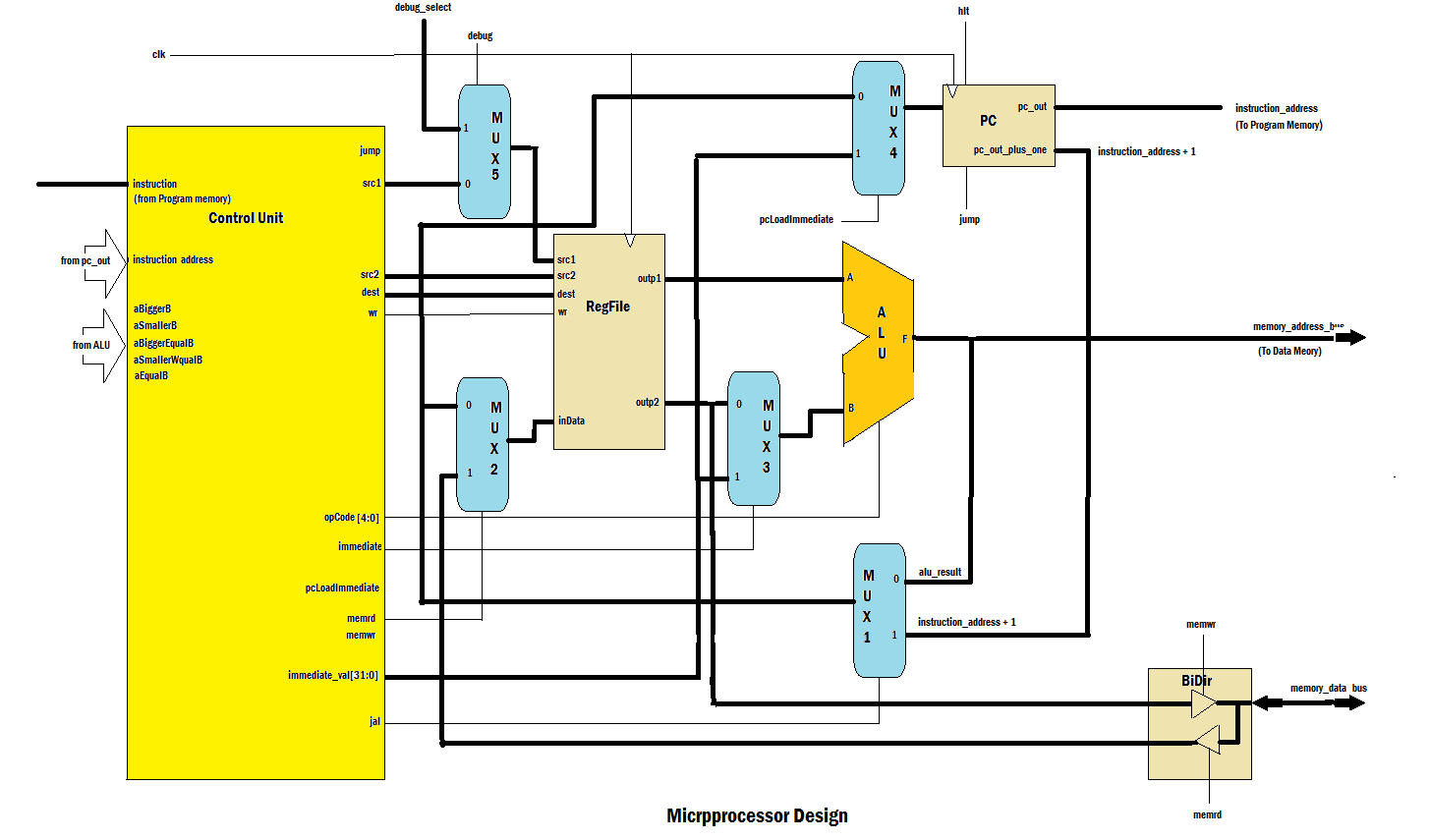
The Lab-Processor

Introduction:

This is a microprocessor designed by me as an attempt to learn system Verilog and hardware description language in general.

The processor design is inspired from experience and self-taught practices to enhance my hardware understanding concepts. It was fully utilized for a sub project to use it as the core processor for a full microcontroller solution with 2 UARTs, 2 Timers, 2 I/O 8bits bidirectional ports and a MAC module for Ethernet connectivity.

