# v5 BPI configuration & fallback

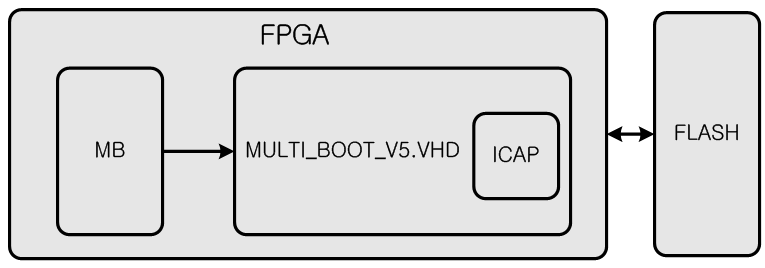
Virtex5는 여러가지 방법으로 configuration 할 수 있습니다. 이번 장에서는 FPGA 외부에 플래시 메모리를 연결해서 configuration 하는 방법에 대해서 알아보도록 하겠습니다.

virtex5는 fallback 기능이 있는데 이 기능은 FPGA가 configuration에 실패했을 경우 미리 지정된 configuration 데이터를 사용하도록 하는 기능 입니다.

이 실험을 위해 자일링스에서 출시된 ML507이라는 데모 보드를 사용 했습니다 하지만 기본적으로 이 원리는 모든 virtex5 제품에 적용 됩니다.

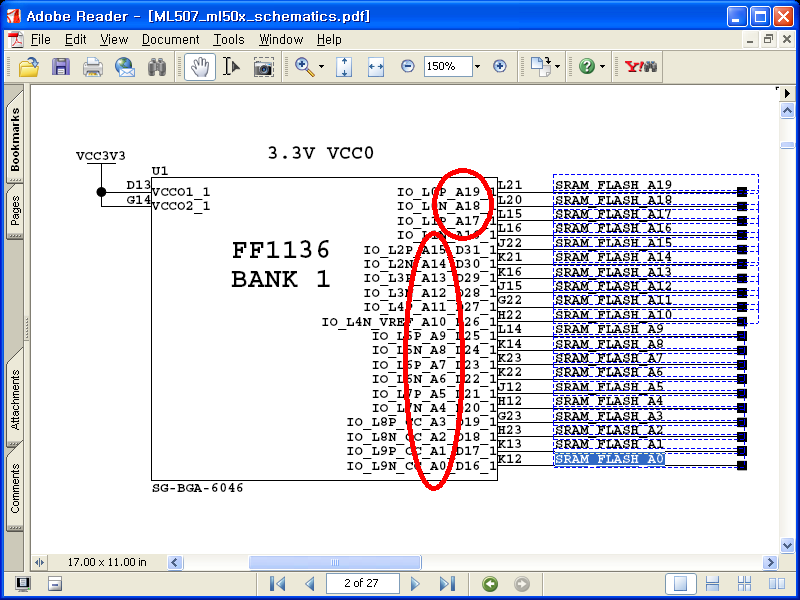
## FPGA와 플래시 메모리 연결

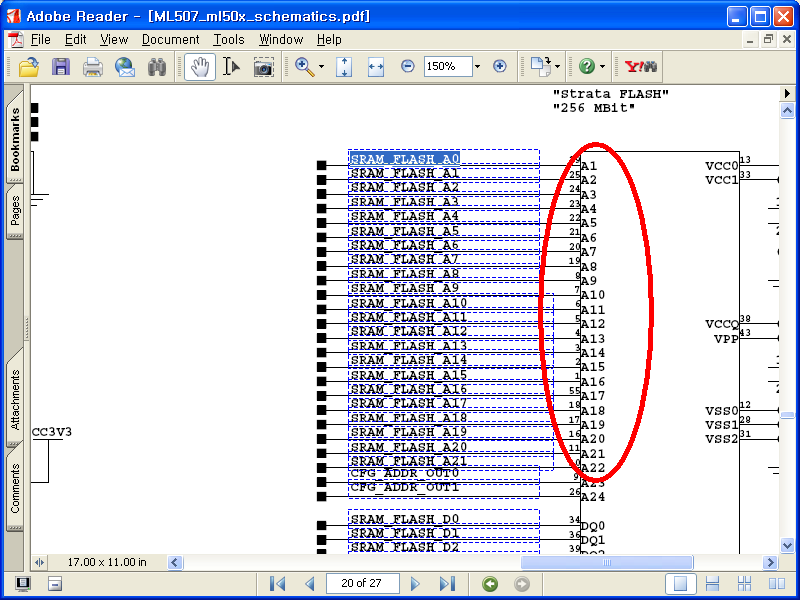
다음 그림은 BPI를 이용해 configuration을 하는 방법과 fallback이 발생했을 경우 FPGA 가 미리 지정된 configuration 데이터를 사용해서 configuration을 할 수 있도록 하기 위한 블록도 입니다.



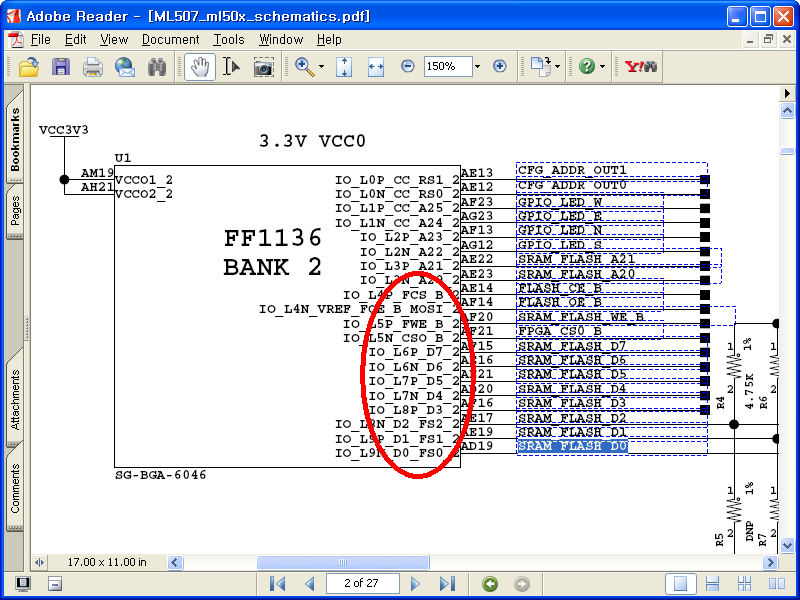
Virtex5에서 플래시 메모리를 BPI 방식의configuration용으로 사용하기 위해서는 미리 지정된 핀을 사용하도록 되어 있습니다.

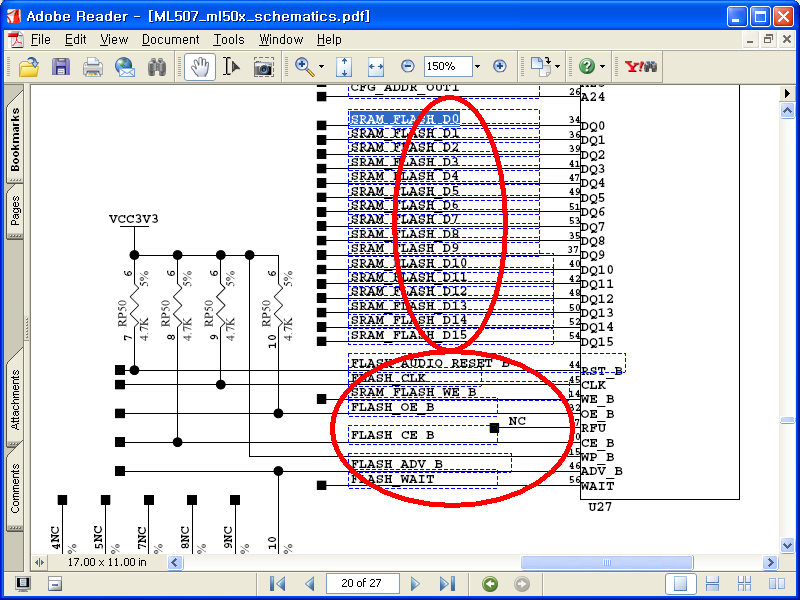
다음 2개의 그림은 플래시 메모리용 어드레스 핀을 보여주고 있습니다. 주의할 점은 플래시 메모리는 16비트 모드를 사용하기 때문에 FPGA의 A0가 플래시 메모리의 A1에 연결되어 있다는 점 입니다.



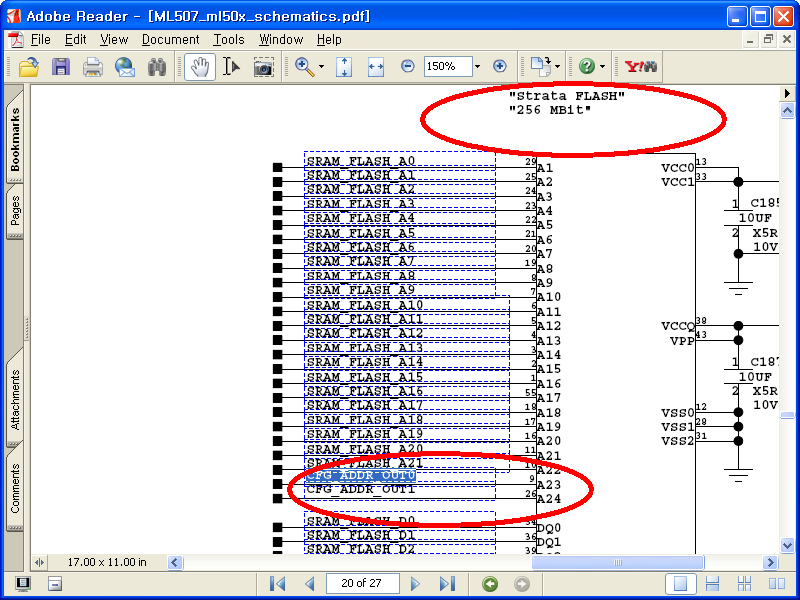


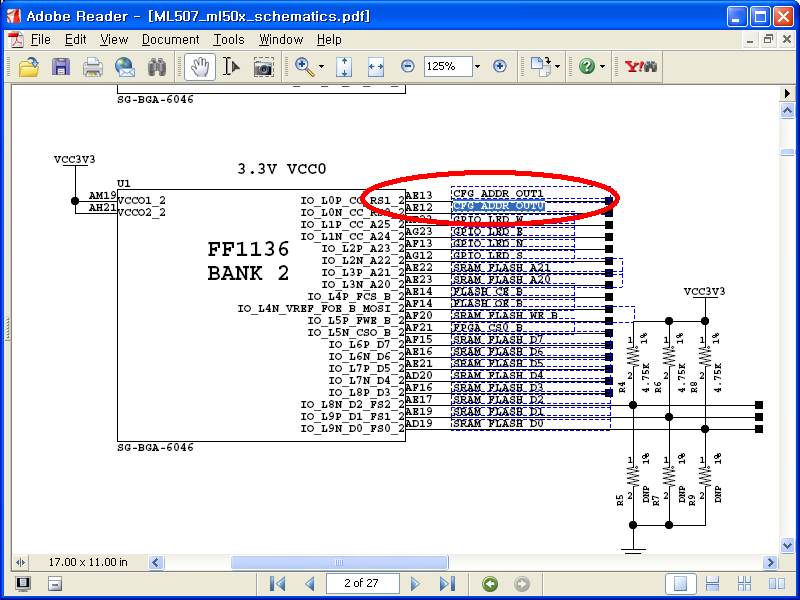
다음 2개의 그림은 FPGA와 플래시 메모리의 데이터 버스를 연결하는 것을 보여주고 있습니다. 추가로 플래시 메모리에 필요한 CS, WE, OE 도 표시되어 있습니다.





