EE 186: Introduction to Embedded Systems

Lecture 3

Zerina Kapetanovic

Optional Readings

- Embedded Systems w/ ARM Cortex-M Microcontrollers in Assembly Language and C (Yifeng Zhu)
  - Chapter 8.1, 8.2, 8.6
  - Chapter 14.1, 14.2, 14.3, 14.6
  - Chapter 19.1

Group Exercise

```
.global _start
.space 8
data:
        .byte 0x12, 20, 0x20, -1
_start:

load:
        mov r0, #0
        mov r4, #0
        movw r1, #:lower16:data
        movt r1, #:upper16:data

loop:
        ldrb r2, [r1], #1
        add r4, r4, r2
        add r0, r0, #1
        cmp r0, #4
        bne loop

deadloop:
        b deadloop
_end:
```

1. What does this program do?
2. What is the final value of registers R0, R2, and R4?
3. What does ***deadloop*** do? Why include it?

What are we learning about today?

- Stack and Application Binary Interface (ABI)
- Memory-mapped I/O (MMIO)
- Bus Architectures
- ARM Advanced Peripheral Bus (APB)

Application Binary Interface

A ”contract” between programmers on how to share hardware resources

When you call a function written by another programmer, that function will need to use registers.

What happens if you have important data in those registers?

Application Binary Interface (ABI)

Some Basic Rules:

1. A subroutine must preserve the contents of the registers R4-R11 and SP
2. Arguments are passed through R0 to R3
3. Return value is placed in R0
4. Allocate space in the stack as needed. Use it as needed.
5. Push and pop an even number of registers to maintain an 8-byte alignment in the stack.

Violation Example

Caller Program

```
MOV r4, #100
...
BL foo
...
ADD r4, r4, #1        ; r4 = 101, not 11
```

Subroutine/Callee

```
foo PROC
    ...
    MOV r4, #10       ; foo changes r4
    ...
    BX  LR
    ENDP
```

How can we solve this issue?

Stack

- The **stack** is last-in-first-out temporary storage
- On the ARM Cortex – M processor, the **stack always operates on 32-bit data**
- The **stack pointer** always points to the 32-bit data on top of the stack
- The stack **grows downward in memory** as we push data on it
  - We refer to the most recent item as the “top of the stack” but it is actually the item stored in the lowest address!

PUSH {Rd}

- SP = SP – 4
- (*SP) = Rd
  - A **full stack** is if the memory location pointed by the SP holds the last item that was pushed onto the stack

Push Multiple Registers

```
PUSH {r7, r6, r8}
```

```
PUSH {r8, r7, r6}
```

```
PUSH {r6}
PUSH {r7}
PUSH {r8}
```

POP {Rd}

- SP = SP + 4
- Rd = (*SP)
  - An **empty stack** is when the memory location pointed by the SP is an empty spot and will store the next item to be pushed onto the stack

