Lab 2: Instruction-Level ARM Simulator

Christian Moser A20173104 Isabel Cook A20208072

March 6, 2022

Chapter 1

Introduction

The purpose of this lab is to get familiar with reading and using ARM instructions, learn how to compile code in C, learn how to run code in C, and about how to model ARM ISA. This is done by creating a simulator that takes in ARM instructions and returns to the user the results of those instructions. Programming these instructions in C relates to the material taught in lectures and expands upon it. In lecture we learned about how to take hexadecimal commands, find the ARM instruction it describes, and actual find the output of that instruction, and in this lab we created a simulator that takes in a hexadecimal command, calculates the ARM instruction, executes the instructions, and outputs the values of the registers after all the instructions are completed.

Our design functions as specified by the baseline design outlined. The design simulates the set of ARM instructions specified for both register and immediate operations. The ARM instructions implemented are ADC, ADD, AND, ASR, B, BIC, BL, CMN, BMP, EOR, LDR, LDRB, LSL, LSR, MOV, MVN, ORR, ROR, SBC, STR, STRB, SUB, TEQ, TST, SWI. Our design also simulates the S suffix for all of the ARM instructions that use it, which allows the CPSR's flags to be set. These flags are carry, zero, overflow, and negative, all of which are implemented in our design. Also, the different shiftings such as register rotating and barrel shifting, and the one of SWI's behaviors that informs the program to end the go command are simulated in our design. However, we were not able to simulate the subset conditions EQ, NE, GE, GT, LT, LE, and AL due to time and if we did have more time this would have been implemented as well.

We tested our design against the given test files: addfirst.x, addiu.x, andor,x, bltest,x, brtest0.x, and memtest0.x. For each test file our design ran smoothly and functioned quite well for what is implemented. There was little to no delay in run time, and the only functional imperfection in our design in due to the lack of subset conditions being implemented.

Chapter 2

Baseline Design

The baseline design specifies an instruction-level simulator which models the behavior of data-processing, branch-processing, and memory-processing instructions in the ARM machine language. The simulator can accept the hexadecimal command, process it into the relevant codes and display it to the user, and then operate on those codes. It uses 15 registers, a program counter register, and a current program status register to operate on with the multitude of commands. The first 15 registers, R0 to R14, are used for holding data. The program counter, PC (or sometimes R15 as it is the 16th register), is used to store the location of the next instruction and branch instructions will directly interact with the PC register to go to the following instruction. Current program status register, CPSR, is used to store the flags on the simulator. These flags are negative (N), zero (Z), overflow (V), and carry (C). These flags are only set given the S suffix high in the ARM instruction command.

The baseline design is partitioned into two sections: 1) the shell environment and 2) the simulation routines. The shell environment is supplied in the handout, so it is the job of the design team to implement the simulation routine. The first command, AND, is given an incomplete implementation as an example of how to use the given shell. 2) requirements consist predominantly of completing the data processing of every instruction type, which is in the sim.c file, and encoding the ARM instruction set, which is stored in the isa.h header file. In conjunction, these files utilize the shell defined in shell.c and shell.h to create the single-cycle ARM instruction simulator. Additionally, various other files are given to make testing and compilation simpler. E.g., a makefile is provided to compile all of the given and created code. The arm2hex python executable is supplied with a list of inputs based on human-readable ARM assembly code and then converted into the ARM machine language with the python script in order to be fed into the simulator. The designer can then use the ‘rdump’ function of the shell to compare the register values of the simulator to the human-readable commands in the origin input file to evaluate their design.

```
The instruction is: e5c05007
33222222222211111111110000000000
10987654321098765432109876543210
--------------------------------
11100101110000000101000000000111

— This is a Data Processing Instruction.
Opcode = 1110
 Rn = 0
 Rd = 5
 Operand2 = 000000000111
 I = 0
 S = 0
 COND = 1110

shamt5 = 000000000000000111
 sh = 00
 Rm = 0111
```

