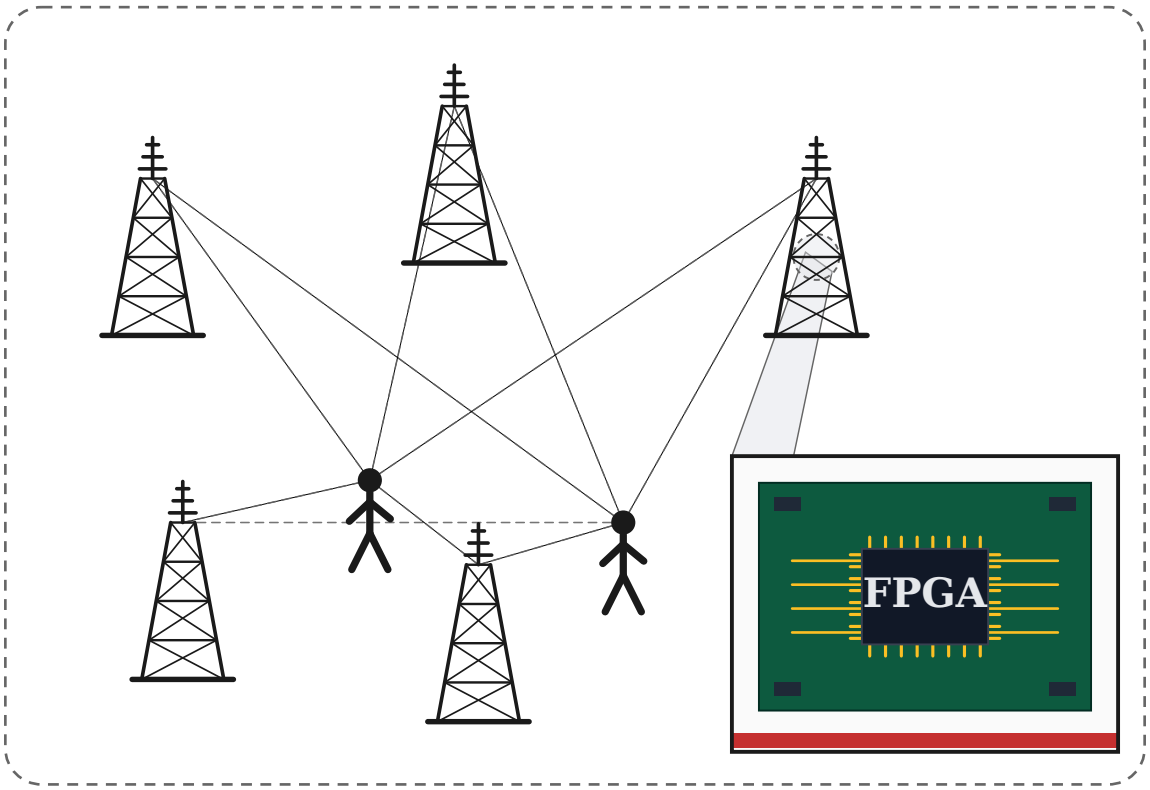
CHALMERS
UNIVERSITY OF TECHNOLOGY



FPGA-Based Platform Architecture for a Distributed MIMO (D-MIMO) 6G Testbed

Master's thesis in Embedded Electronic System Design

Öznur SOLAK

Department Microtechnology and Nanoscience
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026

MASTER'S THESIS IN EMBEDDED ELECTRONIC SYSTEM DESIGN

FPGA-Based Platform Architecture for a Distributed MIMO (D-MIMO) 6G Testbed

Öznur SOLAK

Department of Microtechnology and Nanoscience
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026

FPGA-Based Platform Architecture for a Distributed MIMO (D-MIMO) 6G Testbed
Öznur Solak

Supervisor: Lars Svensson
Examiner: Per Larsson Edefors

Master's Thesis in Embedded Electronic System Design
Department of Microtechnology and Nanoscience
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A D-MIMO service area in which distributed radio units serve several users cooperatively, and an FPGA at one of those units. The accelerator platform developed in this thesis targets that device; see Chapter 5.

Typeset in LaTeX
Gothenburg, Sweden 2026

Abstract

A distributed MIMO (D-MIMO) testbed changes shape as the experiments running on it change. On a conventional FPGA, replacing one accelerator means rebuilding and reprogramming the whole device, which takes the shared platform out of service for everybody. This thesis builds a Dynamic Function eXchange (DFX) platform for the ALINX Z19 board and its AMD Zynq UltraScale+ XCZU19EG, in which three regions of the fabric can be replaced independently while the rest of the design keeps running.

The static design holds the processing system, the DDR and network paths, three AXI DMA engines, the DFX Controller and the ICAPE3 configuration primitive. Around it sit three reconfigurable partitions that share one boundary contract: a 100 MHz clock, an active-low reset, a 32 bit control word, a 32 bit status word, a 32 bit AXI4-Stream port in each direction and a shutdown handshake. Because the three partitions present the same 146 pins, a module is written once; because they sit in different places on the die, it is built once per partition against that partition's abstract shell. An accelerator developer therefore needs neither the parent project nor its bitstream: a routed shell, one 1.75 MB checkpoint per partition, and their own VHDL are enough.

A standalone application on the Cortex-A53 serves a browser console and an HTTP API over lwIP. Uploading a partial bitstream to a slot validates it against the device IDCODE and the partition's own configuration frame address, hands the configuration port from PCAP to ICAPE3, and drives the virtual socket manager through shutdown, address programming, restart and trigger.

The implemented design closes timing with +4.342 ns of setup slack against 10 ns period with no failing endpoints and no unrouted nets, and it occupies 2.31 % of the device LUTs, 1.66 % of its registers and 0.61 % of its block RAM. It produced a 3.63 kB full bitstream and three partials of 2.59–3.19 MB, whose sizes track partition geometry rather than module content. At the 96.97 MHz PL clock recorded in the hardware handoff and the 32 bit ICAPE3 port, their ideal configuration-transfer times are 6.69–8.23 ms against 93.70 ms for the full image.

Hardware validation exposed four problems not revealed by reports or simulation: ICAPE3 required software to transfer ownership of the configuration interface from PCAP; the unsupported Ethernet PHY required explicit RGMII skew programming; link speed had to be read after negotiation; and enabled but unused transceiver peripherals could stall processor initialisation and JTAG until the board was power-cycled. After these corrections, the board boots unattended, answers ping in 110 ms and reconfigures any partition over HTTP while the other two continue running. D-MIMO physical-layer accelerators remain future work.

Keywords: D-MIMO, DFX, partial reconfiguration, abstract shell, Zynq UltraScale+, FPGA

Acknowledgements

I would like to express my sincere gratitude to Lars Svensson for proposing the project and for his continuous guidance and support. His extensive knowledge, regular feedback, and commitment through weekly meetings have been invaluable in keeping this work on track.

I would also like to thank everyone who has supported me throughout this journey, especially my parents, my husband, and my daughter, for their encouragement, patience, and unwavering support.

I would like to express my sincere gratitude to my dear friend, Hakim Male, for his unwavering support, encouragement, and motivation throughout this journey.

Öznur SOLAK, Gothenburg, September 2026

Abbreviations

6G	sixth-generation mobile communications
API	application programming interface
ARP	Address Resolution Protocol
AXI	Advanced eXtensible Interface
AXI4-Lite	AXI4-Lite memory-mapped interface
AXI4-Stream	AXI4-Stream protocol interface
BRAM	block random-access memory
BSP	board support package
CAN	Controller Area Network
CLB	configurable logic block
CPU	central processing unit
D-MIMO	distributed multiple-input multiple-output
DDR	double data rate memory
DFX	dynamic function exchange
DHCP	Dynamic Host Configuration Protocol
DMA	direct memory access
DNS	Domain Name System
DSP	digital signal processing
ELF	Executable and Linkable Format
eMMC	embedded MultiMediaCard
FIFO	first-in, first-out
FPGA	field-programmable gate array
GEM	Gigabit Ethernet MAC
GPIO	general-purpose input/output
GPU	graphics processing unit
GUI	graphical user interface
HDL	hardware description language

HLS	high-level synthesis
HTTP	Hypertext Transfer Protocol
I2C	Inter-Integrated Circuit
ICAP	Internal Configuration Access Port
ICAPE3	Internal Configuration Access Port, UltraScale-generation primitive
ICMP	Internet Control Message Protocol
IDE	integrated development environment
IP	Internet Protocol; intellectual-property core, according to context
IPv4	Internet Protocol version 4
JSON	JavaScript Object Notation
JTAG	Joint Test Action Group debug interface
LUT	look-up table
lwIP	lightweight IP
MAC	media access control
MDIO	management data input/output
MII	media-independent interface
MIMO	multiple-input multiple-output
MIO	multiplexed input/output
MM2S	memory-mapped to stream
MPSoC	multiprocessor system on chip
PC	personal computer
PCIe	Peripheral Component Interconnect Express
PHY	physical-layer transceiver
PL	programmable logic
POSIX	Portable Operating System Interface
PS	processing system
QSPI	quad serial peripheral interface
RF	radio frequency
RFSoC	radio-frequency system-on-chip
RM	reconfigurable module
RP	reconfigurable partition
RTL	register-transfer level
S2MM	stream to memory-mapped
SCP	Secure Copy Protocol

SD	Secure Digital
SDR	software-defined radio
SHA-256	Secure Hash Algorithm 256-bit digest
SSH	Secure Shell
Tcl	Tool Command Language
TCP	Transmission Control Protocol
TLAST	AXI4-Stream packet-end indicator
TLS	Transport Layer Security
TNS	total negative slack
TREADY	AXI4-Stream transfer-ready signal
TVALID	AXI4-Stream data-valid signal
UART	universal asynchronous receiver-transmitter
URL	Uniform Resource Locator
USB	Universal Serial Bus
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VPN	virtual private network
VSM	Virtual Socket Manager
WNS	worst negative slack
XSA	Xilinx Support Archive

List of Figures

1.1 Conventional full-device and modular DFX update workflows. On the right, only the two shaded steps involve the accelerator author's own logic; the platform contributes the shell and the loader. 3

4.1 Reproducible hardware and software build pipeline, and the module build that branches off it. 22

5.1 ALINX Z19 development board and principal external interfaces. Image source: ALINX product page [7]. 25
5.2 System-level architecture of the ALINX Z19 DFX platform. The configuration path is drawn in the one accent colour: everything it touches belongs to the DFX Controller, and none of it passes through an AXI DMA. Line style separates the signals that cross a partition boundary through its decoupler from those that reach the partition directly. . . 29
5.3 One reconfigurable partition and the static logic on either side of its boundary. 31
5.4 The implemented floorplan, annotated over Vivado's device view of the routed design. *design_1_i*/*rp1* is magenta, *rp2* blue and *rp3* green; the static logic is the cyan placed in rows 0 to 2 around the processing system. 34
5.5 Bench Ethernet topology between the developer computer and the Z19. 39
5.6 Standalone application startup and network initialization. 39
5.7 Callback sequence for a status request. 41
5.8 HTTP upload and partial-reconfiguration sequence. 43
5.9 What the platform build produces and what a module author consumes. 47

6.1 Byte order on the path from *write_bitstream* to ICAPE3. The uploaded file carries the right bytes in the wrong words; the board corrects them in place rather than requiring a formatting step off-line. 52
6.2 Ethernet bring-up: what the BSP does with an unrecognised PHY, and the sequence the board actually needs. 55
6.3 One partial-bitstream load as it runs on the board, from the first TCP fragment to a reconfigured partition. 58

7.1 Proposed C/C++-to-module development flow. 61

A.1 *design_1*, the top level of the block design. The three blocks marked with the DFX badge are the reconfigurable partitions; each names the block design it contains. 73
A.2 *static_platform*: the processing system, the control and memory SmartConnects, the reset block, and one control/status GPIO per partition. 75
A.3 *dfx_management*: the DFX Controller, the *ICAPE3* wrapper it drives directly, and the concatenation and AXI GPIO that let software observe both. 77
A.4 *rp1_static*: the partition's AXI DMA and its DFX Decoupler. Only the two stream ports and the status word pass through the decoupler. . . 77
A.5 *rp1_rm0*, the reference module. The stream passes through untouched and the shutdown request is acknowledged immediately; the counter exists to give the status word something to report. 78

List of Tables

4.1 Platform requirements and acceptance criteria. 15
4.2 Standalone Vitis and PetaLinux trade-offs for the prototype. 16

5.1 How the processing system is built up. Each level is applied over the one above it in this table. 27
5.2 Z19 processing-system configuration used by the platform. 28
5.3 The reconfigurable-partition boundary. Every partition presents exactly these signals. 30
5.4 Capacity of the three partitions, counted from the implemented pblock tile lists. 33
5.5 What an abstract shell keeps and what it discards, taken element by element from the routed parent. 37
5.6 HTTP interface exposed by the standalone application. 40

6.1 Routed timing summary for the implemented design. 48
6.2 Placed resource utilization of the complete implemented design. . . . 49
6.3 Generated configuration artifacts and developer-kit checkpoints. . . . 50
6.4 Ideal configuration-payload transfer times at $387.87\,\mathrm{MB\,s^{-1}}$. 51
6.5 Acceptance checks performed on the Z19. 57

Contents

1 Introduction **1**
- 1.1 Problem description . . . 1
- 1.2 Aim and contributions . . . 2
- 1.3 Scope and limitations . . . 3
- 1.4 Report structure . . . 4
- 1.5 Ethical Societal and Ecological Aspects . . . 4

2 Technical background **5**
- 2.1 FPGA processing . . . 5
  - 2.1.1 Generic device and board-specific implementation . . . 5
  - 2.1.2 Dynamic function exchange . . . 6
  - 2.1.3 Decoupling and reconfiguration control . . . 6
- 2.2 AXI communication . . . 7
- 2.3 Accelerator sockets . . . 7
- 2.4 Floorplanning and interface stability . . . 8
- 2.5 Embedded control and remote loading . . . 8
- 2.6 Portable project generation . . . 9

3 Related work **10**
- 3.1 D-MIMO and massive-MIMO testbeds . . . 10
  - 3.1.1 Implementation patterns and FPGA motivation . . . 10
- 3.2 Partial reconfiguration platforms . . . 11
- 3.3 Position of this work . . . 12

4 Methodology **13**
- 4.1 Engineering design approach . . . 13
- 4.2 Design requirements and evaluation criteria . . . 14
- 4.3 Evidence model . . . 14
- 4.4 Embedded software environment selection . . . 14
- 4.5 Development procedure . . . 16
  - 4.5.1 Incremental integration strategy . . . 16
  - 4.5.2 Target and static-platform definition . . . 17
  - 4.5.3 Defining the boundary . . . 17
  - 4.5.4 DFX hierarchy . . . 18
  - 4.5.5 Floorplanning . . . 18

4.5.6 Distributing the platform, not the project . . . 20
4.5.7 Embedded application . . . 20
4.5.8 Portable packaging . . . 21
4.6 Verification and evaluation procedure . . . 21
4.6.1 Verification sequence . . . 21
4.6.2 Collected data and analysis . . . 23
4.6.3 Reproducibility and validity boundaries . . . 23

5 Platform architecture and implementation 25
5.1 Z19 processing-system configuration . . . 25
5.2 Static shell . . . 27
5.3 The partition boundary . . . 29
5.3.1 The reference module . . . 32
5.4 Floorplanning . . . 32
5.4.1 The partitions are not the same size . . . 33
5.4.2 What the floorplan costs . . . 34
5.5 DFX control architecture . . . 35
5.5.1 Who owns the configuration port . . . 36
5.6 Build flow and generated artifacts . . . 36
5.7 The abstract shell . . . 36
5.8 Vitis application and remote interface . . . 38
5.8.1 Network configuration . . . 38
5.8.2 Startup order . . . 39
5.8.3 The HTTP interface . . . 40
5.8.4 The console . . . 42
5.8.5 Loading a partial bitstream . . . 42
5.9 The developer kit . . . 44
5.9.1 One template . . . 44
5.9.2 Three rules . . . 45
5.9.3 Building a module . . . 46
5.9.4 HDL-language support . . . 46

6 Results 48
6.1 Implementation and timing . . . 48
6.2 Resource utilization . . . 49
6.3 Generated artifacts . . . 50
6.4 Configuration-transfer estimate . . . 50
6.5 Byte order at the configuration port . . . 51
6.6 Software build . . . 51
6.7 Bringing the platform up on hardware . . . 52
6.7.1 PCAP owns the configuration port . . . 53
6.7.2 The Ethernet PHY needed code of its own . . . 53
6.7.3 The negotiated speed lags the link . . . 54
6.7.4 Transceivers wedge the debug port . . . 55
6.7.5 Final status of the board implementation . . . 56
6.8 What was not measured . . . 56
6.9 Portability of the source package . . . 57

7 Discussion **59**
7.1 One boundary, three partitions . . . 59
7.2 Relevance to a D-MIMO testbed . . . 60
7.3 The abstract shell makes the platform usable . . . 60
7.4 Proposed HLS authoring path . . . 61
7.5 Partitions and modules in the implemented design . . . 61
7.6 Wrappers, uniformity and reserved capacity . . . 62
7.7 Interpretation of the results . . . 62
7.8 Remote operation and trust . . . 63
7.9 Board specificity and transferability . . . 64
7.10 Evaluation boundaries . . . 64

8 Conclusion and future work **65**
8.1 Future work . . . 66

Bibliography **68**

A The block design as built **72**
A.1 Top level . . . 72
A.2 The static platform . . . 74
A.3 Reconfiguration management . . . 76
A.4 One partition's static logic . . . 76
A.5 The reference module . . . 76

1

Introduction

Wireless testbeds must accommodate changes in research questions, signal-processing methods, and hardware requirements. D-MIMO is a relevant example: geographically separated radio units cooperate to serve users within a common coverage area. This arrangement can improve coverage and spatial diversity, but requires coordinated data movement, synchronization, processing, and system control across multiple nodes [1, 2].

A number of D-MIMO platforms have demonstrated distributed beamforming, localization, and real-time coordination [3, 4, 5]. Physical testbeds expose hardware, channel, synchronization, and timing constraints that are absent from offline simulation. Maintaining their usefulness also requires hardware that can accommodate experiments that were not anticipated during the initial design.

FPGAs support deterministic processing and application-specific parallel datapaths. Their conventional development cycle, however, requires synthesis, placement, routing, bitstream generation, and full-device programming after a hardware modification. Repeating this cycle for each accelerator change can reduce the availability and flexibility of a shared testbed.

Dynamic Function eXchange (DFX) allows a selected FPGA region to be reconfigured while the designed static region remains operational [6]. The design is divided into a static region and one or more reconfigurable partitions (RPs). Each RP has a fixed interface and contains one compatible reconfigurable module (RM) at a time. Alternative RMs may be developed for the same RP when they meet its interface and resource constraints.

This thesis applies DFX to the accelerator subsystem of a future D-MIMO testbed. It does not select or implement D-MIMO algorithms. It defines one accelerator socket, replicates it three times, and builds everything around it that a designer needs in order to write a module, get a partial bitstream out of it, and load that bitstream into a running board over the network.

1.1 Problem description

The motivating problem is how to build an FPGA platform that can be shared and modified without rebuilding the complete system for each experiment. The processing system, control infrastructure, data movement, and network access for other regions must remain unchanged while selected hardware accelerators

are exchanged independently. The boundary between platform-owned logic and experiment-specific logic must also be explicit.

There is a second problem behind the first one, and it is the one that decides whether a platform is actually usable by anybody but its author. A partial bitstream is only valid against the exact routed parent it was built into. If producing one requires the parent project, its sources and an hour of implementation, then every accelerator author becomes a platform maintainer, and the separation the platform was built for does not exist in practice. AMD's answer to this is the abstract shell: the routed static design reduced to the context immediately surrounding one partition, saved as a checkpoint [6]. Whether that is enough to hand somebody a partition and nothing else is a question this work sets out to answer by doing it.

The target device is the AMD Zynq UltraScale+ MPSoC XCZU19EG on the ALINX Z19 development board [7, 8], which provides a Processing System (PS), a large Programmable Logic (PL) fabric, and an internal configuration access port. The Z19 was selected because the XCZU19EG provides enough fabric for three sizeable reconfigurable regions while retaining space for the static shell. The design enables only the services the demonstrator requires: memory, serial communication, Ethernet, boot support, PL clocks, resets, and interrupts. Camera, display, CAN, and other unrelated peripherals are omitted.

Three engineering questions follow:

- What does a partition boundary have to look like for the same source to build into any of three physically different regions?
- How can the three regions be built, loaded and controlled independently while the static design continues to serve the network?
- What has to be handed to an accelerator author so that they can produce a valid partial bitstream without the parent project?

Figure 1.1 contrasts a conventional full-device update with the modular DFX workflow considered here.

1.2 Aim and contributions

The aim is to design, implement and run an FPGA platform that supports runtime replacement of hardware accelerators for future D-MIMO experiments. The work contributes:

- A static platform for the ALINX Z19 carrying the PS, three DDR-facing data paths, control, reconfiguration and network infrastructure;
- Three independently floorplanned RPs that present one identical 146-pin boundary contract, so that one module source can target any of them;
- A reference RM per partition, resident after a full-device program, used to establish the boundary and produce the first partial bitstreams;
- A DFX control path built from three decouplers, a three-manager DFX Controller and the UltraScale+ ICAPE3 primitive;

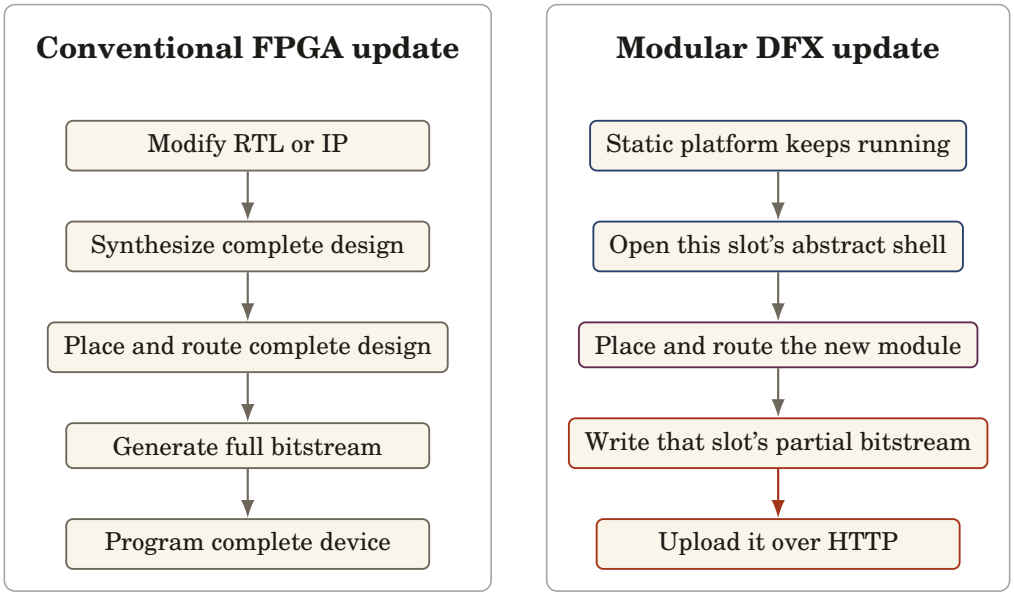


Figure 1.1: *Conventional full-device and modular DFX update workflows. On the right, only the two shaded steps involve the accelerator author's own logic; the platform contributes the shell and the loader.*

- A bare-metal Vitis application that serves a browser console and an HTTP API, validates an uploaded partial against the partition it names, and drives the reconfiguration handshake;
- An abstract-shell developer kit: one checkpoint per partition plus a build script, sufficient to produce a valid partial bitstream without the parent project; and
- A record of what it took to make the above work on real hardware, which is where most of the unrecorded knowledge in this kind of project lives.

Evaluation uses Vivado implementation reports, generated artifacts, Vitis build products and, unlike the plan this work started from, measurements taken on the board itself.

1.3 Scope and limitations

The scope is the FPGA platform and the workflow around deploying it. How D-MIMO algorithms behave internally, and how fast they run, is out of scope. The example modules demonstrate the boundary and nothing about radio processing performance. No radio front end, distributed synchronization system, fronthaul network or end-to-end D-MIMO physical layer is implemented here.

The remote loader is a proof of concept. It has no authentication at all: anything that can reach the board's TCP port can reconfigure the FPGA. That is a deliberate limitation of a bench instrument rather than an oversight, but it means the board belongs on an isolated segment, and production-grade security, hostile multi-tenant isolation, authenticated bitstreams and fault-tolerant recovery would all be required before access could be extended to untrusted users [9].

Integration with a complete radio testbed is also outside the reported evaluation. The results characterize the FPGA platform, its artifacts, its software and its behavior on the bench, not end-to-end wireless performance.

1.4 Report structure

Chapter 2 covers the FPGA, Zynq UltraScale+, AXI and DFX concepts the implementation depends on. Chapter 3 reviews related D-MIMO testbeds and reconfigurable FPGA platforms. Chapter 4 sets out the methodology, the design requirements, and the choice of embedded-software environment. Chapter 5 describes the Z19 architecture and its implementation, and Chapter 6 reports the build results and the bring-up on hardware. Chapters 7 and 8 close with a discussion and with conclusions and future work.

1.5 Ethical Societal and Ecological Aspects

The ability to share and reconfigure FPGA resources creates both access benefits and responsibilities. A remotely accessible shell can allow several researchers to reuse the same expensive platform and can lower the barrier to accelerator experiments. In a genuinely multi-tenant deployment, however, a faulty or hostile module could attempt to infer another tenant's activity through shared resources, corrupt data, exhaust bandwidth or prevent useful work. The present prototype assumes trusted users on an isolated laboratory network and therefore does not demonstrate hostile-tenant isolation. Responsible deployment would require authenticated and authorised loading, signed and traceable partial images, resource quotas, monitoring and audit logs, and a tested recovery path. These controls are prerequisites rather than optional additions when tenants do not trust one another [9] .