Pop Multiple Registers

```
POP {r7, r6, r8}
```

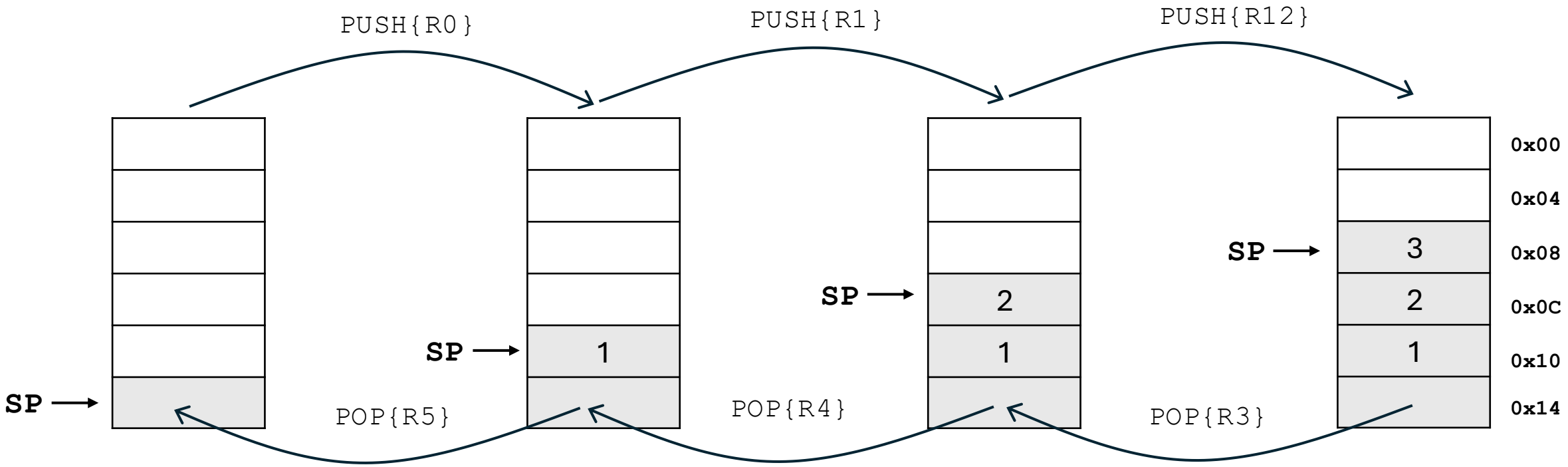
```
POP {r8, r7, r6}
```

```
POP {r6}
POP {r7}
POP {r8}
```

Stack Example

```
PUSH {R0}
PUSH {R1}
PUSH {R2}
POP  {R3}
POP  {R4}
POP  {R5}
```

Assume that initially R0 contains the value 1, R1 contains the value 2, and R3 contains the value 3



Preserve the runtime environment via Stack

Caller Program

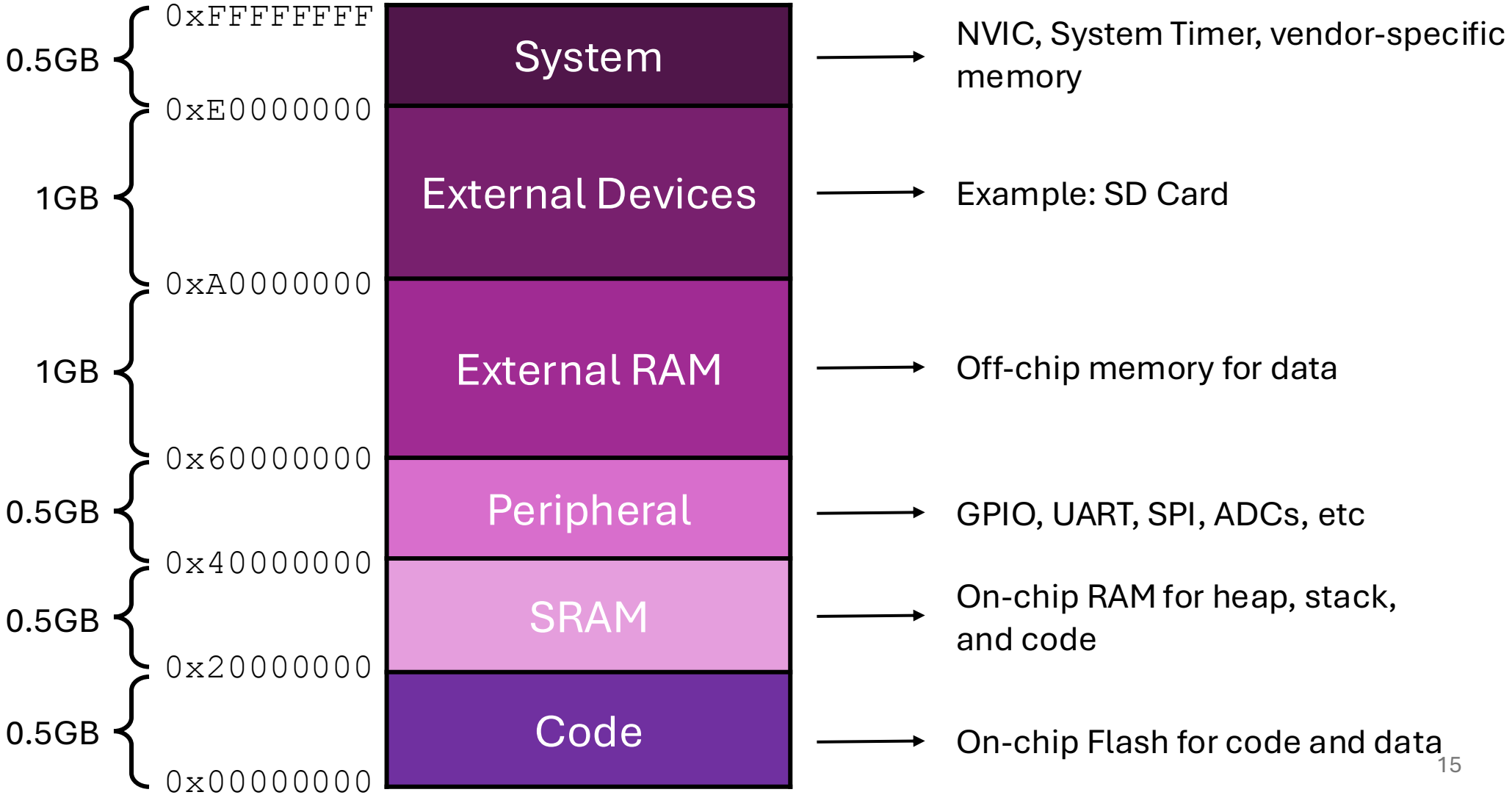
```
MOV r4, #100
...
BL foo
...
ADD r4, r4, #1        ; r4 = 101, not 11
```

