UNIVERSITY OF LJUBLJANA | FRI Faculty of Computer and Information Science

MiMo

Project report for the course Computer Organisation

Author: Nik Uljarević, 63220345

Mentor: Sen. Lect. Dr. Robert Rozman

Table of Contents

Introduction

In this report, I will address and solve the five tasks related to the use and modification of the MiMo CPU model, as described below. Additionally, I will detail the optional/additional work I have undertaken during this project. The full MiMo project, including all relevant files and modifications, is provided alongside this report. This includes:

- The mimo_v05a_OR_EVO.circ circuit file, where the changes described in this report were made
- The basic_microcode.def file, containing definitions and implementations of all instructions from the provided instruction list
- The ucontrol.rom and udecision.rom files, which store the compiled instructions from the basic_microcode.def file
- The test.s and test.ram files, which store the program used to complete Task 3
- The micro_assembler_v2.pl and micro_assembler_v2.exe files, which include the minor changes and fixes to the microcode assembler, as described in the Additional work section of this report
- All other files originally included in the base version of this project, which were utilized during the course of this work but were not edited and do not require further explanation

1. Task

Clearly write down the number and sequence of micro-operations that are executed when performing the machine instruction 'SW Rd, Immed' (example: 'SW r3, 65535').

The macroinstruction “SW Rd, Immed” executes five microinstruction:

1. **fetch: addrsel=pc irload=1**

   The first microinstruction executed is part of the fetch instruction, which is exectued at the start of every macroinstruction. This microinstruction loads the next macroinstruction into the instruction register, using the address stored in the program counter (PC). It achieves this by setting addrsel to pc, directing the RAM to the address stored in the PC and setting irload to 1, which loads the instruction from that address into the instruction register.
2. **pcload=1 pcsel=pc, opcode_jump**

   The next microinstruction, is also part of the fetch instruction. It increments the PC by 1 and jumps to the location of the next operation/instruction in the control and decision ROM, which is at the operation code of the next instruction plus 2. This microinstruction increments the PC by setting pcload to 1, allowing the PC to store the next value it receives and then setting pcsel to pc, so the PC simply stores PC + 1. The opcode_jump is handled by setting indexsel = 1, which causes the jump to occur not at the address currently pointed to by the decision ROM (which is 2), but at that address plus the opcode of the instruction stored in the instruction register, resulting in a jump to opcode + 2.
3. **addrsel=pc imload=1**

This is the first microinstruction unique to this macroinstruction. It stores the Immed parameter into the immediate register and then jumps to the next microinstruction. It accomplishes this by setting addrsel to pc and imload to 1, which loads the immediate register with the value stored at the RAM address currently pointed to by the PC. It then jumps to address 83, as this is the address specified in both the true and false bits at this microinstruction's address in the decision ROM.

4. **addrsel=immed datawrite=1 datasel=dreg, goto pcincr**

   This instruction stores the value from the register defined in the Rd parameter into the address specified by the immediate register and then jumps to the 'pcincr' command. It accomplishes this by setting datawrite to 1, enabling RAM write operations. It sets addrsel to immed and datasel to dreg, so the address is selected from the immediate register and the data written to that address comes from the register specified by the 3 bits that store the 'Rd' parameter of this macroinstruction. It then jumps to pcincr by having the address stored in both the true and false bits in it's decision ROM's address.

5. **pcincr: pcload=1 pcsel=pc, goto fetch**

   This microinstruction is used by most 32-bit macroinstructions, so instructions that have an immediate value. It increments the PC by 1 and then jumps back to the fetch command. It does this because the PC is currently pointing to the immediate value. This is achieved by incrementing the PC in the same manner as the second microinstruction and then setting the fetch command as the next instruction to execute, as described in the third and fourth microinstructions.

2. Task

From the list of machine instructions supported by the processor, select at least four and provide the corresponding sequences of microinstructions for their execution. The selected instructions should be more complex or non-trivial (opcode greater than 15) and from different groups—they should consist of at least 4 or preferably more microinstructions (your success will also be measured by this criterion). Briefly describe the implementation and especially explain the microprogramming records of the added machine instructions in the micro-processor.

Since all the macroinstructions I will explain in this section include the fetch instruction, which is executed before every macroinstruction, which has been explained in the previous section, I will not list it in any of the explanations. Therefore, each of the instructions below consists of the microinstructions I will be detailing, plus the two that are part of fetch and will not be mentioned.

divi Rd,Rs,immed (19)

1. addrsel=pc imload=1
2. aluop=div op2sel=immed dwrite=1 regsrc=aluout, goto pcincr
3. pcincr: pcload=1 pcsel=pc, goto fetch

The **divi** (DIVide Immediate) instruction is used to divide a value stored in a register by an immediate value and then store the result in another register.

