Computer Architecture and Organization

CSN-221

Project Report

Team members:

20114091	Siddharth Singh Rana
20114036	Divyesh
20114021	Bashar Ahmed
20114016	Arpeit Chourasiya
20114087	Shikhar Agrawal
20114007	Alok Raj
20114033	Dheravath Vikas
20114050	Lokendra Singh Parmar

<u>Under the supervision of:</u>

Prof. Peddoju Satish Kumar

Problem statement

The basic idea of this project was to implement a CPU that includes ALU,register files,control circuitry, instructions flow etc on logisim simulator. Rigorous benchmark evaluation on different instructions has to be performed while working on this project.

Our solution

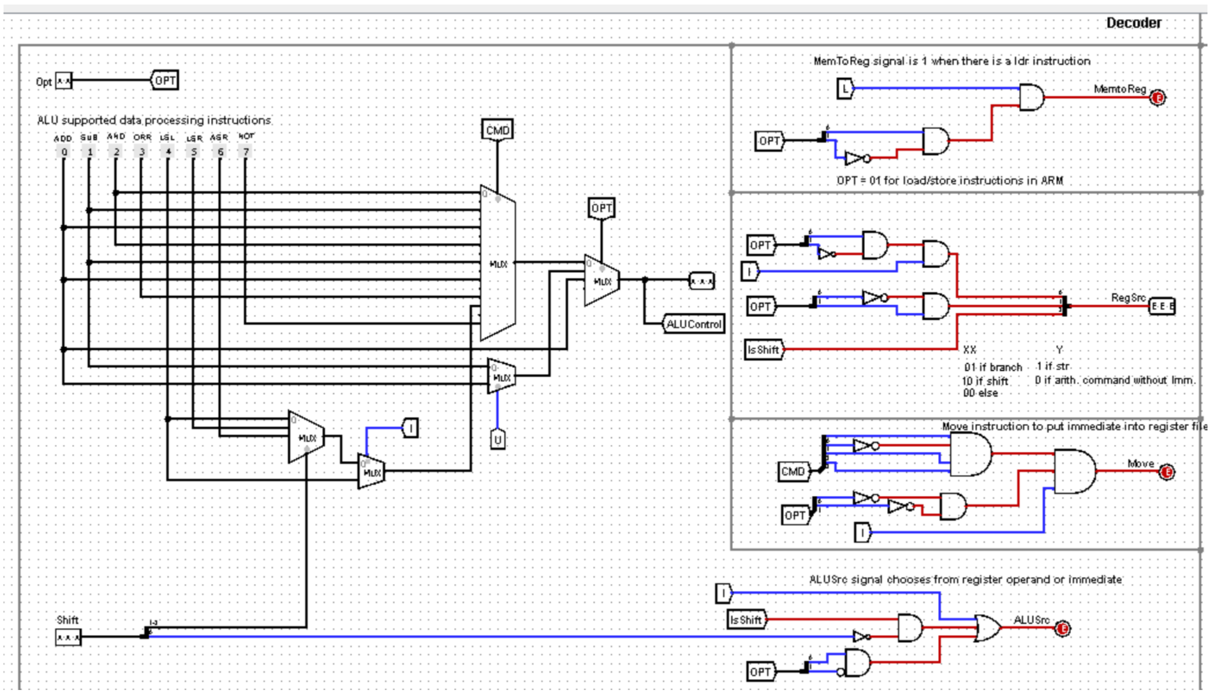
We designed a CPU using the ARM instruction set. Different components of the CPU such as ALU (Arithmetic and Logic unit), control unit, instruction memory, data memory were designed in the Logisim simulator. Also, benchmark evaluations were done regularly in the Logisim simulator itself by testing the processor on some custom instructions.

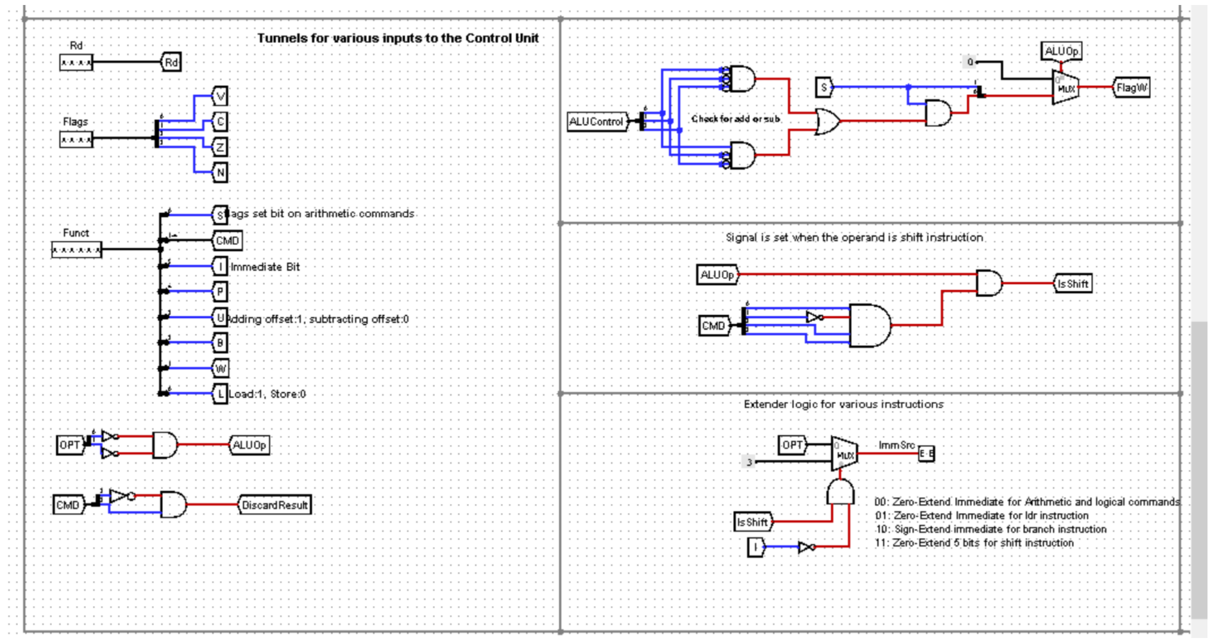
The following report describes the various Architectural implementations and the flow of instructions and data with all the diagrams being made by us as Drawings or Screenshots of our implementation to help describe the same.

