Corewar

1.0

Generated by Doxygen 1.13.2

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Data Structure Documentation

3.1 arena_s Struct Reference

`#include <op.h>`

Data Fields

- unsigned char * vm
- bool live_signals [4]
- bool alive_champions [4]
- int dump
- champion_t * champions [4]
- int nb_champions
- args_t * args
- int nb_live_signal

3.1.1 Field Documentation

3.1.1.1 alive_champions

```
bool arena_s::alive_champions[4]
```

3.1.1.2 args

```
args_t* arena_s::args
```

3.1.1.3 champions

```
champion_t* arena_s::champions[4]
```

3.1.1.4 dump

```
int arena_s::dump
```

3.1.1.5 live_signals

```
bool arena_s::live_signals[4]
```

3.1.1.6 nb_champions

```
int arena_s::nb_champions
```

3.1.1.7 nb_live_signal

```
int arena_s::nb_live_signal
```

3.1.1.8 vm

```
unsigned char* arena_s::vm
```

The documentation for this struct was generated from the following file:

- include/op.h

3.2 args_t Struct Reference

```
#include <op.h>
```

Data Fields

- int offset
- int arg_index
- int arg_type
- int * arg_type_array
- int nb_args

3.2.1 Field Documentation

3.2.1.1 arg_index

```
int args_t::arg_index
```

3.2.1.2 arg_type

```
int args_t::arg_type
```

3.2.1.3 arg_type_array

```
int* args_t::arg_type_array
```

3.2.1.4 nb_args

```
int args_t::nb_args
```

3.2.1.5 offset

```
int args_t::offset
```

The documentation for this struct was generated from the following file:

- include/op.h

3.3 buffer_s Struct Reference

```
#include <my.h>
```

Data Fields

- char buffer [1024]
- int pos
- int len

3.3.1 Field Documentation

3.3.1.1 buffer

```
char buffer_s::buffer[1024]
```

3.3.1.2 len

```
int buffer_s::len
```

3.3.1.3 pos

```
int buffer_s::pos
```

The documentation for this struct was generated from the following file:

- include/my.h

3.4 champion_s Struct Reference

```
#include <op.h>
```

Data Fields

- int registers [16]
- int id
- int position
- char * name
- char * filename
- int len
- proc_t * procs

3.4.1 Field Documentation

3.4.1.1 filename

```
char* champion_s::filename
```

3.4.1.2 id

```
int champion_s::id
```

3.4.1.3 len

```
int champion_s::len
```

3.4.1.4 name

```
char* champion_s::name
```

3.4.1.5 position

```
int champion_s::position
```

3.4.1.6 procs

```
proc_t* champion_s::procs
```

3.4.1.7 registers

```
int champion_s::registers[16]
```

The documentation for this struct was generated from the following file:

- include/op.h

3.5 flags_t Struct Reference

```
#include <op.h>
```

Data Fields

- unsigned int dump
- unsigned int n
- unsigned int a
- char * prog_name

3.5.1 Field Documentation

3.5.1.1 a

```
unsigned int flags_t::a
```

3.5.1.2 dump

```
unsigned int flags_t::dump
```

3.5.1.3 n

```
unsigned int flags_t::n
```

3.5.1.4 prog_name

```
char* flags_t::prog_name
```

The documentation for this struct was generated from the following file:

- include/op.h

3.6 format_s Struct Reference

`#include <my.h>`

Data Fields

- int flags
- int width
- int precision
- int length

3.6.1 Field Documentation

3.6.1.1 flags

`int format_s::flags`

3.6.1.2 length

`int format_s::length`

3.6.1.3 precision

`int format_s::precision`

3.6.1.4 width

`int format_s::width`

The documentation for this struct was generated from the following file:

- include/my.h

3.7 header_s Struct Reference

`#include <op.h>`

Data Fields

- int magic
- char prog_name [128+1]
- int prog_size
- char comment [2048+1]

3.7.1 Field Documentation

3.7.1.1 comment

```
char header_s::comment[2048+1]
```

3.7.1.2 magic

```
int header_s::magic
```

3.7.1.3 prog_name

```
char header_s::prog_name[128+1]
```

3.7.1.4 prog_size

```
int header_s::prog_size
```

The documentation for this struct was generated from the following file:

- include/op.h

3.8 op_s Struct Reference

```
#include <op.h>
```

Data Fields

- char * mnemonique
- char nbr_args
- args_type_t type [4]
- char code
- int nbr_cycles
- char * comment

3.8.1 Field Documentation

3.8.1.1 code

```
char op_s::code
```

3.8.1.2 comment

```
char* op_s::comment
```

3.8.1.3 mnemonique

```
char* op_s::mnemonique
```

3.8.1.4 nbr_args

```
char op_s::nbr_args
```

3.8.1.5 nbr_cycles

```
int op_s::nbr_cycles
```

3.8.1.6 type

```
args_type_t op_s::type[4]
```

The documentation for this struct was generated from the following file:

- include/op.h

3.9 printf_s Struct Reference

```
#include <my.h>
```

Data Fields

- char c
- void(* func)(va_list, buffer_t *, format_t *)

3.9.1 Field Documentation

3.9.1.1 c

```
char printf_s::c
```

3.9.1.2 func

```
void(* printf_s::func) (va_list, buffer_t *, format_t *)
```

The documentation for this struct was generated from the following file:

- include/my.h

3.10 proc_s Struct Reference

`#include <op.h>`

Data Fields

- int pc
- int carry
- unsigned int wait
- struct proc_s * next

3.10.1 Field Documentation

3.10.1.1 carry

`int proc_s::carry`

3.10.1.2 next

`struct proc_s* proc_s::next`

3.10.1.3 pc

`int proc_s::pc`

3.10.1.4 wait

`unsigned int proc_s::wait`

The documentation for this struct was generated from the following file:

- include/op.h

3.11 tab_instructions_t Struct Reference

`#include <instructions.h>`

Data Fields

- char * command
- void(* instruction_func)(arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *proc)

3.11.1 Field Documentation

3.11.1.1 command

```
char* tab_instructions_t::command
```

3.11.1.2 instruction_func

```
void(* tab_instructions_t::instruction_func) (arena_t *arena, __attribute_maybe_unused__↩
champion_t *champion, proc_t *proc)
```

The documentation for this struct was generated from the following file:

- include/instructions.h

Chapter 4

File Documentation

4.1 include/instructions.h File Reference

```
#include "op.h"
```

Data Structures

- struct tab_instructions_t

Functions

- void execute_live (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *proc)
  Executes the `live` instruction.
- void execute_add (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *proc)
  Executes the `add` instruction.
- void execute_sub (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *proc)
  Executes the 'sub' instruction.
- void execute_zjmp (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *proc)
  Executes the 'zjmp' instruction.
- void execute_ld (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *proc)
  Executes the `ld` (load) instruction.
- void execute_st (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *proc)
  Executes the 'st' instruction.
- void execute_and (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *proc)
  Executes the `and` instruction.
- void execute_or (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *proc)
  Executes the 'or' instruction.
- void execute_xor (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *proc)
  Executes the 'xor' instruction.
- void execute_ldi (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *proc)
  Executes the `ldi` (load index) instruction.
- void execute_sti (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *proc)
  Executes the 'sti' instruction.
- void execute_fork (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *proc)

Executes the `fork` instruction.

- void execute_lld (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *proc)

  Executes the 'lld' instruction.
- void execute_lldi (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *proc)

  Executes the 'lldi' instruction.
- void execute_lfork (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *proc)

  Executes the `lfork` (long fork) instruction.
- void execute_aff (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *proc)

  Executes the `aff` instruction.

Variables

- const tab_instructions_t tab [16]

4.1.1 Function Documentation

4.1.1.1 execute_add()

```
void execute_add (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the `add` instruction.

Reads three register numbers from memory, verifies their validity, adds the values from the first two registers, and stores the result in the third register. Updates the carry flag to 1 if the result is 0, or to 0 otherwise.

Parameters

arena	Pointer to the arena structure containing VM memory.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the current process executing the instruction.

4.1.1.2 execute_aff()

```
void execute_aff (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the `aff` instruction.

This function fetches a register number from memory and prints the content of the register as a character (value % 256). If the register number is invalid, the instruction is ignored.

Parameters

arena	Pointer to the arena structure containing the VM memory.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the current process executing the instruction.

4.1.1.3 execute_and()

```
void execute_and (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the `and` instruction.

This function fetches two parameters from memory based on their types, performs a bitwise AND operation, and stores the result in the target register. It also updates the carry flag depending on whether the result is zero.

Parameters

arena	Pointer to the arena containing VM state and memory.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the process executing the instruction.

4.1.1.4 execute_fork()

```
void execute_fork (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the `fork` instruction.

In Corewar, `fork` creates a new process at a position defined relative to the current one. This function is currently a stub and does not yet implement the expected behavior.

Parameters

arena	Pointer to the arena containing VM state and memory.
champion	Pointer to the champion executing the instruction.
proc	Pointer to the process executing the instruction.

4.1.1.5 execute_ld()

```
void execute_ld (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the `ld` (load) instruction.

The `ld` instruction takes a direct or indirect value and loads it into a register. If the loaded value is zero, the carry flag is set.

Parameters

arena	Pointer to the arena structure.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the process executing the instruction.

4.1.1.6 execute_ldi()

```
void execute_ldi (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the `ldi` (load index) instruction.

This function loads data from memory into a register. The memory address is computed by summing the first two parameters and applying the IDX_MOD restriction. The result is stored in the register defined by the third parameter.

Parameters

arena	Pointer to the arena structure containing VM state.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the current process executing the instruction.

4.1.1.7 execute_lfork()

```
void execute_lfork (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the `lfork` (long fork) instruction.

This instruction should duplicate the current process and place the new process at a PC equal to the current PC plus a given direct value. Unlike `fork`, `lfork` does not apply the IDX_MOD restriction.

Parameters

arena	Pointer to the arena structure containing VM state.
champion	Pointer to the champion owning the process.
proc	Pointer to the current process executing the instruction.

4.1.1.8 execute_live()

```
void execute_live (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the `live` instruction.

This instruction announces that the current champion is alive. It reads the champion ID from the memory following the program counter and marks the corresponding champion as live.

Parameters

arena	Pointer to the arena structure.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the process executing the instruction.

4.1.1.9 execute_lld()

```
void execute_lld (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the 'lld' instruction.

Loads a value (direct or indirect) into a register of the champion and updates the carry flag of the current process accordingly.

Parameters

arena	Pointer to the arena structure containing the virtual machine state.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the current process executing the instruction.

4.1.1.10 execute_lldi()

```
void execute_lldi (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the 'lldi' instruction.

Loads a value from memory calculated by adding two parameters, then stores the result in a register and updates the carry flag.

Parameters

arena	Pointer to the arena structure containing the VM state.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the current process executing the instruction.

4.1.1.11 execute_or()

```
void execute_or (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the 'or' instruction.

Performs a bitwise OR between two parameters and stores the result in a specified register.

Parameters

arena	Pointer to the arena structure containing the VM state.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the current process executing the instruction.

4.1.1.12 execute_st()

```
void execute_st (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the 'st' instruction.

Stores the value of a register either into another register or into the arena's memory at a calculated address.

Parameters

arena	Pointer to the arena structure containing the VM state.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the current process executing the instruction.

4.1.1.13 execute_sti()

```
void execute_sti (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the 'sti' instruction.

Stores the value of a register at an address computed from the sum of two parameters relative to the current process's program counter.

Parameters

arena	Pointer to the arena structure containing the VM state.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the current process executing the instruction.

4.1.1.14 execute_sub()

```
void execute_sub (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the 'sub' instruction.

