

# CS203 Project 1

## CPU Simulator

Due: November 15th 2016

Agathe Benichou

# LAFAYETTE COLLEGE

## Table of Contents

User Story -----	2
What is the Central Processing Unit? -----	3
Y86-64 Instruction Set Architecture -----	10
Project Requirements -----	13
High Level Design Overview -----	14
Low Level Design -----	17
Organization of Unit Tests -----	40
Test Programs -----	41
Design Flaws -----	47
Bibliography -----	49

# LAFAYETTE COLLEGE

## User Story

The CPU Simulator Project is to create a simulator for a simple Central Processing Unit (**CPU**) and all of its associated components, shown in Figure 1. The purpose of this simulation is to enhance the understanding of how the CPU works. The CPU is considered the “brain” of a computer and it controls how the computer operates. It does this by processing instructions that it gathers from decoding the code in computer programs. These instructions contain steps which tell the CPU how its various components perform in order to carry out the instructions. This project implements the Y86 Instruction Set which is written in Assembly Language. Assembly Language is a more human readable programming language compared to machine code language, which consists of binary numbers.

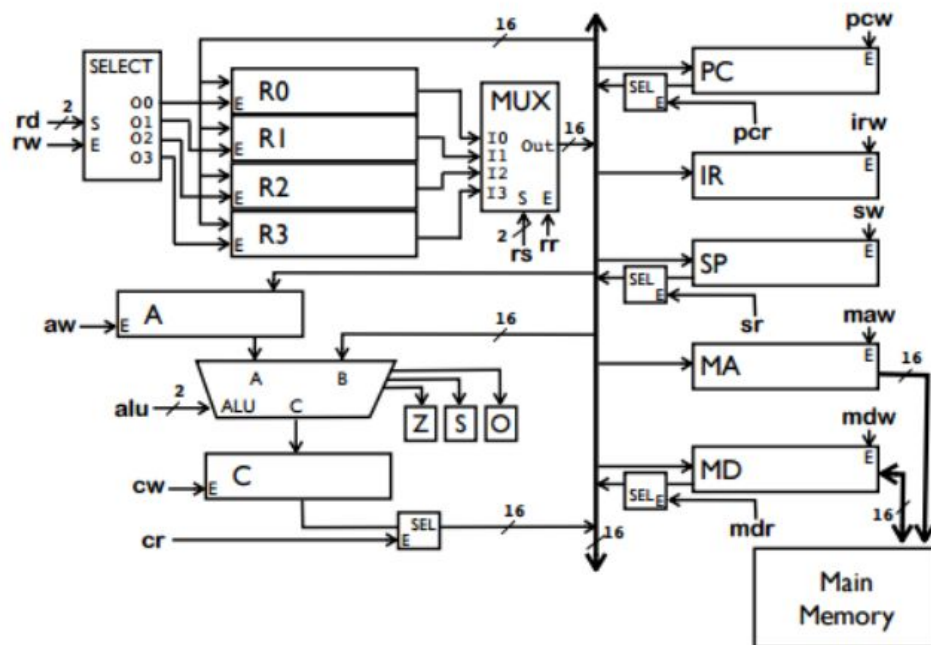


Figure 1a: Overview of CPU Simulator

# LAFAYETTE COLLEGE

## What is the Central Processing Unit? How does it work?

The CPU, or Central Processing Unit, consists of main memory, registers with various uses, buses and controllers. These components are controlled by two big components within the CPU; the Control Unit and the Arithmetic Logic Unit. The Arithmetic Logic Unit performs all the calculated operations that the CPU might need. The Control Unit is in charge of extracting instructions from memory, decoding and executing the instructions using the components of the CPU such as main memory, registers, buses, controllers and the ALU. The main Data Bus is an important part of the CPU. It is the large vertical line that runs down the middle in Figure 1a. The Data Bus transfers data to and from other components of the CPU. The CPU runs using instructions which that **are** represented in byte level encodings known as the instruction set architecture. This project implements the Y86-64 Instruction Set Architecture, which includes the instruction, its binary byte level encoding, and the different components of its state.

The Y86-64 instruction set architecture contains 15 program registers (shown in Figure 2), each of which stores a 64 bit word. They are denoted using the % sign. Registers are temporary storage locations for the data and instructions that CPU uses. Registers are faster to access compared to main memory but they cannot hold as much information as main memory can. Registers usually store an instruction or data that is often used with the CPU, since it will frequently be accessed. Most of these registers are general purpose but the %rsp register is used to point to the part of main memory which stores program data. This project contains only 4 registers, %r0, %r1, %r2, %r3, which are used as general purpose registers. However, there are many other registers such as Program Counter, Stack Pointer, Instruction Register, Memory Address, Memory Data, Register A and Register C, which have specific purposes and storage uses.

%rax	%rsp	%r8	%r12
%rcx	%rbp	%r9	%r13
%rdx	%rsi	%r10	%r14
%rbx	%rdi	%r11	

Figure 2: Y86 Program Registers

# LAFAYETTE COLLEGE

The four general purpose registers are stored separately in the CPU, as seen in Figure 1b. The Data Bus does not have direct access to the Registers, it must go through the Selector to access a Register. The Data Bus sends to the Selector a two bit binary number (since there are 4 registers numbered R0(00), R1(01), R2(10), R3(11), only two bits are necessary), which is related to a Register. The Selector's job is to take the two bit binary number and find which register that number is related to. For example, if the Data Bus sends 00, the Selector identifies R0 (Register 0) as the Register that the Data Bus is looking for.

Main memory, seen at the bottom right in Figure 1b, is the primary storage area of the CPU. It stores the instructions being executed and the data used to process the instructions. Conceptually, memory is a large array of bytes which holds program data and instruction for as long as the program it is pertaining to is operating. Program data is stored in main memory when it is not in immediate use since main memory is slow to access.

Main Memory is implemented as a stack which is a data structure that follows the LIFO or last in first out policy. The stack temporarily stores variables and data created by the current program that the CPU is running. Once a program is finished executing, the stack clears all of the variables stored for that program. The stack grows when variables are pushed onto it and shrinks when variables are popped off. The push operation of a stack just pushes data directly onto the stack, increasing it. The pop operation of a stack removes the most recently added data from the stack.

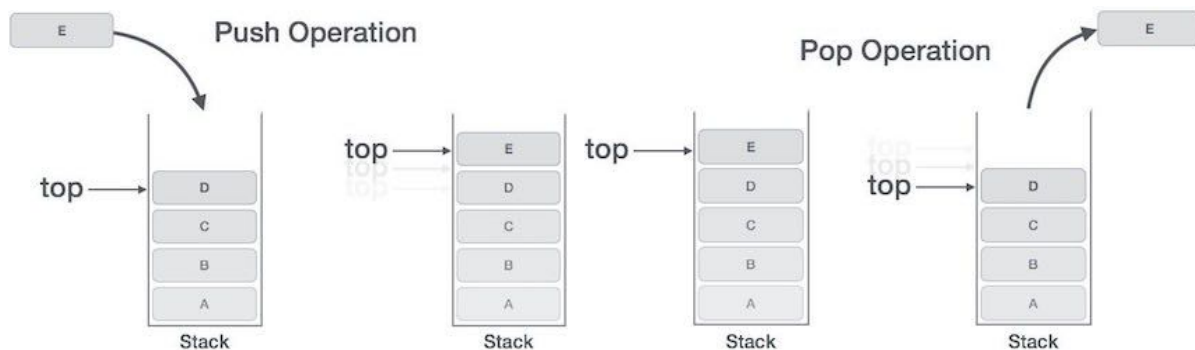


Figure 3a - Pushing data onto a stack

Figure 3b - Pushing data off of a stack

Register `%rsp`, also known as the Stack Pointer, is used on main memory by the Stack Pointer to keep a tab on where the next piece of data should be stored.

# LAFAYETTE COLLEGE

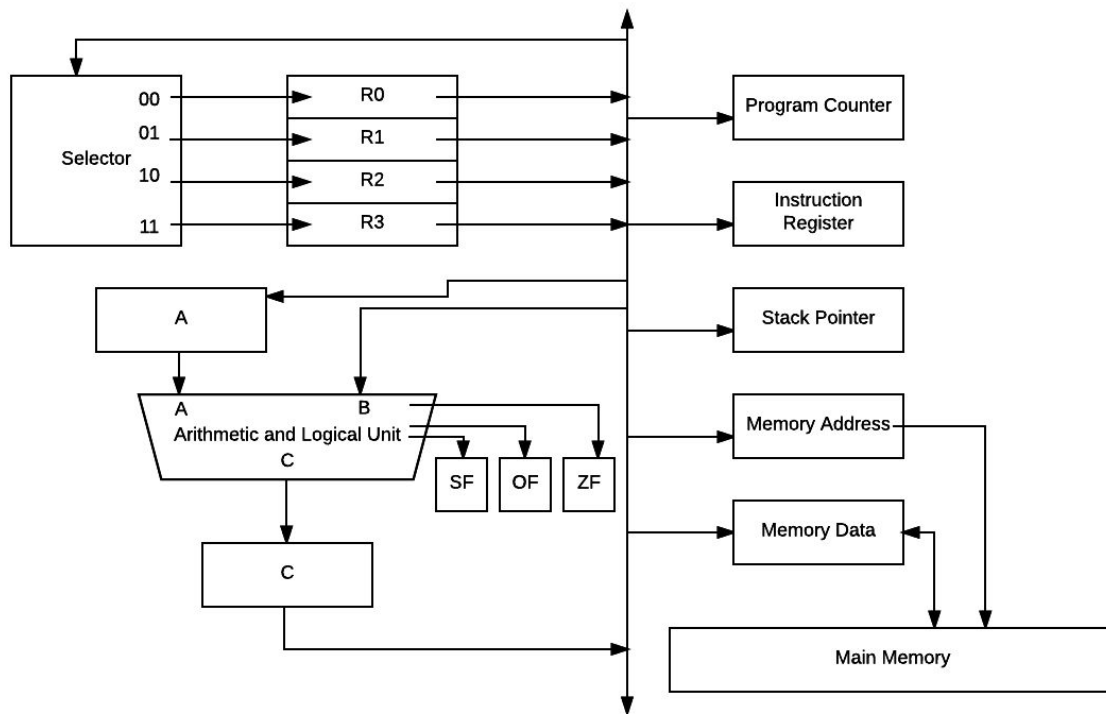


Figure 1b: Simplified overview of CPU

When a component in the CPU needs to access data located in main memory, there are several steps it performs. There must be an address that the component has which points to a location in memory, such as the Stack Pointer. The main data bus will know that the address it needs is located in the Stack Pointer, so it will fetch it from the Stack Pointer, shown in Figure 4a.

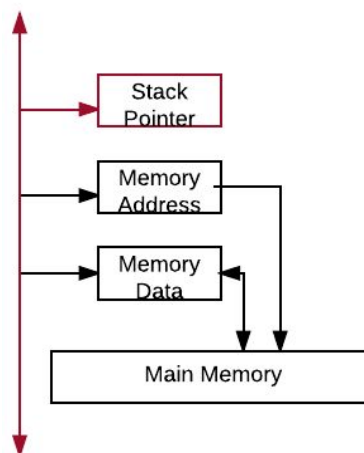


Figure 4a: Data bus fetches address from Stack Pointer

# LAFAYETTE COLLEGE

The Memory Address Register stores the address of data in Main Memory. Whenever something is to be fetched from Main Memory, the Memory Address Register will be sent the address of the data to be fetched. The main Data Bus will take the address is just fetched from the Stack Pointer and store it to the Memory Address Register, seen in Figure 4b.

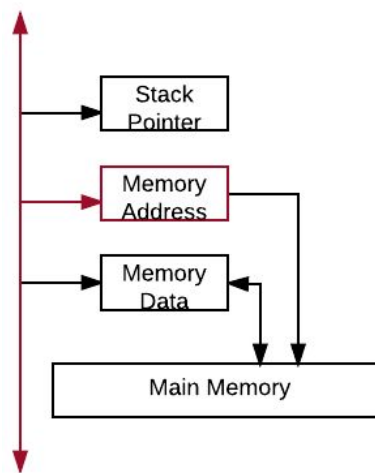


Figure 4b: Data bus stores address in Memory Address Register

Once the Memory Address Register has received a new address, it sends that address to the block of Main Memory on an individual bus, shown in Figure 4c. Main memory will fetch the data stored that address and send it to the Memory Data register via a separate data bus, shown in Figure 4d.

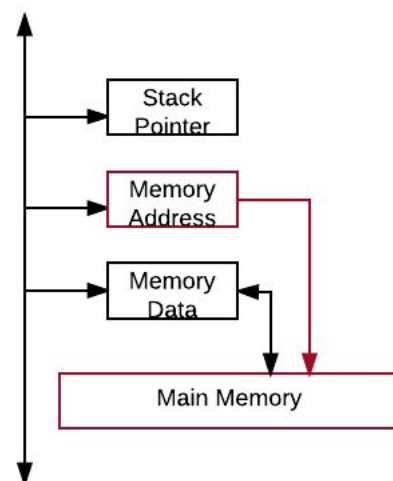


Figure 4C: Memory Address Register sends address to Main Memory

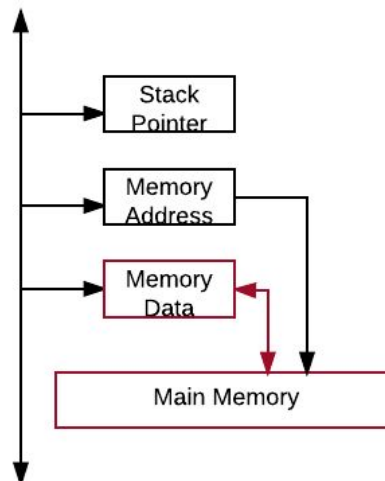


Figure 4D: Main Memory Fetches the data and stores it to Memory Data Register

# LAFAYETTE COLLEGE

Now, the data stored at the address specified by the Stack Pointer is stored in the Memory Data Register. Depending on what the instruction is, the CPU will decide what to do with the data. The data bus would fetch the data from the Memory Data Register and do what the CPU specifies.

