

# 8 PROGRAM FLOW INSTRUCTIONS

The instruction set provides program flow instructions for controlling the sequence in which the DSP executes instructions. Generally, the instructions in a program execute sequentially, one after another, unless otherwise directed by various program structures—branches, loops, subroutines, or interrupts—that intervene and temporarily or permanently redirect this linear flow. These program structures enable an application to respond to events or conditions as they occur. Program flow control instructions include:

- “DO UNTIL (PC relative)” on page 8-22
- “Direct JUMP (PC relative)” on page 8-27
- “CALL (PC relative)” on page 8-30
- “JUMP (PC relative)” on page 8-34
- “Long CALL” on page 8-37
- “Long JUMP” on page 8-40
- “Indirect CALL” on page 8-42
- “Indirect JUMP” on page 8-45
- “Return from Interrupt” on page 8-48
- “Return from Subroutine” on page 8-52
- “PUSH or POP Stacks” on page 8-55
- “PUSH or POP Stacks” on page 8-55

## Program Flow Instructions

- “FLUSH CACHE” on page 8-61
- “Set Interrupt” on page 8-62
- “Clear Interrupt” on page 8-64
- “No Operation” on page 8-66
- “Idle” on page 8-67
- “Mode Control” on page 8-69

This chapter describes each of the move instructions and the following related topics:

- “Conditions” on page 8-2
- “Counter-Based Conditions” on page 8-3
- “CCODE Register” on page 8-4
- “MSTAT Mode Control Register” on page 8-4
- “Branch Options” on page 8-5
- “Addressing Branch Targets” on page 8-6
- “Stacks” on page 8-7
- “Stack Status Flags” on page 8-12
- “Interrupts” on page 8-13
- “Application Performance” on page 8-17

## Conditions

Table 2-8 on page 2-15 lists the conditions used in conditional (IF COND) instructions and their opcodes. Besides these conditions (which mainly relate to the status of the ALU, multiplier, and counter) it is possible to

use the `SWCOND` condition and the value in the `CCODE` register to test for other DSP status conditions. For more information, see [“Condition Code \(CCODE\) Register” on page 2-6](#). Also, you can test for bit states to generate conditions using the `TSTBIT` instruction. For more information, see [“Bit Manipulation: TSTBIT, SETBIT, CLRBIT, TGLBIT” on page 3-18.](#),

## Counter-Based Conditions

Both `IF Condition` (conditional) instructions and the `DO UNTIL` (loop) instructions can base execution on the `NOT CE` condition. Although the `DO UNTIL` instruction uses the `CE` syntax only, the condition actually tested is `NOT CE`—counter not expired.

To use a counter condition with either instruction type, you must load the `CNTR` register with an initial counter value ( $>1$ ) before issuing the instruction that uses the counter condition.

There are some important differences between how conditional and loop instructions implement (decrement and test) the counter condition:

- To implement the `NOT CE` condition in an `IF Condition` (conditional) instruction, the DSP decrements and tests the value loaded in the `CNTR` register before executing the conditional instruction. For a conditional instruction based on `NOT CE`, the DSP tests whether the `CNTR` register contains a value  $>1$ .
- To implement the `CE` condition in a `DO UNTIL` (loop) instruction, the DSP loads the loop counter stack from the `CNTR` register at the start of the loop, then decrements and tests the counter value in the loop counter stack (not the `CNTR` register) at the end of each pass through the loop. For a loop instruction based on `CE`, the DSP tests whether the loop counter stack’s counter value  $>0$ . For more information, see [“Loop Stacks Operation” on page 8-10](#).

### CCODE Register

[Table 2-3 on page 2-7](#) lists the CCODE register values used to test the SWCOND and NOT SWCOND software conditions. Although the source each value tests is specific to each DSP in the ADSP-219x family, these values (except 0x08 and 0x09) map to the software interrupt bits in the IMASK and IRPTL registers.

To test for any software condition, first load the CCODE register with the value of the source you want to test, then test for the true or false state. For example, 0x08 represents ALU saturation status, you might code this sequence:

```
CCODE = 0x08;           /* ALU Saturated (AR_SAT) cond */
AR = AX0 + AY1;
IF SWCOND JUMP fix_data; /* Jump to fix_data if AR_SAT */

fix_data:
  NOP;                 /* code to fix data ALU_SAT */
```

Or, to test for a shifter overflowed result:

```
CCODE = 0x09;           /* Shifter Overflowed (SV) cond */
AR = 3; SE = AR;       /* shift code, left shift 3 bits */
SI = 0xB6A3;           /* value of hi word of input data */
IF NOT SWCOND SR = ASHIFT SI (HI); /* ashift high word if SV */
```

A value written to CCODE isn't available on the next cycle, so you must insert at least one instruction between the write to CCODE and the conditional instruction that tests the software condition. Otherwise, the conditional instruction will test the previous value of CCODE.

### MSTAT Mode Control Register

As shown in [Table 2-6 on page 2-11](#), bits 0 through 7 of the MSTAT register control various DSP modes. These modes determine some conditions for how status flags are set.

## Branch Options

All of the DSP's branch instructions (except `DO UNTIL` and `LJUMP/LCALL`), support two branch options: immediate and delayed. These options determine whether the DSP executes the first two instructions directly following the branch instruction before it executes the instruction at the branch target address. Because of the instruction pipeline, a number of latency cycles (usually four) occur between execution of the branch instruction and execution of the branch target instruction.

By default, the branch instructions perform an immediate branch, which means that the next instruction the DSP executes after the branch instruction is the instruction at the branch target address, but only after a number of `NOP` cycles. Using the delayed option, you can salvage two of the `NOP` cycles and perform useful work. To do so, you include the `(DB)` option in the branch instruction and code in the two delay slots directly following the branch instruction the two instructions that you want executed before the branch target instruction.

- ⊘ You cannot insert `JUMP` or `CALL` instructions in delay branch slots. You can insert only one two-word instruction, and it must occupy the first delay branch slot.

When the DSP executes an `RTI` or `RTS` instruction to return to the main program, it returns to execute the first or third instruction after the branch instruction, depending on whether the branch is immediate or delayed.

Return from immediate branch:

```
IF AV CALL immediate_pump; /* immed branches may be cond */
NOP;                       /* RTS returns program flow here */
NOP;

immediate_pump:
NOP;
RTS;
```

## Program Flow Instructions

Return from delayed branch:

```
CALL delayed_pump (DB); /* delayed branches must be uncond */
NOP;                    /* 1st_delay_instruction */
NOP;                    /* 2nd_delay_instruction */
NOP;                    /* RTS returns program flow here */
NOP;

delayed_pump:
NOP;
RTS;
```

## Addressing Branch Targets

When you issue a `JUMP` or `CALL` instruction, you specify the address of the instruction to branch to in one of three ways:

- **PC relative**  
Offset from the current PC. The immediate value you specify in the instruction is added to the PC of the branch instruction to form the address of the branch target location. For example, the `CALL` in the following code goes to PC-relative address (`find_me`):

```
.EXTERN find_me; /* matches .global in other file */
CALL find_me (DB);
NOP;
```

- **Far absolute**  
The full 24-bit address of the branch target location is specified in the instruction. You can program this instruction explicitly in an `LJUMP/LCALL` instruction. The assembler automatically substitutes this instruction when the actual address assembled from a PC relative address is insufficient.

- **Indirect**  
The address of the branch target location is specified using a DAG index register (I0–I7) and the IJPG page register. For example, the CALL in the following code goes to an address using the indirect address from the I0 register:

```
.EXTERN find_me_too; /* matches .global in other file */
IJPG = 0x0;          /* set memory page for CALL */
I0 = find_me_too;   /* loads I0 with address */
NOP;
NOP;
CALL (I0) (DB);
NOP;
```

## Stacks

Loops and other branch instructions use the DSP's stacks to implement their respective operations.

- **PC stack (33 words × 24 bits)**  
Holds the address of the next instruction to execute on return from a called subroutine. Only the CALL, RTI, RTS, and PUSH/POP PC instructions use this stack.
- **Loop begin stack (8 words × 24 bits)**  
Holds the address of the first instruction in a loop. Only the DO UNTL and PUSH/POP LOOP instructions use this stack.
- **Loop end stack (8 words × 24 bits)**  
Holds the address of the last instruction in a loop. Only the DO UNTL and PUSH/POP LOOP instructions use this stack.
- **Loop counter stack (8 words × 16 bits)**  
Holds the current value of the loop counter that is loaded from the CNTR register. This value—not the value in the CNTR register—is

## Program Flow Instructions

tested and decremented at the end of each pass through the loop. Only the `DO UNTL` and `PUSH/POP LOOP` instructions use this stack.

- Status stack (16 words  $\times$  32 bits)

Holds the current value of the `ASTAT` and `MSTAT` registers. Only `RTI` and `PUSH/POP STS` instructions use this stack. (When globally enabled and unmasked interrupts occur, the DSP automatically saves the two status registers to this stack.)

## PC and Status Stack Operation

Applications use these stacks to implement function calls and interrupt service routines (ISRs).

**Function Calls.** When a `CALL` instruction executes, it automatically pushes onto the PC stack the address of the next instruction to execute upon returning from the subroutine. The `CALL` instruction does not push the status registers onto the status stack.

The `RTS` instruction, executed at the end of the subroutine, returns program execution to either the first or third instruction following the `CALL` instruction, depending on whether the `CALL` was immediate or delayed, respectively.

**ISRs.** When interrupts are globally enabled and an unmasked interrupt occurs, it cause the DSP to automatically save its current state before entering the interrupt's ISR. To do so, the DSP:

- Pushes onto the PC stack the address of the next instruction to execute upon returning from the ISR.

If the interrupt is higher than the core's current level of operation, the DSP pushes the address of the current instruction onto the PC stack and branches immediately to the interrupt's ISR.

If the interrupt is lower than the core's current level of operation, the DSP finishes the current operation, pushes the address of the next sequential instruction onto the PC stack, and then branches to the interrupt's ISR.

- Pushes onto the status stack, in order, the `ASTAT` and `MSTAT` registers.

