



**AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE**

Procesory Sygnałowe w aplikacjach przemysłowych

**Tryb SIMD, przerwania, konfiguracja nóżek
układu**

**IET
Katedra Elektroniki
Kraków 2015
dr inż. Roman Rumian**



Alternate (Secondary) DAG Registers

To facilitate fast context switching, the processor includes alternate register sets for all DAG registers. Bits in the MODE1 register control when alternate registers become accessible. While inaccessible, the contents of alternate registers are not affected by processor operations. Note that there is a one cycle latency between writing to MODE1 and being able to access an alternate register set. The alternate register sets for the DAGs are described in this section. For more information on alternate data and results registers, see “Alternate (Secondary) Data Registers” on page 2-14.

Bits in the MODE1 register can activate alternate register sets within the DAGs: the lower half of DAG1 (I, M, L, B0–3), the upper half of DAG1 (I, M, L, B4–7), the lower half of DAG2 (I, M, L, B8–11), and the upper half of DAG2 (I, M, L, B12–15). Figure 6-8 shows the primary and alternate register sets of the DAGs.

Example 1

```
BIT SET MODE1 SRD1L; /* Activate alternate dag1 lo regs */  
NOP; /* Wait for access to alternates */  
R0 = DM(i0,m1);
```

Example 2

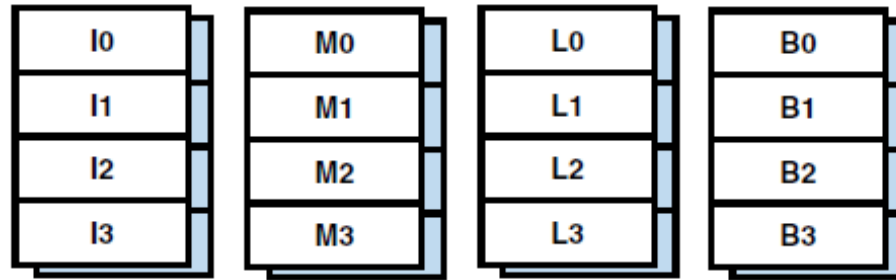
```
BIT SET MODE1 SRD1L; /*activate alternate dag1 lo registers */  
R13 = R12 + R11; /* Any unrelated instruction */  
R0 = DM(I0,M1);
```