Chapter 3

Design

This design creates a single-cycle ARM processor simulator that can utilize the ADC, ADD, AND, ASR, B, BIC, BL, CMN, CMP, EOR, LDR, LDRB, LSL, LSR, MOV, MVN, ORR, ROR, SBC, STR, STRB, SUB, TEQ, TST, and SWI ARM instruction-set architecture. It uses data-processing in a predefined shell environment to first receive a command in machine code, decode and process the instruction, and then respond accordingly to interact with the registers of the simulator. There are 17 available registers; R0 through R14 are data registers, R15 is the program counter, and R16 is the current program status register. The data registers are used for storing information processed from the machine code, the program counter is used for storing the address of the next instruction, and the current program status register is used for storing flags given the condition that they should be set–typically when the S suffix is triggered high in the instruction. The commands are fed into the shell on initial run by specifying the argument. That is, '/sim /path/to/instruction.x' where '/path/to/instruction.x' is a text file storing machine code for an ARM instruction, generated by feeding an instruction.s file into the Python script arm2hex. We can run the 'go' command in the shell to decode and operate all of the instructions from the instruction file and then use 'rdump' to examine and evaluate the instruction simulator by viewing the values of the registers.

We began the design by first finishing the data-processing portion of the sim.c file. This function works by decoding a pass character array *_i* and then subsequently completing the corresponding instruction call. For example, we can see how the op code, register addresses, and operands are decoded here:

```
char d_opcode[5];
  d_opcode[0] = i_[7];
  d_opcode[1] = i_[8];
  d_opcode[2] = i_[9];
  d_opcode[3] = i_[10];
  d_opcode[4] = '\0';
```

```
char d_cond[5];
d_cond[0] = i_[0];
d_cond[1] = i_[1];
d_cond[2] = i_[2];
d_cond[3] = i_[3];
d_cond[4] = '\0';
char rn[5]; rn[4] = '\0';
char rd[5]; rd[4] = '\0';
char operand2[13]; operand2[12] = '\0';
for(int i = 0; i < 4; i++) {
  rn[i] = i_[12+i];
  rd[i] = i_[16+i];
}
for(int i = 0; i < 12; i++) {
  operand2[i] = i_[20+i];
}
```

We then use a series of if-else statements to call the instruction defined by the opcode. This pattern of decoding is used in similar patterns to the branch processing, multiply processing, and transfer processing functions. These functions are then defined in the isa.h header file.

The isa.h file is used to create the entirety of the instruction-set architecture. Inside the file, we define all of the instructions and the condition function ‘check_cond‘ is used for checking the conditions of instructions, which are decoded in the sim.c processing functions. Despite writing the decoding for the multiply instructions, we were unable to implement the multiply instructions in the given time constraint. The code remains in the sim.c file but the corresponding instructions have been commented out in isa.h as we were unable to fully implement them.

Many design principles were utilized in this process. We applied some modularity to the program by designing function calls such as “set_overflow()” to be used in every instruction call to efficiently evaluate the V flag when necessary. All of the instructions exhibited a similar pattern, but shared no common function calls besides overflow flags and were all designed with Single Responsibility Principle in mind to keep the instructions isolated from each other so that each instruction can be heavily altered without affecting any others. The design team used strong communication to understand the work and functionality of designs created by other members, which was very effective for understanding the overarching design.

Chapter 4

Testing Strategy

To test our design we used the directed test strategy through the use of the given test files: addfirst.x, addfirst.s, addiu.x, addiu.s, andor.x, andor.s, bltest.x, bltest.s, brtest0.x, brtest0.s, memtest0.x, and memtest0.s. The design takes in hexadecimal commands given by the .x files, and computes the ARM instructions corresponding to each hexadecimal command, which changes 14 different registers, a program counter, and the CPSR. To test that the ARM instructions used in this lab worked we ran the program using the .x files, and used the .s files to double check the final status of the register files by calculating by hand what the end result of each should be.

