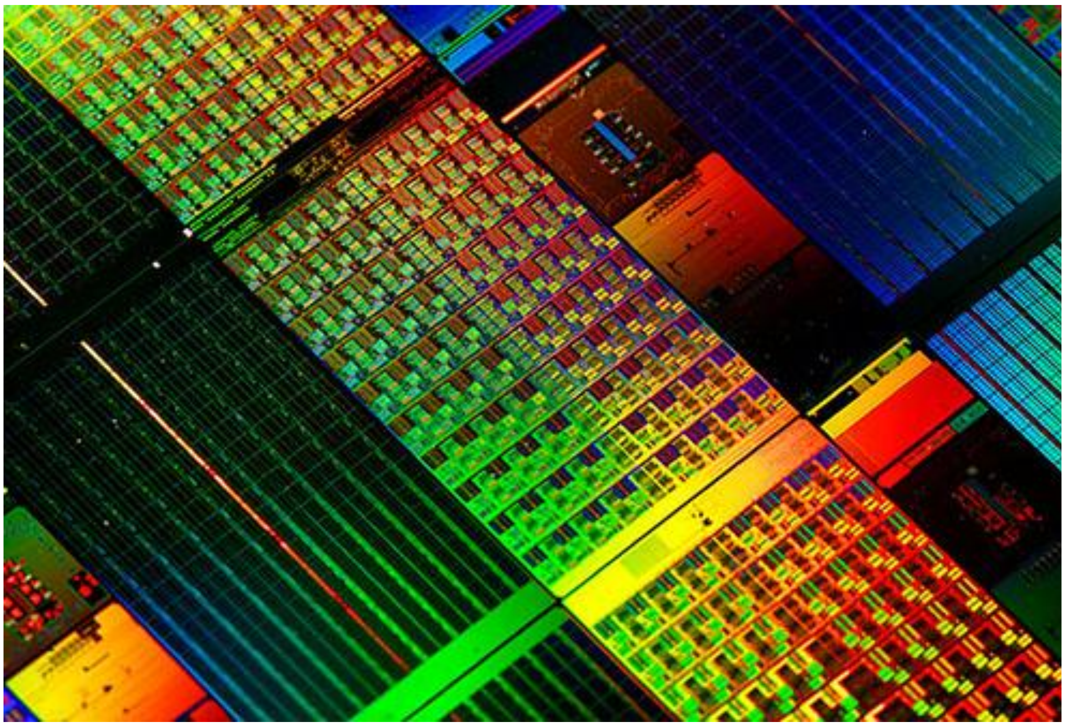


LPDSP32-V3 Hardware Reference Manual

Ver 1.3

System Solutions Co., Ltd.

An ON Semiconductor Company

Copyright © 2014

Table of contents

1. Scope

This manual describes about the hardware aspects of LPDSP32-V3 core and its operations.
It is mainly intended for programmers to understand the core architecture and various kinds of operations supported in brief, so as to make them feel comfortable and at ease to do application coding.

2. Abstract

LPDSP32-V3 is a 32bit fixed point processor, mainly intended for audio DSP algorithms, specially designed for low power embedded systems using a unique retargetable processor design methodology. This DSP is usually programmed using "C" and has a very efficient C compiler which generates optimized assembly code, which is highly comparable to handwritten assembly code. Please note that most of the assembly examples given in this manual are generated by the C compiler, from C code; a few examples are hand-coded in assembly to give a more clear explanation

3. Architecture Overview

LPDSP32-V3 architecture is highly cycle (MCPS) efficient for computational intensive audio algorithms. This DSP is capable of doing two MAC operations, two memory operations (load/store) and two pointer updates in a single cycle. Some of the important features are listed below,

- Highly optimized and efficient “C” compiler.
- Dual Harvard, load store architecture.
- Simple 3-stage pipeline.
- 16M Byte contiguous data memory (DM) space, which is byte(8bit), short(16bit) and word(32bit) accessible. Data memory consists of DMA (8 Mbyte) DMB (4 Mbyte) and DMIO (4 Mbyte), separate memory space is allocated for each.
- Data memory(DM) is Little endian.
- Separate 40M Byte Program memory (PM) space.
- Program memory(PM) is Big endian.
- Two 72bit ALUs (ALU0: performs arithmetic and logical operations, ALU1: performs only arithmetic operations).
- Two 32bit integer/fractional multipliers, capable of performing combination of multiplications multss: signed * signed, multuu: unsigned * unsigned, multsu: signed * unsigned
- 64bit Shifter with signed shift factor (-63…+63)
- Four 64bit accumulators with 8bit overflow (extension bits).
- Four parallel operations support (2 arithmetic and 2 moves and two address update).
- RISC like instruction set, supports two types of instructions (20bit short and 40bit long) suitable for control as well as DSP operations.
- Special addressing modes support (Cyclic and bit reverse addressing).
- Zero overhead looping (hardware loop) nested up to four levels.
- Various control instructions such as conditional unconditional jump, subroutine call and return.
- Power management support (core halt and resume).
- Support for 15 active-high level sensitive interrupts.
- JTAG based On Chip debug interface.
- Hardware wait-state option.
- Support for hardware and software breakpoints.
- Option to set watch points on data memory for store operation

Hardware features:

Feature	LPDSP32-V3 core features
Core Period*	14.28 ns (70 MHz)
Core Area*	91252 µm²
Gates	91.25K gates (inverter{IV} equivalent)
Core Power*	0. 242 mW/Mhz or 2.42 mW @ 10MHz @1.35V

**These figures are pre-layout estimation, assuming generic GSMC 150nm fabrication process under worst case operating conditions.*

3.1 Instruction Pipeline

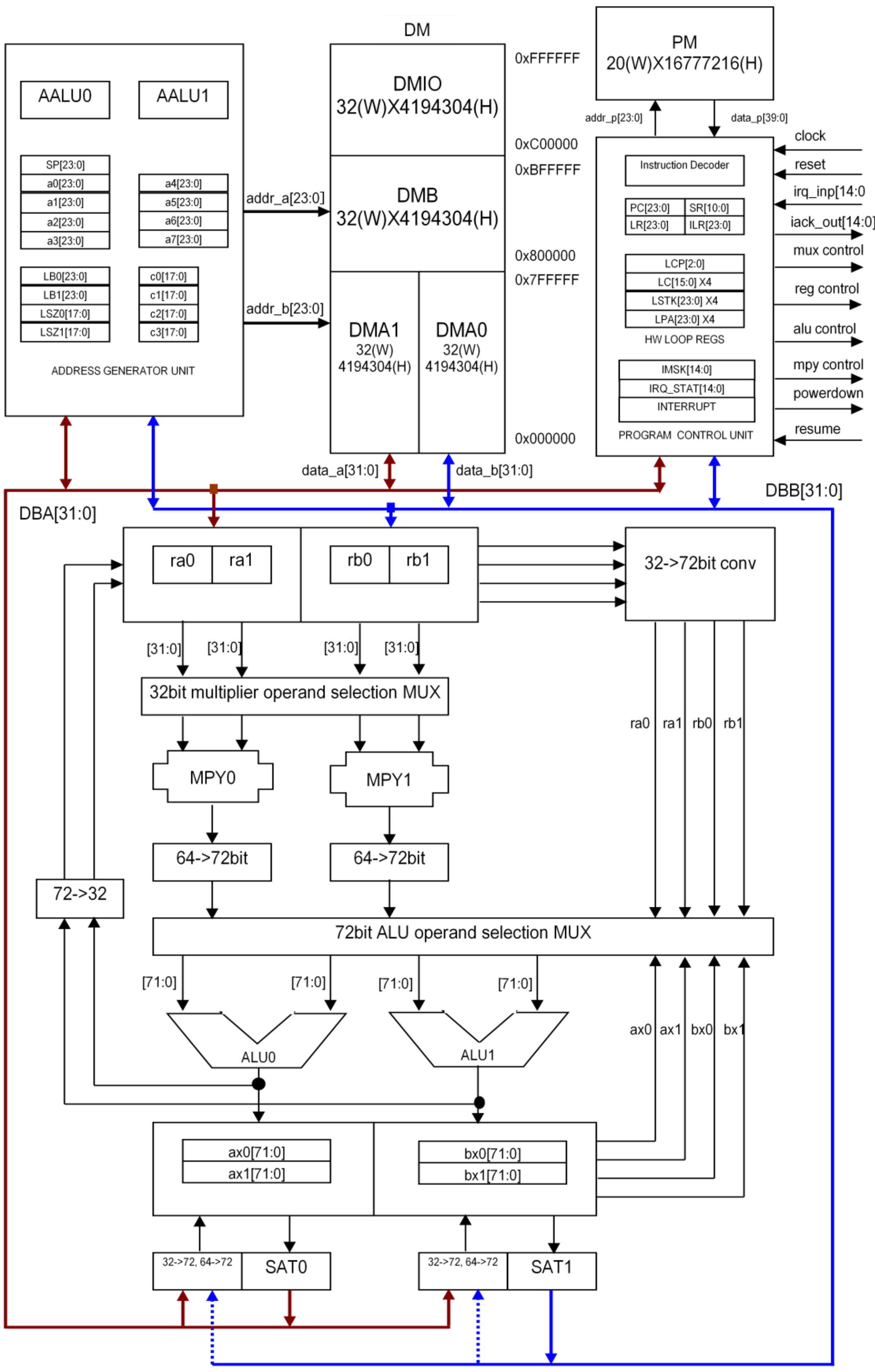
The LPDSP32-V3 has three pipeline stages named F, D and E1. Pipeline diagram is shown below. In the diagram it is assumed that, all instructions are 40bit instructions and stored on even addresses. As shown in the diagram; in cycle-3, instruction A is getting executed, instruction B is being decoded, and instruction C is being fetched from the PM [4]. Also in the same cycle the next PC value (6) is placed on the PM address bus.

	Cycle	1	2	3	4	5	6	7	8
	Next PC	2	4	6	8	10	12	14	16
PC	Instruction								
0	**A**	F	D	E1					
2	**B**		F	D	E1				
4	**C**			F	D	E1			
6	**D**				F	D	E1		
8	**E**					F	D	E1	
10	**F**						F	D	E1

Pipeline action sequence is shown below,

Pipeline stage	Action
F (Fetch)	In this stage, a new instruction is fetched from PM. Program memory address is placed on PM address bus at the end of the previous cycle.
D (Decode)	- In this stage, the instruction that was fetched in the previous cycle is decoded. - The address for load instruction is placed on DM address bus. - Address modification, and address calculation is done in address ALU (aalu0, aalu1) - Unconditional and conditional control instruction modifies the flow of control. - Condition is checked in this stage.
E1 (Execute)	- In this stage ALU, compare, shift and multiply instructions are executed. - Move instructions are executed in this stage. - Address for store instruction is placed on DM address bus and source register data is placed on DM data bus. - For the load instructions, memory places the data on data bus then data is latched into the destination register. - Condition is calculated in this stage.

3.2 Block Diagram



3.3 Hardware Specifications

3.3.1 Core

Item	Specifications
Processing capacity* (MIPS/MOPS)	70 (MIPS) / 280 (MOPS)
Pipeline Stages	3 stages (F, D, E1)
Program Space	16777216(Depth) x 20(Width) [40MByte]
Data Space	DMA: 8388608(Depth) x 8(Width) [8MByte] DMB: 4194304 (Depth) x 8(Width) [4MByte]
IO Space	DMIO: 4194304 (Depth) x 8(Width) [4MByte]
Interrupts	15
Powerdown/Resume	

** Processing capacity is pre-layout estimated figure considering generic GSMC 0.15u typical fabrication process under worst case operating conditions.*

3.3.2 PCU and Hardware loop

Item	Specifications
Program counter (PC)	24 bit
Status register (SR)	11 bit
Link register (LR)	24 bit
Interrupt link register (ILR)	24 bit
Interrupt mask register (IMSK)	15 bit
Interrupt status register (IRQ_STAT)	15 bit
Loop counter pointer (LCP)	3 bit
Loop count register (LC0 ~ LC3)	16 bit x 4
Loop start register (LSTK0 ~ LSTK3)	24 bit x 4
Loop end register (LPA0 ~ LPA3)	24 bit x 4

3.3.3 Address generator units (aalu0 and aalu1)

Item	Specifications
Address registers (A0 ~ A7)	24 bit x 8
Offset registers (C0 ~ C3)	18 bit x 4
Loop start register (LB0 ~ LB1)	24 bit x 2
Loop size register (LSZ0 ~ LSZ1)	18 bit x 2
Stack pointer (SP)	24 bit x 1

3.3.4 Data registers and accumulator

Item	Specifications
Data registers (RA0, RA1, RB0, RB1)	32 bit x 4
Accumulator (AX0, AX1, BX0, BX1)	72 bit x 4 (8 overflow bits)

3.3.5 Functional units, ALU (alu0 & alu1) Multiplier (mpy0 & mpy1)

Item	Specifications
Adder	72 bit x 2
Multiplier	32 bit x 2
Logical operation unit	72 bit x 1
Barrel shifter	64 bit x 1
rounding and saturation unit (sat0, sat1)	72 bit x 2

3.4 Brief architecture description

LPDSP32-V3 has load store architecture; meaning that, before doing any operation first all the operands need to be explicitly moved to the registers from the memory.

Execution units (ALU & multiplier) take the inputs from the registers and perform the operation and write back the result into registers. Then produced result needs to be explicitly moved to the memory.

While designing LPDSP32-V3 a lot of attention was given for the low MHz operation to achieve low power and for the C compiler which can support ANSI C data types.

At the same time the main intention was to get highly optimized code directly compiled by the C compiler without any manual assembly level optimization. In this way the designers can build their applications and bring them to market very quickly, along with maintaining the low power operation.

The architecture block diagram shown earlier on in section [3.2], consists of these main sub blocks:

- ❖Instruction execution unit which consists of:
  - ➣Two multipliers MPY0(33bit x 33bit) and MPY1 (32bit x 32bit)
  - ➣Two Arithmetic and logic units ALU0 and ALU1 (72bit)

  Note: ALU0 can perform arithmetic and logical operations but ALU1 can perform only arithmetic (add/sub) operations.
- ❖Data registers of 32-bit (ra0, ra1, rb0 and rb1)
- ❖Accumulators of 72-bit (ax0, ax1, bx0 and bx1)
- ❖Rounding and saturation units 72bit (sat0 and sat1)
- ❖Data move buses of 32 bit (DBA and DBB)
- ❖Data width conversion blocks (eg. 32bit → 72bit, 72bit → 32bit)
- ❖Address generation unit which consists of:
  - ➣Two address generators (AALU0 and AALU1)
  - ➣Address registers (a0 ~ a7), offset registers (c0 ~ c3)
  - ➣Special addressing registers (LB0, LB1, LSZ0 and LSZ1) and stack pointer (SP)
- ❖Program control unit (PCU) which consists of:
  - ➣Instruction decoder
  - ➣Controller and Interrupt controller
  - ➣Program control registers (PC, SR, LR and ILR)
  - ➣Hardware loop registers (LCP, LC, LSTK and LPA)
- ❖Data memory (DM)
- ❖Program memory (PM)

4. Instruction execution units

4.1 Multipliers

There are two multipliers (mpy0 and mpy1) in LPDSP32-V3, which can perform 32x32 bit multiplication in a single cycle. Multipliers can read their operands from any of the data registers (ra, rb) and perform the multiplication. A 64-bit result is written to any of the accumulators (ax, bx).

The multiplier mpy0 is capable of signed, unsigned and mixed sign operation with the below multiplication modes.

- ➛32-bit Signed x 32-bit Signed
- ➛32-bit Unsigned x 32-bit Unsigned
- ➛32-bit Signed x 32-bit Unsigned

However mpy1 can perform only 32-bit Signed x 32-bit Signed multiplication.
MAC units (acc0 and acc1) are present after the multiplier (mpy0 and mpy1) to do the multiply accumulate (MAC) operation. Units acc0 and acc1 performs the add/sub operation after the multiplication. Complete MAC operation i.e. multiplication and addition happens in a single clock cycle. For doing MAC0/MAC1 operations, one of the input required for doing MAC operation can be selected or swapped either from mpy0 or mpy1, this is done by swap bit.
Multiplier is always accessed via multiply accumulate (MAC) operation.
MAC can be used in 4 modes; pass, negate, add and sub.
To do simple multiplication MAC unit is used in pass mode. In the pass mode other operand of the MAC is zero, so that multiplier result is passed directly to the output.

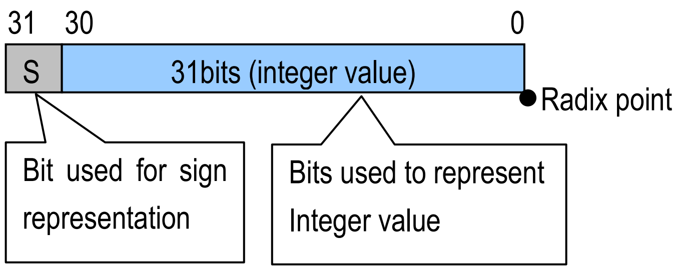
More importantly, multipliers are capable of performing integer or fractional multiplication.
This mode is selected by setting/clearing “im” bit in the status register.

Fractional mode:	im = 0
Integer mode:	im = 1

To understand about this mode of multiplication, a brief explanation is given in the next page.

4.1.1 Integer data type

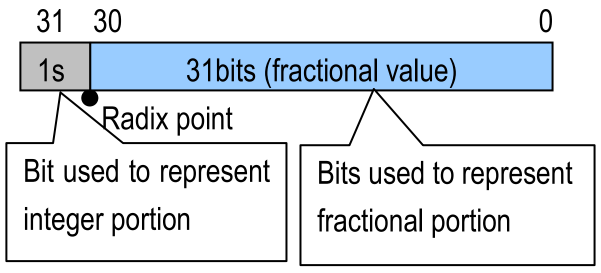
Integer data type is inherently represented as a signed 2's complement number.
Here the MSB (most significant bit) is defined as sign and radix point lies at the bit[0] position.
The data range for N bit 2's complement integer is (-2^(N-1) to (2^(N-1)-1).
The data range for 32-bit integer number is (0x8000_0000) -2,147,483,648 to (0x7FFF_FFFF) +2,147,483,647


31
30
0
S
31bits (integer value)
Radix point
Bit used for sign representation
Bits used to represent Integer value

4.1.2 Fractional data type

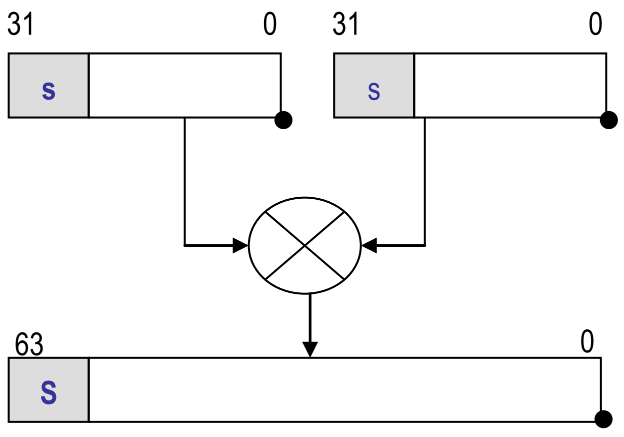
The fractional data type is also represented as a signed 2’s complement number.
Here the MSB (most significant bit) is defined as signed value and radix point lies at the bit[30] position. The data range for N bit fractional number is (-1.0 to (1-2^(1-N)).
The data range for 32bit fractional number is (0x8000_0000) -1 to (0x7FFF_FFFF) +0.99999999953433 including '0.0', and it has a precision of (4.657 x 10^-10)

This format is commonly referred to 1.31 or Q31 format.


31
30
0
1s
31bits (fractional value)
Radix point
Bit used to represent integer portion
Bits used to represent fractional portion

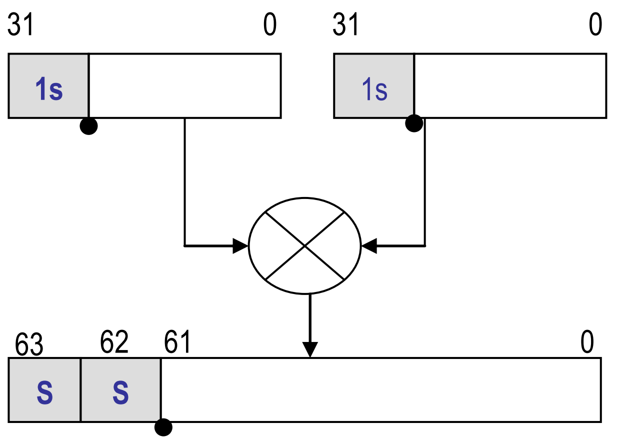
Note that, with the exception of multiplier, ALU operates identically on integer and fractional data. Namely, an addition of two integers will yield the same result (binary number) as the addition of two fractional numbers. The only difference is how the result is interpreted by the user.
Different multiplication modes selection is made by the im bit in status register srmode0, and it must be set accordingly ('0' for Fractional mode, '1' for Integer mode).
This is required because of the implied radix point used by LPDSP32-V3 fractional numbers.
In integer mode multiplying two 32bit integers produces a 64bit integer result.
However, multiplying two 1.31 numbers produces a 2.62 result.
Since LPDSP32-V3 uses 1.31 format for the accumulators, the DSP multiply in fractional mode also includes a left shift of 1bit to keep the radix point properly aligned.
Very important to note that, this feature reduces the resolution of the DSP multiply to 2^(-30) but has no other effects on the computation.

Integer multiplication:



Fractional multiplication:

● Radix point



4.2 Arithmetic and logic unit

There are two arithmetic and logical units (alu0 and alu1) in the LPDSP32-V3.

alu0 is a 72-bit wide arithmetic and logical unit which is capable of doing,

- Addition
- Subtraction
- Division
- Logical operations
- Shifting
- Comparison
- Min & Max. etc.

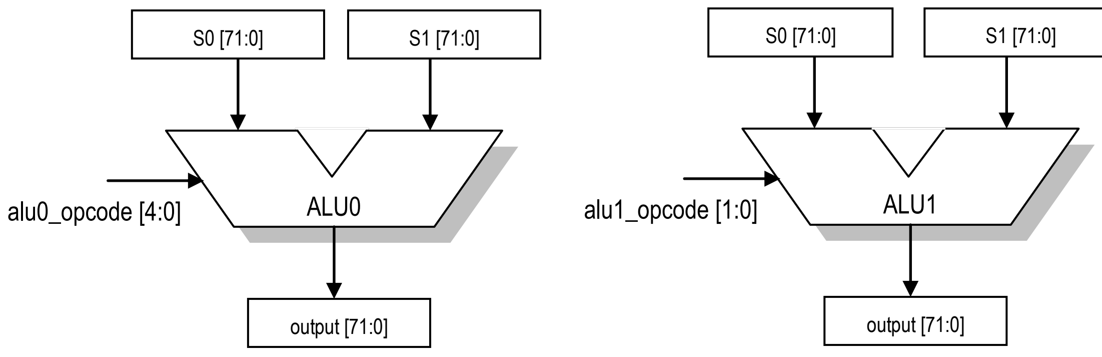
For the single operation always alu0 is used. Unit alu0 can affect the status register negative(N) and zero (Z) flags only for the few instructions like, compare signed (cmp), compare unsigned (cmpu), division (div) and bit test (bt).

Data required for the alu0 operation can come from any of the data registers (ra, rb) or accumulators (ax,bx) and output result can be written to any of the data registers or accumulators.

