마이크로프로세서 설계 무작정 따라하기 part-III (4)

부제: 합성, 배치 및 배선

KAIST 전자전산학과 박사과정 배영돈(http://www.donny.co.kr)

지난 2 회의 강좌를 통해 Synopsys Design Compiler 를 이용하여 회로를 합성하는 방법을 설명하였다. 이번 강좌에서는 지난강좌에서 배운 명령어들을 사용하여 SimpleCore 를 직접 합성해보고 이 과정에서 복잡한 회로를 효율적으로 합성하는 방법을 알아본다.

1. SimpleCore 의 소스 파일의 구성

SimpleCore 의 소스코드(http://www.donny.co.kr/simplecore)는 표 1 과 같이 20 개의 파일로 구성되어 있다. 이중에 debug.v, mem.v, top.v 는 simulation 을 위한 것으로 실제 회로로 구현되지 않으며 합성이 불가능하다. simplecore.v 는 simplecore 의 top module 이다. def.v 는 소스코드에서 사용되는 매크로(INST_ALUI, COND_EQ, 등)를 정의한 것으로 def.v 가 독자적으로는 합성되지 않으며 다른 파일에서 include 되어 합성된다.

표 1. SimpleCore 의 소스 파일의 구성

파일이름	설명
alu.v	ALU
busCtl.v	bus controller
control.v	control
dAreg.v	data address register
dOutreg.v	data output register
datapath.v	datapath
debug.v	simulation 을 위한 debug information module
decode.v	decode unit
def.v	definitions (macros)
execute.v	execution unit
fetch.v	fetch unit
iAreg.v	instruction address register
incr.v	address incrementor
mem.v	simulation 을 위한 memory model
mul.v	multiplier
regFile.v	register file
shifter.v	shifter
simplecore.v	SimpleCore 의 top-level module
sr.v	status register
top.v	simulation 을 위한 top-level module

2. 환경 설정

먼저 Design Compiler 를 사용하기 위한 환경을 설정해야 한다. 환경 설정 과정 없이 dc_shell(또는 design_analyzer)을 실행하면 $SYNOPSYS/admin/setup/synopsys_dc.setup 에 설정된 내용 사용된다. 환경 설정을 하기 위해서는 dc_shell 이나 design_analyzer 가 실행되는

디렉토리에 .synopsys_dc.setup 이란 파일을 만들고 필요한 내용을 추가하면 된다. 다음 sample 에서 굵게 표시된 변수들은 기본적으로 필요한 것이다.

synopsys_dc.setup 파일의 내용은 라이브러리에 따라 차이가 있으므로 칩을 제작하는 공정의 design kit 의 매뉴얼을 참고하여 작성한다.

Sample .synopsys_dc.setup:

```
/*****************************************************/
/* HYUNDAI 0.35 Standard Cell Library                */
/*                                                   */
/* Copyright(c) Integrated Computer Systems Lab., KAIST */
/*****************************************************/

search_path = { . "/tools/synopsys/2000.05/libraries/syn" \
    /home/donny/library/HSC350/1.0.0/logic/synopsys/cb35io122d_000310 \
    /home/donny/library/HSC350/1.0.0/logic/synopsys/cb35io132d_000310 \
    /home/donny/library/HSC350/1.0.0/logic/synopsys/cb35os142d_000310 \
    };
symbol_library = {cb35io122d.sdb cb35io132d.sdb cb35os142d.sdb};
generic_symbol_library = "generic.sdb" ;

/* TYPICAL CASE */
target_library = {cb35io122d_typ.db cb35io132d_typ.db cb35os142d_typ.db};
link_library = {cb35io122d_typ.db cb35io132d_typ.db cb35os142d_typ.db};

designer = "Donny" ;
company = "ICSL";

/*****************************************************************************
**              Naming Rules               **
*****************************************************************************/
define_name_rules hei_rules -max_length "31" \
    -allowed "a-zA-Z0-9_" -type port -first_restricted "$*&_0-9" \
    -last_restricted "_" -map {{"__" , "_"}}
define_name_rules hei_rules -max_length "31" \
    -allowed "a-zA-Z0-9_" -type net -first_restricted "$*&_0-9" \
    -last_restricted "_" -map {{"__" , "_"}}
define_name_rules hei_rules -max_length "31" \
    -allowed "a-zA-Z0-9_" -type cell -first_restricted "$*&_0-9" \
```

```
    -last_restricted "_" -map {{"__" , "_"}}

default_name_rules = hei_rules

alias change_names "change_names -rules hei_rules -hierarchy -verbose"

verilogout_show_unconnected_pins = "TRUE" /* for use of Apollo */

hdlin_enable_vpp = TRUE; /* `ifdef `endif */
```

