
ZPU Reference Handbook



Tiago Gasiba

© 2015

Table of Contents

Introduction.....	3
Instruction Set Summary.....	4
Stack Operation Definitions.....	4
Memory Operations.....	4
Core instructions summary.....	5
Optional instructions (emulated).....	6
Instruction Mapping.....	7
Implemented Instructions.....	8
Instructions Specification.....	9
State Machine Transition Diagram.....	19
Legal Notice.....	20

Introduction

The Zylin ZPU is the worlds smallest 32 bit CPU with GCC tool chain. The ZPU is a small CPU in two ways: it takes up very little resources and the architecture itself is small. The latter can be important when learning about CPU architectures and implementing variations of the ZPU where aspects of CPU design is examined. In academia students can learn VHDL, CPU architecture in general and complete exercises in the course of a year. The current ZPU instruction set and architecture has not changed for the last couple of years and can be considered quite stable. This shall be presented in detail the following chapters.

Part of this work is based on previous work done by Álvaro Lopes - alvieboy@alvie.com (see legal notice) on the ZPUino – a derivative work of the original ZPU core by Øyvind Harboe - oyvind.harboe@zylin.com. The original ZPUino can be found on the internet on the following website: <http://www.alvie.com/zpuino>. Furthermore, the original ZPU and the “ZPU Project” can also be found on the internet on the following website: <https://github.com/zylin/zpu>.

Instruction Set Summary

Stack Operation Definitions

TOS = Top Of Stack = SP
 mem[SP] = valid data = stackA

PUSH:

SP = SP - 1;
 mem[SP] = data;

POP:

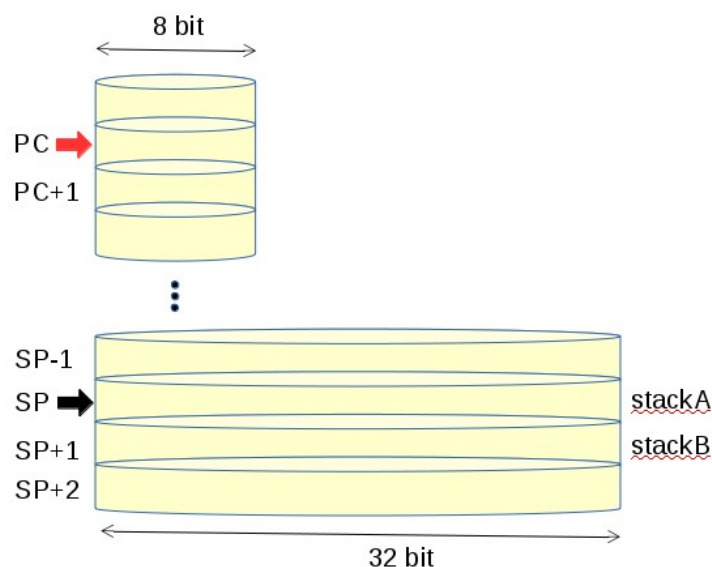
data = mem[SP];
 SP = SP + 1;

Memory Operations

PC : Accesses memory in 8 bit cells

SP : Accesses memory in 32 bit cells

NOTE: although PC points to 8 bit cells, the ZPU state machine always fetches 32 bit words and internally breaks down the words into bytes. The Stack Pointer, however, is a pointer to 32 bit cells which are aligned on 4-byte boundary, i.e. SP results in a memory fetch to address 4*SP and.



NOTE: the CPU implementation in VHDL is such that the TOS

(stackA) and mem[SP+1] (stackB), i.e. both instruction operands, are normally not immediately written back to memory in order to save CPU cycles. Care should be taken while reading the code to exactly understand when (due to state transitions) the stackA and/or stackB need to be written back to memory. See also instruction description below for a better understanding.

This means that stackA and stackB internal variables are actually cached versions of the corresponding memory positions. When SP changes, the stackA and stackB have to be updated accordingly and so does the memory positions corresponding to stackA (SP) and/or stackB (SP+1) before SP is updated.

Core instructions summary

Mnemonic	Opcode	Description
BREAKPOINT	0000 0000	Sets 'break' line to logic '1'
IM x	1xxx xxxx	
STORESP x	010 \bar{x} xxxx	
POP	0101 0000	Implemented using STORESP 0
POPDOWN	0101 0001	Implemented using STORESP 1
LOADSP x	011 \bar{x} xxxx	
ADDSP x	0001 xxxx	
EMULATE x	001x xxxx	
POPPC	0000 0100	
LOAD	0000 1000	
STORE	0000 1100	
PUSHSP	0000 0010	
POPSP	0000 1101	
ADD	0000 0101	
AND	0000 0110	
OR	0000 0111	
NOT	0000 1001	
FLIP	0000 1010	
NOP	0000 1011	

Optional instructions (emulated)

<i>Mnemonic</i>	<i>Opcode</i>	<i>Decimal</i>	<i>Description</i>
?	0010 0000	32	
N/A	0010 0001	33	
LOADH	0010 0010	34	
STOREH	0010 0011	35	
LESSTHAN	0010 0100	36	
LESSTHANOREQUAL	0010 0101	37	
ULESSTHAN	0010 0110	38	
ULESSTHANOREQUAL	0010 0111	39	
SWAP	0010 1000	40	
MULT	0010 1001	41	
LSHIFTRIGHT	0010 1010	42	
ASHIFTLLEFT	0010 1011	43	
ASHIFTRIGHT	0010 1100	44	
CALL	0010 1101	45	
EQ	0010 1110	46	
NEQ	0010 1111	47	
NEG	0011 0000	48	
SUB	0011 0001	49	
XOR	0011 0010	50	
LOADB	0011 0011	51	
STOREB	0011 0100	52	
DIV	0011 0101	53	
MOD	0011 0110	54	
EQBRANCH	0011 0111	55	
NEQBRANCH	0011 1000	56	
POPPCREL	0011 1001	57	
CONFIG	0011 1010	58	
PUSHPC (<i>a</i>)	0011 1011	59	
SYSCALL (<i>a</i>)	0011 1100	60	
PUSHSPADD	0011 1101	61	
HALFMULT	0011 1110	62	
CALLPCREL	0011 1111	63	

