STOL: Programmer's Reference Manual

Dakotah Lambert

$\beta \cdot 25\text{C}$

The information in this manual has been reviewed and is believed to be entirely reliable. However, the author does not assume any liability arising out of the application or use of any product, software, or hardware described herein. The material in this manual is for informational purposes only and is subject to change without notice.

Dakotah Lambert, 2025

Contents

List of Figures

List of Tables

Quick Reference

Opcode	Assembly	Description	C	V	N	Z
C	`mov` *dest*`,`*source*	Move	–	–	–	–
		Arithmetic				
6	`add` *dest*`,`*source*	Add	*	*	*	*
7	`addc` *dest*`,`*source*	Add with Carry	*	*	*	*
8	`cmp` *dest*`,`*source*	Compare	*	*	*	*
31	`neg` *dest*`[,`*source*`]`	Two's-Complement Negate Register	*	*	*	*
4	`sub` *dest*`,`*source*	Subtract	*	*	*	*
5	`subb` *dest*`,`*source*	Subtract with Borrow	*	*	*	*
		Logic				
A	`and` *dest*`,`*source*	Bitwise Logical AND	–	–	*	*
9	`or` *dest*`,`*source*	Bitwise Logical Inclusive-OR	–	–	*	*
B	`xor` *dest*`,`*source*	Bitwise Logical Exclusive-OR	–	–	*	*
		Rotations and Shifts				
34/35	`asl` *dest*`,`*source*	Arithmetic Shift Left	*	*	*	*
36/37	`asr` *dest*`,`*source*	Arithmetic Shift Right	*	0	*	*
32/33	`lsr` *dest*`,`*source*	Logical Shift Right	*	0	*	*
3C/3D	`rlc` *dest*`,`*source*	Rotate Left Through Carry	*	0	*	*
38/39	`rol` *dest*`,`*source*	Rotate Left	0	0	*	*
3A/3B	`ror` *dest*`,`*source*	Rotate Right	0	0	*	*
3E/3F	`rrc` *dest*`,`*source*	Rotate Right Through Carry	*	0	*	*
		Stack Operations				
14	`pop` *dest*	Pop from Stack	–	–	–	–
1A	`pspr` *dest*	Read Program Stack Pointer	–	–	–	–
1B	`psprw` *dest*`,`*source*	Read and Set Program Stack Pointer	–	–	–	–
19	`pspw` *dest*	Set Program Stack Pointer	–	–	–	–
11	`push` *source*	Push to Stack	–	–	–	–
		Control Flow				
04–07	`ba[.`*cond*`]` *address*	Branch Absolute	–	–	–	–
00–03	`br[.`*cond*`]` *offset*	Branch Relative	–	–	–	–
08–0B	`call[.`*cond*`]` *address*	Branch to Subroutine	–	–	–	–
0D*x*0	`ret[.`*cond*`]`	Return from Subroutine	–	–	–	–
0D*x*F	`rtp[.`*cond*`]`	System Return to Program	–	–	–	–
D	`trap[.`*cond*`]`	Software Interrupt	–	–	–	–
		Status Register				
28–2B	`andsr` *dest*`,`*source*	Logical AND into Status	*	*	*	*
20–23	`movsr` *dest*`,`*source*	Replace Status	*	*	*	*
24–27	`orsr` *dest*`,`*source*	Logical Inclusive-OR into Status	*	*	*	*
2C–2F	`xorsr` *dest*`,`*source*	Logical Exclusive-OR into Status	*	*	*	*

Condition Codes

Code	Name	Test	Alt
0000		$\top$	
0001	u<	C	c
0010	v	V	
0011	n	N	
0100	=	Z	z
0101	u<=	$C \vee Z$	
0110	s<	$N \oplus V$	
0111	s<=	$(N \oplus V) \vee Z$	

Code	Name	Test	Alt
1000	f	$\bot$	
1001	u>=	$\neg C$	cc
1010	nv	$\neg V$	
1011	p	$\neg N$	
1100	!=	$\neg Z$	nz
1101	u>	$\neg C \wedge \neg Z$	
1110	s>=	$\neg N \oplus V$	
1111	s>	$(\neg N \oplus V) \wedge \neg Z$	

Registers

- R_0–R_{15}: general purpose registers, R_{15} is system (SSP) or program (PSP) stack pointer
- Flags: Carry (SR:3 = C), Overflow (SR:2 = V), Negative (SR:1 = N), Zero (SR:0 = Z)

Addressing Modes

	Syntax	Mode	Meaning
00	i	immediate	the given value i
01	R_n	register direct	contents of register R_n
10	(R_n)	register indirect	value in memory addressed by R_n
11	$(R_n + i)$	register indirect with offset	value in memory addressed by $R_n + i$

R_n is any general purpose register and i is an integer value.

Additional Assembly Tools

- *`label`*`:` *`instruction`* — assign label
- `dw` *`num`*`[,`*`num`*`...]` — insert 16-bit words
- `res` *`num`* — reserve *`num`* words
- `/define` *`name num`*
- `/BSS` — switch to BSS segment
- `@`, `@+`*`n`*, and `@–`*`n`* — current location ± *n*
- Semicolon (`;`) — comment to end of line
- `'c'` characters and `"string"` strings

Synthetic Instructions

Instruction	Meaning	Equivalent
`halt`	Stop Processor	`br @`
`nop`	No Operation	`br @+1`
`not` *`d`*	Bitwise Logical NOT	`xor` *`d`*`,0xFFFF`

Sample Program

Multiply $R_0 \cdot R_1$ into R_0.

```
      mov     r2,1
      mov     r3,r0
      xor     r0,r0
loop: mov     r4,r2
      and     r4,r1
      br.z    skip
      add     r0,r3
skip: asl     r3,1
      asl     r2,1
      br.cc   loop
```

Chapter 1

Architecture Overview

The Subsettable Tiered Operations for Learning (STOL) architecture provides a simple CISC instruction set. It is intended to be implemented in an educational setting, beginning with a core of single-cycle instructions, then progressing to add the state machine required to support advanced features such as memory access and more addressing modes.

1.1 Programmer's Model

The STOL architecture provides a two-address instruction set with a variety of addressing modes. There are two operating modes: system mode, and program mode. At reset, the processor begins in system mode. However, most programs are generally expected to be run in the less privileged program mode.

1.1.1 Registers

There are sixteen 16-bit general-purpose registers, named R_0–R_{15}, where R_{15} is also known as SP, the stack pointer. The program counter (PC) indicates the current location, and the status register (SR) holds the active condition codes. Only the low four bits of the status register are writable. The user's model is shown in Figure 1.1.

The system programmer's model includes two supplementary registers. In system mode, the program stack pointer (PSP) is shadowed by the system stack pointer (SSP) as register R_{15}. The distinction is made so that user programs can freely manipulate all sixteen general-purpose registers without risk to an operating system. Special instructions exist to allow system code to access and manipulate the PSP as well. Further, the upper bits of the status register, which include the operating mode, interrupt nesting level and active interrupt mask, become writable. These additional features are shown in Figure 1.2.

1.1.2 Initial State

At reset, the program counter is zero and the processor is in system mode at nesting level zero. No further assumptions may be made. Namely, the general-purpose registers, the stack pointers, and the remainder of the status register are in an indeterminate state.

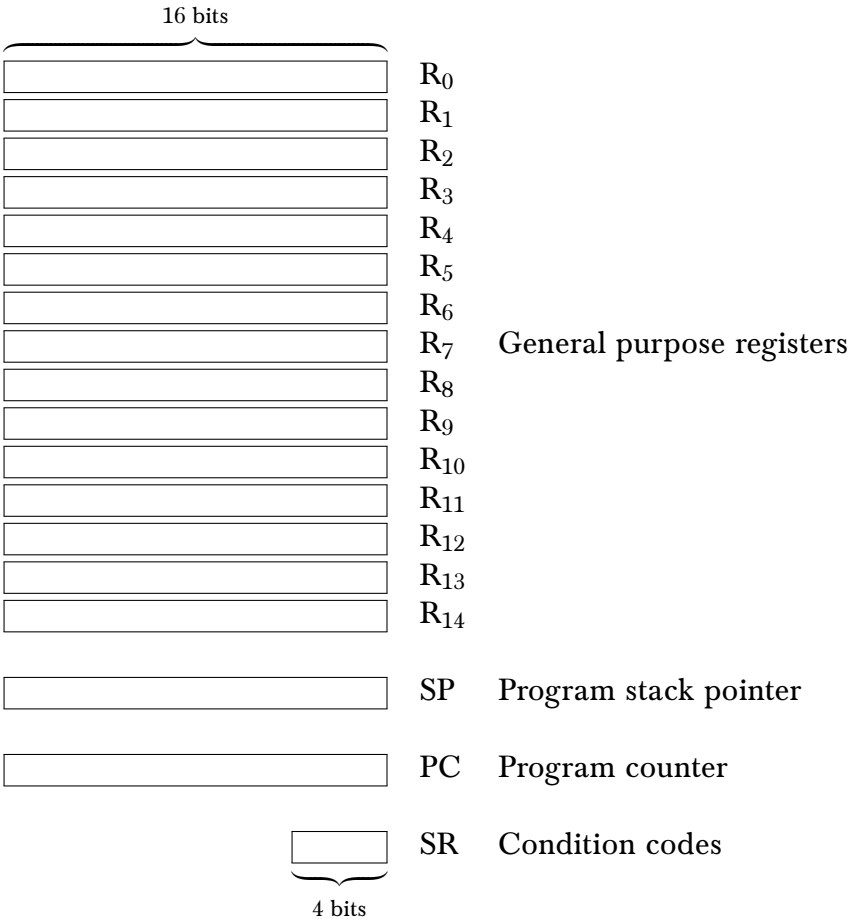


Figure 1.1: User Programmer's Model

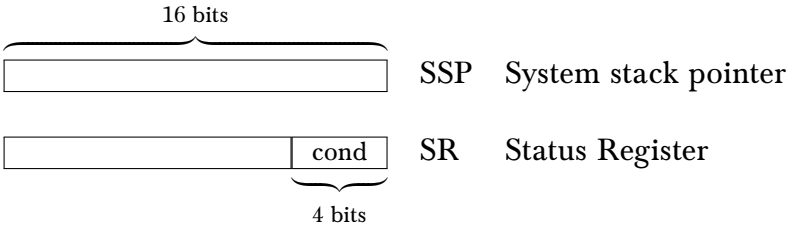
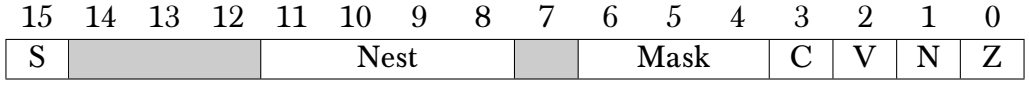


Figure 1.2: Supplements in the System Programmer's Model

1.1.3 Flags and Conditions

The status register is depicted in Figure 1.3. The four bits stored in the condition code portion of the status register are as follows: zero (Z = SR:0), set if the result of an operation is zero, negative (N = SR:1), set if the result of an operation has its most-significant bit set and is thus negative in two's complement signed interpretation, overflow (V = SR:2), set if the result of an arithmetic operation exceeded the available precision and is incorrect under a signed interpretation, and carry (C = SR:3), the unsigned analog of signed overflow. Not all operations affect all flags; see the individual instruction listings for full detail.



S: Program mode (0) or system mode (1)
Mask: Minimum interrupt priority for service
Nest: Interrupt nesting level
CVNZ: Condition flags

Figure 1.3: The Status Register

All control-flow instructions are conditionally executed. If the indicated condition is not satisfied, then the instruction is passed over without effect. The condition codes are summarized in Table 1.1. Each of the four condition flags can be queried individually, and tests are provided for each of the six mathematical comparisons ($<$, $\leqslant$, $=$, $\geqslant$, $>$, and $\neq$) for both signed and unsigned interpretations. Additionally, there are codes to unconditionally execute or skip the instruction.

Table 1.1: Condition Codes

Code	Name	Test	Alt	Code	Name	Test	Alt
0000		$\top$		1000	f	$\bot$	
0001	u<	C	c	1001	u>=	$\neg$C	cc
0010	v	V		1010	nv	$\neg$V	
0011	n	N		1011	p	$\neg$N	
0100	=	Z	z	1100	!=	$\neg$Z	nz
0101	u<=	C $\vee$ Z		1101	u>	$\neg$C $\wedge$ $\neg$Z	
0110	s<	N $\oplus$ V		1110	s>=	$\neg$N $\oplus$ V	
0111	s<=	(N $\oplus$ V) $\vee$ Z		1111	s>	($\neg$N $\oplus$ V) $\wedge$ $\neg$Z	

1.1.4 Memory System

The STOL system is designed as a Princeton architecture, with instruction memory and data memory sharing the same buses and address space. Up to 64K words (128KB) of memory can be addressed at a time. The memory at address zero should be non-volatile, as this is the initial program counter upon reset. The stack pointers should be placed such that pushing a value will use writable memory.

The initial segment of memory must be nonvolatile, as execution begins at address zero upon reset. Additionally, the interrupt handling mechanism executes the contents of memory from locations 0x00E0–0x00F0. Therefore, the recommended memory configuration has a system boot ROM spanning addresses 0x0000–0x00FF, followed by RAM spanning addresses 0x0100–0x7FFF, and main program memory spanning addresses 0x8000–0xFFFF. The main program memory might be ROM for fixed-program systems, or RAM for systems that load programs from external memory. If the system should be connected to external I/O devices, these devices should be memory-mapped and shadow RAM, perhaps dedicating address 0x0100–0x200 for this purpose.

1.1.5 Data Addressing Modes

Most instructions specify a source operand and a destination operand. In order to express programs compactly, four addressing modes are provided.

Immediate mode comes in two variants. Short, nonzero immediate values (1–15, inclusive) can be directly encoded into the instruction word. Other values must be encoded as an extension word. In either case, this data is used directly as the operand. As this is not writable, this addressing mode is illegal for destinations. All others are generally legal.

Register-direct mode uses the contents of a general-purpose register as the operand.

Register-indirect mode uses the contents of a general-purpose register as a pointer into memory. The contents of this memory location are then used as the operand.

Register-indirect with offset mode embeds the offset as an extension word. The sum of this offset with the contents of a general-purpose register is used as a pointer into memory, and the contents of this memory location are then used as the operand.

These modes are summarized in Table 1.2. If both the source operand and the destination operand require an extension word, the source precedes the destination.

Table 1.2: Data Addressing Modes

	Syntax	Mode	Meaning
00	i	immediate	the given value i
01	R_n	register direct	contents of register R_n
10	(R_n)	register indirect	value in memory addressed by R_n
11	$(R_n + i)$	register indirect with offset	value in memory addressed by $R_n + i$

R_n is any general purpose register and i is an integer value.