Reconfiguration can improve hardware reuse: a fixed shell can host different accelerators over its lifetime, which may reduce duplicated equipment and avoid replacing a board whenever the workload changes. FPGAs can also reduce the energy per completed task for workloads that map efficiently to spatial hardware. These advantages do not make the platform automatically sustainable. Device manufacture, static power, synthesis and implementation runs, network infrastructure, and unused capacity in oversized partitions all have environmental costs. This work did not measure energy consumption or embodied impact, so it makes no quantitative sustainability claim. A proper follow-up should report energy per processed data set, shell and reconfiguration overhead, utilisation, and expected hardware lifetime.

Generative-AI tools were used only for limited language editing and occasional clarification of terminology. They were not used to formulate the research questions, select the architecture, make design decisions, perform the implementation or experiments, generate results, conduct the analysis, or derive the conclusions. All technical content was independently verified against primary sources and experimental evidence. Responsibility for the accuracy and integrity of the report remains with the author.

2

Technical background

This chapter covers the concepts the platform architecture rests on. The emphasis throughout is on one boundary in particular: the one between the processing system, the programmable logic, and those regions of logic that can be swapped while the system is running.

2.1 FPGA processing

A field-programmable gate array (FPGA) contains configurable logic, registers, memories, arithmetic resources, routing, clocking, and I/O. A VHDL or Verilog description is synthesized onto these resources and converted into a configuration bitstream. A processor generally executes an instruction stream, whereas an FPGA implements spatial datapaths that can operate concurrently. This concurrency supports deterministic, high-throughput processing, subject to resource and timing constraints [10].

Four resource types are relevant to this report: lookup tables (LUTs), flip-flops, block RAMs (BRAMs), and DSP slices. LUTs and flip-flops implement control and general datapath logic, BRAMs provide on-chip storage, and DSP slices implement operations such as multiply-accumulate. Their physical distribution is important for DFX because an RP can use only the resources within its assigned floorplanning region.

The Zynq UltraScale+ MPSoC used here puts a Processing System (PS) and Programmable Logic (PL) on a single device. The PS holds Arm processors, memory controllers and peripherals; the PL is the configurable UltraScale+ fabric. AXI interfaces stitch the two domains together, so software can reach hardware registers, move data, take interrupts, and drive reconfiguration [11, 12].

2.1.1 Generic device and board-specific implementation

“Zynq UltraScale+” names a device family, not a working board. AMD’s documentation tells you which processors, DDR controllers, MIO peripherals, clocks, interrupts and PS–PL AXI ports exist, and what each Vivado processing-system parameter means [12, 13]. What it cannot tell you is which memory devices a board manufacturer soldered down, which oscillator they chose, which Ethernet PHY they wired up, or how the boot switches and connectors are arranged.

The Z19 is one such board-level realization. Its schematics and factory project pin down the exact XCZU19EG part, the PS reference clock, the DDR devices and their geometry, the MIO routing, the boot storage and the external peripherals [14, 15]. A design that selects only the generic part therefore knows what silicon it has but not enough about the wiring to bring up DDR or talk through the intended UART and Ethernet interfaces. Chapter 5 keeps these board-derived settings clearly separated from the PS–PL interfaces chosen specifically for the DFX platform.

2.1.2 Dynamic function exchange

Dynamic Function eXchange is AMD's mechanism for changing part of the programmable logic while the rest of the implemented design carries on [6]. Its vocabulary:

- The **static region** holds logic that stays configured throughout an exchange.
- A **reconfigurable partition** is a logical hierarchy with a fixed interface and an assigned physical region.
- A **reconfigurable module** is one implementation capable of occupying an RP.
- A **reconfigurable configuration** picks one RM per RP for implementation and bitstream generation.
- A **partial bitstream** carries configuration data for a single RP rather than the whole device.

Only one RM for a given RP is resident at a time. Several RPs can nevertheless be active at once, since each sits in a different physical region. The Vivado cell property *HD.RECONFIGURABLE* marks a hierarchical instance as a reconfigurable partition; in IP Integrator the same effect is obtained by enabling DFX on a block design container, which is the route this implementation takes. An RP with a single example RM establishes that the partition can be implemented and its partial artifact generated, whereas functional replacement requires at least two compatible RMs and runtime evaluation on hardware [6].

2.1.3 Decoupling and reconfiguration control

While configuration frames are being written, signals crossing an RP boundary may be undefined. A DFX Decoupler shields the static system by forcing selected interfaces to safe values for as long as the target RP is unavailable [16]. The decoupler lives in the static region and is driven as part of the exchange sequence.

The DFX Controller coordinates the exchange itself through one or more Virtual Socket Managers (VSMs) [17]. A VSM captures the state and control sequence belonging to one RP; the controller takes commands from software and reports back completion or error. In the architecture described here, each of the three accelerator sockets has its own VSM.

On UltraScale+ devices, configuration data enters the PL through the Internal Configuration Access Port, exposed as the *ICAPE3* primitive [18]. The static platform connects the DFX control path to this primitive. Embedded software supplies the

partial-bitstream buffer address and size, starts the selected VSM, and monitors its status. This configuration path is separate from the AXI DMA path used for accelerator data.

2.2 AXI communication

The Advanced eXtensible Interface (AXI) family provides standardized communication between IP blocks [19]. Three variants are used in this thesis project:

- **AXI4 memory-mapped** addresses locations in a memory or peripheral address space and supports burst transfers.
- **AXI4-Lite** is a reduced memory-mapped protocol used for control and status registers.
- **AXI4-Stream** carries an ordered sequence of data beats without addresses. Its *TVALID*/*TREADY* handshake provides backpressure.

An AXI DMA bridges DDR buffers and AXI4-Stream hardware: the memory-to-stream channel reads a buffer and emits stream beats, while the stream-to-memory channel writes returned beats to DDR [20]. This platform gives every reconfigurable partition its own AXI DMA, so experiment data for all three follows the same path in triplicate. Partial configuration does not use it at all; it follows a separate path through the DFX Controller and the internal configuration access port.

2.3 Accelerator sockets

An accelerator socket is a fixed boundary between platform-owned static logic and replaceable experiment logic. It specifies clocks, resets, data signals, control signals, and the behavior every compatible RM has to implement. The term describes an architectural contract, not any particular algorithm, and a socket is not itself a D-MIMO function: it is the way a future function is attached to a stable platform.

The socket used throughout this work is a 32 bit AXI4-Stream port in each direction, with *TVALID*/*TREADY* backpressure, a control word inbound and a status word outbound. Every partition presents that same boundary, so a module written once can be built into any partition, and the platform does not commit itself to a categorisation of algorithms that do not yet exist. Chapter 7 weighs what that uniformity costs.

An accelerator also need not be written directly in an HDL. High-level synthesis can turn a suitably constrained C or C++ top-level function into RTL, complete with interfaces such as AXI4-Stream, AXI4-Lite or AXI4 master [21]. Inside a DFX system this changes how candidate RM logic gets authored, but it removes none of what follows: the fixed partition interface, the Pblock resource budget, static-parent compatibility, Vivado implementation, and partial-bitstream generation all remain.

2.4 Floorplanning and interface stability

Every RP is assigned a Pblock: a physical area enumerating the logic, memory, and DSP resources it is permitted to use. The Pblock must be large enough for the intended class of accelerators and must respect the device's legal DFX boundaries.

Two Vivado Pblock properties are relevant here, although only the first appears in this project's constraint file.

***SNAPPING_MODE*:** This property permits Vivado to expand requested Pblock ranges to legal DFX resource and configuration-frame boundaries. The implemented floorplan sets it to *ROUTING*.

***RESET_AFTER_RECONFIG*:** This property requests initialization of the reconfigurable region after a partial reconfiguration, so that the new RM begins from its defined initialization state before normal operation resumes. This design does not set it, because the DFX Controller drives each partition's reset directly as part of the exchange sequence. Both are Vivado implementation properties rather than RTL signals [6].

Both the Pblock and the interface form part of the platform contract, and changing either can invalidate existing partial bitstreams. Long-lived sockets must therefore be defined independently of the initial reference logic. Partial-bitstream size is determined primarily by the configured physical region rather than the utilized logic within the RM. Survey treatments of dynamic partial reconfiguration describe the same two constraints: a reconfigurable region is composed of configuration frames, so its size and shape determine the partial bitstream, and every module targeting that region must present the same interface to the static side [22].

2.5 Embedded control and remote loading

The PS can run either a standalone application or a full operating system. This work uses a standalone Vitis application together with the lightweight IP (lwIP) network stack. The application reports status and accepts partial-bitstream uploads through an HTTP endpoint. A complete runtime operation has to validate the request and the uploaded image, make the staged bytes visible to the controller's bus master, quiesce and isolate the target RP, give the controller the buffer address and size, trigger it, confirm completion, and only then release the socket. On UltraScale+ it also has to hand ownership of the configuration port to the ICAP before any of that can take effect.

Remote reconfiguration drags a security boundary along with it. A malformed or incompatible partial bitstream can disrupt the PL, which is reason enough not to mistake a research demonstrator for a production multi-tenant service. Authentication, bitstream signatures, authorization, compatibility metadata, rollback and recovery are all separate requirements for any real deployment [9].

2.6 Portable project generation

Vivado and Vitis workspaces may contain generated metadata and host-specific paths. A transferable package should version the maintained RTL, constraints, Tcl scripts, application sources, and build instructions while regenerating workspace metadata on the receiving computer. Resolving relative paths from the script location avoids assumptions about usernames, drive letters, or installation directories. Tool discovery should likewise use the entry script or the user's initialized AMD environment rather than a fixed installation path.

Distributing the platform is a harder problem than distributing its sources, because a partial bitstream is valid only against the routed parent it was implemented into. AMD's abstract shell addresses this: it is the routed static design reduced to the context around a single partition, saved as a checkpoint, with everything else discarded [6]. The documentation presents it mainly as an intellectual-property mechanism, for handing out a partition without revealing the static design. It is equally useful for the ordinary case, because a developer holding a shell can place, route and generate a partial for that partition without the parent project, the parent bitstream or an hour of implementation. This thesis uses abstract shells as its distribution mechanism, and Chapter 5 describes what one contains.

3

Related work

This chapter relates the work to two bodies of literature: D-MIMO testbeds and reconfigurable FPGA platforms. The comparison concerns platform flexibility and experimental access rather than the relative performance of wireless algorithms.

3.1 D-MIMO and massive-MIMO testbeds

Bao et al. built an automatic D-MIMO testbed supporting coordinated experimentation across distributed radio units [3]. Aabel et al. followed with a time-division-duplex D-MIMO system using 1 bit radio-over-fiber, applied to distributed beamforming and localization [4]. NEC has reported a 28 GHz multi-user D-MIMO testbed aimed at evaluating coverage and spatial multiplexing [5]. These systems demonstrate the importance of physical testbeds for evaluating synchronization, calibration, fronthaul behavior, and real-time execution under implementation constraints.

Other massive-MIMO platforms illustrate the range of scales and implementation choices. LuMaMi was a real-time massive-MIMO testbed assembled around software-defined radios and FPGA processing [23, 24]. Argos and RENEW pursued scalable real-time research infrastructure along similar lines [25, 26], while Techtile explored a physically large, distributed experimental environment [27]. Despite their different objectives, each platform connects reusable infrastructure to experiment-specific processing.

3.1.1 Implementation patterns and FPGA motivation

The cited testbeds use heterogeneous processing architectures. In the automated Chalmers testbed, a PC performs signal processing offline while an FPGA transfers the generated waveform to 12 coherent remote transmitters over 10 $\mathrm{Gbit\,s^{-1}}$ optical links [3]. The later 1 bit radio-over-fiber system similarly concentrates high-rate conversion and waveform handling in a central hardware path. Its authors note that offline processing can leave minutes between estimating and applying a calibration [4]. RENEW combines programmable radio nodes and FPGA functions with host-side C++, Python, MATLAB, and CPU/GPU processing, allowing non-real-time exploration and real-time hardware functions to coexist [26, 28].

A recent preprint places more computation in distributed programmable hardware. The LuLIS testbed uses 16 Zynq UltraScale+ RFSoC boards as processing nodes

for up to 256 coherent RF chains, with real-time beamforming distributed across the FPGA fabric to reduce centralized data movement and latency [29]. Together with scalable cell-free processing studies [2], these systems identify three recurring implementation constraints: concurrent sample streams, deterministic timing and synchronization, and increasing fronthaul traffic when raw data is transported to centralized processing. CPUs provide flexible control, GPUs provide arithmetic throughput, and the cited architectures place FPGA logic near interfaces requiring deterministic streaming, parallel processing, or protocol-level timing.

These observations motivate an FPGA accelerator platform, but do not imply that every D-MIMO function should be implemented in RTL. The platform contribution is the ability to change selected hardware functions while retaining static communication, memory, control, and configuration services. It complements CPU-, GPU-, and SDR-based software: software orchestrates the experiment, while selected real-time paths may be implemented as replaceable RMs. This thesis evaluates that deployment mechanism rather than FPGA performance relative to CPU or GPU implementations.

The cited testbeds principally demonstrate communication-system capabilities. Remote replacement of selected FPGA regions is not a reported objective in their platform descriptions. This thesis addresses that narrower platform-engineering problem without replacing their radio, synchronization, or algorithmic contributions.

3.2 Partial reconfiguration platforms

Partial reconfiguration has been applied to software-defined radio, image processing, and other adaptable accelerators. Koch et al. survey design approaches and tools across these application domains [30], while Vipin and Fahmy survey the field more recently across three axes that this work also separates: device and configuration-interface architecture, design methods such as partitioning and floorplanning, and reported application domains [22]. Their application review is relevant here in two respects. Software-defined and cognitive radio appear as recurring cases, with modulation, coding and filter blocks adapted individually rather than duplicated as separate basebands. The architecture reported for those systems also matches the split used in this work: a static control plane running on the processor and a reconfigurable data plane in the fabric. AMD's DFX methodology defines the current flow for partitions, modules, configurations, floorplanning, and partial-bitstream generation [6].

Cloud and multi-tenant FPGA systems add isolation and security requirements because hardware modules may be only partially trusted. Elnaggar et al. examine these risks in shared FPGA infrastructure [9]. Their findings are relevant to the long-term remote-testbed goal, although the present implementation is limited to a trusted-user proof of concept.

3.3 Position of this work

The contribution combines a future D-MIMO application context with an implemented Z19 accelerator subsystem. Three RPs present one identical boundary through a common static platform, so that a module written once can be built into any of them. The supplied RMs are intentionally simple, allowing evaluation to focus on platform construction, the partition boundary, reproducible builds, and remote deployment.

The implemented flow distributes the platform as one abstract shell per partition rather than as the parent project, and it was run on the board. It does not conceal the static design, sign or version its artifacts, or supply algorithmic RMs; compatibility identifiers checked at upload time and D-MIMO modules for the sockets remain future extensions.

4

Methodology

The study is implementation-led. Requirements taken from the remote-testbed use case were turned into hardware and software structures, and those structures were then evaluated with source inspection, Vivado reports, generated artifacts, Vitis builds, analytical calculation, and finally measurement on the board.

4.1 Engineering design approach

The object of study is the platform artifact itself: its static services, its reconfigurable boundary, its build flow, its embedded loader, and the package handed to somebody who wants to write an accelerator for it. The work follows the design-science stages of problem definition, objective setting, artifact construction, demonstration and evaluation [31]. Comparing telecommunications algorithms is outside the scope.

Five stages made up the process:

- Derive operational platform requirements from the remote-testbed use case and the research questions in Chapter 1;
- Allocate each requirement to static hardware, the partition boundary, embedded software, or the build and packaging structure;
- Implement incrementally, validating the static design before the DFX configurations and the hardware handoff before the software platform;
- Collect tool-generated, artifact-based, analytical, source-inspection and bench evidence against acceptance criteria fixed in advance; and
- Revise the architecture when implementation, DFX legality, timing, portability or bring-up exposed a conflict.

That last stage did most of the work. Two of the design decisions reported in Chapter 5 (the shape of the floorplan and the order of the reconfiguration handshake) were reached by building something that failed and reading the error logs properly, not by reasoning ahead of time. The methodology section says so because the alternative is to present a set of choices as though they were obvious, which would misrepresent both the effort and the evidence.

The reference modules are implementation fixtures. They establish the boundary and let the parent implementation emit a first partial bitstream per partition. They are not evaluated as D-MIMO processing functions.

4.2 Design requirements and evaluation criteria

Chapter 1 states the platform problem, the aim and the research questions. The requirements below turn them into testable criteria and fix which evidence counts for each. They follow the traceability principle of ISO/IEC/IEEE 29148 without claiming formal compliance [32].

Table 4.1 shows platform requriments and acceptance criteria. R10 to R12 did not exist when the work started. The original plan stopped at R9 and treated the board as future work; the criteria were extended once the hardware was available, which is why Chapter 6 reports both kinds of evidence and keeps them apart.

4.3 Evidence model

Five kinds of evidence are used, and they are not interchangeable:

- **Source inspection** shows that a component, interface, constraint or script is present and connected as intended.
- **Tool validation** shows that Vivado synthesized, implemented, analysed timing and generated artifacts, or that Vitis compiled.
- **Artifact inspection** records properties such as generated byte counts, the target part and the presence of an ELF.
- **Analytical estimation** derives configuration-transfer bounds from artifact sizes, datapath width and the recorded configuration clock.
- **Bench measurement** records what the board did: whether it booted, whether it answered on the network, whether a partition reconfigured and what the loaded module reported.

Chapter 6 keeps these separate when reporting. An analytical bound and a stopwatch are different claims, and the report does not have a stopwatch: no reconfiguration wall-clock time, upload throughput or service-interruption figure was measured. What the bench establishes is that the sequence completes and the result is correct, not how long it takes.

No over-the-air experiment or radio-system measurement is included.

4.4 Embedded software environment selection

Two software environments were considered for the PS: a standalone Vitis application and an embedded Linux system generated with PetaLinux. Vitis supports standalone processor domains with a board support package and hardware-specific libraries; PetaLinux provides a kernel, device tree, root file system and boot-image flow [33, 34]. Table 4.2 compares them.

Standalone was chosen. The prototype serves one network service and drives a small set of memory-mapped peripherals; it needs no users, no persistent storage, no shell and no process isolation. Direct register access also keeps a visible correspondence between the DFX Controller sequence in the documentation and the code that drives

Table 4.1: *Platform requirements and acceptance criteria.*

ID	Criterion	Verification and pass condition
R1	Target fidelity	Vivado reports and the XSA identify *xczu19eg-ffvc1760-2-i*.
R2	One boundary, three partitions	Three pblocks exist and the three partition cells present identical pin lists; the parent implementation carries one reference module in each.
R3	Static isolation provisions	One decoupler and one VSM per partition; the DFX Controller and the *ICAPE3* primitive stay in the static design.
R4	Implemented parent	DFX design rules permit implementation, and the routed timing report gives WNS ≥ 0, TNS $= 0$ and no unrouted nets.
R5	Slot-specific artifacts	One full bitstream, one partial *.bit*/*.bin* pair per partition, and one abstract shell per partition are generated, with byte counts recorded.
R6	Software-build consistency	The exported XSA produces a standalone platform, a BSP and a Cortex-A53 application ELF.
R7	Loader implementation	Source inspection confirms bounded upload handling, image validation against device and partition, cache maintenance, the controller sequence, status reporting and error paths.
R8	Portable source organization	Every maintained script resolves its inputs from its own location; the project targets the exact part and needs no installed board file.
R9	Analytical transfer comparison	Payload reduction and the ideal ICAP transfer bound are computed from recorded binary sizes and the PL clock in the hardware handoff, excluding non-configuration overhead.
R10	Author-side buildability	A module written against the template, built only against an abstract shell, produces a partial bitstream the board accepts.
R11	Board bring-up	The board reaches its serving state without operator intervention after programming, and answers ICMP echo on the configured address.
R12	Runtime exchange	Each partition is reconfigured over HTTP while the other two remain resident and responsive, and the loaded module identifies itself afterwards.

Table 4.2: *Standalone Vitis and PetaLinux trade-offs for the prototype.*

Criterion	Standalone Vitis application	PetaLinux system
Execution model	Single application with direct control of initialization and polling	Linux kernel, processes, scheduler, and user/kernel separation
Hardware access	Direct register access using generated addresses and BSP drivers	Kernel drivers, device tree, and user-space interfaces
Networking	lwIP library; the application implements the TCP callbacks and its own HTTP handling	POSIX sockets and established Linux networking and web-server packages
Storage and services	No file system or background services unless added explicitly	File systems, SSH, logging, package integration, and service management
Build products	Platform, BSP, application, and ELF	Kernel, device tree, root file system, boot components, and applications
Best fit	Focused hardware-control prototype with one network service	Multi-service deployment needing operating-system facilities and user isolation

it, which mattered a great deal during bring-up: when a load hung, there were three registers to read and no driver stack in between.

The choice has a cost, and it showed up on the board. lwIP's raw callback API gives TCP/IP without a socket layer, so the application holds the connection state itself [35, 36], and because there is no scheduler, any long-running function called from a callback stalls ping, TCP and the console until it returns. The reconfiguration polling loops are exactly such functions. They are bounded by iteration count rather than by time, which is enough to stop a wedged controller from hanging the service forever but is not a substitute for a non-blocking state machine.

PetaLinux is not the wrong answer in general. It becomes the better one as soon as the requirements include SSH access, TLS, persistent storage, user accounts, several concurrent services, or continuous browser delivery of accelerator output over a WebSocket channel [37]. The selection made here is specific to the scope of this prototype.

4.5 Development procedure

4.5.1 Incremental integration strategy

Integration worked from the least replaceable elements towards the most changeable. The target device and the Z19 processing-system configuration came first, because

DDR, MIO, clocks, reset and the processor interfaces constrain everything downstream. The static AXI, memory, data-movement, control and configuration services followed. Only once those had stable endpoints was the partition boundary frozen; the DFX hierarchy, pblocks, isolation and reference modules were then built around a fixed interface. The export of implemented hardware to Vitis and the generation of the standalone software platform came last.

Each stage used the cheapest relevant check before moving on: source and address-map inspection before synthesis, block-design validation before implementation, DFX design-rule checks before artifact generation, and XSA consistency before application compilation. When a later stage failed, this ordering kept the list of possible causes short.

4.5.2 Target and static-platform definition

The exact XCZU19EG-FFVC1760-2-I part was selected in Vivado, with no installed board file. The processing system was then pared back to what the demonstrator needs: DDR-backed software, UART diagnostics, Ethernet, boot support, PL clocks, resets, AXI access and interrupts. Device and peripheral selections were checked against the Z19 material and the Zynq UltraScale+ documentation [7, 12]. Camera, display, CAN, I2C and PCIe were left out.

The programmable-logic subsystem was then assembled around the PS. Two AXI SmartConnects separate the two kinds of traffic: one on *M_AXI_HPM0_LPD* at 32 bit reaches every control register in the design, and one on *S_AXI_HP0_FPD* at 128 bit carries DDR traffic for the data-movement engines and for the DFX Controller. Each partition then received an identical set of static services: one simple-mode AXI DMA with an MM2S and an S2MM channel, one AXI GPIO carrying its 32 bit control and status words, and one DFX Decoupler at its boundary. A single DFX Controller holding three virtual socket managers, an *ICAPE3* wrapper and a 24 bit input-only AXI GPIO that reports every controller and ICAP event line complete the shell.

4.5.3 Defining the boundary

The partition boundary was defined before anybody decided what the reference modules would do, and it was defined once for all three partitions. Every partition presents the same signals: a clock, an active-low reset, a 32 bit control word inbound, a 32 bit status word outbound, a shutdown request and acknowledgement pair, a 32 bit AXI4-Stream slave port and a 32 bit AXI4-Stream master port, both carrying *TVALID*, *TREADY*, *TKEEP* and *TLAST*. That comes to 146 pins, and the three pin lists are identical.

One interface for all three partitions means one template, one set of protocol rules for an author to get right, and a module source that builds into whichever partition has the resources for it. It also keeps the platform from committing to a categorisation of accelerators before any exist: whatever a module needs beyond a word stream and a control word is expressed inside the module, where it can still be changed, rather

than in the partition pins, where it cannot. Chapter 7 weighs what that uniformity costs.

Fixing the interface first also stops the reference module from defining the contract by accident. Future modules may do anything they like internally, provided they keep the entity ports, the timing assumptions, the resource budget and the protocol behaviour the static design expects.

4.5.4 DFX hierarchy

Each partition is a Vivado block design container: a cell created with *create_bd_cell -type container -reference rpN_rm0* inside *design_1*, carrying *CONFIG.ENABLE_DFX*. Vivado turns a container with that property into a reconfigurable partition on its own, so the three cells appear throughout the flow as *design_1_i/rp1*, *design_1_i/rp2* and *design_1_i/rp3*, and the three reference module block designs are *rp1_rm0*, *rp2_rm0* and *rp3_rm0*.

Choosing containers over the classic flow removed a whole category of bookkeeping. There are no partition definitions to create, no reconfigurable-module objects to attach to them, and no parent/child configuration list to keep consistent with the sources. The constraint file, the artifact export and the software all address the same three cell names, which is one fewer place for a rename to go unnoticed. It also means the boundary is described once, in the module block design, rather than in a black-box entity that has to be kept in step with the real thing.

4.5.5 Floorplanning

Floorplanning can be approached at several levels. Vivado will place an ordinary static design without help, but DFX needs an explicit physical region for every partition. At the other extreme, cell-level placement controls individual sites and would tie the platform to whatever logic happens to be in it today. Hierarchy-level pblocks were chosen as the middle ground: they reserve a legal resource envelope and leave placement and routing inside it to the tool [6, 38].

The literature treats region selection as a constrained multi-objective problem. Resource- and reconfiguration-aware floorplacers account for heterogeneous resource columns and configuration granularity; fixed-outline simulated annealing searches candidate shapes while penalising area, wirelength and resource waste; recursive pseudo-bipartitioning heuristics add aspect-ratio constraints to improve routability [39, 40, 41]. The formal apparatus below frames what a later revision would optimise.