Instruction Mapping

➤	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0000	BRK	NA4	PUSHSP	NA3	POPPC	ADD	AND	OR	LOAD	NOT	FLIP	NOP	STORE	POPSP	NA2	NA
0001	ADDTOP	SHIFT	ADDSP	ADDSP	ADDSP	ADDSP	ADDSP	ADDSP	ADDSP	ADDSP	ADDSP	ADDSP	ADDSP	ADDSP	ADDSP	ADDSP
0010	?	N/A	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU
0011	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU
0100	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP
0101	POP	POPDOWN	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP
0110	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP
0111	DUP	DUPSTACKB	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP
1000	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM
1001	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM
1010	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM
1011	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM
1100	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM
1101	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM
1110	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM
1111	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM

Implemented Instructions

➤	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0000	BRK	NA4	PUSHSP	NA3	POPPC	ADD	AND	OR	LOAD	NOT	FLIP	NOP	STORE	POPSP	NA2	NA
0001	ADDSP	ADDSP	ADDSP	ADDSP	ADDSP	ADDSP	ADDSP	ADDSP	ADDSP	ADDSP	ADDSP	ADDSP	ADDSP	ADDSP	ADDSP	ADDSP
0010	?	N/A	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU
0011	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU	EMU
0100	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP
0101	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP	STORESP
0110	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP
0111	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP	LOADSP
1000	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM
1001	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM
1010	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM
1011	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM
1100	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM
1101	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM
1110	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM
1111	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM	IM

Instructions Specification

OPCODE	IM x																
MACHINE CODE	1 xxxxxxx																
IMPLEMENTED	YES																
EMULATED	NO																
SP ACTION	single PUSH																
DESCRIPTION	Pushes immediate value into TOS																
PSEUDOCODE	<pre> if (idim='0') { // no previous IM idim = 1; mem[sp+1] = stackB; stackB = stackA; stackA = {mem[sp][24:0], x[6:0]}; // sign extend sp = sp - 1; } else { // previous IM stackA = { stackA[31:7], x[6:0] }; } </pre>																
EQUIVALENT CODE	<pre> if (!idim) { push(x); } else { idim = 1; a = pop(); push(a<<7 + x); } </pre>																
INTERNAL LAYOUT	<p>PREVIOUS "IM" (idim='1'):</p> <table border="1"> <thead> <tr> <th>Before</th> <th>After</th> </tr> </thead> <tbody> <tr> <td>SP-1 → []</td> <td>SP-1 → []</td> </tr> <tr> <td>SP → [] ← A</td> <td>SP → [] ← A = {A[31:7], x}</td> </tr> <tr> <td>SP+1 → [] ← B</td> <td>SP+1 → [] → B</td> </tr> </tbody> </table> <p>ELSE (idim='0'):</p> <table border="1"> <thead> <tr> <th>Before</th> <th>After</th> </tr> </thead> <tbody> <tr> <td>SP-1 → []</td> <td>SP → [] ← A = sign_ext(x)</td> </tr> <tr> <td>SP → [] ← A</td> <td>SP+1 → [] → B</td> </tr> <tr> <td>SP+1 → [] ← B</td> <td>SP+2 → [B]</td> </tr> </tbody> </table>	Before	After	SP-1 → []	SP-1 → []	SP → [] ← A	SP → [] ← A = {A[31:7], x}	SP+1 → [] ← B	SP+1 → [] → B	Before	After	SP-1 → []	SP → [] ← A = sign_ext(x)	SP → [] ← A	SP+1 → [] → B	SP+1 → [] ← B	SP+2 → [B]
Before	After																
SP-1 → []	SP-1 → []																
SP → [] ← A	SP → [] ← A = {A[31:7], x}																
SP+1 → [] ← B	SP+1 → [] → B																
Before	After																
SP-1 → []	SP → [] ← A = sign_ext(x)																
SP → [] ← A	SP+1 → [] → B																
SP+1 → [] ← B	SP+2 → [B]																

OPCODE	EMULATE x								
MACHINE CODE	001 xxxxxx								
IMPLEMENTED	YES								
EMULATED	NO								
SP ACTION	PUSH								
DESCRIPTION	If the instruction is not implemented in hardware, this instruction will fired-up the microcode implementation of the function. 0<= x <= 31								
PSEUDOCODE	<pre> mem[sp+1] = stackB; // save cached stackB sp = sp - 1; // make room for push stackB = stackA; stackA = pc + 1; // return address pc = 32*x; // microcode at address 32*x fetch(); </pre>								
EQUIVALENT CODE	<i>call(32*x);</i>								
INTERNAL LAYOUT	<table border="0"> <thead> <tr> <th>Before</th><th>After</th></tr> </thead> <tbody> <tr> <td>SP-1 → []</td><td>SP → [] ← A = return_address</td></tr> <tr> <td>SP → [] ← A</td><td>SP+1 → [] ← B</td></tr> <tr> <td>SP+1 → [] ← B</td><td>SP+2 → [B]</td></tr> </tbody> </table>	Before	After	SP-1 → []	SP → [] ← A = return_address	SP → [] ← A	SP+1 → [] ← B	SP+1 → [] ← B	SP+2 → [B]
Before	After								
SP-1 → []	SP → [] ← A = return_address								
SP → [] ← A	SP+1 → [] ← B								
SP+1 → [] ← B	SP+2 → [B]								