Some instructions implicitly refer to the program counter, status register, or program stack pointer. These instructions along with their implicit references are listed in Table 1.3.

1.2 Interrupt Processing

External interrupts are available in eight priority levels. By manipulating the status register, the minimum priority level can be set anywhere from 0–7. This means that priority levels 0–6 can be masked away, while level 7 is non-maskable. At the beginning of instruction processing,

Table 1.3: Instructions with Implicit References

Instruction		Implied Register(s)
Logical AND into Status	`ANDSR`	SR
Branch Absolute	`BA`	PC
Branch Relative	`BR`	PC
Branch to Subroutine	`CALL`	PC, SP
Replace Status	`MOVSR`	SR
Logical Inclusive-OR into Status	`ORSR`	SR
Pop from Stack	`POP`	SP
Read Program Stack Pointer	`PSPR`	PSP
Set and Read Program Stack Pointer	`PSPRW`	PSP
Set Program Stack Pointer	`PSPW`	PSP
Push to Stack	`PUSH`	SP
Return from Subroutine	`RET`	PC, SP
Return to Program	`RTP`	PC, PSP, SSP
Software Interrupt	`TRAP`	PC, SSP
Logical Exclusive-OR into Status	`XORSR`	SR

the system detects whether an unmasked interrupt is requested. If so, then instead of executing the current instruction, the interrupt is handled.

This proceeds as follows. First, the processor enters system mode and increments the current interrupt nesting level. Then, the program counter is pushed to the system stack, and execution proceeds at address $224 + 2p$, where p is the priority level of the interrupt. The interrupt is then acknowledged. As there are only two words available per priority level, generally these locations should contain either a `HALT` instruction to stop the processor, an `RTP` instruction to ignore the interrupt, or a branch to another location for further processing.

Additionally, program software may raise an internal interrupt by using a `TRAP` instruction. There are eight software interrupts available. Servicing follows the same pattern as external interrupts, except that the external acknowledge pin is not asserted, and execution proceeds at address $240 + 2v$ where v is the interrupt number.

To simplify the implementation, interrupt processing does not duplicate the status register before entering system mode and executing the appropriate handler. Returning from an interrupt handler to a user program should exit system mode. This combination means that nested interrupts cannot be safely handled without additional work. The STOL architecture uses a portion of the status register to indicate the current nesting level, to determine whether to retain system privileges. Due to the limited size of this field, up to sixteen nested interrupts can be safely processed. If it overflows, the processor may relinquish privileges prematurely.

As interrupts may occur at any time, including at arbitrary points during the processing of user programs, it is important that interrupt handler routines preserve essential state. If any registers or condition codes are modified, the relevant data should be pushed to the stack upon entry and popped upon exit. In system mode, the entire status register is writable, but it is generally a bad idea to modify the mode bit or interrupt nesting level bits outside of special circumstances.

1.3 Recommended Startup Procedure

The initial boot code should configure the program and system stack pointers, and initialize any external hardware. To leave system mode and enter a program, push the target address and issue an RTP instruction. The following sample code initializes the program stack pointer to 0x7000 and the system stack pointer to 0x8000, then enters a target program at address 0x8000. This should be sufficient for any system that needs no external device setup and which follows the recommended memory configuration with fixed program ROM in the upper half of memory.

```
mov     sp,0x7000     ; c4f0 7000
pspw    sp            ; 190f
mov     sp,0x8000     ; c4f0 8000
push    sp            ; 110f
rtp                   ; 0d0f
```

Chapter 2

Binary Numbers and Arithmetic

This chapter introduces the arithmetic instructions available in the STOL architecture. For information on the assembly language format, see Appendix B. For a complete instruction listing, see Appendix C.

Decimal numbers use digits that range in value from 0–9. This is a *positional number system* (also called a *place–value* system), where the value contributed by a digit corresponds to its location. The number 329 represents three hundreds plus two tens plus nine, $300+20+9$. That is, $3 \cdot 10^2 + 2 \cdot 10^1 + 9 \cdot 10^0$. The most-significant digit is the leftmost, and the least-significant digit is the rightmost. As computers tend to work with signals that have just two states, either *off* or *on*, numbers are more naturally represented with just two kinds of digit: 0 and 1. In the decimal system, positions are valued in powers of ten, while in the binary system, they are valued in powers of two.

2.1 Addition

When two numbers are added, the addition proceeds one place at a time, from the least-significant position up through the most-significant. If the sum is larger than will fit in a digit, the number of available digits (10 for decimal) is subtracted and an additional 1 is *carried* into the next position:

$$\begin{array}{cccc} & {\scriptstyle 1} & & {\scriptstyle 1} \\ & 3 & 2 & 9 \\ + & 8 & 5 & 3 \\ \hline 1 & {\scriptstyle 1}1 & 8 & {\scriptstyle 1}2 \end{array}$$

The same applies to binary numbers, except that the largest possible digit is 1.

$$\begin{array}{cccc} {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & \\ 1 & 0 & 1 & = 4+1=5 \\ + \quad 1 & 1 & 1 & = 4+2+1=7 \\ \hline 1 & {\scriptstyle 1}1 & {\scriptstyle 1}0 & {\scriptstyle 1}0 \end{array}$$

In the STOL architecture, all numbers involved have 16 positions. Every addition takes two numbers with 16 positions as input and returns a number with 16 positions as output. This

is called *single-precision* arithmetic. What it means is that adding particularly large numbers might exceed the available precision and yield a result that differs from the mathematically correct sum. If a one is carried out of the most-significant position, this is tracked by the system in the *carry flag*.

			1	1	1	1		1		1	1	1	1	1	1	1	
			1	1	1	1	1	0	1	0	1	1	0	0	1	1	1	0
C	+		1	1	0	0	1	0	1	0	1	1	1	1	1	1	1	0
1			1	1	1	0	1	0	1	0	1	1	1	0	1	1	1	0	0

```
mov     r0,42
mov     r1,78
add     r0,r1           ; r0 correctly contains 120
mov     r0,64206
mov     r1,51966
add     r0,r1           ; r0 holds 50636 instead of 116172
```

The "add with carry" instruction, "addc *destination*,*source*", enables higher-precision arithmetic by treating another source/destination pair as a continuation of the operands, adding in the value of the carry flag alongside the first, least-significant position. The highest possible number that can be represented in sixteen positions is $65{,}535$, so a value used as a continuation is worth $65{,}536$ times its actual value.

```
mov     r0,64206
mov     r1,0            ; continuing r0
mov     r2,51966
mov     r3,0            ; continuing r2
add     r0,r2           ; r0 holds 50636 instead of 116172, C set
addc    r1,r3           ; r1 holds 1; 50636 + 1·65536 = 116172
```

Using just one continuation value is *double-precision* arithmetic. Rather than just 16 positions per value, 32 positions are available. This pattern extends to higher precision: add from least-significant up through most-significant, accounting for the carry flag on all additions except for the first. However, with particularly large numbers, one may struggle to fit them into the available registers. Using an indirect addressing mode can alleviate this concern. For instance, one can add two quadruple-precision numbers x stored at *xaddr* and y stored at *yaddr* as follows:

```
mov     r0,xaddr
mov     r1,yaddr
add     (r0),(r1)
addc    (r0+1),(r1+1)
addc    (r0+2),(r1+2)
addc    (r0+3),(r1+3)
```

2.2 Signed Numbers & Two's-Complement Negation

To this point, discussion has centered on *unsigned arithmetic*. However, sometimes it is useful to represent negative numbers. Given a number n, its negation, $-n$, is the number that must be added to it in order to reach zero. Consider adding n to the number n' obtained from n by exchanging 0 and 1 in each position. In this addition, every position will result in a sum of 1 with no carry into the next position. That is, the result will be the number where all positions are 1. Adding 1 into this result will yield a sum of 0 in the first position with a carry into the next. This next position then also yields a sum of 0 and a carry, and this continues until the addition is complete; the final sum is 0 with the carry flag set. In some sense, n' is $-n$.

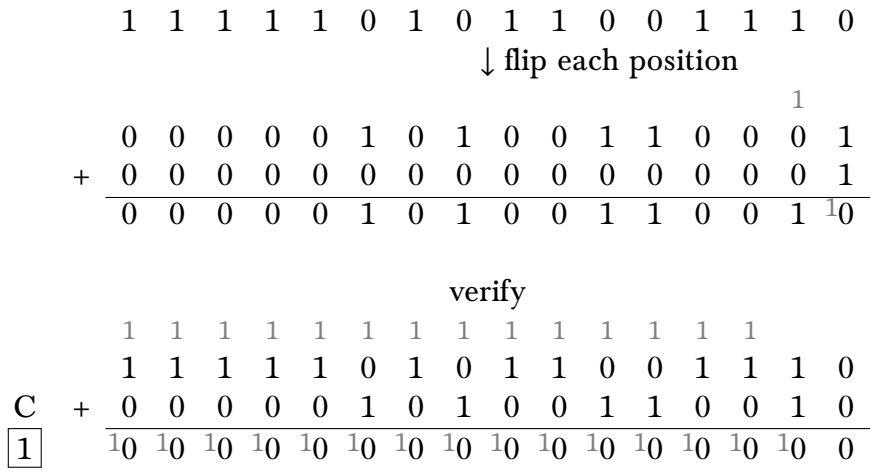


Figure 2.1: Negation

By convention, when a number is treated as a signed number, the most-significant bit is treated as a *negative indicator*, a sign: if it is 1 then the number is negative; otherwise it is nonnegative. It turns out that in this interpretation, computing the value of the number is as simple as treating the most-significant bit as being worth the negative of its usual value. So instead of the most-significant bit being worth $2^{15} = 32768$, it is instead worth $-2^{15} = -32768$. Unsigned 16-position numbers range in value from 0 through 65,535, while signed 16-position numbers range from −32,768 through 32,767. This is called the *two's complement* representation, because in effect it computes $-n$ as $2^{16} - n$.

The instruction "neg *destination*,*source*" places the negation of the *source* operand into *destination*.

2.3 Subtraction

To compute the subtraction $a - b$, one can compute the addition $a + (-b)$. If b' is the number obtained by flipping each position of b, this is $a + b' + 1$. Using this computation, the carry flag will be set whenever the unsigned interpretation of a is greater than or equal to that of b. To extend this to higher precision, notice that the carry flag indicates that *no borrow* is needed; if the carry flag is unset, then a one needed to be borrowed out of the least-significant position of the continuation, so the computation is only $a + b'$. If the carry flag is set, then no

borrow was needed and the computation is $a + b' + 1$. In other words, subtraction with carry is $a + b + \mathrm{C}$, where C is the carry flag.

However, the STOL architecture internally swaps the state of the carry flag into and out of a subtraction operation, for reasons that will be elaborated in the next section. That is, rather than treating it as a "not borrow" indicator, the flag is treated as a "borrow" indicator. Both conventions are widely used by various architectures. Subtraction with and without borrow are computed with the "`subb` *`destination`*`,`*`source`*" and "`sub` *`destination`*`,`*`source`*" instructions, respectively.

2.4 Overflow

The carry flag is used to indicate when the result is not mathematically correct for the *unsigned* interpretation of the operation. Note that this means that negation sets the carry flag for *all* nonzero values, as negative numbers cannot be represented as unsigned values. This is why the carry flag is inverted internally to be treated as "borrow" when doing subtraction operations. An additional flag, overflow (V = SR:2), indicates when the result is not mathematically correct for the *signed* interpretation of the operation. When the mathematically correct result exceeds 32,767 or falls below −32,768, the overflow flag will be set.

When performing an addition between numbers of opposite signs, overflow can never occur. But when both inputs are positive, in order to be correct, the sum must be less than 32,768. The sum of the largest positive signed number (32,767) with itself is 65,534, so there will never be a *carry* out. This means that the sum of two positive numbers results in overflow only when the result is negative. Similarly, an overflow results from adding two negative numbers only when the result is positive. In short, overflow occurs when the inputs have the same sign and the result has the opposite sign. For the subtraction $a - b$, overflow is computed as it would be for $a + (-b)$.

Note that the two's complement representation ensures that even higher-precision arithmetic works identically for signed and unsigned values, with the only difference being whether the programmer should investigate the carry flag or the overflow flag after the final step in the operation. Internal steps use only the carry flag in either case.

2.5 Other Flags

The negative (N = SR:1) flag is set when the most-significant bit of a result is set, when the result is negative under signed interpretation. The zero (Z = SR:0) flag is set when the result is zero.

2.6 Comparisons

After performing the subtraction operation "`sub` *`a`*`,`*`b`*", the condition codes are set to properly reflect how *a* compares to *b*. Each of the six mathematical comparisons ($<$, $\leqslant$, $=$, $\geqslant$, $>$, and $\neq$) is available through one of the condition codes for both signed and unsigned interpretations.

Importantly, this operation also modifies *a*. An instruction is provided to compute the flags in the same way but leave *a* intact: `cmp a,b`.

2.7 Other Operations

It is left as an exercise to determine how to compute higher-precision variants of other operations such as "negate", or operations that are not even provided in single-precision form such as multiplication and division. The advantage of the two's complement representation is that addition and subtraction are computed by the same operation whether the operands are to be treated as signed values or as unsigned values. However, this is not necessarily the case for other operations. For instance, signed and unsigned multiplication differ if computed to full precision.

Table 2.1: Arithmetic Instructions

Encoding		Instruction
6*mds*	ADD	Add
7*mds*	ADDC	Add with Carry
31*ds*	NEG	Two's-Complement Negate Register
4*mds*	SUB	Subtract
5*mds*	SUBB	Subtract with Borrow
8*mds*	CMP	Compare

Chapter 3

Bit Manipulation & Boolean Logic

This chapter describes logical operations and the STOL operations that compute them. The fundamental unit is the *bit* (binary digit), which may be either set (1, true) or cleared (unset, 0, false). Occasionally, base-16 *hexadecimal notation* will be used, as conversion between binary and hexadecimal is only a matter of grouping bits:

0	1	2	3	4	5	6	7
0000	0001	0010	0011	0100	0101	0110	0111

8	9	A	B	C	D	E	F
1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal numbers are indicated by the prefix "0x". For example, the number 0xFACE represents 1111 1010 1100 0000. The most-significant bit is sometimes also called the sign-bit, due to its meaning in signed interpretation of numbers.

3.1 Data Movement Operation

The primary instruction for copying data between locations is "`mov` *`destination`*`,`*`source`*". This allows for transfer between general-purpose registers, between one general-purpose register and main memory, or from a directly provided integer into either a general-purpose register or main memory. Special instructions are provided for managing other registers.

