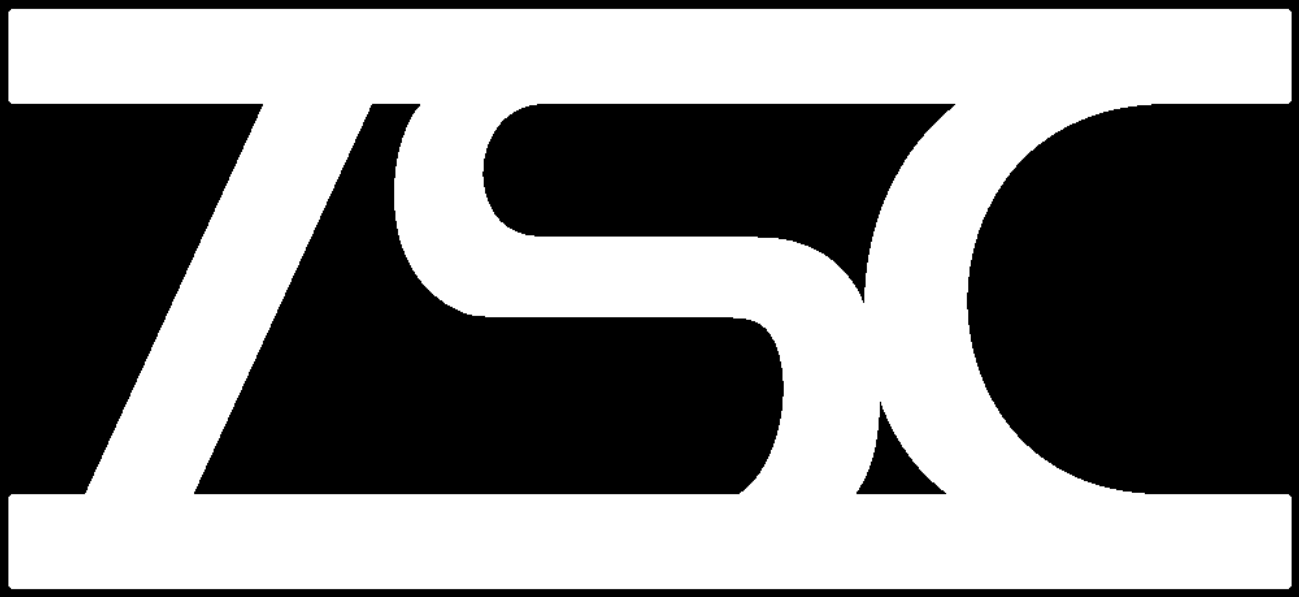

MICROCOMPUTERS

MICROCOMPUTERS

MICROCOMPUTERS

MICROCOMPUTERS

SM1
HARDWARE & PROGRAMMING MANUAL

TURNER SEMICONDUCTOR

SM1 Hardware & Programming Manual

Revision 1

20 / 05 / 2025

SM1 Features

- Sixteen bit parallel processing
- 16 instructions
- 12 ALU Operations
- 13 General purpose registers
- 5 addressing modes
- True indexing capabilities
- Programmable stack pointer
- Interrupt capability
- Use with any type or speed memory
- 16 bit bi-directional Data Bus
- Addressable memory range of up to 65K words
- Bus active output

Pinout

On the SM1, there is 66 pins, these pins are for control signals to/from the CPU and to interface with memory.

Memory interface

Cyan: Trigger interrupted
Yellow: A0-A15 (from left to right)
Red: Memory write
Green: Memory read
Magenta: D0-D15 (data out) (from left to right)
Blue: D0-D15 (data in) (from left to right)

Control I/O

Purple (left): X3-X0 (aux in) (from left to right)
Red: Reset
White: Clock input
Purple (left): X7-X0 (aux out) (from left to right)
Blue: Set carry

Address bus (A0-A15):

16 bit address to allow up to 65K of addressable memory at one time.

Write:

When the SM1 writes data to the memory bus, this pin is pulsed HIGH for a duration of one tick.

Read:

While this pin is high, the SM1 is reading data from the specified address.

Data bus (D0-D15):

16 bit bi-directional data bus, transferring data to and from peripherals.

Reset:

This input is used to initialize the CPU from a halt state, while this pin is held high, the CPU will not execute any instructions, the data bus is set to LOW, the address bus is set to 0xFFFF, and the read pin is set to HIGH. The clock cycle after the reset pin is LOW will set the program counter (PC) to the value on the data bus. The reset cycle will disable interrupts, but the registers and stack pointer are not reset.

