AN14650

SmartDMA Cookbook

Rev. 1.0 — 28 May 2025 Application note

Document information

Information	Content
Keywords	AN14650, SmartDMA, EZH, cookbook, MCX MCU, LPC5500
Abstract	This application note primarily introduces the internal architecture, main functions, and features of EZH or SmartDMA, and finally lists the usage and meanings of the main instructions.

1 Introduction

This application note primarily introduces the internal architecture, main functions, and features of EZH or SmartDMA, and finally lists the usage and meanings of the main instructions. EZH (SmartDMA) is an NXP unit that efficiently manages repetitive and event-driven tasks, reducing the workload on Arm cores. It significantly improves system performance in input and output (I/O) handling, data processing, and real-time operations.

1.1 What is EZH (SmartDMA)?

EZH, also known as SmartDMA, is primarily aimed as a complementary unit to the Arm core within NXP devices.

Rather than replacing the Arm, the primary objective of EZH is to offload tasks from the Arm processor, and optimize performance and efficiency.

EZH provides significant advantages over Arm processing, specifically in:

- **Fast response**: It provides a quick reaction to input and output (I/O), and Boolean events.
- **Efficiency at lower frequencies**: It offers improved performances as EZH eliminates interrupt delays.
- **Expanded instruction set**: It uses a 32-bit instruction set offering broader functionality to the 16-bit thumb instruction set.
- **Advanced extension**: It includes built-in innovative functionalities, such as heartbeat, timer, and supervised execution.
- **Concurrent data access**: It implements a dual AHB bus controller, allowing simultaneous data read and code execution.

EZH ensures that the processor maintains all advantages of Arm, enhancing its performance in specific operational tasks.

1.2 Advantages of choosing EZH

EZH effectively manages repetitive and event-driven I/O operations. It significantly reduces the workload of Arm processors for essential and routine tasks. Table 1 illustrates the advantages of EZH over Arm Cortex-M0+.

Table 1. Comparison between EZH and Arm Cortex-M0+

Functions	EZH	Cortex-M0+
Architecture	Modified Harvard architecture with dual AHB bus (Instruction and Data)	Von Neuman architecture with single system bus
Event latency	Bit-slice operation (no cycle penalty) for I/O event	ISR latency (15 CPU cycles with 0-waitstate memory)
I/O access	Register-based I/O access using 3 core registers — GPO/GPI/GPD, supported by intrinsic instructions.	Memory-based I/O access over AHB; one extra cycle needed for load or store (LDR/STR) operations.
State machine	Intrinsic instructions for state machine, `E_VECTORED_HOLD`	
Determinate execution	Determinate execution with intrinsic instructions like: `E_HEART_RYTHM`,`E_SYNCH_ALL_TO_BEAT`, and `E_WAIT_FOR_BEAT`.	
Hardware circle loop	0-waste hardware circle loop supported by intrinsic instruction	Circle loop execution with a one cycle penalty.

1.3 Typical EZH applications

EZH (SmartDMA) is beneficial in scenarios requiring efficient data manipulation, I/O handling, and real-time operations. The following are some typical use cases:

- **SmartDMA applications**:

Regular DMA is limited to data movement between memory locations without a main processor involved. EZH, working as SmartDMA, can process data or convert data during data transmission. For example, it can convert RGB565 to RGB888.

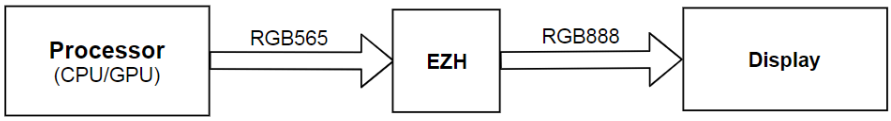

Figure 1. EZH perform RGB565 to RGB888 acting as SmartDMA

There are various data formats for bitmap images; two common ones are RGB565 for 16-bits color and RGB888 for real-color. In microcontroller, RGB565 is commonly used in image processing due to its lower memory usage and less computation effort. A challenge arises when rendering RGB565 to display, as some display interfaces; especially those with 8-bit parallel interfaces like the INTEL 8080 do not support RGB565 rendering. In a legacy system, the CPU first converts the data from RGB565 to RGB888, and then the DMA transfers the converted image to the display interface. With EZH, RGB565 data can be rendered directly to the display without CPU involvement, as EZH performs the data format conversion during transmission.

- **Boolean detection and general I/O handling:**

In applications like key scanning, EZH uses bit-slice operation for key detection. Its register-based I/O handling accelerates key scan response time and enhances sensitivity.

- **Protocol emulation over I/O**:

EZH can be used to simulate serial bus protocols, such as USART, I2C, SPI, ISO7816, and PWM.

- **Shift-based algorithm calculation**:

EZH uses shift-based intrinsic instructions to efficiently execute algorithm operations, such as CRC and SHA.

- **Large-scale data manipulation**:

EZH can be used for data format conversion, serving as a data preprocessing engine. While processor synchronization is required sometimes, it is not a big concern for big data processing. For example, in an image-processing application, the image comes as a bitmap array. However, most algorithm must divide the image into blocks and process them sequentially using convolution or filter. During image preprocessing, EZH can convert a two-dimensional image block into a vector for further calculation, as vector calculation is accelerated in most modern processors.

- **State machine control and streaming GPIO**:

EZH is effective at controlling a state machine and streaming GPIO. It also provides powerful boolean detection, external event monitoring, and single cycle GPIO operation. These capabilities allow EZH to handle tasks ranging from simple applications like traffic light control to more complex implementations such as sophisticated I3C protocol emulation.

- **Multiple quadrature encoders or decoder emulation over IO**:

In polling mode, EZH can detect the phase A and B signals to output the direction and counter value of the encoder. It also supports two or more encoders.

2 EZH Architecture and function description

EZH acts as an AHB controller in a dual-bus system, enabling efficient data handling with minimal CPU load. It also includes bit-slice logic, lightweight processor core, and interfaces for AHB, GPIO, and interrupts, supporting quick detection and real-time control.

2.1 EZH in system

EZH functions as an AHB controller in the system within a dual bus architecture. The instruction bus fetches instructions, while the data bus fetches read-only data (RO) and read or write data (RW).

As an AHB controller device, the AHB matrix priority register in the SYSCON or SYSCTL module can prioritize EZH bus access. See the figures below: Figure 2, Figure 3, and Figure 4

