

Flowstone Assembler opcode reference

This document lists opcodes available in Flowstone assembler, along with their descriptions (with only the relevant information), latencies and throughputs. The values of latencies and throughputs come from multiple sources (which sometimes said contradicting data) and are related to most common Intel processors (nevertheless values for other processors like Pentium or AMD should be fairly similar). If the value is specified as range then different processors have different values or the instruction has variable latency/throughput depending on the input values.

When the instruction reads or write data to/from RAM (has variable as an input/output instead of register) it usually has additional latency for loading data, which may be anywhere from 0-3 when reading from 1st level cache to several hundred on cache miss. Also loading and storing data to/from RAM ma affect the throughput (because storing or loading commonly can't happen in parallel).

Examples for the opcodes will be added over time.

description of operands used in this document further on	
xmm0/xmm1	any xmm 128bit register
var	SSE variable - 128bit array of 32bit values
var[eax]	SSE array - array of SSE values - eax adds offset from the pointer (the index)
reg	general purse register - usable are only eax and ebx in most cases
st(int)	floating point stack register - internally as 80bit float (commonly called long double)
[reg]	variable in RAM (either 128bit or 32bit depending on the instruction) specified by address in general purse register
mem / var[channel]	32bit variable - in case of flowstone assembler a single channel of SSE variable. channel is an integer constant in 0-3 range
int	integer constant, usually in range specified in the opcode description

Instruction syntax	Full name	description	latency	reciprocal throughput	examples
SSE instructions	Streaming SIMD extension	These instructions operate on 128bit values (SSE variables or xmm registers). They process packed 32bit data (4 channels of 32bit variables). Each channel is processed completely separately.			
movaps xmm0,xmm1;	Move aligned packed single-precision floats	moves 128bit data from source register to destination register	1	1	
movaps xmm0,var/var[eax];	Move aligned packed single-precision floats	moves 128bit data from source memory to destination register. May be used to load data from Arrays (all 4 channels at a time) where index is specified by eax register as offset in bytes. Therefore eax must be multiple of 16.	2-3	1	
movaps var/var[eax],xmm0;	Move aligned packed single-precision floats	moves 128bit data from source register to destination memory. May be used to store data into Arrays (all 4 channels at a time) where index is specified by eax register as offset in bytes. Therefore eax must be multiple of 16.	3	1	

movaps xmm0,[eax];	Move aligned packed single-precision floats	moves 128bit data from memory specified by address in eax register to xmm register	2-3	1	
movaps [eax],xmm0;	Move aligned packed single-precision floats	moves 128bit data from register to memory specified by address in eax register	3	1	
andps xmm0,xmm1/var;	logical bitwise AND	Calculates logical bitwise AND of the two operands and stores the result into destination register	1	0.33-1	
orps xmm0,xmm1/var;	logical bitwise OR	Calculates logical bitwise OR of the two operands and stores the result into destination register	1	0.33	
andnps xmm0,xmm1;	logical bitwise AND NOT	Calculates logical bitwise AND NOT of the two operands and stores the result into destination register. This instruction has syntax coloring bug - it is marked as unknown (and the code after it is marked black), but the code will compile normally.	1	0.33-1	
addps xmm0,xmm1/var;	add packed single-precision floats	Adds the two operands and stores the result in the destination register	3	1	
subps xmm0,xmm1/var;	subtract packed single-precision floats	Subtracts the source operand from destination operand and stores the result into destination operand	3	1	
mulps xmm0,xmm1/var;	multiply packed single-precision floats	multiplies the two operands and stores the result into destination operand	4-6	1-2	
divps xmm0,xmm1/var;	divide packed single-precision floats	divides the destination operand by source operand and stores the result into destination operand	12-30	5-30	
minps xmm0,xmm1/var;	minimum of packed single-precision floats	Compares values in the operands and stores the smaller in destination operand	3	1	
maxps xmm0,xmm1/var;	maximum of packed single-precision floats	Compares values in the operands and stores the smaller in destination operand	3	1	
cmpps xmm0,xmm1/var,int;	compare packed single-precision floats	compares two operands and sets all bits true in the destination operand if result is true, otherwise sets all bit false. Control integer sets the comparison mode: 0 ~ equal, 1 ~ less, 2 ~ less or equal, 3 ~ unordered (true if at least one of the values is NaN), 4 ~ not equal, 5 ~ not less (greater or equal), 6 ~ not less nor equal (greater), 7 ~ ordered (nether of values is NaN)	3	1	
rcpps xmm0,xmm1/var;	reciprocal of packed single-precision float	calculates the reciprocal of value in source operand and stores the result in destination operand. This instruction is faster than divps but also yields only half the precision (12bit).	3-5	1-2	
sqrtps xmm0,xmm1;	square root of packed single-precision float	calculates square root of the source operand and stores the result in destination operand	10-25	7-25	

