# Assembler Implementation

# In order to write programs using the instructions specified to run on the processor, an Assembler was built to transform the programs from assembly code to machine code. The Assembler was written in C++ and compiled to run in a UNIX based environment. To implement the Assembler, a few limitations were set on the format of the assembly code. For instance, the Assembler assumes that the fields of an instruction are separated by spaces and given in the following format:

# <opcode> <flags> <operand> [<//comment> ]

# Additionally, the operand is required to be in decimal form. This is because the Assembler was built to only handle decimal to binary conversion. The optional ‘//’ signifies the Assembler of a comment, the Assembler ignores everything on the line at and beyond the comment symbol. Below is a description of the flag symbols the Assembler recognizes and their meaning.

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Description** | **Binary Representation** |
| **$** | Direct Memory Addressing Mode | 000 |
| **($)** | Indirect Memory Addressing Mode | 001 |
| **#** | Immediate Addressing Mode | 010 |
| **<** | Left shift in zero | 000 |
| **<<** | Left shift in one | 001 |
| **>** | Right shift in zero | 010 |
| **>>** | Right shift in one | 011 |
| **=** | Equal | 000 |
| **!=** | Not Equal | 001 |

# With the above specifications a token class was created to store all of the fields of the assembly instruction and a scanner function was built into the Assembler to scan and parse the instructions into tokens. Once the scanner identified and validated the opcode and flag of an instruction to be correct, structures were referenced that contained the binary representation of those values. For instructions that have NULL flags and/or operand fields the scanner placed the correct number of zeros in those fields. Once the entire program has been parsed, the scanner returns a vector of tokens to the calling function. Next, the calling function iterates through the tokens, printing the machine code equivalent of each instruction to an output file specified by the user.

# The input and output files of the Assembler are passed as command line arguments, here is how the Assembler is called on the command line: ./assem <input> <output>

Furthermore, it is worthwhile to note that the machine code is output by the Assembler in a Verilog format which allowed us to simply copy and paste directly into Xilinx.

**9.1 Assembler.cc**

**9.2 Token.h**

# Testing and Verification

## Test Programs

### Straight Line Control Flow

For the first test program, a straight line control flow was implemented that used as many of the instructions as possible without breaking the control flow. This program begins with a one being loaded into the Accumulator register and then performs a number of math operations on the Accumulator. The program ends with a decimal value of 109 stored into register B. Additionally, many instructions were repeated with different addressing modes or operations based on different flags.

### Straight Line Control Flow with Looping Structures

The second program implemented a straight line control flow with conditionally exited looping structures. This was accomplished by writing a program to calculate the first ten Fibonacci numbers and storing them to the first ten memory addresses in data memory. This program allowed us to confirm the functionality of our conditional code register, and conditional branch instruction. In addition, the program assumes that the first two Fibonacci numbers are stored into memory location zero and one.

### Subroutine Execution

The third program tested the architectures ability to execute programs containing subroutines. This program calculated the cube of a number n, where n is less than 7, and was implemented by writing a recursive program which called itself and repeatedly added n to the accumulator, storing the final value into memory location one. Additionally, this program assumes the value of n to be stored into memory location five. Furthermore, the subroutine program confirmed the correct operation of the branch to subroutine and return from subroutine instructions, as well as the jump instruction and the processors ability to execute nested programs at depth of n.

### HVPI System

## Validation

### Straight Line Control Flow

The straight line control flow program, contained in the file streamline.a, allowed verification of most of the instructions and all of the different addressing modes. To correctly validate the operation of this program, the values of the accumulator, registers, and memory addresses referenced were calculated several times by hand and included in the comments of the assembly code. Next, when the Processor was simulated and tested with this program, we were able to confirm the correct values were contained in all the appropriate places before and after every instruction and on a clock cycle basis.

### Straight Line Control Flow with Looping Structures

Verification of the program, which is stored in the file fib.a, required that the processor be simulated over a long enough time period for the program to complete. Then it is observed if the correct Fibonacci numbers were stored into the first ten memory locations.

### Subroutine Execution

In order to confirm the correct execution of the subroutine program, named sub.a, memory location one is checked after the end of the program is reached. The end of the program was signified by an infinite loop of nop instructions. If data memory one contains the cube of the value of memory location five, then program executed correctly. Further confirmation is necessary by validating that the PC stack pops off the correct instruction locations at the end of the program. The last pop off of the PC stack will be 5 if the program executed correctly. Then the program proceeds to execute the infinite loop of nop instructions.