ZPU Reference Handbook

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

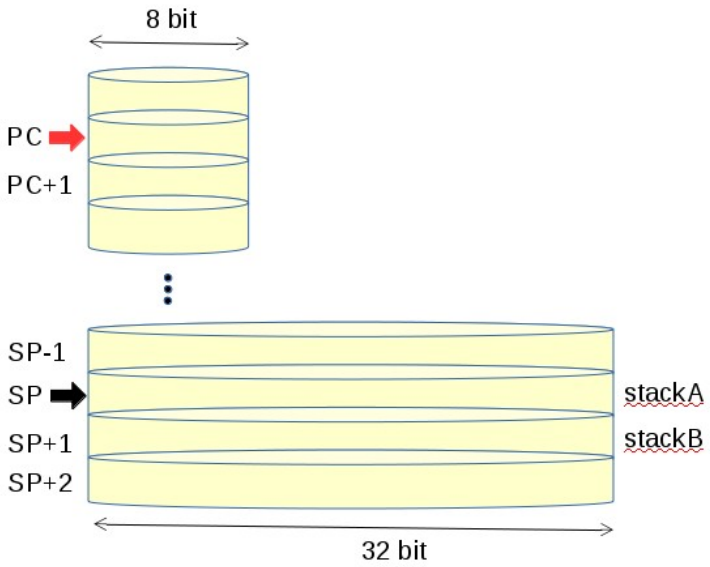
```
TOS      = Top Of Stack = SP
mem[SP] = valid data    = stackA

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	
LOADSP x	011$\overline{x}$ 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)

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

OPCODE	**IM x**
MACHINE CODE	**1xxxxxxx**
IMPLEMENTED	**YES**
EMULATED	**NO**
SP ACTION	***single* PUSH**
DESCRIPTION	Pushes immediate value into TOS

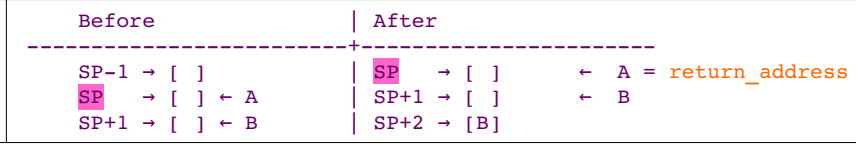
PSEUDOCODE	`if (idim='0') {  // no previous IM`
	`  idim = 1;`
	`  mem[sp+1] = stackB;`
	`  stackB = stackA;`
	`  stackA = {mem[sp][24:0], x[6:0]}; // sign extend`
	`  sp = sp – 1;`
	`} else {  // previous IM`
	`  stackA = { stackA[31:7], x[6:0] };`
	`}`

EQUIVALENT CODE	`if (!idim) {`
	`  push(x);`
	`} else {`
	`  idim = 1;`
	`  a = pop();`
	`  push( a<<7 + x );`
	`}`

INTERNAL LAYOUT	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]`

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	

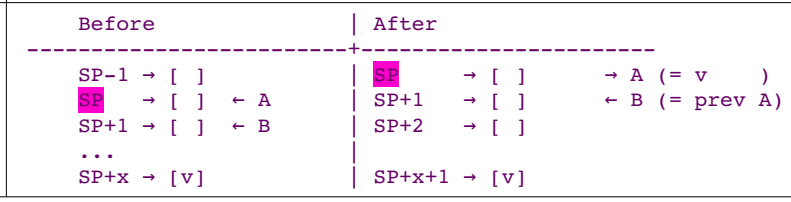
OPCODE	**STORESP x**
MACHINE CODE	**010xxxxx**
IMPLEMENTED	**YES**
EMULATED	**NO**
SP ACTION	**POP**
DESCRIPTION	Pop TOS and store it at mem[SP+x]

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

OPCODE	**ADDSP x**
MACHINE CODE	**0001xxxx**
DESCRIPTION	mem[sp] = mem[sp] + mem[sp+x<<2];

OPCODE	**BREAKPOINT**
MACHINE CODE	**00000000**
DESCRIPTION	call exception vector

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

OPCODE	**PUSHSP**
MACHINE CODE	**00000010**
DESCRIPTION	`mem[sp-1] = sp;` `sp = sp – 1;`

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

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

OPCODE	**POPPC**
MACHINE CODE	**00000100**
DESCRIPTION	`pc = mem[sp];` `sp = sp + 1;`

OPCODE	**ADD**
MACHINE CODE	**00000101**
DESCRIPTION	`mem[sp+1] = mem[sp+1] + mem[sp];` `sp = sp + 1;`

OPCODE	**AND**
MACHINE CODE	**00000110**
DESCRIPTION	`mem[sp+1] = mem[sp+1] & mem[sp];` `sp = sp + 1;`

OPCODE	**OR**
MACHINE CODE	**00000111**
DESCRIPTION	`mem[sp+1] = mem[sp+1] | mem[sp];` `sp = sp + 1;`

OPCODE	**LOAD**
MACHINE CODE	**00001000**
DESCRIPTION	`mem[sp] = mem[ mem[sp] ];`

OPCODE	**NOT**
MACHINE CODE	**00001001**
DESCRIPTION	`mem[sp] = not mem[sp];`

OPCODE	**NOT**
MACHINE CODE	**00001001**
DESCRIPTION	`mem[sp] = not mem[sp];`

OPCODE	**FLIP**
MACHINE CODE	**00001010**
DESCRIPTION	`mem[sp] = flip( mem[sp] );`

OPCODE	**NOP**
MACHINE CODE	**00001011**
DESCRIPTION	`no operation`

OPCODE	**STORE**
MACHINE CODE	**00001100**
DESCRIPTION	`mem[mem[sp]] = mem[sp+1];` `sp = sp + 2;`

OPCODE	**POPSP**
MACHINE CODE	**00001101**
DESCRIPTION	`sp = mem[sp];`

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

DESCRIPTION	`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)`

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)`

