Computer Engineering Coursework

Design:

I implemented a single-instruction, multi-cycle processor, like the one displayed below.

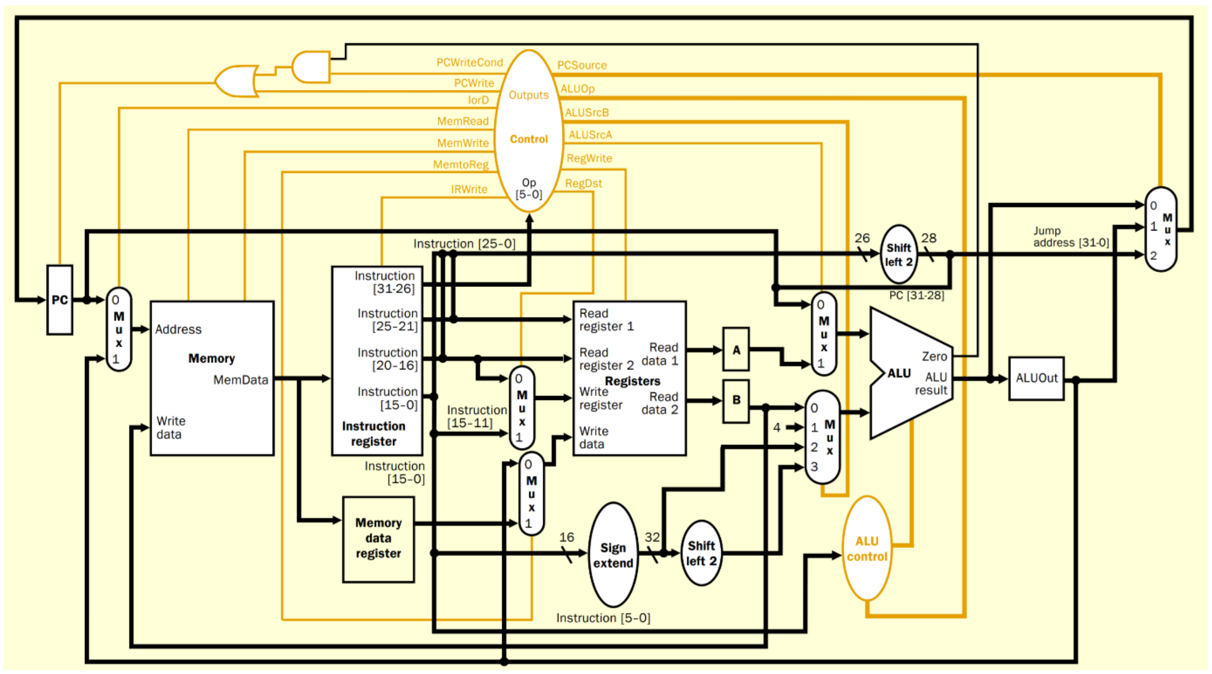


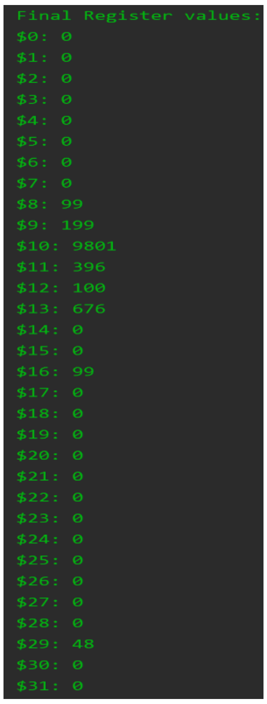
Figure 1: Diagram of implemented multi-cycle "processor"

The register topology used in a MIPS processor, was implemented. Only the register numbers are in the simulator, not the register names. $fp and $gp did not need implementing ($28 and $30) and $sp value was set to the maximum value in the PC (slightly changing its true function). All values in the register file are converted to decimal, for debugging purposes.

The memory layout used did not mimic that used in a MIPS processor and instead simply started at 0 (with a maximum address value of 1000 although this could be increased by simply increasing the array size of *memoryFiler*). The memory is still byte addressed and word aligned.

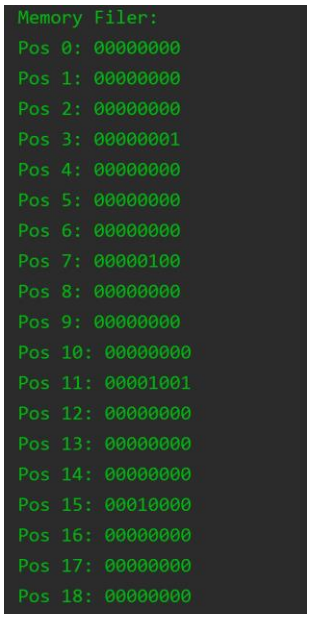
The limited instruction set implemented includes:

- beq
- add
- jr
- j
- jal
- sw
- lw
- addi



```
Final Register values:
$0: 0
$1: 0
$2: 0
$3: 0
$4: 0
$5: 0
$6: 0
$7: 0
$8: 99
$9: 199
$10: 9801
$11: 396
$12: 100
$13: 676
$14: 0
$15: 0
$16: 99
$17: 0
$18: 0
$19: 0
$20: 0
$21: 0
$22: 0
$23: 0
$24: 0
$25: 0
$26: 0
$27: 0
$28: 0
$29: 48
$30: 0
$31: 0
```

