The Sux Instruction Set Manual

mr b0nk 500
`b0nk@b0nk.xyz`

Fraizeraust
`fraizeraust99@gmail.com`

January 23, 2020

Table of Contents

1. Introduction

Sux (Pronounced "sucks") is a new, 64-bit CISC instruction-set architecture, (ISA) that was designed to be very simple, and easy to understand. Our goals in creating Sux include:

- A 100% totally free (as in freedom) ISA that both respects your freedoms, and does not allow anyone to make proprietary versions of it.
- An ISA that is suitable for implementing in hardware, not just emulation, or binary translation.
- An ISA that is very simple, both at the assembly language level, and at the hardware level.
- An ISA that can be implemented with as low a transistor count as possible, and therefore, very cheap to produce.
- An ISA that can actually be understood by any programmer, and is actually possible to write assembly language programs in.
- An ISA that is very memory efficient.
- An ISA that is extremely fast, and allows for an insane number of cores.
- An ISA that does not require uneeded bloat, just to make it "faster".
- An ISA that a hobbiest can use.
- An ISA that is easy to design a computer architecture arround.

The Sux ISA does contain documentation for a reference platform, but the main details of the ISA are not specific to the reference platform.

We called the Instruction Set, Sux because it is meant to be very suckless, or suxless, although, that is not what Sux is short for.

It was also named after a preposed x86 based CPU, which would have been called, Sux86less.

It was never made because we deemed x86, too bloated for our purposes.

It is also meant to be a joke on RISC-V, in the sense that, like RISC-V, Sux is small, and simple, but unlike RISC-V, Sux is more hobbiest oriented, rather than being more academic oriented.

1.1. Sux Hardware Platform Terminology

A Sux hardware platform can contain one, or more Sux compatible cores along with other supporting cores including, non-Sux compatible cores, MMUs, device controllers, and much more.

A component is dubbed a *core* if it is a turing complete CPU. A Sux compatible core might support multiple threads, or diet cores, and are used in the same way as a standard core.

A Sux core might have have extra instruction set extensions, or a *coprocessor*. We use the term *coprocessor* to refer to any type of external device that offloads some processing from the main core.

The system-level organization of a Sux hardware platform can range from a single-core microcontroller, to a 128 core, 1024 thread CPU, or microcontroller, to a multi-CPU server node. But also even a system-on-a-chip configuration.

We'll be using both *$* to denote a hexadecimal value, and *%* to denote a binary value.

1.2. Sux Interrupt Vectors

The way execution is started in a Sux core, is very similar to older 8 bit ISAs, in that it uses vectors to denote where it should start executing code for some type of interrupt.

By default, if there are multiple Sux cores, and each core has multiple threads, it will start the first thread, of the first core, and then read one of those vectors.

Figure 1 shows the layout of the vectors for a Sux core.

Address	**Name**
$FF50-$FF57	Thread Vector
$FF58-$FF5F	
$FF60-$FF67	
$FF60-$FF6F	
$FF70-$FF77	
$FF70-$FF7F	
$FF80-$FF87	
$FF90-$FF97	µDCROM Vector
$FFA0-FFA7	IRQ Vector
$FFC0-$FFC7	Reset Vector
$FFE0-FFE7	BRK Vector

Figure 1: The vectors of Sux.

All Sux threads can have their execution be suspended by a wait-for-interrupt, or `WAI` instuction.

1.3. Sux ISA Overview

The Sux ISA is defined as a base integer ISA, which is required to be present in any implementation, plus any optional extensions to the base ISA. The base ISA is very similar to MOS Technology's 6502 ISA, except that it has a 64 bit data bus, 64 bit address bus, supports multiple threads, multiple cores, and much more.

Sux has been designed to make using external hardware, very easy to implement. Sux has also been designed to make customizing, very easy, by having a Microdecode ROM, or µDCROM for short. The µDCROM is covered in more detail later on.

Beyond the base ISA, we deem it unnecessary to add any extra opcodes, or insturctions to it, and as such, will be unchanged, from this point on. There will still be other extensions, but those will not be accessed in the manner as the base ISA.