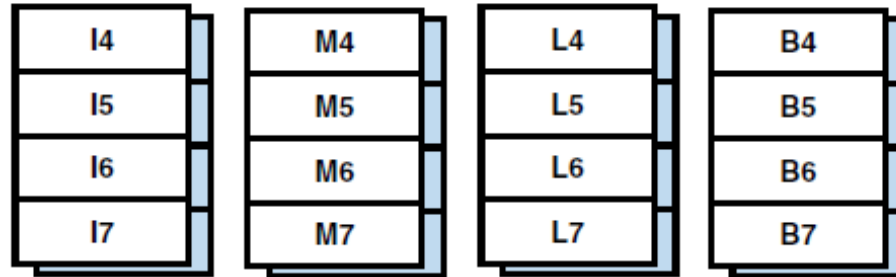
MODE1 SELECT BIT

DAG1 REGISTERS

SRD1L →



SRD1H →

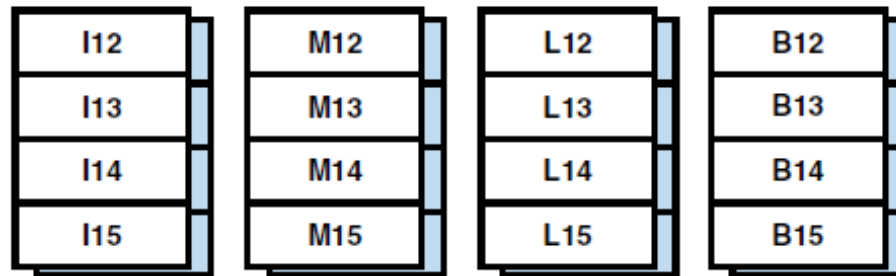


DAG2 REGISTERS

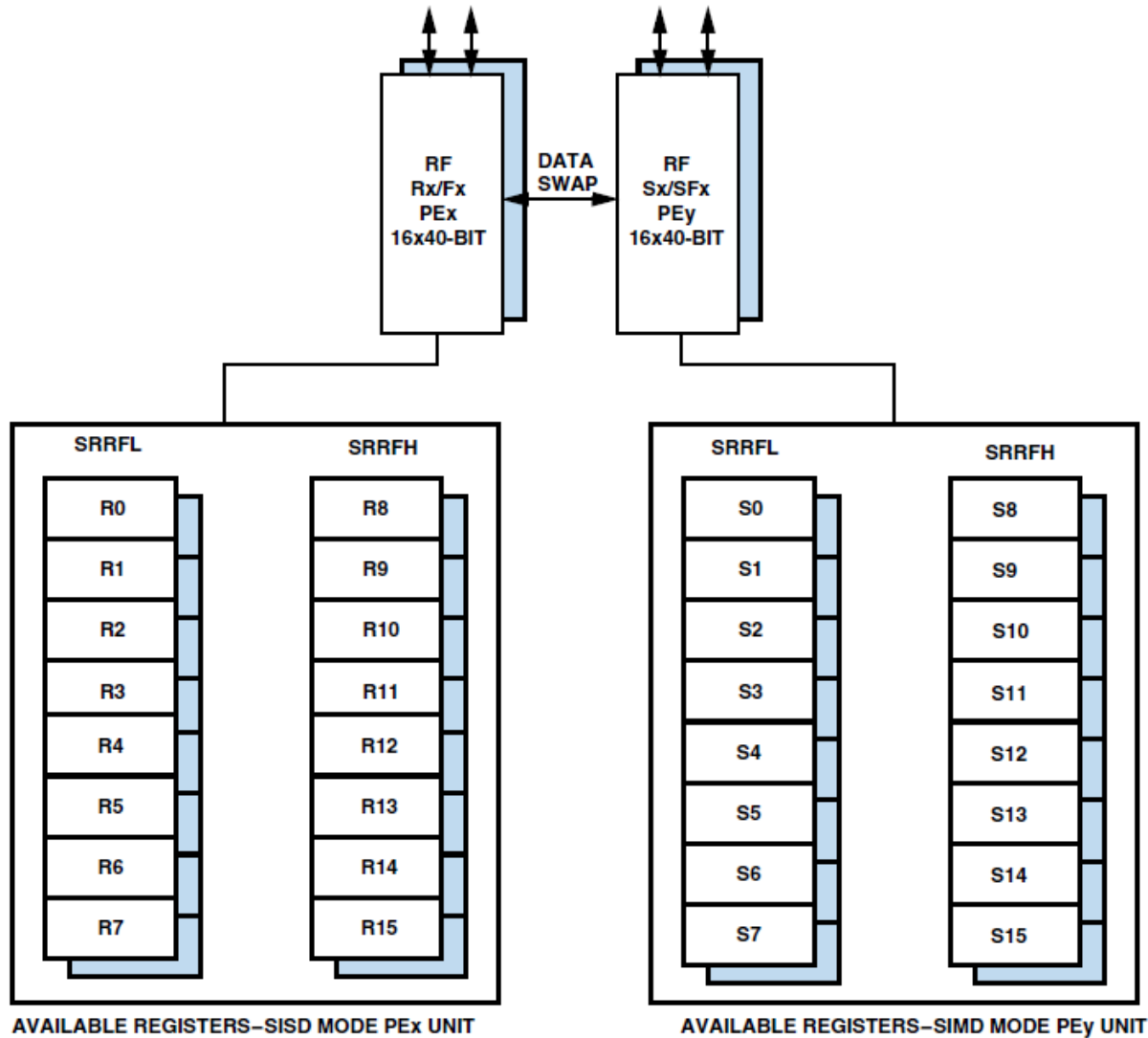
SRD2L →



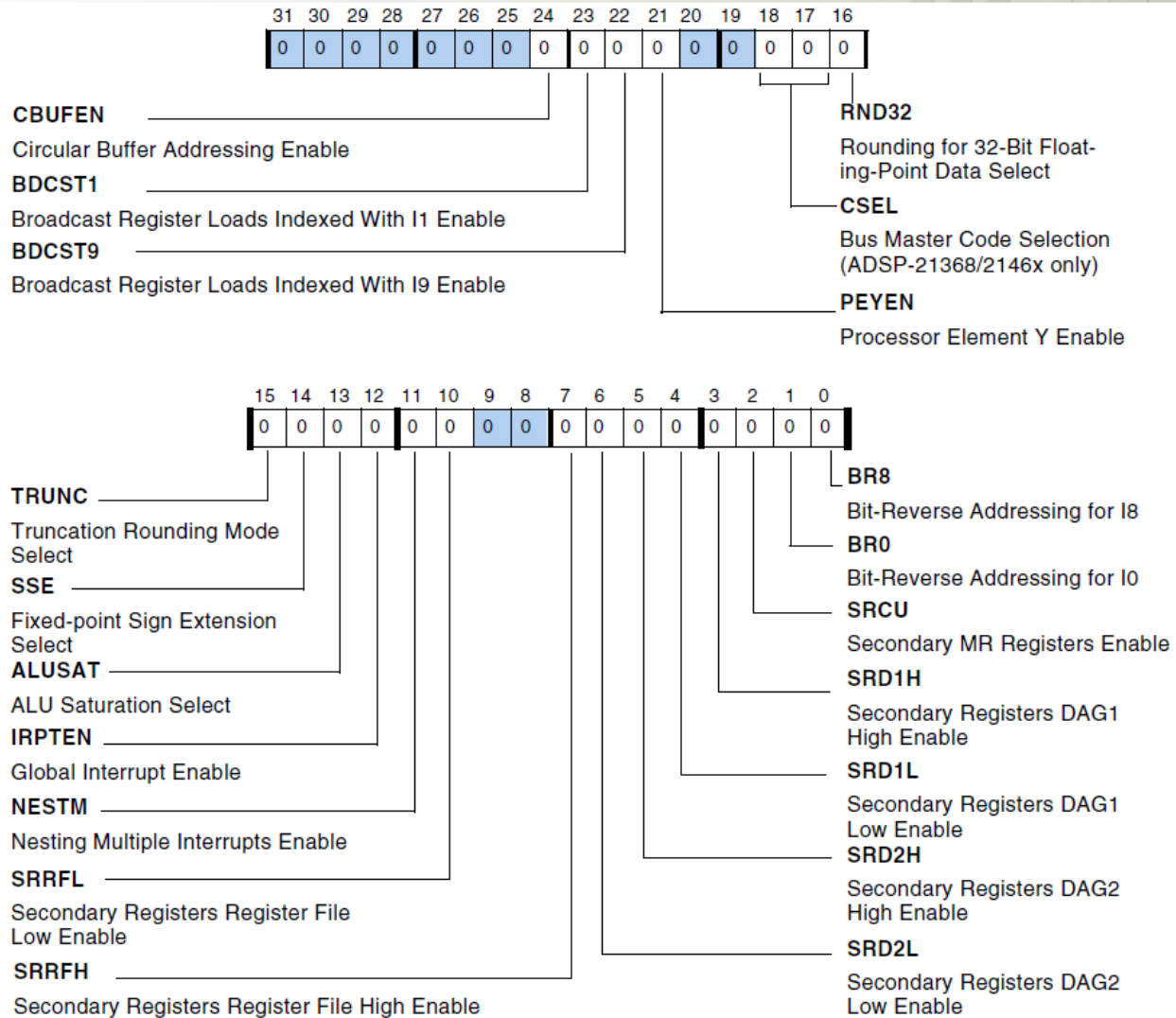
SRD2H →



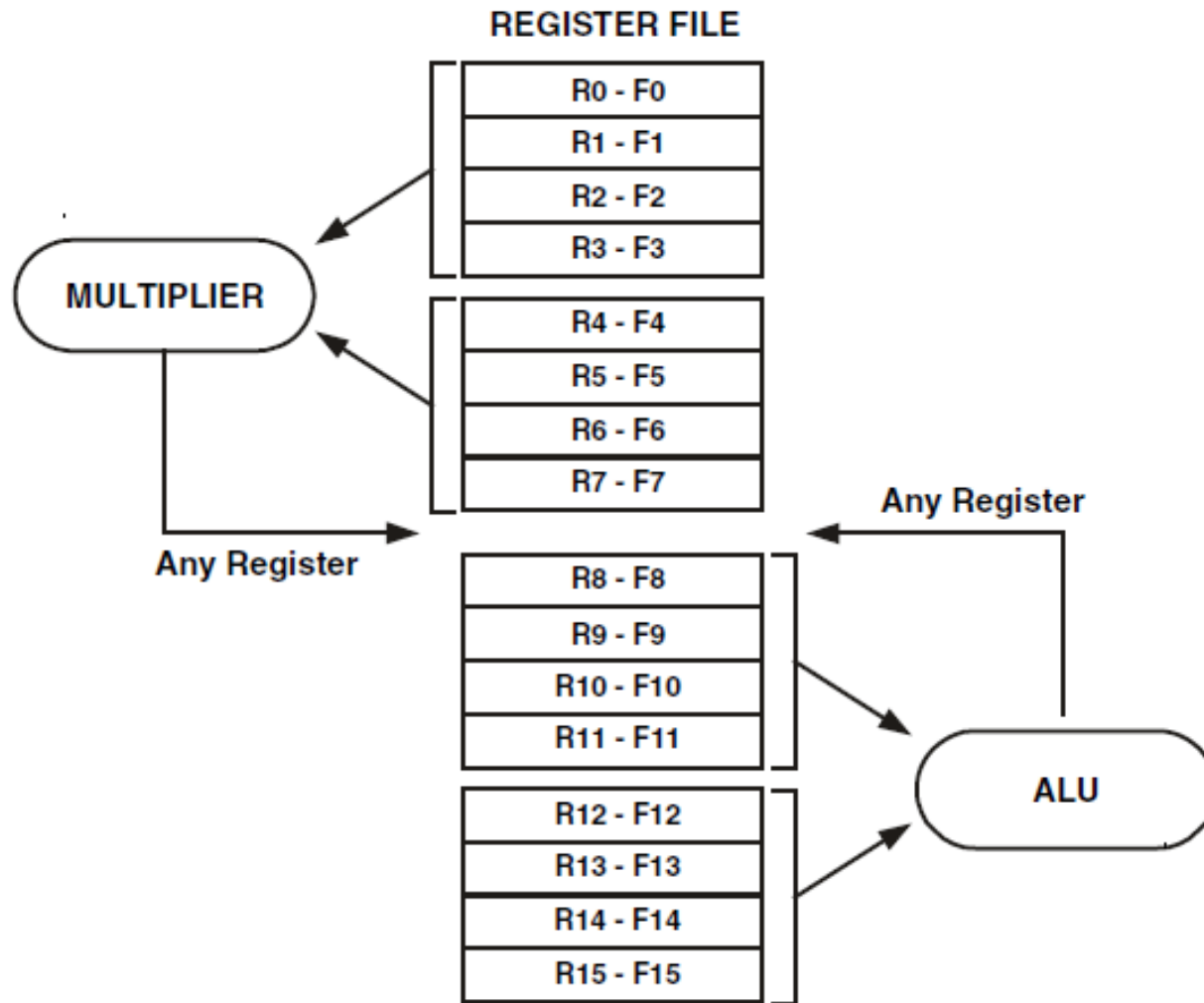
Alternate (Secondary) Data Register File



Mode Control 1 Register



Permitted Input Registers for Multifunction Computations





Compute Instructions in SIMD Mode

```
bit set MODE1 PEYEN;          /* enable SIMD */
nop;                          /* effect latency */

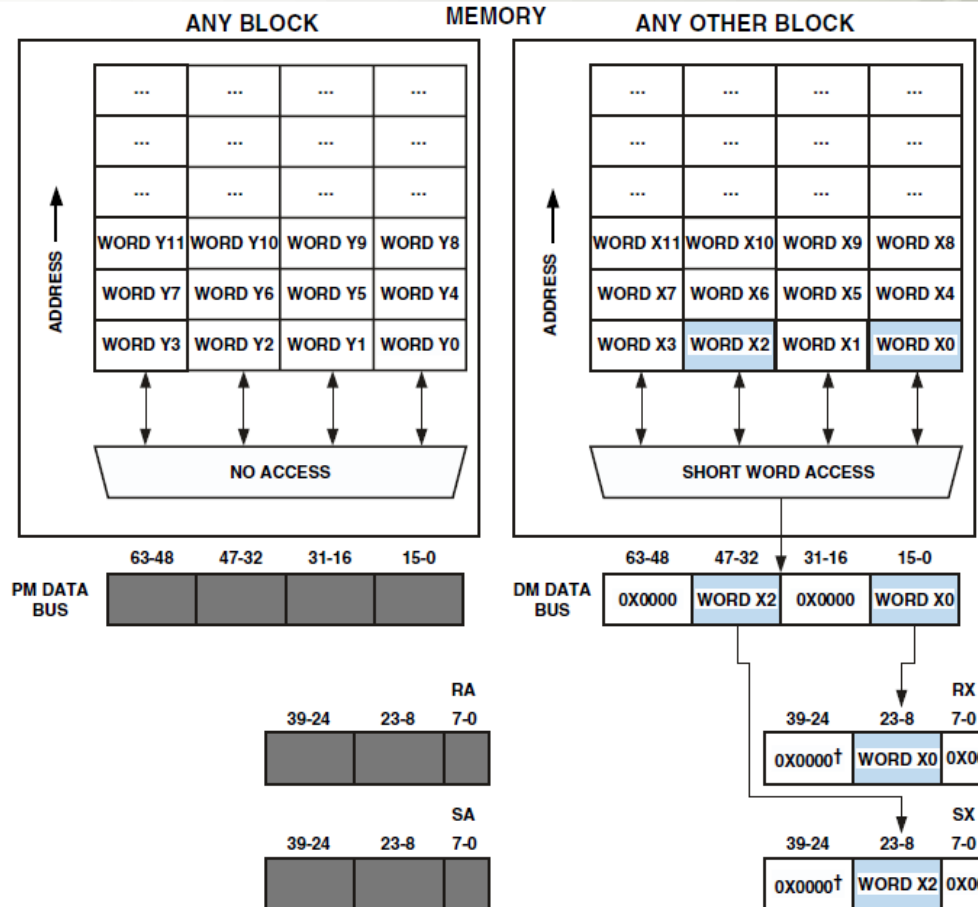
R0 = R1 + R2;                 /* explicit ALU instruction */
S0 = S1 + S2;                 /* implicit ALU instruction */

F0 = F1 * F2;                 /* explicit MUL instruction */
SF0 = SF1 * SF2;             /* implicit MUL instruction */

MRB = MRB - R3 * R2 (SSFR); /* explicit MUL instruction */
MSB = MSB - S3 * S2 (SSFR); /* implicit MUL instruction */

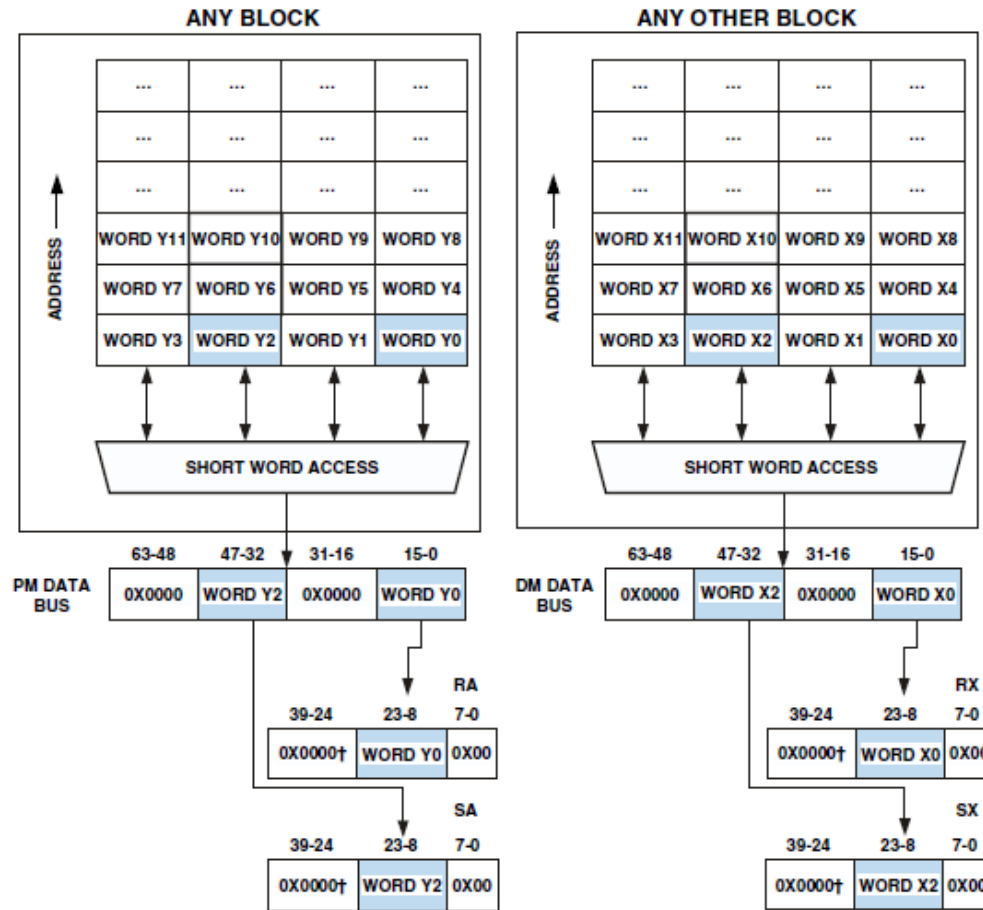
R5 = LSHIFT R6 by <data8>; /* explicit shift imm instruction */
S5 = LSHIFT S6 by <data8>; /* implicit shift imm instruction */
```