Clock:

Clock input to the CPU.

Set carry:

Rising edge on this pin will set the carry bit in the status register.

AUX Inputs (X0-X3):

A rising edge on these pins will set the corresponding bits in the status register.

AUX Outputs (X0-X7):

These bits are the direct state of bits 8-15 of the status register.

Interrupt:

While this pin is HIGH, interrupt disable bit is cleared, and the CPU is in the fetch phase, 0x0000 (BRK) is force loaded into the instruction register, and the break bit in the status register is cleared.

Addressing modes

Immediate addressing:

This addressing mode is available to all instructions, the operand is contained in the second word of the instruction and no further memory addressing is required.

Absolute addressing:

Absolute addressing is only available to LOD (Load) and STO (Store) instructions. In this addressing mode, data is read/written to memory.

Indexed addressing:

Index addressing is only available to LOD and STO instructions. This addressing mode is used in conjunction with registers to read/write to memory at an indexed location. The effective address is calculated by adding the base address to the contents in a register.

Register addressing:

This form of addressing is represented with a one word instruction, implying an operation with one or more registers.

Implied addressing:

In the implied addressing mode, the address containing the operand is implicitly stated in the operation code of the instruction.

Reset, fetch, execute, and interrupt cycles

Reset:

When the reset pin is pulled HIGH, the CPU will read from address 0xFFFF for the start vector to begin execution. The next clock cycle after the reset pin is LOW the CPU will latch the start vector to the PC.

Fetch:

During the fetch cycle, the CPU outputs the current program counter to the address bus to read the next opcode. When the clock signal is pulled HIGH, the CPU latches the opcode into the instruction register, the PC is incremented, and the CPU starts to read the next value as immediate data. When the clock goes back to low, the CPU latches the immediate data, the PC is incremented if immediate data is used in the opcode (otherwise the latched data is discarded), and the CPU is put into the execute phase. If the instruction does not access memory, and a jump is not taken, the CPU returns immediately back to the fetch phase, and the instruction is executed during the fetch.

Execute:

During the execute phase, the instruction is executed in two steps, the first execution phase is executed on the rising edge of the clock, and the second on the falling edge of the clock.

Interrupt:

The interrupt is triggered on the next fetch cycle when: The interrupt disable bit is clear, the CPU is currently not in an interrupted state, and the interrupt pin is pulled HIGH. When an interrupt is triggered, during the fetch cycle, the value 0x0000 (BRK) is force loaded into the instruction register, but the PC is not incremented, the PC is pushed to the stack and

the PC is set to the value set by the INT instruction. During an interrupt, an interrupt is not able to trigger again until the CPU leaves the interrupt state with a return from interrupt (RTI) instruction, exception being the BRK instruction able to re-trigger the interrupt. If the interrupt is triggered by a BRK instruction, and not from the pin, the B (break) bit is set in the status register.

PROGRAMMING

Instruction Set Architecture

Registers:

The SM1 has 16 internal accessible registers. There are 13 general purpose registers and 3 special purpose registers.

Registers 1 to 13 are labeled r0-r12, register 14 is the stack pointer (sp), register 15 is a status register (st), and register 0 is a special immediate data, read-only register. Register 0 is latched with immediate data on the second phase of the fetch cycle, this register is used like any other register to provide a flexible, but simple instruction set.

Instruction encoding:

Instruction are encoded in 16 bit opcodes, the opcode contains the instruction, operand 1, operand 2, and operand 3.

Instruction word format: **IIIIXXXXYYYYZZZZ ← LSB**

Least significant bit is on the right.

I (4 bits): Instruction
X (4 bits): Operand 1 register selector
Y (4 bits): Operand 2 register selector
Z (4 bits): Operand 3 General instruction options / register selector

Each instruction may use one, two or all three operands as inputs. Operand 1 and 2 are selectors to select which registers to use as inputs to the instruction, instructions that generates a result will be stored back to the register selected with operand 1, except the memory write instructions where operand 1 is the register used to write to memory. Operand 3 is used differently depending on the instruction. Operand 3 may be used as the index register for store and load instructions, ALU function selector, or the conditional jump selector.

Stack pointer:

The stack pointer register is a read and writable register that is automatically incremented/decremented when using the stack instructions (push, pop, jump to subroutine, return from subroutine, and return from interrupt). The SP register is a pre-decrement / post-increment model, meaning that when pushing to the stack, the stack pointer is decremented before writing to memory, and reads from memory before incrementing.