OPCODE	STORESP x												
MACHINE CODE	010 xxxxxx												
IMPLEMENTED	YES												
EMULATED	NO												
SP ACTION	POP												
DESCRIPTION	Pop TOS and store it at mem[SP+x]												
PSEUDOCODE	<pre> (storeSP) mem[sp+x] = stackA; // NOTE: x is always unsigned stackA = stackB; sp = sp + 1; (storeSP2) stackB = mem[sp+1]; </pre>												
EQUIVALENT CODE	<pre> mem[SP+x] = TOS; pop(); </pre>												
INTERNAL LAYOUT	<table border="1"> <thead> <tr> <th>Before</th> <th>After</th> </tr> </thead> <tbody> <tr> <td>SP → [] ← A</td> <td>SP → [] ← A (= prev B)</td> </tr> <tr> <td>SP+1 → [] ← B</td> <td>SP+1 → [] → B (fetched)</td> </tr> <tr> <td>SP+2 → []</td> <td></td> </tr> <tr> <td>...</td> <td></td> </tr> <tr> <td>SP+x → [?]</td> <td>SP+x-1 → [A] (modified)</td> </tr> </tbody> </table>	Before	After	SP → [] ← A	SP → [] ← A (= prev B)	SP+1 → [] ← B	SP+1 → [] → B (fetched)	SP+2 → []		...		SP+x → [?]	SP+x-1 → [A] (modified)
Before	After												
SP → [] ← A	SP → [] ← A (= prev B)												
SP+1 → [] ← B	SP+1 → [] → B (fetched)												
SP+2 → []													
...													
SP+x → [?]	SP+x-1 → [A] (modified)												

OPCODE	LOADSP x												
MACHINE CODE	011xxxxx												
IMPLEMENTED	YES												
EMULATED	NO												
SP ACTION	PUSH												
DESCRIPTION	Push value at mem[SP+x] into stack												
PSEUDOCODE	<pre> (LoadSP) mem[sp+1] = stackB; // writeback cached stackB sp = sp - 1; // required for push (LoadSP2) read = mem[SP+x+1]; // fetch mem[SP+x] (LoadSP3) stackB = stackA; // stackB is now = old stackA stackA = read; // stackA = mem[SP+x] </pre>												
EQUIVALENT CODE	<pre> a = mem[sp+x]; push(a); </pre>												
INTERNAL LAYOUT	<table border="1"> <thead> <tr> <th>Before</th><th>After</th></tr> </thead> <tbody> <tr> <td>SP-1 → []</td><td>SP → [] → A (= v)</td></tr> <tr> <td>SP → [] ← A</td><td>SP+1 → [] ← B (= prev A)</td></tr> <tr> <td>SP+1 → [] ← B</td><td>SP+2 → []</td></tr> <tr> <td>...</td><td></td></tr> <tr> <td>SP+x → [v]</td><td>SP+x+1 → [v]</td></tr> </tbody> </table>	Before	After	SP-1 → []	SP → [] → A (= v)	SP → [] ← A	SP+1 → [] ← B (= prev A)	SP+1 → [] ← B	SP+2 → []	...		SP+x → [v]	SP+x+1 → [v]
Before	After												
SP-1 → []	SP → [] → A (= v)												
SP → [] ← A	SP+1 → [] ← B (= prev A)												
SP+1 → [] ← B	SP+2 → []												
...													
SP+x → [v]	SP+x+1 → [v]												

OPCODE	ADDSP x														
MACHINE CODE	0001xxxx														
IMPLEMENTED	YES														
EMULATED	YES														
SP ACTION	POP + PUSH														
DESCRIPTION	TOS = TOS + mem[SP+x]														
PSEUDO CODE	<pre>a = mem[SP+x]; stackA = stackA + a;</pre>														
EQUIVALENT CODE	<pre>a = mem[SP+x]; b = pop(); push (a+b);</pre>														
INTERNAL LAYOUT	<table border="0"> <thead> <tr> <th>Before</th><th>After</th></tr> </thead> <tbody> <tr> <td>-----</td><td>-----</td></tr> <tr> <td>SP-1 → []</td><td>SP-1 → []</td></tr> <tr> <td>SP → [] ← A</td><td>SP → [] ← A (+= mem[SP+x])</td></tr> <tr> <td>SP+1 → [] ← B</td><td>SP+1 → [] ← B</td></tr> <tr> <td>...</td><td></td></tr> <tr> <td>SP+x → []</td><td>SP+x → []</td></tr> </tbody> </table>	Before	After	-----	-----	SP-1 → []	SP-1 → []	SP → [] ← A	SP → [] ← A (+= mem[SP+x])	SP+1 → [] ← B	SP+1 → [] ← B	...		SP+x → []	SP+x → []
Before	After														
-----	-----														
SP-1 → []	SP-1 → []														
SP → [] ← A	SP → [] ← A (+= mem[SP+x])														
SP+1 → [] ← B	SP+1 → [] ← B														
...															
SP+x → []	SP+x → []														

OPCODE	BREAKPOINT
MACHINE CODE	00000000
DESCRIPTION	<i>call exception vector</i>

OPCODE	SHIFTLEFT
MACHINE CODE	00000001
DESCRIPTION	.

