K. Szewczyk

a.k.a. Palaiologos, The Greek Empress

asm2bf, the only true brainfuck assembly toolkit

THE JOURNAL OF A LONE DEVELOPER

2017 - 2020

Abstract

This manual will describe basic components of asm2bf, the history of asm2bf development, feature timeline, example code and complete instruction reference in a comprehensible manner. The author strives to provide as much of useful information as possible, although because of there are no reviewers available, the manual may be incomplete and additional sections may be added when needed. I'd like to thank everyone, who decided to aid me on this journey. Although residual, this support really matters to me.

Contents

Preface

asm2bf development began way in 2016. v0.9 of asm2bf has been published in October of 2017, therefore 2017 is considered the year when asm2bf was born. The initial version has severely changed over the years. K. Szewczyk gives the following incunabulum[1]:

```
#define G goto
#define z case
#define I if(
int m[2000];void a(){while(m[3]--)putchar(m[6]);}void b(){int c,d;m[7]=131;d=0;o1:I d>=m[6])G o2;
o3:c=m[m[7]];m[7]++;I c)G o3;d++;G o1;o2:c=m[m[7]];I!c)return;I c!=49)G o5;c=m[5];o5:I c!=50)G o6
;c=m[4];o6:I c!='*')G o7;c=m[9];G o11;o7:I c!='^')G o8;c=m[10];G o11;o8:I c<'a')G o9;I c>'z')G o9
;c-='a';o11:I c<m[8])G o12;d^=d;m[15]=c-m[8];o14:I m[15]<=d)G o13;putchar('>');d++;G o14;o12:d=0;
m[15]=m[8]-c;o16:I m[15]<=d)G o13;putchar(60);d++;G o16;o13:m[8]=c;G o10;o9:putchar(c);o10:m[7]++
;G o2;}q(){int c=getchar();return c<=0?0:c;}main(){int n;char*s="addanddecdiveq_ge_gt_in_incjmpj"
"nzjz_lblle_lt_modmovmulne_negnotor_outpoppshrclstosubswpclrretendstkorgdb_txtrawa+b+[\0b]\0a[c+"
"d+a-]c[a+c-]d[[-]\0""d]\0""2\0""2[-]\0""2[1+e+2-]e[2+e-]\0""1[e+1-]e[e[-]2[e+d+2-]d[2+d-]e[1-e["
"-]]]\0""2-\0""1[c+1-]c[2[d+e+2-]e[2+e-]d[e+c-[e[-]k+c-]k[c+k-]e[d-[1-d[-]]+e-]d-]1+c]\0""1[d+1-"
"]+2[d-c+2-]c[2+c-]d[1-d[-]]\0""1[d+1-]+2[c+k+e+2-]e[2+e-]k[d[l+e+d-]e[d+e-]+l[c-d-e-l[-]]e[k[-]"
"+e-]k-]c[1-c[-]]d[-]\0""1[d+1-]2[c+k+e+2-]e[2+e-]k[d[l+e+d-]e[d+e-]+l[c-d-e-l[-]]e[k[-]+e-]k-]d"
"[1+d[-]]c[-]\0""2,\0""2+\0a[-]b[-]2[b+c+2-]c[2+c-]\0""1[c+d+1-]c[1+c-]d[a[-]b[-]2[b+c+2-]c[2+c-"
"]d[-]]\0d+1[d[-]c+1-]c[1+c-]d[a[-]b[-]2[b+c+2-]c[2+c-]d[-]]\0c+a[c-d+a-]d[a+d-]c[-d+b[e-c+b-]c["
"b+c-]e[d-e[-]]d[a+d-]c]e[-]\0""1[d+1-]+2[c+k+e+2-]e[2+e-]k[d[l+e+d-]e[d+e-]+l[c-d-e-l[-]]e[k[-]"
