Lab#3:A Simple Processor

Figure 1 shows a digital system that contains a number of nine-bit registers, a multiplexer, an adder/subtractor unit, and a control unit (finite state machine). Data is input to this system via the nine-bit *DIN* input. This data can be loaded through the nine-bit wide multiplexer into the various registers, such as *R*0*,...,R*7 and *A*. The multiplexer also allows data to be transferred from one register to another. The multiplexer’s output wires are called a *bus* in the figure because this term is often used for wiring that allows data to be transferred from one location in a system to another.

Addition or subtraction of signed numbers is performed by using the multiplexer to first place one nine-bit number onto the bus wires and loading this number into register *A*. Once this is done, a second nine-bit number is placed onto the bus, the adder/subtractor unit performs the required operation, and the result is loaded into register *G*. The data in *G* can then be transferred to one of the other registers as required.

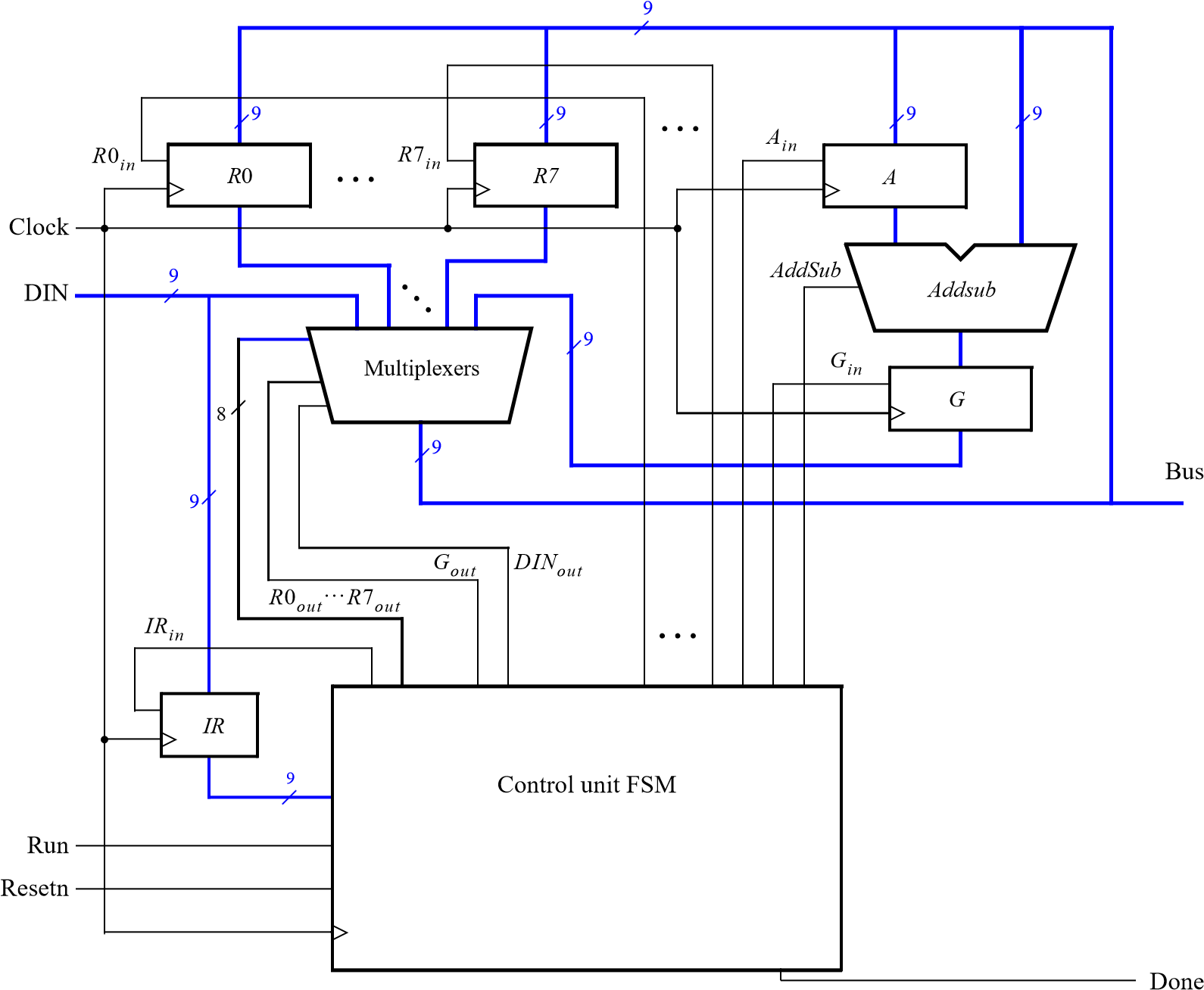


Figure 1: A digital system.

The system can perform different operations in each clock cycle, as governed by the *control unit*. This unit determines when particular data is placed onto the bus wires and it controls which of the registers is to be loaded with this data. For example, if the control unit asserts the signals *R*0*out* and *Ain*, then the multiplexer will place the contents of register *R*0 onto the bus and this data will be loaded on the next active clock edge into register *A*.