cvtps2dq xmm0,xmm1/var;	convert packed singles to doubleword integers	converts 32bit floating-point values in source operand to 32bit integers and stores them into destination operand. If value is inexact, it is rounded to integer. If the input value is above or below range of 32bit signed integer indefinite integer value is put.	3-6	1-6	
cvtdq2ps xmm0,xmm1/var;	convert packed doubleword integers to floats	converts 32bit integer values in source operand to 32bit floats and stores them into destination operand. The values get rounded to nearest floating point value if the input integer has more than 24 significant digits (which is the precision of 32bit float mantissa).	3-6	1-6	
padd xmm0,var;	add packed doubleword integers	Adds the two operands and stores the result in the destination register	1	2	
pslld xmm0,int;	Shift Packed Data Left Logical	performs logical shift left on each 32bit segment in register by number specified in the count operand (integer). For values above 31 the register is cleared to all zeros.	1	1	
pand xmm0,var;	Logical and of packed data	performs logical AND of data in the operands and stores the result into destination operand. Function-wise it's identical with andps, but pand is executed by integer unit.	1	0.33	
shufps xmm0,xmm1,int;	Shuffle packed floating point data	Moves two of the four packed single-precision floating-point values from destination operand (first operand) into the low quadword (channels 0,1) of the destination operand; moves two of the four packed single-precision floating-point values in the source operand into to the high quadword (channels 2,3) of the destination operand. Third operand is a 8bit integer. Each group of 2 bits specifies channel from which the value should be taken.	1	1	
ALU operations	Arithmetic logical unit	These are the basic CPU instructions. They operate on 31bit registers - namely eax (general purpose register), ebx (general purpose register), ecx (loop counter - holds the sample index), ebp (stack base pointer - all variables in stream part are ebx+integer), esp (stack pointer - points to value on top of the stack). These operations also include conditional jumps, through which loops and code branching is implemented. Unlike SSE instructions, some ALU instructions update flags. Comparison flags hold information about previous arithmetic operation (whether result was positive/negative, zero/nonzero) which affects behavior of conditional jumps.			
mov reg,reg/int;	move 32bit data from source operand (integer/register) to destination register	This operation does not update flags	1	0.33-1	
mov reg,mem/[eax]/[ebp+int];	move 32bit data from source operand (memory) to destination register.	You may load data from specific memory pointed by eax or ebp register. This operation does not update flags	2	0.5-1	

