asm2bf, the only true brainfuck assembly toolkit

Kamila Szewczyk

2017 - 2021

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.

Contents

Chapter 1

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. Kamila 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 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

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

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 2

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.

2.1 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`.

2.2 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 convenience 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 emitted programs are slightly constrained and sometimes, the control flow needs to be synthesised 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 a single set of snippets (the first one) - it consumes four memory cells for each logical cellsm but it's much faster (as it doesn't utilize copying). Also, it's shorter.

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

[2] ¿

[3] raw .[and raw .]

2.3 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[4] too.

bfi will allocate memory for the tape in 1024-cell blocks. **bfi** can be compiled with the `--enable-nocheck-bfi` 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. Setting the `--enable-nocheck-bfi` flag can be accomplished via the `./configure` command:

```
# For the list of available switches, try ./configure --help.
./configure --with-target=release --enable-nocheck-bfi && sudo make all install
```

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**.

2.4 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).

2.5 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, thus processing such using bfstrip will yield unpredictable results.

2.6 bfpp

bfpp is a Lua-based preprocessor bundled along asm2bf. To introduce a single line Lua snippet, prepend the line with `#`[5]. Longer blocks can be introduced using `$( / code / )` syntax. The block macros will also emplace the returned value in the code, 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.

2.7 bfmake

bfmake will build an asm2bf program to a brainfuck or C source file [6]. The compile flags along with optional positional arguments for them go *always before the source file*. Example usage: `bfmake file.asm` - builds `file.b`.

- `-h`: display the brief usage (help) section.
- `-s`: disable the `bfstrip` pass.
- `-p`: preprocess-only; don't build BFVM bytecode or brainfuck source. You may want to use `-s` alongside this flag.
- `-l`: disable linking with the standard library (`lib-bfm.lua`).

[4] https://github.com/rdebath/Brainfuck/tree/master/tritium
[5] The hash symbol is expected to go **at the beginning** of the line
[6] using bfvm

- `-b`: build asm2bf bytecode; incompatible with `-p`.
- `-c`: build C code; incompatible with `-p`, a superset of `-b`.
- `-m`: print label mappings (which number / memory address has been picked for the given label).
- `-w`: set line width; wrap the output brainfuck code to length n - `-w n`.
- `-z`: disable size optimizations (emit constants in O(n) space complexity; incompatible with `-p`).
- `-e`: compress code with prefix RLE (incompatible with `-p, -f`).
- `-f`: compress code with postfix RLE (incompatible with `-p, -e`).
- `-t`: output tiny code; disable the branching system (= `end`, `ret`, all jumps and labels won't compile).
- `-x`: apply a bitwidth extender pass if building brainfuck (pass the resulting brainfuck code through bconv); if building bfvm bytecode or C code, set the bitwidth to 32 bits.
- `-f`: in bfvm mode, build freestanding code.
- `-a`: in bfvm mode, set the size of heap in cells. incompatible with `-f`.
- `-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`.

2.8 lib-bfm

lib-bfm is the asm2bf standard library. It defines a few utility aliases to common instructions (for example, an alias `push => psh`) and a few utility functions listed in the table below. Most of these have been defined using **constpp**. **lib-bfm** usage isn't necessary by any means, and can be disabled using the `-l` option for **bfmake**. **lib-bfm** is *not* included by default when using bfpp directly.

It's split across a couple of files in the `lib-bfm/` directory. `lib-bfm/lib-def.lua` contains basic constant definitions, `lib-bfm/lib-bfm.lua` contains basic macros, and `lib-bfm/lib-bfm-stub.lua` contains the stub code, which pulls the definitions into the current compilation unit.

include	call	alloc	free
times	MM_BASE	PAGE_SIZE	signed
gen_text			

Table 2.1: Macros defined by **lib-bfm**

2.9 constpp

constpp is a constant preprocessor bundled along asm2bf. It can be used to create aliases for instructions[7]. 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.

Most instructions in asm2bf take operands as `reg, imm`, `reg, reg`, `imm` or `reg`. **vxcall** is a prefix introduced by `constpp` 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`[8]. For example:

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

[8] assuming lib-bfm is included

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

vxcall is not recommended for most `imm, reg` encodings. It's better to use their reversed versions instead.

2.10 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
@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.

As asm2bf employs the Harvard architecure[9], 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`. **bflabels** 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 .0 (ASCII 49), because asm2bf automatically keeps track of segmentation.
```

asm2bf won't keep track of segmentation if you use *far pointers*. For instance:

[9] Code and data are separated

```
seg 0
&variable
txt "Hi!"

seg 10
; *variable now will be negative => an invalid adress has emerged.
; Adressing *variable now will result in a compilation error.
; There is a way though! You can load *variable's absolute address (but you can't use it):
mov r1, *far variable
```

Another example of segmentation in use:

```
seg 5
&variable db 3
seg 2
mov r1, *variable ; r1 = 3
mov r1, *far variable ; r1 = 5
```

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 addressing limits. Segmentation will allow asm2bf programs to adress (theoretically) infinite amounts of memory, even on 8-bit interpreters. In this case, linear addressing 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).

bfdata also allows you to set the target bitwidth (and override the default 16-bit one) using a special directive. To set the bitwidth to 32-bit.

2.11 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).

2.12 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**. 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 `-freestanding` flag). **bfvm** also supports multiple bitwidthes, the 16 bit one being the one enabled by default, although the `-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 `-heap size`.

Currently, **bfvm** doesn't support all opcodes. As of v1.5.3b, instructions appearing after `cre` are unsupported.

2.13 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 `-O0`. **bfasm** can also emit RLE-compressed code, when compiled with `-zb` flag. The style can be overriden (to output postfix-style brainfuck code) via `-ze`. bfasm can also emit bfvm bytecode, which is required to use `bfmake` with `-c` flag, using `-vm`. **bfasm** can also emit warnings and errors, which you can disable using the `-w` flag (although incorrect code won't work and may behave in strange ways). It's possible to disable the label system using the `-t` flag (outlined in the **bfmake**) section.

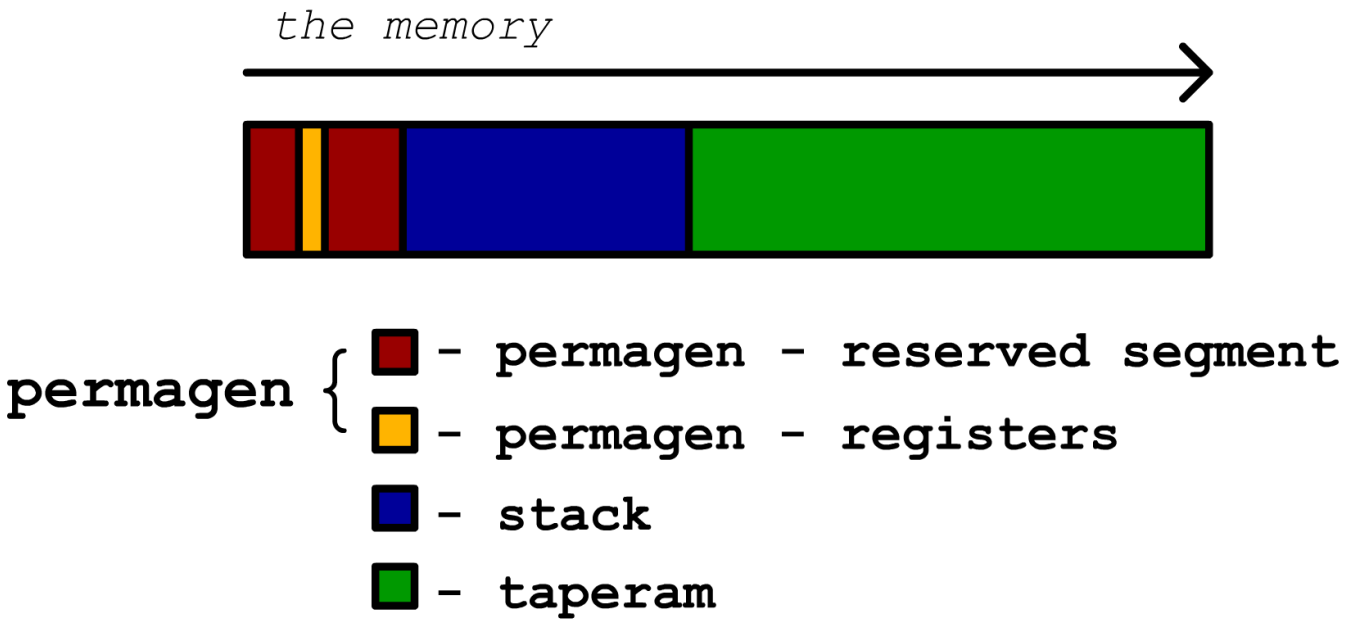
Chapter 3

The Language

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

3.1 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 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

[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

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 mentioned before in the **bfdata** part.

3.2 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 `*`.[4]	`*string`
Data anchors	An identifier preceded by a `&`.[5]	`&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[6] 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]`
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`

[3] the place where the zero cell resides
[4] backreferences don't work with data labels
[5] backreferences don't work with data labels
[6] the stack grows upwards, not downwards

Conditional pipeline primer	Usually starts the conditional pipeline; executes always and sets the conditional flag.	`ceq`, `cxoreq`
Primary operand combination	`reg, imm`, `imm`, `reg, reg` or `reg`	`r1, 5`
Secondary operand combination[7]	`imm, reg` or `imm, imm`; available only with the `vxcall` modifier on select instructions.	`3, 5`
Comment	Arbitrary text prepended with a semicolon, which will be ignored up to the newline.	`; text`

The following table demonstrates the compatibility of various language constructs (*can I use both X and Y within a single instruction?*)[8], [9]:

x	`vxcall`	stack E/A[10]	stack E/A'	E/A
`vxcall`	N/A	Yes	Yes	Yes
stack E/A	Yes	N/A	No	No
stack E/A'	Yes	No	N/A	No
E/A	Yes	No	No	N/A

3.3 Instruction set

Mnemonic	Variant			`vxcall`	acceptable encodings	Action[11]
	Pure	Virtual[12]	Conditional			
`add`	N/A	N/A	`cadd`	No	`reg,imm`; `reg,reg`	`$1 += $2`
`and`	N/A	N/A	N/A	No	`reg,imm`; `reg,reg`	`$1 = $1 && $2` (logical)
`dec`	N/A	N/A	N/A[13]	No	`reg`	`$1--`
`div`	N/A	N/A	`cdi`, `cdiv`	No	`reg,reg`; `reg,imm`	`$1 /= $2`
`eq`	`eq_`	`sgeq`	`ceq`[14]	No	`reg,reg`; `reg,imm`	`$1 = $1 == $2`
`ge`	`ge_`	N/A	`cge`[14]	No	`reg,reg`; `reg,imm`	`$1 = $1 >= $2`
`gt`	`gt_`	N/A	`cgt`[14]	No	`reg,reg`; `reg,imm`	`$1 = $1 > $2`
`in`[15]	`in_`	N/A	`cin`	No	`reg`	`$1 = getchar()`[16]
`inc`	N/A	N/A	N/A[17]	No	`reg`	`$1++`
`jmp`	N/A	`#call()`[18]	N/A	No	`reg`; `imm`	`IP = $1`
`jnz`	N/A	N/A	`cjnz`, `cjn`	No	`reg, reg`; `reg,imm`	`if($1) IP = $2`
`jz`	`jz_`	N/A	`cjz`	No	`reg, reg`; `reg,imm`	`if(!$1) IP = $2`
`lbl`[19]	`lbl`	N/A	N/A	No	`imm`	`$1:`
`le`	`le_`	N/A	`cle`[14]	No	`reg,reg`; `reg,imm`	`$1 = $1 <= $2`
`lt`	`lt_`	N/A	`clt`[14]	No	`reg,reg`; `reg,imm`	`$1 = $1 < $2`
`mod`	N/A	N/A	`cmd`, `cmod`	No	`reg,reg`; `reg,imm`	`$1 \%= $2`
`mov`	N/A	`lea`	`cmov`, `cmo`	No	`reg,reg`; `reg,imm`	`$1 = $2`
`mul`	N/A	N/A	`cmu`, `cmul`	No	`reg,reg`; `reg,imm`	`$1 *= $2`
`ne`	`ne_`	`xor`, `neq`, `sgne`	`cne`[14]	No	`reg,reg`; `reg,imm`	`$1 = $1 != $2`
`neg`	N/A	N/A	N/A	No	`reg`	`$1 = 2 ^ BITWIDTH - $1`
`not`	N/A	N/A	N/A	No	`reg`	`$1 = !$1`
`or`	`or_`	N/A	N/A	No	`reg,reg`; `reg,imm`	`$1 = $1 || $2`

[7] sometimes also called virtual
[8] it's not allowed to have more than one effective adress per instruction.
[9] E/A' and E/A" are mutually exclusive with any other construct
[10] effective adress
[11] `$N` - n-th operand; `#n` - n-th element from the top of the stack; `#` - the stack pointer delta
[12] `lib-bfm` is required to use it
[13] use `csu $1, 1`
[14] a conditional pipeline primer, not a conditional instruction in itself
[15] Not available with the freestanding bfvm target.
[16] EOF = 0
[17] use `cad $1, 1`
[18] a lib-bfm macro, used to call a subprocedure; not a virtual instruction in itself
