Pixie Wranglers' Computer System Final Project Documentation

Kyle Lemmon, Yuntong Lu, Haoze Zhang,

Contents

1 Overview

Our project set out to build a functional computer, capable of three basic types of I/O, glyph based output to a VGA monitor, PS/2 keyboard input, and read and write access to a storage block with a basic filesystem on it. Many compromises in complexity had to be made in order to finish the project, and these are discussed in the sections below.

The goal of the entire system is to be able to read and write files, and for every program that is run on the system to be loaded onto the device as a file. That is to say that every program on the computer comes from the same place, and that is somewhat maliable in the sense that it can be reprogrammed fairly easily.

The design of the system allows for a wide range of possibilities, and with some further improvement and more software written for the device, it would be possible for the device to reprogram itself. If a hex-based text editor were written and included on the filesystem that was capable of reading, editing in hexadecimal, and writing back files to the file system storage, then this machine would be completely reprogrammable. Not only could new programs be created, but existing programs could be edited, including the kernel itself.

2 Application

Application Design Kyle Lemmon

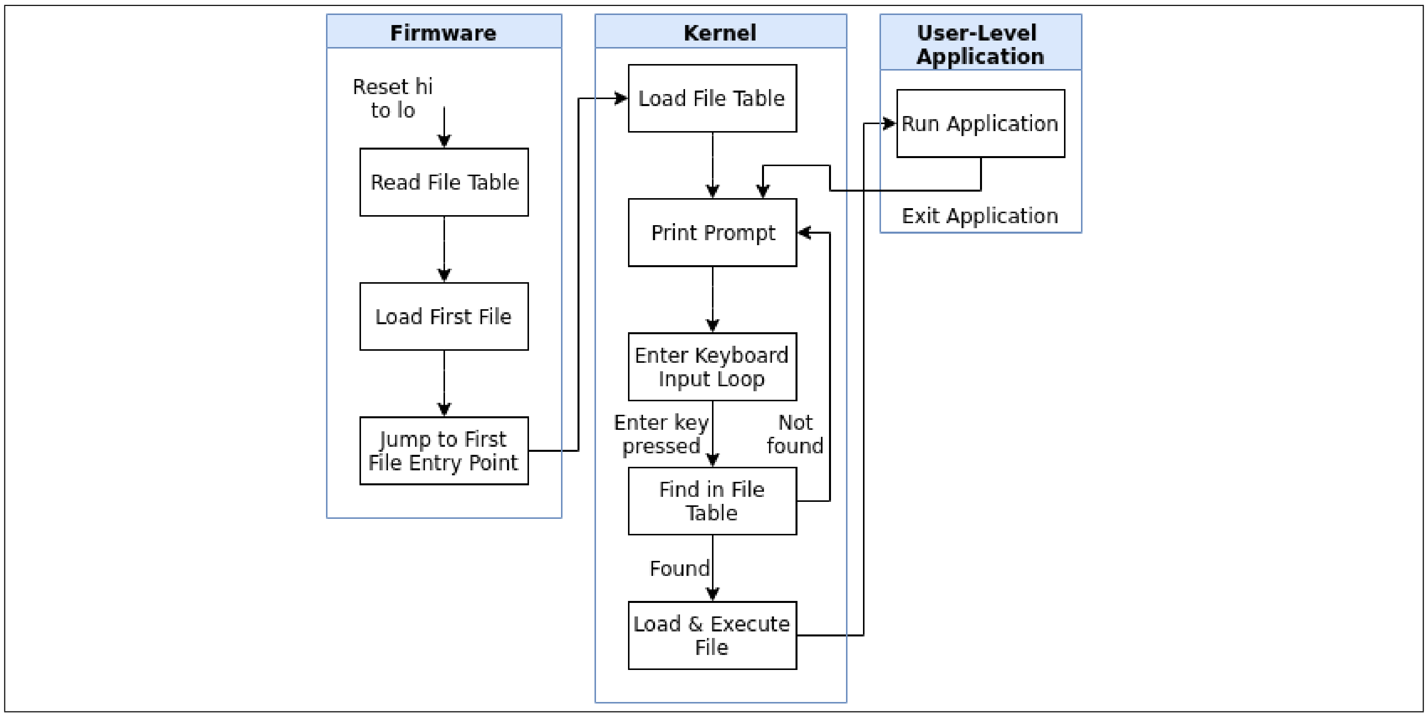


Figure 1: Demonstration of application flow between firmware, kernel, and user level application.

Since the goal of this project is to emulate the functionality of a modern personal computer, the software is somewhat complicated. By design, it must be able to handle I/O and be able to load programs into memory and execute them. Because we wanted this system to be fully upgradable software-wise, the kernel, or operating system is not encoded into memory. It is only stored in the file system storage. Thus, our application consists of three main categories: firmware, kernel, and user level applications.

The firmware's only job is to load and execute the first program listed in the file system. This first program is the kernel, or operating system of the computer. This simple task does not need to be changed, and as such this is loaded into a read only portion of the memory and is executed on reset.

The kernel performs a more complicated task. After being executed, it enteres an input loop that is expecting the user to enter a string that corresponds to another file on the file system. This other file should refer to another program. When the user enters a command that exists on the file system,then it is loaded into

Family	**Function**	**ARG1**	**ARG2**	**RET**
		r10	*r11*	*r12*
String	.STR_LEN	str_addr		
	.STR_EQ	str_addr_a	str_addr_b	are_equal
	.STR_CPY	str_addr_src	str_addr_dest	
Keyboard	.CHECK_KEY	key_code		key_data
	.KEY_NUMPRESSED			key_num
Flow	.DELAY	ticks		
	.PROMPT			
Display	.PRINT_STR	str_addr_src	str_addr_dest	
	.PRINT_WORD_HEX	word_addr_src	str_addr_dest	
	.SET_PRINT_COLOR	color_code		

Figure 2: Table of public kernel subroutines.

memory and executed. The kernel has another job though, and that is to provide a library of subroutines that can be jumped to by the loaded program in order to manipulate data, or interact with the I/O.

Finally, the user level program, which can be anything. The open-ended nature of this design allows a programmer to create any simple program and run it on this hardware without necessarily knowing the ins and outs of the hardware. The idea is with a basic understanding of the memory map, any proficient assembly programmer should be able to use the kernel functions to accomplish the goals of a basic program. The flow of all of this software on the processor is specified in fig 1.