The novelty of the work done:

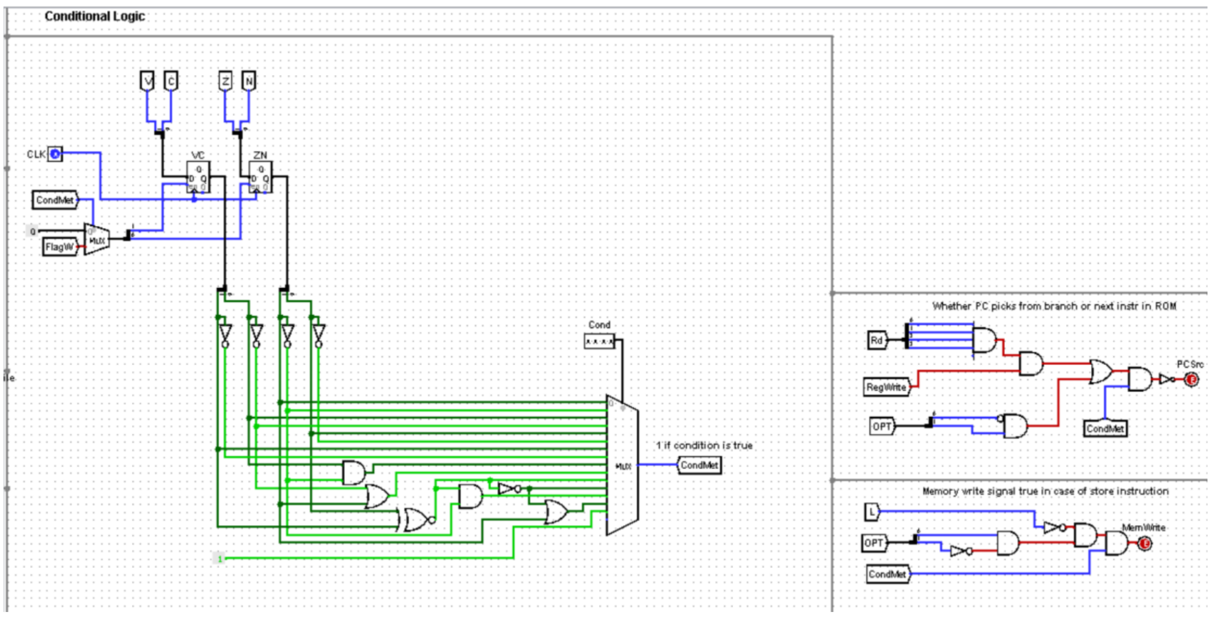
- Designing the control unit for ARM processor.
  We have designed the control unit of our processor from scratch. Although, the basic idea of having a Decoder and conditional logic was adopted from the book Digital design and Architecture:-ARM edition, the combinational logic of various control signals was designed and implemented from scratch. Benchmark testing was rigorously performed over the course of the project which played a crucial role while designing the control unit.
  The following images are the design of Decoder and Conditional Logic of our Control unit.

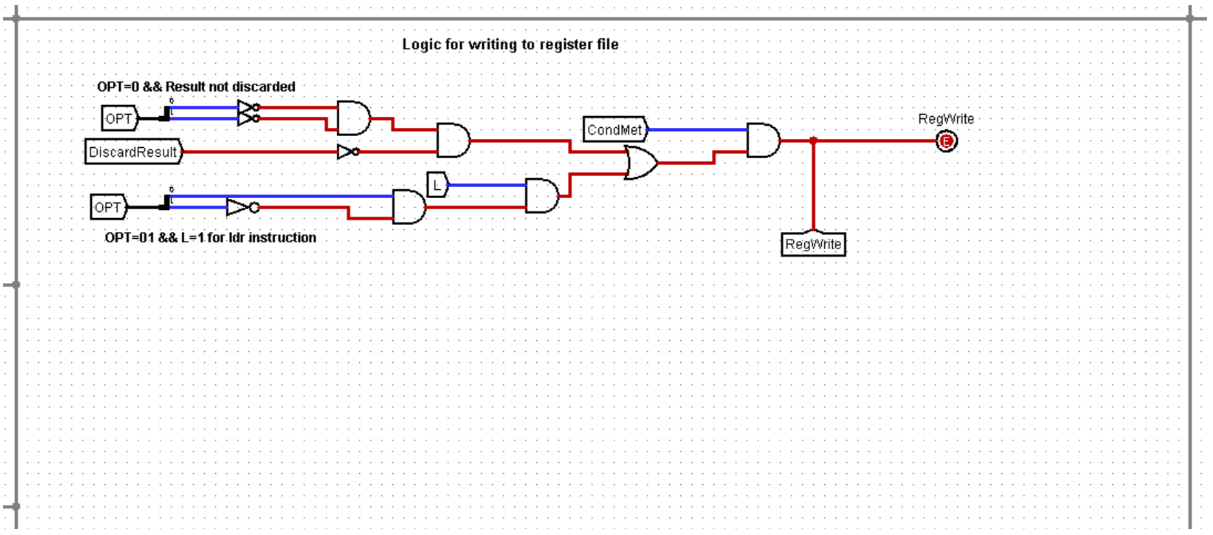
Decoder of control unit


Decoder
Opt
OPT
ALU supported data processing instructions
ADD SUB AND ORR LSL LSR ASR NOT
0 1 2 3 4 5 6 7
CMD
MUX
ALUControl
I
U
Shift
MemToReg signal is 1 when there is a ldr instruction
L
MemtoReg
OPT = 01 for load/store instructions in ARM
IsShift
RegSrc
XX
01 if branch
10 if shift
00 else
Y
1 if str
0 if arith. command without Imm.
Move instruction to put immediate into register file
Move
ALUSrc signal chooses from register operand or immediate
ALUSrc


Tunnels for various inputs to the Control Unit
Rd
Flags
V
C
Z
N
Funct
S
Flags set bit on arithmetic commands
CMD
I
Immediate Bit
P
U
Adding offset:1, subtracting offset:0
B
W
L
Load:1, Store:0
OPT
ALUOp
DiscardResult
ALUControl
Check for add or sub
FlagW
Signal is set when the operand is shift instruction
IsShift
Extender logic for various instructions
ImmSrc
00: Zero-Extend Immediate for Arithmetic and logical commands
01: Zero-Extend Immediate for ldr instruction
10: Sign-Extend immediate for branch instruction
11: Zero-Extend 5 bits for shift instruction

