Innovate by Customized Instructions, but Without Fragmenting the Ecosystem

by Joseph Yiu

White Paper

Arm® Custom Instructions, which was announced in October 2019, is now available in the Cortex-M33 and Cortex-M55 processors. In this paper, we review some of the design considerations and decisions when we created this architecture extension, a range of design considerations for SoC designers when they deploy their hardware accelerators based on this technology, and how it compares to the existing coprocessor interface feature available on the Cortex-M33 and Cortex-M55 processors.

The paper also covers some of the uses cases that Arm has investigated, such as mathematics accelerators, and finally, the topic of how software developers access the accelerators implemented using Arm Custom Instructions is also explained.

Introduction

In the last few years, the processing requirements in recent embedded systems has increased dramatically. This is partly due to the shifting of user's expectations following the enhancements in high-end consumer products. At the same time, the increasing communication speed and the exponential growth of information (e.g. more sensors, better quality cameras) also lead to the changing landscape of embedded computing. For example, there is a trend of increasing computing requirements for endpoint devices, including AI processing capabilities. The change is not just limited to the IoT market, but also industrial, automotive, healthcare, etc. As a result, the design of many modern microcontroller products not only need to address the "control" functions, but also need to address the "compute" needs.

The increasing processing performance has certainly been driving the rising processor performance in microcontrollers and SoCs. At the same time, the use of custom hardware accelerators has also become more widespread. This is not entirely new however, as hardware accelerators, like crypto engines, were already widely available in microcontroller products. The recent expansion in processing needs has seen hardware accelerators being used in a wider range of processing.

Due to the increasing use of these hardware accelerators, new Arm Cortex-M processors have also been adapted to provide better support for custom-defined accelerator solutions. Traditionally, silicon vendors integrate hardware accelerators as memory mapped units (Figure 1.1) externally to the processor and software access those units using memory read/write operations. These accelerators can also have their own bus interface for accessing memories directly.

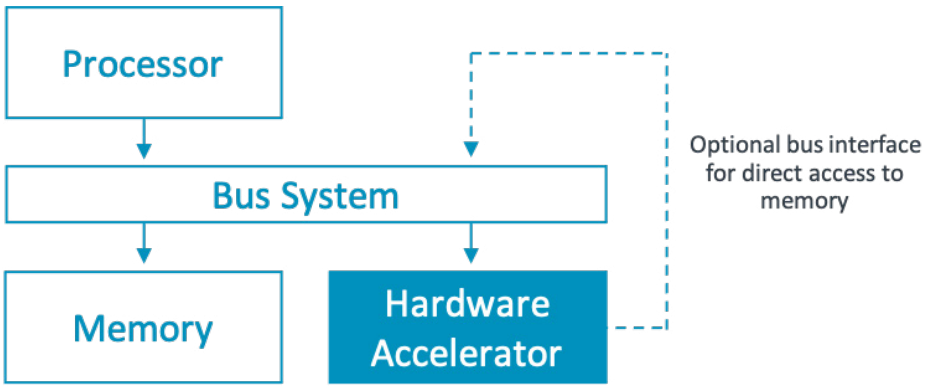

Figure 1.1: Concept of memory mapped hardware accelerator

In 2016, Arm announced the Cortex-M33 processor that supports a coprocessor interface. This feature (Figure 1.2) allows up to 8 external coprocessor hardware units to be connected to the processor via a dedicated interface.

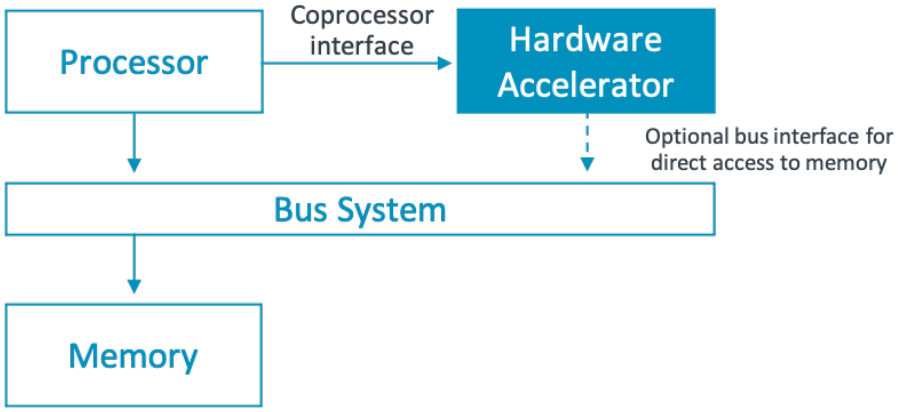

Figure 1.2: Concept of hardware accelerator coupling to a processor using the coprocessor interface

The coprocessor interface allows the hardware accelerators to be closely integrated to the processor, reducing latency in data and command transfers. The 64-bit bandwidth of the coprocessor interface on the Cortex-M33 also doubles the data transfer bandwidth when compared to the 32-bit AMBA® AHB bus interface. However, for some applications there is a strong need to merge specialized data processing functions into the processor to operate directly on general purpose registers. As a result, during Arm TechCon 2019, Arm announced the Arm Custom Instructions feature (Figure 1.3) which allows chip designers to include custom defined data processing instructions directly at the heart of the new Cortex-M processors.

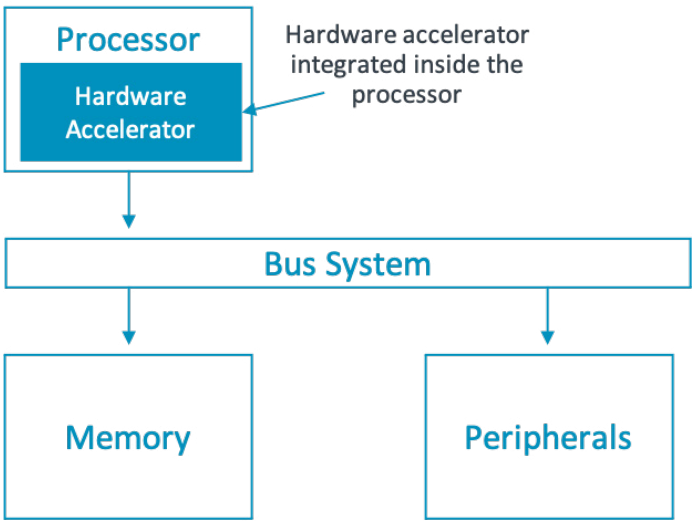

Figure 1.3: Concept of hardware accelerator based on Arm Custom Instructions

The Arm Custom Instructions feature is implemented in the Cortex-M33 revision 1 and Cortex-M55 revision 1.

The Challenges of Introducing Arm Custom Instructions

To support Arm Custom Instructions, Armv8-M architecture requires a new architectural extension, and this is named Custom Datapath Extension (CDE). From the original concepts of allowing customized instructions, to the creation of the CDE architecture extension and then finally to the implementation, there is a range of challenges to consider and resolve:

Fragmentation of the Ecosystem

One of the first worries is about ecosystem fragmentation. While we love to see innovations in the product designs, we need to avoid fragmentation of the processor architecture which can lead to issues with compilation tools, debug tools and middleware (e.g. RTOS). For many years, the consistent architecture of Arm processors enabled software developers to use a single tool chain to develop software for multiple Arm products from different vendors, and offered the ability to reuse their Cortex-M middleware across a wide range of Cortex-M devices. We want to keep this unchanged.

Limitation of the Instruction Encoding Space

The second key challenge is that the instruction encoding space for the Thumb instruction set is already quite crowded, and therefore it is hard to allocate a big block of instruction encoding space for Arm Custom Instructions. The justification of allocating a block of reserved instruction encoding space for the this feature is even harder when considering many customers might not use customized instructions in their design at all. At the same time, Arm must also ensure that there are spaces available for future extensions for the Thumb instruction set.

Scalability of the Design

The design of Arm Custom Instructions needs to be deployable in multiple processor designs. While the Cortex-M33 processor is the first product that supports Arm Custom Instructions, it will also be added to the Cortex-M55 processor and other future Cortex-M processors. Therefore, the design must be scalable so that it can work with processors designs with different pipeline designs.

In addition to these challenges, the design must also satisfy a range of requirements:

- General purpose/Generic – the design of Arm Custom Instructions needs to be able to support a wide range of use cases. While it is impossible to optimize the design for all different processing requirements, it is important to be able to cover a wide range.

- Ease-of-use – The design of Arm Custom Instructions needs to be easily accessible by software developers. For example, the features need to be accessible in C/C++ environment, do not require special changes in existing software (e.g. RTOS) and must not need specialized debug tools.

- Conserve the key characteristics of the Cortex-M processors – Addition of Arm Custom Instructions must not affect the key benefits of the Cortex-M processors, including low interrupt latency, security, and low power.

- Verification – The verification of the design needs to be straight forward, and the interface protocol of signals between the processor and the custom accelerator logic should be easy to understand and implement.

Overview of Arm Custom Instructions

A. Custom Datapath Extension (CDE)

Following over a year of architecture development, the Custom Instructions was born. This optional extension is called the Custom Datapath Extension (CDE) in the architecture reference manual. Whereas the name "Arm Custom Instructions" is for the CDE support feature implemented on Arm processors.

Unlike the custom instruction extensions you can find in other architectures, the encoding of the instructions in CDE are architecturally defined. These instructions are just like instruction templates, with the actual data processing operations inside being defined by silicon vendors.