Short Word Addressing of Single-Data in SIMD Mode



THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
 RX = DM(SHORT WORD X0 ADDRESS);
 OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SIMD, SHORT WORD, SINGLE-DATA TRANSFERS ARE:
 UREG = PM(SHORT WORD ADDRESS);
 UREG = DM(SHORT WORD ADDRESS);
 PM(SHORT WORD ADDRESS) = UREG;
 DM(SHORT WORD ADDRESS) = UREG;

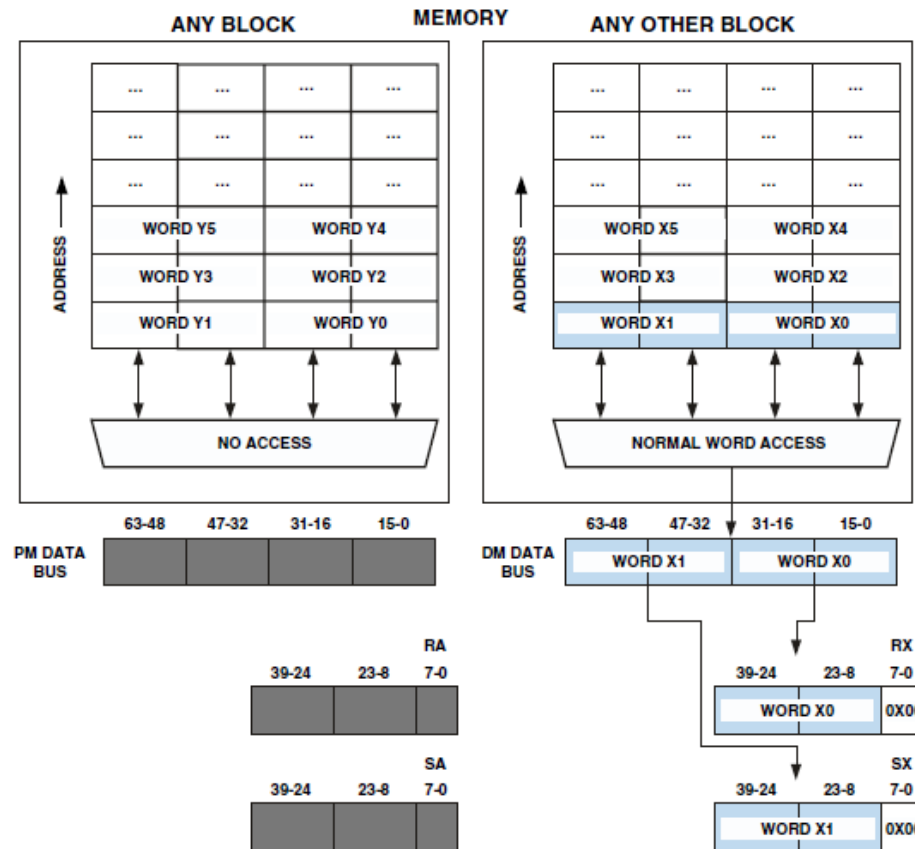
Short Word Addressing of Single-Data in SIMD Mode



THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
 RX = DM (SHORT WORD X0 ADDRESS), RA = PM (SHORT WORD Y0 ADDRESS);

OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SIMD, SHORT WORD, DUAL-DATA TRANSFERS ARE:
 DREG = PM(SHORT WORD ADDRESS), DREG = DM(SHORT WORD ADDRESS);
 PM(SHORT WORD ADDRESS) = DREG, DM(SHORT WORD ADDRESS) = DREG;

Normal Word Addressing of Single-Data in SIMD Mode



THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
 $RX = DM(NORMAL\ WORD\ X0\ ADDRESS);$

OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SIMD, NORMAL WORD, SINGLE-DATA TRANSFERS ARE:

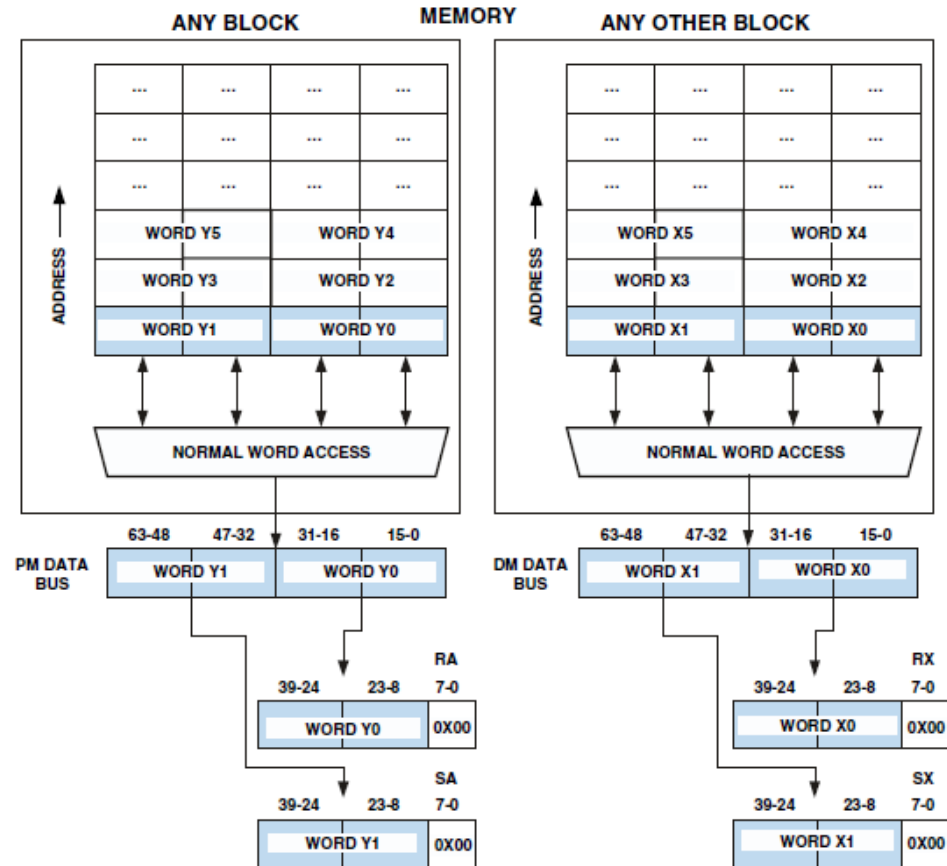
```

| UREG = PM(NORMAL WORD ADDRESS);
| UREG = DM(NORMAL WORD ADDRESS);
| PM(NORMAL WORD ADDRESS) = UREG;
| DM(NORMAL WORD ADDRESS) = UREG;

```



Normal Word Addressing of Dual-Data in SIMD Mode



THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
 RX = DM(NORMAL WORD X0 ADDRESS), RA = PM(NORMAL WORD Y0 ADDRESS);

OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SIMD, NORMAL WORD,
 DUAL-DATA TRANSFERS ARE:

DREG = PM(NORMAL WORD ADDRESS), DREG = DM(NORMAL WORD ADDRESS);
 PM(NORMAL WORD ADDRESS) = DREG, DM(NORMAL WORD ADDRESS) = DREG;