[19] the legacy numeric label system; use code anchors instead.

out[20]	N/A	N/A	cou, cout	No	reg,reg; reg,imm	putchar($1)
pop	N/A	N/A	cpop, cpo	No	reg	$1 = #0, #--
push	psh	N/A	cps, cpsh cpush	No	reg, imm	#0 = $1, #++
movf	rcl	N/A	crcl, crc	Yes	reg,reg, reg,imm	$1 = taperam[$2]
sto	N/A	N/A	csto, cst	Yes	reg,reg, reg,imm	taperam[$1] = $2
sub	N/A	N/A	csub, csu	No	reg,reg, reg,imm	$1 -= $2
swp	N/A	xchg	cswp, csw, cxchg	No	reg,reg	$T = $1,$1 = $2,$2 = $1
clr	N/A	N/A	N/A	No	reg	$1 = 0
ret	N/A	N/A	cre, cret	No	N/A	IP = #0, #--
end	N/A	N/A	N/A	No	N/A	IP = 0[21]
log	N/A	N/A	N/A	No	reg	$1 = $1 >= 0[22]
asl	N/A	u2s[23]	csl, casl	No	reg	$1 <<= 1
asr	N/A	s2u[24]	csr, casr	No	reg	$1 >>= 1
pow	N/A	N/A	cpw	No	reg,reg, reg,imm	$1 = pow($1, $2)
srv	N/A	finv	csrv, crv	No	N/A	swp #0, #1
amp	N/A	N/A	cam, camp	Yes	reg,reg, reg,imm	taperam[$1] += $2
smp	N/A	N/A	csm, csmp	Yes	reg,reg, reg,imm	taperam[$1] -= $2
nav[25]	N/A	N/A	N/A	No	reg	N/A
band	x00	N/A	N/A	No	reg,reg, reg,imm	$1 &= $2
bor	x01	N/A	N/A	No	reg,reg, reg,imm	$1 |= $2
bxor	x02	N/A	N/A	No	reg,reg, reg,imm	$1 ^= $2
bneg	x03	N/A	N/A	No	reg	$1 = ~$1
cflip	x04	N/A	N/A	No	N/A	f1 = !f1
ots	N/A	N/A	cot, cots	Yes	reg,reg, reg,imm	taperam[$2] = $1
gcd	N/A	N/A	cgc, cgcd	No	reg,reg, reg,imm	$1 = gcd($1, $2)
fmul	fmu	N/A	N/A	No	N/A	#2/#1=#4/#3*(#2/#1) #-=2
fdiv	fdi	N/A	N/A	No	N/A	#2/#1=#4/#3/(#2/#1) #-=2
fadd	fad	N/A	N/A	No	N/A	#2/#1=#4/#3+#2/#1 #-=2
fsub	fsu	N/A	N/A	No	N/A	#2/#1=#4/#3-#2/#1 #-=2
freduce	fre	N/A	N/A	No	N/A	T=gcd(#1,#2);#1/=T;#2/=T
par	N/A	cpa, cpar	N/A	No	reg	N/A[26]
candeq	x05	N/A	N/A	No	reg,reg, reg,imm	f1 = f1 && ($1 == $2)
candne	x06	N/A	N/A	No	reg,reg, reg,imm	f1 = f1 && ($1 != $2)
candle	x07	N/A	N/A	No	reg,reg, reg,imm	f1 = f1 && ($1 <= $2)
candge	x08	N/A	N/A	No	reg,reg, reg,imm	f1 = f1 && ($1 >= $2)
candlt	x09	N/A	N/A	No	reg,reg, reg,imm	f1 = f1 && ($1 < $2)
candgt	x0A	N/A	N/A	No	reg,reg, reg,imm	f1 = f1 && ($1 > $2)
coreq	x0B	N/A	N/A	No	reg,reg, reg,imm	f1 = f1 || ($1 == $2)
corne	x0C	N/A	N/A	No	reg,reg, reg,imm	f1 = f1 || ($1 != $2)
corle	x0D	N/A	N/A	No	reg,reg, reg,imm	f1 = f1 || ($1 <= $2)
corge	x0E	N/A	N/A	No	reg,reg, reg,imm	f1 = f1 || ($1 >= $2)
corgt	x10	N/A	N/A	No	reg,reg, reg,imm	f1 = f1 || ($1 > $2)
corlt	x0F	N/A	N/A	No	reg,reg, reg,imm	f1 = f1 || ($1 < $2)
cxoreq	x11	N/A	N/A	No	reg,reg, reg,imm	f1 = f1 ^^ ($1 == $2)
cxorne	x12	N/A	N/A	No	reg,reg, reg,imm	f1 = f1 ^^ ($1 != $2)
cxorle	x13	N/A	N/A	No	reg,reg, reg,imm	f1 = f1 ^^ ($1 <= $2)
cxorge	x14	N/A	N/A	No	reg,reg, reg,imm	f1 = f1 ^^ ($1 >= $2)
cxorgt	x16	N/A	N/A	No	reg,reg, reg,imm	f1 = f1 ^^ ($1 > $2)
cxorlt	x15	N/A	N/A	No	reg,reg, reg,imm	f1 = f1 ^^ ($1 < $2)

[20]Not available with the freestanding bfvm target.
[21]Exit the application
[22]equivalent to gt r1, 0
[23]Convert an unsigned number to a signed number. Because signed range is smaller than the unsigned range, high bits of the register may be lost.
[24]Convert a signed number to an unsigned number. The sign bit will be discarded.
[25]A low level instruction, used to move the memory pointer to the cell, where the register data from the operand resides.
[26]Can't be represented trivially; the instruction is used to compute parirty.

stk	N/A	N/A	N/A[27]	No	imm	N/A
org	N/A	N/A	N/A[27]	No	imm	N/A[28]
seg	N/A	N/A	N/A[27]	No	imm	N/A[29]
rse	N/A	N/A	N/A[27]	No	N/A	N/A[30]
db	N/A	N/A	N/A[31]	No	imm	N/A[32]
txt	N/A	N/A	N/A[31]	No	text imm	N/A[33]
raw	N/A	N/A	N/A[31]	No	imm	N/A[34]
dup	N/A	N/A	N/A	No	N/A	#++, #1=#2
shr	N/A	N/A	N/A	No	reg,reg, reg,imm	times($2) $1=asr($1)
shl	N/A	N/A	N/A	No	reg,reg, reg,imm	times($2) $1=asl($1)
dsc	N/A	N/A	N/A	No	N/A	#--
sgt	N/A	N/A	N/A	No	reg,reg, reg,imm	$1=#$2
spt	N/A	N/A	N/A	Yes	reg,reg, reg,imm	#$2=$1
sle	N/A	N/A	N/A	No	reg	$1=#[35]
tps	N/A	N/A	N/A	No	reg,reg, reg,imm	#$1=$2
fcpush	fps	fcpsh	N/A	No	N/A	#0=f1,#++
fcpop	fpo	N/A	N/A	No	N/A	f1=#0,#--
sgmul	s03	N/A	N/A	No	reg,imm; reg,reg	$1 (sgn)*= $2
sgdiv	s04	N/A	N/A	No	reg,imm; reg,reg	$1 (sgn)/= $2
sgmod	s05	N/A	N/A	No	reg,imm; reg,reg	$1 (sgn)\%= $2
sgsub	s06	N/A	N/A	No	reg,imm; reg,reg	$1 (sgn)-= $2
sgadd	s07	N/A	N/A	No	reg,imm; reg,reg	$1 (sgn)+= $2
cbegin	cbs	N/A	N/A	No	N/A	if(f1) {[36]
cend	cbe	N/A	N/A	No	N/A	}
sgn	s00	N/A	N/A	No	reg	$1=signum($1)
abs	s01	N/A	N/A	No	reg	$1=abs($1)
sneg	s02	N/A	N/A	No	reg	$1=-$1
dup2	dp2	fdup	N/A	No	N/A	#+=2,#1=#2,3
axl	N/A	N/A	N/A	No	N/A	N/A[37]
xlt	N/A	N/A	N/A	No	reg,reg, reg,imm	$1 (sgn)< $2
xle	N/A	N/A	N/A	No	reg,reg, reg,imm	$1 (sgn)<= $2
xgt	N/A	N/A	N/A	No	reg,reg, reg,imm	$1 (sgn)> $2

3.4 Arithmetics

asm2bf supports a wide variety of numerical operations. Most of arithmetical instructions operate on registers (i.e. `add` or `div`), meanwhile some specialized instructions (i.e. `fadd`, `freduce` - simply put, floating point operations) operate on the stack[38].

All numbers are **unsigned by default** and they're expected to be expressed in the program as decimal literals or ASCII character constants. You can express numbers as hexadecimal literals using a clever trick which involves the Lua preprocessor - `mov r1, $(0x30)`. Signed numbers can be stored in normal registers (`r1` - `r6`), but they need to be loaded in a special way using the `signed` macro. Let's look at this example program[39]:

```
; load -2 to r1
mov r1, $(signed(-2))
; read a digit, make it signed
```

[27] evaluated at compile time
[28] sets the stack size; invalidates both the origin and the segment, *but not the segmentation mode.*
[29] sets the current segment
[30] enables real segmentation mode. Available only with the bfvm target.
[31] partially evaluated at compile time; the result of the operation differs depending on the state
[32] memory at the origin is altered; when it's nonzero already, the behavior is undefined.
[33] behaves just like a sequence of db's; will not terminate strings.
[34] embeds a given instruction into the resulting binary; beware of bfvm vs brainfuck target differences!
[35] `#` doesn't exist; querying it is `O(N)`
[36] No jumps / labels are allowed inside conditional blocks.
[37] approximate a floating point number with low precision
[38] idea borrowed from the x87 math coprocessor
[39] recommended to build it without the label system; the program assumes that 0 is positive

```
in r2
sub r2, .0
u2s r2
; add them
sgadd r1, r2
sgn r1 ; r1: 1 if negative, 0 if positive.
; a clever trick: load '+' to r2 if r1 = 0, load '-' to r2 if r1 = 1
; first load '+'
mov r2, .+
; r1 <<= 1, so r1 = 2 or r1 = 0
asl r1
; '+'+2='-', '+'+0='+'
add r2, r1
; display it
out r2
```

Let's try using the floating point extensions now. We will try to calculate the value of $\sum_{k=1}^{n} \frac{1}{n}$ for any $n > 1$.[40]

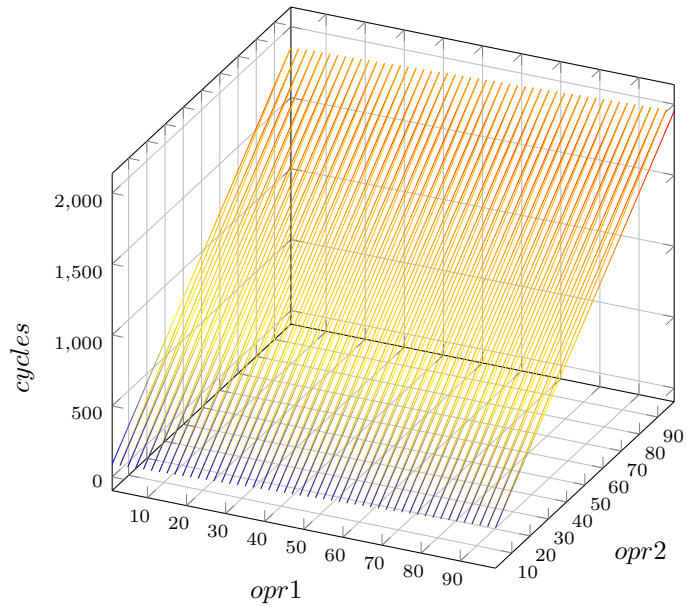
```
stk 5
org 0

; calculate H(9)
mov r1, 9
push %lab
#call("harmonic")
@lab
; H(9) is equal to 7129/2520
    eq  r1, 7129
    eq  r2, 2520
    and r1, r2
    add r1, .0
    out r1