The first numerical test for a candidate region is resource feasibility. Let M_p be the set of modules intended for partition p, let $R_{m,k}$ be the demand of module m for resource type k, and let $C_{p,k}$ be the capacity inside the candidate pblock. A region is feasible only if

$$\max_{m \in M_p} R_{m,k} \le \eta_k C_{p,k} \qquad \text{for every resource type } k, \tag{4.5.5.1}$$

where k ranges over LUTs, registers, BRAMs and DSPs, and $0 < \eta_k < 1$ is an occupancy limit that keeps implementation headroom. There is no universal η_k; it has to be picked for the expected modules and confirmed by implementation. The remaining headroom is

$$H_{p,k} = \left(1 - \frac{\max_{m \in M_p} R_{m,k}}{C_{p,k}}\right) \times 100\,\%. \tag{4.5.5.2}$$

A more general search minimizes a weighted cost such as

$$\min_x J(x) = \alpha W_{\text{res}}(x) + \beta L_{\text{boundary}}(x) + \gamma P_{\text{congestion}}(x) + \delta B_{\text{partial}}(x), \tag{4.5.5.3}$$

subject to Equation 4.5.5.1, non-overlapping regions, legal DFX boundaries and timing closure, where W_{res} measures reserved-resource waste, L_{boundary} the routing distance to static interfaces, $P_{\text{congestion}}$ routing difficulty and B_{partial} the configuration payload.

None of this determined the floorplan actually used, and the reason is stated here rather than presenting the result as an optimization. Equation 4.5.5.1 needs M_p, and the set of modules intended for these partitions was empty: the platform exists so that accelerators can be written later. What did determine the coordinates was device legality, and on this part the legality constraints are strong enough to leave almost no freedom.

Three of them were found by builds that failed, and Section 5.4 gives the errors. A partition one clock region tall is illegal, because this device's programmable units are 180 rows tall and a reconfigurable pblock must contain whole units; Vivado's offered repair for that violation grows the region until it swallows *PS8* and *BUFG_PS* and makes all three partitions overlap. Clock-region rows 0 to 2 are unusable, because *CONFIG_SITE_X0Y0* lies in clock region X3Y1 and that is where *ICAPE3* must be placed. A partition covering it leaves the placer nowhere legal for the primitive that performs the reconfiguration. Clock-region column X0 is left alone, because it holds *PS8* and all 128 UltraRAMs and has no SLICEs below column X22 in these rows.

What survives those three constraints is one partition per remaining clock-region column, three clock-region rows tall, in rows 3 to 5: *pblock_rp1* over *CLOCKREGION_X1Y3:X1Y5*, *pblock_rp2* over *X2Y3:X2Y5* and *pblock_rp3* over *X3Y3:X3Y5*, each with *SNAPPING_MODE* set to *ROUTING*. The method here was elimination, not search.

The three regions are the same shape and different sizes, because the device's columns are not identical: counted from the implemented pblock tile lists, they hold 3600, 6480 and 5040 SLICEs and 288, 72 and 72 DSP48E2 sites respectively. Equalising them would have meant cutting all three to the smallest common denominator and discarding a large fraction of the usable fabric to make a table symmetrical. The asymmetry was kept and documented instead, and the module build script reports per-partition utilization so that an author can see which regions their design fits.

A workload-specific revision is a reasonable thing to do once real accelerators exist. Equations 4.5.5.1 to 4.5.5.3 describe what that revision would optimize, and Vivado's pblock visualization and post-placement congestion analysis are the tools for it [38].

4.5.6 Distributing the platform, not the project

A partial bitstream is only valid against the routed parent it was implemented into, which makes the obvious packaging strategy (ship the sources and let people rebuild) a poor one. It turns every accelerator author into a platform maintainer, and it puts an hour of parent implementation between somebody and their first test.

The method chosen instead was to export an abstract shell per partition [6]. *build_artifacts.tcl* calls *write_abstract_shell -cell* for each of the three partition cells alongside the partial bitstreams, so shells and partials are always generated together from the same routed parent and cannot drift apart. A module is then built by synthesizing it out of context, opening the shell for the target slot, linking the netlist into it, and writing that cell's bitstream with *write_bitstream -cell -bin_file*. Section 5.7 describes what a shell contains and Section 5.9 what the resulting kit looks like.

The reason for the choice is not confidentiality, which is the use case AMD's documentation emphasizes. It is that an accelerator author should not need the parent project at all, and with a shell they do not: they need Vivado, one 1.75 MB checkpoint, one VHDL file and a network route to the board.

4.5.7 Embedded application

The exported XSA feeds a standalone Vitis platform. On top of it the application handles board initialization, lwIP, the HTTP service, the module control interface and the reconfiguration sequence.

The runtime order for a load is fixed, and each step is where it is for a reason:

- Check the slot number and the advertised body length before anything is copied;
- Stream the body straight into that slot's DDR staging window, one TCP fragment at a time, so that a multi-megabyte image needs no heap allocation;
- Walk the uploaded image's configuration packets and check the device IDCODE and the first real frame address against the values recorded for that partition, because nothing inside a partial says which region it belongs to and the configuration port will happily program whichever region the file names;
- Byte-swap the image in place if it is in file order rather than the word order the controller's AXI master requires;
- Flush the CPU data cache over the staging window, because the uploaded bytes are in the cache and the controller is a separate bus master reading DDR;
- Command the VSM to shut down and poll until its shutdown bit sets;
- Write the bitstream-table address and size, which are writable only while the VSM is shut down;
- Command a restart and poll until the VSM leaves the shutdown state, because it ignores triggers until then: a trigger issued from shutdown is never acted on, and produces a controller sitting with no error set and a load that simply never completes;

- Trigger module row 0 and poll the VSM status register until it reports the full state or an error; the shutdown acknowledgement the controller waits for is the module's own *shutdown_ack*, in hardware, and needs no software step; and
- Report the decoded state, the controller's error field and the identity of whatever module is now resident.

Shutdown before the table write and restart before the trigger are not stylistic. Reversing either produces a silent failure rather than an error, which is the worst kind to debug, and both were found that way.

One further step sits outside this sequence and outside the block design entirely. On Zynq UltraScale+, *CSU_PCAP_CTRL.PCAP_PR* decides which port performs partial reconfiguration and comes out of reset selecting PCAP; while PCAP owns the port, *ICAPE3* never asserts its availability and the controller waits forever without reporting anything wrong. The application clears that bit during startup and verifies the write before it will accept any reconfiguration request.

4.5.8 Portable packaging

The transferable package is built from sources and recreation scripts rather than from a copied workspace. Two Tcl scripts do the whole hardware build, and both resolve every input from the location of the executing script, so the repository can be cloned or moved anywhere without editing a path. The project selects the exact part, so no installed ALINX board file is required on the receiving machine. Adding a target board means writing one file containing a part name, a floorplan and a processing-system configuration procedure; nothing else in the flow is board specific.

Generated Vivado and Vitis workspace metadata is treated as disposable output, because it carries host-specific paths and version state. The Vitis platform and application are likewise generated from the XSA and the maintained C sources by a script.

Portability was assessed by inspecting the maintained scripts for absolute development-machine paths, and then by recreating and running the project on a Windows host, which it completed successfully. The Vitis entry points are committed as a shell script and a batch file for each operation; both take the tool location from the environment.

Figure 4.1 draws the hardware and software procedures together. The XSA is the controlled handoff between Vivado and Vitis, and the abstract shells are the controlled handoff between the platform and anybody writing a module for it.

4.6 Verification and evaluation procedure

4.6.1 Verification sequence

The evaluation runs through the following checks, in this order:

- Validate and generate the Vivado block designs;

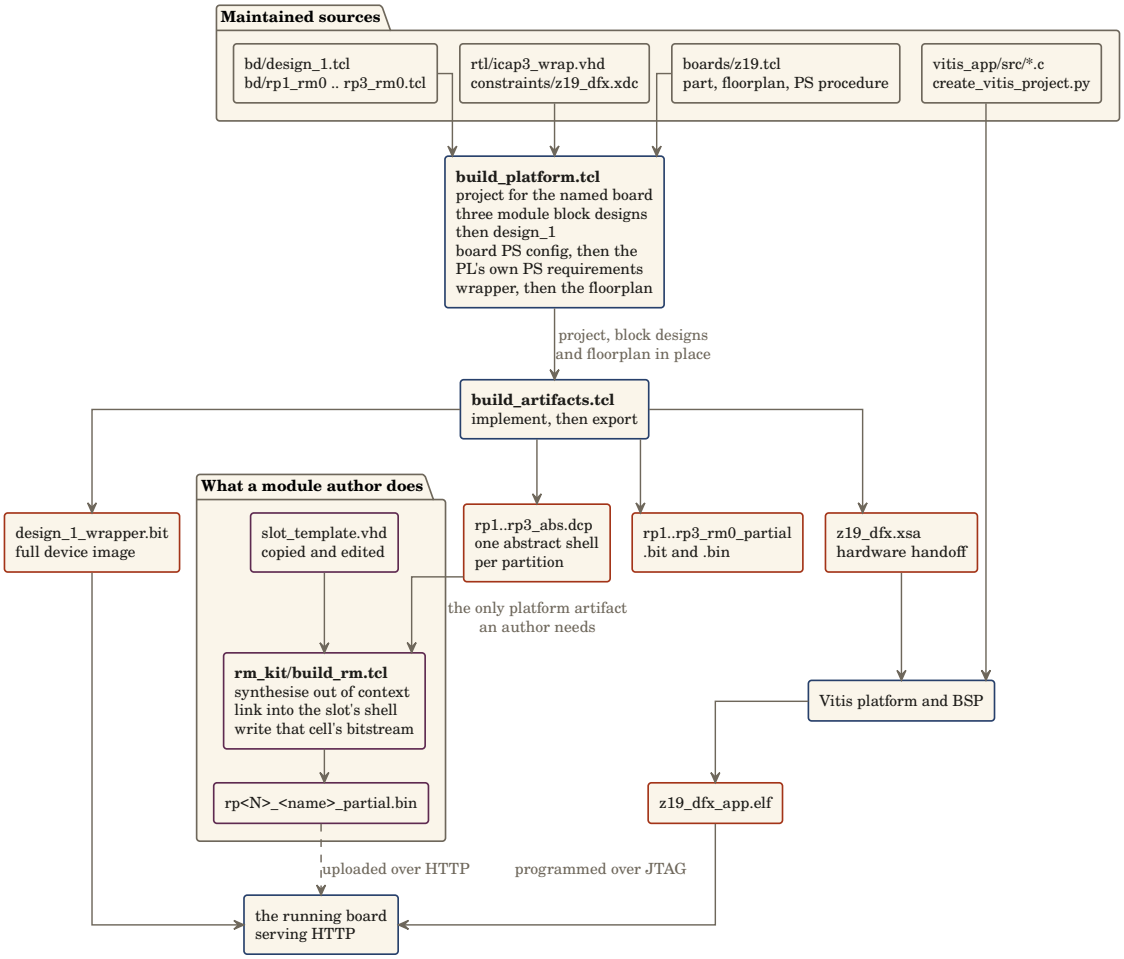


Figure 4.1: *Reproducible hardware and software build pipeline, and the module build that branches off it.*

- Synthesize and implement the parent, and run the DFX design rules;
- Run timing, utilization and route-status reports;
- Generate the full bitstream, one partial per partition and one abstract shell per partition;
- Export the XSA and build the Vitis platform, BSP and application ELF;
- Inspect the maintained scripts for machine-specific paths;
- Calculate the ideal configuration-transfer time from each raw binary size and the recorded PL clock;
- Program the board over JTAG and confirm from the console that it reaches its serving state without intervention;
- Confirm network reachability with ICMP echo from the connected host;
- Build a module from the template against one abstract shell, upload the resulting *.bin* over HTTP, and confirm that the controller reaches its full state and the module identifies itself; and
- Repeat that for all three partitions, confirming that the two partitions not being reconfigured stay resident and responsive throughout.

The ordering creates dependencies between evidence items. A partial is accepted only when it was generated from the intended parent after the DFX checks passed. An ELF establishes hardware and software build consistency only when its platform came from the XSA that the same hardware flow exported. And a bench result counts only against the artifacts that were actually programmed, which is why the artifact byte counts in Chapter 6 come from the same export that produced the images loaded on the board.

4.6.2 Collected data and analysis

The hardware data are the target part, the routed WNS, TNS, WHS and THS, the placed whole-design utilization, the route status, and the byte sizes of the full bitstream, the three partials and the three abstract shells. These come from Vivado reports and from the generated files. The software data are the platform and application build results. The bench data are the console output at startup, the ICMP round-trip result, and the loader's own JSON responses to each upload, which carry the controller state, the error field and the identity the loaded module reported.

Configuration-transfer time is treated strictly as an analytical lower bound. For an artifact of S bytes and a nominal ICAP data rate R,

$$t_{\text{ideal}} = \frac{S}{R}. \tag{4.6.2.4}$$

The implemented *ICAPE3* connection is 32 bit wide [18], and the clock is taken from the generated hardware handoff rather than from the round number that was requested: the design asks for 100 MHz and the handoff records 96.968727 MHz. Four bytes per cycle at that frequency gives an ideal payload rate of $387.87\,\text{MB}\,\text{s}^{-1}$. Controller sequencing, DDR arbitration, cache maintenance, software polling, Ethernet transfer and protocol overhead are all excluded, so the result bounds the configuration payload and nothing else.

4.6.3 Reproducibility and validity boundaries

Reproducing this work depends on the maintained block-design Tcl, the constraint file, the ICAPE3 wrapper, the C sources, the board file and the documented tool version. Generated Vivado and Vitis workspaces are outputs, not inputs. The XSA is the controlled boundary between the hardware and software procedures, and the per-partition frame addresses compiled into the application are derived from the floorplan, so they have to be read again from the generated partials whenever a partition moves.

The evaluation establishes a routed parent, the generated artifacts, a compiled loader, a working board and runtime reconfiguration of all three partitions. It does not establish timing figures for reconfiguration or upload, power consumption, behaviour under sustained load, radio integration or D-MIMO signal quality. The bench measurements were taken on one board, on a direct point-to-point Ethernet

link, with one operator present; they show that the design works, not how it behaves in a rack.

5

Platform architecture and implementation

This chapter describes the platform implemented for the XCZU19EG-FFVC1760-2-I device on the ALINX Z19: a static PS/PL shell, three independently reconfigurable partitions that present the same boundary, and the software that loads modules into them over the network.

Figure 5.1 identifies the principal external interfaces the target board offers. The photograph is there for physical context only: the camera connector, DisplayPort, CAN and USB are board capabilities that the DFX platform never enables. The annotated view is reproduced from the ALINX product material [7].

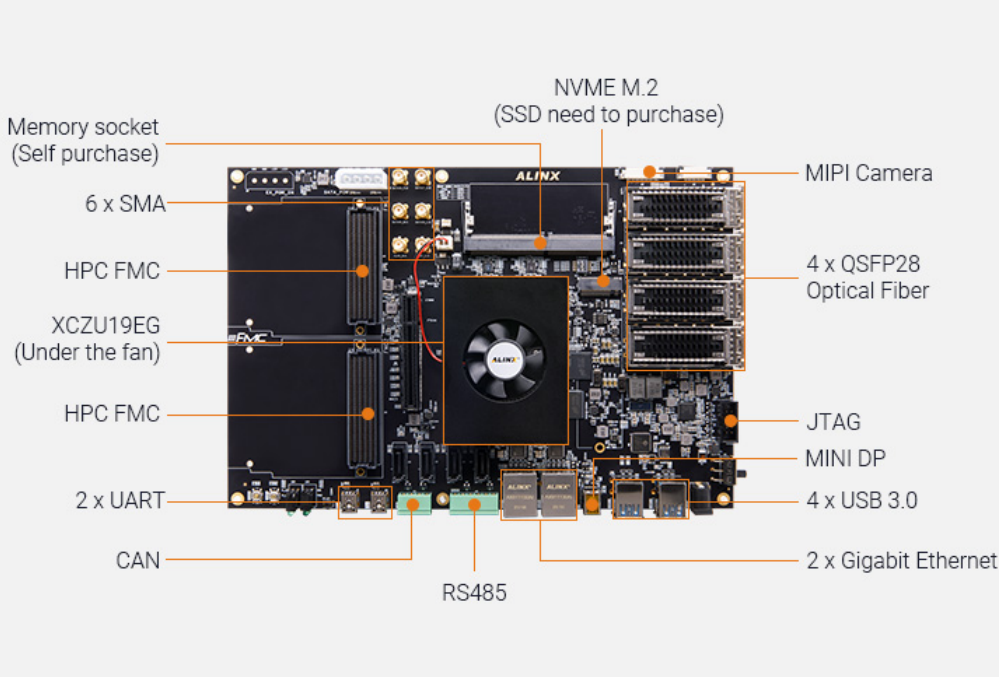


Figure 5.1: *ALINX Z19 development board and principal external interfaces. Image source: ALINX product page [7].*

5.1 Z19 processing-system configuration

Selecting the XCZU19EG part is not enough to bring up a Z19. The PS configuration has to match the memory devices and the pin routing actually fitted to the board,

and getting that wrong produces a design that synthesizes cleanly and then does nothing. The board-specific settings were taken from the ALINX schematics and factory project at commit *8f35a82d3781* and kept in a Tcl procedure of their own, so that they can be checked against those two sources without reading any of the DFX logic [8, 14, 15]. Because the factory project targets Vivado 2020.1, the script applies those settings in the current tool flow rather than shipping the older generated workspace as a preset.

How the configuration is layered matters. A block design exported with *write_bd_tcl* embeds the complete processing-system configuration of the project it was captured from: all 620 settings, 139 of them specific to a carrier board. Applying the Z19 settings on top of an embedded block leaves everything the Z19 settings do not explicitly name still in place underneath: a DDR geometry, an Ethernet controller, a set of UART pins, none of it announcing itself in the block design or in any report. So the script skips the embedded PS block entirely and builds the processing system from a board file instead. The PS then starts from Vivado's defaults for the part, the board file applies ALINX's configuration and switches off the peripherals discussed below, and a third file asserts what the programmable logic needs on any board at all: *M_AXI_HPM0_LPD* for registers, *S_AXI_HP0_FPD* at 128 bit for DDR traffic, *pl_clk0* at 100 MHz and the fabric reset. Board knowledge and design knowledge stay in separate files, and each can be checked against its own source.

The board file does one thing beyond passing on ALINX's settings, and the reason is worth recording. Once their configuration procedure has run, it disables seven peripherals: PCIe, DisplayPort, USB3_0, USB3_1, USB0, USB1 and SATA. Every one of them drives a gigabit transceiver. With any of them enabled, *psu_init* fails part way through its SERDES programming, at *mask_write 0XFE20C200*, and a failed *psu_init* leaves the debug access port in an AXI error state. The board then stops enumerating a PSU target over JTAG and needs a power cycle before it can be programmed again, which is an expensive way to discover that a peripheral nothing uses was left switched on. Since no part of this design drives PCIe, USB, DisplayPort or SATA, the whole transceiver bring-up is avoidable. A later design that needs one of them will have to re-enable it here and sort out the transceiver reference clocks.

Table 5.1 sets out the three levels and where each comes from: what AMD's device documentation fixes, what the Z19 schematics and factory Tcl determine, and what this project asks for [11, 12, 13, 14, 15]. They are applied in that order, and a later level replaces an earlier one wherever both name the same setting; anything a level does not name survives from underneath.

Table 5.2 compares the physical Z19 with the settings this project keeps. The values in the right-hand column were read back from the built design rather than from the script that requested them.

One number in that table is not what was asked for. The project requests 100 MHz on *pl_clk0*; the PLL can only divide the reference to 96.968727 MHz, and that is the value the generated hardware handoff records. Everything in the PL, including the ICAPE3 path, runs at it. The timing constraint remains the requested 10.000 ns period, so the design is analysed against a clock slightly faster than the one it gets, and the 96.97 MHz figure is what Section 6.4 uses for configuration throughput.

Table 5.1: *How the processing system is built up. Each level is applied over the one above it in this table.*

Level	Comes from	What it sets
1. The part	*xczu19eg-ffvc1760-2-i*, Vivado's defaults	What the silicon can do: the PS and PL capabilities, the DDR controller, which MIO groups exist, and the available PS–PL AXI ports, clocks and interrupts
2. The board	*z19_ps_config.tcl*, from the ALINX schematics and factory project	DDR4-2400P, 64 bit, four 16384 Mbit devices; GEM3 on MIO 64–75 with MDIO on 76–77; UART0 on MIO 38–39; QSPI on MIO 0–12, eMMC on SD0, microSD on SD1. Then the transceiver peripherals this platform does not use, switched off: PCIe, DisplayPort, USB, USB3 and SATA
3. The design	*ps_dfx_common.tcl*, board independent	*M_AXI_HPM0_LPD* at 32 bit, *S_AXI_HP0_FPD* at 128 bit, *pl_clk0* requested at 100 MHz, the fabric reset, and the DDR interface

5.2 Static shell

The static shell stays configured while a partition is exchanged. It holds:

- the Zynq UltraScale+ PS with its DDR, UART, Ethernet and boot interfaces;
- an AXI SmartConnect fanning the PS master out to every control register, and a second one with seven slave ports through which everything that touches DDR passes: the six AXI DMA channels and the DFX Controller's reads;
- three AXI GPIO blocks, one per partition, each driving a 32 bit control word and reading back a 32 bit status word;
- three AXI DMA engines in simple mode, one per partition, each with an MM2S and an S2MM channel between DDR and its partition;
- three DFX Decouplers, one per partition, at the partition boundaries;
- a DFX Controller holding three virtual socket managers (VSMs);
- an *ICAPE3* wrapper and a 24 bit input-only AXI GPIO that exposes every controller and ICAP event line to software; and
- the clock and reset infrastructure derived from *pl_clk0*.

The block design groups these into four hierarchies (*static_platform*, *dfx_management*, and one *rpN_static* per partition), which keeps each partition's DMA and decoupler together in the source and in the schematic.

Figure 5.2 separates the two kinds of traffic the design carries. Accelerator data moves between DDR and a partition through that partition's AXI DMA. Configuration data is read by the DFX Controller from a different DDR buffer and pushed through ICAPE3 into the fabric. What the two share is memory and the road to it: both reach DDR as independent masters on the same SmartConnect, which arbitrates between them. They share nothing beyond that. In particular the AXI

Table 5.2: *Z19 processing-system configuration used by the platform.*

Function	Z19 realization	Selected project setting
Device	XCZU19EG in the FFVC1760 package, speed grade −2, industrial temperature grade	Exact part *xczu19eg-ffvc1760-2-i*; no installed board-definition file is required
PS reference	33.33 MHz board oscillator	Used by the generated PS initialization; distinct from the requested PL fabric clock
DDR4	Four 16384 Mbit devices form a 64 bit bus, 8 GiB in total	DDR4-2400P geometry and training parameters from the ALINX configuration; the usable software range comes from the generated XSA and linker script
Ethernet	GEM3 through RGMII on MIO 64–75, with its own MDIO group on MIO 76–77	GEM3 and its MDIO enabled. Because the MDIO group belongs to GEM3, the generated BSP describes the controller that carries the traffic, and the application needs no controller remapping
UART	UART0 on MIO 38–39 through the board's USB–UART route, and UART1 on MIO 32–33	UART0 at 115200 Bd, used for boot and diagnostics. UART1 is enabled by the board configuration and left alone; nothing in this design uses it
Boot storage	Dual-parallel QSPI on MIO 0–12, eMMC on SD0, microSD on SD1	Retained as board boot options; JTAG execution still requires PS initialization from the matching XSA
PL services	Device provides configurable PS–PL AXI ports, fabric clocks, resets, and interrupts	*M_AXI_HPM0_LPD* for control registers, *S_AXI_HP0_FPD* at 128 bit for DDR traffic, fabric reset, and *pl_clk0* requested at 100 MHz
Other peripherals	The board also routes CAN, I2C, DisplayPort, PCIe, USB, camera and expansion interfaces	Omitted. The transceiver-backed ones are switched off explicitly rather than left at their factory values, for the reason given below

DMA plays no part in reconfiguration: the controller fetches the bitstream itself. That is the point the block diagram is drawn to make unambiguous.

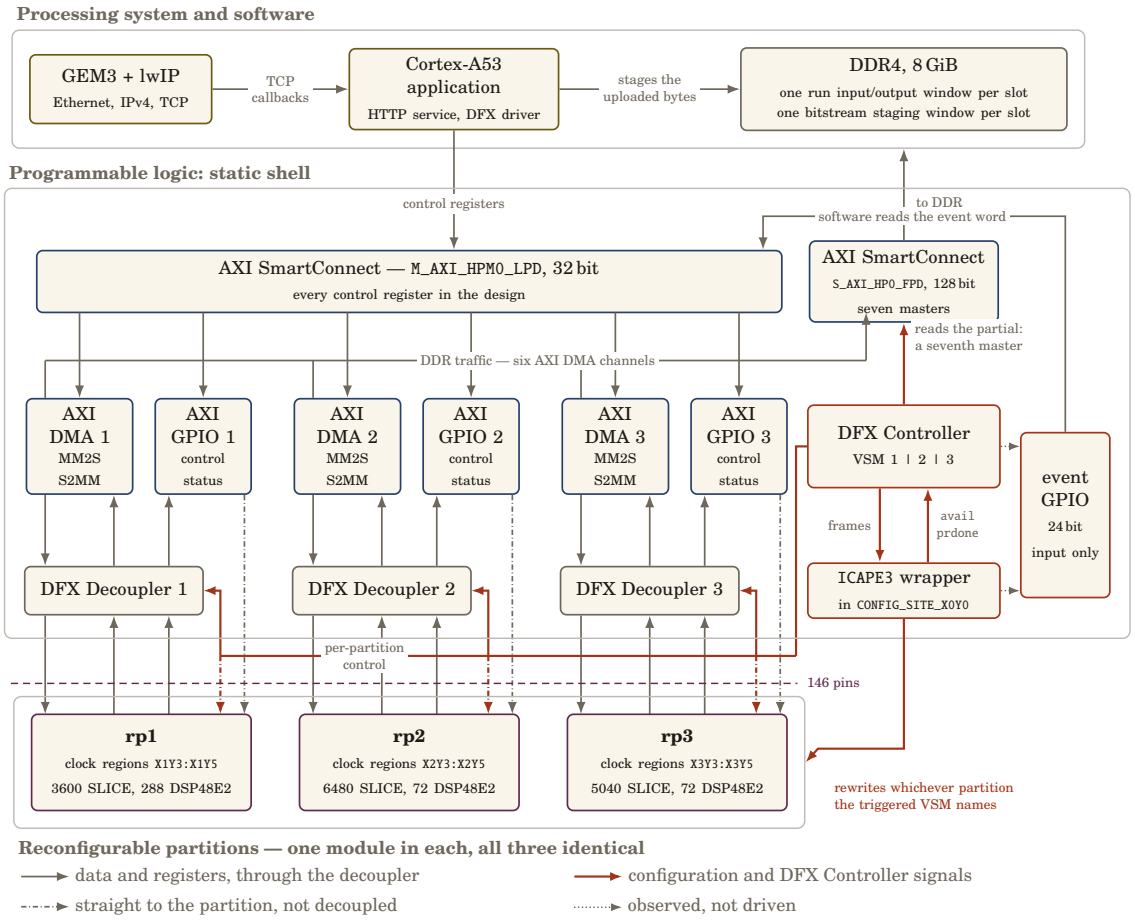


Figure 5.2: *System-level architecture of the ALINX Z19 DFX platform. The configuration path is drawn in the one accent colour: everything it touches belongs to the DFX Controller, and none of it passes through an AXI DMA. Line style separates the signals that cross a partition boundary through its decoupler from those that reach the partition directly.*

The three partitions are block design containers inside *design_1*, each referencing its own block design (*rp1_rm0*, *rp2_rm0*, *rp3_rm0*) and each marked with *ENABLE_DFX*. Vivado turns a container with that property into a reconfigurable partition without any separate partition-definition bookkeeping in the parent, which is why the cells appear in reports as *design_1_i/rp1*, *design_1_i/rp2* and *design_1_i/rp3* and why every constraint and every export in this project addresses them by those names.

