Arm Custom Instructions: Enabling Innovation and Greater Flexibility on Arm

Lauranne Choquin, Staff Information Developer, Arm
Fred Piry, Lead Architect and Fellow, Arm

February 2020

Today, the Arm architecture enables developers to write high-performance portable code, capable of running unmodified on billions of devices. In contrast, the decline of Moore's law, combined with ever increasing demands for computational performance at the edge, lead to a need for product customization and specialization. To address these challenges, Arm Custom Instructions offer implementers the ability to make tradeoffs specific to their own goals.

This white paper introduces Arm Custom Instructions, which enable further innovation with Arm-based system designs, highlighting their features and benefits.

What are Arm Custom Instructions?

Arm Custom Instructions enable chip designers to push performance and efficiency further by adding application domain-specific features in small embedded processors, while maintaining the ecosystem advantages of Arm processors. Arm Custom Instructions are currently available for the Cortex-M33 processor. In 2021, Arm Custom Instructions will be enabled on the Cortex-M55 processor. In this context, Arm Custom Instructions aim to:

- Enable differentiation by giving you the power to innovate within the proven Arm architecture, a worldwide standard.
- Reduce time to market when exploring new classes of user-defined instructions for emerging algorithms and applications.
- Develop a domain-specific architecture by allowing you to implement a customized accelerator with an Arm architecturally compliant CPU as a container.

One way to categorize accelerators based on the connection to the CPU is:

1. Memory-mapped accelerators, such as a GPU, directly connected to the memory bus.
2. Coprocessor interface, recently introduced on the Arm Cortex-M33 processor and for future Cortex-M processors including the Cortex-M55, enables you to build closely coupled accelerators under the direct control of the CPU.
3. Arm Custom Instructions further expand this view of hardware accelerators by enabling tightly coupled accelerators with even closer coupling with the datapath of the processor.

Figure 1. Arm Custom Instructions introduce the ability to add tightly coupled accelerators to Arm Cortex-M CPUs

1. Memory-mapped	2. Coprocessor	3. Tightly coupled
• Decoupled from CPU	• Integrated with CPU	• Tightly Integrated with CPU
• Can have wide register size and direct access to memory	• Additional customer registers	• Access to standard registers
• Not limited by memory bandwidth of CPU	• High-throughput interface with CPU (64 bits per cycle)	• Ideal for low-latency operations (1 to 3 cycles)
• Runs in parallel to CPU	• Ideal for medium latency operations (3 cycles or more)	• Instructions interleaved with Arm standard instructions
	• Runs in parallel to CPU	

How do Arm Custom Instructions Work?

Arm redefines the coprocessor Instruction Set Architecure (ISA) encoding space to enable custom instructions using the Arm architectural registers and flags. You can use this encoding space to add your own, differentiating data processing instructions without compromising performance.

Arm Custom Instructions add a customizable module inside the processor. This module is driven by the pre-decoded instructions and shares the same interface as the standard Arithmetic Logic Unit (ALU) of the CPU. This configuration space enables you to design your own operations where:

- The CPU manages control and dependencies.
- Instructions are either single-cycle or multi-cycle and can be pipelined.

Implementations might contain one configuration space for the CPU and its general purpose registers, and one configuration space for the Floating-Point Unit (FPU) and its floating-point registers. purpose registers, and one configuration space for the FPU and its floating-point registers.

There are multiple regions of the encoding space available for customization. You can choose how many regions to use, up to eight, based on the type of instructions you want to implement. For the regions that are not used, Arm decodes the instruction either for the coprocessor interface, if present and enabled, or as a NOCP exception, if not present or not accessible.

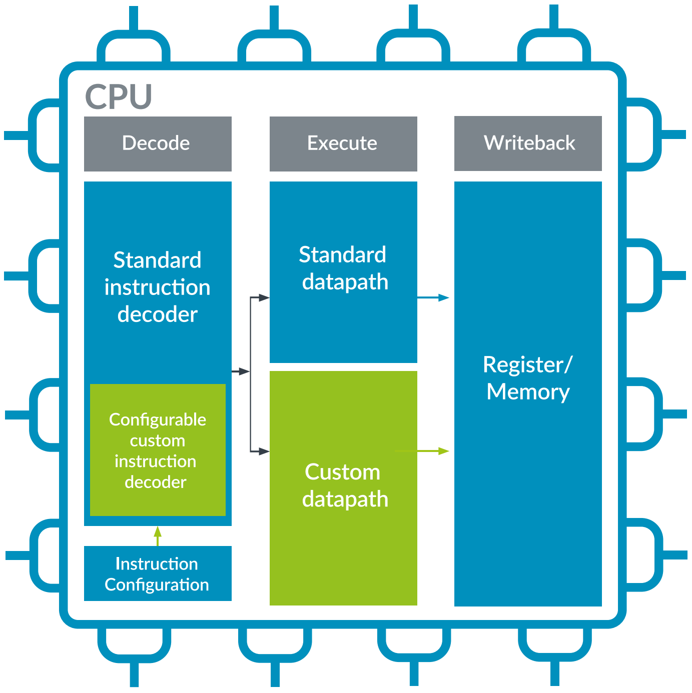
CPU
Decode
Execute
Writeback
Standard instruction decoder
Configurable custom instruction decoder
Instruction Configuration
Standard datapath
Custom datapath
Register/ Memory

