CSCE 491
Computer Systems Engineering

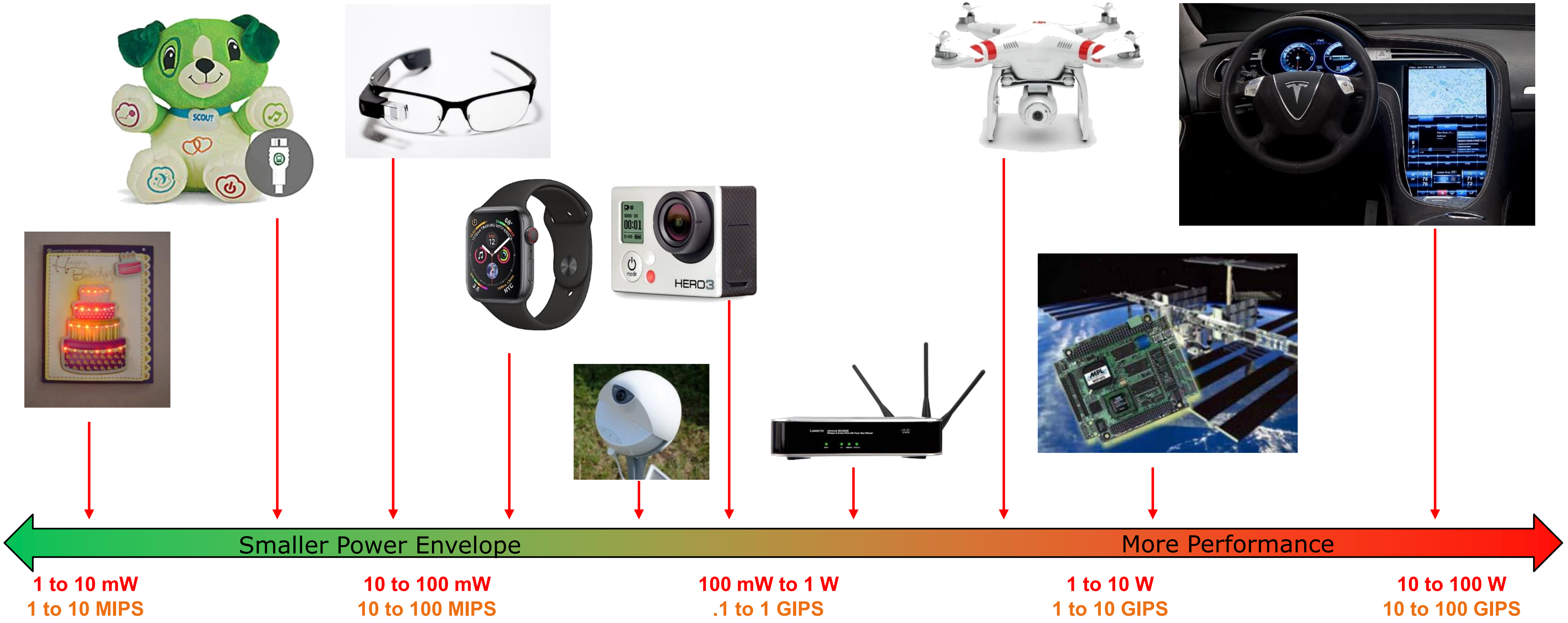
Introduction and HW-SW Interfacing

UofSC South Carolina

Embedded Cyberphysical Systems

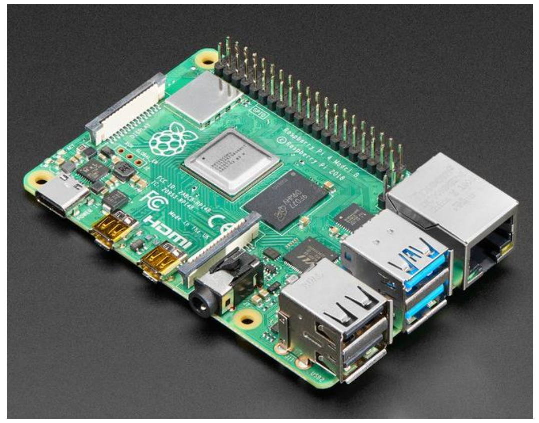
- Embedded system: Computer system (hw+sw) customized for a specific function
- Cyberphysical system: Embedded system that links sensing, computation, and control to the physical world

Tradeoff: Power Envelope vs Performance

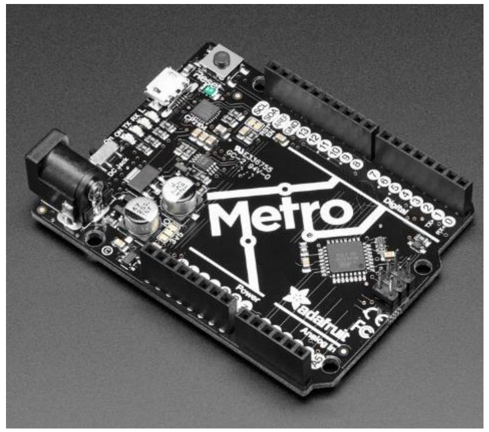


Tradeoff: Power Envelope vs Efficiency

Summit supercomputer
200 Petaflops per second
10 Megawatts
20 Gigaflops per second/watt



Raspberry Pi 4
27.2 Gigaflops per second
15 watts
1.8 Gigaflops per second/watt



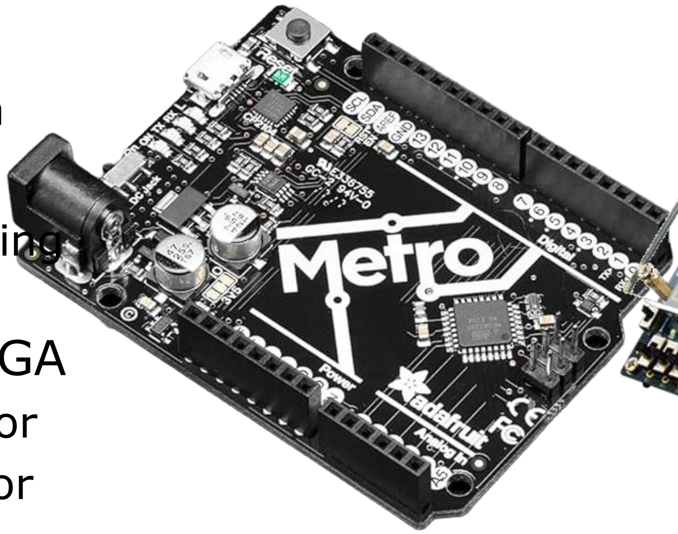
Arduino/ATmega328
~200 Kiloflops per second
20 milliwatts[1]
10 Megaflops per second/watt[2]

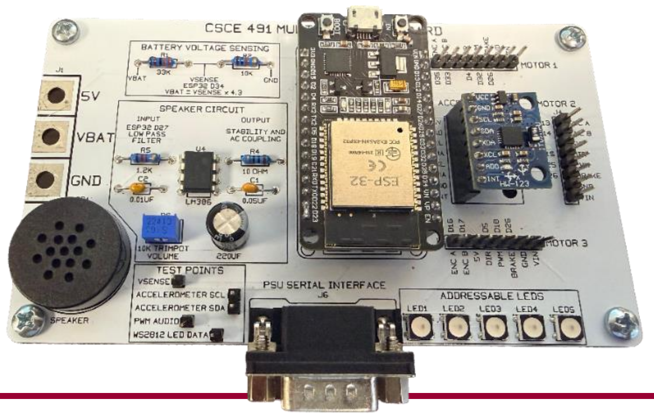
7,000,000X less performant
700,000X less power
11X less power efficient

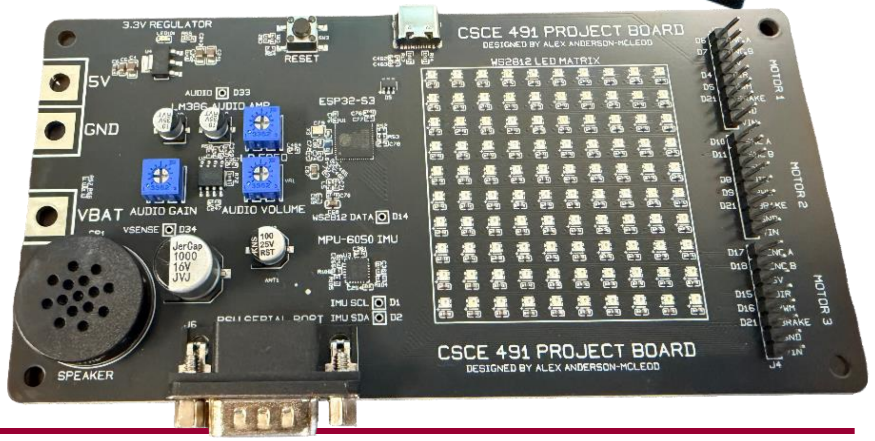
136X less performant
750X less power
180X less power efficient

CSCE 491

- Spring 2020: Atmel ATmega328
  - 8-bit microcontroller, 32 KB of on-chip flash
- Spring 2021: online course; concepts only
  - Control theory, real-time scheduling, queueing theory, CMOS design
- Spring 2022/2023, Fall 2023: DE2-115 FPGA
  - Designed PWM peripheral for Nios2 processor
  - Controlled LED dimming, controller DC motor
- Spring 2025: Espressif ESP32
  - 32-bit dual-core microcontroller
  - Custom PCB
  - Arduino IDE
- Spring 2026: Espressif ESP32-S3
  - 32-bit dual-core microcontroller
  - Custom PCB
  - Arduino IDE





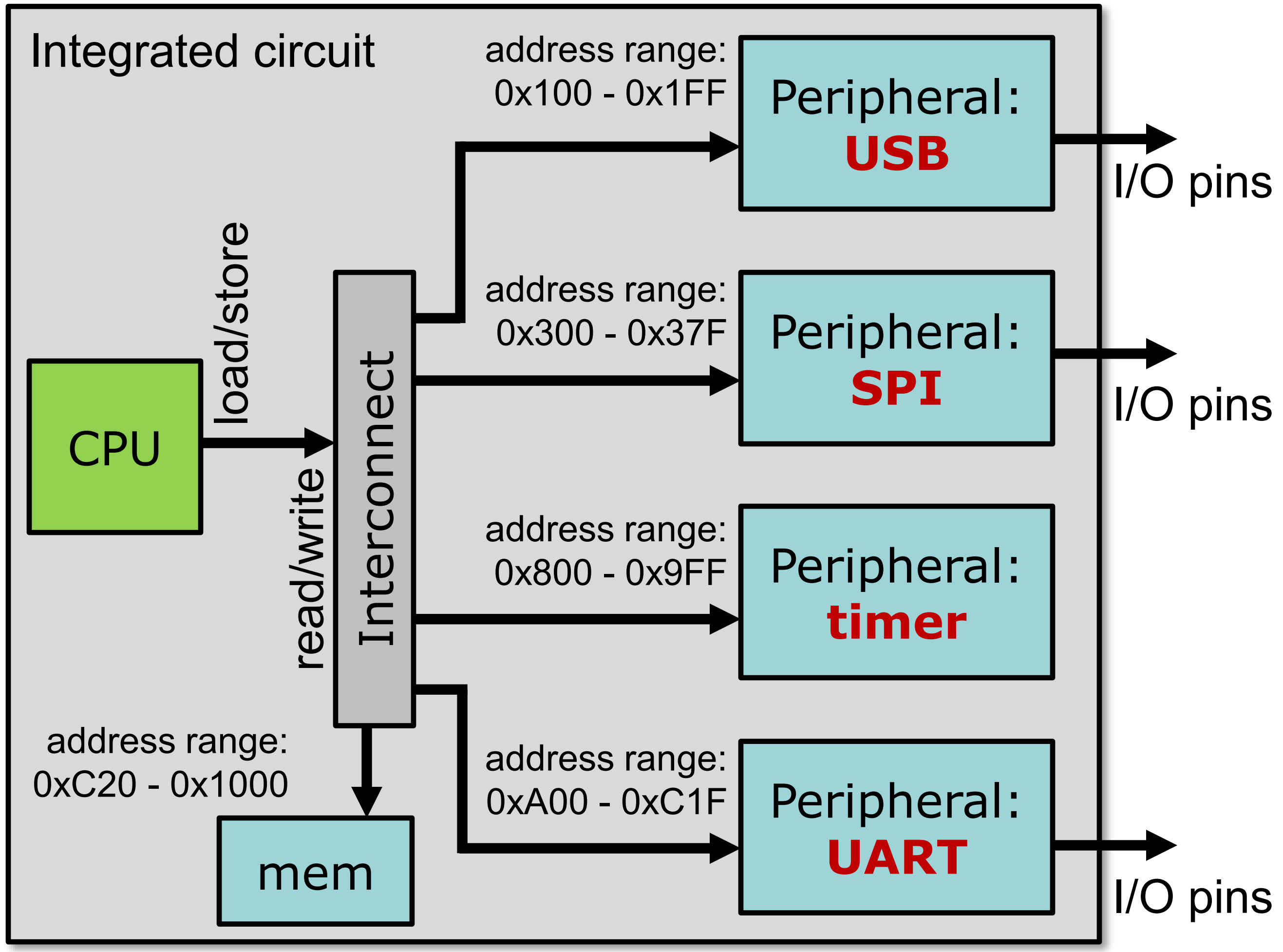


Topics

1. Hardware-software interfacing
2. Control theory
3. Real-time scheduling

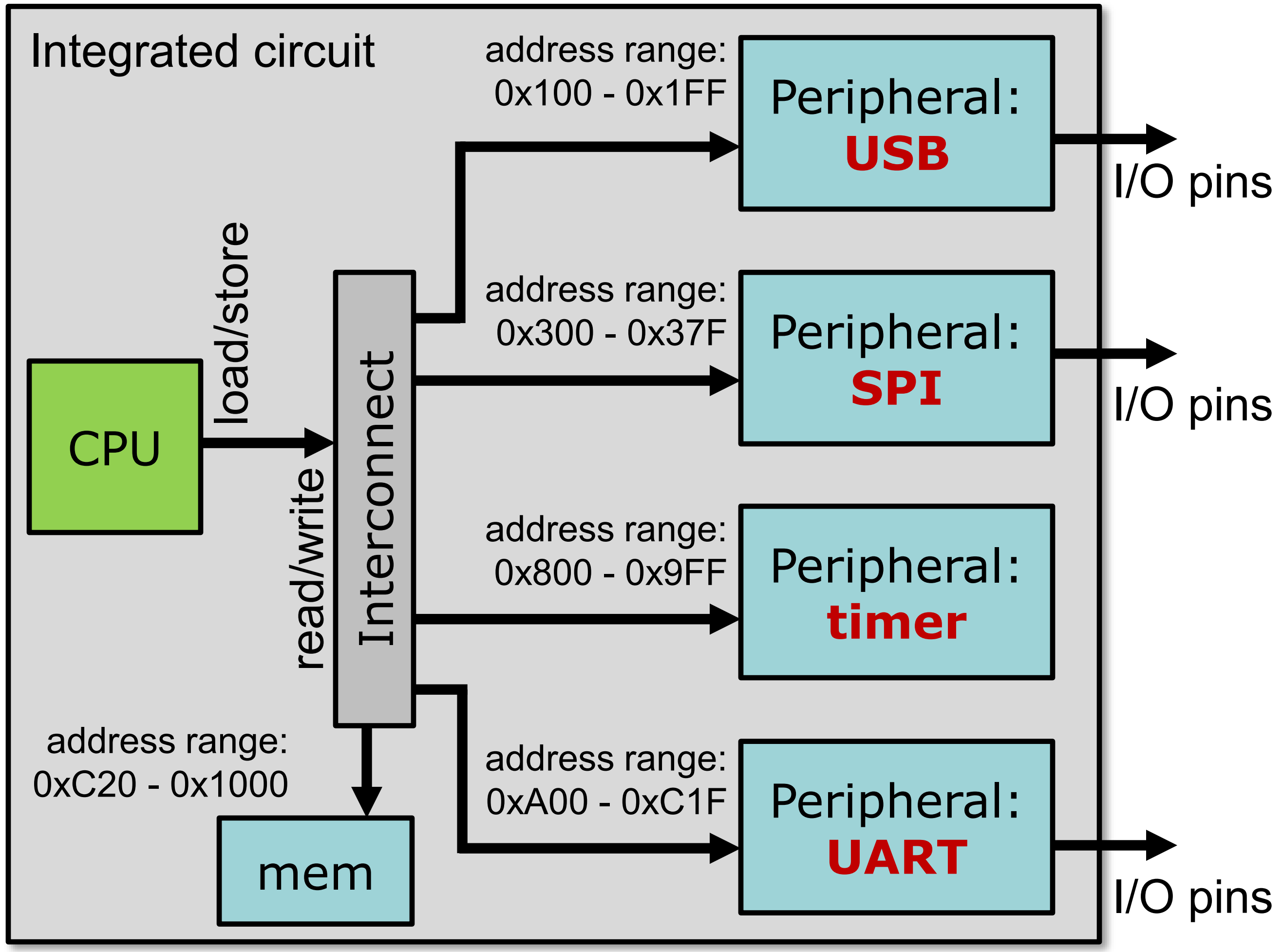
- Hardware-software interacing
  - From software perspective:
    - Programmed I/O
    - Interrupts
    - Direct memory access (DMA)
  - From hardware perspective:
    - I/O protocols (e.g. SPI, I2C, CAN, I2C, UART, JTAG)

Programmed I/O



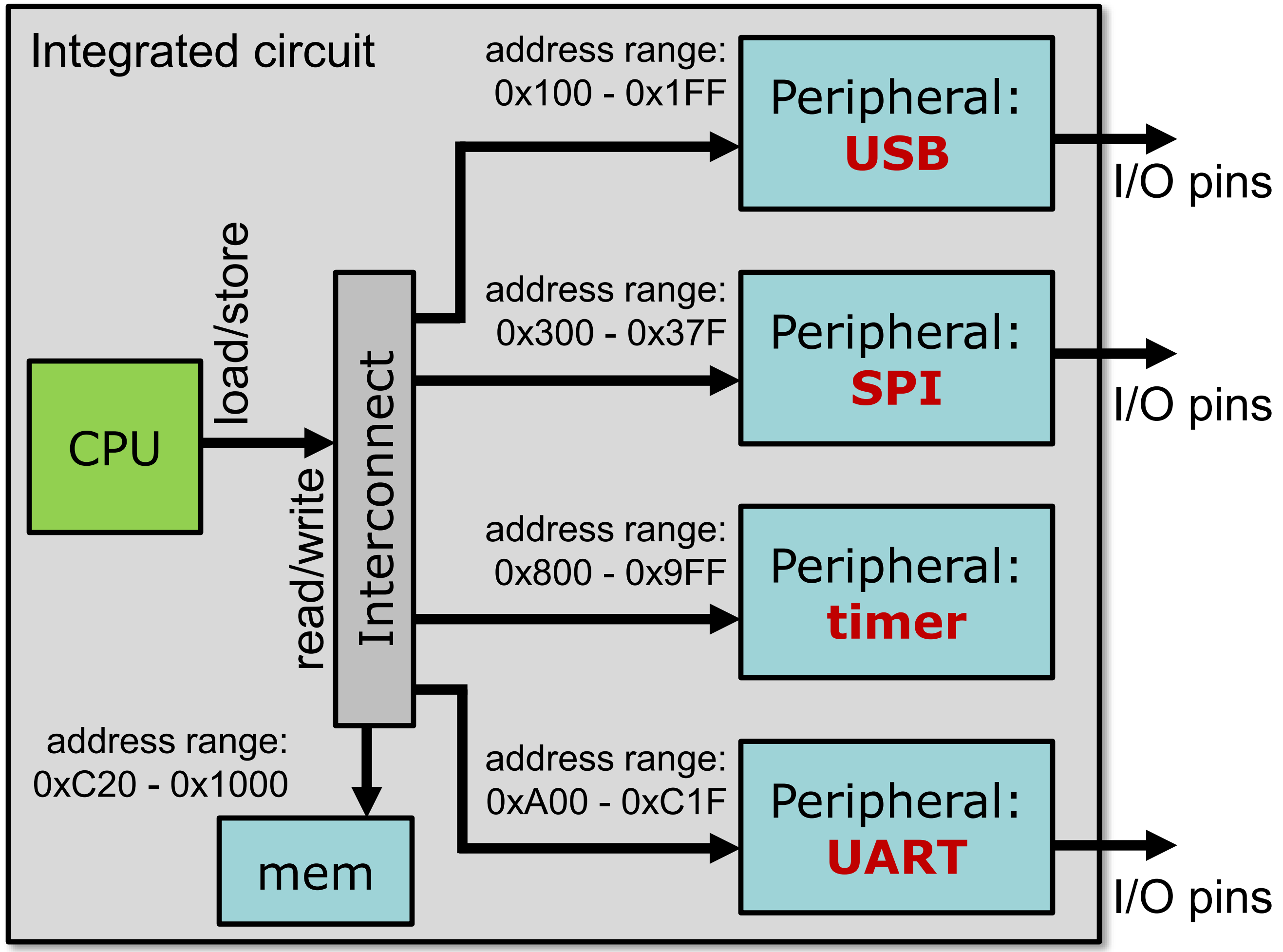
- CPU interacts with peripherals with cache-bypassing load/store instructions that access locations mapped within peripherals
- CPU initiates, peripheral responds
- **I/O registers:**
  - "Status" or "control" registers
  - Poll the state of a peripheral or tell the peripheral to do something (change its state)
  - Specific to each peripheral (need to refer to user guide/datasheet)

Programmed I/O



- What is the size of the USB peripheral's I/O space?
- Answer:
  - 0x1FF – 0x100 + 1 = 0x100
  - = 256 bytes

Programmed I/O



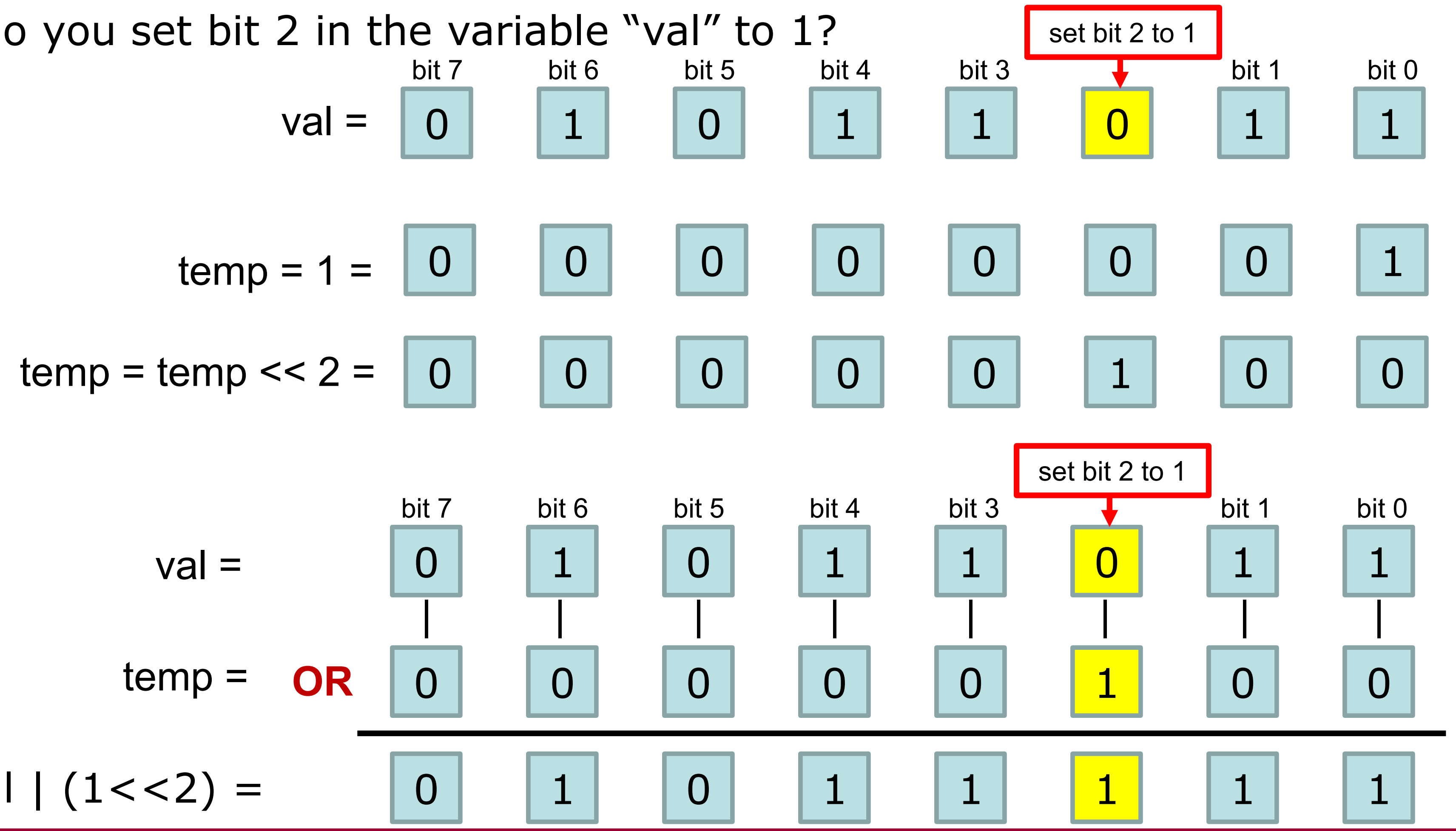
Bit	7	6	5:3	2	1	0
Name	HALT	ONFIRE	TEMP	AA	BB	CC
Mode	w	r	r	w	r	r/w

HALT: set to 1 to halt CPU
ONFIRE: flag that determines if chip is on fire
TEMP[2:0]: 000=cold, 001=toasty, 010 to 110=reserved, 111=white hot

- Assume the USB peripheral has a control/status register at offset 0x10 with the fields shown above
- How can we check the value of ONFIRE and set HALT to 1?

Bit Twiddling: Set a Specific Bit

- How do you set bit 2 in the variable "val" to 1?



val = val | (1<<2) =

Bit Twiddling: Clear a Specific Bit

- How do you set bit 2 in the variable "val" to 0?

set bit 2 to 0

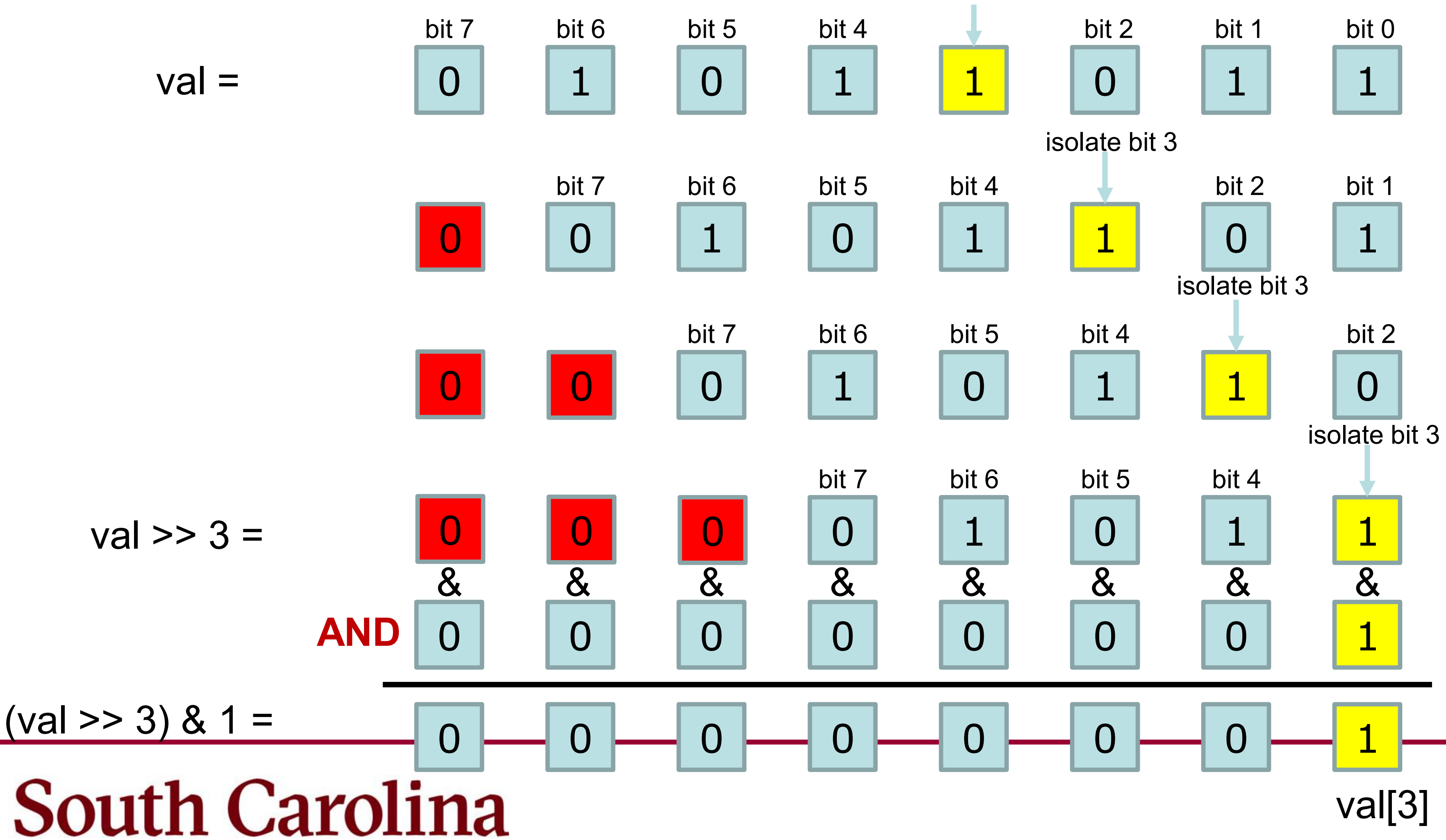
	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
val =	0	1	0	1	1	1	1	1
temp = 1<<2 =	0	0	0	0	0	1	0	0
temp = ~temp =	1	1	1	1	1	0	1	1

set bit 2 to 0

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
val =	0	1	0	1	1	1	1	1
	&	&	&	&	&	&	&	&
temp = **AND**	1	1	1	1	1	0	1	1
val = val & ~(1<<2) =	0	1	0	1	1	0	1	1

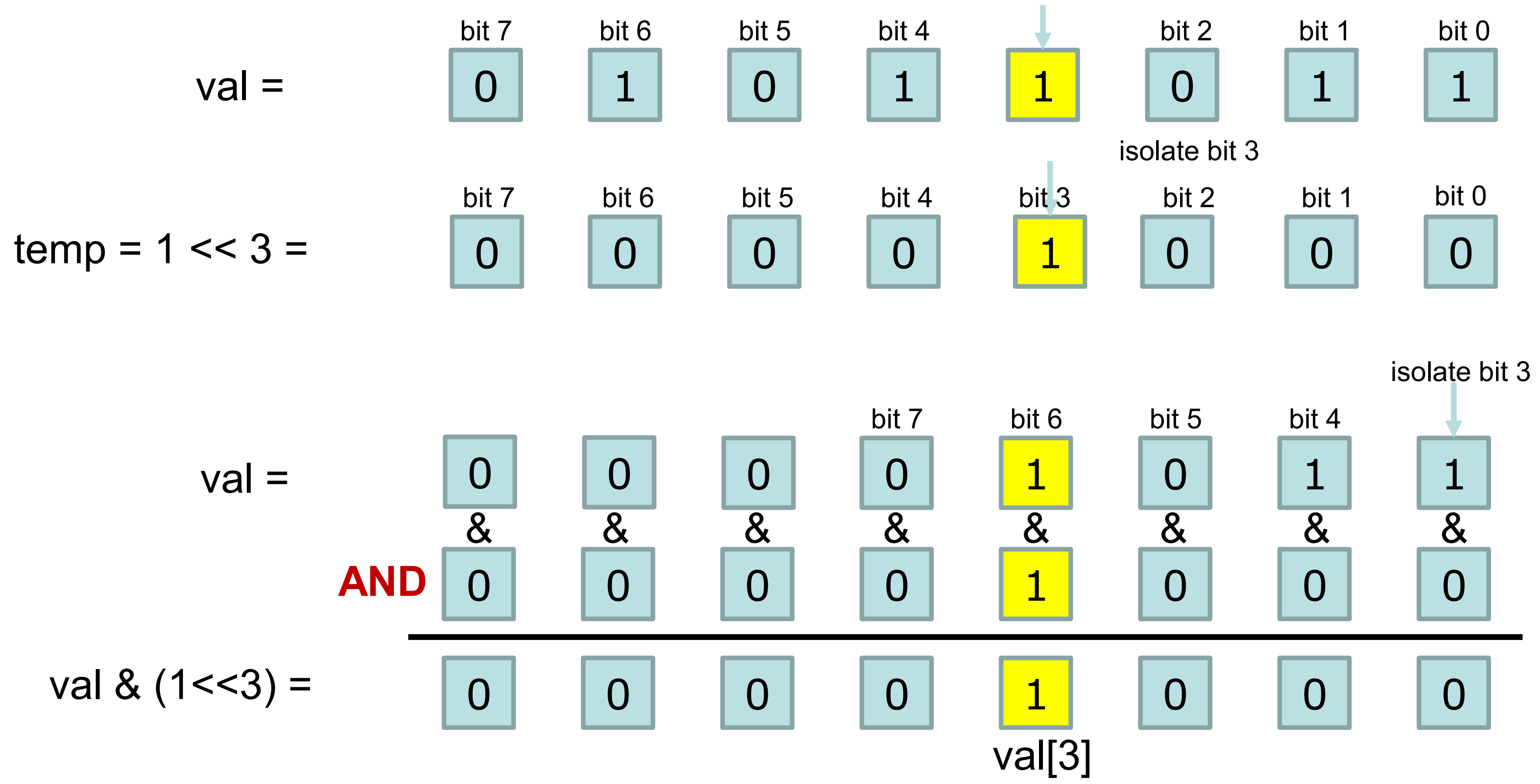
Bit Twiddling: Isolate a Specific Bit Value as LSB

- How do you isolate bit 3 in the variable "val"?



