國立清華大學

系統晶片設計

SOC Design

Lab 5

組別: 第 12 組

學號:111063548、111061624、112501538

姓名:蕭方凱、尤弘瑋、葉承泓

指導老師:賴瑾教授

目錄

1. Block Diagram

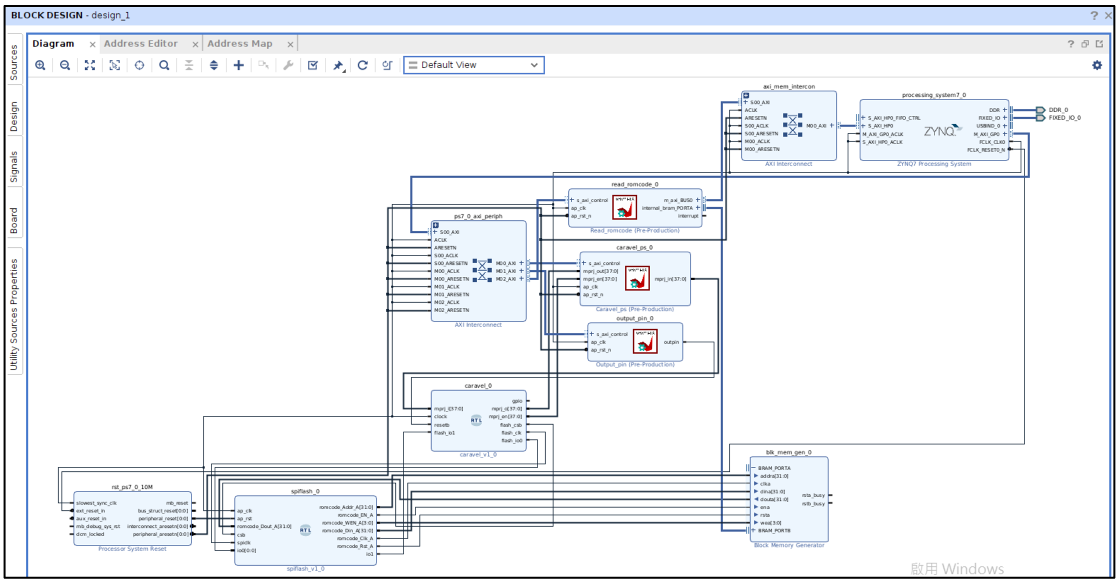
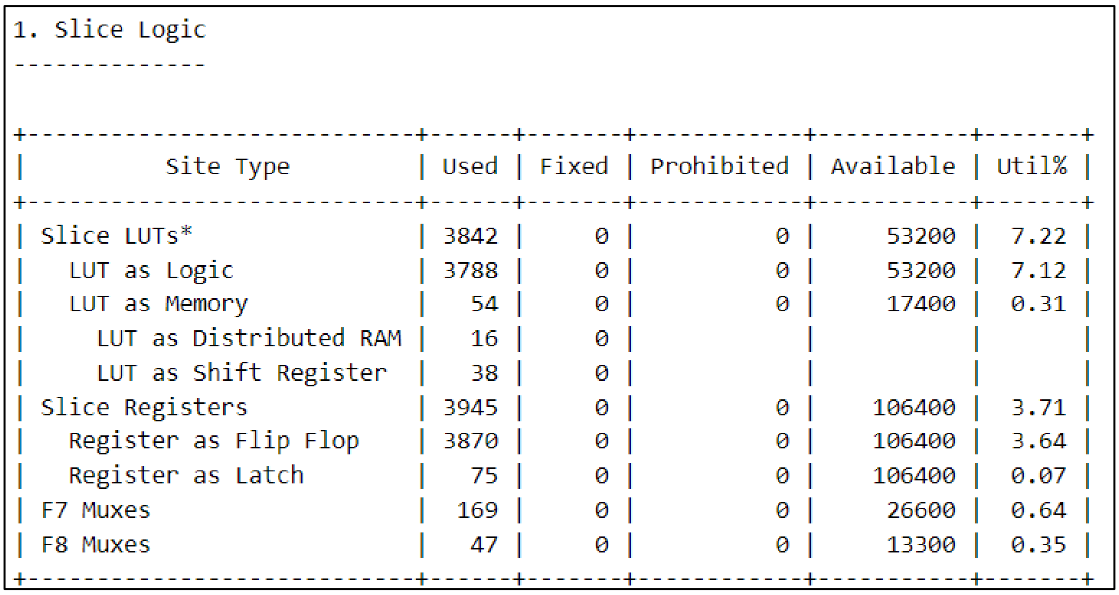


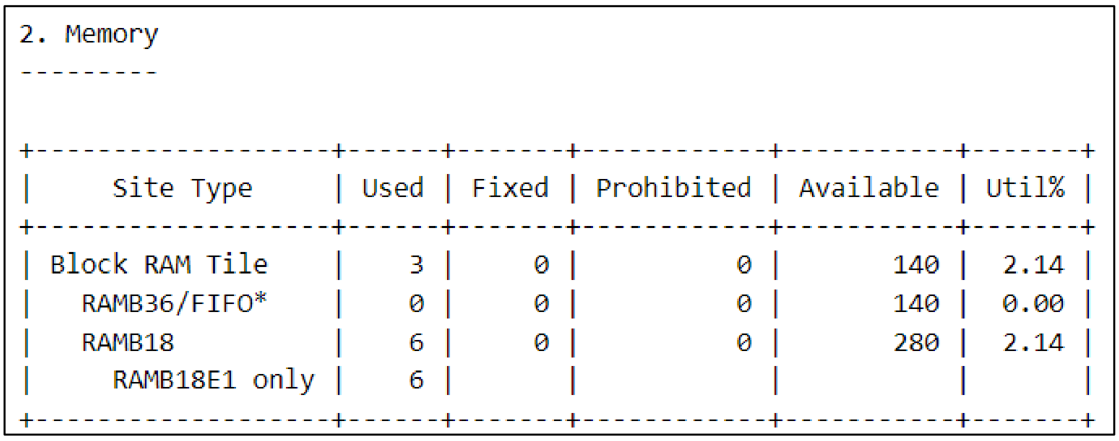
Fig 1 block diagram

2. FPGA Utilization

(1). Caravel utilization

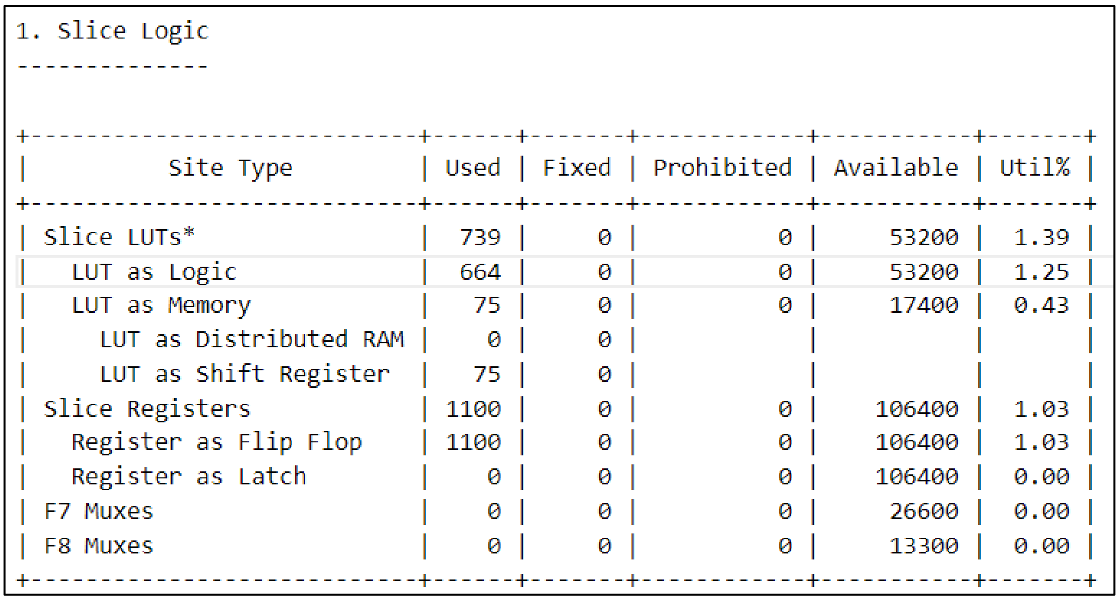


```
1. Slice Logic
--------------

+----------------------------+------+-------+------------+-----------+-------+
|          Site Type         | Used | Fixed | Prohibited | Available | Util% |
+----------------------------+------+-------+------------+-----------+-------+
| Slice LUTs*                | 3842 |     0 |          0 |     53200 |  7.22 |
|   LUT as Logic             | 3788 |     0 |          0 |     53200 |  7.12 |
|   LUT as Memory            |   54 |     0 |          0 |     17400 |  0.31 |
|     LUT as Distributed RAM |   16 |     0 |            |           |       |
|     LUT as Shift Register  |   38 |     0 |            |           |       |
| Slice Registers            | 3945 |     0 |          0 |    106400 |  3.71 |
|   Register as Flip Flop    | 3870 |     0 |          0 |    106400 |  3.64 |
|   Register as Latch        |   75 |     0 |          0 |    106400 |  0.07 |
| F7 Muxes                   |  169 |     0 |          0 |     26600 |  0.64 |
| F8 Muxes                   |   47 |     0 |          0 |     13300 |  0.35 |
+----------------------------+------+-------+------------+-----------+-------+
```

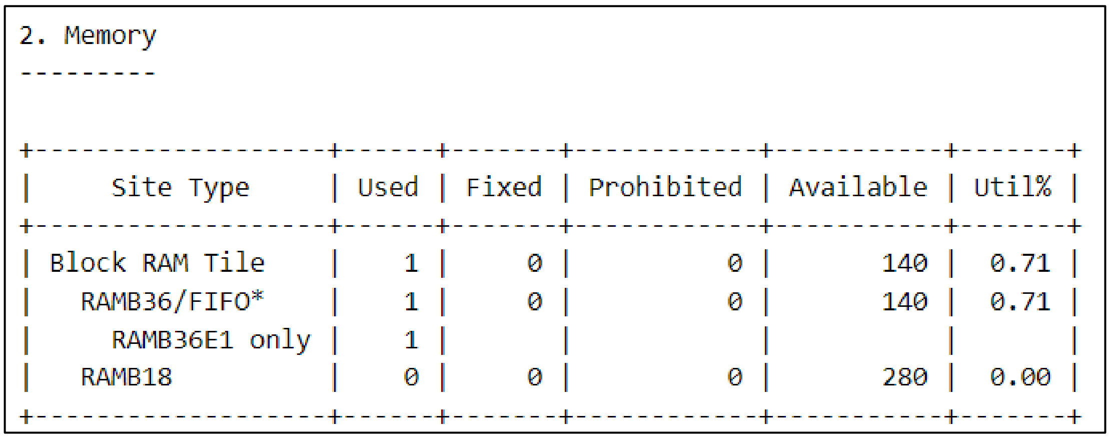


```
2. Memory
---------

+-------------------+------+-------+------------+-----------+-------+
|     Site Type     | Used | Fixed | Prohibited | Available | Util% |
+-------------------+------+-------+------------+-----------+-------+
| Block RAM Tile    |    3 |     0 |          0 |       140 |  2.14 |
|   RAMB36/FIFO*    |    0 |     0 |          0 |       140 |  0.00 |
|   RAMB18          |    6 |     0 |          0 |       280 |  2.14 |
|     RAMB18E1 only |    6 |       |            |           |       |
+-------------------+------+-------+------------+-----------+-------+
```

