Names: Nathaniel Escaro, Hrishi Mehta
CPU Name: Byte LED
I pledge my honor that I have abided by the Stevens Honor System.

Byte LED Documentation

Distribution of Work:

- Hrishi:
    - The Circuit (Design, Construction, Bug Fixing and Testing), and Demo Files for the Circuit.
- Nathaniel:
    - Coding the Compiler that parses the source file and Makefile to easily generate the assembler executable, Language Syntax, The Documentation.

CPU Overview:

This CPU has access to 16 general purpose registers labeled r0 to r15. They each store 8 bits of data and all numbers are considered to be positive integers (specifically each register can hold an integer in the range of [0 - 255]). Register 15 (r15) is a special register that can be interacted with like other registers, but its value will always remain at 0 even if mutated by an instruction.

It supports several arithmetic operations: addition, subtraction, bitwise AND, bitwise inclusive OR, logical left and right shifts, bitwise exclusive OR, and bitwise NOT in the ALU unit.

It also allows for data to be stored in data memory and can be written to or loaded in via user defined labels. All memory addresses are 1 byte long.

Lastly, the CPU controls a 64x64 LED display. Pixels in this display are controlled by the values from r0 to r7, one for each row of 8 pixels.

Setting Up:

Making and Using the Compiler

To create the CPU compiler, *gcc* must be installed on your system. Afterwards:

1) Unzip the folder
2) Navigate to the src directory
3) Run `make` or `gcc -o ../cbld bcode.c parser.c main.c -I./include`

4) The compiler is now in the unzipped folder*.

To use the compiler, simply type the following command into the terminal:

```
./cbld <file_name>
```

where `<file_name>` is the name of the file you wish to compile**. The file must have a `.bld` extension. Once this command is executed, a `.bin` and `.dat` (if a DATA field is within the file) is created within the same directory of the inputted file. The compiler MUST be in the same directory of the source file. If the source file is in a different directory, place the compiler inside the same folder.

**Ensure that the compiler has read, write and execute permissions.*
***File names can only be 25 characters long*

Importing to Logisim
Once the file is compiled, the resulting `.bin` and `.dat` files must be imported to the correct RAM segments in the circuit.

Load the `.bin` file into the RAM module titled “Instructions”. A popup window should appear listing the types of ways to import the file. Select “Binary” and “little-endian”, then click “Load”.

Similarly, load the `.dat` file (if it exists) into the RAM module titled “Data”. A popup window should appear listing the types of ways to import the file. Select “Binary” and “little-endian”, then click “Load”.

Demo Files
Provided in the zipped folder are 3 demo files found in the “demo” directory: `shapes.bld`, `face.bld`, and `move.bld`. These files are already compiled, but you can recompile them if you want to. Each showcases some of the features this CPU can do. Feel free to inspect the code and run them in Logisim!

Byte LED Language:

Data Input and Notation:

- rD, rN, rM - Denotes register[D, N, M]. D, N, M are integers from 0 to 15 inclusive.
- bit8 - A binary number consisting of 8 bits. This can be represented (and written) into the program in several ways.
    - Decimal (from 0 - 255)
    - 8 bit Binary (b_01100011)

    - ASCII Binary (B_~@@~~~@@)
        - The “@” symbol represents 1 and the “~” symbol represents 0.
- label - a string that represents a memory address.

General Syntax:

Dest <- Func(Arg1, Arg2)

- **Dest:** The destination register or destination label (memory address)
- **Func:** The command to execute. If this is empty, this is a SET call and uses a slightly different format than the one above (found in the section *CPU Instructions*).
- **Arg1:** First argument to be provided to the instruction, either a register or a label. This is empty for some instructions.
- **Arg2:** Second argument to be provided to the instruction, either a register or a bit8 number. This is empty for some instructions.