The CDE contains 15 classes of instructions for:

- Different data types
- Various numbers of input parameters

The CDE instruction classes are listed in Table 3 1. (Note: Pn is coprocessor number (0 to 7), Rd/Rn/Rm are integer registers, Sd/Sn/Sm are 32-bit floating-point registers, Dd/Dn/Dm are 64-bit floating-point registers and Qd/Qn/Qm are 128-bit vector registers.)

INSTRUCTION	ASSEMBLY	INPUTS	OUTPUTS
General-purpose registers and NZCV flags			
CX1{A}	CX1{A} Pn, Rd,#imm	Immediate (13bit) and 1x 32-bit GPR/NZCV {same as output}	1x 32-bit GPR or NZCV
CX2{A}	CX2{A} Pn, Rd,Rn,#imm	Immediate (9bit) and 2x 32-bit GPR/NZCV {one same as output}	1x 32-bit GPR or NZCV
CX3{A}	CX3{A} Pn, Rd,Rn,Rm,#imm	Immediate (6bit) and 3x 32-bit GPR/NZCV {one same as output}	1x 32-bit GPR or NZCV
CX1D{A}	CX1D{A} Pn, Rd,Rd+1,#imm	Immediate (13bit) and 1x 32-bit GPR/NZCV {two same as output}	2x 32-bit GPR
CX2D{A}	CX2D{A} Pn, Rd,Rd+1,Rn,#imm	Immediate (9bit) and 2x 32-bit GPR/NZCV {two same as output}	2x 32-bit GPR
CX3D{A}	CX3D{A} Pn, Rd,Rd+1,Rn,Rm,#imm	Immediate (6bit) and 3x 32-bit GPR/NZCV {two same as output}	2x 32-bit GPR
Floating-point / Vector registers			
VCX1{A}	VCX1{A} Pn, Sd,#imm	Immediate (11bit) and 1x 32-bit fp32 register {same as output}	1x 32-bit fp32 register
VCX2{A}	VCX2{A} Pn, Sd,Sn,#imm	Immediate (6bit) and 2x 32-bit fp32 register {one same as output}	1x 32-bit fp32 register
VCX3{A}	VCX3{A} Pn, Sd,Sn,Sm,#imm	Immediate (3bit) and 3x 32-bit fp32 register {one same as output}	1x 32-bit fp32 register
VCX1{A}	VCX1{A} Pn, Dd,#imm	Immediate (11bit) and 1x 64-bit fp64 register {same as output}	1x 64-bit fp64 register
VCX2{A}	VCX2{A} Pn, Dd,Dn,#imm	Immediate (6bit) and 2x 64-bit fp64 register {one same as output}	1x 64-bit fp64 register
VCX3{A}	VCX3{A} Pn, Dd,Dn,Dm,#imm	Immediate (3bit) and 3x 64-bit fp64 register {one same as output}	1x 64-bit fp64 register
VCX1{A}	VCX1{A} Pn, Qd,#imm	Immediate (12bit) and 1x 128-bit vector register {same as output}	1x 128-bit vector register
VCX2{A}	VCX2{A} Pn, Qd,Qn,#imm	Immediate (7bit) and 2x 128-bit vector register {one same as output}	1x 128-bit vector register
VCX3{A}	VCX3{A} Pn, Qd,Qn,Qm,#imm	Immediate (4bit) and 3x 128-bit vector register {one same as output}	1x 128-bit vector register

Table 3 1: 15 classes of CDE instructions

For each instruction, there is

- a coprocessor number (0 to 7)
- destination register (can be flags)
- source register(s) (can also be flags)
- an immediate data value (see pages 9 “Supporting multiple custom instructions” and 16 “Handling of multiple instructions”)

For example, the syntax of the CX3 instruction is as follows:

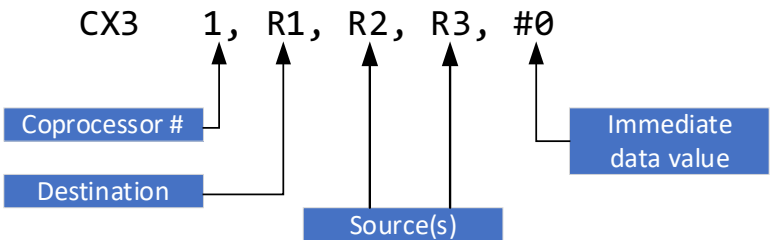

Figure 3.1: Example of CDE instruction syntax

For each class of instruction, there is also an "Accumulative" variant, noted by the A suffix (Figure 3.2). Accumulative variant means that the destination register is also an input of the data processing operation. While the term accumulative is used, it does not necessarily imply an "addition" of the previous value involved in the processing, but any operator implemented by the custom instruction.

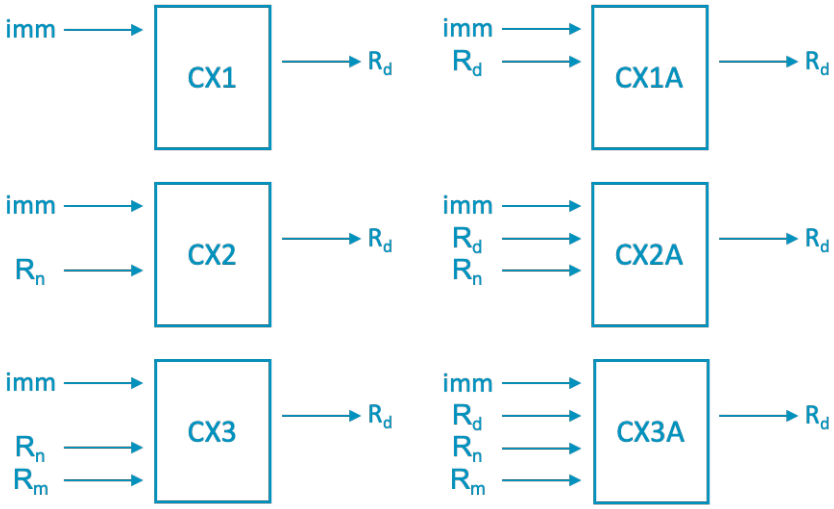

Figure 3.2: Examples of 32-bit integer CDE instructions with and without "A" suffix

For integer CDE operation, the "Dual" variant of the CDE instructions provides 64-bit data operations (Figure 3.3). Whereas for floating-point CDE instructions, the double precision variant provides 64-bit data operations. For processors that support the Helium® vector extension, the vector variant of CDE instructions provides 128-bit data operations.

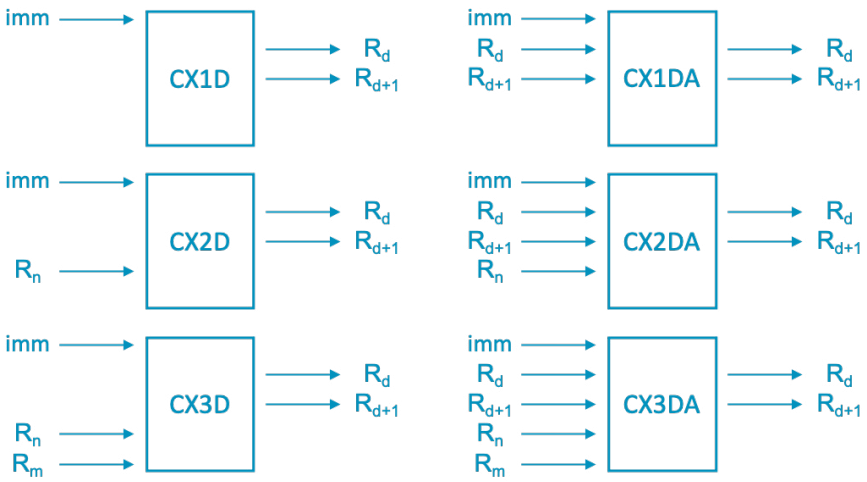

Figure 3.3: Dual variant of integer CDE instructions with and without "A" suffix

The CDE implementation in the Cortex-M33 processor supports integer (32-bit and 64-bit), and single precision floating-point operations. (The optional FPU in the Cortex-M33 processor supports single precision floating-point operations only). The floating-point and vector versions of CDE instructions are similar to the integer CDE instructions (Figure 3.2) in the sense that they support 0 to 3 inputs and an immediate data value. However, the floating-point and vector CDE instruction does not support the use of flags. Revision 1 of the Cortex-M55 supports the full set of CDE instructions.

B. Hardware Design

From a hardware designer point of view, there is no need to fully understand the internal operations of the processor to start designing the custom accelerator logic because the Cortex-M processor already included the logic for instruction decode, register read/write ports, and the immediate value extraction for the CDE instructions. An example is provided in the deliverable as a placeholder so that chip designers can replace the example data processing functions with their own designs. They also need to implement glue logic if multiple CDE accelerator units are implemented as they shared the same hardware interface. The hardware interface provides:

- Handshaking signal to support multi-cycle operations.
- Instruction decode error response signal to allow unsupported operations to be flagged up and handled by fault exception.