3. Constraint 의 설정

constraint 를 설정하는 방법은 크게 세 가지로 구별할 수 있다. 첫번째는 design_analyzer 에서 풀다운 메뉴 등을 이용하는 방법과 두번째는 dc_shell(또는 design_analyer 의 command window)에서 명령어를 이용하는 방법, 마지막 방법인 Verilog 파일에 기술하는 방법을 알아보도록 하자.
먼저, 지난 시간에 합성했던 alu 의 Verilog 기술에 constraint 를 추가해보자.

```
// synopsys dc_script_begin
// set_max_delay 0 -from all_inputs() -to all_outputs()
// set_max_transition 0.5 current_design
// set_load 0.1 all_outputs()
// synopsys dc_script_end
```

이와 같이 주석문(comment)를 모듈기술부분(module();과 endmodule 사이)에 추가해주면, Design Compiler 는 constraint 로 인식한다. 즉, // synopsys dc_script_begin 과 // synopsys dc_script_end 사이의 내용은 Verilog 파일을 읽은 뒤 dc_shell 에서 입력한 것과 동일한 효과를 갖는다. constraint 에 사용된 all_inputs(), all_outputs()와 current_design 은 보다 constraint 를 쉽게 기술하기 위한 기능들로 여러 개의 입출력에 동일한 constraint 를 한번에 적용할 수 있다. 이와 유사하게 find()기능을 사용하면 port 나 net 등의 이름으로 검색하여 조건에 맞는 대상에만 constraint 를 적용할 수도 있다. 자세한 사용법은 Design Compiler 의 reference manual 이나 help(예, help find)기능을 사용하며 알 수 있다.

constraint 에 사용된 값들은 단위가 명시되지않았는데, 일반적으로 시간은 1ns, 캐패시턴스는 pF, 전압은 1V, 전류는 1mA 그리고 전력은 1nW 를 단위로 한다. 이것은 library 에 정의된 내용으로 report_lib 명령을 이용하여 확인할 수 있다. 라이브러리에 따라 다른 단위를 사용할 수 있으므로, 사용 전에 반드시 확인해야 할 내용이다.

이와 같이 Verilog 코드에 constraint 를 함께 기술하는 것은 중요한 constraint 를 소스코드에 포함시켜 추후에 합성할 때도 쉽게 고려할 수 있다는 것과 소스코드의 정리, 리뷰 등을 수행할 때 장점이 있다.

report_lib 의 결과:

Library Type : Technology
Tool Created : 1997.08
Date Created : 17 March 1998
Library Version : 7.1/3.3b
Time Unit : 1ns

Capacitive Load Unit : 1.000000pf
Pulling Resistance Unit : 1kilo-ohm
Voltage Unit : 1V
Current Unit : 1mA
Power Unit : 1nW
Leakage Power Unit : 1nW
Bus Naming Style : %s[%d] (default)

4. Synthesis 용 script 파일의 작성

각 module 의 constraint 는 Verilog 파일에 포함되어있으므로 synthesis 용 script 는 다음과 같이 간단히 작성할 수 있다.

filename: alu.scr

```
read -format verilog /home/donny/simplecore/rtl/alu.v
compile -map_effort high
ungroup -flatten -all
write -format db /home/donny/simplecore/gate/alu.db
report_area > alu.report
report_timing >> alu.report
quit
```

synthesis script 를 이용하여 합성하는 방법은 두 가지가 있다

1. dc_shell 에서 script 를 불러들이는 방법

```
dc_shell> include alu.scr
```

2. dc_shell 의 실행과 함께 script 를 실행

```
$ dc_shell -f alu.scr
```

5. SimpleCore 의 합성

이제 본격적으로 SimpleCore 를 합성해보도록 하자. 앞서 설명한 것과 같이 모든 module 에는 적절한 constraint 가 기술되어 있으며 각각을 합성하기 위한 synthesis script 를 작성한 상태라고 가정한다. 이제 각각의 module 들을 하나씩 합성해나가면 전체 설계의 합성이 끝날 것으로 생각되지만 몇 가지 더 알아야 할 내용이 있다.

