Instruction-Level ARM Simulator

Alex Sensintaffar: A20131972

Josh Minton: A20148053

Introduction:

In this lab we will be creating a C program to run a limited set of instructions for the ARM instruction set. The ARM simulator will be used to introduce us to how ARM software works and to introduce use to the ARM instruction set architecture (ISA).The current lectures are examining how the ARM functions are called upon. The lectures also cover how the different types of instructions are determined and how the different types determine the way the rest of the line is interpreted. Within the 32 bits given for the function commands different bits will control different parts of the instruction and will affect how the instruction acts.

In this lab we were given a premade set of files which contained the partially finished C program. Our job was to finish the program. The largest and most important part of the program that we had to finish was to create the logic behind the given assigned functions. Using the ARM ISA, we were able to implement the 26 functions assigned. We also had to create the decode functions which would use the op code to determine which type of function was being used. Based on the function type the decoders would then determine which instruction is to be used. Within the sim.c file there were multiple decoding functions we were assigned to complete.

There was a total of 26 instruction that we had to implement. Theses instructions were all very common and basic instruction. Theses instructions are ADC, AND, ASR, B, BIC, BL, CMN, CMP, EOR, LDR, LDRB, LSL, LSR, MLA, MOV, MUL, MVN, ORR, ROR, SBC, STR, STRB, SUB, TEQ, TST, and SWI. Input files were given that would test the process for most of these instructions and act as a good testing step. Our design implemented all of the listed instructions and tested all of the listed instructions based on the requirements and standards given in the instructions.

Baseline Design:

The baseline design of the lab is the file given to use. The files given include the sim.c, sim.File, shell.c, shell.h, isa.h, Makefile.file, dumpsim.File. The shell.c and shell.h files were not to be changed. The two files are used to create the user interface. The isa.h file acts as the header file for the isa.c file. The instructions that we were assigned are listed in this file. The Makefile.File, sim.File and dumpsim.File file are used for creating and testing the simulations, but we did not change anything in this file as instructed.

The sim.c file contained a lot of functions that needed to be completed. The first function was the data_process function. The data_process function would assume the op code is set to 00, it will then read bits 24:21 to determine what data processing instruction to use. The next function that needed to be implemented is the branch_process function. The branch_process function works the same as the data_process function but it assumes the op code is set to 01. Data_process will read bit 25:24 to determine if the value of the L bit (bit 24). Based on the value of the L bit the function will determine which branch processing instruction is being called upon. The third function that needed to be implemented is the transfer_process. The transfer_process executes the memory instructions based on the values of the B and L bits and assumes the op code is 01. Other small changes were made to the preexisting function to increase compatibility with our functions.

How this program works is you run the Lab2 application with a .x file as an argument to start the simulation. The simulation uses the file sim.c and the shell files to format the command prompt to appear as the designated user interface. The command prompt waits for you to give it a command. The commands are go, run n, mdump low high, rdump, input reg_number reg_value, ?, and quit. The go command runs the program through all

the lines given. The run n command runs the program through n lines. Run is the command and n is the given argument acting as an int. The mdump low high command prints 4bit lengths of memory into the to the command prompt from the low address to the high address. Both low and high are int arguments. The rdump command returns the values stored in the registers. The input reg_number reg_value takes the inputted values of reg_value and stores it in the register determined by reg_number. Both reg_number and reg_value are ints data types. The? command has the command prompt print out the list of commands that you can use and a basic description of them. The quit command ends the program.

The program takes the .x file given and converts the hexadecimal into binary. The binary number is then examined to determine the op code. The op code is then used to determine which function to use i.e. data_process, transfer_process, or branch_process. The different function then determines which instructions to use based on the given binary code and run the instruction. This process is repeated for each function until the program ends.

Design:

The main part of our program that we designed was the instruction sets. When the program is given the hexadecimal code, it converts it to binary then it reads the op code. Form the op code a function is call that treats the binary code like data function, transfer function, or branch function. Each function has a set of instructions under it defined by their individual function code. The function code determines how each bit is to be interpreted as part of the instruction. Once the instruction is complete the program counter is increased by four and then the next hexadecimal is read until the program finishes.

