



THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 10

직렬 버스

1. 소개

이 LAB에서는 먼저 직렬 버스의 작동 방식과 현재 사용되는 가장 일반적인 직렬 버스 중 하나인 SPI 버스 (섹션 2)의 주요 기능에 대해 설명합니다. 그런 다음 Nexys A7 보드에서 사용할 수 있는 SPI 가속도계에 초점을 맞춥니다. 이 주변 장치에 대한 높은 수준의 사양을 분석하고 기본 연습 (섹션 3 및 4)을 제안한 다음, 초급 수준 구현을 분석하고 몇 가지 고급 연습을 제안합니다 (섹션 5 및 6).

2. 직렬 버스 – SPI 버스

병렬 버스는 한 번에 여러 비트를 보내는 반면 직렬 버스는 한 번에 한 비트를 보냅니다. 먼저 이 두 가지 통신 방식을 비교한 다음 현재 사용되는 가장 일반적인 직렬 버스 중 하나인 SPI (직렬 주변 장치 인터페이스) 프로토콜을 설명합니다. 이 중요한 통신 프로토콜에 대한 지식을 넓히기 위해 인터넷에서 많은 정보를 찾을 수 있습니다.

이전 LAB에서 이미 설명했듯이 임베디드 장치의 주요 목적은 프로세서와 회로를 연결하여 원하는 기능을 만드는 것입니다. 프로세서와 회로가 정보를 공유하려면 공통 통신 프로토콜을 공유해야 합니다. 이러한 데이터 교환을 달성하기 위해 수백 개의 통신 프로토콜이 정의되었으며 일반적으로 병렬 또는 직렬 인터페이스의 두 가지 주요 범주로 나눌 수 있습니다.

병렬 인터페이스는 여러 비트를 병렬로 즉, 동시에 전송합니다. 데이터 버스 (다중 와이어)가 필요합니다. 예를 들어, 프로토콜은 동시에 8, 16 또는 그 이상의 비트를 전송할 수 있습니다 (그림 1 참조). 또한 새로운 N 개의 데이터 비트 그룹이 전송될 준비가 되었을 때 클럭이 필요합니다.

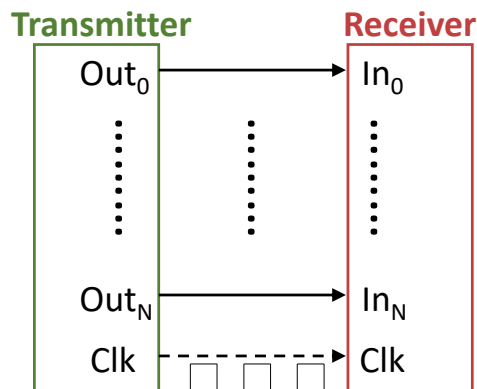


그림 1. 병렬 8 비트 데이터 버스의 예.

병렬 통신과 달리 직렬 인터페이스는 데이터를 한 번에 한 비트씩 스트리밍합니다. 이러한 인터페이스는 최소 1 개의 와이어를 사용하여 작동할 수 있으며 일반적으로 4 개를 초과하지 않습니다. 그림 2는 데이터용 와이어와 클럭용 와이어가 있는 직렬 인터페이스의 예를 보여줍니다. 각각의 새로운 클럭 에지에서 새로운 데이터 비트가 전송됩니다.

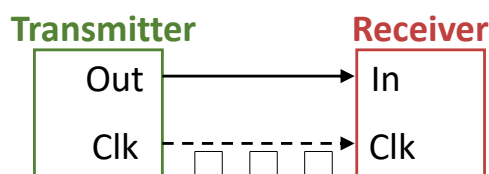


그림 2. 직렬 1 비트 데이터 버스의 예.

병렬 통신은 빠르고 간단하며 비교적 쉽게 구현할 수 있는 이점이 있습니다. 그러나 더 많은 입력/출력 (I/O) 라인이 필요합니다. 따라서 핀이 제한되어 있기 때문에 임베디드 시스템은 종종 직렬 통신을 선택하여 핀 공간에 대한 잠재적인 속도를 희생합니다.

SPI Bus:

SPI (Serial Peripheral Interface) 프로토콜은 마이크로 컨트롤러와 센서, ADC, DAC, 시프트 레지스터, SRAM 등과 같은 주변 IC 간에 가장 널리 사용되는 인터페이스 중 하나입니다. SPI는 컨트롤러-주변 장치 (이전에는 마스터-슬레이브라고 함) 통신을 기반으로 하는 동기식 이중 인터페이스입니다.

SPI 버스는 일반적으로 4 개의 포트를 통해 통신합니다 (그림 3 참조).

- **SDO** – 직렬 데이터 출력: 주변 장치에 대한 컨트롤러 출력
- **SDI** – 직렬 데이터 입력: 주변 장치에서 컨트롤러의 입력
- **SCK** – 직렬 클럭: 컨트롤러에서 주변 장치로 전송
- **CS** – 칩 선택: 액티브 로우 신호; 컨트롤러가 주변기기로 신호 (주변기기를 선택한 경우 0)를 보냅니다.

참고: 역사적으로 SDO는 MOSI (마스터 데이터 출력, 슬레이브 데이터 입력)라고도 하고 SDO는 MISO (마스터 데이터 입력, 슬레이브 데이터 출력)라고 합니다. 이러한 용어는 구식이고 모욕적이지만 여전히 문헌과 문서에 존재합니다.

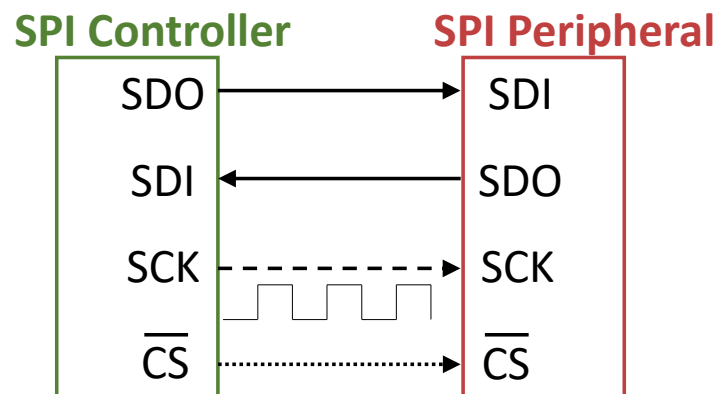


그림 3. 하나의 SPI 컨트롤러와 하나의 SPI 주변 장치가 있는 시스템의 예.

직렬 데이터는 상승 또는 하강 클럭 에지에 동기화됩니다. SPI는 이중 인터페이스입니다. 컨트롤러와 주변기기는 각각 SDO 및 SDI 라인을 통해 동시에 데이터를 전송할 수 있습니다. SPI 인터페이스에는 컨트롤러가 하나만 있지만 주변 장치가 여러 개 있을 수 있습니다. 둘 이상의 주변 장치가 연결된 경우 컨트롤러의 여러 CSbar (Low-Asserted Chip Select Signal)를 사용하여 액세스할 주변 장치를 선택합니다. SDO 및 SDI는 직렬 데이터 라인입니다. SDO (직렬 데이터 출력)는 컨트롤러에서 주변 장치로의 출력 데이터이고 SDI (직렬 데이터 입력)는 주변 장치에서 컨트롤러로의 입력 데이터입니다.

