Simple Computer 0 Documentation

by Connor L.

Contents

1 Summary

The SC0 is a hobby project of mine inspired by **Dr. Yale Patt's** LC-3(b) teaching computers. In creating the SC0, I am hoping to gain a deeper understanding of how computer systems are created, and the architecture behind them. I am studying computer architecture at UT Austin, and am excited to learn more about this field as I progress through my degree.

The project is written in Rust. I chose this for two reasons:

- Rust has a unique and powerful data flow design
- Memory management is completely different (and, in my opinion, stronger) than C++

2 Description of SC0

The SC0 has several advantages (or, disadvantages, depending on how you look at it) from the LC-3(b) systems:

- 32-bit addressability — more instructions, wider range of values for constants
- Multiple types of memory access instructions — direct and indirect
- More (useful) instructions — syscalls, an innate stack, logic and math

Currently, the SC0 has a fixed address space of 32kB (realistically, no program will ever fill up this much space, unless I write a C to SC0 compiler). Registers also can store up to 32 bits of information.

Note: A list of system calls and example program formatting can be found later in this document.

3 Register Map

Register	Type	Notes
R0	GPR	General purpose
R1	GPR	General purpose
R2	GPR	General purpose
R3	GPR	General purpose
R4	GPR	General purpose
R5	GPR	General purpose
R6	GPR	General purpose
R7	GPR	General purpose
R8	GPR	General purpose
R9	GPR	General purpose
R10	GPR	General purpose
R11	GPR	General purpose
R12	GPR/Return Register	Calls store the return PC in this register
R13	SP	Stack pointer
R14	PC	Current instruction address
R15	PSR	Condition Codes and Privelege

The special registers *can* be accessed directly, but it is **highly** advised to not do that. Condition codes are in the format of N, Z, P, representing negative, zero, and positive respectively. Certain instructions set the condition codes (CC) depending on the resultant value. Any functions that are user-defined can have any style of parameter passing, either from the stack or through registers directly.

4 Instruction Listing and Formatting

Instruction	Format	Pseudocode
ADD$^{\times}$	add dest, src1, src2/imm	dest = src1 + src2
SUB$^{\times}$	sub dest, src1, src2/imm	dest = src1 - src2
MUL$^{\times}$	mul dest, src1, src2/imm	dest = src1 * src2
DIV$^{\times}$	div dest, src1, src2/imm	dest = src1 / src2, no floats
MOV$^{\times}$	mov dest, src/imm	dest = src
AND$^{\times}$	and dest, src1, src2/imm	dest = src1 & src2
OR$^{\times}$	or dest, src1, src2/imm	dest = src1 | src2
NOT$^{\times}$	not dest, src	dest = ˜src
XOR$^{\times}$	xor dest, src1, src2/imm	dest = src1 ˆ src2
LSHF$^{\times}$	lshf dest, src1, src2/imm	dest = src1 << src2
RSHF$^{\times}$	rshf dest, src1, src2/imm	dest = src1 >> src2
LEA	lea dest, LABEL	dest = address of label
LDI$^{\times}$	ldi dest, src	dest = DWORD(mem[mem[src]])
LDB$^{\times}$	ldb dest, src1, src2/imm	dest = BYTE(mem[src1 + src2])
LDW$^{\times}$	ldw dest, src1, src2/imm	dest = WORD(mem[src1 + src2])
LDD$^{\times}$	ldd dest, src1, src2/imm	dest = DWORD(mem[src1 + src2])
STI	sti dest, src	DWORD(mem[mem[dest]]) = src
STB	stb dest, src1, src2/imm	BYTE(mem[dest + src2]) = src1
STW	stw dest, src1, src2/imm	WORD(mem[dest + src2]) = src1
STD	std dest, src1, src2/imm	DWORD(mem[dest + src2]) = src1
JMP	jmp reg/LABEL	PC = reg/LABEL address
CALL	call reg/LABEL	R12 = Incremented PC; PC = reg/LABEL address
SYSCALL	syscall CODE	An internal CALL execution
BR(nzp)	BR(nzp) LABEL	Conditional (or unconditional) branch
CMP$^{\times}$	cmp src1, src2/imm	CC = (src1 - src2)
PUSH	push src/imm	mem[sp++] = src
POP$^{\times}$	pop dest	dest = mem[−−sp]

Please note: instructions marked with $\times$ set condition codes on execution.

5 Usage of SC0

User programs are entered into an SC0 program file that ends in .asm, and uses instructions shown above. The parser is case-insensitive, so you cannot have duplicate labels! Any constants must be formatted as either hex (0xNUMBERHERE), or decimal (#NUMBER HERE). The typical formatting for an SC0 program is as follows:

LABEL: INSTRUCTION OPERANDS ; comments

There are four pseudo-ops in the SC0:

- **.ORIG XXXX** — define starting address for program at XXXX
- **.END** — signal end of program for parser
- **.FILL XXXX** — fill that memory location with value XXXX
- **.STRING XXXX** — insert ascii characters in-order, starting at the .STRING memory address

The FILL and STRING pseudo-ops can be prefixed with a label, allowing for ease-of-access in the user program.

Once a user program has been finalized, it may be parsed and assembled by the SC0 program by launching the simulator and loading the file into memory.

6 Syscalls and Examples

HALT	Special call: shuts down SC0.
PRINT	Prints the string found at **R0** until null terminator.
DISPLAY	Displays data found in R0 on-screen. No automatic newline.
DELAY	Waits **R0** cycles, then resumes program execution.
INPUT	Reads one character from input into **R0**.

The above system calls are what are implemented in the SC0. Currently, syscalls are executed on an abstraction layer. In reality, the PC should jump to the syscall value in memory, and read the stored address there like a lookup table, then go execute the syscall. The SC0 simply reads the syscall type and executes it on the Rust layer, not on the assembly layer.

An example program would look as follows:

```
        .ORIG x100 ; start program at memory address 0x100
        LEA R0, LOC1
        LEA R1, LOC2
        ; load numbers from memory
        LDI R0, R0
        LDI R1, R1
        ADD R2, R0, R1
        LEA R3, STORE
        STI R3, R2 ; store sum of num1 and num2 into mem[0x300]
        SYSCALL HALT
STORE .FILL 0x300 ; store location
LOC2  .FILL 0x204 ; location of number 2
LOC1  .FILL 0x200 ; location of number 1
    .END
```

As the comments suggest, this program adds the two 4 byte numbers found at **MEM[0x200]** and **MEM[0x204]** and stores the result in **R2**. Then, the value of **R2** is stored into **MEM[0x300]**. Note the combination of **LEA** and **LDI/STI**. This is a common combination to use the indirect load and store instructions. The **LEA** loads the address of the variables into the specified registers, and then the **LDI/STI** perform the chained memory operation as seen in the **Instruction Listing**.