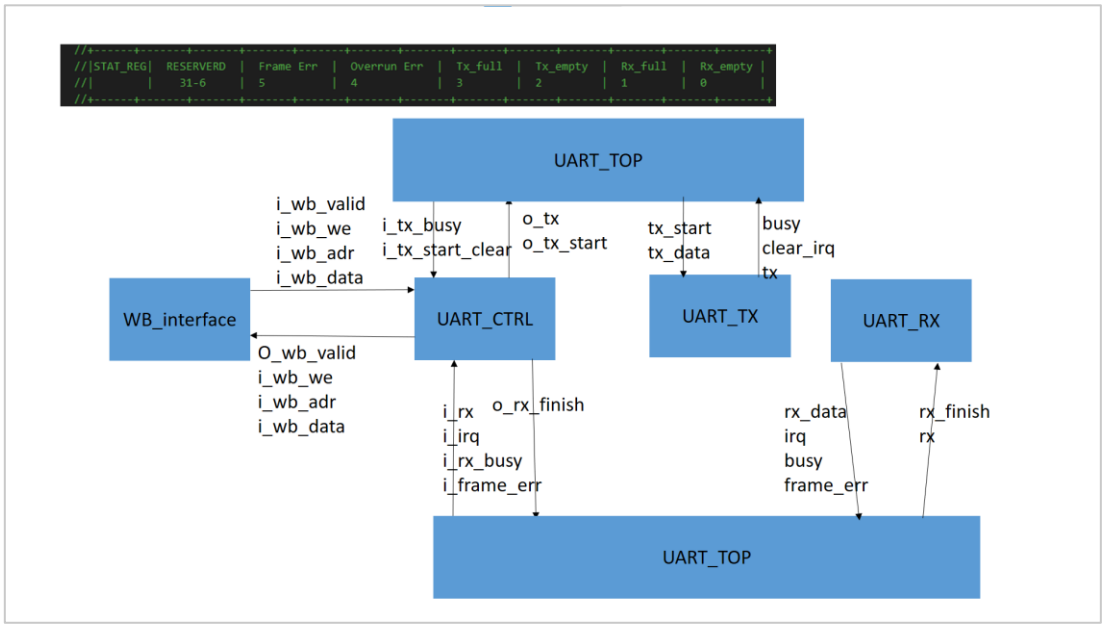
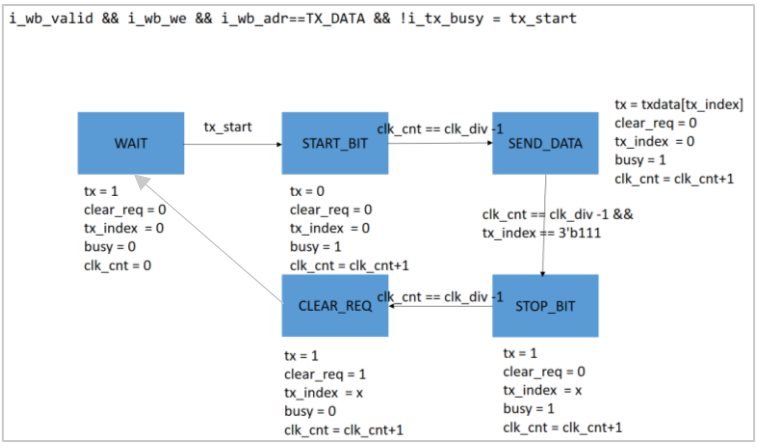
SOC Lab6 report 第六組