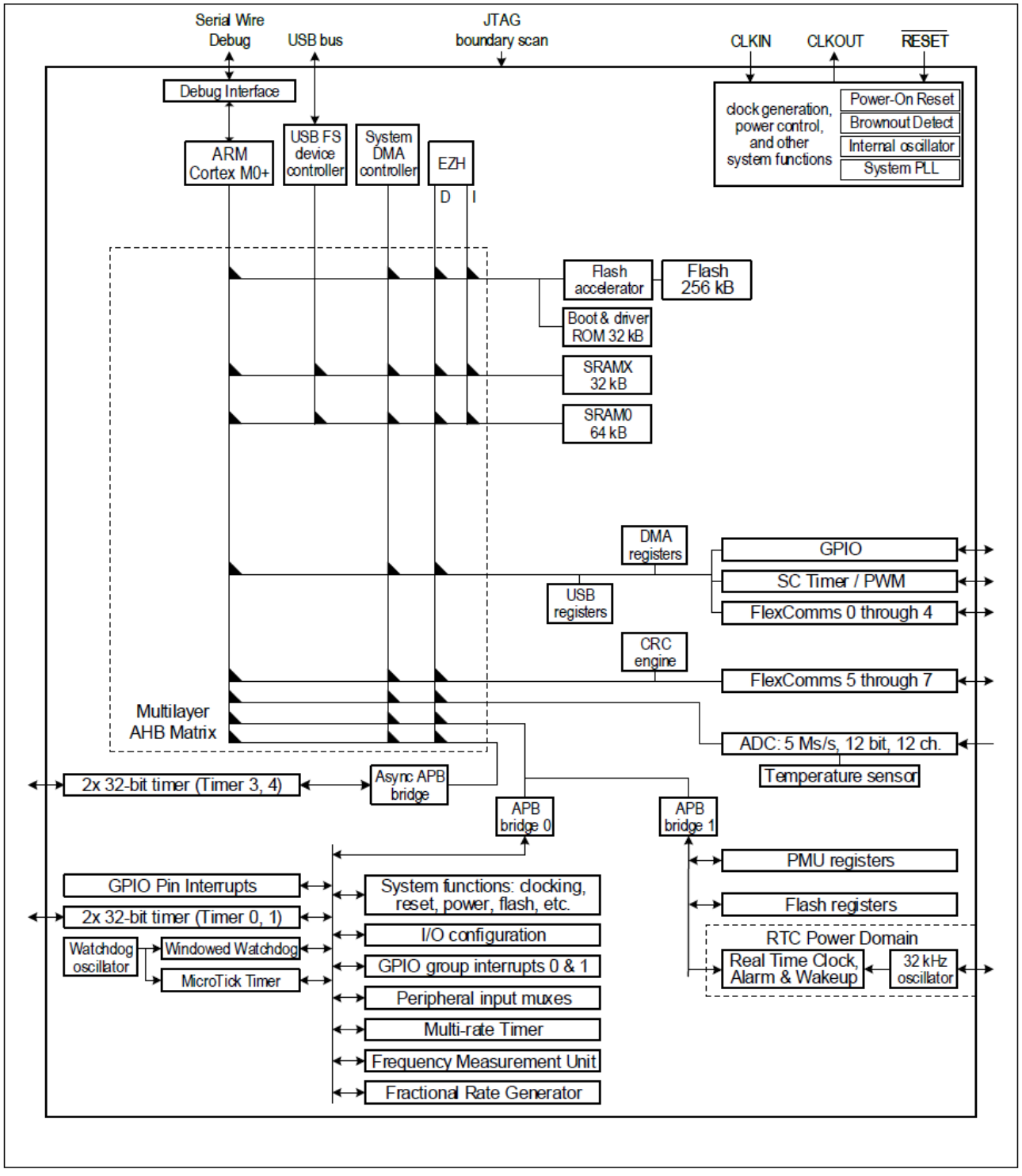

Figure 2. LPC51U68 block diagram

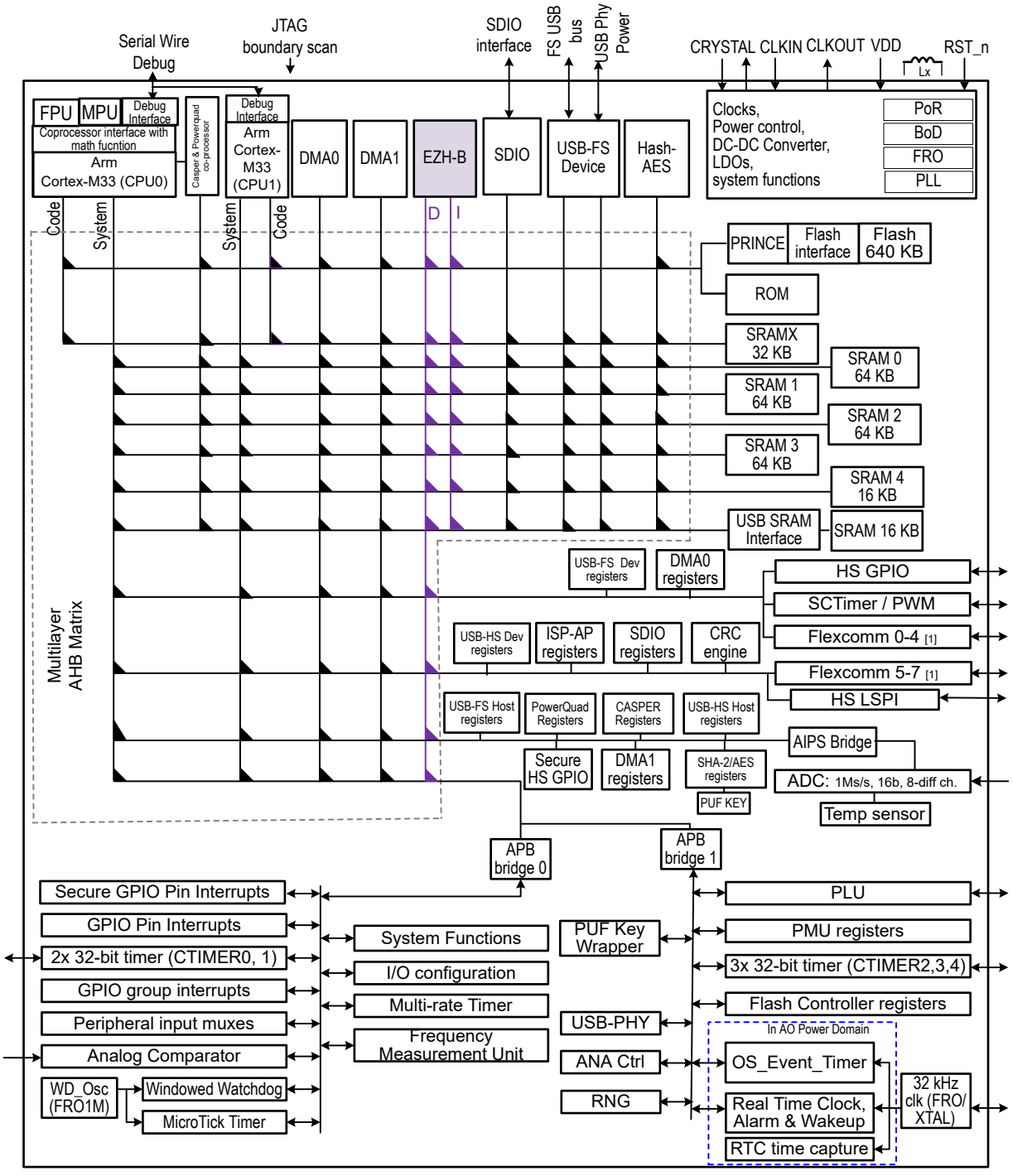

Figure 3. LPC55S6x block diagram

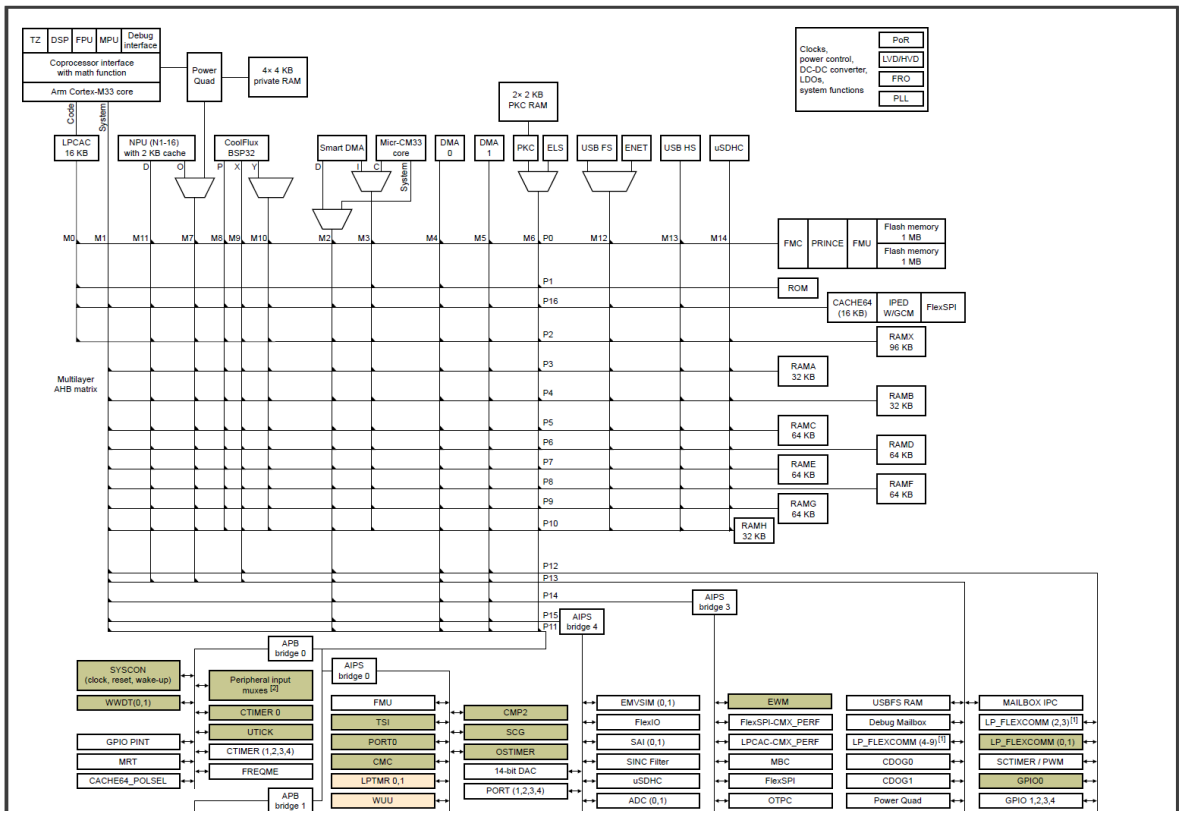

Figure 4. MCXNx4x block diagram

2.2 EZH architecture

As a peripheral, Figure 5 is as follows:

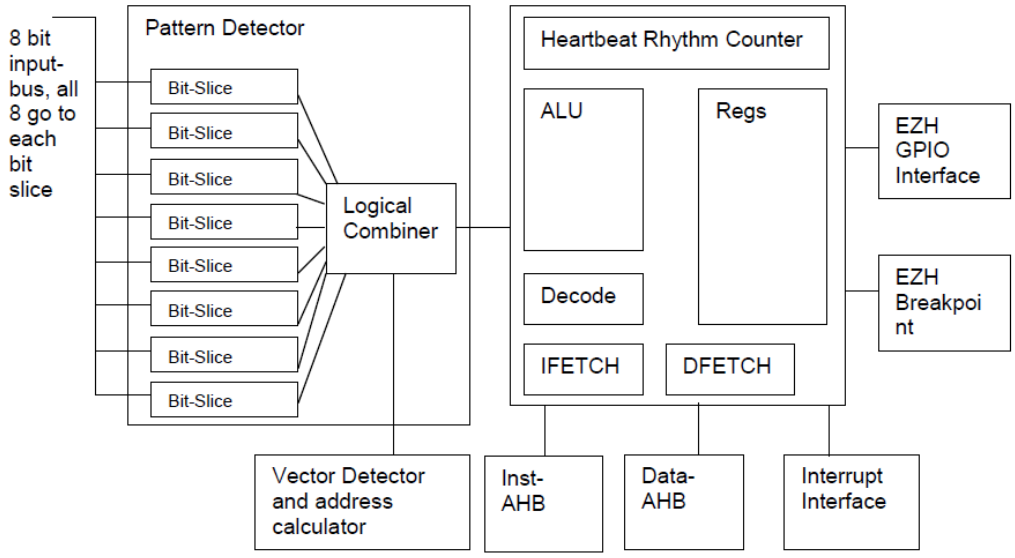

Figure 5. EZH architecture

3 EZH registers

EZH peripherals provide a set of control and internal registers for configuration and operation. These registers are mapped on the APB and are accessible through a defined base address specific to each supported MCU.

3.1 EZH peripheral registers

EZH controller registers are placed on the APB bus. Table 2 provides the base addresses of EZH controller registers in the memory map.

Table 2. EZH controller base address

Part number	EZH control register base address
LPC5410x	0x4004C000
LPC54114/LPC51U68	0x4001D000
LPC55(S)6x/LPC55(S)2x	0x4001D000
LPC55(S)3x	0x4001D000
IMXRT500	0x40027000
MCXNx4x/MCXN23x	0x40033000

EZH controller registers are listed and explained in Table 3.

Table 3. EZH peripheral registers

Name	Access	Offset	Description	Reset Value
EZHB_BOOT	R/W	0x20	32-bit boot address. The boot address must be aligned to a 4-byte boundary.	0x00
EZHB_CTRL	R/W	0x24	• B31:16 - Must be set to 0xC0DE. • B15: 5 - Reserved (must be 0). • B4 - Must be set to 1 to enable AHB synchronization. • B3 - Selects bufferable or nonbufferable mode for AHB write (if AHB buffer is available). • B2 - Ignore AHB bus error. If set to 0, the AHB bus triggers an emergency. • B1 - External flag, which influences the EX/NEX condition logic—when set to 1, it satisfies the EX condition, and when cleared to 0, it satisfies the NEX condition. • B0 - Start bit, which acts as an ignition signal — when set, it triggers the EZH to begin executing instructions.	0x00
EZHB_PC	RO	0x28	EZH program counter	0x00
EZHB_SP	RO	0x2C	EZH stack pointer	0x00
EZHB_BREAK_ADDR	R/W	0x30	32-bit address to EZHB_BREAK_VECT location	0x00
EZHB_BREAK_VECT	R/W	0x34	User debug routine	0x00
EZHB_EMER_VECT	R/W	0x38	Emergency code routine	0x00
EZHB_EMER_SEL	R/W	0x3C	• B9 - Software emergency request • B8 - Enables the software emergency mechanism • B7:0 - Reserved and must be written as 0.	0x00

Table 3. EZH peripheral registers*...continued*

Name	Access	Offset	Description	Reset Value
EZHB_ARM2EZH	R/W	0x40	• B31:2 - General-purpose bits. • B1:0 - Control the EZH interrupt trigger mechanism. When [1:0] == 0b10, a write operation to the EZH2ARM register triggers an interrupt to the Arm processor.	0x00
EZHB_EZH2ARM	R/W	0x44	B31:0 - General-purpose bits. Writing data to the EZHB_EZH2ARM register can trigger an interrupt to the Arm processor when EZHB_ARM2EZH[1:0] == 0b10.	0x00
EZHB_PENDTRAP	R/W	0x48	B31:24 - Reserved B23:16 - Pending trap enable B15:8 - Pending trap status polarity B7:0 - Pending trap request and status ***Note:*** *Arm can have a 'shortcut' to control bit-slices result using pending trap registers. By default, the pending trap feature is disabled.* **Example**: Setting EZHB_PENDTRAP = (1 << 16) | (1 << 0); enables the pending trap for bit-slice 0 and sets its result to 1.	0x00

3.2 EZH internal registers

All EZH registers are 32-bit as shown in Table 4.

Table 4. EZH internal registers

Register	R/W	Purpose
R0 – R7	RW	General-purpose registers
GPO	RW	EZH GPIO output register EZH GPIO output register bits [31:0] control the output states of EZH GPIO pins 31-0. If no PIN is configured as EZH GPIO, the register can be used as general-purpose registers. Write 0 – Clear output bit Write 1 – Set output bit
GPD	RW	EZH GPIO direction register EZH GPIO direction register bits [31:0] select the pin direction of EZH GPIO 31-0. If no PIN is configured as EZH GPIO, the register can be used as general-purpose registers. 1 – Output 0 – Input
GPI	RO	EZH GPIO input register Each bit [31:0] of the EZH GPIO input register reflects the state of EZH GPIO pins 31-0. 0 – Pin is low 1 – Pin is high
CFS	RW	Bit-slice source configuration register Bits 31-8 are used for bit slice source configuration. Each bit slice is configured using 3 bits. Bit slice 0 occupies the low 3 bits [10:8], bit slice 1 occupies bits [13:11], bit slice 2 occupies bits [16:14], and so on. Also, the lower 8 bits [7-0] of the register provide the current input status of the bit slices.
CFM	RW	Bit slice event configuration register

Table 4. EZH internal registers*...continued*

Register	R/W	Purpose
		Bits 31-8 are used for bit slice event configuration. Each bit slice is configured using 3 bits. Bit slice 0 occupies the low 3 bits [10:8], bit slice 1 occupies the bits [13:11], bit slice 2 occupies the bits [16:14], and so on. Each 3-bit field controls the event detection behavior of the corresponding bit slice. The possible event configurations per slice are: • 0 – Always 1 • 1 – Sticky rising edge • 2 – Sticky falling edge • 3 – Sticky any edge • 4 – High level • 5 – Low level • 6 – Always 0 • 7 – Any edge Also, the lower 8 bits [7-0] of the register are used for bit slice output result routing, where B0 corresponds to bit slice 0, B1 corresponds to bit slice 1, and so on. • 1 – Bit-slice output result is routed to a logical combiner. • 0 – Bit-slice result is routed to the next bit slice.
SP	RW	EZH stack pointer register EZH stack pointer register must be initialized, even if the EZH stack is not actively used.
PC	RW	EZH program counter register EZH program counter register is always positioned two instructions ahead of the instruction that is being loaded for execution.
RA	RW	EZH return address register

Note: *If some registers such as CFM, GPD, GPI are not used, they can be used as general-purpose registers.*

4 Application guide

This section introduces the development methods and application cases of SmartDMA or EZH.

4.1 EZH release

EZH mainly provides a C header file in which the machine code of EZH appears as inline assembly code through embedded C macro definition. After the user program initializes the EZH peripheral registers, EZH assembly instructions can be written in a routine to implement the function. For better execution efficiency of EZH, specify this routine to run in RAM.

4.1.1 IDE support

EZH code runs on all popular tool chains—MDK-Keil, MCUXpresso, and IAR supported with code array style. Arm compiler V6 and newer versions support the latest EZH instructions.

4.1.2 Project scatter file

The project scatter file can be modified to relocate EZH code to RAM. During compilation, the scatter file is set up to compile the code and store it in FLASH. When `__main` runs, the scatter loader generated by Arm-clang copies every content marked for RAM to its final destination.

4.1.3 EZH assembly programming

EZH instruction code can be embedded in the Arm compiler through macros called EZH pseudo instructions, which make it easy to write EZH code in the C language. The same EZH code can also be compiled into a library.

4.1.4 EZH controller APIs

EZH controller APIs as summarized in Table 5, provide function-level access to the EZH controller.

Table 5. EZH controller API function reference

Function	Description	Parameter	Return value
void EZH_Init(void)	Initialize the EZH processor. Enable the EZH clock by setting bit 4 of EZHB_CTRL to 1 for AHB synchronization.	None	None
void EZH_Boot(void *p Program, void *pPara, uint32_ t mask)	Start EZH for execution. pProgram: Pointer to EZH program start address (Must be 4-byte aligned) pPara: Pointer to EZH program parameter (Must be 4-byte aligned) uint32_t mask: Controls the behavior of EZH-Arm interaction 0 – Default EZHB_ARM2EZH_REGINT_ MASK: Writing to EZH2ARM register can trigger an EZH INT to Arm EZHB_ARM2EZH_EVENTOUT_ MASK: Writing to the EZH2ARM register can trigger an EZH event out	pProgram, pPara, and uint32_t mask	None
void EZH_Reset (void)	Stop EZH execution.	None	None
void EZH_DeInit (void)	Disable EZH and deactive EZH clock.	None	None
void EZH_SetExternal Flag(uint8_t flag)	Set or clear the external flag in the EZH Ctrl register. flag: External flag to be set or cleared.	flag	None
void EZH_SetEmergency Mode(void *ezh_emergency_ routine, uint8_t enable)	When enabled, the EZH program jumps to the emergency routine if an emergency is triggered. ezh_emergency_routine: EZH emergency routine vector to be set. enable: Enables EZH emergency mode.	ezh_emergency_routine, enable	None
void EZH_TriggerEmergency Mode (void)	Trigger an emergency event.	None	None
void EZH_SetBreakpoint Address (void *break_point_ addr, void *break_point_vect)	Set a breakpoint and enable a user debug routine. break_point_addr: 32-bit address to be swapped to EZHB_ BREAK_VECT location.	break_point_addr, break_point_ vect	None

Table 5. EZH controller API function reference*...continued*

Function	Description	Parameter	Return value
	break_point_vect: User debug routine vector.		