2.1 Firmware

Firmware package is preloaded into FPGA RAM. Firmware accesses file allocation table and finds the first and second entry. The length of the first entry is determined by the start address of the second address via software.

The firmware then loads the first file in the allocation table into memory starting at the first address. When the firmware has completed, the firmware jumps to the first memory address, handing control over to the kernel.

2.2 Kernel

The objective of the kernel is to provide a command-line interface for running programs that exist on the file system. The kernel provides all functions for interfacing with I/O that are called from within the kernel and from user programs executed on the processor.

2.2.1 Subroutines

There are various subroutines available to user programs that are described below.

2.2.1.1 String Functions

- STR_LEN(str_addr): Accepts the starting address of a string, and counts characters until the null termination of the string, then returns the length of the string, excluding the null terminator, in the return register.
- STR_EQ(str_addr_a,str_addr_b): Accepts the addresses of two strings, and iterates through them until either they have two different characters at the same index, in which case a zero is returned, or until they have both terminated at the same point, in which case the strings are equal and a one is returned.
- STR_CPY(str_addr_src,str_addr_dest): Accepts two string addresses and copies a string from the source address to the destination address, stopping after the null terminator in the source string was reached.

2.2.1.2 Keyboard Functions

- CHECK_KEY(key_code): Accepts a key code and checks to see if that key code is currently pressed. If the key is currently pressed, 1 will be returned, otherwise a zero will be returned. Non-blocking.
- KEY_NUMPRESSED(): Will return the key code of the next pressed key. If more than one key is pressed, then usually the lexographically smaller (ASCII code closer to zero) key will be returned. Occasionally it will be possible for a higher lexographical key to be pressed, but this is unlikely. Blocking.

2.2.1.3 Flow Functions

- DELAY(ticks): Accepts a number of ticks to delay by. A tick in this case is exactly 50000 clock cycles, or $\frac{50000}{50000000} = 1\text{ms}$.
- PROMPT(): Returns execution flow to the kernel, effectively a way for a program to exit. The screen is not flushed except for the prompt line and above. Anything written below the prompt line will persist until another program is run.

2.2.1.4 Display Functions

- PRINT_STR(str_addr_src,str_addr_dest): Accepts two string addresses, and copies the string at the source to the location at the destination, with one exception. The color bits of every character copied to the screen will be set to the value saved when SET_PRINT_COLOR was called last.
- PRINT_WORD_HEX(word,str_addr_hex): Accepts a word value stored in the word register, and the address to print the word to. This function writes starting at the destination address four characters, that is the hexadecimal representation of the word that was supplied to it.
- SET_PRINT_COLOR(color_code): Accepts a color code, stored in the lower 8 bits of the color_code word. This is stored and all subsequent print functions called in the kernel will use this stored color. When reset, the color code is initialized to white letters on black background.

2.3 User program

The user program is similar to any program run on a modern machine. The programs are written to the SD card and are loaded into memory by the firmware. They define their own functionality, but it is expected that for I/O from the SD card, the program uses the kernel's defined SD card access functions. The keyboard input and VGA output are handled by shared memory, which the user program may access itself.

2.4 Assembler considerations

The assembler must handle jumps to kernel code by outputting label spec files that contain the relative memory addresses of all labels in the compiled code. These files can be linked against in subsequent compilations to ensure that user programs can be linked against kernel subroutines.

2.4.1 Unified memory

The processor is designed with a unified memory structure in order to load files into memory for execution. This allows us to easily include data for our program in assembly code. The assembler inserts data such as null terminated strings and

Addr, Len	Description
0x0000, kernel len	Kernel
kernel len, prog len	User program
...	Empty
0xC800, 512	Kernel - user shared memory
0xD000, 256	Keyboard memory
0xD800, 256	Storage Memory
0xE000, 4800	VGA Glyph Memory
0xFED3, 300	Firmware - reset entry point!

Figure 3: Table showing memory layout of the processor design.

2.4.2 Assembler Use Flow & Kernel Linking

The assembler must support linking accross files via a linking file. The flow for compiling and creating user programs is as follows:

1. Assemble Kernel → kernel.hex, kernel.ln

   kernel.ln is a text file that links a label with a memory address. This can be included in a user program assembly code, which allows the use of all labels from of kernel source assembly from within the user program code. It is similar to the a header file for the c standard library functions and memory addresses.

2. Assemble user program → prog.hex, prog.ln

   The assembler checks if any undefined labels used in the user program code are defined in the linking file, if so the address from the link file is used.

This system works without further hardware modification because the kernel is addressed by the firmware to start at address 0, so no offset is required for jumping to a kernel address. The link file also contains an end address, that is essentially the size in bytes of the compiled kernel code less one. This is used to calculate offsets for the user program's labels, since it exists above the kernel in memory.

This requires that all user programs are recompiled after a change to the kernel.

2.5 Memory Map

Because the processor is using unified memory, it is relatively simple to have multiple applications in memory on the device. In order for this to work well though, the memory map must be well defined so that applications do not step outside their boundaries, and so that the locations of common resources and kernel subroutines are known at compile time. This is necessary for user applications to be able to jump to kernel functions. Fig. 3 shows the memory boundaries for the different I/O memory sections as well as where the firmware, kernel, and user programs are located.

2.6 Filesystem

Filesystem Documentation Kyle Lemmon

2.7 File System

The file system allows for multiple files of different names and varying lengths, up to 8192 words. The design of the filesystem and the interface would allow for a maximum file size of 65536 words, however compilation of memory of this size was unfeasably long, so the decision was made to cap the file size at 8192.

The file system consists of a file allocation table, which specifies the location of up to 16 other files (again, this could be larger, however the decision to limit this length was made to shorten compilation time). The file system is organized into blocks, with each block either allocated, in which case it contains a file that begins from the start of the block, or unallocated, and available to be written to. The table below shows the layout of these blocks, with the first block reserved for the file allocation table, where the names of the tiles and their respective block numbers.

File no.	Description
0	File allocation table
1	File no. 1
. . .	. . .

2.7.1 File Allocation Table

The file allocation table contains three values for every file: its name, its block number, and its length in words, which can be no more than 8192. The table below shows the structre of a single file system entry, where the addresses indicate a word address.

