Ladybug Documentation

Release 1A

Håkan Thörngren

May 19, 2020

Contents

Contents:

CHAPTER 1

Introduction

Welcome to the Ladybug module for the HP-41 calculator! Ladybug is a powerful tool, intended to be useful when you are debugging or working with low level matters related to computers.

Ladybug provides a new mode for you HP-41 calculator which allows it to work as a customizable integer binary calculator. It makes it easy to work in different number bases, perform arithmetic, bitwise and logical operations in a given fixed word size. Operations that you typically will encounter when working with computers at its lowest level.

The main goal with this module is to blend the majority of the capabilities of an HP-16C into the HP-41 environment, while taking advantage of the extra facilities provided by the HP-41. In other words, to combine the best of both worlds.

1.1 Plug-in module

Ladybug is a module image that needs to be put in some programmable plug-in module hardware. This can be a Clonix module, an MLDL or some kind of ROM emulator. You need to consult the documentation of ROM emulation hardware for this.

It is also possible to use Ladybug on HP-41 emulators.

The Ladybug image is a 2x4K module. Two banks occupies a single 4K page in the normal memory expansion space (page 7–F).

You must also load the separate OS4 module in page 4 for Ladybug to work.

The Boost module is also needed to access the secondary functions in Ladybug.

1.2 This release

This version, 1A is the first that uses the OS4 module.

1.3 Resource requirements

Ladybug allocates two registers from the free memory pool. Apart from this, it does not impose any restrictions on the environment and will run comfortable on any HP-41C, HP-41CV, HP-41CX or HP-41CL.

Due to OS4 and Boost, some additional register space is needed to get it all properly running. You can expect that five registers are allocated by these three modules if you enable Ladybug integer mode.

The XROM number used by this module is 16 and the private storage area is I/O buffer number 0.

1.4 Using this guide

This guide assumes that you have a working knowledge about:

- The HP-41 calculator, especially its RPN system.
- Have a good understanding of different number bases and working with different word sizes. Basically bits as used in most computers at its lowest level.

1.5 Further reading

If you feel that you need to brush up your background knowledge, here are some suggested reading:

- The *Owners Manuals* supplied with the HP-41, Hewlett Packard Company.
- *HP-16C Computer Scientist Owners Handbook*, Hewlett Packard Company.
- *Hackers Delight*, Henry S. Warren, Addison-Wesley, 2003, 2012.
- *Extend your HP-41*, W Mier-Jedrzejowicz, 1985.

As always, learning by doing tends to work best. Insert fresh batteries into your HP-41 and put it to work, it is a great tool, despite being some 30+ years old.

1.6 Integer mode

Providing the HP-41 with an integer mode similar to the way it works on the HP-16C is the main design goal of this module. When the integer mode is enabled, both the keyboard and the display change behavior. Instead of showing floating point numbers, you will see binary integers in the number base and word size chosen.

In integer mode, your HP-41 behaves in the same way as you are used to, except that many keys perform integer operations instead of their previous floating point operations. This transformation takes place both inside, as well as outside user mode. As usual, you can make key assignments in user mode to override the default behavior.

Integer operations also work in program mode. You can write programs based on integer operations, just as you can write floating point programs. In fact, with some care you can actually intermix integer and floating point operations in a program.

1.7 License

The Ladybug software and its manual is copyright by Håkan Thörngren.

MIT License

Note: Ladybug was previously (2017-2019) released under the BSD 3-clause license, but it was changed starting with version 1A.

1.8 The name

The Ladybug name derives from that it is a useful tool for debugging at very low levels, using a calculator that is powered by Lady sized batteries. A ladybug is also a cute little animal.

1.9 Acknowledgments

Thanks to Robert Meyer for contributing the overlay for i41CX+ emulator (iPhone).

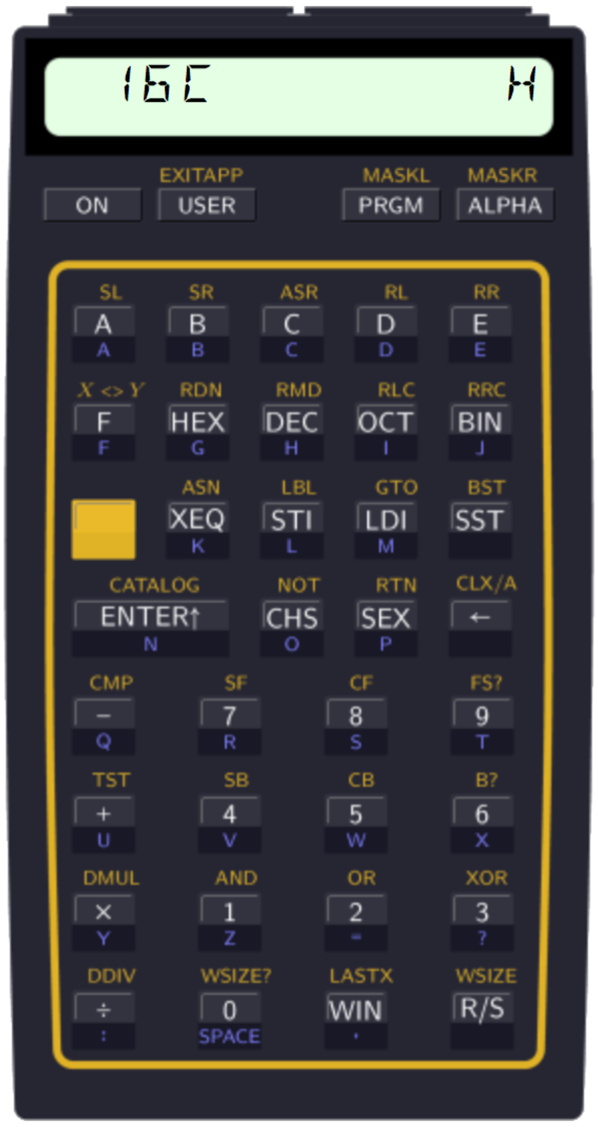
1.10 Feedback

Feedback and suggestions are welcome, the author can be contacted at hth313@gmail.com

CHAPTER 2

Application

Ladybug makes use of the application shell concept from the OS4 module to redefine the HP-41 keyboard to be somewhat different compared to normal, while still retaining essentially all behavior you are used to. The default display behavior is also changed to display integer number in the selected base.