Subtracts the value of the second register from the first register and stores the result in a third register. Updates the carry flag.

Parameters

arena	Pointer to the arena structure containing the VM state.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the current process executing the instruction.

4.1.1.15 execute_xor()

```
void execute_xor (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the 'xor' instruction.

Performs a bitwise XOR operation between two parameters and stores the result in a register.

Parameters

arena	Pointer to the arena structure containing the VM state.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the current process executing the instruction.

4.1.1.16 execute_zjmp()

```
void execute_zjmp (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the 'zjmp' instruction.

Jumps to a new position in the arena if the carry flag is set.

The jump offset is a signed 16-bit value read from the instruction arguments. The actual jump position is calculated modulo IDX_MOD.

Parameters

arena	Pointer to the arena structure containing the VM state.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the current process executing the instruction.

4.1.2 Variable Documentation

4.1.2.1 tab

```
const tab_instructions_t tab[16]
```

Initial value:

```
= {
    {"live", &execute_live},
    {"ld", &execute_ld},
    {"st", &execute_st},
    {"add", &execute_add},
    {"sub", &execute_sub},
    {"and", &execute_and},
    {"or", &execute_or},
    {"xor", &execute_xor},
    {"zjmp", &execute_zjmp},
    {"ldi", &execute_ldi},
    {"sti", &execute_sti},
    {"fork", &execute_fork},
    {"lld", &execute_lld},
    {"lldi", &execute_lldi},
    {"lfork", &execute_lfork},
    {"aff", &execute_aff}
}
```

4.2 instructions.h

Go to the documentation of this file.

```
00001 /*
00002 ** EPITECH PROJECT, 2025
00003 ** corewar
00004 ** File description:
00005 ** instructions
00006 */
00007
00008 #ifndef INSTRUCTIONS_H
00009     #define INSTRUCTIONS_H
00010     #include "op.h"
00011
00012 void execute_live(arena_t *arena, __attribute_maybe_unused__
00013     champion_t *champion, proc_t *proc);
00014
00015 void execute_add(arena_t *arena, __attribute_maybe_unused__
00016     champion_t *champion, proc_t *proc);
00017
00018 void execute_sub(arena_t *arena, __attribute_maybe_unused__
00019     champion_t *champion, proc_t *proc);
00020
00021 void execute_zjmp(arena_t *arena, __attribute_maybe_unused__
00022     champion_t *champion, proc_t *proc);
00023
00024
00025 void execute_ld(arena_t *arena, __attribute_maybe_unused__
00026     champion_t *champion, proc_t *proc);
00027
00028 void execute_st(arena_t *arena, __attribute_maybe_unused__
00029     champion_t *champion, proc_t *proc);
00030
00031 void execute_and(arena_t *arena, __attribute_maybe_unused__
00032     champion_t *champion, proc_t *proc);
00033
00034 void execute_or(arena_t *arena, __attribute_maybe_unused__
00035     champion_t *champion, proc_t *proc);
00036
00037 void execute_xor(arena_t *arena, __attribute_maybe_unused__
00038     champion_t *champion, proc_t *proc);
00039
00040 void execute_ldi(arena_t *arena, __attribute_maybe_unused__
00041     champion_t *champion, proc_t *proc);
00042
00043 void execute_sti(arena_t *arena, __attribute_maybe_unused__
00044     champion_t *champion, proc_t *proc);
00045
00046 void execute_fork(arena_t *arena, __attribute_maybe_unused__
00047     champion_t *champion, proc_t *proc);
00048
00049 void execute_lld(arena_t *arena, __attribute_maybe_unused__
00050     champion_t *champion, proc_t *proc);
00051
00052 void execute_lldi(arena_t *arena, __attribute_maybe_unused__
00053     champion_t *champion, proc_t *proc);
00054
00055 void execute_lfork(arena_t *arena, __attribute_maybe_unused__
00056     champion_t *champion, proc_t *proc);
00057
00058 void execute_aff(arena_t *arena, __attribute_maybe_unused__
00059     champion_t *champion, proc_t *proc);
00060
00061 typedef struct {
00062     char *command;
00063     void (*instruction_func)(arena_t *arena,
00064         __attribute_maybe_unused__ champion_t *champion, proc_t *proc);
00065 } tab_instructions_t;
00066
00067 const tab_instructions_t tab[16] = {
00068     {"live", &execute_live},
00069     {"ld", &execute_ld}, //pas fé
00070     {"st", &execute_st}, //pas fé
00071     {"add", &execute_add},
00072     {"sub", &execute_sub},
00073     {"and", &execute_and}, //pas fé
00074     {"or", &execute_or}, //pas fé
00075     {"xor", &execute_xor}, //pas fé
00076     {"zjmp", &execute_zjmp},
00077     {"ldi", &execute_ldi}, //pas fé
00078     {"sti", &execute_sti}, //pas fé
00079     {"fork", &execute_fork}, //pas fé
00080     {"lld", &execute_lld}, //pas fé
00081     {"lldi", &execute_lldi}, //pas fé
00082     {"lfork", &execute_lfork}, //pas fé
```

```
00083     {"aff", &execute_aff} //pas fé
00084 };
00085 #endif //INSTRUCTIONS_H
```

4.3 include/my.h File Reference

```
#include <stdarg.h>
#include <unistd.h>
#include <limits.h>
#include <stdint.h>
#include <stdlib.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/sysmacros.h>
#include <time.h>
```

Data Structures

- struct buffer_s
- struct format_s
- struct printf_s

Typedefs

- typedef struct buffer_s buffer_t
- typedef struct format_s format_t
- typedef struct printf_s printf_t

Functions

- void my_putchar (char)
- void my_putstr (char const *)
- int my_put_nbr (int)
- void my_puterr (char const *)
- int my_strlen (char const *)
- int my_strcmp (char const *, char const *)
- int my_strncmp (char const *, char const *, int n)
- int my_isalnum (char)
- char * my_strcat (char *, char const *)
- char * my_strncat (char *, char const *, int n)
- char * my_strcpy (char *, char const *)
- char * my_strncpy (char *, char const *, int n)
- char * my_revstr (char *)
- char * my_strstr (char *, char const *)
- char * my_strdup (char const *)
- char * my_strcapitalize (char *)
- char * my_strlowcase (char *)
- char * my_strupcase (char *)
- int my_str_isalpha (char const *)

- int my_str_islower (char const *)
- int my_str_isnum (char const *)
- int my_str_isprintable (char const *)
- int my_str_isupper (char const *)
- int my_getnbr (char const *)
- int my_compute_power_rec (int, int)
- int my_compute_square_root (int)
- int my_find_prime_sup (int)
- int my_is_prime (int)
- int my_isneg (int)
- void my_sort_int_array (int *, int)
- void my_swap (int *, int *)
- int my_showmem (char const *, int)
- int my_showstr (char const *)
- char ** split_string (char const *, char const *)
- int my_show_word_array (char *const *)
- char ** my_str_to_word_array (char const *str, char spliter)
- void put_buffer (buffer_t *buff, int fd)
- int my_printf (char const *format,...)
- void manage_specifiers (char c, va_list args, buffer_t *buff, format_t *spec)
- void printf_putchar (char c, buffer_t *buff)
- void printf_putstr (char const *str, buffer_t *buff, format_t *flags)
- int printf_putnbr (int n, buffer_t *buff, format_t *flags)
- char * convert_hex (int long)
- char * convert_oct (int long)
- void printf_putfloat (double, buffer_t *, format_t *)
- char * printf_pointer (void *adress)
- void printf_scientific (double nb, buffer_t *global, format_t *flags, int up)
- int number_len (int n)
- void printf_n_spec (int *n, buffer_t *buff)
- void printf_decimal (int nbr, buffer_t *buff, format_t *flags)
- void printf_g_spec (double nb, buffer_t *buffer, format_t *flags, int up)
- void printf_tab (char **tab, buffer_t *buff)
- void print_char (va_list, buffer_t *, format_t *)
- void print_string (va_list, buffer_t *, format_t *)
- void print_int (va_list, buffer_t *, format_t *)
- void print_mudulo (va_list, buffer_t *, format_t *)
- void print_float (va_list, buffer_t *, format_t *)
- void print_pointer (va_list args, buffer_t *buff, format_t *spec)
- void print_puthexmin (va_list args, buffer_t *buff, format_t *spec)
- void print_puthexmaj (va_list args, buffer_t *buff, format_t *spec)
- void print_octal (va_list args, buffer_t *buff, format_t *spec)
- void print_scientific (va_list args, buffer_t *buff, format_t *spec)
- void print_capscientific (va_list args, buffer_t *buff, format_t *spec)
- void print_nspec (va_list args, buffer_t *buff, format_t *spec)
- void parse_format (char const *, int *, format_t *, va_list)
- void print_decimal (va_list args, buffer_t *buff, format_t *spec)
- void print_g_low (va_list args, buffer_t *buff, format_t *spec)
- void print_g_up (va_list args, buffer_t *buff, format_t *spec)

4.3.1 Typedef Documentation

4.3.1.1 buffer_t

```
typedef struct buffer_s buffer_t
```

4.3.1.2 format_t

```
typedef struct format_s format_t
```

4.3.1.3 printf_t

```
typedef struct printf_s printf_t
```

4.3.2 Function Documentation

4.3.2.1 convert_hex()

```
char * convert_hex (
            int long )
```

4.3.2.2 convert_oct()

```
char * convert_oct (
            int long )
```

4.3.2.3 manage_specifiers()

```
void manage_specifiers (
            char c,
            va_list args,
            buffer_t * buff,
            format_t * spec)
```

4.3.2.4 my_compute_power_rec()

```
int my_compute_power_rec (
            int ,
            int )
```

4.3.2.5 my_compute_square_root()

```
int my_compute_square_root (
            int )
```

4.3.2.6 my_find_prime_sup()

```
int my_find_prime_sup (
            int )
```

4.3.2.7 my_getnbr()

```
int my_getnbr (
            char const * )
```

4.3.2.8 my_is_prime()

```
int my_is_prime (
            int )
```

4.3.2.9 my_isalnum()

```
int my_isalnum (
            char )
```

4.3.2.10 my_isneg()

```
int my_isneg (
            int )
```

4.3.2.11 my_printf()

```
int my_printf (
            char const * format,
             ...)
```

4.3.2.12 my_put_nbr()

```
int my_put_nbr (
            int )
```

4.3.2.13 my_putchar()

```
void my_putchar (
            char )
```

4.3.2.14 my_puterr()

```
void my_puterr (
            char const * )
```

4.3.2.15 my_putstr()

```
void my_putstr (
            char const * )
```

4.3.2.16 my_revstr()

```
char * my_revstr (
            char * )
```

4.3.2.17 my_show_word_array()

```
int my_show_word_array (
            char *const * )
```

4.3.2.18 my_showmem()

```
int my_showmem (
            char const * ,
            int )
```

4.3.2.19 my_showstr()

```
int my_showstr (
            char const * )
```

4.3.2.20 my_sort_int_array()

```
void my_sort_int_array (
            int * ,
            int )
```

4.3.2.21 my_str_isalpha()

```
int my_str_isalpha (
            char const * )
```

4.3.2.22 my_str_islower()

```
int my_str_islower (
            char const * )
```

4.3.2.23 my_str_isnum()

```
int my_str_isnum (
            char const * )
```

4.3.2.24 my_str_isprintable()

```
int my_str_isprintable (
            char const * )
```

4.3.2.25 my_str_isupper()

```
int my_str_isupper (
            char const * )
```

