Multi-Port I2C Master Controller

Version 0 Revision 1

Setp. 10, 2025 (Oct. 1, 2022)

Future Design Systems, Inc.
www.future-ds.com / contact@future-ds.com

Abstract

This document describes specifications of I2C master controller supporting multiple I2C ports.

Table of Contents

1 Overview

'i2c_master_axi_lite' is an I2C bus (Inter-IC bus) controller.

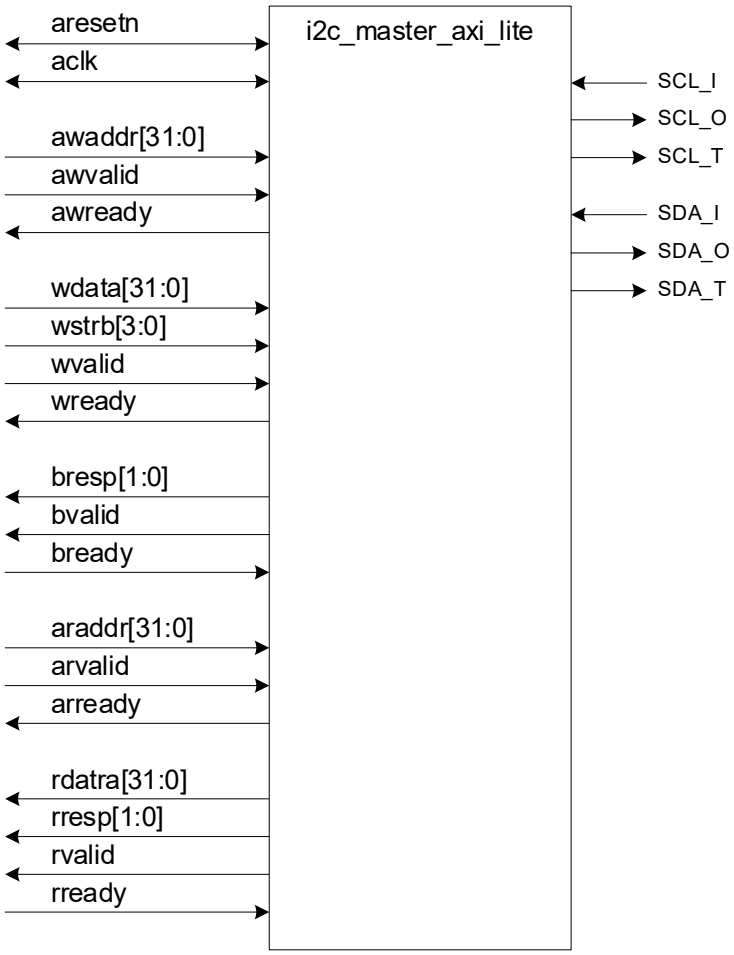

Figure 1: Overview

2 Structure

Figure 2 shows conceptual block diagram of the i2c_master_axi_lite.

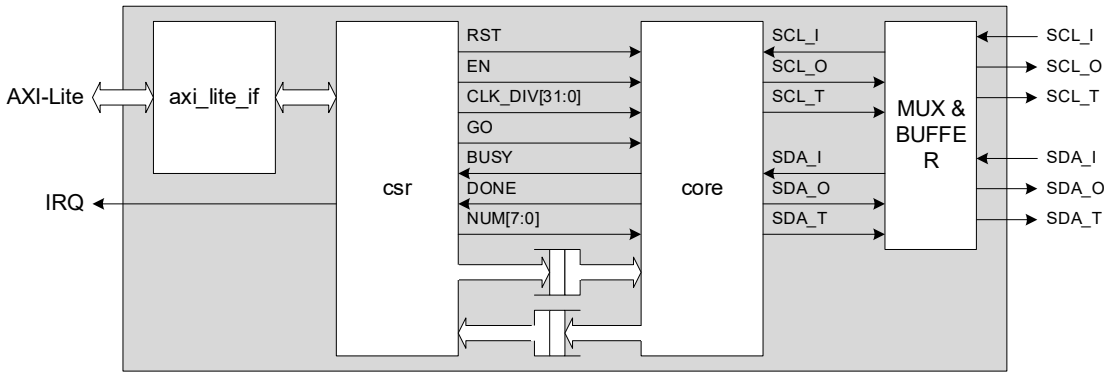

Figure 2: Internal structure

Figure 3 shows how I/O/T signals control interface buffer, where '_T' enables output when 'low'.