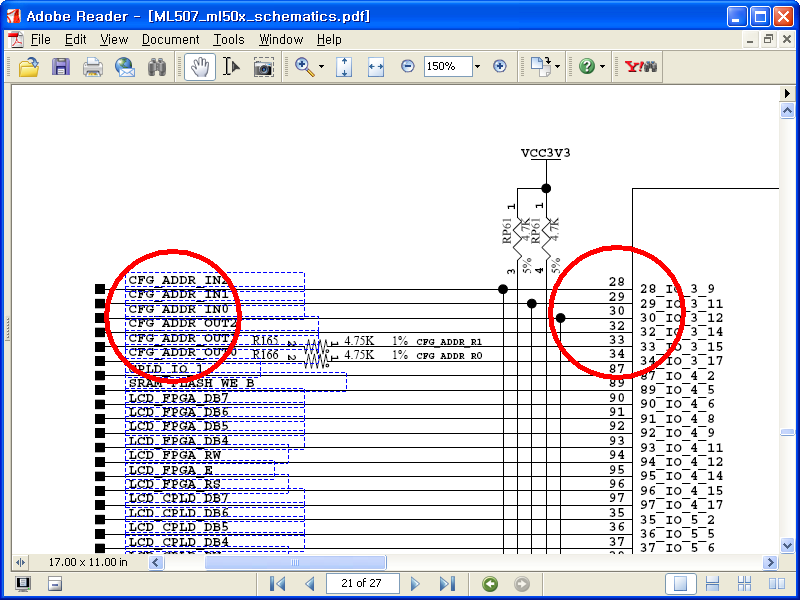
마지막으로 플래시 메모리의 최상위 어드레스 2개는 FPGA의 RS[1:0]에 연결해서 필요에 따라 다른 configuration 데이터를 선택할 수 있도록 합니다.

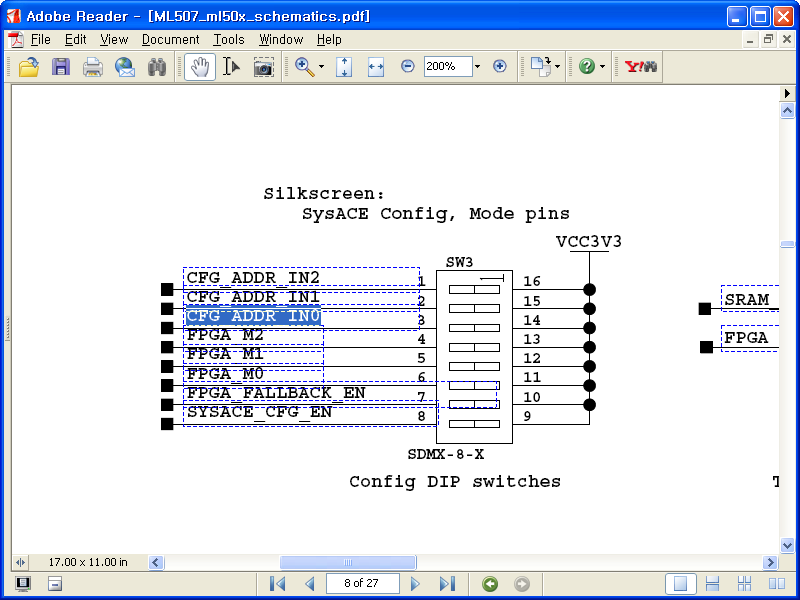




ML507보드에서는 RS[1:0]이 다음 그림과 같이 CPLD와 연결되어 있고 CPLD를 통해 DIP SW3와 연결되어 있습니다

아래 그림에서는 잘 안 보이지만 CFG\_ADDR\_IN0, CFG\_ADDR\_IN1, CFG\_ADDR\_IN2는 모두 PULLDOWN 저항으로 연결되어 있습니다.





다음 2개의 리스트는 CPLD를 통해 DIP SW3과 FPGA의 RS[1:0]이 연결되어 있다는 것을 보여 줍니다.

BPI를 위해서는 굳이 CPLD가 필요 없기 때문에 CPLD 없이 바로 DIP SW와 PULLDOWN, PULLUP 저항으로도 같은 기능 구현이 가능 합니다.

#### CPLD ucf file

NET "cfg\_addr\_out2" LOC ="P32";

NET "cfg\_addr\_out1" LOC ="P33";

NET "cfg\_addr\_out0" LOC ="P34";

NET "cfg\_addr\_in2" LOC ="P28";

NET "cfg\_addr\_in1" LOC ="P29";

NET "cfg\_addr\_in0" LOC ="P30";

#### CPLD verilog source code

output cfg\_addr\_out2;

output cfg\_addr\_out1;

output cfg\_addr\_out0;

input cfg\_addr\_in2;

input cfg\_addr\_in1;

input cfg\_addr\_in0;

assign cfg\_addr\_out2 = cfg\_addr\_in2;

assign cfg\_addr\_out1 = cfg\_addr\_in1;

assign cfg\_addr\_out0 = cfg\_addr\_in0;4

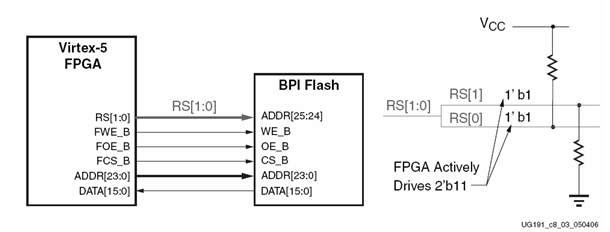
마지막으로 플래시 메모리의 최상위 어드레스 2개는 FPGA의 RS[1:0]에 연결해서 필요에 따라 다른 configuration 데이터를 선택할

다음 표는 virtex5의 configuration 방법을 정하는 MODE[2:0]핀에 관한 정의 입니다. 일반적으로 플래시 메모리를 사용할 경우 BPI-up 을 사용하기 때문에 “010” 으로 MODE 핀 값을 정합니다.



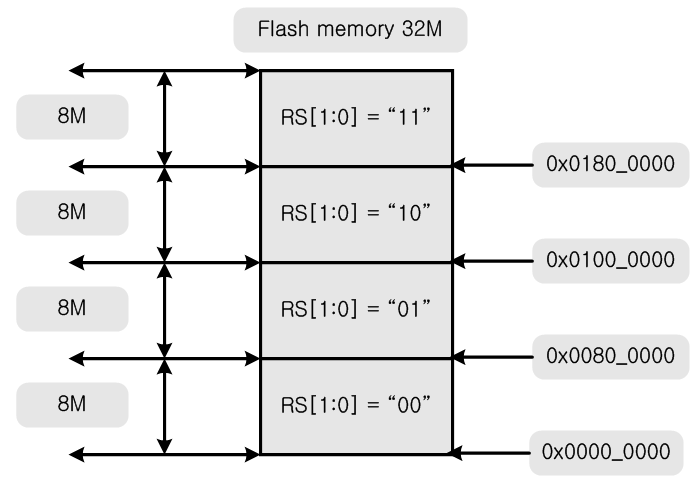
다음 그림은 virtex5와 플래시 메모리를 연결한 모습을 보여 줍니다. 플래시 메모리의 어드레스 최상위 2비트는 RS[1:0]과 연결된 것을 확인할 수 있습니다.

주의!, 이 그림은 단지 플래시 메모리 어드레스의 최상위 비트 2개를 RS[1:0]에 연결해야 한다는 것을 의미 합니다. 플래시 메모리 크기에 따라 최상위 비트는 달라질 수 있습니다. 실제 ML507보드에서는 플래시 메모리 어드레스는 [24:0] 의 크기를 가지고 있습니다.



RS[1:0]가 플래시 메모리의 최상위 어드레스 2개와 연결되어 있기 때문에 플래시 메모리에는 모두 서로 다른 4개의 configuration 데이터를 저장할 수 있습니다.

RS[1:0]은 FPGA 내부에서 weak pull up 저항 때문에 기본 값은 ‘11’을 가지고 있습니다. 따라서 RS[1:0]에 PULL-UP, PULL-DOWN 저항을 걸어 최초 configuration 데이터를 위치를 설계자가 정할 수 있습니다.

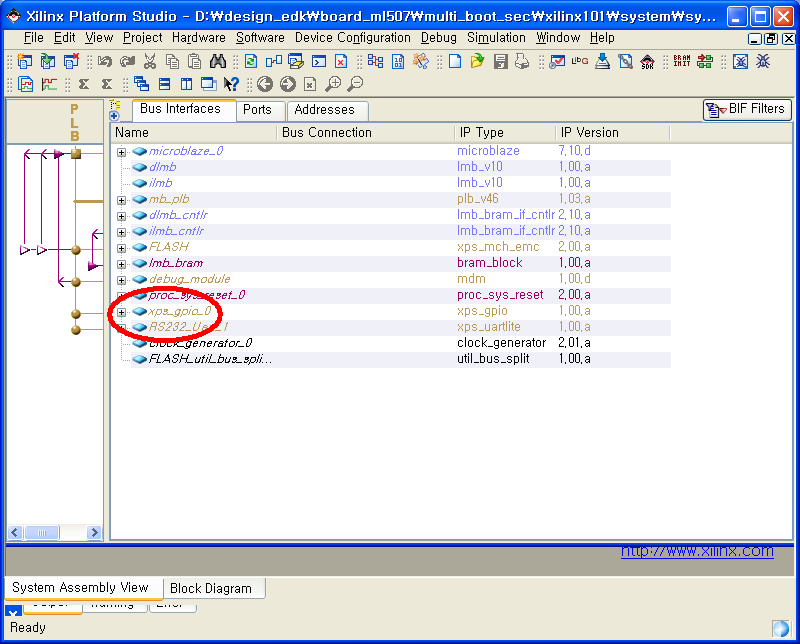


## 프로세서와 multi\_boot\_v5.vhd와 연결

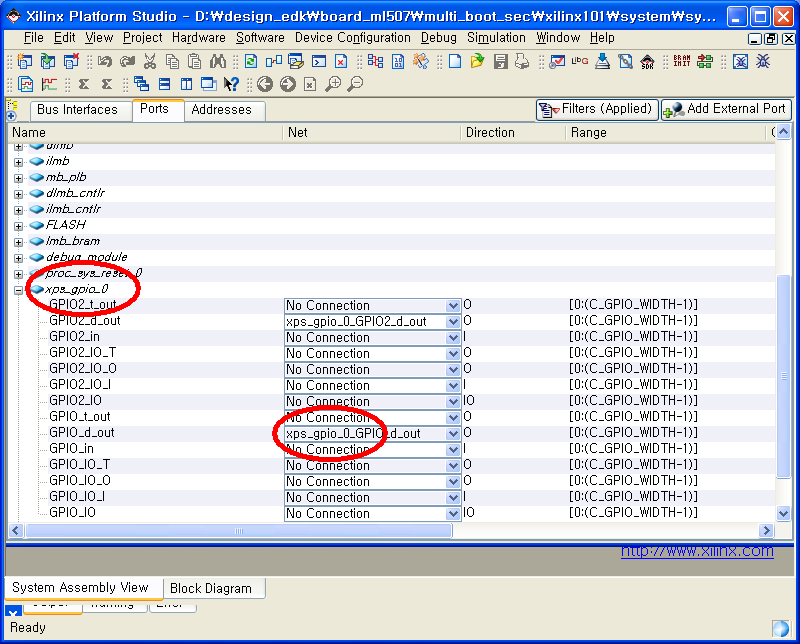
### 프로세서 블록 설계

Multi\_boot\_v5는 외부에서 trigger를 ‘1’로 만들어야 합니다. 누가 이 신호를 ‘1 로 만들어 줄 것인가는 사용자 몫이기는 하지만 여기서는 microblaze를 통해 gpio 32비트를 특정 값, 0x0094, 으로 만들어 줄 때 trigger 를 ‘1’ 로 하도록 하였습니다.

다음 그림은 자일링스를 프로세서 설계툴인 XPS (Xilinx Platform Studio)를 이용해 프로세서와 GPIO가 버스에 연결된 모습을 보여 줍니다.



다음 그림은 XPS\_GPIO\_0에서 2개의 32비트 출력 포트 중 하나를 선택한 것을 보여 줍니다.



다음 리스트는 system.mhs에서 gpio 부분만 편집한 것을 보여 줍니다.

