ZPU Reference Handbook

Zylin CPU

Tiago Gasiba

Table of Contents

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

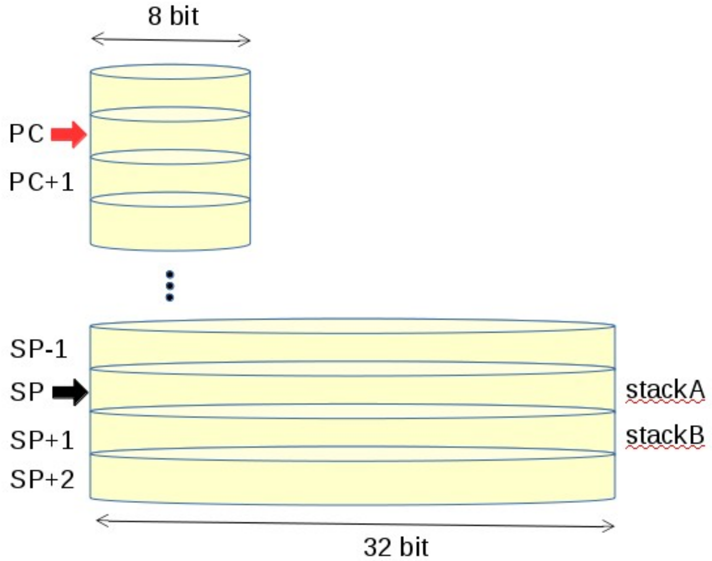
```
TOS     = Top Of Stack = SP
mem[SP] = valid data   = stackA

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$\overline{x}$ xxxx	
POP	0101 0000	Implemented using STORESP 0
POPDOWN	0101 0001	Implemented using STORESP 1
LOADSP x	011$\overline{x}$ xxxx	
DUP	0111 0000	Implemented using LOADSP 0
DUPSTACKB	0111 0001	Implemented using LOADSP 1
ADDSP x	0001 xxxx	
SHIFT	0001 0000	Implemented using ADDSP 0
ADDTOP	0001 0001	Implemented using ADDSP 1
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)

Mnemonic	*Opcode*	*Decimal*	*Description*
?	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	
ASHIFTLEFT	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 *(a)*	0011 1011	59	
SYSCALL *(a)*	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

IM

<table>
<tr><td>OPCODE</td><td>IM x</td></tr>
<tr><td>MACHINE CODE</td><td>1xxxxxxx</td></tr>
<tr><td>IMPLEMENTED</td><td>YES</td></tr>
<tr><td>EMULATED</td><td>NO</td></tr>
<tr><td>SP ACTION</td><td>single PUSH</td></tr>
<tr><td>DESCRIPTION</td><td>Pushes immediate value into TOS</td></tr>
<tr><td>PSEUDOCODE</td><td><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></td></tr>
<tr><td>EQUIVALENT CODE</td><td><pre>
if (!idim) {
      push(x);
} else {
      idim = 1;
      a   = pop();
      push(a<<7 + x);
}
</pre></td></tr>
<tr><td>INTERNAL LAYOUT</td><td><pre>
PREVIOUS “IM” (idim='1'):

     Before | After
 -------------------------+-----------------------
     SP-1 → [ ] | SP-1 → [ ]
     SP   → [ ] ← A | SP   → [ ]   ← A = {A[31:7],x}
     SP+1 → [ ] ← B | SP+1 → [ ]   → B

ELSE (idim='0'):

     Before | After
 -------------------------+-----------------------
     SP-1 → [ ] | SP   → [ ]   ← A = sign_ext(x)
     SP   → [ ] ← A | SP+1 → [ ]   → B
     SP+1 → [ ] ← B | SP+2 → [B]
</pre></td></tr>
</table>

EMULATE

OPCODE	**EMULATE x**
MACHINE CODE	**001xxxxx**
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	`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();`

EQUIVALENT CODE	*call*(32*x);

INTERNAL LAYOUT	`Before                    | After`
	`--------------------------+-----------------------`
	`SP-1 → [ ]                | SP   → [ ]    ←  A = return_address`
	`SP   → [ ] ← A            | SP+1 → [ ]    ←  B`
	`SP+1 → [ ] ← B            | SP+2 → [B]`

STORESP

OPCODE	**STORESP x**
MACHINE CODE	**010xxxxx**
IMPLEMENTED	**YES**
EMULATED	**NO**
SP ACTION	**POP**
DESCRIPTION	Pop TOS and store it at mem[SP+x]

PSEUDOCODE	`(storeSP)`
	`mem[sp+x] = stackA;  // NOTE: x is always unsigned`
	`stackA    = stackB;`
	`sp        = sp + 1;`
	`(storeSP2)`
	`stackB    = mem[sp+1];`

EQUIVALENT CODE	mem[SP+x] = TOS; *pop*();
INTERNAL LAYOUT	see below

```
    Before                   | After
-------------------------+-----------------------
    SP    → [ ]  ← A        |           → [ ]
    SP+1 → [ ]  ← B        | SP        → [ ]       ← A (= prev B)
    SP+2 → [ ]             | SP+1      → [ ]       → B (fetched )
    ...                    |
    SP+x → [?]             | SP+x-1 → [A]           (modified)
```

POP

OPCODE	**POP / STORESP 0**
MACHINE CODE	**01010000**
IMPLEMENTED	**YES**
EMULATED	**NO**
SP ACTION	**POP**
DESCRIPTION	Pops value from stack

PSEUDOCODE	`stackA    = stackB;`
	`sp        = sp + 1;`
	`stackB    = mem[sp+1];`

EQUIVALENT CODE	*pop*();
INTERNAL LAYOUT	see below

```
    Before                   | After
-------------------------+-----------------------
    SP    → [ ]  ← A        |           → [ ]
    SP+1 → [ ]  ← B        | SP        → [ ]       → A (= prev B)
    SP+2 → [ ]             | SP+1      → [ ]       → B (fetched )
```

POPDOWN

OPCODE	**POPDOWN / STORESP 1**
MACHINE CODE	**01010000**
IMPLEMENTED	**YES**
EMULATED	**NO**
SP ACTION	**POP**
DESCRIPTION	Pops two values from stack and pushes first value back to stack.