3.2 Logical AND

The logical AND of two given bits a and b, written $a \wedge b$, is set if both a and b are set; otherwise it is cleared. The `AND` instruction, "and *`destination`*`,`*`source`*", computes this operation in parallel bit-by-bit between its destination and source operands, then stores the result back into the destination.

The C and V flags are never modified by this instruction. The N flag is set if the most-significant bit of the result is set, and the Z flag is set if no bits are set in the result.

The `AND` instruction has two main uses. First, it can clear specified bits in the destination by using a source where all bits are set except for the specified bits. For example, the instruction

$$\begin{array}{rcccc} & \ldots & 1 & 1 & 0 & 0 \\ \wedge & \ldots & 1 & 0 & 1 & 0 \\ \hline & \ldots & 1 & 0 & 0 & 0 \end{array}$$

Figure 3.1: Bitwise Logical AND Operation

"and `r0,0xFFFE`" will clear the least-significant bit of register R_0. Second, it can be used to test if a specific bit is set in a source, by using a destination where all bits are cleared except for the one to test. For example, to test whether the most-significant bit of register R_0 is set, one can first issue the instruction "`mov r1,0x8000`" to set up the destination, and then use "`and r1,r0`". The Z flag will be cleared if that most-significant bit is set. When the destination has multiple bits set, the Z flag is cleared if *any* of the specified bits are set in the source.

3.3 Logical Inclusive-OR

The logical inclusive-OR of two given bits a and b, written $a \vee b$, is set if a is set, or if b is set, or both; otherwise it is cleared. The `OR` instruction, "`or` *`destination,source`*", computes this operation in parallel bit-by-bit between its destination and source operands, then stores the result back into the destination.

$$\begin{array}{rcccc} & \ldots & 1 & 1 & 0 & 0 \\ \wedge & \ldots & 1 & 0 & 1 & 0 \\ \hline & \ldots & 1 & 1 & 1 & 0 \end{array}$$

Figure 3.2: Bitwise Logical Inclusive-OR Operation

The C and V flags are never modified by this instruction. The N flag is set if the most-significant bit of the result is set, and the Z flag is set if no bits are set in the result.

The `OR` instruction is primarily used to set specific bits in a destination; to set bits 5 and 12 of register R_0, use "`or r0,0x1020`":

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0

3.4 Logical Exclusive-OR

The logical exclusive-OR of two given bits a and b, written $a \oplus b$, is set if a is set, or if b is set, but not both; otherwise it is cleared. In other words, the result is set if and only if the inputs differ. The `XOR` instruction, "`xor` *`destination,source`*", computes this operation in parallel bit-by-bit between its destination and source operands, then stores the result back into the destination.

The C and V flags are never modified by this instruction. The N flag is set if the most-significant bit of the result is set, and the Z flag is set if no bits are set in the result.

When a source bit is cleared, the corresponding destination bit is unchanged. When a source bit is set, the corresponding destination bit is toggled. Thus, the `XOR` instruction is

$$\begin{array}{rcccc} & \ldots & 1 & 1 & 0 & 0 \\ \wedge & \ldots & 1 & 0 & 1 & 0 \\ \hline & \ldots & 0 & 1 & 1 & 0 \end{array}$$

Figure 3.3: Bitwise Logical Exclusive-OR Operation

primarily used to toggle specific bits in a destination; to toggle bits 14 and 15 of register R_0, use "`xor r0,0xC000`":

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

3.5 Logical Complement

A bit, a, can be in one of two states. Its complement, written $\sim a$, is the other state. The assembler provides a NOT instruction synthesized from exclusive-OR: "`not` *`destination`*" is equivalent to "`xor` *`destination`*`,0xFFFF`".

3.6 Status Register

The transfer, logical AND, inclusive-OR, and exclusive-OR operations can also be performed between a source and the status register, using MOVSR, ANDSR, ORSR, and XORSR, respectively. In system mode, the entire status register is written. In program mode, only the condition flags are written. In either case, the original content of the status register is placed into the destination location.

3.7 Shifts & Rotations

Shifting the bits of a number one position to the left, placing zeros in the newly unoccupied positions, is equivalent to multiplying by two. The "`asl` *`destination`*`,`*`source`*" instruction performs this operation on the *`destination`* as many times as indicated by the *`source`* operand, setting the carry flag if *any* set bit is shifted out, and setting the overflow flag if the sign changes at any point.

Figure 3.4: Left Shift Operation

Shifting the bits one position to the right divides by two, rounding down. This operation differs depending on whether the input is signed or unsigned. The ASR instruction computes the signed interpretation, while the LSR instruction computes the unsigned interpretation. The difference lies in whether the newly opened bits are filled with copies of the sign bit (ASR) or with zero (LSR). In either case, the instruction performs this operation on the *`destination`*

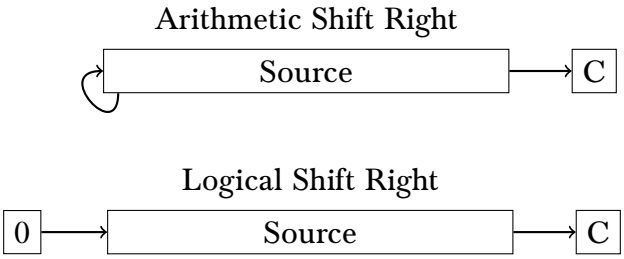


Figure 3.5: Right Shift Operations

as many times as indicated by the *source* operand, setting the carry flag if *any* set bit is shifted out, and clearing the overflow flag.

For rotations, the newly opened positions are filled not by fixed values but by the bits shifted out, in order. The STOL architecture provides leftward and rightward rotations within the *destination* operand (ROL and ROR) or through the carry flag (RLC and RRC).

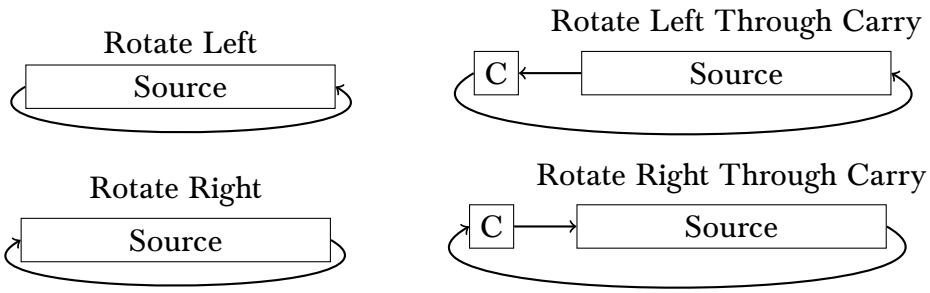


Figure 3.6: Rotation Operations

For each of these shift and rotation operations, the negative flag is set if the most-significant bit is set, and the zero flag is set if the result is zero.

Due to how the carry flag is set, shifts can be extended to higher precision by shifting only one position at a time and making use of the appropriate rotate-through-carry instructions. For a leftward shift, begin at the least-significant position and use the ASL instruction, then proceed to positions of higher significance using RLC to pull in the bit previously shifted out and to store the bit shifted out by this operation. Repeat until finished. For a rightward shift, begin at the most-significant position and use whichever of ASR or LSR is appropriate, then descend through positions of lesser significance using RRC until finished.

Table 3.1: Instructions for Boolean Logic and Bit Manipulation

Encoding		Instruction
C*mds*	MOV	Move
A*mds*	AND	Bitwise Logical AND
9*mds*	OR	Bitwise Logical Inclusive-OR
B*mds*	XOR	Bitwise Logical Exclusive-OR
†	NOT	Bitwise Logical NOT
2*mds*, $0 \leqslant m \leqslant 3$	MOVSR	Logical Exclusive-OR into Status
2*mds*, $8 \leqslant m \leqslant$ B	ANDSR	Logical AND into Status
2*mds*, $4 \leqslant m \leqslant 7$	ORSR	Logical Inclusive-OR into Status
2*mds*, C $\leqslant m \leqslant$ F	XORSR	Logical Exclusive-OR into Status
3*mds*, $4 \leqslant m \leqslant 5$	ASL	Arithmetic Shift Left
3*mds*, $6 \leqslant m \leqslant 7$	ASR	Arithmetic Shift Right
3*mds*, $2 \leqslant m \leqslant 3$	LSR	Logical Shift Right
3*mds*, $8 \leqslant m \leqslant 9$	ROL	Rotate Left
3*mds*, A $\leqslant m \leqslant$ B	ROR	Rotate Right
3*mds*, C $\leqslant m \leqslant$ D	RLC	Rotate Left Through Carry
3*mds*, E $\leqslant m \leqslant$ F	RRC	Rotate Right Through Carry

† NOT is a synthetic instruction implemented as XOR with a source of 0xFFFF

Chapter 4

Stacks & Linear Data

A stack is a data structure into which data can be *pushed* and from which data can be *popped*, where the *last* item to be pushed is the *first* to be popped. A stack can be implemented in software by storing a pointer, moving it in one direction to push data, and moving it in the other direction upon reading data. In one case, the operation should occur before the movement, and in the other, it should occur after the movement. It does not matter which of the four possible configurations is used.

Because a stack data structure is used to maintain state across subroutine calls and returns (see §5.3), the STOL architecture provides hardware that efficiently implements such a structure. There are two stack pointers, of which only one is usable in this capacity at a time: the program stack pointer (`PSP`) and the system stack pointer (`SSP`). The active stack pointer is known as `SP`. The `PSPR`, `PSPRW`, and `PSPW` instructions exist to manipulate the program stack pointer in system mode.

The `PUSH` instruction first decrements the active stack pointer and then writes the given data. The `POP` instruction reads data and then increments the active stack pointer. With this configuration, the stack grows downward, from higher addresses toward lower addresses. The stack pointer points to the *top* of the stack, the lowest address that it currently occupies.

To read the top element without popping it, the register-indirect addressing mode can be used: "`mov r0,(sp)`" reads the top item into register R_0. Register-indirect addressing with offset allows for reading prior elements without popping: "`mov r0,(sp+3)`" reads the element that would become the top after three pops.

4.1 Arrays & Structured Data

Accessing the stack elements without popping them is akin to accessing the stack as an *array*. An array is a linear sequence of data, represented by its starting address. If the starting address of an array of 16-bit words is in register R_0, then the first item is accessed via `(r0+0)`, the second by `(r0+1)`, and so on. If the elements in the array are compound objects, the indices must be multiplied by the size of the objects; for instance, if the array contained 64-bit integers represented by four 16-bit words each, the fifth element is not at `(r0+4)` but at `(r0+16)`, extending through `(r0+19)`. If the array's base address is *addr* and the index is stored in register R_1, the element can also be obtained via `(r1+`*addr*`)`.

An array holds a sequence of data of the same type (and thus the same size). One can also bundle distinct types of data, with potentially different sizes, in a *record structure*, which higher-level languages might call a *tuple*, a *struct* or a *class*. The only requirement is that the relevant offsets from the structure's base address are known. For instance, a program might represent a circle as a single-precision radius alongside double-precision x- and y-coordinates. If they are stored in this order, the radius is at offset zero from the base address, the x-coordinate at offset one (extending to two), and the y-coordinate at offset three (extending to four).

When a sequence of structured objects is to be stored in an array, one has two options. One can make a single array containing each of the structures, or one can make a structure of many arrays, one per component. One is encouraged to consider and compare the access patterns involved in each of these two layouts when reading or writing a particular field of the object at a particular index.

4.2 Strings

A string like "hello" is a sequence of character values; in ASCII this string is the sequence 104, 101, 108, 108, 111. However, this sequence is usually not stored in isolation. When laid out in the same linear stream as instructions and other data, this string on its own has no indication of its size.

The C programming language uses the convention that strings are terminated with a *sentinel* value, specifically the value 0. To encode this string as a C string, one can use:

```
dw      "hello",0
```

This representation has been a lasting source of bugs across many programs. In contrast, Pascal uses the convention that strings begin with an indication of their length. To encode as a Pascal string, one can use:

```
dw      5,"hello"
```

The assembler is not opinionated on this matter; no shortcuts are provided to facilitate the use of either representation.

Table 4.1: Instructions for Stack Manipulation

Encoding		Instruction
110*s*	PUSH	Push to Stack
15*d*0	POP	Pop from Stack
1A*d*0	PSPR	Read Program Stack Pointer
1B*ds*	PSPRW	Read and Set Program Stack Pointer
190*s*	PSPW	Set Program Stack Pointer

Chapter 5

Control Flow & Structured Programming

In assembly language, all deviations from linear control flow are of the form "if this condition holds, go to this instruction". When abused, this results in a structureless meandering through program memory often called *spaghetti code*. This chapter describes tips and techniques for avoiding the spaghetti by mapping some concepts from higher-level programming languages into assembly language.

Higher-level languages generally use structured programming concepts, including if–then–else blocks, while-loops, for-loops, and function calls to simplify program analysis. Throughout this chapter, the relevant structure will be indicated as comments beside the equivalent assembly code. This can be a helpful exercise when structuring a program.

The section on subroutine calls describes a variety of methods for passing parameters and returning values. Higher-level languages may refer to these as *functions* or *procedures*. More complex transfers of control, such as for generators or other *coroutines*, are beyond the scope of this manual.

5.1 Conditional Execution

In a higher-level language, conditional execution might by represented by an if–then–else block structured like the following:

```
If condition Then
   ...true case...
Else
   ...false case...
EndIf
```

A conditional branch to a future label skips the code between that branch and that label if the condition is satisfied. So an assembly-language programmer has two choices: place the *false case* first and use the given condition, or place the *true case* first and invert the condition. In either case, whichever case comes first should end with a branch instruction skipping over the other case. Concretely, consider the following high-level structure, also depicted as a flowchart in Figure 5.1, that retrieves an element from an `array` if the index (R_1) is within the bounds (R_2), or yields -1 otherwise.

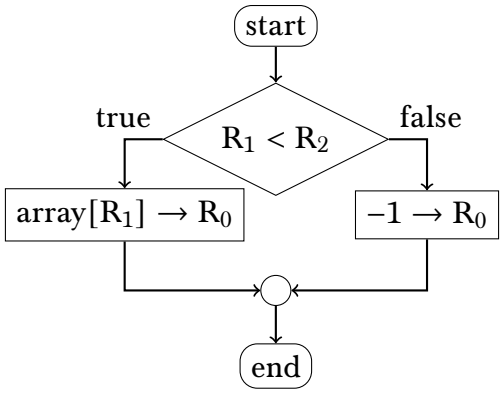


Figure 5.1: Array Bounds Checking

```
If R1 < R2 Then
    array[R1] → R0
Else
    −1 → R0
EndIf
```

A direct conversion to assembly code is as follows:

```
      cmp     r1,r2          ; If R1 < R2 Then
      br.u>=  false          ;   // inverted condition keeps true first
true: mov     r0,array       ;   array[R1] → R0
      add     r0,r1
      mov     r0,(r0)
      br      endif          ; Else
false:mov     r0,-1          ;   −1 → R0
endif:                       ; EndIf
```

If there is no `Else` and thus no *false case*, the branch at the end of the *true case* can be omitted for shorter, faster code. For instance, the following code computes the unsigned absolute value of a signed integer in register R_0.