Data Type	Description
String	File Name
Word	File Block Number
Word	File Size

2.8 Programming References

ISA Documentation Kyle Lemmon

ADD	R_{src}	R_{dest}	R_{dest}+= R_{src}	
ADDI	imm	R_{dest}	R_{dest}+= R_{src}	
SUB	R_{src}	R_{dest}	R_{dest}-= R_{src}	
SUBI	imm	R_{dest}	R_{dest}-= imm	
CMP	R_{src}	R_{dest}	$R_{dest} - R_{src}$	Sets comparison flags based on this op.
CMPI	imm	R_{dest}	$R_{dest} -$ imm	Same as CMP.
AND	R_{src}	R_{dest}	R_{dest}&= R_{src}	Bitwise and.
ANDI	imm	R_{dest}	R_{dest}&= imm	Bitwise and.
OR	R_{src}	R_{dest}	$R_{dest}\vert$= R_{src}	Bitwise or.
ORI	imm	R_{dest}	$R_{dest}\vert$= imm	Bitwise or.
XOR	R_{src}	R_{dest}	R_{dest}ˆ= R_{src}	Bitwise xor.
XORI	imm	R_{dest}	R_{dest}ˆ= imm	Bitwise XOR
MOV	R_{src}	R_{dest}	R_{dest}= R_{src}	Set dest equal to src
MOVI	imm	R_{dest}	R_{dest}= imm	Set dest equal to imm
LSH	R_{amount}	R_{dest}	$R_{dest} << R_{amount}$	Amt can be ± 15
LSHI	imm	R_{dest}	$R_{dest} <<$ imm	imm can be ± 15
LUI	imm	R_{dest}	R_{dest}= (R_{dest}& 0xff) $\vert$ (imm$<<$ 8)	
LOAD	R_{dest}	R_{addr}	R_{dest}= mem[R_{addr}]	
STOR	R_{src}	R_{addr}	mem[R_{addr}] = R_{dest}	
J[cond]	R_{target}		jump_if_[cond](R_{target})	
B[cond]	disp		relative_jump_if_[cond](R_{target})	
JAL	R_{link}	R_{target}	jump_link(R_{target})	
LDSD	R_{block}	R_{offset}		Loads 256W from sd card at address (R_{block}*BS) + R_{offset}into map
STSD	R_{block}	R_{offset}		Stores 256W from mmap to addr (R_{block}*BS) + R_{offset}

Figure 4: All available instructions on our finished processor.

15	14	13	12	11	10	9	8
bg[1]	bg[0]	red[1]	red[0]	grn[1]	grn[0]	blu[1]	blu[0]

7	6	5	4	3	2	1	0
chr[7]	chr[6]	chr[5]	chr[4]	chr[3]	chr[2]	chr[1]	chr[0]

Figure 5: Glyph memory bit map, where bg is the background color, red, grn, and blu are respectively the red, green, and blue foreground color channels, and chr is the character identifier (ACII encoded).

2.9 Custom Instructions

Both the LDSD and STSD instructions perform a unique operation that moves a large amount of data. In the original design, these would load and store a 256 word block of data to and from the SD card, and this would take many clock cycles. In the current implementation, these only take one clock cycle, because we are using memory as though it were an sd card.

The actual design of this module is a paging memory block. Only 256 words of memory are accessable at a time. The memory addresses are calculated in hardware based off of a block and and offset value. The block value addresses to any of 16 8192 word file blocks, and the offset addresses to any of 32 256 word page blocks. That is to say, that although there are $8192 \cdot 16 = 131072$ possible addresses in the file storage memory, only 256 consecutive words are available in memory at any single time.

2.9.1 LDSD

The load from SD instruction copies a particular page, R$_{offset}$, of a particular block, R$_{block}$, into the 256 word memory block indicated in Fig. 3 as the Storage memory region. Here the data can be read like any other memory, and can be written to like any other memory. LDSD always overwrites this area of memory, and changes made to it are not saved to the file storage memory, only changed in program memory.

2.9.2 STSD

The store to SD instruction copies the data in the 256 word memory block indicated in Fig. 3 as the SD memory region to the file storage in the block specified by R$_{block}$and the page specified by R$_{offset}$. All 256 words at this block and page in the file storage is overwritten every time with the complete contents of the Storage memory block.

3 IO

3.1 VGA

Display IO Documentation Kyle Lemmon

The display I/O consists of three main components: VGA module, Glyph Memory, and Character Set ROM. These three components govern the vga display output.

3.1.1 VGA Module

The VGA module is responsible for generating a 640x480, 60Hz, full color VGA signal. This module pulls from both the glyph memory and character set rom in order to compute every pixel in the vga display.

3.1.2 Glyph Memory

The glyph memory is accessed as normal program memory, and starts at the address 0xE000. This special memory block is 4800 words long, with each word governing the glyph displayed at that character index. The upper byte of the word specifies the background color and foreground color of the glyph, while the lower byte specifies which character of the character set is displayed. Fig. 5 shows the binary mapping of a single glyph memory word to the available characters, and the possible colors of that glyph.