With such arrangement, the chip designers using the Cortex-M processor do not need to have a detailed understanding of the processor pipeline to make use of the CDE instructions. (Figure 3.4)

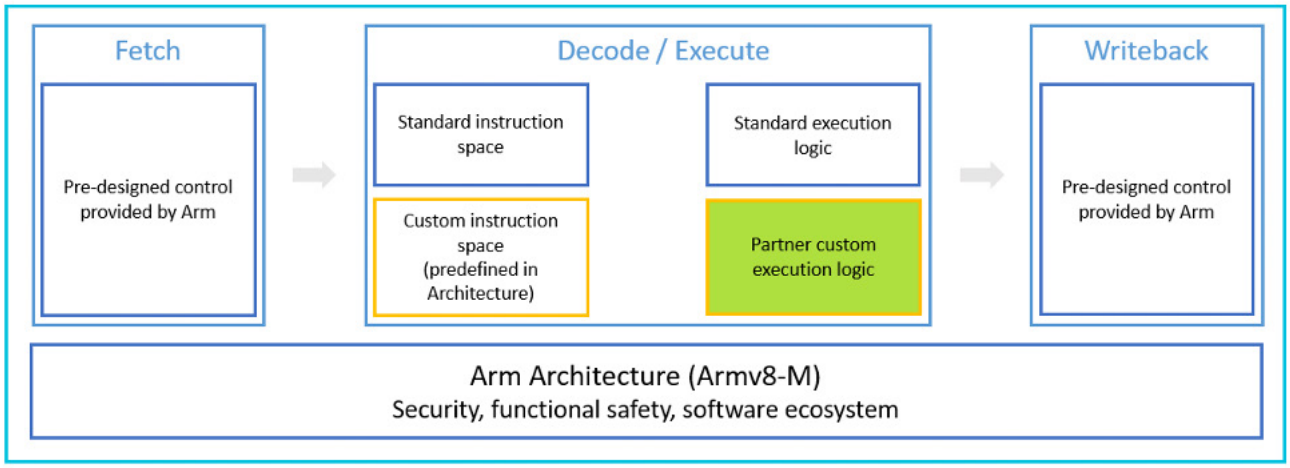

Figure 3.4: Designing hardware accelerators with CDE does not require in depth knowledge of the processor's pipeline

To support verification, interface protocol checkers are included in the deliverable.

C. Supporting Multiple Custom Instructions

Each CDE instruction supports an immediate data field, from 3 bits to 13 bits based on the instruction class. The width of the immediate value available for each class is listed in the INPUTS description in Table 3 1. This can be used to support multiple instructions with the same class. The immediate data value is extracted by the instruction decode logic and is provided to the hardware interface of the CDE logic so that the hardware can easily determine which operation is specified.

D. Security Management

For Cortex-M processors with the TrustZone® security extension, each coprocessor/ACI hardware unit can be assigned as Secure or Non-secure using the Non-secure Access Control Register (NSACR). This allows certain CDE instructions to be restricted to secure software only.

Design Decisions and Trade-Offs

Due to the range of technical challenges and requirements, several design decisions were made when defining the CDE architecture extension.

A. CDE Operations are Limited to Data Processing Operations

One of the most significant decisions in the early design stage is to restrict the extension to data processing operations only. This is needed to allow all CDE instructions to be implemented as data processing intrinsic functions, thus voided the need for customizing the C/C++ compilers, which would result in the fragmentation of the Arm tool ecosystem. In addition, this restriction also helps ensuring that the design can be scalable across a range of Cortex-M processor designs.

Some might wonder if it is a problem that CDE does not offer any branch operations. Because Armv8-M architecture already provides a very rich set of branch operations (e.g. conditional branch, table branch, conditional execution), for most applications there is not any performance benefits of having additional branch operations. To support such operations would also require C compilers to be modified and would result in fragmentation of the architecture and ecosystem – the first item in the design concerns. In addition, while a custom branch instruction can be possible in a short pipeline, it can be problematic for a future high-performance processor with a longer pipeline.

At this stage there are also no memory load/store instructions in the CDE. This is because we cannot ensure correct memory access ordering when the C compiler does not understand the custom defined addressing. In addition, memory access is closely coupled with security features like TrustZone and this area is fairly complex. Given that designs can already get performance gain by using CDE instruction(s) for address generation, and then use the generated address for data accesses, there is little benefit of having custom memory access instructions.

B. CDE Operations are Limited to 3 Inputs

The CDE supports only up to 3 data inputs and 1 result. This is needed to ensure that existing read/write ports can be reused. While it is true that some custom data operations could benefit from having more inputs/outputs, such capability is not supported because:

- The instruction encoding space is limited.
- If additional read/write ports are needed, it will likely increase power and area of the processor and be wasteful for designs that do not need the additional read/write ports.

Operations that require a large number of registers would be better implemented in external coprocessor hardware where the designers can include more hardware registers and can include dedicated bus interface to transfer the data between the accelerator and the memories.

C. No Internal Architectural State Inside CDE Hardware

The architecture does not allow CDE instructions to have custom defined architectural state across instruction boundaries for several reasons:

- Software debug aspect – Because the CDE hardware do not have hidden architectural state, the operations of the CDE operations can be debugged easily.
- Easy context switching – Because all CDE instructions only use processor's registers, context switching in RTOS environment is supported out-of-the-box as RTOS context switching software does not require any changes. In the same aspect, for a processor with the TrustZone security extension both Secure and Non-secure software can use the same Arm Custom Instructions without the risk of accidentally leaking secure information.

- Enable real-time capability – To allow the Cortex-M processor to have a low interrupt latency, the CDE interface design needs to allow multi-cycle CDE instructions to be terminated early in the case of an interrupt event. If the custom accelerator logic has internal states, the states could have been updated while the instruction is abandoned, and can result in incorrect operation when the instruction is restarted.

Please note that internal state is allowed during the execution of a CDE instruction (e.g. Finite State Machine, temporary data storage). It is just that the CDE instruction's operation must not rely on the state from previous execution. For application scenarios where persistent internal state is required, the coprocessor hardware solution would be more suitable.

D. Reuse of Coprocessor Instruction Encoding Space

Since the instruction encoding space is limited, Arm made the decision to reuse coprocessor instruction encoding space for Arm Custom Instructions. The CDE/coprocessor instructions both have a coprocessor number bit field. So, when a coprocessor/CDE instruction is decoded, the coprocessor number is extracted from the instruction, and processed as either a CDE instruction or as a coprocessor instruction based on the hardware configuration of the processor, which is defined by the chip designer (Figure 4.1).

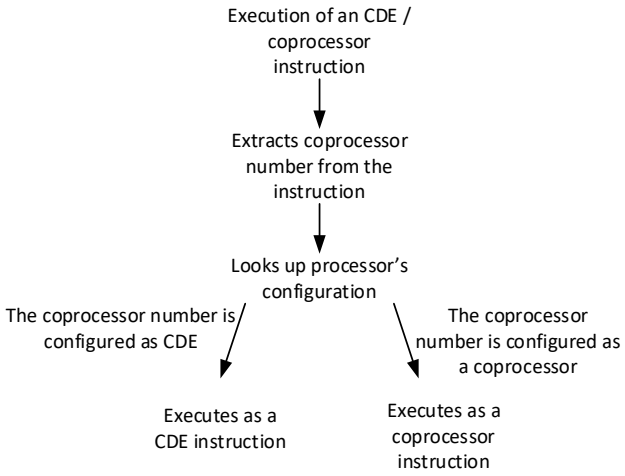

Figure 4.1: Decision of whether an instruction is handled as CDE or coprocessor operation

With this arrangement, CDE and coprocessor accelerators can coexist. And for each coprocessor number it can only be either a CDE accelerator or an external coprocessor accelerator (not both). This should not be an issue because each hardware accelerator can carry multiple functions, and therefore it is unnecessary to have a large number of coprocessor or CDE accelerators.

E. Execution of a Multi-Cycle CDE Instruction can Block Subsequent Instructions

Because CDE instruction execution uses the processor's register bank, it can block subsequent instructions if the CDE operation takes multiple clock cycles. However, in cases where background processing is desirable, the existing coprocessor interface solution can satisfy the needs because the hardware accelerator built around the coprocessor interface has its own register file.

F. Return of NZCV Flags in CDE Instructions

CDE instructions with 32-bit integer result can optionally update NZCV flags (Negative, Zero, Carry, Overflow) in the Application Program Status Register (APSR) if the destination register is set to APSR_nzcv. While it cannot provide a data processing result value and update the NZCV flags at the same time, it is not possible to utilize both information at the same time in typical C/C++ programming method (e.g. via intrinsic functions) anyway.

In addition, by updating NZCV flags only when needed, the branch prediction hardware in a high-end processor does not have to consider potential flag changes caused by CDE instructions unless the CDE instruction explicitly specifies the flags as its output. It means the branch predication hardware in the processor is easier to implement.

If there is a need to have a conditional branch based on the result of a CDE instruction, one quickest method is to use an integer CDE instruction that returns a value (zero / non-zero), and branch if that value is zero or non-zero. In Armv8-M processors, the Compare-and-Branch instructions (CBZ and CBNZ) allows the result from such CDE instruction to be used as branch condition directly, hence provides efficient conditional branches based on CDE operations.

Comparison Between Arm Custom Instructions and Coprocessor Solutions

At first glance, the architecture of Arm Custom Instructions might look a bit restrictive. However, it is important to consider the fact that Arm Custom Instructions and the coprocessor interface are both available and complement each other. Chip designers can use either one based on their application requirements. Table 5.1 lists the key differences between the two features.

