JTAG의 소개 및 원리

글쓴이: 유영창

K.E.L.P.

Korea Embedded Linux Project

Last Modified: 12/26/2002 16:51:49

Foreword

이 글은 K.E.L.P. (**Korea Embedded Linux Project -** 이하 **KELP**로 표기) **Website**에서 **"**유영창**"**님이 강의하신 **[**임베디드 강좌**]**중 **"JTAG**의 소개 및 원리**"**라는 내용의 강좌를 제가 **HTML Format**의 문서로 **Conversion** 작업을 한 글입니다. 원본이 **Text**문서인 관계로 모든 그림은 유영창님께서 직접 작업하신 **ASCII Art**로 되어있습니다. **(**일반 그림보다도 더 그림같은 환상적인 **ASCII Art**를 덤으로 감상하실수 있습니다...^^;**)**

KELP는 임베디드 리눅스 초보자들을 위해 문서로서 도움을 주고자 하는 곳입니다.

Copyright (c) 2001 유영창.
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

- Original Text 문서 작성: 유 영창 / HTML Conversion 작업: 양 창우 -

JTAG를 알아야 하는 이유편...

*등록: **2001-07-25 14:59:59***

음... 정말 정말 글쓰기 위한 맘을 먹기 위해서 세운 하얀 밤이 몇 날 몇일 인지 모르겠네요. 항상 쓴다 쓴다 하면서 이렇게 키보드 두드리기가 너무 힘들군요... 그렇지만 이렇게 펜**?** 아니 키보드를 들어**?** 아니 치기 시작했습니다.

우선 박철님에게 하사 받은 내용 중 **JTAG**라는 훌륭한 디버그 툴이 있어서 이렇게 소개합니다.

자 이 단계는 어떤 단계인가.... **i386**과 **StrongARM** 의 비교 중 어떤 단계일까요**?** 부트 매체에 부팅 프로그램과 커널을 써 넣는 단계입니다. **KELP**의 숙제 중에 부팅 디스크를 만드는 단계가 있습니다. 우리는 이 부팅 디스크를 만드는 단계에서 플로피 매체에 커널과 루트 이미지를 올리게 됩니다. 물론 플로피 매체를 처음 읽어 들이는 것은 바이오스에서 하지요.

그렇다면**?** 아사벳 보드에도 바이오스가 있나요**?** 물론 있습니다. 뭐 **BLOB**라는 프로그램이지요.

커널은 **BLOB**라는 프로그램에서 읽어 오겠네요**?** 맞습니다.

그럼 **BLOB**는 커널을 플로피에서 읽어 오나요**?** 으잉**?** ......

자... 그렇습니다. 아사벳 보드에서는 당연 플로피가 없습니다. 내부에 플레쉬 메모리가 있어서 커널은 이곳에 기록되게 끔 설계되어 있지요.. 또 **BLOB**라는 바이오스에 해당하는 프로그램도 롬으로 박혀 있는 것이 아니라서 이 플레쉬 메모리에 써야 합니다. 그럼.. 보드에서 플레쉬 메모리에 읽고 쓸 수 있도록 무엇인가가 있습니까**?** 당연**!** 있습니다. 그런데 그것은 플레쉬 메모리 전용이 아닙니다. 원래는 **StrongARM**의 코어 로직이나 주변 디바이스를 시험하기 위한 목적으로 만든 것이지요...

그 이름이 바로 그 유명한 **JTAG** 이고 우리는 이것을 **TAP**라는 포트를 이용하여 접근 할 수 있습니다.

자 여러분은 제가 이 강좌를 어떤 식으로 끌고 갈지 궁금하지요**? JTAG**에 대한 내용은 다음과 같은 순서로 이끌어 갈 겁니다.

- **JTAG** 의 소개 및 원리
- 아사벳 보드에서의 **JTAG**
- **JTAG**와 **PC** 와의 연결
- **JTAG** 를 이용한 플레쉬 메모리에 기록하는 방법 및 프로그램 예제..

뭐 이런 식일 겁니다. 물론 여러분은 제 강좌를 읽지 않아도 빠른 시간에 공부 할 수 있지요... **KELP** 자료실에서 **JTAG**에 관련한 내용을 제가 박철님에게서 받아서 올려 놓은 것이 있습니다. 그 중에 영문 매뉴얼이 있는데 그 내용을 다운 받으셔서 읽으셔도 됩니다. 뭐 영어도 그리 어려운 편은 아닙니다.

자 **i386**과 **StrongARM** 의 리눅스 비교 제 **2** 탄을 여기서 마무리 지지요.. 강좌 한편이 너무 길면 읽기 지루하겠죠.... 저도 타이핑하기에 손가락이 너무 아프죠...

참 항상 이야기하는 데요. 이 글의 지적 소유권 관계는 **GPL**을 따릅니다. 물론 상업적인 용도로는 사용할 수 없죠.... 그럼..

JTAG의 소개 및 원리 1편

등록: 2001-07-25 15:00:34

제가 JTAG를 소개하는 글을 올리기 이전에 투정 한번하죠... 솔찍히 하드웨어를 소개하는 글은 텍스트 에디터로 쓰기는 정말 힘들어요..

왜?

그림이 많이 들어가야 이해가 쏙쏙 되는데, 그 그림을 그리는 것이 그리 쉬운 것이 아니거든요... 그래서 말인데요... 이글의 내용에 JTAG관련 그림을 그리는 정성이 보통 정성이 아니라는 점 알아주세요.. 만약 안 알아주면 이제 저 그만 쓸렵니다. ^^;

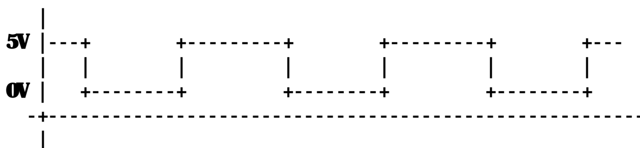
자 그럼 JTAG라는 것이 무엇일까요?

소개하기 이전에, 이글을 읽으시는 분은 하드웨어에 관련된 것을 아주 아주 쬐금 알고 있는 것이 편하실거예요.. 그럼 시작하죠...

제가 알고 있던 하드웨어 동네는요. 원시인 동네였던 것 같네요. 거의 수작업화 했던 디버그 방법을 가지고 있었던 것 같다는 것이 이 강좌를 쓸 시점의 생각입니다. 예전에는 (물론 지금도 이렇게 하시는 분들은 많습니다만) 보드 설계를 하고 PCB를 뜨고 IC를 납땜하고 이것을 테스트합니다. 순서는 이렇습니다.

우선 각 디바이스별 전원의 공급 선이 이상이 없는가를 확인합니다. 물론 테스터기나 점퍼 테스터를 이용하여 하나씩 핀들의 연결 관계를 시험 하지요... 여기서 단자들끼리 납땜이 잘 되어 있나, 전원 공급이 제대로 될 수 있나를 일일이 확인합니다. 전원 공급 배선들이 붙어 있거나 하면 디바이스에 손상이 생기기 때문이죠..

그 다음에 하드웨어 테스트용 프로그램을 하나 짜지요... 물론 아주 간단하게 LED나 기타 IO핀을 억세스해서 토글링하게 하죠. 토글링한다는 것은 이런 파형이 주기적으로 나타나게 하는 것이지요...



그래서 오실로스코프를 이용하여 관찰하는 것입니다. 만약 이런 파형이 나오지 않는다면? 오호~ 그것은 곧 하드웨어 설계자한테는 죽음의 시간이 시작된 겁니다. 일단 CPU가 살아야 뭘 하든 하지 않겠읍니까?

원 칩일 경우야 거의 외부 디바이스와 독립된 동작이 가능하기 때문에,

- 전원 공급선에 이상이 없는가...
- 오실레이터 와의 연결이 제대로 되어 있는가..
- 인터럽트 핀에 다른 신호가 들어가지 않는가.
- 리셋 핀에 이상이 없는가를 확인하면 되지만..

롬이나 램에 연결되어 동작되어야 하는 회로 구성이라면,

- 롬에 배선 연결이 제대로 되어 있는가?
- 롬에 칩셀렉트와의 연결이 제대로 되어 있는가?
- 어드레스 버스가 롬에 제대로 연결되어 있는가?
- 어드레스 디코더가 동작하는 가?
- 리드 라이트 단자에 제대로 신호가 들어 가는 하는 것을

알고 있던 경험과 지식을 바탕으로 이리저리 검사하게 되지요.. 물론 이것을 위해서 리셋 버튼이나 전원 스위치를 수도 없이 키고 끄게 됩니다.

이렇게 하드웨어의 연결이 이상없으면, 시험용 프로그램을 이리 저리 짜서 보드의 각 디바이스들을 시험합니다. 이때에 수많은 롬을 굽게 되거나 원칩을 굽게 되지요.. 돈 많은 회사에 있는 개발자라면 전용 개발툴을 사용해서 쉽게 할 수 있지만 그렇지 못한 회사가 더 많다는 것이 국내 산업의 현실이지요... 전 그래서 아직도 이렇게 시험하는 줄 알았습니다. 그런데 이건 **LSI**칩들을 조합하는 보드에서나 가능한 일이지만 회로가 **PLD**나 **CPLD**같은 대규모 칩에 내장되는 경우는 연결 핀을 일일이 찍어서 시험하기도 힘들고 (핀사이의 간격이 얼마나 조밀한지 여러분은 아시겠지요? 이곳에 오실로 스코프 핀 한번 밀어 넣어서 찍어보세요.. 귓구멍에 전봇대 넣는 것이 더 쉬울 지도 모르죠...) 내부 회로의 연결 관계들을 검사하기도 힘듭니다.

자 이런 고민은 저희 나라만 했겠읍니까? 우리나라 보다 손가락이 길고 두꺼운 미국사람들은 우리보다 고역이 더 심하면 심했지 쉽지는 않았을 겁니다. 역시 필요는 발명의 어머니라고... 개발자의 게으름은 기어코 **JTAG**라는 것을 만들게 되지요...

여기서 **JTAG**라는 것이 무슨 약자인지 궁금하지 않으십니까?

JTAG는 **Joint Test Access Group**의 약자입니다.

이게 한글로는 무슨 뜻인지 전 잘 모르겠습니다. 그런데 웹상에서 **JTAG**라는 키워드로 글을 찾는 것보다는 **Boundary-Scan**이라는 키워드로 글을 찾는 것이 쉽습니다. 일반적으로 **JTAG**란 말보다 **Boundary-Scan**이란 말이 더 많이 사용되고 있죠....

그럼 영어를 잘 못하는 저의 입장에서 **Boundary-Scan**이라는 뜻을 해석해보죠.. 뭐 직역하면 **"주변을 훑어본다"**는 것이 됩니다. 이말이 **JTAG**를 설명하는 말이 처음이자 끝이 됩니다.

이 내용을 구체적으로 살펴보기 이전에 **Boundary-Scan**의 역사를 살펴보면

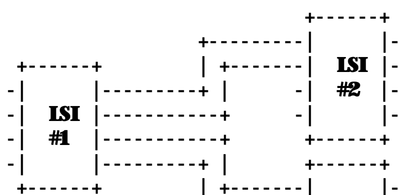
> 1980년대 후반의 **JTAG**라는 곳에서 연구 중이던 **Boundary-scan** 설계를 **IEEE**에서 **1990**년에 표준화하였고 **IEEE std 1149.1**가 제정되었습니다.

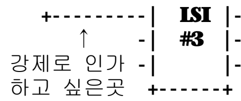
영문서에 보면 이렇게 되어 있습니다.

> IEEE Standard 1149.1-1990 "Test Access Port and Boundary-Scan Architecture," available from the IEEE, 445 Hoes Lane,PO Box 1331, Piscataway, New Jersey 08855-1331,USA

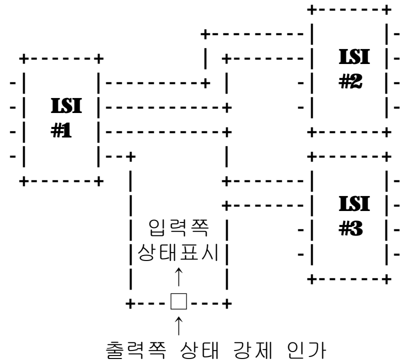
자 그러면 **Boundary-Scan** 이라는 뜻을 좀더 구체적으로 살펴 봅시다.

