3 Instruction Set

The following table describes the instructions supported by the Amber 2x core.

Table 4 Amber 2 core Instruction Set

Name	Type	Syntax	Description
adc	REGOP	`adc{<cond>}{s} <Rd>, <Rn>, <shifter_operand>`	**Add with carry** adds two values and the Carry flag.
add	REGOP	`add{<cond>}{s} <Rd>, <Rn>, <shifter_operand>`	**Add** adds two values.
and	REGOP	`and{<cond>}{s} <Rd>, <Rn>, <shifter_operand>`	**And** performs a bitwise AND of two values.
b	BRANCH	`b{<cond>} <target_address>`	**Branch** causes a branch to a target address.
bic	REGOP	`bic{<cond>}{s} <Rd>, <Rn>, <shifter_operand>`	**Bit** *clear* performs a bitwise AND of one value with the complement of a second value.
bl	BRANCH	`bl{<cond>} <target_address>`	**Branch and link** cause a branch to a target address. The resulting instruction stores a return address in the link register (r14).
cdp	COREGOP	`cdp{<cond>} <coproc>, <opcode_1>, <CRd>, <CRn>, <CRm>, <opcode_2>`	**Coprocessor data processing** tells a coprocessor to perform an operation that is independent of Amber registers and memory. This instruction is not currently implemented by the Amber core because there is no coprocessor in the system that requires it.
cmn	REGOP	`cmn{<cond>}{p} <Rn>, <shifter_operand>`	**Compare negative** compares one value with the twos complement of a second value, simply by adding the two values together, and sets the status flags. If the p flag is set, the pc and status bits are updated directly by the ALU output.
cmp	REGOP	`cmp{<cond>}{p} <Rn>, <shifter_operand>`	**Compare** compares two values by subtracting <shifter operand> from <Rn>, setting the status flags. If the p flag is set, the pc and status bits are updated directly by the ALU output.
eor	REGOP	`eor{<cond>}{s} <Rd>, <Rn>, <shifter_operand>`	**Exclusive OR** performs a bitwise XOR of two values.
ldc	CODTRANS	`lcd{<cond>} <coproc>, <CRd>, <addressing_mode>`	**Load coprocessor** loads memory data from a sequence of consecutive memory addresses to a coprocessor. This instruction is not currently implemented by the Amber core because there is no coprocessor in the system that requires it.
ldm	MTRANS	`ldm{<cond>}<addressing_mode> <Rn>{!}, <registers>`	**Load multiple** loads a non-empty subset, or possibly all, of the general-purpose registers from sequential memory locations. It is useful for block loads, stack operations and procedure exit sequences.
		`ldm{<cond>}<addressing_mode> <Rn>, <registers_without_pc>^`	This version loads User mode registers when the processor is in a privileged mode. This is useful when performing process swaps.
		`ldm{<cond>}<addressing_mode> <Rn>{!}, <registers_and_pc>^`	This version loads a subset, or possibly all, of the general-purpose registers and the PC from sequential memory locations. The status bits are also loaded. This is useful for returning from an exception.
ldr	TRANS	`ldr{<cond>} <Rd>, <addressing_mode>`	**Load register** loads a word from a memory address. If the address is not word-aligned, then the word is rotated left so that the byte addresses appears in bits [7:0] of Rd.
ldrb	TRANS	`ldr{<cond>}b <Rd>, <addressing_mode>`	**Load register byte** loads a byte from memory and zero-extends the byte to a 32-bit word.
mcr	CORTRANS	`mcr{<cond>} <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm>{, <opcode_2>}`	**Move to coprocessor from register** passes the value of register <Rd> to a coprocessor.
mla	MULT	`mla{<cond>}{s} <Rd>, <Rm>, <Rs>, <Rn>`	**Multiply accumulate** multiplies two signed or unsigned 32-bit values, and adds a third 32-bit value. The least significant 32 bits of the result are written to the destination register.
mov	REGOP	`mov{<cond>}{s} <Rd>, <shifter_operand>`	**Move** writes a value to the destination register. The value can be either an immediate value or a value from a register, and