However, alu1 is also 72-bit wide and used only when parallel arithmetic (addition/subtraction) operations are used. This unit does not affect any status register flags.

Since alu1 is used only in parallel operations; source-0 operand can be selected from data register (rb) and source-1 operand can be selected from any of the data registers or accumulators.

Result of the alu1 is always written to accumulator (bx).



alu0 is used for single as well as parallel operations; however alu1 is used only for parallel operations. Instructions supported by alu0 and alu1 are mentioned on the next page.

4.2.1 alu0 instructions

These are the instructions supported on alu0.

Sr.	operations	immediate	Single operand	Double operand	No result	SR flag
1	add (+)	▨		▨		
2	sub (-)			▨		
3	and (&)	▨		▨		
4	or (|)	▨		▨		
5	xor (^)	▨		▨		
6	asl	▨		▨		
7	asr	▨		▨		
8	max			▨		
9	min			▨		
10	move immediate (8bit)	▨				
11	div			▨		▨
12	cmp	▨		▨	▨	▨
13	cmpu	▨		▨	▨	▨
14	negate		▨			
15	nrm		▨			
16	abs		▨			
17	lsl	▨		▨		
18	lsr	▨		▨		
19	sxtd		▨			
20	sxtd8		▨			
21	sxtd16		▨			
22	mask8		▨			
23	mask16		▨			
24	bi	▨		▨		
25	br	▨		▨		
26	bs	▨		▨		
27	bt	▨		▨	▨	▨

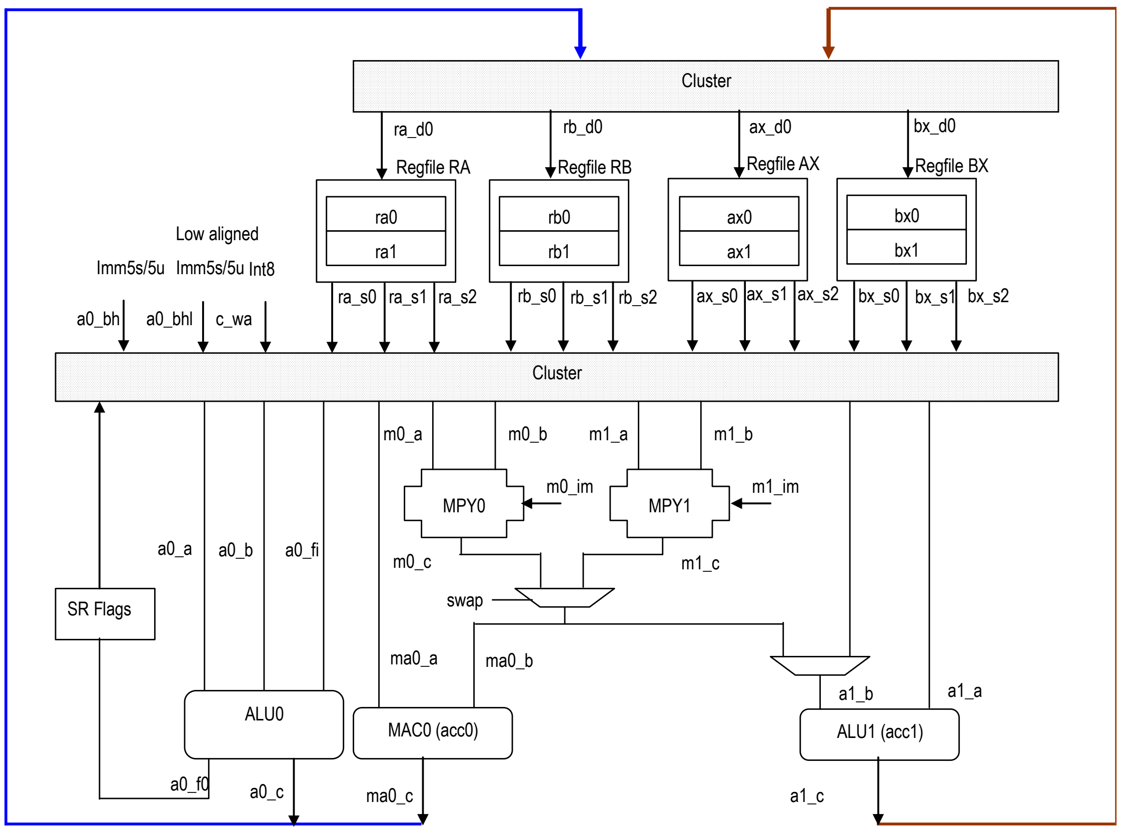
4.2.2 alu1 instructions

These are the only instructions supported by alu1.

Sr.	operations	immediate	Single operand	Double operand	No result	SR flag
1	add (+)	▨		▨		
2	sub (-)			▨		

4.2.3 Dual ALU and MAC data path

To understand the data flow for the dual ALU and MAC operation refer the figure shown below.



4.2.4 Explanation of alu0 instructions

In this section some of the instructions and their operation are described briefly for understanding. Before doing any operation in alu0, first the data is converted to 72-bit. ALU0 can perform operation on data registers of width 32-bit as well as accumulator registers of width 64-bit.

Some of the instructions are self explanatory like add, sub, and, or and xor. Therefore, few are explained here. For a detailed explanation of all the instructions please refer the instruction manual.

4.2.5 Shift instructions

Both logical and arithmetic shift operations are supported. Depending on its width, shift value can be stored in data register (ra, rb) or accumulator (ax, bx) to do shift operation. However, shift factor can be an immediate signed/unsigned value of (7-bit) or it can be an indirect value stored in any of the registers (ra, rb, ax, bx).

After the shift operation result can be stored in data register (ra, rb) or in the accumulator (ax, bx). Shift instructions do not generate any flags.

Types of the shift instructions are mentioned below,

- Logical shift left (lsl)
- Logical shift right (lsr)
- Arithmetic shift left (asl)
- Arithmetic shift right (asr)

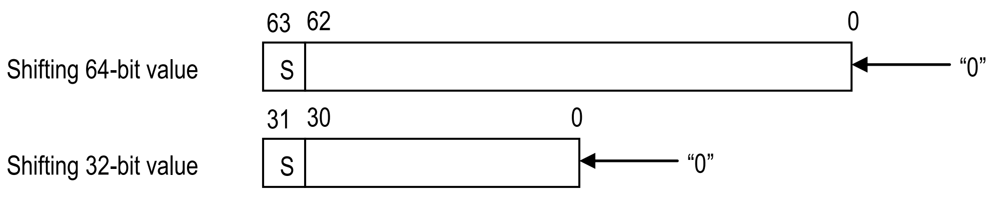
✍ **Important note:**

Maximum allowed range for the shift factor is -63 to +63. If shift factor is < -63, it is treated as zero and no shift operation is performed and input value is passed to output.

If the shift factor exceeds the above range, result is not guaranteed.

4.2.5.1 Logical shift left (lsl)

This is the logical left shift operation.



In the logical shift operation if the sign bit is "1" it is extended to 72 bits. And if the shift factor is a negative value then instead of doing a logical shift left it performs a logical shift right operation.

Examples:

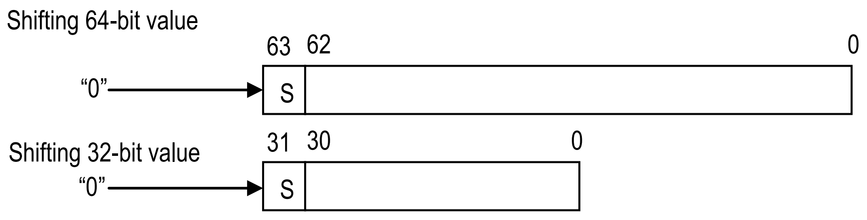
ax0 = lsl(ax0, ra0) //indirect logical shift left on 64-bit long register

ra0 = lsl(ra0, rb1) //indirect logical shift left on 32-bit data register

ax0 = lsl(bx0, 15) //immediate shift left

4.2.5.2 Logical shift right (lsr)

This is the logical right shift operation



If the shift factor is a negative value it performs a logical shift left operation.

Examples:

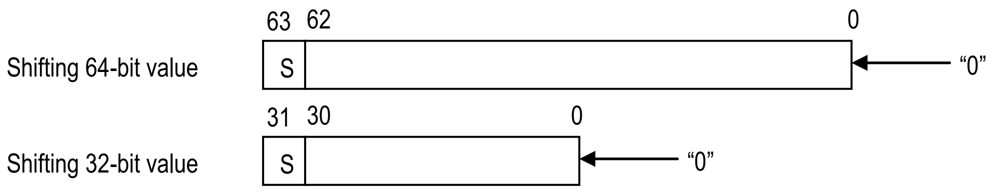
ax0 = lsr(ax0, ra0) //indirect logical shift right on 64-bit long register

ra0 = lsr(ra0, rb1) //indirect logical shift right on 32-bit data register

ax0 = lsr(bx0, 15) //immediate shift right

4.2.5.3 Arithmetic shift left (asl)

Arithmetic shift left operation is almost same as logical shift left operation, only difference is that the sign bit is not extended further in this case.



If the shift factor is a negative value it performs an arithmetic shift right operation.

Examples:

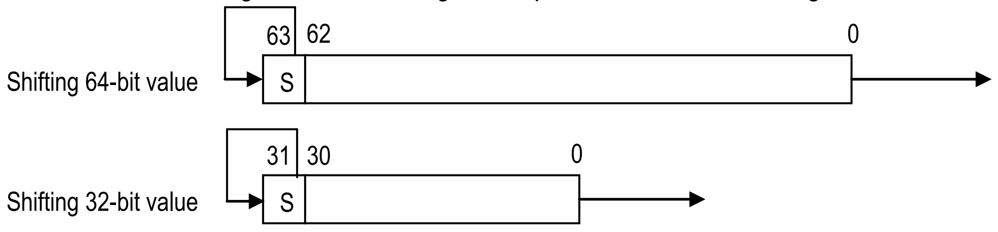
ax0 = asl(ax0, ra0) //indirect arithmetic shift left on 64-bit long register

ra0 = asl(ra0, rb1) //indirect arithmetic shift left on 32-bit data register

ax0 = asl(bx0, 15) //immediate arithmetic shift left

4.2.5.4 Arithmetic shift right (asr)

In arithmetic shift right instruction, sign bit is pushed and shifted to right as shown below.



If the shift factor is a negative value it performs an arithmetic shift left operation.

Examples:

bx0 = asr(ax0, ra0) //indirect arithmetic shift right on 64-bit long register

rb0 = asr(ra0, rb1) //indirect arithmetic shift right on 32-bit data register

bx0 = asr(bx0, 10) //immediate arithmetic shift right

4.2.6 MAX and MIN instructions

These instructions find the maximum value or minimum value between the two operands. To do this operation operands can be stored in (ra, rb, ax, bx) and result is written into any of these registers (ra, rb, ax, bx).

Examples:

ax0 = max(ax0, bx0) //finds the maximum between two 64-bit operands

ra0 = max(ra0, ra1) //finds the maximum between two 32-bit operands

bx0 = min(bx0, bx1) //finds the minimum between two 64-bit operands

ra0 = min(ra0, ra1) //finds the minimum between two 32-bit operands

4.2.7 Sign extension instructions

Three sign extension instructions are provided in LPDSP32.

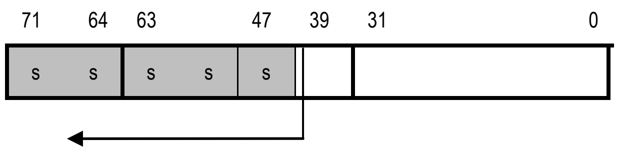
4.2.7.1 Sign extension of 64bit (sxtd)

This instruction will extend the sign bit of long long data.

4.2.7.2 Sign extension of 8bit (sxtd8)

This instruction will be helpful in transferring a 32bit signed variable to a 8bit signed variable.

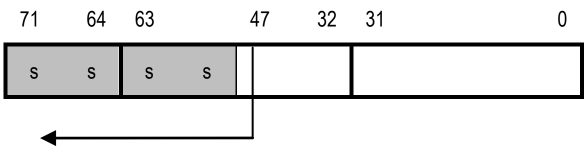
Example:



4.2.7.3 Sign extension of 16bit (sxtd16)

This instruction will be helpful in transferring a 32bit signed variable to a 16bit signed variable.

Example:



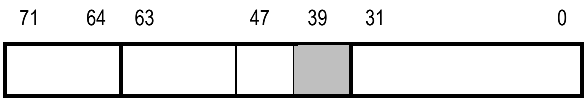
4.2.8 Mask instructions

Two mask instructions are provided in LPDSP32.

4.2.8.1 Mask instruction for 8bit (mask8)

This instruction will extract 8bits from a 32bit register or from a high word of accumulator.

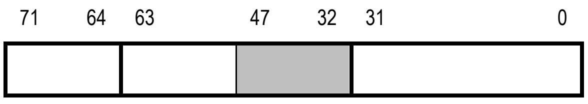
Example:



4.2.8.2 Mask instruction for 16 bit (mask16)

This instruction will extract 16bits from a 32bit register or from a high word of accumulator.

Example:



4.2.9 Bit manipulation instructions

There are four bit manipulation instructions (bs, br, bi and bt), which operate on both 32-bit and 64-bit registers. Result can be written to (ra, rb, ax, bx) registers. Index for the bit manipulation can be an indirect value or a 6-bit immediate index.

Except the bit test (bt) instruction other bit manipulation instructions do not generate status flags.

✍ **Important Note:**

Maximum index value should be within the range (0 to 63) for guaranteed results.

4.2.9.1 Bit set (bs)

This instruction sets the bit indicated by the index in 32-bit or 64-bit register field, where the index can be either an immediate or an indirect value.

4.2.9.2 Bit reset (br)

This instruction resets the bit indicated by the index in 32-bit or 64-bit register field, where the index can be either an immediate or an indirect value.

4.2.9.3 Bit invert (bi)

This instruction inverts the bit indicated by the index in 32-bit or 64-bit register field, where the index can be either an immediate or an indirect value.

4.2.9.4 Bit test (bt)

This instruction tests the bit indicated by the index in 32-bit or 64-bit register field, where index can be immediate or indirect value. This does not produce any result but generates status flag.

4.2.10 Division instruction

LPDSP32-V3 supports division operation which can be performed on short (32bit) and long (64bit) operands. This instruction is implemented by non restoring division algorithm on positive integers.

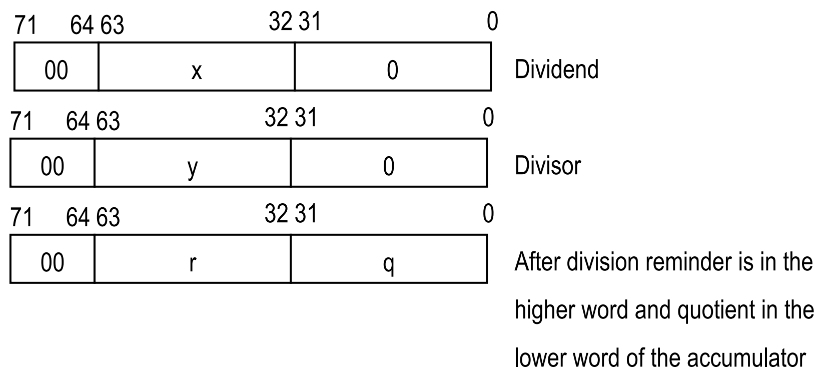
Inherently, the non restoring division algorithm consumes less hardware.

This is a multi-cycle instruction; number of cycles taken for the division operation depends on the step value.

✍ **Important note:** It is not recommended to use this instruction where cycle efficiency is desired.

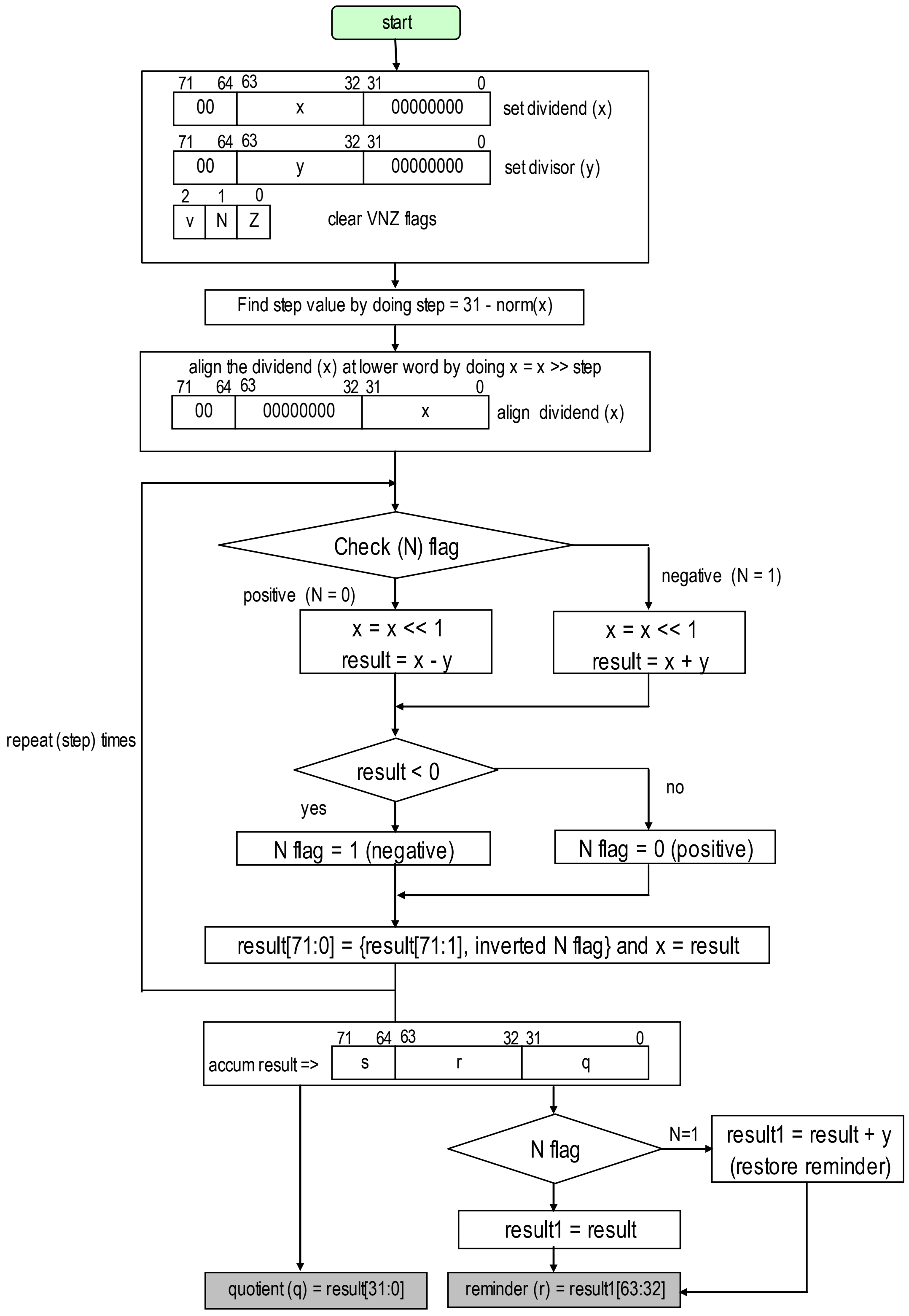
4.2.10.1 Division on 32bit integers

While doing division operation on 32bit integer, dividend (x) and divisor (y) are aligned at higher word of the accumulator and extended with the zeros as below



71 64 63 32 31 0
00 x 0
Dividend
71 64 63 32 31 0
00 y 0
Divisor
71 64 63 32 31 0
00 r q
After division reminder is in the higher word and quotient in the lower word of the accumulator

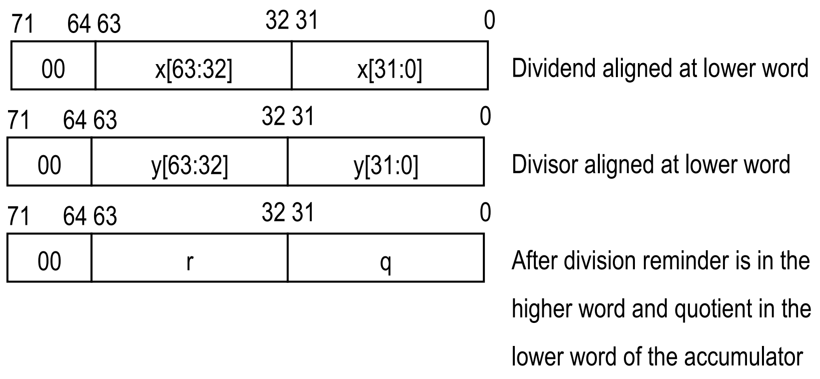
4.2.10.2 Flow chart for 32-bit integer division



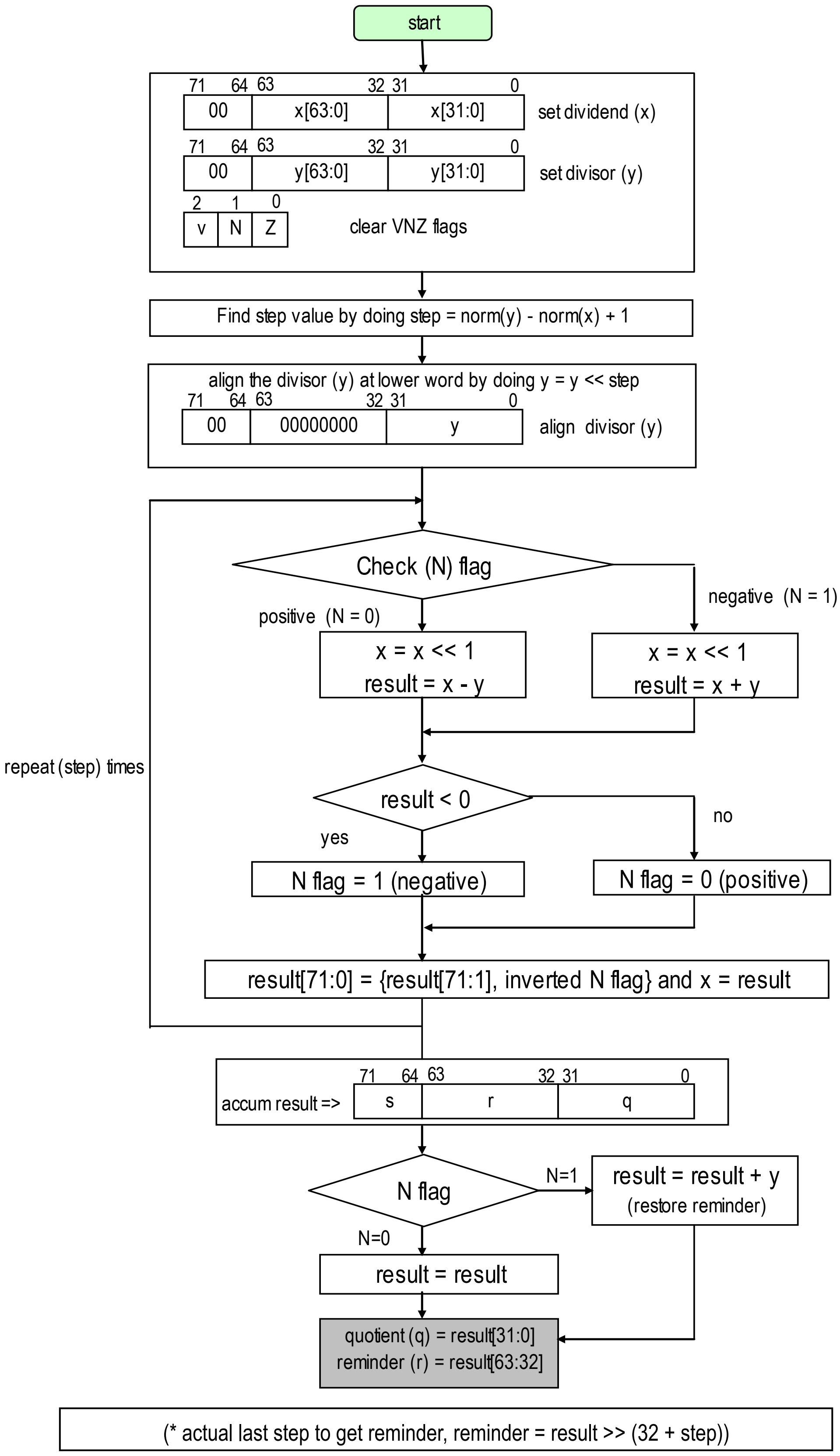
4.2.10.3 Division on 64-bit long long variables

