Design Compiler for Synthesis
(2 : Basic Synthesis)

2014. 02. 17
KYUNG-HEE UNIV.
CSA & VLSI
CHOI MIN SU

1. PAD-cell Synthesis

√ 본격적인 합성을 시작하기 전에 해야 할 작업이 한 가지 있다. 디자인에 입출력 PAD-cell을 붙이는 과정이다. PAD-cell의 삽입은 필수 사항이며 이를 위해 기존의 디자인을 수정해야 한다.

√ 또 생각해보자. Signal의 입출력은 넓적한 금속 판때기 하나 붙이면 될 것을 왜 꼭 PAD-cell을 사용해야만 하는가? **PAD의 기능으로는 Buffering, Level shifting, Noise reduction, IR-drop protection, ESD(electric static discharge) protection 및 decoupling capacitance의 역할 수행 등이 있다.**

√ 익숙하지 않은 단어인 ESD와 decoupling capacitance에 관해서만 설명하도록 하겠다. 완벽한 정전압원과 정정류원이 존재하지 않는다는 사실은 학부 때 이미 배워서 알고 있을 테고 그 외에도 외부로부터 유입되는 과전압 및 전류는 회로에 치명적인 손상을 줄 수 있다. 이러한 손실을 방지하기 위해 PAD-cell 내부에는 ESD 회로를 포함한다.

√ decoupling capacitance는 전압 안정화를 위한 capacitance로 빠른 이해를 원한다면 마이크로프로세서가 포함된 회로도(어떤 것이든 상관없다)를 확인해 보기 바란다. 칩 근처에 103, 104 value의 콘덴서들이 붙어 있는 것을 확인 할 수 있을 것이다. 순간적인 공급 전압의 레벨이 떨어졌을 때 이를 보상해 주기 위해 (임시 충전소 같은 역할) 칩과 최대한 가까운 곳에 배치하는 콘덴서로 Power-PAD-cell은 이런 역할도 수행한다고 한다.

√ 이제 삼성 0.13um에서 제공하는 PAD 종류에 대해 살펴보자. PAD의 종류는 다음과 같다.

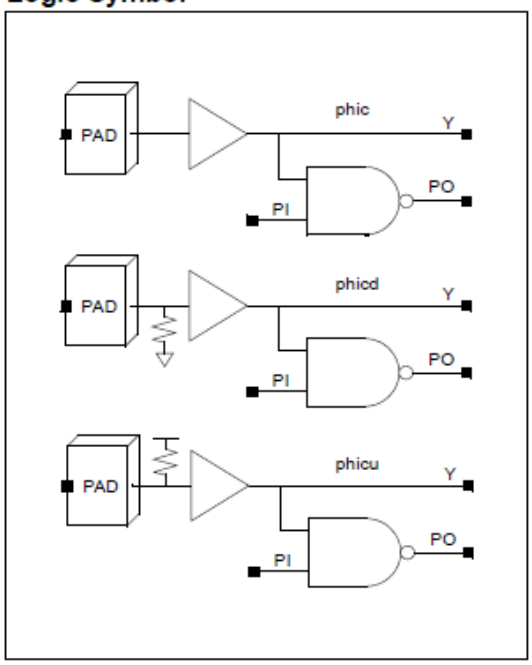
- Input Buffer (입력용)
- Output Buffer (출력용)
- Bi-Directional Buffer (입,출력 겸용)
- Oscillators (CLK)
- Power Pads (POWER)

√ Input Buffer PAD는 다음과 같다.

Cell Name	Function Description
phic/phicd/phicu	3.3V Interface CMOS Level Input Buffers
phtic/phticd/phticu	5V-Tolerant for 3.3V Interface CMOS Level Input Buffers
phis/phisd/phisu	3.3V Interface CMOS Schmitt Trigger Level Input Buffers
phtis/phtisd/phtisu	5V-tolerant for 3.3V Interface CMOS Schmitt Trigger Level Input Buffers
phnc	Metal Connection PAD for 3.3V Interface with ESD Scheme

√ phic는 3.3V Interface를 갖는 CMOS Level Input Buffer이다. 풀업, 풀다운 저항 여부에 따라 phicu, phicd로 분류된다. 여기서 주의해야할 점은 PI, PO다. 정확한 의도는 모르겠지만 반전된 입력 신호를 만드는데 사용된다. PI 값이 1일 경우 PO는 입력과 반전된 신호를 출력한다.

Logic Symbol



Truth Table

PAD	PI	Y	PO
1	1	1	0
0	x	0	1
1	0	1	1

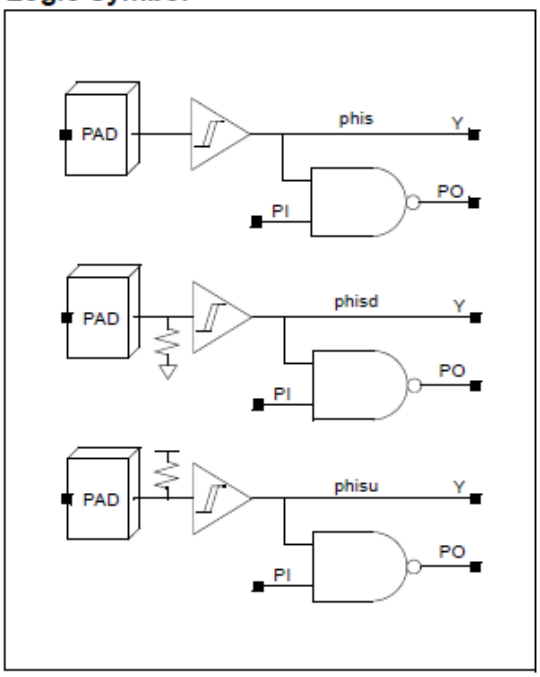
Standard Load (SL)

Cell Name	PI
phic/phicd/phicu	13.79
phtic/phticd/phticu	13.79

√ 삼성 공정의 PAD는 5V-Tolerant 기능을 제공한다. 이것은 3.3V Interface를 기본으로 하지만 **외부의 Level-Shifting 기능 없이 최대 5V Interface까지 사용을 가능하게 한다.** 모듈명은 각각 phtic, phticd(풀다운), phticu(풀업)이다. 입력 Signal의 Level이 확실히 정의되지 않았다면 가급적 Tolerant 기능을 사용할 것을 추천한다.

√ phis는 3.3V Interface를 갖는 CMOS Schmitt Trigger Level Input Buffer이다. Schmitt Trigger는 절대적인 데이터 마진을 갖기 때문에 아날로그 입력이나 노이즈가 많은 시그널에 강한 장점을 갖는다. phisd, phisu는 각각 풀다운 풀업 저항을 포함한 회로를 의미하며 phtis, phtisd, phtisu는 5V-Tolerant의 기능을 추가로 제공한다. 아래 그림은 Schmitt Trigger가 포함된 Input Buffer를 보여준다.

Logic Symbol



Truth Table

PAD	PI	Y	PO
1	1	1	0
0	x	0	1
1	0	1	1

Standard Load (SL)

Cell Name	PI
phis/phisd/phisu	13.79
phtis/phtisd/phtisu	13.79

√ 아쉽지만 삼성 공정에서는 ESD 기능을 포함한 PAD인 phnc를 공개하지 않았다.

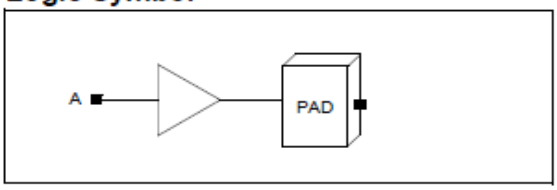
√ Output Buffer PAD는 다음과 같다.

Cell Name	Function Description
phob(2/4/8/12/16/20/24)	3.3V CMOS Normal Output Buffers
phob(4/8/12/16/20/24)sm	3.3V CMOS Normal Output Buffers with Medium Slew-Rate
phob(12/16/20/24)sh	3.3V CMOS Normal Output Buffers with High Slew-Rate
phod(2/4/8/12/16/20/24)	3.3V CMOS Open Drain Output Buffers
phod(4/8/12/16/20/24)sm	3.3V CMOS Open Drain Output Buffers with Medium Slew-Rate
phod(12/16/20/24)sh	3.3V CMOS Open Drain Output Buffers with High Slew-Rate
phot(2/4/8/12/16/20/24)	3.3V CMOS Tri-State Output Buffers
phot(4/8/12/16/20/24)sm	3.3V CMOS Tri-State Output Buffers with Medium Slew-Rate
phot(12/16/20/24)sh	3.3V CMOS Tri-State Output Buffers with High Slew-Rate
phtod(2/4/6)	5V-Tolerant for 3.3V CMOS Open Drain Output Buffers
phtod(4/6)sm	5V-Tolerant for 3.3V CMOS Open Drain Output Buffers with Medium Slew-Rate
phtot(2/4/6)	5V-Tolerant for 3.3V CMOS Tri-State Output Buffers
phtot(4/6)sm	5V-Tolerant for 3.3V CMOS Tri-State Output Buffers with Medium Slew-Rate
phdc	Metal Connection PAD for 3.3V Interface without ESD Scheme

√ 기본적인 3.3V Interface Output Buffer PAD인 phob는 medium 및 high Slew-Rate에 따라 phobsm, phobsh로 분류된다. 아래 그림은 phob의 구조를 보여준다. 간단한 구조이기 때문에 이해하는데 큰 어려움이 없을 것으로 생각된다. 중앙에 선택을 요구하는 숫자는 Current Drive(mA)를 의미한다. 데이터 시트상의 지연시간은 Current Drive가 높을수록 작은 값을 갖는다.

3.3V Interface	phob(2/4/8/12/16/20/24) phob(4/8/12/16/20/24)sm phob(12/16/20/24)sh

Logic Symbol



Truth Table

A	PAD
0	0
1	1

Standard Load (SL)

Cell Name	A
phob(2/4/8/12/16/20/24)	11.60
phob(4/8/12/16/20/24)sm	11.60
phob(12/16/20/24)sh	11.60

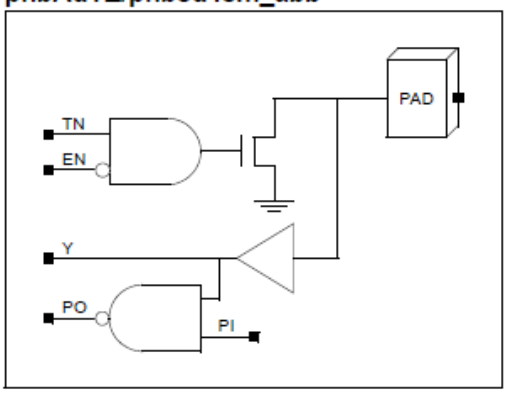
√ Output Buffer PAD는 또한 Open Drain Buffer 및 Tri-State 버퍼를 지원하며 이것들은 Slew Rate 및 5V-Tolerant 기능에 따라서 다양한 종류로 분류되어진다. 이 매뉴얼에서는 해당 패드들에 대한 자세한 언급을 하지 않을 것이며 필요할 경우 업체에서 제공하는 IO 시트를 참고하기 바란다.

√ BI-Directional Buffer PAD의 종류는 다음과 같다. In-out buffer PAD 내용을 잘 이해했다면 아래의 Buffer PAD에 관해서는 따로 설명이 없어도 쉽게 알 수 있으리라 생각한다.

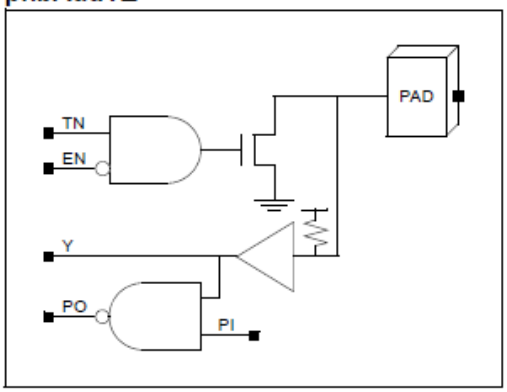
Cell Name	Function Description
phbAdYZ	3.3V Open-Drain Bi-Directional Buffers
phbAudYZ	3.3V Open-Drain Bi-Directional Buffers with Pull-Up
phtbAdYZ	5V-Tolerant for 3.3V Open-Drain Bi-Directional Buffers
phtbAudYZ	5V-Tolerant for 3.3V Open-Drain Bi-directional Buffers with Pull-Up
phbAtYZ	3.3V Tri-State Bi-Directional Buffers
phbAdtYZ	3.3V Tri-State Bi-Directional Buffers with Pull-Down
phbAutYZ	3.3V Tri-State Bi-Directional Buffers with Pull-Up
phtbAtYZ	5V-Tolerant for 3.3V Tri-State Bi-Directional Buffers
phtbAdtYZ	5V-Tolerant for 3.3V Tri-State Bi-Directional Buffers with Pull-Down
phtbAutYZ	5V-Tolerant for 3.3V Tri-State Bi-Directional Buffers with Pull-Up
phbcty_abb	3.3V Analog Tri-State Bi-Directional Buffers
phbsd4sm_abb	3.3V Analog Open-Drain Bi-Directional Buffers