The `RTI` instruction, executed at the end of the ISR, pops the PC stack returning program execution to the instruction at the retrieved address. It also pops the status stack, restoring the `ASTAT` and `MSTAT` registers to their previous values. So, if the ISR enables any of the `MSTAT` mode bits, the `RTI` operation automatically disables them.

**PUSH/POP PC/STS.** You can explicitly push and pop the PC and status stacks as needed. The DSP automatically performs these operations when using nested interrupts.

Pushing (`PUSH STS`) and popping (`POP STS`) the status stack automatically saves or restores the `ASTAT` and `MSTAT` registers. But, pushing (`PUSH PC`) and popping (`POP PC`) the PC stack requires a few more steps that involve the `STACKA` and `STACKP` registers.

For `PUSH/POP PC` operations, the 16-bit `STACKA` register supplies or receives, respectively, the sixteen LSBs of an instruction's 24-bit address, and the 8-bit `STACKP` register supplies or receives the eight MSBs. So, before you


## Program Flow Instructions

issue a `PUSH PC` instruction, you must load the `STACKA` and `STACKP` registers with the appropriate values:

```
STACKA = 0x3521;  
STACKP = 0x02;  
PUSH PC;
```

Likewise, after you pop the PC stack, you can check the contents of the `STACKA` and `STACKP` registers:

```
POP PC;  
AX0 = STACKA;  
AY0 = STACKP;
```

 A `PUSH` or `POP PC` has one cycle of latency for all `SSTAT` register bits, but a `PUSH` or `POP LOOP` or `STS` has one cycle of latency only for the `STKOVERFLOW` bit in the `SSTAT` register.

## Loop Stacks Operation

Applications use this stack to implement loop operations.

When a `DO UNTIL` instruction executes, it automatically pushes data onto the three loop stacks:

- **Loop begin stack** Receives the loop start address (current PC).
- **Loop end stack** Receives the loop end address.
- **Counter stack** Receives the counter value from the `CNTR` register for finite loops. If the loop is infinite, the counter stack still receives the counter value; the DSP decrements this value on the stack, but ignores the result.

**Finite Loops** (`DO <loop> UNTIL CE`). The `CE` terminator specifies a finite loop. To accommodate the write effect latency, you must load the `CNTR` register with the number of iterations to execute the loop at least two instructions before the `DO UNTIL` instruction. When the `DO UNTIL` instruc-

tion executes, it automatically pushes the `CNTR` value onto the counter stack. (The `CNTR` register retains the original value, until explicitly changed with a data move or `POP LOOP` instruction.)

The loop mechanism decrements and tests the value at the top of the counter stack at the end of each pass through the loop. The loop ends when the counter expires (decrements to 1). At loop end, program execution automatically continues with the instruction directly following the end of the loop.

**Infinite Loops** (`DO <loop> [UNTIL FOREVER]`). To end an infinite loop, the loop must contain an explicit `JUMP` to an instruction outside the loop to exit and end it. The `JUMP` is based on a condition created inside the loop and typically branches to a `POP LOOP` instruction to recover the loop stacks—the goal is to adjust the loop stack pointers, not retrieve the loop start and end addresses. After that, another `JUMP` instruction returns program execution to the next sequential instruction following the loop's end.

**PUSH/POP LOOP.** You can explicitly push and pop the loop stacks. These operations are necessary to recover and maintain the loop stacks when you abort a loop.

`PUSH/POP LOOP` instructions operate on all three loop stacks in parallel. Both operations involve the `STACKA`, `STACKP`, `LPSTACKA`, `LPSTACKP`, and `CNTR` registers. Before you issue a `PUSH LOOP` instruction, you must load the `STACKA`, `STACKP`, `LPSTACKA`, and `LPSTACKP` registers with appropriate values.

- The 16-bit `STACKA` and 8-bit `STACKP` register supply or receive the loop start address from the loop begin stack.

`STACKA` holds the sixteen LSBs of the 24-bit, loop start address, and `STACKP` holds the eight MSBs.

## Program Flow Instructions

- The 16-bit `LPSTACKA` and 16-bit `LPSTACKP` register supply or receive the loop end address from the loop end stack. (Only bit 15 and bits 7:0 of `LPSTACKP` are valid—bits 14:8 should always be zero.)

`LPSTACKA` holds the sixteen LSBs of the 24-bit, loop end address, and `LPSTACKP` holds the eight MSBs in bits 7:0 and the loop terminator condition, `CE` or `FOREVER`, in bit 15. When the `FOREVER` bit is set, the loop logic ignores the loop counter value.

- The 16-bit `CNTR` register supplies or receives the counter value from the counter stack.

On a `pop`, the `CNTR` register receives whatever value is at the top of the counter stack. For finite loops, since the value in the counter stack is decremented at the end of each pass through the loop, a `POP` loads `CNTR` with a new value, overwriting the original count value, unless the `POP` occurs before the first pass through the loop.

For infinite loops, the `PUSH LOOP` instruction pushes the current value of the `CNTR` register onto the loop counter stack. This value is irrelevant but pushing it maintains the pointer's correct position in the loop counter stack.



A `PUSH` or `POP PC` has one cycle of latency for all `SSTAT` register bits, but a `PUSH` or `POP LOOP` or `STS` has one cycle of latency only for the `STKOVERFLOW` bit in the `SSTAT` register.

## Stack Status Flags


As shown in [Table 2-7 on page 2-14](#), bits 0 through 7 of the `SSTAT` register record the status of the DSP's stacks. This status information is useful for managing the stack and servicing stack interrupts.

The stack interrupt is always generated by a stack overflow condition, but can also be generated by ORing together the stack overflow status (`STK-`

OVERFLOW) bit and stack high/low level status (PCSTKLVL) bit. The level bit is set when:

- The PC stack is pushed and the resulting level is at the high water mark.
- The PC stack is popped and the resulting level is at the low water-mark.

This spill-fill mode (using the stack to generate a stack interrupt) is disabled on reset. Two bits in the ICNTL register (bit 10 —PC Stack Interrupt Enable) can be used to enable interrupts for the three corresponding stacks.

 When switching on spill-fill mode, a spurious low water mark interrupt may occur (depending on the level of the stack). In this case, the interrupt handler should push some values on the stack to raise the level above the low watermark level.

## Interrupts

The DSP uses interrupts to communicate with the outside world. The DSP's core generates internal interrupts, the peripherals generate external interrupts, and software can generate software interrupts.

When an interrupt occurs, the DSP suspends its current operation, saving the `ASTAT` and `MSTAT` registers, and jumps to the location in memory of the interrupt's service routine (ISR) and begins executing that program code. When it has completed the interrupt's ISR, an `RTI` instruction at the end of the routine forces program flow to return to the suspended operation and continue executing code at the location where it left off, after the DSP restores the `ASTAT` and `MSTAT` registers.

Each interrupt has a priority rank and its own vector address. The interrupt's vector address specifies the location in memory of its service routine. Its priority determines the order in which the interrupt gets ser-

## Program Flow Instructions

vised relative to the other interrupts. An interrupt with higher priority gets serviced before one with lower priority. As shown in [Table 2-5 on page 2-10](#), the lower the interrupt's position in the `IMASK/IRPTL` register the higher its priority.

To implement and use interrupts, your software must perform these tasks:

- Globally enable interrupts.
- Individually enable the particular interrupt.
- At the beginning of the ISR, switch context to secondary register sets and perform the necessary tasks to handle the interrupt condition. For details, see [“Switching Contexts” on page 8-16](#).

If your program requires nested interrupts, it might need to perform a few extra tasks within each interrupt's ISR. For details, see [“Nesting Interrupts” on page 8-16](#).

- At the end of the ISR, insert an `RTI` instruction to branch back (`RTI`) to the suspended operation. The `RTI` instruction automatically switches context back to the primary register sets.
- Continue executing program code at the return address.

### Enabling Interrupts

When an interrupt occurs, the DSP services it only when all interrupts are globally enabled and the particular interrupt is individually enabled. Typically, you enable interrupts both globally and individually in your main program and at the appropriate place wait for an interrupt to occur.

**Global Interrupts.** You can enable and disable interrupts globally using these instructions:

```
ENA INT;      /* Enable interrupts globally */
DIS INT;      /* Disable interrupts globally */
```

With interrupts globally disabled, the DSP does not recognize or latch any interrupts that occur and so cannot service them.

**Individual Interrupts.** You can enable and disable interrupts individually using the register load instruction (for details, see, “[Direct Register Load](#)” on page 7-27). For example, to enable (unmask) interrupts 3, 5, 7, and 8, you set them to 1:

```
IMASK = 0x01A8; /* Enable interrupts 8, 7, 5, & 3 only */
```

Interrupt 0 is nonmaskable in IMASK and cannot be enabled or disabled globally.

With interrupts globally enabled and individual interrupts enabled in IMASK, the DSP automatically services them when it detects their respective bits set in IRPTL.

With interrupts globally enabled and individual interrupts disabled in IMASK, when they occur and are latched in IRPTL, you can choose to unmask their respective bits in IRPTL and service them or to clear their bits and reject them. For example:

```
ENA INT;          /* globally enable ints */
IMASK = 0x0000;   /* individually disable all ints */
NOP;
NOP;              /* any number of instructions */
NOP;
AX0 = IRPTL;      /* load IRPTL into AX0 */
AF = TSTBIT 8 OF AX0; /* test interrupt 8 */
IF NE JUMP normal; /* If 0 continue normal flow */
AR = CLRBIT 8 of AX0; /* else clear interrupt (bit 8) */
IRPTL = AR;       /* load IRPTL with new value */
normal:
NOP;              /* continue normal program flow */
```

IMASK is the interrupt mask register, and IRPTL is the interrupt latch register. As shown in [Table 2-5 on page 2-10](#), the IMASK and the IRPTL registers match each other bit for bit.