The alpha mode keyboard is also slightly modified and hass the `ALDI` function where `ARCL` normally is.

2.1 Activation

Once Ladybug has been plugged into your calculator, simply execute the `INTEGER` instruction to enable integer mode.

2.2 Deactivation

Once activated, the HP-41 stays in integer mode until you execute the `EXITAPP` instruction. This can be done by pressing the shift key followed by the USER key. It is also possible to execute the `EXITAPP` instruction using the `XEQ` key and spell it out as usual.

Another way to disable Ladybug is to turn the calculator off and unplug the module.

The shifted USER key can also be used in alpha mode to exit integer mode.

2.3 Secondary functions

Ladybug makes use of the secondary functions feature of OS4. To access these functions you need the Boost module installed which contains replacements for the `XEQ` and `ASN` functions to allow keyboard access to the secondary functions in the same way as ordinary functions.

CHAPTER 3

Numbers and basic use

Fixed width integer numbers as used in computers are quite different from floating point numbers which aim to be an approximation of real numbers.

The fixed width indicates that we are working with numbers that have a predefined size. There is no such things as infinite numbers, instead numbers wrap around or behave in other ways when they overflow beyond their fixed width representation.

This chapter covers numeric bases, word size, 2-complement and ranges.

3.1 Numeric bases

Ladybug provides four numeric bases, binary (2), octal (8), decimal (10) and hexadecimal (16). Internally, numbers are represented as binary integers. The selected base controls the input and display of integers. The same binary number can be represented in any of the bases and it is easy to go between the bases. Thus, you can at any time during a calculation switch base to the one most suitable at that point. Any result (or partial result, thanks to RPN logic) can easily be shown in any available base.

Four keys (G–J) on the HP-41 keyboard represent `HEXS`, `DECS`, `OCTS` and `BINS` instructions that switch the base. Thus, switching base takes a single key press.

At any time a binary integer is displayed, it is displayed in the active base with a single letter at the right end of the display, indicating the active base.

The one exception to this is the `ALDI` instruction which appends the digits in the active base to the alpha registers, without the active base character.

3.2 2-complement

Binary integers can be interpreted as unsigned or signed. The signed mode used here is 2-complement[1] which is the most commonly used way of representing signed numbers on micro processors today.

[1] https://en.wikipedia.org/wiki/Two's_complement

In 2-complement mode, the leftmost (most significant) bit represents the sign.

In decimal mode, a negative number is displayed with a leading minus sign. In other bases, numbers are displayed without a leading minus sign, even when they are negative.

In word size 8, numbers range from -128 to 127 (decimal).

3.3 Unsigned

In the unsigned mode the leftmost bit adds magnitude to the number, not sign. This essentially doubles the value range.

In word size 8, numbers range from 0–255 (decimal).

3.4 Word size

You can specify word sizes from 1 to 64 bits using the `WSIZE` instruction. This controls the limit of numbers in the same way as is done on computers. You will often find that computers use 64, 32, 16 or even 8 bit values. If you are familiar with the internals of the HP-41, you have probably encountered other word sizes as well.

Note: Using instructions outside Ladybug, it is possible to place floating point values and alpha data on the stack, even when integer mode is enabled. This have no negative effect as stack values are adjusted to fit the current word size when integer operations are applied to them. Such non-integer data is preserved as long as you do not attempt to perform integer operations on them. The exception to this rule is that when you increase the word size, in which case all values on the stack are either a sign or zero extension depending on the sign mode. This is done to preserve the numerical meaning of integer values.

3.5 Windows

Depending on the number base and word size, the display may not be able to display the full number. In such cases the base letter will have a dot next to it indicating that there are more digits outside what is shown. To display such numbers you can use the window feature which is available on the dot key.

Pressing window (dot) gives a prompt for the window. The permitted range is 0–7, with 0 being the least significant part, which is what is shown by default. Simply press the desired window number to show the other parts of the number. The dots beside the base character gives feedback to whether there are more digits to the left or to the right of what is currently shown.

CHAPTER 4

Flags

Flags are fundamental to the integer mode. Flags are used to control behavior and give information about the result of operations.

Flags 0–5 are given special purpose by the integer mode. These flags are used because most are visible in the display, which makes it easy to see the flags while doing manual operations.

Some logic is intended in the assignment of the flag numbers, which makes it easier to remember them.

4.1 Mode flags

Two user flags control the mode. Setting flag 2 enables 2-complement signed mode. Clearing flag 2 enables unsigned mode.

Flag 5 is used to control zero fill of numbers. Zero filling means that all digits in the selected base are always shown. Leading zeros are used fill out the word if needed.

In decimal mode, zero fill has no effect.

If the number is larger than can be displayed, the window feature will enable a dot before the base character to indicate more digits available than is shown. This only happens if there is any non-zero digit above what is shown and works that way even in zero fill mode. This is to make it easy to see if there is anything of real interest there.

4.2 Carry flag

The carry flag is significant to most micro processors. It carries the bit out of a an addition, or the borrow into a subtraction. It is also used in shift operations as the spill bit shifted or rotated out.

Here it is assigned to flag 3, which resembles a C, if you straighten and mirror it.

4.3 Overflow flag

The overflow flag (called out-of-range flag in the HP-16C), tells if the operation overflowed. It is most commonly used in signed mode to signal that the result bits are not enough to hold the real result. For an addition and subtraction, it means that the sign of the result is incorrect (opposite). Flag 4 is assigned as the overflow flag.

