Computer Organization Final Project – Documentation

Tsuf Zamet

Roy Milshtein

fibo.asm

Our Fibonacci series assembly code runs like the following pseudocode:

```
$t0 = 0
store $t0 at 0x100
$t1=1
store $t1 at 0x101
mem = 0x102
overflow = 524,287        # (2^19-1)
while (TRUE)
{
$s2=$t0+$t1
$t0=$t1
$t1=$s2
if ($s0>overflow) break
store $s2 at mem
mem++
}
```

This is a simple while loop code to calculate the Fibonacci series:

Storing the first 2 base values:

```
$t0 = 0                   #fib(0)
$t1 = 1                   #fib(1)
```

Setting our constants:

$s0 = 256 (0x102) #memory location to start storing fib(2) and so on

$s1 = 524287 ($2^{19}$-1) #overflow value to check (biggest unsigned integer using 20 bits)

Then running our loop:

Calculating the next Fibonacci number and storing it in $s2. Then changing our variables for the next iteration ($t0, $t1).

Checking for overflow, and breaking if necessary.

Then storing $s2 (our next fib number) in memory, and incrementing the memory address.

The code will only stop when $s2>overflow as intended. All previous results would be stored in their appropriate locations in memory.

ASSEMBLER

Our assembler is working in the following way:

- Opening program("fibo.asm"), then initialize the program counters (pc, ic) and the instruction table.
- Running the first stage (reading lines, storing instructions (in a special structure), and sorting labels. Also storing ".word" instructions in memory)
- Running second stage (replacing labels in the instructions with absolute locations, and storing instructions in the memory).
- Open a new file (memin.txt) to store the output by printing all the memory to it (includes all instructions and data).
- Closing program.

Structures:

SymbolEntry – a structure that contains 2 fields – "name" and "add". Name is a label found in the assembly program and 'add' is a number representing the absolute address of the label name in the program.

broken_inst – a structure that contains the following fields: opcode, rd, rs, rt, imm_val, imm and imm_int. "imm_val" is the string value of imm, "imm" as Boolean field that gets TRUE if it is an I-type instruction, and "imm_int" is the for integer value of imm_val (absolute label location, or int value of the immediate).

Inst_table – array of instructions from type "broken_inst". The size of the array is the size of the memory (no more than 4096 instructions).

Global variables:

str_inst_table – a table of instructions from type "inst_table". Holds the instructions found in the program, each one stored as a "broken_inst" structure.

sym_table – array of labels found in the assembly program. Every item in the array consists of absolute address and the label name (SymbolEntry type).

label_counter – an integer counter for labels.

pc – our main program counter, defined as a short unsigned int (because contain 12 bits and we don't need it to be longer than 2 bytes). Counts the number of lines in the program (including imm lines)

ic – counter for the number of instructions. Imm lines are not counted here. This will be used in the second stage to iterate over the instructions.

memory[] – array that represents the memory (4096 lines long). Initialized to all zeros. This array will contain all our instructions and data before being stored in the output file. Chose this way in order to make it easier to access lines in memory.

op_names[] – a pointer to an array of all legal opcode names (as strings).

reg_names[] – a pointer to an array of all register names (as strings).

Helper functions:

remove_comment – gets a pointer to the head of a line, finds '#' and removes everything after it. Stops if reached end of line (\n or \0).

get_label – gets a pointer to the start of a label. Stores the label in sym_table and increases the label counter.

split_line_to_tokens – gets a pointer to head of a line, splits to tokens (op,rs,rd,rt,imm) and stores it in the instruction table. Also checks for imm value (true or false) and fills it accordingly.
This function also check the validity of the tokens (opcode and reg name length).

remove_extra_spaces – removes extra spaces at the end of the string by cutting with '\0'.

convert_string_to_int – converting a number in a string form into an int type. This is working for both hex and decimal numbers that appear in the input string.

labelcomp – receiving pointers to 2 labels and comparing them char by char. It is used to find a label in the label table.

store_inst_in_mem – gets an instruction from the inst_table, translating it part by part into inst_to_store, then saving it in memory with imm_counter offset.

The function also checks for imm instruction, then saves imm_val in memory after the instruction and incrementing imm_counter.

