CO PROJECT - ARM SIMULATOR IN C++

MIDWAY CHECKPOINT REPORT

Team Members -

Aayush Aggarwal	(2016002)
Viresh Gupta	(2016118)
Baani Leen Kaur Jolly	(2016234)

Methodology -

We will build a simple ARM simulator that will implement a subset of the ARM Assembly language. We will include basic data processing / data transfer and branch instructions. The ARM code to be executed will be provided in a .mem file that will contain two hexadecimal integers in each line, first one being the address of the instruction and the second one being the instruction to be executed.

Project Plan -

We have to build a program that will take the input file, parse the input file and process the instructions in the five stages of Fetch, Decode, Execute, Memory and Write Back. Also another part of our project will be writing two programs to test our simulator with. These programs will be in a .mem file that will contain the direct machine code (Assembly code after conversion) in form of hexadecimal numbers.

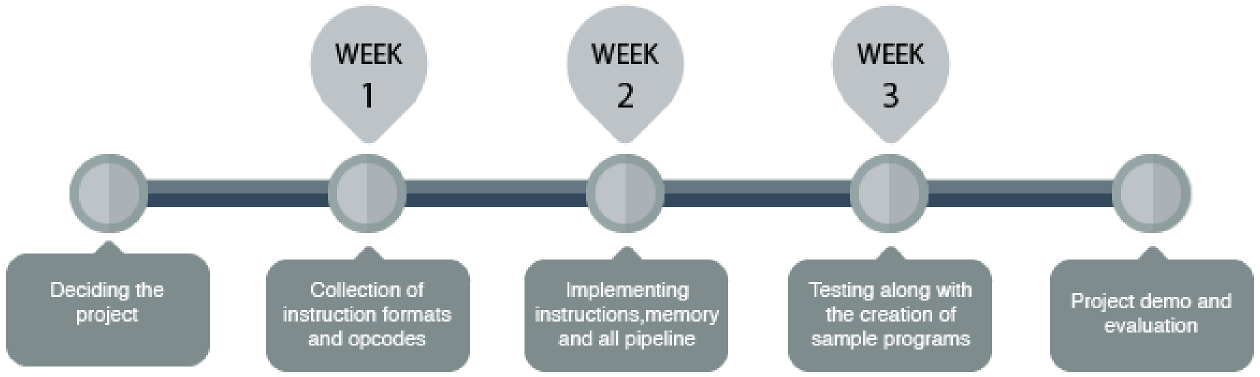
We will implement the basic data processing instructions (like add, sub, mov etc) , data transfer instructions (like ldr, str etc) and branch instructions (like beq, bne, blt etc).
For every instruction type, our simulator will support the corresponding functions and required software interrupts.

We will implement the memory as a linear array and the registers as another array which will store the register contents in their respective indexes. Since a register is 32 bit, and the data type of an int is also 32 bit, we can do this conveniently in C++.

We will also implement support for addressing modes and immediate operations (for 8 bits of immediate value).

Timeline :

Week 1 (30th Oct to 4th Nov) -	Collection of instruction formats and opcodes
Week 2 (5th Nov to 11th Nov) -	Implementing instructions,memory and all pipeline stages
Week 3 (12th Nov to 18th Nov) -	Testing along with the creation of sample programs


WEEK 1
WEEK 2
WEEK 3
Deciding the project
Collection of instruction formats and opcodes
Implementing instructions,memory and all pipeline
Testing along with the creation of sample programs
Project demo and evaluation

DOCUMENTATION

1. Instruction Set Format Summary

1.1 Data Processing / PSR Transfer

31 30 29 28	27	26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
Cond	0	0	I	Opcode	S	Rn	Rd	Operand 2

1.2 Branch

31 30 29 28	27	26	25	24	23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Cond	1	0	1	L	Offset

