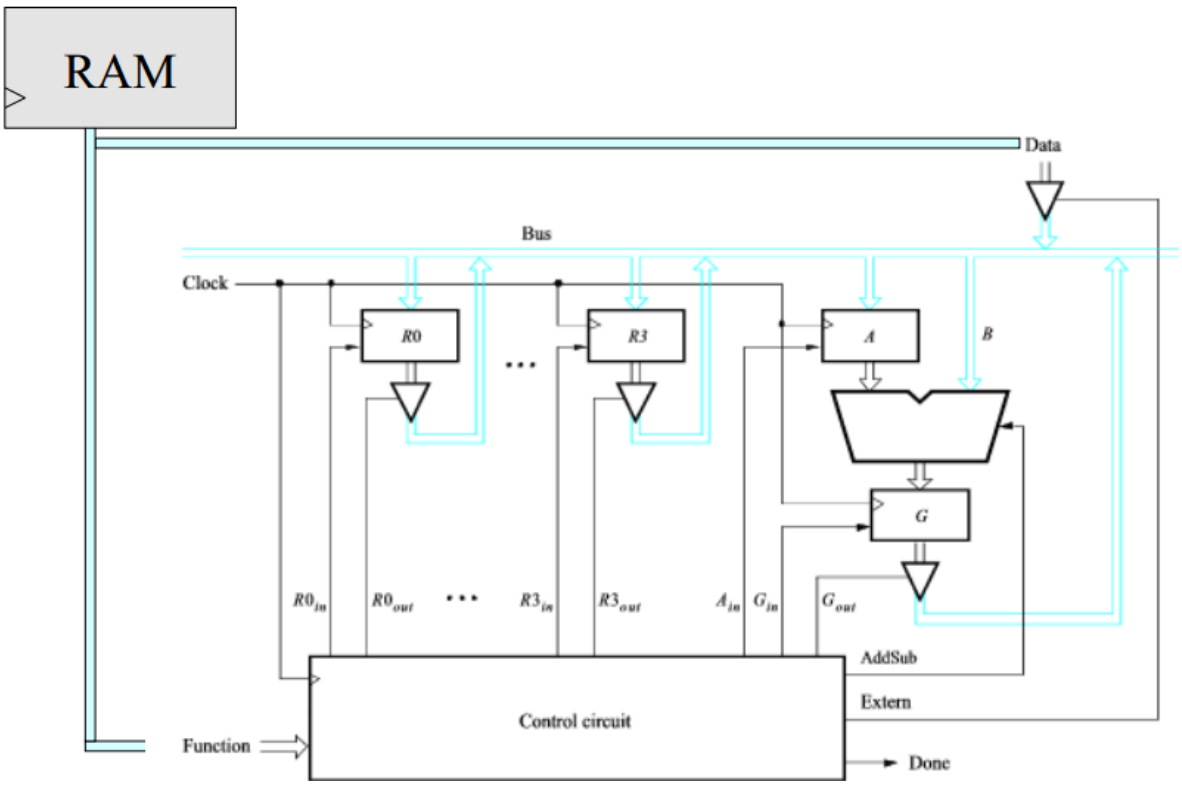
Project Report: Basic 16 Bit Processor

Introduction:

This final project sees us implementing a 16 bit processor capable of implementing a small range of functions that involve at its basis; loading, moving, adding and applying exclusive OR to any 16 bit data value that has uploaded to the registers. To do this required implementing several block components that would function to help maintain and control data and instruction flow, including registers, tri-state buffers, adders, multiplexers and a control unit. To setup the incoming instructions and data, we also implemented a RAM block which stores pre-written functions and data that are uploaded to the processor sequentially via their address. All implementation was done using VHDL and simulated in Quartus II 13.0sp1 and on the Altera DE2 FPGA.