Open Drain Bi-Directional Buffers

phbAdYZ/phbsd4sm_abb

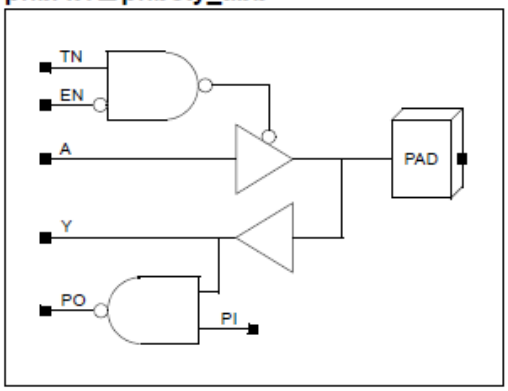


phbAudYZ

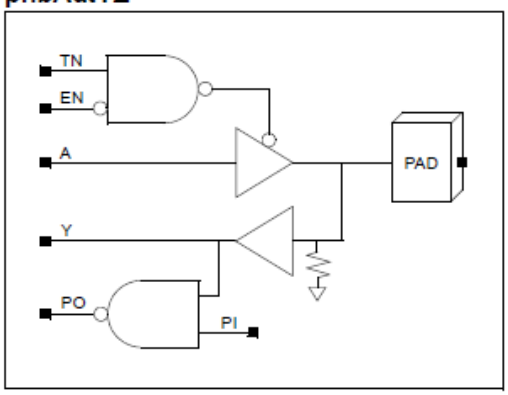


Tri-State Bi-Directional Buffers

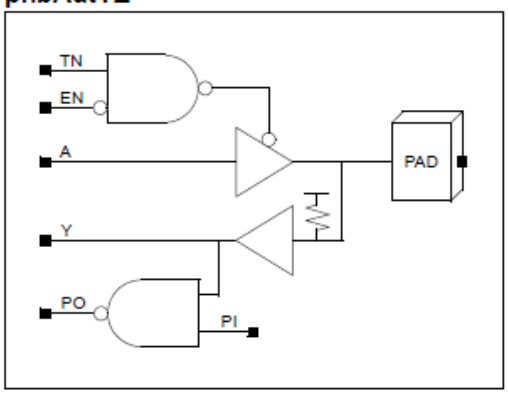
phbAtYZ/phbcty_abb



phbAdtYZ



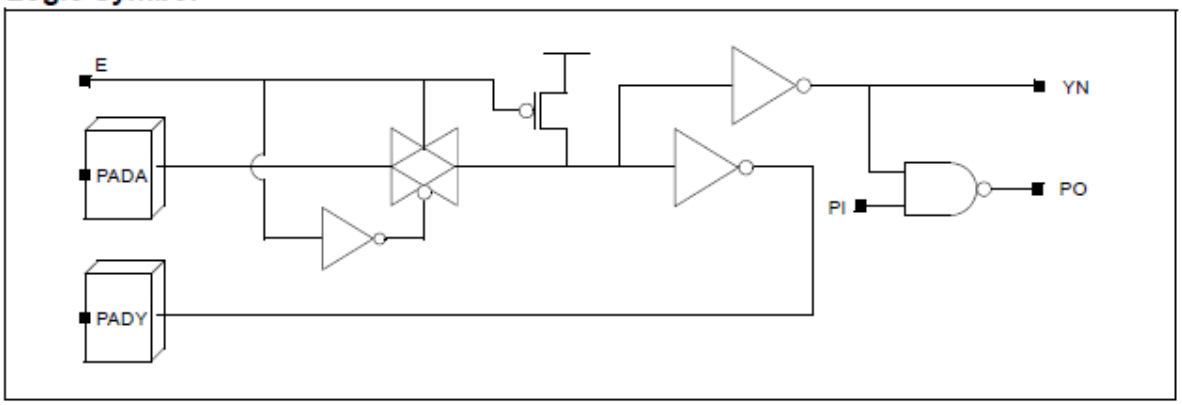
phbAutYZ



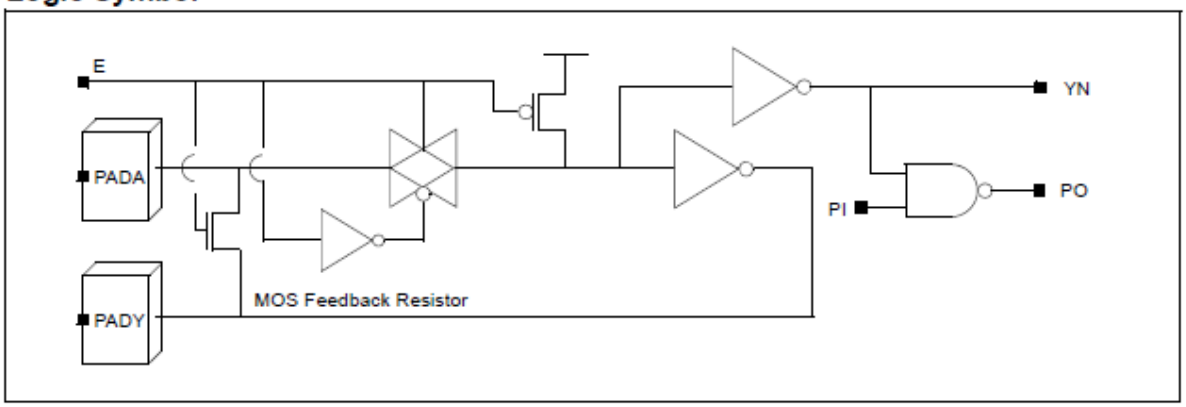
√ 다음은 Oscillator Cell에 관한 PAD의 List를 보여준다. 해당하는 동작주파수에 맞는 패드를 사용하기 바란다. 모든 Oscillator PAD는 Enable 기능을 포함하며 입력된 클럭 주파수와 180도 위상이 다른 반전 클럭을 출력하는 Output 패드와 한 쌍으로 동작한다. 즉, Oscillator는 두 개의 Port를 요구한다. 또한 내부적으로 신호를 반전 시키는 PI, PO pin도 포함되어 있다. 아래 그림은 각각 Oscillator PAD list, Enable 기능이 포함된 Oscillator PAD 및 Enable와 Feedback Register가 포함된 Oscillator PAD를 보여준다. Oscillator PAD의 사용은 기본적으로 2개의 포트가(PADA, PADY) 필요하다.

Cell Name	Function Description
phsosck1	3.3V Interface Oscillator Cell with Enable (~ 100kHz)
phsosck17	3.3V Interface Oscillator Cell with Enable and Feedback Resistor 10MΩ (~ 100kHz)
phsoscm1	3.3V Interface Oscillator Cell with Enable (1M ~ 10MHz)
phsoscm2	3.3V Interface Oscillator Cell with Enable (10M ~ 40MHz)
phsoscm3	3.3V Interface Oscillator Cell with Enable (40M ~ 100MHz)
phsoscm16	3.3V Interface Oscillator Cell with Enable and Feedback Resistor 1MΩ (1M ~ 10MHz)
phsoscm26	3.3V Interface Oscillator Cell with Enable and Feedback Resistor 1MΩ (10M ~ 40MHz)
phsoscm36	3.3V Interface Oscillator Cell with Enable and Feedback Resistor 1MΩ (40M ~ 100MHz)

Logic Symbol



Logic Symbol



√ 만약 입력이 큰 Driver를 요구하지 않을 경우 일반적인 Input buffer PAD를 Oscillator로 사용할 수도 있다. (IDEC 삼성공정 관계자의 답변임)

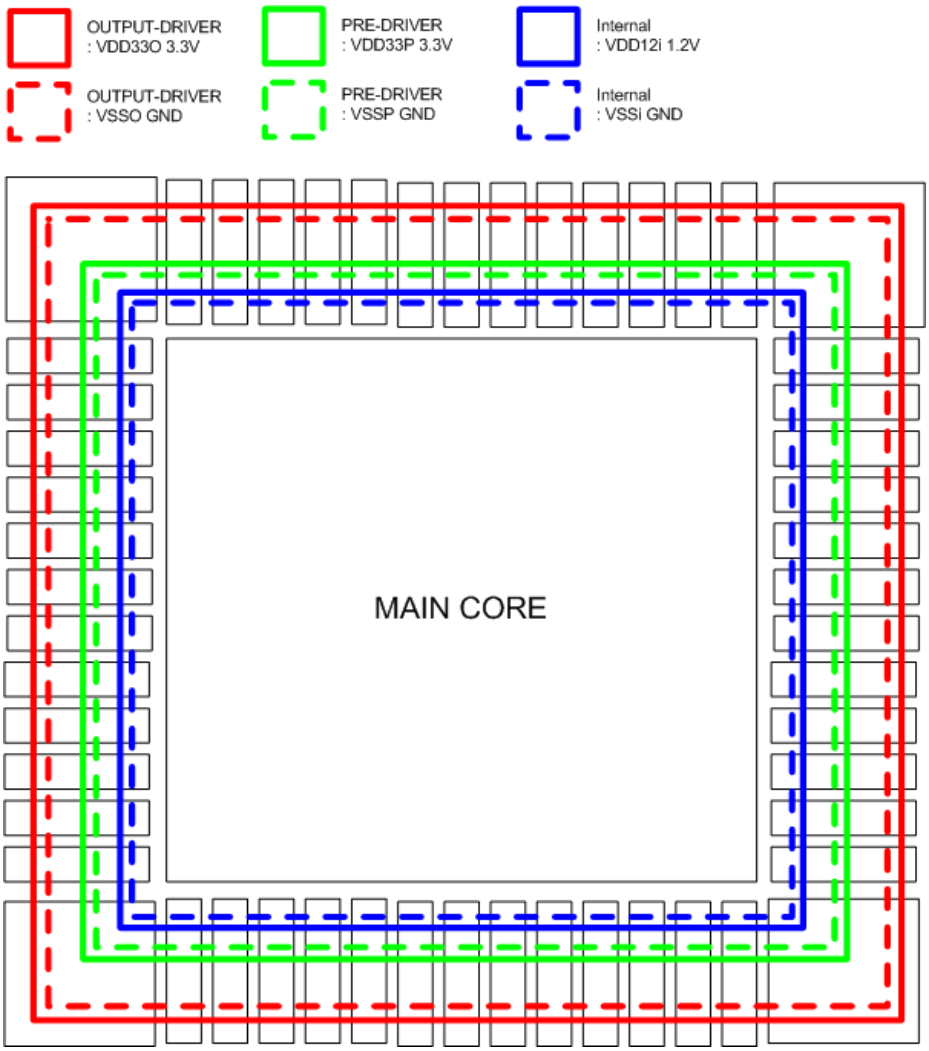
√ 마지막으로 가장 중요한 POWER PAD를 살펴보겠다. 단 이 매뉴얼에서는 Analog Power 패드의 언급은 생략한다. (Only Digital Power PAD)

√ 이제부터 정신 똑바로 차리고 보기 바란다. 아무리 코딩을 잘하고 합성을 잘했어도 POWER PAD를 잘못 사

용한다면 칩은 미동조차 하지 않는다. 이런 최악의 사태를 방지하기 위해 POWER_PAD의 종류에 대한 스터디는 좀 더 신중하고 꼼꼼하게 할 필요가 있다.

√ 삼성 0.13um 공정의 전원은 크게 다음 세 종류의 링으로 구성되어 있다. 자 여기서 링이라는 말을 주목하자. 넓은 면적의 칩의 경우 공급전원이 특정 위치의 포트에만 한정된다면 효과적인 전원 공급에 어려움을 겪을 수 있다. 따라서 아래의 그림과 같이 입력된 전원들은 칩의 최외각에 위치한 포트에 링의 형태로 전원을 연결되어 상/하/좌/우 어디 방향에서든 안정적인 전원을 공급받도록 링 전원을 연결한다. 나중에 P&R을 하면서 자연스럽게 알게 되겠지만 PAD를 통해 코어에 공급되는 전원 또한 코어 외각에 파워 링을 설치하여 전원을 공급하게 된다.

- internal (1.2V)
- Pre-driver (3.3V)
- Output-driver (3.3V)



√ 삼성 0.13um 공정에서는 Main Core 내부에서 사용되는 모든 게이트들의 공급 전압으로 1.2V를 사용한다.

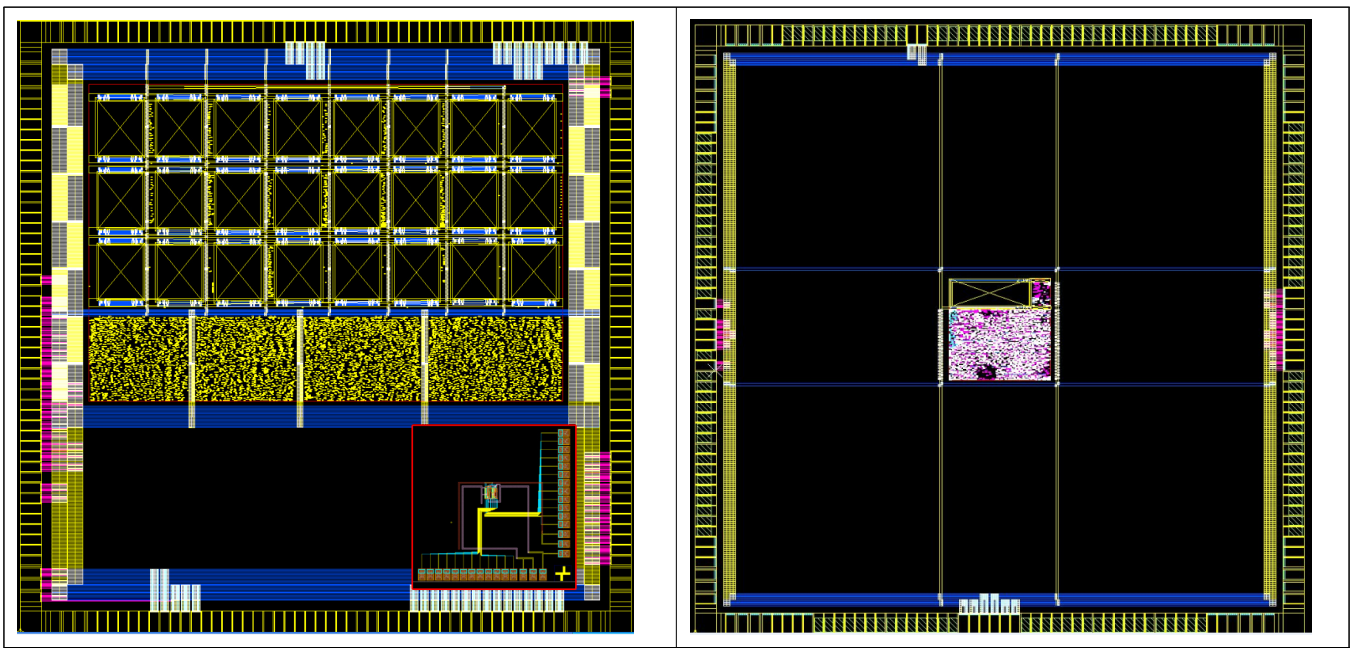
그러나 앞서 Input, Output buffer PAD를 설명할 때 이것들은 3.3V interface를 갖는다고 하였으며 심지어 5V-Tolerant 기능까지 제공한다. 그렇다면 PAD는 3.3V 혹은 5V의 신호를 받아들여 1.2V의 레벨로 낮추는 기능이 포함되어 있어야 하며 다른 의미로 PAD는 3.3V와 1.2V를 전부 공급 받아야 한다.