MAC and Parallel Read With Software Pipeline Coding

```

MRF=0, R5 = DM(I1,M2), R6 = PM(I9,M9); /* first data */
Lcntr=N-1, do (pc,1) unti lce;
MRF = MRF-R5*R6, R5 = DM(I1,M2), R6 = PM(I9,M9); /* loop body */
MRF = MRF-R5*R6; /* last MAC*/

```

IIR Biquad Stage

```

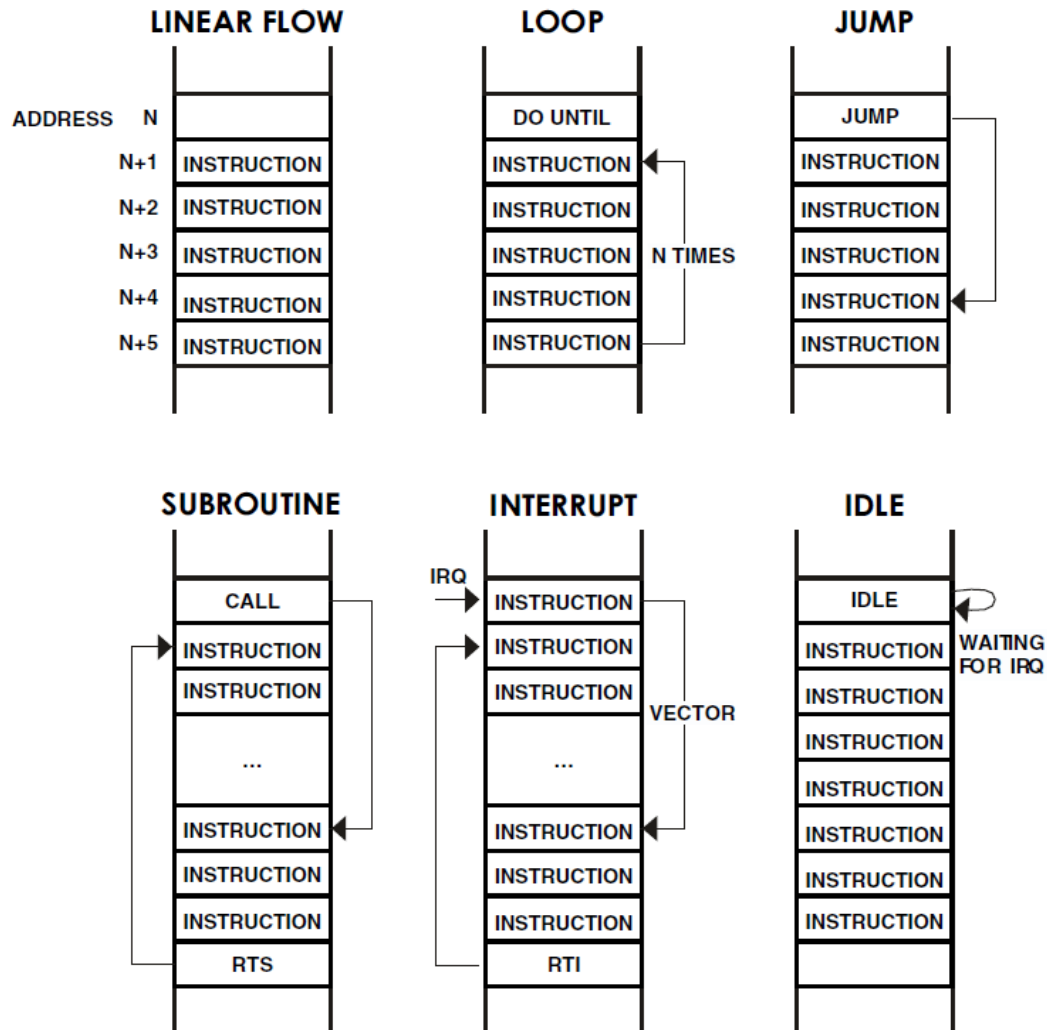
B1=B0;
F12=F12-F12, F2 = DM(I0,M1), F4 = PM(I8,M8); /* first data */
Lcntr=N, do (pc,4) until lce; /* loop body */
F12=F2*F4, F8=F8+F12, F3 = DM(I0,M1), F4 = PM(I8,M8);
F12=F3*F4, F8=F8+F12, DM(I1,M1)=F3, F4 = PM(I8,M8);
F12=F2*F4, F8=F8+F12, F2 = DM(I0,M1), F4 = PM(I8,M8);
F12=F3*F4, F8=F8+F12, DM(I1,M1)=F8, F4 = PM(I8,M8);
RTS (db), F8=F8+F12, /* last MAC */
Nop;
Nop;

```



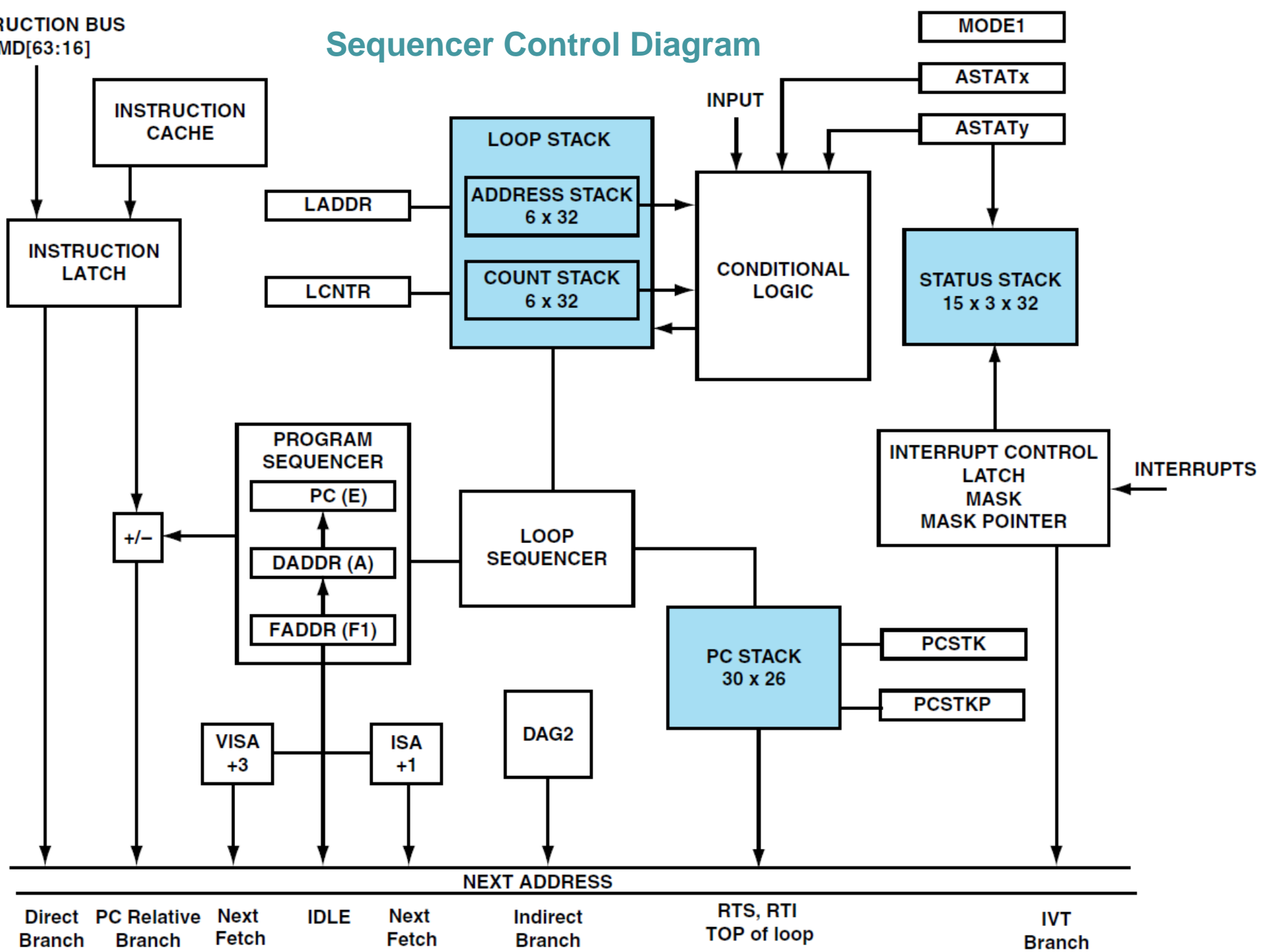
The sequencer controls the following operations.

- **Loops.** One sequence of instructions executes several times with zero overhead.
- **Subroutines.** The processor temporarily breaks sequential flow to execute instructions from another part of program memory.
- **Jumps.** Program flow is permanently transferred to another part of program memory.
- **Interrupts.** Subroutines in which a runtime event (not an instruction) triggers the execution of the routine.
- **Idle.** An instruction that causes the processor to cease operations and hold its current state until an interrupt occurs. Then, the processor services the interrupt and continues normal execution.



