C29x CPU

Reference Guide

Literature Number: SPRUIY2A
NOVEMBER 2024 – REVISED MARCH 2025

Table of Contents

List of Figures

List of Tables

Read This First

About This Manual

This manual describes the CPU architecture, interrupt, addressing modes, safety and security aspects of the CPU. This manual also describes emulation features available on these devices. A summary of the chapters follows.

Architectural Overview

This chapter introduces the CPU that is at the heart of each F29x device. The chapter includes a memory-map and a high-level description of the memory interface that connects the core with memory and peripheral devices.

Central Processing Unit

This chapter describes the architecture, registers, and primary functions of the CPU. The chapter includes detailed descriptions of the flag and control bits in the most important CPU registers, status registers ISTS, DSTS, and ESTS.

CPU Interrupts Architecture Overview

This chapter describes the interrupts and how the interrupts are handled by the CPU. The chapter also explains the effects of a reset on the CPU and includes discussion of the automatic context save performed by the CPU prior to servicing an interrupt.

Addressing Modes

This chapter explains the modes that the assembly language instructions accept data and access register and memory locations. The chapter includes a description of how addressing-mode information is encoded in opcodes.

Safety And Security Unit

This chapter describes safety and security approach adopted by F29x architecture. This chapters explains the concepts of task isolation, LINK, STACK, and ZONE with examples.

Emulation Features

This chapter describes the F29x emulation features that can be used with only a JTAG port and two additional emulation pins.

Related Documentation from Texas Instruments

For a complete listing of related documentation and development-support tools for these devices, visit the Texas Instruments website at www.ti.com.

Glossary

TI Glossary	This glossary lists and explains terms, acronyms, and definitions.

Support Resources

TI E2E™ support forums are an engineer's go-to source for fast, verified answers and design help — straight from the experts. Search existing answers or ask your own question to get the quick design help you need.

Linked content is provided "AS IS" by the respective contributors. They do not constitute TI specifications and do not necessarily reflect TI's views; see TI's Terms of Use.

Trademarks

TI E2E™ is a trademark of Texas Instruments.
All trademarks are the property of their respective owners.

Chapter 1

Architecture Overview

The C29x is a floating-point CPU in the C2000 family. This chapter provides an overview of the architectural structure and components of the CPU.

1.1 Introduction to the CPU

The C29x CPU is a VLIW (Very Long Instruction Word) architecture with a fully protected pipeline. The C29x CPU supports multiple instruction sizes (16/32/48 bits), a variable instruction packet size which can contain multiple instructions that execute in parallel. For example, the C29x CPU architecture can execute up to eight instructions in parallel. This is enabled by multiple functional units inside the CPU that can execute concurrently. A total of 64 working registers, broken into three different categories (Ax, Dx, and Mx register banks) to support the parallel operations in the CPU. In addition to the working registers, the CPU contains multiple status registers (DSTS, ESTS, and ISTS) that maintain different execution and interrupt context related information.

1.2 Data Type

The C29x CPU supports the following data types in memory:

Support 8, 16, 32, 64 Data Types: The CPU supports 8-, 16-, 32-, and 64-bit operations. The CPU can read and write to memory 8-, 16-, 32-, and 64-bit sized data in a single operation (cycle).

Little-Endian Format: All data and registers use little-endian format.

Data Aligned to Word Size Boundaries: A 16-bit access needs to be aligned to a 16-bit word boundary (Address Line 0 = 0). A 32-bit access needs to be aligned to a 32-bit word boundary (Address Lines 1,0 = 0,0). A 64-bit access needs to be aligned to a 64-bit word boundary (Address Lines 2,1,0 = 0,0,0).

32-bit and 64-bit Floating Point: The C29x CPU supports 32-bit and 64-bit floating-point operations using the IEEE format. The values can be moved between fixed-point and floating-point registers without incurring memory stalls.

C Compiler Data Type Compatibility:

Size	C29x CPU Data Type Definitions
char	8 bits
short	16 bits
int	32 bits
long	32 bits
long long	64 bits
float	32 bits
double	64 bits
long double	64 bits
pointer	32 bits

1.3 C29x CPU System Architecture

The C29x CPU system architecture consists of the following main functional blocks as shown in Figure 1-1.

- **C29x CPU Core:** Responsible for generating data- and program-memory addresses; decoding and executing instructions; performing arithmetic, logical, and shift operations; and controlling data transfers among CPU registers, data memory, and program memory
- **CPU Stacks:** Manages the software, protected call and Realtime interrupt stacks in a secure environment.
- **CPU Interface buses:** Signals for interfacing with memory and peripherals, clocking and controlling the CPU and the emulation logic
- **Safety and Security Unit (SSU):** Implements safety, memory management (MPU) and security as one function in hardware.
- **Peripheral Interrupt Priority Expansion (PIPE):** Manages and prioritizes all peripheral interrupt sources. See the *F29H85x and F29P58x Real-Time Microcontrollers Technical Reference Manual* for more details on the PIPE.
- **C29x CPU Debug Interface:** Used for monitoring and controlling various parts and functionality of the MCU and for testing device operation. Interfaces to Debug Sub-system (DebugSS) and Embedded Real-time analysis and Diagnostics (ERAD) Units external to the CPU.

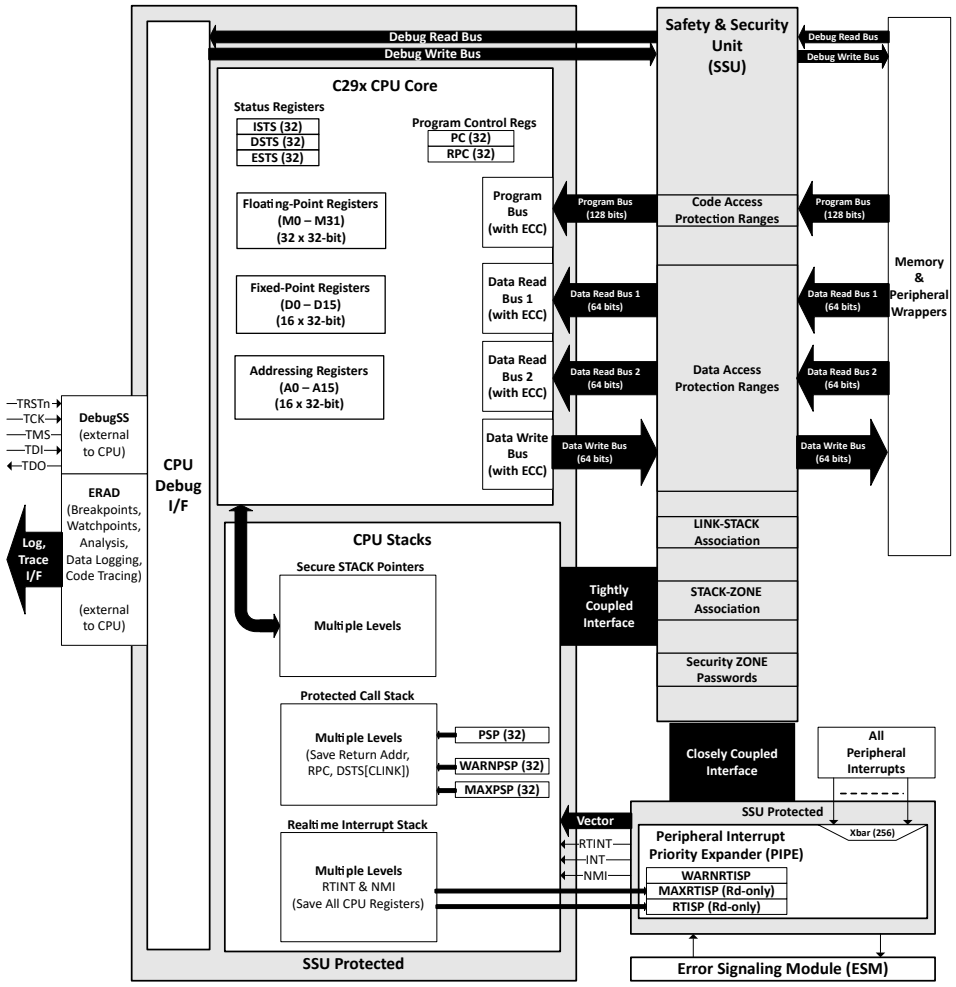

Figure 1-1. C29x CPU System Architecture

1.3.1 Emulation Logic

The emulation logic includes the following features. For more details about these features, see Chapter 6 for more details.

1. **Code Execution control**
   - Ability to download code
   - Support for breakpoints/watchpoint
   - Run, Halt, single-stepping
2. **System Visibility**
   - Access to System memory
   - Access to Peripheral memory
   - Real-time access to memory/peripherals without halting the CPU
3. **Cross Triggering**
   - Mapping events from one CPU subsystem to another
   - Event actions are handled in the respective CPU sub system
4. **Security**
   - Access paths for security challenge response from the debugger to HSM module/SSU based on the device security architecture
5. **Profiling**
   - C29x CPU code profiling is done using ERAD
6. **Trace**
   - C29x CPU trace is done using ERAD PC Discontinuity trace
7. **Reset**
   - Capability for CPU and System reset using the debugger

1.3.2 CPU Interface Buses

The C29x CPU core access code, data and peripheral resources through the following buses:

Program Bus: Program bus is used to fetch instructions from memory subsystem. Data bus width of the program bus is 128 bits. This bus can fetch 128-bits in a single cycle. The C29x CPU supports instruction packets from 16 bits up to 128 bits. Each instruction fetch is ECC protected.

Data Read Bus 1: Data read bus 1 is used to read the data from the memory subsystem or peripherals. Data bus width of the data read bus 1 is 64 bits. This bus can read 8-, 16-, 32-, and 64-bit data in a single cycle. There are two data read buses on the C29x CPU. Data from memories can be simultaneously accessed using these buses if the address falls into different physical memory banks (refer to the device-specific data sheet to identify the physical memory banks). In case of simultaneous accesses to the same bank, accesses can be arbitrated or serviced in any order. Refer to the device-specific data manual for details regarding physical banks. Data Read Bus 1 is ECC protected.

Data Read Bus 2: Data read bus 2 is used to read the data from the memory subsystem or peripherals. Data bus width of the data read bus 2 is 64 bits. This bus can read 8-, 16-, 32-, and 64-bit data in a single cycle. Data Read Bus 2 is ECC protected.

Data Write Bus: Data write bus is used to write data to the memory subsystem or peripherals. Data bus width of the data write bus is 64 bits. This can write 8-, 16-, 32-, 64-bit data in a single cycle.

Note

ECC Feature: The C29x CPU supports ECC granularity of 16/32/64 bits and has the capability for single-bit error correction. Double-bit error detection causes the CPU to enter the FAULT state. When correcting for single-bit error, the CPU stalls the PIPE for 1 cycle.

Debug Data Read Bus: The C29x CPU has a dedicated debug data read bus similar to the data read bus. Data bus width of the debug data read bus is 64 bits. This bus can read 8-, 16-, 32-, and 64-bit data in a single cycle. The Safety and Security Unit (SSU) allows or blocks the debug accesses based on the security settings.

Debug Data Write Bus: The C29x CPU has a dedicated debug data write bus similar to the data write bus. Data bus width of the debug data write bus is 64 bits. This bus can write 8-, 16-, 32-, and 64-bit data in a single cycle. The Safety and Security Unit (SSU) allows or blocks the debug accesses based on the security settings.

Interrupt Bus: The C29x CPU interrupt bus handles Reset, NMI, RTINT, INT interrupt signals and the interrupt vector.

ERAD Interface bus: Breakpoint and watchpoints are implemented from external to C29x CPU using the ERAD (Real-Time Analysis and Diagnostics) module. This bus is used to interface the ERAD with the C29x CPU.

SSU Interface bus: Security implementation is tightly coupled to the C29x CPU using the SSU interface bus.

Error Interface bus: Program read errors, data read errors, and data write errors are interfaced to the Error Aggregator/ESM using the error interface bus.

1.4 Memory Map

The C29x CPU has a dedicated stack pointer (SP = A15) that can access the full 32-bit address range of the CPU.

The C29x CPU supports separate program, data read, and data write buses. The memory is unified with one 4GB image. On the C29x CPU, every peripheral instance is mapped within a 4KB address range.

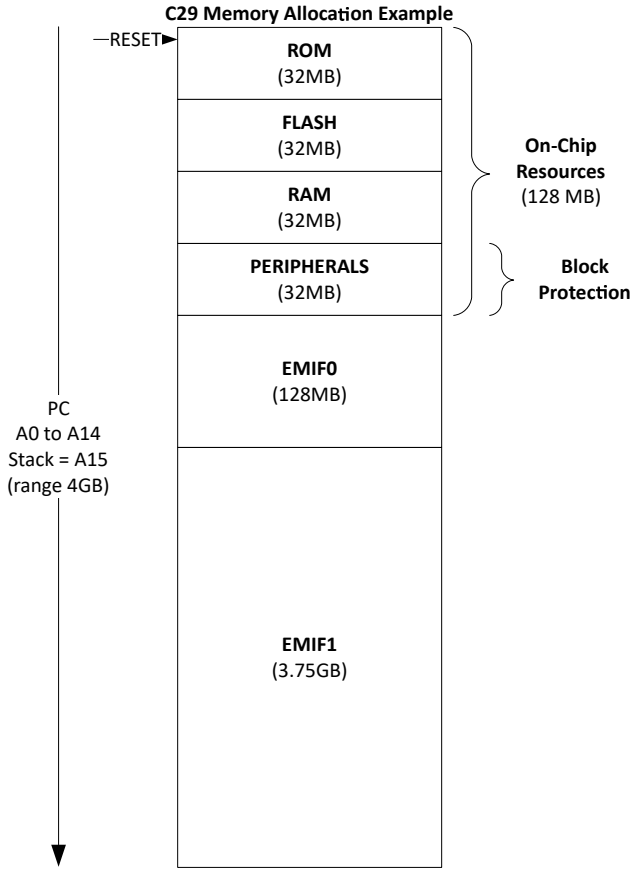

Figure 1-2. Memory Map

Block Protection: This feature protects the order of Read and Write operations to peripheral registers and avoids pipeline effects.

Chapter 2

Central Processing Unit (CPU)

The central processing unit (CPU) is responsible for controlling the flow of a program and the processing of instructions. CPU performs arithmetic, Boolean-logic, multiply, and shift operations. When performing signed math, the CPU uses two's complement notation. This chapter describes the architecture, registers, and primary functions of the CPU.

2.1 C29x CPU Architecture

The C29x CPU is a VLIW (Very Long Instruction Word) architecture with a fully protected pipeline. The CPU supports multiple instruction sizes (16/32/48 bits). The CPU also supports variable instruction packet size, with each packet able to contain up to eight instructions that execute in parallel. For example, the CPU architecture can execute up to eight 16-bit instructions in parallel. This is enabled by multiple functional units inside the CPU that can execute concurrently. A total of 64 working registers, divided into three different categories (Ax, Dx, and Mx register banks) support the parallel operations in the CPU. In addition to the working registers, the CPU contains multiple status registers (DSTS, ESTS, and ISTS) that maintain execution-related and interrupt context-related information.

2.1.1 Features

Following is the list of C29x CPU major features:

- **Ease of use:**
  - Byte addressable CPU.
  - Linear and unified memory map with 4GB address range.
  - Fully Protected Pipeline: 9 stage pipeline that prevents writes and reads from same location from occurring out of order.
  - Deterministic execution and maximum performance without cached memories.
- **Improved parallelism:**
  - Execute from 1 to 8 instructions in parallel.
  - Execute fixed-point, floating-point, and addressing operations in parallel.
  - Multiple parallel functional units.
  - Specialized operations to minimize discontinuities and accelerate decision making code (for example, if-then-else statements and switch statements).
  - Specialized operations targeting real-time control (for example, trigonometric operations and multiphase vector translation operations).
- **Improved bus throughput:**
  - Capable of fetching up to 128-bit instruction packet every cycle.
  - Capable of performing 8/16/32/64-bit dual reads and single writes per cycle.
  - Improved addressing modes reduce overhead in accessing memory and peripheral resources.
  - Improved pipeline allows for additional 0-wait memory to be accessible to CPU for maximum performance.
- **Code efficiency:**
  - Supports variable length instruction set (16-bit, 32-bit, and 48-bit instructions).
  - Rich instruction set optimizes the most common operations in smallest instructions.
- **ASIL-D safety capability with code isolation in hardware:**
  - Lock step core capable of independent execution in split-lock mode (acting as a separate core) or lock step execution (for redundancy).
  - Integrated ECC logic
  - Integrated memory management (MPU) and protection mechanisms in hardware to maximize MIPS.
  - Separate code threads are fully isolated and protected (including software stacks).
- **Multi-zone security in hardware:**
  - Run time content protection and IP protection of code.
  - Individual passwords for each zone to control access.
- **Enhanced debug and trace capabilities:**
  - Specialized data logging and code flow trace instructions.
  - Trace data capable of being logged in on-chip RAM or exported through serial communication peripherals.

2.1.2 Block Diagram

Figure 2-1 shows a block diagram of the C29x CPU.