mov mem/[eax]/[ebp+int],reg;	move 32bit data from source operand (register) to destination memory.	You may store data specific memory pointed by eax or ebp register. This operation does not update flags	3	1	
mov eax,xmm0;	move 32bit data from 128bit register	moves 32bit value in first channel (channel [0]) in xmm register to eax register	1-2	0.33-1	
and reg,reg/int;	Logical bitwise AND of 32bit data	performs logical bitwise AND of two values and stores the result in destination register. This operation does update flags.	1	0.33	
and reg,mem;	Logical bitwise AND of 32bit data	performs logical bitwise AND of two values and stores the result in destination register. This operation does update flags.	1	0.5-1	
or reg,reg;	Logical bitwise OR of 32bit data	performs logical bitwise OR of two values and stores the result in destination register. This operation does update flags.	1	0.33	
add reg,reg/int;	Add 32bit integers	Adds source operand to destination operand. This operation does update flags.	1	0.33-1	
add reg,mem;	Add 32bit integers	Adds source operand to destination operand. This operation does update flags.	1	0.5-1	
inc [reg];	increase 32bit integer by 1	increases 32bit integer at address [eax] by 1. This operation updates comparison flags.	6	2	
sub reg,int;	subtract 32bit integers	Subtracts a constant from destination register. Instructions that would allow to subtract non-constant values (from register or memory) are unfortunately not supported by Flowstone assembler. This operation does update flags.	1	0.33-1	
shl reg,int;	logical shift left of 32bit data	performs logical shift left. May be used to multiply integer by 2^?. Warning: the second operand must be positive - for shifting right use "shr". This operation does update flags.	1	0.5	
shr reg,int;	logical shift right of 32bit data	preforms logical shift right. May be used to multiply integer by 2^? Warning: the second operand must be positive - for shiftingleft use "shl". This operation does update flags.	1	0.5	
push reg;	push register onto memory stack	increases esp by 4 and stores the value in that position in RAM - on top of the stack. The stack pointer must be unchanged after code is executed. Rule of thumb is to push the same amount of times you pop - this is particularly hard when code branches and loops. This operation does not update flags.	3	1	

pop reg;	pops value from memory stack into register	Loads the value from top of the stack (pointed by esp) into register and decreases esp by 4. The stack pointer (esp) must be unchanged after code is executed. Rule of thumb is to push the same amount of times you pop - this is particularly hard when code branches and loops. This operation does not update flags.	2	0.5-1	
cmp reg,reg/int;	compare 32bit integers	Compares the two values and updates comparison flags. Values in both operands are left unchanged. Internally this operation is identical with "sub" - but it does not store the result in destination operand. Because it updates flags it is convenient for setting condition for jumps.	1	0.33-1	
jz label;	Jump if zero to "label"	Checks the comparison flags and jumps to position in code marked with label if previous result was zero.	0	1-2	
jnz label/int;	Jump if not zero to "label" or by int	Checks the comparison flags and jumps to position in code marked with label if previous result was non-zero. With this particular instruction you may also jump by specific amount of instructions. However, because instructions length in code memory differs it is hard to get the value right.	0	1-2	
jl label;	Jump if less to "label"	Checks the comparison flags and jumps to position in code marked with label if previous result was non-zero negative (in case of comparison first value was smaller then second)	0	1-2	
jnl label;	Jump if not less (if greater or equal) to "label"	Checks the comparison flags and jumps to position in code marked with label if previous result was zero or positive(in case of comparison first value was greater or equal to second)	0	1-2	
jg label;	Jump if greater to "label"	Checks the comparison flags and jumps to position in code marked with label if previous result was non-zero and positive(in case of comparison first value was greater then second)	0	1-2	
jng label;	Jump if not greater (if less or equal) to "label"	Checks the comparison flags and jumps to position in code marked with label if previous result was zero or negative(in case of comparison first value was less or equal to second)	0	1-2	

rdtsc;	Read time-stamp counter	Time-stamp counter is a 64bit integer that increases by 1 every clock cycle. This instruction loads the most significant bytes to edx and least significant bytes to eax. It does not update flags. Teoretically it can be used to measure CPU load of peaces of code, but in reality it is not practical nor reliable for several reasons.	?	24	
call reg;	call procedure	This instruction can call procedure by its relative position to this instruction. Currently this instruction has no use.	?	?	
ret;	return from procedure	This instruction jumps back from where procedure was called.	?	?	
FPU instructions	Floating point unit	These instructions operate on 80bit floating point (commonly called long double) stack. The stack is composed of 8 registers named st(0-7) respectively. You rarely input names of the registers as operands - for most operations they are fixed. Some operations push values onto the stack while others pop from the stack. Stack must be empty after code execution otherwise crashes occur. It is particularly hard to keep track of the stack when code loops and branches so, use these instructions with care.			
fld var[channel]/var[edx]/[reg]/st(int);	load single precision float	Loads 32bit float value from memory or FPU stack register, converts it to 80bit float and pushes it onto the stack. Memory may be specified by address in register, channel of particular SSE variable or offset from pointer of array. Data must be 32bit aligned, so eax must increment in multiples of 4.	3	1	
fild var[channel];	load 32bit integer	Loads 32bit integer value from memory, converts it to 80bit float and pushes it onto the stack.	6	1	
fld1;	load (+1.0)	Pushes positive 1 onto the stack	2	2	
fldlg2;	load (log2(10))	Pushes base 2 logarithm of 10 onto the stack	2	2	
fst var[channel];	store as single precision float	stores the value in st(0) register into memory (the value is converted to single precision float).	4	1	

