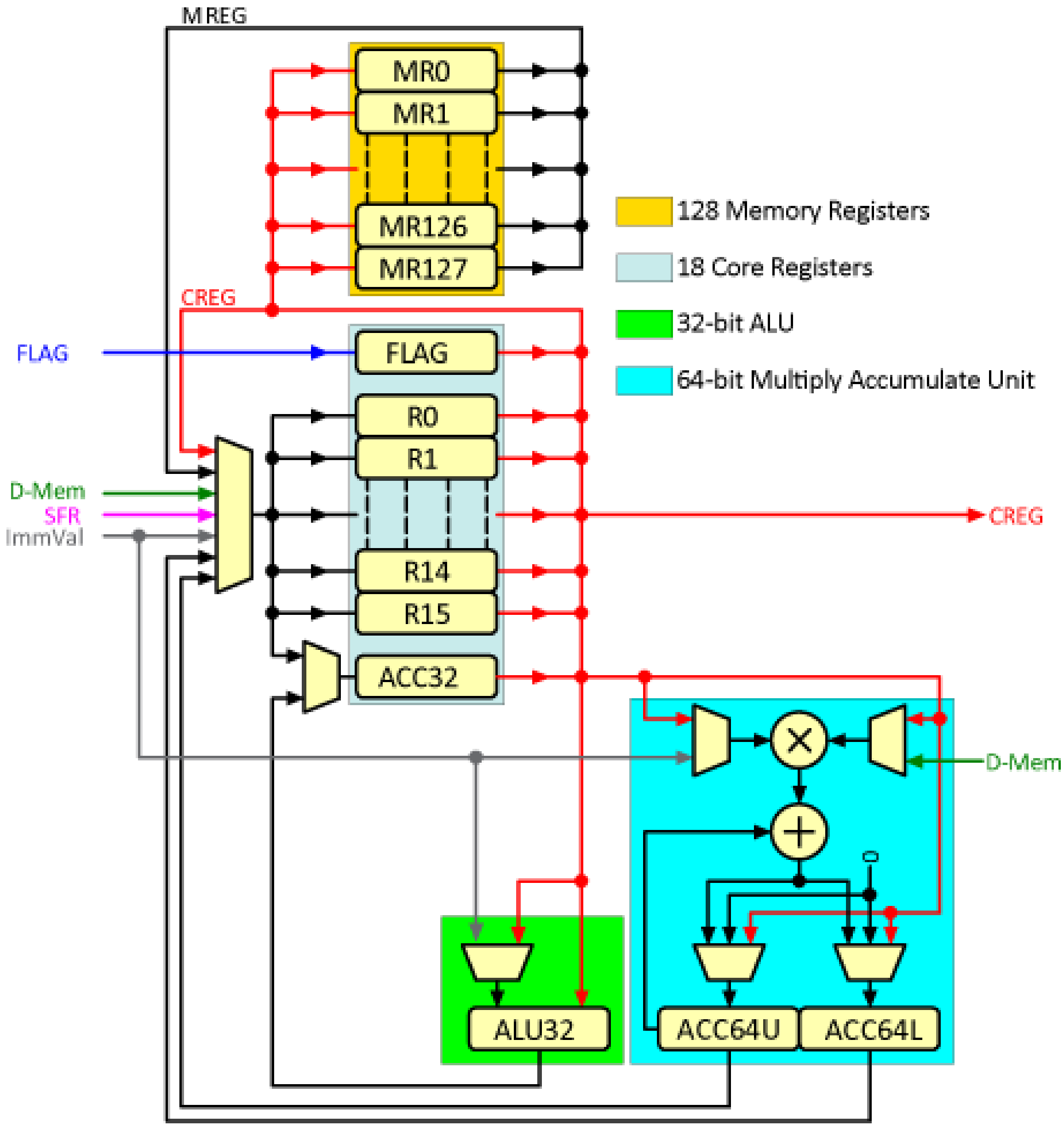
FX Core Instruction Set
V1.0 Feb 2020

Table of Contents

Introduction

Block Diagram



The FX Core has 3 sets of registers, 20 DSP/ALU registers (19 core and a "non-core" 64-bit accumulator ACC64) , 128 32-bit MREG registers and special function registers (SFR). DSP/ALU registers are faster for the DSP/ALU to access and use.

The number of instructions that can be executed in a single sample period is dependent on the instructions used. Instructions take one or more clocks to execute so the total number of clocks a program takes depends on the mix of instructions. Programs have about 3500 clocks per sample period at 48KHz sample rate. There are more clocks available at lower sample rates.

The instruction RAM can hold 1024 instructions but the number of instructions that can be executed in a sample period depends on the mix of instructions and the sample rate the FXCore is being run at.

The FX Core assembler will estimate how many clocks have been used and warn the user when it exceeds 90% when the program is assembled.

The internal FLASH can hold 16 programs, when a program is selected the program is loaded into the instruction RAM and executed from there. Additionally each program has an associated header that will preload values into the core registers, MREG registers and some of the SFRs. This allows users to set initial values in their programs without wasting instructions.

On program change, while the instruction RAM is being loaded with the new program, the outputs are muted, the delay RAM is cleared and the initial values from the program headers are loaded into their target registers.

Registers and Memory

DSP/ALU core registers are 32-bit and are called R0 to R15, ACC32 and FLAGS. While ACC64 resides in the core of the chip it is not part of the core register bank and is not considered a core register.

Name	Encoding	Type	Remark
R0	00000	R/W	General use
R1	00001	R/W	General use
R2	00010	R/W	General use
R3	00011	R/W	General use
R4	00100	R/W	General use
R5	00101	R/W	General use
R6	00110	R/W	General use
R7	00111	R/W	General use
R8	01000	R/W	General use
R9	01001	R/W	General use
R10	01010	R/W	General use
R11	01011	R/W	General use
R12	01100	R/W	General use
R13	01101	R/W	General use
R14	01110	R/W	General use
R15	01111	R/W	General use/PARAM0
ACC32	10000	R/W	General use/result
FLAGS	10001	R	Read only

R0 – R15 are general purpose registers, ACC32 can be used as a general register but is also used as the destination register for many instructions. FLAGS is a read only register and contains flags useful to the user program. In some instructions more information is required than can fit in the 32-bit instruction field, in these cases R15 is used as an additional parameter register referred to as PARAM0.

ACC64 is the 64-bit accumulator for the 32x32 multiplier.

MREG registers are a bank of 128 32-bit registers called MR0 to MR127 that can be used for additional 32-bit storage. These registers can also be read indirectly by the CPY_CMX command allowing the registers to be preloaded by the .MREG directive and used as a lookup table.

Delay RAM is a 32Kx16-bit block and is the slowest to access.

Special function registers (SFRs), while not shown in the block diagram, are a collection of registers that provide specific functions or information to a program. Some SFRs are read only while others are write only. For a complete list of SFRs please see the FXCore datasheet or the “Special Function Registers” portion of the “Reserved Words” section later in this document.

FLAGS Register

The FLAGS register contains status flags that may be useful to a user program, this is a 16-bit LSB aligned register so flag values may be easily isolated with an instruction like ANDI.

Bit	Name	Meaning
15	OUT3OFLO	Output 3 overflow
14	OUT2OFLO	Output 2 overflow
13	OUT1OFLO	Output 1 overflow
12	OUT0OFLO	Output 0 overflow
11	IN3OFLO	Input 3 clip
10	IN2OFLO	Input 2 clip
9	IN1OFLO	Input 1 clip
8	IN0OFLO	Input 0 clip
7	XXX	RESERVED
6	XXX	RESERVED
5	TB2nTB1	0: Tap button "1" event 1: Tap button "2" event
4	TAPSTKY	TAP sticky event, user has pressed the tap button for longer than TAPSTKRLD
3	NEWTT	New tap tempo value in TAPTEMPO
2	TAPRE	Tap button release event, user stopped pressing the tap button
1	TAPPE	Tap button push event, user started pressing the tap button
0	TAPDB	Debounced tap button level, 0 if pressed and 1 if not pressed

Flags are valid for as long as the event occurs (i.e. overflow flags) or for 1 sample period (i.e. new tap value)

Instruction Format

The instruction is a 32-bit word broken into 3 primary fields:

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="7">"I" Field</td><td>0</td><td colspan="8">"R" Field</td><td colspan="16">"M" Field</td></tr>
</table>

31:25 – "I" field, instruction field

24 – Reserved, set to 0

23:16 – "R" field, generally a core register.