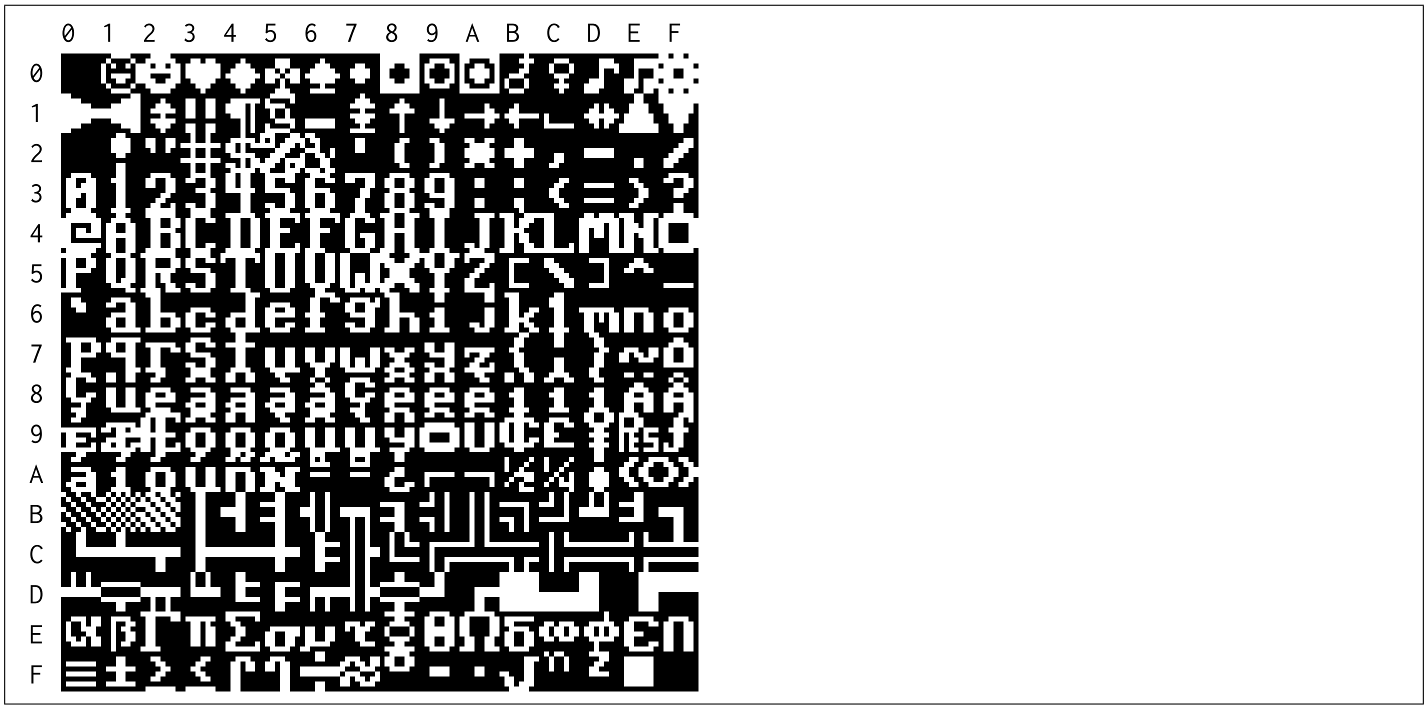

0 1 2 3 4 5 6 7 8 9 A B C D E F
0
1
2
3
4
5
6
7
8
9
A
B
C
D
E
F

Figure 6: The character set embedded in the processor display module. The character's upper nybble is indicated by the left hex digits, and the lower nybble is indicated by the top column digits. **Note: If this image appears blurry, you may need to adjust your pdf viewer's settings.**

3.1.3 Character Set ROM

The character set ROM is non-modifiable and is part of the display module. The character set is similar to ASCII, wherein displayable ASCII characters are mapped to their correct ASCII codes. Non-displayable ASCII characters are remapped to alternate font items in order to have more characters available for display.

The character font is shown in Fig. 6. The codes of all characters increase from zero for the upper left 8x8 block, and increase left to right, top to bottom. In that figure, foreground is denoted as white pixels, and the black pixels are the background.

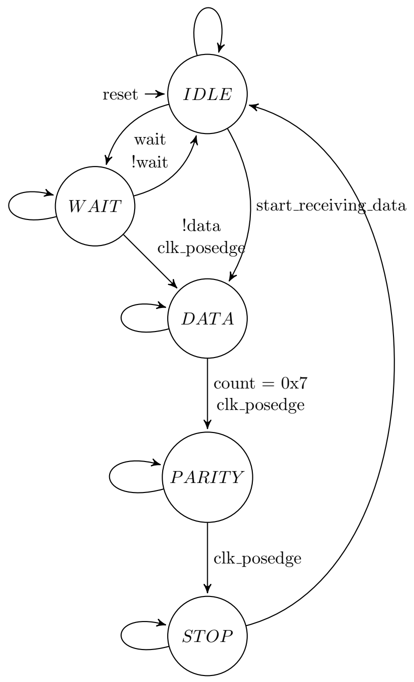
3.2 Keyboard

Keyboard IO Documentation Kyle Lemmon

The keyboard interface is implemented as a simple memory map. There are 101 keys that we recongnize in our project, and as such, we have a memory block with 127 usable memory locations. This memory block is read only from the programmer's perspective, and is written to solely by the keyboard module. Every key is mapped to an address of this 127 block memory module, and when the key is pressed, the corresponding address has a 1 written to it. When any key is released, the corresponding address is reset to be 0. This way the processor can check if more than one key is pressed, and decide what to do. This is useful for keyboard shortcuts or using the shift key.

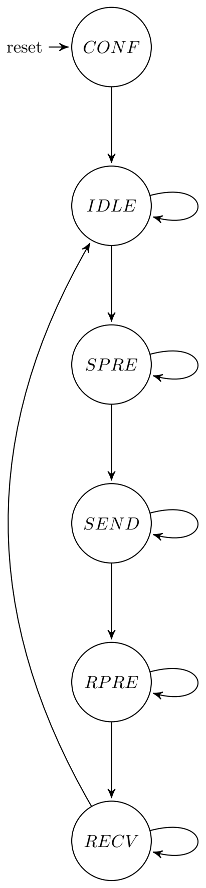
The buttons that have an ASCII value are mapped to their ASCII values, and the ones that do not are mapped to the less commonly used, invisible ASCII characters. This allows a simple conversion to the character set glyphs and makes the programmer's job easier. For reference, the enter key is mapped to ASCII newline character, and the backspace key is mapped to its backspace ASCII key.

3.2.1 PS/2 Module



3.3 SPI Module Interface

This interface did not make it into our final project, but is included in this documentation to document the work that was completed towards the SD card file interface goal.



State	Descridption
CONF	SPI configuration period, cycles clock for 74 cycles as per SD spec.
IDLE	Idle state, clock is held hi and slave select is also held hi. No data is transferred. When the send_cmd line goes hi, the state is pushed to SPRE to begin the transmisson of the 48 bit command.
SPRE	Send precondition. Prepare for transmission of data by sending SS low and MOSI low.
SEND	Transmit data. During this state, the shift register is shifted for each clock cycle. The data is shifted on the falling edge of the sd clock, and the slave should read on the rising edge. Once the entire 48 bit command is sent, the SPI module moves to RPRE.
RPRE	Prepare for reception of data. Wait until MOSI goes low and reception transmission is started.
RECV	Receive data bit by bit. Shift reg is incremented on the clock cycle. A line goes high for one SPI clock cycle after the complete transmission of one byte. When the module controlling the SPI module determines that the reception is complete, a recv_trans_complete wire is sent hi.