Part I: Block Diagram

For the implementation, the processor requires registers to hold data, tri-state buffers to control data flow to the BUS, a 16-bit Adder and a XOR block for the ADD and XOR operations, a RAM block to hold the operations and data and a control unit that is responsible for coordinating these operations. The general arrangement of these blocks can be given as so:



Operational functions such as LOAD, MOV etc… are loaded onto the control unit which subsequently moves to coordinate the flow of data throughout the processor.

The instruction LOAD followed by the register specification will call for the unit to assert Extern to permit data from the RAM to be uploaded to the bus. The chosen register is then enabled for reading in whatever data is currently on the BUS.

For MOV Rx Ry, similar to LOAD, data is loaded onto the BUS which is read into by Rx but in this instance, the data is taken from Ry instead of RAM.

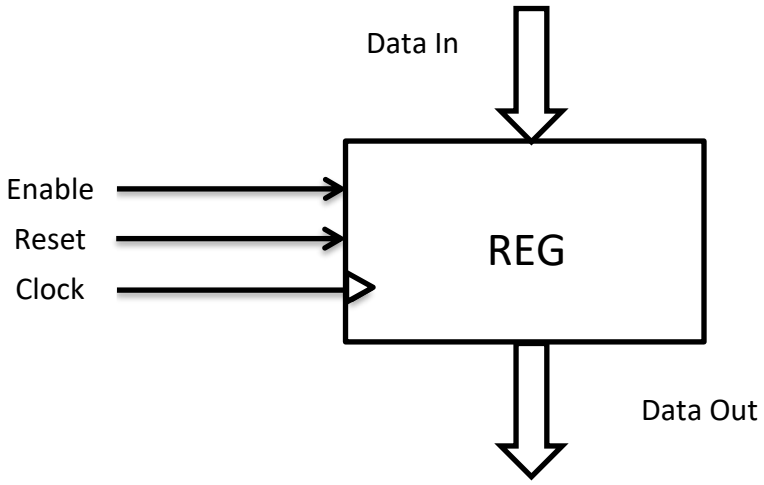
With the ADD Rx Ry, since multiple segments of data from several sources must be employed, this instruction requires a multi-clock implementation. Firstly, data is uploaded to BUS from Rx to the Accumulator (A) which will serve as the first operand. Next, data from Ry is uploaded and sent directly through to the adder/xor block as the second operand, outputting the result to storage register G. Finally, the result is then offloaded from G back into register Rx via the BUS.

Finally the XOR instruction, being identical to the ADD operator in terms of data coordination follows along the same lines as the above, with the exception that the control unit must assert the XOR signal to notify the add/xor block of the intended operation.

Part II: Components

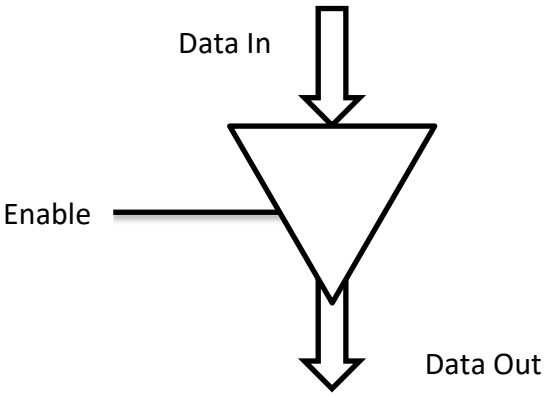
Registers

The registers in the processors serve the important function of maintaining any data (16 bit) that is uploaded into its memory when enabled by the control unit and outputting the data which is kept in check with a tri-state buffer. For the purpose of being able to update and/or store any value, this component will only run during rising clock edges. Data will only be stored when the register receives an enable signal from the control unit on this edge and will set all to 0 when Reset is asserted. For this implementation, there are 3 registers; R0, R1 and R2 and 2 additional registers; the accumulator and the add/xor output.