4.3.2.26 my_str_to_word_array()

```
char ** my_str_to_word_array (
            char const * str,
            char spliter)
```

4.3.2.27 my_strcapitalize()

```
char * my_strcapitalize (
            char * )
```

4.3.2.28 my_strcat()

```
char * my_strcat (
            char * ,
            char const * )
```

4.3.2.29 my_strcmp()

```
int my_strcmp (
            char const * ,
            char const * )
```

4.3.2.30 my_strcpy()

```
char * my_strcpy (
            char * ,
            char const * )
```

4.3.2.31 my_strdup()

```
char * my_strdup (
            char const * )
```

4.3.2.32 my_strlen()

```
int my_strlen (
            char const * )
```

4.3.2.33 my_strlowcase()

```
char * my_strlowcase (
            char * )
```

4.3.2.34 my_strncat()

```
char * my_strncat (
            char * ,
            char const * ,
            int n)
```

4.3.2.35 my_strncmp()

```
int my_strncmp (
            char const * ,
            char const * ,
            int n)
```

4.3.2.36 my_strncpy()

```
char * my_strncpy (
            char * ,
            char const * ,
            int n)
```

4.3.2.37 my_strstr()

```
char * my_strstr (
            char * ,
            char const * )
```

4.3.2.38 my_strupcase()

```
char * my_strupcase (
            char * )
```

4.3.2.39 my_swap()

```
void my_swap (
            int * ,
            int * )
```

4.3.2.40 number_len()

```
int number_len (
            int n)
```

4.3.2.41 parse_format()

```
void parse_format (
            char const * ,
            int * ,
            format_t * ,
            va_list )
```

4.3.2.42 print_capscientific()

```
void print_capscientific (
            va_list args,
            buffer_t * buff,
            format_t * spec)
```

4.3.2.43 print_char()

```
void print_char (
            va_list ,
            buffer_t * ,
            format_t * )
```

4.3.2.44 print_decimal()

```
void print_decimal (
            va_list args,
            buffer_t * buff,
            format_t * spec)
```

4.3.2.45 print_float()

```
void print_float (
            va_list ,
            buffer_t * ,
            format_t * )
```

4.3.2.46 print_g_low()

```
void print_g_low (
            va_list args,
            buffer_t * buff,
            format_t * spec)
```

4.3.2.47 print_g_up()

```
void print_g_up (
            va_list args,
            buffer_t * buff,
            format_t * spec)
```

4.3.2.48 print_int()

```
void print_int (
            va_list ,
            buffer_t * ,
            format_t * )
```

4.3.2.49 print_mudulo()

```
void print_mudulo (
            va_list ,
            buffer_t * ,
            format_t * )
```

4.3.2.50 print_nspec()

```
void print_nspec (
            va_list args,
            buffer_t * buff,
            format_t * spec)
```

4.3.2.51 print_octal()

```
void print_octal (
            va_list args,
            buffer_t * buff,
            format_t * spec)
```

4.3.2.52 print_pointer()

```
void print_pointer (
            va_list args,
            buffer_t * buff,
            format_t * spec)
```

4.3.2.53 print_puthexmaj()

```
void print_puthexmaj (
            va_list args,
            buffer_t * buff,
            format_t * spec)
```

4.3.2.54 print_puthexmin()

```
void print_puthexmin (
            va_list args,
            buffer_t * buff,
            format_t * spec)
```

4.3.2.55 print_scientific()

```
void print_scientific (
            va_list args,
            buffer_t * buff,
            format_t * spec)
```

4.3.2.56 print_string()

```
void print_string (
            va_list ,
            buffer_t * ,
            format_t * )
```

4.3.2.57 printf_decimal()

```
void printf_decimal (
            int nbr,
            buffer_t * buff,
            format_t * flags)
```

4.3.2.58 printf_g_spec()

```
void printf_g_spec (
            double nb,
            buffer_t * buffer,
            format_t * flags,
            int up)
```

4.3.2.59 printf_n_spec()

```
void printf_n_spec (
            int * n,
            buffer_t * buff)
```

4.3.2.60 printf_pointer()

```
char * printf_pointer (
            void * adress)
```

4.3.2.61 printf_putchar()

```
void printf_putchar (
             char c,
             buffer_t * buff)
```

4.3.2.62 printf_putfloat()

```
void printf_putfloat (
             double ,
             buffer_t * ,
             format_t * )
```

4.3.2.63 printf_putnbr()

```
int printf_putnbr (
             int n,
             buffer_t * buff,
             format_t * flags)
```

4.3.2.64 printf_putstr()

```
void printf_putstr (
             char const * str,
             buffer_t * buff,
             format_t * flags)
```

4.3.2.65 printf_scientific()

```
void printf_scientific (
             double nb,
             buffer_t * global,
             format_t * flags,
             int up)
```

4.3.2.66 printf_tab()

```
void printf_tab (
             char ** tab,
             buffer_t * buff)
```

4.3.2.67 put_buffer()

```
void put_buffer (
             buffer_t * buff,
             int fd)
```

4.3.2.68 split_string()

```
char ** split_string (
            char const * ,
            char const * )
```

4.4 my.h

Go to the documentation of this file.

```
00001 /*
00002 ** EPITECH PROJECT, 2024
00003 ** B-CPE-100-LIL-1-1-cpoolday08-theophile.riffe
00004 ** File description:
00005 ** my.h
00006 */
00007
00008 #ifndef MY_H_
00009     #define MY_H_
00010
00011     #include <stdarg.h>
00012     #include <unistd.h>
00013     #include <limits.h>
00014     #include <stdint.h>
00015     #include <stdlib.h>
00016     #include <fcntl.h>
00017     #include <stdio.h>
00018     #include <sys/stat.h>
00019     #include <sys/sysmacros.h>
00020     #include <time.h>
00021
00022 // Character/Output Functions
00023 void my_putchar(char);
00024 void my_putstr(char const *);
00025 int my_put_nbr(int);
00026 void my_puterr(char const *);
00027
00028 // String Length/Comparison Functions
00029 int my_strlen(char const *);
00030 int my_strcmp(char const *, char const *);
00031 int my_strncmp(char const *, char const *, int n);
00032 int my_isalnum(char);
00033
00034 // String Manipulation Functions
00035 char *my_strcat(char *, char const *);
00036 char *my_strncat(char *, char const *, int n);
00037 char *my_strcpy(char *, char const *);
00038 char *my_strncpy(char *, char const *, int n);
00039 char *my_revstr(char *);
00040 char *my_strstr(char *, char const *);
00041 char *my_strdup(char const *);
00042
00043 // String Case Functions
00044 char *my_strcapitalize(char *);
00045 char *my_strlowcase(char *);
00046 char *my_strupcase(char *);
00047
00048 // String Property Functions
00049 int my_str_isalpha(char const *);
00050 int my_str_islower(char const *);
00051 int my_str_isnum(char const *);
00052 int my_str_isprintable(char const *);
00053 int my_str_isupper(char const *);
00054
00055 // Number-related Functions
00056 int my_getnbr(char const *);
00057 int my_compute_power_rec(int, int);
00058 int my_compute_square_root(int);
00059 int my_find_prime_sup(int);
00060 int my_is_prime(int);
00061 int my_isneg(int);
00062
00063 // Array/Sorting Functions
00064 void my_sort_int_array(int *, int);
00065 void my_swap(int *, int *);
00066
00067 // Miscellaneous Functions
00068 int my_showmem(char const *, int);
00069 int my_showstr(char const *);
```

```
00070
00071 // Double array Functions
00072 char **split_string(char const *, char const *);
00073 int my_show_word_array(char *const *);
00074 char **my_str_to_word_array(char const *str, char spliter);
00075
00076 /* PRINTF */
00077
00078 typedef struct buffer_s {
00079     char buffer[1024];
00080     int pos;
00081     int len;
00082 } buffer_t;
00083
00084 typedef struct format_s {
00085     int flags;
00086     int width;
00087     int precision;
00088     int length;
00089 } format_t;
00090
00091 typedef struct printf_s {
00092     char c;
00093     void (*func)(va_list, buffer_t *, format_t *);
00094 } printf_t;
00095
00096 void put_buffer(buffer_t *buff, int fd);
00097 int my_printf(char const *format, ...);
00098 void manage_specifiers(char c, va_list args, buffer_t *buff, format_t *spec);
00099
00100 void printf_putchar(char c, buffer_t *buff);
00101 void printf_putstr(char const *str, buffer_t *buff, format_t *flags);
00102 int printf_putnbr(int n, buffer_t *buff, format_t *flags);
00103 int my_printf(char const *format, ...);
00104 char *convert_hex(int long);
00105 char *convert_oct(int long);
00106 void printf_putfloat(double, buffer_t *, format_t *);
00107 char *printf_pointer(void *adress);
00108 void printf_scientific(double nb, buffer_t *global, format_t *flags, int up);
00109 int number_len(int n);
00110 void printf_n_spec(int *n, buffer_t *buff);
00111 void printf_decimal(int nbr, buffer_t *buff, format_t *flags);
00112 void printf_g_spec(double nb, buffer_t *buffer, format_t *flags, int up);
00113 void printf_tab(char **tab, buffer_t *buff);
00114
00115 void print_char(va_list, buffer_t *, format_t *);
00116 void print_string(va_list, buffer_t *, format_t *);
00117 void print_int(va_list, buffer_t *, format_t *);
00118 void print_mudulo(va_list, buffer_t *, format_t *);
00119 void print_float(va_list, buffer_t *, format_t *);
00120 void print_pointer(va_list args, buffer_t *buff, format_t *spec);
00121 void print_puthexmin(va_list args, buffer_t *buff, format_t *spec);
00122 void print_puthexmaj(va_list args, buffer_t *buff, format_t *spec);
00123 void print_octal(va_list args, buffer_t *buff, format_t *spec);
00124 void print_scientific(va_list args, buffer_t *buff, format_t *spec);
00125 void print_capscientific(va_list args, buffer_t *buff, format_t *spec);
00126 void print_nspec(va_list args, buffer_t *buff, format_t *spec);
00127 void parse_format(char const *, int *, format_t *, va_list);
00128 void print_decimal(va_list args, buffer_t *buff, format_t *spec);
00129 void print_g_low(va_list args, buffer_t *buff, format_t *spec);
00130 void print_g_up(va_list args, buffer_t *buff, format_t *spec);
00131
00132 #endif /* !MY_H_ */
```

4.5 include/op.h File Reference

```
#include "my.h"
#include <stdbool.h>
#include <ctype.h>
```

Data Structures

- struct op_s
- struct header_s

- struct proc_s
- struct champion_s
- struct args_t
- struct arena_s
- struct flags_t

Macros

- #define MEM_SIZE (6 ∗ 1024)
- #define IDX_MOD 512 /∗ modulo of the index < ∗/
- #define MAX_ARGS_NUMBER 4 /∗ this may not be changed 2^∗IND_SIZE ∗/
- #define COMMENT_CHAR '#'
- #define LABEL_CHAR ':'
- #define DIRECT_CHAR '%'
- #define SEPARATOR_CHAR ','
- #define LABEL_CHARS "abcdefghijklmnopqrstuvwxyz_0123456789"
- #define NAME_CMD_STRING ".name"
- #define COMMENT_CMD_STRING ".comment"
- #define REG_NUMBER 16 /∗ r1 <--> rx ∗/
- #define T_REG 1 /∗ register ∗/
- #define T_DIR 2 /∗ direct (ld #1,r1 put 1 into r1) ∗/
- #define T_IND 4
- #define T_LAB 8 /∗ LABEL ∗/
- #define IND_SIZE 2
- #define DIR_SIZE 4
- #define REG_SIZE DIR_SIZE
- #define PROG_NAME_LENGTH 128
- #define COMMENT_LENGTH 2048
- #define COREWAR_EXEC_MAGIC 0xea83f3
- #define CYCLE_TO_DIE 1536 /∗ number of cycle before beig declared dead ∗/
- #define CYCLE_DELTA 5
- #define NBR_LIVE 40
- #define BIGINT_SIZE 12
- #define MAX_CHAMPIONS 4
- #define MAX_ARGS 4