(2). Read_romcode utilization

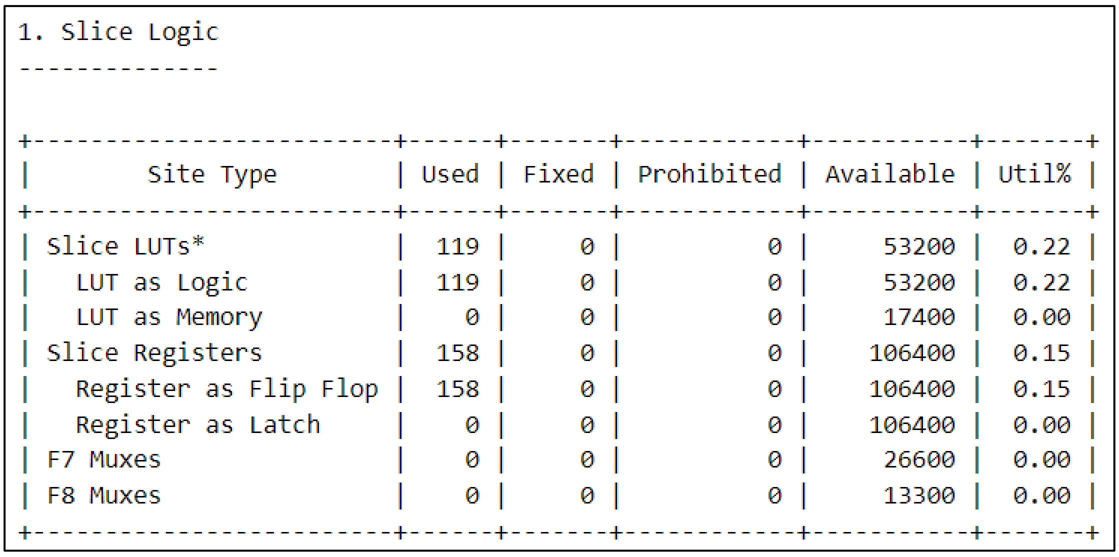


```
1. Slice Logic
--------------

+----------------------------+------+-------+------------+-----------+-------+
|          Site Type         | Used | Fixed | Prohibited | Available | Util% |
+----------------------------+------+-------+------------+-----------+-------+
| Slice LUTs*                |  739 |     0 |          0 |     53200 |  1.39 |
|   LUT as Logic             |  664 |     0 |          0 |     53200 |  1.25 |
|   LUT as Memory            |   75 |     0 |          0 |     17400 |  0.43 |
|     LUT as Distributed RAM |    0 |     0 |            |           |       |
|     LUT as Shift Register  |   75 |     0 |            |           |       |
| Slice Registers            | 1100 |     0 |          0 |    106400 |  1.03 |
|   Register as Flip Flop    | 1100 |     0 |          0 |    106400 |  1.03 |
|   Register as Latch        |    0 |     0 |          0 |    106400 |  0.00 |
| F7 Muxes                   |    0 |     0 |          0 |     26600 |  0.00 |
| F8 Muxes                   |    0 |     0 |          0 |     13300 |  0.00 |
+----------------------------+------+-------+------------+-----------+-------+
```

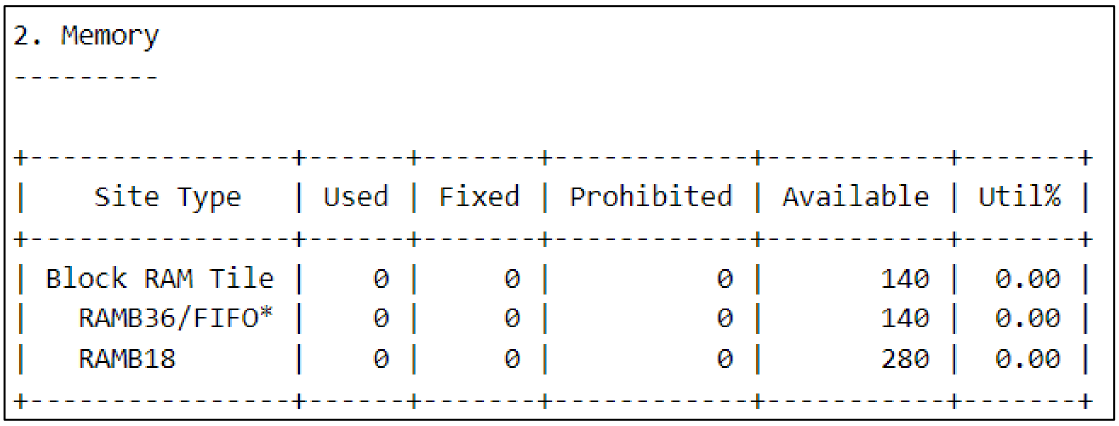


```
2. Memory
---------

+-------------------+------+-------+------------+-----------+-------+
|     Site Type     | Used | Fixed | Prohibited | Available | Util% |
+-------------------+------+-------+------------+-----------+-------+
| Block RAM Tile    |    1 |     0 |          0 |       140 |  0.71 |
|   RAMB36/FIFO*    |    1 |     0 |          0 |       140 |  0.71 |
|     RAMB36E1 only |    1 |       |            |           |       |
|   RAMB18          |    0 |     0 |          0 |       280 |  0.00 |
+-------------------+------+-------+------------+-----------+-------+
```

(3). Caravel_ps utilization

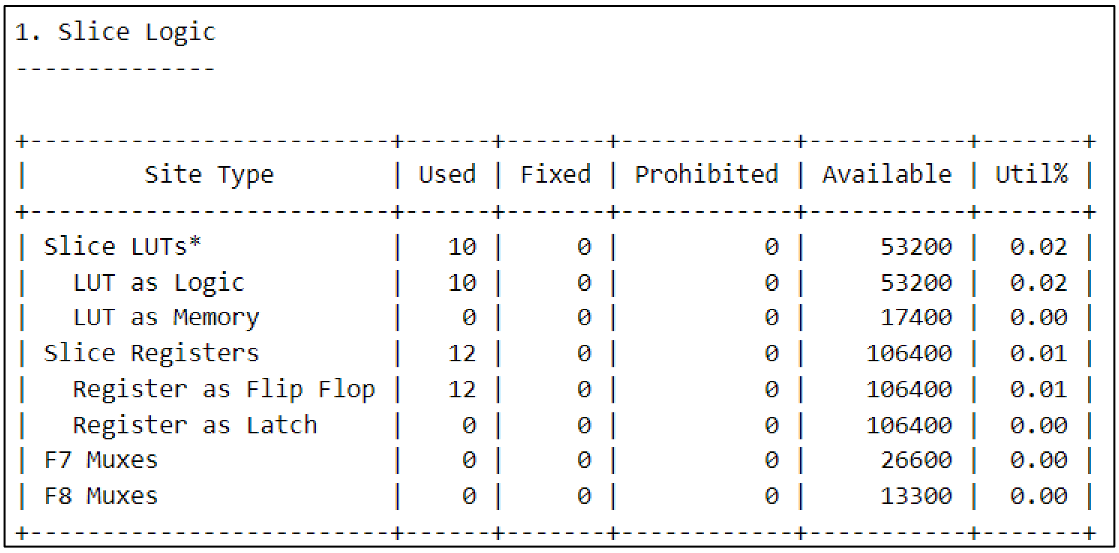


```
1. Slice Logic
--------------

+-------------------------+------+-------+------------+-----------+-------+
|        Site Type        | Used | Fixed | Prohibited | Available | Util% |
+-------------------------+------+-------+------------+-----------+-------+
| Slice LUTs*             |  119 |     0 |          0 |     53200 |  0.22 |
|   LUT as Logic          |  119 |     0 |          0 |     53200 |  0.22 |
|   LUT as Memory         |    0 |     0 |          0 |     17400 |  0.00 |
| Slice Registers         |  158 |     0 |          0 |    106400 |  0.15 |
|   Register as Flip Flop |  158 |     0 |          0 |    106400 |  0.15 |
|   Register as Latch     |    0 |     0 |          0 |    106400 |  0.00 |
| F7 Muxes                |    0 |     0 |          0 |     26600 |  0.00 |
| F8 Muxes                |    0 |     0 |          0 |     13300 |  0.00 |
+-------------------------+------+-------+------------+-----------+-------+
```

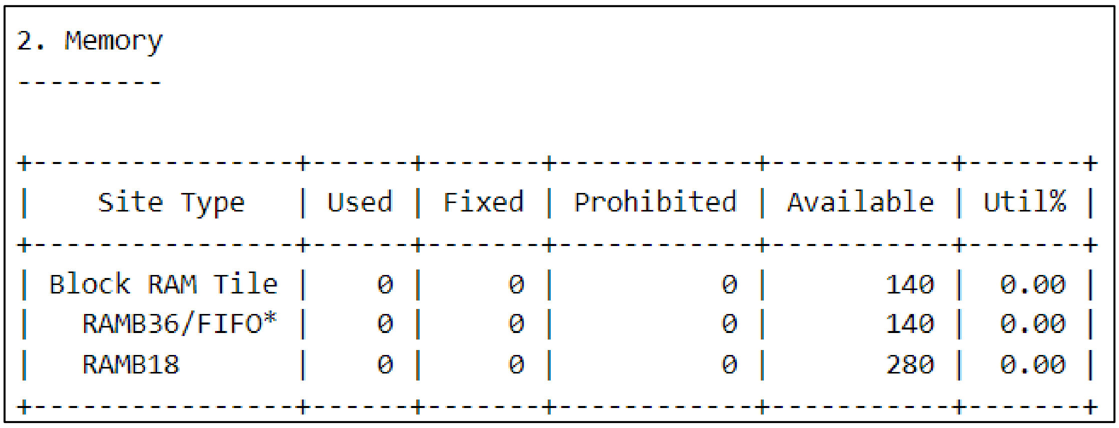


```
2. Memory
---------

+----------------+------+-------+------------+-----------+-------+
|   Site Type    | Used | Fixed | Prohibited | Available | Util% |
+----------------+------+-------+------------+-----------+-------+
| Block RAM Tile |    0 |     0 |          0 |       140 |  0.00 |
|   RAMB36/FIFO* |    0 |     0 |          0 |       140 |  0.00 |
|   RAMB18       |    0 |     0 |          0 |       280 |  0.00 |
+----------------+------+-------+------------+-----------+-------+
```

(4). Output_pin utilization

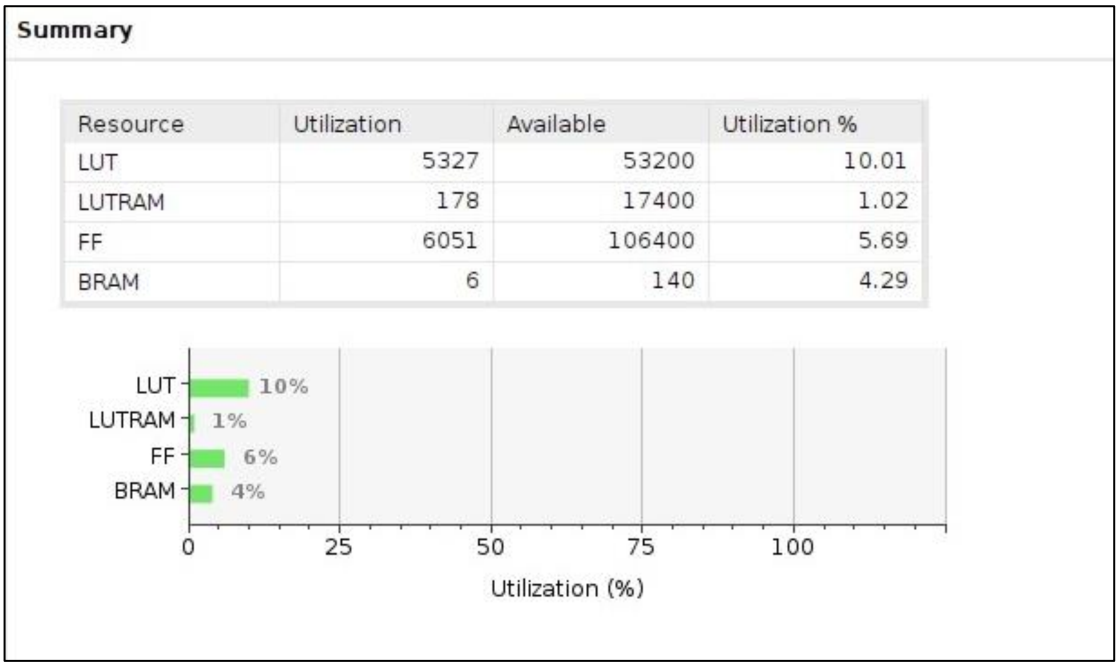


```
1. Slice Logic
--------------

+-------------------------+------+-------+------------+-----------+-------+
|        Site Type        | Used | Fixed | Prohibited | Available | Util% |
+-------------------------+------+-------+------------+-----------+-------+
| Slice LUTs*             |   10 |     0 |          0 |     53200 |  0.02 |
|   LUT as Logic          |   10 |     0 |          0 |     53200 |  0.02 |
|   LUT as Memory         |    0 |     0 |          0 |     17400 |  0.00 |
| Slice Registers         |   12 |     0 |          0 |    106400 |  0.01 |
|   Register as Flip Flop |   12 |     0 |          0 |    106400 |  0.01 |
|   Register as Latch     |    0 |     0 |          0 |    106400 |  0.00 |
| F7 Muxes                |    0 |     0 |          0 |     26600 |  0.00 |
| F8 Muxes                |    0 |     0 |          0 |     13300 |  0.00 |
+-------------------------+------+-------+------------+-----------+-------+
```



```
2. Memory
---------

+----------------+------+-------+------------+-----------+-------+
|   Site Type    | Used | Fixed | Prohibited | Available | Util% |
+----------------+------+-------+------------+-----------+-------+
| Block RAM Tile |    0 |     0 |          0 |       140 |  0.00 |
|   RAMB36/FIFO* |    0 |     0 |          0 |       140 |  0.00 |
|   RAMB18       |    0 |     0 |          0 |       280 |  0.00 |
+----------------+------+-------+------------+-----------+-------+
```