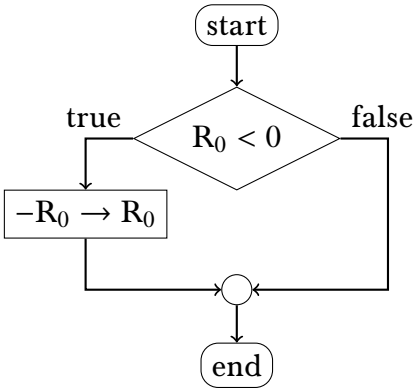


Figure 5.2: Absolute Value

```
        or      r0,r0           ; If R0 < 0 Then
        br.p    end             ;   // inverted condition
        neg     r0              ;   −R0 → R0
end:                            ; EndIf
```

5.2 Looping Control Flow

A while-loop is a lot like an if–then–else block, except that the end of the *true case* jumps not ahead of the *false case*, but instead back up to the conditional test. Consider the problem of finding the sum of an array stored in memory, depicted in Figure 5.3, written as follows.

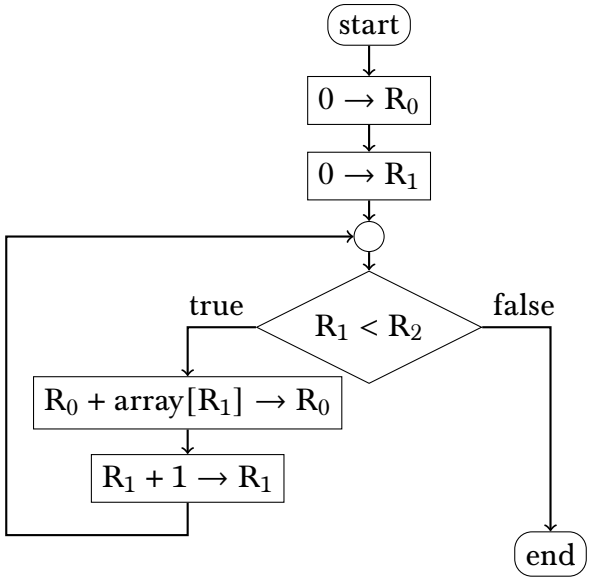


Figure 5.3: Array Sum

```
        mov     r0,0            ; 0 → R0
        mov     r1,0            ; 0 → R1
loop:   cmp     r1,r2           ; While R1 < R2
        br.u>=  end
        mov     r3,array        ;   R0 + array[R0] → R0
        add     r3,r1
        add     r0,(r3)
        add     r1,1            ;   R1 + 1 → R1
        br      loop            ; EndWhile
end:
```

The preceding examples used the BR instruction, for *relative* branches. In this case, the program code contains not the actual target address, but its offset from the current location. Using BA, for *absolute* branches, would store the actual target address directly. On some systems, relative branches are smaller and faster, but compromise in only being able to store relatively short offsets. On the STOL system, this is not the case: absolute and relative targets

generally require the same amount of time, and either can cover the entire address space. However, one reason to prefer relative branches is that the resulting machine code executes identically regardless of where it is loaded in memory; this simplifies building programs from composable parts.

5.3 Subroutines & Function Calls

A simple procedure that takes no parameters and returns no result, such as a procedure that updates some external hardware, can be written and used by placing a label at its first instruction and a return (`RET`) as its final instruction, then issuing a `CALL` to that label. The `CALL` instruction pushes the location of the next instruction to the stack and then transfers control to the given subroutine. The `RET` instruction later pops the location off of the stack and restores control back to that point. This is depicted in Figure 5.4.

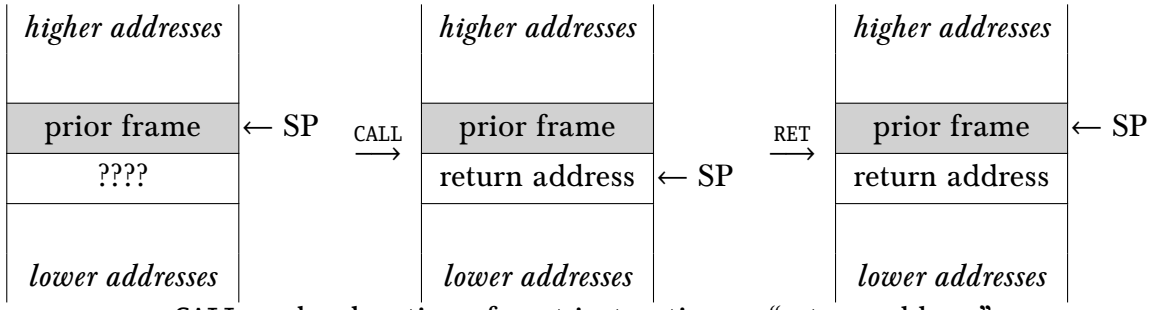


Figure 5.4: Stack Operations for Call & Return

For more complex functions, a system must define an *application binary interface* (ABI) that describes how parameters and return values are exchanged. This section describes a few potential solutions, including the use of registers or the stack to exchange information.

Care must be taken to ensure that the return address is not overwritten and that the stack pointer is in the right place when the `RET` instruction is reached. Failing to do so can cause unpredictable results, as program execution will resume not from the instruction following the `CALL`, but from whatever address finds itself at the top of the stack upon return. The stack pointer is a general-purpose register, and it can be manipulated freely just like any of the others. And the `PUSH` and `POP` instructions allow the stack pointer to be used to efficiently traverse linear structures. But if this is done, it must be returned to the correct state later.

As an example of a simple procedure, suppose that a system is configured with a terminal output device mapped into memory at address 0x100. The terminal will display any value written to this address, interpreting its seven least-significant bits as an ASCII character. The display is cleared by sending a form feed character (ASCII 0xC). The following program calls a procedure that prints the word “hello” to the terminal output. This program also demonstrates using the stack pointer to iterate through an array. This makes use of the fact that the stack grows downward; to push is to decrease the pointer and then write data, and to pop is to read data and then increase the pointer.

```
      call    _greet          ; greet()
      halt
_greet:                       ; Procedure greet()
      mov     r0,0x100
      mov     r3,sp           ;   SP → R3 // backup
      mov     sp,str          ;   "hello" → SP
      pop     r1              ;   (SP+) → R1 // length
loop: or      r1,r1           ;   While R1 ≠ 0
      br.z    end
      pop     r2              ;       (SP+) → (Terminal)
      mov     (r0),r2
      sub     r1,r1           ;       R1 − 1 → R1
      br      loop            ;   EndWhile
end:  mov     sp,r3           ;   R3 → SP // restore
      ret                     ; EndProcedure
str:  dw      5,"hello"       ; // length-prefixed string
```

5.3.1 Passing Information via Registers

One way to exchange information between a calling routine (the "caller") and the subroutine so called (the "callee") is to place the parameters and the return value into specific registers. For instance, parameters could be placed, in order, into registers R_0, R_1, and so on. The return value could be placed into register R_0. Larger values such as higher-precision numbers or other structures might span multiple registers.

```
      mov     r0,5            ; mul(5,21)
      mov     r1,21
      call    _mul
      halt
_mul:                         ; Function mul(x,y)
      push    r4
      mov     r2,1            ;   1 → mask
      xor     r3,r3           ;   0 → out
loop: or      r1,r1           ;   While y ≠ 0
      br.z    end
      mov     r4,r2           ;       If y ∧ mask ≠ 0 Then
      and     r4,r1
      br.z    skip
      add     r3,r0           ;           out + x → out
      xor     r1,r2           ;           y ⊕ mask → y
skip:                         ;       EndIf
      asl     r2,1            ;       mask · 2 → mask
      asl     r0,1            ;       x · 2 → x
      br      loop            ;   EndWhile
end:  mov     r0,r3           ;   Return out
      pop     r4
      ret                     ; EndFunction
```

During subroutine execution, it is almost certainly the case that some registers will be modified. Part of the ABI is to specify which routine is responsible for preserving the values in which registers.

Callee-Saved Registers

A *callee-saved* register is one that is not allowed to appear as if it has changed between entering and exiting the subroutine. If such a register is used, its value must be stored and restored. The previous two examples were written under the convention that register R_4 is callee-saved, so the multiplication function needed to `PUSH` it before use and `POP` it before returning. Notice that this changes where the stack pointer is; forgetting to `POP` the value would not only violate the convention but also case the subsequent `RET` to transfer control to an unknown location.

Caller-Saved Registers

A *caller-saved* register is exactly the opposite of a callee-saved register. Its value can be freely changed by the called subroutine. So if the caller wishes to preserve the value, then it must store it somewhere before the `CALL` and restore it afterward. The preceding examples were written under the convention that registers R_0–R_3 are caller-saved.

If all temporary storage is via the stack, then caller-saved registers will be at higher memory locations than the return address of the callee, and callee-saved registers will be below. This situation is depicted in Figure 5.5.

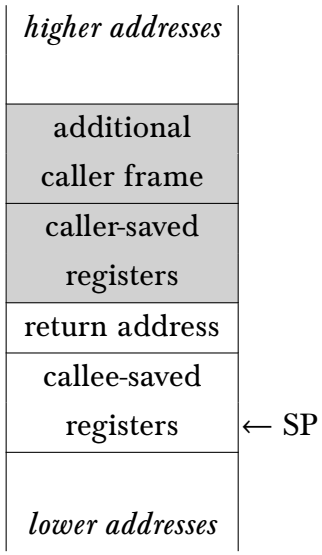


Figure 5.5: Caller- and Callee-Saved Registers

5.3.2 Passing Information via the Stack

Another option is to pass parameters and return values by placing them into the stack. This allows for passing more, larger values than can fit in the available registers. There is some freedom with respect to the stack layout, but the most important choice is whether the return

address precedes or follows the parameters and return value. One approach is to push the parameters in reverse order, then CALL the subroutine, and use one of the parameter slots to store the return value. The caller must push the values onto the stack *and* restore the stack upon return. The multiplication function can be rewritten in this form as follows. Inside the _mul function, the stack will resemble Figure 5.6.

```
        mov     r0,21           ; mul(5,21)
        push    r0
        mov     r0,5
        push    r0
        call    _mul
        add     sp,1
        pop     r0
        halt
_mul:                           ; Function mul(x,y)
        mov     r0,1            ;    1 → mask
        xor     r1,r1           ;    0 → out
loop:   or      (sp+2),(sp+2)   ;    While y ≠ 0
        br.z    end
        mov     r2,r0           ;        If y ∧ mask ≠ 0 Then
        and     r2,(sp+2)
        br.z    skip
        add     r1,(sp+1)       ;            out + x → out
        xor     (sp+2),r0       ;            y ⊕ mask → y
skip:                           ;        EndIf
        asl     r0,1            ;        mask · 2 → mask
        add     (sp+1),(sp+1)   ;        x · 2 → x
        br      loop            ;    EndWhile
end:    mov     (sp+2),r1       ;    Return out
        ret                     ; EndFunction
```

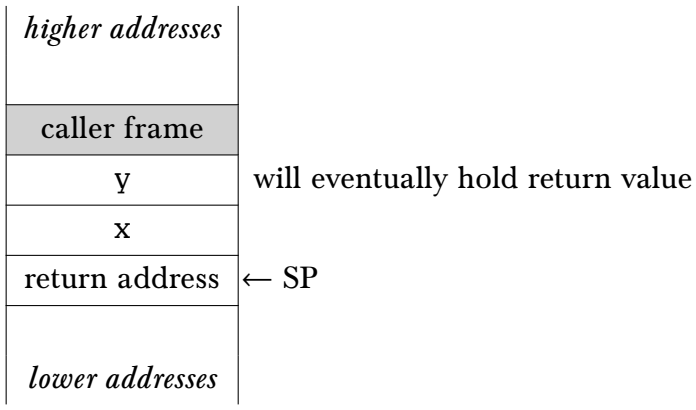


Figure 5.6: Stack-Based Parameter Passing

5.3.3 Nested Subroutines with Lexical Scope

Some additional information is needed for nested functions and procedures to be able to refer to variables in their containing scope. In order to facilitate this, a subroutine maintains a *static link* referring to the stack frame of its containing scope. Additionally, for convenience, a subroutine may have a *frame pointer* which points to the base of the stack where local variables and parameters are stored. The assembler recognizes FP as an alias for register R_{14}, the suggested register for this use. Then each subroutine should maintain a *dynamic link*, referring to the stack frame of the calling routine, to be restored upon exit. Consider the following complicated set of nested procedures.

```
Procedure a(n)
    Procedure b(m)
        Local i
        0 → i
        While i < n − m
            "␣␣" → (Terminal)
            i + 1 → i
        EndWhile
        0 → i
        While i < m
            "[]" → (Terminal)
            i + 1 → i
        EndWhile
        "\n" → (Terminal)
        Return m
    EndProcedure
    Procedure c(n)
        Local s
        0 → s
        If n > 0 Then
            s + b(n) → s
            s + c(n − 2) → s
            s + b(n) → s
        EndIf
        Return s
    EndProcedure
    Return c(n)
EndProcedure
a(7)
```

Assuming lexical scope, the *n* referenced by procedure *b* is not the *n* of procedure *c* but that of *a*. This means that *b* needs to know where the variables that belong to *a* are. To call a subroutine with a convention that links and parameters are accessed with positive offsets from the frame pointer and local variables with negative offsets:

- Push the parameters in reverse order,
- Decrement the stack pointer by two to make room for the static and dynamic links,

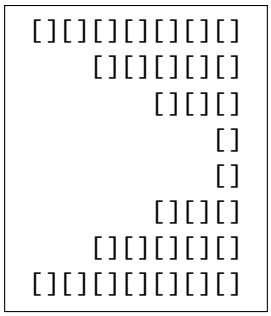


Figure 5.7: Nested Subroutine Output

- Set up the static link if necessary,
- CALL to push the return address and execute the subroutine, and
- Adjust the stack and pull out the return value, if any.

The called subroutine itself should do the following.

- Store the incoming dynamic link,
- Point the frame pointer at the top of the stack,
- Decrement the stack pointer to make room for local variables,
- Save any callee-save registers in use,
- Execute its operation,
- Fill its return value slot,
- Restore any callee-save registers in use,
- Restore the stack pointer,
- Restore the incoming dynamic link, and
- RET to pop the return address and proceed.