Conditional logic



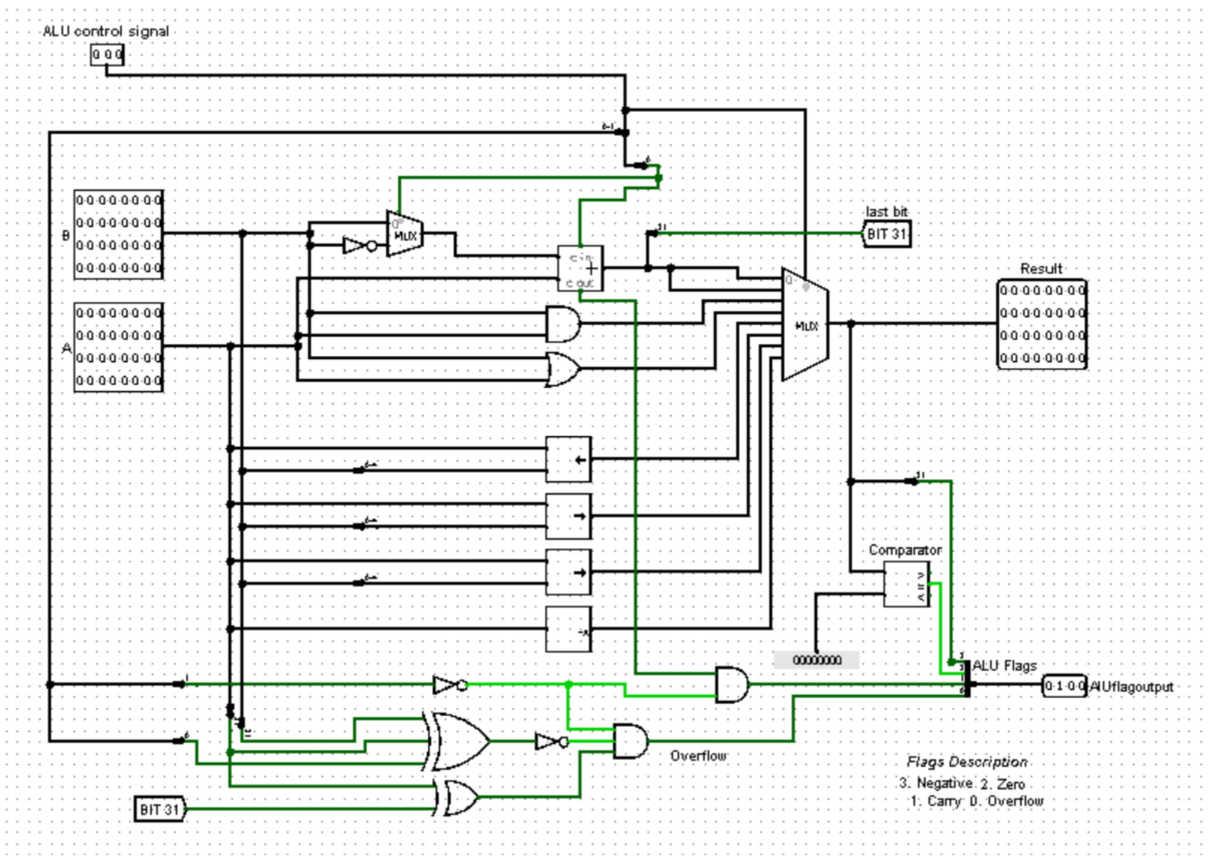

Logic for writing to register file
OPT=0 && Result not discarded
OPT
DiscardResult
CondMet
L
OPT
OPT=01 && L=1 for ldr instruction
RegWrite
RegWrite

- Designing a custom ALU for ARM instruction set

  Custom ALU is designed for the ARM processor which performs the following functions.
  - AND
  - OR
  - NOT
  - ADD
  - SUB
  - LSL
  - LSR
  - ASR

  4 Flags are used for the ALU output
  - Carry Flag
  - Zero Flag
  - Overflow Flag
  - Sign Flag

The following image is our ALU design for the ARM processor.

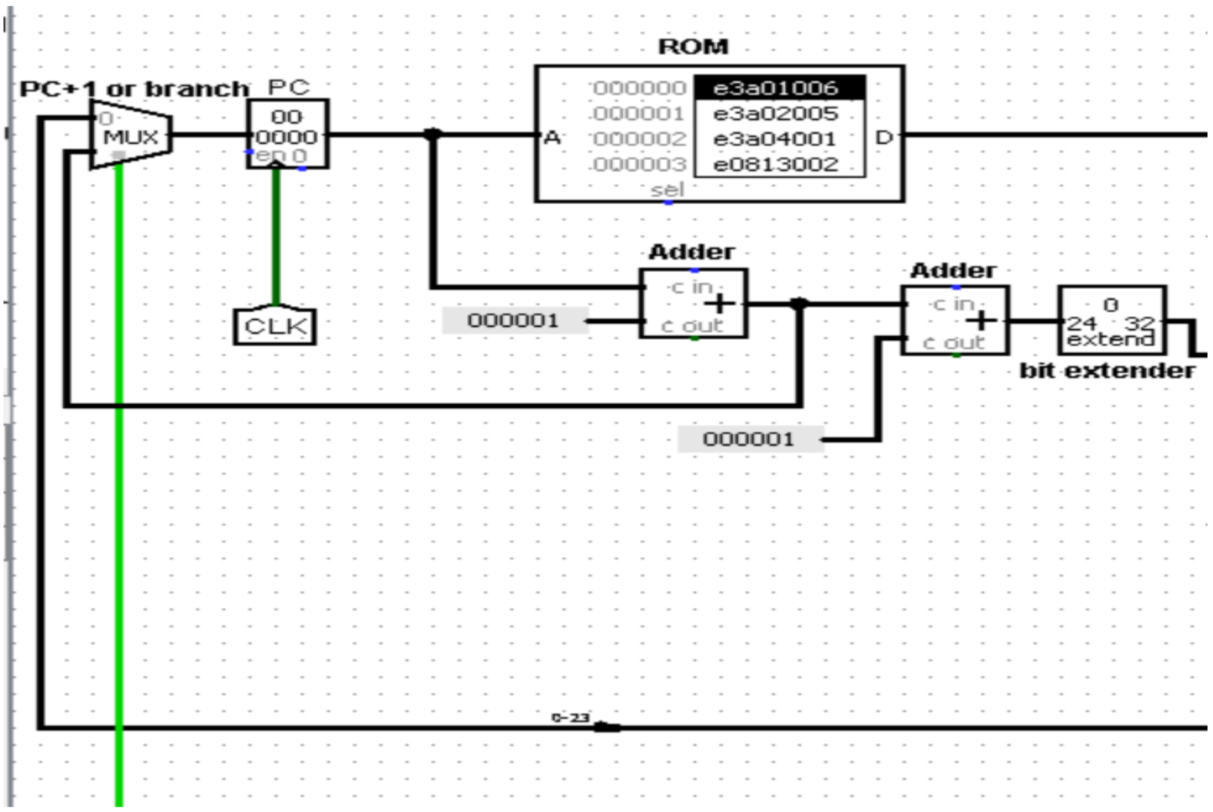


Methodology and Working

This project is made out of ARM instruction set, which is a standard instruction set architecture. We adopted rigorous evaluations and deliberated over many issues to make this processor work for a handful but enough number of instructions available in the ARM ISA to support basic operations in a processor.

Firstly, I'll explain the working of the main file of the processor one by one.