OPCODE	PUSHSP								
MACHINE CODE	00000010								
IMPLEMENTED	YES								
EMULATED	NO								
SP ACTION	PUSH								
DESCRIPTION	This instruction pushes the SP value into the stack								
PSEUDO CODE	<code>push(SP);</code>								
EQUIVALENT CODE	<pre> mem[SP+1] = stackB; stackB = stackA; stackA = SP; SP = SP - 1; </pre>								
INTERNAL LAYOUT	<table border="1"> <thead> <tr> <th>Before</th><th>After</th></tr> </thead> <tbody> <tr> <td>SP-1 → []</td><td>SP → [] ← A (= sp)</td></tr> <tr> <td>SP → [] ← A</td><td>SP+1 → [] → B (= old stackA)</td></tr> <tr> <td>SP+1 → [] ← B</td><td>SP+2 → [B]</td></tr> </tbody> </table>	Before	After	SP-1 → []	SP → [] ← A (= sp)	SP → [] ← A	SP+1 → [] → B (= old stackA)	SP+1 → [] ← B	SP+2 → [B]
Before	After								
SP-1 → []	SP → [] ← A (= sp)								
SP → [] ← A	SP+1 → [] → B (= old stackA)								
SP+1 → [] ← B	SP+2 → [B]								

OPCODE	POPINT
MACHINE CODE	00000011
DESCRIPTION	<pre> pc = mem[sp]; sp = sp + 1; fetch() ; decode() ; clear_interrupt_flag(); </pre>

OPCODE	POPPC								
MACHINE CODE	00000100								
IMPLEMENTED	YES								
EMULATED	NO								
SP ACTION	POP								
DESCRIPTION	Sets PC to value popped from stack.								
PSEUDO CODE	<pre> pc = mem[sp]; sp = sp + 1; (resynch) stackA = mem[SP]; stackB = mem[SP+1]; </pre>								
EQUIVALENT CODE	<code>pc = pop();</code>								
INTERNAL LAYOUT	<table border="1"> <thead> <tr> <th>Before</th><th>After</th></tr> </thead> <tbody> <tr> <td>SP → [] ← A</td><td>SP-1 → []</td></tr> <tr> <td>SP+1 → [] ← B</td><td>SP → [B] ← A</td></tr> <tr> <td>SP+2 → []</td><td>SP+1 → [] ← B</td></tr> </tbody> </table>	Before	After	SP → [] ← A	SP-1 → []	SP+1 → [] ← B	SP → [B] ← A	SP+2 → []	SP+1 → [] ← B
Before	After								
SP → [] ← A	SP-1 → []								
SP+1 → [] ← B	SP → [B] ← A								
SP+2 → []	SP+1 → [] ← B								

OPCODE	ADD								
MACHINE CODE	00000101								
IMPLEMENTED	YES								
EMULATED	NO								
SP ACTION	POP + POP + PUSH								
DESCRIPTION	This instruction pops two values from stack: X and Y. It then pushes back into the stack the value given A+B.								
PSEUDO CODE	<pre> a = pop(); b = pop(); b = b + a; push(b); </pre>								
EQUIVALENT CODE	<pre> mem[sp+1] = mem[sp+1] + mem[sp]; sp = sp + 1; </pre>								
INTERNAL LAYOUT	<table border="1"> <thead> <tr> <th>Before</th><th>After</th></tr> </thead> <tbody> <tr> <td>SP → [] ← A</td><td>SP-1 → []</td></tr> <tr> <td>SP+1 → [] ← B</td><td>SP → [] ← A</td></tr> <tr> <td>SP+2 → []</td><td>SP+1 → [] → B</td></tr> </tbody> </table>	Before	After	SP → [] ← A	SP-1 → []	SP+1 → [] ← B	SP → [] ← A	SP+2 → []	SP+1 → [] → B
Before	After								
SP → [] ← A	SP-1 → []								
SP+1 → [] ← B	SP → [] ← A								
SP+2 → []	SP+1 → [] → B								

OPCODE	AND
MACHINE CODE	00000110
DESCRIPTION	<code>mem[sp+1] = mem[sp+1] & mem[sp];</code> <code>sp = sp + 1;</code>

OPCODE	OR
MACHINE CODE	00000111
DESCRIPTION	<code>mem[sp+1] = mem[sp+1] mem[sp];</code> <code>sp = sp + 1;</code>

OPCODE	LOAD
MACHINE CODE	00001000
DESCRIPTION	<code>mem[sp] = mem[mem[sp]];</code>

OPCODE	NOT
MACHINE CODE	00001001
DESCRIPTION	<code>mem[sp] = not mem[sp];</code>

OPCODE	NOT
MACHINE CODE	00001001
DESCRIPTION	<code>mem[sp] = not mem[sp];</code>

OPCODE	FLIP
MACHINE CODE	00001010
DESCRIPTION	<code>mem[sp] = flip(mem[sp]);</code>

OPCODE	NOP
MACHINE CODE	00001011
DESCRIPTION	<code>no operation</code>

OPCODE	STORE
MACHINE CODE	00001100
DESCRIPTION	<code>mem[mem[sp]] = mem[sp+1]; sp = sp + 2;</code>