"+e-]k-]d[1-d[-]]c[-]\0""1[d+1-]2[c+k+e+2-]e[2+e-]k[d[l+e+d-]e[d+e-]+l[c-d-e-l[-]]e[k[-]+e-]k-]c"
"[1+c[-]]d[-]\0""2[n+2-]1[m+>-[<<]<[[>+<-]<<]>>1-]m[1+2+m-]n[2+n-]\0""1[-]2[1+e+2-]e[2+e-]\0""1["
"d+1-]d[2[1+e+2-]e[2+e-]d-]\0""1[d+1-]2[d-e+2-]e[2+e-]d[1+d[-]]\0""2[e-2-]e[2+e-]\0""2-[e-2-]e[2"
"+e-]\0""1[e+1-]e[1-e[-]]2[e+d+2-]d[2+d-]e[1[-]-e[-]]\0""2.\0""2[-]q[-]>[>>]<<->[<<<[<<]>+>[>>]>"
"-]<<<[<<]>[2+q-]\0""2[e+q+2-]e[2+e-]q>[>>]+<<[<<]>[>[>>]<+<[<<]>-]\0""1[-]2[e+o+*>+<2-]e[2+e-]*"
">[[>>]+[<<]>>-]+[>>]<[<[<<]>+1+*>[>>]<-]<[<<]>[>[>>]<+<[<<]>-]>[>>]<<[-<<]>\0""1[e+*>+<1-]e[1+e"
"-]2[e+*+2-]e[2+e-]*>[[>>]+[<<]>>-]+[>>]<[-]<[<<]>[>[>>]<+<[<<]>-]>[>>]<<[-<<]>\0""2[1-e+2-]e[2+"
"e-]\0""1[e+1-]2[1+2-]e[2+e-]\0""2[-]\0a[-]b[-]q[-]>[>>]<<->[<<<[<<]>+>[>>]>-]<<<[<<]>[b+q-]\0a["
"-]b[-]\0";for(n=0;n<1900; n++)m[n+20]=s[n];m[9]=18;m[10]=20;b();m[11]=m[1]=1;A:m[0]=q();B:I 0[m]
)G C;I m[1]==2)G D;G E;C:I m[0]!='\n'&&m[0]!='\r')G F;I m[1]==2)G D;m[1]=1;G A;F:I m[1]==4){I m[0
]=='\"'){m[1]=1;G A;}m[2]=34;m[3]=m[0];G D;}I m[0]==32||m[0]==8||m[1]==0)G A;I m[1]!=3)G H;I m[0]
!='\"')G J;m[1]=4;G A;H:I m[0]!=';')G K;I m[1]==2)G D;m[1]=0;G A;K:I m[1]!=1)G L;m[2]=q();m[3]=q(
);m[4]=0;M:m[5]=m[4]+20;m[6]=m[m[5]];I m[0]!=m[6])G N;m[5]++;m[6]=m[m[5]];I m[2]!=m[6])G N;m[5]++
;m[6]=m[m[5]];I m[3]==m[6])G O;N:m[4]+=3;I m[4]==111)G J;G M;O:m[1]=2;m[2]=m[4]/3;m[3]=0;m[4]=0;m
[5]=0;G A;L:I m[2]!=35)G P;m[1]=3;G B;P:I m[0]!='r')G R;m[0]=q();m[0]-='1';I m[0]>3)G J;m[4]=m[0]
+'f';G A;R:I m[0]!=',')G S;m[5]=m[4];m[4]=0;G A;S:I m[0]!='.')G T;m[3]=q();G A;T:m[0]-='0';I m[0]
>9)G J;m[3]*=10;m[3]+=m[0];G A;D:I m[4]&&m[4]==m[5]){m[6]=22;m[5]='j';b();m[5]=m[4];m[4]='j';}I m
[11]==1&&m[2]!=12){m[6]=2;b();m[11]=0;}switch(m[2]){z 0:I!m[4]){m[6]=4;m[4]=m[5];b();m[6]='+';a()
;G U;}G W;z 9 ...11:z 30:z 31:m[11]=1;m[12]=1;G W;z 12:I!m[11]){m[6]=3;b();}m[11]=1;m[4]='e';m[6]
=4;b();m[6]='+';a();m[6]=18;b();G U;z 16:I!m[4]){m[6]=5;m[4]=m[5];b();m[6]=43;a();G U;}G W;z 27:I
!m[4]){m[6]=4;m[4]=m[5];b();m[6]='-';a();G U;}G W;z 32:m[9]=m[3]*2+18;G U;z 33:m[10]=m[3]*2+m[9]+
2;G U;z 34:m[6]=4;m[4]='^';b();m[6]='+';a();m[10]+=2;I m[1]==4)G A;G U;z 36:putchar(m[3]);G U;}W:
I!m[4]){m[6]=4;m[4]='j';b();m[6]='+';a();m[3]++;}m[6]=m[2]+6;b();I!m[3])G U;m[6]=5;b();U:m[1]=1;I
m[12]==1){m[6]=3;b();m[12]=0;}G B;J:return 1;E:I m[11]==0||m[12]==1){m[6]=3;b();}m[6]=2;b();m[6]=
37;b();m[6]=3;b();m[6]=1;b();}
```