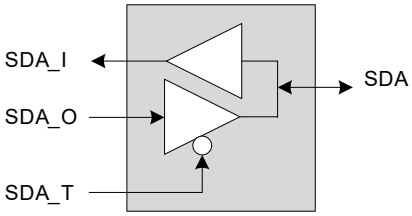

Figure 3: I/O/T signal control

2.1 Macros and parameters

Table 1: Macros

Macros	Note
SIM	pure RTL simulation (do not define 'SIM' and 'SYN' at the same time)
SYN	logic synthesis, which makes use of Xilinx specific things
RIGOR	rigorously check for simulation use this with 'SIM
ISE	for Xilinx ISE devices such as Spartan-6, Virtex-6
VIVADO	for Xilinx Vivado device such as Zynq-7000, UltraScale, UltraScale+

Table 2: Parameters

Parameter	Note
P_CLK_FREQ	Frequency of PCLK ● 100_000_000 for 100MHz
P_I2C_SPEED	Default frequency of SCL; Actual frequency can be changed through CSR. ● 400_000 for 400kHz
P_RX_DEPTH	Depth of FIFO for receiving I2C response and data ● 8
P_TX_DEPTH	Depth of FIFO for sending I2C command and data.

	● 8
P_I2C_NUM	Number of I2C port ● 3: 3 ports

2.2 Configuration and status register

Name	Address offset	Bit#	description
VERSION	+00h	RO	RTL version (default: 0x2018_1001)
		31:0	RTL version
CONTROL	+04h	RW	
		31	RST I2C part reset when 1 Should be written 0 to return normal
		20-2	reserved
		1	IE (Interrupt enabled when 1)
		0	EN (Enable I2C part when 1)
STATUS	+08h	RO	
		31	RST of I2C part
		30	BUSY of I2C ● remain 1 from start to stop
		29-28	reserved
		27-24	reserved (ID for multi-master case)
		23-16	P_RX_DEPTH
		15-8	P_TX_DEPTH
		7-2	reserved
		1	IP
		0	EN
CONFIG	+0Ch	RW	
		31-28	P_I2C_NUM
		27-16	reserved
		15-12	GAP (>=1) the number of SCL cycles between consecutive I2C transactions, i.e., between STOP to START. see Figure 4
		11-8	POST (>=1) see Figure 4
		7-4	MID (>=1) see Figure 4
		3-0	PRE (>=1) see Figure 4
SEL	+10h		
		31-4	Reserved
		3-0	Select I2C port

CLK_FREQ	+20h		RO	PCLK frequency
			31:0	PCLK frequency 100_000_000 for 80Mhz
I2C_SPEED	+24h		RW	SCL frequency
			31:0	SCL frequency 400_000: 400kHz
CLK_DIV	+28h		RO	Clock division
			31:0	Clock division for SCL PCLK/SCL
Reserved	+2Ch			
RUN	+30h		RW	
			31-24	I2C CMD NUM
			23-16	the number of items in receiving FIFO
			15-8	the number of rooms in sending FIFO
			7-2	reserved
			1	I2C_DONE
			0	GO This block run I2C protocol when it is 1. It returns 0 on completion.
Reserved	+34h			
Reserved	+38h			
Reserved	+3Ch			
TX_DATA	+40h			
			31-13	reserved
			12	repeat flag Expecting START without STOP, i.e., Sr
			11	stop flag Make a stop condition, i.e., P It always comes with the last data.
			10	receive flag receive data
			9	drive flag Drive data
			8	start flag Make a START condition, i.e., S It always comes with ‘drive’ for control byte.
			7-0	I2C data
Reserved	+44h			
Reserved	+48h			
Reserved	+4Ch			
RX_DATA	+50h		RO	CMD FIFO status
			31-9	reserved
			8	acknowledge ● 1 for ACK in general