Figure 2: Register Topology



```
Memory Filer:
Pos 0: 00000000
Pos 1: 00000000
Pos 2: 00000000
Pos 3: 00000001
Pos 4: 00000000
Pos 5: 00000000
Pos 6: 00000000
Pos 7: 00000100
Pos 8: 00000000
Pos 9: 00000000
Pos 10: 00000000
Pos 11: 00001001
Pos 12: 00000000
Pos 13: 00000000
Pos 14: 00000000
Pos 15: 00010000
Pos 16: 00000000
Pos 17: 00000000
Pos 18: 00000000
```

Figure 3: Incomplete sample of outputted memory file

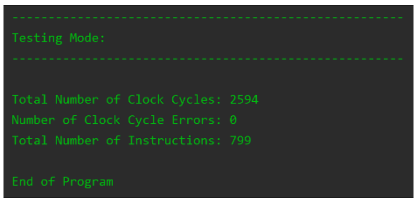
These instructions are sufficient to implement a fully functioning MIPS processor which can process the showcase program, along with leafed/non-leafed procedures and arrays.

Functionality:

The system has an automated MIPS assembly to machine code translator. This is implemented by a class, which uses multiple methods to differentiate between the IS and produce the binary string instruction (machine code). The MIPS assembly .txt file is inputted into the translator class.

The menu of the program allows you to pick from 3 different choices.

- Picking option 0 will run through the program cycle by cycle, simply by pressing the enter key. When an instruction is completed after a certain number of cycles, the instruction will be outputted to the console, letting the user know what instruction has just been performed. The values of all the registers in the register file is outputted after each clock cycle is executed. This could be very useful to the user when debugging their own MIPS assembly code.
- Picking option 1 will simply run through the entire program and output the final values in memory along with the final values in the register file.
- Picking option 2 will take the user to the testing suite. This suite does not test their code but instead tests the simulator and shows information about the code that was just executed. This information includes
  - The total number of clock cycles executed by the simulator
  - The number of clock cycle errors
    - These would arise if an instruction (for example a j) would take more/less cycles than expected from the instruction
  - The total number of instructions executed



```
-----------------------------------------------------
Testing Mode:
-----------------------------------------------------

Total Number of Clock Cycles: 2594
Number of Clock Cycle Errors: 0
Total Number of Instructions: 799

End of Program
```

Figure 4: Example of the testing suite

The correct result being produced by the program can be checked by looking at the register file output at the end of the program.

The processor consists of modules, and each module (ALU, ALU out, Control, Instruction register, Memory file, Memory data register, PC, Register File, registers A and B) has its own class, whose job is to filter inputs and carry out any operations the specific module may do. This includes multiplexer operations (implemented by switch, case statements), arithmetic, binary sorting etc.

Each of these modules are called by the *runInstructions* function. This incrementally steps through the instructions (based on the value in the PC) and calls other nested functions which carry out a single clock cycle/stage. At the beginning of each nested function, the control signals for that specific clock cycle are setup, ensuring that the correct dataflow occurs through the Datapath. These control signals are shown in appendix A. All instructions do stages 1 (*runIF*) and 2 (*runID*), the instruction fetch, and instruction decode stages.

runIF ensures that the instruction register stores the instruction at the PC address. It then increments the PC by 4.

$$IR = instruction[PC\ address]$$

$$PC = PC + 4$$

runID ensures that the temporary registers, A and B, store the rs and rt values for the next clock cycle. This also ensures that the ALU Out temporary register, stores the sign-extended, shifted left by 2, PC referenced value to also be used in the next clock cycle.

$$A = rs, B = rt$$

$$ALUOut = PC + (sign - extend(imm) \ll 2)$$

From here on, the cycles become instruction dependant. The opcode is used to differentiate between different instructions and from there, the correct nested functions can be called.

The beq command (opcode 000100) is implemented using *runBeqEXE*. This carries out a referenced jump (the reference being the value of the PC), depending on whether or not the zero flag has been raised by the ALU. The inputs the ALU are A and B, and the ALU is subtracting them letting us know whether they are equal if the zero flag is raised.

$$if(A == B), PC = ALUOut$$

This instruction takes a total of 3 clock cycles.

The add command (opcode 000000) uses the funct section of its instruction (100000) to setup the correct ALU operation. In clock cycle 3, *runAddEXE* is called such that the values in A and B can added together to give the ALU Out temporary register value.

$$A + B = ALUOut$$

runAddWB is then called to write ALU Out to register rd, in the register file.

$$reg[rd] = ALUOut$$

This instruction takes a total of 4 clock cycles.

The jr command (opcode 000000) also uses the funct section of its instruction (001000) to setup the correct ALU operation. In clock cycle 3, *runJrEXE* is called such that the values in A and B can added together to give the ALU Out temporary register value.

$$A + B = ALUOut$$

runJrWB is then called to write ALU Out to the PC, as a form of direct addressing

$$PC = ALUOut$$

This instruction takes a total of 4 clock cycles.

The j and jal commands (opcodes 000010 and 000011 respectively) EXE stage is implemented using the *runJEXE.* This carries out a direct jump, based on the values in the 26 bit immediate.

$$PC = PC[31 - 28] | (IR[25 - 0] \ll 2)$$

This instruction takes a total of 3 clock cycles.

