**LAB Turnin Video**

**9-Bit Instruction Set Architecture**

By: Ari Cortes, Nidhi Giridhar

**Link to demo video**

[**https://urldefense.proofpoint.com/v2/url?u=https-3A\_\_ucsd.zoom.us\_rec\_share\_RA1LmyBkUO7HvuJQUFjLvEKezgE-2DPaMbzc8IlkThh5SV15-2DdJsHArxZO2kNHjl1Z.LhYLsURcK4BFgx-2Da&d=DwQFAg&c=-35OiAkTchMrZOngvJPOeA&r=UuQo\_pPIaknJkAGtzbbCYKet4oLtE8uP6PLA6VER\_Vk&m=N71xZfqokRE8iJJ93hY4Bzi8iRhnyFGrHd3\_M4zIiV8&s=LMFHbsSsQ8bQhTLVqka-TEVpeVJmvFFj639GZFHj9P4&e=**](https://urldefense.proofpoint.com/v2/url?u=https-3A__ucsd.zoom.us_rec_share_RA1LmyBkUO7HvuJQUFjLvEKezgE-2DPaMbzc8IlkThh5SV15-2DdJsHArxZO2kNHjl1Z.LhYLsURcK4BFgx-2Da&d=DwQFAg&c=-35OiAkTchMrZOngvJPOeA&r=UuQo_pPIaknJkAGtzbbCYKet4oLtE8uP6PLA6VER_Vk&m=N71xZfqokRE8iJJ93hY4Bzi8iRhnyFGrHd3_M4zIiV8&s=LMFHbsSsQ8bQhTLVqka-TEVpeVJmvFFj639GZFHj9P4&e=)

Passcode: 5zp25\*c0

**Lab 1 Highlights**

Our processor uses 8 8-bit registers and has an 8-bit memory of size 256. Our instructions have two formats:

Pointer Format

|  |  |  |
| --- | --- | --- |
| operation | r1 | r2 |
| 4 bits | 3 bits | 2 bits |

Immediate Format

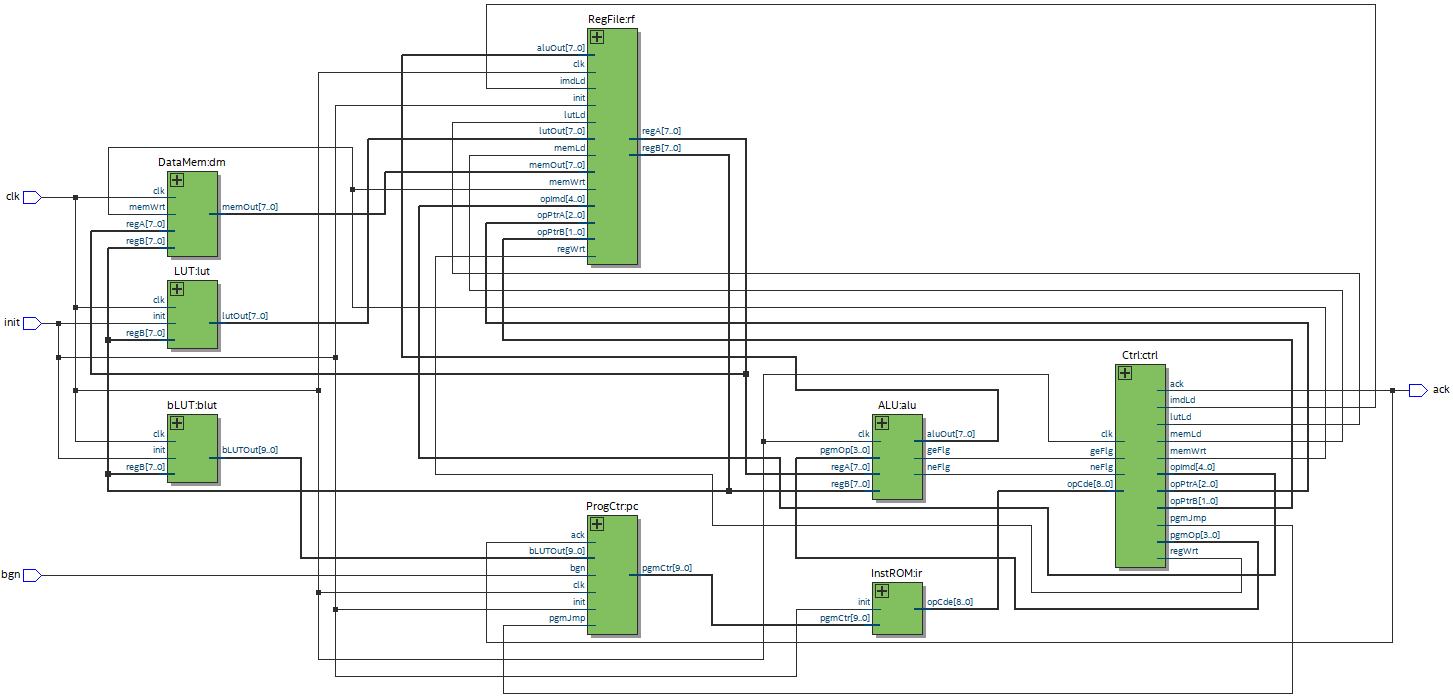
|  |  |
| --- | --- |
| operation | Immediate |
| 4 bits | 5 bits |