개발자가 하드웨어를 테스트 하다 보면 문제의 원인을 알아내기 위해서 특정 단자의 상태를 임으로 설정해 줄 필요가 생깁니다. 그런데 각 디바이스는 **PCB**상태에서 이미 다른 디바이스와 연결되어 있어서 강제로 인가하려면 핀의 연결을 칼 같은 것으로 끊고 강제로 인가하는 수밖에 없습니다. 그런데 여러 조건을 시험하려면 이렇게 하는 것이 그리 쉽지 않죠.... **CPU**칩에 그 수많은 배선을 끊고 있고 하는 것이 쉽겠읍니까? 현미경이나 갖다 놓고 해야 하지 않을까요? 잠깐 그림으로 볼까요?





여기서 우린 **LSI #3**번에 입력되고 있는 신호가 이상한지 **LSI**의 디바이스 칩이 이상한지가 궁금한 겁니다. 시스템이 동작 중에는 이런 행위를 할 수 없죠**?** 그렇다면 이곳에 이런 장치가 있다고 가정합시다.



이런 장치를 붙이면 **LSI#1**에서 나온 출력 신호를 볼수도 있고 **LSI#3**쪽에 입력되는 신호를 개발자 임의대로 넣을 수 있으면 굳이 배선을 끊지도 않아도 되고 좋겠죠**?** 문제는 이런 회로를 일일이 집어 넣는 것도 역시 문제고 그 자체가 문제를 일으킬 소지가 있다는 것입니다.

그래서 절충안으로 나온 것이 **CPLD**와 같은 대규모 **LSI**에 넣으려는 코어 로직에 아예 위 그림같은 기능을 하는 로직을 집어 넣어 버리는 것입니다. 그러면 물론 **CPLD**에 넣을 수 있는 용량의 일부분을 낭비하는 결과를 초래할 수는 있지만 그에 비해 하드웨어 체크를 하기 위해 소모하는 시간에 비하면 절대 아까운것이 아닐지도 모르죠...

그러면 모든 로직 연결마다 다 하느냐**?** 물론 그러면 아깝죠.. 개발자 경험상 입력과 출력 쪽만 이런 장치를 하면 대부분의 문제가 해결된다는 것을 알죠... 그래서 주변에만 위와 같은 회로를 추가하는 것입니다. 그래서 **Boundary**라는 말이 붙은것이지요...

그러면 **Scan**은 왜 붙을까요**?** 우선 윗 그림에 입력 쪽 상태를 표시하고 출력 쪽 상태를 강제로 인가할 수 있는 기능을 작는 로직의 이름을 한번 붙여 봅시다. 그 이름하여 **"Boundary-Scan Cell"** 이라고요 .....

화장실을 잠깐 가야 하는 이유로 인하여 일단 **3**탄은 여기서 접죠.

참 항상 이야기하는 데요. 이 글의 지적 소유권 관계는 **GPL**을 따릅니다. 물론 상업적인 용도로는 사용할 수 없죠.... 그럼..

JTAG의 소개 및 원리 2편

등록: 2001-07-25 15:01:03

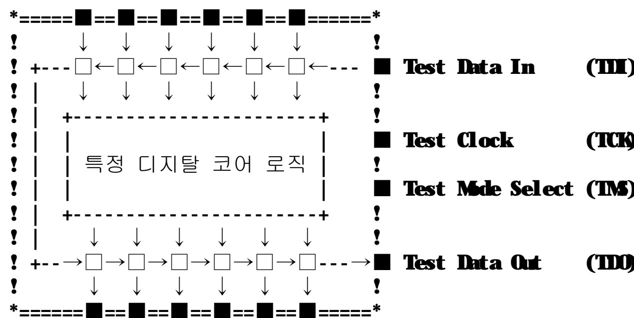
1 탄에서 소개한대로 **JTAG**이 만들어 진 이유는 조금 아셨을 겁니다.

자자 이전 편에서 **Boundary-Scan Cell** 이라는 것이 생겼습니다. 하지만 이것은 아직 완전한 모습이 아니죠... **3**탄에서 그린 그림에는 문제가 있습니다. 만약 **LSI** 단자가 한 **10**개 짜리가 있는데 한 단자에 **Boundary-Scan Cell**을 하나씩 단다면.. 새로 생긴 핀이 **2**개씩 더 생기죠... 그러면 총 **30**개의 측정 핀이 생깁니다. 이걸 이용해서 디버깅을 한다? 한번 최악의 그림을 볼까요?

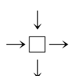
....

아이구 도저히 그림을 그릴수 없네요.... 그정도로 귀찮아 진다는 이야기입니다. 그러면 **180**개 핀이 달린 칩에 하나씩 붙이면? 음... 상상하기 싫군요.... 이걸 해결하는 방법은? 디버깅용 핀은 최소한이 좋지 않겠습니까? 그래서 **JTAG**아저씨들은 기가 막힌 방법을 생각해 냅니다. 그것이 무엇이냐... **Boundary-Scan Cell**을 시프트 레지스터 형식으로 만드는 겁니다. 혹시 시프트 레지스터가 뭐냐고 물으시는 분들은 잠깐 강좌를 읽는 것을 중지하시고... 하드웨어 로직에 관련된 개론책을 잠깐 읽고 오실래요... 정중한 부탁입니다.

Boundary-Scan의 기본 아키텍처는 다음과 같습니다. 물론 이것은 원리적인 것입니다.



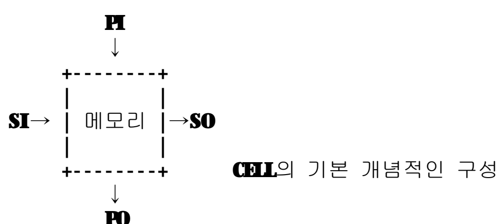
우후 정말 그리기 힘들다....



이 그림상 이것이 바로 **Boundary-Scan Cell** 이라는 것을 알겠죠? 만약 그렇게 보이지 않는다면 텍스트 표현의 한계라고 밖에 전 이야기 할수 없군요.... (^^;)

이렇게 **Cell**들이 직렬로 쭉 연결되게 하면 테스트 핀들에 대한 문제가 해결되지요... 역시 양놈들 머리 좋습니다. 뭐 우리나라 개발자라면 금방 만들었겠지만요... 당연히 쉬프트 레지스터라면 쉬프트를 하기 위한 클락이 필요하지요..

자자.. 일단 오른쪽에 있는 놈들의 정체가 곧 밝혀질려고 합니다. 이전에 **Cell**의 모습을 좀더 확대하여 알아 봅시다. **Cell**을 약간 만 확대 하면 이런 문구들을 볼수 있습니다.



CELL의 기본 개념적인 구성

여기서,

- **PI** 란 말은 **Parallel Input**의 약자로 외부 신호 입력 또는 디지탈 로직에서 나오는 출력을 **Cell**입장에서는 입력으로 볼수 있는 겁니다.
- **PO** 란 말은 **Parallel Output**의 약자로 외부 신호 출력 또는 디지탈 로직에 인가되는 입력을 **Cell**입장에서는 출력으로 볼수 있는 겁니다.
- **SI** 라는 것은 **Scan Input**으로 직렬 입력 단자입니다.
- **SO** 라는 것은 **Scan Output**로 직렬 출력 단자 입니다.

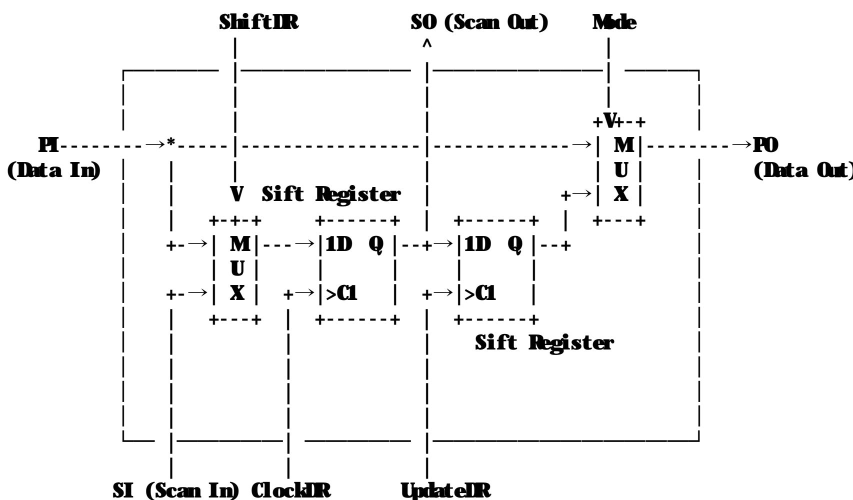
자 **CELL**은 어떤 기능을 가지고 있을까요?

- **PI**의 신호를 캡쳐하는 기능이 있읍니다. **PI**에 들어 오는 신호를 메모리에 기억시키는 기능이 있습니다.
- 메모리에 있는 내용을 **PO**에 인가하는 기능이 있습니다.
- **PI**의 신호를 **PO**에 바로 넘기는 기능이 있읍니다. 이때는 **PI**의 신호가 바로 **PO**에 전달됩니다. 보통 디지탈 로직이 사용될때 이용되겠죠?
- **SI**의 입력을 메모리에 기억시키고 메모리의 내용을 **SO**에 옮겨오는 기능이 있읍니다. 이 기능을 이용하여 각 **CELL**의 내용을 외부에서 읽어 볼수 있고 특정 상태를 내부에 인가를 할수 있는 것입니다.

이런 기능은 **CELL**외부에 있는 레지스터에 의해 제어를 받게 되고 이것에 대해서는 이 글 이후에 설명할 것입니다. 이 내용이 좀 난해하거든요... 개념은 쉬운데 말이죠...

그럼 이 **CELL**을 좀더 확대하여 볼까요?

윽....... 엄두가 안나는 군요... 이 복잡한것을 어떻게 그리죠? 내참..... (고민고민)



이 그림 인간 승리죠?

자 확대해 보니, 원래 보던 단자보다 더 많이 생겼죠? 모두 무엇에 쓰는 것일까요? 우리모두 알아 맞춰 봅시다. 일단 **PI** 와 **PO SO SI**를 빼면, **SiftDR**, **ClockDR**, **UpateDR**, **Mode**가 남죠? 일단 위의 회로를 보고 위에서 논의되었던 **CELL**의 기능에 대하여 알아 봅시다.

문제와 답식으로 볼까요?

문제 **1)** **PI**의 신호를 캡쳐하는 기능이 있읍니다. **PI**에 들어 오는 신호를 메모리에 기억시키는 기능이 있읍니다.

답 **1)** 이것은 **ShiftIR**을 **0**으로 하고
ClockIR에 한 클럭을 주면 됩니다.
물론 이 값은 **SO**에 인가된 상태가 되겠죠..

문제 **2)** 메모리에 있는 내용을 **PO**에 인가하는 기능이 있읍니다.

답 **2) Mode** 가 **1**이고
UpdateDR에 한 클럭을 주면 됩니다.

문제 **3) PI**의 신호를 **PO**에 바로 넘기는 기능이 있읍니다.
이때는 **PI**의 신호가 바로 **PO**에 전달됩니다.
보통 디지탈 로직이 사용될때 이용되겠죠?

답 **3) Mode**가 **0**이면 됩니다.
이것이 정상적인 동작입니다.

문제 **4) SI**의 입력을 메모리에 기억시키고 메모리의 내용을 **SO**에 옮겨오는 기능이
있읍니다.
이 기능을 이용하여 각 **CELL**의 내용을 외부에서 읽어 볼수 있고
특정 상태를 내부에 인가를 할수 있는 것입니다.

답 **4)** 문제 **1**번 동작을 하고
CELL의 갯수 만큼 **ClockDR**을 인가하고
이때 원하는 입력 상태를 **CELL** 갯수만큼 **ClockDR**인가시에 공급하고
문제 **2**번을 수행하면 되겠죠...

자자 .. 여기까지가 **Boundary-Scan Cell**의 동작 방식입니다.

여기까지 이해하셨다면 한가지 의문이 생길겁니다. 그것이 무엇일까요**?** 외부 단자에는 **SiftDR**, **ClockDR**, **UpateDR**, **Mode**가 없거든요**?** 이게 무척 궁금하실겁니다. 아님 할수 없고 ^^;

그림을 그리는데 내공을 너무 심하게 낭비해서 잠시 쉬어야 겠네요....

참 항상 이야기하는 데요. 이 글의 지적 소유권 관계는 **GPL**을 따릅니다. 물론 상업적인 용도로는 사용할수 없죠.... 그럼..