√ Output-Driver 및 PRE-Driver Power Ring은 아래 리스트의 3.3V Output-driver and pre-driver PAD 인 'vdd33oph' 포트를 통해 공급받고 Power Ring으로 연결된다. GND PAD는 'vssoh'를 사용한다.

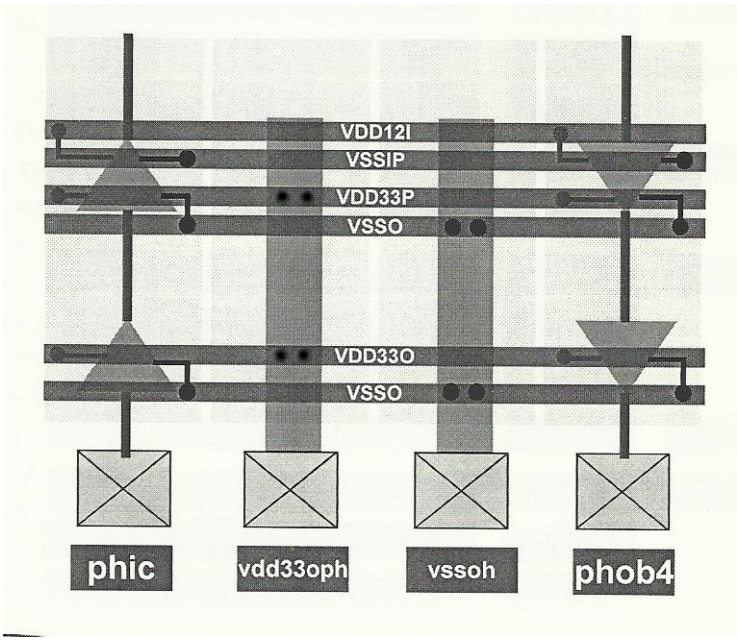
Cell Type		Cell Name	Supply Voltage	Description
Power	3.3V Interface Digital I/Os	vdd12ih	1.2V	1.2V internal for 3.3V interface
		vdd12ih_core	1.2V	1.2V core only for 3.3V interface
		vdd33oph	3.3V	3.3V output-driver and pre-driver
Ground	3.3V Interface Digital I/Os	vssiph		Internal and pre-driver GND for 3.3V interface
		vssoh		Output-driver GND for 3.3V interface

√ 이 PAD를 사용한 포트의 특징은 무엇인가? 제일 주목해야 할 부분은 코어 내부로 어떠한 전원이나 Signal 라인도 입력되지 않는다는 점이다. 단지 PAD내에 있는 3.3V로 동작하는 로직들을 위한 전원 공급 포트이다. 따라서 이 포트들은 내부에 어떠한 라인 연결도 없기 때문에 주로 칩의 모서리 부분에 위치시키는 것이 바람직하다. 만약 코어 내부로 연결되는 포트가 칩 모서리에 위치하고 MAIN CORE의 크기가 충분히 크지 않다면 이 라인들은 Auto P&R 과정에서 구불구불 복잡하게 연결될 것이고 외관상 좋지 않을 뿐만 아니라 그만큼 Delay도 증가하게 될 것이다.

√ 아래 그림은 잘못 배치된 대표적인 예이다. 왼쪽 그림의 오른쪽 상단, 왼쪽 하단을 모서리를 보자. 패드에서 내부 코어의 파워 링으로 라인들이 달라붙었다. 큰 전원 라인만 저 정도지 사실 확대하면 더 많은 라인들이 코어로 들어가기 위해 뒤죽박죽 연결되어 있다. 반대로 오른쪽 그림의 배치를 보자. 모서리 부분에 어떤 라인도 내부로 들어오지 않는 깔끔한 구조다.



√ 아래 그림의 설명으로 조금 더 명확하게 이해할 수 있으리라 생각한다. phic는 Input buffer PAD이며 phob4는 output buffer PAD이다. vdd33oph는 앞에서 언급했던 것처럼 Output Driver(VDD33O)와 PRE-DRIVER(VDD33P)와 연결되어 있으며 이들의 GND인 vssoh또한 적절하게 연결되어 있음을 알 수 있다.



√ Input PAD는 3.3V 인터페이스를 가지고 있다. 따라서 패드 내부의 첫 번째 게이트는 가장 외각에 위치한 Output-driver인 vdd33oph를 통해 전원을 공급 받는다. 물론 vdd33oph는 자체적으로 전원을 만들어내는 놈이 아니다. 설계자인 너님이 외부에서 저 패드에 3.3V 전원을 연결해 줘야 한다. GND도 마찬가지..

√ 이렇게 3.3V 레벨을 충전한 입력 Signal은 두 번째 게이트로 전달되며 여기서 PRE-Driver와 Internal power 양쪽 모두에 연결되어 있는 게이트를 통해 Level-shifting이 되어 코어로 전달된다. Output buffer PAD는 반대로 생각하면 이해하기 쉬울 것이다. 이정도면 Output-Driver와 PRE-Driver 이름의 의미가 무엇이었는지 대충 파악했으리라 생각한다.

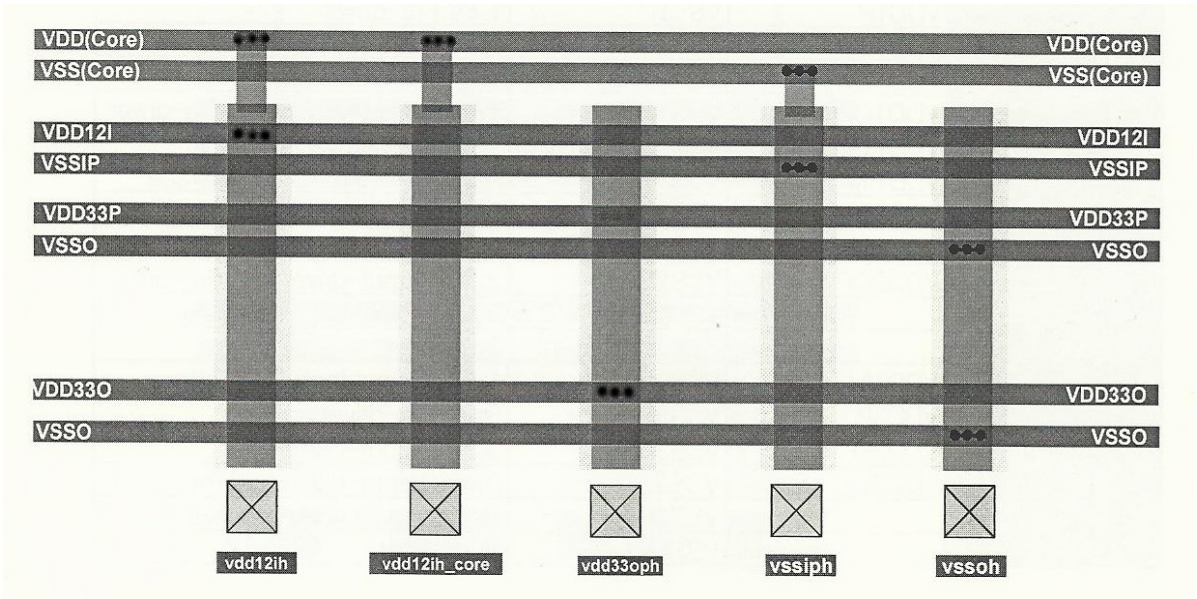
√ 그럼 Core에 전달되는 1.2V는 어떤 방식으로 공급될까? 1.2V를 공급하는 전원 PAD는 두 종류가 있다. 첫 번째는 코어 내부로만 전원을 전달하는 vdd12h_core이고 두 번째는 코어 내부와 internal_power 라인에 전원을 공급하는 vdd12ih이다. GND는 1.2V 전용 vssiph로 동일한 것을 사용한다.

√ 아래 그림은 Power PAD의 연결 상태를 보여준다. vdd12ih는 외부에서 사용자로부터 연결된 1.2V의 전압을 internal power ring과 내부 코어로 전달하는 역할을 한다. vdd12h_core는 외부에서 입력되는 1.2V의 전압을 오직 Core에게만 전달한다. 코어로 전달되는 전원은 상/하/좌/우 모든 방향의 중앙에 위치시킬 것을 추천한다.

√ GND PAD는 포트의 여유가 있다면 가급적 많이 삽입하는 것이 전원 안정화에 유리하며 CLK과 같은 높은 주파수의 PAD 주변은 GND PAD 및 라인으로 주변을 감싸 다른 신호를 간섭하는 것을 최소화 하는 것이 좋다.

√ 아래 그림의 VDD(core), VSS(core)는 코어 내에 존재하는 로직들의 전원을 공급하기 위한 Power ring으로 P&R 과정에서 사용자가 직접 만들어 넣는다. 이전 페이지의 칩 사진에 보이는 노란색과 파란색 굵은 라인이 그것이다.

√ 이제 PAD에 관한 설명을 끝내겠다. 자세한 내용은 공정에서 제공하는 데이터 시트를 참조하기 바라며 지금 설명한 PAD는 삼성 공정을 위한 PAD이다. 물론 비슷한 부분도 있겠지만 다른 공정의 경우 다른 PAD의 종류와 구조를 갖고 있다. 해당 공정을 사용하기 전에 PAD에 관한 정보는 반드시 숙지하고 넘어가기 바란다.



√ 본격적인 PAD 파일을 만들어보겠다. 삼성 공정에서는 MPW 프로젝트 수행을 위해 4x4mm 코어 사이즈, 208핀의 칩을 설계할 수 있는 PAD 좌표데이터 tdf파일을 전달했다. (이것은 P&R할 때 자세히 설명할 것임) 즉, 우리가 사용할 수 있는 포트의 최대 수는 208개이며 각 면마다 52개의 포트가 배치된다.

√ PAD cell은 하나의 게이트처럼 HDL코딩이 가능하며 최상위(Top) 모듈보다 상위 계층에 설계되고 PAD를 포함한 모듈이 Design의 최종 Top모듈이 된다.

√ 자, 그렇다면 현재 우리가 실습하게 될 디자인의 Top module은 어떻게 구성되었는지 살펴보자. 단일 클럭인 clk와 동기식 reset이 사용되고 8bit-width를 갖는 입력이 2개(16개의 input ports)존재한다. output port는 12bit-width를 갖는다.

- input clk
- input reset
- input [7:0] in_MB;
- input [7:0] in_SR;
- output [11:0] out_4by4SAD

√ Total 30개의 Port가 필요하다. 와우! 178개의 포트가 남았다.. 빵빵한 전원과 안정적인 GND를 얻을 수 있다. 이 얼마나 Lucky한 경우란 말인가... 라고 생각할 수 있지만 사실 이것은 의도한 설계다. 지난 3번의

MPW프로젝트 중 첫 번째가 통신 칩 남은 두 번의 경우가 영상 칩 설계였었다. 영상 칩을 설계했을 때 포트의 수가 여유 있었던 적이 단 한 번도 없었다. 입/출력 영상 데이터의 대역폭이 좀 넓은가, 다른 경우도 마찬가지일 것이다. 특히 완성된 칩이 정상 파형 출력에 실패했을 때 설계자가 할 수 있는 것이 아무것도 없다는 점을 고려한다면, 모든 디자인은 문제를 파악하기 쉬운 구조로 파티셔닝이 되어야만 하고 파티셔닝이 된 후 각 스테이지의 결과는 포트를 통해 출력되어야 한다. 그뿐인가 DFT를 시작하고 나면 SCAN-CHAIN 관련 포트들의 추가도 고려해야 한다. 결국 어떤 설계자도 포트 수 제약은 한번쯤 생각해봐야할 문제가 될 것이다. 해결책은 HDL을 설계하는 사람이 상황과 조건에 맞춰 스스로 찾아야 한다.

