DUNK Assembly

David Farrell

July 14, 2025

1 About

Note: in the current state, DUNKasm is only compatible with linux-based systems. Compilation is only supported by gnu make, windows-style filepaths are not supported, and the terminal output uses unix colour codes.

2 Compiling, Installing and Using DUNKasm

To compile DUNKasm, simply run `make` in the folder `utils/dunkasm`. To install, run `sudo make install`. This copies the binary to `/usr/bin`. Syntax highlighting for DUNKasm can be enabled in gtksourceview-based rtext editors such as Pluma by running `install_lang`. The text editor will typically need to be restarted for this to take effect.

If you have any issues with compilation or installation, feel free to contact me at `davidjfarrell96@gmail.com` or create an issue on github.

DUNKasm source files may be assembled to executable binary via the command

```
dunkasm [input files] -o [output filename]
```

The files are processed in order, with the first listed considered the "main file", and contains the code that will run when the binary is executed. The code contained in the main file is appended with a `halt_and_catch_fire` instruction, so that execution halts after it is executed. This can be disabled via the option `--no-hcf`.

If no output path is specified, the output will be written to `[filename]-assembled`. The default output format is v3.0 Hex words addressed, which is a text format describing a binary file in hexadecimal. This is done so that binaries can be imported into logisim with no additional clicks. The `-b` flag switches the output format to raw binary. Additionally, the flag `-v`, equivalently the option `--verbose`, instructs the assembler to output detailed information to the terminal.

3 Preservation Of Registers

The first eight general-purpose registers are preserved accross function calls. For example, after running the below code, the value of `r0` is still `0xdead`.

```
set r0 0xdead
call function
halt_and_catch_fire

function:
        set r0 0xbeef
        return
```

This is a convenience feature, although it is achieved by a process, "stacking", of pushing the values of any of the preserved registers which are modified between functions calls to the stack, as well as an additional word, the "stacking mask", detailing which registers were preserved. This induces an overhead in function calls and returns. If one has no need for registers to be preserved accross a given call, the `fastcall` instruction can be used. However, fastcall still must include an extra `push` to indicate which registers are preserved (none), otherwise it would break `return`. It was decided not to include a `fastreturn` instruction which does not expect the extra word, as this would result in two mutually incompatible classes of instructions, and code duplication. The slowdown was accepted as a valid tradeoff in light of the convenience provided by automatically (and optionally) preserved registers.

Another side effect of the register preservation is that, since the stack contains additional data, of non-deterministic length, arguments cannot be passed using the stack, as they are not in a consistent position relative to the value of `sp` at the time of the function's execution. Therefore the general-purpose registers `r[8-f]` are, by convention, used for argument passing, and return values. They are given special aliases for this purpose.

4 Syntax

The syntax of DUNKasm is designed to be more human-readable than the typical assembly language. For instance, instruction names are not abbreviated. Additionally, the assembler has support for *aliases* - essentially, rudimentary macros, which may be defined and undefined inline, allowing registers, memory addresses, and so on, to effectively be given variable names.

The syntax of DUNKasm is line based; every line of a source file is treated as a separate entity, expected to either be empty, or contain exactly one (preprocessor/assembler) directive, one symbol definition, or one instruction, with the appropriate number of arguments given.

Directives and arguments have the same syntax; the name is given, followed by all arguments, separated by spaces. The syntax for symbol definitions is to give the symbol name followed by a colon. It is not permitted to follow a symbol definition with an instruction or directive on the same line. Consider the following code snippet defining a function which computed the remainder of one (unsigned) integer on division by another.

```
    include "hello_world.dasm"

alias x r0
alias y r1

remainder:
    set x argument1
    set y argument2

    remainder_loop:
            subtract x y

            goto_if_less x y remainder_done % if x < y, we are done

            goto remainder_loop

    remainder_done:
            set result1 x
            return
```

4.1 Instructions & Their Arguments

DUNK instruction admit 0-3 arguments. An argument may be either a constant (or *immediate* value), a register, or a pointer, which is either described by a constant or a register. Additionally, string literals can be used as arguments; under the hood, string literals are replaced by their address in the strings section. A string literal may be given wherever the assembler expects a constant.