1.4. Memory

A Sux thread has a byte-addressable address space of 2^{64} bytes for all memory accesses. A *word* of memory is defined as 16 bits (2 bytes), a *doubleword* of memory is defined as 32 bits (4 bytes), and a *quadword* of memory is defined as 64 bits (8 bytes). The address space is circular, so that the byte at address 2^{64}-1 is adjacent to the byte at address zero. Accordingly, memory address computations done by the hardware ignore overflow, and instead wrap arround modulo 2^{64}.

The mapping of hardware resources, is determined by the architecture of the hardware. These addresses may either (a) be open bus, (b) contain main memory, (c) be vacant, or (d) contain one, or more I/O devices. Reading, and writing to I/O devices may have visible side effects, but accessing main memory cannot. While you could have an architecture that only accesses I/O, it is usually expected that some of the address space will be main memory.

When a Sux platform has multiple threads, unless an MMU is used, the entire address space must be shared with all threads.

Executing each Sux machine instruction, most of the time requires one memory access, but sometimes it might require two memory accesses. These instructions are divided into multiple addressing modes.

2. The Sux Base Instruction Set, Version 1.0

This chapter describes version 1.0 of the Sux base instruction set.

2.1. Programmers' Model for the Base ISA

Figure 2 Shows the state for the base ISA. The base ISA only has four registers, each of which are 64 bits in size, each register has a special property, but can be still be used like general purpose registers.

The accumulator, or A is the main register of the base ISA, and is used for all ALU based instructions. the B register is a secondary ALU register, and can be used in place of a memory address for all ALU instructions, the X register is a register that can be used for indexing, but has a special property, as it is the only register that can set the stack pointer, and finally, the Y register, like the X register can be used for indexing, both normal, and pointer based.

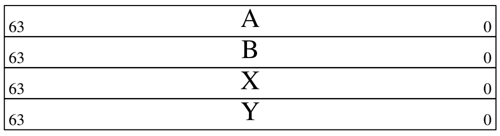

Figure 2: Sux base register state.

2.2. Base Instruction Format

The base Sux ISA has variable length instructions ranging from a single byte, to as many as 10 bytes, and are always byte aligned. The base ISA is also little endian, with no possiblity for bi-endian setups.

The reasoning for why we chose little endian, was because it is much simpler to work with, plus, it is very easy to understand, and is very intuitive.

The high code density is achived by using a prefix byte, which is not part of the instruction, but rather tells the CPU extra information like, how many bytes to read/write, whether it should use 8/16 bit addresses, or 32/64 bit addresses, or what instruction set extension it should use.

The base ISA is made up of 198 total opcodes, comprised of 93 instructions. All of the opcodes are one byte in length. The max operand count for the base ISA is one. These might look like typos, but all of them were done on purpose, both in order to make the base ISA easier to understand, and to keep the opcode length at one byte.

The instruction formatting for the Sux base ISA is shown in Figure 2.1.

opr[63:32]	opr[31:16]	opr[15:8]	opr[7:0]	opcode	prefix
71 40	39 24	23 16	15 8	7 0	7 0

Figure 2.1: Sux base instruction format. There are subfields for the operand because the prefix byte can specify the size of the operand, from 8 bits, all the way up to 64 bits.

The processor status register, is a register that contains flags for use with conditional branches, Figure 2.2 shows the layout of these flags, the processor status register is 64 bits in size, but only eight bits are used, this is so that we can have upto eight sets of these flags, and make conditional branching with threads, much easier.

The Carry flag, or C denotes when a register has carried over. The Zero flag, or Z denotes when a register is zero. The Interrupt flag, or I is used to disable/enable maskable interrupts, when set, it disables interrupts, when cleared, it enables interrupts. The Stack protection flag, or S is used to prevent stack overflows, and stack underflows, by not incrementing, or decrementing if the stack pointer is at it’s lowest value, or it’s highest value. The Overflow flag, or V denotes whenever an overflow of a register occurs. And the Negative flag, or N is used to denote a negative value, and allows the use of signed integers.

N	V			S	I	Z	C
7	6			3	2	1	0

Figure 2.2: The flags of the Processor status register.

2.3. Base Instruction Opcodes

This section describes each of the instructions, for the Sux base ISA, and provides a table, listing the opcodes for every instruction.