5.3 The partition boundary

There is one boundary contract in this design, and all three partitions present it. That is the single most consequential architectural decision in the work: it is what makes a module source buildable into any partition, and it is what keeps the partition pins free of assumptions about accelerators that do not exist yet. Section 4.5.3 gives the reasoning and Chapter 7 what it costs.

The design puts one contract on every partition and an AXI DMA behind each. Table 5.3 gives it in full. The pin lists of the three partitions are identical, 146 pins each, which is what makes a single module source buildable into any of them.

Table 5.3: *The reconfigurable-partition boundary. Every partition presents exactly these signals.*

Signal group	Direction	Meaning
clk	in	96.97 MHz *pl_clk0*, constrained at 10.000 ns
resetn	in	Active low, driven by the partition's VSM
control[31:0]	in	Host-to-module word: *enable* at bit 0, *mode* in bits 7:4, *param* in bits 31:16
status[31:0]	out	Module-to-host word: *busy*, *done*, *error*, *irq* in bits 3:0, *code* in bits 11:4, *info* in bits 27:12
*s_axis_in_**	in	32 bit AXI4-Stream slave with *TVALID*, *TREADY*, *TKEEP* and *TLAST*; the words the host sent
*m_axis_out_**	out	32 bit AXI4-Stream master with the same signals; the words the module returns
shutdown_req	in	The controller asking the module to quiesce before it is replaced
shutdown_ack	out	The module confirming it has

Keeping *control* and *status* out of the datapath is what lets software start an operation, pass a small parameter and read a result code back without any of it appearing in the stream. Their bit layout is fixed platform-wide; the software unpacks it in *dfx_contract.h* and the module template unpacks the same fields in VHDL, so a module cannot quietly assign them differently. One mode value is reserved: mode *0xF* selects identify mode, and a module answering it puts its own identifier in *code* and its version in *info*. The console uses that to name what is resident in each slot instead of reporting three anonymous partitions.

The shutdown handshake is the part authors most often get wrong, so it needs spelling out. The DFX Controller will not begin writing configuration frames into a partition until that partition's module has confirmed it is finished. A module with no buffered state can wire *shutdown_ack* straight to *shutdown_req* and be done. A module that has words in flight has to stop accepting input, drain what it holds, and only then acknowledge. Otherwise the DMA on the other side of the boundary is left waiting for beats from logic that no longer exists.

Not every signal in Table 5.3 passes through the decoupler, and the split matters. Each *dfx_decoupler* is configured for three interfaces only: the two AXI4-Stream ports and the status word. Clock, reset, the control word and the shutdown handshake go straight to the partition. That is the right division: the decoupler exists to stop a half-configured partition from driving garbage into the static design, and only outputs and handshakes can do that, while the clock and the reset have to keep

reaching a partition that is being rewritten. One consequence is worth naming because software depends on it: the GPIO's status channel reads the decoupler's static-side output rather than the partition itself, so while a partition is decoupled its status word reads zero. That is hardware behavior, not a convention the application maintains.

One thing the contract does not carry is any route to a device pin. The design uses no programmable-logic I/O at all: the implementation report records zero bonded IOBs of the 512 available, because every external interface the platform needs, DDR, Ethernet, UART and the boot devices, belongs to the processing system's MIO rather than to the fabric. A module therefore reaches the outside world only through its stream ports and its control and status words, by way of DDR and the processing system. That suited this platform, whose accelerators are fed from memory, and it kept the boundary small. It also rules out a module that wants to drive an interface of its own, and Section 8.1 returns to what it would take to change that.

Figure 5.3 shows one partition and the static services around it. All three are wired identically.

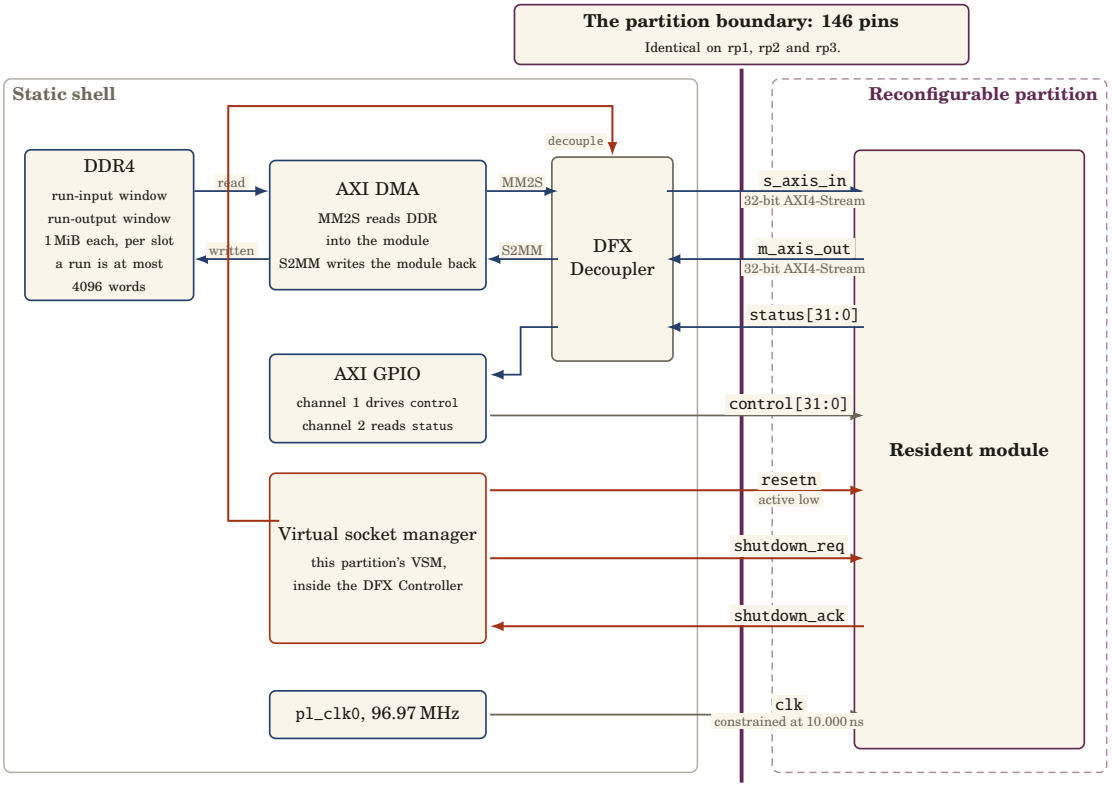


Figure 5.3: *One reconfigurable partition and the static logic on either side of its boundary.*

5.3.1 The reference module

Each partition is delivered occupied. A reference module sits in every partition after a full-device program, so that a freshly booted board is in a defined state and so that the parent implementation produces a first partial bitstream for each region. The reference is deliberately trivial: the input stream is connected straight to the output stream, *shutdown_req* is wired straight to *shutdown_ack*, and a 32 bit counter implemented in a DSP48E2 drives *status*, gated by *control* bit 0 and cleared by reset. It exercises the whole boundary (both stream directions, both control words, the shutdown handshake) without asserting anything about what an accelerator should do. It answers identify mode with zero, which is why the console shows “Unnamed module” for all three slots on a fresh boot.

5.4 Floorplanning

Each partition occupies one clock-region column, three clock-region rows tall, in rows 3 to 5. Listing 5.1 is the whole floorplan; there is no more to it than this.

```
create_pblock pblock_rp1
set_property SNAPPING_MODE ROUTING [get_pblocks pblock_rp1]
resize_pblock pblock_rp1 -add {CLOCKREGION_X1Y3:CLOCKREGION_X1Y5}
add_cells_to_pblock pblock_rp1 [get_cells design_1_i/rp1]

create_pblock pblock_rp2
set_property SNAPPING_MODE ROUTING [get_pblocks pblock_rp2]
resize_pblock pblock_rp2 -add {CLOCKREGION_X2Y3:CLOCKREGION_X2Y5}
add_cells_to_pblock pblock_rp2 [get_cells design_1_i/rp2]

create_pblock pblock_rp3
set_property SNAPPING_MODE ROUTING [get_pblocks pblock_rp3]
resize_pblock pblock_rp3 -add {CLOCKREGION_X3Y3:CLOCKREGION_X3Y5}
add_cells_to_pblock pblock_rp3 [get_cells design_1_i/rp3]
```

Listing 5.1: *The complete floorplan constraint file.*

Expressing the regions in clock-region units rather than site ranges is what keeps the file this short, and it is also what makes it legal. Three constraints fixed these coordinates, and each of them was found by a build that failed rather than assumed in advance.

A partition must be a whole number of programmable units tall. The first attempt gave each partition a single clock region. Vivado rejected it with HDPR-42: this device’s programmable units are 180 rows tall, reported as *SLICE_X22Y0:SLICE_X22Y179*, and a reconfigurable pblock has to contain whole units and be aligned to them. A clock region is 60 rows, so the minimum legal height is three of them. Vivado offers to repair the violation automatically, and its suggested repair should be refused: it grows the region until it is legal, which on this device pulls *PS8* and *BUFG_PS* into the partition and makes all three partitions overlap.

Rows 0 to 2 are unusable. *CONFIG_SITE_X0Y0* lies in clock region X3Y1. That site is where *ICAPE3* has to be placed, and *ICAPE3* belongs to the static design, because it is the primitive that performs the reconfiguration. A partition covering it leaves the placer with nowhere legal to put it:

```
ERROR: [Place 30-1100] Cannot place instance ... ICAPE3_inst in site
CONFIG_SITE_X0Y0 ... is outside its area constraints.
```

Listing 5.2: *What Vivado's placer reports when a partition covers the configuration site.*

Starting the partitions at row 3 clears the configuration site and the processing system underneath it in one step.

Clock-region column X0 is left alone. It contains the processing system and all 128 UltraRAMs, it carries the densest static routing on the die, and in rows 3 to 5 it has no SLICEs below column X22 at all. Using it would have bought little fabric and cost the static design its best routing territory.

What is left is one partition per remaining clock-region column, which is the arrangement in Listing 5.1. Figure 5.4 shows it on the device. It also shows how little of the device the result uses: the static design and the three partitions all sit in clock-region rows 0 to 5, and rows 6 to 10 carry no placed logic at all. That spare area is not free capacity in any simple sense. A fourth partition, or a taller one, would have to satisfy the same three constraints settled above, and none of those arrangements has been built or checked.

5.4.1 The partitions are not the same size

Because the three regions are the same shape but sit over different columns of the die, they do not have the same capacity. Table 5.4 gives what each one actually contains, counted from the tile lists Vivado exports for the implemented pblocks rather than estimated from the clock-region geometry.

Table 5.4: *Capacity of the three partitions, counted from the implemented pblock tile lists.*

Partition	Clock regions	SLICE	DSP48E2	RAMB36
rp1	X1Y3:X1Y5	3600	288	72
rp2	X2Y3:X2Y5	6480	72	72
rp3	X3Y3:X3Y5	5040	72	72

The asymmetry is the device's, not the design's, and it is useful rather than merely tolerated: rp1 is the arithmetic-dense region with four times the DSP columns of the other two, while rp2 has almost twice rp1's general logic. An author who runs out of DSPs in rp2 can rebuild the same source for rp1 by changing one command-line argument. What an author cannot do is assume that a module fitting one partition fits all of them, and the build script reports per-partition utilization for exactly that reason.

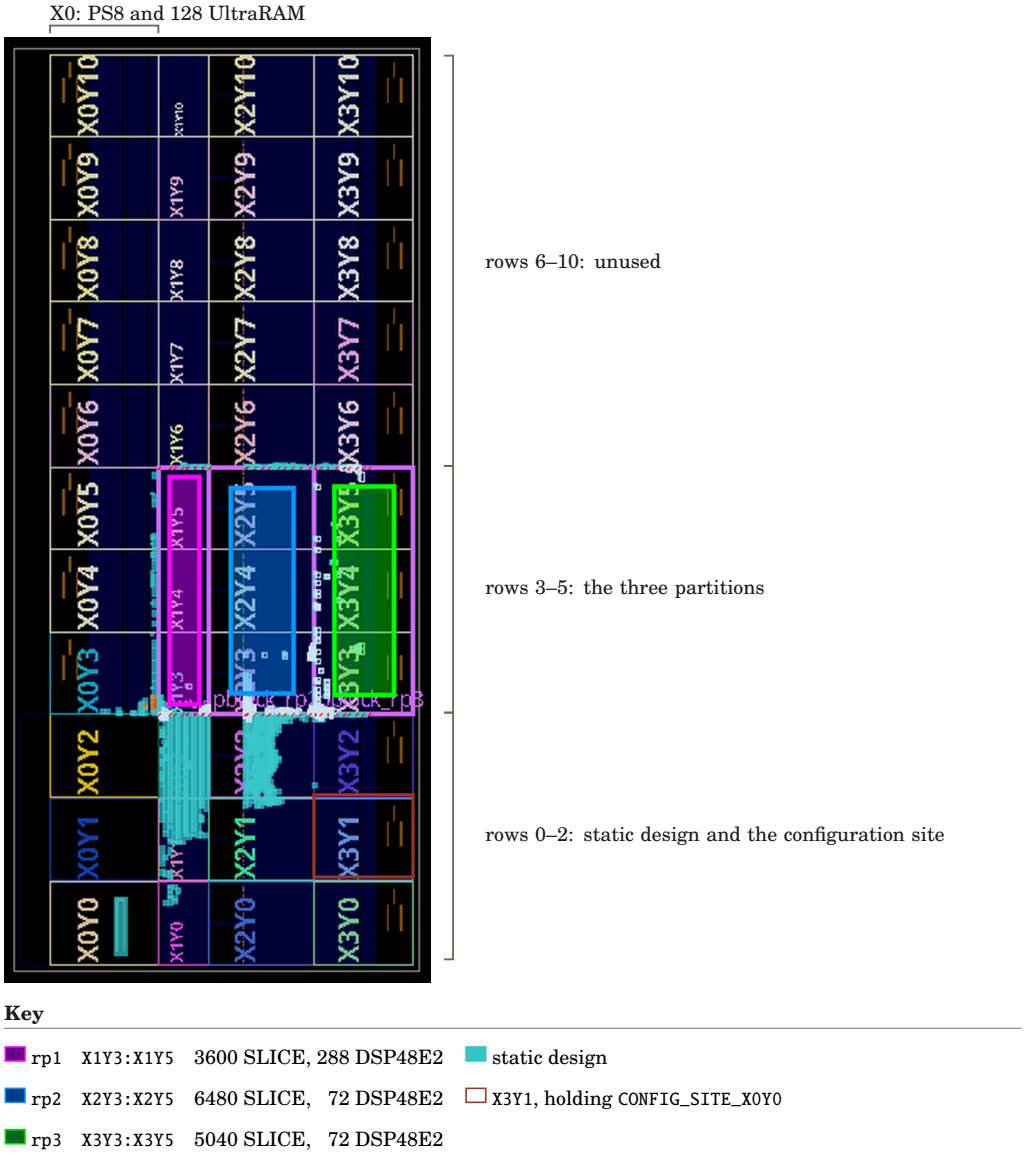


Figure 5.4: *The implemented floorplan, annotated over Vivado's device view of the routed design.* design_1_i/rp1 *is magenta,* rp2 *blue and* rp3 *green; the static logic is the cyan placed in rows 0 to 2 around the processing system.*

Equal-sized partitions were considered and rejected. They would have required cutting all three down to the smallest common denominator, which on this device means discarding roughly 40 % of the available fabric to make a table look tidy.

5.4.2 What the floorplan costs

Pblock geometry, not module content, determines how large a partial bitstream is. UltraScale+ configuration frames hold 93 words of 32 bit [18], so for a partition covering $N_{\mathrm{frame},p}$ frames the payload is approximately

$$B_p \approx 4\,N_{\mathrm{frame},p}(93) + B_{\mathrm{overhead},p}, \tag{5.4.2.1}$$

where $B_{\text{overhead},p}$ covers headers, configuration commands and alignment. A nearly empty module and a full one in the same partition produce nearly the same number of bytes. Section 6.3 reports the generated sizes, which follow the capacity ordering in Table 5.4 rather than anything about the reference modules.

Resource feasibility is also not sufficient on its own, because a legal pblock can still produce routes that miss timing. For each timing path i the design uses the usual slack relation

$$s_i = T_{\text{required},i} - T_{\text{arrival},i}, \qquad \text{WNS} = \min_i(s_i) \geq 0, \tag{5.4.2.2}$$

and long routes between a static service and a partition boundary raise $T_{\text{arrival},i}$ enough to invalidate a placement that is comfortable on resources alone. AMD accordingly treats floorplanning, congestion and routed timing as one closure activity rather than three [38]. Chapter 6 applies Equation 5.4.2.2 to the implemented design, and every module built later has to pass the same check inside its own partition.

There are alternatives to a hand-written floorplan. An automated search over location, aspect ratio, wirelength, resource waste and congestion is the systematic approach, and the literature describes several [39, 41]. Its objective function needs credible resource and connectivity estimates for the modules that will eventually be loaded, which a platform built before its accelerators does not have. Three fixed regions chosen against three hard legality constraints was the appropriate level of effort here; a workload-specific revision, once real accelerators exist, is a reasonable thing to do later.

5.5 DFX control architecture

The DFX Controller holds three VSMs, indexed to match rp1–rp3. Each VSM drives the reset and the decouple request for its own partition, receives that partition's *shutdown_ack*, and exposes its state through a register window at *0x80* intervals. The controller's memory master reads the partial bitstream straight from DDR; its ICAP signals go to the *ICAPE3* wrapper.

A VSM is a control state machine and not a container. The three partitions are separate physical regions sitting outside the controller entirely, and their decouplers belong to the static shell, placed at the boundaries so that isolation stays active while the logic inside a partition is being rewritten. Of everything in this section, the resident module is the only thing a partial bitstream replaces.

A 24 bit input-only AXI GPIO carries every interesting line from this subsystem back to software in one read: each decoupler's status, each VSM's shutdown request, decouple and reset outputs, each VSM's software shutdown and startup requests, and ICAP's *avail*, *prdone* and *prerror*. It costs almost nothing in fabric and it is the difference between diagnosing a stalled reconfiguration and guessing at it.

5.5.1 Who owns the configuration port

One register outside the PL decides whether any of this works, and nothing in the block design refers to it. On Zynq UltraScale+, *CSU_PCAP_CTRL.PCAP_PR* selects which port performs partial reconfiguration. It resets to 1, meaning PCAP, the processor configuration access port the boot ROM uses. While PCAP owns the port, *ICAPE3* never asserts its *avail* output.

The failure this produces is quiet. The DFX Controller accepts the trigger, enters its loading state, and waits there indefinitely. No error is reported, because from the controller's point of view nothing has gone wrong: it is waiting for a configuration port that simply never becomes free. Software therefore clears *PCAP_PR* during startup and verifies that the write stuck, before it will accept any reconfiguration request. Every other bit of the register is preserved, and the full bitstream has already been programmed by this point, so no PCAP transfer is in flight to interrupt.

5.6 Build flow and generated artifacts

Two scripts produce everything. *build_platform.tcl* takes a board name, sources that board's definition, creates the project against the exact part, builds the three module block designs and then the parent, applies the board's PS configuration followed by the design's own PS requirements, generates the wrapper, and adds the floorplan. *build_artifacts.tcl* implements the result and exports it. Both resolve every input relative to the script's own location, so the repository can be moved or cloned anywhere.

The export step produces four kinds of output:

- one full bitstream for the whole device;
- one partial bitstream per partition, in both *.bit* and raw *.bin* form, written with *write_bitstream -cell*;
- one abstract shell per partition, written with *write_abstract_shell -cell*; and
- the XSA that Vitis builds the software platform from.

Everything the Z19 contributes is confined to one file: a part name, a floorplan, and a PS configuration procedure. Nothing else in the flow is board specific. That is not a portability claim so much as a reviewability one: device knowledge, carrier knowledge and DFX knowledge each sit in a file that can be checked against its own source, and the floorplan, which is the one input that cannot be derived from anything, has a file to itself.

5.7 The abstract shell

A partial bitstream is valid only against the exact routed parent it was implemented into. That is a property of how DFX works: the wires crossing a partition boundary were placed when the parent was routed, and a module has to connect to those specific partition pins over that specific surrounding routing. Taken at face value, it

means anybody producing a module needs the parent project, and the separation between platform and accelerator that DFX is supposed to provide stops at the point where somebody actually wants to use it.

An abstract shell is AMD's answer [6]. It is the routed parent reduced to just the context around one partition (the partition pins, the routing that reaches them, the clocking, and the constraints), with the partition itself left as a black box, saved as a design checkpoint. Everything not needed to place and route logic into that one region is discarded, including the rest of the static design. For this platform the three shells are 1.75 MB, 1.73 MB and 1.72 MB, against a full routed parent checkpoint of 14.58 MB, a little over eight times the size.

Table 5.5 takes the design element by element and describes what happens to each one. The shell for *rp1* is written with *write_abstract_shell -cell design_1_i/rp1*.

Table 5.5: *What an abstract shell keeps and what it discards, taken element by element from the routed parent.*

Element of the design	**In the shell**	**Why**
Processing system: PS8, DDR4, GEM3, UART, boot	Discarded	None of it is needed to place logic into one partition
The two AXI SmartConnects	Discarded	As above
The three AXI DMA, AXI GPIO and DFX Decouplers	Discarded	As above
DFX Controller, *ICAPE3* and event GPIO	Discarded	As above
rp2 and *rp3*	Discarded	A shell is about one partition; the other two have shells of their own
rp1 itself	Kept, empty	A black box waiting for the author's netlist
rp1's 146 partition pins	Kept	Fixed where the parent routed them; a module must connect to these exact pins
The routing that reaches those pins	Kept	Without it the netlist has nothing to connect to
clk_pl_0 at 10.000 ns, and the constraints	Kept	The module is placed and routed against the parent's timing, not its own
Checkpoint	*design_1_wrapper_routed.dcp* is 14582263 B; *rp1_abs.dcp* is 1751660 B, a little over an eighth of it	

Three things follow from this, and together they are why the developer kit in Section 5.9 is as small as it is.

The parent design is not distributed. A shell contains the partition's context, not the design around it. An author can build a module without the block design, the PS configuration, the bitstream or the sources. That is useful here for practical reasons rather than confidentiality ones: the parent takes about an hour to implement and the shell opens in seconds.

One shell per partition, and the partition is what a shell is about. The three partitions present the same 146 pins, so one module source builds into any of them. But each build has to open the right shell, because the physical partition behind that identical boundary is in a different place with different pins and different routing. Selecting a slot is therefore selecting a shell, which is the single line that makes an otherwise slot-agnostic build slot-specific.

A shell binds a partial to one parent implementation. Rebuild the static design in any way that moves a partition, changes its boundary or reroutes its pins, and every existing partial for that partition is invalid, along with the shells that produced them. This is not a limitation of the shell mechanism; the same is true of partials built the long way. The shell just makes it visible, because the shells and the partials are regenerated together by the same export step.

5.8 Vitis application and remote interface

The XSA feeds a standalone Vitis platform for the Cortex-A53. The application runs on lwIP's raw callback API with no operating system, file system or shell [35, 36]. Its job is to bring the board up, keep an HTTP service running, and drive the DFX Controller when somebody uploads something.

5.8.1 Network configuration

Network settings are compile-time constants, because a bare-metal program has no persistent configuration store. The board takes a static *192.168.10.10*/*24* address with a gateway at *192.168.10.1*, a locally administered MAC address, and TCP port 80 so that a browser reaches it without a port suffix. DHCP support is present and compiled out: setting one macro makes the application request a lease at startup, wait up to twelve seconds, and fall back to the same static address if no server answers. Figure 5.5 shows the bench arrangement, which is a direct cable between the developer's computer and the board.

There is no authentication. Any client that can open a TCP connection to this port can reconfigure the FPGA, and the traffic is unencrypted. This is stated in the source, in the console page and here, because it is the kind of limitation that is easy to forget once something works: the board belongs on an isolated segment and its port must never be forwarded. Section 7.8 returns to what would have to change before that stopped being true.

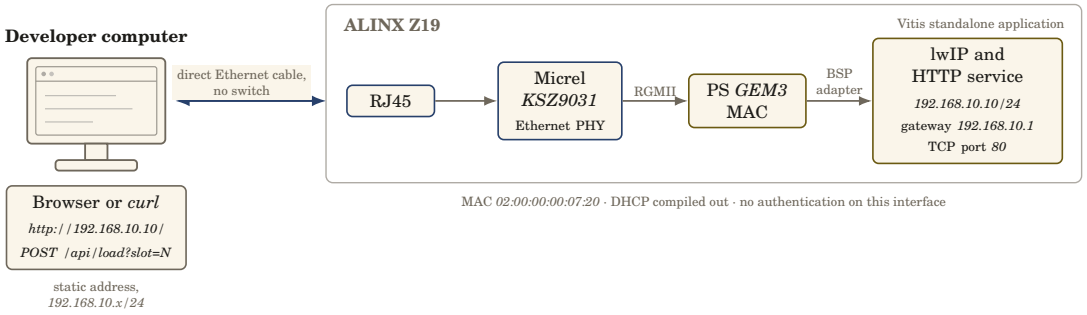


Figure 5.5: *Bench Ethernet topology between the developer computer and the Z19.*

5.8.2 Startup order

Figure 5.6 gives the startup sequence. Two orderings in it are not free choices, and both were established by getting them wrong first.

