CSCE 2303 – Computer Organization and Assembly Language Programming
Summer 2020

Project 2: ARM THUMB Simulator/Disassembler

Report

Farida Tarek Yousry 900171942

Marwan Awad 900172585

Mohamed El-Fekky 900170147

TABLE OF CONTENTS	
1.	**APPLICATION OVERVIEW**
1.1	Design rationale
1.2	Implementation & Algorithms
1.3	Member contribution
2.	**SYSTEM OVERVIEW**
2.1	Challenges
2.2	Design limitations

1. APPLICATION OVERVIEW

1.1 DESIGN RATIONALE

The design of the disassembler applications is based on dissecting the inputted binary code in a file into individual instructions consisting of 32 bits (4 bytes) each, in order to begin identifying the instruction and its parameters. The program first analyzes the first 3 bits to identify the instruction type, and hence pinpointing what and where its parameters are located within the instruction itself. Afterwards, through matching each segment of the instruction against the look-up table, the program outputs its equivalent in assembly code.

1.2 IMPLEMENTATION (Algorithms & Data Structures)

The application mainly consists of one key function:

```
int simulate(unsigned short)
```

and one supplementary function:

```
void regPrint(unsigned int, int).
```

The `simulate` function takes as a parameter a 2-byte binary instruction and is responsible for parsing, identifying its type, and disassembling it into its equivalent ARM THUMB assembly instruction. It does this by assigning the leftmost 3 bits to the int variable `fmt`, through bit-masking and anding, as it is the common denominator of all instruction types. The first 3 bits from the right are assigned to register rd in format 1,2,4,7,9. In format 3,6 register rd is assigned bits 8-10. In remaining formats (13,14,16,17,18,19) there is no register rd.register rb and rs are the same and they are assigned in formats 1,2,4,7,8,9 in bits 3-5 and in other formats doesn't exist.. There are multiple immediate values assigned in multiple formats offset5,offset8,word8,sword7 and are assigned different bit positions in different formats.

After extracting the first 3 bits into fmt and extracting the different previously mentioned registers, offsets,and immediates are used in the different formats after shifting right and masked by the appropriate number of bits for each variable. The masking was to ensure that we extracted exactly the number of bits required for each variable, and the masking consisted of the repetition of ones x the number of bits needed to be extracted.

The first thing to be checked was the first 3 bits and then after that different bits were noticed that differentiated between the formats. For example format 4-5 have different op- codes over bits 6-9 and 8-10. format s 6, 7 were realized from the 12th bit if it was 0-1 since they had the same first 3 bits. Formats 13,14 were realized from the 10th bits. Formats 18 and 19 were also realized from their 12th bit. After these first validations switch cases were made to identify the different instructions. In format 3,4,5 it was realized from the opcode .in format 7,9 it was realized from the L,B bits. In format 13 we define the instruction based on the s bit. In format 14 its based on L,R in format 16,17 its based on the cond. Bits .

ALGORITHM

- Format 1: after the initial first 3 bits case 1 (format 1) is established. And afterwards an op variable to handle switch cases inside the format if op(**Instruction 1** Regs Rs is left shifted logically by a value of the immediate offset5 and placed in rd. **Instruction2** regs rs is shifted right by a value of the immediate offset5 and placed in rd. **Instruction3** arithmetic right shift on rs by the value in immediate offset5 and placed into regs rd.
- Format 2: opcode variable is manipulated (bit 9) then offset 3 is derived by masking from offset 5 and set as a register value(rn). After a condition

is made if i flag (offset5 & 0x10) is zero then add rs to rn and set to rd . if it's 1 then add rs to immediate offset 3. The same process is carried out in subtraction after going to the else statement from format 1 (op!=3).