Status register:

The status is a read and write register containing the flags that hold the CPU's state, the flags are: Carry (C), Zero (Z), Negative (N), Interrupt disable (I), Break (B), AUX0-3 (X0-X3), they correspond to bit 0, 1, 2, 3, 4, 8, 9, 10, 11 respectively. A carry is generated when shifting a bit out with shift functions, or when a carry occurs from an arithmetic function. The zero flag is set when a value written to a register is NULL. The negative flag is set when a value written to a register has the most significant bit set. Writing to this register will not cause Z or N bit to change/written with unexpected values. Bitwise operations on the status register itself will not yield unexpected results on the Z/N bits, allowing successful setting/clearing of flags manually. The break flag is set if the interrupt was triggered from a break instruction. Aux bits are set with the rising edge on the AUX Inputs (X0-X3) pins.

Instructions

The SM1 only has 16 instructions, but due to the instruction encoding, every one of these instructions can be powerful.

0x0 – BRK

BReaK enters an interrupt routine. When this instruction is executed, the CPU pushes the current PC to the stack and sets the PC to the address set with the INT instruction. The B flag is set in the status.
No operands.

0x1 – JMP

JuMP if the condition specified in operand 2/3 is true.
Operand 1: Address to jump to
Operand 2: Extension to operand 2; Bitwise AND with the X0–X3 bits in the status register.
Operand 3: Bitwise AND with the first three bits of the status register. If the result is non-zero, PC is set to operand 1. MSB of operand 3 will invert the condition. Operand 2 extends operand 3 to the X bits.

0x2 – MOV

MOVe copies contents from one register to another.
Operand 1: Destination register
Operand 2: Source register
Modifies: ZN

0x3 – LOD

LoaD from memory at the specified address.

Operand 1: Destination register

Operand 2: Address

Operand 3: If not null, this will specify an index register. (Effective address = OP2+OP3)

Modifies: ZN

0x4 – STO

STOre to memory at the specified address.

Operand 1: Source register

Operand 2: Address

Operand 3: If not null, this will specify an index register. (Effective address = OP2+OP3)

0x5 – ALU

Perform an ALU operation on two registers and store the result back to the first operand. If the ALU function is bit shifting, operand 2 is ignored, but the fetch phase does not account for this, so if set to 0 the immediate data is still loaded.

Operand 1: Data

Operand 2: Data

Operand 3: ALU function

Modifies: (C)ZN (Does not modify the carry on bitwise operations and the negate operation)

ALU functions:

0x0 – ADD (Add)

0x1 – ADC (Add with carry)

0x2 – SUB (Subtraction)

0x3 – SBC (Subtract with carry)

0x4 – SHL (Shift left bits in operand 1 by one bit, MSB is shifted into the carry flag)

0X5 – ROL (Same as SHL, except the carry flag is shifted in to the LSB)

0x6 – SHR (Shift right bits in operand 1 by one bit, LSB is shifted into the carry flag)

0X7 – ROR (Same as SHR, except the carry flag is shifted in to the MSB)

0x8 – AND (Bitwise AND)

0x9 – OR (Bitwise OR)

0xA – XOR (Bitwise Exclusive OR)

0xF – NEG (Negate / Generate 2's compliment)

0x6 – CMP

CoMPare two values. Same as ALU, except the result is not stored back into operand 1

Operand 1: Data

Operand 2: Data

Operand 3: ALU function

Modifies: (C)ZN (Does not modify the carry on bitwise operations and the negate operation)

0x7 – RPC

Read Program Counter. Read the current program counter and store it into operand 1.

Operand 1: Destination register

0x8 – NOP

No Operation. Waste a clock cycle.

0x9 – INT

INTerrupt set. This instruction sets the interrupt vector.

Operand 1: Interrupt vector

0xA – RESERVED FOR FURTHER USE

0xB – PUSH

PUSH data onto the stack.

Operand 1: Data

0xC – POP

POP data off the stack and write it to a register.

Operand 1: Destination register

Modifies: ZN

0xD – JSR

Jump to SubRoutine. Push the current PC to the stack and set the PC to operand 1.

Operand 1: Address

0xE – RTS

ReTurn from Subroutine. Pop the return address from the stack and write it to the PC.

0xF – RTI

ReTurn from Interrupt. Pop the return address from the stack and write it to the PC, and exit from the interrupt state.

ASSEMBLY REFERENCE

Introduction

The written assembly is a way to represent the machine code in a human readable format. For the SM1, the written assembly mnemonics does not always correspond directly to the machine's instruction names, this is to make reading the assembly much easier due to how the machine instructions are encoded. Assembly lines start with the instruction to perform, along with zero, one, or two parameters to go along with the instruction. A parameter can be a register name, or a constant integer in decimal, hexadecimal, binary, or octal. The parameter format must follow the supported class for the instruction.