	Arm Custom Instructions	Coprocessor
Placement of the accelerator(s)	Inside the processor	Outside the processor
Registers used	Processor's registers	Coprocessor hardware have their own registers
Data width	32, 64, and 128 bits (Cortex-M33 does not support 128-bit vector)	32 and 64-bit transfer operations are supported.
Execution	Serialize with other instructions in the pipeline (except for the case of vector lane overlap in a processor with Helium® technology).	Able to operate in the background using its own register set
Other interface(s)	Not allowed	Can have optional bus manager interface for direct memory accesses. Can have optional bus subordinate interface for debug access to coprocessor registers. Can have optional output(s) to NVIC and other hardware.
Hardware sharing	It is not possible to share an ACI hardware accelerator between two processors	If the coprocessor has a bus subordinate interface, it can be accessed by another processor
Power domain	Same power domain as the processor's integer/floating-point data-path	Each coprocessor can have its own power domain. Architectural power control mechanism is provided.

Table 5.1: Comparison between Arm Custom Instructions and the coprocessor feature

In general:

- If a data processing operation is very short, i.e. single cycle or just a few clock cycles, implementing it as a CDE instruction could be better because it does not have the overhead of transferring the data to the coprocessor and back to the processor.
- If a data processing operation takes a long time to handle, then using the coprocessor method is better – while waiting for the data processing operations to be completed, the processor can work on other tasks.

Depending on the requirements, in some cases the hardware accelerator(s) can only be implemented using one of these methods. For example, if the hardware accelerator requires local data storage, or require additional bus interface, then it is obvious that the coprocessor interface method should be used. However, some of the hardware accelerator needs can be addressed by either Arm Custom Instructions or coprocessor interface methods and there is no clear boundary of when to use what. For example, a math function like sine/cosine, which might take less than 10 clock cycles, could use either method - both methods have their advantages:

- If using coprocessor method, the processor can interleave coprocessor offloaded instructions with other data operations. By allowing the processor to handle other workloads when the coprocessor is being used, we can achieve higher performance.
- If using Arm Custom Instructions, the same math function could be used by multiple tasks in a multi-tasking system, or by interrupt handlers, without worrying about access conflicts between tasks / handlers. If a coprocessor hardware is shared by multiple tasks, semaphores and context saving/restoring might be needed.

Software Support

A. Achieving the Goal of Avoiding Ecosystem Fragmentation

Software support is a critical part of the Arm Customer Instructions development. Earlier we mentioned that the instructions defined in CDE are restricted to data processing instructions only. This means that the compiler support for CDE instructions can be handled as intrinsic functions. And by introducing standardized intrinsic functions for CDE as a part of the Arm C Language Extension (ACLE, reference 1), CDE became a regular supported feature in all ACLE compliant tool chains without the need for any further customization. This allows us to meet the goal of avoiding fragmentation in the ecosystem and at the same time make it easier for software developers to take advantage of Arm Custom Instructions.

B. Intrinsic Functions

The details of the intrinsic functions defined in ACLE for supporting CDE is available on the Arm website (reference 2). Here we list the function prototypes:

Instruction	Intrinsic function
CX1	uint32_t __arm_cx1(int coproc, uint32_t imm);
CX1A	uint32_t __arm_cx1a(int coproc, uint32_t acc, uint32_t imm);
CX2	uint32_t __arm_cx2(int coproc, uint32_t n, uint32_t imm);
CX2A	uint32_t __arm_cx2a(int coproc, uint32_t acc, uint32_t n, uint32_t imm);
CX3	uint32_t __arm_cx3(int coproc, uint32_t n, uint32_t m, uint32_t imm);
CX3A	uint32_t __arm_cx3a(int coproc, uint32_t acc, uint32_t n, uint32_t m, uint32_t imm);

Table 6.1: Intrinsic functions for 32-bit integer CDE instructions

Instruction	Intrinsic function
CX1D	uint64_t __arm_cx1d(int coproc, uint32_t imm);
CX1DA	uint64_t __arm_cx1da(int coproc, uint64_t acc, uint32_t imm);
CX2D	uint64_t __arm_cx2d(int coproc, uint32_t n, uint32_t imm);
CX2DA	uint64_t __arm_cx2da(int coproc, uint64_t acc, uint32_t n, uint32_t imm);
CX3D	uint64_t __arm_cx3d(int coproc, uint32_t n, uint32_t m, uint32_t imm);
CX3DA	uint64_t __arm_cx3da(int coproc, uint64_t acc, uint32_t n, uint32_t m, uint32_t imm);

Table 6.2: Intrinsic functions for 64-bit integer CDE instructions

Instruction	Intrinsic function
VCX1	uint32_t __arm_vcx1(int coproc, uint32_t imm);
VCX1A	uint32_t __arm_vcx1a(int coproc, uint32_t acc, uint32_t imm);
VCX2	uint32_t __arm_vcx2(int coproc, uint32_t n, uint32_t imm);
VCX2A	uint32_t __arm_vcx2a(int coproc, uint32_t acc, uint32_t n, uint32_t imm);
VCX3	uint32_t __arm_vcx3(int coproc, uint32_t n, uint32_t m, uint32_t imm);
VCX3A	uint32_t __arm_vcx3a(int coproc, uint32_t acc, uint32_t n, uint32_t m, uint32_t imm);

Table 6.3: Intrinsic functions for 32-bit FPU CDE instructions

Instruction	Intrinsic function
VCX1	uint64_t __arm_vcx1d(int coproc, uint32_t imm);
VCX1A	uint64_t __arm_vcx1da(int coproc, uint64_t acc, uint32_t imm);
VCX2	uint64_t __arm_vcx2d(int coproc, uint64_t n, uint32_t imm);
VCX2A	uint64_t __arm_vcx2da(int coproc, uint64_t acc, uint64_t n, uint32_t imm);
VCX3	uint64_t __arm_vcx3d(int coproc, uint64_t n, uint64_t m, uint32_t imm);
VCX3A	uint64_t __arm_vcx3da(int coproc, uint64_t acc, uint64_t n, uint64_t m, uint32_t imm);

Table 6.4: Intrinsic functions for 64-bit FPU CDE instructions

Instruction	Intrinsic function
VCX1	uint8x16_t __arm_vcx1q_u8 (int coproc, uint32_t imm); T __arm_vcx1q_m(int coproc, T inactive, uint32_t imm, mve_pred16_t p);
VCX1A	T __arm_vcx1qa(int coproc, T acc, uint32_t imm); T __arm_vcx1qa_m(int coproc, T acc, uint32_t imm, mve_pred16_t p);
VCX2	uint8x16_t __arm_vcx2q_u8(int coproc, T n, uint32_t imm); T __arm_vcx2q(int coproc, T n, uint32_t imm); T __arm_vcx2q_m(int coproc, T inactive, U n, uint32_t imm, mve_pred16_t p);
VCX2A	T __arm_vcx2qa(int coproc, T acc, U n, uint32_t imm); T __arm_vcx2qa_m(int coproc, T acc, U n, uint32_t imm, mve_pred16_t p);
VCX3	uint8x16_t __arm_vcx3q_u8(int coproc, T n, U m, uint32_t imm); T __arm_vcx3q(int coproc, T n, U m, uint32_t imm); T __arm_vcx3q_m(int coproc, T inactive, U n, V m, uint32_t imm, mve_pred16_t p);
VCX3A	T __arm_vcx3qa(int coproc, T acc, U n, V m, uint32_t imm); T __arm_vcx3qa_m(int coproc, T acc, U n, V m, uint32_t imm, mve_pred16_t p);

Table 6.5: Intrinsic functions for 128-bit vector CDE instructions – multiple variants are available to support different vector types and optional predication feature

The vector CDE instructions are supported in Armv8.1-M processors supporting Helium technology. Because there can be different data types inside vectors, the intrinsic functions in Table 6.5 are polymorphic in the T, U and V types. Additional helper intrinsic functions are also introduced (also a part of ACLE) for converting vector types. Because Helium technology supports vector predication (conditional execution on a vector lane basis), Helium CDE intrinsic functions support the "_m" suffix (merging) which indicates that false-predicated lanes are not written to and keep the same value as they had in the first argument of the intrinsic (reference 6).

Today, Arm Custom Instructions is supported by Arm Compiler 6 which is available through Arm Development Studio and Keil® Microcontroller Development Kit (MDK) and GNU Compiler (GCC). Arm Custom Instructions will also be supported in future releases of IAR Embedded Workbench for Arm. Please contact IAR Systems® for details of the release schedule.

C. Handling of Multiple Instructions

Each CDE instruction supports an immediate data value which can be used to support multiple instructions in the same class. Arm Custom Instructions support the hardware extract of the immediate data value and passes it to the custom accelerator logic so that the accelerator can determine which operation is needed.

It is also possible to divide the immediate data field into two parts:

- One part to allow selection of multiple instructions, and
- Another part to be used as a parameter for the custom processing function

For example, assuming a custom instruction is created for an LCD controller application to map a pixel position {X, Y} into a memory address:

- Coprocessor ID is 1 in this example
- Return value is 32-bit
- A 3-bit "mode" parameter is needed to define the LCD's display mode

With such operation, we can use the CX3 instruction. Since the CX3 instruction supports 6 bits of immediate data, after taking 3 bits for the mode parameter, there are 3 bits left reserved for separating different custom instructions that also use CX3. To support software development, we can define some C macros as follows:

```
#define conv_address(X, Y, mode) __arm_cx3( 1, X, Y, ((0<<3) | (mode & 0x7)))

#define range_check(X, Y, mode)  __arm_cx3( 1, X, Y, ((1<<3) | (mode & 0x7)))
```

The custom accelerator logic receiving the immediate data value can determine if the instruction is the address conversion operation from bit 5 to bit 3 of the value and extract the mode parameter using bit 2 to bit 0.