4.4 Sign and zero flags

Sign and zero flags are very common on micro processors (but not present in the HP-16C). Here they are available with their usual meaning.

The zero flag is represented by flag 0 and is set when the last operation results in 0.

The sign flag is the most significant bit of the result. In signed mode this is the actual sign of the result. For decimal numbers, such numbers are displayed with a leading minus sign.

Flag 1 is used for sign, because it is a bit just copied from the result and it signals negative result when this bit is 1.

Note: The zero flag is set to indicate that the representable part of the result is zero. If you add two numbers so that they produce a carry out, but the lower parts consists of only 0 bits, the zero flag is set. Mathematically, the result is not zero, but this is how micro processors work, numbers wrap. You also have the carry flag and the overflag to inspect to interpret the result.

CHAPTER 5

Memory and storage

Instructions that manipulate memory work in many ways similar to its floating point counterparts, with a few key differences.

Accessing memory locations on the HP-41 breaks down to two categories. Numbered data storage registers and stack registers. Using synthetic programming, there is actually a third category, status registers, which allows access to the internal state of the HP-41.

In addition to this, you can also perform indirection on any of these locations to refer to a numbered storage register.

5.1 Nibble memory

In integer mode, the data storage size is based on the selected word size. The size required is rounded up to the nearest full nibble (multiple of 4 bits).

In the floating point mode, data storage registers are always 56-bits wide (14 nibbles).

The same data register area is shared by the floating point and integer modes.

In integer mode it means that with word size 8, each storage value needs 2 nibbles. Starting at floating point register 0, the first 7 (56/8=7) integer registers overlap with the first floating point register. Integer register 0 is in the rightmost two nibbles of floating point register 0, the following registers are allocated one by one as long as the are nibbles available.

In word size 32, nibble register 0 is inside floating point register 0, nibble register 1 takes the first 24 (56-32=24) bits from register 0, and 8 bits from floating point register 1.

Changing the word size effectively changes the boundaries used in the storage register area, but will not alter the contents of the memory in any way. It just changes the interpretation of the data memory area.

Mixing floating point registers operations and nibble register operations in the same program is possible, but will require some care.

5.2 Stack registers

Accessing stack registers in integer mode takes the extension part of the stack registers held in the I/O buffer in account and allows for accessing full 64-bit stack values.

5.3 Status registers

These are only available using synthetic programming and are treated as fixed 56-bit storage registers. Using them with integer mode makes most sense in word size 56, but they can be accessed in any word size. For larger word sizes, values are truncated to 56 bits. For smaller word sizes, values are truncated to the selected word size.

5.4 Register indirection

Register indirection in integer mode works conceptionally as before, but uses the integer value in the storage location to point to a numeric nibble memory register.

The range of nibble storage registers that can be accessed is 0–4095. Actual amount of available memory may put limitations on this, but for small word sizes it is possible to get quite far. With word size 4, addressing the entire allowed range requires 292 registers, which is possible on an HP-41.

5.5 Memory manipulation

In addition to `LDI` and `STI` to load and store integers, the `DECI` and `INCI` operations make it easy to add or subtract one to a register. As these operations also set zero and sign flags, it is simple to implement loops based on an integer counter. Loops may also be implemented using the `DSZI` which does both decrement and skip next instruction on zero.

CHAPTER 6

Instruction reference

This chapter goes through the instructions provided by Ladybug.

6.1 Mode related

The following instructions control mode settings. Mode settings are preserved if you switch out of integer mode and then back again.

Signed mode and zero filling are controlled by flags that are shared between integer and floating point modes. If you change such flag in floating point mode, it will affect the behavior in integer mode as well.

6.1.1 INTEGER

Switch to integer mode. The first time you enter integer mode the word size is set to 16 and the number base is 16 (hexadecimal).

Affected flags

Stack lift flag enabled.

6.1.2 EXITAPP

Leave integer mode by exiting Ladybug and switching back to floating point mode. This restores the keyboard and display to its normal floating point behavior.

Technically it leaves the current active application shell and goes back to the shell that was active before it. The very last such shell is the standard calculator behavior.

Integer instructions still work when the Ladybug application is inactive. What happens is that the Ladybug display and keyboard overrides are no longer active.

Affected flags

Stack lift flag enabled.

6.1.3 BINS

Enable base 2, work with binary integers.

Affected flags

None (stack lift flag left unchanged).

6.1.4 OCTS

Enable base 8, work with octal integers.

Affected flags

None (stack lift flag left unchanged).

6.1.5 DECS

Enable base 10, work with decimal integers.

Affected flags

None (stack lift flag left unchanged).

6.1.6 HEXS

Enable base 16, work with hexadecimal integers.

Affected flags

None (stack lift flag left unchanged).

6.1.7 WSIZE _

Set word size.

Affected flags

None (stack lift flag left unchanged).

6.1.8 WSIZE?

Return the active word size to X register.

Affected flags

Stack lift flag enabled.

6.1.9 SF 02

Enable signed 2-complement mode.

6.1.10 CF 02

Enable unsigned mode (disable signed 2-complement mode).

6.1.11 SF 05

Enable zero fill mode.

6.1.12 CF 05

Disable zero fill mode.

6.2 Stack operations

The integer stack shares the stack with the ordinary floating point stack. As integers larger than 56 bits will not fit in a stack register, extra storage on the side (the I/O buffer) is used to keep track of the extra bits. Ladybug provides a set of instructions that duplicate already existing stack manipulation operations, but which takes the stack register extension parts in account.

Hint: If you work in word size of 56 or less, you can actually use the corresponding built in stack manipulation instructions intended for floating point numbers instead. This is especially useful in a program as they takes less space compared to the integer mode counterparts.

6.2.1 ENTERI

Lift the stack, duplicate the number in X to Y and disable stack lift.

Affected flags

Stack lift flag disabled.