User can refer to the code examples for a better understanding of API usage.

4.1.5 APIs example

These examples show how to initialize, configure, and run the EZH controller with basic API usage (use LPC55xx code as examples).

```
void EZH_Init(void)
{
    CLOCK_EnableClock(kCLOCK_EzhArchB0);
    LPC_EZH_ARCH_B0->EZHB_CTRL = (0xC0DE0000 | (1<<EZHB_ENABLE_GPISYNCH));
}
```

```
void EZH_Boot(void * pProgram, void *pPara, uint32_t mask)
{
  LPC_EZH_ARCH_B0->EZHB_ARM2EZH = ((uint32_t)pPara | mask);
    LPC_EZH_ARCH_B0->EZHB_BOOT = (uint32_t) pProgram;
  SYSCON->AHBMATPRIO = PRI_EZHD(1) | PRI_EZHI(1);   //M6 - Inst, M7 - Data
LPC_EZH_ARCH_B0->EZHB_CTRL = 0xC0DE0011 | (0<<EZHB_MASK_RESP) |
(0<<EZHB_ENABLE_AHBBUF);
};
```

```
void EZH_Deinit(void)
{
    LPC_EZH_ARCH_B0->EZHB_CTRL = 0xC0DE0000;
    CLOCK_DisableClock(kCLOCK_EzhArchB0);
}
```

```
void EZH_Reset(void)
{
    RESET_PeripheralReset(kEZH_ARCH_B0_RSTn);
    LPC_EZH_ARCH_B0->EZHB_CTRL = (0xC0DE0000 | (1<<EZHB_ENABLE_GPISYNCH));
}
```

```
void EZH_SetExternalFlag(uint8_t flag)
{
    volatile uint32_t ezh_ctrl = (LPC_EZH_ARCH_B0->EZHB_CTRL & 0x0000FFFF);
    if (flag == 0) {
        ezh_ctrl &= ~(1 << EZHB_EXTERNAL_FLAG);
    } else {
        ezh_ctrl |= (1 << EZHB_EXTERNAL_FLAG);
    }
    LPC_EZH_ARCH_B0->EZHB_CTRL = (0xC0DE0000 | ezh_ctrl);
}
```

```
void EZH_SetEmergencyMode(void *ezh_emergency_routine, uint8_t enable)
{
    if (enable != 0) {
        LPC_EZH_ARCH_B0->EZHB_EMER_VECT = (uint32_t)ezh_emergency_routine;
        LPC_EZH_ARCH_B0->EZHB_EMER_SEL &= ~(1 << 8);
```

```
    } else {
        LPC_EZH_ARCH_B0->EZHB_EMER_VECT = (uint32_t)ezh_emergency_routine;
        LPC_EZH_ARCH_B0->EZHB_EMER_SEL |= (1 << 8);
    }
}
```

```
void EZH_TriggerEmergencyMode(void)
{
    LPC_EZH_ARCH_B0->EZHB_EMER_SEL |= (1 << 9);
}
```

```
void EZH_SetBreakpointAddress(void *break_point_addr, void *break_point_vect)
{
    LPC_EZH_ARCH_B0->EZHB_BREAK_ADDR = (uint32_t) break_point_addr;
    LPC_EZH_ARCH_B0->EZHB_BREAK_VECT = (uint32_t) break_point_vect;
}
```

How to initialize and start the EZH for code execution:

```
 EZH_Init(); //enable EZH clock and enable GPI AHB Synchronize
NVIC_EnableIRQ(IOH_IRQn); //enable interrupt of EZH
EZH_Pin_Init();//set the pin function as EZH function
 EZH_Boot(EZH_Code, &para, 0);//boot code from EZH_Code, para is parameter for
 EZH
```

The EZH then begins executing the code.

4.1.6 EZH clock and reset

The system control module handles both clock and reset control for EZH. The EZH clock must be enabled before configuring an EZH controller and executing an EZH program.

4.1.7 EZH PINMUX

EZH can access and manipulate up to 32 I/O pins when those pins are set to the 'EZH Function' in the PINMUX register. In the user manual spreadsheet, 'EZH Function' corresponds to:

- FUNC 4 on LPC5410x, LPC5411x, and LPC51U68
- FUNC 15 on LPC55(S)3x, LPC55(S)6x, and IMXRT500
- FUNC 7 on MCXNx4x

4.1.8 EZH Trigger

EZH has 8 bit-slices; each bit-slice can be connected to one external trigger. The INPUTMUX can be used to configure the input to each EZH bit-slice. As shown in Figure 6, platform-dependent triggers can be mux to EZH bit-slice input. Details about the EZH trigger mux register and the supported triggers can be found in the INPUTMUX register description.

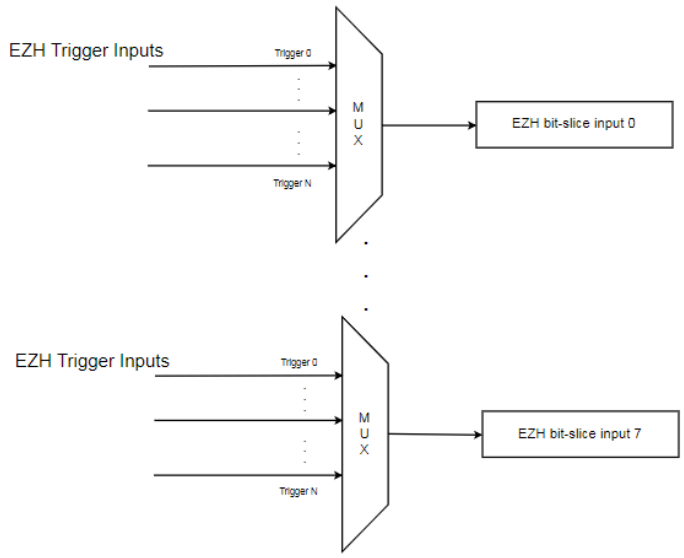

Figure 6. EZH trigger mux

4.1.9 EZH Interrupt

Like any other interrupt service routine(ISR), the Arm interrupt for EZH can be enabled through the Arm Nested Vectored Interrupt controller (Arm NVIC). The ISR function must be added to the ISR table in the startup file. The EZH IRQ number can be found in the NVIC chapter of the user manual.

4.2 Functions

This section introduces the basic functions of EZH, such as bus access, Arithmetic logic unit, Barrel shifter, Flags, and Algorithmic or branch control, among others.

4.2.1 Dual bus

EZH, like other CPUs, uses a pipeline to **fetch**, **decode**, and **execute** instructions. To optimize instructions and data fetching, EZH uses a dual interface to AHB:

- **Instruction bus (Inst-AHB)**: It is exclusively used to fetch opcodes.
- **Data bus (Data-AHB)**: It is used for fetching any other type of data access such as literals (RO data or memory accesses) anywhere across the chip.

Unlike an Arm core, EZH fetches data literals over the Data-AHB rather than Inst-AHB. This choice brings three advantages:

- **Simplification of the program counter mechanism:** The instruction bus is consistently two instructions ahead of the decode or execution stage, streamlining pipeline operation.
- **Deterministic correlation between IFETCH and DECODE:** The predictable correlation between IFETCH and DECODE stages enables the system to avoid unnecessary pipeline flush on a branch and precisely track the number of extra opcodes fetched.
- **Efficient scheduling of data accesses:** By routing all data reads through the DFETCH unit, the system enables efficient scheduling of reads and writes. This ensures that data, offset, and address fetches are managed without stalling the EZH pipeline.

4.2.2 Arithmetic Logic Unit(ALU)

Arithmetic logic unit (ALU) is used for performing both arithmetic and logical operations.

- Arithmetic operations: It supports maximum signed 32-bit arithmetic operations, including addition and subtraction.
- Logic operations: It supports logic operations, such as AND, OR, and XOR, including a combination of AND followed by OR, with most operations completed within a single cycle.

The EZH architecture can perform a combination of operations - FEND/FBIT + LSR + ALU + S - in a single cycle. This allows either Preshift + ALU or Postshift execution without extra delay.

Note:

- *If the destination is the program counter (PC), no flag can be set during an ALU.*
- *If a set flag ALU opcode is used with the PC as the destination, it acts as a request for a scheduled branch to the new value of the PC.*

Refer to the following examples:

```
E_ORS(R2,R1,R3)          //  OR R1 and R3 and store in R2, set flags based on the
 result
E_COND_ORS(PO,R2,R1,R3)      //  If Positive flag is set, OR R1 and R3 and store
 result in R2, then set flags
E_COND_ORS(PO,PC,R1,R3)      // If Positive flag is set, OR R1 and R3, then
 schedule branch to the result
E_COND_OR(PO,PC,R1,R3)     // If Positive flag is set, OR R1 and R3, then branch
 to the result
```

4.2.3 Barrel shifter

The EZH barrel shifter supports shifting and rotating operations with an immediate value ranging from 0 to 31.

In cases where the shift amount operand comes from a register, the EZH supports the shift and rotate value determined by the bottom byte of a register. Depending on the shift amount, the expected outcomes for the result and the carry flag are summarized in Table 6.

Table 6. Barrel shifter operation

Operation	How many bits to shift	Result	Carry bit
LSL	32	0	bit0
LSR	32	0	0
ASR	32	sign bit	sign bit
ROR	32	unchanged	bit 31
LSL	33 or more	0	0
LSR	33 or more	0	0
ASR	33 or more	sign bit	sign bit
ROR	33 or more	rotate % 32	modulus 32-bit

4.2.4 Flip or invert

The flip or invert feature provides bits order flip and bytes order flip capabilities before an ALU operation. This allows input data to be optionally preprocessed for the algorithm without any concern of bytes order. The expected results for flip and invert operations based on different input data are summarized in Table 7.

Table 7. Flip and invert operation

Input data	FEND (bytes order flip)	FBIT (Flip bitwise)	Invert
0x01234567	0x67452301	0xE6A2C480	0xFEDCBA98
0x89ABCDEF	0xEFCDAB89	0xF7B3D591	0x76543210

The flip and invert can be performed together with immediate load operation. Since `E_LOAD_IMM` only supports loading an 11-bit signed immediate, a 32-bit immediate load must be implemented using either load-then-shift operations, or load-then-invert operations (`E_LOAD_SIMM`, `E_LOAD_IMMN`, `E_LOAD`, and `SIMMN`) plus logical OR operation.

For example, the following sequence loads the 32-bit value 0xFAC68800 into the CFS register.

```
E_LOAD_SIMM(R0, 0xFA, 24)  // R0 = 0xFA000000
E_LOAD_SIMM(R1, 0xC6, 16)  // R1 = 0x00C60000
E_OR(R0, R0, R1)          // R0 = 0xFAC60000
E_LOAD_SIMM(R1, 0x88, 8)  // R1 = 0x00008800
E_OR(CFS, R0, R1)       // CFS = 0xFAC68800
```

4.2.5 Flags

MOV, LOAD, and ALU instructions, a bit-slice event, or an external flag match can signal the EZH flag, which drives conditional execution.

The EZH flags can be described in 3 categories:

- ALU flags
- Algorithmic or branch control flags
- Boolean or external detection flags

4.2.5.1 ALU flags

ALU flags are settable using any opcode which involves an ALU operation, or a MOV operation, or a LOAD immediate value. The various ALU flags, along with their Arm equivalents and purposes, are summarized in Table 8.

Table 8. ALU flags and their purposes

Flag	Arm equivalent	Purpose
EU	<None>	Execute Unconditionally
ZE	EQ	Zero
PO	PL/GE	Positive
NE	MI	Negative
AZ	GT	Above zero
ZB	LE	Zero or below
CA	CS	Carry set
NC	CC	Carry not set
CZ	<None>	Carry set and zero

The opcodes that are not capable of setting flags include any data bus read or write operations like:

- `E_PER_WRITE`

- E_PER_READ

Also, the following opcodes do not set flags:

- E_NOP
- E_INT_TRIGGER
- E_GOTO
- E_GOSUB
- E_GPO_BYTE_MODIFY
- E_TIGHT_LOOP
- E_HOLD
- E_VECTORED_HOLD
- E_HEART_RYTHM
- E_SYNCH_ALL_TO_BEAT
- E_WAIT_FOR_BEAT

Example for ALU flags:

```
start
E_ADDS(R1, R2, R3)      // Add R2 to R3, store result in R1 and set flags
E_COND_LDR(ZE, R1, R2) // If result above was Zero, read from address pointed to
 by R2 into R1
E_COND_LDR(NZ, R1, R3) // If result was non Zero, read from address
pointed to by R3 into R1
E_COND_GOTO(AZ, start) // If result was above zero, go back to start
```

Note:

- *If the destination is set to the PC, no flags can be set on an ALU transaction.*
- *When a set-flag ALU opcode is used with the destination as the PC, it requests a scheduled branch to the new value of the PC.*

4.2.5.2 Algorithmic or branch control

Algorithmic control and branch control use the same flag opcodes; SPO/UNS(0xa) and SNE/NZS(0x0b).