15:0 – "M" field, may be used to point into delay memory, used to address a second register if required, the 5-bit value for a shift, 16-bit coefficient, etc. LSB aligned field.

For Extended Operations both the R and M fields may be used in a different manner.

Symbols used in instruction descriptions

<table>
<tr><th colspan="2">FXCore notation conventions</th></tr>
<tr><td>+</td><td>Arithmetic addition</td></tr>
<tr><td>-</td><td>Arithmetic subtraction</td></tr>
<tr><td>*</td><td>Arithmetic multiplication</td></tr>
<tr><td>/</td><td>Arithmetic division</td></tr>
<tr><td>~</td><td>Bit inversion (1's compliment)</td></tr>
<tr><td>&</td><td>Bitwise AND</td></tr>
<tr><td>&&</td><td>Logical AND</td></tr>
<tr><td>|</td><td>Bitwise OR</td></tr>
<tr><td>||</td><td>Logical OR</td></tr>
<tr><td>⊕</td><td>Bitwise XOR</td></tr>
<tr><td>{A,B}</td><td>Concatenation of 2 values to create a larger value, i.e. {0x12, 0x34} results in the value 0x1234</td></tr>
<tr><td>[X]</td><td>The value in a register or at a specific memory address. X is a register number or a memory address. I.e [CREG0] (value in core reg 0), [FLAGS] (FLAGS sfr)</td></tr>
<tr><td>|X|</td><td>Absolute value of X</td></tr>
<tr><td>X^Y</td><td>X raised to the Y power</td></tr>
<tr><td>$X_{b:a}$</td><td>Bit range of b to a of value X</td></tr>
<tr><td>@REG</td><td>Indirect addressing, REG is used as a pointer into another memory</td></tr>
<tr><td>@@REG</td><td>Indirect addressing, REG is used as a pointer into delay memory without adding address counter</td></tr>
<tr><td>ImmX</td><td>An immediate integer value of X bits, unsigned</td></tr>
<tr><td>SImmX</td><td>An immediate integer value of X bits including a sign bit</td></tr>
<tr><td>S.X</td><td>An immediate signed fractional value in the range of -1.0 to +0.99.. of 1+X bits</td></tr>
<tr><td>SX.Y</td><td>An immediate signed real value with X integer bits and Y fractional bits, total length of 1+X+Y bits</td></tr>
<tr><td>CREG</td><td>Any core register is valid (CREG0-15, ACC32, FLAGS)</td></tr>
<tr><td>CREGNA</td><td>Only core registers 0 to 15 and ACC32 are valid</td></tr>
<tr><td>MREG</td><td>Memory register</td></tr>
<tr><td>MEM</td><td>Delay memory</td></tr>
<tr><td>SFR</td><td>Special function register</td></tr>
<tr><td>X<<Y</td><td>Left bit shift of X by the value represented by Y</td></tr>
<tr><td>X>>Y</td><td>Right bit shift of X by the value represented by Y</td></tr>
</table>

There is the 64-bit CPU register for 32x32 MAC ops. While in the core it is not considered a core register as it cannot be used directly in most instructions.

The FX Core assembler attempts to resolve values based on the type expected in an instruction, an ADDSI instruction expect a value between -1.0 and +0.999… and will issue an error is the value is outside that range. An ANDI expects an integer value to use as a mask and will take only the integer portion of the value. However there may be times the user needs to load a 32-bit fractional value into a register but there are no 32-bit load commands as the data field is limited to 16-bits in the instruction word. This can be accomplished by use of the ".l" and ".u" extensions to parameter values. As an example:

```
// load a 32-bit value into a register, coeff is a 32-bit fractional value, max32 is 0x7FFFFFFF
wrdld r3, (coeff*max32).u   // load the upper 16-bits into R3, clear lower 16-bits
ori r3 (coeff*max32).l // load the lower 16-bits so r3 now has the 32-bit fractional coefficient in it
```

Math Operations

ABS – Absolute value of a core register

Use: ABS CREG

ACC32 = |CREG|

Encoding: `0000 0000 000C CCCC 0000 0000 0000 0000`

C: Core register

Description: The absolute value of the core register. The result is placed in ACC32.

Example: ABS R0

CLRACC64 – Clear the 64-bit accumulator

Use: CLRACC64

ACC64 = 0

Encoding: `0000 0010 0000 0000 0000 0000 0000 0000`

Description: ACC64 is set to 0x00000000

Example: CLRACC64

ADDI – Add a 16bit signed integer to 32 bit core register, not saturated (modulo 2^{32})

Use: ADDI CREG, SImm16

ACC32 = CREG + {SSSS SSSS SSSS SSSS, SImm16}

Encoding: `0000 0100 000C CCCC SIII IIII IIII IIII`

C: Core register

SI: Signed Imm16

Description: SImm16 is LSB aligned, the sign bit is extended across bits 31:16 prior to the 32-bit add. The result is placed in ACC32 and is allowed to "roll over".

Example: ADDI R0, -1

ADD – Unsigned addition of two core registers, not saturated (modulo 2^{32})

Use: ADD CREGX, CREGY

ACC32 = CREGX + CREGY

Encoding: `0000 0110 000X XXXX 0000 0000 000Y YYYY`

X: Core register X

Y: Core register Y

Description: Two core registers are added together. The result is placed in ACC32 and is allowed to "roll over".

Example: ADD R2, R3

ADDS – Signed addition of two core registers, saturated

Use: ADDS CREGX, CREGY

ACC32 = CREGX + CREGY

Encoding: `0000 1000 000X XXXX 0000 0000 000Y YYYY`

X: Core register X
Y: Core register Y
Description: Two core registers are added together. The result is placed in ACC32 and is saturated to the maximum positive (0x7FFFFFFF) or negative (0x80000000) value as appropriate
Example: ADDS ACC32, R0

ADDSI – Signed addition of 32-bit S.31 core register with 16-bit S.15 (MSB aligned), saturated

Use: ADDS CREG, S.15
ACC32 = CREG + {S.15<<16, 0x0000]}
Encoding: `0000 1010 000C CCCC SFFF FFFF FFFF FFFF`
C: Core register
S: Sign bit
F: Fractional bits
Description: Prior to the 32-bit addition, the 16-bit S.15 immediate value is shifted left by 16 bits and zero padded, converting the S.15 immediate value into an S.31 value. . The result is placed in ACC32 and is saturated to the maximum positive (0x7FFFFFFF) or negative (0x80000000) S.31 signed fractional value.
Example ADDSI R0, 0.5

SUB – Unsigned subtraction of two core registers, not saturated (modulo 2^{32})

Use: SUB CREGX, CREGY
ACC32 = CREGX - CREGY
Encoding: `0000 1100 000X XXXX 0000 0000 000Y YYYY`
X: Core register
Y: Core register
Description: Unsigned subtraction of two core registers. The result is placed in ACC32 and is allowed to "roll over".

SUBS – Signed subtraction of two core registers, saturated

Use: SUBS CREGX, CREGY
ACC32 = CREGX - CREGY
Encoding: `0000 1110 000X XXXX 0000 0000 000Y YYYY`
X: Core register
Y: Core register
Description: Signed subtraction of core register Y from core register X. The result is placed in ACC32 and is saturated to the maximum positive (0x7FFFFFFF) or negative (0x80000000) value as appropriate
Example SUBS R0, R1

SL – Shift left logical using an immediate 5-bit value

Use: SL CREG, Imm5

ACC32 = CREG << Imm5, $ACC32_0$ <- 0

Encoding: `0001 0000 000C CCCC 0000 0000 000I IIII`

C: Core register

I: Imm5

Description: CREG is shifted left by the number of bit positions specified by the Imm5 value. Zeros are inserted into the bit positions emptied by the shift. The result is placed in ACC32.