@harmonic
    inc r1
    psh r1
    psh 1
@loop
    inc r1
    psh r1
    psh 1
    fadd
    freduce
    ceq r1, 9
    cjz %loop
    pop r1
    pop r2
    ret
```

3.4.1 add

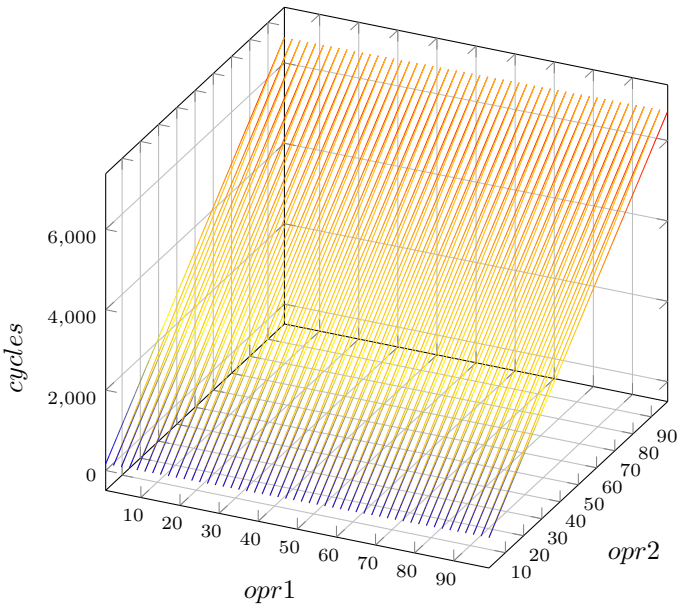
`add` is an instruction used for adding two unsigned numbers. It was observed to execute much quicker in the first part of the permagen. `add`'s complexity grows in regards to the second operand, so it's preferable to use `add`'s `regA, regB` encoding where values $A < B$. `reg, imm` encoding is special-cased within the compiler to emit an $O(n)$ size store in regards to B. The overflow behavior is undefined, but `bfi` assumes wrapping over.

[40] This program only checks for the validity of the answer. Approximating the result using 'axl' and printing the resulting value is left as an exercise for the reader.

add in the first part of the permagen



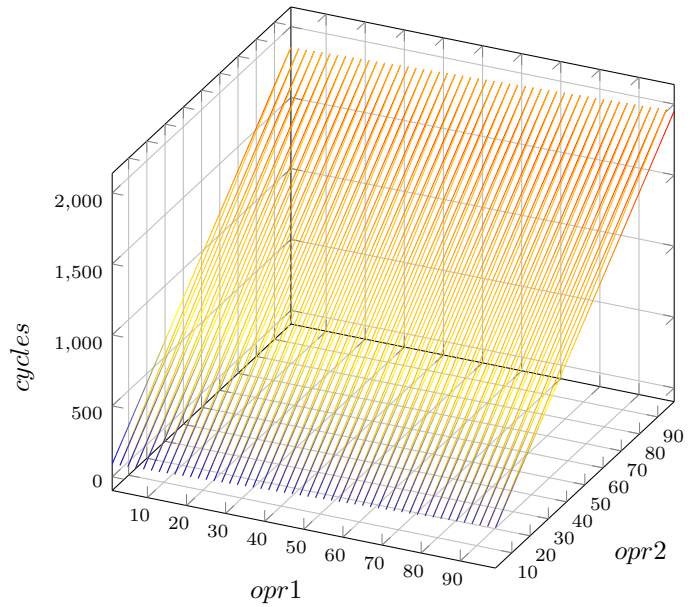
add in the second part of the permagen



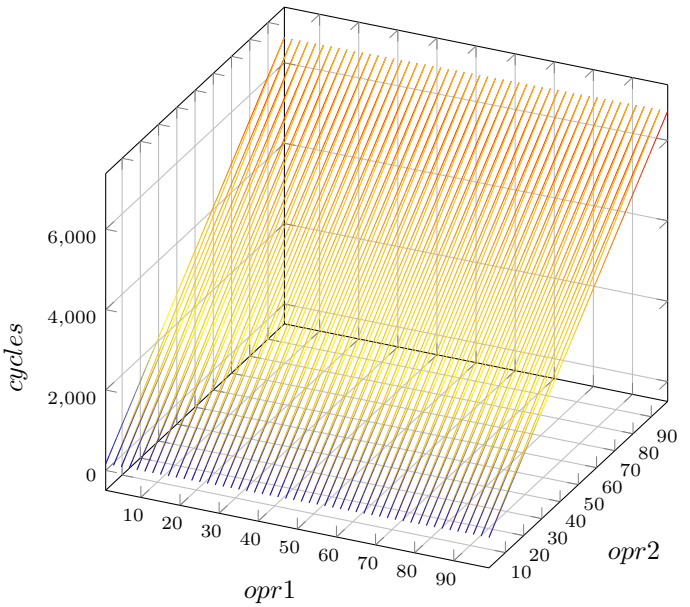
3.4.2 sub

`sub` is an instruction used for subtracting two unsigned numbers. Just like `add`, it was observed to execute much quicker in the first part of the permagen. `sub`'s complexity grows in regards to the second operand, so it's preferable to use `add`'s `regA, regB` encoding where values $A < B$. `reg, imm` encoding is special-cased within the compiler to emit an $O(n)$ size store in regards to B. The underflow behavior is undefined, but `bfi` assumes wrapping over.

sub in the first part of the permagen



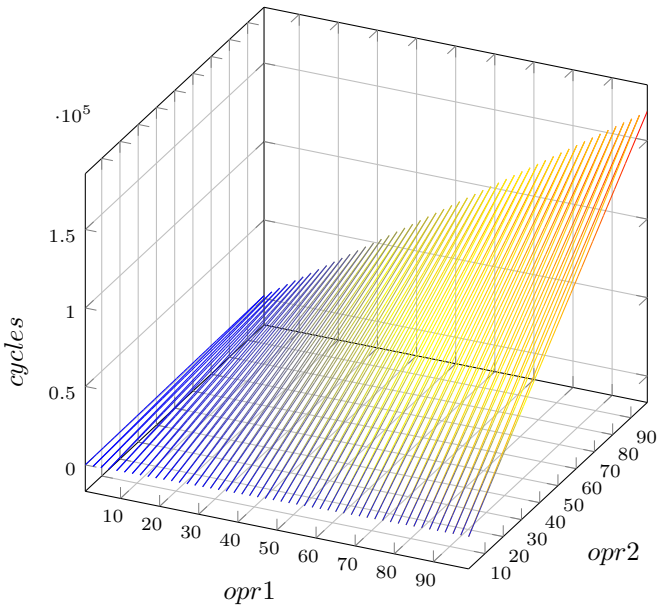
sub in the second part of the permagen



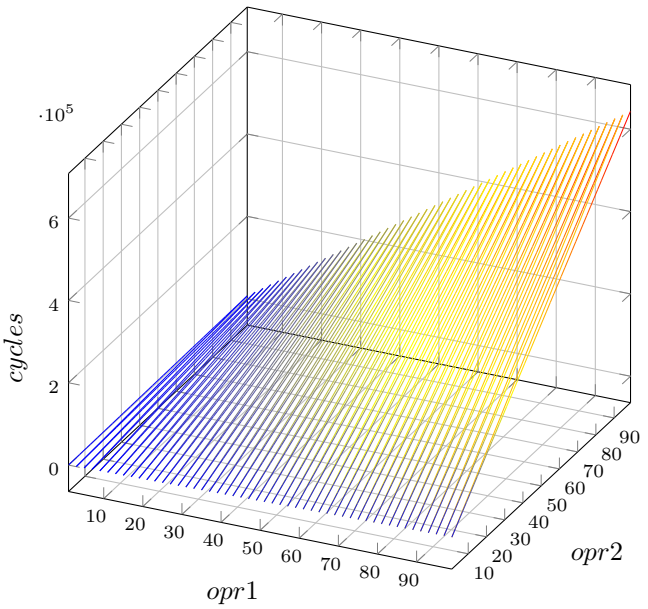
3.4.3 mul

`mul` is an instruction used for unsigned multiplication. It's $O(n^2)$ in the general case and it was observed to execute quicker in the first part of the permagen. `mul`'s complexity grows in regards to the both operands at the same rate, so no `mul` encoding is favoured.

