Introduction

The as10k1 is an assembler designed for the emu10k1, the chip present in the SBLive, SB PCI 512 and SB PCI 1024 sound cards from creative. The emu10k1 is a DSP chip geared toward audio signal processing. It contains many common DSP features, such as the MAC instruction, as well as some more exotic feature, such as the TRAM delay-line engine. The first part of this manual will present an overview of the emu10k1's architecture. The second part will cover the as10k1 syntax.

Index

Last update: Nov 8, 2000

Overview of the Emu10k1 Audio Signal Processor

Introduction

The Emu10k1 is a custom 32 bit fixed point digital signal processor with an instruction set geared towards audio effects. The Emu10k1 has instructions for handling both fractional as well as integer based fixed point arithmetic. The Emu contains a convention Multiply-Accumulate Core (MAC) as well as a Harvard Architecture such as those found in most commercial DSPs. The emu10k1 features a patented TANK (a.k.a TRAM) engine for handling delay lines and table lookups. A summary of the EMU10k1 specs are shown in the table below:

Fine print:This info has been collected form Patents, articles, and reverse-engineering (by myself or others). I cannot guarantee that it is all correct. If you feel you have found a mistake please email me to inform me. Thanks.

Specifications

Parameter	Value	Unit
Sampling Rate	48000	Samples/sec
Instruction Execution Time	40.69	nsec
Instructions per Sample	512	Instructions
Program Memory Size	512	Instructions
Data Memory Size	256*4	bytes
Internal Tram Size	8k	samples (2 bytes/sample)
Int. Tram Maximum delay	170.667	msec
External Tram Size	up to 1	Mega Samples (2 bytes/sample)
Ext. Tram Maximum delay	21.84	sec

- The Emu10k1 is sample locked, i.e exactly 512 instructions are executed per sample. Since no branch instruction are implemented, this guarantees that programs will maintain real-time execution.
- Instructions always have 4 operands the values of which represent addresses of registers. The Memory map of registers is shown in the next section below.
- Audio data on Tram delay lines are compressed down 16 bits. Tram is implement both on onboard ram, as well as host based ram (i.e. your PC).

Internal Memory Map [1]

Name	Address	Description
INPUTS		
FX[0:15]	0x000 0x001 . . . . . . 0x00F	
In0[L:R]	0x010 0x011	Analog recording source for AC97 Codec IC. Can be Cd, mic, etc.
In1[L:R]	0x012 0x013	S/PDIF CD digital in on card.
In2[L:R]	0x014 0x015	mic? - Unknown
In3[L:R]	0x016 0x017	LiveDrive -- TOSLink Optical In
In4[L:R]	0x018 0x019	LiveDrive -- Line/Mic In 1
In5[L:R]	0x01A 0x01B	LiveDrive -- Coaxial S/PDIF Input
In6[L:R]	0x01C 0x01D	LiveDrive -- Line/Mic In 2
Unknown	0x01D 0x01E	Unknown
OUTPUTS		
Out0[L:R]	0x020 0x021	Front Analog Output
Out1[L:R]	0x022 0x023	LiveDrive -- TOSLink Optical Out
Out2[L:R]	0x024 0x025	Unknown
Out3[L:R]	0x026 0x027	LiveDrive -- headphone out
Out4[L:R]	0x028 0x029	Rear channel
Out5[L:R]	0x02A 0x02B	ADC recording buffer
Out6[L:R]	0x02C	microphone recording buffer
Unknown	0x02D 0x03f	Unknown or Unimplemented outputs
CONSTANTS		
Constant	0x040	0x00000000
Constant	0x041	0x00000001

Constant	0x042	0x00000002
Constant	0x043	0x00000003
Constant	0x044	0x00000004
Constant	0x045	0x00000008
Constant	0x046	0x00000010
Constant	0x047	0x00000020
Constant	0x048	0x00000100
Constant	0x049	0x00010000
Constant	0x04a	0x00080000
Constant	0x04b	0x10000000
Constant	0x04c	0x20000000
Constant	0x04d	0x40000000
Constant	0x04e	0x80000000
Constant	0x04f	0x7fffffff
Constant	0x050	0xffffffff
Constant	0x051	0xfffffffe
Constant	0x052	0xc0000000
Constant	0x053	0x4f1bbcdc
Constant	0x054	0x5a7ef9db
Constant	0x055	0x00100000
Hardware Registers		
Accumulator	0x056	67 bit wide accumulator
CCR	0x057	Condition Code Register
Noise Sources	0x058 0x059	Random number Generator--unknown distribution
Interupt Register	0x05A	Write to MSB to generate Interupt
Delay Base Address Counter	0x5B	See section on tram
Unknown Area	0x05C 0x05f	Unknown area
Unimplemented	0x060-0x0ff	Reserved/Unimplemented Area
General Purpose Registers (GPRs)		
GPR	0x100 0x101 . . . . . . 0x102	General Purpose Registers
Tram Data Registers		
ITRAM data registers	0x200 0x201 . . . . . . 0x27F	Internal Tram Data Access Registers

XTRAM data registers	0x280 0x281 . . . . . . 0x29F	External Tram Data Access Registers
unimplemented	0x2A0-0x2FF	Reserved/Unimplemented Area
Tram Address Registers		
ITRAM data registers	0x300 0x301 . . . . . . 0x37F	Internal Tram Address Registers
XTRAM data registers	0x380 0x381 . . . . . . 0x39F	External Tram Address Registers
unimplemented	0x3A0-0x3FF	Reserved/Unimplemented Area