This compiler's dialect differs from the current-day asm2bf principles. There are no error messages, and the error presence is signified by the return value. All instructions work (unlike the later version of v0.9, which has had one instruction broken). A lot of concepts are still present (for example, the current day asm2bf provides just `seg` instruction for changing segment; `txt` and `db` commands have been around since the humble origins of

[1] A single page compiler for v0.9 dialect

asm2bf; also, `bfvm` allows so-called real segmentation mode, which will be discussed later). One significant change is the ecosystem around the core. The first toolkit parts included the `bfasm`, responsible for assembling simple mnemonics to brainfuck code. At first, the instruction set was very constraining (for example, one couldn't define an instruction of length other than 3 due to the limitations of the engine). The first significant change appeared in `v0.9.4`, the first version to have bundled an external program for preprocessing the asm2bf code.

Chapter 1

The Toolkit

asm2bf is a set of tools for assembling to brainfuck. These tools include the `bfi`, `bfasm` (the core), `bflabels`, and so on. In this very first chapter, I'd like to shed some light on the internals of asm2bf, including the expected specification.

1.1. Initial considerations

Before we begin, I'd like to introduce a few definitions and concepts I'll use in this part of the manual.

- RLE encoding / compression - a technique of packing (in this case) brainfuck code by fusing the operations together. Strictly speaking, for each n-element vector of the instruction t, it can be replaced with a `[n, t]` (the prefix notation) or a `[t, n]` (the postfix notation) vector. For example, `>>>>` gets translated to `4>` or `>4`.

1.2. bconv

bconv is a tool supplied with asm2bf since `v1.1.0`[1]. It's primary goal is providing a layer of compatibility between 8-bit, 16-bit and 32-bit brainfuck interpreters. Because asm2bf model (for the convience of the programmer) assumes the registers to be at least 16 bits big, the requirement can't be satisfied on, for example, an 8-bit interpreter with the code left as-is. 8-bit interpreters open way more problems for asm2bf, the most significant one being, inability to index memory cells over 255 (note: while this could be theoretically solvable using the segment feature from `v1.2.9`, this approach will emit a lot of pointer movements[2]). Another problem is the label indexing - a program can't have more than 256 code labels, therefore the programs are slightly constrained and sometimes, the control flow needs to be synthetized using native brainfuck loops[3].

bconv will accept a brainfuck file on it's standard input, and output a tweaked version of it (it's smart to pass it through **bfstrip** later), so that the code will run on a 8-bit interpreter, while retaining 16-bit registers. One strength of the approach used is the universality, id est, the code will always double the bitwidth, therefore on 32-bit interpreters, the registers will be 64-bit big. **bconv** will use premade brainfuck snippets to replace each brainfuck instruction with it's doubled representation.

```
> >>>>                    >>>
< <<<<                    <<<
+ >+<+[>-]>[->>+<]<<      +>+[<->[->>+<<]]>>[-<<+>>]<<<[->>+<<]
- >+<[>-]>[->>-<]<<-      +>[<->[->>+<<]]>>[-<<+>>]<<<[->>-<<]>-<
[ >+<[>-]>[->+>[<-]<      >[<+>[->>+<<]]>>[-<<+>>]<[<<
  [<]>[-<+>]]<-[+<        +>>[->+<]]>[-<+>]<<<[[-]
] >+<[>-]>[->+>[<-]<      >[<+>[->>+<<]]>>[-<<+>>]<[<<
  [<]>[-<+>]]<-]<         +>>[->+<]]>[-<+>]<<<]
```

bconv provides two sets of snippets. The first one will consume four memory cells for each logical cells, and the second one will consume only three, but the first one will be faster, because it does no copying. Also, it's shorter. They are togglable during the compile-time [4]:

[1] https://github.com/kspalaiologos/asmbf/commits/master/bconv.c

[2] >

[3] raw .[and raw .]

[4] The same technique can be used to apply other compile-time options to asm2bf toolkit

```
# Will force the use of 1st conversion schema.
# The second one is enabled by default.
setenv OPTIONS "-DDOUBLE_NOCOPY"
make all setup
```

bconv will copy all the non-brainfuck instructions over to the resulting program, this way hopefully preserving breakpoints and such. That being said, **bconv** will work only on the standard subset of brainfuck, and will produce invalid code when fed with RLE-encoded brainfuck. Also, the output is not guaranteed to be correct for values larger than 256 or the bounds of the original bitwidth. In such case, the output may include only the lower part of the number.

Finally, it's not recommended to use **bconv** for larger programs, because they tend to become very slow (the measurements show, around five times slower).

1.3. bfi

bfi is a simple brainfuck interpreter bundled with asm2bf. It's fully compiliant to the asm2bf specification, has a dynamic-length tape, and allows basic debugging capabilities. It will error if the Brainfuck loops are unbalanced and in case of a tape underflow (in most of the cases, due to a stack underflow, dubious operand+opcode combination, or an incorrect memory access). It's moderately fast and it can match clear loops (`[-]`). `asm2bf` can work well with Tritium interpreter[5] too.

bfi will allocate memory for the tape in 1024-cell blocks. **bfi** can be compiled with the `-DBFI_NOCHECKS` flag, which disables the runtime checks (like underflows) for a bit better performance. The interpreter sets the cell value on `EOF` to `0` for the convience of the programmer.

bfi supports a set of basic runtime flags:

- `-d`: display the output data as numbers, opposed to the default ASCII output format.
- `-x`: enable memory dumps on `*`.
- `-c`: count cycles and display the performance statistics after a successful program execution.

bfi also comes with **bfi-rle**, which will unpack RLE-compressed brainfuck code using **bfderle** and then execute it using **bfi**.

1.4. bfderle

bfderle is a tool to unpack RLE-compressed brainfuck programs, it takes a single, optional argument on it's commandline and will uncompress programs in the UNIX fashion (take input from stdin; write output to stdout). Two RLE variants are supported - the prefix one and the postfix one (**bfderle** takes them respectively as `prefix` and `postfix` commandline parameters).

1.5. bfstrip

bfstrip is a brainfuck stripper. It's goal is to remove unnecessary operations (like `>><<`) and optimize the code (`+++.+` becomes `+++.` and `+++--` becomes `+`). **bfstrip** doesn't optimize the RLE-compressed code and the **bfvm** bytecode.

1.6. report generation

generate_report is a tool bundled with asm2bf. It's a simple bash script that will do the following:

- rebuild asm2bf and run the unit tests, storing the operation log.

