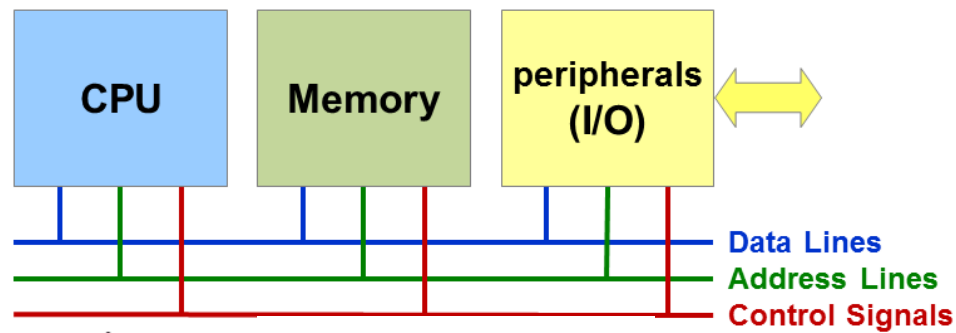


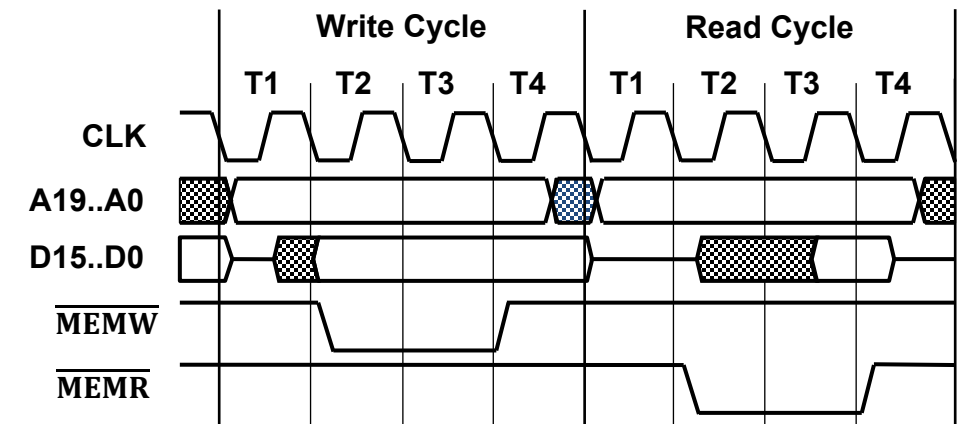
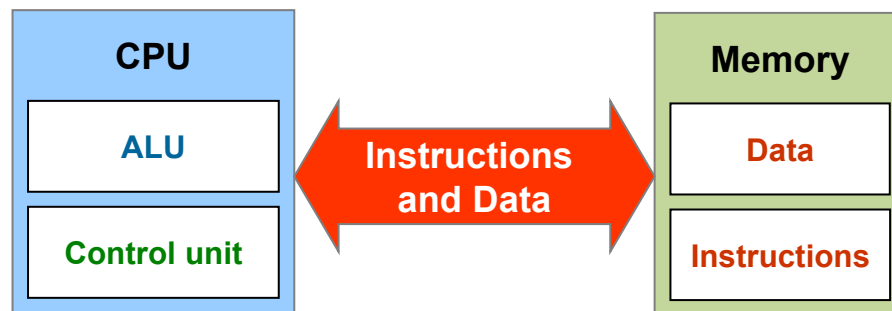
Microcontroller Basics

Computer Engineering 2

■ Connecting a CPU to the Outside World



von Neumann Architecture



A peripheral is a configurable hardware block of a microcontroller that accepts a specific task from the CPU, performs this task and reports back the status (e.g., task completion, error). Many peripherals are interfaces to the outside world. Examples include GPIO, UART, SPI, ADC ...

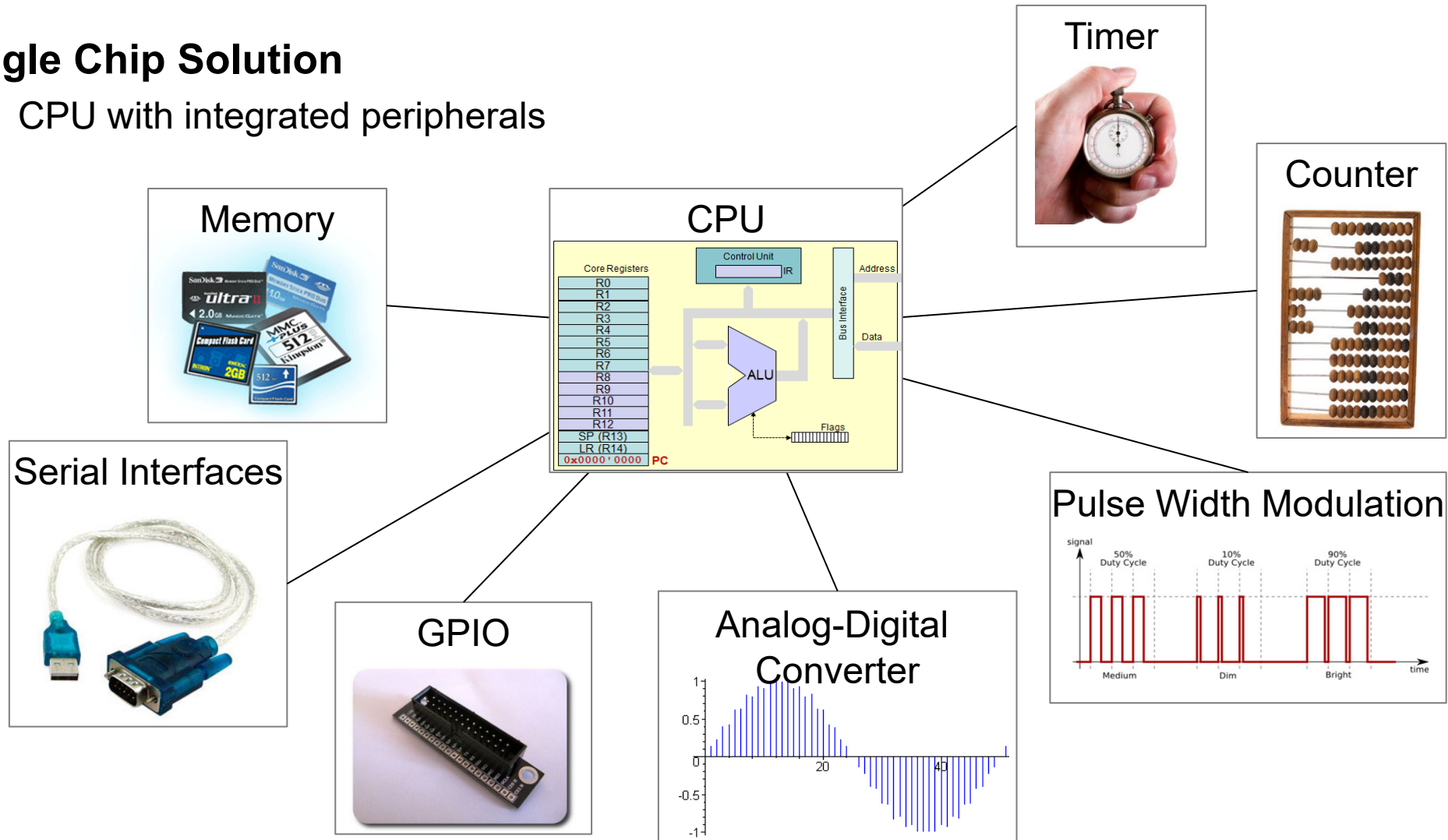
- **Microcontroller**
- **System Bus**
- **Digital Logic Basics**
- **Synchronous Bus**
- **Control and Status Registers**
- **Address Decoding**
- **Slow Slaves (Peripherals)**
- **Bus Hierarchies**
- **Accessing Control Registers in C**
- **Conclusions**

At the end of this lesson, you will be able

- to enumerate the signal groups of a system bus
- to distinguish between 'synchronous' and 'asynchronous' bus timing
- to know what the term 'tri-state' means
- to interpret simple bus timing diagrams
- to describe the concept of a bus and how data is transferred on a bus
- to describe the function and purpose of control and status registers
- to explain the terms 'full address decoding' and 'partial address decoding'
- to access control registers from C
- to explain the meaning of the qualifier 'volatile' in C
- to analyze address decoding logic and to derive the applicable addresses or address ranges
- to derive an address decoding logic for a given address (range)
- to explain the function of wait states