4.2.5.2.1 Algorithmic control

The two algorithmic flags (SPO and SNE) are secondary versions of PO and NE. However, unlike PO and NE, which get set anytime a FLAG-SET instruction is executed, the SPO and SNE flags can only be set under specific conditions. They are updated only if the opcode attempting to set them is executed with a conditional check on SPO or SNE. For example, `E_COND_LSL_ORS (SPO, R0, R0, R1, 1)`

The two algorithmic flags have the following meaning:

- **SPO – Shift-only-when-Positive**: Perform the extra operation when the top bit is **1**.
- **SNE – Shift-only-when-Negative**: Perform the extra operation when the top bit is **0**.

The following example demonstrates the behavior between SPO and PO flags:

```
E_LOAD_IMM(R1, 1) // R1 = 1
E_LOAD_IMMS(R0, 1)                    // R0 = 1 (PO = 1, SPO = 0)
E_COND_LSL_ORS(SPO, R0, R0, R1, 1)   // SPO == 0 -> Shift then OR
(PO = 1, SPO = 1)
E_PUSH(R0)                            // R0 = 3
E_LOAD_IMMS(R0, 1)                    // R0 = 1 (PO = 1, SPO unchanged)
E_COND_LSL_ORS(SPO, R0, R0, R1, 31) // SPO == 1 -> Shift only
```

```
E_PUSH(R0)                          // R0 = 0x80000000 (PO = 0, SPO = 0)
E_LOAD_IMM(R0, 1)                   // R0 = 1
E_COND_LSL_ORS(PO, R0, R0, R1, 1)   // PO == 0 -> Not executed
E_PUSH(R0)                          // R0 = 1
E_COND_LSL_ORS(SPO, R0, R0, R1, 1)  // SPO == 0 -> Shift then OR
E_PUSH(R0)                          // R0 =
```

The following are the twofold purposes of the algorithmic flags:

- To allow the user to implement a shift-based algorithm (for example, binary multiplication), where a shift always occurs, but an ALU operation is conditionally executed based on the value of the most significant bit (MSB). These flags enable the user to ensure that the shift always takes place, while the other operation can be canceled when not required.
- To allow the user to perform other flag-based operation and later resume the algorithmic shift using the previously saved SPO/SNE flags—as long as the intermediate operations did not affect the algorithmic flags.

Consider the following example of a 16-bit x 16-bit multiplication. The highlighted line presents the core of the multiplication algorithm. In the example below, the core of the algorithm relies on the ability to conditionally perform an add operation when the MSB is high.

```
E_LDR(R0, PC, 7)   // R0 load from DCD after result
E_LDR(R1, PC, 7)   // R1 load from DCD
E_LOAD_IMM(R3, 15)  // Repeat 15 times after initial iteration
E_ADD_IMM(R4, PC, 2*4) // Define end of iteration loop
E_TIGHT_LOOP(R4, R3) // Enable loop
E_COND_LSLS(SPO, R0, R0 ,16)  // Shift Multiplicand Left by 16
start_of_loop
E_COND_LSL_ADDS (SPO, R0, R1, R0,1)// Multiply - This will execute 16
                                   // times, shifting or shifting and
                                   // adding depending on Multiplier
end_ofloop
E_STR(PC, R0, 0)     // Write result back on top of first operand
E_SUB_IMM(PC, PC, 8)  // while 1
result
DCD 0x0000abcd   // Multiplicand
DCD 0x0000ef01   // Multiplier
```

4.2.5.2.2 Branch control

Typically, a branch flushes the instructions in the pipeline and immediately updates the program counter (PC). However, with a scheduled branch, the two instructions delay the branch, allowing execution to continue without flushing the pipeline.

The flag NZS (schedule branch when the NZ flag is set) and UNS (unconditionally schedule) are used for the scheduled branch. These flags can only be used for qualifying an `E_COND_GOTO` opcode.

In the example below, are R0, R1, R4 pushed to the stack:

```
E_COND_GOTO(UNS, ne2) // Schedule branch unconditionally
E_PUSH(R0)    // Executed - Pipeline is not flushed
E_PUSH(R1)    // Executed - Pipeline is not flushed
E_PUSH(R2)
ne2
E_GOTO(ne3)    // Branch immediately
E_PUSH(R3)    // Not executed
ne3
E_PUSH(R4)
```

4.2.5.2.3 Boolean and external detection flags

EZH allows the user to be able to monitor the state of the external condition input. This provides a way to allow the user to extend the usage of EZH by using a flag that is fed from outside the EZH. The EZH has no architectural support for directly setting or clearing this flag. However, the user can observe the external flag status in any opcode that supports conditional execution. The flags used for this purpose are:

- EX - External flag is set
- NEX - External flag is not set

Both EZH and Arm can write the EZH control register (EZHB_CTRL) bit 1 to change the EZH state.

Second, the EZH architecture also allows the user to monitor the value of the Boolean monitoring logic combiner. The available flags are:

- BS - Boolean-expression set
- NBS - Not Boolean-expression set

The benefit of this feature is that it allows the user to perform Boolean monitoring on-the-fly.

There is an important aspect to consider is that if CFM and CFS registers (Boolean detector configuration registers) are being loaded from a literal fetch through the Data bus, the configuration update cannot be immediate. There is no protection mechanism in place to prevent EZH from evaluating opcodes that are conditional on BS or NBS before the configuration of the Boolean detector is fully updated.

This behavior contrasts with `E_VECTORED_HOLD` and `E_HOLD`, both of which stall any detection until any outstanding reads into CFM or CFS are complete and data arrives back in the EZH.

These instructions operate by artificially holding the BS flag low as long as updates to CFM or CFS are still pending through the data bus.

Finally, if an `E_HOLD` or `E_VECTORED_HOLD` is executed based on BS, there is no guarantee that the BS flag matches the actual evaluation of the `E_HOLD` or `E_VECTORED_HOLD` itself. This match is only guaranteed if the inputs to the Boolean detector don’t change.

For example, `E_COND_VECTORED_HOLD(BS,PC)`

If the BS flag is high, in the example above, the instruction executes a vectored hold. At first glance, this appears to allow the code to jump to an active vector only if a known vector is already active. However, the risk lies in timing: if the inputs to the Boolean detector change between the time the instruction is decoded and the time it executes (typically at least one cycle later), the code can jump to a vector that is no longer valid. This can cause the code to stall, waiting for a vector that matched at decode time but has since disappeared.

4.2.6 GPIO

The EZH supports a 32-bit direct access to bidirectional GPIO pins, which can be programmed directly through the EZH instructions. This allows faster access as compared to using them through the system AHB bus.

GPIO is controlled through GPO and GPD. The GPI register is a read-only register.

In addition to access these registers through any EZH opcodes, there is also a dedicated opcode called `E_MODIFY_GPO_BYTE`. This opcode enables simultaneous bit clearing, bit setting, and bit toggling of a mask on 8 bits of GPO in a single cycle.

4.2.7 Heartbeat timer

EZH features a 16-bit down counter known as the heartbeat timer, designed to provide an efficient mechanism for scheduling tasks. It is an automatically reloadable down counter which 'beats' each time it reaches 0. The reloadable value is set using the `E_HEART_RYTHM` opcode. Any use of this opcode changes the reloadable value and updates the maximum value of the counter and starts counting down from it on the following cycle.

By default, the heartbeat count is set to 0, resulting in a 'beat' on every cycle.

Note: *When the heartbeat is set to 0, the counter is clock-gated to conserve power.*

The heartbeat timer can be used in the following two ways:

- **Beat-based halting with `E_WAIT_FOR_BEAT`:** Using the `E_WAIT_FOR_BEAT` instruction, the user halts execution until the next beat. This is useful for aligning task execution with regular intervals. For example, in a serial sampling, it helps to ensure that sampling occurs at deterministic timing.
- **Synchronized execution mode with `E_SYNCH_ALL_TO_BEAT`:** Alternatively, using the `E_SYNCH_ALL_TO_BEAT` instruction switches the EZH in an execution mode, where all instructions are executed only on heartbeat ticks. This provides multiple benefits:
  - **Power efficiency:** As the EZH operates at a slower rate placed by the heartbeat rhythm count, it can save power when full-speed execution is not required. For example, if the EZH can complete tasks faster than needed, controlling execution speed using the heartbeat can help reduce energy consumption.
  - **Deterministic 'snap-to-grid execution:** In this mode, the EZH stalls between beats. During this time, it fetches and decodes the next instruction and performs any outstanding memory or data transactions. This enables highly deterministic execution, even if the EZH is running from shared memory with another AHB controller.

4.2.8 Boolean detection (bit-slice)

The EZH includes eight Boolean detection engines, known as bit-slice (BS) Figure 7 shows the bit-slice architecture.

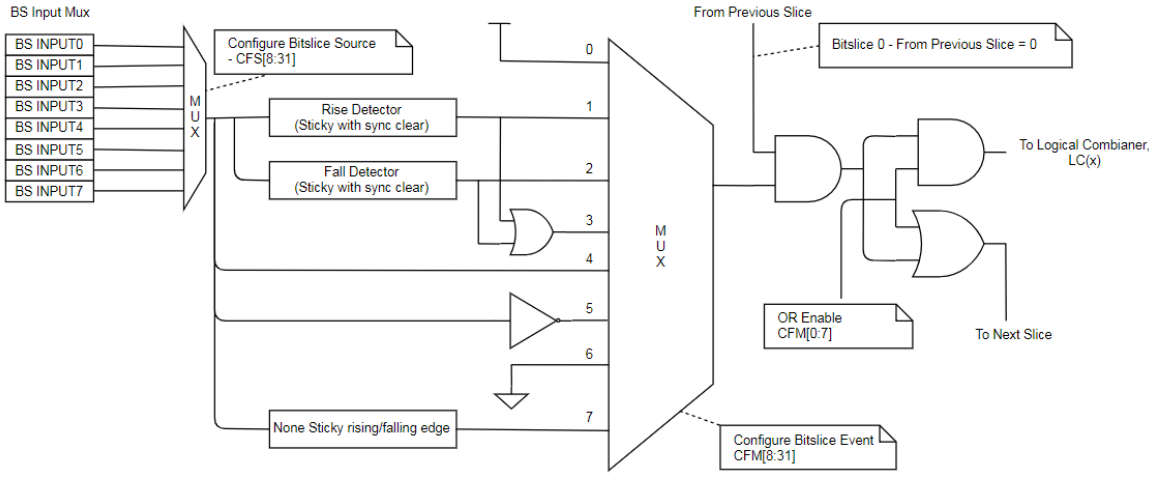

Figure 7. Bit-slice architecture

Before using a bit-slice, `BS_INPUTX` must be bound to an EZH trigger using the INPUTMUX of the system. Similar to DMA triggers, the EZH trigger source can be selected from 'GPIO', 'Interrupts', or 'timer match event', depending on the device.

Each bit-slice takes one input signal from the eight available bit-slice inputs (`BS_INPUT0` to `BS_INPUT7`). The configuration is set through the CFS register.

- Bits [8:31] of CFS register select the source input for each bit-slice (BS0 to BS7).
- Each bit-slice uses 3 bits for source selection.
- For example: CFS = 0x12345600 defines the mapping of input signals to the bit-slices.

Table 9. CFS configuration for bit-slice input selection

Bit fields	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
CFS	000	100	100	011	010	001	010	110
INPUTX	Input 0	Input 4	Input 4	Input 3	Input 2	Input 1	Input 2	Input 6

Each bit slice can detect the following input events: 'Rising edge', 'falling edge', 'edge', 'high level', or 'low level'. There is also 'sticky' or 'nonsticky' edge detection. Sticky detection modes require the user to clear the sticky flag manually by writing to the CFM register before a new event can be detected. Table 10 defines the meaning of the event mux bits:

Table 10. Example mux bits behavior (from CFM):

EZH CFM event mux bits	Mux output
000b	Mux output is always 1, regardless of the input signal.
001b	Mux outputs 1 when the input signal detects a rising edge. The user must clear the sticky bit manually (any write to the CFM register) for next event detection.
010b	Mux outputs 1 when the input signal detects a falling edge. The user must clear the sticky bit manually (any write to the CFM register) for next event detection.
011b	Mux outputs 1 when the input signal detects either a rising or falling edge. The user must clear the sticky bit manually (any write to the CFM register) for next event detection.
100b	Mux outputs 1 when the input signal is high.
101b	Mux outputs 1 when the input signal is low.
110b	Mux output is always 0, regardless of the input signal.
111b	Mux outputs 1 when the input signal detects either a rising or falling edge.

Each bit-slice has two outputs:

- One routes to the Logical Combiner (LC) block.
- The other connects to the next bit-slice in the chain.

The `OR_Enable` flag controls the output behavior, which corresponds to bits [0:7] of the CFM register, one bit per bit-slice.

Note: *If `OR_Enable` is set for bit-slice 7, the output is routed to the Logical Combiner regardless of its configuration.*

All bit-slice outputs are routed to LC, which EZH uses to determine whether the specified condition is met. By implementation, EZH continues to fetch and execute instructions until it encounters a `HOLD`, `VECTOR_HOLD`, or `ACC_VECTORED_HOLD` instruction. Execution halts when the LC output evaluates to 1, according to the logic:

`LC0|LC1|LC2|LC3|LC4|LC5|LC6|LC7`

Figure 8 illustrates how each bit-slice contributes to the LC. Each bit-slice detects a specific event (for example, BS0 detects event A, BS1 detects event B). LC0, LC1,…...,LC7 represent the outputs from bit-slices to the LC, each controlled by the corresponding `OR_Enable` flag.