PSEUDOCODE	`sp = sp + 1;` `stackB = mem[sp+1];`

EQUIVALENT CODE	*a = pop*();
	pop();
	push(a);

INTERNAL LAYOUT	<pre> Before | After
	-------------------------+----------------------
	    SP → [ ] ← A | → [ ]
	    SP+1 → [ ] ← B | SP → [ ] ← A (= prev A)
	    SP+2 → [ ] | SP+1 → [ ] → B (fetched)</pre>

LOADSP

OPCODE	**LOADSP x**
MACHINE CODE	**011xxxxx**
IMPLEMENTED	**YES**
EMULATED	**NO**
SP ACTION	**PUSH**
DESCRIPTION	Push value at mem[SP+x] into stack

PSEUDOCODE	`(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]`

EQUIVALENT CODE	a = mem[sp+x]; *push*(a);
INTERNAL LAYOUT	` 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]`

DUP

<table>
<tr><td>OPCODE</td><td>DUP / LOADSP 0</td></tr>
<tr><td>MACHINE CODE</td><td>01110000</td></tr>
<tr><td>IMPLEMENTED</td><td>YES</td></tr>
<tr><td>EMULATED</td><td>NO</td></tr>
<tr><td>SP ACTION</td><td>PUSH</td></tr>
<tr><td>DESCRIPTION</td><td>Push TOS again into stack</td></tr>
<tr><td>PSEUDOCODE</td><td><pre>
mem[sp+1] = stackB;   // writeback cached stackB
sp   = sp – 1;   // required for push
stackB   = stackA;   // stackB is now = old stackA
</pre></td></tr>
<tr><td>EQUIVALENT CODE</td><td><pre>
a = pop();
push(a);
push(a);
</pre></td></tr>
<tr><td>INTERNAL LAYOUT</td><td><pre>
    Before | After
-------------------------+-----------------------
    SP-1 → [ ] | SP   → [ ]   → A (= prev A)
    SP   → [ ] ← A | SP+1   → [ ]   ← B (= prev A)
    SP+1 → [ ] ← B | SP+2   → [ ]
</pre></td></tr>
</table>

DUPSTACKB

OPCODE	**DUPSTACKB / LOADSP 1**
MACHINE CODE	**01110001**
IMPLEMENTED	**YES**
EMULATED	**NO**
SP ACTION	**PUSH**
DESCRIPTION	Push stackB again into stack

PSEUDOCODE	`A         = stackA;  // save old stackA`
	`B         = stackB;  // save old stackB`
	`stackA    = stackB;  // new stackA = old stackB`
	`stackB    = A;       // new stackB = old stackA`
	`mem[sp+1] = B;       // writeback cached (old) stackB`
	`sp        = sp – 1;  // required for push`

EQUIVALENT CODE	`a = pop();    // get stackA`
	`b = pop();    // get stackB`
	`push(b);      // save back stackB`
	`push(a);      // save back stackA`
	`push(b);      // duplicate stackB`

INTERNAL LAYOUT	`    Before                  │ After`
	`-------------------------+----------------------`
	`    SP-1 → [ ]              │ SP      → [ ]      → A (= prev A)`
	`    SP   → [ ]  ← A         │ SP+1    → [ ]      ← B (= prev A)`
	`    SP+1 → [ ]  ← B         │ SP+2    → [B]`

ADDSP

OPCODE	**ADDSP x**
MACHINE CODE	**0001xxxx**
IMPLEMENTED	**YES**
EMULATED	**YES**
SP ACTION	**POP + PUSH**
DESCRIPTION	TOS = TOS + mem[SP+x]
PSEUDO CODE	`a      = mem[SP+x];` `stackA = stackA + a;`

EQUIVALENT CODE	`a = mem[SP+x];`
	`b = pop();`
	`push (a+b);`

INTERNAL LAYOUT	See layout below

```
         Before                 | After
  ------------------------------+------------------------
         SP-1 → [ ]             | SP-1 → [ ]
         SP   → [ ] ← A         | SP   → [ ]      ←  A  ( += mem[SP+x] )
         SP+1 → [ ] ← B         | SP+1 → [ ]      ←  B
...
         SP+x → [ ]             | SP+x → [ ]
```

ADDTOP

OPCODE	**ADDTOP / ADDSP 1**
MACHINE CODE	**00010001**
IMPLEMENTED	**YES**
EMULATED	**YES**
SP ACTION	**POP + PUSH**
DESCRIPTION	TOS = TOS + mem[SP+1]
PSEUDO CODE	`stackA = stackA + stackB;`

EQUIVALENT CODE	`A = pop();   // get stackA`
	`B = pop();   // get stackB`
	`push(B);     // push back stackB`
	`push(A+B);   // push A+B`

INTERNAL LAYOUT	See layout below

```
         Before                 | After
  ------------------------------+------------------------
         SP-1 → [ ]             | SP-1 → [ ]
         SP   → [ ] ← A         | SP   → [ ]      ←  A  ( += stackB )
         SP+1 → [ ] ← B         | SP+1 → [ ]      ←  B
```

SHIFT

OPCODE	**SHIFT / ADDSP 0**
MACHINE CODE	**00010000**
IMPLEMENTED	**YES**
EMULATED	**YES**
SP ACTION	**POP + PUSH**
DESCRIPTION	TOS = 2*TOS
PSEUDO CODE	`stackA = stackA << 1;`

EQUIVALENT CODE	`A = pop();   // get stackA`
	`push(A<<1); // push 2*A`

INTERNAL LAYOUT	<pre> Before | After
	-------------------------+----------------------
	     SP-1 → [ ] | SP-1 → [ ]
	     SP → [ ] ← A | SP → [ ] ← A (= A<<1)
	     SP+1 → [ ] ← B | SP+1 → [ ] ← B</pre>

BREAKPOINT

OPCODE	**BREAK**
MACHINE CODE	**00000000**
IMPLEMENTED	**YES**
EMULATED	**NO**
SP ACTION	**None**
DESCRIPTION	Sets the break output line to '1' for one clock cycle.