The sw, lw and addi commands (opcodes 101011, 010111 and 001000 respectively) EXE stage is implemented using *runSwLwAddiEXE*. This sign-extends the immediate value, adds the value in the temporary register, A, and returns it to ALU Out.

$$ALUOut = A + sign - extend(imm)$$

The operations then become instruction dependant.

sw uses *runSwMA* to write to memory, the value of register B, at memory position ALU Out.

$$memory[ALUOut] = B$$

This instruction takes a total of 4 clock cycles.

lw passes the data in memory at the position ALU Out to the MDR using *runLwMA*.

$$MDR = memory[ALUOut]$$

It then uses runLwWb to write the value in the MDR to the register file at position, rt.

$$reg[rt] = MDR$$

This instruction takes a total of 5 clock cycles.

addi writes the value in ALU Out to the register file, also at position rt.

$$reg[rt] = ALUOut$$

Basic Testing:

At the end of every function, there is a commented-out section which starts with ‘proof working’. All of these check that each clock cycle (depending on instruction) is performing the correct operation and producing the correct results. The correct results are all showcased in the above equations and this is what the ‘proof working’ checks for, that the equations are being fulfilled by the function.

Along with this, the clock cycle method (mode 0), was implemented early on to ensure that each clock cycle had the correct ‘proof working’ results, that each instruction had the correct number of clock cycles and (by outputting all of the values in the register file) that each register had the correct value associated with it. Using these in conjunction with the IDE debugger allowed me to fully debug any potential issues that could arise in the simulator.

I created 3 test programs to ensure correct operation of the simulator, all of which were checked with the mentioned methods I implemented to ensure correct operation of the simulator. These test programs checked that every instruction in my IS operated as expected.

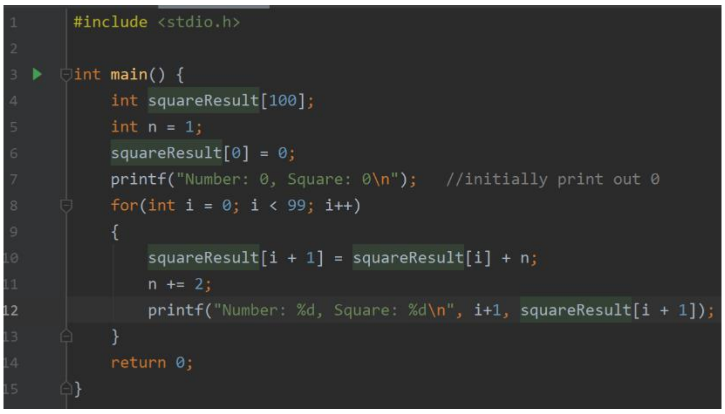
I initially used mode 2 to ensure that the number of clock cycles was correct for each instruction. I then used mode 0 to ensure that the register file values were correctly updating after each cycle. At the end of all of this, mode 1 could be used to check that the final register values were correct and that the result had been correctly stored. Full details of testing these programs are in appendix B.

Showcase Testing:

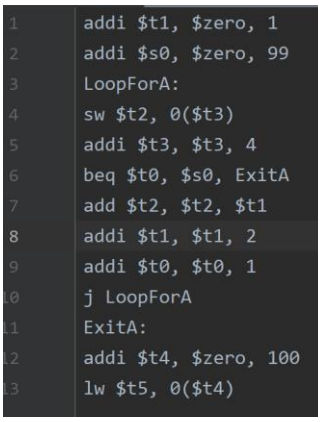
The showcase program uses the following algorithm.

$$i_n = i_{n-1} + n, where\ n = 1, 3, 5, 7 \ldots$$

This program was initially written in C and then translated into MIPS assembly code.



```c
#include <stdio.h>

int main() {
    int squareResult[100];
    int n = 1;
    squareResult[0] = 0;
    printf("Number: 0, Square: 0\n");   //initially print out 0
    for(int i = 0; i < 99; i++)
    {
        squareResult[i + 1] = squareResult[i] + n;
        n += 2;
        printf("Number: %d, Square: %d\n", i+1, squareResult[i + 1]);
    }
    return 0;
}
```

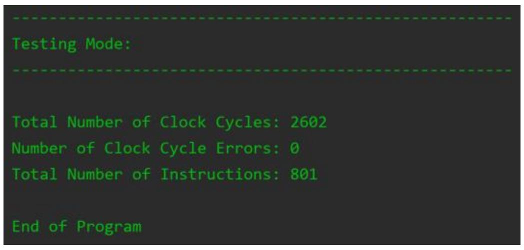


```
addi $t1, $zero, 1
addi $s0, $zero, 99
LoopForA:
sw $t2, 0($t3)
addi $t3, $t3, 4
beq $t0, $s0, ExitA
add $t2, $t2, $t1
addi $t1, $t1, 2
addi $t0, $t0, 1
j LoopForA
ExitA:
addi $t4, $zero, 100
lw $t5, 0($t4)
```

Figure 5: C and MIPS assembly showcase Program

This program created an array to store the results of the square of all of the values from 0 to 99. This is then stored in memory from the position 0. At the end of the program, is a load word. This was used to test if the correct answer to 25 squared was acquired. This is checked by looking at register $t5 ($13), which gives us 625. $t2 ($10) gives us the final result, and should, therefore, be 99 squared, which is 9801.