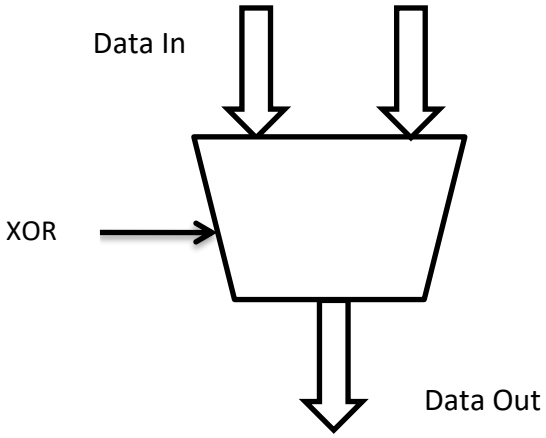
Tri –State Buffers

These buffers will allow the control unit to prevent or allow data to be passed onto the Bus which is important when preventing contention where multiple sources attempt to send data through the same route, which will confuse the signal. This segment also receives an enable signal from the control circuit which will determine whether the DATA IN will be sent to DATA OUT, otherwise high impedance (Z) will be set. Unlike the registers, this will run independently of the clock. Each register in the processor (other than the accumulator), will have an associated state buffer run by enable ($R_{out(0-3)}$, G_{out}) and any data that is loaded onto the processor will be maintained by control signal Extern.

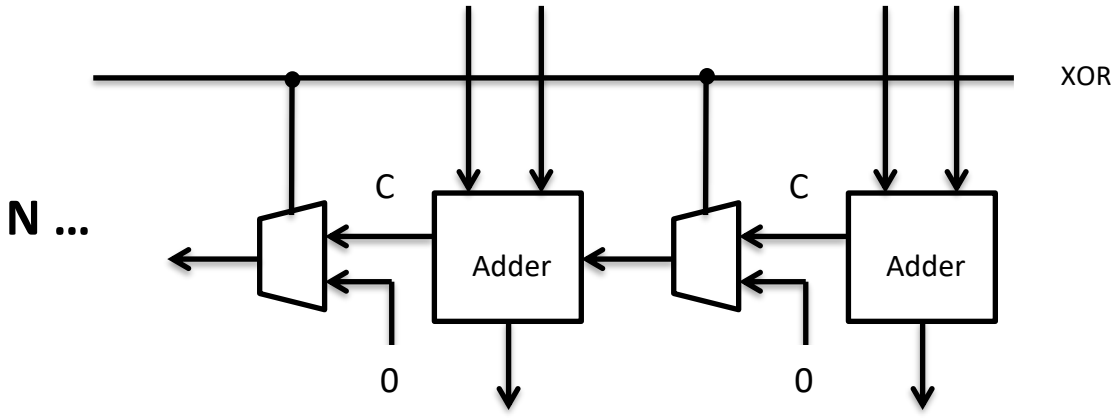


Adder/Xor Block

This block is responsible for performing a 16 bit addition or XOR taking DATA from the accumulator register and the BUS and outputting the result to the output register G. The requirement of the accumulator and the output registers is necessary to maintain only a single flow of data on the bus to prevent two or more signals from interfering which will distort the data. To differentiate from an ADD or XOR operation, a signal (XOR) from the control unit is present, with 0 selecting ADD and 1 being XOR.