Example: SL R0, 8

SLR – Shift left logical using the 5-LSBs of a core register

Use: SLR CREGX, CREGY

ACC32 = CREGX << $CREGY_{4:0}$, $ACC32_0$ <- 0

Encoding: `0001 0010 000X XXXX 0000 0000 000Y YYYY`

X: Core register

Y: Core register

Description: CREGX is shifted left by the number of bit positions specified by $CREGY_{4:0}$. Zeros are inserted into the bit positions emptied by the shift. The result is placed in ACC32.

Example: SLR R0, R1

SLS – Shift left arithmetic with saturation using an immediate 5-bit value

Use: SLS CREG, Imm5

ACC32 = CREG << Imm5, $ACC32_0$ <- 0 or saturate to max +/- value

Encoding: `0001 0100 000C CCCC 0000 0000 000I IIII`

C: Core register

I: Imm5

Description: CREG is shifted left by the number of bit positions specified by the Imm5 value. Zeros are inserted into the bit positions emptied by the shift. The result is placed in ACC32 and is saturated to the maximum positive (0x7FFFFFFF) or negative (0x80000000) value as appropriate

Example SLS R0, 8

SLSR – Shift left arithmetic with saturation using the 5-LSBs of a core register

Use: SLSR CREGX, CREGY

ACC32 = CREGX << $CREGY_{4:0}$, $ACC32_0$ <- 0 or saturate to max +/- value

Encoding: `0001 0110 000X XXXX 0000 0000 000Y YYYY`

X: Core register

Y: Core register

Description: CREGX is shifted left by the number of bit positions specified by $CREGY_{4:0}$. Zeros are inserted into the bit positions emptied by the shift. The result is placed in ACC32 and is saturated to the

maximum positive (0x7FFFFFFF) or negative (0x80000000) value as appropriate
Example SLSR R0, R1

SR – Shift right logical using an immediate 5-bit value

Use: SR CREG, Imm5
ACC32 = CREG >> Imm5, 0 -> $ACC32_{31}$
Encoding: `0001 1000 000C CCCC 0000 0000 000I IIII`
C: Core register
I: Imm5
Description: CREG is shifted right by the number of bit positions specified by the Imm5 value. Zeros are inserted into the bit positions emptied by the shift. The result is placed in ACC32
Example: SR R0, 8

SRR – Shift right logical using the 5-LSBs of a core register

Use: SRR CREGX, CREGY
ACC32 = CREGX >> $CREGY_{4:0}$, 0 -> $ACC32_{31}$
Encoding: `0001 1010 000X XXXX 0000 0000 000Y YYYY`
X: Core register
Y: Core register
Description: CREGX is shifted right by the number of bit positions specified by $CREGY_{4:0}$. Zeros are inserted into the bit positions emptied by the shift. The result is placed in ACC32
Example: SRR R0, R1

SRA – Shift right arithmetic using an immediate 5-bit value

Use: SRA CREG, Imm5
ACC32 = CREG >> Imm5, $CREG_{31}$ -> $ACC32_{31}$
Encoding: `0001 1100 000C CCCC 0000 0000 000I IIII`
C: Core register
I: Imm5
Description: CREG is shifted right by the number of bit positions specified by the Imm5 value. The sign bit of CREG ($CREG_{31}$) is inserted into the bit positions emptied by the shift. The result is placed in ACC32
Example: SRA R0, 8

SRAR – Shift right arithmetic using the 5-LSBs of a core register

Use: SRAR CREGX, CREGY
ACC32 = CREGX >> $CREGY_{4:0}$, $CREGX_{31}$ -> $ACC32_{31}$
Encoding: `0001 1110 000X XXXX 0000 0000 000Y YYYY`
X: Core register
Y: Core register

Description: CREGX is shifted right by the number of bit positions specified by $CREGY_{4:0}$. The sign bit of CREGX ($CREGX_{31}$) is inserted into the bit positions emptied by the shift. The result is placed in ACC32
Example: SRAR R0, R1

MACRR – Multiply and accumulate using 64-bit accumulator (S.63), 32-bit (S.31) x 32-bit (S.31) multiply with saturation

Use: MACRR CREGX, CREGY
ACC64 = ACC64 + CREGX * CREGY
Encoding: `0010 0000 000X XXXX 0000 0000 000Y YYYY`
X: Core register
Y: Core register
Description: CREGX is multiplied by CREGY and added to the 64-bit accumulator ACC64. ACC64 is saturated to the maximum positive (0x7FFFFFFFFFFFFFFF) or negative (0x8000000000000000) value as appropriate
Example: MACRR R0, R1

MACRI – Multiply and accumulate using 64-bit accumulator (S.63), 32-bit (S.31) x 16-bit (S.15) coefficient multiply

Use: MACRI CREG, S.15
ACC64 = ACC64 + CREG * {S.15, 0x0000}
Encoding: `0010 0010 000C CCCC SFFF FFFF FFFF FFFF`
C: Core register
S: Sign bit
F: Fractional bits
Description: CREG is multiplied by the S.15 coefficient zero padded to 32-bits and added to the 64-bit accumulator ACC64. ACC64 is saturated to the maximum positive (0x7FFFFFFFFFFFFFFF) or negative (0x8000000000000000) value as appropriate
Example: MACRI R0, -0.8

MACRD – Multiply and accumulate using 64-bit accumulator (S.63), 32-bit (S.31) x 16-bit (S.15) delay memory multiply

Use: MACRD CREG, ADDRESS
ACC64 = ACC64 + CREG * {[ADDRESS], 0X0000}
Encoding: `0010 0100 000C CCCC 0AAA AAAA AAAA AAAA`
C: Core register
A: Address
Description: CREG is multiplied by the S.15 data at ADDRESS zero padded to 32-bits and added to the 64-bit accumulator ACC64. ACC64 is saturated to the maximum positive (0x7FFFFFFFFFFFFFFF) or negative (0x8000000000000000) value as appropriate
Example: MACRD R0, 100

MACID – Multiply and accumulate using 64-bit accumulator (S.63), 8-bit (S.7) coefficient multiply x 16-bit (S.15) delay memory

Use: MACID S.7, ADDRESS
ACC64 = ACC64 + {S.7,0x000000} * {[ADDRESS], 0X0000}
Encoding: `0010 0110 SFFF FFFF 0AAA AAAA AAAA AAAA`
S: Sign bit
F: Fractional bits
A: Address
Description: The S.7 coefficient is zero padded to 32-bits and is multiplied by the S.15 data at ADDRESS zero padded to 32-bits then added to the 64-bit accumulator ACC64. ACC64 is saturated to the maximum positive (0x7FFFFFFFFFFFFFFF) or negative (0x8000000000000000) value as appropriate
Example: MACHID 0.5, 111

MACHRR – Multiply and accumulate using 64-bit accumulator (S3.60), 32-bit (S.31) x 32-bit (S.31 shifted to S3.28) multiply

Use: MACHRR CREGX, CREGY
ACC64 = ACC64 + CREGX * (CREGY >> 3)
Encoding: `0010 1000 000X XXXX 0000 0000 000Y YYYY`
X: Core register
Y: Core register
Description: CREGX is multiplied by CREGY which has been arithmetically shifted 3 bits to the right to create 3-bits of headroom for addition then added to the 64-bit accumulator ACC64. ACC64 is saturated to the maximum positive (0x7FFFFFFFFFFFFFFF) or negative (0x8000000000000000) value as appropriate
Example: MACHRR R0, ACC32

MACHRI – Multiply and accumulate using 64-bit accumulator (S3.60), 32-bit (S.31) x 16-bit (S.15 shifted right and zero appended to S3.28) multiply

