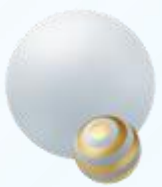




Computer Organization

Lab13 Cache(1)

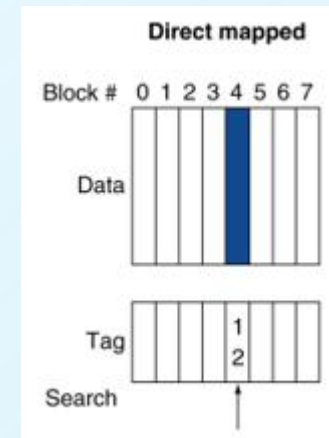
Load program on
CPU (optional)



Topic

➤ Cache (1)

- Direct Mapped Cache(implementation)
 - Components in Cache
 - Read/Write



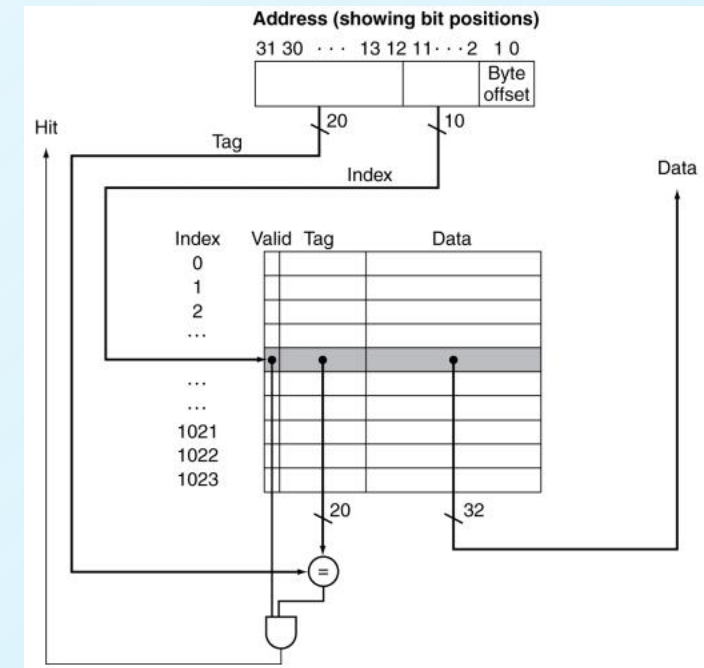
➤ Load program on CPU (optional)

- Re-program the FPGA chip with updated bitstream file
- (*) No need to re-program the FPGA chip by getting the updated coe files through Uart port



Direct Mapped Cache

- Direct mapped cache
 - **ONE** data in **MEMORY** is mapped to **ONLY ONE** location in **CACHE**.
 - **Location determined by the ADDRESS:**
 - The **lower bits of address** define the place of the unit(aka **block**) in the cache.
 - The **higher bits of address** are called **tag** which would be **stored in the cache unit**(aka cache block).





Direct Mapped Cache continued

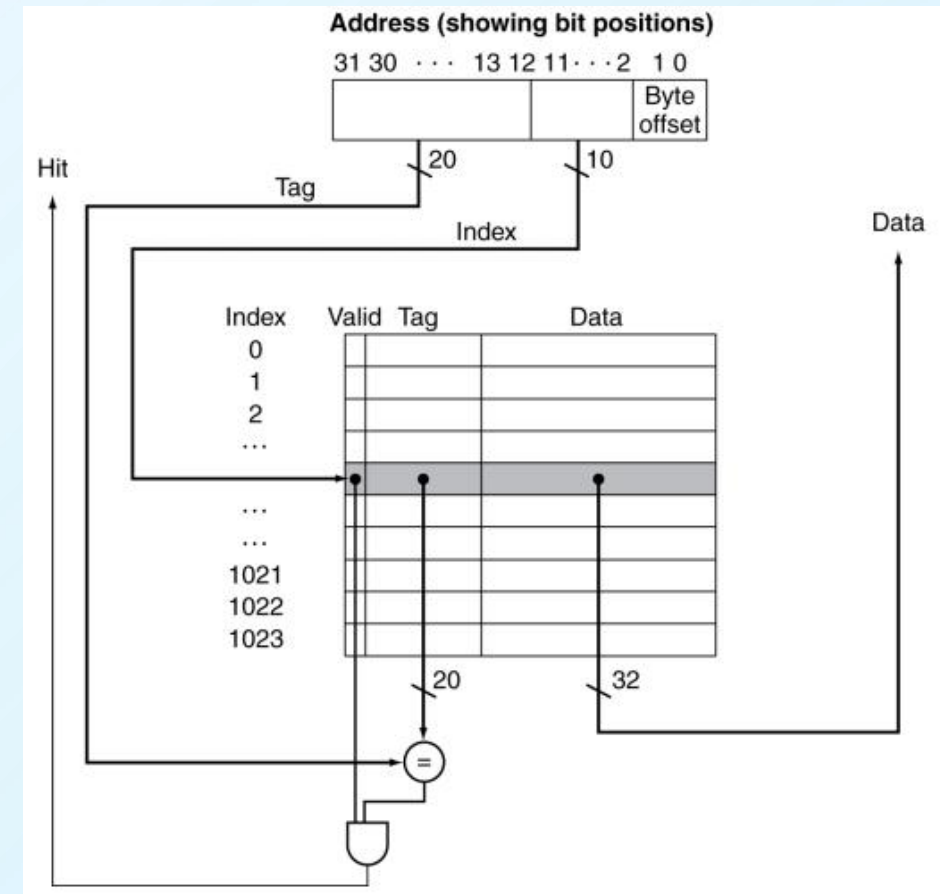
➤ Componets in Cache

➤ Valid(1bit)

- Initial value is 1' b0
- Turns to 1' b1 while the cache unit has been written which means there has been data in the cache unit

➤ Tag: Higer bits of the address

➤ Data: Store the data which has been cached





Direct Mapped Cache continued

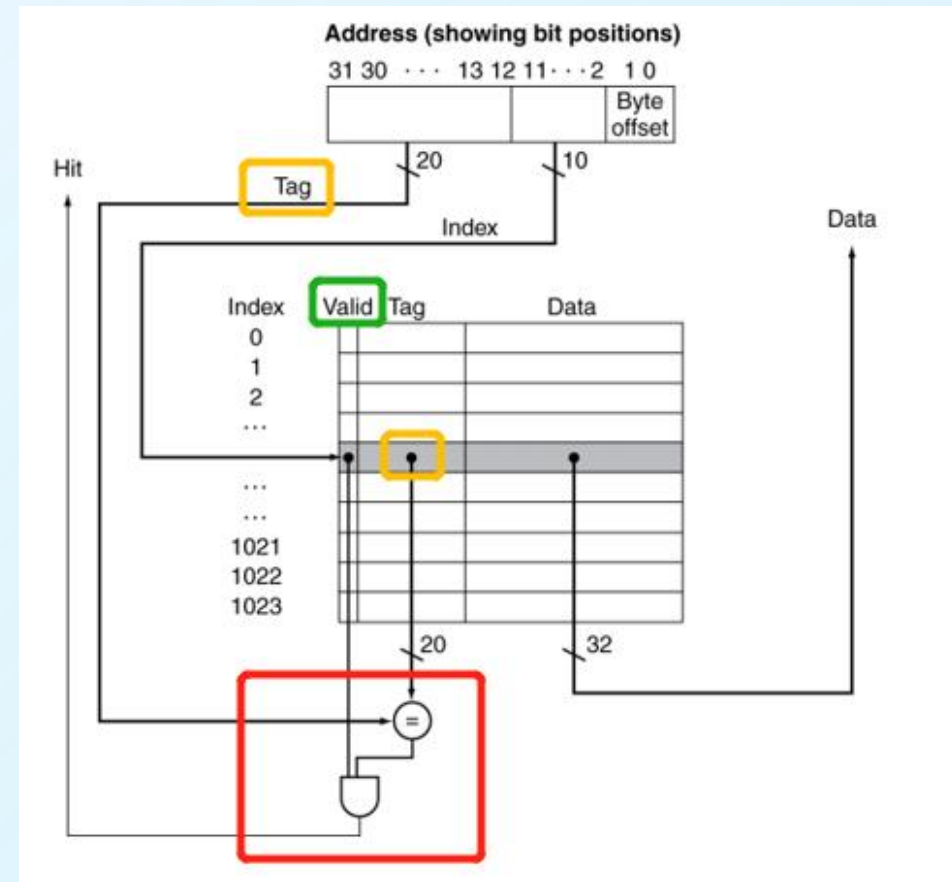
➤ Cache Hit vs Cache Miss

➤ Hit

- Find the cache unit based on the 'Index' of 'Address'
- Find the 'valid' and 'Tag' parts from the cache unit
- If 'valid' is 1 and the 'Tag' is the same as the tag part(higer bits) of 'Address', then it is cache hit.