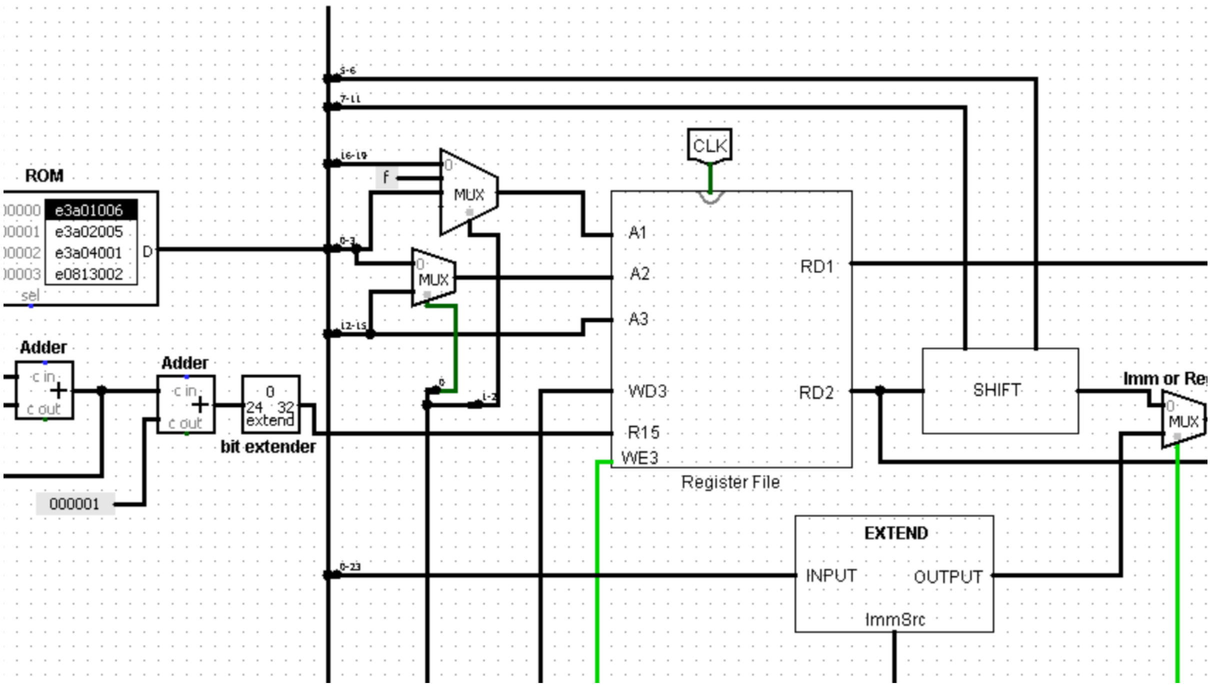
Program Counter and Instruction Memory(ROM):



The program counter has been designed to keep two things in mind. Firstly, it might be used to move to the next instruction, in that case, PC<-PC+1, and also might move to some other address in case of a branch instruction. The MUX which decided between these two receives a PcSrc signal from the control unit, which is 1 in case of normal instruction and 0 for a branching instruction.

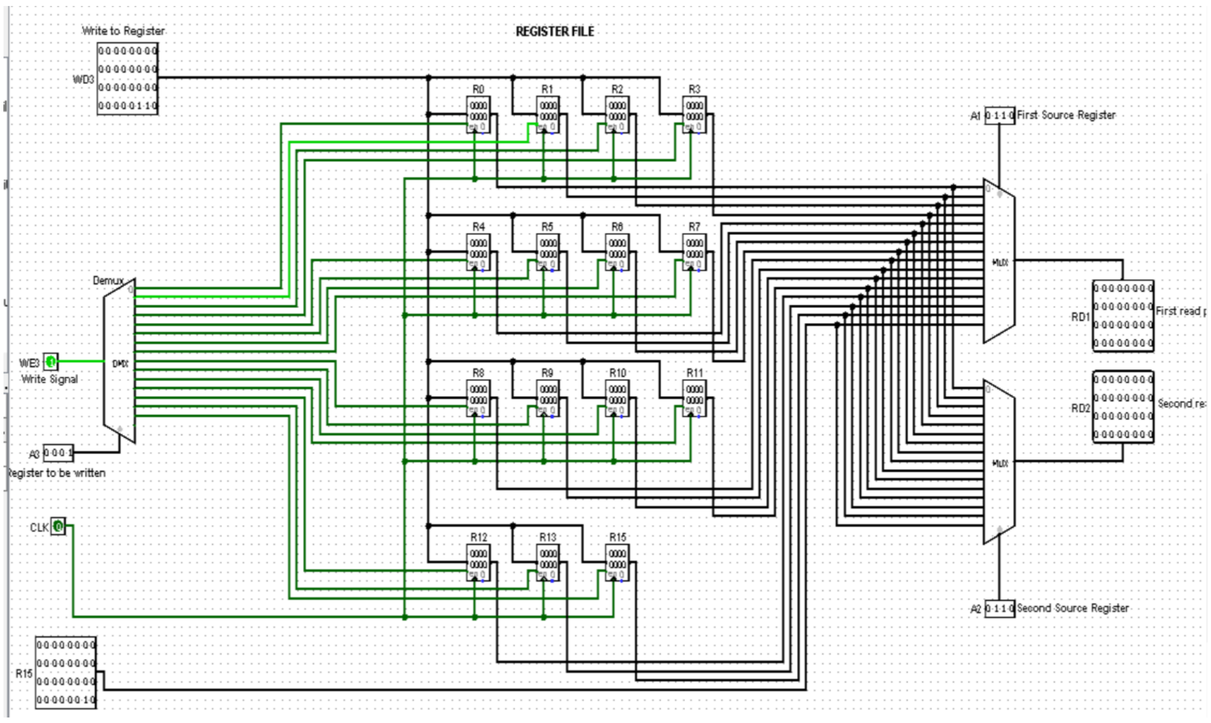
The ROM holds some instructions for showing the execution of the implemented ISA.

Register File:



The register file consists of two ports for the two source registers, A1 and A2. The register input A1 is chosen from the instruction bits 16-19 in case of arithmetic instructions, f(15 in hexadecimal) in case of writing to RD15(program counter register), and 0-3 in case of a shifter operand. The MUX handling this receives signals from the **RegSrc** in control unit. The register input A2 receives signals from **RegSrc** in the control unit and takes input from 12-15 in case of a store and

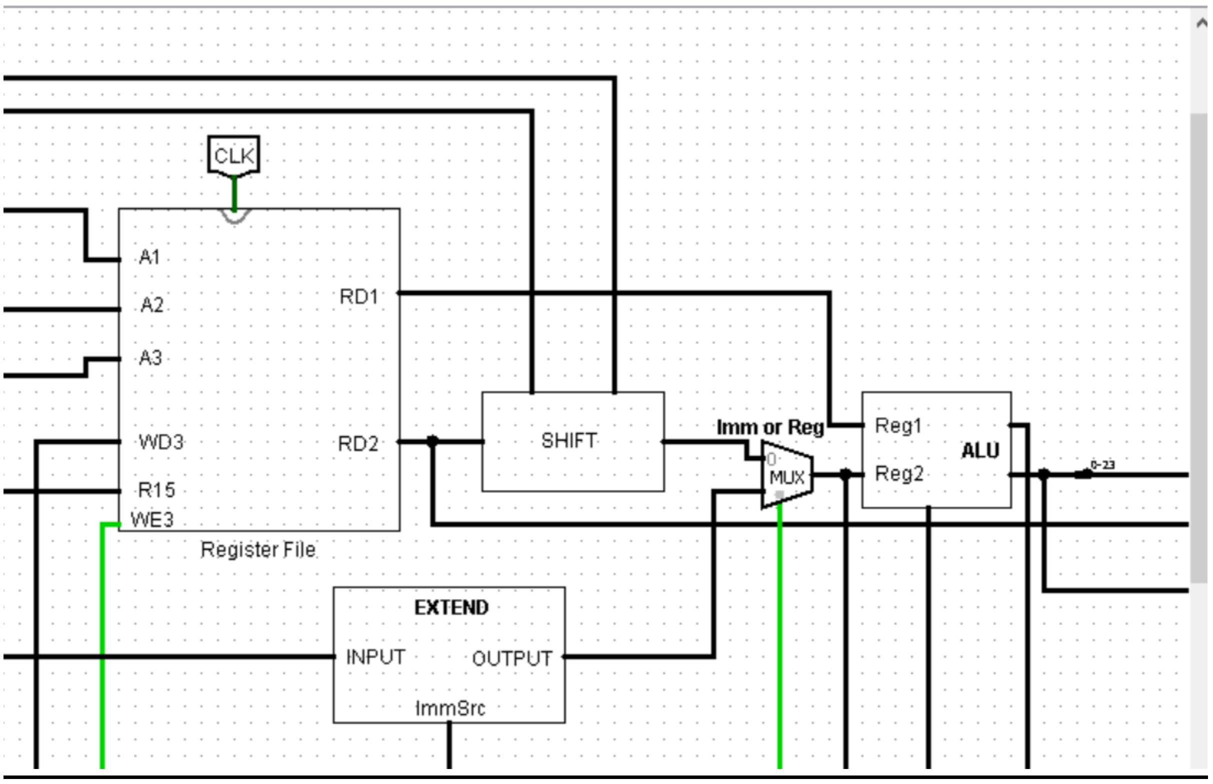
0-3 in case of a data processing instruction. The A3 port takes in the 12-15 bits denoting the register to which the data must be written in case of a write operation. WD3 is the data to be written which comes to the register file after the data is fetched from memory or the calculated ALU result is taken to the WD3 port for storing to a register. In case of a load, the WD3 receives values from the RAM(data memory), in case of data processing instruction, the ALU result goes into this port and in case of a move instruction, the immediate or the data in the second source register is moved to the destination register. The following is the implementation of the register file.



