Moxie

Overview

Moxie is a general purpose bi-endian load-store processor, with sixteen 32-bit general purpose registers and a comprehensive ISA. It was originally designed to be an ideal target for the GNU Compiler Collection, and has since evolved to include many supervisory level instructions required to run an embedded RTOS such has RTEMS.

Most moxie instructions are 16-bits long, while the remainder include an additional 16- or 32-bit immediate value resulting in 32- and 48-bit instructions. A variable width instruction architecture was chosen over a fixed-width RISC implementation in order to optimize for instruction memory bandwidth, a key performance limiter for many FPGA applications.

As a fair warning to readers, it should be mentioned that the moxie architecture is still evolving. That being said, there have been few changes in recent history. Feedback, of course, is certainly welcome! Please send all comments to the author.

Registers

Moxie defines 16 32-bit registers as follows:

Name	Description
`$fp`	the frame pointer
`$sp`	the stack pointer
`$r0` through `$r13`	general purpose registers

In addition, there are a number of special registers whose values are accessible only with the Get Special Register (`gsr`) and Set Special Registers (`ssr`) instructions. Some of these registers have special purposes:

Special Register	Description
0	status register with the following bit values:

1	a pointer to the Exception Handler routine (invoked by `swi`, IRQs, Divide by Zero and illegal instructions (`bad`))
2	upon invocation of the Excecption Handler (see above), special register 2 will have one of four values..
3	the `swi` request number (by convention)
4	address of the supervisor mode stack on which exceptions are executed
5	return address upon entering the exception handler
6	reserved
7	reserved
8	reserved
9	an optional non-zero pointer to the Device Tree blob describing this device

Instruction Set

The moxie instruction set and encoding is evolving. Here's the current list of instructions and encodings supported in by the moxie toolchain.

All instructions are 16-bits long. Some 16-bit instructions are followed by a 32-bit immediate value. All of the opcode space not consumed by the encodings below is filled with the `bad` instruction.

and **00100110AAAABBBB**

Logical and. Performs a logical and operation on the contents of registers `$rA` and `$rB` and stores the result in `$rA`.

add **00000101AAAABBBB**

Add, long. Adds the contents of registers `$rA` and `$rB` and stores the result in `$rA`.

ashl **00101000AAAABBBB**

Arithmetic shift left. Performs an arithmetic shift left of `$rA` byt `$rB` bits and stores the result in `$rA`.

ashr **00101101AAAABBBB**

Arithmetic shift right. Performs an arithmetic shift right of `$rA` byt `$rB` bits and stores the result in `$rA`.

beq **110000vvvvvvvvvv**

Branch if equal. If the results of the last `cmp` demonstrated that `$rA` is equal to `$rB`, branch to the address computed by adding the signed 10-bit immediate value shifted to the left by 1 to the program counter. The branch is relative to the start of this instruction.

bge **110110vvvvvvvvvv**

Branch if greater than or equal. If the results of the last `cmp` demonstrated that the signed 32-bit value in `$rA` is greater than or equal to the signed 32-bit value in `$rB`, branch to the address computed by adding the signed 10-bit immediate value shifted to the left by 1 to the program counter. The branch is relative to the address of this instruction.

bgeu **111000vvvvvvvvvv**

Branch if greater than or equal, unsigned. If the results of the last `cmp` demonstrated that the unsigned 32-bit value in `$rA` is greater than or equal to the unsigned 32-bit value in `$rB`, branch to the address computed by adding the signed 10-bit immediate value shifted to the left by 1 bit to the program counter. The branch is relative to the address of this instruction.

bgt **110011vvvvvvvvvv**

Branch if greater than. If the results of the last `cmp` demonstrated that the signed 32-bit value in `$rA` is greater than the signed 32-bit value in `$rB`, branch to the address computed by adding the signed 10-bit immediate value shifted to the left by 1 bit to the program counter. The branch is relative to the address of this instruction.

bgtu **110101vvvvvvvvvv**

Branch if greater than, unsigned. If the results of the last `cmp` demonstrated that the unsigned 32-bit value in `$rA` is greater than the unsigned 32-bit value in `$rB`, branch to the address computed by the adding the signed 10-bit immediate value shifted to the left by 1 bit to the program counter. The branch is relative to the address of this instruction.

ble **110111vvvvvvvvvv**

Branch if less than or equal. If the results of the last `cmp` demonstrated that the signed 32-bit value in `$rA` is less than or equal to the signed 32-bit value in `$rB`, branch to the address computed by adding the signed 10-bit immediate value shifted to the left by 1 bit to the program counter. The branch is relative to the address of this instruction.

bleu **111001vvvvvvvvvv**

Branch if less than or equal, unsigned. If the results of the last `cmp` demonstrated that the unsigned 32-bit value in `$rA` is less than or equal to the unsigned 32-bit value in `$rB`, branch to the address computed by adding the signed 10-bit immediate value to the program counter. The branch is relative to the address of this instruction.