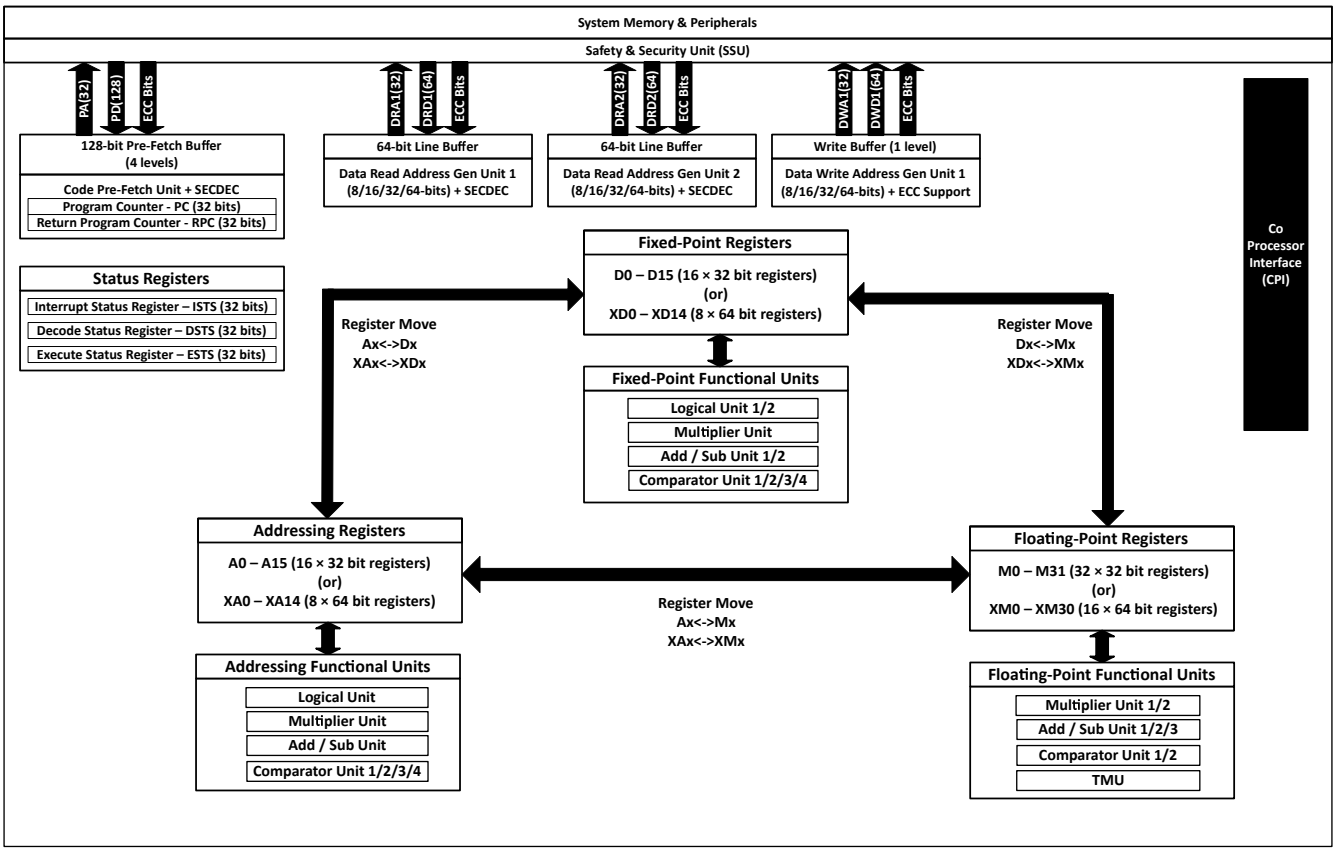

Figure 2-1. C29x CPU Block Diagram

2.2 CPU Registers

The C29x CPU core consists of following CPU registers:

- Addressing Register (Ax / XAx)
  - Stack Pointer (A15 = SP) register
- Fixed-Point Registers (Dx / XDx)
- Floating-Point Registers (Mx / XMx)
- Program Counter (PC)
- Return Program Counter (RPC)
- Status Registers
  - Interrupt Status Register (ISTS)
  - Decode Status Register (DSTS)
  - Execute Status Register (ESTS)

2.2.1 Addressing Registers (Ax/XAx)

Table 2-1. Addressing Registers (Ax/XAx)

Registers		Size	Description
A0	XA0	32 bits	**Addressing Registers (16 × Ax, 8 × XAx):** • The Ax registers are primarily used for addressing operations. All addressing modes operate on the Ax registers. • There are 16 × 32-bit addressing registers (A0-A15) or 8 × 64-bit addressing register pairs (XA0-XA14). • The Ax registers can also be used to perform MPY, ADD, COMP, SHIFT, and AND/OR/XOR operations. The registers are used to generate 32-bit addresses for the C29x CPU memory space (4GB). • The registers can be used as individual 32-bit registers or for 64-bit register pairs as 64-bit load/store operations to and from memory or 64-bit register-to-register moves (8 pairs, XA0, XA2, to XA14). • Register A15 is dedicated as the Stack Pointer (A15 = SP) register. **Value after Reset : 0x0000 0000**
A1		32 bits	
A2	XA2	32 bits	
A3		32 bits	
A4	XA4	32 bits	
A5		32 bits	
A6	XA6	32 bits	
A7		32 bits	
A8	XA8	32 bits	
A9		32 bits	
A10	XA10	32 bits	
A11		32 bits	
A12	XA12	32 bits	
A13		32 bits	
A14	XA14	32 bits	
A15 (SP)		32 bits	

2.2.2 Fixed-Point Registers (Dx/XDx)

Table 2-2. Fixed-Point Registers (Dx/XDx)

Registers		Size	Description
D0	XD0	32 bits	**Fixed-Point Registers (16 × Dx, 8 × XDx):** • The Dx registers are primarily used for performing fixed-point data operations. • There are 16 × 32-bit fixed-point registers (D0-D15) or 8 × 64-bit fixed-point registers pairs (XD0-XD14). • The registers can be used as individual 32-bit registers (Dx) or for 64-bit register pairs (XDx) as 64-bit operations, 64-bit load/store operations to and from memory or 64-bit register-to-register moves (8 pairs, XD0, XD2, to X14). **Value after Reset : 0x0000 0000**
D1		32 bits	
D2	XD2	32 bits	
D3		32 bits	
D4	XD4	32 bits	
D5		32 bits	
D6	XD6	32 bits	
D7		32 bits	
D8	XD8	32 bits	
D9		32 bits	
D10	XD10	32 bits	
D11		32 bits	
D12	XD12	32 bits	
D13		32 bits	
D14	XD14	32 bits	
D15		32 bits	

2.2.3 Floating-Point Register (Mx/XMx)

Table 2-3. Floating-Point Registers (Mx/XMx)

Registers		Size	Description
M0	XM0	32 bits	**Floating-Point Registers (32 × Mx, 16 × XMx):** • The Mx registers are primarily used for performing floating-point data operations. • There are 32 × 32-bit addressing registers (M0-M15) or 16 × 64-bit addressing register pairs (XM0-XM14). • The registers can be used as individual 32-bit registers (Mx) or for 64-bit register pairs (XMx) as 64-bit operations, 64-bit load/store operations to and from memory or 64-bit register-to-register moves (32 pairs, XM0, XM2, to XM30). **Value after Reset : 0x0000 0000**
M1		32 bits	
M2	XM2	32 bits	
M3		32 bits	
M4	XM4	32 bits	
M5		32 bits	
M6	XM6	32 bits	
M7		32 bits	
M8	XM8	32 bits	
M9		32 bits	
M10	XM10	32 bits	
M11		32 bits	
M12	XM12	32 bits	
M13		32 bits	
M14	XM14	32 bits	
M15		32 bits	
M16	XM16	32 bits	
M17		32 bits	
M18	XM18	32 bits	
M19		32 bits	
M20	XM20	32 bits	
M21		32 bits	
M22	XM22	32 bits	
M23		32 bits	
M24	XM24	32 bits	
M25		32 bits	
M26	XM26	32 bits	
M27		32 bits	
M28	XM28	32 bits	
M29		32 bits	
M30	XM30	32 bits	
M31		32 bits	

2.2.4 Program Counter (PC)

When the pipeline is full, the 32-bit PC always points to the instruction that is currently being processed. The PC increments from low to high memory and always points to the next executable instruction packet.

2.2.5 Return Program Counter (RPC)

This register holds the return address when performing a CALL or servicing a low-priority interrupt (INT). The previous RPC value is saved on the stack. On a RET (return from a function) or RETI.INT (return from a low-priority interrupt) operation, the return address is fetched from the RPC register and then RPC is restored with the value that was saved on the stack as part CALL or INT operation.

2.2.6 Status Registers

The C29x CPU core supports three status registers: ISTS (Section 2.2.6.1), DSTS (Section 2.2.6.2), and ESTS (Section 2.2.6.3) that contain flag and control bits. The ESTS and DSTS status registers are stored into and loaded from data memory, enabling the status of the CPU to be saved and restored for subroutines.

2.2.6.1 Interrupt Status Register (ISTS)

Table 2-4. Interrupt Status Register (ISTS)

Bit	Bitfield	Reset Value	Description
0	INTF	0h	This flag gets set when PIPE generates INT interrupt
1	RTINTF	0h	This flag gets set when PIPE generates RTINT interrupt
2	NMIF	0h	This flag gets set when PIPE generates NMI interrupt
3-7	RESERVED	0h	RESERVED
8-15	ATOMIC COUNTER	0h	**ATOMIC Counter:** When the ATOMIC #N operation is executed, the counter is loaded with the specified count value #N. The counter then starts decrementing on every instruction packet execution. Interrupts are blocked until the counter reaches zero. The maximum #N value supported is 64 packets.
16-19	CURRSP	0h	**Current Stack Pointer:** The C29x CPU system supports multiple software STACKs. This field reflects the current active STACK.
20-23	INTSP	0h	**Interrupt Stack Pointer:** The INT interrupt can only be executed from one selected STACK which is reflected by the INTSP field. This value is programmed in the interrupt controller (PIPE). Reset value is determined the value driven by PIPE.
24-26	RESERVED	0h	RESERVED
27-30	CURRLINK	0h	**Current LINK:** The C29x CPU security and safety system supports a concept of LINKs (described in Chapter 5). This field reflects the current active LINK value in the D2 phase.
31	RESERVED	0h	RESERVED

2.2.6.2 Decode Phase Status Register (DSTS)

Table 2-5. Decode Phase Status Register (DSTS)

Bit	Bitfield	Reset Value	Description
0	A.Z	0h	**Ax Register Operation Flags:** These flags are set on fixed-point operations involving the Ax registers. Tested conditions: A.EQ : Equal To Zero A.NEQ : Not Equal To Zero A.GT : Greater Than Zero A.GEQ : Greater Than Or Equal To Zero A.LT : Lesser than Zero A.LEQ : Lesser than (or) Equal to Zero A.HI : Higher A.HIS : Higher (or) Same A.LO : Lower A.LOS : Lower Or Same A.EQANDNZ: Equal AND Not Zero(useful for character string searches) A.NEQORZ : Not Equal OR Zero (useful for character string searches)
1	A.N	0h	
2	A.C	0h	
3	A.ZV	0h	
4-5	RESERVED	0h	RESERVED
6	DBGM	0h	Debug Mask Bit , Enables or disables debug requests.
7-10	CLINK(1)	0h	Used for indicating the origin of a protected CALL operation.
11	RESERVED	0h	RESERVED
12	TA0	0h	**Ax Register Test Flags:** These test flags can store multiple conditions by testing the Ax operation Flags. These test flags can then be used to combine multiple combinations of tested conditions. This enables the reduction of multiple conditional branch operations. Tested conditions: TAx.Z TAx Equal To Zero TAx.NZ TAx Not Equal To Zero TA.MAP(#x16ta) Test TAx FLAGS Using 4:1 LUT Combination
13	TA1	0h	
14	TA2	0h	
15	TA3	0h	
16	INTE	0h	Interrupt (INT) Enable Bit
17-18	INTS(2)	0h	**Interrupt Status:** These bits indicate the current active interrupt ISTS = 0 : Main code active ISTS = 1 : INT active ISTS = 2 : RTINT active ISTS = 3 : NMI active
19-26	ISR PRIORITY(2)	FFh	The ISR PRIORITY level is between 0 (highest priority) to 255 (lowest priority).
27-30	RLINK (1)	0h	Used for indicating the origin of a protected RET operation.
31	RESERVED	0h	RESERVED

(1) CLINK and RLINK are only updated by the hardware. Load, Move and Mask operations does not change the state of these fields.
(2) INTS and ISR PRIORITY are only updated by the hardware and RETI.INT instruction. Any other Load, Move and Mask operations does not change the state of these fields.

2.2.6.3 Execute Phase Status Register (ESTS)

Table 2-6. Execute Phase Status Register (ESTS)

Bit	Bitfield	Reset Value	Description
0	D.Z	0h	**Dx Register Operation Flags:** These flags are set on fixed-point operations involving the Dx registers. Tested conditions: D.EQ Equal To Zero D.NEQ Not Equal To Zero D.GT Greater Than Zero D.GEQ Greater Than Or Equal To Zero D.LT Less Than Zero D.LEQ Less Than Or Equal To Zero D.HI Higher D.HIS Higher Or Same D.LO Lower D.LOS Lower Or Same D.EQANDNZ Equal AND Not Zero (useful for character string searches) D.NEQORZ Not Equal OR Zero (useful for character string searches) D.OV Integer Overflow D.OVNEG Integer Overflow Negative
1	D.N	0h	
2	D.C	0h	
3	D.ZV	0h	
4	D.OV(1) (2) (3)	0h	
5	D.OVNEG(1) (2) (3)	0h	
6-7	RESERVED	0h	
8	M.ZF	0h	
9	M.NF		
10	M.LUF(1) (2)	0h	
11	M.LVF(1) (2)		
12	TDM0	0h	
13	TDM1		
14	TDM2	0h	
15	TDM3		
16	RNDF32	0h	
17	RNDF64		
18	IDIV.Z	0h	
19	IDIV.N		
20	IDIV.TF	0h	
21	FDIV.TF		
22	FDIV.N	0h	
23	TMU.TF		
24-31	RESERVED	0h	RESERVED

(1) On the C29x CPU, all flags (except D.OV, D.OVNEG, M.LUF, and M.LVF) are only affected by compare (CMP), test bit (TBIT), or test flag (TFLG) type operations. Load/Store, MPY, ADD, SUB, SHIFT, AND, OR, and XOR type operations do not affect the flags.
(2) D.OV, D.OVNEG, M.LUF, and M.LVF are sticky flags, that is, once set, the flags remain set until cleared by software.
(3) D.OV and D.OVNEG get set as a pair. D.OVNEG is updated on first occurrence of D.OV, that is, if sequence of instructions updating D.OV more than once, D.OVNEG captures overflow status on first D.OV occurrence.

2.3 Instruction Packing

The C29x CPU has a variable size instruction set. The supported instruction sizes are 16-bit, 32-bit, and 48-bit. The VLIW architecture of the CPU allows multiple instructions to be issued in a single cycle. The number of instructions that are executed in parallel is decided at build time and all the parallel instructions are packed into a single instruction packet. This section explains the formation and structure of the instruction packets. The maximum allowed instruction packet size is 128-bits. Hence, any combination of 16-bit, 32-bit, and 48-bit instructions can form an instruction packet as long as the maximum packet size is not exceeded.

The following is a non-exhaustive list of examples of valid instruction combinations within an instruction packet:

- 8 × 16-bit instructions.
- 4 × 32-bit instructions.
- 2 × 32-bit, 1 × 48-bit, 1 × 16-bit instructions.
- 3 × 32-bit, 2 × 16-bit instructions.

Table 2-7 shows the structure of the three possible instruction sizes.

Table 2-7. Instruction Sizes and Encoding

Instruction Size	Word 0 (low address)				Word 1 (next address)	Word 2 (next address)
	15	14	13	12:0	31:16	47:32
16	I_Link	1	opcode			
32	I_Link	0	1	opcode	16-bit parameters	
48	I_Link	0	0	opcode	Low 16 bits of 32-bit parameters	High 16 bits of 32-bit parameters

2.3.1 Standalone Instructions and Restrictions

Following are the restrictions:

- Discontinuity instructions cannot be included in delay slots. No hardware check. But, assembler shall flag this error.
- IDLE instruction cannot be executed in delay slots.
- IDLE instruction cannot be placed in packets covered by XC
- IDLE instruction packet cannot be parallelized.
- PRESERVE instructions can only be executed in parallel to Protected call or Protected branch or Protected return. No hardware check. But, assembler shall flag this error.
- EMUSTOP0 cannot be included in delay slots.
- XC packets consisting of more than one instruction packet are not allowed in delay slots.
- ECCSELFTEST cannot be parallelized and executed as standalone only.
- MOV Ax,RPC and LD.32 RPC, @MEM is not allowed in delay slot of CALL instruction. RPC load with Return address in delay slot 3 is not protected.
- XC/XCP instructions cannot be executed in parallel to ISRn.PROT/ENTRYn.PROT/EXITn.PROT

2.3.2 Instruction Timeout

Instruction decode many not be able to form a legal packet due to packing errors or uncorrectable error or incorrect #delay setting. In such cases, timeout logic is used by CPU enters FAULT state. Timeout counter resets whenever new instruction enters pipeline (or) when HALT is entered. Timeout counter is incremented when no instruction is in D2. If the timeout counter ever exceeds the specified timeout value then CPU is taken to FAULT state.

2.4 Stacks

The C29x CPU contains following stacks:

1. Software Stack
2. Protected Call Stack
3. Real Time Interrupt Stack

2.4.1 Software Stack

Register A15 in the available addressing registers is dedicated as the stack pointer register (SP = A15). The A15 register can access the full 32-bit address range (4GB) of the CPU (see Figure 2-2).

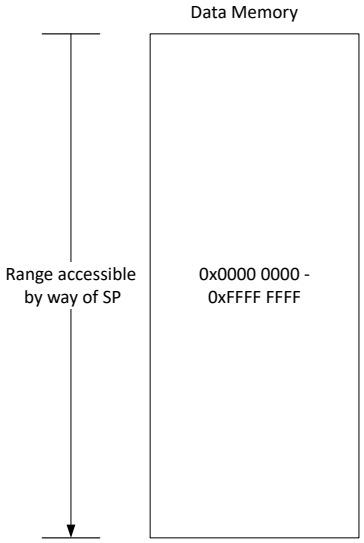

Figure 2-2. Address Reach of the Stack Pointer

The operation of the stack is as follows:

- The stack grows from low memory address to high memory address.
- The stack pointer always points to the next empty location in the stack.
- At reset, the stack pointer is initialized to 0x0000 0000.
- The C29x stack pointer always aligned to a 64-bit word boundary.

Note

The C29x stack pointer (SP = A15) must always be aligned to a 64-bit word boundary. Any misaligned stack causes CPU to enter FAULT state.

If passing parameters on the software stack is required, the assembly code structure looks as follows:

```
; Allocate Parameter Space On Stack:
ADD.U16 A15, A15, #PARAMETER_SPACE

;....pass parameters on stack…

;....pass parameters in registers…

CALL.PROT @func
; 32-bit RPC automacally pushed on stack
; A15 = A15 + 8 (stack pointer incremented by 8 bytes = 64-bits)
; There is a 32-bit hole in the stack that can be used to save
```

```
; De-allocate Parameter Space From Stack
SUB.U16 A15, A15, #PARAMETER_SPACE
```

If allocating local variables on the software stack is required, the code structure looks as follows:

```
func:
    ; Allocate Local Variable Space On Stack:
    ADD.U16 A15, A15, #PARAMETER_SPACE

    ; ... save on stack any registers used that need to be preserved across call..

    ; ... function code....

    ; ... restore from stack any registers used that need to be preserved across call..

    ; De-allocate Local Variable Space From Stack
    SUB.U16 A15, A15, #PARAMETER_SPACE


    RET.PROT
    ; 32-bit RPC automacally popped from stack
    ; A15 = A15 - 8 (stack pointer decremented by 8 bytes = 64 bits)
```

On normal function call: Return program counter (RPC, holds previous return address) register contents are pushed on software stack pointed by stack pointer (SP = A15). Then RPC register is initialized with new return address.

On normal function return: Return address is read from RPC register. Then RPC register is restored with value on software stack.

For more details regarding the RPC, see Section 2.2

2.4.2 Protected Call Stack

The Protected Call Stack is a dedicated hardware stack used to make protected function call and return. This stack is directly controlled by CPU and is inaccessible to user code. The basic protection concept of the C29x CPU is based on LINKs, STACKs, and ZONEs. The protected function call and return is the method used to make a function call by the current executing code to another function residing in a different STACK. The C29x security architecture allows definition of legal callable function labels using the instructions ENTRY1.PROT and ENTRY2.PROT. This makes sure that code from another STACK can only make function calls or branches to labels with the instruction packet “ENTRY1.PROT || ENTRY2.PROT” present. This prevents malicious code from randomly entering into code regions without permission. Nesting of protected calls is allowed up to the number of levels supported by the protected call stack. Table 2-8 shows the rules of code execution across stack.

Protected call Stack Pointer (PSP) register: The PSP register keeps track of the utilization of protected call stack and shows the current value of protected call stack pointer. This register is automatically incremented and decremented by HW on a protected call (CALL.PROT) and protected return (RET. PROT) respectively.

Warning level for Protected call Stack Pointer (WARNPSP) register: This WARNPSP is a user configurable register which allows early warning of protected stack overflow detection. When PSP register >= WARNPSP register, error signal is generated to ESM.

