Lance Tecson lat4nyv S24 Computer Architecture

The purpose of this learning activity is to design a MMIO (memory-mapped input output) for a UART module and to write an assembly file to test its ability to receive and transmit UART data. UART is a communication protocol that is idle low and transfers 8 data bits (beginning with the LSB) prefaced by a start bit (to identify the start of a transmission) and ending with a parity bit then stop bit. The parity bit serves as a means to detect errors. The signal then returns to an idle high.

The module implements the UART transmission and receiving capabilities from past learning activities but updates the address decoder of a CPU made in prior learning activities to include memory for the status register, write register, read register, and UART enable. The status register has a 32 bit output with the 0th bit containing TxRDY, the 1st bit containing RxRDY, the 2nd RxParityErr, and the remaining bits being 0. {0: TxRDY, 1: RxRDY, 2: RxParityErr, 3:31: 0}. It inputs a clock, the above bits, an enable, a reset, and an output enable. If the output is disabled, the output is 32 bits of high impedance. The baud clock register outputs 2 baud clocks; one operates at 16 times the speed of the other. The status register is continuously polled to determine if the transmit buffer is empty or the receive buffer is full.

Address Decoder(MemWrite, Addr) => (RAM_CS, RAM_WE, ROM_CS, UART_WR, UART_RD, CE_SR, CE_UART)

All inputs and outputs are 1 bit except for Addr which is 32 bits. MemWrite indicates if an address is being written to (1) or read (0). Addr is the address being written or read. RAM_CS, ROM_CS, CE_UART, and CE_SR are 1 if Addr is in their range of memory; otherwise they are 0. They are all mutually exclusive.

{ROM_CS: 0x0 : 0x400, RAM_CS: 0x400 : 0x4FF, CE_UART: 0x500, CE_SR: 0x504 }
RAM_WE and UART_WR are high if MemWrite is high; otherwise they are low. The opposite relationship is true for UART_RD.
No other components were changed from prior learning activities.

Verification

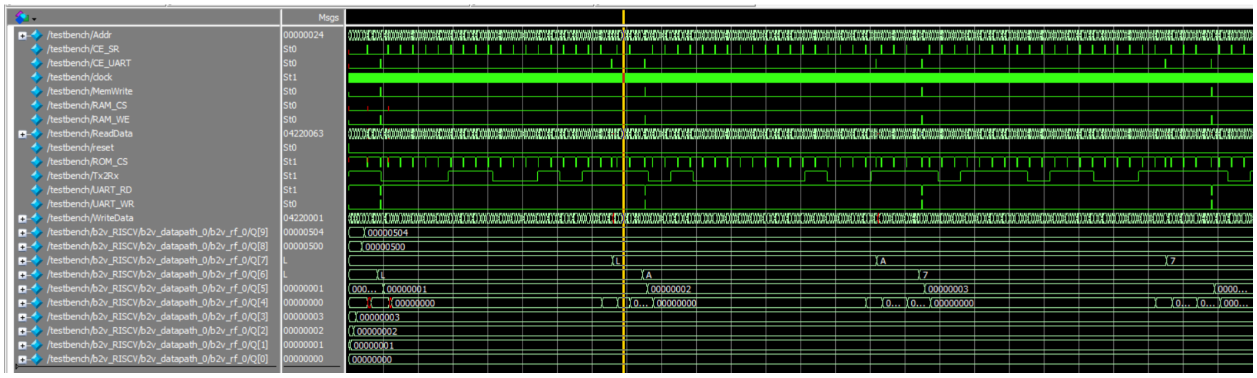
An assembly program was written that used the opcodes understandable by RISCV architecture. The program is in the appendix and works by using registers 0x0 - 0x9. 0x0-0x3 are kept constant at 0, 1, 2, and 3 respectively. 0x4 holds the value of the status register. 0x5 is a counter which lets the program know which ascii character to transmit. 0x6 holds the character to transmit. 0x7 holds the status register output to store into the status register. 0x8 and 0x9 hold the address of the UART register and status register respectively.

0x0: 0 0x1: 1 0x2: 2 0x3: 3 0x4: status register status 0x5: counter 0x6: received byte
0x7: transmitted byte 0x8: UART register address 0x9: status register address

At a high level, the program sets its constants, polls until ready to transmit or receive, transmits a “L”, increments the counter, branches to the polling macro, (at this point it is ready to read) receives the transmitted character, and repeats from the transmit step until this step until all 3 characters are transmitted and received. The transmit macros know to switch since they compare the counter to constants. After transmitting and receiving all 3 bytes, the program continuously loops waiting for input. The 3 outputted characters are “LA7” in that order.