A constant may be given either as a number, expressed (by default) in decimal, or in hexadecimal or binary using the prefixes `0x` and `0b`, an ASCII character, wrapped in single-quotes, a string literal, wrapped in double quotes, a symbol, or an alias which reduces to one of the other valid constant expressions.

A register can be specified by either `rN` or `srN` or an alias, where `N` is a single hexadecimal digit, i.e., `0-f`. There are 32 registers that may be referred to in this way; 16 general purpose registers, `r0-rf`, and 16 special-purpose registers, `sr0-srf`. Many registers are given special aliases, listed below. Note that some general-purpose registers have special-purpose alises given. These are still considered general purpose registers; these alises are just for convention, and may be safely ignored. The special purpose registers, on the other hand, may mutate unexpectedly (especially `sr4`) or affect the functioning of DUNK in unexpected ways if used for general data-handling purposes, and this is discouraged.

Special register aliases:

- `pk`: `sr0`,

- `sp`: `sr1`,
- `interrupt_handler1`: `sr8`,
- `interrupt_handler2`: `sr9`,
- `interrupt_handler3`: `sra`,
- `interrupt_handler4`: `srb`,
- `interrupt_handler5`: `src`,
- `interrupt_handler6`: `srd`,
- `interrupt_handler7`: `sre`,
- `interrupt_handler8`: `srf`.

General purpose register aliases:

- `argument`: `r8`,
- `result`: `r8`,
- `argument1`: `r8`,
- `argument2`: `r9`,
- `argument3`: `ra`,
- `argument4`: `rb`,
- `argument5`: `rc`,
- `argument6`: `rd`,
- `argument7`: `re`,
- `argument8`: `rf`,
- `result1`: `r8`,
- `result2`: `r9`,
- `result3`: `ra`,
- `result4`: `rb`,
- `result5`: `rc`,
- `result6`: `rd`,
- `result7`: `re`,
- `result8`: `rf`,

Pointers are indicated by an asterisk. A pointer can be given by a constant or a register. In the case of a register, an offset can optionally be given. The syntax for this is `*([register][+/-][offset])`. The parentheses are mandatory; for example, `*r2+3` will yield an error. The offset can be given with either a plus or a minus, and the standard syntax for constants applies, although string literals are not permitted. For example, the following is valid dunkasm.

```
set *sp "Error!"
set *0xbeef 0xdead

alias OFFSET 0xa
set *(r3-OFFSET) 0b1101001
set r4 *(sp+3)
```

4.2 Directives

4.2.1 Preprocessor Directives

The preprocessor accepts four directives: `include`, `include_first`, `alias` and `dealias`.

4.2.2 Including Files

From one dunkasm source file, others can be included. The preprocessor directive `include` instructs the assembler to add the specified file to the list of files to assemble, allowing access to the symbols defined therein. In effect, the content of the included file is appended to the content of the including file. The assembler will, however, ignore the inclusion of any files, with a warning. Double-include warnings can be turned off via the option `--permit doubleinclude`, or elevated to errors via the option `--error doubleinclude`. Additionally, the first file

The directive `include_first` will, in effect, cause the content of the included file to be *pre*-pended to the current. This is done in the order in which `include_first` directives are encountered.

4.2.3 Aliases

The directive `alias` allows one to define a macro that will be replaced identically by its definition, inline, by the assembler. The syntax is `alias [replacee] [replacer]`. For example, the code

```
alias stack_2nd_word *(sp+1)
alias x r0

    set x stack_2nd_word
```

is processed by the assembler identically to the single line

```
set r0 *(sp+1)
```

whose effect is to store the value of the word stored in the second (from the top) position on the stack into the register `r0`.

4.2.4 Asembler Directives

Currently, there is only one directive, `word`, which actively modifies the output binary instead of directing the preprocessor to insert or modify code prior to compilation. The directive `word` accepts one argument, a constant, and simply inserts the given word at that point in the code. An example of the use of this can be found in `roms/firmware.dasm`, which begins with a series of words giving the addresses of the system call functions provided in the firmware. The `syscall` instruction works by fetching the appropriate address from this initial segment of the firmware ROM and then calling the function at that address.