Typedefs

- typedef char args_type_t
- typedef struct op_s op_t
- typedef struct header_s header_t
- typedef struct proc_s proc_t
- typedef struct champion_s champion_t
- typedef struct arena_s arena_t

Functions

- int check_help (int argc, char **argv)

  Checks if the help flag is provided.
- void display_help (void)

  Displays the help message.
- int parse_flag (arena_t *arena, int argc, char **argv)

  Parses command line flags and initializes arena settings.
- arena_t * init_arena (void)

  Initializes the arena.
- void free_arena (arena_t *arena)

  Frees memory allocated for the arena structure.
- void init_champions (arena_t *arena)

  Initializes champions in the arena.
- unsigned char * init_vm (void)

  Allocates and initializes the virtual machine memory.
- void store_positions (arena_t *arena)

  Stores starting positions for all champions.
- void init_alive_champions (arena_t *arena)

  Initializes alive champions in the arena.
- int get_names (arena_t *arena)

  Extracts the names of all champions in the arena.
- int store_instructions (arena_t *arena)

  Stores each champion's instructions in the VM memory.
- int check_coding_byte (unsigned char instruction)

  Determines whether an instruction includes a coding byte.
- int launch_corewar (arena_t *arena)

  Launches the Corewar game loop.
- int check_instruction (unsigned char byte)

  Checks if a byte corresponds to a valid instruction.
- int check_args (arena_t *arena, int position)

  Validates the arguments of a Corewar instruction.
- unsigned char * extract_instructions (champion_t *champion)

  Extracts the instructions of a champion from its binary file.
- int get_wait_time (int index)
- void reset_array (int *arr)

  Resets a 4-element integer array to -1.
- int get_param (champion_t *champion, arena_t *arena, int arg_type, int position)

  Retrieves a parameter value from memory or registers.
- void create_node (champion_t *champion, int position)

  Creates and appends a new process node to a champion's process list.

4.5.1 Macro Definition Documentation

4.5.1.1 BIGINT_SIZE

```
#define BIGINT_SIZE 12
```

4.5.1.2 COMMENT_CHAR

```
#define COMMENT_CHAR '#'
```

4.5.1.3 COMMENT_CMD_STRING

```
#define COMMENT_CMD_STRING ".comment"
```

4.5.1.4 COMMENT_LENGTH

```
#define COMMENT_LENGTH 2048
```

4.5.1.5 COREWAR_EXEC_MAGIC

```
#define COREWAR_EXEC_MAGIC 0xea83f3
```

4.5.1.6 CYCLE_DELTA

```
#define CYCLE_DELTA 5
```

4.5.1.7 CYCLE_TO_DIE

```
#define CYCLE_TO_DIE 1536 /* number of cycle before beig declared dead */
```

4.5.1.8 DIR_SIZE

```
#define DIR_SIZE 4
```

4.5.1.9 DIRECT_CHAR

```
#define DIRECT_CHAR '%'
```

4.5.1.10 IDX_MOD

```
#define IDX_MOD 512 /* modulo of the index < */
```

4.5.1.11 IND_SIZE

```
#define IND_SIZE 2
```

4.5.1.12 LABEL_CHAR

```
#define LABEL_CHAR ':'
```

4.5.1.13 LABEL_CHARS

```
#define LABEL_CHARS "abcdefghijklmnopqrstuvwxyz_0123456789"
```

4.5.1.14 MAX_ARGS

```
#define MAX_ARGS 4
```

4.5.1.15 MAX_ARGS_NUMBER

```
#define MAX_ARGS_NUMBER 4 /* this may not be changed 2^*IND_SIZE */
```

4.5.1.16 MAX_CHAMPIONS

```
#define MAX_CHAMPIONS 4
```

4.5.1.17 MEM_SIZE

```
#define MEM_SIZE (6 * 1024)
```

4.5.1.18 NAME_CMD_STRING

```
#define NAME_CMD_STRING ".name"
```

4.5.1.19 NBR_LIVE

```
#define NBR_LIVE 40
```

4.5.1.20 PROG_NAME_LENGTH

```
#define PROG_NAME_LENGTH 128
```

4.5.1.21 REG_NUMBER

```
#define REG_NUMBER 16 /* r1 <--> rx */
```

4.5.1.22 REG_SIZE

```
#define REG_SIZE DIR_SIZE
```

4.5.1.23 SEPARATOR_CHAR

```
#define SEPARATOR_CHAR ','
```

4.5.1.24 T_DIR

```
#define T_DIR 2 /* direct (ld #1,r1 put 1 into r1) */
```

4.5.1.25 T_IND

```
#define T_IND 4
```

4.5.1.26 T_LAB

```
#define T_LAB 8 /* LABEL */
```

4.5.1.27 T_REG

```
#define T_REG 1 /* register */
```

4.5.2 Typedef Documentation

4.5.2.1 arena_t

```
typedef struct arena_s arena_t
```

4.5.2.2 args_type_t

```
typedef char args_type_t
```

4.5.2.3 champion_t

```
typedef struct champion_s champion_t
```

4.5.2.4 header_t

```
typedef struct header_s header_t
```

4.5.2.5 op_t

```
typedef struct op_s op_t
```

4.5.2.6 proc_t

```
typedef struct proc_s proc_t
```

4.5.3 Function Documentation

4.5.3.1 check_args()

```
int check_args (
            arena_t * arena,
            int position)
```

Validates the arguments of a Corewar instruction.

This function reads the argument types from the coding byte or defaults to T_DIR if there is no coding byte, and checks them against the expected types from the op_tab.

Parameters

arena	Pointer to the arena structure containing state and memory.
position	The index of the instruction in memory.

Returns

1 if all arguments are valid, 0 otherwise.

4.5.3.2 check_coding_byte()

```
int check_coding_byte (
            unsigned char instruction)
```

Determines whether an instruction includes a coding byte.

Instructions like `live`, `zjmp`, `fork`, and `lfork` do not use a coding byte. This function returns 0 for those, and 1 for others.

Parameters

instruction	The opcode of the instruction.

Returns

1 if it uses a coding byte, 0 otherwise.

4.5.3.3 check_help()

```
int check_help (
            int argc,
            char ** argv)
```

Checks if the help flag is provided.

This function verifies the command line arguments to determine if the -h flag has been passed. If so, it displays the help message.

Parameters

argc	The number of command line arguments.
argv	An array of command line argument strings.

Returns

1 if the help flag is detected and help is displayed, 0 otherwise.

4.5.3.4 check_instruction()

```
int check_instruction (
            unsigned char byte)
```

Checks if a byte corresponds to a valid instruction.

Compares a byte against the `op_tab` to verify if it matches a known instruction code.

Parameters

byte	The byte to check.

Returns

The index in `op_tab` if valid, -1 otherwise.

4.5.3.5 create_node()

```
void create_node (
            champion_t * champion,
            int position)
```

Creates and appends a new process node to a champion's process list.

This function allocates a new process (proc_t), sets its default state (carry = 1, wait = 0, pc = position), and adds it to the end of the champion's linked list of processes.

Parameters

champion	Pointer to the champion structure.
position	Initial program counter (pc) for the new process.

4.5.3.6 display_help()

```
void display_help (
            void )
```

Displays the help message.

This function prints the usage instructions and descriptions to the standard output.

4.5.3.7 extract_instructions()

```
unsigned char * extract_instructions (
            champion_t * champion)
```

Extracts the instructions of a champion from its binary file.

This function reads the instructions of a champion from its binary file, starting after the header and name, and stores them in the `instructions` field of the `champion_t` structure.

Parameters

champion	A pointer to the champion structure containing the filename.

Returns

A pointer to the allocated instructions data, or NULL on failure.

4.5.3.8 free_arena()

```
void free_arena (
            arena_t * arena)
```

Frees memory allocated for the arena structure.

This function frees the virtual machine memory, then iterates over the champions array freeing each allocated champion before finally freeing the arena structure.

Parameters

arena	Pointer to the arena structure.

4.5.3.9 get_names()

```
int get_names (
            arena_t * arena)
```

Extracts the names of all champions in the arena.

This function iterates over all champions in the arena and calls `extract_name` for each of them to populate their `name` fields.

Parameters

arena	A pointer to the arena structure containing the champions.

Returns

0 on success, 1 if any champion's name could not be extracted.

4.5.3.10 get_param()

```
int get_param (
            champion_t * champion,
            arena_t * arena,
            int arg_type,
            int position)
```

Retrieves a parameter value from memory or registers.

This function extracts the value of a parameter depending on its type: register, indirect, or direct. For indirect and direct types, it reconstructs the value from bytes in memory.

Parameters

champion	Pointer to the current champion structure.
arena	Pointer to the arena structure (VM state).
arg_type	The type of the argument (T_REG, T_IND, or T_DIR).
position	The memory position from which to extract the value.

Returns

The integer value corresponding to the argument type.

4.5.3.11 get_wait_time()

```
int get_wait_time (
            int index)
```

4.5.3.12 init_alive_champions()

```
void init_alive_champions (
            arena_t * arena)
```

Initializes alive champions in the arena.

This function sets each champions life to true. It is the global array that is modified when CYCLE_TO_DIE is passed.

Parameters

arena	Pointer to the arena structure.

4.5.3.13 init_arena()

```
arena_t * init_arena (
            void )
```

Initializes the arena.

This function allocates the arena structure, initializes the virtual machine, live signals, champions, and sets a default dump flag.

Returns

Pointer to the initialized arena, or NULL if an error occurred.

4.5.3.14 init_champions()

```
void init_champions (
            arena_t * arena)
```

Initializes champions in the arena.

This function allocates memory for each champion in the corewar arena and initializes them with default values.

Parameters

arena	Pointer to the arena structure.

4.5.3.15 init_vm()

```
unsigned char * init_vm (
            void )
```

Allocates and initializes the virtual machine memory.

This function allocates memory for the virtual machine, initializes all bytes to 0, and appends a null terminator at the end.

Returns

A pointer to the initialized virtual machine memory, or NULL if memory allocation fails.

4.5.3.16 launch_corewar()

```
int launch_corewar (
            arena_t * arena)
```

Launches the Corewar game loop.

Initializes the necessary data and repeatedly processes cycles until only one champion remains. Frees the arena at the end.

Parameters

arena	Pointer to the arena structure.

Returns

0 on successful completion of the game, 84 on error.

4.5.3.17 parse_flag()

```
int parse_flag (
            arena_t * arena,
            int argc,
            char ** argv)
```

Parses command line flags and initializes arena settings.

This function iterates through the command line arguments and calls `check_args` to set champion attributes and dump options. It ensures that at least two champions are provided and returns an error if flag parsing fails.

Parameters

arena	Pointer to the arena structure.
argc	Number of command line arguments.
argv	Array of command line argument strings.

Returns

0 if parsing is successful, or 1 if an error is detected.

4.5.3.18 reset_array()

```
void reset_array (
            int * arr)
```

Resets a 4-element integer array to -1.

Used to clear argument type arrays or other temporary storage.

Parameters

arr	The array to reset.

