             uint32_t* in0, int input0_size,
50             uint32_t* in1, int input1_size,
51             uint32_t* in2, int input2_size,
52             uint32_t* in3, int input3_size,
53             uint32_t* in4, int input4_size,
54             uint32_t* in5, int input5_size,
55             uint32_t* in6, int input6_size,
56             uint32_t* golden0,
57             uint32_t* golden1,
58             uint32_t* golden2,
59             uint32_t* golden3,
60             uint32_t* golden4,
61             uint32_t* golden5,
62             uint32_t* golden6,
63             uint32_t* out0, int output0_size,
64             uint32_t* out1, int output1_size,
65             uint32_t* out2, int output2_size,
66             uint32_t* out3, int output3_size,
67             uint32_t* out4, int output4_size,
68             uint32_t* out5, int output5_size,
69             uint32_t* out6, int output6_size)
70 {
71     int i;
72     int errCount = 0;
73     int totalErrCount = 0;
74     uint64_t mem_in0Addr = (uint64_t)in0;
75     printf("in0=%llx\n", mem_in0Addr);
76     uint64_t mem_in1Addr = (uint64_t)in1;
77     printf("in1=%llx\n", mem_in1Addr);
78     uint64_t mem_in2Addr = (uint64_t)in2;
79     printf("in2=%llx\n", mem_in2Addr);
80     uint64_t mem_in3Addr = (uint64_t)in3;
81     printf("in3=%llx\n", mem_in3Addr);
82     uint64_t mem_in4Addr = (uint64_t)in4;
83     printf("in4=%llx\n", mem_in4Addr);
84     uint64_t mem_in5Addr = (uint64_t)in5;
85     printf("in5=%llx\n", mem_in5Addr);
86     uint64_t mem_in6Addr = (uint64_t)in6;
```

```
87      printf("in6=%llx\n", mem_in6Addr);
88      uint64_t mem_out0Addr = (uint64_t)out0;
89      printf("out0=%llx\n", mem_out0Addr);
90      uint64_t mem_out1Addr = (uint64_t)out1;
91      printf("out1=%llx\n", mem_out1Addr);
92      uint64_t mem_out2Addr = (uint64_t)out2;
93      printf("out2=%llx\n", mem_out2Addr);
94      uint64_t mem_out3Addr = (uint64_t)out3;
95      printf("out3=%llx\n", mem_out3Addr);
96      uint64_t mem_out4Addr = (uint64_t)out4;
97      printf("out4=%llx\n", mem_out4Addr);
98      uint64_t mem_out5Addr = (uint64_t)out5;
99      printf("out5=%llx\n", mem_out5Addr);
100     uint64_t mem_out6Addr = (uint64_t)out6;
101     printf("out6=%llx\n", mem_out6Addr);
102
103     printf("Starting test w/ cu\n");
104
105     Xil_Out32(mm2s_base + M_IN0_OFFSET, (uint32_t) mem_in0Addr);
106     Xil_Out32(mm2s_base + M_IN0_OFFSET + 4, 0);
107     Xil_Out32(mm2s_base + M_IN0_SIZE_OFFSET, input0_size);
108
109     Xil_Out32(mm2s_base + M_IN1_OFFSET, (uint32_t) mem_in1Addr);
110     Xil_Out32(mm2s_base + M_IN1_OFFSET + 4, 0);
111     Xil_Out32(mm2s_base + M_IN1_SIZE_OFFSET, input1_size);
112
113     Xil_Out32(mm2s_base + M_IN2_OFFSET, (uint32_t) mem_in2Addr);
114     Xil_Out32(mm2s_base + M_IN2_OFFSET + 4, 0);
115     Xil_Out32(mm2s_base + M_IN2_SIZE_OFFSET, input2_size);
116
117     Xil_Out32(mm2s_base + M_IN3_OFFSET, (uint32_t) mem_in3Addr);
118     Xil_Out32(mm2s_base + M_IN3_OFFSET + 4, 0);
119     Xil_Out32(mm2s_base + M_IN3_SIZE_OFFSET, input3_size);
120
121     Xil_Out32(mm2s_base + M_IN4_OFFSET, (uint32_t) mem_in4Addr);
122     Xil_Out32(mm2s_base + M_IN4_OFFSET + 4, 0);
123     Xil_Out32(mm2s_base + M_IN4_SIZE_OFFSET, input4_size);
124
125     Xil_Out32(mm2s_base + M_IN5_OFFSET, (uint32_t) mem_in5Addr);
126     Xil_Out32(mm2s_base + M_IN5_OFFSET + 4, 0);
127     Xil_Out32(mm2s_base + M_IN5_SIZE_OFFSET, input5_size);
128
129     Xil_Out32(mm2s_base + M_IN6_OFFSET, (uint32_t) mem_in6Addr);
130     Xil_Out32(mm2s_base + M_IN6_OFFSET + 4, 0);
131     Xil_Out32(mm2s_base + M_IN6_SIZE_OFFSET, input6_size);
132
133     Xil_Out32(s2mm_base + S_OUT0_OFFSET, (uint32_t) mem_out0Addr);
134     Xil_Out32(s2mm_base + S_OUT0_OFFSET + 4, 0);
135     Xil_Out32(s2mm_base + S_OUT0_SIZE_OFFSET, output0_size);
136
137     Xil_Out32(s2mm_base + S_OUT1_OFFSET, (uint32_t) mem_out1Addr);
138     Xil_Out32(s2mm_base + S_OUT1_OFFSET + 4, 0);
139     Xil_Out32(s2mm_base + S_OUT1_SIZE_OFFSET, output1_size);
140
141     Xil_Out32(s2mm_base + S_OUT2_OFFSET, (uint32_t) mem_out2Addr);
142     Xil_Out32(s2mm_base + S_OUT2_OFFSET + 4, 0);
143     Xil_Out32(s2mm_base + S_OUT2_SIZE_OFFSET, output2_size);
144
```

```
145     Xil_Out32(s2mm_base + S_OUT3_OFFSET, (uint32_t) mem_out3Addr);
146     Xil_Out32(s2mm_base + S_OUT3_OFFSET + 4, 0);
147     Xil_Out32(s2mm_base + S_OUT3_SIZE_OFFSET, output3_size);
148
149     Xil_Out32(s2mm_base + S_OUT4_OFFSET, (uint32_t) mem_out4Addr);
150     Xil_Out32(s2mm_base + S_OUT4_OFFSET + 4, 0);
151     Xil_Out32(s2mm_base + S_OUT4_SIZE_OFFSET, output4_size);
152
153     Xil_Out32(s2mm_base + S_OUT5_OFFSET, (uint32_t) mem_out5Addr);
154     Xil_Out32(s2mm_base + S_OUT5_OFFSET + 4, 0);
155     Xil_Out32(s2mm_base + S_OUT5_SIZE_OFFSET, output5_size);
156
157     Xil_Out32(s2mm_base + S_OUT6_OFFSET, (uint32_t) mem_out6Addr);
158     Xil_Out32(s2mm_base + S_OUT6_OFFSET + 4, 0);
159     Xil_Out32(s2mm_base + S_OUT6_SIZE_OFFSET, output6_size);
160
161     Xil_Out32(mm2s_base + M_PULSE_OFFSET, 2);
162     Xil_Out32(mm2s_base + M_CTRL_OFFSET, 1);
163     Xil_Out32(s2mm_base + S_CTRL_OFFSET, 1);
164
165     printf("AI Engine graph init\n");
166     mygraph.init();
167
168     printf("AI Engine graph run\n");
169     mygraph.run();
170
171     int iteration = 0;
172     int num_out0_samples = 0;
173     int num_out1_samples = 0;
174     int num_out2_samples = 0;
175     int num_out3_samples = 0;
176     int num_out4_samples = 0;
177     int num_out5_samples = 0;
178     int num_out6_samples = 0;
179     while(1) {
180        uint32_t v =
181           Xil_In32(s2mm_base + S_CTRL_OFFSET);
182        if(iteration % 1000 == 0) {
183
184           if(out0[0] != 0xABCDEF00) {
185              while((num_out0_samples < output0_size) &&
186                    (out0[num_out0_samples] != 0xABCDEF00)) {
187                 int incr = (output0_size < (num_out0_samples + 50)) ? 1 :
    50;
188                 num_out0_samples += incr;
189              }
190              printf("%d out of %d samples have been processed for
    new_desired_ram_cmd...\n", num_out0_samples, output0_size);
191           }
192
193           if(out1[0] != 0xABCDEF00) {
194              while((num_out1_samples < output1_size) &&
195                    (out1[num_out1_samples] != 0xABCDEF00)) {
196                 int incr = (output1_size < (num_out1_samples + 50)) ? 1 :
    50;
197                 num_out1_samples += incr;
198              }
```

```
199             printf("%d out of %d samples have been processed for
    new_ram_sum_out...\n", num_out1_samples, output1_size);
200         }
201 
202         if(out2[0] != 0xABCDEF00) {
203             while((num_out2_samples < output2_size) &&
204                   (out2[num_out2_samples] != 0xABCDEF00)) {
205                 int incr = (output2_size < (num_out2_samples + 50)) ? 1 :
    50;
206                 num_out2_samples += incr;
207             }
208             printf("%d out of %d samples have been processed for
    new_ram_pos_fb...\n", num_out2_samples, output2_size);
209         }
210 
211         if(out3[0] != 0xABCDEF00) {
212             while((num_out3_samples < output3_size) &&
213                   (out3[num_out3_samples] != 0xABCDEF00)) {
214                 int incr = (output3_size < (num_out3_samples + 50)) ? 1 :
    50;
215                 num_out3_samples += incr;
216             }
217             printf("%d out of %d samples have been processed for
    new_ddv_sum_out...\n", num_out3_samples, output3_size);
218         }
219 
220         if(out4[0] != 0xABCDEF00) {
221             while((num_out4_samples < output4_size) &&
222                   (out4[num_out4_samples] != 0xABCDEF00)) {
223                 int incr = (output4_size < (num_out4_samples + 50)) ? 1 :
    50;
224                 num_out4_samples += incr;
225             }
226             printf("%d out of %d samples have been processed for
    new_ddv_position_feedback...\n", num_out4_samples, output4_size);
227         }
228 
229         if(out5[0] != 0xABCDEF00) {
230             while((num_out5_samples < output5_size) &&
231                   (out5[num_out5_samples] != 0xABCDEF00)) {
232                 int incr = (output5_size < (num_out5_samples + 50)) ? 1 :
    50;
233                 num_out5_samples += incr;
234             }
235             printf("%d out of %d samples have been processed for
    new_currrentfeedback_amp...\n", num_out5_samples, output5_size);
236         }
237 
238         if(out6[0] != 0xABCDEF00) {
239             while((num_out6_samples < output6_size) &&
240                   (out6[num_out6_samples] != 0xABCDEF00)) {
241                 int incr = (output6_size < (num_out6_samples + 50)) ? 1 :
    50;
242                 num_out6_samples += incr;
243             }
244             printf("%d out of %d samples have been processed for new_to_pwm
    ...\n", num_out6_samples, output6_size);
245         }
```

