Computer Architecture Project - Documentation

Tsuf Zamet 205532864

Shay Yehuda 315001297

Roy Milshtein 318228343

Project's goal:

Write a simulator of a CPU using Tomasulo Algorithm, such that number of functional units/reservation stations and units delay's are configurable values.

Definitions:

As defined in the requirements, we have the following definitions:

```
#define NUM_OF_REGS 16
#define MAX_MEMIN_LINES 4096
#define INST_QUEUE_SIZE 16
#define TAG_SIZE 6
```

We defined the length of the "tag" to be 6 – composed of 3 letters + 2 digits + \0.

Structs:

Instruction – contains all basic fields of an instruction:

opcode, dst, src0, src1.

In addition, we added other fields relevant for reservation stations and traces:

pc, result, tag, raw_inst, clk_fetched, clk_issued, clk_start, clk_end, clk_write.

Such that result will hold the instruction's results from execution stage, the tag field will hole the name of the reserveation station the instruction is assigned to (ADD0 for example) and raw_inst is the instruction's line from memin.txt.

reservation station – contains all basic fields of the reservation station table:

name, pc, busy, opcode, Vj, Vk, Qj, Qk. In addition, we have the fields: num_func_unit and index that are used for addressing a specific station in the stations array.

Functional unit – contains delay and busy fields, holding the specific functional unit's delay and a Boolean for whether its busy or free to get new instruction to execute.

Instruction queue – contains the queue (array of instructions) named inst_q and counters for "front" and "rear" that mark the queue position.

reg – contains the fields relevant for each register: value, tag and "valid" (used for updating registers, similar to the register status table we know).

CDB_station – contains the fields: busy, data, tag, dest and cdb_type (2 - ADD, 4 - MUL, 5 – DIV), Such

that dest will hold the destination register to write to (integer).

trace_CDB – storing data in each writing to CDB in the fields: cycle, pc, cdb_type, data, and tag. Will

be in use for writing to the tracecdb file.

Global variables:

We defined some global variables, that will be in use by many different functions along our code. We will describe some here, next to their definitions.

```
reg regs[NUM_OF_REGS] → register's_status array, contains 16 elements
inst inst_arr[MAX_MEMIN_LINES] → instructions array that will hold all the
instructions from memin.txt file
inst_queue instruction_queue; → Instruction's queue
trace_CDB trace_cdb[MAX_MEMIN_LINES]; → An array of trace_CDB items to be printed
at the end of execution
// Definitions of functional units and reservation stations (of each type) as
array's of the relevant structures.
func_unit* add_units;
func_unit* mul_units;
func_unit* div_units;
res_station* add_reservation;
res_station* mul_reservation;
res_station* div_reservation;
// Definitions of the 3 CDBs
cdb add_cdb;
cdb mul_cdb;
cdb div_cdb;
```

In addition to the variables described above, we also defined counters for different uses and variables that will hold the data from configuration file. Initializations of those variables will be in functions along the code.

Simulator

In order to implement the Tomasulo simulator we used many functions. We will describe now the main function, and from there we will continue explaining about the functions we used.

In general, all functions have declarations at the beginning of the code (in the order of appearance) split into general categories.

Initialization:

- Extract variables from cfg.txt
- Extract instructions from memin.txt argument
- Initialize the number of free functional units of each type to the number of units.
- Build and initialize reservation stations, functional units, instructions queue and registers array.

While loop – each iteration is a clock cycle – looping as long as **halt** and **done** are not both "1".

- Fetching instructions – up to two for each cycle, if we have available slots in instruction queue and we haven't fetched **halt** instruction
- Sending instructions to reservation stations (based on instruction type).
- Executing ready instructions in free functional units.
- Writing ready results to cdb, and from there to registers and waiting stations

Closing and stopping:

- Creating output files – regout.txt, traceinst.txt, tracecdb.txt using helper functions:
- Freeing memory allocated during the code
- Exiting

We will give some more details now on the stages and functions:

Initialization:

- Checking all input files exist (total of 6 arguments).
- Call config function that gets cfg.txt, opens it and sets the right values to the relevant variables: number of functional units, number of reservation stations, and calculation delays (for each operation "add", "mul", "div").
- Initialize number of free functional units for each type to the number of units from configuration file.
- Call memin_read function to read all lines in memin.txt, create instructions and store them in array. We use a helper function called create_inst that receives the line of text, parses the relevant information, and creates an instruction structure from it. This function also initializes all empty fields of the instruction for later use.
- Call initial_queue function to initialize the instructions queue – setting both rear and front pointers to -1.
- Allocate memory for functional units arrays and call build_functional_units function to create the relevant number of units of each type. In this function we loop over the number of units to create, and initialize the relevant fields for each station.
- Allocate memory for reservation stations arrays and Call build_reservation_station function to create the relevant number of stations of each type. We loop over the number of stations to create and initialize the relevant fields for each station.
- Call create_regs to create the registers and initialize their values as needed.
- Call init_cdb to initialize the values of the 3 CDB stations (ADD, MUL, DIV).