Use: MACHRI CREG, S.15
ACC64 = ACC64 + CREG * {(S.15 >> 3), 0x0000}
Encoding: `0010 1010 000C CCCC SFFF FFFF FFFF FFFF`
C: Core register
S: Sign bit
F: Fractional bits
Description: CREGX is multiplied by the S.15 coefficient which has been arithmetically shifted 3 bits to the right to create 3-bits of headroom for addition and zero padded to 32-bits then added to the 64-bit accumulator ACC64. ACC64 is saturated to the maximum positive (0x7FFFFFFFFFFFFFFF) or negative (0x8000000000000000) value as appropriate
Example: MACHRI R0, -0.8

MACHRD – Multiply and accumulate using 64-bit accumulator (S3.60), 32-bit (S.31) x 16-bit (S.15 shifted right to S3.15 and zero appended to S3.28) delay memory

Use: MACHRD CREG, ADDRESS

ACC64 = ACC64 + CREG * {[ADDRESS]>>3, 0x0000}

Encoding: `0010 1100 000C CCCC 0AAA AAAA AAAA AAAA`

C: Core register

A: Address

Description: CREGX is multiplied by the S.15 data at the location specified by ADDRESS which is arithmetically shifted 3 bits to the right to create 3-bits of headroom for addition and zero padded to 32-bits then added to the 64-bit accumulator ACC64. ACC64 is saturated to the maximum positive (0x7FFFFFFFFFFFFFFF) or negative (0x8000000000000000) value as appropriate

Example: MACHRD R0, 100

MACHID – Multiply and accumulate using 64-bit accumulator (S3.60), 8-bit (S.7 zero appended to S.32) coefficient x 16-bit (S.15 shifted right to S3.15 and zero appended to S3.28) delay memory multiply

Use: MACID S.7, ADDRESS

ACC64 = ACC64 + {S.7, 0x000000} * {[ADDRESS]>>3, 0x0000}

Encoding: `0010 1110 SFFF FFFF 0AAA AAAA AAAA AAAA`

S: Sign bit

F: Fractional bits

A: Address

Description: The S.7 coefficient is zero padded to 32-bits and is multiplied by the S.15 data at the location specified by ADDRESS which is arithmetically shifted 3 bits to the right to create 3-bits of headroom for addition and zero padded to 32-bits then added to the 64-bit accumulator ACC64. ACC64 is saturated to the maximum positive (0x7FFFFFFFFFFFFFFF) or negative (0x8000000000000000) value as appropriate

Example: MACHID –0.8, 100

MULTRR – Saturated multiply of two core registers, 32-bit (S.31) x 32-bit (S.31), 32 MSBs of the 64-bit product in ACC32

Use: MULTRR CREGX, CREGY

ACC32 = $(\text{CREGX} * \text{CREGY})_{63:32}$

Encoding: `0011 0000 000X XXXX 0000 0000 000Y YYYY`

X: Core register

Y: Core register

Description: CREGX is multiplied by CREGY and the 32 MSBs are placed in ACC32. If both CREGX and CREGY are -1.0 the result will be saturated to the maximum positive (0x7FFFFFFF) value

Example: MULTRR R0, R1

MULTRI – Multiply to 32-bit accumulator (S.31), 32-bit (S.31) x 16-bit (S.15) multiply

Use: MULTRI CREG, S.15

ACC32 = (CREG * {S.15, 0x0000})$_{63:32}$

Encoding: `0011 0010 000C CCCC SFFF FFFF FFFF FFFF`

C: Core register

S: Sign bit

F: Fractional bits

Description: CREG is multiplied by the S.15 coefficient zero padded to 32-bits and the 32 MSBs are placed in ACC32. If both CREGX and the S.15 coefficient are -1.0 the result will be saturated to the maximum positive (0x7FFFFFFF) value

Example: MULTRI R0, -0.9

NEG – Negate a core register, 2's complement saturated

Use: NEG CREG

ACC32 = -CREG

Encoding: `0011 0100 000C CCCC 0000 0000 0000 0000`

C: Core register

Description: The 2's complement of CREG is placed in ACC32. If CREGNE is -1.0 the result will be saturated to the maximum positive (0x7FFFFFFF) value

Example: NEG R0

LOG2 – Calculate the log base 2 of the absolute value of a register

Use: LOG2 CREG

ACC32 = LOG_2 (|CREG|)

Encoding: `0011 0110 000C CCCC 0000 0000 0000 0000`

C: Core register

Description: The $\log_2$ of the absolute value in CREG is placed in ACC32, the result is in an S5.26 format where S is the sign followed by the 5 bit integer and 26 bit fraction.

Example: LOG2 R0

EXP2 – Calculate 2^ value of a register

Use: EXP2 CREG

ACC32 = 2^{CREG}

Encoding: `0011 1000 000C CCCC 0000 0000 0000 0000`

C: Core register

Description: 2 is raised to the power in CREG then placed in ACC32, the number format in the CREG must be S5.26 and S must be 1 indicating a negative number.

Example: EXP2 R0

Copy Operations

CPY_CC – Copy from one core register to another

Use: CPY_CC CREGNAX, CREGY

CREGNAX = CREGY

Encoding: 0110 0000 000X XXXX 0000 0000 000Y YYYY

X: Core register

Y: Core register

Description: CREGY is copied to CREGNAX

Example: CPY_CC R0, FLAGS

CPY_CM – Copy from MREG register to core register

Use: CPY_CM CREGNA, MREG

CREGNA = MREG

Encoding: 0110 0010 000C CCCC 0000 0000 0MMM MMMM

C: Core register

M: MREG register

Description: MREG is copied to CREGNA

Example: CPY_CM R0, MR64

CPY_CS – Copy from special function register (SFR) to core register

Use: CPY_CS CREGNA, SFR

CREGNA = SFR

Encoding: 0110 0100 000C CCCC 0000 0000 00SS SSSS

C: Core register

S: SFR register

Description: SFR is copied to CREGNA. Note that some SFRs are 16 bit and may be MSB or LSB aligned or may be write only and not readable (i.e. OUT0) please see the “Special Function Registers” section for further information.

Example: CPY_CS R0, IN0

CPY_MC – Copy from core register to MREG register

Use: CPY_MC MREG, CREG

MREG = CREG

Encoding: 0110 0110 000C CCCC 0000 0000 0MMM MMMM

C: Core register

M: MREG register

Description: CREG is copied to MREG

Example: CPY_MC MR17, FLAGS

CPY_SC – Copy from core register to special function register (SFR)

Use: CPY_SC SFR, CREG
SFR = CREG
Encoding: `0110 1000 000C CCCC 0000 0000 00SS SSSS`
C: Core register
S: SFR register
Description: CREG is copied to SFR. Note that some SFRs are not 32-bit and may be MSB or LSB aligned or may be read only and not writeable (i.e. IN0) please see the "Special Function Registers" section for further information.
Example: CPY_SC OUT0, R0

CPY_CMX – Copy from MREG register to core register using a second core register as the index register to address the MREG

Use: CPY_CMX CREGNAX, CREGY
CREGNAX = MREG(@$CREGY_{6:0}$)
Encoding: `0110 1010 000X XXXX 0000 0000 000Y YYYY`
X: Core register
Y: Core register
Description: $CREGY_{6:0}$ are used to address which MREG register to read. This allows the MREG registers to be used as a look up table if they were initialized to values using .MREG directives
Example: CPY_CMX R0, R1

Load/store Operations

RDACC64U – Copy the upper 32-bits from the 64-bit accumulator to a core register

Use: RDACC64U CREGNA

CREGNA = $ACC64_{63:32}$

Encoding: 1000 0000 000C CCCC 0000 0000 0000 0000

C: Core register

Description: $ACC64_{63:32}$ is copied to CREGNA

Example: RDACC64U R0

RDACC64L – Copy the lower 32-bits from the 64-bit accumulator to a core register

Use: RDACC64L CREGNA

CREGNA = $ACC64_{31:0}$

Encoding: 1000 0010 000C CCCC 0000 0000 0000 0000

C: Core register

Description: $ACC64_{31:0}$ is copied to CREGNA

Example: RDACC64L R0

LDACC64U – Load the upper 32-bits of the 64-bit accumulator from a core register

Use: LDACC64U CREG

$ACC64_{63:32}$ = CREG

Encoding: 1000 0100 000C CCCC 0000 0000 0000 0000

C: Core register

Description: CREG is copied to $ACC64_{63:32}$

Example: LDACC64U R0

LDACC64L – Load the lower 32-bits of the 64-bit accumulator from a core register

Use: LDACC64L CREG

$ACC64_{31:0}$ = CREG

Encoding: 1000 0110 000C CCCC 0000 0000 0000 0000

C: Core register

Description: CREG is copied to $ACC64_{31:0}$

Example: LDACC64L R0

RDDEL – Read from delay memory into a core register using an immediate address

Use: RDDEL CREGNA, ADDRESS

CREGNA = {[Address], 0x0000}

Encoding: 1000 1000 000C CCCC 0AAA AAAA AAAA AAAA

C: Core register

A: Address

Description: The data at ADDRESS is zero appended and placed in CREGNA
Example: RDDEL R0, 200