1.3 Software Interrupt

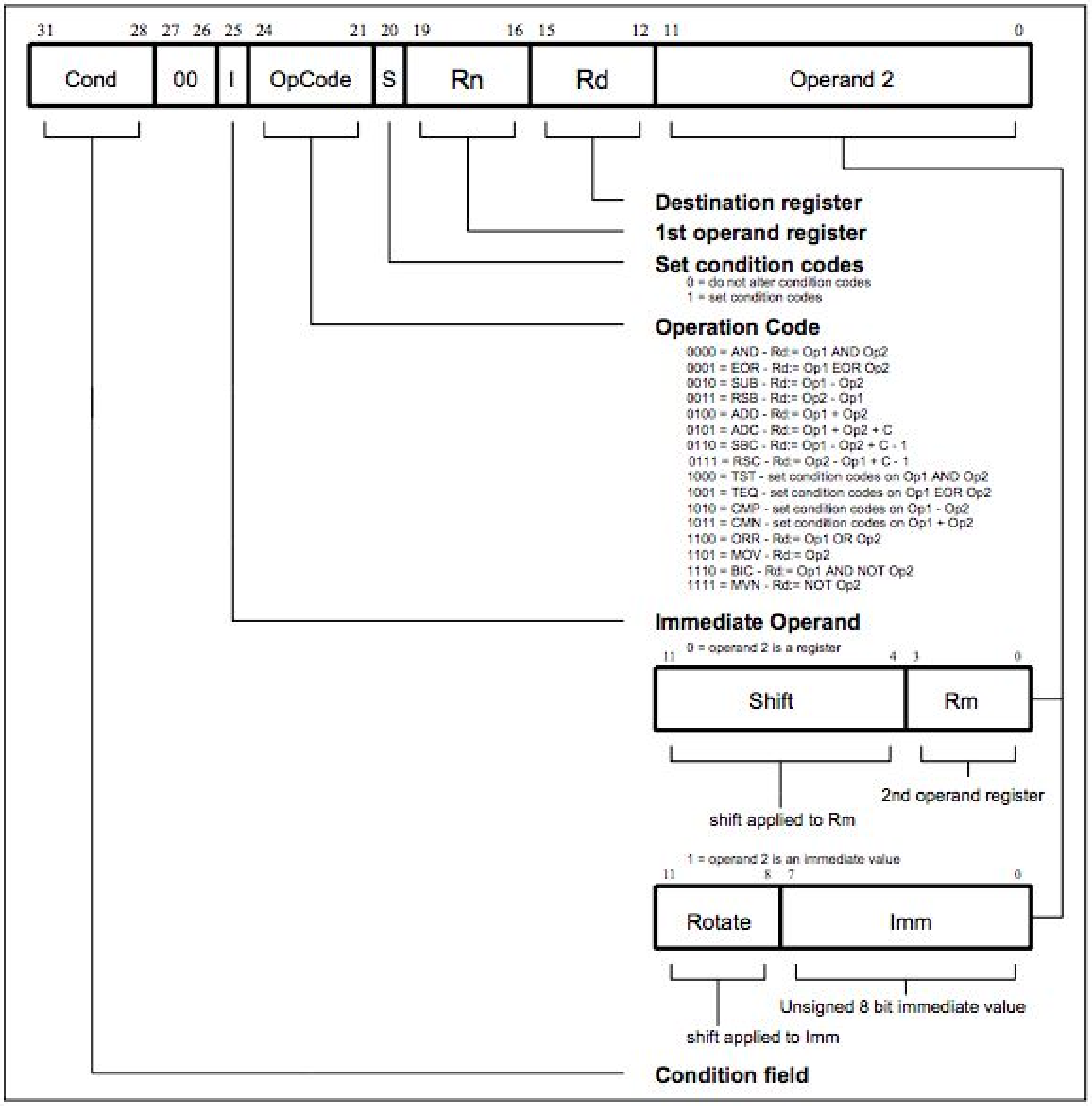
31 30 29 28	27	26	25	24	23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Cond	1	1	1	1	Ignored by Compiler

1.4 Single Data Transfer

31 30 29 28	27	26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
Cond	0	1	I	P	U	B	W	L	Rn	Rd	Offset

2. Data Processing

The instruction encoding in Data Processing is as given below :


31 28 27 26 25 24 21 20 19 16 15 12 11 0
Cond
00
I
OpCode
S
Rn
Rd
Operand 2
Destination register
1st operand register
Set condition codes
0 = do not alter condition codes
1 = set condition codes
Operation Code
0000 = AND - Rd:= Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2
0011 = RSB - Rd:= Op2 - Op1
0100 = ADD - Rd:= Op1 + Op2
0101 = ADC - Rd:= Op1 + Op2 + C
0110 = SBC - Rd:= Op1 - Op2 + C - 1
0111 = RSC - Rd:= Op2 - Op1 + C - 1
1000 = TST - set condition codes on Op1 AND Op2
1001 = TEQ - set condition codes on Op1 EOR Op2
1010 = CMP - set condition codes on Op1 - Op2
1011 = CMN - set condition codes on Op1 + Op2
1100 = ORR - Rd:= Op1 OR Op2
1101 = MOV - Rd:= Op2
1110 = BIC - Rd:= Op1 AND NOT Op2
1111 = MVN - Rd:= NOT Op2
Immediate Operand
0 = operand 2 is a register
11 4 3 0
Shift
Rm
shift applied to Rm
2nd operand register
1 = operand 2 is an immediate value
11 8 7 0
Rotate
Imm
shift applied to Imm
Unsigned 8 bit immediate value
Condition field

The instruction performs the respective arithmetic or logical operation(based on the Opcodes given above) on one of two operands. The first operand is always a register while the second one can either be a register or a rotated 8 bit immediate value.
If the second operand is a shifted register, the operation is controlled by the Shift field in the instruction, which decides the type of shift to be performed, i.e. a logical left shift or a right shift, etc.)