Lance Tecson　　lat4nyv　　S24　　Computer Architecture

The testing program was written in assembly but was then translated to RISCV opcodes; both are seen in the appendix. The file containing the opcodes was the actual file inputted into the UART supporting CPU.



	Msgs
/testbench/Addr	00000024
/testbench/CE_SR	St0
/testbench/CE_UART	St0
/testbench/clock	St1
/testbench/MemWrite	St0
/testbench/RAM_CS	St0
/testbench/RAM_WE	St0
/testbench/ReadData	04220063
/testbench/reset	St0
/testbench/ROM_CS	St1
/testbench/Tx2Rx	St1
/testbench/UART_RD	St1
/testbench/UART_WR	St0
/testbench/WriteData	04220001
/testbench/b2v_RISCV/b2v_datapath_0/b2v_rf_0/Q[9]	00000504
/testbench/b2v_RISCV/b2v_datapath_0/b2v_rf_0/Q[8]	00000500
/testbench/b2v_RISCV/b2v_datapath_0/b2v_rf_0/Q[7]	L
/testbench/b2v_RISCV/b2v_datapath_0/b2v_rf_0/Q[6]	L
/testbench/b2v_RISCV/b2v_datapath_0/b2v_rf_0/Q[5]	00000001
/testbench/b2v_RISCV/b2v_datapath_0/b2v_rf_0/Q[4]	00000000
/testbench/b2v_RISCV/b2v_datapath_0/b2v_rf_0/Q[3]	00000003
/testbench/b2v_RISCV/b2v_datapath_0/b2v_rf_0/Q[2]	00000002
/testbench/b2v_RISCV/b2v_datapath_0/b2v_rf_0/Q[1]	00000001
/testbench/b2v_RISCV/b2v_datapath_0/b2v_rf_0/Q[0]	00000000

Figure 1: Verification

As seen by the data path 7, 3 bytes are transmitted and received: “LA7”. ReadData goes on to repeat the polling macro. 0x0 : 0x3 hold constants (which refer to status register statuses, detailed explanation in a prior learning activity). 0x9 and 0x9 hold the address of the UART register and status register. 0x6 holds the received character. 0x7 holds the transmitted byte. Some time after the character is transmitted from 0x7, the same character is received in 0x6, showing that my program is able to transmit then receive at least 2 bytes worth of data.

```
00100093
00200113
00300193
00000293
50000413
50400493
0004a203
00327213
00120663
04220063
fe0008e3
00029a63
04c00313
00642023
00128293
fc000ee3
00129a63
04100313
00642023
00128293
fc0004e3
03700313
00642023
00128293
fa000ce3
00042383
fa0008e3
```

Figure 2: Opcode Translation of the ASM Code

```
consts:
    addi    x1, x0, 1  # TxRDY
    addi    x2,    x0,    2  # RxRDY
    addi    x3,    x0, 3  # RxParityErr
    addi    x5, x0, 0  # counter
            addi    x8, x0, 0x500  #UART reg
            addi    x9, x0, 0x504  #status reg

polling:
    lw         x4,    0(x9)    # r4 = (status reg)
    andi    x4, x4, 0x3
    beq              x4, x1, TxRDY
    beq              x4, x2, RxRDY
            beq      x0, x0, polling

TxRDY:
            bne      x5, x0, TxRDY2
    addi    x6, x0, 76      # x6 = L
    sw         x6, 0(x8) # store ascii into uart reg
    addi    x5, x5, 1    # inc counter
    beq              x0, x0, polling
    # if 1
    #repeat stuff above


TxRDY2:
    bne              x5, x1, TxRDY3
    addi    x6, x0, 65      # x6 = T
    sw         x6, 0(x8) # store ascii into uart reg
    addi    x5, x5, 1    # inc counter
    beq              x0, x0, polling

TxRDY3:
    addi    x6, x0, 55      # x6 = 7
    sw         x6, 0(x8) # store ascii into uart reg
    addi    x5, x5, 1    # inc counter
    beq              x0, x0, polling

RxRDY:
    lw     x7, 0(x8)
    beq              x0, x0, polling
```

Figure 3: ASM Test File to be Translated to RISCV Opcodes