```
246         }
247         ++iteration;
248         if(v & 6) {
249             break;
250         }
251     }
252 
253 
254 
255     printf("\n******************************** Test Results
    ********************************\n\n");
256 
257     errCount = 0;
258     for(i = 0;i < output0_size; ++i) {
259         if(out0[i] != golden0[i]) {
260             printf("Mismatch found for output signal new_desired_ram_cmd in
    sample %d."
261                    "Hardware output: %x,"
262                    "Model Composer output:%x\n",
263                    i, out0[i], golden0[i]);
264             ++errCount;
265         }
266     }
267     if (errCount == 0) {
268         printf("****** Model Composer and Hardware outputs match for all %d
    samples "
269                "for output signal new_desired_ram_cmd\n", output0_size);
270     } else {
271         printf("****** Mismatch(es) found between Model Composer and Hardware
     outputs "
272                "for output signal new_desired_ram_cmd\n");
273     }
274     totalErrCount += errCount;
275 
276     errCount = 0;
277     for(i = 0;i < output1_size; ++i) {
278         if(out1[i] != golden1[i]) {
279             printf("Mismatch found for output signal new_ram_sum_out in sample
     %d."
280                    "Hardware output: %x,"
281                    "Model Composer output:%x\n",
282                    i, out1[i], golden1[i]);
283             ++errCount;
284         }
285     }
286     if (errCount == 0) {
287         printf("****** Model Composer and Hardware outputs match for all %d
    samples "
288                "for output signal new_ram_sum_out\n", output1_size);
289     } else {
290         printf("****** Mismatch(es) found between Model Composer and Hardware
     outputs "
291                "for output signal new_ram_sum_out\n");
292     }
293     totalErrCount += errCount;
294 
295     errCount = 0;
296     for(i = 0;i < output2_size; ++i) {
```

```
297         if(out2[i] != golden2[i]) {
298             printf("Mismatch found for output signal new_ram_pos_fb in sample %d."
299                    "Hardware output: %x,"
300                    "Model Composer output:%x\n",
301                    i, out2[i], golden2[i]);
302             ++errCount;
303         }
304     }
305     if (errCount == 0) {
306         printf("****** Model Composer and Hardware outputs match for all %d samples "
307                "for output signal new_ram_pos_fb\n", output2_size);
308     } else {
309         printf("****** Mismatch(es) found between Model Composer and Hardware outputs "
310                "for output signal new_ram_pos_fb\n");
311     }
312     totalErrCount += errCount;
313 
314     errCount = 0;
315     for(i = 0;i < output3_size; ++i) {
316         if(out3[i] != golden3[i]) {
317             printf("Mismatch found for output signal new_ddv_sum_out in sample %d."
318                    "Hardware output: %x,"
319                    "Model Composer output:%x\n",
320                    i, out3[i], golden3[i]);
321             ++errCount;
322         }
323     }
324     if (errCount == 0) {
325         printf("****** Model Composer and Hardware outputs match for all %d samples "
326                "for output signal new_ddv_sum_out\n", output3_size);
327     } else {
328         printf("****** Mismatch(es) found between Model Composer and Hardware outputs "
329                "for output signal new_ddv_sum_out\n");
330     }
331     totalErrCount += errCount;
332 
333     errCount = 0;
334     for(i = 0;i < output4_size; ++i) {
335         if(out4[i] != golden4[i]) {
336             printf("Mismatch found for output signal new_ddv_position_feedback in sample %d."
337                    "Hardware output: %x,"
338                    "Model Composer output:%x\n",
339                    i, out4[i], golden4[i]);
340             ++errCount;
341         }
342     }
343     if (errCount == 0) {
344         printf("****** Model Composer and Hardware outputs match for all %d samples "
345                "for output signal new_ddv_position_feedback\n", output4_size);
```

```
346     } else {
347        printf("****** Mismatch(es) found between Model Composer and Hardware
         outputs "
348               "for output signal new_ddv_position_feedback\n");
349     }
350     totalErrCount += errCount;
351 
352     errCount = 0;
353     for(i = 0;i < output5_size; ++i) {
354        if(out5[i] != golden5[i]) {
355           printf("Mismatch found for output signal new_currrentfeedback_amp
        in sample %d."
356                  "Hardware output: %x,"
357                  "Model Composer output:%x\n",
358                  i, out5[i], golden5[i]);
359           ++errCount;
360        }
361     }
362     if (errCount == 0) {
363        printf("****** Model Composer and Hardware outputs match for all %d
        samples "
364               "for output signal new_currrentfeedback_amp\n", output5_size);
365     } else {
366        printf("****** Mismatch(es) found between Model Composer and Hardware
         outputs "
367               "for output signal new_currrentfeedback_amp\n");
368     }
369     totalErrCount += errCount;
370 
371     errCount = 0;
372     for(i = 0;i < output6_size; ++i) {
373        if(out6[i] != golden6[i]) {
374           printf("Mismatch found for output signal new_to_pwm in sample %d."
375                  "Hardware output: %x,"
376                  "Model Composer output:%x\n",
377                  i, out6[i], golden6[i]);
378           ++errCount;
379        }
380     }
381     if (errCount == 0) {
382        printf("****** Model Composer and Hardware outputs match for all %d
        samples "
383               "for output signal new_to_pwm\n", output6_size);
384     } else {
385        printf("****** Mismatch(es) found between Model Composer and Hardware
         outputs "
386               "for output signal new_to_pwm\n");
387     }
388     totalErrCount += errCount;
389 
390     return totalErrCount;
391 }
392 
393 /*************************************************************************
    */
394 int main()
395 {
396     uint32_t* PL_Subsystem_new_desired_ram_cmd;
```

```
397     uint32_t* PL_Subsystem_new_ram_sum_out;
398     uint32_t* PL_Subsystem_new_ram_pos_fb;
399     uint32_t* PL_Subsystem_new_ddv_sum_out;
400     uint32_t* PL_Subsystem_new_ddv_position_feedback;
401     uint32_t* PL_Subsystem_new_currrentfeedback_amp;
402     uint32_t* PL_Subsystem_new_to_pwm;
403     int totalErrCount;
404 
405     Xil_DCacheDisable();
406 
407     init_platform();
408     sleep(1);
409     printf("Beginning test\n");
410 
411     InitData(&PL_Subsystem_new_desired_ram_cmd, OUTPUT_0_SIZE);
412     InitData(&PL_Subsystem_new_ram_sum_out, OUTPUT_1_SIZE);
413     InitData(&PL_Subsystem_new_ram_pos_fb, OUTPUT_2_SIZE);
414     InitData(&PL_Subsystem_new_ddv_sum_out, OUTPUT_3_SIZE);
415     InitData(&PL_Subsystem_new_ddv_position_feedback, OUTPUT_4_SIZE);
416     InitData(&PL_Subsystem_new_currrentfeedback_amp, OUTPUT_5_SIZE);
417     InitData(&PL_Subsystem_new_to_pwm, OUTPUT_6_SIZE);
418 
419     totalErrCount = RunTest(MM2S_BASE, S2MM_BASE,
420                         PL_Subsystem_new_currentfb_indata, INPUT_0_SIZE,
421                         PL_Subsystem_new_ddv_ratio_indata, INPUT_1_SIZE,
422                         PL_Subsystem_new_ddv_sum_indata, INPUT_2_SIZE,
423                         PL_Subsystem_new_flutter_exc_indata, INPUT_3_SIZE,
424                         PL_Subsystem_new_ram_ratio_indata, INPUT_4_SIZE,
425                         PL_Subsystem_new_ram_sum_indata, INPUT_5_SIZE,
426                         PL_Subsystem_new_setpoint_indata, INPUT_6_SIZE,
427                         PL_Subsystem_new_desired_ram_cmd_goldendata,
428                         PL_Subsystem_new_ram_sum_out_goldendata,
429                         PL_Subsystem_new_ram_pos_fb_goldendata,
430                         PL_Subsystem_new_ddv_sum_out_goldendata,
431                         PL_Subsystem_new_ddv_position_feedback_goldendata,
432                         PL_Subsystem_new_currrentfeedback_amp_goldendata,
433                         PL_Subsystem_new_to_pwm_goldendata,
434                         PL_Subsystem_new_desired_ram_cmd, OUTPUT_0_SIZE,
435                         PL_Subsystem_new_ram_sum_out, OUTPUT_1_SIZE,
436                         PL_Subsystem_new_ram_pos_fb, OUTPUT_2_SIZE,
437                         PL_Subsystem_new_ddv_sum_out, OUTPUT_3_SIZE,
438                         PL_Subsystem_new_ddv_position_feedback, OUTPUT_4_SIZE
    ,
439                         PL_Subsystem_new_currrentfeedback_amp, OUTPUT_5_SIZE,
440                         PL_Subsystem_new_to_pwm, OUTPUT_6_SIZE);
441 
442     if (totalErrCount == 0) {
443 
444         printf("\n****** TEST PASSED\n");
445     } else {
446         printf("ERROR: TEST FAILED! Error count: %d \n",
447                 totalErrCount);
448     }
449 
450     printf("\n****** VMC_TEST_DONE\n");
451     cleanup_platform();
452 
453     return totalErrCount;
```

```
}
```

This code runs on the processing system. The `main` function calls the `RunTest` function, which initializes and starts the DMAs and then initializes and starts the AIE graph. If a mismatch in the hardware output is detected, an error message is printed on the serial terminal; if not, a TEST PASSED message is displayed.

B.2 ACE AIE_subsystem.h

```
#ifndef __XMC_AIE_SUBSYSTEM_H__
#define __XMC_AIE_SUBSYSTEM_H__

#include <adf.h>
#include "kernels/IIR_30.hpp"
#include "kernels/IIR_2.hpp"
#include "kernels/IIR_1.hpp"
#include "kernels/IIR_input_block_1.hpp"
#include "kernels/IIR_input_block_2.hpp"
#include "kernels/IIR_input_block_3.hpp"
#include "kernels/IIR_CONTROLLER_1.hpp"
#include "kernels/IIR_CONTROLLER_2.hpp"
#include "kernels/IIR_CONTROLLER_3.hpp"
#include "kernels/IIR_25.hpp"
#include "kernels/IIR_20.hpp"
#include "kernels/IIR_19.hpp"
#include "kernels/IIR_18.hpp"

class AIE_Subsystem_base : public adf::graph {
public:
   adf::kernel IIR_30_0;
   adf::kernel IIR_2_0;
   adf::kernel IIR_1_0;
   adf::kernel IIR_input_block_1_0;
   adf::kernel IIR_input_block_2_0;
   adf::kernel IIR_input_block_3_0;
   adf::kernel IIR_controller_1;
   adf::kernel IIR_controller_2;
   adf::kernel IIR_controller_3;
   adf::kernel IIR_25_0;
   adf::kernel IIR_20_0;
   adf::kernel IIR_19_0;
   adf::kernel IIR_18_0;

public:
   adf::input_port TF1_in, TF2_in, TF19_in, TF20_in, input_block_1_in,
   controller_1_in, TF25_in, input_block_2_in, controller_2_in, TF18_in,
   TF30_in, input_block_3_in, controller_3_in;
   adf::output_port TF1_out, TF2_out, TF19_out, TF_20_out,
   input_block_1_out, controller_1_out, TF_25_out, input_block_2_out,
   controller_2_out, TF18_out, TF30_out, input_block_3_out,
   controller_3_out;