■ Single Chip Solution

- CPU with integrated peripherals



■ Embedded Systems

**LOW
COST**

USB stick. consumer,
power supplies

Real Time

anti-lock braking system,
process control

extreme environment

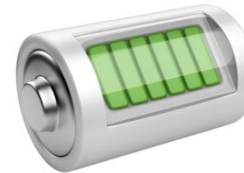


automotive, satellites



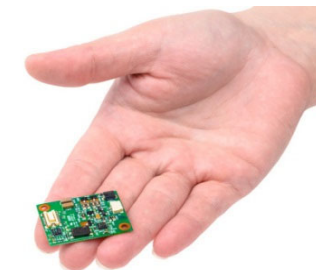
medical instruments

low power



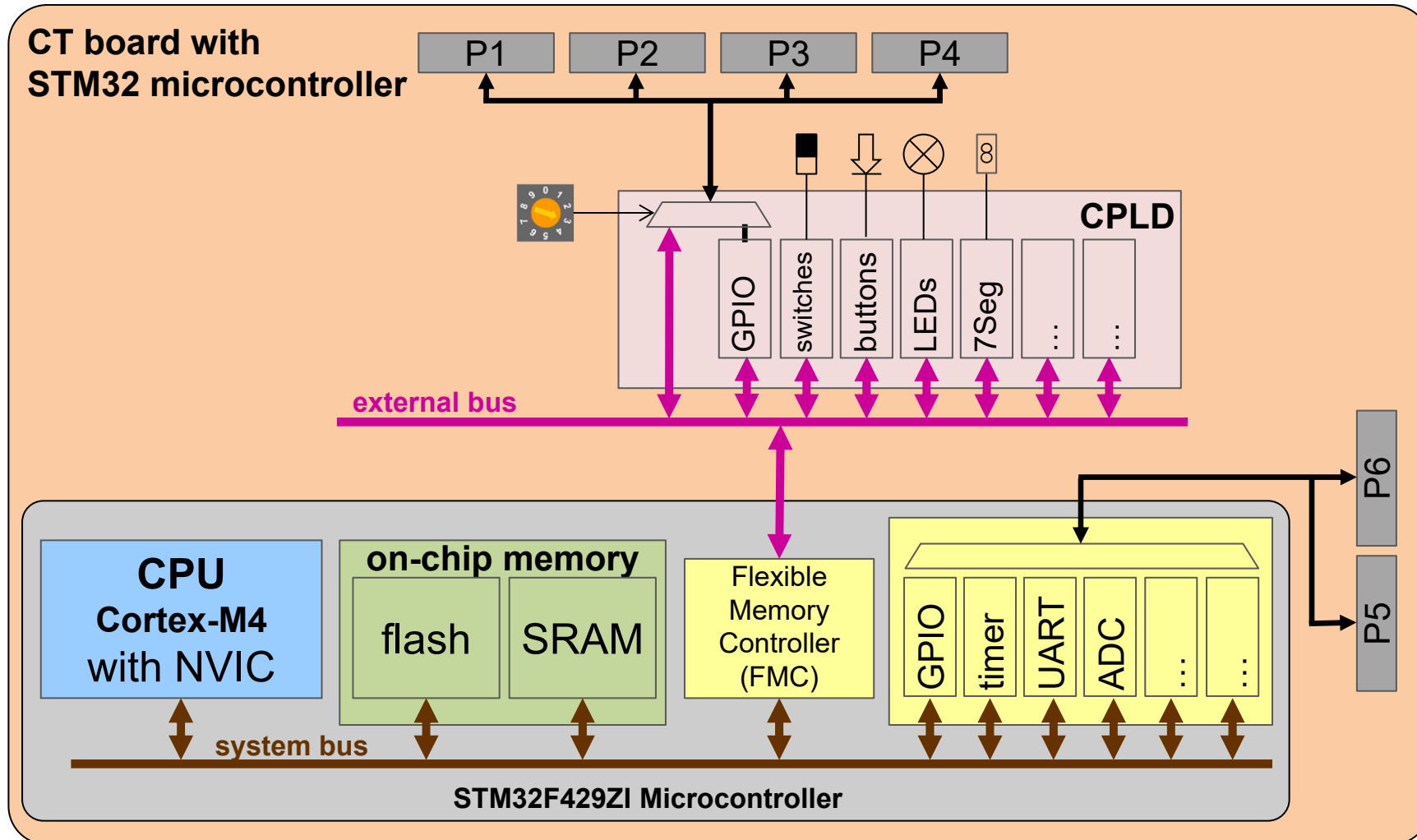
sensor networks,
autarkic systems

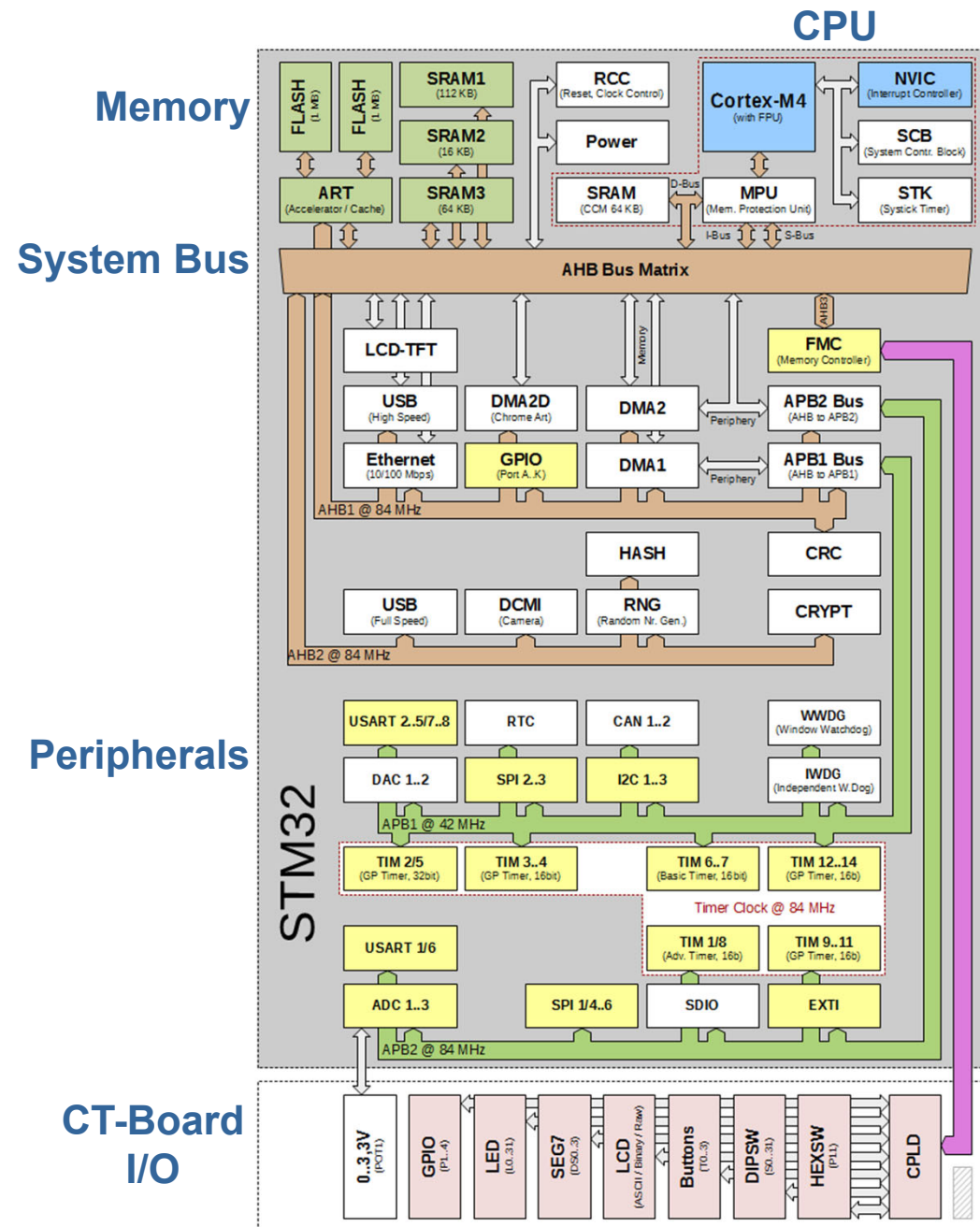
miniaturized



wearables

CT Board with STM32 Microcontroller



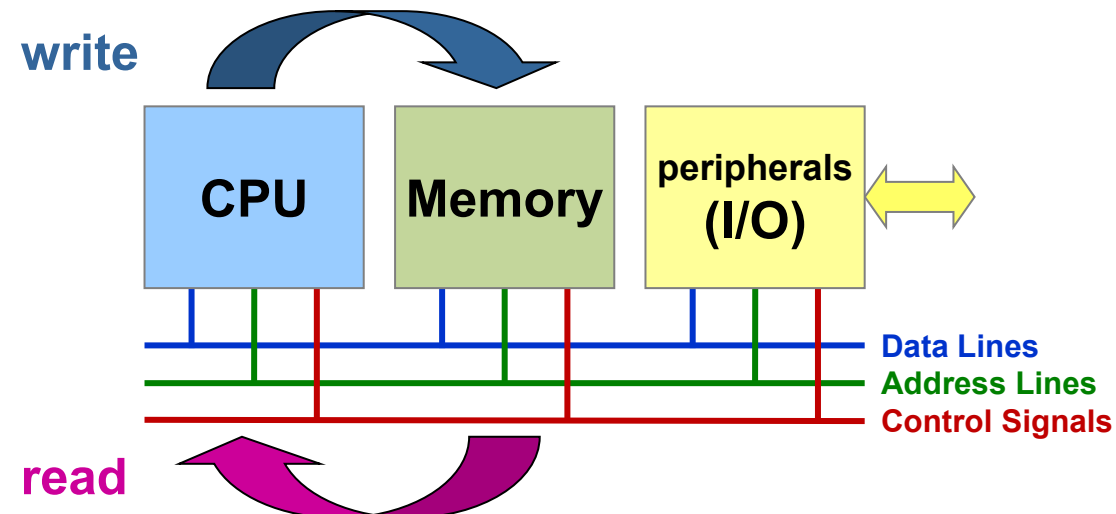


STM32 microcontroller

■ System Bus

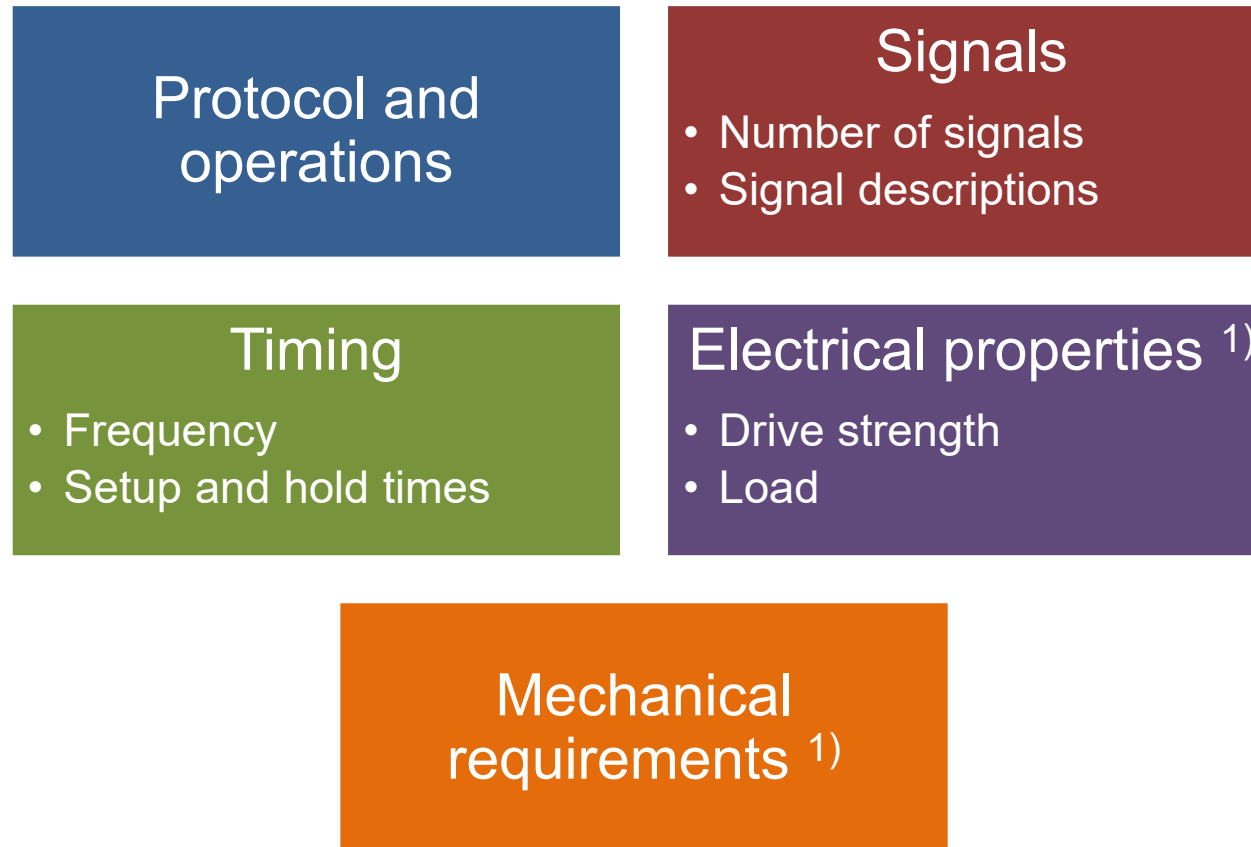
- Interconnects CPU with memory and peripherals
- CPU acts as master ¹⁾
 - Initiating and controlling all transfers
- Peripherals and memory act as slaves
 - Responding to requests

Etymology
Bus from Latin omnibus, i.e. "for all"



1) multi-master systems are not covered here

■ Bus Specification



¹⁾ not covered in this course

■ Signal Groups

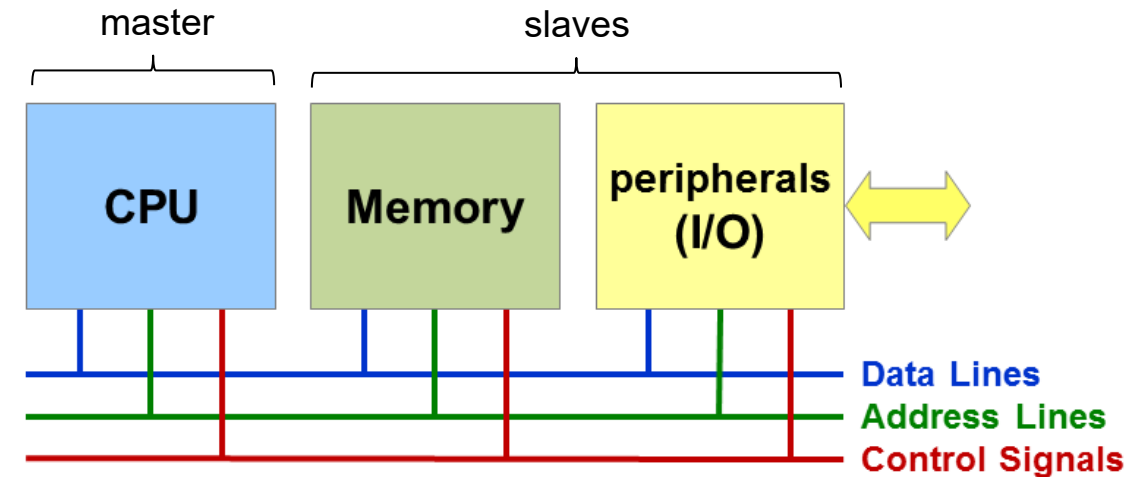
- Address lines
 - Unidirectional: From master to slave
 - Number of lines → size of address space
- Data lines
 - 8, 16, 32 or 64 parallel lines of data
 - bidirectional (read/write)
- Control signals
 - Control read/write direction
 - Provide timing information

Cortex-M

32 address lines

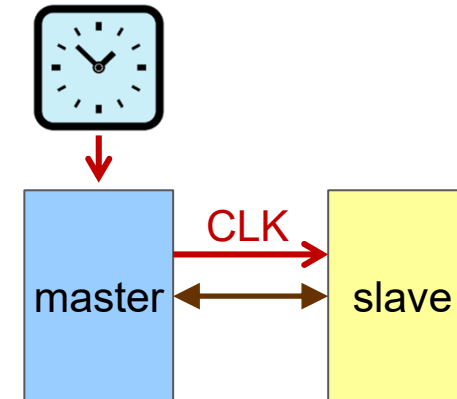
→ $2^{32} = 4$ Giga addresses

0x0000'0000 – 0xFFFF'FFFF

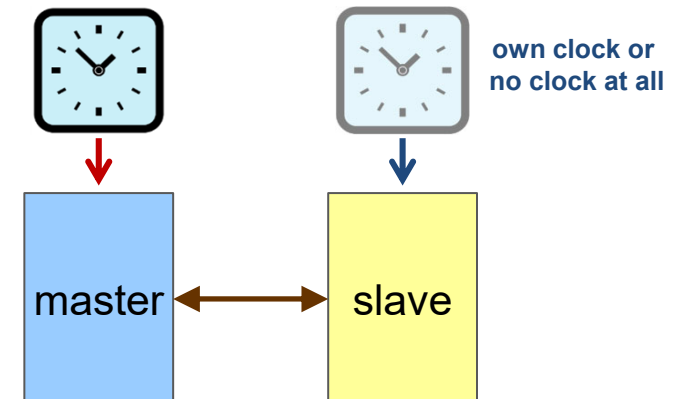


■ Bus Timing Options

- Synchronous
 - Master and slaves use a common clock¹⁾
 - Clock edges control bus transfer on both sides
 - Used by most on-chip busses
 - Off-chip: DDR and synchronous RAM



