SoC Design Lab6

111064559 徐詠祺　　112061527 紀承龍

■ **How do you verify your answer from notebook:**

- 透過和 testbench 相同方式去檢測 mprj checkbits 在執行各個 task 後的輸出結果，來判斷是否 CPU 有完成正確的運算，並且此處為了解決 PS access checkbits 太慢導致 PS 來不及讀取 checkbits 就已經改變的情況，我們在 firmware 中對 checkbits 進行賦值後增加了 while loop 空轉的部分以增加 checkbits 兩次變化的時間差，藉此來避免 PS 上述的問題，透過此方法在 notebook 上即可順利完成驗證。

```python
async def task_check():
    while((ipPS.read(0x1c)&0xffff0000) != 0xAB410000):
        continue
    print("matmul started")
    while((ipPS.read(0x1c)&0xffff0000) != 0x003E0000):
        continue
    print("matmul 1 passed")
    while((ipPS.read(0x1c)&0xffff0000) != 0x00500000):
        continue
    print("matmul 2 passed")

    while((ipPS.read(0x1c)&0xffff0000) != 0xAB420000):
        continue
    print("qsort started")
    while((ipPS.read(0x1c)&0xffff0000) != 0x00280000):
        continue
    print("qsort 1 passed")
    while((ipPS.read(0x1c)&0xffff0000) != 0x23710000):
        continue
    print("qsort 2 passed")

    while((ipPS.read(0x1c)&0xffff0000) != 0xAB430000):
        continue
    print("fir 1 started")
    while((ipPS.read(0x1c)&0xffff0000) != 0x00000000):
        continue
    print("fir 2 started")
    while((ipPS.read(0x1c)&0xffff0000) != 0x044A0000):
        continue
    print("fir passed")
    print("ALL Test Done")
```

```c
// TASK START
reg_mprj_data1 = 0xAB410000;
int c = 0;
while (c<100000) { c = c+1 };

// MATMUL
int *tmp = matmul();
reg_mprj_data1 = *tmp << 16;
c = 0;
while (c<100000) { c = c+1 };
reg_mprj_data1 = *(tmp+3) << 16;
c = 0;
while (c<100000) { c = c+1 };

// QSORT
reg_mprj_data1 = 0xAB420000;
c = 0;
while (c<100000) { c = c+1 };
int* tmp2 = qsort();
reg_mprj_data1 = *tmp2 << 16;
c = 0;
while (c<100000) { c = c+1 };
reg_mprj_data1 = *(tmp2+9) << 16;
c = 0;
while (c<100000) { c = c+1 };

// FIR
reg_mprj_data1 = 0xAB430000;
c = 0;
while (c<100000) { c = c+1 };
int* tmp3 = fir();
reg_mprj_data1 = *tmp3 << 16;
c = 0;
while (c<100000) { c = c+1 };
reg_mprj_data1 = *(tmp3+10) << 16;
c = 0;
while (c<100000) { c = c+1 };
```