Name	Type	Syntax	Description
			can be shifted before the write.
mrc	CORTRANS	`mrc{<cond>} <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm>{, <opcode_2>}`	**Move to register from coprocessor** causes a coprocessor to transfer a value to an Amber register or to the condition flags.
mul	MULT	`mul{<cond>}{s} <Rd>, <Rm>, <Rs>`	**Multiply** multiplies two signed or unsigned 32-bit values. The least significant 32 bits of the result are written to the destination register.
mvn	REGOP	`mvn{<cond>}{s} <Rd>, <shifter_operand>`	**Move not** generates the logical ones complement of a value. The value can be either an immediate value or a value from a register, and can be shifted before the MVN operation.
orr	REGOP	`orr{<cond>}{s} <Rd>, <Rn>, <shifter_operand>`	**Logical OR** performs a bitwise OR of two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the OR operation.
rsb	REGOP	`rsb{<cond>}{s} <Rd>, <Rn>, <shifter_operand>`	**Reverse subtract** subtracts a value from a second value.
rsc	REGOP	`rsc{<cond>}{s} <Rd>, <Rn>, <shifter_operand>`	**Reverse subtract with carry** subtracts one value from another, taking account of any borrow from a preceding less significant subtraction. The normal order of the operands is reversed, to allow subtraction from a shifted register value, or from an immediate value.
sbc	REGOP	`sbc{<cond>}{s} <Rd>, <Rn>, <shifter_operand>`	**Subtract with carry** subtracts the value of its second operand and the value of NOT(Carry flag) from the value of its first operand. The first operand comes from a register. The second operand can be either an immediate value or a value from a register, and can be shifted before the subtraction.
stc	CODTRANS	`stc{<cond>} <coproc>, <CRd>, <addressing_mode>`	**Store coprocessor** stores data from a coprocessor to a sequence of consecutive memory addresses. This instruction is not currently implemented by the Amber core because there is no coprocessor in the system that requires it.
stm	MTRANS	`stm{<cond>}<addressing_mode> <Rn>{!}, <registers>`	**Store multiple** stores a non-empty subset (or possibly all) of the general-purpose registers to sequential memory locations. The '!' causes Rn to be updated. The registers are stored in sequence, the lowest-numbered register to the lowest memory address (start_address), through to the highest-numbered register to the highest memory address (end_address).
		`STM{<cond>}<addressing_mode> <Rn>, <registers>^`	This version stores a subset (or possibly all) of the User mode general-purpose registers to sequential memory locations. The registers are stored in sequence, the lowest-numbered register to the lowest memory address (start_address), through to the highest-numbered register to the highest memory address (end_address).
str	TRANS	`str{<cond>} <Rd>, <addressing_mode>`	**Store register** stores a word from a register to memory.
strb	TRANS	`str{<cond>}b <Rd>, <addressing_mode>`	**Store register byte** stores a byte from the least significant byte of a register to memory.
sub	REGOP	`sub{<cond>}{s} <Rd>, <Rn>, <shifter_operand>` `i.e. Rd = Rn - shifter_operand`	**Subtract** subtracts one value from a second value.
swi	SWI	`swi{<cond>} <immed_24>`	**Software interrupt** causes a SWI exception. <immed_24> Is a 24-bit immediate value that is put into bits[23:0] of the instruction. This value is ignored by the Amber core, but can be used by an operating system SWI exception handler to determine what operating system service is being requested.
swp	SWAP	`swp{<cond>} <Rd>, <Rm>, [<Rn>]`	**Swap** loads a word from the memory address given by the value of register <Rn>. The value of register <Rm> is then stored to the memory address given by the value of <Rn>, and the original loaded value is written to register <Rd>. If the same register is specified for <Rd> and <Rm>, this instruction swaps the value of the register and the value at the memory address.
swpb	SWAP	`swp{<cond>}b <Rd>, <Rm>, [<Rn>]`	**Swap Byte** swaps a byte between registers and memory. It loads a byte from the memory address given by the value of register <Rn>. The value of the least significant byte of register <Rm> is stored to the memory address given by

Name	Type	Syntax	Description
			<Rn>, the original loaded value is zero-extended to a 32-bit word, and the word is written to register <Rd>. Can be used to implement semaphores.
teq	REGOP	`teq{<cond>}{p} <Rn>, <shifter_operand>`	**Test equivalence** compares a register value with another arithmetic value. The condition flags are updated, based on the result of logically XORing the two values, so that subsequent instructions can be conditionally executed. If the p flag is set, the pc and status bits are updated directly by the ALU output.
tst	REGOP	`tst{<cond>}{p} <Rn>, <shifter_operand>`	**Test** compares a register value with another arithmetic value. The condition flags are updated, based on the result of logically ANDing the two values, so that subsequent instructions can be conditionally executed. If the p flag is set, the pc and status bits are updated directly by the ALU output.