JTAG의 소개 및 원리 3편

등록: 2001-07-25 15:01:37

4탄에서 여러분은 **Boundary-Scan Cell**의 벌거 벗은 그 흉직한 모습을 보셨을 겁니다. 이 로직을 이용하여 **JTAG**의 디버그 방법론이 나옵니다. 우선 이전 내용을 말로 정리해 봅시다.

- **JTAG**의 핵심은 **Boundary-Scan Cell**이며 **Boundary-Scan Cell**은 쉬프트 레지스터 형식을 가진다.
- 각 셀의 상태를 읽어 오거나 써 넣는 것은 시리얼 입출력 방식을 이용한다.
- **Boundary-Scan Cell**은 디바이스의 입출력단에만 있다.

뭐 이정도의 정리면 되겠지요... 여러분은 사실 이용만 할것이기 때문에 내부의 동작원리는 몰라도 됩니다.

이젠 접근을 외부에서 해봅시다. 내부에서의 구조와 외부에서의 접근을 통하여 중간 계층에 무엇이 있는지를 알수 있겠죠**?** 여러분이 스토롱암의 매뉴얼을 보면 **JTAG**에 대한 내용이 나옵니다. 그 내용을 보면 **JTAG**는 **CPU**의 코어가 들어있는 **CPLD**의 외부 단자중 딱 **5**개의 단자만을 사용합니다. 이 단자에는 어떤 것이 있을까요**?**

- **TDI**
- **TDO**
- **TMS**
- **TCK**
- **TRST**

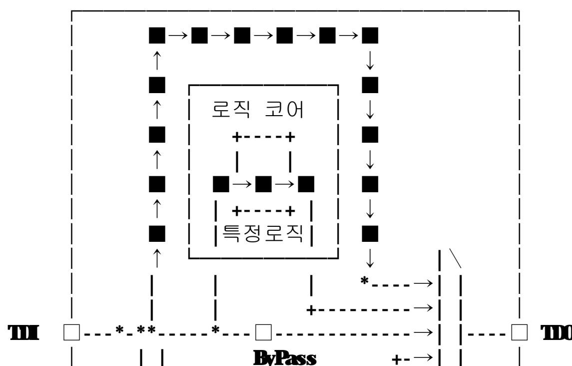
이 단자들을 묶어서 **TAP**라고 합니다. **TAP**를 풀어쓰면 **Test Access Port**라고 합니다. **JTAG**의 **TAP**단자명들은 이전 강좌에서 소개한 **Boundary-Scan Cell**에서 사용되는 핀의 이름과는사뭇 다른 이름을 사용하고 있읍니다. 이것은 **JTAG**에는 **Boundary-Scan Cell**이외에 무언가가 있다는 것이지요. 우리는 이 정체를 밝혀 내야 합니다. 물론 제가 밝혀 드리겠지요 ^^;

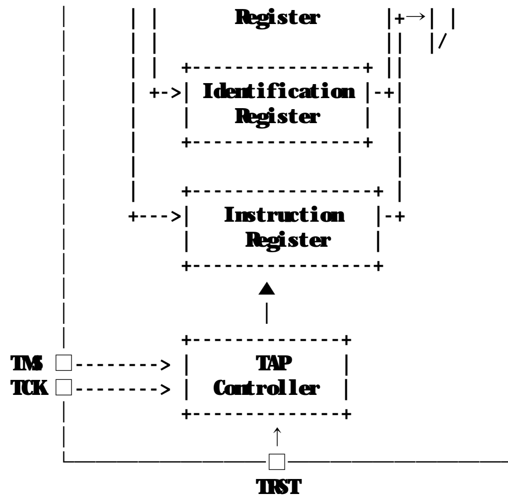
우선 위에 써 놓은 각 단자명을 영문으로 풀어내어 봅시다.

- **TDI** : Test Data In
- **TDO** : Test Data Out
- **TMS** : Test Mode Select
- **TCK** : Test Clock
- **TRST** : Test Reset

이런 이름들의 약자들입니다.

자 그럼 이제 **JTAG**의 진짜 내부 모습을 보여드리지요.. 그런데 이 그림을 제가 어떻게 그릴수 있을까요... 흑흑 **(**텍스트가 미워**)** 참 이그림은 약간 약화 시킨 그림입니다. 스트롱 암에서 작동되는 자세한 동작 방식에 대한 설명을 위한 그림은 나중에 다시 그리지요...





무슨 그림인지 알아 보시겠어요?. 이것이 **JTAG**의 완성된 블록도 입니다. 여기서 ■처럼 표현한 것은 **Boundary-Scan Cell**을 의미합니다. 아직까지는 **Boundary-Scan Cell**과 **TAP**의 관계가 불분명해 보이죠. 이 비밀은 **Boundary-Scan Cell**이외에 각종 **Register**에 해답이 있습니다.

하나씩 하나씩 비밀을 벗겨 볼까요**?**

이궁 벌써 저녁 **10**시 이군요... 오늘은 이만 퇴근해야 할것 같네요... 조만간 바로 다음 편을 올리겠읍니다. 사실 **JTAG**의 진 면목은 여기서 부터 입니다.

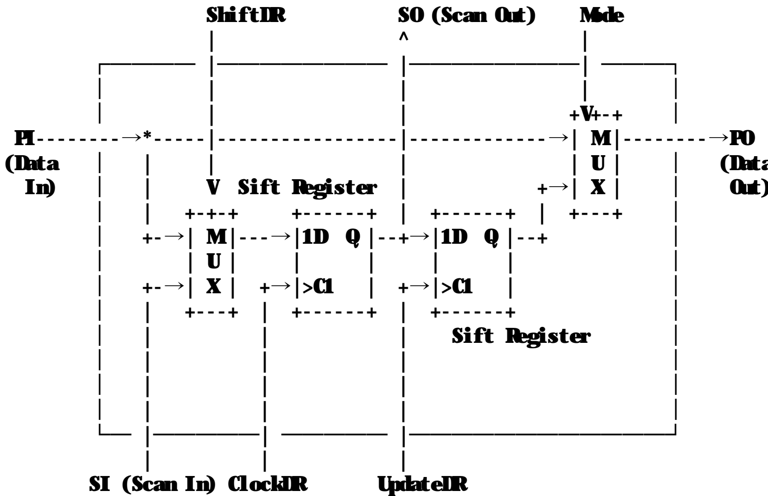
참 항상 이야기하는 데요. 이 글의 지적 소유권 관계는 **GPL**을 따릅니다. 물론 상업적인 용도로는 사용할수 없죠.... 그럼..

JTAG의 소개 및 원리 4편

등록: 2001-07-25 15:02:06

이제 우리는 JTAG의 진짜 동작 원리를 알아야 합니다. 이 강좌 이후가 JTAG를 어떻게 사용할수 있는 지를 알 수 있죠. 다시 한번 정리해 보죠

Boundary-Scan Cell은 다음과 같은 내부적 연결 단자가 있읍니다. (강좌 4탄 그림 참조)



PI	Parallel Input
PO	Parallel Output
SI	Scan Input
SO	Scan Output
SiftDR	PI 단자로 입력된 신호를 Sift Register로 인가할것인지 아니면 PO쪽으로 인가할 것인지를 결정합니다.
ClockDR	쉬프트 레지스터로 인가된 PI신호를 랫치하여 기억 시켜 놓을지를 결정합니다.
UpateDR	쉬프트 레지스터로 기억된 상태를 PO쪽에 인가시킬 것인지를 결정합니다.
Mode	쉬프트 레지스터에서 나온 출력을 PO에 인가 할것인지 아니면 PI쪽에서 온 출력을 PO에 인가 할 것인지를 결정합니다.

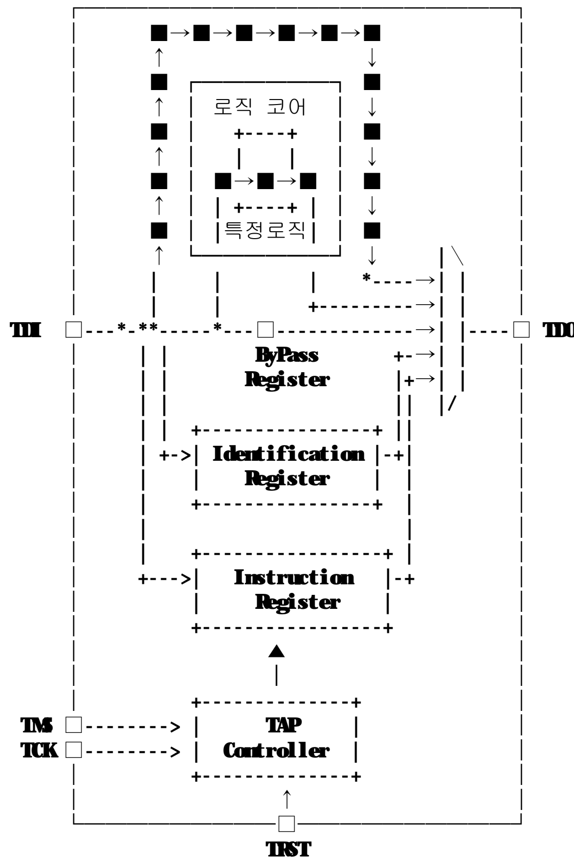
외부에는 TAP라고 하는 단자가 있읍니다. 이단자는 다음과 같이 정의 되어 있습니다.

TDI	Test Data In
TDO	Test Data Out
TMS	Test Mode Select
TCK	Test Clock
TRST	Test Reset

여러분은 여기서 이상한 점을 알 것입니다.

Boundary-Scan Cell과 TAP 사이의 단자들은 서로 매칭이 되지 않는다는 것 입니다. 그렇다면? 여기에는 무언가가 숨겨져 있읍니다. 이것이 무엇일까요?...

그것이 강좌 5탄에서 그렸던 JTAG 블럭도에 표시되어 있는 TAPC (TAP Controller) 라는 것이 숨겨진 바로 그놈입니다. 이 TAPC는 TMS와 TCK의 제어를 받습니다. 그리고 TAPC의 단자들과 Boundary-Scan Cell의 단자들(SiftDR, ClockDR, UpateDR, Mode 단자)과 연결되어 있는 것이지요. JTAG의 내부에는 Boundary-Scan Cell 이외에 여러가지가 내장되어 있읍니다. 우선 강좌 5탄에 그렸던 그림을 다시 그려보죠...



이그림 자주 이용될겁니다. 꼭 꼭 기억 해주세요..

그림을 보면 제가 아직 설명하지 않은 것들이 있죠? 그것이 무엇인지를 이제는 설명할 때가 된것 같습니다. 가장 하단에 TAP Controller이것은 무엇일까요? 이 놈이 JTAG의 핵심이라고 할수 있죠 일면 TAPC라고 합니다. 이놈은 나머지를 모두 제어하죠... 그래서 이놈은 다음에 논의하기로 합시다.

Instruction Register	이놈은 내부에 4비트로 구성되어 있읍니다. 한글로 직역하면 명령 기억 장소 인데, 이놈에 의해서 TDI와 TDO가 어디에 연결될지를 결정합니다.
Identification Register	이놈은 총 32비트로 구성되어 있죠.. JTAG를 사용하는 장치(디바이스)에 대한 정보가 기억되어 있읍니다. 이 레지스터를 이용하여 해당 칩셋이 어떤 것인지를 알아낼수 있읍니다. 제가 JTAG의 동작을 설명할때 가장 먼저 접근할 놈이 이놈입니다.
ByPass Register	이놈은 말 그대로 TDI를 TDO로 바로 연결시켜 버리는 놈입니다. 물론 TCK를 한클럭 소모하기는 하지만요. 이것은 여러 CPLD가 연결되어 있을때 접근 속도를 효율적으로 하기 위한 놈이죠... 물론 나중에 설명할 겁니다.