   AIE_Subsystem_base() {
      // create kernel IIR_30_0
      IIR_30_0 = adf::kernel::create(IIR_30);
      adf::source(IIR_30_0) = "kernels/IIR_30.cpp";
```

```
43
44      // create kernel IIR_2_0
45      IIR_2_0 = adf::kernel::create(IIR_2);
46      adf::source(IIR_2_0) = "kernels/IIR_2.cpp";
47
48      // create kernel IIR_1_0
49      IIR_1_0 = adf::kernel::create(IIR_1);
50      adf::source(IIR_1_0) = "kernels/IIR_1.cpp";
51
52      // create kernel IIR_input_block_1_0
53      IIR_input_block_1_0 = adf::kernel::create(IIR_input_block_1);
54      adf::source(IIR_input_block_1_0) = "kernels/IIR_input_block_1.cpp";
55
56      // create kernel IIR_input_block_2_0
57      IIR_input_block_2_0 = adf::kernel::create(IIR_input_block_2);
58      adf::source(IIR_input_block_2_0) = "kernels/IIR_input_block_2.cpp";
59
60      // create kernel IIR_input_block_3_0
61      IIR_input_block_3_0 = adf::kernel::create(IIR_input_block_3);
62      adf::source(IIR_input_block_3_0) = "kernels/IIR_input_block_3.cpp";
63
64      // create kernel IIR_controller_1
65      IIR_controller_1 = adf::kernel::create(IIR_CONTROLLER_1);
66      adf::source(IIR_controller_1) = "kernels/IIR_CONTROLLER_1.cpp";
67
68      // create kernel IIR_controller_2
69      IIR_controller_2 = adf::kernel::create(IIR_CONTROLLER_2);
70      adf::source(IIR_controller_2) = "kernels/IIR_CONTROLLER_2.cpp";
71
72      // create kernel IIR_controller_3
73      IIR_controller_3 = adf::kernel::create(IIR_CONTROLLER_3);
74      adf::source(IIR_controller_3) = "kernels/IIR_CONTROLLER_3.cpp";
75
76      // create kernel IIR_25_0
77      IIR_25_0 = adf::kernel::create(IIR_25);
78      adf::source(IIR_25_0) = "kernels/IIR_25.cpp";
79
80      // create kernel IIR_20_0
81      IIR_20_0 = adf::kernel::create(IIR_20);
82      adf::source(IIR_20_0) = "kernels/IIR_20.cpp";
83
84      // create kernel IIR_19_0
85      IIR_19_0 = adf::kernel::create(IIR_19);
86      adf::source(IIR_19_0) = "kernels/IIR_19.cpp";
87
88      // create kernel IIR_18_0
89      IIR_18_0 = adf::kernel::create(IIR_18);
90      adf::source(IIR_18_0) = "kernels/IIR_18.cpp";
91
92      // create kernel constraints IIR_30_0
93      adf::runtime<ratio>(IIR_30_0) = 0.9;
94
95      // create kernel constraints IIR_2_0
96      adf::runtime<ratio>(IIR_2_0) = 0.9;
97
98      // create kernel constraints IIR_1_0
99      adf::runtime<ratio>(IIR_1_0) = 0.9;
100
```

```
101         // create kernel constraints IIR_input_block_1_0
102         adf::runtime<ratio>(IIR_input_block_1_0) = 0.9;
103 
104         // create kernel constraints IIR_input_block_2_0
105         adf::runtime<ratio>(IIR_input_block_2_0) = 0.9;
106 
107         // create kernel constraints IIR_input_block_3_0
108         adf::runtime<ratio>(IIR_input_block_3_0) = 0.9;
109 
110         // create kernel constraints IIR_controller_1
111         adf::runtime<ratio>(IIR_controller_1) = 0.9;
112 
113         // create kernel constraints IIR_controller_2
114         adf::runtime<ratio>(IIR_controller_2) = 0.9;
115 
116         // create kernel constraints IIR_controller_3
117         adf::runtime<ratio>(IIR_controller_3) = 0.9;
118 
119         // create kernel constraints IIR_25_0
120         adf::runtime<ratio>(IIR_25_0) = 0.9;
121 
122         // create kernel constraints IIR_20_0
123         adf::runtime<ratio>(IIR_20_0) = 0.9;
124 
125         // create kernel constraints IIR_19_0
126         adf::runtime<ratio>(IIR_19_0) = 0.9;
127 
128         // create kernel constraints IIR_18_0
129         adf::runtime<ratio>(IIR_18_0) = 0.9;
130 
131         // create nets to specify connections
132         adf::connect< adf::stream > net0 (TF1_in, IIR_1_0.in[0]);
133         adf::connect< adf::stream > net1 (TF2_in, IIR_2_0.in[0]);
134         adf::connect< adf::stream > net2 (TF19_in, IIR_19_0.in[0]);
135         adf::connect< adf::stream > net3 (TF20_in, IIR_20_0.in[0]);
136         adf::connect< adf::stream > net4 (input_block_1_in,
    IIR_input_block_1_0.in[0]);
137         adf::connect< adf::stream > net5 (controller_1_in, IIR_controller_1.
    in[0]);
138         adf::connect< adf::stream > net6 (TF25_in, IIR_25_0.in[0]);
139         adf::connect< adf::stream > net7 (input_block_2_in,
    IIR_input_block_2_0.in[0]);
140         adf::connect< adf::stream > net8 (controller_2_in, IIR_controller_2.
    in[0]);
141         adf::connect< adf::stream > net9 (TF18_in, IIR_18_0.in[0]);
142         adf::connect< adf::stream > net10 (TF30_in, IIR_30_0.in[0]);
143         adf::connect< adf::stream > net11 (input_block_3_in,
    IIR_input_block_3_0.in[0]);
144         adf::connect< adf::stream > net12 (controller_3_in, IIR_controller_3.
    in[0]);
145         adf::connect< adf::stream > net13 (IIR_30_0.out[0], TF30_out);
146         adf::connect< adf::stream > net14 (IIR_2_0.out[0], TF2_out);
147         adf::connect< adf::stream > net15 (IIR_1_0.out[0], TF1_out);
148         adf::connect< adf::stream > net16 (IIR_input_block_1_0.out[0],
    input_block_1_out);
149         adf::connect< adf::stream > net17 (IIR_input_block_2_0.out[0],
    input_block_2_out);
```

```
150         adf::connect< adf::stream > net18 (IIR_input_block_3_0.out[0],
    input_block_3_out);
151         adf::connect< adf::stream > net19 (IIR_controller_1.out[0],
    controller_1_out);
152         adf::connect< adf::stream > net20 (IIR_controller_2.out[0],
    controller_2_out);
153         adf::connect< adf::stream > net21 (IIR_controller_3.out[0],
    controller_3_out);
154         adf::connect< adf::stream > net22 (IIR_25_0.out[0], TF_25_out);
155         adf::connect< adf::stream > net23 (IIR_20_0.out[0], TF_20_out);
156         adf::connect< adf::stream > net24 (IIR_19_0.out[0], TF19_out);
157         adf::connect< adf::stream > net25 (IIR_18_0.out[0], TF18_out);
158     }
159 };
160
161 class AIE_Subsystem : public adf::graph {
162 public:
163     AIE_Subsystem_base mygraph;
164
165 public:
166     adf::input_plio TF1_in, TF2_in, TF19_in, TF20_in, input_block_1_in,
    controller_1_in, TF25_in, input_block_2_in, controller_2_in, TF18_in,
    TF30_in, input_block_3_in, controller_3_in;
167     adf::output_plio TF1_out, TF2_out, TF19_out, TF_20_out,
    input_block_1_out, controller_1_out, TF_25_out, input_block_2_out,
    controller_2_out, TF18_out, TF30_out, input_block_3_out,
    controller_3_out;
168
169     AIE_Subsystem() {
170         TF1_in = adf::input_plio::create("TF1_in",
171                 adf::plio_64_bits,
172                 "./data/input/TF1_in.txt");
173
174         TF2_in = adf::input_plio::create("TF2_in",
175                 adf::plio_64_bits,
176                 "./data/input/TF2_in.txt");
177
178         TF19_in = adf::input_plio::create("TF19_in",
179                 adf::plio_64_bits,
180                 "./data/input/TF19_in.txt");
181
182         TF20_in = adf::input_plio::create("TF20_in",
183                 adf::plio_64_bits,
184                 "./data/input/TF20_in.txt");
185
186         input_block_1_in = adf::input_plio::create("input_block_1_in",
187                 adf::plio_64_bits,
188                 "./data/input/input_block_1_in.txt");
189
190         controller_1_in = adf::input_plio::create("controller_1_in",
191                 adf::plio_64_bits,
192                 "./data/input/controller_1_in.txt");
193
194         TF25_in = adf::input_plio::create("TF25_in",
195                 adf::plio_64_bits,
196                 "./data/input/TF25_in.txt");
197
198         input_block_2_in = adf::input_plio::create("input_block_2_in",
```