The final directive is symbol definition. To define a symbol, on a new line, the symbol name is given, which must consist only of letters, numbers and underscores, followed by a colon. Any instance of the symbol appearing in the code will then be replaced, in the resulting binary, with the address of the first instruction following the symbol definition in the code. One can uniformly apply an offset to the symbol addresses appearing in the binary using the `--offset` option, whose syntax is `--offset [offset]`. This is used, for example, to generate the firmware ROM: the firmware ROM is memory-mapped, with the address of the first word in the ROM being `0xf000`. Therefore, for the `syscall` instruction to work, `firmware.dasm` must be compiled with `--offset 0xf000`.

5 Instructions

5.1 Control

5.1.1 nop

Arguments: none.

5.1.2 goto

Sets the program kounter equal to the given constant and proceeds execution from there.

Arguments: `constant`.

5.1.3 goto_if_zero

Sets the program kounter equal to the given constant and proceeds execution from there, provided that the given register contains 0.

Arguments: `[register], [constant]`.

5.1.4 goto_if_nonzero

Sets the program kounter equal to the given constant and proceeds execution from there, provided that the given register does not contain 0.

Arguments: `[register], [constant]`.

5.1.5 goto_if_negative

Sets the program kounter equal to the given constant and proceeds execution from there, provided that the given register contains a negative value.

Arguments: `[register], [constant]`.

5.1.6 goto_if_nonnegative

Sets the program kounter equal to the given constant and proceeds execution from there, provided that the given register contains a non-negative value.

Arguments: `[register], [constant]`.

5.1.7 goto_if_positive

Sets the program kounter equal to the given constant and proceeds execution from there, provided that the given register contains a positive value.

Arguments: `[register], [constant]`.

5.1.8 goto_if_nonpositive

Sets the program kounter equal to the given constant and proceeds execution from there, provided that the given register contains a non-positive value.

Arguments: `[register], [constant]`.

5.1.9 goto_if_equal

Sets the program kounter equal to the given constant and proceeds execution from there, provided that the given registers contain the same value.

Arguments: `A : [register], B : [register|constant], [constant]`.

5.1.10 goto_if_unequal

Sets the program kounter equal to the given constant and proceeds execution from there, provided that the given registers do not contain the same value.

Arguments: `A : [register], B : [register|constant], [constant]`.

5.1.11 goto_if_less

Sets the program kounter equal to the given constant and proceeds execution from there, provided that $A < B$.

Arguments: `A : [register], B : [register|constant], [constant]`.

5.1.12 goto_if_greater

Sets the program kounter equal to the given constant and proceeds execution from there, provided that $A > B$.

Arguments: `A : [register], B : [register|constant], [constant]`.

5.1.13 goto_if_leq

Sets the program kounter equal to the given constant and proceeds execution from there, provided that $A \leq B$.

Arguments: `A : [register], B : [register|constant], [constant]`.

5.1.14 goto_if_geq

Sets the program kounter equal to the given constant and proceeds execution from there, provided that `A` $\geq$ `B`.
Arguments: `A : [register], B : [register|constant], [constant]`.

5.1.15 goto_if_less_unsgn

Sets the program kounter equal to the given constant and proceeds execution from there, provided that `A` $<$ `B`, considered as unsigned integers.
Arguments: `A : [register], B : [register|constant], [constant]`.

5.1.16 goto_if_greater_unsgn

Sets the program kounter equal to the given constant and proceeds execution from there, provided that `A` $>$ `B`, considered as unsigned integers.
Arguments: `A : [register], B : [register|constant], [constant]`.

5.1.17 goto_if_leq_unsgn

Sets the program kounter equal to the given constant and proceeds execution from there, provided that `A` $\leq$ `B`, considered as unsigned integers.
Arguments: `A : [register], B : [register|constant], [constant]`.

5.1.18 goto_if_geq_unsgn

Sets the program kounter equal to the given constant and proceeds execution from there, provided that `A` $\geq$ `B`, considered as unsigned integers.
Arguments: `A : [register], B : [register|constant], [constant]`.

5.2 Setting

5.2.1 swap

Swaps the values of `A` and `B`.
Arguments: `A : [register], B : [register]`.

5.2.2 set

The universal setting instruction. Sets `A` equal to `B`. This actually refers to 48 distinct instructions, each with their own microcode. The assembler uses the types of the given arguments to determine the correct opcode, so that, whenever the programmer wishes to save the value of something (a register, word of memory, etc) somewhere else, they may simply use `set`, without having to think about what actual instruction will be executed. The only invalid argument combinations are those with a (non-pointer) constant in the first argument, as, though a perfectly capable CPU (sans, e.g., privilege levels), DUNK still cannot change the value of fixed mathematical constants.