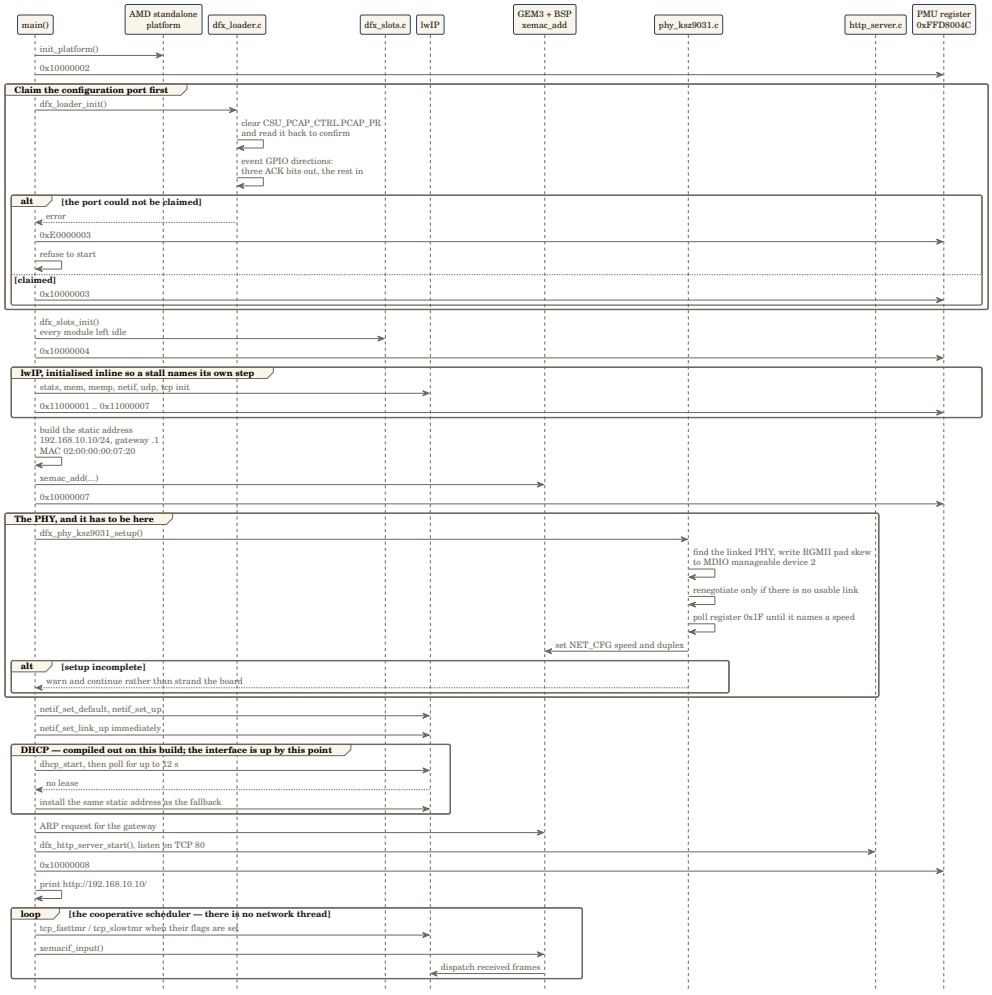


Figure 5.6: *Standalone application startup and network initialization.*

The configuration port is claimed before anything else touches DFX, for the reason in Section 5.5.1. The PHY is configured *after xemac_add()*, not before, because that call software-resets the PHY and would discard the settings; Section 6.7.2 describes what those settings are and why the BSP does not supply them. The interface is also

marked link-up immediately rather than waiting for *eth_link_detect* from a timer callback: under the 2025.2 standalone timer that first callback may never arrive, which leaves the interface administratively up but unable to transmit even an ARP frame. Startup then ARPs the configured gateway, which costs nothing and gives a cheap check that the path works, because a healthy LAN answers before any browser connects. Startup also writes a progress code to a PMU general-storage register at each step, which is readable over JTAG and was the only diagnostic available on the first boots, when the console had not yet been shown to work.

5.8.3 The HTTP interface

The service implements a small subset of HTTP/1.1: one request per connection, two methods, fixed routes, and length-delimited bodies with no chunked encoding [42, 43]. Table 5.6 lists what it exposes.

Table 5.6: *HTTP interface exposed by the standalone application.*

Route	Effect
GET /	The console page, served from a string constant in the binary
GET /api/status	Every partition's VSM state and its module's decoded control and status words
GET /api/system	Facts that do not change while the board runs: platform name, device IDCODE, and per-slot frame address, register addresses, staging addresses and capacity
GET /api/identify?slot=N	Puts the module into identify mode, reads its identifier and version, and restores the previous control word
POST /api/ctrl?slot=N& enable=&mode=&param=	Drives one module's control word
POST /api/vsm?slot=N& cmd=	Shuts down or restarts one partition's VSM
POST /api/run?slot=N& mode=&param=	Streams the request body through the slot's DMA into the module and returns what came back, as raw little-endian words
POST /api/load?slot=N	Receives one partial bitstream and loads it

Two design decisions in this list are worth a sentence each. */api/system* exists so that the console describes the hardware that is actually running rather than the hardware it was written against: the page fetches it once at load and labels itself from the answer, so an address map change in Vivado shows up in the browser without anybody editing HTML. And */api/run* returns raw words rather than a JSON array because a module built against this boundary answers with a vector, not a scalar; formatting four thousand numbers as text on this processor would cost more than the transfer, and truncating the array to keep the reply small would hide

part of the answer. The counts travel in *X-DFX-Sent* and *X-DFX-Received* response headers instead.

Request bodies are never buffered in the heap. Both binary endpoints validate the slot and the advertised *Content-Length*, then switch the connection into a mode that copies each arriving TCP fragment straight to that slot's DDR staging window, so a multi-megabyte partial bitstream needs no allocation larger than the connection object. Figure 5.7 traces the read-only path, which is the simplest illustration of why explicit TCP handling is still needed with lwIP underneath: a request can arrive across several receive callbacks and the parser has to hold bounded state until it sees the blank line ending the header. The reply side needs the same treatment for the opposite reason: a response larger than the TCP send buffer cannot be queued in one call, so the connection remembers how far it has got and resumes from the sent callback as the client acknowledges what it already has.

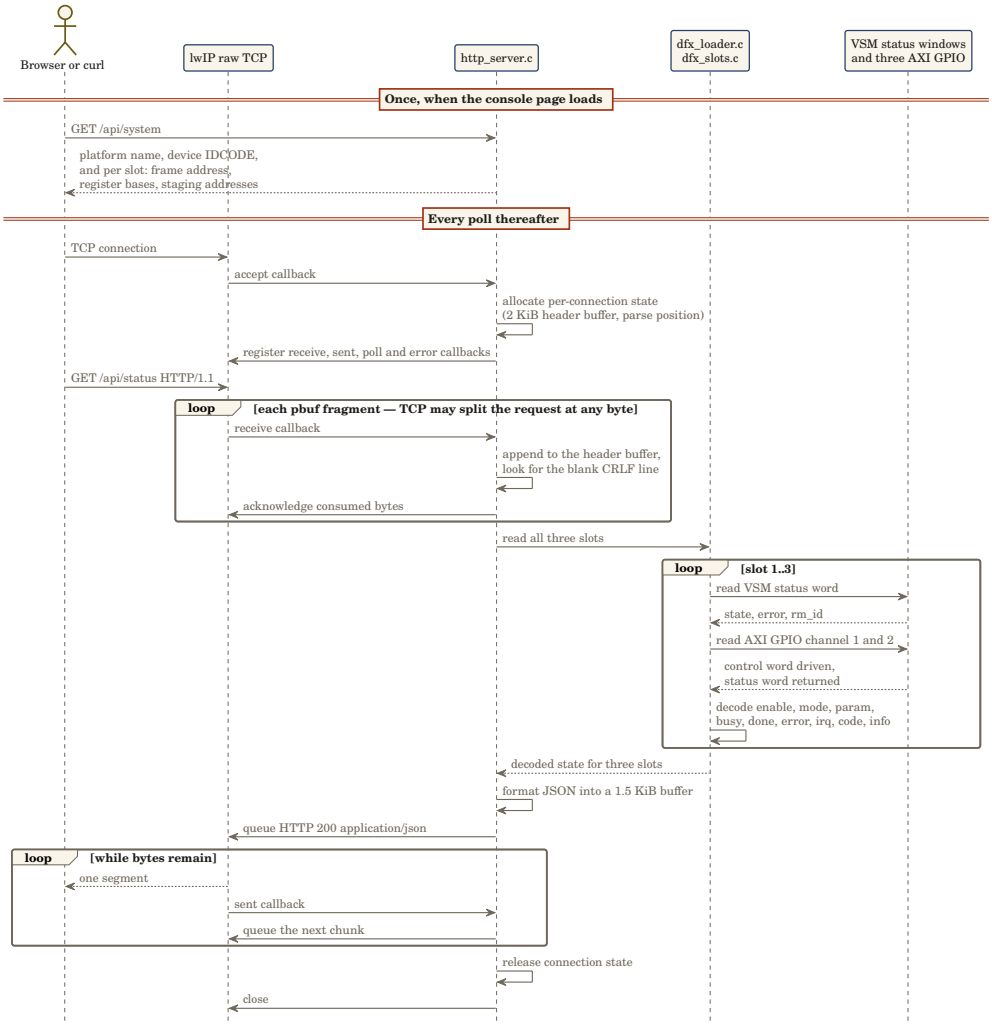


Figure 5.7: *Callback sequence for a status request.*

5.8.4 The console

The page served by *GET* / is one card per partition. Each card shows the module's name, the VSM state, the module's current output value, and buttons to start it, stop it, empty the partition and load a new module. A *.bin* file dropped onto a card is uploaded to that slot. A timer polls */api/status* and, for a running module, pushes a short burst of words through */api/run* so the displayed value tracks what the module is doing.

The naming comes from the identify mode described in Section 5.3: the page holds a small catalogue mapping slot and module identifier to a display name, asks each module who it is after a load, and shows "Unnamed module" for anything not in the catalogue. That is what the reference modules report, since they answer identify with zero.

5.8.5 Loading a partial bitstream

Figure 5.8 follows one upload from the first TCP fragment to a reconfigured partition. The division of labour is the point of the figure: software stages the bytes in DDR and drives a register handshake, and the DFX Controller, not the AXI DMA, reads the configuration data afterwards.

Validation before anything is programmed. Nothing inside a partial bitstream says which partition it belongs to. An operator can upload a perfectly valid file to the wrong slot, and the configuration port will accept it and reconfigure whichever region the file's frame addresses point at, leaving the software convinced it had programmed a different one. What a partial does carry is the frame address at which it starts writing, and that address is fixed by the floorplan: every module built for one partition starts at the same frame, and the three partitions start at different ones. The application therefore walks the uploaded image's configuration packets (it reads packet headers only and never interprets frame data), extracts the device IDCODE and the first real frame address, and rejects the upload if either does not match. The three frame addresses are recorded per slot alongside the expected IDCODE, with a comment stating plainly that they are floorplan-derived and must be read again if a partition moves.

The rejections are phrased for the person who uploaded the file rather than for the person who wrote the loader: "not a configuration bitstream", "built for a different device", "incomplete image: no frame address found", and "built for a different slot". The last of these also reports which slot the image does belong to. A *force=1* query parameter bypasses the check, which exists so that an image the reader cannot recognise can still be pushed at the hardware during experiments. It disables the only thing standing between a wrong file and the wrong partition, and is documented as such.

Byte order. The DFX Controller does not consume file bytes; it consumes the 32 bit words its AXI master reads from DDR. AMD's hardware check requires those reads to yield *000000BB*, *11220044*, *AA995566* [17]. A raw *.bin* from *write_bitstream*, copied

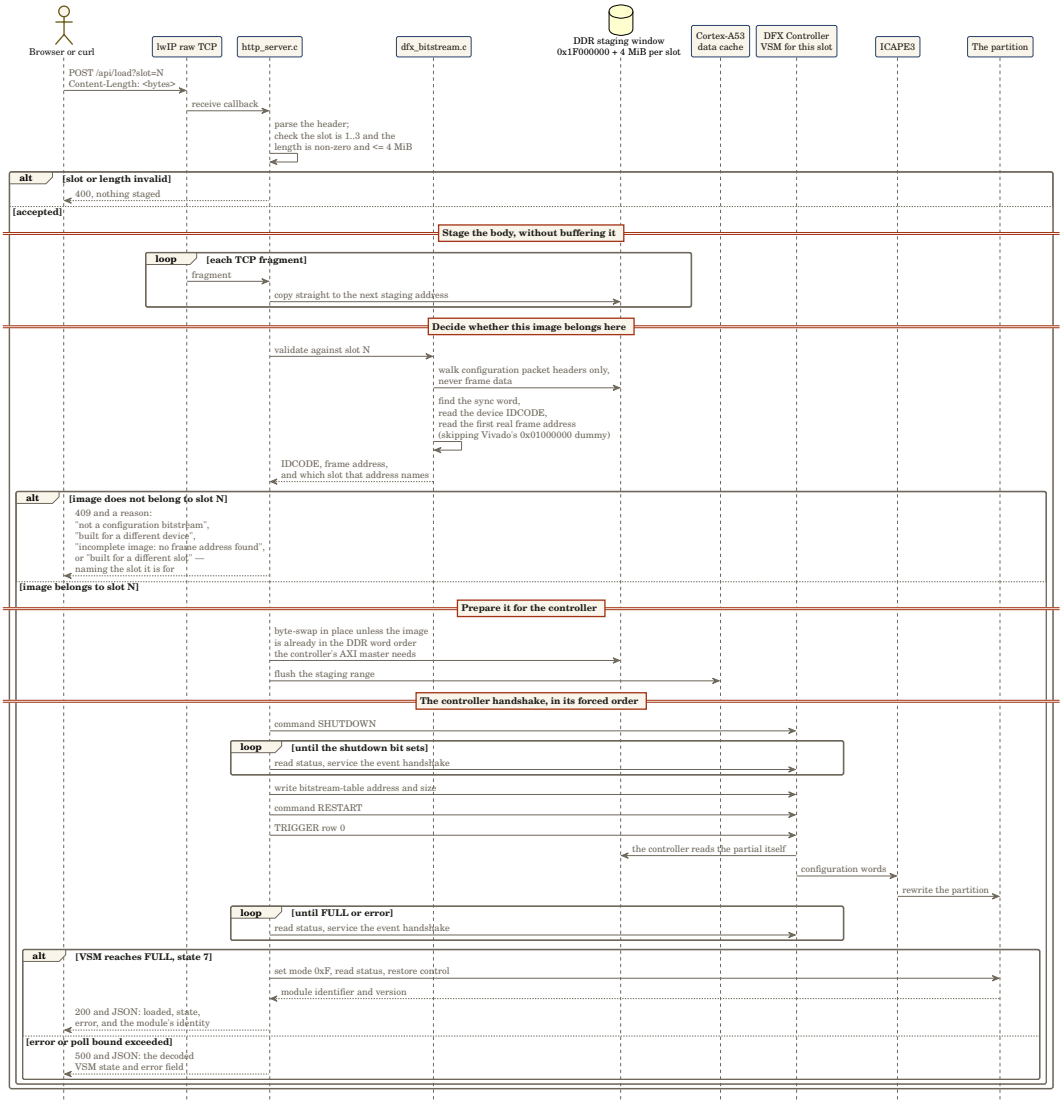


Figure 5.8: *HTTP upload and partial-reconfiguration sequence.*

byte-for-byte into little-endian DDR by an HTTP upload, presents them as *BB000000*, *44002211*, *665599AA*: the same bytes, the wrong words. AMD's own answer is a separate formatting step before the file leaves the build machine. This platform does the conversion on the board instead, in place, immediately after validation and before the controller is given the address. The reason is not elegance: it means an ordinary *write_bitstream* output is directly uploadable, so the developer kit does not have to ship a formatting tool and an author cannot forget to run one. An already-formatted image is detected during validation and passed through unmodified.

The controller handshake. The sequence is shutdown, program the bitstream table, restart, trigger, and the order is not interchangeable. The bitstream-table registers are writable only while the VSM is shut down, and the VSM ignores triggers until it has been restarted: a trigger issued from the shutdown state is silently never acted on, leaving the controller sitting with no error set and a load that never completes. Between the two, software flushes the CPU data cache over

the staging window, because the uploaded bytes are in the cache and the controller is a separate bus master that will read DDR. Software then polls the VSM status register for the full state and returns the decoded VSM state, the controller's error field and the identity of whatever module is now resident. The only handshake the controller waits on during that time is the module's *shutdown_ack*, which is hardware; no software acknowledgement is involved.

An earlier version of this sequence alternated between two module rows in the bitstream table, so that a fresh upload could not be mistaken for the resident module. That requires a controller with two rows actually defined; this one declares a single *RM_0* per VSM, and triggering an undefined row makes the controller reject the load outright. The shutdown and restart already return the partition to its empty state, so row 0 reloads from the freshly written address each time.

5.9 The developer kit

Everything above exists so that this section can be short. To write an accelerator for this platform a person needs Vivado, one abstract shell, one VHDL file, and a network route to the board. They do not need the parent project, the parent bitstream, or any knowledge of the floorplan beyond which slot they are targeting.

5.9.1 One template

Because all three partitions present the same boundary (Section 5.3), there is one template, not three. Listing 5.3 shows it with the housekeeping removed. The entity is frozen: an author renames it, adds generics, and changes everything inside, but does not add, remove or rename a port. The control and status words are unpacked and packed by the same field definitions the software uses.

```
entity slot_template is
    generic (
        -- Clock cycles between steps.
        TICK_DIVIDE : positive := 100000000;
        -- Identity and packed version reported in IDENTIFY mode.
        RM_ID       : natural  := 0;
        RM_VERSION  : natural  := 16#1000#
    );
    port (
        clk    : in std_logic;
        resetn : in std_logic;

        -- control : enable (0), mode (7 downto 4), param (31 downto 16)
        control : in  std_logic_vector(31 downto 0);
        -- status : busy (0), done (1), error (2), irq (3),
        --          code (11 downto 4), info (27 downto 12)
        status  : out std_logic_vector(31 downto 0);

        -- The controller asks the module to go quiet before it is swapped out.
        shutdown_req : in  std_logic;
        shutdown_ack : out std_logic;

        -- Words sent from host.
        s_axis_in_tdata  : in  std_logic_vector(31 downto 0);
        s_axis_in_tkeep  : in  std_logic_vector(3 downto 0);
        s_axis_in_tlast  : in  std_logic;
        s_axis_in_tvalid : in  std_logic;
        s_axis_in_tready : out std_logic;

        -- Words sent back to host.
        m_axis_out_tdata  : out std_logic_vector(31 downto 0);
        m_axis_out_tkeep  : out std_logic_vector(3 downto 0);
        m_axis_out_tlast  : out std_logic;
        m_axis_out_tvalid : out std_logic;
        m_axis_out_tready : in  std_logic
    );
end entity slot_template;
```

Listing 5.3: *The module template, common to all three partitions.*

5.9.2 Three rules

The kit documents three failure modes that are easy to hit and hard to diagnose, because each of them produces a board that looks healthy.

Do not gate timekeeping on *enable*. The host raises *enable* only for the few microseconds of a transfer and lowers it afterwards. A counter gated on it advances a handful of cycles per read and looks frozen. Free-running logic should run from *resetn* and its own tick; *enable* is still worth reporting back as *status.busy*.

Emit one output beat per input beat, and pass *TLAST* through. *TLAST* is the packet boundary the receiving DMA uses to end its transfer. Swallow it and the run hangs until its timeout. Produce extra beats and the channel reports an internal error instead of the module's data.

Acknowledge *shutdown_req*. The controller will not swap a module out without it, and a load that is waiting on a missing acknowledgement looks identical to one that has failed.

5.9.3 Building a module

One script does the build, in three steps that correspond exactly to the three things a partial bitstream needs to be valid.

- **Synthesize the module out of context.** Out of context matters: the ports face the static design across a partition boundary, not device pins, and must not acquire I/O buffers.
- **Open the selected slot's abstract shell and link the netlist into it.** This is where the module acquires the parent's partition pins and surrounding routing, and it is why the result is valid for that one slot and that one parent implementation. A port-list mismatch fails here, with a message naming the slot.
- **Write the bitstream for that cell only.** *-cell* restricts the output to the partition; *-bin_file* produces the raw *.bin* the loader consumes.

The script refuses to write a partial that missed timing. Once a partial is on the board there is no way to tell that it was marginal, and a module that works on the bench and fails in a rack is worse than one that never built.

Listing 5.4 shows a complete session: build the worked example for slot 1, upload it, and ask what is resident.

```
vivado -mode batch -source rm_kit/build_rm.tcl -tclargs \
    -slot 1 -vhdl rm_kit/examples/squares_rm.vhd \
    -top squares_rm -name squares

curl.exe --data-binary @rm_kit/export/rp1_squares_partial.bin \
     -H "Content-Type: application/octet-stream" \
     "http://192.168.10.10/api/load?slot=1"

curl.exe -s "http://192.168.10.10/api/identify?slot=1"
```

Listing 5.4: *Building a module and loading it into slot 1.*

The same source builds for another slot by changing *-slot*, which selects a different shell and produces a different file. Uploading one to the wrong slot is refused, and the loader decides that from the frame address inside the image rather than from the filename. Loading a module into one partition does not disturb the modules resident in the other two: they keep running throughout the exchange, which was checked on the board and is recorded in Table 6.5.

Figure 5.9 places this alongside the platform build, showing which artifacts cross between them and which never leave the platform side.

5.9.4 HDL-language support

The boundary is not inherently VHDL-specific. Vivado synthesis accepts VHDL, Verilog, SystemVerilog and mixed-language source sets [44], and a top level written in another language describing the same ports would be equally valid. The build script as written takes VHDL files and marks them as such, so the tested path is the supplied VHDL template. Wrapping a Verilog or SystemVerilog core inside a small VHDL entity that carries the boundary works today and requires no change to

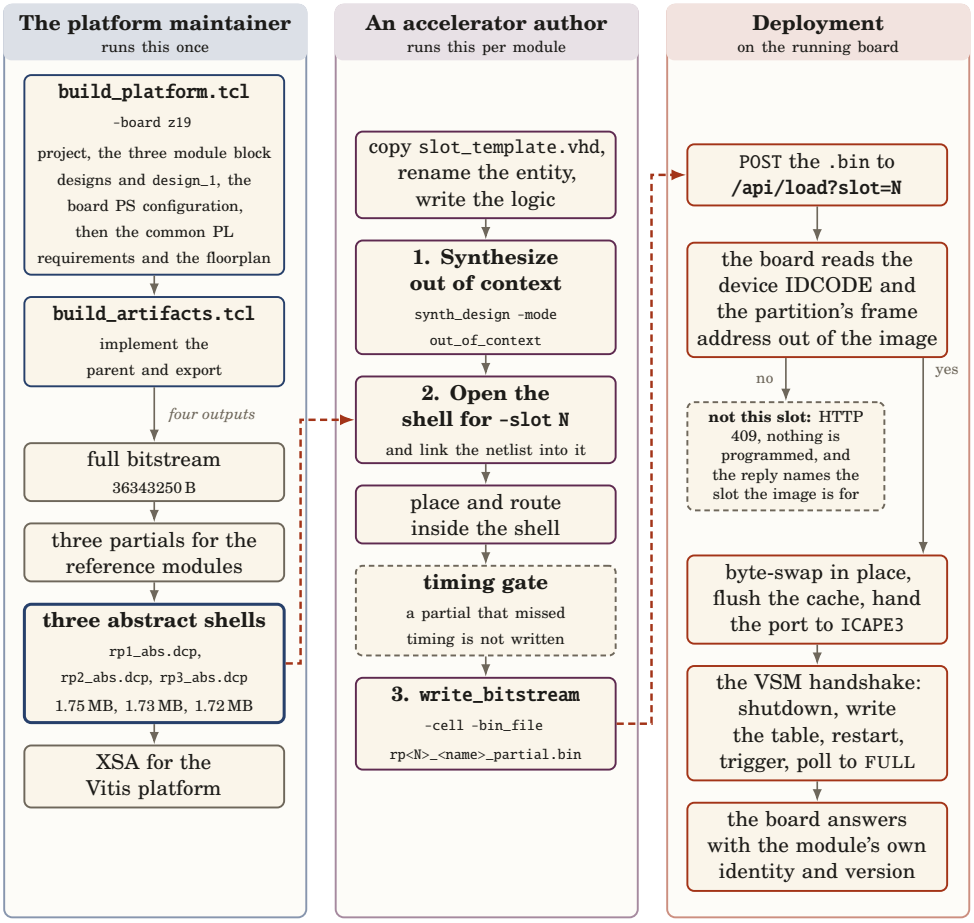


Figure 5.9: *What the platform build produces and what a module author consumes.*

the script. Full language-neutral support would mean accepting mixed file types, inferring or declaring each one, and comparing the elaborated ports against the boundary before implementation. That is a change to how a module is authored, not to the partition, the shell or the loader.

There is likewise no automated path for importing HLS-generated RTL or packaged IP. Chapter 7 discusses what that would involve; it was not built.

6

Results

This chapter reports what the platform build produced and what happened when it was run on the board. The two halves are kept apart on purpose. Vivado reports and generated files establish that the design is legal, closes timing and yields the artifacts the workflow needs; they say nothing about whether the thing works. The second half is the part that was missing when this work was planned, and it turned out to contain the four findings that cost the most time.