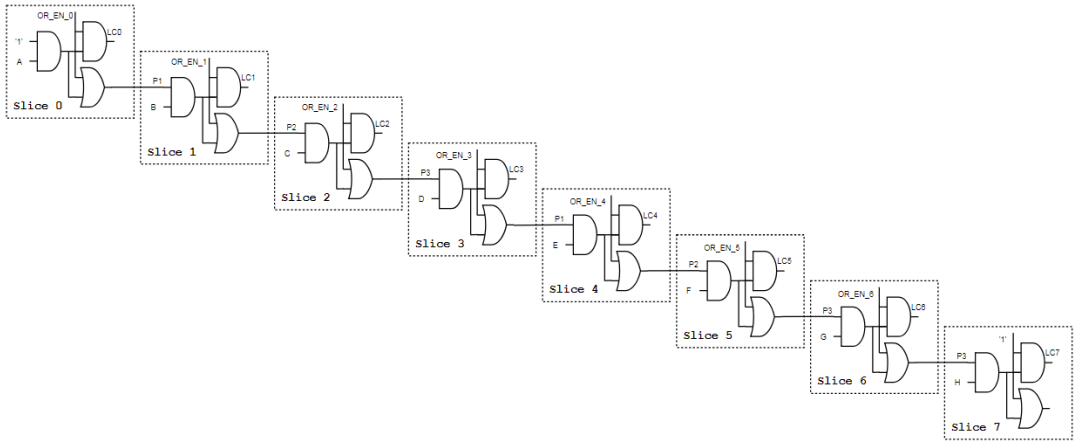

Figure 8. Cascade bit-slices for logical combiner

As shown below, the CFS and CFM registers are configured to detect a rising edge event on bit-slice 0. The EZH halts execution at `E_HOLD`, waiting for the rising edge to occur on bit-slice 0. Once the event is detected, the EZH resumes execution and proceeds with `E_PUSH` instruction.

```
E_LOAD_IMM(CFS, 0)
E_LOAD_IMM(CFM, 0x101)
…
E_HOLD
E_PUSH(R0)
```

An alternative Boolean detection feature in EZH is the `VECTORED_HOLD` instruction. It allows simultaneous monitoring of multiple conditions without executing any instruction until a condition is met. It operates similarly to the `HOLD` instruction, with the key difference being that `VECTORED_HOLD` halts execution until one of the logic OR terms are encountered.

Once an event is detected, EZH jumps to the corresponding vector table entry and begins execution from that address. The vector table contains eight enteries by default, and a pointer to it is passed through a register by the `VECTORED_HOLD` opcode. If its output is '1', each bit-slice can call a vector. In cases where multiple bit-slices outputs '1', and their outputs are ANDed. The vector to be executed corresponds to the last bit-slice in the ANDed expression.

For example, If BS0 detects event A and BS1 detects event B, and the logical combiner is monitoring the expression ABC+D+EFGH, the lowest 8 bits in the CFM register (that is, the `OR_Enable` field) must be set accordingly. Take for instance, `OR_Enable` = 10001100b enables only the relevant bit-slice outputs for this condition.

The following code demonstrates how `VECTORED_HOLD` can be used to handle an event triggered by the logical combination ABC+D+EFGH. The `VECTORED_HOLD` opcode can be interpreted as a combination of 'HOLD + GOTO/GOSUB to vector', similar to how WFI works in Arm.This means that when a vector fires, the EZH can back up the location into its RA register. After the vector handler completes, the EZH code resumes from where it left off.

```
E_VECTORED_HOLD(PC)
E_NOP
E_NOP
Vector_Table:
E_NOP       // BS0_vector
E_NOP      // BS1_vector
E_GOTO(BS2_ISR)    // BS2_vector
E_GOTO(BS3_ISR)    // BS3_vector
```

```
E_NOP       // BS4_vector
E_NOP       // BS5_vector
E_NOP       // BS6_vector
E_GOTO(BS7_ISR)    // BS7_vector
BS2_ISR
E_BSET_IMM(CFM, CFM, 2)
E_LOAD_IMM(R2, 2)
E_GOTO(bf)
BS3_ISR
E_BSET_IMM(CFM, CFM, 3)
E_LOAD_IMM(R2, 3)
E_GOTO(bf)
BS7_ISR
E_BSET_IMM(CFM, CFM, 7)
E_LOAD_IMM(R2, 7)
bf
…
```

The two instructions that follow `E_VECTORED_HOLD` flush from the pipeline and do not execute, which adds latency to the event handler. `E_ACC_VECTORED_HOLD` provides an opcode that keeps the instructions after the `HOLD` operation in the pipeline, so they execute quickly when an event happens.

`E_ACC_VECTORED_HOLD` takes an event handler vector and a mask that shows which event to accelerate. In the code below, the mask shows that bit-slice (LC2 = 1) is the event to accelerate. When this event occurs, the instructions `E_GOTO(UNS,label`, and `E_BTOG_IMM` (GPO, GPO, 0) execute. If a different bit-slice event occurs, those two instructions flush from the pipeline and do not execute.

```
label
E_ACC_VECTORED_HOLD(PC,0x04)
E_GOTO(UNS,label)
E_BTOG_IMM(GPO,GPO,0)
…
Vect2
E_BTOG_IMM(GPO,GPO,0)
```

EZH also supports large vector tables. Each entry in these tables includes four instructions instead of just one. The reason for using four instructions is that a scheduled branch followed by two 'unflushed' opcodes usually uses only three instructions. This means that the large vector table entry has enough space for one more instruction, allowing a single cycle branch to the rest of the handler if needed.

See the following samples:

```
Label0
E_ACC_VECTORED_HOLD_LV(PC,1<<1);
E_BTOG_IMM(GPO,GPO,18);          //non-flushed instruction 1
E_BTOG_IMM(GPO,GPO,18);          //non-flushed instruction 2
…
E_BTOG_IMM(GPO,GPO,18);          //LC1  1st opcode
E_BTOG_IMM (GPO,GPO,18);         //LC1  2nd opcode
E_BTOG_IMM(GPO,GPO,18);          //LC1  3rd opcode
E_GOTO(label1);              //LC1  4th opcode, go to label1
…
Label1
E_BTOG_IMM(GPO,GPO,18);          //1st opcode in longer vector
E_GOTO(Label0);           // 2nd opcode in longer vector , go to label0
```

Note:

- *E_ACC_VECTORED_HOLD_LV, E_ACC_VECTORED_HOLD, E_ VECTORED_HOLD_LV, and E_VECTORED_HOLD, each set of four instructions takes about six cycles from the moment the trigger occurs to when holding is released.*
- *Any write to the CFM and CFS clears the BS flag.*

4.2.9 Pending trap

EZH supports a pending trap function that can latch the edge of input IO, and does not change the status until the flag is cleared. This bit change acts as a trigger source for bit-slice operations.

4.2.9.1 Use case 1 (EZH pending trap status usage)

This use case helps to detect a logic level change on the pin mux input using the pending trap feature. Follow the steps to configure EZH and monitor the input transactions.

Steps:

1. Enable EZH.
2. Configure the pin mux to EZH.

Then:

Bits 7-0 in `EZHB_PENDTRAP` represent the logic level change of the pin mux input IO. The pending trap uses a latching logic '2' state. This means that bit 7 to 0 changes from '0' to '1' when the input IO logic changes from '0' to '1'. However, they do not change back to '0' until the bit is manually cleared, even if the input IO logic returns to '0'.

4.2.9.2 Use case 2 (EZH pending trap status polarity usage)

This use case demonstrates how to configure the pending trap to detect a falling edge (from high to low) on the pin mux input by setting the polarity bits.

Steps:

1. Enable EZH.
2. Configure the pin mux to EZH.
3. Set bits 15-8 (polarity) of the `EZHB_PENDTRAP` register to '0xFF'.

Then:

Bits 7-0 in `EZHB_PENDTRAP` represent the opposite logic level change of the pin mux input IO. The pending trap uses a latching logic 0 state. This means bit 7 to 0 changes from '0' to '1' when the input IO logic changes from '1' to '0'. However, they do not change back to '0' until the bit is manually cleared, even if the input IO logic returns to '0'.

4.2.9.3 Use case 3 (EZH slice vector usage)

This use case explains how to configure the pending trap to trigger a slice vector-using software.

Steps:

1. Enable EZH.
2. Configure the pin mux to EZH.
3. Enable the pending trap by setting bit 23-18 in `EZHB_PENDTRAP` to '0xFF'.
4. Enable the EZH interrupt.

Then:

Set the CFS and CFM registers for bit-slice functionality. When bit 7-0 in EZHB_PENDTRAP changes from '0' to '1', it causes the slice states to match and trigger the slice vector. Under this condition, the actual IO level or edge is no longer recognized by the bit-slice. Therefore, bit 7-0 can be modified manually to trigger slice actions through software.

Table 11. Bit-slice state mapping

Logic of bit 7-0 in EZHB_PENDTRAP	Present states to bit-slice
'0'	Low level
'1'	High level
From '0' to '1'	Rising edge
From '1' to '0'	Falling edge

4.2.10 Tight loop

The EZH offers an additional branching mechanism that introduces no overhead cycles during the branch itself. This is known as the tight loop mechanism. In a traditional CPU loop, execution typically follows a decrement-compare-branch approach. That is, the loop counter (iterator) is decremented until it reaches 0, and a comparison opcode checks whether the loop can continue or exit. Once the condition is met, the branch is taken. With a tight loop, however, the EZH hardware manages the loop control internally, reducing software overhead, and improving efficiency.

The examples below show two implementations for toggling PIO_0 sixteen times, using either the traditional decrement-compare-branch or tight loop.

Example 1: Toggle a PIO using a decrement-compare-branch mechanism.

```
E_BSET_IMM(GPO, GPO, 0) // Set PIO_0 value high
E_BSET_IMM(GPD, GPD, 0)    // Set PIO_0 direction output
E_LOAD_IMM(R0, 16)         // Set loop counter 16
start
E_BTOG_IMM(GPO, GPO, 0)    // Toggle PIO_0
E_SUB_IMMS(R0, R0, 1)      // R0 = R0 - 1
E_COND_GOTO(NZ, start)     // Jump to start if non-zero flag
```

Example 2: Toggle a PIO using a tight loop mechanism.

```
E_BSET_IMM(GPO, GPO, 0) // Set PIO_0 value high
E_LOAD_IMM(R0, 15)      // Set loop counter 16 initial loop + 15 loops
                          // following
E_ADD_IMM(R1, PC, 8)      // R1->the instruction after
// E_BTOG_IMM(GPO,GPO, 0)
E_TIGHT_LOOP(R1, R0)      // Setup tight loop
                          // R1 = end address, R0 = loop counter
E_BSET_IMM(GPD, GPD, 0)   // Set PIO_0 direction output
                          // only execute once at initial loop
E_BTOG_IMM(GPO, GPO, 0)   // Toggle PIO_0, execute 16 times
```

Notes on tight loop usage:

- **Reusable register:** The registers used for tight loop setup can be reused within the loop body. For example, in the case above, R0 and R1 are used to set up a tight loop using E_TIGHT_LOOP (R1, R0). R0 and R1 can be rewritten without affecting loop execution.
- **First instruction behavior:** The instruction immediately following E_TIGHT_LOOP executes only once during the initial loop entry.

- **Execution count:**`E_TIGHT_LOOP` executes for (1 + loop counter) times.

Performance note:

The code is designed to fetch data from SRAM (with R0 points to the memory address where data is stored). It can convert the endianness of the data, and write it to an APB peripheral register. The codeon the left uses DFETCH, executing four consecutive loads into R2, R3, R4, and R5. This approach allows EZH to operate without stalling. In contrast, the code on the right is less efficient. As R2, R3, R4, and R5 are used immediately for endianness conversion (FEND opcode) right after the load instruction, EZH must be stalled to avoid conflicts.

```
1  E_LDR_POST(R2, R0, 1)     1  E_LDR_POST(R2, R0, 1)
2  E_LDR_POST(R3, R0, 1)     2  E_FEND_LSR(R2, R2, 0)
3  E_LDR_POST(R4, R0, 1)     3  E_PER_WRITE(R2, 0x32200)
4  E_LDR_POST(R5, R0, 1)     4  E_LDR_POST(R3, R0, 1)
5  E_FEND_LSR(R2, R2, 0)     5  E_FEND_LSR(R3, R3, 0)
6  E_PER_WRITE(R2, 0x32200)  6  E_PER_WRITE(R3, 0x32204)
7  E_FEND_LSR(R3, R3, 0)     7  E_LDR_POST(R4, R0, 1)
8  E_PER_WRITE(R3, 0x32204)  8  E_FEND_LSR(R4, R4, 0)
9  E_FEND_LSR(R4, R4, 0)     9  E_PER_WRITE(R4, 0x32208)
10  E_PER_WRITE(R4, 0x32208)  10  E_LDR_POST(R5, R0, 1)
11  E_FEND_LSR(R5, R5, 0)    11  E_FEND_LSR(R5, R5, 0)
12  E_PER_WRITE(R5, 0x3220C)  12  E_PER_WRITE(R5, 0x3220C)
```

4.2.11 Stack support and return address

The EZH contains a stack pointer (SP) register, which the user manages manually. Inherently, the hardware itself does perform automatic stack management. The software must perform any operations such as pushing or popping. The `E_PUSH` and `E_POP` opcodes act as wrappers for `E_LDR` and `E_STR` instructions. These opcodes hardcode the pointer register as SP and currently enforce an upward growing stack. However, if required, the macro definition for `E_PUSH` and `E_POP` can be changed in the future to allow downward stack growth.