## Program Flow Instructions

### Switching Contexts

The DSP has two sets of DAG address registers and two sets of data registers that enable you to quickly switch between the context of normal processing and the context of interrupt processing as needed. The secondary register sets eliminate the need to save the state of the data and address registers before processing an interrupt and reduces interrupt latency. (For details on DSP modes, see [“MSTAT Mode Control Register” on page 8-4.](#))

Typically, you switch from primary to secondary registers at the beginning of the interrupt’s ISR. To do so, you use the following instructions:

```
ENA SEC_REG;      /* enable secondary data registers */
ENA SEC_DAG;     /* enable secondary DAG address registers */
```

You use the RTI instruction at the end of the routine to return program flow to the main program. This instruction automatically switches context back to the primary registers when it restores the ASTAT, MSTAT, and SSTAT registers. So, for example, an interrupt service routine might look like this:

```
service_interrupt:
    ENA SEC_REG, ENA SEC_DAG; /* enable secondary registers */
    NOP;
    NOP;                      /* ISR code */
    NOP;
    RTI;                      /* return from interrupt and */
                              /* enable primary registers */
```

### Nesting Interrupts

Nested interrupts enable the DSP to respond to more than one interrupt at a time. A higher priority interrupt suspends a lower priority interrupt’s routine. After the higher priority interrupt’s RTI executes, the lower priority interrupt’s routine continues executing.

Without nested interrupts, only one interrupt at a time gets serviced, so other interrupts remain pending until the RTI of the current routine executes. Then the pending interrupt with highest priority gets serviced.

To use nested interrupts, you must enable them in the `ICNTL` register. To do so, you explicitly set bit 4 of `INCTL`:

```
ICNTL = 0x0010;
```

Once enabled, any interrupt with higher priority than the currently executing ISR suspends that ISR's execution. [Table 2-4 on page 2-8](#) lists and describes the bits of the `ICNTL` register.

The DSP supports up to sixteen nested interrupts, but has only one set of secondary data and DAG address registers. So, if your application uses deeply nested interrupts, you may need to manually save the state of the data registers and DAG address registers to memory in your ISR routines.

To do so:

- Set up a segment in memory to save the current state of the data and DAG address registers.
- In the ISR, save to memory the state of the data registers and the state of the DAG address registers that you intend to use.

## Application Performance

The ADSP-219x's instruction set provides many ways to optimize code to accommodate particular applications. This section discusses optimization strategies for these topics:

- Exiting a loop
- Using long jumps and calls
- Effect latencies

### Exiting a Loop

When you exit an infinite loop or abort a finite loop prematurely, the loop hardware fixes and restores the loop stacks before the `POP LOOP` instruction

## Program Flow Instructions

executes. So, with few restrictions, you can branch out of a loop from almost any location, regardless of the length of the loop. However, for optimal performance, consider these scenarios:

- Jumps or calls nearby loop ends may add extra cycles of loop stack clean-up when the jump or call is taken.

```
CNTR = 5;
MX0 = 0xFF;
MY0 = 0xFF;
DO mac_loop UNTIL CE; /* start of mac_loop */
    NOP;
    NOP;
    MR = MX0 * MY0 (SS);
    IF MV JUMP abort_loop;
mac_loop:
    NOP; /* end of mac_loop */
NOP; /* 1st instr after mac_loop */
NOP; /* 2nd instr after mac_loop */
NOP; /* 3rd instr after mac_loop */
abort_loop: /* loop exit routine */
    POP LOOP;
    JUMP mac_loop + 1;
```

The jump to `abort_loop` takes 1, 2, or 3 extra cycles, depending on whether the first, second, and third instruction after the end of the `mac_loop` are also loop ends, to clean up the loop stacks before the `POP LOOP` instruction executes. Impact on performance is minimal if the `POP` occurs only once.

- Jumps or calls nearby loop ends add 1, 2, or 3 extra cycles each time the branch is taken.

```
CNTR = 5;
DO little_loop UNTIL CE;
    NOP; /* 1st instr. of little_loop */
    NOP; /* 2nd instr. of little_loop */
    NOP; /* 3rd instr. of little_loop */
    IF MV CALL fix_my_data;
little_loop:
    NOP; /* end of little_loop */
    NOP; /* 1st instr. after little_loop */
```

```

NOP;          /* 2nd instr. after little_loop */
NOP;          /* 3rd instr. after little_loop */
NOP;          /* 4th instr. after little_loop */

fix_my_data:
NOP;
NOP;
RTS;

```

The call to `fix` takes 1, 2, or 3 extra cycles, depending on whether the first, second, and third instructions after the end of `little_loop` are also loop ends, to clean up the loop stacks. To avoid the degradation in performance this construct incurs, you could move the `CALL` instruction further up in the loop or insert the called subroutine in the loop.

- Because the loop begin stack and PC stack are separate and distinct, this loop construct causes a loop to fall gracefully through the next loop end.

```

CNTR = 5;
DO bigger_loop UNTIL CE;
NOP;          /* 1st instr. of bigger_loop */
NOP;          /* 2nd instr. of bigger_loop */
NOP;          /* 3rd instr. of bigger_loop */
IF MV CALL fix_bigger_data;
NOP;
NOP;
bigger_loop:
NOP;          /* end of bigger_loop */
NOP;          /* 1st instr. after bigger_loop */
NOP;          /* 2nd instr. after bigger_loop */
NOP;          /* 3rd instr. after bigger_loop */

fix_bigger_data:
NOP;
NOP;
RTS;

```

The call to `fix_bigger_data` takes no extra cycles, unless the loop is aborted. One abort routine can serve all loops in nearby code space

## Program Flow Instructions

since the routine is identical for each. Even after the loop is aborted, the end of the `bigger_loop` still executes, and the loop falls gracefully out.

### Using Long Jumps and Calls

The instruction set provides several jump/call instructions that support different address ranges for addressing branch targets:

- -4096 to +4095      [“Direct JUMP \(PC relative\)” on page 8-27](#)
- -32768 to +32767      [“CALL \(PC relative\)” on page 8-30](#)
- -32768 to +32767      [“JUMP \(PC relative\)” on page 8-34](#)
- -16777216 to +16777215 [“Long CALL” on page 8-37](#)
- -16777216 to +16777215 [“Long JUMP” on page 8-40](#)

Usually, programmers must determine in advance the offset of the target from the branch and use the appropriate branch instruction, making sure the address of the branch target falls within the address range of the branch instruction.

However, using an option provided in the assembler and in the linker with any of the PC relative branch instructions, you can let the tools determine and select which branch instruction to use based on the actual address of the branch target. To do so, you encode PC relative branch instructions and use the assembler’s and linker’s `-jcs21` option, which directs the tools to substitute, during linking, `LJUMP` or `LCALL` for any particular PC relative branch instruction as appropriate. For details, see the *Assembler Manual for ADSP-219x Family DSPs* and the *Linker & Utilities Manual for ADSP-219x Family DSPs*.

When using the linker’s `-jcs21` option, you need to understand how it alters the linker’s operation, so you can fine tune your code accordingly.

When the linker substitutes `LJUMP` or `LCALL` for a corresponding PC relative branch instruction:

- It substitutes an absolute address for the PC relative address.
- If it encounters the `(DB)` option in a PC relative instruction, it moves the 48 bits (either two one-word instructions or one two-word instruction) from the two delay slots following the PC relative instruction and inserts them directly in front of the `LCALL` or `LJUMP` instruction.

For conditional PC relative instructions, this procedure could change the condition code upon which the branch instruction is predicated. To avoid this potential bug, base `(DB)` branch instructions on negated conditions (`IF NOT COND`), not positive ones (`IF COND`).

- For unconditional PC relative instructions, it always encodes the `TRUE` condition.

### Effect Latencies

An effect latency occurs when some instructions write or load a value into a register, which changes the value of one or more bits in the register. Effect latency refers to the time it takes after the write or load instruction for the effect of the new value to become available for other instructions to use. [For more information, see “Register Load Latencies” on page 7-9.](#)

## Program Flow Instructions

### DO UNTIL (PC relative)

```
DO <Imm12> [UNTIL <Term>] ;
```

#### FUNCTION

Sets up the looping circuitry for zero-overhead looping. The `DO UNTIL` instruction uses the current PC (Program Counter) as the basis for determining the beginning of the loop and the PC-relative 12-bit offset value (`Imm12`) as the end of a loop.

#### INPUT

`Imm12` 12-bit, positive offset value added to the address (PC) of the `DO UNTIL` instruction. Valid values range from 1 to 4095. (For good programming practice, use declared labels.)

`Term` Loop terminator. Valid loop termination conditions are `FOREVER` and `CE`.

#### OUTPUT

None.

#### STATUS FLAGS

Affected Flags—set or cleared by the operation	Unaffected Flags
LPSTKEMPTY (always cleared), LPSTKFULL, STKOVERFLOW	PCSTKEMPTY, PCSTKFULL, PCSTKLVL, STSSTKEMPTY
For information on these status bits in the SSTAT register, see <a href="#">Table 2-7 on page 2-14</a> .	

### DETAILS

The loop begins at the instruction directly following the `DO UNTIL` instruction ( $PC + 1$ ) and ends at the instruction located at the offset address specified in the `DO UNTIL` instruction—( $PC + \langle imm12 \rangle$ ).

When using the `FOREVER` (infinite loop) termination condition, you must explicitly exit the loop by generating a status condition on which to base a jump to a location outside the loop. If you omit a terminator (`DO <loop>`), the instruction defaults to `FOREVER`.

When using the `CE` (counter expired) termination condition, before entering the loop, you must load the `CNTR` register with the number of times to execute the loop. Each pass through the loop decrements and tests the counter value in the loop counter stack, not the `CNTR` register (for details, see “[Loop Stacks Operation](#)” on page 8-10). When the counter expires, looping terminates.



If using `CE` termination, you must load a value  $>1$  in the `CNTR` register.


Finite loops (`CE` terminator) execute repeatedly until the loop terminator occurs. Infinite loops (`FOREVER`) execute repeatedly until a condition occurs that invokes an explicit jump to the address of an instruction outside the loop.

At execution, the `DO UNTIL` instruction pushes:

- The address of the loop start instruction ( $PC + 1$ ) onto the loop begin stack.
- The address of the loop end instruction ( $PC + \langle imm12 \rangle$ ) and the code of the loop terminator onto the loop end stack.
- The contents of the `CNTR` register onto the loop counter stack.