mul in the first part of the permagen



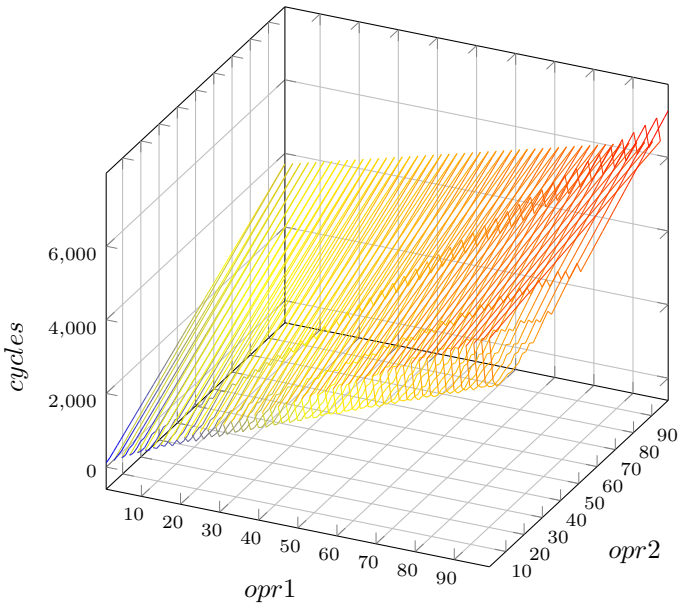
mul in the second part of the permagen



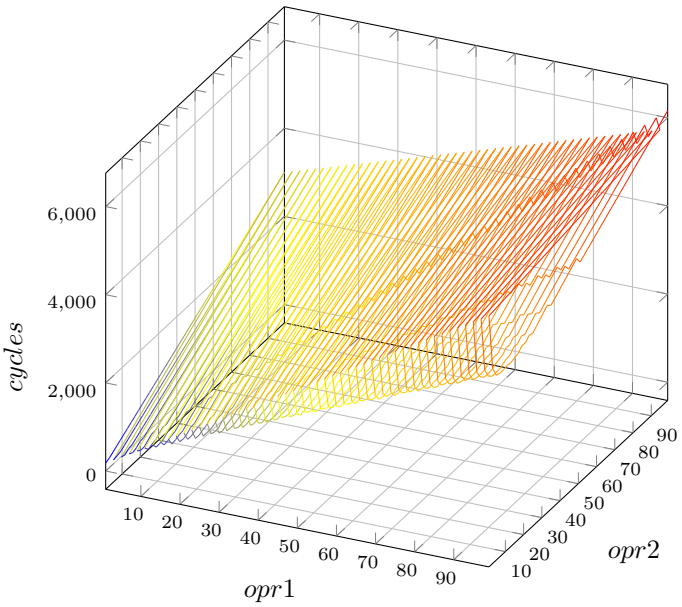
3.4.4 div

`div` is an instruction used for unsigned division. It's compoutational complexity grows fairly slowly (compared to `mul`) in the general case. Division by zero results in undefined behavior, which for the `asmbf` implementation of the asm2bf language happens to be an infinite loop. `div`'s behavior in the first part of the permagen looks amortized; `div` in the second part of the permagen is slower or faster depending on the operands. The performance plot suggests that for `div A, B`, $A > B$ will be faster than $A < B$ for a specific range of different A and B. This property is illustrated in more detail by the plot.

div in the first part of the permagen



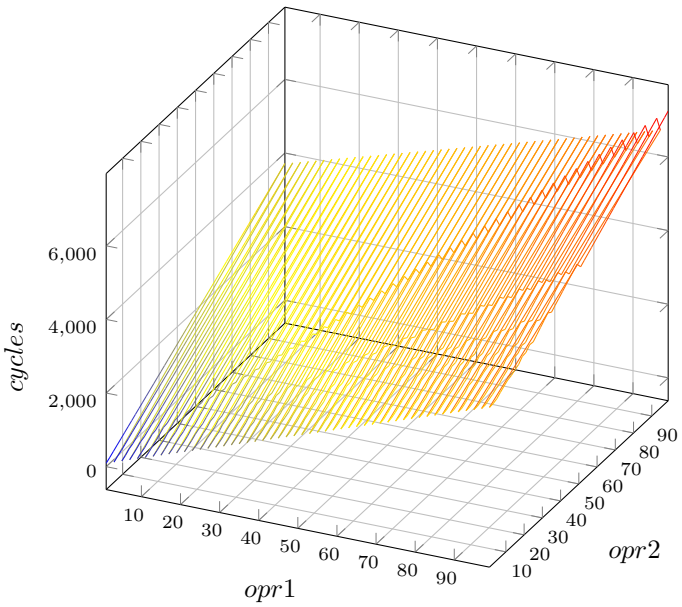
div in the second part of the permagen



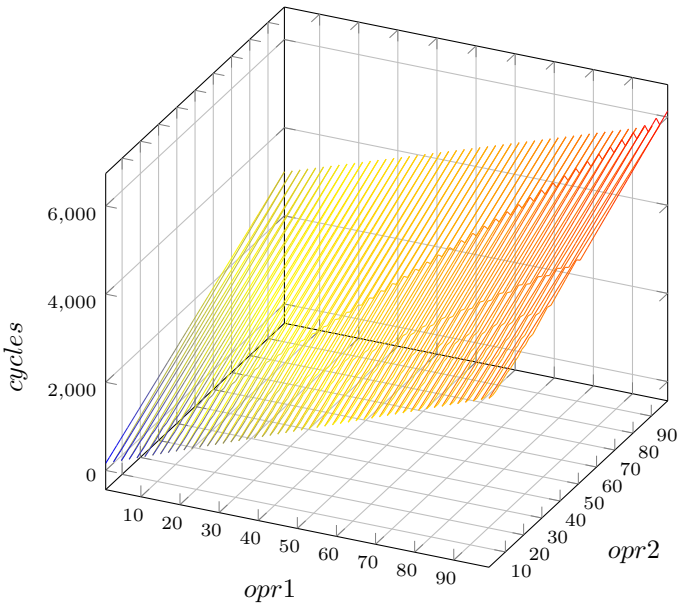
3.4.5 mod

`mod` is an instruction used for unsigned modulus. It's compoutational complexity grows in a way comparable to `div`. Division by zero results in undefined behavior, which for the `asmbf` implementation of the asm2bf language happens to be an infinite loop. `mod` doesn't exhibit the amortization behavior; `mod` in the second part of the permagen is slower. The performance plot suggests that for `div A, B`, $A > B$ will be faster than $A < B$ for a specific range of different A and B. This property is illustrated in more detail by the plot.

mod in the first part of the permagen



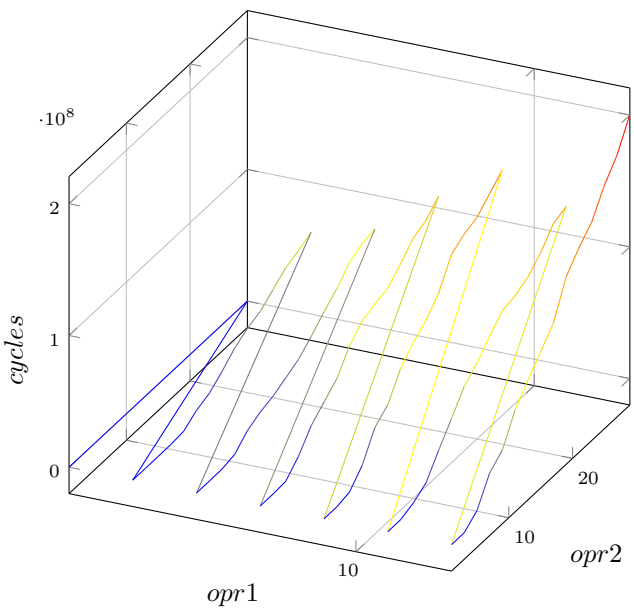
mod in the second part of the permagen



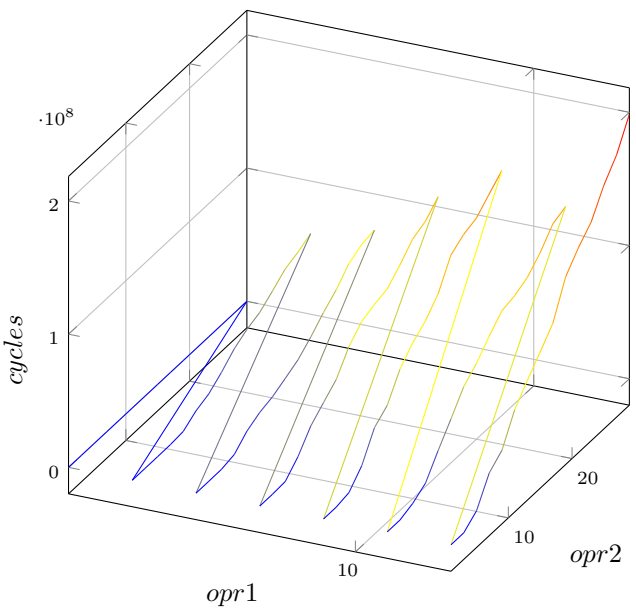
3.4.6 pow

