**Chapter 13 Assignment 1 – Creating a Simple ALU**

Jeff Cates

Study.com Learning

Computer Science 306: Computer Architecture

July 14, 2023

Table of Contents

[Note to Reader 4](#_Toc140167358)

[Overall Architectural Decisions 5](#_Toc140167359)

[Signed vs Unsigned Values 5](#_Toc140167360)

[Accumulator vs Output Register 5](#_Toc140167361)

[Accepting Constant Values 5](#_Toc140167362)

[Command Arguments 5](#_Toc140167363)

[Command Length 6](#_Toc140167364)

[Managing Variable Arguments 7](#_Toc140167365)

[Potential Improvements 7](#_Toc140167366)

[System Architecture 8](#_Toc140167367)

[Command Length 8](#_Toc140167368)

[Register Code Table 9](#_Toc140167369)

[Opcode Table 10](#_Toc140167370)

[Overloaded Assembly Commands 11](#_Toc140167371)

[Assembly Language Conventions 12](#_Toc140167372)

[Phase One 14](#_Toc140167373)

[8-bit ALU Implementation 14](#_Toc140167374)

[Additional Instruction Justification 15](#_Toc140167375)

[Example Assembly Using Additional Instructions 16](#_Toc140167376)

[Additional Recommended Commands 16](#_Toc140167377)

[8-Bit ALU Instruction Set 17](#_Toc140167378)

[Phase Two 18](#_Toc140167379)

[Opcode Table 18](#_Toc140167380)

[Operand Count Justification 19](#_Toc140167381)

[Assembly to Machine Language Table 20](#_Toc140167382)

[Phase Three 20](#_Toc140167383)

[Control Unit Implementation 20](#_Toc140167384)

[Phase Four 21](#_Toc140167385)

[Program 1 21](#_Toc140167386)

[Program 2 22](#_Toc140167387)

[Program 3 23](#_Toc140167388)

[Program 4 24](#_Toc140167389)

[Additional Resources 26](#_Toc140167390)

[Circuit Figures 26](#_Toc140167391)

[The CPU, ROM, Keyboard, and Output (Figure 1) 26](#_Toc140167392)

[Inside the CPU: Control Unit, ALU, and Register Bank (Figure 2) 27](#_Toc140167393)

[Inside the Control Unit (Figure 3) 28](#_Toc140167394)

[Inside the Register Bank (Figure 4) 29](#_Toc140167395)

[The Write Register Selector (Figure 5) 30](#_Toc140167396)

[An Individual Register Implementation (Figure 6) 31](#_Toc140167397)

[Inside the ALU (Figure 7) 32](#_Toc140167398)

[Inside the 8-bit Adder (Figure 8) 33](#_Toc140167399)

[Inside the 1-bit Adder (Figure 9) 34](#_Toc140167400)

[Inside the 8-bit Equality Checker (Figure 10) 35](#_Toc140167401)

[Inside the 4-bit Equality Checker (Figure 11) 36](#_Toc140167402)

[Inside the 1-bit Equality Checker (Figure 12) 36](#_Toc140167403)

[8-bit Left and Right Shift Circuits (Figure 13 and 14) 36](#_Toc140167404)

[8-bit to ASCII Encoder (Figure 15) 38](#_Toc140167405)

[8-bit from ASCII Decoder (Figure 16) 39](#_Toc140167406)

[Reusable Assembly Sub-Processes 40](#_Toc140167407)

[Accepting User Input 40](#_Toc140167408)

[Printing Register Value to Output 43](#_Toc140167409)

[Printing Register Value as Bits to Output 45](#_Toc140167410)

[Excel Assembler 46](#_Toc140167411)

[Logism Implementation and Code Examples 47](#_Toc140167412)

[External References 48](#_Toc140167413)

# Note to Reader

The design and development of this 8-bit ALU, CPU, register bank, control unit, and interfaces to memory, keyboard and display was a joy to undertake. The final product is a fully implemented microcomputer with these features:

1. Fully implemented machine language commands
2. 3 Reserved registers (RG for input, RF for output, and RH for memory address of current command)
3. Fully implemented 8-bit CPU, Control Unit, Register Bank, and 8-bit ALU.
   1. The CPU reads commands from the current ROM memory address, supplied to RH
   2. RH is a counter register that increments unless overwritten by a JMP or CJMP command
   3. The control unit decodes 16-bit commands into input registers, operations, and constant values
4. ROM memory module preloaded with sample programs.
5. Functioning keyboard
   1. The keyboard is wired into the RG register for access within assembly programs.
6. Functioning output display
   1. The output display is wired into the RF register for access within assembly programs.
7. LED matrix display
   1. The LED matrix displays the contents of other registers, for reference only.
8. An excel file configured to “assemble” the assembly language into machine language that can be easily loaded into the Logism ROM module.

Additional sections were included in this document to outline the architecture and circuit figures in more detail. They follow the 4 phases requested in the assignment.

I have also included the assembly language programs, excel “assembler” and a reference to my github project and several youtube videos demonstrating the final product.

Logism Circuit, Assembly, and Excel: <https://github.com/jeffbcates/8bitcpu>

Walk-through of 8-bit CPU and system: <https://youtu.be/QRCL1FTgDJQ>

Using the Excel assembler: <https://youtu.be/nm02wkrpm0k>

Live demo of assembler programs in Logism: <https://youtu.be/PCbke2RCTlo>

To edit my assembly files, I used BBEdit (formally TextWranger), a free text editor for Mac which includes line-numbers. I set my tab spacing to 8 to ensure adequate spacing between labels, commands, and comments.

## Overall Architectural Decisions

After implementing my basic 8-bit ALU, I began to think about what it would take to implement the control unit and how the control unit would interface with the ALU and memory. During this process, I asked myself key questions which drove the system architecture. These included:

1. Signed vs Unsigned Values: Should I implement signed or unsigned values?
2. Accumulator vs Output Register: Should I use an accumulator register or an output register?
3. Accepting Constant Values: How can I pass constant values to my commands?
4. Command Arguments: How many arguments should my commands allow?
5. Command Length: What is my optimal machine language command length?
6. Final Thoughts: What exceptions would I need to consider?

## Signed vs Unsigned Values

The decision to implement my ALU, CPU, and system with signed versus unsigned values was an easy one for me. From previous work developing games executed on older 386 and 486 systems I am aware of the limitations of 16-bit and 8-bit processors. To maximize memory space, I often used unsigned values stored as 8-bit integers. Thus, I decided to implement an unsigned system.