SPI 통신을 시작하려면 컨트롤러가 주변 장치를 선택한 다음 (CSbar 신호를 강제, 즉 CSbar = 0) 주변 장치로 클럭 신호를 전송해야 합니다. SPI 통신 중에 데이터는 각각 SDO 및 SDI 신호를 통해 컨트롤러에서 동시에 컨트롤러로 전송됩니다. 직렬 클럭 (SCK) 에지는 데이터 샘플링을 동기화 합니다.

SPI 인터페이스는 또한 클럭의 유희 상태와 신호 샘플링을 위한 위상을 선택하도록 추가적으로 CPOL 및 CPHA 신호를 제공합니다. 클럭 극성 (CPOL) 신호는 클럭 (SCK)이 0 에서 유희 상태일 때 0 이고 1 에서 유희 상태일 때 1 입니다. 클럭 위상 (CPHA) 신호는 데이터를 전송하고 샘플링할 클럭의 위상을 선택합니다. CPHA = 0 이면 데이터 (SDI 또는 SDO 에서)가 리딩 에지 (즉, SCK 이후의 첫 번째 에지가 유희 상태를 중지하고 그 이후의 모든 사이클에서)에서 샘플링됩니다. 따라서 데이터 (SDI 및 SDO)는 그림 4 의 맨 위 두 타이밍 다이어그램에 표시된 것처럼 후행 에지에서 변경되어야 합니다. CPHA = 1 은 그 반대를 수행합니다. 데이터는 후행 에지에서 샘플링되고 데이터는 선행 에지에서 변경됩니다. 이 직렬 통신은 일반적으로 시프트 레지스터를 사용하여 구현되기 때문에 새 데이터가 전송되는 에지를 *시프팅 에지(shifting edge)* 라고도 합니다.

이 LAB 에서 사용하는 SPI 인터페이스는 CPHA = 0 및 CPOL = 0 이므로 SCK 는 낮게 유희 상태로 유지되고 컨트롤러 및 주변 장치 샘플 데이터는 상승 에지에서 각 하강 에지 직후에 새 데이터를 라인 (SDO 또는 SDI)으로 이동합니다. 그림 4 의 상단 타이밍 다이어그램에서 볼 수 있습니다. SCK 가 유희 상태일 때 상승하기 직전에 SDO 와 SDI 는 다음 데이터 바이트의 최상위 비트를 전달해야 합니다.

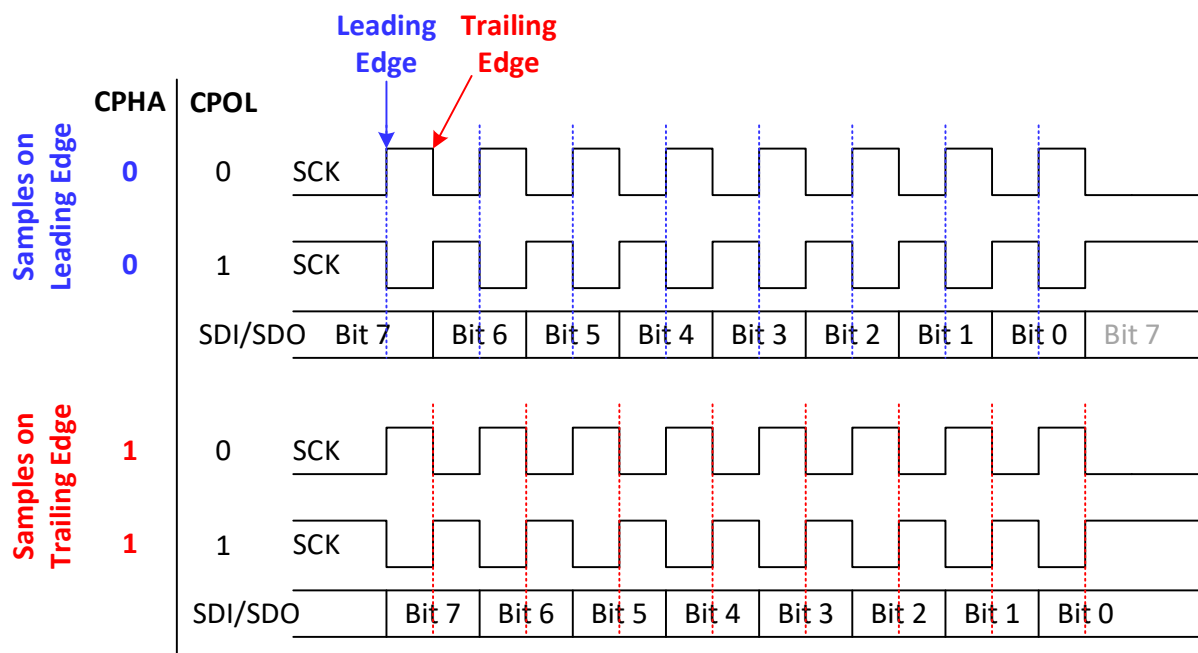


그림 4. CPHA/CPOL 과 샘플링/전송 데이터의 관계

3. SPI 가속도계: 고급 사양

많은 주변 장치에는 SPI 인터페이스가 포함되어 있습니다. 예를 들어 Nexys A7 보드의 가속도계에는 SPI 인터페이스가 있습니다. 이 섹션에서는 RVfpga 시스템의 SPI 컨트롤러의 고급 사양을 설명하고 Nexys A7 보드에 포함된 ADXL362 가속도계를 소개합니다. 가속도계를 사용하는 실습도 소개합니다.

A. SPI 컨트롤러 사양

RVfpgat 시스템의 SPI 모듈은 OpenCores (https://opencores.org/projects/simple_spi) 에서 제공됩니다.
패키지를 다운로드하면 모듈의 고급 사양을 설명하는 문서가 제공됩니다. 이 문서는 여기에서도 제공됩니다.

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/spi/docs/simple_spi.pdf

SPI 모듈의 주요 작동 및 기능을 요약합니다. 그러나 추가 정보는 위 문서를 참조하십시오.

이 모듈에는 다음과 같은 주요 기능이 있습니다.

- Motorola 의 SPI 사양과 호환됩니다.
- 8 비트 WISHBONE RevB.3 클래식 인터페이스를 사용합니다.
- 4 개 항목 읽기 FIFO 버퍼와 4 개 항목 쓰기 FIFO 버퍼 포함
- 1, 2, 3, 4 바이트 전송 후 인터럽트 발생 가능
- 다양한 입력 클럭 주파수로 동작 가능
- 완전 합성 가능

SPI 코어 사양의 섹션 3에서는 SPI 모듈 내에서 사용할 수 있는 제어 및 상태 레지스터를 설명하며, 각 레지스터는 서로 다른 주소에 할당됩니다 (표 1 참조). SPI 컨트롤러의 기본 주소는 **0x80001100** 입니다. 이러한 레지스터는 아래에 자세히 설명되어 있습니다.