4.5.3.19 store_instructions()

```
int store_instructions (
            arena_t * arena)
```

Stores each champion's instructions in the VM memory.

Iterates over all champions in the arena, extracting and copying their instructions into the virtual machine's memory using `fill_vm`.

Parameters

arena	Pointer to the arena structure.

Returns

0 on success, 84 if an error occurs during instruction loading.

4.5.3.20 store_positions()

```
void store_positions (
            arena_t * arena)
```

Stores starting positions for all champions.

This function iterates through all champions in the arena. If a champion's position is unset (equal to -1), it computes and assigns a starting position.

Parameters

arena	Pointer to the arena structure.

4.6 op.h

Go to the documentation of this file.

```
00001 /*
00002 ** EPITECH PROJECT, 2025
00003 ** robotfactory
00004 ** File description:
00005 ** op
00006 */
00007
00008 #ifndef _OP_H_
00009     #include "my.h"
00010     #include <stdbool.h>
00011     #include <ctype.h>
00012     #define _OP_H_
00013     #define MEM_SIZE (6 * 1024)
00014     #define IDX_MOD 512 /* modulo of the index < */
00015     #define MAX_ARGS_NUMBER 4 /* this may not be changed 2^*IND_SIZE */
00016     #define COMMENT_CHAR '#'
00017     #define LABEL_CHAR ':'
00018     #define DIRECT_CHAR '%'
00019     #define SEPARATOR_CHAR ','
00020     #define LABEL_CHARS "abcdefghijklmnopqrstuvwxyz_0123456789"
00021     #define NAME_CMD_STRING ".name"
00022     #define COMMENT_CMD_STRING ".comment"
00023     /*
00024     ** regs
00025     */
00026     #define REG_NUMBER 16 /* r1 <--> rx */
00027     #define T_REG 1 /* register */
00028     #define T_DIR 2 /* direct  (ld  #1,r1  put 1 into r1) */
00029     #define T_IND 4
00030     //indirect always relative (ld 1,r1 put what's in
00031     //the address (1+pc) into r1 (4 bytes ))
00032
00033     #define T_LAB 8 /* LABEL */
00034     /*
00035     ** size (in bytes)
00036     */
00037     #define IND_SIZE 2
00038     #define DIR_SIZE 4
00039     #define REG_SIZE DIR_SIZE
00040     /*
00041     ** header
00042     */
00043     #define PROG_NAME_LENGTH 128
00044     #define COMMENT_LENGTH 2048
00045     #define COREWAR_EXEC_MAGIC 0xea83f3
00046     /*
00047     ** live
00048     */
00049     #define CYCLE_TO_DIE 1536 /* number of cycle before beig declared dead */
00050     #define CYCLE_DELTA 5
00051     #define NBR_LIVE 40
00052
00053     #define BIGINT_SIZE 12
00054
00055 /*
00056 **
00057 */
00058 typedef char args_type_t;
00059
00060 typedef struct op_s {
00061     char *mnemonique;
00062     char nbr_args;
00063     args_type_t type[MAX_ARGS_NUMBER];
00064     char code;
00065     int nbr_cycles;
00066     char *comment;
00067 } op_t;
00068
00069 typedef struct header_s {
00070     int magic;
00071     char prog_name[PROG_NAME_LENGTH + 1];
```

```
00072     int prog_size;
00073     char comment[COMMENT_LENGTH + 1];
00074 } header_t;
00075
00076
00080
00081     #define MAX_CHAMPIONS 4
00082     #define MAX_ARGS 4
00083
00084 typedef struct proc_s {
00085     int pc;
00086     int carry;
00087     unsigned int wait;
00088     struct proc_s *next;
00089 } proc_t;
00090
00091 typedef struct champion_s {
00092     int registers[REG_NUMBER];
00093     int id;
00094     int position;
00095     char *name;
00096     char *filename;
00097     int len;
00098     proc_t *procs;
00099 } champion_t;
00100
00101 typedef struct {
00102     int offset;
00103     int arg_index;
00104     int arg_type;
00105     int *arg_type_array;
00106     int nb_args;
00107 } args_t;
00108
00109 typedef struct arena_s {
00110     unsigned char *vm;
00111     bool live_signals[MAX_CHAMPIONS];
00112     bool alive_champions[MAX_CHAMPIONS];
00113     int dump;
00114     champion_t *champions[MAX_CHAMPIONS];
00115     int nb_champions;
00116     args_t *args;
00117     int nb_live_signal;
00118 } arena_t;
00119
00120 typedef struct {
00121     unsigned int dump;
00122     unsigned int n;
00123     unsigned int a;
00124     char *prog_name;
00125 } flags_t;
00126
00128
00129 int check_help(int argc, char **argv);
00130 void display_help(void);
00131
00133
00135
00136 int parse_flag(arena_t *arena, int argc, char **argv);
00137
00139
00141
00142 arena_t *init_arena(void);
00143 void free_arena(arena_t *arena);
00144 void init_champions(arena_t *arena);
00145 unsigned char *init_vm(void);
00146 void store_positions(arena_t *arena);
00147 void init_alive_champions(arena_t *arena);
00148 int get_names(arena_t *arena);
00149 int store_instructions(arena_t *arena);
00150
00152
00154
00155 int check_coding_byte(unsigned char instruction);
00156 int launch_corewar(arena_t *arena);
00157 int check_instruction(unsigned char byte);
00158 int check_args(arena_t *arena, int position);
00159 unsigned char *extract_instructions(champion_t *champion);
00160 int get_wait_time(int index);
00161
00163
00165
00166 void reset_array(int *arr);
00167 int check_instruction(unsigned char byte);
00168 int check_coding_byte(unsigned char instruction);
00169 int get_param(champion_t *champion, arena_t *arena,
00170     int arg_type, int position);
```

```
00171 void create_node(champion_t *champion, int position);
00172
00174
00175 #endif
```

4.7 include/tab.h File Reference

```
#include "op.h"
```

Variables

- const op_t op_tab [ ]

  Global operation table defining all Corewar instructions.

4.7.1 Variable Documentation

4.7.1.1 op_tab

```
const op_t op_tab[]  [extern]
```

Global operation table defining all Corewar instructions.

Each instruction is defined by:

- mnemonic (e.g., "ld", "st")
- number of arguments (1 to 3)
- accepted argument types for each position (bitwise OR of T_REG, T_DIR, T_IND)
- opcode (unique ID for instruction)
- number of cycles required to execute
- description of the instruction

4.8 tab.h

Go to the documentation of this file.

```
00001 /*
00002 ** EPITECH PROJECT, 2025
00003 ** robotfactory
00004 ** File description:
00005 ** tab
00006 */
00007
00008 #ifndef TAB_H_
00009     #define TAB_H_
00010     #include "op.h"
00011
00012 /*
00013 ** op_tab
00014 */
00015 extern const op_t op_tab[];
00016
00017 #endif /* !TAB_H_ */
```

4.9 src/check_args.c File Reference

Functions to validate instructions and their arguments.

```
#include "op.h"
#include "tab.h"
```

Functions

- static bool use_short_dir (int opcode)

  Determines if an instruction uses a short direct value.
- static int calculate_arg_size (const op_t *op, int arg_type)

  Calculates the size in bytes for a given argument type.
- static int check_type (const op_t *op, arena_t *arena, int expected)

  Validates a single argument type and updates offset.
- static int extract_arg_type (arena_t *arena, int coding_byte)

  Extracts the argument type from the coding byte.
- static int handle_coding_byte (const op_t *op, arena_t *arena, int coding_byte)

  Handles argument validation when a coding byte is present.
- static int handle_no_coding_byte (const op_t *op, arena_t *arena)

  Handles validation of argument when no coding byte exists.
- static int check_args_given_codingbyte (arena_t *arena, int position)

  Dispatcher for argument checking based on coding byte presence.
- int check_args (arena_t *arena, int position)

  Validates the arguments of a Corewar instruction.

4.9.1 Detailed Description

Functions to validate instructions and their arguments.

This file contains functions to validate the arguments of instructions in the Corewar virtual machine. It ensures that the arguments match the expected types and calculates their sizes.

4.9.2 Function Documentation

4.9.2.1 calculate_arg_size()

```
static int calculate_arg_size (
            const op_t * op,
            int arg_type)  [static]
```

Calculates the size in bytes for a given argument type.

This function determines the number of bytes to read for a given argument type depending on the instruction's specification.

Parameters

op	A pointer to the op_t structure describing the instruction.
arg_type	The type of the argument (T_REG, T_DIR, T_IND).

Returns

The size in bytes of the argument, or 0 if the type is invalid.

4.9.2.2 check_args()

```
int check_args (
            arena_t * arena,
            int position)
```

Validates the arguments of a Corewar instruction.

This function reads the argument types from the coding byte or defaults to T_DIR if there is no coding byte, and checks them against the expected types from the op_tab.

Parameters

arena	Pointer to the arena structure containing state and memory.
position	The index of the instruction in memory.

Returns

1 if all arguments are valid, 0 otherwise.

4.9.2.3 check_args_given_codingbyte()

```
static int check_args_given_codingbyte (
            arena_t * arena,
            int position)  [static]
```

Dispatcher for argument checking based on coding byte presence.

Selects the appropriate handler based on whether the instruction uses a coding byte, and validates the current argument.

Parameters

arena	Pointer to the arena structure.
position	Current memory index of the instruction.

Returns

1 if the argument is valid, 0 otherwise.

4.9.2.4 check_type()

```
static int check_type (
            const op_t * op,
            arena_t * arena,
            int expected)  [static]
```

Validates a single argument type and updates offset.

This function checks whether the given argument type is allowed for the current instruction. If valid, it updates the offset and stores the size in the argument size array.

Parameters

op	Pointer to the instruction definition in op_tab.
arena	Pointer to the arena containing the args and memory.
expected	Expected type(s) for the current argument.

Returns

1 if the type is valid, 0 otherwise.

4.9.2.5 extract_arg_type()

```
static int extract_arg_type (
            arena_t * arena,
            int coding_byte)  [static]
```

Extracts the argument type from the coding byte.

The coding byte contains the types of the arguments. This function extracts the type of the current argument based on its index.

Parameters

arena	Pointer to the arena structure.
coding_byte	The coding byte to extract from.

Returns

Extracted argument type as T_REG, T_DIR, or T_IND.

4.9.2.6 handle_coding_byte()

```
static int handle_coding_byte (
            const op_t * op,
            arena_t * arena,
            int coding_byte)  [static]
```

Handles argument validation when a coding byte is present.

Uses the coding byte to extract and validate the current argument type.

Parameters

op	Pointer to the instruction structure.
arena	Pointer to the arena structure.
coding_byte	The coding byte read from memory.

Returns

1 if the argument is valid, 0 otherwise.

4.9.2.7 handle_no_coding_byte()

```
static int handle_no_coding_byte (
            const op_t * op,
            arena_t * arena)  [static]
```

Handles validation of argument when no coding byte exists.

If the instruction doesn't have a coding byte, its only argument is assumed to be a direct value (T_DIR).

Parameters

op	Pointer to the instruction definition.
arena	Pointer to the arena structure.

Returns

1 if the argument is valid, 0 otherwise.

4.9.2.8 use_short_dir()

```
static bool use_short_dir (
            int opcode)  [static]
```

Determines if an instruction uses a short direct value.

Some instructions in Corewar use a short direct value (2 bytes) instead of the standard direct value (4 bytes). This function checks if the given opcode corresponds to one of these instructions.

Parameters

opcode	The opcode of the instruction.

Returns

true if the instruction uses a short direct value, false otherwise.