Subroutine/Callee

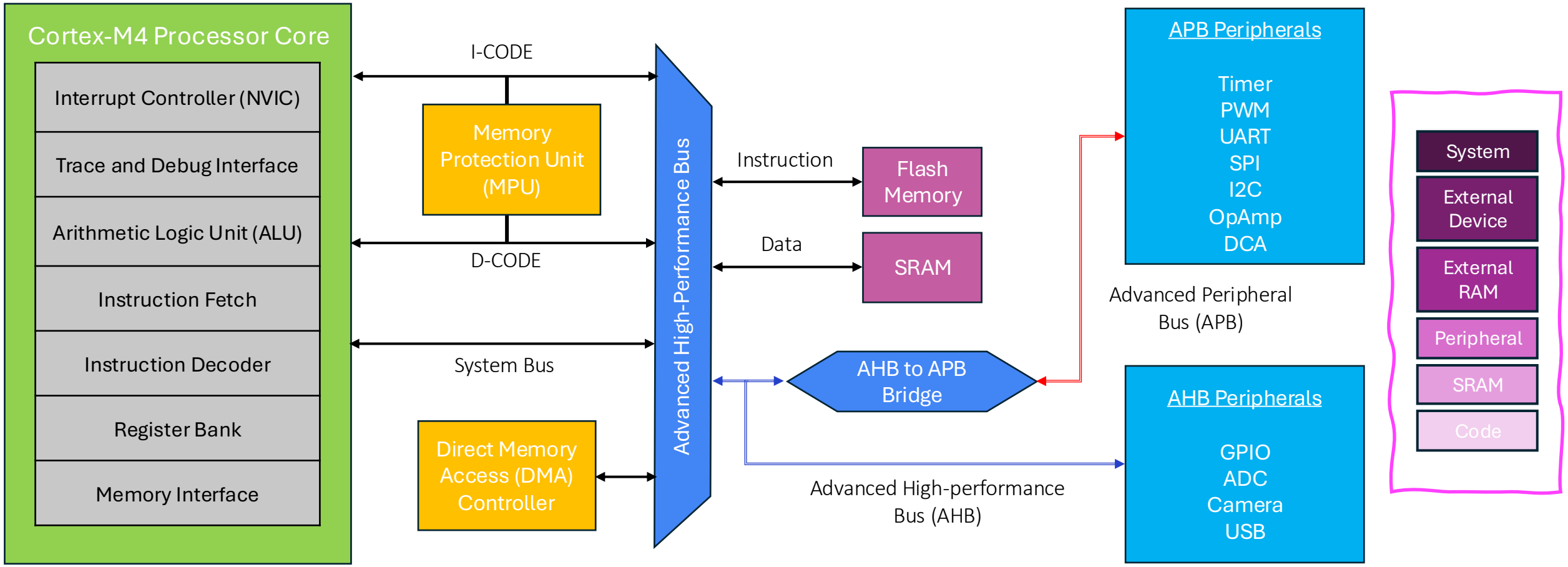
```
foo PROC
    PUSH {r4}         ; preserve r4
    ...
    MOV r4, #10       ; foo changes r4
    ...
    POP  {r4}         ; recover r4
    BX   LR
    ENDP
```