표 1. SPI 레지스터

Name	Address	Width	Access	Description
SPCR	0x80001100	8	R/W	제어 레지스터
SPSR	0x80001108	8	R/W	상태 레지스터
SPDR	0x80001110	8	R/W	데이터 레지스터
SPER	0x80001118	8	R/W	확장 등록
SPCS	0x80001120	8	R/W	CS 레지스터

SPCR (SPI Control Register)은 SPI 모듈을 제어합니다. 표 2 는 각 비트의 기능을 보여줍니다.

표 2. SPCR 비트

Bit	Access	Name & Description
0:1	R/W	SPR SPI 클럭 속도: 이 비트는 SPI 클럭 속도를 선택합니다.
2	R/W	CPHA 클럭 위상: 샘플링 및 데이터 전송 단계를 결정합니다. CPHA = 1 이면 새 데이터가 앞쪽 가장자리에서 와이어로 이동되고 데이터가 뒤쪽 가장자리에서 샘플링 됩니다. CPHA = 0 인 경우 새 데이터는 후행 가장자리에서 와이어로 이동되고 선행 가장자리에서 샘플링 됩니다.
3	R/W	CPOL 클럭 극성: SPI 클럭 (SCK)의 유효 상태를 결정합니다. CPOL = 0 이면 SCK 는 0 에서 유효 상태이고 CPOL = 1 이면 SCK 는 1 에서 유효 상태입니다.
4	R/W	MSTR 모드 선택: MSTR = 1 일 때 SPI 코어는 컨트롤러 장치입니다. 이 컨트롤러에 대해 유일하게 지원되는 모드입니다.

6	R/W	SPE SPI 활성화: SPE = 1 이면 SPI 코어가 활성화됩니다. 지워지면 (SPE = 0) SPI 코어가 비활성화 됩니다.
7	R/W	SPIE SPI 인터럽트 활성화: SPIE = 1 일 때 상태 레지스터의 SPI 인터럽트 플래그가 설정되면 호스트가 인터럽트 됩니다.

SPSR (SPI Status Register)은 SPI 모듈의 상태를 제공합니다. 표 3 은 각 비트의 기능을 보여줍니다.

표 3. SPSR 비트

Bit	Access	Description
0	R/W	RFEMPTY Read FIFO Empty: RFEMPTY = 1 이면 읽기 FIFO 가 비어 있습니다.
1	R/W	RFFULL Read FIFO Full: RFFULL = 1 이면 읽기 FIFO 가 가득 찼습니다.
2	R/W	WFEMPTY Write FIFO Empty: WFEMPTY = 1 이면 쓰기 FIFO 가 비어 있습니다.
3	R/W	WFFULL Write FIFO Full: WFFULL = 1 이면 쓰기 FIFO 가 가득 찼습니다.
6	R/W	WCOL Write Collision flag: WCOL = 1 이면 Write FIFO 가 가득 찬 동안 SPDATA 레지스터에 기록되었습니다. WCOL 에 1 을 쓰면 이 비트가 지워집니다.
7	R/W	SPIF SPI 인터럽트 플래그: 전송 블록 완료시 SPIF = 1. SPIF 가 강제 ('1')되고 SPIE 가 설정되면 인터럽트가 생성됩니다. SPIF 에 1 을 쓰면 지워집니다.

SPDR (SPI 데이터 레지스터)은 읽거나 쓸 데이터를 제공합니다. SPI 컨트롤러에는 4 개의 8 비트 쓰기 버퍼와 4 개의 8 비트 읽기 버퍼가 포함되어 있습니다.

SPER (SPI 확장 레지스터)는 몇 가지 추가 기능을 제공합니다. 표 4 는 여기에 포함된 다양한 필드를 설명합니다.

표 4. SPER 비트

Bit	Access	Description
0:1	R/W	ESPR Extended SPI Clock Rate Select: SPR (SPI 클럭 속도 선택)에 2 비트를 추가합니다.
6:7	R/W	ICNT 인터럽트 수: 전송 블록 크기를 결정합니다. SPIF 비트는 ICNT 전송 후에 설정됩니다. 따라서 인터럽트 서비스 호출 감소로 인해 커널 오버 헤드를 줄일 수 있습니다.

마지막으로 SPCS (SPI Chip Select) 레지스터는 사용할 주변 장치를 선택합니다. 이 신호의 폭은 매개 변수 SS_WIDTH (SPI 선택 폭)를 통해 구성할 수 있습니다. RVfpga 시스템에서는 각 SPI 인터페이스에 대해 하나의 주변 장치만 존재하므로 SS_WIDTH = 1 입니다.

작업: SPI 모듈에서 레지스터 SPCR, SPSR, SPDR, SPER 및 SPCS 의 선언과 해당 주소의 정의를 찾습니다. SPI 모듈은 `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/spi` 폴더에서 사용할 수 있습니다.

B. ADXL362 가속도계 사양

Nexys A7 보드에는 Analog Devices ADXL362 가속도계가 포함되어 있습니다. 다음 위치에 있는 데이터 시트에서 장치에 대한 전체 정보를 찾을 수 있습니다:

<https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL362.pdf>

ADXL362 는 3 축 MEMS 가속도계로 100Hz 출력 데이터 속도에서 2 μ A 미만을 소비하고 모션 트리거 웨이크업 모드에서 270nA 를 소비합니다. 낮은 해상도로 충분할 때보다 효율적인 단일 바이트 전송을 위해 8 비트 형식의 데이터도 제공되지만 12 비트 출력 해상도를 제공합니다. $\pm 2g$, $\pm 4g$ 및 $\pm 8g$ 의 측정 범위는 $\pm 2g$ 범위에서 1mg / LSB 의 분해능으로 제공됩니다. ADXL362 가 측정 모드에 있는 동안 가속 데이터를 X- 데이터, Y- 데이터 및 Z- 데이터 레지스터에 지속적으로 측정하고 저장합니다.

ADXL362 가속도계에는 사용자가 이를 구성하고 가속 데이터를 읽을 수 있는 여러 레지스터 (표 5)가 포함되어 있습니다. 장치는 제어 레지스터에 기록하여 구성하고 가속도계 데이터는 장치 레지스터를 읽어서 찾습니다. 장치와의 모든 통신은 통신이 읽기인지 쓰기인지를 나타내는 레지스터 주소와 플래그를 지정해야 합니다. 데이터 전송은 레지스터 주소와 통신 플래그가 장치로 전송된 후에 발생합니다.

이 가속도계는 SPI 통신 방식을 사용하는 주변 장치 역할을 합니다. FPGA 와 가속도계 간의 인터페이스는 그림 5 에 나와 있습니다.