Condition Code Register

The 5 bit CCR has the following format:

Bits	31-5	4	3	2	1	0
Function	Don't Care	S	Z	M	N	B

- S is set on Saturation
- Z is set on zero
- M is set on negative (Minus)
- N is set on Normalized (MSB=previous MSB)
- B is set on borrow

The CCR is set after each instruction based upon the Result of the instruction (R operand)

Accumulator

The Emu's accumulator is 67 bits wide. Multiplication of two 32 bit 2's-compliment numbers has at most a 63 bit result. This leaves 4 guard bits for sucessive MAC operations where intermediate instructions may overflow.

bits	66-64	62-31	31-0
name	guard bits	HIGH ACCUM	LOW ACCUM

yes, there is a 1 bit overlap between HIGH ACCUM and LOW ACCUM, this is normal.

Logarithmic Representation

The emu has the ability to convert values into a logarithmic representation, similar to floating point representation, but a bit more powerfull. The format follows the conventional sign-exponent-mantissa form, however the size of the exponent is variable. This allows a much wider range of numbers to be represented at the sacrifice of mantissa presision. The Logarithmic representation is best described in the section covering the LOG and EXP instructions which are used to convert from linear to log and back.[2]

Contents   Next

Emu10k1 Tram Engine[3]

The Emu10k1 has a special patented tram engine used to implement Delay lines as well as table look-ups. The tram engine can use internal RAM (4K samples) or external host based ram (up to 1M samples). From the programmer's perspective tram is accessed via a pair of registers. Tram address registers are used to store an address into the tram (i.e. the address from which you wish to read or write). Tram data registers are used to write/read data to/from the tram.

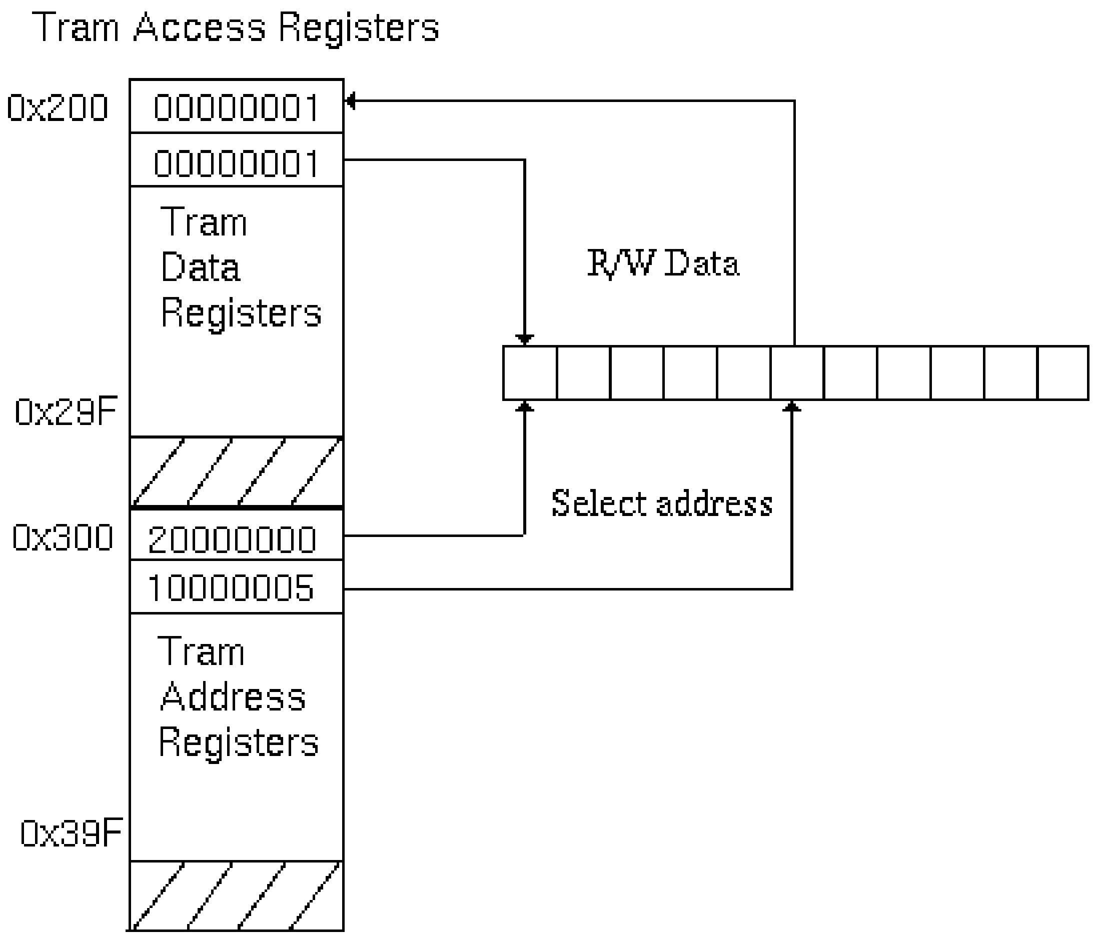

Fig. 1

As is shown in Fig. 1, the tram address registers contain a value which points to a delay-line/look-up location. The data is either written to or read from the tram data register pair. The pairing is straight forward, 0x200 is paired with 0x300, 0x201 is paired with 0x301, etc. There are actually two separate tram spaces. Internal tram is accessed via 0x200/0x300-0x27F/0x27F, external host based tram is accessed via 0x27F/0x37F-0x29F/0x39F. The tram engine updates the values from/in the tram data registers once per sample, thus the emu10k1 is physically limited to 0x7F internal accesses and 0x20 external accesses.