(5). Utilization 總整理（counter）

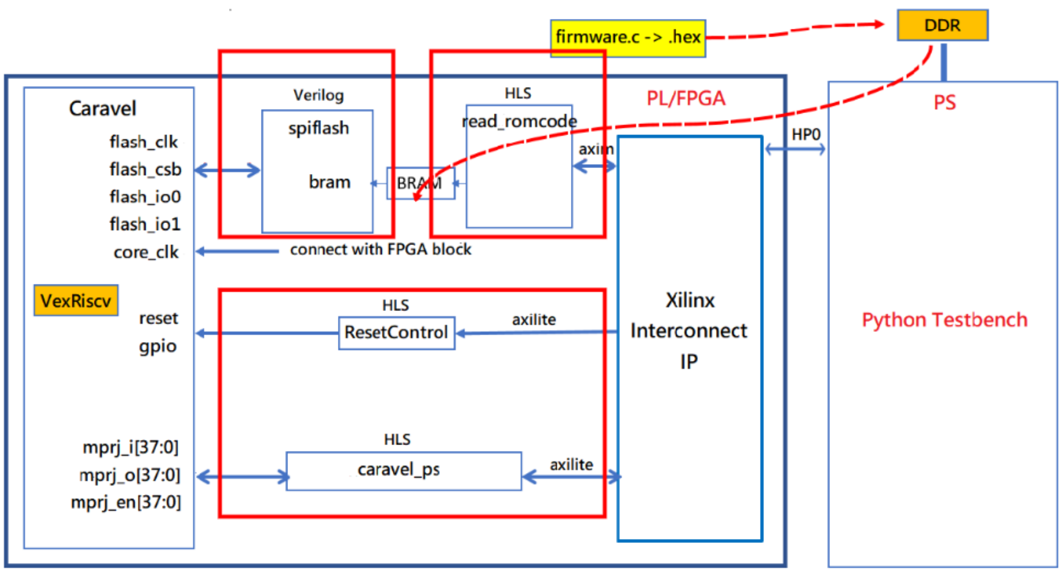


Summary

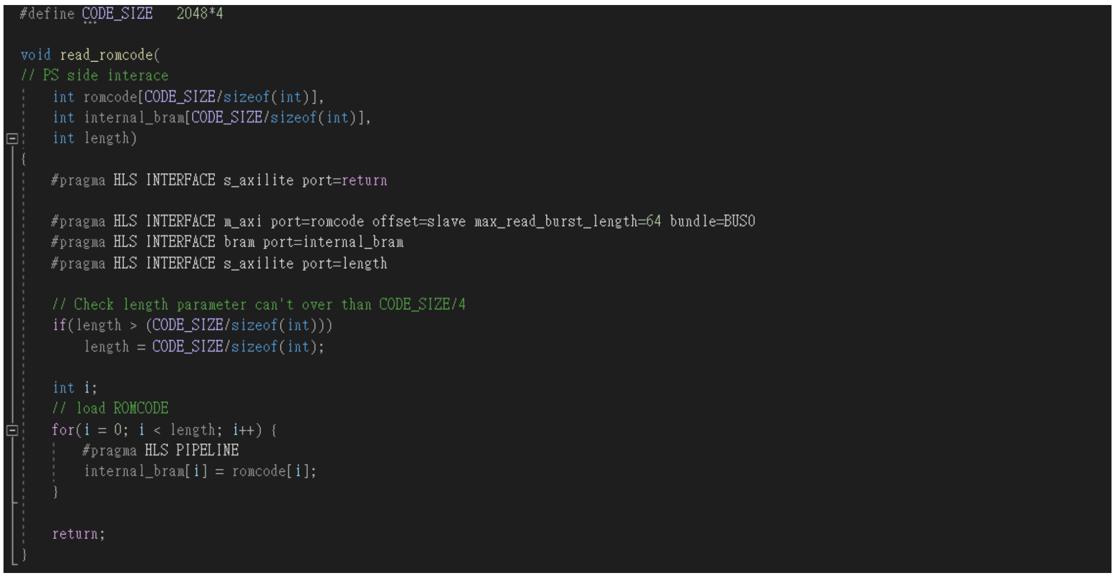
Resource	Utilization	Available	Utilization %
LUT	5327	53200	10.01
LUTRAM	178	17400	1.02
FF	6051	106400	5.69
BRAM	6	140	4.29

3. Explain the function of IP in this design

在底下，我們主要說明的是 Github 上 labi 的說明 PDF 檔（/caravel-soc_fpga-lab/labi/lab5-caravel FPGA.pdf）中下圖中有框起來的這 4 個 module 的功能：



(1). read_romcode



```
#define CODE_SIZE   2048*4

void read_romcode(
// PS side interace
    int romcode[CODE_SIZE/sizeof(int)],
    int internal_bram[CODE_SIZE/sizeof(int)],
    int length)
{
   #pragma HLS INTERFACE s_axilite port=return

   #pragma HLS INTERFACE m_axi port=romcode offset=slave max_read_burst_length=64 bundle=BUS0
   #pragma HLS INTERFACE bram port=internal_bram
   #pragma HLS INTERFACE s_axilite port=length

    // Check length parameter can't over than CODE_SIZE/4
    if(length > (CODE_SIZE/sizeof(int)))
        length = CODE_SIZE/sizeof(int);

    int i;
    // load ROMCODE
    for(i = 0; i < length; i++) {
        #pragma HLS PIPELINE
        internal_bram[i] = romcode[i];
    }

    return;
}
```

Fig 2 read_romcode.cpp

在執行 run_vitis.sh 時，會透過 Vitis_HLS 將/labi/vitis_hls_project/hls_read_romcode /src/read_romcode.cpp 進行高階合成產生 IP 並 export 成 Vivado 軟體的相容格式。因此，read_ROMcode 這個 module 的主要功能及行為定義在 read_romcode.cpp 中，如上圖所示。其中的 romcode[]這個 array 主要存放 software code 經過 compile 而得到的 firmware code（binary code），是由 PS side 的 DDR memory input 而來的，而 internal_bram[]這個 array 則是對應到與 BRAM 之間的接口（interface），

這個 interface 之間的 protocol 是使用”bram”的 interface，因此圖中使用

#pragma HLS INTERFACE bram port=internal_bram

這個#pragma 來限制合成方式。上述兩個 array 的大小皆被限制為 8KB，因此只能存放”8KB/sizeof(int)”個整數。

圖中的”length”則是 code 的大小，也就是 binary file(即 compile 後產生的.hex 檔)中以整數為單位的長度，這個資訊也會由 PS side 來提供，interface 是使用 AXI-Lite 來提供資訊。由於 array 的大小被限制在 8KB 以內，因此當 code size 大過這個大小時（也就是圖中 ”if(length > (CODE_SIZE/sizeof(int)))” 條件判斷式），就只能領到 8KB 的大小，因此 length 就會被限制在” CODE_SIZE/sizeof(int)”。

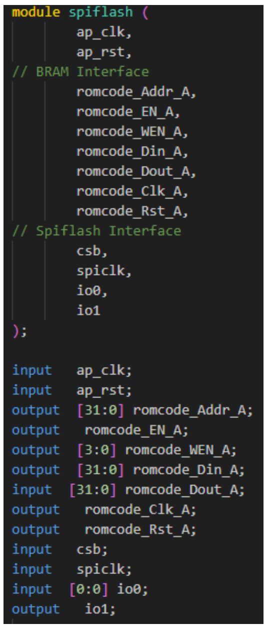
接者就是此 module 最主要的功能：將 ROM code 存放至 BRAM 中，也就是將這些 code 一行行放入與 BRAM 之間的 interface 讓 BRAM 去存取，因此使用

```
for(i = 0; i < length; i++) {
    internal_bram[i] = romcode[i];
}
```

來達成，並使用 pipeline 的#pragma 來加速存取。

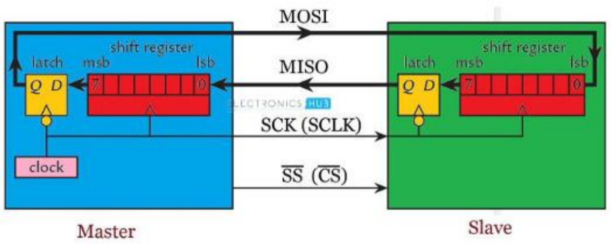
(2). spiflash

spiflash 位於/caravel-soc_fpga-lab/labi/vvd_srcs/spiflash.v，為 Caravel SoC 要向 BRAM 索取 CPU 要執行的 firmware code 時的橋樑，因此它有兩個 interface — BRAM 以及 Caravel SoC (spiflash)，如下圖所示：



```
module spiflash (
        ap_clk,
        ap_rst,
// BRAM Interface
        romcode_Addr_A,
        romcode_EN_A,
        romcode_WEN_A,
        romcode_Din_A,
        romcode_Dout_A,
        romcode_Clk_A,
        romcode_Rst_A,
// Spiflash Interface
        csb,
        spiclk,
        io0,
        io1
);

input   ap_clk;
input   ap_rst;
output  [31:0] romcode_Addr_A;
output   romcode_EN_A;
output  [3:0] romcode_WEN_A;
output  [31:0] romcode_Din_A;
input  [31:0] romcode_Dout_A;
output   romcode_Clk_A;
output   romcode_Rst_A;
input   csb;
input   spiclk;
input  [0:0] io0;
output   io1;
```

由於 BRAM 中的 code 為 read-only，因此 romcode_WEN_A 一直為 0，而要讀取 BRAM 的 address 則是依照 spi_address 來提供，BRAM 輸出的 Data_out (romcode_Dout_A) 則依照 address(byte-address 的形式)取出並存到 memory[7:0]中，共存了 1 個 byte 起來。接著按照下圖的方式輸出及輸入：



上圖為 labi 的說明 PDF 檔（/caravel-soc_fpga-lab/labi/lab5-caravel FPGA.pdf）中的介紹圖。Spi input 進來的 io0 這個 bit 會被放到 buffer 的末端，形成 shift register，buffer 會依據 bytecount（每個 cycle bitcount 會增加 1，而當 bitcount 滿 7 後，就會 trigger bytecount 增加 1）而決定要放到 spi_addr 的哪一個 byte 位置。由於只支援 spi_cmd == 'h03（即 read command），BRAM 會依據 address 依序吐出 ROMcode，並且再透過 shift register 的方式將 memory[7:0]的值依序放至 outbuf 中，再一個 bit 接著一個 bit 輸出到 io1（即 outbuf[7]位置）中輸出給 Caravel SoC。