EZH ia able to automatically back up the return address (RA)—the location from which a branch was made. If the user wants to back this return address onto the stack, the user must use the `E_PUSH` and `E_POP` opcodes.

Furthermore, by default, only `E_VECTORED_HOLD` and `E_GOSUB` automatically back-up the return address into the RA register. `E_VECTORED_HOLD` can be configured to skip this behavior using NRA (no return address) option.

In the case of `E_GOSUB`, the RA register is always overwritten, so users must explicitly preserve it if needed by stacking. With `E_GOTO`, the 'L' option indicates that the return address must be saved into the RA register.

The following example demonstrates how the RA register behaves during an EZH function call using `E_GOSUB`:

```
E_LOAD_IMM(R0, 0xDD)  // R0 = 0xDD
E_GOSUB (func1)     // Call sub routine func1, return address in RA
endfc
E_GOSUB(endfc)     // End loop
func1
…
E_PUSH(RA)           // RA stack must be preserved by user
E_GOSUB(func2)     // Call sub routine func2, return address in RA
E_POP(RA)           // Recover RA
E_GOTO_REG(RA)     // Return to func1_caller
func2
…
E_GOTO_REG(RA)     // Return to func2_caller
```

4.2.12 Breakpoint support

The EZH supports a debug mechanism called ‘Breakpoint’ support. It works by providing the EZH with an address where a break must occur and the corresponding service code resides.

EZH replaces the instruction `EZHB_BREAK_ADDR` with a branch to the breakpoint vector address `EZHB_BREAK_VECT`. If the opcode is flushed, the breakpoint has no effect. It only takes effect when the opcode is about to be executed, as expected for a breakpoint.

The opcode, which is set to `EZHB_BREAK_ADDR`, cannot be executed because `EZHB_BREAK_VECT` is executed instead, unless `EZHB_BREAK_ADDR` is reset. Therefore, it is a good practice to set `EZHB_BREAK_ADDR` to an `E_NOP` opcode, so that the breakpoint does not impact the program logic.

The following examples demonstrate how to use the EZH breakpoint. `EZHB_BREAK_ADDR` is set to an `E_NOP` operation placed after `E_PUSH` (R0). `EZHB_BREAK_VECT` is set to the start of the `EZH_RegistersDump` routine, which serves as a debug handler. In the example, the registers, R4, R5, R6, and R7 are pushed onto the stack so their values can be viewed in the IDE. The debug routine then returns to the regular proh=gram by jumping to `EZH_BREAKADDR + 4`, the address of the operation just after `EZHB_BREAK_ADDR` is executed.

```
__asm void EZH_CODE EZH_SetRegisters(void)
{
   E_NOP
   E_PER_READ(R6, EZH_ARM2EZH)
   E_LSR(R6, R6, 2)
   E_LSL(R6, R6, 2)
   E_LDR(SP, R6, 0)
   E_LOAD_IMM(R0, 0xC0)
   E_LOAD_IMM(R1, 0xC1)
   E_LOAD_IMM(R2, 0xC2)
   E_LOAD_IMM(R3, 0xC3)
   E_LOAD_IMM(R4, 0xC4)
   E_LOAD_IMM(R5, 0xC5)
   E_LOAD_IMM(R6, 0xC6)
   E_LOAD_IMM(R7, 0xC7)
   E_PUSH(R0)
   E_NOP          // Breakpoint Address
   E_PUSH(R1)
   E_PUSH(R2)
   E_PUSH(R3)
   E_INT_TRIGGER(0)
end9
   E_GOTO(end9)
}
```

```
__asm void EZH_CODE EZH_RegistersDump(void)
{
   E_PUSH(R4)       // Breakpoint Vector Address
   E_PUSH(R5)
   E_PUSH(R6)
   E_PUSH(R7)
   E_PER_READ(R0, EZH_BREAKADDR)
   E_ADD_IMM(R0, R0, 4)
   E_GOTO_REG(R0)
}
```

Note: *Avoid placing breakpoints at locations where a scheduled branch is happening. While the breakpoint functions technically, it can make it difficult for the user to trace and reconstruct the execution of the program flow accurately.*

4.2.13 Error and emergency mode

EZH supports an emergency mode designed to allow controlled shutdown of the system in response to an external emergency event or an AHB error response.

The mechanism works by the system providing the EZH with an emergency vector location. When the emergency input of the EZH goes high or an error is detected on either of the AHB buses, the program branches at the next possible instruction cycle to the emergency vector location.

It is important to note that emergency mode does not set any restrictions in terms of what can or cannot be executed. However, after emergency mode is entered, it cannot be left without resetting the EZH—either pulling the ignition line low or resetting the block. This also means that subsequent assertions of emergency or AHB error have no effect after emergency mode is already entered.

4.2.14 Arm or EZH interrupt

EZH can raise interrupts to the Arm processor through three different mechanisms. The Arm core can then handle these interrupts within the EZH (SmartDMA) debug routine.

- **`E_INT_TRIGGER` instruction:** This instruction takes one parameter, which must be any positive number greater than 0. `E_INT_TRIGGER` is one of the only opcodes that cannot be used with a condition flag.
- **Handshake through `EZHB_EZH2ARM` and `EZHB_ARM2EZH` registers:** When the `EZHB_ARM2EZH` register has bit [1:0] = 0b10, any write to the `EZHB_EZH2ARM` register can trigger an interrupt to the Arm core.
- **Hijack function:** EZH provides an interrupt output bus of 32 bits(or fewer than 32 outputs on some platforms), which allow EZH to emulate different peripheral toward the Cortex-M CPU. The value passed into `E_INT_TRIGGER` is interpreted as an interrupt bus signal to the Arm. Each bitmaps to one of the interrupts to hijack. To hijack the third interrupt, the code can be written as: `E_INT_TRIGGER(1<<3)`.

Separately, EZH also has its own interrupt slot in the NVIC, which is an OR of all bits passed into `E_INT_TRIGGER`. To implement this interrupt, the hijack function must be enabled in the SYSCON registers as shown in Figure 9:

1.

Function

Bits directly control the SmartDMA hijacking the system interrupts.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R									INT23	INT22	INT21	INT20	INT19	INT18	INT17	INT16
W	Reserved															
Reset	u	u	u	u	u	u	u	u	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	INT15	INT14	INT13	INT12	INT11	INT10	INT9	INT8	INT7	INT6	INT5	INT4	INT3	INT2	INT1	INT0
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-24 —	Reserved Read value is undefined, only zero should be written.
23 INT23	SmartDMA hijack NVIC IRQ77 0b - Disable 1b - Enable

Figure 9. EZH interrupt

4.2.15 Scheduled branch

Scheduled branching enables controlled program flow changes using either direct PC based offset jumps or label-based jump tables with predefined offsets.

4.2.15.1 One method

An ALU operation where the destination register is PC can be used to implement a scheduled branch. This is possible only if the user sets the necessary flags:

For example:

```
E_ADD_IMMS(PC,PC,8) // perform a scheduled branch to a location 2 words ahead
```

Note: *As the PC is always two instructions ahead, this results in branching four instructions ahead of the current one.*

4.2.15.2 Another method

Use PC based branching to implement a jump-label function:

1. Define the label offset.

```
/* DON'T Touch: the jump-to-label offset in EZH code */
                                    #define offset_start  0   //label 0
 macro definition
```

```
                                        #define offset_VSYNC    1 //label 1
 macro definition
                                        #define offset_pixel  2   //label 2
 macro definition
                                        #define offset_end  3     //label 3
 macro definition
                                        #define offset_VSYNC_fall 4//label 4
 macro definition
```

2. Store the label addresses in the R7.

```
E_ADD_IMM(R7,PC,-4);  //load start into R7
E_DCD(start);         //label 0 pointer
E_DCD(VSYNC);         //label 1 pointer
E_DCD(pixel);         //label 2 pointer
E_DCD(end);           //label 3 pointer
E_DCD(VSYNC_F);       //label 4 pointer
```

3. Implement the label code.

```
E_LABEL("start");
…
E_LABEL("VSYNC");
…
E_LABEL("VSYNC_F");
…
E_LABEL("pixel");
…
E_LABEL("end");
…
```

4. Perform a jump-label.
   Conditional scheduled jump:

```
E_MOVS(R0,R1);
E_COND_LDR(AZ,PC,R7,offset_start); // if R0 above zero, jump to start
```

   Unconditional scheduled jump:

```
E_LDR(PC,R7,offset_start);          // jump to start
```

5 Instruction set

The following section outlines the EZH instruction set. The Section 5 provides a summary of the instructions:

Table 12. EZH instruction set summary

No	Instruction set	Function	Performance
1	`E_MOV`, `LOAD_SIMM`, and `LOAD_IMM`	Moving data/Load Shift/Load immediate data	Single cycle
2	`E_ADD`	Adding data; allows postshift (LSL/LSR/ASR/ROR) 0-31 bits	Single cycle
3	`E_SUB`	Subtracting data; allows postshift (LSL/LSR/ASR/ROR) by 0-31 bits	Single cycle
4	`E_ADC`	Adding with carry flag; allows postshift (LSL/LSR/ASR/ROR) by 0-31 bits	Single cycle

Table 12. EZH instruction set summary*...continued*

No	Instruction set	Function	Performance
5	E_SBC	Subtracting with carry flag; allows postshift (LSL/LSR/ASR/ROR) by 0-31 bits	Single cycle
6	E_AND	ANDing data; and allows postshift (LSL/LSR/ASR/ROR) by 0-31 bits	Single cycle
7	E_OR	O Ring data; allows postshift (LSL/LSR/ASR/ROR) by 0-31 bits	Single cycle
8	E_XOR	XORing data; allows postshift (LSL/LSR/ASR/ROR) by 0-31 bits	Single cycle
9	E_ANDOR	ANDing and O Ring	Single cycle
10	E_LSL, E_LSR, E_ASR, and E_ROR	Preshifting and ALU operation	Single cycle
11	E_FEND and E_FBIT	Preswapping endianness or bitwise, followed by right shifting and ALU operation	Single cycle
12	E_RLSL, E_RLSR, E_RASR, and E_RROR	Preswapping endianness or bitwise, and ALU operation	Single cycle
13	E_BTST, E_BSET, E_BCLR, and E_BTOG	Bit manipulation (set, clear, toggle) and testing	Single cycle
14	E_MODIFY_GPO_BYTE	Clearing and toggling GPO bottom byte	Single cycle
15	E_TIGHT_LOOP	Loop without extra cost	Single cycle
16	E_HOLD	Halt execution until the Boolean pattern is matched	Single cycle
17	E_NOP	No operation in one cycle	Single cycle or more, see instruction detail
18	E_HEART_RYTHM	Loading the Heartbeat counter	Single cycle
19	E_SYNCH_ALL_TO_BEAT	Enable synchronizing all instructions to beat feature	Single cycle
20	E_WAIT_FOR_BEAT	Halting until next beat	Single cycle or more until next beat
21	E_INT_TRIGGER	Triggering a single cycle interrupt output	Single cycle
22	E_GOTO	Branching	Single cycle for scheduled branch Three cycles for an ordinary branch
23	E_GOSUB	Branching to an immediate 32-bit address location	Three cycles
24	E_LDR	Data reading from chip into EZH register, optional postoffset, or preoffset update of pointer	Single cycle or more, see instruction detail
25	E_STR	Data writing from EZH register to chip memory, optional postoffset, or preoffset update of pointer	Single cycle or more, see instruction detail
26	E_LDR_REG	Data reading from the chip into the EZH register by using the register address,	Single cycle or more, see instruction detail

Table 12. EZH instruction set summary*...continued*

No	Instruction set	Function	Performance
27	`E_STR_REG`	Data writing from EZH register to chip memory by using the register address	Single cycle or more, see instruction detail
28	`E_PER_READ` and `E_PER_WRITE`	Data reading/writing from a peripheral	Single cycle or more, see instruction detail
29	`E_PUSH` and `E_POP`	Stach push or pop	Single cycle or more, see instruction detail

The following Table 13 is an explanation of the notation used.

Table 13. Opcode symbol notation

Symbol	Notation
ADD	Add operation
SUB	Subtract operation
ADC	Add operation with Carry
SBC	Subtract operation with Carry
AND	AND boolean operation
OR	OR boolean operation
XOR	XOR boolean operation
COND	Conditional
N	Invert ALU result
S	Set Flags (except for LDRB instruction where S stands for ‘signed’ access)
F/FEND	Byte order flip operation
FBIT	Bits order flip operation
LSL	Logical shift left
LSR	Logical shift right
ASR	Arithmetic shift right
ROR	Right rotate
IMM	Immediate
R/REG	Register based argument or operand
NRA	No update of RA register
LV	Large vector table
ACC	Accelerated vectored hold
PER	Peripheral
BTST	Bit test
BSET	Bit set
BCLR	Bit clear
BTOG	Bit toggle

The opcodes are constructed in the same order as they are executed in the pipeline. For example, `E_SUBN_LSLS` indicates execution in the order: subtraction, followed by negation, then a logical shift left, and finally setting the flags.

The opcode for tables must be read from left to right. Any fields highlighted in orange are optional and can be omitted. For all other columns, you must select one valid combination: one option from the first column, combined with one from the second, and followed by one from the third column. If an option is color coded, the corresponding argument must use the same color. If not, it must be omitted. Refer to the Table 14.