- Asynchronous
 - Slaves have no access to the clock of the master
 - Control signals carry timing information to allow synchronization
 - Widely used for low data-rate off-chip memories
→ parallel flash memories and asynchronous RAM



1) Often this is a dedicated clock signal from master to slave but the clock can also be encoded in a data signal

■ Multiple Devices Driving the Same Data Line

- What if one device drives a logic '1' (Vcc) and the other a logic '0' (Gnd)?
 - Electrical short circuit → bus contention (dt. "Streitigkeit")

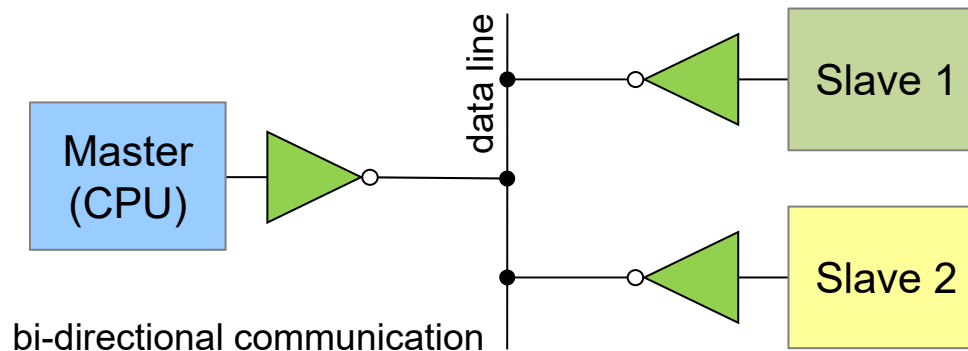


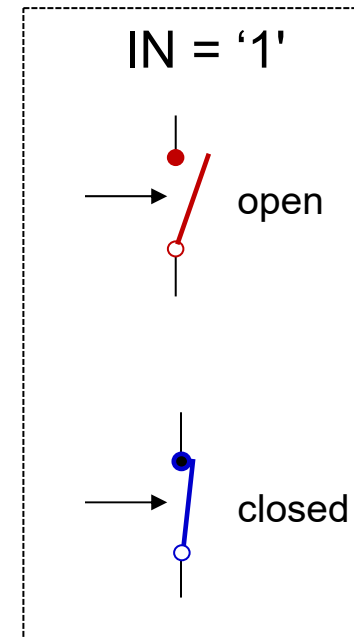
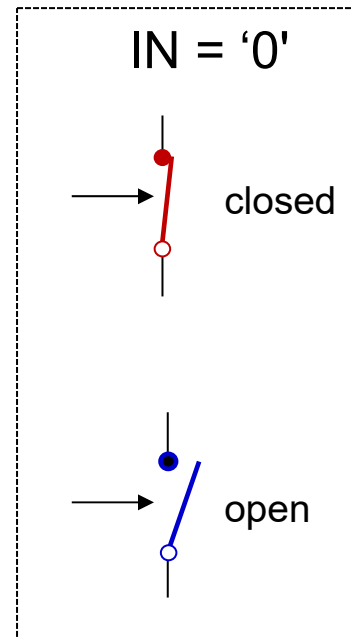
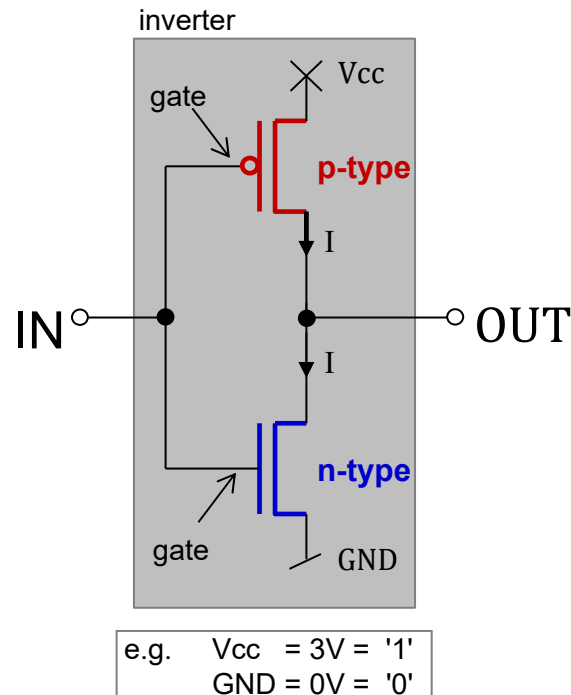
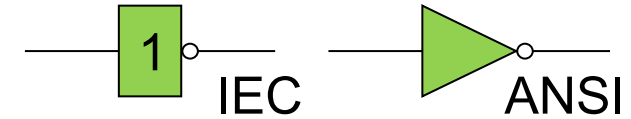
Figure only shows the output paths.
Input paths are omitted.

- CPU defines who drives the data bus at which moment in time
 - Write CPU drives bus all slave drivers disconnected
 - Read CPU driver disconnected single slave drives bus selected through values on address lines other slave drivers disconnected

But how can a driver be disconnected electrically?

■ CMOS¹⁾ Inverter

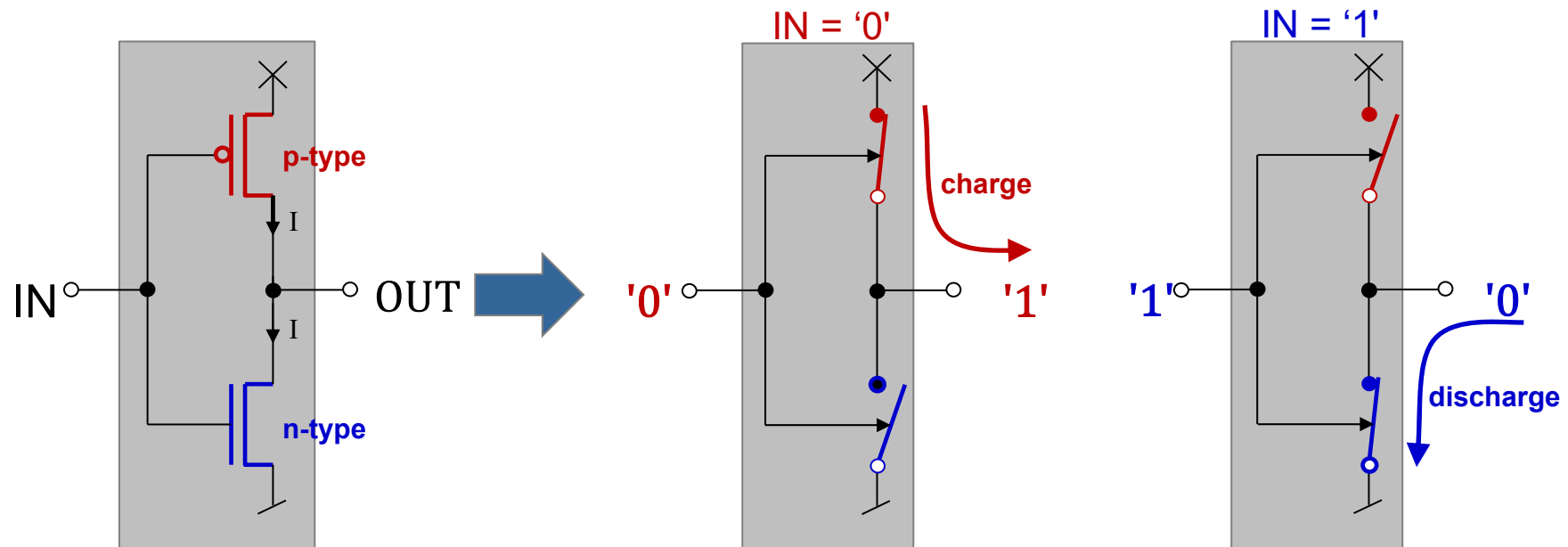
- Complementary switches (transistors)
 - **p-type** and **n-type** have opposite open-close behavior



1) CMOS: Complementary Metal-Oxide-Semiconductor

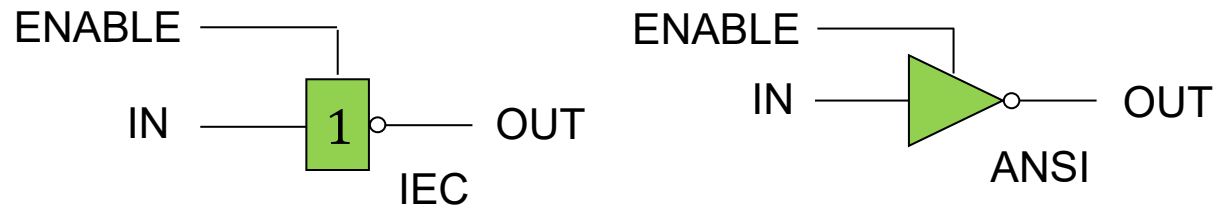
■ CMOS Inverter

- Complementary switches (transistors)



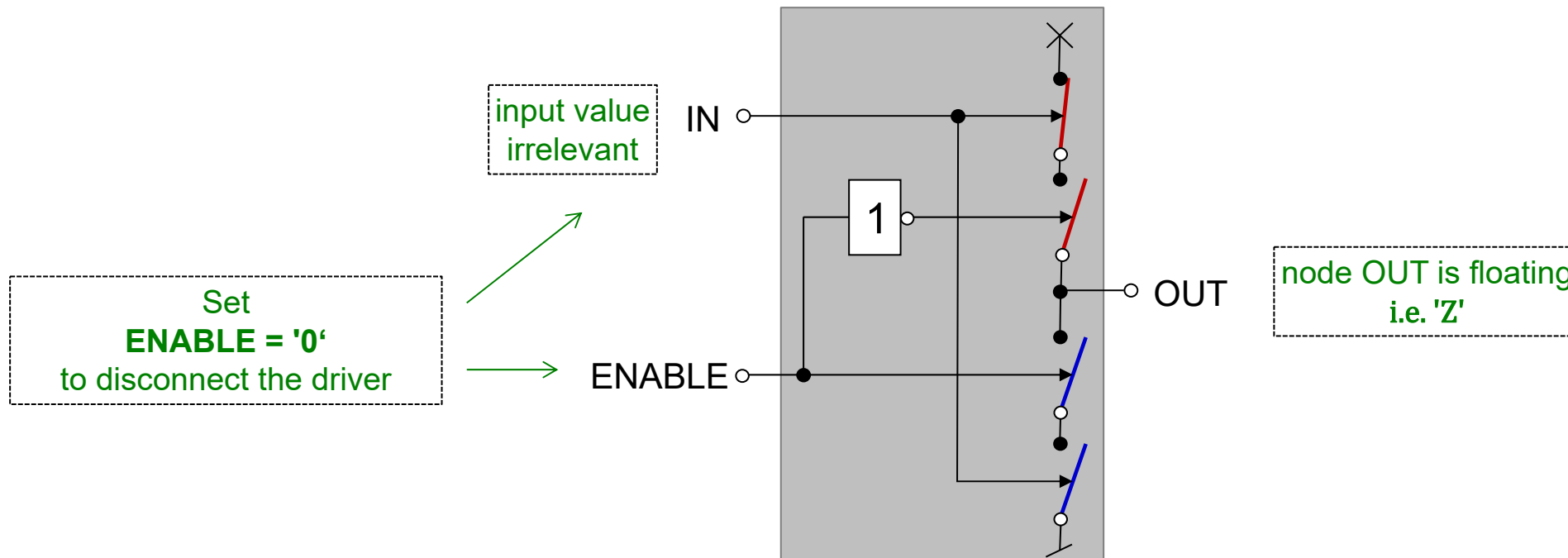
- A buffer is built by connecting two inverters in series