The instruction first loads the integer provided in the immed parameter into the immediate register. It then divides the value stored in the register specified by Rs by the immed value, stores the result in the register defined by the Rd parameter, and jumps to the pcincr microinstruction. The pcincr microinstruction then increments the PC by one, so it points to the next instruction instead of the immediate value and finally jumps to the fetch instruction.

asri Rd,Rs,immed (28)

1. addrsel=pc imload=1
2. aluop=asr op2sel=immed dwrite=1 regsrc=aluout, goto pcincr
3. pcincr: pcload=1 pcsel=pc, goto fetch

The **asri** (Arithmetic Shift Right Immediate) instruction performs an arithmetic shift right on a value stored in a register by a specified immediate value and then stores the result in another register.

The instruction first loads the immediate value into the immediate register. It then uses the ALU to perform an arithmetic shift right on the value stored in the register defined by the Rs parameter, shifting it by the amount stored in the immediate register. The result is stored in the register defined by the Rd parameter, after which the instruction jumps to the pcincr microinstruction. The pcincr microinstruction increments the PC and then jumps to the fetch instruction.

beq Rs,Rt,immed

1. addrsel=pc imload=1
2. aluop=sub ps2sel=treg, if z then branch else pcincr
3. branch: pcload=1 pcsel=pcimmed, goto fetch **OR** pcincr: pcload=1 pcsel=pc, goto fetch

The **beq** (Branch if EQual) instruction is used to compare the values stored in two registers and, if they are the same, perform a relative jump by the immediate value (PC + immediate).

First, the instruction loads the immediate value into the immediate register. It then subtracts the value stored in the register defined by the Rs parameter from the value stored in the register defined by the Rt parameter. If the subtraction results in zero (indicating the values are equal), the Z (zero) flag is set. If the Z flag is set, the instruction jumps to the branch microinstruction; otherwise, it proceeds to the fetch instruction. The branch microinstruction adds the value stored in the immediate register to the current PC value and stores the result in the PC, effectively performing the branch. The pcincr microinstruction, however, increments the PC so that it points to the next macroinstruction.

jsr immed (59)

1. addrsel=pc imload=1
2. aluop=sub op2sel=const1 regsrc=aluout swrite=1
3. pcload=1 pcsel=pc
4. datawrite=1 datasel=pc addrsel=sreg, goto jump
5. jump: pcload=1 pcsel=immed, goto fetch

The **jsr** (Jump to SubRoutine) instruction stores the address of the next instruction on the top of the stack and then jumps to a subroutine located at the address specified by the immediate value.

First, the instruction loads the immediate value into the immediate register. Next, it decrements the value stored in the R7 register, which points to the topmost value in the stack, so that it now points to the top of the stack, i.e., the next free space in the stack. It then increments the PC to point to the next macroinstruction in RAM. The current PC value is then stored at the address pointed to by the R7 register, effectively placing it on the stack. The instruction then jumps to the jump microinstruction. Finally, the jump microinstruction loads the value stored in the immediate register (which is the address of the subroutine in RAM) into the PC and jumps to the fetch instruction.

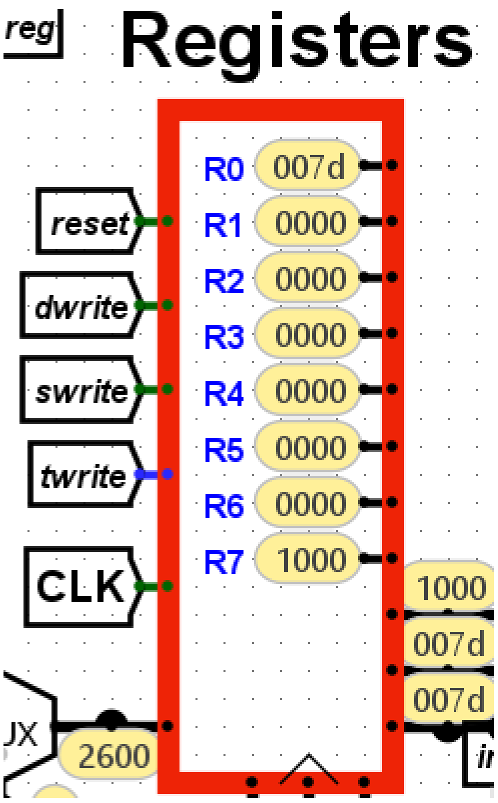
3. Task

Use the new machine instructions from task 2 in your own test program in the processor and test their functionality. Write a program that thoroughly test the added instructions. Explain the content of the relevant machine instructions. Describe the events occurring with each machine instruction and determine how many clock cycles each of them takes and the total for all instructions.