Eg.

```
ADD     r2,r1,r3
```

The ARM command calculates the sum of the values in r1 and r3 and places it in r2.

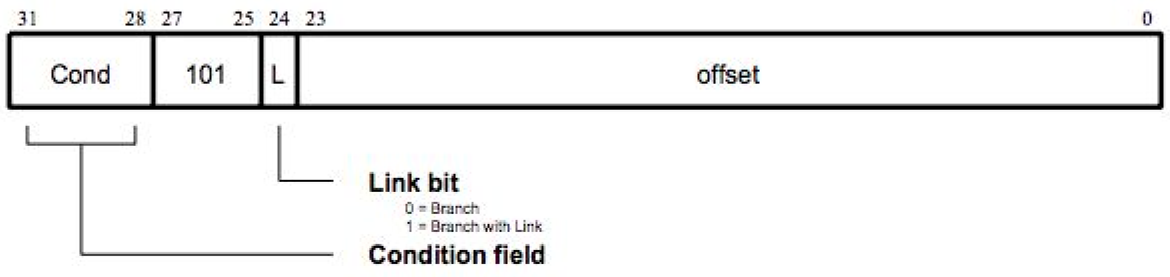
The hexadecimal code for this command is 0xE0812003, i.e.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1

As, can be noticed, here the opcode (from bit 24 to 21) is 0100 (for left to right) indicating the addition operation.

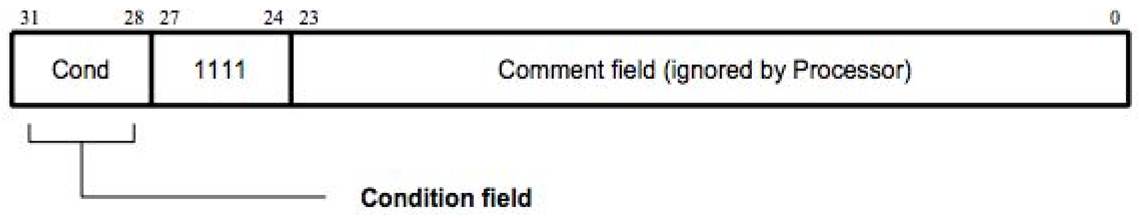
3. Branch and Branch with Link

The instruction encoding for Branch and Branch with link is given below:



The Branch instructions contain a 24 bit offset (2's complement representation), a 1 bit Link, to indicate if the Branch is with link or not.
The 24 bit offset is extracted from the instruction word, sign extended to 32 bits, shifted left by 2 bits.
In case of a Branch with link instruction, the current PC is written in the R14 register or the link register.
Then the PC is updated by adding the offset to it, to move the Program Counter to the address of the next instruction.