WRDEL – Write to delay memory from a core register using an immediate address

Use: WRDEL ADDRESS, CREG
[ADDRESS] = $CREG_{31:16}$
Encoding: `1000 1010 000C CCCC 0AAA AAAA AAAA AAAA`
C: Core register
A: Address
Description: $CREG_{31:16}$ is written to ADDRESS.
Example: WRDEL R0, 200

RDDELX – Read from delay memory into a core register using a second core register as the index register to address the delay memory

Use: RDDELX CREGNAX, CREGY
CREGNAX = {[@$CREGY_{14:0}$], 0x0000}
Encoding: `1000 1100 000X XXXX 0000 0000 000Y YYYY`
X: Core register
Y: Core register
Description: $CREGY_{14:0}$ are used as the address into delay memory. The data at this address is zero appended and placed in CREGNAX
Example: RDDELX R0, ACC32

WRDELX – Write the value in a core register to delay memory using a second core register as the index register to address the delay memory

Use: WRDELX CREGX, CREGY
@$CREGX_{14:0}$ = $CREGY_{31:16}$
Encoding: `1000 1110 000X XXXX 0000 0000 000Y YYYY`
X: Core register
Y: Core register
Description: $CREGX_{14:0}$ are used as the address into delay memory. $CREGY_{31:16}$ are written to this address.
Example: WRDELX R0, ACC32

RDDIRX – Read from delay memory into a core register using a second core register as the index register to address the delay memory without adding address counter

Use: RDDIRX CREGNAX, CREGY
CREGNAX = { [@@$CREGY_{14:0}$], 0x0000}
Encoding: `1001 0000 000X XXXX 0000 0000 000Y YYYY`
X: Core register

Y: Core register
Description: $CREGY_{14:0}$ are used as the absolute address into delay memory, the address counter is not added to this address. The data at this address is zero appended and placed in CREGNAX
Example: RDDIRX R0, ACC32

WRDIRX – Write the value in a core register to delay memory using a second core register as the index register to address the delay memory without adding address counter

Use: WRDIRX CREGX, CREGY
@@$CREGX_{14:0}$ = $CREGY_{31:16}$
Encoding: `1001 0010 000X XXXX 0000 0000 000Y YYYY`
X: Core register
Y: Core register
Description: $CREGX_{14:0}$ are used as the address into delay memory, the address counter is not added to this address. $CREGY_{31:16}$ are written to this address.
Example: WRDIRX R0, ACC32

SAT64 – Copy the 32 MSBs from ACC64 (S3.60), shift left 3 bits and saturate to an S.31 format and place into core register

Use: SAT64 CREGNA
CREGNA = $ACC64_{63:32}$ << 3 with saturation
Encoding: `1001 0100 000C CCCC 0000 0000 0000 0000`
C: Core register
Description: $ACC64_{63:32}$ are shifted left 3 bit to remove the headroom added by instructions like MACHRR. The result is placed in CREGNA and is saturated to the maximum positive (0x7FFFFFFF) or negative (0x80000000) value as appropriate
Example: SAT64 R0

WRDLD – Load a 16-bit immediate value to the upper 16-bits of a core register, 0s to LSBs

Use: WRDLD CREGNA, Imm16
CREGNA = {Imm16, 0x0000}
Encoding: `1001 0110 000C CCCC IIII IIII IIII IIII`
C: Core register
I: 16-bit immediate value
Description: The 16-bit immediate value is 0 appended and placed in CREGNA. Note that a WRDLD instruction immediately followed by an ORI instruction can be used to load a 32-bit immediate value into ACC32
Example: WRDLD 0x4711

Logic Operations

INV – Invert (1's comp) core register

Use: INV CREG

ACC32 = ~CREG

Encoding: `1010 0000 000C CCCC 0000 0000 0000 0000`

C: Core register

Description: Every bit in CREG is inverted (1's comp) and the result is placed in ACC32

Example: INV R0

OR – Bitwise OR of 2 core registers to ACC32

Use: OR CREGX, CREGY

ACC32 = CREGX | CREGY

Encoding: `1010 0010 000X XXXX 0000 0000 000Y YYYY`

X: Core register

Y: Core register

Description: CREGX and CREGY are ORed together and the result is placed in ACC32

Example: OR R0, R1

ORI – Bitwise OR of core register with a 16-bit immediate value 0 extended to ACC32

Use: ORI CREG, Imm16

ACC32 = CREG | {0x0000, Imm16}

Encoding: `1010 0100 000C CCCC IIII IIII IIII IIII`

C: Source/destination core register

I: 16-bit immediate value

Description: The Imm16 value is 0 extended and ORed with CREG and the result is placed in ACC32. Note that a WRDLD instruction immediately followed by an ORI instruction can be used to load a 32-bit immediate value into ACC32

Example: ORI R0, 0x4711

AND – Bitwise AND of 2 core registers to ACC32

Use: AND CREGX, CREGY

ACC32 = CREGX & CREGY

Encoding: `1010 0110 000X XXXX 0000 0000 000Y YYYY`

X: Core register

Y: Core register

Description: CREGX and CREGY are ANDed together and the result is placed in ACC32

Example: AND R0, R1

ANDI – Bitwise AND of core register with a 16-bit immediate value 0 extended to ACC32

Use: ANDI CREG, Imm16

ACC32 = REGD & {0x0000, Imm16}

Encoding: `1010 1000 000C CCCC IIII IIII IIII IIII`

C: Core register

I: 16-bit immediate value

Description: The Imm16 value is 0 extended and ANDed with CREG and the result is placed in ACC32.

Example: ANDI FLAGS, 0x0100

XOR – Bitwise XOR of 2 core registers to ACC32

Use: XOR CREGX, CREGY

ACC32 = CREGX ⊕ CREGY

Encoding: `1010 1010 000X XXXX 0000 0000 000Y YYYY`

X: Core register

Y: Core register

Description: CREGX and CREGY are XORed together and the result is placed in ACC32

Example: XOR R0, ACC32

XORI – Bitwise XOR of core register with a 16-bit immediate value 0 extended to ACC32

Use: XORI CREG, Imm16

ACC32 = CREG ⊕ [0x0000, Imm16]

Encoding: `1010 1100 000C CCCC IIII IIII IIII IIII`

C: Core register

I: 16-bit immediate value

Description: The Imm16 value is 0 extended and XORed with CREG and the result is placed in ACC32.

Example: ANDI FLAGS, 0x0100

JGEZ – Jump if core register value is >= 0

Use: JGEZ CREG, OFFSET

If (CREG >= 0) PC = PC + OFFSET

Encoding: `1010 1110 000C CCCC 0000 oooo oooo oooo`

C: Core register

o: Offset added to address counter, positive only

Description: If CREG is greater than or equal to zero then the offset is added to the current program counter (PC). OFFSET is always positive no back jumping allowed. Current PC + OFFSET must not exceed the address of the last instruction within the FXCore program.