A direct assembly-language translation of the high-level source is provided below. Size and speed are both sacrificed for readability. After successful execution, the program should leave 32 = 0x20 in register R_0 and print the text shown in Figure 5.7 to the screen.

```
mov     r0,7            ; a(7)
push    r0              ; // provide parameter
sub     sp,2            ; // open room for links
call    _a              ; // call
add     sp,2            ; // skip links
pop     r0              ; // fetch return value
halt                    ; // end program
```

```
_a:                         ; Procedure a(n)
      mov     (sp+1),fp     ;   // set up dynamic link
      mov     fp,sp
      mov     r0,(fp+3)     ;   Return c(n)
      push    r0            ;   // provide parameter
      sub     sp,2          ;   // open room for links
      mov     (sp+1),fp     ;   // static link [c is inside a]
      call    c
      add     sp,2          ;   // skip links
      pop     r0            ;   // fetch return value
      mov     (fp+3),r0     ;   // copy it into return slot
                            ; EndProcedure [inner definitions follow]
      mov     fp,(fp+1)     ; // restore dynamic link
      ret
```

```
b:                          ; Procedure b(m) // inside a(n)
      mov     (sp+1),fp     ;   // set up dynamic link
      mov     fp,sp
      mov     r0,0x100
      xor     r1,r1         ;   Local i
                            ;   0 → i
                            ;   // reading n from a into R2
      mov     r2,(fp+2)     ;   // first get a's FP
      mov     r2,(r2+3)     ;   // and fetch the first parameter, n
      sub     r2,(fp+3)     ;   // register R2 holds n − m
top:  cmp     r1,r2         ;   While i < n − m
      br.s>=  Xtop
      mov     (r0),' '      ;       "␣␣" → (Terminal)
      mov     (r0),' '
      add     r1,1          ;       i + 1 → i
      br      top           ;   EndWhile
Xtop: xor     r1,r1         ;   0 → i
      mov     r2,(fp+3)     ;   // register R2 holds m
bot:  cmp     r1,r2         ;   While i < m
      br.s>=  Xbot
      mov     (r0),'['      ;       "[]" → (Terminal)
      mov     (r0),']'
      add     r1,1          ;       i + 1 → i
      br      bot           ;   EndWhile
Xbot: mov     (r0),'\n'     ;   '\n' → (Terminal)
                            ;   Return m // already in place
                            ; EndProcedure
      mov     fp,(fp+1)     ; // restore dynamic link
      ret
```

```
c:                          ; Procedure c(n) // inside a(n)
      mov     (sp+1),fp     ;   // set up dynamic link
      mov     fp,sp
      xor     r0,r0         ;   Local s
      push    r0            ;   0 → s
      cmp     (fp+3),0      ;   If n > 0 Then
      br.s<=  exit

      mov     r0,(fp+3)     ;       s + b(n) → s
      push    r0            ;       // provide parameter
      sub     sp,2          ;       // open room for links
      mov     (sp+1),(fp+2) ;       // static link [b is inside a]
      call    b
      add     sp,2          ;       // skip links
      pop     r1            ;       // fetch return value
      add     (fp-1),r1

      mov     r0,(fp+3)     ;       s + c(n - 2) → s
      sub     r0,2
      push    r0            ;       // provide parameter
      sub     sp,2          ;       // open room for links
      mov     (sp+1),(fp+2) ;       // static link [c is inside a]
      call    c
      add     sp,2          ;       // skip links
      pop     r1            ;       // fetch return value
      add     (fp-1),r1

      mov     r0,(fp+3)     ;       s + b(n) → s
      push    r0            ;       // provide parameter
      sub     sp,2          ;       // open room for links
      mov     (sp+1),(fp+2) ;       // static link [b is inside a]
      call    b
      add     sp,2          ;       // skip links
      pop     r1            ;       // fetch return value
      add     (fp-1),r1
exit:                       ;   EndIf
      pop     r0            ;   Return s
      mov     (fp+3),r0
                            ; EndProcedure
      mov     fp,(fp+1)     ; // restore dynamic link
      ret
```

The c procedure is recursive. During the first call to b inside of the first recursive call to c (the second call to c overall), the stack contains an a frame, two c frames, and a b frame, in that order, and appears as in Figure 5.8.

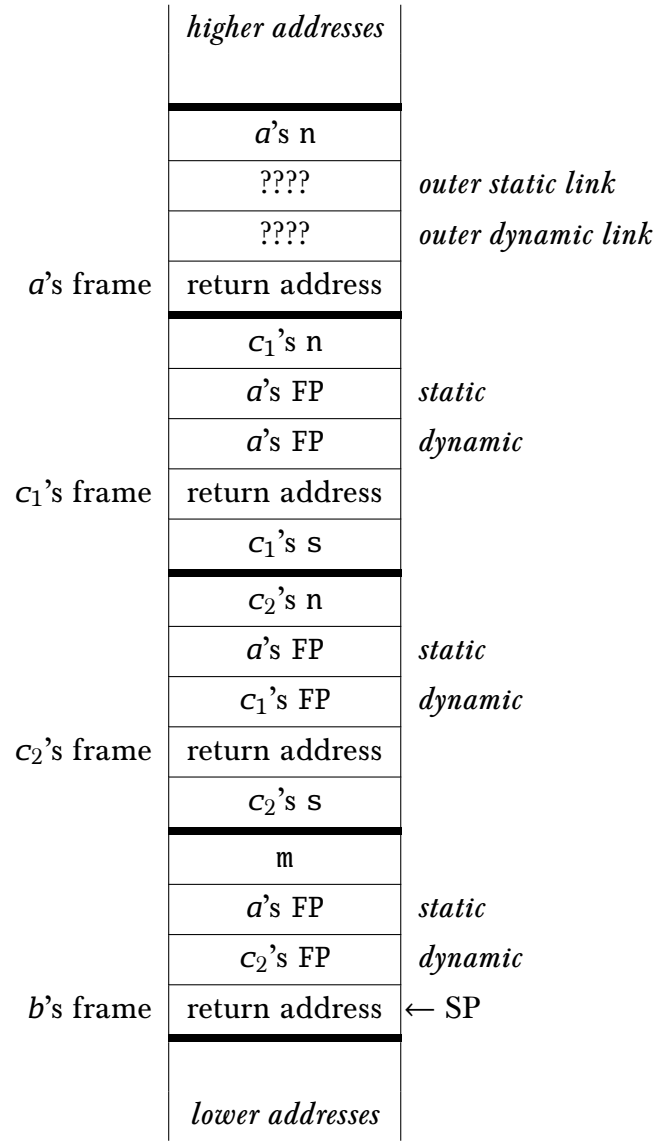


Figure 5.8: Stack Frames for Nested Procedures

5.4 Transfer of Control Between System & Programs

A user program can raise a signal to an operating system by means of the TRAP instruction. This is similar to a CALL in that the return address is pushed to the stack and control is transferred to a new location, except the stack used is the *system* stack, and the location is limited to one of eight available system entry points. The system routine entered by this mechanism should end with RTP to return to the user program.

Interrupts are handled by the same mechanism. A return address is pushed to the system stack, and program execution proceeds in system mode at a location specified by the interrupt priority level. As an interrupt may occur at any time, system software has special responsibilities: all registers that may be used by a user program, including the condition codes, must be in the same state at exit as they were in at entry. Typically, this is accomplished by wrapping the system routine as follows.

```
        push    r0
        orsr    r0,0
        push    r0
...
        pop     r0
        movsr   r0,r0
        pop     r0
        rtp
```

These responsibilities are a large part of the cost of switching between a user program and the system context.

Table 5.1: Instructions for Control Flow

Encoding		Instruction
0*mcs*, $0 \leqslant m \leqslant 3$	BR	Branch Relative
0*mcs*, $4 \leqslant m \leqslant 7$	BA	Branch Absolute
0*mcs*, $8 \leqslant m \leqslant$ B	CALL	Branch to Subroutine
0D*c*0	RET	Return from Subroutine
D5*cv*	TRAP	Software Interrupt
0D*c*F	RTP	Return to Program

Appendix A

Suggested Implementation Strategy

The STOL architecture is intended to be large enough to be useful and a viable target for a simple compiler, yet small enough to be implemented in an educational setting using a circuit simulator such as Logisim Evolution. To facilitate this, the instruction encodings are designed to require minimal decode logic, and the architecture is stratified into layers that build upon one another. Figure A.1 depicts a model of a complete computer system. The CPU itself can be built up in stages before connecting to a bus or external devices. The sequence presented in this appendix is far from the only possible approach.

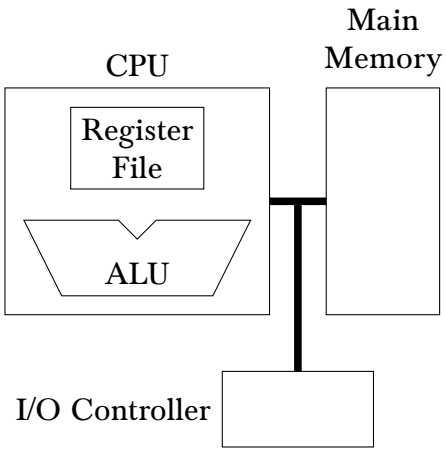


Figure A.1: Computer System Overview

A.1 Combinational Components

The simplest circuits are combinational circuits, in which output is dependent only on input, not time. With that in mind, construction should begin with the Arithmetic and Logic Unit (ALU), as that portion is purely combinational.

Specifically, components should be created to perform the 16-bit arithmetic, logic, and shift operations provided by the STOL architecture, those whose opcode field (the four most-significant bits) is between 2 and C, inclusive. Each should have inputs for two operands (A

and B) and the status, and should have outputs for the result and flags. Test cases should be created and evaluated at all points, by adjusting the input pins and ensuring that the associated outputs are correct.

Once the operations have been constructed, the ALU can be assembled by computing all operations in parallel and using multiplexers to select only the relevant outputs.

A.2 Registers

After the ALU is in place, the sixteen general-purpose registers should be added into a *register file*. It should have at least two outputs, to supply the A and B inputs of the ALU. Its inputs should include two four-bit selections, a data input, a write-enable for the destination register, and the clock.

The A and B outputs of the register bank should be connected to the associated inputs of the ALU, but not directly. By running each through a multiplexer, immediate data can be used as an input without first loading it into a register. The output of the ALU can, for now, be directly connected to the input of the register bank. Once this configuration is in working order, an input to the system can be added from which to read instructions from external memory. By detecting whether the source is a register or a short-format immediate value, this input can be configured to represent some sample instructions to verify that the results are as intended.

A.3 First External Memory

A reasonable next step would be to add a program counter to the system alongside a reset pin. The reset pin can clear all of the registers, or just the program counter. The program counter can initially be wired in such a way that it increases at every clock cycle and is connected to an output "address" pin. Then even in this degraded state, the system can be connected to a ROM device in order to execute purely sequential programs built of only single-cycle instructions. This is sufficient to compute, say, a specific number in the Fibonacci sequence. Testing is easier when the test program can be loaded into ROM rather than entered by hand.

A.4 Multicycle Instructions & Other Addressing Modes

Multicycle operations are coordinated by an internal state machine. This state machine should begin in a predictable state upon reset. After each instruction is fetched from main memory, it can be decoded to determine whether all necessary information is present, or if more memory accesses must occur to retrieve the desired operands.

A reasonable next step is to implement this state machine and allow purely sequential programs to use all four available addressing modes. In order to write to memory, some RAM needs to be added into the memory map, and an address decoder should be constructed that determines which memory module is being read from or written to at any given time.

A.5 Branching Control Flow

At this point, the relative and absolute branching instructions can be implemented with little additional hardware. The subroutine call and return instructions will likely require additional states in the state machine and additional hardware to control the stack pointer, R_{15}. After implementing these instructions alongside `PUSH` and `POP`, most of the sample code in this manual can be run.

A.6 Distinguishing System Mode & Program Mode

To this point, nothing has required privilege separation except perhaps ensuring correct functionality of the instructions that act upon the status register. The separation is achieved by implementing a system to select between two stack pointers based on privilege mode, and ensuring that `PSPR`, `PSPRW`, and `PSPW` only ever act on the program stack pointer. The `RTP` instruction could have been implemented as an alias for `RET` in the previous phase, but now must properly act upon the nesting-level field of the status register. The `TRAP` instruction is similar to a special-case of `CALL`, but must also set up the status register. After this phase is complete, the CPU is essentially finished. A working system can be built around this core.

A.7 Interrupt Handling

The final step is to account for interrupts by adding the request and priority inputs and the acknowledge output. The state machine is involved, as interrupts should only be handled *between* instructions, not *within* them.

A.8 Advanced Stages

At this point, a fully working CPU has been built and installed in a system with external memory. Those seeking to go further could choose among several additional features to add.

- **Memory protection**: A simple address decoder only determines where in memory an instruction is to be read or written. The CPU can be augmented with output pins indicating whether it is reading, writing, or executing the memory at the specified address, and the address decoder can be augmented to deny certain combinations at specific locations. This transforms the address decoder into a *memory protection unit*. A process running in system mode should be able to change the protections but a process in program mode attempting to do the same should cause a signal to be raised.
- **Architecture extensions**: Modern systems have a variety of instructions that perform different kinds of operations in hardware that would previously have been implemented in software. The hardware implementation is generally faster and more energy-efficient than the equivalent software. Using instruction words of the form `E`*xyz*, one could add a few additional instructions. Using such instructions as *instruction prefixes* allows for a large number of extensions.

- **Coprocessors**: In the same vein, early systems used separate computing hardware for floating-point operations and other advanced features. One could add a mechanism for communicating with a coprocessor. The coprocessor might be attached as yet another memory-mapped I/O device. Alternatively, one could address a coprocessor using port I/O by creating additional instructions of the form F*xyz*.
- **Cache memory**: Large main memory systems tend to be slower than the processors to which they are attached. To simulate this, one could place the main memory behind a clock divider. When data is not available, the processor must wait for it to arrive. Placing a cache between the processor and main memory can alleviate this concern when accesses are local.
- **Pipelining**: In the reference implementation of the STOL architecture, the shortest instructions retire in a single clock cycle but the longest require up to six cycles to execute. By modifying the state machine and adding additional components, the average time required per instruction could be brought down by overlapping portions of their execution. What happens when a control-flow instruction is executed?

Appendix B

Assembly Language Syntax

This appendix describes the syntax expected by the `stolas` assembler and used in examples throughout this manual.

Assembly source listings are provided in `typewriter font`. Text in *italics* represents a metavariable to be replaced by a suitable object, be that an integer constant, a label, a register name, or otherwise. Other text is to be written exactly as it appears. Within code listings, instruction mnemonics are written in **boldface** for clarity, as a rudimentary form of syntax highlighting.