PORT xps\_gpio\_0\_GPIO\_d\_out\_pin = xps\_gpio\_0\_GPIO\_d\_out, DIR = O, VEC = [31:0]

PORT xps\_gpio\_0\_GPIO2\_d\_out\_pin = xps\_gpio\_0\_GPIO2\_d\_out, DIR = O, VEC = [31:0]

BEGIN xps\_gpio

PARAMETER INSTANCE = xps\_gpio\_0

PARAMETER HW\_VER = 1.00.a

PARAMETER C\_IS\_BIDIR = 0

PARAMETER C\_BASEADDR = 0x81400000

PARAMETER C\_HIGHADDR = 0x8140ffff

PARAMETER C\_IS\_DUAL = 1

PARAMETER C\_IS\_BIDIR\_2 = 0

BUS\_INTERFACE SPLB = mb\_plb

PORT GPIO\_d\_out = xps\_gpio\_0\_GPIO\_d\_out

PORT GPIO2\_d\_out = xps\_gpio\_0\_GPIO2\_d\_out

END

다음 리스트는 top모듈에서 프로세서 블록을 이용해 GPIO 포트를 trigger신호를 만드는 것을 보여 줍니다. 이 trigger 값은 프로세서가 gpio에 쓴 값에 따라 ‘1’ 또는 ‘0’을 가지게 됩니다.

Inst\_system: entity work.system PORT MAP(

fpga\_0\_RS232\_Uart\_1\_RX\_pin =>fpga\_0\_RS232\_Uart\_1\_RX\_pin ,

fpga\_0\_RS232\_Uart\_1\_TX\_pin =>fpga\_0\_RS232\_Uart\_1\_TX\_pin ,

fpga\_0\_FLASH\_Mem\_A\_pin =>fpga\_0\_FLASH\_Mem\_A\_pin ,

fpga\_0\_FLASH\_Mem\_DQ\_pin =>fpga\_0\_FLASH\_Mem\_DQ\_pin ,

fpga\_0\_FLASH\_Mem\_ADV\_LDN\_pin =>fpga\_0\_FLASH\_Mem\_ADV\_LDN\_pin ,

fpga\_0\_FLASH\_Mem\_WEN\_pin =>fpga\_0\_FLASH\_Mem\_WEN\_pin ,

fpga\_0\_FLASH\_Mem\_OEN\_pin =>fpga\_0\_FLASH\_Mem\_OEN\_pin ,

fpga\_0\_FLASH\_Mem\_CEN\_pin =>fpga\_0\_FLASH\_Mem\_CEN\_pin ,

mb\_clk => clk,

xps\_gpio\_0\_GPIO\_d\_out\_pin =>xps\_gpio\_0\_GPIO\_d\_out\_pin,

xps\_gpio\_0\_GPIO2\_d\_out\_pin =>xps\_gpio\_0\_GPIO2\_d\_out\_pin,

sys\_clk\_pin =>sys\_clk\_pin,

sys\_rst\_pin =>sys\_rst\_pin

);

process(clk)

begin

if clk'event and clk='1' then

if xps\_gpio\_0\_GPIO\_d\_out\_pin = x"0094" then

trigger <= '1';

else

trigger <= '0';

end if;

end if;

end process;

## C 코드와 BIN 파일 만들기

플래시 메모리에 모두 4개의 configuration 데이터를 기록하기 위해서는 모두 4개의 서로 다른 BIT파일을 만들어야 합니다.

이 예제에서는 하드웨어를 수정하지 않고 프로세서가 출력할 메시지를 다르게 해서 서로 다른 4개의 BIT파일을 만듭니다.

다음 리스트는 프로세서가 사용할 프로그램으로 각 configuration 데이터는 각각 다른 메시지를 출력 합니다.

#include "xparameters.h"

#include "xgpio\_l.h"

void main()

{

xil\_printf("\n\rhello it is 1st image %s %s", \_\_DATE\_\_, \_\_TIME\_\_);

xil\_printf("\n\r press anykey to reboot");

inbyte();

XGpio\_mSetDataReg(XPAR\_XPS\_GPIO\_0\_BASEADDR, 1, 0x00000000);

XGpio\_mSetDataReg(XPAR\_XPS\_GPIO\_0\_BASEADDR, 1, 0x94);

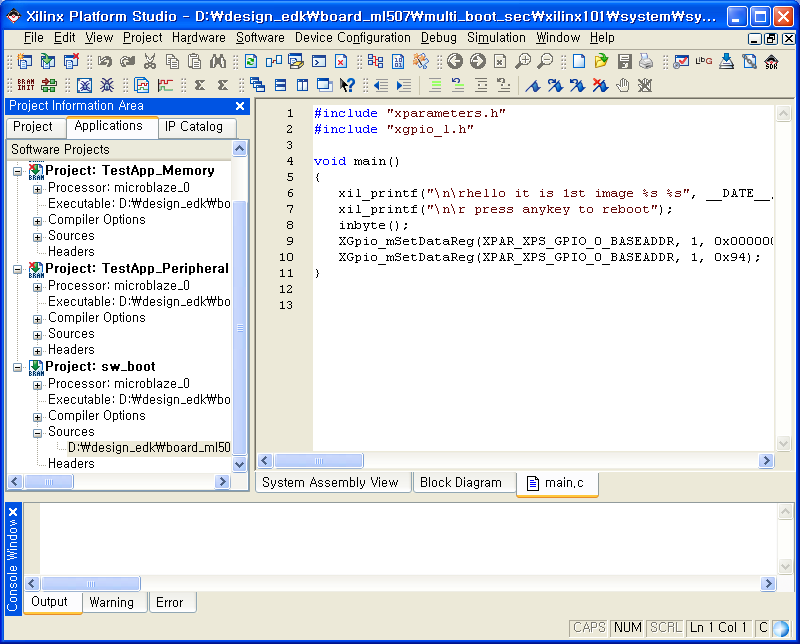
}

참고로 현재 microblaze용 프로그램은 모두 FPGA 내부에 있는 BRAM에서 있습니다. 다음 그림은 프로세서용 프로그램을 수정하기 위한 XPS 화면을 보여 줍니다.

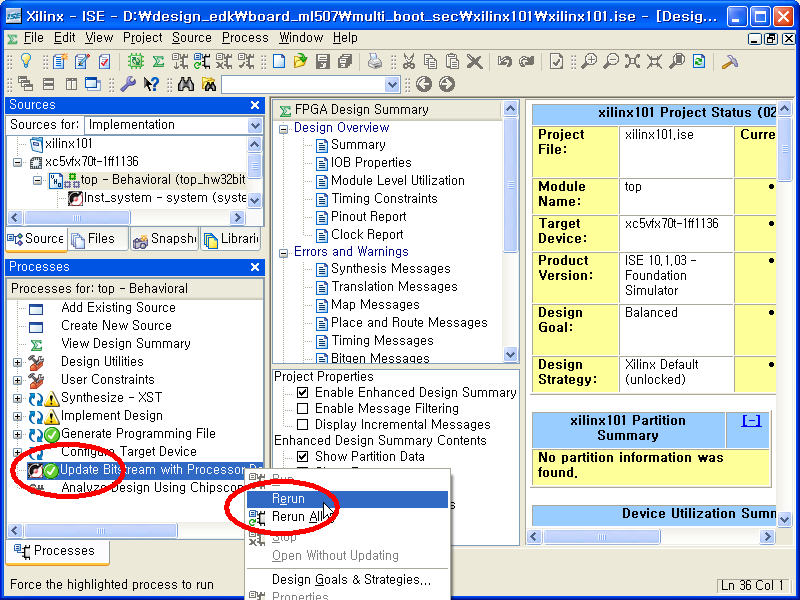
다음 비디오 클립을 통해 플래시 메모리용 bin 파일을 만드는 과정을 참조 하시기 바랍니다.

[D:\design\_edk\board\_ml507\multi\_boot\_sec\xilinx101\camtasia\wmv\how\_to\_gen\_bin\_file](file:///C:\design_edk\board_ml507\multi_boot_sec\xilinx101\camtasia\wmv\how_to_gen_bin_file\how_to_gen_bin_file.wmv)

XPS에서 소프트웨어 코드를 수정하고 저장 합니다.

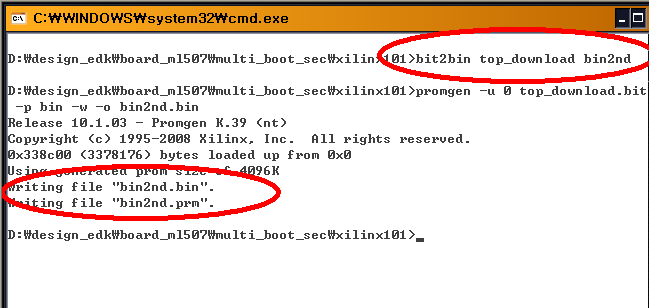


ISE에서 Update Bistream with Processor를 선택하면 ISE는 소프트웨어를 컴파일하고 하드웨어에 소프트웨어 이미지를 갱신 합니다.



Update Bistream을 선택하면 기존 top.bit 파일에 프로세서용 프로그램을 갱신하고 나서 top\_download.bit 파일을 만들어 줍니다.

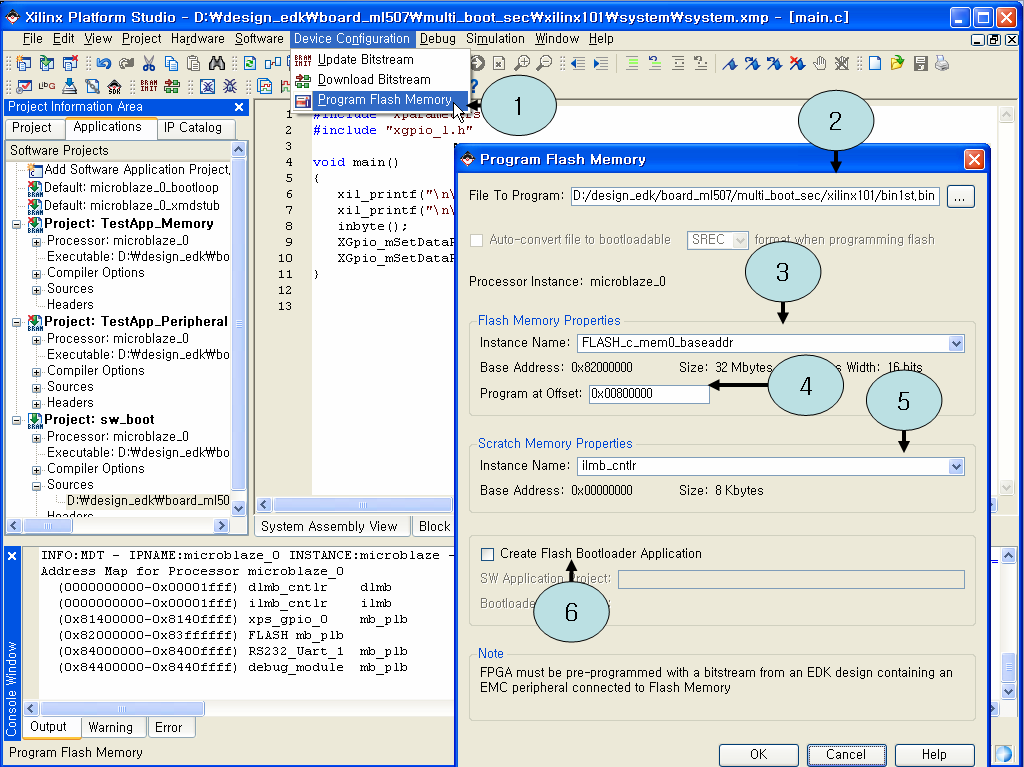
플래시 메모리에는 bit 파일 대신 bin 파일 포맷을 사용하는데 다음과 같이 간단히 배치 파일을 만들어서 bit 파일을 bin 파일로 만들어 줍니다.



다음 리스트는 BIT2BIN 배치파일 입니다.

promgen -u 0 %1.bit -p bin -w -o %2.bin

다음 그림은 XPS에서 FPGA에 연결된 플래시 메모리에 BIN파일을 쓰기위한 화면 입니다.



* Program Flash Memory

현재 FPGA는 플래시 메모리와 연결되어 있고 프로세서는 플래시 메모리 콘트롤러를 연결하고 있어야 합니다. 물론 FPGA는 현재 configuration이 완료된 상태여야 합니다.