[5] https://github.com/rdebath/Brainfuck/tree/master/tritium

- `tar` all the compiled testcases and their actual outputs.
- generate the core `diff` (optional).
- `tar` all of these and build a fresh tar archive to include along an issue report on the tracker.

When reporting bugs, the author is expected to submit the test case, it's input and the expected output as an unit test.

1.7. bfpp

bfpp is a Lua-based preprocessor bundled along asm2bf. To introduce a single line Lua snippet, prepend the line with `#`[6]. Longer blocks can be introduced using `$( / code / )` syntax. The block macros will also emplace the value, unlike the `#` counterpart. For example, `mov r1, $(6 * 6)` will compile to `mov r1, 36`.

bfpp introduces *no* builtin macros. All the standard definitions are declared by **lib-bfm**, mentioned below.

1.8. bfmake

bfmake will build an asm2bf program to a brainfuck or C source file [7]. The compile flags along with optional positional arguments for them go *always before the source file*. Example usage: `~/.asmbf/bfmake file.asm`[8], that will build `file.b`.

bfmake will output C code if asm2bf has been built with `-DBFVM` option and the correct *runtime* flag has been specified. If just one of these conditions is satisfied, the outcome is undefined.

- `-c`: build C code (assumes asm2bf is compiled with `-DBFVM`).
- `-l`: disable linking along the standard library (`lib-bfm.lua`).
- `-p`: preprocess-only; don't build BFVM bytecode or brainfuck source. You may want to use `-s` alongside this flag.
- `-s`: disable the `bfstrip` pass.
- `-o`: override the default output file. `-c` flag sets the default filename to `file.c` (assuming the compiled unit is named `file.asm`), otherwise it's assumed as `file.b`.

1.9. lib-bfm

lib-bfm is the core file of the asm2bf standard library. It defines a few utility aliases to common instructions (for example, an alias `push => psh`) and a few utility functions.

1.10. vxcall

vxcall is a virtual instruction preprocessor. Most instructions in asm2bf take operands as `reg, imm`, `reg, reg`, `imm` or `reg`. **vxcall** is a tool made to allow programmers use the `imm, reg` or even `imm, imm` combination where feasible. It's worth noting, that such instruction doesn't exist and most probably will never exists, therefore the name *virtual*. **vxcall** instruction modifier can be used only alongside `sto, amp, smp, cst, cam, csm, cot, ots, spt` and `cots, csmp, camp, csto`[9]. For example:

[6] The hash symbol is expected to go **at the beginning** of the line

[7] using bfvm

[8] asm2bf is installed by default in the home directory unless the user decided to install it manually, i.e. not using *setup* make target.

[9] assuming lib-bfm is included

band	bor	bxor	bneg
cflip	xor	push	xchg
cots	movf	lea	cgcd
cret	finv	fmul	fdiv
freduce	fadd	fsub	cadd
csub	cmul	cdiv	cmod
casl	casr	cpow	cpush
cpsh	cpop	cxchg	cswp
csrv	cmov	crcl	csto
cout	cjnz	cpar	candeq
candne	candle	candge	candlt
candgt	coreq	corne	corle
corge	corlt	corgt	cxoreq
cxorne	cxorle	cxorlt	cxorgt

Table 1.1: Aliases defined by **lib-bfm**

include	call	alloc	free
times	MM_BASE	PAGE_SIZE	

Table 1.2: Macros defined by **lib-bfm**

```
vxcall sto 3, .0
; roughly equivalent to
mov r1, 3
sto r1, .0
; note the vxcall version doesn't trash r1!
```

1.11. constpp

constpp is a constant preprocessor bundled along asm2bf. It can be used to create aliases for instructions[10]. For example:

```
?bp=r1
?sp=r2
mov bp, sp
; after preprocessing, equivalent to
mov r1, r2
```

The definitions and their replacements are assumed to start with either a letter or a floor, followed by any number of letters, floors or digits.

1.12. bflabels

asm2bf is internally using a system of numbered labels. **bflabels** is a tool to translate code utilizing named labels to such code, which utilizes numeric labels instead. A label reference is introduced using the percent symbol, for instance - `%name`. A label is defined using the at symbol, for example - `@name`. The label definition can be the *only* thing on a given line. As an example, let's look at this code:

```
jmp %skip
@infinite
    jmp %infinite
```

[10] aliases aren't expanded inside string constants. It's advised to avoid creating single letter constant definitions

```
@skip

; compiles to the equivalent of...

jmp 1
lbl 2
    jmp 2
lbl 1
```

It's *not* recommended to use the `lbl` instruction inside the code, as it may collide with another label defined by the programmer and introduce all sorts of nasty crashes if used carelessly enough. It's also not recommended to start the label names with an underscore (mainly because the asm2bf toolkit may use this namespace internally), although it's not forbidden.

1.13. bfdata

bflabels preprocessor works only for code. As asm2bf employs the Harvard architecure[11], code labels have no relation to data labels.