```
199             adf::plio_64_bits,
200             "./data/input/input_block_2_in.txt");
201
202     controller_2_in = adf::input_plio::create("controller_2_in",
203             adf::plio_64_bits,
204             "./data/input/controller_2_in.txt");
205
206     TF18_in = adf::input_plio::create("TF18_in",
207             adf::plio_64_bits,
208             "./data/input/TF18_in.txt");
209
210     TF30_in = adf::input_plio::create("TF30_in",
211             adf::plio_64_bits,
212             "./data/input/TF30_in.txt");
213
214     input_block_3_in = adf::input_plio::create("input_block_3_in",
215             adf::plio_64_bits,
216             "./data/input/input_block_3_in.txt");
217
218     controller_3_in = adf::input_plio::create("controller_3_in",
219             adf::plio_64_bits,
220             "./data/input/controller_3_in.txt");
221
222     TF1_out = adf::output_plio::create("TF1_out",
223             adf::plio_64_bits,
224             "TF1_out.txt");
225
226     TF2_out = adf::output_plio::create("TF2_out",
227             adf::plio_64_bits,
228             "TF2_out.txt");
229
230     TF19_out = adf::output_plio::create("TF19_out",
231             adf::plio_64_bits,
232             "TF19_out.txt");
233
234     TF_20_out = adf::output_plio::create("TF_20_out",
235             adf::plio_64_bits,
236             "TF_20_out.txt");
237
238     input_block_1_out = adf::output_plio::create("input_block_1_out",
239             adf::plio_64_bits,
240             "input_block_1_out.txt");
241
242     controller_1_out = adf::output_plio::create("controller_1_out",
243             adf::plio_64_bits,
244             "controller_1_out.txt");
245
246     TF_25_out = adf::output_plio::create("TF_25_out",
247             adf::plio_64_bits,
248             "TF_25_out.txt");
249
250     input_block_2_out = adf::output_plio::create("input_block_2_out",
251             adf::plio_64_bits,
252             "input_block_2_out.txt");
253
254     controller_2_out = adf::output_plio::create("controller_2_out",
255             adf::plio_64_bits,
256             "controller_2_out.txt");
```

```
257
258         TF18_out = adf::output_plio::create("TF18_out",
259                 adf::plio_64_bits,
260                 "TF18_out.txt");
261
262         TF30_out = adf::output_plio::create("TF30_out",
263                 adf::plio_64_bits,
264                 "TF30_out.txt");
265
266         input_block_3_out = adf::output_plio::create("input_block_3_out",
267                 adf::plio_64_bits,
268                 "input_block_3_out.txt");
269
270         controller_3_out = adf::output_plio::create("controller_3_out",
271                 adf::plio_64_bits,
272                 "controller_3_out.txt");
273
274         adf::connect<> (TF1_in.out[0], mygraph.TF1_in);
275         adf::connect<> (TF2_in.out[0], mygraph.TF2_in);
276         adf::connect<> (TF19_in.out[0], mygraph.TF19_in);
277         adf::connect<> (TF20_in.out[0], mygraph.TF20_in);
278         adf::connect<> (input_block_1_in.out[0], mygraph.input_block_1_in);
279         adf::connect<> (controller_1_in.out[0], mygraph.controller_1_in);
280         adf::connect<> (TF25_in.out[0], mygraph.TF25_in);
281         adf::connect<> (input_block_2_in.out[0], mygraph.input_block_2_in);
282         adf::connect<> (controller_2_in.out[0], mygraph.controller_2_in);
283         adf::connect<> (TF18_in.out[0], mygraph.TF18_in);
284         adf::connect<> (TF30_in.out[0], mygraph.TF30_in);
285         adf::connect<> (input_block_3_in.out[0], mygraph.input_block_3_in);
286         adf::connect<> (controller_3_in.out[0], mygraph.controller_3_in);
287         adf::connect<> (mygraph.TF1_out, TF1_out.in[0]);
288         adf::connect<> (mygraph.TF2_out, TF2_out.in[0]);
289         adf::connect<> (mygraph.TF19_out, TF19_out.in[0]);
290         adf::connect<> (mygraph.TF_20_out, TF_20_out.in[0]);
291         adf::connect<> (mygraph.input_block_1_out, input_block_1_out.in[0]);
292         adf::connect<> (mygraph.controller_1_out, controller_1_out.in[0]);
293         adf::connect<> (mygraph.TF_25_out, TF_25_out.in[0]);
294         adf::connect<> (mygraph.input_block_2_out, input_block_2_out.in[0]);
295         adf::connect<> (mygraph.controller_2_out, controller_2_out.in[0]);
296         adf::connect<> (mygraph.TF18_out, TF18_out.in[0]);
297         adf::connect<> (mygraph.TF30_out, TF30_out.in[0]);
298         adf::connect<> (mygraph.input_block_3_out, input_block_3_out.in[0]);
299         adf::connect<> (mygraph.controller_3_out, controller_3_out.in[0]);
300     }
301 };
302
303 #endif // __XMC_AIE_SUBSYSTEM_H__
```

This source code contains the definition and application of constraints of the AIE graph. All the inputs and outputs of the graph are also connected to the specific `.txt` file containing input data and golden output, so this can be used to simulate the AIE array.

B.3 ACE AIE_subsystem.cpp

```
1 #include "AIE_Subsystem.h"
```

```
2
3  // instantiate ADF graph
4  AIE_Subsystem mygraph;
5
6  // initialize and run the dataflow graph
7  #if defined(__AIESIM__) || defined(__X86SIM__)
8  int main(void) {
9     mygraph.init();
10    mygraph.run();
11    mygraph.end();
12    return 0;
13 }
14 #endif
```

This code initializes, runs and ends the AIE graph execution.

B.4 ACE s2mm.cpp

```
1  #include <hls_stream.h>
2  #include <ap_int.h>
3  #include <ap_axi_sdata.h>
4  #include <stdint.h>
5
6
7
8     template <int BITWIDTH>
9     void Push(hls::stream<ap_axiu<BITWIDTH, 0, 0, 0> >& s, uint32_t* target,
       int size){
10     constexpr int dwidth = ((BITWIDTH/32) + (((BITWIDTH%32)==0) ? 0 : 1))
      *32;
11     for(int i = 0; i < size; i+=dwidth/32) {
12 #pragma HLS PIPELINE II=1
13       ap_axiu<BITWIDTH, 0, 0, 0> v = s.read();
14
15       target[i]   = uint32_t(v.data);
16       if constexpr (BITWIDTH == 64)
17         target[i+1] = uint32_t(v.data >> 32);
18       else if constexpr (BITWIDTH == 128) {
19         target[i+1] = uint32_t(v.data >> 32);
20         target[i+2] = uint32_t(v.data >> 64);
21         target[i+3] = uint32_t(v.data >> 96);
22       } else if constexpr (BITWIDTH > 32) {
23         for (int shift = 32,j=1; shift < BITWIDTH; shift+=32,j++) {
24           target[i+j] = uint32_t(v.data >> shift);
25         }
26       }
27
28     }
29   }
30
31   void PushAll(
32     hls::stream<ap_axiu<32, 0, 0, 0> >& in0, uint32_t* out0, uint32_t size0
      ,
33     hls::stream<ap_axiu<32, 0, 0, 0> >& in1, uint32_t* out1, uint32_t size1
      ,
```

```
34      hls::stream<ap_axiu<32, 0, 0, 0> >& in2, uint32_t* out2, uint32_t size2
        ,
35      hls::stream<ap_axiu<32, 0, 0, 0> >& in3, uint32_t* out3, uint32_t size3
        ,
36      hls::stream<ap_axiu<32, 0, 0, 0> >& in4, uint32_t* out4, uint32_t size4
        ,
37      hls::stream<ap_axiu<32, 0, 0, 0> >& in5, uint32_t* out5, uint32_t size5
        ,
38      hls::stream<ap_axiu<32, 0, 0, 0> >& in6, uint32_t* out6, uint32_t size6
        ) {
39  #pragma HLS DATAFLOW
40      Push<32>(in0, out0, size0);
41      Push<32>(in1, out1, size1);
42      Push<32>(in2, out2, size2);
43      Push<32>(in3, out3, size3);
44      Push<32>(in4, out4, size4);
45      Push<32>(in5, out5, size5);
46      Push<32>(in6, out6, size6);
47    }
48
49   extern "C" {
50    void s2mm(
51      hls::stream<ap_axiu<32, 0, 0, 0> >& in0,
52      uint32_t* out0, uint32_t size0,
53      hls::stream<ap_axiu<32, 0, 0, 0> >& in1,
54      uint32_t* out1, uint32_t size1,
55      hls::stream<ap_axiu<32, 0, 0, 0> >& in2,
56      uint32_t* out2, uint32_t size2,
57      hls::stream<ap_axiu<32, 0, 0, 0> >& in3,
58      uint32_t* out3, uint32_t size3,
59      hls::stream<ap_axiu<32, 0, 0, 0> >& in4,
60      uint32_t* out4, uint32_t size4,
61      hls::stream<ap_axiu<32, 0, 0, 0> >& in5,
62      uint32_t* out5, uint32_t size5,
63      hls::stream<ap_axiu<32, 0, 0, 0> >& in6,
64      uint32_t* out6, uint32_t size6) {
65  #pragma HLS INTERFACE ap_ctrl_hs port=return bundle=control
66  #pragma HLS INTERFACE s_axilite port=return bundle=control
67  #pragma HLS INTERFACE axis port=in0
68  #pragma HLS INTERFACE m_axi port=out0 bundle=gmem0 max_widen_bitwidth=4
69  #pragma HLS INTERFACE s_axilite port=out0 bundle=control
70  #pragma HLS INTERFACE s_axilite port=size0 bundle=control
71  #pragma HLS INTERFACE axis port=in1
72  #pragma HLS INTERFACE m_axi port=out1 bundle=gmem1 max_widen_bitwidth=4
73  #pragma HLS INTERFACE s_axilite port=out1 bundle=control
74  #pragma HLS INTERFACE s_axilite port=size1 bundle=control
75  #pragma HLS INTERFACE axis port=in2
76  #pragma HLS INTERFACE m_axi port=out2 bundle=gmem2 max_widen_bitwidth=4
77  #pragma HLS INTERFACE s_axilite port=out2 bundle=control
78  #pragma HLS INTERFACE s_axilite port=size2 bundle=control
79  #pragma HLS INTERFACE axis port=in3
80  #pragma HLS INTERFACE m_axi port=out3 bundle=gmem3 max_widen_bitwidth=4
81  #pragma HLS INTERFACE s_axilite port=out3 bundle=control
82  #pragma HLS INTERFACE s_axilite port=size3 bundle=control
83  #pragma HLS INTERFACE axis port=in4
84  #pragma HLS INTERFACE m_axi port=out4 bundle=gmem4 max_widen_bitwidth=4
85  #pragma HLS INTERFACE s_axilite port=out4 bundle=control
86  #pragma HLS INTERFACE s_axilite port=size4 bundle=control
```

```
87 #pragma HLS INTERFACE axis port=in5
88 #pragma HLS INTERFACE m_axi port=out5 bundle=gmem5 max_widen_bitwidth=4
89 #pragma HLS INTERFACE s_axilite port=out5 bundle=control
90 #pragma HLS INTERFACE s_axilite port=size5 bundle=control
91 #pragma HLS INTERFACE axis port=in6
92 #pragma HLS INTERFACE m_axi port=out6 bundle=gmem6 max_widen_bitwidth=4
93 #pragma HLS INTERFACE s_axilite port=out6 bundle=control
94 #pragma HLS INTERFACE s_axilite port=size6 bundle=control
95
96     PushAll(
97       in0, out0, size0,
98       in1, out1, size1,
99       in2, out2, size2,
100      in3, out3, size3,
101      in4, out4, size4,
102      in5, out5, size5,
103      in6, out6, size6);
104   }
105
106 }
```

This code contains an HLS description of a stream-to-memory mapped DMA. The s2mm kernel starts when triggered via AXI-Lite (ap_start). It reads data from AXI-Stream inputs (in0 to in6) and writes to memory via AXI-MM. Each stream is processed in parallel using HLS DATAFLOW, storing sizeX words to outX. DMA-like behavior is implemented in hardware; activation depends on the host writing control registers.

B.5 ACE mm2s.cpp