Arguments: `A : [*constant|register|*register], B : [constant|*constant|register|*register]`.

5.3 Arithmetic and Logic Operations

Arithmetic and logic instructions only operate on registers, although some accept constants in the second or third argument. Additionally, they have variable numbers of arguments. For example, instructions corresponding to binary operations such as addition can take two or three arguments, i.e., `add r0 r1` and `add r0 r0 r1` both add the values of `r0` and `r1` and save the result in `r0`. Note that a general rule for dunkasm argument syntax is that, if some register or memory address is mutated by the instruction, it is listed *first* in the argument list.

5.3.1 add

Adds the values of `A` and `B` and saves the result in `A` if given two arguments, or `D` if given three.
Arguments: `A : [register], B : [register|constant]`, or
Arguments: `D : [register], A : [register], B : [register|constant]`.

5.3.2 sub

Subtracts the value of `B` from the value of `A` and saves the result in `A` if given two arguments, or `D` if given three.
Arguments: `A : [register], B : [register|constant]`, or
Arguments: `D : [register], A : [register], B : [register|constant]`.

5.3.3 mul

Multiplies the values of `A` and `B` and saves the result in `A` if given two arguments, or `D` if given three.
Arguments: `A : [register], B : [register|constant]`, or
Arguments: `D : [register], A : [register], B : [register|constant]`.

5.3.4 div

Divides the value of `A` by `B` and saves the result in `A` if given two arguments, or `D` if given three.
Arguments: `A : [register], B : [register|constant]`, or
Arguments: `D : [register], A : [register], B : [register|constant]`.

5.3.5 rem

Computes the remainder of `A` on division by `B` and saves the result in `A` if given two arguments, or `D` if given three.
Arguments: `A : [register], B : [register|constant]`, or
Arguments: `D : [register], A : [register], B : [register|constant]`.

5.3.6 inc

Adds 1 to the value of `A` and saves the result in `A` if given one arguments, or `D` if given two.
Arguments: `A : [register]`, or
Arguments: `D : [register], A : [register]`.

5.3.7 dec

Subtracts 1 from the value of `A` and saves the result in `A` if given one arguments, or `D` if given two.
Arguments: `A : [register]`, or
Arguments: `D : [register], A : [register]`.

5.3.8 not

Performs bitwise `not` on `A` and saves the result in `A` if given one arguments, or `D` if given two.
Arguments: `A : [register]`, or
Arguments: `D : [register], A : [register]`.

5.3.9 and

Computes the bitwise `and` of `A` and `B` and saves the result in `A` if given two arguments, or `D` if given three.
Arguments: `A : [register], B : [register|constant]`, or
Arguments: `D : [register], A : [register], B : [register|constant]`.

5.3.10 or

Computes the bitwise `or` of `A` and `B` and saves the result in `A` if given two arguments, or `D` if given three.
Arguments: `A : [register], B : [register|constant]`, or
Arguments: `D : [register], A : [register], B : [register|constant]`.

5.3.11 xor

Computes the bitwise `xor` of `A` and `B` and saves the result in `A` if given two arguments, or `D` if given three.
Arguments: `A : [register], B : [register|constant]`, or
Arguments: `D : [register], A : [register], B : [register|constant]`.

5.3.12 cmp

Compares the values of `A` and `B` and saves the result in `A` if given two arguments, or `D` if given three. If `A` < `B`, the result is `-1`. If `A` = `B`, the result is `0`. If `A` > `B`, the result is `1`.

Arguments: `A : [register], B : [register|constant]`, or
Arguments: `D : [register], A : [register], B : [register|constant]`.

5.3.13 ucmp

Compares the values of `A` and `B`, considered as *unsigned* integers, and saves the result in `A` if given two arguments, or `D` if given three. If `A` < `B`, the result is `-1`. If `A` = `B`, the result is `0`. If `A` > `B`, the result is `1`.

Arguments: `A : [register], B : [register|constant]`, or
Arguments: `D : [register], A : [register], B : [register|constant]`.

5.3.14 lshift

Performs a bitwise left shift (by 1 bit) on `A` and saves the result in `A` if given one arguments, or `D` if given two.

Arguments: `A : [register]`, or
Arguments: `D : [register], A : [register]`.