* File to Program

Bit2bin 배치파일로 만든 bin 파일을 선택 합니다.

* Flash Memory Properties, Program at Offset

XPS는 플래시 메모리가 연결된 콘트롤러를 자동 선택하고 offset 번지를 정하도록 합니다.

현재 32MBYTE의 플래시 메모리 최상위 번지가 RS[1:0]에 연결되어 있으니까 각각 8Mbyte 단위로 영역을 지정해야 합니다. 2번째이니까 0x0080\_0000로 offset을 정합니다.

참고로 2^20은 1M 입니다.

* Create Flash Bootloader Application

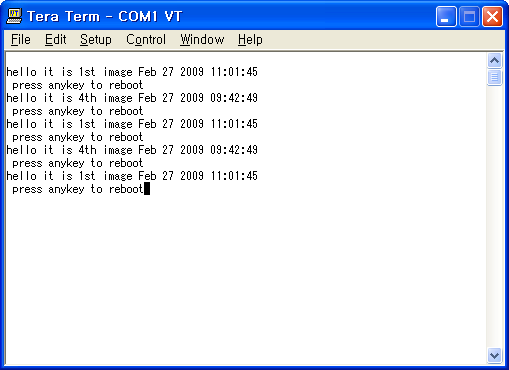
이 예제에서는 부트로더는 사용하지 않습니다.

다음 비디오 클립은 모두 4개의 BIN 파일을 만들어 각각의 플래시 메모리 영역에 BIN 파일을 기록하는 과정을 보여 줍니다.

[XXXXXXXXXXXXXXXXXX](file:///C:\Users\hyukk\Documents\spi.zip)

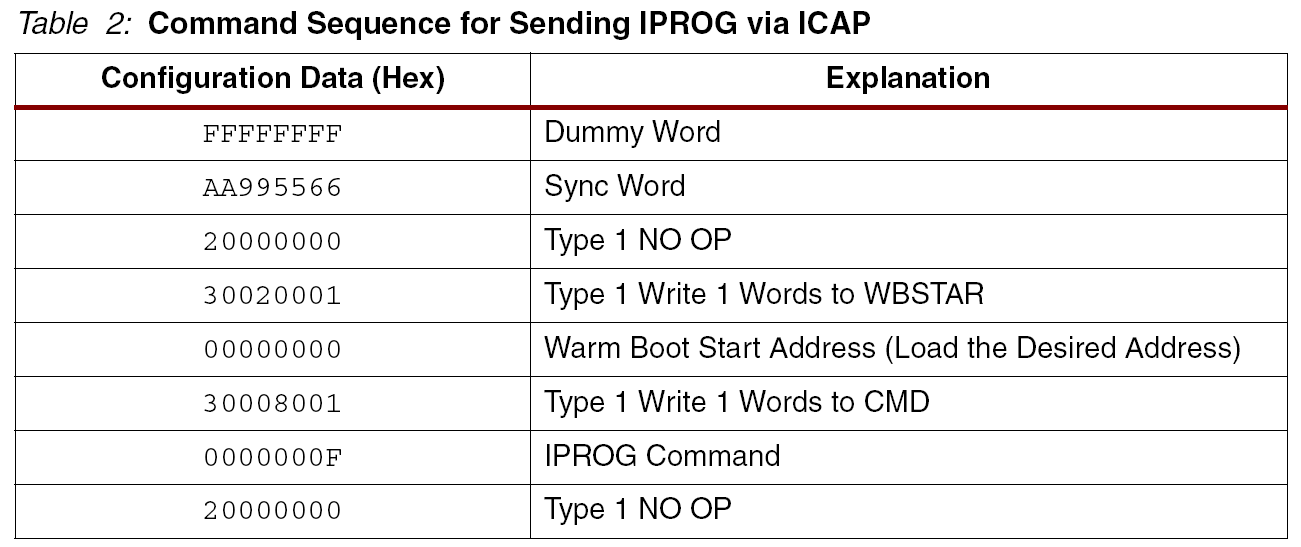
#### 출력 화면

다음 그림은 첫번째 이미지로 configuration을 하고 나서 다시 네번째 이미지로 다시 configuration을 하는 모습을 반복적으로 보여 줍니다.

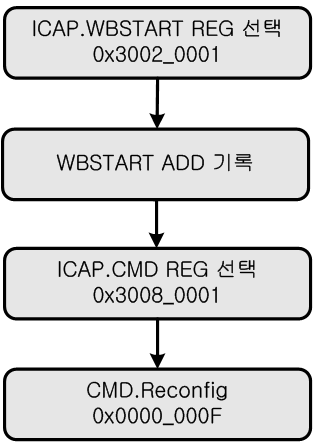


## multi\_boot\_v5.vhd

ICAP을 통해 FPGA를 configuration하기 위해서는 다음 그림과 같이 ICAP 단자에 일정한 값을 순서대로 넣어 주어야 합니다.



다음 그림과 리스트는 데이터 쉬트에 정의된 값을 차례대로 넣어 주기 위해 SIGNAL을 선언하고 초기화 하는 과정을 보여 줍니다.



signal icap\_ce : std\_logic := '0';

signal icap\_write : std\_logic := '0';

signal icap\_clk : std\_logic := '0';

signal icap\_prepare : std\_logic := '0';

signal icap\_next\_en : std\_logic := '0';

signal icap\_word0 : std\_logic\_vector(31 downto 0) := x"00000000";

signal icap\_word1 : std\_logic\_vector(31 downto 0) := x"AA995566"; -- sync

signal icap\_word2 : std\_logic\_vector(31 downto 0) := x"20000000"; -- type1 no op

signal icap\_word3 : std\_logic\_vector(31 downto 0) := x"30020001"; -- type1 write1 wbstart

signal icap\_word4 : std\_logic\_vector(31 downto 0) := x"1C000000"; -- warm boot start add

signal icap\_word5 : std\_logic\_vector(31 downto 0) := x"30008001"; -- type1 write1 word to cmd

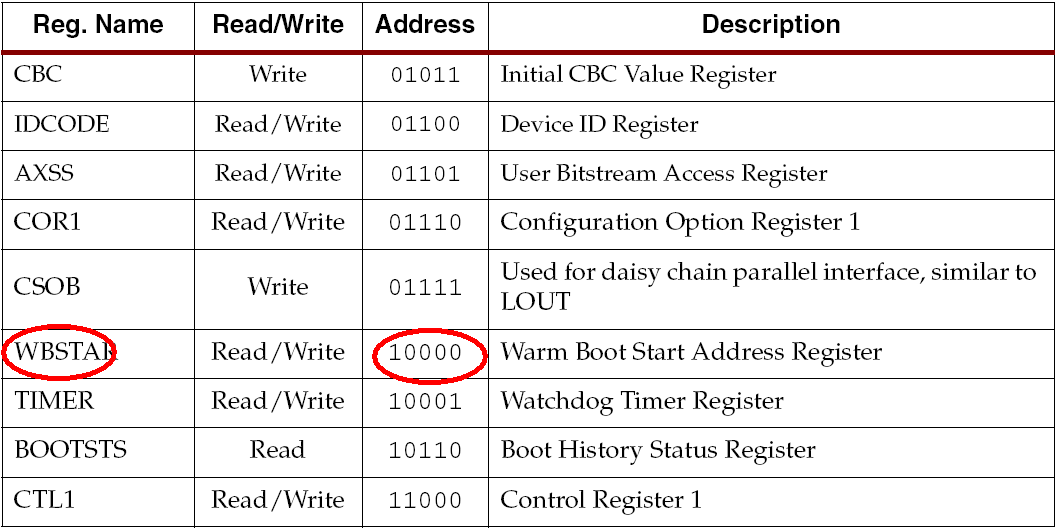
signal icap\_word6 : std\_logic\_vector(31 downto 0) := x"0000000F"; -- iprog cmd

signal icap\_word7 : std\_logic\_vector(31 downto 0) := x"20000000"; -- type1 no op

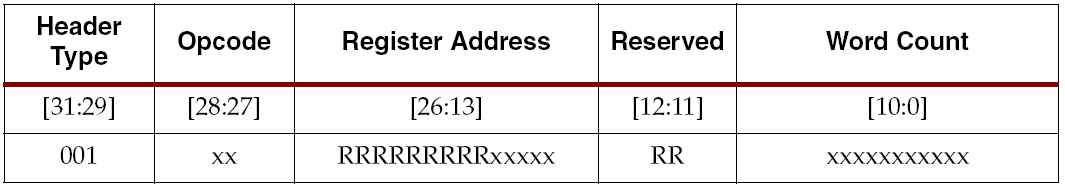
signal icap\_word8 : std\_logic\_vector(31 downto 0) := x"AAAABBBB"; -- dummy

signal icap\_write\_en : std\_logic\_vector(8 downto 0) := "100000001"; --active low write enable

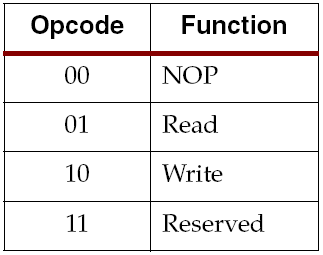
Warm Boot Start Add 레지스터는 ICAP 내부에 있는 TYPE1 레지스터 중 하나이며 어드레스로는 0x1\_0000 값을 가지고 있습니다.



이 레지스터에 특정 값을 써주기 위해서는 다음과 같은 type1 명령어 형식을 만들어야 ICAP에 전달해야 합니다.



OPCode는 다음과 같이 정해 집니다.



다음 리스트 BOOTSTS 레지스터를 읽기 위한 값을 찾아내는 절차를 보여 줍니다.

001\_01\_RRRRRRRRR\_10110\_RR\_00000000001

001\_0\_\_1RRR\_\_RRRR\_\_RR10\_\_110R\_\_R\_000\_\_0000\_\_0001

001\_0\_\_1000\_\_0000\_\_0010\_\_1100\_\_0\_000\_0000\_0001

0010\_1000\_0000\_0010\_1100\_0000\_0000\_0001

0x2802\_C001

이 같은 방식으로 정해진 WBSTART 레지스터에 값을 쓰기 위한 명령어는 다음과 같이 정해 집니다.

0x3002\_0001 값이 나오게 된 절차는 아래와 같습니다

**001\_1**0\_RRRRRRRRRRR\_10000\_RR\_00000000001

여기서 R는 Reserved로 모도 0으로 처리 합니다.

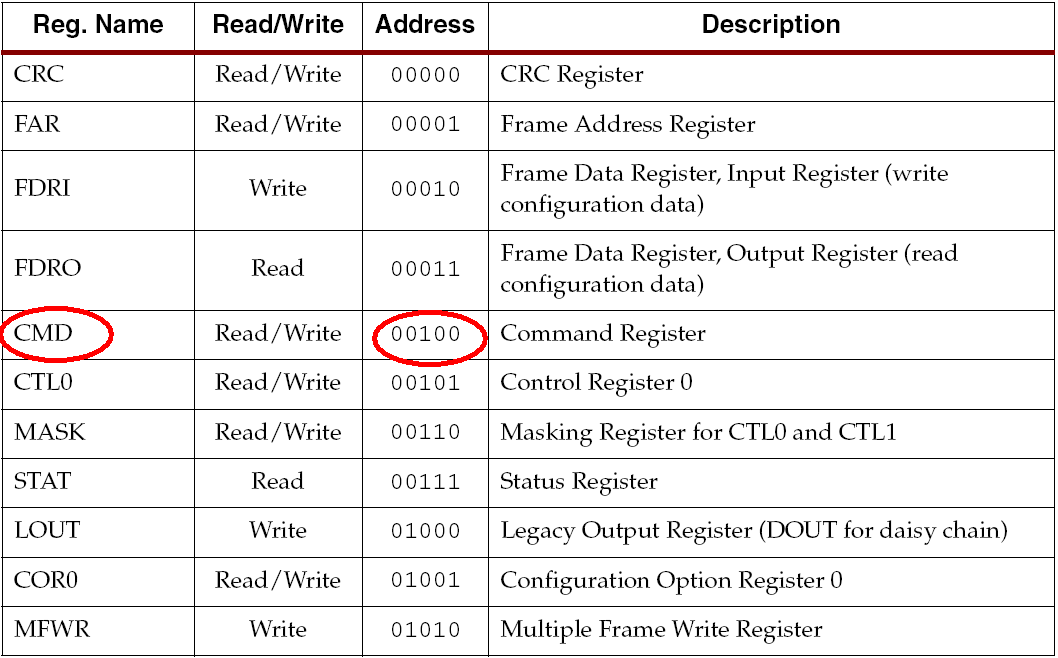
**0011\_\_**0RRR\_\_RRRR\_\_RR10\_\_000R\_\_R\_000\_\_0000\_\_0001

이것을 다시 4비트 단위로 나눕니다.