- 最後讀取 mprj_io 的值也正確。

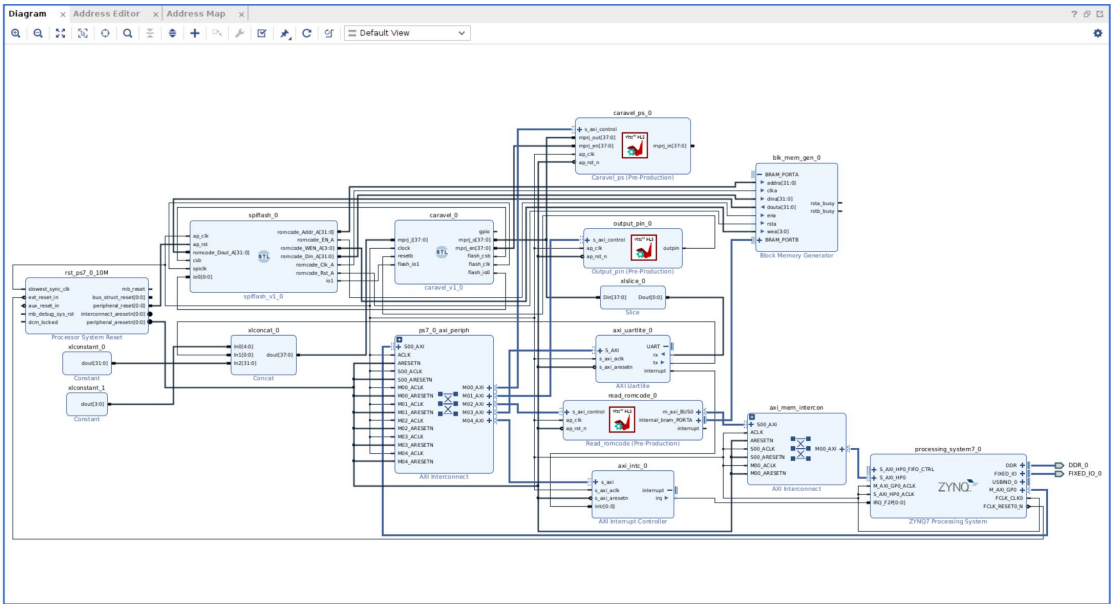
```
    asyncio.run(async_main())
```

```
Start Caravel Soc
Waitting for interrupt
matmul started
matmul 1 passed
matmul 2 passed
qsort started
qsort 1 passed
qsort 2 passed
fir 1 started
fir 2 started
fir passed
ALL Test Done
hello
```

```
    print ("0x10 = ", hex(ipPS.read(0x10)))
    print ("0x14 = ", hex(ipPS.read(0x14)))
    print ("0x1c = ", hex(ipPS.read(0x1c)))
    print ("0x20 = ", hex(ipPS.read(0x20)))
    print ("0x34 = ", hex(ipPS.read(0x34)))
    print ("0x38 = ", hex(ipPS.read(0x38)))
```

```
0x10 =  0x0
0x14 =  0x0
0x1c =  0xab510040
0x20 =  0x0
0x34 =  0x20
0x38 =  0x3f
```

■ Block Design:



■ Timing report/ resource report after synthesis:

- Timing report:

```
----------------------------------------------------------------------------------------------------
From Clock:  clk_fpga_0
  To Clock:  clk_fpga_0

Setup :            0  Failing Endpoints,  Worst Slack        9.241ns,  Total Violation        0.000ns
Hold  :            0  Failing Endpoints,  Worst Slack        0.019ns,  Total Violation        0.000ns
PW    :            0  Failing Endpoints,  Worst Slack       11.250ns,  Total Violation        0.000ns
----------------------------------------------------------------------------------------------------
```

```
Max Delay Paths
--------------------------------------------------------------------------------------
Slack (MET) :             9.241ns  (required time - arrival time)
  Source:                 design_1_i/processing_system7_0/inst/PS7_i/FCLKCLK[0]
                            (clock source 'clk_fpga_0'  {rise@0.000ns fall@12.500ns period=25.000ns})
  Destination:            design_1_i/caravel_ps_0/inst/control_s_axi_U/int_ps_mprj_out_reg[15]/D
                            (rising edge-triggered cell FDRE clocked by clk_fpga_0  {rise@0.000ns fall@12.500ns period=25.000ns})
  Path Group:             clk_fpga_0
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            12.500ns  (clk_fpga_0 rise@25.000ns - clk_fpga_0 fall@12.500ns)
  Data Path Delay:        5.276ns  (logic 0.374ns (7.089%)  route 4.902ns (92.911%))
  Logic Levels:           3  (BUFG=1 LUT1=1 LUT6=1)
  Clock Path Skew:        2.646ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    2.646ns = ( 27.646 - 25.000 )
    Source Clock Delay      (SCD):    0.000ns = ( 12.500 - 12.500 )
    Clock Pessimism Removal (CPR):    0.000ns
  Clock Uncertainty:      0.377ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.750ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns
```

```
Min Delay Paths
--------------------------------------------------------------------------------------
Slack (MET) :             0.019ns  (arrival time - required time)
  Source:                 design_1_i/processing_system7_0/inst/PS7_i/FCLKCLK[0]
                            (clock source 'clk_fpga_0'  {rise@0.000ns fall@12.500ns period=25.000ns})
  Destination:            design_1_i/caravel_ps_0/inst/control_s_axi_U/int_ps_mprj_out_reg[14]/D
                            (rising edge-triggered cell FDRE clocked by clk_fpga_0  {rise@0.000ns fall@12.500ns period=25.000ns})
  Path Group:             clk_fpga_0
  Path Type:              Hold (Min at Fast Process Corner)
  Requirement:            0.000ns  (clk_fpga_0 rise@0.000ns - clk_fpga_0 rise@0.000ns)
  Data Path Delay:        1.703ns  (logic 0.071ns (4.170%)  route 1.632ns (95.830%))
  Logic Levels:           2  (BUFG=1 LUT6=1)
  Clock Path Skew:        1.186ns (DCD - SCD - CPR)
    Destination Clock Delay (DCD):    1.186ns
    Source Clock Delay      (SCD):    0.000ns
    Clock Pessimism Removal (CPR):    -0.000ns
  Clock Uncertainty:      0.377ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.750ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns
```