This operation is performed by calling inbuilt c function called "div64_pos_called"

While doing division operation on 64bit long long variables, dividend (x) and divisor (y) are aligned at lower word and extended with the zeros and result is unsigned long long


71 64 63 32 31 0
00
x[63:32]
x[31:0]
Dividend aligned at lower word
71 64 63 32 31 0
00
y[63:32]
y[31:0]
Divisor aligned at lower word
71 64 63 32 31 0
00
r
q
After division reminder is in the higher word and quotient in the lower word of the accumulator

4.2.10.4 Flow chart for 64-bit long long division



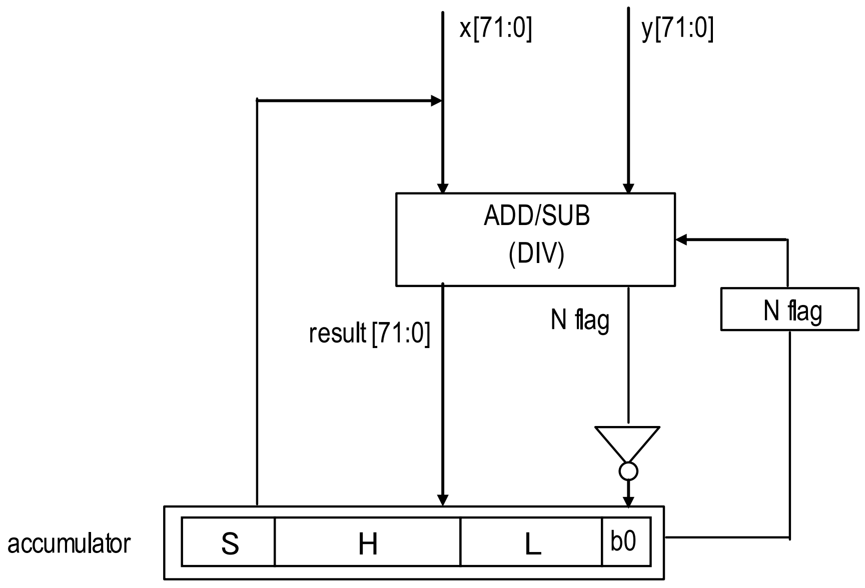
4.2.10.5 Hardware implementation of division

Division operation is controlled by hardware loop instruction and utilizing negative (N) flag. Hardware implemented is only addition and subtraction controlled by N flag.

Example macro for doing division operation:

```
axs1 = [0]
ra0 = 31; ae1 = zero
ra1 = nrm(ax1) //normalization for calculating step
axs0 = [4]
ra0 = ra0 - ra1 //step value
lp [ra0] 2 //repeat until step value
nop; flags = zero
ax1 = asr(ax1,ra0); ae0 = zero //shift divisor x = x >> step
ax1 = div(ax1,ax0) // division operation
retdb
[8] = al1
nop; ra0 = zero
```

Below is the hardware logic for division



4.2.11 Normalization instruction

This instruction detects the decimal point to be shifted and computes the shift factor.

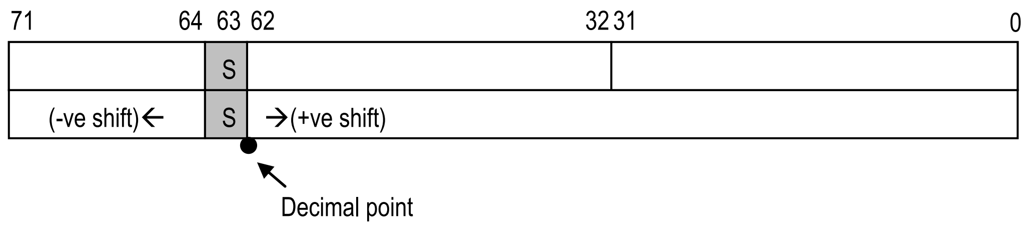
To perform the normalization, value can be in data register or accumulator. Normally normalization operation is performed in the accumulator registers (ax, bx).

After the normalization operation, detected shift factor can be stored in data register or accumulator. This shift factor can be a positive value or a negative value. Positive shift factor means shift left the value by shift factor and negative shift factor means shift right the value by shift factor.

This operation counts leading zeros in case of positive numbers excluding sign bit[63], in case of negative numbers counts leading ones. If first ‘0’ or ‘1’ is detected in bit[62] to bit[0] of the accumulator, shift factor is positive. If first ‘0’ or ‘1’ is detected in bit[71] to bit[64] of the accumulator, shift factor is negative.

Example:

```
1. ax0 = 72’h000000000004000000, nrm(ax0) = 36
2. ax0 = 72’h000000008000000000, nrm(ax0) = 23
3. ax0 = 72’h002000000000000000, nrm(ax0) = 1
4. ax0 = 72’h004000000000000000, nrm(ax0) = 0
5. ax0 = 72’h008000000000000000, nrm(ax0) = -1
6. ax0 = 72’h080000000000000000, nrm(ax0) = -5
7. ax0 = 72’h400000000000000000, nrm(ax0) = -8
8. ax0 = 72’hFFFFFFFFFFFFFFFFFF, nrm(ax0) = 63
9. ax0 = 72’h7FFFFFFFFFFFFFFFFF, nrm(ax0) = -8
```

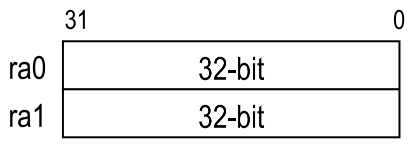


4.3 Data registers

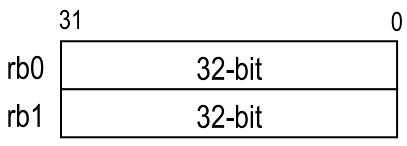
Instead of a single common register file, LPDSP32-V3 implements separate distributed register files for data registers, accumulators, address registers etc.

There are two data register files RA and RB. Register file RA consists of two 32-bit registers [ra0, ra1] and register file RB consists of two 32-bit registers [rb0, rb1]. In total there are 4 data registers.

Data registers are used for storing the operands for ALU and multiplier also for loading/storing the value from/into the memory.



Register file RA

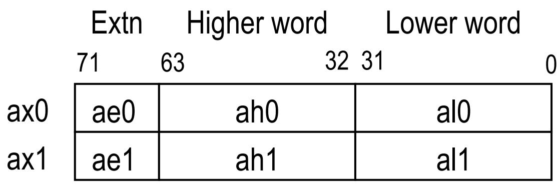


Register file RB

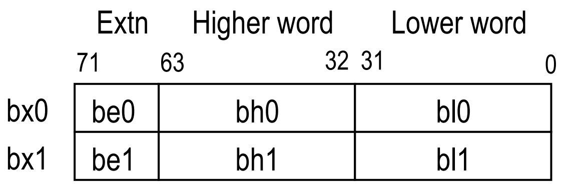
4.4 Accumulator

Similarly for the accumulator there are two register files AX and BX. Register file AX consists of two 72-bit registers [ax0, ax1] and register file BX consists of [bx0, bx1]. Each 72-bit accumulator has sub components which can be accessed separately as shown in the figure. There are 8 overflow bits which permit up to 256 consecutive sums of product (MAC) operations without generating any overflow.

Accumulators are used for storing the ALU results and loading/storing the value from/to the memory.



Register file AX



Register file BX

When an immediate value needs to be assigned to the accumulator, there is an option to keep it either in the higher word or the lower word of the accumulator. To keep the value in the lower word of the accumulator suffix “L” is used next to the value. However when there is no suffix, value is assigned to the higher word of the accumulator.

Example:

```
ax0 = 123L      //value is assigned to the lower word
ax0 = 123       //value is assigned to the higher word
```

4.5 Rounding and saturation units

There are two rounding and saturation units (sat0 and sat1) in the LPDSP32-V3.
Unit sat0 is used for accumulator (ax0, ax1) and sat1 is used for accumulator (bx0, bx1).
When a 72-bit value needs to be moved from the accumulator, saturation unit converts the 72-bit value to a 32-bit value or a 64-bit value depending on the type of move i.e. short move or long move.
While converting a 72-bit value to a 32-bit or a 64-bit value it requires saturation and rounding.

4.5.1 Operations in sat0

- A 72-bit value is converted to either a 32-bit or a 64-bit value.
- While extracting a 32-bit value from a 72-bit value; scaling (divide by /2), rounding and saturation operations are performed.
- A 64-bit value can be extracted with or without doing scaling and saturation operation.
- Sub components (lower word, higher word and extn.) of the accumulator are selected here.
- Rounding and saturation modes can be selected by setting the srmode bits in the status register (SR).

Example:

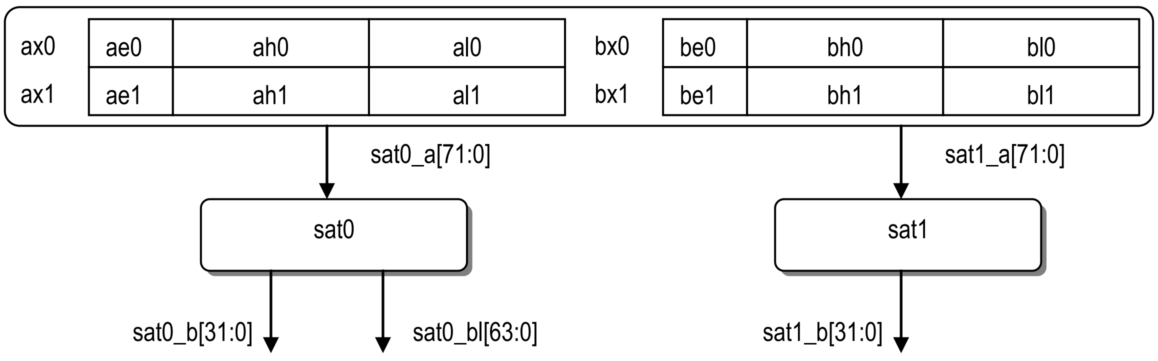
r = 1 //rounding is enabled

s = 1 //saturation is enabled

- Scaling mode is selected by the instruction.

4.5.2 Operations in sat1

- A 72bit value is converted to a 32-bit value.
- While extracting a 32-bit value from a 72-bit value; scaling (divide by /2), rounding and saturation operations are performed.
- Sub components (lower word, higher word and extension) of the accumulator are selected here.
- Rounding and saturation modes can be selected by setting the srmode bits in the status register (SR).
- Scaling mode is selected by the instruction.



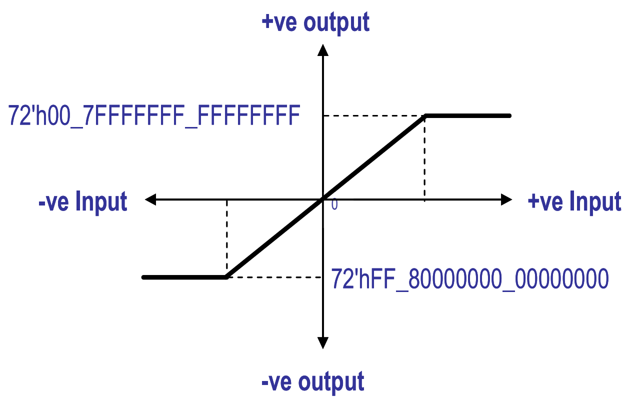
4.5.3 Saturation

Saturation means, if the value is greater than the maximum value that can be represented, it is converted back to its maximum and if the value is less than the minimum value that can be represented, it is converted back to its minimum. Saturation function can be enabled by setting s bit of srmode, which is part of the status register (SR) to '1'.

Saturation function macro:

```
if (s == 1) //saturation is set
  if ( accum[71:0] < 72'hFF_80000000_00000000) //more negative
     accum[71:0] = 72'hFF_80000000_00000000
  else if (accum[71:0] > 72'h00_7FFFFFFF_FFFFFFFF) //more positive
     accum[71:0] = 72'h00_7FFFFFFF_FFFFFFFF
  else
     accum[71:0] = accum[71:0]
```

Therefore when the saturation bit (s) is set high, value in the accumulator is clipped if it exceeds above the range. To understand the saturation operation diagram is shown below,



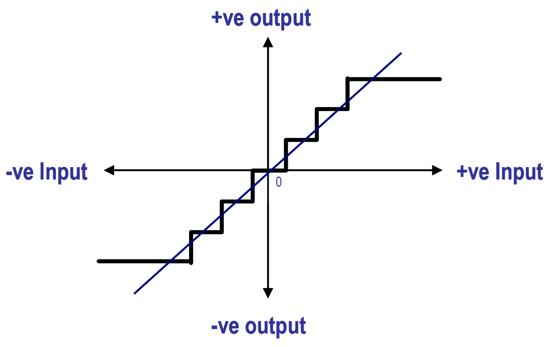
4.5.4 Rounding

Rounding away from zero (towards ∞) algorithm is applied for the LPDSP32-V3. This is a simple rounding method to perform rounding operation on a fractional value to bring it to an integer value which is away from zero. Note that when the fractional component of a full-precision value is precisely ½ , the round away from zero algorithm will introduce a statistical bias into the output time series because these borderline cases are always rounded towards (+∞) for positive full-precision values and towards (-∞) for negative full-precision values.
Rounding function can be enabled by setting r bit of srmode, which is part of the status register (SR) to ‘1’.

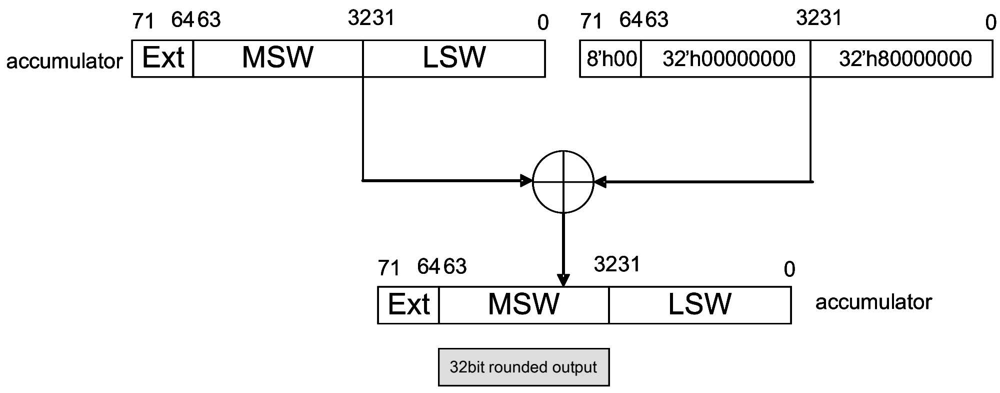
Rounding function macro:

```
if (r == 1) //rounding is set
accum[71:0] = accum[71:0] + 72'h0000000080000000;
```

To understand the rounding operation, diagram is shown below.

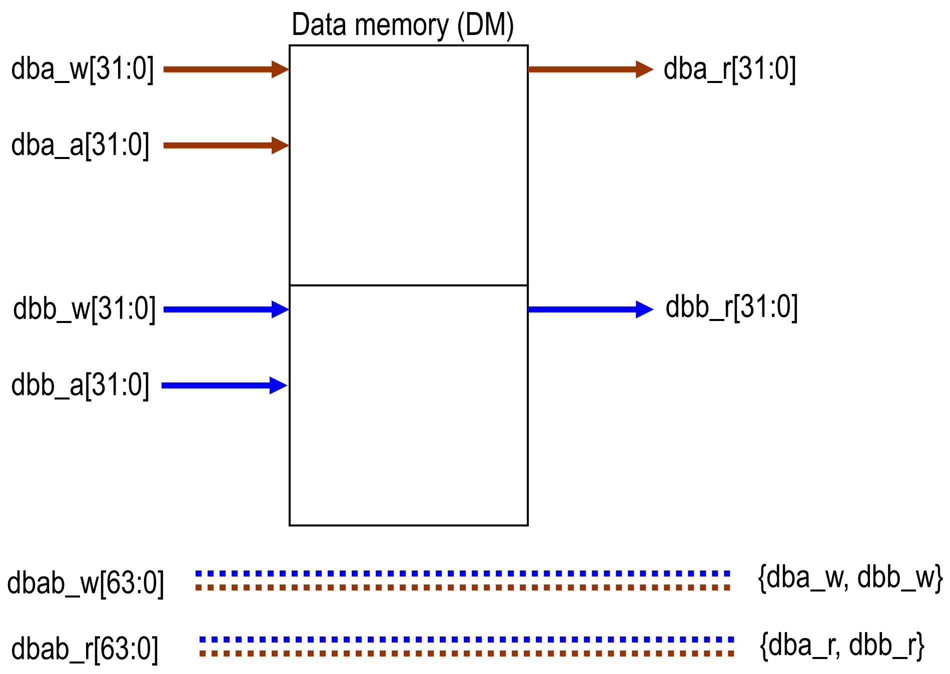


Also rounding mechanism is shown below,



5. Data move buses

There are two 32-bit data move buses; data bus a (dba) and data bus b (dbb). These are used to move the data almost from any register to any register of different data widths.
These data buses (dba and dbb) connect all the registers and the data memory.
There are separate data buses for the read and write. For the data memory (DMA), data read bus is (dba_r) and data write bus is (dba_w). Similarly for the data memory (DMB), data read bus is (dbb_r) and data write bus is (dbb_w). There are two 24-bit address buses. Address bus dba_a is used for addressing DMA and dbb_a is used for addressing DMB, same address bus is used for read/write operation.
All types of data transfers like move and load/store operations make use of either or both of these data move buses. Since the move and load store operation can be done in parallel with the arithmetic and MAC operation, the required data needed for the functional blocks is transferred and moved through these buses.
To transfer a single 32-bit data or data which is less than 32-bit, data bus dba or dbb is used depending on the type of move and the register used for the move operation.
And to transfer 64-bit data, both the data buses dba and dbb are used in a combined form dbab. Data bus dbab does not exist, but it is virtually formed by combining dba and dbb buses.
This dbab data bus is used to move data from accumulator to accumulator or loading/storing long 64-bit data to data memory.



6. Data width conversion

In the different sections of the LPDSP32-V3 core there are various registers of different data widths.

Data move bus dba and dbb is used for moving the data from source to destination.

For the move operation the source and destination can be a register or memory.

Therefore data conversion is very much necessary when data width of the source and destination is not same, for example when:

- Moving the data from one register to other register of different data width.
- Moving the data to/from accumulator from/to data register or memory.
- Fetching the operands from 32-bit data registers for doing the ALU and MAC operation.
- Storing the result of ALU operation to data registers.

6.1 Conversion for register to register move

When moving the data from a register of width (< 32-bit) to a 32-bit register through one of data buses, data is positioned and aligned to the LSB side. If the value is signed, sign bit is extended till 32-bits, else in case of an unsigned value, zeros are padded on the MSB side.

Example:

ra0 = a1 //a1 (24-bit) assigned to ra0 (32-bit), since it is an unsigned value zeros are padded

However, when moving data from a register of width 32-bit to a register of width (< 32-bit), through one of the data buses, value is extracted from LSB and remaining bits on the MSB side are simply discarded.

Example:

a0 = ra1 //ra1 (32-bit) assigned to a0 (24-bit), 8 MSB bits are discarded.

6.2 Conversion for a move from move bus to accumulator

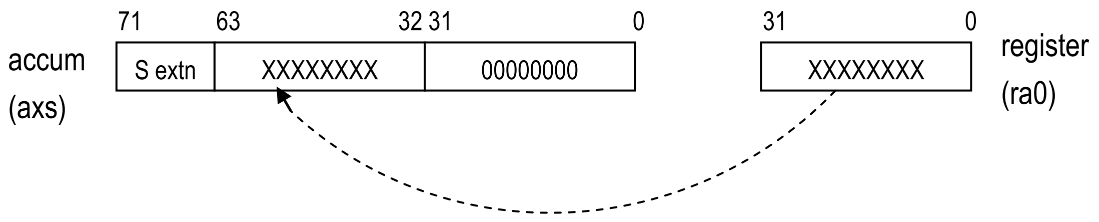
When the destination is a complete accumulator (axs, bxs) and the source is 32-bit, the operation being a memory load or a register move,

Example:

axs0 = ra0 //assigning 32-bit register to accumulator

axs1 = [a0 + c0] //indirect load 32-bit from data memory to accumulator

This assignment uses only one of the data move buses depending on the destination accumulator. The 32-bit value of the source is converted to complete 72-bit by sign extending with 8-bits and padding 32 zeros on the LSB side.



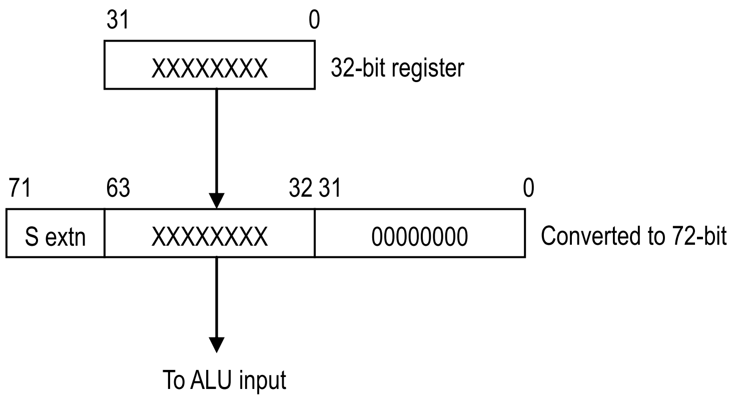
6.3 Conversion for a move from accumulator to move bus

When moving the data from a 72-bit accumulator to data move bus (dba, dbb) conversion happens in the rounding and saturation units (sat0, sat1), which have been explained in the earlier section.

6.4 Conversion for ALU inputs

Arithmetic and logical units (alu0, alu1) have 72-bit input and 72-bit output. When data operands are fetched from the 32-bit data registers (RA, RB), first they are converted to a 72-bit value.

For the conversion 32-bit value is sign extended to 8-bits and 32 zeros are padded on the LSB side.