The CPU executes a program using a cycle known as the Fetch Execute Cycle, shown in Figure 6. However, before the CPU can start processing the instructions, it must load them into main memory. Normally, it loads instructions from a computer program or an input device. For this project, the instructions are written as machine code in a file that is passed into the program and stored into main memory.

**Fetch** - A register known as the Program Counter stores the address of the current instruction being executed, it keeps a tab on which location in main memory the instruction is stored. For the CPU to process an instruction, it must fetch the address of the instruction from the Program Counter, shown in Figure 5a. Since the instruction is stored in main memory, the steps described in Figures 4a to 4d must be completed in order to fetch the instruction. After these steps have been completed, the instruction is stored in the Memory Data Register and the data bus fetches it from there (Figure 5b) and sends it to another register known as the Instruction Register, shown in Figure 5c.

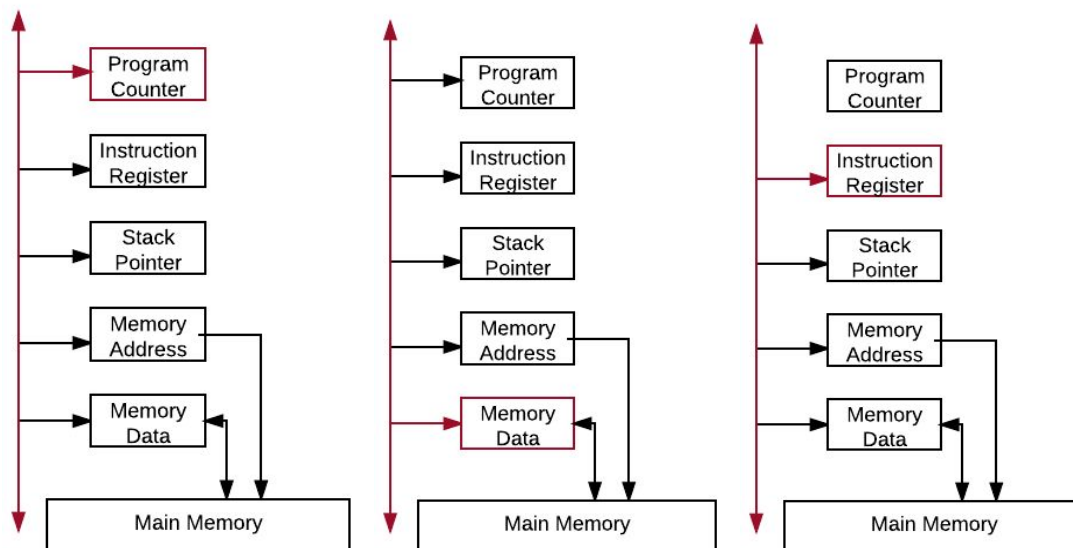


Figure 5a: Data bus fetches Address from Program Counter

Figure 5b: Data bus fetches instruction from Memory Data

Figure 5c: Data bus sends instruction to the Instruction Register



# LAFAYETTE COLLEGE

**Decode** - Now, the CPU must understand the instruction. The Instruction Register, which now stores the instruction, must decode it in order to properly dictate to the CPU what it must do. Since the instruction is represented in binary, it must be parsed and understood. In terms of the project, this involves knowing which binary encoding of the instruction set architecture relates to this instruction. Each instruction and each general purpose register has a unique binary encoding defined by files which are inputted to the project. Decoding the instruction means knowing which instruction it is, which parameters it has and what its steps are. Since this is project specific, it will be defined in a later section.

**Execute** - Having decoded the instruction, the CPU now knows how it should operate to carry out the instruction. The Control Unit sends signals in steps to each of the components which must work together to execute the instruction. Each instruction executes differently, this is shown in the Register Transfer Notation of each instruction. It is important to note that the data bus can only carry a certain amount of bits at a time, so only one step can be executed at a time.

**Store** - Usually, the output of an instruction has some sort of output. Whether it's writing to memory or storing data to a register, the CPU must write this output to the same main memory where the instructions are stored (just another section of it).

Once the instruction has finished executing, the Program Counter points to the next instruction in main memory and the same cycle repeats until the execution of the program. When the CPU finishes execution for the program, the Program Counter points back to where it was and all registers are cleared.

# LAFAYETTE COLLEGE

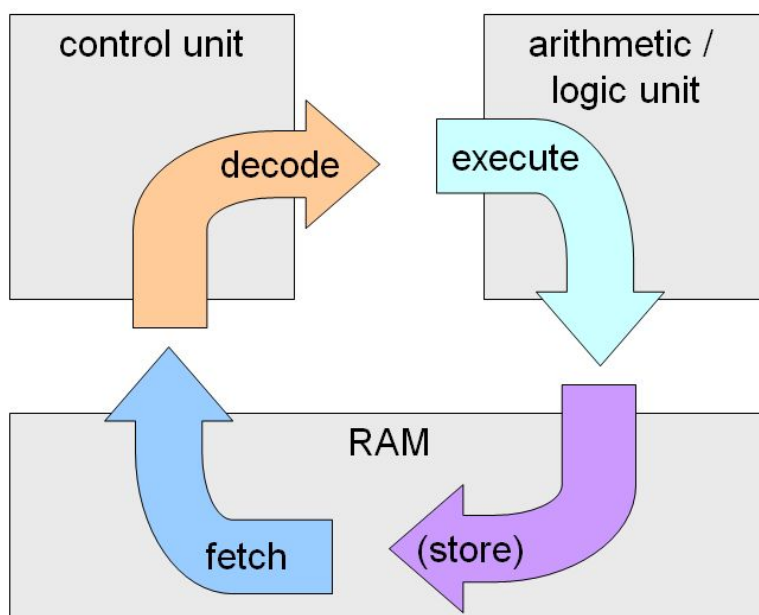


Figure 6 - Fetch Execute Cycle

An important component of the CPU is the clock which produces a signal on every rising clock edge which acts to give the components a green flag to operate their step within the instruction. The clock signal goes from 0 to 1 and repeats. When the clock is going from 0 to 1, known as the rising or positive clock edge, the next step of the instruction is fulfilled. Either the ALU completes an operation or the data bus sends information a component. The clock is not implemented in this project but it is an important detail to note about the CPU.

# LAFAYETTE COLLEGE

## Y86-64 Instruction Set Architecture

The Y86-64 Instruction Set Architecture has fewer addressing modes and a smaller set of operations. There are four move instructions, four integer operation instructions, seven jump instructions, six conditional move instructions and 4 main memory (stack) related instructions, shown in Figure 7.

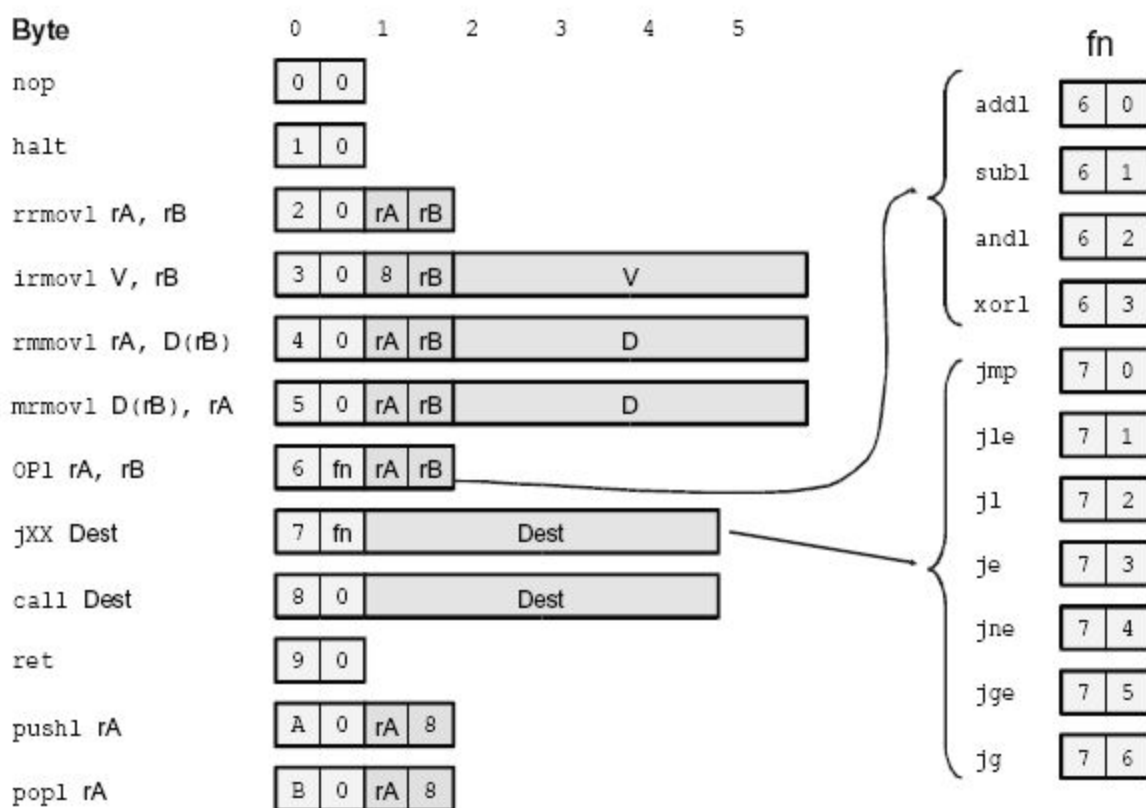


Figure 7: Y86 Instruction Set encodings

The four move instructions are rrmovq, irmovq, mrmovq, and rmmovq. The first two letters of these move instructions indicate the source and the destination, respectively. The source is either immediate(i), register(r), or memory(m) and this is designated by the first letter in the instruction name. The destination is either register(r) or memory(m) and this is designated by the second letter in the instruction name. The irmovq instruction moves an immediate value, represented with a \$ sign such as \$3 or \$0x02, into a register. The rrmovq instruction moves the value stored at the source register into the destination register. The mrmovq instruction moves data from main memory into a register. The rmmovq instruction moves data from a register into main memory. The 'q' at the end of the instruction indicates the word size of the data being moved.

# LAFAYETTE COLLEGE

The four integer operation instructions are `addq`, `subq`, `andq` and `xorq`. These instructions only operate on register data (compared to immediate values or main memory data). The `addq` instructions performs an addition on the values of two registers, the `subq` performs a subtraction on the values of two registers, the `andq` instruction performs a logical AND (& or ^) on the values of two registers and the `xorq` instruction performs a logical XOR ( $\oplus$ ) on the values of two registers. These instructions set the condition codes of the CPU. Y86-64 contains three single-bit condition codes: ZF, SF and OF, which store information about the effect of the most recent operation performed in the ALU. The ZF (zero flag) is 1 if the result of an operation is 0, and it is 0 otherwise. The SF (sign flag) is 1 if the result of an operation is negative, and it is 0 otherwise. The OF (overflow flag) is 1 if the result is too large to be stored into a register, and it is 0 otherwise.

The seven jump instructions are `jmp`, `jle`, `jl`, `je`, `jne`, `jge`, and `jg`. The jump instructions specify a destination (such as an address in memory) that the Program Counter should jump to and start executing the instruction there. The `jmp` instruction is a definite jump, meaning the Program Counter will just to the defined destination no matter what. The rest of the jump instructions have conditionals which are based off the logical conditions shown in Figure 8.

Instruction	Synonym	Jump condition	Description
<code>jmp Label</code>		1	Direct jump
<code>jmp *Operand</code>		1	Indirect jump
<code>je Label</code>	<code>jz</code>	ZF	Equal / zero
<code>jne Label</code>	<code>jnz</code>	$\sim$ ZF	Not equal / not zero
<code>js Label</code>		SF	Negative
<code>jns Label</code>		$\sim$ SF	Nonnegative
<code>jg Label</code>	<code>jnle</code>	$\sim$ (SF $\wedge$ OF) & $\sim$ ZF	Greater (signed >)
<code>jge Label</code>	<code>jnl</code>	$\sim$ (SF $\wedge$ OF)	Greater or equal (signed $\geq$ )
<code>jl Label</code>	<code>jnge</code>	SF $\wedge$ OF	Less (signed <)
<code>jle Label</code>	<code>jng</code>	(SF $\wedge$ OF)   ZF	Less or equal (signed $\leq$ )
<code>ja Label</code>	<code>jnbe</code>	$\sim$ CF & $\sim$ ZF	Above (unsigned >)
<code>jae Label</code>	<code>jnb</code>	$\sim$ CF	Above or equal (unsigned $\geq$ )
<code>jb Label</code>	<code>jnae</code>	CF	Below (unsigned <)
<code>jbe Label</code>	<code>jna</code>	CF   ZF	Below or equal (unsigned $\leq$ )

Figure 8 - Logical Jump Conditions

# LAFAYETTE

---

## COLLEGE

The six conditional move instructions are `cmovle`, `cmovl`, `cmovne`, `cmovge`, and `cmovg`. These move instructions have the same format as the register to register move instruction `rrmovq` but the destination is updated only if the condition codes satisfy the required constraints. These conditional moves follow the same logical conditions as the jumps (seen in Figure 8) but instead of jumping to a new section of main memory, they move values from one register to another. These are the only set of Y86 instructions that are not implemented in the simulation.

The 4 main memory related stack instructions include `push`, `pop`, `call`, and `ret`. The `push` and `pop` instructions pertain to the stack data structure that main memory is implemented with. The `push` instruction pushes data onto the stack and the `pop` instruction pops data off of the stack. The `call` instruction pushes the return address on the stack and jumps to the destination address. The `ret` instruction returns from said `call` instruction. Finally, the `halt` instruction stops the instruction execution of the program. This causes the entire system to suspend operation.