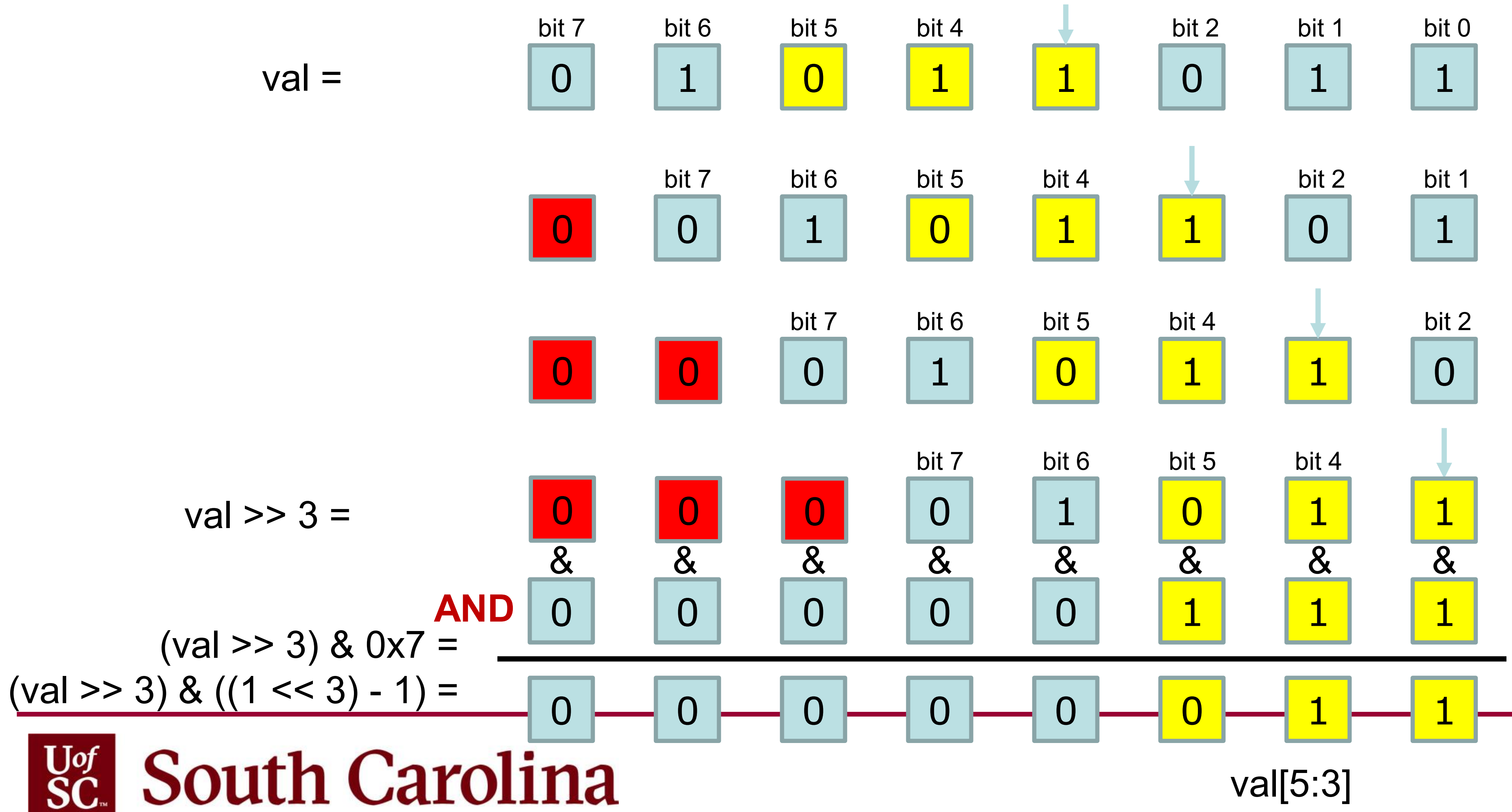
Bit Twiddling: Isolate a Specific Bit Value as Boolean

- How do you isolate bit 3 in the variable "val"?



Bit Twiddling: Isolate a Specific Bit Range as LSB

- How do you isolate bits 5:3 in the variable “val”?



Bit Twiddling: Clear and Set a Bit Field

- How do you set bits 5:3 in the variable “val” to value 4 (0b100)?

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
val =	0	1	0	1	1	0	1	1
(1<<3)-1 << 3 =	0	0	1	1	1	0	0	0
~((1<<3)-1 << 3) =	1	1	0	0	0	1	1	1

Bit Twiddling: Clear and Set a Bit Field

- How do you set bits 5:3 in the variable “val” to value 4 (0b100)?

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
val =	0	1	0	1	1	0	1	1
&	1	1	0	0	0	1	1	1
=	0	1	0	0	0	0	1	1
0b100 << 3	0	0	1	0	0	0	0	0
=	0	1	1	0	0	0	1	1

Summary of Bit Twiddling Ops

- Set bit N to 1:

```
val = val | (1<<N);
```

- Set bit N to 0:

```
val = val & ~(1<<N);
```

- Isolate/extract/read bits N:M (where N >= M):

```
val = (val >> M) & (1 << N-M+1) - 1;
```

- Quick way to test a flag located in bit N:

```
if (val & (1 << N)) {
  …
}
```

Bit Twiddling

Bit	7	6	5:3	2	1	0
Purpose	HALT	ONFIRE	TEMP	AA	BB	CC

PNTLS

```
#define USBADDR        0x100
#define PNTLSREG       0x10
#define HALT           7
#define ONFIRE         6
#define TEMP           3
```

- To read a flag:

```
volatile uint8_t *PNTLS = USBADDR + PNTLS;
int onfire = (REG_READ(PNTLS) & (1 << ONFIRE)) >> ONFIRE;
int temp = (REG_READ(PNTLS) >> 3) & ((1 << 3)-1);
```

- To set the HALT bit to 1 or 0:

```
REG_WRITE(PNTLS,REG_READ(PNTLS) | (1 << HALT)); // set to 1
REG_WRITE(PNTLS,REG_READ(PNTLS) & ~(1<<HALT)); // set to 0
```

Operator Precedence

- ~, +/-, <</>>, &/| (highest to lowest)

- Set a 3-bit field whose LSB is 11 to value f:

```
val = reg & ~1<<3-1<<11 | f << 11;
```

1 3 2 4 6 5

- Where do we need to add parentheses?

Operator Precedence

- ~, +/-, <</>>, &/| (highest to lowest)
  - All of these are evaluated left-to-right

- Set a 3-bit field whose LSB is 11 to value f:

```
val = reg & ~((1<<3)-1<<11) | f << 11;
            4    1    2  3    6    5
```

Structure Bit Fields

```
struct IO_MUX_GPIOn_REG {
   union {
      uint32_t raw;
      struct {
         uint32_t MCU_OE : 1;
         uint32_t SLP_SEL : 1;
         uint32_t MCU_WPD : 1;
         uint32_t MCU_WPU : 1;
         uint32_t MCU_IE : 1;
         uint32_t MCU_DRV : 2;
         uint32_t FUN_WPU : 1;
         uint32_t FUN_IE : 1;
         uint32_t FUN_DRV : 2;
         uint32_t MCU_SEL : 3;
         uint32_t res : 17;
      } fields;
   };
};
```

To use:

```
struct IO_MUX_GPIOn_REG my_reg;
my_reg.raw = REG_READ(IO_MUX_GPIO12_REG);
my_reg.fields.SLP_SEL = 1;
REG_WRITE(IO_MUX_GPIO12_REG,my_reg.raw);
```

Load and Store in C

- Load value at address "addr":
  - Dereference on RHS

```
<target> = *addr;
```

- Store value at address "addr":
  - Dereference on LHS

```
*addr = <val>;
```

- Some tools require the address to be declared as "volatile" pointer to bypass cache

```
volatile uint32_t *addr;
```

Load and Store in C

- ESP32 has custom macros
  - For accessing control/status regsters:

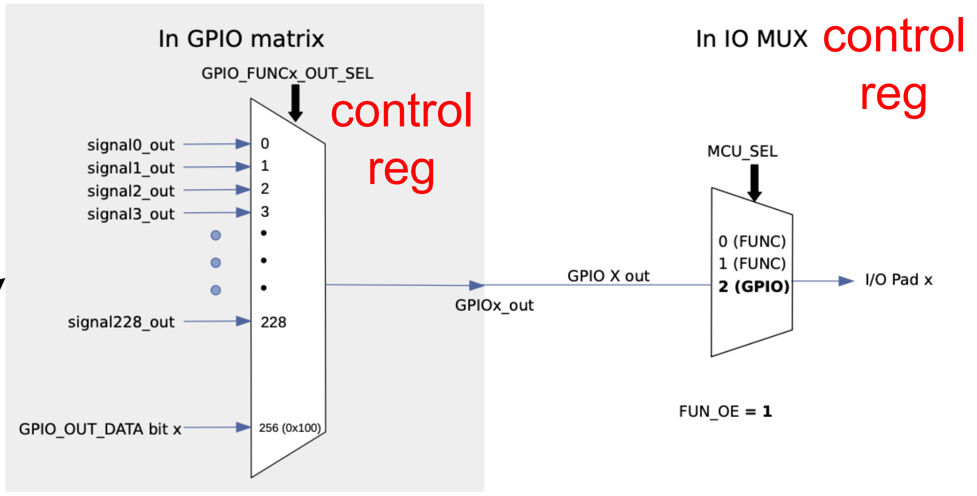
```
REG_READ(addr);
REG_WRITE(addr,val);
```

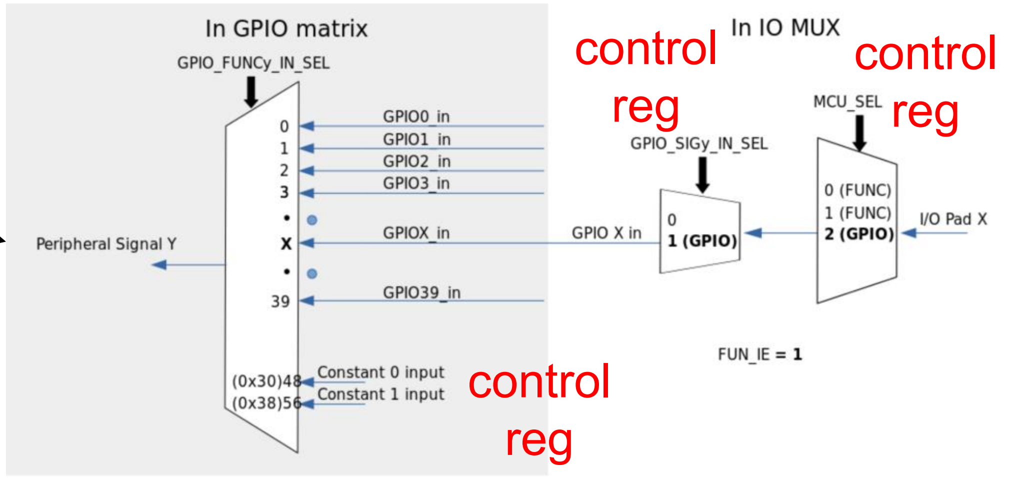
  - For setting/clearing bits in a value:

```
REG_SET_BIT(addr, 1<<3);
REG_CLR_BIT(addr, (0b111 << 12));
```

Example

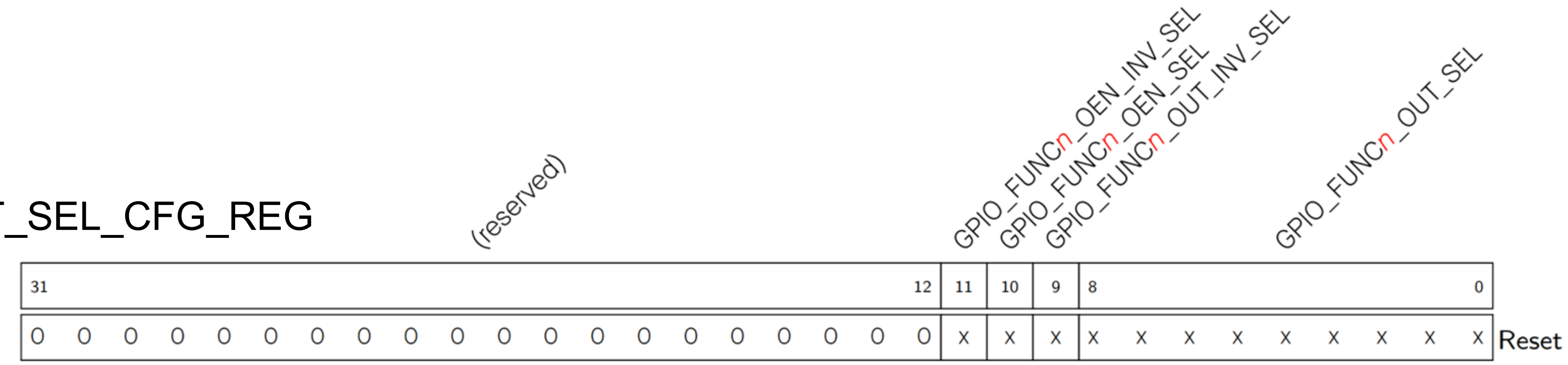
- ESP32 has:
  - 48 pins
    - Of these, 40 are user-configurable ("GPIO pins")
    - 40 GPIO_out signals and 40 GPIO_in signals
- ESP32 has a 228 x 40 "GPIO matrix" that allows:
  - Any of 190 internal signals to drive any of the ESP32's 40 GPIO_out pins
  - Any of the ESP32's 40 GPIO_in to drive any of 173 internal signals
- Each pin is controlled by 2 IO muxes (one for input and one for output)





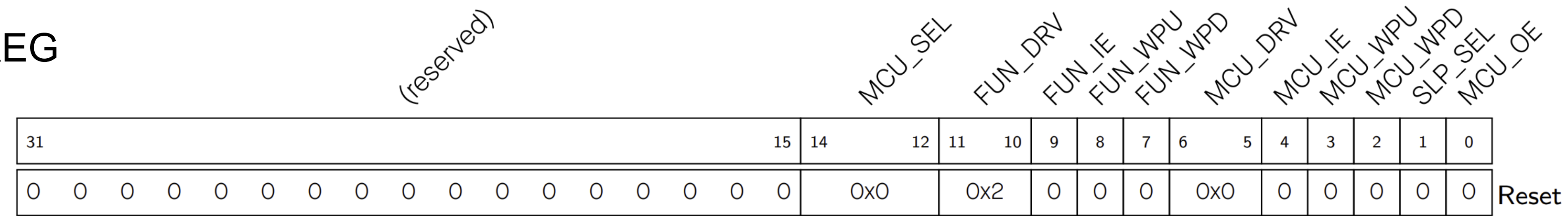
Bit Twiddling

GPIO_FUNC12_OUT_SEL_CFG_REG



	(reserved)	GPIO_FUNCn_OEN_INV_SEL	GPIO_FUNCn_OEN_SEL	GPIO_FUNCn_OUT_INV_SEL	GPIO_FUNCn_OUT_SEL
	31 … 12	11	10	9	8 … 0
Reset	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	x	x	x	x x x x x x x x x

IO_MUX_GPIO12_REG

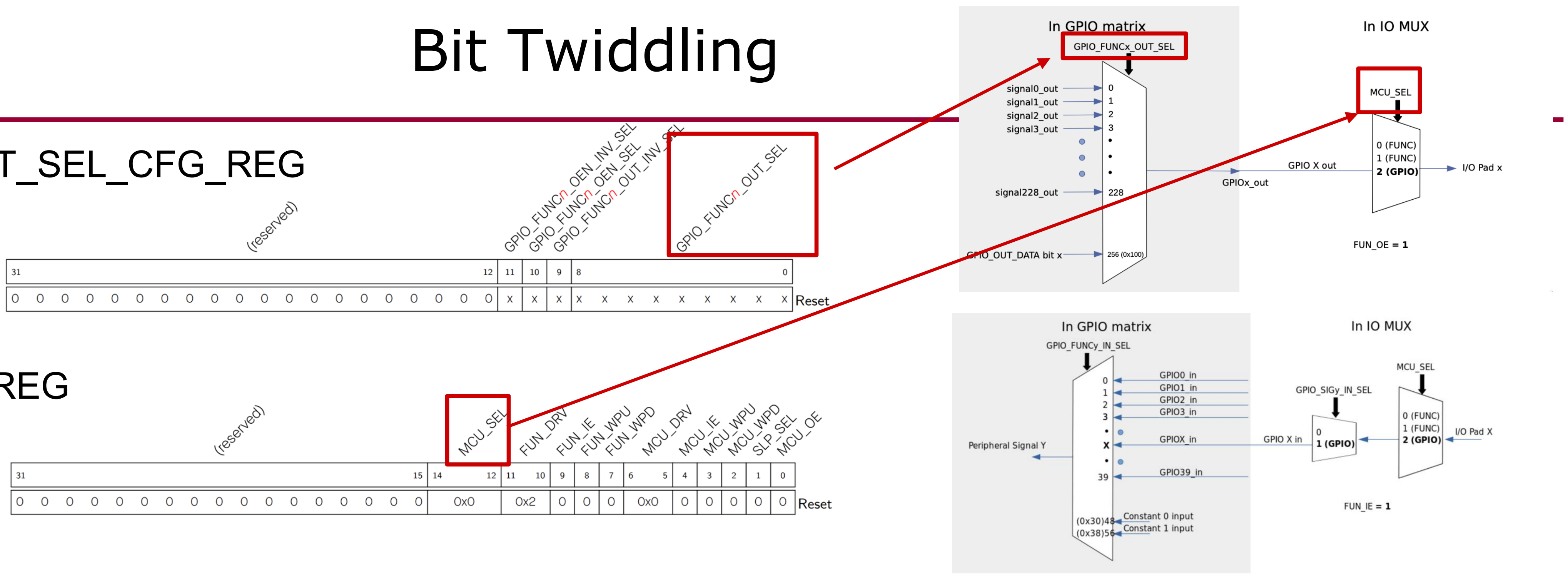


	(reserved)	MCU_SEL	FUN_DRV	FUN_IE	FUN_WPU	FUN_WPD	MCU_DRV	MCU_IE	MCU_WPU	MCU_WPD	SLP_SEL	MCU_OE
	31 … 15	14 … 12	11 … 10	9	8	7	6 … 5	4	3	2	1	0
Reset	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0x0	0x2	0	0	0	0x0	0	0	0	0	0

Bit Twiddling

GPIO_FUNC12_OUT_SEL_CFG_REG

IO_MUX_GPIO12_REG



	(reserved)	GPIO_FUNCn_OEN_INV_SEL	GPIO_FUNCn_OEN_SEL	GPIO_FUNCn_OUT_INV_SEL	GPIO_FUNCn_OUT_SEL
Bits	31 … 12	11	10	9	8 … 0
Reset	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	x	x	x	x x x x x x x x x

	(reserved)	MCU_SEL	FUN_DRV	FUN_IE	FUN_WPU	FUN_WPD	MCU_DRV	MCU_IE	MCU_WPU	MCU_WPD	SLP_SEL	MCU_OE
Bits	31 … 15	14 … 12	11 … 10	9	8	7	6 … 5	4	3	2	1	0
Reset	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0x0	0x2	0	0	0	0x0	0	0	0	0	0

- There is one GPIO_FUNCn_OUT_SEL_CFG_REG register for each of the 40 pins
- There is one GPIO_FUNCn_IN_SEL_CFG_REG register for each of the 40 pins
- There is one IO_MUX_GPIOn_REG for each of the 40 pins

Example

- Goal: connect a "simple" IO output to pin 12
  - Set GPIO_FUNC12_OUT_SEL_CFG_REG.GPIO_FUNC12_OUT_SEL to 256
  - Set IO_MUX_GPIO12_REG.MCU_SEL to 3

```
#define GPIO_FUNC12_OUT_SEL_CFG_REG          0x3FF44560
#define IO_MUX_GPIO12_REG                    0x3FF49034
#define GPIO_OUT_DATA                        0x3FF44004

REG_CLR_BIT(GPIO_FUNC12_OUT_SEL_CFG_REG,(1<<9) - 1);
REG_SET_BIT(GPIO_FUNC12_OUT_SEL_CFG_REG,256);

REG_CLR_BIT(IO_MUX_GPIO12_REG,(1<<3) - 1<<12);
REG_SET_BIT(IO_MUX_GPIO12_REG,3<<12);
REG_SET_BIT(GPIO_OUT_DATA,1<<12);
```

Bit Twiddling

Write a line of C/C++ code that sets the value of bit 4 to the value 0 in the variable named myvar, leaving the other bits unchanged.

Bit Twiddling

Write a line of C/C++ code that sets the value of bit 4 to the value 0 in the variable named myvar, leaving the other bits unchanged

```
myvar = myvar & ~(1<<4);
```

Bit Twiddling

Consider the following snippet of C code:

```
uint8_t x;
uint8_t y = ((x >> 4) | ((~(x & 0x0f )) << 4));
```

Indicate which of the following lines of C code will cause the same value to be stored in the variable "y".

```
a. uint8_t y = ~((x & 0x0f ) << 4) | ((x & 0xf0) >> 4);

b. uint8_t y = ((x >> 4) | ((~(x)) << 4));

c. uint8_t y = ~(~(x >> 4) | (((~x)) << 4));

d. uint8_t y = (((x << 4) & 0xf0) | x >> 4 & 0xf0);
```

Bit Twiddling

Consider the following snippet of C code:

```
uint8_t x;
uint8_t y = ((x >> 4) | ((~(x & 0x0f )) << 4));
```

bits 7:4	bits 3:0
~x[3:0]	x[7:4]

Indicate which of the following lines of C code will cause the same value to be stored in the variable "y".

a. `uint8_t y = ~((x & 0x0f ) << 4) | ((x & 0xf0) >> 4);`

bits 7:4	bits 3:0
~x[3:0]	~x[7:4]

b. `uint8_t y = ((x >> 4) | ((~(x)) << 4));`

bits 7:4	bits 3:0
~x[3:0]	x[7:4]

c. `uint8_t y = ~(~(x >> 4) | (((~x)) << 4));`

bits 7:4	bits 3:0
x[3:0]	x[7:4]

d. `uint8_t y = (((x << 4) & 0xf0) | x >> 4 & 0xf0);`

bits 7:4	bits 3:0
x[3:0]	0000

Topics So Far...