PSEUDO CODE	
EQUIVALENT CODE	

INTERNAL LAYOUT	<pre> Before | After
	-------------------------+----------------------
	     SP-1 → [ ] | SP-1 → [ ]
	     SP → [ ] ← A | SP → [ ] ← A
	     SP+1 → [ ] ← B | SP+1 → [ ] ← B</pre>

SHIFTLEFT

OPCODE	**SHIFTLEFT**
MACHINE CODE	**00000001**
DESCRIPTION	.

PUSHSP

OPCODE	**PUSHSP**
MACHINE CODE	**00000010**
IMPLEMENTED	**YES**
EMULATED	**NO**
SP ACTION	**PUSH**
DESCRIPTION	**This instruction pushes the SP value into the stack**
PSEUDO CODE	push(SP);

EQUIVALENT CODE	mem[SP+1] = stackB;
	stackB = stackA;
	stackA = SP;
	SP = SP – 1;

INTERNAL LAYOUT	Before | After --------------------------+----------------------- SP-1 → [ ] | SP → [ ] ← A (= sp) SP → [ ] ← A | SP+1 → [ ] → B (= old stackA) SP+1 → [ ] ← B | SP+2 → [B]

POPINT

OPCODE	**POPINT**
MACHINE CODE	**00000011**

DESCRIPTION	pc = mem[sp];
	sp = sp + 1;
	fetch() ;
	decode() ;
	clear_interrupt_flag();

POPPC

OPCODE	**POPPC**
MACHINE CODE	**00000100**
IMPLEMENTED	**YES**
EMULATED	**NO**
SP ACTION	**POP**
DESCRIPTION	Sets PC to value popped from stack.

PSEUDO CODE	`pc     = mem[sp];`
	`sp     = sp + 1;`
	`(resynch)`
	`stackA = mem[SP];`
	`stackB = mem[SP+1];`

EQUIVALENT CODE	`pc = pop();`

INTERNAL LAYOUT	<pre> Before | After
	-------------------------+-----------------------
	    SP → [ ] ← A | SP-1 → [ ]
	    SP+1 → [ ] ← B | SP → [B] ← A
	    SP+2 → [ ] | SP+1 → [ ] ← B</pre>

ADD

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. tIt then pushes back into the stack the value given by A+B.

PSEUDO CODE	`a = pop();`
	`b = pop();`
	`b = b + a;`
	`push(b);`

EQUIVALENT CODE	`mem[sp+1] = mem[sp+1] + mem[sp];` `sp        = sp + 1;`

INTERNAL LAYOUT	<pre> Before | After
	-------------------------+-----------------------
	    SP → [ ] ← A | SP-1 → [ ]
	    SP+1 → [ ] ← B | SP → [ ] ← A
	    SP+2 → [ ] | SP+1 → [ ] → B</pre>

AND

OPCODE	**AND**
MACHINE CODE	**00000110**
IMPLEMENTED	**YES**
EMULATED	**NO**
SP ACTION	**POP + POP + PUSH**
DESCRIPTION	This instruction pops two values from stack: X and Y. tIt then pushes back into the stack the value given by A AND B.

PSEUDO CODE	`a = pop();`
	`b = pop();`
	`b = b AND a;`
	`push(b);`

EQUIVALENT CODE	`mem[sp+1] = mem[sp+1] AND mem[sp];` `sp        = sp + 1;`

INTERNAL LAYOUT	<pre> Before | After
	-------------------------+----------------------
	    SP → [ ] ← A | SP-1 → [ ]
	    SP+1 → [ ] ← B | SP → [ ] ← A
	    SP+2 → [ ] | SP+1 → [ ] → B</pre>

OR

OPCODE	**OR**
MACHINE CODE	**00000111**
IMPLEMENTED	**YES**
EMULATED	**NO**
SP ACTION	**POP + POP + PUSH**
DESCRIPTION	This instruction pops two values from stack: X and Y. tIt then pushes back into the stack the value given by A OR B.