```
1  #include <hls_stream.h>
2  #include <ap_int.h>
3  #include <ap_axi_sdata.h>
4  #include <stdint.h>
5
6
7
8  template <int BITWIDTH>
9          void Push(uint32_t* data, hls::stream<ap_axiu<BITWIDTH, 0, 0, 0> >&
       target, uint32_t size, int pulse){
10     constexpr int dwidth = ((BITWIDTH/32) + (((BITWIDTH%32)==0) ? 0 : 1))
       *32;
11     for(int i = 0; i < size * pulse; i+=dwidth/32) {
12 #pragma HLS PIPELINE II=1
13         int index = i % size;
14         ap_axiu<BITWIDTH, 0, 0, 0> v;
15         v.last = 0;
16         v.keep = 0xFFFF;
17         v.data = data[index];
18         if constexpr (BITWIDTH == 64)
19             v.data += (uint64_t(data[index+1])<<32);
20         else if constexpr (BITWIDTH == 128)
21             v.data += (ap_int<128>(data[index+1])<<32)+
22                       (ap_int<128>(data[index+2])<<64)+
23                       (ap_int<128>(data[index+3])<<96);
24         else if constexpr (BITWIDTH > 32) {
25             for (int shift = 32,j=1; shift < BITWIDTH; shift+=32,j++) {
```

```
26                  v.data += (ap_int<BITWIDTH>(data[index+j])<<shift);
27              }
28          }
29          target.write(v);
30      }
31  }
32
33
34    void PushAll(
35      uint32_t* in0, hls::stream<ap_axiu<32, 0, 0, 0> >& out0, uint32_t size0
      ,
36      uint32_t* in1, hls::stream<ap_axiu<32, 0, 0, 0> >& out1, uint32_t size1
      ,
37      uint32_t* in2, hls::stream<ap_axiu<32, 0, 0, 0> >& out2, uint32_t size2
      ,
38      uint32_t* in3, hls::stream<ap_axiu<32, 0, 0, 0> >& out3, uint32_t size3
      ,
39      uint32_t* in4, hls::stream<ap_axiu<32, 0, 0, 0> >& out4, uint32_t size4
      ,
40      uint32_t* in5, hls::stream<ap_axiu<32, 0, 0, 0> >& out5, uint32_t size5
      ,
41      uint32_t* in6, hls::stream<ap_axiu<32, 0, 0, 0> >& out6, uint32_t size6
      ,
42      int pulse) {
43  #pragma HLS DATAFLOW
44      Push<32>(in0, out0, size0, pulse);
45      Push<32>(in1, out1, size1, pulse);
46      Push<32>(in2, out2, size2, pulse);
47      Push<32>(in3, out3, size3, pulse);
48      Push<32>(in4, out4, size4, pulse);
49      Push<32>(in5, out5, size5, pulse);
50      Push<32>(in6, out6, size6, pulse);
51    }
52
53  extern "C" {
54    void mm2s(
55      uint32_t* in0,
56      hls::stream<ap_axiu<32, 0, 0, 0> >& out0, uint32_t size0,
57      uint32_t* in1,
58      hls::stream<ap_axiu<32, 0, 0, 0> >& out1, uint32_t size1,
59      uint32_t* in2,
60      hls::stream<ap_axiu<32, 0, 0, 0> >& out2, uint32_t size2,
61      uint32_t* in3,
62      hls::stream<ap_axiu<32, 0, 0, 0> >& out3, uint32_t size3,
63      uint32_t* in4,
64      hls::stream<ap_axiu<32, 0, 0, 0> >& out4, uint32_t size4,
65      uint32_t* in5,
66      hls::stream<ap_axiu<32, 0, 0, 0> >& out5, uint32_t size5,
67      uint32_t* in6,
68      hls::stream<ap_axiu<32, 0, 0, 0> >& out6, uint32_t size6,
69      int pulse) {
70  #pragma HLS INTERFACE ap_ctrl_hs port=return bundle=control
71  #pragma HLS INTERFACE s_axilite port=return bundle=control
72  #pragma HLS INTERFACE m_axi port=in0 bundle=gmem0 max_widen_bitwidth=4
73  #pragma HLS INTERFACE s_axilite port=in0 bundle=control
74  #pragma HLS INTERFACE axis port=out0
75  #pragma HLS INTERFACE s_axilite port=size0 bundle=control
76  #pragma HLS INTERFACE m_axi port=in1 bundle=gmem1 max_widen_bitwidth=4
```

```
77 #pragma HLS INTERFACE s_axilite port=in1 bundle=control
78 #pragma HLS INTERFACE axis port=out1
79 #pragma HLS INTERFACE s_axilite port=size1 bundle=control
80 #pragma HLS INTERFACE m_axi port=in2 bundle=gmem2 max_widen_bitwidth=4
81 #pragma HLS INTERFACE s_axilite port=in2 bundle=control
82 #pragma HLS INTERFACE axis port=out2
83 #pragma HLS INTERFACE s_axilite port=size2 bundle=control
84 #pragma HLS INTERFACE m_axi port=in3 bundle=gmem3 max_widen_bitwidth=4
85 #pragma HLS INTERFACE s_axilite port=in3 bundle=control
86 #pragma HLS INTERFACE axis port=out3
87 #pragma HLS INTERFACE s_axilite port=size3 bundle=control
88 #pragma HLS INTERFACE m_axi port=in4 bundle=gmem4 max_widen_bitwidth=4
89 #pragma HLS INTERFACE s_axilite port=in4 bundle=control
90 #pragma HLS INTERFACE axis port=out4
91 #pragma HLS INTERFACE s_axilite port=size4 bundle=control
92 #pragma HLS INTERFACE m_axi port=in5 bundle=gmem5 max_widen_bitwidth=4
93 #pragma HLS INTERFACE s_axilite port=in5 bundle=control
94 #pragma HLS INTERFACE axis port=out5
95 #pragma HLS INTERFACE s_axilite port=size5 bundle=control
96 #pragma HLS INTERFACE m_axi port=in6 bundle=gmem6 max_widen_bitwidth=4
97 #pragma HLS INTERFACE s_axilite port=in6 bundle=control
98 #pragma HLS INTERFACE axis port=out6
99 #pragma HLS INTERFACE s_axilite port=size6 bundle=control
100 #pragma HLS INTERFACE s_axilite port=pulse bundle=control
101
102     PushAll(
103       in0, out0, size0,
104       in1, out1, size1,
105       in2, out2, size2,
106       in3, out3, size3,
107       in4, out4, size4,
108       in5, out5, size5,
109       in6, out6, size6,
110       pulse);
111   }
112
113 }
```

The mm2s kernel reads data from memory (inX) and writes it to AXI streams (outX) across 7 channels, repeating the transfer pulse times. Each Push sends data as ap_axiu packets on the HLS stream. If target.write() blocks, it means the receiver (e.g., DMA) is not reading fast enough: the FIFO is full and the kernel waits. This indicates that the next sample cannot be sent until the previous one is consumed.

Appendix C

Kernel codes

The kernel codes reported in this Section are the custom C++ functions run on AIE (or AIE-ML) used in the ACE development.

C.1 Single IIR filter order 2 - buffer

```
#include <adf.h>
#include <aie_api/aie.hpp>
#include <aie_api/aie_adf.hpp>
#include <aie_api/utils.hpp>

/*
x = [x, 0, 0, 0]
p = [x(n-1). x(n-2), y(n-1), y(n-2)]
c = [b2, b3, a2, a3, b1 g, 0, 0]
y = [y, 0, 0, 0]
p_out = p updated for next iteration
*/

void IIR_2_custom
(
    adf::input_buffer<float,adf::extents<4>>& __restrict x, // input sample
    adf::input_buffer<float,adf::extents<8>>& __restrict c, // coefficients
    adf::input_buffer<float,adf::extents<4>>& __restrict p, // partial
   results
    adf::output_buffer<float,adf::extents<4>>& __restrict p_out, // partial
    results, output
    adf::output_buffer<float,adf::extents<4>>& __restrict y // output
) {
    // iterators for in/out buffers
    auto x_int_I=aie::begin_vector<4>(x);
    auto v_coeff_I=aie::begin_vector<8>(c);
    auto p_int_I=aie::begin_vector<4>(p);
    auto p_out_I=aie::begin_vector<4>(p_out);
    auto y_I=aie::begin_vector<4>(y);

    // iterator -> vector
    auto x_int = *x_int_I++;
    auto p_int = *p_int_I++;
```

```
32      auto v_coeff = *v_coeff_I++;
33      x_int[0] = x_int[0]*v_coeff[5]; // *g multiplication just on input
        sample
34
35      // v_sample should have [x(t-1), x(t-2), y(t-1), y(t-2), x, 0, 0, 0]
36      aie::vector<float, 8> v_sample = aie::concat(p_int, x_int);
37
38      // operations for output
39      auto v_mul = aie::mul(v_sample,v_coeff);// multiplication
40      float y_int = aie::reduce_add(v_mul.to_vector<float>(0));// sum of
        v_mul
41      aie::vector<float, 4> v_result = aie::zeros<float,4>();
42      v_result.push(y_int);// out must be a vector
43      *y_I++=v_result;
44
45      // partial results update
46      aie::vector<float, 4> p_out_v;
47      p_out_v[1] = p_int[0];
48      p_out_v[0] = x_int[0];
49      p_out_v[3] = p_int[2];
50      p_out_v[2] = v_result[0];
51      *p_out_I++=p_out_v;
52  }
```

This code describes with AIE APIs an IIR filter of order 2, as explained in Section 4.1.1. This kernel, with buffer inputs, would also need a manual handshake of the AXI protocol. For this reason, in the next versions, coefficients and partial products are stored in AIE memory, while the input type stream will allow automatic AXI4 handshake management.

C.2 Single IIR filter order 2 - stream (not optimal)

```
1  #include <adf.h>
2  #include <aie_api/aie.hpp>
3  #include <aie_api/aie_adf.hpp>
4  #include <aie_api/utils.hpp>
5
6  /*
7  x = [x]
8  p = [x(n-1). x(n-2), y(n-1), y(n-2)]
9  c = [b2, b3, -a2, -a3, b1 g, 0, 0]
10 y = [y]
11 */
12
13 void IIR_2_custom
14 (
15      input_stream<float>*  x, // input sample
16      output_stream<float>*  y // output
17 ) {
18      static aie::vector<float, 4> p = aie::zeros<float, 4>(); // clear
        partial results
19      const aie::vector<float, 8> coeff(0.6868f, 0.f, 0.9973f, 0.f, 0.6868f,
        1.f, 0.f, 0.f);
20
21      // input reading
```