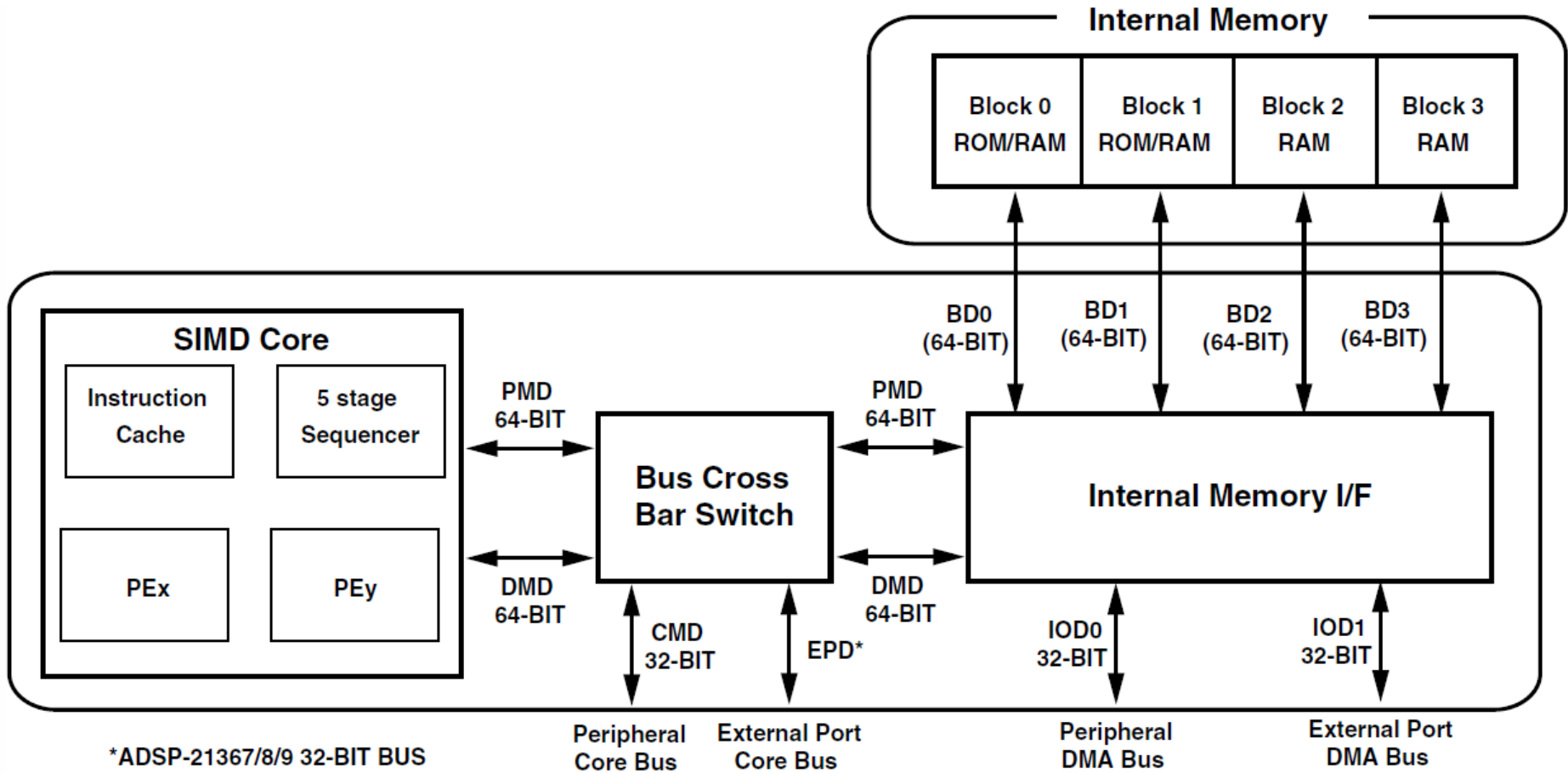
INSTRUCTION BUS
PMD[63:16]

Sequencer Control Diagram





Memory and Internal Buses Block Diagram



*ADSP-21367/8/9 32-BIT BUS
ADSP-21371/5 48-BIT BUS
ADSP-214xx 64-BIT BUS

Instruction Pipeline Processing Stages

Stage	ISA	VISA Extension
Fetch1	In this stage, the appropriate instruction address is chosen from various sources and driven out to memory. The instruction address is matched with the cache to generate a condition for cache miss/hit. The next NW address is auto incremented by one.	Next SW address is auto incremented by three for every 48-bit fetch
Fetch2	This stage is the data phase of the instruction fetch memory access wherein the data address generator (DAG) performs some amount of pre-decode. Based on a cache condition, the instruction is read from cache/driven from the memory instruction data bus.	Stores 3 x 16-bit instruction data into the IAB buffer and presents 1 instruction/cycle to the decoder
Decode	The instruction is decoded and various conditions that control instruction execution are generated. The main active units in this stage are the DAGs, which generate the addresses for various types of functions like data accesses (load/store) and indirect branches. DAG pre-modify (M+I) operation is performed. For a cache miss, instruction data read from memory are loaded into the cache.	Decode VISA instruction; store its length information in short words.
Address	The addresses generated by the DAGs in the previous stage are driven to the memory through memory interface logic. The addresses for the branch operation are made available to the fetch unit. For instruction branches (Call/Jump) the address is forward to the Fetch1 stage. For a do until instruction the next address is fetched.	
Execute	The operations specified in the instruction are executed and the results written back to memory or the universal registers. For interrupt branch the IVT address is forward to the Fetch1 stage. ISA instructions always increment PC value by 1 each cycle.	Executing VISA instructions the PC value is incremented by 1, 2 or 3 depending on length information from the Instruction decode.



Hardware Stacks

- Program count stack – Used to store the return address (call, IVT branch, do until).
- Status stack – Used to store some context of status registers.
- “Loop Stack” for address and count – Used for hardware looping (unnested and nested).

Attribute	PC Stack	Loop Address Stack	Loop Count Stack	Status Stack
Stack Size	30 x 26 bits	6 x 32 bits	6 x 32 bits	15 x 3 x 32 bits
Top Entry	Return Address	Loop End Address	Loop iteration count	MODE1 ASTATx/ASTATy
Empty Flag	PCEM	LSEM		SSEM
Full Flag	PCFL	LSOV		SSOV
Stack Pointer	PCSTKP	No		No
Exception IRQ	SOVFI	SOVFI		SOVFI
Automated Access				
Push Condition	CALL, IVT branch DO UNTIL	DO UNTIL		IVT Branch (Timer, $\overline{TRQ2-0}$ only)
Pop Condition	RTS, RTI	CURLCNTR = 1 or COND = true		RTI (Timer, $\overline{TRQ2-0}$ only)
Manual Access				
Register Access	PCSTK	LADDR	CURLCNTR	No
Explicit Push	Push PCSTK	Push Loop		Push STS
Explicit Pop	Pop PCSTK	Pop Loop		Pop STS



Interrupts

Interrupt Branch Mode

Interrupts are a special case of subroutines triggered by an event at runtime and are also another type of nonsequential program flow that the sequencer supports. Interrupts may stem from a variety of conditions, both internal and external to the processor. In response to an interrupt, the sequencer processes a subroutine call to a predefined address, called the interrupt vector. The processor assigns a unique vector to each type of interrupt and assigns a priority to each interrupt based on the Interrupt Vector Table (IVT) addressing scheme.

The interrupt controller **is enabled by setting the global IRPTEN bit in the MODE1 register**. The processor supports three prioritized, individually- maskable external interrupts, each of which can be programmed to be either **level- or edge-triggered**. External interrupts occur when an external device asserts one of the processor's interrupt inputs (IRQ2–0). The processor also supports internally generated interrupts. An internal interrupt can occur due to arithmetic exceptions, stack overflows, DMA completion and/or peripheral data buffer status, or circular data buffer overflows. Several factors control the processor's response to an interrupt. When an interrupt occurs, the interrupt is synchronized and latched in the interrupt latch register (IRPTL). The processor responds to an interrupt request if:

- The processor is executing instructions or is in an idle state
- The interrupt is not masked
- Interrupts are globally enabled
- A higher priority request is not pending

When the processor responds to an interrupt, the **sequencer branches the program execution with a call to the corresponding interrupt vector address**. Within the processor's program memory, the interrupt vectors are grouped in an area called the **interrupt vector table (IVT)**. The interrupt vectors in this table **are spaced at 4-instruction intervals**. Longer service routines can be accommodated by branching to another region of memory.

Program execution returns to normal sequencing when the return from interrupt (RTI) instruction is executed. Each interrupt vector has associated latch and mask bits.

Interrupt Categories

- ❖ Non maskable interrupts (RESET/emulator/boot peripheral)
- ❖ Maskable interrupts (core/IO)
- ❖ Software interrupts (core)

The processor responds to interrupts in three stages:

1. Synchronization (1 cycle)
2. Latching and recognition (1 cycle)
3. Branching to the interrupt vector table (4 instruction cycles)

The following example uses delayed branches to reduce latency.

```
ISR_IRQ2:      rti;
               rti;
               rti;
               rti;
ISR_IRQ1:      instruction; /* IVT branch address */
               jump ISR (db);
               instruction;
               instruction;
ISR_IRQ0:      rti;
               rti;
               rti;
               rti;
```



Interrupt Processing Stages

The processor also has extensive programmable interrupt support. These interrupts are described in the processor-specific hardware references.