- Programmed I/O
  - Obtaining I/O register addresses
  - Loading and storing I/O registers

```
REG_READ(addr);
REG_WRITE(addr,val);
REG_SET_BIT(addr, 1<<3);
REG_CLR_BIT(addr, (0b111 << 12));
```

- Bit "twiddling"
  - Sitting individual bits and bit fields
  - Extracting bit fields
- How to convey analog outputs from a digital pin: PWM
- How peripherals can affect their own "agency": interrupts
- How to communicate with external peripherals: SPI, I2C
- How to debug hardware: JTAG

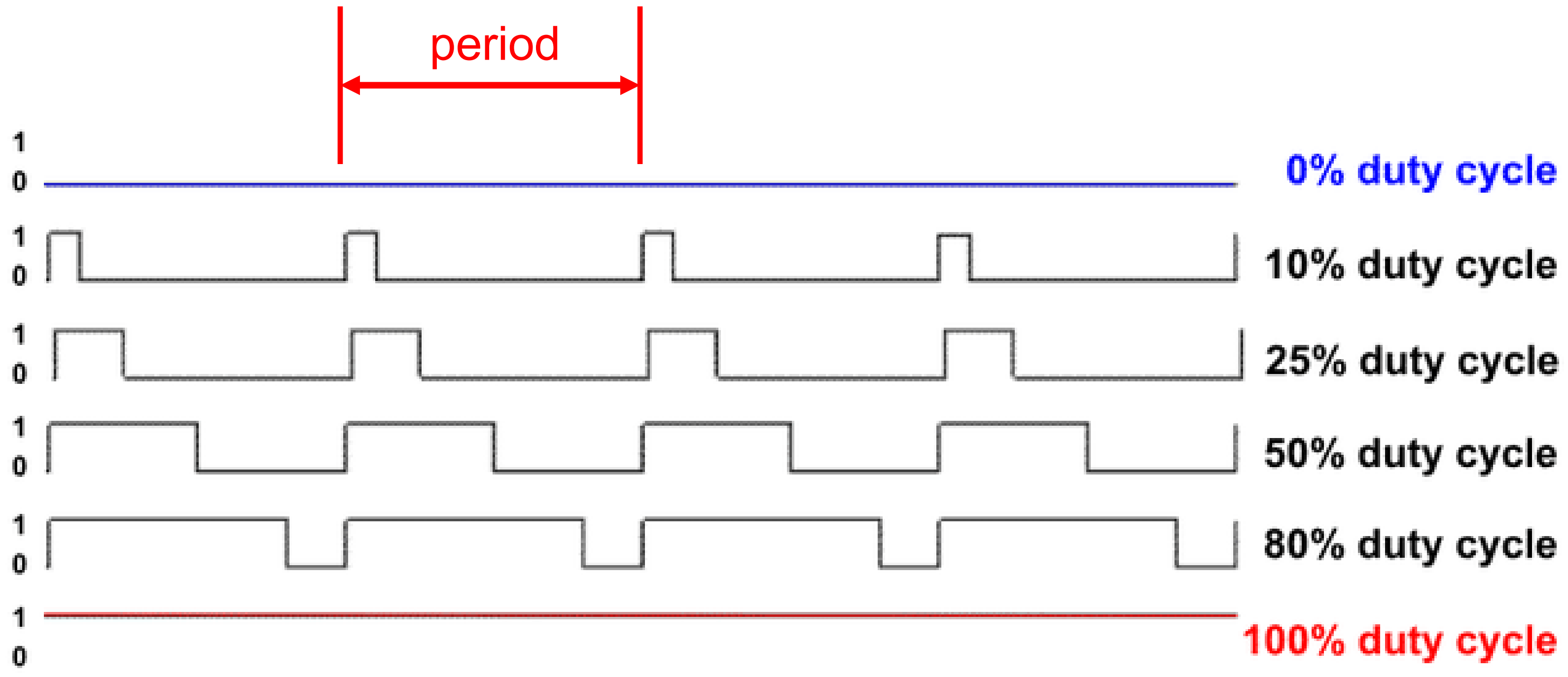
Pulse Width Modulation (PWM)

- To communicate with:
  - electric motors, servos
  - power electronics
  - LEDs
  - audio outputs

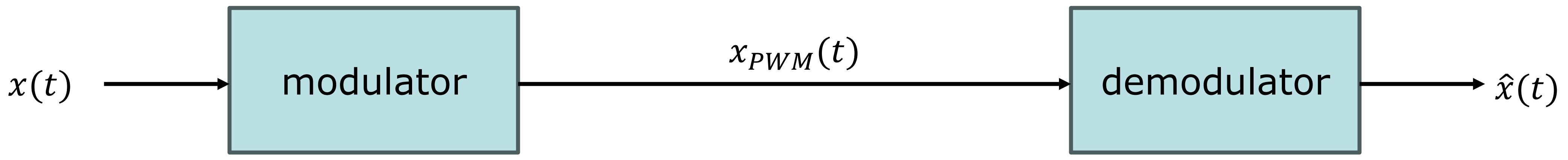
- …need a "cheap" way to output analog signal from digital pin

Pulse Width Modulation

- Way to implement an analog value with a digital pin
- Periodic signal whose duty cycle determines average voltage over time

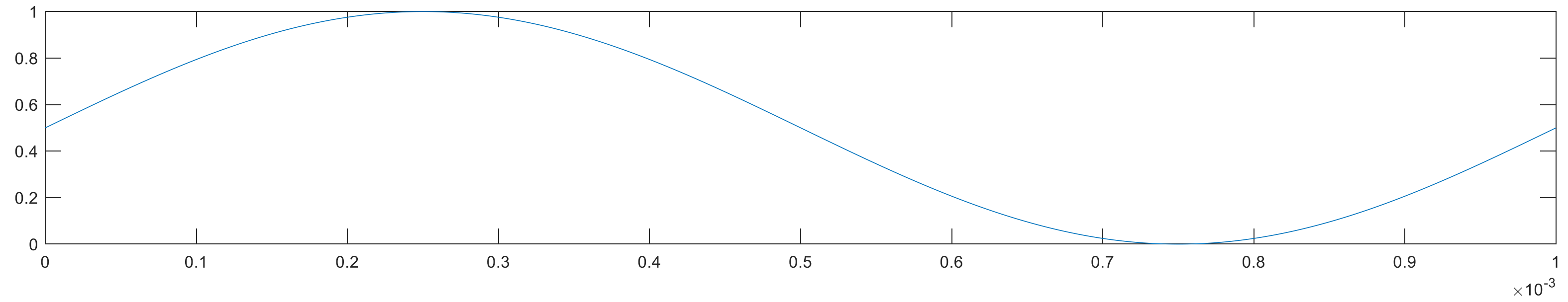


PWM Modulation

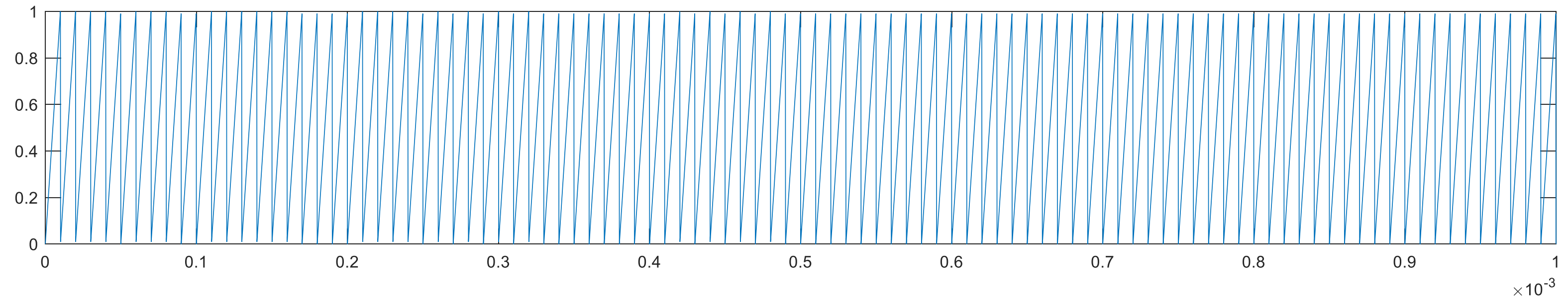


PWM

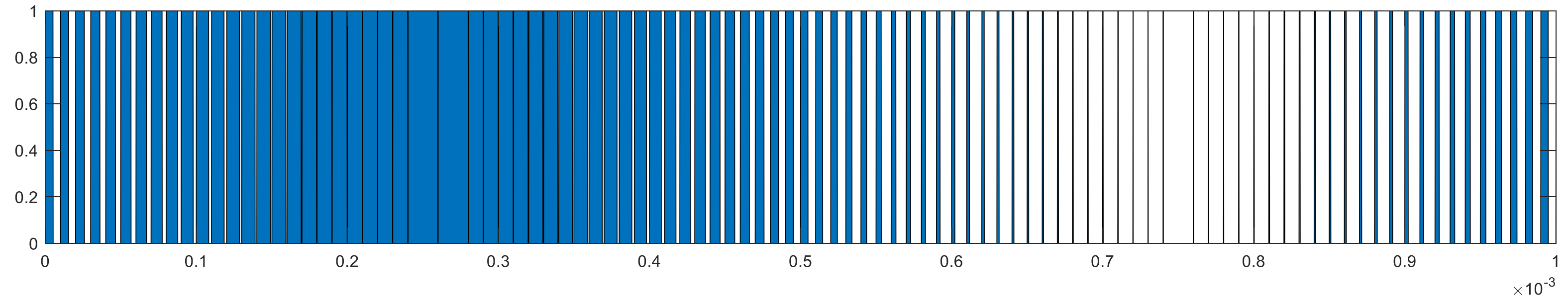
sin(x)
period = 1ms



sawtooth
period = 1us



PWM
sawtooth < sin(x)



Pulse Width Modulation

What analog voltage is generated with a 3.3 V PWM signal having a 30% duty cycle?

Pulse Width Modulation

What analog voltage is generated with a 3.3 V PWM signal having a 30% duty cycle?

3.3 V * 0.3 = 0.99 V

Interrupts

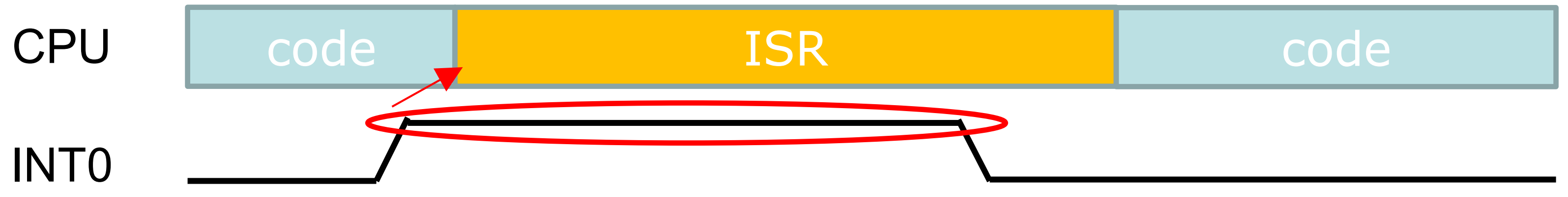
- Signal from a peripheral when an unpredictable event occurs
  - Peripheral initiates, CPU responds
  - e.g. keystroke, network packet received, transfer between memory and disk has completed
  - Similar concept: exceptions (or traps) that originate from internal sources
    - e.g. page fault, memory error (null pointer dereferenced)

- Between each peripheral and the CPU's interrupt controller:
  - Usually one interrupt output wire
  - Optionally, also an interrupt acknowledge input wire

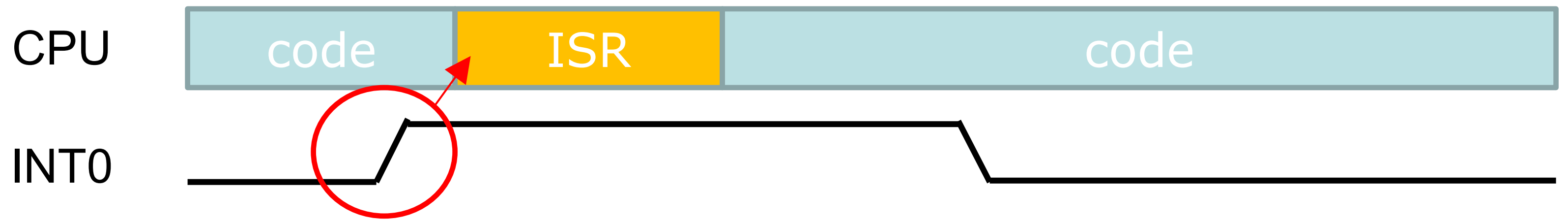
- Causes CPU to automatically jump to a location in the OS or HAL that performs the necessary actions to respond to the interrupt

Interrupts Without Acknowledge

level triggered (high):



rising-edge triggered:



Interrupts

Suppose a peripheral is connected to the CPU’s external interrupt input. Assume the peripheral can change the value of the external interrupt no more frequently than once every 50 microseconds. Assume the CPU can execute its interrupt service routine in 75 microseconds. What is the maximum number of interrupts per second possible when using rising edge-triggered interrupts or level sensitive interrupts?

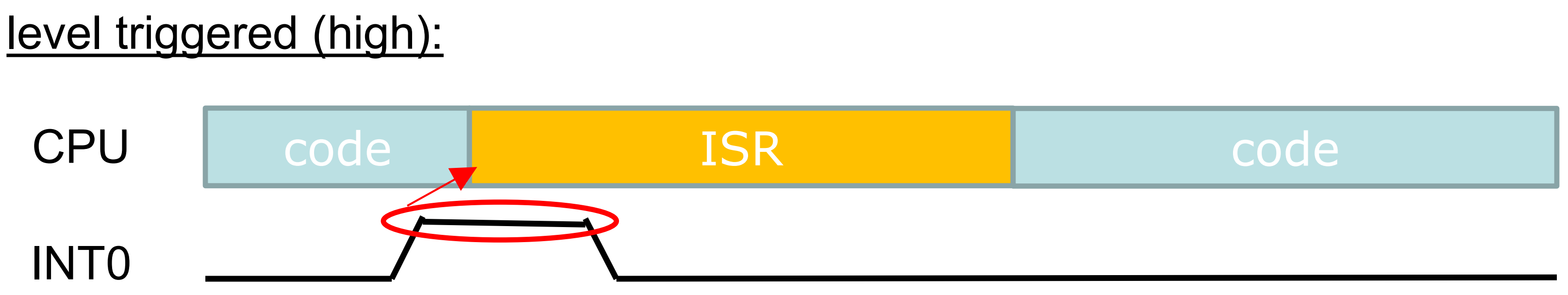
Interrupts

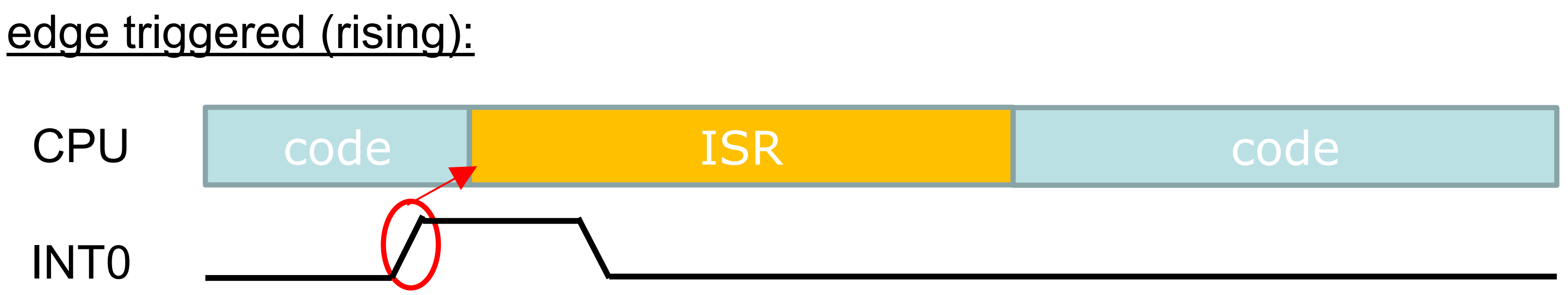
Suppose a peripheral is connected to the CPU's external interrupt input. Assume the peripheral can change the value of the external interrupt no more frequently than once every 50 microseconds. Assume the CPU can execute its interrupt service routine in 75 microseconds. What is the maximum number of interrupts per second possible when using rising edge-triggered interrupts or level sensitive interrupts?

edge-triggered: min(1 / (75 μs),1 / (100 μs))
= min(13 KHz,10 KHz)
= 10 KHz

level sensitive: min(1 / (75 μs),1 / (100 μs))
= min(13 KHz,10 KHz)
= 10 KHz

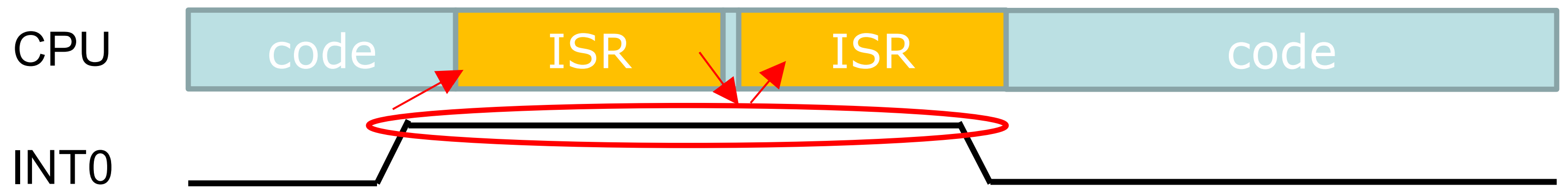
Interrupts Without Acknowledge



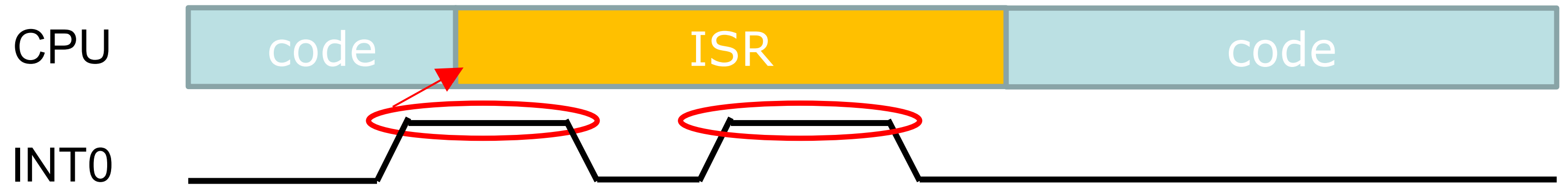


Interrupts Without Acknowledge

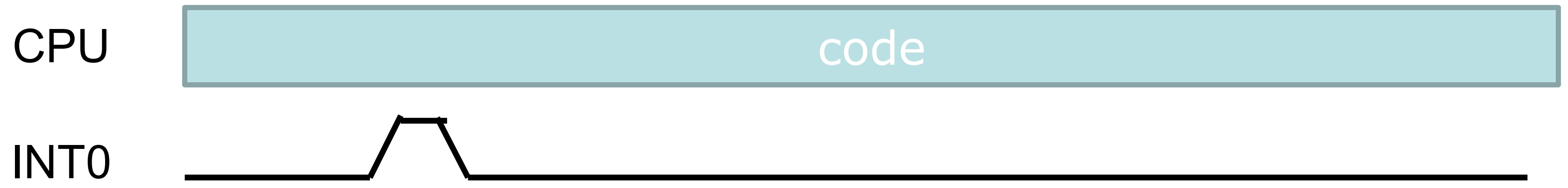
<u>hold INT0 too long:</u>



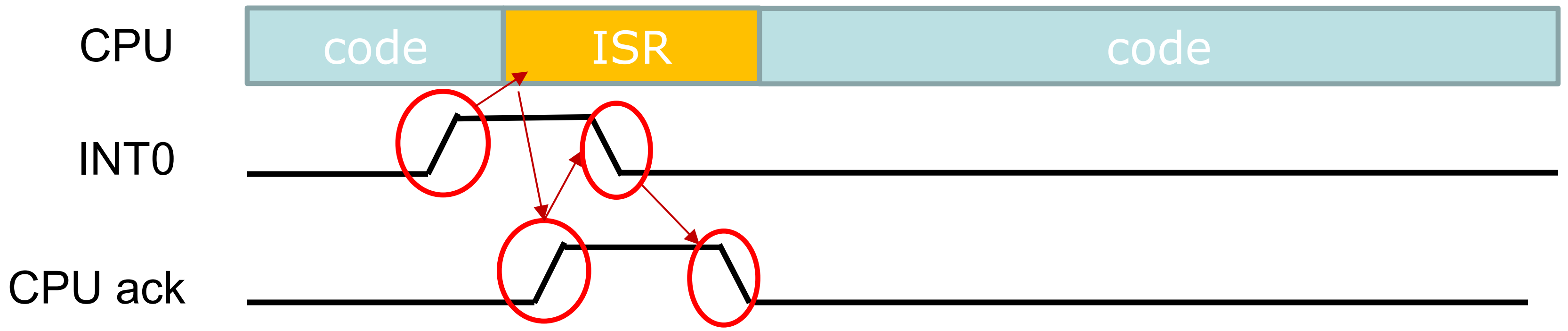
<u>interrupts occur faster than ISR:</u>



<u>INT0 not held long enough:</u>



Interrupts With Acknowledge



Interface Channels and Protocols

- Chip-to-chip or board-to-board communication

- Communication channels (physical) and protocols (datalink) differ depending on distance and performance expectation
  - Inner-chip/SoC (<10 mm): e.g. AXI
  - Chip-to-chip (<1 m): e.g. PCIe, QPI, RapidIO, SPI, I2C
  - Local area network (<10 m): e.g. Ethernet, Infiniband

- Chip-to-chip channels come in high-thoughput and low throughout varieties
  - Short haul: PCIe, Thunderbolt: 1 GB/s/channel
  - Longer haul: USB3, SATA: 500 MB/s/channel
  - Low-power, short haul: SPI and I2C (< 100 MB/s/channel)
  - UART: for console I/O (< 14 KB/s)

Bus Protocols

- Chip-to-chip channels (bus protocols) are either:
  - Asynchronous
    - No clock signal in the bus signals
    - Example: UART, PCIe, SATA
  - Synchronous
    - Includes clock signal as a control signal
    - Devices communicate with a protocol that is relative to the clock
    - Examples: SPI, I^{2}C, JTAG

SPI (Serial Peripheral Interface)

- Synchronous protocol
  - There is a "master side" of the channel that drives the clock and determines the channel speed
- Only master side can initiate transaction
  - Similar to programmed I/O
- All communications require an equal amount of data to be exchanged between both sides of channel
  - (even if the application doesn’t call for it)

SPI (Serial Peripheral Interface)

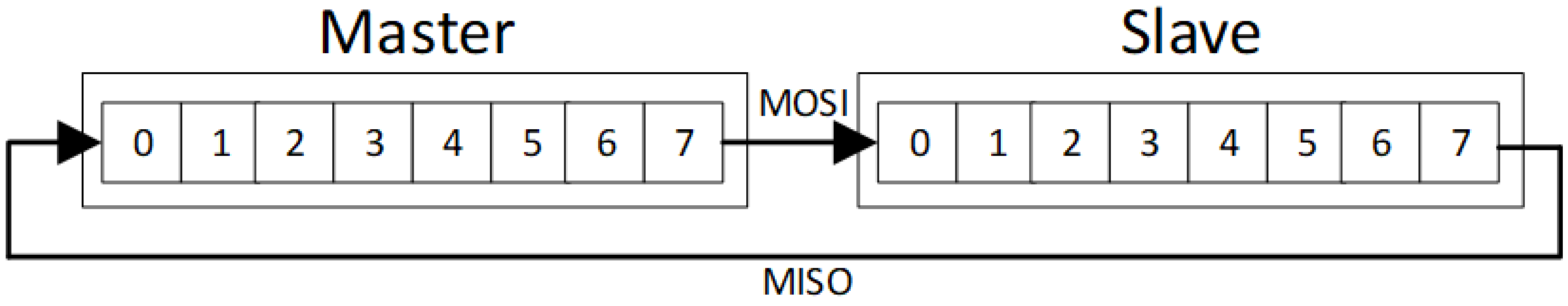
- To summarize:
  - Roles: one master, one or more slaves
  - Master (CPU) initiates transaction, slave (peripheral) must follow through on all read and write requests
  - SPI is byte-oriented (transmissions are multiples of 8 bits)
  - Bit ordering is application-defined

SPI (Serial Peripheral Interface)

- Includes four signals:
  - SCLK: clock
  - MOSI: master out, slave in
  - MISO: master in, slave out
  - SS: slave select
- clk is idle outside of an active transmission
- Active-low SS signal allows master to "activate" a slave

SPI Exchange

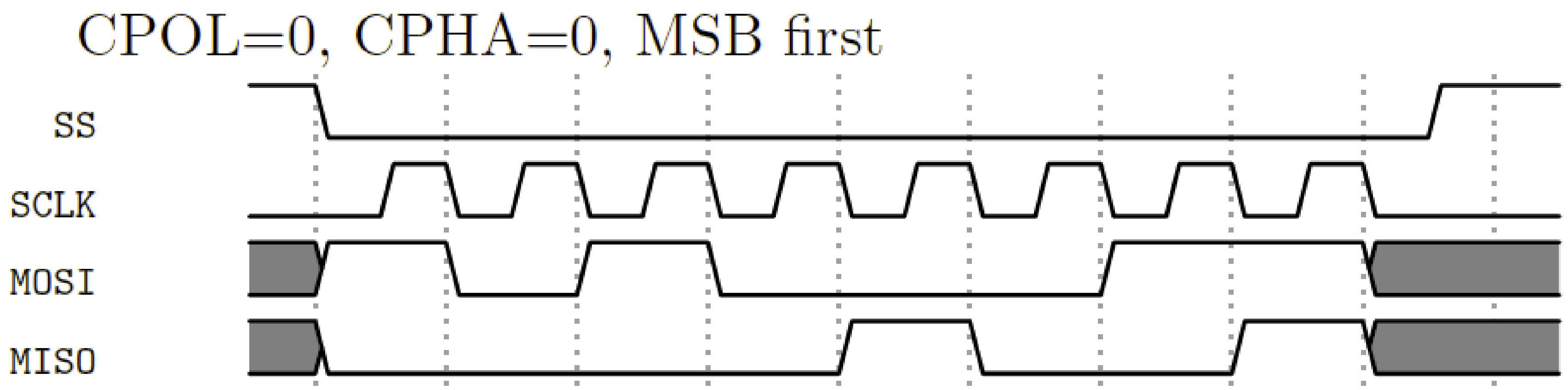
- Master:
  - Drives clock, typically a few MHz
  - “Selects” slave by setting SS to logic low (starts transmission)
  - During each cycle, master sends a bit to slave on MOSI and slave sends a bit to master on MISO



SPI (Serial Peripheral Interface) Settings

- There are four timing modes, determined by clock polarity (CPOL) and clock phase (CPHA)
  - CPOL=0: clock idles at 0
  - CPOL=1: clock idles at 1
  - CPHA=0: read data on first clock edge
    - data changes one-half cycle before first clock edge
  - CPHA=1: read data on second clock edge
    - data changes on first clock edge
  - {CPOL,CPHA} determines the SPI mode

SPI (Serial Peripheral Interface)

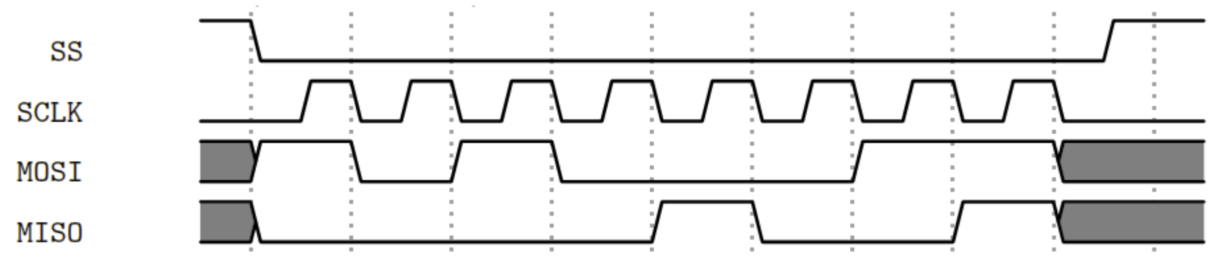


SPI (Serial Peripheral Interface)

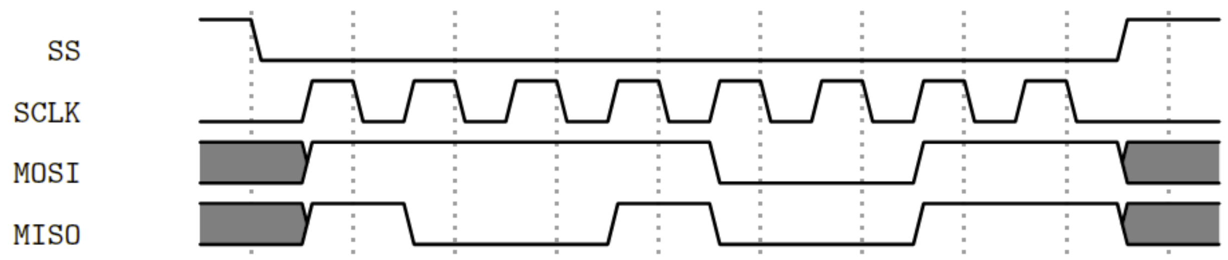
CPHA=0

CPHA=1

CPOL=0

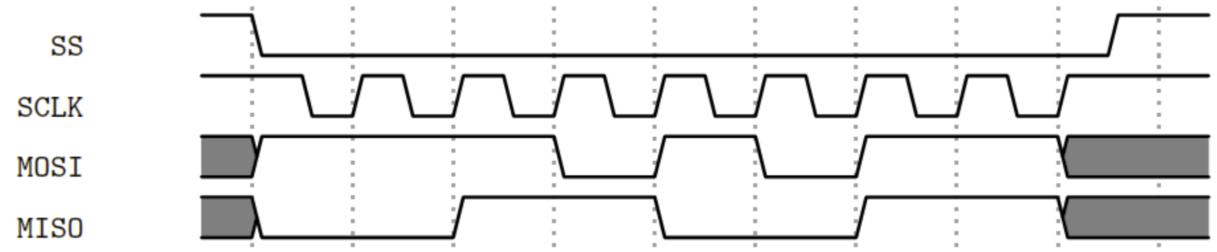


1. Falling edge of ss
2. Rising edge of sclk

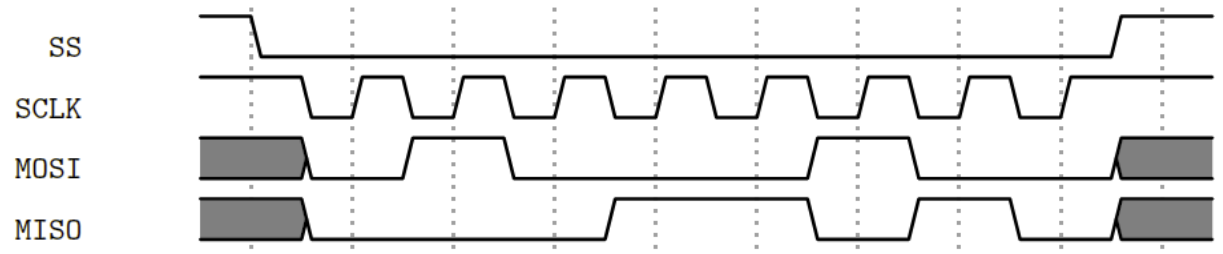


1. Falling edge of ss
2. Falling edge of sclk

CPOL=1



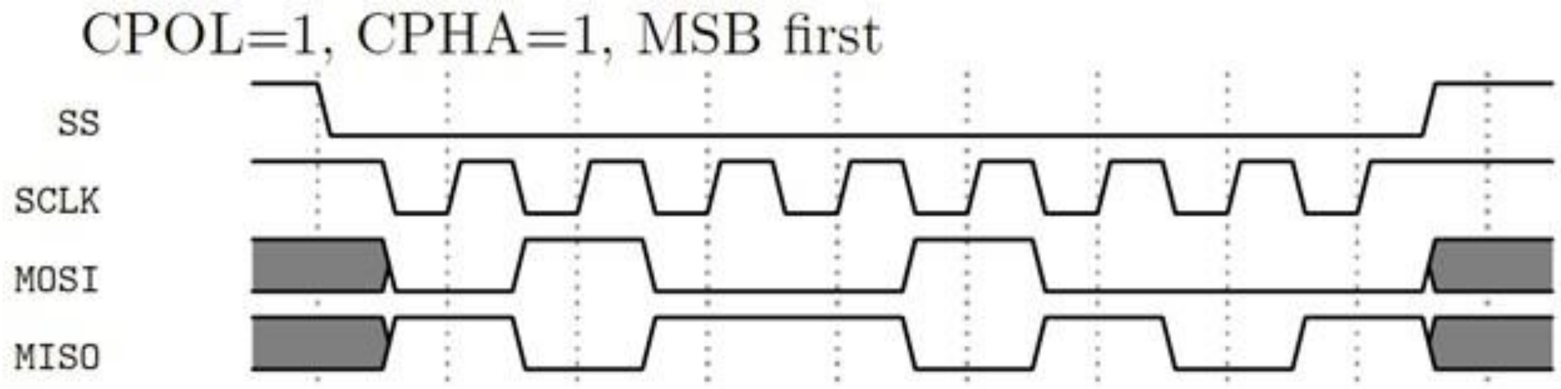
1. Falling edge of ss
2. Falling edge of sclk



1. Falling edge of ss
2. Rising edge of sclk

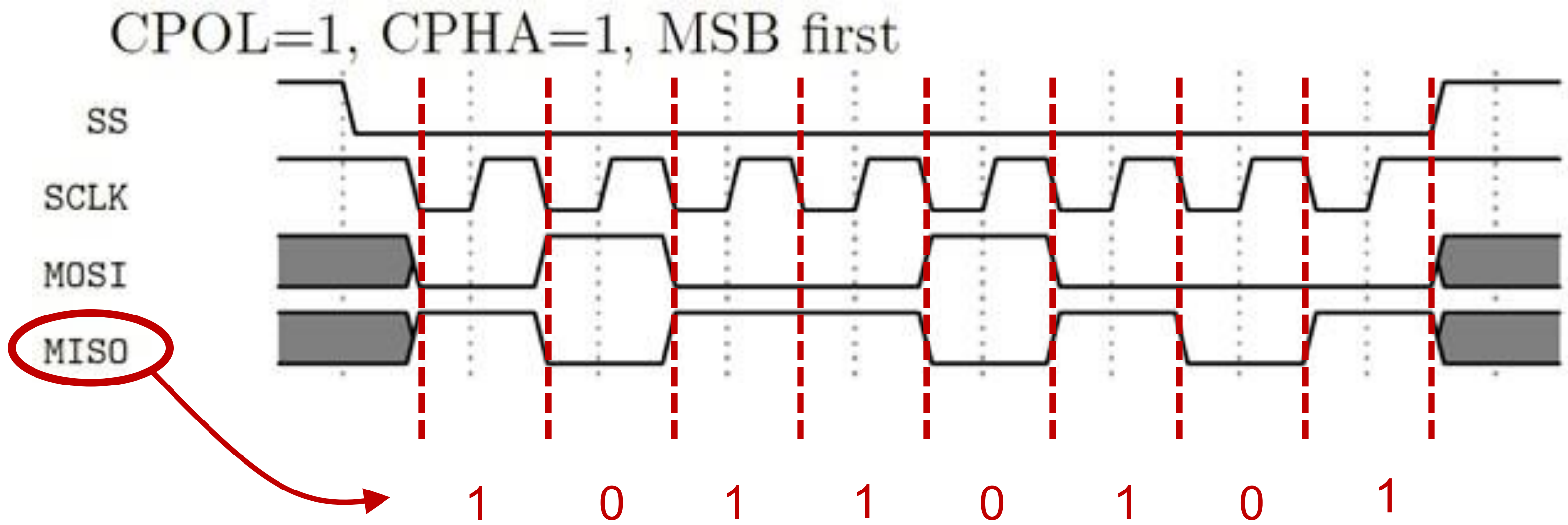
SPI

- In the following SPI exchange, what is the value sent from the slave to the master?