## Accumulator vs Output Register

When considering using an accumulator or an output register, the output register seemed more natural to me. It seemed like an accumulator register could make working with multiple registers and multiple values at the same time cumbersome and confusing. I thus chose to implement all commands using an output register.

## Accepting Constant Values

Once I decided against an accumulator register, I then realized I would need a way to pass constants to my commands. This made me wonder how I would store whether a command argument was a constant or a register.

After thinking about this, I decided that all commands would accept only register values except for the SET command. This SET command would allow me to set a register value to a constant. All other commands would manipulate register values. This kept my commands very simple and straightforward.

## Command Arguments

When thinking about command arguments, I realized that some commands would work better with 3 arguments. This was novel compared to any assembly language I have seen before. But it seemed to make sense. For example:

ADD RA RB RC Add register A and B store in register C.  
SUB RA RB RC Subtract register B from A and store in register C.

Other command would need only 2 arguments, such as:

SET 154 RA Set the register A value to the constant 154

## Command Length

The length of all machine-language commands for this implementation is 16-bits or 1 word.

What naturally followed separating constants into their own SET command, with allowing 3 argument commands was the realization that all these commands fit nicely into 16 bits. Coincidently both ROM and RAM store values grouped into words. This confirmed my decision to use 16-bit commands and led me to the following conclusions:

## Managing Variable Arguments

Assuming that all machine-language commands contain either 3 register addresses, or a constant value and an additional register address makes sense. However, there are some commands that are less intuitive in assembly and other commands that may not make use of all available bits. What to do in those cases?

INC RA \_ \_ Do we really need to specify that the increment command writes back to RA?

What do we do with the unused arguments?

MOV RA \_ RB Again, moving one register to another only requires 2 register arguments.

What do we do with the unused arguments?

My solution, while potentially unconventional, seems to well for me:

Argument A Represents the input register A address.

May also represent a constant value. In these cases, Argument B is unavailable.

Argument B Represents the input register B address. For some commands, this argument is

ignored and left as 0000 when converted from assembly to machine language.

Argument C Always represents the output register of the operation.

## Potential Improvements

The current implementation of the CPU and related systems functions as intended and works well enough. However, there are several opportunities for improvement that I see in a future version:

1. Register values utilize only 3 bits instead of 4.

The current implementation of the register addresses uses only 3 bits leaving the 4th bit unused and reserved for commands that provide a full 8-bit constant, such as SET and JMP. Using the 4th bit as part of register address would expand the possible registers from 8 to 16.  
  
See Figure 3 in [Circuit Figures](#_Circuit_Figures)

1. Register bank limited to 8 registers.

The current implementation of the register bank contains only 8 registers. Of these registers, three are reserved for communication with the display, keyboard, and memory addressing. These leaves only 5 registers available for the assembly code to utilize.

1. Memory addressing limited to 255 due to using a single byte addressing.

With additional registers, it would be possible to reserve two registers for memory addressing, creating a page / block style address. In this manner, Register P could represent the memory page, of 256 pages. Register O could represent the specific location in that page.

1. No ability to access directly access memory.

The current implementation only uses memory to store the lines of machine language responsible for program execution. Utilizing an additional register, the system could allow access to memory in the same manner that commands are read into the CPU. This could be implemented with

# System Architecture

This section contains an overview of the system architecture and techniques used to create the 8-bit CPU, machine language, and assembly language.

## Command Length

The length of all machine-language commands is 16-bits or 1 word. Commands fall into one of structures:

Command Structure I

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Operation | | | | Argument A | | | | Argument B | | | | Argument C | | | |
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |

These commands are comprised of the following 16 bits:

1. Operation - 4 bits representing 1 of 16 different operations.
2. Argument A - 4 bits – representing the name of the first input register.
3. Argument B - 4 bits – representing the name of the second input register.
4. Argument C - 4 bits – representing the name of the third output register.

Example assembly in this structure family:

COM RB RC RA Compare register value RB with RC and store in RA.

MOV RC RB Move register value RC to RB

ADD RD RC RA Add register value RD and RC and store in RA.

Command Structure II

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Operation | | | | Constant Value | | | | | | | | Argument C | | | |
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |

These commands are comprised of the following 16 bits:

1. 4 bits – representing 1 of 16 different instructions.
2. 8 bits – representing the constant value.
3. 4 bits – representing the input register.

Example assembly in this structure family:

SET 154 RA Set register RA to a constant value of 154.

JMP 88 Jump to line 88

CJMP 88 RA Jump to line 88 when RA is equality value from COM command.

## Register Code Table

The CPU implements 8 registers. Two of these registers are dedicated and should not be used within assembly language for the storage or manipulation of program values.

Register Code Table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Register** | **Code** | **Internal** | **Assm.** | **Description** | **Notes as implemented** |
| A | 0000 | 00000001 | RA | General 8-Bit A Register |  |
| B | 0010 | 00000010 | RB | General 8-Bit B Register |  |
| C | 0100 | 00000100 | RC | General 8-Bit C Register |  |
| D | 0110 | 00001000 | RD | General 8-Bit D Register |  |
| E | 1000 | 00010000 | RE | General 8-Bit E Register |  |
| \*F | 1010 | 00100000 | RF | Reserved for Constant Output | Used to display ASCII output |
| G | 1100 | 01000000 | RG | Reserved for Constant Input | Used to accept keyboard input |
| H | 1110 | 10000000 | RH | Current command memory address. | Counter register, reserved for command address in ROM. Updateable for JMP. |