➤ Miss: If not hit, then miss

➤ hit ratio = hits / accesses





Implement Direct Map Cache(ports)

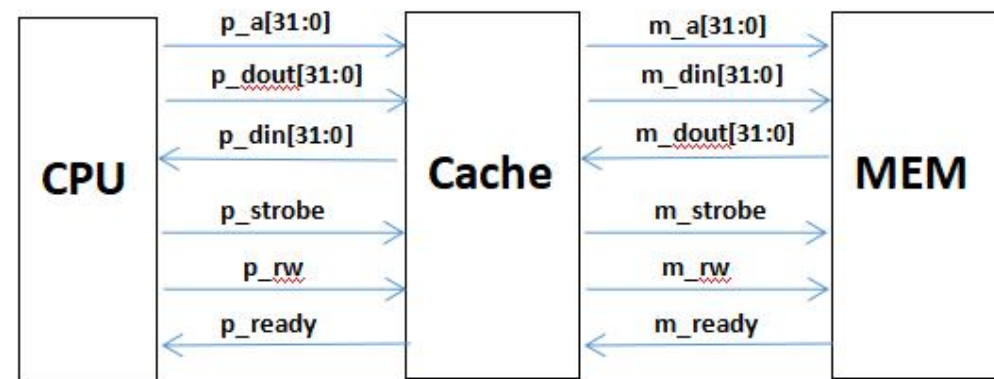
```
module cache
#(parameter A_WIDTH=32, parameter C_INDEX=10, parameter D_WIDTH=32)
(clk, resetn,
p_a, p_dout, p_din, p_strobe, p_rw, p_ready,
m_a, m_dout, m_din, m_strobe, m_rw, m_ready);
```

```
input clk, resetn;
```

```
input [A_WIDTH-1:0] p_a; //address of memory to be accessed
input [D_WIDTH-1:0] p_dout; //the data from cpu
output [D_WIDTH-1:0] p_din; //the data to cpu
input p_strobe; // 1 means to do the reading or writing
input p_rw; // 0:read, 1:write
output p_ready; // tell cpu, outside of cpu is ready
```

```
output [A_WIDTH-1:0] m_a; //address of memory to be accessed
input [D_WIDTH-1:0] m_dout; //the data from memory
output [D_WIDTH-1:0] m_din; //the data to memory
output m_strobe; //1 means to do the reading or writing
output m_rw; //0:read, 1:write
input m_ready; //memory is ready
```

Q. What's the relationship between **A_WIDTH**, **C_INDEX** and **D_WIDTH**?



TIPS:
parameter in verilog makes the design more **flexiable**, which is highly recommended.



Implement Direct Map Cache(components)

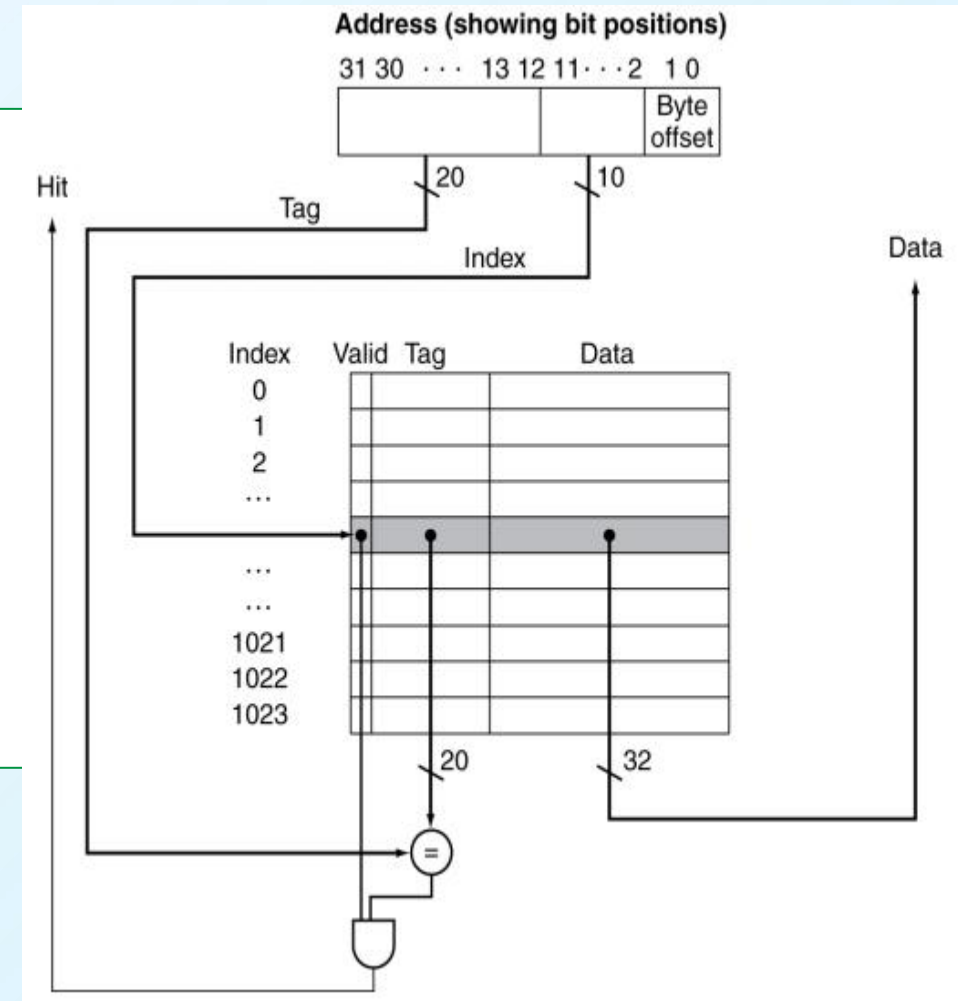
Build the cache, complete the following statement

```
// d_valie is a piece of memory stored the valid info for every block  
reg d_valid [ 0 : complete code here p7-1];
```

```
// T_WIDTH is the width of 'Tag'  
localparam T_WIDTH = complete code here p7-2 ;  
//d_tags is a piece of memory stored the tag info for every block  
reg [T_WIDTH-1: 0] d_tags [0 : complete code here p7-1];
```

```
//d_data is a piece of memory stored the data for every block  
reg [D_WIDTH-1:0] d_data [0 : complete code here p7-1];
```

A_WIDTH : the width of 'Address', here its value is 32.
C_INDEX : the width of 'Index', here its value is 10.
T_WIDTH : used as the width of 'Tag'. 'Byte offset' : 2 bit-width





Implement Direct Map Cache (cache hit vs cache miss)

Complete the following code to implement **cache hit**

```
// d_valie is a piece of memory stored the valid info for every block in cache  
// d_tags is a piece of memory stored the tag info for every block in cache  
// d_data is a piece of memory stored the data for every block in cache
```

```
wire [C_INDEX-1:0] index = p_a[ C_INDEX+1 : 2 ] ;  
wire [T_WIDTH-1:0] tag   = p_a[ A_WIDTH-1 : C_INDEX+2 ] ;
```

```
wire valid = d_valid[index];  
wire [T_WIDTH-1:0] tagout = d_tags[index];  
wire [D_WIDTH-1:0] c_dout = d_data[index];
```

```
//cache control
```

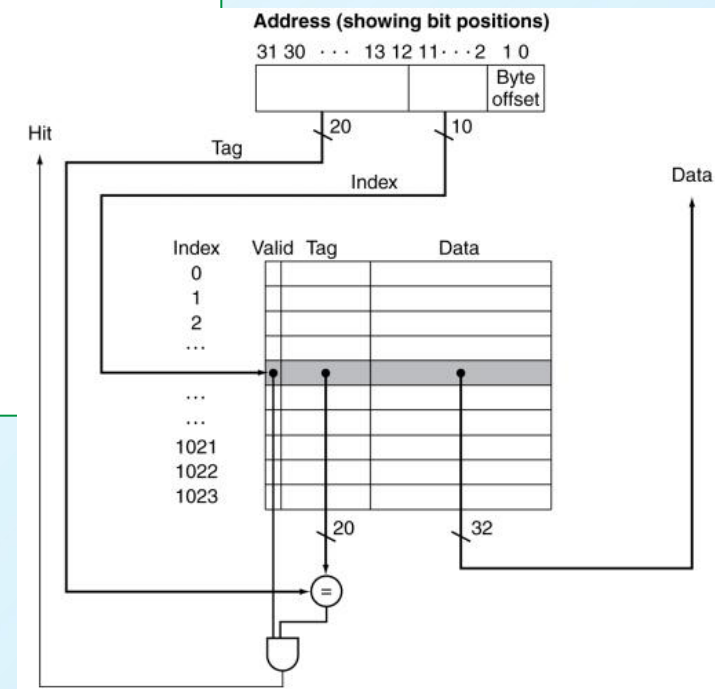
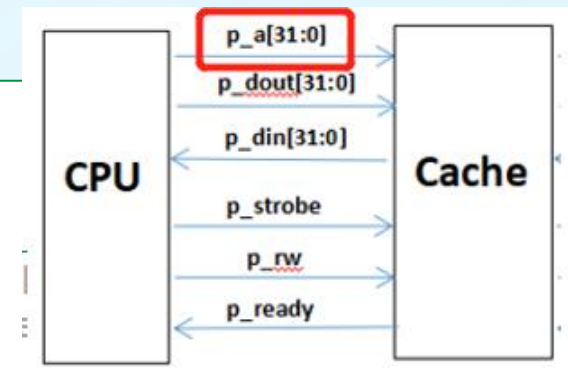
```
wire cache_hit = complete the code here p8
```

```
wire cache_miss = ~cache_hit;
```

A_WIDTH : the width of 'Address', here its value is 32.