# LAFAYETTE COLLEGE

## Project Requirements

This entire program should be written in Java using object oriented design as the software approach. Each component of the CPU should be represented as an object with properties specific to that component's functionality. The program should have a graphical user interface in order to give the user a visual understanding of how the simulator works. It is important that each component of the simulator contains unit testing in order to fully demonstrate that the program works the way it is expected to. The program should accept several files in order to define CPU functionality. The configuration file defines CPU properties such as machine word size (32 or 64 bit machine), the bus size (8 or 16 bits) and functionality and size of the various components within the CPU. The machine code file will contain the instructions that the simulation is to run. This file will contain a hexadecimal address for each instruction to indicate where that instruction is to be stored in memory, followed by a 16 bit binary number which indicates the specific instructions and its parameters. The RTN (Register Transfer Notation) file will specify how each instruction in the Y86 Instruction Set Architecture works with the hardware and specific components of the CPU. The format of the RTN file consists of several lines per instruction where each line looks like : <destination><-<source>. So if the instruction includes fetching data from main memory using a specific memory address, the RTN will look like: M[MD]<-MA. This RTN is stored in the Control Unit of the CPU since it controls and steers the hardware in order to implement the functionality of each instruction. The main memory should be byte addressable and it should store the instructions passed into the program for simulation. Every Y86 instruction should have a fixed width for this project (even though in reality it has variable width encoding) and its own binary encoding so that the CPU can easily recognize the instruction and execute it.

## High Level Design Overview

# LAFAYETTE COLLEGE

The class diagram for my design is shown in Figure 9a. There are classes for every component of the CPU: Main Memory, Stack Pointer Register, Program Counter Register, Memory Data Register, Memory Address Register, Selector, Register, Instruction Register, ALU, Register C and Register A. The RTN Control class is a part of the Control Unit and its corresponding RTN classes are important to carry out instructions.

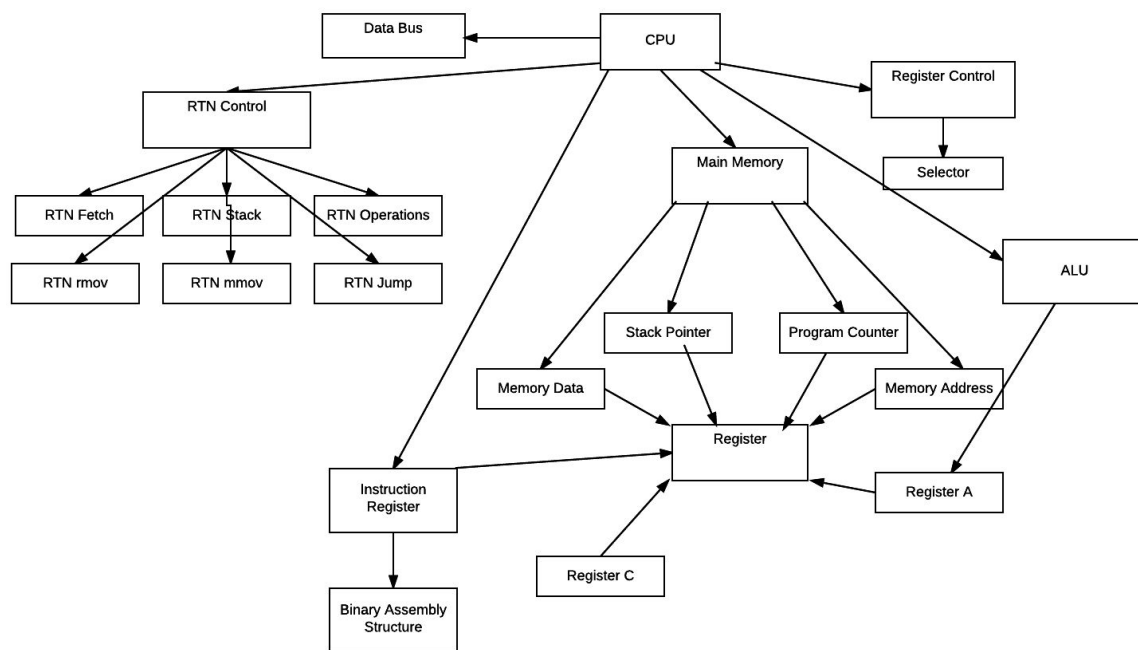


Figure 9a - High Level Class Diagram

The simulator runs as so: The Y86 binary Instruction Set file is loaded into the Binary Assembly Structure Class to be stored there in a dictionary like data structure. The machine code file is loaded into the Main Memory class to be stored. The configuration file is loaded into the Main class to give the simulation structure. The RTN file which specifies the RTN for each instruction is loaded into RTN Control class where subsequent RTN classes are created for each instruction. The instructions are grouped into sections which depend on similarity. All steps relating to fetching the instruction from main memory are stored in the RTN Fetch class, all steps relating to the push, pop, halt, ret and call instructions are stored in the RTN Stack class, all steps relating to the irmovq and rrmovq instructions are stored in the RTN RMOV class, all steps relating to the rmmovq and rmmovq instructions are stored in the RTN MMOV class and finally all steps relating to any of the jump instructions (conditional and not) are stored in the RTN Jump Class. The Register Control class loads in a file which specifies the number and name of all general



# LAFAYETTE COLLEGE

purpose registers to be created in the simulation. This class creates a dictionary for the general purposes registers and their binary equivalents which is used by the Selector class within the Register Control class to select a register specified by the Data Bus. Using the names from the register dictionary, the Register Control class creates a Register object for each name and stores them within the class. The Stack Pointer, Memory Data, Program Counter and Memory Address are all registers which have access to/revolve around Main Memory so they are all grouped under Main Memory. They are all still connected to the Data Bus through the CPU class and are operated in the CPU class. The ALU class is created along with its corresponding registers A and C which are used to store input and output data. Once all of the input files are read in and the class components of the CPU are set up, of these classes as well as the Data Bus class are passed through the CPU class where the simulation starts.

A flow chart of how the program works is shown in Figure 9b, this sort of diagram is known as a Process View and it shows the general steps of the simulation. First, all the files are loaded into the various classes that they will be used in. This happens simultaneously within the program and that is why that step is within two black horizontal lines.

Once the CPU class is created and all of its components are passed through, the simulation officially begins. The Program Counter sends its instruction address to the Memory Address Register and the Memory Address Register sends that address to Main Memory to be fetched from there. Once Main Memory retrieves that instruction, it sends it to the Memory Data Register to be stored there. The Data Bus fetches that address from the Memory Data Register and stores it in the Instruction Register to be decoded. After the Instruction Register has decoded the instruction, it is sent to the CPU to be run. The CPU uses the RTN Control class to determine the exact steps within each RTN for each instruction that must be used to execute the instruction. The execution takes place with the Register Control (moving data to Registers or retrieving data from Registers), the ALU (various operations done on values) and the Main Memory (data pulled from or stored in Main Memory).

This repeats until the Program Counter is no longer pointing to a space in memory that stores the instruction.



# LAFAYETTE COLLEGE

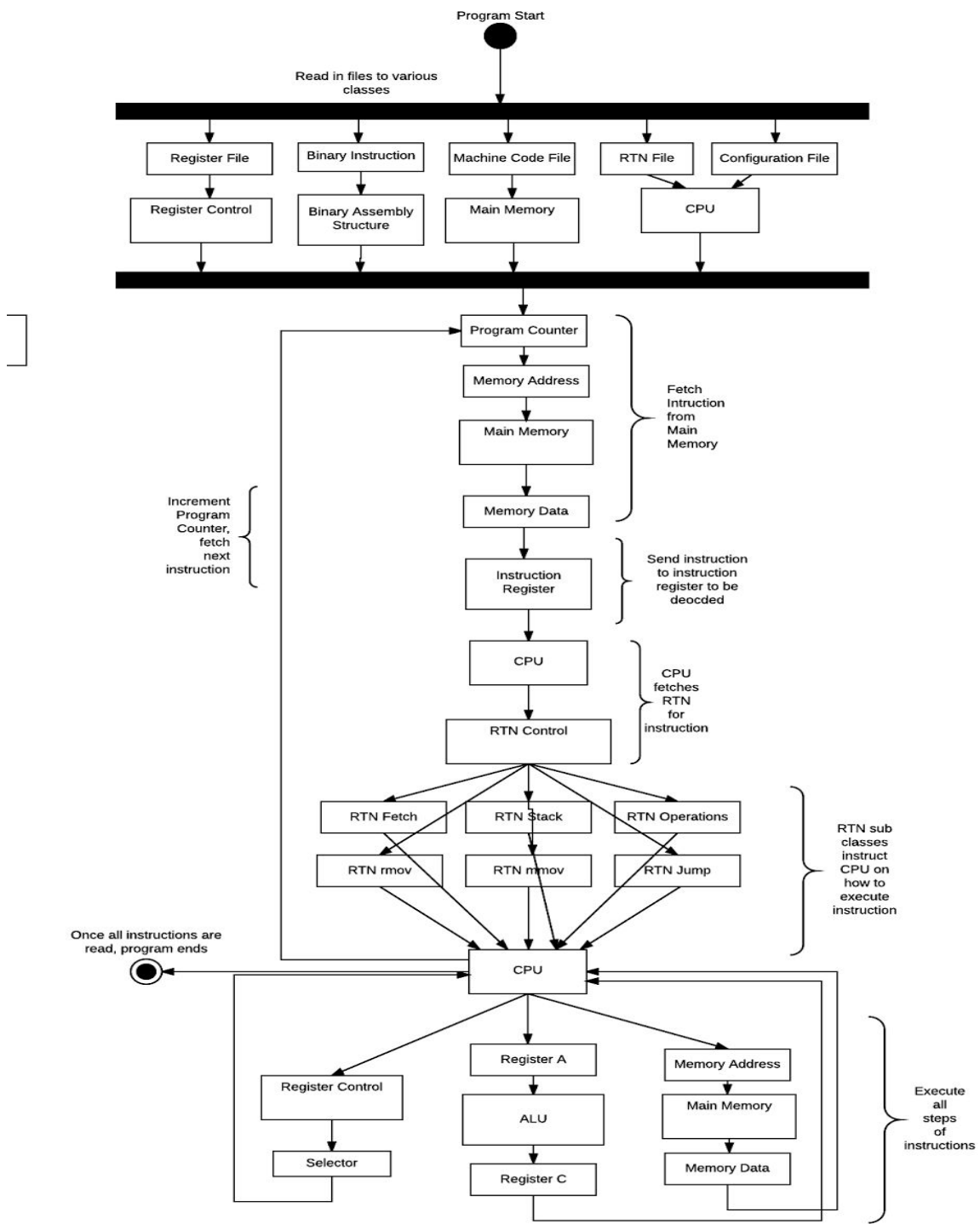


Figure 9b: Process View of Simulation

# LAFAYETTE COLLEGE

## Low Level Design

### Y86-64 Instruction Format

Instructions in a computer are represented in binary and each processor has different binary encodings. A unique binary instruction set was written for this project. The instructions were formatted using fixed-width format meaning all instructions are 16 bits long. Instructions that do not fill up the entire space get padded with 0s as a proper alignment. Figure 10 shows the general format of all the instructions. The first 8 bits indicate which instruction it is and the last 8 bits represent the source and destination of the instruction. Not all instructions have a source and destination as their parameters but since most of them do, it was a necessary add.

#### General Form:

4 bits	4 bits	4 bits	4 bits
Instruction	Function	Source	Destination

Figure 10: Instruction General Format

The four general purpose registers are Register 0 (R0, %r0), Register 1 (R1, %r1), Register 2(R2, %r2) and Register 3(R3, %r3). The binary format of these registers are shown in Figure 11, where the binary number matches the decimal number of the register. Since registers are often sources and destinations, their 4 bit binary numbers are used to fill those parameter bits.

R0	0000
R1	0001
R2	0010
R3	0011

Figure 11: Binary Encoding of general purpose registers

# LAFAYETTE COLLEGE

The move instructions (irmovq, rrmovq, rmmovq, mrmovq) shown in Figure 12 which presents the equivalent decimal and binary encodings:

<i>moves</i>									
rrmov RA, RB	2	0	RA	RB	0010	0000	RA	RB	
rmmov RA, (RB)	3	0	RA	RB	0011	0000	RA	(RB)	
mrmov (RA), RB	4	0	RA	RB	0100	0000	(RA)	RB	
irmov RA, RB	5	0	RA	RB	0101	0000	1xxx	RB	

Figure 12: Instruction Format of the Moves

Each move instruction is given a unique 4 bit instruction number starting at 2(0010) until 5(0101). Since the 4 bit instruction numbers are unique, there is no need for making the 4 bit function number unique. The rrmovq instruction is given the 8 bit format of 00100000 and its source and destination will be two registers with their own 4 bit format shown in Figure 11. Say the instruction is rrmov R0, R1: the full 16 bit instruction will be 00100000|0000|0001 (stored without the |). The rmmovq instruction is given the 8 bit format of 00110000 and its source is a register with a 4 bit format. Its destination is a memory address denoted as (RB) where the () indicate that the value is to be stored in main memory. The mrmovq instruction is given the 8 bit format of 01000000 and its destination is a register with a 4 bit format. Its source is a memory address denoted as (RA) where the () indicate that the value must be pulled from main memory. The main memory address will never be greater than decimal 15 (binary 1111) and will therefore never need more than 4 bits to be represented. The irmovq instruction is given the 8 bit format of 01010000 where the destination is a register with a 4 bit format. Its source is an immediate value which is indicated by its first character being a 1. The registers have binary values of (R0) 0000, (R1) 0001, (R2) 00010, and (R3) 0011. An immediate value must start with a 1 but the remaining 3 bits can be any single digit decimal number. A value of 1002 will be read as an immediate value of 2 and a value of 1021 will be read as an immediate value of 21. This way, the only numerical limitation an immediate has is that it can be no bigger than 999.