\* Note: setting the 8th bit of register F to 1 triggers the keyboard input to be cleared. In the example assembly programs you will see lines of code like below, which write a value to the output and clear the input in one command:

~~~ ASSUME RA CONTAINS THE ASCII ENCODED VALUE “5” ~~~  
  
SET 128 RB Set clear bit on register B

ADD RA RB RF Add that clear bit to the value “5” in register A and write to output register F.

Effectively writing a character to output and clearing the keyboard cache.

SET 0 RF Clear the output register value so that the character does not repeat.

## Opcode Table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Op #** | **Opcode** | **Assm.** | **Instruction** | **Operands Used** | **Result** |
| 1 | 0000 | COM | COMPARE | 3, A and B, C | C:= 2 (A=B), 1 (A<B), 4 (A>B) |
| 2 | 0001 | SUB | SUBTRACTION | 3, A and B, C | C:= A – B |
| 3 | 0010 | ADD | ADDITION | 3, A and B, C | C:= A+B |
| 4 | 0011 | DEC | ARITHMETIC DECREMENT | 2, A and C | C:= A - 1 |
| 5 | 0100 | INC | ARITHMETIC INCREMENT | 2, A and C | C:= A + 1 |
| 6 | 0101 | NOT | BINARY NOT | 2, A and C | C:= ~A |
| 7 | 0110 | AND | BINARY AND | 3, A and B, C | C:= A&B |
| 8 | 0111 | ORR | BINARY OR | 3, A and B, C | C:= A|B |
| 9 | 1000 | SLT | SHIFT LEFT | 2, A and C | C:= A << 1 |
| 10 | 1001 | SRT | SHIFT RIGHT | 2, A and C | C:= A >> 1 |
| 11 | 1010 | MOV | MOVE REGISTER A TO C | 2, A and C | C := A |
| 12 | 1011 | SET | SET REGISTER C TO CONSTANT | 2, Constant and C | C := CONSTANT VALUE |
| *131* | 1100 | SMEM | SET MEMORY | 3, A and B, C | Memory at block B byte C := A |
| *141* | 1101 | GMEM | GET MEMORY | 3 A and B, C | C:= Memory at block A byte B |
| 15 | 1110 | CMOV | MOV A TO C WHEN B = 2 | 3, A and B, C | C := A when B == 2 |
| 16 | 1111 | CJMP | CONDITIONAL JUMP | 2, A and C | RH := Constant when C == 2 |
| *172* | 1011 | JMP | JUMP TO CONSTANT ADDRESS | 1 Constant | RH := Constant |
| *182* | 1110 | AJMP | CONDITIONAL ADDRESS JUMP | 2, A and B | RH := A when B == 2 |

1: Not yet implemented.

These opcode addresses are reserved for a future implementation of read/write accessible memory.  
  
2: Overloaded assembly command.

See explanation in [Overloaded Assembly Commands](#_Overloaded_Assembly_Commands) section.

## Overloaded Assembly Commands

Several assembly commands are implemented by overloading an existing command. These alias commands are used to simplify assembly programming but accomplish the same function as the overloaded counterparts. For illustration the default arguments are provided below and the aliased arguments are left blank.

Alias Arguments Command Defaults

JMP 1, Constant SET Argument C defaults to RH when assembled to machine.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Operation | | | | Constant Value | | | | | | | | Argument C | | | |
| 1 | 0 | 1 | 1 |  |  |  |  |  |  |  |  | 1 | 1 | 1 | 0 |

Alias Arguments Command Defaults

AJMP 3, A B CMOV Argument C defaults to RH when assembled to machine.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Operation | | | | Argument A | | | | Argument B | | | | Argument C | | | |
| 1 | 1 | 0 | 1 |  |  |  |  |  |  |  |  | 1 | 1 | 1 | 0 |

## Assembly Language Conventions

A program can become convoluted quickly in any language. This is especially true with Assembly. Thus, I decided on several conventions when writing my assembly programs. This helped reduce confusion.

1. Reserve register RD and RE for use in main loop

I assumed in my code that when I jump to a sub-process that sub-process should not manipulate the values in register RD or RE. Those would be reserved for the main loop to maintain state after returning from a sub-process.

1. Implement sub-processes using JMP and CJMP commands.

One reason assembly becomes confusing is the programs tend to run together, instead of using procedural techniques of higher languages. To combat this, I decided to implement sub-processes to perform specific tasks and return to the main process.

1. Reserve register RC for communication between main loop and sub-processes

When the main loop did jump to a sub-process that sub-process should use RC as a parameter for its input. For example, the “print” sub-process would assume that the value to print is stored in RC.

1. Reserve register RE as a return address register

To reuse code, I use the JMP command to jump to generalized processes, such as printing a register to output. These processes assume that register “RE” contains the address to JMP back on completion.

In code, this looks like:

01 :start SET RA 12 ; set RA to constant value 12

02 SET RB 10 ; set RB to constant value 10

03 ADD RA RB RC ; add RA to RB and save back to RC

04 SET :return RE ; set RE to location of :return label (06 in this case)

05 JMP :print ; jump to print command, assume it prints RC to output

06 :return SET RA 4 ; do some additional work

~~~  
  
50 :print

~~~ DO SOME WORK ~~~

60 AJMP RE ; jump back to location in register RE

1. Use of assembly language labels

My Excel assembler makes use of labels by applying a row number to each line of code, which translates into the decimal memory location of that line of machine code in ROM. When the excel file detects a label (prefixed with a colon), it uses xlookup to determine the line number and substitutes that line number in the generated machine code.

1. Use of assembly comments

The assembler ignores anything after the 3rd argument of a line of assembly code, assuming this is commenting. It displays this in the “Psuedo-Code / Comment” column on the excel sheet.

1. Minimize register use where possible.

When I initially wrote my assembly programs my conditional jumps when RB < RC looked like this:

COM RB RC RA ; compare RB to RC and write to RA. Value is 1 for “<” , 2 for “=”, and 4 for “>”

SET 1 RD ; we want to check for “less than”, so set RD to that value

COM RA RD RA ; compare RA to RD to see if it is “less than”

CJMP 12 RA ; jump to line 12 when RA represents equality, meaning RB < RC

This meant I needed 2 extra registers to complete this kind of comparison. After some thought, I realized I could rewrite the above assembly as follows:

COM RB RC RA ; compare RB to RC and write to RA. Value is 1 for “<” , 2 for “=”, and 4 for “>”

SLT RA RA ; shift RA to the right, converting 1 “<” to 2 “=”

CJMP 12 RA ; jump to line 12 when RA represents equality, meaning RB < RC

This saves 1 line of assembly and saves 1 register. A big deal with only a few registers available.

You will notice this convention used throughout the sample assembly provided.

# Phase One

## 8-bit ALU Implementation

The 8-bit ALU implements the following 9 minimum operations:

* Addition ADD RA RB RC add RA and RB and write results to RC
* Increment INC RA RA increment RA and write results back to RA
* Decrement DEC RA RA decrement RA and write results back to RA
* Comparison COM RA RB RC compare RA and RB, write results to RC
* Logic Bitwise Not NOT RA RB RC apply logic bitwise not to RA, RB, save to RC
* Logic Bitwise And AND RA RB RC apply logic bitwise and to RA, RB, save to RC
* Logic Bitwise Or ORR RA RB RC apply logic bitwise or to RA, RB, save to RC
* Register Right Logic Shift SRT RA RC apply register right shift to RA, save to RC
* Register Left Logic Shift SLT RA RC apply register left shift to RA, save to RC

The implementation of the ALU can be seen in [Inside the ALU (Figure 7)](#_Inside_the_ALU) within the circuit figures section.

## Additional Instruction Justification

Justification of additional five instructions defined by the 8-bit ALU and CPU.

The ALU is required to implement a minimum 9 operations as outlined in the previous section. Below are the five additional instructions I implemented. In the following sections I outline the full instruction set with details, examples, and results.  
  
The five additional instructions I implemented to fully round out my assembly language and CPU were:

1. SUB Arithmetic Subtraction:  
   I chose to implement all register values as unsigned 8-bit integers. In doing so arithmetic subtraction is necessary. Subtract the second value from the first value and return the result. Assembled as: *SUB RA RB RC Subtract register B from register A and write to register C.*
2. MOV Move value stored in register at address A to register at address C.

This is a critical assembly operation that is necessary for any complete assembly language implementation. It allows us to move values to different registers while implementing more advanced math and logic operations. Using the 3-argument approach, the assembly looks like this:  
*MOV RD RC Move the value in register RD to register RC.*

1. SET Set a register to a constant value.  
     
   Based on my architectural decisions outlined above, there needs to be a way set a register to a constant value. This operation accomplishes that critical task. Using the 2-argument constant:  
   *SET 154 RA Set the value in register RA to constant value 154.*
2. CMOV Conditionally move A to C when B = 2

Move the value stored in register at address A to register at address C when value at register RB is equal to the equality value of “2”, produced from a COM operation. Using the 3-argument approach the assembly looks like this:  
*CMOV RD RB RC Move the value stored in RD to RC when RB is “2” meaning comparison is equal.*

1. AJMP Conditionally jump to code at address A when B = 2 (overload of CMOV)

The AJMP command is an overload of CMOV, where argument C is defaulted to register RH when assembled to machine language. Register RH is the register used to set the address of the current memory location for the program. This command allows programs to branch to a sub-process and then return to the original calling process once the sub-process completes.  
*AJMP RE RB* *Move to address at RE when RB = 2*

## Example Assembly Using Additional Instructions

An example implementation including the above 4 operations would look like this:

Assume user enters “1”, ASCII value 49 on keyboard input register RG. We want to retrieve the ASCII value entered, convert it to a numeric value and transfer it to register RD only when it is within a range of 0 to 9.

Line Assembly Comments  
  