Maximum Protected call Stack Pointer (MAXPSP) register: The MAXPSP register is not user configurable register. When PSP register = MAXPSP register, CPU enters fault state as protected call stack is full.

Table 2-8. Rules of Code Execution Across STACKs

Program Flow Operation	Comments and CPU Action
Linear code execution within the same LINK	Allowed without any restriction
Branches, calls and returns within the same LINK	
Branches, calls and returns across different LINKs, but within the same STACK	
Protected function return (RET.PROT) where the return address is on a different STACK compared to the current STACK	
Protected function calls (CALL.PROT @label/Ax) where source and destination are on same STACK	
Protected function return (RET.PROT) where the return address is on a same STACK	
Linear code execution crossing LINK, but within the same STACK	Not allowed, CPU enters FAULT state.
Branches where source and destination are on different STACKs	
Function calls (CALL{D} @label/Ax) where source and destination are on different STACKs	
Execution of a function return instruction (RET{D} /RET{D} <addr1>) where the return address is on a different STACK compared to the current STACK	
Realtime Interrupt (RTINT) and NMI	This is handled in the hardware and does not need any consideration in the user code. The Interrupt service routine can reside in the same or a different LINK/STACK/ZONE.
Interrupts (INT)	ISR must be on the same stack. If not, CPU enters FAULT state.

2.4.3 Real Time Interrupt / NMI Stack

The Realtime Interrupt Stack (RTINT) is a dedicated hardware stack used by the Realtime Interrupt (RTINT) and the Non Maskable Interrupt (NMI). For details on the differences between the various interrupt types, see Chapter 3. When either of these interrupts are triggered, all C29x CPU working registers (Ax, Dx, Mx, RPC, DSTS, and ESTS) and return address are saved on the RTINT stack within 8 cycles and restored in 8 cycles when the RETI.RTINT instruction is executed. Nesting of RTINT is allowed up to the number of levels supported by the Realtime Interrupt Stack minus 1 level, with the NMI interrupt always having one reserved level.

Real Time Interrupt stack pointer (RTISP) register: The RTISP register keeps track of the utilization of stack and shows the current value of Real Time Interrupt stack pointer. This register is automatically incremented by hardware when Real Time Interrupt or NMI interrupt is triggered and decremented when RETI.RTINT instruction is executed.

Warning level for Real Time Interrupt stack pointer (WARNISP) register: This WARNISP is a user configurable register which allows early warning of real time interrupt stack overflow detection, when RTISP register is greater than or equal to WARNISP register value.

Maximum Real Time Interrupt Stack Pointer (MAXISP) register: The MAXISP register is not user configurable register. When ISP register equals to MAXISP register, CPU enters fault state as real time interrupt stack is full.

For more details on registers related to the Real Time Interrupt stack, see Section 3.4.3.

Note

Real Time Interrupt Stack Pointer (ISP) register, Warning level for Real Time Interrupt Stack Pointer (WARNISP) register, and Maximum Real Time Interrupt Stack Pointer (MAXISP) register are registers in PIPE (Peripheral Interrupt Priority and Expansion).

Chapter 3

Interrupts

What are Interrupts?

Interrupts are hardware- or software-driven signals that cause the CPU to suspend current program sequence and execute a subroutine. Typically, interrupts are generated by peripherals or hardware devices that need to give data to or take data from the C29x CPU (for example ADC, DAC, EPWM, and other processors). Interrupts can also signal that a particular event has taken place (for example, a timer has finished counting).

C29x CPU Interrupts

On the C29x CPU, there are four types of interrupt lines in the system. These are listed here from highest priority to lowest priority:

- RESET
- NMI (non-maskable interrupts)
- RTINT (real-time interrupts, maskable)
- INT (low-priority interrupts, maskable, disableable)

These four interrupt lines handle all interrupts and exceptions on the device. All C29x devices come with a PIPE (Peripheral Interrupt Priority and Expansion) module that provides additional prioritization and arbitration of the RTINT and INT lines. Details regarding the PIPE module are provided in the *F29H85x and F29P58x Real-Time Microcontrollers Technical Reference Manual*.

3.1 CPU Interrupts Architecture Block Diagram

Figure 3-1 shows a block diagram of the C29x CPU interrupt architecture.

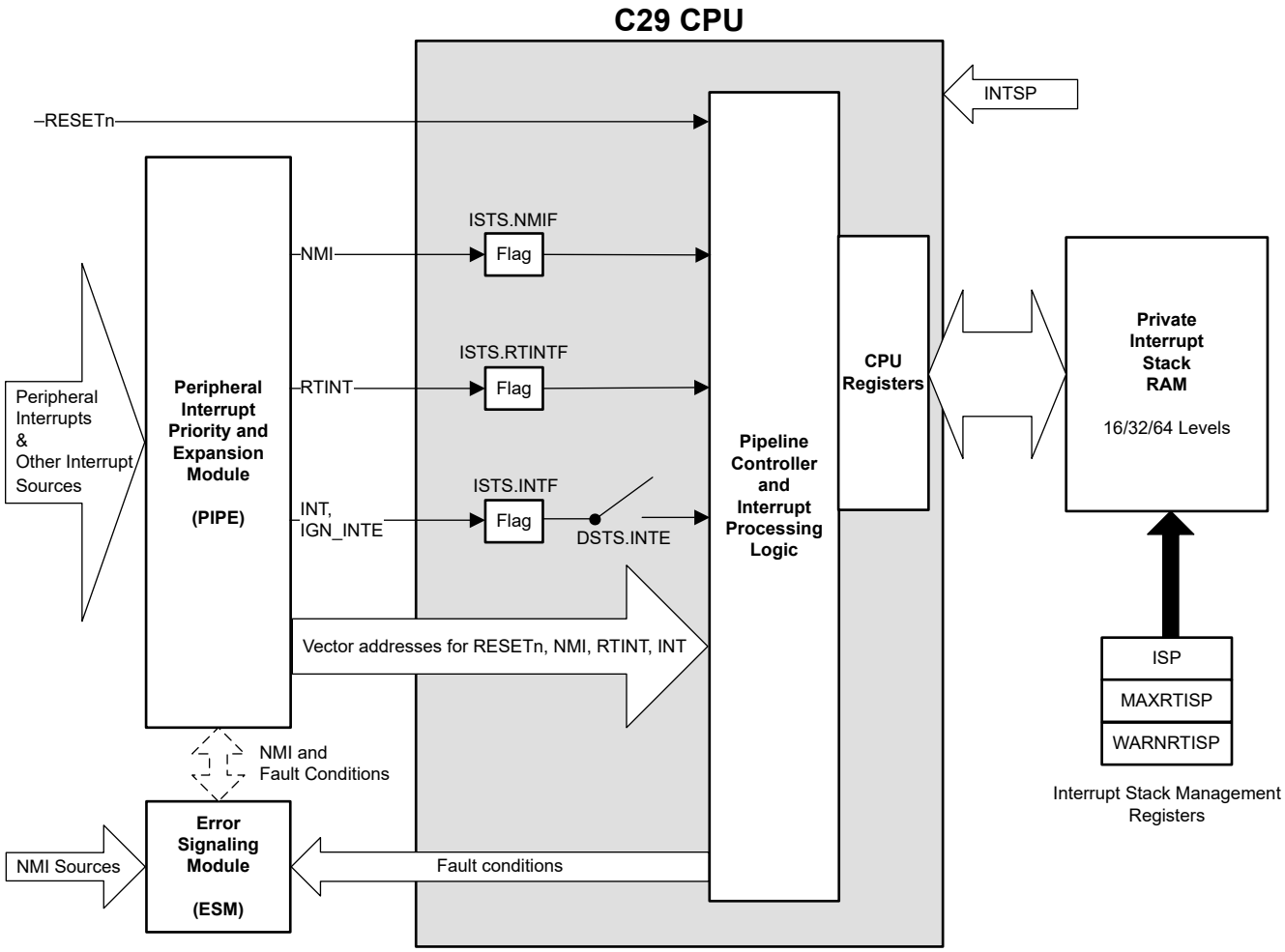

Figure 3-1. C29x CPU Interrupts Architecture Block Diagram

3.2 RESET, NMI, RTINT, and INT

This section explains the four interrupt lines available in the C29x CPU architecture.

3.2.1 RESET (CPU reset)

The CPU Reset is the highest-priority interrupt line, and occurs when the RESETn line receives an active-low signal. This causes the CPU to undergo a hardware reset internally. This cannot be aborted or nested-in.

All current and pending operations in the pipeline are aborted, and the pipeline is flushed during reset.

All CPU registers are reset to the reset value (all 0) as indicated in Table 3-1.

Table 3-1. CPU Registers Reset Values

Registers	Reset Value
A0 through **A15**	0x0000 0000
D0 through **D15**	0x0000 0000
M0 through **M31**	0x0000 0000
DSTS	0x07F8 0000
ESTS	0x0000 0000
RPC	0x0000 0000
ISTS	0x0000 0000

3.2.1.1 Required Instructions (RESET)

NMI and RTINT interrupts can potentially have the respective interrupt service routines (ISRs) residing in a different LINK/STACK. Therefore, NMI and RTINT ISRs require that the first instruction packet of every vector address contain the (ISR1.PROT || ISR2.PROT) instructions. The CPU pipeline control hardware checks for these required instructions and generates a FAULT, if these instructions are not the first instruction packet of the ISR. These required instructions are inserted automatically by the compiler, but must be configured to do so for the appropriate vectors within a separate security settings file. See Section 3.6 for more details.

ISR1.PROT also initializes the stack pointer (A15) to the appropriate STACK by performing the following operation: A15 = SECSPn (where n is the current STACK indicated by ISTS.CURRSP).

For more details on the security implications of the LINK/STACK/ZONE and memory space for CPU interrupts, see Section 3.6.

3.2.2 NMI (Non-Maskable Interrupt)

The NMI (non-maskable interrupt) is the second highest-priority interrupt line, and receives system exception interrupts.

This NMI input line is used for any device-level critical condition and various faults either inside or outside the CPU that needs immediate attention.

3.2.2.1 Blocking and Masking (NMI)

NMIs cannot be masked or blocked in the CPU. There is no global enable/disable bit for the NMI line in the CPU. Because of this, any interrupts that are received on the NMI line are directly passed to the CPU for prioritization. Priority is then decided amongst the interrupt types (NMI, RTINT, and INT lines). NMI always has highest priority and asserts within any RTINT or INT currently executing. ATOMIC instructions in RTINT or INT ISRs cannot block or prevent NMI from asserting. ATOMIC instructions have no effect on NMI.

3.2.2.2 Signal Propagation (NMI)

The NMIn input to the CPU is typically generated from an Error Signaling Module (ESM) unit located outside the CPU. Such a unit allows for aggregation and prioritization of system faults. Even fault conditions that occur internal to the CPU propagates the fault information to a common device level aggregator unit, which then generates an NMI back to the CPU.

The NMIn input latches inside the CPU, and is handled with higher priority than all other interrupt types (except Reset events).

3.2.2.3 Stack (NMI)

This interrupt line uses the protected Real Time Interrupt Stack for context save and restore. This SSU-protected (Safety and Security Unit) stack has protection features to prevent stack overflow during nesting, when nesting is requested by the PIPE module. The WARNRTISP and MAXRTISP CPU registers serve this purpose in the C29x CPU system.

This protection limits nesting of RTINT up to the number of levels supported by the RTINT Stack minus one level (which is always reserved for NMI interrupt).

For security, the SSU protection of the RTINT Stack are designed so that the contents of the stack are not visible. Registers are also zeroed to prevent visibility into what was happening before the interrupt was serviced.

See Section 2.4 for details on stack overflow protection using the WARNRTISP and MAXRTISP registers.

3.2.2.4 Required Instructions (NMI)

NMI and RTINT interrupts can potentially have the respective interrupt service routines (ISRs) residing in a different LINK/STACK. Therefore, NMI and RTINT ISRs require that the first instruction packet of every vector address contain the (ISR1.PROT || ISR2.PROT) instructions. The CPU pipeline control hardware checks for these required instructions and generates a FAULT, if these instructions are not the first instruction packet of the ISR. These required instructions are inserted automatically by the compiler, but must be configured to do so for the appropriate vectors within a separate security settings file. See Section 3.6 for more details.

ISR1.PROT also initializes the stack pointer (A15) to the appropriate STACK by performing the following operation: A15 = SECSPn (where n is the current STACK indicated by ISTS.CURRSP).

For more details on the security implications of the LINK/STACK/ZONE and memory space for CPU interrupts, see Section 3.6.

3.2.3 RTINT (Real-Time Interrupt)

The RTINT (real-time interrupt) is the third highest-priority interrupt line, and receives signals driven by an interrupt expansion and aggregation unit (the PIPE module for most C29x CPU systems).

3.2.3.1 Blocking and Masking (RTINT)

RTINT sources are able to be masked, but the actual RTINT line connected to the CPU can never be blocked/disabled by user code. There is no global enable/disable bit for the RTINT line in the CPU. Because of this, any interrupts that are received on the RTINT line are directly passed to the CPU for prioritization. Priority is then decided among any interrupts on the NMI or INT lines. The RTINT signal line can only be stopped from nesting within INTs by using the ATOMIC instruction within the INT ISR, and only for a finite number of instruction packets. However, interrupts ISRs can be prioritized/blocked before the interrupts reach the RTINT line using the external PIPE module.

3.2.3.2 Signal Propagation (RTINT)

The PIPE module provides external interrupt aggregation and arbitration for the RTINT and INT lines. This allows for many signals to be categorized as real-time interrupts (RTINT) or low-priority interrupts (INT), and then prioritized before passing to the CPUs RTINT or INT interrupt line.

The PIPE effectively multiplexes the single RTINT CPU interrupt line to be able to receive from multiple incoming RTINT interrupts in the appropriate order.

The module allows for enabling and disabling of RTINT signals before the signals reach the RTINT line of the CPU. The module also allows nesting capability amongst other interrupts categorized as RTINT. See the *F29H85x and F29P58x Real-Time Microcontrollers Technical Reference Manual* for details on the PIPE module features.

3.2.3.3 Stack (RTINT)

This interrupt line uses the protected Real Time Interrupt Stack for context save and restore. This SSU-protected (Safety and Security Unit) stack has protection features to prevent stack overflow during nesting, when nesting is requested by the PIPE module. The WARNRTISP and MAXRTISP CPU registers serve this purpose in the C29x CPU system.

This protection limits nesting of RTINT up to the number of levels supported by the RTINT Stack minus one level (which is always reserved for NMI interrupt).

For security, the SSU protection of the RTINT Stack are designed so that the contents of the stack are not visible. Registers are also zeroed to prevent visibility into what was happening before the interrupt was serviced.

See Section 2.4 for details on stack overflow protection using the WARNRTISP and MAXRTISP registers.

3.2.3.4 Required Instructions (RTINT)

NMI and RTINT interrupts can potentially have the respective interrupt service routines (ISRs) residing in a different LINK/STACK. Therefore, NMI and RTINT ISRs require that the first instruction packet of every vector address contain the (ISR1.PROT || ISR2.PROT) instructions. The CPU pipeline control hardware checks for these required instructions and generates a FAULT, if these instructions are not the first instruction packet of the ISR. These required instructions are inserted automatically by the compiler, but must be configured to do so for the appropriate vectors within a separate security settings file. See Section 3.6 for more details.

ISR1.PROT also initializes the stack pointer (A15) to the appropriate STACK by performing the following operation: A15 = SECSPn (where n is the current STACK indicated by ISTS.CURRSP).

For more details on the security implications of the LINK/STACK/ZONE and memory space for CPU interrupts, see Section 3.6.

3.2.4 INT (Low-Priority Interrupt)

The INT (low-priority interrupt) is the lowest-priority interrupt line, and receives signals driven by an interrupt expansion and aggregation unit (the PIPE module for most C29x CPU systems). This interrupt line is typically used for lower priority operations and task schedulers.

3.2.4.1 Blocking and Masking (INT)

INT sources are able to be masked, and the INT line can also be blocked/disabled by user code using the DSTS.INTE enable bit. If DSTS.INTE is enabled, then any interrupts received on the INT line are directly passed to the CPU for prioritization. Priority is then decided among the interrupts on the NMI or RTINT lines. To prevent an RTINT interrupt from nesting within a INT interrupt, the ATOMIC instruction can be used for a finite number of instruction packets.

On entering a INT ISR, further INTs are automatically disabled using the DSTS.INTE bit. To allow nesting, enable interrupts using the ENINT instruction. There also exists a DISINT instruction for disabling the INT line again.

The C29x CPU also provides a special INT called as Supervisor Interrupt. Supervisor Interrupt is essentially an INT which can override the DSTS.INTE setting. For example, Supervisor Interrupt can be a certain task monitor interrupt which requires the interrupt to not get blocked by erroneous setting of the DSTS.INTE.

3.2.4.2 Signal Propagation (INT)

The PIPE module provides external interrupt aggregation and arbitration for the RTINT and INT lines. This allows for many signals to be categorized as real-time interrupts (RTINT) or low-priority interrupts (INT), and then prioritized before passing to the CPU's RTINT or INT interrupt line.

The PIPE effectively multiplexes the single INT CPU interrupt line to be able to receive from multiple incoming INT interrupts in the appropriate order.

The module allows for enabling and disabling of INT signals before the signals reach the INT line of the CPU. The module also allows nesting capability amongst other interrupts categorized as INT or RTINT. See the *F29H85x and F29P58x Real-Time Microcontrollers Technical Reference Manual* for details on the PIPE module features.

3.2.4.3 Stack (INT)

Unlike the NMI or RTINT interrupt lines, the INT line uses the standard software stack for context save and restore. Only one of the multiple available CPU stacks can be used for INT. This is configured by the INTSP register in the external PIPE module. If an INT vector points to the wrong LINK which is associated with a different STACK (security-assigned stack), then an NMI fault is generated.

If the current stack pointer is not pointing to the INTSP, then any pending INT remains pending until the stack pointer points to the selected INTSP stack.

3.3 Conditions Blocking Interrupts

There are certain CPU pipeline conditions that cause an uninteruptible boundary for the CPU. These conditions prevent entry of interrupts until the conditions are over, effectively blocking interrupts during the hold time. Table 3-2 explains these situations:

Table 3-2. Conditions That Block Interrupts

<table>
<tr><th>Condition Description</th><th>INT Blocked</th><th>RTINT Blocked</th><th>NMI Blocked</th></tr>
<tr><td>Conditional instructions in packet not all completed</td><td colspan="3" rowspan="10">Yes</td></tr>
<tr><td>Discontinuity instruction delay slot not completed</td></tr>
<tr><td>Multicycle instructions like branch, call, return not completed</td></tr>
<tr><td>For CALL.PROT instruction: first instruction at the call destination not executed</td></tr>
<tr><td>For RET.PROT instruction: first instruction of the return address not executed</td></tr>
<tr><td>The first instruction of the previous asserted interrupt has entered the D2 stage</td></tr>
<tr><td>CPU "pipeline ready" not asserted</td></tr>
<tr><td>CPU pipeline stalled due to memory RD/WR access</td></tr>
<tr><td>CPU pipeline stalled due to no instruction in the instruction buffer</td></tr>
<tr><td>Instruction packet stalled in D2 phase of pipeline due to pipeline hazard, but the packet is not ready to move to R1 phase of pipeline.</td></tr>
<tr><td>LP Interrupt is disabled in DSTS.INTE</td><td>Yes</td><td colspan="2">NO</td></tr>
<tr><td>ATOMIC instruction counter not completed</td><td colspan="2">Yes</td><td>NO</td></tr>
</table>

3.3.1 ATOMIC Counter

The C29x CPU supports an instruction "**ATOMIC.REG #u8**" that loads an internal counter (ISTS.ATOMIC COUNTER) with an 8-bit value. This counter decrements once for each instruction packet executed. As long as this atomic counter is not zero, an interrupt (RTINT or INT) cannot enter the CPU pipeline. Hence, this instruction allows the user code to block interrupts for up to 256 instruction packets.