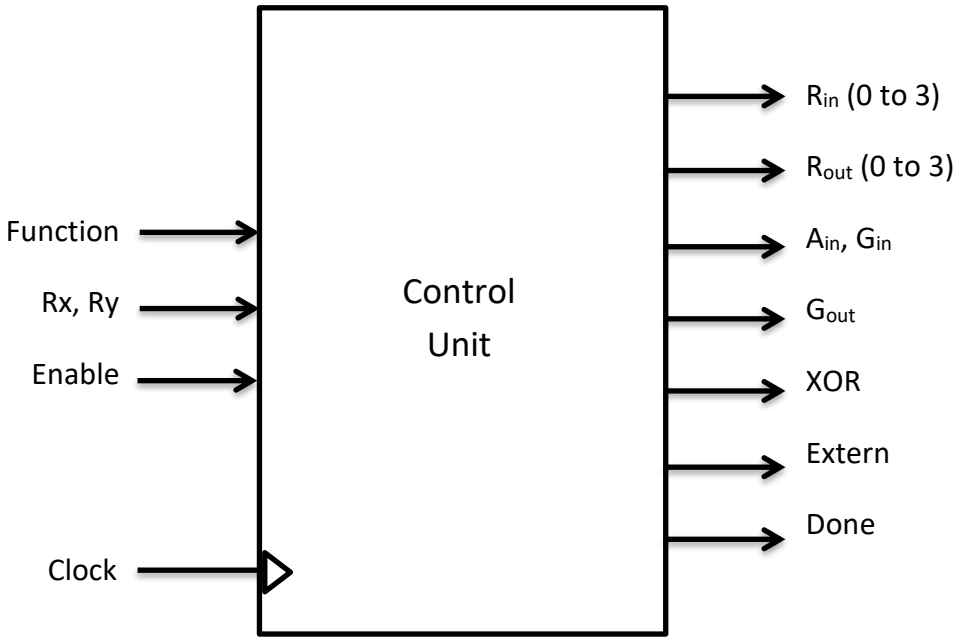


This component combines both the functionality of the Adder and the XOR operators by manipulating the carries between each adder using a multiplexer that employs XOR to select from either the carry (C) from the previous adder during ADD or 0 during XOR. This block operates independently from clock.



Control Unit

While the above components are important in maintaining operations, the control unit in the processor is vital in the coordination of these components, enabling and disabling specific elements in order to allow for data to be properly moved to where it needs to be sent and preventing excess information from overloading the BUS within the same clock cycle when implementing any operation.



To operate, this unit requires 4 sets of inputs, with the Function and Rx, Ry being the primary signals that will be decoded (12-bit):