C_INDEX : the width of 'Index', here its value is 10.

T_WIDTH : used as the width of 'Tag'. 'Byte offset' : 2 bit-width





Implement Direct Map Cache(cache write)

Complete the following code to implement **cache write**

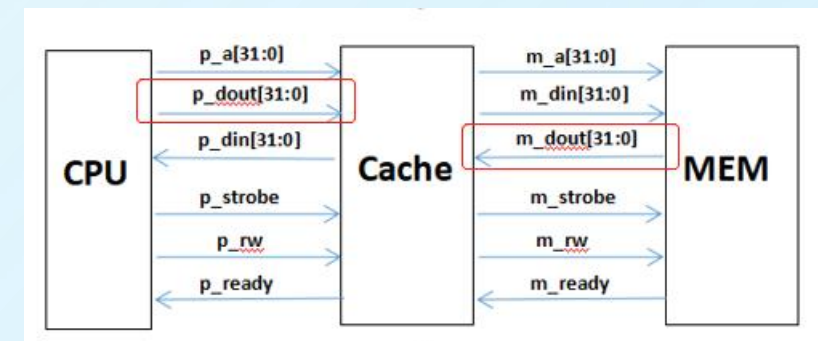
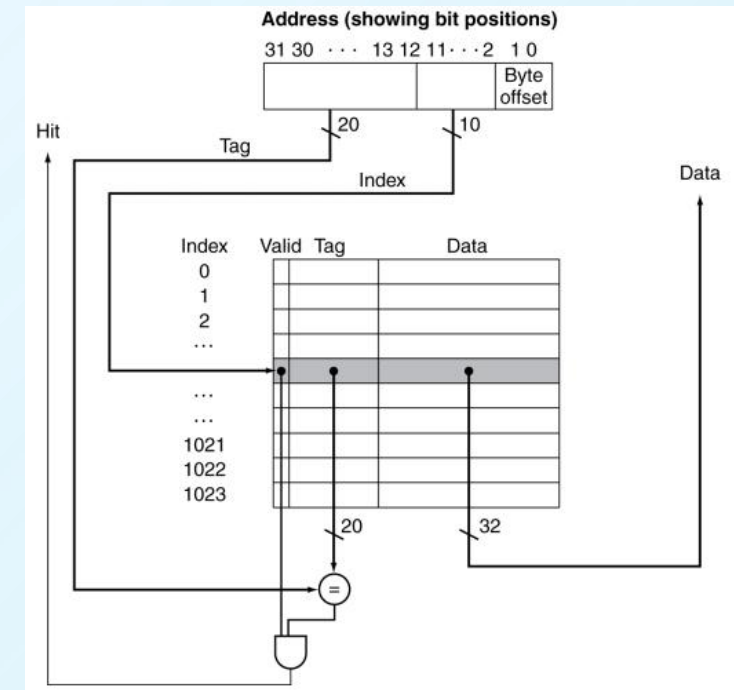
```
wire c_write = p_rw | cache_miss & m_ready ;

integer i;
always @ (posedge clk, negedge resetn)
  if( resetn == complete code here p9-1 ) begin
    for( i=0; i<( complete code here p9-2 ); i=i+1 )
      d_valid[i] <= 1'b0;
  end
  else if(c_write==1'b1)
    d_valid[index] <= 1'b1;
```

```
////////////////////////////////////
always @ (posedge clk)
  if(c_write==1'b1) d_tags[index] <= tag;
```

```
////////////////////////////////////
wire sel_in = p_rw ;
wire [D_WIDTH-1:0] c_din = complete code here p9-3 ;
```

```
always @ (posedge clk)
  if(c_write==1'b1) d_data[index] <= c_din;
```





Implement Direct Map Cache(memory write)

Complete the following code to implement memory write (write_through)

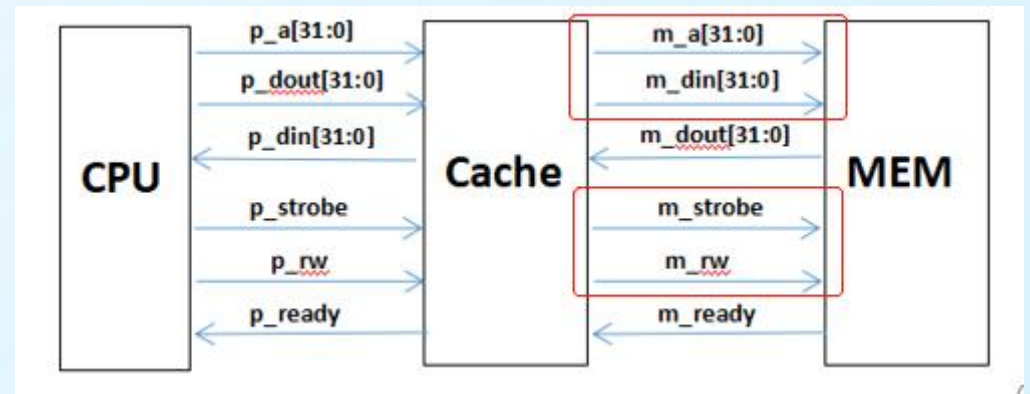
```
// write data (m_din) to the memory unit specified by m_a
  assign m_a = p_a;

  assign m_din = p_dout;

// p_strobe (1 means to do the reading or writing, 0 means else)
// m_strobe (1 means to do the reading or writing, 0 mean else)
// p_rw, m_rw ( 0:read, 1:write)

  assign m_strobe = p_strobe & (p_rw | cache_miss);

  assign m_rw = complete code here p10;
```



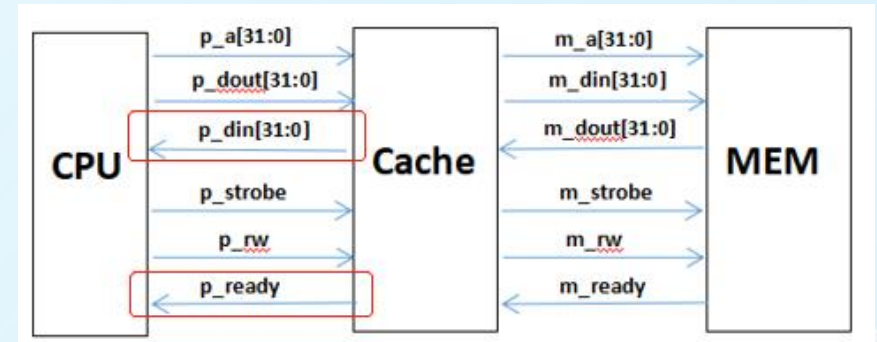
Implement Direct Map Cache(CPU read)

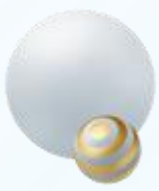
Complete the following code to implement CPU read

```
//read data to CPU
// if cache hit, read c_dout from cache to p_din, else read m_dout to p_in
// wire [D_WIDTH-1:0] c_dout = d_data[index];
wire sel_out = cache_hit;

assign p_din = complete code here p11 ? c_dout : m_dout;

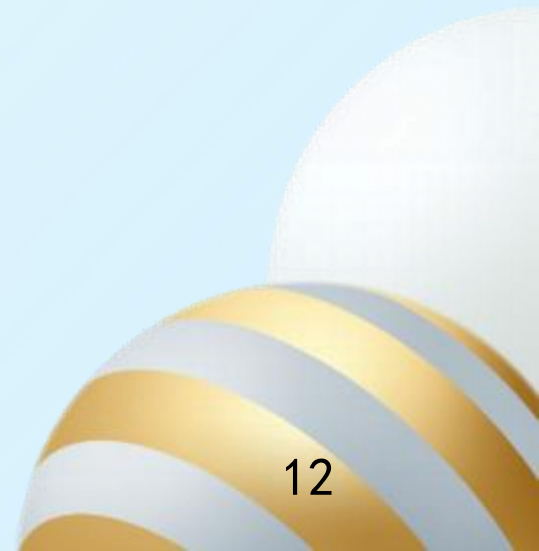
assign p_ready = ~p_rw & cache_hit | (cache_miss | p_rw) & m_ready;
```

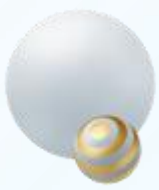




Practice

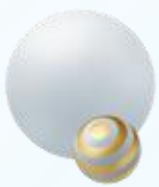
- 1. Complete the code of the Directly Mapped Cache(page 7-11)
- 2. Build a testbench to verify the function of the Directly Mapped Cache.