■ CMOS Tri-State Inverter

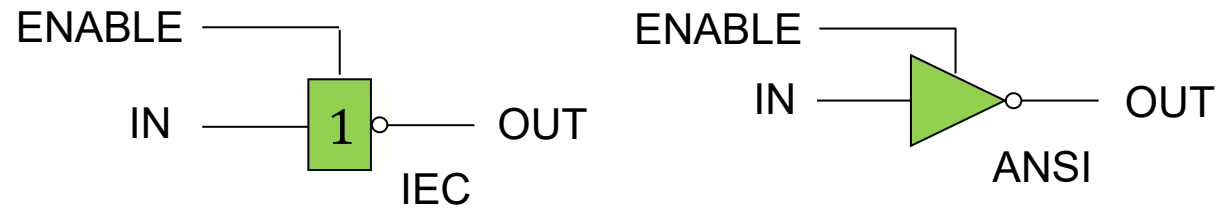


ENABLE	OUT
'1'	! IN
'0'	'Z'

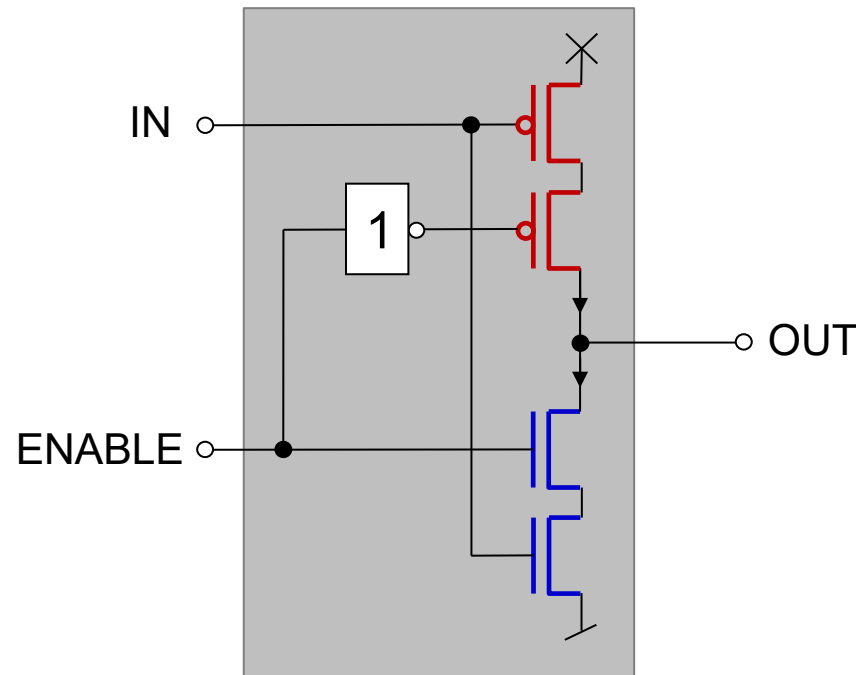
'Z' = high impedance



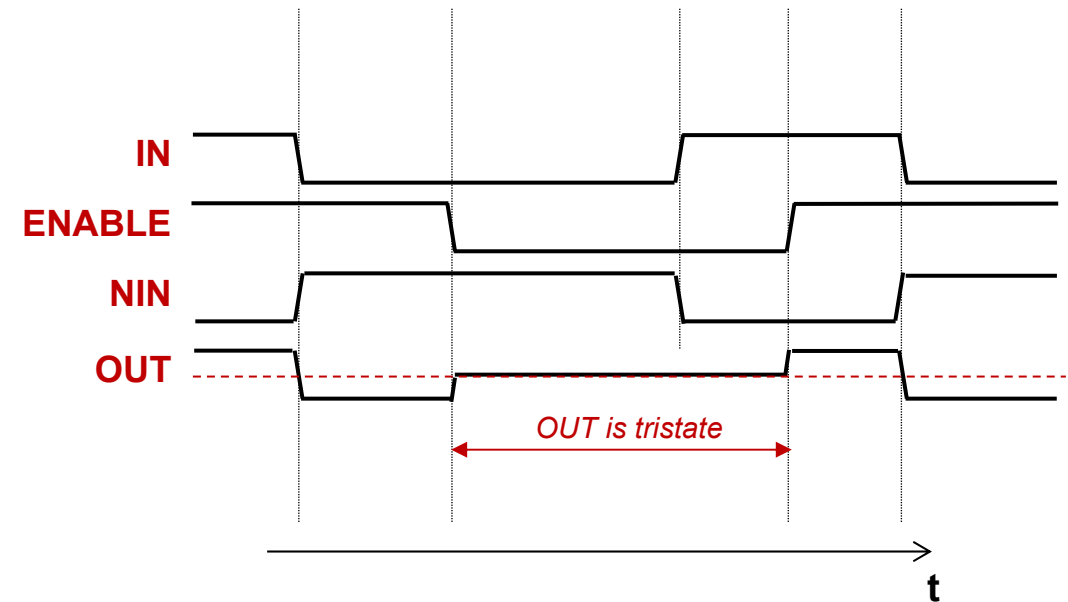
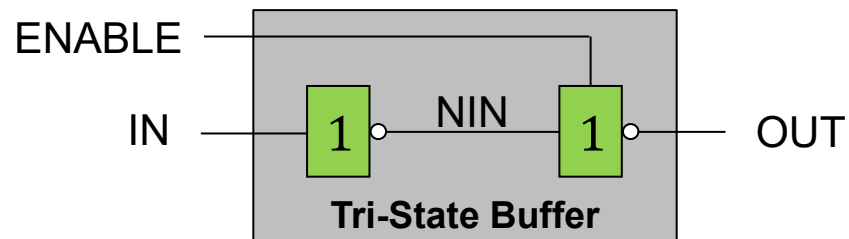
■ CMOS Tri-State Inverter - Implementation



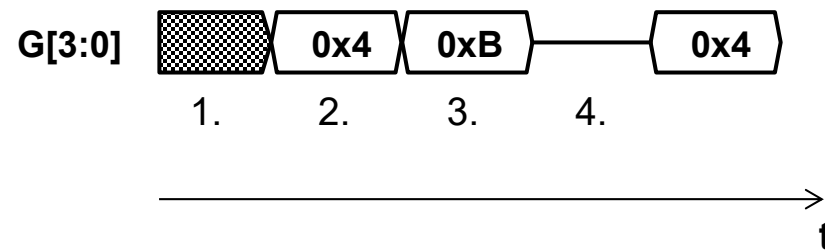
ENABLE	OUT
'1'	! IN
'0'	'Z'



■ CMOS Tri-State Buffer – Timing Diagram



■ Bus Timing Diagrams: Notations



Group G of 4 signals

1. unknown values
The values on each of the 4 signals are either '1' or '0', but unknown
2. The bus holds the value 0x4
i.e. $G[3] = '0'$, $G[2] = '1'$, $G[1] = '0'$, $G[0] = '0'$
3. The bus holds the value 0xB
i.e. $G[3] = '1'$, $G[2] = '0'$, $G[1] = '1'$, $G[0] = '1'$
4. Tri-state
All signals G[3:0] are tri-state (i.e. 'Z' or high-impedance)

■ Example Uses External Bus from ST Microelectronics

- Reason: Internal workings of the system bus are not disclosed by STM
- Signal names, bus protocol and timing based on external synchronous STM32F429xx mode instead
 - *For details see*
 - ▶ *Chapter 37, Flexible memory controller (FMC) in ST Reference Manual RM0090*
 - ▶ *Datasheet STM32F429xx*
 - Figure 60 Synchronous non-multiplexed NOR/PSRAM read timings*
 - Figure 61 Synchronous non-multiplexed PSRAM write timings*

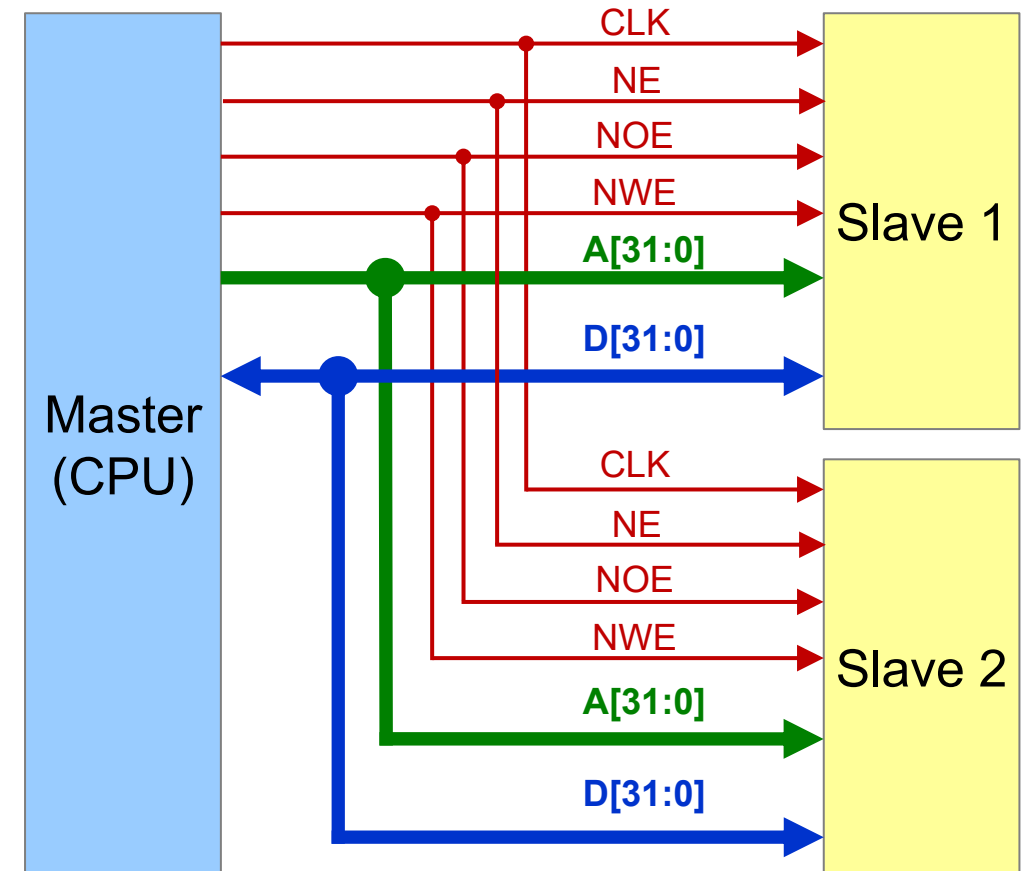
■ Naming Convention

- Letter 'N' prefix in signal name (Nxxx) means active-low signal
 - E.g. NOE means 'NOT OUTPUT ENABLE'

NOE = '0' → output enabled
NOE = '1' → output disabled

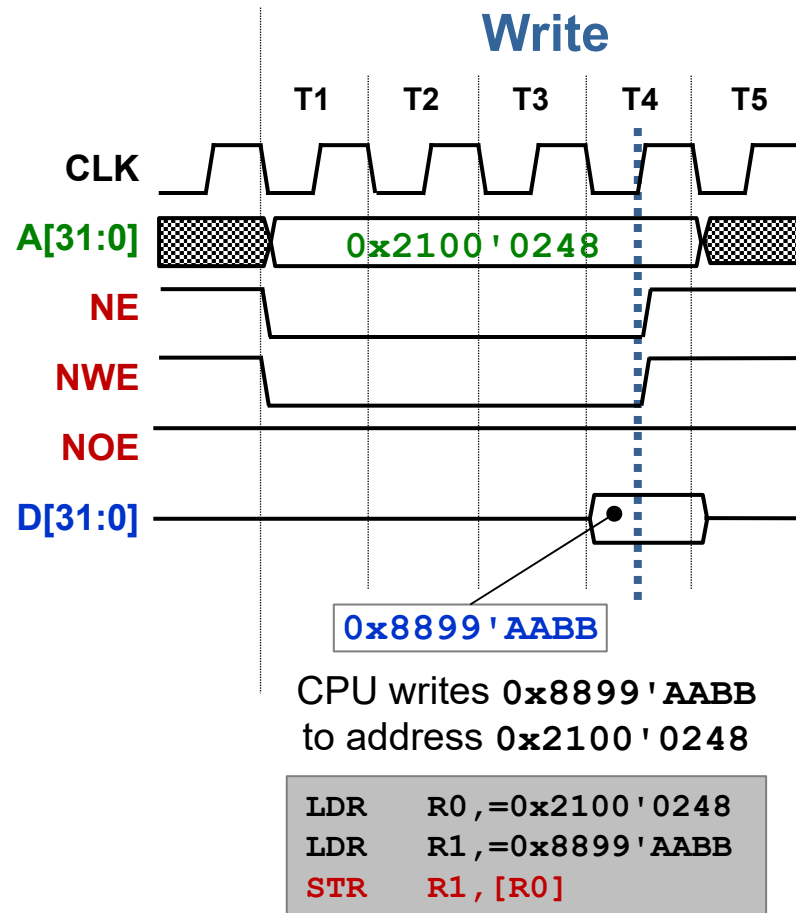
■ Block Diagram

- Address lines A[31:0]
- Data lines D[31:0]
- Control
 - CLK
 - NE **Not Enable**
indicates start and end of cycle, active-low
 - NWE **Not Write Enable**
active-low
 - NOE **Not Output Enable**
(read), active-low