6.2.2 CLXI

Clear X and disable stack lift.

Affected flags

Stack lift flag disabled.

6.2.3 X<>YI

Swap X and Y registers.

Affected flags

Stack lift flag enabled.

6.2.4 LASTXI

Recall the last X register (L).

Affected flags

Stack lift flag enabled.

6.2.5 RDNI

Rotate the stack down one step.

Affected flags

Stack lift flag enabled.

6.2.6 R^I

Rotate the stack up one step.

Affected flags

Stack lift flag enabled.

6.3 Arithmetic operations

Instructions that perform some kind of calculation, i.e. arithmetic, logical and bit manipulation instructions, consume their arguments and place the result on the stack. The original value of X is placed in the L (Last X) register. If the instruction consumes more arguments from the stack than it produces, the stack drops and the contents of the top register (T) is duplicated.

6.3.1 ADD

Add X with Y, the result is placed in X and the stack drops.

Affected flags

Sign, zero, overflow and carry flags set according to the result. Stack lift flag enabled.

6.3.2 SUB

Subtract X from Y, the result is placed in X and the stack drops.

Affected flags

Sign, zero, overflow and carry flags set according to the result. Stack lift flag enabled.

6.3.3 MUL

Multiply X with Y, the result is placed in X and the stack drops.

Affected flags

Sign, zero and overflow flags set according to the result. The sign flag will have the correct value of the real result. Carry is not affected. Stack lift flag enabled.

6.3.4 DIV

Divide Y by X, the quotient is placed in X and the stack drops.

Affected flags

Sign, zero and overflow flags set according to the result. The sign flag will have the correct value of the real result. Carry set if remainder is non-zero, cleared otherwise. Stack lift flag enabled.

6.3.5 RMD

Divide Y by X, the remainder is placed in X and the stack drops.

Affected flags

Sign, zero and overflow flags set according to the result. Carry is not affected. Stack lift flag enabled.

6.3.6 NEG

Negate X.

In signed mode the smallest negative number does not have a corresponding positive counterpart. Negating that number ends up with the same number as the input. In this case the overflow flag is set to indicate that the result could not be represented. For all other signed values, the input is negated and the overflow flag is cleared.

In unsigned mode, the number is negated, giving the same bit pattern as would result in signed mode. However, as all numbers are considered positive, a negative number can not be represented and the overflow flag will be set to indicate this. The only case you will not get an overflow flag is when the input is 0 (as 0 negated is also 0).

Affected flags

Sign, zero and overflow flags set according to the result. Stack lift flag enabled.

6.3.7 ABSI

Absolute value of X.

In signed mode, negative numbers are negated to make them positive. As negation does the same code as `NEG`, see `NEG` for a discussion on how the smallest negative number behaves.

In unsigned mode all numbers are considered positive, and negation is never done. The overflow flag is always cleared in this case.

Affected flags

Sign, zero and overflow flags set according to the result. Stack lift flag enabled.

6.4 Double operations

Multiplication and divide are also available in double versions.

6.4.1 DMUL

Multiply X with Y, the double result is placed in X and Y (high part in X).

Affected flags

Sign and zero flags set according to the result. The sign flag will have the correct value of the result. Overflow flag is cleared. Stack lift flag enabled.

6.4.2 DDIV

Divide the double value in Z and Y (high part in Y) by X. The double quotient result is placed in X and Y (high part in X). Stack drops one step.

Affected flags

Sign and zero flags set according to the result. Overflow flag is cleared. Carry set if remainder is non-zero, cleared otherwise. Stack lift flag enabled.

6.4.3 DRMD

Divide the double value in Z and Y (high part in Y) by X. The single precision remainder result is placed in X. Stack drops two steps.

Affected flags

Sign, zero and overflow flags set according to the result. Carry is not affected. Stack lift flag enabled.

6.5 Logical operations

6.5.1 AND

Logical AND between X and Y, result is placed in X and the stack drops.

Affected flags

Sign and zero flags set according to the result. Stack lift flag enabled.

6.5.2 OR

Logical OR between X and Y, result is placed in X and the stack drops.

Affected flags

Sign and zero flags set according to the result. Stack lift flag enabled.

6.5.3 XOR

Logical XOR between X and Y, result is placed in X and the stack drops.

Affected flags

Sign and zero flags set according to the result. Stack lift flag enabled.

6.5.4 NOT

Bitwise NOT (negation) X, makes all bits the opposite.

Affected flags

Sign and zero flags set according to the result. Stack lift flag enabled.

6.6 Shift operations

6.6.1 SL _

Shift X left by the given number of steps. The most recently shifted out bit is placed in the carry bit.

Postfix argument

The number of steps to shift, or a register indirection to a nibble register which holds the number of steps to shift. Valid range is 0–64.

Affected flags

Sign and zero flags set according to the result. Carry holds the last shifted out bit. Stack lift flag enabled.

6.6.2 SR _

Shift X right by the given number of steps. The most recently shifted out bit is placed in the carry bit.

Postfix argument

The number of steps to shift, or a register indirection to a nibble register which holds the number of steps to shift. Valid range is 0–64.

Affected flags

Sign and zero flags set according to the result. Carry holds the last shifted out bit. Stack lift flag enabled.

6.6.3 RL _

Rotate X left by the given number of steps. Bits going out at the left end appear again at the right hand side. In other words, bits are rotated around. The most recently bit that wrapped around is also copied to the carry.

Postfix argument

The number of steps to shift, or a register indirection to a nibble register which holds the number of steps to shift. Valid range is 0–64.

Affected flags

Sign and zero flags set according to the result. Carry holds the last shifted out bit. Stack lift flag enabled.

6.6.4 RR _

Rotate X right by the given number of steps. Bits going out at the right end appear again at the left hand side. In other words, bits are rotated around. The most recently bit that wrapped around is also copied to the carry.