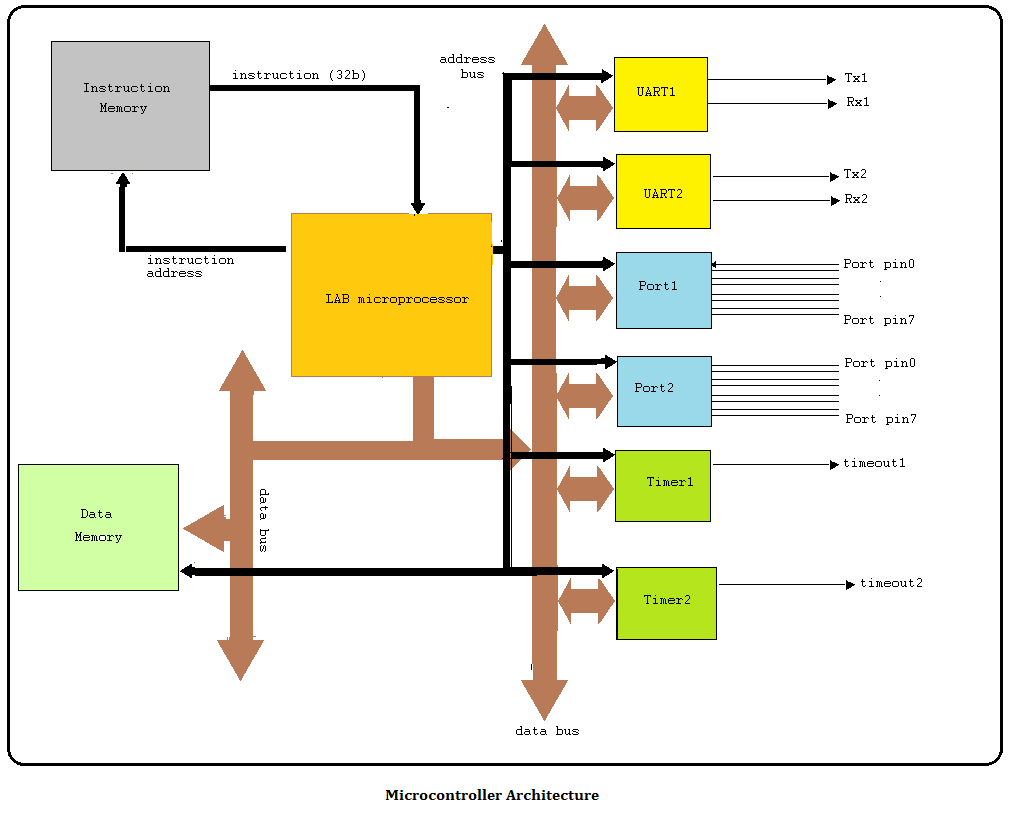


The code compiler for the microcontroller/microprocessor has been written for a Simple BASIC language compiler (SBLC) for simplicity (C compiler can be similarly easily written).

The processor was designed; verified and simulated using Modelsim free edition and a compiler SBLC (Simple Basic Language compiler) and assembler has been developed for use with the processor.

The compiler produces assembly instructions from the basic code written and then get assembled with the assembler (note that the assembler can be invoked individually to write pure assembly code, or automatically after compiling a basic program).

See attached sample program and the compiled/assembled versions and the screen shot from Modelsim that verifies the proper operation of the compiled program.