Example: JGEZ R0, Vader

JNEG – Jump if core register value is < 0

Use: JNEG CREG, OFFSET
If (CREG < 0) PC = PC + OFFSET
Encoding: `1011 0000 000C CCCC 0000 oooo oooo oooo`
C: Core register
o: Offset added to address counter, positive only
Description: If CREG is less than zero then the offset is added to the current program counter (PC). OFFSET is always positive no back jumping allowed. Current PC + OFFSET must not exceed the address of the last instruction within the FXCore program.
Example: JNEG R0, Yoda

JNZ – Jump if core register value is != 0

Use: JNZ CREG, OFFSET
If (CREG!= 0) PC = PC + OFFSET
Encoding: `1011 0010 000C CCCC 0000 oooo oooo oooo`
C: Core register
o: Offset added to address counter, positive only
Description: If CREG is not equal to zero then the offset is added to the current program counter (PC). OFFSET is always positive no back jumping allowed. Current PC + OFFSET must not exceed the address of the last instruction within the FXCore program.
Example: JNZ R0, Luke

JZ – Jump if core register value = 0

Use: JZ CREG, OFFSET
If (CREG == 0) PC = PC + OFFSET
Encoding: `1011 0100 000C CCCC 0000 oooo oooo oooo`
C: Core register
o: Offset added to address counter, always positive no back jumping allowed
Description: If CREG is equal to zero then the offset is added to the current program counter (PC). OFFSET is always positive no back jumping allowed. Current PC + OFFSET must not exceed the address of the last instruction within the FXCore program.
Example: JZ R0, Leia

JZC – Jump if core register value is different sign from acc32

Use: JZC CREG, OFFSET
If (SGN(CREG) != SGN(ACC32)) PC = PC + OFFSET
Encoding: `1011 0110 000C CCCC 0000 oooo oooo oooo`
C: Core register
o: Offset added to address counter, positive only
Description: If the sign of CREG is not equal to the sign of ACC32 then the offset is added to the current

program counter (PC). OFFSET is always positive no back jumping allowed. Current PC + OFFSET must not exceed the address of the last instruction within the FXCore program.
Example: JZ R0, Chewbacca

JMP – Jump always

Use: JMP OFFSET
PC = PC + OFFSET
Encoding: 1011 1000 0000 0000 0000 oooo oooo oooo
o: Offset added to address counter, positive only
Description: OFFSET is added to the current program counter (PC). OFFSET is always positive no back jumping allowed. Current PC + OFFSET must not exceed the address of the last instruction within the FXCore program.
Example: JMP, ObiWan

Extended Operations

APA – First instruction for all-pass filter with fixed values

Use: APA S.7, ADDRESS

ACC32 = [ADDRESS]*S.7 +ACC32, R15=[ADDRESS]

Encoding: 1100 0000 SFFF FFFF 0AAA AAAA AAAA AAAA

S: Sign Bit

F: Fractional bits

A: Address of tail of all-pass delay block

Description: ACC32 holds the input to the all-pass and will be over written, R15 (PARAM0) will be overwritten with the tail of the all-pass delay block. APA and APB should be used as a pair to create the all-pass. Note that the coefficient in APA should be the inverse (2's comp) of the coefficient in APB, i.e. if APB is a positive coefficient then APA should be negative

Example: APA -0.4, 200

APB – Second instruction for all-pass filter with fixed values

Use: APB S.7, ADDRESS

[ADDRESS] = ACC32, ACC32 = (ACC32 * S.7) + R15

Encoding: 1100 0010 SFFF FFFF 0AAA AAAA AAAA AAAA

S: Sign Bit

F: Fractional bits

A: Address of head of all-pass delay block

Description: R15 (PARAM0) must have the tail of the all-pass delay block from the preceding APA, ACC32 will hold the result of the all-pass. APA and APB should be used as a pair to create the all-pass. Note that the coefficient in APB should be the inverse (2's comp) of the coefficient in APA, i.e. if APA is a negative coefficient then APB should be positive

Example: APB 0.4, 0

APRA – First instruction for all-pass filter using register for coefficient value

Use: APRA CREGN, ADDRESS

ACC32 = [ADDRESS]*-CREGN + ACC32, R15=[ADDRESS]

Encoding: 1100 0100 000C CCCC 0AAA AAAA AAAA AAAA

C: Core register

A: Address of tail of all-pass delay block

Description: ACC32 holds the input to the all-pass and will be over written, R15 (PARAM0) will be overwritten with the tail of the all-pass delay block. APRA and APRB should be used as a pair to create the all-pass. Note that APRA will multiply the memory value read from the tail of the all-pass delay block with the 2's complement of the all-pass coefficient within CREGN

Example: APRA R0, 200

APRB – Second instruction for all-pass filter using register for coefficient value

Use: APRB CREGN, ADDRESS

[ADDRESS] = ACC32, ACC32 = (ACC32* CREGN) + R15

Encoding: 1100 0110 000C CCCC 0AAA AAAA AAAA AAAA

C: Core register

A: Address of head of all-pass delay block

Description: R15 (PARAM0) must have the tail of the all-pass delay block from the preceding APRA, ACC32 will hold the result of the all-pass. APRA and APRB should be used as a pair to create the all-pass

Example: APRB R0, 0

APRRA – First instruction for all-pass filter using register for both values

Use: APRRA CREGNX, CREGNY

ACC32 = @$CREGNY_{14:0}$*-CREGNX + ACC32, R15=[$CREGNY_{14:0}$]

Encoding: 1100 1000 000X XXXX 0000 0000 000Y YYYY

X: Register that holds the all-pass coefficient

Y: Register that holds the address of tail of all-pass delay block

Description: ACC32 holds the input and will be over written , R15 (PARAM0) will be overwritten with the tail of the all-pass delay block. APRRA and APRRB should be used as a pair to create the all-pass. Note that APRRA will multiply the memory value read from the tail of the all-pass delay block with the 2's complement of the all-pass coefficient within CREGNX

Example: APRRA R0, R1

APRRB – Second instruction for all-pass filter using register for both values

Use: APRRB CREGNX, CREGNY

@$CREGNY_{14:0}$ = ACC32, ACC32 = (ACC32* CREGNX) + R15

Encoding: 1100 1010 000X XXXX 0000 0000 000Y YYYY

X: Register that holds the all-pass coefficient

Y: Register that holds address of head of all-pass delay block

Description: R15 (PARAM0) should have the tail of the all-pass delay block from APRRA, ACC32 will hold the result of the all-pass. APRRA and APRRB should be used as a pair to create the all-pass

Example: APRRB R0, R1

APMA – First instruction for all-pass filter using MREG for delay

Use: APMA CREGN, MREG

ACC32 = MREG*-CREGN + ACC32, R15=MREG

Encoding: 1100 1100 000C CCCC 0000 0000 0MMM MMMM

C: Register that holds the all-pass coefficient

M: MREG that holds single delay

Description: APMA is part of a pair of instructions to perform an all-pass on a single delay element located in an MREG, this is useful for a phaser or other single delay AP. ACC32 holds the input and will

be over written, R15 (PARAM0) will be overwritten with value in the MREG. APMA and APMB should be used as a pair to create the all-pass. Note that APMA will multiply the value read from the all-pass delay register MREG with the 2's complement of the all-pass coefficient within CREGN
Example: APMA R0, MR10

APMB – Second instruction for all-pass filter using MREG for delay

Use: APMB CREGN, MREG
MREG = ACC32, ACC32 = (ACC32* CREGN) + R15
Encoding: `1100 1110 000C CCCC 0000 0000 0MMM MMMM`
C: Register that holds the all-pass coefficient
M: MREG that holds single delay
Description: R15 (PARAM0) should have value from MREG from APMA, ACC32 will hold the result of the all-pass. APMA and APMB should be used as a pair to create the all-pass
Example: APMB R0, MR10

CHR – Chorus on delay mem

Use: CHR LFO|W|N, ADDRESS
$R15_{30:16}$ = Depth in sample set by user, $R15_{31}$ must be 0, ACC32 = Chorus result
Encoding: `1101 0000 0000 NLLW 0AAA AAAA AAAA AAAA`
N: 0 – use positive SIN or COS, 1 – use negative SIN or COS
LFO: LFO to use
W: SIN (0) or COS (1)
A: Address of head of delay block
Description: This instruction will execute a chorus on the delay block starting at ADDRESS using the LFO and phase as defined in the instruction. $R15_{30:16}$ (PARAM0) must contain the maximum depth in number of samples that the chorus will go into the delay and must be less than the length of the delay, $R15_{31}$ must be 0.
Example: CHR LFO0|SIN, ChorusDelay