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)`

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)`

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

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

OPCODE	**LESSTHAN**
MACHINE CODE	**00100100**

DESCRIPTION	`if ( (mem[sp]-mem[sp+1]) < 0 ){`
	`  mem[sp+1] = 1`
	`} else {`
	`  mem[sp+1] = 0;`
	`}`
	`sp = sp + 1;`

OPCODE	**LESSTHANOREQUAL**
MACHINE CODE	**00100101**

DESCRIPTION	`if ( (mem[sp]-mem[sp+1]) <= 0 ){`
	`  mem[sp+1] = 1`
	`} else {`
	`  mem[sp+1] = 0;`
	`}`
	`sp = sp + 1;`

OPCODE	**ULESSTHAN**
MACHINE CODE	**00100101**

DESCRIPTION	`if ( (unsigned(mem[sp])-unsigned(mem[sp+1])) < 0 ){`
	`  mem[sp+1] = 1`
	`} else {`
	`  mem[sp+1] = 0;`
	`}`
	`sp = sp + 1;`

OPCODE	**ULESSTHANORQUAL**
MACHINE CODE	**00100110**

DESCRIPTION	`if ( (unsigned(mem[sp])-unsigned(mem[sp+1])) <= 0 ){`
	`  mem[sp+1] = 1`
	`} else {`
	`  mem[sp+1] = 0;`
	`}`
	`sp = sp + 1;`

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

OPCODE	**MULT**
MACHINE CODE	**00101001**
DESCRIPTION	`mem[sp+1] = mem[sp+1] * mem[sp];` `sp  = sp + 1;`

OPCODE	**LSHIFTRIGHT**
MACHINE CODE	**00101010**
DESCRIPTION	`mem[sp+1] = mem[sp+1] >> (mem[sp] & 0x1f);` `sp  = sp + 1;`

OPCODE	**ASHIFTLEFT**
MACHINE CODE	**00101011**
DESCRIPTION	`mem[sp+1] = mem[sp+1] << (mem[sp] & 0x1f);` `sp  = sp + 1;`

OPCODE	**ASHIFTRIGHT**
MACHINE CODE	**00101100**
DESCRIPTION	`mem[sp+1] = mem[sp+1] signed>> (mem[sp] & 0x1f);` `sp = sp + 1;`

OPCODE	**CALL**
MACHINE CODE	**00101101**

DESCRIPTION	`a = mem[sp];`
	`mem[sp] = pc + 1;`
	`pc = a;`

OPCODE	**EQ**
MACHINE CODE	**00101110**

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

OPCODE	**NEQ**
MACHINE CODE	**00101111**

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

OPCODE	**NEG**
MACHINE CODE	**00110000**
DESCRIPTION	`mem[sp] = NOT(mem[sp]) + 1;`

OPCODE	**SUB**
MACHINE CODE	**00110001**
DESCRIPTION	`mem[sp+1] = mem[sp+1] – mem[sp];` `sp = sp+1;`

OPCODE	**XOR**
MACHINE CODE	**00110010**
DESCRIPTION	`mem[sp+1] = mem[sp+1] XOR mem[sp];` `sp = sp+1;`

OPCODE	**LOADB**
MACHINE CODE	**00110011**
DESCRIPTION	`mem[sp] = byte[ mem[sp] ];`

OPCODE	**STOREB**
MACHINE CODE	**00110100**
DESCRIPTION	`byte[mem[sp]] = (mem[sp+1] & 0xFF);` `sp = sp + 2;`

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

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

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

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

OPCODE	**NEQBRANCH**
MACHINE CODE	**00111000**

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

OPCODE	**POPPCREL**
MACHINE CODE	**00111001**
DESCRIPTION	`pc = pc + mem[sp];` `sp = sp + 1;`

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

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

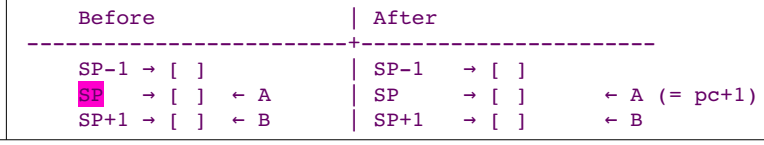
OPCODE	**SYSCALL**
MACHINE CODE	**00111100**
DESCRIPTION	.