4. Software Interrupt

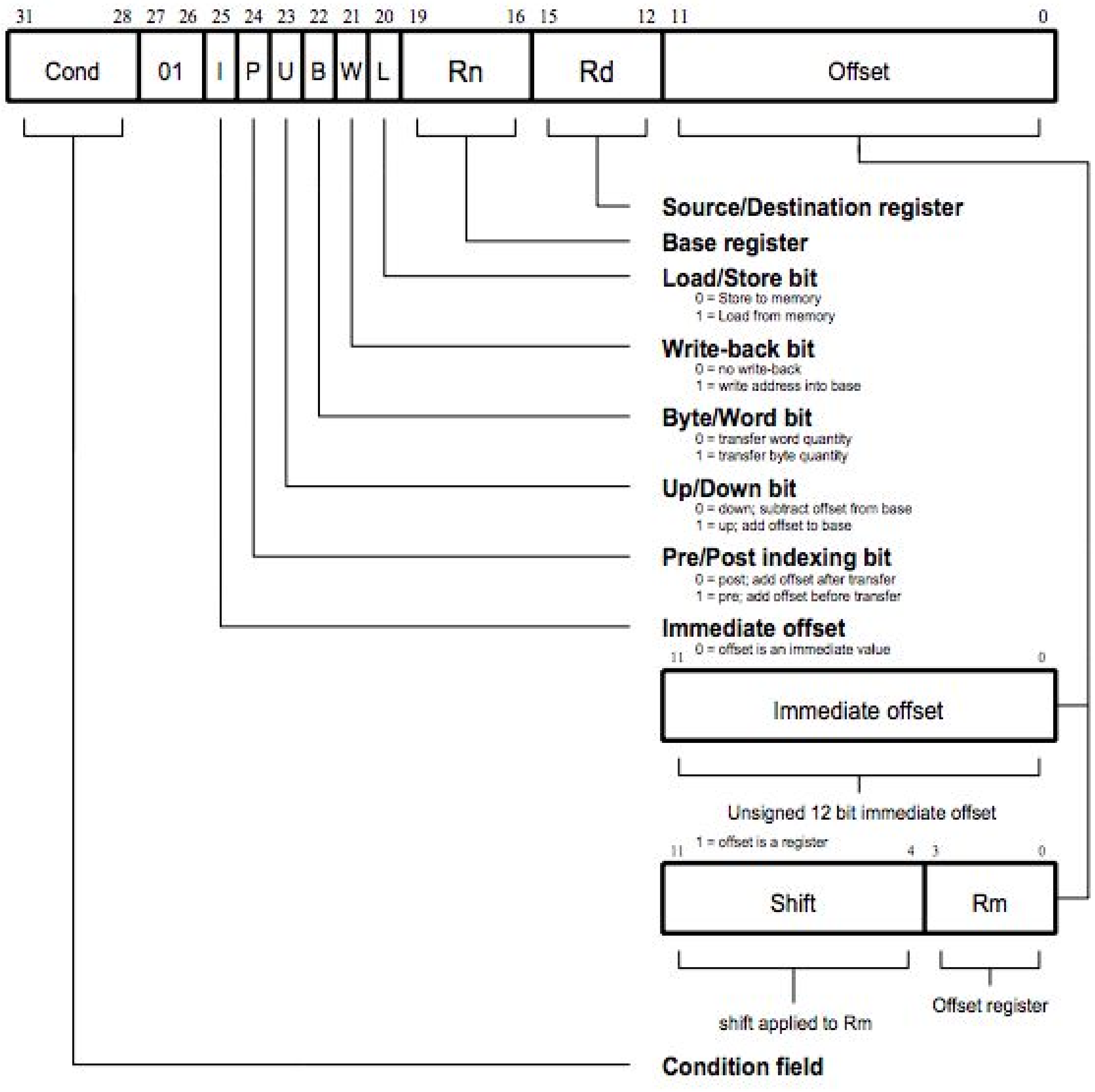


It is used to enter the Supervisor mode. The PC is forced to a fixed value (0x08).
The program counter is stored in R14_svc upon entering the Software Interrupt trap and the PC points to the word after the SWI instruction. After execution, the value of R14_svc can be moved to the PC to return to the calling program.

The right most 24 bits are ignored by the Processor and can be used to convey information to the supervisor.

In our simulator, since we lack co-processor, we have implemented software interrupts using direct calls to underlying programming languages methods for input output.

5. Single Data Transfer (LDR, STR)



The single data transfer is used to load or store data, i.e. move the data from the registers to the memory, or vice-versa.The address of the memory location is calculated by adding or subtracting the offset from a base register.

Eg.

```
LDR     r0,=Table
```

The ARM command places the address of the location of variable Table in r0.

The hexadecimal code for this command is 0xE59F0014, i.e. (Assuming location of Table variable is at 8 bytes after the ldr command).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	1	1	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0

6. Implementation Details

Firstly, we have read all the instructions and loaded them into an instruction memory, using the load_word method.

We have implemented our project in 5 stages, fetch, decode, execute, memory and write back, just like the 5 stage procedure in the ARM instruction set.

- **Fetch:** The instruction is loaded from the instruction memory using the load_word routine, using the program counter register, and gets stored in a global variable.

- **Decode:** The instruction gets processed:
  We check whether the instruction is an ALU instruction, single data transfer, branching instruction, or software interrupt.
  Depending on the the instruction we get, we use bit manipulation to retrieve the source, destination, and immediate value(or register) along with the shift amount to be applied. Afterwards, we use the opcode to figure out exactly what the instruction really is. To avoid code repetition, we are maintaining a lambda function to assign what is to be done in the execution stage (if applicable), memory stage(if applicable), and write-back stage (if applicable).

- **Execute:** The instruction gets executed according to what was processed in the decode stage, and how the lambda was rigged. We simply call the lambda to execute whatever is applicable, do necessary logic and arithmetic, and store the result into the global destination register gained from the decode stage.
  The lambda is a void function in case the instruction doesn't have the execute stage.

- **Memory:** The memory operation gets executed if the instruction is of such type. If the decode operation had a memory operation, the mem_func lambda gets rigged and it uses the load_word or the store_word method, whatever applicable, and reads/writes the register value from/to the memory starting from the given base register and offset (after shift).

- **<u>Write Back:</u>** After the write execute or memory operation, if any changes are to be written to some register, that is done in this stage. We have implemented this stage using a global variable for a destination register (index) and a value, and rigging a lambda function to write the resultant value into the destination register.