6.1 Implementation and timing

Vivado 2025.2 implemented the design for the *xczu19eg-ffvc1760-2-i* against the −2 PRODUCTION speed file. Table 6.1 gives the routed timing summary. All user-specified constraints are met, in all three checks, with no failing endpoints.

Table 6.1: *Routed timing summary for the implemented design.*

Check	Worst slack	Total slack	Failing	Endpoints
Setup	4.342 ns	0.000 ns	0	60611
Hold	0.013 ns	0.000 ns	0	60611
Pulse width	2.881 ns	0.000 ns	0	20810

The setup margin is comfortable, and it should be: the clock constraint is the requested 10.000 ns period while the PS actually delivers 96.97 MHz, so the design is analysed against a clock marginally faster than the one it runs on. Hold slack of 13 ps looks alarming next to it and is not. Hold closure is a routing-delay question rather than a frequency one, and the number to check is that nothing fails, which nothing does.

Routing completed with no errors. Of 42880 logical nets, 23923 needed routing and all of them were routed; the remainder are internal to a site or have no loads. The routed design-rule check reports no violations of any DFX rule. What it does report is six *REQP-1935* block-RAM collision advisories, one *RTSTAT-10* for a net with no routable load, and three *AVAL-156* DSP advisories. All of them are warnings or advisories that come from generated IP, and none is specific to the reconfigurable regions. An earlier version of this design produced HDPR errors at every attempt, and their absence here is the meaningful result, not the advisories.

6.2 Resource utilization

Table 6.2 gives the placed utilization of the whole design: the static shell and all three resident reference modules together.

Table 6.2: *Placed resource utilization of the complete implemented design.*

Resource	Used	Available	Utilization
CLB LUTs	12046	521376	2.31 %
as logic	10166		
as memory	1880		
CLB registers	17267	1042752	1.66 %
CLBs occupied	2727	65172	4.18 %
CARRY8	80	65172	0.12 %
Block RAM tiles	6	984	0.61 %
UltraRAM	0	128	0.00 %
DSP48E2	3	1968	0.15 %
ICAPE3	1	2	50.00 %
PS8	1	1	100.00 %
BUFG_PS	1	72	1.39 %
Bonded IOB	0	512	0.00 %

Three rows deserve comment. All six block-RAM tiles are *RAMB36E2*; no *RAMB18* and no UltraRAM are used at all, which is worth recording because column X0 of the die holds every one of the device's 128 UltraRAMs and the floorplan gives that column to the static design. The three DSP48E2 are the reference modules' status counters, one each; Vivado implements the 32 bit counter in a DSP slice rather than in fabric. And no bonded I/O is used anywhere: nothing in the programmable logic reaches a package pin in this design, because every external interface belongs to the processing system.

The headline percentages are the least interesting numbers in the table. A module loaded into this platform is bounded by the resources inside its own partition, not by what is free elsewhere on the die, and Table 5.4 is the budget that actually applies to it. Device-wide utilization says only that the static shell is small, which it is.

6.3 Generated artifacts

The export step produced one full bitstream, one partial per partition in both formats, one abstract shell per partition, and the XSA. Table 6.3 gives their sizes.

Table 6.3: *Generated configuration artifacts and developer-kit checkpoints.*

Artifact	**Region**	***.bit*** (B)	***.bin*** (B)
Full device image	whole PL	36343250	n/a
rp1 partial	rp1	2594715	2594564
rp2 partial	rp2	3191631	3191480
rp3 partial	rp3	2754951	2754800
rp1_abs.dcp	rp1		1751660
rp2_abs.dcp	rp2		1728520
rp3_abs.dcp	rp3		1718269

The *.bit* and *.bin* forms of a partial differ by exactly 151 bytes, which is the header the *.bit* carries and the raw form does not. The loader takes the *.bin*.

The ordering of the three partial sizes is the result worth reading. All three reference modules are the same trivial pass-through, so if partial size tracked module content they would be identical. They are not: rp2 is the largest at 3.19 MB, rp3 next at 2.75 MB, rp1 smallest at 2.59 MB. That is exactly the order of their SLICE capacities in Table 5.4 (6480, 5040, 3600), and it confirms Equation 5.4.2.1 on the built design. A partial covers the whole partition regardless of how much of it the module occupies. An author who wants a smaller file has to be given a smaller partition; writing less logic will not do it.

The abstract shells are the other number the workflow depends on. At roughly 1.75 MB each they are small enough to keep in a repository and hand around, and that is what makes the developer kit in Section 5.9 viable at all. Their sizes are close together because the parent's routing near the three boundaries is comparable, and they are unrelated to partition capacity: a shell holds context, not fabric.

6.4 Configuration-transfer estimate

The ICAPE3 data port is 32 bit wide [18] and the generated hardware handoff records the PL0 frequency as 96968727 Hz. One 32 bit word per cycle gives an ideal payload rate of

$$R_{\mathrm{ICAP}} = 4\,\mathrm{B\,cycle^{-1}} \times 96.97 \times 10^{6}\,\mathrm{cycle\,s^{-1}} = 387.87\,\mathrm{MB\,s^{-1}}. \tag{6.4.1}$$

Dividing each raw binary by that rate gives Table 6.4.

Table 6.4: *Ideal configuration-payload transfer times at* 387.87 MB s^{-1}.

Artifact	Size (B)	Ideal time	Reduction
Full device image	36343250	93.70 ms	n/a
rp1 partial	2594564	6.69 ms	92.86 %
rp2 partial	3191480	8.23 ms	91.22 %
rp3 partial	2754800	7.10 ms	92.42 %

Loading one partition instead of the whole device moves about an order of magnitude less configuration data, and by this measure costs roughly a hundredth of a second rather than a tenth. Take the absolute numbers as a floor and nothing more. They count payload bytes divided by port width times clock, and exclude the controller's own state transitions, DDR arbitration behind the controller's AXI master, the cache maintenance software performs before handing over the buffer, the polling loop that waits for completion, and the network transfer that put the file in DDR in the first place. On this platform the network transfer alone is far larger than any of these figures. What the table supports is the comparison between rows, not the values in them.

6.5 Byte order at the configuration port

The DFX Controller does not consume file bytes. It consumes the 32 bit words its AXI master reads from DDR, and AMD's hardware check requires those reads to yield *000000BB*, *11220044*, *AA995566* [17]. A raw *.bin* from *write_bitstream*, copied byte-for-byte into little-endian DDR by an HTTP upload, presents the same bytes as *BB000000*, *44002211*, *665599AA*. The controller sees no sync word and the load never starts.

AMD's answer is a separate formatting pass before the file leaves the build machine. This platform does the conversion on the board instead, shown in Figure 6.1: after validation, before the controller is given an address, the staged image is swapped in place. The reason is practical rather than aesthetic. An ordinary *write_bitstream* output becomes directly uploadable, the developer kit does not have to ship a formatting tool, and there is no step an author can forget to run. An image that already arrives in the required layout is recognised during validation and passed through untouched, so both forms work.

6.6 Software build

The exported XSA generated a standalone platform and board support package for the Cortex-A53, against which the application, the lwIP stack and the generated drivers compile. The result is *z19_dfx_app.elf*: 212192 B of text, 9688 B of initialised

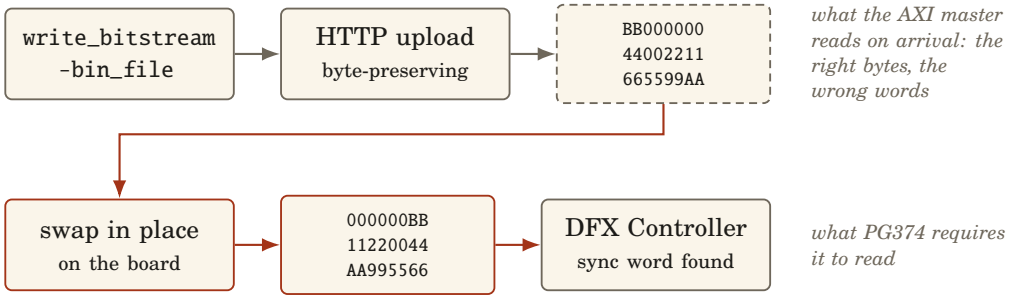


Figure 6.1: *Byte order on the path from* write_bitstream *to ICAPE3. The uploaded file carries the right bytes in the wrong words; the board corrects them in place rather than requiring a formatting step off-line.*

data and 6312792 B of *bss*, the last of which is dominated by lwIP's static buffer pools rather than by anything this application allocates.

That establishes consistency between the application sources, the BSP, lwIP and the exported hardware description. It establishes nothing about behavior, which is the subject of the rest of this chapter.

Listing 6.1 gives one partition's entry in a real */api/status* response.

```
{"slots":[
  {"slot":1,"capacity":4096,
   "vsm":{"state":7,"error":0,"rm_id":0},
   "rm":{"enable":0,"mode":0,"param":0,"busy":0,"done":0,
         "error":0,"irq":0,"code":0,"info":0,
         "raw":0,"raw_ctrl":0}},
  ...
]}
```

Listing 6.1: *One slot's entry in the status response.*

State is decoded from the VSM status register's bits [2:0], and 7 is the controller's full state, meaning a module is loaded. *error* comes from bits [6:3] and *rm_id* from bits [23:8] [17]. The *rm* object is the decoded control and status word from that slot's AXI GPIO, so a client can see both what the partition contains and what the module inside it is reporting. *Capacity* is the largest run the slot accepts, in words, and is the same for all three because the three datapaths are the same.

6.7 Bringing the platform up on hardware

Everything above was true before the board was ever powered on, and none of it predicted what happened when it was. Four findings account for almost all of the bring-up effort. Three of them have a shape in common: in each case the system reported success or reported nothing, while doing the wrong thing, and none produced an error message, which is why each took far longer than its eventual one-line fix suggests. The fourth did report a failure, but in the wrong subsystem.

6.7.1 PCAP owns the configuration port

The first reconfiguration attempt hung. The DFX Controller accepted the trigger, moved into its loading state, and stayed there. No error bit was set, no timeout fired in hardware, and the status register kept reporting a load in progress for as long as anyone cared to poll it.

The cause is one bit outside the programmable logic and outside the block design entirely. On Zynq UltraScale+, *CSU_PCAP_CTRL.PCAP_PR* at *0xFFCA3008* selects which port performs partial reconfiguration, and it comes out of reset selecting PCAP, the port the boot ROM uses. The requirement itself is documented, if not prominently: AMD's own ICAP driver notes that on this device the *PCAP_PR* bit must be cleared before the ICAP can be accessed at all [45]. What is documented nowhere is the failure mode when it is not. On this board *ICAPE3* simply never asserted its *avail* output, and the controller was therefore behaving correctly: it was waiting for a configuration port to become free, that port was never going to become free, and nothing in its interface exists to say so.

The fix is to clear the bit during startup, before the service accepts any request, and to read the register back to confirm the write stuck. Every other bit is preserved, and by this point the full bitstream has already been programmed over JTAG so no PCAP transfer is in flight to interrupt. Section 5.5.1 describes where this sits in the startup order.

Nothing in the Vivado flow warns about this, because from Vivado's point of view the design is complete and correct. It is a property of the device that only appears at run time.

6.7.2 The Ethernet PHY needed code of its own

The board came up with a working link and no traffic. The PHY and the host agreed the link was gigabit and full duplex, autonegotiation reported complete, the MAC was enabled. Not one frame crossed RGMII in either direction. The controller reported zero receives *and* zero receive errors, which is the signature that made this hard: a broken cable produces errors, a wrong address produces errors, and a mis-timed source-synchronous bus produces silence, because from the controller's side nothing arrived to be wrong about.

The cause is split across two vendors' documentation and stated in neither. The BSP's speed-detection code recognises four PHY identifiers, Marvell, TI, Realtek and ADI, and routes anything else to its Marvell handler [46]. This board's PHY is a Micrel KSZ9031, whose identifier register reads *0x0022* [47], so it takes that path. The Marvell handler selects register page 2, sets an RGMII delay bit there, and software-resets the PHY on the way past [46]. The KSZ9031 has no such page-selected register: its RGMII skew lives in MDIO manageable device 2 [47], which that sequence never touches. Both halves are published. Their combination is not, and what it produces is a PHY left on its power-on skew defaults, a link that negotiates 1000 full, and no traffic.

The correction is four register writes into MDIO manageable device 2, whose pad-skew registers are 4h for the control signals, 5h and 6h for receive and transmit data, and 8h for the clocks [47]: control *0x0070*, RXD *0x7777*, TXD *0x0000*, clock *0x03F9*. These values were not derived. They were read back over MDIO from the board's own running Linux image, and three of the four differ from the datasheet's power-on defaults, which is the concrete sense in which the defaults were wrong here. With them the interface passes traffic. The likely reason is that they compensate for the trace lengths between the PS RGMII pins and the PHY, which would make them a property of this carrier rather than of this program; no trace was measured to confirm that, but it is the reason to treat them as board data and to expect another carrier to need values of its own.

Two ordering constraints came out of the same investigation, and Figure 6.2 shows both alongside the sequence above.

The skew setup has to run *after xemac_add()*, because that call software-resets the PHY and would discard anything written before it. It must also not restart autonegotiation when a usable link already exists. Restarting it drops the link underneath a live MAC, and the receiver does not reliably come back. That produced the worst symptom of the whole exercise: an intermittent startup in which the link came up, autonegotiation completed, the MAC was armed, and no frame was received. It cleared on the next programming run, so it looked like a hardware flake for some time before it turned out to be an ordering mistake.

6.7.3 The negotiated speed lags the link

With the skew right, the board still passed no traffic, and this time the console said why: "link up at 10 Mbps half duplex", on a link that both the PHY and the host agreed was 1000 full.

KSZ9031 register *0x1F* is the PHY Control register: bits 6, 5 and 4 report the final speed status at 1000BASE-T, 100BASE-TX and 10BASE-T, and bit 3 the duplex [47]. What the datasheet does not say is how long after link-up those bits take to settle. Read at the moment the link comes up, the register returns a word with none of them set, which falls through the decoder to its slowest option. The MAC was configured for 10 Mbps half duplex against a gigabit link. That combination passes nothing and reports no error, because as far as the controller is concerned it is running exactly as it was told. The hardware confirmed it directly: *NET_CFG* read back *0x013F20C0*, with the gigabit and full-duplex bits both clear, while the PHY read *0x034C*.

The fix is to poll register *0x1F* until it names a speed. The bound on that poll is short on purpose. The link already exists by this point and the PHY only has to latch its result, so a generous bound would buy nothing and would hide a PHY that never reports at all.

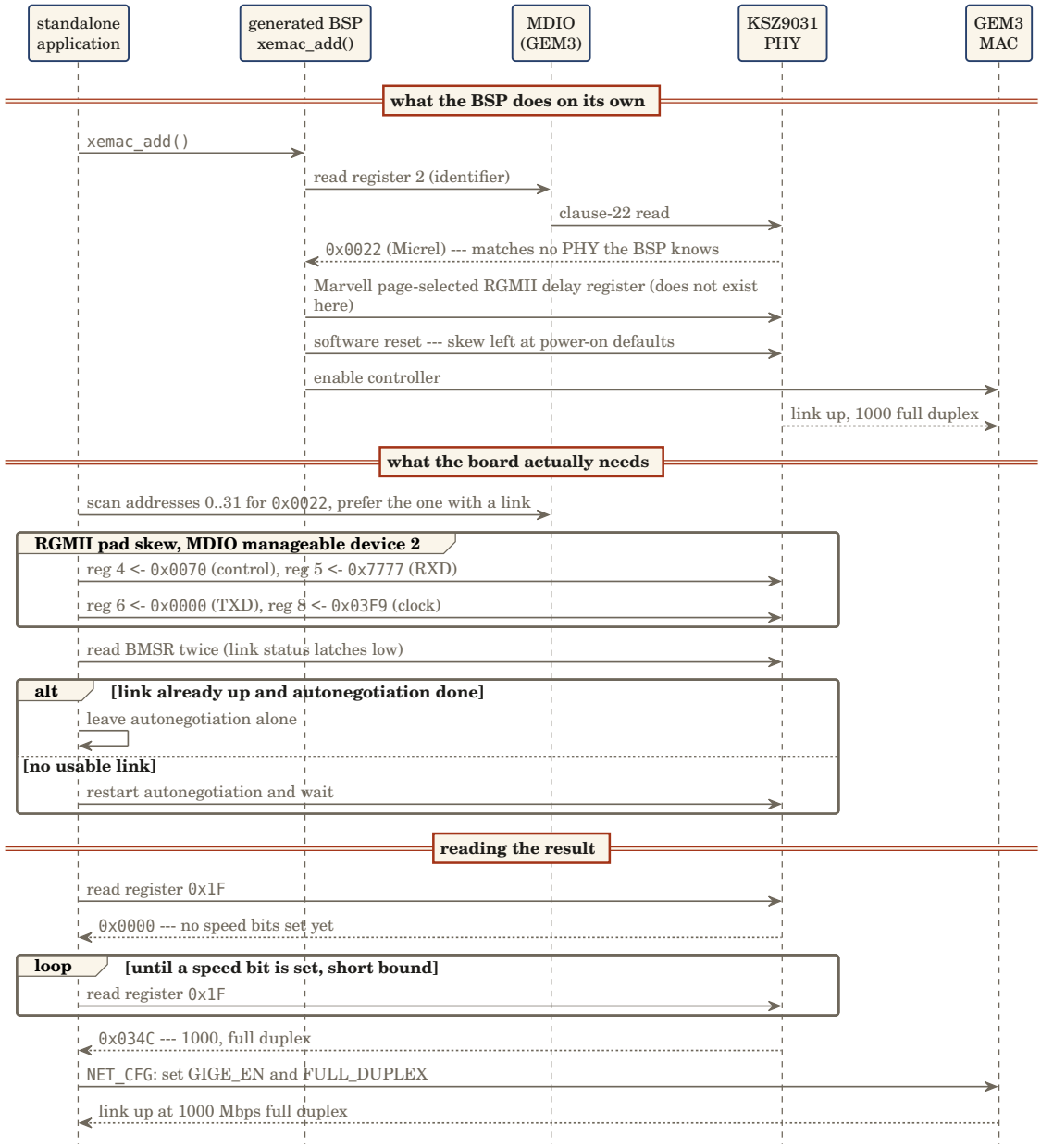


Figure 6.2: *Ethernet bring-up: what the BSP does with an unrecognised PHY, and the sequence the board actually needs.*

6.7.4 Transceivers wedge the debug port

The fourth finding cost the most time per line of eventual fix, and it happens before any of the others: it prevents the board being programmed at all.

The symptom is that *targets* reports no PSU and no Cortex-A53. Only the JTAG chain itself enumerates. That reads as a board which is unpowered or held in reset, and it sends you to the cables and the boot-mode switches, because a device that answers JTAG but shows no processor looks disconnected rather than broken. It is neither: the debug access port is in an AXI error state, and software cannot clear

it. A system reset issued from the PMU does not help, nor does an *srst* attempt, nor restarting *hw_server*. Only a power cycle recovers it.

The cause is upstream of the debug port. ALINX's factory configuration enables several gigabit-transceiver peripherals that this platform never uses, and with them enabled *psu_init* fails part way through its SERDES programming, at

mask_write 0XFE20C200 0x00023FFF 0x00022457

A *psu_init* that fails leaves the port wedged, so the failure presents one step later than it occurs and in a subsystem that has nothing to do with the cause.

The fix is to not program the transceivers. PCIe, DisplayPort, both USB controllers, both USB3 controllers and SATA are switched off in the board configuration described in Section 5.1, which removes the whole SERDES bring-up. Nothing in this design drives any of them. The launcher also now tests for the condition and says what it is rather than reporting no targets, so the next person to meet it is told to power-cycle instead of checking cables.

This one differs from the other three in an instructive way. Those were silent: the system reported success, or reported nothing, while doing the wrong thing. This one reports a failure, but reports it in the wrong place, which is the other way a bug becomes expensive.

6.7.5 Final status of the board implementation

Following those four corrections, the platform operated unattended during hardware testing. Table 6.5 lists the checks performed on hardware and their outcomes.

Figure 6.3 traces one load as it runs, with the console output each step produces. The order of the handshake is the part that is easy to get wrong and hard to see. The controller's register interface makes the bitstream-table entries writable only while the virtual socket manager is shut down, and the manager accepts no trigger until it has been restarted [17]. A trigger issued between those two states is therefore silently never acted on, and that is what took the time to find: the failure looks like a load that simply never completes, with no error set anywhere.

6.8 What was not measured

The hardware results above are functional. Nothing in them is timed. Reconfiguration wall-clock time, upload throughput and the length of any service interruption were not instrumented, so the figures in Section 6.4 remain analytical bounds and are not compared against a measurement. Published surveys of configuration controllers report throughputs from under $10\,\mathrm{MB\,s^{-1}}$ for processor-driven ICAP wrappers to roughly $400\,\mathrm{MB\,s^{-1}}$ for DMA-driven ones [22]; this design uses a DMA-capable controller, and only a measurement on the board can place it in that range.

None of the accelerators was evaluated. The reference modules and the worked examples in the developer kit exercise the boundary and the loader. They do no signal processing, and no claim about D-MIMO performance follows from any number in this chapter.

Table 6.5: *Acceptance checks performed on the Z19.*

Check	Observation	Outcome
Unattended startup	The application reaches its serving state without intervention and prints the address it is listening on	Pass
Network reachability	*ping* 4 sent, 4 received, 0.110 ms round trip	Pass
Console served	*GET* / returns the page; */api/system* and */api/status* answer	Pass
Upload path	A multi-megabyte partial streams into its DDR window without a heap allocation of that size	Pass
Image validation	A partial built for another slot is refused, naming the slot it belongs to	Pass
Reconfiguration	All three partitions load a partial over HTTP and reach the controller's full state	Pass
Independence	Reconfiguring one partition leaves the other two running	Pass
Module identity	A loaded module answers identify mode with its own identifier and version	Pass

6.9 Portability of the source package

Every maintained script resolves its inputs from its own location. *build_platform.tcl*, *build_artifacts.tcl* and the developer kit's *build_rm.tcl* derive the repository root from *[info script]*, the Vitis project generator does the same in Python, and the developer kit reads its abstract shells from the export directory that *build_artifacts.tcl* writes them to. A search of the maintained sources for home directories and drive letters finds nothing but the batch files' list of default install locations. The only installation-specific input is the location of Vivado and Vitis themselves: the Vitis wrappers, a shell script and a batch file for each of the two operations, take it from the *XILINX_VITIS* environment variable and fall back to the usual install locations, and Vivado is invoked directly by whichever path the user's environment provides.

The project has been recreated and run on a Windows host as well as on the Linux machine it was written on, using the same Tcl and the same Python through the two sets of wrappers. That is the extent of the portability claim: the flow is not tied to the operating system or the checkout location, but it has been exercised on two machines by one person, and it still assumes Vivado and Vitis 2025.2.

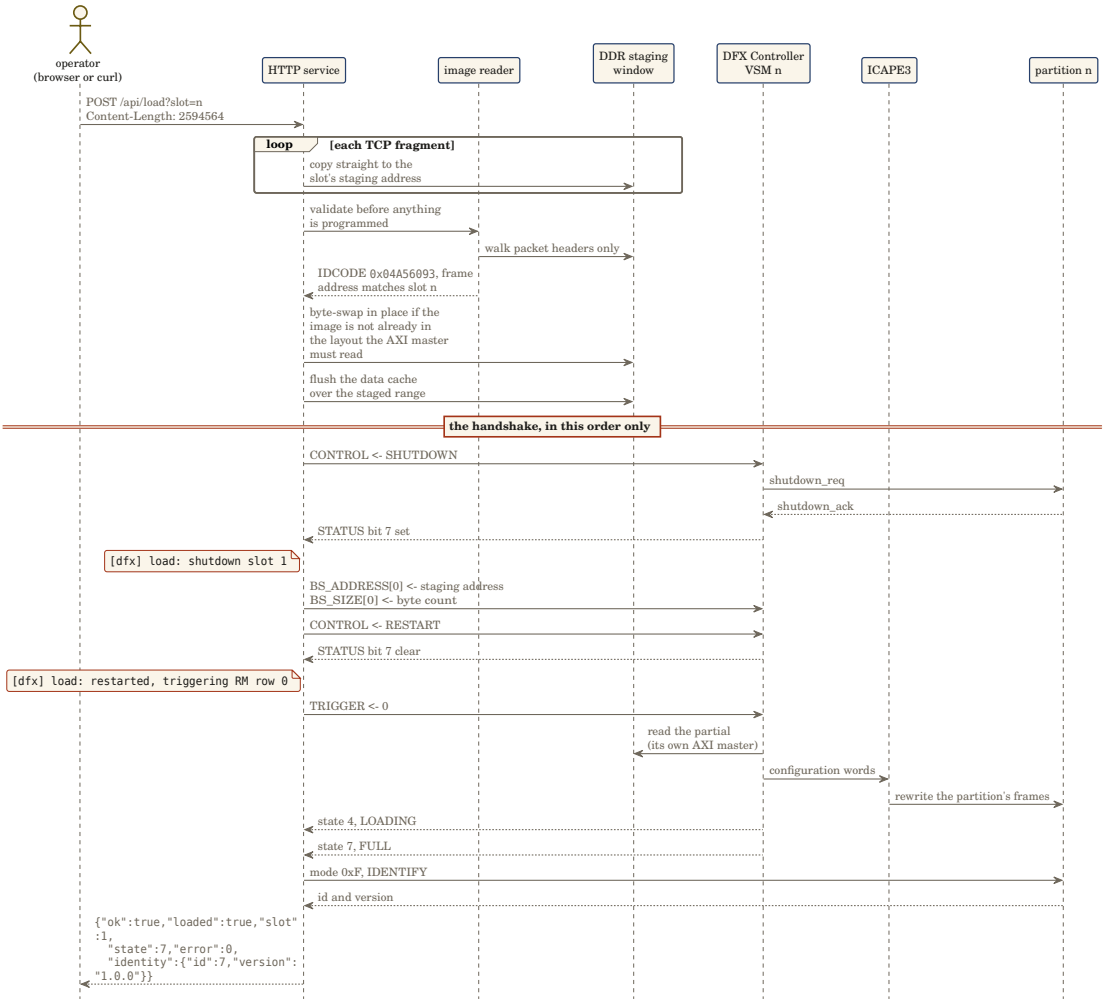


Figure 6.3: *One partial-bitstream load as it runs on the board, from the first TCP fragment to a reconfigured partition.*

7

Discussion

This chapter assesses the result as an FPGA platform architecture. D-MIMO signal-processing algorithms and radio-system performance stay outside the implemented scope.