Explanation:

WD3 contains the 32 bit data to be written to some register(R0-R14) in case the WE3(write to register) signal is 1. The demultiplexer at the left chooses the select bits from A3(register to be written to) in case WE3 is active and is passed to the enable input of all the registers. In case of a write command, the data in a particular register changes and is passed as an output to the two read ports, RD1 and RD2, which are selected by respective multiplexers with select bits coming in from the ISA encoding (A1 and A2). In case of a normal read operation, the WE3 is set to 0 and data from registers goes through the output of register to ports RD1 and RD2. The R15 register contains the PC+2 since in ARM ISA, reading the R15 gives PC+2*(general offset). In case of a branch instruction, the R15 has to be read via the A1 port and its data must be passed on to the PC in the next cycle, hence forming branch instructions.

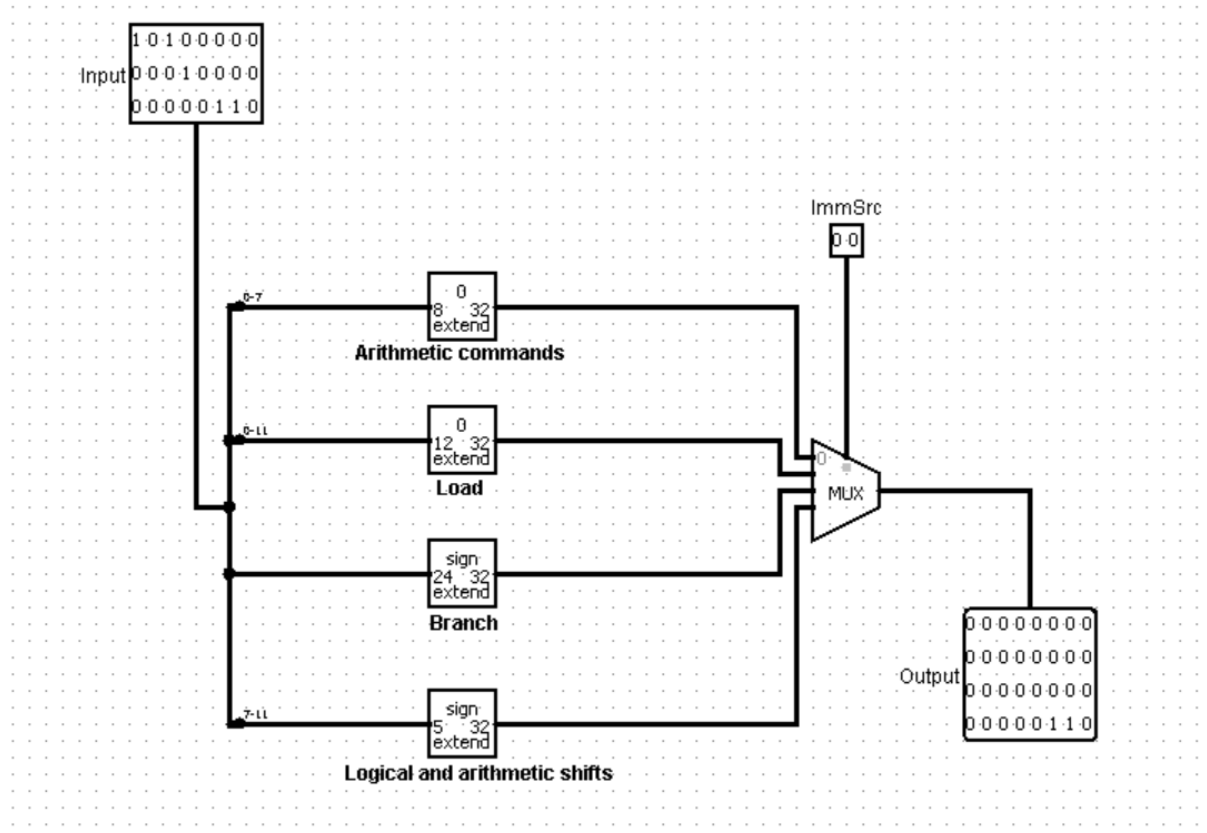
<u>Extender, Shifter and ALU:</u>



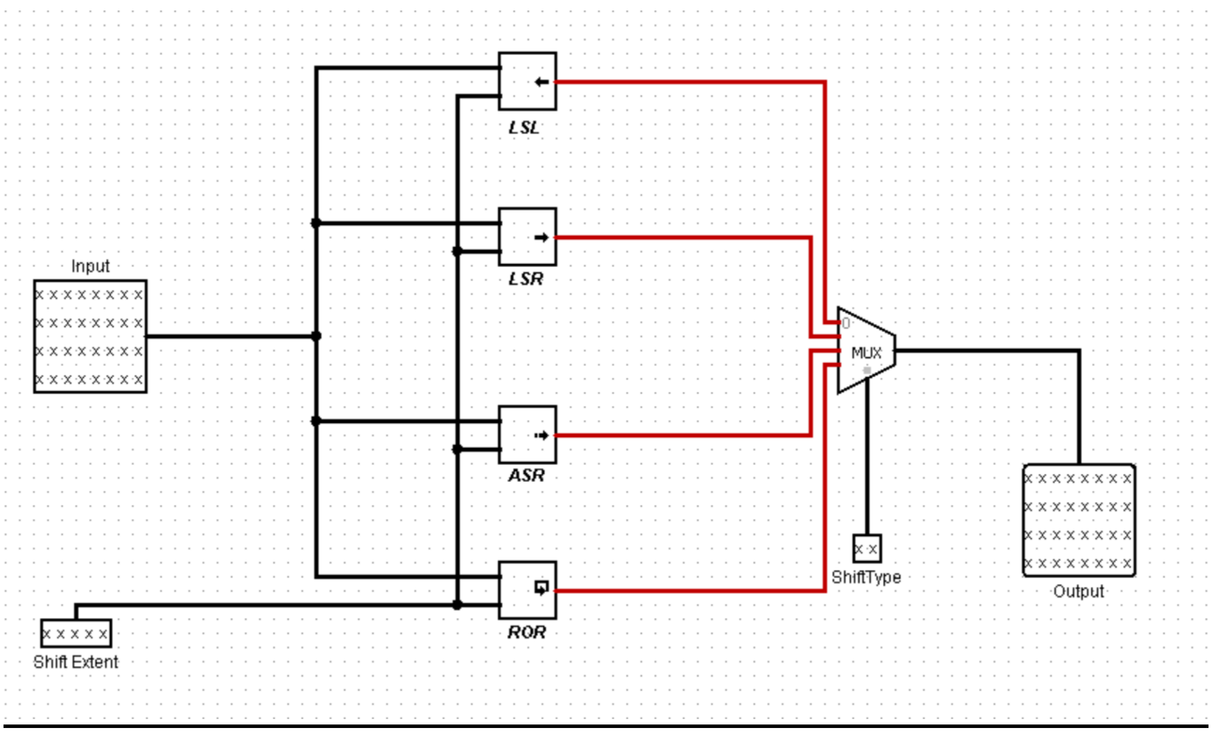