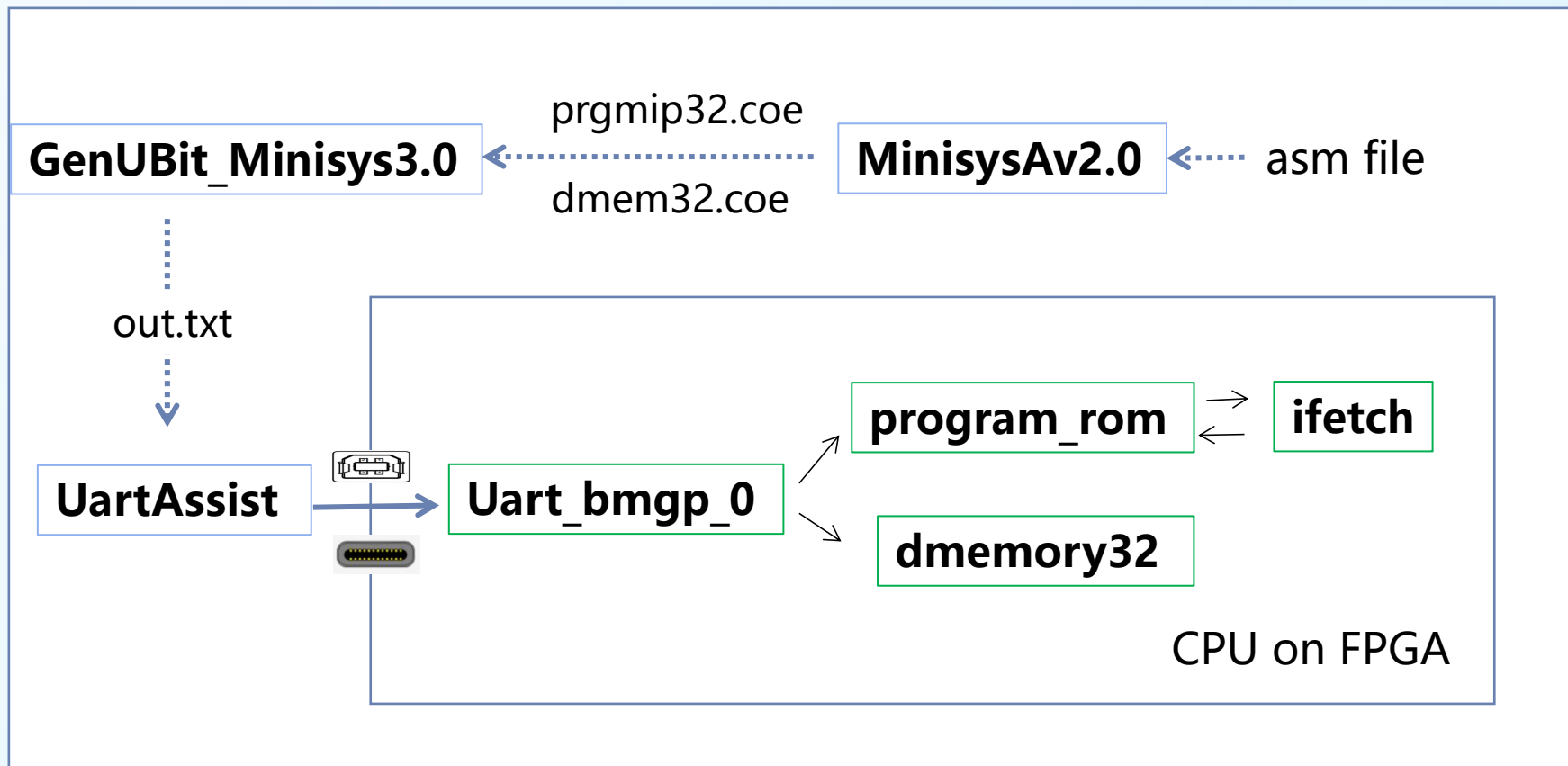


How to make CPU work on a new program?

- How to make CPU work on a new program?
 - new program -> new machine code and initial data (new coe(s))
 - Solution1:
 - update the **ProgramRom** and **DataRAM** of CPU with new coe(s) -> re-generate bitstream of updated CPU -> re-program the FPGA chip by updated bitstream file
 - **Solution2:**
 - While CPU work on **Communicate mode**: CPU get the new coe(s) by uart port, then rewrite its '**ProgramROM**' and **DataRAM**
 - While CPU work on **Normal mode**: work on the updated program
 - 2 tools(GenUBit_Minisys3.0, UartAssist) and some modifications on the CPU are needed for solution2.

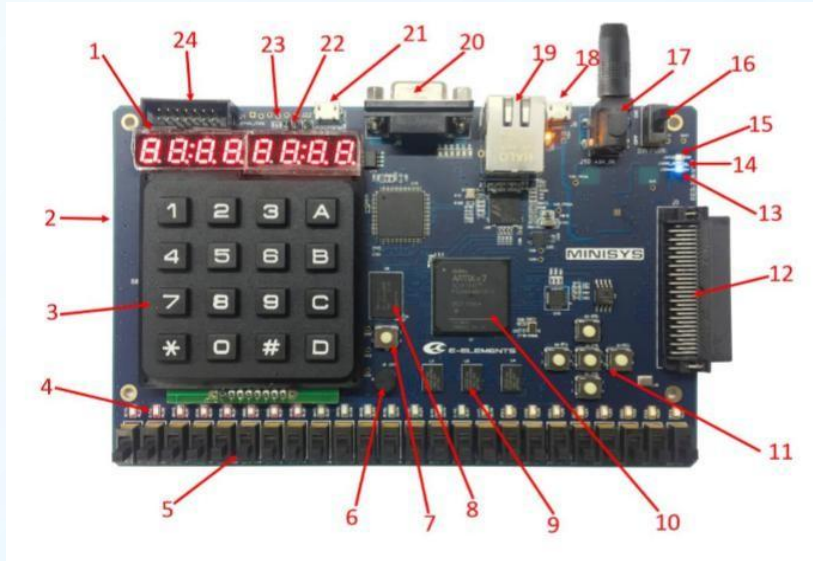


Solution2 on load program on CPU



"GenUBit_Minisys3.0" , "UartAssist" and "UARTCoe_v3.0" could be found in the "uart_tools.rar" of "labs/lab13_uart" on BlackBoard site

Uart Interface on Minisys Board and EGO1 Board



For **Minisys board**(old version, the type of USB_Jtag interface is **typeB**), **port 18**(as shown on the left hand) is the USB to UART interface.

For **Minisys board**(new version, the type of USB_Jtag interface is **typeC**), USB_Jtag and USB to UART interface share the same port.



For **EGO1 board**, USB_Jtag and USB(**typeC**) to UART interface share the same port.

The handbook of Minisys board and EGO1 board could be found in the "Handbook_of_Minisys_EGO" of "labs" on BlackBoard site

Changes on Single Cycle CPU

1. Two working modes on the CPU

- Normal mode vs Uart Communication mode

2. A new module(Uart_bmgp_0) which works as Uart interface

3. A new clock for uart communication

4. Changes

- 3-1) CPU top:

New module, new ports, new internal connection and new logic

- 3-2) Changes on Data-memory

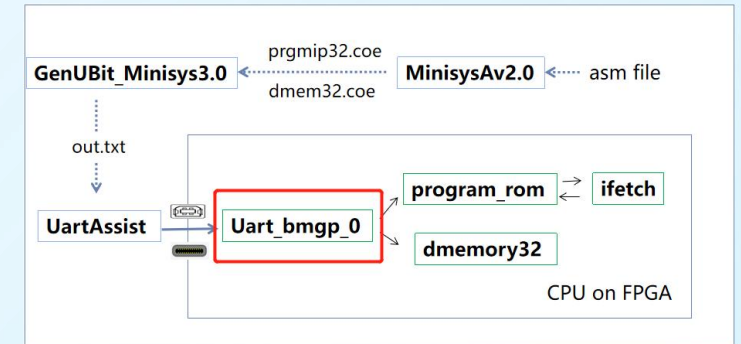
Working mode: Normal mode vs Uart Communication mode

- 3-3) Changes on IFetch

- Change IP core "prgrom" from ROM to RAM

- Working mode: Normal mode vs Uart Communication mode

- Separate "prgrom" from IFetch (optional)

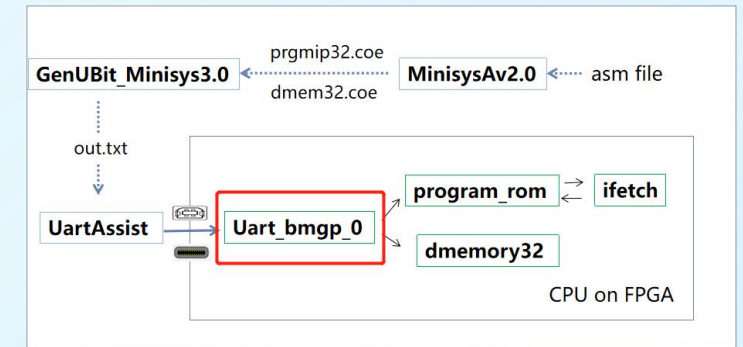




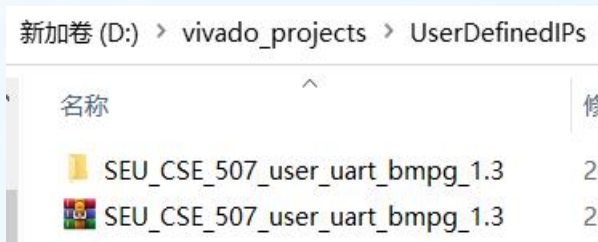
Add an IP core Which Processes Uart Data

Step1: Add the **IP core** to IP catalog(User Repository) of vivado.