Postfix argument

The number of steps to shift, or a register indirection to a nibble register which holds the number of steps to shift. Valid range is 0–64.

Affected flags

Sign and zero flags set according to the result. Carry holds the last shifted out bit. Stack lift flag enabled.

6.6.5 RLC _

Rotate X left by the given number of steps through carry. A bit that is rotated out goes to the carry, the previous carry is rotated in at the right hand side.

Postfix argument

The number of steps to shift, or a register indirection to a nibble register which holds the number of steps to shift. Valid range is 0–64.

Affected flags

Sign and zero flags set according to the result. Carry holds the last shifted out bit. Stack lift flag enabled.

6.6.6 RRC _

Rotate X right by the given number of steps through carry. A bit that is rotated out goes to the carry, the previous carry is rotated in at the left hand side.

Postfix argument

The number of steps to shift, or a register indirection to a nibble register which holds the number of steps to shift. Valid range is 0–64.

Affected flags

Sign and zero flags set according to the result. Carry holds the last shifted out bit. Stack lift flag enabled.

6.6.7 ASR _

Aritmetic right shift. This duplicates the sign bit as the number is shifted right. The most recent shifted out bit is placed in the carry.

Postfix argument

The number of steps to shift, or a register indirection to a nibble register which holds the number of steps to shift. Valid range is 0–64.

Affected flags

Sign and zero flags set according to the result. Carry holds the last shifted out bit. Stack lift flag enabled.

6.7 Bitwise operations

6.7.1 MASKL _

Create a left justified bit mask (all bits set), of the width specified in its argument.

A width of 0 results in 0, a width of 64 results in all bits set regardless of the active word size.

Postfix argument

The width of the mask, or a register indirection to a nibble register which holds the width of the mask. Valid range is 0–64.

Affected flags

Stack lift flag enabled.

6.7.2 MASKR _

Create a right justified bit mask (all bits set), of the width specified in its argument.

A width of 0 results in 0, a width of 64 results in all bits set regardless of the active word size.

Postfix argument

The width of the mask, or a register indirection to a nibble register which holds the width of the mask. Valid range is 0–64.

Affected flags

Stack lift flag enabled.

6.7.3 SEX _

Sign extend the value in X by the word width specified in its argument.

```
SEX 08
```

Will interpret the value in X as a signed 8-bit value. If it is negative, the value is sign extended to fit the active word size.

Postfix argument

A word size, or a register indirection to a nibble register which holds the word size. Valid range is 1–64.

Affected flags

Sign and zero flags set according to the result. Stack lift flag enabled.

6.7.4 CB _

Clear a single bit in X as specified by the argument.

Postfix argument

A bit number, or a register indirection to a nibble register which holds the bit number. Valid range is 0–63.

Affected flags

Sign and zero flags set according to the result. Stack lift flag enabled.

6.7.5 SB _

Set a single bit in X as specified by the argument.

Postfix argument

A bit number, or a register indirection to a nibble register which holds the bit number. Valid range is 0–63.

Affected flags

Sign and zero flags set according to the result. Stack lift flag enabled.

6.7.6 B? _

Test if a bit of X is set, skip next instruction in a program if the bit is not set. In keyboard mode, the result is displayed as `YES` or `NO`.

Postfix argument

A bit number, or a register indirection to a nibble register which holds the bit number. Valid range is 0–63.

Affected flags

Stack lift flag enabled.

6.7.7 BITSUM _

Count the number of bits in X and place that number in X.

Postfix argument

A register, or a register indirection to a nibble register.

Affected flags

Sign and zero flags set according to the result. Stack lift flag enabled.

6.8 Compare and test

Comparing values with Ladybug offers a way that is more like it works on machine instruction sets, which differs from what you may be used to on an HP calculator. Instead of comparing X to Y, or X to 0, you test flags set by the previous operation. There are three variants to this:

1. To compare two numbers, use the `CMP` instruction which works similar to a compare on a microprocessor. It performs a subtraction, setting flags according to the result and discards the numerical result. The actual comparison between two numbers starts with a `CMP`, followed by a flag conditional operation which conditionally skips the following instruction.
2. To compare to 0, use the `TST` instruction followed by a test of flag 0.
3. Furthermore, arithmetic and bit manipulation instructions set flags according to the result, making it possible to just test suitable flags after such operation.

There is now also a set of HP-41 compare instructions (`=I`, `≠I`, `<I` and `<=I`). In program mode they either execute the following line or skips it, depending on the outcome of the test. In keyboard mode `YES` or `NO` is displayed. Current sign mode is obeyed.

Here are the provided instructions that are related to comparing values:

6.8.1 CMP _

The argument specifies a register value that is subtracted from X. The result is dropped, but flags are set according to the result. Useful for comparing X to any value.

Postfix argument

A register, or a register indirection to a nibble register.

Affected flags

Sign, zero, overflow and carry flags are set according to result of the subtraction. Stack lift flag enabled.

6.8.2 TST _

The argument specifies a register value that will affect the sign and zero flags. Useful for testing if any register value is zero, positive or negative.

Postfix argument

A register, or a register indirection to a nibble register.

Affected flags

Sign and zero flags set according to the value in the register. Stack lift flag enabled.

6.8.3 GE?

Perform next instruction in a program if the previous `CMP` instruction indicates that X is greater than or equal to the other value, otherwise skip next line. Current sign mode is obeyed. In keyboard mode, `YES` or `NO` is displayed.

Affected flags

Stack lift flag enabled.

6.8.4 GT?

Perform next instruction in a program if the previous `CMP` instruction indicates that X is greater than the other value, otherwise skip next line. Current sign mode is obeyed. In keyboard mode, `YES` or `NO` is displayed.

Affected flags

Stack lift flag enabled.

6.8.5 LE?

Perform next instruction in a program if the previous `CMP` instruction indicates that X is less than or equal to the other value, otherwise skip next line. Current sign mode is obeyed. In keyboard mode, `YES` or `NO` is displayed.