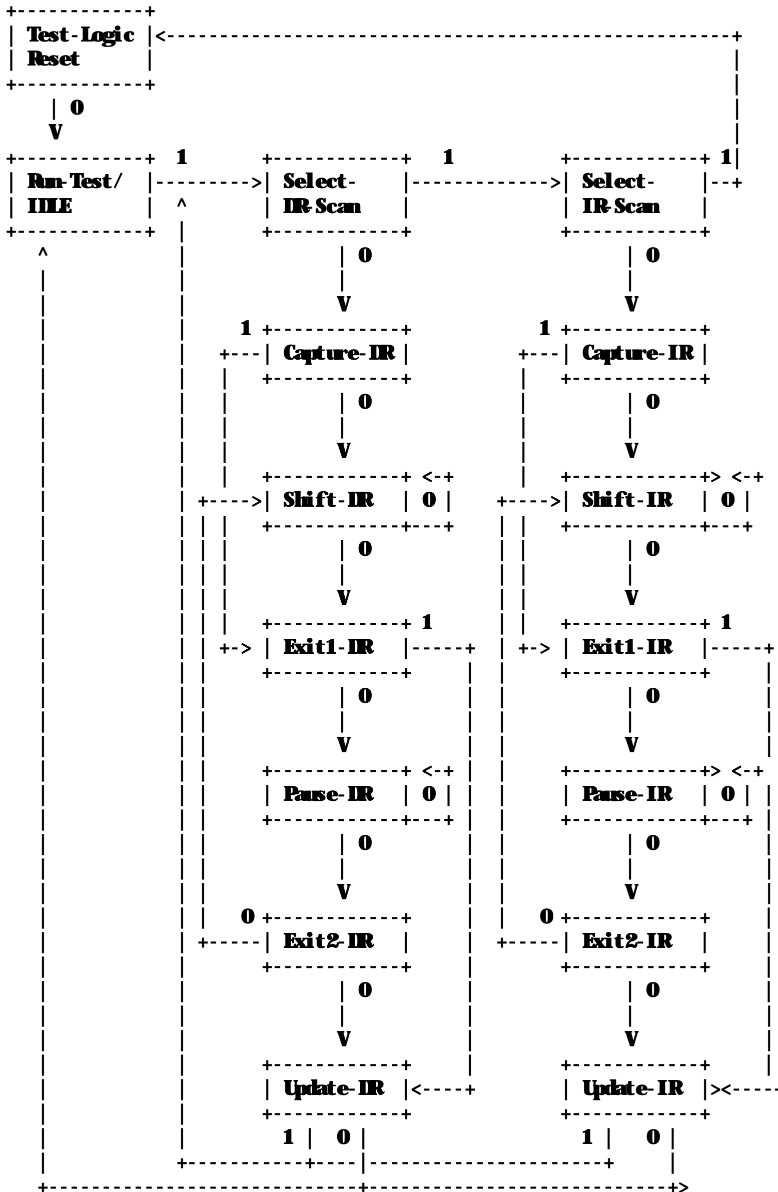
그 다음이 Boundary-Scan Cell Register죠. 이것이 우리가 그동안 접근 공부한 놈이죠. 물론 JTAG가 궁극적으로 접근하려는 놈이기도 하죠...

이젠 TAPC를 설명할때가 되었네요... TAPC를 모르면 JTAG를 모르는 것과 진배 없읍니다. 이놈을 이해해야만 하므로 여러분은 정신 바짝 차리고 잘 읽어야 합니다. TAPC는 TMS와 TCK를 이용하여 제어가 되는데 제가 TAPC의 동작방식을 이해한후 감탄을 안할수가 없었읍니다. 우선 이것을 이해하기 전에 여러분은 KELP의 자료실에서 JTAG 시뮬레이션 프로그램을 다운 받으신후에 실행하신후, 이 강좌를 계속 읽어 주시기를 부탁

드립니다. 그래야 이해가 빨리 될수 있거든요...

TAPC는 내부에 자기 상태를 갖습니다. 이 상태에 따라 무엇을 할것인가를 결정합니다. 우선 상태도를 그려 볼까요...

저에 고생은 또 시작 될것 같네요....



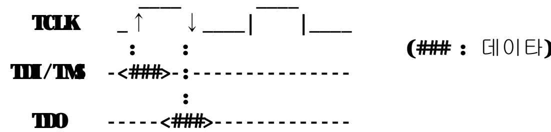
다 그렸네요... 푸하하하하....

자 이것이 **TAPC**가 가질수 있는 모든 상태도 입니다. 상태도라는 것이 무엇이냐**?** 음... 이걸 제가 설명하기에는 시간 낭비라서 그냥 진행하지요...

우선 하드웨어적으로 **JTAP**를 리셋 시키려면, **TRST**단자를 **LOW**로 떨어뜨리고 약간의 시간이 지나면 **TAPC**와 그와 관련된 모든 상태가 초기화 됩니다. 초기화가 되면 **TAPC**는 **Test-Logic Reset** 상태가 됩니다. 자 여기서 **TMS**단자와 **TCLK**단자에 신호를 주면 **TAPC**의 상태가 변화가 생깁니다.

우선 어떻게 신호를 주면 되는 지 보죠...

JTAG의 신호 다이어 그램은 다음과 같습니다.



그림에 보듯이 **TDI**나 **TMS**를 원하는 상태로 인가한후, **TCLK**를 **LOW** 상태에서 **HIGH**로 올리면 해당 상태가 **JTAG**내부로 인가되지요. **JTAG**내부에 있는 **TDO**를 외부에 인가하려면 **TCLK**를 **HIGH**에서 **LOW**로 인가하면 됩니다. 그러므로 **TCLK**가 한클럭 즉 **LOW**에서 **HIGH**로 다시 **LOW**로 되면, **TDI**와 **TMS**값이 내부로 적용되고 **TDO**값이 외부로 배출되는 것입니다. 저는 이것을 **TAP Accept**싸이클이라고 부르겠읍니다.(이건 제가 임으로 붙인 이름이므로 정식명칭이 아님을 밝힙니다.) 이 동작으로 **TAP**의 모든 동작을 제어하게 됩니다. 외부에서 연결 시켜서 동작 시키는 방식은 간단하지요?

자 그럼 위 상태도에서 **TAPC**가 **Test-Logic Reset** 상태로 되어 있다고 합시다. 이 상태를 **Capure-IR**상태까지로 이동시키려면 어떻게 해야 할까요.. **TAPC**의 상태는 **TDI**값과는 무관합니다. 단지 **TMS**와 **TCLK**를 조작하면 상태를 이동시킬수 있읍니다.

우리 한번 해보죠.. 이거 할때 우리는 단순 무식해져야 합니다. ^^;

위 그림에서 **Test-Logic Reset**에서 **Capure-IR**까지의 상태를 이동시키려면 다음과 같은 상태로 변해가야 합니다.

1. Test-Logic Reset
2. Run-Test/Idle
3. Select-DR-Scan
4. Select-IR-Scan
5. Capure-IR

자 그럼 미로추적을 시작해보죠...

- **1**상태에서 **2**상태로 갈려면, **TMS**를 **0(LOW)**를 주고 한클럭 주면 됩니다. **(TAP Accept** 싸이클**)** 여기서 한클럭 준다는 것은 **TCLK**를 **"Low → High → Low"**로 한다는 것이죠.
- **2**상태에서 **3**상태로 갈려면, **TMS**를 **1(HIGH)**를 주고 한클럭 주면 됩니다.
- **3**상태에서 **4**상태로 갈려면, **TMS**를 **1(HIGH)**를 주고 한클럭 주면 됩니다.
- **4**상태에서 **5**상태로 갈려면, **TMS**를 **0(LOW)**를 주고 한클럭 주면 됩니다.

뭐 쉽죠? 그럼 **Capure-IR**에서 **Run-Test/Idle**로 이동하려면 어떻게 하여야 할까요?

1. Capture-IR
2. Exit1-IR
3. Update-IR
4. Run-Test/Idle

이렇게 가면 됩니다. (가장 최단 코스죠 ^^)

- **1**상태에서 **2**상태로 갈려면, **TMS**를 **1(HIGH)**를 주고 한클럭 주면 됩니다.
- **2**상태에서 **3**상태로 갈려면, **TMS**를 **1(HIGH)**를 주고 한클럭 주면 됩니다.
- **3**상태에서 **4**상태로 갈려면, **TMS**를 **1(HIGH)**를 주고 한클럭 주면 됩니다.
- **4**상태에서 **5**상태로 갈려면, **TMS**를 **0(LOW)**를 주고 한클럭 주면 됩니다.

이렇게 상태를 변화 시키면 됩니다. 각각의 상태는 **TAPC**가 무언가를 하게 됩니다. 여기서 중요한것은 **TAPC**가 어떤 상태로 되어 있든 **Test-Logic-Reset**상태로 이동하려면, **TMS**를 **1(HIGH)**로 주고 **6**클럭만 주면 됩니다. 그러면 어떤 경우이든 **Test-Logic-Reset**이 됩니다. 한번 직접 한번 미로 테스트를 해보세요... 맞나 틀리나...

자 우선 각 **TAPC**의 상태에 대하여 알아 봅시다. 각 상태가 하는 것이 무엇을 하는 것인지... 대부분의 상태는 무언가를 수행하는 기능은 없읍니다. 단지 원하는 상태까지 도달하기 위한 중간 상태역활이 대부분이죠. 각 상태에 대한 설명을 간단하게 하죠...

Test-Logic-Reset	TAPC 및 JTAG와 관련된 모든 내용을 초기화 합니다. 동시에 상태 시작점이기도 합니다.
Run-Test/Idle	JTAG를 동작상태로 진입시키고 TAP를 통하여 적용한 내용이 각 장비에 영향을 끼치게 합니다.
Select DR-Scan	명령 레지스터에 의해 선택된 **Boundary-Scan Cell**에 제어를 할것인가 아닌가에 대한 중간 상태 값입니다. 상태 변이용 중간 상태이지 특별히 하는 기능은 없읍니다.
Capture DR	명령 레지스터에 의해 선택된 **Boundary-Scan Cell**의 **PI**단자 값을 내부 시프트 레지스터쪽으로 적용되게 합니다. 즉 현재 상태를 쉬프트할수 있게 준비하는 기능입니다.
Shift DR	명령 레지스터에 의해 선택된 **Boundary-Scan Cell**의 내부 값을 **SO**에 출력 시키고 **SI**값을 내부에 적용할수 있게 합니다. 이 상태로 있을때 **TCLK**값을 한 클럭 줄때마다 **TDI**값이 명령 레지스터에 의해 선택된 **Boundary-Scan Cell**의 **SI**에 연결되고 **SO**의 값이 **TDO**에 연결됩니다.
Exit1 DR	상태 변이용 중간 상태이지 특별히 하는 기능은 없읍니다.
Pause DR	상태 변이용 중간 상태이지 특별히 하는 기능은 없읍니다.
Exit2 DR	상태 변이용 중간 상태이지 특별히 하는 기능은 없읍니다.
Update DR	명령 레지스터에 의해 선택된 **Boundary-Scan Cell**의 **PO**단자 값에 내부 시프트 레지스터쪽의 내용을 적용시킵니다.
Select IR-Scan	명령 레지스터에 제어를 할것인가 하지 않을것인지에 대한 중간 상태 값입니다. 상태 변이용 중간 상태이지 특별히 하는 기능은 없읍니다.
Capture IR	명령 레지스터의 **Boundary-Scan Cell**의 **PI**단자 값을 내부 시프트 레지스터쪽으로 적용되게 합니다. 즉 현재 상태를 쉬프트할수 있게 준비하는 기능입니다.
Shift IR	명령 레지스터의**Boundary-Scan Cell**의 내부 값을 **SO**에 출력 시키고 **SI**값을 내부에 적용할수 있게 합니다. 이 상태로 있을때 **TCLK**값을 한 클럭 줄때마다 **TDI**값이 명령 레지스터 **Boundary-Scan Cell**의 **SI**에 연결되고 **SO**의 값이 **TDO**에 연결됩니다.
Exit1 IR	상태 변이용 중간 상태이지 특별히 하는 기능은 없읍니다.
Pause IR	상태 변이용 중간 상태이지 특별히 하는 기능은 없읍니다.
Exit2 IR	상태 변이용 중간 상태이지 특별히 하는 기능은 없읍니다.
Update IR	명령 레지스터에 의해 선택된 **Boundary-Scan Cell**의 **PO**단자 값에 내부 시프트 레지스터쪽의 내용을 적용시킵니다.

에궁 이번 강좌는 여기서 쫑을 보아야 겠네요.. 지가 약속시간이 다되서요.... 잉

참 항상 이야기하는 데요. 이 글의 지적 소유권 관계는 **GPL**을 따릅니다. 물론 상업적인 용도로는 사용할수 없죠.... 그럼..

JTAG의 소개 및 원리 5편

등록: 2001-07-25 15:02:47

에궁 **6**탄에서 설명한 내용중에서요... **Instruction Register**가 **4**비트로 구성되어 있다고 했는데요. 이게 **4**비트가 원래 규격이라고 책에는 되어 있는데, **SA1110** 같은 경우에는 **5**비트로 되어 있더라구요. 참고 하세요....