- The Communication between this IP core and Uart port:
 - **Receive** data from Uart port and forward to data-memory and instruction-memory
 - **Send** data back to uart port to info that all the data has been received.

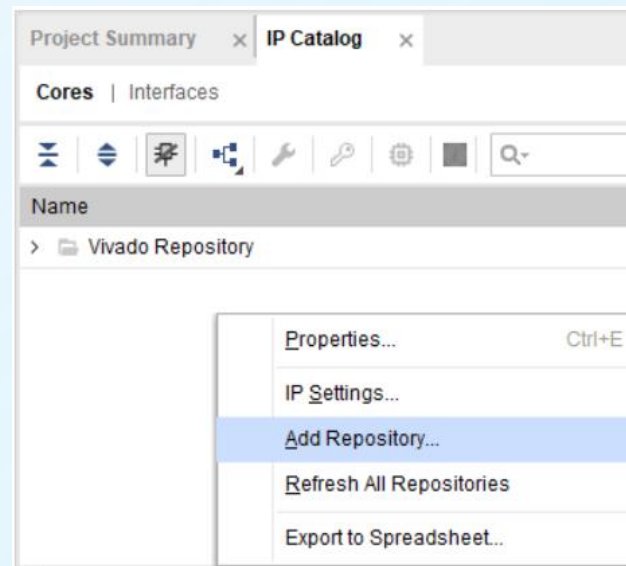


Step1-1: download the ziped IPcore file, then unzip it

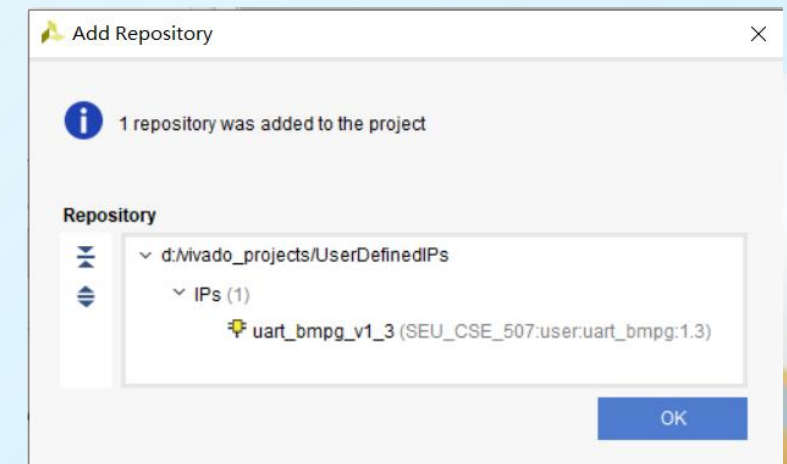


The IP core could be found in the "labs/lab13_uart" on BlackBoard site

Step1-2: open "IP Catalog" pane, right click on the blank space and select "Add Repository"



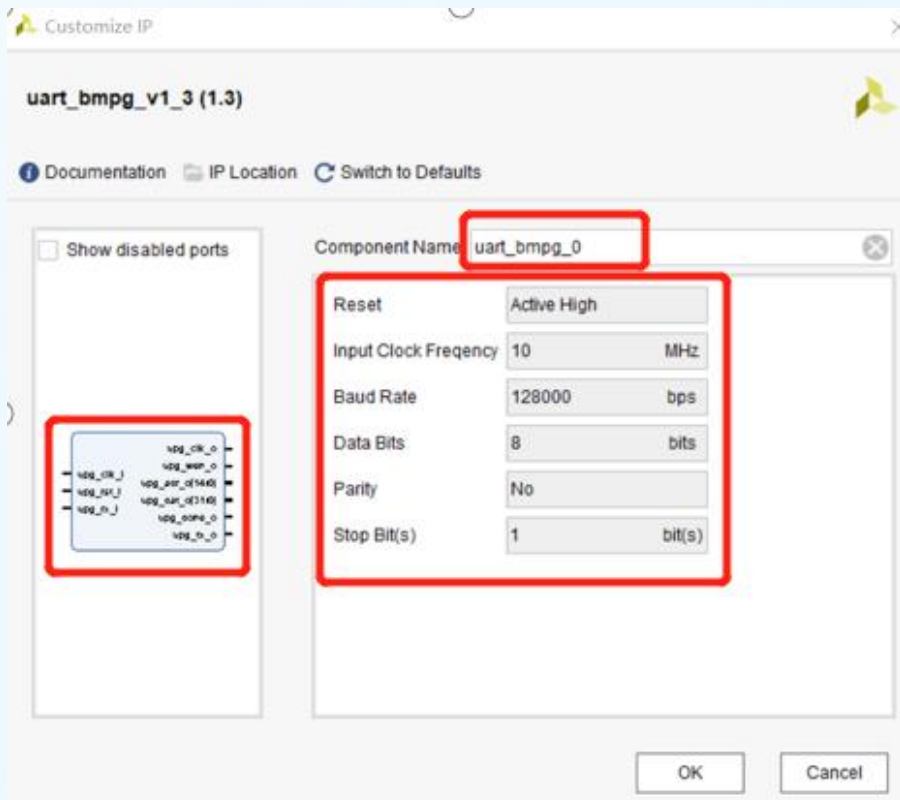
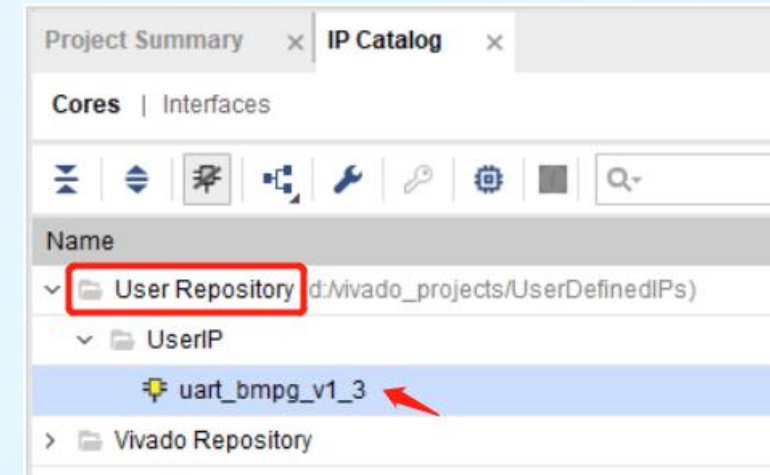
Step1-3: in "Add Repository" pane select the directory where the unzipped IPcore is placed. vivado would detect the IPcore and pop up the following prompt window, click "OK".





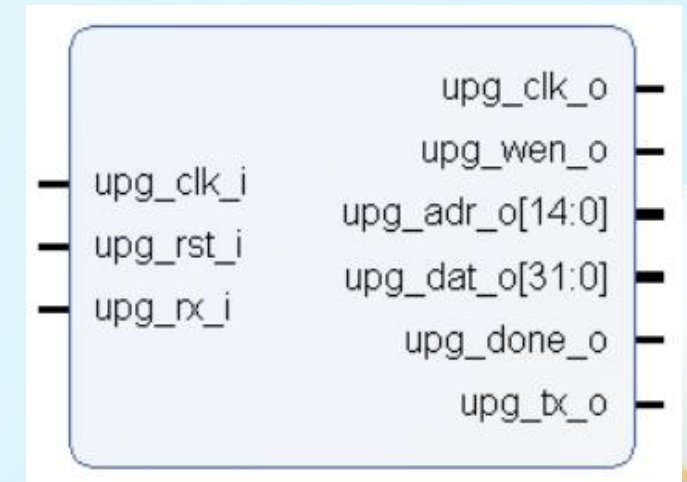
Add an IP core Which Processes Uart Data continued

Step2: Add the IP core(**uart_bmpg_0**) from IP catalog into vivado project: in “**IP catalog**” pane, the IPcore(**uart_bmpg_v1_3**) could be found in the “**User Repository**” , click it to add it to your vivado project.



NOTE:
Don't change the settings of this IP core.