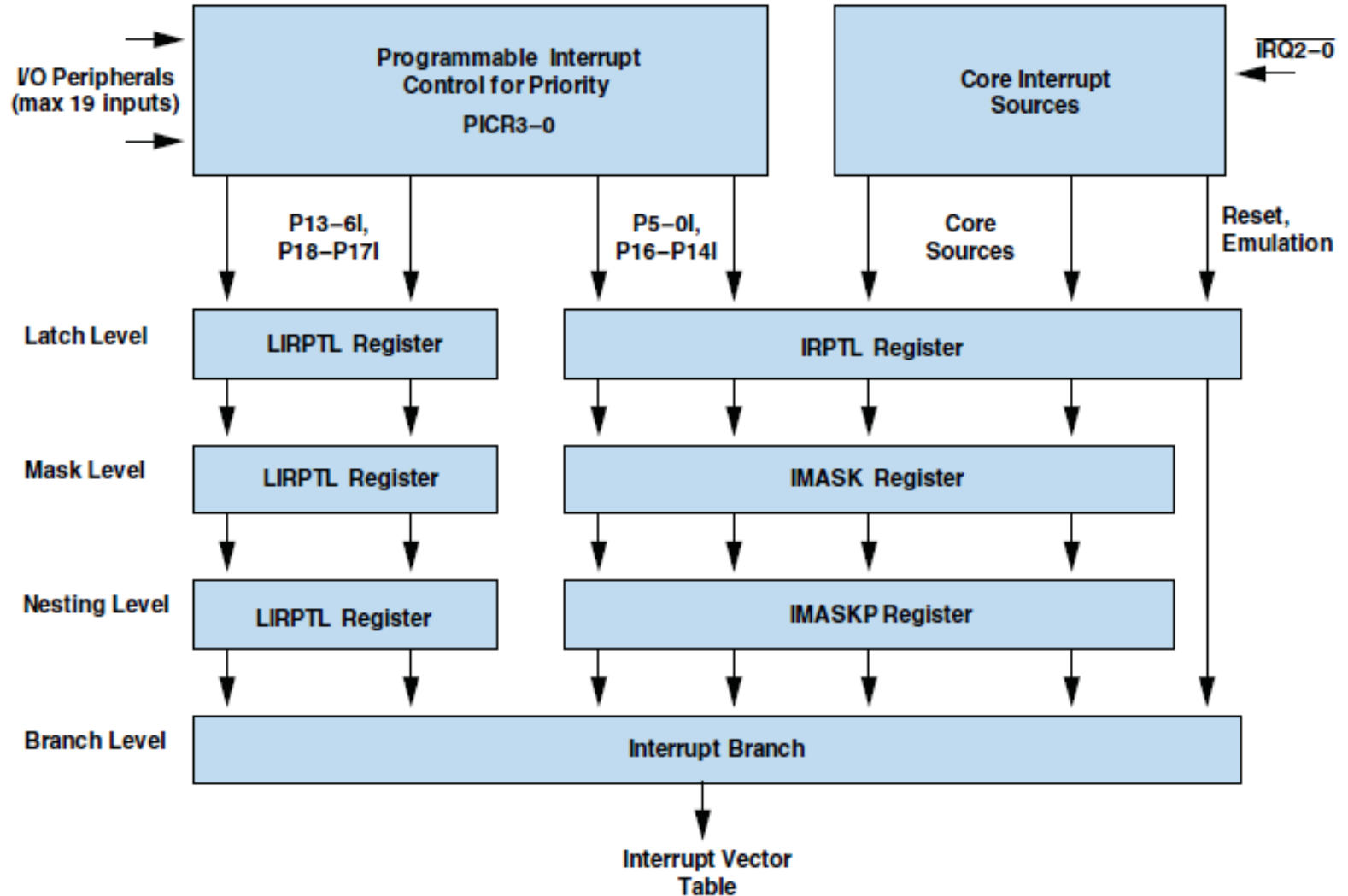
To process an interrupt, the program sequencer:

1. Outputs the appropriate interrupt vector address.
2. Pushes the current PC value (the return address) onto the PC stack.
3. Automatically pushes the current value of the $ASTAT_{x/y}$ and $MODE1$ registers onto the status stack (only if the interrupt is from $IRQ2-0$ or the timer).
4. Resets the appropriate bit in the interrupt latch register ($IRPTL$ and $LIRPTL$ registers).
5. Alters the interrupt mask pointer bits ($IMASKP$ register) to reflect the current interrupt nesting state, depending on the nesting mode. The $NESTM$ bit in the $MODE1$ register determines whether all the interrupts or only the lower priority interrupts are masked during the service routine.

At the end of the interrupt service routine, the sequencer processes the RTI instruction and performs the following sequence.

1. Returns to the address stored at the top of the PC stack.
2. Pops this value off the PC stack.
3. Automatically pops the status stack (only if the $ASTAT_{x,y}$ and $MODE1$ status registers were pushed for the $IRQ2-0$, or timer interrupt).
4. Clears the appropriate bit in the interrupt mask pointer register ($IMASKP$).

Interrupt Process Flow





Core Interrupt Sources

According to the IVT table the core supports different groups of interrupts such as:

- Reset – hardware/software
- emulator – debugger, breakpoints, BTC
- core timer – high, low priority
- illegal memory access – forced long word, illegal IOP space
- stack exceptions – PC, Loop, Status
- IRQ2-0 – hardware inputs
- DAGs – Circular buffer wrap around
- Arithmetic exceptions – fixed-point, floating-point
- Software interrupts – programmed exceptions

Note that the interrupt priorities of the core are fixed and cannot be changed.

The interrupt latch bits in the IRPTL register correspond to interrupt mask bits in the IMASK register. (In the LIRPTL register both mask and latch bits are present). In both registers, the interrupt bits are arranged in order of priority. The interrupt priority is from 0 (highest) up to 41 (lowest).

Interrupt priority determines which interrupt must be serviced first, when more than one interrupt occurs in the same cycle. Priority also determines which interrupts are nested when the processor has interrupt nesting enabled.

Programmable Interrupt Priorities for Peripherals

Peripheral interrupts can be routed to a set of programmable interrupts (18–0). This increases the flexibility across different I/O DMA channels and priorities. For more details see the processor-specific hardware reference manual.



Interrupt Mask Mode

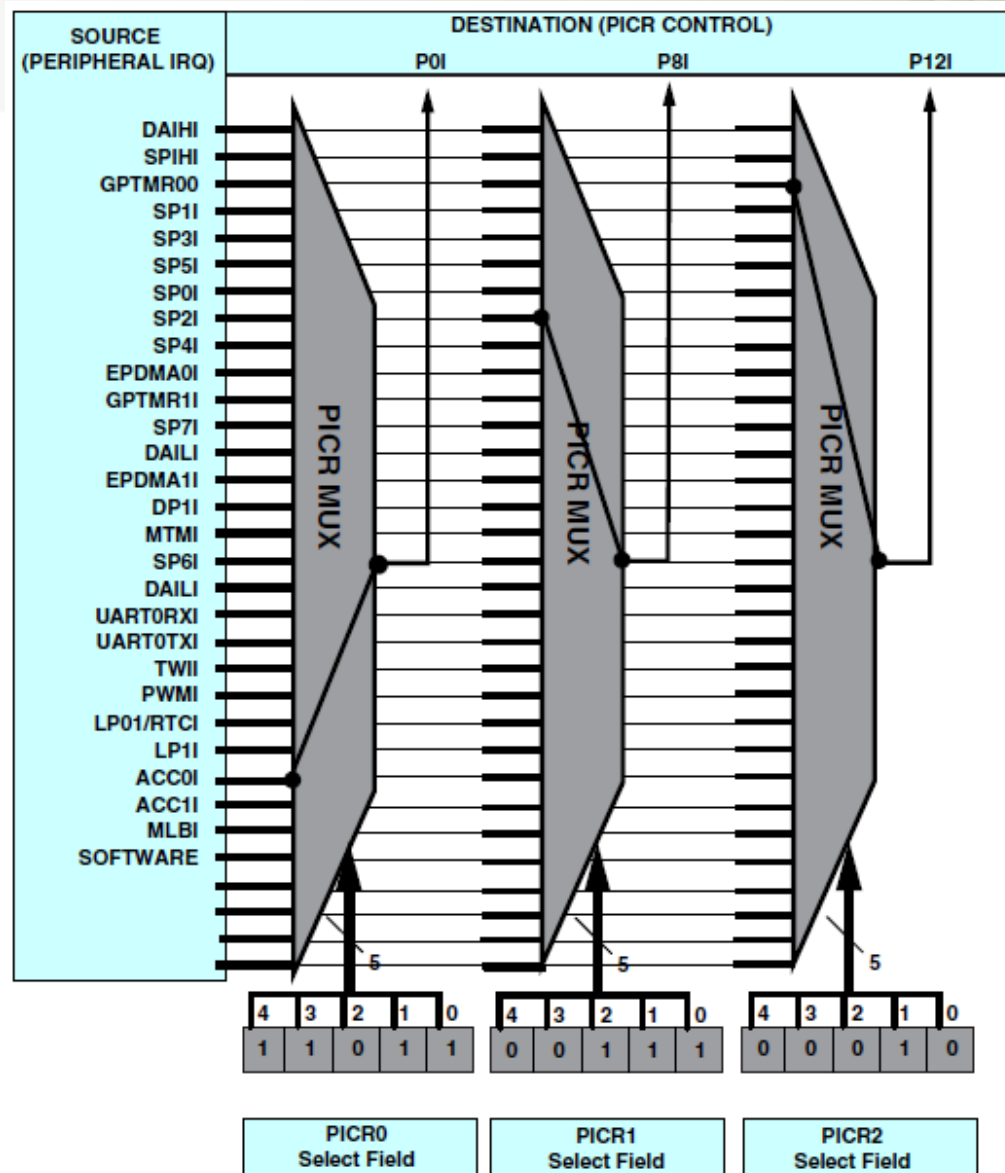
Bits that are set in the MMASK register are used to clear bits in the MODE1 register when the processor's status stack is pushed. This effectively disables different modes when servicing an interrupt, or when executing a PUSH STS instruction. The processor's status stack is pushed in two cases:

1. When executing a PUSH STS instruction explicitly in code.
2. When an IRQ2–0 or timer expired interrupt occurs.

Programmable Interrupt Priority Control

The processor core supports 19 programmable prioritized interrupts, which are shown in an example routing. The highest priority interrupt is P0I while the lowest priority is P18I. Any peripheral interrupt output may be connected to any programmable priority interrupt input. All peripheral interrupt output signals are considered as source signals. The 19 prioritized peripheral interrupts (P0I–P18I) of the core are considered destination interrupts. The PICR register controls the connectivity between the source and destination. The interrupt output of every peripheral can be programmed to connect to any one of the 19 peripheral interrupts. Moreover, the peripherals are grouped in two broad categories—DAI or DPI, each having its own interrupt controller. These interrupt controllers program the polarity, priority and the destination of each peripheral interrupt output. Therefore, all peripheral interrupts can also be connected to the core as DAI or DPI interrupts.