PSEUDO CODE	`a = pop();`
	`b = pop();`
	`b = b OR a;`
	`push(b);`

EQUIVALENT CODE	`mem[sp+1] = mem[sp+1] OR mem[sp];` `sp        = sp + 1;`

INTERNAL LAYOUT	`    Before                   │ After`
	`-------------------------+-----------------------`
	`    SP   → [ ] ← A         │ SP-1 → [ ]`
	`    SP+1 → [ ] ← B         │ SP   → [ ]     ←  A`
	`    SP+2 → [ ]             │ SP+1 → [ ]     →  B`

LOAD

OPCODE	**LOAD**
MACHINE CODE	**00001000**
IMPLEMENTED	**YES**
EMULATED	**NO**
SP ACTION	**PUSH**
DESCRIPTION	Push value at mem[stackA] into stack

PSEUDOCODE	`mem[SP+1] = stackB;`
	`SP        = SP-1;`
	`stackB    = stackA;`
	`stackA    = mem[stackA];`

EQUIVALENT CODE	`a = pop();`
	`b = mem[a];`
	push`(b);`

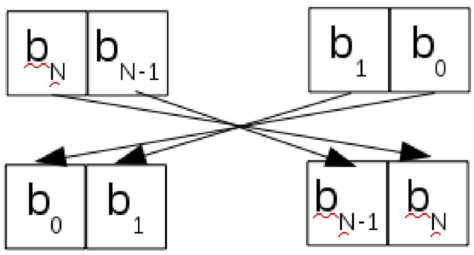
INTERNAL LAYOUT	<pre> Before | After
	-------------------------+------------------------
	    SP-1 → [ ] | SP → [ ] ← A (= mem[stackA])
	    SP → [ ] ← A | SP+1 → [ ] ← B (= prev stackA)
	    SP+1 → [ ] ← B | SP+2 → [B]</pre>

NOT

OPCODE	**NOT**
MACHINE CODE	**00001001**
IMPLEMENTED	**YES**
EMULATED	**YES**
SP ACTION	**POP + PUSH**
DESCRIPTION	This instruction pops one value from stack (X). It then pushes back into the stack the value NOT X;
PSEUDO CODE	`a = pop();` `push(~a);`
EQUIVALENT CODE	`stackA = NOT stackA;`

INTERNAL LAYOUT	<pre> Before | After
	-------------------------+------------------------
	    SP → [ ] ← A | SP → [ ] ← A
	    SP+1 → [ ] ← B | SP+1 → [ ] ← B
	    SP+2 → [ ] | SP+2 → [ ]</pre>

FLIP

OPCODE	**FLIP**
MACHINE CODE	**00001010**
IMPLEMENTED	**YES**
EMULATED	**YES**
SP ACTION	**POP + PUSH**
DESCRIPTION	This instruction pops one value from stack (X). It then pushes back into the stack the value flip(X). The flip function rearranges the bits of X such that bit{n} = bit{L-1-n}, where L is the word length in bits. 
PSEUDO CODE	`a = pop();` `push( flip(a) );`
EQUIVALENT CODE	`mem[sp] = flip( mem[sp] );`

INTERNAL LAYOUT	`    Before                   | After`
	`-------------------------+------------------------`
	`    SP+1 → [ ]           | SP-1 → [ ]`
	`    SP   → [ ] ← A       | SP   → [ ]      ←  A  (= flip stackA )`
	`    SP+2 → [ ] ← B       | SP+1 → [ ]      ←  B`

NOP

OPCODE	**NOP**
MACHINE CODE	**00001011**
IMPLEMENTED	**YES**
EMULATED	**NO**
SP ACTION	**none**
DESCRIPTION	No operation
PSEUDOCODE	
EQUIVALENT CODE	

INTERNAL LAYOUT	<pre> Before | After
	-------------------------+----------------------
	    SP-1 → [ ] | SP-1 → [ ]
	    SP → [ ] ← A | SP → [ ] ← A
	    SP+1 → [ ] ← B | SP+1 → [ ] ← B</pre>

STORE

OPCODE	**STORE**
MACHINE CODE	**00001100**
IMPLEMENTED	**YES**
EMULATED	**NO**
SP ACTION	**POP + POP**
DESCRIPTION	Pop memory address A and value B from stack. Write to memory at address A the value B.

PSEUDOCODE	`mem[stackA] = stackB;`
	`SP          = SP+2;`
	`resynch();   // reloads stackA and stackB`

EQUIVALENT CODE	`A      = pop();    // memory address`
	`B      = pop();    // value to write`
	`mem[A] = B;        // write value to memory`

INTERNAL LAYOUT	<pre> Before | After
	-------------------------+----------------------
	    SP → [ ] ← A | SP-2 → [ ]
	    SP+1 → [ ] ← B | SP-1 → [ ]
	    SP+2 → [ ] | SP → [ ] → A
	    SP+3 → [ ] | SP+1 → [ ] → B</pre>

POPSP

OPCODE	**POPSP**
MACHINE CODE	**00001101**
IMPLEMENTED	**YES**
EMULATED	**NO**
SP ACTION	**new stack frame**
DESCRIPTION	Load SP with TOS
PSEUDOCODE	`sp = stackA;` `resynch();`
EQUIVALENT CODE	SP = TOS;

INTERNAL LAYOUT	<pre> Before | After
	-------------------------+----------------------
	    SP → [ ] ← A |
	    SP+1 → [ ] ← B |
	                         ...