blt **110010vvvvvvvvvv**

Branch if less than. If the results of the last `cmp` demonstrated that the signed 32-bit value in `$rA` is less than the signed 32-bit value in `$rB`, branch to the address computed by adding the signed 10-bit immediate value shifted to the left by 1 bit to the program counter. The branch is relative to the address of this instruction.

bltu **110100vvvvvvvvvv**

Branch if less than, unsigned. If the results of the last `cmp` demonstrated that the unsigned 32-bit value in `$rA` is less than the unsigned 32-bit value in `$rB`, branch to the address computed by adding the signed 10-bit immediate value shifted to

the left by 1 bit to the program counter. The branch is relative to the address of this instruction.

bne **110001vvvvvvvvvv**

Branch if not equal. If the results of the last `cmp` demonstrated that `$rA` is not equal to `$rB`, branch to the address computed by adding the signed 10-bit immediate value shifted to the left by 1 bit to the program counter. The branch is relative to the address of this instruction.

brk **00110101xxxxxxxx**

Break. The software breakpoint instruction.

cmp **00001110AAAABBBB**

Compare. Compares the contents of `$rA` to `$rB` and store the results in the processor's internal condition code register. This is the only instruction that updates the internal condition code register used by the branch instructions.

dec **1001AAAAiiiiiiii**

Decrement. Decrement register `$rA` by the 8-bit value encoded in the 16-bit opcode.

div **00110001AAAABBBB**

Divide, long. Divides the signed contents of registers `$rA` and `$rB` and stores the result in `$rA`. Two special cases are handled here: division by zero asserts an Divide by Zero [[Exceptions|Exception]], and INT_MIN divided by -1 results in INT_MIN.

gsr **1010AAAASSSSSSSS**

Get special register. Move the contents of the special register S into `$rA`.

inc **1000AAAAiiiiiiii**

Increment. Increment register `$rA` by the 8-bit value encoded in the 16-bit opcode.

jmp **00100101AAAAxxxx**

Jump. Jumps to the 32-bit address stored in `$rA`. This is not a subroutine call, and therefore the stack is not updated.

jmpa **00011010xxxxxxxx iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii**

Jump to address. Jumps to the 32-bit address following the 16-bit opcode. This is not a subroutine call, and therefore the stack is not updated.

jsr **00011001AAAAxxxx**

Jump to subroutine. Jumps to a subroutine at the address stored in `$rA`.

jsra **00000011xxxxxxxx iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii**

Jump to subroutine at absolute address. Jumps to a subroutine identified by the 32-bit address following the 16-bit opcode.

ld.b **00011100AAAABBBB**

Load byte. Loads the 8-bit contents stored at the address pointed to by `$rB` into `$rA`. The loaded value is zero-extended to 32-bits.

ld.l **00001010AAAABBBB**

Load long. Loads the 32-bit contents stored at the address pointed to by `$rB` into `$rA`.

ld.s **00100001AAAABBBB**

Load short. Loads the 16-bit contents stored at the address pointed to by `$rB` into `$rA`. The loaded value is zero-extended to 32-bits.

lda.b **00011101AAAAxxxx iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii**

Load absolute, byte. Loads the 8-bit value pointed at by the 32-bit address following the 16-bit opcode into register `$rA`. The loaded value is zero-extended to 32-bits.

lda.l **00001000AAAAxxxx iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii**

Load absolute, long. Loads the 32-bit value pointed at by the 32-bit address following the 16-bit opcode into register `$rA`.

lda.s **00100010AAAAxxxx iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii**

Load absolute, short. Loads the 16-bit value pointed at by the 32-bit address following the 16-bit opcode into register `$rA`. The loaded value is zero-extended to 32-bits.

ldi.l **00000001AAAAxxxx iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii**

Load immediate, long. Loads the 32-bit immediate following the 16-bit opcode into register %rA.

ldi.b **00011011AAAAxxxx iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii**

Load immediate, byte. Loads the 32-bit immediate following the 16-bit opcode into register %rA. This is a poor encoding, as the 32-bit value really only contains 8-bits of interest.

ldi.s **00100000AAAAxxxx iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii**

Load immediate, short. Loads the 32-bit immediate following the 16-bit opcode into register %rA. This is a poor encoding, as the 32-bit value really only contains 16-bits of interest.

ldo.b **00110110AAAABBBB iiiiiiiiiiiiiiii**

Load offset, byte. Loads into `$rA` the 8-bit value from memory pointed at by the address produced by adding the 16-bit value following the 16-bit opcode to `$rB`. The loaded value is zero-extended to 32-bits.

ldo.l **00001100AAAABBBB iiiiiiiiiiiiiiii**

Load offset, long. Loads into `$rA` the 32-bit value from memory pointed at by the address produced by adding the 16-bit value following the 16-bit opcode to `$rB`.

ldo.s **00111000AAAABBBB iiiiiiiiiiiiiiii**

Load offset, short. Loads into `$rA` the 16-bit value from memory pointed at by the address produced by adding the 16-bit value following the 16-bit opcode to `$rB`. The loaded value is zero-extended to 32-bits.