Affected flags

Stack lift flag enabled.

6.8.6 LT?

Perform next instruction in a program if the previous `CMP` instruction indicates that X is less than the other value, otherwise skip next line. Current sign mode is obeyed. In keyboard mode, `YES` or `NO` is displayed.

Affected flags

Stack lift flag enabled.

6.8.7 =I _ _

Test if two register operands are equal

Two postfix arguments

This function performs an equality compare between two registers. In program mode it skips over the next instruction if the two operands are not equal. In keyboard mode it displays `YES` or `NO`. This is a secondary function.

Affected flags

Stack lift flag enabled.

6.8.8 ≠I _ _

Test if two register operands are not equal

Two postfix arguments

This function performs an equality compare between two registers. In program mode it skips over the next instruction if the two operands are equal. In keyboard mode it displays `YES` or `NO`. This is a secondary function.

Affected flags

Stack lift flag enabled.

6.8.9 <I _ _

Test if the first register operand is less than the second register operand

Two postfix arguments

This function performs an less-than compare between two registers, obeying current sign mode. In program mode it skips over the next instruction if the test is not true. In keyboard mode it displays `YES` or `NO`. This is a secondary function.

Affected flags

Stack lift flag enabled.

6.8.10 <=I _ _

Test if the first register operand is less than or equal to the second register operand

Two postfix arguments

This function performs an less-than-or-equal compare between two registers, obeying current sign mode. In program mode it skips over the next instruction if the test is not true. In keyboard mode it displays `YES` or `NO`. This is a secondary function.

Affected flags

Stack lift flag enabled.

Note: The two operand compare operations takes allows for comparing two arbitrary register operands. If you want to compare greater-than, simply swap the operands and use the corresponding less-than function.

6.9 Memory related instructions

6.9.1 LDI _

Load X from the specified register.

Postfix argument

A register, or a register indirection to a nibble register.

Affected flags

Sign and zero flags set according to the value loaded. Stack lift flag enabled.

6.9.2 STI _

Store X in the specified register.

Postfix argument

A register, or a register indirection to a nibble register.

Affected flags

Stack lift flag enabled.

6.9.3 <>I _ _

Exchange between two registers

Two postfix arguments

This function performs a register to register exchange, using arbitrary registers, or register indirect operands. This is a secondary function.

Affected flags

Stack lift flag enabled.

6.9.4 VIEWI _

View the specified register without affecting the stack.

Postfix argument

A register, or a register indirection to a nibble register. This is a secondary function.

Affected flags

None

6.9.5 DECI _

Subtract one from the register specified in the argument.

Postfix argument

A register, or a register indirection to a nibble register.

Affected flags

Sign and zero flags set according to the new value. Stack lift flag enabled.

6.9.6 DSZI _

Subtract one from the register specified in the argument, skip next instruction if the result is zero. This is useful for implementing loops. Flags are not affected.

Postfix argument

A register, or a register indirection to a nibble register.

Affected flags

Stack lift flag enabled.

6.9.7 INCI _

Add one to the register specified in the argument.

Postfix argument

A register, or a register indirection to a nibble register.

Affected flags

Sign and zero flags set according to the new value. Stack lift flag enabled.

6.9.8 CLRI _

Clear the contents of the specified register.

Postfix argument

A register, or a register indirection to a nibble register.

Affected flags

Stack lift flag enabled.

6.10 Miscellaneous instructions

6.10.1 ALDI _

Append a register value to the alpha register obeying the current word size, selected base, active sign mode and zero fill flag.

Postfix argument

A register, or a register indirection to a nibble register.

Affected flags

Stack lift flag enabled.

6.10.2 WINDOW _

This instruction makes it possible to view different parts of a number that is too large to show in the display. Dots around the base character indicates whether there are digits not shown on either side of the currently shown window. This is a non-programmable instruction to make it possible to inspect numbers (literals) in program mode as well.

Postfix argument

The window number, 0–7. The rightmost window is 0, which is what is shown by default.

CHAPTER 7

Prompting instructions

The HP-41 operating system allows for extension (XROM) instructions to prompt for arguments, but this only works when executed from the the keyboard. There is no support for storing such instructions together with its operand in programs.

This module offers an extension called semi-merged instructions that allows XROM instructions to have operands in programs. It is called semi-merged because the instructions are not fully merged like the built in prompting instructions.

Executing a prompting instruction will cause it to prompt for its argument. Outside program mode, this is no problem as it was always supported. In program mode, the instruction and its postfix operand will be inserted in the program as a semi-merged instruction.

7.1 Postfix operands

The postfix operand is a byte that cannot be safely stored directly in the program memory. The reason is that it can be any byte, even the first byte of a multi-byte instruction, which would consume innocent bytes following it, potentially making a total mess of the program memory.

The solution used here is to wrap the operand byte in a wrapper that makes it safe. The wrapper used is the alpha literal wrapper. Thus, an instruction such as `SL` (shift left) consists of two visible instructions in program memory:

```
10 ...
11 SL 36
12 "$"
13 ...
```

Note that the `SL 36` instruction is fully shown as a merged instruction, but it is followed by its wrapped postfix byte (36 corresponds to ASCII $).

This also works when using indirect addressing:

```
10 ...
11 SL IND Z
12 "*"
13 ...
```

As the SL instruction is a two byte XROM instruction followed by a single letter ASCII constant, the whole instruction requires four bytes of program memory.

When executed, the text literal is simply skipped and has no effect on the alpha register.

Note: It is intentional that the postfix byte is shown. While it can be possible to hide it somewhat, it is judged to be better to actually show what is going on. This provides better control over program memory editing, as the postfix part actually does take a program step and will not be considered merged when following an instruction that skips the next line. (You may still be able to use it after such skip instruction, but it will execute the text literal in this case, altering the alpha register.)