We used this testing strategy as it provided an overall performance review of the design. These different test files given have a large range of ARM instruction usage in terms of the number of different instructions used and the variety of register and immediate value type of instructions used. For each file we went hexadecimal command by hexadecimal command and verified that its proportional ARM instruction was fulfilled accurately and precisely. For example, we can run our program with the input file addfirst.x using the command "./src/sim ./inputs/addfirst.x" from our main folder and typing "go" in to run the program. The first hexadecimal command (E3A01005) which converts to the ARM instruction "mov r1, 5" which will convert register 1 to be 0x00000005. Next the design will take hexadecimal command E2812F48 which converts to ARM instruction "add r2, r1, 300" and changes register 2 to be register 1 + 300. Register 2 ends up being 0x00000131. Finally, hexadecimal instruction EF00000A is taken in which converts to "swi 10" which tells the program to stop. We then type "rdump" in to see the program's final register values.

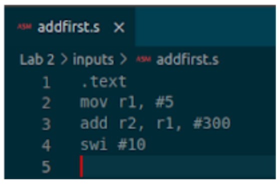
This type of testing was done for all of the test cases, and for each of the test files our design correctly computed the register values for the most part. The only cases where the registers were not computed accurately is when one of the subset conditions is used, for we did not have time to implement these into our design.

Overall this was the test strategy we chose to use based on its ARM instruction diverseness, and it contained the .s files that showed the ARM instructions line by line for the hexadecimal commands contained in their respective .x files. This aided in lessening the time consumption of creating random hexadecimal commands and personally converting them to ARM instructions to then find the final register values for. Instead by using these test files provided we were able to cut down the time cost of testing and just have to calculate the register values from the .s ARM instructions and compare them to the output register values from the program.

Chapter 5

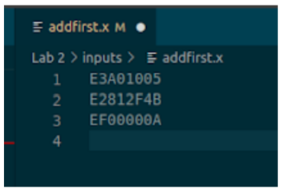
Evaluation

We tested our design by running our code with each of the given test files and ensuring that the final values of the registers were accurately represented based on the commands given by the input files.



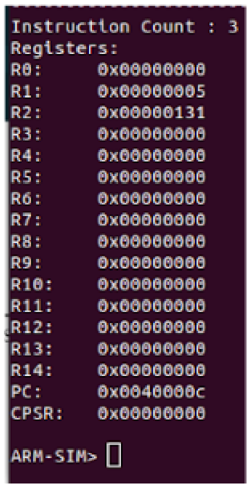
```
.text
mov r1, #5
add r2, r1, #300
swi #10
```

Figure 5.1: addfirst.s with ARM instructions



```
E3A01005
E2812F4B
EF00000A
```

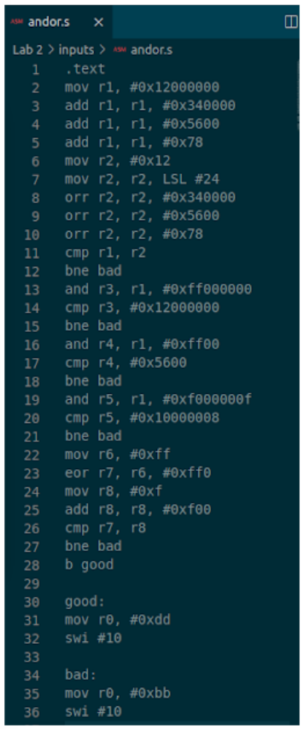
Figure 5.2: addfirst.x with machine language



```
Instruction Count : 3
Registers:
R0:     0x00000000
R1:     0x00000005
R2:     0x00000131
R3:     0x00000000
R4:     0x00000000
R5:     0x00000000
R6:     0x00000000
R7:     0x00000000
R8:     0x00000000
R9:     0x00000000
R10:    0x00000000
R11:    0x00000000
R12:    0x00000000
R13:    0x00000000
R14:    0x00000000
PC:     0x0040000c
CPSR:   0x00000000

ARM-SIM>
```

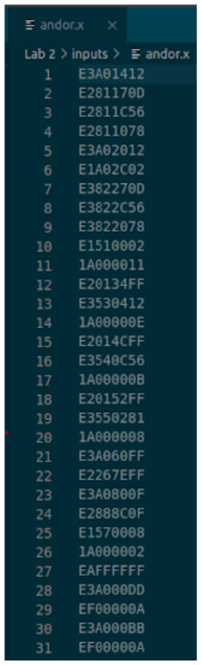
Figure 5.3: Final register dump after running addfirst.x