Our branching uses the immediate format to send indices into a branching look-up table. The reason for this is that using 5-bit immediate only allowed us to branch +-15 lines. Switching to a look-up table allows us to access 32 locations ranging from 0 to 1024.

Our processor supports 15 operations

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Opcode** | **Instruction** | **Description** |
| Load R0 | 0000 | ld0 #c | Loads R0 with immediate |
| Move | 0001 | mov r1, r2 | Loads R1 with R2 value |
| Look Up Table | 0010 | lut r1 r2 | Loads R1 with LUT[R2] value |
| Load Register | 0011 | ldr r1 r2 | Loads R1 with mem[R2] value |
| Store | 0100 | str r1 r2 | Stores R1 into mem[R2] location |
| Add | 0101 | add r1 r2 | Loads R1 with R1+R2 |
| Subtract | 0110 | sub r1, r2 | Loads R1 with R1-R2 |
| Compare | 0111 | cmp r1 r2 | Uses R1-R2 to set >= and != Flags |
| Bitwise And | 1000 | and r1 r2 | Loads R1 with R1&R2 |
| Bitwise Or | 1001 | or r1 r2 | Loads R1 with R1|R2 |
| Bitwise XOr | 1010 | xor r1 r2 | Loads R1 with R1^R2 |
| Logical Shift Left | 1011 | lsl r1 r2 | Logical Shifts R1 by R2 bits |
| Reduction XOr | 1100 | rxr r1 r2 | Loads R1 with ^R2 |
| Branch | 1101 | brn #c | Jumps to bLUT[#C] |
| Branch >= | 1110 | bge #c | Jumps to bLUT[#C] if geFlag is set |
| Branch != | 1111 | bne #c | Jumps to bLUT[#C] if neFlag is set |

**LAB 2 Highlights**



1

2

3

4

5

5

5

5

6

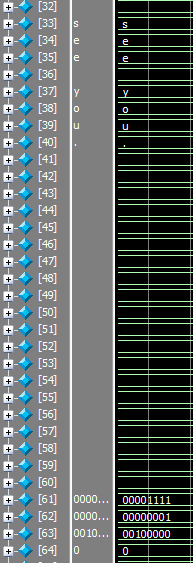
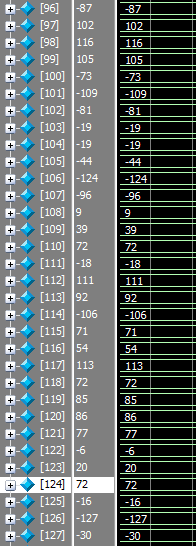
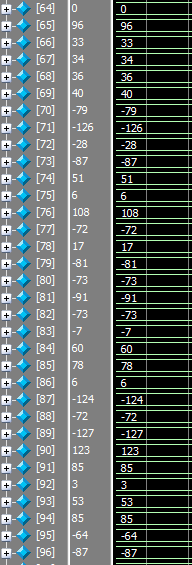
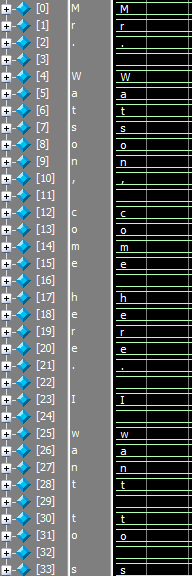
Our Hardware has two custom components, LUT and bLUT. The reason for their inclusion was to deal with small 5-bit immediates when needing larger ones. Using the immediates as look-up table indices allows us access to use larger 10-bit constants and branching values.