Block Diagram for UART：

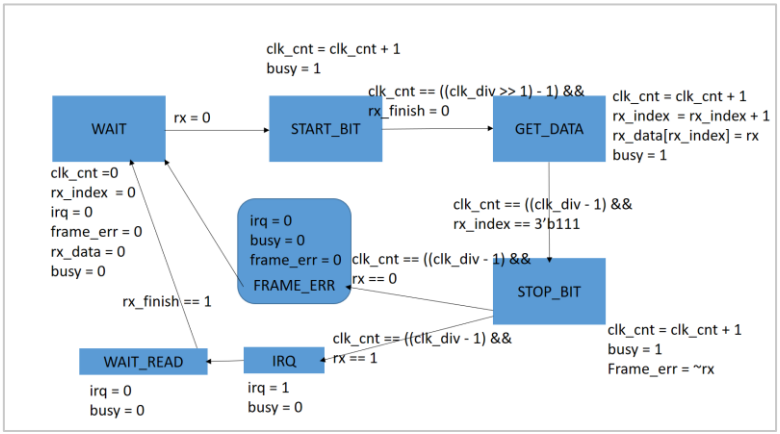


可以觀察到 ctrl 的輸出入剛好和 tx、rx 是相反方向的

FSM Diagram for uart_tx：



FSM Diagram for UART_rx



Method to verify our answer from notebook：

參考了 github 討論區#175 的做法：

```python
import time
async def loop():
    print("start looping")
    start = time.time()
    while(True):
        reg_mprj_datal = hex(ipPS.read(0x1c))[:6]
        print(f"checkbits:{reg_mprj_datal},time:{(time.time()-start)*10e6}us")
        prev = time.time()
```

卡 while loop 去 print checkbit 和對應時間

```python
# Python 3.7+
async def async_main():
    task2 = asyncio.create_task(caravel_start())
    task1 = asyncio.create_task(uart_rxtx())
    task0 = asyncio.create_task(loop())
    # Wait for 5 second
    await asyncio.sleep(10)
    task1.cancel()
    task0.cancel()
    try:
        await task1
    except asyncio.CancelledError:
        print('main(): uart_rx is cancelled now')
```

Top.c 的部分參考同個討論區另一位同學的方法：

在每個 reg_mprj_datal 輸出的間隔塞 while loop，使韌體的運行速度變慢，就能夠在 ipynb 上確認正確性。

```c
    int j;
    // mm
    int *tmp = matmul();
    reg_mprj_datal = *tmp << 16;
    i = 0;
    while (i < 5000)
    {
        i++;
    }
    reg_mprj_datal = *(tmp+1) << 16;
    i = 0;
    while (i < 5000)
    {
        i++;
    }
    reg_mprj_datal = *(tmp+2) << 16;
    i = 0;
    while (i < 5000)
    {
        i++;
    }
    reg_mprj_datal = *(tmp+3) << 16;
    i = 0;
    while (i < 5000)
    {
        i++;
    }

    reg_mprj_datal = *(tmp+9) << 16;
    i = 0;
    while (i < 5000)
    {
        i++;
    }
    reg_mprj_datal = 0xAB510000;
     i = 0;
    while (i < 10000)
    {
        i++;
    }
```

Firmware 啟動：

```
checkbits:0x8,time:22895.336151123047us
checkbits:0xab40,time:25815.963745117188us
```

mm 開始：

```
checkbits:0xab40,time:41680.335998535156us
checkbits:0x3e00,time:44293.40362548828us
checkbits:0x3e00,time:82421.30279541016us
checkbits:0x4400,time:85790.15731811523us
checkbits:0x4400,time:120456.21871948242us
checkbits:0x4a00,time:123715.40069580078us
checkbits:0x4a00,time:157217.97943115234us
checkbits:0x5000,time:161325.93154907227us
checkbits:0x5000,time:198333.2633972168us
checkbits:0x4400,time:202136.03973388672us
```

Mm 結束：

```
checkbits:0x4400,time:237660.40802001953us
checkbits:0xab51,time:240545.27282714844us
```

Fir 開始：

```
checkbits:0xab51,time:261201.8585205078us
checkbits:0xab40,time:356793.4036254883us
checkbits:0x40,time:360553.2646179199us
```

Fir 結束：

```
checkbits:0x40,time:376236.4387512207us
checkbits:0xab61,time:834686.7561340332us
```