~~~ Assume a check has been performed for a valid ASCII value greater than 48 (“0”) ~~~

Text

Description automatically generated with medium confidence

The code above demonstrates each of the additional assembly commands and how they could interact with each other in an assembly program. For additional examples, see the complete assembly programs at my github.

## Additional Recommended Commands

Several additional commands that I would like to implement in a future version of the CPU surround the ability to access memory. Assume a reserved register RP represented the page of application accessible memory, and that the memory was stored in 8-byte values segmented into 256 bytes per block and 256 blocks per page and 256 pages in total. Our system could then make 16,777,216 bytes (16 megabytes) of memory available to applications while only using 8-byte register values. That is 256 pages made of 256 blocks made of 256 bytes.

1. SMEM Set the memory stored in the current page at block B and byte C to value in A.  
     
   SET 10 RB Set RB to 10 meaning the 10th block of memory.  
   SET 2 RC Set RC to 2 meaning the 2nd byte at that page of memory.

SMEM RA RB RC Write value stored in RA to block 10 and byte 2 of current page of memory.

1. GMEM Get the memory stored in the current page at block A and byte B into register C.

SET 10 RB Set RB to 10 meaning the 10th block of memory.  
SET 2 RC Set RC to 2 meaning the 2nd byte at that page of memory.

GMEM RB RC RA Read value from block 10 and byte 2 of current page into register RA.

## 8-Bit ALU Instruction Set

Complete instruction set for 8-bit ALU

1. **Compare** Operands: 2, A and B  
   Operation: A == B Output: 2: A == B, 1: A < B, 4: A > B
2. **Arithmetic subtraction** Operands: 2, A and B  
   Operation: A – B Example: 9 – 4 = 5
3. **Arithmetic Addition** Operands: 2, A and B  
   Operation: A + B Example: 9 + 4 = 13
4. **Arithmetic Decrement** Operands: 1, A Only  
   Operation: A - 1 Example: 9 – 1 = 8
5. **Arithmetic Increment** Operands: 1, A Only  
   Operation: A + 1 Example: 9 + 1 = 10
6. **Logical Bitwise Not** Operands: 1, A Only  
   Operation: ~A Example: ~1001 = 0110
7. **Logical Bitwise And** Operands: 2, A and B  
   Operation: A&B Example: 1001 & 1100 = 1000
8. **Logical Bitwise Or** Operands: 2, A and B  
   Operation: A|B Example: 1001 | 1100 = 1101
9. **Shift Left** Operands: 1, A Only  
   Operation: A<<1 Example: 0101<<1 = 1010
10. **Shift Right** Operands: 1, A Only  
    Operation: A>>1 Example: 1010>>1 = 0101

# Phase Two

## Opcode Table

Opcode table for ALU and CPU instructions in Phase One

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Op #** | **Opcode** | **Assm.** | **Instruction** | **Operands Used** | **Result** |
| 1 | 0000 | COM | COMPARE | 3, A and B, C | C:= 2 (A=B), 1 (A<B), 4 (A>B) |
| 2 | 0001 | SUB | SUBTRACTION | 3, A and B, C | C:= A – B |
| 3 | 0010 | ADD | ADDITION | 3, A and B, C | C:= A+B |
| 4 | 0011 | DEC | ARITHMETIC DECREMENT | 2, A and C | C:= A - 1 |
| 5 | 0100 | INC | ARITHMETIC INCREMENT | 2, A and C | C:= A + 1 |
| 6 | 0101 | NOT | BINARY NOT | 2, A and C | C:= ~A |
| 7 | 0110 | AND | BINARY AND | 3, A and B, C | C:= A&B |
| 8 | 0111 | ORR | BINARY OR | 3, A and B, C | C:= A|B |
| 9 | 1000 | SLT | SHIFT LEFT | 2, A and C | C:= A << 1 |
| 10 | 1001 | SRT | SHIFT RIGHT | 2, A and C | C:= A >> 1 |
| 11 | 1010 | MOV | MOVE REGISTER A TO C | 2, A and C | C := A |
| 12 | 1011 | SET | SET REGISTER C TO CONSTANT | 2, Constant and C | C := CONSTANT VALUE |
| *131* | 1100 | SMEM | SET MEMORY | 3, A and B, C | Memory at block B byte C := A |
| *141* | 1101 | GMEM | GET MEMORY | 3 A and B, C | C:= Memory at block A byte B |
| 15 | 1110 | CMOV | MOV A TO C WHEN B = 2 | 3, A and B, C | C := A when B == 2 |
| 16 | 1111 | CJMP | CONDITIONAL JUMP | 2, A and C | RH := Constant when C == 2 |
| *172* | 1011 | JMP | JUMP TO CONSTANT ADDRESS | 1 Constant | RH := Constant |
| *182* | 1110 | AJMP | CONDITIONAL ADDRESS JUMP | 2, A and B | RH := A when B == 2 |

1: Not yet implemented.