Our hardware flow starts with Instruction Rom which uses the current Program Counter value to grab lines from the assembly code. These lines then go to the Controller which parses the Operation, Pointers, and Immediates. Controller also sets the flags that will be used throughout the rest of the instruction. Register File then uses the R1 and R2 pointers to access the desired values or load an immediate into R2. From here Data Memory, LUT, bLUT, and ALU all use the R1 and R2 values to produce their own set of outputs. If the memWrt flag is set, Data Memory will also write to memory on this step. On write back, the proper output is selected using the flags set by the controller and written to R1.

**Program 1**

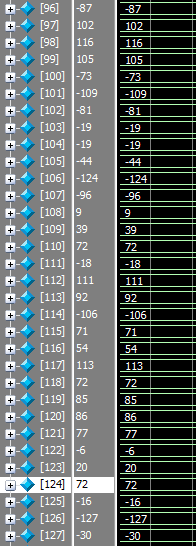
We used the first test bench to test all the programs and only used it to initialize values. All of our debugging and verification were done using waveform and $display outputs. In order to see if program1 worked correctly, we added a dataMem Core wave and checked its final state.

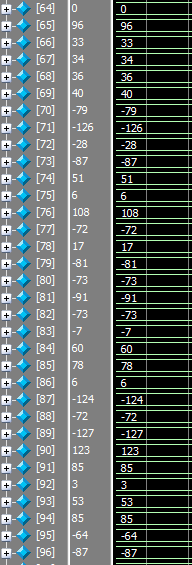
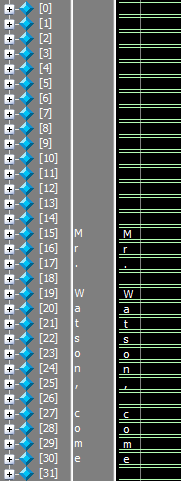
Original Message Encrypted Message



**Program 2**

Since we did not create a test bench for each program, we ran program 1 to fill in the encrypted bits in memory for programs 2 and 3 to decrypt. In order to see if program 2 was correctly decrypting the message, we 0’d out the first 64 bits before running the program.

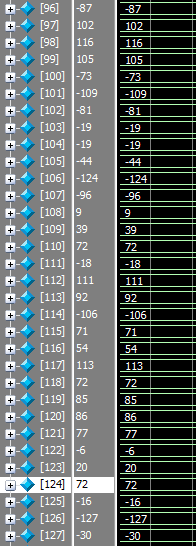
Decrypted Message Encrypted Message

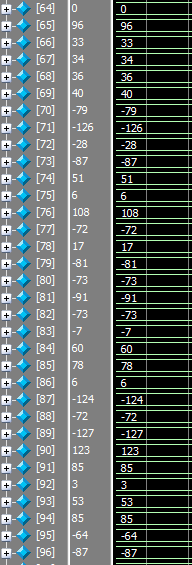
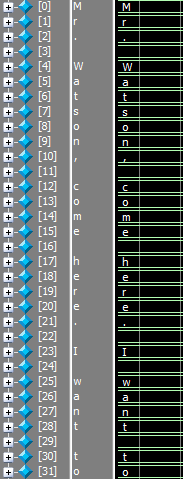
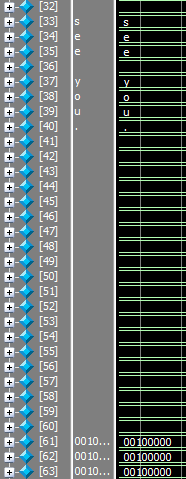


// Notice how the first 15 characters are spaces

**Program 3**

Program 3 utilized program 1 in the same way program 2 did. The first 64 bits were then 0’d out before running program 3.

Decrypted Message Encrypted Message



// Notice how spaces have been removed

**Notes about programs**

Program 1 works only for 49 bytes messages

**How to run the programs**

* Program 1:
  + Run file “encrypt\_tb.sv” with machine code file “Program1.txt” as the input to the instROM
* Program 2:
  + Run file “encrypt\_tb.sv” with machine code file “Program2&1.txt” as the input to the instROM
* Program 3:
  + Run file “encrypt\_tb.sv” with machine code file “Program3&1.txt” as the input to the instROM