				● 0 for NACK for the last read
			7-0	I2C data
Reserved	+54h			
Reserved	+58h			
Reserved	+5Ch			

‘CONFIG’ register should not change while ENABLE is high.

3 I2C protocols

3.1 I2C timing

PRE, MID, POST, and GAP are counted by SCL cycles.

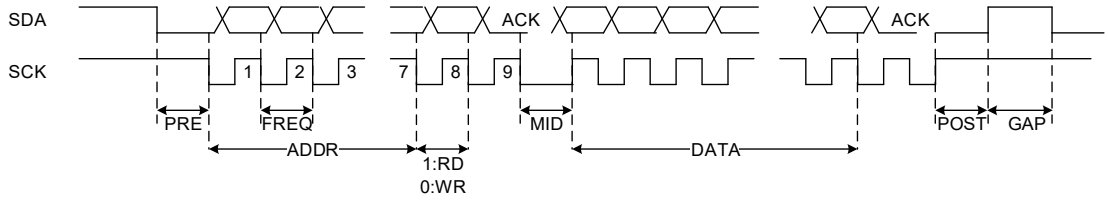

Figure 4: I2C timing configuration

SDA should not change during SCK is high while transferring data excepting STAT and STOP condition.

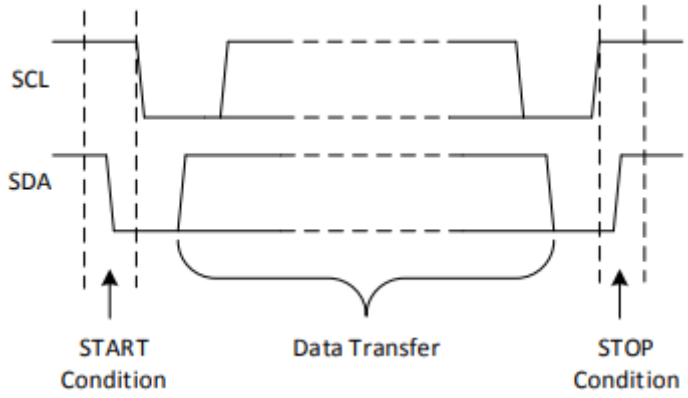

Figure 5: I2C START and STOP condition

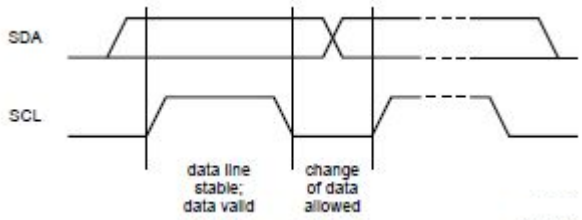

Figure 6: I2C bit transfer

All I2C transaction starts with control byte.