(3). ResetControl

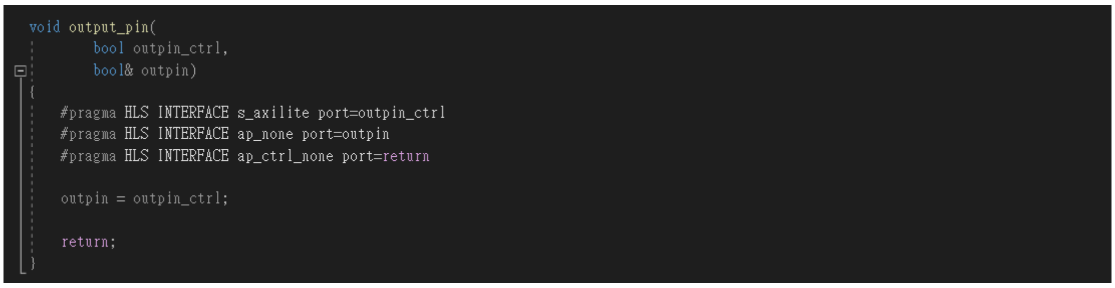
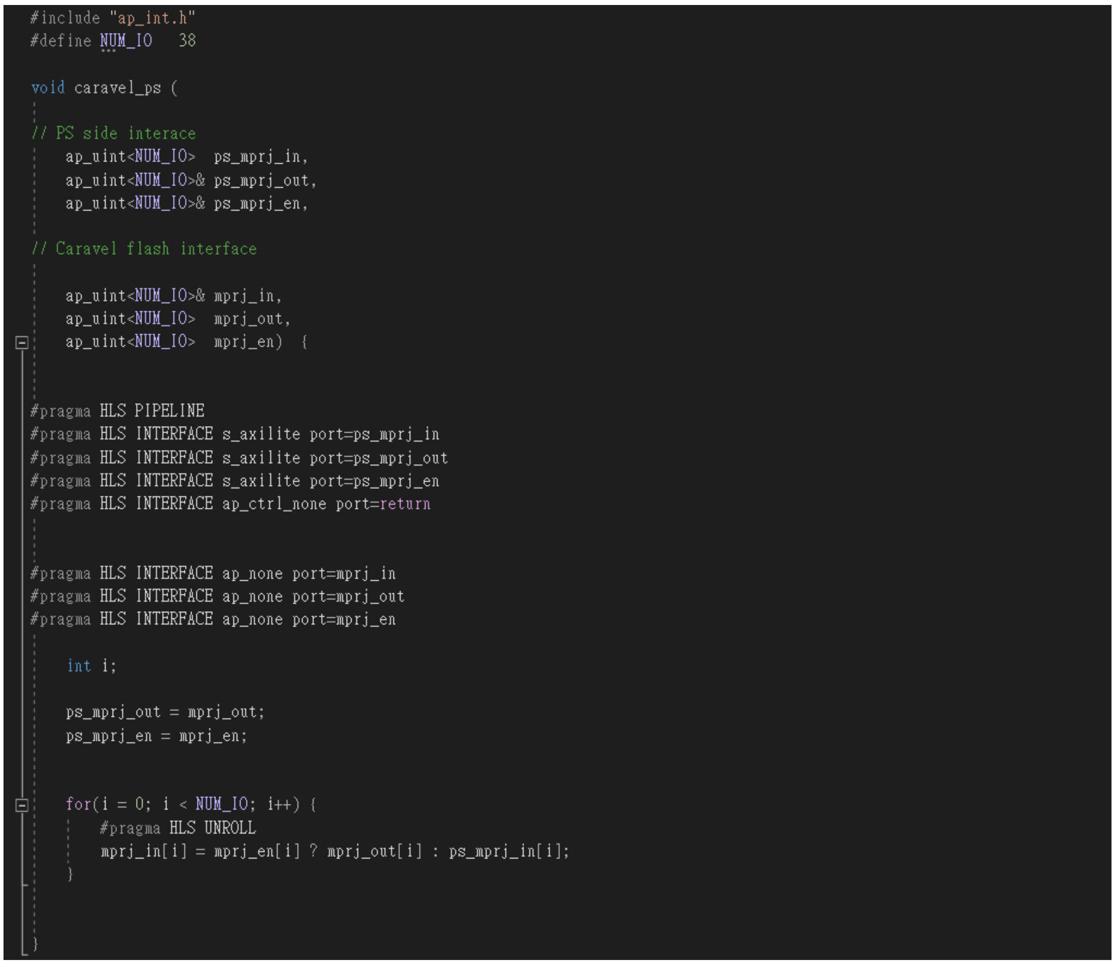

```
void output_pin(
        bool outpin_ctrl,
        bool& outpin)
{
    #pragma HLS INTERFACE s_axilite port=outpin_ctrl
    #pragma HLS INTERFACE ap_none port=outpin
    #pragma HLS INTERFACE ap_ctrl_none port=return

    outpin = outpin_ctrl;

    return;
}
```

Fig 3 output_pin.cpp

在執行 run_vitis.sh 時，會透過 Vitis_HLS 將/labi/vitis_hls_project/hls_output_pin/src/output_pin.cpp 進行高階合成產生 IP 並 export 成 Vivado 軟體的相容格式。因此，ResetControl 這個 module 的主要功能及行為定義在 output_pin.cpp 中，如上圖所示。此 module 的主要功能用於將一個布林值 outpin_ctrl 複製給另一個布林值引用 outpin。

“outpin = outpin_ctrl”這一行的作用是將 outpin_ctrl 的值賦給 outpin，這樣 outpin 就會擁有和 outpin_ctrl 相同的布林值。

(4). caravel_ps



```
#include "ap_int.h"
#define NUM_IO   38

void caravel_ps (

// PS side interace
    ap_uint<NUM_IO>  ps_mprj_in,
    ap_uint<NUM_IO>& ps_mprj_out,
    ap_uint<NUM_IO>& ps_mprj_en,

// Caravel flash interface

    ap_uint<NUM_IO>& mprj_in,
    ap_uint<NUM_IO>  mprj_out,
    ap_uint<NUM_IO>  mprj_en)  {


#pragma HLS PIPELINE
#pragma HLS INTERFACE s_axilite port=ps_mprj_in
#pragma HLS INTERFACE s_axilite port=ps_mprj_out
#pragma HLS INTERFACE s_axilite port=ps_mprj_en
#pragma HLS INTERFACE ap_ctrl_none port=return


#pragma HLS INTERFACE ap_none port=mprj_in
#pragma HLS INTERFACE ap_none port=mprj_out
#pragma HLS INTERFACE ap_none port=mprj_en

    int i;

    ps_mprj_out = mprj_out;
    ps_mprj_en = mprj_en;


    for(i = 0; i < NUM_IO; i++) {
        #pragma HLS UNROLL
        mprj_in[i] = mprj_en[i] ? mprj_out[i] : ps_mprj_in[i];
    }


}
```

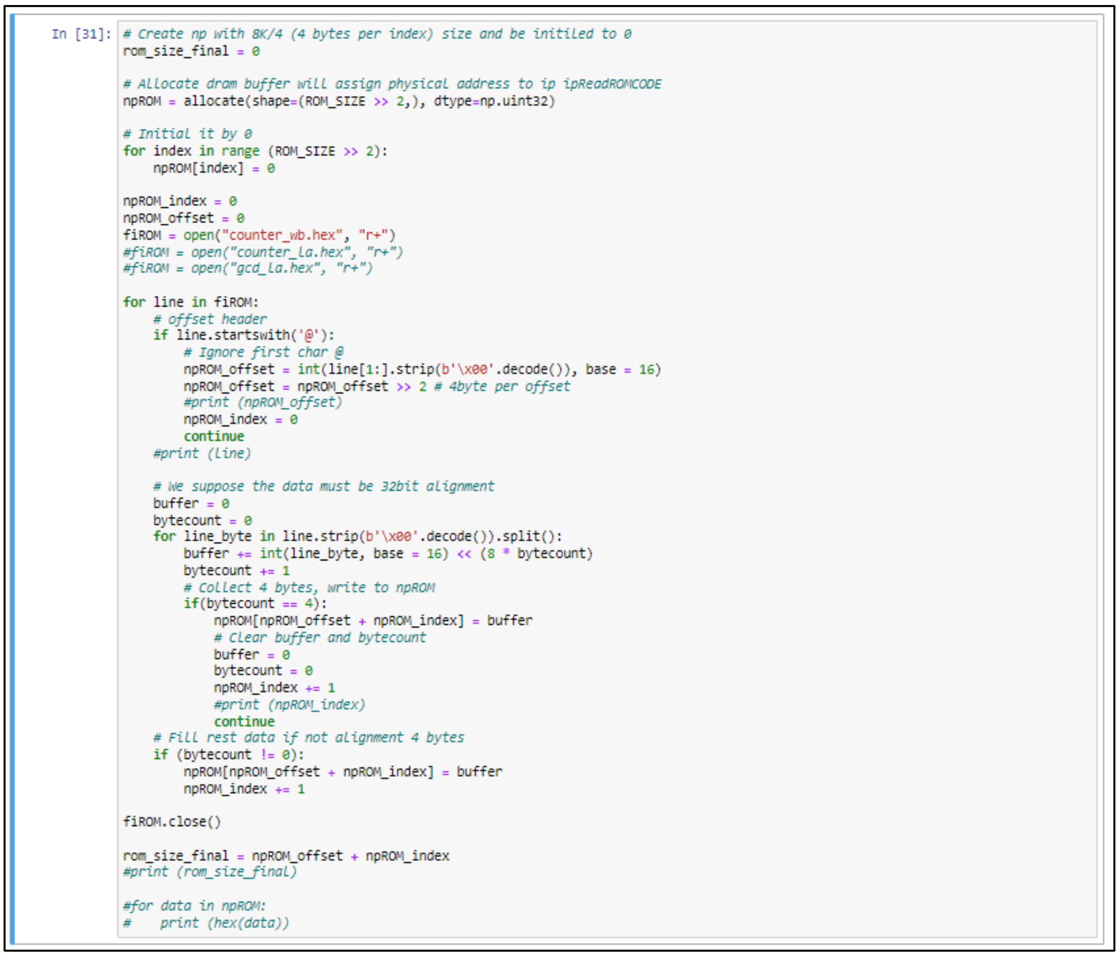
Fig 4 caravel_ps.cpp

在執行 run_vitis.sh 時，會透過 Vitis_HLS 將/labi/vitis_hls_project/hls_caravel_ps /src/caravel_ps.cpp 進行高階合成產生 IP 並 export 成 Vivado 軟體的相容格式。因此，caravel_ps 這個 module 的主要功能及行為定義在 caravel_ps.cpp 中，如上圖所示。此 module 的主要功能為提供 PS CPU AXI Lite 介面以讀取 MPRJ_IO/OUT/EN bits，並透過 HLS 實作並匯出 IP 以供 Vivado 專案使用。
PS 端的輸入和輸出接口: ps_mprj_in、ps_mprj_out、ps_mprj_en
Caravel flash 端的輸入和輸出接口: mprj_in、mprj_out、mprj_en

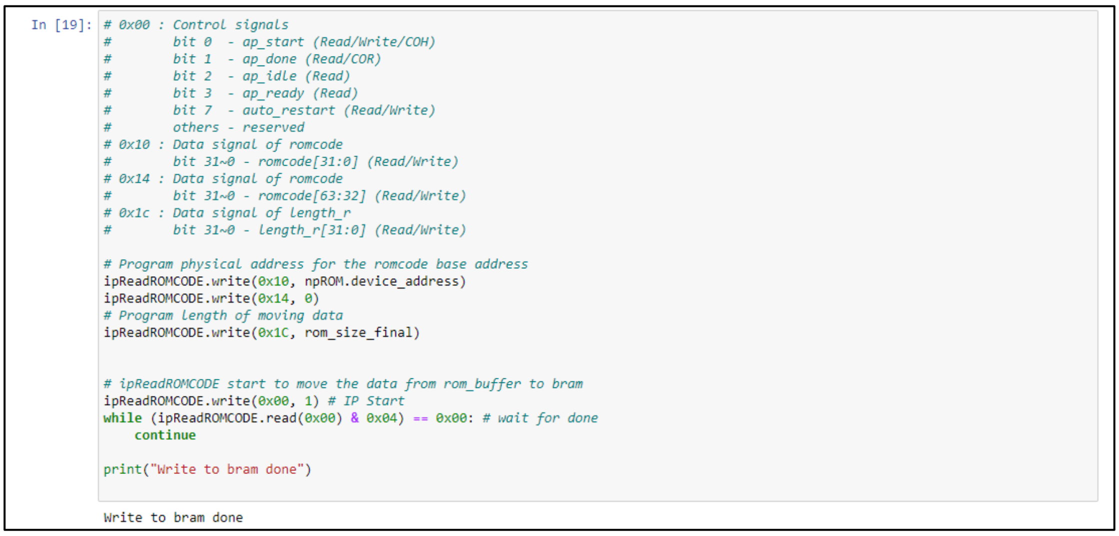
函數將值從 Caravel 閃存端(mprj_out 和 mprj_en)複製到 PS 端(ps_mprj_out 和 ps_mprj_en)。如果 mprj_en[i] 為真，則將 mprj_out[i] 複製到 mprj_in[i]；否則，將 ps_mprj_in[i] 複製到 mprj_in[i]。#pragma HLS UNROLL 指令表明這個循環可以展開以提高性能。