In Figure 5.3 we can see that the only registers changed are r1, which is changed to 0x00000005, and r2, which is changed to 0x00000131. These are the final results of the registers after the program ran addfirst.x seen in Figure 1b. These hexadecimal commands in addfirst.x are broken down by the program into its corresponding ARM instructions which we can see in Figure 1a (addfirst.s) which is not part of the actual running of the simulation but is a file given with addfirst.x. In addfirst.s we see that r1 is only changed once in the command "mov r1, 5" which is the first hexadecimal command in addfirst.x "E3A01005." This command moves register 1 to 0x00000005 which is also the final value shown for r1. We also see that r2 is only changed one in the command "add r2, r1, 300" (E2812F48) which adds r1 and 300 together and sets the corresponding value to register 2. This ends up being 0x00000131 which again we see is the final output value of r2. The final input instruction is "swi 10" (EF00000A) which tells the program to end the "go" cycle, in which it does.



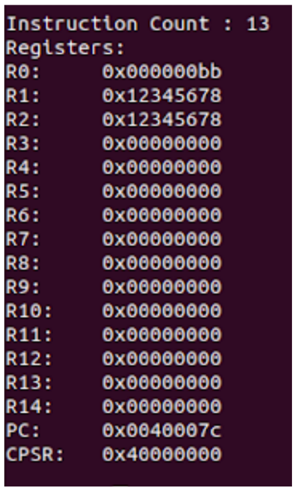
```
.text
mov r1, #0x12000000
add r1, r1, #0x340000
add r1, r1, #0x5600
add r1, r1, #0x78
mov r2, #0x12
mov r2, r2, LSL #24
orr r2, r2, #0x340000
orr r2, r2, #0x5600
orr r2, r2, #0x78
cmp r1, r2
bne bad
and r3, r1, #0xff000000
cmp r3, #0x12000000
bne bad
and r4, r1, #0xff00
cmp r4, #0x5600
bne bad
and r5, r1, #0xf000000f
cmp r5, #0x10000008
bne bad
mov r6, #0xff
eor r7, r6, #0xff0
mov r8, #0xf
add r8, r8, #0xf00
cmp r7, r8
bne bad
b good

good:
mov r0, #0xdd
swi #10

bad:
mov r0, #0xbb
swi #10
```

Figure 5.4: andor.s with ARM instructions



```
E3A01412
E281170D
E2811C56
E2811078
E3A02012
E1A02C02
E382270D
E3822C56
E3822078
E1510002
1A000011
E20134FF
E3530412
1A00000E
E2014CFF
E3540C56
1A00000B
E20152FF
E3550281
1A000008
E3A060FF
E2267EFF
E3A0800F
E2888C0F
E1570008
1A000002
EAFFFFFF
E3A000DD
EF00000A
E3A000BB
EF00000A
```

Figure 5.5: andor.x with machine language

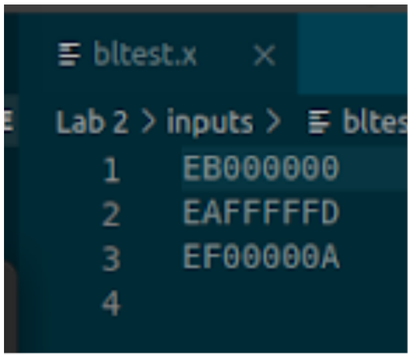


```
Instruction Count : 13
Registers:
R0:     0x000000bb
R1:     0x12345678
R2:     0x12345678
R3:     0x00000000
R4:     0x00000000
R5:     0x00000000
R6:     0x00000000
R7:     0x00000000
R8:     0x00000000
R9:     0x00000000
R10:    0x00000000
R11:    0x00000000
R12:    0x00000000
R13:    0x00000000
R14:    0x00000000
PC:     0x0040007c
CPSR:   0x40000000
```