# LAFAYETTE COLLEGE

The operation instructions (addq, subq, andq, xorq) are shown in Figure 13 which presents the equivalent decimal and binary encodings:

add RA, RB	6	0	RA	RB	0110	0000	RA	RB
sub RA, RB	6	1	RA	RB	0110	0001	RA	RB
and RA, RB	6	2	RA	RB	0110	0010	RA	RB
xor RA, RB	6	3	RA	RB	0110	0011	RA	RB

Figure 13: Instruction Format of the Operations

Each operation is given the same 4 bit instruction number of 6(0110). The 4 bit function numbers are unique and they go from 0(0000) to 4(1000). The addq instruction is given the 8 bit format of 01100000 and its source and destination parameters are 2 registers with their own 4 bit format shown in Figure 11. Say the instruction is addq R1, R2: the full 16 bit binary encoding of the instruction will be 01100000|0001|0010 (the | are there to split up the bits). The subq instruction is given the 8 bit format of 01100001 and its source and destination parameters are 2 registers with their own 4 bit format. The andq instruction is given the 8 bit format of 01100010 and its source and destination parameters are 2 registers with their own 4 bit format. The xorq instruction is given the 8 bit format of 01100011 and its source and destination parameters are 2 registers with their own 4 bit format.

# LAFAYETTE COLLEGE

Finally, the jump operations are shown in Figure 14, which presents the equivalent decimal and binary encodings.

## Jumps

jmp address	7	0	RA	RB	0111	0000	1/2 addr	1/2 addr
jmp address	7	1	RA	RB	0111	0001	1/2 addr	1/2 addr
jmp address	7	2	RA	RB	0111	0010	1/2 addr	1/2 addr
jmp address	7	3	RA	RB	0111	0011	1/2 addr	1/2 addr
jmp address	7	4	RA	RB	0111	0100	1/2 addr	1/2 addr
jmp address	7	5	RA	RB	0111	0101	1/2 addr	1/2 addr
jmp address	7	6	RA	RB	0111	0110	1/2 addr	1/2 addr

Figure 14:

The stack instructions (push, pop, call, halt, ret) are shown in Figure 14 which presents the equivalent decimal and binary encodings.

push rA	A	0	RA	0	1010	0000	RA	0000
pop rA	B	0	RA	0	1011	0000	RA	0000
call address	8	0	0	0	1000	0000	1/2 addr	1/2 addr
halt	0	0	0	0	0000	0000	0000	0000
ret	9	0	0	0	100	0000	0000	0000

Figure 14: Instruction Format of the Stack Operations

# LAFAYETTE COLLEGE

Each stack instruction is given a unique 4 bit instruction number so there is no need for making the 4 bit function number unique. The pushq instruction is given the 8 bit format of 10100000, its source parameter is a register value that is to be pushed to the top of the stack. Since the one register is the only parameter, the remaining 4 bits in the destination parameter are padded with 0s. Say the instruction is pushq R3: the full 16 bit binary encoding of the instruction will be 10100000|0011|0000 (the | are there to split up the bits). The popq instruction is given the 8 bit format of 1011, its source parameter is a register that is going to store the value that will be popped off the top of the stack. Since the one register is the only parameter, the remaining 4 bits in the destination parameter are padded with 0s. The call instruction is given the 8 bit format of 10000000, its source and destination parameters are halves of the address which store the instruction the program is calling. Say the instruction is call 0x12: the full 16 bit binary encoding of the instruction will be 10000000|0001|0010. The halt instruction takes no parameters and therefore its entire 16 bit binary format is 0000000000000000. The ret instruction also takes no instructions so its entire 16 bit binary format is 1000000000000000.

## Input Files

The simulator must read in several files in order to run. Computer have a great deal of information already hardcoded within its processor and the input files is the simulators way of doing the same. As explained in the Project Requirements section, there is a Machine Code file, a RTN file, a Register file and a Binary Instruction Set file.

The Machine Code file is the real input into the program, the program is actually going to run what it in this file. This file, shown in Figure 15, gives the program a hexadecimal address and a corresponding 16 bit instruction. The instruction includes the 8 bits which identifies the instruction and the 8 bits which identifies the instruction parameters. This binary instruction is to be loaded into the Main Memory address indicated on the same line. As previously stated, Main Memory stores both program instructions and program data. The Main Memory class loads in this file upon its creation; the exact format of how the program implements Main Memory will be explained in greater detail later but right now it is important to note that the Machine Code Input file specifies the instructions to be executed within the program and the addresses in Main Memory where each instruction is to be stored.

# LAFAYETTE

## COLLEGE

```

0x00 0101000010020000
0x02 0100000010100001
0x04 0110000000000001
0x06 0111000000001000
0x08 0011000000011011

```

Figure 15: Machine Code Input File

The Binary Instruction Set file loads in the 8 bit binary numbers that act as identifiers of the Y86 Instruction set, this was explained in the previous section. The full file is shown in Figure 16. Each instruction is given a unique 8 bit binary number. The 4 registers are also included in the file as well as the format for loading in an immediate value. This file is loaded into the Binary Assembly Structure class, where it is parsed and stored into a Hashmap<String, String>. The key is the assembly instruction and the corresponding values are their 8 bit binary equivalents. This HashMap with the Binary Assembly Structure class will be used by the Instruction Register class in order to decode the binary instruction.

```

add 01100000
sub 01100001
and 01100010
xor 01100011
rrmov 00100000
rmmov 00110000
rrmmov 01000000
irmov 01010000
jmp 01110000
jle 01110001
jl 01110010
je 01110011
jne 01110100
jge 01110101
jg 01110110
push 10100000
pop 10110000
call 10000000
halt 00000000
R0 0000
R1 0001
R2 0010
R3 0011
Im 1xxx

```

Figure 16: Binary Instruction Set Input File

# LAFAYETTE COLLEGE

The Register File is a small file which lists the names and binary equivalents of the general purpose registers, this file is shown in Figure 17a. This file is loaded into the Register Control class and stored in a `HashMap<String, String>` where the key is the name, such as R0, and the value is the binary equivalent, which would be 0000. For every general purpose register listed in the file, the Register Control class creates Register objects. These Registers are stored in a Register array, Figure 17b, in the Register Control class. This Register array is often used by the CPU when executing instructions which include storing data into or pulling data from Registers but cannot be directly accessed by the Data Bus because the signal must first go through the Selector.

R0 0000
R1 0001
R2 0010
R3 0011

Figure 17a: Binary Register  
Set Input File

R0	R1	R2	R3
----	----	----	----

Figure 17b: Register Object Array

The RTN(Register Transfer Notation) File describes what the CPU does with each of its components in order to execute an instruction. This file is seen in Figure 18. Each Y86 instruction has its own RTN. The parts within the RTN are abbreviated, in the “fetch” RTN: c is Register C, pc is Program Counter, ma is Memory Address Register, md is Memory Data Register, ir is Instruction Register.

This file is loaded into the Instruction Register class because as the Instruction Register decodes the instruction from binary to readable English, it must also find its corresponding RTN for the CPU to be able to know how to execute the instruction. It loads the data into a `HashMap<String, LinkedList<String>>` where the key is the instruction, such as fetch, and the `LinkedList` stored each step of the RTN (separated by “ ”) into individual indexes.



# LAFAYETTE COLLEGE

```

fetch c<-pc+2 ma<-pc pc<-c md<-m[ma] ir<-md
irmov r[ra]<-#
rrmov r[rb]<-r[ra]
mrmov ma<-r[ra] md<-m[ma] r[rb]<-md
rmmov ma<-r[rb] md<-m[ma] md<-r[ra]
add a<-r[ra] b<-r[rb] c<-a+b r[rb]<-c
sub a<-r[ra] b<-r[rb] c<-a-b r[rb]<-c
xor a<-r[ra] b<-r[rb] c<-a^b r[rb]<-c
and a<-r[ra] b<-r[rb] c<-a&b r[rb]<-c
jmp pc<-m[ma]
jle pc<-m[ma]
jl pc<-m[ma]
je pc<-m[ma]
jne pc<-m[ma]
jge pc<-m[ma]
jg pc<-m[ma]
pop ma<-rsp md<-m[ma] r[ra]<-md c<-rsp+8 rsp<-c
push ma<-rsp c<-rsp-8 md<-m[ma] rsp<-c
call c<-rsp+8 ma<-c rsp<-c md<-pc m[ma]<-md ma<-value md<-m[ma] pc<-md
halt
ret c<-rsp+8 ma<-c rsp<-c md<-m[ma] pc<-md

```

Figure 18: Register Transfer Notation Input File

# LAFAYETTE COLLEGE

## Main Memory

The Main Memory class contains a `String[]` array which represents the space which will store program instructions and program data. The Main Memory Array, shown in Figure 19, is of length 21. Since memory is byte addressable (8 bits should be accessed at a time), each index of the array is a byte.

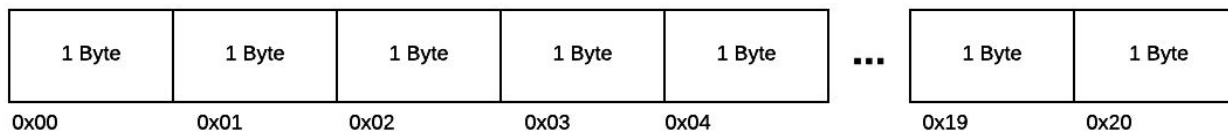


Figure 19: Main Memory String[] Array

One of the first things that the Main Memory class does is load in the Machine Code input file. The Main Memory is split up so that the first 10 bytes (indexes 0 to 9 in the array) store the program instructions and the remaining 10 bytes (indexes 10 to 20) store the program data. This way, Main Memory stores what it needs to but in an organized way. The indexes match up with the hexadecimal address that would be implemented in Main Memory so that index 0 is address 0x00, index 1 is address 0x01 and so on. This is the reason why the addresses in the Machine Code file start at 0x00 and will never be greater than 0x09. This implementation is seen in Figure 20.

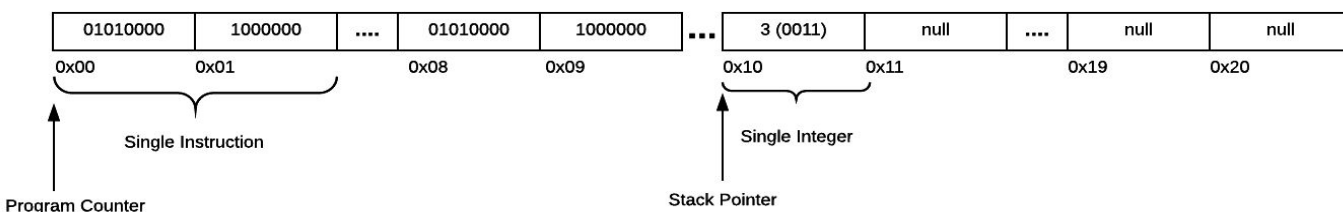


Figure 20: Main Memory Implementation

As Figure 20 shows, the first 10 indexes are for the instructions and the remaining 10 are for the program data. The instructions are split up into two bytes of memory, so that the first byte contains the instruction identifier and the second byte contains the instruction parameters. The Program Counter class stores the address in memory that go from 0x00 to 0x09. The last 10 indexes store program data and it is implemented as a stack. The Stack Pointer class stores the address of the most recently added data element. Each address (index) stores a single piece of data. This data is not spread over several bytes of memory like the instructions are.

# LAFAYETTE COLLEGE

The Main Memory class contains the Program Counter class, the Stack Pointer class, the Memory Address class and the Memory Data class as subclass. All of these subclasses extend the Register class. The reason for these being subclass is that the Program Counter class and Stack Pointer class are accessed only by the main Data Bus class for the addresses they store in order to fetch information from the Main Memory class. The Memory Address class and Memory Data class are subclasses because they have direct data buses that connect to the Main Memory class. Their purpose is solely for the access and fetching of data and instructions stored in Main Memory. These subclasses are separate components of the CPU but because their purposes are all Main Memory related, it was more organized to store them all together.

The Main Memory class is able to receive an address sent by the Memory Address Register class on its own separate Data Bus, fetch the instruction or data stored at that address and store that value inside the Memory Data Register class. This process is seen in Figure 21. In the first step, the Memory Address class contains an address of 0x09 and the Memory Data class contains no value. When the main Data Bus sends a signal (basically instructing) for the Memory Address class to send its value to the Main Memory class through a separate Data Bus, it does so in the second step. Since data and instructions from Main Memory are often fetched, it makes sense for there to be an individual Data Bus for the process. Here, the Main Memory class identifies the data for that sent address address. In the third step, the Main Memory class extracts the data from the array at that location and send it to the Memory Data class, again on its own Data Bus, to be stored there. Depending on the instruction, the Data Bus will fetch the data from the Memory Data class accordingly.

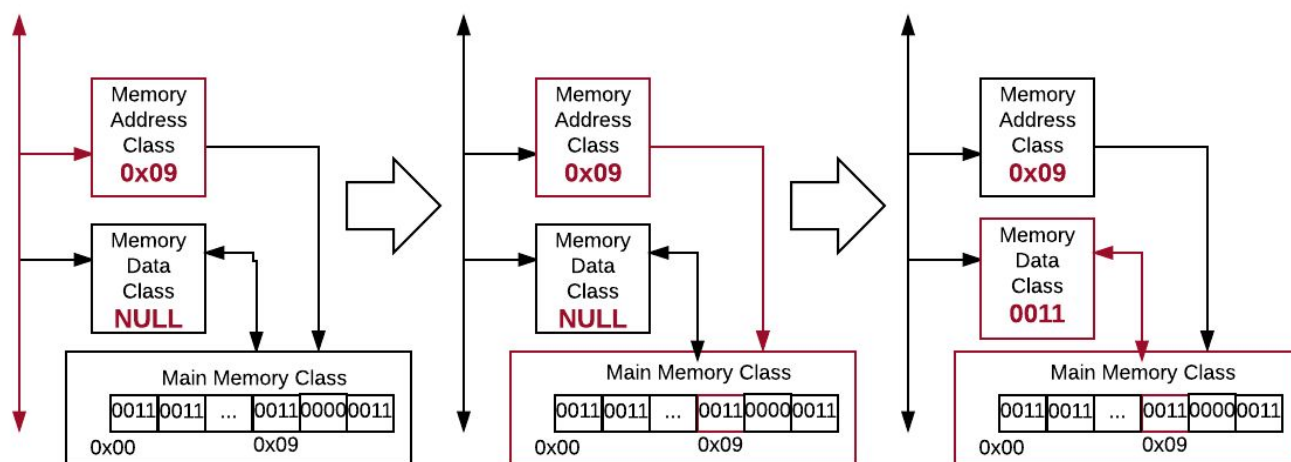


Figure 21: Main Memory class receives address from MA class, fetches the data stored at that address in Main Memory array and sends that value to MD class to be stored.