Many of the data processing instructions use the operand2 and I bit in their instructions. The I bit determines if operand2 acts as an immediate, register, or register-shifted register. All three of the different types have to interpret the given data different from each other. If the I bit is 1 then bits 11 through 8 are used to determine the rotational value while bits 7 through 0 are the number value being rotated. If the I bit is 0 then bit number 4 is checked. If bit number 4 is a 0 then operand2 is treated as a register. As a register bit 11 through 7 are the amount the register will be shifted. Bits 6 and 5 are the type of shifting. Bit 3 through 0 is the register that will be shifted and is the second source operand. If bit 4 is 1 then bits 11 through 8 determine the register that contains the amount being shifted. Bits 6 and 5 determine what type of shifting will be performed. Bits 3 through 0 is the register being shifted and is the second source operand. This entire checking process takes multiple lines of code and is used on a lot of the instructions. We created a function called addressing_mode_handler which will take two integer argument. The arguments are the value of the I bit and Operand2. The function then creates variables of each possible register grouping regardless of what the I bit, or bit 4 value is. Conditional checks then check the I and if needed the bit 4 value to determine how it should act. The repeatable function addressing_mode_handler allows for us to reuse the same process that is needed on multiple instructions but without having to type the program. Another function was created called addressing_mode_mem which handled the I bit for the transfer processes.

Another important part of our program was the conditional check. The function called check_cond took an integer input and returned the conditional output. An example would be if the program gave a 6-digit hexadecimal input starting with a 1, the program would run the 1 through the check_cond function. The function was checking its value using a switch and case to see that it should return ~Z_N&1. ~Z_N&1 is the not equals check so when it is return the program is looking for the values to be not equal. Most of the time the hexadecimal numbers start with an E which is the unconditional condition. Unconditional conditions are when the program does not check for an existing condition to be met.

Testing Strategy:

Our testing process was pretty simple. While creating the instruction sets we would use the c compiler to compile the program and do a basic syntax check. Due to the simplicity of most of the instructions and the need for most of the instructions to be complete before the given input can be run, we waited to run full test till after the

instructions were completed. We then ran the given input programs and calculated the expected values by hand and compared the results. We would then run the program on the given inputs but when we ran into a complication or a fail test, we would run the input step by step.

To run the step by step program we would first look at the given output to determine where the program failed. Once we determined a general idea where the program failed, we then ran the program up until the line before the failure. If the line before had the expected result we would run the program again but now including one more line. We would repeat this process until we found the line or multiple line where the program failed. From there we simply would look over the instruction that failed and determine which parts of it fail and how to fix it. If we could not determine which parts of the program that failed, we would create a test set of inputs focusing on the specific parts we were investigating. Upon continuous testing and examination of the different parts of the program we eventually would determine the exact problem of the instruction and then fix it.

When we finished testing based on the given inputs, we preformed test that would check every instruction we created and test the different types of inputs in the instructions. Any instruction that used operand2 we tested to see if the instructions worked with us using the different values for the I bit and bit 4. We also check the memory functions to see if the performed correctly in the given test by giving it our own set of instructions that would test everything needed.

We ran the program using every test file given as well as running our own test on the programs. The program passed every test in the end and had not function problems. We determined the program has been sufficiently tested and has met the standards given to us.

Evaluation:

The entire program worked as intended. The simulation was able to take in the proper files and acted upon those files in the expected way.

While looking at different design implementations we found that there was little we could individually change on the program as what part we were supposed to complete and how we were supposed to complete the program was predefined. We added in functions to simplify the instruction sets but they were still fundamentally the same. While working on the program we used the design instructions given to us by the ARM ISA. The implementation instructions were precise and gave use clear information on how the instruction works.

When using the brand instructions, we ran into a problem with the program counter (PC). At the end of each instruction execution the PC was increased by 4 as a way of telling the program to go to the next line in the given instructions. When using the branch instructions, we had to increase the PC by 8 at the branch instruction call. This led to a problem where when we called the branch instruction the PC was going up by a total of 12. To fix the we simply had the branch increase the PC by 4 and the continuous increase of 4 at the end of the program resulted in the desired PC increase by 8.

Another major problem we ran into was when we used the branch instructions. We received an input of EAFFFFFF which would call the branch instruction and give it the maximum possible value in the immediate 24 section (imm24). When the instruction ran the PC should have become -4 and then have the default PC increase it by 4 making the PC 0. Because we are using integers which are 32-bit numbers the value stored in the variable imm24 was exceeding the 24-bit limit we intended but did not exceed the integer bit limit. When the 24-bit limit was exceeded it should have become a negative number. In order to fix this, we had to manually implement code that would cause the value to go negative once it exceeded the 24-bit limit.

This lab was used to show use how the ARMv4 language works behind the scenes. We had to understand how the different bit values could change how the rest of the bits should have been interpreted which we had to take account of when creating the instructions. The biggest part of this lab was to implement the instructions given to us but we also had to implement functions required to make the simulation work. The parts of the program that were

given to us all involve the design and implementation of the ARMv4 portions of the program. All of the part of the program that were based around the simulator was given to us. We used the example instruction ADD to base the rest of our instruction off of.