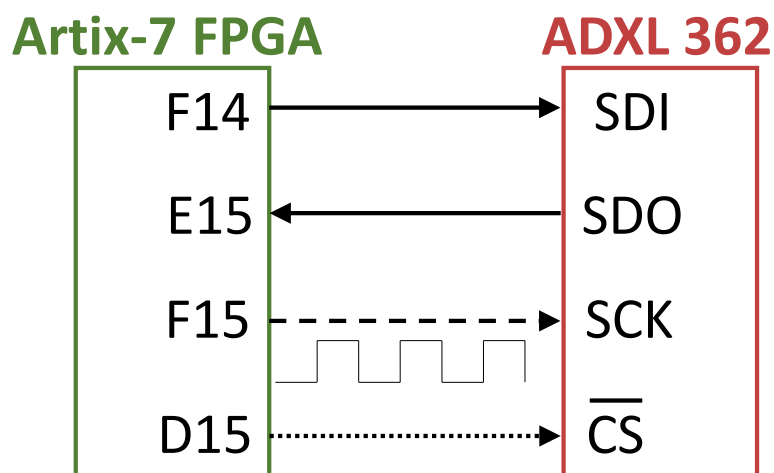


그림 5. Nexys A7 보드가있는 ADXL362 가속도계 인터페이스

권장되는 SPI 클럭 주파수 범위는 1-5MHz 입니다. SPI 는 SPI 모드 0 (CPOL = 0 및 CPHA = 0)에서 작동합니다. SPI 포트는 멀티 바이트 구조를 사용하며 첫 번째 바이트는 통신이 레지스터 읽기 (0x0B)를 수행하는지 레지스터 쓰기 (0x0A)를 수행 하는지를 나타냅니다.

<CS down> <Write/Read (0x0A/0x0B)> <address byte> <data byte> <CS up>

그림 6 및 그림 7 은 SPI 컨트롤러 (컨트롤러)와 가속도계 (주변 장치) 간의 통신에 대한 두 가지 예를 보여줍니다. 그림 6 은 레지스터 읽기를 보여주고 그림 7 은 레지스터 쓰기를 보여줍니다.

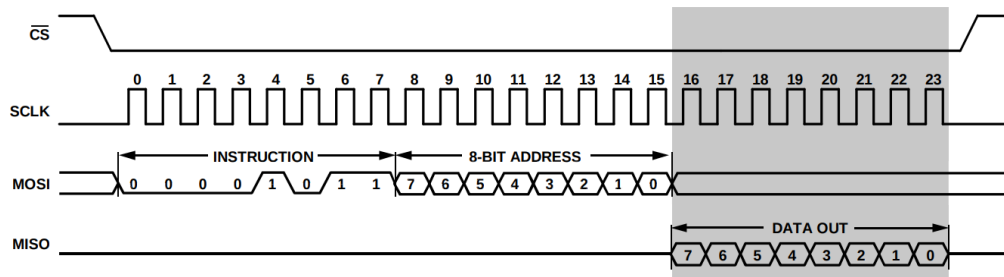


그림 6. 레지스터 읽기

(그림 출처 <https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL362.pdf>)

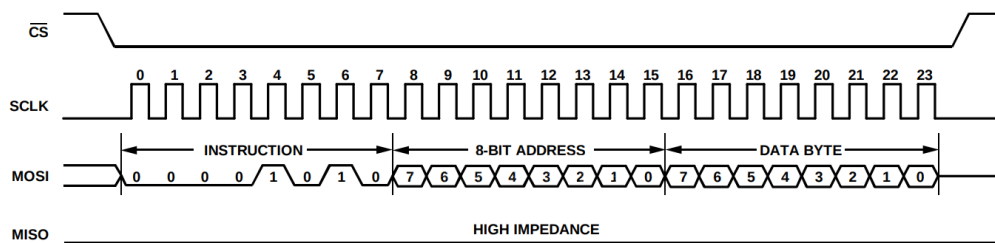


그림 7. 레지스터 쓰기

(그림 출처 <https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL362.pdf>)

표 5 는 ADXL362 가속도계에서 사용 가능한 레지스터를 보여줍니다. 전체 레지스터 설명은 ADXL362 데이터 시트를 참조하십시오:

<https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL362.pdf>

표 5. ADXL362 가속도계 레지스터

(표 출처 <https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL362.pdf>)

Reg	Name	Bits	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Reset	RW	
0x00	DEVID_AD	[7:0]	DEVID_AD[7:0]								0xAD	R	
0x01	DEVID_MST	[7:0]	DEVID_MST[7:0]								0x1D	R	
0x02	PARTID	[7:0]	PARTID[7:0]								0xF2	R	
0x03	REVID	[7:0]	REVID[7:0]								0x01	R	
0x08	XDATA	[7:0]	XDATA[7:0]								0x00	R	
0x09	YDATA	[7:0]	YDATA[7:0]								0x00	R	
0x0A	ZDATA	[7:0]	ZDATA[7:0]								0x00	R	
0x0B	STATUS	[7:0]	ERR_USER_REGS	AWAKE	INACT	ACT	FIFO_OVER-RUN	FIFO_WATER-MARK	FIFO_READY	DATA_READY	0x40	R	
0x0C	FIFO_ENTRIES_L	[7:0]	FIFO_ENTRIES_L[7:0]								0x00	R	
0x0D	FIFO_ENTRIES_H	[7:0]	UNUSED							FIFO_ENTRIES_H[1:0]	0x00	R	
0x0E	XDATA_L	[7:0]	XDATA_L[7:0]								0x00	R	
0x0F	XDATA_H	[7:0]	SX				XDATA_H[3:0]				0x00	R	
0x10	YDATA_L	[7:0]	YDATA_L[7:0]								0x00	R	
0x11	YDATA_H	[7:0]	SX				YDATA_H[3:0]				0x00	R	
0x12	ZDATA_L	[7:0]	ZDATA_L[7:0]								0x00	R	
0x13	ZDATA_H	[7:0]	SX				ZDATA_H[3:0]				0x00	R	
0x14	TEMP_L	[7:0]	TEMP_L[7:0]								0x00	R	
0x15	TEMP_H	[7:0]	SX				TEMP_H[3:0]				0x00	R	
0x16	Reserved	[7:0]	Reserved[7:0]								0x00	R	
0x17	Reserved	[7:0]	Reserved[7:0]								0x00	R	
0x1F	SOFT_RESET	[7:0]	SOFT_RESET[7:0]								0x00	W	
0x20	THRESH_ACT_L	[7:0]	THRESH_ACT_L[7:0]								0x00	RW	
0x21	THRESH_ACT_H	[7:0]	UNUSED				THRESH_ACT_H[2:0]				0x00	RW	
0x22	TIME_ACT	[7:0]	TIME_ACT[7:0]								0x00	RW	
0x23	THRESH_INACT_L	[7:0]	THRESH_INACT_L[7:0]								0x00	RW	
0x24	THRESH_INACT_H	[7:0]	UNUSED				THRESH_INACT_H[2:0]				0x00	RW	
0x25	TIME_INACT_L	[7:0]	TIME_INACT_L[7:0]								0x00	RW	
0x26	TIME_INACT_H	[7:0]	TIME_INACT_H[7:0]								0x00	RW	
0x27	ACT_INACT_CTL	[7:0]	RES		LINKLOOP		INACT_REF	INACT_EN	ACT_REF	ACT_EN	0x00	RW	
0x28	FIFO_CONTROL	[7:0]	UNUSED				AH	FIFO_TEMP	FIFO_MODE		0x00	RW	
0x29	FIFO_SAMPLES	[7:0]	FIFO_SAMPLES[7:0]								0x80	RW	
0x2A	INTMAP1	[7:0]	INT_LOW	AWAKE	INACT	ACT	FIFO_OVER-RUN	FIFO_WATER-MARK	FIFO_READY	DATA_READY	0x00	RW	
0x2B	INTMAP2	[7:0]	INT_LOW	AWAKE	INACT	ACT	FIFO_OVER-RUN	FIFO_WATER-MARK	FIFO_READY	DATA_READY	0x00	RW	
0x2C	FILTER_CTL	[7:0]	RANGE		RES	HALF_BW	EXT_SAMPLE	ODR			0x13	RW	
0x2D	POWER_CTL	[7:0]	RES	EXT_CLK	LOW_NOISE		WAKEUP	AUTOSLEEP	MEASURE		0x00	RW	
0x2E	SELF_TEST	[7:0]	UNUSED								ST	0x00	RW