B.1 Statements

Each line of an assembly-language program is a statement, composed of four fields, each of which optional. These fields are a label, an instruction mnemonic, an operand list, and a comment. An operand list cannot be provided if an instruction mnemonic is omitted. The label, if present, is separated from the remainder of the line by a colon (`:`), and the comment, if present, is separated by a semicolon (`;`).

```
label: mnemonic operands; comment
```

Instruction mnemonics are case-insensitive. An operand list is a sequence of one or more operands, with each but the first separated from the previous by a comma (`,`). Each instruction has a particular type and number of operands expected. The assembly language consistently places the destination location, if any, first in the operand list, and the instruction encoding is designed to match this.

B.2 Operands

There are four addressing modes available. The immediate addressing mode is indicated by a numeric constant, "*imm*", or the name of a label "*label*".

The register direct addressing mode, which uses the contents of a register directly, is indicated by a register name alone, "*reg*".

The register-indirect mode, which uses the contents of memory at the location specified by the value in a register, is indicated by a register name in parentheses, "(*reg*)".

Finally, the register-indirect with offset mode, which uses the contents of memory at the location specified by the sum of the value in a register and a numeric constant, is indicated by a register name in parentheses, a plus sign (+) or minus sign (−), and then the numeric value. "(*reg* + *imm*)" or "(*reg* − *imm*)".

B.3 Register Names

The case-insensitive strings `r0`, `r1`, `r2`, `r3`, `r4`, `r5`, `r6`, `r7`, `r8`, `r9`, `r10`, `r11`, `r12`, `r13`, `r14`, `r15`, `fp`, and `sp` refer to the named registers, and are reserved words. They cannot be used as label names.

B.4 Labels

A label is a case-sensitive sequence of characters containing Latin alphabetic letters (`a`–`z` and `A`–`Z`), underscores (`_`), and numeric digits (`0`–`9`). While a label may contain digits, it may not begin with one. As previously mentioned, the names of registers are reserved and cannot be used as label names.

A label followed by a colon (`:`) binds the current address to the label. It is an error to bind more than one address to the same label. When used as an operand, a label represents the address to which it is bound. The expressions "`@`", "`@` + *imm*", and "`@` − *imm*" refer to the current address, a positive offset from the current address, and a negative offset from the current address, respectively.

The assembler also provides special syntax to assign a specific value other than the current location to a label, which essentially defines an integer constant.

```
/define label imm
```

B.5 Numeric Values

Constant numeric values may be specified in one of three bases:

- Hexadecimal: constant begins with `0x` and contains the digits `0`–`9` or the letters `a`–`f`, which are treated as 10–15. Constants are case-insensitive.
- Octal: constant begins with `0` and contains the digits `0`–`7`.
- Decimal: constant begins with a digit `1`–`9` and contains the digits `0`–`9`.

Note that `09` is *not* a valid numeric constant. Constants can be negated by prefixing a minus sign (−). Additionally, an ASCII character surrounded by single quotes is a numeric constant whose value is its ASCII value. Some escape sequences that stand in for particular characters, summarized in Table B.1, are provided.

Table B.1: Escape Sequences

Escape	Value	Name
\a	7	Bell
\b	8	Backspace
\f	12	Form Feed
\n	10	New Line
\r	13	Carriage Return
\t	9	Horizontal Tab
\v	11	Vertical Tab
\"	34	Quotation Mark
\'	39	Apostrophe
\\	92	Backslash
nnn		*octal value*

The octal escape may have one to three digits in the range `0`–`7`.

B.6 Sections

The `stolas` assembler is aware of two segments: one segment combining code and read-only data ("the code segment") in which assembly begins, and the data segment, which contains uninitialized variables in main memory. The code segment includes assembled instructions, but can also contain read-only data such as constants and strings. The `DW` pseudo-operation requires a nonempty operand list and places each operand as a literal 16-bit word directly into the instruction stream.

```
dw imm[,imm...]
```

Any of the operands to `dw` may be a quoted string: a series of characters, including the escape sequences of Table B.1, surrounded by double-quotes (").

The data segment is entered with the following statement.

```
/bss
```

Variables can be declared by binding a label and reserving the requisite amount of space, specified as a number of 16-bit words.

```
variable:  res nwords
```

No other statements should appear in the data section.

Appendix C

Instruction Set Summary

This appendix provides a brief overview of the STOL instruction set. Each page is headed with the internal mnemonic and short description of an instruction. A mathematical or logical description of the operation is provided, as well as the standard assembler syntax.

Operands

$\mathtt{R}_n$	general-purpose register	`PSP`	program stack pointer
i	integer value	`SSP`	system stack pointer
`PC`	program counter	`SP`	stack pointer

Qualifiers

⟨register⟩:*n*[:*m*]	n^{th} [through m^{th}] bit of the register, 0 is least-significant,
(⟨register⟩)	value in memory addressed by the register
(−⟨register⟩)	value in memory addressed by the predecremented register
(⟨register⟩+)	value in memory addressed by the postincremented register
(⟨register⟩ + *i*)	value in memory addressed by the sum of the register and *i*

Operations

→	write the left operand into the location specified by the right operand(s)
+	add
−	subtract
~	bitwise logical NOT
∧	bitwise logical AND
∨	bitwise logical inclusive-OR
⊕	bitwise logical exclusive-OR
◃	shift the left operand leftward by the number of positions specified by the right
▹	shift the left operand rightward by the number of positions specified by the right
↺	rotate the left operand leftward by the number of positions specified by the right
↻	rotate the left operand rightward by the number of positions specified by the right

Operation

Destination + Source → Destination

Assembler Syntax

add *destination*,*source*

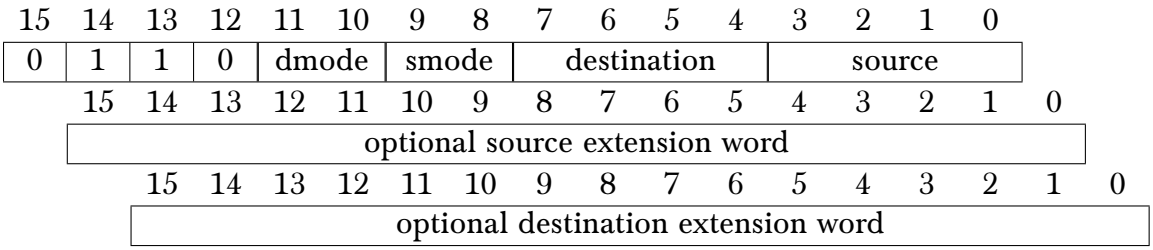
Description

Add the source operand to the destination operand and store the result in the destination location.

Addressing Modes

Mode		Syntax	Source	Destination
Immediate	00	*i*		×
Register Direct	01	R_n		
Register Indirect	10	(R_n)		
Register Indirect with Offset	11	(R_n+*i*)		

Instruction Encoding



Flags

C: set if carry occurred
N: set if result is negative
V: set if overflow occurred
Z: set if result is zero

Operation

Destination + Source + SR:3 → Destination

Assembler Syntax

addc *destination*,*source*

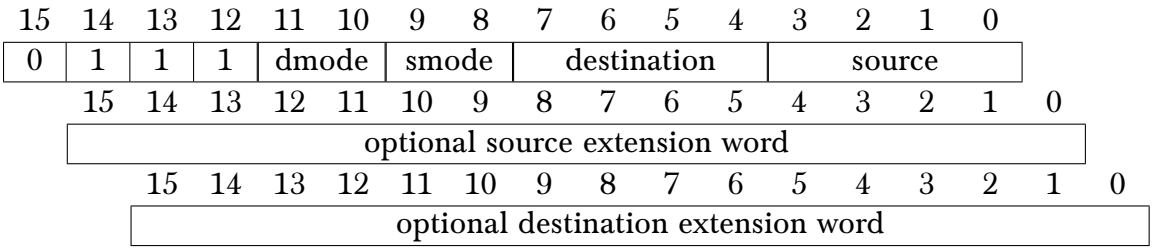
Description

Add the source operand to the destination operand along with the carry bit and store the result in the destination location.

Addressing Modes

Mode		Syntax	Source	Destination
Immediate	00	*i*		×
Register Direct	01	R_n		
Register Indirect	10	(R_n)		
Register Indirect with Offset	11	(R_n+*i*)		

Instruction Encoding



Flags

C: set if carry occurred
N: set if result is negative
V: set if overflow occurred
Z: set if result is zero

Operation

Destination ∧ Source ⟶ Destination

Assembler Syntax

and *destination*,*source*

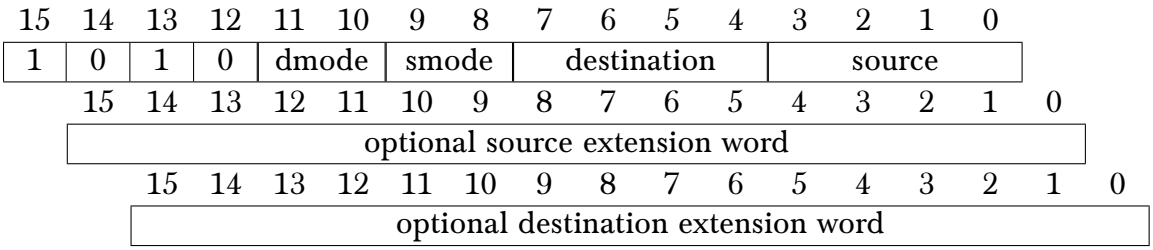
Description

Compute the bitwise logical AND of the source operand and destination operand and store the result in the destination location.

Addressing Modes

Mode		Syntax	Source	Destination
Immediate	00	*i*		×
Register Direct	01	R_n		
Register Indirect	10	(R_n)		
Register Indirect with Offset	11	(R_n+*i*)		

Instruction Encoding



Flags

C: unaffected
N: set if result is negative
V: unaffected
Z: set if result is zero

Operation

SR → Destination
if system mode:
    SR ∧ Source → SR
else:
    SR ∧ Source → SR:0:3

Assembler Syntax

`andsr` *`destination`*`,`*`source`*

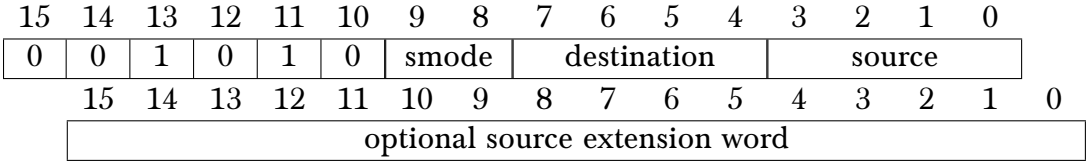
Description

Store the content of the status register into the destination location. Additionally, compute the bitwise logical AND of the source operand and status register and store the result back into the status register. In system mode, the entire status register is written. In program mode, only the condition flags are modified. Note that these operations are simultaneous; if the source and destination refer to the same location, the original source is used.

Addressing Modes

Mode		Syntax	Source	Destination
Immediate	00	*i*		×
Register Direct	01	R_n		
Register Indirect	10	(R_n)		×
Register Indirect with Offset	11	(R_n+*i*)		×

Instruction Encoding



Flags

C: set by operation
N: set by operation
V: set by operation
Z: set by operation

Operation

Destination ◃ Source ⟶ Destination

Assembler Syntax

`asl` *`destination`*`,`*`source`*

Description

Shift the destination register leftward the number of times indicated by the unsigned source operand and store the result in the destination register. The carry flag is set if any set bit is shifted out at any point. The overflow flag is set if the sign changes at any point. Zeros are shifted into newly open positions.

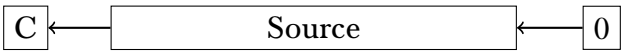


Figure C.1: Arithmetic Shift Left Operation

Addressing Modes

Mode		Syntax	Source	Destination
Immediate	00	*i*	†	×
Register Direct	01	R_n		
Register Indirect	10	(R_n)	×	×
Register Indirect with Offset	11	(R_n+i)	×	×

† only short immediate data (1–15) is permitted.

Instruction Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	i/r	destination				source			

Flags

C: set if any set bit is shifted out
N: set if result is negative
V: set if sign changes during operation
Z: set if result is zero

Operation

Destination ▹ Source ⟶ Destination

Assembler Syntax

`asr` *`destination`*`,`*`source`*

Description

Shift the destination register rightward the number of times indicated by the unsigned source operand and store the result in the destination register. The carry flag is set if any set bit is shifted out at any point. The sign bit is shifted into newly open positions.

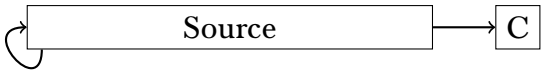


Figure C.2: Arithmetic Shift Right Operation

Addressing Modes

Mode		Syntax	Source	Destination
Immediate	00	*i*	†	×
Register Direct	01	R_n		
Register Indirect	10	(R_n)	×	×
Register Indirect with Offset	11	(R_n+*i*)	×	×

† only short immediate data (1–15) is permitted.

Instruction Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	1	i/r	destination				source			

Flags

C: set if any set bit is shifted out
N: set if result is negative
V: always cleared
Z: set if result is zero

Operation

if Condition:
    Source → PC

Assembler Syntax

`ba[.`*`condition`*`] `*`source`*

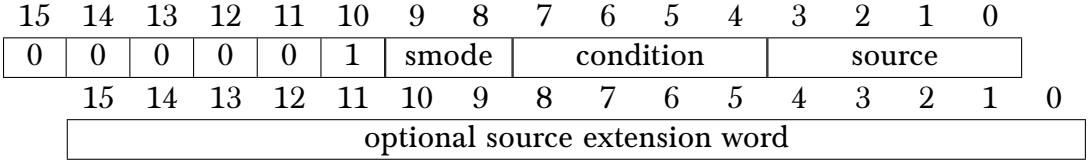
Description

If the specified condition is met, program execution continues from the address specified by the source operand. Otherwise, execution proceeds to the next instruction.

Addressing Modes

Mode		Syntax	Source
Immediate	00	*i*	
Register Direct	01	R_n	
Register Indirect	10	(R_n)	
Register Indirect with Offset	11	(R_n+*i*)	

Instruction Encoding



Flags

C: unaffected
N: unaffected
V: unaffected
Z: unaffected

Operation

if Condition:
    PC + Source → PC

Assembler Syntax

br[.*condition*] *source*

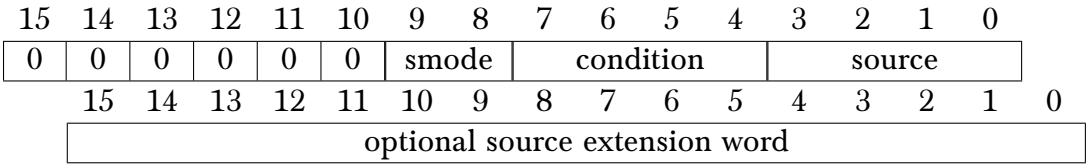
Description

If the specified condition is met, the program counter is displaced by the source operand and program execution continues from the resulting location. Otherwise, execution proceeds to the next instruction.