Table 1. Sux base opcode table

	x0	**x1**	**x2**	**x3**	**x4**	**x5**	**x6**	**x7**	**x8**	**x9**	**xA**	**xB**	**xC**	**xD**	**xE**	**xF**
0x	CPS	ADC #	AAB	ADC a	JMP ind	ADC zm	PHB		PHP	LDA #	LDY #	LDX #	TAB		LDB #	
1x	JMP a	SBC #	SAB	SBC a	JMP inx	SBC zm	PLB		PLP	LDA a	LDY a	LDX a	TBA		LDB a	
2x	JSR	AND #	ABA	AND a	JMP iny	AND zm	CPB #		STT	STA a	STY a	STX a	TAY		STB a	
3x	BPO	ORA #	OAB	ORA a	JSR ind	ORA zm	CPB a		SEI	LDA zm	LDY zm	LDX zm	TYA		LDB zm	
4x	BNG	XOR #	XAB	XOR a	JSR inx	XOR zm	CPB zm		CLI	STA zm	STY zm	STX zm	TAX		STB zm	
5x	BCS	LSL #	LLB	LSL a	JSR iny	LSL zm	CPB ind		SEC	LDA zmx	LDY zmx	LDX zmy	TXA		LDB zmx	
6x	BCC	LSR #	LRB	LSR a	BPO zm	LSR zm	CPB inx		CLC	STA zmx	STY zmx	STX zmy	TYX		STB zmx	
7x	BEQ	ROL #	RLB	ROL a	BNG zm	ROL zm	CPB iny		SSP	LDA zmy	LDY ind	LDX ind	TXY		LDB zmy	
8x	BNE	ROR #	RRB	ROR a	BCS zm	ROR zm	INY		CSP	STA zmy	STY ind	STX ind	TSX		STB zmy	
9x	BVS	MUL #	MAB	MUL a	BCC zm	MUL zm	DEY		SEV	LDA ind	LDY inx	LDX iny	TXS		LDB ind	
Ax	BVC	DIV #	DAB	DIV a	BEQ zm	DIV zm	INX		CLV	STA ind	STY inx	STX iny	PHY		STB ind	
Bx	RTS	CMP #	CAB	CMP a	BNE zm	CMP zm	DEX		ENT	LDA inx	CPY #	CPX #	PLY		LDB inx	
Cx	RTI	INC A	IAB	INC a	BVS zm	INC zm			WAI	STA inx	CPY a	CPX a	PHX		STB inx	
Dx	JMP zm	DEC A	DBA	DEC a	BVC zm	DEC zm				LDA iny	CPY zm	CPX zm	PLX		LDB iny	
Ex	JSL	ASR #	ARB	ASR a		ASR zm			NOP	STA iny	CPY ind	CPX ind	PHA		STB iny	
Fx	RTL	CMP ind		CMP inx		CMP iny			BRK		CPY inx	CPX iny	PLA			

"#" means Immediate data.
"a" means Absolute addressing.
"zm" means Zero Matrix, which refers to the first 4 GiB of the address space.
"zmx" means Zero Matrix, indexed with the X register.
"zmy" means Zero Matrix, indexed with the Y register.
"ind" means Indirect addressing, Also known as pointer addressing.
"inx" means Indexed Indirect addressing.
"iny" means Indirect Indexed addressing.
"A" means Accumulator.
And no operand means implied addressing.

2.3.1. Descriptions for each instruction of the Sux Base ISA

CPS	Clear Processor Status register.
Opcode	$00

The CPS instruction clears all 64 bits of the processor status register.

ADC	ADd with Carry.
Opcodes	#=$01 a=$03 zm=$05
Flags	N V Z C

The ADC instruction adds the operand with the accumulator, and then stores it back into the accumulator. If the carry flag is set, then it adds the carry flag along with the sum.

AAB	Add Accumulator, and B with carry.
Opcode	$02
Flags	N V Z C

The AAB instruction is just like the ADC instruction, except that it uses the B register as an operand, rather than memory.

PHB	PusH the B register.
Opcode	$06

The PHB instruction pushes the number of bytes specified, from the B register, onto the stack.

PHP	PusH the Processor status register.
Opcode	$08

The PHP instruction pushes the number of bytes specified, from the processor status register, onto the stack.