4. 기본 실습

연습 1. X 축, Y 축 및 Z 축 가속도 데이터의 최상위 8 비트를 읽고 해당 값을 8 자리 7 세그먼트 디스플레이에 표시하는 RISC-V 어셈블리 프로그램을 만듭니다. 구성 및 등록 정보는 섹션 B 를 참조하십시오. 다음 서브 루틴을 사용하여 SPI 모듈에 액세스합니다. 서브 루틴을 사용하기 전에 SPI 모듈에 대한 섹션 A 에 제공된 정보를 기반으로 이해하십시오. 다음은 각 서브 루틴에 대한 간략한 요약입니다.

- spiInit 기능: SPI 모듈을 초기화합니다.
- spiCS 기능: CS 상태를 SPCS 레지스터로 보냅니다.
- spiCSUp 기능: 서브 루틴 spiCS 를 호출하여 CS 라인을 high 로 당깁니다.
- spiCSDown 기능: 서브 루틴 spiCS 를 호출하여 CS 라인을 low 로 당깁니다.
- spiSendGetData 기능: SPI 를 통해 바이트를 보내고 주변 데이터를 다시 가져옵니다.

```
# Register addresses for SPI Peripheral
```

```
#define SPCR      0x80001100
#define SPSR      0x80001108
#define SPDR      0x80001110
#define SPER      0x80001118
#define SPCS      0x80001120
```

```
# Function: Initialize SPI peripheral
```

```
# call:  by call ra, spiInit
# inputs: None
# outputs: None
# destroys: t0, t1

spiInit:
    li t1, SPCR # control register
    li t0, 0x53 # 01010011 no ints, core enabled, reserved, controller,
                cpol=0, cha=0, clock divisor 11 for 4096
    sb t0, 0(t1)
    li t1, SPER # extension register
    li t0, 0x02 # int count 00 (7:6), clock divisor 10 (1:0) for 4096
    sb t0, 0(t1)
ret
```

```
# Function: Pull CS Line to either high or low - Provides quick calls spiCSUp
            and spiCSDown
# call:  by call ra, spiCS
# inputs: CS status in a0 (0 is low, 1 is high)
# outputs: None
# destroys: t0

spiCS:
    li t0, SPCS # CS register
    sb a0, 0(t0) # Send CS status
ret

spiCSUp:
    li a0, 0x00
    j spiCS

spiCSDown:
    li a0, 0xFF
    j spiCS
```

```
# Function: Send byte through SPI and get the peripheral data back
# call:  by call ra, spiSendGetData
# inputs: data byte to send in a0
# outputs: received data byte in a1
# destroys: t0, t1

spiSendGetData:
internalSpiClearIF: # internal clear interrupt flag
    li t1, SPSR # status register
    lb t0, 0(t1) # clear SPIF by writing a 1 to bit 7
    ori t0,t0,0x80
    sb t0, 0(t1)
internalSpiActualSend:
    li t0, SPDR # data register
    sb a0, 0(t0) # send the byte contained in a0 to spi
internalSpiTestIF:
    li t1, SPSR # status register
    lb t0, 0(t1)
    andi t0, t0, 0x80
    li t1, 0x80
    bne t0,t1,internalSpiTestIF # loop while SPSR.bit7 == 0. (transmission
                                in progress)
internalSpiReadData:
    li t0, SPDR # data register
    lb a1, 0(t0) # read the message from SPI
ret
```

5. 기초 수준의 구현

A. SPI 가속도계 기초 수준 구현

이 LAB의 첫 번째 부분에서는 RVfpga 시스템의 SPI 모듈을 사용하는 방법을 보여주었으며, 이 LAB의 마지막 부분에서는 SPI 모듈이 RVfpga에서 구현되는 방법을 설명합니다. 이전 랩의 형식과 유사하게 SPI 컨트롤러의 분석을 세 단계로 나눕니다.

1. SoC와 가속도계 간의 물리적 연결 (그림 8의 왼쪽 그림자 영역)
2. SweRVolfX 시스템 컨트롤러에 포함된 SPI 컨트롤러 통합 (그림 8의 중간 그림자 영역)
3. SPI 컨트롤러와 SweRV EH1 코어 간의 연결 (그림 8의 오른쪽 그림자 영역)