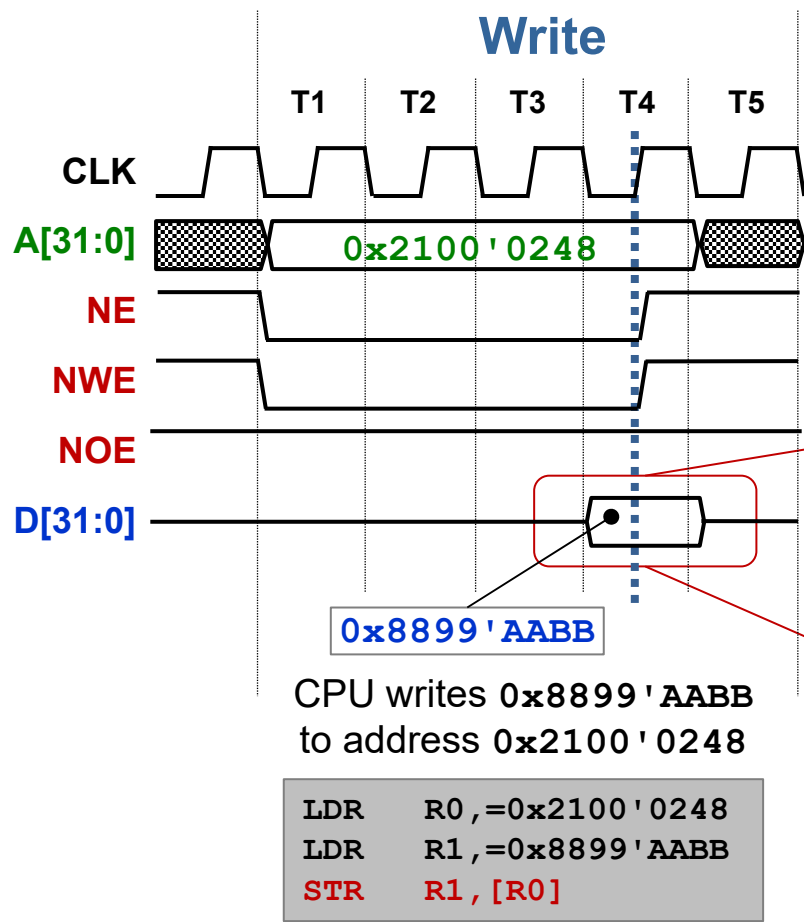
→ single line
→ multiple lines

■ Timing Diagram



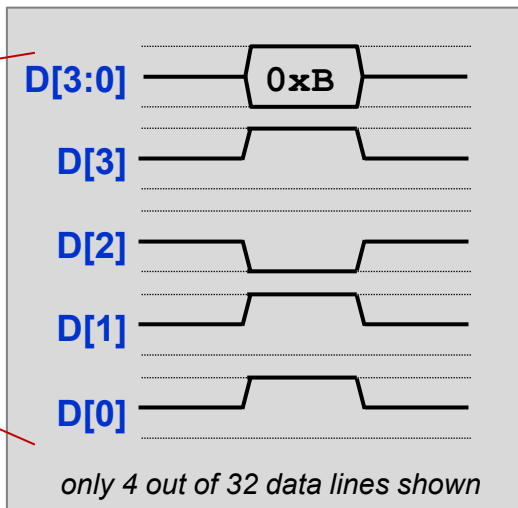
sample data in slave

■ Timing Diagram



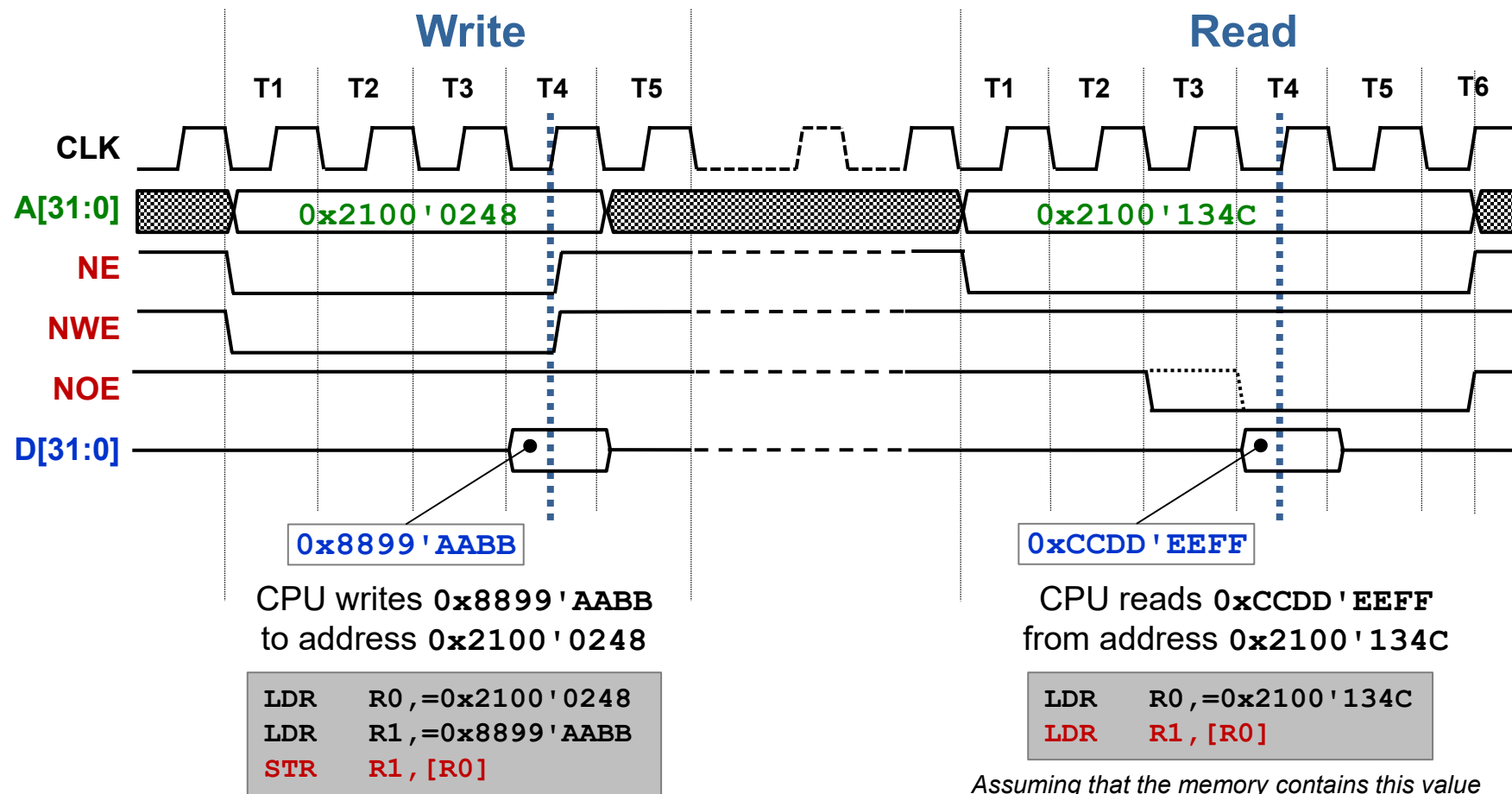
memory view after write access

0x2100'0248	0xBB	D[7:0]
0x2100'0249	0xAA	D[15:8]
0x2100'024A	0x99	D[23:16]
0x2100'024B	0x88	D[31:24]



■ Timing Diagram

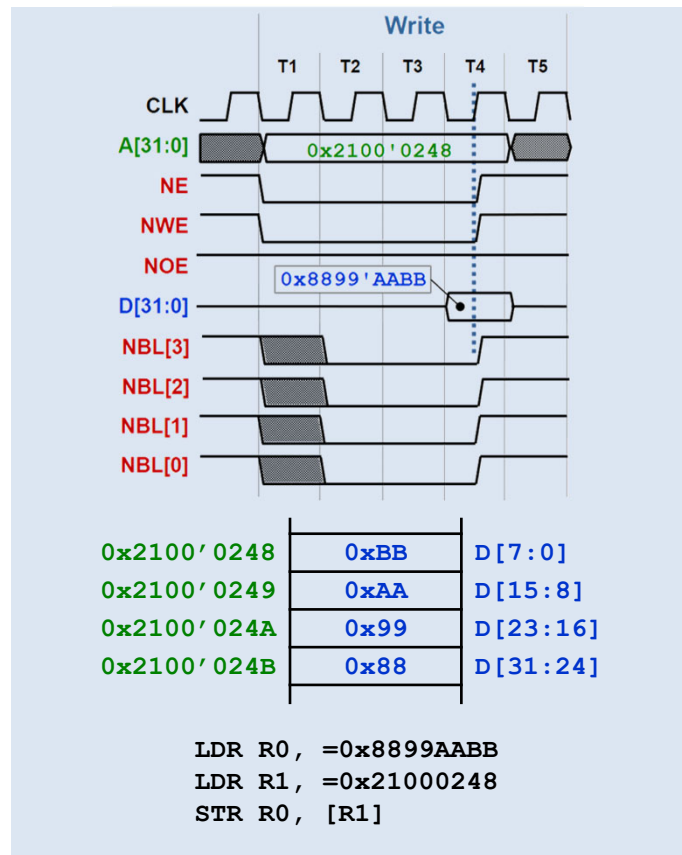
Note: The given timing is based on many design decisions by ST



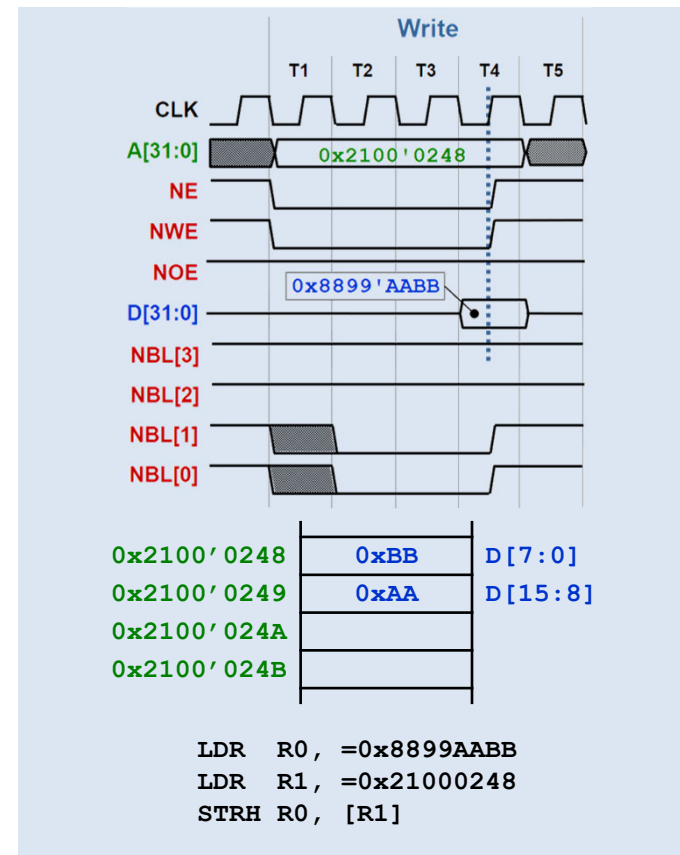
■ Bus Access Size

→ NBL (Not Byte Line) signals indicate valid bytes. See examples.