The Tram is 20 bit addressable, meaning only 20 bits of the 32bit tram address register actually represent an address. Bits 20-23 are an "opcode", they tell the tram address what to do (i.e. read or write). The opcodes are only read/writable by the host cpu, dsp programs cannot change a tram opcode, infact, dsp programs cannot even see the opcodes. The address can however be changed. The tram address as viewed from the host cpu are shown below. The two tram spaces do not overlap, so address 0x00000 in In Internal tram is different from 0x00000 in external tram. *Table 1, TRAM Address register viewed from Host CPU*

Bit #	23	22	21	20	0-19
Function	CLEAR	ALIGN	WRITE	READ	Address

The DSP code sees the tram address register differently. The whole register except the MSB (sign bit) is used to represent the tram address. Tram engine, however, only uses the 20 bits from 11-30. The least significant bits are ignored but maintain (one can use these bits when calculating addresses, these bits can represent "fractional" delay value for your dsp programs).

Table 2, TRAM Address register viewed from Emu10k1 processor

Bit #	31	30-11	10-0
Function	0	address	Fractional address

Circular Addressing:

The tram space is actually a circular address space. The addresses specified in the Tram Address Registers are added to a pointer known as the *Delay Base Address Counter(DBAC)*, and is used to access the actual tram memory. Fig 2. shows the calculation of the actual tram address. At each cycle, the Address Counter is decremented by 1, and thus an address specified in the tram address register will circulate around the tram space. The circulating space allows the creation of delay-lines as the data will appear to moving at each cycle.

Example:

t=0

DBAC=0x34, addr 0x11, actual address=0x11+0x34=0x45

t=1

DBAC=0x33, The data writen at t=0 is still in the same address (0x45), but from the dsp program's perpective it appears at addr = 0x45-0x33 = 0x12 . Thus, the data appears to have moved forward to the next tram memory location.