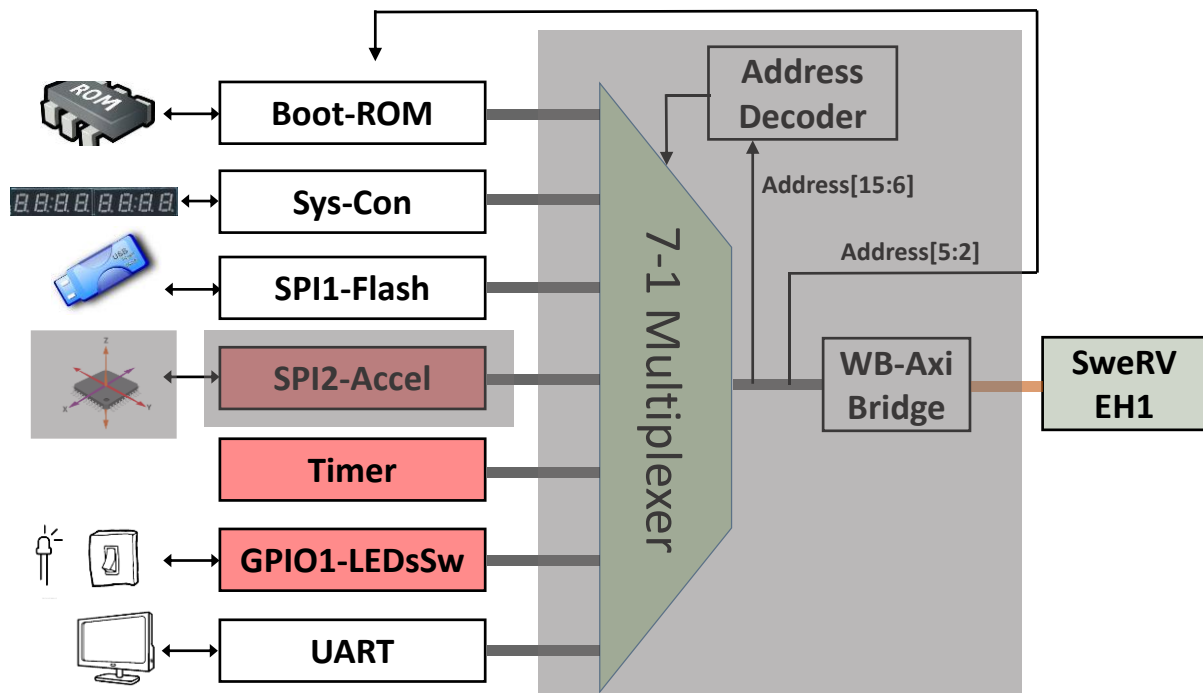


그림 8. RVfpga 시스템에 통합된 SPI 컨트롤러

1. 가속도계와 SoC의 물리적 연결

다른 주변 장치와 마찬가지로 RVfpgaNexys 제약 조건 파일에는 가속도계에 대한 물리적 연결이 포함되어야 합니다. 프로젝트의 constraint 파일 (*[RVfpgaPath]/RVfpga/src/rvfpganexys.xdc*)은 입력/출력 SoC 신호와 보드 장치 간의 연결을 정의합니다. 가속도계의 4 개 핀을 SoC 와 연결하는 신호는 *o_accel_cs_n*, *o_accel_mosi* (신호 SDO 에 해당), *i_accel_miso* (신호 SDI 에 해당) 및 *accel_sclk* 라고 합니다. 이러한 신호는 오래된 이름을 참조하지만 RVfpg 시스템에서 사용하는 OpenCores 의 SPI 모듈에서 사용하는 이름과 일관성을 유지하기 위해 유지합니다 (그림 11 에서 이 모듈의 인스턴스화를 볼 수 있음). 그림 9 는 이러한 4 개의 연결이 정의된 Verilog 코드를 보여줍니다.

```

78 ##Accelerometer
79 set_property -dict { PACKAGE_PIN E15 IOSTANDARD LVCMOS33 } [get_ports { i_accel_miso }]; #IO L11P T1 SRCC 15 Sch=acl_miso
80 set_property -dict { PACKAGE_PIN F14 IOSTANDARD LVCMOS33 } [get_ports { o_accel_mosi }]; #IO L5N T0 AD9N 15 Sch=acl_mosi
81 set_property -dict { PACKAGE_PIN F15 IOSTANDARD LVCMOS33 } [get_ports { accel_sclk }]; #IO L14P T2 SRCC 15 Sch=acl_sclk
82 set_property -dict { PACKAGE_PIN D15 IOSTANDARD LVCMOS33 } [get_ports { o_accel_cs_n }];

```

그림 9. SoC 와 가속도계의 연결 (파일 *rvfpganexys.xdc*).

RVfpgaNexys 시스템의 상단 모듈 (예: **rvfpganexys** 모듈)의 52-55 행에서 SoC (그림 10 의 왼쪽 부분)에 연결된 이 4 개의 신호를 볼 수 있으며 해당 모듈의 끝은 **swervolf_core** 모듈에 (그림 10 의 오른쪽 부분) 연결 됩니다.

```

25 module rvfpga
26     #(parameter bootrom_file = "")
27     (input wire      clk,
28      input wire      rstn,
29      output wire [12:0] ddram_a,
30      output wire [2:0] ddram_ba,
31      output wire      ddram_ras_n,
32      output wire      ddram_cas_n,
33      output wire      ddram_we_n,
34      output wire      ddram_cs_n,
35      output wire [1:0] ddram_dm,
36      inout wire [15:0] ddram_dq,
37      inout wire [1:0] ddram_dqs_p,
38      inout wire [1:0] ddram_dqs_n,
39      output wire      ddram_clk_p,
40      output wire      ddram_clk_n,
41      output wire      ddram_cke,
42      output wire      ddram_odt,
43      output wire      o_flash_cs_n,
44      output wire      o_flash_mosi,
45      input wire      i_flash_miso,
46      input wire      i_uart_rx,
47      output wire      o_uart_tx,
48      inout wire [15:0] i_sw,
49      output reg [15:0] o_led,
50      output reg [7:0] AN,
51      output reg      CA, CB, CC, CD, CE, CF, CG,
52      output wire      o_accel_cs_n,
53      output wire      o_accel_mosi,
54      input wire      i_accel_miso,
55      output wire      accel_sclk);

```

```

248 .o_ram_bready (cpu.b_ready),
249 .i_ram_rid (cpu.r_id),
250 .i_ram_rdata (cpu.r_data),
251 .i_ram_rresp (cpu.r_resp),
252 .i_ram_rlast (cpu.r_last),
253 .i_ram_rvalid (cpu.r_valid),
254 .o_ram_rready (cpu.r_ready),
255 .i_ram_init_done (litedram_init_done),
256 .i_ram_init_error (litedram_init_error),
257 .io_data ({i_sw[15:0],gpio_out[15:0]}),
258 .AN (AN),
259 .Digits_Bits ({CA,CB,CC,CD,CE,CF,CG}),
260 .o_accel_sclk (accel_sclk),
261 .o_accel_cs_n (o_accel_cs_n),
262 .o_accel_mosi (o_accel_mosi),
263 .i_accel_miso (i_accel_miso));
264
265 always @(posedge clk_core) begin
266     o_led[15:0] <= gpio_out[15:0];
267 end
268
269 assign o_uart_tx = 1'b0 ? litedram_tx : cpu_tx;
270
271 endmodule

```

그림 10. 가속도계와 최상위 모듈 (파일 *rvfpganexys.sv*)의 연결.

작업: constraint 파일에서 SweRVolfX SoC 모듈로 이 네 가지 신호 (*o_accel_cs_n*, *o_accel_mosi*, *i_accel_miso* 및 *accel_sclk*)를 따릅니다. 다음 파일을 확인해야 합니다.

[RVfpgaPath]/RVfpga/src/rvfpganexys.xdc

[RVfpgaPath]/RVfpga/src/rvfpganexys.sv

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v

2. SoC 에 SPI2-Accelerometer 모듈 통합

모듈 **swervolf_core** ([RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v) 의 387-403 행에서 가속도계 용 SPI 모듈이 인스턴스화 됩니다 (그림 错误!未找到引用源。 참조).

```

382 // SPI for the Accelerometer
383 wire [7:0] spi2_rdt;
384 assign wb_s2m_spi_accel_dat = {24'd0,spi2_rdt};
385 wire spi2_irq;
386
387 simple_spi spi2
388 (// Wishbone slave interface
389 .clk_i (clk),
390 .rst_i (wb_rst),
391 .adr_i (wb_m2s_spi_accel_adr[2] ? 3'd0 : wb_m2s_spi_accel_adr[5:3]),
392 .dat_i (wb_m2s_spi_accel_dat[7:0]),
393 .we_i (wb_m2s_spi_accel_we),
394 .cyc_i (wb_m2s_spi_accel_cyc),
395 .stb_i (wb_m2s_spi_accel_stb),
396 .dat_o (spi2_rdt),
397 .ack_o (wb_s2m_spi_accel_ack),
398 .inta_o (spi2_irq),
399 // SPI interface
400 .sck_o (o_accel_sclk),
401 .ss_o (o_accel_cs_n),
402 .mosi_o (o_accel_mosi),
403 .miso_i (i_accel_miso));

```

그림 11. 타이머 모듈 통합 (파일 swervolf_core.v).