Figure 2. Arm Custom Instructions configuration space

Adding custom instructions to a customizable CPU requires two steps:

1. Providing a configuration file that lists the regions you want to use for adding your own custom instructions.
2. Building the datapath for your own custom instructions and integrating it into the configuration space.

Decoding logic is automatically configured to decode your custom instructions and control your custom datapath. On top of the decoder, the CPU resolves all required control signals, instruction interlocking, and data dependencies.

The custom ALUs follow the execution resources available on the customizable CPU:

- Operating out of the extended register file, that is registers in the Floating-Point Unit (FPU) or M-profile Vector Extension (MVE), is only possible on a CPU that implements the extended register file.
- The support for multi-cycle instructions matches the supported latencies in the customizable CPU.

Arm provides all required control signals and operands, and writes results into the register file for the custom datapath. Arm control logic handles all hazarding logic. As a result, any declared required operand or flag, and any declared result write, requires the appropriate hazarding to be handled, even if not used by the custom instructions.

For CPUs with a coprocessor interface, the encoding space for each coprocessor can be dedicated to either the external coprocessor or the customizable ISA extension, with mutual exclusion.

Based on the configuration file you provided, Arm configures the instruction decoder and provides all control logic to drive your custom datapath. Arm also verifies all the control logic interlocks and forwarding. You design and verify the custom datapath.

Arm provides a set of assertions and properties to check compliance with the custom datapath interface protocol.

Arm provides a testbench to verify the integration of the custom instructions into the customizable CPU. You develop the integration test suites to execute the custom instructions and check correctness.

Which Custom Instructions are Now Enabled?

Arm introduces 2 × 3 classes of instruction extension in the coprocessor instruction space:

- Three classes operate on the general-purpose register file, including the condition code flags APSR_nzcv.
- Three classes operate on the floating-point/Single Instruction Multiple Data (SIMD) register file only.

The three classes are defined by the following instruction patterns:

```
<operation code> <destination register>
<operation code> <destination register>, <source register>
<operation code> <destination register>, <source register 1>, <source register 2>
```

The destination register or the destination register pair of an instruction might be read, as well as written (non-accumulator and accumulator variants).

The operation code can be split between a true operation code in the custom datapath and an immediate value used in the custom datapath.

Immediate consequences of the above are:

- No operations on the floating-point registers can set condition codes.
- There are no operations using registers from both register files.

Operations on the general-purpose register file operate on 32-bit registers, or a dual-register consisting of a 64-bit value constructed from an even-numbered, general-purpose register and its immediately following odd pair.

Instruction	Assembly	Inputs	CPU	Imm	Outputs
CX1 {A}	CX1{A} Pn, Rd, #imm	Immediate and 1x 32-bit GPR/ NZCV {same as output}	M33, M55	13b	1x 32-bit GPR or NZCV
CX2 {A}	CX2{A} Pn, Rd, Rn, #imm	Immediate and 2x 32-bit GPR/ NZCV {one same as output}	M33, M55	9b	1x 32-bit GPR or NZCV
CX3 {A}	CX3{A} Pn, Rd, Rn, Rm, #imm	Immediate and 3x 32-bit GPR/ NZCV {one same as output}	M33, M55	6b	1x 32-bit GPR or NZCV
CX1D {A}	CX1D{A} Pn, Rd, Rd+1, #imm	Immediate and 1x 32-bit GPR/ NZCV {two same as output}	M33, M55	13b	2x 32-bit GPR
CX2D {A}	CX2D{A} Pn, Rd, Rd+1, Rn, #imm	Immediate and 2x 32-bit GPR/ NZCV {two same as output}	M33, M55	9b	2x 32-bit GPR
CX3D {A}	CX3D{A} Pn, Rd, Rd+1, Rn, Rm, #imm	Immediate and 3x 32-bit GPR/ NZCV {two same as output}	M33, M55	6b	2x 32-bit GPR