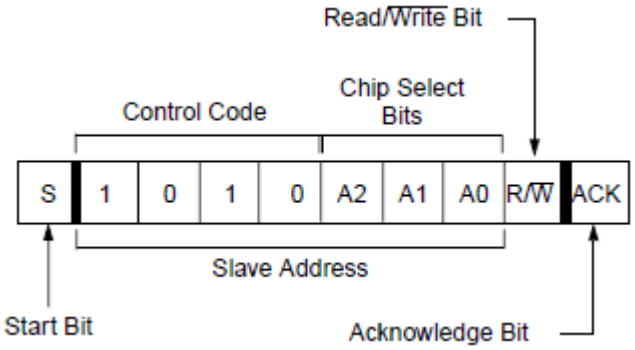

Figure 7: General form of control byte for 7-bit device address

3.2 7-bit device address 8-bit register address I2C

3.2.1 Single write / random write

```
/* send device address */
dataW ← {repeat(0),stop(0),receive(0), drive(1), start(1),dev_addr[6:0],0}
CSRA_I2C_TX_DATA ← dataW
/* send register address */
dataW ← {repeat(0),stop(0),receive(0), drive(1), start(0),reg_addr[7:0]}
CSRA_I2C_TX_DATA ← dataW
/* send register data for write */
dataW ← {repeat(0),stop(1),receive(0), drive(1), start(0),reg_data[7:0]}
CSRA_I2C_TX_DATA ← dataW
/* let go I2C  transaction by setting Go bit 1 */
dataW ← {3<<I2C_RUN_num,I2C_RUN_go_MSK}
/* wait for GO bit zero */
do { dataR ← CSRA_I2C_RUN; } while (dataR&I2C_RUN_go_MSK);
/* may need time to complete write on the device */
```

3.2.2 Single read / random read

```
/* send device address */
dataW ← {repeat(0),stop(0),receive(0), drive(1), start(1),dev_addr[6:0],0}
```

```
CSRA_I2C_TX_DATA ← dataW
/* send register address */
dataW ← {repeat(1),stop(0),receive(0), drive(1), start(0),reg_addr[7:0]}
CSRA_I2C_TX_DATA ← dataW
/* send device address */
dataW ← {repeat(0),stop(0),receive(0), drive(1), start(1),dev_addr[6:0],1}
CSRA_I2C_TX_DATA ← dataW
/* send register data for read */
dataW ← {repeat(0),stop(1),receive(1), drive(0), start(0)}
CSRA_I2C_TX_DATA ← dataW
/* let go I2C  transaction by setting Go bit 1 */
dataW ← {3<<I2C_RUN_num,I2C_RUN_go_MSK}
/* wait for GO bit zero */
do { dataR ← CSRA_I2C_RUN; } while (dataR&I2C_RUN_go_MSK);
```