	                          | SP → [ ] → A
	                          | SP+1 → [ ] → B</pre>

COMPARE

OPCODE	**COMPARE / IPSUM**
MACHINE CODE	**00001110**

DESCRIPTION	<pre>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)</pre>

SNCPY

OPCODE	**SNCPY**
MACHINE CODE	**00001111**

DESCRIPTION	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)

SNCPY2

OPCODE	**SNCPY**
MACHINE CODE	**00100000**

DESCRIPTION	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)

WCPY

OPCODE	**WCPY**
MACHINE CODE	**00100001**

DESCRIPTION	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)

LOADH

OPCODE	**LOADH**
MACHINE CODE	**00100010**
DESCRIPTION	`mem[sp] = halfword[ mem[sp] ];`

STOREH

OPCODE	**STOREH**
MACHINE CODE	**00100011**
DESCRIPTION	`halfword[mem[sp]] = (mem[sp+1] & 0xFFFF);` `sp = sp + 2;`

LESSTHAN

OPCODE	**LESSTHAN**
MACHINE CODE	**00100100**
IMPLEMENTED	**YES**
EMULATED	**YES**
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 1 if A<B using signed comparison; otherwise it pushes back the value 0;

PSEUDO CODE	`a = pop();`
	`b = pop();`
	`if ((signed)a < (signed)b) {`
	`    push(1);`
	`} else {`
	`    push(0);`
	`}`

EQUIVALENT CODE	`if ( mem[sp] < mem[sp+1] ) {  // signed comparison`
	`    a = 1;`
	`} else {`
	`    a = 0;`
	`}`
	`mem[sp+1] = a;`
	`sp        = sp + 1`

INTERNAL LAYOUT	`    Before                    | After`
	`-------------------------+-----------------------`
	`    SP   → [ ] ← A         | SP-1 → [ ]`
	`    SP+1 → [ ] ← B         | SP   → [ ]     ←  A`
	`    SP+2 → [ ]             | SP+1 → [ ]     →  B`

LESSTHANOREQUAL

OPCODE	**LESSTHANOREQUAL**
MACHINE CODE	**00100101**
IMPLEMENTED	**YES**
EMULATED	**YES**
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 1 if A<=B using signed comparison; otherwise it pushes back the value 0;

PSEUDO CODE	`a = pop();`
	`b = pop();`
	`if ((signed)a <= (signed)b) {`
	`    push(1);`
	`} else {`
	`    push(0);`
	`}`

EQUIVALENT CODE	`if ( mem[sp] <= mem[sp+1] ) {  // signed comparison`
	`    a = 1;`
	`} else {`
	`    a = 0;`
	`}`
	`mem[sp+1] = a;`
	`sp        = sp + 1`

INTERNAL LAYOUT	` Before	After` `-------------------------+----------------------` ` SP → [ ] ← A	SP-1 → [ ]` ` SP+1 → [ ] ← B	SP → [ ] ← A` ` SP+2 → [ ]	SP+1 → [ ] → B`

ULESSTHAN

OPCODE	**ULESSTHAN**
MACHINE CODE	**00100101**
IMPLEMENTED	**YES**
EMULATED	**YES**
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 1 if A<B using unsigned comparison; otherwise it pushes back the value 0;

PSEUDO CODE	`a = pop();`
	`b = pop();`
	`if ((unsigned)a < (unsigned)b) {`
	`    push(1);`
	`} else {`
	`    push(0);`
	`}`

EQUIVALENT CODE	`if ( mem[sp] < mem[sp+1] ) {  // unsigned comparison`
	`    a = 1;`
	`} else {`
	`    a = 0;`
	`}`
	`mem[sp+1] = a;`
	`sp        = sp + 1`

INTERNAL LAYOUT	`    Before                 | After`
	`-------------------------+-----------------------`
	`    SP   → [ ] ← A       | SP-1 → [ ]`
	`    SP+1 → [ ] ← B       | SP   → [ ]    ←  A`
	`    SP+2 → [ ]           | SP+1 → [ ]    →  B`

ULESSTHANOREQUAL

OPCODE	**ULESSTHANOREQUAL**
MACHINE CODE	**00100110**
IMPLEMENTED	**YES**
EMULATED	**YES**
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 1 if A<=B using unsigned comparison; otherwise it pushes back the value 0;

PSEUDO CODE	`a = pop();`
	`b = pop();`
	`if ((unsigned)a <= (unsigned)b) {`
	`    push(1);`
	`} else {`
	`    push(0);`
	`}`

EQUIVALENT CODE	`if ( mem[sp] <= mem[sp+1] ) {  // unsigned comparison`
	`    a = 1;`
	`} else {`
	`    a = 0;`
	`}`
	`mem[sp+1] = a;`
	`sp        = sp + 1`

INTERNAL LAYOUT	`    Before                  | After`
	`-------------------------+-----------------------`
	`    SP   → [ ] ← A       | SP-1 → [ ]`
	`    SP+1 → [ ] ← B       | SP   → [ ]     ←  A`
	`    SP+2 → [ ]           | SP+1 → [ ]     →  B`

SWAP

OPCODE	**SWAP**
MACHINE CODE	**00101000**
DESCRIPTION	.

MULT

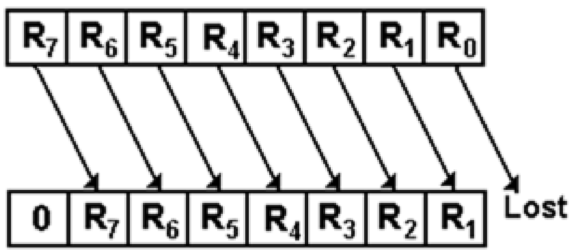
OPCODE	**MULT**
MACHINE CODE	**00101001**
IMPLEMENTED	**YES**
EMULATED	**YES**
SP ACTION	**POP + POP + PUSH**
DESCRIPTION	This instruction pops two values from stack: X and Y. It then pushes back into the stack the value X*Y. Note, only the lower order bits are pushed into the stack.

PSEUDO CODE	a = pop();
	b = pop();
	push(a*b);

EQUIVALENT CODE	`stackA     = stackA * stackB;`
	`mem[SP+1] = stackB;`
	`sp        = sp + 1;`

INTERNAL LAYOUT	<pre> Before │ After
	--------------------------+----------------------
	    SP → [ ] ← A │ SP-1 → [ ]
	    SP+1 → [ ] ← B │ SP → [ ] ← A
	    SP+2 → [ ] │ SP+1 → [ ] → B</pre>

LSHIFTRIGHT

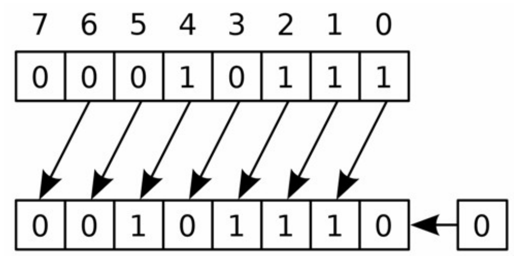
OPCODE	**LSHIFTRIGHT**
MACHINE CODE	**00101010**
IMPLEMENTED	**YES**
EMULATED	**YES**
SP ACTION	**POP + POP + PUSH**
DESCRIPTION	This instruction pops two values from stack: the shift parameter N and the shift variable V. It then pushes back into the stack the right logic shift value given by V>>N. The figure below shows an example for only 8 bit (this CPU works on 32bit). 