These opcode addresses are reserved for a future implementation of read/write accessible memory.  
  
2: Overloaded assembly command.

See explanation in [Overloaded Assembly Commands](#_Overloaded_Assembly_Commands) section.

## Operand Count Justification

In my architectural decision process, covered in the Command Length section of the [System Architecture](#_Command_Length), I decided that my machine language commands would be comprised of 16 bits. I worked backwards from this into the optimal number of arguments in my assembly language.

I decided on a novel approach to machine and assembly command arguments. I believe this approach creates an elegant machine language and flexible assembly language.

The machine language commands contain either:

1. a single 4-bit operation, plus three 4-bit register address (3 arguments).
2. a single 4-bit operation, plus an 8-bit constant and one 4-bit register address.

This allows my machine commands to remain at 16-bits but still handle both constant and register values. It also allows flexibility in commands, such as:

ADD RA RB RC Add two registers, leaving them untouched and write to third register.

ADD RA RB RA Add two registers and write back to the first.

ADD RA RB RB Add two registers and write back to the second register.

The only commands that accept constant values accept them by combining the 8-bits after the 4-bit operation into a single constant value within the control unit. They are:  
  
 SET 154 RA Set register A to constant value 154

JMP 10 Jump register H to line 10 (this is an alias to “SET 10 RH”).

CJMP 10 RA Jump register H to line 10 when RA is equal to “2” (the equality value of COM)

Refer to Figure 3 in [Circuit Figures](#_Circuit_Figures) to see how this is achieved.

## Assembly to Machine Language Table

The below table illustrates how the assembly language commands are translated into 16-bit machine commands for the CPU to consume:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Desired Operation** | **Assembly** | | | | **Encoded CPU Command (Highlighted Bits Ignored)** | | | |
|  | Operation | ARG 1 | ARG 2 | ARG 3 | OP. | A | B | C |
| Set Register RB to Constant 4 | SET | 4 |  | RB | 1011 | 00000100 | | 0010 |
| Move Value from RB to RE | MOV | RB |  | RE | 1010 | 0001 | 0000 | 1000 |
| Add RG and RB, Write to RH | ADD | RG | RB | RH | 0010 | 1100 | 0010 | 1110 |
| Increment RF Register Value | INC | RF |  |  | 0100 | 1010 | 0000 | 1010 |
| Compare RG and RF, Write to RH | COM | RG | RG | RH | 0000 | 1100 | 1010 | 1110 |
| Decrement RB Register Value | DEC | RB |  |  | 0011 | 0010 | 0000 | 0010 |
| Subtract RG and RB, Write to RH | SUB | RG | RB | RH | 0001 | 1100 | 0010 | 1110 |
| Binary Or RB and RC, Write to RD | ORR | RB | RC | RD | 0111 | 0010 | 0100 | 0110 |

*Note that the register commands are 4-bit width, though they only use the 3 high bits. This is to leave room for the “SET” constant command, which uses 8 bits for the first argument to represent an 8-bit constant number. In this way all commands are 16 bits.*

## Phase Three

## Control Unit Implementation

My implemented control unit accepts a 16-bit command as input and splits that command into the operation code from the OpCode table along with 3 register addresses or an 8-bit constant value and single register address.

The control unit can be seen in Figure 3, [Inside the Control Unit](#_Inside_the_Control).  
  
The control unit is responsible for splitting the 16-bit command into 16 different operations, an input constant and 3 input register addresses). As seen in the CU reference diagram, the current command is “CJMP”. For this command the CU combines the middle 8 bits of the command into a single 8-bit constant. It also takes the last 4 bits of the command as the output register. For this command, the two input registers are ignored.

# Phase Four

## Program 1

Assignment: Write a program that adds two operands.

For the first program, I implemented the complete logic to accept two operands from user input and then add them together, printing the results to the output display as ASCII. The user inputs can be 1 to 3 digit numbers. I did not perform a check for 3-digit numbers that exceed the 255-value limit. The main loop then repeats.

The program is broken into 3 sections:

1. The main loop:  
     
   A screenshot of a computer

   Description automatically generated with low confidence

The main loop jumps off to two sub-processes. The first sub-process accepts keyboard input into register RC and the second process prints RC to the output display.

In line #9 we are adding the input operand from register RC to the second operand, stored in RC.

The rest of the code involves receiving input from the keyboard and displaying the final output.

1. Get Input sub-process (:getinput)  
     
   The reusable get input sub-process records user input into register RC as a 1-3 digit numeric value and returns to the location stored in RE through a conditional jump to an address (AJMP).  
     
   The code is outlined in [Accepting User Input](#_Accepting_User_Input) of the Reusable Code section of this document.
2. Print output sub-process (:print)  
     
   The reusable print sub-process writes out the contents of register RC to the output display as ASCII-encoded value and returns to the location stored in RE through a condition jump (AJMP).  
     
   The code is outlined in [Printing Register Value to Output](#_Printing_Register_Value) of the [Reusable Assembly Sub-Processes](#_Reusable_Assembly_Sub-Processes) section of this document.

## Program 2

Assignment: Write a program that adds operands until the new value to be added is 0. You do not need to implement the input operations to modify the contents of the registers. Just assume that by the end of each iteration, the register content is modified.

For the second program, I built on my first program, adding an additional loop to the body of the main section to repeat the addition process until the second operand is 0.

The program is broken into 3 sections:

1. The main loop:  
     
   A screenshot of a computer

   Description automatically generated with low confidence

The main loop jumps off to two sub-processes. The first sub-process accepts keyboard input into register RC and the second process prints RC to the output display.

In line #9 we are adding the input operand from register RC to the second operand, stored in RC.

The rest of the code involves receiving input from the keyboard and displaying the final output.