5.3.15 rshift

Performs a bitwise right shift (by 1 bit) on `A` and saves the result in `A` if given one arguments, or `D` if given two.

Arguments: `A : [register]`, or
Arguments: `D : [register], A : [register]`.

5.3.16 shift

Shifts `A` leftward by `B` bits if `B` is positive, and rightward by `-B` bits if `B` is negative, and saves the result in `A` if given two arguments, or `D` if given three.

Arguments: `A : [register], B : [register|constant]`, or
Arguments: `D : [register], A : [register], B : [register|constant]`.

5.4 The Stack

5.4.1 push

Decrements `sp` and, if given, sets `*sp` equal to `A` - in that order, so that, after execution, `*sp` contains the value of `A`.

Arguments: `none`, or
Arguments: `A : [constant|*constant|register|*register]`.

5.4.2 pop

Increments `sp` and, if given, sets `A` equal to `*sp` - in reverse order, so that, after execution, `A` contains the value of `*(sp-1)`.

Arguments: `none`, or
Arguments: `A : [*constant|register|*register]`.

5.4.3 call

Performs a function call. To do this, dunk pushes the address of the first instruction following the function call to the stack, then performs "stacking"; pushes the values of any of registers `r[0-7]` which have been written to since the last function call onto the stack, followed by a word, the "stacking mask", indicating which registers were written to the stack. Finally, it sets `pk` to `A`.

Arguments: `A : [constant]`.

5.4.4 fastcall

Performs a "fast function call", which is the same as a regular function call, except stacking is skipped.

Arguments: `A : [constant]`.

5.4.5 return

Performs a function return. To do this, dunk first performs "unstacking"; reads the stacking mask from the stack and restores the values of any registers preserved on the stack. Following this, it pops the saved return address off the stack into `pk`.

Arguments: `none`.

5.5 I/O

5.5.1 pinmode_input

Sets the I/O pin given by the argument to input mode.

Arguments: `[constant`register]—

5.5.2 pinmode_output

Sets the I/O pin given by the argument to output mode.

Arguments: `[constant`register]—

5.5.3 set_pin_low

Sets the value output by the indicated pin to 0.

Arguments: `[constant`register]—

5.5.4 set_pin_high

Sets the value output by the indicated pin to 1.

Arguments: `[constant`register]—

5.5.5 read_pin

Reads the value of the indicated pin into the indicated register, extended by 0 to a 16-bit word.

Arguments: `P : [constant`register], D : [register]—

NOTE: pins can only be read into general-purpose registers. For instance, the line `read_pin 2 sr3` will result into an error.

5.5.6 write_pin

Sets the output value of the indicated pin to `0` if `A` $=$ `0`, and `1` if `A` $\neq$ `0`.

Arguments: `P : [constant`register], A : [register]—

5.5.7 prints

Prints the given string to the terminal.

Arguments: `S : [constant`*register], A : [*constant]—

Note: due to a quirk, the string address, if given as a constant, is not given as a pointer. Since string literals compile down to constants, one can write string literals in place. If one tries to pass the address of a string as a pointer, this will result in an error. Additionally, this instruction requires the MMIO address where the printing unit is connected. Unless the `.circ` files has been modified, this is the value given by the alias `PRINT_ADDR`. An example of valid code is given below.

```
prints "Hello, " PRINT_ADDR
set r0 "world!"
prints *r0 PRINT_ADDR
```

5.5.8 syscall

Calls the system call function, stored in the formware ROM, according to the given argument.

Arguments: `[constant]`

There are built-in alises for system call codes, given below.

- `PRINT_DIGIT`
- `PRINT_INTEGER`

- `PRINT_INTEGER_DECIMAL`
- `PRINT_INTEGER_HEX`
- `PRINT_INTEGER_BINARY`

For example, the following dunkasm code will print "`10`" to the terminal.

```
set argument 10
syscall PRINT_INTEGER_DECIMAL
```

5.6 Other

Finally, there is one "inaccessible" instruction: this is used only internally by the control unit, and isn't accepted by the assembler. It is injected whenever an interrupt is detected.

5.6.1 handle_interrupt

Calls the appropriate interrupt handler, whoose address is stored in the `interrupt_handler` registers, `sr[8-f]`, addording to which interrupt pin has recieved a signal.