Please note that since the mode parameter is encoded into the instruction, the parameter must be a compile time constant.

D. Check List for using Arm Custom Instructions / CDE features

To use the CDE intrinsic functions, software developers need to ensure that:

- The program code included the C header file <arm_cde.h>
- The CDE access for the corresponding coprocessor number is enabled. If the TrustZone security extension is used, the software developers need to make sure the CDE hardware is enabled for the correct security domain. More information about the registers to be programmed is listed below.
- Additional command line options are required to tell the toolchain which coprocessor number are allocated for Arm Custom Instructions / CDE feature.

To enable software to access CDE, the program code needs to take care of:

1. Coprocessor Access Control Register (SCB->CPACR)

This register enables access to Arm Custom Instructions / Coprocessor feature for each coprocessor number.

The CPACR bit assignments are:

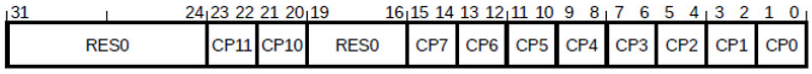

Figure 6.1:
Bit fields in CPACR

For each coprocessor number, two bits are allocated:

- 00 = disabled
- 01 = Privileged only
- 11 = Full access

The setting up of this register is always needed.

2. Non-secure Access Control Register (SCB->NSACR)

If TrustZone security extension is implemented, this register determines if Non-secure world is allowed to access an Arm Custom Instructions hardware unit / coprocessor unit. This register is accessible in Secure privileged state only.

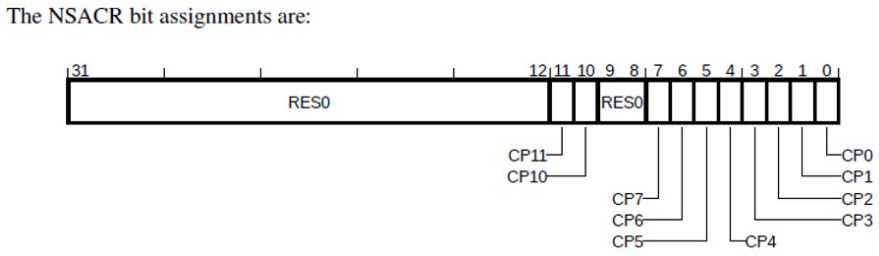

Figure 6.2:
Bit fields in NSACR

If a bit in NSACR is set to 0 (default), it is Secure access only. To enable Non-secure world access, the corresponding bit needs to be set to 1.

The setting up of this register is required only if the TrustZone security extension is used.

3. Coprocessor Power Control Register (SCB->CPPWR)

The SU[n] bits in this register defines for each Arm Custom Instructions / Coprocessor unit, whether it is allowed to enter a non-retentive power state (e.g. powered down for power saving). In the case of a coprocessor hardware which is outside the processor, it can have its only power domain and it can be powered down if its SU[n] bit is set to 1.

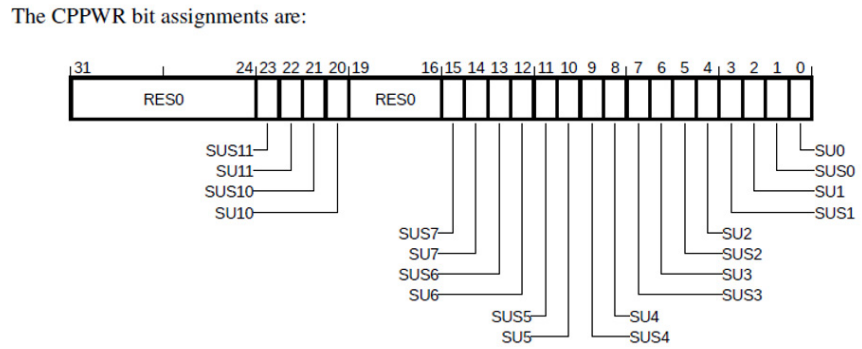

Figure 6.3:
Bit fields in CPPWR

If TrustZone security extension is implemented, the SUS[n] bits in this register defines whether the SU[n] bit is Secure access only (i.e. if set to 1, it is Secure only).

Because Arm Custom Instructions hardware in the Cortex-M33 processor uses the same power domain as the processor, it cannot be powered down when the processor is powered up. However, to access an Arm Custom Instructions hardware accelerator, the corresponding SU[n] bit in the CPPWR must still be 0.

The configuration of this register is optional because by default the SU[n] bits are 0, which means that they are not powered down.

4. Command Line Options

Because the encoding of CDE instructions and coprocessor instructions overlaps, tool chains need a way to know which coprocessor number(s) is assigned for Arm Custom Instructions. To address this requirement, new command line options are added to tool chains.

In Arm Compiler 6, the compiler (armclang, from release 6.14.1, reference 3) allows the CDE features to be specified. For example:

- "--target=arm-arm-none-eabi -march=armv8-m.main**+cdecpN**" or "--target=arm-arm-none-eabi -mcpu=cortex-m33**+cdecpN**", where N is in the range 0-7, for an Armv8-M / Cortex-M33 target with the Main Extension.
- "--target=arm-arm-none-eabi -march=armv8.1-m.main**+cdecpN**" or "--target=arm-arm-none-eabi -mcpu=cortex-m55 **+cdecpN**", where N is in the range 0-7, for an Armv8.1-M / Cortex-M55 target with the Main Extension.

The binary utility (fromelf) also has a new command line option, for example, for it to know how to disassemble instructions correctly:

- "--coprocN=value"
  - Where N is the coprocessor ID in the range of 0 to 7, and
  - value = cde (or upper case CDE). By default the coprocessor IDs are assigned as coprocessors (value = generic).

For example, to disassemble a program image called image.elf, the command line can be

"fromelf -c --cpu=8.1-M.Main.mve.fp --coproc0=cde image.elf"

Please note when using "--coprocN=cde", the CPU option (e.g. "--cpu=cortex-m33") must be used.

In GNU C compiler (GCC) similar options has been added (available in GCC 10, reference 4). For example:

- "-march=armv8-m.main+cdecpN –mthumb" (Cortex-M33 processor with Arm Custom Instructions)
- "-march=armv8-m.main+fp+cdecpN –mthumb" (Cortex-M33 processor with floating-point and Arm Custom Instructions)
- "-march=armv8.1-m.main+mve+cdecpN –mthumb" (Cortex-M55 processor with Arm Custom Instructions)

A number of commercial GCC based toolchains provide proprietary linkers and binary utilities. Please refer to the documentation for those tool chains for additional information about using Arm Custom Instructions with them.

Applications of Arm Custom Instructions

A. Overview

The Arm Custom Instructions feature can be used in a wide range of scenarios. For example:

- Specialized bit field processing:
  - Colour conversion in image processing
  - Population count (counting the number of bits set to 1 in a register).
- Acceleration of general data processing:
  - Cyclic redundancy check (CRC)
  - Mathematic functions (e.g. sine, cosine)
  - Functions using non-standard data types (e.g. 4-bit machine learning)

In contrast, the following examples would be more suitable to coprocessor implementation because they could benefit from having their own bus manager interface for accessing data in memories directly:

- Cryptographic engines that encrypt / decrypt blocks of data (e.g. AES engine)
- DSP data engines that process blocks of data (e.g. FFT engine)

To demonstrate what can be achieved, the Arm engineering team has prepared a number of examples, and prototyped some of them in FPGA prototypes. These examples are explained on pages 20-30.

B. Trigonometric Functions

1. Methods Investigated

Sine & Cosine trigonometric functions are common in motor control, power electronics and robotics applications, but are quite slow to evaluate. Typically, pre-computed values stored in a look-up table are linearly interpolated, or a Taylor polynomial is evaluated in software. These approaches take around 30 clock cycles for 32-bit fixed point numbers. This could be reduced significantly by evaluating them in a hardware using ACIs. There are many approaches for sine & cosine hardware designs, and in this evaluation, we focus on:

- Method 1: Taylor series expansion
- Method 2: CORDIC (COordinate Rotation DIgital Computer) methods

Both methods apply to fixed-point and floating point. It has been chosen to concentrate on 32-bit fixed point implementations with 16-bit result accuracy (+/-2.5e-5). These methods can be easily adjusted to suit specific precision requirements and PPA (Performance, Power, Area) trade-offs.

2. Taylor Series Expansion Method

For a 16-bit accurate result, the Taylor series expansion hardware need to be an 8th order approximation (or higher). The input (32-bit value, i.e. 0 to 2^32) is mapped to a full circle of the trigonometric function. In order to simplify the design, the hardware accelerator only process the first quadrant of the input data range:

- The input angle needs to be pre-processed (by hardware), and,
- The Taylor series expansion hardware ignores the top two bits of the input, and
- The output needs to be negated accordingly.