PITCH – Pitch shift on delay mem

Use: PITCH RAMP|LENGTH|XFADE, ADDRESS
ACC32 = Pitch result
Encoding: `1101 0010 00XX LL0R 0AAA AAAA AAAA AAAA`
X: Crossfade shape to use
L: Block length, 512 (00), 1024 (01), 2048 (10) or 4096 (11)
R: Ramp to use.
A: Address of head of delay block
Description: This instruction will execute a pitch shift through a defined block in delay memory. Please see the application note "Pitch Shifting in FXCore" for details.
Example: PITCH RMP0|L2048|XF0, Pdelay

SET – Set a user bit high or low using the selected bit of a register

Use: SET USERBIT|N, CREG

USERBIT = $CREG_N$

Encoding: 1101 0100 000C CCCC 0000 0000 00UN NNNN

C: Core register

U: User bit number

N: Bit number within a core register

Description: Bit N of CREG will be written to either USER0 or USER1 pin as determined by USERBIT

Example: SET USER0|15, R0

INTERP – Do a linear interpolation between two samples in a delay line

Use: INTERP CREG, ADDRESS

ACC32 = ([@$CREG_{30:16}$ + ADDRESS + 1] - [@$CREG_{30:16}$ + ADDRESS]) * ($CREG_{15:0}$ << 15) + [@$CREG_{30:16}$ + ADDRESS]

Encoding: 1101 0110 000C CCCC 0AAA AAAA AAAA AAAA

C: Core register

A: Base address

Description: Linear interpolation using ($CREG_{30:16}$ + ADDRESS) as the address of the first sample, ($CREG_{30:16}$ + ADDRESS + 1) as the address of the second sample and $CREG_{15:0}$ as the interpolation coefficient. $CREG_{15:0}$ is treated as an unsigned fractional number and a sign bit of “0” is prepended to it. Please see the application note “Using INTERP in FXCore” for details.

Example: INTERP R0, 200

Reserved Words

The follow are the words reserved in the assembler.

Core Registers

R0 through R15 – 32-bit general purpose register
ACC32 – 32-bit accumulator, not settable by .creg, cleared to 0 on program change
FLAGS – Flag register, read only, may not be used as a destination register

Non-core CPU Registers

ACC64 – 64-bit accumulator S.63 or S3.60 format, only available to 64-bit MAC instructions, not cleared on program change

SRAM Based Registers

MR0 through MR127 – 32-bit SRAM based registers

Special Function Registers

Write values less than 32-bits in size are LSB aligned. “Settable” indicates if the value of the register can be set in the program header using a “.SREG” directive

Name	Address	Description	Bits	Read/Write	Settable
IN0	0	ADC input 0	32	R	N
IN1	1	ADC input 1	32	R	N
IN2	2	ADC input 2	32	R	N
IN3	3	ADC input 3	32	R	N
OUT0	4	DAC output 0	32	W	N
OUT1	5	DAC output 1	32	W	N
OUT2	6	DAC output 2	32	W	N
OUT3	7	DAC output 3	32	W	N
PIN	8	Raw user input values	16 LSBs	R	N
SWITCH	9	Debounced user inputs	16 LSBs	R	N
POT0_K	10	POT0 smoothing coefficient	5 LSBs	R/W	Y
POT1_K	11	POT1 smoothing coefficient	5 LSBs	R/W	Y
POT2_K	12	POT2 smoothing coefficient	5 LSBs	R/W	Y
POT3_K	13	POT3 smoothing coefficient	5 LSBs	R/W	Y
POT4_K	14	POT4 smoothing coefficient	5 LSBs	R/W	Y
POT5_K	15	POT5 smoothing coefficient	5 LSBs	R/W	Y
POT0	16	Raw POT0 value, S.12 format S always 0	13 MSBs	R	N
POT1	17	Raw POT1 value, S.12 format S always 0	13 MSBs	R	N
POT2	18	Raw POT2 value, S.12 format S always 0	13 MSBs	R	N
POT3	19	Raw POT3 value, S.12 format S always 0	13 MSBs	R	N
POT4	20	Raw POT4 value, S.12 format S always 0	13 MSBs	R	N
POT5	21	Raw POT5 value, S.12 format S always 0	13 MSBs	R	N
POT0_SMTH	22	Smoothed POT0 value, S.31 S always 0	32	R	N
POT1_SMTH	23	Smoothed POT1 value, S.31 S always 0	32	R	N
POT2_SMTH	24	Smoothed POT2 value, S.31 S always 0	32	R	N
POT3_SMTH	25	Smoothed POT3 value, S.31 S always 0	32	R	N
POT4_SMTH	26	Smoothed POT4 value, S.31 S always 0	32	R	N
POT5_SMTH	27	Smoothed POT5 value, S.31 S always 0	32	R	N

LFO0_F	28	LFO0 frequency coefficient	32	R/W	Y
LFO1_F	29	LFO1 frequency coefficient	32	R/W	Y
LFO2_F	30	LFO2 frequency coefficient	32	R/W	Y
LFO3_F	31	LFO3 frequency coefficient	32	R/W	Y
RAMP0_F	32	RAMP0 frequency coefficient	32	R/W	Y
RAMP1_F	33	RAMP1 frequency coefficient	32	R/W	Y
LFO0_S	34	LFO0 SINE output	32	R	N
LFO0_C	35	LFO0 COSINE output	32	R	N
LFO1_S	36	LFO1 SINE output	32	R	N
LFO1_C	37	LFO1 COSINE output	32	R	N
LFO2_S	38	LFO2 SINE output	32	R	N
LFO2_C	39	LFO2 COSINE output	32	R	N
LFO3_S	40	LFO3 SINE output	32	R	N
LFO3_C	41	LFO3 COSINE output	32	R	N
RAMP0_R	42	RAMP0 output	32	R	N
RAMP1_R	43	RAMP1 output	32	R	N
MAXTEMPO	44	Maximum number of samples allowed between two tap button presses	32	R/W	Y
TAPTEMPO	45	Latest measured tap tempo time	32	R	Y
SAMPLECNT	46	Number of sample periods, reset to 0 on program change. 32-bit Unsigned counter, rolls over.	32	R	N
NOISE	47	Random number/noise	32	R	N
BOOTSTAT	48	See table in datasheet for bits in this word	32	R	N
TAPSTKRLD[1]	--	Sets the number of samples that the tap button must be pressed for the state to be "sticky"	16	X	Y
TAPDBRLD[1]	--	Tap button debounce time in samples	16	X	Y
SWDBRLD[1]	--	User switch input debounce time in samples	16	X	Y
PRGDBRLD[1]	--	Program selection switch debounce time in samples	16	X	Y
OFLRLD[1]	--	Overflow LED on time in samples	16	X	Y

[1]TAPSTKRLD, TAPDBRLD, SWDBRLD, PRGDBRLD and OFLRLD are configuration registers and can only be preset in the program header, they cannot be read or written by user code.