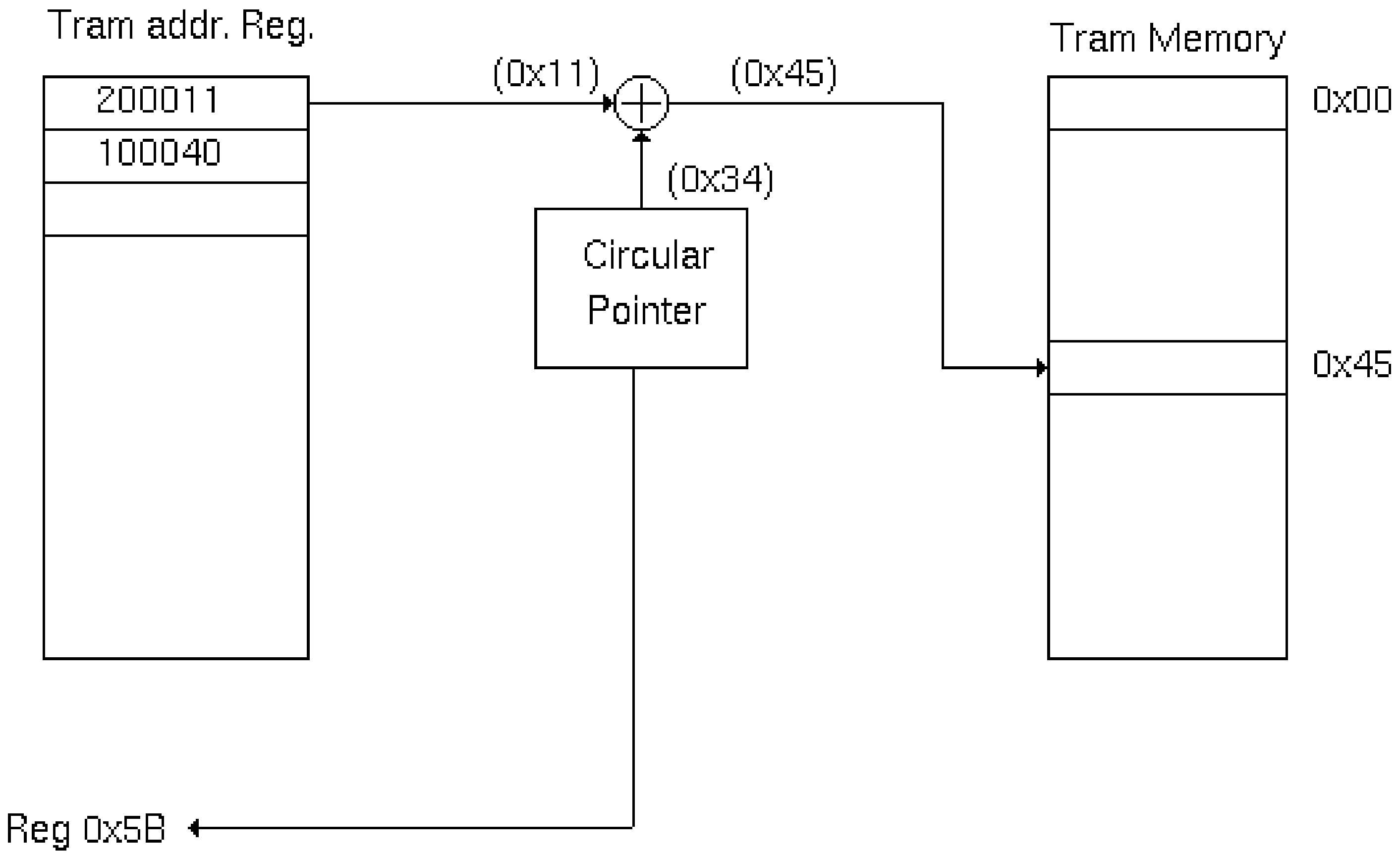

Fig. 2

In addition, the DBAC is available for dsp programs at register address 0x5B, this allows dsp programs to access tram in a non-circular fasion by "subtrating out" the pointer (more on that later).

Modulo addressing:

One final point about the DBAC is that it always counts from 0xfffff to 0 (or 0x7fff800 to 0 counting down by 0x800, from the dsp programs point of view), and starts over once it reaches 0. The amount of External tram depends on how much was allocated by the driver. Not shown above in Fig 2 is a modulo operation with respect to the tram size. This occurs after the summation operation of the address and the DBAC. The table below shows the possible tram sizes as determined by the driver, note that they sizes are all powers of 2 for simplicity.

Table 3, Possible External Tram Sizes (determined by driver)

	Number of Samples	**Modulo value (real)**	**Modulo Value (with 0x800 shift)**	**Ram allocated by driver (2 bytes/sample)**
0	8k	0x00002000	0x01000000	16k
1	16k	0x00004000	0x02000000	32k
2	32k	0x00008000	0x04000000	64k
3	64k	0x00010000	0x08000000	128k
4	128k	0x00020000	0x10000000	256k
5	256k	0x00040000	0x20000000	512k
6	512k	0x00080000	0x40000000	1024k
7	1024k	0x00100000	0x80000000	2048k

Tram Uses

Delay-lines

Tram can be used to create delay lines. Delay-lines are a basic DSP element. Delay line have a length N+1 (counted from 0). Data written at the beginning of the delay line, propagates down the delay line at the rate of the sample period. Thus for the emu10k1 who internal sample rate is 48Khz, a unit delay is 1/48kHz=20.83usec. To create a 1msec delay, for example, we would need 1m*48k=48 samples.

A delay line can be subdivided into multiple delay-line by writing at multiple places. For example, if we wanted a 16 sample delay-line line and a 32 sample delay-line we would construct the following read/write addresses(from host including read/write ops):

0x00200000	Write at 0
0x00100010	Read at 16
0x00200011	Write at 17
0x00100031	Read at 49

The second write, at 17, over writes the previous data effectively starting a new tram line. The above subdividing can be used to divide the entire tram spaces into as many delay-line as is needed by the dsp programs. The address can be written by the host dsp program loader, or can be written to by the dsp program itself. The former allows the delay-line length to be varied (or "modulated").

Since dsp programs and their delay-lines can be relocated by the patch loader, the dsp programs can use the address contained in the write register as an offset when calculating the modulated read address:

[read address]=[write address]+[calculated delay length]

Another important thing to consider is the 11 bit shift of addresses. When calculating the number of desired samples of delay, one must multiply this value by 0x800 to account for the shift. The [write address] is already shifted, thus the final equations becomes:

[read address]=[write address]+[calculated number of delay samples]*0x800

Look-up Tables

Unknown yet

Synchronizing Delay-lines (i.e. finding "0")

Tram can be used as a means of transfering data between the host CPU and the dsp programs (AC3 data, for example). Such a transfer typically would warrant the placement of data starting at know location in tram. Since the tram space appears to be circulating from the dsp program's view, we must have a way to find the beginning of the tram space.

To do so, we can use the Delay Base Address Regiser at 0x5B. Since tram address registers are summed with the DBAC, we must subtract it from from our address, before writing it to the Tram address register.

```
        macints    write.a,OFFSET,$5B,$50      ;;
```

Back Contents Next

Introduction

The As10k1's assembly syntax was originally intended to be similar to MC68K assembly. It has since evolved and now has a unique syntax.

General Assembly Syntax

Assembly lines generally have four fields separated by spaces or tabs:

Name_Field	Opcode_field	Operand_Field	Comment_Field
[symbol]	[mnemonic]	[operands]	[text]

With this assembler, each line can have a maximum of 256 characters and each symbol can be a maximum of 11 characters. Symbols ARE case sensitive, opcodes ARE NOT.

Symbols

A symbols is a words which placed at the beginning of a line (it must start at column 1) and is used by assembly directive. Symbols may be used to reference a register, an assembly-time equate, to reference a position in the program (the skip instruction), or just as a label to indicate to a reader where they are in a program (e.g. "start"). The use of a symbol is covered in greater detail in the sections describing assembly directives.

The first character in a symbol must be an alpha character* (a-z,A-Z). The remaining characters may be alphanumeric (a-z,A-Z,0-9) or an underscore (_). Symbols cannot exceed a length of 32 characters.

*An exception to this rule occurs when symbols are used for the skip command, in which case the symbol must start with a period.