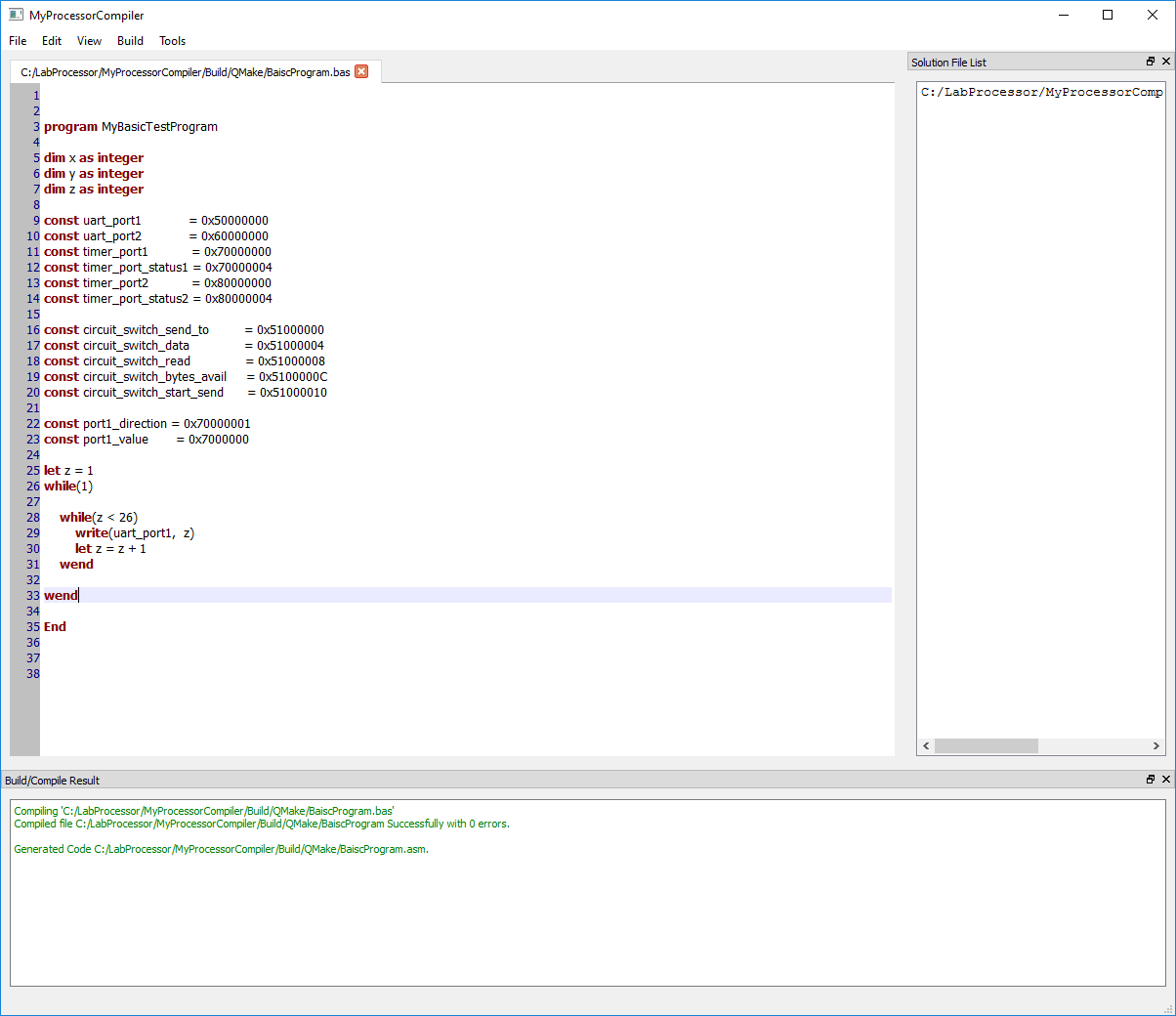


The UART1 and UART2 has their own address map that can write to their TX FIFOs to start sending straight away, the RX part contains similar FIFOs for receiving. So writing to the address 0x50000000 a value of 0x01 will cause the UART1 to start sending the value 0x01 on the TX1 wire, if a similar UART RX pin is attached to the TX1 pin, its FIFO will contain the value 0x01 after completing the send operation from the TX1 (verified using the Modelsim by wiring the TX1 to RX2).

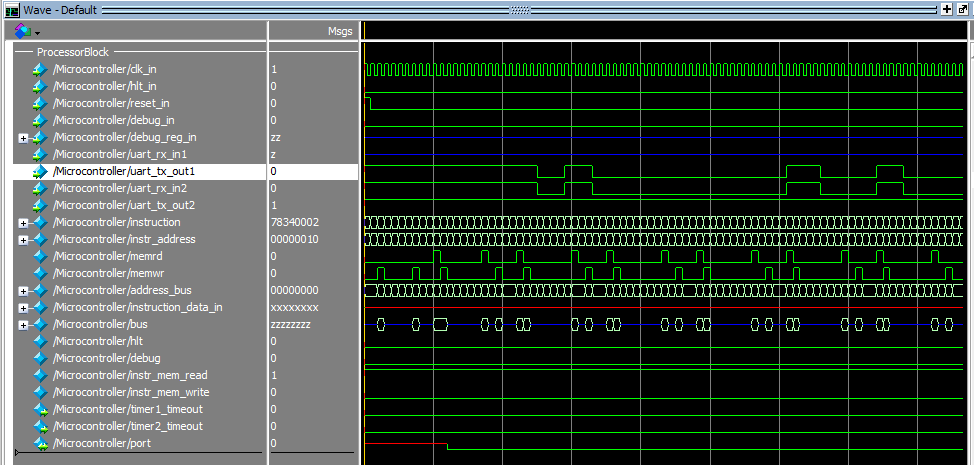
The values in the UART1 FIFO can be interrogated using the read instruction from the address 0x50000100 which gives the counter values of the last still to be read received bytes. Once we read the bytes with a read operation/s from 0x50000000 locations, the counter values decrement until we read all the received bytes.