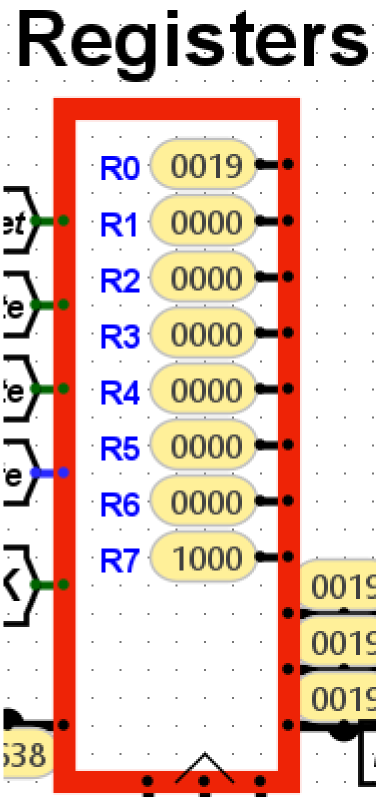
To test the instruction added in the previous section of this report, I created the program shown below. I have also listed the number of cycles that each instruction takes. Instructions that are skipped or not executed are marked with an "X" next to the number of cycles.

	Instructions:	Number of cycles/microinstructions:
main:	li r0, 125	4
	li r7, 0x1000	4
	divi r0, r0, 5	5
	jsr skip	7
	li r4, 9999	4 X
skip:	li r1, -16	4
	li r2, -4	4
	beq r1, r2, end	5
	asri r1, r1, 2	5
end:	beq r1, r2, end	5
		Number of all cycles: 47 | Number of cycles executed: 43

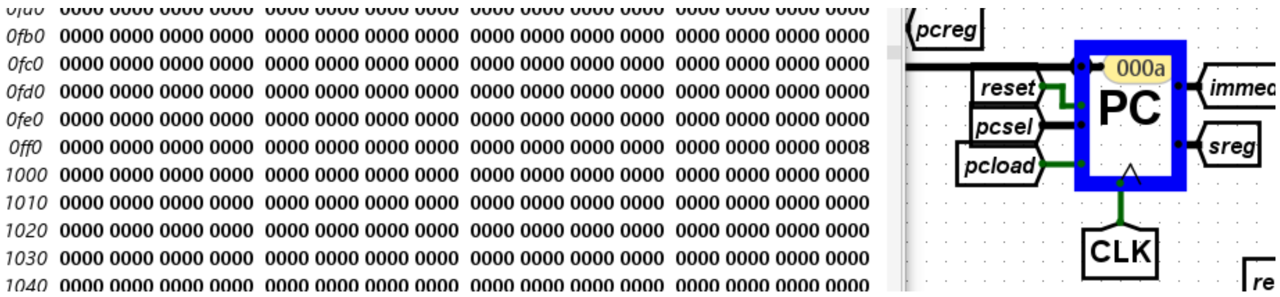
The program first stores 125 (0x7d) in register R0, which will be used to test the divi instruction. It also defines the top/start of the stack by storing the chosen address (0x1000) in the R7 register, which will be used to test the jsr instruction.



The divi instruction is then called to divide the value stored in R0 by 5 and store the result back into R0, which gives us a value of 25 (0x19).

