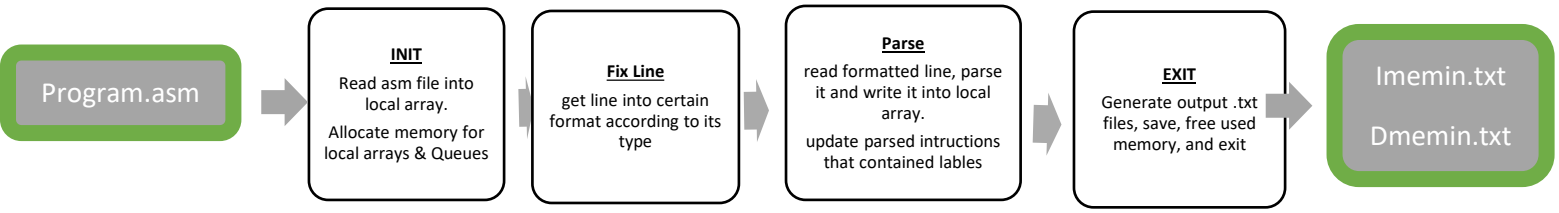
Computer Structure – Assembler / Simulator Documentation

ASSEMBLER FLOW



Func explanations

```
void init_DS();
```

allocate memory for local use arrays and queues. The arrays will be copied to output files and queues are used for updating instructions with labels.

```
int fix_line_for_parsing(char** broken_res, char *lineBuffer);
```

gets line as it was read from asm file as lineBuffer, break it down to an array of strings as broken_res, without comment if there is any.

```
void parseLine(char **line, int lLength, int *pc, int *dtable_size);
```

gets line and its length from fix_line_for_parsing as line and lLength, determine its type, sending it to corresponding parsing function, and updates pc or dmemin local array biggest index if necessary.

```
void update_labled_instructions();
```

after parsing all lines from asm file and writing it into local arrays, scan queue which is (1) instruction's pc which has label and the desired label name and update the label by scanning other queue which is (2) label's name and its corresponding pc.

```
int break_buffer(char **broken_buffer, char *lineBuffer);
```

gets line as it was read from asm file as lineBuffer and break it down to an array of strings according to a set of delimiters into broken_buffer. Returns the amount of strings contained in broken_buffer

```
void drop_comment(char **line, int *line_len);
```

search line, which is the result of break_buffer, for ‘#’, drop all strings from this point onward and update the amount of strings in line.

```
lineType get_lineType(int line_len);
```

determine which kind of line was passed into parseLine according to the line length. After fix_line_for_parsing there is one-to-one match between line’s length and its type.

```
void parseInstruction(char** line, int pc);
```

gets line and pc, generating the parsed version of line into temporary string, a copying it to local array at pc index.

```
void parseLabel(char *label_name, int pc);
```

gets label’s name and corresponding pc, making a new queue’s node and adding it to label’s queue for later addressing.

```
void parseWord (char **line, int *dtable_size);
```

gets pseudo-command to add word as line, adding data to local dmem array at address index.

```
void update_pc(int *pc, lineType lt);
```

gets current pc and kind of instruction that was just parsed, updating pc if needed according to lineType.

```
void update_dtable_last_idx (int *dtable_size, int curr_address);
```

gets current biggest index known in dmem local array and current index from read word-line and updating the biggest index if needed.

```
void add_op_to_result(char* result, char* str);
```

gets str which is the op code as it was read from asm file, converting it to parsed version and copying it into result. Copy and not concatenating because it is the first string inside parsed version of line.

```
void add_reg_to_result(char* result, char* str);
```

getting a register as it was read from asm file as str and concatenating its parsed version into result.

```
int add_imm_to_result(char* result, char* str);
```

getting an immediate as it was read from asm file as str and concatenating its parsed version into result. If it's a label, concatenate a flag for later use, and make a node of this instruction for instructions_with_label queue for later addressing.

```
void str2param (char* result, const char *str);
```

gets str which is op or register as it read from asm file and convert it to its parsed version, concatenate it into result. Result size vary due to sizes' indifference of register and opcode.

```
int isLabel(char *str);
```

gets a str which is immediate as it was read from asm file and determine if it's a label by checking its first character.

```
int isHexa (char* str);
```

gets a string representation of a number as str and checks if it's a hexadecimal representation by checking if its prefix is 0x / 0X.

```
int str_to_2complement(char* str, int hex_len);
```

gets a decimal number (positive or negative) as str and returns the number which is the equivalent of str in 2's complement for an integer with hex_len*4 bits.

```
void num2hexa (char *result, char *str, int hex_len);
```