Other Reserved Words and their value

NOTE: These words and values are used to build values for instructions like CHR or a mask value to use against the FLAGS or BOOTSTAT registers

Name	Value
LFO0	0x0000
LFO1	0x0002
LFO2	0x0004
LFO3	0x0006

SIN	0x00
COS	0x01
POS	0x00
NEG	0x08
RMP0	0x00
RMP1	0x01
L512	0x00
L1024	0X04
L2048	0x08
L4096	0x0C
XF0	0x00
XF1	0x10
XF2	0x20
XF3	0x30
USER0	0x00
USER1	0x20
OUT3OFLO	0x8000
OUT2OFLO	0x4000
OUT1OFLO	0x2000
OUT0OFLO	0x1000
IN3OFLO	0x0800
IN2OFLO	0x0400
IN1OFLO	0x0200
IN0OFLO	0x0100
TB2NTB1	0x0020
TAPSTKY	0x0010
NEWTT	0x0008
TAPRE	0x0004
TAPPE	0x0002
SW0	0x0001
SW1	0x0002
SW2	0x0004
SW3	0x0008
SW4	0x0010
SW0RE	0x0020
SW1RE	0x0040
SW2RE	0x0080
SW3RE	0x0100
SW4RE	0x0200
SW0PE	0x0400
SW1PE	0x0800
SW2PE	0x1000
SW3PE	0x2000
SW4PE	0x4000
TAP	0x0040
PLLRANGE0	0x0001

PLLRANGE1	0x0002
MNS	0x0004
I2CA0	0x0008
I2CA1	0x0010
I2CA2	0x0020
I2CA3	0x0040
I2CA4	0x0080
I2CA5	0x0100
I2CA6	0x0200
PR0	0x0001
PR1	0x0002
PR2	0x0004
PR3	0x0008
PR4	0x0010
PR5	0x0020
PR6	0x0040
PR7	0x0080
PR8	0x0100
PR9	0x0200
PR10	0x0400
PR11	0x0800
PR12	0x1000
PR13	0x2000
PR14	0x4000
PR15	0x8000

Assembler Directives

.EQU – Equate a name to a value

Use: .EQU NAME VALUE

When LABEL is used in an instruction it is replaced with VALUE by the assembler. VALUE may be another NAME, a numeric value or a simple arithmetic operation. Legal arithmetic operations are +, -, *, /, (,) and ^ (raise to a power)

Examples:

```
.equ fs 48000; define sample rate
.equ t 1/fs
.equ pi 3.141592654 ; value of pi
.equ e 2.718281828 ; value of e
.equ freq 500 ; desired corner frequency
// e^(-2*PI*F*t) to calculate coeff for a single pole IIR
.equ coeff e^(-2*pi*freq*t); note negative sign inside parenthesis
```

.RN – Rename a register

Use: .RN ALTNAME REGISTER

Allows a user to rename a register to give it a more descriptive name in their program. A register may only have one alternate name in a given program.

Examples:

```
.rn hp_filt R0 // R0 can now be referenced as hp_filt in user code
```

.MEM – Declare a memory block

Use: .MEM NAME SIZE

Declares a delay memory block called NAME of size SIZE+1

Examples:

```
.mem delay0 1000
.equ fs 48000
.mem delay1 fs/4; can use values defined in .equ statements and same math operations
```

.CREG – Set a core register to an initial value when program is loaded

Use: .CREG CORE_REGISTER VALUE

Set an initial value for a core register prior to the first iteration of a program. An optional ".i" may be added to the directive to indicate the value is to be treated as an integer (this will also truncate any fractional portion) rather than a fractional value.

Examples:

```
.creg r1 ; will error as 1.0 exceeds maximum S.31 format
.creg hp_filt coeff; can use aliased name and value from .equ
.creg r2 -1.0 ;
.creg.i r3 pi ; handy to calculate a counter value and use only the integer portion
```

.MREG – Set a memory register to an initial value when program is loaded

Use: .MREG MEM_REGISTER VALUE

Set an initial value for a memory register prior to the first iteration of a program. An optional ".i" may be added to the directive to indicate the value is to be treated as an integer (this will also truncate any fractional portion) rather than a fractional value.

Examples:

```
.mreg mr1 0.005; will preset MR1 to 0.005 prior to first execution of the program
.mreg hp_filt coeff; can use aliased name and value from .equ
.mreg mr2 -1.0;
```

.SREG – Set a special function register to an initial value when program is loaded

Use: .SREG SPECIAL_FUNCTION_REGISTER VALUE

Set an initial value for a memory register prior to the first iteration of a program. An optional ".i" may be added to the directive to indicate the value is to be treated as an integer (this will also truncate any fractional portion) rather than a fractional value.

Examples:

```
.sreg lfo0freq 10;
.sreg chorus_lfo_freq chorus_speed; can use aliased names and values from .equ
```

.L – Use the lower 16-bits of a 32-bit word in an instruction (appended to value)

Use: ORI r0, 0x12345678.l

Forces the assembler to use only the lower 16-bits of a 32-bit value in an instruction. This is very useful for loading a 32-bit value into a register.

Examples:

```
wrdld r0, 0x12345678.u     ; loads the upper 16-bits of 0x12345678 into the upper 16-bits of r0
ori r0, 0x12345678.l       ; loads the lower 16-bits  of 0x12345678 into the lower 16-bits of r0 so r0
                           ; now contains 0x12345678
```

.U – Use the upper 16-bits of a 32-bit word in an instruction (appended to value)

Use: WRDLD r0, 0x12345678.u

Forces the assembler to use only the upper 16-bits of a 32-bit value in an instruction. This is very useful for loading a 32-bit value into a register.

Examples:

```
wrdld r0, 0x12345678.u     ; loads the upper 16-bits of 0x12345678 into the upper 16-bits of r0
ori r0, 0x12345678.l       ; loads the lower 16-bits  of 0x12345678 into the lower 16-bits of r0 so r0
                           ; now contains 0x12345678
```

Tap Tempo Operation

The FX Core includes an integrated tap tempo that will monitor button pushes, handle de-bounce and count the number of sample periods between button pushes. It also includes extended functionality that can allow the programmer to provide additional features in their product.

In normal operation a user would tap the button twice to indicate the desired tap tempo time in sample periods. The program has access to the TAPTEMPO register (read only) where the program can read the number of sample periods between taps. In addition the NEWTT flag is set in the FLAGS register for one sample period so the program can check if the value has been updated.

If the user waits too long after the first tap then the tap tempo unit will reset and the next tap will be considered the first tap. This timeout is set by the MAXTEMPO register and is set in samples.

If a user presses and holds the tap button longer than the number of samples set in the TAPSTKRLD register the TAPSTKY bit in the FLAGS register will be set and remain set for as long as the user presses the button.

If the user presses and holds the button in excess of TAPSTKRLD on the first press then upon release the tap tempo unit will reset and the next tap will be considered the first tap of a new tap set. This allows a program to switch between options based on the user pressing and holding the tap tempo button on the first tap.

If a user presses and holds the tap tempo button on the second tap the TAPTEMPO count will be updated per normal operation, the NEWTT flag will be set and the TAPSTKY bit will be set after the timeout and will remain set as long as the user presses the button. This could be used to adjust a parameter in the program after tapping in a new value, i.e. the program could divide the tap value by 3 to do a triplet delay.

The TB2nTB1 bit in the FLAGS register indicates if it is the first tap (TB2nTB1 = 0) or second tap (TB2nTB1 = 1)

There are 3 additional flags related to the tap tempo:
TAPPE : Set upon detection of the tap tempo button being pushed
TAPRE : Set upon detection of the tap tempo button being released
TAPDB : The de-bounced level of the TAP input pin

There is an additional SFR named TAPDBRLD that the user can set to the desired debounce time in samples.

Memory Management

Every sample period a counter is decremented by 1 and the result is added to the addresses in the instructions to calculate the physical address into the delay memory. By doing this we cause the memory to act as a circular buffer where we write to a lower address and read from a higher address.

.MEM statements are used to define blocks of memory for use in your code.

Example:
.MEM delaya 1000

This would allocate a block of 1001 memory locations. It is 1 greater than the defined size to allow both a read and write pointer into the block. When you define a block in this manner a total of 3 symbols are generated in the assembler, in this example they would be:

delaya : Points to the head (write) address of the block
delaya# : Points to the tail (read) address of the block
delaya! : Is set to the length of the block (1000)

Example:

```
//Assume core register r0 has the data you want to write to the start of a delay then you would code
//the write as
wrdel r0, delaya // write the value in r0 to start of the delay line
// Reading is similar
rddel acc32, delaya# // read from the tail of the delay and put value in ACC32
```

The value in memory is stored in S.15 format but will be converted to S.31 format automatically.

Experimental Noize Inc.

Scottsdale, AZ USA

www.xnoize.com

sales@xnoize.com