Addressing Modes

Mode		Syntax	Source
Immediate	00	*i*	
Register Direct	01	R_n	
Register Indirect	10	(R_n)	
Register Indirect with Offset	11	(R_n+*i*)	

Instruction Encoding



Flags

C: unaffected
N: unaffected
V: unaffected
Z: unaffected

Operation

if Condition:
    PC + E + 1 → (−SP)
    Source → PC
where E is the number of extension words present (0 or 1)

Assembler Syntax

`call[.`*`condition`*`] `*`source`*

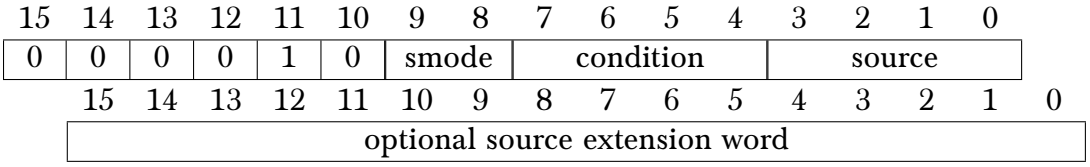
Description

If the specified condition is met, the location of the next instruction is pushed to the stack and then program execution continues from the location specified by the source operand. Otherwise, execution proceeds to the next instruction. Note that the final execution address is computed before the location is pushed to the stack.

Addressing Modes

Mode		Syntax	Source
Immediate	00	*i*	
Register Direct	01	R$_n$	
Register Indirect	10	(R$_n$)	
Register Indirect with Offset	11	(R$_n$+*i*)	

Instruction Encoding



Flags

C: unaffected
N: unaffected
V: unaffected
Z: unaffected

Operation

Destination – Source

Assembler Syntax

`cmp` *`destination`*`,`*`source`*

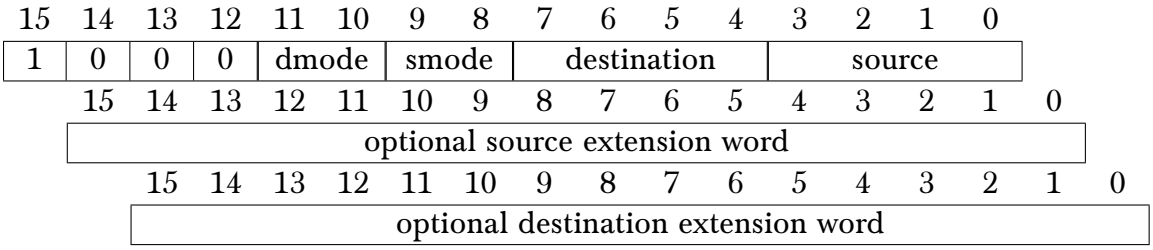
Description

Subtract the source operand from the destination operand and set the condition codes accordingly, but do not store the result anywhere. Note that in the reference implementation, a write is still performed, but the value is the original contents of the destination location. This can be an issue if the location is memory-mapped I/O.

Addressing Modes

Mode		Syntax	Source	Destination
Immediate	00	*i*		×
Register Direct	01	R_n		
Register Indirect	10	(R_n)		
Register Indirect with Offset	11	(R_n+*i*)		

Instruction Encoding



Flags

C: set if carry occurred
N: set if result is negative
V: set if overflow occurred
Z: set if result is zero

Operation

if Condition:
    PC → PC

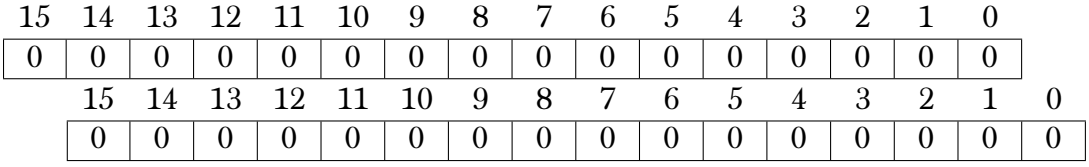
Assembler Syntax

`halt[.`*`condition`*`]`

Description

If the specified condition is met, the program counter is unchanged and program execution cannot continue until an interrupt is received. Otherwise, execution proceeds to the next instruction.

Instruction Encoding



Flags

C: unaffected
N: unaffected
V: unaffected
Z: unaffected

Synthesis

```
        br      @
```

Operation

Destination ▹ Source ⟶ Destination

Assembler Syntax

`lsr` *`destination`*`,`*`source`*

Description

Shift the destination register rightward the number of times indicated by the unsigned source operand and store the result in the destination register. The carry flag is set if any set bit is shifted out at any point. Zeros are shifted into newly open positions.

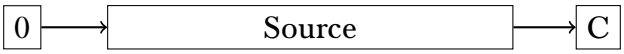


Figure C.3: Logical Shift Right Operation

Addressing Modes

Mode		Syntax	Source	Destination
Immediate	00	*i*	†	×
Register Direct	01	R_n		
Register Indirect	10	(R_n)	×	×
Register Indirect with Offset	11	(R_n+*i*)	×	×

† only short immediate data (1–15) is permitted.

Instruction Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	1	i/r	destination				source			

Flags

C: set if any set bit is shifted out
N: set if result is negative
V: always cleared
Z: set if result is zero

Operation

Source → Destination

Assembler Syntax

`mov` *`destination`*`,`*`source`*

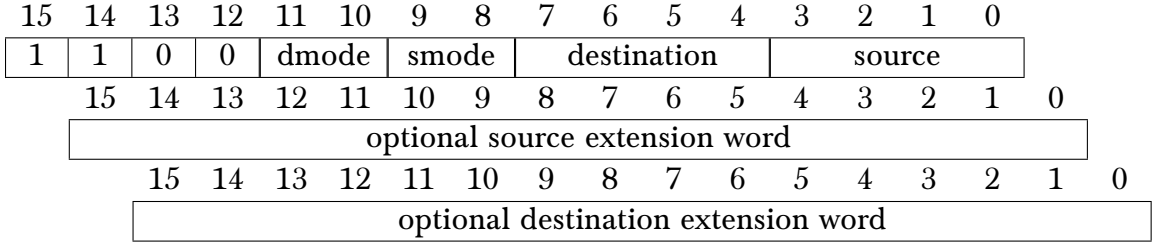
Description

Copy the source operand into the destination location. Note that in the reference implementation, this instruction performs a memory read on indirectly addressed destinations even though the result is ignored. This can be an issue if the location is memory-mapped I/O.

Addressing Modes

Mode		Syntax	Source	Destination
Immediate	00	*i*		×
Register Direct	01	R_n		
Register Indirect	10	(R_n)		
Register Indirect with Offset	11	(R_n+*i*)		

Instruction Encoding



Flags

C: unaffected
N: unaffected
V: unaffected
Z: unaffected

Operation

SR → Destination
if system mode:
    Source → SR
else:
    Source → SR:0:3

Assembler Syntax

`movsr` *`destination`*`,`*`source`*

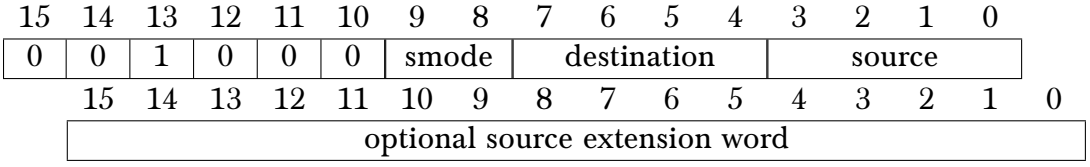
Description

Store the content of the status register into the destination location. Additionally, copy the source operand into the status register. In system mode, the entire status register is written. In program mode, only the condition flags are modified. Note that these operations are simultaneous; if the source and destination refer to the same location, the original source is used.

Addressing Modes

Mode		Syntax	Source	Destination
Immediate	00	*i*		×
Register Direct	01	R_n		
Register Indirect	10	(R_n)		×
Register Indirect with Offset	11	(R_n+*i*)		×

Instruction Encoding



Flags

C: set by operation
N: set by operation
V: set by operation
Z: set by operation

Operation

0 − Source → Destination

Assembler Syntax

neg *destination*[,*source*]

Description

Place the negation of the source register into the destination register. If the source is unspecified, the destination register is used instead.

Addressing Modes

Mode		Syntax	Source	Destination
Immediate	00	*i*	×	×
Register Direct	01	R_n		
Register Indirect	10	(R_n)	×	×
Register Indirect with Offset	11	(R_n+*i*)	×	×

Instruction Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	0	1	destination				source			

Flags

C: set if result is nonzero
N: set if result is negative
V: set if result is 0x8000 = −32768
Z: set if result is zero

Operation

None

Assembler Syntax

nop

Description

Execution proceeds to the next instruction.

Instruction Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Flags

C: unaffected
N: unaffected
V: unaffected
Z: unaffected

Synthesis

```
br      @+1
```

Operation

~Destination → Destination

Assembler Syntax

`not` *`destination`*

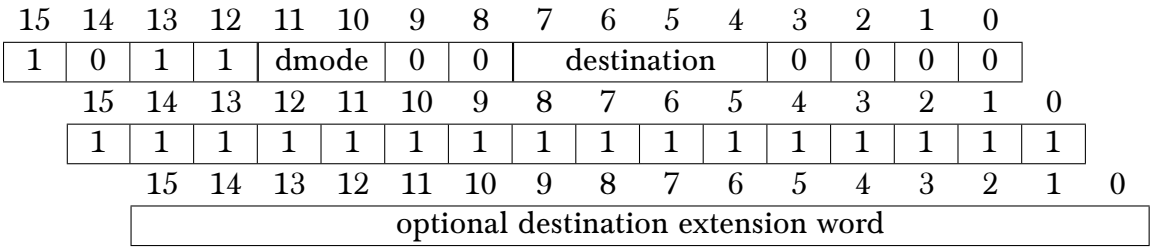
Description

Compute the bitwise logical NOT of the destination operand and store the result in the destination location.

Addressing Modes

Mode		Syntax	Destination
Immediate	00	*i*	×
Register Direct	01	R_n	
Register Indirect	10	(R_n)	
Register Indirect with Offset	11	(R_n+*i*)	

Instruction Encoding



Flags

C: unaffected
N: set if result is negative
V: unaffected
Z: set if result is zero

Synthesis

```
xor     destination,0xFFFF
```

Operation

Destination ∨ Source ⟶ Destination

Assembler Syntax

or *destination*,*source*

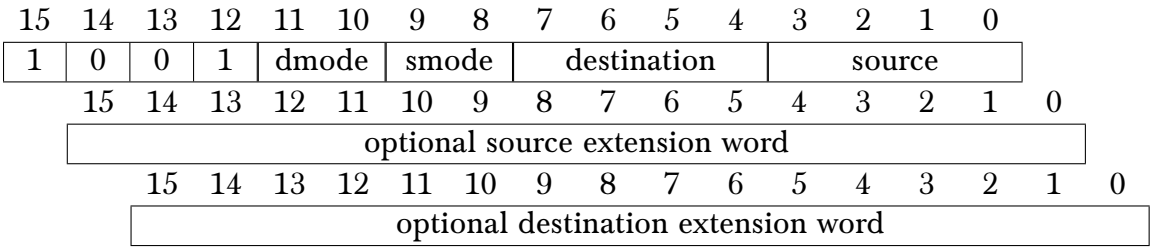
Description

Compute the bitwise logical inclusive-OR of the source operand and destination operand and store the result in the destination location.

Addressing Modes

Mode		Syntax	Source	Destination
Immediate	00	*i*		×
Register Direct	01	R_n		
Register Indirect	10	(R_n)		
Register Indirect with Offset	11	(R_n+*i*)		

Instruction Encoding



Flags

C: unaffected
N: set if result is negative
V: unaffected
Z: set if result is zero

Operation

SR → Destination
if system mode:
    SR ∨ Source → SR
else:
    SR ∨ Source → SR:0:3

Assembler Syntax

`orsr` *`destination`*`,`*`source`*

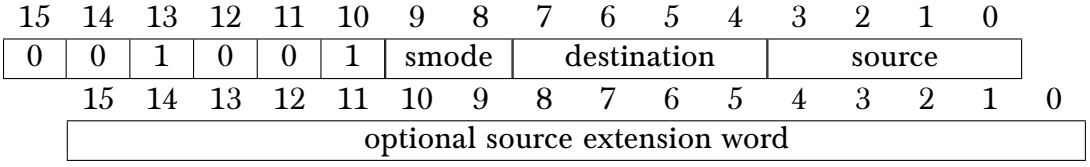
Description

Store the content of the status register into the destination location. Additionally, compute the bitwise logical inclusive-OR of the source operand and status register and store the result back into the status register. In system mode, the entire status register is written. In program mode, only the condition flags are modified. Note that these operations are simultaneous; if the source and destination refer to the same location, the original source is used.

Addressing Modes

Mode		Syntax	Source	Destination
Immediate	00	*i*		×
Register Direct	01	R_n		
Register Indirect	10	(R_n)		×
Register Indirect with Offset	11	(R_n+*i*)		×

Instruction Encoding



Flags

C: set by operation
N: set by operation
V: set by operation
Z: set by operation

Operation

(SP+) → Destination

Assembler Syntax

pop *destination*

Description

Copy the contents of the memory location addressed by the postincremented stack pointer into the destination register.

Addressing Modes

Mode		Syntax	Destination
Immediate	00	*i*	×
Register Direct	01	R_n	
Register Indirect	10	(R_n)	×
Register Indirect with Offset	11	(R_n+*i*)	×

Instruction Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	1	destination				0	0	0	0

Flags

C: unaffected
N: unaffected
V: unaffected
Z: unaffected

Operation

PSP → Destination

Assembler Syntax

`pspr` *`destination`*

Description

Copy the contents of program stack pointer register into the destination register.

Addressing Modes

Mode		Syntax	Destination
Immediate	00	*i*	×
Register Direct	01	R_n	
Register Indirect	10	(R_n)	×
Register Indirect with Offset	11	(R_n+*i*)	×

Instruction Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	1	0	destination				0	0	0	0

Flags

C: unaffected
N: unaffected
V: unaffected
Z: unaffected

Operation

PSP → Destination
Source → PSP

Assembler Syntax

`psprw` *`destination`*`,`*`source`*

Description

Copy the contents of the source register and the current program stack pointer register into the program stack pointer register and destination register, respectively. Note that the copies are simultaneous; the destination register receives the original program stack pointer and the program stack pointer receives the original source register, even if the source and destination are the same.