Programmable Prioritized Interrupts



The **IRPTL** register indicates latch status for interrupts.

Interrupt Mask Register (IMASK) Each bit in the IMASK register corresponds to a bit with the same name in the IRPTL registers. The bits in the IMASK register unmask (enable if set, =1), or mask (disable if cleared, = 0) the interrupts that are latched in the IRPTL register. Except for the RSTI and EMUI bits, all interrupts are maskable.

Interrupt Mask Pointer Register (IMASKP)

When interrupt nesting is enabled, the bits in the IMASKP register mask interrupts that have a lower priority than the interrupt that is currently being serviced. Other bits in this register unmask interrupts having higher priority than the interrupt that is currently being serviced. Interrupt nesting is enabled using NESTM in the MODE1 register.

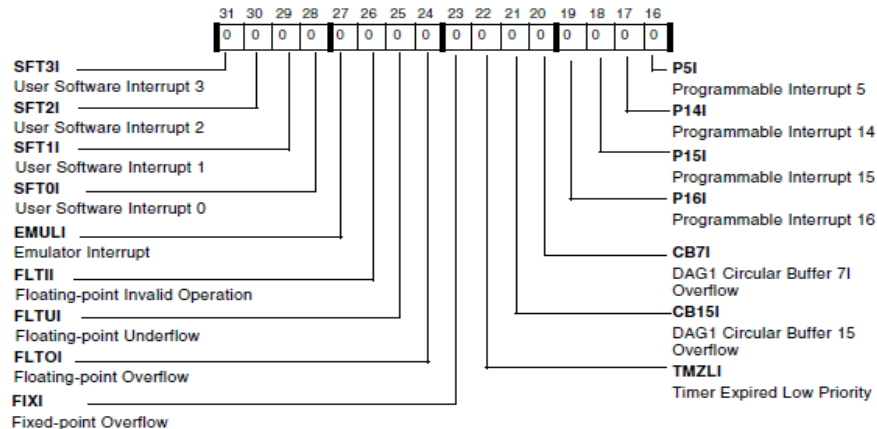


Figure A-6. IRPTL, IMASK, and IMASKP Registers (Bits 31–16)

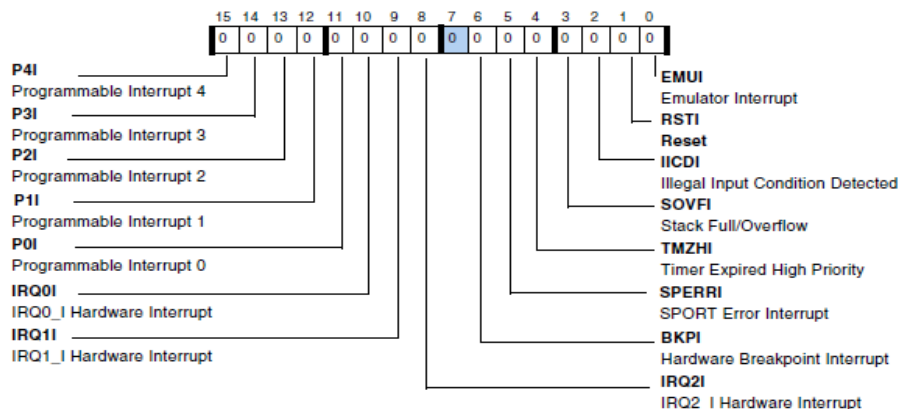


Figure A-7. IRPTL, IMASK, and IMASKP Registers (Bits 15–0)

The LIRPTL register indicates latch status, select masking, and displays mask pointers for interrupts.

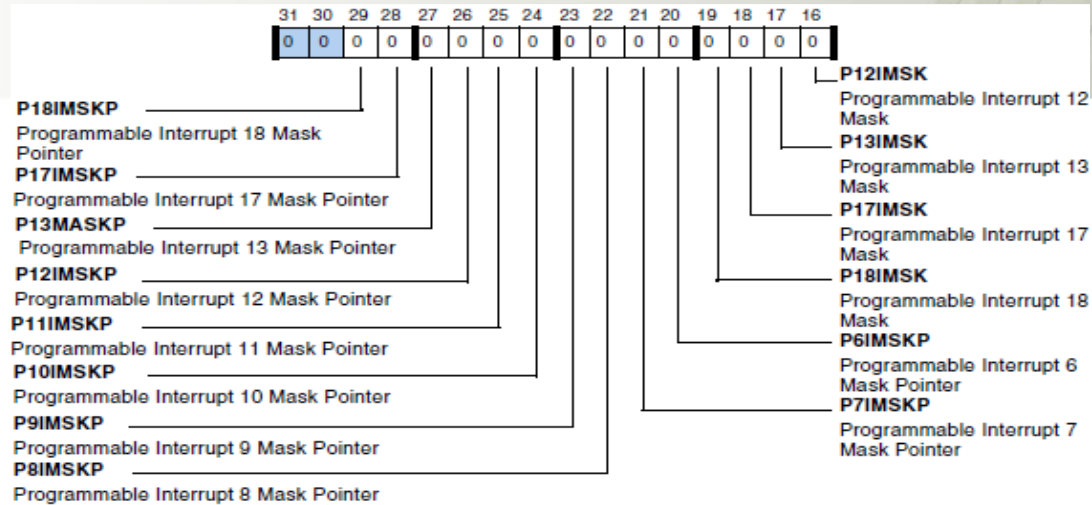


Figure A-8. LIRPTL Register (Bits 31–16)