Table 1. General-purpose registers and NZCV flags

Instruction	Assembly	Inputs	CPU	Imm	Outputs
VCX1{A}.F	VCX1{A}.F Pn, Sd, #imm	Immediate and 1x 32-bit fp32 register {same as output}	M33, M55	11b	1x 32-bit fp32 register
VCX2{A}.F	VCX2{A}.F Pn, Sd, Sn, #imm	Immediate and 2x 32-bit fp32 register {one same as output}	M33, M55	6b	1x 32-bit fp32 register
VCX3{A}.F	VCX3{A}.F Pn, Sd, Sn, Sm, #imm	Immediate and 3x 32-bit fp32 register {one same as output}	M33, M55	3b	1x 32-bit fp32 register
VCX1{A}.D	VCX1{A}.D Pn, Dd, #imm	Immediate and 1x 64-bit fp64 register {same as output}	M55	11b	1x 64-bit fp64 register
VCX2{A}.D	VCX2{A}.D Pn, Dd, Dn, #imm	Immediate and 2x 64-bit fp64 register {one same as output}	M55	6b	1x 64-bit fp64 register
VCX3{A}.D	VCX3{A}.D Pn, Dd, Dn, Dm, #imm	Immediate and 3x 64-bit fp64 register {one same as output}	M55	3b	1x 64-bit fp64 register
VCX1{A}.Q	VCX1{A}.Q Pn, Qd, #imm	Immediate and 1x 128-bit vector register {same as output}	M55	12b	1x 128-bit vector register
VCX2{A}.Q	VCX2{A}.Q Pn, Qd, Qn, #imm	Immediate and 2x 128-bit vector register {one same as output}	M55	7b	1x 128-bit vector register
VCX3{A}.Q	VCX3{A}.Q Pn, Qd, Qn, Qm, #imm	Immediate and 3x 128-bit vector register {one same as output}	M55	4b	1x 128-bit vector register

Table 2. FPU/M-Profile Vector Extension (MVE) registers

For each class, the imm field can be partitioned between what operation is being requested and a constant immediate value that the operation can consume.

Example in practice

Consider a population count function.

```
int popcount(uint32_t x) {
     int n = 0;
     for (int i = 0; i < 32; ++i) {
          n += (x >> i) & 1;
     }
     return n;
}
```

A standard, hand-optimized Arm implementation would look like this:

```
MOV.W    r1, #0x55555555
AND.W    r1, r1, r0, LSR #1
SUBS     r0, r0, r1
MOV.W    r1, #0x33333333
AND.W    r1, r1, r0, LSR #2
BIC      r0, r0, #0xCCCCCCCC
ADD      r0, r1
MOV.W    r1, #0x01010101
ADD.W    r0, r0, r0, LSR #4
BIC      r0, r0, #0xF0F0F0F0
MULS     r0, r1, r0
LSRS     r0, r0, #24
```

This could be replaced with a single user-defined instruction that can be implemented in one cycle:

```
CX1A p0, r0, #0 // population in r0, return r0
```

Such an instruction can be used either directly in a specialized library or be added as an intrinsic instruction within C code. All standard compilers conform to a future release of Arm C Language Extension (ACLE) will support intrinsic functions for user-defined instructions.

What are the Benefits of Arm Custom Instructions?

The possibility to add built-in instructions into the CPU provides significant benefits for some use cases.

Arm introduces an extended datapath and instructions pre-decoded by the decode function. The CPU manages Read/Write registers, as well as control and dependencies.

The set of custom instructions is defined in the Arm architecture, therefore, all major compilers support it. Also, because the encoding of the custom instructions is integrated with tools, such as compilers and debuggers, you do not need to develop or distribute any special version of the tools.

Arm Custom Instructions bring three main benefits:

- Performance boost for low-latency instructions.
- Easy integration with the existing software ecosystem.
- Scalability across Arm Cortex-M CPUs.

Although a strictly compatible interface across multiple CPUs might not be achievable, Arm aims to maintain consistency between different implementations to simplify porting accelerators across the Arm Cortex-M portfolio.

Innovate with Greater Flexibility and Differentiation

The growth of on-device processing means that optimization and the use of accelerators are key. Arm Custom Instructions offer more flexibility to innovate within the Arm worldwide standard, by integrating an accelerator for specific use cases, using the CPU as a container. Arm Custom Instructions allow you to customize your Cortex-M33 and Cortex-M55 (available in 2021) CPU further by adding your own data processing instructions, while boosting the performance of your CPU. With Arm Custom Instructions, you have more flexibility to innovate within the Arm worldwide standard at your own pace. Arm Custom Instructions ensure easy integration with the existing software ecosystem and are scalable across Arm Cortex-M CPUs.

Find more information on Arm Custom Instructions here. Alternatively, if you have any questions, get in touch with an Arm expert here.

arm