4.10 src/corewar.c File Reference

Corewar game logic implementation.

```
#include "op.h"
#include "instructions.h"
#include "tab.h"
```

Functions

- static int is_one_standing (arena_t *arena)

  Checks if only one champion is still alive.
- static void check_if_alive (arena_t *arena)

  Updates the alive status of champions based on live signals.
- static void check_cycle_to_die (arena_t *arena, int cycle_to_die)

  Checks if the cycle-to-die threshold has been reached.
- static void initialize_execution (arena_t *arena, champion_t *champion, int *first_loop, proc_t *curr_proc)

  Initializes and executes the current instruction for a process.
- static void execute_instructions (arena_t *arena, champion_t *champion, int *first_loop, proc_t *curr_proc)

  Executes instructions for a given champion's process.
- static void browse_champs (arena_t *arena, proc_t *tmp_proc, int *first_loop)

  Iterates through all champions and executes their processes.
- static int check_winning (arena_t *arena)
- int launch_corewar (arena_t *arena)

  Launches the Corewar game loop.

4.10.1 Detailed Description

Corewar game logic implementation.

This file contains the main functions for managing the Corewar game, including checking champion statuses, executing instructions, and determining the winner.

4.10.2 Function Documentation

4.10.2.1 browse_champs()

```
static void browse_champs (
            arena_t * arena,
            proc_t * tmp_proc,
            int * first_loop)  [static]
```

Iterates through all champions and executes their processes.

For each champion, traverses its list of processes and executes the current instruction if applicable.

Parameters

arena	Pointer to the arena structure.
tmp_proc	Temporary pointer to a process (unused here).
first_loop	Pointer to the first loop counter.

4.10.2.2 check_cycle_to_die()

```
static void check_cycle_to_die (
            arena_t * arena,
            int cycle_to_die)  [static]
```

Checks if the cycle-to-die threshold has been reached.

If the number of cycles is a multiple of CYCLE_TO_DIE, then the function checks which champions are still alive.

Parameters

arena	Pointer to the arena structure.
cycle_to_die	Current cycle count to check against CYCLE_TO_DIE.

4.10.2.3 check_if_alive()

```
static void check_if_alive (
            arena_t * arena)  [static]
```

Updates the alive status of champions based on live signals.

Champions that have not sent a live signal are marked as dead and a message is printed indicating their death.

Parameters

arena	Pointer to the arena structure.

4.10.2.4 check_winning()

```
static int check_winning (
            arena_t * arena)  [static]
```

4.10.2.5 execute_instructions()

```
static void execute_instructions (
            arena_t * arena,
            champion_t * champion,
            int * first_loop,
            proc_t * curr_proc)  [static]
```

Executes instructions for a given champion's process.

Verifies the instruction and its arguments. If the process has a wait time, it is decremented. If the process is ready, the instruction is executed.

Parameters

arena	Pointer to the arena structure.
champion	Pointer to the champion structure.
first_loop	Pointer to the first loop counter.
curr_proc	Pointer to the current process.

4.10.2.6 initialize_execution()

```
static void initialize_execution (
            arena_t * arena,
            champion_t * champion,
            int * first_loop,
            proc_t * curr_proc)  [static]
```

Initializes and executes the current instruction for a process.

Checks the current instruction in memory and executes it if valid. If the instruction is not valid, the program counter is simply advanced. Handles waiting cycles and instruction execution.

Parameters

arena	Pointer to the arena structure.
champion	Pointer to the champion structure.
first_loop	Pointer to the first loop counter.
curr_proc	Pointer to the current process.

4.10.2.7 is_one_standing()

```
static int is_one_standing (
            arena_t * arena)  [static]
```

Checks if only one champion is still alive.

Iterates through the alive_champions array and counts the number of champions still marked as alive. Returns 1 if one or none are alive, otherwise returns 0.

Parameters

arena	Pointer to the arena structure.

Returns

1 if one or no champions are alive, 0 otherwise.

4.10.2.8 launch_corewar()

```
int launch_corewar (
            arena_t * arena)
```

Launches the Corewar game loop.

Initializes the necessary data and repeatedly processes cycles until only one champion remains. Frees the arena at the end.

Parameters

arena	Pointer to the arena structure.

Returns

0 on successful completion of the game, 84 on error.

4.11 src/flags.c File Reference

Handles flags for the corewar arena.

```
#include "op.h"
```

Functions

- static int check_champion (arena_t *arena, int *index, char *arg)

  Checks if the provided file is a valid champion.
- static int handle_champion_flags (arena_t *arena, char **args, int *index, int *i)

  Handles flags provided before a champion's binary in the arguments.
- static int check_arguments (arena_t *arena, char **args, int *index, int *i)

  Checks if the current argument is a valid champion or a flag.
- int parse_flag (arena_t *arena, int argc, char **argv)

  Parses command line flags and initializes arena settings.

4.11.1 Detailed Description

Handles flags for the corewar arena.

This file contains functions to parse the command line flags for the corewar arena, including champion files and numeric options.

4.11.2 Function Documentation

4.11.2.1 check_arguments()

```
static int check_arguments (
            arena_t * arena,
            char ** args,
            int * index,
            int * i)  [static]
```

Checks if the current argument is a valid champion or a flag.

This function first checks whether the current argument is a valid champion file. If not, it tries to handle it as a flag using handle_champion_flags.

Parameters

arena	Pointer to the arena structure.
args	Command-line arguments.
index	Pointer to the current champion index.
i	Pointer to the current argument index.

Returns

0 if argument was processed, 1 on error.

4.11.2.2 check_champion()

```
static int check_champion (
            arena_t * arena,
            int * index,
            char * arg)  [static]
```

Checks if the provided file is a valid champion.

This function attempts to open the file specified by `arg`. If the file exists and its extension matches the expected champion extension ("cor"), the champion's filename and name are set and the champion index is incremented.

Parameters

arena	Pointer to the arena structure.
index	Pointer to the current champion index.
arg	Path to the potential champion file.

Returns

1 if a valid champion is found, 0 otherwise.

4.11.2.3 handle_champion_flags()

```
static int handle_champion_flags (
            arena_t * arena,
            char ** args,
            int * index,
            int * i)  [static]
```

Handles flags provided before a champion's binary in the arguments.

This function interprets and assigns values for -a (position), -n (ID), and -d (dump cycle) flags, updating the current champion or arena accordingly.

Parameters

arena	Pointer to the arena structure.
args	Command-line arguments.
index	Pointer to the current champion index.
i	Pointer to the current argument index.

Returns

0 on success, 1 on invalid flag or number.

4.11.2.4 parse_flag()

```
int parse_flag (
            arena_t * arena,
            int argc,
            char ** argv)
```

Parses command line flags and initializes arena settings.

This function iterates through the command line arguments and calls `check_args` to set champion attributes and dump options. It ensures that at least two champions are provided and returns an error if flag parsing fails.

Parameters

arena	Pointer to the arena structure.
argc	Number of command line arguments.
argv	Array of command line argument strings.

Returns

0 if parsing is successful, or 1 if an error is detected.

4.12 src/free.c File Reference

Frees allocated memory for flags and arena structures.

```
#include "op.h"
```

Functions

- void free_flags (flags_t *flags)

  Frees memory allocated for the flags structure.
- void free_arena (arena_t *arena)

  Frees memory allocated for the arena structure.

4.12.1 Detailed Description

Frees allocated memory for flags and arena structures.

This file contains functions to free the memory allocated for the corewar flags and arena.

4.12.2 Function Documentation

4.12.2.1 free_arena()

```
void free_arena (
            arena_t * arena)
```

Frees memory allocated for the arena structure.

This function frees the virtual machine memory, then iterates over the champions array freeing each allocated champion before finally freeing the arena structure.

Parameters

arena	Pointer to the arena structure.

4.12.2.2 free_flags()

```
void free_flags (
            flags_t * flags)
```

Frees memory allocated for the flags structure.

This function frees the string holding the program's name and then frees the flags structure.

Parameters

flags	Pointer to the flags structure.

4.13 src/help.c File Reference

Displays help information for the program.

```
#include "../include/op.h"
```

Functions

- void display_help (void)

  Displays the help message.
- int check_help (int argc, char **argv)

  Checks if the help flag is provided.

4.13.1 Detailed Description

Displays help information for the program.

This file contains functions that display the usage instructions and help messages when the -h flag is passed as a command line argument.

4.13.2 Function Documentation

4.13.2.1 check_help()

```
int check_help (
            int argc,
            char ** argv)
```

Checks if the help flag is provided.

This function verifies the command line arguments to determine if the -h flag has been passed. If so, it displays the help message.

Parameters

argc	The number of command line arguments.
argv	An array of command line argument strings.

Returns

1 if the help flag is detected and help is displayed, 0 otherwise.

4.13.2.2 display_help()

```
void display_help (
            void )
```

Displays the help message.

This function prints the usage instructions and descriptions to the standard output.

4.14 src/init/init_arena.c File Reference

Initializes the corewar arena.

```
#include "op.h"
```

Functions

- void init_alive_champions (arena_t *arena)
  Initializes alive champions in the arena.
- static void init_live_signals (arena_t *arena)
  Initializes live signals in the arena.
- static int check_initialization (arena_t *arena)
  Checks that the arena components are properly initialized.
- arena_t * init_arena (void)
  Initializes the arena.

4.14.1 Detailed Description

Initializes the corewar arena.

This file contains functions to initialize the arena, including the virtual machine, live signals, and champions. It is part of the EPITECH PROJECT, 2025.

4.14.2 Function Documentation

4.14.2.1 check_initialization()

```
static int check_initialization (
            arena_t * arena)  [static]
```

Checks that the arena components are properly initialized.

This function verifies that the virtual machine is allocated and filled, and that the champions array is properly populated.

Parameters

arena	Pointer to the arena structure.

Returns

0 if initialization is valid, 1 if an error occurred.

4.14.2.2 init_alive_champions()

```
void init_alive_champions (
            arena_t * arena)
```

Initializes alive champions in the arena.

This function sets each champions life to true. It is the global array that is modified when CYCLE_TO_DIE is passed.

Parameters

arena	Pointer to the arena structure.

4.14.2.3 init_arena()

```
arena_t * init_arena (
            void )
```

Initializes the arena.

This function allocates the arena structure, initializes the virtual machine, live signals, champions, and sets a default dump flag.

Returns

Pointer to the initialized arena, or NULL if an error occurred.

4.14.2.4 init_live_signals()

```
static void init_live_signals (
            arena_t * arena)  [static]
```

Initializes live signals in the arena.

This function sets the first four live signals to false. It is modified when champions send the 'live' instructions.

Parameters

arena	Pointer to the arena structure.

4.15 src/init/init_champions.c File Reference

Initializes champions for the corewar arena.

```
#include "op.h"
```

Functions

- static void fill_champion (champion_t *champion, int id)

  Fills the champion structure with default values.
- void init_champions (arena_t *arena)

  Initializes champions in the arena.

4.15.1 Detailed Description

Initializes champions for the corewar arena.

This file contains functions to allocate and initialize champion structures in the corewar arena. Each champion is assigned an identifier and default values upon initialization.

4.15.2 Function Documentation

4.15.2.1 fill_champion()

```
static void fill_champion (
            champion_t * champion,
            int id)  [static]
```

Fills the champion structure with default values.

This function initializes the registers to 0, sets the champion as alive, assigns an identifier, and initializes other attributes.

Parameters

champion	Double pointer to the champion structure.
id	The identifier for the champion.

4.15.2.2 init_champions()

```
void init_champions (
            arena_t * arena)
```

Initializes champions in the arena.