4. Run these workload on caravel FPGA

counter_wb.hex

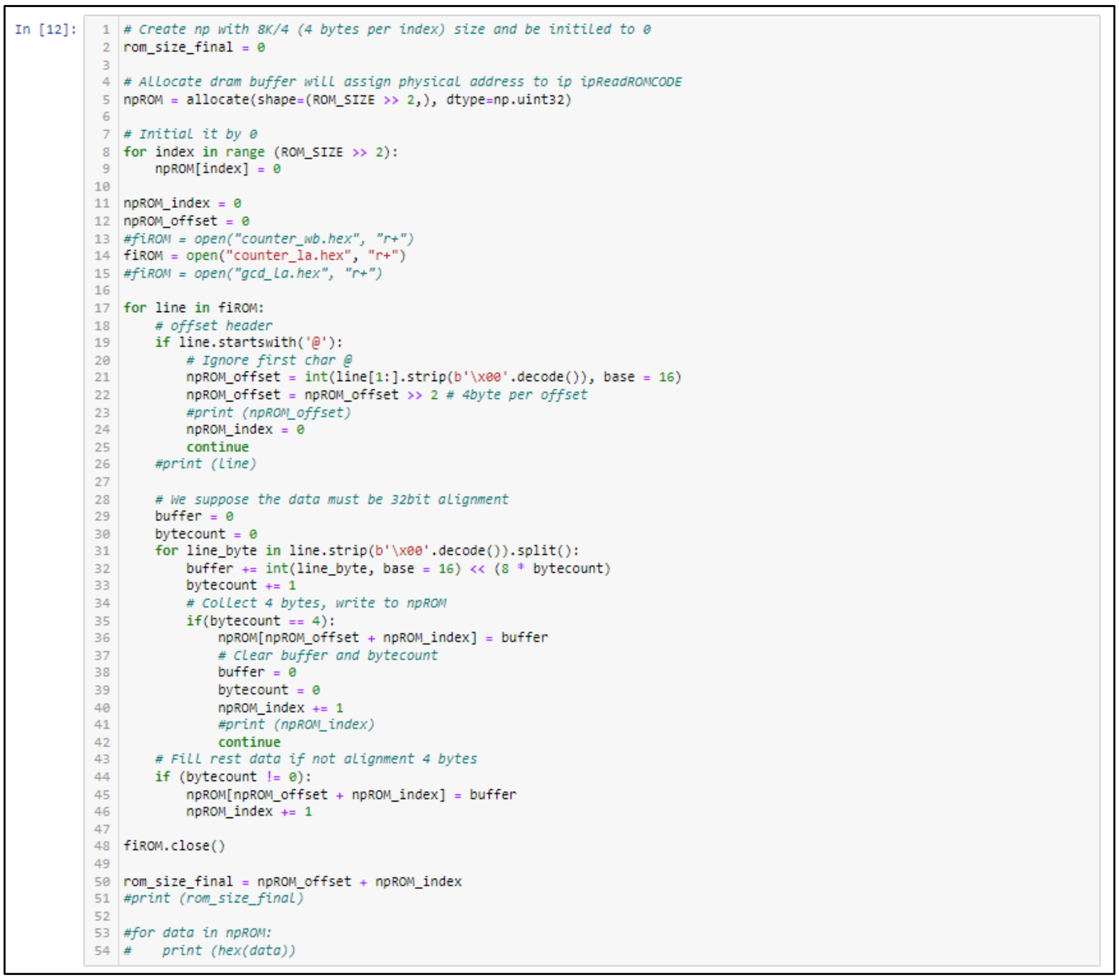


```
In [31]: # Create np with 8K/4 (4 bytes per index) size and be initiled to 0
rom_size_final = 0

# Allocate dram buffer will assign physical address to ip ipReadROMCODE
npROM = allocate(shape=(ROM_SIZE >> 2,), dtype=np.uint32)

# Initial it by 0
for index in range (ROM_SIZE >> 2):
    npROM[index] = 0

npROM_index = 0
npROM_offset = 0
fiROM = open("counter_wb.hex", "r+")
#fiROM = open("counter_la.hex", "r+")
#fiROM = open("gcd_la.hex", "r+")

for line in fiROM:
    # offset header
    if line.startswith('@'):
        # Ignore first char @
        npROM_offset = int(line[1:].strip(b'\x00'.decode()), base = 16)
        npROM_offset = npROM_offset >> 2 # 4byte per offset
        #print (npROM_offset)
        npROM_index = 0
        continue
    #print (line)

    # We suppose the data must be 32bit alignment
    buffer = 0
    bytecount = 0
    for line_byte in line.strip(b'\x00'.decode()).split():
        buffer += int(line_byte, base = 16) << (8 * bytecount)
        bytecount += 1
        # Collect 4 bytes, write to npROM
        if(bytecount == 4):
            npROM[npROM_offset + npROM_index] = buffer
            # Clear buffer and bytecount
            buffer = 0
            bytecount = 0
            npROM_index += 1
            #print (npROM_index)
            continue
    # Fill rest data if not alignment 4 bytes
    if (bytecount != 0):
        npROM[npROM_offset + npROM_index] = buffer
        npROM_index += 1

fiROM.close()

rom_size_final = npROM_offset + npROM_index
#print (rom_size_final)

#for data in npROM:
#    print (hex(data))
```



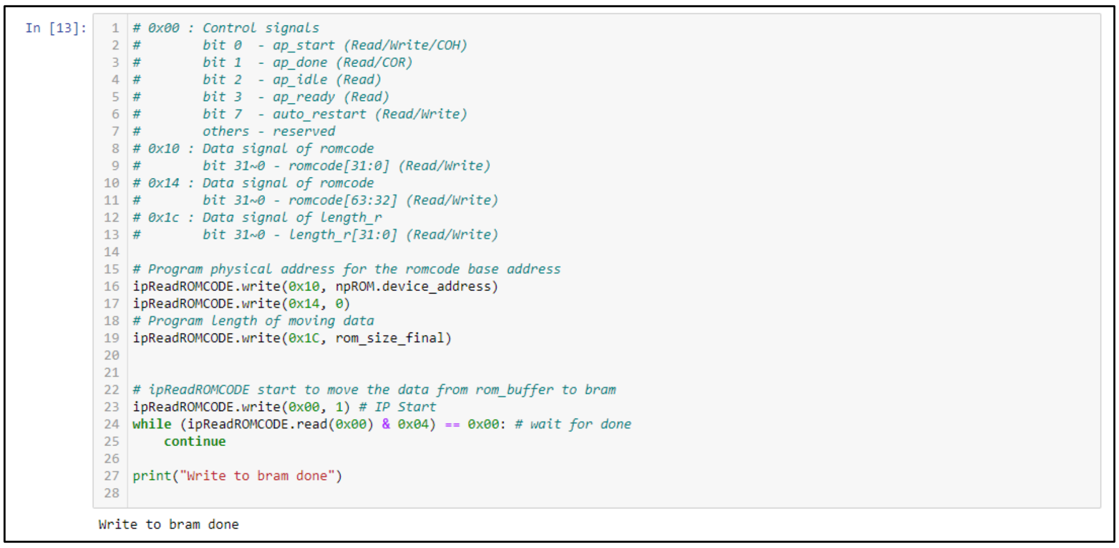
```
In [19]: # 0x00 : Control signals
#        bit 0  - ap_start (Read/Write/COH)
#        bit 1  - ap_done (Read/COR)
#        bit 2  - ap_idle (Read)
#        bit 3  - ap_ready (Read)
#        bit 7  - auto_restart (Read/Write)
#        others - reserved
# 0x10 : Data signal of romcode
#        bit 31~0 - romcode[31:0] (Read/Write)
# 0x14 : Data signal of romcode
#        bit 31~0 - romcode[63:32] (Read/Write)
# 0x1c : Data signal of length_r
#        bit 31~0 - length_r[31:0] (Read/Write)

# Program physical address for the romcode base address
ipReadROMCODE.write(0x10, npROM.device_address)
ipReadROMCODE.write(0x14, 0)
# Program Length of moving data
ipReadROMCODE.write(0x1C, rom_size_final)


# ipReadROMCODE start to move the data from rom_buffer to bram
ipReadROMCODE.write(0x00, 1) # IP Start
while (ipReadROMCODE.read(0x00) & 0x04) == 0x00: # wait for done
    continue

print("Write to bram done")

Write to bram done
```

counter_la.hex



In [12]:

```python
# Create np with 8K/4 (4 bytes per index) size and be initiled to 0
rom_size_final = 0

# Allocate dram buffer will assign physical address to ip ipReadROMCODE
npROM = allocate(shape=(ROM_SIZE >> 2,), dtype=np.uint32)

# Initial it by 0
for index in range (ROM_SIZE >> 2):
    npROM[index] = 0

npROM_index = 0
npROM_offset = 0
#fiROM = open("counter_wb.hex", "r+")
fiROM = open("counter_la.hex", "r+")
#fiROM = open("gcd_la.hex", "r+")

for line in fiROM:
    # offset header
    if line.startswith('@'):
        # Ignore first char @
        npROM_offset = int(line[1:].strip(b'\x00'.decode()), base = 16)
        npROM_offset = npROM_offset >> 2 # 4byte per offset
        #print (npROM_offset)
        npROM_index = 0
        continue
    #print (line)

    # We suppose the data must be 32bit alignment
    buffer = 0
    bytecount = 0
    for line_byte in line.strip(b'\x00'.decode()).split():
        buffer += int(line_byte, base = 16) << (8 * bytecount)
        bytecount += 1
        # Collect 4 bytes, write to npROM
        if(bytecount == 4):
            npROM[npROM_offset + npROM_index] = buffer
            # Clear buffer and bytecount
            buffer = 0
            bytecount = 0
            npROM_index += 1
            #print (npROM_index)
            continue
    # Fill rest data if not alignment 4 bytes
    if (bytecount != 0):
        npROM[npROM_offset + npROM_index] = buffer
        npROM_index += 1

fiROM.close()

rom_size_final = npROM_offset + npROM_index
#print (rom_size_final)

#for data in npROM:
#    print (hex(data))
```

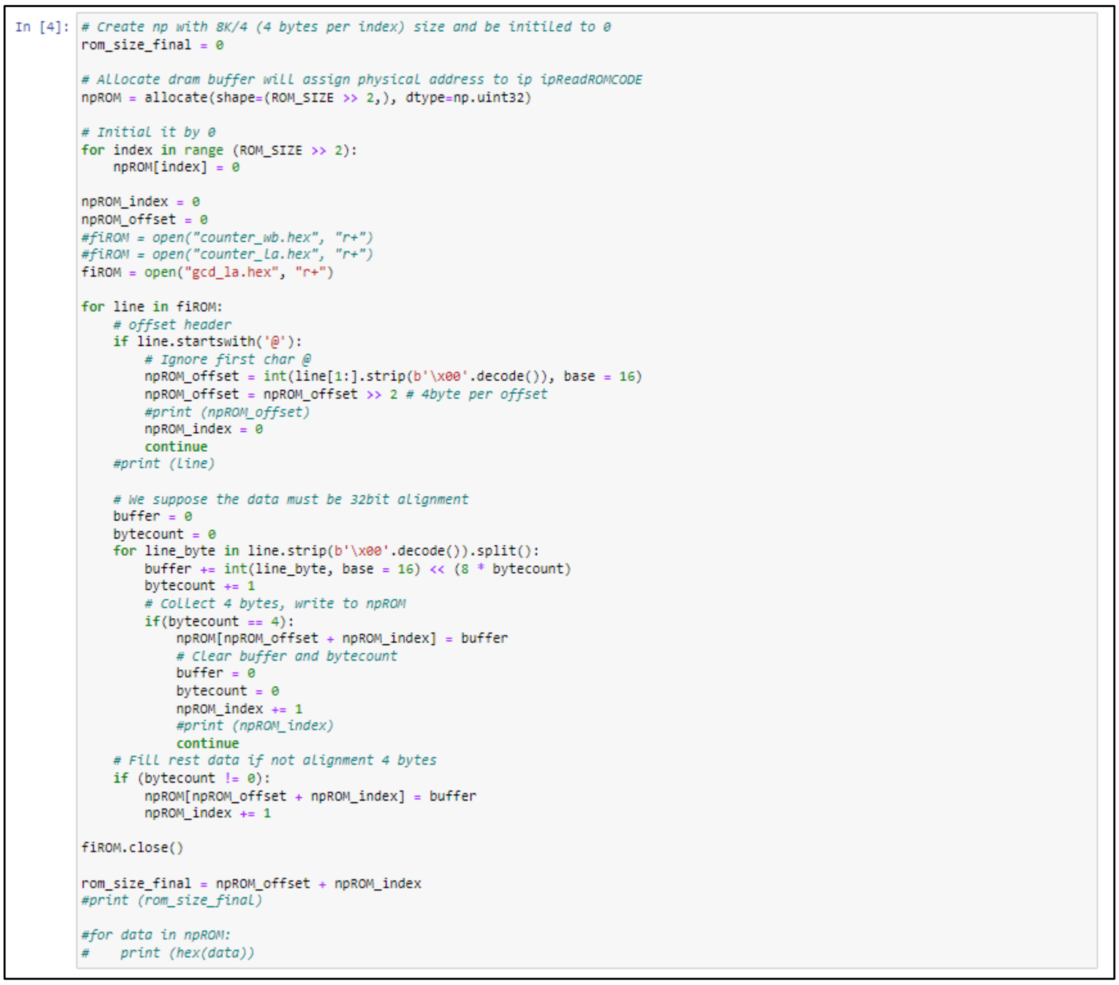


In [13]:

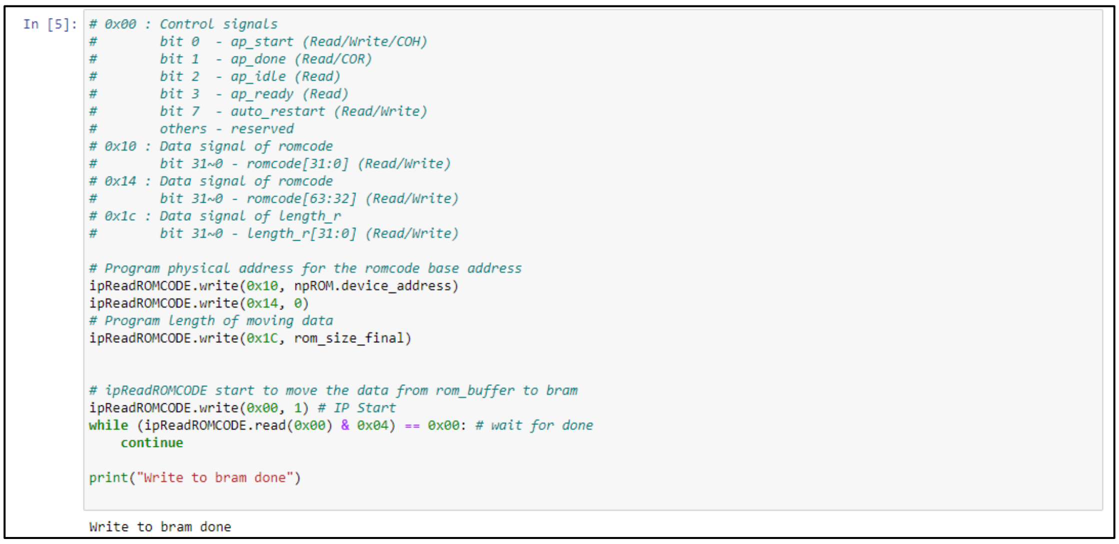
```python
# 0x00 : Control signals
#        bit 0  - ap_start (Read/Write/COH)
#        bit 1  - ap_done (Read/COR)
#        bit 2  - ap_idle (Read)
#        bit 3  - ap_ready (Read)
#        bit 7  - auto_restart (Read/Write)
#        others - reserved
# 0x10 : Data signal of romcode
#        bit 31~0 - romcode[31:0] (Read/Write)
# 0x14 : Data signal of romcode
#        bit 31~0 - romcode[63:32] (Read/Write)
# 0x1c : Data signal of length_r
#        bit 31~0 - length_r[31:0] (Read/Write)

# Program physical address for the romcode base address
ipReadROMCODE.write(0x10, npROM.device_address)
ipReadROMCODE.write(0x14, 0)
# Program Length of moving data
ipReadROMCODE.write(0x1C, rom_size_final)


# ipReadROMCODE start to move the data from rom_buffer to bram
ipReadROMCODE.write(0x00, 1) # IP Start
while (ipReadROMCODE.read(0x00) & 0x04) == 0x00: # wait for done
    continue

print("Write to bram done")

```

```
Write to bram done
```

gcd_la.hex



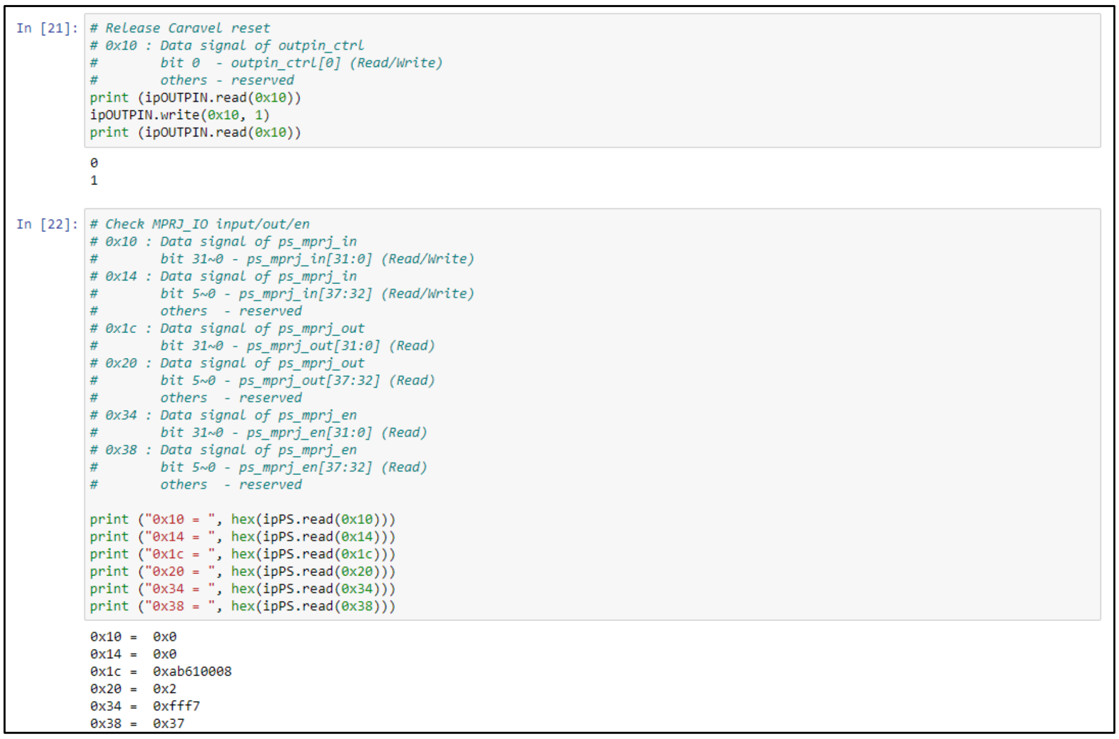
```
In [4]: # Create np with 8K/4 (4 bytes per index) size and be initiled to 0
        rom_size_final = 0

        # Allocate dram buffer will assign physical address to ip ipReadROMCODE
        npROM = allocate(shape=(ROM_SIZE >> 2,), dtype=np.uint32)

        # Initial it by 0
        for index in range (ROM_SIZE >> 2):
            npROM[index] = 0

        npROM_index = 0
        npROM_offset = 0
        #fiROM = open("counter_wb.hex", "r+")
        #fiROM = open("counter_la.hex", "r+")
        fiROM = open("gcd_la.hex", "r+")

        for line in fiROM:
            # offset header
            if line.startswith('@'):
                # Ignore first char @
                npROM_offset = int(line[1:].strip(b'\x00'.decode()), base = 16)
                npROM_offset = npROM_offset >> 2 # 4byte per offset
                #print (npROM_offset)
                npROM_index = 0
                continue
            #print (line)

            # We suppose the data must be 32bit alignment
            buffer = 0
            bytecount = 0
            for line_byte in line.strip(b'\x00'.decode()).split():
                buffer += int(line_byte, base = 16) << (8 * bytecount)
                bytecount += 1
                # Collect 4 bytes, write to npROM
                if(bytecount == 4):
                    npROM[npROM_offset + npROM_index] = buffer
                    # Clear buffer and bytecount
                    buffer = 0
                    bytecount = 0
                    npROM_index += 1
                    #print (npROM_index)
                    continue
            # Fill rest data if not alignment 4 bytes
            if (bytecount != 0):
                npROM[npROM_offset + npROM_index] = buffer
                npROM_index += 1

        fiROM.close()

        rom_size_final = npROM_offset + npROM_index
        #print (rom_size_final)

        #for data in npROM:
        #    print (hex(data))
```



```
In [5]: # 0x00 : Control signals
        #        bit 0  - ap_start (Read/Write/COH)
        #        bit 1  - ap_done (Read/COR)
        #        bit 2  - ap_idle (Read)
        #        bit 3  - ap_ready (Read)
        #        bit 7  - auto_restart (Read/Write)
        #        others - reserved
        # 0x10 : Data signal of romcode
        #        bit 31~0 - romcode[31:0] (Read/Write)
        # 0x14 : Data signal of romcode
        #        bit 31~0 - romcode[63:32] (Read/Write)
        # 0x1c : Data signal of length_r
        #        bit 31~0 - length_r[31:0] (Read/Write)

        # Program physical address for the romcode base address
        ipReadROMCODE.write(0x10, npROM.device_address)
        ipReadROMCODE.write(0x14, 0)
        # Program Length of moving data
        ipReadROMCODE.write(0x1C, rom_size_final)


        # ipReadROMCODE start to move the data from rom_buffer to bram
        ipReadROMCODE.write(0x00, 1) # IP Start
        while (ipReadROMCODE.read(0x00) & 0x04) == 0x00: # wait for done
            continue

        print("Write to bram done")

        Write to bram done
```

5. Screenshot of Execution result on all workload

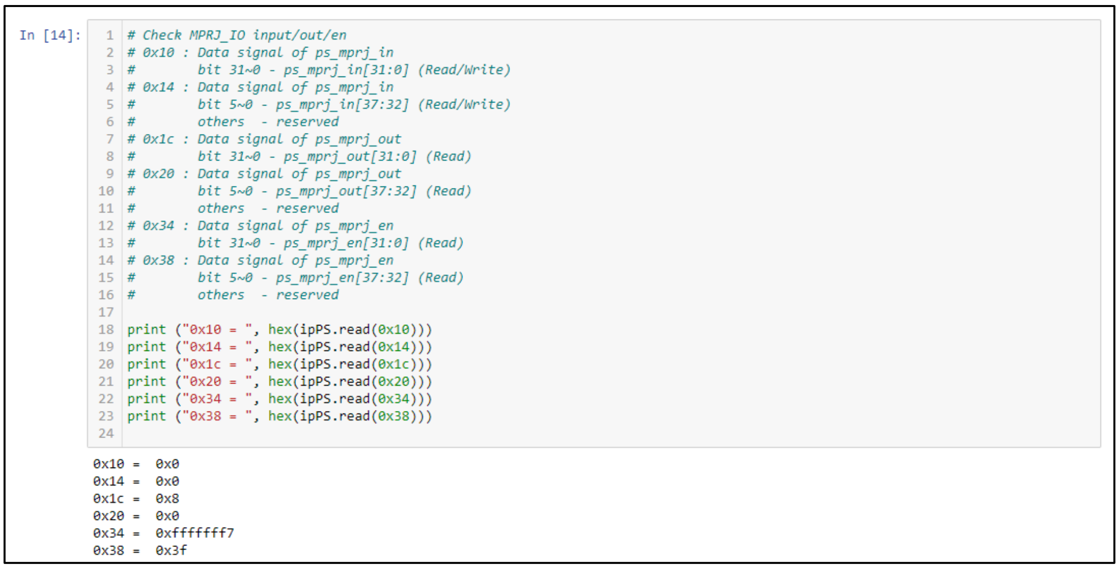
Counter_wb



```
In [21]: # Release Caravel reset
         # 0x10 : Data signal of outpin_ctrl
         #        bit 0  - outpin_ctrl[0] (Read/Write)
         #        others - reserved
         print (ipOUTPIN.read(0x10))
         ipOUTPIN.write(0x10, 1)
         print (ipOUTPIN.read(0x10))

         0
         1

In [22]: # Check MPRJ_IO input/out/en
         # 0x10 : Data signal of ps_mprj_in
         #        bit 31~0 - ps_mprj_in[31:0] (Read/Write)
         # 0x14 : Data signal of ps_mprj_in
         #        bit 5~0 - ps_mprj_in[37:32] (Read/Write)
         #        others  - reserved
         # 0x1c : Data signal of ps_mprj_out
         #        bit 31~0 - ps_mprj_out[31:0] (Read)
         # 0x20 : Data signal of ps_mprj_out
         #        bit 5~0 - ps_mprj_out[37:32] (Read)
         #        others  - reserved
         # 0x34 : Data signal of ps_mprj_en
         #        bit 31~0 - ps_mprj_en[31:0] (Read)
         # 0x38 : Data signal of ps_mprj_en
         #        bit 5~0 - ps_mprj_en[37:32] (Read)
         #        others  - reserved

         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 =  0xab610008
         0x20 =  0x2
         0x34 =  0xfff7
         0x38 =  0x37
```