## Program Flow Instructions

During execution of a finite loop, the DSP tests and decrements the loop counter value stored in the loop counter stack—not the value in the `CNTR` register—at the end of each pass of the loop.

 The `CNTR` register retains the original loop counter value until you load it with a new value, either explicitly with a load instruction or with a `POP LOOP` instruction.

To test the current value of the decrementing loop counter, you pop the value off the loop counter stack into the `CNTR` register, move the `CNTR` contents into a data register, and then push the `CNTR` contents back onto the stack.

During execution of an infinite loop, the DSP pushes the current value of the `CNTR` register and the `FOREVER` bit onto the loop counter stack. When the `FOREVER` bit is set, the loop logic ignores the loop counter value. If you set up an infinite loop with the `PUSH LOOP` instruction instead of the `DO UNTIL` instruction, you must set the `FOREVER` bit of `LPSTACKP` (bit 15). (For details, see [“Loop Stacks Operation” on page 8-10](#) and [“PUSH or POP Stacks” on page 8-55](#).)

You can nest up to eight loops because each of the loop stacks have eight locations. The DSP pushes the loop begin stack, loop end stack, and loop counter stack for each level of nesting.

Follow these guidelines when coding loops:

- For nested loops, set up a separate counter for each loop, and end each loop with a separate instruction.
- Do not use the `RTI` or `RTS` instruction inside a loop.
- Do not use a `PUSH` or `POP` instruction in the last seven lines of a loop. Avoid using `PUSH` or `POP` instructions within loops.

- Do not use a CALL instruction in the last line of a loop because the return address then resides outside of the loop, a condition that causes incorrect sequencing.
- You can use a JUMP instruction in the last line of an infinite loop.
- If you use a JUMP or CALL instruction to abort a loop, make sure you handle the loop stacks properly (POP LOOP). POP LOOP automatically pops each of the loop stacks. For details, see “Stacks” on page 8-7 and “PUSH or POP Stacks” on page 8-55.

### EXAMPLES

```
/* a finite loop example */
CNTR = 0xF;
IOPG = 0x1;
SI = AX0;
DM(IO += M0) = SI;
MR=0, MX0 = DM(IO+=M0), MY0 = PM(I4+=M4);
DO a_finite_loop UNTIL CE;
    MR = MR+MX0*MY0(SS), MX0 = DM(IO+=M0), MY0=PM(I4+=M4);
    MR = MR+MX0*MY0(RND);
a_finite_loop:
    IO(0xFF) = MR1;

/* an infinite loop example */
IOPG = 0x1;
IO = 0x1000;
M0 = 1;
L0 = 0;
DO an_infinite_loop;
    AX0 = DM(IO+=M0);
    AR = AX0 + AY0;
    IF AV JUMP exit_an_infinite_loop;
an_infinite_loop:
    DM(IO + 100) = AR;
NOP;                /* 1st instruction after an infinite loop */
NOP;
NOP;                /* any number of instructions */
NOP;
exit_an_infinite_loop:
    POP LOOP;
```

## Program Flow Instructions

```
JUMP an_infinite_loop +1;

/* a nested loop example */
AX0 = DM(I0 += M0), AY0 = PM(I4 += M4);
CNTR = 10;
DO outer_nested_loop UNTIL CE;
  CNTR = 20;
  DO middle_nested_loop UNTIL CE;
    CNTR = 30;
    DO inner_nested_loop UNTIL CE;
      AR =AX0 + AY0, AX0=DM(I0 += M0), AY0=PM(I4 += M4);
      inner_nested_loop:
        DM(I2 += M2) = AR;
    middle_nested_loop:
      NOP;
  outer_nested_loop:
    NOP;
```

### SEE ALSO

- [“Type 11: Do ... Until” on page 9-34](#)
- [“Conditions” on page 8-2](#)
- [“Counter-Based Conditions” on page 8-3](#)
- [“Stacks” on page 8-7](#)

## Direct JUMP (PC relative)

```
[IF COND] JUMP <Imm13> [(DB)] ;
```

### FUNCTION

This branch instruction causes program execution to continue at the offset address specified in the instruction. The offset address is the sum of the PC of the JUMP instruction and the 13-bit immediate value supplied in the instruction ( $PC + \langle imm13 \rangle$ ).

If execution is based on an optional condition, the JUMP instruction executes only if the condition evaluates true and a NOP operation executes if the condition evaluates false. Omitting the condition forces unconditional execution of the loop. For a list of valid conditions, see [“Conditions” on page 8-2](#).

The branch can be immediate or delayed (using the optional `((DB))`). If immediate, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency of four NOP cycles.

If using the optional delayed branch `((DB))` syntax, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency equal to four cycles. The two instructions directly following the JUMP instruction execute in sequence during the first two latency cycles if the branch is taken. Even if the branch is not taken, the instructions occupying the two branch delay slots still execute.

### INPUT

**Imm13** A 13-bit, twos-complement offset value added to the address (PC) of the JUMP instruction. Valid values range from  $-4096$  to  $+4095$ .

For good programming practice, always use a label, rather than a numeric value, since a label is relocatable.

# Program Flow Instructions

## OUTPUT

None.

## STATUS FLAGS

None.

## DETAILS

When using the (DB) option, you cannot insert the following instructions after the JUMP instruction, in the two delayed branch slots:

- Stack manipulation instructions—PUSH/POP
- Branch instructions—JUMP, CALL, RTI, RTS
- Loop instruction—DO UNTIL

You can use the Indirect 16-bit Memory Write—Immediate Data instruction. For details, see [“Indirect 16-bit Memory Write—immediate data” on page 7-55](#). Because it is a double-word instruction, you must place it in the first delay branch slot, right after the CALL instruction.

The number of cycles required to perform a JUMP operation depends on whether the branch is taken or not. With the immediate branch option, when the branch is taken, the DSP flushes the instruction pipeline except for the JUMP instruction and inserts four NOP cycles. As shown in [Table 8-1 on page 8-29](#), when you use the (DB) option, the operation still takes five cycles (JUMP instruction + four cycles of latency), but the DSP executes in sequence the two instructions following the JUMP instruction, flushing only the top of the instruction pipeline.

If the address range of this instruction is inadequate, you can use the LJUMP instruction, but lose use of the (DB) option, or you can retain the (DB) option and let the tools determine during assembly/linking whether to use this instruction or substitute the LJUMP instruction. For details see [“Using Long Jumps and Calls” on page 8-20](#).

Table 8-1. Branch (JUMP) Execution Cycles

Branch Case	Time to Execute	Delayed Branch Fills	Delayed Branch NOPs
Taken	5 cycles	2 cycles	2 cycles
Not Taken	1 cycle	0 cycles	0 cycles

### EXAMPLES

```

    JUMP first_branch_target; /* immediate branch jump */
    NOP;                      /* any number of instructions */
first_branch_target:
    NOP;                      /* any number of instructions */
    NOP;
    Jump second_branch_target (DB); /* delayed branch jump */
    AR = PASS 0;
    AR = AX0 + AY0; /* these two instr. after jump execute */
    NOP;
    NOP;                      /* any number of instructions */
second_branch_target:
    NOP;
    NOP;                      /* any number of instructions */
    IF NE JUMP third_branch_target (DB);
    MR = 0; /* these two instr. after (DB) jump execute */
    AR = PASS 0; /* whether or not cond branch is taken */
    NOP;
    NOP;                      /* any number of instructions */
third_branch_target:
    NOP;
    NOP;                      /* any number of instructions */

```

### SEE ALSO

- [“Type 10: Direct Jump” on page 9-32](#)
- [“Branch Options” on page 8-5](#)
- [“Addressing Branch Targets” on page 8-6](#)

## Program Flow Instructions

### CALL (PC relative)

```
CALL <Imm16> [(DB)] ;
```

#### FUNCTION

This instruction causes the Program Sequencer to branch to the offset address specified in the instruction and execute the subroutine at that location. The offset address is the sum of the PC of the CALL instruction and the 16-bit immediate value supplied in the instruction ( $PC + \langle imm16 \rangle$ ).

The branch can be immediate or delayed (using the optional ((DB)). If immediate, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency of four NOP cycles.

If using the optional delayed branch ((DB)) syntax, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency equal to four cycles. The two instructions directly following the CALL instruction execute in sequence during the first two latency cycles of the branch.

#### INPUT

*imm16* 16-bit, twos-complement offset value added to the address (PC) of the CALL instruction or a declared label. Valid values range from  $-32768$  to  $+32767$ .

For good programming practice, always use a label, rather than a numeric value, since a label is relocatable.

#### OUTPUT

None.

## STATUS FLAGS

Affected Flags—set or cleared by the operation	Unaffected Flags
PCSTKFULL, PCSTKLVL, STKOVERFLOW, PCSTKEMPTY	LPSTKEMPTY, LPSTKFULL, STSSTKEMPTY
For information on these status bits in the SSTAT register, see <a href="#">Table 2-7 on page 2-14</a> .	

## DETAILS

Before branching, the Program Sequencer automatically pushes onto the PC stack the return address of the next instruction to execute after returning from the called subroutine. The next instruction to execute is:

- Immediate CALL      The first instruction following the CALL instruction.
- Delayed CALL        The third instruction following the CALL instruction.

To return from the subroutine, you must explicitly issue an RTS instruction. For details, see [“Return from Subroutine” on page 8-52](#).

When using the (DB) option, you cannot insert the following instructions after the CALL instruction, in the two delay branch slots:

- Stack manipulation instructions—PUSH/POP
- Branch instructions—JUMP, CALL, RTI, RTS
- Loop instruction—DO UNTIL

You can use the Indirect 16-bit Memory Write—Immediate Data instruction. For details, see [“Indirect 16-bit Memory Write—immediate data” on page 7-55](#). Because it is a double-word instruction, you must place it in the first delay branch slot, right after the CALL instruction.

## Program Flow Instructions

The number of cycles required to perform a CALL operation depends on whether the branch is taken or not. With the immediate branch option, when the branch is taken, the DSP flushes the instruction pipeline except for the CALL instruction and inserts four NOP cycles. As shown in [Table 8-2 on page 8-32](#), when you use the (DB) option, the operation still takes five cycles (CALL instruction + four cycles of latency), but the DSP executes in sequence the two instructions following the CALL instruction, flushing only the top of the instruction pipeline.