3.2.3 Burst write / sequential write

3.2.4 Burst read / sequential read

3.3 7-bit device address 16-bit register address I2C

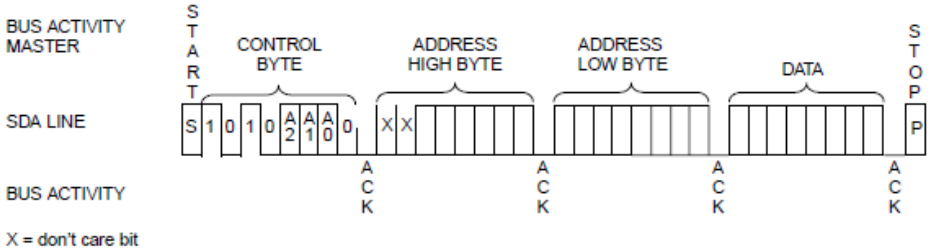

Figure 8: Byte write

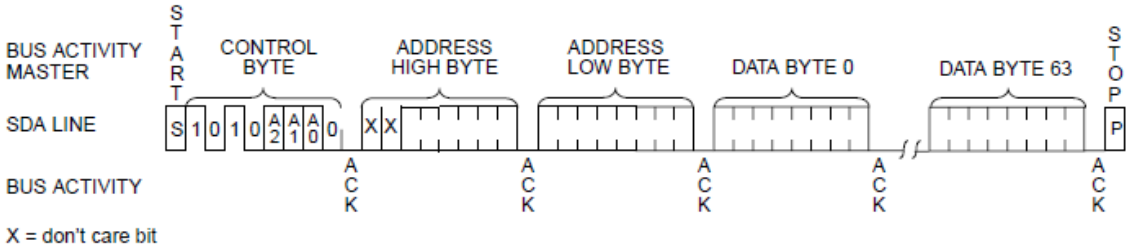

Figure 9: Page write

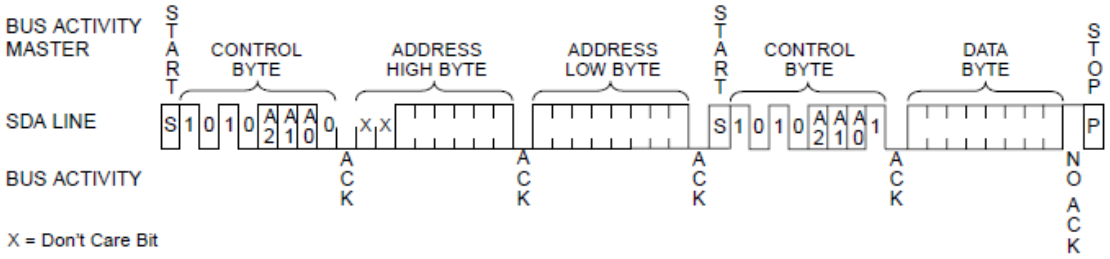

Figure 10: Random read

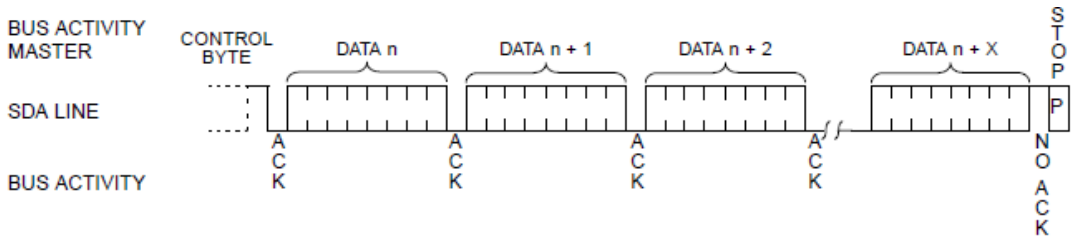

Figure 11: Sequential read

3.4 10-bit device address I2C

To be added.

4 Control API

As shown in Figure 12, API consists of three levels, where low-level API provides read/write functions to access registers in the I2C master controller, mid-level API provides control functions, and high-level API provides functions to access I2C registers.

High level API (It refers to the I2C registers of I2C device)	i2c_read_d7r8(); i2c_write_d7r8()
Mid level API (It controls I2C contoller)	i2c_rst(); i2c_enable(); i2c_disable(); i2c_sel(); i2c_config(); i2c_init()
Low level API (It refers to the registers of I2C controller)	i2c_csr_read() i2c_csr_write()

Figure 12: API levels

4.1 i2c_init()

It sets SCL speed.

```
int i2c_init ( unsigned int  i2c_freq );
```

- i2c_freq: SCL frequency in Hz (400Khz by default).

4.2 i2c_enable() and i2c_disable()

It enables or disables this block.

```
int i2c_enable( void );
int i2c_disable( void );
```

4.3 i2c_config()

It configures the way how SCL and SDA works.

```
int i2c_config ( unsigned int gap
               , unsigned int post
               , unsigned int mid
               , unsigned int pre );
```

- gap: the number of SCL cycles between consecutive I2C packets.
- post: the number of SCL cycles between the end of SCL and the end of SDA.
- mid:
- pre: the number of SCL cycles between starting of SDA and starting of SCL.