1. Get Input sub-process (:getinput)  
     
   The reusable get input sub-process records user input into register RC as a 1-3 digit numeric value and returns to the location stored in RE through a conditional jump to an address (AJMP).  
     
   The code is outlined in [Accepting User Input](#_Accepting_User_Input) of the Reusable Code section of this document.
2. Print output sub-process (:print)  
     
   The reusable print sub-process writes out the contents of register RC to the output display as ASCII-encoded value and returns to the location stored in RE through a condition jump (AJMP).  
     
   The code is outlined in [Printing Register Value to Output](#_Printing_Register_Value) of the [Reusable Assembly Sub-Processes](#_Reusable_Assembly_Sub-Processes) section of this document.

## Program 3

Assignment: Write a program that increments by 2 the content of a register 10 times.

For the third program, I also used both my pre-written “getinput” and “print” commands to add functionality.

The program is broken into 3 sections:

1. The main loop:  
     
   A picture containing text, screenshot, receipt, document

   Description automatically generated

The main loop gets input from the user and stores it in RC as a starting point. It then loops 10 times, using register RA to count. At each iteration of the loop, it adds the number 2 to register RC and prints a period to the screen.  
  
After 10 iterations, the loop ends and the program jumps to the print command to print the result.

The result will be 20 units larger than the number input by the user (2 x 10). The program sets register RE to :start to repeat the process with a new input from the user.

1. Get Input sub-process (:getinput)  
     
   The reusable get input sub-process records user input into register RC as a 1-3 digit numeric value and returns to the location stored in RE through a conditional jump to an address (AJMP).  
     
   The code is outlined in [Accepting User Input](#_Accepting_User_Input) of the Reusable Code section of this document.
2. Print output sub-process (:print)  
     
   The reusable print sub-process writes out the contents of register RC to the output display as ASCII-encoded value and returns to the location stored in RE through a condition jump (AJMP).  
     
   The code is outlined in [Printing Register Value to Output](#_Printing_Register_Value) of the [Reusable Assembly Sub-Processes](#_Reusable_Assembly_Sub-Processes) section of this document.

## Program 4

Assignment: Write a program that shifts the content of a register until the least significant bit is 0. Think of a way to stop shifting if the content of the register is 11111111 and add it to your program.

For the fourth program, I also used my pre-written “getinput” command but wrote a new “printbits” command that prints out the contents of a register as a binary number.

The program is broken into 3 sections:

1. The main loop:  
     
   A picture containing text, receipt, screenshot

   Description automatically generated

The main loop gets input from the user and stores it in RC as a starting point. It then enters a looping process labeled :loopbits which loops through these tasks until a condition is met:

1. Jump to :printbits to print out the current contents of the register.
2. If the value equals 11111111 (256), jump to “:lotsofones” to print “ONES” and restart.
3. If the LSB in the value equals “0”, jump to “:lsbzero” to print “LSB=0” and restart.
4. Check if the LSB of the value equals zero, as follows:
   1. Line #10 applies a bitwise AND between the value and “1”, saving to RD.
   2. We then compare that value back to “1”, to see if they are the same.
   3. Shift the comparison left, converting “less than” (1) to “equality” (2).
   4. Conditionally jump to “:lsbzero” when that shifted comparison is “equality”

The other two labels in the main loop, “:lsbzero” and “:lotsofones” are responsible for printing a message to the user and restarting the main loop from the beginning, depending on conditions.

1. Get Input sub-process (:getinput)  
     
   The reusable get input sub-process records user input into register RC as a 1-3 digit numeric value and returns to the location stored in RE through a conditional jump to an address (AJMP).  
     
   The code is outlined in [Accepting User Input](#_Accepting_User_Input) of the Reusable Code section of this document.
2. Print Bits output sub-process (:printbits)  
     
   The reusable print bits sub-process writes out the contents of register RC to the output display as a human-readable binary number.  
     
   The code is outlined in [Printing Register Value as Bits to Output](#_Printing_Register_Value_1) of the [Reusable Assembly Sub-Processes](#_Reusable_Assembly_Sub-Processes) section of this document.

# Additional Resources

## Circuit Figures

### The CPU, ROM, Keyboard, and Output (Figure 1)

The below figure demonstrates the completed CPU, along with dedicated ROM memory. The commands from ROM are fed into the CPU as 16-bit commands. Register RH returns from the CPU as the memory address to access command. Register RH is a counter register that automatically increments unless updated by a SET or JMP command. Registers RF is used as the output register for the display on the right.

Diagram

Description automatically generated

Figure 1 The CPU, ROM, Keyboard, and Output

### Inside the CPU: Control Unit, ALU, and Register Bank (Figure 2)

The CPU takes a 16-bit command and passes that to the control unit, which decodes that command into the operand (seen as the 10 1-bit connections extending into the ALU and 6 additional 1-bit connections signifying non-ALU commands such as SET, MOV, and CJMP).

Diagram

Description automatically generated

Figure 2 Inside the CPU: Control Unit, ALU and Register Bank

### Inside the Control Unit (Figure 3)

The control unit is responsible for splitting the 16-bit command into 16 different operations, an input constant and 3 input register addresses). As seen below, the current command is “CJMP”. For this command the CU combines the middle 8 bits of the command into a single 8-bit constant. It also takes the last 4 bits of the command as the output register. For this particular command, the two input registers are ignored.

Diagram, schematic

Description automatically generated

Figure 3 Inside the Control Unit

### Inside the Register Bank (Figure 4)

The register bank is comprised of 8 registers. I implemented a custom register circuit for my registers instead of relying on the Flip Flop registers available.

The registers continuously output their value to their corresponding “Constant Output” and conditionally output their value to Output A and / or Output B if they are designated as Register A and / or Register B.

The registers update their value when the Clock signal goes low.

Diagram, schematic

Description automatically generated

Figure 4 Inside the Register Bank

### The Write Register Selector (Figure 5)

Within the register bank, you will see a write register selector circuit. This circuit accepts a register address and constant input value. It outputs the input value and a write flag to the appropriate output register based on the register address. The register address internally is implemented as a simple 8-bit data input where 1 bit represents each register.

A picture containing bar chart

Description automatically generated

Figure 5 The Write Register Selector

### An Individual Register Implementation (Figure 6)

Within an individual register we have an 8-bit input for the value, a write flag, and an output A and output B flag indicating if that register should return a value as Argument A and/or Argument B. Note that with this implementation a register can participate in a command as both argument A, argument B, and output as argument C.

The register value is stored in a Flip Flop and outputs its value on up to 3 8-bit outputs. Each register always outputs its value to the “Constant Output” path, along with A and/or B path.