Opcode

All instructions require 4 operands, they have the format

opcode R,A,X,Y

Here is a summary of the 16 opcodes. [1]

Opcode Number	Opcode Mnemonic	Instruction	comment
0x0	MACS	R = A + (X * Y >> 31)	saturation
0x1	MACS1	R = A + (-X * Y >> 31)	saturation
0x2	MACW	R = A + (X * Y >> 31)	wraparound
0x3	MACW1	R = A + (-X * Y >> 31)	wraparound
0x4	MACINTS	R = A + X * Y	saturation
0x5	MACINTW	R = A + X * Y	wraparound (31-bit)
0x6	ACC3	R = A + X + Y	saturation

0x7	MACMV	R = A, acc += X * Y	67 bit accum, you can grab MS 32 bits of LS 32 bits
0x8	ANDXOR	R = (A & X) ^ Y	
0x9	TSTNEG	R = (A >= Y) ? X : ~X	
0xa	LIMIT	R = (A >= Y) ? X : Y	
0xb	LIMIT1	R = (A < Y) ? X : Y	
0xc	LOG	...	
0xd	EXP	...	
0xe	INTERP	R = A + (X * (Y - A) >> 31)	; saturation
0xf	SKIP	R,CCR,CC_TEST,COUNT	

Operands

Operands can be specified as either a symbol or a value. hex values are prefixed by $, octal by @, and binary by %.

e.g.:

123 decimal value
$123 hex value
@123 octal value
%01101 binary value

The operands for emu10k1 instructions are always addresses of registers, there are no instruction which take immediate values.

Some directives have there own special operand forms

Comments

A semicolon ";" indicates to the assembler that everything afterwards is a comment.

Anything after a completed instruction (i.e. one that has four operands) will also be ignored. However, to be on the safe side, I recommend aways using a ";" to make sure that the assembler knows to ignore what follows.

Example

Here's an example of an instruction:

```
foo    macs    one,two,three,four ;comment
```

"foo" is a symbol (in this case it does nothing), "macs" is our instruction, and "one,two,three,four" is our operand, ";comment" is ignored.

---

back contents next

Emu10k1 Instructions

The instructions again where: [1]

Opcode Number	Opcode Neumonic	Instruction	comment
0x0	MACS	R = A + (X * Y >> 31)	saturation
0x1	MACS1	R = A + (-X * Y >> 31)	saturation
0x2	MACW	R = A + (X * Y >> 31)	wraparound
0x3	MACW1	R = A + (-X * Y >> 31)	wraparound
0x4	MACINTS	R = A + X * Y	saturation
0x5	MACINTW	R = A + X * Y	wraparound (31-bit)
0x6	ACC3	R = A + X + Y	saturation
0x7	MACMV	R = A, acc += X * Y	67 bit accum, you can grab MS 32 bits of LS 32 bits
0x8	ANDXOR	R = (A & X) ^ Y	
0x9	TSTNEG	R = (A >= Y) ? X : ~X	
0xa	LIMIT	R = (A >= Y) ? X : Y	
0xb	LIMIT1	R = (A < Y) ? X : Y	
0xc	LOG	...	
0xd	EXP	...	
0xe	INTERP	R = A + (X * (Y - A) >> 31)	; saturation
0xf	SKIP	R,CCR,CC_TEST,COUNT	

Operand Formats:

opcode R,A,X,Y

Important: The Operands always represents an address to a register. The Emu10k1 architeture does not implement indirect, immediate or any other type of addressing mode.

Instruction Specifics

In this section I will try to give a detailed description of each instruction. Since Creative/Emu have never released any official documents, some of the instructions are still not fully understood. As well, in some instance, special quirks arise (such as the ACCUM quirk), more quirk may also be lurking so beware.

If you discover any inacuracies or would like to add something don't hesitate to email me.

MACS/MACS1

MACS:

Formula: R = A + (X * Y >> 31)

MACS1:

Formula: R = A + (-X * Y >> 31)

Behaviour:

These instructions are frequently refered to as "fractional" macs. They performes the multiplication of operands X(or -X) and Y. Of the resulting 63 bit value, the 32 most significant (the "High" value) bits are taken and added with operand A. The result of the operation is then stored in the R operand.

They are called fractional macs because the operation of using the 32 MSB is equivalent to dividing by 2^31. Thus one could say that the formula is also given by: R = A+X*Y/(2^31)

When defining constants to be stored into general purpose registers (GPRs), a special operand can be used for the macs instruction. The "#" indicates to the assembler that the value of the operand should be multiplied by (2^31-1).

For example: #0.5 would be mutiplied (2^31-1) to give a value (in hex) $3fffffff. Fractional macs allow us to implement non-integer values (such as filter coefficients) using integer mathematics. The real numbers (between 1 and -1) are essentially quantized to 2^31 different values, giving us a delta (smallest increment between two values) of 2^(-31)=4.65661287307739e-10 which is more than accurate for most calculations.

For more info on storing values in GPRs see the DC directive.

Overflow:

These instructions saturate on overflow.

MACW/MACW1

MACW:

Formula: R = A + (X * Y >> 31)

MACW1:

formula: R = A + (-X * Y >> 31)

Behaviour:

These instruction behave in exactly the same way as the MACS and MASC1 instruction except that it handles overflows differently.

Overflow Behaviour

These instructions wraparound upon overflow. A simple illustration, imagine a number system with possible values between -9 and 9.

With saturation 8+2=9
with overflow 8+4=-8

The saturate method results in smaller increases in error with overflow (errors result in noise), a thus at first glance offers a better handling of overflow. However, the saturate method prevents us from using an important property of two's complement arithmetic:

"If several 2's-complement numbers whose sum would not overflow are added, then the result of 2's-complement accumulation of these numbers is correct, even though intermediate sums might overflow"[4].