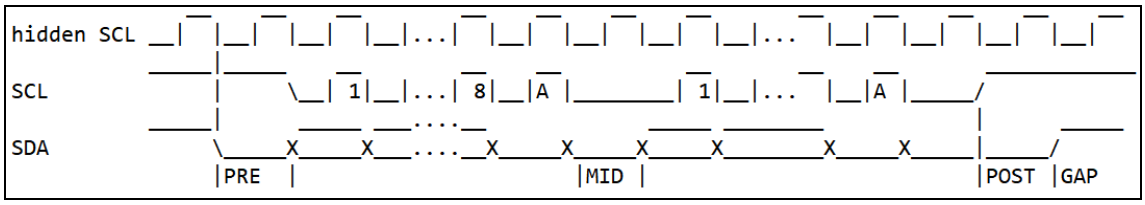

Figure 13: I2C cycle format

4.4 i2c_sel()

It selects which I2C port.

```
int i2c_sel ( unsigned int sel );
```

- sel: the port ID, 0 by default.

4.5 i2c_read_d7r8()

It reads 8-bit data from I2C device register, where I2C device is specified by 7-bit I2C address and the register is 8-bit data.

```
int i2c_read_d7r8( unsigned char  dev_addr
                 , unsigned char  reg_addr
```

```
                , unsigned char *reg_data )
```

- dev_addr: device address in lower 7-bit
- reg_addr: register address to read
- reg_data: pointer to the buffer to be filled after read.

4.6 i2c_write_d7r8()

It writes 8-bit value to I2C device register, where I2C device is specified by 7-bit I2C address and the register is 8-bit data.

```
int i2c_read_d7r8( unsigned char  dev_addr
                , unsigned char  reg_addr
                , unsigned char *reg_data )
```

- dev_addr: device address in lower 7-bit
- reg_addr: register address to read
- reg_data: pointer to the buffer to be written.

4.7 Typical usage

Following code shows a typical usage of API.

```
i2c_rst(); // optional
i2c_disable(); // optional
i2c_init(400000); // optional
i2c_config(1, 1, 1, 1); // optional

i2c_sel(0);
i2c_enable()
i2c_read(dev_addr, reg_addr, &reg_data);
i2c_write(dev_addr, reg_addr, &reg_data):

i2c_disable()
i2c_rst();
```

5 Real chip cases

5.1 PCM1865 I2C

PCM1865 uses 0x4A (100_1010) in 7-bit format.

Figure 14 shows an example of reading register of PCM1865 through I2C port, where repeated start is used between two consecutive I2C transactions.

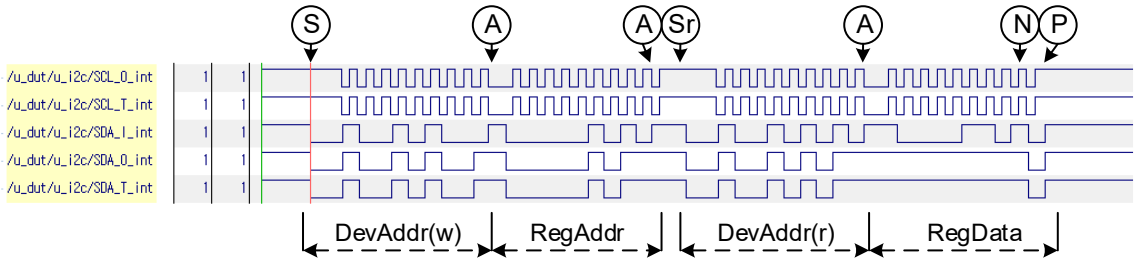

Figure 14: PCM1865 I2C read page 0 register 5

Figure 15 shows an example of writing register of PCM1865 through I2C port, where register data byte follows register address.