```
    float x_int = readincr(x);


    // to vector
    x_int = x_int*coeff[5]; // *g multiplication just on input sample
    aie::vector<float, 4> x_int_v = aie::zeros<float,4>();
    x_int_v.push(x_int);// out must be a vector

    // v_sample should have [x(t-1), x(t-2), y(t-1), y(t-2), x, 0, 0, 0]
    aie::vector<float, 8> v_sample = aie::concat(p, x_int_v);

    // operations for output
    auto v_mul = aie::mul(v_sample,coeff);// multiplication
    float y_int = aie::reduce_add(v_mul.to_vector<float>(0));// sum of
   v_mul
    writeincr(y, y_int);

    // partial results update
    p[1] = p[0];
    p[0] = x_int;
    p[3] = p[2];
    p[2] = y_int;
}
```

This is a streaming version of the single IIR filter of order 2, but calling scalar multiplication and sum functions as '*' and '+' would result in the call of an appropriate function that would waste too many clock cycles (AIE does not contain a scalar FP unit). This operation is transformed into a vectorial one in the next versions of the code.

C.3 Single IIR filter order 2 - stream

```
#include <adf.h>
#include <aie_api/aie.hpp>
#include <aie_api/aie_adf.hpp>
#include <aie_api/utils.hpp>

/*
x = [x]
p = [x(n-1). x(n-2), y(n-1), y(n-2)]
c = [b2, b3, -a2, -a3, b1 g, 0, 0]
y = [y]
*/

void IIR_2_custom
(
    input_stream<float>*  x, // input sample
    output_stream<float>*  y // output
) {
    static aie::vector<float, 4> p = aie::zeros<float, 4>(); // clear
   partial results
    const aie::vector<float, 8> coeff(0.6868f, 0.f, 0.9973f, 0.f, 0.6868f,
   1.f, 0.f, 0.f);

    // input reading
    float x_int = readincr(x);

```

```
    //  -> vector
    aie::vector<float, 4> x_int_v(x_int, 0.0f, 0.0f, 0.0f);
    x_int_v = aie::mul(x_int_v,coeff[5]).to_vector<float>(0);

    // v_sample should have [x(t-1), x(t-2), y(t-1), y(t-2), x, 0, 0, 0]
    aie::vector<float, 8> v_sample = aie::concat(p, x_int_v);

    // operations for output
    auto v_mul = aie::mul(v_sample,coeff);// multiplication
    float y_int = aie::reduce_add(v_mul.to_vector<float>(0));// sum of
   v_mul
    writeincr(y, y_int);

    // partial results update
    p[1] = p[0];
    p[0] = x_int;
    p[3] = p[2];
    p[2] = y_int;
}
```

64 bit version

```
#include <adf.h>
#include <aie_api/aie.hpp>
#include <aie_api/aie_adf.hpp>
#include <aie_api/utils.hpp>

/*
x = [x]
p = [x(n-1). x(n-2), y(n-1), y(n-2)]
c = [b2, b3, -a2, -a3, b1 g, 0, 0]
y = [y]
*/

void IIR_1
(
    input_stream<uint64>*  x, // input sample
    output_stream<uint64>*  y // output
) {
    static aie::vector<float, 4> p = aie::zeros<float, 4>(); // clear
   partial results
    const aie::vector<float, 8> coeff(0.0f, 0.f, 0.0f, 0.f, 1.f, 1.f, 0.f,
   0.f);


    // input reading
    uint64 x_real = readincr(x);
    float* floats = (float*)&x_real;
    float x_int = floats[0];

    //  -> vector
    aie::vector<float, 4> x_int_v(x_int, 0.0f, 0.0f, 0.0f);
    x_int_v = aie::mul(x_int_v,coeff[5]).to_vector<float>(0);

    // v_sample should have [x(t-1), x(t-2), y(t-1), y(t-2), x, 0, 0, 0]
    aie::vector<float, 8> v_sample = aie::concat(p, x_int_v);

    // operations for output
```

```
    auto v_mul = aie::mul(v_sample,coeff);// multiplication
    float y_int = aie::reduce_add(v_mul.to_vector<float>(0));// sum of
   v_mul
    float raw[2] = {y_int, y_int};
    uint64 y_real = *((uint64_t*)raw);
    writeincr(y, y_real);

    // partial results update
    p[1] = p[0];
    p[0] = x_int_v[0];
    p[3] = p[2];
    p[2] = y_int;
}
```

This is the final version of the IIR filter of order 2. Input and output are 64-bit in order to allow a single sample streaming (min is 64-bit from PL to AIE and vice versa).

C.4 Single IIR filter order 4 - stream (improved)

```
#include <adf.h>
#include <aie_api/aie.hpp>
#include <aie_api/aie_adf.hpp>
#include <aie_api/utils.hpp>

/*
x = [x]
p = [x(n-1), y(n-1), x(n-2), y(n-2), x(n-3), y(n-3), x(n-4), y(n-4)]
c = [b2, -a2, b3, -a3, b4, -a4, b5,-a5, b1, g, 0, 0, 0, 0, 0, 0]
y = [y]
*/

void IIR_4_custom
(
    input_stream<float>*  x, // input sample
    output_stream<float>*  y // output
) {
    static aie::vector<float, 8> p = aie::zeros<float, 8>(); // clear
   partial results
    const aie::vector<float, 16> coeff(0.0567984f, 1.6190948f, 0.0820072f,
   -1.1351098f, 0.0567984f, 0.3802160f, 0.0157913f, -0.0547274f, 0.0157913f
   , 1.0f, 0.f, 0.f, 0.f, 0.f, 0.f);

    // iterators for in/out buffers
    float x_int = readincr(x);


    // input reading
    aie::vector<float, 8> x_int_v(x_int, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f
   , 0.0f);
    x_int_v = aie::mul(x_int_v,coeff[5]).to_vector<float>(0);

    // v_sample should have [x(t-1), x(t-2), y(t-1), y(t-2), x, 0, 0, 0]
    aie::vector<float, 16> v_sample = aie::concat(p, x_int_v);

```

```
32      // operations for output
33      auto v_mul = aie::mul(v_sample,coeff);// multiplication
34      float y_int = aie::reduce_add(v_mul.to_vector<float>(0));// sum of
        v_mul
35      writeincr(y, y_int);
36
37      // partial results update
38      p.push(y_int);
39      p.push(x_int);
40  }
```

This is an order-4 version of the previous code.

C.5 Controller

```
1  #include <adf.h>
2  #include <aie_api/aie.hpp>
3  #include <aie_api/aie_adf.hpp>
4  #include <aie_api/utils.hpp>
5
6  /*
7  x = [x]
8  p = [x(n-1), y(n-1), x(n-2), y(n-2)]
9  c = [b2, -a2, b3, -a3, b1, g, 0, 0]
10 y = [y]
11 */
12
13 // constants' section
14 static const bool LIMITER_ENABLE = true;
15 static const float LIMITER_LOWER_LIMIT = -30.0f;
16 static const float LIMITER_UPPER_LIMIT = 30.0f;
17 static const bool INTEGRATOR_RST = false;
18 static const aie::vector<float, 8> c_A(0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f,
       0.0f, 0.0f);//13
19 static const aie::vector<float, 8> c_B(0.0f, 0.0f, 0.0f, 0.0f, 350.0f, 1.0f
       , 0.0f, 0.0f);//14
20 static const aie::vector<float, 8> c_C(0.0f, 0.0f, 0.0f, 0.0f, 1.701f, 1.0f
       , 0.0f, 0.0f);//15
21 static const aie::vector<float, 8> c_D(0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f,
       0.0f, 0.0f);//16
22 static const aie::vector<float, 8> c_E(0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,
       0.0f, 0.0f);//17
23
24 void IIR_CONTROLLER
25 (
26      input_stream<float>*  x, // input sample
27      output_stream<float>*  y // output
28 ) {
29
30      // partial results
31      static aie::vector<float, 4> p_A = aie::zeros<float, 4>();
32      static aie::vector<float, 4> p_B = aie::zeros<float, 4>();
33      static aie::vector<float, 4> p_C = aie::zeros<float, 4>();
34      static aie::vector<float, 4> p_D = aie::zeros<float, 4>();
35      static aie::vector<float, 4> p_E = aie::zeros<float, 4>();
36
```

```
37      // iterators for in/out buffers
38      float x_int = readincr(x);
39
40      ////////////////////////////////////////////////// FILTER A
41      //  -> vector
42      aie::vector<float, 4> x_int_vA(x_int, 0.0f, 0.0f, 0.0f);
43      x_int_vA = aie::mul(x_int_vA,c_A[5]).to_vector<float>(0);
44
45      // v_sample should be [x(t-1), y(t-1), x(t-2), y(t-2), x, 0, 0, 0]
46      aie::vector<float, 8> v_sample_A = aie::concat(p_A, x_int_vA);
47
48      // operations for output
49      auto v_mul = aie::mul(v_sample_A,c_A);// multiplication
50      float y_A = aie::reduce_add(v_mul.to_vector<float>(0).extract<4>(0));//
        sum of v_mul
51      aie::vector<float, 4> y_A_v(y_A, 0.0f, 0.0f, 0.0f);
52      y_A = aie::add(y_A_v, v_mul.to_vector<float>(0).extract<4>(1))[0];
53
54      // partial results update
55      p_A.push(y_A);
56      p_A.push(x_int_vA[0]);
57
58      ////////////////////////////////////////////// FILTERS B,C,E
59      // gain multiplication
60      aie::vector<float, 4> x_step2(y_A, y_A, y_A, 0.0f);
61      aie::vector<float, 4> g_vector(c_B[5], c_C[5], c_E[5], 0.0f);
62      x_step2 = aie::mul(x_step2, g_vector).to_vector<float>(0);
63
64      // product
65      aie::vector<float, 16> v_sample_BCE = aie::concat(p_B, p_C, p_E,
        x_step2);
66      aie::vector<float, 16> v_coeff_BCE = aie::concat(c_B.extract<4>(0), c_C
        .extract<4>(0), c_E.extract<4>(0), aie::vector<float, 4>(c_B[4], c_C[4],
        c_E[4], 0.0f));
67      auto v_mul_BCE = aie::mul(v_sample_BCE,v_coeff_BCE);// multiplication
68
69      // reduction for the different filters
70      aie::vector<float, 4> y_B(aie::reduce_add(v_mul_BCE.extract<4>(0).
        to_vector<float>(0)), 0.0f, 0.0f, 0.0f);
71      y_B = aie::add(y_B, v_mul_BCE.to_vector<float>(0).extract<4>(3));
72      aie::vector<float, 4> y_C(0.0f, aie::reduce_add(v_mul_BCE.extract<4>(1)
        .to_vector<float>(0)), 0.0f, 0.0f);
73      y_C = aie::add(y_C, v_mul_BCE.to_vector<float>(0).extract<4>(3));
74      aie::vector<float, 4> y_E(0.0f, 0.0f, aie::reduce_add(v_mul_BCE.extract
        <4>(2).to_vector<float>(0)), 0.0f);
75      y_E = aie::add(y_E, v_mul_BCE.to_vector<float>(0).extract<4>(3));
76
77      // partial results update
78      p_B.push(y_B[0]);
79      p_B.push(x_step2[0]);
80      p_C.push(y_C[1]);
81      p_C.push(x_step2[1]);
82      p_E.push(y_E[2]);
83      p_E.push(x_step2[2]);
84
85      ////////////////////////// OUT PART
86      // partial sum
87      auto y_BE = aie::add(y_B, aie::shuffle_down(y_E,2));
```