Assembly Instruction Reference

INSTRUCTION	CLASS	DESCRIPTION
jmp	AB	Jump to another piece of code at the address in r1
bcs	AB	Branch if carry set
bcc	AB	Branch if carry clear
beq	AB	Branch if zero
bne	AB	Branch if not zero
bmi	AB	Branch if minus
bpl	AB	Branch if not minus
bxa	AB	Branch if bit X0 set
bna	AB	Branch if bit X0 clear
bxb	AB	Branch if bit X1 set
bnb	AB	Branch if bit X1 clear
bxc	AB	Branch if bit X2 set
bnc	AB	Branch if bit X2 clear
bxd	AB	Branch if bit X3 set
bnd	AB	Branch if bit X3 clear
mov	CDFGHIJKLMNO	Copy the contents from the second operand to the first
add	CDE	Add op1 and op2, store to op1
adc	CDE	Add with carry

sub	CDE	Subtract op1 with op2, store to op1
sbc	CDE	Sub with carry
shl	A	Shift op1 left by one, bit shifted out is stored in the carry flag
rol	A	Rotate left, same as shl but the bit shifted in is from the carry flag
shr	A	Shift op1 right by one, bit shifted out is stored in the carry flag
ror	A	Rotate right, same as shr but the bit shifted in is from the carry flag
and	CDE	Bitwise AND op1 with op2, stored to op1
or	CDE	Bitwise OR op1 with op2, stored to op1
xor	CDE	Bitwise XOR op1 with op2, stored to op1
neg	A	Generate the two's compliment of op1, stored to op1
adds	CDE	Non storing version of add
adcs	CDE	Non storing version of adc
cmp / subs	CDE	Compare two values. Equivalent to a non-storing subtract. Suggested to use cmp when comparing values.
sbcs	CDE	Non storing version of sbc
shls	A	Non storing version of shl
rols	A	Non storing version of rol
shrs	A	Non storing version of shr
rors	A	Non storing version of ror
bit / ands	CDE	Non storing version of and.
ors	CDE	Non storing version of or
xors	CDE	Non storing version of xor
negs	A	Non storing version of

		neg
nop	0	No operation, waste a cycle
int	AB	Set the interrupt vector
push	AB	Push a value to the stack
pop	A	Pop a value from the stack
jsr	AB	Jump to a subroutine
rts	0	Return from a subroutine
brk	0	Trigger an interrupt
rti	0	Return from an interrupt
clc	0	Clear carry
sec	0	Set carry
clz	0	Clear zero
sez	0	Set zero
cln	0	Clear negative
sen	0	Set negative
cli	0	Clear interrupt disable
sei	0	Set interrupt disable
clxa	0	Set aux bit X0
sexa	0	Clear aux bit X0
clxb	0	Set aux bit X1
sexb	0	Clear aux bit X1
clxc	0	Set aux bit X2
sexc	0	Clear aux bit X2
clxd	0	Set aux bit X3
sexd	0	Clear aux bit X3

Classes

An operand class are the supported assembly formats which that instruction support, operands enclosed by “[“ and “]” reference a memory location pointed to by the effective address inside those brackets. U16 is an unsigned 16 bit integer. I16 is a signed or unsigned 16 bit integer. REG references a register (r0-r12).

```
0: OPER
A: OPER REG
B: OPER I16
C: OPER REG, REG
D: OPER REG, I16
E: OPER I16, REG
F: OPER REG, [U16]
G: OPER REG, [REG]
H: OPER REG, [REG+U16]
I: OPER REG, [REG+REG]
J: OPER [U16], REG
K: OPER [REG], REG
L: OPER [REG], I16
M: OPER [REG+U16], REG
N: OPER [REG+REG], REG
O: OPER [REG+REG], I16
```

Jumping and branching
jmp, beq, bne, bcs, bcc, bmi, bpl, bxa, bna, bxb, bnb, bxc, bnc, bxd, bnd

```
jmp 0x1000 ; Jump to address 0x1000
jmp r1     ; Jump to the address contained in register r1
beq 0x1234 ; Jump to address 0x1234 if the zero flag is set
bne 0x1234 ; Jump to address 0x1234 if the zero flag is clear
…
```

Jumps are a way to move code execution to another piece of code, jmp is an unconditional jump and will always be taken. beq, bne, etc. are conditional jumps and will be taken only when the condition is true, like for example if the zero flag is set.