Data Field:
To create a data field for the file, a “> DATA” (the space between “>” and “DATA” must be included) at the top of the file. Afterwards, you can define labels and the data located in them by the following syntax:

Label <- number(s)

Each label accepts a sequence of numbers, separated by a space. Please note that the label name consists of at most **10 LETTERS** (uppercase and/or lowercase). Additionally, the sequence of numbers only supports **50 CHARACTERS** (including spaces). Failing to adhere to these rules will cause the compiler to fail. Additionally, make sure that there are no trailing spaces after each label line.

To exit the data field, “> START” (the space between “>” and “START” must be included) must be typed in a new line. CPU instructions can be written after this line.

IMPORTANT: *If your file does NOT contain a DATA field segment, DO NOT write “> START” at the top of the file.*

CPU Instructions:

INSTRUCTION	SYNTAX	DESCRIPTION
SET REGISTER	rD <- rM	Sets the register (rD) to the value

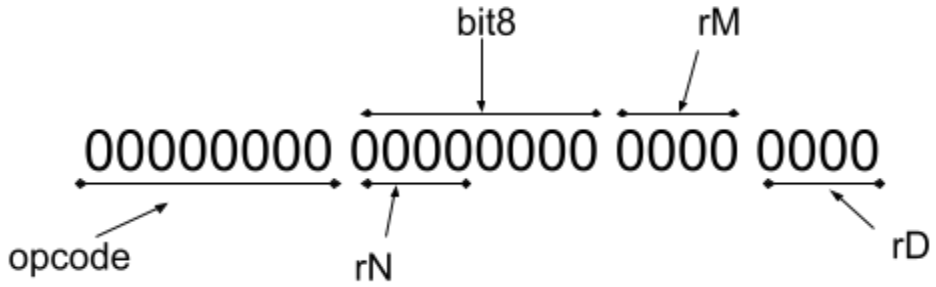
	rD <- bit8	stored in (rM) or the provided bit8 number
ADDITION	rD <- ADD(rM, rN) rD <- ADD(rM, bit8)	Adds the values in two registers (rM and rN) or one register and a bit 8 number and stores the sum into a register (rD).
SUBTRACTION	rD <- SUB(rM, rN) rD <- SUB(rM, bit8)	Subtracts the values in two registers (rM and rN) or one register and a bit 8 number and stores the difference into a register (rD).
BITWISE AND	rD <- AND(rM, rN) rD <- AND(rM, bit8)	Performs the bitwise AND operation on the values in two registers (rM and rN) or one register and a bit 8 number and stores the result into a register (rD).
BITWISE OR	rD <- ORR(rM, rN) rD <- ORR(rM, bit8)	Performs the bitwise inclusive OR operation on the values in two registers (rM and rN) or one register and a bit 8 number and stores the result into a register (rD).
LOGICAL BITSHIFT LEFT*	rD <- LBS(rM, rN) rD <- LBS(rM, bit8)	Performs a logical bitwise left shift on the value inside rM by the value of rN or the provided bit8 number.
LOGICAL BITSHIFT RIGHT*	rD <- RBS(rM, rN) rD <- RBS(rM, bit8)	Performs a logical bitwise right shift on the value inside rM by the value of rN or the provided bit8 number.
BITWISE XOR	rD <- XOR(rM, rN) rD <- XOR(rM, bit8)	Performs the bitwise exclusive OR operation on the values in two registers (rM and rN) or one register and a bit 8 number and stores the result into a register (rD).
BITWISE NOT	rD <- NOT(rM)	Performs the bitwise NOT operation on the value inside a register (rM) and stores the result into a register (rD)
REGISTER SAVING	label <- S(rD, rM)	Saves the value stored inside rD into the data memory address calculated from the label's address plus the value of rM.
REGISTER LOADING	rD <- L(label, rM)	Loads the value found in the data memory address calculated from the label's address plus the value of rM, into register rD.
FLASH	r15 <- FLASH()	Prints the register values r0 to r7 onto the LED screen, where r0 is the topmost row and r7 is the bottommost row. Each bit in the number corresponds to a pixel on a screen: MSB

		is the leftmost and LSB is the rightmost. The pixel turns on if the bit is 1, otherwise it turns off.
FLUSH	r15 <- FLUSH()	Clears the LED screen.
DELAY	r15 <- DELAY()	An empty instruction; it does not do anything.