4 Processor

Processor Documentation Yuntong Lu, formatted by Kyle Lemmon

4.1 Datapath

Inside of the data path, we connect ALU, regfile, Program counter and memory as the picture shows. For the ALU, the input values come from regfile. The result will go to a MUX. The flags values will be sent to Flag block. Flag block stores 5bit flags. From the lower bit to higher bit, the flags are CLFZN. For the Regfile, the input comes form a MUX and FSM. MUX will choose either ALU result or Memory out. The FSM controls the reset of the input's ports. The outputs will go to ALU, Program counter and Memory. For the Program counter, the ld_pc value is rd2 value which comes from Regfile. The other inputs come from FSM. The output will go to a MUX. For the Memory, the inputs come from either rd2 or pc value. The pc value decides the address value. rd2 value decide the data port. The rest of the input ports come from FSM. The output value will send to a MUX. Fig. 7 shows the datapath.

4.2 ALU

ALU will take care of all the calculate part. The 8 bit aluop will be the OPCODE[15:12], OPCODE[7:4]. Depend on the aluop, the ALU would know which calculation needs to perform. The immediate value has two parts. The IMMHI part will be included into aluop, and IMMLO part will be directly sent to ALU. ALU will also compare the values while calculating them. The flags will be raised depend on the values. Fig. 8 shows the ALU.

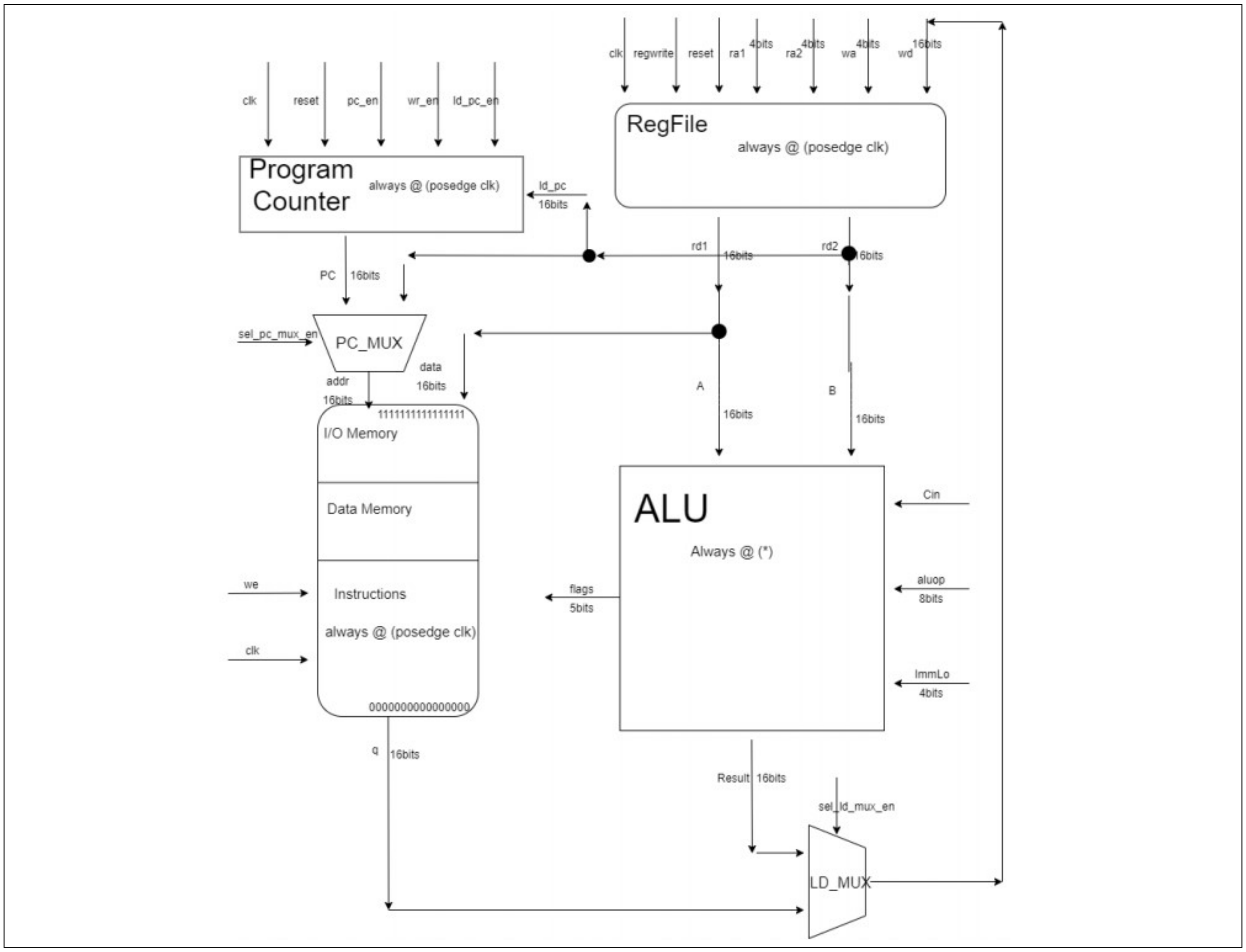


Figure 7: Input and output of the program counter module.