A system like the one in Figure 1 is often called a *processor*. It executes operations specified in the form of *instructions*. Table 1 lists the instructions that the processor has to support for this exercise. The left column shows the name of an instruction and its operands. The meaning of the syntax R*x* ← [R*y*] is that the contents of register R*y* are loaded into register R*x*. The mv (move) instruction allows data to be copied from one register to another. For the mvi (move immediate) instruction the expression R*x* ← D indicates that the nine-bit constant D is loaded into register R*x*.

|  |  |
| --- | --- |
| Operation | Function performed |
| mv *Rx*,*Ry* | *Rx* ← [*Ry*] |
| mvi *Rx*,#*D* | *Rx* ← *D* |
| add *Rx,Ry* | *Rx* ← [*Rx*] + [*Ry*] |
| sub *Rx,Ry* | *Rx* ← [*Rx*] − [*Ry*] |

Table 1: Instructions performed in the processor.

Each instruction can be encoded using the nine-bit format *IIIXXXYYY*, where *III* specifies the instruction, *XXX* gives the R*x* register, and *YYY* gives the R*y* register. Although only two bits are needed to encode our four instructions, we are using three bits because other instructions will be added to the processor in later parts of the exercise. Assume that *III* = 000 for the mv instruction, 001 for movi, 010 for add, and 011 for sub. Instructions are loaded from the the external input *DIN*, and stored into the *IR* register, using the connection indicated in Figure 1. For the mvi instruction the *YYY* field has no meaning, and the immediate data #*D* has to be supplied on the *DIN* input in the clock cycle after the mvi instruction word is stored into *IR*.

Some instructions, such as an addition or subtraction, take more than one clock cycle to complete, because multiple transfers have to be performed across the bus. The finite state machine in the control unit “steps through” such instructions, asserting the control signals needed in successive clock cycles until the instruction has completed. The processor starts executing the instruction on the *DIN* input when the *Run* signal is asserted and the processor asserts the *Done* output when the instruction is finished. Table 2 indicates the control signals that can be asserted in each time step to implement the instructions in Table 1. Note that the only control signal asserted in time step 0 is *IRin*, so this time step is not shown in the table.

*T*1 *T*2 *T*3

|  |  |  |
| --- | --- | --- |
| *RYout*, *RXin*,  *Done* |  |  |
| *DINout*, *RXin*,  *Done* |  |  |
| *RXout*, *Ain* | *RYout*, *Gin* | *Gout*, *RXin*,  *Done* |
| *RXout*, *Ain* | *RYout*, *Gin*, | *Gout*, *RXin*, |
|  | *AddSub* | *Done* |

(mv): *I*0

(mvi): *I*1 (add): *I*2

(sub): *I*3

Table 2: Control signals asserted in each instruction/time step.

# Execution & Report

Design and implement the processor shown in Figure 1 using Verilog code as follows:

1. Create a new Quartus project for this exercise.
2. Generate the required Verilog file, include it in your project, and compile the circuit. A suggested skeleton of the Verilog code is attached together with two subcircuit modules that can be used in this code .
3. Use functional simulation to verify that your code is correct. An example of the output produced by a functional simulation for a correctly-designed circuit is attached. It shows the value (010)8 being loaded into *IR* from *DIN* at time 30 ns. This pattern represents the instruction mvi R0,#D, where the value *D* = 5 is loaded into *R*0 on the clock edge at 50 ns. The simulation then shows the instruction mv R1,R0 at 90 ns, add R0,R1 at 110 ns, and sub R0,R0 at 190 ns. Note that the simulation output shows *DIN* and *IR* in octal, and it shows the contents of other registers in hexadecimal.
4. Now, create another Quartus project which will be used for implementation of the circuit on your Intel FPGA DE-series board. This project should consist of a top-level module that contains the appropriate input and output ports for the DE-series board. Instantiate your processor in this top-level module. Use switches *SW*8−0 to drive the *DIN* input port of the processor and use switch *SW*9 to drive the *Run* input. Also, use pushbutton *KEY*0 for *Resetn* and *KEY*1 for *Clock*. Connect the processor bus wires to *LEDR*8−0 and connect the *Done* signal to *LEDR*9.
5. Perform full functional simlution –include in your report simulation vectors and results that test all processor Hardware
6. Perform full timing simlution –derive the maximum processor frequency
7. Add to your project the necessary pin assignments for your board. Compile the circuit and download it into the FPGA chip.
8. Test the functionality of your circuit by toggling the switches and observing the LEDs. Since the processor’s clock input is controlled by a pushbutton switch, it is possible to step through the execution of instructions and observe the behavior of the circuit.
9. In addition to the above four instructions implement one of the following instructions:

**The teaching team will instruct each group which instruction out of the 3 to implement**

1. "ones" - Calculate the number of bits which have the value '1' in the register and insert the value to the second register:

ones, Rx, Ry -> Ry will show the number of bits with a value of one found in Rx.

For example if Rx = 000110011, after the function will be performed Ry will show  000000100.

Note that an implementation with a for loop and an explanation on what the for loop does will provide extra points.

1. "swap" swap between the values stored in the two input registers, in the implementation you can only use basic logic operations (i.e addition, subtraction, shift and xor).

please provide a diagram with the logic gates used (no points will be given without the diagram).

swap,Rx, Ry -> Rx = Ry, Ry = Rx.

1. "specialMult" multiply a register by 3.5 without explicitly using a multiplier and insert the value to the second register.

specialMult, Rx, Ry -> Ry = Rx\*3.5.

You can assume that this function should only work on multiples of 10, i.e Rx will always be a multiple of 10.