7.1 One boundary, three partitions

The platform presents a single partition boundary and replicates it three times. Uniformity has a cost, and that cost should be stated. An accelerator that naturally wants a memory-shaped port has to express itself as a stream: a block operation over a buffer becomes a pass of words in and a pass of words out, and anything that would keep persistent state on chip across an exchange has to keep it in DDR instead. An accelerator that would rather not implement *TREADY* has to implement it anyway. Neither is fatal, but both are work the module has to do that a boundary shaped for that access pattern would have done on its behalf.

What the single boundary buys is worth more for this platform. There is one template, so the documented rules an author has to get right are one set. A module source builds into any of the three partitions by changing a command-line argument, which means an author who runs out of DSP sites in one region can move to the region with four times as many without rewriting anything. And spare capacity anywhere on the device is reachable by any accelerator, rather than being reserved for one category of module.

The decisive argument is about what this platform is for. Its accelerators do not exist yet. Any specialization of the boundary is a bet on what those accelerators will want, placed before anyone knows, and a wrong bet is expensive: it is baked into the routed parent, the partition pins and every partial bitstream built against them. A uniform boundary defers that decision to the modules, where it can still be changed.

That reasoning would change if a class of accelerator with a genuinely awkward fit turned up: a large block transform that wants a frame buffer resident across exchanges is the obvious candidate. The first response then is not to specialize a partition but to add the buffer to the static shell and reach it from the stream boundary, which keeps the module interchangeable. Section 8.1 lists specialized sockets as the step after that.

7.2 Relevance to a D-MIMO testbed

D-MIMO experiments involve continuous stream transformations, sample-level operations and buffered block processing. A single stream boundary with a control and status side channel accommodates the first two directly and the third with buffering in DDR, while the processing system, network service, memory paths and reconfiguration infrastructure stay put across every exchange.

The reference modules validate the boundary and the build flow, not D-MIMO algorithms. That separation is the point: the platform developer decides how an accelerator is connected, controlled, isolated and deployed, and the communications researcher decides what it computes. The two roles meet at 146 pins and an abstract shell.

The implemented architecture is one subsystem of a possible testbed. A complete installation would also need radios, fronthaul, timing and frequency synchronization, calibration, experiment orchestration and the domain-specific processing itself.

7.3 The abstract shell makes the platform usable

Of everything in this work, the abstract shell is the piece that most changes who can use the result.

Without it, the instruction to an accelerator author is: clone the repository, install the same Vivado version, rebuild the parent, wait about an hour, and then build your module against your own copy. That works, and it makes every author a platform maintainer. It also invites a specific failure: two people build parents that differ in some small way, both produce partials, and the partials are not interchangeable even though the sources were identical.

With it, the instruction is: take this checkpoint, write one entity, run one script. The shell is roughly 1.75 MB, opens in seconds, and contains one partition's context and nothing else: not the block design, not the processing-system configuration, not the bitstream. The authors' builds are anchored to one specific routed parent by construction, because the shell *is* that parent's context around that partition. Two authors working from the same shell cannot produce mutually incompatible partials.

The cost is coupling, and it is unavoidable rather than incidental. A shell binds a partial to one parent implementation. Move a partition, change the boundary, or reroute the partition pins, and every shell and every partial built from it becomes invalid together. Nothing warns an author of this, because a stale partial is a perfectly well-formed file: it will be refused by the loader only if it lands in the wrong slot, and a partial built against an obsolete parent for the right slot has the same frame address as a current one. Regenerating shells and partials in the same export step, as *build_artifacts.tcl* does, keeps the platform's own artifacts consistent; it does nothing for a copy somebody took last month. A shell version identifier checked at upload time is the obvious fix, and Section 7.8 lists it among the things this demonstrator does not do.

AMD documents abstract shells mainly as an intellectual-property mechanism, for handing out a partition without revealing the static design [6]. That use case does not apply here, because the whole platform is open, but the mechanism turns out to be just as valuable for ordinary engineering reasons, and this work is a straightforward demonstration of that.

7.4 Proposed HLS authoring path

The reference modules and examples are handwritten VHDL. A worthwhile extension would let a developer describe a kernel and its test bench in C or C++ and use Vitis HLS for C simulation, synthesis and C/RTL co-simulation. Vitis HLS generates AXI4-Stream, AXI4-Lite, AXI4 master and block-level control interfaces from a suitably constrained top-level function [21].

HLS changes how candidate logic is authored; it replaces none of the DFX flow. The generated RTL still has to be wrapped to present exactly the 146-pin boundary, synthesized out of context, linked into the target partition's abstract shell, and checked for timing before a partial is written. Figure 7.1 shows the proposed front end against the existing back end.

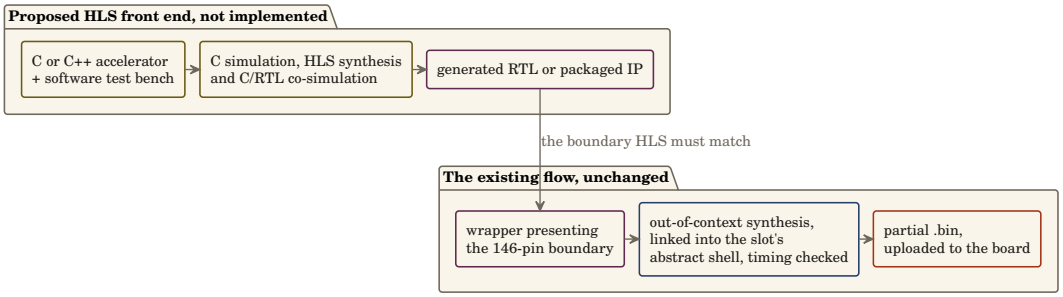


Figure 7.1: *Proposed C/C++-to-module development flow.*

The uniform boundary helps here. There is one wrapper, and it maps onto an HLS stream interface almost directly; a memory-mapped HLS interface would have needed a much fiddlier one. A small FIR filter or complex multiplier would make a suitable first case: testable in C++, synthesizable with a fixed 32 bit stream signature, and comparable against handwritten RTL on latency, initiation interval, resource use, timing and numerical agreement. Extra HLS control ports cannot be introduced without changing the partition interface, so the block-level control has to be carried in the existing control and status words.

7.5 Partitions and modules in the implemented design

A partition is a persistent logical and physical socket; a module is the configuration that occupies it. The three partitions exist concurrently, and only one module can occupy any one of them at a time.

The parent implementation delivers one reference module per partition, which is enough to establish the hierarchy and emit the first three partials. On its own that would show only that a partition can be implemented, not that a module can be exchanged for another one. The developer kit closes that gap: the two worked examples, *squares_rm* and *countdown_rm*, are built from the same template against the abstract shells, and loading either over HTTP replaces the reference module resident in the target partition. Chapter 6 reports that this was done for all three partitions, with the other two staying resident and answering throughout.

What that establishes is functional replacement of one module by another, on hardware, without touching the static design. What it does not establish is anything about accelerators of realistic size. The examples occupy a small fraction of their partitions, so their timing and routing behaviour says little about a module that fills one.

7.6 Wrappers, uniformity and reserved capacity

With a single boundary there are no compatibility classes among modules. A module is not tied to a partition by its interface; it is tied to one only by the shell it was built against, and rebuilding for another slot is one argument.

The static side still does real work at the boundary. Each partition's decoupler holds its interface at safe values while frames are written, and each partition's AXI DMA absorbs the difference between a stream of beats and a DDR buffer. Keeping the buffering and the protocol recovery in the static region means they stay available while a module is reset or absent, which is exactly when they are needed.

The three pblocks reserve much more logic than any module built for them so far requires. That is deliberate capacity for accelerators that do not exist yet, and it has a price: a partial bitstream covers the whole physical region, so the artifacts are between 2.59 MB and 3.19 MB whether the module inside is a pass-through or a filled region. Smaller regions would produce smaller partials and faster reconfiguration, and would also cap what the platform can host. That trade off cannot be evaluated properly until representative accelerators exist to size against.

The regions are not equal, and Chapter 5 explains why equalizing them was rejected. The consequence for an author is that a module fitting rp2 does not necessarily fit rp1, and the build script reports per-partition utilization so that this is visible rather than surprising.

7.7 Interpretation of the results

Positive slack WNS and zero TNS establish that the parent meets its constraints with margin, and the margin is large: over 4 ns on a 10 ns period. Device-wide utilization is low in every category. Neither number says much about a future accelerator, which is constrained by the resources and routing inside its own pblock and needs its own analysis.

The partial sizes track partition geometry, as Equation 5.4.2.1 predicts: the largest partition produces the largest partial, and the reference modules inside them are identical. Using the 96.97 MHz clock recorded in the hardware handoff gives an ideal ICAP payload rate of 387.87 $\mathrm{MB\,s^{-1}}$ and transfer bounds between 6.69 ms and 8.23 ms, against 93.70 ms for the full image.

Those are bounds on the payload transfer and nothing else. They exclude controller sequencing, DDR arbitration, cache maintenance, software polling and the network transfer, and the network transfer is almost certainly the dominant term in the observed wall-clock time for an upload. Published comparisons of configuration controllers show how wide the gap between bound and measurement can be, with reported throughputs ranging from under 10 $\mathrm{MB\,s^{-1}}$ for processor-driven vendor ICAP wrappers to around 400 $\mathrm{MB\,s^{-1}}$ for DMA-driven controllers [22]. This design uses a DMA-capable controller, so the bound is not obviously unreachable, but nothing here measures where it actually falls. That measurement is the most valuable single thing a follow-up could add.

The bench results change the character of the evidence rather than adding a number to it. Before them, the report could say the sequence was implemented as documented; after them, it can say the sequence completes, the controller reaches its full state, the loaded module identifies itself and the other two partitions keep running. What it still cannot say is how long any of that takes.

The bring-up findings in Chapter 6 deserve a comment here, because three of them followed the same failure pattern. In each case the design was correct on paper, the tools reported nothing, and the symptom was silence: a controller waiting on a configuration port it did not own, a link that negotiated and carried no frames, a MAC configured for 10 $\mathrm{Mbit\,s^{-1}}$ on a gigabit link. None of the three would have appeared in simulation, and none produced an error message. The fourth, the wedged debug port, did produce a failure, but one that pointed at cables and boot switches rather than at the processing-system configuration that caused it. A design that is verified only against reports and generated artifacts will pass all of its checks and not work, and that is an argument about evaluation methodology rather than about these specific bugs.

7.8 Remote operation and trust

The HTTP interface lets a developer upload a partial bitstream from a browser or a single *curl* command, without the Vitis command line and without JTAG. That is the point of it, and it works.

There is no authentication. Not weak authentication, not a shared token that could be sniffed. None. Every request that reaches the port is honored, including the ones that reconfigure the FPGA, and the traffic is plain HTTP. Anything with a route to the board can rewrite a third of its programmable logic. The source says so, the console page says so, and it is repeated here because this is the kind of limitation that becomes invisible once the thing works: the board belongs on an isolated segment and its port must not be forwarded.

The image validation described in Section 5.8.5 is a correctness check, not a security control. It stops a valid partial from being written into the wrong partition and a foreign device's image from being written at all, which are the mistakes a well-intentioned operator makes. It does nothing against a deliberately malformed image, and the *force* parameter disables even that.

Extending this beyond a trusted laboratory network would require authenticated transport, authorization, signed metadata, target-side hash verification, a shell compatibility identifier checked before programming, and a defined recovery path from an interrupted exchange [9, 18]. Network reception and DFX polling should also move to a non-blocking state machine, so that a stalled controller cannot hold up the event loop. Today it can, for as long as a bounded polling loop runs.

7.9 Board specificity and transferability

The architecture is reusable; this particular project is board specific. DDR timing, MIO placement, the Ethernet controller, the PHY, clocks, boot devices and processing-system initialization all follow the Z19 hardware, and the floorplan follows the XCZU19EG die.

What keeps that specificity manageable is that it is collected rather than scattered. The part, the floorplan and the processing-system configuration are three files; the block designs, the ICAPE3 wrapper and the boundary contract refer to none of them. The processing system is the piece most likely to go wrong quietly, because an exported block design embeds a complete PS configuration and applying a new one on top of it leaves the rest of the old one underneath. Building the PS from a board file instead of from an embedded block is what stops that, and it is the reason the settings in Table 5.2 could be checked one by one against the ALINX schematics rather than taken on trust.

Moving the architecture to another device would mean writing those same three things. The floorplan is the part that cannot be mechanical, because clock-region geometry, the configuration site and the processing system sit in different places on every device, and Section 5.4 is essentially a worked example of finding them.

7.10 Evaluation boundaries

The evaluation rests on implementation reports, generated artifacts, software build outputs, source inspection, analytical bounds and bench measurement. Together they characterize the FPGA platform and show that it runs. They do not constitute a performance evaluation.

The modules exercised on hardware are much smaller than their partitions, and resource and timing behaviour will differ for application-scale accelerators. The measurements come from one board, one operator and one direct Ethernet link. Integration with radios and a D-MIMO experiment remains future work, so the conclusions are limited to the accelerator platform and the workflow around it.

8

Conclusion and future work

This thesis built a DFX accelerator platform for the XCZU19EG on the ALINX Z19, and then ran it. The processing system, DDR and network paths, data movement, control and reconfiguration infrastructure stay in the static design. Three independently floorplanned partitions sit around it, all presenting the same 146-pin boundary: a clock, an active-low reset, a 32 bit control word, a 32 bit status word, a shutdown handshake and a 32 bit AXI4-Stream port in each direction. Each has its own AXI DMA, its own control and status registers, its own decoupler and its own virtual socket manager. Because the boundary is identical, a module is written once; because the partitions sit in different places on the die, it is built once per partition against that partition's abstract shell.

Vivado closed timing on the parent with a setup slack of 4.342 ns and a hold slack of 0.013 ns, no failing endpoints and no unrouted nets. The design occupies 2.31 % of the device LUTs, 1.66 % of its registers, 0.61 % of its block RAM tiles and 0.15 % of its DSP slices. The build produced one 36343250 B full bitstream, three partial bitstreams of 2594564 B, 3191480 B and 2754800 B, and three abstract shells of roughly 1.75 MB each. The exported XSA was sufficient to build the standalone platform and the network-loader application.

The partial binaries are between 91 % and 93 % smaller than the full image, and their sizes follow partition geometry rather than module content. With the 96.97 MHz PL clock recorded in the hardware handoff and the 32 bit ICAPE3 port, the ideal payload-transfer bounds are 6.69 ms, 8.23 ms and 7.10 ms, against 93.70 ms for the full image. These exclude controller sequencing, memory arbitration, software and network overhead, and are not measurements of service interruption.

On the board, it works. After programming, the application reaches its serving state without operator intervention, answers ICMP echo in 0.110 ms, and reconfigures each of the three partitions over HTTP while the other two stay resident and responsive. A module written from the template and built only against an abstract shell (no parent project, no parent bitstream) produces a partial the loader accepts, and identifies itself afterwards through the status word.

Getting there took four major design obstacles that no report, simulation or design review would have produced, and each of them is worth as much as the architecture. Three were silent. The first is that the fabric never reconfigures at all until software takes the configuration port away from PCAP: *CSU_PCAP_CTRL.PCAP_PR* comes out of reset selecting the processor configuration access port, *ICAPE3* never reports itself available while that is true, and the DFX Controller waits in its loading state

indefinitely without setting an error, because from its point of view nothing has gone wrong. The second is that this board's Ethernet PHY is one the generated BSP does not recognise, so it falls through to a routine for a different manufacturer's part, writes a register that does not exist and resets the PHY on the way past; the link then negotiates gigabit and carries no frames in either direction, with zero receives and zero receive errors. The third is that the negotiated speed must not be read the instant the link comes up, because the PHY's result bits are not valid yet and decode as the slowest option. That configured the MAC for $10\,\mathrm{Mbit\,s^{-1}}$ half duplex on a gigabit link, which passed nothing and reported no error. The fourth finding was not silent but misplaced: with the board's unused gigabit-transceiver peripherals left enabled, processor initialisation failed part way through its SERDES programming and left the debug access port wedged, so the board enumerated no processor over JTAG until it was power-cycled, and looked disconnected rather than misconfigured.

All four share a same failure pattern: correct on paper, wrong or misleading in the tools, and diagnosable only with the board on the bench. A platform verified against reports and generated artifacts alone would have passed every check in this thesis and not worked.

The contribution is the reusable platform boundary and the package around it, rather than a new D-MIMO processing algorithm. The single contract lets later modules keep the processing system, data movement, network service and reconfiguration path untouched, and the abstract shells let somebody produce a valid partial bitstream without ever opening the parent project. The reference and example modules establish that the interfaces synthesize, implement and exchange on hardware; they are not presented as telecommunications algorithms.

8.1 Future work

The most valuable next step would be measurement. Everything reported about configuration time here is an analytical bound, and the interesting numbers can only come from the board: how long an upload takes, how long the controller spends writing frames, and how long a partition is actually unavailable.

Further work should address:

- Trying other partition geometries. The three regions here were sized by the device's own legality constraints rather than by any workload, and they came out unequal: 3600, 6480 and 5040 SLICE, with four times as many DSP columns in the first as in the other two. Nothing establishes that this is a good split. Smaller partitions would cut partial-bitstream size and reconfiguration time; larger or differently shaped ones would host accelerators that will not fit today. The floorplan is one file, so the experiment is cheap to run, and the useful measurement is which shapes still close timing once an accelerator, rather than a pass-through, is placed in them;
- Specializing a socket for a data-access pattern the stream boundary serves poorly, once such accelerators exist: a sample-by-sample port without module-side backpressure, or a memory-mapped port onto a static dual-port BRAM that

keeps its contents across an exchange. Either is a change to the partition pins, so it invalidates every partial built for that partition and should follow, not precede, a first library of stream-boundary modules;

- Bringing a slice of programmable-logic I/O into the partition boundary, so that a module can drive an interface of its own, SPI, I^2C or something custom, rather than reaching the outside world only through DDR. The boundary is frozen by every partial bitstream built against it, so adding pins invalidates all of them at once; it is therefore a change to make early, before a library of modules exists, or not at all;
- Measurement of upload time, controller reconfiguration time and service interruption on the physical Z19, and where the DMA-driven controller falls against the published range for configuration throughput;
- Hierarchical utilization and timing analysis for each partition against accelerator-scale candidate modules rather than the small examples used so far;
- A shell and parent compatibility identifier, checked before programming, so that a partial built against a superseded parent is refused rather than loaded;
- Authenticated transport, authorization, signed metadata and a defined recovery path from an interrupted exchange, none of which the present demonstrator has;
- Non-blocking handling of network traffic and DFX state transitions, so that a stalled controller cannot hold up the bare-metal event loop;
- Generation of a persistent boot image, so that the board comes up serving without JTAG;
- A PetaLinux variant with DMA-backed result buffering, authenticated WebSocket streaming for live browser plots, SSH and SCP administration, user accounts, logging and concurrent clients [37, 34];
- A language-neutral module import flow accepting VHDL, Verilog or SystemVerilog source sets, selecting the correct top level and validating the elaborated ports against the boundary before implementation; and
- An HLS-authored module, including C tests, C/RTL co-simulation, interface checks, implementation into an abstract shell, timing analysis and pblock-fit verification.

These would test the architecture under accelerator-scale and deployment conditions, while keeping the single boundary contract that this work established.

Bibliography

[1] Hien Quoc Ngo et al. “Cell-free massive MIMO versus small cells”. In: *IEEE Transactions on Wireless Communications* 16.3 (2017), pp. 1834–1850. DOI: 10.1109/TWC.2017.2655515.

[2] Emil Björnson and Luca Sanguinetti. “Scalable cell-free massive MIMO systems”. In: *IEEE Transactions on Communications* 68.7 (2020), pp. 4247–4261. DOI: 10.1109/TCOMM.2020.2987311.

[3] Husileng Bao et al. “Automatic Distributed MIMO Testbed for Beyond 5G Communication Experiments”. In: *2021 IEEE MTT-S International Microwave Symposium (IMS)*. 2021, pp. 697–700. DOI: 10.1109/IMS19712.2021.9574865. URL: https://research.chalmers.se/publication/527040/file/527040_Fulltext.pdf (visited on 08/17/2026).

[4] Lise Aabel et al. “A TDD Distributed MIMO Testbed Using a 1-Bit Radio-Over-Fiber Fronthaul Architecture”. In: *IEEE Transactions on Microwave Theory and Techniques* 72.10 (2024), pp. 6140–6152. DOI: 10.1109/TMTT.2024.3389151. URL: https://arxiv.org/abs/2403.17476 (visited on 08/17/2026).

[5] *28 GHz Multi-User Massive Distributed-MIMO with Spatial Multiplexing and Calibration (NEC Technical Journal)*. Tech. rep. Accessed: 2026-01-15. 2023. URL: https://www.nec.com/en/global/techrep/journal/g23/n01/pdf/230111.pdf.

[6] Advanced Micro Devices (AMD). *Vivado Design Suite User Guide: Dynamic Function eXchange (UG909)*. Accessed: 2026-01-15. 2023. URL: https://docs.amd.com/r/en-US/ug909-vivado-partial-reconfiguration.

[7] *ALINX Z19 Zynq UltraScale+ MPSoC Development Board*. Accessed: 2026-01-15. ALINX. URL: https://www.en.alinx.com/Product/SoC-Development-Boards/Zynq-UltraScale-plus-MPSoC/Z19.html.

[8] ALINX Electronic Ltd. *Z19 vendor repository: hardware schematics, factory Vivado project and Linux sources*. Vendor repository, tracking its master branch. The schematics and the factory processing-system configuration relied on in this work are pinned at commit 8f35a82d3781498f38affbf7db9c6c1e182e8b05. ALINX. URL: https://github.com/alinxalinx/Z19 (visited on 08/25/2026).

[9] R. Elnaggar et al. “Multi-Tenant FPGA-based Reconfigurable Systems: Attacks and Defenses”. In: Accessed: 2026-01-15. 2019. URL: https://past.date-conference.com/proceedings-archive/2019/pdf/0528.pdf.

[10] “FPGA Devices”. In: *Introduction to Embedded System Design Using Field Programmable Gate Arrays*. London: Springer London, 2009, pp. 53–80. ISBN:

978-1-84882-016-6. DOI: 10.1007/978-1-84882-016-6_3. URL: https://doi.org/10.1007/978-1-84882-016-6_3.

[11] Advanced Micro Devices, Inc. *Zynq UltraScale+ MPSoC Data Sheet: Overview*. DS891. Version 1.11.1. 2025. URL: https://docs.amd.com/v/u/en-US/ds891-zynq-ultrascale-plus-overview (visited on 08/17/2026).

[12] Advanced Micro Devices, Inc. *Zynq UltraScale+ MPSoC Technical Reference Manual*. UG1085. Version 2.5. 2025. URL: https://docs.amd.com/r/en-US/ug1085-zynq-ultrascale-trm (visited on 08/17/2026).

[13] Advanced Micro Devices, Inc. *Zynq UltraScale+ MPSoC Processing System LogiCORE IP Product Guide*. PG201. Version 3.4. 2022. URL: https://docs.amd.com/r/3.4-English/pg201-zynq-ultrascale-plus-processing-system/General (visited on 08/17/2026).

[14] ALINX Electronic Ltd. *Z19 and ACU19EG Hardware Schematics*. Z19 carrier and ACU19EG processor-module schematics, from directory hardware/Schematics in the ALINX Z19 vendor repository, pinned revision. 2022. URL: https://github.com/alinxalinx/Z19/tree/8f35a82d3781498f38affbf7db9c6c1e182e8b05/hardware/Schematics (visited on 08/17/2026).

[15] ALINX Electronic Ltd. *Z19 Factory Vivado Processing-System Configuration*. Processing-system configuration script in the ALINX Z19 vendor repository, pinned revision. 2023. URL: https://github.com/alinxalinx/Z19/blob/8f35a82d3781498f38affbf7db9c6c1e182e8b05/vivado/auto_create_project/ps_config.tcl (visited on 08/17/2026).

[16] AMD (Xilinx). *Dynamic Function eXchange Decoupler LogiCORE IP Product Guide*. PG375. Version 1.0. Accessed: Thursday 17th September, 2026. Advanced Micro Devices, Inc. May 2022. URL: https://docs.amd.com/r/en-US/pg375-dfx-decoupler.

[17] AMD (Xilinx). *Dynamic Function eXchange Controller LogiCORE IP Product Guide*. PG374. Version 1.0. Accessed: 2026-08-21. Advanced Micro Devices, Inc. June 2020, pp. 71–72. URL: https://docs.amd.com/v/u/en-US/pg374-dfx-controller.

[18] Advanced Micro Devices, Inc. *UltraScale Architecture Configuration User Guide*. UG570. Version 1.20.1. 2025. URL: https://docs.amd.com/r/en-US/ug570-ultrascale-configuration (visited on 08/17/2026).

[19] ARM Ltd. *AMBA AXI and ACE Protocol Specification*. ARM IHI 0022D. 2011. URL: http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf.

[20] Xilinx Inc. *AXI DMA v7.1 LogiCORE IP Product Guide (PG021)*. 2021. URL: https://docs.amd.com/v/u/en-US/pg021_axi_dma.

[21] Advanced Micro Devices, Inc. *Vitis High-Level Synthesis User Guide*. UG1399. Version 2025.1. 2025. URL: https://docs.amd.com/r/2025.1-English/ug1399-vitis-hls (visited on 08/17/2026).

[22] Kizheppatt Vipin and Suhaib A. Fahmy. “FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications”. In: *ACM Computing Surveys* 51.4 (2018), 72:1–72:39. DOI: 10.1145/3193827.

[23] Joao Vieira et al. “A flexible 100-antenna testbed for Massive MIMO”. In: *2014 IEEE Globecom Workshops (GC Wkshps)*. IEEE, 2014, pp. 287–293. DOI: 10.1109/GLOCOMW.2014.7063446.