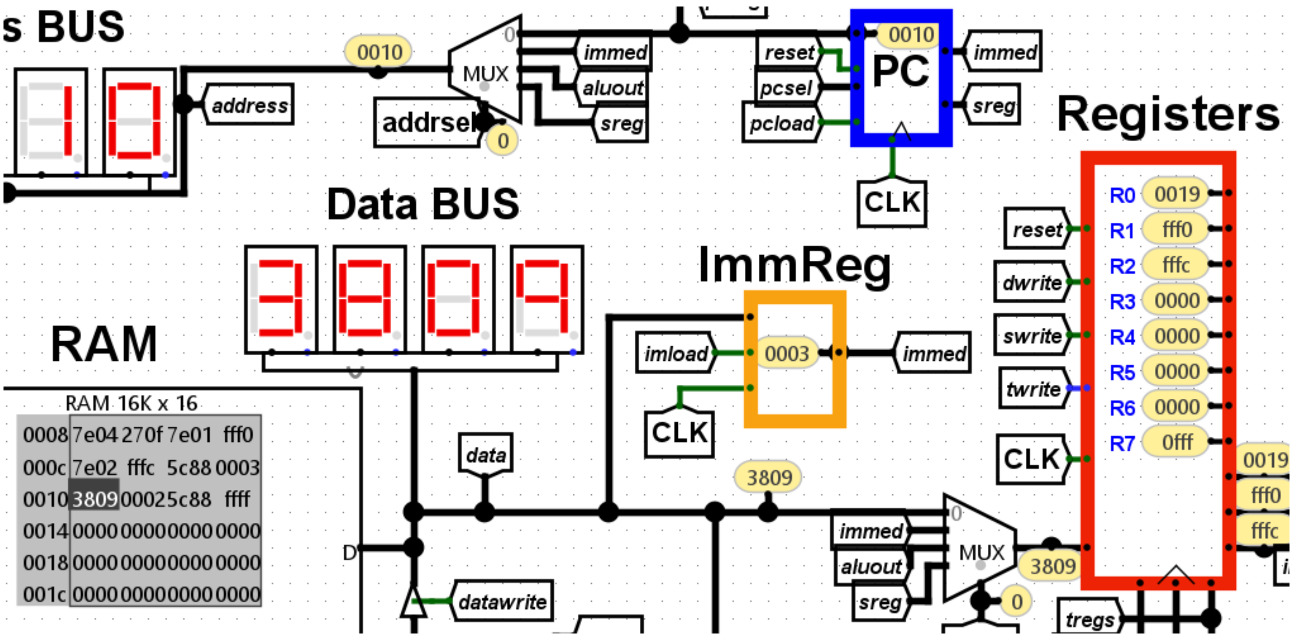


The jsr instruction then stores the address of the li r4, 9999 instruction (0x0008) at the top of the stack and jumps to the skip section (0x000a). The image below demonstrates that the li instruction's address is correctly stored at address 0x0fff, as the value in R7 is decremented before the jsr instruction stores the location of the next instruction. The image also shows that the PC successfully jumped to the address of the skip section (0x000a).



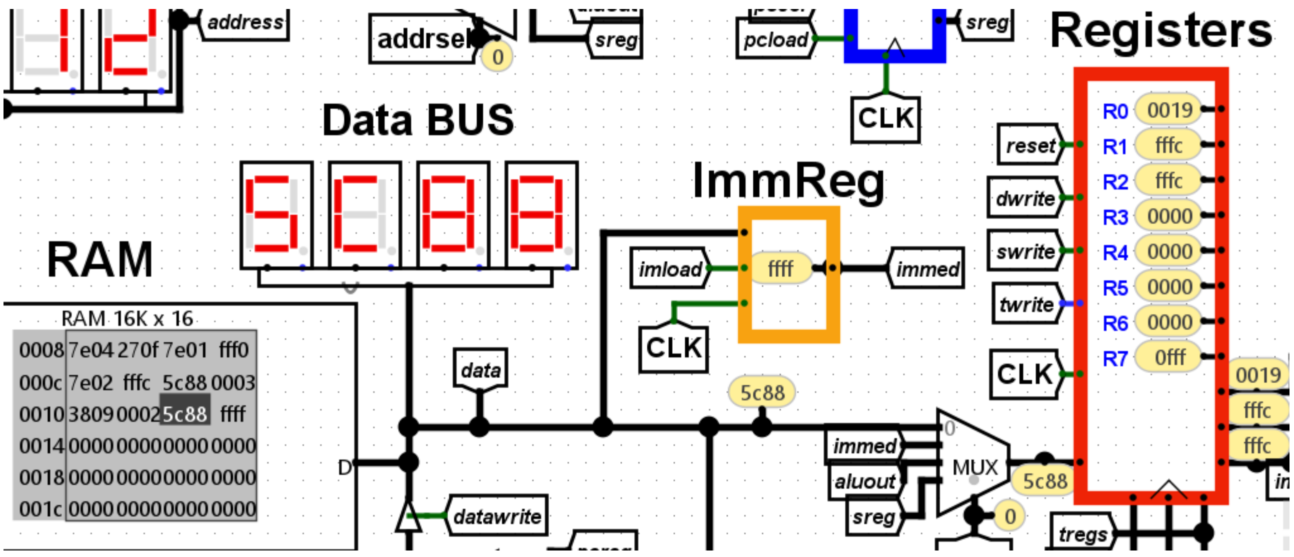
```
0fb0 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0fc0 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0fd0 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0fe0 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0ff0 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0008
1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
1010 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
1020 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
1030 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
1040 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

In the skip section, we load the R1 and R2 registers with -16 (0xfff0) and -4 (0xfffc), respectively, and then compare the two using the beq instruction. Since the two values are not equal, the program does not branch to the end section.