Restrictions on the use of ATOMIC instruction:

- The ATOMIC instruction cannot be in the delay slot of any discontinuity instruction or executed in parallel to a branch instruction.
- The ATOMIC instruction cannot be in parallel to any discontinuity instruction.
- Executing ATOMIC instructions back to back cannot be used to block interrupts beyond the maximum count.
- Executing an ATOMIC instruction when the ATOMIC count is not zero resets the ATOMIC counter.
- Protected calls and returns reset the ATOMIC counter.
- The ATOMIC instruction or counter CANNOT block NMI. Anytime the ISTS.NMIF flag is set (indicating that a NMI event has been registered), the NMI is taken and the ATOMIC counter is reset.

3.4 CPU Interrupt Control Registers

This section covers the three types of CPU registers that control interrupt-related functionality.

3.4.1 Interrupt Status Register (ISTS)

The Interrupt Status Register (ISTS) contains status information of various interrupt flags, stack pointers, current link, and various counters.

-	CURRLINK				-			INTSP				CURRSP			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

ATOMIC {counter}								-	-	-	-	-	NMIF	RTINTF	INTF
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Current LINK (CURRLINK): All resources including memory, peripherals, stacks are associated with LINK ID. Links divide the boundaries of context in which the CPU is operating. Hence the code source address of the instruction packet in the D2 pipeline stage is resolved to the corresponding code's LINK. This information is critical to validate, update and access permissions in the interrupt vector table. The information is also critical to configuration settings associated with each interrupt. Therefore, the CURRLINK register provides the current LINK.

Stack pointers (CURRSP, INTSP): The C29x CPU, with embedded virtualization, has multiple stacks. The user can assign a particular stack for INT, but RTINT and NMI use the RTINT Stack. The current stack pointer, which points to the current STACK in use by the CPU, is represented by the CURRSP field. The STACK that is chosen to be used by low-priority interrupts (INTs) is represented by the INTSP field. INTs do not enter the pipeline until the INTSP matches the CURRSP.

ATOMIC counter (ATOMIC): The CPU allows up-to 256 instruction packets executed at one stretch without being interrupted by RTINT or INT. The number of remaining instruction packets of ATOMIC execution is reflected in the ATOMIC counter. Interrupts are not picked up for processing by the CPU, if the ATOMIC counter is ticking. NMI is not affected by the ATOMIC counter, and operation is stopped and the counter is reset if an NMI is received. See Section 3.3.1 for more details on the ATOMIC counter.

Interrupt flags (INTF, RTINTF, NMIF): Independent interrupt flags are registered including INTF, RTINTF, and NMIF. These flags are set whenever a corresponding interrupt is asserted to the CPU and cleared upon exiting the corresponding ISR. If there are multiple nested interrupts that are taken by CPU, then all corresponding flags are set and those are cleared only upon servicing all interrupts in the respective category.

3.4.2 Decode Phase Status Register (DSTS)

The Decode Phase Status Register (DSTS) contains information regarding the interrupt and link status of the CPU. The following table highlights fields related to interrupt operation. This information is used by software for building predictable priority and security behavior.

-	RLINK				ISR_PRIORITY								INTS		INTE
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

TA3	TA2	TA1	TA0	-	CLINK				DBGM	-	-	A.ZV	A.C	A.N	A.Z
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Return Link (RLINK): This represents the LINK of the origin from where the protected return was executed.

ISR Priority (ISR_PRIORITY): If CPU is servicing an interrupt and INTS is either INT or RTINT, then the priority level of the interrupt is reflected in this field. This is an 8 bit register field.

Interrupt status (INTS): This field tracks the status of CPU execution, specifying whether the execution is in the main loop, in an NMI ISR, in an RTINT ISR, or in an INT ISR. This field is also used by the external PIPE module to track the present phase of CPU to decide when the next ready RTINT or INT interrupt can be forwarded to CPU. See Table 3-3 for details on the status values of the INTS field.

Table 3-3. INTS - Interrupt Status Values

INTS[1]	INTS[0]	CPU State	Comment
0	0	Main code	Not in any task, interrupt or exception
0	1	INT Handler	In a normal interrupt service routine
1	0	RTINT Handler	In a real-time interrupt service routine
1	1	NMI Handler	In a NMI handler routine

INT Enable (INTE): The INT enable bit reflects whether an INT interrupt can be accepted by the CPU or not. This bit needs to be 1 to allow the next higher priority INT to be accepted in the CPU (which allows nesting of INTs). Upon accepting INT, this bit gets automatically set to 0 and so ISR code needs to explicitly set this back to 1 to enable INT interrupt nesting.

Caller Link (CLINK): The CLINK field represents the LINK of the origin which made a call to execute the function. This includes execution calls from ISRs.

To enhance security, the CLINK field can be checked within a given ISR (or function) to determine if the CLINK field matches a predefined Link. The ISR (or function) can then exit automatically, if the Link does not match.

3.4.3 Interrupt-Related Stack Registers

The C29x CPU has three types of stacks, with related pointers for each. These are outlined in Table 3-4 as a high-level overview. The section that follows provides details on the pointers of the interrupt-related High-Priority Interrupt Stack.

Table 3-4. C29x CPU Stack Types

Stack Type	Related Pointers
Normal Software Stack	SECSPx, where x = 0 to 15
Protected Call Stack	PSP, WARNPSP, MAXPSP
RTINT Stack	ISP, WARNRTISP, MAXRTISP

RTISP (RTINT Stack Pointer): This points to the stack that is used by NMI and RTINT interrupt lines. This stack is SSU-protected. See the Stack subsection of Section 3.2.3 for more details on the RTINT Stack.

WARNRTISP level: This level is pre-programmed by secure software code. If the ISP from CPU meets this level then the external PIPE module stops sending RTINTs to the CPU. This is to slow down stack progression or excessive nesting that can lead to a stack overflow. WARNRTISP level can be updated by the user meeting the required software security checks. Modification of WARNRTISP level is typically done after reset.

MAXRTISP level: This is a fixed-level equal to the total of number of nestings allowed by the High Priority Interrupt Stack minus one. This is to allow one reserved interrupt stack space for an NMI to trigger, to prevent stack overflow. The PIPE raises a fault when this level is reached, which in turn generates an NMI to resolve this critical condition.

3.5 Interrupt Nesting

Nesting is supported at the hardware level in the C29x CPU. At the CPU interrupt level, nesting is possible amongst the three non-reset interrupt lines (NMIn, RTINTn, INTn). Interrupt lines can nest inside the ISR of lower priority interrupt lines. So an NMI can nest within RTINT or INT. RTINT can nest within INT. INT cannot nest in other interrupt lines. However, additional nesting within interrupt types RTINT and INT is possible using the PIPE module.

A detailed look at the nesting available on the C29x CPU is explained below (along with the expanded abilities afforded by the PIPE module).

NMI: No interrupt (including other NMIs) can nest within an NMI that is currently running. Anytime the ISTS.NMIF flag is set (indicating that a NMI event has been registered), the NMI is taken and the ATOMIC counter is reset.

RTINT: NMIs always nest within RTINT. This nesting cannot be stopped with the ATOMIC instruction. Using the PIPE module, higher priority RTINTs can nest within lower priority RTINTs. The ATOMIC instruction can delay entry of a nested RTINT until the ATOMIC counter expires.

INT: NMIs always nest within INT. This nesting cannot be stopped with the ATOMIC instruction. RTINTs always nest within INT, but the ATOMIC instruction can delay entry of a nested RTINT until the ATOMIC counter expires. Using the PIPE module, higher priority INTs can nest within lower priority INTs. The ATOMIC instruction can delay entry of a nested INT (or RTINT) until the ATOMIC counter expires.

3.5.1 Interrupt Nesting Example Diagram

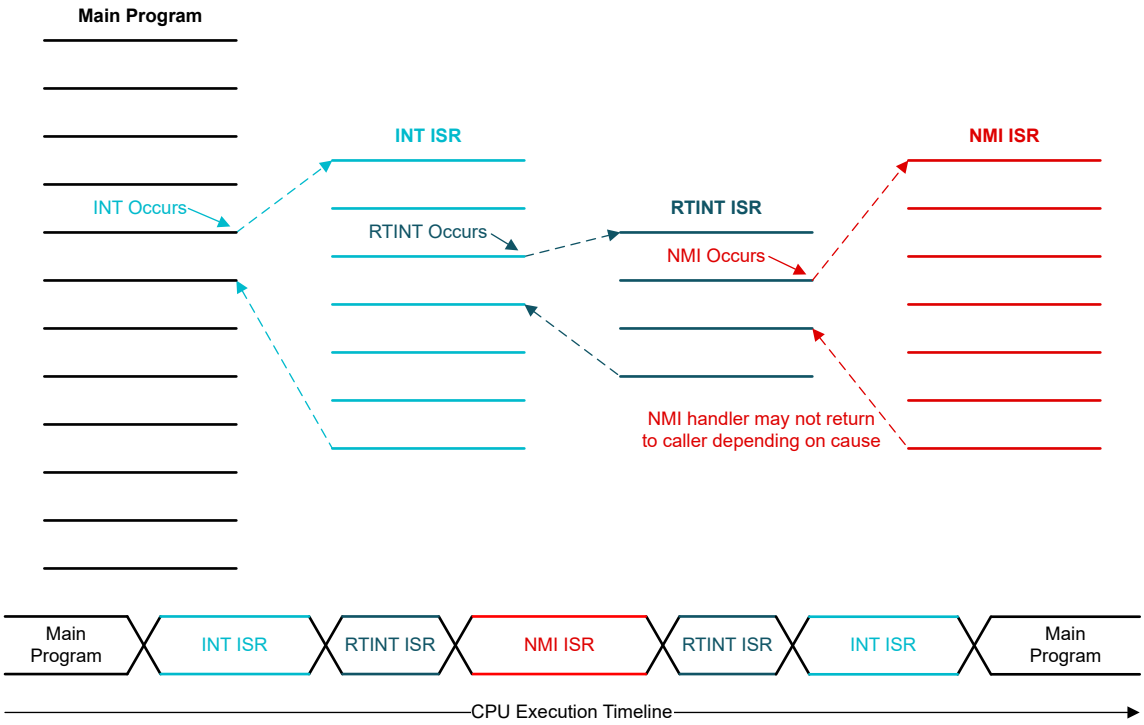

Figure 3-2. Interrupt Nesting Example Diagram

3.6 Security

The C29x CPU security features extend into the interrupts domain to make sure of software security. This section covers security features related to the CPU interrupts architecture. See Chapter 5 for further details on the C29x CPU security architecture and SSU (Safety and Security Unit) features.

3.6.1 Overview

Central to the security architecture of C29x CPU is the concept of LINKS, STACKS, and ZONES. For a detailed overview of how each of these work, see Chapter 5. The following sections provide details on how the CPU interrupts utilize each of these security features.

3.6.2 LINK

A LINK is a unique collection of ownerships and access permissions tagged together by ID which is used for resource allocation and sharing. Link ID binds CPU resources like stack, memory regions, peripheral instance accesses, interrupt source and vectors, DMA channels and permissions, and more. This prevents resources from being accessed from a hacker (or erroneous code) running from another Link. Of particular interest to this chapter is the access provided by a LINK to particular interrupt sources and vectors.

The following provides the types of links utilized for interrupt operation.

1. Owner link: every interrupt line and associated interrupt vector has an owner link.
   - This link has related resources of the source events like peripheral, GPIO or error mechanism.
   - Owner link has access to memory, DMA or other resources needed to service the specific event.
   - Owner link has access to control and status register to read flag, enable/disable an interrupt line, clear the flag or force the flag high. As well has permissions to read and clear the overflow flag.
   - The Current Link ID provided by the CPU is compared against owner link for Interrupt operational registers.
2. Boot link: boot link handles device boot and initialization including that of interrupts.
   - User Boot Link: User boot links do not have special privileges to access PIPE or interrupt registers.
3. Secure root Link: This is the root of trust for security code of the device and shall have access to PIPE configuration registers and vector table. Configuration registers hold information like the owner link, caller link, priority, vector address of the interrupt line.
4. Caller Link: Caller links are used to share common code libraries across multiple links.
   - The owner link of an interrupt line (like RTINTn) can call a common code function (from another link) in an ISR. In such a case, it is checked whether the caller link is the owner link.
   - Configuration registers like priority can only be updated by the “secure root link”. In this case, the owner link can be the caller link of a “Secure Root Link” function.

3.6.3 STACK

The C29x CPU uses multiple STACKs to make sure integrity and separation between different processes. Every LINK shall have an associated STACK mapped at device initialization. Multiple LINKs can share a STACK but multiple STACKs do not share a LINK. The following lists the stacks related to PIPE and interrupts, and the corresponding safety features:

- INT Stack: The user can choose and allocate a single stack for all INTs. This stack is one of the normal software stacks available on the device. The INT asserted to CPU remains in the pending state until the CPU returns to this stack. Normally this is expected to be the stack of main process.
- RTINT Stack: This is dedicated stack is used for context save and restore of RTINT and NMI. This stack is not accessible or visible to any user code for security, and incorporates ECC (error correction code) along with registers. Registers are zeroed to prevent visibility into what was happening before the interrupt was serviced. Features available on the High Priority Interrupt Stack include:
  - *WARNRTISP level:* : This level is pre-programmed by secure software code. If the ISP from CPU meets this level then the external PIPE module stops sending RTINTs to the CPU. This is to slow down stack progression or excessive nesting that can lead to a stack overflow. WARNRTISP level can be updated by the user meeting the required software security checks. Modification of WARNRTISP level is typically done after reset.
  - *MAXRTISP level:* : This is a fixed-level equal to the total of number of nestings allowed by the High Priority Interrupt Stack minus one. This is to allow one reserved interrupt stack space for an NMI to trigger, to prevent stack overflow. The PIPE raises a fault when this level is reached, which in turn generates an NMI to resolve this critical condition.

3.6.4 ZONE

Multiple STACKs and corresponding LINKs can be combined to make a ZONE. The items that make up this ZONE can be reflected through the configuration of LINKs and STACKs. Zone association is used for debug permissions.

Chapter 4

Addressing Modes

This chapter describes the addressing modes of the C29x CPU and provides examples.

4.1 Addressing Modes Overview

The C29x CPU supports several addressing modes to provide faster execution and smaller code size.

4.1.1 Documentation and Implementation

Throughout the documentation, instructions that utilize addressing modes are written in a manner similar to the following: "LD.32 Dx,ADDR1".

In actual assembly code implementation, the **field** "ADDR1" is replaced with an actual **addressing mode** with the parameters substituted. For example: "*(Ax+#8)".

These addressing modes are categorized into different **types**. For example, *(Ax+#8) is of the type "Pointer Addressing With #Immediate Offset".

The following figures show a visual explanation of how the fields, addressing modes, and types work together in the documentation and implementation. Both images use the same **field** (ADDR1), but have different **addressing modes** and addressing mode **types**:

In Figure 4-1, ADDR1 is replaced with the specific addressing mode ***(A15++#u8imm)**, which is one of several addressing modes available in the **Stack Addressing** type of addressing mode.

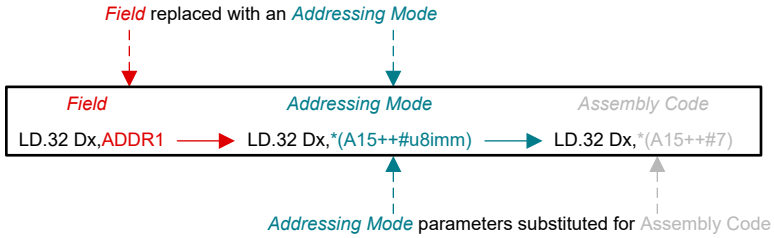

Figure 4-1. ADDR1 Field Replaced with a Stack Addressing Type

In Figure 4-2, ADDR1 is replaced with the specific addressing mode ***(Ax+#u10imm)**, which is one of several addressing modes available in the **Pointer Addressing With #Immediate Offset** type of addressing mode.

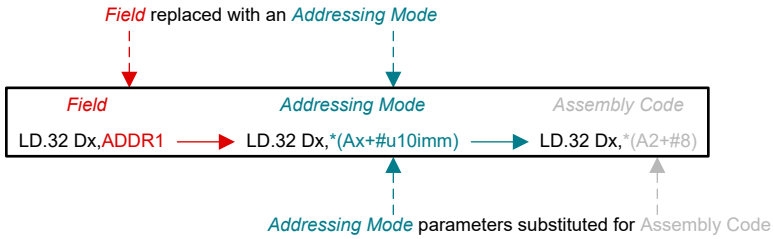

Figure 4-2. ADDR1 Field Replaced with a Pointer Addressing With #Immediate Offset Type

4.1.2 List of Addressing Mode Types

The following lists the types of addressing modes available natively in the device. For more details on each of these addressing modes, see Section 4.4.

1. **Direct Addressing:** direct read or write access to any location in the 32-bit memory space with the immediate address provided in the instruction.
2. **Pointer Addressing with #Immediate Offset:** indirect read or write access to any location in the 32-bit memory space with the pointer address from one of the addressing registers, A0 to A14, and an optional immediate offset provided in the instruction.
3. **Pointer Addressing with Pointer Offset:** indirect read or write access to any location in the 32-bit memory space with the pointer address (base address register) from one of the addressing registers, A0 to A14, and an offset provided by an additional pointer (index register) in the instruction.
4. **Pointer Addressing with #Immediate Increment/Decrement:** indirect read or write access to any location in the 32-bit memory space with the pointer address from one of the addressing registers, A0 to A14. An immediate pre or post increment or decrement of the register is applied.
5. **Pointer Addressing with Pointer Increment/Decrement:** indirect read or write access to any location in the 32-bit memory space with the pointer address from one of the addressing registers, A0 to A14, and a pre or post increment or decrement of the register is applied using the value located in an additional pointer register.
6. **Stack Addressing:** indirect read or write access to any location in the stack space with the address provided in addressing register A15, which is the dedicated Stack Pointer (SP).

The types of addressing modes can be implemented using different combinations of offsets and shifts. All available addressing modes are provided as rows in Section 4.1.3.

Note

Addressing register **A15** is the dedicated Stack Pointer (SP). Any references to the "Stack Pointer" or "SP" in this document are referring to addressing register A15.

4.1.2.1 Additional Types of Addressing

There two are additional types of addressing that can be implemented using instructions dedicated for the task. These instructions are used to take an address, and modify/compare the address while accessing the address. This allows for operations that normally require multiple instructions to occur in a single instruction. This eliminates the need for a dedicated addressing mode for such functionality. There is no performance impact when comparing native addressing-mode support to these instructions.

1. **Circular Addressing:** Circular pointers typically used for implementing finite impulse response (FIR), least mean squares (LMS), or convolution filters.
2. **Bit-Reversed Addressing:** Reverse addressing typically used to re-order data for Fast Fourier Transform (FFT) and similar algorithms.