LDA	LoaD Accumulator.
Opcodes	#=$09 a=$19 zm=$39 zmx=$59 zmy=$79 ind=$99 inx=$B9 iny=$D9
Flags	N Z

The LDA instruction loads the value from the operand, into the Accumulator. It also sets the N, and Z flags.

LDY	LoaD value into Y register.
Opcodes	#=$0A a=$1A zm=$3A zmx=$5A ind=$7A inx=$9A
Flags	N Z

The LDY instruction loads the value from the operand, into the Y register. It also sets the N, and Z flags.

LDX	LoaD value into X register.
Opcodes	#=$0B a=$1B zm=$3B zmx=$5B ind=$7B inx=$9B
Flags	N Z

The LDX instruction loads the value from the operand, into the X register. It also sets the N, and Z flags.

TAB	Transfer value from the Accumulator, to the B register.
Opcode	$0C
Flags	N Z

The TAB instruction transfers the value that is in the Accumulator, to the B register. It also sets the N, and Z flags.

LDB	LoaD value into B register.
Opcodes	#=$0E a=$1E zm=$3E zmx=$5E zmy=$7E ind=$9E inx=$BE iny=$DE
Flags	N Z

The LDB instruction loads the value from the operand, into the B register. It also sets the N, and Z flags.

JMP	JuMP to memory address.
Opcodes	a=$10 zm=$D0 ind=$04 inx=$14 iny=$24

The JMP instruction sets the program counter to the address, specified by the operand.

SBC	SuBtract with Carry.
Opcodes	#=$11 a=$13 zm=$15
Flags	N V Z C

The SBC instruction subtracts the operand with the accumulator, and then stores the result back into the accumulator. If the carry flag is set, then it borrows from the carry flag along with the sum.

SAB	Subtract Accumulator, and B with carry.
Opcode	$12
Flags	N V Z C

The SAB instruction is just like the SBC instruction, except that it uses the B register as an operand, rather than memory.

PLB	PuLl value from the stack, into the B register.
Opcode	$16

The PLB instruction pulls/pops the number of bytes specified, off the stack, and into the B register.

PLP	PuLl value from the stack, into the Processor status register.
Opcode	$18

The PLB instruction pulls/pops the number of bytes specified, off the stack, and into the Processor status register.

TBA	Transfer value from the B register to the Accumulator.
Opcode	$1C
Flags	N Z

The TBA instruction transfers the value that is in the B register, to the Accumulator. It also sets the N, and Z flags.

JSR	Jump to SubRoutine.
Opcodes	zm=$20 ind=$34 inx=$44 iny=$54

The JSR instruction pushes the current value of the program counter, onto the stack, and then sets the program counter to the address, specified by the operand.

AND	bitwise AND accumulator.
Opcodes	#=$21 a=$23 zm=$25
Flags	N Z

The AND instruction takes an operand, and ANDs it with the accumulator, then saves the results back into the accumulator.

ABA	bitwise And Accumulator, with the B register.
Opcode	$22
Flags	N Z

The ABA instruction is just like the AND instruction, except that it uses the B register as an operand, rather than memory.

CPB	ComPare the B register, with an operand.
Opcodes	#=$26 a=$36 zm=$46 ind=$56 inx=$66 iny=$76
Flags	N V Z C

The CPB instruction compares the value in the B register, with an operand, and sets the flags accordingly. It compares the two values by subtracting them.

STT	STart a Thread.
Opcode	$28

The STT instruction, sets the current running threads register, and then has the threads read their thread vector. The current running threads register is an 8 bit register, that states which threads are running currently, the first 7 bits of the register states which of the 7 other threads are running, the remaining 16 bits of the operand, if specified, are used to tell the instruction which core to start, and are read bytewise, rather than bitwise. This instruction will not be implemented, if the core count, and thread count are both 1, otherwise, it will.

STA	STore the Accumulator, into memory.
Opcodes	a=$29 zm=$49 zmx=$69 zmy=$89 ind=$A9 inx=$C9 iny=$E9

The STA instruction writes the value of the Accumulator, into a memory address.

STY	STore the Y register, into memory.
Opcodes	a=$2A zm=$4A zmx=$6A ind=$8A inx=$AA

The STY instruction writes the value of the Y register, into a memory address.