The first source register is the first input to the ALU of the processor. The second input might be an immediate operand or a register operand. If the second source is a register operand, it might have some shift immediate associated with it(bits 7-11 show the extent of shift and bits 5-6 show the type of shift(lsl,lsr,asr,ror), hence the value is passed through a shifter to get the final result out. In case of a store, the value in the destination register is selected via A2 and passed through the RD2 port and into the data memory. The MemWrite signal into the data memory is enabled to 1 in order to store the destination reg data into the source data. In case the second source is an immediate, it is 0 extended to

32 bits in case of data processing instruction, sign extended for a branch instruction, and the 12 bit offset is 0 extended for a ldr instruction. This immediate is passed to the Reg2 input of the ALU and the ALU result is prepared inside the ALU.

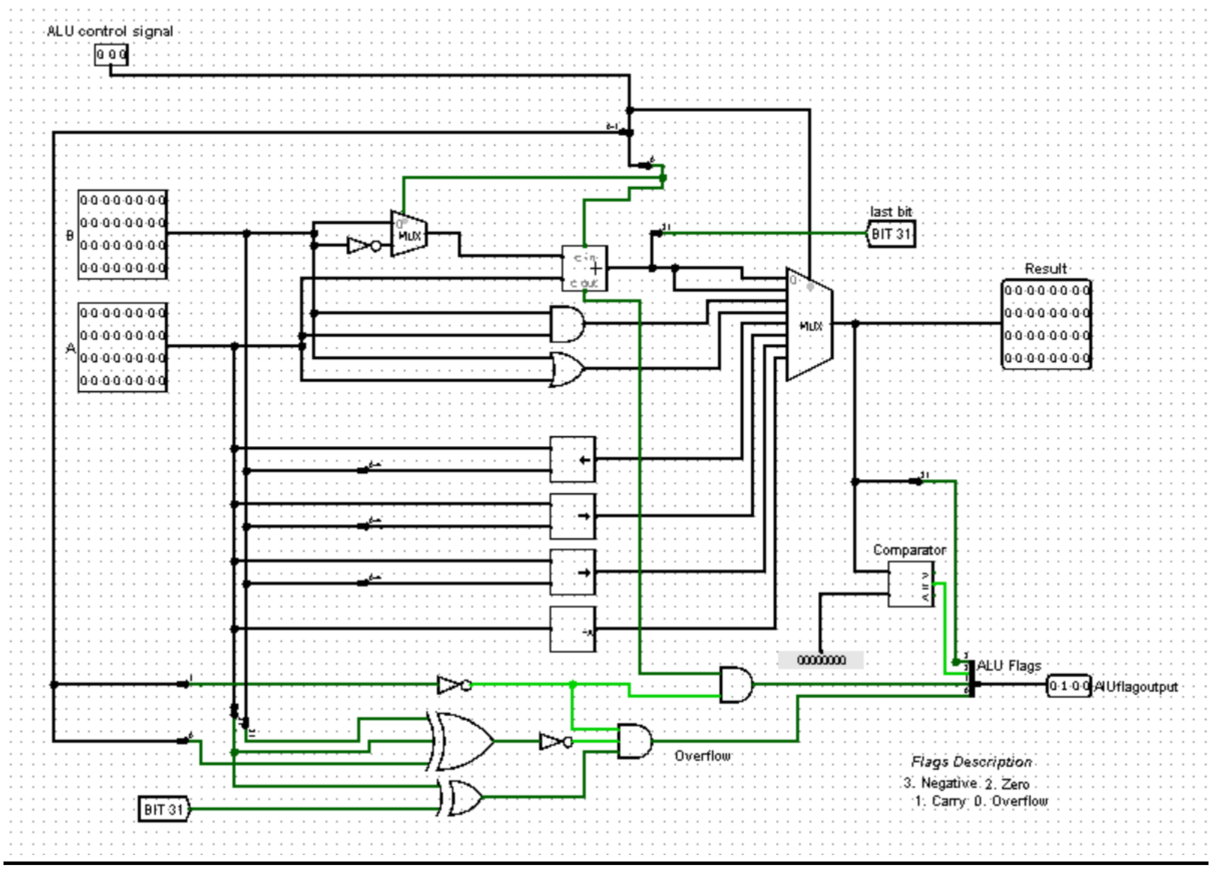
BIT-Extender(for immediate second source operand):



Shifter:


Input
LSL
LSR
ASR
ROR
MUX
ShiftType
Output
Shift Extent

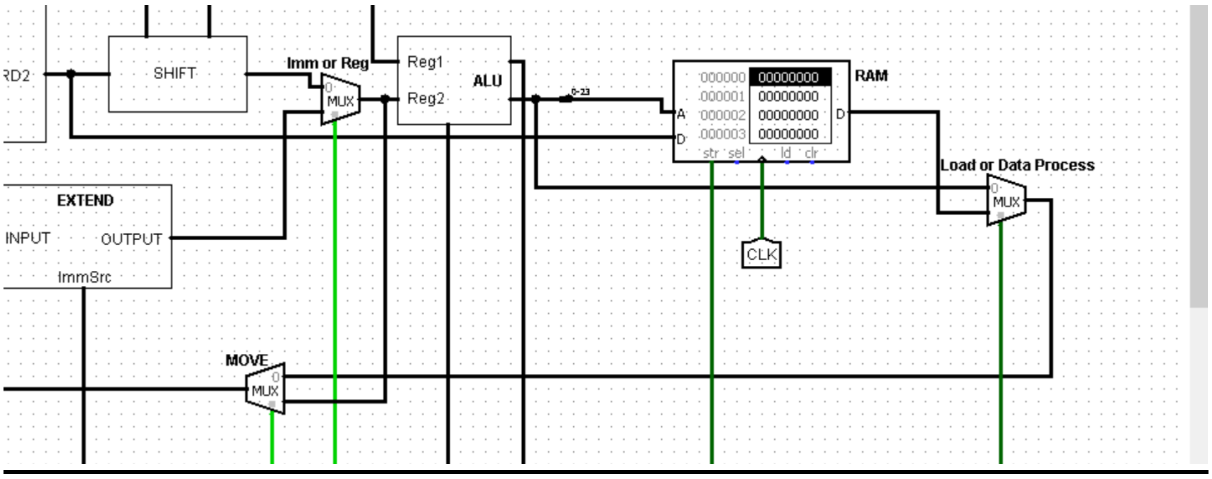
ALU:


ALU control signal
B
A
MUX
last bit
BIT 31
Result
Comparator
00000000
ALU Flags
AlUflagoutput
Overflow
BIT 31
Flags Description
3. Negative 2. Zero
1. Carry 0. Overflow

EXPLANATION:

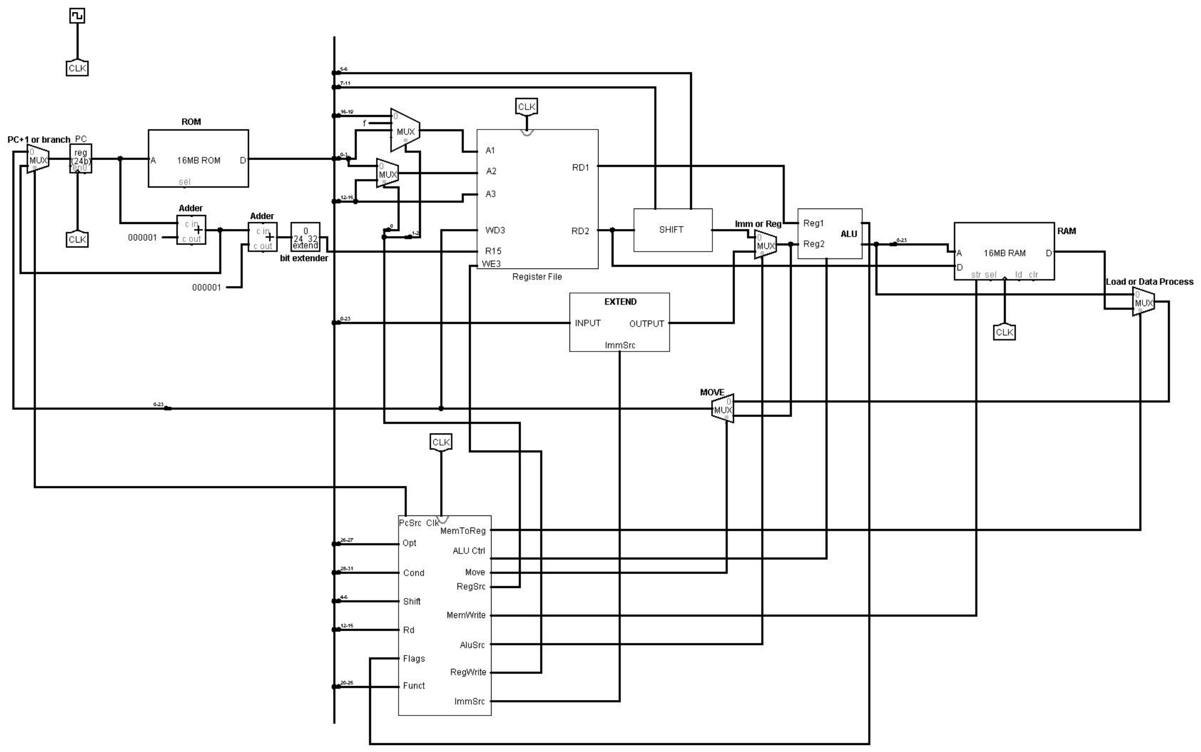
The 0th bit of ALU control signal is used as a select bit for the uppermost MUX to check if its addition or subtraction(take 2s complement of B). The BIT 31 tunnel signifies the 31st bit of the ALU result and is used to check for the negative flag(N flag). Other ALU outputs might be chosen from AND, OR, LSL, LSR, ASR, NOT(complement). The zero flag(Z flag) is checked by comparing the ALU result to 0 using a comparator.

Final Result:



The Load or data processing MUX chooses from the data memory and the ALU result respectively. The data memory has a store signal which comes from **MemWrite** and is used for str command to be executed. The output is passed to a MUX finding if it is a simple MOV instruction or needs the ALU output. This is then sent to WD3 port of register file which writes to the register in case WE3 signal is 1. This completes the methodology and explanation of the whole circuit.

The main circuit is given below:-



INSTRUCTION ENCODING:

General Format: -

Cond	Type	
32 29	28 27	

Cond represents the condition which needs to be evaluated before the instruction must be executed.

Type refers to the instruction type- whether it is (ldr, str), branch or data processing instruction.

Data processing instruction: -

cond	00	I	opcode	S	rs	rd	Shifter operand/ immediate
32 29	28 27	26	25 22	21	20 17	16 13	12 0

For ALU instructions, type is 00, I is the immediate bit to tell if the second operand is shifter or an immediate, opcode is used for the instruction code, S is the suffix bit for setting CPSR flags, rs is the first source register and rd is the destination register.

Encoding Shifter operand: -

Shift immediate	type	0	rt
12 8	7 6	5	4 1

Shift immediate		type	1	rt
12 9		7 6	5	4 1

Load/store instructions: -

cond	01	I	P	U	B	W	L	rs	rd	Shifter operand/immediate
32 29	28 27	26	25	24	23	22	21	20 17	16 13	12 0

Here, the type is 01 and immediates have to be zero extended from 12 to 32 bits.

<u>Branch instructions: -</u>

cond	101	L	offset
32 29	28 26	25	24 0

The offset is first expanded to 32 bits by sign extension and then added to the PC+2 for generating branch target

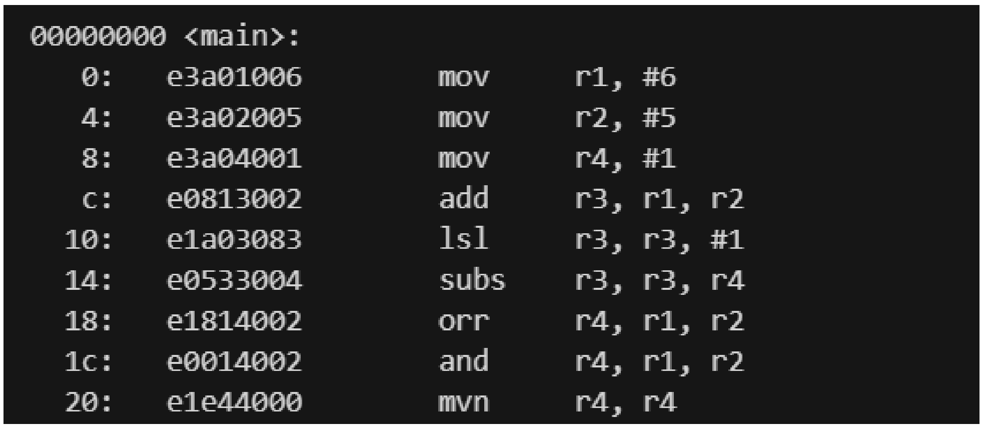
Evaluation parameters and approach:

For benchmark evaluation, custom instructions were written and tested rigorously in the logisim simulator itself. We were able to successfully run the simulations on the logisim simulator and rectify the various errors which we made over the course of the project. We did a ‘write’ of the hexcode of custom instructions in the instruction memory to perform benchmark evaluation.