`pow` is an instruction used for unsigned integer power. It's computational complexity grows very quickly and it can be approximated to be around $O(x^{y!})$. Assumes $n^0 = 1$, even for $n = 0$.

pow in the first part of the permagen



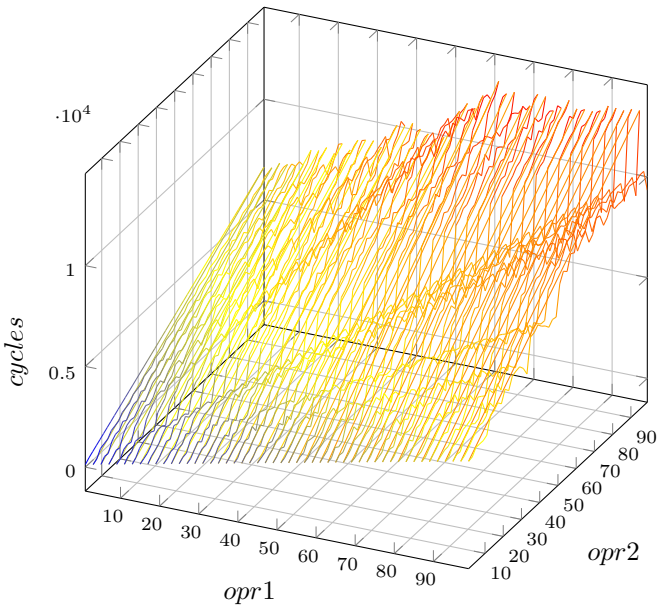
pow in the second part of the permagen



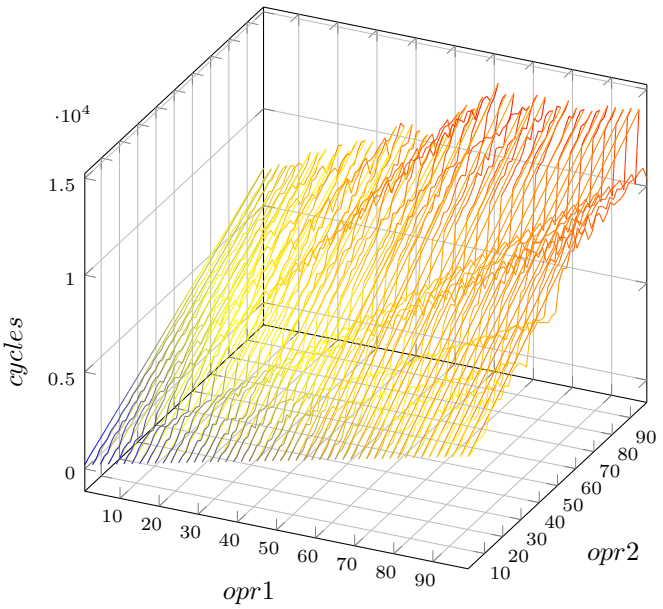
3.4.7 gcd

`gcd` is an instruction used for unsigned integer greatest common divisor calculation. It's computational complexity is considered to be $O(n)$. It's faster in the first part of the permagen and it's generally faster if $A > B$ in `gcd A, B`.

gcd in the first part of the permagen



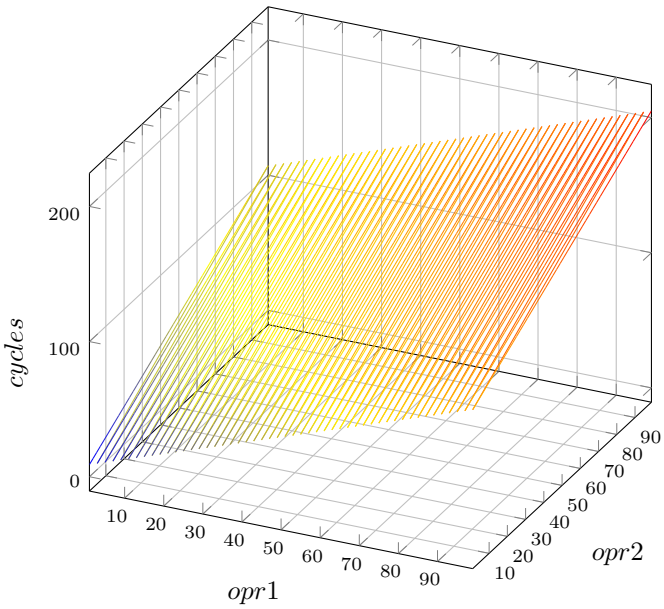
gcd in the second part of the permagen



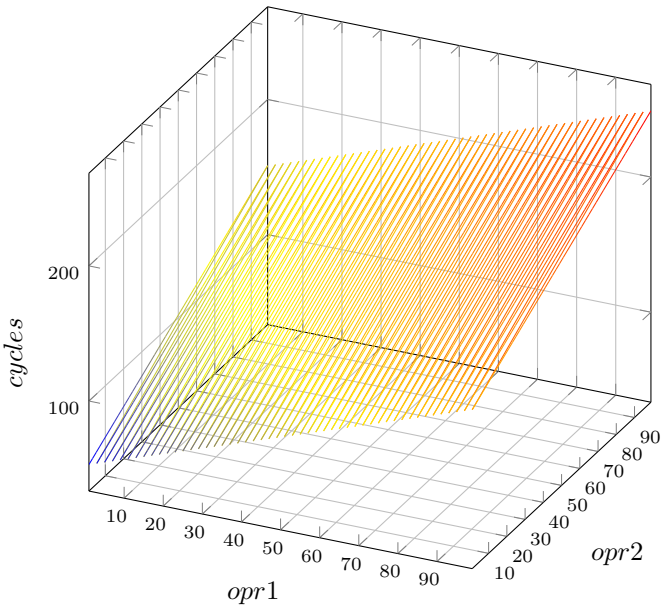
3.4.8 dyadic bit operations

`band`, `bor` and `bxor`, respectively take bitwise and, bitwise or, and bitwise xor of two unsigned integers. Their performance plots look fairly similiar:

bxor in the first part of the permagen

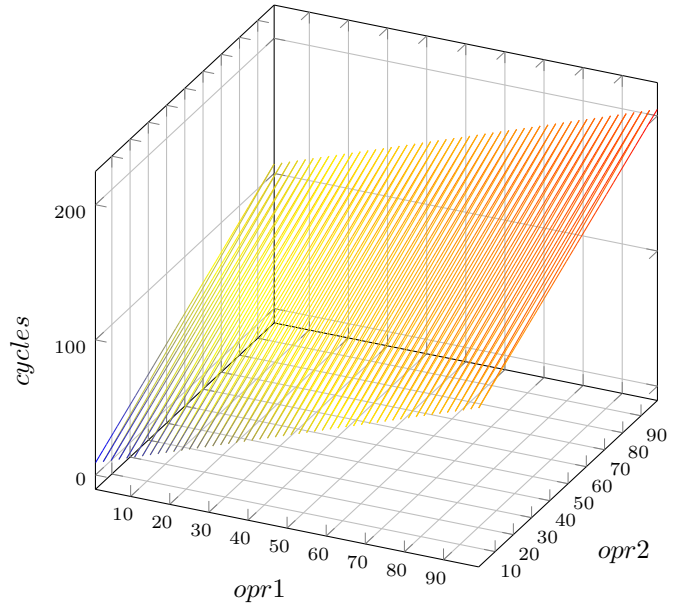


bxor in the second part of the permagen

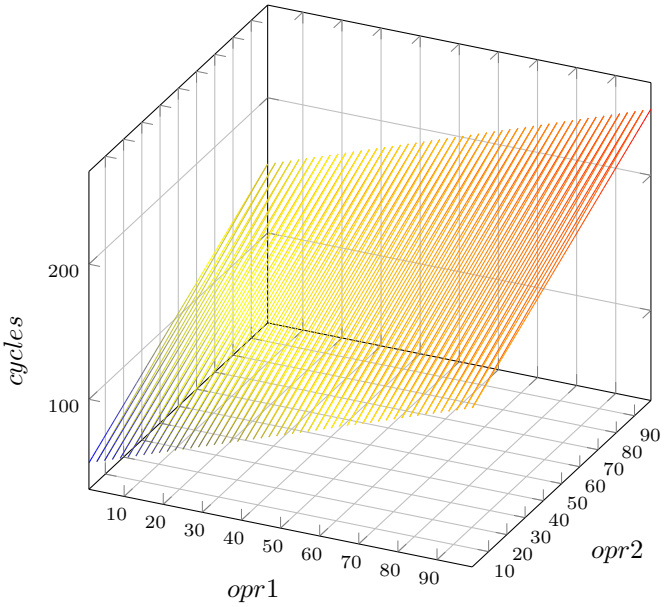


All of them are faster in the 1st part of the permagen. It might be worth considering to rewrite some of the operations that already utilized `mul`, `div`, or other classical operations to using bit operations. For example, `sgn r1` can be optimized as `band r1, 1` (because $x \mod 2 \Leftrightarrow x\&1$, or, more generally speaking, $x \mod 2^n \Leftrightarrow x\&(2^n - 1)$)

band in the first part of the permagen

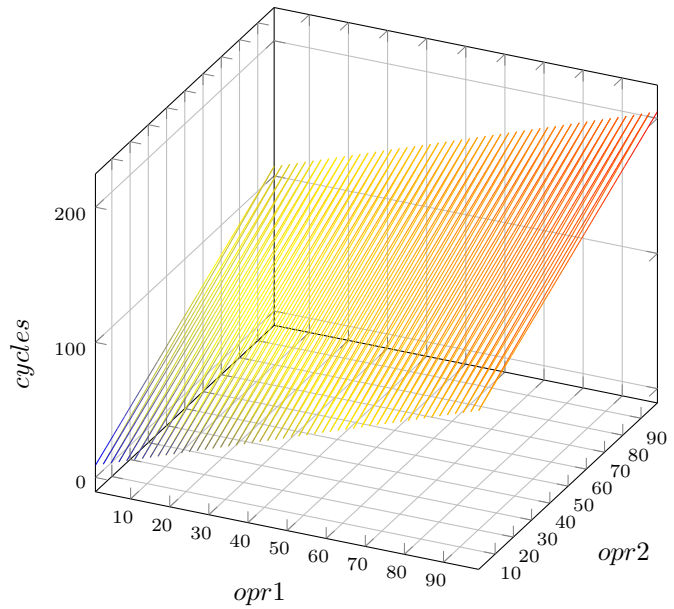


band in the second part of the permagen

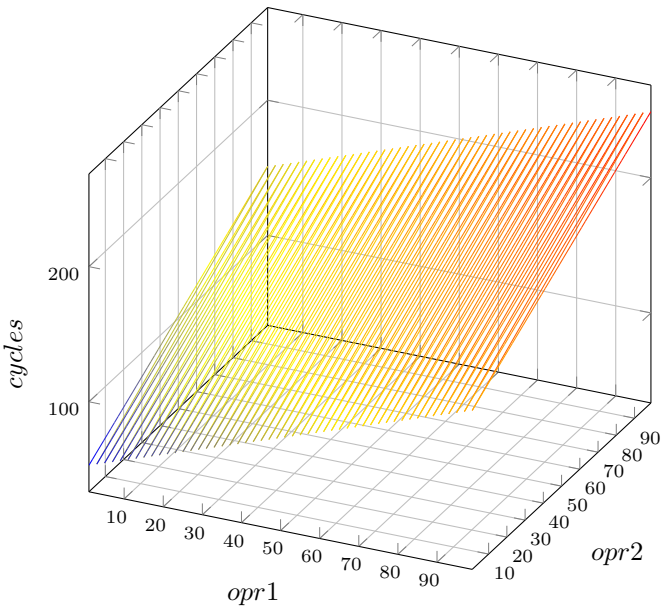


The only difference between the microcode of these instructions is the iterated operation (the operation that executes on individual bits), so the plots don't differ, because the performance of all iterated operations is comparable.