OPCODE	POPSP
MACHINE CODE	00001101
DESCRIPTION	<code>sp = mem[sp];</code>

OPCODE	COMPARE / IPSUM
MACHINE CODE	00001110
DESCRIPTION	<code>c = mem[sp]; s = mem[sp+1]; sum = 0; while (c-->0){ sum += halfword(mem[s],s); s += 2; }; sp = sp+1; mem[sp] = sum; (overwrites mem[0] & mem[4] words)</code>

OPCODE	SNCPY
MACHINE CODE	00001111
DESCRIPTION	<code>c = mem[sp]; d = mem[sp+1]; s = mem[sp+2]; while (*(char*)s != 0 && c>0){ *((char*)d++) =* ((char*)s++); c-- }; sp = sp+3; (overwrites mem[0] & mem[4] words)</code>

OPCODE	SNCPY
MACHINE CODE	00100000
DESCRIPTION	<pre> c = mem[sp]; d = mem[sp+1]; s = mem[sp+2]; while (c-->0) { mem[d++] = mem[s++]; } sp = sp+3; (overwrites mem[0] & mem[4] words) </pre>

OPCODE	WCPY
MACHINE CODE	00100001
DESCRIPTION	<pre> v = mem[sp]; c = mem[sp+1]; d = mem[sp+2]; while (c-->0) { mem[d++] = v; } sp = sp+3; (overwrites mem[0] & mem[4] words) </pre>

OPCODE	LOADH
MACHINE CODE	00100010
DESCRIPTION	<code>mem[sp] = halfword[mem[sp]];</code>

OPCODE	STOREH
MACHINE CODE	00100011
DESCRIPTION	<pre> halfword[mem[sp]] = (mem[sp+1] & 0xFFFF); sp = sp + 2; </pre>

OPCODE	LESSTHAN
MACHINE CODE	00100100
DESCRIPTION	<pre> if ((mem[sp]-mem[sp+1]) < 0){ mem[sp+1] = 1 } else { mem[sp+1] = 0; } sp = sp + 1; </pre>

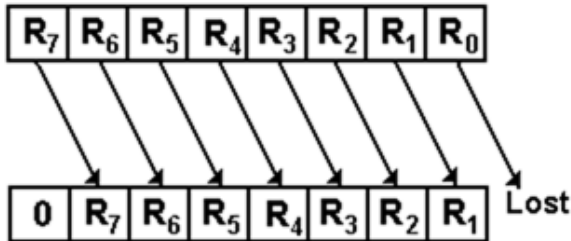
OPCODE	LESSTHANOREQUAL
MACHINE CODE	00100101
DESCRIPTION	<pre> if ((mem[sp]-mem[sp+1]) <= 0){ mem[sp+1] = 1 } else { mem[sp+1] = 0; } sp = sp + 1; </pre>

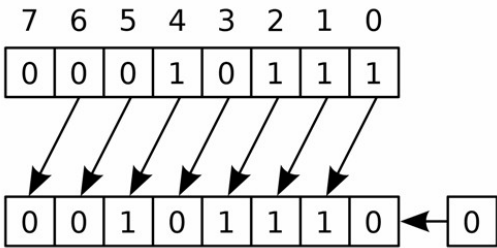
OPCODE	ULESSTHAN
MACHINE CODE	00100101
DESCRIPTION	<pre> if ((unsigned(mem[sp])-unsigned(mem[sp+1])) < 0){ mem[sp+1] = 1 } else { mem[sp+1] = 0; } sp = sp + 1; </pre>

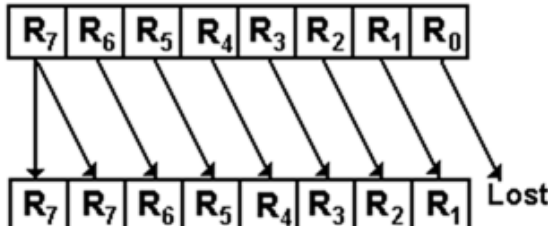
OPCODE	ULESSTHANORQUAL
MACHINE CODE	00100110
DESCRIPTION	<pre> if ((unsigned(mem[sp])-unsigned(mem[sp+1])) <= 0){ mem[sp+1] = 1 } else { mem[sp+1] = 0; } sp = sp + 1; </pre>

OPCODE	SWAP
MACHINE CODE	00101000
DESCRIPTION	.

OPCODE	MULT
MACHINE CODE	00101001
DESCRIPTION	<pre> mem[sp+1] = mem[sp+1] * mem[sp]; sp = sp + 1; </pre>

OPCODE	LSHIFTRIGHT								
MACHINE CODE	00101010								
IMPLEMENTED	YES								
EMULATED	YES								
SP ACTION	POP + POP + PUSH								
DESCRIPTION	<p>This instruction pops two values from stack: the shift parameter N and the shift variable V. It then pushes back into the stack the right logic shift value given by $V \gg N$. The figure below shows an example for only 8 bit (this CPU works on 32bit).</p> 								
PSEUDO CODE	<pre> a = pop(); // nr of shifts b = pop(); // value to shift b = (unsigned)b >> a; push(b); </pre>								
EQUIVALENT CODE	<pre> if (TOS!=0) { while (stackA) { stackA = stackA - 1; stackB = ('0', stackB[31:1]); } } stackA = stackB; stackB = mem[SP+2]; SP = SP + 1; </pre>								
INTERNAL LAYOUT	<table border="1"> <thead> <tr> <th>Before</th> <th>After</th> </tr> </thead> <tbody> <tr> <td>SP → [] ← A</td> <td>SP-1 → []</td> </tr> <tr> <td>SP+1 → [] ← B</td> <td>SP → [] ← A</td> </tr> <tr> <td>SP+2 → []</td> <td>SP+1 → [] → B</td> </tr> </tbody> </table>	Before	After	SP → [] ← A	SP-1 → []	SP+1 → [] ← B	SP → [] ← A	SP+2 → []	SP+1 → [] → B
Before	After								
SP → [] ← A	SP-1 → []								
SP+1 → [] ← B	SP → [] ← A								
SP+2 → []	SP+1 → [] → B								