This property can be used in FIR filtering were one has taken care to design the system such that the total output does not overflow. Note that this property does not hold true through a multiplication, however (thus you cannot blindly use this in IIR filtering).

MACINTS/MACINTW

Formula: R = A + X * Y

Behaviour:

Both these perform an integer mac operation.

Overflow:

MACINTS saturates upon overflow.

With MACINTW, the result is wrapped around but the sign bit (bit 31) is zeroed. Essentially the wrap around occurs around bit 30 instead of bit 31 (I have no idea why this would be useful).

ACC3

Formula: R = A + X + Y

Behaviour

ACC3 is perty straight forward. It simply sums the three operands placing the result in R. The result is saturated upon overflow.

It should be noted that the accumulation accurs in the High Accumulator.

MACMV

Formula: R = A, acc += X * Y

The MACMV instruction combines a multiply accumulate and a parallel move into one instruction. The result of the X Y multiplication is accumulated into the accumulator. The result must be fetched via a MAC,MACINT , or ACC3 instruction following the series of MACMV instrution. The Accumulator register address (0x56) can only be specified as the A registers, if used in X or Y, the emu10k1 will use 0 instead.

The accumulator is 67 bits wide, and will wraparound on overflow. The ACC3 and MACS will fetch the HIGH accumulator, were as the MACSINT instruction will fetch the LOW accumulator. When fetched, if the accumulator contains value greater than 63 bits one length, the accumulator will be saturated

The MACMV instruction is most useful for FIR filters as it can process each delayed unit in one instruction, including shifting the delays. Hence a N order FIR filter uses just a little over N+1 instructions (plus a few overhead instructions).

ANDXOR

Formula R = (A & X) ^ Y

Behaviour

The ANDXOR can be used to synthesis standard logical instructions. The table below shows some of the logical operations that can be synthesised.[5]

A	X	Y	Result
A	X	Y	(A AND X)XOR Y
A	X	0	A AND X
A	0xFFFFFFFF	Y	A XOR Y
A	0xFFFFFFFF	0xFFFFFFFF	NOT A
A	X	~X	A OR Y
A	X	0xFFFFFFFF	A NAND X

TSTNEG/LIMIT/LIMIT1

TSTNEG

Formula: R = (A >= Y) ? X : ~X

LIMIT

Formula: R = (A >= Y) ? X : Y

LIMIT1

Formula: R = (A < Y) ? X : Y

Behaviour:

The Result of these operations are condition upon the values of A and Y. The result of the TSTNEG instruction will be complemented if A<Y. The Limit and limit1 instructions function in a similar maner, but the value of the resultant can be X or Y depending on the conditional.

LOG

Behaviour: The LOG formula converts linear data into a Sign-Exponent-Mantissa form. The size of the exponent (i.e. number of bits occupied by) is variable between 2 and 5 bits. The sign occupies one bit and the mantissa occupies the rest. The LOG instruction can also perform absolute value, negative absolute, as well as negative on the resultant.[2]

The Resultant is stored in the following exponential form:

Sign	Exponent	Mantissa
1 bit	2-5 bits	29-26 bits

Instruction Format: LOG R,Lin_data,Max_exponent,Y

Where:

2 <= Max_exponent <= 31 (=0x1f);

And "sign" is a register containing a two bit number that has the following properties:

Value in Sign Operand	Description

0 0	Normal
0 1	absolute value*
1 0	negative of absolute value*
1 1	negative*

** Note that a 1 bit error exist in inverted values because it is actually the 1's compliment that is taken. (Still close enough for Rock and Roll! :-)*

The way it works [?]:

- 1. The sign bit is stored
- 2. The absolute value is taken of the data
- 3. The data is shifted left (<<)towards the binary point (the MSB).
- 4. Exp = Max_Exp_Size - Num_Of_Shifts
- 5. If MSB=1 and Exp <= Max_Exp_Size then: the implicit MSB is remove by shifting << one more bit and Exp is incremented.
- 6. The Resulting mantissa is Right Shifted by sizeof(Max_Exp_Size)+1
- 7. The sign bit and sign operand are compared and proper action is taken according to the table shown above.

Example of conversions (max_exp=7(3 bits),sign=00):

Linear Value	**LOG value**	**Break Down**			
		Sign	**Exponent**	**Mantissa**	**Implicit MSB?**
0x40004000	0x70001000	0	7	0x0001000	Yes
0x20002000	0x60001000	0	6	0x0001000	Yes
0x10001000	0x50001000	0	5	0x0001000	Yes
0x08000800	0x40001000	0	4	0x0001000	Yes
0x04000400	0x30001000	0	3	0x0001000	Yes
0x02000200	0x20001000	0	2	0x0001000	Yes
0x01000100	0x10001000	0	1	0x0001000	Yes
0x00800080	0x08000800	0	0	0x8000800	No
0xff008000	0xf008000f	1	0	0x008000f	No

EXP

Behaviour

This instruction performs oposite of the LOG instruction.

INTERP

Formula: R = A + (X * (Y - A) >> 31)

Behaviour:

Used for linear interpolating between two points. "X" should be positive and represents a fractional value between 0 and 1. "x" is the fraction of the interval between A and Y where the desired value is located.