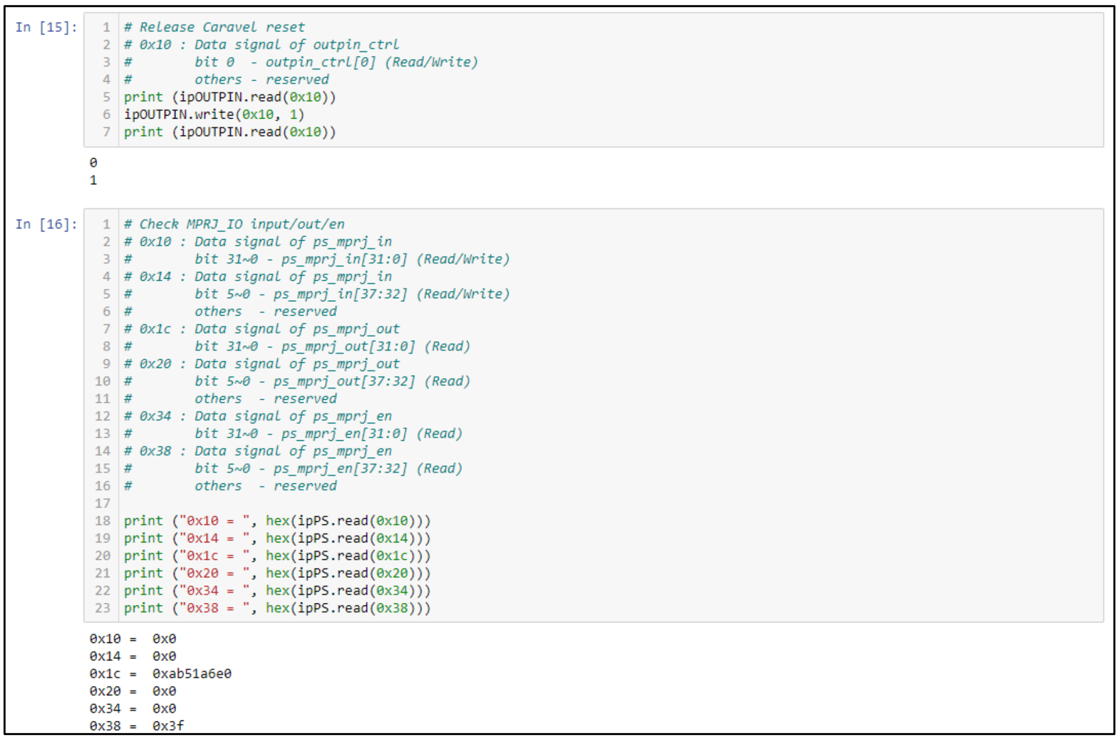
最終 0x1c 的位置的值最高 bit 為 AB61，符合期待值！

Counter_la



In [14]:
```python
# Check MPRJ_IO input/out/en
# 0x10 : Data signal of ps_mprj_in
#         bit 31~0 - ps_mprj_in[31:0] (Read/Write)
# 0x14 : Data signal of ps_mprj_in
#         bit 5~0 - ps_mprj_in[37:32] (Read/Write)
#         others  - reserved
# 0x1c : Data signal of ps_mprj_out
#         bit 31~0 - ps_mprj_out[31:0] (Read)
# 0x20 : Data signal of ps_mprj_out
#         bit 5~0 - ps_mprj_out[37:32] (Read)
#         others  - reserved
# 0x34 : Data signal of ps_mprj_en
#         bit 31~0 - ps_mprj_en[31:0] (Read)
# 0x38 : Data signal of ps_mprj_en
#         bit 5~0 - ps_mprj_en[37:32] (Read)
#         others  - reserved

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 =  0x8
0x20 =  0x0
0x34 =  0xfffffff7
0x38 =  0x3f
```



In [15]:
```python
# Release Caravel reset
# 0x10 : Data signal of outpin_ctrl
#         bit 0  - outpin_ctrl[0] (Read/Write)
#         others - reserved
print (ipOUTPIN.read(0x10))
ipOUTPIN.write(0x10, 1)
print (ipOUTPIN.read(0x10))
```

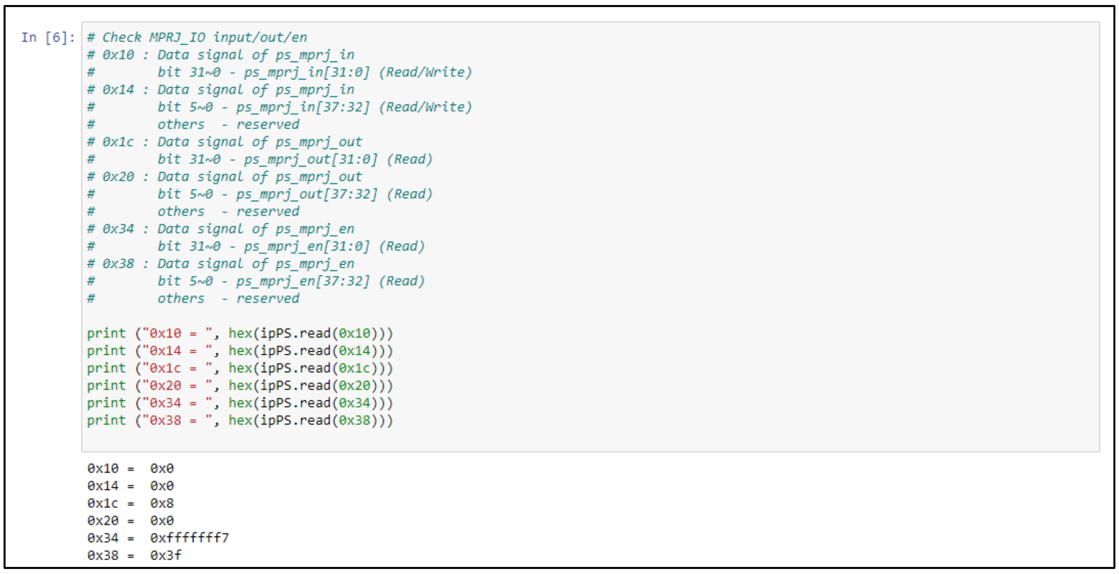
```
0
1
```

In [16]:
```python
# Check MPRJ_IO input/out/en
# 0x10 : Data signal of ps_mprj_in
#         bit 31~0 - ps_mprj_in[31:0] (Read/Write)
# 0x14 : Data signal of ps_mprj_in
#         bit 5~0 - ps_mprj_in[37:32] (Read/Write)
#         others  - reserved
# 0x1c : Data signal of ps_mprj_out
#         bit 31~0 - ps_mprj_out[31:0] (Read)
# 0x20 : Data signal of ps_mprj_out
#         bit 5~0 - ps_mprj_out[37:32] (Read)
#         others  - reserved
# 0x34 : Data signal of ps_mprj_en
#         bit 31~0 - ps_mprj_en[31:0] (Read)
# 0x38 : Data signal of ps_mprj_en
#         bit 5~0 - ps_mprj_en[37:32] (Read)
#         others  - reserved

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)))
```

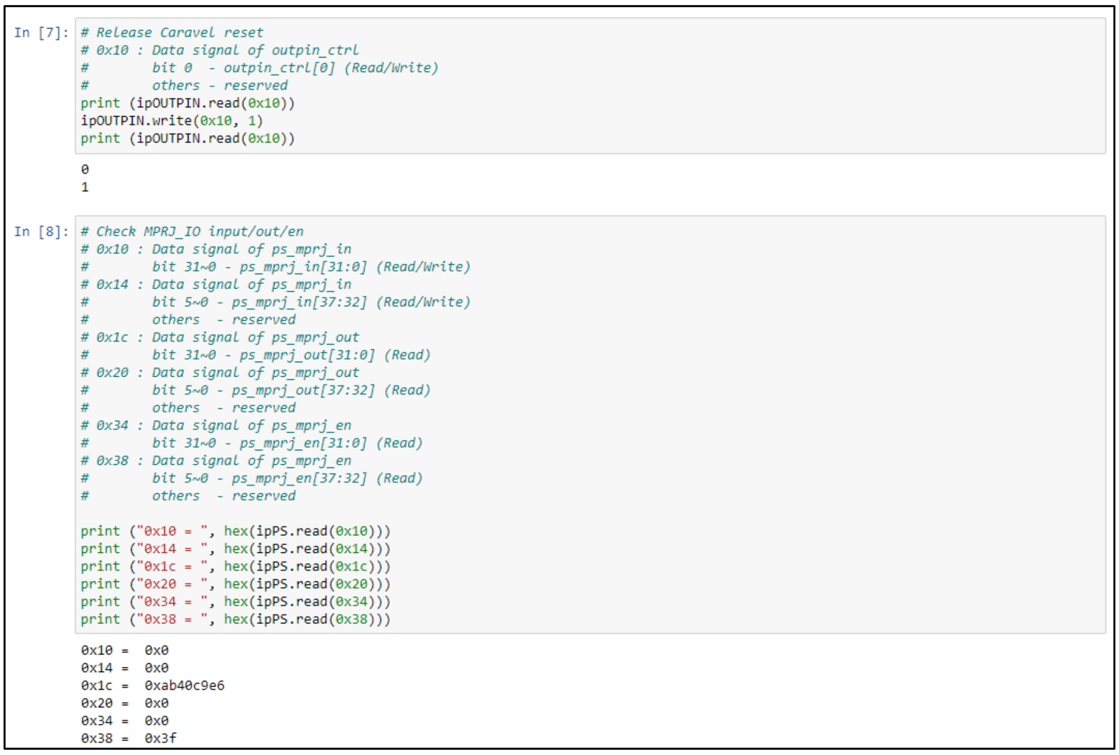
```
0x10 =  0x0
0x14 =  0x0
0x1c =  0xab51a6e0
0x20 =  0x0
0x34 =  0x0
0x38 =  0x3f
```

最終 0x1c 的位置的值最高 bit 為 AB51，符合期待值！

Gcd



```
In [6]: # Check MPRJ_IO input/out/en
        # 0x10 : Data signal of ps_mprj_in
        #        bit 31~0 - ps_mprj_in[31:0] (Read/Write)
        # 0x14 : Data signal of ps_mprj_in
        #        bit 5~0 - ps_mprj_in[37:32] (Read/Write)
        #        others  - reserved
        # 0x1c : Data signal of ps_mprj_out
        #        bit 31~0 - ps_mprj_out[31:0] (Read)
        # 0x20 : Data signal of ps_mprj_out
        #        bit 5~0 - ps_mprj_out[37:32] (Read)
        #        others  - reserved
        # 0x34 : Data signal of ps_mprj_en
        #        bit 31~0 - ps_mprj_en[31:0] (Read)
        # 0x38 : Data signal of ps_mprj_en
        #        bit 5~0 - ps_mprj_en[37:32] (Read)
        #        others  - reserved

        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 =  0x8
        0x20 =  0x0
        0x34 =  0xfffffff7
        0x38 =  0x3f
```