```
RAM 16K x 16
0008 7e04 270f 7e01 fff0
000c 7e02 fffc 5c88 0003
0010 3809 0002 5c88 ffff
0014 0000 0000 0000 0000
0018 0000 0000 0000 0000
001c 0000 0000 0000 0000
```

We then call the asri instruction with an immediate value of 2 to arithmetically shift the -16 value by two bits to the right, effectively dividing the value by 4 and resulting in -4 (0xfffc). The beq instruction is then called again to compare the two registers, and it correctly halts the program as the values are now equal, demonstrating that the instruction works as intended.

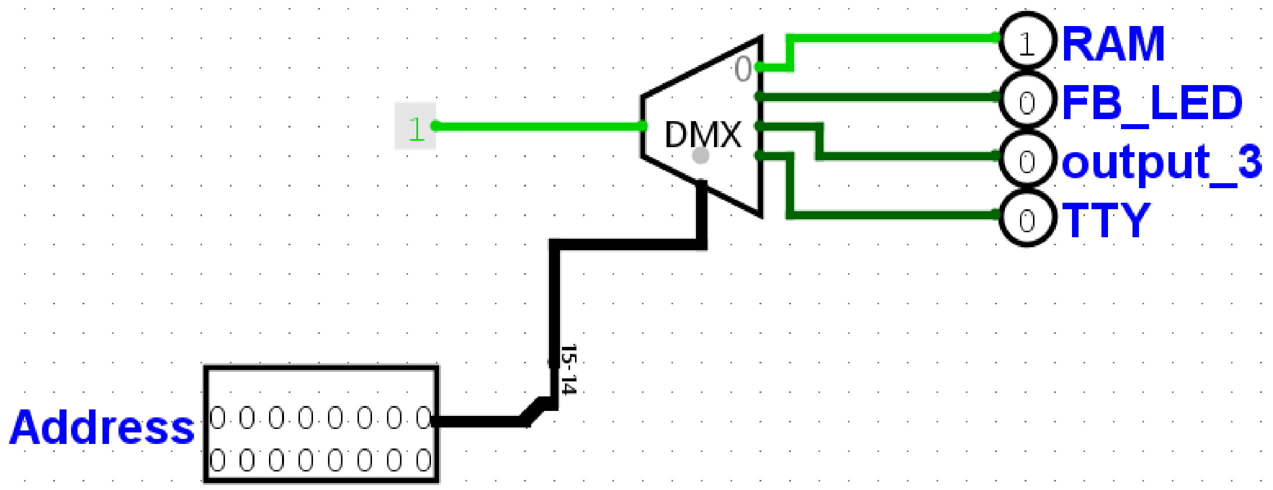


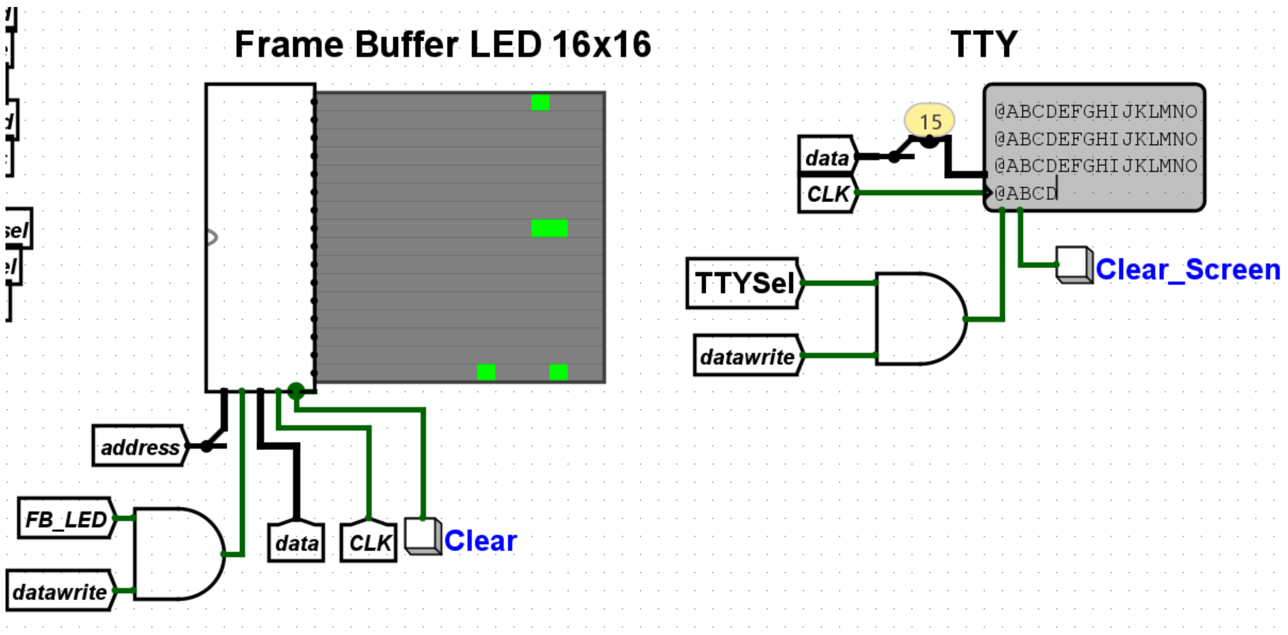
```
RAM 16K x 16
0008 7e04 270f 7e01 fff0
000c 7e02 fffc 5c88 0003
0010 3809 0002 5c88 ffff
0014 0000 0000 0000 0000
0018 0000 0000 0000 0000
001c 0000 0000 0000 0000
```

4. Task

Properly map the output devices FB 16x16 and TTY into the memory space using the 'memory-mapped I/O' principle—you can use incomplete address decoding. Ensure that the test program for I/O devices executes correctly after this change—meaning that the output devices should appear for writing only at their assigned addresses.

To fix the address decoder using the principle of memory-mapped I/O (MMIO), I used a 1/4 demultiplexer with a constant input of 1. The demultiplexer uses the 15th and 14th bits of the address as the select bits, reserving a quarter of the address space for each of the four I/O devices. I connected the output of the demultiplexer to separate pins, with the pins for both the TTY and FB devices connected to separate AND gates along with the datawrite tunnel. This ensures that the devices can only be written to, not read from, as both the device's address space and the datawrite flag need to be set for the operation to occur.