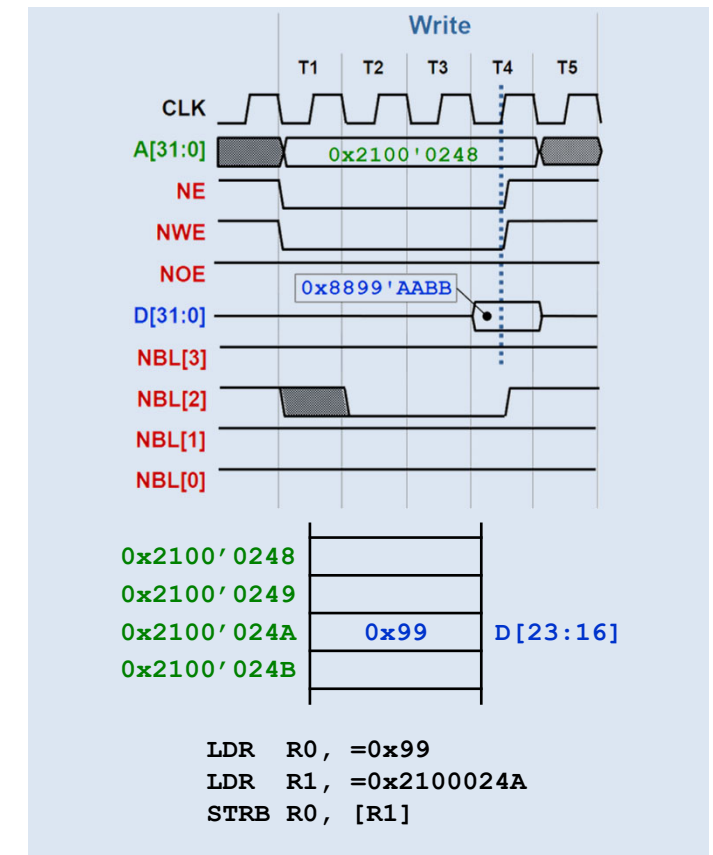
Word `STR R0, [R1]`



Half-word `STRH R0, [R1]`



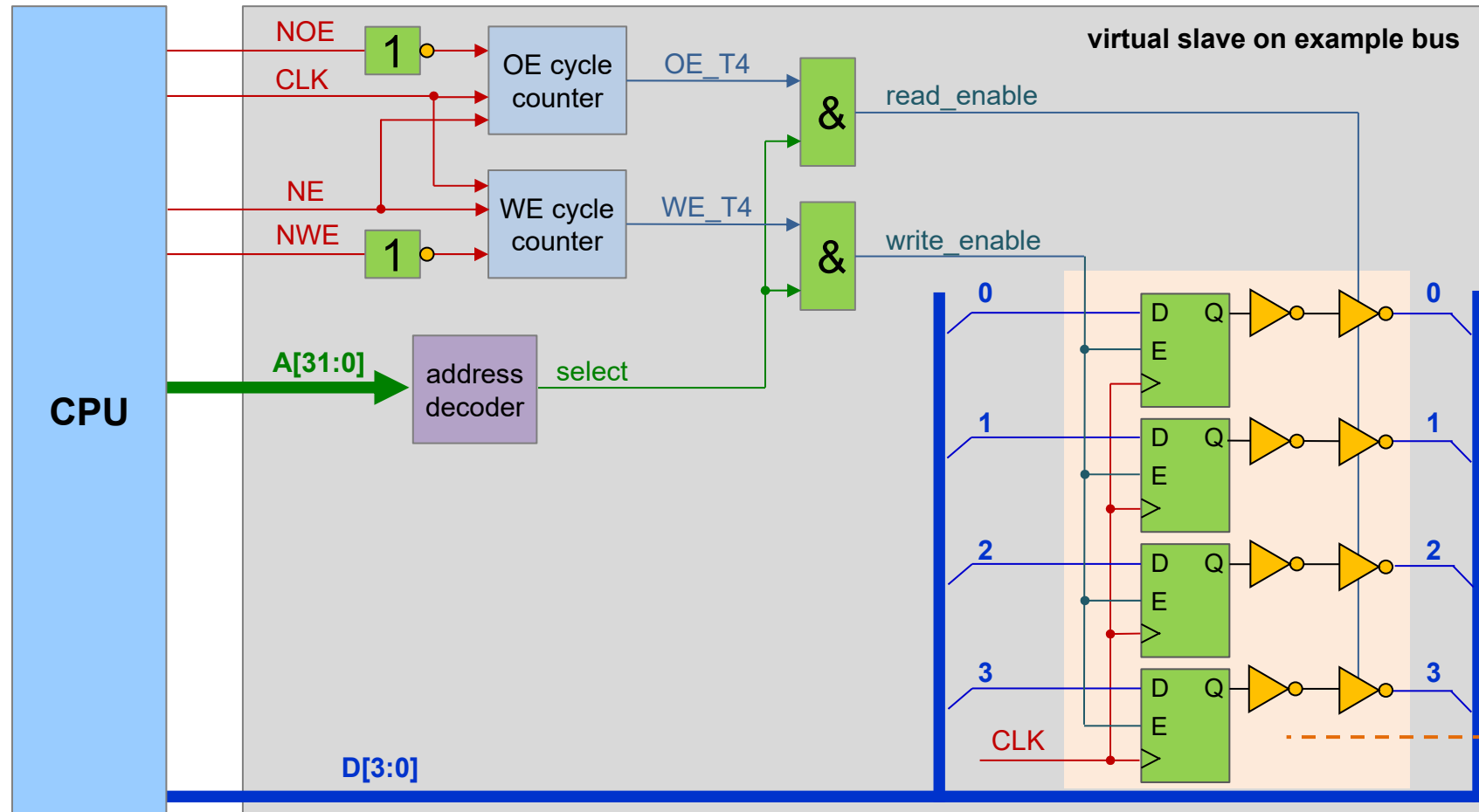
Byte `STRB R0, [R1]`



- Exact position of falling edge on NBL varies with chip version
- Value on unused data lines are unknown, figures show assumptions

■ Hardware Slave (Peripheral)

Figure only shows lower 4 data bits



Register
CPU can write into it and read from it

Control and Status Registers

■ Control Bits

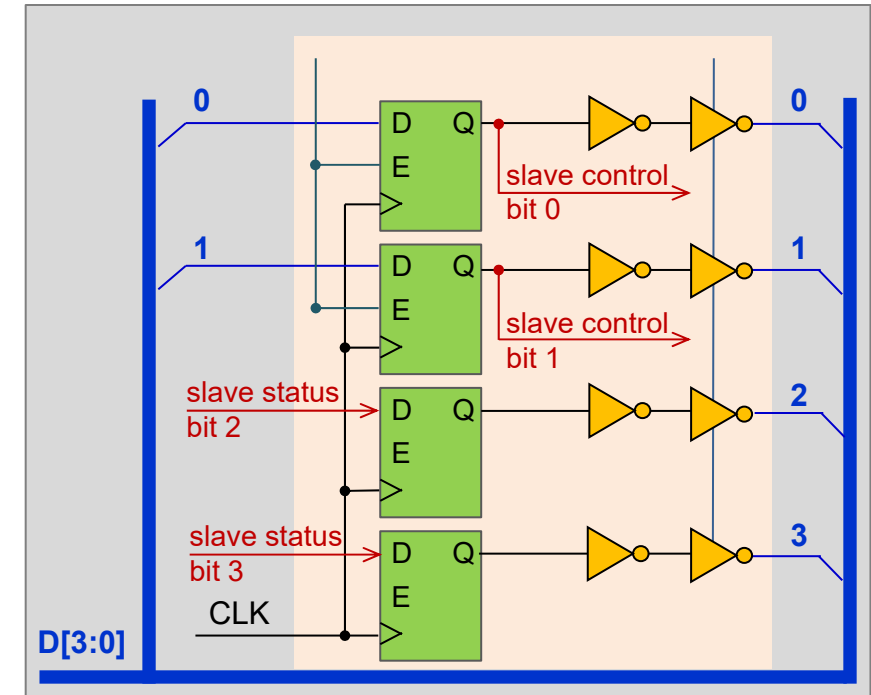
- Allow CPU to configure slave
- CPU (software) writes to register bit
- Slave hardware uses output of register bit
- Usually read/write

■ Status Bits

- Allow CPU to monitor a slave
- Slave writes status into register bit
- CPU (software) reads register bit
- Usually read-only

Same register may contain control and status bits

Control/status register on example bus



control bits 0 and 1 / status bits 2 and 3

Examples

Status	Switch, "Data byte received on serial interface"
Control	LED, "enable interface"

■ Control and Status Registers on CT Board

- Chip-internal and external registers

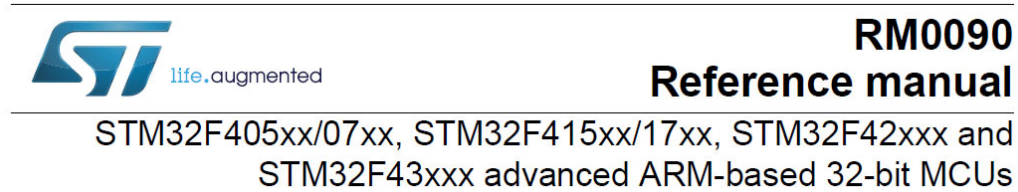
0x0000'0000	system (boot)	
0x1FFF'FFFF		
0x2000'0000	on-chip RAM	
0x3FFF'FFFF		
0x4000'0000	ST peripherals	1
0x5FFF'FFFF		
0x6000'0000	CT board I/O	2
0x7FFF'FFFF		
0x8000'0000	external memory	
0x9FFF'FFFF		
0xA000'0000		
0xBFFF'FFFF		
0xC000'0000		
0xDFFF'FFFF		
0xE000'0000		
0xFFFF'FFFF	Cortex-M peripherals	3

1 ST peripherals
e.g. Timers, ADC, UART, SPI, ...

2 CT board I/O
LEDs, dip switches, LCD, ...

3 ARM Cortex-M
NVIC, ...

■ ST Registers



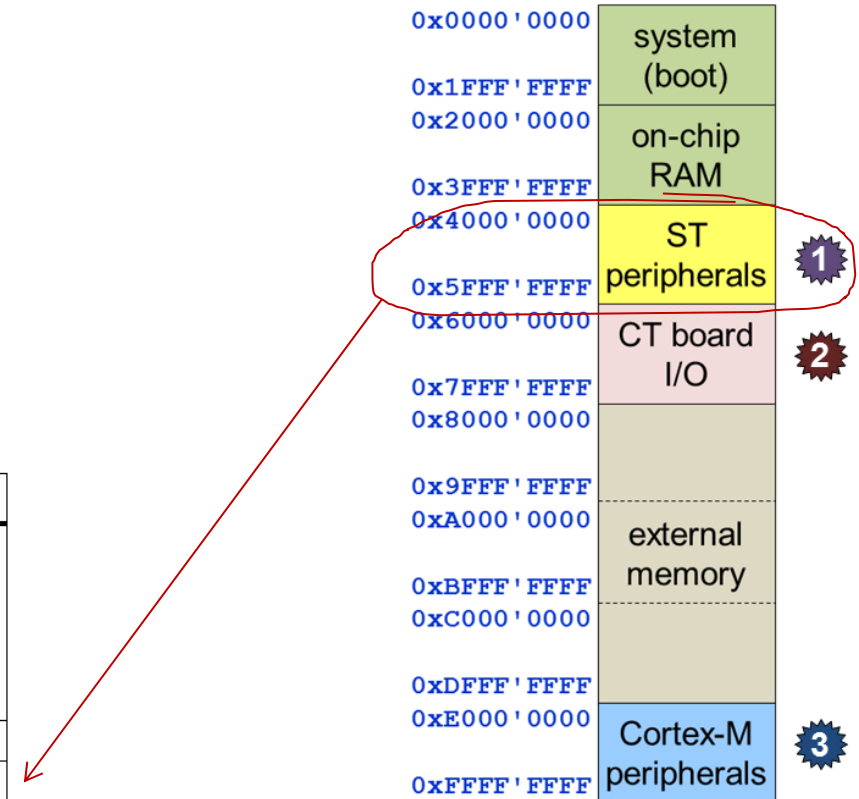
2.3 Memory map

See the datasheet corresponding to your device for a comprehensive diagram of the memory map. [Table 2](#) gives the boundary addresses of the peripherals available in all STM32F4xx devices.