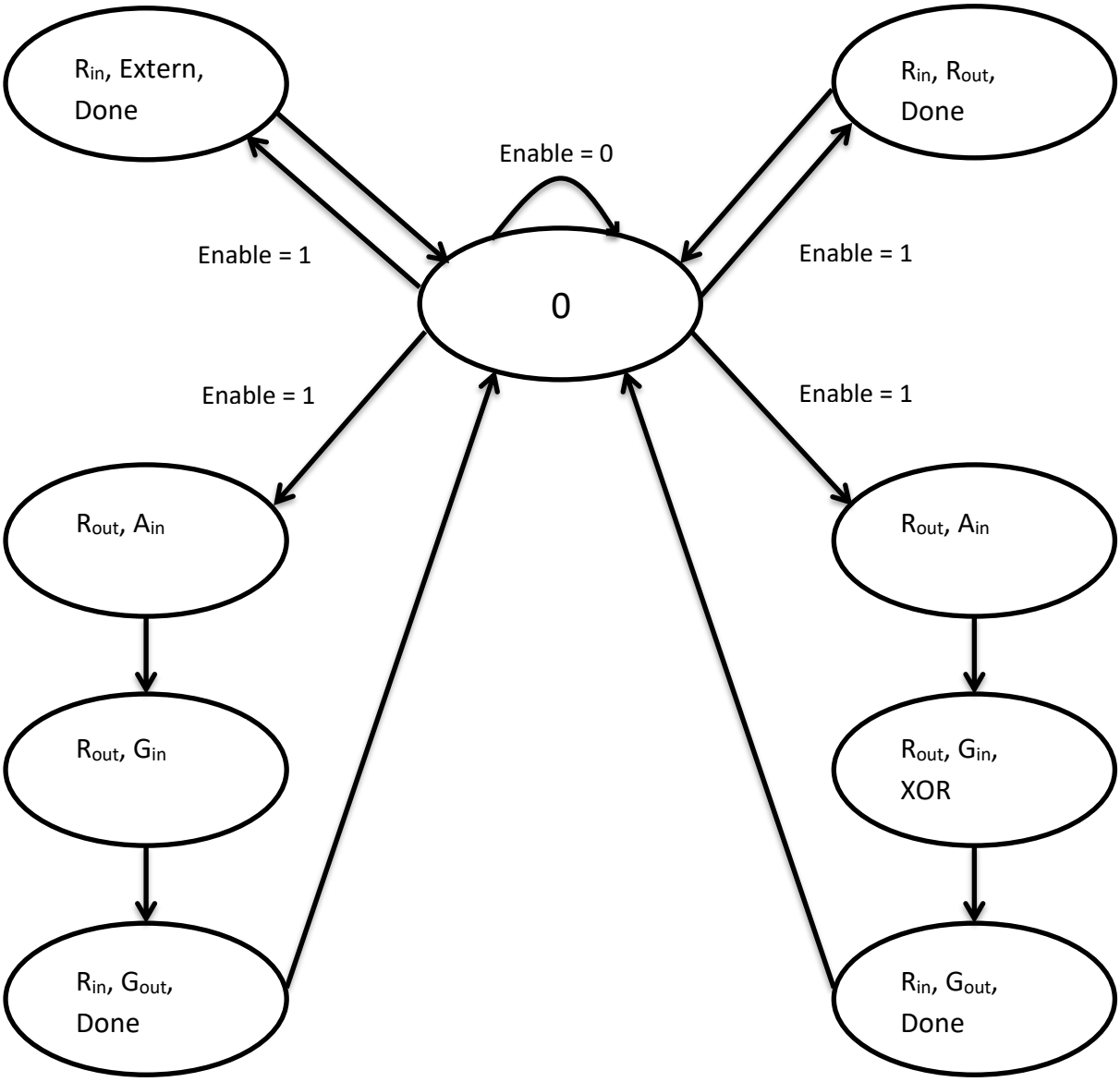
- **Function**: This is a 4 bit input that specifies what particular operation the control circuit will perform (LOAD, MOV, ADD or XOR)
- **Rx and Ry**: The specified register/s in which the operation will be performed on (each 4 bit)
- **Enable**: A signal that controls whether the control circuit will operate
- **Clock**: A timing input to allow for a multiple sets of outputs to be sent over multiple cycles, particularly important for multi-cycle operations such as ADD or XOR

From this, a 12-bit output is generated:

- **R_{in}(0 to 3)**: A logic vector responsible for permitting the registers (R0, R1 and/or R2 respectively) to read in from the BUS.
- **R_{out}(0 to 3)**: A logic vector controlling the tri-state buffers that regulate the register (R0, R1 and/or R2 respectively) outputs to the BUS.
- **A_{in}, G_{in}**: Logic outputs that determine whether the accumulator and the write output registers are allowed to read in from the BUS and the Adder/Xor block respectively.

- **Gout**: An enable for the write output tri-state, maintaining control over register G writing to the BUS
- **Xor**: A load signal to a multiplexer in the Add/Xor block that selects whether the ADD or XOR operation will be performed
- **Extern**: An enable signal for the tri-state that controls when any DATA loaded into the processor will be allowed into the BUS
- **Done**: A signal which is asserted whenever the processor has finished performing an instruction that is sent to a program counter

Hence, for the control circuit to an instruction, it requires the Function input that identifies the instruction along with the register/s selected for the operation. From here, the unit asserts the required outputs to 1 and disable unneeded components using 0. The finite state diagram description can be seen below where for simplicity; any arguments within the state are asserted to 1 while those not present will be 0:



With regard to the registers, decoding from X_{in} or Y_{in} to X_D or Y_D will be performed as so:

Register	R(x/y)	R_{in} and R_{out}
R0	0000	100
R1	0001	010
R2	0010	001
-	-	000

(Note: '-' indicates that input does not matter)

For the decoding, each instruction will produce the following signals in accordance with the block diagram and the state machine:

- **LOAD Rx**

Function	Rx	Ry	T	R_{in}	R_{out}	A_{in}	G_{in}	G_{out}	XOR	Extern	Done
0001	X_{in}	-	1	X_D	000	0	0	0	0	1	1

As shown, the control unit will identify the function input and will decode Rx to load in Data into the specified register. The rest of the LOAD decoding will be set to: **X_D000000011**, only asserting Extern for the data uploading and 'Done' to notify our program counter.