Figure 5.6: final register dump after running andor.x

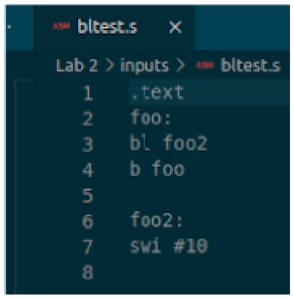
While our program works accurately for what is implemented in our design, when running the test cases there are some inconsistencies due to the subset conditions not being implemented in our design. For example, if we look at

Figure 5.5 which shows the ARM instructions, Hexadecimal commands, and the final output of the register values you will see that the register values do not quite line up. Firstly you can see that only register 1 and 2 have non zero values but in andor.s you can see commands that change the values of r4-8. This discrepancy is because of the command "bne" which is not implemented in our design due to time conflicts. The program does not recognize this command and proceeds to ignore it and stop altogether. With more time we can easily fix this by implementing the subset conditions: EQ, NE, GE, GT, LT, LE, and AL. However, if you look at register 1 and 2 for addiu.x shown in Figure 5.5 you see that they are computed accurately, and if you were to just run branching instructions as in bltest.x , as shown in Figure 5.6, our design is again accurate.



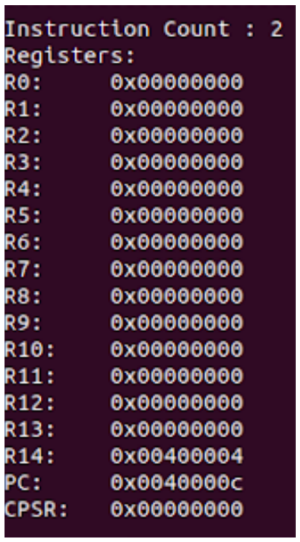
```
EB000000
EAFFFFFD
EF00000A
```

Figure 5.7: bltest.s with ARM instructions



```
.text
foo:
bl foo2
b foo

foo2:
swi #10
```

Figure 5.8: bltest.x with machine language



```
Instruction Count : 2
Registers:
R0:     0x00000000
R1:     0x00000000
R2:     0x00000000
R3:     0x00000000
R4:     0x00000000
R5:     0x00000000
R6:     0x00000000
R7:     0x00000000
R8:     0x00000000
R9:     0x00000000
R10:    0x00000000
R11:    0x00000000
R12:    0x00000000
R13:    0x00000000
R14:    0x00400004
PC:     0x0040000c
CPSR:   0x00000000
```

Figure 5.9: final register dump after running bltest.x

Figure 5.7–9 shows the use of branching using our simulation. As you can see in Figure 5.9 there is no changing of register values, but the PC value is very different then those seen in the previous figures. This value change comes from the branch commands used by the test file bltest.x.

Overall our design accurately takes in hexadecimal commands, figures out the corresponding ARM instruction for the hexadecimal command, and edit the 14 possible register values and process counter accordingly. Our design accounts for a number of ARM instructions: ADC, ADD, AND, ASR, B, BIC, BL, CMN, CMP, EOR, LDR, LDRB, LSL, LSR, MOV, MVN, ORR, ROR, SCV, STR, STRB, SUB, TEQ, TST, and SWI. These instructions can either use register operations or immediate operations, and can either barrel shift or register rotate. Our design also implements a functionality of CPSR when the S suffix is on an ARM instruction. This functionality is the setting of four condition flags: negative (N), overflow (V), zero (Z), and carry (C).

Our results would change to be more accurate and versatile if more conditions, functionalities, or parameters were added because the design would have more specifications in breaking down the imputed hexadecimal commands. We can learn from this that a design is only as accurate as its given code is. The design does not know what it is not told. For example, the addiu.x test uses some conditions that our design does not have implemented, and it can not process that information as accurately as we would expect or want.

Overall the design, for what it is, does not have large delays or hold ups when compiling or running the code, and it functions accurately with the instructions it knows how to handle. If we were to add more features, functionalities, or conditions to our design it would not impact the runtime environment, but would strengthen this design as an ARM instruction simulator.