# LAFAYETTE COLLEGE

The Program Counter cannot access Main Memory directly. In order for an instruction to be fetched, the CPU must follow the steps highlighted in Figure 22a-22d. Figure 22a shows how the main Data Bus fetches the address of the instruction stored in the Program Counter class which at the start of the program is always going to be address 0x00. When the Data Bus fetches an address or piece of data to bring from one component to the other, the Data Bus class temporarily stores the address and marks itself as unavailable so that no other CPU component can attempt to also send data to the Data Bus. The Data Bus class knows which instruction is being executed (done so through various methods within the class). Once the address or piece of data is delivered to its correct destination component, the Data Bus clears and is marked as available.

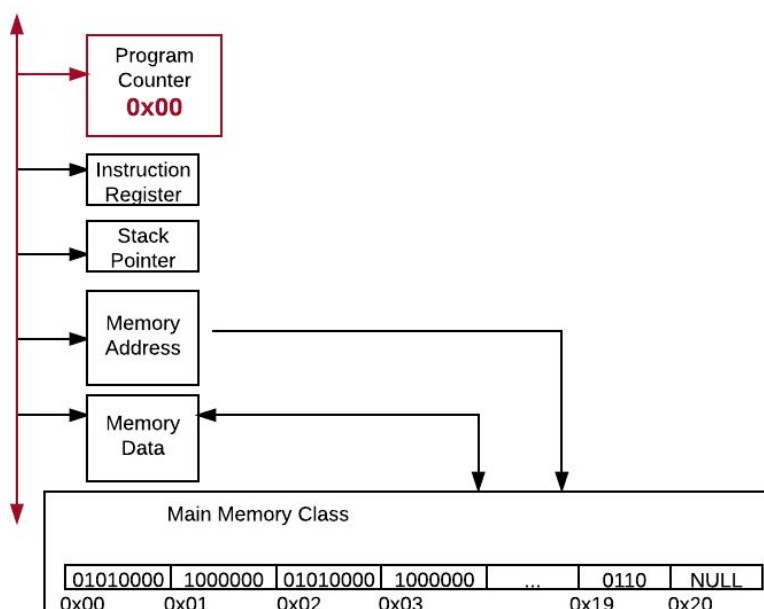


Figure 22a: Data Bus class fetches address of current instruction from the Program Counter class

The address of the instruction fetched from the Program Counter class is sent to the Memory Address class to be stored there, seen in Figure 22b. The Memory Address class receives the signal to fetch data and the address from the Data Bus class and knows to send a signal and the address to Main Memory class. Every instruction or piece of data that wants to be fetched from Main Memory must do so through the Memory Address class. This class is similar to a gatekeeper which only has access to the castle that is Main Memory.

# LAFAYETTE COLLEGE

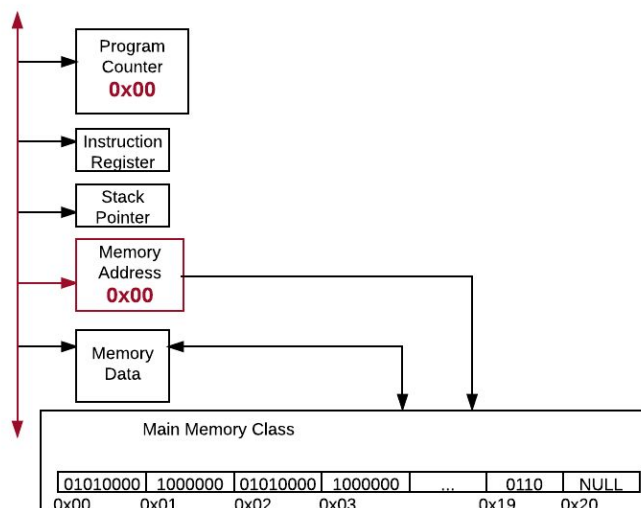


Figure 22b: Instruction stored at Memory Address Register.

Next, the Memory Address class sends the address to the Main Memory class so that the instruction can be fetched, seen in Figure 22c. The Main Memory class knows that when an address from 0x00 to 0x09 is being fetched from its array, the data to be returned is an instruction. Since instructions are stored in 2 bytes of memory, Main Memory class fetches the first half of the instruction which is stored at that address (0x00) and then the second half of the instruction which is stored at that address plus 1(0x01). The Main Memory class combines the two 8 bit pieces of an instruction before sending it to the Memory Data class.

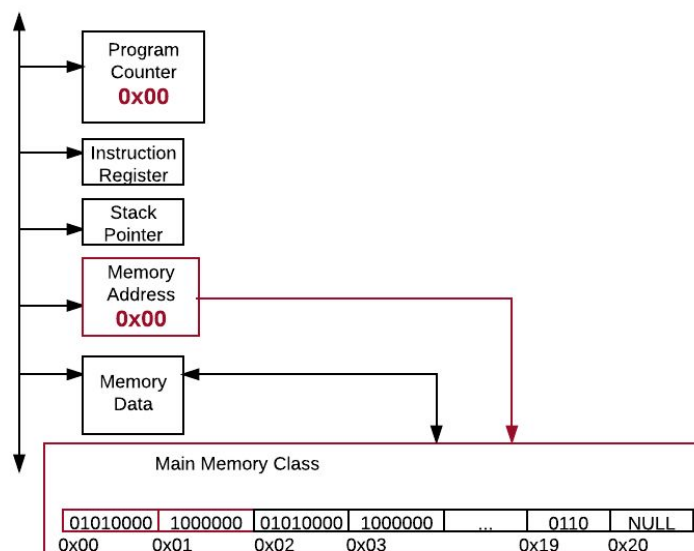


Figure 22c: The Memory Address class sends the instruction address to the Main Memory class to be fetched.

# LAFAYETTE COLLEGE

The Main Memory class combines the two bytes of instruction into one, sends that data to the Memory Data and the main Data Bus sends that instruction to the Instruction Register to be decoded before executing the next instruction, seen in Figure 22d. Once the Memory Data class receives a new instruction or piece of data from the Main Memory class, it signals the Data Bus class. The Data Bus class, knowing what to do with the instruction/data, fetches it from the Memory Data class. Since an instruction was being fetched, the Data Bus class temporarily stores the instruction (global variable), marks itself as unavailable(boolean) and sends it to the Instruction Register to be stored(methods).

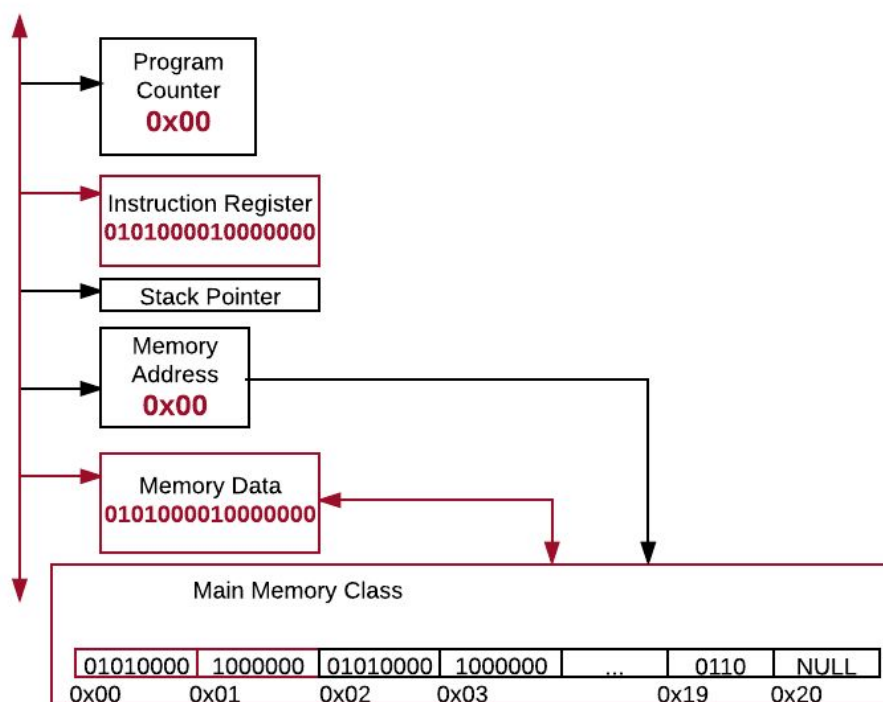


Figure 22d: The Main Memory class sends the instruction to the Memory Data class and the main Data bus fetches the instruction from the Memory Data class and stores it into the Instruction Register class

The Stack Pointer also does not have access to the Main Memory class. It goes through the same steps as the Program Counter. The only difference is that the Main Memory class knows that if the address is between 0x10 to 0x20, then the data is in the stack portion of the Main Memory array. The Main Memory class will then only fetch the data at that single address, send it to the Memory Data class to be stored and then depending on the instruction step it is executing, the main Data Bus will fetch the data from the Memory Data class and store it accordingly.

# LAFAYETTE COLLEGE

## Instruction Register

The Instruction Register class uses the HashMap stored in the Binary Assembly Structure class to decode instructions. Upon an instruction fetch, the newly retrieved instruction goes from Main Memory into the Instruction Register to be decoded. Since computers work in binary, decoding means comparing the sequence of bits to determine which instruction it matches up with. The computer will keep this instruction in binary but for simulation purposes, the Instruction Register class decodes it from binary to Assembly Language. The process that the Instruction Register class takes to decode the instruction is shown in Figure 23. Upon receiving the full 16 bit instruction from the Data Bus class (indirectly sent from the Memory Data class), the Instruction Register class splits it up. As described in the Y86 Instruction Set section, the first 8 bits of the instruction determine which Y86 Assembly Instruction it is. The Instruction Register class takes those first 8 bits and looks through the HashMap from the Binary Assembly Structure class. (As a reminder, this HashMap contains Y86 Assembly Instructions are keys and their 8 bit binary equivalent as values.) The Binary Assembly Structure class is being passed into the Instruction Register class for easy reference to the HashMap. Depending on which instruction it is, it decodes the last 8 bits accordingly. Since the instruction being decoded in Figure 23 is add, the Instruction Register knows that the last 8 bits are two 4 bit registers. The Instruction Register class splits up these 8 bits into two 4 bits and looks for the English name of the registers. It does so by also looking through the HashMap of the Binary Assembly Structure class, since this data structure stored both instructions and possible general purpose register parameters. Once these registers are identified as R0 and R1, the Instruction Register completes the Assembly form of “add R0, R1”. This instruction can be read as ‘take the value from R0, add it to the value from R1 and store the result in R1’. The Instruction Register class sends this instruction to the CPU class through the Data Bus class. The Data Bus class still knows that the instruction being executed is a fetch so it knows that once the Instruction Register has decoded the instruction, it must send that to the CPU “motherboard” to be processed. This signals the end of the fetch instruction and the beginning of the execution of this new instruction. The CPU class knows the Register Transfer Notation for all of the instructions by reading in the RTN file, creating a HashMap which stores the instruction name as the key and a Linked List (where each index of the list is a step of the instruction) of that instructions RTN as the value. Now the instruction being executed is add R0, R1 and the CPU uses the RTN steps to dictate the Data Bus class on how to operate.

# LAFAYETTE COLLEGE

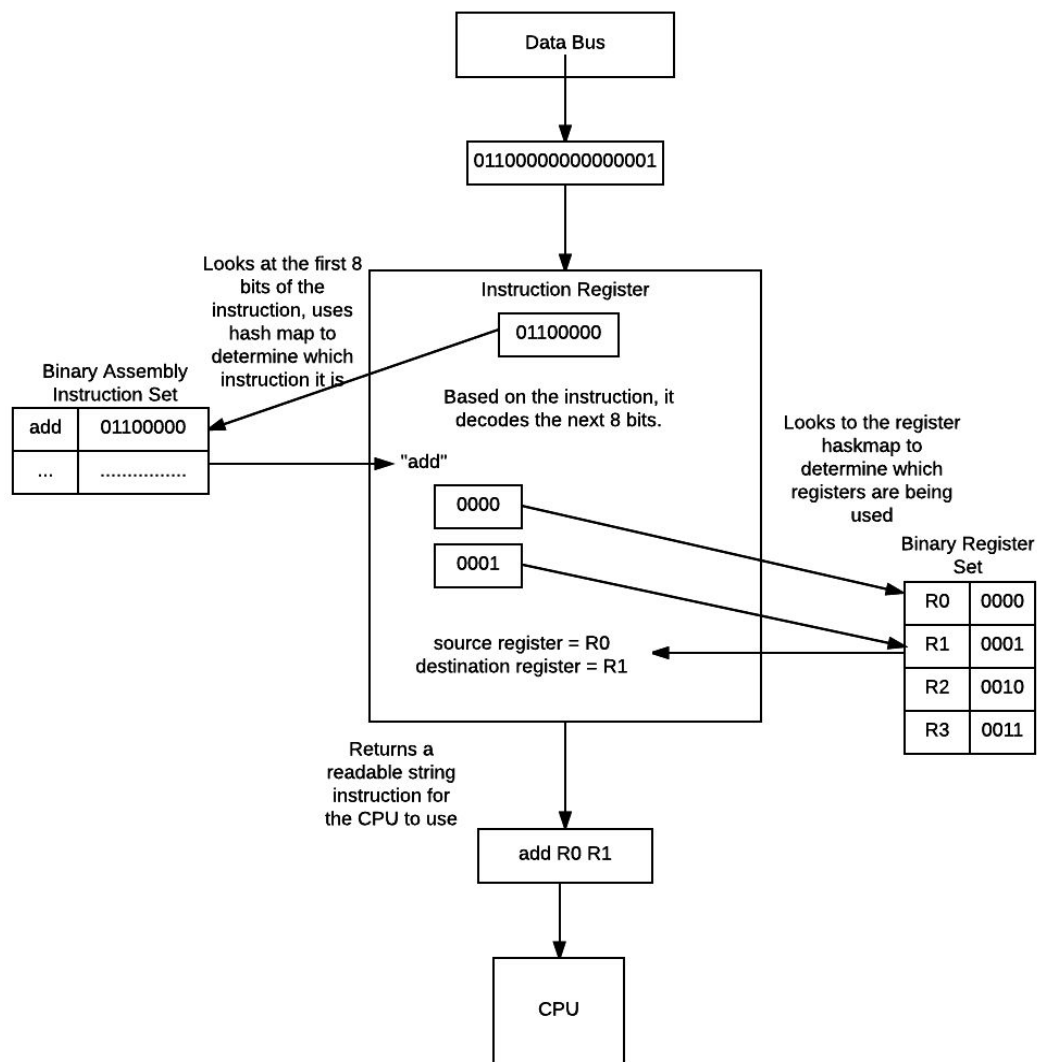


Figure 23: Process of decoding a binary instruction within Instruction Register class



# LAFAYETTE COLLEGE

## RTN Control

The CPU class uses the RTN Control class to execute the instruction. The RTN Control class receives an instruction and links the CPU class to the proper RTN subclass. This is seen in Figure 24. The class receives an instruction from the CPU class, identifies it and sends the relating subclass back out to the CPU class. This RTN subclass, RTN Operations class, knows how to process the add instruction. These subclasses were created in order to avoid packing everything into the CPU class. This subclass takes the RTN for the instruction and processes it step by step. Since the subclass cannot directly do anything within the CPU, it controls the Data Bus class to do through various methods. The subclass knows about all of the CPU components but does not alter them. The CPU class uses the RTN Control class and its various sub classes as the map to guiding the Data Bus class.