STX	STore the X register, into memory.
Opcodes	a=$2B zm=$4B zmx=$6B ind=$8B inx=$AB

The STX instruction writes the value of the X register, into a memory address.

TAY	Transfer value from the Accumulator, to the Y register.
Opcode	$2C
Flags	N Z

The TAY instruction transfers the value that is in the Accumulator, to the Y register. It also sets the N, and Z flags.

STB	STore the B register, into memory.
Opcodes	a=$2E zm=$4E zmx=$6E zmy=$8E ind=$AE inx=$CE iny=$EE

The STB instruction writes the value of the B register, into a memory address.

BPO	Branch if POsitive.
Opcodes	a=$30 zm=$64

The BPO instruction will branch, if the Negative flag is cleared.

ORA	bitwise OR Accumulator.
Opcodes	#=$31 a=$33 zm=$35
Flags	N Z

The ORA instruction takes an operand, and ORs it with the accumulator, then saves the results back into the accumulator.

OAB	bitwise Or Accumulator, with the B register.
Opcode	$32
Flags	N Z

The OAB instruction is just like the ORA instruction, except that it uses the B register as an operand, rather than memory.

SEI	SEt the Interrupt flag.
Opcode	$38
Flags	I

The SEI instruction, sets the Interrupt flag, and thus disabling maskable interrupts.

TYA	Transfer value from the Y register to the Accumulator.
Opcode	$3C
Flags	N Z

The TYA instruction transfers the value that is in the Y register, to the Accumulator. It also sets the N, and Z flags.

BNG	Branch if NeGative.
Opcodes	a=$40 zm=$74

The BNG instruction will branch, if the Negative flag is set.

XOR	bitwise XOR accumulator.
Opcodes	#=$41 a=$43 zm=$45
Flags	N Z

The XOR instruction takes an operand, and XORs it with the accumulator, then saves the results back into the accumulator.

XAB	bitwise Xor Accumulator, with the B register.
Opcode	$42
Flags	N Z

The XAB instruction is just like the XOR instruction, except that it uses the B register as an operand, rather than memory.

CLI	CLear the Interrupt flag.
Opcode	$48
Flags	I

The CLI instruction, clears the Interrupt flag, and thus enabling maskable interrupts.

TAX	Transfer value from the Accumulator to the X register.
Opcode	$4C
Flags	N Z

The TAX instruction transfers the value that is in the Accumulator, to the X register. It also sets the N, and Z flags.

BCS	Branch if Carry Set.
Opcodes	a=$50 zm=$84

The BCS instruction will branch, if the Carry flag is set.

LSL	Logical Shift Left accumulator.
Opcodes	#=$51 a=$53 zm=$55
Flags	N Z C

The LSL instruction takes an operand, and shifts the accumulator left by the number of bits from the operand, then saves the results back into the accumulator. The Carry flag is set whenever a one is shifted out of the accumulator.

LLB	Logical shift Left accumulator, with the B register.
Opcode	$52
Flags	N Z C

The LLB instruction is just like the LSL instruction, except that it uses the B register as an operand, rather than memory.

SEC	SEt the Carry flag.
Opcode	$58
Flags	C

The SEC instruction, sets the Carry flag.

TXA	Transfer value from the X register to the Accumulator.
Opcode	$5C
Flags	N Z

The TXA instruction transfers the value that is in the X register, to the Accumulator. It also sets the N, and Z flags.

BCC	Branch if Carry Clear.
Opcodes	a=$60 zm=$94

The BCC instruction will branch, if the Carry flag is cleared.

LSR	Logical Shift Right accumulator.
Opcodes	#=$61 a=$63 zm=$65
Flags	N Z C

The LSR instruction takes an operand, and shifts the accumulator right by the number of bits from the operand, then saves the results back into the accumulator. The Carry flag is set whenever a one is shifted out of the accumulator.

LRB	Logical shift Right accumulator, with the B register.
Opcode	$62
Flags	N Z C

The LRB instruction is just like the LSR instruction, except that it uses the B register as an operand, rather than memory.

CLC	CLear the Carry flag.
Opcode	$68
Flags	C

The CLC instruction, clears the Carry flag.