주변기기와 마찬가지로 모듈의 인터페이스는 Wishbone 신호 (표 6)와 외부 I/O 신호 (표 7)의 두 블록으로 나눌 수 있습니다. Wishbone 신호를 통해 SweRV EH1 Core 가 SPI 프로토콜을 사용하여 ADC 와 통신할 수 있습니다.

표 6. Wishbone 신호

Port	Width	Direction	Description
cyc_i	1	Inputs	유효한 버스사이클 표시 (코어 선택)
adr_i	15	Inputs	주소 입력
dat_i	32	Inputs	데이터 입력
dat_o	32	Outputs	데이터 출력
sel_i	4	Inputs	데이터 버스의 유효한 바이트를 나타냅니다 (유효사이클 동안 0xf 여야 함).
ack_o	1	Output	승인 출력 (정상적인 트랜잭션 종료를 나타냄)
err_o	1	Output	오류 확인 출력 (비정상적인 트랜잭션 종료를 나타냄)
rty_o	1	Output	미사용
we_i	1	Input	높음으로 주장 될 때 트랜잭션 쓰기
stb_i	1	Input	유효한 데이터 전송사이클을 나타냅니다.
inta_o	1	Output	인터럽트 출력

표 7. 외부 I / O 신호

Port	Width	Direction	Description
------	-------	-----------	-------------

miso_i	1	Input	컨트롤러 데이터 입력-주변 데이터 출력
mosi_o	1	Output	컨트롤러 데이터 출력-주변 데이터 입력
ss_o	1	Output	칩 선택
sck_o	1	Output	시스템 시계

그림 11 과 같이 Wishbone 버스 신호 (*wb_m2s_spi_accel_adr [5: 2]*)에서 코어가 제공하는 주소의 비트 [5:2]는 사용 가능한 5 개의 SPI 레지스터 중 하나를 선택하는 데 사용됩니다 (표 1).

3. SPI 컨트롤러와 SweRV EH1 코어 간의 연결

이전 LAB 에서 설명한 것처럼 장치 컨트롤러는 멀티플렉서와 브리지를 통해 SweRV EH1 Core 에 연결됩니다 (그림 8). 7:1 멀티플렉서 (그림 12)는

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.v 파일에서 구현되며 파일 [RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.vh 의 104-205 행에서 인스턴스화 됩니다. 이 후자의 파일은 아래 위치에 있는 **swervolf_core** 모듈의 168 행에 포함되어 있습니다. [RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v

```

108 wb_mux
109 #(.num_slaves (7),
110 .MATCH_ADDR ({32'h00000000, 32'h00001000, 32'h00001040, 32'h00001100, 32'h00001200, 32'h00001400, 32'h00002000}),
111 .MATCH_MASK ({32'hffffff00, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffff00}))
112 wb_mux io
113 (.wb_clk_i (wb_clk_i),
114 .wb_rst_i (wb_rst_i),
115 .wbm_adr_i (wb_io_adr_i),
116 .wbm_dat_i (wb_io_dat_i),
117 .wbm_sel_i (wb_io_sel_i),
118 .wbm_we_i (wb_io_we_i),
119 .wbm_cyc_i (wb_io_cyc_i),
120 .wbm_stb_i (wb_io_stb_i),
121 .wbm_cti_i (wb_io_cti_i),
122 .wbm_bte_i (wb_io_bte_i),
123 .wbm_dat_o (wb_io_dat_o),
124 .wbm_ack_o (wb_io_ack_o),
125 .wbm_err_o (wb_io_err_o),
126 .wbm_rty_o (wb_io_rty_o),
127 .wbs_adr_o ({wb_rom_adr_o, wb_sys_adr_o, wb_spi_flash_adr_o, wb_spi_accel_adr_o, wb_ptc_adr_o, wb_gpio_adr_o, wb_uart_adr_o}),
128 .wbs_dat_o ({wb_rom_dat_o, wb_sys_dat_o, wb_spi_flash_dat_o, wb_spi_accel_dat_o, wb_ptc_dat_o, wb_gpio_dat_o, wb_uart_dat_o}),
129 .wbs_sel_o ({wb_rom_sel_o, wb_sys_sel_o, wb_spi_flash_sel_o, wb_spi_accel_sel_o, wb_ptc_sel_o, wb_gpio_sel_o, wb_uart_sel_o}),
130 .wbs_we_o ({wb_rom_we_o, wb_sys_we_o, wb_spi_flash_we_o, wb_spi_accel_we_o, wb_ptc_we_o, wb_gpio_we_o, wb_uart_we_o}),
131 .wbs_cyc_o ({wb_rom_cyc_o, wb_sys_cyc_o, wb_spi_flash_cyc_o, wb_spi_accel_cyc_o, wb_ptc_cyc_o, wb_gpio_cyc_o, wb_uart_cyc_o}),
132 .wbs_stb_o ({wb_rom_stb_o, wb_sys_stb_o, wb_spi_flash_stb_o, wb_spi_accel_stb_o, wb_ptc_stb_o, wb_gpio_stb_o, wb_uart_stb_o}),
133 .wbs_cti_o ({wb_rom_cti_o, wb_sys_cti_o, wb_spi_flash_cti_o, wb_spi_accel_cti_o, wb_ptc_cti_o, wb_gpio_cti_o, wb_uart_cti_o}),
134 .wbs_bte_o ({wb_rom_bte_o, wb_sys_bte_o, wb_spi_flash_bte_o, wb_spi_accel_bte_o, wb_ptc_bte_o, wb_gpio_bte_o, wb_uart_bte_o}),
135 .wbs_dat_i ({wb_rom_dat_i, wb_sys_dat_i, wb_spi_flash_dat_i, wb_spi_accel_dat_i, wb_ptc_dat_i, wb_gpio_dat_i, wb_uart_dat_i}),
136 .wbs_ack_i ({wb_rom_ack_i, wb_sys_ack_i, wb_spi_flash_ack_i, wb_spi_accel_ack_i, wb_ptc_ack_i, wb_gpio_ack_i, wb_uart_ack_i}),
137 .wbs_err_i ({wb_rom_err_i, wb_sys_err_i, wb_spi_flash_err_i, wb_spi_accel_err_i, wb_ptc_err_i, wb_gpio_err_i, wb_uart_err_i}),
138 .wbs_rty_i ({wb_rom_rty_i, wb_sys_rty_i, wb_spi_flash_rty_i, wb_spi_accel_rty_i, wb_ptc_rty_i, wb_gpio_rty_i, wb_uart_rty_i}));
139
140 endmodule

```

CPU/Controller Signals

Peripheral Signals

그림 12. 7-1 멀티플렉서는 CPU (*wb_intercon.v*)에 연결할 주변 장치를 선택합니다.

멀티플렉서는 주소에 (110-111 라인) 따라 CPU (*wb_io_* * 신호-그림 12 의 115-126 행)를 Wishbone 버스 (그림 12 의 127-138 행)에 연결하여 읽고 쓸 주변 장치를 선택합니다. 예를 들어 CPU 에서 생성된 주소가 0x80001100-0x8000113F 범위에 있으면 가속도계 모듈이 선택되므로 *wb_io_* * 신호는 *wb_spi_accel_* * 신호에 연결됩니다.