SPI

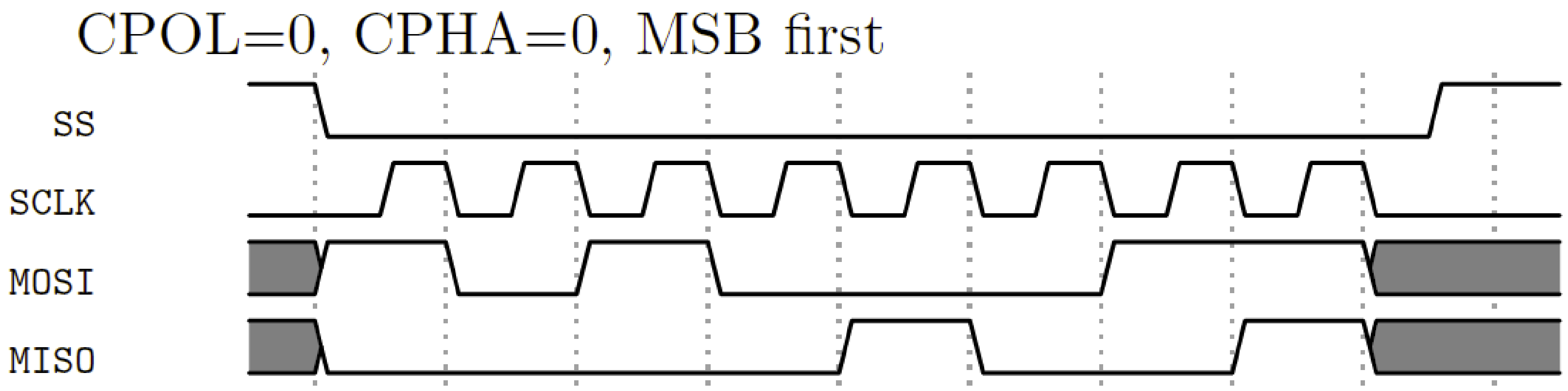
- In the following SPI exchange, what is the value sent from the slave to the master?



10110101 = 0xB5

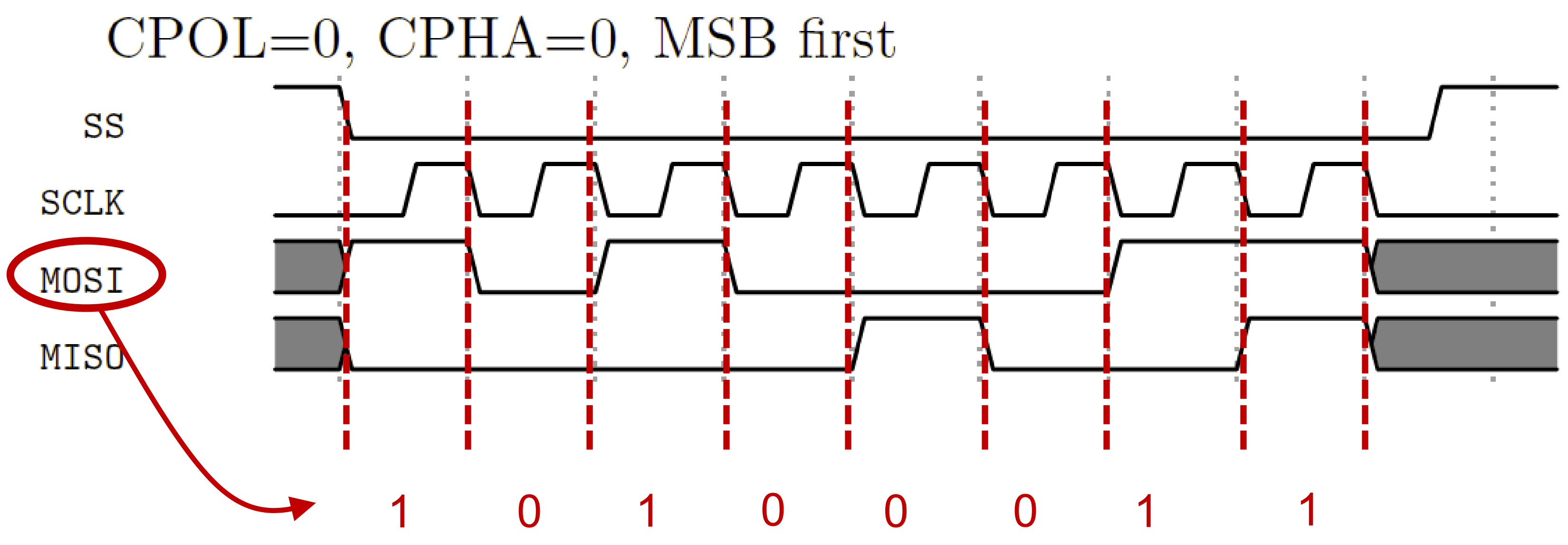
SPI Exchange

- In the following SPI exchange, what is the value sent from the master to the slave?



SPI Exchange

- In the following SPI exchange, what is the value sent from the master to the slave?



10100011 = 0xA3

Terminology

- Applications usually want a “programmed I/O-style” interface

- Important terminology:
  - SPI exchange: in 8 cycles, master sends 8 bits to slave (MOSI) and slave sends 8 bits to master (MISO)

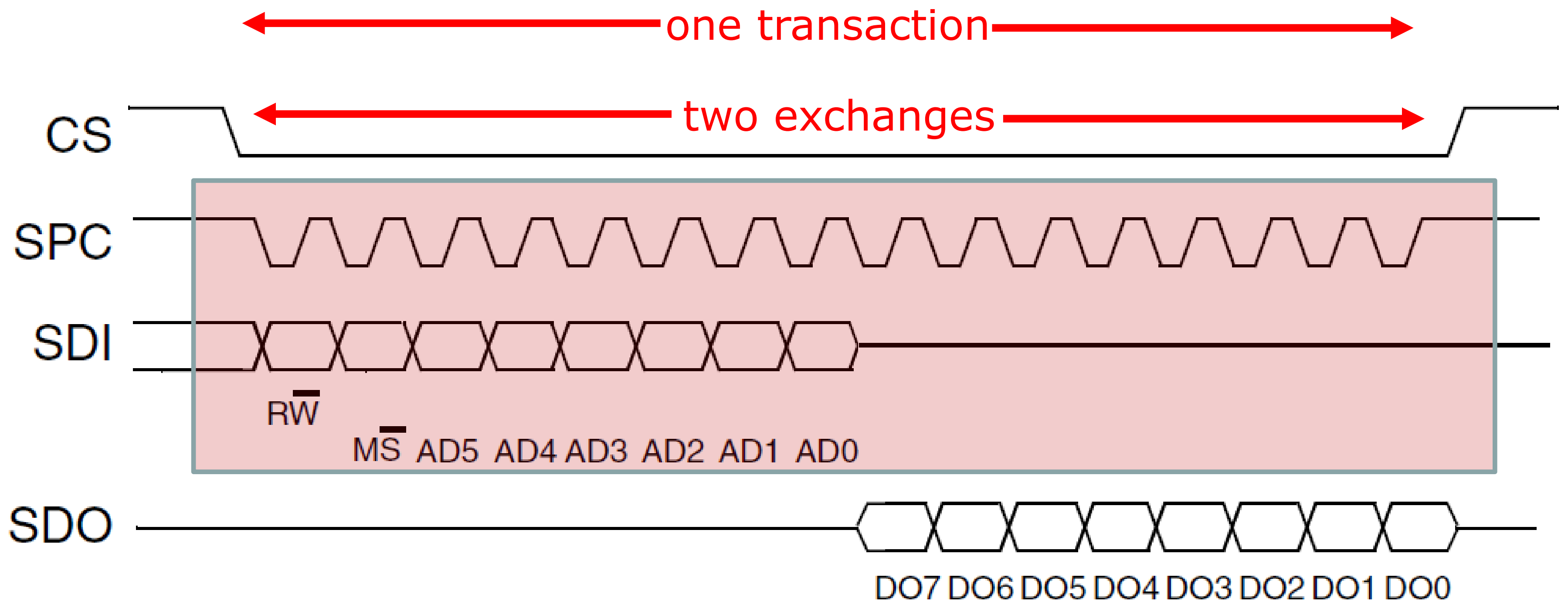
  - SPI transaction: all the exchanges that occur while the slave-select is asserted
    - May contain multiple exchanges
    - First exchange establishes the master’s requested operation, subsequent exhanges are for data
    - Must contain at least two exchanges to read or write a register on the slave

(Sample) SPI Transaction Protocol

- Master reads from slave
  - Byte 1:
    - Master sends => address and flags
    - Slave sends => dummy
  - Byte 2:
    - Master sends => dummy
    - Slave sends => contents of address

- Master writes to slave
  - Byte 1:
    - Master sends => address and flags
    - Slave sends => dummy
  - Byte 2:
    - Master sends => data to write
    - Slave sends => dummy

SPI Read Operation



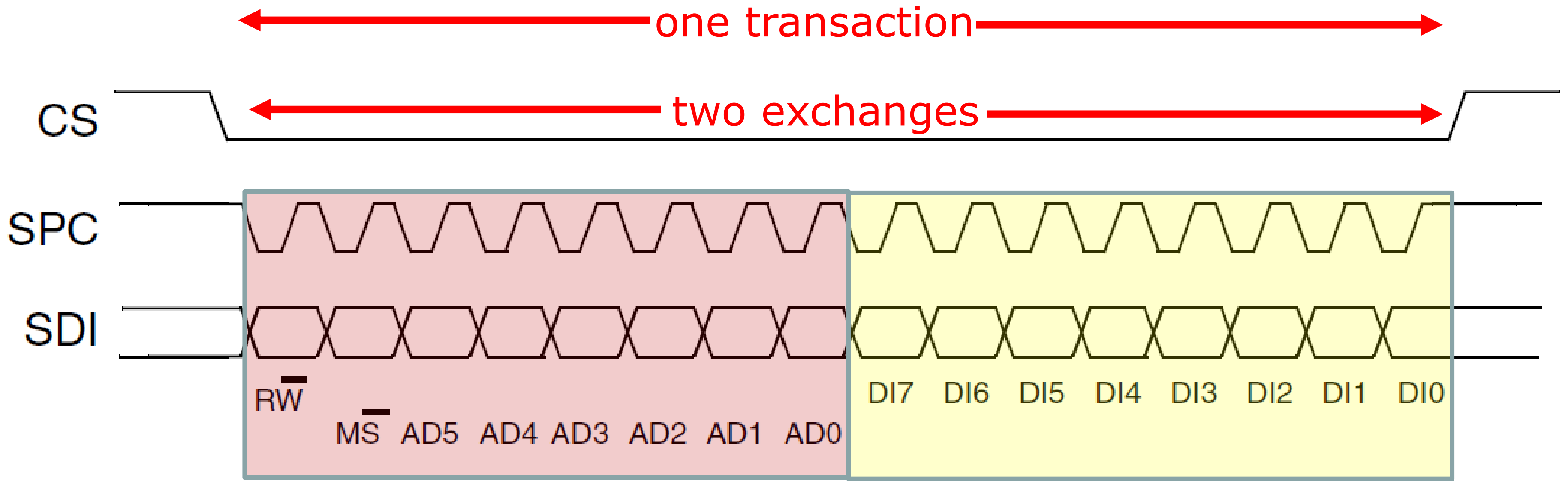
master->slave:
read flag, address

slave->master:
don’t care

master->slave:
don’t care

slave->master:
contents of read register

SPI Write Operation



master->slave:
read flag, address

slave->master:
don't care

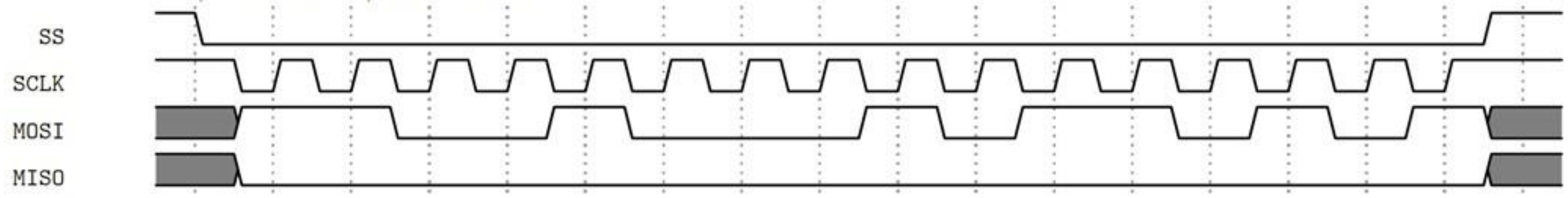
master->slave:
data

slave->master:
don't care

SPI Transaction

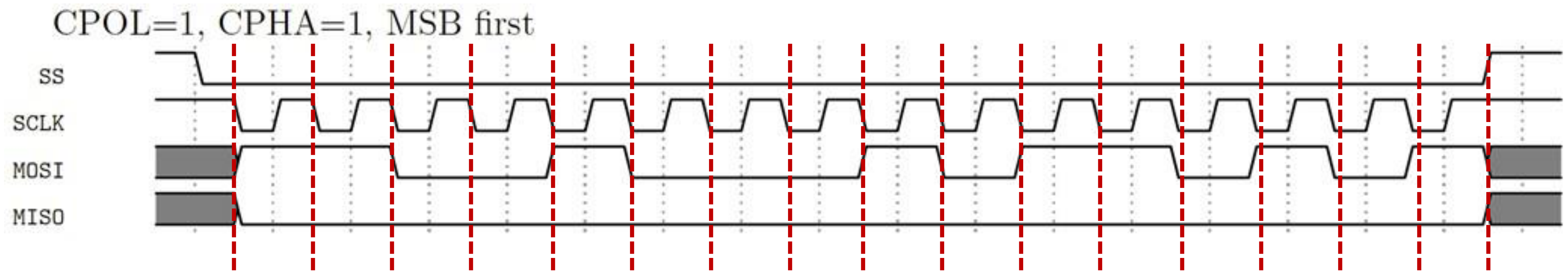
In the following SPI transaction, assume the transaction address is stored in the least significant 5 bits and the $\overline{R}$/W flag is stored in the most significant bit of the first byte transmitted from master to slave. What is the transaction?

CPOL=1, CPHA=1, MSB first



SPI Transaction

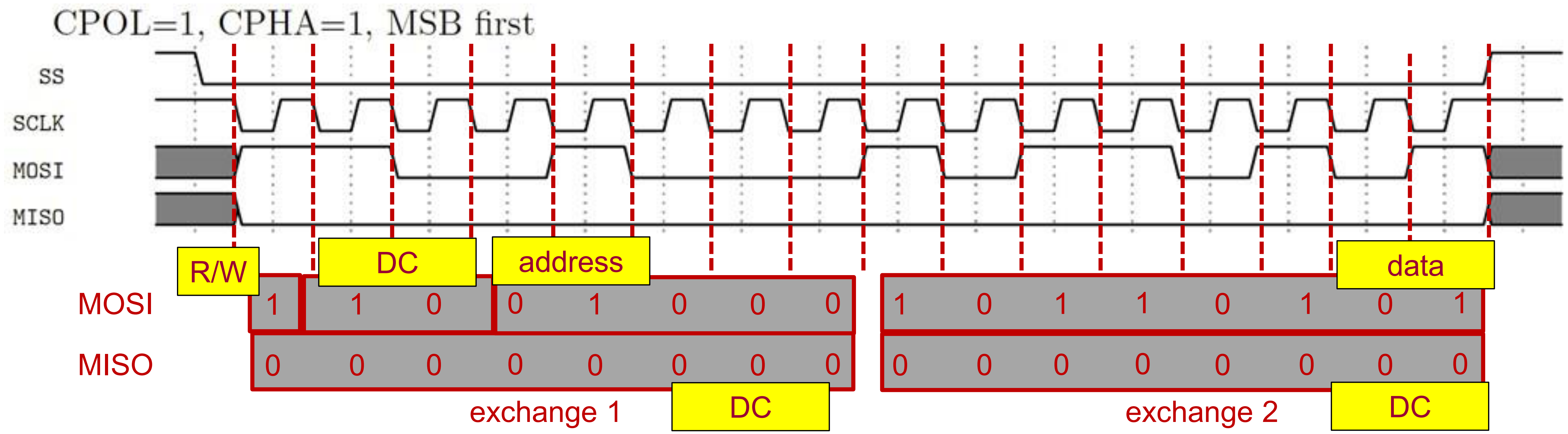
In the following SPI transaction, assume the transaction address is stored in the least significant 5 bits and the $\overline{\text{R}}$/W flag is stored in the most significant bit of the first byte transmitted from master to slave. What is the transaction?



MOSI	1	1	0	0	1	0	0	0	1	0	1	1	0	1	0	1
MISO	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	exchange 1								exchange 2							

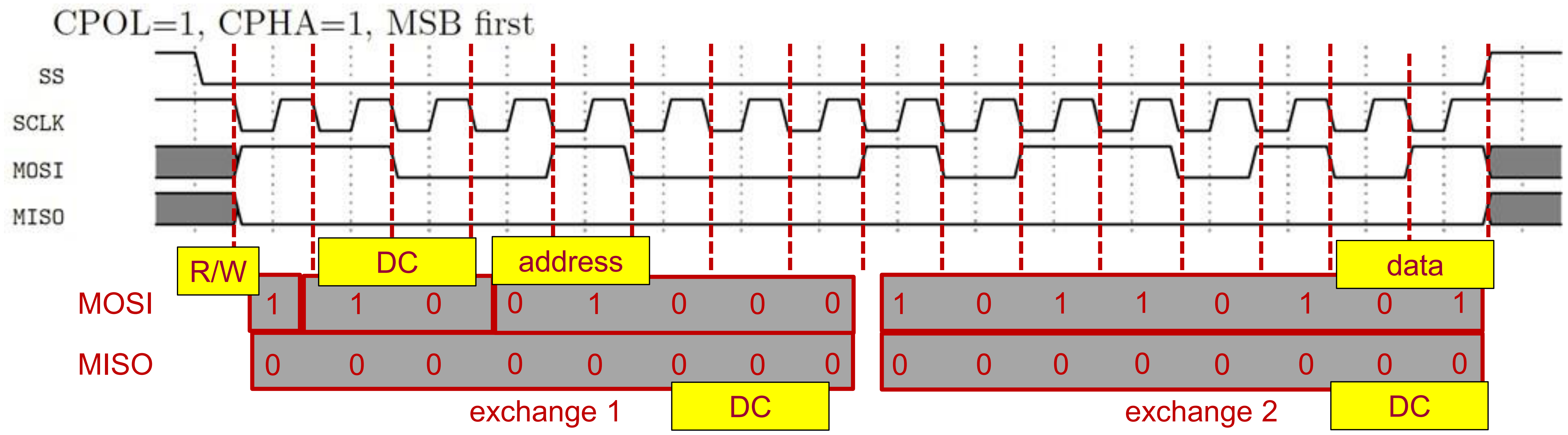
SPI Transaction

In the following SPI transaction, assume the transaction address is stored in the least significant 5 bits and the $\overline{R}/W$ flag is stored in the most significant bit of the first byte transmitted from master to slave. What is the transaction?



SPI Transaction

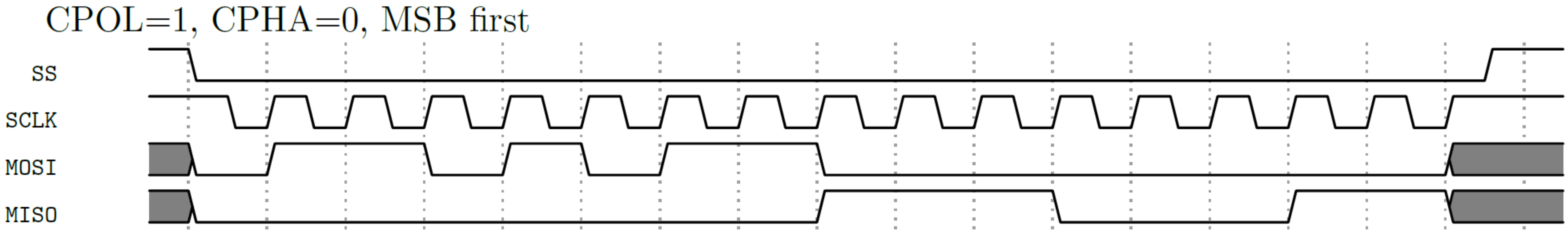
In the following SPI transaction, assume the transaction address is stored in the least significant 5 bits and the $\overline{R}$/W flag is stored in the most significant bit of the first byte transmitted from master to slave. What is the transaction?



Master writes value 0xB5 to address 0x8

SPI Transaction

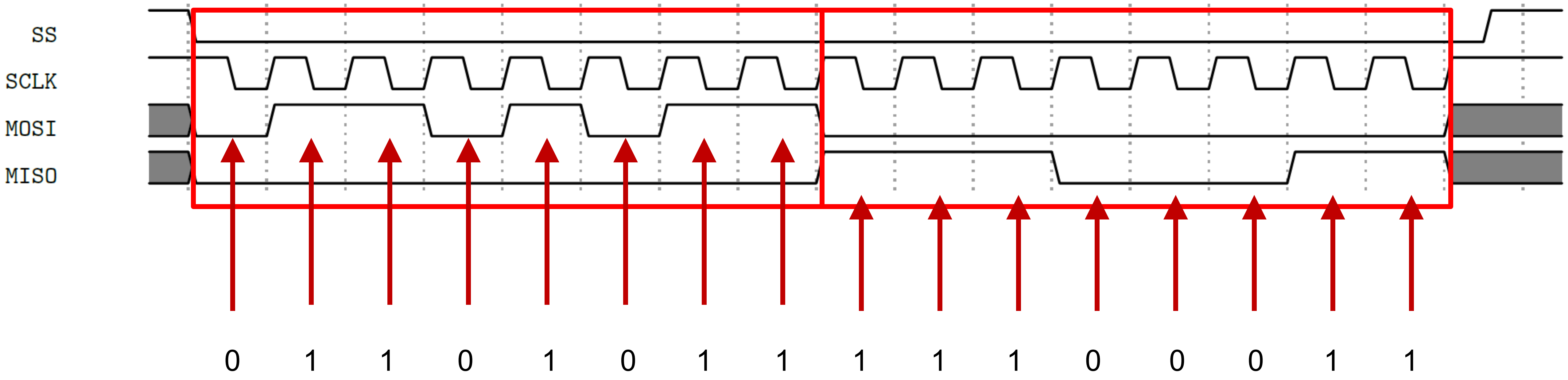
In the following SPI transaction, assume the transaction address is stored in the least significant 5 bits and the $\overline{R}$/W flag is stored in the most significant bit of the first byte transmitted from master to slave. What is the transaction?



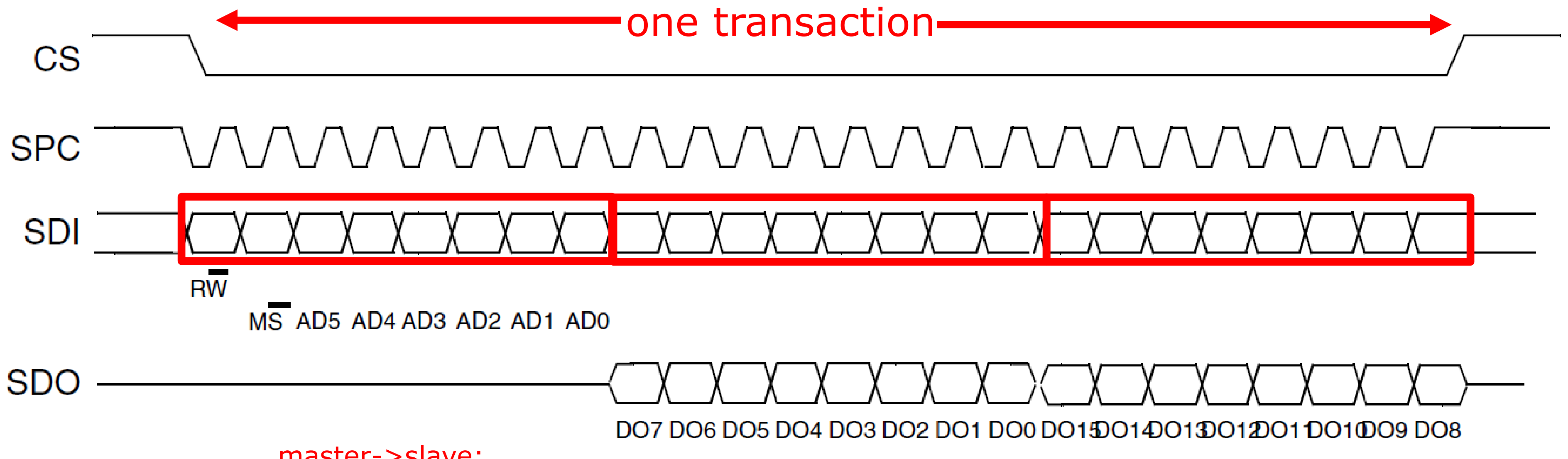
SPI Transaction

In the following SPI transaction, assume the transaction address is stored in the least significant 5 bits and the $\overline{R}$/W flag is stored in the most significant bit of the first byte transmitted from master to slave. What is the transaction?