PSEUDO CODE	`a = pop();              // nr of shifts`
	`b = pop();              // value to shift`
	`b = (unsigned)b >> a;`
	`push(b);`

EQUIVALENT CODE	`if (TOS!=0) {`
	`    while (stackA) {`
	`        stackA = stackA – 1;`
	`        stackB = ('0', stackB[31:1]);`
	`    }`
	`}`
	`stackA = stackB;`
	`stackB = mem[SP+2];`
	`SP     = SP + 1;`

INTERNAL LAYOUT	`    Before                     | After`
	`-------------------------+-----------------------`
	`    SP   → [ ] ← A          | SP-1 → [ ]`
	`    SP+1 → [ ] ← B          | SP   → [ ]      ←  A`
	`    SP+2 → [ ]              | SP+1 → [ ]      →  B`

ASHIFTLEFT

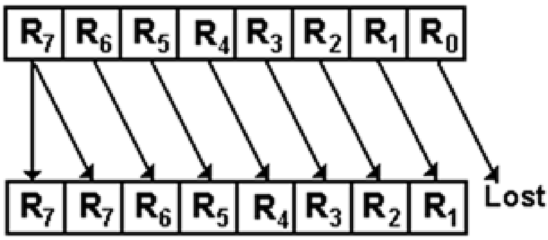
OPCODE	**ASHIFTLEFT**
MACHINE CODE	**00101011**
IMPLEMENTED	**YES**
EMULATED	**YES**
SP ACTION	**POP + POP + PUSH**
DESCRIPTION	This instruction pops two values from stack: the shift parameter N and the shift variable V. It then pushes back into the stack the left arithmetic shift value given by V<<N. The figure below shows an example for only 8 bit (this CPU works on 32bit). 

PSEUDO CODE	a = pop(); // nr of shifts
	b = pop(); // value to shift
	b = b << a;
	push(b);

EQUIVALENT CODE	if (TOS!=0) {
	    while (stackA) {
	        stackA = stackA – 1;
	        stackB <<= 1;
	    }
	}
	stackA = stackB;
	stackB = mem[SP+2];
	SP = SP + 1;

INTERNAL LAYOUT	Before | After --------------------------+------------------------ SP → [ ] ← A | SP-1 → [ ] SP+1 → [ ] ← B | SP → [ ] ← A SP+2 → [ ] | SP+1 → [ ] → B

ASHIFTRIGHT

OPCODE	**ASHIFTRIGHT**
MACHINE CODE	**00101100**
IMPLEMENTED	**YES**
EMULATED	**YES**
SP ACTION	**POP + POP + PUSH**
DESCRIPTION	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>>N. The figure below shows an example for only 8 bit (this CPU works on 32bit). 

PSEUDO CODE	a = pop(); // nr of shifts
	b = pop(); // value to shift
	b = b >> a;
	push(b);

EQUIVALENT CODE	if (TOS!=0) {
	    while (stackA) {
	        stackA = stackA – 1;
	        stackB >>= 1;
	    }
	}
	stackA = stackB;
	stackB = mem[SP+2];
	SP = SP + 1;

INTERNAL LAYOUT	Before | After -------------------------+----------------------- SP → [ ] ← A | SP-1 → [ ] SP+1 → [ ] ← B | SP → [ ] ← A SP+2 → [ ] | SP+1 → [ ] → B

CALL

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	`x      = stackA[L-1:0]; // (POP) L: mem address lines`
	`stackA = pc + 1;         // (PUSH) return address`
	`pc     = x;             // jump to x`

EQUIVALENT CODE	x = *pop*(); *call*(x); // pushes PC+1 into stack

INTERNAL LAYOUT	`    Before                 | After`
	`-------------------------+----------------------`
	`    SP-1 → [ ]             | SP-1   → [ ]`
	`    SP   → [ ]  ← A        | SP     → [ ]       ← A (= pc+1)`
	`    SP+1 → [ ]  ← B        | SP+1   → [ ]       ← B`

EQ

OPCODE	**EQ**
MACHINE CODE	**00101110**
IMPLEMENTED	**YES**
EMULATED	**YES**
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 1 if A==B; otherwise it pushes back the value 0;

PSEUDO CODE	`a = pop();`
	`b = pop();`
	`if (a==b) {`
	`    push(1);`
	`} else {`
	`    push(0);`
	`}`

EQUIVALENT CODE	`if ( mem[sp] == mem[sp+1] ) {`
	`    a = 1;`
	`} else {`
	`    a = 0;`
	`}`
	`mem[sp+1] = a;`
	`sp        = sp + 1`

INTERNAL LAYOUT	`    Before                   | After`
	`-------------------------+-----------------------`
	`    SP   → [ ] ← A        | SP-1 → [ ]`
	`    SP+1 → [ ] ← B        | SP   → [ ]      ←  A`
	`    SP+2 → [ ]            | SP+1 → [ ]      →  B`

NEQ

OPCODE	**NEQ**
MACHINE CODE	**00101111**
IMPLEMENTED	**YES**
EMULATED	**YES**
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 0 if A==B; otherwise it pushes back the value 1;

PSEUDO CODE	<pre>a = pop();
	b = pop();
	if (a!=b) {
	    push(1);
	} else {
	    push(0);
	}</pre>

EQUIVALENT CODE	<pre>if (mem[sp] != mem[sp+1]) {
	    a = 1;
	} else {
	    a = 0;
	}
	mem[sp+1] = a;
	sp = sp + 1</pre>

INTERNAL LAYOUT	<pre> Before | After
	-------------------------+-----------------------
	    SP → [ ] ← A | SP-1 → [ ]
	    SP+1 → [ ] ← B | SP → [ ] ← A
	    SP+2 → [ ] | SP+1 → [ ] → B</pre>

NEG

OPCODE	**NEG**
MACHINE CODE	**00110000**
IMPLEMENTED	**YES**
EMULATED	**YES**
SP ACTION	**POP + PUSH**
DESCRIPTION	This instruction pops one value from stack (X). It then pushes back into the stack the value -X;
PSEUDO CODE	a = pop(); push(-a);
EQUIVALENT CODE	`stackA = 1 + NOT stackA;`

INTERNAL LAYOUT	<pre> Before | After
	-------------------------+----------------------
	        SP → [ ] ← A | SP → [ ] ← A
	        SP+1 → [ ] ← B | SP+1 → [ ] ← B
	        SP+2 → [ ] | SP+2 → [ ]</pre>

SUB

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	a = pop();
	b = pop();
	b = b - a;
	push(b);

EQUIVALENT CODE	`mem[sp+1] = mem[sp+1] - mem[sp];` `sp        = sp + 1;`

INTERNAL LAYOUT	<pre> Before | After
	-------------------------+----------------------
	    SP → [ ] ← A | SP-1 → [ ]
	    SP+1 → [ ] ← B | SP → [ ] ← A
	    SP+2 → [ ] | SP+1 → [ ] → B</pre>

XOR

OPCODE	**XOR**
MACHINE CODE	**00110010**
IMPLEMENTED	**YES**
EMULATED	**YES**
SP ACTION	**POP + POP + PUSH**
DESCRIPTION	This instruction pops two values from stack: X and Y. tIt then pushes back into the stack the value given by A XOR B.