4.1.3 Addressing Modes Summarized

Table 4-1 summarizes all supported addressing modes and the various forms.

Table 4-1. Available Addressing Modes

Opcode Field	Mnemonic	Shorthand	Address Generation
Direct Addressing			
DIRM	*(0:#u32imm)	@u32imm	addr = #u32imm
Pointer Addressing with #Immediate Offset: (Ax = A0 to A14 [2], Az = A4 to A7)			
DIRM	*(Ax+#u28imm)	*Ax[#u28imm]	addr = Ax + #u28imm (#u28imm = 0 to 256MB range)
ADDR1	*(Ax+#u10imm)	*Ax[#u10imm]	addr = Ax + #u10imm (#u10imm = 0 to 1KB range)
ADDR1	*(Ax+#u10imm<<2)	*Ax[#u10imm]	addr = Ax + #u10imm<<2 (#u10imm << 2 = 0 to 4KB range, 4B steps)
ADDR3	*(Ax+#u8imm<<2)	*Ax[#u8imm]	addr = Ax + #u8imm<<2 (#u8imm << 2 = 0 to 1KB range, 4B steps)
ADDR2	*Az	*Az	addr = Az
Pointer Addressing with Pointer Offset: (Ax = A0 to A14 [2], Aj = A0 to A14, Ak = A0 to A3, Az = A4 to A7)			
ADDR1	*(Ax+Ak<<#u2imm)	*Ax[Ak]	addr = Ax + Ak << #u2imm (#u2imm = 0, 1, 2, 3)
ADDR1	*(Aj=(Ax+Ak<<#u2imm))	*Aj=Ax[Ak]	addr = Ax + Ak << #u2imm, Aj = addr (#u2imm = 0, 1, 2, 3)
ADDR2	*(Az+A0<<#scale)	*Az[A0]	addr = Az + A0 << (0/1/2/3) [1]
ADDR2	*(Az+A1<<#scale)	*Az[A1]	addr = Az + A1 << (0/1/2/3) [1]
Pointer Addressing with #Immediate Increment/Decrement: (Ax = A0 to A14 [2], Az = A4 to A7)			
ADDR1	*(Ax++#u8imm)	*Ax++[#u8imm]	addr = Ax, Ax = Ax + #u8imm (#u8imm = 0 to 255 range)
ADDR1	*(Ax--#n8imm)	*Ax--[#n8imm]	addr = Ax, Ax = Ax - #n8imm (#n8imm = 1 to 256 range)
ADDR1	*(Ax-=#n8imm)	*Ax-=[#n8imm]	Ax = Ax - #n8imm, addr = Ax (#n8imm = 1 to 256 range)
ADDR2	*(Az++#size)	*Az++	addr = Az, Az = Az + (1/2/4/8) (#size = 1,2,4,8) [1]
ADDR2	*(Az--#size)	*Az--	addr = Az, Az = Az – (1/2/4/8) (#size = 1,2,4,8) [1]
ADDR2	*(Az-=#size)	*--Az	Az = Az – (1/2/4/8), addr = Az (#size = 1,2,4,8) [1]
Pointer Addressing with Pointer Increment/Decrement: (Ax = A0 to A14 [2], Ak = A0 to A3, Az = A4 to A7)			
ADDR1	*(Ax+#u7imm)++Ak	*Ax[#u7imm]++Ak	addr = Ax + #u7imm, Ax = Ax + Ak (#u7imm = 0 to 128)
ADDR2	*(Az++A0)	*Az++A0	addr = Az, Az = Az + A0
ADDR2	*(Az++A1)	*Az++A1	addr = Az, Az = Az + A1
Stack Addressing: (A15 = SP)			
ADDR1	*(A15-#n13imm)	*A15-[#n13imm]	addr = A15 - #n13imm (#n13imm = 1 to 8192)
ADDR1	*(A15++#u8imm)	*A15++[#u8imm]	addr = A15, A15 = A15 + #u8imm (#u8imm = 0 to 255)
ADDR1	*(A15-=#n8imm)	*A15-=[#n8imm]	A15 = A15 - #n8imm, addr = A15 (#n8imm = 1 to 256)

(1) The ADDR2 opcode field modes do not specify the increment step size ("#size") or scale amount ("#scale"). This is automatically performed by the CPU hardware based on the word size being accessed by the instruction. See Section 4.2 for more details.
(2) The Ax[0-14] addressing field can support the A15 register, however this is the stack pointer (SP) register and for some of the addressing modes, the operation is not valid for the SP and hence the addressing mode can not be used

4.2 Addressing Mode Fields

This section explains how the various addressing modes are represented in each instruction.

Explanation of Terminology

For instructions that use addressing modes, this document uses four different "**fields**":

- ADDR1
- ADDR2
- ADDR3
- DIRM

These four fields are placeholders for actual addressing modes. The four fields are separated based on the number of bits required to encode them in the instruction (for example ADDR1 uses 16 bits and ADDR2 uses 5 bits).

In actual assembly code, the user or compiler must substitute fields for the desired addressing mode. For example, in the documentation, the "field" name ADDR1 is used. But in assembly code, ADDR1 can be replaced with a real addressing mode.

"LD.32 Ax,**ADDR1**" (field)

becomes

"LD.32 A8,***(A4+#0x4)**" (addressing mode).

This is only one possible way to convert the 16 bits available in the ADDR1 field into an actual addressing mode ("Pointer Addressing With #Immediate Offset" type of addressing mode).

The following subsections explain each of the four fields, the available addressing modes, and the encodings of these addressing modes. There is also a section explaining some additional fields used with addressing modes.

4.2.1 ADDR1 Field

This is a 16-bit field for indirect encoding of addresses that can be used in all "Pointer Addressing" and "Stack Addressing" modes.

Table 4-2 shows the various ways the 16 bits can be used to encode the address.

Table 4-2. ADDR1 Field Encodings

ADDR1 Field: (Ax = A0 to A14, Aj = A0 to A14, Ak = A0 to A3)																			
Mnemonic [2]	**Shorthand**	**Address Generation**	**47**	**46**	**45**	**43**	**42**	**41**	**40**	**39**	**38**	**37**	**36**	**35**	**34**	**33**	**32**	**31**
Mnemonic	**Shorthand**	**Address Generation**	**31**	**30**	**29**	**28**	**27**	**26**	**25**	**24**	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**
*(Ax+#u10imm)	*Ax[#u10imm]	addr = Ax + #u10imm (#u10imm = 0 to 1KB range)	0	0	#u10imm										Ax[0-14] [1]			
*(Ax+#u10imm<<2)	*Ax[#u10imm]	addr = Ax + #u10imm<<2 (#u10imm << 2 = 0 to 4KB range, 4B steps)	0	1	#u10imm										Ax[0-14] [1]			
*(Ax+#u7imm)++Ak	*Ax[#u7imm]++Ak	addr = Ax + #u7imm, Ax = Ax + Ak (#u7imm = 0 to 128)	1	0	0	#u7imm							Ak[0-3]		Ax[0-14] [1]			
*(A15-#n13imm)	*A15-[#n13imm]	addr = A15 - #n13imm (#n13imm = 1 to 8192)	1	0	1	#n13imm												
*(Ax++#u8imm)	*Ax++[#u8imm]	addr = Ax, Ax = Ax + #u8imm (#u8imm = 0 to 255 range)	1	1	0	0	#u8imm								Ax[0-14] [1]			
*(Ax--#n8imm)	*Ax--[#n8imm]	addr = Ax, Ax = Ax - #n8imm (#n8imm = 1 to 256 range)	1	1	0	1	#n8imm								Ax[0-14] [1]			
*(Ax-=#n8imm)	*Ax-=[#n8imm]	Ax = Ax - #n8imm, addr = Ax (#n8imm = 1 to 256 range)	1	1	1	0	#n8imm								Ax[0-14] [1]			
*(Ax+Ak<<#u2imm)	*Ax[Ak]	addr = Ax + Ak << #u2imm (#u2imm = 0, 1, 2, 3)	1	1	1	1	#u2imm		1	1	1	1	Ak[0-3]		Ax[0-14] [1]			
*(Aj=(Ax+Ak<<#u2imm))	*Aj=Ax[Ak]	addr = Ax + Ak << #u2imm, Aj = addr (#u2imm = 0, 1, 2, 3)	1	1	1	1	#u2imm		Aj[0-14]				Ak[0-3]		Ax[0-14] [1]			

(1) The Ax[0-14] addressing field can support the A15 register, however this is the stack pointer (SP) register and for some of the addressing modes, the operation is not valid for the SP and hence the addressing mode can not be used.
(2) Data Move operations have two ADDR1 fields, all other operations only have one ADDR1 field. Except for Data Move operations, the ADDR1 field is located in bits [31:16] of the instruction opcode.

The following are the instructions that can use the ADDR1 field:

ADD.32, ADD.S16, ADD.S8, AND.16, AND.8, AND.U16, AND.U8, ANDOR.B0, ANDOR.W0, LD.32, LD.64, LD.B0, LD.B1, LD.B2, LD.B3, LD.S16, LD.S8, LD.U16, LD.U8, LD.W0, LD.W1, MV.16, MV.32, MV.64, MV.8, MV.U16, MV.U8, OR.16, OR.8, RET{D}, S16TOF, ST.16, ST.32, ST.64, ST.8, ST.B0, ST.B1, ST.B2, ST.B3, ST.W0, ST.W1, SUB.32, SUB.S16, SUB.S8, SUBR.32, SUBR.S16, SUBR.S8, U16TOF, XOR.16, XOR.8

Examples:

```
; Load the 32-bit value in ADDR1 into Mx, using a base address + offset
; (#u7imm) and then post increment by Ak (Ak is number of bytes to increment)
; NOTE: make sure 32-bit alignment of base address (Ax) and offset
LD.32 Mx,ADDR1              ; field
LD.32 Mx,*(Ax+#u7imm)++Ak   ; addressing mode
LD.32 M1,*(A14+#100)++A2    ; actual assembly code

; OR #x16 with the address pointed to by ADDR1, and store the result
; into the location pointed to by ADDR1. Then post decrement the Ax register
; by the #n8imm value
; NOTE: make sure 16-bit alignment of base address (Ax) and offset
OR.16 ADDR1,#x16            ; field
OR.16 *Ax--[#n8imm],#x16    ; addressing mode
OR.16 *A3--[#70],#50110     ; actual assembly code
```

4.2.2 ADDR2 Field

This is a 5-bit field for indirect encoding of addresses that can be used in all "Pointer Addressing" modes.

Table 4-3 shows the various ways the 5 bits can be used to encode the address.

Table 4-3. ADDR2 Field Encodings

ADDR2 Field: (Az = A4 to A7)							
Mnemonic	**Shorthand**	**Address Generation**	**4**	**3**	**2**	**1**	**0**
*(Az++A0)	*Az++A0	addr = Az, Az = Az + A0	0	0	0	Az[4-7]	
*(Az++A1)	*Az++A1	addr = Az, Az = Az + A1	0	0	1	Az[4-7]	
*(Az+A0<<#scale)	*Az[A0]	addr = Az + A0 << (0/1/2/3) (1)	0	1	0	Az[4-7]	
*(Az+A1<<#scale)	*Az[A1]	addr = Az + A1 << (0/1/2/3) (1)	0	1	1	Az[4-7]	
*Az	*Az	addr = Az	1	0	0	Az[4-7]	
*(Az++#size)	*Az++	addr = Az, Az = Az + (1/2/4/8) (#size = 1,2,4,8) (1)	1	0	1	Az[4-7]	
*(Az--#size)	*Az--	addr = Az, Az = Az – (1/2/4/8) (#size = 1,2,4,8) (1)	1	1	0	Az[4-7]	
*(Az-=#size)	*--Az	Az = Az – (1/2/4/8), addr = Az (#size = 1,2,4,8) (1)	1	1	1	Az[4-7]	

Note

The ADDR2 opcode field modes do not specify the increment step size ("#size") or scale amount ("#scale"). This is automatically performed by the CPU hardware based on the word size being accessed by the instruction.

- Byte access increments/decrements by #size=1, or scales by #scale=0 (multiply by 1)
- 16-bit access increments/decrements by #size=2, or scales by #scale=1 (multiply by 2)
- 32-bit access increments/decrements by #size=4, or scales by #scale=2 (multiply by 4)
- 64-bit access increments/decrements by #size=8, or scales by #scale=3 (multiply by 8)

The following are the instructions that can use the ADDR2 field:

AND.32, ANDOR, LD.32, LD.64, MV.16, MV.32, MV.64, MV.8, OR.32, ST.16, ST.32, ST.64, ST.8, XOR.32

Examples:

```
; Register XMx is loaded with the 64-bit word at the memory location
; addressed using the ADDR2 addressing mode. This address is fetched from Az.
LD.64 XMx,ADDR2             ; field
LD.64 XMx,*Az               ; addressing mode
LD.64 XM4,*A4               ; actual assembly code

; The 8-bit immediate value specified is stored at the memory location
; addressed using the ADDR2 addressing mode. The address is fetched from Az,
; which is then post-decremented by the amount in #size. The #size is
; automatically chosen by the CPU to be 1 since the word size accessed by
; this instruction is 8-bit.
ST.8 ADDR2,#0x0B            ; field
ST.8 *(Az--#size),#0x0B     ; addressing mode
ST.8 *(Az--#1),#0x0B        ; actual assembly
```

4.2.3 ADDR3 Field

This is a 12-bit field for indirect encoding of addresses used only for "Pointer Addressing with #Immediate Offset."

Table 4-4 shows the various ways the 12 bits can be used to encode the address.

Table 4-4. ADDR3 Field Encodings

ADDR3 Field: (Ax = A0 to A14)														
Mnemonic	**Shorthand**	**Address Generation**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
*(Ax+#u8imm<<2)	*Ax[#u8imm]	addr = Ax + #u8imm<<2 (#u8imm << 2 = 0 to 1KB range, 4B steps)	Ax[0-14] [1]				#u8imm							

(1) The Ax[0-14] addressing field can support the A15 register, however this is the stack pointer (SP) register and for some of the addressing modes, the operation is not valid for the SP and hence the addressing mode can not be used.

The following are the instructions that can use the ADDR3 field:

MV.32

Example:

```
; The 32-bit content at the memory location addressed using the ADDR3
; addressing mode, ADDR3_x, is copied to the memory location addressed using
; the ADDR3 addressing mode, ADDR3_y. Both ADDR3 fields use the same
; addressing mode "*(Ax+#u8imm<<2)", which calculates the address using a
; base pointer added with an 8-bit immediate (#u8imm) that is multiplied by 4
; (#u8imm<<2). NOTE: The base address must be 32-bit aligned.
MV.32 ADDR3_y,ADDR3_x                       ; field
MV.32 *(Ax+#u8imm<<2),*(Ax+#u8imm<<2)       ; addressing mode
MV.32 *(A0+#4<<2),*(A1+#8<<2)               ; actual assembly
```

4.2.4 DIRM Field

DIRM Field

This is a 33-bit encoding used for direct and indirect encoding of addresses used only for "Direct Addressing" and “Pointer Addressing With #Immediate Offset.”

Table 4-5 shows the various ways the 12 bits can be used to encode the address.

Table 4-5. DIRM Field Encodings

DIRM Field: (Ax = A0 to A14)						
Mnemonic		**Address Generation**	**0**	**31:20**	**19:16**	**47:32**
*(0:#u32imm)	@u32imm	addr = #u32imm	0	#u32imm		
*(Ax+#u28imm)	*Ax[#u28imm]	addr = Ax + #u28imm (#u28imm = 0 to 256MB range)	1	#u28imm (lower 12-bits)	Ax[0-14] [1]	#u28imm (upper 16-bits)

(1) The Ax[0-14] addressing field can support the A15 register, however this is the stack pointer (SP) register and for some of the addressing modes, the operation is not valid for the SP and hence the addressing mode can not be used.

The following are the instructions that can use the DIRM field:

LD.32, LD.64, LD.B0, LD.B1, LD.B2, LD.B3, LD.S16, LD.S8, LD.U16, LD.U8, LD.W0, LD.W1, S16TOF, ST.32, ST.64, ST.B0, ST.B1, ST.B2, ST.B3, ST.W0, ST.W1, U16TOF

Examples:

```
; Bits [7:0] of register Ax are loaded with the 8-bit value at the memory
; location addressed using the DIRM addressing mode. DIRM is supplied with a
; 32-bit unsigned immediate value found in parklSine:
; parklSine = 0x00008000
LD.B0 Ax,DIRM               ; field
LD.B0 Ax,@u32imm            ; addressing mode
LD.B0 A8,@parklSine         ; actual assembly

; The upper 16-bit content of register Ax is stored at the memory location
; addressed using the DIRM addressing mode. The DIRM field is replaced with
; the "*(Ax+#u28imm)" addressing mode, where the address is found using
; base pointer Ax and the #u28imm immediate value.
ST.W1 DIRM,Ax               ; field
ST.W1 *(Ax+#u28imm),A10     ; addressing mode
ST.W1 *(A3+#0x4),A10        ; actual assembly
```

4.2.5 Additional Fields

In addition to the addressing mode fields, there are #immediate fields that are used within the actual addressing modes, such as "#u10imm" in the "*(Ax+#u10imm)" addressing mode. Most of these #immediate fields (also called constants) are self explanatory (for example, #u10imm is an unsigned 10-bit immediate).

However, there are two negative #immediate fields that are explained in further detail using a table for clarity:

#n13imm Field

The #n13imm field is a 13-bit negative offset #immediate used in the "*(A15-#n13imm)" addressing mode. This addressing mode is one of the available ADDR1 fields (requires 16 bits for encoding) and is of type "Stack Addressing".

A negative 13-bit value is provided using this #immediate, and bits 13 to 31 are padded with 1s to create the 32-bit negative offset constant.

Note

Bits 13 to 31 are padded with 1s to create a 32-bit negative offset constant that is then added to the addressing register.

Table 4-6. #n13imm Field Encoding

12	11	10	9	8	7	6	5	4	3	2	1	0	Encoded Value	Sign-extended Value
1	1	1	1	1	1	1	1	1	1	1	1	1	1	-1
1	1	1	1	1	1	1	1	1	1	1	1	0	2	-2
...														
1	0	0	0	0	0	0	0	0	0	0	0	1	4095	-4095
1	0	0	0	0	0	0	0	0	0	0	0	0	4096	-4096
0	1	1	1	1	1	1	1	1	1	1	1	1	4097	-4097
...														
0	0	0	0	0	0	0	0	0	0	0	0	1	8191	-8191
0	0	0	0	0	0	0	0	0	0	0	0	0	8192	-8192

#n8imm Field

The #n8imm field is a 8-bit negative offset #immediate used in the following addressing modes:

- n
- *(Ax--#n8imm), which is addressing mode type "Pointer Addressing With #Immediate Increment/Decrement"
- *(Ax-=#n8imm), which is addressing mode type "Pointer Addressing With #Immediate Increment/Decrement"
- *(A15-=#n8imm), which is addressing mode type "Stack Addressing"

These addressing modes are all part of the available ADDR1 fields (all require 16 bits for encoding).

A negative 8-bit value is provided using this #immediate, and bits 8 to 31 are padded with 1s to create the 32-bit negative offset constant.

Note

Bits 8 to 31 are padded with 1s to create a 32-bit negative offset constant that is then added to addressing register.