CPOL=1, CPHA=0, MSB first



SPI Multiple Read


one transaction
CS
SPC
SDI
RW
MS AD5 AD4 AD3 AD2 AD1 AD0
SDO
DO7 DO6 DO5 DO4 DO3 DO2 DO1 DO0 DO15 DO14 DO13 DO12 DO11 DO10 DO9 DO8

master->slave: read flag, address

slave->master: ???????

master->slave: ???????

slave->master: contents of read register 1

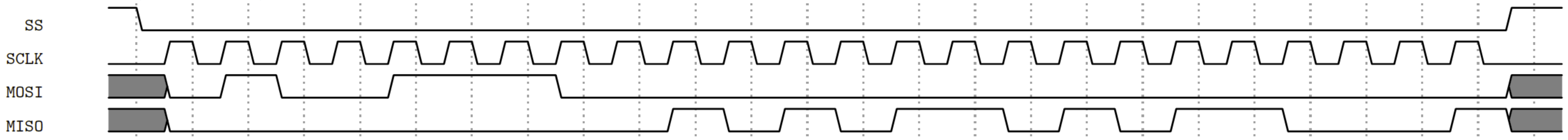
master->slave: ???????

slave->master: contents of read register 2

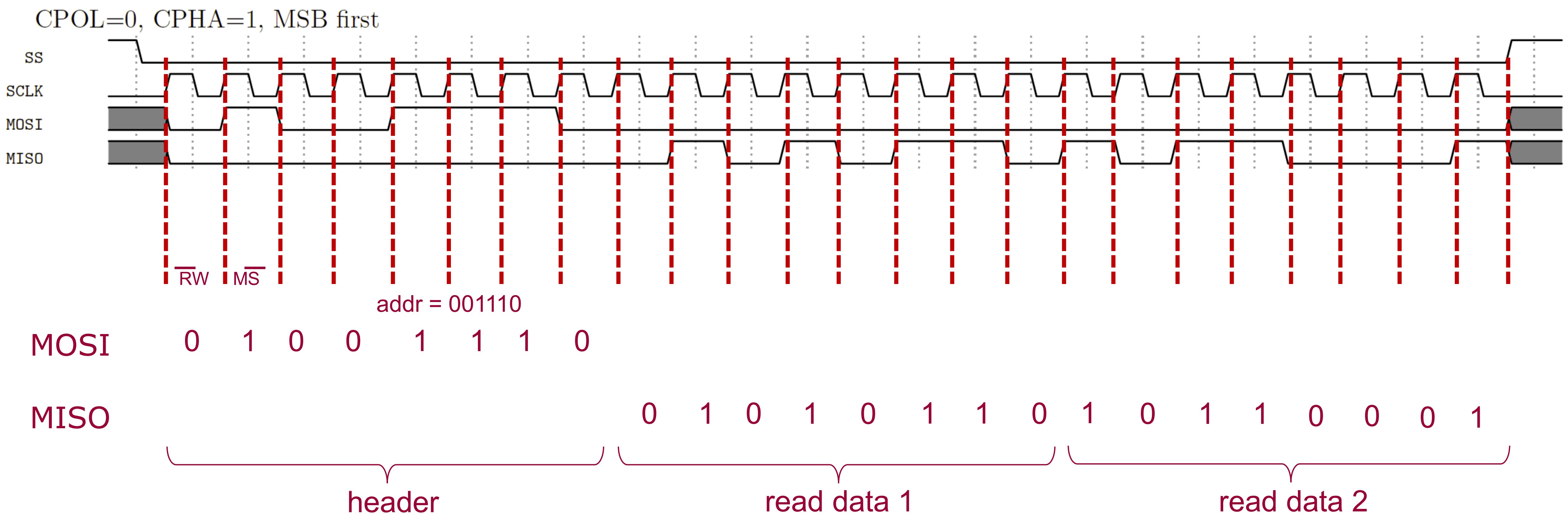
three exchanges

SPI Multiple Read

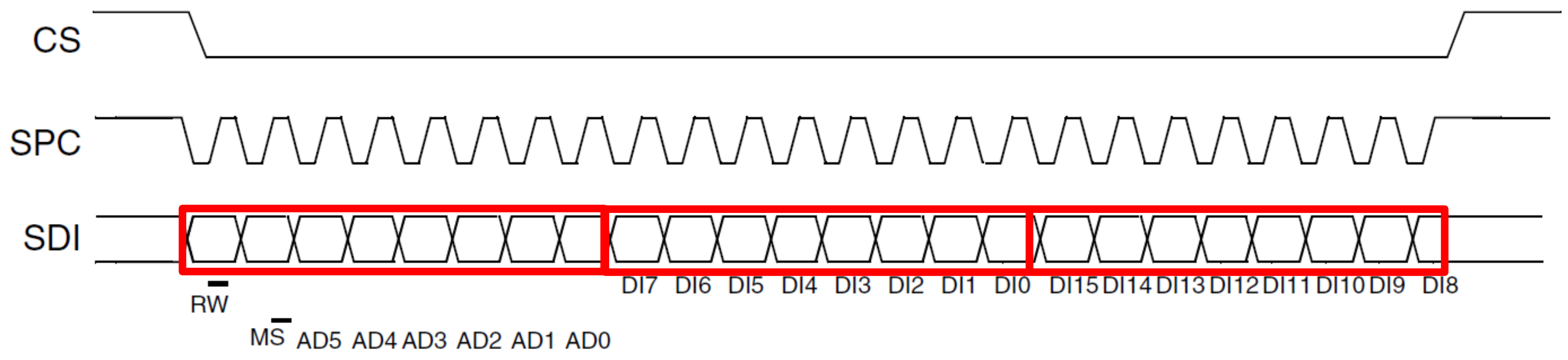
CPOL=0, CPHA=1, MSB first



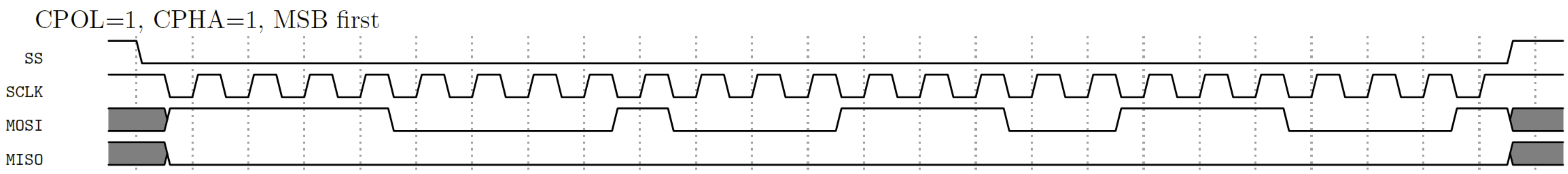
SPI Multiple Read



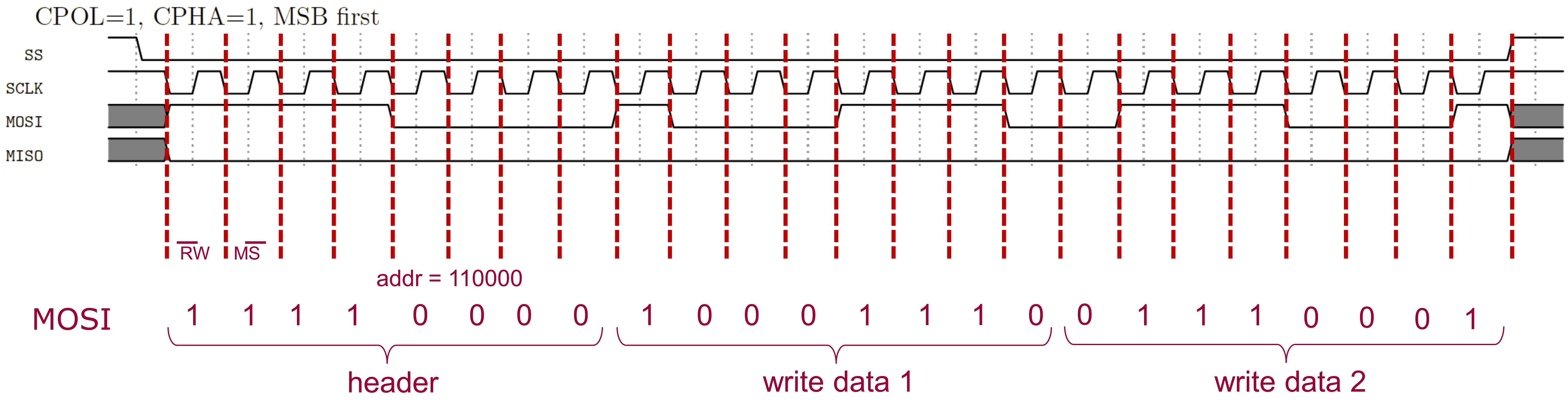
SPI Multiple Write


CS
SPC
SDI
RW
MS
AD5 AD4 AD3 AD2 AD1 AD0
DI7 DI6 DI5 DI4 DI3 DI2 DI1 DI0 DI15 DI14 DI13 DI12 DI11 DI10 DI9 DI8

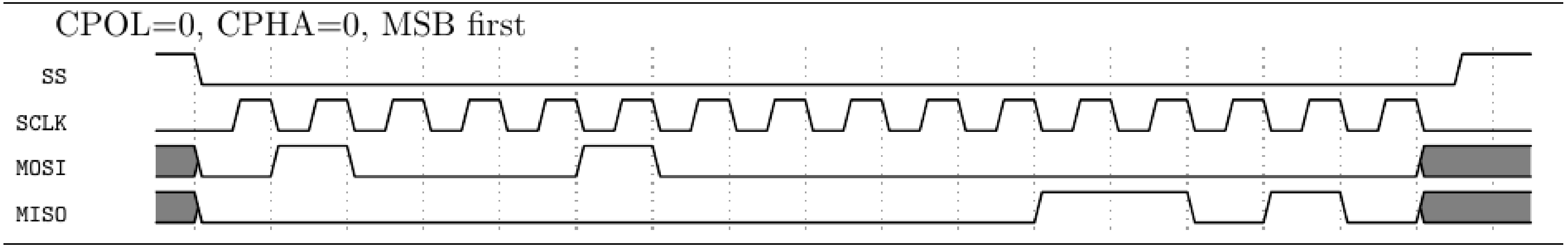
SPI Transaction



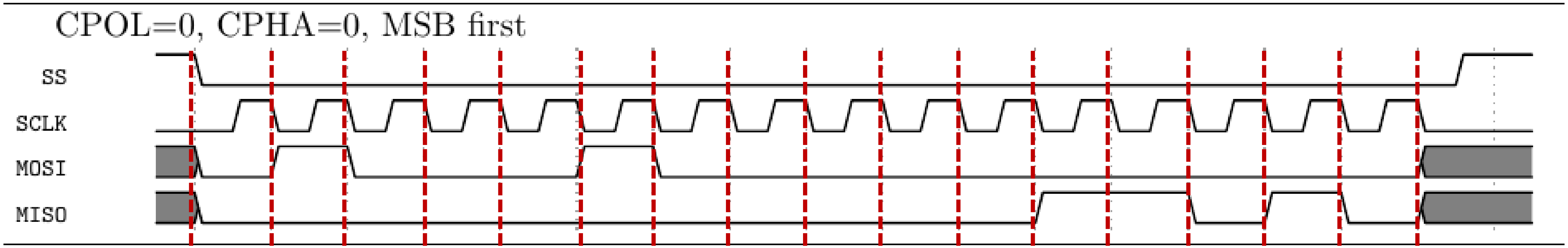
SPI Transaction



Transaction-Level Protocol



Transaction-Level Protocol



MOSI	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0
MISO	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0

SPI Summary

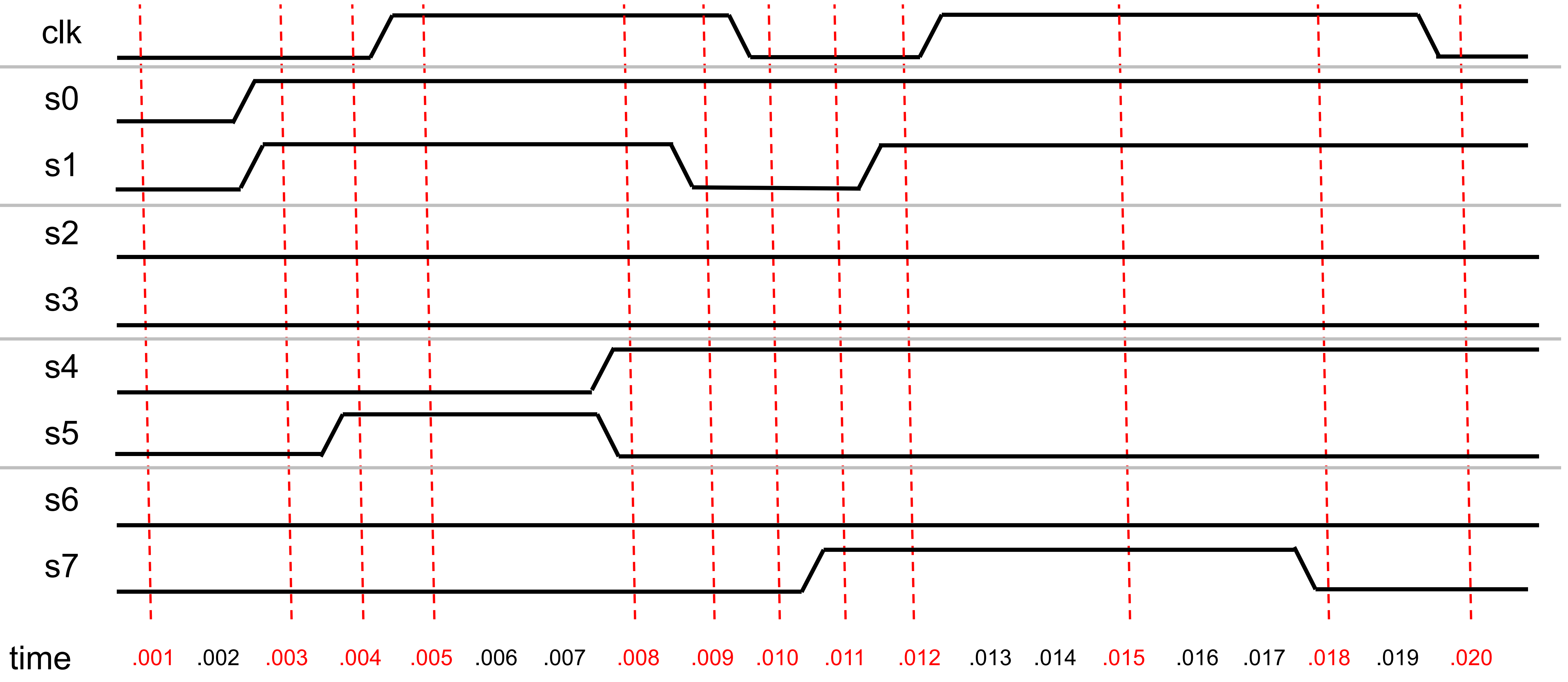
- Generally single master, single slave (point-to-point)
- Synchronous, byte-oriented
- 4 wires per channel
- Four timing modes (CPOL,CPHA = 00,01,10,11)
- Physical layer is based on notion of "exchanges"
- SPI is a *de facto* standard, not standardized
- Upper, protocol layers are defined by the application

Lab 1 Objective

- Read a trace file containing SPI transactions
- Tasks:
  1. Decode simple read and write transactions for CPOL=0, CPHA=0
  2. Decode streaming (multi-byte) read and write transactions
  3. Decode simple and streaming transactions with arbitrary CPOL/CPHA values
- Read and interpret a signal trace file
  - We wrote a parser for you
- Trace contains a list of captured samples at non-uniform intervals of:
  - a clock signal
  - several data bits, meant to be read in parallel
- Print the corresponding sequence of clocked parallel values
- Run solution with:

```
sh grade.sh
```

Signal Trace



Lab 1

- The objective of Lab 1 is to write a program that interprets a signal trace, e.g.

```
12                                                          # samples
clk     s0      s1      s2      s3      s4      s5      s6      s7
1       1       1       1       1       1       1       1       1
0.001   0       0       0       0       0       0       0       0       0
0.003   0       1       1       0       0       0       0       0       0
0.004   0       1       1       0       0       0       1       0       0
0.005   1       1       1       0       0       0       1       0       0
0.008   1       1       1       0       0       1       0       0       0
0.009   1       1       0       0       0       1       0       0       0
0.010   0       1       0       0       0       1       0       0       0
0.011   0       1       0       0       0       1       0       0       1
0.012   0       1       1       0       0       1       0       0       1
0.015   1       1       1       0       0       1       0       0       1
0.018   1       1       1       0       0       1       0       0       0
0.020   0       1       1       0       0       1       0       0       0
```

Lab 1

- The objective of Lab 1 is to write a program that interprets a signal trace, e.g.

```
12
clk     s0      s1      s2      s3      s4      s5      s6      s7
1       1       1       1       1       1       1       1       1
0.001   0       0       0       0       0       0       0       0       0
0.003   0       1       1       0       0       0       0       0       0
0.004   0       1       1       0       0       0       1       0       0
0.005   1       1       1       0       0       0       1       0       0
0.008   1       1       1       0       0       1       0       0       0
0.009   1       1       0       0       0       1       0       0       0
0.010   0       1       0       0       0       1       0       0       0
0.011   0       1       0       0       0       1       0       0       1
0.012   0       1       1       0       0       1       0       0       1
0.015   1       1       1       0       0       1       0       0       1
0.018   1       1       1       0       0       1       0       0       0
0.020   0       1       1       0       0       1       0       0       0
```

Signals

Lab 1

- The objective of Lab 1 is to write a program that interprets a signal trace, e.g.

```
12
clk     s0      s1      s2      s3      s4      s5      s6      s7
1       1       1       1       1       1       1       1       1
0.001   0       0       0       0       0       0       0       0       0
0.003   0       1       1       0       0       0       0       0       0
0.004   0       1       1       0       0       0       1       0       0
0.005   1       1       1       0       0       0       1       0       0
0.008   1       1       1       0       0       1       0       0       0
0.009   1       1       0       0       0       1       0       0       0
0.010   0       1       0       0       0       1       0       0       0
0.011   0       1       0       0       0       1       0       0       1
0.012   0       1       1       0       0       1       0       0       1
0.015   1       1       1       0       0       1       0       0       1
0.018   1       1       1       0       0       1       0       0       0
0.020   0       1       1       0       0       1       0       0       0
```

Signal widths

Lab 1

- The objective of Lab 1 is to write a program that interprets a signal trace, e.g.

```
12
clk      s0       s1       s2       s3       s4       s5       s6       s7
1        1        1        1        1        1        1        1        1
0.001    0        0        0        0        0        0        0        0        0
0.003    0        1        1        0        0        0        0        0        0
0.004    0        1        1        0        0        0        1        0        0
0.005    1        1        1        0        0        0        1        0        0
0.008    1        1        1        0        0        1        0        0        0
0.009    1        1        0        0        0        1        0        0        0
0.010    0        1        0        0        0        1        0        0        0
0.011    0        1        0        0        0        1        0        0        1
0.012    0        1        1        0        0        1        0        0        1
0.015    1        1        1        0        0        1        0        0        1
0.018    1        1        1        0        0        1        0        0        0
0.020    0        1        1        0        0        1        0        0        0
```

Sample capture time

Values

Sample API Provided by Us

- Instance data structure:
  - C:
    ```
    waves *mywaves;
    ```
  - Python:
    ```
    mywaves = Waves()
    ```
- Read an input file (from stdin):
  - C:
    ```
    mywaves = parse_file(stdin);
    …
    free_waves(mywaves);
    ```
  - Python:
    ```
    mywaves.loadText(sys.stdin.read())
    ```

Sample API Provided by Us

- Locate time of rising and/or falling edge:
  - C:
    ```
    float after = 2.0; bool posedge = true; bool negedge = false;
    float edge_time = next_edge(mywaves, "clk", after, posedge, negedge);
    ```
  - Python:
    ```
    start_time = 2.0
    posedge = true
    negedge = false
    edge_time,found = mywaves.nextEdge("clk",start_time,posedge,negedge)
    ```
- Read signal value:
  - C:
    ```
    int32_t val = signal_at(mywaves, "s0", edge_time);
    ```
  - Python:
    ```
    val = mywaves.signalAt("s0",edge_at)
    ```

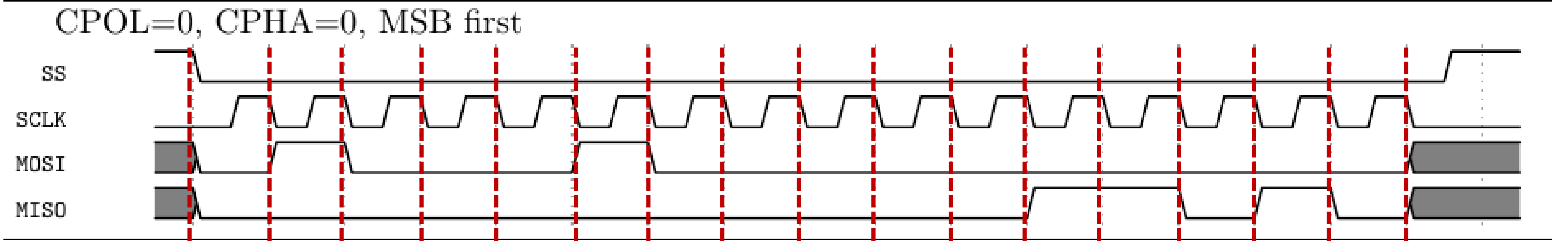
Sample API Provided by Us

- Log:
  - C:

    ```
    log("read signal %s at time %0.4f %d\n","s0",edge_time,val);
    ```
  - Python:

    ```
    sys.stderr.write(…)
    ```

Transaction-Level Protocol



MOSI	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0
MISO	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0

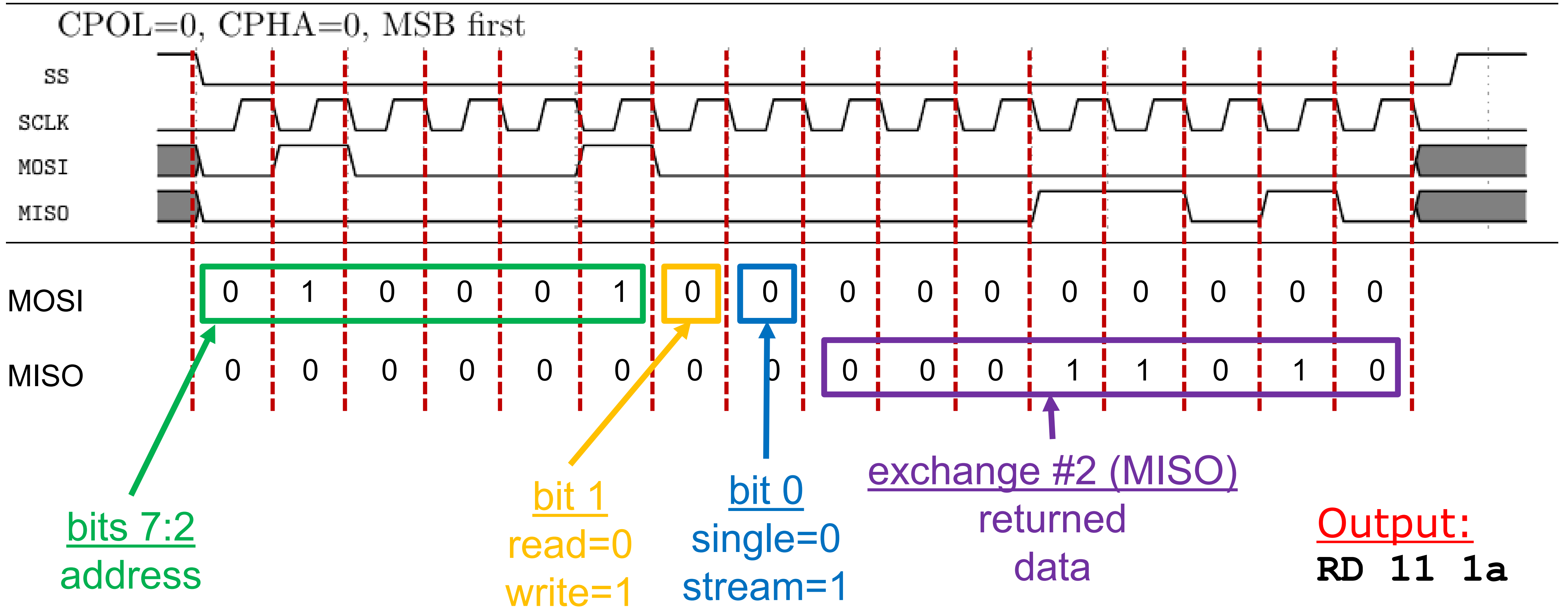
<u>bits 7:2</u>
address

<u>bit 1</u>
read=0
write=1

<u>bit 0</u>
single=0
stream=1

<u>exchange #2 (MISO)</u>
returned
data

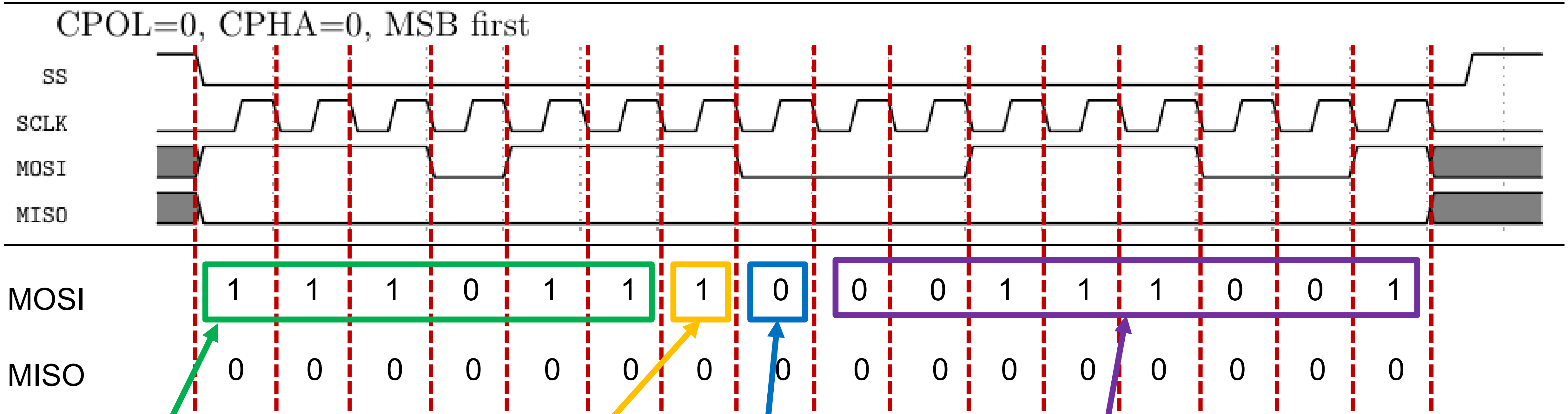
Transaction-Level Protocol



Output:

```
RD 11 1a
```

Transaction-Level Protocol



MOSI	1	1	1	0	1	1	1	0	0	0	1	1	1	0	0	1
MISO	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

bits 7:2
address

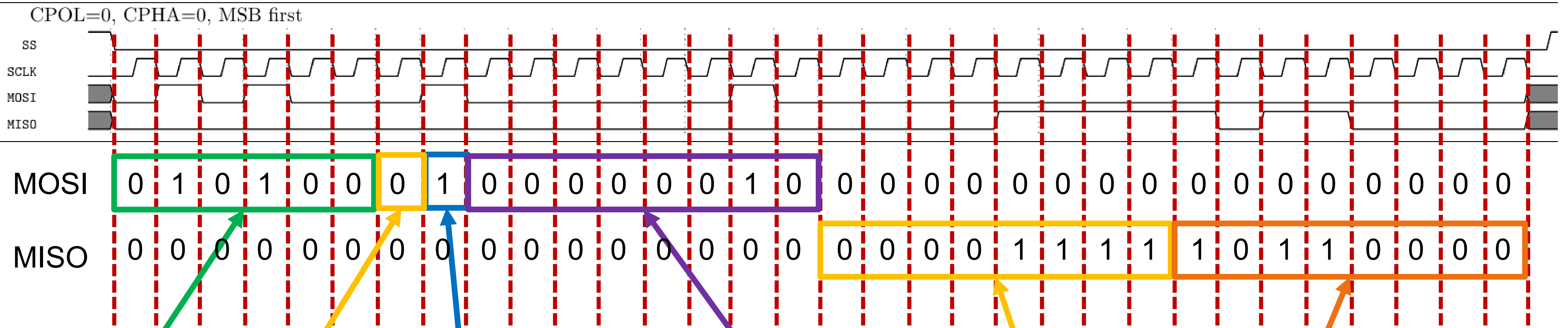
bit 1
read=0
write=1

bit 0
single=0
stream=1

exchange #2 (MOSI)
written data

Output:
WR 3b 39

Steaming Read

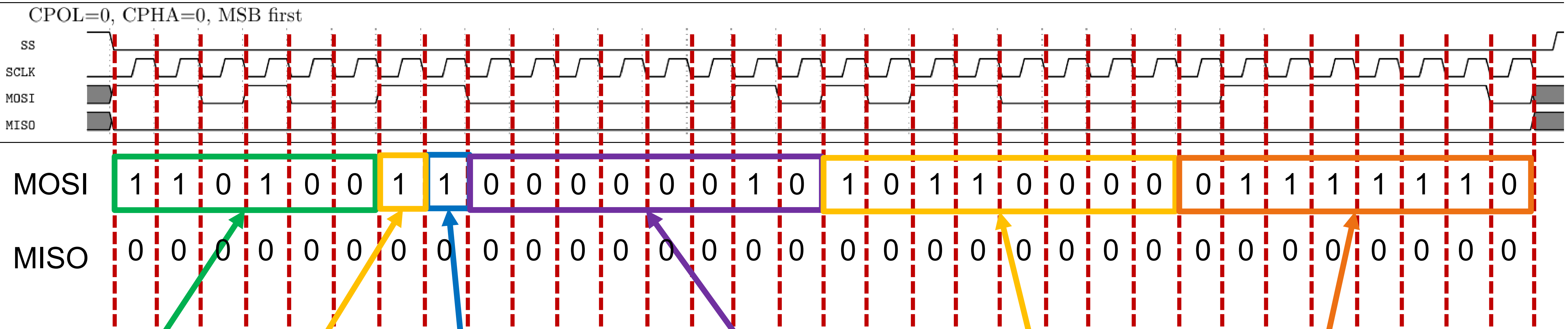


MOSI	0	1	0	1	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MISO	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	1	1	0	0	0	0

Output:

```
RD STREAM 24 0f b0
```

Steaming Write


CPOL=0, CPHA=0, MSB first
SS
SCLK
MOSI
MISO
MOSI 1 1 0 1 0 0 1 1 0 0 0 0 0 0 1 0 1 0 1 1 0 0 0 0 0 1 1 1 1 1 1 0
MISO 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

bits 7:2
address

bit 1
read=0
write=1

bit 0
single=0
stream=1

exchange #2 (MOSI)
stream length
=2

exchange #3 (MOSI)
stream byte 1

exchange #4 (MOSI)
stream byte 2

Output:

```
WR STREAM 34 b0 7e
```

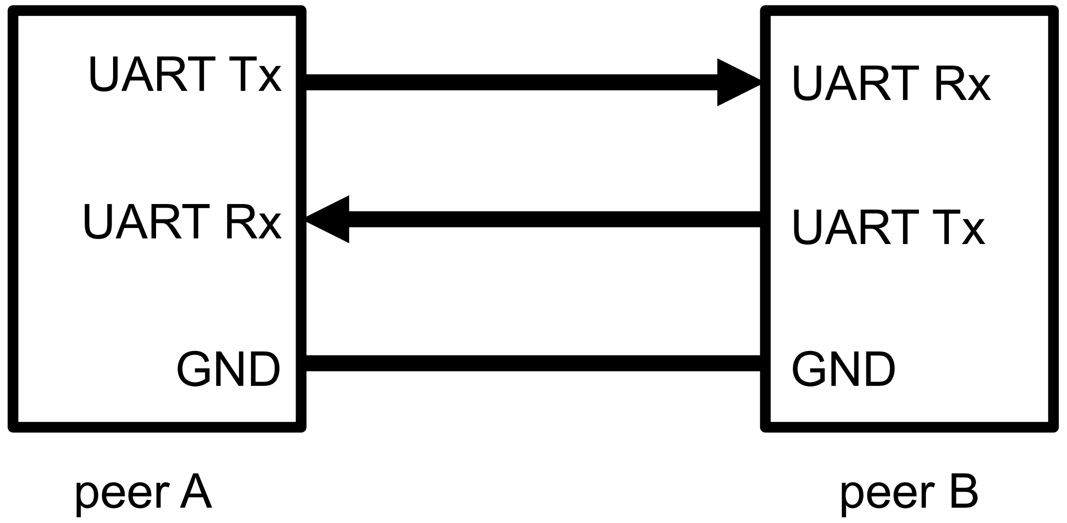
Part 3

- CPOL and CPHA will be provided as static signals

```
sclk    mosi    miso    ss      cpol    cpha
1       1       1       1       1       1
0.0     1       0       0       1       1       1
1000    1       0       0       1       1       1
1050.0 1        0       0       0       1       1
```

UART

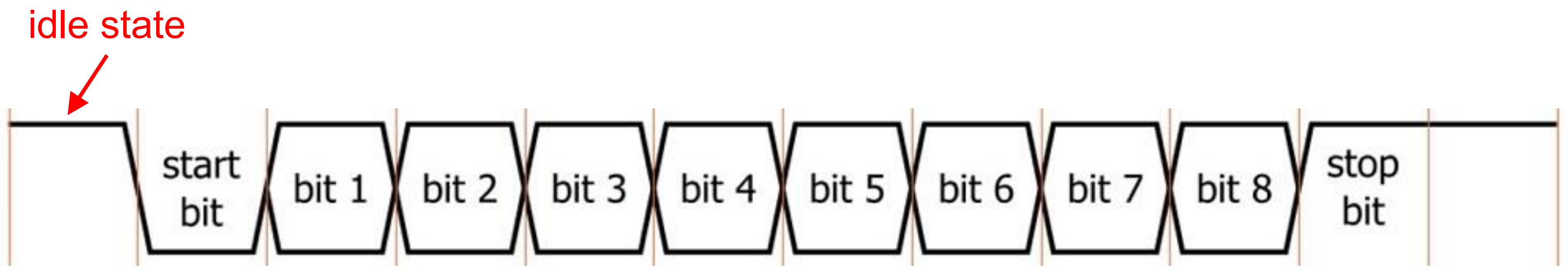
- Universal Asynchronous Receiver/Transmitter
- No clock; transmitter and receiver must agree on bitrate



UART

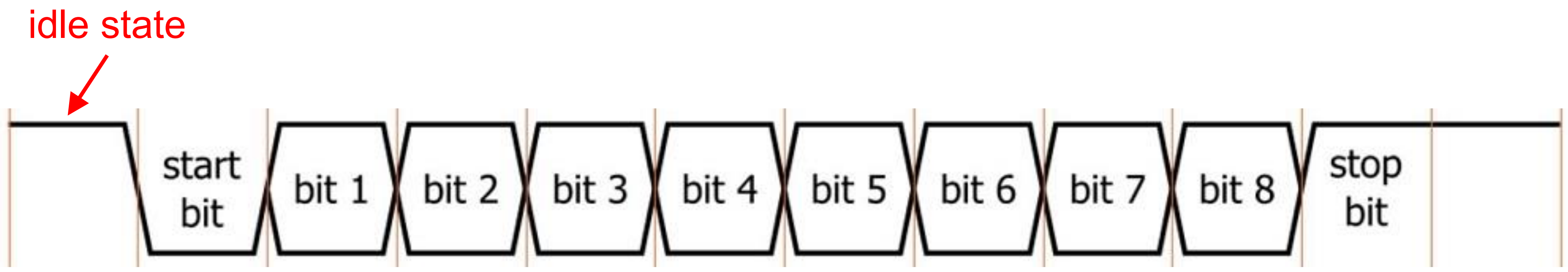
- **Start bit:** The first bit of a one-byte UART transmission
  - Indicates that the data line is leaving its idle state
  - The idle state is typically logic high, so the start bit is logic low

- **Stop bit:** The last bit of a one-byte UART transmission
  - Logic level is the same as the signal's idle state, i.e., logic high



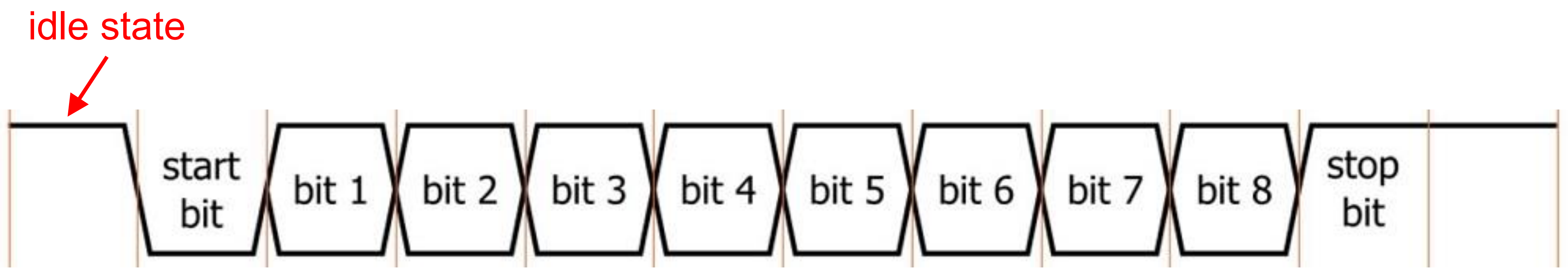
Question

Why is the stop bit 1?