OPCODE	ASHIFTLEFT						
MACHINE CODE	00101011						
IMPLEMENTED	YES						
EMULATED	YES						
SP ACTION	POP + POP + PUSH						
DESCRIPTION	<p>This instruction pops two values from stack: the shift parameter N and the shift variable V. It then pushes back into the stack the left arithmetic shift value given by $V \ll N$. The figure below shows an example for only 8 bit (this CPU works on 32bit).</p> <div style="text-align: center;">  </div>						
PSEUDO CODE	<pre> a = pop(); // nr of shifts b = pop(); // value to shift b = b << a; push(b); </pre>						
EQUIVALENT CODE	<pre> if (TOS!=0) { while (stackA) { stackA = stackA - 1; stackB <=< 1; } } stackA = stackB; stackB = mem[SP+2]; SP = SP + 1; </pre>						
INTERNAL LAYOUT	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left; padding: 5px;">Before</th> <th style="text-align: left; padding: 5px;">After</th> </tr> </thead> <tbody> <tr> <td style="padding: 5px;"> <div style="display: flex; justify-content: space-between;"> SP → [] ← A SP+1 → [] ← B </div> </td> <td style="padding: 5px;"> <div style="display: flex; justify-content: space-between;"> SP-1 → [] SP → [] ← A </div> </td> </tr> <tr> <td style="padding: 5px;"> <div style="display: flex; justify-content: space-between;"> SP+2 → [] </div> </td> <td style="padding: 5px;"> <div style="display: flex; justify-content: space-between;"> SP+1 → [] → B </div> </td> </tr> </tbody> </table>	Before	After	<div style="display: flex; justify-content: space-between;"> SP → [] ← A SP+1 → [] ← B </div>	<div style="display: flex; justify-content: space-between;"> SP-1 → [] SP → [] ← A </div>	<div style="display: flex; justify-content: space-between;"> SP+2 → [] </div>	<div style="display: flex; justify-content: space-between;"> SP+1 → [] → B </div>
Before	After						
<div style="display: flex; justify-content: space-between;"> SP → [] ← A SP+1 → [] ← B </div>	<div style="display: flex; justify-content: space-between;"> SP-1 → [] SP → [] ← A </div>						
<div style="display: flex; justify-content: space-between;"> SP+2 → [] </div>	<div style="display: flex; justify-content: space-between;"> SP+1 → [] → B </div>						

OPCODE	ASHIFTRIGHT								
MACHINE CODE	00101100								
IMPLEMENTED	YES								
EMULATED	YES								
SP ACTION	POP + POP + PUSH								
DESCRIPTION	<p>This instruction pops two values from stack: the shift parameter N and the shift variable V. It then pushes back into the stack the left arithmetic shift value given by $V \gg N$. The figure below shows an example for only 8 bit (this CPU works on 32bit).</p> 								
PSEUDO CODE	<pre> a = pop(); // nr of shifts b = pop(); // value to shift b = b >> a; push(b); </pre>								
EQUIVALENT CODE	<pre> if (TOS!=0) { while (stackA) { stackA = stackA - 1; stackB >>= 1; } } stackA = stackB; stackB = mem[SP+2]; SP = SP + 1; </pre>								
INTERNAL LAYOUT	<table border="1"> <thead> <tr> <th>Before</th><th>After</th></tr> </thead> <tbody> <tr> <td>SP → [] ← A</td><td>SP-1 → []</td></tr> <tr> <td>SP+1 → [] ← B</td><td>SP → [] ← A</td></tr> <tr> <td>SP+2 → []</td><td>SP+1 → [] → B</td></tr> </tbody> </table>	Before	After	SP → [] ← A	SP-1 → []	SP+1 → [] ← B	SP → [] ← A	SP+2 → []	SP+1 → [] → B
Before	After								
SP → [] ← A	SP-1 → []								
SP+1 → [] ← B	SP → [] ← A								
SP+2 → []	SP+1 → [] → B								

OPCODE	CALL								
MACHINE CODE	00101101								
IMPLEMENTED	YES								
EMULATED	YES								
SP ACTION	POP + PUSH								
DESCRIPTION	jumps to memory address TOP, while adjusting the stack with the return address pc+1								
PSEUDO CODE	<pre> x = stackA[L-1:0]; // (POP) L: mem address lines stackA = pc + 1; // (PUSH) return address pc = x; // jump to x </pre>								
EQUIVALENT CODE	<pre> x = pop(); call(x); // pushes PC+1 into stack </pre>								
INTERNAL LAYOUT	<table border="1"> <thead> <tr> <th>Before</th><th>After</th></tr> </thead> <tbody> <tr> <td>SP-1 → []</td><td>SP-1 → []</td></tr> <tr> <td>SP → [] ← A</td><td>SP → [] ← A (= pc+1)</td></tr> <tr> <td>SP+1 → [] ← B</td><td>SP+1 → [] ← B</td></tr> </tbody> </table>	Before	After	SP-1 → []	SP-1 → []	SP → [] ← A	SP → [] ← A (= pc+1)	SP+1 → [] ← B	SP+1 → [] ← B
Before	After								
SP-1 → []	SP-1 → []								
SP → [] ← A	SP → [] ← A (= pc+1)								
SP+1 → [] ← B	SP+1 → [] ← B								