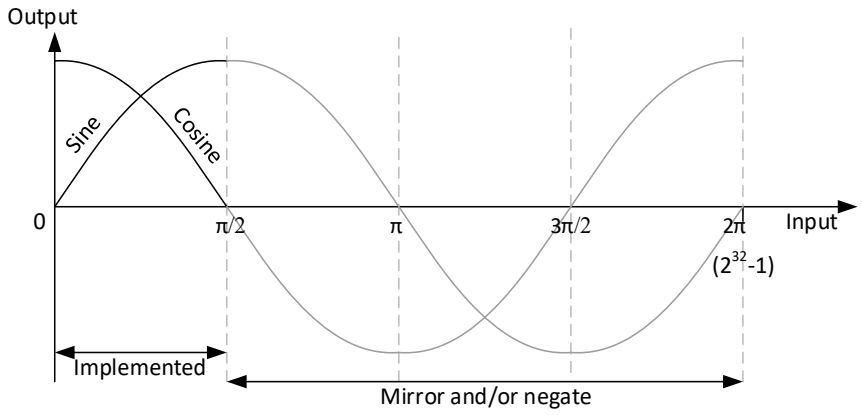

Figure 7.1:
The nature of Sine and Cosine functions means only the first quadrant is needed

The operations to be implemented were thus reduced to the following:

$$\cos\left(\frac{2\pi x}{2^{32}}\right) \approx 1 - k_2x^2 + k_4x^4 - k_6x^6 + k_8x^8 \qquad k_i = \frac{(2\pi)^i}{i!} \qquad x \in [0, (2^{30} - 1)]$$

$$\sin\left(\frac{2\pi x}{2^{32}}\right) = \cos\left(\frac{2\pi(2^{30} - x)}{2^{32}}\right)$$

In fact, the sine and cosine functions can also share the same hardware as the output values are mirrored within the quadrant. So, we have implemented the Cosine function only, which is able to obtain sine(θ) function result by calculating cosine(π/2 – θ).

Multipliers and flops can be reused across the 4 clock cycles, to save area further. A circuit diagram of the datapath synthesized for the Taylor series approach is shown in Figure 7.2.

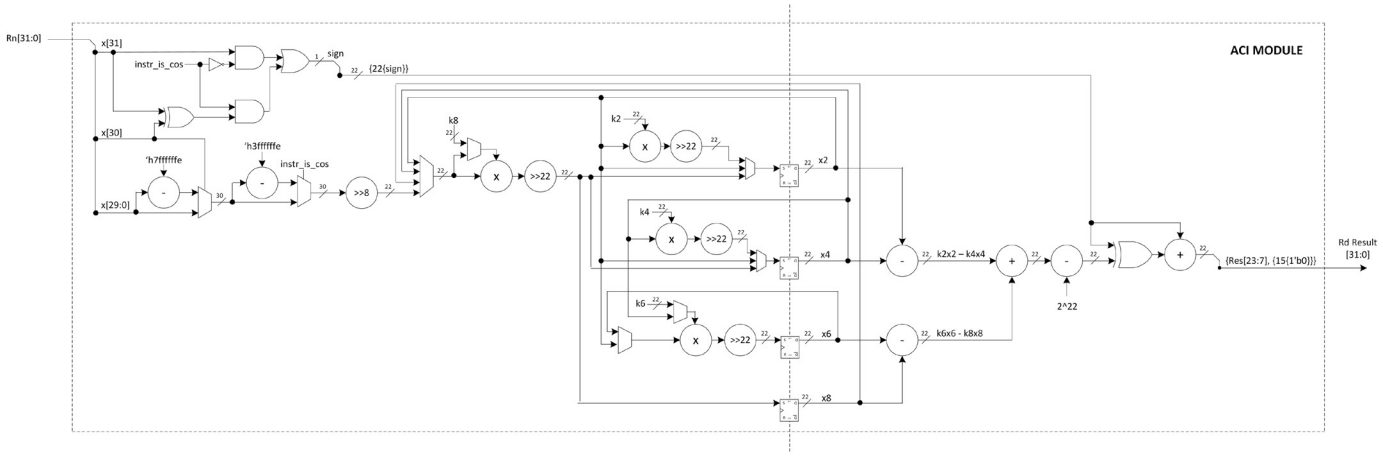

Figure 7.2: The design of the cosine/sine hardware accelerator based on Taylor series expansion

In an ACI prototype created with Cortex-M55, the hardware cost is about 14K gates. The synthesis trial is carried out with a 40LP 9 track library, and from the results the ACI module can run at 220MHz. If higher resolution is needed, the area of the hardware accelerator increases significantly because wider multipliers are needed and can end up slower – either running at a lower clock frequency, or change the design so that each multiplication is pipelined into two clock cycles.

3. CORDIC Method

The CORDIC (COordinate Rotation DIgital Computer) algorithm uses only additions, subtractions, bitshifts and a small LUT, meaning no multipliers are required. It operates by iteratively rotating an initial vector in one or the other direction by size-decreasing steps, until the desired angle has been achieved (usually taking 1 iteration per bit of precision). More details are available at https://www.mathworks.com/help/fixedpoint/ug/compute-sine-and-cosine-using-cordic-rotation-kernel.html.

When compared to the Taylor series method, the CORDIC method can be much smaller, but take a larger number of clock cycles. For an experiment, several implementations were tested:

- 1 iteration per cycle (19 iterations, 20 clock cycles) – ~3k gates
- 2 iterations per cycle (20 iterations, 11 clock cycles) – ~9k gates
- 3 iterations per cycle – this ended up slowing down the maximum clock frequency of the system and therefore was abandoned.

The CORDIC based designs behave differently from the Taylor series design in several ways:

- The input angle scales differently – Because CORDIC operation works only if the iteration converges, the input of the accelerator is designed to support input in the range of [-π/2, π/2]. Similar to Taylor series, calculation outside this range can be handled by additional logic. To simplify scaling, we use an input of [-4, 4] maps to input range of [-231, 231-1]. However, the actual supported input values are limited to [-π, π].
- Output is an integer in range [-231, 231-1] mapped to [-1,1] by 2's complement.

The internal data path is designed to be 24-bit. However, the design implemented 16-bit accuracy and the lowest bits are truncated at the output stage. The result is expressed as a 17-bit value including 1 sign bit.

The first implementation, producing 1 iteration per cycle is shown in Figure 7.3. The left-hand side logic detects which quadrant the input range is and then scale the input to the correct quadrant range. The arctan values are hardcoded and selected by the iteration counter, and then feed into the hardware that handles the iterated processing.

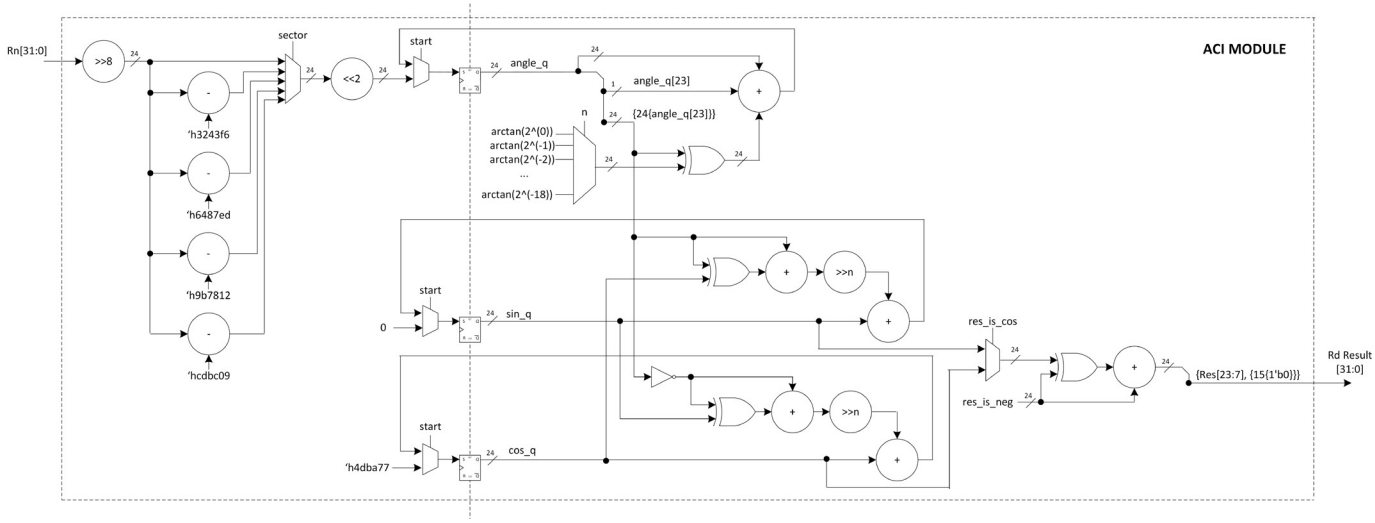

Figure 7.3: CODRIC hardware design implementation 1 – 1 iteration per cycle

After the first implementation was completed, a second implementation processing two iterations per cycle was designed and tested. The only difference is the duplication of processing hardware and the iteration count increment by 2 per cycle.