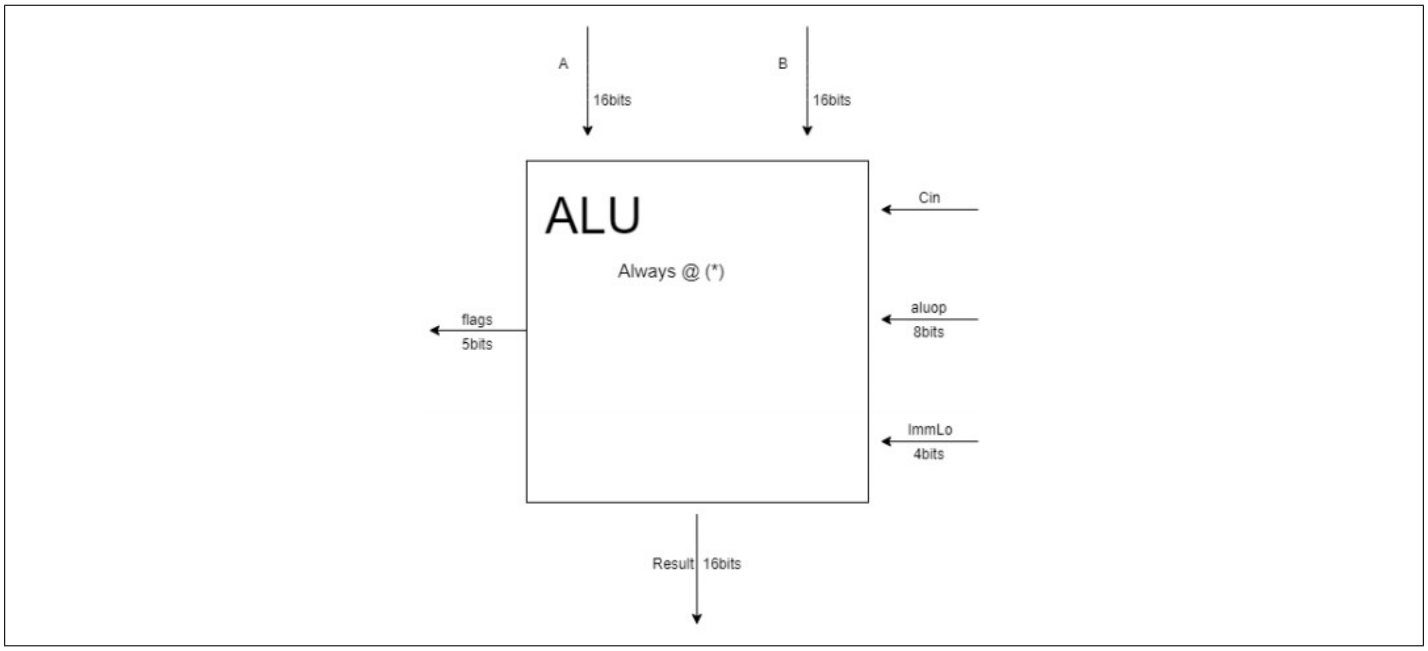


Figure 8: Input and output of the program counter module.

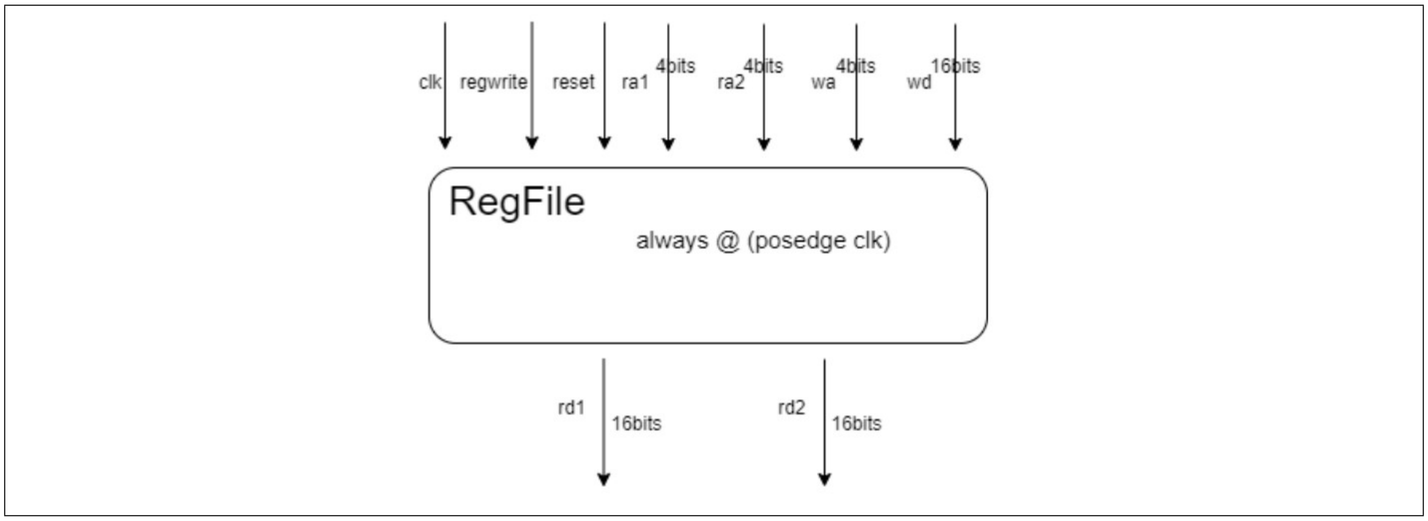


Figure 9: Input and output of the program counter module.

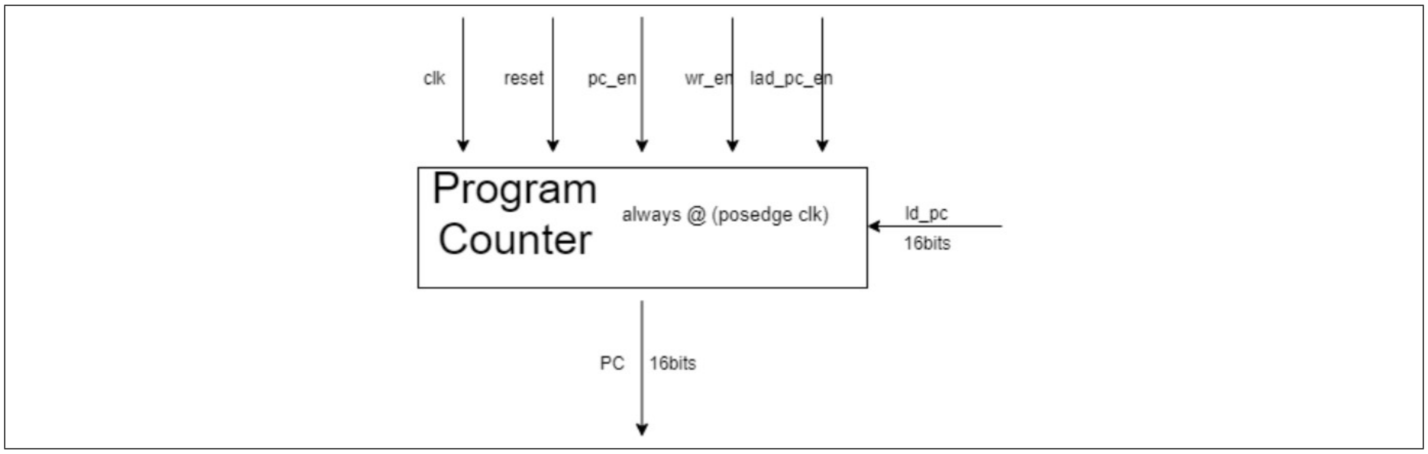


Figure 10: Input and output of the program counter module.