OPCODE	EQ
MACHINE CODE	00101110
DESCRIPTION	<pre> if (mem[sp] == mem[sp+1]) { a = 1; } else { a = 0; } mem[sp+1] = a; sp = sp + 1 </pre>

OPCODE	NEQ
MACHINE CODE	00101111
DESCRIPTION	<pre> if (mem[sp] != mem[sp+1]) { a = 1; } else { a = 0; } mem[sp+1] = a; sp = sp + 1 </pre>

OPCODE	NEG
MACHINE CODE	00110000
DESCRIPTION	<code>mem[sp] = NOT(mem[sp]) + 1;</code>

OPCODE	SUB								
MACHINE CODE	00110001								
IMPLEMENTED	YES								
EMULATED	NO								
SP ACTION	POP + POP + PUSH								
DESCRIPTION	This instruction pops two values from stack: X and Y. It then pushes back into the stack the value given B-A.								
PSEUDO CODE	<pre> a = pop(); b = pop(); b = b - a; push(b); </pre>								
EQUIVALENT CODE	<pre> mem[sp+1] = mem[sp+1] - mem[sp]; sp = sp + 1; </pre>								
INTERNAL LAYOUT	<table border="1"> <thead> <tr> <th>Before</th><th>After</th></tr> </thead> <tbody> <tr> <td>SP → [] ← A</td><td>SP-1 → []</td></tr> <tr> <td>SP+1 → [] ← B</td><td>SP → [] ← A</td></tr> <tr> <td>SP+2 → []</td><td>SP+1 → [] → B</td></tr> </tbody> </table>	Before	After	SP → [] ← A	SP-1 → []	SP+1 → [] ← B	SP → [] ← A	SP+2 → []	SP+1 → [] → B
Before	After								
SP → [] ← A	SP-1 → []								
SP+1 → [] ← B	SP → [] ← A								
SP+2 → []	SP+1 → [] → B								

OPCODE	XOR
MACHINE CODE	00110010
DESCRIPTION	<pre> mem[sp+1] = mem[sp+1] XOR mem[sp]; sp = sp+1; </pre>

OPCODE	LOADB
MACHINE CODE	00110011
DESCRIPTION	<code>mem[sp] = byte[mem[sp]];</code>

OPCODE	STOREB
MACHINE CODE	00110100
DESCRIPTION	<code>byte[mem[sp]] = (mem[sp+1] & 0xFF); sp = sp + 2;</code>

OPCODE	DIV
MACHINE CODE	00110101
DESCRIPTION	.

OPCODE	MOD
MACHINE CODE	00110110
DESCRIPTION	.

OPCODE	EQBRANCH
MACHINE CODE	00110111
DESCRIPTION	<code>if (mem[sp+1] == 0) { pc = pc + mem[sp]; sp = sp + 2; }</code>

OPCODE	NEQBRANCH
MACHINE CODE	00111000
DESCRIPTION	<code>if (mem[sp+1] != 0) { pc = pc + mem[sp]; sp = sp + 2; }</code>

OPCODE	POPPCREL
MACHINE CODE	00111001
DESCRIPTION	<code>pc = pc + mem[sp]; sp = sp + 1;</code>

OPCODE	POPPCREL								
MACHINE CODE	00111001								
IMPLEMENTED	YES								
EMULATED	YES								
SP ACTION	POP								
DESCRIPTION	Adds to PC the value popped from stack.								
PSEUDO CODE	<pre> pc = pc + mem[sp]; sp = sp + 1; (resynch) stackA = mem[SP]; stackB = mem[SP+1]; </pre>								
EQUIVALENT CODE	<code>pc = pop();</code>								
INTERNAL LAYOUT	<table border="1"> <thead> <tr> <th>Before</th><th>After</th></tr> </thead> <tbody> <tr> <td>SP → [] ← A</td><td>SP-1 → []</td></tr> <tr> <td>SP+1 → [] ← B</td><td>SP → [B] ← A</td></tr> <tr> <td>SP+2 → []</td><td>SP+1 → [] ← B</td></tr> </tbody> </table>	Before	After	SP → [] ← A	SP-1 → []	SP+1 → [] ← B	SP → [B] ← A	SP+2 → []	SP+1 → [] ← B
Before	After								
SP → [] ← A	SP-1 → []								
SP+1 → [] ← B	SP → [B] ← A								
SP+2 → []	SP+1 → [] ← B								

OPCODE	CONFIG
MACHINE CODE	00111010
DESCRIPTION	.

OPCODE	PUSHPC
MACHINE CODE	00111011
DESCRIPTION	<pre> sp = sp - 1; mem[sp] = pc; </pre>