While using the IPcore, **its name, features on uart communication and ports are important !**

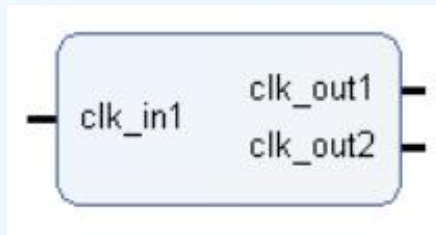




Add a New Clock For The New IP core

- Reset the “cpuck” IP core to make a new clock
 - Add a new clk_out (**clk_out2**) whose frequency is **10 Mhz** for the **IP core(uart_bmpg_0)** which is used for Uart communication(on last page)

>  cpuck : cpuck (cpuck.xci)



Component Name: cpuck

Clocking Options | **Output Clocks** | Port Renaming | PLLE2 Settings | Summary

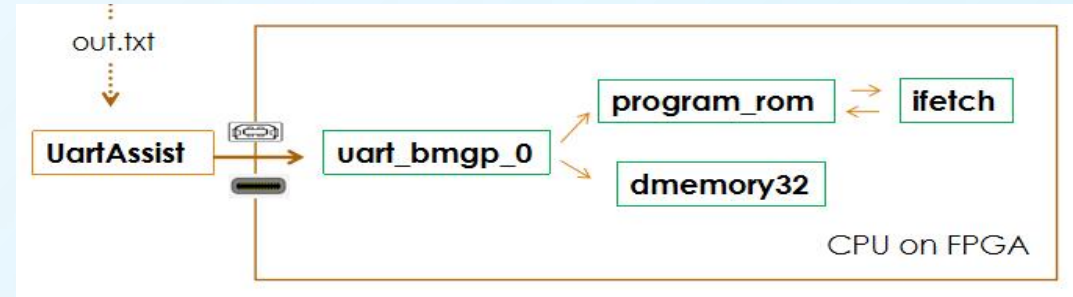
The phase is calculated relative to the active input clock.

| Output Clock | Port Name | Output Freq (MHz) | | Phase (deg) |
|--|-----------|-------------------|--------|-------------|
| | | Requested | Actual | |
| <input checked="" type="checkbox"/> clk_out1 | clk_out1 | 23.0 | 23.000 | 0.000 |
| <input checked="" type="checkbox"/> clk_out2 | clk_out2 | 10.0 | 10.000 | 0.000 |

Handwritten red annotations: "single_cycle_cpu_clk" with an arrow pointing to the 23.0 MHz value, and "uart clk" with an arrow pointing to the 10.0 MHz value.

Changes on CPU Top Module

```
module CPU_TOP(  
    input  fpga_rst,    //Active High  
    input  fpga_clk,  
    input[23:0]  switch2N4,  
    output[23:0] led2N4,  
  
    // UART Programmer Pinouts  
    // start Uart communicate at high level  
    input  start_pg,    // Active High  
    input  rx,          // receive data by UART  
    output tx           // send data by UART  
);
```



```
// For Minisys, the package_pin relationship in the constraints file  
set_property -dict {IOSTANDARD LVCMOS33 PACKAGE_PIN Y19} [get_ports rx]  
set_property -dict {IOSTANDARD LVCMOS33 PACKAGE_PIN V18} [get_ports tx]  
  
// For EGO1, the package_pin relationship in the constraints file  
set_property -dict {IOSTANDARD LVCMOS33 PACKAGE_PIN T4} [get_ports rx]  
set_property -dict {IOSTANDARD LVCMOS33 PACKAGE_PIN N5} [get_ports tx]
```

Here the usage of “fpga_rst” and “start_pg” are only one type of implements, not the request.

The **Y19**(UART_RX) and **V18**(UART_TX) are the USB-UART pins of the FPGA chip(**Artix7 fgg484**) on **Minisys** Board.

The **T4**(UART_RX) and **N5**(UART_TX) are the USB-UART pins of the FPGA chip(**Artix7 csg324**) on **EGO1** Board



Changes on CPU Top Module continued

```
module CPU_TOP(  
    input  fpga_rst,    //Active High  
    input  fpga_clk,  
    input[23:0] switch2N4,  
    output[23:0] led2N4,  
  
    // UART Programmer Pinouts  
    // start Uart communicate at high level  
    input  start_pg,    // Active High  
    input  rx,          // receive data by UART  
    output tx           // send data by UART  
);
```

```
// UART Programmer Pinouts  
wire upg_clk, upg_clk_o;  
wire upg_wen_o;    //Uart write out enable  
wire upg_done_o;   //Uart rx data have done  
  
//data to which memory unit of program_rom/dmemory32  
wire [14:0] upg_adr_o;  
  
//data to program_rom or dmemory32  
wire [31:0] upg_dat_o;
```

```
wire spg_bufg;  
BUFG U1(.I(start_pg), .O(spg_bufg)); // de-twitter  
// Generate UART Programmer reset signal  
reg upg_rst;  
always @ (posedge fpga_clk) begin  
    if (spg_bufg)    upg_rst = 0;  
    if (fpga_rst)    upg_rst = 1;  
end  
//used for other modules which don't relate to UART  
wire rst;  
assign rst = fpga_rst | !upg_rst;
```

Q1. How many types of working mode on the CPU which support uart communication to download the program and the data?

How to identify different types of working mode?

Q2. What's the relationship between the working mode and the "fpga_rst" and "start_pg"?

TIPS: Here the usage of "fpga_rst" and "start_pg" are only one type of implements, not the request.

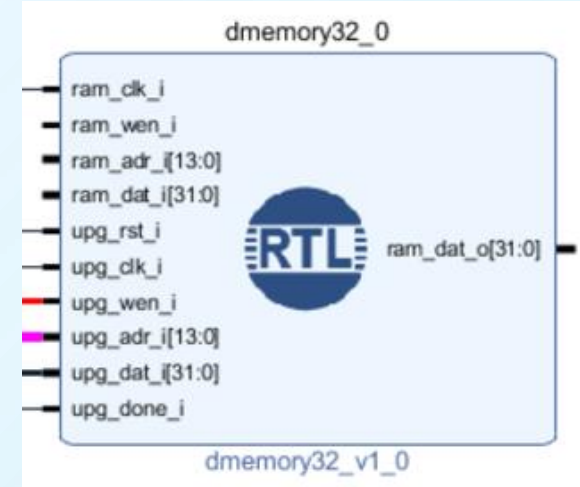
Changes on Demeory32

```

module dmemory32 (
    input      ram_clk_i,           // from CPU top
    input      ram_wen_i,           // from Controller
    input [13:0] ram_adr_i,          // from alu_result of ALU
    input [31:0] ram_dat_i,         // from read_data_2 of Decoder
    output [31:0] ram_dat_o,        // the data read from data-ram

    // UART Programmer Pinouts
    input      upg_rst_i,           // UPG reset (Active High)
    input      upg_clk_i,           // UPG ram_clk_i (10MHz)
    input      upg_wen_i,           // UPG write enable
    input [13:0] upg_adr_i,          // UPG write address
    input [31:0] upg_dat_i,         // UPG write data
    input      upg_done_i           // 1 if programming is finished
);

```



```

wire ram_clk = !ram_clk_i;

```

```

/* CPU work on normal mode when kickOff is 1.
CPU work on Uart communicate mode when kickOff is 0.*/
wire kickOff = upg_rst_i | (~upg_rst_i & upg_done_i);

```

```

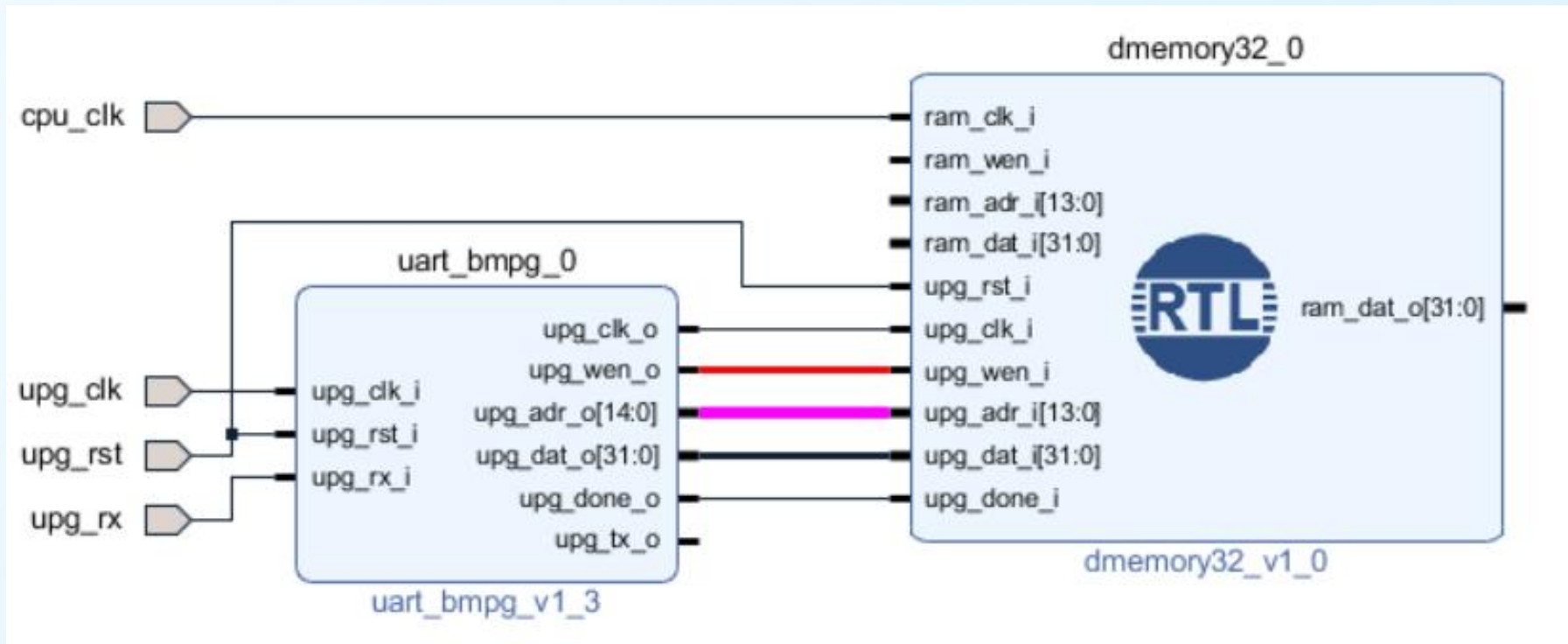
ram ram (
    .clka (kickOff ? ram_clk : upg_clk_i),
    .wea (kickOff ? ram_wen_i : upg_wen_i),
    .addra (kickOff ? ram_adr_i : upg_adr_i),
    .dina (kickOff ? ram_dat_i : upg_dat_i),
    .douta (ram_dat_o)
);