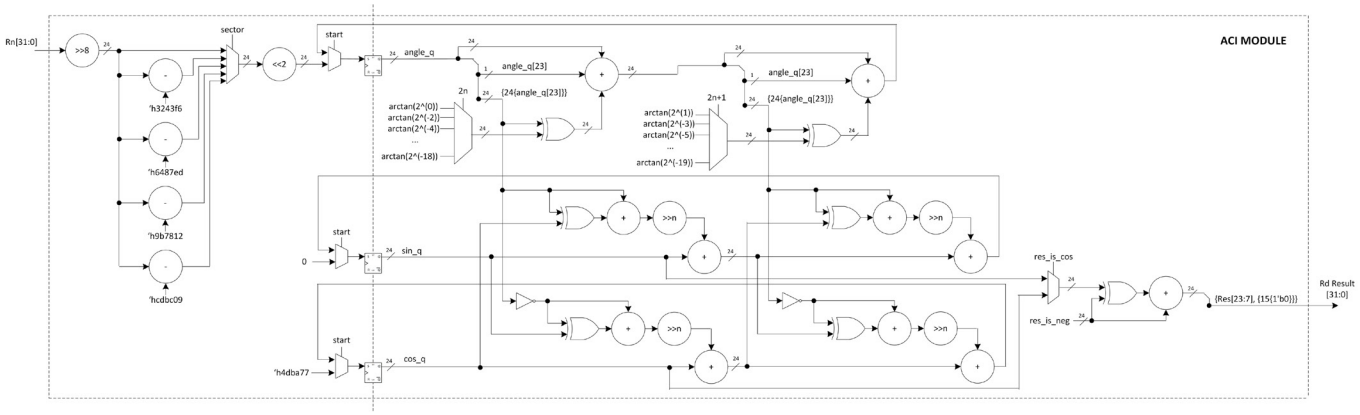

Figure 7.4: CODRIC hardware design implementation 2 – 2 iterations per cycle

Most of the possible input values has an output error of 0 or 1 (LSB). There are a few cases where the output error is slightly higher than 1 (LSB). Due to the nature of the algorithm, the hardware cannot compute when the input angle is zero, and therefore a special case is needed in the hardware logic to deal with this.

4. Performance Gains in the Trigonometric Functions

The use of hardware accelerators enables trigonometric functions to be carried out much quicker.

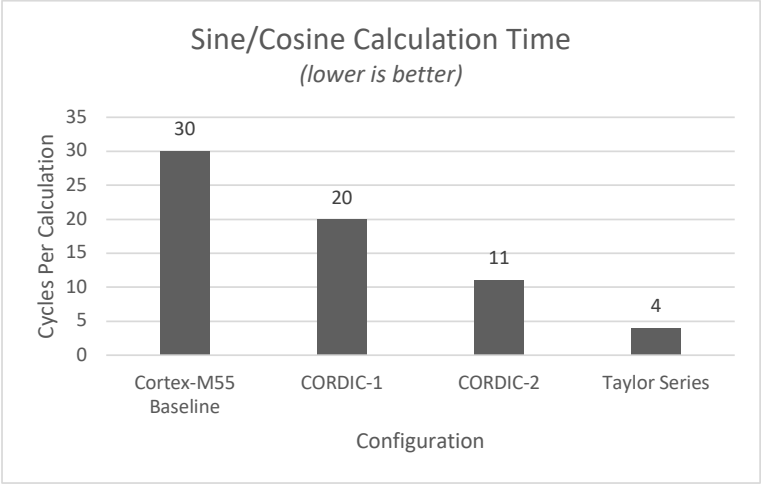

Figure 7.5: Sine/cosine hardware design performance summary

Not surprisingly, there is a trade-off between performance and area/power.

C. Image/Pixel Handling Functions

1. Overview

The Arm Custom Instructions feature supports multiple data types, including vector data if Helium technology is available on the processor. The ability to support vector data makes Arm Custom Instructions very attractive for image pixel manipulations (for example, in graphic user interface designs), because this can significantly improve the performance and user experiences.

Two areas of image processing algorithms have been investigated:

- Colour information conversions (RGB16 unpacking, packing)
- RGB16 Alpha blending

2. RGB16 Unpacking and Packing

Usually, images are represented by arrays of RGB pixels. In order to reduce data storage size, sometimes image pixels could be stored as 16-bit (RGB565, reference 5) and they need to be unpacked into individual colour channels for further processing before being displayed. Sometimes pixel data packing could also be needed for storage of computed images, or for display methods that do not support 8-bit colour scheme (RGB888 format).

Multiple types of packing and unpacking instructions have been developed. For unpacking, 12 variants of unpacking instructions are designed:

- 3 groups of unpacking instructions for the 3 colour channels (R, G and B)
- For each group, unpacked data can be in the lower 8 bit or upper 8 bit of the 16-bit space.
- Accumulative variant that reserves the other 8 bits unchanged, and non-accumulative variant that clears the other 8 bits.

In the unpack operations, each time 8 pixels are unpacked. Half of the destination vector register is either cleared to zero or unchanged. The additional bits can be useful later during processing (e.g. scaling by multiplication) to reduce the chance of overflow. The 12 unpacking instructions implemented with CDE is shown in Figure 7.6.

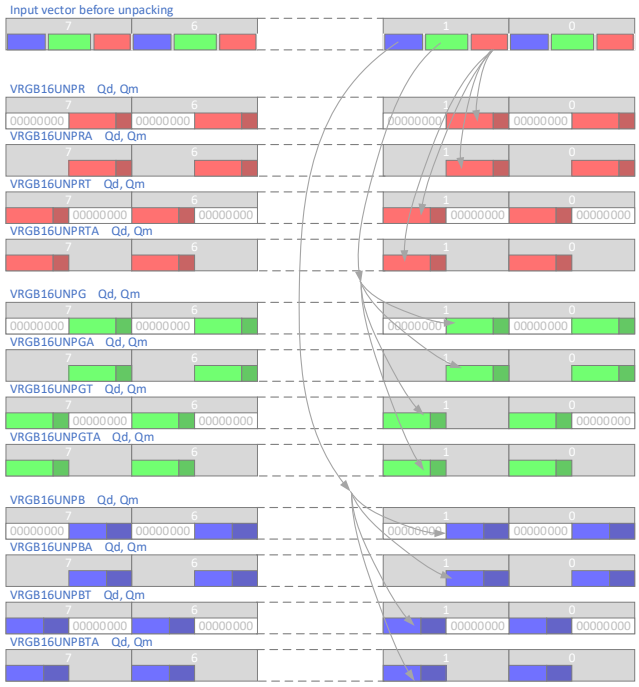

Figure 7.6: RGB unpack instructions

Pack operations can be viewed as the reverse of unpacking. There are a total of 8 variants of packing:

- 3 groups of packing instructions for each of the 3 colour channels (R, G and B), and
- 1 group of packing instructions that handles R, G and B input channels at the same time.
- For each group, there are two instructions: the input of the colour channel can be lower 8 bits or upper 8 bits

Unlike unpacking operations, all these instructions use the accumulative variant of CDE instructions because the other bits in the destination registers always have data from other colour channels after the packing is carried out.

The packing instructions for individual colour channels are shown in Figure 7.7:

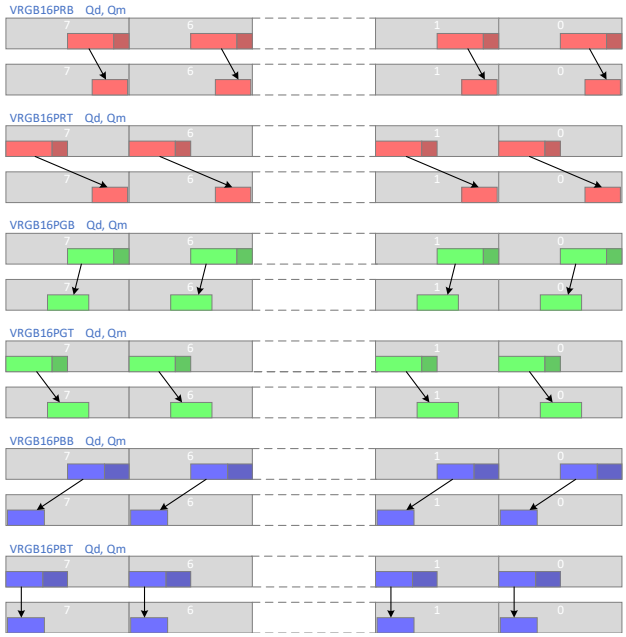

Figure 7.7:
RGB pack instructions for individual colour channels

The packing instructions that process all three colour channels at the same time are as follows (Figure 7.8):

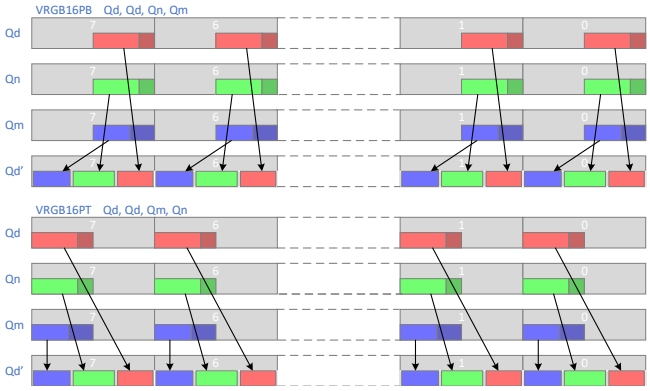

Figure 7.8:
RGB pack instructions for three colour channels at the same time

These packing and unpacking instructions allow the colour channels to be extracted, processed, and remerged efficiently. Intensive pixel manipulation operations like these packing and unpacking can be time-consuming in software, but can be implemented as custom vector instructions with minimal hardware cost.