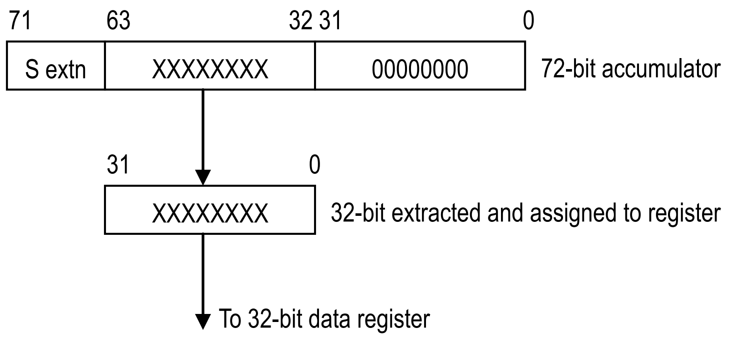


Another conversion is required for the multiplier output to MAC input. When multiplication is done in fractional mode (im = 0 default mode) a 64-bit product is generated, which is sign extended to 8-bits and shift left by 1 bit to get the correct result.

When multiplication is done in integer mode (im = 1) a 64-bit product is generated and sign extended to 8-bits and output is the result itself.

6.5 Conversion for ALU outputs

After performing arithmetic and logical operations in alu0, output is available in the accumulator (AX, BX) and result is assigned to one of the data registers (RA, RB). Here conversion is performed. For the conversion, simply higher word of the accumulator is extracted and assigned to the 32-bit data register. Remaining 8-bit extension part and LSB side lower word are discarded.



7. Address generation units

Since LPDSP32-V3 has load/store architecture, data is transferred from memory to register and from register to memory. The load/store operation is part of a move operation.

There are two address generation units, called as address ALUs, (aalu0) and (aalu1).

Address generation unit generates address for the data memory to perform load/store operation.

Address ALU (aalu0) is used to generate address for the complete data memory (DM) space including DM-A, DM-B and DMIO. In case of dual load/store operation address ALU (aalu1) is used to generate address for the data memory DM-B space.

Address bus dba_a[23:0] is used to address complete data memory (DM) address space.

However address bus dbb_a[23:0] is used only for dual 32 bit data access.

Complete address space for the data memory is (0x000000 ~ 0xFFFFFF), 16777216 Bytes i.e. 16M Bytes. This address space is further divided in to 3 parts for the 3 different memories.

LDMA:

Called as long data memory-A; allocated address space is (0x000000 ~ 0x7FFFFF) 8MB.

Used for long long (64-bit), byte (8-bit), short (16-bit) and int (32-bit) data storage.

DMB:

Called as data memory-B; allocated address space is (0x800000 ~ 0xBFFFFF) 4MB

Used for byte (8-bit), short (16-bit) and int (32-bit) data storage.

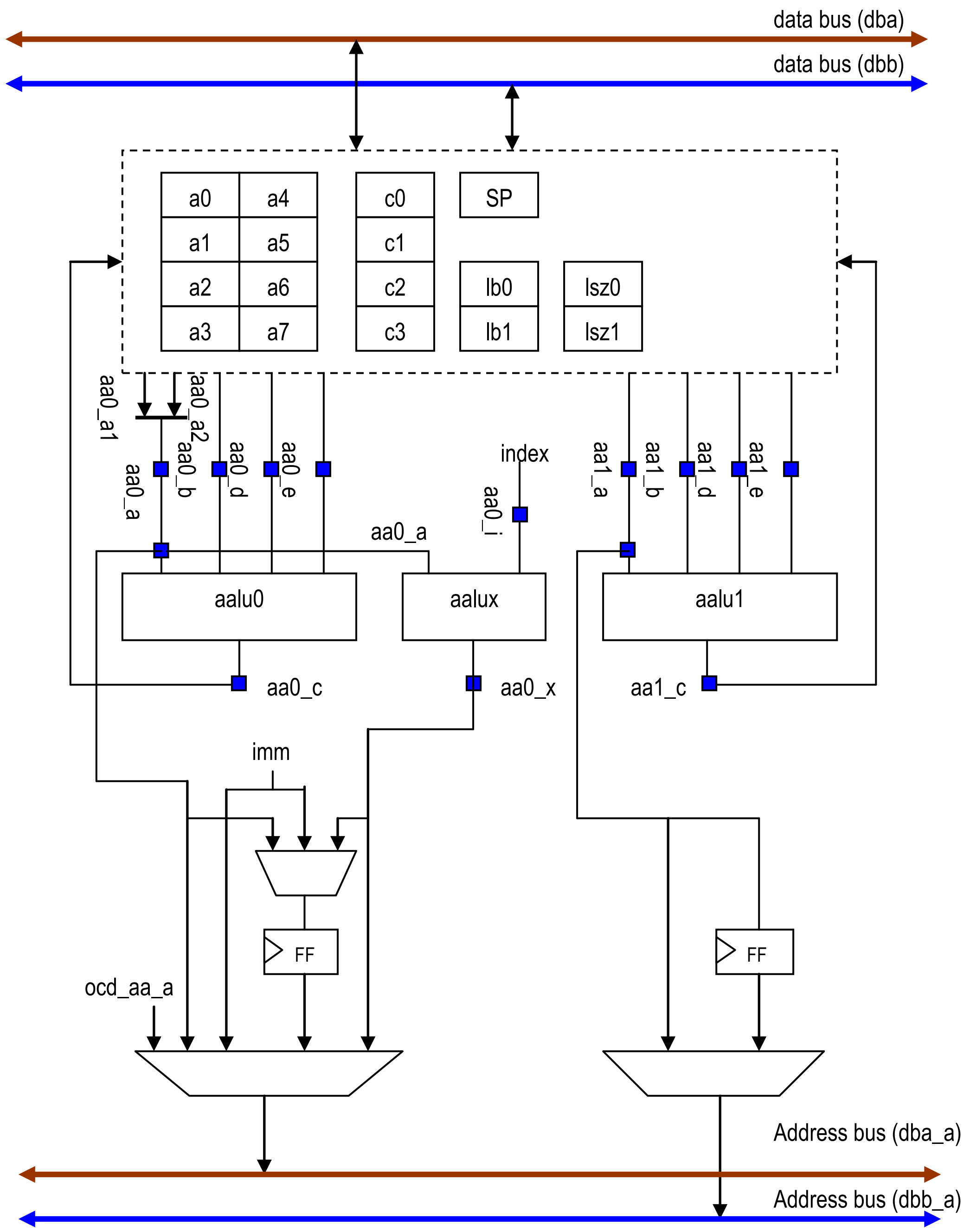
DMIO:

Called as data memory-IO; allocated address space is (0xC00000 ~ 0xFFFFFF) 4MB

Used for byte (8-bit), short (16-bit) and int (32-bit) data storage

While performing single memory access complete address space is treated as contiguous and single data element is accessed. In case of dual memory access, data memories DMA and DMB are treated as separate memories and each can be accessed simultaneously to fetch two data elements separately. It is not possible to access two data from the same memory i.e. either DMA or DMB.

7.1 Block diagram address generation unit



Note: aalux is a functional unit to calculate address for immediate indexed addressing mode.

Following registers are used by address generation units (aalu0, aalu1) to independently calculate the address for the data memory.

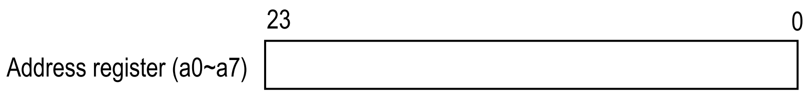
- Address registers (a0, a1, a2, a3, a4, a5, a6 and a7)
- Offset registers (c0, c1, c2 and c3)
- Stack pointer (sp)
- Loop address registers (lb0, lb1, lsz0 and lsz1) used for special (cyclic and bit reverse) addressing modes

7.2 Address registers (a0 ~ a7)

There are 8 address registers (a0 ~ a7) of 24-bit each, which hold the address for the data memory.

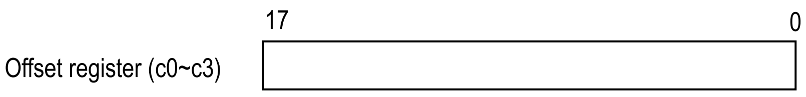
For a single data access (load/store) aalu0 uses all the address registers (a0 ~a7) for the address calculation which addresses to complete data memory space.

However, for the dual data access (load/store) aalu0 uses address registers (a0~a3) which addresses DMA address space and aalu1 uses address registers (a4 ~ a7) for the address calculation to address the DMB address space.



7.3 Offset registers (c0 ~c3)

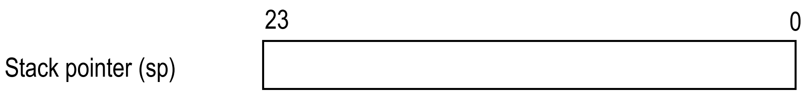
There are 4 offset registers (c0 ~ c3) of 18-bit each, which hold the offset value to update the address register. All the offset registers can be used by aalu0 and aalu1.



7.4 Stack pointer (SP)

Stack pointer is of 24-bits and it is used to address data memory in SP indexed addressing mode. Stack pointer holds the base address for the stack frame which is allocated in the data memory-A (DMA).

After reset stack pointer is initialized to zero. Before using it the stack pointer value must be set.



7.5 Loop address registers

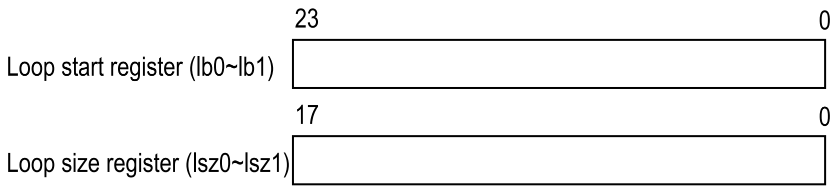
There are four loop address registers (lb0, lb1) of 24-bits and (lsz0, lsz1) of 18-bits each. These are used for calculating the address in address generation unit when it is used in special addressing modes like cyclic addressing and bit reverse addressing.

Registers (lb0, lb1) are called as loop start registers and used to store starting address of the loop buffer which is addressed in cyclic or bit-reverse fashion.

Registers (lsz0, lsz1) are called as loop size registers and used to store the loop size. Loop end address is calculated based upon this loop size register.

In special addressing mode there are two modes of operation mode-0 and mode-1. In mode-0 register pair (lb0 and lsz0) is used and in mode-1 register pair (lb1 and lsz1) is used.

Details of the special addressing modes are described in the later section [7.7.6].



7.6 Pipeline sequence for the address generation unit

For addressing the data memory, same address bus is used for both load and store operation.

- Address calculation for load and store operation is done in pipeline stage “D”.
- For the load operation address is placed on the address bus in pipeline stage “D”.
- However, for the store operation address is latched and delayed by 1 cycle and then it is placed on the address bus in pipeline stage “E1”.

7.7 Addressing modes

Data memory (DM-A, DM-B and DM-IO) is byte(8-bit), short(16-bit) and int(32-bit) accessible. Accessing long long (64-bit) data type is also supported, which is limited to only data memory DM-A. Data memory DM-A is aliased to LDMA which is called as long data memory-A to store long long variable data type.

Data is aligned byte wise in the data memory.

sr	Access type	Bits	Offset to address (step value)
1	byte	8-bit	1
2	short	16-bit	2
3	integer	32-bit	4
4	long long	64-bit	8

Address generation unit supports following addressing modes to address data memory.

- Indirect addressing with post modification
- Immediate indexed addressing
- Stack pointer (SP) indexed addressing
- Direct addressing
- Cyclic addressing
- Bit reverse addressing

7.7.1 Indirect addressing with post modification

In this mode, address registers (a0 ~ a7) hold the base address and offset registers hold the offset value to access the data memory. For single access, address registers (a0~a7) are used. However, for dual access, address registers (a0~a3) hold the address for data memory-A; and address registers (a4~a7) hold the address for data memory-B.

The address registers (a0 ~ a7) contain 24-bit unsigned values, used as pointers for the data memory. And the offset registers (c0 ~ c3) are used as pointer modifiers which hold the offset value. Offset value can be added to or subtracted from the base address.

In this mode, along with indirect addressing, post address modification is also performed simultaneously in the address generation unit (aalu0, aalu1).

Post address modification modifies the pointer that is being used. So, at the end of the processor cycle a new value for the pointer is stored back in the same address register which was used for indirect addressing.

Example:

```
ra0 = [a0 + c0]       // indirect load with post address modification
[a4 + c0] = axs1      // indirect store with post address modification
```

7.7.2 Immediate indexed addressing mode

In this mode, address registers (a0 ~a7) hold the base address and the offset is an immediate value to access the data memory. This addressing mode is used only when single data is accessed. The address registers (a0 ~ a7) contain 24-bit unsigned values, which are used as pointers for the data memory and the immediate value is used as an offset to the base address contained in the address register.

In this addressing mode, pointer value in an address register does not get modified.

Example:

nop; ra0 = a0[4]   // immediate indexed load

nop; a0[4] = axs1   // immediate indexed store

In the above example, assume value a0 = 0x000000 and offset is 4.

address = base_addr + offset

So the effective address (0 + 4 = 4) is calculated in aalux and address is put on the address bus (dba_a).

7.7.3 Stack pointer (SP) indexed addressing mode

This addressing mode is used to address the stack frame, which usually resides in the data memory-A (DMA). This mode of addressing is used when single data is accessed from stack area, also called as spill area. Stack pointer holds the base address (24-bit unsigned value) to access data from the stack frame which resides in data memory (DMA). And the immediate value is an offset to the base address.

In this addressing mode, value of the stack pointer does not get automatically updated. An instruction to update the stack pointer (SP) should be used to modify SP. After reset stack pointer is initialized to zero. First it should be assigned a value before using it.

Example:

nop; sp[4] = ra0 //push ra0 to stack frame addressed by SP with offset of 4

nop; rb0 = sp[4] //pop to rb0 from stack frame addressed by SP with offset of 4

In the above example, assume sp is initialized to 65520 (sp = 65520) and offset is 4.

address = base_addr + offset

So effective address (65520 + 4 = 65524) is calculated in aalu0 and address is put on address bus (dba_a).

Note:

- Assignment to SP should be aligned at multiples of 8 (eg. sp = 65528)
- Stack frame allocation should be within DMA address range

7.7.4 Definition of stack frame

Stack frame is initialized in the (lpdsp32.bcf) file and stack grows downward from the base address defined by the stack pointer.

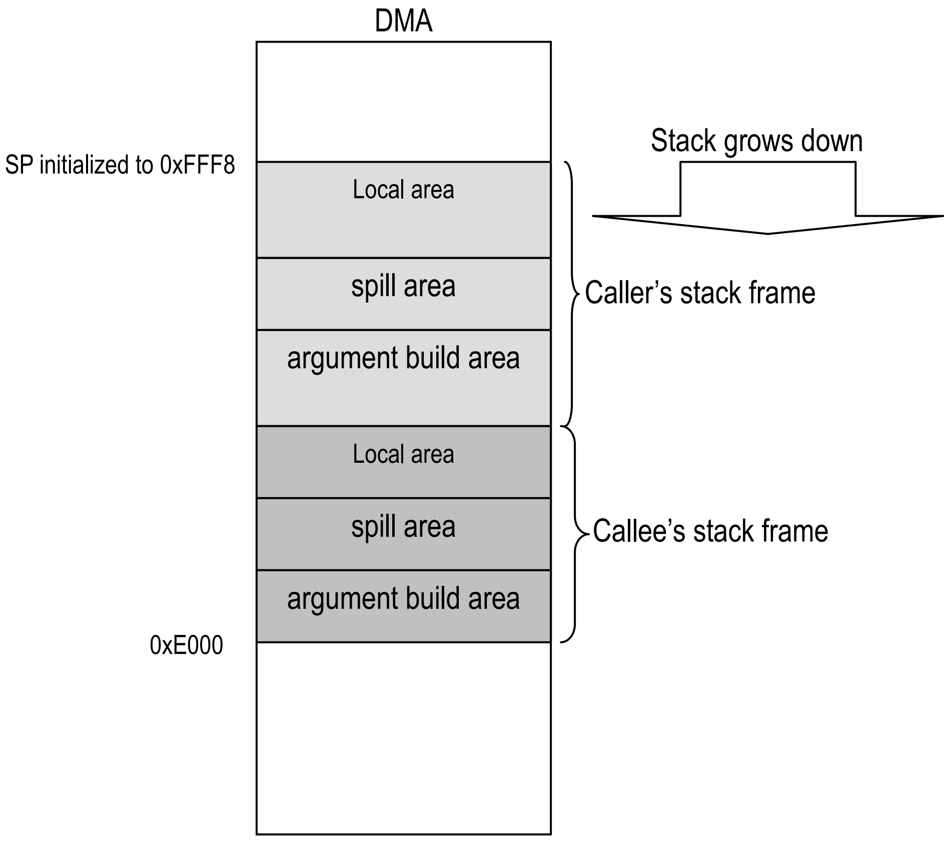
Definition of stack area:

```
_stack DMA 0xe000 8184
```

The above definition reserves the stack area from address 0xe000 plus 8184 (8K) locations.

In this case SP will be assigned with value sp = 65528 (0xFFF8).

Stack frame is used to store local variables, building the arguments and to backup registers during context switching, also called as spilling.



7.7.5 Direct addressing

In the direct addressing mode, immediate address for the data memory is directly provided in the instruction itself. Immediate value is 24-bit unsigned; with that complete data memory address space can be addressed.

Direct addressing mode is only available in the long instructions.

Example:

```
ra0 = [0x8FFFFF]  //load data from DM to ra0 (address is 24-bit unsigned immediate value)
rb0 = [0xC0000F]  //load data from DM to rb0
[0xC0000E] = rb0  //store rb0 to DM
[0x7FFFFE] = rb1  //store rb1 to DM
```

7.7.6 Cyclic addressing

Cyclic addressing and bit reverse addressing modes are the special kinds of addressing modes available in LPDSP32-V3.

When cyclic addressing mode is used, cyclic buffer addressing gets simplified for doing the filter operation (FIR/IIR). The biggest advantage of cyclic addressing is to avoid software overhead for checking the pointer values after every update.

There is a special hardware for address generation, which generates address for the buffer in cyclic order. This hardware wraps around the pointer, once it goes beyond the buffer boundary.

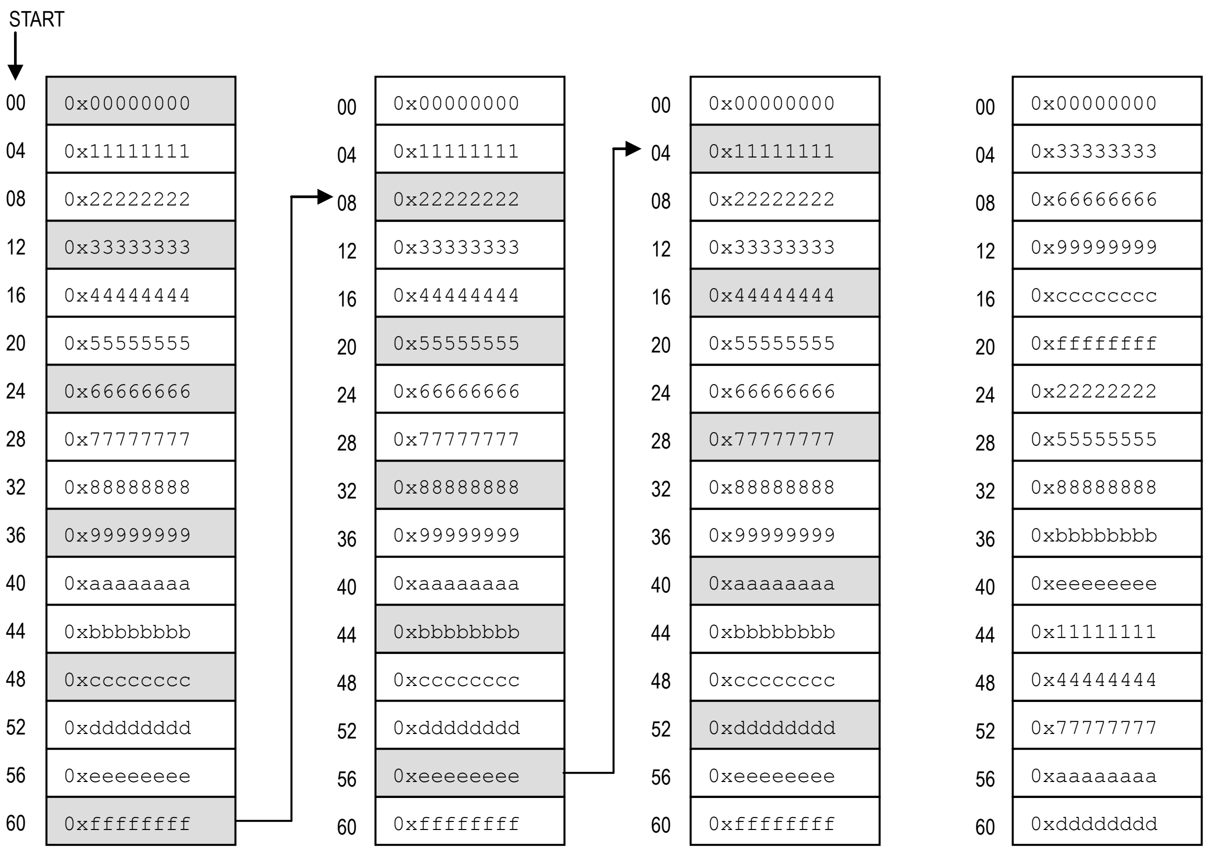
A cyclic buffer is a set of memory locations that stores data in the data memory. An index pointer steps through the buffer, being post-modified and updated by the addition of a specified value (positive or negative) for each step. If the modified address pointer falls outside the buffer address range, the length of the buffer is subtracted from or added to the value, as required to wrap the index pointer back to the start of the buffer. There are no restrictions on the value of the base address for a cyclic buffer.

To understand cyclic addressing, see the diagram on the next page.

For example:

Buffer length (size) = 16; start_address = 0; modifier offset (step) = 3

Input cyclic buffer: In_buff; Output buffer: out_buff



	In_buff (pass-1)	In_buff (pass-2)	In_buff (pass-3)	out _buff
00	0x00000000	0x00000000	0x00000000	0x00000000
04	0x11111111	0x11111111	0x11111111	0x33333333
08	0x22222222	0x22222222	0x22222222	0x66666666
12	0x33333333	0x33333333	0x33333333	0x99999999
16	0x44444444	0x44444444	0x44444444	0xcccccccc
20	0x55555555	0x55555555	0x55555555	0xffffffff
24	0x66666666	0x66666666	0x66666666	0x22222222
28	0x77777777	0x77777777	0x77777777	0x55555555
32	0x88888888	0x88888888	0x88888888	0x88888888
36	0x99999999	0x99999999	0x99999999	0xbbbbbbbb
40	0xaaaaaaaa	0xaaaaaaaa	0xaaaaaaaa	0xeeeeeeee
44	0xbbbbbbbb	0xbbbbbbbb	0xbbbbbbbb	0x11111111
48	0xcccccccc	0xcccccccc	0xcccccccc	0x44444444
52	0xdddddddd	0xdddddddd	0xdddddddd	0x77777777
56	0xeeeeeeee	0xeeeeeeee	0xeeeeeeee	0xaaaaaaaa
60	0xffffffff	0xffffffff	0xffffffff	0xdddddddd

In_buff (pass-1) In_buff (pass-2) In_buff (pass-3) out _buff

Cyclic addressing mode operation:

This addressing mode is selected when value inside the loop size register (lsz0/lsz1) is not equal to zero. In this mode address is calculated relatively (base + offset). Loop start address is stored in register (lb0 or lb1) and loop size is stored in (lsz0 or lsz1) either of these registers are selected according to mode-0 and mode-1 operation.