In our while loop we divided the operation to the known stages for order and simplicity:

Fetch stage:

This stage inserts up to 2 instructions to the queue, if there is an available space. It uses the function enqueue_inst_q that checks the conditions, updates the queue and copying the instruction from instructions array to the queue. We stop fetching instructions when we fetched a "halt" instruction. For each instruction we fetch we update the clock_fetched field in order to avoid issue of it on the same clock.

Issue stage:

This stage reads up to 2 instructions from queue and assigns them to relevant reservation stations. Issue is done in order, as we learned. We make sure to check if instruction was issued in the same clock cycle, and have it wait for the next clock. To check for available station, we use the function into_res_station, which looks for a station of the relevant type (ADD, MUL, DIV), and fills the relevant fields in the station's line. If we found an available station for the instruction – we dequeue it from the instraction queue.
We also update the clock_issued field of the instruction in the instructions array (inst_arr) for trace and later checks.

Execute stage:

In this stage we execute ready commands in stations when we have available units. For each operation we check if we have free units, and if we have instructions ready to execute. If so, we use the function execute_inst to execute it. If we don't have enough units for the ready instructions, we chose based on their "pc" – lower pc will be executed first (using the qsort function).
The function execute_inst places the instruction in a free unit and does the calculations. It marks the clk_end time for the instruction based on the delay required, and stores the result.

Write result stage:

In this stage we check if there are any instructions that finished execution, if there is more than one instruction per type we will choose the oldest one according to their pc. We clear their station (with the function clear_station), update the clk_write, and write them to the relevant CDB with the function cdb_write. When writing to CDB we update the relevant registers according to their tag, the trace_cdb, and then the values of other instructions in stations that are awaiting execution (according to the Qj, Qk values).

is_done:

If received halt flag, makes sure we finish executing the instructions that came before it before halting – checking all reservation stations are empty, and queue is empty. Marking done=1 exits the loop and we finish execution.

After exiting the while loop, we have the exiting stage:

print_regout – receives the filename for printing the final register values, opens a new file and writes them in the required format.

print_trace_inst – receives file name for printing the trace, and opens a new file to print to. It loops over all instructions in the array, and prints to file all the relevant values of it in the required format.

print_trace_cdb – receives file name for printing the trace, and opens a new file to print to. Loops over all lines in the trace_cdb array, and prints to file all the relevant values of it in the required format.

Then finally, freeing memory for stations and units, and exiting.

More helper functions:

safe_file_open(const char* name, const char* mode) – opens a file and checks if something went wrong – throws an error.

find_res_station(res_station* reservation_station, int station_num) – looking for not busy station by station type, and returning the number of the station in the array. Called from into_res_station in the issue stage.

Queue functions:

- queue_is_full()
- queue_is_empty()
- dequeue() – dequeue instruction and returned it.

- enqueue_inst_q(inst* instruction, int inst_pc) – enqueue instruction to queue and update clk_fetched.

is_ready_for_execution(res_station* station) – returning if the station has the value needed for execute (Vj, Vk). Called in execute stage.

compare_pc(const void *a, const void *b) – check which station has older instruction.

find_free_unit(int opcode) – find free functional unit according to type.

Test files:

We have 3 folders containing tests we ran on our simulator. Each folder contains the input files "cfg.txt", "memin.txt" and the corresponding output files: "regout.txt", "traceinst.txt" and "tracecdb.txt".

Test number 1 focuses on trying to fill the queue, and making sure we don't miss instructions.

Test number 2 focuses on having more than 10 stations (longer TAG), and making sure we get the same results with correct TAG copying and printing. We also set longer calculations time to make sure we use more than 10 stations.

Test number 3 focuses on testing hazards regarding minimum stations and execution units. We also check for correct stopping when reaching HALT, and not reading further instructions.