I used mode 2 to see how many total clock cycles were required to implement this, giving me 2602 clock cycles. Further evidence of results is in appendix C.



```
-------------------------------------------------------
Testing Mode:
-------------------------------------------------------


Total Number of Clock Cycles: 2602
Number of Clock Cycle Errors: 0
Total Number of Instructions: 801

End of Program
```

Figure 6: Testing Mode Showcase program output

Conclusions:

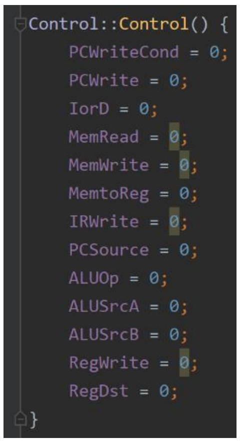
Overall, I am very content with my final simulator. I believe it gives the user an easy to use interface with a lot of functionality. The different modes of use allow for methods of debugging for both the simulator itself along with the MIPS assembly code that has been inputted to it. All of the functionality I planned to achieve by the end of the coursework (with regards to the simulator) were accomplished and the code is presented in an easy to understand format, with comments and commented out extra testing for the user of the code to easily understand how the simulator works.

The simulator can succesfully implement all of the testing programs along with the showcase program, and it allows the user to

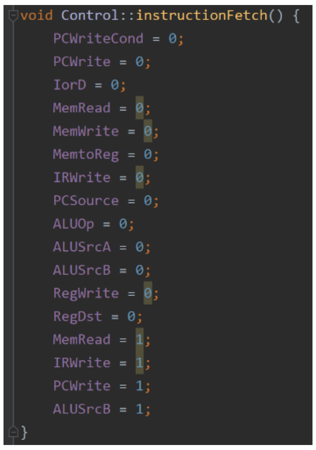
- Run the simulator cycle by cycle
- Run the simulator straight to the end
- Run a testing booth of the simulated MIPS code

If I were to continue making the processor, I would expand the ISA to include more instructions such as la, sll etc. and enhance the capability of the translater to a wider IS. The testing booth could also potentially give more data with regards to the simulation, however, none of these improvements are necessities to the simulator and I believe that the finished simulator is more than a sufficient MIPS processor simulator.

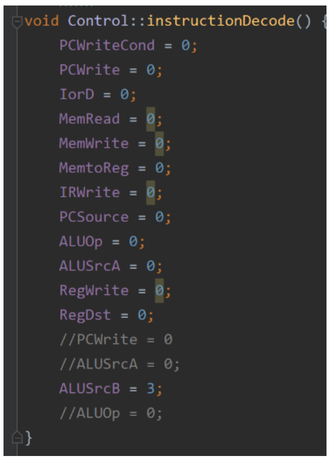
Appendix A: Control Signals



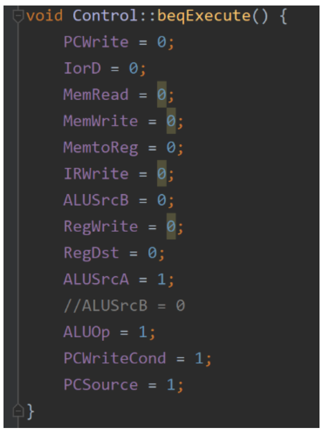
```
Control::Control() {
    PCWriteCond = 0;
    PCWrite = 0;
    IorD = 0;
    MemRead = 0;
    MemWrite = 0;
    MemtoReg = 0;
    IRWrite = 0;
    PCSource = 0;
    ALUOp = 0;
    ALUSrcA = 0;
    ALUSrcB = 0;
    RegWrite = 0;
    RegDst = 0;
}
```



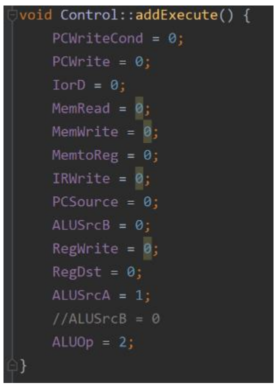
```
void Control::instructionFetch() {
    PCWriteCond = 0;
    PCWrite = 0;
    IorD = 0;
    MemRead = 0;
    MemWrite = 0;
    MemtoReg = 0;
    IRWrite = 0;
    PCSource = 0;
    ALUOp = 0;
    ALUSrcA = 0;
    ALUSrcB = 0;
    RegWrite = 0;
    RegDst = 0;
    MemRead = 1;
    IRWrite = 1;
    PCWrite = 1;
    ALUSrcB = 1;
}
```



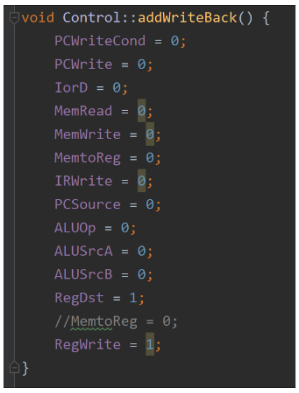
```
void Control::instructionDecode() {
    PCWriteCond = 0;
    PCWrite = 0;
    IorD = 0;
    MemRead = 0;
    MemWrite = 0;
    MemtoReg = 0;
    IRWrite = 0;
    PCSource = 0;
    ALUOp = 0;
    ALUSrcA = 0;
    RegWrite = 0;
    RegDst = 0;
    //PCWrite = 0
    //ALUSrcA = 0;
    ALUSrcB = 3;
    //ALUOp = 0;
}
```