Data transferring
mov

```
mov r1, 0x1234        ; Copy value 0x1234 into register r1
mov r1, r2            ; Copy contents of r2 to r1
mov r1, [0x1234]      ; Load into r1 the contents at address 0x1234
mov r1, [r2]          ; Load into r1 the contents at address contained in r2
mov r1, [r2+0x1234]   ; Load into r1 the contents at address 0x1234+r2
mov r1, [r2+r3]       ; Load into r1 the contents at address r2+r3
mov [0x1234], r1      ; Store the contents of r1 to address 0x1234
mov [r2], r1          ; Store r1 to the address contained in r2
mov [r2], 0x1234      ; Store 0x1234 to the address contained in r2
mov [r2+0x1234], r1   ; Store r1 to the address 0x1234+r2
mov [r2+r3], r1       ; Store r1 to the address r2+r3
mov [r2+r3], 0x1234   ; Store 0x1234 to the address r2+r3
```

The mov instruction is to copy the value from the second operand to the first operand. If the operand is enclosed by square brackets (“[“, “]”), the value at the memory location at the effective address within the brackets is used.

ALU operations
add, adc, sub, sbc, shl, rol, shr, ror, and, or, xor, neg
ads, adcs, subs, sbcs, shls, rols, shrs, rors, ands, ors, xors, negs
cmp, bit

```
add r1, r2      ; Add r1 and r2, store the result to r1
adc r1, r2      ; Add r1 and r2 with carry
sub r1, r2      ; Subtract r1 with r2
sbc r1, r2      ; Subtract r1 with r2 with carry
shl r1          ; Shift r1 left by one
shr r1          ; Shift r1 right by one
rol r1          ; Shift r1 left by one, bit shifted in is the carry flag
ror r1          ; Shift r1 right by one, bit shifted in is the carry flag
and r1, r2      ; Bitwise AND r1 with r2
or r1, r2       ; Bitwise OR r1 with r2
xor r1, r2      ; Bitwise XOR r1 with r2
neg r1          ; Generate the two’s compliment of r1
cmp r1, r2      ; Compare r1 with r2
```

The ALU operations are for arithmetic and logic, with the combination of these functions operations like multiplication and division can be performed. Non storing versions of the instructions have an “s” appended to

them, these variations are for testing values and bits without overwriting any registers. When comparing two values it is suggested to use the compare instruction for readability of the assembly code; subs exists to keep continuity with the other operations.

The result of the cmp instruction is the states in the status register, carry (c), zero (z) and negative (n). The combination of these flags indicate whether the register contents are less than, equal, or greater than. A table below summarizes the what each flag indicates.

Compare result	N	Z	C
op1 < op2	*	0	0
op1 = op2	0	1	1
op1 > op2	*	0	1

* The N flag will be bit 15 of op1-op2

Stack operations
push, pop

```
push r1    ; Push r1 to the stack
push 10    ; Push the value 5 to the stack
pop r1     ; Pop the top value of the stack into r1
```

Subroutines
jsr, rts

```
jsr 0x1234 ; Jump to the subroutine at address 0x1234
jsr r1     ; Jump to the subroutine at the address contained in r1
rts        ; Return from a subroutine back to the latest jsr instruction
```

Subroutines are a way to have a piece of code that can be reused many time from different locations. When a subroutines is called via jsr, the program counter is pushed to the stack and code execution jumps to the subroutine. When a subroutine returns via an rts, the top value of the stack is stored into the program counter.

Interrupts
brk, rti, int

```
brk        ; Trigger interrupt
rti        ; Return from and interrupt
int 0x8000 ; Set the interrupt routine vector to 0x8000
int r1     ; Set the interrupt routine vector to the contents of r1
```

Interrupts are a way to force the cpu to perform a different task at any point during execution. An interrupt is triggered when the interrupt disable bit is cleared in the status register and the interrupt pin is pulled high. When triggered, the brk instruction is force loaded into the instruction register, the program counter is pushed to the stack and the program counter is set to the value set by the int instruction. The brk instruction can be used directly in code too. When returning form an interrupt, rti must be used to clear the interrupted state of the cpu.

Setting and clearing flags
clc, sec, clz, sez, cln, sen, cli, sei, clxa, sexa, clxb, sexb, clxc, sexc, clxd, sexd

```
clc  ; Clear the carry flag
sec  ; Set the carry flag
…
```

These instructions clears and sets flags in the status register. The flag clearing/setting instructions get expanded to a bitwise AND or bitwise OR with the status in the machine code.