Table 8-2. Branch (CALL) Execution Cycles

Branch Case	Time to Execute	Delayed Branch Fills	Delayed Branch NOPs
Taken	5 cycles	2 cycles	2 cycles
Not Taken	1 cycle	0 cycles	0 cycles

If the address range of this instruction is inadequate, you can use the LCALL instruction, but you lose use of the (DB) option, or you can retain the (DB) option and let the tools determine during assembly/linking whether to use this instruction or substitute the LCALL instruction. For details see [“Using Long Jumps and Calls” on page 8-20](#) and [“Long CALL” on page 8-37](#).

### EXAMPLES

```
CALL data_shift_subroutine (DB);
  AX0 = DM(I0 += M1), AY0 = PM(I4 += M5); /* these two instr. */
  AR = PASS 0; /* execute before (DB) branch starts */
DM(I1 += M1) = SR0; /* RTS returns here */
NOP;
NOP; /* any number of instructions */
NOP;
data_shift_subroutine:
  AR = AX0 + AY0;
  AX1 = 3; SE = AR;
  SR = ASHIFT SI (HI);
  RTS; /* returns operation */
```

### SEE ALSO

- [“Type 10: Direct Jump” on page 9-32](#)
- [“Branch Options” on page 8-5](#)
- [“Addressing Branch Targets” on page 8-6](#)

## Program Flow Instructions

### JUMP (PC relative)

```
JUMP <Imm16> [(DB)] ;
```

#### FUNCTION

This branch instruction causes program execution to continue at the offset address specified in the instruction. The offset address is the sum of the PC of the `JUMP` instruction and the 16-bit immediate value supplied in the instruction ( $PC + \langle imm16 \rangle$ ).

The branch can be immediate or delayed (using the optional `((DB))`). If immediate, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency of four NOP cycles.

If using the optional delayed branch `((DB))` syntax, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency equal to four cycles. The two instructions directly following the `JUMP` instruction execute in sequence during the first two latency cycles of the branch.

#### INPUT

`imm16` 16-bit, twos-complement offset value added to the address (PC) of the `JUMP` instruction or a declared label. Valid values range from  $-32768$  to  $+32767$ .

For good programming practice, always use a label, rather than a numeric value, since a label is relocatable.

#### OUTPUT

None.

### STATUS FLAGS

None.

### DETAILS

When using the (DB) option, you cannot insert the following instructions in the two delay branch slots directly after the JUMP instruction:

- Stack manipulation instructions—PUSH/POP
- Branch instructions—JUMP, CALL, RTI, RTS
- Loop instruction—DO UNTIL

You can use the Indirect 16-bit Memory Write—Immediate Data instruction. For details, see [“Indirect 16-bit Memory Write—immediate data” on page 7-55](#). Because it is a double-word instruction, you must place it in the first delay branch slot, right after the CALL instruction.

The number of cycles required to perform a JUMP operation depends on whether the branch is taken or not. With the immediate branch option, when the branch is taken, the DSP flushes the instruction pipeline except for the JUMP instruction and inserts four NOP cycles. As shown in [Table 8-1 on page 8-29](#), when you use the (DB) option, the operation still takes five cycles (JUMP instruction + four cycles of latency), but the DSP executes in sequence the two instructions following the JUMP instruction, flushing only the top of the instruction pipeline.

If the address range of this instruction is inadequate, you can use the LJUMP instruction, but you lose use of the (DB) option, or you can retain the (DB) option and let the tools determine during assembly/linking whether to use this instruction or substitute the LJUMP instruction. For details see [“Using Long Jumps and Calls” on page 8-20](#) and [“Long JUMP” on page 8-40](#).

# Program Flow Instructions

## EXAMPLES

```
.SECTION/PM seg_code;
  JUMP my_cod2_label; /* jumps to 16-bit relative address */
  NOP;
  NOP;                /* any number of instructions */
  NOP;
my_code_exit_label:
  NOP;                /* jump from seg_cod2 comes here */
  NOP;
  NOP;                /* any number of instructions */
  NOP;
.SECTION/PM seg_cod2;
my_cod2_label:
  NOP;
  NOP;                /* any number of instructions */
  NOP;
  JUMP my_code_exit_label;
```

## SEE ALSO

- [“Type 10: Direct Jump” on page 9-32](#)
- [“Branch Options” on page 8-5](#)
- [“Addressing Branch Targets” on page 8-6](#)

## Long CALL

```
[IF COND] LCALL <Imm24> ;
```

### FUNCTION

This instruction causes the Program Sequencer to branch to the absolute address specified in the instruction and execute the subroutine at that location. The absolute address is the 24-bit immediate value supplied in the instruction. The 24-bit immediate value enables programs to access any location in program memory address space.

If execution is based on a condition, the `JUMP` instruction executes only if the condition evaluates true and a `NOP` operation executes if the condition evaluates false. Omitting the condition forces unconditional execution of the loop. For a list of valid conditions, see [“Conditions” on page 8-2](#).

### INPUT

`Imm24` 24-bit, twos-complement value or a declared label. Values range from  $-16777216$  to  $+16777215$ .

For good programming practice, always use a label, rather than a numeric value, since a label is relocatable.

### OUTPUT

None.

# Program Flow Instructions

## STATUS FLAGS

Affected Flags—set or cleared by the operation	Unaffected Flags
PCSTKFULL, PCSTKLVL, STKOVERFLOW, PCSTKEMPTY	LPSTKEMPTY, LPSTKFULL, STSSTKEMPTY
For information on these status bits in the SSTAT register, see <a href="#">Table 2-7 on page 2-14</a> .	

## DETAILS

For details on using the assembler's and linker's `-jcs21` option to direct the tools to determine when to replace PC relative `CALL` instructions with this instruction, see [“Using Long Jumps and Calls” on page 8-20](#).

This is a double-word instruction, so it executes in six (2 instruction + 4 latency) cycles.

Before branching, the Program Sequencer automatically pushes onto the PC stack the return address of the next instruction to execute after returning from the called subroutine. The next instruction to execute is the first instruction following the `LCALL` instruction.

To return from the subroutine, you must explicitly issue an `RTS` instruction. For details, see [“Return from Subroutine” on page 8-52](#).

## EXAMPLES

```
.SECTION/PM seg_code;
IF EQ LCALL my_faraway_routine;
  NOP;                /* execution returns here */
  NOP;
  NOP;                /* any number of instructions */
  NOP;

.SECTION/PM seg_cod2;
my_faraway_routine:
  NOP;
  NOP;                /* any number of instructions */
```

NOP;  
RTS;

### SEE ALSO

- “Type 36: Long Jump/Call” on page 9-59
- “Branch Options” on page 8-5
- “Addressing Branch Targets” on page 8-6
- “Using Long Jumps and Calls” on page 8-20.

## Program Flow Instructions

### Long JUMP

```
[IF COND] LJUMP <Imm24> ;
```

#### FUNCTION

This branch instruction causes program execution to continue at the absolute address specified in the instruction. The absolute address is the 24-bit immediate value supplied in the instruction. The 24-bit immediate value enables programs to access any location in program memory address space.

If execution is based on a condition, the JUMP instruction executes only if the condition evaluates true and a NOP operation executes if the condition evaluates false. Omitting the condition forces unconditional execution of the loop. For a list of valid conditions, see [“Conditions” on page 8-2](#).



This instruction is a two-word instruction and requires (at minimum) six cycles to execute. [For more information, see “Type 36: Long Jump/Call” on page 9-59.](#)

#### INPUT

Imm24 24-bit, twos-complement value or a declared variable. Values range from  $-16777216$  to  $+16777215$ .

For good programming practice, always use a label, rather than a numeric value, since a label is relocatable.

#### OUTPUT

None.

#### STATUS FLAGS

None.

## DETAILS

For details on using the ADSP-219x assembler's `-jcs2l` (convert Jump/Call Short to Long) option to direct the tools to determine when to replace PC relative JUMP instructions with LJUMP instructions, see [“Using Long Jumps and Calls” on page 8-20](#).

## EXAMPLES

```

/* Long JUMP example nearby half */

.SECTION/PM seg_code;
.GLOBAL my_local_exit_label;
.EXTERN my_faraway_label;
    LJUMP my_faraway_label; /* jumps to 24-bit relative addr */
    NOP;
    NOP; /* any number of instructions */
    NOP;
my_local_exit_label:
    NOP; /* jump from seg_cod2 comes here */
    NOP;
    NOP; /* any number of instructions */

/* Long JUMP example faraway half */

.SECTION/PM seg_xpmc;
.GLOBAL my_faraway_label;
.EXTERN my_local_exit_label;
my_faraway_label:
    NOP; /* any number of instructions */
    NOP;
    LJUMP my_local_exit_label;

```

## SEE ALSO

- [“Type 36: Long Jump/Call” on page 9-59](#)
- [“Branch Options” on page 8-5](#)
- [“Addressing Branch Targets” on page 8-6](#)
- [“Using Long Jumps and Calls” on page 8-20](#).

## Program Flow Instructions

### Indirect CALL

```
[IF COND] CALL (<Ireg>) [(DB)] ;
```

#### FUNCTION

This instruction causes the Program Sequencer to branch to the address pointed to by the DAG index register (*Ireg*). The *Ireg* supplies the sixteen LSBs of the 24-bit address, and the *IJPG* register supplies the eight MSBs (page address) of the 24-bit address. You must explicitly load the *IJPG* register with the eight MSBs of the address before executing this instruction (for details, see [“Data Move Instructions” on page 7-1](#)).

If execution is based on a condition, the *CALL* instruction executes only if the condition evaluates true, and a *NOP* operation executes if the condition evaluates false. Omitting the condition forces unconditional execution of the loop. For a list of valid conditions, see [“Conditions” on page 8-2](#).

The branch can be immediate or delayed (using the optional *((DB))*). If immediate, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency of four *NOP* cycles.

If using the optional delayed branch *((DB))* syntax, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency equal to four cycles. The two instructions directly following the *CALL* instruction execute in sequence during the first two latency cycles if the branch is taken. Even if the branch is not taken, the instructions occupying the two branch delay slots still execute.

