Lab 6 Report

Name: 陳揚哲/李柏辰/楊正宇

Overview

這次 lab6 的內容主要也是基於前幾次 lab 的基礎下再進行延伸，原先在 workload 的部分，lab4 只有進行 FIR 的實作。而在 lab6 則是另外加入了矩陣乘法以及 quick sort，並且要在同一次的執行中完成這三項功能。此外，在 Caravel 與 PS 端之間的資料交換採取 interrupt 的方式，當有資料準備好要傳給對方時，會透過 UART protocol 發送訊息給對方的 IRQ，收到這個 interrupt 的就會先放下原先在執行的指令，優先去接收即將到來的資料。

Verify answer from Notebook

根據不同 workload 運算完畢後的 checkbit，可以確認我們在 FPGA 上的行值結果是否正確。我們選擇比對 last pattern 的 checkbit，當作不同運算的檢查。

```
Call function fir() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x044a
================
[Passed]FIR passed!
================
```

```
Call function matmul() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x0050
================
[Passed]Matmul passed!
================
```

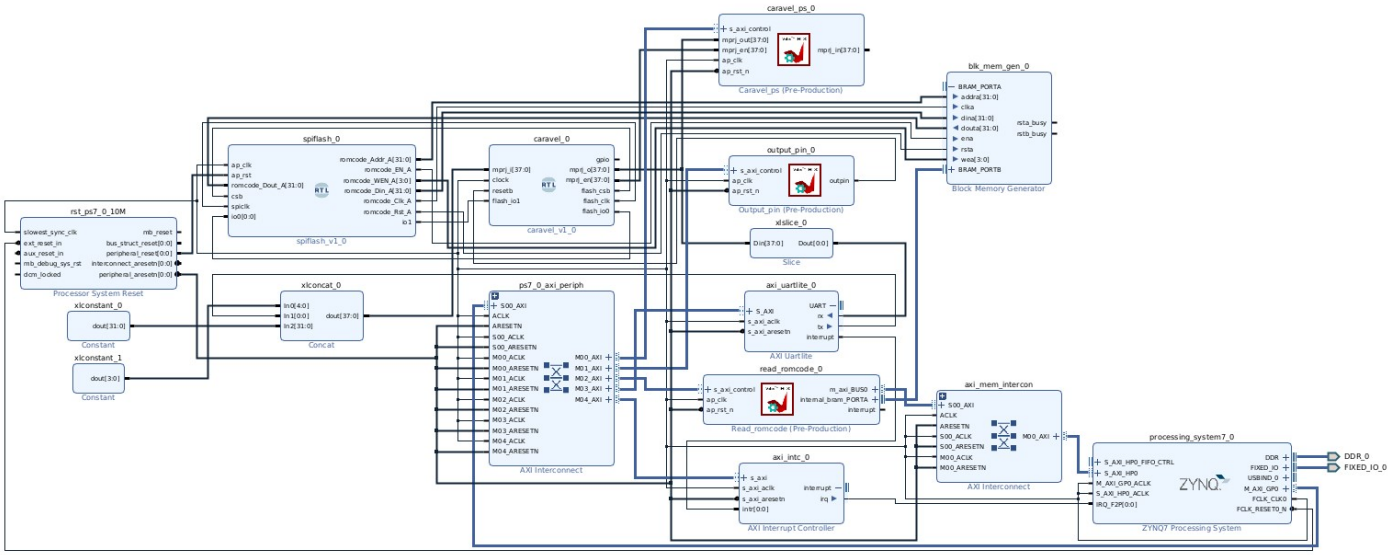
```
Call function qsort() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x2371
================
[Passed]Qsort passed!
================
```

```
async def checkbit():
    while((hex(ipPS.read(0x1c))) [:6] == 0x044a ) :
        continue
    print("last pattern is 0x044a,fir passed")
    while((hex(ipPS.read(0x1c))) [:6] ==0x0050):
        continue
    print("last pattern is 0x0050,mm passed")

    while((hex(ipPS.read(0x1c))) [:6]  == 0x2371) :
        continue
    print("last pattern is 0x2371,qs passed")



# Python 3.7+
async def async_main():
    task2 = asyncio.create_task(caravel_start())
    task1 = asyncio.create_task(uart_rxtx())
    task3 = asyncio.create_task(checkbit())
    # Wait for 5 second
    await asyncio.sleep(10)
    task1.cancel()

    try:
        await task1
    except asyncio.CancelledError:
        print('main(): uart_rx is cancelled now')
```

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

```
Start Caravel Soc
Waitting for interrupt
last pattern is 0x044a,fir passed
last pattern is 0x0050,mm passed
last pattern is 0x2371,qs passed
hellomain(): uart_rx is cancelled now
```

Block Design



● Time Report / Source Report

✓ Time Report

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	8.557 ns	Worst Hold Slack (WHS):	0.026 ns	Worst Pulse Width Slack (WPWS):	11.250 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	12669	Total Number of Endpoints:	12669	Total Number of Endpoints:	5261

All user specified timing constraints are met.