먼저 소스코드에는 필요에 따라 합성이 불가능한 부분이 있을 수 있는데, 합성 시에는 이런 부분을 제거해 주어야 한다. 예를 들어, SimpleCore 에는 simulation 을 할 때 프로세서의 상태를 쉽게 알아보기 위한 디버깅정보가 포함되어있다. 이러한 정보들을 매번 합성을 할 때 찾아서 삭제(또는 주석처리)하는 것은 번거로운 작업이다. 이때, synopsys translate_off 와 synopsys translate_on 기능을 사용하면 편리하다.

```
always@(posedge clk)
begin
    a = b;
// synopsys translate_off
    $write("%d", a);
// synopsys translate_on
end
```

그림 1 은 SimpleCore 의 계층구조를 단순화하여 보여주고 있다. 매우 간단한 회로를 제외하고는 대부분의 회로들은 이러한 계층구조를 갖고 있다. 계층적인 회로를 합성하는 방법은 top-down 방식과 bottom-up 방식 두 가지로 구분할 수 있다. 먼저 top-down 은 쉽게 설명하면 simplecore 전체를 한번에 합성하는 방법이다. 합성 툴이 자동으로 simplecore 의 계층을 이동하며 모든 모듈들을 합성한다. 이 방법은 전체 회로의 크기가 작을 경우에는 편리하고 좋은 결과를 얻을 수 있는 방법이지만 복잡회로의 경우에는 합성시간이 매우 오래 걸리고 좋은 결과를 얻기도 힘들다. bottom-up 방식은 낮은 계층의 모듈부터 합성하는 것으로 작은 회로단위로 처리하기 때문에 속도가 빠르고, 각각의 모듈에 대하여 충분한 최적화를 할 수 있다. 그러나, 전체 회로의 관점에서 최적화를 하기 어렵다. 즉, 어떤 모듈이 임계경로(critical path) 상에 있는지, 외부의 큰 부하(load)가 있는지 회로 전체를 보지않고는 알 수 없기 때문이다.

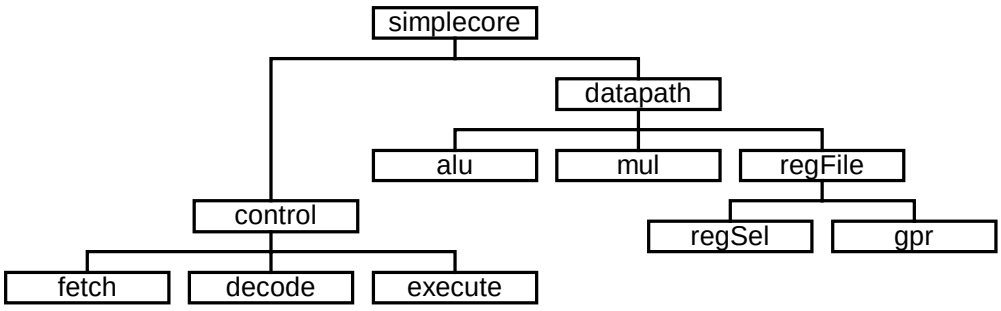

그림 1. SimpleCore 의 계층구조

따라서, 효율적인 합성방법은 설계자의 통찰력을 바탕으로 합성 툴의 기능을 활용하는 것이다. 즉, 합성 툴은 비교적 단순한 문제(주어진 조건에서의 최적화)는 효율적을 수행하지만 이는 반복적인 계산결과 중 가장 좋은 결과를 선택하는 것으로 복잡한 회로에서는 설계자의 도움이 없이는 효율적인 결과를 얻기 힘들다.

이제 bottom-up 방식으로 합성을 하는 방법을 알아보도록 하자. 먼저 계층의 가장 하위에서부터 차례대로 합성을 해야 한다. 예를 들어 control 모듈의 하위 모듈인 fetch, decode, execute 모듈를 합성한 다음 control 모듈을 합성하면 fetch, decode, execute 모듈은 이미 합성된 결과를 사용하여야 한다. 하지만, Design Compiler 에서는 이미 합성된 결과가 있더라도 control 모듈에 포함된 모든 회로를 처음부터 새로 합성한다. 즉, bottom-up 방식으로 합성하기 위해서는 다음과 같이 이미 합성된 모듈을 사용하기위한 constraint 가 필요하다. (synthesis script 나 Verilog code 에 추가하면 된다.)