7.1.1 Default operand

If the postfix operand is missing, the instruction reverts to a default behavior. For a shift instruction, it means shift one step:

```
10 ...
11 SL 01
12 ...        ; not a single letter text literal
```

Such instruction costs two bytes (the XROM itself without any postfix operand). It executes as a single shift as shown. As it is a single instruction, it also works well following a test instruction.

If you enter the `SL 01` instruction, it takes advantage of the default and does not store a postfix byte in program memory.

If you delete the postfix operand from program memory, the instruction that used it will change to its default behavior, which can be seen when the instruction is shown.

Note: Some care is needed when using default behavior with prompting instructions. It will still look for its argument and if you have a single character alpha constant that you intended to be an alpha constant, then it will become part of the previous instruction. This should seldom happen, but if it does, the easiest way to deal with it is by rearranging instructions.

7.2 Single stepping

When you single step a semi-merged instruction in run mode (to execute the program step by step), it works properly, but visual feedback of the instruction when the SST key is pressed and held, is just the bare instruction without any postfix operand.

7.3 Integer literals

To store an integer literal in a program, just type it in when you are in program mode. This takes the selected base in account, word size 64 and no zero filling. This is because it cannot really know what the word size will be when the program is executed later.

To enter an integer literal in another base, switch out of program mode, change the base and switch program mode back on.

Integers in programs are always displayed using the current base. If you enter a hexadecimal number at one point, then edit the program at a later point in decimal mode, you will see the hexadecimal number displayed as a decimal number.

Numbers that are to need more than 8 digits will turn on the dot by the base character to indicate that there are more digits than is shown. The window key can be used to inspect other parts of the number, just as you can do outside program mode.

Storing integer literals in a program works in a similar way as prompting instructions. A special `#LIT` instruction is used to prefix the literal, and the literal is encoded as a binary alpha string on the following line.

If you single step past the shown integer literal, the alpha literal is shown:

```
10 ...
11 F80     H
12 "**"
13 ..
```

The default behavior for `#LIT` is to act as `0`. As the postfix alpha literal can be of variable length, it is somewhat more likely to end up interfering with a following alpha literal in a program compared to the single byte postfix instructions.

Note: As program editing sometimes can be a bit slow on the HP-41 and you may briefly see the `#LIT` instruction. The name was picked to avoid clashes with other things, yet give some hint what it is about when briefly seen.

CHAPTER 8

Programming examples

This chapter goes through some programming snippets and examples to give a feeling about how the instructions work.

8.1 Left justify

To left justify the number, one can take advantage of the sign flag which will be set when the leftmost bit is set. To avoid an infinite loop, one need to test the input value in X for zero. The following code snippet performs a left justify:

```
TST X     ; set sign and zero flags according to X
FS? 00    ; zero?
GTO 00    ; yes, done
LBL 01    ; no, loop to left justify
 FS? 01   ; negative?
 GTO 00   ; yes, done
 SL  01   ; shift left, setting flags
 GTO 01
LBL 00
```

CHAPTER 9

Differences to the HP-16C

Being inspired by the HP-16C there are a lot of similarities between the Ladybug module for the HP-41 and the HP-16C. Apart from the obvious differences, such as form factor, battery life, alpha capabilities on the HP-41 and dedicated decorated keyboard on the HP-16C, there are other less obvious differences, many of which are discussed in this chapter.

9.1 Flags

Flags are arranged differently. The reason for this is the annunciators 0-4 on the HP-41 that can be used to provide additional feedback compared to the HP-16C. The arrangement was inspired by making it logical, rather than trying to be compatible with the HP-16C.

The sign and zero flags are very common on micro processors and gives additional feedback, being shown by the annunciators. It also allows for implementing compares in a way more similar to micro processors, which is discussed next.

9.2 Compares

Compares on the HP-16C is done using traditional HP calculator operations, by providing a set of operations that compares between X and Y, and X to 0. Ladybug works more like a micro processor. It provides a `CMP` instruction that compares X to any register value and set flags accordingly. After `CMP` you can interpret the result using a flag test operation `FC?`, `FS?` or some of the built in relation test instruction like `LT?` and friends. It is also possible to use the `TST` instruction, or rely on the flags set after doing an operation.

Note: `CMP Y` subtracts Y from X, which is the opposite order compared to `SUB`.

9.3 Co-existing modes

The HP-16C keeps the floating point and integer modes separate. Changing between the modes affect stack contents and word sizes. On the HP-41 the integer mode is basically a keyboard and display shell, the functional behavior remains the same when you switch mode.

As a result, the mode separation is not as strict and stack register values are not normally pruned when changing mode, as we will discuss next.

9.4 Word size change affecting stack

The HP-16C truncates values on the stack according to the active word size. Changing a word size to a smaller one and then back will not preserve all value bits. Ladybug only alters value bits on the stack when going to a *larger* word size. Reducing to a smaller word size will *not* affect value bits on the stack.

Increasing the word size will affect all registers on the stack with Ladybug. In signed mode, values are sign extended, and in unsigned mode values are zero extended. This is done to preserve numerical values.

It is possible to keep 56-bit floating point values on the stack, provided that the word size is not increased. This allows for mixing floating point operations with integer mode.

The reason for doing it this way, is that the HP-41 can perform floating point operations at any time. Keeping the stack properly masked would be a quite elaborate task and would get in the way with the ability to keep floating point values around. The HP-16C has an easier task here, as it has a more strict separation between floating point mode and integer mode. On the positive side, you gain the ability to have both integers and floating points on the stack at the same time.

Numbers are in general masked as needed when they are used as input to operations in Ladybug, not because they are laying around somewhere.

9.5 Postfix operands

Ladybug takes advantage of the prompting instructions on the HP-41 to allow for accessing stack registers in the same ways as numeric registers. Indirection can also be done on any register, not just the single index register as on the HP-16C.