Addressing Modes

Mode		Syntax	Source	Destination
Immediate	00	*i*	×	×
Register Direct	01	R_n		
Register Indirect	10	(R_n)	×	×
Register Indirect with Offset	11	(R_n+*i*)	×	×

Instruction Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	1	1	destination				source			

Flags

C: unaffected
N: unaffected
V: unaffected
Z: unaffected

Operation

Source → PSP

Assembler Syntax

`pspw` *source*

Description

Copy the contents of the source register into the program stack pointer register.

Addressing Modes

Mode		Syntax	Source
Immediate	00	*i*	×
Register Direct	01	R_n	
Register Indirect	10	(R_n)	×
Register Indirect with Offset	11	(R_n+*i*)	×

Instruction Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	1	0	0	0	0	source			

Flags

C: unaffected
N: unaffected
V: unaffected
Z: unaffected

Operation

Source → (−SP)

Assembler Syntax

push *source*

Description

Copy the contents of the source register into the memory location addressed by the predecremented stack pointer.

Addressing Modes

Mode		Syntax	Source
Immediate	00	*i*	×
Register Direct	01	R_n	
Register Indirect	10	(R_n)	×
Register Indirect with Offset	11	(R_n+*i*)	×

Instruction Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	1	0	0	0	0	source			

Flags

C: unaffected
N: unaffected
V: unaffected
Z: unaffected

Operation

if Condition:
    (SP+) → PC

Assembler Syntax

`ret[.`*`condition`*`]`

Description

If the specified condition is met, program execution continues from an address popped from the stack. Otherwise, execution proceeds to the next instruction.

Instruction Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	condition				0	0	0	0

Flags

C: unaffected
N: unaffected
V: unaffected
Z: unaffected

Operation

[SR:3 Destination] ↻ Source → Destination

Assembler Syntax

`rlc` *`destination`*`,`*`source`*

Description

Rotate the destination register leftward through the carry flag the number of times indicated by the unsigned source operand and store the result in the destination register.

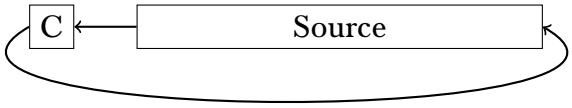


Figure C.4: Rotate Left Through Carry Operation

Addressing Modes

Mode		Syntax	Source	Destination
Immediate	00	*i*	†	×
Register Direct	01	R_n		
Register Indirect	10	(R_n)	×	×
Register Indirect with Offset	11	(R_n+*i*)	×	×

† only short immediate data (1–15) is permitted.

Instruction Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	0	i/r	destination				source			

Flags

C: set by operation
N: set if result is negative
V: always cleared
Z: set if result is zero

Operation

Destination ↻ Source → Destination

Assembler Syntax

`rol` *`destination`*`,`*`source`*

Description

Rotate the destination register leftward the number of times indicated by the unsigned source operand and store the result in the destination register.

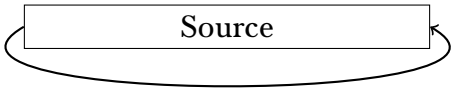


Figure C.5: Rotate Left Operation

Addressing Modes

Mode		Syntax	Source	Destination
Immediate	00	*i*	†	×
Register Direct	01	R_n		
Register Indirect	10	(R_n)	×	×
Register Indirect with Offset	11	(R_n+*i*)	×	×

† only short immediate data (1–15) is permitted.

Instruction Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	i/r	destination				source			

Flags

C: always cleared
N: set if result is negative
V: always cleared
Z: set if result is zero

Operation

Destination ↻ Source → Destination

Assembler Syntax

`ror` *`destination`*`,`*`source`*

Description

Rotate the destination register rightward the number of times indicated by the unsigned source operand and store the result in the destination register.

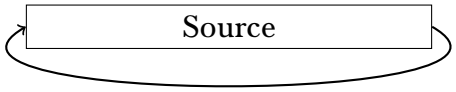


Figure C.6: Rotate Right Operation

Addressing Modes

Mode		Syntax	Source	Destination
Immediate	00	*i*	†	×
Register Direct	01	R_n		
Register Indirect	10	(R_n)	×	×
Register Indirect with Offset	11	(R_n+*i*)	×	×

† only short immediate data (1–15) is permitted.

Instruction Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	1	i/r	destination				source			

Flags

C: always cleared
N: set if result is negative
V: always cleared
Z: set if result is zero

Operation

[SR:3 Destination] ↻ Source → Destination

Assembler Syntax

`rrc` *`destination`*`,`*`source`*

Description

Rotate the destination register rightward through the carry flag the number of times indicated by the unsigned source operand and store the result in the destination register.

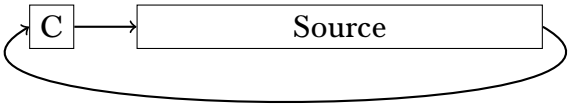


Figure C.7: Rotate Right Through Carry Operation

Addressing Modes

Mode		Syntax	Source	Destination
Immediate	00	*i*	†	×
Register Direct	01	R_n		
Register Indirect	10	(R_n)	×	×
Register Indirect with Offset	11	(R_n+*i*)	×	×

† only short immediate data (1–15) is permitted.

Instruction Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	1	i/r	destination				source			

Flags

C: set by operation
N: set if result is negative
V: always cleared
Z: set if result is zero

Operation

if system mode and Condition:
    if interrupt nesting level is zero:
        0 → SR:15
    SR:8:11 – 1 → SR:8:11
    (SSP+) → PC
else if program mode and Condition:
    (PSP+) → PC

Assembler Syntax

`rtp[.condition]`

Description

In program mode, this instruction behaves in a manner identical to RET. In system mode, it is more involved. If the specified condition is not met, execution proceeds to the next instruction. If, on the other hand, the condition is met, then the current interrupt nesting level is decreased by one. Program execution continues from an address popped from the system stack. If the nesting level was originally zero, then the processor will exit system mode and resume operation in program mode.

Instruction Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	condition				1	1	1	1

Flags

C: unaffected
N: unaffected
V: unaffected
Z: unaffected

Operation

Destination – Source → Destination

Assembler Syntax

sub *destination*,*source*

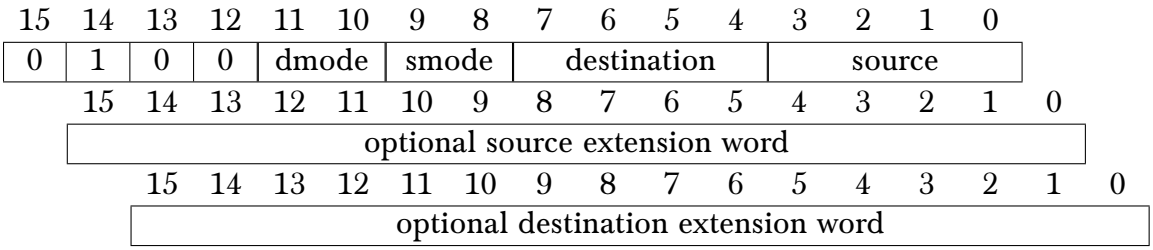
Description

Subtract the source operand from the destination operand and store the result in the destination location.

Addressing Modes

Mode		Syntax	Source	Destination
Immediate	00	*i*		×
Register Direct	01	R_n		
Register Indirect	10	(R_n)		
Register Indirect with Offset	11	(R_n+*i*)		

Instruction Encoding



Flags

C: set if borrow occurred
N: set if result is negative
V: set if overflow occurred
Z: set if result is zero

Operation

Destination – Source – SR:3 → Destination

Assembler Syntax

`subb` *`destination`*`,`*`source`*

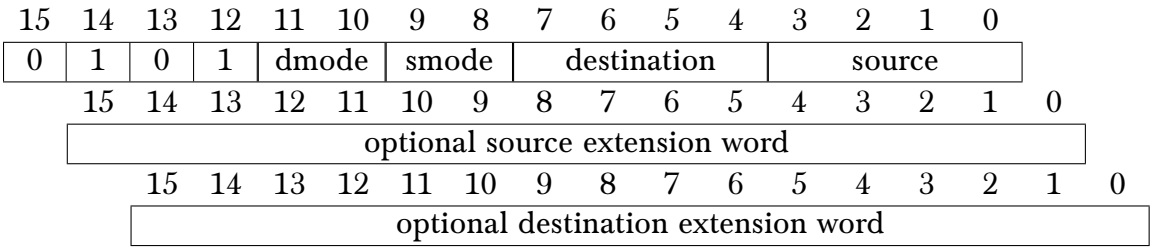
Description

Subtract the source operand and the carry bit (treated as a borrow bit) from the destination operand and store the result in the destination location.

Addressing Modes

Mode		Syntax	Source	Destination
Immediate	00	*i*		×
Register Direct	01	R_n		
Register Indirect	10	(R_n)		
Register Indirect with Offset	11	(R_n+*i*)		

Instruction Encoding



Flags

C: set if borrow occurred
N: set if result is negative
V: set if overflow occurred
Z: set if result is zero

Operation

if Condition:
    1 → SR:15
    SR:8:11 + 1 → SR:8:11
    PC + 1 → (−SSP+)
    240 + N + N → PC

Assembler Syntax

`trap[.condition] n`

Description

If the specified condition is not met, then execution proceeds to the next instruction. If, however, the condition is met, the processor enters system mode and the interrupt nesting level is increased by one. The location of the next instruction is pushed to the system stack, and program execution continues from the location associated with the requested interrupt, 240 + N + N.

Instruction Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	condition				n			0

Flags

C: unaffected
N: unaffected
V: unaffected
Z: unaffected

Operation

Destination ⊕ Source → Destination

Assembler Syntax

xor *destination*,*source*

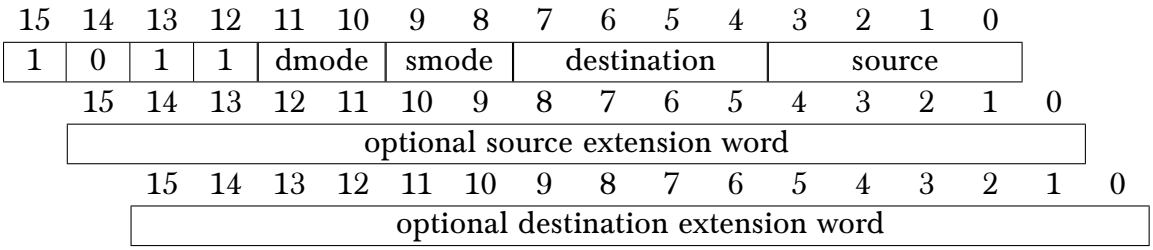
Description

Compute the bitwise logical exclusive-OR of the source operand and destination operand and store the result in the destination location.

Addressing Modes

Mode		Syntax	Source	Destination
Immediate	00	*i*		×
Register Direct	01	R_n		
Register Indirect	10	(R_n)		
Register Indirect with Offset	11	(R_n+*i*)		

Instruction Encoding



Flags

C: unaffected
N: set if result is negative
V: unaffected
Z: set if result is zero

Operation

SR → Destination
if system mode:
    SR ⊕ Source → SR
else:
    SR ⊕ Source → SR:0:3

Assembler Syntax

`xorsr` *`destination`*`,`*`source`*

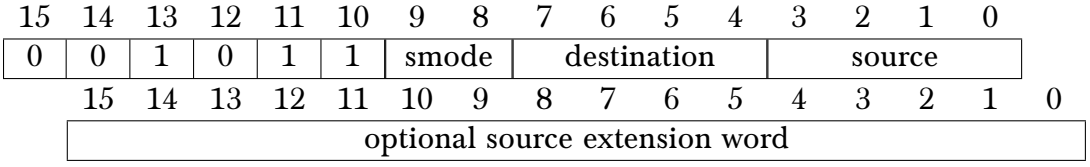
Description

Store the content of the status register into the destination location. Additionally, compute the bitwise logical exclusive-OR of the source operand and status register and store the result back into the status register. In system mode, the entire status register is written. In program mode, only the condition flags are modified. Note that these operations are simultaneous; if the source and destination refer to the same location, the original source is used.

Addressing Modes

Mode		Syntax	Source	Destination
Immediate	00	*i*		×
Register Direct	01	R_n		
Register Indirect	10	(R_n)		×
Register Indirect with Offset	11	(R_n+*i*)		×

Instruction Encoding



Flags

C: set by operation
N: set by operation
V: set by operation
Z: set by operation

Instruction Set in Numerical Order

Encoding		Instruction
0*mcs*, $0 \leqslant m \leqslant 3$	BR	Branch Relative
0*mcs*, $4 \leqslant m \leqslant 7$	BA	Branch Absolute
0*mcs*, $8 \leqslant m \leqslant C$	CALL	Branch to Subroutine
0D*c*0	RET	Return from Subroutine
0D*c*F	RTP	Return to Program
110*s*	PUSH	Push to Stack
15*d*0	POP	Pop from Stack
190*s*	PSPW	Set Program Stack Pointer
1A*d*0	PSPR	Read Program Stack Pointer
1B*ds*	PSPRW	Read and Set Program Stack Pointer
2*mds*, $0 \leqslant m \leqslant 3$	MOVSR	Replace Status
2*mds*, $4 \leqslant m \leqslant 7$	ORSR	Logical Inclusive-OR into Status
2*mds*, $8 \leqslant m \leqslant B$	ANDSR	Logical AND into Status
2*mds*, $C \leqslant m \leqslant F$	XORSR	Logical Exclusive-OR into Status
31*ds*	NEG	Two's-Complement Negate Register
3*mds*, $2 \leqslant m \leqslant 3$	LSR	Logical Shift Right
3*mds*, $4 \leqslant m \leqslant 5$	ASL	Arithmetic Shift Left
3*mds*, $6 \leqslant m \leqslant 7$	ASR	Arithmetic Shift Right
3*mds*, $8 \leqslant m \leqslant 9$	ROL	Rotate Left
3*mds*, $A \leqslant m \leqslant B$	ROR	Rotate Right
3*mds*, $C \leqslant m \leqslant D$	RLC	Rotate Left Through Carry
3*mds*, $E \leqslant m \leqslant F$	RRC	Rotate Right Through Carry
4*mds*	SUB	Subtract
5*mds*	SUBB	Subtract with Borrow
6*mds*	ADD	Add
7*mds*	ADDC	Add with Carry
8*mds*	CMP	Compare
9*mds*	OR	Bitwise Logical Inclusive-OR
A*mds*	AND	Bitwise Logical AND
B*mds*	XOR	Bitwise Logical Exclusive-OR
C*mds*	MOV	Move
D5*cv*	TRAP	Software Interrupt
E*xyz*		*Reserved for extensions*
F*xyz*		*Reserved for extensions*

- *c*: Condition
- *d*: Destination
- *m*: Mode
- *s*: Source
- *v*: Software interrupt level, times two