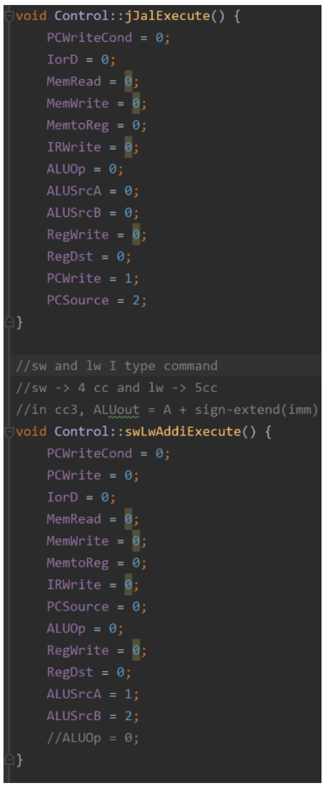
```
void Control::beqExecute() {
    PCWrite = 0;
    IorD = 0;
    MemRead = 0;
    MemWrite = 0;
    MemtoReg = 0;
    IRWrite = 0;
    ALUSrcB = 0;
    RegWrite = 0;
    RegDst = 0;
    ALUSrcA = 1;
    //ALUSrcB = 0
    ALUOp = 1;
    PCWriteCond = 1;
    PCSource = 1;
}
```



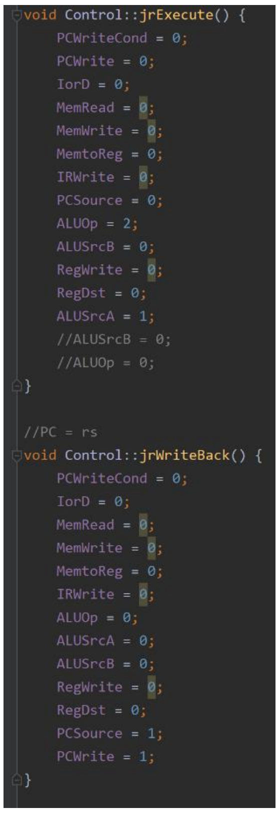
```
void Control::addExecute() {
    PCWriteCond = 0;
    PCWrite = 0;
    IorD = 0;
    MemRead = 0;
    MemWrite = 0;
    MemtoReg = 0;
    IRWrite = 0;
    PCSource = 0;
    ALUSrcB = 0;
    RegWrite = 0;
    RegDst = 0;
    ALUSrcA = 1;
    //ALUSrcB = 0
    ALUOp = 2;
}
```



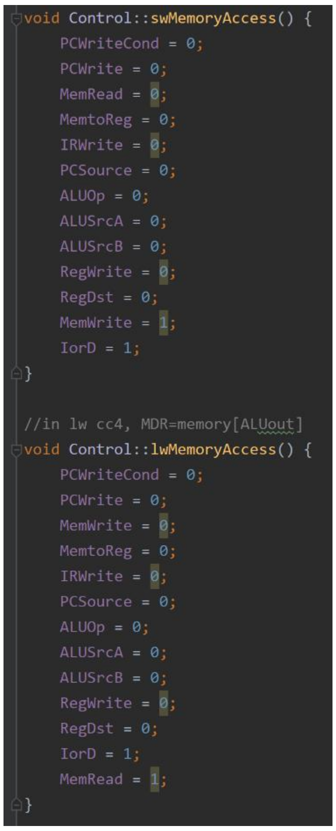
```
void Control::addWriteBack() {
    PCWriteCond = 0;
    PCWrite = 0;
    IorD = 0;
    MemRead = 0;
    MemWrite = 0;
    MemtoReg = 0;
    IRWrite = 0;
    PCSource = 0;
    ALUOp = 0;
    ALUSrcA = 0;
    ALUSrcB = 0;
    RegDst = 1;
    //MemtoReg = 0;
    RegWrite = 1;
}
```