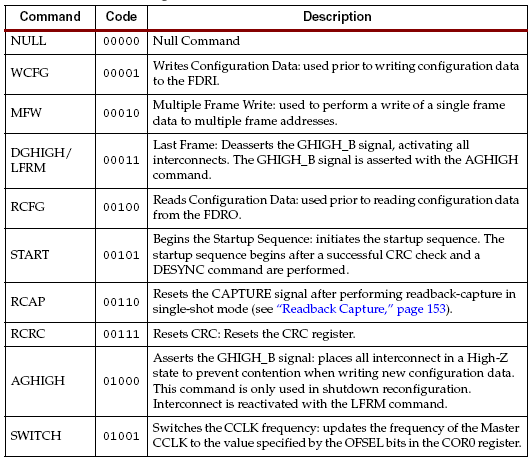
3\_0\_0\_2\_0\_0\_0\_1

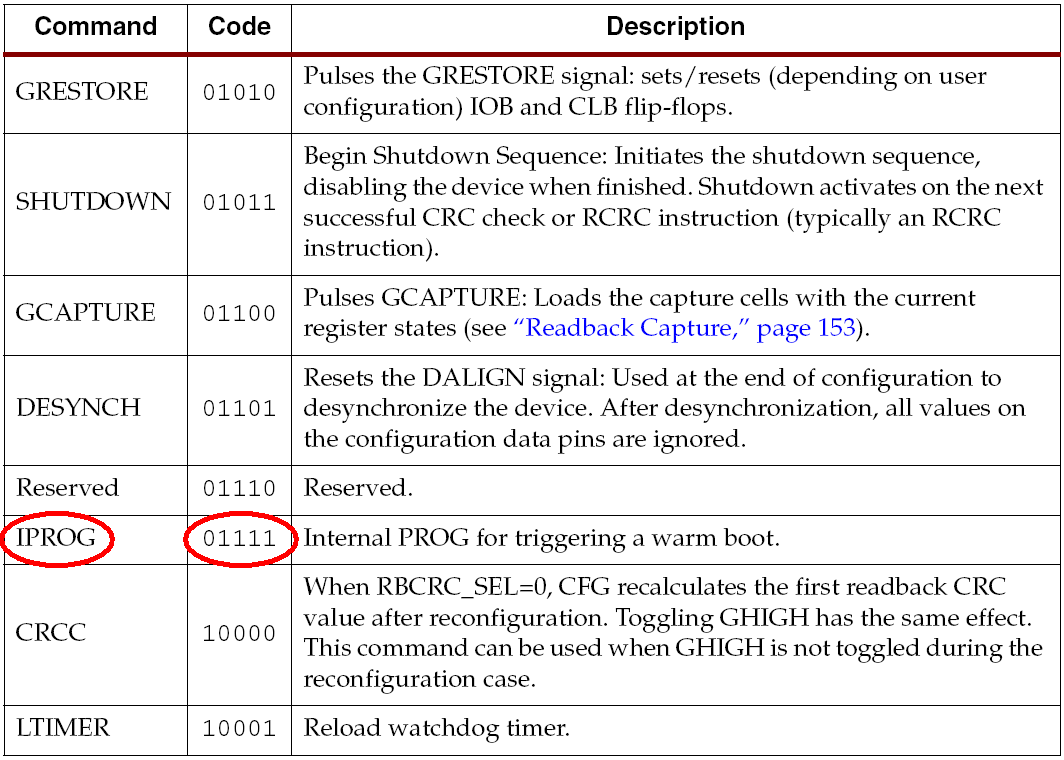
조합을 하면0x30020001 값을 가지게 됩니다

다음 표는 ICAP내부에 있는 레지스터 이름과 오프셋 값입니다..



다음 표는 ICAP에 요청할 수 있는 명령어 리스트 입니다.



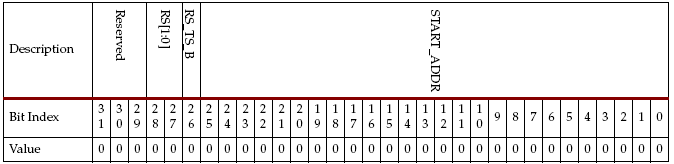


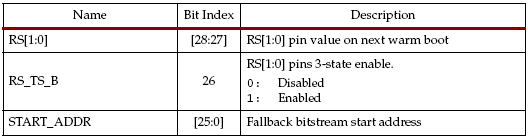
Warm Boot Start Add를 정하는 방법은 다음 테이블에 정의되어 있습니다. 이 테이블에 의하면 0x1C00\_0000은 RS[1:0] = “11” 이 되고 RS\_TS\_B도 ‘1’ 값을 가지게 됩니다.

나머지 어드레스는 모두 0x0이 되므로 ML507보드에 있는 플래시 메모리 어드레스는 [24:23] = “11” 나머지는 ‘0’으로 정해지기 때문에 0x0180\_0000에서 시작하는 configuration 데이터를 사용해 configuration을 시작할 것 입니다.

만일 플래시 메모리 0x0 에서 시작하는 configuration 데이터를 사용한다면 이 레지스터의 값은 0x0400\_0000 이 됩니다.

항상 RS\_TS\_B는 ‘1’이 되어야 RS[1:0]에 의해 정해진 플래시 메모리 어드레스를 사용할 수 있습니다.





외부에서 trigger가 ‘1’ 이 되면 다음 프로세서 문에 의해서 icap\_word# 에 저장된 값이 하나씩 옆으로 쉬프트 되면서 ICAP에 입력 됩니다.

icap\_control : process(clk)

begin

if clk'event and clk = '1' then

if((trigger = '1' and icap\_clk = '0' and icap\_prepare = '0') or icap\_next\_en = '1') then

icap\_prepare <= '1';

else

icap\_prepare <= '0';

end if;

if icap\_prepare = '1' then

icap\_clk <= '1';

else

icap\_clk <= '0';

end if;

if icap\_clk = '1' then

icap\_next\_en <= '1';

else

icap\_next\_en <= '0';

end if;

if icap\_next\_en = '1' then

icap\_word0 <= icap\_word1;

icap\_word1 <= icap\_word2;

icap\_word2 <= icap\_word3;

icap\_word3 <= icap\_word4;

icap\_word4 <= icap\_word5;

icap\_word5 <= icap\_word6;

icap\_word6 <= icap\_word7;

icap\_word7 <= icap\_word8;

icap\_word8 <= icap\_word0;

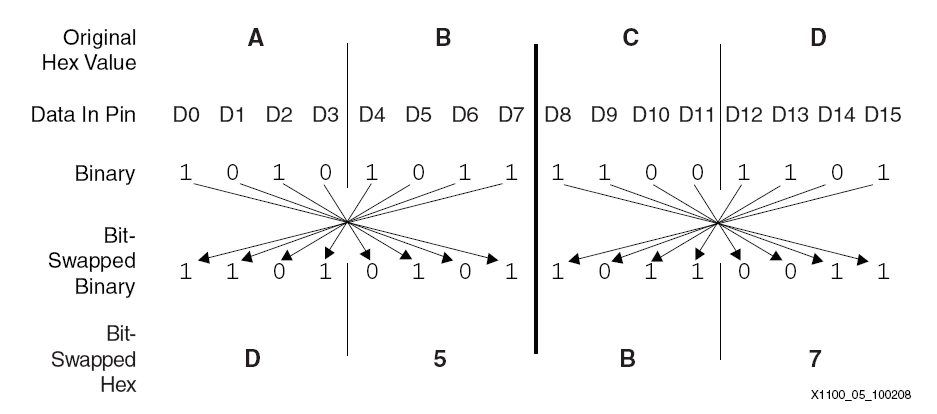
icap\_write\_en <= icap\_write\_en(0) & icap\_write\_en(8 downto 1);

end if;

end if;

end process icap\_control;

실제 ICAP에 입력될 때는 다음 그림과 같이 비트 단위로 위치를 바꿔줘야 합니다. Generic 문을 사용해 4개 바이트를 각각 그 위치를 바꾸어 줍니다.



icap\_rev0 : for i in 0 to 7 generate

icap\_i(i) <= icap\_word0(7 - i);

end generate icap\_rev0;

icap\_rev1 : for i in 8 to 15 generate

icap\_i(i) <= icap\_word0(15 - (i - 8));

end generate icap\_rev1;

icap\_rev2 : for i in 16 to 23 generate

icap\_i(i) <= icap\_word0(23 - (i - 16));

end generate icap\_rev2;

icap\_rev3 : for i in 24 to 31 generate

icap\_i(i) <= icap\_word0(31 - (i - 24));

end generate icap\_rev3;

다음 코드는 multi\_boot\_v5.vhd 전체 코드 입니다.

library IEEE;

use IEEE.Std\_Logic\_1164.all;

use IEEE.numeric\_std.all;

library unisim;

use unisim.vcomponents.all;

entity multi\_boot\_v5 is

port (

trigger: in std\_logic;

clk : in std\_logic);

end multi\_boot\_v5;

architecture Behavioral of multi\_boot\_v5 is

component ICAP\_VIRTEX5

generic ( ICAP\_WIDTH : string := "X32");

port (

BUSY : out std\_ulogic; -- Busy output

O : out std\_logic\_vector (31 downto 0); -- 32-bit data output

CE : in std\_ulogic; -- Clock enable input

CLK : in std\_ulogic; -- Clock input

I : in std\_logic\_vector (31 downto 0); -- 32-bit data input

WRITE : in std\_ulogic -- Write input

);

end component;

--ICAP signals

signal icap\_ce : std\_logic := '0';

signal icap\_write : std\_logic := '0';

signal icap\_clk : std\_logic := '0';

signal icap\_prepare : std\_logic := '0';

signal icap\_next\_en : std\_logic := '0';

signal icap\_i : std\_logic\_vector(31 downto 0);

signal icap\_word0 : std\_logic\_vector(31 downto 0) := x"00000000";

signal icap\_word1 : std\_logic\_vector(31 downto 0) := x"AA995566"; -- sync

signal icap\_word2 : std\_logic\_vector(31 downto 0) := x"20000000"; -- type1 no op

signal icap\_word3 : std\_logic\_vector(31 downto 0) := x"30020001"; -- type1 write1 wbstart

signal icap\_word4 : std\_logic\_vector(31 downto 0) := x"1C000000"; -- warm boot start add

signal icap\_word5 : std\_logic\_vector(31 downto 0) := x"30008001"; -- type1 write1 word to cmd

signal icap\_word6 : std\_logic\_vector(31 downto 0) := x"0000000F"; -- iprog cmd

signal icap\_word7 : std\_logic\_vector(31 downto 0) := x"20000000"; -- type1 no op

signal icap\_word8 : std\_logic\_vector(31 downto 0) := x"AAAABBBB"; -- dummy

signal icap\_write\_en : std\_logic\_vector(8 downto 0) := "100000001"; --active low write enable

begin

ICAP\_VIRTEX5\_inst : ICAP\_VIRTEX5

generic map (ICAP\_WIDTH => "X32") -- "X8" or "X32

port map (

BUSY => open, -- Busy output

O => open, -- 32-bit data output

CE => icap\_ce, -- Clock enable input

CLK => icap\_clk, -- Clock input

I => icap\_i, -- 32-bit data input

WRITE => icap\_write -- Write input

);

icap\_control : process(clk)

begin

if clk'event and clk = '1' then

if((trigger = '1' and icap\_clk = '0' and icap\_prepare = '0') or icap\_next\_en = '1') then

icap\_prepare <= '1';

else

icap\_prepare <= '0';

end if;

if icap\_prepare = '1' then

icap\_clk <= '1';

else

icap\_clk <= '0';

end if;

if icap\_clk = '1' then

icap\_next\_en <= '1';

else

icap\_next\_en <= '0';

end if;

if icap\_next\_en = '1' then

icap\_word0 <= icap\_word1;

icap\_word1 <= icap\_word2;

icap\_word2 <= icap\_word3;

icap\_word3 <= icap\_word4;

icap\_word4 <= icap\_word5;

icap\_word5 <= icap\_word6;

icap\_word6 <= icap\_word7;

icap\_word7 <= icap\_word8;

icap\_word8 <= icap\_word0;

icap\_write\_en <= icap\_write\_en(0) & icap\_write\_en(8 downto 1);

end if;

end if;

end process icap\_control;

icap\_write <= icap\_write\_en(0);

icap\_ce <= icap\_write\_en(0);

icap\_rev0 : for i in 0 to 7 generate

icap\_i(i) <= icap\_word0(7 - i);

end generate icap\_rev0;

icap\_rev1 : for i in 8 to 15 generate

icap\_i(i) <= icap\_word0(15 - (i - 8));

end generate icap\_rev1;

icap\_rev2 : for i in 16 to 23 generate

icap\_i(i) <= icap\_word0(23 - (i - 16));

end generate icap\_rev2;