		Stores the value in st(0) register into memory or another st() register and pops the stack. Memory may be specified as channel of SSE variable, offset from pointer of array or by specific memory address. Data must be 32bit aligned so, therefore eax must increment in multiples of 4. When using this instruction to move st(0) to another st() register, be aware, that the stack will be popped. That means the value actually end in st(int-1). fstp st(0); simply pops (discards) the value on top of the stack.			
fstp var[channel],var[edx],[reg],st(int);	store as single precision float and pop register stack		4	1	
fist var[channel];	store 32bit signed integer	converts the value in st(0) into signed 32bit integer and stores it in specified channel of variable.	7	1-2	
fistp var[channel];	store 32bit signed integer and pop register stack	converts the value in st(0) into signed 32bit integer and stores it in specified channel of variable. Then pops the register stack	7	1-2	
fxch;	exchange st(0) and st(1)	Exchanges the content of st(0) and st(1) registers.	0-1	0.5-1	
fadd;	add st(0) to st(1) and pop	Adds content of st(0) register to st(1) register and pops the stack (so the result is actually in st(0) in the end)	3	1	
fsub;	subtract st(0) from st(1) and pop	subtracts content of st(0) register from st(1) register and pops the stack (so the result is actually in st(0) in the end)	3	1	
fmul;	multiply st(0) with st(1) and pop	multiplies content of st(0) register with st(1) register, stores result in st(1) and pops the stack (so the result is actually in st(0) in the end)	5	1	
frndint;	Round st(0) to integer	Round st(0) to integer. This operation is slow - recommend using different rounding implementations if possible.	11-22	22	
fscale;	scale st(0) by integer power of two of st(1)	Rounds down st(1) to integer and adds this to the exponent of st(0). st(0)=st(0)*2^rounddown(st(1)) Unlike previous operations this one doesn't pop the stack, so both the scaled number and the one that was used to scale with are kept	12	12	
fprem;	partial remainder of st(0)/st(1)	computes the modulo of st(0)/st(1) and stores it in the st(0). DSP code a%b is code the equivalent this instruction	14-21	14-21	
f2xm1;	2^x -1	computes st(0)= 2^st(0) -1 The value of st(0) must be in <-1,1> range.	58-68	58-68	

f _{sin} ;	sine	Calculates sine of st(0) and stores it in st(0). Input angle is in radians.	40-120	40-120	
f _{cos} ;	cosine	Calculates cosine of st(0) and stores it in st(0). Input angle is in radians.	40-120	40-120	
f _{sincos} ;	sine and cosine	Calculates sine and cosine of st(0). First stores the sine in st(0) and then pops cosine onto the stack. (in the end st(0)=cosine, st(1)=sine). It is significantly faster than computing sine and cosine separately.	43-130	43-130	
f _{ptan} ;	partial tangent	Computes tangent of st(0), stores the result in st(0) and pushes +1 onto the stack. (in the end st(0)=+1, st(1)=tan	45-130	45-130	
f _{yl2x} ;	y*log ₂ (x);	calculates st(1)*log ₂ (st(0)), stores it in st(1) and pops the stack. (so the result ends up in st(0) in the end)	80-140	80-140	