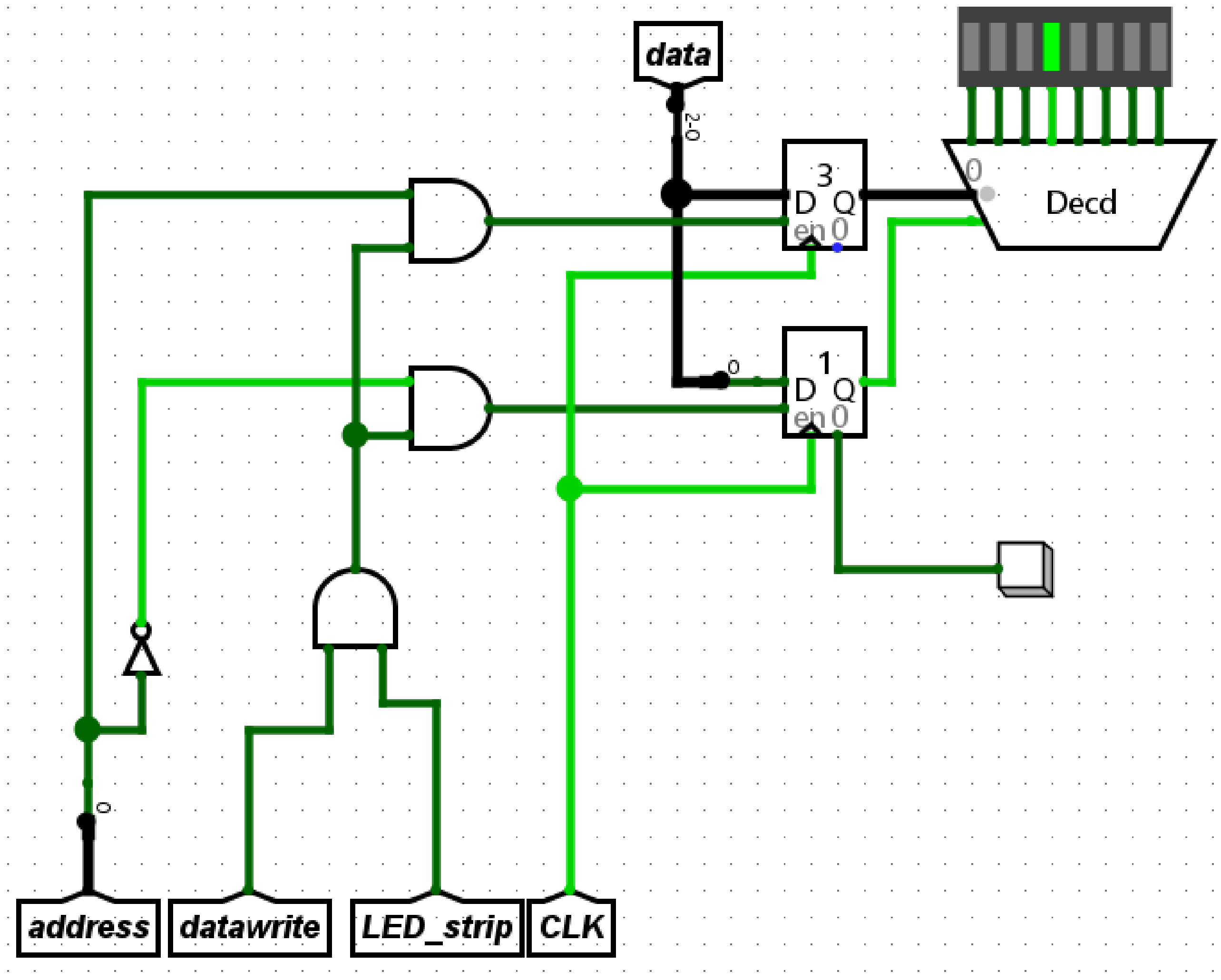
5. Task

In the MiMo model, space has been prepared for an additional I/O device. Describe all the necessary steps to correctly integrate the additional I/O device into the MiMo model, implement it and test it with an appropriate assembly program. Provide a clear description of all the steps with all the necessary details.

For the fourth I/O device, I have chosen to create an LED strip that stores its enabled/disabled state in a 1-bit register and the specific LED to be lit in a 3-bit register. I've configured it so that both the datawrite signal and the pin responsible for the fourth I/O device's address signal must be set, ensuring this device can only be accessed when data is being written to its address space. Additionally, the first bit in the device's address space determines whether the data sent will be written into the enable/disable register or the LED selector register. If the least significant bit is 0, the data will be written into the enable/disable register; otherwise, it will be written into the LED selector register.

The image below shows the device state after the following instructions have been executed:

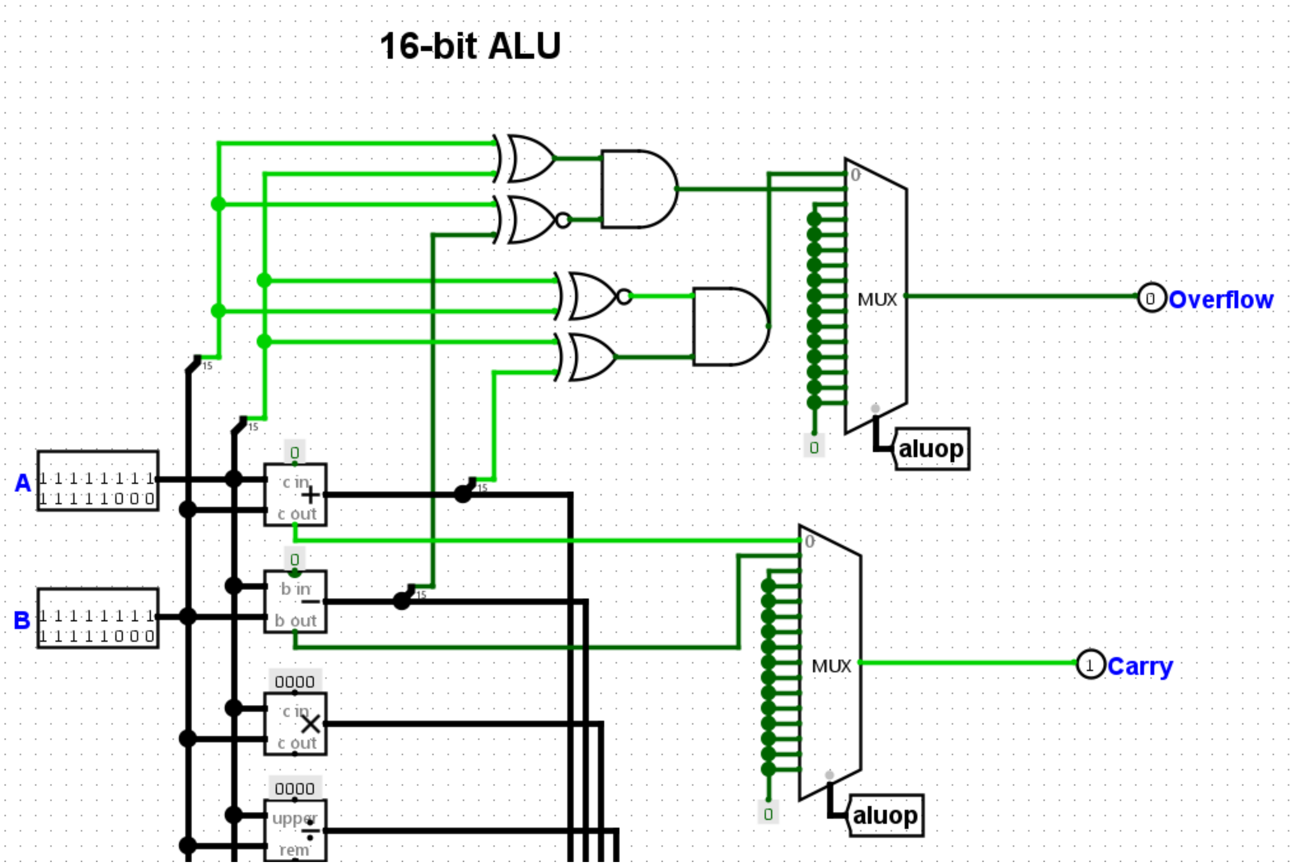
```
li r0, 1              // Stores the value that will be written into the enable/disable register
li r1, 3              // Stores the value that will be written into the LED selector register.
sw r0, 49152          // Writes the value of 1 into the enable/disable register, thereby enabling the
                      // LED strip
sw r1, 49153          // Writes the value of 3 into the LED selector register, thereby lighting the