✓ Source Report

1. Slice Logic

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs	5330	0	0	53200	10.02
LUT as Logic	5142	0	0	53200	9.67
LUT as Memory	188	0	0	17400	1.08
LUT as Distributed RAM	18	0			
LUT as Shift Register	170	0			
Slice Registers	6159	0	0	106400	5.79
Register as Flip Flop	6159	0	0	106400	5.79
Register as Latch	0	0	0	106400	0.00
F7 Muxes	170	0	0	26600	0.64
F8 Muxes	47	0	0	13300	0.35

3. Memory

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	6	0	0	140	4.29
RAMB36/FIFO*	3	0	0	140	2.14
RAMB36E1 only	3				
RAMB18	6	0	0	280	2.14
RAMB18E1 only	6				

Latency for a character loop back using UART

```
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):
                tend = time.time()
                tperiod = tend - tstart
                print("latency:",tperiod)
            if i<len(tx_str):
                tstart = time.time()
                ipUart.write(TX_FIFO, ord(tx_str[i]))
                i=i+1
        print(buf, end=', ')
```

我們在 Jupyter Notebook 程式碼裡負責 UART 傳收的 function 中，加入了 timer 幫助計算 UART 在傳送字元所花費的時間。先在 ipUart.write 的地方，也就是 UART 中 TX 進行發送資料的動作，設為 Start 的時間。接著在 ipUart.read 的地方，也就是 UART 中 RX 進行收取資料的動作，設為 End 的時間，但為了避免 RX 還沒實際讀到值的時候 timer 就被啟動，有個用來判斷 tx 剩下字串長度的變數 i，當 i 還是 1 的時候代表 TX 實際上還沒傳值過來，這時候即使 RX 做 read 也不會啟動到 timer。

接著在讓程式把 start 與 end 兩個時間點相減，就可以得到 UART 傳送與收取一個字元所花費的時間。

```
Start Caravel Soc
Waitting for interrupt
h, latency: 0.003417491912841797
e, latency: 0.003958225250244141
l, latency: 0.0030786991119384766
l, latency: 0.0033111572265625
o, latency: 0.004184246063232422

, main(): uart_rx is cancelled now
```

執行的成果如上圖，可以看到每傳一個字元就會同時回傳它所花費的時間，這五個字元平均花費的時間約為 0.00359 秒。

● Improving latency for UART loop back

```
void __attribute__ ( ( section ( ".mprj" ) ) ) uart_end()
{
    endflag = 1;
}

void __attribute__ ( ( section ( ".mprj" ) ) ) uart_write(int n)
{
    while(((reg_uart_stat>>3) & 1));
    reg_tx_data = n;
}

void __attribute__ ( ( section ( ".mprj" ) ) ) uart_write_char(char c)
{
    if (c == '\n')
        uart_write_char('\r');

    // wait until tx_full = 0
    while(((reg_uart_stat>>3) & 1));
    reg_tx_data = c;
}
```

```
.mprj           0x0000000010000a6c
 .mprj          0x0000000010000a6c
                0x0000000010000a6c
                0x0000000010000a90
                0x0000000010000ad8
                0x0000000010000b40
                0x0000000010000b94
                0x0000000010000c08
```

在觀察 UART 的 firmware code 時，我們發現當中的 function 在讀取 instruciton 的位置是”.mprj”，進到 output.map 查看後才發現其指向的位址原來是 spiflash。等於每發生一次 interrupt，就必須要與 spiflash 進行資料交換，我們認為這會是使 interrupt 的 overhead 如此可觀的重要原因之一。因此若是能夠將 UART 的儲存位置改到 user project wrapper 中的 bram，減少需要跑到外部與 spiflash 溝通的成本，應該可以有效地降低 UART 傳收資料的 latency。