這邊 fir 跟 qs 應該是同時並行的，所以中間有一些 data 和 ab71 可能被蓋掉了，每次重跑 kernal 的輸出都會有些不一樣，多跑幾次這個 kernal 就可以看到是 ab71 結尾的情況，也就是 qs 是最後完成的 operation，符合 top.c 的 behavior。

Qs 開始：

```
checkbits:0xab40,time:334520.3399658203us
checkbits:0xffe7,time:522246.3607788086us
checkbits:0x2300,time:554890.6326293945us
checkbits:0x44a0,time:766339.3020629883us
checkbits:0x44a0,time:794563.2934570312us
checkbits:0x2800,time:893433.0940246582us
checkbits:0x2800,time:921359.0621948242us
checkbits:0xa6d0,time:1020076.2748718262us
checkbits:0xa6d0,time:1037640.5715942383us
checkbits:0xca10,time:1040360.9275817871us
checkbits:0xca10,time:1049022.6745605469us
checkbits:0x120e,time:1148674.488067627us
checkbits:0x120e,time:1157391.07131958us
checkbits:0x1631,time:1160173.4161376953us
```

Qs 結束：

```
checkbits:0x1631,time:1176609.992980957us
checkbits:0x2371,time:1275489.330291748us
checkbits:0xab71,time:1278579.2350769043us
checkbits:0xab71,time:1281425.952911377us
checkbits:0xab71,time:1284201.1451721191us
checkbits:0xab71,time:1286985.8741760254us
```

Timing report/ resource report after synthesis：

```
204  ------------------------------------------------------------------------------------------------
205  From Clock:  clk_fpga_0
206    To Clock:  clk_fpga_0
207
208  Setup :            0  Failing Endpoints,  Worst Slack        8.557ns,  Total Violation        0.000ns
209  Hold  :            0  Failing Endpoints,  Worst Slack        0.026ns,  Total Violation        0.000ns
210  PW    :            0  Failing Endpoints,  Worst Slack       11.250ns,  Total Violation        0.000ns
211  ------------------------------------------------------------------------------------------------
```

```
337  ------------------------------------------------------------------------------------------------
338  Path Group:  **async_default**
339  From Clock:  clk_fpga_0
340    To Clock:  clk_fpga_0
341
342  Setup :            0  Failing Endpoints,  Worst Slack       16.891ns,  Total Violation        0.000ns
343  Hold  :            0  Failing Endpoints,  Worst Slack        0.526ns,  Total Violation        0.000ns
344  ------------------------------------------------------------------------------------------------
```

合成後的 delay 確實都有 MET（slack > 0）

Latency for a character loop back using UART：

```
In [9]: import time

async def uart_rxtx():
    # Reset FIFOs, enable interrupts
    ipUart.write(CTRL_REG, 1<<RST_TX | 1<<RST_RX | 1<<INTR_EN)
    print("Waitting for interrupt")
    tx_str = "hello\n"
    ipUart.write(TX_FIFO, ord(tx_str[0]))
    i = 1
    while(True):
        await intUart.wait()
        buf = ""
        # Read FIFO until valid bit is clear
        while ((ipUart.read(STAT_REG) & (1<<RX_VALID))):
            buf += chr(ipUart.read(RX_FIFO))
            if i>1:
                print(f"  time : {(time.time() - start_time):.6f} s")
            if i<len(tx_str):
                start_time = time.time()
                ipUart.write(TX_FIFO, ord(tx_str[i]))
                i=i+1
        print(buf, end='')

async def caravel_start():
    ipOUTPIN.write(0x10, 0)
    print("Start Caravel Soc")
    ipOUTPIN.write(0x10, 1)
```

```
In [10]: 1 asyncio.run(async_main())

Start Caravel Soc
Waitting for interrupt
h  time : 0.003027 s
e  time : 0.006397 s
l  time : 0.003376 s
l  time : 0.004674 s
o  time : 0.003228 s

main(): uart_rx is cancelled now
```