lshr **00100111AAAABBBB**

Logical shift right. Performs a logical shift right of register `$rA` by `$rB` bits and stores the result in `$rA`.

mod **00110011AAAABBBB**

Modulus, long. Compute the modulus of the signed contents of registers `$rA` and `$rB` and stores the result in `$rA`.

mov **00000010AAAABBBB**

Move register to register. Move the contents of `$rB` into `$rA`.

mul **00101111AAAABBBB**

Multiply. Multiplies the contents of registers `$rA` and `$rB` and stores the lower 32-bits of a 64-bit result in `$rA`.

mul.x **00010101AAAABBBB**

Signed multiply, upper word. Multiplies the contents of registers `$rA` and `$rB` and stores the upper 32-bits of a 64-bit result in `$rA`.

neg **00101010AAAABBBB**

Negative. Changes the sign of `$rB` and stores the result in `$rA`.

nop **00001111xxxxxxxx**

Do nothing.

not **00101100AAAABBBB**

Logical not. Performs a logical not operation on the contents of register `$rB` and stores the result in register `$rA`.

or **00101011AAAABBBB**

Logical or. Performs a logical or operation on the contents of registers `$rA` and `$rB` and stores the result in `$rA`.

pop **00000111AAAABBBB**

Pop the 32-bit contents of the top of the stack pointed to by `$rA` into `$rB` and update the stack pointer in `$rA`. Stacks grown down.

push **00000110AAAABBBB**

Push the contents of `$rB` onto a stack pointed to by `$rA` and update the stack pointer in `$rA`. Stacks grown down.

ret **00000100xxxxxxxx**

Return from subroutine.

sex.b **00010000AAAABBBB**

Sign-extend byte. Sign-extend the lower 8-bits of `$rB` into a `$rA` as a 32-bit value.

sex.s **00010001AAAABBBB**

Sign-extend short. Sign-extend the lower 16-bits of `$rB` into a `$rA` as a 32-bit value.

ssr **1011AAAASSSSSSSS**

Set special register. Move the contents of `$rA` into special register S.

st.b **00011110AAAABBBB**

Store byte. Stores the 8-bit contents of `$rB` into memory at the address pointed to by `$rA`.

st.l **00001011AAAABBBB**

Store long. Stores the 32-bit contents of `$rB` into memory at the address pointed to by `$rA`.

st.s **00100011AAAABBBB**

Store short. Stores the 16-bit contents of `$rB` into memory at the address pointed to by `$rA`.

sta.b **00011111AAAAxxxx iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii**

Store absolute, byte. Stores the lower 8-bit contents of `$rA` into memory at the 32-bit address following the 16-bit opcode.

sta.l **00001001AAAAxxxx iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii**

Store absolute, long. Stores the full 32-bit contents of `$rA` into memory at the 32-bit address following the 16-bit opcode.

sta.s **00100100AAAAxxxx iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii**

Store absolute, short. Stores the lower 16-bit contents of `$rA` into memory at the 32-bit address following the 16-bit opcode.

sto.b **00110111AAAABBBB iiiiiiiiiiiiiiii**

Store offset, byte. Stores the 8-bit contents of `$rB` into memory at the address roduced by adding the 16-bit value following the 16-bit opcode to `$rA`.

sto.l **00001101AAAABBBB iiiiiiiiiiiiiiii**

Store offset, long. Stores the 32-bit contents of `$rB` into memory at the address roduced by adding the 16-bit value following the 16-bit opcode to `$rA`.

sto.s **00111001AAAABBBB iiiiiiiiiiiiiiii**

Store offset, short. Stores the 16-bit contents of `$rB` into memory at the address roduced by adding the 16-bit value following the 16-bit opcode to `$rA`.

sub **00101001AAAABBBB**

Subtract, long. Subtracts the contents of registers `$rA` and `$rB` and stores the result in `$rA`.

swi **00110000xxxxxxxx iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii**

Software interrupt. Trigger a software interrupt, where the interrupt type is encoded in the 32-bits following the 16-bit opcode.

udiv **00110010AAAABBBB**

Divide unsigned, long. Divides the unsigned contents of registers `$rA` and `$rB` and stores the result in `$rA`.

umod **00110100AAAABBBB**

Modulus unsigned, long. Compute the modulus of the unsigned contents of registers `$rA` and `$rB` and stores the result in `$rA`.

umul.x **00010100AAAABBBB**

Unsigned multiply, upper word. Multiplies the contents of registers `$rA` and `$rB` and stores the upper 32-bits of an unsigned 64-bit result in `$rA`.

xor **00101110AAAABBBB**

Logical exclusive or. Performs a logical exclusive or operation on the contents of registers `$rA` and `$rB` and stores the result in `$rA`.

zex.b **00010010AAAABBBB**

Zero-extend byte. Zero-extend the lower 8-bits of `$rB` into a `$rA` as a 32-bit value.

zex.s **00010011AAAABBBB**

Zero-extend short. Zero-extend the lower 16-bits of `$rB` into a `$rA` as a 32-bit value.

Updated on: Mon 29 December 2014