SKIP

The skip command is available to provide some flow control in the dsp programs. The skip command has the following format:

Skip R,CCR,CC_TEST,COUNT

COUNT is a register containing the number of instructions o skip.

CC_TEST is a register containing a value which indicates under which conditions to skip on.

CCR is address of the CCR register. This can be any register, thus a previously saved CCR can be reused, or a constant can be used to implement an always skip (a NOP).

R is an address to copy the value of the CCR to for future use.

The CC_TEST operand uses one of 4 equations.

Form 1	(S * Z * M * B * N * S'* Z' * M' * B' * N') + (S * Z * M * B * N * S'* Z' * M' * B' * N') + (S * Z * M * B * N * S'* Z' * M' * B' * N')
Form 2	(S + Z + M + B + N + S'+ Z' + M' + B' + N') * (S + Z + M + B + N + S'+ Z' + M' + B' + N') * (S + Z + M + B + N + S'+ Z' + M' + B' + N')
Form 3	(S * Z * M * B * N * S'* Z' * M' * B' * N') + (S * Z * M * B * N * S'* Z' * M' * B' * N') + (S + Z + M + B + N + S'+ Z' + M' + B' + N')
Form 4	(S + Z + M + B + N + S'+ Z' + M' + B' + N') * (S + Z + M + B + N + S'+ Z' + M' + B' + N') + (S * Z * M * B * N * S'* Z' * M' * B' * N')

Presumably, the two most significant bits of CC_TEST select one of the above equations. The remaining 30 bits act as a mask that controls whether an element is active.

As10k1 Directives

The Directives are special instructions which tell the assembler to do certain things at assembly time. Directives can be used to tell the assembler to reserve Registers or Tram, to assign a name to the DSP program, to create macro, or to perform an assembly time 'For' loop. The following section describes each of the Directives used in the as10k1. A summary table is listed below.

Directive	symbol required(yes/no)	Arguments	Summary
NAME	No	"string"	Gives A name to the dsp patch
CONTROL	Yes	value,min,max	Defines a patch manager controllable register with initial value
DYNAMIC	Yes	number_of_gprs	defines 1 or more temporary gprs (may be reused by other patches)
STATIC	Yes	init_value1,init_value2,...	Defines one or more Static gprs with initial values
CONSTANT	yes	value1,value2,...	Defines one or more constants
EQU	Yes	Value_Equated	Creates an assembly time Equate
DELAY	Yes	Delay_length	Declares a delay line of given length
TABLE	Yes	Table_length	Declares a lookup table of given length
TREAD	Yes	tram_id,offset	creates a tram read associated with tram element "tram_id", with a given offset
TWRITE	Yes	tram_id,offset	creates a tram write associated with tram element "tram_id", with a given offset
INCLUDE	No	"file_name"	Include a file
MACRO	Yes	arg1,arg2,arg3,...	Defines a macro with given arguments
ENDM	No		Ends a Macro
FOR	No	symbol=start:finish	Assembly Time 'for' statement
ENDFOR	No		Ends a for loop
END	No		end of asm file

NAME

Specifies the name of the patch for identification purposes.

EQU

Equates a symbol to a be constant which is substituted at assembly time:

syntax:

<symbol> EQU <Value equated>

The value is a 16-bit integer value.

GPR Related Directives

Tips on selection of GPR types:

- 1. If you want a user controllable GPR, use type *control*
- 2. If you just want to create a temp GPR for intermediate calculations, use type *dynamic*
- 3. If you are carrying calculated values from one sample to the next, use type *static*
- 4. If you don't intend to write or change the value, use type *constant*

CONTROL

Declares a static GPR. The value in the control GPR is modifiable via the new mixer interface. The mixer is informed of the min and max values and will create a slider with value within that range. Upon loading the patch, the GPR is also loaded with the specified initial value.

Syntax:
<symbol> CONTROL <initialValue>,<MAX>,<MIN>

The arguments are 32-bit integer values.

DYNAMIC

Declares an automatic GPR from the emu10k1. This should be used for temporary storage only. It the GPR may be reused by other patches. No initial value is loaded to it.

Syntax
<symbol> DYN <numberOfStorageSpaces>
or
<symbol> DYNAMIC <numberOfStorageSpaces>

The argument can be an equated symbol, if no value is given 1 GPR is allocated.

STATIC

Declares a static GPR. The GPR is loaded with the specified initial value. The GPR is not shared with any other patch.

Syntax:
<symbol> STA <numberOfStorageSpaces>
or
<symbol> STATIC <numberOfStorageSpaces>

The argument is a 32-bit integer value

CONSTANT

Declares a constant GPR. Usage of Constant GPR allows for a greater efficient use of GPRs as patch which need access to the same constant will all share this GPR. The sharing is done by the patch loader at load time.

The patch loader will also use a hardware constant (see the register map) instead if the GPR constant happens to be one of them. (i.e. if you were to declare a constant with value 1, the patch loader would make your dsp program use the hw constant at address 0x41 instead).

Syntax:
<symbol> CONSTANT <value>
or
<symbol> CON <value>

Tram Memory Directives

To create a delay-line/look-up table, you'll want to first reserve some tram (using *delay* or *table*) then declare tram reads and writes (using *tread* and *twrite* respectively).

Note, lookup tables currently don't work (because we don't understand how to)

DELAY

Define Delay, used for allocating an amount of TRAM for a delay line.