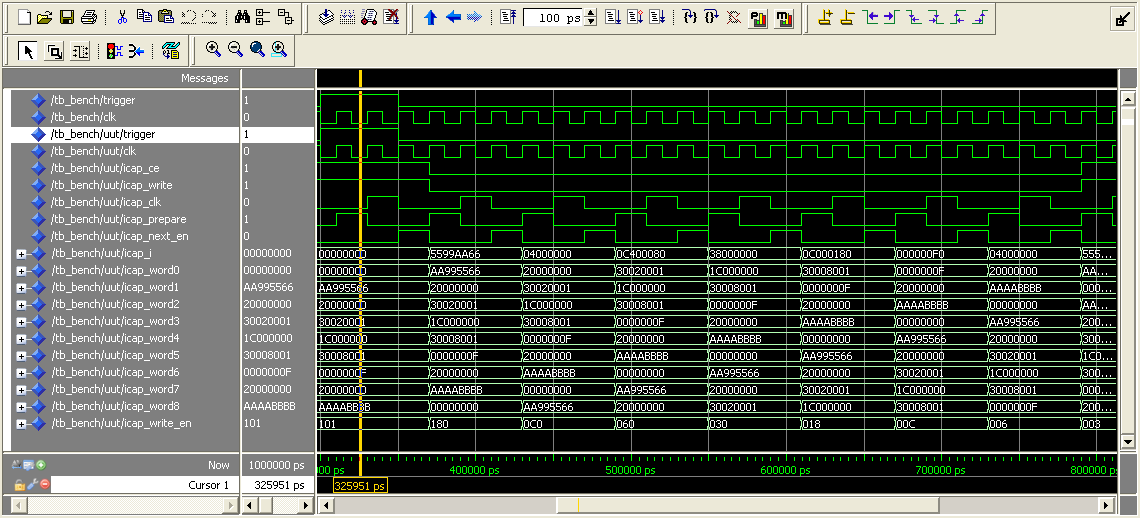
icap\_rev3 : for i in 24 to 31 generate

icap\_i(i) <= icap\_word0(31 - (i - 24));

end generate icap\_rev3;

end Behavioral ;

다음 그림은 모델심을 통해 ICAP에 원하는 값을 차례대로 쓰는 과정을 보여 줍니다.



다음 리스트는 모델심용 테스트 벤치 파일 입니다.

LIBRARY ieee;

USE ieee.std\_logic\_1164.ALL;

USE ieee.std\_logic\_unsigned.all;

USE ieee.numeric\_std.ALL;

ENTITY tb\_bench IS

END tb\_bench;

ARCHITECTURE behavior OF tb\_bench IS

-- Component Declaration for the Unit Under Test (UUT)

COMPONENT multi\_boot\_v5

PORT(

trigger : IN std\_logic;

clk : IN std\_logic

);

END COMPONENT;

--Inputs

signal trigger : std\_logic := '0';

signal clk : std\_logic := '0';

BEGIN

clk <= not clk after 10 ns;

trigger <= '1' after 300 ns , '0' after 350 ns;

-- Instantiate the Unit Under Test (UUT)

uut: multi\_boot\_v5 PORT MAP (

trigger => trigger,

clk => clk

);

END;

## SIGNAL을 초기화 하는 것에 대해 좀더 깊이 이해하기

앞에 예제에서는 SIGNAL을 선언하면서 동시에 초기화 했습니다. 따라서 리셋 조건에 의해서 로직을 초기화 하지 않아도 원하는 결과를 얻을 수 있기 때문에 리셋에 필요한 FPGA 내부에 있는 라우팅 채널을 추가로 사용하지 않았습니다.

Multi\_boot\_v5.vhd는 기본적으로 ICAP에 데이터를 순서대로 써 주는 일을 해 줍니다. 이 예제에서는 쉬프트 연산을 수행하는 부분이 있는데 리셋이 없기 때문에 FPGA 내부에 있는 LUT를 쉬프트 레지스터로 사용하는 것이 가능해 졌습니다.

만일 리셋이나 그 밖에서 다른 콘트롤 로직이 삽입되는 경우에는 앞에 선언된 모든 SIGNAL은 실제 슬라이스에 있는 플립플롭으로 사용되기 때문에 전체적인 로직 사용량은 늘어나게 됩니다.

다음 2가지 예를 통해 그 차이를 살펴보겠습니다.

### 쉬프트 연산으로만 이루어진 MUTLI\_BOOT\_V5.VHD

다음 리스트는 앞에서 설명한 multi\_boot\_v5.vhd와 같습니다. 이때는 SIGNAL이 모두 원하는 값으로 초기화 되어 있고 클럭과 클럭 인에이블 신호에 맞춰 각 SIGNAL 값이 차례대로 ICAP으로 전달됩니다.

library IEEE;

use IEEE.Std\_Logic\_1164.all;

use IEEE.numeric\_std.all;

library unisim;

use unisim.vcomponents.all;

entity multi\_boot\_v5 is

port (

trigger: in std\_logic;

boot\_addr : in std\_logic\_vector(31 downto 0);

clk : in std\_logic);

end multi\_boot\_v5;

architecture Behavioral of multi\_boot\_v5 is

component ICAP\_VIRTEX5

generic ( ICAP\_WIDTH : string := "X32");

port (

BUSY : out std\_ulogic; -- Busy output

O : out std\_logic\_vector (31 downto 0); -- 32-bit data output

CE : in std\_ulogic; -- Clock enable input

CLK : in std\_ulogic; -- Clock input

I : in std\_logic\_vector (31 downto 0); -- 32-bit data input

WRITE : in std\_ulogic -- Write input

);

end component;

--ICAP signals

signal icap\_ce : std\_logic := '0';

signal icap\_write : std\_logic := '0';

signal icap\_clk : std\_logic := '0';

signal icap\_prepare : std\_logic := '0';

signal icap\_next\_en : std\_logic := '0';

signal icap\_i : std\_logic\_vector(31 downto 0);

signal icap\_word0 : std\_logic\_vector(31 downto 0) := x"00000000";

signal icap\_word1 : std\_logic\_vector(31 downto 0) := x"AA995566"; -- sync

signal icap\_word2 : std\_logic\_vector(31 downto 0) := x"20000000"; -- type1 no op

signal icap\_word3 : std\_logic\_vector(31 downto 0) := x"30020001"; -- type1 write1 wbstart

signal icap\_word4 : std\_logic\_vector(31 downto 0) := x"04000000"; -- warm boot start add

signal icap\_word5 : std\_logic\_vector(31 downto 0) := x"30008001"; -- type1 write1 word to cmd

signal icap\_word6 : std\_logic\_vector(31 downto 0) := x"0000000F"; -- iprog cmd

signal icap\_word7 : std\_logic\_vector(31 downto 0) := x"20000000"; -- type1 no op

signal icap\_word8 : std\_logic\_vector(31 downto 0) := x"AAAABBBB"; -- dummy

signal icap\_write\_en : std\_logic\_vector(8 downto 0) := "100000001"; --active low write enable

begin

ICAP\_VIRTEX5\_inst : ICAP\_VIRTEX5

generic map (ICAP\_WIDTH => "X32") -- "X8" or "X32

port map (

BUSY => open, -- Busy output

O => open, -- 32-bit data output

CE => icap\_ce, -- Clock enable input

CLK => icap\_clk, -- Clock input

I => icap\_i, -- 32-bit data input

WRITE => icap\_write -- Write input

);

icap\_control : process(clk)

begin

if clk'event and clk = '1' then

if icap\_next\_en = '1' then

icap\_word0 <= icap\_word1;

icap\_word1 <= icap\_word2;

icap\_word2 <= icap\_word3;

icap\_word3 <= icap\_word4;

icap\_word4 <= icap\_word5;

icap\_word5 <= icap\_word6;

icap\_word6 <= icap\_word7;

icap\_word7 <= icap\_word8;

icap\_word8 <= icap\_word0;

icap\_write\_en <= icap\_write\_en(0) & icap\_write\_en(8 downto 1);

end if;

end if;

end process icap\_control;

end Behavioral ;

다음 리스트는 이 로직을 합성할 때 필요한 로직 사용량을 보여 줍니다. 플립플롭의 사용량과 슬라이스의 사용량만 살펴 보면 다음과 같습니다.

* Number of Slice Registers: 17 out of 44,800
* Number of occupied Slices: 7 out of 11,200

Design Summary

--------------

Number of errors: 0

Number of warnings: 1

Slice Logic Utilization:

Number of Slice Registers: 17 out of 44,800 1%

Number used as Flip Flops: 17

Number of Slice LUTs: 15 out of 44,800 1%

Number used as logic: 1 out of 44,800 1%

Number using O6 output only: 1

Number used as Memory: 14 out of 13,120 1%

Number used as Shift Register: 14

Number using O6 output only: 14

Slice Logic Distribution:

Number of occupied Slices: 7 out of 11,200 1%

Number of LUT Flip Flop pairs used: 17

Number with an unused Flip Flop: 0 out of 17 0%

Number with an unused LUT: 2 out of 17 11%

Number of fully used LUT-FF pairs: 15 out of 17 88%

Number of unique control sets: 2

Number of slice register sites lost

to control set restrictions: 1 out of 44,800 1%

A LUT Flip Flop pair for this architecture represents one LUT paired with

one Flip Flop within a slice. A control set is a unique combination of

clock, reset, set, and enable signals for a registered element.

The Slice Logic Distribution report is not meaningful if the design is

over-mapped for a non-slice resource or if Placement fails.

IO Utilization:

Number of bonded IOBs: 2 out of 640 1%

Specific Feature Utilization:

Number of BUFG/BUFGCTRLs: 1 out of 32 3%

Number used as BUFGs: 1

Number of ICAPs: 1 out of 2 50%

Peak Memory Usage: 369 MB

Total REAL time to MAP completion: 16 secs

Total CPU time to MAP completion: 13 secs

### 쉬프트 레지스터로 구현되지 않는 경우

[..\..\..\..\design\_edk\board\_ml507\multi\_boot\_sec\test\_mult\_boot\_v5\test\_mult\_boot\_v5.ise](file:///C:\design_edk\board_ml507\multi_boot_sec\test_mult_boot_v5\test_mult_boot_v5.ise) 를 선택하시면

다음 리스트는 앞에서 설명한 코드와 다르게 다음과 같이 icap\_word4를 초기화 하는 다른 코드가 삽입되었습니다.

if((trigger = '1' and icap\_clk = '0' and icap\_prepare = '0') or icap\_next\_en = '1') then

icap\_word4 <= boot\_addr;

icap\_prepare <= '1';

이것은 일반적으로 SIGNAL을 초기화하는 리셋과 같은 역할을 하기 때문에 쉬프트 연산에 필요한 레지스터를 LUT로 대치할 수 없고 슬라이스에 있는 플립플롭을 사용할 수 밖에 없습니다.