3. Alpha Blending

Alpha blending operation allows two images to be merged based on a mixing ratio alpha (α):

$z = x * \alpha + y * (1 - \alpha)$

Where:

- x is the colour information from the 1st image,
- y is the colour information from the 2nd image,
- α is the mixing ratio alpha (range of 0 to 1), and
- z is the output image pixel

For each pixel, the R, G, and B colour channels are processed in the same way. The operation "VRGB16MIX Qd, Qn, Qm" is designed using one of the vector CDE instruction "VCX3A" to support the colour mixing where:

- Qd is a vector of 8 pixels from image 1, and is also the computed output
- Qn is a vector of 8 pixels from image 2
- Qm holds the mixing ratios (0 to 1 range is mapped in 0-127 unsigned short range)

The design of the mixing operation is as follows:

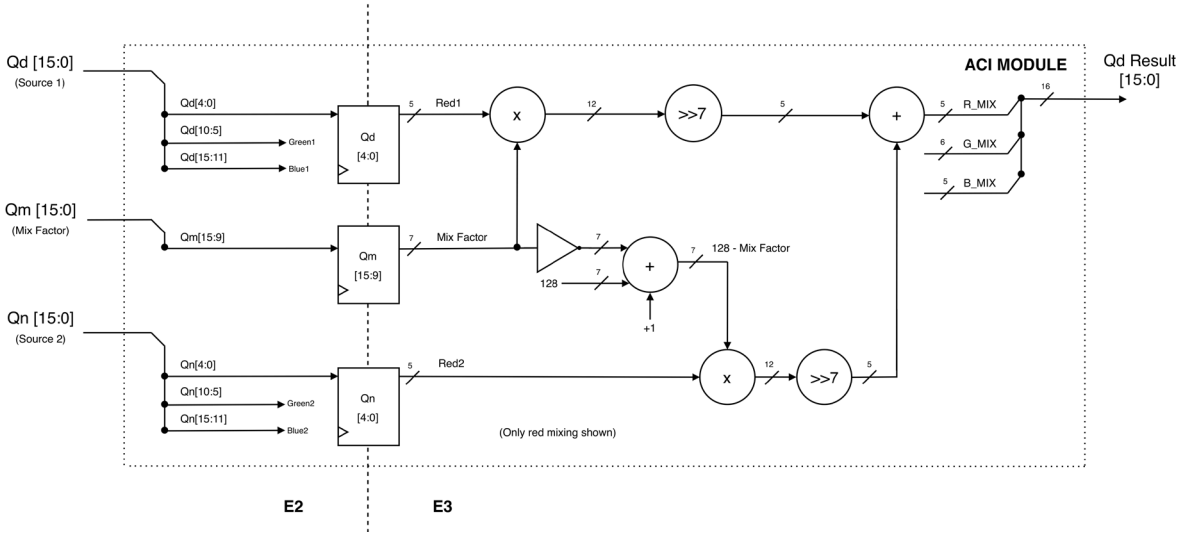

Figure 7.9: Alpha blending custom instruction module

Due to the presence of multiple single-cycle multipliers, the area cost of the design is about 8.69KGates.

4. Demonstration of Alpha Blending

After the alpha blending module is designed, it has been prototyped in a Cortex-M55 FPGA platform.

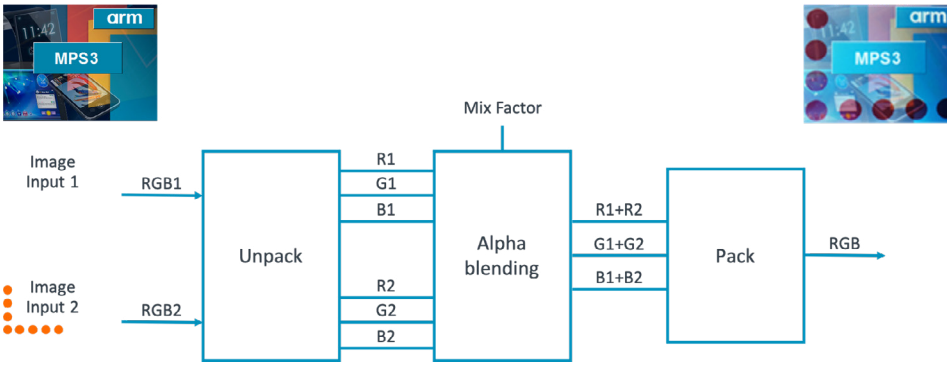

Figure 7.10: The images being used for alpha blending test

Due to the nature of the LCD interface on the FPGA board being used, a significant amount of the processor time is spent on transferring image data to the LCD module. However, the test result demonstrates that Helium optimized software can shorten the alpha blending processing time, and Arm Custom Instructions push this optimization further.

	Frames per minute	Processing time per frame (ms)	Image processing time excluding LCD interface (ms)
Processor running LCD interface code only (no alpha blending)	981	61.2	-
Processor running LCD interface code with alpha blending in optimized scalar code	471	127.4	66.2
Processor running LCD interface code with alpha blending in Helium optimized code	829	72.4	11.2
Processor running LCD interface code with alpha blending in Helium + CDE optimized code	928	64.7	3.5

Table 7.1: Comparison of alpha blending using various methods

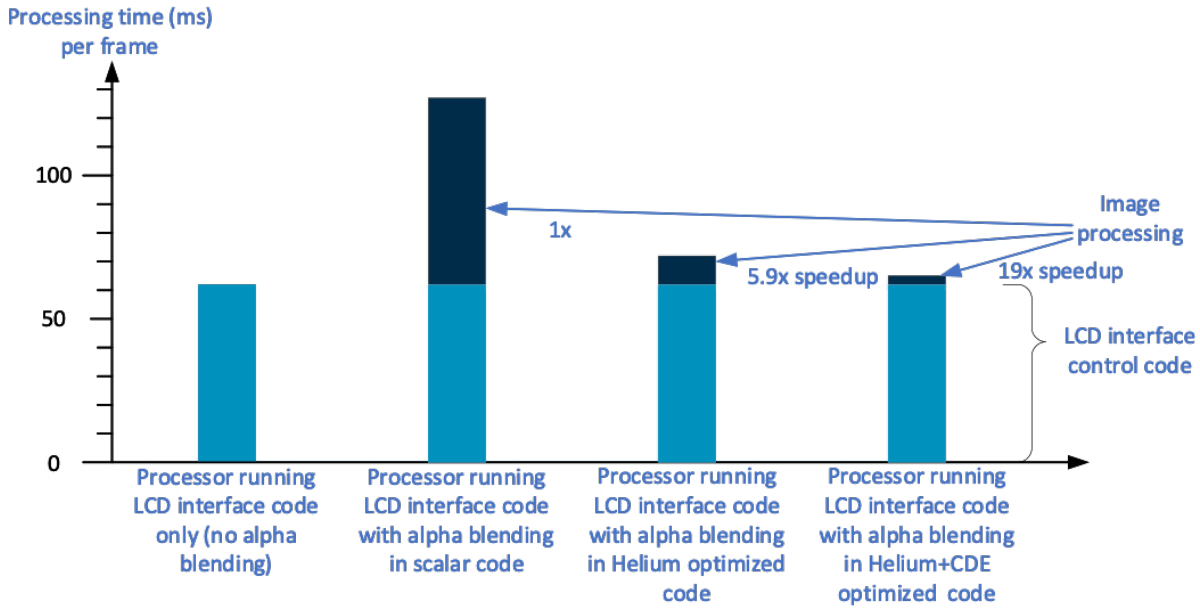

Figure 7.11: Comparison of the speedup of processing time per frame using various methods

In addition to speeding up processing, the use of custom instructions can also reduce the code size for the processing functions. While the code size reduction might not be significant when considering the size of the whole program image, reducing the code size in compute intensive operations might help reduce cache misses in the instruction cache for other parts of the application.

Conclusions

In summary, Arm Custom Instructions is now available in Arm Cortex-M33 and Cortex-M55 processors.

With the architecturally defined instructions in the Custom Datapath Extension (CDE) and standardized software interface using intrinsic functions defined in Arm C Language Extension (ACLE), software developers can make use of the custom defined instructions with standard tool chains, avoiding the fragmentation of the ecosystem for Arm development tools.

The design of the CDE also ensures that:

- software utilizing CDE features can be debugged,
- there is no impact to existing software (e.g. RTOS),
- there is no need to have full understanding of the processor's pipeline to integrate custom accelerator hardware,
- the architecture is scalable to multiple generations of Cortex-M processors.

Arm Custom Instructions can coexist with existing coprocessor instructions and complement each other. While Arm Custom Instructions can offer custom processing operations with very low latency, in some application scenarios, the coprocessor interface features could be more suitable.

References

[1] Arm C Language Extension (ACLE) specifications
https://developer.arm.com/architectures/system-architectures/software-standards/acle

[2] Custom Datapath Extension (CDE) in Arm C language Extension (ACLE)
https://developer.arm.com/documentation/101028/0012/9--Custom-Datapath-Extension

[3] Armclang command line options
https://www.keil.com/support/man/docs/armclang_ref/armclang_ref_chr1392632801932.htm#/chr1392632801932__custom_datapath_extension

[4] GNU C Compiler Arm options
https://gcc.gnu.org/onlinedocs/gcc/ARM-Options.html

[5] RGB555
http://www.barth-dev.de/online/rgb565-color-picker/

[6] M-profile Vector Extension (MVE) intrinsics
https://developer.arm.com/documentation/101028/0012/14--M-profile-Vector-Extension--MVE--intrinsics