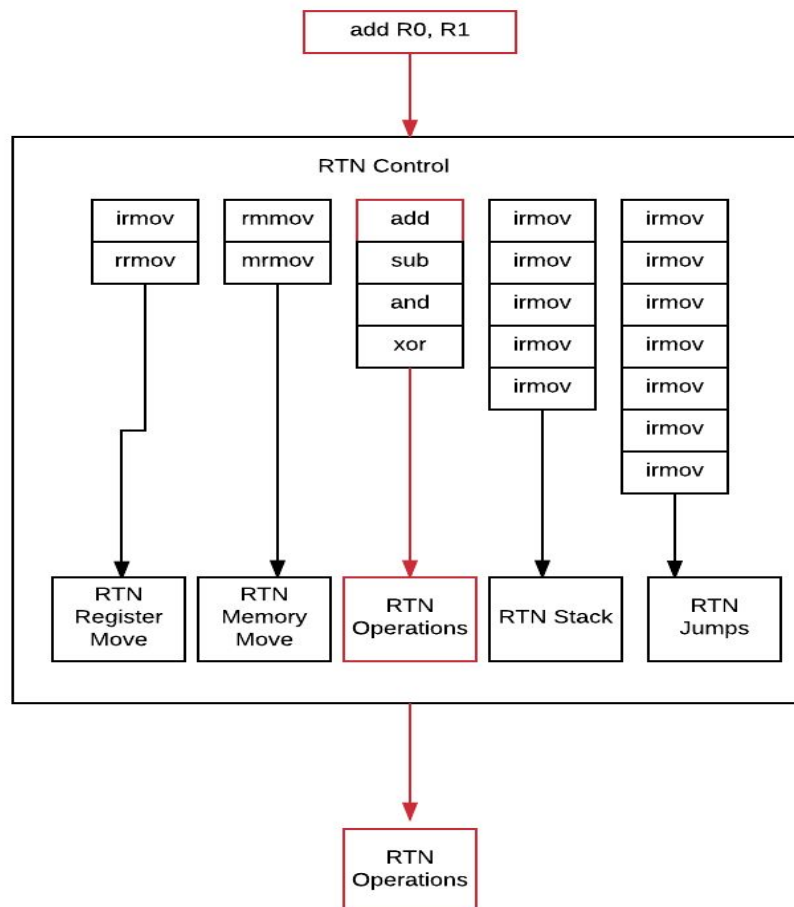


Figure 24: RTN Control class

# LAFAYETTE COLLEGE

## Register Control

The Register Control class manages the access to 4 general purposes registers in the CPU. The Register Control class contains the Selector component of the CPU as well as the array of Register object created in this class. As previously mentioned, this class loads in a file of general purpose Registers, stores it to a HashMap within the Register Control class where the key is the English name for the register and the value is its binary equivalent, and creates a Register object for each general purpose Register. The overview of this class is seen in Figure 25b. The Selector class is a subclass of the Register Control class, it is the only component of the CPU which has access to the Register array.

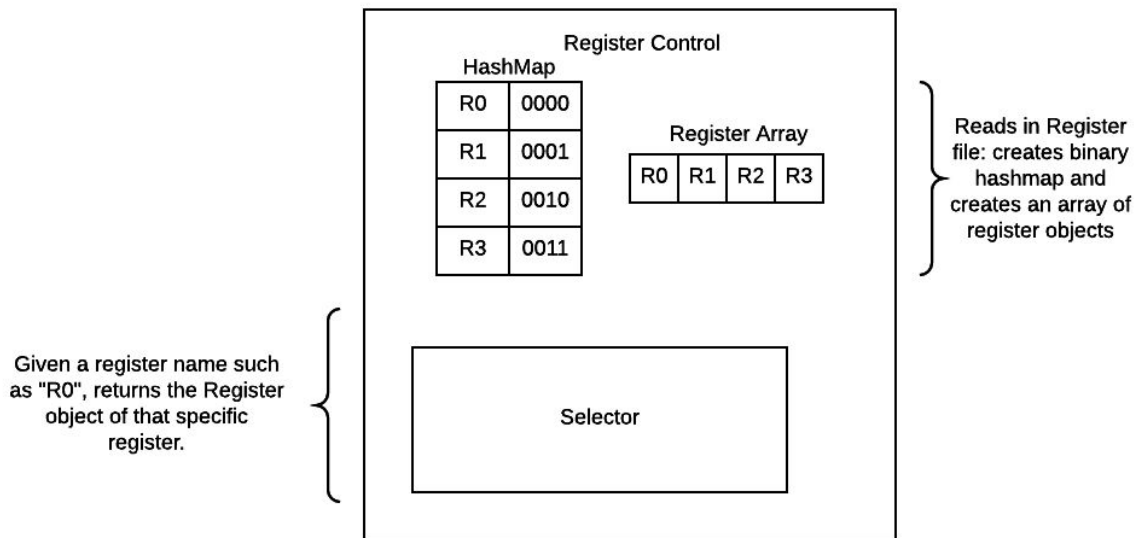


Figure 25: Overview of Register Control class

# LAFAYETTE COLLEGE

This relates to the top left portion of the CPU, shown in Figure 26. If the Data Bus class needs to fetch a value from a Register or store data to a register, it does so through the Selector class. The Selector class receives a signal and an input from the Data Bus class. The signal indicates that something is needed from one of the Registers and the input is the binary name of one of the registers.

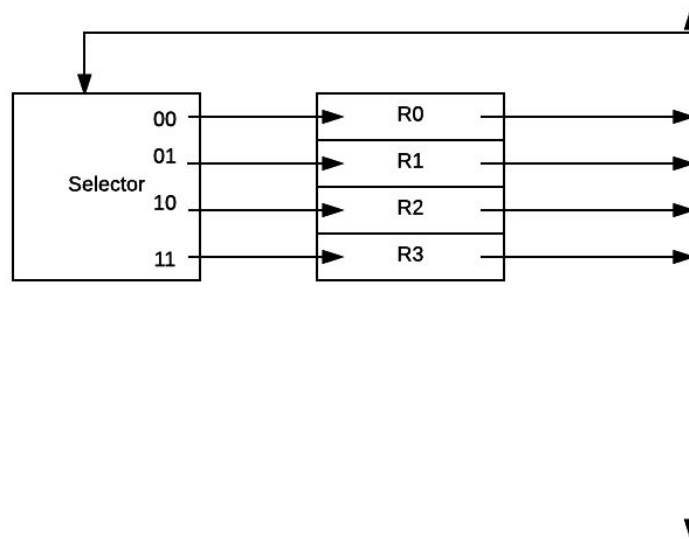


Figure 26: Register Control class within CPU

The process of selecting something from a register is seen in Figure 27. The register to be selected has a binary equivalent of 0000 which is sent to the Selector class. The Selector class uses the HashMap loaded in from the Registers file to determine which Register object that binary name relates to. Once a Register object is selected, the Selector class uses the signal sent from the Data Bus class to either update a value or send out a value. This signal is processed through methods within the Selector class and the result is sent back out to the Data Bus class.

# LAFAYETTE COLLEGE

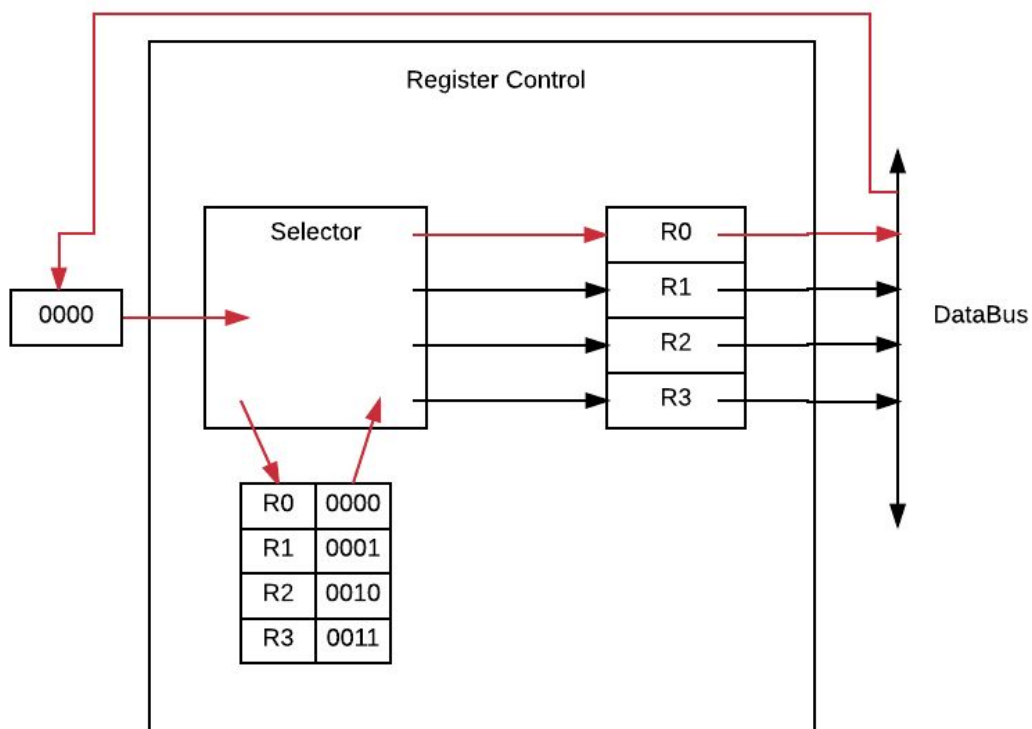


Figure 27: How the Selector class works within the Register Control class

# LAFAYETTE

---

## COLLEGE

### Arithmetic and Logical Unit (ALU)

The ALU class is the class which performs all the operations of the CPU. When the Program Counter class needs to be incremented so that it can point to the next instruction in the program, the incrementation is done with the ALU class. When the Stack Pointer class needs to be incremented or decremented depending on which operation is done on the stack within Main Memory, the incrementation or decrementation is done with the ALU class. The CPU is very reliant on the ALU because of its operational convenience. The overview of how the ALU class works is seen in Figure 28. When two values need to be added or subtracted, the first value to be pulled is stored in Register A which is a class that inherits from the Register class. For example, the add R0, R1 instruction fetches the value from R0 and stores that into Register A. Register A stores the value and waits until the other value is sent to the ALU. This is done because the main Data Bus class cannot carry more than one value and the ALU class cannot proceed with the operation until it is given both values. Register A is basically on hold until the second value is passed through. Once a value is stored there, the ALU class knows that it will receive a value from the data bus coming to B. The value going to be sent to B is the value from R1. The main Data Bus class sends this value directly from its temporary storage area. Once that value is sent from the main Data Bus class. Along with the signal of which operation is to be performed (this signal is sent through methods), the ALU class performs the operation. Once the result of the operation is calculated, the ALU class checks the result against all the possible flags that can be thrown. If the result is 0, the Zero Flag is set to 1. If the result is negative, the Sign Flag is set to 1. If the result is greater than what the registers can store, the Overflow Flag is set to 1. The result of the operation is stored in a register known as Register C. This register will send the result back to the main Data Bus class, which knows what to do with it. It is important for the ALU class to check the result of the operation against the conditions of the three flags after each operation. These flags are used in the conditional jumps.