Table 14. Instruction format example: MOV and MVN

Instruction macro Prefix	Opcode name	Optional suffix	Arguments
E_	MOV	S	DST, SRC1
	MVN		
E_COND_			Flag, DST, SRC1

The above Table 14 shows following instruction options:

- `E_MOV(DST, SRC1)`
- `E_MOVS(DST, SRC1)`
- `E_MVN(DST, SRC1)`
- `E_MVNS(DST, SRC1)`
- `E_COND_MOV(Flag, DST, SRC1)`
- `E_COND_MOVS(Flag, DST, SRC1)`
- `E_COND_MVN(Flag, DST, SRC1)`
- `E_COND_MVNS(Flag, DST, SRC1)`

5.1 E_MOV

`E_MOV` opcode is used to move data either from one register to another, or from immediate data to a register.

Supported options:

- Conditional execution
- Set flag
- Invertible data
- Register-to-register transfer
- Immediate-to-register transfer

Note: *IMM is an 11-bit sign value that can be shifted left by 0 to 31 positions.*

Table 15. `E_MOV` instruction format

Instruction macro prefix	Opcode name	Optional suffixes		Arguments
E_	MOV	S		Flag, DST, SRC1
	MVN			
E_COND_				
E_	LOAD_IMM	N	S	Flag, DST, imm11s, shift5
	LOAD_SIMM			
E_COND_				

Restrictions:

- SPO or SNE flags are not supported.
- When the destination is PC, setting the flags is not allowed. In such cases, the PC branch is considered as scheduled instead.

5.2 E_ADD

`E_ADD` opcode is used to add data either from one register to another, or from immediate data to a register.

Supported options:

- Conditional Execution
- Set flag
- Invertible data
- Register-to-register operation
- Result undergoes a byte order flip
- Postshift operation (LSL, LSR, ASR, ROR) by 0 bits to 31 bits

Note: *IMM is a 12-bit sign value.*

Table 16. `E_ADD` instruction format - Register-to-register addition with postshift

Instruction macro prefix	Opcode name	Optional suffixes				Arguments
E_	ADD	N	F	LSL	S	Flag, DS, SRC1, SRC2, Shift5imm
E_COND_				LSR		
				ASR		
				ROR		

Table 17. `E_ADD` instruction format - Immediate addition

Instruction macro prefix	Opcode name	Optional suffixes			Arguments
E_	ADD	N	_IMM	S	Flag, DST, SRC1, imm12s
E_COND_					

Restriction:

When the destination is PC, setting the flags is not allowed. In such cases, the PC branch is considered as scheduled instead.

5.3 E_SUB

`E_SUB` opcode is used to subtract data either from one register to another, or from immediate data to a register.

Supported options:

- Conditional execution
- Set flag
- Invertible data
- Register-to-register operation
- Result undergoes a byte order flip

- Postshift operation (LSL, LSR, ASR, ROR) by 0 bits to 31 bits

Note: *IMM is a 12-bit sign value.*

Table 18. `E_SUB` instruction format - Register-to-register subtraction with postshift

Instruction macro prefix	Opcode name	Optional suffixes				Arguments
E_	SUB	N	F	LSL	S	Flag, DST, SRC1, SRC2, Shift5imm
E_COND_				LSR		
				ASR		
				ROR		

Table 19. `E_SUB` instruction format - Immediate subtraction

Instruction macro prefix	Opcode name	Optional suffixes			Arguments
E_	SUB	N	_IMM	S	Flag, DST, SRC1, imm12s
E_COND_					

Restriction:

When the destination is PC, setting the flags is not allowed. In such cases, the PC branch is considered as scheduled instead.

5.4 E_ADC

`E_ADC` opcode enables addition of the carry flag along with data, either from one register to another, or from immediate data to a register.

Supported options:

- Conditional execution
- Set flag
- Invertible data
- Register-to-register operation
- Result undergoes a byte order flip
- Postshift (LSL, LSR, ASR, ROR) by 0 bits to 31 bits

Note: *IMM is a 12-bit sign value.*

Table 20. `E_ADC` instruction format - Register-to-register addition with carry and postshift

Instruction macro prefix	Opcode name	Optional suffixes				Arguments
E_	ADC	N	F	LSL	S	Flag, DST, SRC1, SRC2, Shift5imm
E_COND_				LSR		
				ASR		
				ROR		

Table 21. E_ADC instruction format - Immediate action with carry

Instruction macro prefix	Opcode name	Optional suffixes			Arguments
E_	ADC	N	_IMM	S	Flag, DST, SRC1, imm12s
E_COND_					

Restriction:

When the destination is PC, setting the flags is not allowed. In such cases, the PC branch is considered as scheduled instead.

5.5 E_SBC

E_SBS opcode is used to subtract data from one register and another, followed by subtracting the carry flag. It can also subtract immediate data from a register, and then carry the bag.

Supported options:

- Conditional execution
- Set flag
- Invertible data
- Register-to-register operation
- Result undergoes a byte order flip
- Postshift (LSL, LSR, ASR, ROR) by 0 bits to 31 bits

Note: *IMM is a 12-bit sign value.*

Table 22. E_SBS instruction format - Register-to-register subtraction with carry and postshift

Instruction macro prefix	Opcode name	Optional suffixes				Arguments
E_	SBC	N	F	LSL	S	Flag, DST, SRC1, SRC2, Shift5imm
E_COND_				LSR		
				ASR		
				ROR		

Table 23. E_SBS instruction format - Immediate subtraction with carry

Instruction macro prefix	Opcode name	Optional suffixes			Arguments
E_	SBC	N	_IMM	S	Flag, DST, SRC1, imm12s
E_COND_					

Restriction:

When the destination is PC, setting the flags is not allowed. In such cases, the PC branch is considered as scheduled instead.

5.6 E_AND

E_AND opcode is used to perform a bitwise AND operation using data either from one register to another, or from immediate data to a register.

Supported options:

- Conditional execution
- Set flag
- Invertible data
- Register-to-register operation
- Result undergoes a byte order flip
- Postshift (LSL, LSR, ASR, ROR) by 0 bits to 31 bits

Note: *IMM is a 12-bit sign value.*

Table 24. `E_AND` instruction format - Register-to-register AND with postshift

Instruction macro prefix	Opcode name	Optional suffixes				Arguments
E_	AND	N	F	LSL	S	Flag, DST, SRC1, SRC2, Shift5imm
E_COND_				LSR		
				ASR		
				ROR		

Table 25. `E_AND` instruction format - Immediate AND

Instruction macro prefix	Opcode name	Optional suffixes			Arguments
E_	AND	N	_IMM	S	Flag, DST, SRC1, imm12s
E_COND_					

Restriction:

When the destination is PC, setting the flags is not allowed. In such cases, the PC branch is considered as scheduled instead.

5.7 E_OR

`E_OR` opcode is used to perform a bitwise OR operation using data either from one register to another, or from immediate data to a register.

Supported options:

- Conditional execution
- Set flag
- Invertible data
- Register-to-register operation
- Result undergoes a byte order flip
- Postshift (LSL, LSR, ASR, ROR) by 0 bits to 31 bits

Note: *IMM is a 12-bit sign value.*

Table 26. `E_OR` instruction format - Register-to-register OR with postshift

Instruction macro prefix	Opcode name	Optional suffixes				Arguments
E_	OR	N	F	LSL	S	Flag, DST, SRC1, SRC2, Shift5imm
E_COND_				LSR		
				ASR		

Table 26. E_OR instruction format - Register-to-register OR with postshift...*continued*

Instruction macro prefix	Opcode name	Optional suffixes				Arguments
				ROR		

Table 27. E_OR instruction format - Immediate OR

Instruction macro prefix	Opcode name	Optional suffixes			Arguments
E_	OR	N	_IMM	S	Flag, DST, SRC1, imm12s
E_COND_					

Restriction:

When the destination is PC, setting the flags is not allowed. In such cases, the PC branch is considered as scheduled instead.

5.8 E_XOR

E_XOR opcode is used to perform a bitwise XOR operation using data either from one register to another, or from immediate data to a register.

Supported option:

- Conditional execution
- Set flag
- Invertible data
- Register-to-register operation
- Result undergoes a byte order flip
- Postshift (LSL, LSR, ASR, ROR) by 0 bits to 31 bits

Note: *IMM is a 12-bit sign value.*

Table 28. E_XOR instruction format - Register-to-register XOR with postshift

Instruction macro prefix	Opcode name	Optional suffixes				Arguments
E_	XOR	N	F	LSL	S	Flag, DST, SRC1, SRC2, Shift5imm
E_COND_				LSR		
				ASR		
				ROR		

Table 29. E_XOR instruction format - Immediate XOR

Instruction macro prefix	Opcode name	Optional suffixes			Arguments
E_	XOR	N	_IMM	S	Flag, DST, SRC1, imm12s
E_COND_					

Restriction:

When the destination is PC, setting the flags is not allowed. In such cases, the PC branch is considered as scheduled instead.

5.9 E_ANDOR

E_ANDOR opcode is used to perform both AND and OR operations on two register values together with a third. This operation is useful for simultaneously setting and clearing specific bits.

Supported options:

- Conditional execution
- Set flag

Table 30. E_ANDOR instruction format

Instruction macro prefix	Opcode name	Optional suffixes	Arguments
E_	ANDOR	S	Flag, DST, Rsrc, Rand, Ror
E_COND_			

Restrictions:

- SPO or SNE flags are not supported.
- When the destination is PC, setting the flags is not allowed. In such cases, the PC branch is considered as scheduled instead.

5.10 E_LSL, E_LSR, E_ASR, and E_ROR

E_LSL, E_LSR, E_ASR, and E_ROR opcodes are used to perform preshifting, prerotating, and ALU operations.

Supported options:

- Conditional execution
- Set flag
- Logical shift left (LSL), Logical shift right (LSR), Arithmetic shift right (ASR), and Rotate right (ROR) by imm between 0 to 31
- ALU operation

Table 31. E_LSL, E_LSR, E_ASR, and E_ROR instruction format

Instruction macro prefix	Opcode name	Optional suffixes		Arguments
E_	LSL	ADD	S	Flag, DST, Roperand, R2shift, Shift5imm
E_COND_	LSR	ADC		
	ASR	SUB		
	ROR	SBC		
		AND		
		OR		
		XOR		

Restriction:

When the destination is PC, setting the flags is not allowed. In such cases, the PC branch is considered as scheduled instead.

5.11 E_FEND and E_FBIT

`E_FEND` and `E_FBIT` opcodes are used for preswapping bytes or bits order flip operations, followed by right shifting and an ALU operation.

Supported options:

- Conditional execution
- Set flag
- Logical shift left (LSL), Logical shift right (LSR), Arithmetic shift right (ASR), and Rotate right (ROR) by immediate number between 0 to 31
- ALU operation

Table 32. `E_FEND` and `E_FBIT` instruction format

Instruction macro prefix	Opcode name	Optional suffixes			Arguments
E_	FEND	LSR	ADD	S	Flag, DST, Roperand, R2shift, Shift5imm
E_COND_	FBIT	ASR	ADC		
			SUB		
			SBC		
			AND		
			OR		
			XOR		

Restriction:

When the destination is PC, setting the flags is not allowed. In such cases, the PC branch is considered as scheduled instead.

5.12 E_RLSL, E_RLSR, E_RASR, and E_RROR

`E_RLSL`, `E_RLSR`, `E_RASR`, and `E_RROR`

This opcode is used for pre or post operation byte or bit order flipping, based on the amount specified by the lowest byte of the register (ranging from 0 to 255), in combination with an ALU operation.

Supported options:

- Conditional execution
- Set flag
- Logical shift left (LSL), Logical shift right (LSR), Arithmetic shift right (ASR), and Rotate right (ROR) using lower 8-bits of a shift register
- Optional ALU operation
- Optional inversion of result

Table 33. `E_RLSL`, `E_RLSR`, `E_RASR`, and `E_RROR` instruction format - ALU operation followed by shift

Instruction macro prefix	Opcode name	Optional suffixes			Arguments
E_	RLSL	ADD	N	S	Flag, DST, Roperand, R2shift, Rshift
E_COND_	RLSR	ADC			
	RASR	SUB			

Table 33. E_RLSL, E_RLSR, E_RASR, and E_RROR instruction format - ALU operation followed by shift...*continued*

Instruction macro prefix	Opcode name	Optional suffixes			Arguments
	RROR	SBC			
		AND			
		OR			
		XOR			

Table 34. E_RLSL, E_RLSR, E_RASR, and E_RROR instruction format - Shift operation followed ALU

Instruction macro prefix	Optional suffixes		Opcode name	Opti suffi	Arguments
E_	ADD	N	RLSL	S	Flag, DST, Roperand, R2shift, Rshift
E_COND_	ADC		RLSR		
	SUB		RASR		
	SBC		RROR		
	AND				
	OR				
	XOR				

Restriction:

When the destination is PC, setting the flags is not allowed. In such cases, the PC branch is considered as scheduled instead.

5.13 E_BTST, E_BSET, E_BCLR, and E_BTOG

E_BTST, E_BSET, E_BCLR, and E_BTOG opcodes are used for bit manipulation (set, clear, toggle) and bit testing.

Supported options:

- Conditional execution
- Set flag
- Bit selection using a register or a 5-bit immediate value

Table 35. Bit manipulation instruction format using register bit select

Instruction macro prefix	Opcode name	Optional suffixes	Arguments
E_	BTST	S	Flag, DST, Rdata, Rbit
E_COND_	BSET		
	BCLR		
	BTOG		

Table 36. Bit manipulation instruction format using immediate bit select

Instruction macro prefix	Opcode name	Immediate field	Optional suffixes	Arguments
E_	BTST	_IMM	S	V, DST, Rdata, imm5
E_COND_	BSET			
	BCLR			
	BTOG			

Restriction:

- SPO or SNE flags are not supported.
- When the destination is PC, setting the flags is not allowed. In such cases, the PC branch is considered as scheduled instead.