```

Q. While “**kickOff**” is 1'b1, what's the working mode of the CPU ?
How about while “**kickOff**” 1'b0?

Changes on Demeory32 continued

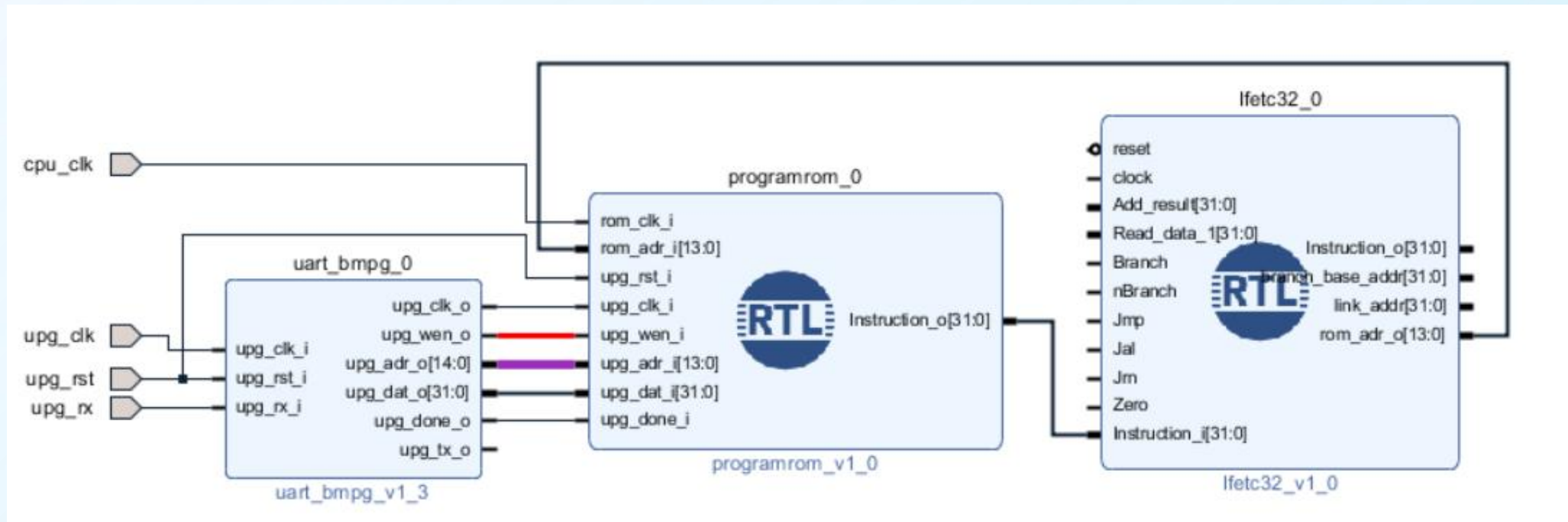
- **upg_wen_i** (uart write enable on **Dmemory32**) :
 - determined by: **upg_wen_o**(from uart_bmpg_0) & **upg_adr_o[14]** (from uart_bmpg_0)
- **upg_adr_i[13:0]** (uart write address on **Dmemory32**):
 - connect with: **upg_adr_o[13:0]** (from uart_bmpg_0)



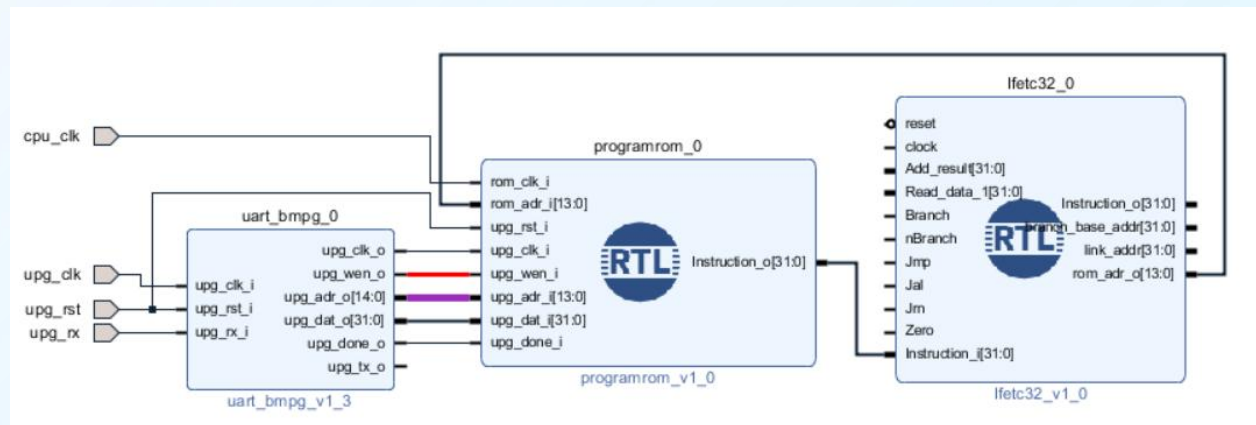


Changes on IFetch

- **Separate IP core("programrom_0")** which stores the Instruction from IFetch(optional)
- **Change program from ROM to RAM**(writeable while work on Uart communication mode, read only while work on normal mode)
- **upg_wen_i** (uart write enable on "programrom"), determined by: **upg_wen_o**(from **uart_bmpg_0**) & **(!upg_adr_o[14])** (from **uart_bmpg_0**)
- **upg_adr_i[13:0]** (uart write address on "programrom"), connect with **upg_adr_o[13:0]** (from **uart_bmpg_0**)



Changes on IFetch continued



```

module programrom (
    // Program ROM Pinouts
    input          rom_clk_i,          // ROM clock
    input[13:0]    rom_adr_i,          // From IFetch
    output [31:0]  Instruction_o,      // To IFetch
    // UART Programmer Pinouts
    input          upg_rst_i,          // UPG reset (Active High)
    input          upg_clk_i,          // UPG clock (10MHz)
    input          upg_wen_i,          // UPG write enable
    input[13:0]    upg_adr_i,          // UPG write address
    input[31:0]    upg_dat_i,          // UPG write data
    input          upg_done_i         // 1 if program finished
);
    
```

/ if kickOff is 1 means CPU work on normal mode,
otherwise CPU work on Uart communication mode */*

wire kickOff = upg_rst_i | (~upg_rst_i & upg_done_i);

```

prgrom instmem (
    .clka (kickOff ? rom_clk_i : upg_clk_i ),
    .wea (kickOff ? 1'b0 : upg_wen_i ),
    .addra (kickOff ? rom_adr_i : upg_adr_i ),
    .dina (kickOff ? 32'h00000000 : upg_dat_i ),
    .douta (Instruction_o)
);
endmodule
    
```

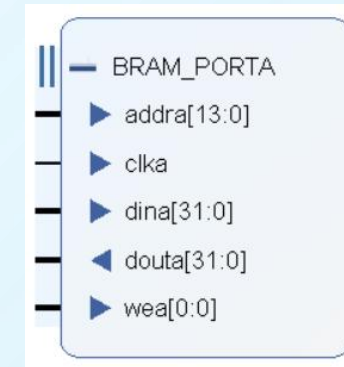



Changes on IFetch continued

Make a **new** programrom(which is a **RAM** memory):

TIPS about the "programrom" :

- While on CPU communication mode, "programrom" is writable.
- While on CPU normal mode, "programrom" is readOnly.



