**F1 Starting Lights in RISCV – Testing in GTKWave**

When the program is run, all registers are equal to 0. The first 4 instructions are run, and they perform their roles, loading 20 into x10(limit for the counter), loading 0 into x12(start of the counter), loading FF into x13(maximum lights for f1), loading 0 into x20(f1 light register), and branching when x31 is not 0 (trigger is when x31 is set to 0 using vBuddy, but for GTKWave the trigger is as soon as the program starts).

Graphical user interface

Description automatically generated

The PC has been set to 0 rather than the memory map recommendation of BFC00000 for the sake of simplicity in this test.

Graphical user interface

Description automatically generated

Scrolling along to the 6th instruction JAL x1, count – we can see that register x20 has the value 0, which corresponds to no lights on. We can also see that PC has jumped to address 54, this is the JAL instruction being executed as we jump to the first instance of using the counter. The next PC instruction 18, is stored in x1 for when we finish the count and return. The next instruction 00160613 starts the counter and here we can see the register x12 increments by one, whilst the PC address jumps between 24 and 28, signalling that the next instruction BNE also works.

Graphical user interface

Description automatically generated

The program keeps looping until the count register x12 is equal to x10 which is hex 14 in this case. The program is no longer bound by the branch and instead continues to the next 2 instructions, resetting the counter and returning to the next instruction stored in x1, 18. This instruction is the SLL one, which shifts by one to the left and adds 1, making x20 equal to 1, which means the first f1 light is now on. But since this isn’t equal to FF(11111111) or 8 lights being on, we branch back to the JAL at address 14 and begin the 1 second count again

Graphical user interface

Description automatically generated

1 second has been counted, counter resets, jump back to address 18, another shift and add 1, 2nd f1 light turns on (x20=11), branch back to jump and next address (18) stored in x1.

Graphical user interface

Description automatically generated

1 second has been counted, counter resets, jump back to shift left instruction, 3rd f1 light turns on (x20=111), branch back to jump and next address (18) stored in x1.

Graphical user interface

Description automatically generated

1 second has been counted, counter resets, jump back to shift left instruction, 4th f1 light turns on (x20=1111), branch back to jump and next address (18) stored in x1.

Graphical user interface

Description automatically generated

1 second has been counted, counter resets, jump back to shift left instruction, 5th f1 light turns on (x20=11111), branch back to jump and next address (18) stored in x1.

Graphical user interface

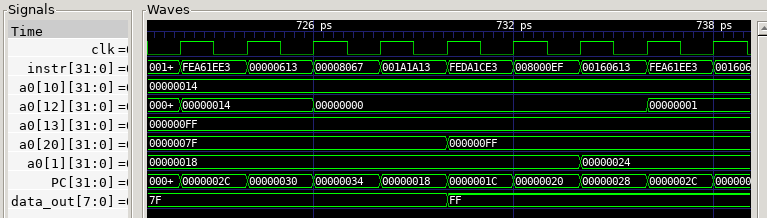
Description automatically generated

1 second has been counted, counter resets, jump back to shift left instruction, 6th f1 light turns on (x20=111111), branch back to jump and next address (18) stored in x1.

Graphical user interface

Description automatically generated

1 second has been counted, counter resets, jump back to shift left instruction, 7th f1 light turns on (x20=1111111), branch back to jump and next address (18) stored in x1.



1 second has been counted, counter resets, jump back to shift left instruction, 8th f1 light turns on (x20=11111111), branch back to jump and next address (18) stored in x1. Also in this case, x20 is now equal to x13 (FF) so we don’t branch again and instead move to the next instruction, which counts 1 more second before terminating the lights. This time, the JAL stores the address 24 in x1 rather than 18 it has done the last 8 times.

Graphical user interface

Description automatically generated

Here, we reset the counter, return back to the instruction at address 24 which is JALR x0, x0, jump back to instruction 0, and reset data out or x20 so that the lights turn off.

From this test of one f1 starting light sequence, you can see that the lights do cycle properly by using a logical shift and adder, it also waits the required amount between each light using a counter, though it is hard to simulate how long in GTKWave. We’ve used the required JAL instruction as defined in the project spec.

A screenshot of a computer

Description automatically generated with low confidenceText

Description automatically generatedAssembly code Machine code