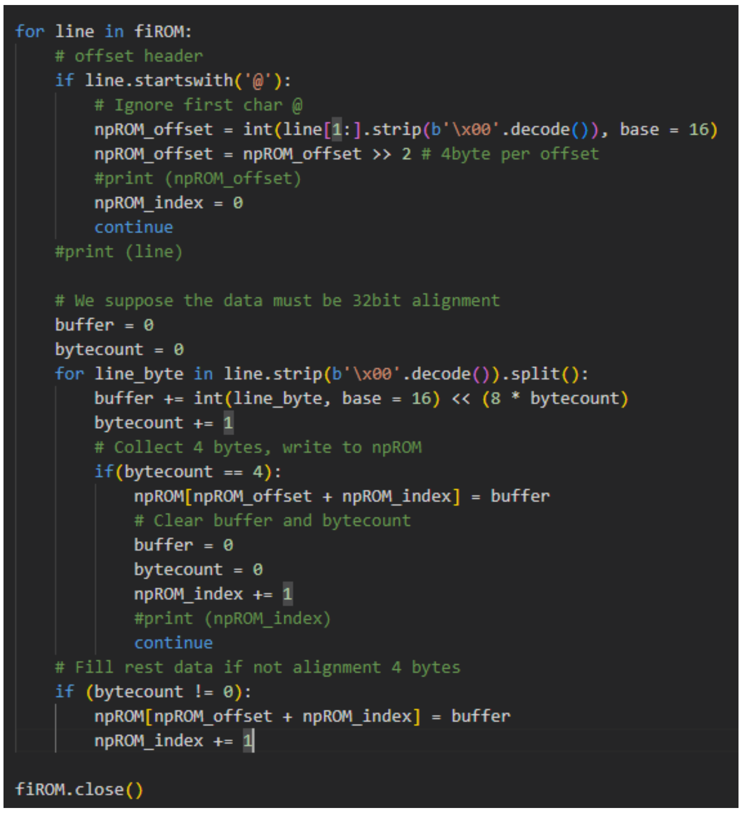


```
In [7]: # Release Caravel reset
        # 0x10 : Data signal of outpin_ctrl
        #        bit 0  - outpin_ctrl[0] (Read/Write)
        #        others - reserved
        print (ipOUTPIN.read(0x10))
        ipOUTPIN.write(0x10, 1)
        print (ipOUTPIN.read(0x10))

        0
        1

In [8]: # Check MPRJ_IO input/out/en
        # 0x10 : Data signal of ps_mprj_in
        #        bit 31~0 - ps_mprj_in[31:0] (Read/Write)
        # 0x14 : Data signal of ps_mprj_in
        #        bit 5~0 - ps_mprj_in[37:32] (Read/Write)
        #        others  - reserved
        # 0x1c : Data signal of ps_mprj_out
        #        bit 31~0 - ps_mprj_out[31:0] (Read)
        # 0x20 : Data signal of ps_mprj_out
        #        bit 5~0 - ps_mprj_out[37:32] (Read)
        #        others  - reserved
        # 0x34 : Data signal of ps_mprj_en
        #        bit 31~0 - ps_mprj_en[31:0] (Read)
        # 0x38 : Data signal of ps_mprj_en
        #        bit 5~0 - ps_mprj_en[37:32] (Read)
        #        others  - reserved

        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 =  0xab40c9e6
        0x20 =  0x0
        0x34 =  0x0
        0x38 =  0x3f
```

最終 0x1c 的位置的值最高 bit 為 AB40，符合期待值！

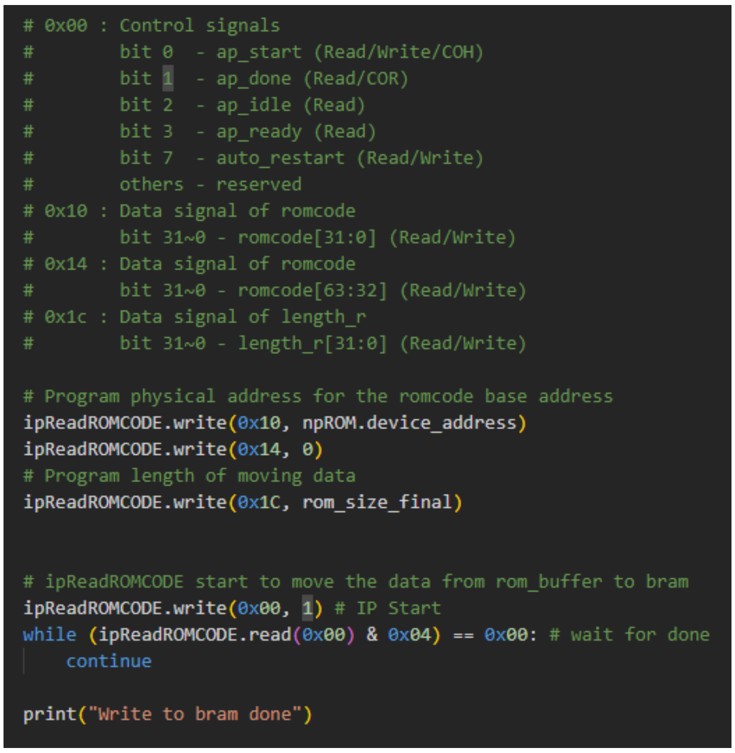
6. Study caravel_fpga.ipynb, and be familiar with caravel SoC control flow

Step 1. 將 hex 檔讀進 ROM 裡



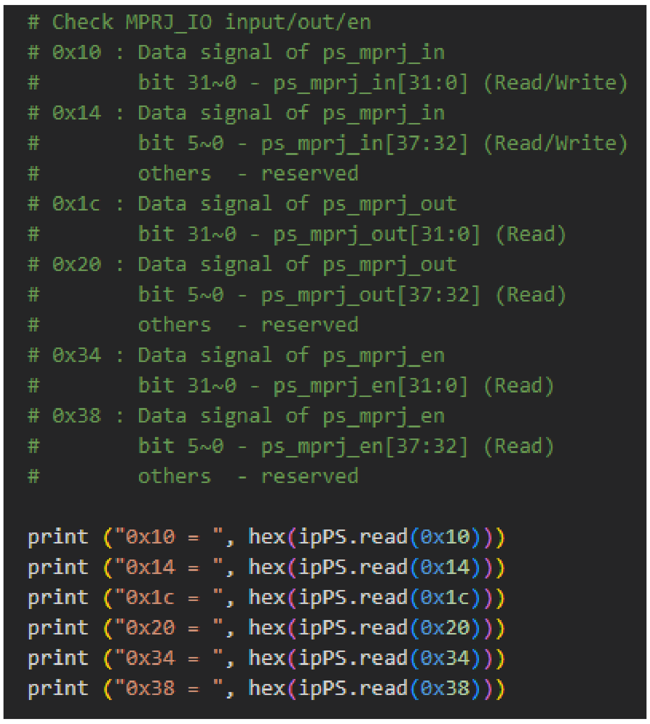
```python
for line in fiROM:
    # offset header
    if line.startswith('@'):
        # Ignore first char @
        npROM_offset = int(line[1:].strip(b'\x00'.decode()), base = 16)
        npROM_offset = npROM_offset >> 2 # 4byte per offset
        #print (npROM_offset)
        npROM_index = 0
        continue
    #print (line)

    # We suppose the data must be 32bit alignment
    buffer = 0
    bytecount = 0
    for line_byte in line.strip(b'\x00'.decode()).split():
        buffer += int(line_byte, base = 16) << (8 * bytecount)
        bytecount += 1
        # Collect 4 bytes, write to npROM
        if(bytecount == 4):
            npROM[npROM_offset + npROM_index] = buffer
            # Clear buffer and bytecount
            buffer = 0
            bytecount = 0
            npROM_index += 1
            #print (npROM_index)
            continue
    # Fill rest data if not alignment 4 bytes
    if (bytecount != 0):
        npROM[npROM_offset + npROM_index] = buffer
        npROM_index += 1

fiROM.close()
```

Step 2. 將 ROM code 放進 BRAM



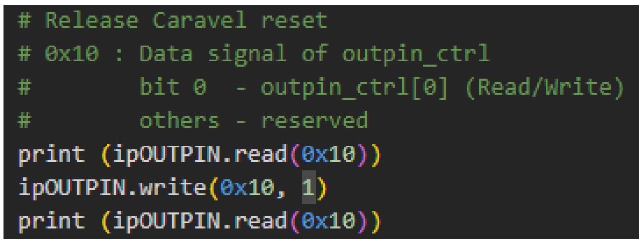
```python
# 0x00 : Control signals
#        bit 0  - ap_start (Read/Write/COH)
#        bit 1  - ap_done (Read/COR)
#        bit 2  - ap_idle (Read)
#        bit 3  - ap_ready (Read)
#        bit 7  - auto_restart (Read/Write)
#        others - reserved
# 0x10 : Data signal of romcode
#        bit 31~0 - romcode[31:0] (Read/Write)
# 0x14 : Data signal of romcode
#        bit 31~0 - romcode[63:32] (Read/Write)
# 0x1c : Data signal of length_r
#        bit 31~0 - length_r[31:0] (Read/Write)

# Program physical address for the romcode base address
ipReadROMCODE.write(0x10, npROM.device_address)
ipReadROMCODE.write(0x14, 0)
# Program length of moving data
ipReadROMCODE.write(0x1C, rom_size_final)


# ipReadROMCODE start to move the data from rom_buffer to bram
ipReadROMCODE.write(0x00, 1) # IP Start
while (ipReadROMCODE.read(0x00) & 0x04) == 0x00: # wait for done
    continue

print("Write to bram done")
```

Step 3. 讀取 mprj_in 的值



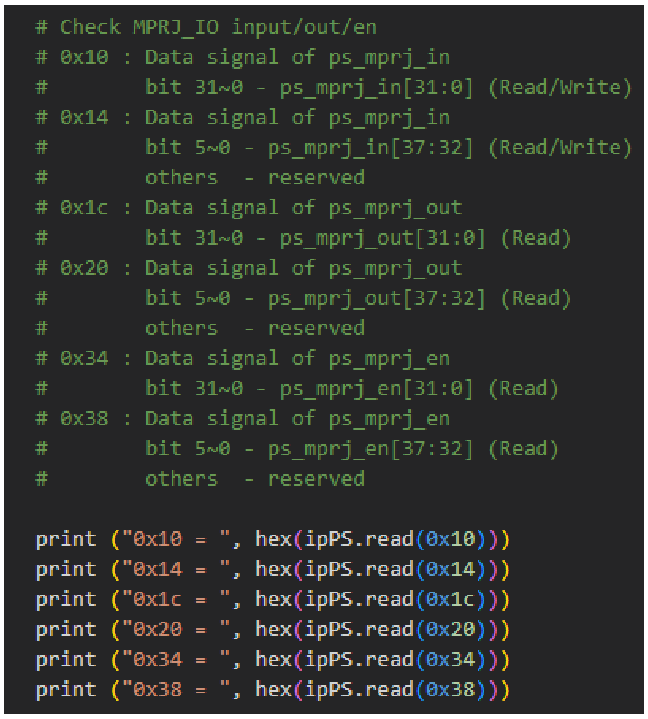
```
# Check MPRJ_IO input/out/en
# 0x10 : Data signal of ps_mprj_in
#        bit 31~0 - ps_mprj_in[31:0] (Read/Write)
# 0x14 : Data signal of ps_mprj_in
#        bit 5~0 - ps_mprj_in[37:32] (Read/Write)
#        others  - reserved
# 0x1c : Data signal of ps_mprj_out
#        bit 31~0 - ps_mprj_out[31:0] (Read)
# 0x20 : Data signal of ps_mprj_out
#        bit 5~0 - ps_mprj_out[37:32] (Read)
#        others  - reserved
# 0x34 : Data signal of ps_mprj_en
#        bit 31~0 - ps_mprj_en[31:0] (Read)
# 0x38 : Data signal of ps_mprj_en
#        bit 5~0 - ps_mprj_en[37:32] (Read)
#        others  - reserved

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)))
```

Step 4.   RESET



```
# Release Caravel reset
# 0x10 : Data signal of outpin_ctrl
#        bit 0  - outpin_ctrl[0] (Read/Write)
#        others - reserved
print (ipOUTPIN.read(0x10))
ipOUTPIN.write(0x10, 1)
print (ipOUTPIN.read(0x10))
```

Step 5. 讀取 mprj_out 的值



```
# Check MPRJ_IO input/out/en
# 0x10 : Data signal of ps_mprj_in
#        bit 31~0 - ps_mprj_in[31:0] (Read/Write)
# 0x14 : Data signal of ps_mprj_in
#        bit 5~0 - ps_mprj_in[37:32] (Read/Write)
#        others  - reserved
# 0x1c : Data signal of ps_mprj_out
#        bit 31~0 - ps_mprj_out[31:0] (Read)
# 0x20 : Data signal of ps_mprj_out
#        bit 5~0 - ps_mprj_out[37:32] (Read)
#        others  - reserved
# 0x34 : Data signal of ps_mprj_en
#        bit 31~0 - ps_mprj_en[31:0] (Read)
# 0x38 : Data signal of ps_mprj_en
#        bit 5~0 - ps_mprj_en[37:32] (Read)
#        others  - reserved

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)))
```