다음 리스트는 쉬프트 레지스터를 LUT로 구현하지 못하는 코드 입니다.

library IEEE;

use IEEE.Std\_Logic\_1164.all;

use IEEE.numeric\_std.all;

library unisim;

use unisim.vcomponents.all;

entity multi\_boot\_v5 is

port (

trigger: in std\_logic;

boot\_addr : in std\_logic\_vector(31 downto 0);

clk : in std\_logic);

end multi\_boot\_v5;

architecture Behavioral of multi\_boot\_v5 is

component ICAP\_VIRTEX5

generic ( ICAP\_WIDTH : string := "X32");

port (

BUSY : out std\_ulogic; -- Busy output

O : out std\_logic\_vector (31 downto 0); -- 32-bit data output

CE : in std\_ulogic; -- Clock enable input

CLK : in std\_ulogic; -- Clock input

I : in std\_logic\_vector (31 downto 0); -- 32-bit data input

WRITE : in std\_ulogic -- Write input

);

end component;

--ICAP signals

signal icap\_ce : std\_logic := '0';

signal icap\_write : std\_logic := '0';

signal icap\_clk : std\_logic := '0';

signal icap\_prepare : std\_logic := '0';

signal icap\_next\_en : std\_logic := '0';

signal icap\_i : std\_logic\_vector(31 downto 0);

signal icap\_word0 : std\_logic\_vector(31 downto 0) := x"00000000";

signal icap\_word1 : std\_logic\_vector(31 downto 0) := x"AA995566"; -- sync

signal icap\_word2 : std\_logic\_vector(31 downto 0) := x"20000000"; -- type1 no op

signal icap\_word3 : std\_logic\_vector(31 downto 0) := x"30020001"; -- type1 write1 wbstart

signal icap\_word4 : std\_logic\_vector(31 downto 0) := x"04000000"; -- warm boot start add

signal icap\_word5 : std\_logic\_vector(31 downto 0) := x"30008001"; -- type1 write1 word to cmd

signal icap\_word6 : std\_logic\_vector(31 downto 0) := x"0000000F"; -- iprog cmd

signal icap\_word7 : std\_logic\_vector(31 downto 0) := x"20000000"; -- type1 no op

signal icap\_word8 : std\_logic\_vector(31 downto 0) := x"AAAABBBB"; -- dummy

signal icap\_write\_en : std\_logic\_vector(8 downto 0) := "100000001"; --active low write enable

begin

ICAP\_VIRTEX5\_inst : ICAP\_VIRTEX5

generic map (ICAP\_WIDTH => "X32") -- "X8" or "X32

port map (

BUSY => open, -- Busy output

O => open, -- 32-bit data output

CE => icap\_ce, -- Clock enable input

CLK => icap\_clk, -- Clock input

I => icap\_i, -- 32-bit data input

WRITE => icap\_write -- Write input

);

icap\_control : process(clk)

begin

if clk'event and clk = '1' then

if((trigger = '1' and icap\_clk = '0' and icap\_prepare = '0') or icap\_next\_en = '1') then

icap\_word4 <= boot\_addr;

icap\_prepare <= '1';

else

icap\_prepare <= '0';

end if;

if icap\_clk = '1' then

icap\_next\_en <= '1';

else

icap\_next\_en <= '0';

end if;

if icap\_next\_en = '1' then

icap\_word0 <= icap\_word1;

icap\_word1 <= icap\_word2;

icap\_word2 <= icap\_word3;

icap\_word3 <= icap\_word4;

icap\_word4 <= icap\_word5;

icap\_word5 <= icap\_word6;

icap\_word6 <= icap\_word7;

icap\_word7 <= icap\_word8;

icap\_word8 <= icap\_word0;

icap\_write\_en <= icap\_write\_en(0) & icap\_write\_en(8 downto 1);

end if;

end if;

end process icap\_control;

end Behavioral ;

다음 리스트는 같은 기능을 하는 로직을 구현하는데 필요한 로직 사용량을 보여 줍니다. 플립플롭의 사용량과 슬라이스의 사용량만 살펴 보면 다음과 같습니다.

* Number of Slice Registers: 100 out of 44,800
* Number of occupied Slices: 37 out of 11,200

Design Summary

--------------

Number of errors: 0

Number of warnings: 1

Slice Logic Utilization:

Number of Slice Registers: 100 out of 44,800 1%

Number used as Flip Flops: 100

Number of Slice LUTs: 98 out of 44,800 1%

Number used as logic: 33 out of 44,800 1%

Number using O6 output only: 33

Number used as Memory: 65 out of 13,120 1%

Number used as Shift Register: 65

Number using O6 output only: 65

Slice Logic Distribution:

Number of occupied Slices: 37 out of 11,200 1%

Number of LUT Flip Flop pairs used: 100

Number with an unused Flip Flop: 0 out of 100 0%

Number with an unused LUT: 2 out of 100 2%

Number of fully used LUT-FF pairs: 98 out of 100 98%

Number of unique control sets: 2

Number of slice register sites lost

to control set restrictions: 3 out of 44,800 1%

A LUT Flip Flop pair for this architecture represents one LUT paired with

one Flip Flop within a slice. A control set is a unique combination of

clock, reset, set, and enable signals for a registered element.

The Slice Logic Distribution report is not meaningful if the design is

over-mapped for a non-slice resource or if Placement fails.

IO Utilization:

Number of bonded IOBs: 34 out of 640 5%

Specific Feature Utilization:

Number of BUFG/BUFGCTRLs: 1 out of 32 3%

Number used as BUFGs: 1

Number of ICAPs: 1 out of 2 50%

Peak Memory Usage: 375 MB

Total REAL time to MAP completion: 18 secs

Total CPU time to MAP completion: 18 secs

로직 사용량의 차이를 보면 실제 쉬프트 레지스터를 구현하기 위해서 리셋이 얼마나 많은 영향을 미치는지 확인 했습니다

## FALLBACK현상이 발생했을 때

Virtex5는 플래시 메모리를 이용해 여러가지 configuration 데이터를 저장할 수 있고 초기에 사용하기 원하는 configuration 데이터를 RS[1:0]을 이용해 선택할 수 있습니다.

이렇게 되면 개발자는 FPGA로 만든 시스템에 적다한 통신채널, 예를 들어 이더넷, 을 확보할 수 있다면 FPGA의 configuration 데이터를 쉽게 수정할 수 있게 됩니다.

이런 개념이 처음 나온 것은 아니지만 이런 기능을 수행하기 위해서는 위부에 configuration과정을 감시하는 CPLD나 조그만한 프로세서가 필요 했습니다

만일 FPGA configuration 데이터를 수정하는 도중에 전원이상과 같이 어떤 원인에 의해서 수정하려던 작업이나 데이터에 이상이 생겨 플래시 메모리를 완전히 갱신하는데 실패했다면 FPGA가 configuration을 할 수 없게 됩니다.

그래서 FPGA를 원격지에서 수정할 수 있는 기능을 사용하려면 이러한 면도 고려해야 합니다.

Fallback은 FPGA가 configuration에 실패했을 경우에 미리 지정된 configuration 데이터를 가지고 다시 configuration을 시도하는 것을 의미 합니다.

### Fallback 이 발생했을 때 FPGA의 configuration 절차

#### 처음 전원이 켜진 후 FPGA는 FPGA내부에 있는 모든 로직을 깨끗이 지우는 작업을 먼저 시도 합니다 (INIT).

#### 그후 FPGA의 mode[2:0]을 검사해서 BPI 모드를 사용했지는 검사 합니다.

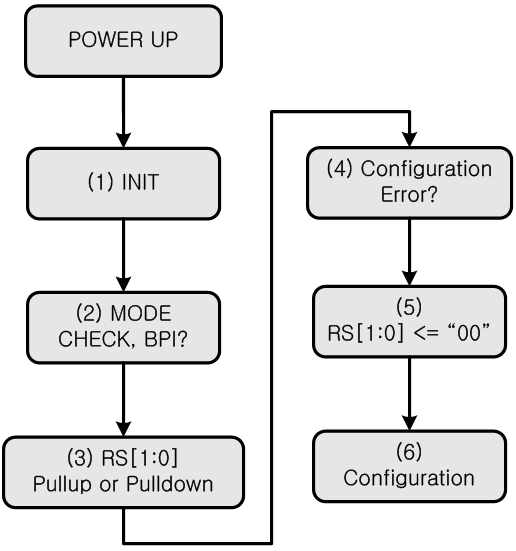
#### BPI 모드인 경우에는 RS[1:0]에 연결된 PULL-UP, PULL-DOWN 에 의해서 정해진 플래시 메모리 어드레스를 가지고 configuration을 시도 합니다. PULL-UP, PULL-DOWN 저항이 연결되지 않았을 경우에는 RS[1:0] = “11”을 가지고 있습니다.

#### Configuration이 실패했을 경우 FPGA는 내부에서 configuration을 다시 시도 합니다.

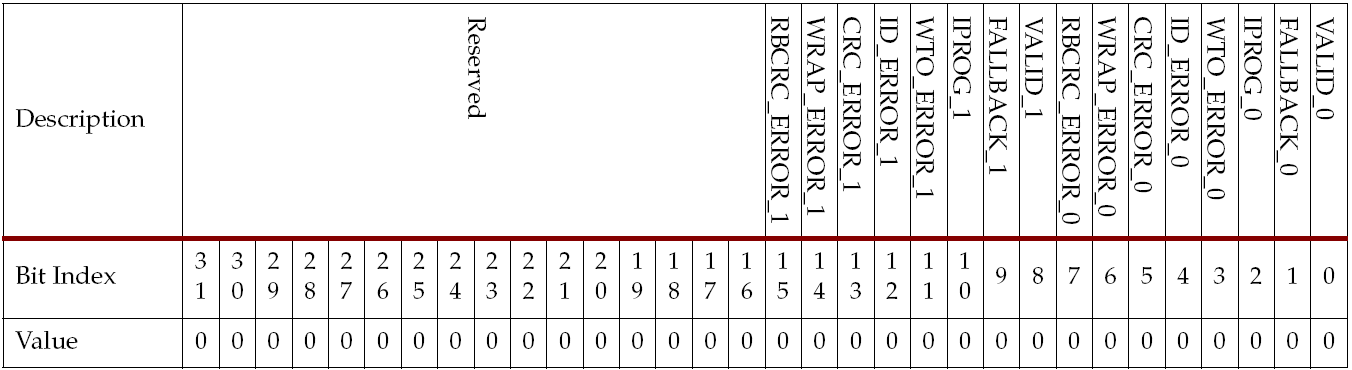
#### 다시 시도하기 전에 먼저 RS[1:0] 값을 강제로 “00”으로 출력 합니다.