- Resource:

1. Slice Logic

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs	5349	0	0	53200	10.05
LUT as Logic	5161	0	0	53200	9.70
LUT as Memory	188	0	0	17400	1.08
LUT as Distributed RAM	18	0			
LUT as Shift Register	170	0			
Slice Registers	6175	0	0	106400	5.80
Register as Flip Flop	6175	0	0	106400	5.80
Register as Latch	0	0	0	106400	0.00
F7 Muxes	169	0	0	26600	0.64
F8 Muxes	47	0	0	13300	0.35

1.1 Summary of Registers by Type

Total	Clock Enable	Synchronous	Asynchronous
0	_	-	-
0	_	-	Set
0	_	-	Reset
0	_	Set	-
0	_	Reset	-
0	Yes	-	-
283	Yes	-	Set
1031	Yes	-	Reset
130	Yes	Set	-
4731	Yes	Reset	-

2. Slice Logic Distribution

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice	2378	0	0	13300	17.88
SLICEL	1678	0			
SLICEM	700	0			
LUT as Logic	5161	0	0	53200	9.70
using O5 output only	0				
using O6 output only	4122				
using O5 and O6	1039				
LUT as Memory	188	0	0	17400	1.08
LUT as Distributed RAM	18	0			
using O5 output only	0				
using O6 output only	2				
using O5 and O6	16				
LUT as Shift Register	170	0			
using O5 output only	43				
using O6 output only	81				
using O5 and O6	46				
Slice Registers	6175	0	0	106400	5.80
Register driven from within the Slice	3038				
Register driven from outside the Slice	3137				
LUT in front of the register is unused	2052				
LUT in front of the register is used	1085				
Unique Control Sets	318		0	13300	2.39

* * Note: Available Control Sets calculated as Slice * 1, Review the Control Sets Report for more

3. Memory

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	10	0	0	140	7.14
RAMB36/FIFO*	7	0	0	140	5.00
RAMB36E1 only	7				
RAMB18	6	0	0	280	2.14
RAMB18E1 only	6				

* Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate only
FIFO18E1 occupies a Block RAM Tile, that tile can still accommodate a RAMB18E1

4. DSP

Site Type	Used	Fixed	Prohibited	Available	Util%
DSPs	0	0	0	220	0.00

```
4. DSP
------

+-----------+------+-------+------------+-----------+-------+
| Site Type | Used | Fixed | Prohibited | Available | Util% |
+-----------+------+-------+------------+-----------+-------+
| DSPs      |    0 |     0 |          0 |       220 |  0.00 |
+-----------+------+-------+------------+-----------+-------+


5. IO and GT Specific
---------------------

+-----------------------------+------+-------+------------+-----------+--------+
|          Site Type          | Used | Fixed | Prohibited | Available |  Util% |
+-----------------------------+------+-------+------------+-----------+--------+
| Bonded IOB                  |    0 |     0 |          0 |       125 |   0.00 |
| Bonded IPADs                |    0 |     0 |          0 |         2 |   0.00 |
| Bonded IOPADs               |  130 |   130 |          0 |       130 | 100.00 |
| PHY_CONTROL                 |    0 |     0 |          0 |         4 |   0.00 |
| PHASER_REF                  |    0 |     0 |          0 |         4 |   0.00 |
| OUT_FIFO                    |    0 |     0 |          0 |        16 |   0.00 |
| IN_FIFO                     |    0 |     0 |          0 |        16 |   0.00 |
| IDELAYCTRL                  |    0 |     0 |          0 |         4 |   0.00 |
| IBUFDS                      |    0 |     0 |          0 |       121 |   0.00 |
| PHASER_OUT/PHASER_OUT_PHY   |    0 |     0 |          0 |        16 |   0.00 |
| PHASER_IN/PHASER_IN_PHY     |    0 |     0 |          0 |        16 |   0.00 |
| IDELAYE2/IDELAYE2_FINEDELAY |    0 |     0 |          0 |       200 |   0.00 |
| ILOGIC                      |    0 |     0 |          0 |       125 |   0.00 |
| OLOGIC                      |    0 |     0 |          0 |       125 |   0.00 |
+-----------------------------+------+-------+------------+-----------+--------+
```