Before storing the instruction, the function also checks for illegal opcode/register names, or illegal operations, then exits.

first_stage – receiving pointers to the program, instruction table, and program counter (pc). Running a loop reading lines from the program one by one. Removing comment, and checking for label.

If label found, storing it in sym_table with its name and address. Then checking for instruction in the line. If found, splitting it to tokens and storing it in the instruction table. If a ".word" instruction was found, then storing the correct value in the proper location in memory array.

Increasing the program counter (pc) depending on the type of instruction (1 for R-type, 2 for I-type).

second_stage – receiving the instruction table and the program counters. Running a loop on all the instructions found. Checking if they have labels in the imm field, and replacing them with their absolute address (stored in the sym_table). Finally, sending our instruction to the function "**store_inst_in_mem**" that that checks for valid instruction and saves it to the memory.

SIMULATOR

Our simulator works in the following way:

- Opening “memin.txt” to read from, and “trace.txt” to write to.
- Running the simulate function (reading all the file memin.txt to memory, then writing to “trace.txt” and executing from memory)
- Initializing and writing to new destination files:
    - memout.txt – copying final memory line by line to the new file.
    - regout.txt – printing to file the final values of the registers.
    - cycles.txt – printing the number of clocks the program ran for.
- Closing all files and exiting.

Global variables:

Regs[16] – int array initialized to 0, containing the values of the registers during the run of the code.

pc – our running counter that points to the current instruction needed to execute. Incrementing by 1 after every execution. The pc can jump when branching or jumping.

ic – counting the number of cycles performed: 1 for every instruction, +1 for I-type instructions and +1 for lw/sw instructions. This is the number of cycles we will print to cycles.txt.

memory[4096] – int array initialized to 0, containing the entire memory for our program (instructions and data). Will be printed to memout.txt at the end of the simulation.

halt_flag – integer initialized to 0. Will be set to 1 when encountering a “halt”. Used to stop the run.

Helper functions:

hex_to_int – getting a char in hex and returning its value as integer.

hex_string_to_int – getting a pointer to a string and returning its value as an integer (also used for sign extension).

print_inst_to_trace – getting a pointer to trace.txt, pc and inst (integer representing the instruction). Printing to trace.txt the HEX values of pc, current instruction, and all register values. The printing is according to the specified format.

execute_op – getting the opcode and variables (regs numbers), and executing the operation. Calling the needed function using switch case for all 19 options.

Simulate – looping over lines in “memin.txt”, converting them to int, and storing them in the memory. After loading everything into memory, starting to execute with a “do-while” loop: decoding instruction, writing it to “trace.txt”, and executing it. Eventually updating the pc for the next instruction. When halt_flag==1 we stop the loop, and exit.

Operation functions:

op_add – receiving rd, rs, rt, and performing: regs[rd] = regs[rs] + regs[rt]

op_sub,op_mul – similar to op_add, performing subtraction or multiplication.

op_and,op_or,op_xor – performing bitwise and/or/xor.

op_sll,op_srl – performing on rs logical shift to the right/left by the amount rt, and storing in rd.

op_sra – performing arithmetic shift to the right with sign extension, and storing in rd.

op_beq– checking if regs[rs] == regs[rt]. Then changing pc to the value of regs[rd] if the condition was met.

op_bne – checking if regs[rs] != regs[rt]. Then changing pc to the value of regs[rd] if the condition was met.

op_blt – checking if regs[rs] < regs[rt]. Then changing pc to the value of regs[rd] if the condition was met.

op_bgt – checking if regs[rs] > regs[rt]. Then changing pc to the value of regs[rd] if the condition was met.

op_ble – checking if regs[rs] <= regs[rt]. Then changing pc to the value of regs[rd] if the condition was met.

op_bge – checking if regs[rs] >= regs[rt]. Then changing pc to the value of regs[rd] if the condition was met.

op_jal – storing pc in regs[rd] ($ra), and changing pc to the address in regs[rs].

op_lw – loading into regs[rd] the value of memory found in (regs[rs] + regs[rt])mod4096. The “mod” is to prevent addresses outside memory. This is a cyclical approach to our memory.

op_sw – storing in memory address: (regs[rs] + regs[rt])mod4096 , the value in regs[rd]. as in op_lw, “mod” is to prevent addresses outside memory. This is a cyclical approach to our memory.

op_halt – setting halt_flag = 1. Then the simulator will know to stop the run.