```
88      float y_mux;
89
90      // multiplexer
91      if(INTEGRATOR_RST){
92          y_mux = 0.0f;
93      }else{
94          y_mux = p_D[1];
95      }
96      aie::vector<float, 4> y_mux_v(y_mux, 0.0f, 0.0f, 0.0f);
97
98      //final sum
99      auto y_fin = aie::add(y_BE, y_mux_v);
100     writeincr(y, y_fin[0]);
101
102
103     ////////////////////////// LIMITER
104     aie::vector<float, 4> y_lim_in(aie::add(y_mux_v, aie::shuffle_down(y_C
        ,1))[0],0.0f,0.0f,0.0f);
105     auto y_lim_out = y_lim_in;
106     if (LIMITER_ENABLE){
107         if (aie::gt(y_lim_in, LIMITER_UPPER_LIMIT).test(0)) {
108             y_lim_out[0] = LIMITER_UPPER_LIMIT;
109         } else if (aie::lt(y_lim_in, LIMITER_LOWER_LIMIT).test(0)) {
110             y_lim_out[0] = LIMITER_LOWER_LIMIT;
111         }
112     }
113
114
115     ////////////////////////// FILTER D
116     auto x_D = aie::mul(y_lim_out,c_D[5]).to_vector<float>(0);
117     aie::vector<float, 8> v_sample_D = aie::concat(p_D, x_D);
118     v_mul = aie::mul(v_sample_D,c_D);// multiplication
119     float y_D = aie::reduce_add(v_mul.to_vector<float>(0).extract<4>(0));//
        sum of v_mul
120     aie::vector<float, 4> y_D_v(y_D, 0.0f, 0.0f, 0.0f);
121     y_D = aie::add(y_D_v, v_mul.to_vector<float>(0).extract<4>(1))[0];
122     p_D.push(y_D);
123     p_D.push(x_D[0]);
124 }
```

64 bit version

```
1  #include <adf.h>
2  #include <aie_api/aie.hpp>
3  #include <aie_api/aie_adf.hpp>
4  #include <aie_api/utils.hpp>
5
6  /*
7  x = [x]
8  p = [x(n-1), y(n-1), x(n-2), y(n-2)]
9  c = [b2, -a2, b3, -a3, b1, g, 0, 0]
10 y = [y]
11 */
12
13 // constants' section
14 static const bool LIMITER_ENABLE = false;
15 static const float LIMITER_LOWER_LIMIT = 0.0f;
16 static const float LIMITER_UPPER_LIMIT = 0.0f;
```

```
17 static const bool INTEGRATOR_RST = false;
18 static const aie::vector<float, 8> c_A(0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f,
       0.0f, 0.0f);//3
19 static const aie::vector<float, 8> c_B(0.0f, 0.0f, 0.0f, 0.0f, 0.16f, 1.0f,
        0.0f, 0.0f);//4
20 static const aie::vector<float, 8> c_C(0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,
       0.0f, 0.0f);//5
21 static const aie::vector<float, 8> c_D(0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,
       0.0f, 0.0f);//6
22 static const aie::vector<float, 8> c_E(0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,
       0.0f, 0.0f);//7
23
24 void IIR_CONTROLLER_1
25 (
26     input_stream<uint64>*  x, // input sample
27     output_stream<uint64>*  y // output
28 ) {
29
30     // partial results
31     static aie::vector<float, 4> p_A = aie::zeros<float, 4>();
32     static aie::vector<float, 4> p_B = aie::zeros<float, 4>();
33     static aie::vector<float, 4> p_C = aie::zeros<float, 4>();
34     static aie::vector<float, 4> p_D = aie::zeros<float, 4>();
35     static aie::vector<float, 4> p_E = aie::zeros<float, 4>();
36
37     // input section
38     uint64 x_real = readincr(x);
39     float* floats = (float*)&x_real;
40     float x_int = floats[0];
41
42     ////////////////////////////////////////////////// FILTER A
43     //  -> vector
44     aie::vector<float, 4> x_int_vA(x_int, 0.0f, 0.0f, 0.0f);
45     x_int_vA = aie::mul(x_int_vA,c_A[5]).to_vector<float>(0);
46
47     // v_sample should be [x(t-1), y(t-1), x(t-2), y(t-2), x, 0, 0, 0]
48     aie::vector<float, 8> v_sample_A = aie::concat(p_A, x_int_vA);
49
50     // operations for output
51     auto v_mul = aie::mul(v_sample_A,c_A);// multiplication
52     float y_A = aie::reduce_add(v_mul.to_vector<float>(0).extract<4>(0));//
       sum of v_mul
53     aie::vector<float, 4> y_A_v(y_A, 0.0f, 0.0f, 0.0f);
54     y_A = aie::add(y_A_v, v_mul.to_vector<float>(0).extract<4>(1))[0];
55
56     // partial results update
57     p_A.push(y_A);
58     p_A.push(x_int_vA[0]);
59
60     ////////////////////////////////////////////// FILTERS B,C,E
61     // gain multiplication
62     aie::vector<float, 4> x_step2(y_A, y_A, y_A, 0.0f);
63     aie::vector<float, 4> g_vector(c_B[5], c_C[5], c_E[5], 0.0f);
64     x_step2 = aie::mul(x_step2, g_vector).to_vector<float>(0);
65
66     // product
67     aie::vector<float, 16> v_sample_BCE = aie::concat(p_B, p_C, p_E,
      x_step2);
```

```
68      aie::vector<float, 16> v_coeff_BCE = aie::concat(c_B.extract<4>(0), c_C
        .extract<4>(0), c_E.extract<4>(0), aie::vector<float, 4>(c_B[4], c_C[4],
        c_E[4], 0.0f));
69      auto v_mul_BCE = aie::mul(v_sample_BCE,v_coeff_BCE);// multiplication
70
71      // reduction for the different filters
72      aie::vector<float, 4> y_B(aie::reduce_add(v_mul_BCE.extract<4>(0).
        to_vector<float>(0)), 0.0f, 0.0f, 0.0f);
73      y_B = aie::add(y_B, v_mul_BCE.to_vector<float>(0).extract<4>(3));
74      aie::vector<float, 4> y_C(0.0f, aie::reduce_add(v_mul_BCE.extract<4>(1)
        .to_vector<float>(0)), 0.0f, 0.0f);
75      y_C = aie::add(y_C, v_mul_BCE.to_vector<float>(0).extract<4>(3));
76      aie::vector<float, 4> y_E(0.0f, 0.0f, aie::reduce_add(v_mul_BCE.extract
        <4>(2).to_vector<float>(0)), 0.0f);
77      y_E = aie::add(y_E, v_mul_BCE.to_vector<float>(0).extract<4>(3));
78
79      // partial results update
80      p_B.push(y_B[0]);
81      p_B.push(x_step2[0]);
82      p_C.push(y_C[1]);
83      p_C.push(x_step2[1]);
84      p_E.push(y_E[2]);
85      p_E.push(x_step2[2]);
86
87      ////////////////////////// OUT PART
88      // partial sum
89      auto y_BE = aie::add(y_B, aie::shuffle_down(y_E,2));
90      float y_mux;
91
92      // multiplexer
93      if(INTEGRATOR_RST){
94          y_mux = 0.0f;
95      }else{
96          y_mux = p_D[1];
97      }
98      aie::vector<float, 4> y_mux_v(y_mux, 0.0f, 0.0f, 0.0f);
99
100     //final sum
101     auto y_fin = aie::add(y_BE, y_mux_v);
102     float raw[2] = {y_fin[0], y_fin[0]};
103     uint64 y_real = *((uint64_t*)raw);
104     writeincr(y, y_real);
105
106
107     ////////////////////////// LIMITER
108     aie::vector<float, 4> y_lim_in(aie::add(y_mux_v, aie::shuffle_down(y_C
        ,1))[0],0.0f,0.0f,0.0f);
109     auto y_lim_out = y_lim_in;
110     if (LIMITER_ENABLE){
111         if (aie::gt(y_lim_in, LIMITER_UPPER_LIMIT).test(0)) {
112             y_lim_out[0] = LIMITER_UPPER_LIMIT;
113         } else if (aie::lt(y_lim_in, LIMITER_LOWER_LIMIT).test(0)) {
114             y_lim_out[0] = LIMITER_LOWER_LIMIT;
115         }
116     }
117
118
119     ////////////////////////// FILTER D
```

```
120     auto x_D = aie::mul(y_lim_out,c_D[5]).to_vector<float>(0);
121     aie::vector<float, 8> v_sample_D = aie::concat(p_D, x_D);
122     v_mul = aie::mul(v_sample_D,c_D);// multiplication
123     float y_D = aie::reduce_add(v_mul.to_vector<float>(0).extract<4>(0));//
        sum of v_mul
124     aie::vector<float, 4> y_D_v(y_D, 0.0f, 0.0f, 0.0f);
125     y_D = aie::add(y_D_v, v_mul.to_vector<float>(0).extract<4>(1))[0];
126     p_D.push(y_D);
127     p_D.push(x_D[0]);
128 }
```

This is the 64-bit version of the controller block, whose behavior is described in 4.1.1.

C.6 Input block