TYX	Transfer value from the Y register to the X register.
Opcode	$6C
Flags	N Z

The TYX instruction transfers the value that is in the Y register, to the X register. It also sets the N, and Z flags.

BEQ	Branch if EQual.
Opcodes	a=$70 zm=$A4

The BEQ instruction will branch, if the Zero flag is set.

ROL	ROtate Left accumulator.
Opcodes	#=$71 a=$73 zm=$75
Flags	N Z C

The ROL instruction takes an operand, and rotates the accumulator left by the number of bits from the operand, then saves the results back into the accumulator. The Carry flag is set whenever a one is shifted out of the accumulator.

RLB	Rotate Left accumulator, with the B register.
Opcode	$72
Flags	N Z C

The RLB instruction is just like the ROL instruction, except that it uses the B register as an operand, rather than memory.

SSP	Set the Stack Protection flag.
Opcode	$78
Flags	S

The SSP instruction, sets the Stack protection flag.

TXY	Transfer value from the X register to the Y register.
Opcode	$7C
Flags	N Z

The TXY instruction transfers the value that is in the X register, to the Y register. It also sets the N, and Z flags.

BNE	Branch if Not Equal.
Opcodes	a=$80 zm=$B4

The BNE instruction will branch, if the Zero flag is cleared.

ROR	ROtate Right accumulator.
Opcodes	#=$81 a=$83 zm=$85
Flags	N Z C

The ROR instruction takes an operand, and rotates the accumulator right by the number of bits from the operand, then saves the results back into the accumulator. The Carry flag is set whenever a one is shifted out of the accumulator.

RRB	Rotate Right accumulator, with the B register.
Opcode	$82
Flags	N Z C

The RRB instruction is just like the ROR instruction, except that it uses the B register as an operand, rather than memory.

INY	INcrement the Y register.
Opcode	$86
Flags	N Z

The INY instruction increments the Y register.

CSP	Clear the Stack Protection flag.
Opcode	$88
Flags	S

The CSP instruction, clears the Stack protection flag.

TSX	Transfer value from the Stack pointer to the X register.
Opcode	$8C
Flags	N Z

The TSX instruction transfers the value that is in the Stack pointer, to the X register. It also sets the N, and Z flags.

BVS	Branch if oVerflow Set.
Opcodes	a=$90 zm=$C4

The BVS instruction will branch, if the Overflow flag is set.

MUL	MULtiply accumulator, by operand.
Opcodes	#=$91 a=$93 zm=$95
Flags	N V Z C

The MUL instruction takes an operand, and multiplys the accumulator by the operand, then saves the results back into the accumulator. If the Carry flag is set, then it will add one to the result.

MAB	Multiply Accumulator, with the B register.
Opcode	$92
Flags	N V Z C

The MAB instruction is just like the MUL instruction, except that it uses the B register as an operand, rather than memory.

DEY	DEcrement the Y register.
Opcode	$96
Flags	N Z

The DEY instruction decrements the Y register.

SEV	SEt the oVerflow flag.
Opcode	$98
Flags	V

The SEV instruction, sets the Overflow flag.

TXS	Transfer value from the X register to the Stack pointer.
Opcode	$9C
Flags	N Z

The TXS instruction transfers the value that is in the X register, to the Stack pointer, if the prefix byte has a high nibble of $1, then it will turn into an immediate data instruction, and the operand will set the Stack Bank Register, the SBR is a 16 bit register that states where the starting address of the Stack pointer is. It also sets the N, and Z flags.

BVC	Branch if oVerflow Clear.
Opcodes	a=$A0 zm=$D4

The BVC instruction will branch, if the Overflow flag is cleared.

DIV	DIVide accumulator, by operand.
Opcodes	#=$A1 a=$A3 zm=$A5
Flags	N V Z

The DIV instruction takes an operand, and divides the accumulator by the operand, then saves the results back into the accumulator.

DAB	Divide Accumulator, with the B register.
Opcode	$A2
Flags	N V Z

The DAB instruction is just like the DIV instruction, except that it uses the B register as an operand, rather than memory. It also let's you dab on all those lesser ISAs.

INX	INcrement the X register.
Opcode	$A6
Flags	N Z

The INX instruction increments the X register.

CLV	CLear the oVerflow flag.
Opcode	$A8
Flags	V

The CLV instruction, clears the Overflow flag.