- Format 3: in this format the op is in bit 11 ,12 and instructions are handled through the value of op in switch cases add,substr,cmp,mv are carried out. Adding ,substr are regs rd+ immediate offset8. Mov instruction moves offset8 to the regs rd. and cmp \\
- Format 4: after setting the initial switch case then readjusting the opcode value to be 6-9 in format 4. The **first instruction** does a bitwise and between rd ,rs .**instruction 2** xor rs,rd. **Instruction 3,4** shifts rd by the value in rs. Left and right respectively. **Instruction 5** rsr. **Instruction 6** adds rd ,rs and a carry flag and sets them into rd. **Instruction 7** subtracts rs, !carry bit from rd and sets them into rd.**instruction 8** rotates right and if the right most bit is 1 rd gets shifted right by 1 and 1 is added from left of rd . if the right most bit is 0 automatically shifts right and 0 is added. **Instruction 9. Instruction 10** simply negates the values of rs and sets into rd.**Instruction 11, 12 . instruction 13** implements an or between rs and rd and sets into rd. **Instruction 14** multiplies rd and rs then sets the result to rd. **instruction 15** Implements and anding bitwise operation between rd and the not rs (!regs[rs]). **Instruction 16**
- Format 5(extra):
- Format 6: after initial previously mention validations there is no more validations needed since its only one instructions. Which implements a very similar algorithm used to load word in format 7 but instead adds to PC and loads into Rd register
- Format 7: After setting the LB 10-11 bits using masking and shifting we start making a switch statement handling all four instructions of the format. In the format we either set register rd into a new address in memory defined by adding register rb+ro or we set the memory that's defined by

rb+ro into rd. The implementation for handling store/load a word was made by shifting 8 bits and adding 1 to the index until we complete the word.

- Format 9:After setting the BL 11-12 bits using masking and shifting we start making a switch statement handling all four instructions of the format. In the format we either set register rd into a new address in memory defined by adding register rb(also rs) and offset5 which is an immediate or we set the memory that's defined by rb also(rs)+offet5 into rd.The implementation for handling store/load a word was made by shifting 8 bits and adding 1 to the index until we complete the word. In the case of the load we or to concatenate the bits.
- Format 13: after initial validations to identify the format and the instruction. , a #define sp reg[13] was made to define the stack pointer variable . Then the sword7 which is a 7 bit immediate is added or subtracted to it based on a conditional statement set by the S bit
- Format 14: a conditional algorithm is made by setting L, R bits 0,1. When L=0,R=0 then if these conditions are met a specified register(s) by rlink is pushed onto the stack. When L=0 R=1 Link register and any other register in rlist are pushed onto stack. L=1 R=0 pops values from stack into register set from the rlist .L=1 R=1 pops values from stack and loads them into registers specified by rlist.
- Format 16: after validation/extraction the cond variable . if it's not 15 then we initiate format 16. Switch case is made to accommodate all 15 cases that have branching instructions with each branching depending on the csrp codes and then updates the PC
- Format 17: after validation/extraction the cond variable If it's 15 then we initiate format 17.
- Format 18:

- Format 19: after initial validation is made based on the first 3 bits the H bit (bit 11) is used as a flag to take the upper 11 bits in the register that gets shifted 12 bits and then adds that value into pc counter and then the resulting address is set in LR. the second instruction when H=0 takes the lower 11 bits then shifts by 1 then is added to LR to have all the bits.
- Bonus: the bonus utilizes format 17 for the software interrupts where each value in value8 specifies what kind of action is taken.

Participation:

- We divided the instructions as follows;

  Farida handled the test cases(logic) for instruction formats 3,4,5,6, and 19.
  Mohamed handled the test cases(logic) for instructions 7,9,13.
  While Marawan wrote the test cases(logic) for instructions 14,16, and 17.

- As for modifying the main function, Farida set up the file stream to store the first two 32-bits as the initial values for the SP and PC, as well as handling the ‘0xDEAD’ delimiter.

- All members contributed to debugging the code.
- All 3 members contributed to this report.

2. SYSTEM OVERVIEW

2.1 CHALLENGES

- Getting used to identifying different formats by using different bits each time.
- Some bit flags in format 4.
- Understanding and properly implementing some instructions such “ror,tlt” in format 4.
- Understanding how push/pop works on the stack
- Properly implementing load/store words
- Simulating the branching instructions to properly branch, not the output of the simulator but the action of branching itself
- Changing the PC and SP values.
- Understanding how funct14 separated its registers.

2.2 DESIGN LIMITATIONS

- **Could not find the value8 values that correspond to reading an int, char and string an assumption was made which reading an int corresponds to 0x03, reading a char corresponds to 0x04 and reading a string corresponds 0x04**
- **We struggled to simulate the branch to jump to a certain part in the file.**