Where,

(+%0): special address mode-0, means start address is in lb0 and loop size is in lsz0

(+%1): special address mode-1, means start address is in lb1 and loop size is in lsz1

To utilize this address mode in "C" code an intrinsic function called as "cyclic_add" is provided. This addressing mode can be used for both single and dual load /store operation.

Address generation for cyclic addressing mode

This is the functional macro for cyclic address generation in aalu.

```
cyclic_add (ptr, step, buff_strt_addr, buff_size)
{
start = buff_strt_addr; //cyclic buffer start address
result = ptr + step; //pointer calculation
end = buff_strt_addr + buff_size; //end address calculation
if (step < 0 && result < start) //step is negative
result = result + size; //increment
else if (step >= 0 && result >= end) //step is positive
result = result - size; //decrement
}
```

Example:

```
24      lb0 = -8388608   //cyclic buffer start pointer
26      a1 = 64
28      a2 = 0
30      lsz0 = 64        //cyclic buffer size (lsz0 != 0)
32      lp 16 5          //hw_loop lc = 16, loop size is 5
34      a0 = lb0         //delay slot-1
35      c0 = 4; c1 = 12  //delay slot-2
37      a3 = a0 + 0      //hw_loop starts here
38      ra0 = [a0+%0c1]  //cyclic addressing mode
39      [a2+c0] = ra0
40      [a1+c0] = a3     //hw_loop ends here
41      ret
```

Contents of the HW loop registers at the first iteration are:

```
LC = 16       LSTK = 37     LPA = 40
```

7.7.7 Bit reverse addressing

The bit reverse addressing mode simplifies the address calculation for the FFT (Fast Fourier Transform) algorithm. There is a special hardware provided in the address generation unit to generate address for the data buffer in bit reverse fashion. Therefore using this addressing mode reduces software overhead of rearranging or reordering the data in the data buffer to do FFT operation.

In the bit reverse addressing mode, address generation unit selects 16-bits from 24-bit address and does the bit reverse operation.

Bit reverse addressing mode operation:

This addressing mode is selected when value inside the loop size register (lsz0/lsz1) is equal to zero. To understand the concept of bit reverse addressing mode, please see the below table. For the sake of simplicity please assume pointer value is 4-bit and step value is 4.

Access	Step	Rev(access)	Rev(step)	result	Bit rev addr = rev(result)
Start(1000)	0100	0001	0010	0011	1100
1100	0100	0011	0010	0101	1010
1010	0100	0101	0010	0111	1110
1110	0100	0111	0010	1001	1001
1001	0100	1001	0010	1011	1101
1101	0100	1011	0010	1101	1011
1011	0100	1101	0010	1111	1111
1111	0100	1111	0010	10001	1000

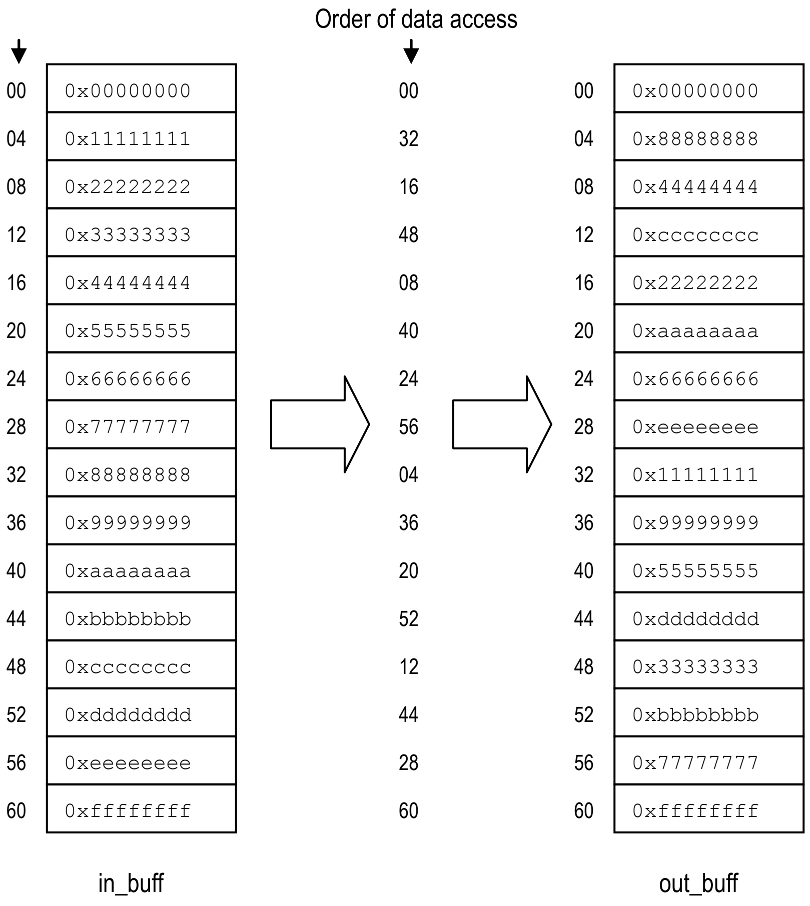
This is the functional macro for bit reverse address generation in aalu.

```
reverse_add (ptr, step, buff_strt_addr) {
result = reverse (reverse (ptr) + reverse (step))
}
```

Example:

Buffer length (size) = 16; start_address = 0; modifier offset (step) = 8

Input buffer: in_buff; Output buffer: out_buff



	in_buff	Order of data access		out_buff
00	0x00000000	00	00	0x00000000
04	0x11111111	32	04	0x88888888
08	0x22222222	16	08	0x44444444
12	0x33333333	48	12	0xcccccccc
16	0x44444444	08	16	0x22222222
20	0x55555555	40	20	0xaaaaaaaa
24	0x66666666	24	24	0x66666666
28	0x77777777	56	28	0xeeeeeeee
32	0x88888888	04	32	0x11111111
36	0x99999999	36	36	0x99999999
40	0xaaaaaaaa	20	40	0x55555555
44	0xbbbbbbbb	52	44	0xdddddddd
48	0xcccccccc	12	48	0x33333333
52	0xdddddddd	44	52	0xbbbbbbbb
56	0xeeeeeeee	28	56	0x77777777
60	0xffffffff	60	60	0xffffffff

To utilize this addressing mode in “C” code an intrinsic function called as “reverse_add” is provided. This addressing mode can be used for both single and dual load /store operations.

Example:

```
24      a1 = 0             //buffer start address
26      a4 = -8388608
28      lsz0 = zero        //select bit reverse mode (lsz0 = 0)
29      lp 16 5            //hw_loop LC = 16, loop size = 5
31      a0 = 64            //delay slot-1
33      c0 = 4; c1 = 32   //delay slot-2
35      [a0+c0] = a1       //hw_loop starts here
36      nop
37      bxs0 = [a1+%0c1] //bit reverse addressing
38      [a4+c0] = bh0      //hw_loop ends here
39      ret
```

Contents of the HW loop registers at the first iteration are:

```
LC = 16       LSTK = 35      LPA = 38
```

Limitations:

- Array should be aligned at power of 2
- Maximum addressable array size allowed is 16K (Maximum 16384 point FFT)

8. Program control unit (PCU)

The program control unit (PCU) controls the flow or sequence of the instructions in the program. Controlling involves fetching the instruction from the program memory and issuing it to the instruction decoder in a pipelined manner. While a new instruction is being fetched, the previously fetched instruction is decoded. It also involves handling of control instructions like jump, subroutine call, hardware loop and interrupts by keeping track of status flags in the status register (SR).

8.1 Registers used for PCU

Program control unit consists of various registers which are used for program sequencing.

- Program counter (PC)
- Status register (SR)
- Link register (LR)
- Registers used for controlling hardware loop

  Loop counter pointer (LCP)

  Loop count register (LC)

  Loop start register (LSTK)

  Loop end register (LPA)
- Registers used for interrupt controller

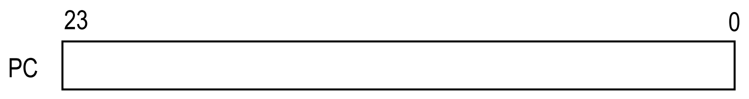
  Interrupt link register (ILR)

  Interrupt mask register (IMSK)

  Interrupt status register(IRQ_STAT)

8.1.1 Program counter (PC)

The program counter (PC) is a 24-bit register which is used for addressing the program memory (PM) from where the instructions need to be fetched.



After the core reset, program counter is initialized to zero and starts fetching the instructions from PM address zero. Program counter (PC) is not accessible to the user through an instruction, because there is no such instruction available to read from or write to the PC. Program counter can only be modified by using jump instruction. Of course the user can monitor this register in the instruction set simulator and the on chip debugger.

8.1.2 Status register (SR)

The status register (SR) is of 11-bit. This register is used for controlling the mode of core operations and storing the status flags (v, n, z).

10	9	8	7	6	5	4	3	2	1	0
s0	r0	im0	ie0	s	r	im	ie	v	n	z
srmode0				srmode				srflags		

Bitwise descriptions of status register:

Sr.	SR bit	Bit no.	Description
1	z	0	Zero flag
2	n	1	Negative flag
3	v	2	Overflow flag
4	ie	3	Interrupt enable, 1: enable 0: disable
5	im	4	Integer/fractional multiplier mode selection 0: fractional (accumulator << 1) 1: integer
6	r	5	rounding, 1: enable 0: disable
7	s	6	saturation, 1: enable 0: disable
8	ie0	7	Backup (ie)
9	im0	8	Backup (im)
10	r0	9	Backup (r)
11	s0	10	Backup (s)

After core reset, the status register is initialized to zero. Control bits (srmode bits) are enabled in lpdsp32_init.s which contains the initialization code before starting the main program.
Except srflags, each bit of the status register is independently accessible like

ie = 1 //enabling interrupt
r = 1 //enabling rounding
s = 1 //enabling saturation

By default srmode bit im = 0, that means multiplication is done in fractional mode.
Other control bits srmode0 are used for keeping the backup of srmode bits during interrupts.

8.1.3 SR flags and condition codes

In the status register there are 3 status flags (srflags) called zero flag (z), negative flag (n) and overflow flag (v). Out of these, the overflow flag (v) is not being used and it always remains zero. This is because 8 overflow bits already exist in the accumulator to take care of overflow. Earlier it was used for storing the flag for accum_t (72-bit) data type operations.

Zero flag (z) and negative flag (n) are generated by the alu0 operations. The only operations, which generates the srflags are signed compare (cmp), unsigned compare (cmpu), division (div) and bit test (bt).

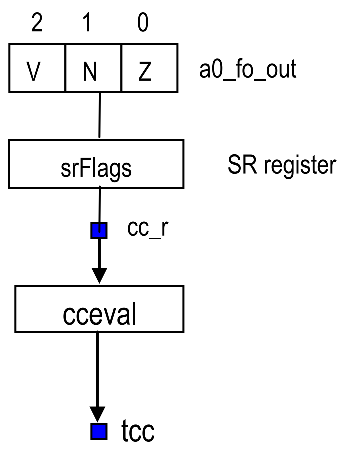
These srflags (v, n, z) are accessible for reading and writing through move operations.

Example:

```
flags = 0x7        //enabling all three flags (v, n, z)
ra0 = flags        //reading srflags
sr = 0x7FF         //writing '1' to all 11-bits of status register
rb0 = sr           //reading all 11-bits of status register
```

These srflags are translated to condition codes which are later used by the program controller unit to control the execution of conditional jump instructions.

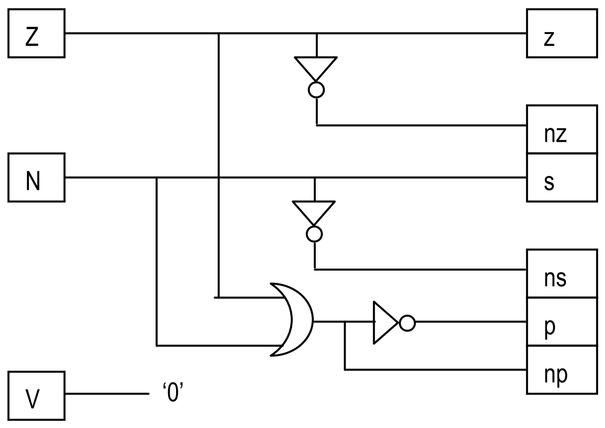
Sr.	(cc) Notation	Explanation
1	z	1: When operand-1 equal to operand-2 else 0
2	nz	1: When operand-1 not equal to operand-2 else 0 (z = 0)
3	s	Negative value (1: when N flag = 1)
4	ns	Not negative value (1: when N flag = 0)
5	p	Positive value (1: when N & Z flags = 0)
6	np	Not positive value (1: when N flag = 1 or Z flag = 1)



tcc: The signal used by controller to decide to take a jump or not

8.1.4 Condition code evaluation

All the condition codes are translated and computed in the functional unit called (cceval). Generation of condition code is shown in the diagram below,



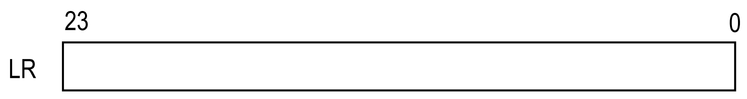
Below are the examples of using condition codes,

Example:

if (np) jps 11	*//short conditional jump*
if (nz) jpsdb 103	*//short conditional jump with delay slot*
if (z) jp 15348	*//direct jump*
if (nz) jpdb 15464	*//direct jump with delay slot*

8.1.5 Link register (LR)

There is one link register (LR) which is of 24-bit. It is used to store the return address (incremented PC) when executing subroutine call instructions.



Example:

26 nop; ra0 = 0xf

28 call 0xff

30 nop; ra0 = ra1

In the above example, call instruction is at location 28. Therefore link register stores the value 30, which is return address after the subroutine call.

After core reset, link register (LR) is initialized to zero.

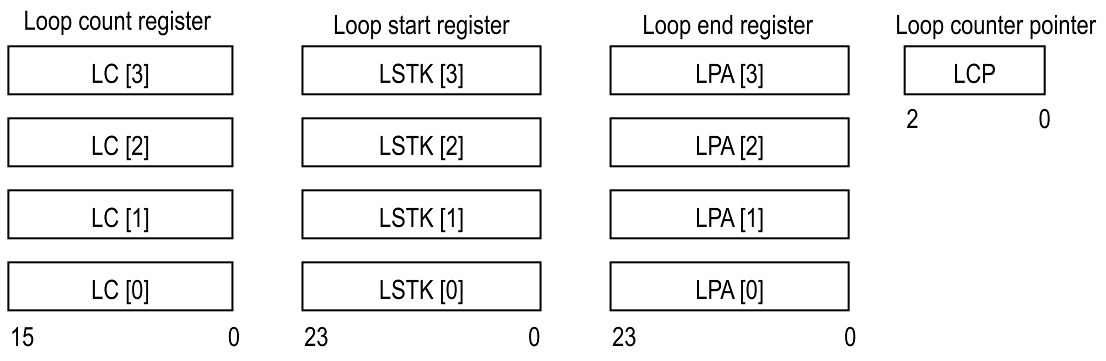
Since there is only one link register, nested subroutine call is not supported.

8.1.6 Hardware loop stack registers

Hardware loop is implemented and controlled by sets of hardware loop registers also called as register stack. This register stack has a depth of 4; meaning that hardware loop nesting up to 4 levels is supported by the hardware.

When a hardware loop is initiated, the loop start address, loop count and loop end address are pushed on to the registers LSTK (24-bit), LC (16-bit) and LPA (24-bit) respectively. Register stack is pointed by the loop counter pointer LCP (3-bit), which is used to count the loop levels.

These are the hardware loop registers:



After core reset the loop counter pointer (LCP) is initialized to (LCP = 4), that means no hardware loop are initiated. And other registers LC, LSTK and LPA are initialized with the value zero.

LCP value	Nesting level
100 (4)	Hardware loop is not initiated (Default)
011 (3)	Nesting level-4 (Top level)
010 (2)	Nesting level-3
001 (1)	Nesting level-2
000 (0)	Nesting level-1 (Inner most level)

Details of the hardware loop will be explained in a later section.

8.2 Other registers in the controller

Other than the list of earlier mentioned registers, there are some more registers used in the controller.

- Instruction fetch buffer; reg_if0 (20-bit), reg_if1 (20-bit) and reg_ir1 (20-bit) for saving second half instruction word from the complete 40-bit fetched instruction.
- Instruction register (IR), controller fetches the 40-bit instruction from the program memory (PM) and passes it to the instruction decoder via 40-bit output signal (trn_IR_D_out).
  This signal (trn_IR_D_out) is formed by combining two 20-bit registers.
  Again, inside the instruction decoder this instruction register is registered and prepared into two copies (reg_IR_D and reg_IR_E1). Register (reg_IR_D) is used to control the "D"/decode stage operations and (reg_IR_E1) is used to control "E1"/execution stage operations.

8.3 Classification of short and long instructions in the decoder

Two instruction widths are supported by LPDSP32-V3, short (20-bit) and long (40-bit).
To maintain a constant issue rate of one 40-bit instruction word per clock cycle, one 40-bit instruction word must be fetched in each cycle. In order to obtain a simple implementation of the program memory (PM), instructions are always fetched as 40-bits and at even addresses. Therefore, it is not possible to fetch individual 20-bit short instructions separately.

Pre-decoding of the instruction is first done in the controller to classify the instruction types.

If bit IR[19] and IR[18] are both zero then the instruction type is (A_short).
Else if IR[19] is one then the instruction type is (M_short).
Otherwise the instruction is a long instruction.

8.4 Program control unit (PCU) action sequence

Program memory (PM) is a synchronous memory; where address is latched or registered first. At the end of one cycle a new address is registered and in the next cycle the instruction at that address appears on the output port of PM.

For single cycle instructions, this is the action sequence of the program controller.

- An instruction is fetched from PM. This instruction is stored in IR, reg_if0, reg_if1and reg_ir1.
- The instructions that are present in the IR registers are decoded.
  For each version of IR (reg_IR_D, reg_IR_E1) the control signals enable the actions for the corresponding pipeline stage.
- The program counter (PC) value is incremented in steps of 0, 1 or 2.
  The new value is stored back in the PC register
- In order to break the PM critical path(timing), from PM read data to PM read address, the program counter value to be sent to the PM is incremented in steps of 1, 2 or 3, independent of the current PM read data, in the Next PM addr logic as shown in figure. Increment steps 1, 2 or 3 will be depending on alignment or jump. A mismatch in the actual PC increment and the increment value for the PM address will be handled by the controller.

Deviations from this basic PC sequence occur when a multi cycle instruction, delay slot instruction, or a control flow instruction is executed or when interrupts occur.

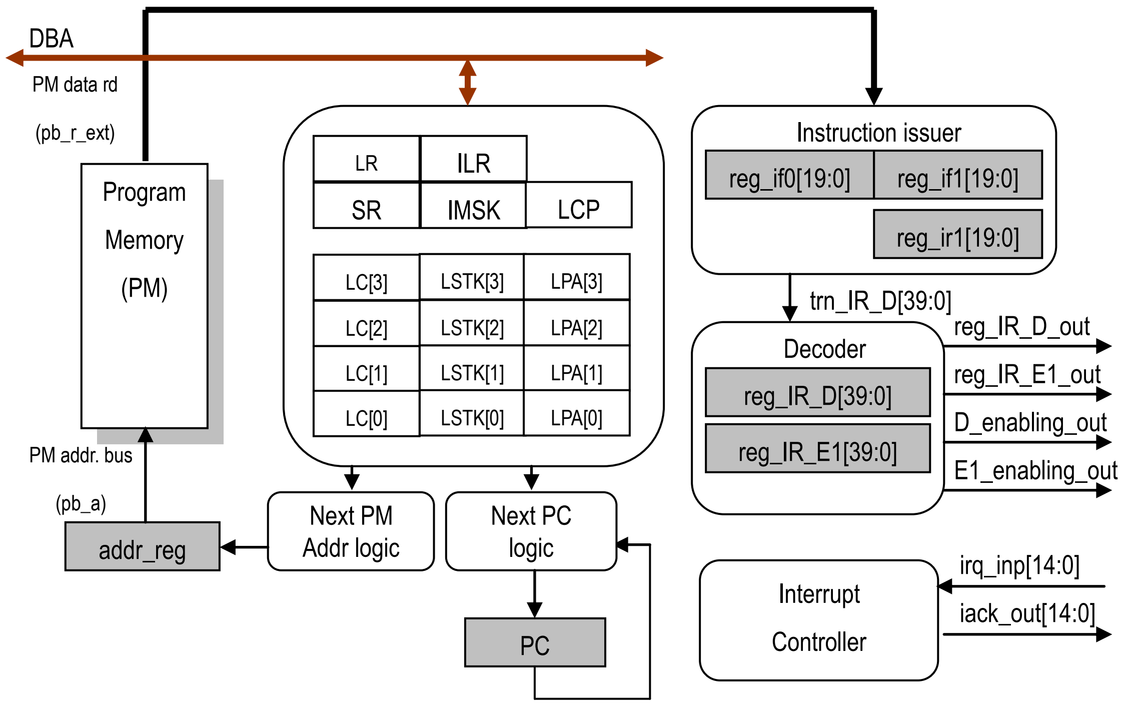


Fig: Program control unit

8.5 Multi cycle instruction

Due to pipelined action, a multi cycle instruction takes more than one cycle for instruction execution. Typically control instructions are multi cycle instructions, because these instructions change the normal flow of program sequence and the pipeline needs to be flushed and refilled due to change in the program counter. For example, a conditional jump instruction takes two cycles for execution when the condition is true, else it takes one cycle.
When a multi cycle instruction is issued, the next instruction is issued n cycles later. The PCU controls the issuing of multi cycle instructions by means of the Boolean signal (issue_sig_in). When executing multi cycle instructions, issue_sig_in is always logic 0. When a multi cycle instruction is issued, a signal (multicycle_dlyslot) in the controller becomes logic 1.
When the specified number of cycles are over, multicycle_dlyslot becomes 0 again.
The multicycle_dlyslot signal will be logic 1 for (n – 1) cycles. An instruction that was fetched may only be issued when issue_sig_in is logic 1. When the issue_sig_in is logic 0, the instruction that was fetched in the previous cycle is not issued, but is kept in the fetch buffer register reg_if0, reg_if1.

8.6 Delay slot instruction

Since the multi cycle instructions (jump) can take a number of cycles before the jump is taken, in these cycles other instructions can be executed. This is an alternate approach of executing instructions instead of stalling the processor during the multi cycle instructions. Instruction execution is continued in a normal way while the jump is taken. The (jpdb) instruction is an example of an instruction with delay slots.

During the delay slots and multi-cycle instructions, interrupt is disabled.
Below signals are used to disable the interrupt during delay slot operation,
Signal name: diid: disables the interrupt in D stage
Signal name: diie1: disables the interrupt in E1 stage

8.6.1 Instructions inside the delay slot

Not all the instructions can be used as delay slot instructions. The instructions which take single cycle are allowed in the delay slot. Multi cycle instructions are not allowed to be kept in the delay slot. When there is no instruction to put in delay slot, a no operation (NOP) instruction is introduced.