A data label is introduced using the ampersand character, for example `&region`. It can be referenced using an asterisk, for example `*region`. **bfdata** keeps track of the segmentation and the current origin. Let's look at this illustrative example:

```
&variable
db 0

&string
txt "Hi!"
db 0

&temp
db 5

; *variable => 0
; *string => 1
; *temp => 5
```

Segmentation handling is a bit more tricky.

```
db .1
seg 5
&label
db .0
seg 0
; *label points now to .1 (ASCII 49), because the track of segmentation
; is disabled by deafult.
```

This behaviour can be overriden by compiling the asm2bf toolkit with `-DACCOUNT_SEGMENTS`, so that `*label` points at `.0` (ASCII 48), but this enforces linear adressing mode relative to `seg 0`. This can be changed too (so that asm2bf respects segmentation in all cases) with `-DRELATIVE_SEGMENTATION` alongside the setting mentioned above. This brings the following concerns though:

```
seg 0
&variable
```

[11] Code and data are separated

```
txt "Hi!"

seg 10
; *variable now will be negative => an invalid adress has emerged.
; Adressing *variable now will result in a compilation error.
```

Segmentation is often used to either:

- Split the memory regions across the code modules. This approach is a bit sloppy, because it may require Lua code to ensure dense packing of variables in memory, but it's a perfectly fine way of structuring your application.
- Overcome adressing limits. Segmentation will allow asm2bf programs to adress (theoretically) infinite amounts of memory, even on 8-bit interpreters. In this case, linear adressing defeats the purpose of even using segmentation in this case (because an overflow will happen *eventually*, while attempting to read or write a memory region).

1.14. effective

effective is a quite complex preprocessor used to allow the usage of effective adresses inside asm2bf code. All the features are supported, including stack-based effective adresses and the memory-based ones. The syntax follows:

- `mov r1, [sp - 2]` - Extract the second topmost element from the stack and put it in r1.
- `inc [sp - 2]'` - Increment the second topmost element on the stack. Note the single apostrophe. This construction is called a *primed effective address*; in this case, it's a stack-relative primed effective address. The *priming* for stack-based effective addresses first fetches the value (like normal stack-based effective addresses), but also flushes the operation result into the memory. For example, `inc [sp - 2]` may be considered a no-operation (because the value is fetched, then mutated, and then finally discarded).
- `lea r1, D(B,I,S)` - Load the computed effective address into r1. Note, this operation doesn't perform any dereferences. It's equivalent to `mov r1, D(B,I,S)`, as `lea` is linked to `mov`. The formula for calculating the effective adress value follows - $D + B + I * S$, where D is the displacement (expected to be a **numeric** immediate), B is the base (expected to be a **general-purpose** register), I is the index (expected to be a **general purpose register**) and S is the scale factor (expected to be a **numeric** immediate). Using character constants inside effective adresses is forbidden. The case for register prefix letter (`r`, like in `r1`) doesn't matter, but it has to be uniform (over the entire effective adress, the programmer should use either uppercased `R` or lowercased `r`), for example `*string(r5, r6, 4)`.
- `movf r1, D(B,I,S)` - Dereference the computed effective adress into r1. Equivalent to singly-primed memory-based effective adress; `mov r1, D(B,I,S)'`, `lea r1, D(B,I,S)'` or even `rcl r1, D(B,I,S)`.
- `inc D(B,I,S)''` - Dereference the computed effective adress, increment the value pointed by it and flush it back into the memory. As with the stack-based effective adresses, `inc D(B,I,S)'` or `inc D(B,I,S)` is considered a no-operation. In the given example, it'd be more efficient to use `amp D(B,I,S)', 1`, because it's a bit smaller (no need to poke back the value again).

1.15. bfvm

bfvm is a tool bundled with asm2bf used to compile asm2bf bytecode to C (mostly for testing and performance reasons). To build asm2bf bytecode, use the `-c` flag for **bfmake**, and compile the toolkit with the `-DBFVM` flag. Currently, **bfvm** supports only basic instruction set for asm2bf (many instructions simply aren't implemented yet).

bfvm can output usermode code (by default) and freestanding code (which will place the tape at 0x0000:0x7000; which you can enable with the `-DFREESTANDING` flag). **bfvm** also supports multiple bitwidthes, the 16 bit one being the one enabled by default, although the `-DBFVM_32` flag will enable the 32-bit mode for **bfvm**. The usermode target assumes 65536-cell big tape, although this setting can be changed with `-DBFVM_HEAP="(your size)"`.

1.16. bfasm

bfasm is the core compiler which processes simplified-down and preprocessed code, and outputs either asm2bf bytecode OR brainfuck code. By default, it optimizes stores (for example, 48 becomes `>++++++[<+++++++++>-]` instead of the naive, unrolled approach). This behaviour can be overriden with `-DDISABLE_OPT`. **bfasm** can also emit RLE-compressed code, when compiled with `-DRLE` flag. The style can be overriden (to output postfix-style brainfuck code) via `-DRLE_POSTFIX`.