Table 2. STM32F4xx register boundary addresses

Boundary address	Peripheral	Bus	Register map
0xA000 0000 - 0xA000 0FFF	FSMC control register (STM32F405xx/07xx and STM32F415xx/17xx)/ FMC control register (STM32F42xxx and STM32F43xxx)	AHB3	Section 36.6.9: FSMC register map on page 1573 Section 37.8: FMC register map on page 1653
0x5006 0800 - 0x5006 0BFF	RNG	AHB2	Section 24.4.4: RNG register map on page 752
0x5006 0400 - 0x5006 07FF	HASH		Section 25.4.9: HASH register map on page 776
0x5006 0000 - 0x5006 03FF	CRYP		Section 23.6.13: CRYP register map on page 745
0x5005 0000 - 0x5005 03FF	DCMI		Section 15.8.12: DCMI register map on page 473
0x5000 0000 - 0x5003 FFFF	USB OTG FS		Section 34.16.6: OTG_FS register map on page 1303

see chapter 2.3 of reference manual, page 64ff



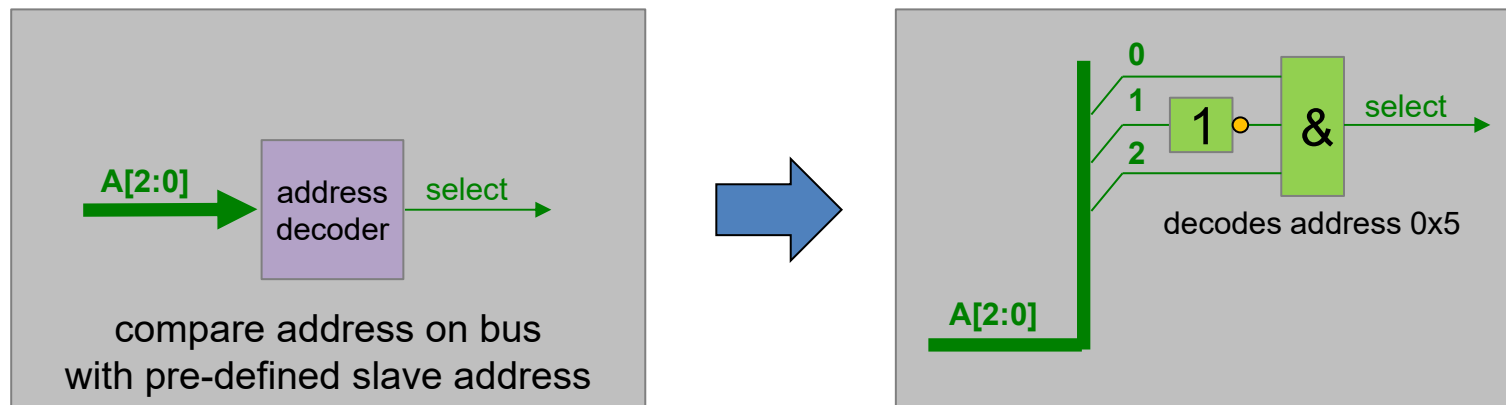
only part of table shown

■ How Does a Slave Know, That It Is the Target of an Access?

- Answer: Address decoding

■ Address Decoding

- Interpretation of address line values
 - See whether bus access targets a particular address or address range
- Example with 3 address lines
 - Select equal '1' indicates that the CPU wants to access address 0x5



Full Address Decoding

- All address lines are decoded (checked)
- A control register can be accessed at exactly one location
- 1 : 1 mapping
 - A unique address maps to a single hardware register (physical memory location)

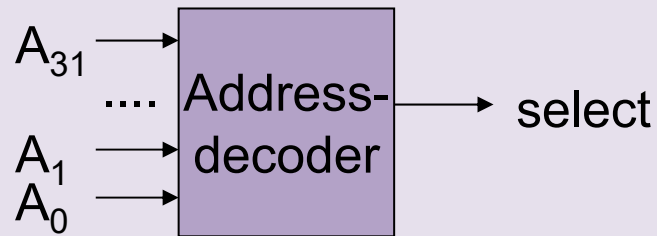
Partial Address Decoding

- Only a sub-set of the address lines is decoded
- Detects an address range or a set of addresses
- n : 1 mapping
 - n unique addresses map to the same hardware register (physical memory location)
- Motivation
 - Simpler decoding
 - Aliasing: Map a hardware register to several addresses

■ Example for 32-bit address space of Cortex-M

Full Address Decoding

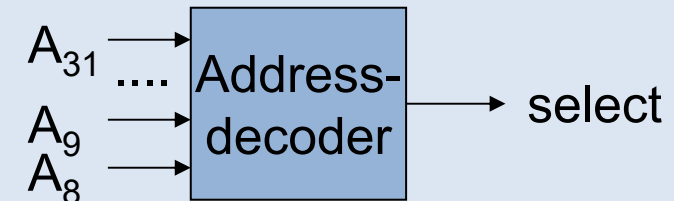
- all addresses from A31 to A0



- select is active for exactly one address
- E.g. at **`0x4000'8234`**

Partial Address Decoding

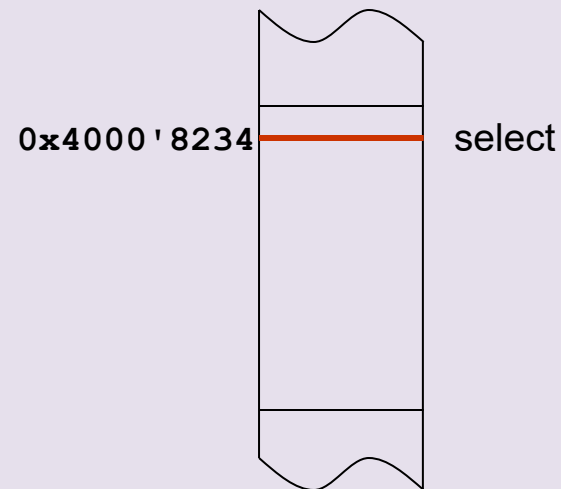
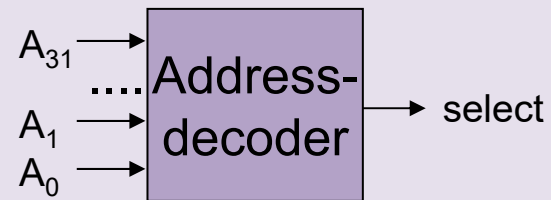
- only addresses from A31 to A8



- select is active for any address within a given range (e.g. ignoring some lower address lines)
- E.g. from **`0x4000'8200`** to **`0x4000'82FF`**
→ **`0x4000'82xx`**

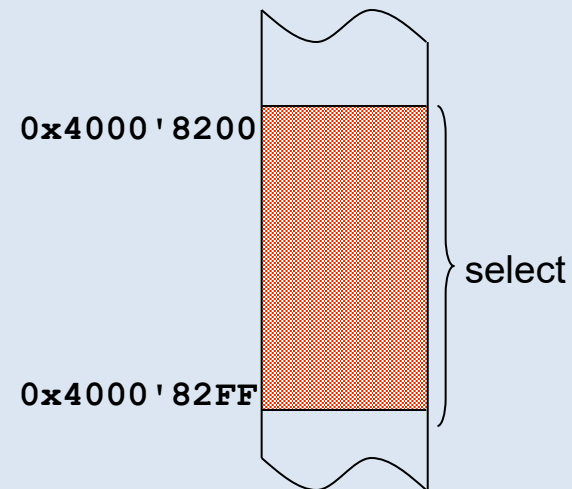
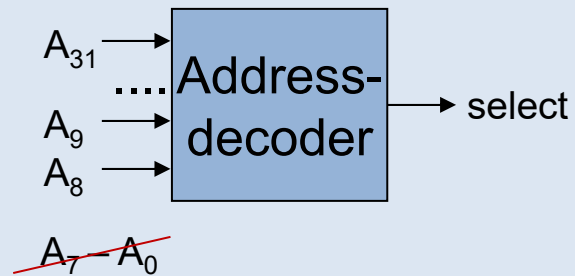
Full Address Decoding

all addresses

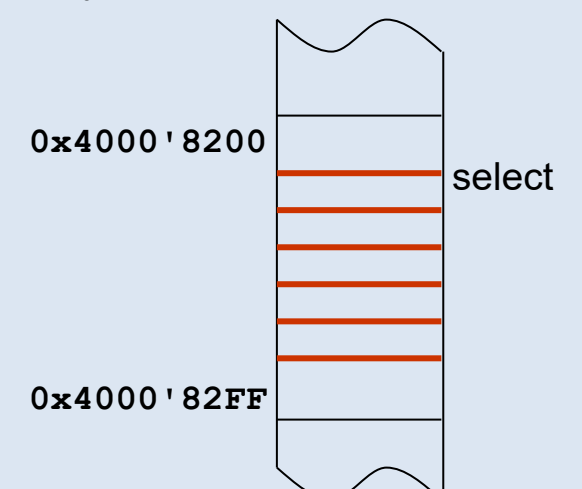
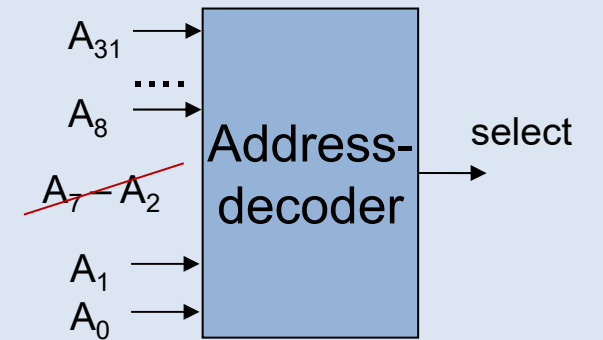


Partial Address Decoding

e.g. only high addresses



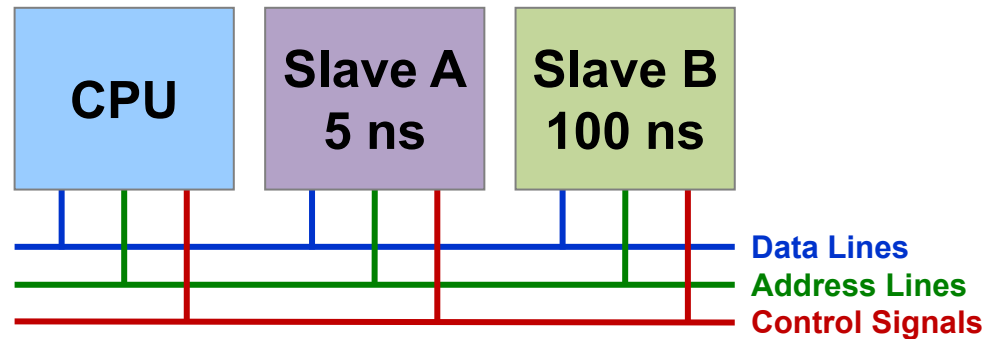
e.g. high and low addresses



address values are examples

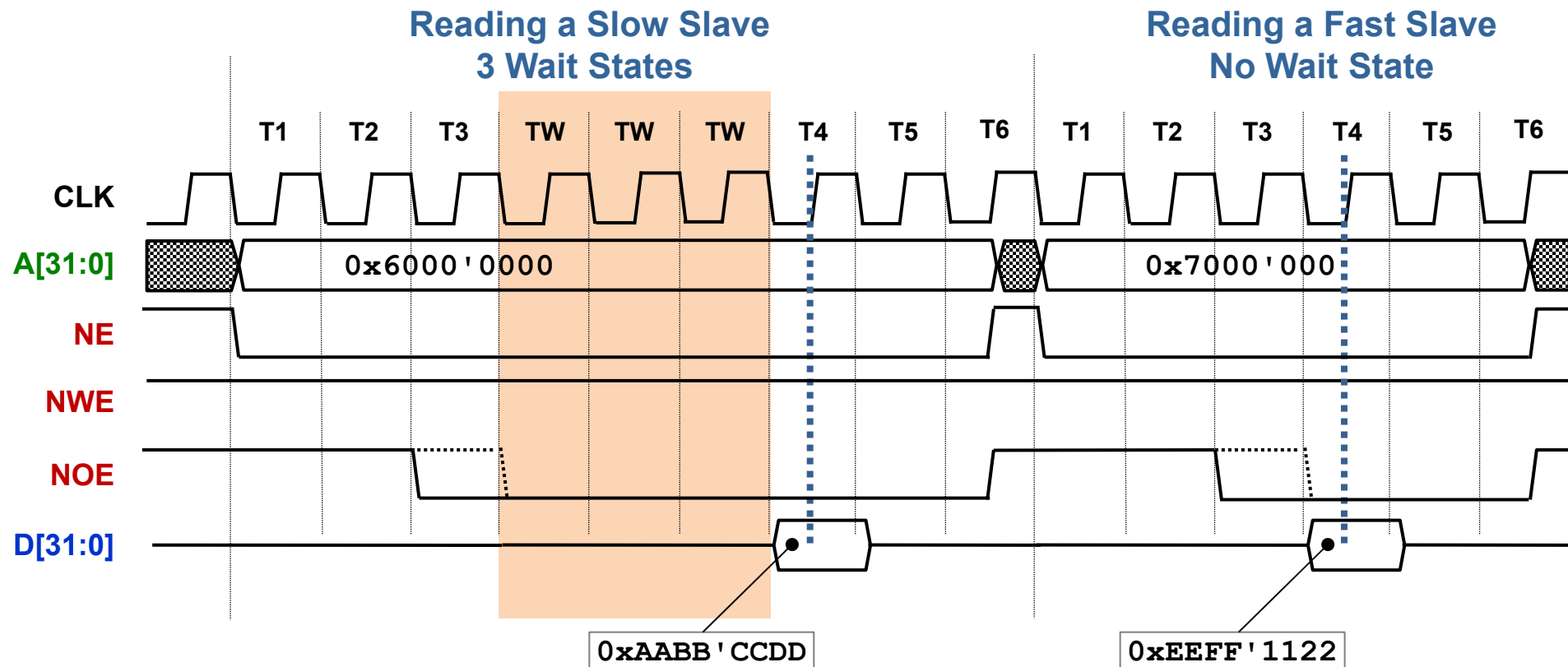
■ Problem: Individual Slave Access Times

- If slowest slave defines bus cycle time
 - Reduced bus performance
- How can we get an individual bus cycle time for each slave?