set_dont-touch find(cell, Ifetch)
set_dont-touch find(cell, Idecode)
set_dont-touch find(cell, Iexecute)

앞서 설명한 것과 같이 bottom-up 방식으로 합성하기 위해서는 각 모듈의 외부에 대한 정보를 constraint 를 이용해 설정하는 것이 중요하다. 해당 모듈의 출력이 큰 부하를 구동하는 경우에는 'set_load'를 이용하여 출력포트에 load 값을 정해주고, 특정 입력이 늦게 입력되는 경우에는 'set_input_delay'를 이용한다. 또한, 모듈 자체가 임계경로에 위치하면 해당 경로의 지연속도를 최소로 할 수 있도록 해야 한다.

이제 전체회로를 bottom-up 방식으로 합성하였다. 만일 gpr 모듈의 설계를 변경하였다고 생각해보자 top-down 방식이었다면 모든 회로를 처음부터 합성해야 할 것이다. bottom-up 방식에서는 그림 2 와 같이 변경된 부분만 새로 합성하면 되기 때문에 설계시간을 단축시킬 수 있다.

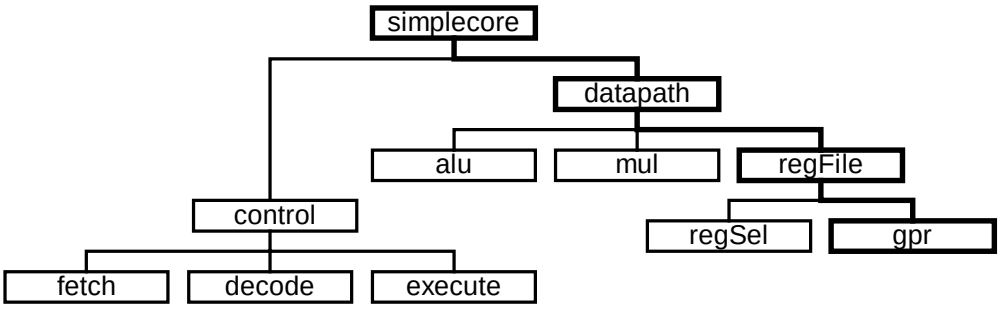

그림 2. 설계 변경

그러나, 설계를 수정할 때마다 이러한 계층구조를 생각해서 다시 합성을 하는 것에도 번거로운 일이 발생하는데, 관련이 없다고 생각하고 합성을 하지 않는 경우가 발생할 수 있고, 한번에 여러 모듈을 고치면 복잡해질 수도 있다. 이를 쉽게 해결할 수 있는 방법이 바로 make 명령을 쓰는 방법이다. make 는 주로 C 프로그램을 compile 할 때 변경된 C 코드와 의존관계가 있는 코드만 선택적으로 컴파일하기위해 사용한다. 이 make 기능을 회로합성에도 사용할 수 있다. 다음은 SimpleCore 를 합성하기 위한 Makefile 이다.

filename: Makefile
.SUFFIXES: .v .db

```
simplecore.db     : simplecore.v simplecore.scr control.db datapath.db

.v.db :
        dc_shell -f $*.scr

fetch.db        : fetch.v fetch.scr
decode.db       : decode.v decode.scr
execute.db      : execute.v execute.scr
control.db      : control.v control.scr fetch.db decode.db execute.db
alu.db          : alu.v alu.scr
iAreg.db        : iAreg.v iAreg.scr
dAreg.db        : dAreg.v dAreg.scr
dOutreg.db      : dOutreg.v dOutreg.scr
incr.db         : incr.v incr.scr
mul.db          : mul.v mul.scr
regFile.db      : regFile.v regFile.scr
shifter.db      : shifter.v shifter.scr
sr.db           : sr.v sr.scr
datapath.db     : datapath.v datapath.scr alu.db iAreg.db dAreg.db dOutreg.db incr

clean :
        rm *.db \
        rm *.report
```

지금까지 복잡한 회로를 효율적으로 합성하는 방법을 SimpleCore 를 통해 알아보았다. 다음 회에서는 마지막 과정인 배치 및 배선에 대해서 설명한다.

본 강좌에 사용된 예제는 홈페이지(http://www.donny.co.kr/simplecore)를 통해 제공됩니다.