PHY	PusH the Y register.
Opcode	$AC

The PHY instruction pushes the number of bytes specified, from the Y register, onto the stack.

RTS	ReTurn from Subroutine.
Opcodes	$B0

The RTS instruction is the opposite of the JSR instruction, in that it pulls from the stack, the address that JSR had pushed, and places it in the program counter.

CMP	CoMPare accumulator, with operand.
Opcodes	#=$B1 a=$B3 zm=$B5 ind=$F1 inx=$F3 iny=$F5
Flags	N V Z C

The CMP instruction compares the value in the B register, with an operand, and sets the flags accordingly. It compares the two values by subtracting them.

CAB	Compare Accumulator, with the B register.
Opcode	$B2
Flags	N V Z C

The CAB instruction is just like the CMP instruction, except that it uses the B register as an operand, rather than memory.

DEX	DEcrement the X register.
Opcode	$B6
Flags	N Z

The DEX instruction decrements the X register.

ENT	ENd a Thread.
Opcode	$B8

The ENT instruction, clears the bits in the Current Running Threads register, specified by the operand.

CPY	ComPare the B register, with an operand.
Opcodes	#=$BA a=$CA zm=$DA ind=$EA inx=$FA
Flags	N V Z C

The CPY instruction compares the value in the Y register, with an operand, and sets the flags accordingly. It compares the two values by subtracting them.

CPX	ComPare the X register, with an operand.
Opcodes	#=$BB a=$CB zm=$DB ind=$EB inx=$FB
Flags	N V Z C

The CPX instruction compares the value in the X register, with an operand, and sets the flags accordingly. It compares the two values by subtracting them.

PLY	PuLl value from the stack, into the Y register.
Opcode	$BC

The PLY instruction pulls/pops the number of bytes specified, off the stack, and into the Y register.

RTI	ReTurn from Interrupt.
Opcodes	$C0

The RTI instruction is like the RTS instruction, in that it pulls from the stack, the previous value of the processor status register, and the previous address that had been pushed from an interrupt, and places them into, the processor status register, and the program counter, respectivly.

INC	INCrement accumulator, or memory address.
Opcodes	A=$C1 a=$C3 zm=$C5
Flags	N Z

The INC instruction increments either the accumulator, or the contents of a memory address.

IAB	Increment Accumulator, with the B register.
Opcode	$C2
Flags	N Z

The IAB instruction increments both the accumulator, and the B register.

WAI	WAIt for an interrupt.
Opcode	$C8

The WAI instruction, puts the CPU into a sleep mode, where it is only awake enough to be able to respond to an interrupt, and as such, not only uses less power, but also allows for very fast interrupt response times.

PHX	PusH the X register.
Opcode	$CC

The PHX instruction pushes the number of bytes specified, from the X register, onto the stack.

DEC	DECrement accumulator, or memory address.
Opcodes	A=$D1 a=$D3 zm=$D5
Flags	N Z

The DEC instruction decrements either the accumulator, or the contents of a memory address.

DBA	Dncrement Accumulator, with the B register.
Opcode	$D2
Flags	N Z

The DBA instruction decrements both the accumulator, and the B register.

PLX	PuLl value from the stack, into the X register.
Opcode	$DC

The PLX instruction pulls/pops the number of bytes specified, off the stack, and into the X register.

JSL	Jump to Subroutine, Long addressing.
Opcode	a=$E0

The JSL instruction, is just like the JSR instruction, except that it pushes a 64 bit address onto the stack.

ASR	Arithmetic Shift Right accumulator.
Opcodes	#=$E1 a=$E3 zm=$E5
Flags	N Z C

The ASR instruction takes an operand, and shifts the accumulator right by the number of bits from the operand, then saves the results back into the accumulator. But unlike LSR, ASR shifts in the sign bit into the accumulator. The Carry flag is set whenever a one is shifted out of the accumulator.

ARB	Arithmetic shift Right accumulator, with the B register.
Opcode	$E2
Flags	N Z C

The ARB instruction is just like the ASR instruction, except that it uses the B register as an operand, rather than memory.

NOP No OPeration.
Opcode $E8

The NOP instruction does what you think it does, it does nothing at all.