√ 자, 그럼 지금 수행하고 있는 프로젝트의 상황과 조건을 따져보자. 일단 인상적인 것은 4x4mm의 매우 널찍한 코어사이즈다. MPW라는 프로젝트 특성상 코어 혹은 다이사이즈의 변경은 불가능하다. 그렇다면 컨트롤러와의 부피와 SRAM의 수를 늘림으로써 포트의 수를 절약 하는 방법을 택할 수 있다. 추가적인 장점으로 만약 넓은 코어에 작은 크기의 모듈이라면 어떤 일이 발생할까? 아마 포트부터 코어까지 연결되는 긴 와이어를 목격하게 될 것이고 이것이 Transition과 Hold time에 어마어마한 영향을 미친다는 사실 그리고 결국 엄청난 수의 버퍼 체인 혹은 인버터 체인을 삽입해야 한다는 사실을 깨닫게 된다. 고로 모듈의 크기를 적당히 부풀릴 필요도 있다는 말이다. **하지만 반드시 기억하기 바란다. 이건 내가 지금 하고 있는 프로젝트에만 해당하는 이야기다. 너님의 프로젝트는 너님의 상황과 환경에 맞춰 설계조건을 설정해라.**

√ 이제 TEST PORT에 관해서 생각해보자. 이 프로젝트에서는 실제 칩을 만들지 않는다. 물론 MPW를 수행했던 공정의 자료를 이용했지만 지금 MPW를 하고 있는 것은 아니다. Post-Sim까지가 최종 목표다. Post-Sim에서는 내가 설계한 디자인의 어느 부위든 구석구석 살펴볼 수 있다. 아~ TEST PORT 쓰지 말자! 난 필요 없다. **단, 다시 언급하지만 매우 주의하기 바란다. 나는 현재 내가 하고 있는 프로젝트의 상황과 조건을 기준으로 설계지침을 만들고 있다. 만약 너님이 실제로 칩을 만들 때 지금의 내 기준을 생각 없이 그대로 적용하게 된다면 칩이 실패했을 때 왜 동작을 안 하냐는 교수님의 질문을 받게 될 것이고 너님께서는 손가락을 빨며 보드나 계측기 혹은 정전기로 칩이 불량 났다는 핑계를 생각해야 할지도 모른다. '시뮬레이션은 잘 나왔었는데' 라는 말을 덧붙이며... 반드시 어떻게 설계할지 생각해보기 바란다.**

√ 이러한 조건을 기반으로 아래와 같이 PAD 모듈의 디자인을 설계하였다.

```
module pad_SAD_4by4_Reference(clk, clk_out, reset, in_MB, in_SR, out_4by4SAD);

        input clk;
        input reset;
        input [7:0] in_MB;
        input [7:0] in_SR;
        output clk_out;         // CLK는 두 개의 PAD를 요구하기 때문에 포트 추가!
        output [11:0] out_4by4SAD;


        wire clk;
        wire reset;
        wire [7:0] in_MB;
```

```
 wire [7:0] in_SR;
 wire [11:0] out_4by4SAD;

 wire clk_out;

// _p : PAD의 출력 신호, MAIN 모듈에 실제로 입력되는 데이터
 wire clk_p;
 wire reset_p;
 wire [7:0] in_MB_p;
 wire [7:0] in_SR_p;
 wire [11:0] out_4by4SAD_p;

 ////////////////////////// LEFT
 // PAD1 ~ PAD14 : 칩의 모서리 부분, PAD IO 전원 공급 3.3V
 vssoh          pad1();
 vssoh          pad2();
 vssoh          pad3();
 vdd33oph        pad4();
 vssoh          pad5();
 vssoh          pad6();
 vssoh          pad7();

 vssoh          pad8();
 vssoh          pad9();
 vssoh          pad10();
 vdd33oph        pad11();
 vssoh          pad12();
 vssoh          pad13();
 vssoh          pad14();

// PAD15 ~ PAD21 : RESET, 추후 이 부분은 SCAN CHAIN PAD로 변경될 예정
 vssoh          pad15();
 vssoh          pad16();
 vssoh          pad17();
 phic           pad18(.PAD(reset), .PI(1'b0), .PO(), .Y(reset_p));
 vssoh          pad19();
 vssoh          pad20();
 vssoh          pad21();
```

```
// PAD22 ~ PAD 28 : 칩의 중앙, 코어 내부 및 Internal Ring 전원 공급
vssiph          pad22();
vssiph          pad23();
vssiph          pad24();
vdd12ih         pad25();
vssiph          pad26();
vssiph          pad27();
vssiph          pad28();

//  PAD29, PAD32 : 신호 간섭 방지용 클럭 실드 GND
//  PAD32, PAD31 : 클럭 PAD, 기본적으로 두 개의 패드를 요구함.
vssiph          pad29();
phsoscm3        pad30(.PADA(clk), .E(1'b1), .PI(), .PO(), .YN(clk_p), .PADY(clk_out));

vssiph          pad32();

// PAD33 ~ PAD39 : 칩의 중앙, 코어 내부 및 Internal Ring 전원 공급
vssiph          pad33();
vssiph          pad34();
vssiph          pad35();
vdd12ih         pad36();
vssiph          pad37();
vssiph          pad38();
vssiph          pad39();

// PAD40 ~ PAD52 : 칩의 모서리 부분, PAD IO 전원 공급 3.3V
vssoh           pad40();
vssoh           pad41();
vssoh           pad42();
vdd33oph        pad43();
vssoh           pad44();
vssoh           pad45();
vssoh           pad46();

vssoh           pad47();
vssoh           pad48();
vssoh           pad49();
vdd33oph        pad50();
vssoh           pad51();
vssoh           pad52();
```

```
//////////////////////////////////////////// BOTTOM

// PAD53~ PAD66 : 칩의 모서리 부분, PAD IO 전원 공급 3.3V
vssoh           pad53();
vssoh           pad54();
vssoh           pad55();
vdd33oph         pad56();
vssoh           pad57();
vssoh           pad58();
vssoh           pad59();

vssoh           pad60();
vssoh           pad61();
vssoh           pad62();
vdd33oph         pad63();
vssoh           pad64();
vssoh           pad65();
vssoh           pad66();

// PAD67 ~ PAD74 : DATA INPUT BUFFER PAD
phic            pad67(.PAD(in_MB[0]), .PI(1'b0), .PO(), .Y(in_MB_p[0]));
phic            pad68(.PAD(in_MB[1]), .PI(1'b0), .PO(), .Y(in_MB_p[1]));
phic            pad69(.PAD(in_MB[2]), .PI(1'b0), .PO(), .Y(in_MB_p[2]));
phic            pad70(.PAD(in_MB[3]), .PI(1'b0), .PO(), .Y(in_MB_p[3]));
phic            pad71(.PAD(in_MB[4]), .PI(1'b0), .PO(), .Y(in_MB_p[4]));
phic            pad72(.PAD(in_MB[5]), .PI(1'b0), .PO(), .Y(in_MB_p[5]));
phic            pad73(.PAD(in_MB[6]), .PI(1'b0), .PO(), .Y(in_MB_p[6]));
phic            pad74(.PAD(in_MB[7]), .PI(1'b0), .PO(), .Y(in_MB_p[7]));

// PAD75 ~ PAD81 : 칩의 중앙, 코어 내부 및 Internal Ring 전원 공급
vssiph          pad75();
vssiph          pad76();
vssiph          pad77();
vdd12ih          pad78();
vssiph          pad79();
vssiph          pad80();
vssiph          pad81();
```

```
 // PAD82 ~ PAD89 : DATA INPUT BUFFER PAD
 phic            pad82(.PAD(in_SR[0]), .PI(1'b0), .PO(), .Y(in_SR_p[0]));
 phic            pad83(.PAD(in_SR[1]), .PI(1'b0), .PO(), .Y(in_SR_p[1]));
 phic            pad84(.PAD(in_SR[2]), .PI(1'b0), .PO(), .Y(in_SR_p[2]));
 phic            pad85(.PAD(in_SR[3]), .PI(1'b0), .PO(), .Y(in_SR_p[3]));
 phic            pad86(.PAD(in_SR[4]), .PI(1'b0), .PO(), .Y(in_SR_p[4]));
 phic            pad87(.PAD(in_SR[5]), .PI(1'b0), .PO(), .Y(in_SR_p[5]));
 phic            pad88(.PAD(in_SR[6]), .PI(1'b0), .PO(), .Y(in_SR_p[6]));
 phic            pad89(.PAD(in_SR[7]), .PI(1'b0), .PO(), .Y(in_SR_p[7]));

// PAD90~ PAD104 : 칩의 모서리 부분, PAD IO 전원 공급 3.3V
 vssoh           pad90();
 vssoh           pad91();
 vssoh           pad92();
 vdd33oph         pad93();
 vssoh           pad94();
 vssoh           pad95();
 vssoh           pad96();

 vssoh           pad97();
 vssoh           pad98();
 vssoh           pad99();
 vdd33oph         pad100();
 vdd33oph         pad101();
 vssoh           pad102();
 vssoh           pad103();
 vssoh           pad104();


////////////////////////////////////////////  RIGHT

// PAD105 ~ PAD118 : 칩의 모서리 부분, PAD IO 전원 공급 3.3V
 vssoh           pad105();
 vssoh           pad106();
 vssoh           pad107();
 vdd33oph         pad108();
 vssoh           pad109();
 vssoh           pad110();
 vssoh           pad111();
```

```
vssoh           pad112();
vssoh           pad113();
vssoh           pad114();
vdd33oph        pad115();
vssoh           pad116();
vssoh           pad117();
vssoh           pad118();

// PAD119 ~ PAD125 : 칩의 중앙, 코어 내부 전원 공급
vssiph          pad119();
vssiph          pad120();
vssiph          pad121();
vdd12ih_core    pad122();
vssiph          pad123();
vssiph          pad124();
vssiph          pad125();

// PAD126 ~ PAD132 : 칩의 중앙, 코어 내부 및 Internal Ring 전원 공급
vssiph          pad126();
vssiph          pad127();
vssiph          pad128();
vdd12ih         pad129();
vssiph          pad130();
vssiph          pad131();
vssiph          pad132();

// PAD133 ~ PAD139 : 칩의 중앙, 코어 내부 전원 공급
vssiph          pad133();
vssiph          pad134();
vssiph          pad135();
vdd12ih_core    pad136();
vssiph          pad137();
vssiph          pad138();
vssiph          pad139();

// PAD140 ~ PAD156 : 칩의 모서리 부분, PAD IO 전원 공급 3.3V
vssoh           pad140();
vssoh           pad141();
vssoh           pad142();
```

```
        vdd33oph        pad143();
        vssoh           pad144();
        vssoh           pad145();
        vssoh           pad146();

        vssoh           pad147();
        vssoh           pad148();
        vssoh           pad149();
        vssoh           pad150();
        vdd33oph        pad151();
        vdd33oph        pad152();
        vssoh           pad153();
        vssoh           pad154();
        vssoh           pad155();
        vssoh           pad156();

      ////////////////////////////////////  TOP

        // PAD157 ~ PAD173 : 칩의 모서리 부분, PAD IO 전원 공급 3.3V
        vssoh           pad157();
        vssoh           pad158();
        vssoh           pad159();
        vssoh           pad160();
        vdd33oph        pad161();
        vdd33oph        pad162();
        vssoh           pad163();
        vssoh           pad164();
        vssoh           pad165();
        vssoh           pad166();

        vssoh           pad167();
        vssoh           pad168();
        vssoh           pad169();
        vdd33oph        pad170();
        vssoh           pad171();
        vssoh           pad172();
        vssoh           pad173();

        // PAD174 ~ PAD185 : OUTPUT BUFFER PAD
        phob12          pad174(.A(out_4by4SAD_p[0]), .PAD(out_4by4SAD[0]));
```

```
phob12          pad175(.A(out_4by4SAD_p[1]), .PAD(out_4by4SAD[1]));
phob12          pad176(.A(out_4by4SAD_p[2]), .PAD(out_4by4SAD[2]));
phob12          pad177(.A(out_4by4SAD_p[3]), .PAD(out_4by4SAD[3]));
phob12          pad178(.A(out_4by4SAD_p[4]), .PAD(out_4by4SAD[4]));
phob12          pad179(.A(out_4by4SAD_p[5]), .PAD(out_4by4SAD[5]));
phob12          pad180(.A(out_4by4SAD_p[6]), .PAD(out_4by4SAD[6]));
phob12          pad181(.A(out_4by4SAD_p[7]), .PAD(out_4by4SAD[7]));
phob12          pad182(.A(out_4by4SAD_p[8]), .PAD(out_4by4SAD[8]));
phob12          pad183(.A(out_4by4SAD_p[9]), .PAD(out_4by4SAD[9]));
phob12          pad184(.A(out_4by4SAD_p[10]), .PAD(out_4by4SAD[10]));
phob12          pad185(.A(out_4by4SAD_p[11]), .PAD(out_4by4SAD[11]));


// PAD186 ~ PAD190 : OUTPUT SIGNAL 공간 확보를 위한 GND
vssoh           pad186();
vssoh           pad187();
vdd33oph        pad188();
vssoh           pad189();
vssoh           pad190();

// PAD191 ~ PAD197 : 칩의 중앙, 코어 내부 및 Internal Ring 전원 공급
vssiph          pad191();
vssiph          pad192();
vssiph          pad193();
vdd12ih         pad194();
vssiph          pad195();
vssiph          pad196();
vssiph          pad197();

// PAD198 ~ PAD208 : 칩의 모서리 부분, PAD IO 전원 공급 3.3V
vssoh           pad198();
vssoh           pad199();
vssoh           pad200();
vssoh           pad201();
vdd33oph        pad202();
vdd33oph        pad203();
vdd33oph        pad204();
vssoh           pad205();
vssoh           pad206();
vssoh           pad207();
```

```
        vssoh               pad208();



    // MAIN CORE MAPPING
       SAD_4by4_Reference s44r(      .clk(clk_p),
                                     .reset(reset_p),
                                     .in_MB(in_MB_p),
                                     .in_SR(in_SR_p),
                                     .out_4by4SAD(out_4by4SAD_p)
                                );

endmodule
```

√ 참고로 몇몇 포트의 정보는 SCAN-CHAIN 및 관련된 제어신호 삽입으로 인하여 변경될 것이다.

2. Basic Synthesis

√ 모든 준비가 끝났다. HDL 디자인도 있고 PAD도 붙였고 라이브러리도 준비되었다. 이제 합성의 첫 단추인 라이브러리 설정을 시작해 보자.

√ 라이브러리를 설정 하는 방법은 세 가지로 구분된다.

√ 첫 번째로 System-wide 설정 방법이 있다. 이것은 Synopsys에서 제공하는 .synopsys_dc.setup 파일을 수정하여 설정 할 수 있다. 이 파일은 $SYNOPSYS/admin/setup(Tool 설치 시 자동 생성 폴더)에 존재하며 Target, link, symbol library, Path 및 각종 Tool 실행 환경 변수 및 path들을 설정할 수 있다. 이 파일은 시스템 관리자만이 변경할 수 있으며 DC가 실행될 때 자동으로 실행된다.