#### INPUT

*Ireg* I0–I3 (DAG1 index registers) or I4–I7 (DAG2 index registers)

#### OUTPUT

None.


## STATUS FLAGS

Affected Flags—set or cleared by the operation	Unaffected Flags
PCSTKFULL, PCSTKLVL, STKOVERFLOW, PCSTKEMPTY	LPSTKEMPTY, LPSTKFULL, STSSTKEMPTY
For information on these status bits in the SSTAT register, see <a href="#">Table 2-7 on page 2-14</a> .	

## DETAILS

Before branching, the Program Sequencer automatically pushes onto the PC stack the return address of the next instruction to execute after returning from the called subroutine. The next instruction to execute is:

- Immediate CALL      The first instruction following the CALL instruction.
- Delayed CALL        The third instruction following the CALL instruction.

 To return from the subroutine, you must explicitly issue an RTS instruction. For details, see [“Return from Subroutine” on page 8-52](#).

When using the (DB) option, you cannot insert the following instructions after the CALL instruction, in the two delay branch slots:

- Stack manipulation instructions—PUSH/POP
- Branch instructions—JUMP, CALL, RTI, RTS
- Loop instruction—DO UNTIL

You can use the Indirect 16-bit Memory Write—Immediate Data instruction (for details, see [page 7-55](#)), but because it is a double-word instruction, you must place it in the first delay branch slot, right after the CALL instruction.

## Program Flow Instructions

The number of cycles required to perform a `CALL` operation depends on whether the branch is taken or not. With the immediate branch option, when the branch is taken, the DSP flushes the instruction pipeline except for the `CALL` instruction and inserts four `NOP` cycles. As shown in [Table 8-2 on page 8-32](#), when you use the `(DB)` option, the operation still takes five cycles (`CALL` instruction + four cycles of latency), but the DSP executes in sequence the two instructions following the `CALL` instruction, flushing only the top of the instruction pipeline.

### EXAMPLES

```
I5 = sampling_routine;
AR = AR + AX0;
IF EQ CALL (I5) (DB);
    DM(IO += M1) = AR; /* these two instr. execute */
    AR = 0; /* whether or not the branch is taken */
AR = PASS 0; /* RTS returns execution to this instr. */
NOP;
NOP; /* any number of instructions */
NOP;

sampling_routine:
    MX0 = DM(IO+=M1);
    MR = MX0 * MY0 (SS);
    NOP;
    NOP; /* any number of instructions */
    NOP;
    RTS;
```

### SEE ALSO

- [“Type 19: Indirect Jump/Call” on page 9-41](#)
- [“Branch Options” on page 8-5](#)
- [“Addressing Branch Targets” on page 8-6](#)

## Indirect JUMP

```
[IF COND] JUMP (<Ireg>) [(DB)] ;
```

### FUNCTION

This branch instruction causes program execution to continue at the address pointed to by the DAG index register (*Ireg*). The *Ireg* supplies the sixteen LSBs of the 24-bit address, and the *IJPG* register supplies the eight MSBs (page address) of the 24-bit address. You must explicitly load the *IJPG* register with the eight MSBs of the address before executing this instruction (for details, see [“Data Move Instructions” on page 7-1](#)).

If execution is based on a condition, the *JUMP* instruction executes only if the condition evaluates true, and a *NOP* operation executes if the condition evaluates false. Omitting the condition forces unconditional execution of the loop. For a list of valid conditions, see [“Conditions” on page 8-2](#).

The branch can be immediate or delayed (using the optional *((DB))*). If immediate, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency of four *NOP* cycles.

If using the optional delayed branch *((DB))* syntax, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency equal to four cycles. The two instructions directly following the *JUMP* instruction execute in sequence during the first two latency cycles if the branch is taken. Even if the branch is not taken, the instructions occupying the two branch delay slots still execute.

### INPUT

*Ireg* I0–I3 (DAG1 index registers) or I4–I7 (DAG2 index registers)

### OUTPUT

None.

# Program Flow Instructions

## STATUS FLAGS

None.

## DETAILS

Loading the `IJPG` register or an `Ireg` has a zero (0) effect latency for this instruction, so the new value is available on the next instruction cycle.

When using the `(DB)` option, you cannot insert the following instructions after the `JUMP` instruction, in the two delay branch slots:

- Stack manipulation instructions—`PUSH/POP`
- Branch instructions—`JUMP, CALL, RTI, RTS`
- Loop instruction—`DO UNTIL`

You can use the `Indirect 16-bit Memory Write—Immediate Data` instruction. For details, see [“Indirect 16-bit Memory Write—immediate data” on page 7-55](#). Because it is a double-word instruction, you must place it in the first delay branch slot, right after the `JUMP` instruction.

The number of cycles required to perform a `JUMP` operation depends on whether the branch is taken or not. With the immediate branch option, when the branch is taken, the DSP flushes the instruction pipeline except for the `JUMP` instruction and inserts four `NOP` cycles. As shown in [Table 8-1 on page 8-29](#), when you use the `(DB)` option, the operation still takes five cycles (`JUMP` instruction + four cycles of latency), but the DSP executes in sequence the two instructions following the `JUMP` instruction, flushing only the top of the instruction pipeline.

## EXAMPLES

```
I4 = sampling;  
I5 = next_sample;  
  
sampling:  
AR = AR + AX0;  
IF EQ JUMP (I5) (DB);
```

```
DM(I0 += M1) = AR; /* these two instr. execute */
AR = 0; /* whether or not the branch is taken */
JUMP (I4) (DB);
    AX0 = DM(I0 += M1); /* these two instr. execute */
    AR = AX0; /* before the branch starts */

next_sample:
    MX0 = DM(I0 += M1);
    MR = MX0 * MY0 (SS);
    NOP;
    NOP; /* any number of instructions */
    NOP;
    JUMP (I4); /* goes back to sampling */
```

### SEE ALSO

- [“Type 19: Indirect Jump/Call” on page 9-41](#)
- [“Branch Options” on page 8-5](#)
- [“Addressing Branch Targets” on page 8-6](#)

## Program Flow Instructions

### Return from Interrupt

```
[IF COND] RTI [(DB)] [(SS)] ;
```

#### FUNCTION

This instruction executes a return from an interrupt service routine (ISR). It returns program execution to the address of either the first or third instruction following the branch instruction that launched the ISR.

If execution is based on a condition, the `RTI` instruction executes only if the condition evaluates true, and a `NOP` operation executes if the condition evaluates false. Omitting the condition forces unconditional execution of the loop. For a list of valid conditions, see [“Conditions” on page 8-2](#).

The branch can be immediate or delayed (using the optional `((DB))`). If immediate, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency of four `NOP` cycles.

If using the optional delayed branch `((DB))` syntax, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency equal to four cycles. The two instructions directly following the `RTI` instruction execute in sequence during the first two latency cycles if the branch is taken. Even if the branch is not taken, the instructions occupying the two branch delay slots still execute.

For emulation, the `RTI` instruction supports an additional option, the single-step `(SS)` return interrupt. This option causes the instruction at the return address to generate an interrupt when it executes during emulation.

#### INPUT

None.

**OUTPUT**


None.

**STATUS FLAGS**

Affected Flags—set or cleared by the operation	Unaffected Flags
PCSTKFULL, PCSTKEMPTY, PCSTKLVL	LPSTKEMPTY, LPSTKFULL, STKOVERFLOW, STSSTKEMPTY
For information on these status bits in the SSTAT register, see <a href="#">Table 2-7 on page 2-14</a> .	

**DETAILS**

This instruction pops and uses the address at top of the PC stack for the return address. It also pops the value at the top of the status stack and loads it into the arithmetic status register (ASTAT) and the mode status register (MSTAT). So if the ISR enabled secondary registers or changed other DSP modes in MSTAT, the RTI instruction automatically disables them when it executes.

 Do not use an RTI instruction inside a loop without explicitly performing stack maintenance.

When using the (DB) option, you cannot insert the following instructions after the RTI instruction, in the two delay branch slots:

- Stack manipulation instructions—PUSH/POP
- Branch instructions—JUMP, CALL, RTI, RTS
- Loop instruction—DO UNTIL

You can use the Indirect 16-bit Memory Write—Immediate Data instruction. For details, see [“Indirect 16-bit Memory Write—immediate data” on page 7-55](#). Because it is a double-word instruction, you must place it in the first delay branch slot, directly following the RTI instruction.

## Program Flow Instructions

The number of cycles required to perform an RTI depends on whether the branch is taken or not. With the immediate branch option, when the branch is taken, the DSP flushes the instruction pipeline except for the RTI instruction and inserts four NOP cycles. As shown in Table 8-3, when you use the (DB) option, the operation still takes five cycles (RTI instruction + four cycles of latency), but the DSP executes in sequence the two instructions following the RTI instruction, flushing only the top of the instruction pipeline.

Table 8-3. Branch (RTI) Execution Cycles

Branch Case	Time to Execute	Delayed Branch Fills	Delayed Branch NOPs
Taken	5 cycles	2 cycles	2 cycles
Not Taken	2 cycle	0 cycles	1 cycle

### EXAMPLES

```
interrupt_setup:
    /* defined addr of inter. priority registers in IO() memory */
    #define IPR0 0x203
    #define IPR1 0x204
    #define IPR2 0x205
    #define IPR3 0x206
    /* loads interrupt priorities into IPR registers */
    AX0 = 0x3210;
    IO(IPR0) = AX0; /* set priorities for peripherals 3-0 */
    AX0 = 0x7654;
    IO(IPR1) = AX0; /* set priorities for peripherals 7-4 */
    AX0 = 0xBA98;
    IO(IPR2) = AX0; /* set priorities for peripherals 11-8 */
    AX0 = 0x0BBB;
    IO(IPR3) = AX0; /* set priorities for peripherals 14-12 */

    ICNTL = 0x0010; /* set GIE global interrupt enable bit */
    IMASK = 0x4000; /* unmask interrupt ID 14, which is
                    assigned to Timer Interrupt A by IPR2 */
    ENA INT; /* enable interrupts */
```

## Return from Interrupt

```
wait_here_for_interrupt: /* loop waiting for interrupt */
    NOP;
    NOP;           /* any number of instructions */
    NOP;
    JUMP wait_here_for_interrupt;

.SECTION/PM irq_14; /* map this ISR to addr. 0x01C0 with LDF */
timer_a_int:
    ENA SEC_REG, ENA SEC_DAG;
    NOP;
    NOP;           /* up to 32 instructions */
    NOP;
    RTI;
```

### SEE ALSO

- [“Type 20: Return” on page 9-42](#)
- [“Interrupts” on page 8-13](#)
- [“Set Interrupt” on page 8-62](#)
- [“Clear Interrupt” on page 8-64.](#)

## Program Flow Instructions

### Return from Subroutine

```
[IF COND] RTS [(DB)] ;
```


#### FUNCTION

This instruction executes a return from a subroutine. It returns program execution to the address of either the first or third instruction following the branch instruction that called the subroutine.