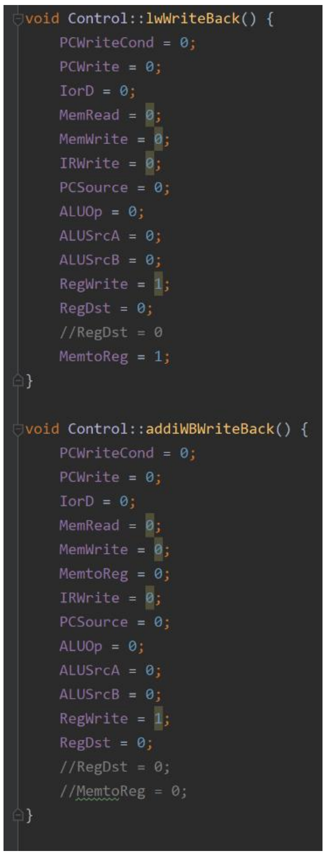
```
void Control::jJalExecute() {
    PCWriteCond = 0;
    IorD = 0;
    MemRead = 0;
    MemWrite = 0;
    MemtoReg = 0;
    IRWrite = 0;
    ALUOp = 0;
    ALUSrcA = 0;
    ALUSrcB = 0;
    RegWrite = 0;
    RegDst = 0;
    PCWrite = 1;
    PCSource = 2;
}

//sw and lw I type command
//sw -> 4 cc and lw -> 5cc
//in cc3, ALUout = A + sign-extend(imm)
void Control::swLwAddiExecute() {
    PCWriteCond = 0;
    PCWrite = 0;
    IorD = 0;
    MemRead = 0;
    MemWrite = 0;
    MemtoReg = 0;
    IRWrite = 0;
    PCSource = 0;
    ALUOp = 0;
    RegWrite = 0;
    RegDst = 0;
    ALUSrcA = 1;
    ALUSrcB = 2;
    //ALUOp = 0;
}
```



```
void Control::jrExecute() {
    PCWriteCond = 0;
    PCWrite = 0;
    IorD = 0;
    MemRead = 0;
    MemWrite = 0;
    MemtoReg = 0;
    IRWrite = 0;
    PCSource = 0;
    ALUOp = 2;
    ALUSrcB = 0;
    RegWrite = 0;
    RegDst = 0;
    ALUSrcA = 1;
    //ALUSrcB = 0;
    //ALUOp = 0;
}

//PC = rs
void Control::jrWriteBack() {
    PCWriteCond = 0;
    IorD = 0;
    MemRead = 0;
    MemWrite = 0;
    MemtoReg = 0;
    IRWrite = 0;
    ALUOp = 0;
    ALUSrcA = 0;
    ALUSrcB = 0;
    RegWrite = 0;
    RegDst = 0;
    PCSource = 1;
    PCWrite = 1;
}
```



```
void Control::swMemoryAccess() {
    PCWriteCond = 0;
    PCWrite = 0;
    MemRead = 0;
    MemtoReg = 0;
    IRWrite = 0;
    PCSource = 0;
    ALUOp = 0;
    ALUSrcA = 0;
    ALUSrcB = 0;
    RegWrite = 0;
    RegDst = 0;
    MemWrite = 1;
    IorD = 1;
}

//in lw cc4, MDR=memory[ALUout]
void Control::lwMemoryAccess() {
    PCWriteCond = 0;
    PCWrite = 0;
    MemWrite = 0;
    MemtoReg = 0;
    IRWrite = 0;
    PCSource = 0;
    ALUOp = 0;
    ALUSrcA = 0;
    ALUSrcB = 0;
    RegWrite = 0;
    RegDst = 0;
    IorD = 1;
    MemRead = 1;
}
```



```
void Control::lwWriteBack() {
    PCWriteCond = 0;
    PCWrite = 0;
    IorD = 0;
    MemRead = 0;
    MemWrite = 0;
    IRWrite = 0;
    PCSource = 0;
    ALUOp = 0;
    ALUSrcA = 0;
    ALUSrcB = 0;
    RegWrite = 1;
    RegDst = 0;
    //RegDst = 0
    MemtoReg = 1;
}

void Control::addiWBWriteBack() {
    PCWriteCond = 0;
    PCWrite = 0;
    IorD = 0;
    MemRead = 0;
    MemWrite = 0;
    MemtoReg = 0;
    IRWrite = 0;
    PCSource = 0;
    ALUOp = 0;
    ALUSrcA = 0;
    ALUSrcB = 0;
    RegWrite = 1;
    RegDst = 0;
    //RegDst = 0;
    //MemtoReg = 0;
}
```

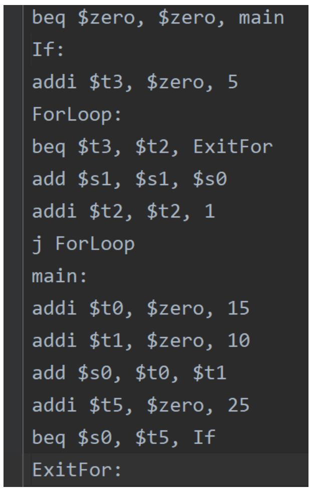
Figure 7: Control Signals

Appendix B: Detailed Testing

Test Program 1:

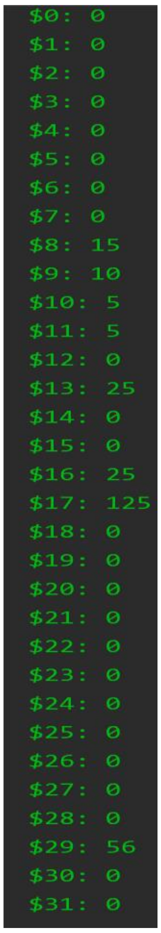
The output to the program should be in $s1 ($17) and it should be 125

This program produces 2 variables (x and y) and then sums them. If the sum==25 (which it is), the program will then increment the sum 5 times in a for loop and store the result to $17 (125).



```
beq $zero, $zero, main
If:
addi $t3, $zero, 5
ForLoop:
beq $t3, $t2, ExitFor
add $s1, $s1, $s0
addi $t2, $t2, 1
j ForLoop
main:
addi $t0, $zero, 15
addi $t1, $zero, 10
add $s0, $t0, $t1
addi $t5, $zero, 25
beq $s0, $t5, If
ExitFor:
```

Figure 8: Test Program 1 MIPS



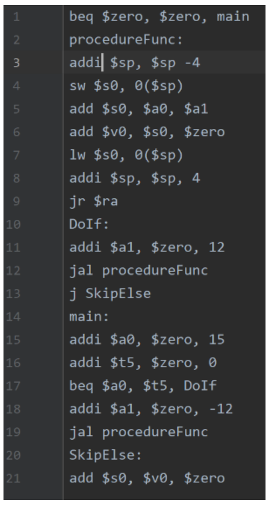
```
$0: 0
$1: 0
$2: 0
$3: 0
$4: 0
$5: 0
$6: 0
$7: 0
$8: 15
$9: 10
$10: 5
$11: 5
$12: 0
$13: 25
$14: 0
$15: 0
$16: 25
$17: 125
$18: 0
$19: 0
$20: 0
$21: 0
$22: 0
$23: 0
$24: 0
$25: 0
$26: 0
$27: 0
$28: 0
$29: 56
$30: 0
$31: 0
```

Figure 9: Test Program 1 register file result

Test Program 2:

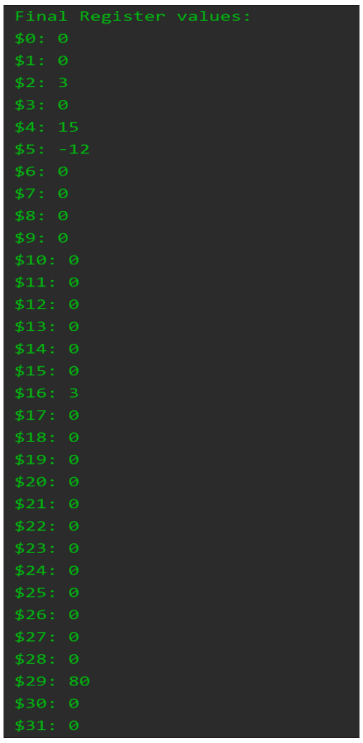
This sets a variable a ($a0) to 15, and a variable b to -12. It then runs a leaf procedure which increments the sum of the 2 variables.