√ 실제 이 파일을 살짝 열어보자. 보이는가? 라이브러리 설정하는 부분이? 현재 이 파일에는 기본적인 공통 변수 및 Path만 설정되었을 뿐 라이브러리도 설정은 되어 있지 않다.

```
# from the System Variable Group
set link_force_case  "check_reference"
set link_library  { * your_library.db }

set search_path [list . ${synopsys_root}/libraries/syn ${synopsys_root},
set target_library  your_library.db
set synthetic_library  ""
set command_log_file  "./command.log"
set designer  ""
set company  ""
set find_converts_name_lists  "false"

set symbol_library  your_library.sdb
```

√ 두 번째는 user-defined 설정 방법이다. 이것은 라이브러리 설정 뿐 아니라 사용자가 추가로 설정해야 할 Design의 공통 변수 혹은 search_path등을 설정할 수 있다. 또한 System-wide 파일이 설정된 이후에 실행되며 덮어쓰기 방식으로 설정된다. 이 파일은 .synopsys_dc.setup 파일명으로 사용자의 Home 폴더에 위치시켜야 DC 실행과 함께 자동 실행된다.

√ 마지막 방법은 Design-specific 설정 방법이다. 특별한 경우 공정사에서 제공하는 몇 가지 option 들을 이 파일에 기술 하는 경우가 있다. 가령 don't_use의 정의 혹은 name rule 같은 것들이 있는데 이와 같은 경우는 공정사에서 제공하는 setup file을 참조하여 사용하면 될 것이다. 물론 이 파일을 이용하여 라이브러리를 설정 할 수도 있다. 이 파일은 프로젝트의 CWD(current working directory) 폴더에 .synopsys_dc.setup 파일명을 갖고 있어야 DC 수행과 함께 자동으로 실행되어진다.

√ DC는 툴 시작과 함께 환경 파일인 .synopsys_dc.setup을 실행하기 위해 찾기 시작하는데 위에서 언급한 순서대로 ① system-wide($SYNOPSYS/admin/setup), ② user-defined(User home directory), ③ design-specific(Project Working directory 'CWD')의 순서로 찾아 실행하며 기존의 인식된 파일 위에 다시 설정되어지는 방식이다. 세 군데 중 라이브러리를 설정한 파일이 하나는 반드시 존재해야 한다.

√ 삼성은 친절하게도 공정에 최적화된 setup 파일을 제공한다. 따라서 우리는 세 번째 방법인 Design-specific 설정 방법을 사용할 것이다. 또한 가시적인 결과 확인을 위해 자동 실행 파일(.synopsys_dc.setup)을 사용하지 않고 따로 정의한 파일을 DC 시작 전에 직접 실행하도록 하겠다.

√ 공정에서 제공한 라이브러리 및 환경설정 파일의 이름은 'synopsys_dc_sam0.3.setup.tcl'이다. 앞에서도 설명했지만 이 파일의 이름을 .synopsys_dc.setup로 변경한 후 CWD 폴더에 위치시키면 DC 툴이 실행됨과 동시에 자동 실행된다. 주의 할 점 파일명을 보면 알겠지만 이것은 숨겨진 속성을 갖는 파일이다.

√ 이제 이 파일의 내용을 간략히 살펴보자. 라이브러리만 입력한 후 그냥 실행하고 써도 되겠지만 우리가 단순히 합성만 시켜주는 기계도 아니고 최소한 뭐가 있구나! 정도는 알아야 하지 않겠는가.

```
#*****************************************************************************
#**                Synopsys environment file for STD130 library             **
#*****************************************************************************
remove_design -all

setenv SEC_SYNOPSYS /home/Dell/mschoi/dk_home/samsung013/
sec050915_0050_STD150E_regular_DK_Synopsys_N/sec150e_synopsys

setenv SEC_SYNOPSYS_AUX /Dell/home/mschoi/dk_home/samsung013/
sec050915_0050_STD150E_regular_DK_Synopsys_N/sec150e_synopsys/aux

#*****************************************************************************
#**                Set New Variable for 'search_path'                       **
#*****************************************************************************
set dk_home [getenv SEC_SYNOPSYS]
```

```
set dk_home_aux [getenv SEC_SYNOPSYS_AUX]

#*****************************************************************************
#**                Set 'search_path'                                        **
#*****************************************************************************
set search_path [list . [format "%s%s"  $synopsys_root /libraries/syn] \
    [format "%s%s"  $dk_home /syn/STD150E] $dk_home_aux ]
```

제일 먼저 설치된 라이브러리의 PATH를 지정한다.
앞서 주절주절한 것처럼 개인계정에 설치한 라이브러리 PATH를 등록했다. (너님은 너님 path 등록해 써라)
각각 SEC_SYNOPSYS, SEC_SYNOPSYS_AUX로 정의한다.
정의된 PATH는 dk_home, dk_home_aux라는 이름의 변수로 설정되고
이 녀석으로 search_path를 설정한다.
내 계정에 설치된 삼성 라이브러리의 search_path를 설정하는 과정이었다.
※ setevn은 C언어의 Define과 유사한 기능을 갖는 Tcl 명령이임.

```
#*****************************************************************************
#**                Set libraries                                            **
#*****************************************************************************

# Variables for later use
set STD_TYP std150e_typ_120_p025
set STD_TYP_MEM std150e_typ_120_p025_memory

# Set libraries: Target Tech & Link
set target_library [concat \
        $STD_TYP.db \
        $STD_TYP_MEM.db
]

set link_library [concat \
        * \
        $target_library
]

set link_path $link_library

set symbol_library [list std150e_veri.sdb]
```

Target, Link, Symbol 라이브러리를 설정한다.
기본적으로 라이브러리를 설정하는 방법은 아래와 같다.
- set target_library your_library.db
- set link_library * your_library.db
- set symbol_library your_library.sdb

concat는 문자열을 연결해 주는 명령어로 다수의 라이브러리를 설정할 때 유용하게 사용할 수 있다.
여기서는 Standard 라이브러리와 Memory를 위한 라이브러리를 동시에 설정했다.
각 라이브러리의 설명은 DC매뉴얼 (1)장 '디자인 합성의 시작'을 참고하기 바란다.

```
#******************************************************************************
#** NAMING RULE FOR Verilog HDL:                                              **
#** If you have a plan to layout your chip with Apollo,                       **
#** you MUST use this naming rule.                                            **
#**  [1] max_length was removed                                               **
#**  [2] '!' character was allowed for CTS-ed netlist.                        **
#******************************************************************************
define_name_rules sec_verilog -type port \
                              -equal_ports_nets  \
                              -allowed {A-Z a-z 0-9 _ [] !} \
                              -first_restricted {0-9 _ !}   \
                              -last_restricted {_ !}

define_name_rules sec_verilog -type cell \
                              -allowed {A-Z a-z 0-9 _ !} \
                              -first_restricted {0-9 _ !} \
                              -last_restricted {_ !} \
                              -map {{{"*cell*", "U"}, {"*-return", "RET"}}}

define_name_rules sec_verilog -type net \
                              -equal_ports_nets \
                              -allowed {A-Z a-z 0-9 _ !} \
                              -first_restricted {0-9 _ !} \
                              -last_restricted {_ !} \
                              -map {{{"*cell*", "U"}, {"*-return", "RET"}}}
```

다음은 Naming Rule에 관한 내용이다. Verilog naming rule 부분만 가져왔다.
각각 port, cell, net에 관한 규칙이며 port의 내용만 살펴보면 port와 net의 이름은 동일할 수 있고
대소문자, 숫자, 그리고 [] !(느낌표) 만 사용 가능하다
첫 문자로 숫자 혹은 ! 기호가 올수 없고
마지막 문자로 !를 사용할 수 없다. 뭐 대충 이런 내용이다. -_-;;

이 밖에 잡다한 환경 설정이 있는데 대부분 사용 하나 안하나(true, false)를 결정하기 위한 것이니
참고하기 바란다.

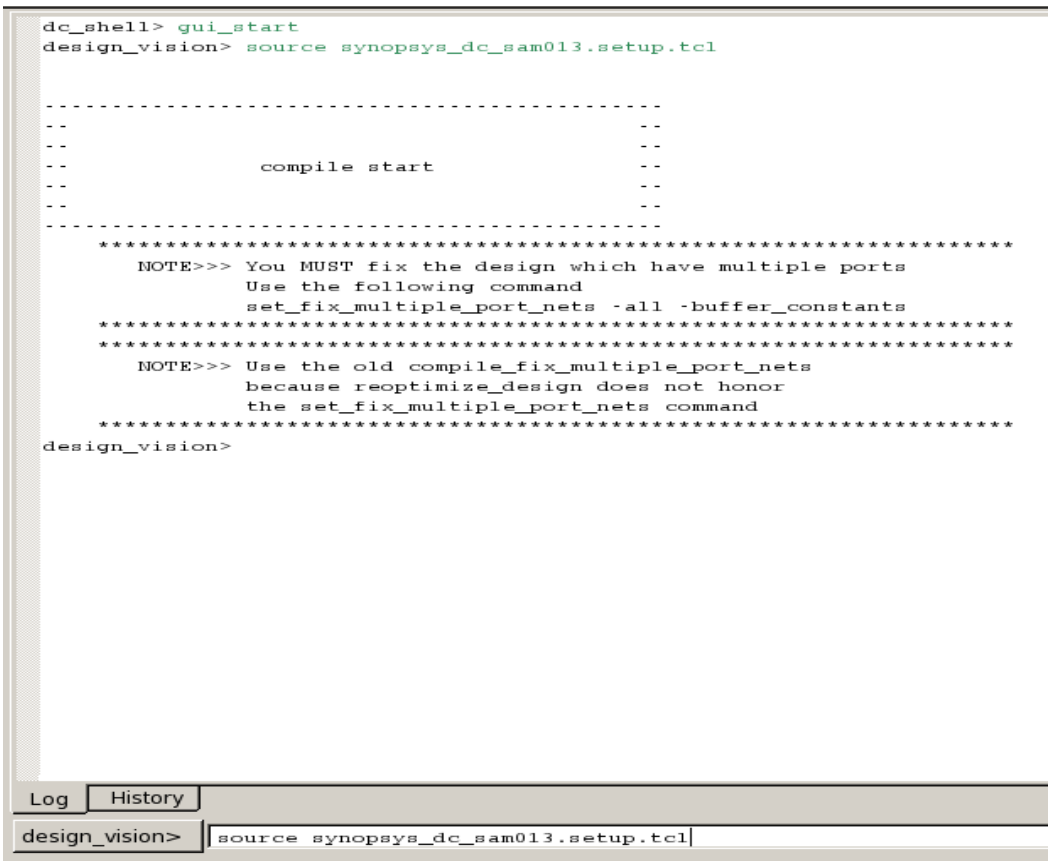
√ 이렇게 완성된 파일을 아래와 같이 지난번 합성을 위해 생성한 폴더에 copy한 후 실행해보자.

```
[mschoi@Dell syn]$ ls
command.log  db  design  log  synopsys_dc_sam013.setup.tcl  work
```

√ 실행 방법은 간단하다. 현재 폴더에서 디자인 컴파일러를 실행시킨 후 GUI창 Commend line 에서 가볍게 아래 명령어를 실행한다.

source synopsys_dc_sam013.setup.tcl

√ 아래 그림처럼 나오면 성공이다. 별거 아니지만 축하한다. GUI 화면에 관한 설명은 생략하겠다. 공돌이라면 누구나 쉽게 알아먹을 수 있는 그저 그런 디자인에 고전적인 구조의 창이므로 쉽게 이해할 것이라 예상한다. 앞으로 간단한 것들은 종종 빼먹을 예정이다. 사실 여기까지 10분이면 끝날 작업인데 보이는가? 벌써 23페이지다. 이런 속도라면 덜덜;; 한숨밖에 안 나온다. 괜히 시작했다는 생각이 든다. 그러니 이런 잡 설명 좀 빼먹어도 이해해주기 바란다. -_-;;



√ 다음 단계로 합성을 위한 HDL 코드를 Read해 보자. 현재 열려있는 GUI창의 메뉴로 하나하나 클릭할 수도 있다. 합성을 한 번에 성공할 자신 있으면 그렇게 하고 아니라면 명령어를 스크립트를 작성하여 앞서 실행했던 것처럼 Source 명령어로 한 번에 실행하자.

√ 만약 이 스크립트 파일을 실행 했을 때 너무 많은 오류로 문제를 파악하기 어렵다면 작성된 스크립트의 명령어를 각 라인별로 복사 붙여넣기 하여 단계별로 하나씩 실행 하며 확인 할 수 있다. 참고로 DC의 경우 선행 명령어에 오류가 발생했다면 그 뒤에 있는 명령어 또한 전부 다 오류를 만든다.