bor in the first part of the permagen



bor in the second part of the permagen



3.4.9 clr, inc and dec

`inc` and `dec` always execute in one cycle (assuming no pointer navigation is requried). They're strictly equivalent to `add rX, 1` and `sub rX, 1`, respectively. They've been provided for the programmer's convience and to serve semantical meaning. `clr` emits the same output as `mov rX, 0`, and the rationale behind adding it is exactly the same. `clr` has $O(n)$ computational complexity due to platform limitations (`[-]`, the most efficient way to zero a cell, is $O(n)$)[41]

3.4.10 and, or, not, log

`and`, `or`, `not` and `log` are logical operations. First two are dyadic, the other two are monadic. `and A, B` takes the least cycles for the case where $A = B = 0$, and the most cycles for $A = B = 1$. `and` takes marginally more cycles for $A < B$. `and` *will* work for $A > 1$ or $B > 1$, but it will take more cycles to process these cases. `or` exhibits the same behavior. `neg A` is usually implemented as $1 - A$[42]. For this reason, the `neg` will *not* work correctly for $A > 1$. To work around it, the `log` instruction

[41] ...but bfi and most optimizing interpreters will fuse the clear operation into a single $O(1)$ operation.

[42] $1 - 1 = 0$, $1 - 0 = 1$, $1 - 2 = -1 = 65535$

has been created. It turns a truthy value into a literal `true` (1), and a falsy value to literal `false` (0). It's operation is essentialy $A > 0$ for `log A`, because asm2bf assumes unsigned numbers[43].

3.4.11 mov

[43]this would work with signed numbers too, because 0 is literal 0, and every other value can be reasonably casted implicitly to an unsigned number, and in this representation it's truthy