                      // fourth LED
```



Additional work

In this section, I will outline all the additional work I have done on this project and explain some of the more interesting aspects. To begin with, I defined and implemented all the instructions in the instruction list. Since there are 74 instructions in total, I will not list them all here, but they are included in the “basic_microcode.def” file accompanying this report. I will, however, explain the implementation of some of the more intriguing ones.

Additionally, I implemented the ‘jgt’, ‘jle’, and similar instructions to compare signed values. To accommodate this, I replaced the ‘norz’ flag with an overflow flag, enabling the correct implementation of these instructions. The image below illustrates how I integrated the overflow flag into the MiMo model.



I also renamed the ‘norz’ flag to ‘v’ and addressed a minor issue in the ‘micro_assembler.pl’ file. After making the necessary fixes, I packaged it into an executable file named micro_assembler_v2. The issue involved the assembler incorrectly translating the ‘if x then y’ partial jumps. The solution was straightforward: I simply needed to rename the variable in line 169 from $jump to $tjump.

Instruction explanations

Now, I will explain some of the more interesting instruction implementations:

jgt Rs,Rt,immed ; if Rs > Rt, PC <- immed else PC <- PC + 2

addrsel=pc imload=1
aluop=sub op2sel=treg, if z then pcincr
aluop=sub op2sel=treg, if n then jvset else jvnset

jvset: aluop=sub op2sel=treg, if v then jump else pcincr
jvnset: aluop=sub op2sel=treg, if v then pcincr else jump

This is the only conditional jump/branch instruction I will explain, as all of them are quite similar. The gt (greater than) condition is met when Z == 0 and N == V.