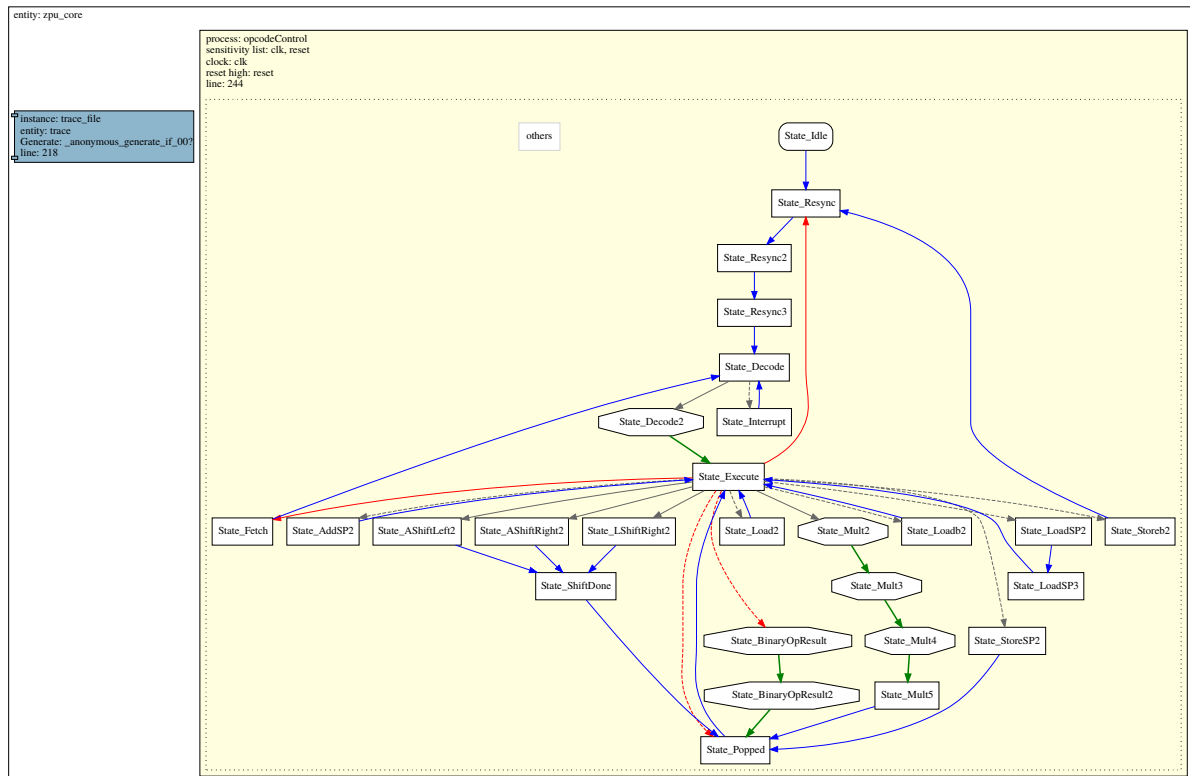
OPCODE	SYSCALL
MACHINE CODE	00111100
DESCRIPTION	.

OPCODE	PUSHSPADD
MACHINE CODE	00111101
DESCRIPTION	<code>mem[sp] = sp + (mem[sp] << 2)</code>

OPCODE	HALFMULT
MACHINE CODE	00111110
DESCRIPTION	<code>mem[sp+1] = 16bits(mem[sp]) * 16bits(mem[sp+1]);</code> <code>sp = sp + 1;</code>

OPCODE	CALLPCREL										
MACHINE CODE	00111110										
IMPLEMENTED	YES										
EMULATED	YES										
SP ACTION	POP + PUSH										
DESCRIPTION	jumps to memory address PC + TOP, while adjusting the stack with the return address pc+1										
PSEUDOCODE	<code>x = stackA[L-1:0]; // (POP) L: mem address lines</code> <code>stackA = pc + 1; // (PUSH) return address</code> <code>pc = pc + x; // jump to pc+x</code>										
EQUIVALENT CODE	<code>x = pop();</code> <code>call(pc+x); // pushes PC+1 into stack</code>										
INTERNAL LAYOUT	<table> <tr> <th>Before</th><th>After</th></tr> <tr> <td>-----+-----</td><td></td></tr> <tr> <td>SP-1 → []</td><td>SP-1 → []</td></tr> <tr> <td>SP → [] ← A</td><td>SP → [] ← A (= pc+1)</td></tr> <tr> <td>SP+1 → [] ← B</td><td>SP+1 → [] ← B</td></tr> </table>	Before	After	-----+-----		SP-1 → []	SP-1 → []	SP → [] ← A	SP → [] ← A (= pc+1)	SP+1 → [] ← B	SP+1 → [] ← B
Before	After										
-----+-----											
SP-1 → []	SP-1 → []										
SP → [] ← A	SP → [] ← A (= pc+1)										
SP+1 → [] ← B	SP+1 → [] ← B										

State Machine Transition Diagram



Legal Notice

```
-- Copyright 2004-2008 oharboe      - Øyvind Harboe - oyvind.harboe@zylin.com
-- Copyright 2008      alvieboy     - Álvaro Lopes  - alvieboy@alvie.com
-- Copyright 2015      gatekeeper   - Tiago Gasiba   - tiago.gasiba@gmail.com
--
-- The FreeBSD license
--
-- Redistribution and use in source and binary forms, with or without
-- modification, are permitted provided that the following conditions
-- are met:
--
-- 1. Redistributions of source code must retain the above copyright
--    notice, this list of conditions and the following disclaimer.
-- 2. Redistributions in binary form must reproduce the above
--    copyright notice, this list of conditions and the following
--    disclaimer in the documentation and/or other materials
--    provided with the distribution.
--
-- THIS SOFTWARE IS PROVIDED BY THE ZPU PROJECT "AS IS" AND ANY
-- EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
-- THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
-- PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
-- ZPU PROJECT OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT,
-- INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
-- (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
-- OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
-- HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
-- STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
-- ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
-- ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
--
-- The views and conclusions contained in the software and documentation
-- are those of the authors and should not be interpreted as representing
-- official policies, either expressed or implied, of the ZPU Project.
--
```