√ 그럼 다시 스크립트 작성을 위한 명령어를 살펴보자. Verilog 파일을 읽기 위한 명령어는 read_verilog이다. 해당 명령어를 사용하여 ./design 폴더에 위치한 HDL 파일들을 전부 읽어 들이자 (작성한 PAD파일 포함). 단, HARD IP 타입의 SRAM (HARD IP VERIFICATION 매뉴얼 참조) 및 테스트 벤치 그리고 테스트를 위한 백터 자료들은 read하지 않는다. PAD는 합성된 netlist 결과에 연결정보만 붙여서 사용할 수 있고 디자인에 함께 포함해 합성할 수도 있다. PAD를 적용하는 방법에 따라 Constraint 설정법도 달라진다. 우리는 좀 더 정밀한 방법인 후자의 방법을 사용할 것이다.

```
read_verilog ./design/Controller.v      //  for VHDL : read_vhdl
read_verilog ./design/SAD_4by4_Reference.v
read_verilog ./design/Stage1_PE_SAD.v
read_verilog ./design/Stage2_RCA_8bits.v
read_verilog ./design/Stage3_RCA_8bits.v
read_verilog ./design/data_holder.v
read_verilog ./design/SAD_4by4.v
read_verilog ./design/data_dist.v
read_verilog ./design/pad_SAD_4by4_Reference.v
```

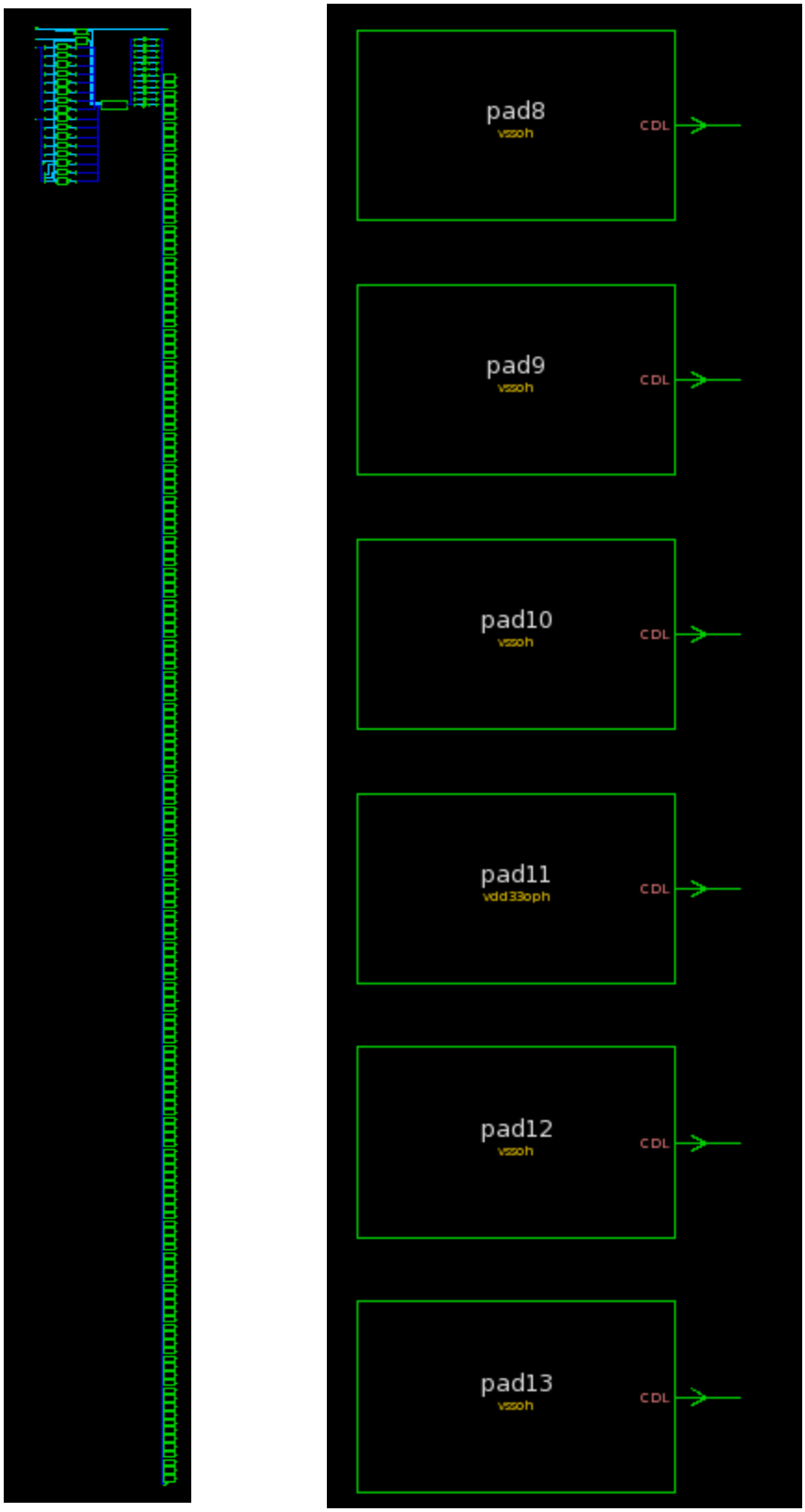
√ Read 명령어의 기능은 전편 매뉴얼이서 이미 설명했다. 이것은 HDL Compiler의 문법 구문 검사 및 GTECH 기반의 게이트 레벨로 변환시키는 Translation 작업을 수행한다.

√ 그렇다면 GTECH의 능력을 확인해 보자. 모든 파일을 read 했을 때 오류 메시지는 하나도 없어야 한다. 그림과 같이 붉은 색으로 오류의 위치까지 표시되니 꼼꼼히 살펴본 후 디버깅하기 바란다.

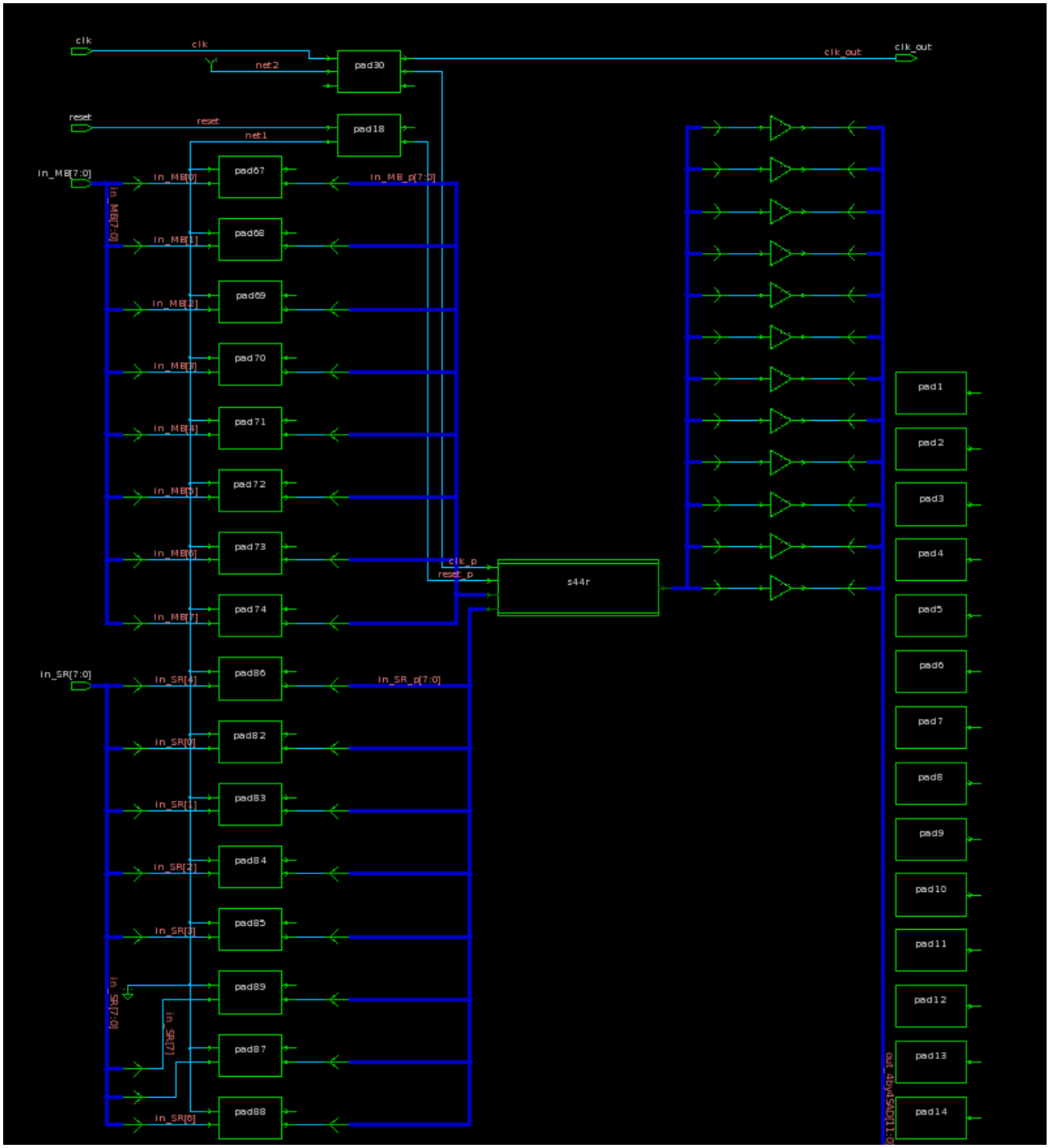
```
design_vision> read_verilog ./design/pad_SAD_4by4_Reference.v
Loading verilog file '/home/Dell/mschoi/dk_home/SAD4by4_Reference/syn/design/pad_SAD_4by4_Reference.v'
Detecting input file type automatically (-rtl or -netlist).
Running DC verilog reader
Reading with Presto HDL Compiler (equivalent to -rtl option).
Running PRESTO HDLC
Compiling source file /home/Dell/mschoi/dk_home/SAD4by4_Reference/syn/design/pad_SAD_4by4_Reference.v
Error:  /home/Dell/mschoi/dk_home/SAD4by4_Reference/syn/design/pad_SAD_4by4_Reference.v:60: Syntax error at or near
*** Presto compilation terminated with 1 errors. ***
Error: Can't read 'verilog' file '/home/Dell/mschoi/dk_home/SAD4by4_Reference/syn/design/pad_SAD_4by4_Reference.v'.
No designs were read
design_vision>
```

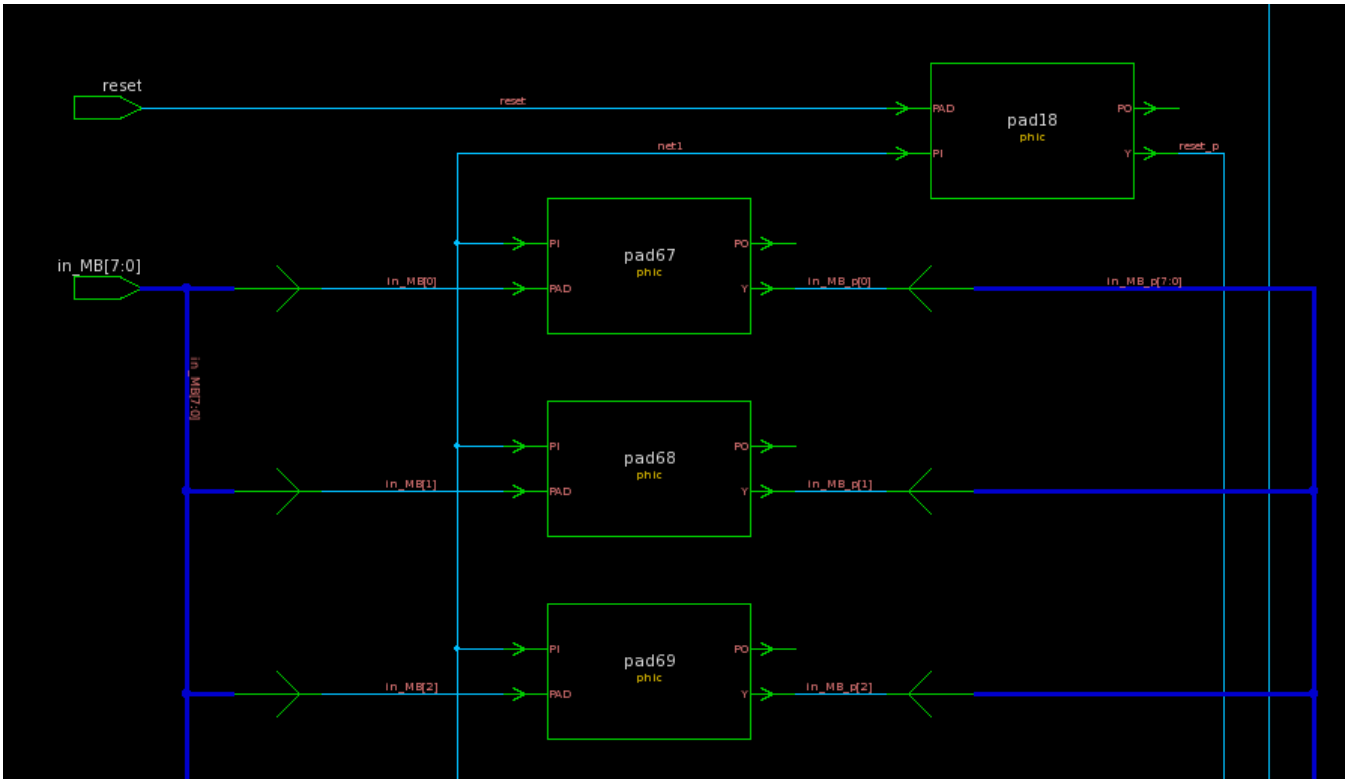
√ pad 파일의 60번 라인에 괄호 하나가 빠져있는 간단한 오류다. read 단계에서 수정하는 문법오류는 대부분 간단한 것이기 때문에 log message를 확인하며 차근차근 수정하기 바란다.