The instruction begins by loading the immediate value into the immediate register. It then subtracts the two registers and checks the Z flag. If the Z flag is set, the condition was not met, so the instruction skips to the next macroinstruction. If the Z flag is not set, it proceeds to the next microinstruction, which checks if the N flag is set. If the N flag is set, it jumps to the jvset (Jump if oVerflow SET) instruction, which checks the V flag. If the V flag is set, it proceeds to the jump instruction. If the N flag is not set, it jumps to the jvnset (Jump if oVerflow Not SET) instruction, which also proceeds to the jump instruction if the V flag is not set.

clr Rs ; Rs <- 0   PC <- PC + 1

swrite=1 aluop=mul op2sel=const0 regsrc=aluout, goto fetch

The clr (clear) instruction may seem simple to implement at first glance, but the MiMo model doesn't provide a straightforward way to directly store a 0 value into a register. To work around this limitation, I implemented the instruction by multiplying the value in the register by 0 and then storing the result back into the same register. This effectively clears the register by setting its value to 0.

neg Rs ; Rs <- -Rs   PC <- PC + 1

swrite=1 aluop=not regsrc=aluout
swrite=1 aluop=add op2sel=const1 regsrc=aluout, goto fetch

To implement the neg (negate) instruction, I first used the NOT gate to invert all the bits of the value in the register, effectively reversing the bits and storing the result back into the same register. After that, I added 1 to the register, which completed the process of calculating the two's complement of the original value, effectively negating it.