The following tests were performed to meet the purpose:

- Tests for Data-processing instructions

Instr. Hexcode Instruction



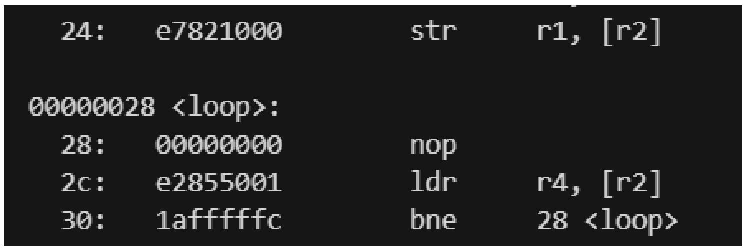
```
00000000 <main>:
   0:   e3a01006        mov     r1, #6
   4:   e3a02005        mov     r2, #5
   8:   e3a04001        mov     r4, #1
   c:   e0813002        add     r3, r1, r2
  10:   e1a03083        lsl     r3, r3, #1
  14:   e0533004        subs    r3, r3, r4
  18:   e1814002        orr     r4, r1, r2
  1c:   e0014002        and     r4, r1, r2
  20:   e1e44000        mvn     r4, r4
```

In the above instructions, we first store immediate values in registers r1,r2 and r4.

Further we implement the ALU instructions and compared the contents of destination register with the expected result after implementing the given above instructions.

- Load-store and branch instructions

Instr. Hexcode Instruction



```
  24:   e7821000        str     r1, [r2]

00000028 <loop>:
  28:   00000000        nop
  2c:   e2855001        ldr     r4, [r2]
  30:   1afffffc        bne     28 <loop>
```

The str instruction stores the contents of r1 in memory location in r2 register. After performing this instruction, we observed that in the instruction memory,the value stored in r1(00000006) was stored at memory address 00000005(this value was stored in r2).

The ldr instruction loads the contents of memory location in r2 register into the r4 register.

After the ldr instruction, contents of r4 register were 00000006. ‘00000006’ was stored in memory location

‘00000005’ which was stored in r2 register.

Hence we verified both the load as well as store instructions.

Branch instruction ‘BNE’ will move the control to instruction no. 28, if the zero flag is clear.Hence in our case, the control shifted to instruction no. 28, and hence we verified the branch instruction in this manner.

Individual contributions:

<u>Divyesh (20114036)</u>

- Design the ALU for the processor.
- Contributed to the design of Decoder section of control unit.
- Wrote custom instructions for testing, therefore serving the purpose of benchmark evaluation.
- Made the project report.

<u>Siddharth Singh Rana (20114091)</u>

- Design and implement the complete Register File.
- Implement the logic of the main circuit to assemble components and control signals at the right places.
- Contributed to the design of control unit.
- Methodology of the report.

Bashar Ahmed (20114021)

- Implemented and Designed the Control Circuit
- Implemented the Extender
- Designed the Shifter
- Helped in the simulation and correctness of the circuit

Lokendra Singh Parmar (20114050)

- Contributed to design for the ALU for the processor
- Contributed to writing custom instruction for testing
- Made project presentation

Alok Raj (20114007)

- Contributed to designing of shifter
- Made project report

Shikhar Agrawal (20114087)

- Contributed to implementing the logic of main circuit to assemble components and control signals.
- Made project report

Arpeit chourasiya (20114016)

- Contributed to designing of extender
- Made project presentation

Vikas Dheravath (20114033)

- Made project report
- Contributed to designing and implementation of register file.

Results and Discussions

So, in totality, our project was to design a processor with all the essential features and working. For this we had chosen the ARM instruction set (RISC). Our final circuit is quite elaborate with many finer circuits integrated into it, such as the ALU (Arithmetic Control Unit), Control Circuit, Register File, Extender and Shifter.

Our processor is capable of executing the following instructions in 24-bit format with 32-bit data:

- Branch Instructions
- LOAD & STORE Instructions
- ALU Instructions:-
    - ADD
    - SUB
    - AND
    - OR
    - NOT
    - LSR
    - ASR
    - LSL

Our project also includes simulated testbenches for some the instructions for evaluation purposes. These instructions are (with their respective hexcodes):

- mov
- add
- lsl
- subs
- orr
- and
- mvn
- str
- nop
- ldr
- Branch Instructions (enabled via flags – Carry, Zero, Overflow, Sign Flags)

Drawbacks

Although we have managed to complete our project of making a processor along with memory and cache management, one of the major drawbacks our project had was that we were unable to effectively incorporate the working of the cache with that of the memory.

Conclusion

We have taken up this project because we were interested in processor design and wanted to know more about it. There are various implementations of CPU’s in papers and on the internet by various people and we took help from the sources mentioned in the bibliography whenever we were in doubt. We made all designs from scratch and implemented our architecture. We confirmed our project was working by running simulations on the same. From deciding how to make a ALU to designing control units for our CPU, we were able to complete its implementation after dealing with all the simple and complex architectural design problems that came along with.

A lot of debugging was required for our control unit and a lot of simulations were required to correct it. We were able to implement a cache separately but combining it with the rest of the circuit was too tough a job for us and despite all of our efforts we could not get it to work.

Our learning from this project was immense, we learnt a number of things including the use of Logisim, which includes all its functions and features. We also have practice in debugging circuits in Logisim, though some more effort is still required in that area. We got an in-depth understanding of the processor design and the compromises that we have to keep in mind while designing a processor.

We have implemented several functions of a processor such as ADD, SUB, AND, OR etc and see the necessity and complications of implementing several basic functions such that we can compute all these operations on a single processor. This was truly a learning and highly informative experience for us and we were able to know things in depth and about the challenges faced in reality.

Additionally, working in a gathering was a brilliant experience and we additionally had great involvement with planning the work done by one another lastly incorporating them to have the total implementation and completion of project.

Bibliography

- Computer Organization and Architecture – Smruti Ranjan Sarangi
- CSN221 lectures -- **Prof. Peddoju Satish Kumar**
- Computer Organization and Architecture – Smruti Ranjan Sarangi lecture Videos https://youtube.com/playlist?list=PL1iLu2CSC9EWAo0ysorNI_nebwF6Rwkr0
- Digital Design and computer architecture ARM edition –Sarah L. Harris & David Money Harris