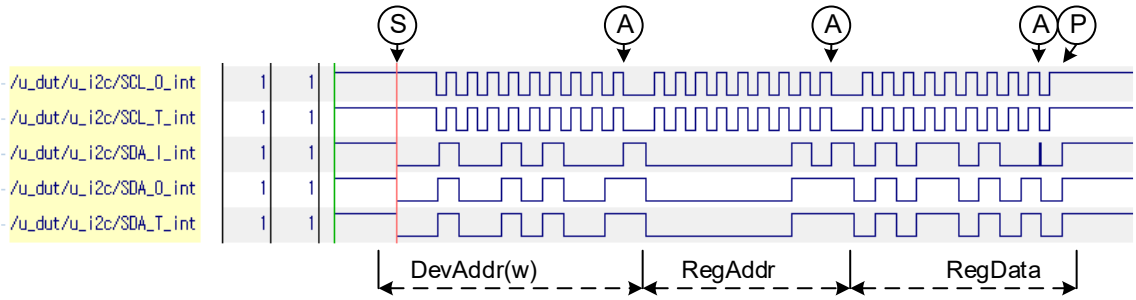

Figure 15: PCM1865 I2C write page 0 register 1

5.2 M24C0

M24C0 uses 0x51 (101_0001) in 7-bit format.

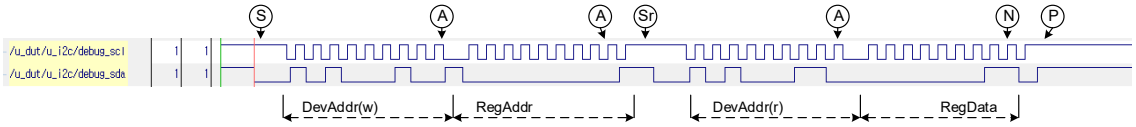

Figure 16: M24C0 I2C read register 0

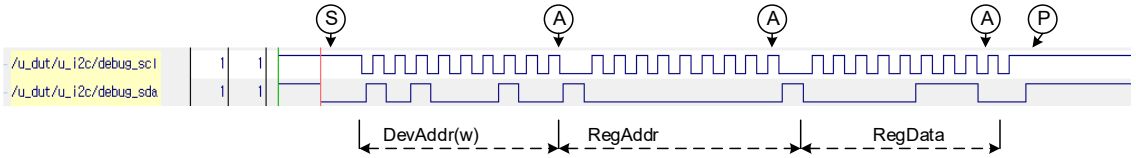

Figure 17: M24C0 I2C write register 0

5.3 24LC128

24LC128 uses 0x51 (101_0001) in 7-bit format with 16-bit register address.

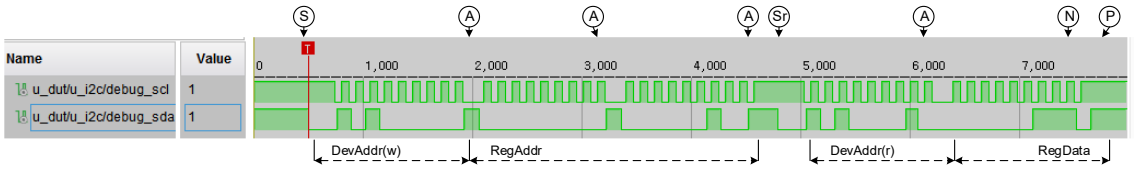

Figure 18: 24LC128 I2C read register 2

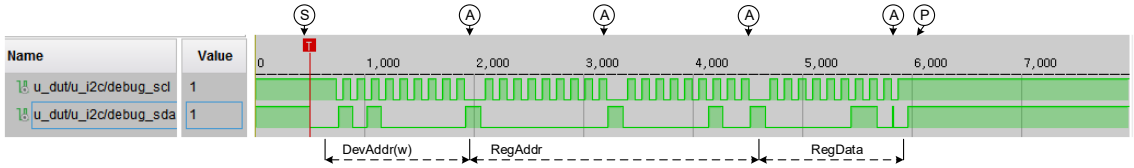

Figure 19: 24LC128 I2C write register 1

6 References

[1] NXP, I2C-bus Specification and User Manual, UM10204, Rev. 6, 4 April 2014.

Wish list

Revision history

- ☐ 2025.09.10: Updated by Ando Ki.
- ☐ 2022.10.01: Started by Ando Ki (adki@future-ds.com)

– End of document –