OPCODE	**PUSHSPADD**
MACHINE CODE	**00111101**
DESCRIPTION	`mem[sp] = sp + (mem[sp] << 2)`

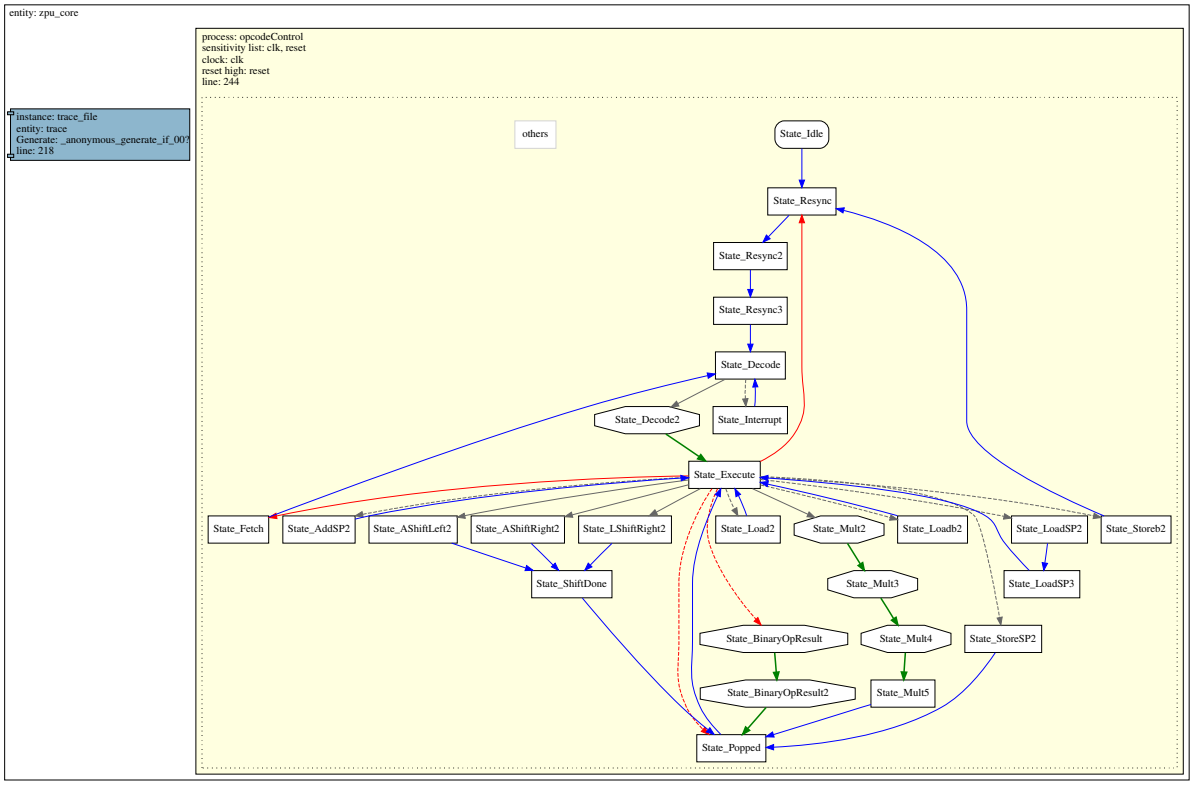
OPCODE	**HALFMULT**
MACHINE CODE	**00111110**
DESCRIPTION	`mem[sp+1] = 16bits(mem[sp]) * 16bits(mem[sp+1]);` `sp  = sp + 1;`

OPCODE	**CALLPCREL**
MACHINE CODE	**00111110**
IMPLEMENTED	**YES**
EMULATED	**YES**
SP ACTION	**POP and PUSH**
DESCRIPTION	Push value at mem[SP+x] into stack

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	

State Machine Transition Diagram



Legal Notice