5.14 E_MODIFY_GPO_BYTE

`E_MODIFY_GPO_BYTE` opcode is used for single cycle setting, clearing, and toggling of the GPO bottom byte.

Supported option:

- 8-bit immediate AND mask, OR mask, and XOR mask

Table 37. `E_MODIFY_GPO_BYTE` instruction format

Instruction macro prefix	Opcode name	Arguments
E_	MODIFY_GPO_BYTE	AND8imm, OR8imm, XOR8imm

5.15 E_TIGHT_LOOP

`E_TIGHT_LOOP` is an execution supervisor opcode used to define a section of code that must be repeated automatically, along with the number of times the repetition must occur.

Supported options:

- Conditional execution
- Register is used to point to the end of the loop and should be set to the address of the instruction immediately following the last opcode to be repeated.
- Register specifies how many times to repeat the block after its initial execution, meaning the code runs one more time than the count indicates.

Table 38. `E_TIGHT_LOOP` instruction format

Instruction macro prefix	Opcode name	Arguments
E_	TIGHT_LOOP	Flag, Rend, Rcount
E_COND_		

Restriction:

- The instruction immediately following `E_TIGHT_LOOP` is executed only once. The following instruction is considered the start of the loop to be repeated.
- To repeat a single instruction, set the Rend register to point 3 words ahead of the `E_TIGHT_LOOP` opcode address.

5.16 E_HOLD

E_HOLD opcode is used to halt execution until a Boolean pattern is matched.

Supported options:

- Conditional execution
- Vectored HOLD (that is branch to an appropriate vector when the pattern is matched)
- Optional exclusion from backing up the next address on the RA register (by default, RA is updated)
- Large vector table entries (4 words per entry; default is 1)
- Vector acceleration
- If there is Vectored HOLD, the vector table base is passed through the register. The address of the first vector is calculated as Rptable value+ 4 (allowing the use of PC as the base pointer if the table follows two opcodes that do not get flushed during acceleration)
- 8-bit immediate value for selecting which vectors are to be acc.elerated when matched (set bit 1 in the appropriate position of the vector that needs acceleration)

Table 39. E_HOLD instruction format - Simple HOLD

Instruction macro prefix	Opcode name	Arguments
E_	HOLD	Flag
E_COND_		

Table 40. E_HOLD instruction format - Vectored HOLD with acceleration

Instruction macro prefix	Opcode name	Optional suffixes			Arguments
E_	ACC_	VECTORED_HOLD	LV	NRA	Flag, Rptable, mask8imm
E_COND_					

5.17 E_NOP

E_NOP performs no operation.

No operation

Supported option:

None

Table 41. E_NOP instruction format

Instruction macro prefix	Opcode name
E_	NOP

5.18 E_HEART_RYTHM

E_HEART_RYTHM opcode is used for the loading the heartbeat counter. This can be achieved either through an immediate value or from a register.

Supported option:

Load using immediate value or load from a register.

Table 42. `E_HEART_RYTHM` instruction format

Instruction macro prefix	Opcode name	Arguments
E_	HEART_RYTHM_IMM	IMM16
E_	HEART_RYTHM	Rcount

5.19 E_SYNCH_ALL_TO_BEAT

`E_SYNCH_ALL_TO_BEAT` opcode is used to enable or disable the synchronization of all instructions to the beat feature.

This opcode is for switching on or off the 'synchronize all instructions to beat' feature.

Supported option:

Boolean control (true or false)

Table 43. `E_SYNCH_ALL_TO_BEAT` instruction format

Instruction macro prefix	Opcode name	Arguments
E_	SYNCH_ALL_TO_BEAT	Bool1imm

5.20 E_WAIT_FOR_BEAT

`E_WAIT_FOR_BEAT` is used to halt execution until the next heartbeat cycle.

Before using this instruction, the heartbeat must be initialized using the `E_WAIT_FOR_BEAT` instruction.

Example：

`E_HEART_RYTHM_IMM`(time_delay_counter)

`E_WAIT_FOR_BEAT`

Supported option:

None

Table 44. `E_WAIT_FOR_BEAT` instruction format

Instruction macro prefix	Opcode name
E_	WAIT_FOR_BEAT

Restriction:

None

Performance:

Halts for as many cycles as it takes to reach the next heartbeat.

5.21 E_INT_TRIGGER

`E_INT_TRIGGER` opcode is for triggering a single cycle interrupt output from both the common interrupt output and any of the 24 indivudial interrupt out channels.

Supported option:

An immediate value is used to specify which interrupts should be triggered.

Note: *The common interrupt output is always triggered.*

Table 45. `E_INT_TRIGGER` instruction format

Instruction macro prefix	Opcode name	Arguments
E_	INT_TRIGGER	(channels24imm)

5.22 E_GOTO

`E_GOTO` opcode is used for branching.

Supported options:

- Conditional execution
- Branch to address:
  - 21-bit immediate (branch address has both LSB[1:0] cleared and 9 top bits remain same as PC at time of execution) or address 32-bit set by register
  - 32-bit address set by the register
- Optional back up of the next address in RA:
  - If there is a scheduled branch, the address backed-up in RA is the one after the two instructions that do not get flushed.
  - For a scheduled branch, RA stores the current PC minus 4.
  - For an ordinary branch, RA stores the current PC plus 4.
- Optional scheduled branch (unconditional, or if NZ flag is set)

Table 46. `E_GOTO` instruction format using immediate addressing

Instruction macro prefix	Opcode name	Optional suffixes	Arguments
E_	GOTO	L	Flag, addr21imm
E_COND_			

Table 47. `E_GOTO` instruction format using register addressing

Instruction macro prefix	Opcode name	Optional suffixes		Arguments
E_	GOTO	REG	L	(Flag, Raddr)
E_COND_				

Restrictions:

- The Arm Compiler Arm-clang in the MDK IDE does not support this instruction.
- You can replace this opcode with a PC branch opcode.

Performance:

- Single Cycle – for a scheduled branch (that is, the next two opcodes do not get flushed)
- Three cycles – for ordinary branch

Note:

- *SPO or UNS condition causes a scheduled branch.*

- *If NZ is set (otherwise, no branch), SNE or NZS condition causes a scheduled branch.*

5.23 E_GOSUB

`E_GOSUB` opcode is used to branch to an immediate 32-bit address location. The RA register is updated with the return address to allow stacking and returning.

Supported option:

30-bit immediate address (word address)

Table 48. `E_GOSUB` instruction format

Instruction macro prefix	Opcode name	Arguments
E_	GOSUB	addr30imm

5.24 E_LDR

`E_LDR` opcode is used for data reading from chip into EZH register (using immediate offset). See `E_LDR_REG` for register plus address calculation.

Supported options:

- Conditional Execution
- Optional read byte (default is word)
- Optional read byte with sign extension
- Offset by immediate (8-bit signed; word pointer offset for word access, byte pointer offset for byte access)
- Optional postoffset or preoffset pointer update

Table 49. `E_LDR` instruction format - Standard load

Instruction macro prefix	Opcode name	Optional suffixes	Arguments
E_	LDR	PRE	Flag, DST, Raddr, offset8imms
E_COND_		POST	

Table 50. `E_LDR` instruction format - Byte or sign extended load

Instruction macro prefix	Opcode name	Optional suffixes			Arguments
E_	LDR	B	S	PRE	Flag, DST, Raddr, offset8imms
E_COND_				POST	

Restrictions:

- Post increment of PC is not allowed.
- Pointer cannot have a postupdate and be the destination for the read. In that case, the postupdate is ignored.
- There is no support for SPO or SNE algorithmic flags.
- If CFS is used as a pointer for a read, the bottom 8-bits of CFS is 0x00.
- CFS cannot be used as a destination for a byte read.

Performance:

- The read is submitted in a single cycle. However, if the destination register is used for any other operation (apart from a write), execution stalls until the data returns from the AHB data bus.
- The total latency, including the cycle for submitting the read, is at least 2 cycles. If data is not required immediately, the DFETCH unit collects the data in the background.
- The DFETCH unit buffers one transaction for later issue. However, if two consecutive reads are made into the same register, the second read executes only after the previous read completes.

5.25 E_STR

`E_STR` opcode is used for data writing from the EZH register to chip memory. See `E_STR_REG` for register plus address calculation.

Supported options:

- Conditional execution
- Optional write byte (default is word)
- Offset by immediate (8-bit signed, word pointer offset if there is word access, byte pointer offset if there is byte)
- Optional postoffset or preoffset update of pointer register

Table 51. `E_STR` instruction format - Standard store

Instruction macro prefix	Opcode name	Optional suffixes	Arguments
E_	STR	PRE	Flag, Raddr, Rdata, offset8imms
E_COND_		POST	

Table 52. `E_STR` instruction format - Byte store

Instruction macro prefix	Opcode name	Optional suffixes		Arguments
E_	STR	B	PRE	Flag, Raddr, Rdata, offset8imms
E_COND_			POST	

Restrictions:

- Post increment of PC is not allowed.
- If a pointer is both postupdated and used as the destination for the read, the postupdate is ignored.
- There is no support for SPO or SNE algorithmic flags.

Performance:

The performance characteristics of `E_STR` are the same as those of `E_LDR`.

5.26 E_LDR_REG

`E_LDR_REG` opcode is used for data reading from chip into the EZH register using an address plus register offset.

Supported options:

- Conditional execution
- Optional read byte (default is word)
- Optional read byte with sign extension

- Offset by register(offset size is one byte)

Table 53. `E_LDR_REG` instruction format - Standard load with register offset

Instruction macro prefix	Opcode name	Arguments
E_	LDR	Flag, DST, Raddr, Roffset
E_COND_		

Table 54. `E_LDR_REG` instruction format - Byte or sign extended load with register offset

Instruction macro prefix	Opcode name	Optional suffixes		Arguments
E_	LDR	B	S	Flag, DST, Raddr, Roffset
E_COND_				

Restrictions:

- There is no support for SPO or SNE algorithmic flags.
- If CFS is used as the address pointer for a read, the bottom 8-bits of CFS is 0x00.
- CFS cannot be used as a destination for a byte read.

Performance:

- The read is submitted in a single cycle. However, if the destination register is used for any other operation (apart from a write), execution stalls until the data returns from the AHB data bus.
- The total latency, including the cycle for submitting the read, is at least 2 cycles. If data is not required immediately, the DFETCH unit collects the data in the background.
- The DFETCH unit buffers one transaction for later issue. However, if two consecutive reads are made into the same register, the second read executes only after the previous read completes.

5.27 E_STR_REG

`E_STR_REG` opcode is used for data writing from the EZH register to chip memory using an address and offset from registers.

Supported options:

- Conditional execution
- Optional write byte (default is word)
- Offset by register

Table 55. `E_STR_REG` instruction format - Standard store with register offset

Instruction macro prefix	Opcode name	Arguments
E_	STR	Flag, Raddr, Rdata, Roffset
E_COND_		

Table 56. `E_STR_REG` instruction format - Byte store with register offset

Instruction macro prefix	Opcode name	Opti suffi	Arguments
E_	STR	B	Flag, Raddr, Rdata, Roffset
E_COND_			

Restriction:

There is no support for SPO or SNE algorithmic flags.

Performance:

The performance characteristics are the same as those of `LDR_REG`.

5.28 E_PER_READ and E_PER_WRITE

`E_PER_READ` and `E_PER_WRITE` opcodes are used for data reading and writing from the peripheral region of the Cortex-M family with immediate addressing (useful for saving the need to use a register for calculating offset into the peripheral).

Supported options:

- Conditional execution
- 20-bit immediate offset from 0x4000 0000
  Note: *The bottom two bits are set to 0; therefore, this covers a 22-bit offset region in the Cortex-M family peripheral memory range.*

Table 57. Peripheral read format

Instruction macro prefix	Opcode name	Arguments
E_	PER_READ	Flag, DST, offset20imm
E_COND_		

Table 58. Peripheral write format

Instruction macro prefix	Opcode name	Arguments
E_	PER_WRITE	Flag, SRC1, offset20imm
E_COND_		

Restriction:

There is no support for SPO or SNE algorithmic flags.

Performance:

The performance characteristics are the same as those of the `E_LDR` instruction.

5.29 E_PUSH and E_POP

`E_PUSH` and `E_POP` opcodes are used for pushing and popping onto the stack. It is important to note that the stack grows upward in terms of address space.

Supported options:

- Conditional execution
- Optional stacking of byte (default is word)
- Optional read byte with sign extension

Table 59. Standard push or pop format

Instruction macro prefix	Opcode name	Arguments
E_	PUSH	Flag, Reg
E_COND_	POP	

Table 60. Byte or sign extended push format

Instruction macro prefix	Opcode name	Optional suffixes		Arguments
E_	PUSH	B	S	Flag, Reg
E_COND_	POP			

Restrictions:

- There is no support for SPO or SNE algorithmic flags.
- Although byte stacking is supported, the users must not mix byte and word stacking across non-word-aligned boundaries. For example, the bottom two bits of the SP register are ignored in word access address calculation.

Performance:

The performance characteristics are the same as those of `E_LDR/E_STR`.

6 Note about the source code in the document

Example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2025 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN

ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

7 Revision history

Table 61 summarizes the revisions to this document.

Table 61. Revision history

Document ID	Release date	Description
AN14650 v.1.0	28 May 2025	Initial public release

Legal information

Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at https://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Suitability for use in non-automotive qualified products — Unless this document expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

HTML publications — An HTML version, if available, of this document is provided as a courtesy. Definitive information is contained in the applicable document in PDF format. If there is a discrepancy between the HTML document and the PDF document, the PDF document has priority.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP B.V. — NXP B.V. is not an operating company and it does not distribute or sell products.

Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

Contents