Table 4-7. #n8imm Field Encoding

7	6	5	4	3	2	1	0	Encoded Value	Sign-extended Value
1	1	1	1	1	1	1	1	1	-1
1	1	1	1	1	1	1	0	2	-2
...									
1	0	0	0	0	0	0	1	127	-127
1	0	0	0	0	0	0	0	128	-128
0	1	1	1	1	1	1	1	129	-129
...									
0	0	0	0	0	0	0	1	255	-255
0	0	0	0	0	0	0	0	256	-256

4.3 Alignment and Pipeline Considerations

This section covers the requirements for addressing alignment and pipeline considerations for addressing modes.

4.3.1 Alignment

All data accesses are aligned to the nearest word size. This is enforced by the memory or peripheral being accessed.

This means that the following are required for all data accesses:

- The base pointer address must be aligned to the data word width.
- Any offsets or increment/decrement sizes must be a multiple of the data size.
- The final address (base pointer with any offsets or increment/decrements applied) must be aligned to the data word width.

CAUTION

The compiler automatically takes care of appropriate offset indexing and scaling based on the word size. However, if the base pointer was loaded from a memory location, the compiler assumes that the contents are properly aligned. If the contents are not aligned, then a CPU addressing fault is generated if the generated address for that particular word size is not aligned.

An example regarding the base address: The base pointer address must be aligned to the data word width. So if a 64-bit (8-byte) data instruction like "LD.64" is used, the base address must be aligned to the 64-bit word boundary. Therefore, the last three digits of the address in binary must be 0, since that means the value is divisible by 8. "LD.32 D2,*(0:**#0xF8**)" can therefore be valid (because in binary, this is 0b1111 1**000**), but "LD.32 D2,*(0:**#0xF9**)" can not be valid (because in binary, this is 0b1111 1**001**).

An example regarding the offsets: Any offset value used (which is in bytes) must be a multiple of the instruction's data size. So if a 32-bit (4-byte) data instruction like "LD.32" is used, the offset must be a multiple of 4. "LD.32 D2,*(A2 + **#4**)" can therefore be valid, but "LD.32 D2,*(A2 + **#5**)" can not be valid. Alignment of the base pointer is also required for these instructions.

Some additional examples of correct and incorrect alignment are provided here:

```
MV.32       A2,#ArrayX          ; Assume that the array is aligned
                                ; to a 64-bit word boundary for this example.
; CORRECT Examples:
; Pointer Addressing With #Immediate Offset Examples
LD.B0       D0,*(A2+#9)     ; Byte offset can be any value
LD.U16      D1,*(A2+#10)    ; 16-bit offset can only be a multiple of 2 bytes
LD.32       D2,*(A2+#4)     ; 32-bit offset can only be a multiple of 4 bytes
LD.64       XD4,*(A2+#16)   ; 64-bit offset can only be a multiple of 8 bytes
; Scaled values (left shift or multiplied values)
LD.U16      D1,*(A2+#1<<1)  ; 16-bit offset can only be a multiple of 2 bytes
LD.U16      D1,*(A2+#3<<1)  ; 16-bit offset can only be a multiple of 2 bytes
LD.64       XD4,*(A2+#2<<3) ; 64-bit offset can only be a multiple of 8 bytes
; Pointer Addressing with #Immediate Increment/Decrement Examples
LD.B0       D0,*(A2++#9)    ; Byte offset can be any value
LD.U16      D1,*(A2++#10)   ; 16-bit offset can only be a multiple of 2 bytes
LD.32       D2,*(A2++#4)    ; 32-bit offset can only be a multiple of 4 bytes
LD.64       XD4,*(A2++#24)  ; 64-bit offset can only be a multiple of 8 bytes

; INCORRECT Examples:
LD.U16      D1,*(A2++#5)    ; INCORRECT: offset can only be a multiple of 2
LD.U16      D1,*(A2+#3<<0)  ; INCORRECT: offset can only be a multiple of 2
LD.64       XD4,*(A2+#10)   ; INCORRECT: offset can only be a multiple of 8
; If ArrayX is not aligned to a 32-bit boundary and LD.32 is called,
; then a CPU addressing fault is generated.
```

Note

The ADDR2 opcode field modes do not specify the increment step size ("#size") or scale amount ("#scale"). This is automatically performed by the CPU hardware based on the word size being accessed by the instruction.

- Byte access increments/decrements by #size=1, or scales by #scale=0 (multiply by 1)
- 16-bit access increments/decrements by #size=2, or scales by #scale=1 (multiply by 2)
- 32-bit access increments/decrements by #size=4, or scales by #scale=2 (multiply by 4)
- 64-bit access increments/decrements by #size=8, or scales by #scale=3 (multiply by 8)

4.3.2 Pipeline Considerations

While the C29x CPU implements a fully protected pipeline, there are some considerations required:

- Up to two loads and one store can occur in parallel within an instruction packet. This can include modifying the Ax addressing registers. One Ax register can be read multiple times in parallel, but the register cannot be written to more than once in parallel in a single packet.
- The assembly language tools flag if any assembly code has more than one instruction modifying the same destination register within the same instruction packet.

An example of valid code is provided with two loads and one store in a single packet:

```
LD.32       D0,*A2+A0       ; Use A0 as an index from A2
||LD.32     D1,*A2+A1       ; Use A1 as an index from A2
||ST.32     *(A2-=#4),D3    ; Pre-Decrement A2
||ADD       A0,A0,#6        ; Add #6 to A0 register
||SUB       A1,A1,#10       ; Sub #10 from A1 register
; each Ax register is only modified once here.
```

4.4 Types of Addressing Modes

This section provides details on each of the types of addressing modes available in the C29x CPU.

4.4.1 Direct Addressing

The Direct Addressing type allows direct read or write access to any location in the 32-bit memory space with the immediate address provided in the instruction.

The typical use case is for accessing fixed address locations (such as peripheral registers) or variables that are at fixed memory locations (at build time).

Drawbacks of the type include that this addressing mode is only available on 48-bit instructions and so code that extensively uses this type of addressing mode uses more space.

Benefits of the type include that this addressing mode does not require any addressing pointer. If the values being accessed are few and randomly dispersed in the user program, this type of addressing mode can be more efficient than trying to initialize a pointer and use pointer addressing.

Examples

```
LD.U16 D0,@ADC2.ResultReg5  ; Load register D0 with the location of
                            ; ADC2 peripheral result register 5
```

4.4.2 Pointer Addressing

4.4.2.1 Pointer Addressing with #Immediate Offset

The Pointer Addressing with #Immediate Offset type allows indirect read or write access to any location in the 32-bit memory space with the pointer address from one of the addressing registers, A0 to A14, and an optional immediate offset provided in the instruction.

The immediate offset is added to the base register using a full 32-bit unsigned ADD operation. If the value overflows, the value wraps around.

The typical use case is for indexing into a given data array, or a peripheral, in any random order multiple times. Each addressing mode within this type is tailored to a specific use:

*(Ax+#u28imm)	For implementing position independent code, or when accessing very large data arrays.
*(Ax+#u10imm)	For accessing data arrays of 1KB or less.
*(Ax+#u10imm<<2)	For accessing peripheral registers (peripherals have register ranges of 4KB, or multiples of 4KB, and are aligned on 32-bit word boundary).
*(Ax+#u8imm<<2)	For moving multiple data entries between two 32-bit data arrays of less than 1KB. Used only in one data move instruction, which enables this functionality to in a compact 32-bit instruction.

Note

All data accesses must be aligned to the nearest word size. See Section 4.3.1 for more details and cautions regarding alignment.

4.4.2.2 Pointer Addressing with Pointer Offset

The Pointer Addressing with Pointer Offset type allows indirect read or write access to any location in the 32-bit memory space with the pointer address (base address register) from one of the addressing registers, A0 to A14, and an offset provided by an additional pointer (index register) in the instruction.

This structure allows for easy data array access using a variable index. An example of this can be demonstrated with a 32-bit integer array. If the eighth int in the array is needed, the element can be accessed as follows:

```
; Because the array is of type int, each element is 4 bytes (32 bits) long.
;    So the index must be multiplied by 4 (which is the same as <<2)
; Starting parameters:
;    int arr[7] = 12
;    A2 = arr (base address)
;    A0 = i (index) = 7 (the eigth int in the array)

LD.32 D0,*(A2+A0<<2)    ; D0 = arr + (7<<2 byte offset)

; Result:
;    D0 = 12
```

- 8-bit accesses do not need the index "i" to be multiplied (no shift needed). 16-bit accesses require "i" to be multiplied by 2 (<<1) for 2-byte alignment.
- 32-bit accesses require "i" to be multiplied by 4 (<<2) for 4-byte alignment.
- 64-bit accesses require "i" to be multiplied by 8 (<<3) for 8-byte alignment.

The offset provided from the register and shift is added to the base register using a full 32-bit unsigned ADD operation. If the value overflows, the value wraps around.

This allows for negative index values to wrap around:

```
; Starting parameters:
;    A2 = arr =  8 = 0x0000 0008 (base address at 8th byte in memory space)
;    A0 = i   = -1 = 0xFFFF FFFF (index at -1)
*(A2+A0) = 8 + (-1) = 7th byte in memory space
```

Note

All data accesses must be aligned to the nearest word size. See Section 4.3.1 for more details and cautions regarding alignment.

4.4.2.3 Pointer Addressing with #Immediate Increment/Decrement

The Pointer Addressing with #Immediate Increment/Decrement type allows indirect read or write access to any location in the 32-bit memory space with the pointer address from one of the addressing registers, A0 to A14. An immediate pre or post increment or decrement of the register is applied.

The increment offset size provided from the #Immediate value and is added to the base register using a full 32-bit unsigned ADD operation. If the value overflows, the value wraps around.

The decrement offset size provided from the #Immediate value and is added to the base register using a full 32-bit unsigned SUB operation. If the value underflows, the value wraps around.

Note

All data accesses must be aligned to the nearest word size. See Section 4.3.1 for more details and cautions regarding alignment.

Note

The ADDR2 opcode field modes do not specify the increment step size ("#size") or scale amount ("#scale"). This is automatically performed by the CPU hardware based on the word size being accessed by the instruction.

- Byte access increments/decrements by #size=1, or scales by #scale=0 (multiply by 1)
- 16-bit access increments/decrements by #size=2, or scales by #scale=1 (multiply by 2)
- 32-bit access increments/decrements by #size=4, or scales by #scale=2 (multiply by 4)
- 64-bit access increments/decrements by #size=8, or scales by #scale=3 (multiply by 8)

4.4.2.4 Pointer Addressing with Pointer Increment/Decrement

The Pointer Addressing with #Immediate Increment/Decrement type allows indirect read or write access to any location in the 32-bit memory space with the pointer address from one of the addressing registers, A0 to A14, and a pre or post increment or decrement of the register is applied using the value located in an additional pointer register.

One of the addressing modes in this type, "*(Ax+#u7imm)++Ak", allows for a pointer increment/decrement along with an offset. This is useful in code where values are accessed close to a variable index. An example of this can be seen in the C and resultant assembly code:

C code:

```
For (i=0; i<N; i++)
{
    ArrayY[i]   = ArrayX[i] + ArrayX[i+1];
    ArrayY[i+1] = ArrayX[i] - ArrayX[i+1];
}
```

Resultant assembly code:

```
; Initialize ArrayX and ArrayY Pointers and i:
MV          A0,#4                   ; A0 = i = 4 = increment step size
MV          A2,#ArrayX              ; A2 = ArrayX base address
MV          A3,#ArrayY              ; A3 = ArrayY base address
...
; This code is repeated N times:
LD.32       D0,*(A2 + #0)           ; D0 = ArrayX[i]
||LD.32     D1,*(A2 + #1*4)++A0     ; D1 = ArrayX[i+1], A2 = A2 + A0
ADD         D2,D1,D0                ; D2 = ArrayX[i] + ArrayX[i+1];
||SUB       D3,D1,D0                ; D3 = ArrayX[i] - ArrayX[i+1];
ST.32       *(A3 + #0),D2           ; ArrayY[i] = D2
ST.32       *(A3 + #1*4)++A0        ; ArrayY[i+1] = D3, A3 = A3 + A0
```

The increment offset size provided from the #Immediate value and is added to the base register using a full 32-bit unsigned ADD operation. If the value overflows, the value wraps around.

This wrap around can be used to implement a decrementing index. For example:

```
; Starting parameters:
;    A2 = arr =  8 = 0x0000 0008 (base address at 8th byte in memory space)
;    A0 = i   = -1 = 0xFFFF FFFF (index at -1)
*(A2+A0) = 8 + (-1) = 7th byte in memory space
```

Note

All data accesses must be aligned to the nearest word size. See Section 4.3.1 for more details and cautions regarding alignment.

4.4.3 Stack Addressing

The Stack Addressing type allows read or write access to any location in the stack space with the address provided in addressing register A15, which is the dedicated Stack Pointer (SP).

Following is a list of key information regarding the stack pointer that helps in understanding these addressing modes:

- Addressing register A15 is the dedicated stack pointer.
- The stack grows from low address to high address.
- The stack pointer always points to the next empty location at the top of the stack.
- The stack pointer must always be aligned to a 64-bit word boundary. Interrupts, Call, and Return operations generate a fault, if the stack pointer is not aligned.
- If a stack pointer is not aligned to the word size being accessed, this also generates a fault.

When allocating stack space and accessing values on the stack, the recommended procedure is as follows:

- Allocate stack space in increments of 8-bytes (64-bits)
- Access the stack contents using the ***(A15-#n13imm)** addressing mode (all accesses must be aligned to the word size being accessed)
- When done, de-allocate stack space (in decrements of 8-bytes)

For example: The program needs to allocate space for:

- 1* 64-bit value
- 3 * 32-bit values
- 1 * 16-bit value
- 3 * 8-bit values (bytes)

The total number of bytes to allocate, taking into account the alignment can be 32 (which is the nearest 64-bit address above the required 25 bytes):

```
64-bit  +8      8  bytes total      requires 8 bytes allocated
32-bit  +4      12 bytes total      requires 16 bytes allocated
32-bit  +4      16 bytes total
32-bit  +4      20 bytes total      requires 24 bytes allocated
16-bit  +2      22 bytes total
8-bit   +1      23 bytes total
8-bit   +1      24 bytes total
8-bit   +1      25 bytes total      requires 32 bytes allocated

Total Used  = 25 bytes
Allocated   = 32 bytes (closest multiple of 8-bytes [64-bits])
```

4.4.3.1 Allocating and De-allocating Stack Space

Stack space can be allocated and de-allocated as shown in the following examples:

Allocate 32-bytes (must be a multiple of 8-bytes):

```
ADD.U16    A15,A15,#32    ; SP = SP + 32
```

De-allocate 32-bytes (must be a multiple of 8-bytes):

```
SUB.U16    A15,A15,#32    ; SP = SP - 32
```

The compiler automatically allocates and de-allocates stack space and forces alignment to the 64-bit word boundary.

It is also possible to use the ***(A15++#u8imm)** addressing mode to push something on the stack and also allocate additional stack space if required and if the stack size is less than 256 bytes. For example:

```
ST.64   *(A15++#32),XD0     ; Push 64-bit XD0 value on stack, then allocate
                            ; 32 bytes on stack (SP = SP + 32)
```

Similarly you can de-allocate and pop something from the stack using the ***(A15 -= #n8imm)** addressing mode. For example:

```
LD.64   XD0,*(A15-=#32)     ; De-allocate 32-bytes from stack (SP = SP - 32),
                            ; and pop 64-bit value from stack into XD0
```

If required to access a value on the stack that is a distance greater than 8192 bytes, an addressing pointer needs to be used to access the value. For example: to access a 32-bit value that is 8216 bytes away from top of stack:

```
SUB.U16     A0,A15,#8216    ; A0 = SP - #8216
LD.32    D0,*A0          ; D0 = contents of stack at SP-8216
```

Typically, for large stacks, the compiler allocates one of the Ax addressing mode registers as a frame pointer and can use the available pointer addressing modes to index into the stack.

The above approach can also be used to initialize pointers within the stack for situations where local variables located on stack need to be accessed frequently or if required to use pointer increment/decrement operations on the data.

Note that regardless of the addressing mode used, any stack memory access must be aligned to the accessed word size and any non-aligned access generates a fault. The compiler takes care of alignment of any data on the stack space.

Note

1. If a pointer is used to access the stack contents, assume that the value is appropriately aligned to the right word access size. If the value gets corrupted, it can cause an access fault.
2. The CALL operations automatically push the RPC (return PC) value on the stack and increment the stack pointer by 8, hence always keeping stack alignment. The 32-bit RPC is stored in the lower 32-bits of the 64-bit word. Similarly, a RET operation pops the RPC value from the stack and decrements the SP by 8 maintaining stack alignment. If the stack pointer is not aligned when executing the CALL or RET operation, a fault is generated.

4.4.4 Circular Addressing Instruction

The C29x CPU does not support a native addressing mode for circular addressing like on the C28x CPU. However, the functional parallelism present in the C29x CPU architecture makes sure that there is no performance impact for the lack of native circular addressing mode.

Circular addressing is performed by instructions that modify the addressing registers in a circular fashion. These are 16-bit instructions that require 1 cycle to execute. The instructions supported are:

INC.CIRC Ay,Ax:

Increment Ay until the limit (Ax) is reached, then reset the value to 0.

```
if (Ay >= Ax)   Ay = 0
else            Ay = Ay + 1
; where  Ay = A0 to A3
; and    Ax = A0 to A14
```

DEC.CIRC Ay,Ax:

Decrement Ay until the limit (0) is reached, then reset the value to Ax.

```
if (Ay <= 0)    Ay = Ax
else            Ay = Ay - 1
; where  Ay = A0 to A3
; and    Ax = A0 to A14
```

This type of addressing mode is typically used for implementing finite impulse response (FIR), least mean squares (LMS), or convolution filters.