이전 **6**탄에서 우리는 **TAPC**의 상태 전이 방법을 알았잖아요... 이제 좀더 구체적으로 **JTAG**를 제어해 보죠. 이번 강좌에서는 **IDCODE**를 내용을 읽어 오는 방법을 알아보죠... **CPLD**의 내부 로직을 넣을때 **JTAG** 를 포함시켰다면요.. 일반적으로 **CPLD**의 정보를 **JTAG**의 **ID CODE**에 담아 놓는다고 합니다. 물론 **ID CODE**를 집어 넣지 않은 **DEVICE**도 있지만 대부분 집어 넣습니다. **ID CODE**를 담고 있는 **Identification Register**는 총 **32** 비트로 구성되어 있죠.. 이놈을 그림으로 그리면 다음과 같습니다.

비트 번호	31 - 27	26 - 12	11 - 1	0
내용	버전 번호	부품 번호	제조 회사	IDCODE 존재 유무

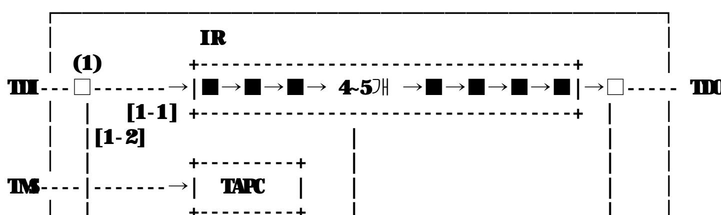
이중 제조회사는 **JEDEC106-A** 코드로 압축되어 있다고 책에는 써 있는데, 도대체 무슨 말인지는 모르지만 각 **DEVICE**의 제조 회사를 표기하는 것 같습니다.

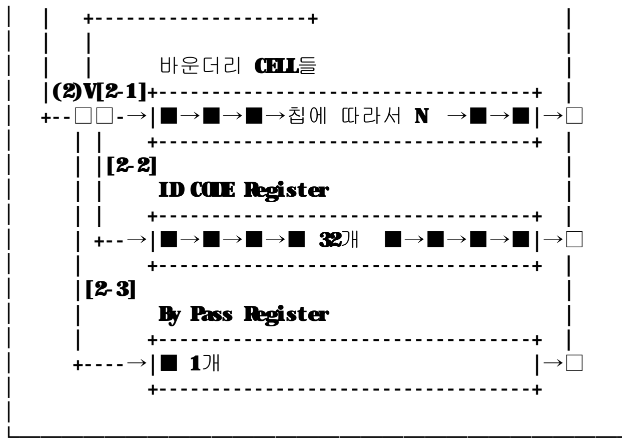
LSB 즉 **0**번 비트는요. 해당 **DEVICE**가 **ID CODE**를 담고 있는 **Identification Register** 를 가지고 있는지에 대한 유무를 나타 냅니다. 만약 레지스터가 있으면 항상 **1**로 되고요. 그렇지 않으면 **0**입니다. 만약 이값이 **0**이면 **TAPC** 초기화 상태이후로 바로 **ByPass** 레지스터 상태가 됩니다. 이전 강좌에 **TAPC**가 **Test-Logic Reset** 상태로 되어 있으면 **DEVICE**가 초기화 된다고 말씀드린 것 기억 나시는 지 모르겠지만 한가지 아셔야 할 것은, **Test-Logic Reset**상태에서는 **Instruction Register**가 **Identification Register** 를 가르키게 됩니다. 만약 **Identification Register**가 없다면 **ByPass Register**를 가리키고요... 물론 저는 직접 실험해 보지는 않았읍니다. 책에 그렇게 설명되어있죠.

참 이 강좌를 쓰고 있는 시점에 전 박철님 회사에서 일요일에 **JTAG**를 실습할수 있는 영광을 가졌읍니다. 제가 직접 **JTAG**를 억세스 하는 프로그램을 하나 작성해서 직접 **SA1110**과 **50100** 칩을 제어해 보았읍니다. 정말 신기하더군요... 예전에 보드 테스트 해보려면 일일이 테스트용 프로그램을 짜야 했는데 그냥 일반 프로그램으로도 칩 제어가 되기 때문에 그럴 필요가 없었읍니다. 이 **JTAG**를 이용하여 플레쉬 메모리에 직접 데이타를 기록하거나 읽어 들일수도 있답니다.

지금은 윈도우 환경에서 델파이로 코딩했는데요.. 이글을 쓰는 시점에 카이릭스가 나왔다고 합니다. 곧 구해서 설치하면 바로 리눅스용으로 만들 생각입니다. 물론 쏘스는 공개 할겁니다. **(**그래야 우리나라 국력이 강해질것 같아서요 ^^;**)** 완성된 후에 사용하실분은 사용해 보세요... 허접하나만 박철님이 쓸만하다고 하시는 군요.... 하하.... 아마도 쏘스 공개는 **3**월 초순이 될것 같네요.... 에궁 이야기가 많이 옆길로 세어 나갔네요...

자 다시 **JTAG**의 **ID CODE**를 읽어 오는 방법에 대하여 이야기 하죠... 우선 외부 단자 **TDI**, **TMS**, **TCK**, **TDO** 만을 이용하여 어떻게 저 **ID CODE**를 가져 올까요**?** 이것을 설명하기 전에 **JTAG**의 내부 모습을 다시 한번 그려 보죠... 물론 이것은 설명을 편하게 하기 위하여 약간 개념적으로 그린 것입니다. 참**!** 지금부터 **IR** 은 **Instruction Register**의 준말입니다.





□ 표시는 라인이 어디로 갈지 분기되는 것을 표기한 겁니다. **(1)** 표시는 **TAPC**의 상태에 영향을 받게 됩니다. **Shift-DR** 이면 **TDI**가 **[1-2]**로 연결되고요. **Shift-IR** 이면 **TDI**가 **[1-1]**로 연결됩니다. **(2)** 표시는 **IR**의 상태에 영향을 받게 됩니다.

차근 차근 위 그림을 보면서 설명을 드리겠읍니다. 우선 중요한것은 **IR**값 입니다. **IR**이 하는 중요한 것은 **TDI**와 **TDO**를 디바이스 내부에 있는 바운더리 셀중 어떤 것을 처리할것인가를 결정하는 중요한 역활을 합니다. 이 **IR**은 원칙적으로 **4**비트로 구성되는데 표준안은 다음과 같습니다.

B3 B2 B1 B0	내용
1 1 1 1 **BYPASS**	**TDI**와 **TDO**를 **ByPass** 레지스터에 연결합니다. **ByPass**레지스터란 놈은 **JTAG**의 효율을 높이기 위해 고안해 놓은 것입니다. 만약 시험하고자 하는 **CPLD**가 여러개가 있다고 합시다. 만약 각 **CPLD**가 **128**개의 **CELL**들을 갖고 있다면 우리는 전체를 시험하기 위해서는 **128** 곱하기 **3**번을 쉬프트 시키는 작업을 하여야 합니다. 이 얼마나 낭비겠읍니까? 그래서 측정 대상이 되지 않는 **CPLD**를 **TDI**와 **TDO**에 **ByPass**레지스터에 연결 시켜서 한번 쉬프트 시키는 행위만을 하게 하는 거죠. 그렇게 되면 **CPLD**가 **3**개 있더라도 한 **CPLD**만 시험한다면 **128 + 2**번만 쉬프트 시키면 원하는 디바이스의 바운더리 셀의 내용을 얻거나 설정할수 있게 하는 것이죠. 이명령은 **JTAG**를 지원하는 디바이스에는 필수적으로 갖추어야할 명령입니다.

B3 B2 B1 B0	내용
0 0 1 0 **IDCODE**	**TDI**와 **TDO**를 **ID CODE**를 갖는 **ID** 레지스터에 연결합니다. 디바이스의 정보를 얻는데 사용합니다. 이명령은 **JTAG**를 지원하는 디바이스에는 선택적으로 갖출수 있는 명령입니다. 경우에 따라 없을수도 있다는 말이지요..

B3 B2 B1 B0	내용
0 0 0 1 **SAMPLE**	**TDI**와 **TDO**를 바운더리 셀에 연결합니다. 그러나 시스템의 동작에는 전혀 영향을 끼치지 않습니다. 이 기능은 시스템의 상태를 실시간으로 감시하기 위한 기능으로 만이 사용합니다. 이명령은 **JTAG**를 지원하는 디바이스에는 필수적으로 갖추어야할 명령입니다.

B3 B2 B1 B0	내용
1 0 0 1 **INTEST**	**TDI**와 **TDO**를 바운더리 셀에 연결합니다. 그러나 외부 핀에 영향을 끼치지 않습니다. 이 기능은 디바이스 내부의 동작 상태를 시험하기 위한 기능으로 만이 사용합니다.

보통 CPLD의 로직을 시험할때 외부에 영향을 끼치지
않고 내부의 동작만을 동작시켜서 시험해야 할 경우가
있읍니다.
이 기능은 이때 사용하는 겁니다.

이명령은 JTAG를 지원하는 디바이스에는 선택적으로
갖출수 있는 명령입니다.
경우에 따라 없을수도 있다는 말이지요..

```
B3 B2 B1 B0     내용
-----------     -----------------------------------------------
0  0  0  0      TDI와 TDO를 바운더리 셀에 연결합니다.
                그러나 내부 로직에는 영향을 끼치지 않고
EXTEST          외부 핀에만 영향을 끼칩니다.
                이 기능은 디바이스 외부 즉 보드상에 디바이스에 연결된
                디바이스의 동작 상태를 시험하기 위한
                기능으로 사용합니다.
                가장 많이 사용되는 일반적인 기능입니다.

                이명령은 JTAG를 지원하는 디바이스에는 필수적으로
                갖추어야할 명령입니다.

B3 B2 B1 B0     내용
-----------     -----------------------------------------------
1  0  0  0      모든 외부 핀들의 출력상태를 하이
HIGHZ           임피던스 상태로 만듭니다.

                이 기능은 디바이스 외부 즉 보드상에 디바이스에 연결된
                디바이스의 동작 상태를 시험하기 위한
                기능으로 사용합니다.

                EXTEST와 다른점은 디바이스의 JTAG기능을 이용하여
                외부 핀들의 상태를 설정하는 것이 아니라,
                시험 지그와 같은 장비에서 시험하는 방식처럼
                외부에서 직접 신호를 주고 그 응답 특성을 알아보기 위한
                기능이죠.

                보통 5V전원을 출력 단자에 직접 인가해도 디바이스에
                손상을 주지 않게 됩니다.

                왜? 디바이스 외부와 내부를 하이 임피던스 상태로
                만듬으로써 완전히 단절 시켜 버리거든요...

                이명령은 JTAG를 지원하는 디바이스에는 선택적으로
                갖출수 있는 명령입니다.
                경우에 따라 없을수도 있다는 말이지요..
```

그외에 디바이스 업체에 따라 다른 상태를 넣어서 기능을 부여 할수 있읍니다. 이 기능들은 각 디바이스 매뉴얼을 참조하면 됩니다. 전 가난한 관계로 그런 디바이스는 구경을 못했읍니다. (^^;) 아까도 말씀드렸지만 **50100**보드나 **Intel386Ex**같은 디바이스들은 **4**비트만 사용하는데 스트롱암은 **5**비트를 사용하더군요... 여러분은 각 디바이스의 매뉴얼중 **JTAG**부분에서 이 부분을 꼼꼼히 읽고 사용하십시요... **JTAG**가 전혀 동작하지 않을수도 있거든요 이건 실전 경험에서 얻은 겁니다.

자 우리는 **IR** 레지스터가 무엇을 하는 놈인지 알았읍니다. 그럼 우리에 목적인 **IDCODE**를 어떻게 읽어 올수 있을까요 우선 순서를 적어 보겠읍니다.

1. **IR** 레지스터에 **IDCODE** 명령 패턴을 입력한다.
2. **IDCODE**의 내용을 읽어온다.

뭐 쉽죠? 그럼 이게 다냐? 푸하하하 아니죠. 이제 이것을 좀더 분해해 봐야죠.

자 **"IR 레지스터에 IDCODE 명령 패턴을 입력한다."**를 어떻게 해야 하나요.... 이건 이런 순서로 해야 합니다.

1. **TAPC**상태를 초기화 한다.
2. **TAPC**상태를 **Shift-IR**상태로 한다.
3. **IDCODE** 명령 패턴을 밀어 넣는다.
4. **TAPC**상태를 **Run-Test/Idle**상태로 만든다.