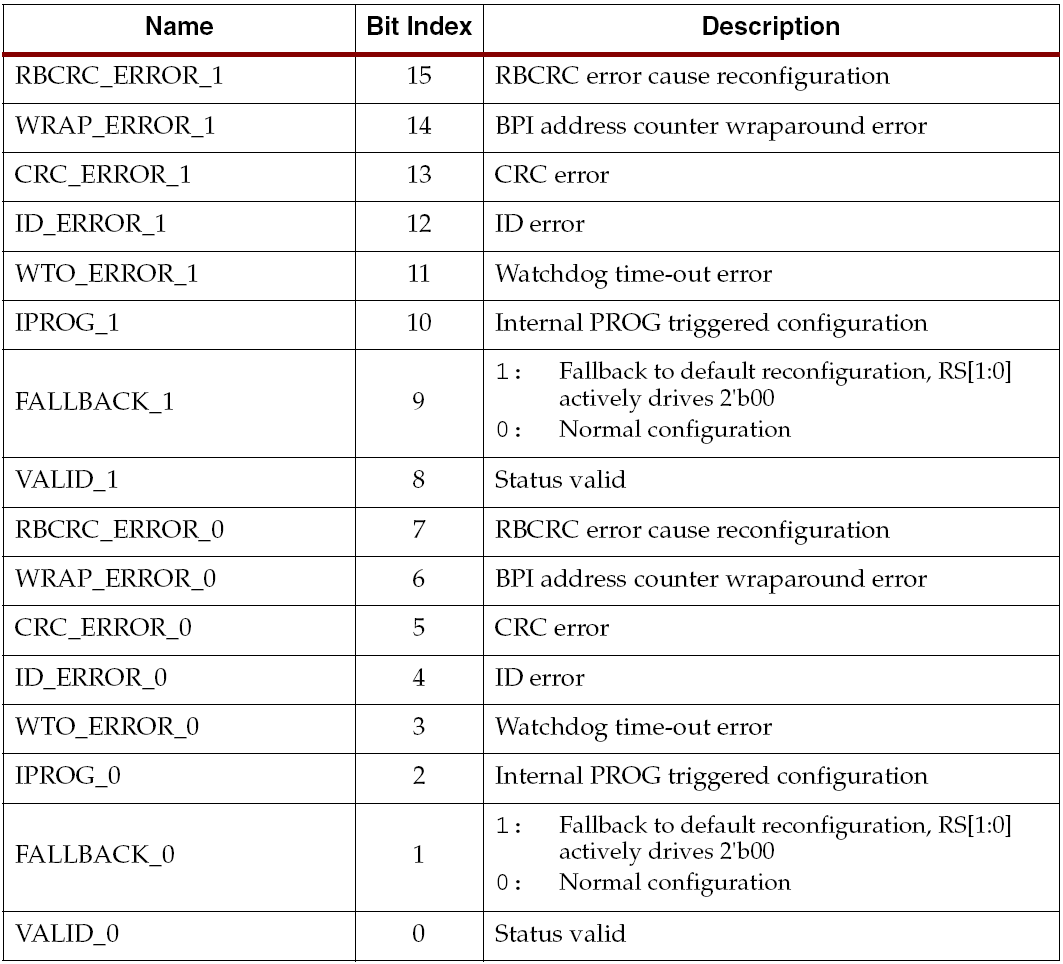
#### configuration을 다시 시도 합니다.

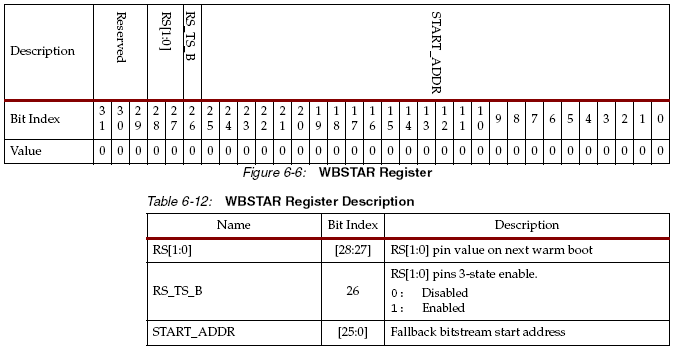
다음 그림은 fallback이 발생했을 경우 FPGA 내부에서 일어나는 작업 순서를 보여 주고 있습니다.



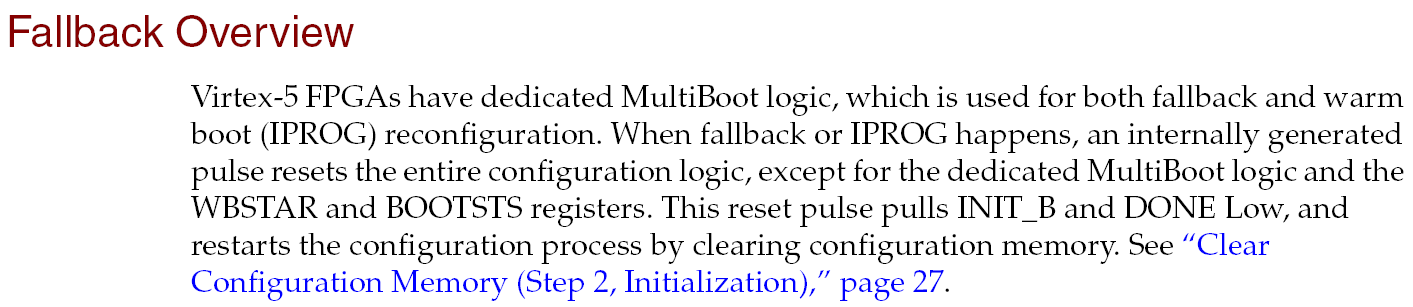
### BOOT STATUS REGISTER







## FALLBACK & WARMBOOT



Virtex-5 FPGAs have dedicated MultiBoot logic, which is used for both fallback and warm boot (IPROG) reconfiguration. When fallback or IPROG happens, an internally generated pulse resets the entire configuration logic, except for the dedicated MultiBoot logic and the WBSTAR and BOOTSTS registers.

virtex5에 있는 MultiBoot logic은 fallback과 warm boot 용으로 사용 됩니다. 만약 fallback 현상이 발생하거나 IPROG를 트리거 시킨 경우 FPGA는 MultiBoot logic과 WBSTART (Warm boot Start) 레지스터와 BOOTSTS 레지스터를 제외한 모든 로직은 configuration 절차에 따라 다시 초기화가 이루어 집니다.

During fallback reconfiguration, the FPGA drives new values on the two dual-mode pins RS[1:0] (Revision Select). RS[1:0] are 3-stated and weakly pulled up during the first configuration, and weakly pulled down after configuration by default.

기본적으로 virtex5 FPGA는 RS[1:0] 핀이 존재하면 처음 전원이 ON이 되었을 때 이 2개의 핀은 weak pull up 상태를 유지하고 있다가 configuration이 끝나면 weak pull down 상태를 가지게 됩니다.

When a configuration error is detected, the configuration logic generates an internal reset pulse and actively drives RS[1:0] to 00 for loading the fallback bitstream.

만약 configuration 도중 문제를 발견하면 FPGA는 RS[1:0]을 00으로 만들어서 다시 configuration을 시도합니다.

***Caution!***

The iMPACT indirect programming solution drives all FPGA address lines (A[25:0]) during ISP operations on the BPI PROM. The FPGA address lines must be connected directly to the BPI PROM address lines. If the upper BPI PROM address signals are tied to the FPGA RS[1:0] pins for a Fallback or Multiboot implementation, the indirect programming solution cannot erase or program the BPI PROM address space accessed by the upper two address signals. It is necessary to jumper the FPGA RS[1:0] pins with the FPGA upper address signals to combine the two setups.

만약 RS[1:0]을 사용하는 경우에는 impact을 이용해서 플래시 메모리를 직접 WRITE할 수 없습니다.

#### map file

Design Summary

--------------

Number of errors: 0

Number of warnings: 1

Slice Logic Utilization:

Number of Slice Registers: 17 out of 44,800 1%

Number used as Flip Flops: 17

Number of Slice LUTs: 15 out of 44,800 1%

Number used as logic: 1 out of 44,800 1%

Number using O6 output only: 1

Number used as Memory: 14 out of 13,120 1%

Number used as Shift Register: 14

Number using O6 output only: 14

Slice Logic Distribution:

Number of occupied Slices: 7 out of 11,200 1%

Number of LUT Flip Flop pairs used: 17

Number with an unused Flip Flop: 0 out of 17 0%

Number with an unused LUT: 2 out of 17 11%

Number of fully used LUT-FF pairs: 15 out of 17 88%

Number of unique control sets: 2

Number of slice register sites lost

to control set restrictions: 1 out of 44,800 1%

A LUT Flip Flop pair for this architecture represents one LUT paired with

one Flip Flop within a slice. A control set is a unique combination of

clock, reset, set, and enable signals for a registered element.

The Slice Logic Distribution report is not meaningful if the design is

over-mapped for a non-slice resource or if Placement fails.

IO Utilization:

Number of bonded IOBs: 2 out of 640 1%

Specific Feature Utilization:

Number of BUFG/BUFGCTRLs: 1 out of 32 3%

Number used as BUFGs: 1

Number of ICAPs: 1 out of 2 50%

Peak Memory Usage: 369 MB

Total REAL time to MAP completion: 16 secs

Total CPU time to MAP completion: 13 secs

#### shift register로 구현됨.

library IEEE;

use IEEE.Std\_Logic\_1164.all;

use IEEE.numeric\_std.all;

library unisim;

use unisim.vcomponents.all;

entity multi\_boot\_v5 is

port (

trigger: in std\_logic;

clk : in std\_logic);

end multi\_boot\_v5;

architecture Behavioral of multi\_boot\_v5 is

component ICAP\_VIRTEX5

generic ( ICAP\_WIDTH : string := "X32");

port (

BUSY : out std\_ulogic; -- Busy output

O : out std\_logic\_vector (31 downto 0); -- 32-bit data output

CE : in std\_ulogic; -- Clock enable input

CLK : in std\_ulogic; -- Clock input

I : in std\_logic\_vector (31 downto 0); -- 32-bit data input

WRITE : in std\_ulogic -- Write input

);

end component;

--ICAP signals

signal icap\_ce : std\_logic := '0';

signal icap\_write : std\_logic := '0';

signal icap\_clk : std\_logic := '0';

signal icap\_prepare : std\_logic := '0';

signal icap\_next\_en : std\_logic := '0';

signal icap\_i : std\_logic\_vector(31 downto 0);

signal icap\_word0 : std\_logic\_vector(31 downto 0) := x"00000000";

signal icap\_word1 : std\_logic\_vector(31 downto 0) := x"AA995566"; -- sync

signal icap\_word2 : std\_logic\_vector(31 downto 0) := x"20000000"; -- type1 no op

signal icap\_word3 : std\_logic\_vector(31 downto 0) := x"30020001"; -- type1 write1 wbstart

signal icap\_word4 : std\_logic\_vector(31 downto 0) := x"04000000";

signal icap\_word5 : std\_logic\_vector(31 downto 0) := x"30008001"; -- type1 write1 word to cmd

signal icap\_word6 : std\_logic\_vector(31 downto 0) := x"0000000F"; -- iprog cmd

signal icap\_word7 : std\_logic\_vector(31 downto 0) := x"20000000"; -- type1 no op

signal icap\_word8 : std\_logic\_vector(31 downto 0) := x"AAAABBBB"; -- dummy

signal icap\_write\_en : std\_logic\_vector(8 downto 0) := "100000001"; --active low write enable

begin

ICAP\_VIRTEX5\_inst : ICAP\_VIRTEX5

generic map (ICAP\_WIDTH => "X32") -- "X8" or "X32

port map (

BUSY => open, -- Busy output

O => open, -- 32-bit data output

CE => icap\_ce, -- Clock enable input

CLK => icap\_clk, -- Clock input

I => icap\_i, -- 32-bit data input

WRITE => icap\_write -- Write input

);

icap\_control : process(clk)

begin

if clk'event and clk = '1' then

if((trigger = '1' and icap\_clk = '0' and icap\_prepare = '0') or icap\_next\_en = '1') then

icap\_prepare <= '1';

else

icap\_prepare <= '0';

end if;

if icap\_prepare = '1' then

icap\_clk <= '1';

else

icap\_clk <= '0';

end if;

if icap\_clk = '1' then

icap\_next\_en <= '1';

else

icap\_next\_en <= '0';

end if;

if icap\_next\_en = '1' then

icap\_word0 <= icap\_word1;

icap\_word1 <= icap\_word2;

icap\_word2 <= icap\_word3;

icap\_word3 <= icap\_word4;

icap\_word4 <= icap\_word5;

icap\_word5 <= icap\_word6;

icap\_word6 <= icap\_word7;

icap\_word7 <= icap\_word8;

icap\_word8 <= icap\_word0;

icap\_write\_en <= icap\_write\_en(0) & icap\_write\_en(8 downto 1);

end if;

end if;

end process icap\_control;

icap\_write <= icap\_write\_en(0);

icap\_ce <= icap\_write\_en(0);

icap\_rev0 : for i in 0 to 7 generate

icap\_i(i) <= icap\_word0(7 - i);

end generate icap\_rev0;

icap\_rev1 : for i in 8 to 15 generate

icap\_i(i) <= icap\_word0(15 - (i - 8));

end generate icap\_rev1;

icap\_rev2 : for i in 16 to 23 generate

icap\_i(i) <= icap\_word0(23 - (i - 16));

end generate icap\_rev2;

icap\_rev3 : for i in 24 to 31 generate

icap\_i(i) <= icap\_word0(31 - (i - 24));

end generate icap\_rev3;

end Behavioral ;

#### map file