6. 고급 실습

연습 2. UART (Universal Asynchronous Receiver-Transmitter)는 비동기 직렬 통신 프로토콜입니다. RVfpga 시스템에는 기본 설계에 UART 모듈이 포함되어 있습니다 (그림 8 참조). 사양은 [RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/uart/docs/UART_spec.pdf에서 확인할 수 있습니다.

SPI 가속도계에 대해 섹션 A 에서 수행한 것과 유사하게 RVfpga 시스템에서 이 모듈의 기초 수준 구현을 먼저 분석합니다.

그런 다음 직렬 포트를 통해 PaltformIO 셀에 메시지를 인쇄하는 RISC-V 어셈블리 프로그램을 만듭니다. 다음 서브 루틴을 사용하여 UART 모듈에 액세스합니다. 서브 루틴을 사용하기 전에 이해하려고 노력하십시오. 다음은 각 서브 루틴에 대한 간략한 요약입니다.

- `uartInit` 함수: UART 모듈을 초기화합니다.
- `uartSendByte` 함수: UART 를 통해 바이트를 보냅니다.
- `uartSendString` 함수: UART 를 통해 문자열을 보냅니다.

```
# Register addresses for UART Peripheral
# -----

#define CONSOLE_ADDR 0x80001008
#define HALT_ADDR    0x80001009
#define UART_BASE    0x80002000

#define REG_BRDL (4*0x00) /* Baud rate divisor (LSB) */
#define REG_IER  (4*0x01) /* Interrupt enable reg. */
#define REG_FCR  (4*0x02) /* FIFO control reg. */
#define REG_LCR  (4*0x03) /* Line control reg. */
#define REG_LSR  (4*0x05) /* Line status reg. */
#define LCR_CS8  0x03 /* 8 bits data size */
#define LCR_1_STB 0x00 /* 1 stop bit */
#define LCR_PDIS 0x00 /* parity disable */

#define LSR_THRE 0x20
#define FCR_FIFO 0x01 /* enable XMIT and RCVR FIFO */
#define FCR_RCVCLR 0x02 /* clear RCVR FIFO */
#define FCR_XMITCLR 0x04 /* clear XMIT FIFO */
#define FCR_MODE0 0x00 /* set receiver in mode 0 */
#define FCR_MODE1 0x08 /* set receiver in mode 1 */
#define FCR_FIFO_8 0x80 /* 8 bytes in RCVR FIFO */
```

```
.section .data

welcome:
.string "\nHELLO WORLD !!!\n"
```

```
# Function: Initialize UART peripheral
# call: by call ra, uartInit
# inputs: None
# outputs: None
# overwrites: t0, t1
# -----

uartInit:
    li      t0, UART_BASE

    /* Set DLAB bit in LCR */
    li      t1, 0x80
    sb      t1, REG_LCR(t0)

    /* Set divisor regs */
    li      t1, 27
    sb      t1, REG_BRDL(t0)

    /* 8 data bits, 1 stop bit, no parity, clear DLAB */
    li      t1, LCR_CS8 | LCR_1_STB | LCR_PDIS
    sb      t1, REG_LCR(t0)
```

```

li    t1, FCR_FIFO | FCR_MODE0 | FCR_FIFO_8 | FCR_RCVRLR | FCR_XMITCLR
sb    t1, REG_FCR(t0)

/* disable interrupts */
sb    zero, REG_IER(t0)

ret

```

```

# Function: Send byte through UART
# call: by call ra, uartSendByte
# inputs: a0, byte to be sent
# outputs: None
# destroys: t0, t1
# -----

```

```

uartSendByte:
    li t1, UART_BASE

    /* Check for space in UART FIFO */
    lb t0, REG_LSR(t1)
    andi t0, t0, LSR_THRE
    beqz t0, uartSendByte
    sb a0, 0(t1)

    ret

```

```

# Function: Send string through UART (terminated by \0)
# call: by call ra, uartSendString
# uses: uartSendByte
# inputs: a0, address of first character of string to be sent
# outputs: None
# destroys: t0, t1, t2
# -----

```

```

uartSendString:
    li t1, UART_BASE
    add t2,zero,ra # save caller address
    add a1,zero,a0 # use a1 as index
    /* Load first byte */
    lb a0, 0(a1)

internalNextChar:
    call ra, uartSendByte
    addi a1, a1, 1
    lb a0, 0(a1)
    bne a0, zero, internalNextChar

    add ra,zero,t2 # restore caller address
    ret

```

연습 3. C 언어로 다음 세 가지 함수를 구현합니다.

- `char uart_getchar(void)`: 이 함수는 키보드가 UART 를 통해 Nexys A7 보드로 문자를 보낼 때까지 기다린 다음 이 문자를 출력 매개 변수로 반환합니다. 문자는 ASCII 코드 (<https://www.ascii-code.com/>) 로 표시됩니다.
- `int uart_putchar(char c)`: 이 함수는 문자를 입력 인수로 받아 UART 를 통해 직렬 콘솔에 표시합니다. WD 의 BSP (Western Digital 의 보드 지원 패키지)에서 제공하는 `printfNexys` 기능을 사용하는 대신 UART 레지스터에 액세스하는 자체 기능을 구현해야 합니다.

- `int SevSegDispl(char c):` 이 함수는 문자를 입력 인수로 받아 7 세그먼트 디스플레이의 가장 오른쪽 자리에 표시하고 나머지 자리를 왼쪽으로 한 자리 이동합니다 (가장 왼쪽 자리는 손실 됨). 7 세그먼트 디스플레이는 0-9, A, B, C, D, E 및 F 문자만 표시하므로 다른 문자에 대해서는 0 을 표시할 수 있습니다. 여러분은 Lab7-연습 3 에서 구현된 7 세그먼트 디스플레이 확장 컨트롤러를 이용하여, 더 많은 문자를 표시하는 연습을 할 수 있습니다.

처음 두 기능을 구현하려면 `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/uart/docs/UART_spec.pdf` 에 있는 UART 모듈 사양 문서를 사용해야 합니다.

위의 세 가지 기능을 기반으로 키보드에서 문자를 받아 직렬 터미널과 7 세그먼트 디스플레이 모두에 표시하는 프로그램을 C 로 만듭니다.

UART 모듈 초기화를 위해 WD 의 BSP 에서 제공하는 `uartInit` 함수를 사용할 수 있습니다.

연습 4. 또 다른 일반적인 직렬 통신 프로토콜은 I2C ("eye two see" 또는 "eye squared see"로 발음)라고 합니다. Nexys A7 보드의 온도 센서는 이 프로토콜을 사용합니다. RVfpga 시스템을 확장하여 I2C 컨트롤러를 포함하고 Nexys A7 보드의 ADT7420 온도 센서 (<https://www.analog.com/media/en/technical-documentation/data-sheets/adt7420.pdf>) 와 연결합니다. 그런 다음 이 새 주변 장치와 통신하고 7 세그먼트 디스플레이에 온도를 표시하는 프로그램을 작성하십시오.