다시 말하지만 쉽죠? (약올리나...) 우린 위에 처럼 하기 위해서 TDI, TMS, TCK를 이용합니다. 이전 강좌를 기억하시나요? 그럼 시작합니다. (TAPC 상태도를 참고하시고, 시뮬레이션을 돌려 보세요)

(1) TAPC상태를 초기화 한다.

TMS를 1(HIGH) 상태로
TCK를 6클럭을 줍니다.

```
TMS  1 1 1 1 1 1 1 1 1 1 1 1
TCK  0 1 0 1 0 1 0 1 0 1 0 1
```

이 상태가 되면 TACP는 Test-Logic-Reset상태가 됩니다.
원칙적으로 이 상태이면 IR 레지스터는 IDCODE 명령이
적용된다고 책에는 있읍니다. (전 안해 봤어요.. 흑..)

(2) TAPC상태를 Shift-IR상태로 한다.

(1)상태에서

```
TMS  0 0
TCK  0 1    ; Run-Test/Idle 상태

TMS  1 1
TCK  0 1    ; Select DR-SCan 상태

TMS  1 1
TCK  0 1    ; Select IR-SCan 상태

TMS  0 0
TCK  0 1    ; Capture IR 상태
```

이상태면 IR 내용이 쉬프트되서 TDO로
빠져 나올 준비가 됩니다.

```
TMS  0 0
TCK  0 1    ; Shift IR 상태
```

이상태면 IR 레지스터와 TDI입력이 연결되고
TDO출력이 연결됩니다.

(3) IDCODE 명령 패턴을 밀어 넣는다.

자 이상태면 TDI -> IR 레지스터 -> TDO 와 같은 연결 상태가
됩니다.

이때 TMS를 0으로 주면 TAPC는 계속 Shift-IR 상태를
유지하게 됩니다.

문제는 여기에서 IDCODE는 B0부터 B3순으로 입력 시켜야
한다는 것입니다.

다시 말하면 꺼꾸로 집어 넣어야 하는 거죠
바운더리 스캔 셀은 반대로 되어 있어요...

자 집어 넣어 봅시다.

IDCODE 비트 패턴은 0010이죠 그럼 집어 넣는 순서는 0100이겠죠?

(2)상태에서

```
TDI  0 0
TCK  0 1    ; B3가 입력됩니다.

TDI  1 1
TCK  0 1    ; B2가 입력됩니다.

TDI  0 0
TCK  0 1    ; B1가 입력됩니다.
```

여기서 고만...
왜?

이거 이거 이부분이 무척 중요합니다.

TMS와 TDI는 TCK의 업 엣지에 영향을 받는다는 사실을 아시죠?

즉 TCK 가 0에서 1로 변할때 TMS와 TDI가 JTAG내부로 인가됩니다.

우린 패턴을 밀어넣은후에 TAPC상태를 Run-Test/Idle상태로 만들기로

예정되어 있죠?

그렇다면 **TMS**를 변화시켜야 하는데
이때 **TDI**상태도 그대로 밀려 들어가 버립니다.
그래서 **TAPC**의 상태를 **Shhift-IR**상태에서 **Exit1-IR**상태로
바꿀때 **TDI**도 같이 싫어서 **TCK**를 **0**에서 **1**로 바꾸어야 하는 거죠.

(4) TAPC상태를 **Run-Test/Idle**상태로 만든다.

자 **(3)**에서 마지막에 지적한 내용을 기억하시고

```
TDI  0 0
TMS  1 1
TCK  0 1    ; Exit1 - IR 상태
              동시에 마지막 B4값 TDI에 인가

TMS  1 1
TCK  0 1    ; Update-IR 상태
              우리가 쉬프트한 값이 이때 IR 레지스터에
              적용됩니다.

TMS  0 0
TCK  0 1    ; Run-Test/Idle 상태

              주의 할것은 Update-IR상태에서
              Run-Test/Idle 상태로 이동해야지
              만약 Test-Logic-Reset상태로 이동해 버리면
              여짓껏 한 행위가 모두 도로나무아비타불입니다. ^^;
```

에궁 이번 강좌는 여기서 쫑을 보아야 겠네요.. 지금 시간이 귀신이 나오는 00시네요. 집에 가야 할것 같아요. 잉~

참 항상 이야기하는 데요. 이 글의 지적 소유권 관계는 GPL을 따릅니다. 물론 상업적인 용도로는 사용할수 없죠.... 그럼..

JTAG의 소개 및 원리 6편

등록: 2001-07-25 15:03:23

집에 와서 이 글을 쓸 줄은 몰랐네요. 7탄 쓰고 집에 오니 2시인데... 잠이 안오고 7탄을 쓰다 말아 찜찜해서 마무리를 지으려고 키보드에 손가락을 올려 놓았읍니다. 사실 JTAG에 대하여 박철님에게 5시간 강좌 듣고 이 글을 쓸때만해도 그리 오래 걸리지 않을 것으로 알았는데, 이거 이거이 정말... 이렇게 많은 내용이 될줄은 상상도 못했네요... 물론 자료를 약간 더 찾아서 궁금했던 내용을 더 추가 했지만요... 일단 7탄의 내용이 이어집니다. 따라라라란~~~~

어디까지 했더라? 맞아! IDCODE 명령 패턴을 IR 레지스터에 집어 넣는것까지 했구만... 그럼 이젠 IDCODE 값을 얻어야 겠지요?

2) IDCODE의 내용을 읽어온다.

이거 역시 이전 내용과 거의 유사합니다. 그럼 어떻게 하느냐? 여러분이 상태도를 보면 Shift-DR이라고 하는 것 보이시죠? 우린 이 상태를 이용합니다. 그럼 이 DR이 무엇이냐? 바로 IR레지스터에 선택된 바운더리 스캔 셀 레지스터를 의미합니다. 바운더리 스캔 셀은 원래 핀에 연결된 놈을 말하는데 구조상 ByPass 레지스터나 ID 레지스터 등도 유사하므로 이런 놈들을 통칭할때 DR을 사용하는 것입니다.

자 그럼 IDCODE의 내용을 읽어온다라는 말을 풀어 봅시다.

1. TAPC상태를 Run-Test/Idle상태로 만든다.
2. TAPC상태를 Shift-DR상태로 한다.
3. ID CODE 패턴을 읽어 온다.
4. TAPC상태를 Run-Test/Idle상태로 만든다.

이렇게 하면 됩니다. 이걸 7탄에서 하는 것처럼 한번 해보죠....

참.참.참..... 이거 하기 전에 ID CODE를 읽어 올때 주의 할것은요 가장 먼저 TDO로 나온놈이 LSB(비트 번호 0) 이라는 거예요 즉 MSB(비트번호 31)가 가장 나중에 나온다는 것을 기억해 두세요...

그럼 시작합니다. (TAPC 상태도를 참고하시고, 시뮬레이션을 돌려 보세요)

```
(1) TAPC상태를 Run-Test/Idle상태로 만든다.
    이건 이전에 IR 레지스터에 ID CODE선택 명령을 넣을때의 마지막
    상태죠.
    그러므로 특별히 할일이 없죠?

(2) TAPC상태를 Shift-DR상태로 한다.

    TMS  1 1
    TCK  0 1    ; Select DR-SCan 상태

    TMS  0 0
    TCK  0 1    ; Capture DR 상태
                  이상태면 ID 레지스터의 내용이 TDO로
                  빠져 나올 준비가 됩니다.

    TMS  0 0
    TCK  0 1    ; Shift IR 상태
                  이상태면 DR 레지스터( ID 레지스터)와 TDI입력이 연결되고
                  TDO출력이 연결됩니다.

     경보! 경보!
     여러분은 여기서 주의하셔야 됩니다.
     이거 무시하고 넘어가면 JTAG때문에 머리털 뽑힙니다.

     제가 왜 여기서 경보를 줄까요?

     TCK의 상태를 잘 보세요.. 1로 끝났죠?
```

왜 **0**으로 끝내지 않을까요**?**

이건 **TDO**값은 **TCK**가 **1**에서 **0**으로 변할때 즉 다운 엣지에서
내부에서 외부로 밀려 나오게 되기 때문이죠

1로 끝내야만 **ID** 값이 밀려나오지 않죠**...**
(이거 이해가 되실려나 안되면 말고 **^^)**

(3) ID CODE 패턴을 읽어 온다**.**

자 여기서는
TMS와 **TDI**를 **0**으로 하고
TCK를 **0**에서 **1**로 바꾸는 행위를 **32**번 하면 됩니다**.**

```
TMS : 계속 0
TDI : 계속 0
TCK : 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
      1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
      1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
      1 0 1 0

TDO : TCK가 1에서 0으로 변할때마다 밀려 나옵니다.
      이값을 모으면 ID CODE값이 되지요...
```

(4) TAPC상태를 **Run-Test/Idle**상태로 만든다**.**

```
TMS  1 1
TCK  0 1    ; Exit1 - IR 상태

TMS  1 1
TCK  0 1    ; Update-IR 상태
              뭐 이렇게 값을 업데이트해도요,
              IDCODE레지스터는 값을 수용하지 않기 때문에
              전혀 문제가 없읍니다.

TMS  0 0
TCK  0 1    ; Run-Test/Idle 상태
```

이렇게 하면 **IDCODE**값을 **TDO**에서 읽어서 얻을 수 있지요. 사실 바운더리 셀의 내용을 넣고 가져오는 것도 거의 이와 같은 요령입니다.

IDCODE를 읽어오는 프로그램을 만드셨다면 그다음에 다른 일을 하는 것은 그리 어렵지 않습니다. 제가 직접 경험해 보니 그렇더라구요...

그럼 이번 강좌는 이걸로 끝내도 되겠군요.,. 다음 강좌는 이제 **EXTEST**와 **PC**를 이용한 하드웨어를 어떤식으로 만들면 되는지를 소개하지요... 그럼 이 긴긴 **JTAG**강좌도 거의 끝을 보게 될겁니다. 그럼....

참 항상 이야기하는 데요. 이 글의 지적 소유권 관계는 **GPL**을 따릅니다. 물론 상업적인 용도로는 사용할수 없죠.... 그럼..

JTAG의 소개 및 원리 7편

등록: 2001-07-25 15:06:13

지금 시간 새벽 **2**시.. **1**시간만 쓰고 올리렵니다.

제 **7**탄과 **8**탄에서 디바이스의 **IDCODE**를 읽어 오는 것을 알아보았읍니다. 이제 **JTAG**의 기본 원리를 아셨을 것으로 믿습니다.

이전 강좌에 제가 **IR**명령 종류에 대하여 올려 놓은 내용중 일부가 다른 자료에 의하면 약간 다른 내용이 있어서 다시 올립니다.

이 **IR**명령은 제 추측으로는 디바이스마다 약간식 구현 방법이 다른 모양입니다. 제가 직접 각종 디바이스마다 일일이 검사할수 있는 것도 아니고 이제 막 배우고 강좌를 올리기 때문에 상이한 내용은 여러분이 경험하실때가 있으면 보완해서 올려 주시기를 부탁 드립니다. 이 내용은 **SA1110(StrongARM)**에 해당됩니다.

```
B3 B2 B1 B0     다른 점 및 보완 내용
-----------     ------------------------------------------------
1  1  1  1      이 명령은 시스템 핀에 영향을 주지 않는다.
BYPASS

B3 B2 B1 B0     다른 점
-----------     ------------------------------------------------
0  0  0  0      이 명령은 내부 로직과 시스템 핀에 영향을 준다.
EXTEST          원래는 외부 핀에만 영향을 끼친다고
                책에는 나와 있읍니다.

B3 B2 B1 B0     추가
-----------     ------------------------------------------------
0  1  0  0      이건 정확이 무엇을 의미하는지 모르겠읍니다.
CLAMP
```

이 강좌에서는 **50100**보드를 기준으로 글을 계속 쓰겠습니다.

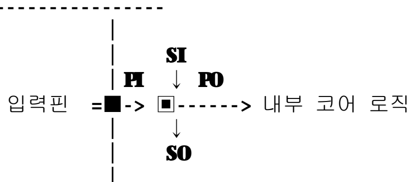
이젠 외부 보드를 테스트하기 위하여 **EXTEST**를 사용하는 방법에 대하여 알아보겠읍니다. **EXTEST**모드란 디바이스 외부 회로의 점검용으로 정의되어 있읍니다. 이 모드를 이용하면 디바이스 핀에 출력용에 원하는 신호를 실어서 내 보낼수도 있고 입력용 핀에 어떤 신호가 들어 오는지도 알수 있습니다.