The FIFOs for both the TX and the RX is 256 bytes deep, which means that the UART1 and UART2 can keep sending/receiving bytes without the processor interaction until its FIFOs gets full.

The general memory reads and writes are interpreted by the compiler writes/reads to data memory using write/read commands that takes the address as the first argument and the values as the second.



In code snippet shown, it can be seen in the code that we are writing to the uartPort1 (which is a defined constant in the code which have the value of 0x50000000)



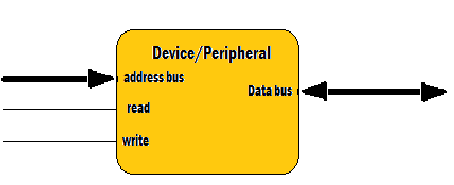
The Processor module:

The processor module can be used as a library component to be hooked in any project. The architecture is flexible to add any peripheral to be designed under certain conditions.

It can be also included with its peripheral components as a general purpose microcontroller with as many timers, UARTs and other designed peripherals as required.

In order to maintain the flexibility, the processor should be hooked via a 32 bit data bus that is bidirectional so that any device/peripheral needs to be attached to the processor will resemble the same architecture interface (data bus (bidirectional), address bus, a read and a write pin).

The peripheral should have an address map to access it’s internal ram and/or registers for the processor to use.

****

A full duplex UART (with internal RX and TX FIFOs) has been designed to be attached to the processor, note that we can attach as many UART instances as we can, we need to specify the address map for the RX/TX fifo buffers as parameters to the modules when instantiated.

Also a Timer module has been designed and simulated having the same interface. Similarly we need to specify the address map for the Timer to write timeout values and so on.

The Processor instruction set (6 bits):

|  |  |  |  |
| --- | --- | --- | --- |
| **opcode** | **Description** | **Code(Binary)** | **Example** |
| nop | No operation (does nothing) | 000000 | nop |
| add | Adding 2 operands | 000001 | add $r3, $r1, $r2 |
| addi | Adding 2 operands the second is literal | 010001 | addi $r3, 0x1234 |
| sub | subtracting 2 operands | 000010 | sub $r3, $r1, $r2 |
| subi | subtracting 2 operands the second is literal | 010010 | subi $r3, 0x1234 |
| and | anding 2 operands | 000011 | and $r3, $r1, $r2 |
| andi | anding 2 operands the second is literal | 010011 | andi $r3, 0x1234 |
| or | oring 2 operands | 000100 | or $r3, $r1, $r2 |
| ori | oring 2 operands the second is literal | 010100 | ori $r3, 0x1234 |
| xor | xoring 2 operands | 000101 | xor $r3, $r1, $r2 |
| xori | xoring 2 operands the second is literal | 010101 | xori $r3, 0x1234 |
| mul | multiplying 2 operands | 000111 | mul $r3, $r1, $r2 |
| muli | multiplying 2 operands the second is literal | 010111 | muli $r3, 0x1234 |
| div | dividing 2 operands | 001000 | div $r3, $r1, $r2 |
| divi | dividing 2 operands the second is literal | 011000 | divi $r3, 0x1234 |
| shl | Shift left operand 1 by the value in the second register |  | shl $r3, $r1 |
| shli | Shift left operand 1 by the literal value in |  | Shli $r3, 3 (r3<<3) |
| shr | Shift right operand 1 by the value in the second register |  | shr $r3, $r1 |
| shri | Shift right operand 1 by the literal value in |  | Shri $r3, 3 (r3>>3) |
| lod | Load instruction from memory |  | lod $r3, (2) $r1 |
| str | Store instruction to memory |  | str $r3, (2) $r1 |
| jal | Jump and link |  |  |
| ret | Return from call |  |  |
| jmp | Jump unconditionally |  |  |
| lui | Load upper 16bit |  |  |
| mov | Move the contents of the source to the destination register |  |  |
| movi | Move the contents of the literal value to the destination register |  |  |
| jeq | Jump if registers values are equal |  |  |
| jneq | Jump if registers values are not equal |  |  |
| jg | Jump if register1 value is greater than register2 |  |  |
| jge | Jump if register1 value is greater than or equal register2 |  |  |
| jl | Jump if register1 value is less than register2 |  |  |
| jle | Jump if register1 value is greater than or equalregister2 |  |  |
| spc | Save the program counter to a register |  |  |