gets a number as str and returns its hexadecimal representation into result with hex_len bytes.

```
void make_new_node(data_node *new, char *name, int pc);
```

gets a pointer to a new node and fills it with data according to inputs.

```
void add_to_Queue(data_node *new, int isLabel);
```

add the node new to queue. Queue is determined by flag that is passed by input

```
data_node* find_node(char* search_term, int LabelQ);
```

returns pointer the node that contains search_term in the queue which is determined by the flag from the input.

```
void free_Queues();
```

free the memory allocated in both queues. The function called after the queues' work is done.

```
void add_line_to_table(char *parsedLine, int hex_address, int is_intruction);
```

gets parsed line to be copied into local array in address index. The table is determined (dmem/imem) by the flag from input.

```
void copy_table_to_file(char **table, int table_size, char *file_path);
```

gets local array and its size and copies it into the file which is named file_path.

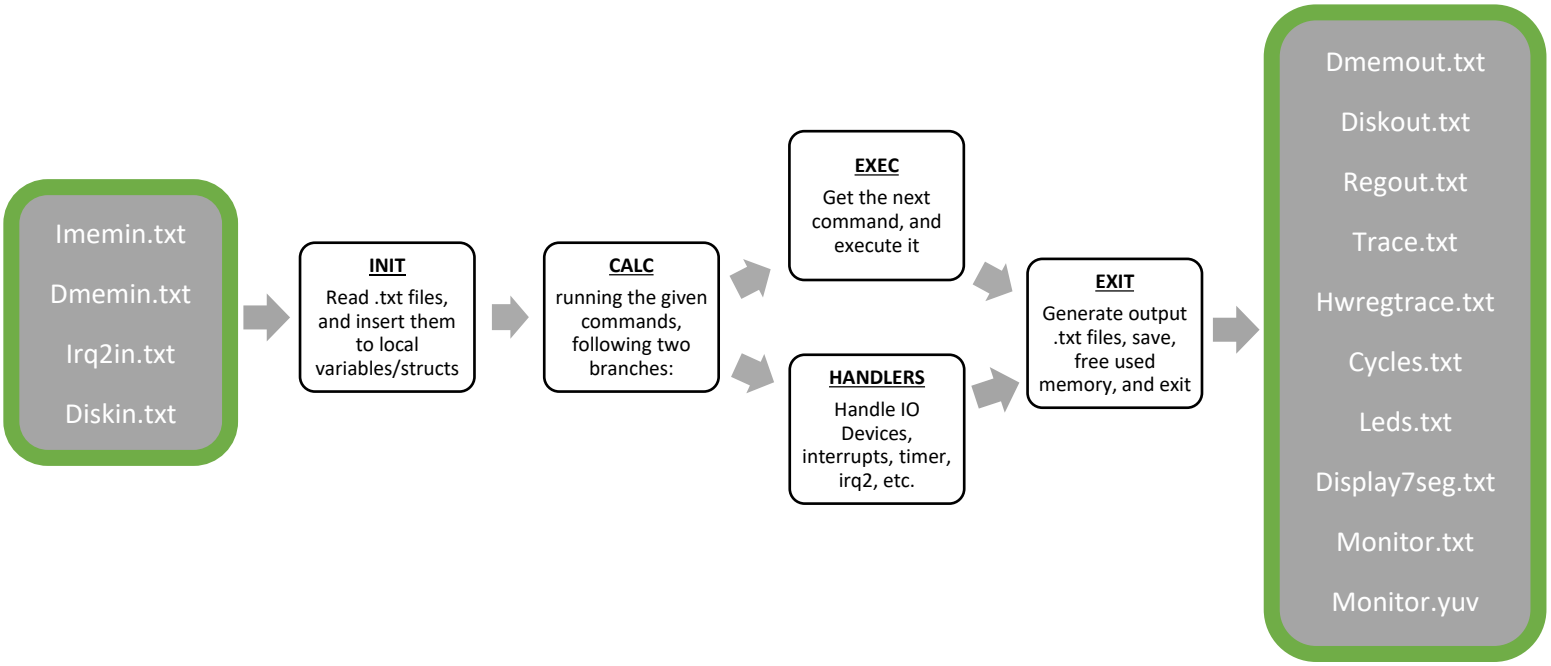
```
void free_table(char **table, int start, int end);
```

free the local array pointed by table in range [start, end).

```
int init_unparsed_instructions(FILE *fp, int len);
```

gets file pointer to program.asm file and len for each line's buffer and copy its content into local array. Returns the amount of lines read from the asm file.