Question

Why is the stop bit 1?



Because it is distingushable from a start bit and allows back-to-back transactions

UART

- **Parity bit:** An error-detection bit added to the end of the byte
  - "odd parity": use parity bit to ensure data+parity has odd number of bits
    - parity bit will be logic high if the data byte contains an *even* number of logic-high bits
  - "even parity": use parity bit to ensure data+parity has even number of bits
    - parity bit will be logic high if the data byte contains an *odd* number of logic-high bits.

- **Baud rate:** The rate in bits per second, or bps at which data can be transferred
  - Includes the start, stop, and parity bits

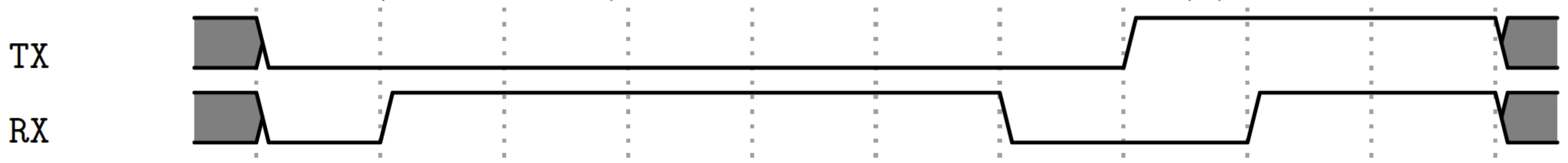
- **Throughput:** Data transmission rate without control bits
  - baud rate * (# data bits) / (# data+start+parity+stop bits)

UART

In the following UART transaction, what bytes are exchanged?

LSB to MSB 0000001
MSB to LSB 100 0000
0x40

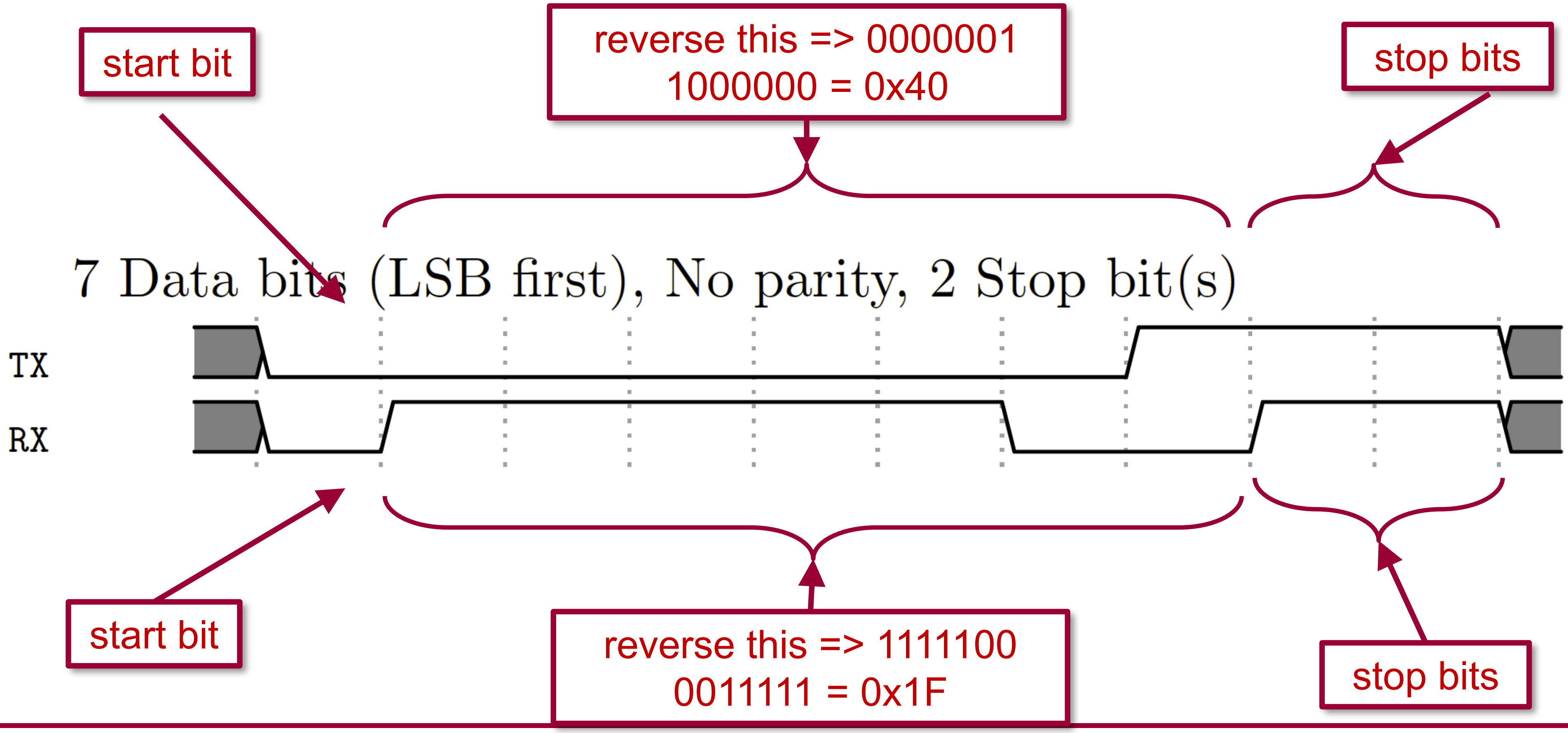
7 Data bits (LSB first), No parity, 2 Stop bit(s)



NOTE that the transmit and the receive transactions do not need to be aligned, as shown here.

UART

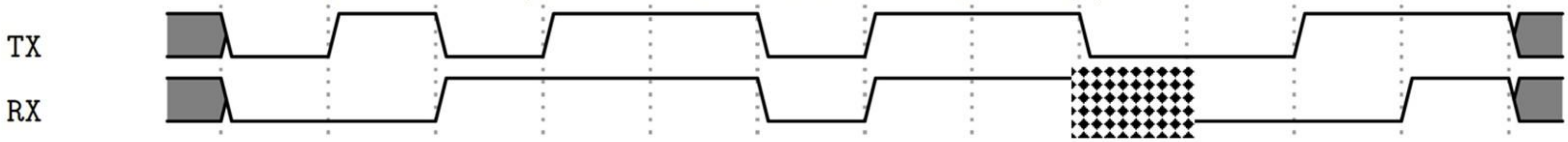
In the following UART transaction, what bytes are exchanged?



UART

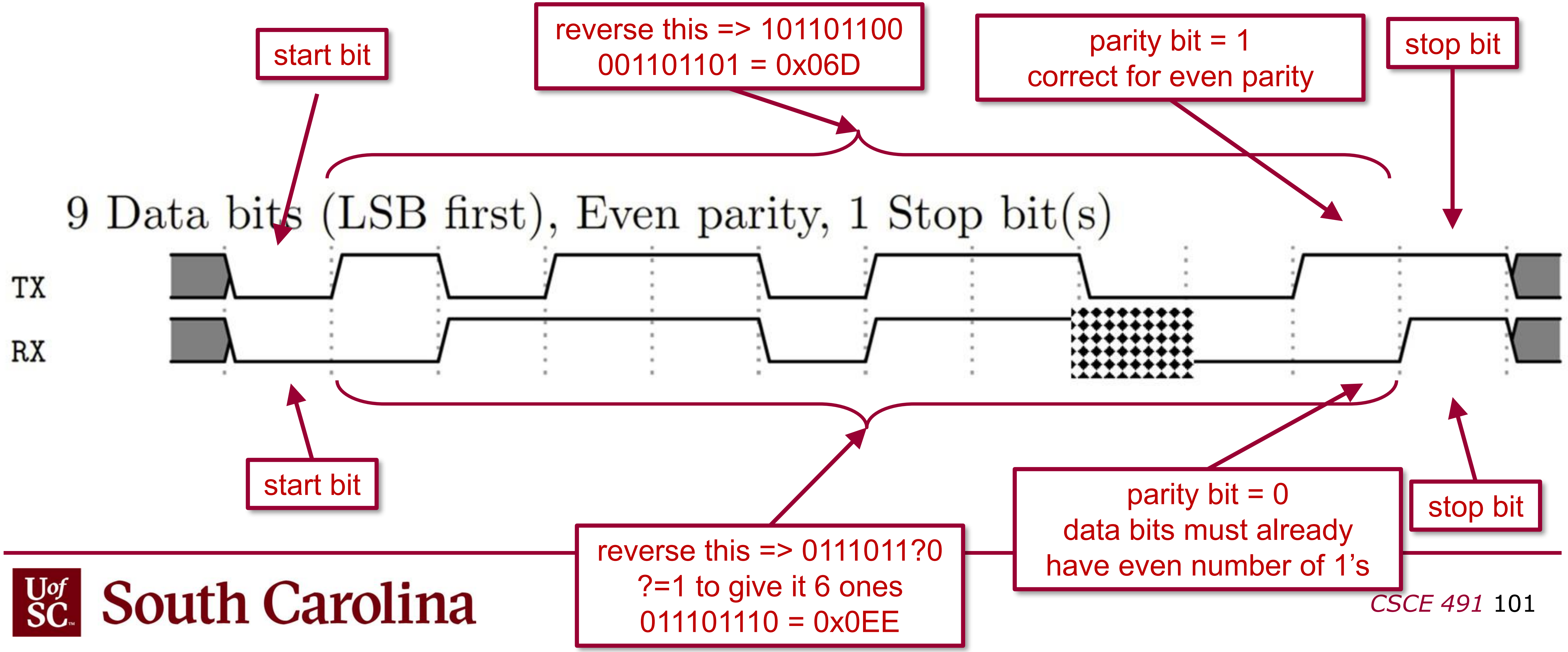
In the following UART transaction, what bytes are exchanged?

9 Data bits (LSB first), Even parity, 1 Stop bit(s)



UART

In the following UART transaction, what bytes are exchanged?


start bit
reverse this => 101101100
001101101 = 0x06D
parity bit = 1
correct for even parity
stop bit
9 Data bits (LSB first), Even parity, 1 Stop bit(s)
TX
RX
start bit
reverse this => 0111011?0
?=1 to give it 6 ones
011101110 = 0x0EE
parity bit = 0
data bits must already have even number of 1's
stop bit

UART

Assume a UART channel operates at baud rate = 9600 bps and has 7 data bits, one parity bit, and one stop bit. What is its throughput?

UART

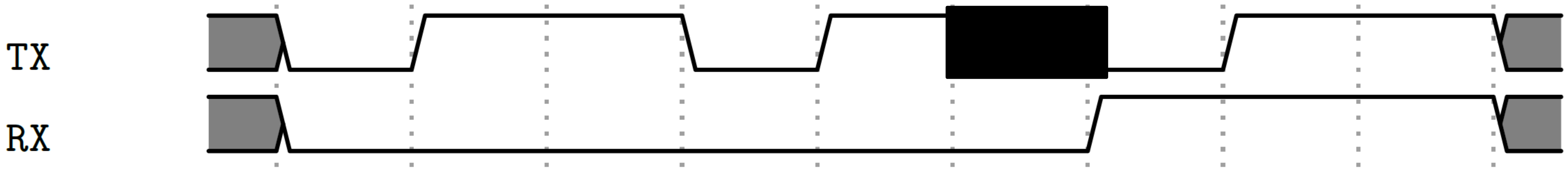
Assume a UART channel operates at baud rate = 9600 bps and has 7 data bits, one parity bit, and one stop bit. What is its throughput?

7 data bits for every start bit, 7 data bits, parity bit, and stop bit, so there’s 7 bits for every 1+7+1+1=10 bits. Thus only 7/10=70% of the bits are used for data. The throughput is 70% x 9600 = 6720 bits per second.

UART Example

What bit is missing?

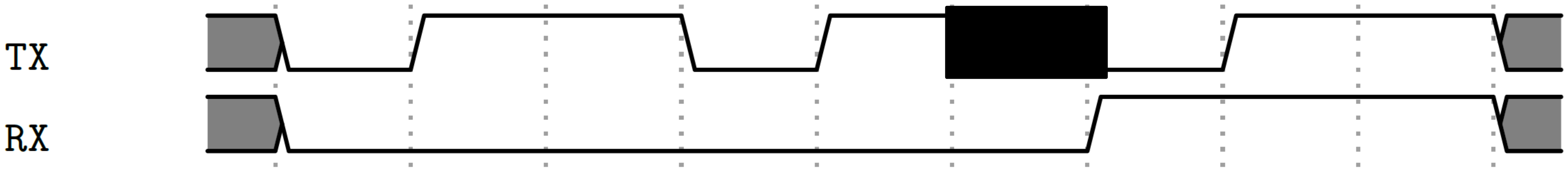
5 Data bits (LSB first), Odd parity, 2 Stop bit(s)



UART Example

What bit is missing?

5 Data bits (LSB first), Odd parity, 2 Stop bit(s)

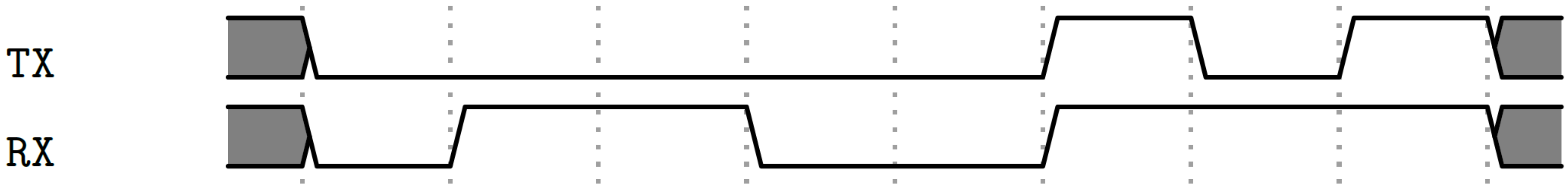


0

UART Example

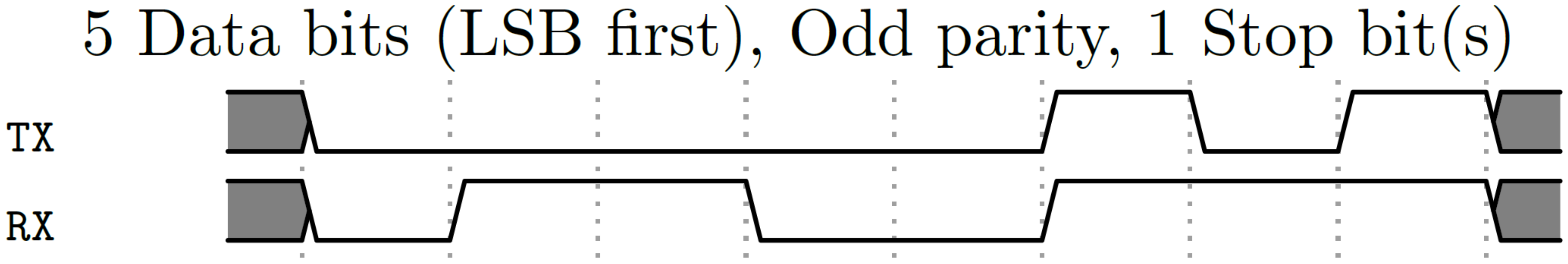
Are there errors in the transaction below?

5 Data bits (LSB first), Odd parity, 1 Stop bit(s)



UART Example

Are there errors in the transaction below?



Received parity is incorrect

Reminders

- Take Quiz 2 and Quiz 3, due this Friday (Feb. 13)
- Next class will meet in Swearingen 3D22

Shared Connections

- SPI can only have one master
  - Only one entity can drive MOSI, SS, and SCLK

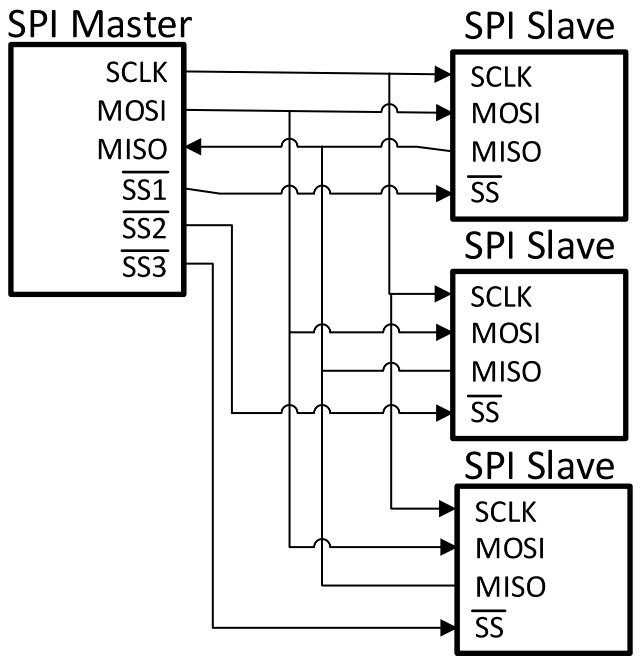
- Multiple slaves
  - Option 1:
    - Connect MOSI and SCLK to all slaves, one SS to each slave
    - All slaves share MISO
  - Option 2:
    - Connect SCLK and SS to all slaves
    - Daisy chain slaves: Connect MISO of slave 1 to MOSI to slave 2 and MISO of slave 2 to MISO of master

SPI (Serial Peripheral Interface)

Option 1:

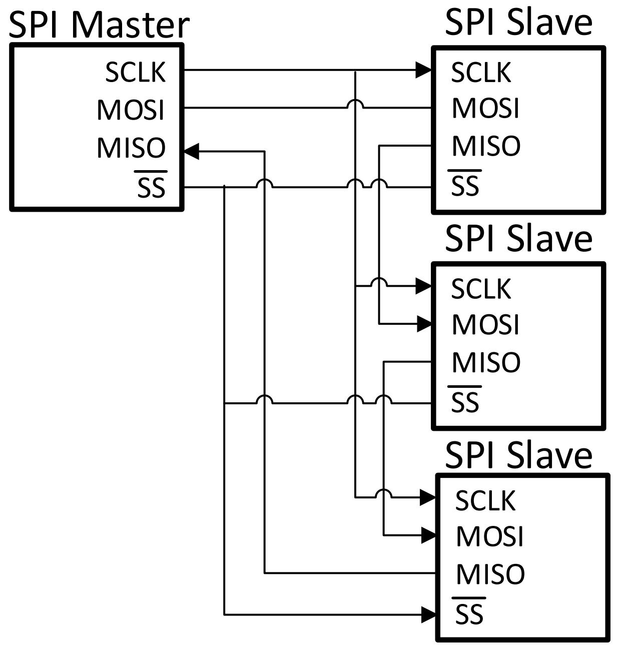
Independent

8 bits per exchange
1 slave at a time



Option 2:

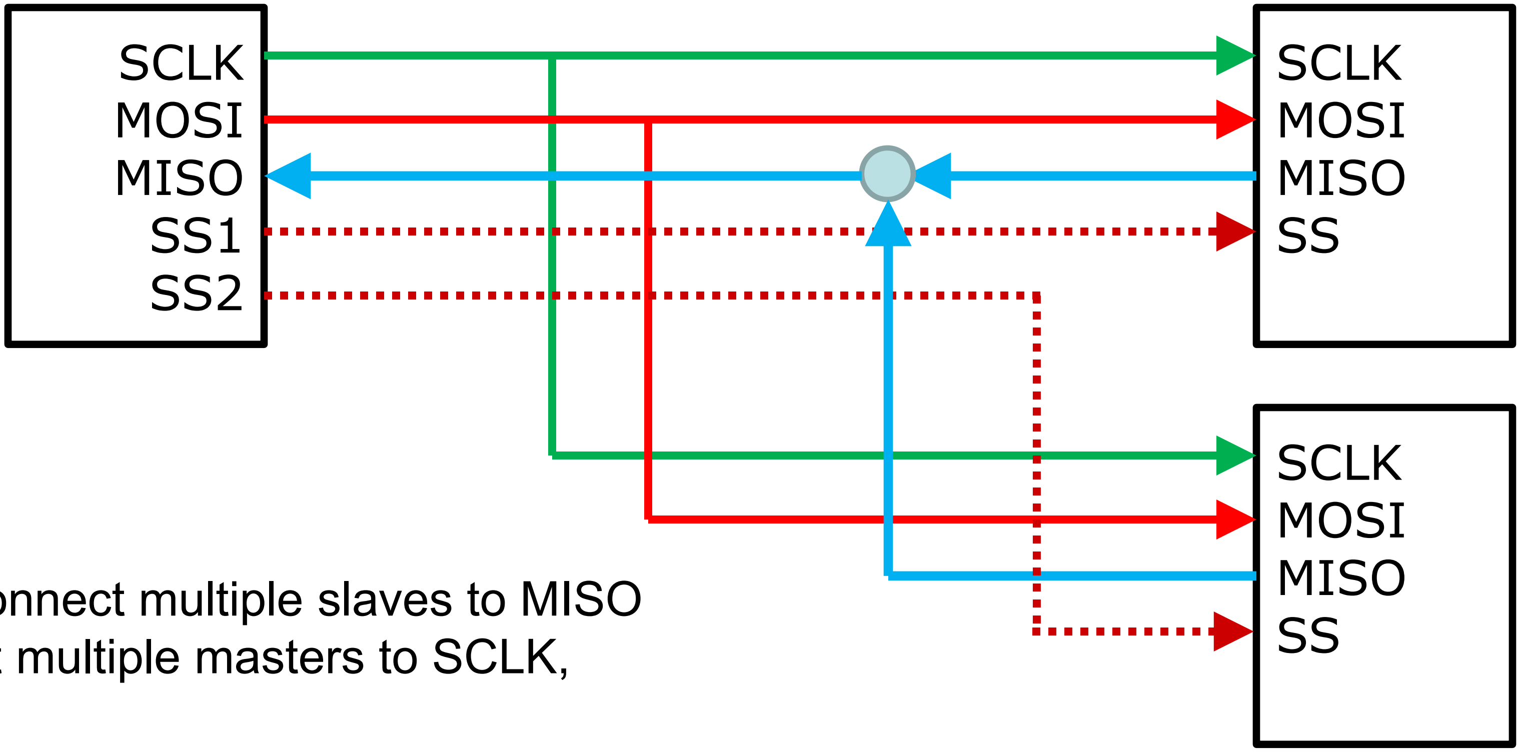
Daisy-chained



8x3=24 bits per exchange
All slaves

Multi-Slave SPI

Q: What value is driven on the MISO wire of a non-daisy chained multiple-slave SPI bus when the corresponding SS wire is 1? A: high impedance / not connected



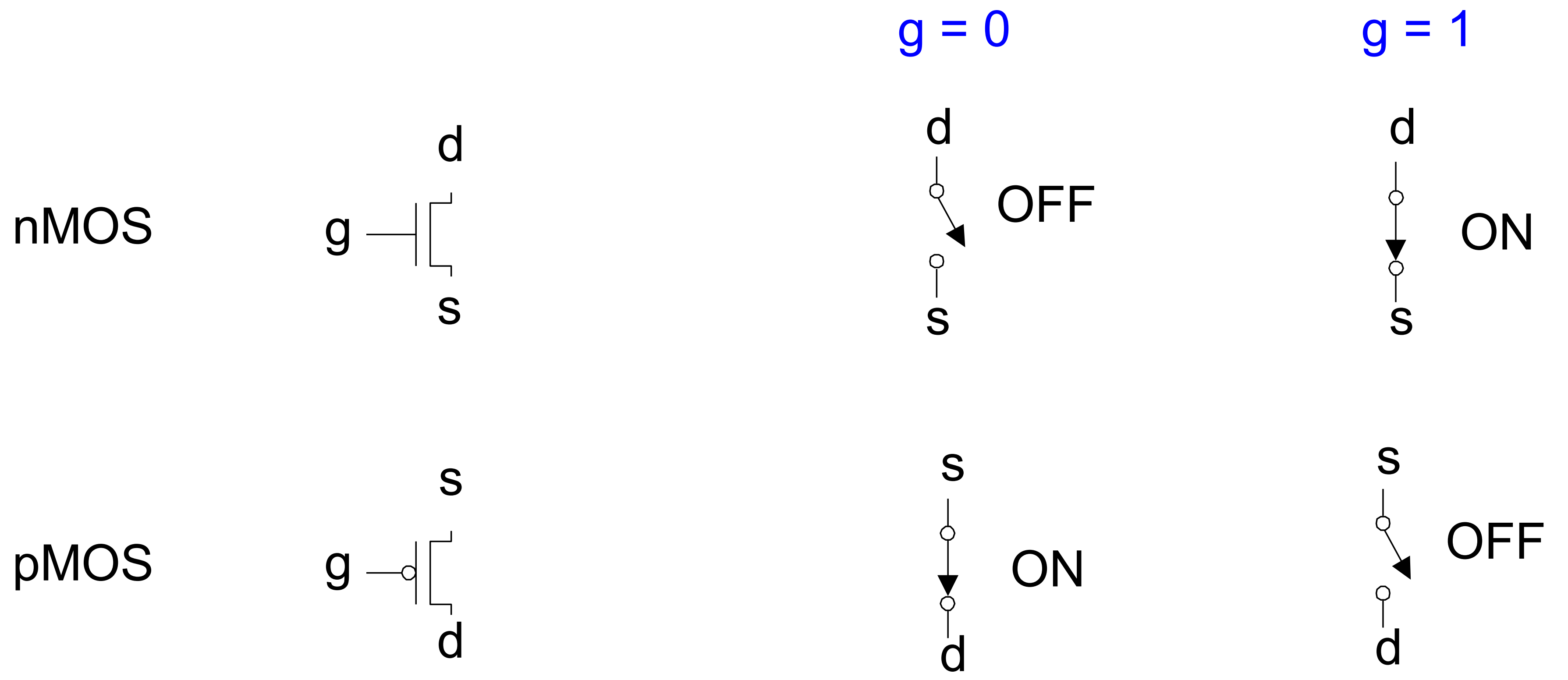
- Use tri-states to connect multiple slaves to MISO
- No way to connect multiple masters to SCLK, MOSI, SS!

Multi-Master Protocol

- What would it take for a bus to be multi-master?