If execution is based on a condition, the `RTS` instruction executes only if the condition evaluates true, and a `NOP` operation executes if the condition evaluates false. Omitting the condition forces unconditional execution of the branch. For a list of valid conditions, see [“Conditions” on page 8-2](#).

The branch can be immediate or delayed (using the optional `((DB))`). If immediate, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency of four `NOP` cycles.

If using the optional delayed branch `((DB))` syntax, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency equal to four cycles. The two instructions directly following the `RTS` instruction execute in sequence during the first two latency cycles if the branch is taken. Even if the branch is not taken, the instructions occupying the two branch delay slots still execute.

 Do not use an `RTS` instruction inside a loop without explicitly performing stack maintenance. For details, see [begin stack](#).

#### INPUT

None.

#### OUTPUT

None.

## STATUS FLAGS

Affected Flags—set or cleared by the operation	Unaffected Flags
PCSTKFULL, PCSTKEMPTY, PCSTKLVL	LPSTKEMPTY, LPSTKFULL, STKOVERFLOW, STSSTKEMPTY
For information on these status bits in the SSTAT register, see <a href="#">Table 2-7 on page 2-14</a> .	

## DETAILS

This instruction pops and uses the address at top of the PC stack for the return address.

When using the (DB) option, you cannot insert the following instructions after the RTS instruction, in the two delay branch slots:

- Stack manipulation instructions—PUSH/POP
- Branch instructions—JUMP, CALL, RTI, RTS
- Loop instruction—DO UNTIL

You can use the Indirect 16-bit Memory Write—Immediate Data instruction (for details, see [page 7-55](#)), but because it is a double-word instruction, you must place it in the first delay branch slot, right after the RTI instruction.

The number of cycles required to perform an RTS depends on whether the branch is taken or not. With the immediate branch option, when the branch is taken, the DSP flushes the instruction pipeline except for the RTS instruction and inserts four NOP cycles. As shown in [Table 8-4 on page 8-54](#), when you use the (DB) option, the operation still takes five cycles (RTS instruction + four cycles of latency), but the DSP executes in

## Program Flow Instructions

sequence the two instructions following the RTS instruction, flushing only the top of the instruction pipeline.

Table 8-4. Branch (RTS) Execution Cycles

Branch Case	Time to Execute	Delayed Branch Fills	Delayed Branch NOPs
Taken	5 cycles	2 cycles	2 cycles
Not Taken	1 cycle	0 cycles	0 cycle

### EXAMPLES

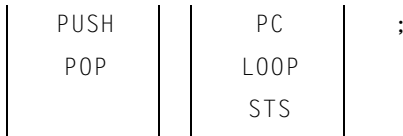
```
I5 = sample_routine;
AR = AR + AX0;
IF EQ CALL (I5) (DB);
    DM(I0 += M1) = AR; /* these two instr. execute */
    AR = 0; /* whether or not the branch is taken */
AR = PASS 0; /* RTS returns execution to this instr. */
NOP;
NOP; /* any number of instructions */
NOP;
```

```
sample_routine:
    MX0 = DM(I0 += M1);
    MR = MX0 * MY0 (SS);
    NOP;
    NOP; /* any number of instructions */
    NOP;
    RTS;
```

### SEE ALSO

- [“Type 20: Return” on page 9-42](#)
- [“Branch Options” on page 8-5](#)
- [“Addressing Branch Targets” on page 8-6](#)

## PUSH or POP Stacks



### FUNCTION

This instruction PUSHes (stores) or POPs (retrieves) a value from the top of the specified stack: PC, LOOP, or STS.

- PC

On a PUSH, stores onto the top of the PC stack a 24-bit address value assembled from the STACKA and STACKP registers. STACKA provides the sixteen LSBs of the address, and STACKP provides the eight MSBs of the address.

On a POP, retrieves the most recently stacked 24-bit address value from the top of the PC stack into the STACKA and STACKP registers. STACKA receives the sixteen LSBs of the address, and STACKP receives the eight MSBs of the address.

- LOOP

On a PUSH, stores onto the top of the loop begin stack the 24-bit loop start address assembled from the STACKA and STACKP registers, pushes onto the top of the loop end stack the 24-bit loop end address assembled from the LPSTACKA and LPSTACKP registers, and pushes onto the top of the loop counter stack the current loop counter value from the CNTR register.

On a POP, retrieves the most recently stacked 24-bit loop start address from the top of the loop begin stack into the STACKA and STACKP registers, pops the most recently stacked 24-bit loop end address from the top of the loop end stack into the LPSTACKA and

## Program Flow Instructions

LPSTACKP registers, and pops the current loop counter value from the top of the loop counter stack into the CNTR register.

- STS

On a PUSH, stores the current values of the ASTAT and MSTAT registers onto the status stack. After each push, the status stack pointer increments by one to access the next available location in the stack.

On a POP, retrieves the most recently stacked 16-bit value of the ASTAT and MSTAT registers from the top of the status stack. After each individual pop, the status stack pointer decrements by one to access the next lowest location (next register value) in the stack.



A PUSH or POP PC has one cycle of latency for all SSTAT register bits, but a PUSH or POP LOOP or STS has one cycle of latency only for the STKOVERFLOW bit in the SSTAT register.

### INPUT

None.

### OUTPUT

None.

### STATUS FLAGS

Affected Flags—set or cleared by the operation	Unaffected Flags
PCSTKEMPTY (affected on POP), PCSTK-FULL, PCSTKLVL (affected on POP), LPSTKEMPTY, LPSTKFULL, STSSTKEMPTY (affected on POP), STKOVERFLOW	(none)
For information on these status bits in the SSTAT register, see <a href="#">Table 2-7 on page 2-14</a> .	

### DETAILS (PUSH)

You can push up to two stacks in parallel by issuing two `PUSH` instructions on the same instruction line, pushing either:

```
PUSH PC, PUSH STS;
```

or

```
PUSH LOOP, PUSH STS;
```

 Do not push the PC and LOOP stacks in parallel (`PUSH PC, PUSH LOOP;`).


If you push the PC and loop stacks in parallel, you push the same address value onto both the PC stack and the loop begin stack. This occurs because `STACKA` and `STACKP` serve as the source registers for both stacks.

Regardless of the number of stacks pushed, this instruction always executes in a single cycle.


Subroutines, loops, and interrupts automatically push certain stacks:

- Calls to subroutines and entry into interrupt service routines automatically push the PC stack.
- Execution of the `DO UNTIL` instruction pushes the loop begin stack, the loop end stack, and the loop counter stack.

Do not use this instruction in either of the two slots directly following a delayed branch instruction.

 Do not use this instruction inside a loop without explicitly performing stack maintenance. For details, see [“PUSH or POP Stacks” on page 8-55](#)

## Program Flow Instructions

 If you set up an infinite loop with the `PUSH LOOP` instruction, you must set bit 15 of `LPSTACKP` to indicate `FOREVER`. Although the `LPSTACKP` register has sixteen bits, but only bit 15 and bits 7:0 are valid. When the `FOREVER` bit is set (bit 15 = 1), the loop logic ignores the loop counter value. When the `FOREVER` bit is cleared (bit 15 = 0), `CE` is the loop terminator condition, and the loop logic decrements the loop counter value.


### DETAILS (POP)

You can pop up to two stacks in parallel by issuing two `POP` instructions on the same instruction line, popping either:

```
POP PC, POP STS;
```

or

```
POP LOOP, POP STS;
```



 Do not pop the PC and loop stacks in parallel (`POP PC, POP LOOP;`).

If you pop the PC and loop stacks in parallel, you lose the loop start address retrieved from the loop begin stack. This occurs because `STACKA` and `STACKP` serve as the destination registers for values popped from both the PC stack and the loop begin stack. In this case, `STACKA` and `STACKP` receive the most recently stacked PC value.

Regardless of the number of stacks popped, this instruction always executes in a single cycle.

Subroutines, loops, and interrupts automatically pop certain stacks:

- Upon exiting, `RTS` and `RTI` instructions automatically pop the PC stack.
- Loop termination automatically pops the loop begin stack, the loop end stack, and the loop counter stack

-  Do not use this instruction in either of the two slots directly following a delayed branch instruction.
-  Do not use this instruction inside a loop without explicitly performing stack maintenance. For details, see [“PUSH or POP Stacks” on page 8-55](#).