PSEUDO CODE	a = pop();
	b = pop();
	b = b XOR a;
	push(b);

EQUIVALENT CODE	mem[sp+1] = mem[sp+1] XOR mem[sp]; sp = sp + 1;

INTERNAL LAYOUT	<pre> Before	After
		--------------------------+-----------------------
		      SP → [ ] ← A
			SP-1 → [ ]
			      SP+1 → [ ] ← B
				SP → [ ] ← A
				      SP+2 → [ ]
					SP+1 → [ ] → B</pre>

LOADB

OPCODE	**LOADB**
MACHINE CODE	**00110011**
IMPLEMENTED	**YES**
EMULATED	**NO**
SP ACTION	**POP + PUSH**
DESCRIPTION	Pops the memory address A and pushes back the byte at mem[stackA] as a 32bit uint into stack
PSEUDOCODE	`mem[sp] = byte[ stackA ];`
EQUIVALENT CODE	a = pop(); b = BYTE{ mem[a] }; *push*((uint32)b);

INTERNAL LAYOUT	<pre> Before	After
		-------------------------+-----------------------
		    SP-1 → [ ]
			SP-1 → [ ]
			    SP → [ ] ← A
				SP → [ ] ← A (= byte[stackA])
				    SP+1 → [ ] ← B
					SP+1 → [ ] ← B</pre>

STOREB

OPCODE	**STOREB**
MACHINE CODE	**00110100**
IMPLEMENTED	**YES**
EMULATED	**NO**
SP ACTION	**POP + POP**
DESCRIPTION	Pops the memory address A and byte B (as 32bit) and writes back a byte with value B to memory at address A
PSEUDOCODE	`mem[stackA] = byte[ stackB ];`

EQUIVALENT CODE	`a = pop();`
	`b = pop();`
	`BYTE{ mem[A] } = BYTE{ b & 0x000000FF };`

INTERNAL LAYOUT	<pre> Before | After
	-------------------------+-----------------------
	     SP-1 → [ ] | SP-1 → [ ]
	     SP → [ ] ← A | SP → [ ] ← A (= byte[stackA])
	     SP+1 → [ ] ← B | SP+1 → [ ] ← B</pre>

DIV

OPCODE	**DIV**
MACHINE CODE	**00110101**
DESCRIPTION	.

MOD

OPCODE	**MOD**
MACHINE CODE	**00110110**
DESCRIPTION	.

EQBRANCH

OPCODE	**EQBRANCH**
MACHINE CODE	**00110111**

DESCRIPTION	`if ( mem[sp+1] == 0 ) {`
	`    pc = pc + mem[sp];`
	`    sp = sp + 2;`
	`}`

NEQBRANCH

OPCODE	**NEQBRANCH**
MACHINE CODE	**00111000**
IMPLEMENTED	**YES**
EMULATED	**YES**
SP ACTION	**POP + POP**
DESCRIPTION	This instruction pops two values from stack: X and Y; and adjusts the stack pointer accordingly. If Y!=0, then load PC with PC+X

PSEUDO CODE	`A  = pop();`
	`B  = pop();`
	`SP = SP + 2;`
	`if ( B!=0 ) {`
	`    PC = PC + A;`
	`}`

EQUIVALENT CODE	`if ( stackB != 0 ) {`
	`    pc = pc + stackA;`
	`}`
	`sp = sp + 2;`
	`resynch();`

INTERNAL LAYOUT	`    Before                 | After`
	`-------------------------+----------------------`
	`    SP   → [ ] ← A        | SP-2 → [ ]`
	`    SP+1 → [ ] ← B        | SP-1 → [ ]`
	`    SP+2 → [ ]            | SP   → [ ]      →  A`
	`    SP+2 → [ ]            | SP+1 → [ ]      →  B`

POPPCREL

OPCODE	**POPPCREL**
MACHINE CODE	**00111001**
IMPLEMENTED	**YES**
EMULATED	**YES**
SP ACTION	**POP**
DESCRIPTION	Adds to PC the value popped from stack.