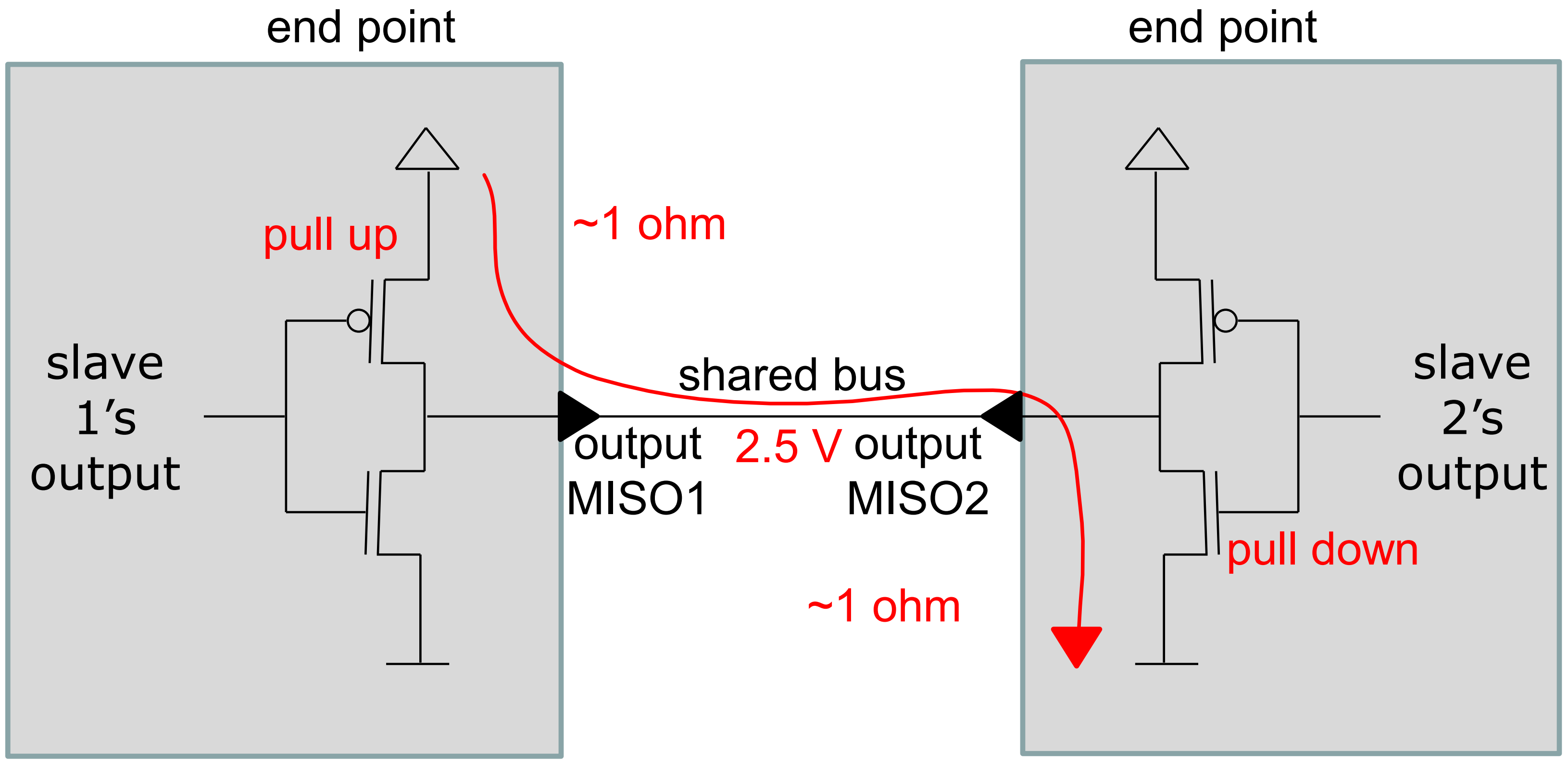
- Issues to resolve:
  1. Need to ensure that two hosts driving the same wire can't cause problems
  2. During a transaction, need to tightly coordinate hand-offs on shared output

Transistor Function



"Push-Pull"-Type Interface

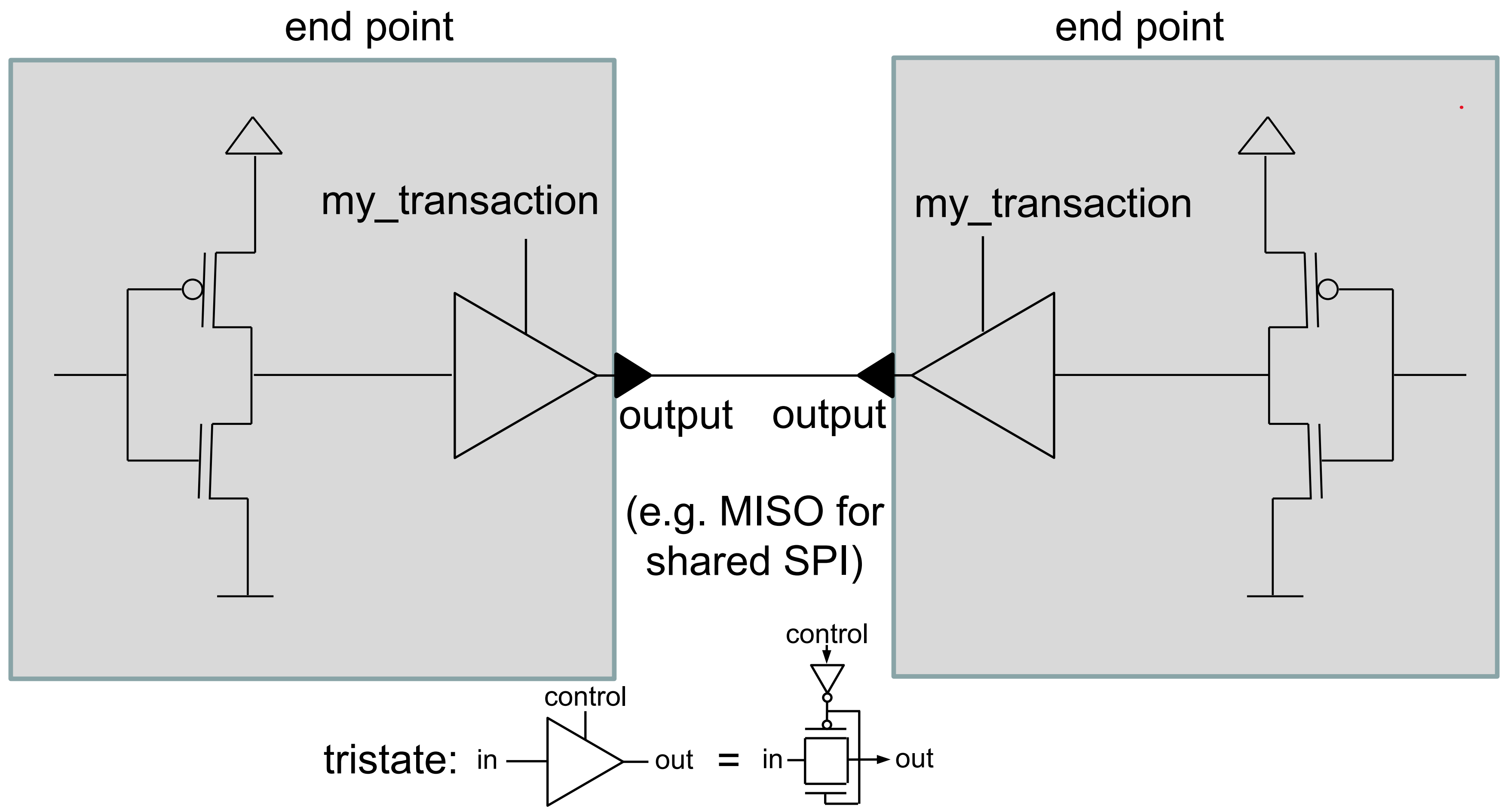
Drive the output to 1



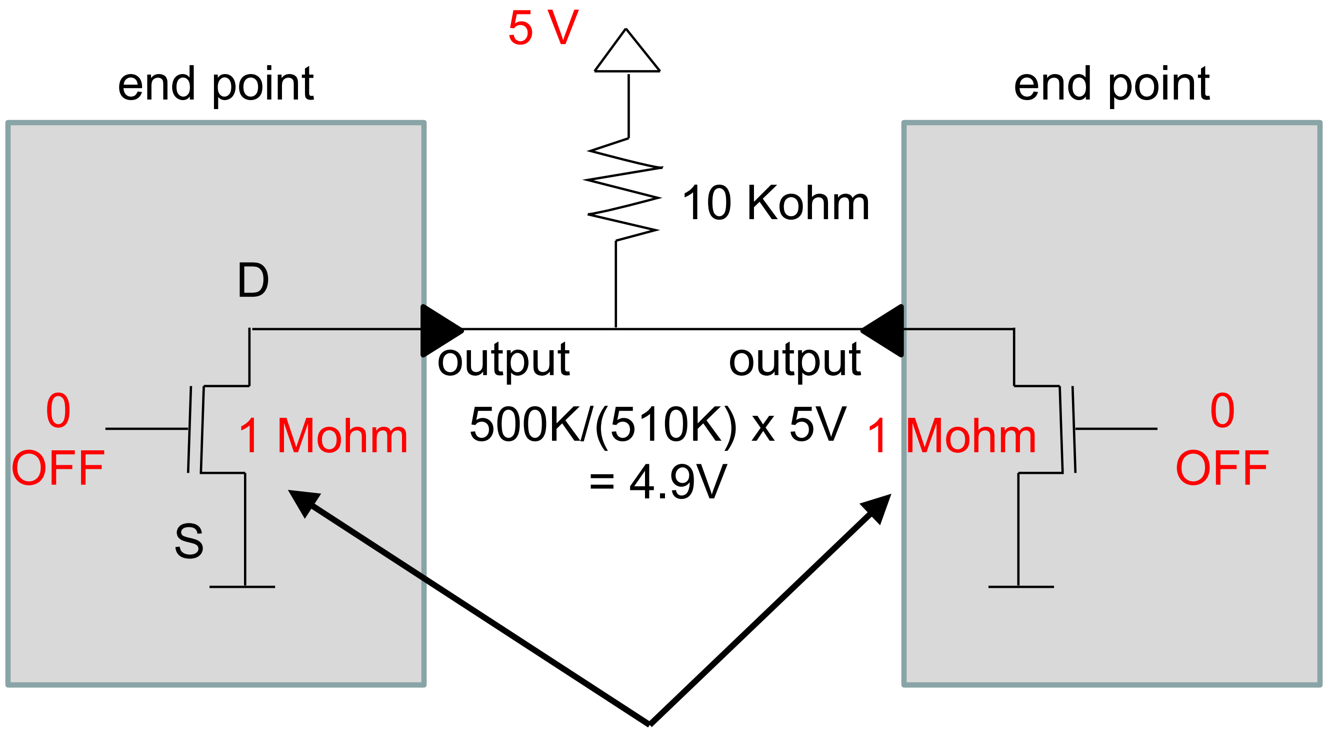
Drive the output to 0

5 V over 2 ohm = 2.5 amps!

"Push-Pull"-Type Interface

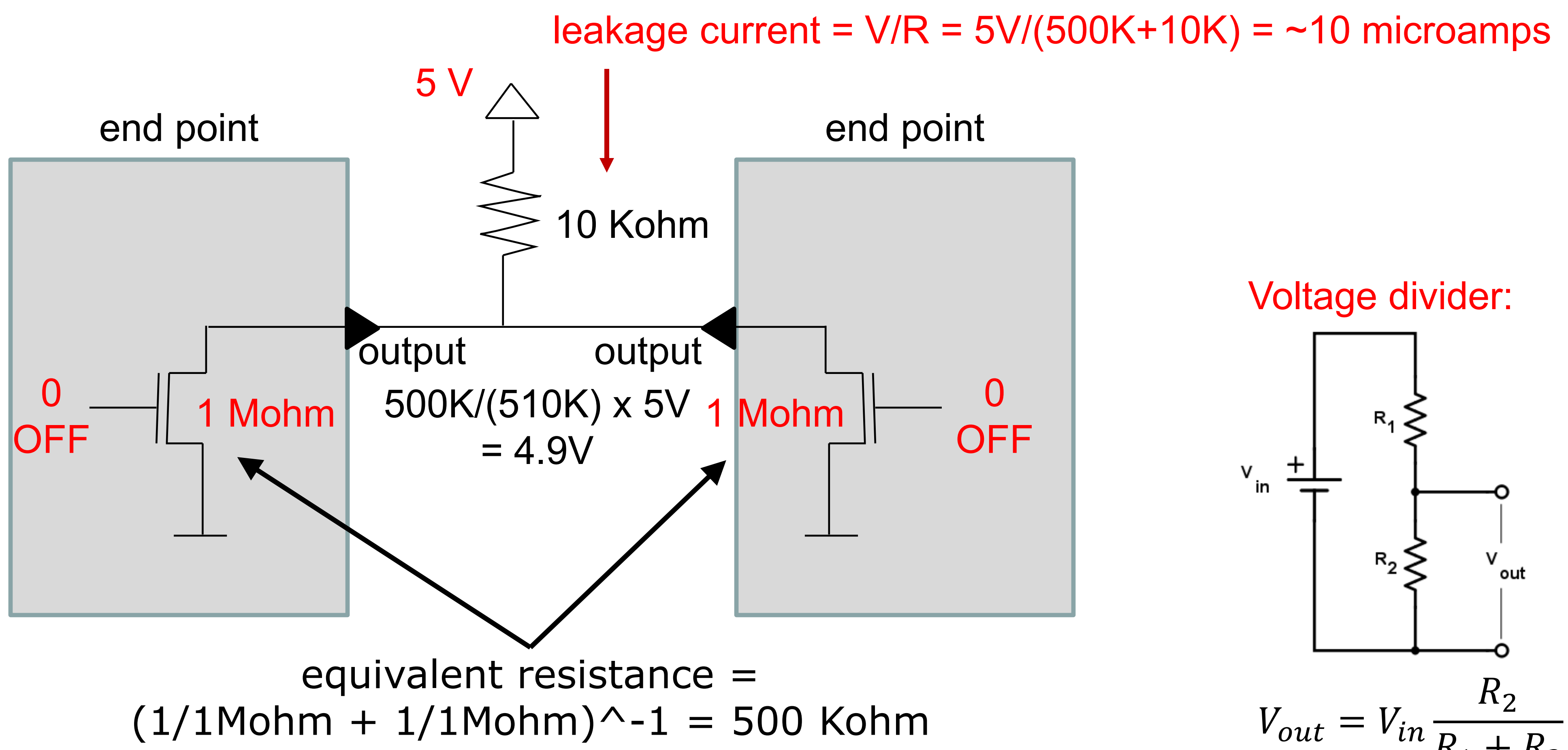


Open Drain-Type Interface

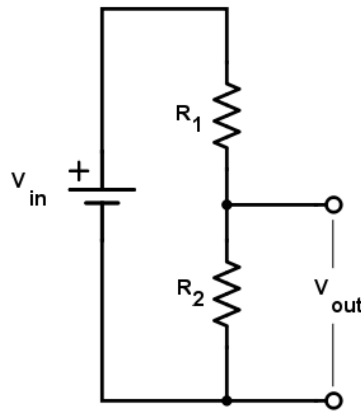


equivalent resistance =
(1/1Mohm + 1/1Mohm)^-1 = 500 Kohm

Open Drain-Type Interface

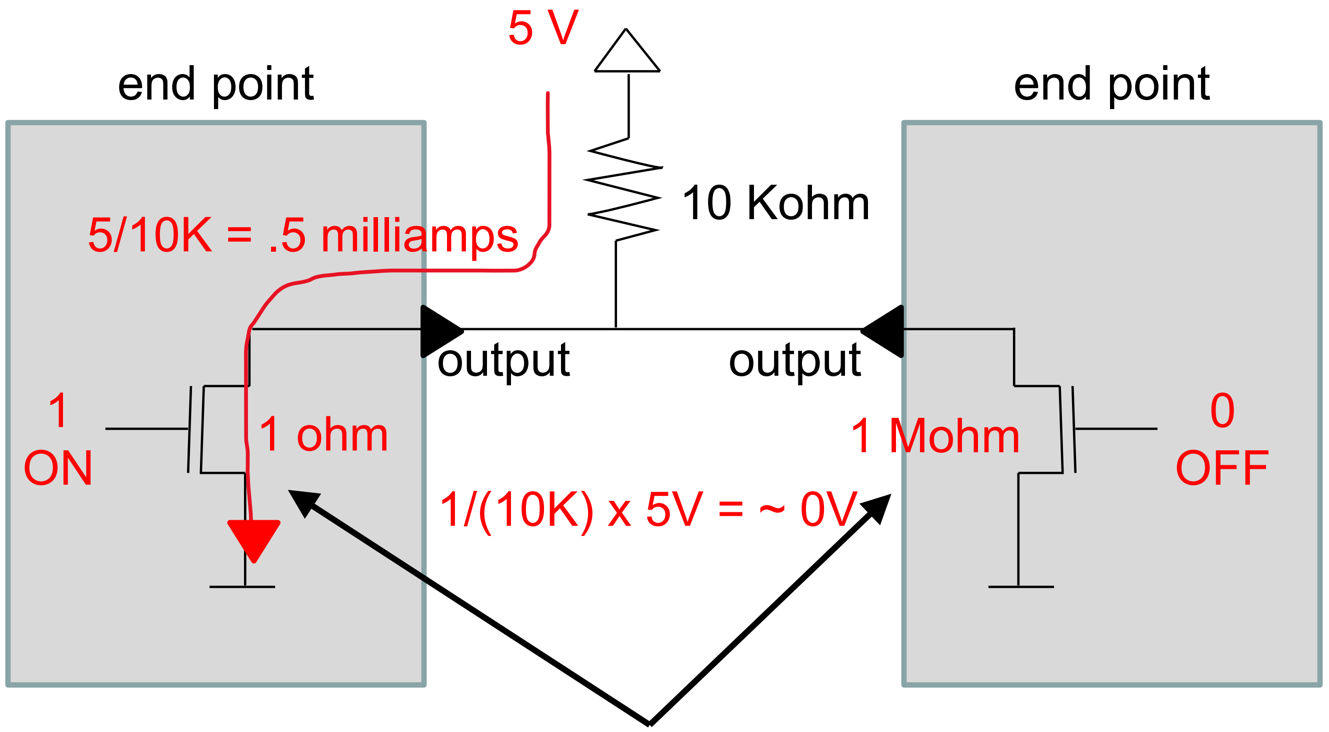

leakage current = V/R = 5V/(500K+10K) = ~10 microamps
5 V
10 Kohm
end point
end point
output
output
0
OFF
1 Mohm
1 Mohm
0
OFF
500K/(510K) x 5V = 4.9V
equivalent resistance = (1/1Mohm + 1/1Mohm)^-1 = 500 Kohm

Voltage divider:



$$V_{out} = V_{in}\frac{R_2}{R_1 + R_2}$$

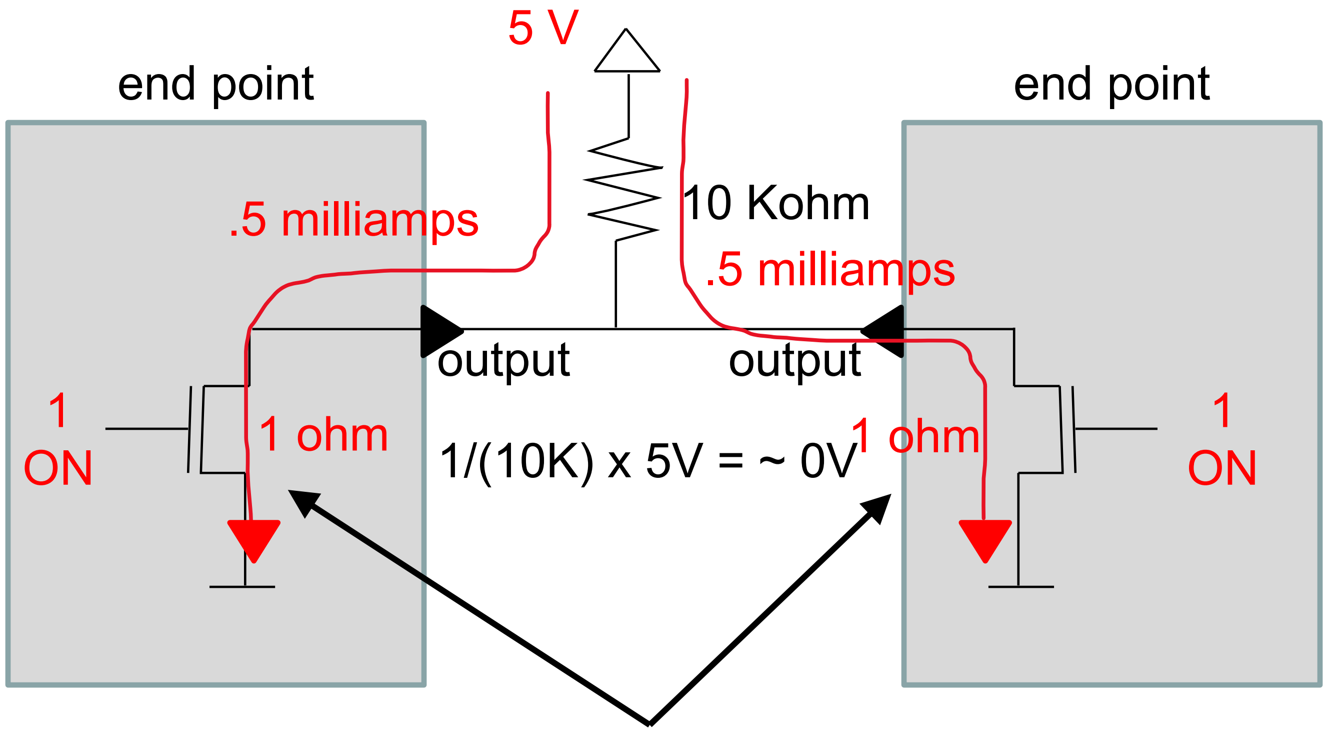
Open Drain-Type Interface



equivalent resistance =
(1/1Mohm + 1/1Mohm)^-1 = 500 Kohm

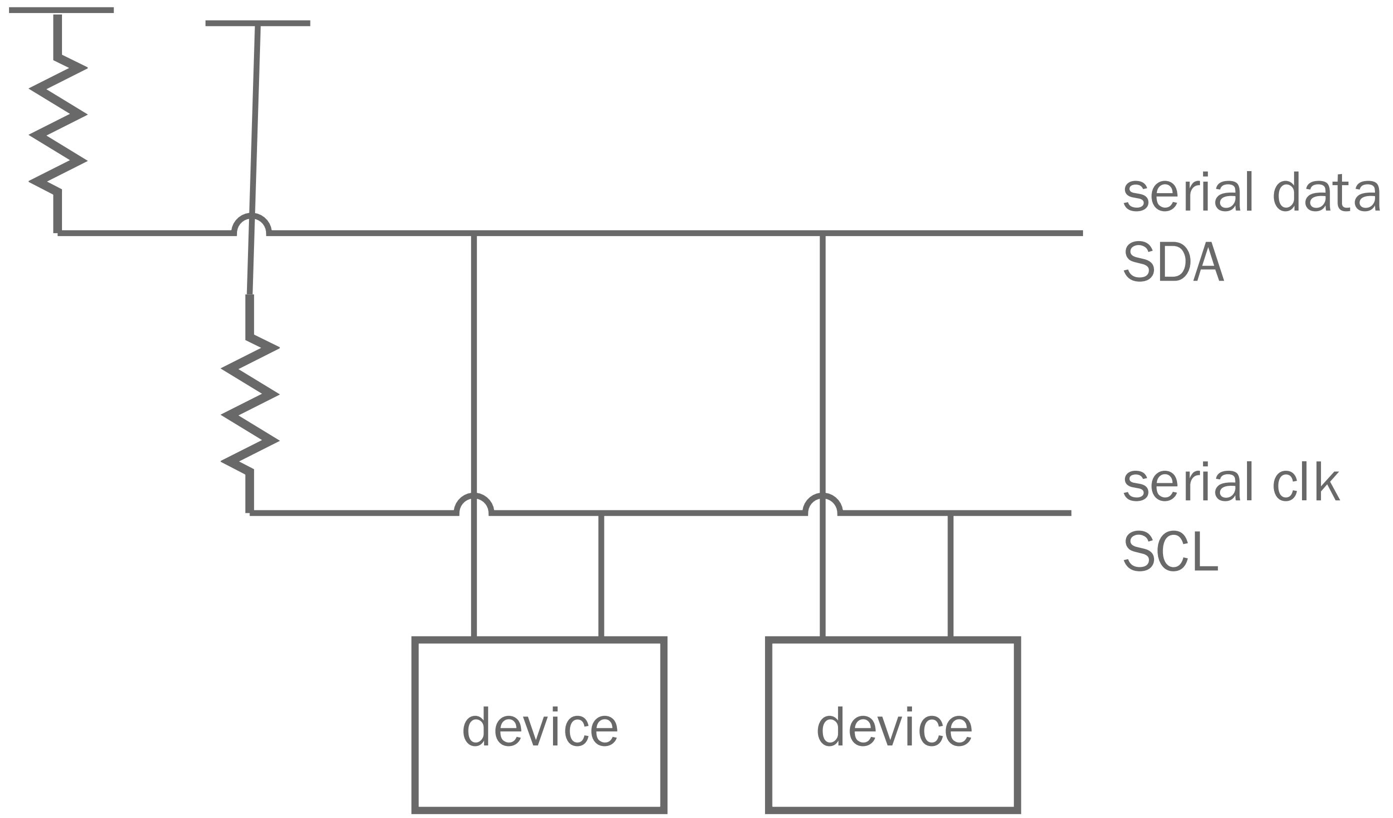
Note: push-pull sources ~100X less current when driving 1

Open Drain-Type Interface



equivalent resistance =
(1/1Mohm + 1/1Mohm)^-1 = 500 Kohm

I^{2}C (Inter-Integrated Circuit)



I^{2}C

If two devices are attached to an open-drain bus, what is the logic value of the bus if one device sends a 1 and another sends a 0?

I^{2}C

If two devices are attached to an open-drain bus, what is the logic value of the bus if one device sends a 1 and another sends a 0?

0

SPI vs I^{2}C

- Like SPI, I^{2}C is synchronous
- Unlike SPI, I^{2}C:
  - …is multi-master ☺
  - …requires 2 wires instead of 4 ☺
  - …includes a acknowledgement mechanism to allow the sender to know if each part of the transaction—address and data—were successfully received ☺
  - …has standard bit ordering ☺
  - …has no predefined settings (e.g. CPOL/CPHA, parity bit, #stop bits) ☺
    - Just the speed mode
  - …all transfers use programmed I/O behavior ☺
  - …allows for stalls during transmissions ☺

  - …is half-duplex ☹

I^{2}C

- All data sent as 8-bit bytes from MSB to LSB (UART is opposite!)
- Initiator: master
- Receiver: slave
- Bits are gated when SCL is high (level sensitive)
- During a transaction, SDA only changes when SCL is LOW
- The slave acknowledges address bits
  - Master releases SDA
  - Slave pulls SDA low for an ACK or leaves high for NACK
- The slave acknowledges data bits on write
- The master acknowledges data bits on read
- Changing SDA when SCL is HIGH signals a control message (start or stop)
  - Start bit: high-to-low transition of SDA
  - Stop bit: low-to-high transisiton of SDA

I^2C

- Arbitration:
  - Device starts when bus is inactive
  - When two devices start, both can detect a cycle where both devices attempt to send a different bit
- Each endpoint has an address, which may be fixed or programmable
- Transfer format:

	start	address	R/W’	ack	data	ack	…	stop
bits	1*	7**	1	1	8	1		1*

* more of a signal than a bit
** 10-bit addresses also also possible

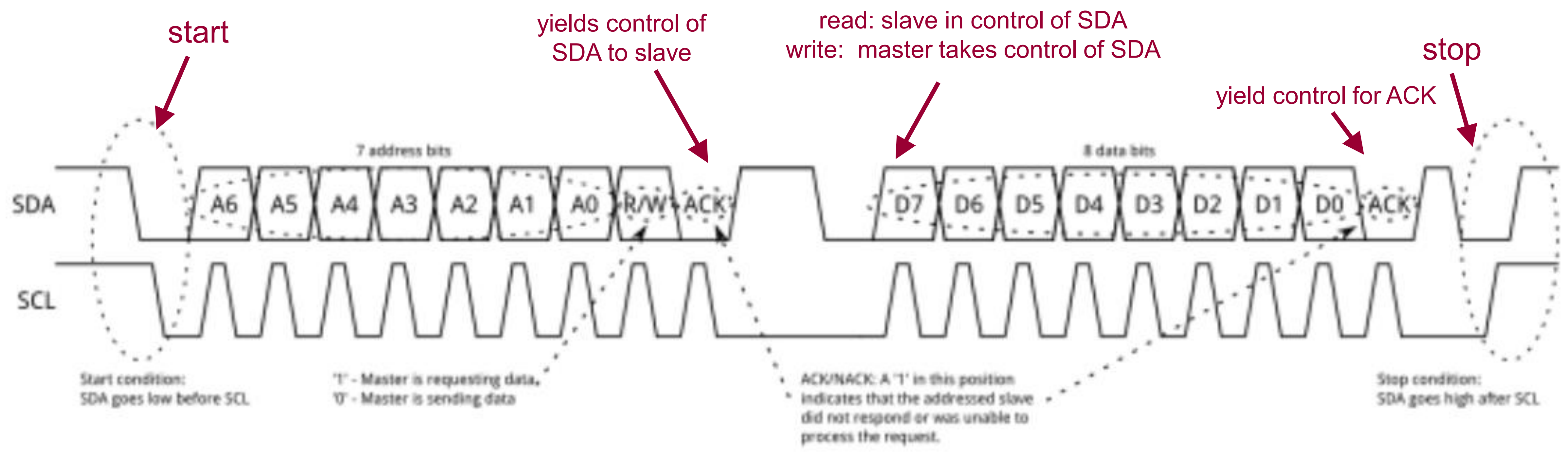
I^{2}C

- One transaction always contains address phase (9 bits) and data phase (9 bits)
- Master controls SCL always
- For write operation
  - Master controls SDA for 8 cycles of address phase and 8 cycles of data phase
  - Slave controls SDA for 1 cycle of addresss phase and 1 cycle of data phase
- For read operation
  - Master controls SDA for 8 cycles of address phase and 1 cycle of data phase
  - Slave controls SDA for 1 cycle of addresss phase and 8 cycles of data phase
- Start bit: high-to-low transition of SDA while SCL is high
- Stop bit: low-to-high transisiton of SDA while SCL is high

I2C

- 4 most important things to know:
  1. Master controls SCL, control of SDA changes hands
  2. Each "side" of the transaction is 9 bit times (8 + ACK)
  3. SDA does not change when SCL is high, except for start/stop signals
  4. Data is latched when SCL is high

I^{2}C

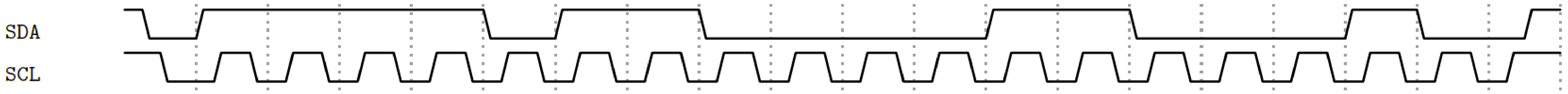

start
yields control of SDA to slave
read: slave in control of SDA
write: master takes control of SDA
stop
yield control for ACK
7 address bits
8 data bits
SDA
SCL
A6
A5
A4
A3
A2
A1
A0
R/W
ACK
D7
D6
D5
D4
D3
D2
D1
D0
ACK
Start condition: SDA goes low before SCL
'1' - Master is requesting data, '0' - Master is sending data
ACK/NACK: A '1' in this position indicates that the addressed slave did not respond or was unable to process the request.
Stop condition: SDA goes high after SCL

Master always in control of SCL

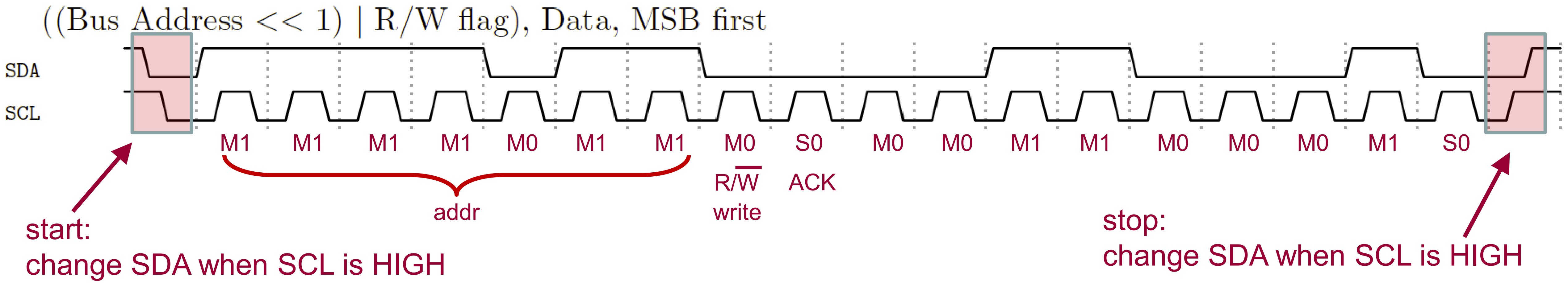
R/W flag: 0 (write) 1 (read)

I^{2}C Transaction

((Bus Address << 1) | R/W flag), Data, MSB first



I^{2}C Transaction


((Bus Address << 1) | R/W flag), Data, MSB first
SDA
SCL
M1
M1
M1
M1
M0
M1
M1
M0
S0
M0
M0
M1
M1
M0
M0
M0
M1
S0
addr
R/$\overline{W}$
write
ACK
start:
change SDA when SCL is HIGH
stop:
change SDA when SCL is HIGH

I^{2}C

What is the stop condition of I^{2}C?

a. rising edge of SDA when SCL is 1

b. rising edge of SCL when SDA is 1

c. rising edge of SDA when SCL is 0

d. rising edge of SCL when SDA is 0

I^{2}C

What is the stop condition of I^{2}C?

a. rising edge of SDA when SCL is 1
b. rising edge of SCL when SDA is 1
c. rising edge of SDA when SCL is 0
d. rising edge of SCL when SDA is 0

I2C Transaction: Which Side is in Control?