This function allocates memory for each champion in the corewar arena and initializes them with default values.

Parameters

arena	Pointer to the arena structure.

4.16 src/init/init_positions.c File Reference

Computes and assigns starting positions for champions.

```
#include "op.h"
```

Functions

- static void compute_position (arena_t *arena, int i)

  Computes and assigns the starting position for a champion.
- void store_positions (arena_t *arena)

  Stores starting positions for all champions.

4.16.1 Detailed Description

Computes and assigns starting positions for champions.

This file contains functions for computing and storing starting positions for all champions in the arena, ensuring they are evenly distributed.

4.16.2 Function Documentation

4.16.2.1 compute_position()

```
static void compute_position (
            arena_t * arena,
            int i)  [static]
```

Computes and assigns the starting position for a champion.

This function calculates the starting position for the champion at index `i` by dividing the arena's memory evenly among all champions.

Parameters

arena	Pointer to the arena structure.
i	Index of the champion.

4.16.2.2 store_positions()

```
void store_positions (
            arena_t * arena)
```

Stores starting positions for all champions.

This function iterates through all champions in the arena. If a champion's position is unset (equal to -1), it computes and assigns a starting position.

Parameters

arena	Pointer to the arena structure.

4.17 src/init/init_vm.c File Reference

Initializes the virtual machine memory.

```
#include "op.h"
```

Functions

- unsigned char * init_vm (void)

  Allocates and initializes the virtual machine memory.

4.17.1 Detailed Description

Initializes the virtual machine memory.

This file contains the implementation of the function to allocate and initialize the virtual machine for the corewar arena.

4.17.2 Function Documentation

4.17.2.1 init_vm()

```
unsigned char * init_vm (
            void )
```

Allocates and initializes the virtual machine memory.

This function allocates memory for the virtual machine, initializes all bytes to 0, and appends a null terminator at the end.

Returns

A pointer to the initialized virtual machine memory, or NULL if memory allocation fails.

4.18 src/instructions/add.c File Reference

Implements the `add` instruction for the Corewar VM.

```
#include "op.h"
```

Functions

- void execute_add (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *curr_proc)
  Executes the `add` instruction.

4.18.1 Detailed Description

Implements the `add` instruction for the Corewar VM.

This file contains the function that executes the `add` operation, which sums the contents of two registers and stores the result in a third register. It also sets the carry flag based on the result.

4.18.2 Function Documentation

4.18.2.1 execute_add()

```
void execute_add (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the `add` instruction.

Reads three register numbers from memory, verifies their validity, adds the values from the first two registers, and stores the result in the third register. Updates the carry flag to 1 if the result is 0, or to 0 otherwise.

Parameters

arena	Pointer to the arena structure containing VM memory.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the current process executing the instruction.

4.19 src/instructions/aff.c File Reference

Implements the `aff` instruction for the Corewar VM.

```
#include "op.h"
```

Functions

- void execute_aff (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *curr_proc)
  Executes the `aff` instruction.

4.19.1 Detailed Description

Implements the `aff` instruction for the Corewar VM.

This instruction prints the value of a register as an ASCII character. It reads a register number from memory and prints the value in that register modulo 256 as a character.

4.19.2 Function Documentation

4.19.2.1 execute_aff()

```
void execute_aff (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the `aff` instruction.

This function fetches a register number from memory and prints the content of the register as a character (value % 256). If the register number is invalid, the instruction is ignored.

Parameters

arena	Pointer to the arena structure containing the VM memory.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the current process executing the instruction.

4.20 src/instructions/and.c File Reference

Implements the `and` instruction for the Corewar VM.

```
#include "op.h"
```

Functions

- void execute_and (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *curr_proc)
  Executes the `and` instruction.

4.20.1 Detailed Description

Implements the `and` instruction for the Corewar VM.

This instruction performs a bitwise AND operation between two operands and stores the result in a register. It also updates the carry flag based on whether the result is zero.

4.20.2 Function Documentation

4.20.2.1 execute_and()

```
void execute_and (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the `and` instruction.

This function fetches two parameters from memory based on their types, performs a bitwise AND operation, and stores the result in the target register. It also updates the carry flag depending on whether the result is zero.

Parameters

arena	Pointer to the arena containing VM state and memory.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the process executing the instruction.

4.21 src/instructions/fork.c File Reference

Stub implementation of the `fork` instruction for Corewar.

```
#include "op.h"
```

Functions

- void execute_fork (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *curr_proc)
  Executes the `fork` instruction.

4.21.1 Detailed Description

Stub implementation of the `fork` instruction for Corewar.

This file defines the `fork` instruction handler, which is intended to duplicate a process at a specified address. This current implementation is a placeholder and does not perform any operations.

4.21.2 Function Documentation

4.21.2.1 execute_fork()

```
void execute_fork (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the `fork` instruction.

In Corewar, `fork` creates a new process at a position defined relative to the current one. This function is currently a stub and does not yet implement the expected behavior.

Parameters

arena	Pointer to the arena containing VM state and memory.
champion	Pointer to the champion executing the instruction.
proc	Pointer to the process executing the instruction.

4.22 src/instructions/ld.c File Reference

Implements the `ld` instruction in the Corewar VM.

```
#include "op.h"
```

Functions

- void execute_ld (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *curr_proc)

  Executes the `ld` (load) instruction.

4.22.1 Detailed Description

Implements the `ld` instruction in the Corewar VM.

The `ld` instruction loads a value into a register. This file contains its execution logic, including memory handling and carry flag update.

4.22.2 Function Documentation

4.22.2.1 execute_ld()

```
void execute_ld (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the `ld` (load) instruction.

The `ld` instruction takes a direct or indirect value and loads it into a register. If the loaded value is zero, the carry flag is set.

Parameters

arena	Pointer to the arena structure.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the process executing the instruction.

4.23 src/instructions/ldi.c File Reference

Implements the `ldi` instruction in the Corewar virtual machine.

```
#include "op.h"
```

Functions

- void execute_ldi (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *curr_proc)
  Executes the `ldi` (load index) instruction.

4.23.1 Detailed Description

Implements the `ldi` instruction in the Corewar virtual machine.

The `ldi` instruction loads a value indirectly into a register using the sum of two parameters as an address offset.

4.23.2 Function Documentation

4.23.2.1 execute_ldi()

```
void execute_ldi (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the `ldi` (load index) instruction.

This function loads data from memory into a register. The memory address is computed by summing the first two parameters and applying the IDX_MOD restriction. The result is stored in the register defined by the third parameter.

Parameters

arena	Pointer to the arena structure containing VM state.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the current process executing the instruction.

4.24 src/instructions/lfork.c File Reference

Implements the `lfork` instruction in the Corewar virtual machine.

```
#include "op.h"
```

Functions

- void execute_lfork (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *curr_↩ proc)

  Executes the `lfork` (long fork) instruction.

4.24.1 Detailed Description

Implements the `lfork` instruction in the Corewar virtual machine.

The `lfork` (long fork) instruction is used to create a new process at a specific offset from the current process's PC, without applying the IDX_MOD. This allows for larger jumps in memory.

4.24.2 Function Documentation

4.24.2.1 execute_lfork()

```
void execute_lfork (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the `lfork` (long fork) instruction.

This instruction should duplicate the current process and place the new process at a PC equal to the current PC plus a given direct value. Unlike `fork`, `lfork` does not apply the IDX_MOD restriction.

Parameters

arena	Pointer to the arena structure containing VM state.
champion	Pointer to the champion owning the process.
proc	Pointer to the current process executing the instruction.

4.25 src/instructions/live.c File Reference

Implements the `live` instruction for the Corewar virtual machine.

```
#include "op.h"
```

Functions

- int id_to_index (int id, arena_t *arena)

  Retrieves the index of a champion by its ID.
- void execute_live (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *curr_proc)

  Executes the `live` instruction.

4.25.1 Detailed Description

Implements the `live` instruction for the Corewar virtual machine.

The `live` instruction is used by a champion to indicate that it is still alive. It helps the virtual machine determine which champions are active and which can be eliminated.

4.25.2 Function Documentation

4.25.2.1 execute_live()

```
void execute_live (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the `live` instruction.

This instruction announces that the current champion is alive. It reads the champion ID from the memory following the program counter and marks the corresponding champion as live.

Parameters

arena	Pointer to the arena structure.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the process executing the instruction.

4.25.2.2 id_to_index()

```
int id_to_index (
            int id,
            arena_t * arena)
```

Retrieves the index of a champion by its ID.

Searches the list of champions in the arena to find the one with the given ID.

Parameters

id	The ID of the champion to search for.
arena	Pointer to the arena structure.

Returns

The index of the champion in the arena's champion list, or -1 if not found.

4.26 src/instructions/lld.c File Reference

```
#include "op.h"
```

Functions

- void execute_lld (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *curr_proc)
  Executes the 'lld' instruction.

4.26.1 Function Documentation

4.26.1.1 execute_lld()

```
void execute_lld (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the 'lld' instruction.

Loads a value (direct or indirect) into a register of the champion and updates the carry flag of the current process accordingly.

Parameters

arena	Pointer to the arena structure containing the virtual machine state.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the current process executing the instruction.

4.27 src/instructions/lldi.c File Reference

```
#include "op.h"
```

Functions

- void execute_lldi (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *curr_proc)
  Executes the 'lldi' instruction.

4.27.1 Function Documentation

4.27.1.1 execute_lldi()

```
void execute_lldi (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the 'lldi' instruction.

Loads a value from memory calculated by adding two parameters, then stores the result in a register and updates the carry flag.

Parameters

arena	Pointer to the arena structure containing the VM state.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the current process executing the instruction.

4.28 src/instructions/or.c File Reference

```
#include "op.h"
```

Functions

- void execute_or (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *curr_proc)
  Executes the 'or' instruction.

4.28.1 Function Documentation

4.28.1.1 execute_or()

```
void execute_or (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the 'or' instruction.

Performs a bitwise OR between two parameters and stores the result in a specified register.

Parameters

arena	Pointer to the arena structure containing the VM state.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the current process executing the instruction.

4.29 src/instructions/st.c File Reference

```
#include "op.h"
```

Functions

- void execute_st (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *curr_proc)
  Executes the 'st' instruction.

4.29.1 Function Documentation

4.29.1.1 execute_st()

```
void execute_st (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the 'st' instruction.

Stores the value of a register either into another register or into the arena's memory at a calculated address.

Parameters

arena	Pointer to the arena structure containing the VM state.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the current process executing the instruction.

4.30 src/instructions/sti.c File Reference

```
#include "op.h"
```

Functions

- void execute_sti (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *curr_proc)
  Executes the 'sti' instruction.

4.30.1 Function Documentation

4.30.1.1 execute_sti()

```
void execute_sti (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the 'sti' instruction.

Stores the value of a register at an address computed from the sum of two parameters relative to the current process's program counter.

Parameters

arena	Pointer to the arena structure containing the VM state.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the current process executing the instruction.

4.31 src/instructions/sub.c File Reference

```
#include "op.h"
```

Functions

- void execute_sub (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *curr_proc)
  Executes the 'sub' instruction.

4.31.1 Function Documentation

4.31.1.1 execute_sub()

```
void execute_sub (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the 'sub' instruction.

Subtracts the value of the second register from the first register and stores the result in a third register. Updates the carry flag.

Parameters

arena	Pointer to the arena structure containing the VM state.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the current process executing the instruction.

4.32 src/instructions/xor.c File Reference

```
#include "op.h"
```

Functions

- void execute_xor (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *curr_proc)
  Executes the 'xor' instruction.