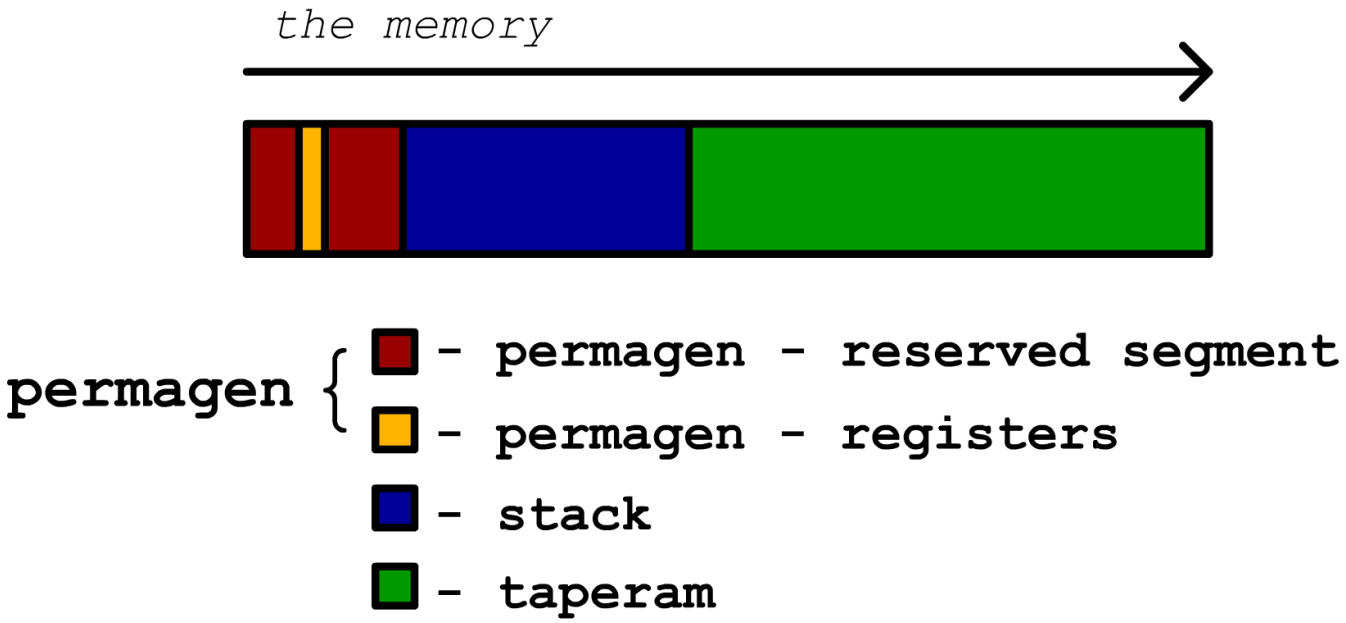
Chapter 2

The language

In this chapter I'd like to describe various aspects of programming in asm2bf.

2.1. Memory model

The memory is split into a couple of pieces:



Most operations in asm2bf affect the reserved segment of the permagen[1] one way or another.

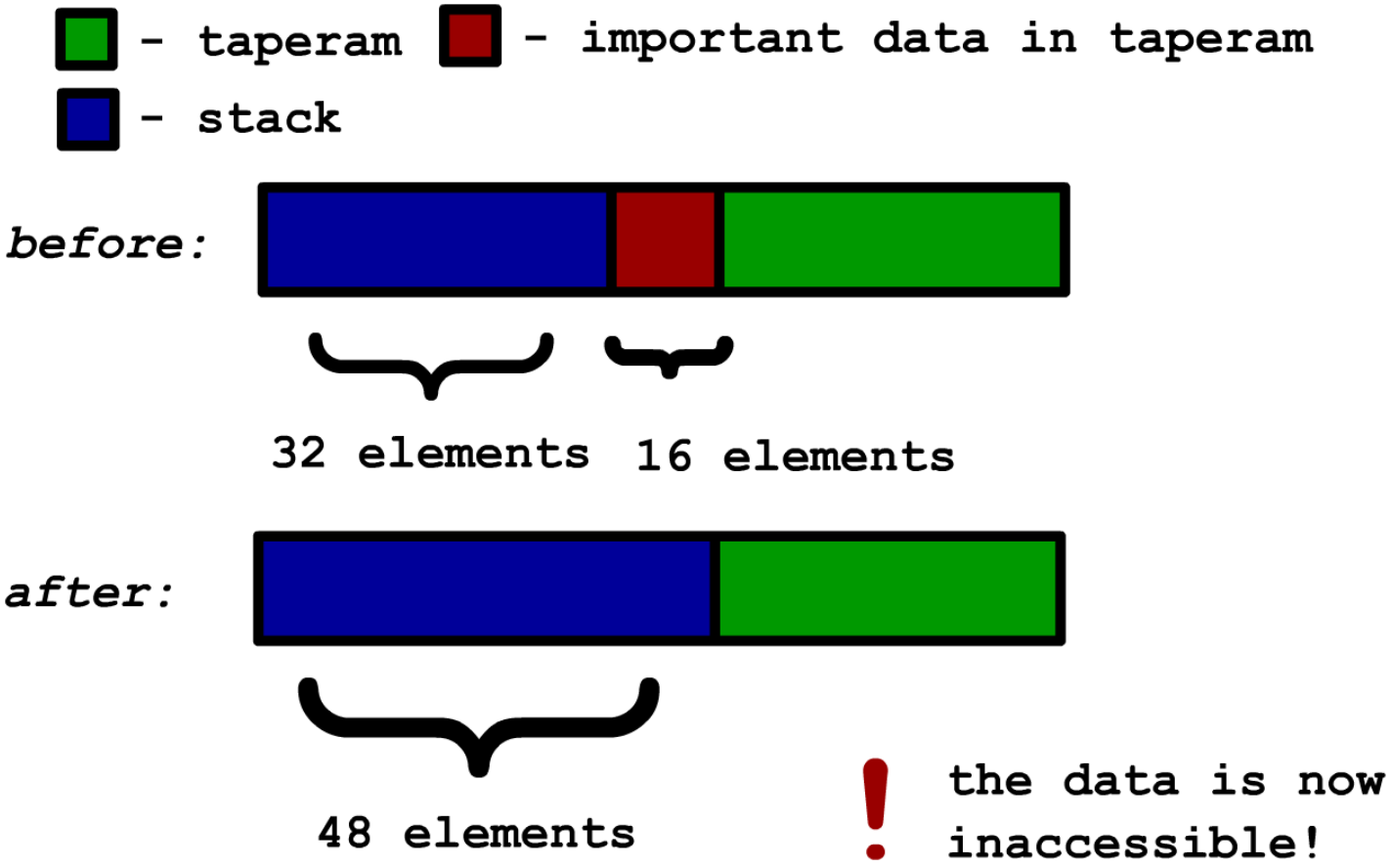
The permanent generation ends at around 20th cell. After it comes the stack (of definable, constant size) and the taperam (possibly unbounded, possibly bounded, possibly not available). The permanent generation also contains a memory region dedicated to registers. There are six general-purpose registers, from `r1` to `r6`, a flag register (`f1`, not recommended to actually use it for reasons other than potential preservation and recalling) and temporary, builtin registers (`f2+`, it's *really* not recommended to tweak with these; most operations will reject them and storing anything inside may end up with your data getting trashed by one operation or another).