8.7 Multi cycle and delay slot instructions

This table gives an overview of different multi cycle instructions present in the LPDSP32-V3 in a single look. The table also indicates number of cycles required for executing the instruction and number of delay slots. When there is a delay slot, number of delay slots is number of cycles minus 1. Other than these instructions all other instructions take a single cycle for executing.

Sr.	Instruction	Description	No. of cycles	No. of delay slots
1	jps	Relative jump	2	NA
2	jpsdb	Relative jump with delay slot	2	1
3	if (cc) jps	Conditional short jump: Condition is true Condition is false	 2 1	 NA NA
4	If (cc) jpsdb	Cond. short jump with delay slot Condition is true Condition is false	 2 1	 1 1
5	jp	Direct jump	2	NA
6	jpdb	Direct jump with delay slot	2	1
7	if (cc) jp	Direct cond. jump: Condition is true Condition is false	 2 1	 NA NA
8	If (cc) jpdb	Direct cond. jump with delay slot Condition is true Condition is false	 2 1	 1 1
9	jp [a]	Indirect jump	3	NA
10	jpdb [a]	Indirect jump with delay slot	3	2
11	call	Direct subroutine call	2	NA
12	calldb	Direct subroutine call with delay slot	2	1
13	call [a]	Indirect subroutine call	3	NA
14	calldb [a]	Indirect subroutine call with delay slot	3	2
15	ret	Subroutine return	3	NA
16	retdb	Subroutine return with delay slot	3	2
17	reti	Interrupt return	3	NA
18	retidb	Interrupt return with delay slot	3	2
19	sint	Software interrupt/interrupt	2	NA
20	powerdown	Core Halt	2	NA
21	lp	Hardware loop	3	2
22	swbreak	Software break	2	NA

8.8 Handling unaligned jump targets

In the example below, assembly instructions (A, B, CD, E, FG and H) are stored on the PM at start address 10. Instruction A, B, E and H are 20bit instructions and instruction CD & FG are 40bit instructions; which are stored on program memory (PM) as shown below. PM consists of two parts PM0 and PM1 which cannot be accessed independently, partitioning is done at output of the PM.

PCU will first fetch from PM at address 10. This will result in a long instruction word that contains two short instructions A and B. Instruction A can then be issued immediately, while instruction B can be saved in a buffer register. In order to issue instruction B, a new instruction fetch is not needed. Next the long instruction at PM[12] is fetched and issued. Then the instruction word at PM[14] is fetched. As E is a short instruction, F is again saved in the buffer register. Note that F is the first part of the long instruction FG. Before issuing instruction FG, first instruction word at PM[16] is fetched. It is also possible that there is a jump to instruction FG. In that case PM[14] is fetched to obtain the first part F. As the instruction is still incomplete, it cannot be issued; first PM[16] has to be fetched to obtain the second part G. Therefore, in the case of a jump to a long instruction at an unaligned target address, one cycle is lost in which no instruction is issued and stall signal is issued in order to stall the pipeline.

Addr.	PM0	PM1
10	A	B
12	C	D
14	E	F
16	G	H

Unaligned long instruction word

Normally, compiler tries to align the instructions such that there is no or rare unaligned jump target. Therefore, this is a very rare case.

In case of unaligned jump target, below is the instruction issue sequence in the controller.

IR[39:0] => {ii0[19:0], ii1[19:0]} => {A, M_nop} or {A_nop, M} => {is0[19:0], is1[19:0]}

When PC is aligned and no jump,

{is0[19:0], is1[19:0]} <= {if0[19:0], if1[19:0]}

In case of unaligned jump target, if0 is discarded and replace with if1 and stall signal is issued

{is0[19:0], is1[19:0]} <= {if1[19:0], if1{19:0}}

And in case of aligned jump target, if0 is discarded and replace with reg_ir1, which is used for storing second half word.

{is0[19:0], is1[19:0]} <= {reg_ir1[19:0], if0[19:0]}

8.9 Hardware loop

To avoid software overhead for looping over a set of instructions, hardware looping is supported. This is also called zero overhead looping. Hardware loop is completely controlled by hardware and can be nested up to 4 levels.

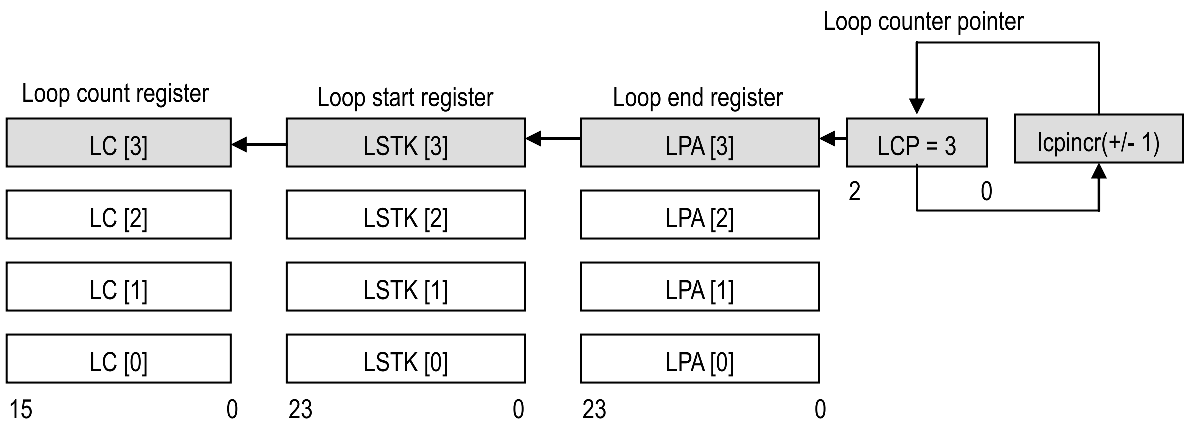
With the hardware loop instruction (lp), fast looping is possible over the set of instructions within a loop body; without any additional cycles being used for performing the looping task.

Hardware loop executes the number of iterations which are pre-specified and controlled by specific hardware. Status of the hardware loops are stored in a set of registers dedicated for hardware loops.

As described earlier, these are the registers used for controlling the hardware loop.

Register	Description	No. of registers	Bits	Purpose
LCP	Loop counter pointer register	1	3	Keeps the track of number of hardware loops that are active, and to address the LSTK, LPA and LC registers. LCP has an initial value of 4, which indicates that no hardware loop is active.
LC	Loop count register	4	16	Stores the remaining no. of iterations of the loop. LC is a register file with 4 registers, one for each loop level.
LSTK	Loop start register	4	24	Stores the address of the first instruction in a loop. LSTK is a register file with 4 registers, one for each loop level.
LPA	Loop end register	4	24	Stores the address of the last instruction in a loop. LPA is a register file with 4 registers, one for each loop level.

Loop counter pointer (LCP) indicates the number of hardware loops active and points to the register file to be used as shown in the below figure.



There are two delay slots for a hardware loop instruction, which means that after a hardware loop (lp) instruction another two instructions can be executed. If there are instructions available which can be kept in delay slots, they will be placed else nops will be inserted.

8.9.1 Hardware loop control

Hardware looping is also managed by the controller.

These are the operations performed in the controller to control hardware loops,

- Hardware loop end detection. When the program counter (PC) equals the loop end address, loop end is detected.
- When loop end is detected, PC jumps back to the loop start address (LSTK) and decrements the loop count (LC), until the loop iterations are finished and LC becomes 1. Once all the iterations of a loop are over, LCP is incremented by 1.
- It checks whether a hardware loop is active or not by checking the LCP value. If value in LCP is less than 4, then a do loop is active, so it sends the current value of LCP(pointer) to the LSTK, LPA and LC registers to get the corresponding values.

There are few control hazards, related to the hardware loop. These hazards will be discussed in the hazard section [12].

Example:

```
24 lp 2 17                  //top level loop runs 2 times, loop body size is 17
26 ra0 = [0]                //delay slot-1
28 nop; nop                 //delay slot-2
30     lp 4 11              //level-3 loop runs 4 times,
32     nop                  //delay slot-1
33     nop                  //delay slot-2
34         lp 6 6           //level-2 loop runs 6 times
36         nop              //delay slot-1
37         nop              //delay slot-2
38             lp 8 1       //level-1 innermost loop runs 8 times, loop body size is 1
40             nop          //delay slot-1
41             nop          //delay slot-2
42                 ra0 = ra0 + 1 //instruction inside the loop
43             nop
44         nop
```

8.10 Interrupt controller

Interrupt controller is a part of program controller unit (controller). There are 15 interrupt inputs for the interrupt controller called as irq_inp_in[14:0]. Hardware interrupt can be triggered by using one of these irq_inp_in pins. Fixed interrupt priority is assigned to the each interrupt input.
To trigger software interrupt, there is an instruction available called “sint”.

8.10.1 Interrupt vector table

This is the interrupt vector table with the 15 interrupts.

Interrupt request no.	irq pin	Vector Address	comments
0	-	0	Reset
1	irq_inp_in[0]	2	Interrupt 1
2	irq_inp_in[1]	4	Interrupt 2
3	irq_inp_in[2]	6	Interrupt 3
4	irq_inp_in[3]	8	Interrupt 4
5	irq_inp_in[4]	10	Interrupt 5
6	irq_inp_in[5]	12	Interrupt 6
7	irq_inp_in[6]	14	Interrupt 7
8	irq_inp_in[7]	16	Interrupt 8
9	irq_inp_in[8	18	Interrupt 9
10	irq_inp_in[9]	20	Interrupt 10
11	irq_inp_in[10]	22	Interrupt 11
12	irq_inp_in[11]	24	Interrupt 12
13	irq_inp_in[12]	26	Interrupt 13
14	irq_inp_in[13]	28	Interrupt 14
15	irq_inp_in[14]	30	Interrupt 15

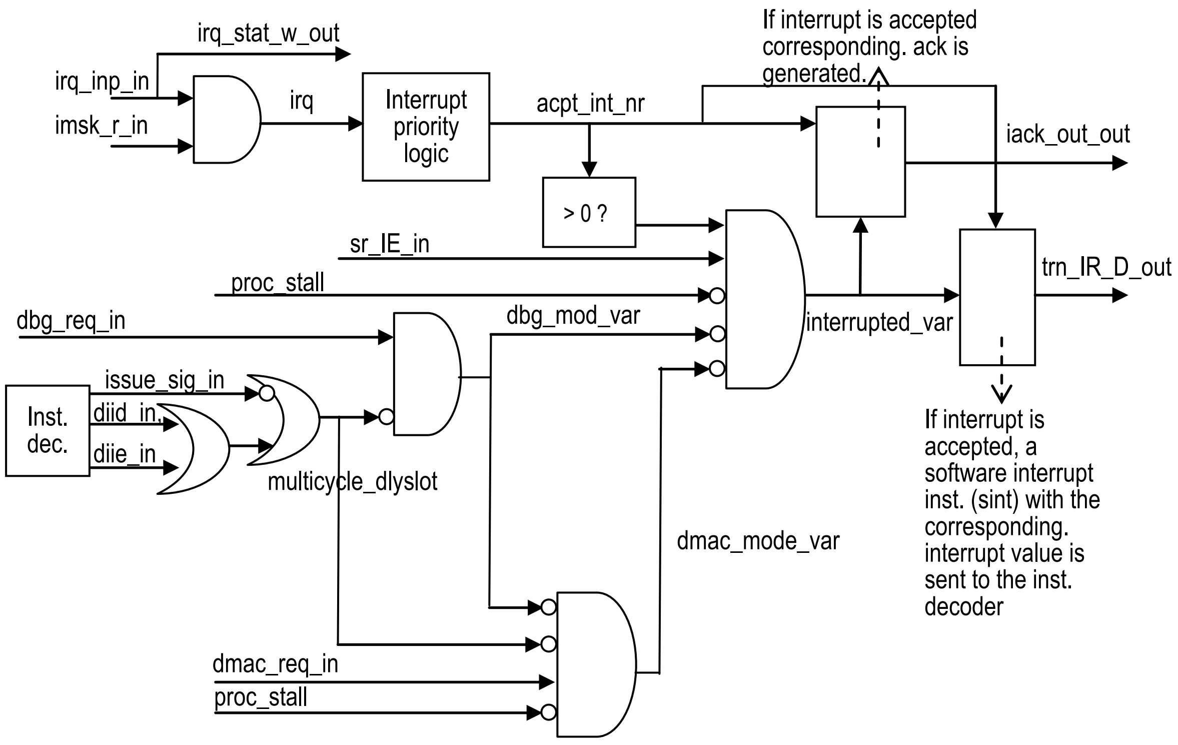
An interrupt forces an interrupt subroutine call to predefined address called an interrupt vector. Interrupt vector table always resides in the initial part of program memory (PM) starting from address 0. The instructions inside the vector table should be either absolute jump (jp) or interrupt return (reti) instructions. There should not be any conditional jump or delay slot instruction used in the vector table. Vector address 0 has same effect as reset, in which program counter (PC) is set to zero.

8.10.2 Interrupt circuit

Interrupt inputs are level sensitive, when there is a high level on one of the irq_inp_in pins, corresponding interrupt is accepted and acknowledged if certain conditions are satisfied.

It is recommended to synchronize the interrupts with respect to the system clock before being given to the core.

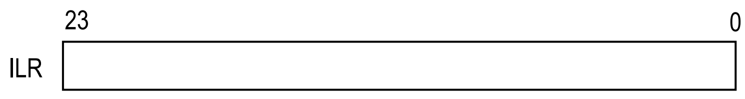
To understand the interrupt circuit, see the figure below.



These are the registers used by the interrupt controller.

8.10.3 Interrupt link register (ILR)

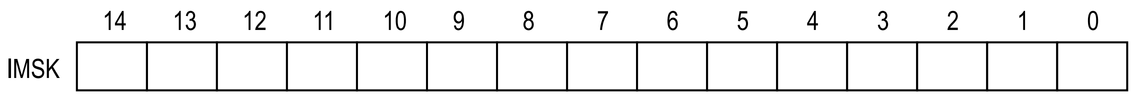
There is a single interrupt link register (ILR) which is of 24-bit. When an interrupt is occurred it is used to store the return address after the interrupt subroutine call.



After the core reset interrupt link register (ILR) is initialized to zero.

8.10.4 Interrupt mask register (IMSK)

There is a single interrupt mask register (IMSK) which is of 15-bit. By setting the respective bits in the interrupt mask register (IMSK) to ‘0’, disables the particular interrupt. After disabling an interrupt, if the interrupt occurs, it will be serviced only after enabling interrupt mask bit to ‘1’ again. When interrupt mask bit is ‘1’, if there is a corresponding pending interrupt, it will be served.



After the core reset interrupt mask register is initialized to (imsk = 0x7FFF).

Interrupt mask register is accessible to the user; values can be written or read from the register.

Example:

imsk = 0x7FFF //setting all bits high

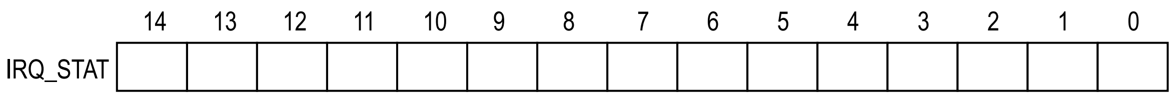
imsk= 0x7FFE //clearing LSB bit

ra0 = imsk //reading imsk register

At the “C” level this register can be accessed via inline functions. To write a value in the imsk register there is an inline function called set_interrupt_mask(int m) and the inline function to read the register is get_interrupt_mask().

8.10.5 Interrupt status register (IRQ_STAT)

Interrupt status register gives the status of the present interrupt inputs irq_inp_in of the core. This register can be read by the software using the inline function get_irq_stat().



8.10.6 Enabling and disabling the interrupts

Interrupts can be enabled by setting the interrupt enable bit (ie = 1) and can be disabled by clearing interrupt enable bit (ie = 0). Interrupt enable bit (ie) is a srmode bit which is part of the status register (SR). Interrupt is enabled in the initialization code specified in (lpdsp32_init.s) before entering into the main program.

At the "C" level this bit of the srmode can be accessed via inline functions. To enable the interrupts there is an inline function called enable_interrupts() and to disable the interrupts the inline function is disable_interrupts ().

Example:

```
ie = 1    //interrupt is enabled
nop
ie = 0    //interrupt is disabled
nop
```

8.10.7 Software interrupt

Software interrupt can be triggered by using the instruction (sint). Functional behavior of the software interrupt instruction is similar to hardware interrupt triggered by any of the irq_inp_in pins. Software interrupts cannot be masked by setting IMSK or IE registers, hence also known as non maskable interrupts.

Example:

```
sint 4     //4 is vector table address, range is (0, 2, 4, 6, 8, 10, 12, …., 28, 30)
```

For the detailed explanation of the interrupt controller and handling of interrupt service routine please refer the interrupt support manual [Ref.].

8.11 Halting the core

This is the very essential feature for saving the core power. To halt the core, the instruction (powerdown) is used. If powerdown instruction is executed, core will be halted and (powerdown_out) signal is asserted. Core operation can be resumed via raising the interrupts or by raising the resume input pin.

During powerdown, an external logic is required to enable or disable the clock to the core. The external logic should be monitoring resume and irq_inp_in pins continuously. irq_inp_in and resume signals are asynchronous and need to be synchronized outside of the core. As soon as the powerdown_out signal is asserted, the clock to the core can be completely disabled, provided the external logic can register the resume or interrupt pins and generate the clock to the core before giving the registered resume or interrupt signal to the core, to come out of powerdown. (Synchronization of the signals can be done by registering the signals two times with respect to the core clock.)

powerdown is a multi cycle operation and it takes 2 cycles to execute. There are no delay slots for this instruction.

Example:

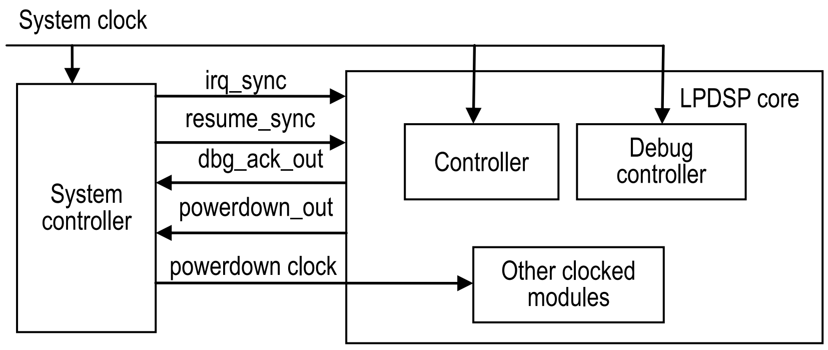
powerdown //no parameters required

For C programmers the inline assembly function core_halt() is available for executing powerdown.

Two cases should be considered if clock is stopped to the core and debugging needs to be done during powerdown:

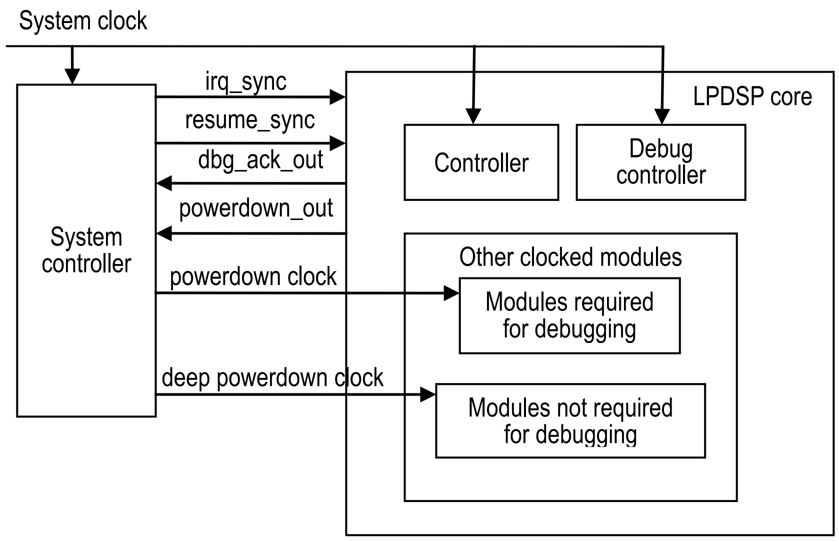
Case1:

To debug the core during the powerdown mode, the clock going to the debug controller and the controller should be always enabled. Any debugger operation is indicated by a logic high on the debug acknowledge(dbg_ack) signal generated by the controller. During powerdown, the external clock control logic can monitor this signal and whenever it is asserted, the clock to the core can be enabled; else it can be disabled as long as powerdown_out signal is high. Below block diagram shows one method of implementation.



When the core is not in powerdown, powerdown clock is same as system clock. During powerdown it is switched off; it is turned on only in debug mode.

Case2:
To further reduce the power consumption during debugger operation within the powerdown, clock can be enabled only to the registers required for the debugger and clock can be disabled to the rest of the registers. Below block diagram shows one method of implementation.



When the core is not in powerdown, powerdown clock and deep powerdown clock are same as system clock. During powerdown:
- powerdown clock is switched off; it is turned on only in debug mode.
- deep powerdown clock is completely switched off.

9. Memories

There are two independent memories in LPDSP32-V3 to store program and data,

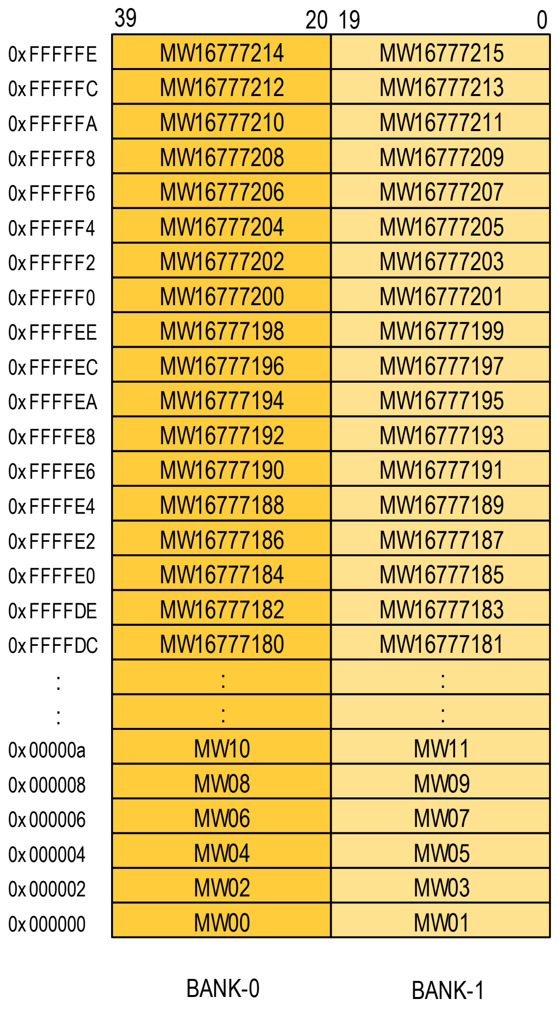
- Program memory (PM)
- Data memory (DM)

9.1 Program memory (PM)

Program memory is a single port synchronous memory. In a single cycle 40-bits are accessed from the program memory, but for the compiler it is treated as a 20-bit memory.

This is the program memory (PM) configuration.

- Address bus (pb_a) width 24-bit
- Write data bus (pb_w) width 40-bit
- Read data bus (pb_r) width 40-bit
- Address range: 0x000000 ~ 0xFFFFFF Size: 20(W) x 16777216(H)
- Instructions are stored in Big-endian format.
- While storing data on PM it is stored in Big-endian format.