- **Latency for a character loop back using UART:**
  - 收送完 hello 字元的 latency 大概 290ms，所以一個字元的 latency 大概是 290ms/5=58ms。

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

Start Caravel Soc
Waitting for interrupt
hello
Uart latency: 290.3437614440918 ms
```

- **Suggestion for improving latency for UART loop back:**
  - 第一時間想到減少 UART 收送時間的方法就是提升 baud rate，不過改變 baud rate 很高機率會出現收送失敗，而出現亂碼的情況。另一種方法就是 Compiler Optimization，Lab4-2 有其他組提出了此種方式，利用 GCC compiler 的參數讓 firmware 能夠更有效率地跟 CPU 溝通，產生出來的.hex 檔 GCC compiler 也會想盡辦法把檔案大小(code)壓到最小，因此我們就在 run-sim 加上**-Oz**，github 討論區有人提出了-O1, -O2, -O3, -Ofast, -Os 能使用，不過也有額外找到-Og, -Oz 能使用，在 jupyter 上實測下來最後加上**-Oz** 給 compiler 優化後的 Uart latency 減少了約 90ms。

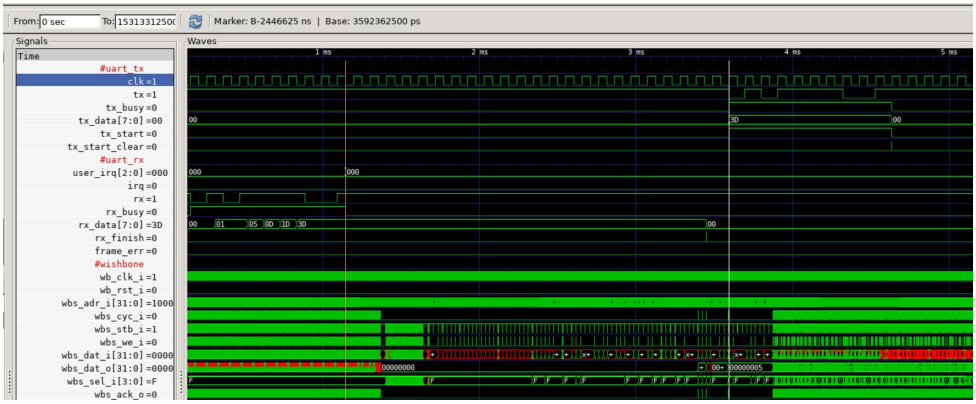
```
In [10]:  1  asyncio.run(async_main())