The expected result is in $s0 ($16) and should be 3.



```
beq $zero, $zero, main
procedureFunc:
addi $sp, $sp -4
sw $s0, 0($sp)
add $s0, $a0, $a1
add $v0, $s0, $zero
lw $s0, 0($sp)
addi $sp, $sp, 4
jr $ra
DoIf:
addi $a1, $zero, 12
jal procedureFunc
j SkipElse
main:
addi $a0, $zero, 15
addi $t5, $zero, 0
beq $a0, $t5, DoIf
addi $a1, $zero, -12
jal procedureFunc
SkipElse:
add $s0, $v0, $zero
```

Figure 10: Test Program 2 MIPS



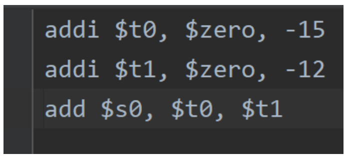
```
Final Register values:
$0: 0
$1: 0
$2: 3
$3: 0
$4: 15
$5: -12
$6: 0
$7: 0
$8: 0
$9: 0
$10: 0
$11: 0
$12: 0
$13: 0
$14: 0
$15: 0
$16: 3
$17: 0
$18: 0
$19: 0
$20: 0
$21: 0
$22: 0
$23: 0
$24: 0
$25: 0
$26: 0
$27: 0
$28: 0
$29: 80
$30: 0
$31: 0
```

Figure 11: Test Program 2 register file result

Test Program 3:

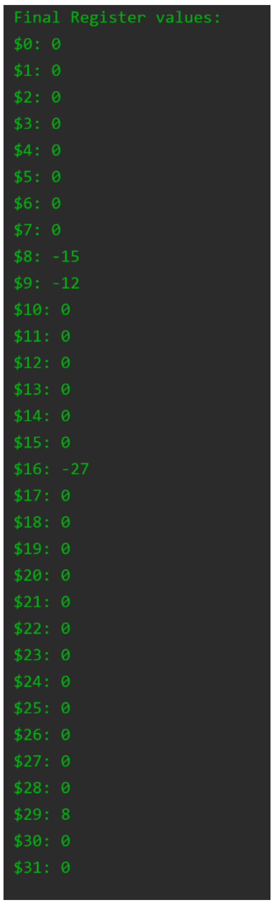
This is a very simple program and checks that addi and add can use negative values.

The expected result in $s0 ($16) is -27.



```
addi $t0, $zero, -15
addi $t1, $zero, -12
add $s0, $t0, $t1
```

Figure 12: Test Program 3 MIPS



```
Final Register values:
$0: 0
$1: 0
$2: 0
$3: 0
$4: 0
$5: 0
$6: 0
$7: 0
$8: -15
$9: -12
$10: 0
$11: 0
$12: 0
$13: 0
$14: 0
$15: 0
$16: -27
$17: 0
$18: 0
$19: 0
$20: 0
$21: 0
$22: 0
$23: 0
$24: 0
$25: 0
$26: 0
$27: 0
$28: 0
$29: 8
$30: 0
$31: 0
```

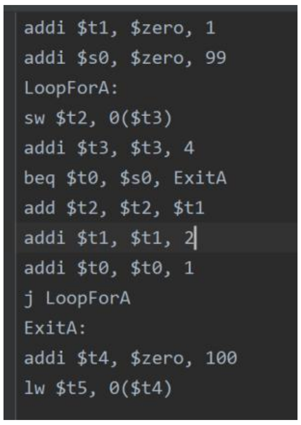
Figure 13: Test Program 3 register file result

Appendix C: Showcase Program

This program gets the squares of all integers from 0 to 99.

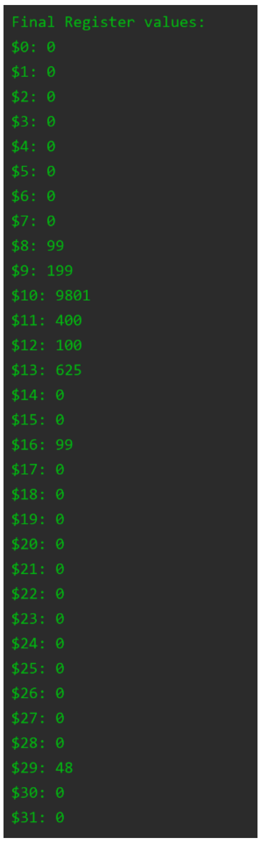
The final square result (99) is stored in $t2 ($10 -> 9801) and the loaded square of 25 is stored is $t5 ($13) and is 625.

This was used in conjuction with modes 0 and 2 for debugging.