*Shift amounts can only be between 0 and 7. If the provided amount is greater than 7, the 3 least significant bits are used for shift amounts.

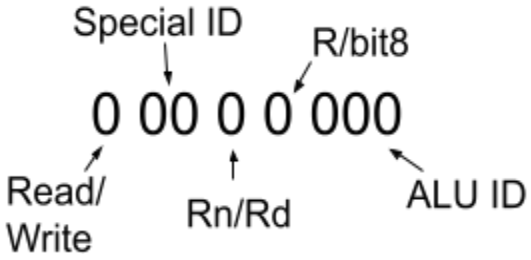
Binary Encoding:

Each instruction takes up 24 bits and is split up as such:



OP Code

The first 8 bits represent the operation code of the instruction. This is further divided into smaller categories to correctly turn on and off specific control signals the CPU uses.



- **Read/Write:** If 1, the CPU is in write mode, something will be overwritten. Otherwise, the CPU will just read values.
- **Special ID:** A 2 bit ID that is used to differentiate between FLASH, FLUSH, LOAD, and SAVING calls.
- **Rn/Rd:** If 1, the value stored in rD will be used for CPU operations. Otherwise, the value stored in rN is used. This is specifically used for the SAVING instruction.

- **R/bit8:** If 1, bit8 values are passed into the ALU as a second argument. Otherwise, a register value (either rN or rD depending on Rn/Rd bit) will be used instead.
- **ALU ID:** This flag denotes what mathematical operation should be conducted on the rM and the second argument. Below are the list of codes and their corresponding operations:
    - 000 - ADD
    - 001 - SUB
    - 010 - AND
    - 011 - ORR
    - 100 - LBS
    - 101 - RBS
    - 110 - XOR
    - 111 - NOT

Encoding Catalog

Key:

- MMMM: Register M's number (in binary)
- DDDDD: Register D's number (in binary)
- NNNN: Register N's number (in binary)
- LLLLLLLL: Label's memory address (in binary)

INSTRUCTION	ENCODING
rD -> rM	10001000000000000MMMMDDDD
rD <- bit8	10001000BBBBBBBB1111DDDD
rD <- ADD(rM, rN)	10000000NNNN0000MMMMDDDD
rD <- ADD(rM, bit8)	10001000BBBBBBBBMMMMDDDD
rD <- SUB(rM, rN)	10000001NNNN0000MMMMDDDD
rD <- SUB(rM, bit8)	10001001BBBBBBBBMMMMDDDD
rD <- AND(rM, rN)	10000010NNNN0000MMMMDDDD
rD <- AND(rM, bit8)	10001010BBBBBBBBMMMMDDDD
rD <- ORR(rM, rN)	10000011NNNN0000MMMMDDDD

rD <- ORR(rM, bit8)	10001011BBBBBBBBMMMMDDDD
rD <- LBS(rM, rN)	10000100NNNN0000MMMMDDDD
rD <- LBS(rM, bit8)	10001100BBBBBBBBMMMMDDDD
rD <- RBS(rM, rN)	10000101NNNN0000MMMMDDDD
rD <- RBS(rM, bit8)	10001101BBBBBBBBMMMMDDDD
rD <- XOR(rM, rN)	10000110NNNN0000MMMMDDDD
rD <- XOR(rM, bit8)	10001110BBBBBBBBMMMMDDDD
label <- S(rD, rM)	11010000LLLLLLLLMMMMDDDD
rD <- L(label, rM)	10101000LLLLLLLLMMMMDDDD
rD <- NOT(rM)	1000111100000000MMMMDDDD
r15 <- FLASH()	010010000000011111111
r15 <- FLUSH()	001010000000011111111
r15 <- DELAY()	000000000000000011111111