A typical FIR filter algorithm in C:

```
sum = 0;
circ_index = save_circ_index;
for(i=0; i < N_taps; i++)
{
    sum += Data[circ_index] * Coef[i];
    circ_index++;
    if( circ_index >= N_taps)
        circ_index = 0;
}
save_circ_index = circ_index;
```

The main kernel for the filter can be coded as follows (example for a 7-tap FIR):

```
LD.32           A0,@save_circ_index ; A0 = circ_index
MV              A6,#N-1             ; A6 = filter taps, N = 7
MV              A4,#Data            ; A4 -> Data Array
MV              A5,#Coef            ; A5 -> Coef Array
LD.32           M0,*(A4+A0)         ; Read Data From Current Index
||LD.32         M1,*A5++            ; Read Coef, Increment Coef Pointer
||INC.CIRC      A0,A6               ; if(A0 >= A6) A0 = 0 else A0 = A0 + 1
SMPYF           M4,M0,M1
||LD.32         M0,*(A4+A0)         ; Read Data From Current Index
||LD.32         M1,*A5++            ; Read Coef, Increment Coef Pointer
||INC.CIRC      A0,A6               ; if(A0 >= A6) A0 = 0 else A0 = A0 + 1
SMPYF           M5,M0,M1
||LD.32         M0,*(A4+A0)         ; Read Data From Current Index
||LD.32         M1,*A5++            ; Read Coef, Increment Coef Pointer
||INC.CIRC      A0,A6               ; if(A0 >= A6) A0 = 0 else A0 = A0 + 1
SMPYF           M6,M0,M1
||LD.32         M0,*(A4+A0)         ; Read Data From Current Index
||LD.32         M1,*A5++            ; Read Coef, Increment Coef Pointer
||INC.CIRC      A0,A6               ; if(A0 >= A6) A0 = 0 else A0 = A0 + 1
SMPYF           M7,M0,M1
||LD.32         M0,*(A4+A0)         ; Read Data From Current Index
||LD.32         M1,*A5++            ; Read Coef, Increment Coef Pointer
||INC.CIRC      A0,A6               ; if(A0 >= A6) A0 = 0 else A0 = A0 + 1
SMPYF           M4,M0,M1
||SADDF         M6,M6,M4
||LD.32         M0,*(A4+A0)         ; Read Data From Current Index
||LD.32         M1,*A5++            ; Read Coef, Increment Coef Pointer
||INC.CIRC      A0,A6               ; if(A0 >= A6) A0 = 0 else A0 = A0 + 1
SMPYF           M5,M0,M1
||SADDF         M7,M7,M5
||LD.32         M0,*(A4+A0)         ; Read Data From Current Index
||LD.32         M1,*A5++            ; Read Coef, Increment Coef Pointer
||INC.CIRC      A0,A6               ; if(A0 >= A6) A0 = 0 else A0 = A0 + 1
SMPYF           M4,M0,M1
||SADDF         M6,M6,M4
ADDF            M7,M7,M5
ADDF            M6,M6,M4
ST.32           @save_circ_index,A0 ; Save current circ index position
ADDF            M7,M7,M6            ; final sum = M7
```

4.4.5 Bit Reversed Addressing Instruction

The C29x CPU does not support a native bit reversed addressing mode like on the C28x CPU. However, the functional parallelism present in the C29x CPU architecture makes sure that there is no performance impact for the lack of native bit reversed addressing mode.

Bit reversed addressing is performed by an instruction that modifies the addressing registers in a bit reversed fashion and is typically used for re-ordering data for Fast-Fourier Transform (FFT) type algorithms.

Table 4-8. Bit Reversed Addressing Visualized

Address	Value	Bit Reversed Address	Bit Reversed Value
0000	0	0000	0
0001	1	1000	8
0010	2	0100	4
0011	3	1100	12
0100	4	0010	2
0101	5	1010	10
0110	6	0110	6
0111	7	1110	14
1000	8	0001	1
1001	9	1001	9
1010	10	0101	5
...	...	...	...

The supported instruction for bit reversed addressing is:

ADD.BITREV Az,Ay,Ax

Perform the ADD operation, but add the bits from left to right (unlike a standard ADD that is from right to left). An example is:

```
; Ax = 0011 1001
; Ay = 0000 1000
; Az = 0011 0101 (after a bit reversed add):
ADD          Az,Ay,Ax    ; Normal Add:        Az = 0100 0001
ADD.BITREV   Az,Ay,Ax    ; Bit Reversed Add:  Az = 0011 0101
```

The following example shows how this operation is used to reverse an array of Data in bit reversed order:

```
BitReversedIndex        = 0;
BitReversedIncrement    = N/2;
for(i=0; i < N; i++)
{
    BitReversedDataArray[BitReversedIndex] = NormalDataArray[i];
    BitReversedIndex = BitReversedAdd(BitReversedIndex+BitReversedIncrement);
}
```

Typically, when bit reversing data, the data array is a multiple of 2 in size (N = 16, 32, 64, 128, and so on).

The BitReversedIncrement then needs to be set to half the array size (N/2) to increment by 1 in bit reversed order.

The assembly code for the previous operation is:

```
MV              A0,#0                       ; A0 = BitReversedIndex = 0
MV              A8,#N/2                     ; A8 = Increment Step = N/2
MV              A4,#NormalDataArray         ; A4 = Stating Address Of
                                            ;      NormalDataArray
MV              A5,#BitReversedDataArray    ; A5 = Stating Address Of
                                            ;      BitReversedDataArray

; Repeat N times:
LD.32           D0,*A4++                    ; Read From NormalDataArray
ST.32           *(A5+A0),D0                 ; Write To BitReversedDataArray
||ADD.BITREV    A0,A0,A8                    ; Increment BitReversedIndex
LD.32           D0,*A4++                    ; Read From NormalDataArray
ST.32           *(A5+A0),D0                 ; Write To BitReversedDataArray
||ADD.BITREV    A0,A0,A8                    ; Increment BitReversedIndex
....
LD.32           D0,*A4++                    ; Read From NormalDataArray
ST.32           *(A5+A0),D0                 ; Write To BitReversedDataArray
||ADD.BITREV    A0,A0,A8                    ; Increment BitReversedIndex
```

Chapter 5

Safety and Security Unit (SSU)

The Safety and Security Unit (SSU) implements safety, memory management (MPU) and security as one function. This chapter provides a brief overview of the SSU module. Details regarding the SSU are provided in the *F29H85x and F29P58x Real-Time Microcontrollers Technical Reference Manual*.

5.1 SSU Overview

The SSU acts as a filter or firewall between the CPU, memory and peripherals and enforces the user protection policy as the CPU attempts to access peripherals and memory on the chip. All device resources such as Flash, ROM, RAM, and peripherals need to be assigned to Access Protection (AP) ranges. Any code running on the C29x CPU is associated with a LINK through an APx region. This LINK is then associated with a STACK, which is in turn associated with a specific ZONE.

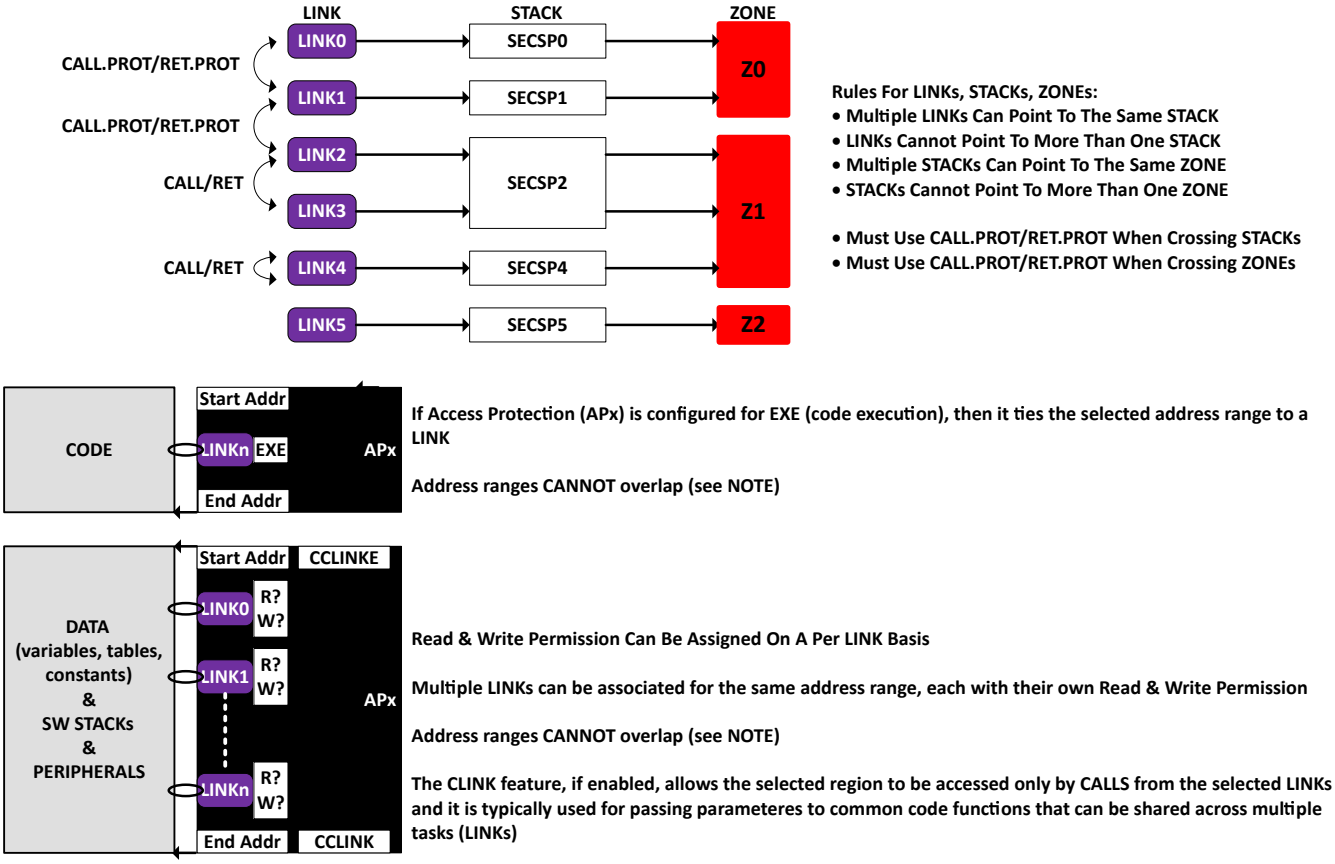

Figure 5-1. SSU Overview

Note

- Standard CALL/RET operations can be used to jump between LINKS, as long as the operations do not jump between STACKs.
- CALL.PROT/RET.PROT can be used even without crossing STACKs, which can be beneficial when initially segmenting code before allocating each function to different STACKs.
- Rules which apply for CALL.PROT applies for LB.PROT
- For a system containing multiple C29x CPUs, each CPU has LINKs, STACKs, and AP regions, while ZONEs are shared across the device.

5.2 Links and Task Isolation

The Safety and Security approach is based on the concept of task isolation. For this section, the example of two distinct tasks (a control task and a communication task) is used. Tasks are unable to view or corrupt the unique program memory, data memory, software stack, and peripheral access of other tasks. From a debugging perspective, each secure zone (ZONE1, ZONE2) has a security password which can only be accessed by enabling the zone for debug with a matching password. Each task has an associated secure Stack Pointer (SECSP2, SECSP3) that is copied into the CPU stack pointer SP = A15 when entering the respective task. When exiting the task, the current contents of the CPU stack pointer is copied into the respective Stack Pointer (SECSP2, SECSP3).

The C29x CPU utilizes the concept of a LINK to tie execution code to a specific task. For example, LINK2 is associated with SECSP2 and ZONE1. Similarly, LINK3 is associated with SECSP3 and ZONE2 as shown in Figure 5-2.

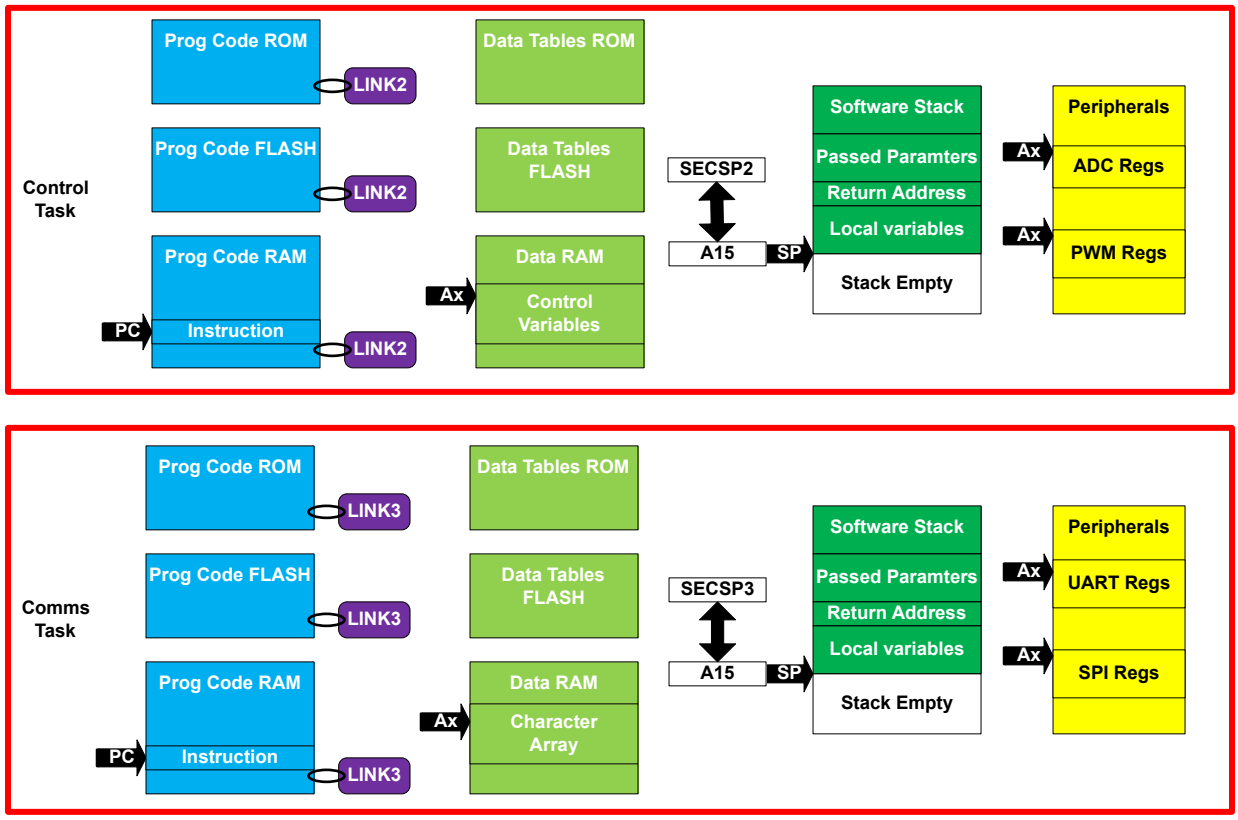

Figure 5-2. Concept of Links for Creating Task Isolation

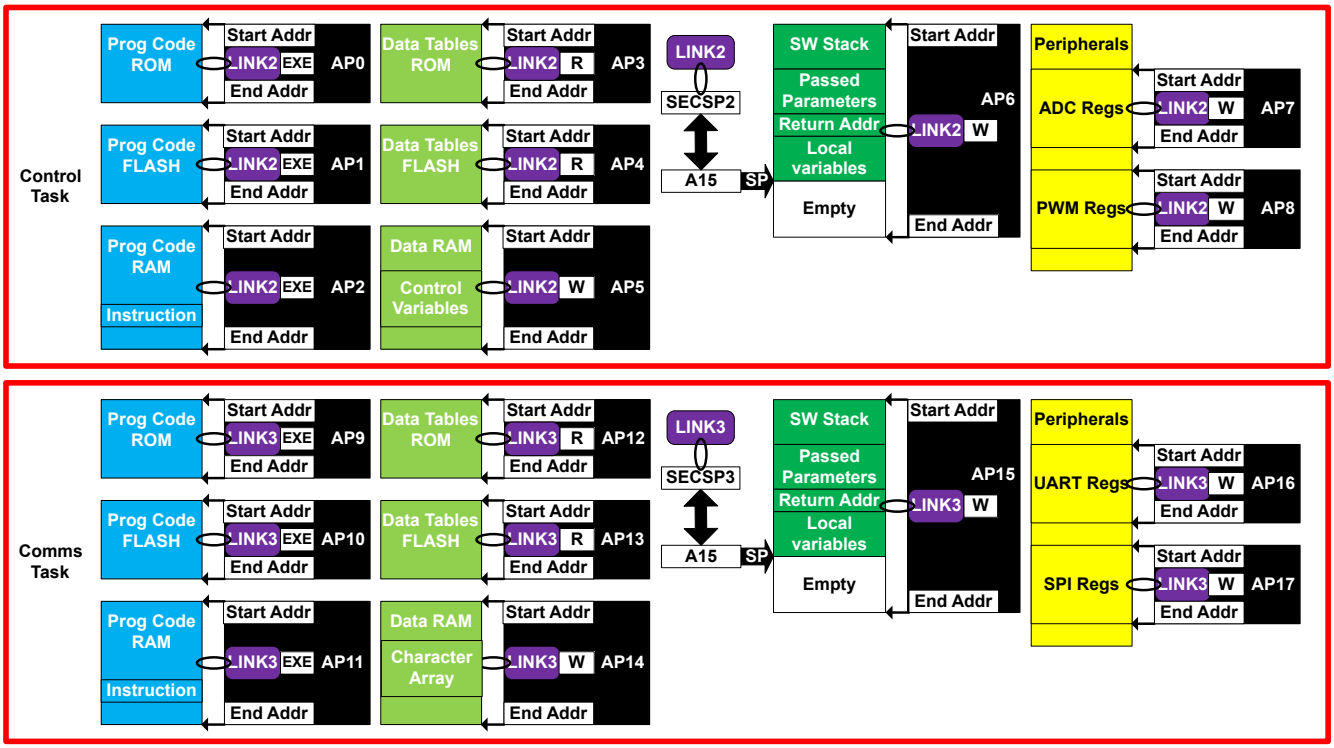

Figure 5-3. Concept of Access Protection to Memories and Peripherals

Note

The minimum address granularity for all Access Protection (AP) regions is 4KB.

5.3 Sharing Data Outside Task Isolation Boundary

When Control Task (LINK2) needs to share data with Communication Task (LINK3), a shared RAM (with AP18) needs to be used with LINK2 having WRITE attribute and LINK3 having READ attribute. In the same way, when Communication Task (LINK3) wants to share data with Control Task (LINK2), another shared RAM (with AP19) needs to be used with LINK3 having WRITE attribute and LINK2 having READ attribute. Similarly, ADC2 Result registers (with AP20) are assigned to LINK2 and LINK3 with READ attribute. This allows both communication and control task to read ADC2 result registers.