平均下來 Latency 為 4.17 ms 左右

Suggestion for improving latency for UART loop back：

1. Baud Rate：

在此次實驗中是採用 40Mhz，或許我們可以嘗試看看更高的頻率。而較高的頻率通常具有較低的延遲。我們選擇發送器和接收器都可以支援且不會出現錯誤的頻率。

2. DMA：

改採用 DMA 而不使用 UART 的話，我們就不須透過 processer 發送 address，可以獨立地直接讀寫 memory，從而減少延遲。

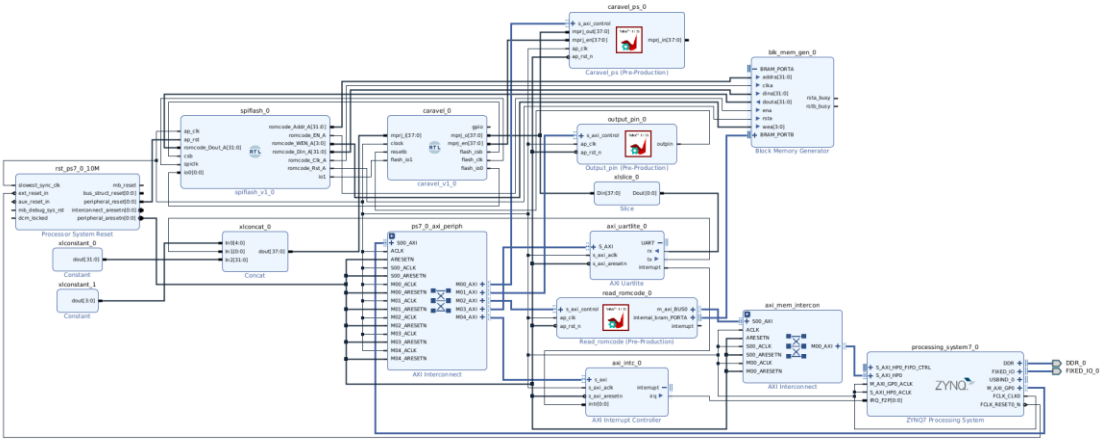
RTL 驗證:

```
initial begin
    fork
        test_mm_qs_fir;
        test_uart;
    join
    $finish;
end
```

同時 fork 兩個 process，同時驗證 mm_qs_fir 以及 uart 的 functionality。

```
ubuntu@ubuntu2004:~/SoCLab/lab-wlos_baseline/testbench/top$ source run_clean
ubuntu@ubuntu2004:~/SoCLab/lab-wlos_baseline/testbench/top$ source run_sim
Reading top.hex
top.hex loaded into memory
Memory 5 bytes = 0x6f 0x00 0x00 0x0b 0x13
LA mm 1 started
LA uart 1 started
tx data bit index 0: 1
tx data bit index 1: 1
tx data bit index 2: 1
tx data bit index 3: 1
tx data bit index 4: 0
tx data bit index 5: 0
tx data bit index 6: 0
tx data bit index 7: 0
tx complete 1
rx data bit index 0: 1
rx data bit index 1: 1
rx data bit index 2: 1
rx data bit index 3: 1
rx data bit index 4: 0
rx data bit index 5: 0
rx data bit index 6: 0
rx data bit index 7: 0
Received 15
LA uart 1 passed
Call function matmul() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x003e
Call function matmul() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x0044
Call function matmul() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x004a
recevied word  15
Call function matmul() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x0050
LA mm 2 passed
LA fir 1 started
LA fir 2 passed
LA qs 1 started
Call function qsort() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x0028
Received    40
Received   893
Received  2541
Received  2669
LA qs 2 passed
```

Vivado block diagram:



Firmware (top.c):

```
void main()
{
#ifdef USER_PROJ_IRQ0_EN
    int mask;
#endif
    reg_mprj_io_31 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_30 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_29 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_28 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_27 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_26 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_25 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_24 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_23 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_22 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_21 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_20 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_19 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_18 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_17 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_16 = GPIO_MODE_MGMT_STD_OUTPUT;

    reg_mprj_io_15 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_14 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_13 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_12 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_11 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_10 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_9  = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_8  = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_7  = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_4  = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_3  = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_2  = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_1  = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_0  = GPIO_MODE_MGMT_STD_OUTPUT;

    reg_mprj_io_6  = GPIO_MODE_USER_STD_OUTPUT;
    reg_mprj_io_5  = GPIO_MODE_USER_STD_INPUT_NOPULL;
```

```
#ifdef USER_PROJ_IRQ0_EN
    // unmask USER_IRQ_0_INTERRUPT
    mask = irq_getmask();
    mask |= 1 << USER_IRQ_0_INTERRUPT; // USER_IRQ_0_INTERRUPT = 2
    irq_setmask(mask);
    // enable user_irq_0_ev_enable
    user_irq_0_ev_enable_write(1);
#endif

    // Now, apply the configuration
    reg_mprj_xfer = 1;
    while (reg_mprj_xfer == 1);

    reg_la0_oenb = reg_la0_iena = 0x00000000;    // [31:0]
    reg_la1_oenb = reg_la1_iena = 0xFFFFFFFF;    // [63:32]
    reg_la2_oenb = reg_la2_iena = 0x00000000;    // [95:64]
    reg_la3_oenb = reg_la3_iena = 0x00000000;    // [127:96]

    // Flag start of the test
    reg_mprj_datal = 0xAB400000;
    // Set Counter value to zero through LA probes [63:32]
    reg_la1_data = 0x00000000;
    // Configure LA probes from [63:32] as inputs to disable counter write
    reg_la1_oenb = reg_la1_iena = 0x00000000;

    int j;
    // mm
    int *tmp = matmul();
    reg_mprj_datal = *tmp << 16;
    reg_mprj_datal = *(tmp+1) << 16;
    reg_mprj_datal = *(tmp+2) << 16;
    reg_mprj_datal = *(tmp+3) << 16;

    reg_mprj_datal = *(tmp+9) << 16;
    reg_mprj_datal = 0xAB510000;
```

Firmware 的一開始先設置 mprj 的位置，緊接著就 enable uart interrupt，最後才送 3 個 workload(mm, qs, fir)。

```
// mm
int *tmp = matmul();
reg_mprj_datal = *tmp << 16;
reg_mprj_datal = *(tmp+1) << 16;
reg_mprj_datal = *(tmp+2) << 16;
reg_mprj_datal = *(tmp+3) << 16;

reg_mprj_datal = *(tmp+9) << 16;
reg_mprj_datal = 0xAB510000;


// fir
reg_mprj_datal = 0xAB400000;
tmp = fir();
reg_mprj_datal = *tmp << 16;
reg_mprj_datal = *(tmp+1) << 16;
reg_mprj_datal = *(tmp+2) << 16;
reg_mprj_datal = *(tmp+3) << 16;
reg_mprj_datal = *(tmp+4) << 16;
reg_mprj_datal = *(tmp+5) << 16;
reg_mprj_datal = *(tmp+6) << 16;
reg_mprj_datal = *(tmp+7) << 16;
reg_mprj_datal = *(tmp+8) << 16;
reg_mprj_datal = *(tmp+9) << 16;
reg_mprj_datal = *(tmp+10) << 16;
reg_mprj_datal = 0xAB610000;

// qs
reg_mprj_datal = 0xAB400000;
tmp = qsort();
reg_mprj_datal = *tmp << 16;
reg_mprj_datal = *(tmp+1) << 16;
reg_mprj_datal = *(tmp+2) << 16;
reg_mprj_datal = *(tmp+3) << 16;
reg_mprj_datal = *(tmp+4) << 16;
reg_mprj_datal = *(tmp+5) << 16;
reg_mprj_datal = *(tmp+6) << 16;
reg_mprj_datal = *(tmp+7) << 16;
reg_mprj_datal = *(tmp+8) << 16;
reg_mprj_datal = *(tmp+9) << 16;
// reg_mprj_datal = *tmp << 16;
reg_mprj_datal = 0xAB710000;
```