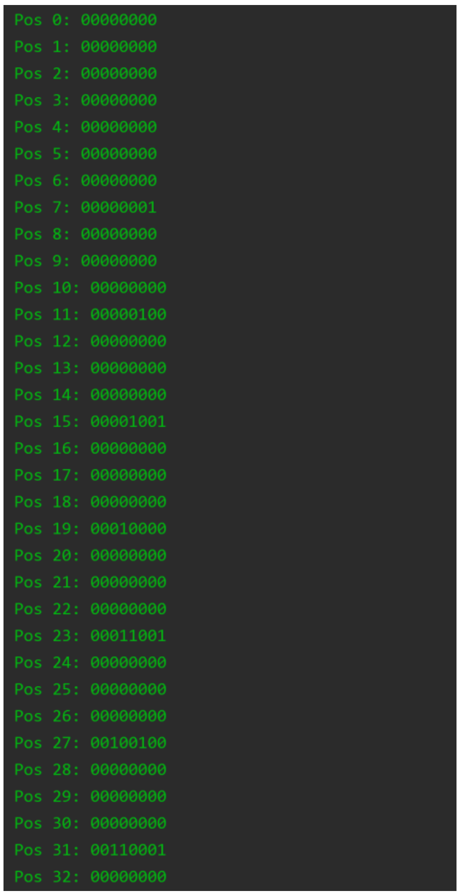
```
addi $t1, $zero, 1
addi $s0, $zero, 99
LoopForA:
sw $t2, 0($t3)
addi $t3, $t3, 4
beq $t0, $s0, ExitA
add $t2, $t2, $t1
addi $t1, $t1, 2
addi $t0, $t0, 1
j LoopForA
ExitA:
addi $t4, $zero, 100
lw $t5, 0($t4)
```

Figure 14: Showcase Program MIPS



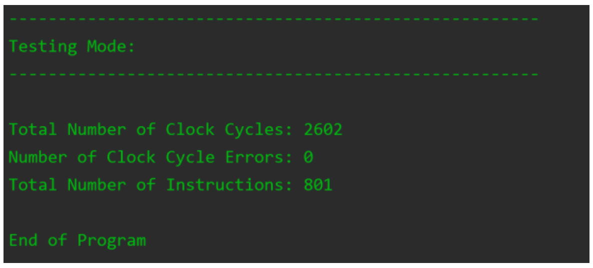
```
Final Register values:
$0: 0
$1: 0
$2: 0
$3: 0
$4: 0
$5: 0
$6: 0
$7: 0
$8: 99
$9: 199
$10: 9801
$11: 400
$12: 100
$13: 625
$14: 0
$15: 0
$16: 99
$17: 0
$18: 0
$19: 0
$20: 0
$21: 0
$22: 0
$23: 0
$24: 0
$25: 0
$26: 0
$27: 0
$28: 0
$29: 48
$30: 0
$31: 0
```

Figure 15: Showcase Program register file result



```
Pos 0: 00000000
Pos 1: 00000000
Pos 2: 00000000
Pos 3: 00000000
Pos 4: 00000000
Pos 5: 00000000
Pos 6: 00000000
Pos 7: 00000001
Pos 8: 00000000
Pos 9: 00000000
Pos 10: 00000000
Pos 11: 00000100
Pos 12: 00000000
Pos 13: 00000000
Pos 14: 00000000
Pos 15: 00001001
Pos 16: 00000000
Pos 17: 00000000
Pos 18: 00000000
Pos 19: 00010000
Pos 20: 00000000
Pos 21: 00000000
Pos 22: 00000000
Pos 23: 00011001
Pos 24: 00000000
Pos 25: 00000000
Pos 26: 00000000
Pos 27: 00100100
Pos 28: 00000000
Pos 29: 00000000
Pos 30: 00000000
Pos 31: 00110001
Pos 32: 00000000
```

Figure 16: First 8 words in memory for Showcase Program



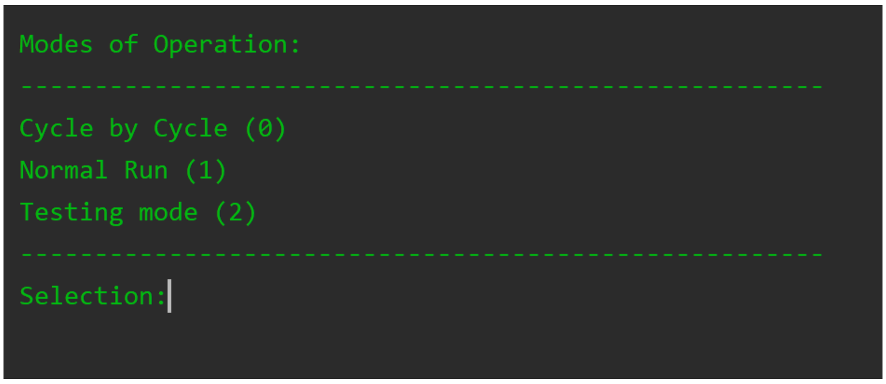
```
----------------------------------------------------
Testing Mode:
----------------------------------------------------

Total Number of Clock Cycles: 2602
Number of Clock Cycle Errors: 0
Total Number of Instructions: 801

End of Program
```

Figure 17: Showcase Program Testing Booth

Appendix D: Simulator Menu

The user can use this menu to input the mode they wish to use the simulator in.

- 0 -> cycle by cycle
- 1 -> normal run
- 2 -> testing booth



```
Modes of Operation:
-------------------------------------------------------
Cycle by Cycle (0)
Normal Run (1)
Testing mode (2)
-------------------------------------------------------
Selection:
```

Figure 18: Simulator Menu