Diagram

Description automatically generated

Figure 6 An Individual Register Implementation

### Inside the ALU (Figure 7)

Diagram

Description automatically generated

Figure 7 Inside the ALU

The ALU accepts two 8-bit input values (A and B). The ALU accepts 10 flag inputs for the 10 implemented ALU commands: Compare, Subtract, Addition, Decrement, Increment, Bitwise Not, Bitwise And, Bitwise Or, Left Shift, Right Shift.

It has 3 outputs:

* 8-bit output value
* Carry Flag
* Overflow Flag

The other 6 operations are implemented within the control unit, as they are not arithmetic in nature.

### Inside the 8-bit Adder (Figure 8)

Addition and subtraction operations are both performed by the 8-bit adder circuit. This circuit accepts two 8-bit values and a flag for addition or subtraction. It chains 8 1-bit adders together with carry flags to combine into an 8-bit output along with a carry and overflow flag. This design is an adaptation of the 1-bit adder demonstrated in Building an ALU Using Logism (Study.com, n.a.).

Diagram, schematic

Description automatically generated

Figure 8 Inside the 8-bit Adder

### Inside the 1-bit Adder (Figure 9)

The 1-bit adder accepts 2 1-bit input flags (A and B), along with an input flag for the carry flag of chained 1-bit adders and an output flag to chain to additional adders. This allows any number of 1-bit adders to be chained together with carry flags passed along them. It also has a single 1-bit output. This design is an adaptation of the 1-bit adder demonstrated in Using Logism to Build Half & Full Adders (Study.com, n.a.).

Diagram

Description automatically generated

Figure 9 Inside the 1-bit Adder

### Inside the 8-bit Equality Checker (Figure 10)

Like the 8-bit adder, the 8-bit equality checker combines multiple lower-level circuits. In this case split the 8-bit comparison into two 4-bit comparisons.

1. When both 4-bit comparisons are equal, the 8-bit comparison is equal.
2. If the high 4-bit comparison is greater, then result is greater.
3. If the high 4-bit comparison is lesser, the result is lesser.
4. If the high 4-bit comparison is equal, then the result is based on the lower 4-bit comparison.

Diagram

Description automatically generated

Figure 10 Inside the 8-bit Equality Checker

### Inside the 4-bit Equality Checker (Figure 11)

The 8-bit equality checker is built on 2 4-bit equality checkers. The 4-bit equality checker is built on 4 1-bit checkers. The 4-bit checker is more complex than the 8-bit as it requires several more gates to combine the 4 1-bit checkers within it.

Diagram

Description automatically generated

Figure 11 Inside the 4-bit Equality Checker

### Inside the 1-bit Equality Checker (Figure 12)

The 1-bit equality checker is quite simple. If A is high and B are both high or low it returns Equal. If A is high and B is low, it returns greater. If B is high and A is low, it returns lesser.

Diagram, schematic

Description automatically generated

Figure 12 Inside the 1-bit Equality Checker

### 8-bit Left and Right Shift Circuits (Figure 13 and 14)

For both bit-shifting circuits, I used two splitters connected to each other. These splitters discard the excess bit and include a constant 0 for the additional bit.

Diagram

Description automatically generated

Figure 13 8-bit Left Shift

Diagram

Description automatically generated

Figure 14 8-bit Right Shift

### 8-bit to ASCII Encoder (Figure 15)

The output display component accepts a 7-bit input representing the ASCII value to display. Our registers are 8-bit, so there is an extra bit (the high bit), not used by the display.

The 8-bit to ASCII encoder circuit uses a splitter to separate out that extra bit and pass it along to be used as a “Clear Input” flag combining the remaining 7-bits into an output suitable for the display.

Diagram

Description automatically generated

Figure 15 8-bit to ASCII Encoder

### 8-bit from ASCII Decoder (Figure 16)

The keyboard outputs only 7-bits representing the current ASCII encoding character. This decoder adds an additional constant bit as the 8th bit and combines that back for input into a register.

Diagram

Description automatically generated

Figure 16 ASCII to 8-bit Decoder

## Reusable Assembly Sub-Processes

The programs written in Phase Four reuse two sub-processes that I developed and tested on the 8-bit CPU. While adding and shifting registers is simple in assembly, the process to accept input and output from the user takes quite a bit more lines of code.

### Accepting User Input

:getinput Collects a 1 to 3 digit numeric value from user and writes it to register RC. Returns to address at register RE when the process completes.  
  
Output: RC numeric value entered by the user (1-3 digits)

Input: RE address in ROM to return when the operation completes

The first section of code checks for several conditions:

1. Is there input on register RG
2. Is the input on register RG 10, the “ENTER” value.
3. Is the input on register RG at least 48, the ASCII code for “0”
4. Is the input on register RG no more than 57, the ASCII code for “9”

A picture containing text, receipt, screenshot

Description automatically generated

Once we determine the key entered is a valid value between “0” and “9”, we need to adjust it from ASCII encoding to a numeric value and add it to Register C, where we are combining all digits by using a bit-shifting technique to multiply by 10: This is covered in the next section.

To add the current ASCII value as the appropriate digit of RC we need to determine how many digits are in RC. Logic for each digit is separated three additional sub-sections.

digit1:  
If the current value in RC is zero, we are on the first digit. Just subtract 48 (ASCII “0”) from RG and move into RC. Then jump to the valid input section, which will print out that character, clear the keyboard and loop.

A picture containing table

Description automatically generated

digit2:  
If the current value in RC is between 1 and 9 we are on the second digit. Multiply the current value in RC by 10 through a bit-shift multiplication technique seen in lines 56-61.  
  
RC = RC \* 2 (SRT x1) + RC \* 4 (SRT x2) + RC \* 4 (SRT x2)

A picture containing text, receipt, screenshot

Description automatically generated

Binary multiplication is an involved process, as outlined in the Binary Division & Multiplication: Rules and Examples *(Study.com, n.a.). The approach taken here simplifies the multiplication process by breaking it into three separate bit shifting operations and adding the results.*

*This technique works because we know that X \* 10 = X \* 2 + X \* 2 + X \* 2 X \* 4. Each of these multiplications can be accomplished by shifting bits in the register value.*

digit3:  
If the current value in RC is greater than 9 we are on the third digit.

We proceed just as digit2, multiplying RC by 10 and then adding in the new digit.

However, we only support 3 digits, so we end the input automatically and return to the calling process.