[24] Steffen Malkowsky et al. “The world’s first real-time testbed for massive MIMO: Design, implementation, and validation”. In: *IEEE Access* 5 (2017), pp. 9073–9088. DOI: 10.1109/ACCESS.2017.2705561.

[25] Clayton Shepard et al. “Argos: Practical many-antenna base stations”. In: *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking (MobiCom ’12)*. ACM, 2012, pp. 53–64. DOI: 10.1145/2348543.2348553.

[26] Rahman Doost-Mohammady et al. “RENEW: Programmable and observable massive MIMO networks”. In: *2018 52nd Asilomar Conference on Signals, Systems, and Computers*. IEEE, 2018, pp. 1654–1658. DOI: 10.1109/ACSSC.2018.8645552.

[27] Gilles Callebaut et al. “Techtile – Open 6G R&D testbed for communication, positioning, sensing, WPT and federated learning”. In: *2022 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit)*. IEEE, 2022, pp. 417–422. DOI: 10.1109/EuCNC/6GSummit54941.2022.9815734.

[28] *RENEW Wireless System Architecture*. RENEW Wireless. URL: https://wiki.renew-wireless.org/en/architectures/systemarchitecture (visited on 08/17/2026).

[29] Dumitra Iancu et al. “A Scalable 256-Antenna Distributed MIMO Testbed with Real-Time Fully Digital Beamforming”. In: (2026). DOI: 10.48550/arXiv.2605.02388. arXiv: 2605.02388 [eess.SP]. URL: https://arxiv.org/abs/2605.02388 (visited on 08/17/2026).

[30] Dirk Koch, Christian Beckhoff, and Jürgen Teich. “Partial reconfiguration on FPGAs: Architectures, tools and applications”. In: *ACM Transactions on Reconfigurable Technology and Systems* 6.4 (2013), pp. 1–30. DOI: 10.1145/2535378.

[31] Ken Peffers et al. “A Design Science Research Methodology for Information Systems Research”. In: *Journal of Management Information Systems* 24.3 (2007), pp. 45–77. DOI: 10.2753/MIS0742-1222240302.

[32] ISO/IEC/IEEE. *Systems and Software Engineering—Life Cycle Processes—Requirements Engineering*. ISO/IEC/IEEE 29148:2018. 2018. URL: https://www.iso.org/standard/72089.html (visited on 08/17/2026).

[33] Advanced Micro Devices, Inc. *Vitis Unified Software Platform Documentation: Embedded Software Development*. UG1400. Version 2025.2. Accessed: 2026-08-16. 2025. URL: https://docs.amd.com/r/en-US/ug1400-vitis-embedded.

[34] Advanced Micro Devices, Inc. *PetaLinux Tools Documentation: Reference Guide*. UG1144. Version 2025.2. Accessed: 2026-08-16. 2025. URL: https://docs.amd.com/r/en-US/ug1144-petalinux-tools-reference-guide.

[35] Advanced Micro Devices, Inc. *OS and Libraries Document Collection*. UG643. Version 2025.1. 2025. URL: https://docs.amd.com/r/2025.1-English/oslib_rm/Documentation (visited on 08/17/2026).

[36] *lwIP Callback-Style Raw API*. lwIP project. URL: https://lwip.nongnu.org/2_1_x/group__callbackstyle__api.html (visited on 08/17/2026).

[37] Ian Fette and Alexey Melnikov. *The WebSocket Protocol*. RFC 6455. RFC Editor, Dec. 2011. DOI: 10.17487/RFC6455. URL: https://www.rfc-editor.org/rfc/rfc6455.html (visited on 08/17/2026).

[38] Advanced Micro Devices, Inc. *Vivado Design Suite User Guide: Design Analysis and Closure Techniques*. UG906. Version 2025.2. Accessed: 2026-08-17. 2025. URL: https://docs.amd.com/r/2025.2-English/ug906-vivado-design-analysis/Floorplanning.

[39] Alessio Montone et al. "Placement and Floorplanning in Dynamically Reconfigurable FPGAs". In: *ACM Transactions on Reconfigurable Technology and Systems* 3.4 (2010), 24:1–24:34. DOI: 10.1145/1862648.1862654.

[40] Pingakshya Goswami and Dinesh K. Bhatia. "Floorplanning of Partially Reconfigurable Design on Heterogeneous FPGA". In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Abstract. 2016, p. 275. DOI: 10.1145/2847263.2847323.

[41] Norbert Deak, Octavian Creţ, and Horia Hedeşiu. "Efficient FPGA Floorplanning for Partial Reconfiguration-Based Applications". In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2019, p. 309. DOI: 10.1109/FCCM.2019.00050.

[42] Roy T. Fielding, Mark Nottingham, and Julian Reschke. *HTTP Semantics*. RFC 9110. RFC Editor, June 2022. DOI: 10.17487/RFC9110. URL: https://www.rfc-editor.org/rfc/rfc9110.html (visited on 08/17/2026).

[43] Roy T. Fielding, Mark Nottingham, and Julian Reschke. *HTTP/1.1*. RFC 9112. RFC Editor, June 2022. DOI: 10.17487/RFC9112. URL: https://www.rfc-editor.org/rfc/rfc9112.html (visited on 08/17/2026).

[44] Advanced Micro Devices, Inc. *Vivado Design Suite User Guide: Synthesis (UG901)*. Version 2025.2. Accessed: 2026-08-17. 2025. URL: https://docs.amd.com/r/2025.2-English/ug901-vivado-synthesis.

[45] Advanced Micro Devices, Inc. *XHwIcap driver: device documentation*. Source file in the AMD embeddedsw repository, release tag `xilinx_v2025.2`. 2025. URL: https://github.com/Xilinx/embeddedsw/blob/xilinx_v2025.2/XilinxProcessorIPLib/drivers/hwicap/src/xhwicap.c (visited on 08/25/2026).

[46] Advanced Micro Devices, Inc. *xemacpsif_physpeed.c: PHY speed detection in the lwIP 2.2.0 port*. Source file in the AMD embeddedsw repository, release tag `xilinx_v2025.2`. 2025. URL: https://github.com/Xilinx/embeddedsw/blob/xilinx_v2025.2/ThirdParty/sw_services/lwip220/src/lwip-2.2.0/contrib/ports/xilinx/netif/xemacpsif_physpeed.c (visited on 08/25/2026).

[47] Microchip Technology Inc. *KSZ9031RNX Gigabit Ethernet Transceiver with RGMII Support*. DS00002117J. 2022. URL: https://ww1.microchip.com/downloads/aemDocuments/documents/UNG/ProductDocuments/DataSheets/KSZ9031RNX-Data-Sheet-DS00002117J.pdf (visited on 08/25/2026).

A

The block design as built

The drawings in this appendix are the Vivado IP integrator block design itself, exported from the tool with *write_bd_layout* rather than redrawn. They are here rather than in Chapter 5 because Figure 5.2 already carries the architecture in a form built for reading: it separates the two traffic classes by colour and leaves out the pin-level detail that would bury them. What follows is the artifact behind that figure, for a reader who wants to check it against the design that was actually built.

The block design is hierarchical, so one drawing cannot show all of it. Each level is given its own figure, from the top level inward. Two families of hierarchy repeat three times over, one per reconfigurable partition, and only the first of each is reproduced; Sections A.4 and A.5 say exactly what differs in the other two.

Software sources are not reproduced here. Printing four and a half thousand lines of C would not let anyone verify anything that reading the repository would not, and it would be stale as soon as the code moved. The interfaces that the report makes claims about are given as tables in the body instead: the partition boundary in Table 5.3 and the HTTP surface in Table 5.6.

A.1 Top level

Figure A.1 is *design_1*, the design whose wrapper is the synthesis top. Eight blocks sit at this level. Three of them (*rp1*, *rp2* and *rp3*) carry the DFX badge in their top-left corner and name the block design each one contains, *rp1_rm0.bd* and its siblings. Those three are the reconfigurable partitions, and everything else on the page is static.

The thing worth tracing here is that each partition faces exactly one *rpN_static* hierarchy and nothing else. There is no shared datapath between partitions and no route from one to another: a module can only reach DDR through its own hierarchy, which is what makes the three independent at runtime as well as on the die.

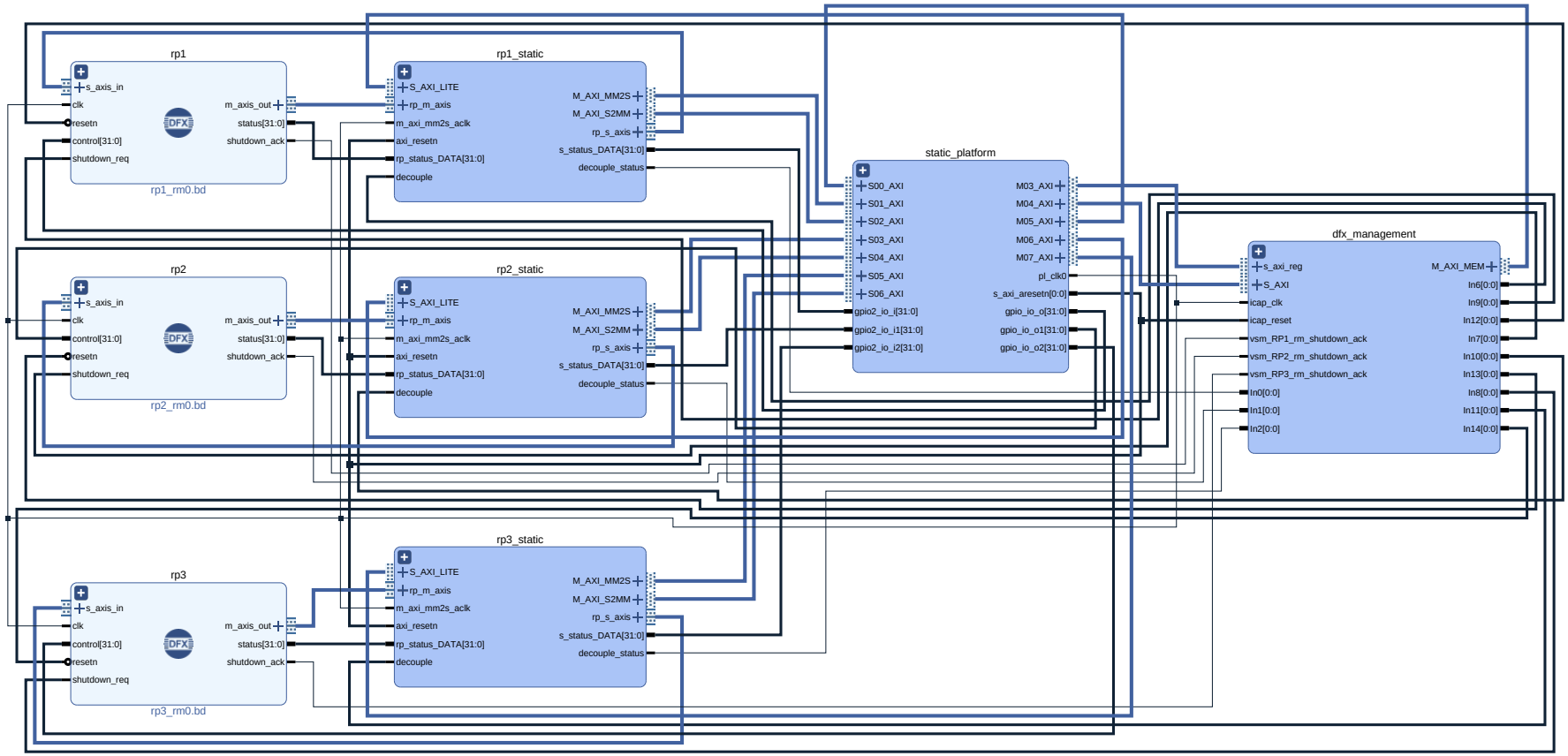


Figure A.1: design_1, *the top level of the block design. The three blocks marked with the DFX badge are the reconfigurable partitions; each names the block design it contains.*

A.2 The static platform

Figure A.2 is *static_platform*, which holds the processing system and everything that fans out from it.

Two AXI SmartConnects divide the work, and the split is easier to see here than anywhere else in the report. *smartconnect_0* hangs off *M_AXI_HPM0_LPD* and drives eight masters: the three per-partition register blocks, the three DMA register interfaces, and the two register interfaces in *dfx_management*. *dfx_mem_sc* runs the other way, collecting seven slave ports onto *S_AXI_HP0_FPD*: six DMA channels, two per partition, plus the DFX Controller's own memory master. That seventh port is the one the controller reads a partial bitstream through, and it is the whole of the configuration path's contact with DDR.

The three *rpN_regs* blocks are ordinary AXI GPIO. Channel 1 drives a partition's control word and channel 2 reads its status word back, which is the entire side channel described in Section 5.3.

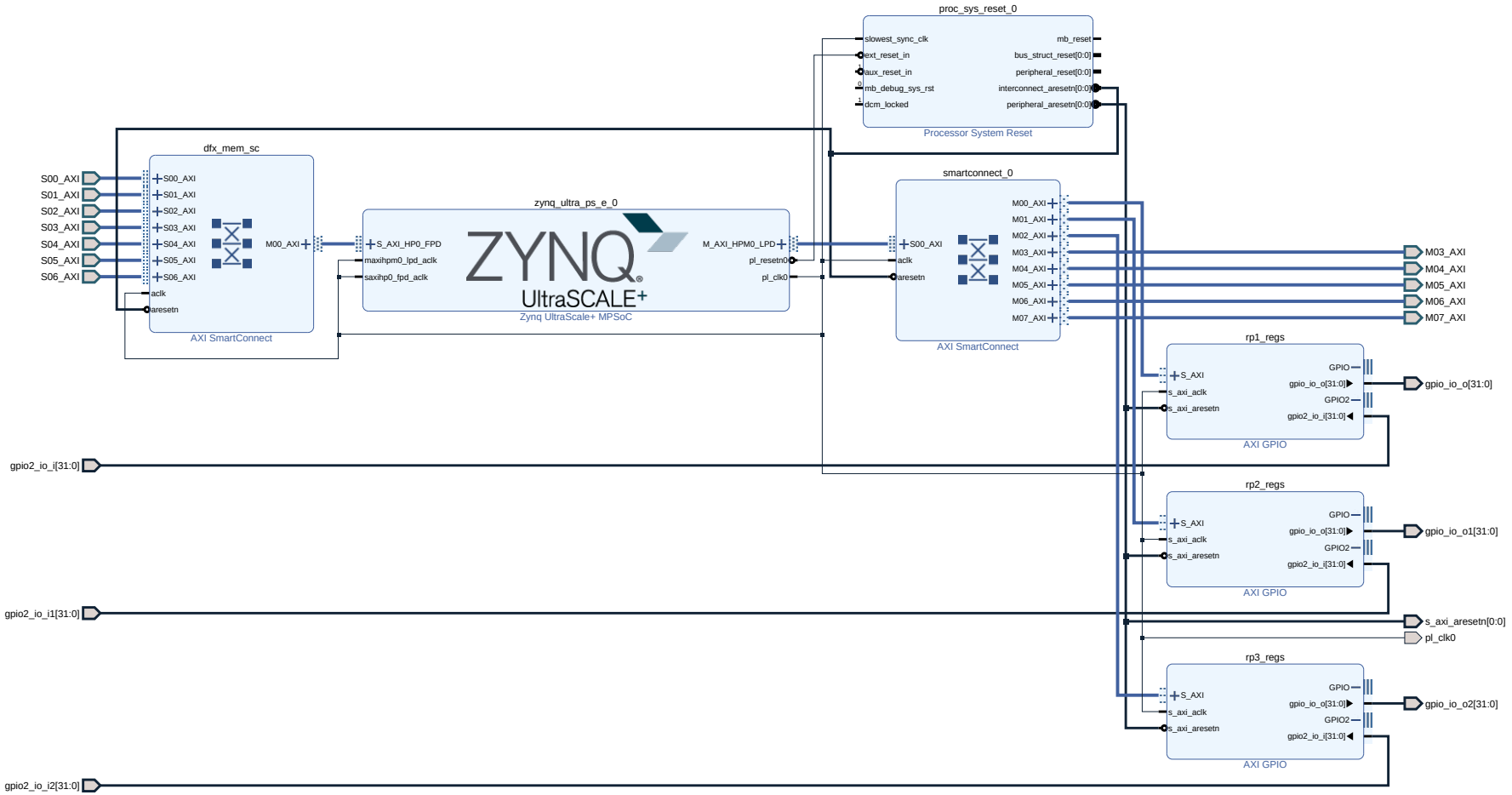


Figure A.2: static_platform*: the processing system, the control and memory SmartConnects, the reset block, and one control / status GPIO per partition.*

A.3 Reconfiguration management

Figure A.3 is *dfx_management*, and it settles a question the system-level figure can only assert.

dfx_ctrl drives *u_icap* directly. The controller's *icap_o*, *icap_csib* and *icap_rdwrb* go straight into the wrapper's "i", "csib" and "rdwrb" pins, and the wrapper returns "o", *avail*, *prdone* and *prerror*. Nothing sits between them. The event GPIO is not in that path at all: *dfx_mon_concat* gathers twenty-four single-bit lines from the controller and from the ICAP wrapper into one vector, and *dfx_mon* presents that vector to software as a read-only register. It observes the configuration path; it does not carry it.

The per-VSM signals are worth counting on the drawing. Each of the three virtual socket managers exposes its own *rm_shutdown_req*, *rm_decouple*, *rm_reset*, *event_error*, *sw_shutdown_req* and *sw_startup_req*, and takes back its own *rm_shutdown_ack*. That is what makes the three partitions independently reconfigurable from one controller.

A.4 One partition's static logic

Figure A.4 is *rp1_static*. Two blocks: *axi_dma_rp1* and the decoupler *rp1_iso*.

This drawing is the clearest statement of which signals are isolated during reconfiguration and which are not. The decoupler carries three things: the stream in, the stream out, and *rp_status_DATA[31:0]*. It takes *decouple* from the partition's VSM. The clock, the reset, the control word and the shutdown handshake do not appear on it, because they run past it directly to the partition. That asymmetry is deliberate and is argued in Section 5.3: only outputs and handshakes can misbehave while a partition is half-written, and a partition being rewritten still needs its clock.

rp2_static and *rp3_static* are the same drawing with the index changed. Comparing the generated sources, the only differences are which SmartConnect master port feeds the DMA's *S_AXI_LITE* (*M05*, *M06* and *M07* respectively) and which VSM's decouple net drives the decoupler. No block, port or connection differs otherwise.

A.5 The reference module

Figure A.5 is *rp1_rm0*, the module resident in a partition after a full-device program.

It is as small as it looks. *s_axis_in* connects straight to *m_axis_out*, so words pass through untouched, and *shutdown_req* connects straight to *shutdown_ack*, which is legitimate only because there is no buffered state to drain. The rest is a 32 bit binary counter: *ilslice_0* picks bit 0 out of the control word to gate its clock enable, *ilvector_logic_0* inverts the active-low reset into the counter's synchronous clear, and the count drives *status[31:0]*. Vivado implements that counter in a DSP48E2, which is where the three DSP slices in Table 6.2 come from.

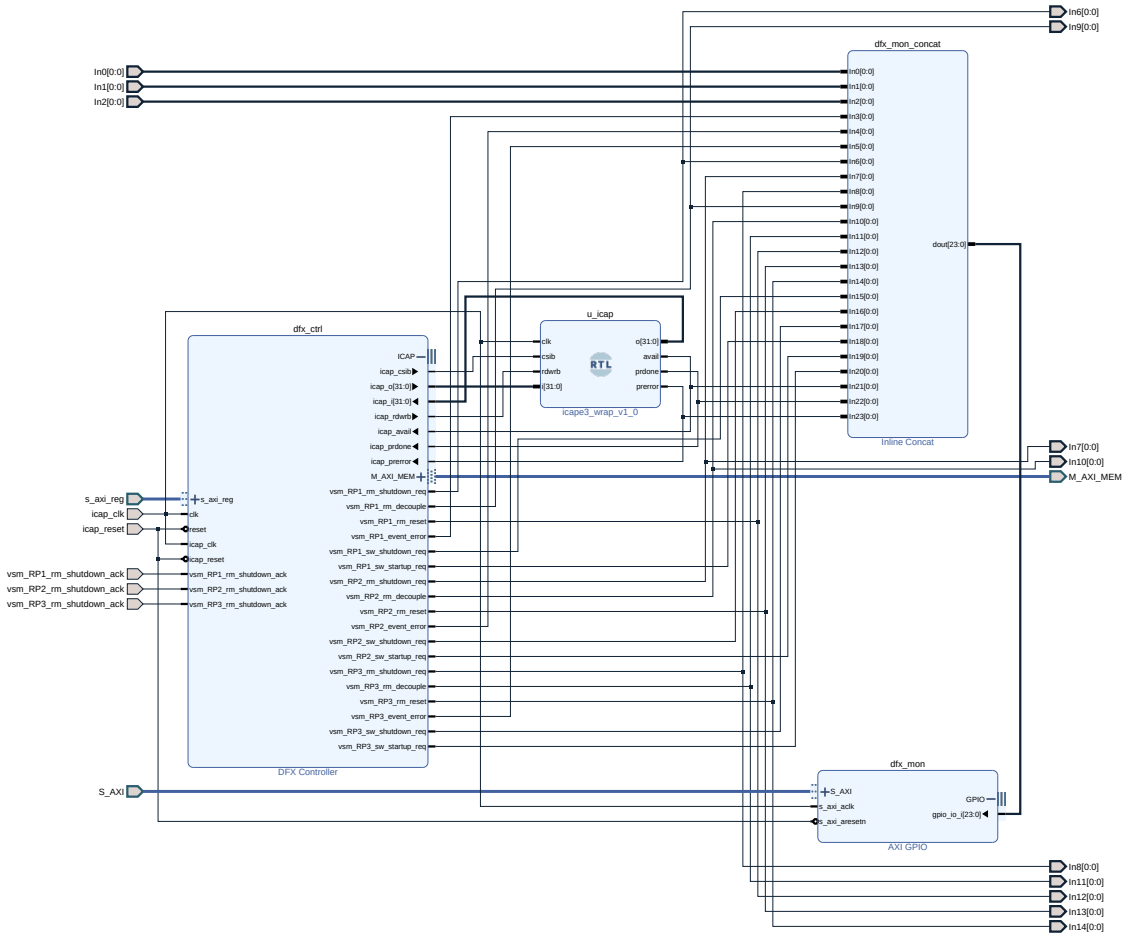


Figure A.3: dfx_management*: the DFX Controller, the* ICAPE3 *wrapper it drives directly, and the concatenation and AXI GPIO that let software observe both.*

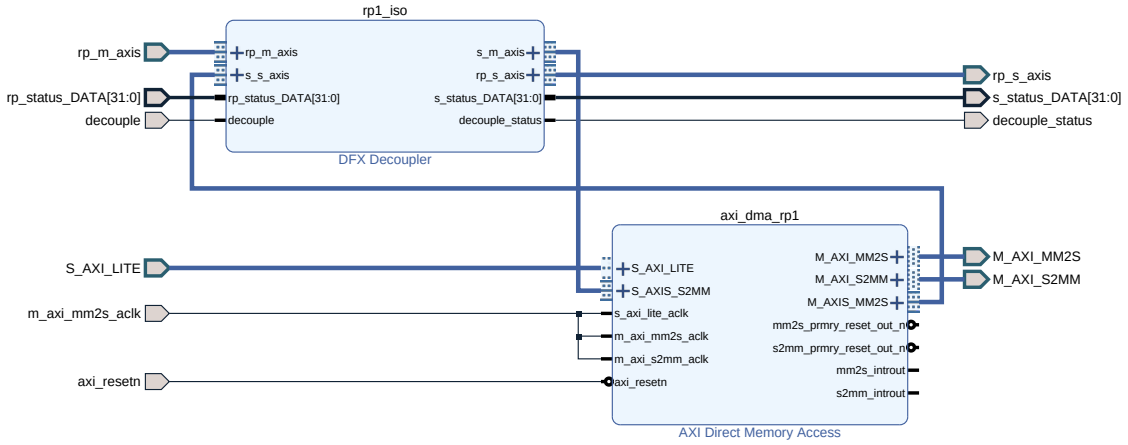


Figure A.4: rp1_static*: the partition's AXI DMA and its DFX Decoupler. Only the two stream ports and the status word pass through the decoupler.*

What the module does not do is as much the point as what it does. It exercises both stream directions, both control words and the shutdown handshake, and asserts nothing about what an accelerator should compute.

rp2_rm0 and *rp3_rm0* are identical to this design. Their generated sources differ only in the order in which two statements are emitted; every block, port, property and connection is the same.

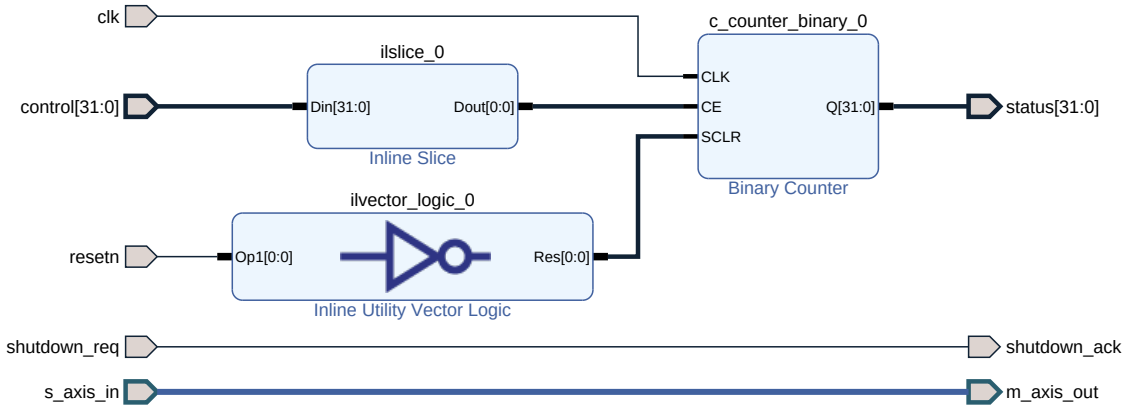


Figure A.5: rp1_rm0*, the reference module. The stream passes through untouched and the shutdown request is acknowledged immediately; the counter exists to give the status word something to report.*