- **MOVE Rx Ry**

Function	Rx	Ry	T	R_{in}	R_{out}	A_{in}	G_{in}	G_{out}	XOR	Extern	Done
0010	X_{in}	Y_{in}	1	X_D	Y_D	0	0	0	0	0	1

For the MOV function, the circuit will simply permit Ry to write to BUS and Rx to read in from the BUS on the same clock cycle so data flows directly from one register to the other. As so, we are not required to assert any signal other than 'Done': **X_DY_D000001**.

- **ADD Rx Ry**

Function	Rx	Ry	T	R_{in}	R_{out}	A_{in}	G_{in}	G_{out}	XOR	Extern	Done
0011	X_{in}	Y_{in}	1	000	X_D	1	0	0	0	0	0
			2	000	Y_D	0	1	0	0	0	0
			3	X_D	000	0	0	1	0	0	1

Unlike the previous instruction, ADD requires multiple clock cycles to complete, as the BUS is only capable to carrying data from a single source. In the first cycle, Rx loads its data into the BUS and fed into the accumulator which is automatically read by the addXor block: **000X_D100000**. In the next cycle, Ry uploads its data to the BUS which is sent directly to the addXor block and the result is written to G: **000Y_D010000**. Finally, G writes out to BUS which in turn is read into Rx: **X_D000001001**.

- **XOR Rx Ry**

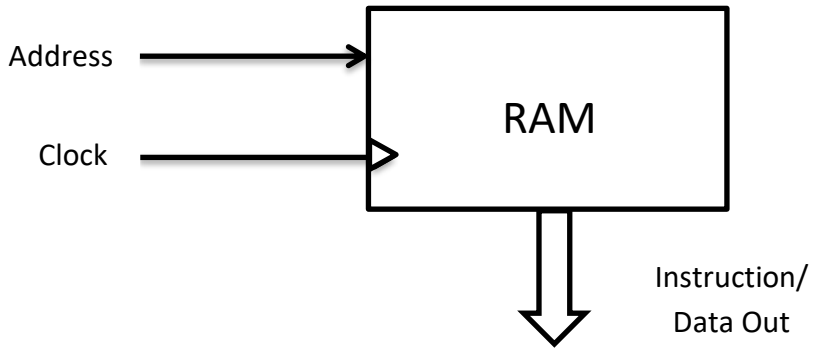
Function	Rx	Ry	T	R_{in}	R_{out}	A_{in}	G_{in}	G_{out}	XOR	Extern	Done
0100	X_{in}	Y_{in}	1	000	X_D	1	0	0	0	0	0
			2	000	Y_D	0	1	0	1	0	0

	3	X_D	000	0	0	1	0	0	1

The XOR instruction follows almost precisely like ADD with the exception in clock cycle 2 where the control unit must also assert XOR to 1 to set exclusive OR functionality: **000Y_D010100.**