√ 아래 그림은 GTECH를 이용한 Translation 결과를 보여준다. 구성 요소들이 제대로 생성되었는지 하나하나 살펴보자. 그림이 참 구분하기 힘들겠지만 일단 설명을 하겠다. 제일 왼쪽 그림에서 길게 꼬리처럼 내려온 사각형 모양, 아무것도 연결 안 된 단순한 모듈이 POWER PAD다. Power line은 옵션에서 보이지 않게 숨겨 놨기 때문에 연결정보 없이 모듈만 덩그러니 나왔다. 잘못된 건이 아니니 오해하지 마라. 오른쪽에 확대한 그림을 첨부하겠다.


pad8
vssoh
CDL
pad9
vssoh
CDL
pad10
vssoh
CDL
pad11
vdd33oph
CDL
pad12
vssoh
CDL
pad13
vssoh
CDL

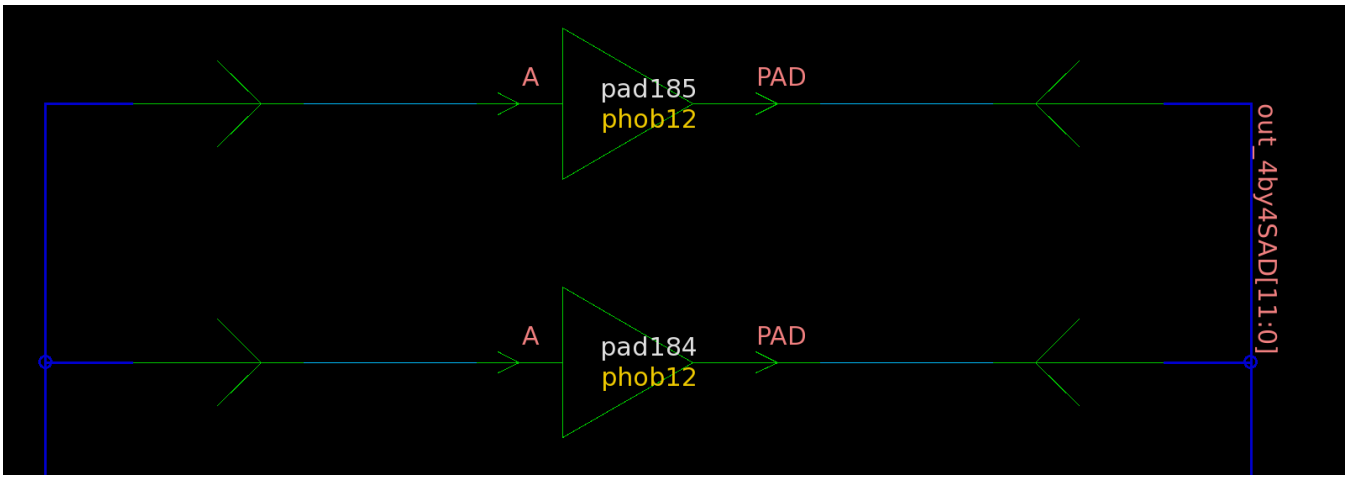
√ 그렇다면 이제 Signal PAD와 CLK PAD가 잘 붙었는지 확인해 보자. 가운데 Main Core를 기준으로 오른쪽에 다닥다닥 붙어있는, 이번에는 무언가 연결되어 있는 사각형 박스들이 보이나? 이것들이 Input buffer PAD다. 살짝 안쪽 라인에 두 개의 사각형은 각각 CLK, reset PAD이며 버퍼 모양의 심볼은 Output buffer PAD다. 각각 확대한 사진을 함께 첨부하겠다.





<Reset, Input buffer PAD>

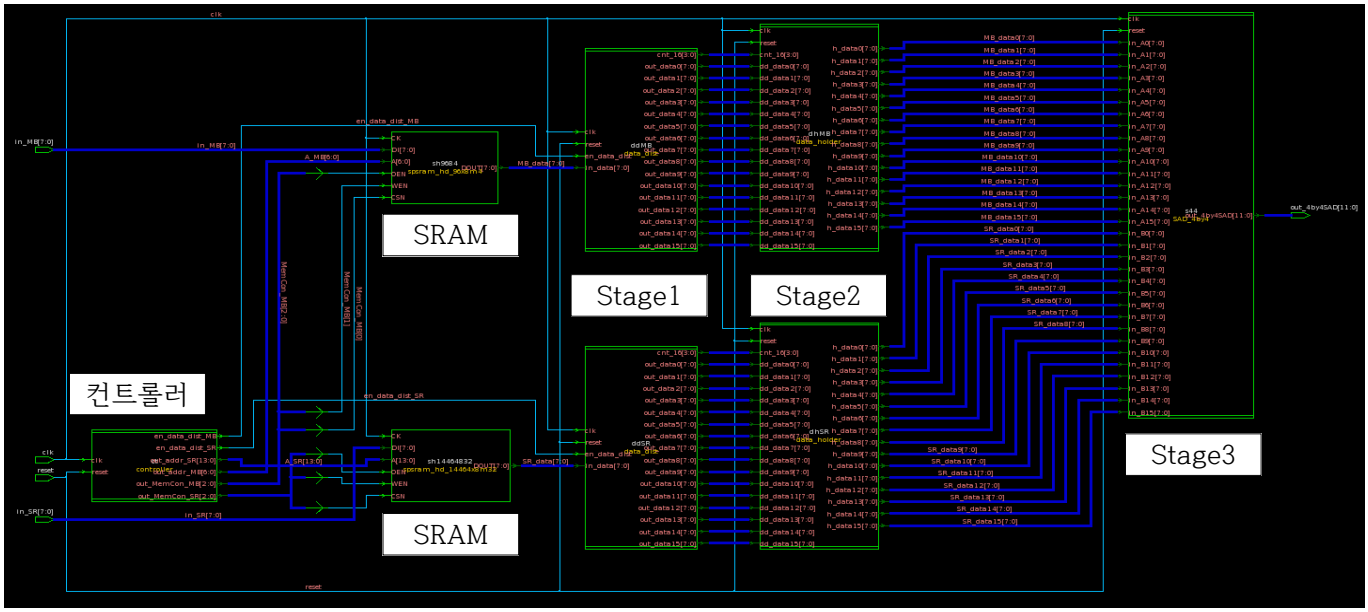
<CLK PAD>



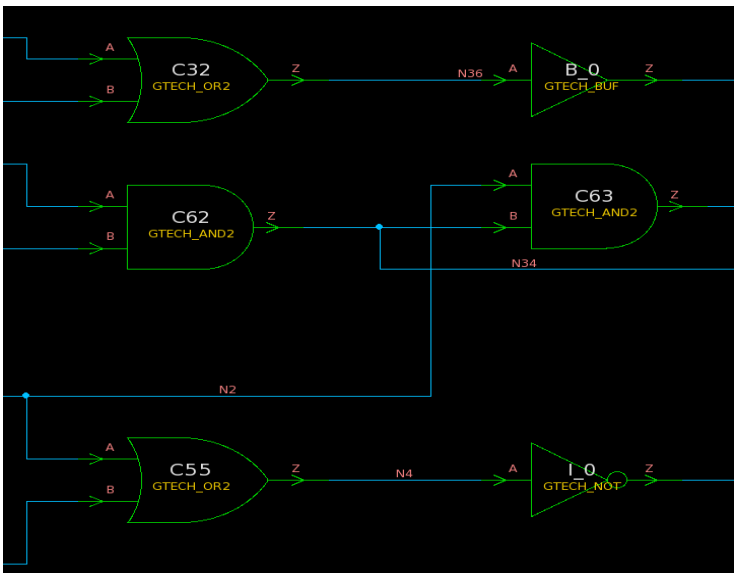
<Oouput buffer PAD>

√ CLK PAD는 두 개의 포트를 할당했다 (중요, 포트 넘버 밀려 쓰지 말 것). Output buffer PAD의 심볼은 왜 사각형이 아니고 버퍼 모양일까? 앞서 설명 했던 Output buffer PAD의 회로도를 확인해봐라. 고개가 끄덕여지나? 이로써 라이브러리에서 PAD를 정상적으로 불러왔음을 확인했다. 다음은 메인 코어의 내부를 파헤쳐보자.

√ 컨트롤러 2개의 HARD IP형태의 SRAM 3-stage 파이프라인 SAD 모듈까지 그럴싸한 그림이 나왔다. 하지만 아직까지 포장된 모듈만 보일 뿐 우리가 원하는 GTECH는 보이지 않는다. 더 깊숙이 들어가 보자. State1의 PE 모듈 내부로 들어가 보겠다. (참고로 HARD IP 형태의 SRAM은 박스 형태에서 더 이상의 접근이 불가능하다)



√ 드디어 나왔다. GTECH!!! 자신이 GTECH인 것을 알리고 싶어 명찰까지 부착한 게이트가 떡 하니 박힌 것 보니 제대로 출력된 것 같다. Read 과정은 이걸로 마친다.



√ Read 명령어는 analyze 및 elaborate 명령어의 기능을 함께 포함하고 있다. 이게 갑자기 무슨 소린가? 당황할 수도 있겠지만 애초에 HDL 파일을 읽어 GTECH로 Translation 하는 방법은 두 가지이다. 첫째가 앞서 실습했던 read_verilog 명령어를 사용하는 것이고 둘째가 analyze와 elaborate 명령어를 함께 사용하는 것이다. 사용하는 방법은 아래와 같다.

```
analyze -format verilog ./design/Controller.v
analyze -format verilog ./design/SAD_4by4_Reference.v
analyze -format verilog ./design/Stage1_PE_SAD.v
analyze -format verilog ./design/Stage2_RCA_8bits.v
analyze -format verilog ./design/Stage3_RCA_8bits.v
analyze -format verilog ./design/data_holder.v
analyze -format verilog ./design/SAD_4by4.v
analyze -format verilog ./design/data_dist.v
analyze -format verilog ./design/pad_SAD_4by4_Reference.v

elaborate pad_SAD_4by4_Reference  => File name (x), Design name (O)
```

√ analyze는 HDL을 읽고 문법 구문 에러를 체크하며 GTECH 변환 전 HDL의 중간 포맷 형식으로 HDL을 사용자가 정의한 특정 위치에 저장시켜 놓는다. (정의하지 않았다면 현재 작업 폴더에 파일들이 생성되는 것을 확인할 수 있을 것이다.)

√ 저장된 HDL 중간 포맷은 elaborate 명령어를 통해 GTECH로 변환한다. 이 경우 뒤에서 언급할 link 명령어는 자동으로 수행되어 진다.

√ 실습했던 Read 명령어는 analyze와 elaborate의 기능을 동시에 실행한다. 일타쌍피의 기능이지만 이 녀석은 뒤에 link라는 명령어를 반드시 사용해야 한다. VHDL의 경우 .mr 혹은 .st와 같은 중간 파일을 생성한 후 변환하지만 Verilog는 중간 파일의 생성 없이 변환을 수행한다.

√ 중간 파일의 생성을 원한다면 .synopsys_dc.setup 파일 내에 다음의 변수를 true 값으로 변경해라.
'hdlin_auto_save_templates ture'

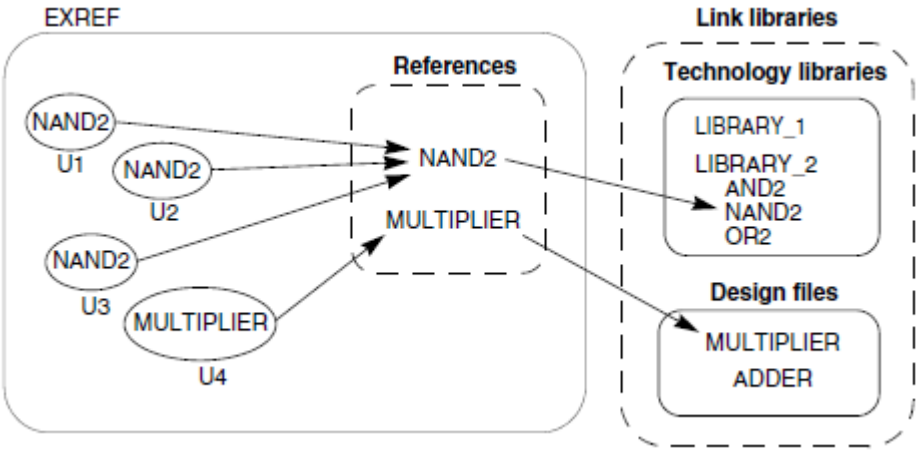
√ Read 작업이 끝나면 링크 명령어를 수행한다.

```
set top pad_SAD_4by4_Reference
current_design $top
link
```

√ link 명령은 link_library가 반드시 설정되어 있어야 사용할 수 있다. link의 근본적인 역할은 서로 다른 두 종류의 라이브러리의 기본적인 연결 정보를(Mapping) 생성하는 것이다.

√ 이것은 Target 라이브러리를 사용한 Optimization 및 Mapping과는 또 다른 개념이다. 여기서 의미하는 Mapping은 Optimization된 게이트와의 연결을 의미하며 Optimization은 GTECH가 link 명령어를 통해 연결된 Target 라이브러리의 기본 게이트를 사용자 요구에 맞춰 최적화된 게이트로 변환하는 것을 말한다. 즉, link는 GTECH의 기본 게이트를 Target 라이브러리의 기본 게이트와 1:1 Mapping 하는 과정을 의미한다.

√ 그림을 통해 다시 설명하도록 하겠다. HDL로 코딩된 NAND게이트와 MULTIPLIER는 Read 과정에서 GTECH로 변환된다. 여기서 사용된 GTECH는 그림의 References라 할 수 있다. 이것은 link라는 명령어를 통해 Technology libraries (여기서는 삼성 0.13um공정 라이브러리)의 동일한 게이트로 1:1 Mapping 되어진다. 이 후 사용자가 정한 Constraint에 따라 Compile 명령을 실행하면 link된 게이트와 netlist정보는 Optimization을 수행하고 최적화된 게이트로 다시 Mapping을 수행하게 되는 것이다.



√ link 명령어를 수행하기 전에 Current design의 계층의 가장 상위의 디자인으로 설정해야 한다.