Start Caravel Soc
Waitting for interrupt
hello
Uart latency: 204.24127578735352 ms
```

- **What else do you observe:**

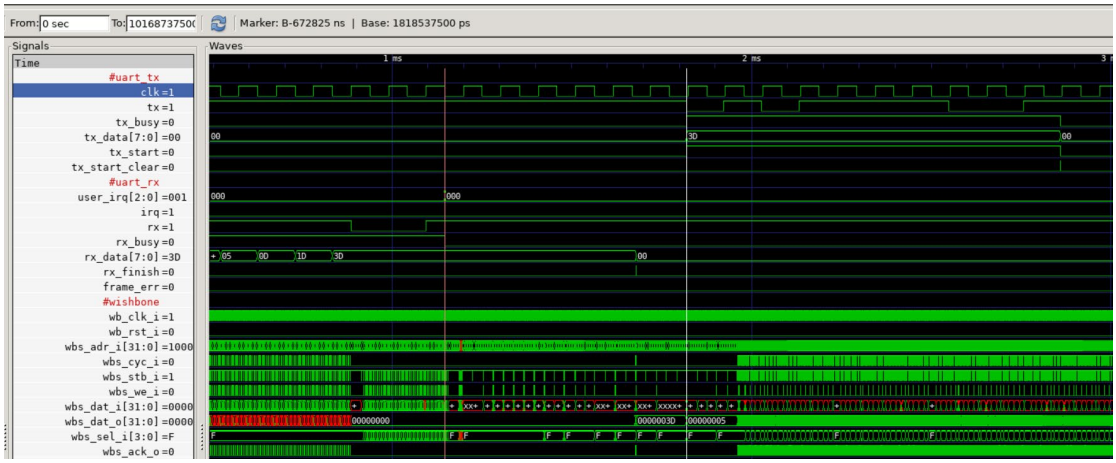
1. 從 compile 完的.out 中可以看到 ISR function(Interrupt Service Routine)，當系統中斷發生時，CPU 會中斷正在執行的程序，轉而執行 ISR，並且由下圖 ISR 內容可以看出中斷發生後 CPU 會執行 uart_read 來收走 uart rx 所收到的資料，並接著進行 uart_write 把收到的資料由 uart tx 送出以完成一次 loopback。由 ISR 的地址可知他是放在 flash 裡面的，若是能把 ISR 和 uart_read/write 也放進 user bram 中應能減少 CPU access instruction 所需的時間，也就能減少 uart loop-back latency。

```
10000238 <isr>:
10000238: fe010113          addi  sp,sp,-32
1000023c: 00112e23          sw  ra,28(sp)
10000240: 00812c23          sw  s0,24(sp)
10000244: 00912a23          sw  s1,20(sp)
10000248: 02010413          addi  s0,sp,32
1000024c: fc5ff0ef          jal ra,10000210 <irq_pending>
10000250: 00050493          mv  s1,a0
10000254: f95ff0ef          jal ra,100001e8 <irq_getmask>
10000258: 00050793          mv  a5,a0
1000025c: 00f4f7b3          and a5,s1,a5
10000260: fef42623          sw  a5,-20(s0)
10000264: fec42783          lw  a5,-20(s0)
10000268: 0047f793          andi  a5,a5,4
1000026c: 02078063          beqz  a5,1000028c <isr+0x54>
10000270: 00100513          li  a0,1
10000274: f3dff0ef          jal ra,100001b0 <user_irq_0_ev_pending_write>
10000278: 029000ef          jal ra,10000aa0 <uart_read>
1000027c: fea42423          sw  a0,-24(s0)
10000280: fe842503          lw  a0,-24(s0)
10000284: 6a4000ef          jal ra,10000928 <uart_write>
10000288: 00000013          nop
1000028c: 00000013          nop
10000290: 01c12083          lw  ra,28(sp)
10000294: 01812403          lw  s0,24(sp)
10000298: 01412483          lw  s1,20(sp)
1000029c: 02010113          addi  sp,sp,32
100002a0: 00008067          ret
```

2. 從 rx 發起 irq 到 tx 傳輸這段時間 CPU 正在處理 interrupt，大概花了 2446625/25 = 97865 個 cycles



若交由 compiler 去優化，這期間縮短至 672825/25 = 26913 個 cycle，幾乎減少了三倍的時間，cache 肯定佔了很多的功勞。



3. 發生中斷 CsrPlugin_mepc 會記錄一個地址，去看.out 檔發現記錄這個地址要做的是 uart 要準備寫入一個字元。


```
295   100003f4: fe079ce3            bnez  a5,100003ec <uart_write_char+0x1c>
296   100003f8: 300007b7            lui a5,0x30000
```

4. 跟 Interrupt 有關連的是這個_zz_CsrPlugin_csrMapping_readDataInit_1，這是一個地址，不過目前還不知道是怎麼運作的，我在猜想應該是處理 exception 的某個狀態用的。


```
3350    assign externalInterrupt = (_zz_CsrPlugin_csrMapping_readDataInit_1 != 32'h0);
```