Table

Description automatically generated with medium confidence

The next sections of the get input sub-process print out the digit for user feedback and loop or return when no more digits are possible. We also return if the user hits ENTER before entering a third digit.

Table

Description automatically generated with medium confidence

The final section prints the last digit entered by the user and returns to the calling process using AJMP.

### Printing Register Value to Output

:print Prints the value in register RC to the output. This process overwites the values in registers RA-RC. It returns to address at register RE when the process completes.

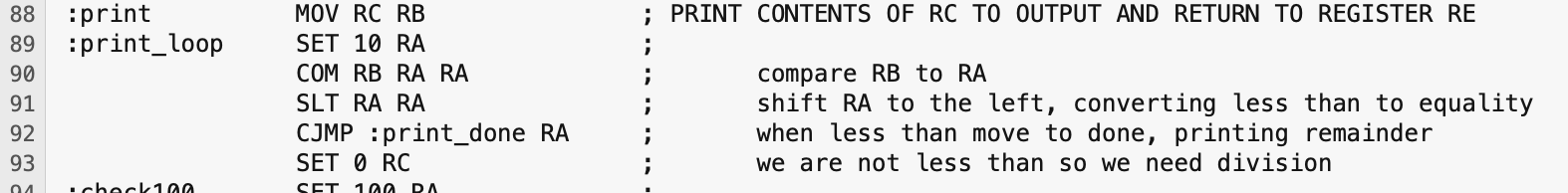
Input: RC numeric value printed to output. When complete, RA-RC will be ZERO.

Input: RE address in ROM to return when the operation completes

To print the value of a register we need to retrieve the individual digits of the number and print each one out. I do this by generally following this process:

1. If RC < 10 just print it and return.
2. If RC > 100, repeatedly subtract 100 until RC < 100. Then print that count and repeat from step #1
3. If RC < 100, repeatedly subtract 10 until RC < 10. Then print that count and repeat from step #1

The first section of code checks if the value is less than 10. If it is, we jump to printing out the remainder, which in this case is the original value:



The next section checks if the value is greater than 100, and if so, loops subtracting 100 and tracking the quotient until the value falls below 100. Once below 100, we jump to print\_next: to print the quotient and repeat the outer loop.

Table

Description automatically generated with low confidence

The next section performs the same logic by repeatedly subtracting 10 from the value and tracking the quotient until the value falls below 10. Once below 10, we jump to print\_next to print the quotient and repeat the outer loop.

Text

Description automatically generated

The final section prints the final remainder (or original value if it was already less than 10) and returns to the calling process referenced in register RE:

Table

Description automatically generated with low confidence

### Printing Register Value as Bits to Output

:printbits Prints the value in register RC to the output as a binary number. This process DOES NOT overwrite the value in register RC. It returns to address at register RE when the process completes.

Input: RC numeric value printed to output. When complete, value in RC is retained.

Input: RE address in ROM to return when the operation completes

The process of printing a register value as bits to the output display is straight-forward compared to printing as ASCII-encoded decimal value. The code for this sub-process follows:  
  
A screenshot of a computer

Description automatically generated with low confidence

To accomplish the printing, we set a secondary register RA to 128 (10000000) and loop through a simple process:

1. Apply a bitwise AND comparison between RA and RC, writing to a third register RB.
2. We then compare that results back to RA.
3. When that comparison is equality, we jump to a sub-section that prints an ASCII “1”.
4. Otherwise, we continue to a sub-section that prints an ASCII “0”.
5. At the end of both sections we jump to the section “:bitdone”.
6. Within that section we right shift RA (shifting 10000000 to 01000000 and so forth).
7. If the shifted value in RA equals ZERO, we have printed all digits and jump to the final section.
8. If the shifted value in RA does not equal ZERO, we repeat this loop from step 1 above.
9. In the final section, we write a carriage return and jump back to the calling address, stored in RE

## Excel Assembler

Writing assembly language and manually converting it into the machine language required by the system quickly becomes tedious. The compiled machine language is stored in 16-bit hexadecimal encoded words. Those words must be converted from the 16-bit binary machine command.

To speed development and testing of my sample applications, I created a simple Excel document that acted as an assembler. This Excel document converted assembly language into the machine language commands and those commands into the hexadecimal encoded words required by the ROM module in logism. A simple copy-and-paste operation from the Excel document to a text-file allows the full program to be loaded into logism ROM module. See below:  
  
Graphical user interface, application, table, Excel

Description automatically generated

Figure 17: Excel Assembler Demonstrating Typing App

The figure above demonstrates the 7 lines of code required by my assembly language to copy the ASCII keyboard input to the output and loop. This emulates basic typing directly from keyboard to output display. The basic functionality of the Excel document follows:

Column A: The hard-coded program line number, starting at 0.

Column B: An optional label for the program line number. ***The label must begin with :.***

Column C-L: These hidden columns accept the pasted, tab-separated assembly language.

Column M: A spacer column

These columns are all calculated, based on the above hidden columns C-L:

Column N: The full assembly command parsed from the pasted text in columns C-L

Column O: The operation of the assembly command (ADD, SET, COM, MOV, etc).

Column P: The A argument. Either a register (RA – RH), a constant number, or a label name.

Column Q: The B argument. This is optional depending on the command.

When not used by the command the binary value “0000” is used as a placeholder.

Column R: The C argument. This is generally the output register for the operation.

Column S: The descriptive pseudo code of the command.

Column T: The 4-bit binary encoded command from Column C

Column U: The 4-bit binary encoded register from Column D representing Argument A or,

for SET and JMP commands the 8-bit encoded constant value from Column D.

Column V: The 4-bit binary encoded register from Column E representing Argument B or,

For SET and JMP commands this is left blank

Column W: The 4-bit binary encoded register from Column F representing Argument C.

Column X: The full 16-bit machine command (concatenation of I, J, and K).

Column Y: The 4-character hexadecimal representation of column L.

For a full copy of the file, refer to the File References section.

## Logism Implementation and Code Examples

All source code mentioned within this document along with the Logism project are available at my github:

<https://github.com/jeffbcates/8bitcpu>

## External References

Study.com. (n.a.) *Binary Division & Multiplication: Rules & Examples.*

<https://study.com/academy/lesson/practical-application-for-computer-architecture-computer-arithmetic.html>

Study.com. (n.a.) *Using Logism to Build Half & Full Adders*

<https://study.com/academy/lesson/how-to-use-logisim.html>

Study.com (n.a.) *Building an ALU in Logism*

<https://study.com/academy/lesson/building-an-alu-using-logisim.html>