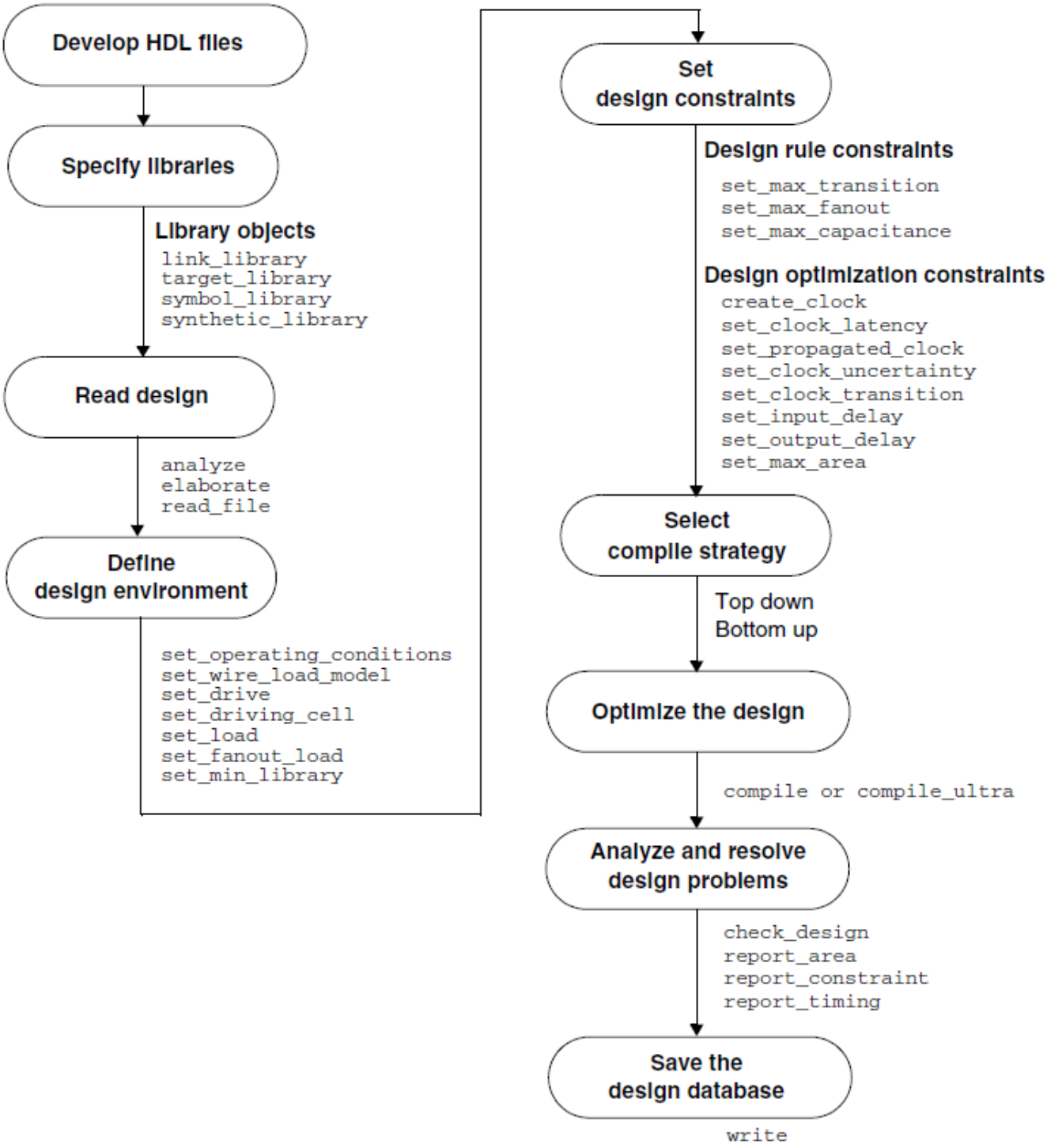
- set top pad_SAD_4by4_Reference
- current_design $top

√ link 명령을 수행한 결과는 아래와 같다. 해당 디자인과 연결된 라이브러리의 종류를 보여준다.

```
design_vision> link
  Linking design 'pad_SAD_4by4_Reference'
  Using the following designs and libraries:
  --------------------------------------------------------------------------
  * (9 designs)              /home/Dell/mschoi/dk_home/SAD4by4_Reference/syn/design/pad_SAD_4by4_Reference.db, etc
  std150e_typ_120_p025 (library) /home/Dell/mschoi/dk_home/samsung013/sec050915_0050_STD150E_regular_DK_Synopsys_N/sec150e_synopsys/syn/STD150E/std150e_typ_120_p025.db
  std150e_typ_120_p025_memory (library) /home/Dell/mschoi/dk_home/samsung013/sec050915_0050_STD150E_regular_DK_Synopsys_N/sec150e_synopsys/syn/STD150E/std150e_typ_120_p025_memory.db

1
```

√ 이제 우리가 수행한 작업이 어느 정도 까지 왔는지 점검해 보자. 아래 그림은 전체 합성을 위한 과정을 보여준다. 우리는 현재 Develop HDL files, Specify Libraries, Read design까지 완료한 상태다. 그리고 이번 매뉴얼에서는 CLK 정보만 Constraint으로 설정한 후 바로 Compile을 실행하고 결과를 확인할 것이다. 중간 과정의 Constraint은 다음 매뉴얼에서 자세하게 살펴본다.



√ 위의 그림에서는 DC에서 사용할 수 있는 모든 명령어와 Constraint을 언급하지는 않았지만 중요도가 높은 대표적인 것들의 파악은 가능하다. 지금부터 신경 쓰면 머리만 아프니까 이번 매뉴얼에서 필요한 것만 살펴보고 빨리 넘어가자.

√ create_clock 명령어를 사용하여 동작 주파수를 결정할 것이다. 물론 최적화된 클럭이 아닌 여유있는 클럭을 줄 것이다. CLK의 최적화는 다양한 종류의 Constraint을 적용하여 찾을 수 있으며 간혹 Constraint을 아무리 쥐어짜도 만족할 만한 속도가 나오지 않을 경우 HDL 설계로 돌아가 디자인 구조를 수정해야 하는 경우가 발생할 수도 있다.

√ CLK의 생성은 create_clock 명령어를 사용하여 적용할 수 있다. -period 옵션은 주기를 나타내며 동작 주파수를 설정할 수 있다. 단위는 ns이다. 현재 디자인에서는 20ns의 CLK period를 적용하였고 이것의 동작 주파수는 50MHz이다. -waveform은 duty cycle을 의미하며 50%로 설정하였다. Design에서 실제로 사용하는 CLK 포트명을 기입한다. 현재 디자인의 경우 clk를 포트명으로 사용한다.

```
create_clock -period 20 -waveform {0 10} [get_ports clk]
set_dont_touch_network [get_clocks clk]
```

√ 다시 이야기 하지만 이것은 초기 값일 뿐이다. 모든 Constraint 적용 후 타이밍의 Slack의 여유가 많다면 계속해서 고속의 동작 주파수로 변경을 시도할 것이다. (요구하는 동작 주파수를 만족했더라도 요구치보다 더 강한 netlist 생성을(확실한 동작 보증을 위함) 위해 한계치까지 계속 몰아라) 아마 많은 횟수의 합성을 요구하게 될 것이며 그 때마다 결과를 확인하고 조절하는 작업을 반복할 것이다.

√ 일반적으로 합성 단에서 동작 주파수는 약 110 ~ 120% 정도의 마진으로 합성하는 것이 일반적이다. 즉, 100MHz의 동작 주파수가 요구 사항이라면 실제 합성은 110MHz 또는 120MHz로 정의한다.

√ 단일 클럭이 입력될 경우 클럭 port에서 입력되는 clk 신호를 클럭으로 정의하면 되는데, 일반적으로 이런 경우보다 클럭 포트로부터 입력된 clk가 PLL과 같은 clock generation block을 만나 분주하여 공급되는 경우가 대부분일 것이다. 이 경우 PLL의 output 신호, 즉 첫 F/F으로 입력되는 clk를 클럭 신호로 정의해야 한다. PLL의 입력을 CLK로 정의할 경우 실제 분주되어 나오는 clk 신호들은 클럭 신호로 인식하지 못해 Fanout rule에 따라 클럭 라인에 버퍼가 추가되는 황당한 경험을 하게 될 것이다. 이러한 문제를 막기 위해 클럭에 set_dont_touch_network를 적용한다.

√ Multiple clk를 사용할 경우 각각의 클럭 신호마다 create_clock을 생성해야 한다. 이것은 Synchronous 및 Asynchronous multiple clock scheme로 구분할 수 있는데 false path를 적용하는 여부가 다르다. Constraint에서 더 자세하게 살펴보겠다.

√ 이제 compile 명령어를 사용하여 가장 기본적인 합성을 완료하자. 옵션에 관해서는 다음 매뉴얼에서 자세하게 다를 것이고 일단 합성을 시도하는 것에만 초점을 맞추자.

```
compile -boundary -only_design_rule
```

√ 완료된 합성 결과를 살펴보자. 이 또한 Constraint 매뉴얼에서 자세하게 다룰 예정이며 자세한 분석이 요구되는 작업이지만 여기서는 일단 기본적인 세 가지 결과(power, area, timing)를 확인하는 방법만 살펴보자.

아~ 이렇게 나왔구나! 정도만 확인하고 넘어가기 바란다. 자세한 분석 및 다양한 report 출력 방법은 다음 매뉴얼 및 PrimeTime 매뉴얼에서 설명 하도록 하겠다.

√ report_area 명령어를 사용하여 생성된 Area 정보를 확인해 보자

report_area

```
Number of ports:                          31
Number of nets:                           64
Number of cells:                         208
Number of combinational cells:           207
Number of sequential cells:                0
Number of macros:                          0
Number of buf/inv:                         0
Number of references:                      9

Combinational area:       9445.003001
Noncombinational area:    82883.999956
Net Interconnect area:       65.275591

Total cell area:          92329.002957
Total area:               92394.278547
1
```

√ report_power 명령어를 사용하여 생성된 power 정보를 확인해 보자

report_power

```
Global Operating Voltage = 1.2
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1mW


  Cell Internal Power  =  12.1162 mW   (91%)
  Net Switching Power  =   1.1612 mW    (9%)
                         ---------
Total Dynamic Power    =  13.2774 mW  (100%)

Cell Leakage Power     =  90.9130 uW


                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             10.1651            1.1358        2.2761e-02           11.3236  (  84.70%)
memory              1.4062        4.3905e-04        6.0196e-02            1.4668  (  10.97%)
black_box           0.0000            0.0000            0.0000            0.0000  (   0.00%)
clock_network       0.0000            0.0000            0.0000            0.0000  (   0.00%)
register            0.5179        4.6013e-03        2.0582e-03            0.5246  (   3.92%)
sequential      2.9312e-03        1.7770e-03        6.3559e-05        4.7718e-03  (   0.04%)
combinational   2.4048e-02        1.8611e-02        5.8336e-03        4.8492e-02  (   0.36%)
--------------------------------------------------------------------------------------------------
Total              12.1162 mW         1.1612 mW     9.0913e-02 mW        13.3683 mW
1
```

√ report_timing 명령어를 사용하여 생성된 timing 정보를 확인해 보자. -_-;; 20ns를 줬는데 17ns가 남았다. 너무 여유롭구나. Constraint 작업할 때 빡세게 합성해서 동작 속도를 높여야 할 것이다.

report_timing

```
Des/Clust/Port     Wire Load Model       Library
------------------------------------------------
pad_SAD_4by4_Reference
                   113_e_100k_41m        std150e_typ_120_p025
SAD_4by4_Reference 113_e_100k_41m        std150e_typ_120_p025
data_dist_1        113_e_5000_41m        std150e_typ_120_p025

Point                                                   Incr       Path
--------------------------------------------------------------------------
clock clk' (rise edge)                                 10.00      10.00
clock network delay (ideal)                             0.00      10.00
s44r/sh14464832/CK (spsram_hd_14464x8m32)               0.00      10.00 r
s44r/sh14464832/DOUT[6] (spsram_hd_14464x8m32)          2.36      12.36 f
s44r/ddSR/in_data[6] (data_dist_1)                      0.00      12.36 f
s44r/ddSR/U26/Y (ad2bd2_hd)                             0.23      12.58 f
s44r/ddSR/out_data3_reg[6]/D (fd1eqd1_hd)                0.00      12.58 f
data arrival time                                                  12.58

clock clk' (rise edge)                                 30.00      30.00
clock network delay (ideal)                             0.00      30.00
s44r/ddSR/out_data3_reg[6]/CK (fd1eqd1_hd)               0.00      30.00 r
library setup time                                     -0.25      29.75
data required time                                                 29.75
--------------------------------------------------------------------------
data required time                                                 29.75
data arrival time                                                 -12.58
--------------------------------------------------------------------------
slack (MET)                                                        17.17
```

√ 이번 매뉴얼에서는 최적화를 거의 사용하지 않는 기본적인 합성방법에 대해서 살펴봤다. 만약 실제 칩을 만드는 것이 아니라 연구용 합성결과 특히 Reference 모듈과 비교를 위한 합성 결과만이 필요하다면 지금까지의 작업만으로 충분하다. 최적화를 위한 Constraint의 적용은 오히려 잘못된 비교 결과를 생성할 수 있다.

√ 예를 들어 Reference A 디자인과 Proposed B 디자인의 성능을 비교한다고 가정하자. 두 디자인은 당연하겠지만 다른 구조다. 디자인 최적화는 HDL 구조에 따라 다르게 적용된다. 만약 A가 10의 값을 갖고 B가 8의 값을 갖는다고 가정하자. 여기에 어떤 Constraint을 동일하게 적용하여 최적화를 시도하게 되면 A는 3만큼의 최적화를 수행하여 10-3 = 7이 될 수 있고 B는 1만큼의 최적화만 적용되어 8-1 = 7이 될 수 있다. 이런 일이 발생할 수 있는 이유는 서로 다른 디자인 구조를 갖고 있고 최적화는 구조에 따라 다르게 적용되기 때문이다. 두 디자인은 같은 성능을 갖는다는 황당한 결과를 얻을 지도 모른다. 주의하기 바란다.

√ compile명령어로 합성이 완료되었다면 다음 명령어를 사용하여 세 종류의 결과(netlist, sdf, sdc)를 출력하자. netlist는 게이트 레벨로 합성된 결과이며 sdf, sdc는 각각 타이밍 정보 Constraint정보를 포함한다.

```
write -f verilog $top -h -o ./db/$top.v
write_sdf -version 1.0        ./db/$top.sdf
write_sdc                     ./db/$top.sdc
```

√ 지금까지 설명한 합성 명령어들을 사용하여 하나의 스크립트 파일을 생성할 수 있으며 매 합성마다 파라미터를 바꿔 Source 명령어를 사용하여 한 번에 실행 할 수 있다.