### EXAMPLES (PUSH)

```

/* Pushing an infinite loop-loop terminator condition =
FOREVER: */

STACKA = 0x0045;
STACKP = 0x03;
LPSTACKA = 0x004C;
LPSTACKP = 0x03;
PUSH LOOP;

/* Saving the DSP's current state: */

STACKA = 0x0022;
STACKP = 0x05;
PUSH PC, PUSH STS;

```

### EXAMPLES (POP)

```

/* Restoring the DSP's current state: */

POP PC, POP STS;
AR = TSTBIT 6 OF AX0;
IF EQ CALL primary;
AX1 = STACKA;
AY1 = STACKP;
IJPG = AY1;
primary: DIS SEC_DAG, DIS SEC_REG;
        RTS;

/* Aborting a loop: */

CNTR = 10;
MX0 = DM(I2 += M2),
MY0 = PM(I5 += M5);
DO mac UNTIL CE;

```

## Program Flow Instructions

```
MR = MR + MX0 * MY0 (SS),
MX0 = DM(I2 += M2),
MY0 = PM(I5 += M5);
IF MV JUMP abort;
mac:
  DM(I0 += M1) = MR0;
NOP;
NOP; /* any number of instructions */
NOP;
abort:
  POP LOOP;
  JUMP mac + 1;
```

### SEE ALSO

- [“Type 26:Push/Pop/Cache” on page 9-50](#)
- [“MSTAT Mode Control Register” on page 8-4](#)
- [“Stacks” on page 8-7](#)
- [“PC and Status Stack Operation” on page 8-8](#)
- [“Loop Stacks Operation” on page 8-10](#)

## FLUSH CACHE

FLUSH CACHE ;

### FUNCTION

This instruction flushes the instruction cache, invalidating all instructions currently cached, so the next instruction fetch results in a memory access.

Use this instruction when program memory changes to resynchronize the instruction cache with program memory.

### INPUT

None.

### OUTPUT

None.

### STATUS FLAGS

None.

### DETAILS

This operation may require up to six cycles to take effect.



Do not use this instruction in either of the two slots directly following a delayed branch instruction.

### EXAMPLES

FLUSH CACHE ;

### SEE ALSO

- [“Type 26:Push/Pop/Cache” on page 9-50](#)

## Program Flow Instructions

### Set Interrupt

```
SETINT n ;
```

#### FUNCTION

This instruction sets bit  $n$  ( $n = 1$ ) in the interrupt latch register (IRPTL) and its associated interrupt. If the specified interrupt is unmasked, its corresponding bit in the IMASK register is set, program flow immediately branches to and executes the interrupt's service routine. Otherwise, the interrupt request remains latched but ignored until the program unmask it or clears it. If unmasked, the interrupt's ISR executes; if cleared, the interrupt request is rejected.

#### INPUT

$n$  Specifies which bit (and interrupt) in the IRPTL register to set.

Valid values range from 0–15.

The mapping of bits to interrupts is specific to particular DSPs in the ADSP-219x family. For details, see the *ADSP-219x/2191 DSP Hardware Reference*.

#### OUTPUT

None.

#### OPTIONS

None.

**STATUS FLAGS**

Affected Flags—set or cleared by the operation	Unaffected Flags
PCSTKFULL, PCSTKEMPTY, PCSTKLVL, STSSTKEMPTY, STKOVERFLOW	LPSTKEMPTY, LPSTKFULL
For information on these status bits in the SSTAT register, see <a href="#">Table 2-7 on page 2-14</a> .	

**DETAILS**

This instruction has no associated effect latency.

**EXAMPLES**

```
MR = MR+MX0*MY0(SS), MX0 = DM(IO+=M0), MY0 = PM(I4+=M4);
IF MV SAT MR;
IF LT JUMP adjust;

adjust: SETINT 12;
```

**SEE ALSO**

[“Stacks” on page 8-7.](#)

[“Interrupts” on page 8-13.](#)

[“Clear Interrupt” on page 8-64.](#)

[“Type 37: Interrupt” on page 9-60.](#)

## Program Flow Instructions

### Clear Interrupt

```
CLRINT n ;
```

#### FUNCTION

This instruction clears bit  $n$  ( $n = 0$ ) in the interrupt latch register (IRPTL) and its associated interrupt.

This instruction clears a pending interrupt. It is used, in an ISR for example, to clear a pending interrupt that has been detected, but not yet been serviced.

#### INPUT

$n$  Specifies which bit (and interrupt) in the IRPTL register to clear.

Valid values range from 0–15.

The mapping of bits to interrupts is specific to particular DSPs in the ADSP-219x family. For details, see the *ADSP-219x/2191 DSP Hardware Reference*.

#### OUTPUT

None.

#### OPTIONS

None.

## STATUS FLAGS

Affected Flags—set or cleared by the operation	Unaffected Flags
	PCSTKFULL, PCSTKEMPTY, PCSTKLVL, STSSTKEMPTY, STKOVERFLOW, LPSTKEMPTY, LPSTKFULL
For information on these status bits in the SSTAT register, see <a href="#">Table 2-7 on page 2-14</a> .	

## DETAILS

This instruction has no associated latency.

## EXAMPLES

```
AX0 = IRPTL;
AR = TSTBIT 12 of AX0;
IF EQ JUMP clear;

clear: CLRINT 12;
```

## SEE ALSO

- “Stacks” on page 8-7.
- “Interrupts” on page 8-13.
- “Return from Interrupt” on page 8-48.
- “Set Interrupt” on page 8-62.
- “Type 37: Interrupt” on page 9-60.

# Program Flow Instructions

## No Operation

```
NOP ;
```

### FUNCTION

This instruction causes the DSP's core to perform no operation for one cycle. Program execution continues with the instruction directly following the NOP instruction.

### INPUT

None.

### OUTPUT

None.

### STATUS FLAGS

None.

### DETAILS

Only the core ceases operation for one cycle; the on-chip peripherals continue their respective operations.

### EXAMPLES

```
NOP; /* no operation */
```

### SEE ALSO

[“Type 30: NOP” on page 9-52.](#)

## Idle

```
IDLE ;
```

### FUNCTION

This instruction directs the DSP's core to wait indefinitely in a low-power state until an interrupt occurs. When an interrupt occurs, the DSP's core exits the low-power state, services the interrupt, then continues program execution at the instruction directly following the `IDLE` instruction.

### INPUT

None.

### OUTPUT

None.

### OPTIONS

$\langle imm4 \rangle$  A 4-bit divisor value used to calculate the reduction in frequency of the DSP's internal clock during Slow Idle mode. The Slow Idle rate equals the frequency of the internal clock divided by this value.

Valid values are 0-15.



This option is not available on the 2192 or 2191.

### STATUS FLAGS

None.

### DETAILS

Applications typically use this instruction to implement a low-power standby loop:

```
standby: IDLE(16);
        JUMP standby;
```

## Program Flow Instructions

```
next_instruction;
```

In Idle mode, the DSP's response time to incoming interrupts is one cycle. In Slow Idle mode, the DSP's response time to incoming interrupts slows accordingly. When an incoming interrupt occurs, full recovery from the `IDLE(imm4)` state takes up to `<imm4>` DSP cycles.

Using `IDLE(imm4)` in systems that have an externally-generated serial clock may result in a serial clock rate that is faster than the DSP's reduced internal clock rate. Because of this and the DSP's reduced response time to interrupts, applications must avoid generating interrupts faster than the DSP can service them.

Some DSPs in the ADSP-219x family also support a *sleep mode*, in which the DSP's core and all of its on-chip peripherals enter Idle mode. To invoke sleep mode, the application must program the appropriate bits in the PLL control and I/O clock control registers and use the standard `IDLE` instruction. Exiting sleep mode requires a hardware reset.

### EXAMPLES

```
IDLE;          /* Idle at internal clock's rate */
```

### SEE ALSO

[“No Operation” on page 8-66.](#)

[“Type 31: Idle” on page 9-53.](#)

## Mode Control

ENA	SEC_REG	;
DIS	BIT_REV	
	AV_LATCH	
	AR_SAT	
	M_MODE	
	TIMER	
	SEC_DAG	
	INT	

### FUNCTION

This instruction enables (ENA) or disables (DIS) from one to seven DSP modes in parallel. To enable or disable a mode, this instruction sets (1) or clears (0), respectively, the mode's bit in the mode status register, `MSTAT` (for details, [“Mode Status \(MSTAT\) Register” on page 2-11](#)). The DSP modes are:

- SEC\_REG      Secondary computation register bank (`MSTAT[0]`).
- BIT\_REV      Bit-reversed addressing mode (`MSTAT[1]`).
- AV\_LATCH    ALU overflow status mode (`MSTAT[2]`).
- AR\_SAT      ALU AR register saturation mode (`MSTAT[3]`).
- M\_MODE      MAC integer operand format mode (`MSTAT[4]`).
- TIMER        Timer enable (`MSTAT[5]`).
- SEC\_DAG     Secondary DAG address register bank (`MSTAT[6]`).
- INT          Global interrupts

### INPUT

SEC\_REG, BIT\_REV, AV\_LATCH, AR\_SAT, M\_MODE, TIMER, SEC\_DAG, INT

# Program Flow Instructions

## OUTPUT

None.


## STATUS FLAGS

None.


## DETAILS

You can enable or disable one or more modes in parallel by issuing multiple DIS or ENA instructions on the same instruction line, as in:

```
ENA AR_SAT, ENA M_MODE, ENA AV_LATCH, ENA SEC_REG;
```

 You cannot issue both DIS and ENA instructions on the same instruction line to enable and disable modes in parallel (as in: ENA AR\_SAT, DIS AV\_LATCH, ENA SEC\_DAG;)

As shown in [Table 7-2 on page 7-10](#), changing modes using this instruction, as opposed to register writes or popping the status stack, does not incur any cycles of latency. Latency is the delay, in number of instruction cycles, between the time the mode change instruction executes and the time when the mode change takes effect, such that other instructions can execute operations based on the new value. A latency of 0 means that mode change is available to the instruction directly following the mode change instruction.

 ENA/DIS INT sets or clears bit 5 in the ICNTL register. The write takes effect on the next instruction cycle.

## EXAMPLES

```
/* Switching contexts during an ISR: */  
  
ENA INT;  
IMASK = 0x21A0;  
ENA SEC_REG, ENA SEC_DAG;  
AYO = DM(IO += MO);  
RTI (DB);
```

```
AR = AX0 + AY0;  
DM(IO += M0) = AR;  
  
/* Bit-reversing DAG1 output to memory: */  
  
ENA BIT_REV;  
AY0 = DM(IO += M0);  
AR = AX0 + AY0;  
DM(IO += M0) = AR;  
DIS BIT_REV;
```

### SEE ALSO

- [“Type 18: Mode Change” on page 9-40](#)
- [“MSTAT Mode Control Register” on page 8-4](#)
- [“Enabling Interrupts” on page 8-14](#)
- [“Effect Latencies” on page 8-21](#)

# Program Flow Instructions