4.32.1 Function Documentation

4.32.1.1 execute_xor()

```
void execute_xor (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the 'xor' instruction.

Performs a bitwise XOR operation between two parameters and stores the result in a register.

Parameters

arena	Pointer to the arena structure containing the VM state.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the current process executing the instruction.

4.33 src/instructions/zjmp.c File Reference

```
#include "op.h"
```

Functions

- void execute_zjmp (arena_t *arena, __attribute_maybe_unused__ champion_t *champion, proc_t *curr_↩ proc)

  Executes the 'zjmp' instruction.

4.33.1 Function Documentation

4.33.1.1 execute_zjmp()

```
void execute_zjmp (
            arena_t * arena,
            __attribute_maybe_unused__ champion_t * champion,
            proc_t * curr_proc)
```

Executes the 'zjmp' instruction.

Jumps to a new position in the arena if the carry flag is set.

The jump offset is a signed 16-bit value read from the instruction arguments. The actual jump position is calculated modulo IDX_MOD.

Parameters

arena	Pointer to the arena structure containing the VM state.
champion	Pointer to the champion executing the instruction.
curr_proc	Pointer to the current process executing the instruction.

4.34 src/main.c File Reference

Entry point for the program.

```
#include "op.h"
```

Functions

- static int init_main (arena_t *arena, int ac, char **av)

  Launches the corewar simulation.
- int main (int ac, char **av)

  Program entry point.

4.34.1 Detailed Description

Entry point for the program.

This file contains the main function which initializes the arena, checks command line arguments, and runs the corewar simulation.

4.34.2 Function Documentation

4.34.2.1 init_main()

```
static int init_main (
            arena_t * arena,
            int ac,
            char ** av)  [static]
```

Launches the corewar simulation.

This function processes command line flags, initializes champion names, stores starting positions and instructions, and finally frees the arena memory. In case of a flag parsing error, the arena is freed and the function returns an error code.

Parameters

arena	Pointer to the arena structure.
ac	Number of command line arguments.
av	Array of command line argument strings.

Returns

0 on success, or 84 on error.

4.34.2.2 main()

```
int main (
            int ac,
            char ** av)
```

Program entry point.

This function parses command line arguments, checks for help flag, initializes the arena, and runs the corewar simulation. If any step fails, it frees allocated memory and returns an error code.

Parameters

ac	Number of command line arguments.
av	Array of command line argument strings.

Returns

0 on success, or 84 on error.

4.35 src/op_tab.c File Reference

Defines the instruction set for the Corewar virtual machine.

```
#include "tab.h"
```

Variables

- const op_t op_tab [ ]

  Global operation table defining all Corewar instructions.

4.35.1 Detailed Description

Defines the instruction set for the Corewar virtual machine.

This file contains the array of `op_t` structures, each representing a Corewar instruction. Each entry includes the mnemonic, number of arguments, argument types, opcode, cycle cost, and a description.

4.35.2 Variable Documentation

4.35.2.1 op_tab

```
const op_t op_tab[]
```

Initial value:

```
= {
    {"live", 1, {T_DIR}, 1, 10, "alive"},
    {"ld", 2, {T_DIR | T_IND, T_REG}, 2, 5, "load"},
    {"st", 2, {T_REG, T_IND | T_REG}, 3, 5, "store"},
    {"add", 3, {T_REG, T_REG, T_REG}, 4, 10, "addition"},
    {"sub", 3, {T_REG, T_REG, T_REG}, 5, 10, "soustraction"},
    {"and", 3, {T_REG | T_DIR | T_IND, T_REG | T_IND | T_DIR, T_REG}, 6, 6,
        "et (and  r1, r2, r3   r1&r2 -> r3"},
    {"or", 3, {T_REG | T_IND | T_DIR, T_REG | T_IND | T_DIR, T_REG}, 7, 6,
        "ou  (or   r1, r2, r3   r1 | r2 -> r3"},
    {"xor", 3, {T_REG | T_IND | T_DIR, T_REG | T_IND | T_DIR, T_REG}, 8, 6,
        "ou (xor  r1, r2, r3   r1^r2 -> r3"},
    {"zjmp", 1, {T_DIR}, 9, 20, "jump if zero"},
    {"ldi", 3, {T_REG | T_DIR | T_IND, T_DIR | T_REG, T_REG}, 10, 25,
        "load index"},
    {"sti", 3, {T_REG, T_REG | T_DIR | T_IND, T_DIR | T_REG}, 11, 25,
        "store index"},
    {"fork", 1, {T_DIR}, 12, 800, "fork"},
    {"lld", 2, {T_DIR | T_IND, T_REG}, 13, 10, "long load"},
    {"lldi", 3, {T_REG | T_DIR | T_IND, T_DIR | T_REG, T_REG}, 14, 50,
        "long load index"},
    {"lfork", 1, {T_DIR}, 15, 1000, "long fork"},
    {"aff", 1, {T_REG}, 16, 2, "aff"},
    {0, 0, {0}, 0, 0, 0}
}
```

Global operation table defining all Corewar instructions.

Each instruction is defined by:

- mnemonic (e.g., "ld", "st")
- number of arguments (1 to 3)
- accepted argument types for each position (bitwise OR of T_REG, T_DIR, T_IND)
- opcode (unique ID for instruction)
- number of cycles required to execute
- description of the instruction

4.36 src/parse_file.c File Reference

Functions to parse champion files and extract their names and instructions.

```
#include "op.h"
#include <stdio.h>
#include <stdlib.h>
```

Functions

- static unsigned char * get_data_from_champion (FILE *file, unsigned char *buffer, int offset, int len)

  Reads data from a champion file.
- static int extract_name (champion_t *champion)

  Extracts the name of a champion from its binary file.
- int get_names (arena_t *arena)

  Extracts the names of all champions in the arena.
- unsigned char * extract_instructions (champion_t *champion)

  Extracts the instructions of a champion from its binary file.

4.36.1 Detailed Description

Functions to parse champion files and extract their names and instructions.

4.36.2 Function Documentation

4.36.2.1 extract_instructions()

```
unsigned char * extract_instructions (
            champion_t * champion)
```

Extracts the instructions of a champion from its binary file.

This function reads the instructions of a champion from its binary file, starting after the header and name, and stores them in the `instructions` field of the `champion_t` structure.

Parameters

champion	A pointer to the champion structure containing the filename.

Returns

A pointer to the allocated instructions data, or NULL on failure.

4.36.2.2 extract_name()

```
static int extract_name (
            champion_t * champion)  [static]
```

Extracts the name of a champion from its binary file.

This function reads the name of a champion from its binary file and stores it in the `name` field of the `champion_t` structure.

Parameters

champion	A pointer to the champion structure containing the filename.

Returns

0 on success, 1 on failure (e.g., file could not be opened or memory allocation failed).

4.36.2.3 get_data_from_champion()

```
static unsigned char * get_data_from_champion (
            FILE * file,
            unsigned char * buffer,
            int offset,
            int len)  [static]
```

Reads data from a champion file.

This function reads a specific portion of a binary file starting at a given offset and returns the data as a null-terminated string.

Parameters

file	The file pointer to the champion file.
buffer	A pointer to a temporary buffer for reading.
offset	The starting position in the file to read from.
len	The number of bytes to read.

Returns

A pointer to the allocated string containing the data, or NULL on failure.

4.36.2.4 get_names()

```
int get_names (
            arena_t * arena)
```

Extracts the names of all champions in the arena.

This function iterates over all champions in the arena and calls `extract_name` for each of them to populate their `name` fields.

Parameters

arena	A pointer to the arena structure containing the champions.

Returns

0 on success, 1 if any champion's name could not be extracted.

4.37 src/store_instructions.c File Reference

Functions to store instructions in the virtual machine.

```
#include "op.h"
```

Functions

- static int fill_vm (arena_t *arena, int i)

  Fills the VM memory with a champion's instructions.
- int store_instructions (arena_t *arena)

  Stores each champion's instructions in the VM memory.

4.37.1 Detailed Description

Functions to store instructions in the virtual machine.

This file contains the implementation of functions that handle the storage of champion instructions into the Corewar virtual machine's memory space.

4.37.2 Function Documentation

4.37.2.1 fill_vm()

```
static int fill_vm (
            arena_t * arena,
            int i)  [static]
```

Fills the VM memory with a champion's instructions.

Copies a champion's compiled instructions into the virtual machine's memory starting at a given position. Handles wrapping around the memory size (circular memory model).

Parameters

arena	Pointer to the arena structure.
i	Index of the champion whose code is being loaded.

Returns

0 on success, 84 on failure (e.g., memory allocation error).

4.37.2.2 store_instructions()

```
int store_instructions (
            arena_t * arena)
```

Stores each champion's instructions in the VM memory.

Iterates over all champions in the arena, extracting and copying their instructions into the virtual machine's memory using `fill_vm`.

Parameters

arena	Pointer to the arena structure.

Returns

0 on success, 84 if an error occurs during instruction loading.

4.38 src/utilities.c File Reference

Utility functions for the Corewar virtual machine.

```
#include "op.h"
#include "tab.h"
```

Functions

- int get_param (champion_t *champion, arena_t *arena, int arg_type, int position)
  Retrieves a parameter value from memory or registers.
- void reset_array (int *arr)
  Resets a 4-element integer array to -1.
- int check_instruction (unsigned char byte)
  Checks if a byte corresponds to a valid instruction.
- int check_coding_byte (unsigned char instruction)
  Determines whether an instruction includes a coding byte.
- void create_node (champion_t *champion, int position)
  Creates and appends a new process node to a champion's process list.

4.38.1 Detailed Description

Utility functions for the Corewar virtual machine.

Contains helper functions used throughout the Corewar implementation, including parameter fetching, wait time calculation, instruction validation, and memory array resets.

4.38.2 Function Documentation

4.38.2.1 check_coding_byte()

```
int check_coding_byte (
            unsigned char instruction)
```

Determines whether an instruction includes a coding byte.

Instructions like `live`, `zjmp`, `fork`, and `lfork` do not use a coding byte. This function returns 0 for those, and 1 for others.

Parameters

instruction	The opcode of the instruction.

Returns

1 if it uses a coding byte, 0 otherwise.

4.38.2.2 check_instruction()

```
int check_instruction (
            unsigned char byte)
```

Checks if a byte corresponds to a valid instruction.

Compares a byte against the `op_tab` to verify if it matches a known instruction code.

Parameters

byte	The byte to check.

Returns

The index in `op_tab` if valid, -1 otherwise.

4.38.2.3 create_node()

```
void create_node (
            champion_t * champion,
            int position)
```

Creates and appends a new process node to a champion's process list.

This function allocates a new process (proc_t), sets its default state (carry = 1, wait = 0, pc = position), and adds it to the end of the champion's linked list of processes.

Parameters

champion	Pointer to the champion structure.
position	Initial program counter (pc) for the new process.

4.38.2.4 get_param()

```
int get_param (
            champion_t * champion,
            arena_t * arena,
            int arg_type,
            int position)
```

Retrieves a parameter value from memory or registers.

This function extracts the value of a parameter depending on its type: register, indirect, or direct. For indirect and direct types, it reconstructs the value from bytes in memory.

Parameters

champion	Pointer to the current champion structure.
arena	Pointer to the arena structure (VM state).
arg_type	The type of the argument (T_REG, T_IND, or T_DIR).
position	The memory position from which to extract the value.

Returns

The integer value corresponding to the argument type.

4.38.2.5 reset_array()

```
void reset_array (
             int * arr)
```

Resets a 4-element integer array to -1.

Used to clear argument type arrays or other temporary storage.

Parameters

arr	The array to reset.

Index