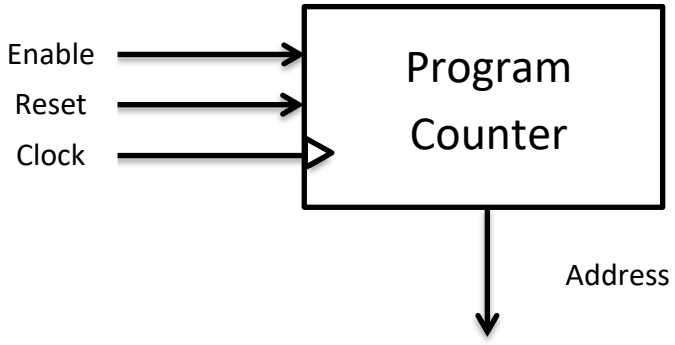
RAM

While a processor utilising only the above components would be capable of implementing all the required instructions, a processor by design should also be developed capable of reading instructions and data from memory. In this case, the information is stored within the RAM block, held sequentially in their address (4-bit), with new instructions or data uploaded (both 16-bit) to the processor with each assertion of 'Done'. This 'Done' signal serves as an indicator that the processor is ready for the next command by incrementing a program counter that operates to designate the address in which the next instruction or data line is stored. While typical processors are also capable of writing back into memory, for the implementation of our project, we are employing a solely read out RAM block.



Program Counter

Since the instructions and data in the RAM are to be implemented in the processor sequentially according to their address, it would be most logical to set up a counter that held the address value and would be used by the processor to retrieve the instruction. Once the processor has finished performing the instruction, the control unit then notifies (Done) the program counter which will increment the address, which will subsequently point to the new line of memory that contains the processor's next instruction. In addition, a synchronous reset is embedded that once triggered, will reset the counter to 0 to start the program again.



Part III: Testing

Testing was undertaken on each component individually using the Quartus II simulator:

- **Registers**: Data was inputted on both enable = 1 and 0 and observed to see if data was stored
- **Tri-state buffers**: Data was inputted on both enable = 1 and 0 and observed to check whether data was passed
- **Add/Xor**: Ran data through the block for each operation to confirm correct functionality
- **Control Unit**: Different instruction functions and registers were set to the unit and the output was examined for consistency with the decoding results as defined in the control unit component tables
- **RAM**: The program counter was incremented and instruction/data out was observed and checked with their address

Further testing was performed on the Altera DE2 board to ensure processor as a whole was operation. For this, RAM was disabled and inputs were set manually by SW (17 DOWNTO 0) and KEY (0):

- SW(17): Reset
- SW(16): Enable
- SW(15 DOWNTO 13): Function (largest bit taken as 0)
- SW(12 DOWNTO 11): Rx (larger bits taken as 0)
- SW(10 DOWNTO 9): Ry (larger bits taken as 0)
- SW(8 DOWNTO 0): Data (due to switch limitation, all larger bits taken as 0)
- KEY(0): Clock

The results were represented on LEDR (17 DOWNTO 0) which was used to read component outputs during the tests e.g. register contents, control unit outputs, BUS data etc…

For the combined processor and RAM, the Quartus II waveform simulator was employed using only 3 inputs; Reset, Enable and Clock while the instructions stored in RAM were to be sent to the processor and performed sequentially using our sample program:

```
--enter your own program here
result(0) := "0001000100000000"; --LOAD R1
result(1) := "0000000000000011"; --DATA
result(2) := "0001000000000000"; --LOAD R0
result(3) := "0000000000000001"; --DATA
result(4) := "0011000000010000"; --ADD R0,R1
result(5) := "0010001000000000"; --MOV R2,R0
result(6) := "0001000000000000"; --LOAD R0
result(7) := "0000000000001011"; --DATA
result(8) := "0100000000010000"; --XOR R0,R1
```

Part IV: Results

The results of the component testing showed consistency with expected results.

Part V: Conclusion

This project has evidently served as an assessment of our ability to apply concepts and techniques developed throughout lectures and tutorials (flip-flops, adders, counters etc…) and amalgamating this knowledge to aid in building a simple 16-bit processor. The understanding of the many components that constitute a processor presents us with a further understanding of the importance of coordination that directs data and information flow by the control circuit that interprets instructions to perform specific tasks. Furthermore, the use of instruction decoding in the unit serves as a basis in comprehending how machine language is formulated. Another important aspect evident within the development of the processor is the idea of minimising the number of clock cycles required for receiving and performing instructions which aids in design efficiency. With these considerations along with many others, more powerful and efficient processors can be developed capable of execution greater computational-intensive tasks.