<symbol> DELAY <Size>

The symbol is used to identify this delay line. The Size is the amount of TRAM allocated. The argument is a 32-bit integer value. A '&' indicates that the value represents an amount of time, the assembler will calculate the amount of samples required for the delay.

e.g:
foo DELAY &100e-3   ;a 100msec delay line
bar DELAY 1000   ;a 1000 sample delay line

TABLE

Defines lookup Table

same as DELAY but for lookup tables.

TREAD

Define read: used for defining a TRAM read point

<symbol> TREAD <lineName>,<Offset>

The tram read is associated with the delay or lookup line given by "lineName". This must be the same symbol used in defining the delay/lookup line. The Offset indicates an offset from the beginning of the delay/lookup line for which the read will occur.

"Symbol" will be given the address of the TRAM data register associated with this TRAM read operation. The assembler will create <symbol1>.a which has the address of the TRAM address register.

example:

```
 fooread tread 100e-3,foo
            macs fooread.a,one,two,three    ; writes a new tram read address
            macs temp,fooread,one,two       ; reads the data from the delay line
```

TWRITE

Same as TREAD but used for writing data to a delay line.

<symbol1> TWRITE <symbol2>,<value>

INCLUDE

The include directive is used to include external asm files into the current asm file.

Syntax:

INCLUDE <"file name">

The file name Must be enclosed in "".

examples:

END

The END directive should be placed at the end of the assembly source file. If the END directive is not found, a warning will be generated. All text located after the END directive is ignored.

Syntax:

[symbol] END   include "foobar.asm"

MACRO

Used for defining a macro

Defining Macro:

```
 <symbol> macro              arg1,arg2,arg3....
          ....
 <opcode> arg4,arg1,arg2... ;;for example
          ....
          ....
 endm
```

were the <symbol> used is the mnemonic representing the macro.

arg1,arg2,arg3... can be any symbols (auto-defining and local to a macro) as long as the symbol is not already in use outside the macro (i.e. as a DC, DS, etc.).

There's no limit to how many arguments can be used.

Using Macro:

```
   <macro nmeumonic> arg1,arg2,arg3....
```

where arg1,arg2,arg3,... are values or symbols.

Assembly-time For loop

usage:

```
    For <symbol>=<start>:<stop>
         ...
         ...
         macs <symbol>,....
         ...
         endfor
```

<start> and <stop> must be integers

Handling Skips

The as10k1 assembler handles skips in a special way explained best by an example:

```
 skip CRR,CRR,CC_test,.foo
       ...
       ...
       ...
 .foo  ...
```

the "." tell the assembler that the symbol is for skipping purposes, it will automatically define a GPR when parsing the skip instruction, and when the second .foo is encountered it will insert the number of instructions to skip. (the skip instruction needs a GPR by design, so don't blame me for the half-assness of it).

Back Contents Next

Summary of Argument Usage

Arguments can be a bit confusing, so I'm going to give a summary here. Firstly, the programmer should be aware of the two different types of arguments: addresses and values.

addresses:

Address are the location of GPRs, hw registers, tram registers, etc. Address are the only type of arguments to the instructions. The assembler can do basic assembly time arithmetic (add, sub, mult divide) to the addresses. Symbols used by the assembler are always addresses too. Thus in *foo sta $4*, foo is the address of the static register containing 0x4.

values:

Values are always 32 bit signed integers, because this is only thing the emu10k1 can handle. The assembler can do fancy assembly time conversions, but one should always realize that the end result is a 32-bit signed integer.

The possible assembly time conversions are:

Fractional

The 32-bit values can represent fractional value between 1 and -1. To accomplish this, the value given to the assembler by the programmer (which is between 1 and -1) is multiplied by 2^31-1.

example:

foo STA 0.5

--> Results in the 32 bit value: 0x3fffffff

Time

Since the sample rate is a fixed 48kHz, values of time can be converted. Times specified by the programmer (using &), is multiplied by 48k to give the number of samples required for the given amount of time. furthermore, when the the value is intended to be stored in a GPR, a mult by 0x800 is performed to account for the 11 bit shift in delay values when accessing tram.

thus:
STATIC, CONTROL, CONSTANT -->mult by 0x800
DELAY, TREAD, TWRITE -->no mult

examples:

foo con &0.02

--> Results in the 32 bit value: 0x001E0000

foo delay &0.02

--> Results in a delay-line being 0x3C1 samples long.

Summary of argument type modifiers

Modifier Symbol	Type/Conversion

None/Integer	Integer
None/fractional(1,-1)	Fractional
$	Hexadecimal
@	Octal
%	Binary
&	Time

Back Contents Next

Back Contents Next

I'll be putting a few example programs here

Back Contents Next

References

- 1. Koukola, Sousa, et al, dsp.txt. Linux Opensource Drivers, 1999-2000
- 2. Creatives Labs Inc., *Processor with Instruction Set for Audio Effects*. WIPO patent:WO 9901814 (A1), Jan. 14, 1999
- 3. Creatives Labs Inc., *Audio Effects Processor having Decoupled Instruction Execution and Audio Data Sequencing*. WIPO patent: WO 9901953 (A1), Jan. 14, 1999
- 4. Oppenheim and Schafer, *Discrete-Time Signal Processing*, p374. 2nd Edition , Prentice-Hall 1999, ISBN: 0-13-754920-2
- 5. Creatives Labs Inc., Patent WO 9901814 Fig. 4B, Substitute sheet (rule 26)