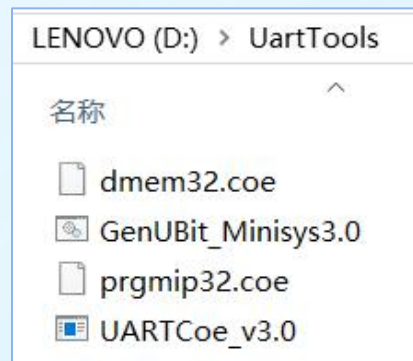
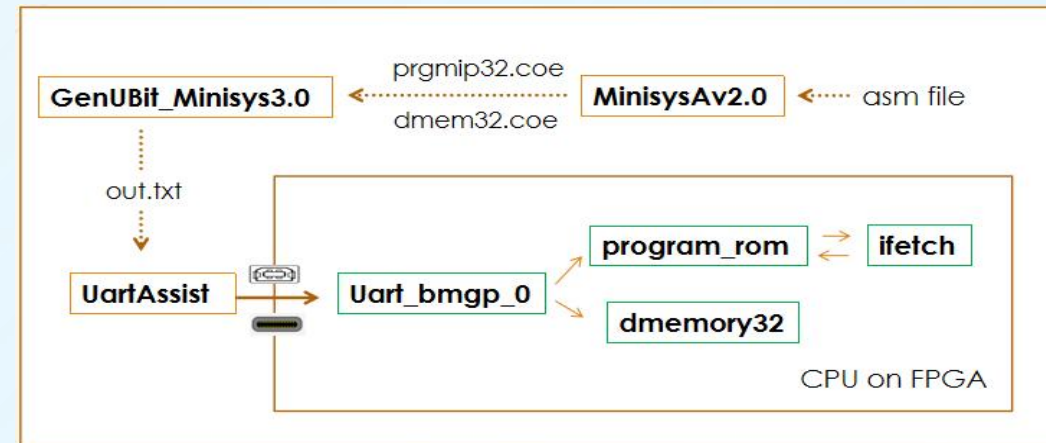
| Basic | Port A Options | Other Options | Summary |
|---|----------------------------|---|---------|
| Interface Type | Native | <input type="checkbox"/> Generate address interface with 32 | |
| Memory Type | Single Port RAM | <input type="checkbox"/> Common Clock | |
| ECC Options | | | |
| ECC Type | No ECC | | |
| <input type="checkbox"/> Error Injection Pins | Single Bit Error Injection | | |
| Write Enable | | | |
| <input type="checkbox"/> Byte Write Enable | | | |
| Byte Size (bits) | 9 | | |
| Algorithm Options | | | |
| Defines the algorithm used to concatenate the block RAM primitives. Refer datasheet for more information. | | | |
| Algorithm | Minimum Area | | |
| Primitive | 8kx2 | | |

| Basic | Port A Options | Other Options | Summary |
|--|---|-------------------------------|---------|
| Memory Size | | | |
| Write Width | 32 | Range: 1 to 4608 (bits) | |
| Read Width | 32 | | |
| Write Depth | 16384 | Range: 2 to 1048576 | |
| Read Depth | 16384 | | |
| Operating Mode Write First Enable Port Type Always Enabled | | | |
| Port A Optional Output Registers | | | |
| <input type="checkbox"/> Primitives Output Register | <input type="checkbox"/> Core Output Register | | |
| <input type="checkbox"/> SoftECC Input Register | <input type="checkbox"/> REGCEA Pin | | |
| Port A Output Reset Options | | | |
| <input type="checkbox"/> RSTA Pin (set/reset pin) | Output Reset Value (Hex) | 0 | |
| <input type="checkbox"/> Reset Memory Latch | Reset Priority | CE (Latch or Register Enable) | |
| READ Address Change A | | | |

| Basic | Port A Options | Other Options | Summary |
|---|--|---------------------------------------|---------|
| Pipeline Stages within Mux | 0 | Mux Size: 4x1 | |
| Memory Initialization | | | |
| <input checked="" type="checkbox"/> Load Init File | | | |
| Coe File | ../../../../minisys.srscs/sources_1/ip/prgrom/prgmip32.coe | <input type="button" value="Browse"/> | |
| <input checked="" type="checkbox"/> Fill Remaining Memory Locations | | | |
| Remaining Memory Locations (Hex) | 0 | | |
| Structural/UniSim Simulation Model Options | | | |
| Defines the type of warnings and outputs are generated when a read-write or write-write collision occurs. | | | |
| Collision Warnings | All | | |
| Behavioral Simulation Model Options | | | |
| <input type="checkbox"/> Disable Collision Warnings | <input type="checkbox"/> Disable Out of Range Warnings | | |

Tools(1): Generate the Data For Uart Port

- Step1: Using “MinisysAv2.0” to assemble the asm file and generate the coe files(prgmip32.coe and dmem32.coe)
- Step2: Using “GenUBit_Minisys3.0” to merge the coe files(prgmip32.coe and dmem32.coe) into one file “out.txt”
 - Tips on Step2:
 - put “prgmips32.coe” and “dmem32.coe” into the same directory with “UARTCoe_v3.0” and “GenUBit_Minisys3.0” , or you will need to make some modification on GenUBit_Minisys3.0



```
d:\UartTools>GenUBit_Minisys3.0.bat
2 files are read successfully
Hexadecimal file(s) detected.
Done.
```

“GenUBit_Minisys3.0” , “UartAssist” and “UARTCoe_v3.0” could be found in the “uart_tools.rar” of “labs/lab13_uart” on BlackBoard site



Tools(2): Using “UartAssist”

- **Step 1: Connect the Computer** which runs “UartAssist” with **Minisysboard** on which your designed CPU has already been programed on its FPGA chip.
- **Step 2:** Double click on “**UartAssist**” to **open** it
- **Step 3: Set** the items in “**串口设置**” as the settings of screen snap on the right hand, then click on “**打开**”
 - **NOTICE:** “**串口号**” could be an **serial port** other than “COM4” , which **is up to your Computer**. The port which you choose here and then click on “**打开**” hasn't report error is the right port.

串口设置

| | |
|-----|--------|
| 串口号 | COM4 |
| 波特率 | 128000 |
| 校验位 | NONE |
| 数据位 | 8 |
| 停止位 | 1 |

☒ 打开

Tips: Using “UartAssist” continued

- **Step 4:** Make sure the **CPU on FPGA works on uart communication mode.**
- **Step 5:** Set the items in “**发送区设置**” as the screen snap on the right hand. Click on “**启用文件数据源**”, find the data file which is to be transformed by uart port to FPGA chip.
- **Step 6:** Wait until a notice info “**Program done!**” has appeared in the “**串口数据接收**” window as the screen snap on the right hand.

UartAssist

接收区设置

- ☐ 接收转向文件...
- ☐ 显示接收时间
- ☐ 十六进制显示
- ☐ 暂停接收显示

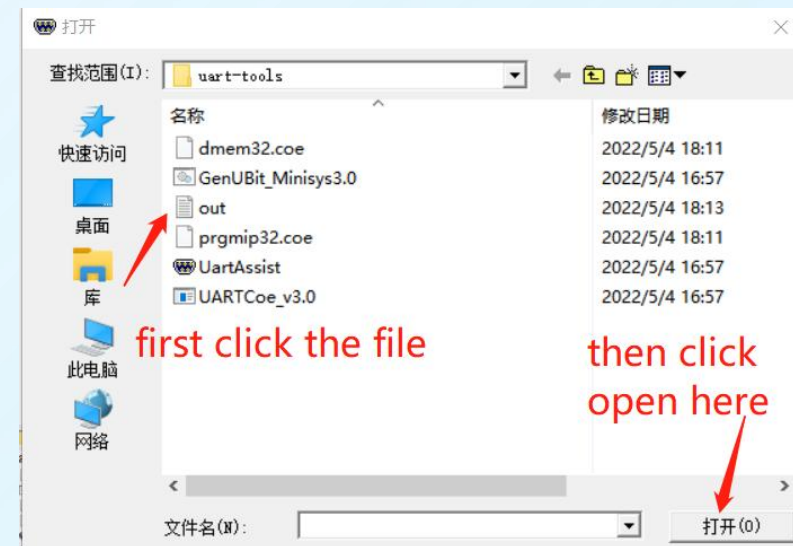
[保存数据](#) [清除显示](#)

发送区设置

- ☒ 启用文件数据源...
- ☐ 自动发送附加位
- ☐ 发送完自动清空
- ☒ 按十六进制发送
- ☐ 数据流循环发送

发送间隔 毫秒

[文件载入](#) [清除输入](#)



串口数据接收

0x00020000 bytes read. Program done!



Practice(optional)

- 1. Modify the Data-memory module, do the unit test on the updated Data-memory.
- 2. Modify the IFetch, do the unit test on the updated IFetch.
- 3. Update the CPU with uart communication implemented.
- 4. Do the test: Program FPGA chip only once, make the CPU run the different programs by using uart communication to update the instructions and data of new program.

It is strongly recommended to conduct unit test first, and then conduct integration test after passing the unit test.