하지만 **EXTEST**를 이용하기 전에 알아야 할 내용이 있읍니다. 그것이 무엇이냐**?** 바로 바운더리 셀과 디바이스 핀과의 관계입니다. 만약 디바이스 핀이 **100**개이면 바운더리 셀도 **100**개 일까요**?** 여러분은 어떻게 생각하세요..

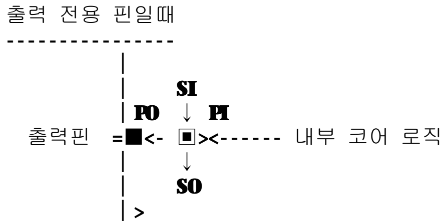
정답은 아니다 입니다. 왜냐.. 디바이스 핀은 입출력 핀뿐만 아니라.. 전원 핀이 있읍니다. 이 핀은 **JTAG**에서는 바운더리 셀에 연결되어 있지 않습니다. 또 **TAP** 핀들도 바운더리 셀에 연결되어 있지 않습니다.

그럼 이런 핀을 제외한 갯수만큼 바운더리 셀이 있는가**?** 그것 역시 정답이 아닙니다. 왜 일까요**?** 핀에는 입력 전용핀도 있고 출력 전용핀도 있읍니다. 그런데 입출력이 모두 되는 핀들도 있읍니다. 바로 이 입출력이 모두 되는 핀들이 더 많은 바운더리 셀을 가지게 됩니다.

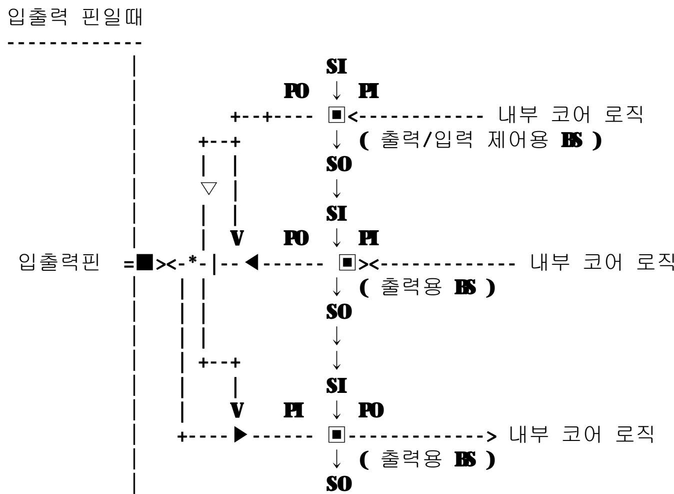
자 그림으로 볼까요**?**



이때는 입력핀이 **PI**에 연결되어 있고 바운더리 셀을 통하여 내부 코어로 전달되게 되는 구조가 됩니다.



이때는 출력핀이 **PO**에 연결되어 있고 내부 코어 로직을 바운더리 셀의 **PI** 를 통하여 **PO** 로 전달되게 되는 구조가 됩니다.



즉 입출력일때는 내부 코어 로직이 입력쪽으로 작용할것인지 아니면 출력 쪽으로 작용할 것인지에 대한 바운더리 셀이 존재 하고요. 입력용과 출력용의 바운더리 셀이 존재하는 겁니다. 그래서 핀보다 보통 많은 바운더리 셀이 존재합니다. 이런 내용은 각 디바이스의 매뉴얼을 보면 설명하고 있으므로 꼭 매뉴얼을 읽고서 시험하세요...

또 한가지 알아야 할것 핀번호순으로 셀이 있는 디바이스가 있고요 그 반대로 셀이 있는 디바이스도 있습니다.

에궁 벌써 **3**시 **20**분전이네요... **3**시까지 버티려고 했는데.... 잠이 와서리... 내일 일찍 나가야 하거든요....
다음 **10** 탄에는 **EXTEST**모드 사용법을 **50100**을 예제로 설명하죠....

참 항상 이야기하는 데요. 이 글의 지적 소유권 관계는 **GPL**을 따릅니다. 물론 상업적인 용도로는 사용할수 없죠.... 그럼..

JTAG의 소개 및 원리 8편

등록: 2001-07-25 15:06:40

에궁 오늘은 눈이 펑펑 내리는 군요... 오늘은 목요 정기 세미나가 있는데 못갈것 같네요... 대신에 이글을 올리기로 합니다. 항상 그렇듯이 시간에 쫓기고 살아서 강좌를 쓰기가 여의치 않군요.. 다른 동호회에 강좌를 쓰시는 분들의 정성이 이젠 존경스러움을 넘어서 감격 수준입니다. 여러분도 다른 분들의 강좌를 읽으실때 내용에 상관없이 감사하는 마음을 가져주셨으면 하네요 ^^;

자 이제 **EXTEST**를 통한 디바이스를 제어해 봅시다.

여러분이 임베디드 리눅스를 공부하시고 실제로 구현하기 위해서는 리눅스 커널을 올리기 전까지의 부팅 프로그램을 보드에 올려야 합니다. 뭐 어떤 분이 이런 부팅 프로그램을 올리기 위해서 일반적으로 사용되는 것이 **BGM**이란 장비라고 하네요.. 지가 사실 임베디드에는 초짜라서 이런 장비가 있는지도 몰랐읍니다. 하지만 **JTAG**가 지원되는 보드이고 **JTAG**를 알고 있다면 그런 장비가 필요없더군요... 박철님네 회사에서도 **JTAG**를 이용하여 부팅 프로그램을 플래쉬 메모리에 올리더군요... 단지 **PC**와 약간의 전압 레벨 조정용 보드만 있으면 됩니다. 이 보드역시 손수 만들어도 될 정도로 간단합니다.

각설은 이만하고 **EXTEST**모드를 이용하여 특정 핀을 제어하는 것까지 따라가 봅시다. 대상 보드는 **50100**입니다. 현재 박철님네 회사에서 제작하고 있는 평가보드 역시 시험해 보았는데 제가 정확히 이해하고 있는 것 같습니다. 그러므로 강좌의 내용을 믿으시기를 .....

우선 여러분이 **50100**보드에서의 **CELL**에 대한 정의를 아셔야 합니다. **50100**보드에는 **ARM**이 내장되어 있읍니다.

이 **ARM CELL** 정의는 다음과 같습니다.

```
000 "UARXD1 INPUT"
001 "nUADSR0 OUTPUT"
002 "UATXD0 OUTPUT"
003 "UADTR0 INPUT"
004 "UARXD0 INPUT"
005 "SDA ODOUTPUT"
006 "SDA ODINPUT"
007 "SCL ODOUTPUT"
008 "SCL ODINPUT"
009 "penb[17]"
010 "P[17] TSOUTPUT"
011 "P[17] INPUT"
012 "penb[16]"
013 "P[16] TSOUTPUT"
014 "P[16] INPUT"
015 "penb[15]"
016 "P[15] TSOUTPUT"
017 "P[15] INPUT"
018 "penb[14]"
019 "P[14] TSOUTPUT"
020 "P[14] INPUT"
021 "penb[13]"
022 "P[13] TSOUTPUT"
023 "P[13] INPUT"
024 "penb[12]"
025 "P[12] TSOUTPUT"
026 "P[12] INPUT"
027 "penb[11]"
028 "P[11] TSOUTPUT"
029 "P[11] INPUT"
030 "penb[10]"
031 "P[10] TSOUTPUT"
032 "P[10] INPUT"
033 "penb[9]"
034 "P[9]O TSOUTPUT"
035 "P[9]I INPUT"
036 "penb[8]"
```

037 "P[8] TSOUTPUT"
038 "P[8] INPUT"
039 "penb[7]"
030 "P[7] TSOUTPUT"
041 "P[7] INPUT"
042 "penb[6]"
043 "P[6] TSOUTPUT"
044 "P[6] INPUT"
045 "penb[5]"
046 "P[5] TSOUTPUT"
047 "P[5] INPUT"
048 "penb[4]"
049 "P[4] TSOUTPUT"
050 "P[4] INPUT"
051 "penb[3]"
052 "P[3] TSOUTPUT"
053 "P[3] INPUT"
054 "penb[2]"
055 "P[2] TSOUTPUT"
056 "P[2] INPUT"
057 "penb[1]"
058 "P[1] TSOUTPUT"
059 "P[1] INPUT"
060 "penb[0]"
061 "P[0] TSOUTPUT"
062 "P[0] INPUT"
063 "XDATA[31] TSOUTPUT"
064 "XDATA[31] INPUT"
065 "XDATA[30] TSOUTPUT"
066 "XDATA[30] INPUT"
067 "XDATA[29] TSOUTPUT"
068 "XDATA[29] INPUT"
069 "XDATA[28] TSOUTPUT"
070 "XDATA[28] INPUT"
071 "XDATA[27] TSOUTPUT"
072 "XDATA[27] INPUT"
073 "XDATA[26] TSOUTPUT"
074 "XDATA[26] INPUT"
075 "XDATA[25] TSOUTPUT"
076 "XDATA[25] INPUT"
077 "XDATA[24] TSOUTPUT""
078 "XDATA[24] INPUT"
079 "XDATA[23] TSOUTPUT"
080 "XDATA[23] INPUT""
081 "XDATA[22] TSOUTPUT"
082 "XDATA[22] INPUT"
083 "XDATA[21] TSOUTPUT"
084 "XDATA[21] INPUT"
085 "XDATA[20] TSOUTPUT"
086 "XDATA[20] INPUT"
087 "XDATA[19] TSOUTPUT"
088 "XDATA[19] INPUT"
089 "XDATA[18] TSOUTPUT"
090 "XDATA[18] INPUT"
091 "XDATA[17] TSOUTPUT"
092 "XDATA[17] INPUT"
093 "XDATA[16] TSOUTPUT"
094 "XDATA[16] INPUT"
095 "XDATA[15] TSOUTPUT"
096 "XDATA[15] INPUT"
097 "XDATA[14] TSOUTPUT"'
098 "XDATA[14] INPUT"
099 "XDATA[13] TSOUTPUT"
100 "XDATA[13] INPUT"
101 "XDATA[12] TSOUTPUT"
102 "XDATA[12] INPUT"
103 "XDATA[11] TSOUTPUT"
104 "XDATA[11] INPUT"
105 "XDATA[10] TSOUTPUT""
106 "XDATA[10] INPUT"
107 "XDATA[9] TSOUTPUT"
108 "XDATA[9] INPUT"
109 "XDATA[8] TSOUTPUT"
110 "XDATA[8] INPUT"
111 "XDATA[7] TSOUTPUT"
112 "XDATA[7] INPUT"
113 "XDATA[6] TSOUTPUT"
114 "XDATA[6] INPUT"
115 "XDATA[5] TSOUTPUT"
116 "XDATA[5] INPUT"