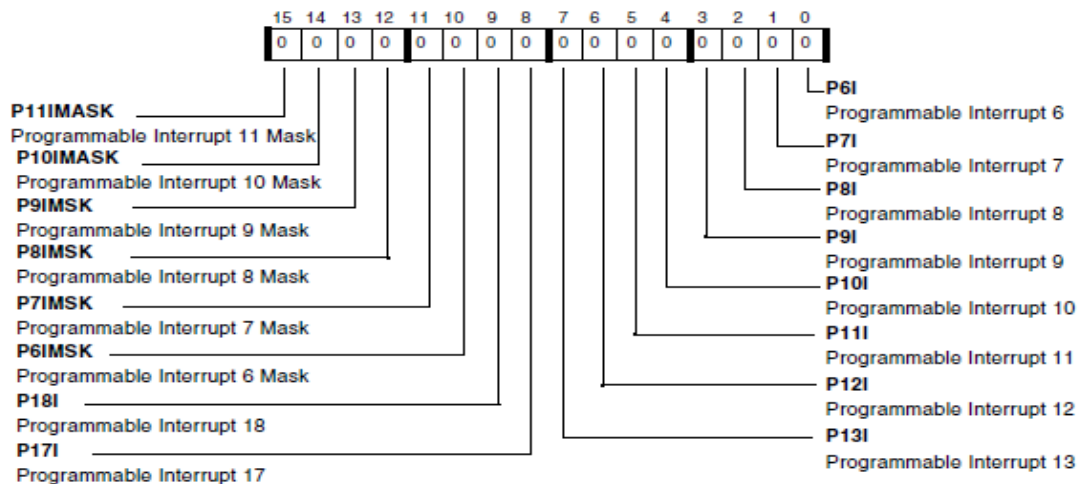
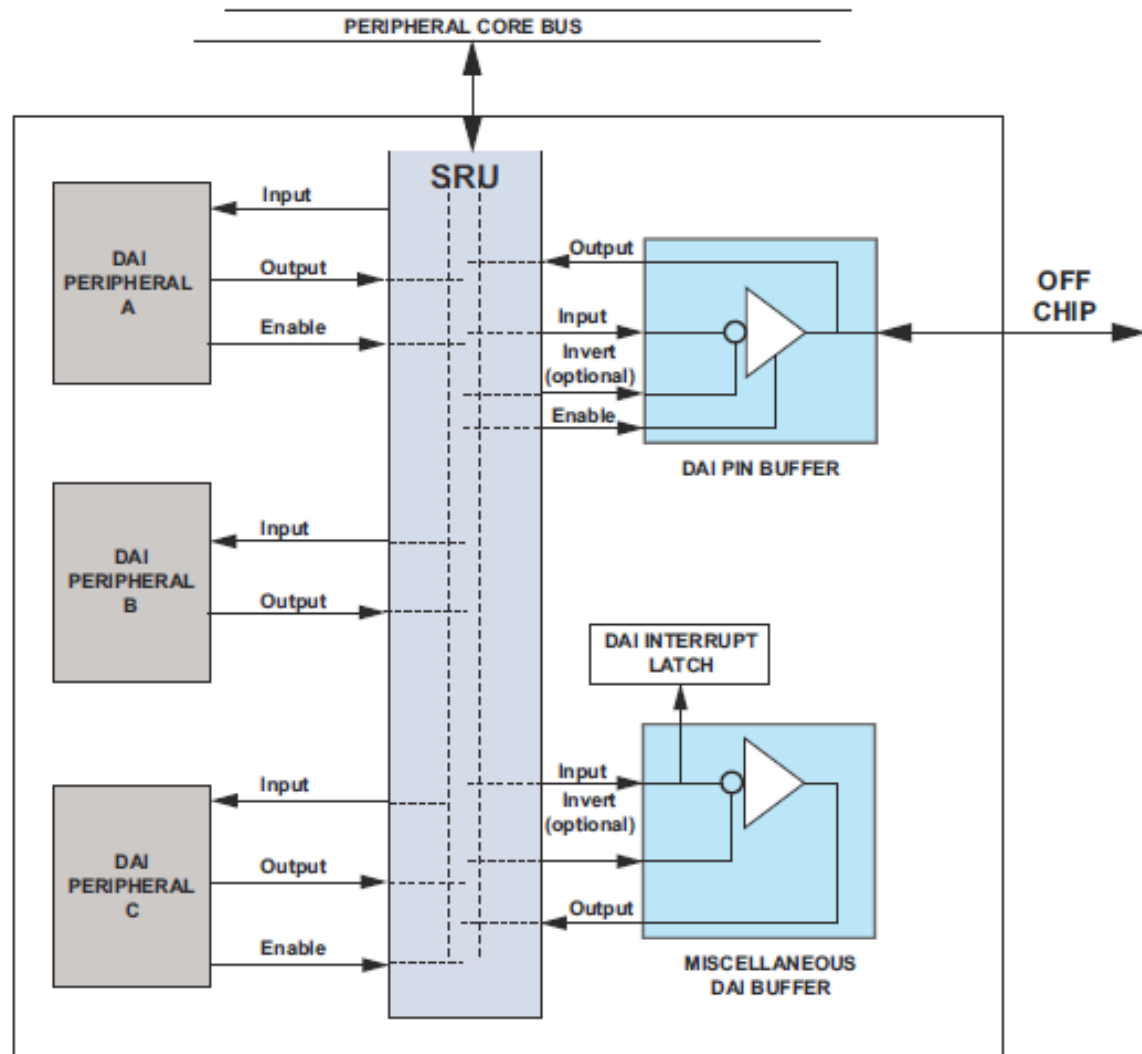
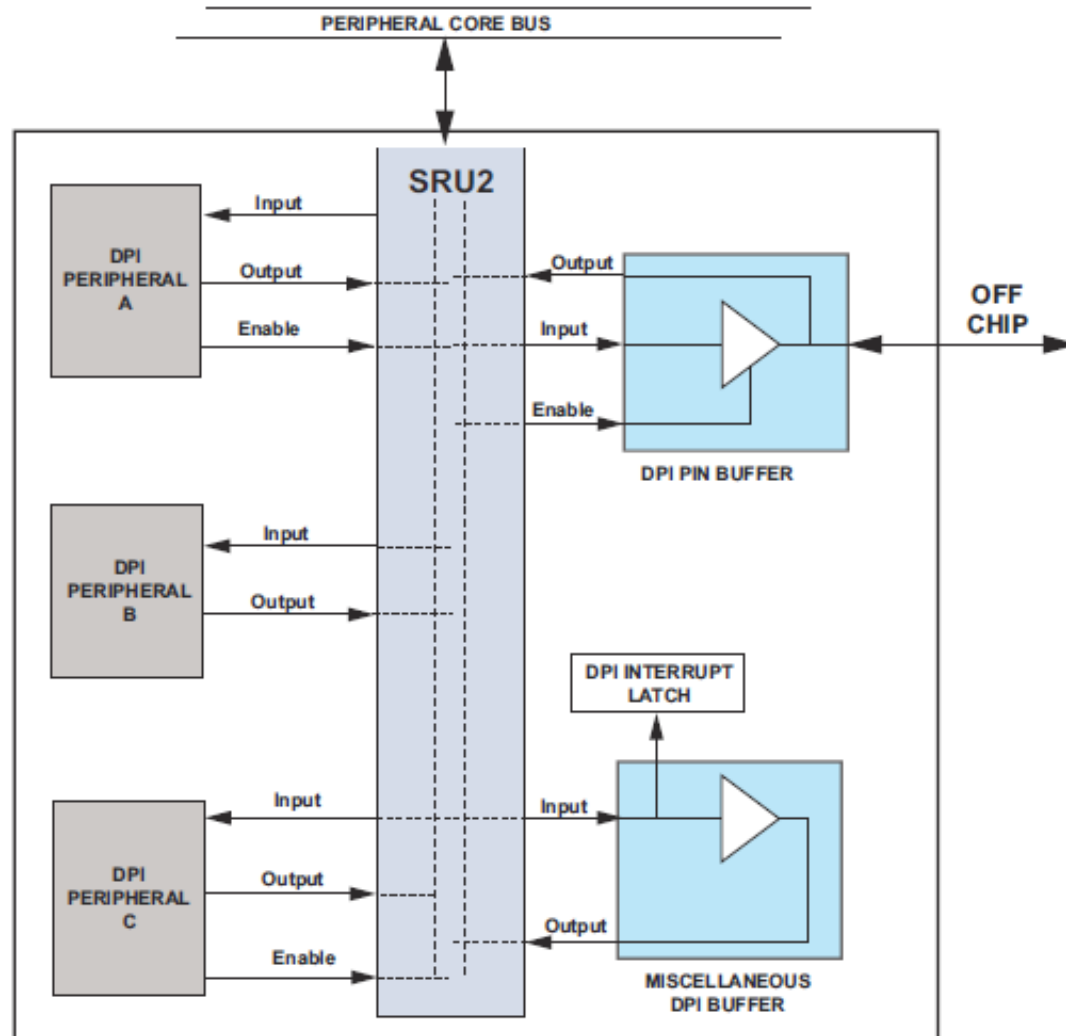


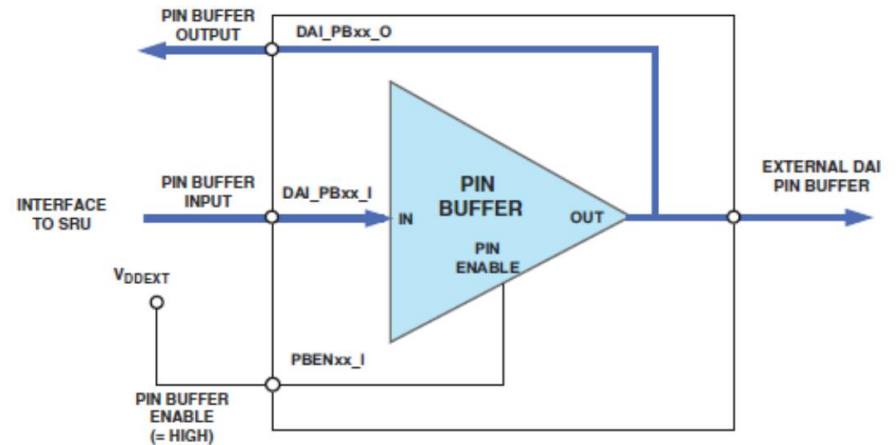
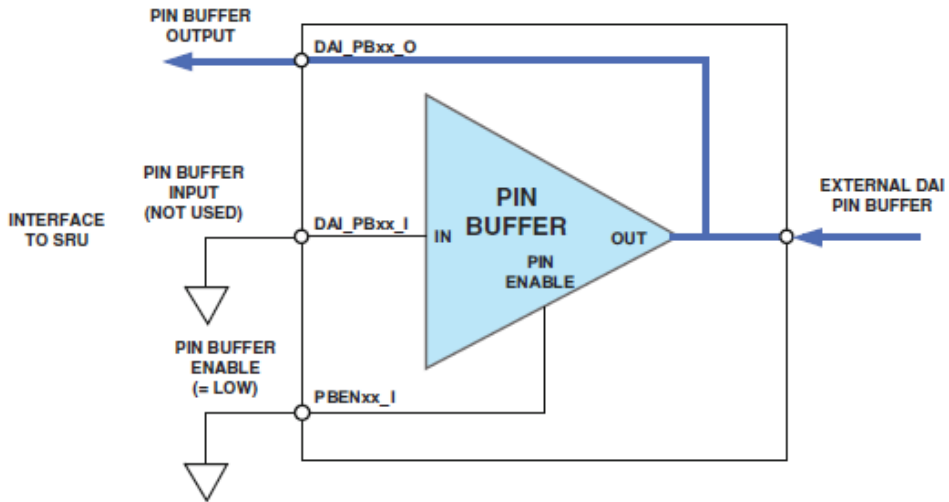
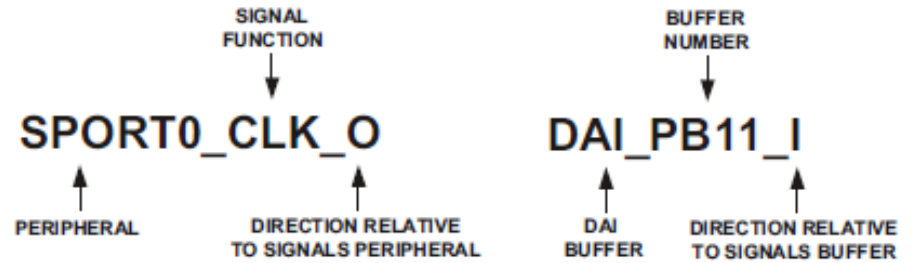
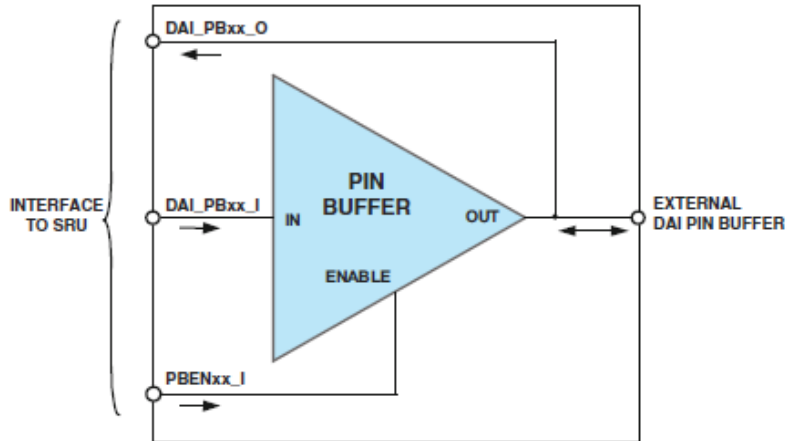
Figure A-9. LIRPTL Register (Bits 15–0)

DAI Functional Block Diagram



DPI Functional Block Diagram





Example DAI SRU Group A Multiplexing (SRU_CLKx)