	39 20	19 0
0xFFFFFE	MW16777214	MW16777215
0xFFFFFC	MW16777212	MW16777213
0xFFFFFA	MW16777210	MW16777211
0xFFFFF8	MW16777208	MW16777209
0xFFFFF6	MW16777206	MW16777207
0xFFFFF4	MW16777204	MW16777205
0xFFFFF2	MW16777202	MW16777203
0xFFFFF0	MW16777200	MW16777201
0xFFFFEE	MW16777198	MW16777199
0xFFFFEC	MW16777196	MW16777197
0xFFFFEA	MW16777194	MW16777195
0xFFFFE8	MW16777192	MW16777193
0xFFFFE6	MW16777190	MW16777191
0xFFFFE4	MW16777188	MW16777189
0xFFFFE2	MW16777186	MW16777187
0xFFFFE0	MW16777184	MW16777185
0xFFFFDE	MW16777182	MW16777183
0xFFFFDC	MW16777180	MW16777181
:	:	:
:	:	:
0x00000a	MW10	MW11
0x000008	MW08	MW09
0x000006	MW06	MW07
0x000004	MW04	MW05
0x000002	MW02	MW03
0x000000	MW00	MW01
	BANK-0	BANK-1

9.2 Data memory (DM)

There is a single contiguous data memory (DM), which is partitioned into three sub regions.

Data memory-A (DM-A):

Address range:	0x000000 ~ 0x7FFFFF
Size:	8388608(Depth) x 8(Width) [8MByte, (4 + 4) Banks]

Data memory-B (DM-B):

Address range:	0x800000 ~ 0xBFFFFF
Size:	4194304 (Depth) x 8(Width) [4MByte, 4 Banks]

Data memory-IO (DM-IO):

Address range:	0xC00000 ~ 0xFFFFFF
Size:	4194304 (Depth) x 8(Width) [4MByte, 4 Banks]

Data memory supports parallel data transfer (load/store) operations. During parallel data transfer; data and address buses of both the data memories DM-A and DM-B are used.

During the parallel data transfer only integer types (32bits) are accessed, it is not possible to do dual load/store for byte and short data types.

For the single data transfer complete data memory (DM) space can be accessed linearly and data and address buses of data memory-A (DM-A) are used.

Data memory (DM-A, DM-B and DM-IO) is byte (8-bit), short (16-bit) and int (32-bit) accessible.

In the data memory data is aligned byte wise. To access short, int and long long data, multiple bytes are accessed as shown below.

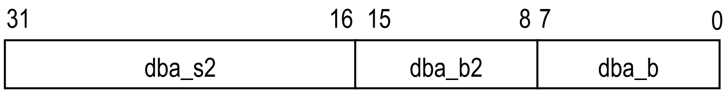
Sr.	Access type	Bits	No. of bytes accessed
1	byte	8-bit	1
2	short	16-bit	2
3	integer	32-bit	4
4	long long	64-bit	8

Accessing long long (64-bit) data type is also supported, which is limited to only data memory DM-A. Alias name for 64-bit data memory-A is long data memory-A, known as (LDMA) .For the 64-bit data transfer (load/store), data is combined into two 32-bits and two data buses are used for write (dba_w, dbb_w) and read (dba_r, dbb_r). However for addressing, address bus (dba_a) is used.

In the data memory, data is stored in Little-endian format.

9.2.1 Data memory (DM) data and address buses

For the byte, short and integer access data bus (dba) is used which is divided into three sub types as shown below.

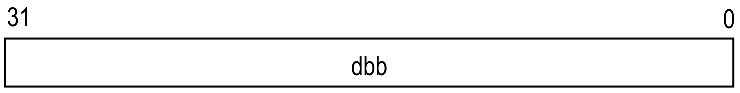


Data bus-A dba {16, 8, 8}

dba_b[7:0] : Used in byte, half word and word access

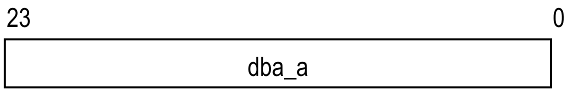
dba_b2[7:0]: Used in half-word access

dba_s2[15:0]: Used in word access

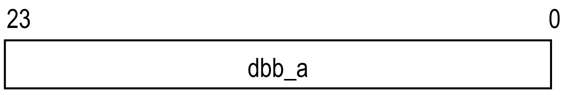


Data bus-B dbb {32}

Used in dual word and long(64bit) accesses



Address bus-A dba_a {24}



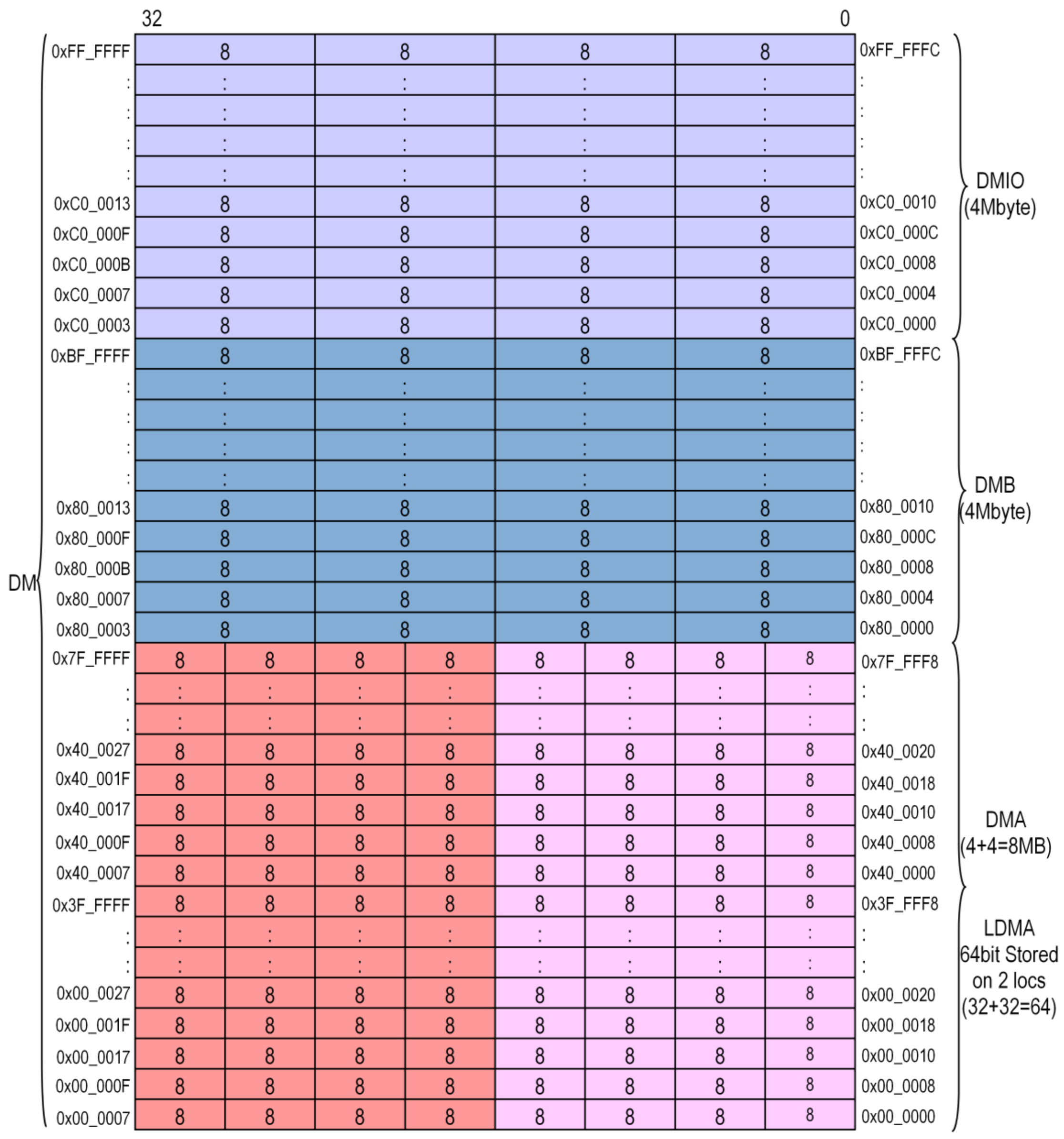
Address bus-B dbb_a {24}

9.2.2 I/O Memory space

Input/Output devices for the LPDSP32-V3 are memory mapped. I/O memory space is 4M Bytes. It is addressed by address generator unit (aalu0). Address and data buses used for I/O area are dba_a and dba respectively. From a programmer's point of view, I/O read and write behaves same as a memory read and write.

9.2.3 Data memory (DM) structure

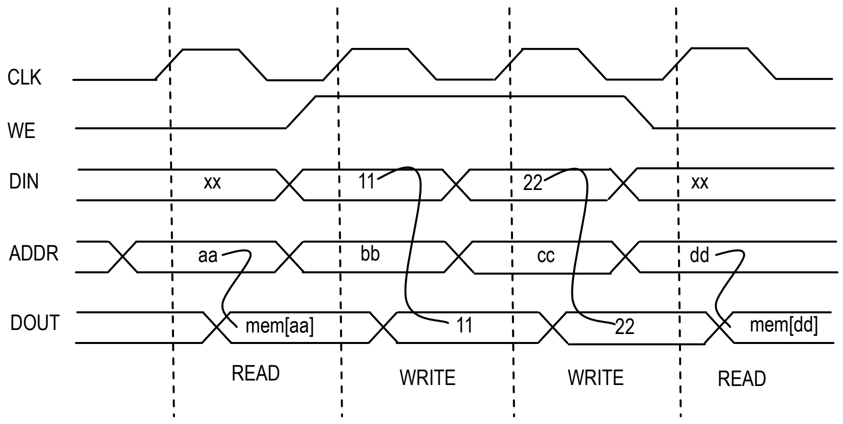
This is the data memory (DM) structure.



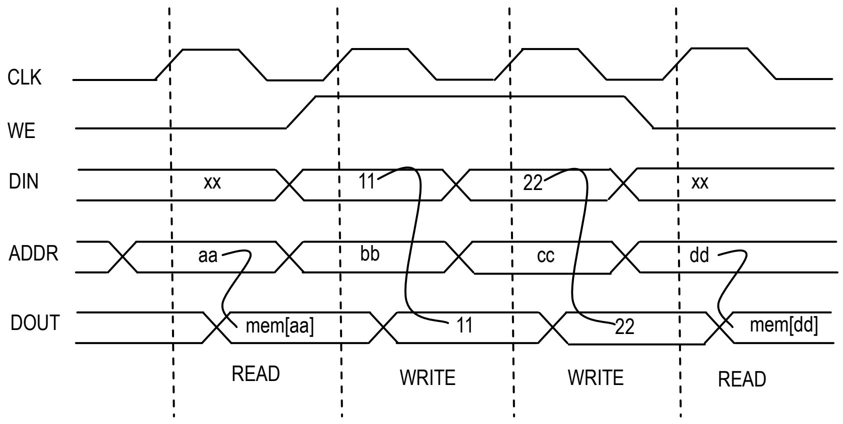
9.3 Memory timings

9.3.1 Data memory (DM) timings

Write operations are synchronous to the rising edge of the clock. The data on the DIN port is written into the memory location selected by the address on the rising edge of the clock when WE is active.



9.3.2 Program memory (PM) timings



9.4 Memory wait states

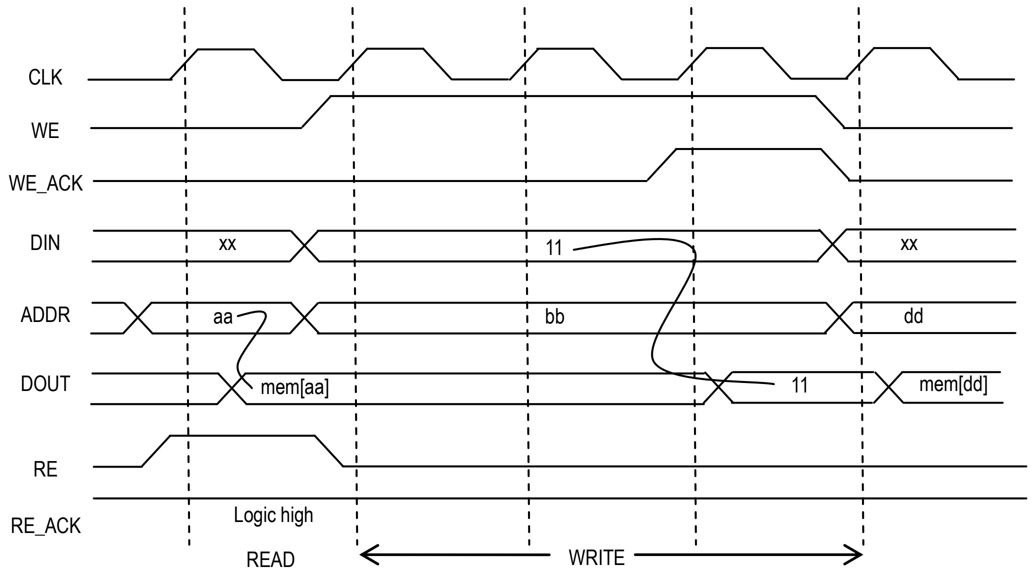
In LPDSP32-V3 an option is provided to interface the core with memories or memory mapped peripherals which cannot perform a read or write operation within one cycle. Once the core initiates a memory read or memory write, it waits for an acknowledge signal from the memory indicating the completion of the read or write operation. During this waiting period none of the registers(including the registers inside the controller) are updated, so the processor does not perform any operation and remains in the same state. As soon as the acknowledge signal is received the core proceeds with its operations. Wait states can be used for both data memory(DM) and program memory(PM).

Memory wait states can be used when a slow memory such as a Flash memory is used for storing instructions or data. It is also useful when interfacing peripheral IPs which have a certain timing protocol for register read/write operations and need more than one cycle to complete the operation. In such cases, an interface logic is required between the core and slow memories or peripherals to generate the acknowledge signals. When a combination of normal (memories without wait states) and slow memories are used, the acknowledge signals for the normal memories should be set to logic high, which is equivalent to zero wait states.

The below timing diagrams show examples for write and read timings with two wait cycles.

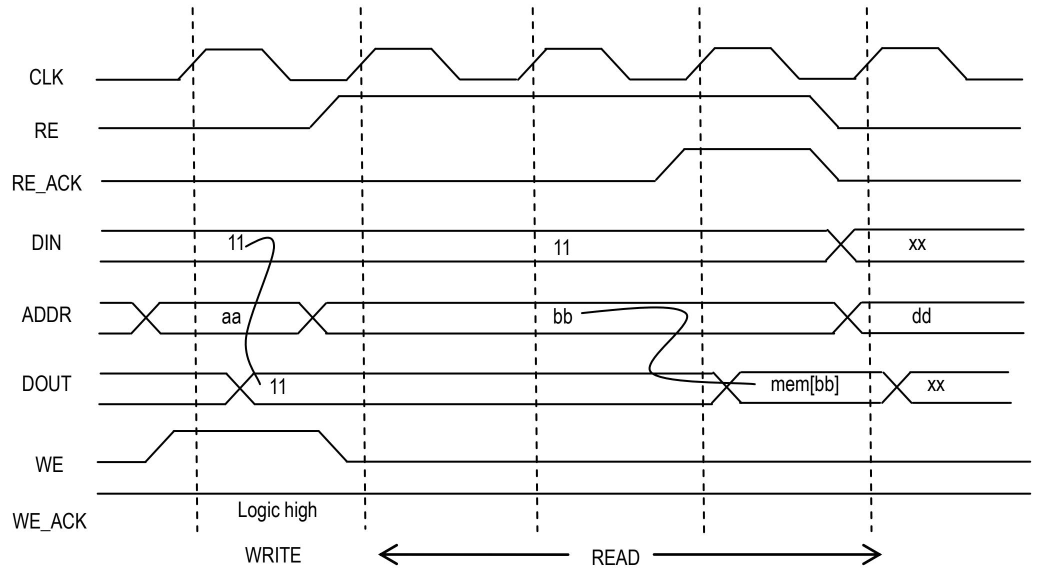
9.4.1 Memory write with wait states

This example shows a memory write operation performed with wait states and memory read operation performed without wait states.



9.4.2 Memory read with wait states

This example shows a memory read operation performed with wait states and memory write operation performed without wait states.



10. Summary of instructions and its operations

This section gives a summary of instructions supported in LPDSP32-V3 and its operations. Instructions are categorized into different sub types as shown below.

10.1 Instruction types

Sr.	Instruction sub type	Instruction width (Bits)	Notation	Category
1	Arithmetic	20	A	Short
2	Short control	20	A_Cntrl	Short
3	Move	20	M	Short
4	Arithmetic & Move	40	AM	Long
5	Long	40	L	Long

There are 20-bit and 40-bit mixed length instructions. Depending upon the type of the instruction, a single instruction (A or M) can perform a single operation or parallel operations in a single processor cycle. However Arithmetic and Move (AM) instructions can perform 4 parallel operations in a single cycle. Long instructions (L) are used for the direct operations like direct load/store, move, jump and subroutine calls.

10.2 Operations

Above instructions can perform the following operations:-

Sr.	Operation				Explanation
1	Arithmetic(A)				Single arithmetic
2	Arithmetic(a0)		Arithmetic(a1)		Parallel arithmetic
3	Move(M)				Single move
4	Move(a)		Move(b)		Parallel move
5	Arithmetic(A)		Move(M)		Single arithmetic & single move
6	Arithmetic(a0)	Arithmetic(a1)	Move(a)	Move(b)	Dual arithmetic & dual move
7	Short control (A_cntrl)		Move(M)		Short control & move
8	Arithmetic(A)		Short control (M_cntrl)		Arithmetic & Short control
9	Long (L)				Long

10.3 Single operations

These are the three single operations that can be performed:

- Single arithmetic operations
- Single move operations
- Long operations

10.3.1 Single arithmetic operations

Single arithmetic operations include all operations done in alu0 and mpy0.

Single arithmetic operations are always performed in the alu0 and mpy0 functional units.

10.3.2 Single move operations

Single move operations include register to register move and memory load/store operations.

For this operation, both the data move buses (dba, dbb) are used.

Address generation and pointer updating is performed in the aalu0 functional units.

Address generation unit (aalu0) uses all the address registers (a0 ~ a7) and offset registers (c0 ~ c3) for the address calculation.

10.3.3 Long operations

Long operations include direct memory load/store, long immediate assignment and control operations. These operations influence the whole processor.

10.4 Parallel operations

In the parallel operations, various combinations of operations are possible.

- Parallel arithmetic
- Parallel move
- Single arithmetic and single move
- Dual arithmetic and dual move
- Short control and move
- Arithmetic and short control

10.4.1 Parallel arithmetic

Parallel arithmetic includes, dual arithmetic (add/sub) or dual multiply accumulate (mac) operations. Dual arithmetic operations are performed in the alu0 and alu1.

And for the dual multiply accumulate (mac) operation, one multiply accumulate operation is performed in the (mpy0 and mac0) and other is performed in the (mpy1 and alu1).

Here, functional unit (mac0) is used only for doing addition or subtraction after the multiplication.

10.4.2 Parallel move

Parallel move includes only dual memory load/store operations.

For doing dual memory load/store operation, move (a) is performed on the data memory-A (DMA) using data bus A (dba_r) for read and data bus (dba_w) for write operation.

For the other move (b) which is performed on the data memory-B (DMB) using data bus B (dbb_r) for read and data bus (dbb_w) for write operation.

Address generation unit (aalu0) and address registers (a0 ~ a3) are used for the address generation for the move (a). Address generation unit (aalu1) is used and address registers (a4 ~ a7) are used for address generation for the move (b). Offset registers (c0 ~ c3) are shared by both the address generation units.

For parallel move operations the following combinations are possible.

Move (a)	Move(b)
Load	Load
Store	Store
Load	Store
Store	Load

10.4.3 Single arithmetic and single move

In this type of operation, a single arithmetic operation and a single move operation are performed simultaneously.

10.4.4 Dual arithmetic and dual move

In this type of operation, two arithmetic operations which can be either add/sub or two multiply accumulate operations and two data memory load/store operations are performed simultaneously.

10.4.5 Short control and move

In this type of operation, one short control operation (short jump, indirect jump, indirect subroutine call and subroutine return) and on the other side a move operation can be performed. The move operation can be a single operation or parallel operations as described above.

Here short control operation is put on the arithmetic (A) side of the instruction word.

10.4.6 Arithmetic and short control

In this type of operation, arithmetic (A) operation is performed, which can be either a single operation or parallel operations. On the other side, a short control operation (short jump, indirect jump, indirect subroutine call and subroutine return) is performed.
Here short control operation is put on the move (M) side of the instruction word.

For the complete details of the instructions and its operations please refer the instruction manual [References:4]

11. On Chip debugging (OCD)

This section gives an overall view of on-chip debugging hardware. It does not give complete hardware and software implementation details of the OCD.

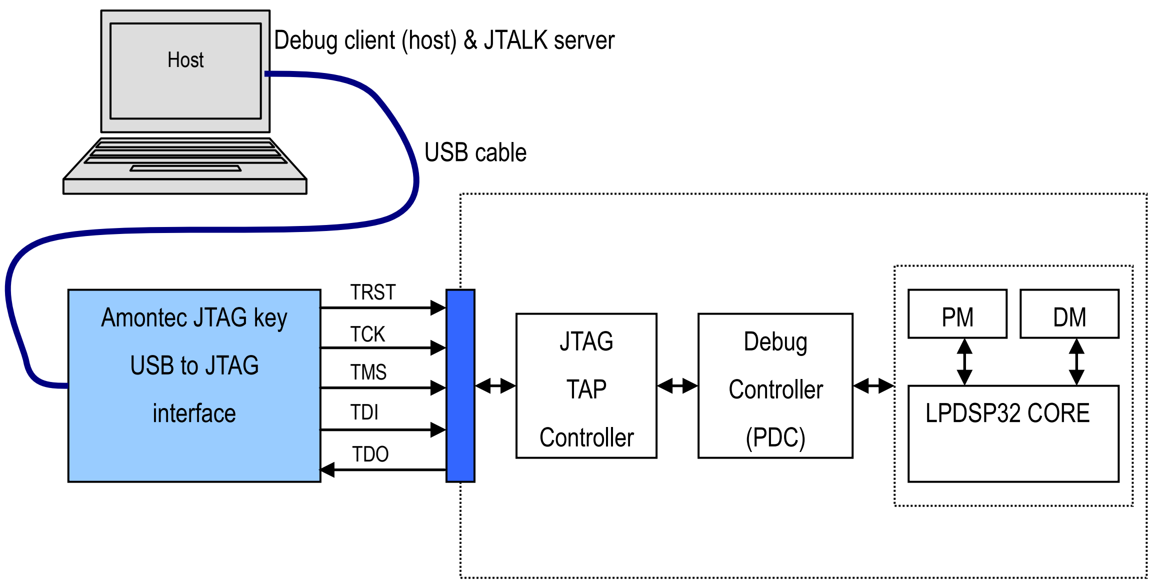
11.1 Overview and debugging features

For testing and debugging the application software on the LPDSP32-V3 core, as well as testing the core itself on the hardware (FPGA prototype or ASIC) a JTAG based on-chip debugger with a graphical user interface is available. On-chip graphical user interface is almost similar to instruction set simulator (ISS), with the following features available in real time.