```
1  #include <adf.h>
2  #include <aie_api/aie.hpp>
3  #include <aie_api/aie_adf.hpp>
4  #include <aie_api/utils.hpp>
5
6  /*
7  x = [x]
8  p = [x(n-1), y(n-1), x(n-2), y(n-2)]
9  c = [b2, -a2, b3, -a3, b1, g, 0, 0]
10 y = [y]
11 */
12
13 // constants' section
14 static const float BIAS = 0.0f;
15 static const aie::vector<float, 8> c_r1(0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f,
       0.0f, 0.0f);//22
16 static const aie::vector<float, 8> c_r2(0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f,
       0.0f, 0.0f);//21
17 static const aie::vector<float, 8> c_s1(0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f,
       0.0f, 0.0f);//24
18 static const aie::vector<float, 8> c_s2(1.0f,
      0.980093260759718254426786643307423219085f, 0.0f, 0.0f, 1.0f,
      0.009953369620140867582436250415867107222f, 0.0f, 0.0f);//23
19
20
21 void IIR_input_block_1
22 (
23     input_stream<uint64>* x, // ratio, sum
24     output_stream<uint64>* y // ratio, sum
25 ) {
26
27     // partial results
28     static aie::vector<float, 4> p_r1 = aie::zeros<float, 4>();
29     static aie::vector<float, 4> p_r2 = aie::zeros<float, 4>();
30     static aie::vector<float, 4> p_s1 = aie::zeros<float, 4>();
31     static aie::vector<float, 4> p_s2 = aie::zeros<float, 4>();
32
33     // input reading
34     uint64 x_real = readincr(x);
35     float* floats = (float*)&x_real;
36     aie::vector<float, 4> x_int_v1(floats[0], floats[1], 0.0f, 0.0f);
```

```
37
38
39      ////////////////////////////////////////////////// FILTER r1, s1
40      aie::vector<float, 4> g_int_v1(c_r1[5], c_s1[5], 0.0f, 0.0f);
41      x_int_v1 = aie::mul(x_int_v1,g_int_v1).to_vector<float>(0);
42
43      // v_sample should be [x(t-1), y(t-1), x(t-2), y(t-2), x, 0, 0, 0]
44      aie::vector<float, 16> v_sample_1 = aie::concat(p_r1, p_s1, x_int_v1,
        aie::zeros<float, 4>());
45      aie::vector<float, 16> v_coeff_1 = aie::concat(c_r1.extract<4>(0), c_s1
        .extract<4>(0), aie::vector<float, 4>(c_r1[4], c_s1[4], 0.0f, 0.0f), aie
        ::zeros<float, 4>());
46
47      // operations for output
48      auto v_mul = aie::mul(v_sample_1,v_coeff_1);// multiplication
49      float y_r1 = aie::reduce_add(v_mul.to_vector<float>(0).extract<4>(0));
        // sum of v_mul
50      float y_s1 = aie::reduce_add(v_mul.to_vector<float>(0).extract<4>(1));
        // sum of v_mul
51      aie::vector<float, 4> y_1_v(y_r1, y_s1, 0.0f, 0.0f);
52      y_1_v = aie::add(y_1_v, v_mul.to_vector<float>(0).extract<4>(2));
53
54      // partial results update
55      p_r1.push(y_1_v[0]);
56      p_r1.push(x_int_v1[0]);
57      p_s1.push(y_1_v[1]);
58      p_s1.push(x_int_v1[1]);
59
60      ////////////////////////////////////////////////// FILTERS r2,s2
61      // -> vector
62      aie::vector<float, 4> g_int_v2(c_r2[5], c_s2[5], 0.0f, 0.0f);
63      y_1_v = aie::mul(y_1_v,g_int_v2).to_vector<float>(0);
64
65      // v_sample should be [x(t-1), y(t-1), x(t-2), y(t-2), x, 0, 0, 0]
66      aie::vector<float, 16> v_sample_2 = aie::concat(p_r2, p_s2, y_1_v, aie
        ::zeros<float, 4>());
67      aie::vector<float, 16> v_coeff_2 = aie::concat(c_r2.extract<4>(0), c_s2
        .extract<4>(0), aie::vector<float, 4>(c_r2[4], c_s2[4], 0.0f, 0.0f), aie
        ::zeros<float, 4>());
68
69      // operations for output
70      v_mul = aie::mul(v_sample_2,v_coeff_2);// multiplication
71      float y_r2 = aie::reduce_add(v_mul.to_vector<float>(0).extract<4>(0));
        // sum of v_mul
72      float y_s2 = aie::reduce_add(v_mul.to_vector<float>(0).extract<4>(1));
        // sum of v_mul
73      aie::vector<float, 4> y_2_v(y_r2, y_s2, 0.0f, 0.0f);
74      y_2_v = aie::add(y_2_v, v_mul.to_vector<float>(0).extract<4>(2));
75
76      // partial results update
77      p_r2.push(y_2_v[0]);
78      p_r2.push(y_1_v[0]);
79      p_s2.push(y_2_v[1]);
80      p_s2.push(y_1_v[1]);
81
82      // BIAS sum
83      aie::vector<float, 4> bias_v(BIAS, 0.0f, 0.0f, 0.0f);
84      y_2_v = aie::add(y_2_v, bias_v);
```

```
85
86      // output
87      float raw[2] = {y_2_v[0], y_2_v[1]};
88      uint64 y_real = *((uint64_t*)raw);
89      writeincr(y, y_real);
90
91  }
```

This is the input block, whose behavior is described in 4.1.1.

Bibliography

[1] AMD. *Versal: The First Adaptive Compute Acceleration Platform (ACAP)*, 2020. URL https://docs.amd.com/v/u/en-US/wp505-versal-acap. Accessed: 2025-04-01.

[2] AMD. *Versal ACAP DSP Engine Architecture Manual (AM004)*, 2022. URL https://docs.amd.com/r/en-US/am004-versal-dsp-engine/Overview?tocId=tItzyS6FCFNHd7m~GFfslQ. Accessed: 2025-04-01.

[3] AMD. *Versal Adaptive SoC AI Engine Architecture Manual*, 2023. URL https://docs.amd.com/viewer/book-attachment/juyVkQQPlmZlTyA3UczsEA/IHjwvrFtwe3n2UxK4COiHQ-juyVkQQPlmZlTyA3UczsEA. Accessed: 2025-04-01.

[4] AMD. *AXI4-Stream Infrastructure IP Suite v3.0*, 2023. URL https://docs.amd.com/viewer/book-attachment/iDBE_pP8gVw3O_auFL6zrQ/1SslvnphHVUfedW4_R2Hxg-iDBE_pP8gVw3O_auFL6zrQ. Accessed: 2025-07-11.

[5] AMD. *Versal Adaptive SoC AIE-ML Architecture Manual (AM020)*, 2024. URL https://docs.amd.com/r/en-US/am020-versal-aie-ml. Accessed: 2025-04-01.

[6] AMD. *AI Engine Tools and Flows User Guide*, 2024. URL https://docs.amd.com/r/en-US/ug1076-ai-engine-environment. Accessed: 2025-04-04.

[7] AMD. *AI Engine Kernel and Graph Programming Guide (UG1079)*, 2024. URL https://docs.amd.com/r/en-US/ug1079-ai-engine-kernel-coding?tocId=_G~tNVucqwC0CCt0l_v6bA. Accessed: 2025-04-01.

[8] AMD. *VCK190 Evaluation Board User Guide (UG1366)*, 2024. URL https://docs.amd.com/r/en-US/ug1366-vck190-eval-bd. Accessed: 2025-06-012.

[9] AMD. *Vitis Model Composer User Guide (UG1483)*, 2024. URL https://docs.amd.com/r/en-US/ug1483-model-composer-sys-gen-user-guide. Accessed: 2025-04-02.

[10] AMD. *Versal Adaptive SoC System and Solution Planning Methodology Guide (UG1504)*, 2024. URL https://docs.amd.com/r/en-US/ug1504-acap-system-solution-planning-methodology. Accessed: 2025-04-03.

[11] AMD. *Versal Adaptive SoC Programmable Network on Chip and Integrated Memory Controller v1.1*, 2025. URL https://docs.amd.com/r/en-US/pg313-network-on-chip. Accessed: 2025-06-12.

[12] AMD. *Vivado Design Suite User Guide: Programming and Debugging (UG908)*, 2025. URL https://docs.amd.com/r/en-US/ug908-vivado-programming-debugging/Trigger-At-Startup. Accessed: 2025-06-25.

[13] M. R. Casu and L. Macchiarulo. Adaptive latency-insensitive protocols. *IEEE Design & Test of Computers*, 24(5):442–552, 2007.

[14] Y. P. Chen, P. Maillard, R. D. Veggalam, S. R. Madem, E. Crabill, and J. Barton. 64mev proton single-event evaluation of xilinx single event mitigation (xilsem) firmware on 7nm versal™ acap devices. *IEEE Radiation Effects Data Workshop (REDW) (in conjunction with 2022 NSREC)*, 2022.

[15] F. Flores, O. L. Sánchez, L. J. Álvarez Ruiz de Ojeda, M. D. V. Peña, and J. M. V. Sánchez. Acceleration of a compute-intensive algorithm for power electronic converter control using versal ai engines. *Conference on Design of Circuits and Integrated Circuits (DCIS)*, 2024.

[16] J. Lei and E. S. Quintana-Orti. Mapping Parallel Matrix Multiplication in GotoBLAS2 to the AMD Versal ACAP for Deep Learning. *PECS '24*, 1607.

[17] P. Maillard, Y. P. Chen, J. Barton, and M. L. Voogel. Single event latchup (sel) and single event upset (seu) evaluation of xilinx 7nm versal™ acap programmable logic (pl). *IEEE Radiation Effects Data Workshop (REDW)v*, 2021.

[18] A. Mège and N. Wazad. Challenges in the board implementation of VERSAL ACAP on a space board. *Airbus Defence & Space, Space Electronics - EDHPC*, 2023.

[19] L. M. Ni and P. K. McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *Computer*, 26(2):62–76, 1993.

[20] R. F. Stengel. Toward intelligent flight control. *IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS*, 23(6):1699–1717, 1993.

[21] R. Sticca and A. Demichelis. Model Based Design (MBD) for hardware FPGA in Digital Flight Control Computer (DFCC). *POLARIS Innvation Journal*, 24:13–18, 2015.

[22] I. Swarbrick, D. Gaitonde, S. Ahmad, B. Jayadev, J. Cuppett, A. Morshed, B. Gaide, and Y. Arbel. VERSAL NETWORK-on-CHIP (NoC). *IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 13–17, 2019.

[23] TE connectivity. LVDT tutorial. https://www.te.com/en/products/sensors/position-sensors/resources/lvdt-tutorial.html, 2017. Accessed: 2025-05-23.

[24] Trenz Electronic. Amd versal ai edge evalboard with ve2302 device. https://shop.trenz-electronic.de/en/TE0950-03-EGBE21C-AMD-Versal-AI-Edge-Evalboard-with-VE2302-device-8-GB-DDR4-SDRAM-15-x12-cm, 2025. Accessed: 2025-04-03.

[25] XD100. *Vitis Tutorials: AI Engine Development (XD100)*, 2024. URL https://docs.amd.com/r/en-US/Vitis-Tutorials-AI-Engine-Development/Vitis-Tutorials-AI-Engine-Development-XD100. Accessed: 2025-26-05.