4.3 Register File

Regfile has 16 reg spaces. Each reg space can store 16 bit values. At the posedge of the clock, the value will be written or read. ra1 and ra2 controls the selected address. The regfile has two port outputs, the rd1 and rd2. They can be read together. When the regwrite is enabled, the value will be written to the regfile, the output will be the written value. Fig. 9 shows the register file.

4.4 Program Counter

Program counter has one output pc. The output has three different sources. The first is when pc_en is on, ld_en and wr_pc is off, the pc value will pulse 1 at each posedge of the clock. When pc_en and ld_en is on, the pc value will be current pc value pulse ld_pc at each posedge of the clock. When pc_en and wr_en is on, the pc value will be overwritten to ld_pc value at the posedge of the clock. We can use it to perform JCOND and SCOND by changing the value of ld_pc. Fig. 10 shows the program counter.

4.5 Memory

Memory is generated from templet. add will decides the address. we means write enable, data port will be used when write the new data into the memory. The memory will store instructions from the address 0. The total space is 65535. Each space stores 16bit values. The assembler will decide how to use the memory. Fig. 11 shows the memory module.

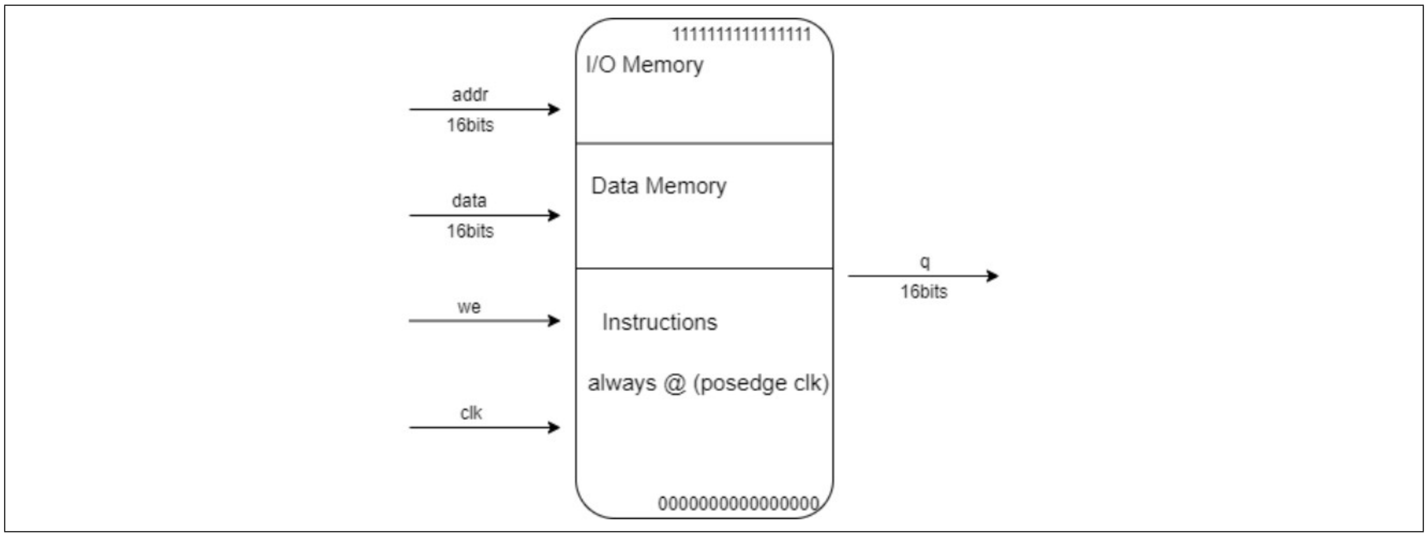


Figure 11: Input and output of the program counter module.

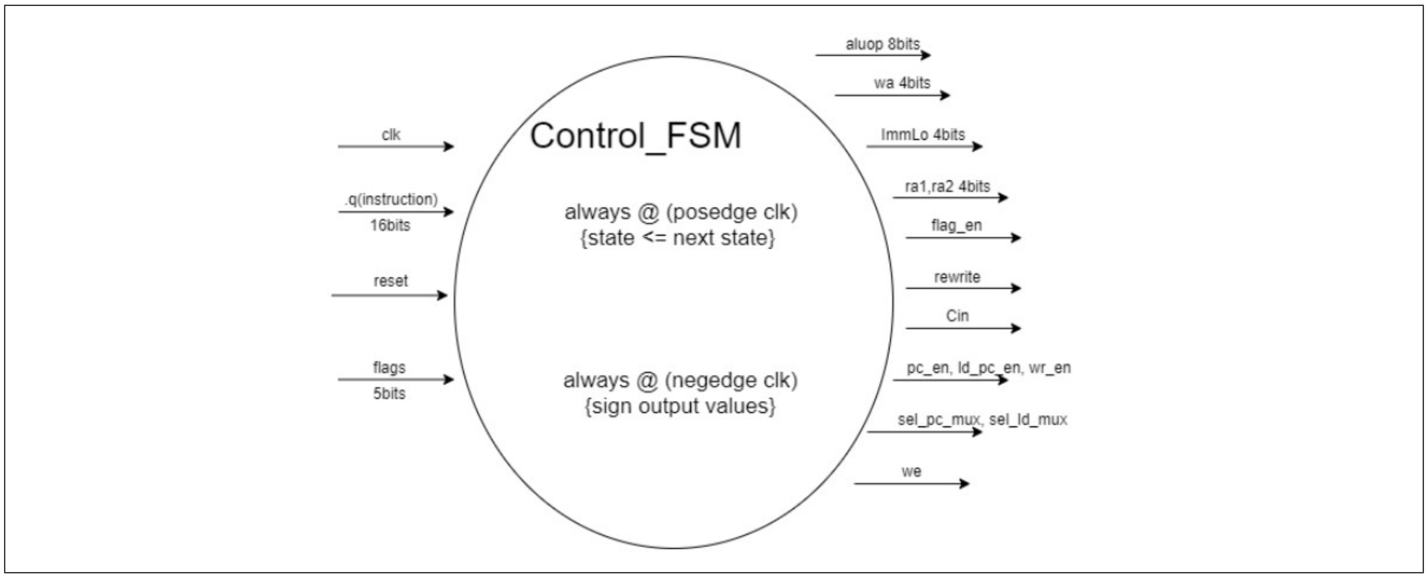


Figure 12: Input and output of the program counter module.