- SCL: always the master
  - Slave can hold SCL down for a feature called clock stretching
- SDA:
  - In a read:
    - first 8 bits: master
    - bits 9 to 17: slave
    - bit 18: master
  - In a write:
    - first 8 bits: master
    - bit 9: slave
    - bits 10 to 17: master
    - bit 18: slave

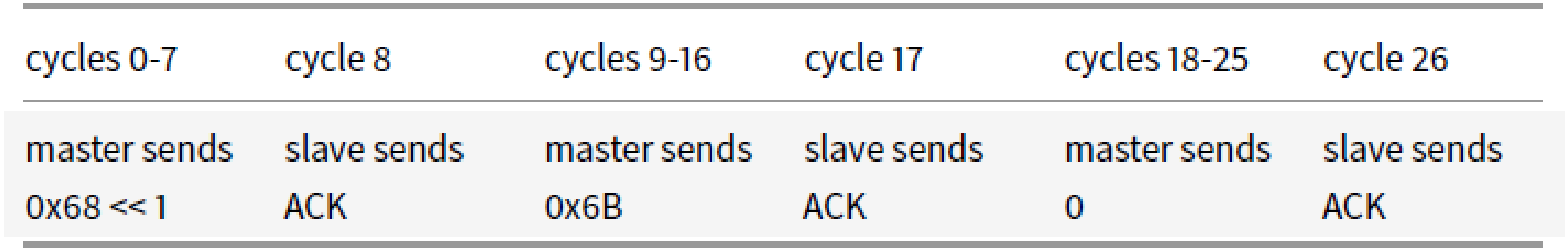
Lab 2: Communicating with the MPU-6050 IMU

- I2C communication at up to 400 KHz (2.5 μs/bit)
- 3-axis accelerometer, ranges +/-2g, +/-4g, +/-8g, +/-16g
- Goal: compute angle of orientation of the sensor

- I2C interface:
  - Device address: 0x68
  - The device has 82 registers
- To write (one I2C transaction):
  - Write the register address to 0x68
  - Send a third byte with the desired value
- To read (two I2C transactions):
  - Write the register address to 0x68
  - Read from address 0x68 (can read >1 bytes)

Example

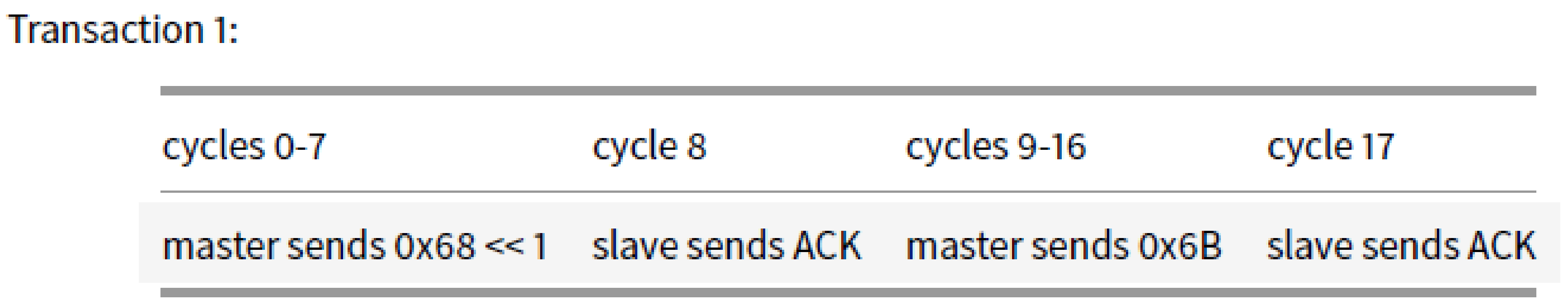
- To write value 0 to register 0x6B:



cycles 0-7	cycle 8	cycles 9-16	cycle 17	cycles 18-25	cycle 26
master sends 0x68 << 1	slave sends ACK	master sends 0x6B	slave sends ACK	master sends 0	slave sends ACK

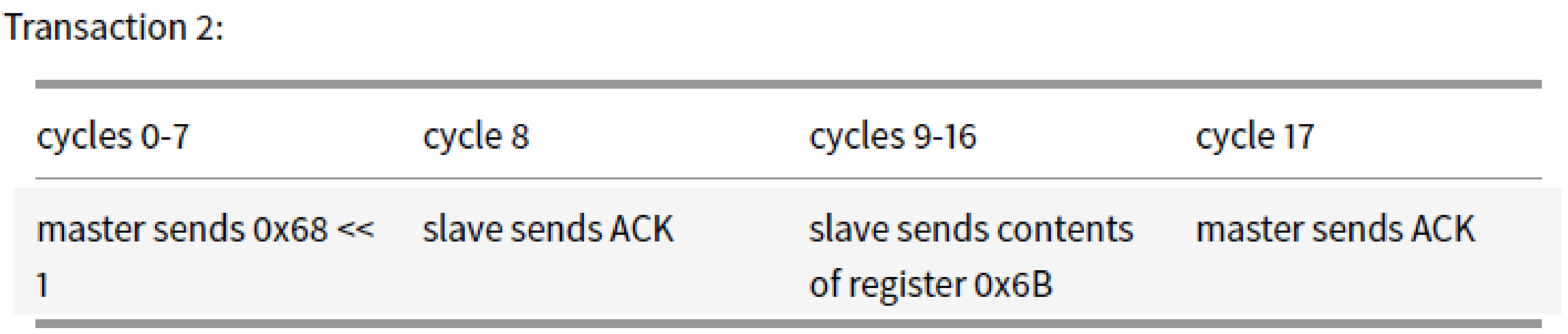
Example

- To read register 0x6B:



Transaction 1:

cycles 0-7	cycle 8	cycles 9-16	cycle 17
master sends 0x68 << 1	slave sends ACK	master sends 0x6B	slave sends ACK

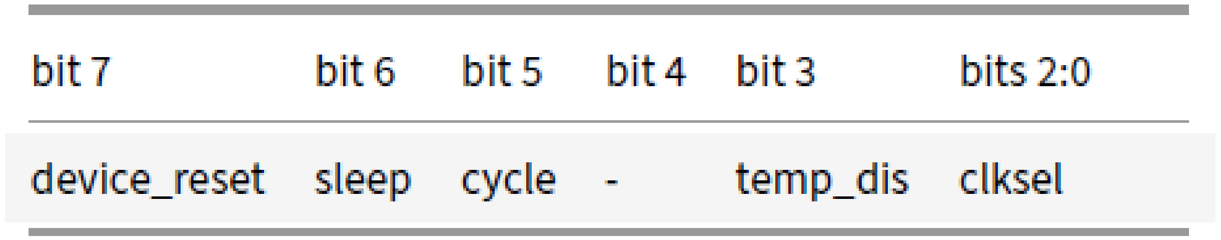


Transaction 2:

cycles 0-7	cycle 8	cycles 9-16	cycle 17
master sends 0x68 << 1	slave sends ACK	slave sends contents of register 0x6B	master sends ACK

Registers

Address	Purpose
0x3B	X accel [15:8]
0x3C	X accel [7:0]
0x3D	Y accel [15:8]
0x3E	Y accel [7:0]
0x3F	Z accel [15:8]
0x40	Z accel [7:0]
0x6B	Power management register



bit 7	bit 6	bit 5	bit 4	bit 3	bits 2:0
device_reset	sleep	cycle	-	temp_dis	clksel

(write to 0 on startup)

Bit Banging I2C

- Define pin numbers:

```
#define SCL 1
#define SDA 2
```

- Pulse SCL:

```
pinMode(SCL, OUTPUT_OPEN_DRAIN);
digitalWrite(SCL, HIGH);
delayMicroseconds(10);
digitalWrite(SCL, LOW);
```

- Set SDA:

```
pinMode(SDA, OUTPUT_OPEN_DRAIN);
digitalWrite(SDA, LOW);
```

- Read SDA:

```
pinMode(SDA, INPUT_PULLUP);
digitalRead(SDA);
```

Conversion from Acceleration to Angle of Orientation

- Acceleration is signed 16-bit

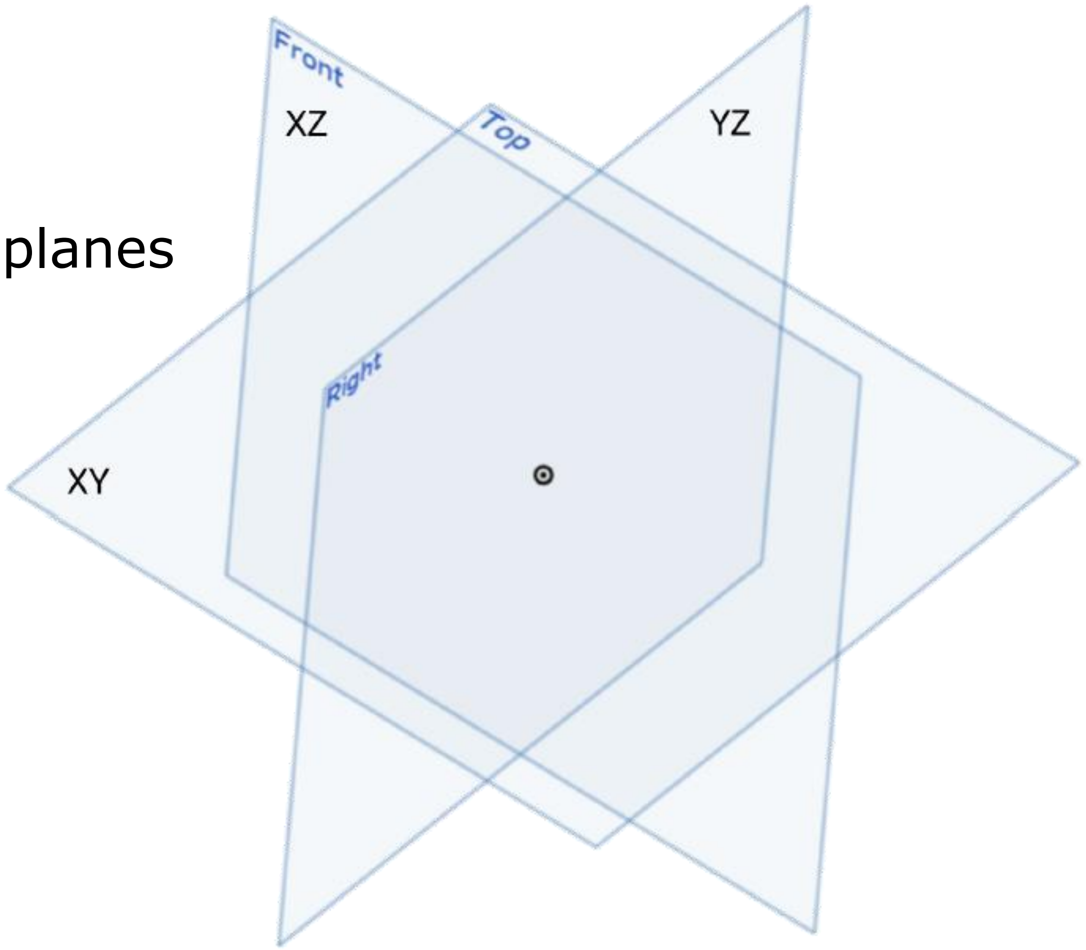
  -32768 to 32767 => -2 g to +2 g

- Goal: measure orientation of XY, XZ, and YZ planes

  XY = atan2(accel_x,accel_y)

  XZ = atan2(accel_x,accel_z)

  YZ = atan2(accel_y,accel_z)



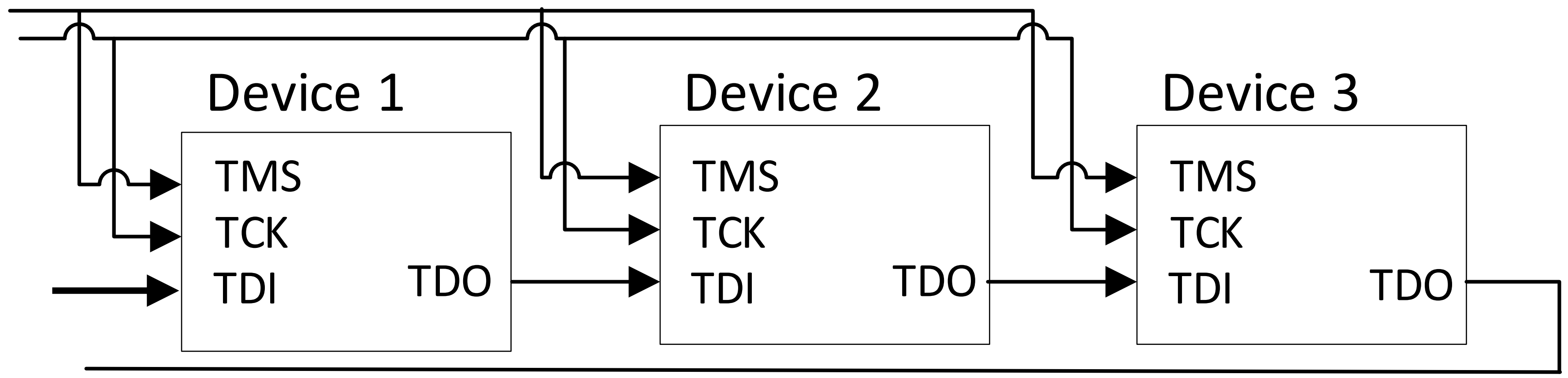
JTAG

- Designed for testing and debugging

- Signals:
  - TDI: test data in
  - TDO: test data out
  - TCK: test clock
  - TMS: test mode select (controls state machine)
  - TRST: test reset (optional)

- Reduced pin count:
  - TMSC: test serial data
  - TCKC: test clock

JTAG

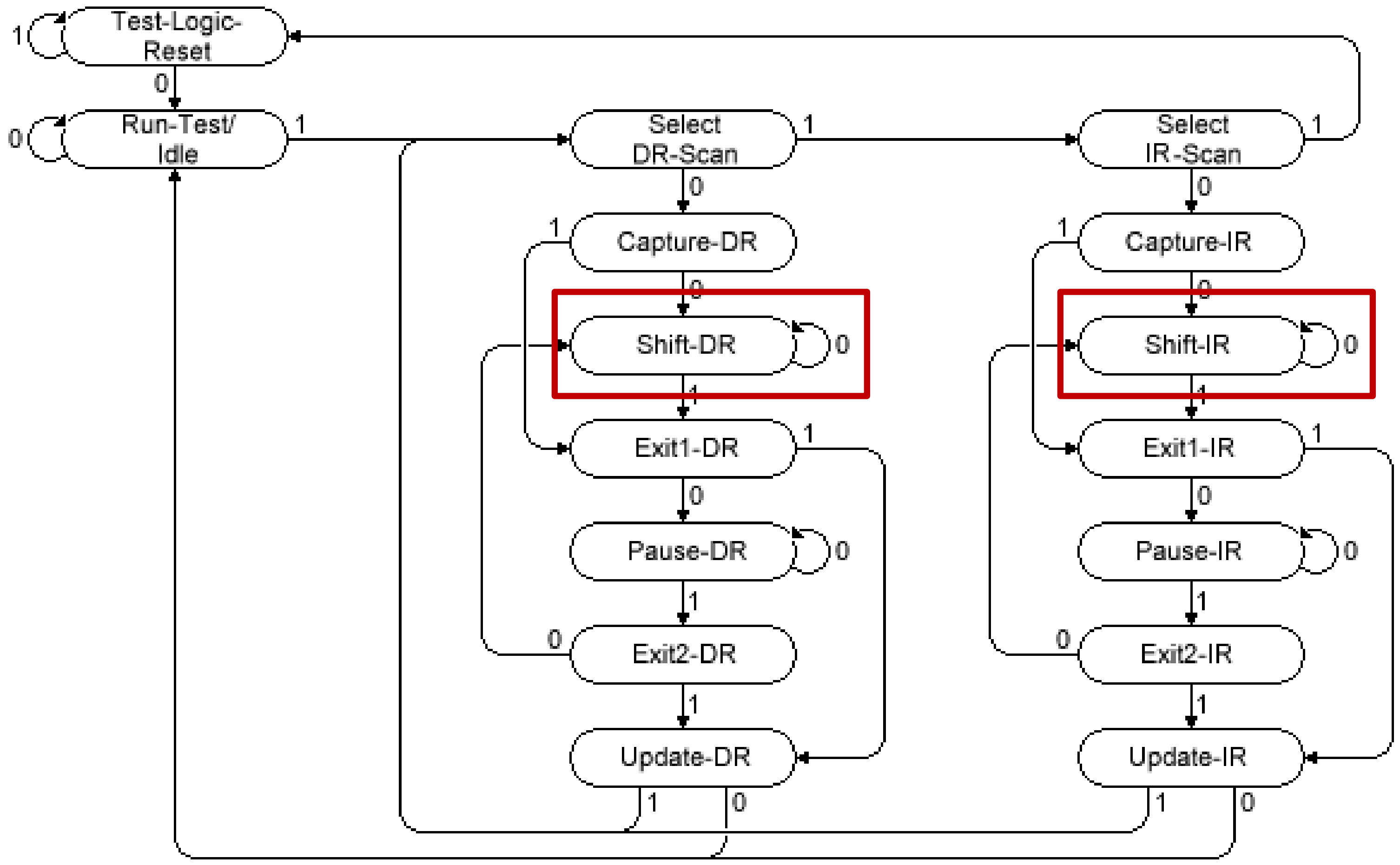

Device 1
TMS
TCK
TDI
TDO
Device 2
TMS
TCK
TDI
TDO
Device 3
TMS
TCK
TDI
TDO

JTAG

- All controllers in the chain must be in same state
  - TMS=1 for five clocks, switch to Test-Logic-Reset

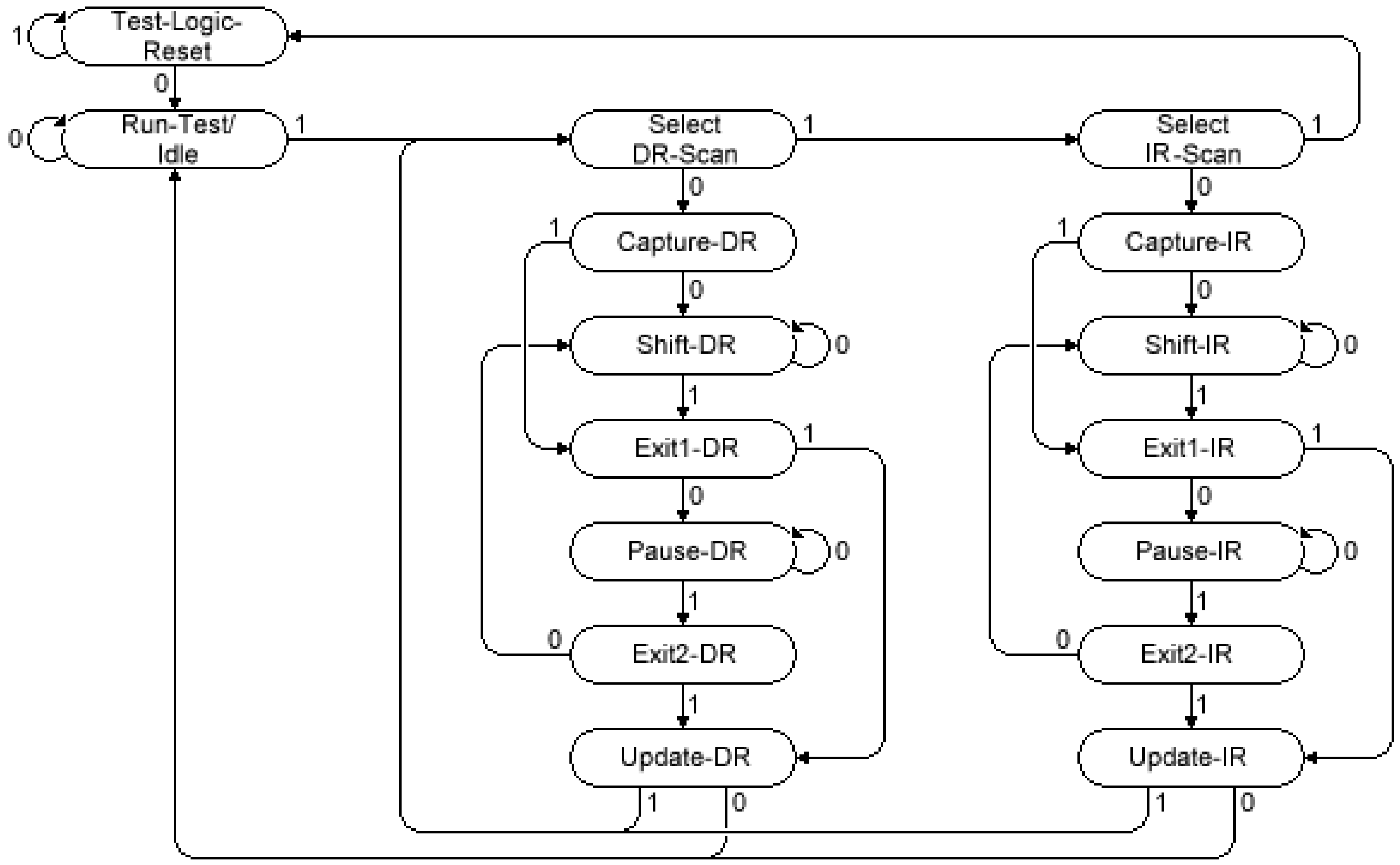
- In the shift-DR and shift-IR controller states:
  - TDO is updated on the falling edge of TCK by Target
  - Sampled on the rising edge of TCK by Host
- TMS and TDI are sampled on the rising edge of TCK by Target
- Updated on the falling edge of TCK by Host

- Test Access Port (TAP) Controller



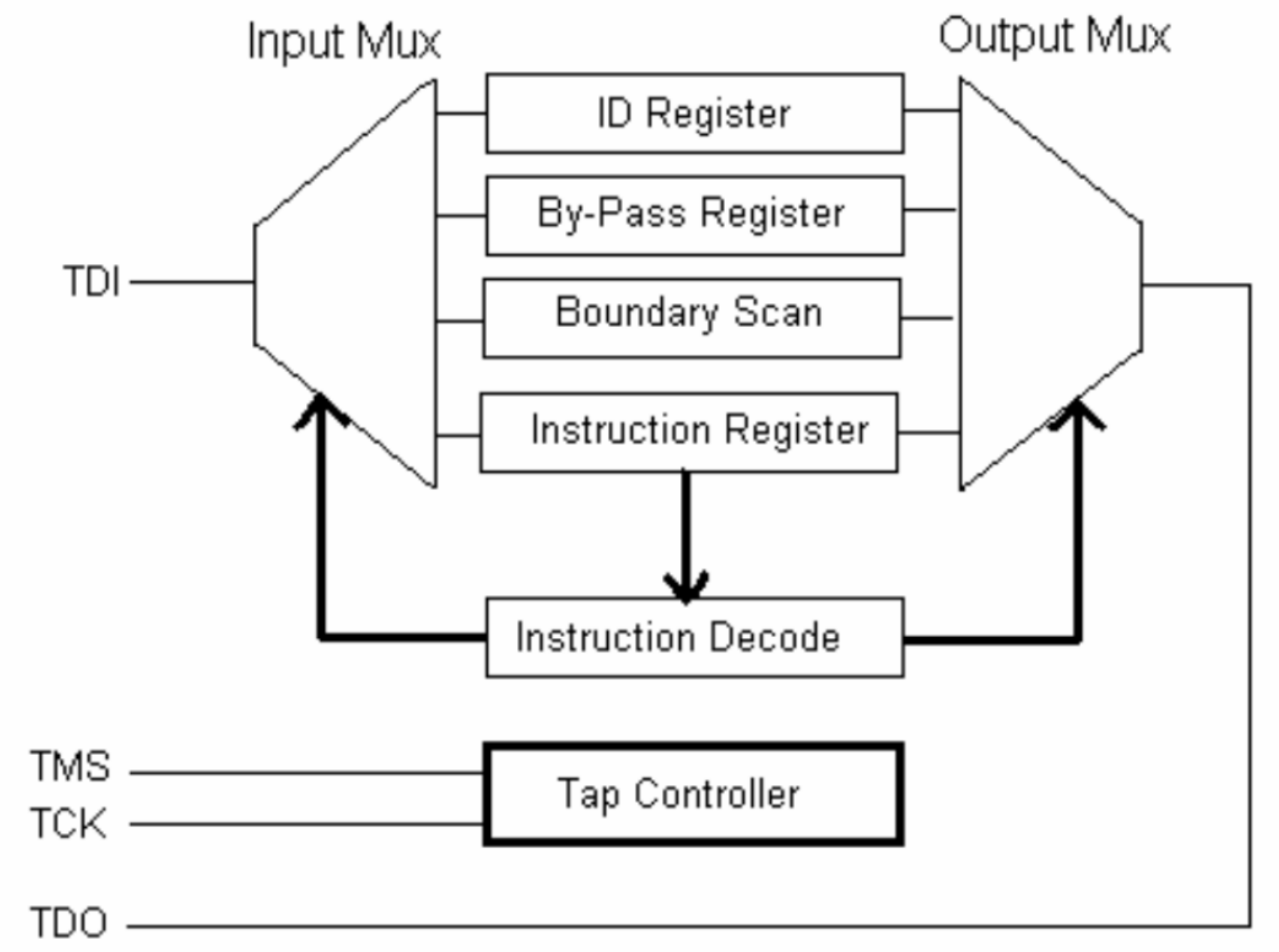
JTAG

- TMS sequences:
  - 11111: go to Test-Logic-Reset from any state
  - 0100: enter Shift-DR state, keep TMS=0 to stay there
  - From Shift-DR state, 11 to register the value shifted
  - 01100: enter Shift-IR state, keep TMS=0 to stay there
  - From Shift-IR state, 11



JTAG

Bypass register: 1 bit
Instruction register: 5 bits
Boundary scan: 14 bits
ID register: 32 bits



I^2C, CAN, I^2S

- CAN (Controller Area Network):
  - Automotive applications
  - Similar to I^2C

- I^2S:
  - Digital Audio
  - Three pins: clock SCK, word select WS, data SD
  - Word select selects between left and right channels

Comparison

	I^{2}C	SPI	JTAG
duplex	half duplex	full duplex	full duplex
performance	≤ 3.4 Mbps	no limit	no limit
driver	open drain	push-pull	push-pull
word size	8-bit	no restriction	no restriction
addressing	fixed	none	none
pins	2	4	4
acknowledgement	Yes	no	no