# LAFAYETTE COLLEGE

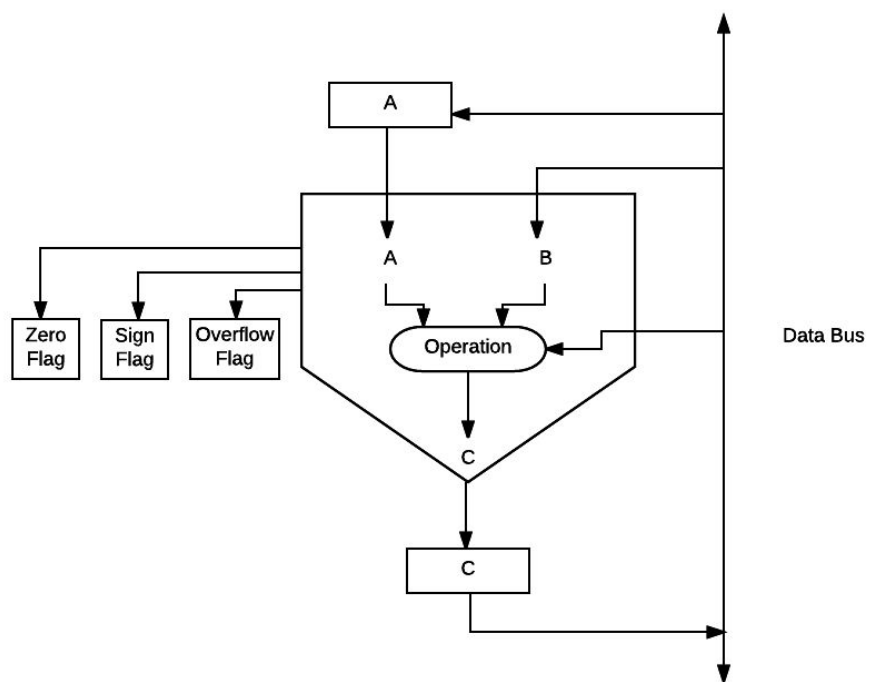


Figure 28: Overview of the ALU class

# LAFAYETTE COLLEGE

## CPU and Data Bus

Within the program, all of the components described above are passed through to the CPU class within its constructor. These components operate individually but only under the control of the CPU class and the Data Bus class. The overview of the CPU class is seen in Figure 29. While the Data Bus class does a great deal of work, it is actually itself a subclass of the CPU class. The CPU class uses every single one of the classes shown within the diagram to process an instruction. It is important to note that the RTN Control class and its attached classes (RTN Fetch, RTN RMOV, RTN MMIOV, RTN Jump, RTN Operation, RTN Stack) are not passed into the CPU class. This is because the information regarding the RTN steps are normally hard coded within a computer's processor and are not a part of the Central Processing Unit.

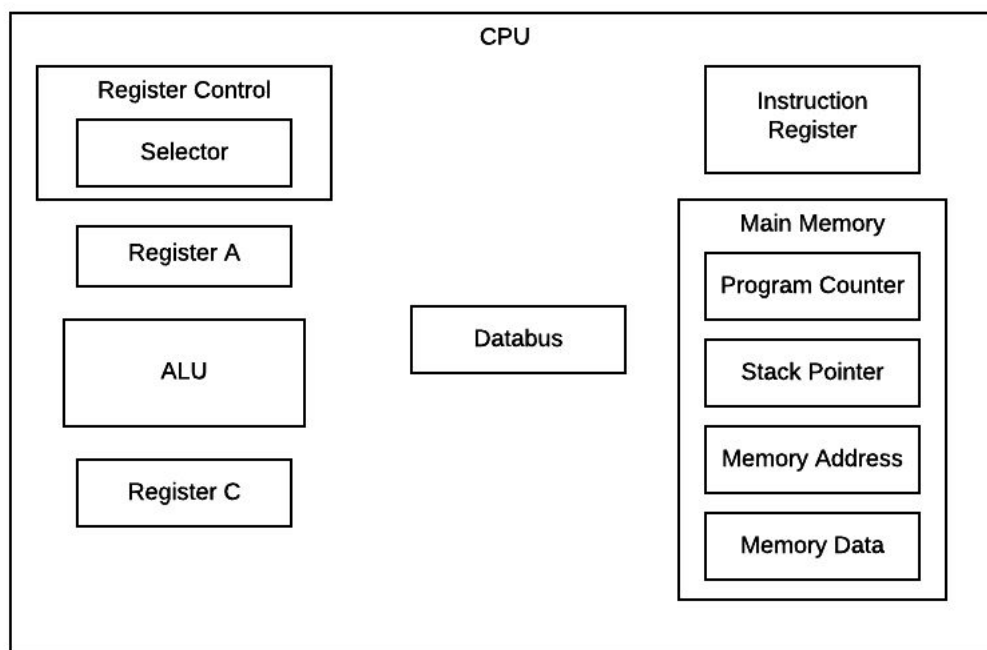


Figure 29: Overview of CPU class

# LAFAYETTE COLLEGE

## Organization of Unit Tests

Every class within the program is given an individual test. The ALU class is tested by feeding the class several inputs and checking to make sure that the correct results are outputted and the correct flags are set if the results match up. The Register C and Register A classes can be tested in a similar fashion as the Register class was testing but these are testing with the ALU. Register A can be given a value and the tests ensure that that value is not sent to the ALU class until a B value is sent to the ALU. Register C can be tested by ensuring that the result of the operation from the ALU class is the same value as what is stored in Register C. The Binary Assembly Structure class is tested by ensuring that certain instructions match up with their given binary instruction format. The Main Memory class is tested by ensuring that the Program Counter class only has access to address locations 0x00 to 0x09 and that the Stack Pointer class only has access to address locations 0x10 to 0x20. It can also be tested by checking to make sure that the Machine Code file was properly loaded into each byte of memory, this can be done by testing the first instruction against the first 2 byte of memory. The Register class is a classic object class which is tested by creating Register objects and ensuring that when a value is inserted, that same value is returned as the data and so on. The Register Control class can be tested by ensuring that the HashMap of registers match up with what the user knows the file has. It can also be tested by sending in several different inputs to the Selector class and ensuring that the correct Register data is returned. The RTN Control class can be tested by sending in instructions and ensuring that the correct RTN sub class is being linked and sent back to the CPU. The RTN subclasses (Fetch, Stack, RMOV, MMov, Stack, Jump) can be tested with the CPU. It can be ensured that the subclasses are instructing the CPU to complete the correct steps within the RTN.

The CPU class is the biggest class to test, since all the components are passed through to it. The test would consist of creating objects for all of those elements and passing them through to a CPU object. From there, the Y86 instructions would have to be passed through to the CPU manually. These instructions are usually passed in from a file (a new file could be created with new instructions) but the CPU class is testing for instruction functionality rather than ability for the program to decode instructions from a file. The test would ensure that all the components are working the way the RTN indicates and that by the end of the instruction, all registers that were supposed to be filled or emptied are, etc..

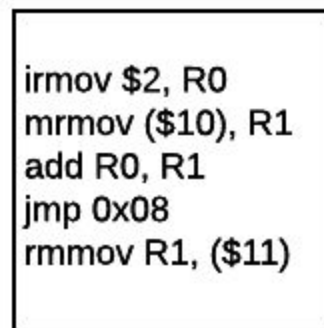
Unfortunately, I was unable to create any unit tests for this project. However, the detailed descriptions above give an idea of what the unit tests would have been like had they been implemented.



# LAFAYETTE COLLEGE

## Test Programs

There are 5 instructions that the program is running during its simulation and these come from the Machine Code file loaded in. These instructions can be seen in Figure 30.



```
irmov $2, R0
mrmov ($10), R1
add R0, R1
jmp 0x08
rmmov R1, ($11)
```

Figure 30: Instructions being loaded into program from Machine Code file

The simulation is able to correctly run all of the Y86 instructions except call, halt and ret. Unfortunately, time ran out before I was able to implement those three instructions. They all have RTN in the RTN file but weren't implemented within the CPU. The call instruction calls a label within program and the Program Counter should move its pointer to the address where that label starts. The code starts running from that address in the program. The ret instruction indicates that a function with a program has finished running and it returns the most recent register that has had a value stored to it. The halt function terminates execution of the program.

In order to run the simulation in BlueJ:

Right click on the Main class, click the "void main(String[] args)" option. The simulation will run and a GUI will appear on the screen with a Step and an Exit button. Click on the Step button to see how the program runs through each instruction being processed. There are only 5 instructions being processed. The GUI will indicate when the program has reached the end of the simulation.

# LAFAYETTE COLLEGE

The first instruction to be completed is “irmov \$2, R0” which will move an immediate value of 2 into Register 0. Figure 31a shows what the program prints out upon execution of this instruction. The Program Counter starts off by pointing at address 0x00 within Main Memory. The instruction that is pulled from Main Memory at address 0x00 is ‘0101000010020000’ in binary. From there, it is sent to the Instruction Register to be decoded and it is determined that the program will execute the ‘irmov 2 R0’ instruction. The CPU looks for the RTN of the instruction using the RTN Control class which directs it to the RTN RMOV class where it executes the first step of moving the immediate value (#) into the register. The available Data Bus sends the value of 2 with a destination of register R0 to the Selector in the Register Control class. The Selector finds the destination register and stores a value of 2 in it, which is 10 in binary.

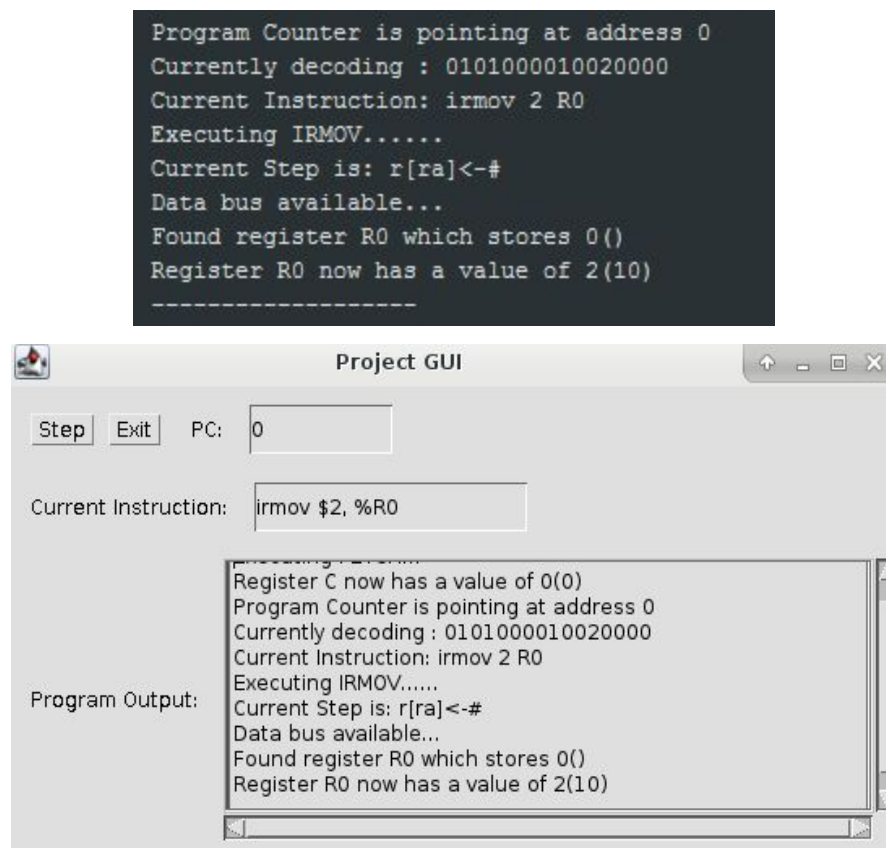


Figure 31a

# LAFAYETTE COLLEGE

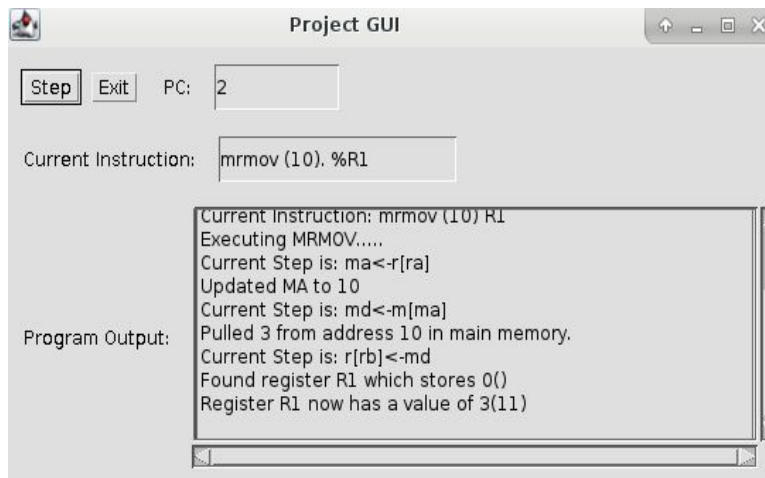
The next instruction to be completed is “mrmov (10) R1” which store the value at address 10 in Main Memory to register R1. Figure 31b shows what the program prints out upon execution of this instruction. The Program Counter has incremented (using the ALU to increment the address it is pointing at, the result Register C now stores the incremented address) and it now pointing at address 0x02 in Main Memory. The instruction that is pulled from Main Memory at address 0x02 is ‘01000000010100001’ in binary. From there, it is sent to the Instruction Register to be decoded and it is determined that the program will execute the ‘mrmov (10) R1’ instruction. The CPU looks for the RTN of the instruction using the RTN Control class which directs it to the RTN MMOV class where it executes the first step of sending the given address (0x10) to Memory Address register. Now that the Memory Address Register contains the address of the data that is to be pulled, the CPU executes the next step which is to use the address to fetch the data. The Memory Address class sends the address to the Main Memory class, the Main Memory class finds a value of 3 stored at that address in memory and sends it to the Memory Data class. Now that the Memory Data class contains a value of 3, the Data Bus class will fetch that value and send it to the destination register (R1). Register R1 currently stores nothing but once the value has gone through the Selector, it now stores a value of 3 which is 11 in binary.

```

Register C now has a value of 2(10)

Program Counter is pointing at address 2
Currently decoding : 01000000010100001
Current Instruction: mrmov (10) R1
Executing MRMOV.....
Current Step is: ma<-r[ra]
Updated MA to 10
Current Step is: md<-m[ma]
Pulled 3 from address 10 in main memory.
Current Step is: r[rb]<-md
Found register R1 which stores 0()
Register R1 now has a value of 3(11)
-----

```



Project GUI

Step Exit PC: 2

Current Instruction: mrmov (10). %R1

Program Output:

```

Current Instruction: mrmov (10) R1
Executing MRMOV.....
Current Step is: ma<-r[ra]
Updated MA to 10
Current Step is: md<-m[ma]
Pulled 3 from address 10 in main memory.
Current Step is: r[rb]<-md
Found register R1 which stores 0()
Register R1 now has a value of 3(11)