- Loading of application program in (PM) and application data in (DM).
- Viewing and modifying program memory (PM), data memory (DM) contents.
- Viewing and modifying the register contents.
- Core reset.
- Debug mode and normal processing mode.
- Hardware breakpoints on program counter (PC) and watchpoints on data memory locations during store operations (combined maximum 8).
- Single step execution.
- Tracing the assembly instruction for corresponding C code.
- Hosted IO.
- Software break points.
- Accessing the core for debugging remotely. etc.

11.2 On-chip debug environment

This figure gives an overall view of the on-chip debug environment.



11.2.1 JTALK server

JTALK server is a PC based program that interfaces to the lpdsp32_debug_client. The JTALK server receives commands from the debug client on a TCP/IP socket. It calls driver routines to generate JTAG signals on the device connected to the USB port.

11.2.2 Debug client

Debug client issues commands to debug controller to get the information of all memories and registers. By using the debug client, user can load the memories and run the application programs and observe the results in memory and data registers.
There are two hardware blocks called JTAG tap controller and debug controller used for the on chip debugging.

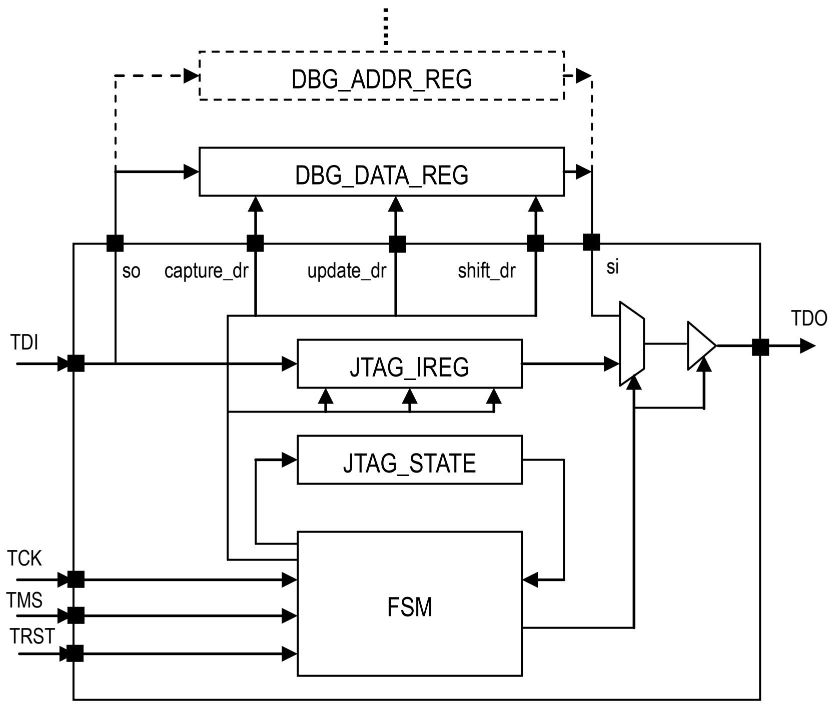
11.3 JTAG tap controller

This block provides serial communication interface to the debug controller. JTAG tap controller is a FSM controlled by TMS signal. Depending on the state of the FSM tap controller, it places either JTAG instruction register (jtag_ireg) or PDC registers (DBG_DATA_REG, DBG_ADDR_REG…) in the scan path.

This module is accessed by five pins:

Sr.	Pin	Name	Explanation
1	TRST	Test reset	Asynchronous reset for the TAP controller state machine. This signal is active low. TRST is an optional signal
2	TCK	Test clock	The shifting of data through scan registers and the state transitions of the TAP controller state machine are synchronous to TCK. TDI and TMS inputs are sampled on the rising edge of TCK.
3	TMS	Test mode select	The TMS signal selects the next state in the TAP state machine. It is sampled on the rising edge of TCK.
4	TDI	Test data input	Provides a serial input data stream to the JTAG and PDC scan registers. It is sampled on the rising edge of TCK.
5	TDO	Test data output	Provides a tri-state capable serial output data stream from the JTAG and PDC scan registers. It is driven in the Shift-DR and Shift-IR states of the TAP controller state machine. Changes in the state of this signal occur on the falling edge of TCK.

11.3.1 Block diagram of JTAG tap controller



11.4 Debug controller

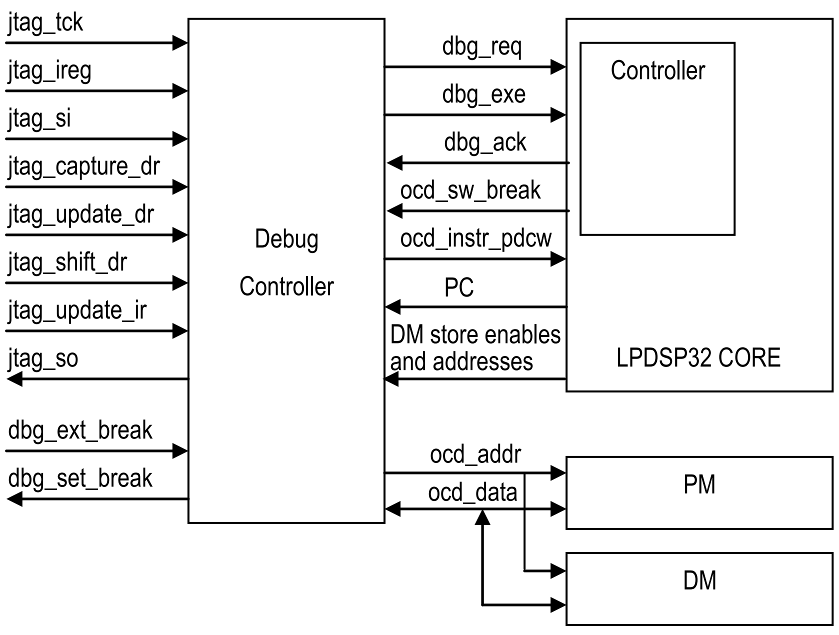
The debug controller or Processor Debug Controller(PDC) decodes the instructions in the JTAG instruction register and performs the necessary actions. To summarize, debug controller handles below tasks.

- Performs debug load/store operation.
- Break point detection.
- Watch point detection.
- Contains JTAG scan registers (DBG_ADDR, DBG_DATA, DBG_INSTR, DBG_STATUS)
- Decodes the debug instructions like (resume, step, reset, request, execute…)
- Synchronization between test clock (TCK) and processor clock.

✍ **Important note:**

Debug controller uses data memory DMIO [0xC00000] address location for reading and writing the registers. Therefore this address location cannot be used other than for on-chip debugging.

11.4.1 Block diagram of debug controller



11.4.2 Debug instructions

There are various debug instructions which are issued by debug client to debug controller by scanning specific code in the jtag_ireg register. Then, the debug controller decodes the instruction issued by the debug client and executes it. Debug controller is assigned with a unique core ID (Core id = 1). All debug instructions are summarized here.

Sr.	Instruction	Opcode	Explanation with actions
1	DBG_REQUEST_INSTR	11'b00000010001	This is a debug request instruction, puts the core into debug mode - scan request instruction into jtag_ireg register - Debug controller decodes request instruction and asserts dbg_req signal - PCU stops fetching new instructions - Freeze the PC - Disable interrupt - Complete all fetched instructions
2	DBG_RESUME_INSTR	11'b00000010010	This instruction releases the core from debug mode - scan resume instruction into jtag_ireg register - Debug controller decodes resume instruction and de-asserts dbg_req signal - Controller resumes normal operation
3	DBG_RESET_INSTR	11'b00000010011	This instruction resets the core
4	DBG_STEP_INSTR	11'b00000010100	This instructs the core to fetch and execute a single instruction from PM
5	DBG_EXECUTE_INSTR	11'b00000010101	This instructs the core to execute the instruction present in the DBG_INSTR_REG.
6	DBG_DM_LOAD_INSTR	11'b00001000000	This instruction loads the data from DM
7	DBG_DM_STORE_INSTR	11'b00001000001	This instruction stores the data to DM
8	DBG_PM_LOAD_INSTR	11'b00001000010	This instruction loads the data from PM
9	DBG_PM_STORE_INSTR	11'b00001000011	This instruction stores the data to PM
10	DBG_BPbbb_ENABLE_INSTR	11'b000001bbb00	This instruction enables breakpoint bbb
11	DBG_BPbbb_EXPORT_INSTR	11'b000001bbb01	This instruction exports breakpoint bbb to other cores (multi core)
12	DBG_BPbbb_DISABLE_INSTR	11'b000001bbb10	This instruction disables breakpoint bbb
13	DBG_WPbbb_ST_DM_dba_a_ext_ENABLE_INSTR	11'b0001bbb0000	This instruction enables store watchpoint bbb on DM
14	DBG_WPbbb_ST_DM_dbb_a_ext_ENABLE_INSTR	11'b0001bbb0001	This instruction enables store watchpoint bbb on DMB
15	DBG_SYNC_REQUEST_INSTR	16'b1111100000010001	Used to synchronize multi core debug request
16	DBG_SYNC_RESUME_INSTR	16'b1111100000010010	Used to synchronize multi core resume
17	DBG_SYNC_RESET_INSTR	16'b1111100000010011	Used to synchronize multi core reset
18	DBG_SYNC_STEP_INSTR	16'b1111100000010100	Used to synchronize multi core step execution

11.4.3 Debug registers

The processor debug controller contains a set of registers that govern its behavior, and that are accessible via the JTAG interface. Most PDC registers are bi-directional. Communication is described from the point of view of the PDC. Registers that are written by the debug client and read by the PDC, are identified as input registers. Registers that are written by the PDC and read by the debug client, are identified as output registers.

DBG_DATA_REG (bi-directional, 32 bit)

The debug data register. This register is used to transport data values from the debug client to the core registers or data memory, and vice versa.

DBG_ADDR_REG (bi-directional, 24 bit)

The debug address register. The debug client writes to this register to specify the memory address when the PDC is instructed to execute a memory load or store operation. The debug client reads from this register to obtain the current value of the program counter (PC) register of the core.

DBG_INSTR_REG (bi-directional, 40 bit)

The debug instruction register. This register is used to transport the LPDSP32 instructions from the debug client to the program memory, and vice versa. The debug client also uses this register to supply LPDSP32 instructions to be executed by the LPDSP32 core. A typical example is to execute a LPDSP32 instruction that moves a certain register to the DBG_DATA_REG register such that the debug client can read the register value from there.

DBG_STATUS_REG (output, 16 bit)

The status register is a 16 bit register, of which 15 bits are currently used. The bits of this register have the following meaning.

Bit	Description
0	Set when the processor is in debug mode, cleared otherwise.
1 - 8	Set when breakpoint/watchpoint (0 - 7) are hit respectively.
9	Set when the processor stopped execution due to a local breakpoint hit.
10	Set when the processor stopped execution due to an external breakpoint hit.
11	Set when a local breakpoint hit is exported to other cores.
12	Set when a step debug instruction is executed.
13	Set when a core resume instruction is executed.
14	Set when a software breakpoint is hit.

11.4.4 Debug interface

The DM access done by the debugger has been changed to always doing 32-bit access, instead of always doing 8-bit access, which was used in the previous version LPDSP32-V2. Even with this change, 8 bit and 16 bit updates are still possible. The below two points explain how it has been achieved in the debug client software:

- First, the Checkers GUI translates all memory access (DM,SDM,WDM,LDMA) to 8-bit access on the root memory DM.
- Secondly, the Checkers_pdc_interface class translates all 8-bit access to 32-bit access. For example, there is a write-merge-buffer, such that when doing consecutive byte stores on DM, a single 32-bit store is done. When only updating a single byte, Checkers_pdc_interface first reads the full 32-bit word, updates the byte, and then stores the updated word.

Some consequences as a result of the change to 32-bit access are:

- This will speedup memory/register access: as a single 32-bit transfer is done (instead of 4 single byte transfers).
- The area cost is somewhat higher, as the ocd_data_register(DBG_DATA_REG) increases from 8-bit to 32-bit
- From the debugger it is possible to change any bits of peripheral registers(mapped to DMIO) even if they have a 32 bit interface. When LPDSP32-V2 debugger had an 8 bit interface and peripherals had a 32 bit interface, from the debugger it was possible to change only the lower 8 bits of the registers and not the upper bits.

11.4.5 Hardware breakpoints and watchpoints

The hardware breakpoints and watchpoints implementation use the same hardware (e.g. registers and address comparators). The total number of breakpoints and watchpoints that can be used are eight. The debug client has eight registers for holding the addresses of breakpoints and watchpoints. Depending on whether a hardware breakpoint is set or a data memory watchpoint is set, each register is compared against the program counter value or the data memory address of the core respectively. Since watchpoints are enabled only for store operations, the comparison is done only when the core data memory store signal is active.

Each breakpoint/watchpoint register can also be enabled or disabled. When a breakpoint or watchpoint hit occurs, the core is put in debug mode. A breakpoint/watchpoint hit event can be exported to other cores in a multi-core system.

11.4.6 Software breakpoints

An alternative solution to hardware breakpoints is software breakpoints. Wherever software breakpoint is set, the debug client replaces the instruction with a software break instruction. In this way, we can set an unlimited number of breakpoints in the debug client (the only condition is that the instructions are in program RAM, such that the debug client can replace the instruction with the swbreak instruction). In this case, hardware breakpoints are only needed for code in ROM. In the debug client there is an option to configure the breakpoint as hardware or software breakpoint.

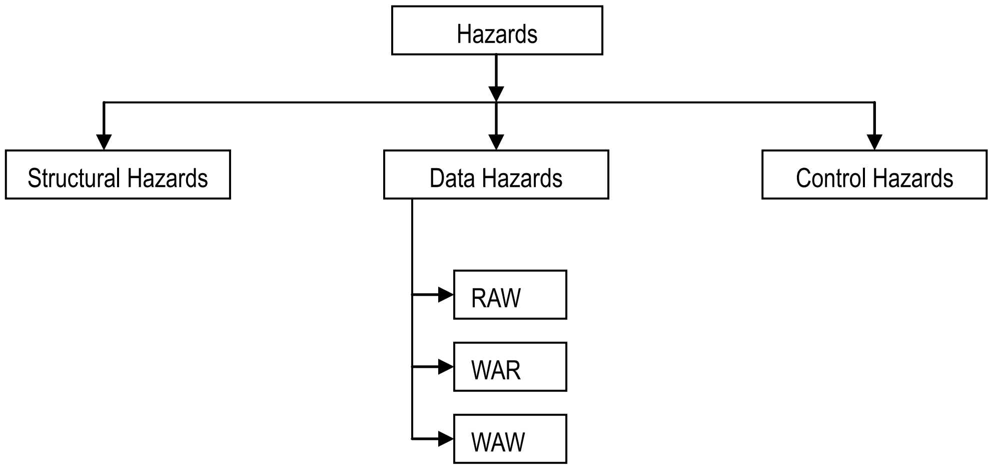
Software break is a single-word, 2-cycle break instruction which will send an 'ocd_swbreak' signal to the debug controller. This indicates a breakpoint hit and the core is put in debug mode. This will be informed to the debug client through a bit in the PDC status register, DBG_STATUS_REG.

12. Hazards

Before going into, what types of hazards are there in LPDSP32-V3, let's understand the concept of hazards. Hazards are the in born artifacts of the pipeline execution of the instructions.

Mainly there are three types of hazards.

- Structural hazard
- Data hazard
- Control hazard



12.1 Structural hazard

Also called as resource hazard, this can occur when same hardware resource (address bus) is used in different pipeline stages by the different instructions.

12.2 Data hazard

This hazard occurs when the same register is accessed in different pipeline stages by different instructions. A typical example is the read-after-write (RAW) hazard that occurs when a register is written in the E1 stage and that register is also read (by another instruction) in the D stage.

12.3 Control hazard

This hazard occurs when a jump instruction is issued in D stage, which results in a change of the program counter in a later pipeline stage (E1). Control hazards are more in case of deep pipeline stages. In LPDSP32-V3 since there are only 3 pipeline stages, the overhead of control hazard is less.

12.4 Software stall

All the hazards (structural, data and control) can be solved by organizing the sequence of instructions in such a way that the hazard does not occur. This approach is called software stalling. By applying “sw_stall” rules, compiler removes the hazards.

By predefined “sw_stall” rules, compiler takes care that no such instruction sequence is generated which triggers the hazard. While doing scheduling, compiler tries to avoid the hazards by reordering the instructions. If reordering can’t avoid hazards only then compiler inserts the “NOP” instruction.

12.4.1 Software stall (structural hazard)

Software stall is applied when the same resource (address bus) is used by different instructions in different pipeline stages. Essentially this is done to avoid address bus conflict on either of the address buses (dba_a) or (dbb_a) while performing a load/store operation.

In LPDSP32-V3, the same address bus is used for load and store operations; load operation is done in the D stage and store happens in the E1 stage, but the address is calculated in the same D stage for both the load and store operations. Therefore there is a restriction that, load instruction next to store instruction is not allowed, a “NOP” is required between the two instructions or a rearrangement of instructions is required.

Example:

~~[a1+c1] = ra0   //store~~

~~ra0 = [a0 + c0] //load~~

```
[a1+c1] = ra0   //store
nop             //insert nop or rearrange
ra0 = [a0 + c0] //load
```

```
ra0 = [a0 + c0] //load
[a1+c1] = ra0   //store
```

12.4.2 Software stall (data hazard)

To avoid the read after write (RAW) data hazards software stall is applied.

In LPDSP32-V3 software stall is applied for the below register write (move) instructions.

- Writing to status register srFlags (* setting SR bits through an instruction).
- Writing to address registers a0-a7
- Writing to loop start (LB), loop size (LSZ) and stack pointer (SP) registers

For all of the above registers write action happens in the E1 stage. After writing to these registers, immediately in the next cycle the same register cannot be read, at least 1 cycle offset is required.

Example (SR register):

```
~~sr = 0x7       //setting v,n,z flags in SR~~
~~if (n) jps 10 //next instruction is a conditional jump~~
```

```
sr = 0x7       //setting v,n,z flags in SR
nop            //nop or rearrange the instructions
if (n) jps 10 //next instruction is a conditional jump
```

Example (address registers a0 ~ a7):

```
~~c0 = 1        //setting offset register~~
~~a0 = 252      //setting address register~~
~~ra0 = [a0+c0] //immediately using same addr. reg.~~
```

```
c0 = 1        //setting offset register
a0 = 252      //setting an address register
nop           //added sw stall by nop
ra0 = [a0+c0]//immediately using the same addr. reg.
```

```
a0 = 252      //setting an address register
c0 = 1        //setting offset register (rearrange)
ra0 = [a0+c0] //immediately using the same addr. reg.
```

Example (SP):

```
~~sp = 65512  //setting stack pointer value~~
~~ra0 = sp[4] //sp indexed addressing~~
```

```
sp = 65512  //setting stack pointer value
nop         //added sw stall by nop or rearrange
ra0 = sp[4] //sp indexed addressing
```

Example (LB and LSZ):

```
~~c0 = 1       //setting offset register~~
~~a0 = 0       //setting address register~~
~~lb0 = 252     //setting loop start register~~
~~lsz0 = 32     //setting loop size register~~
~~ra0 = a0+%0c1 //immed. using lsz0 for cyclic addr.~~
```

```
c0 = 1       //setting offset register
a0 = 0       //setting address register
lb0 = 252     //setting loop start register
lsz0 = 32     //setting loop size register
nop           //added sw stall by nop
ra0 = a0+%0c1 //immed. using lsz0 for cyclic addr.
```

```
a0 = 0       //setting address register
lb0 = 252     //setting loop start register
lsz0 = 32     //setting loop size register
c0 = 1       //setting offset register
ra0 = a0+%0c1 //immed. using lsz0 for cyclic addr.
```

12.5 Hazards related to hardware loop

At the last instruction of a hardware loop, end of loop test takes place and the program counter PC is updated when the HW loop is not complete. If a jump instruction is positioned at last instruction of a HW loop, there is a conflict of PC update and jump instruction. To avoid this hazard, jump instruction should not be positioned near the end of loop instruction.
End of loop operation is done in the instruction fetch (F) pipeline stage, but jump happens in the D stage.

Example:

```
lp 4 4      //hw loop
inst_ds1     //delay slot 1
inst_ds2     //delay slot 2
inst_1      //loop start
inst_2      //some inst. in hw loop
jpsdb [a0]  //jump before last instr.
inst_4      //last instr. of the loop
inst_5      //loop end here
```

```
lp 4 4      //hw loop
inst_ds1     //delay slot 1
inst_ds2     //delay slot 2
jpsdb [a0]  //keep offset 1 to 2 cycles.
inst_1      //
inst_2      //some inst. in hw loop
inst_4      //last instruction of the loop
inst_5      //loop end here
```

In the above example, it is shown that if there is a jump instruction just before the end of loop instruction, there will be a conflict of simultaneous PC update. To avoid this conflict, the jump instruction must be positioned at least two instructions before the end of loop instruction.

12.5.1 Control hazards related to hardware loop

There are various control hazards related to hardware loop. These hazards occur as a result of PC conflict. One of these is, in case of nested HW loops, two loop end addresses should not be the same. Software stall is applied in this case, if there are no instructions to keep.
Also there are some control hazards specially related to end of loop in HW loop instruction.
Either of these instructions (direct jump, direct conditional jump, indirect jump and indirect conditional jump) should not be positioned just before the loop end instruction.

12.6 Bypassing

Bypassing, also known as data forwarding, is an operation where in the processor bypasses the registers and forwards the contents of the registers wherever needed in the pipeline, at an additional hardware cost.
Bypassing is only applicable to avoid data hazard, read after write (RAW).
By doing bypassing on some of the registers, pipeline stalls (additional NOP insertion) is avoided.
By adding the bypass logic for some of the registers, processor performance is increased by reducing MIPS count.
In LPDSP32-V3, bypass is implemented for srFlags (V N Z) and offset registers (c0 ~ c3)

12.6.1 Bypass (srFlags)

Status register flags V, N and Z are computed in the E1 stage, but while doing conditional jump these flags are required in the D stage of the pipeline. Bypass is implemented to read these flags from E1 stage to D stage. Therefore the below sequence of instructions is possible.
Example:

```
cmp (ra0, ra1) //compute the flag in E1 stage
if (cc) jps 10 //cond. jump reading flags in D stage
```

12.6.2 Bypass (offset registers c0 ~ c3)

Offset register (C) is written in the E1 stage by short immediate (11bit signed) move instruction as well as immediate (8bit signed) assignment to offset register which is implemented in alu0.

For doing address computation aalu0 and aalu1 need this offset register’s (C) value in the D stage.

Address computation is done in pipeline stage D.

Bypass is implemented to read these registers from E1 stage to D stage.

Therefore the below sequence of instructions is possible.

Example:

```
a1 = 40   //immediate assignment to address register
c0 = 255  //imm. assignment to offset register in E1 stage
ra0 = [a1 + c0] //load address is computed in D stage
```

13. References

1. LPDSP32-V3 Assembly programmer’s manual
2. LPDSP32-V3 Interrupt support manual
3. User guide-LPDSP32
4. LPDSP32-V3 Instruction encoding manual

14. Revision History

Version	Revision History	Date
1.0	Initial version	June 2011
1.1	Updated version(corrections to content and block diagrams)	March 2012
1.2	Company logo is changed	August 2013
1.3	Company name is changed	October 2014