■ Introduce Individual Wait States for Slow Slaves

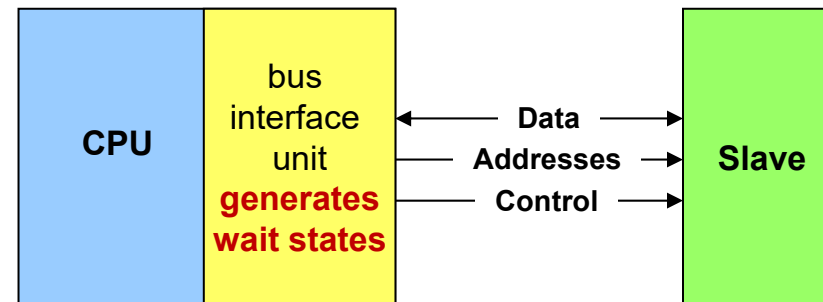
- Wait states are inserted depending on the address of an access



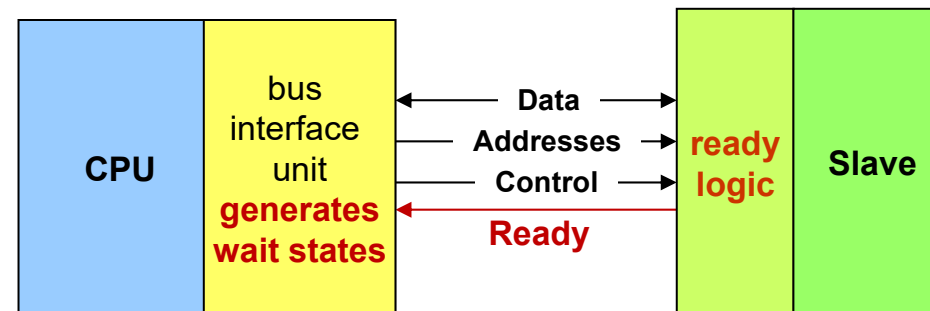
Slow Slaves (Peripherals)

■ Two Possibilities

1. Individual wait states can be programmed at a bus interface unit
 - Depending on the address of the bus cycle



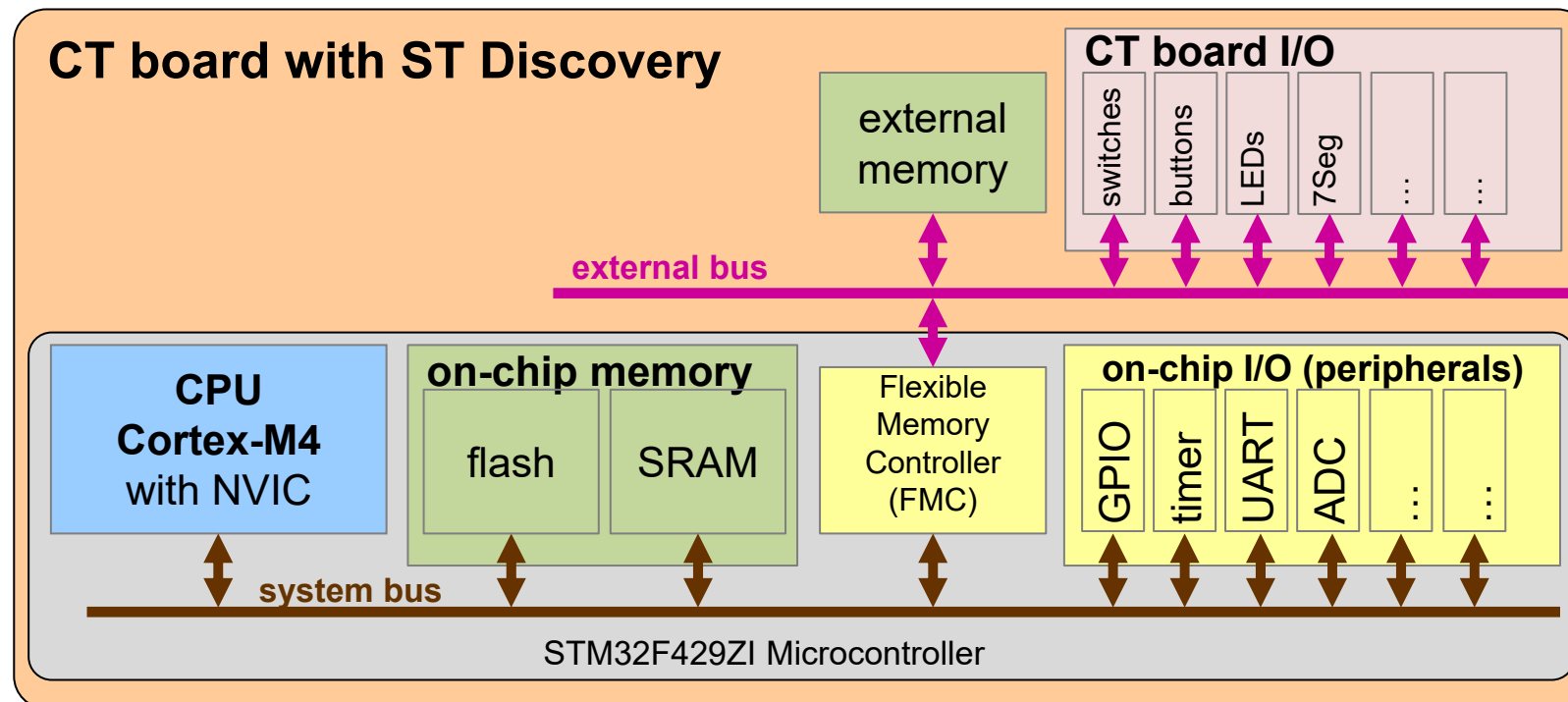
2. Slave tells bus interface unit when it is ready
 - Well suited for slaves with long or variable access times



Ready signal indicates when the slave is ready to provide the data

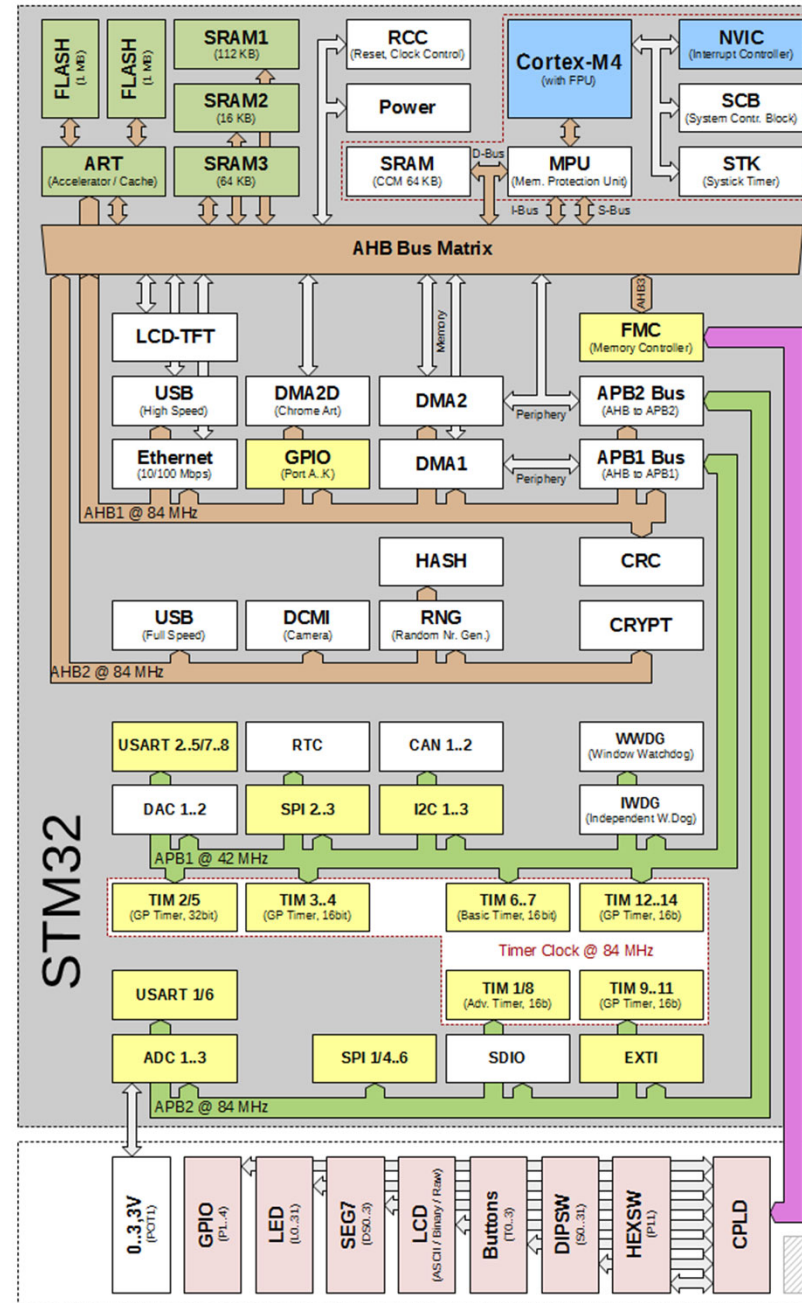
■ Simplified Model STM32F429ZI

- On-chip system bus 32 data lines, 32 address lines and control signals
- Off-chip external bus 16 data lines, 26 address lines and control signals



Bus Hierarchies

- Real-world Systems Are Partitioned into Multiple Busses



Accessing Control Registers in C

■ **Hardware View**

- Signals
- Timing
- Address decoding
- Wait states
- Control and status registers

■ **Software View**

- Accessing control and status registers in C

■ Situation

- Compiler may remove statements that have no effect from the compiler's point of view

```
uint32_t ui;  
  
void ex_func(void)  
{  
    ui = 0xAAAAAAA;  
    ui = 0BBBBBBB;  
    while (ui == 0){  
        ...  
    }  
}
```

Optimizing compiler will
remove these statements as
they seem to have no effect

- Accesses to control registers (read/write) are memory accesses
- If writing has a side effect on the hardware and/or if reading may result in a different value than was set before in the code, the program will not behave as intended.

■ Solution

- Use qualifier volatile in variable declaration

```
volatile uint32_t ui;  
  
void ex_func(void)  
{  
    ui = 0xAAAAAA;  
    ui = 0BBBBBBB;  
    while (ui == 0){  
        ...  
    }  
}
```

statements will not be removed by compiler

- Tell compiler that variable may change outside the control of the compiler
 - E.g. by hardware or by an interrupt handler
- The compiler cannot make any assumption on the value
 - Needs to execute all read/write accesses as programmed
 - Prevents compiler optimizations

Accessing Control Registers in C

■ Access through Pointers

- E.g. writing to and reading from CT Board I/O

```
// a pointer called p_reg pointing to
// a volatile uint32_t
volatile uint32_t *p_reg;
```

cast 'unsigned integer' to
'pointer to volatile uint32_t'

```
// set LEDs
```

```
p_reg = (volatile uint32_t *) (0x60000100);
*p_reg = 0xAA55AA55;
```

write 0xAA55'AA55 to LEDs


```
// wait for dip_switches to be non-zero
```

```
p_reg = (volatile uint32_t *) (0x60000200);
while ( *p_reg == 0 ) {
}
```

read dip-switches

0x0000'0000	system (boot)
0x1FFF'FFFF	
0x2000'0000	on-chip RAM
0x3FFF'FFFF	
0x4000'0000	ST peripherals
0x5FFF'FFFF	
0x6000'0000	CT board I/O
0x7FFF'FFFF	
0x8000'0000	
0x9FFF'FFFF	
0xA000'0000	external memory
0xBFFF'FFFF	
0xC000'0000	
0xDFFF'FFFF	
0xE000'0000	Cortex-M peripherals
0xFFFF'FFFF	

■ Using Preprocessor Macros → #define



```
#define LED31_0_REG    (*(volatile uint32_t *) (0x60000100))

#define BUTTON_REG     (*(volatile uint32_t *) (0x60000210))

// Write LED register to 0xBBCC'DDEE
LED31_0_REG = 0xBBCCDDEE;

// Read button register to aux_var
aux_var = BUTTON_REG;
```

- **Microcontrollers → Embedded Systems**
 - Low cost, real time, low power, extreme environments
- **System Bus**
 - Address, data and control lines
 - Synchronous or asynchronous
 - CPU (master) reads from or writes to slave
 - Timing sequences
 - Wait states
- **Address Decoding**
 - Who is the CPU talking to?
 - Full vs. partial address decoding
- **Accessing Control Registers in C**
 - Qualifier volatile
 - Use of pointers for memory accesses