The stack doesn't necessarily need to be set up. In some cases (like, the ones where the user program doesn't attempt to access the taperam) the `stk` and `org` declaration can be omitted. The same goes in similiar cases (i.e. where the stack isn't used). The programmer should pay close attention to stack bounds and resize it with care[2]. Careless operation with the stack may result in the following to happen:

The stack pointer in asm2bf is tacit. This means, asm2bf doesn't actually keep track of the stack pointer. It is in fact possible to take out select elements from it (refer to the stack-based effective adresses) and push desired

[1] the permanent generation - it will *always be there*, and it is guaranteed that you will have access to the entirety of it; meanwhile the stack or taperam may be restricted by the memory available to the interpreter

[2] the stack can be resized only at the compile time, and the existing taperam content won't move to the right



data onto the stack without actually keeping track of the tail. The stack length can be computed (not queried!) using the `sle` instruction.

The taperam operations often refer to the concepts of offsets and segments. For example, `txt` and `db` operations will move the offset so that it points the next logical cell everytime. Starting with offset 3, `txt "Hi!"` will move the offset to 6, so that the upcoming `db` or `txt` operation will start writing at offset 6. The offset is invalidated everytime the stack is reallocated (that means; you have to use `org` instruction straight after `stk` instruction if you intend to use the taperam). The segmentation on the other hand is a bit scarier topic.

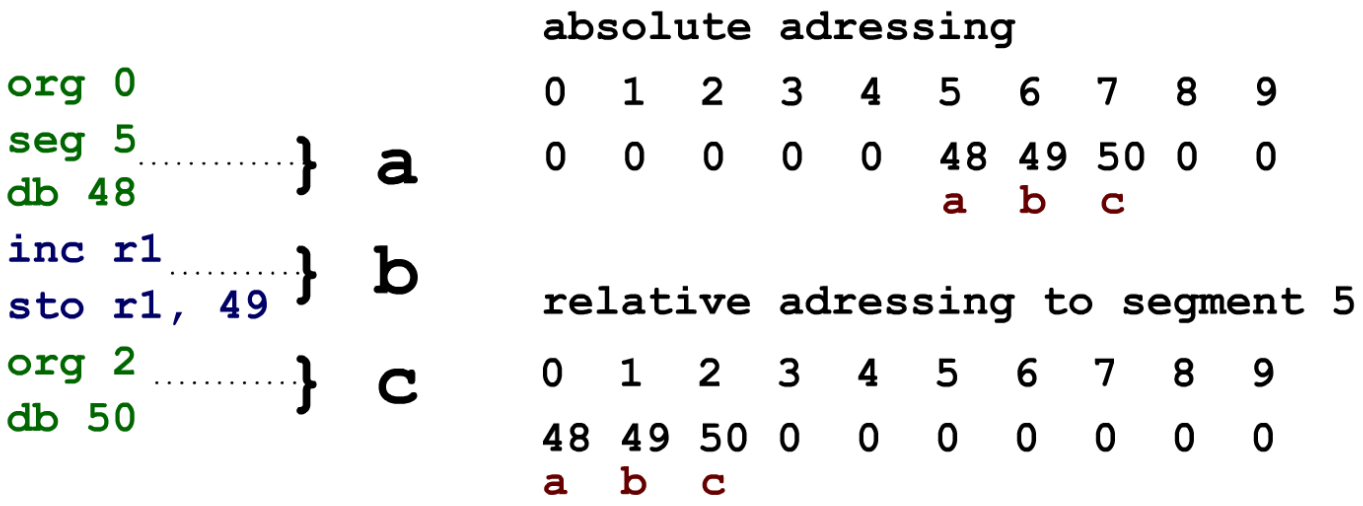
Simply put, `seg N` will reset and then *move* the logical taperam start pointer[3] right N cells. This has all sorts of implications.