SIMULATOR FLOW



Func explanations

```
void add_to_cmd_lst(Instruction *cmdLst, char *inst);
```

This function gets hexadecimal instruction from imemin.txt, and parses it to struct instruction, with decimal fields for opcode, registers, and immediates.

```
void main_loop();
```

Called after initiating all vars and structures – and it runs on cmdLst, calls handlers, and calls run_command for execution – main_loop is the second part of the main() function. Main() inits variable, and main_loop runs main flow.

```
void interrupt_handler();
```

Checks if any interrupts have been called, and if so, runs the flow needed to handle them (Whether it is changing PC, calling IO devices, etc.)

```
void update_irqs_state(int *irqState);
```

Updates irq IO Registers – checks if an interrupt has been given.

```
void diskIO_handler();
```

This function handles reads/write from/to disk – checks disk cycles, and writes/reads after 1024 cycles.

```
void timer_handler();
```

Handles timer – checks if timer needs to be increased, and that it hasn't reached the limit.

```
void update_monitor_pixels();
```

Updates current monitor pixels, based on information for monitor IO registers.

```
void update_irq2(int cycle);
```

Checks if irq2 reached the next cycle it calls an interrupt, and handles it.

```
int run_command(Instruction instruction);
```

Executing given command – separates R/I format operations, and calls needed func to execute current opcode.

```
void run_arithmetic(Instruction instruction, int id);
```

This function is called by run_command if the next command is an arithmetic command – executes it on needed registers.

```
void run_jump_branch_commands(Instruction instruction, int id);
```

This function is called by run_command if the next command is a branch command – executes it on needed registers.

```
void run_memory_command(Instruction instruction , int id);
```

This function is called by run_command if the next command is a memory command – executes it on needed registers.

```
void run_IOregister_operation(Instruction instruction , int id);
```

This function is called by run_command if the next command is an in/out command – executes it on needed IO registers.

```
void init_txt_files();
```

Creates required txt files, so they can be appended upon change (for files that change along the run, such as leds, display7seg, etc.).

```
void write_exit_txt_files();
```

Called when program terminates successfully – spills all given information to .txt files as required in the assignment.

Simulator / Assembler Util functions

```
/*----------Read Functions----------*/
int read_from_file(FILE *fp, int len, Mode mode);
int init_disk_lst(FILE *fp, char *line, int len);
int init_data_lst(FILE *fp, char *line, int len);
int init_inst_lst(FILE *fp, char *line, int len);
int init_irq2_lst(FILE *fp, char *line, int len);
int add_to_inst_lst(char *instruct, char *line);
int add_to_data_lst(int *mem, char *data);
int add_to_irq2_lst(int *irq2, char *data);
void read_from_disk(int* disk_sector, int* mem_buffer, int buffer);
/*----------Write Functions----------*/
int write_to_file(FILE *fp, int len, Mode mode);
int write_int_arr_to_file(FILE *fp, char *line, int line_len, int *arr, int arr_len);
int write_str_to_file(FILE *fp, char *line);
int write_diskout(FILE *fp, char *line, int len);
int write_dmemout(FILE *fp, char *line, int len);
int write_registers(FILE *fp, char *line, int len);
int write_trace(FILE *fp, char *line, int len);
int write_hwregtrace(FILE *fp, char *line, int len);
int write_led_7seg(FILE *fp, char *line, int len, int IORegIndex);
int write_cycle(FILE *fp, char *line, int len);
int write_monitor(FILE *fp, char *line, int len);
void write_to_disk(int* disk_sector, int* mem_buffer, int sector);
```

These function's purpose is reading/writing to txt files – reading when simulator starts running to have imemin/dmemin/diskin/irq2in as variables so the program can run its main loop more fluently and writing when program terminates to output .txt files.

```
void fill_with_null(int start, int end, Mode mode);
```

Fills given (mode) variable/struct with nulls, from [start] to [end].

```
char cut_string_by_index(char *str, int i);
```

Cuts given str on index i.

```
int compare (const void * a, const void * b);
```

compares obj a and b, for qsort func.

```
void sign_ext(int *num);
```

Performs sign extension to num.

```
void dec2hexa(char* result, int num, int len);
```

Converts num from dec to hexa, and writes output in result.

```
int hexa2dec(char *hex_rep, int len);
```

Converts hex_rep from hexa to dec, and returns result as an integer.

```
void set_line_to_zero(char *line, int len);
```

Sets line from index 0 to len to zeros.

```
int get_max(int a, int b);
```

returns max between a and b.