117 "XDATA[4] TSOUTPUT"
118 "XDATA[4] INPUT"
119 "XDATA[3] TSOUTPUT"
120 "XDATA[3] INPUT"
121 "XDATA[2] TSOUTPUT"
122 "XDATA[2] INPUT"
123 "XDATA[1] TSOUTPUT"
124 "XDATA[1] INPUT"
125 "XDATA[0] TSOUTPUT"
126 "XDATA[0] INPUT"
127 "denb"
128 "PADDR[21] TSO"
129 "PADDR[20] TSO"
130 "PADDR[19] TSO"
131 "PADDR[18] TSO"
132 "PADDR[17] TSO"
133 "PADDR[16] TSO"
134 "PADDR[15] TSO"
135 "PADDR[14] TSO"
136 "PADDR[13] TSO"
137 "PADDR[12] TSO"
138 "PADDR[11] TSO"
139 "PADDR[10] TSO"
130 "PADDR[9] TSO"
141 "PADDR[8] TSO"
142 "PADDR[7] TSO"
143 "PADDR[6] TSO"
144 "PADDR[5] TSO"
145 "PADDR[4] TSO"
146 "PADDR[3] TSO"
147 "PADDR[2] TSO"
148 "PADDR[1] TSO"
149 "PADDR[0] TSO"
150 "ExtMACK OUTPUT"
151 "ExtMREQ TSO"
152 "nMWE[3] TSO"
153 "nMWE[2] TSO"
154 "nMWE[1] TSO"
155 "nMWE[0] TSO"
156 "nDWE TSO"
157 "nCAS[3] TSO"
158 "nCAS[2] TSO"
159 "nCAS[1] TSO"
160 "nCAQS[0] TSO"
161 "nRAS[3] TSO"
162 "nRAS[2] TSO"
163 "nRAS[1] TSO"
164 "nRAS[0] TSO"
165 "nRCS[5] TSO"
166 "nRCS[4] TSO"
167 "nRCS[3] TSO"
168 "nRCS[2] TSO"
169 "nRCS[1] TSO"
170 "CLKSEL INPUT"
171 "nRESET INUT"
172 "MCLK INPUT"
173 "MCKLO OUTPUT"
174 "CLKOEN INPUT"
175 "nRCS[0] TSO"
176 "BOSIZE[1] INPUT"
177 "BOSIZE[0] INPUT"
178 "nOE TSO"
179 "nEWAIT INPUT"
180 "nECS[3] TSO"
181 "nECS[2] TSO"
182 "nECS[1] TSO"
183 "nECS[0] TSO"
184 "dis bus"
185 "UCLK INPUT"
186 "TMODE INPUT"
187 "MDC OUTPUT"
188 "LITTLE"
189 "mdio oe"
190 "MDIO TSO"
191 "MDIO INPUT"
192 "TXEN/TXEN10M OUTPUT"
193 "TXCLK/TXCLK10M INPUT"
194 "TXERR/PCOMP10M OUTPUT"
195 "TXD3 OUTPUT"
196 "TXD2 OUTPUT"

```
197 "TXD1/LOOP10 OUTPUT"
198 "TXD0/TXD10M OUTPUT"
199 "COL/COL10M INPUT"
200 "RXCLK/RXCLK10M INPUT"
201 "RX ERR INPUT"
202 "RXD3 INPUT"
203 "RXD2 INPUT"
204 "RXD1 INPUT"
205 "RXD0/RXD10M INPUT"
206 "RXDV/LINK10 INPUT"
207 "CRS/CRS10M INPUT"
208 "txcben"
209 "TXCB TSOUT"
200 "TXCB INPUT"
211 "nSYNCB OUTPUT"
212 "RXCB INPUT"
213 "nDCDB INPUT"
214 "nCTSB INPUT"
215 "TXDB OUTPUT"
216 "nRTSB OUTPUT"
217 "RXDB"
218 "nDTRB OUTPUT"
219 "txcaen"
210 "TXCA OUTPUT"
221 "TXCA INPUT"
222 "nSYNCA OUTPUT"
223 "RXCA INPUT"
224 "nDCDA INPUT"
225 "nCTSA INPUT"
226 "TXDA OUTPUT"0
227 "nRTSA OUTPUT"
228 "RXDA INPUT"
229 "nDTRA OUTPUT"
220 "nUADSR1 OUTPUT"
231 "UATXD1 OUTPUT"
232 "UADTR1 INPUT"
```

총 **233 CELL** 이군요..... 즉 밀어 넣어야 할 비트 패턴이 **233**개란 이야기 입니다.

에궁 일딴 여기까지만 써야 할것 같네요... 그리고용 위 **ARM CELL** 이름은 박웅근씨가 작업해 주셨읍니다.

참 항상 이야기하는 데요. 이 글의 지적 소유권 관계는 **GPL**을 따릅니다. 물론 상업적인 용도로는 사용할수 없죠.... 그럼..

JTAG의 소개 및 원리 9편

등록: 2001-07-25 15:07:31

JTAG의 소개 및 원리 8편에서 정의된 CELL을 참조 하셔서 이 내용을 이해 하셔야 합니다.

EXTEST를 접근하는 방식은 7탄이후에 설명한 IDCODE를 얻어오는 방식과 유사합니다. 자 그럼 EXTEST의 접근 방법을 알아 봅시다.

1. TAPC상태를 초기화 한다.
2. TAPC상태를 Shift-IR상태로 한다.
3. EXTEST 명령 패턴을 밀어 넣는다.
4. TAPC상태를 Run-Test/Idle상태로 만든다.
5. 원하는 핀상태를 미리 결정하여 비트 패턴을 만든다.
6. TAPC상태를 Shift-DR상태로 한다.
7. 결정된 비트 패턴을 밀어 넣으면서 밀려 나온 패턴을 저장한다.
8. TAPC상태를 Run-Test/Idle상태로 만든다.
9. 출력 상태를 바꾸거나 입력 핀상태를 알아보려면 (7) 항부터 계속 반복한다.

이전과 비슷하지요? 자 그럼 각각을 수행해 봅시다.

(1) TAPC상태를 초기화 한다.

```
TMS를 1(HIGH) 상태로
TCK를 6클럭을 줍니다.

TMS  1 1 1 1 1 1 1 1 1 1 1 1
TCK  0 1 0 1 0 1 0 1 0 1 0 1

이 상태가 되면 TACP는 Test-Logic-Reset상태가 됩니다
```

(2) TAPC상태를 Shift-IR상태로 한다.

```
TMS  0 0
TCK  0 1    ; Run-Test/Idle 상태

TMS  1 1
TCK  0 1    ; Select DR-SCan 상태

TMS  1 1
TCK  0 1    ; Select IR-SCan 상태

TMS  0 0
TCK  0 1    ; Capture IR 상태
              이상태면 IR 내용이 쉬프트되서 TDO로
              빠져 나올 준비가 됩니다.

TMS  0 0
TCK  0 1    ; Shift IR 상태
              이상태면 IR 레지스터와 TDI입력이 연결되고
              TDO출력이 연결됩니다
```

(3) EXTEST 명령 패턴을 밀어 넣는다.

```
EXTEST 명령 패턴은 0000 입니다.

TDI  0 0
TCK  0 1    ; B3가 입력됩니다.

TDI  1 0
TCK  0 1    ; B2가 입력됩니다.

TDI  0 0
TCK  0 1    ; B1가 입력됩니다.

다시 강조 하지만 여기서 고만... 이전에도 여기서 주의를 주었죠?
```

이거 이거 이부분이 무척 중요합니다.

TMS와 **TDI**는 **TCK**의 업 엣지에 영향을 받는다는 사실을 아시죠?
즉 **TCK** 가 **0**에서 **1**로 변할때 **TMS**와 **TDI**가 **JTAG**내부로 인가됩니다.

우린 패턴을 밀어넣은후에 **TAPC**상태를 **Run-Test/Idle**상태로 만들기로
예정되어 있죠?

그렇다면 **TMS**를 변화시켜야 하는데
이때 **TDI**상태도 그대로 밀려 들어가 버립니다.
그래서 **TAPC**의 상태를 **Shhift-IR**상태에서 **Exit1-IR**상태로
바꿀때 **TDI**도 같이 싫어서 **TCK**를 **0**에서 **1**로 바꾸어야 하는 거죠.

(4) TAPC상태를 **Run-Test/Idle**상태로 만든다.

```
TDI  0 0
TMS  1 1
TCK  0 1    ; Exit1 - IR 상태
              동시에 마지막 B4값 TDI에 인가

TMS  1 1
TCK  0 1    ; Update-IR 상태
              우리가 쉬프트한 값이 이때 IR 레지스터에
              적용됩니다.

TMS  0 0
TCK  0 1    ; Run-Test/Idle 상태

              다시 한번 주의 하십시오.
              Update-IR상태에서 Run-Test/Idle 상태로 이동해야지
              만약 Test-Logic-Reset상태로 이동해 버리면,
              여짓껏 한 행위가 모두 도로나무아비타불입니다. ^^;
```

(5) 원하는 핀상태를 미리 결정하여 비트 패턴을 만든다.

자 이전 강좌에서 보셨겠지만... **CELL** 수가 **233**개 입니다.
그러므로 만들어야할 비트 패턴도 **233**개죠.

이전 강좌에서의 **CELL** 정의를 다시 보면

021 "penb[13]"
022 "P[13] TSOUTPUT"
023 "P[13] INPUT"

이렇게 되어 있죠?

이게 **P13**핀에 연결되어 있는 **CELL**들이죠
P13은 입출력이 모두 가지고 있읍니다.

022 CELL은 **P13 PIN**에 출력될 값을 가지게 되고요.
023 CELL은 **P13 PIN**에 입력된 값을 가지게 되지요.

그럼 **021**은 무엇일까요?
그건 **P13**이 입력용으로 사용될것인지 출력용으로 사용될것인지를
선택하는 놈입니다.

이전 강좌에서 설명한 그림을 한번 볼까요? **(9**탄 참조**)**

(출력/입력 제어용 **BS)**

50100 에서는 **021**에 **0**을 주면 출력용이 됩니다.
자 이런 원리를 이용하여 비트 패턴을 만듭니다.
CELL번호가 **000**이 가장 먼저 입력되어야 합니다.

(6) TAPC상태를 **Shift-IR**상태로 한다.

```
TMS 1 1
TCK 0 1    ; Select IR-SCan 상태

TMS 0 0
TCK 0 1    ; Capture IR 상태
             이상태면 ID 레지스터의 내용이 TDO로
             빠져 나올 준비가 됩니다.

TMS 0 0
TCK 0 1    ; Shift IR 상태
             이상태면 IR 레지스터(ID 레지스터)와 TDI입력이 연결되고
             TDO출력이 연결됩니다.
```

다시 한번 경보! 경보!

TCK의 상태를 잘 보세요.. **1**로 끝났죠?
왜 **0**으로 끝내지 않을까요?

이건 **TDO**값은 **TCK**가 **1**에서 **0**으로 변할때 즉 다운 엣지에서
내부에서 외부로 밀려 나오게 되기 때문이죠.

1로 끝내야만 **ID** 값이 밀려나오지 않죠...

(7) 결정된 비트 패턴을 밀어 넣으면서 밀려 나온 패턴을 저장한다.

TMS : 계속 **0**
TDI : 계속 **0**
TCK : 1 0 을 **233**번 반복합니다.
TDO : TCK가 **1**에서 **0**으로 변할때마다 밀려 나옵니다.
이값을 모으면 핀에 연결된 **CELL** 값들이 되지요...

(8) TAPC상태를 **Run-Test/Idle**상태로 만든다

```
TMS 1 1
TCK 0 1    ; Exit1 - IR 상태

TMS 1 1
TCK 0 1    ; Update-IR 상태
             이렇게 값을 업데이트합니다.
             이때 핀의 출력 상태가 변하게 됩니다.

TMS 0 0
TCK 0 1    ; Run-Test/Idle 상태
```

(9) 출력 상태를 바꾸거나 입력 핀상태를 알아보려면 **(7)** 항부터 계속 반복한다.

주의 하실것은 밀려나온 상태는 **Scan**해서 밀어 넣은 출력에 응답한
디바이스의 입력이 아니라는 점입니다.

바로 응답 이전값입니다.

제가 이것 때문에 플레쉬 메모리제어 할때 잠깐 해멨어요...

자 이렇게 해서 **EXTEST**에 대하여 알아보았읍니다. 쉽지요...

제가 강좌 순서에서

- **JTAG** 의 소개 및 원리
- 아사벳 보드에서의 **JTAG**
- **JTAG**와 **PC** 와의 연결
- **JTAG** 를 이용한 플레쉬 메모리에 기록하는 방법 및 프로그램 예제..

이런식으로 하려고 했는데요..

- **JTAG** 의 소개 및 원리..는 여기서 끝내야 할것 같고요..

- **JTAG** 를 이용한 플레쉬 메모리에 기록하는 방법 및 프로그램 예제..는 제가 프로그램하나 올릴께요.. 그걸로 대치하려고 합니다. 왜냐하면 이게 글로 쓰려니 장난이 아니라서. 만약 여러분이 강좌대신 세미나를 요청하시면 그때 제가 하지요...
- 아사벳 보드에서의 **JTAG**는 제가 아사벳 보드를 구하지 못했어요.. 그래서 힘들것 같네요.. 아사벳 보드에는 **CPLD**가 여러개 있어서 **BYPASS**와 관련된 것을 할수 있을텐데.. 혹시 아사벳 보드 가지신분 계시면 연락 주세요 그러면 제가 시험하고 강좌를 올리지요. 제 연락 메일은 frog6502@hanmail.net입니다.

그래서 남은것은 **JTAG**와 **PC**와의 연결이란 강좌만 남았네요... 다음 강좌는 이것을 하지요.. 이걸 끝으로 **JTAG**강좌는 끝내겠습니다. **JTAG**강좌가 끝나면 **BLOB**강좌를 시작 할것이고요. 그 다음엔 **NETWORK** 부팅에 관련된 강좌를 박철님에게 하사 받으면 하지요...

그럼 ....

참 항상 이야기하는 데요. 이 글의 지적 소유권 관계는 **GPL**을 따릅니다. 물론 상업적인 용도로는 사용할수 없죠.... 그럼..

- E N D -