PHA PusH the Accumulator.
Opcode $EC

The PHA instruction pushes the number of bytes specified, from the Accumulator, onto the stack.

RTL ReTurn from subroutine, Long addressing.
Opcodes $F0

The RTL instruction, is just like the RTS instruction, except that it pulls a 64 bit address from the stack.

BRK BReaKpoint.
Opcode $F8

The BRK instruction triggers a software interrupt, which pushes the value in the processor status register, and the address in the program counter, onto the stack, then reads the BRK vector, and then jumps to the address located there.

PLA PuLl value from the stack, into the Accumulator.
Opcode $FC

The PLA instruction pulls/pops the number of bytes specified, off the stack, and into the accumulator.

2.4. The Prefix Byte

As mentioned before, there is a prefix byte. The prefix byte is a byte that occurs before the opcode, and gives the CPU more control over things like, the number of bytes that it will read/write, switching between 8/16 bit addressing, and 32/64 bit addressing, and switching between different instruction set extensions.

Figure 2.3 shows the layout of the prefix byte.
Table 2 explains what the prefix bits do.

EX1	EX0	RS1	RS0	AM	1	1	1
7	6	5	4	3	2	1	0

Figure 2.3: The control bits of the prefix byte.

Table 2. Description of the prefix bits

AM	**Addressing Modes**
0	Normal Addressing Modes (8/16 bit)
1	Extended Addressing Modes (32/64 bit)
RS1/RS0	**Register Size**
0/0	8 bit register
0/1	16 bit register
1/0	32 bit register
1/1	64 bit register
EX1/EX0	**Extension Selection**
0/0	Base ISA
0/1	GFsuX Graphics Oriented Extension
1/0	Unused
1/1	μDCROM

If no prefix byte is found, then it treats that byte as an opcode, and acts as if it had a prefix byte, with all the prefix bits set to 0.

2.5. Assembly Language Syntax for the Sux Base ISA

The syntax of Sux assembly, was designed to be simple, and very easy to understand. The syntax looks like this.

ADC.W #$1000

This is what it means.

ADC	Instruction
.W	Register Size Suffix
#$1000	Operand

For the Register Size Suffixes, there are four options.

None	One byte (8 bits)
.W/.2	Two bytes (16 bits)
.D/.4	Four bytes (32 bits)
.Q/.8	Eight bytes (64 bits)

These Suffixes control the prefix byte. If there is no suffix, then it will not place a prefix byte, saving a byte.

The operand can be a few things.

Syntax	**Description**
Value	Decimal Address
$<Value>	Hexadecimal Address
%<Value>	Binary Address
#[token]<Value>	Immediate Data Value
[token]<Value>, <X/Y>	Zero Matrix, Indexed with X, or Y
([token]<Value>)	Indirect Addressing
([token]<Value>, X)	Indexed Indirect Addressing
([token]<Value>), Y	Indirect Indexed Addressing

2.6. The µDCROM

The µDCROM is a type of Programmable Logic Array (PLA) that has each of it's interconnects connected to an SRAM bit. This SRAM can then be accessed just like any other part of memory, and as such, is fully reprogrammable by the user.

The starting location of the µDCROM, is value located at the µDCROM vector.

The byte before the start of the µDCROM is the control byte, which is used to specify how the µDCROM should be accessed.

3. The Sux Reference Platform

The Sux Reference Platform, or SuRP, is the reference architecture for Sux, and what most other architectures should be based on.

SuRP will have two boot firmwares, one called SuBAsm, or the Sux Bootstrapped Assembler, and a bootstrapped version of the Tiny C Compiler, called SuBTinyCC.

SuRP will have an MMU chip specifically designed for Sux, called the MinMMU, a PCI Express controller called MinPCIE, and an RS-232 controller called the Min232.

The reasoning for using an assembler, and a C compiler as boot firmware, was because we wanted to make sure that no one could run prorpietary code on any Sux based system, and to allow for a possible Maximite like computer, called the SuRPimite, which will use a microcontroller variant of Sux, called MicroSux.

MicroSux will come with SuBAsm, and SuBTinyCC built-in to the chip, but will still be on a flash chip, so that you can update it, but cannot be directly accessed by the user, and as such, will have a second flash for user programs.