PSEUDO CODE	`pc     = pc + mem[sp];`
	`sp     = sp + 1;`
	`(resynch)`
	`stackA = mem[SP];`
	`stackB = mem[SP+1];`

EQUIVALENT CODE	`pc = pop();`
INTERNAL LAYOUT	` Before	After` `-------------------------+-----------------------` ` SP → [ ] ← A	SP-1 → [ ]` ` SP+1 → [ ] ← B	SP → [B] ← A` ` SP+2 → [ ]	SP+1 → [ ] ← B`

CONFIG

OPCODE	**CONFIG**
MACHINE CODE	**00111010**
DESCRIPTION	.

PUSHPC

OPCODE	**PUSHPC**
MACHINE CODE	**00111011**
DESCRIPTION	`sp      = sp - 1;` `mem[sp] = pc;`

SYSCALL

OPCODE	**SYSCALL**
MACHINE CODE	**00111100**
DESCRIPTION	.

PUSHSPADD

OPCODE	**PUSHSPADD**
MACHINE CODE	**00111101**
IMPLEMENTED	**YES**
EMULATED	**YES**
SP ACTION	**POP + PUSH**
DESCRIPTION	This instruction replaces TOS with SP+4*A
PSEUDO CODE	A = pop(); push(SP+4*A);
EQUIVALENT CODE	`mem[sp] = sp + (stackA << 2)`

INTERNAL LAYOUT	<pre> Before | After
	-------------------------+-----------------------
	      SP-1 → [ ] | SP-1 → [ ]
	      SP → [ ] ← A | SP → [ ] ← A (=SP + old stackA)
	      SP+1 → [ ] ← B | SP+1 → [ ] ← B</pre>

HALFMULT

OPCODE	**HALFMULT**
MACHINE CODE	**00111110**
DESCRIPTION	`mem[sp+1] = 16bits(mem[sp]) * 16bits(mem[sp+1]);` `sp        = sp + 1;`

CALLPCREL

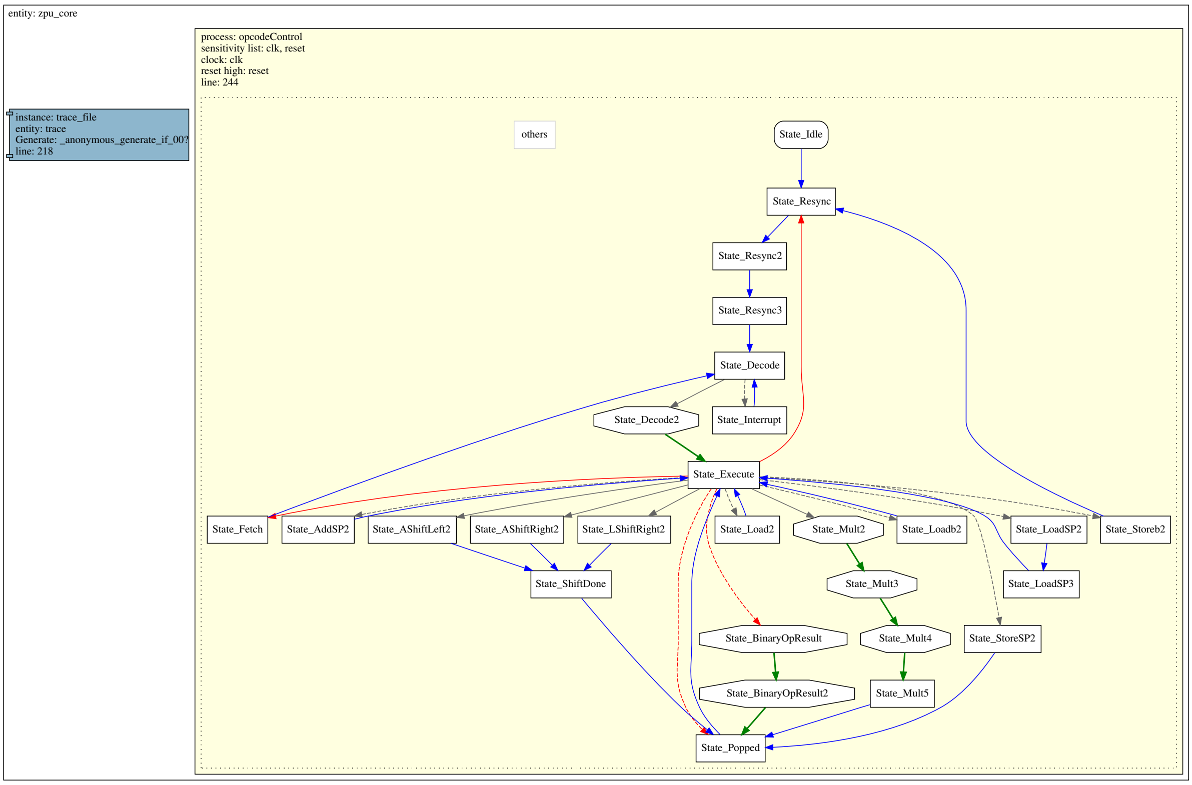
OPCODE	**CALLPCREL**
MACHINE CODE	**00111110**
IMPLEMENTED	**YES**
EMULATED	**YES**
SP ACTION	**POP + PUSH**
DESCRIPTION	jumps to memory address PC + TOP, while adjusting the stack with the return address pc+1

PSEUDOCODE	`x      = stackA[L-1:0]; // (POP) L: mem address lines`
	`stackA = pc + 1;        // (PUSH) return address`
	`pc     = pc + x;        // jump to pc+x`

EQUIVALENT CODE	x = *pop*(); *call*(pc+x); // pushes PC+1 into stack

INTERNAL LAYOUT	`    Before                 │ After`
	`-------------------------+----------------------`
	`    SP-1 → [ ]           │ SP-1   → [ ]`
	`    SP   → [ ]  ← A      │ SP     → [ ]      ← A (= pc+1)`
	`    SP+1 → [ ]  ← B      │ SP+1   → [ ]      ← B`

State Machine Transition Diagram



Legal Notice