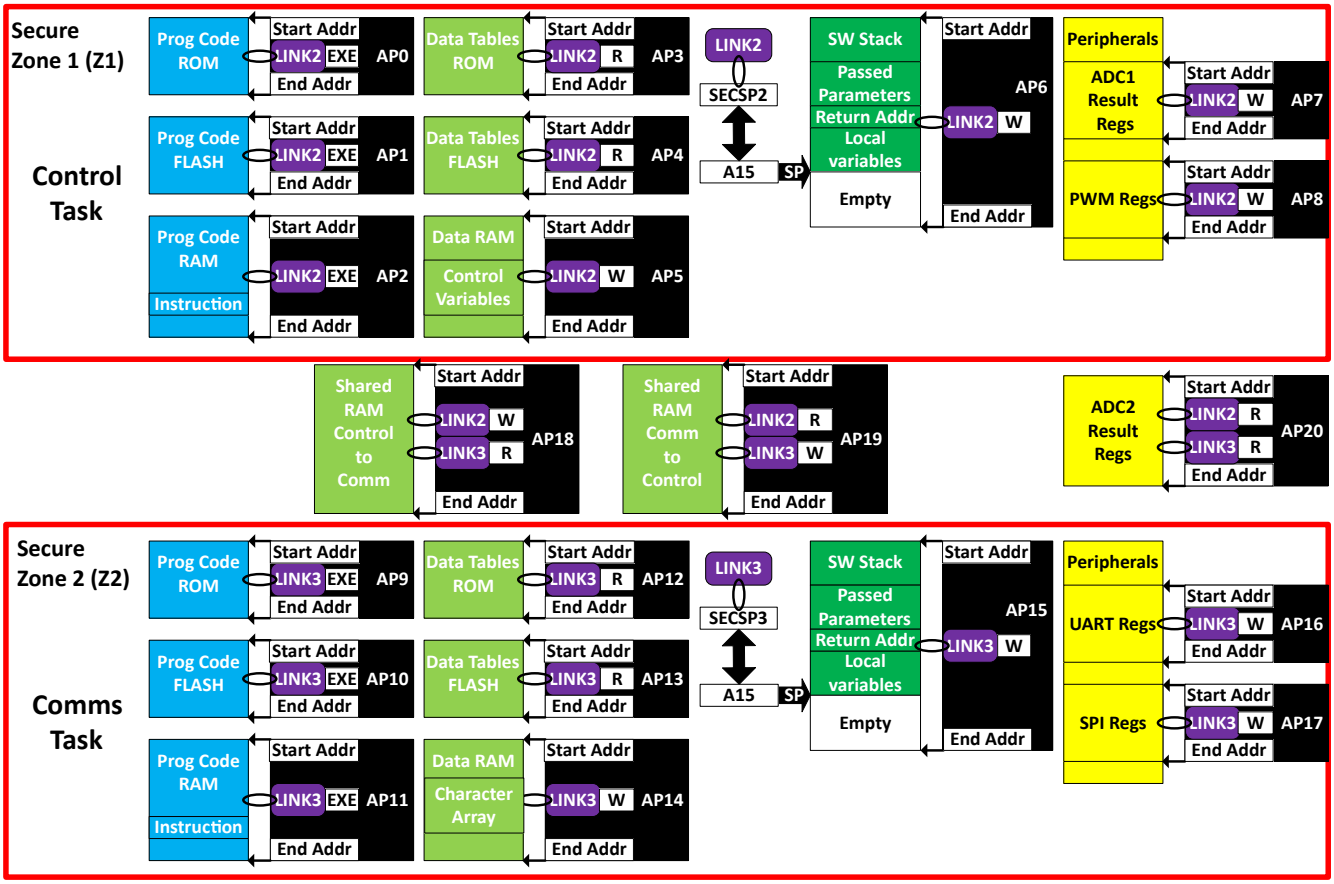

Figure 5-4. Concept of Sharing Data Across LINKS

5.4 Protected Call and Return

The CALL.PROT operation is permitted to cross STACKs. When arriving at the destination address of the CALL.PROT, the ENTRY1.PROT and ENTRY2.PROT instructions must be the first instructions executed. If these instructions are not present, the CPU enters the FAULT state. Upon returning from the CALL.PROT operation, the very first instruction at the return address must be the EXIT.PROT instruction. If the EXIT.PROT instruction is not present, the CPU enters the FAULT state. These ENTRY and EXIT instructions control the entry and exit points of the code when security boundaries are crossed and enable user code to check any passed parameters or data before being used.

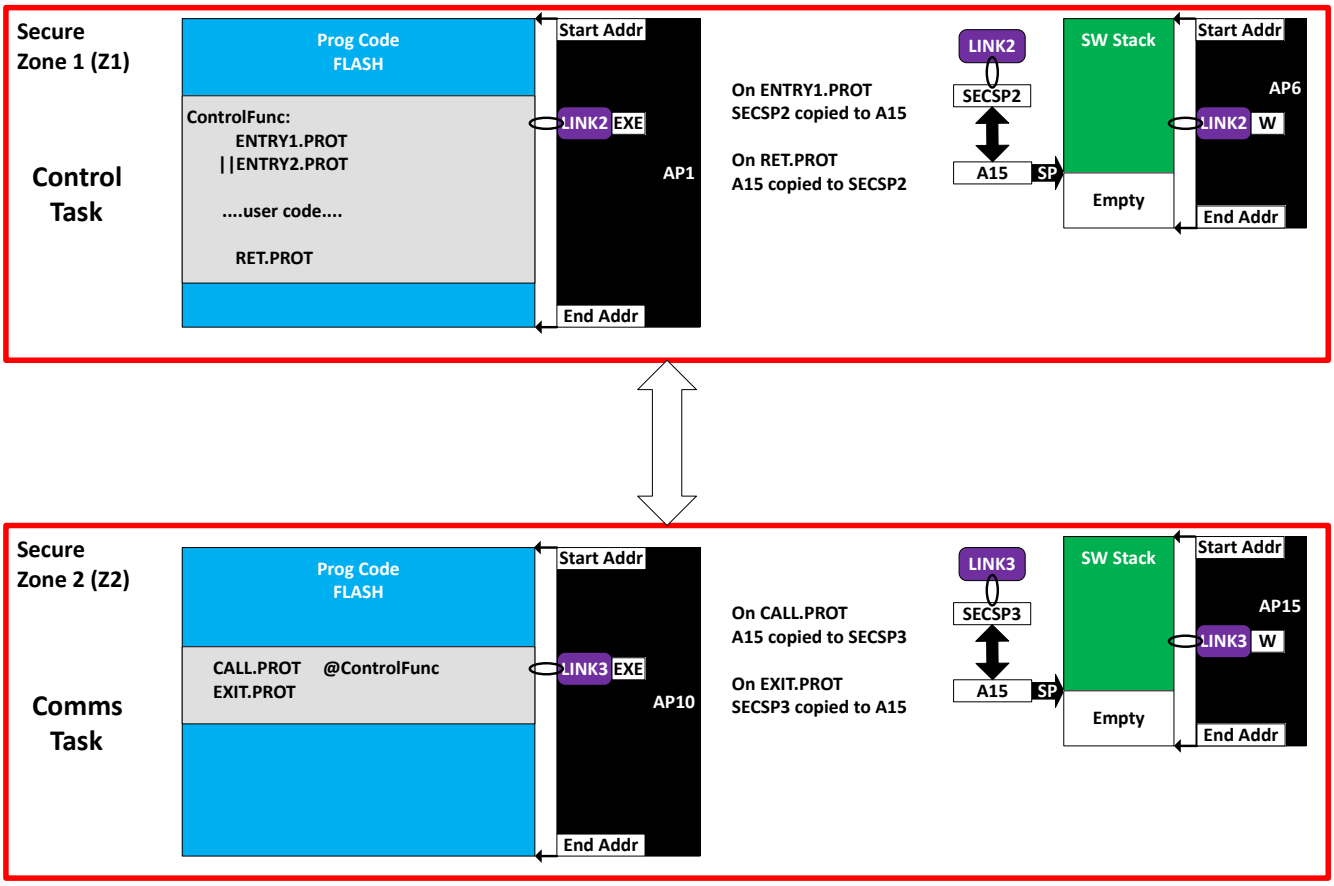

Figure 5-5. Protected Call and Return

Chapter 6

Emulation

The debug controller in the CPU contains hardware extensions for advanced emulation features that can assist you in the development of your application system (software and hardware). This chapter describes the emulation features that are available on all F29x devices using only the JTAG port (with TI extensions).

6.1 Overview of Emulation Features

The CPU's hardware extensions for advanced emulation features provide simple, inexpensive, and speed independent access to the CPU for sophisticated debugging and economical system development, without requiring the costly cabling and access to processor pins required by traditional emulator systems.

This access is provided without intruding on system resources. The on-chip development interface provides:

- Minimally intrusive access to internal and external memory
- Minimally intrusive access to CPU and peripheral registers
- Control of the execution of code:
  - Break on a software breakpoint instruction (instruction replacement)
  - Break on a specified program or data access without requiring instruction replacement
  - Break on external attention request from debug host or additional hardware
  - Break after the execution of a single instruction (single-stepping)
  - Control over the execution of code from device power up
- Nonintrusive determination of device status:
  - Detection of a system reset, emulation/test-logic reset, or power-down occurrence
  - Detection of the absence of a system clock or memory-ready signal
  - Determination of whether global interrupts are enabled
  - Determination of why debug accesses can be blocked
- A cycle counter for performance benchmarking.

6.2 Debug Terminology

The following definitions aid in understanding the information in this chapter:

- **Debug-halt state.:** The state in which the device does not execute code.
- **Debug event:** An action, such as the decoding of a software breakpoint instruction, the occurrence of a breakpoint/watchpoint, external system trigger, or a request from a host processor that can result in special debug behavior, such as halting the device.
- **Break event:** A debug event that causes the device to enter the debug-halt state.

6.3 Debug Interface

The target-level TI debug interface uses the five standard IEEE 1149.1 (JTAG) signals (TRST, TCK, TMS, TDI, and TDO) and the two TI extensions (EMU0 and EMU1). Figure 6-1 shows the 14-pin JTAG header that is used to interface the target to a scan controller, and Table 6-1 defines the pins. As listed in Table 6-1, the header requires more than the five JTAG signals and the TI extensions. The header also requires a test clock return signal (TCK_RET), the target supply (VCC), and ground (GND). TCK_RET is a test clock out of the scan controller and into the target system. The target system uses TCK_RET, if the target system does not supply a test clock (in which case TCK can not be used). In many target systems, TCK_RET is connected to TCK and used as the test clock.

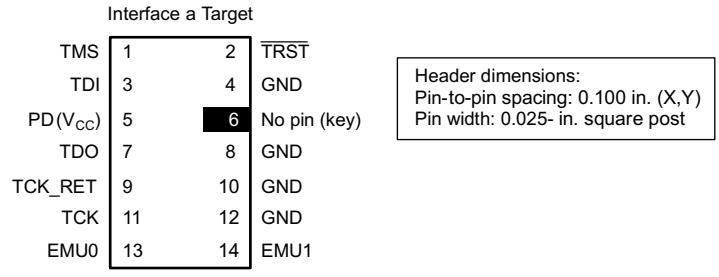

Figure 6-1. JTAG Header to Interface a Target to the Scan Controller

Table 6-1. 14-Pin Header Signal Descriptions

Signal	Description	Emulator State(1)	Target State(1)
EMU0	Emulation pin 0	I	I/O
EMU1	Emulation pin 1	I	I/O
GND	Ground		
PD (V_{CC})	Presence detect. Indicates that the emulation cable is connected and that the target is powered up. PD must be tied to V_{CC} in the target system.	I	O
TCK	Test clock. TCK is a clock source from the emulation cable pod. This signal can be used to drive the system test clock.	O	I
TCK_RET	Test clock return. Test clock input to the emulator. Can be a buffered or unbuffered version of TCK.	I	O
TDI	Test data input	O	I
TDO	Test data output	I	O
TMS	Test mode select	O	I
$\overline{\text{TRST}}$ (2)	Test reset	O	I

(1) I = input; O = output
(2) Do not use pull-up resistors on $\overline{\text{TRST}}$: an internal pull-down resistor is on the device. In a low-noise environment, $\overline{\text{TRST}}$ can be left floating. In a high-noise environment, an additional pull-down resistor can be needed. (The size of this resistor can be based on electrical current considerations.)

The state of the TRST, EMU0, and EMU1 signals at device power up determines the operating mode of the device. The operating mode takes effect as soon as the device has sufficient power to operate. If the TRST signal rises, the EMU0 and EMU1 signals are sampled on the rising edge and the operating mode is latched. Some of these modes are reserved for test purposes, but those that can be of use in a target system are detailed in Table 6-2. A target system is not required to support any mode other than normal mode.

Table 6-2. Selecting Device Operating Modes By Using $\overline{\text{TRST}}$, EMU0, and EMU1

$\overline{\text{TRST}}$	EMU1	EMU0	Device Operating Mode	JTAG Cable Active?
Low	Low	Low	**Peripheral mode**. Disables the CPU and memory portions of the C29x CPU. Another processor treats the C29x CPU as a peripheral.	No
Low	Low	High	Reserved for testing	No
Low	High	Low	**Wait-in-reset mode**. Prolongs the device's reset until released by external means. This allows a C29x CPU to power up in reset, provided external hardware holds EMU0 low only while power-up reset is active.	Yes
Low	High	High	**Normal mode with emulation disabled**. This is the setting that must be used on target systems when a scan controller (such as the XDS510) is not attached. TRST is pulled down and EMU1 and EMU0 pulled up within the C29x CPU; this is the default mode.	No
High	Low or High	Low or High	**Normal mode with emulation enabled**. This is the setting to use on target systems when a scan controller is attached (the scan controller controls TRST). TRST must not be high during device power-up.	Yes

6.4 Execution Control Mode

The C29x CPU supports stop mode debug execution control mode. Stop mode provides complete control of program execution, allowing for the disabling of all interrupts. In this execution mode program execution is suspended at break events, such as occurrence of software breakpoint instructions or specified program-space or data-space accesses.

Stop mode causes break events, such as breakpoints and watchpoints, to suspend program execution at the next interrupt boundary (which is usually identical to the next instruction boundary). When execution is suspended, all interrupts (including NMI and RS) are ignored until the CPU receives a directive to run code again.

In stop mode, the CPU can operate in the following execution states:

- **Debug-halt state:** In the stop mode debug-halt state, the CPU is halted. This state is entered when the CPU is running with debug enabled and encounters a break event such as a breakpoint or watchpoint, hardware trigger or user initiated halt request.
  - User Halt: User issues a Debug-halt request from the debugger.
  - Hardware Breakpoint : ERAD can be setup to generate hardware breakpoints on a specified Program Address. This causes the CPU to go to a halted condition, if the instruction packet at the designated address is about to enter the Decode2 phase of the CPU pipeline.
  - Software breakpoint : This is setup by the debugger by putting the EMUSTOP0 instruction at a desired program address. This causes the CPU to go to a halted condition, if the EMUSTOP0 is about to enter the Decode2 phase of the CPU pipeline.
  - Watchpoint : ERAD can be configured to generate a watchpoint when the CPU makes a designated data memory access or some other system condition or a combination of these. Once this defined event occurs, ERAD generates a Watchpoint request to the Debug controller that can cause the CPU to halt.
  - External Triggers: Triggers at the device level coming from various sources outside the CPU can be configured to make a HALT request to the CPU and can also cause the CPU to HALT. This is typically used when halting of one CPU requires triggering the halt of another CPU when multiple CPUs are being controlled by the Debugger.

  In the debug-halt state, since the CPU is halted, the CPU cannot service any interrupts, including NMI and RS (reset). When multiple instances of the same interrupt occur without the first instance being serviced when the CPU is in halted debug state, the later instances are lost.
- **Single-Step state**: This state is entered when the user indicates to the debugger to execute a single instruction packet. The CPU executes the single instruction packet pointed to by the PC and then returns to the debug-halt state (the CPU executes from one interrupt boundary to the next). The CPU is only in the single-instruction state until that single instruction is done. If an interrupt occurs in this state, the command used to enter this state determines whether that interrupt can be serviced. If DINT is set, the CPU can service the interrupt;.if DINT is not set, the CPU can not service interrupts even if the interrupt is NMI or RS.
- **Run state**: This state is entered from a halted condition when user issues a run command from the debugger interface while stop debug mode is enabled. The CPU executes instructions until a debugger command or a debug event returns the CPU to the debug-halt state. The CPU can service all interrupts in this state. When an interrupt occurs simultaneously with a debug event, the debug event has priority; however, if interrupt processing began before the debug event occurred, the debug event cannot be processed until the interrupt service routine begins.
- **Free-run**: This state is entered from a halted condition when user issues a run command from the debugger interface after disabling stop mode debug mode. The CPU resumes execution and ignores further debug events like breakpoints, watchpoints and triggers and continues execution as if the debugger is no longer connected.
- **Synchronous Run**: This is merely an extension of the basic run state. Based on the configuration, the debug controller can be configured to receive a run request such that the CPU starts the actual run only on a certain global synchronization signal going active. This method is used when starting execution simultaneously on multiple CPUs being controlled by the debugger.

6.5 Breakpoints, Watchpoints, and Counters

6.5.1 Software Breakpoint

The CPU supports the ESTOP instruction that can be used to trigger a halt when executed by the CPU while the debugger is connected. When the debugger is not connected, this instruction is executed as a NOP.

6.5.2 Hardware Debugging Resources

Each C29x CPU-based system has an ERAD (Embedded Real-time analysis and Diagnostics) module that aids with debug and system analysis capabilities. These capabilities can be used either with the debugger connected or as part of real-time application too. The two main components are the Enhanced bus comparator block (EBC) and the System event counter block (SEC), with an optional PC trace module. The number of instances of the EBC and SEC are device dependent.

6.5.2.1 Hardware Breakpoint

The ERAD Enhanced bus comparator (EBC) module is a scalable module that consists of many identical bus comparator units. These units can generate hardware breakpoints. A hardware breakpoint, acts just like a software breakpoint instruction (in this case, the ESTOP0 instruction) but does not require a modification to the application software. Hardware breakpoints allow masking of address bits. Additionally hardware breakpoints allow masking of address bits allowing a breakpoint to be triggered over an address range with just one EBC resource. A hardware breakpoint triggers a debug event and this halts the CPU before the instruction is executed. A bus comparator watches the program address bus, comparing the contents against a reference address and a bit mask value.

6.5.2.2 Hardware Watchpoint

The ERAD Enhanced bus comparator (EBC) module bus comparator units that generate hardware watch points to the CPU by monitoring either the data read address bus or data write address bus.

A hardware watchpoint triggers a debug event when either an address or an address and data match a compare value. The address portion is compared against a reference address and bit mask, and the data portion is compared against a reference data value and a bit mask.

When comparing two addresses, you can set two watchpoints. When comparing an address and a data value, you can set only one watchpoint. When performing a read watchpoint, the address is available a few cycles earlier than the data; the watchpoint logic accounts for this.

The point where execution stops depends on whether the watchpoint was a read or write watchpoint, and whether the watchpoint was an address or an address/data read watchpoint. In the following example, a read address watchpoint occurs when the address X is accessed, and the CPU stops with the instruction counter (IC) pointing three instructions after that point.

For a read watchpoint that requires both an address and data match, the CPU stops with the IC pointing six instructions after that point.

In the following example, a write address watchpoint occurs when the address Y is accessed, and the CPU stops with the IC pointing six instructions after that point.

6.5.2.3 Benchmark Counters

The System event counter module consists of many identical counter units. The number of units available is device specific. These units can be used for various types of system scenarios like:

1. Using counter as a simple system timer
2. Counting of system events (like interrupts, critical system events etc.)
3. Generating interrupts/events based on counter threshold
4. Profiling code segments
5. Measuring number of wait states in code segments
6. Counting duration between system events
7. Counting duration between specified memory reads and writes
8. Counting duration between specified memory reads/writes and system events
9. Measuring minimum and maximum time taken between a pair of events measured over multiple iterations
10. Chaining counters to either link events or to create a larger counter.

This module is accessible both by the debugger and the application software. The access to application software enables the use of the debug and profiling abilities even in the absence of the debugger. This is essential in many real-time systems since it is not always possible to connect a debugger and perform intrusive debug. Under such situations, the user code sets up and controls these modules and is still able to debug and profile the system without disturbing the end application.

6.5.3 PC Trace

This ERAD module has an optional Program Counter trace block which helps keep track of PC discontinuity/ jumps, which can in turn help track the complete sequence of software that got executed at any given point of time. The module only tracks the discontinuity in the instruction fetches/execution (non-sequential), the sequential code execution can be easily reconstructed using software. Once a trace is completed/stopped, the trace data can be read out using the debugger to reconstruct the code execution sequence. There are multiple trace modes to control when to enable tracing and when to disable tracing based on events generated by ERAD EBC events or some critical system level events.

Trace data is stored in a ram space which can be read at any time with additional status information on trace validity that can be used for reconstructing the code execution sequence. For every discontinuity, two PC values are stored, that is, the source of discontinuity and the destination.

Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from November 7, 2024 to March 31, 2025 (from Revision * (November 2024) to Revision A (March 2025))	Page
• Removed design-specific information.	5

This page intentionally left blank.

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265