9.6 Zero fill mode

With Ladybug, the zero fill mode does not indicate available digits above the current window if they are all zero. The HP-16C will always indicate that there is more to see, even if it only filled 0 digits.

The reason for this difference is that it is believed that instantly knowing if there is anything non-zero to see outside the display is more useful than to be constantly reminded that the word size is actually larger than what can be shown in a single display.

9.7 One complement mode

The one complement mode is not present in Ladybug.

9.8 Window display

The window display only provides for moving a full window at a time, not by single digits which is also available on the HP-16C.

The keyboard layout to do this does not require pressing a shift key, which makes it somewhat easier to work with windows with Ladybug, compared to the HP-16C.

9.9 Double divide

Double divide will result in a double quotient. The HP-16C gives a single word quotient, or an error if a double result would have been needed. Giving the full quotient is believed to be more useful, but changes may be needed to HP-16C programs that uses `DDIV`.

9.10 Machine status

There is currently no machine status display in Ladybug. Most of the information about the status is already visible in the display, the rest can be queried using `WSIZE?` or `FS? 05` for zero fill mode.

9.11 Square root

Ladybug does not offer an integer square root function, which is present on the HP-16C.

9.12 Floating point conversions

There are no support for floating point number conversions built in to Ladybug at this point. It is something that is considered for a future extension.

9.13 Prompting instructions

Ladybug takes full advantage of the prompting facility of the HP-41. Instructions such as `MASKL` and `WSIZE` prompt for their argument and are not limited to take it from the X register. To get the same behavior as on the HP-16C, use the indirect X postfix argument:

```
WSIZE IND X
```

However, for `MASKL` which takes two values on the HP-16C, such straight translation would not work as the instruction would take the the stack registers in opposite order.

In most cases you will probably just use a postfix numeric argument rather than a register indirection:

```
MASKL 4
```

Shift operations prompt for the shift count, which makes it unnecessary to have two instructions to implement the same shift operation, compared to the HP-16C.

Note: No savings would be made by making two instructions, as the default behavior of the semi-merged shift instructions is to shift by 1. In other words, the shift instructions do dual duty as shift by one and shift by arbitrary number of steps.

9.14 Left justify

Is currently not present in Ladybug.

CHAPTER 10

Implementation

This chapter aims to give an understanding of how the HP-41 has been extended to work with integer numbers.

10.1 I/O buffer

The HP-41 stack uses 56-bit values. Such values are most often floating point numbers, but alpha data and non-normalized numbers are also possible.

To make it possible to work with up to 64-bit integers, an I/O buffer is used to store the extra bits. The lower 56 bits are stored in the ordinary stack and the buffer provides storage for the remaining bits.

A buffer is a private storage area allocated from the free memory area, much like Time module alarms.

Stack space is shared with the ordinary floating point stack and the extra bits (when needed), are kept in the buffer. Instructions that work on the stack, such as `ENTER` and `RDN` have an integer counterpart instruction that works on the full 64-bit value.

Note: If you are using a word size of 56 or less, you can use the floating point counterpart instructions for stack manipulations, as they behave the same. This makes most sense in a program as those instructions are shorter (occupy less memory space) and execute faster.

As the lower 56 bits are stored in the normal stack registers, it makes it easier to interact with the ordinary 56-bit calculator registers for special purposes. The main disadvantage is that instructions such as `CAT 4`, `TIME` and `DATE` that leaves a number in X register, which will corrupt the integer stack if word size is larger than 56.

Hint: If you corrupt the stack in this way, you can usually recover most of it by executing the floating point `RDN` instruction.

In addition to the extra bits for the stack, the buffer keeps track of all other things related to the integer mode. The complete integer state is preserved when you turn your calculator off. In fact, turning the HP-41 off and on will not cause the HP-41 to leave the integer mode. Integer mode stays active until you make an explicit switch to floating point mode (just press the gold shift key and the `PI` (0) key to get out of integer mode).

If you unplug Ladybug, the next time you turn the HP-41 on, the HP-41 will reclaim the buffer registers and make them available in the free memory pool.

10.2 Keboard layout

The keyboard layout uses the existing similar functions on keys whenever possible. However, the stack manipulation operations `RDN` and `X<>Y` are moved to shifted keys as it was judged more important to have access to all digit entry keys without pressing shift.

The base change keys are ordered in the same way as on the HP-16C and are located after the `F` digit.

The window key is the dot key. It was selected because the display uses dots to indicate presence of more windows. It is also very close to the 0–3 keys, so you will find the typical argument keys close to the window selection key.

The bit set, clear and test instructions are located on the row below the corresponding flag operations.

Negation is the same thing as `CHS`, bitwise `NOT` is on the same key (shifted) as it is closely related to negation.

Shift and rotate operations are plentiful. They are arranged together on the upper part of the keyboard.

Double `MUL` and `DIV` operations are the shifted variant of the corresponding key.

`CMP` is actually a subtract operation, so it is on the shifted `SUB` key. `TST` is related to `CMP`, so it was placed next to it. These keys are also where comparisons are on the floating point keyboard, so that also makes some sense.

The logical operations are held together in the usual order you say them, AND, OR, XOR, which also is the alphabetical order.

Sign extension is on the `EEX` key, which is almost spelled the same.

The shifted `USER` key performs the `EXITAPP` function which exists the Ladybug shell and takes you back to the previous shell. This key was chosen as it is always free and the `USER` key is related to affecting the keyboard layout.

CHAPTER 11

Limitations

This chapter covers known limitations with the current implementation.

11.1 Printer

A printer, especially in trace mode, does not work well with integer mode.

11.2 Auto assigned keys

The HP-41 performs automatic assignment on the two top row keys to correspond to labels in the current program. This may slow down response to such keys significantly.

In integer mode, this feature is disabled as the top row keys serve as digit entry for hexadecimal values, and you most likely want digit entry to always be responsive.

If you want to use the auto assign feature, switch to floating point mode.

Index