- Segmentation allows to adress (theoretically) infinite amount of memory. After swapping the segment to, for example, 256, will allow linear adresses and stores to for example physical cell `300`, although the code refers to cell 44[4].
- Segmentation allows to modularize the code. For example, a module may claim it's own adress space via segmentation without needing to worry about pointer juggling.
- Segmentation disallows backreferences. Trying to adress `0:0` from `10:0` will result in a compilation error (because the offset is negative) when the toolkit is compiled with `-DRELATIVE_SEGMENTATION`. Otherwise, the segmentation is ignored when *referencing*[5] to cross-segment variables (a common source of bugs). `-DACCOUNT_SEGMENTS` will take segments into the account, so that label at `0:0` will not point the same place as `10:0`, *but* the offsets will be linear (relative to segment 0), not relative to the current segment, which may not be the desired behaviour (linear adresses, which implies the inability to execute point 1; but also inability to backreference labels, due to aforementioned negative segment problem arising).

[3] the place where the zero cell resides

[4] While there is *theoretically* no restriction on the segment, due to asm2bf core using 16-bit integers, be sure to not overflow it; if it eventually happens for any bizzare reason, feel free to talk to me asking to bring a definable 32-bit version of the compiler

[5] not declaring!



2.2. Syntax

asm2bf syntax is defined by a few rules:

Type	Syntatic rule	Example
String constant	A quoted string of characters, supports `\0`, `\n`, `\f`, `\r` escape sequences, the content is assumed to be fine as long as it belongs in the printable ASCII range (32-126, except the quote itself which will terminate the string). Usable only with the `txt` mnemonic.	`"Hello, world!"`
Char constant	A character prepended by a dot, usable in most contexts (except effective adresses)	`.K`
Numeric constant	A sequence of decimal digits.	`654`
Single line macro	Lua code prepended with a hash symbol at the beginning of the line.	`#include("A.asm")`
Pure mnemonic	A three or two character identifier recognized by **bfasm**.	`mov`
Virtual mnemonic	A mnemonic defined with **constpp**	`freduce`
Identifier	A sequence of alphanumeric characters, starting with a floor or a letter.	`mov`
Multi line macro	Lua code enclosed between `$(` and `)`. Emplaces the return value.	`$(2 + 2)`
Virtual call	Any instruction prepended with `vxcall`.	`vxcall sto 1, 2`
Immediate value	A character constant, a numeric constant or a string constant.	N/A
Register	A digit from 1 to 6, prepended with lowercase or uppercase `r`.	`r5`
Internal register	A digit from 1 to 4, prepended with lowercase or uppercase `f`.	`F1`
Const mnemonics	`?a=b`, where identifier `a` is the mnemonic to match, and identifier `b` is the replacement.	`?sp=r6`
Data references	An identifier preceded by a `*`.[6]	`*string`
Data anchors	An identifier preceded by a `&`.[7]	`&string`
Code references	An identifier preceded by a `%`.	`%label`
Code anchors	An identifier preceded by a `@`.	`@label`
Stack-based (or stack-relative) effective adress	An interfix expression enclosed in braces (`[]`), which provides an adress relative to the top of the stack[8] with a single, general-purpose register or numeric constant modifier. `sp` is expected to be lowercase, `rX` may be upcased if desired.	`[sp-1]`, `[sp-r5]`

[6] backreferences don't work with data labels
[7] backreferences don't work with data labels
[8] the stack grows upwards, not downwards

Singly primed stack-based effective adress	Will update the value pointed by the calculated effective adress after the current instruction finishes executing. Built by appending a single apostrophe to the stack-based relative adress.	`[sp-3]'`
Effective adress	Built as a three-argument function of a numeric, constant value - `D(B,I,S)`; calculated as `D + B + I * S`. A closer explanation can be found in the **effective** toolchain element chapter.	`*str(r5, r6, 2)`
Singly primed effective adress	Built by appending an apostrophe to the effective adress.	`*str(r5, r6, 2)'`
Doubly primed effective adress	Built by appending an apostrophe to the singly primed effective adress.	`*str(r5, r6, 2)''`
Conditional pipeline element / instruction	A conditional variant of an instruction which will perform the desired operation only if the condition flag is set, often built by prepending a `c` to the instruction. Not every pure instructions have their conditional variants, sometimes the conditional variant is a virtual instruction. Sometimes conditional instructions don't have their regular counterpart	`cadd`
Conditional pipeline primer	Usually starts the conditional pipeline; executes always and sets the conditional flag.	`ceq`, `cxoreq`