4.6 Control FSM

FSM controls almost all the data path input ports. It just needs the flag information and instructions in the jump conditions. FSM decide when to open and how long to open the port. In the FSM, I put lots of states in here. The states are depending on the instructions. Same with ALU, each instruction has its own state. But the state can be separate into three major part. First is regToreg, which means the value comes from register and goes to register. Second is regTomem, which means the value comes from register or memory and goes to the other side. The third is immediate values, the value comes from instructions and the result goes to register. At the posedge of the clock, the state will update to the next state. At the negedge of the clock, the output value will be decided in the state. For the regToreg and immediate states, it takes 1 clock cycle to finish the job. For the store instruction, it takes 3 clock cycles. For the load instructions, it takes 4 clock cycles, for the jump instructions, it take 3 clock cycles. Fig. 12 shows the processor control module, and Fig. 13 shows the state diagram for the control module.

5 Assembler Usage

Assembler Documentation Kyle Lemmon

The assembler runs in two passes, the first pass gathers the location of all of the labels, and the second pass

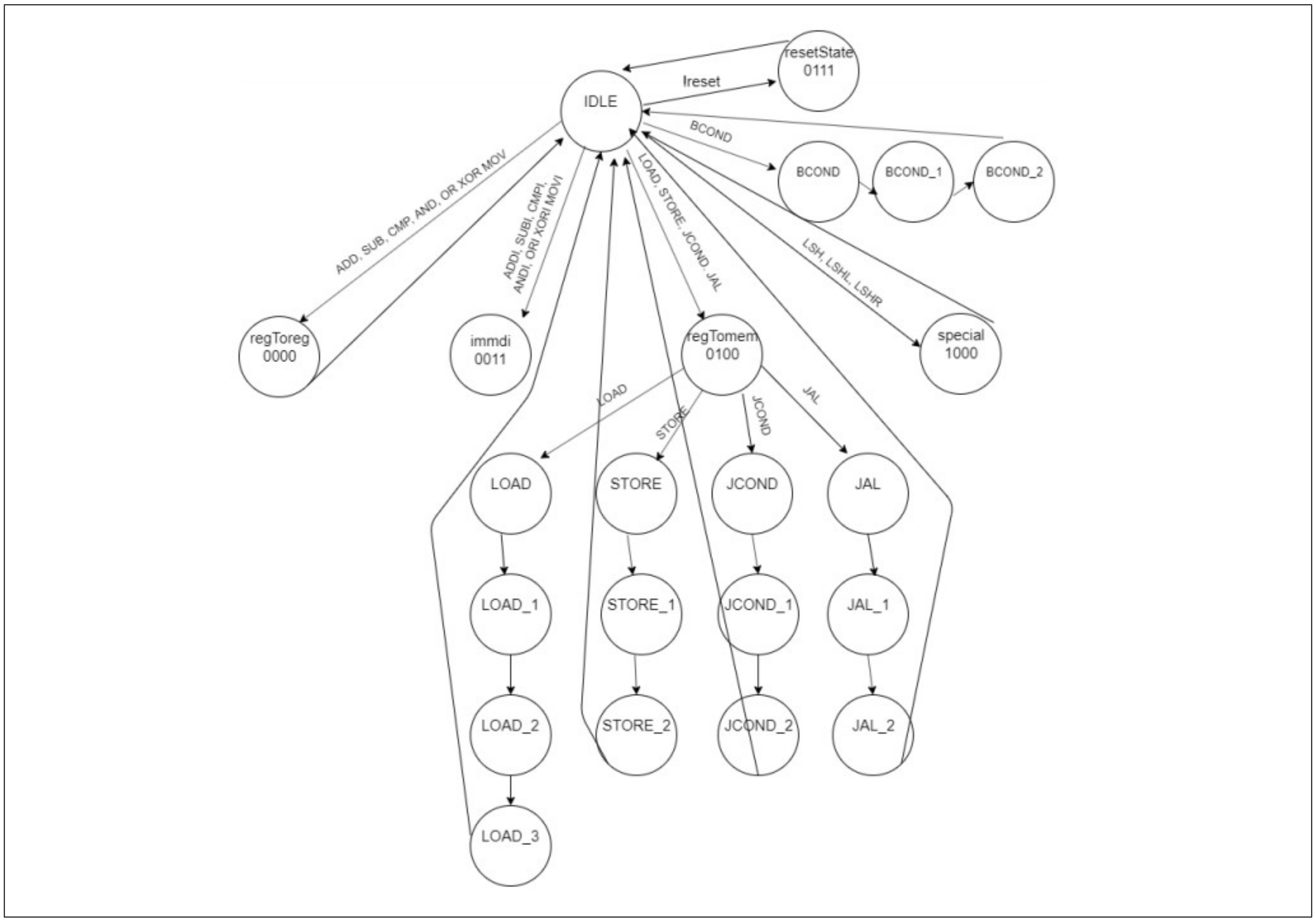


Figure 13: Input and output of the program counter module.

converts the assembly into binary strings. Pseudoinstructions, or instructions that decompose into more than one instruction in machine code are decomposed at this step. The first pass accounts for the decomposition of these pseudoinstructions by adding the appropriate offset to the index of each label when it detects a pseudoinstruction.

In order to run the assembler, only two arguments are needed, the first specifies the input assembly file, and the second specifies the output file. See the provided makefile for examples of how this should work.

There is also a provided python script called fixmem.py that will convert the binary strings into hexadecimal strings, and pad with the appropriate number of zeros.