```

Figure 31b

# LAFAYETTE COLLEGE

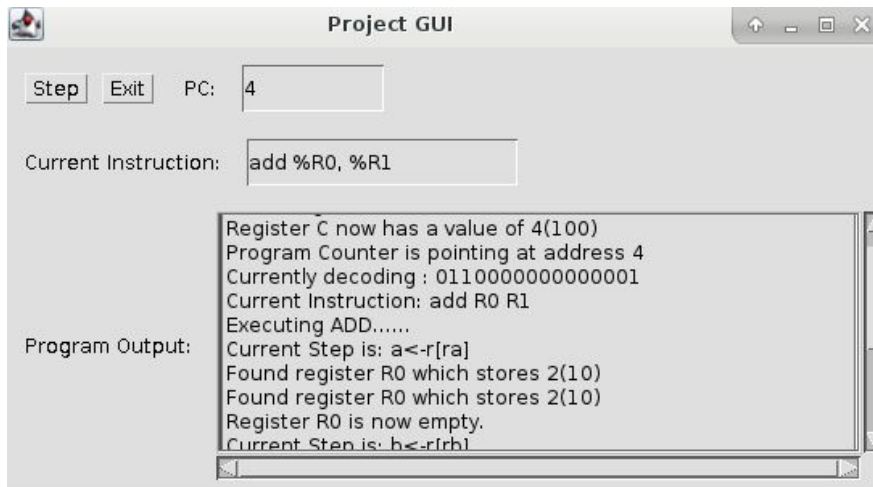
The next instruction to be completed is “add R0 R1” which takes the value from R0, adds it to the value in R1 and stores the result in R1. Figure 31c shows what the program prints out upon execution of this instruction. The Program Counter has incremented (using the ALU to increment the address it is pointing at, the result Register C now stores the incremented address) and it is now pointing at address 0x04 in Main Memory. The instruction that is pulled from Main Memory at address 0x04 is ‘0110000000000001’ in binary. From there, it is sent to the Instruction Register to be decoded and it is determined that the program will execute the ‘add R0 R1’ instruction. The CPU looks for the RTN of the instruction using the RTN Control class which directs it to the RTN Operations class where it executes the first step of identifying the value stored at R0. This is a value of 2 and it is sent to the Register A. The value of register R1 is identified as 3 and it is sent to the B value of the ALU. The addition operation is completed and Register C now stores a value of 5 which is 101 in binary. Since the result of the add instruction is to be stored at R1, the final step is to store the value in Register C to register R1.

```

Register C now has a value of 4(100)

Program Counter is pointing at address 4
Currently decoding : 0110000000000001
Current Instruction: add R0 R1
Executing ADD.....
Current Step is: a<-r[ra]
Found register R0 which stores 2(10)
Found register R0 which stores 2(10)
Register R0 is now empty.
Current Step is: b<-r[rb]
Found register R1 which stores 3(11)
Current Step is: c<-a+b
Register C now has a value of 5(101)
Current Step is: r[rb]<-c
Found register R1 which stores 3(11)
Register R1 now has a value of 5(101)
-----

```



The screenshot shows a window titled "Project GUI" with standard window controls. It contains a "Step" button, an "Exit" button, and a "PC:" label followed by a text box containing the number "4". Below this is a "Current Instruction:" label followed by a text box containing "add %R0, %R1". At the bottom is a "Program Output:" label followed by a large text area. This text area contains the same output text as the code block above, showing the step-by-step execution of the "add R0 R1" instruction, including register lookups, the addition operation, and the final update of register R1.

Figure 31c

# LAFAYETTE COLLEGE

The next instruction to be completed is “jmp 08” which unconditionally moves the address of the Program Counter from where it currently is to the address indicated by the address following the jmp instruction. Figure 31d shows what the program prints out upon execution of this instruction. The Program Counter has incremented (using the ALU to increment the address it is pointing at, the result Register C now stores the incremented address) and it now pointing at address 0x06 in Main Memory. The instruction that is pulled from Main Memory at address 0x06 is ‘0111000000001000’ in binary. From there, it is sent to the Instruction Register to be decoded and it is determined that the program will execute the ‘jmp 08’ instruction. The CPU looks for the RTN of the instruction using the RTN Control class which directs it to the RTN Jump class where it executes the first step of updating the value of the Program Counter to be 0x08.

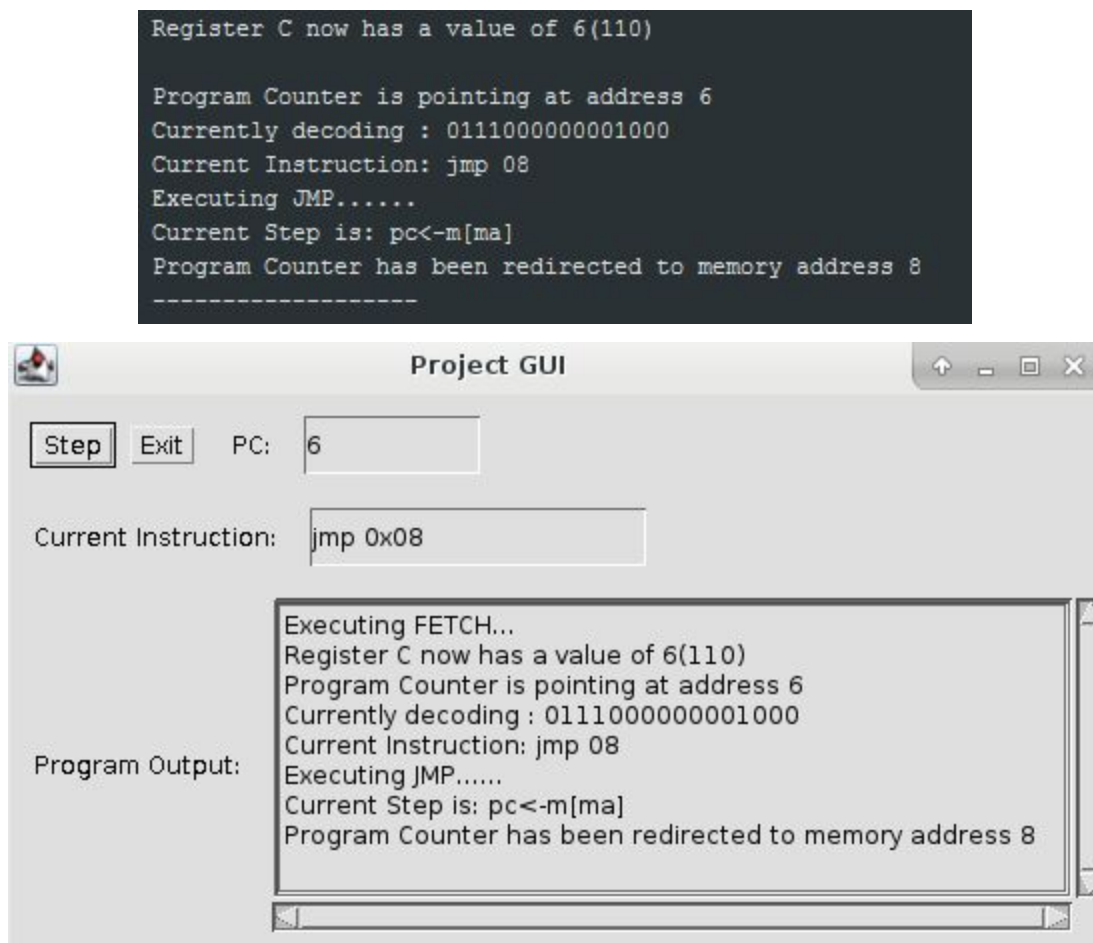


Figure 31d

# LAFAYETTE COLLEGE

The next instruction to be completed is “rmmov R1 (11)” which moves the value stored at register R1 to the byte in memory indicated by address 0x11. Figure 31e shows what the program prints out upon execution of this instruction. The Program Counter has incremented (using the ALU to increment the address it is pointing at, the result Register C now stores the incremented address) and it now pointing at address 0x08 in Main Memory. The instruction that is pulled from Main Memory at address 0x08 is ‘0011000000011011’ in binary. From there, it is sent to the Instruction Register to be decoded and it is determined that the program will execute the ‘rmmov R1 (11)’ instruction. The CPU looks for the RTN of the instruction using the RTN Control class which directs it to the RTN MMOV class where it executes the first step of updating the Memory Address register with that of the memory location that a value is to be stored to, which is address 0x11. The next step is to use that memory address to fetch the value stored at that address in main memory which is currently null since nothing is stored there. Then, the value of the register R1 (a value of 5 after the add instruction) is moved to the Memory Data register which updated the Main Memory array.

```
Register C now has a value of 8(1000)

Updated MA to 10
Program Counter is pointing at address 8
Currently decoding : 0011000000011011
Current Instruction: rmmov R1 (11)
Executing RMMOV.....
Current Step is : ma<-r[rb]
Updated MA to 11
Current Step is : md<-m[ma]
Pulled null from address 11 in main memory.
Current Step is : md<-r[ra]
Found register R1 which stores 5(101)
5 has been stored at memory location 11
-----
```

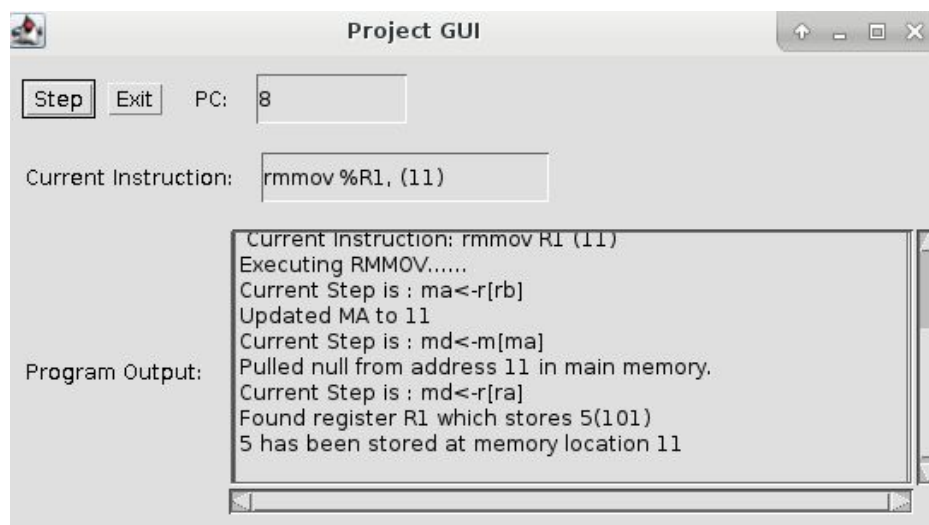


Figure 31e



# LAFAYETTE COLLEGE

## Design Flaws

My design definitely had its flaws because no design is perfect. Everything in the Configuration File is basically hard coded into the program. Adjusting for the Configuration File is something that would alter my program a significant amount: many components would have to change and more variables would have to be added. Register A is a variable inside the ALU rather than a separate Register object. This is something that could easily be fixed with some slight adjustments of the code. A new class could be created with the same functionality as the Register class, the ALU could be altered so that it accepted the value of Register A only when the Data Bus sent over a value for B. The Data Bus doesn't work exactly the way it should be working: sending signals and carrying information. It works mainly through methods implemented specifically for certain RTN steps.

This project was a huge learning opportunity for me: design wise and coding wise. Implementing the CPU taught me how a CPU worked overall as well as how each component in the CPU worked. The coding portion made me appreciate Object Oriented Design (and the Java language) because of how easy it is to implement basic designs.

# LAFAYETTE COLLEGE

## Bibliography

Figure 1:

Pfaffmann, Jeffrey. CPU Simulator. 2016. Photograph.

[http://www.cs.lafayette.edu/~pfaffmaj/courses/fl6/cs203/projects/project1/CS203Project1\\_F16.pdf](http://www.cs.lafayette.edu/~pfaffmaj/courses/fl6/cs203/projects/project1/CS203Project1_F16.pdf). Web. 11 November 2016.

Figure 2:

Bryant, Randal E., and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Boston: Prentice Hall, 2011. Print.

Figure 3:

Stack. Digital image. *Tutorials Point*. N.p., n.d. Web. 14 Nov. 2016.

Figure 4 :

Fetch-Execute Cycle. Digital image. *Learning Computing*. N.p., n.d. Web. 11 Nov. 2016.

Bryant, Randal E., and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Boston: Prentice Hall, 2011. Print.

Beal, By Vangie. "What Is Central Processing Unit (CPU)? Webopedia Definition." *What Is Central Processing Unit (CPU)? Webopedia Definition*. N.p., n.d. Web. 14 Nov. 2016.

Wood, Sussman, Herman, Plan. "Assembly Language Programming." *CMSC 216* (n.d.): n. pag. *Www.cs.umd.edu*. University of Maryland. Web. 14 Nov. 2016.

"How The Computer Works: The CPU and Memory." *How The Computer Works: The CPU and Memory*. URI, n.d. Web. 14 Nov. 2016.

"What Is Program Counter? - Definition from WhatIs.com." *WhatIs.com*. N.p., n.d. Web. 14 Nov. 2016.

"What Is A CPU and What Does It Do? [Technology Explained]." *MakeUseOf*. N.p., n.d. Web. 14 Nov. 2016.

"Teach-ICT OCR GCSE Computing - the Central Processing Unit (CPU)." *Teach-ICT OCR GCSE Computing - the Central Processing Unit (CPU)*. N.p., n.d. Web. 14 Nov. 2016.



# LAFAYETTE COLLEGE

"What's a Clock?" *What's a Clock?* N.p., n.d. Web. 14 Nov. 2016.

Gribble, Paul. "7. Memory : Stack vs Heap." *7. Memory : Stack vs Heap*. N.p., n.d. Web. 14 Nov. 2016.