Memory Mapped I/O (MMIO)

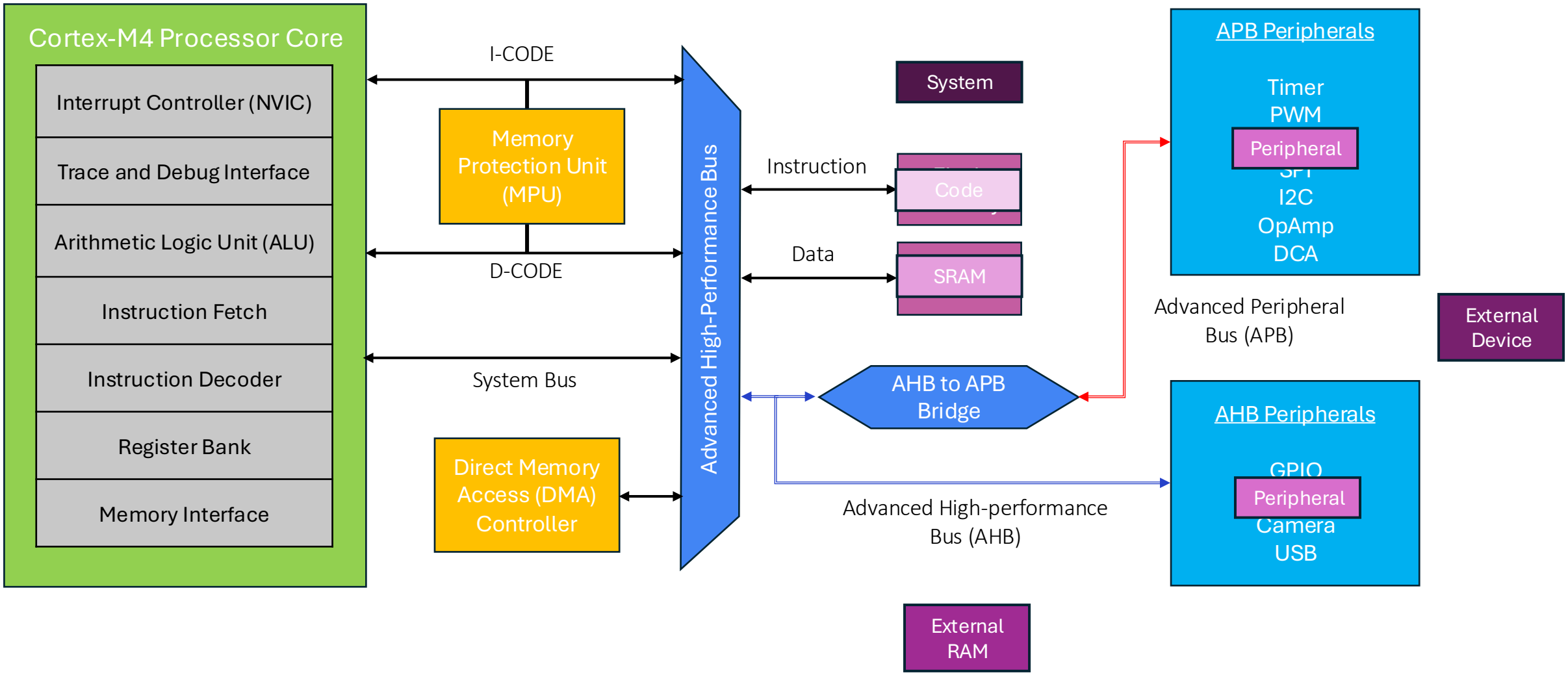
Cortex-M4 Memory Map



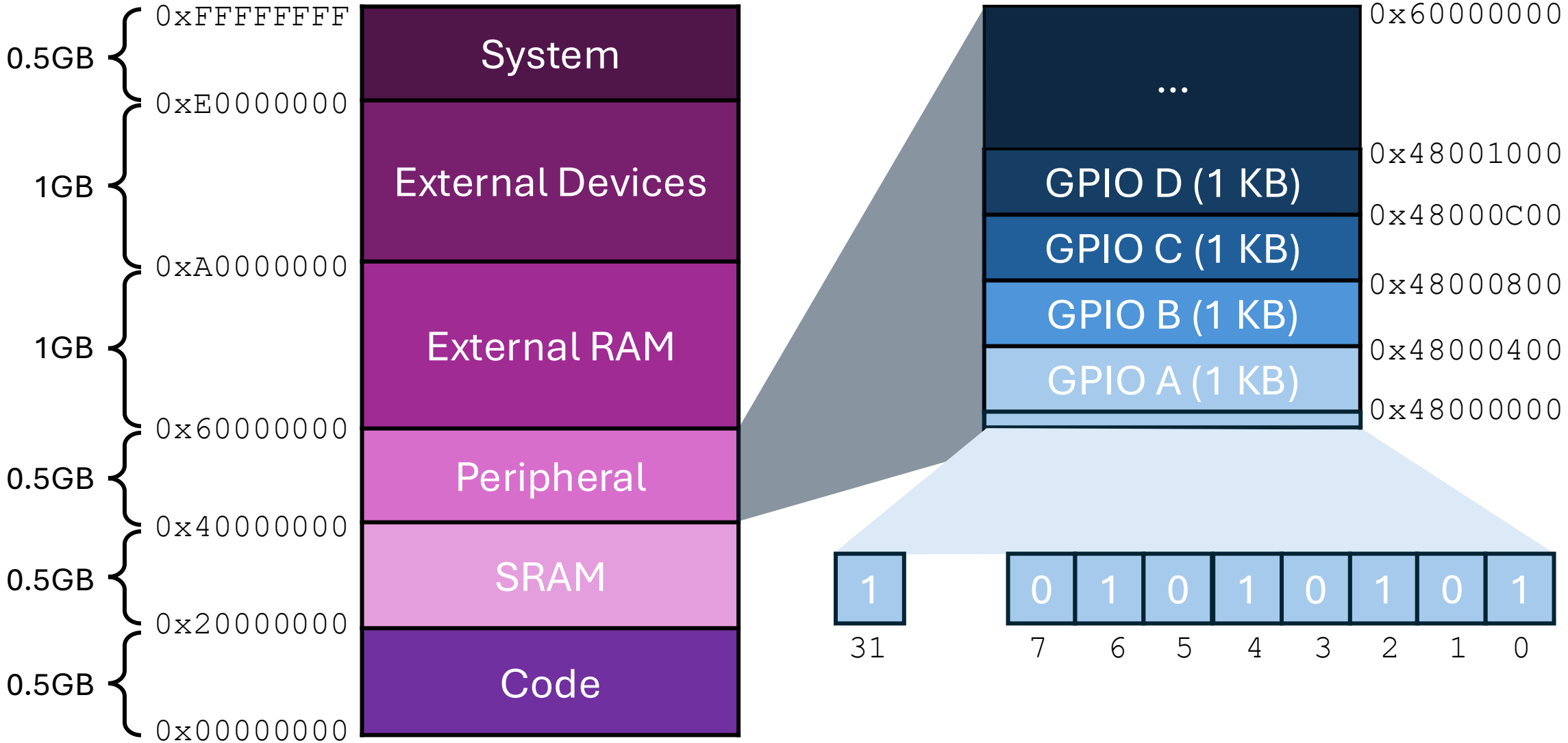
Memory blocks are distributed



Memory blocks are distributed



Memory Map for GPIO Peripheral



Basic Idea – Memory Mapped I/O

Port-mapped I/O uses special machine instructions that are designed specifically for I/O instructions

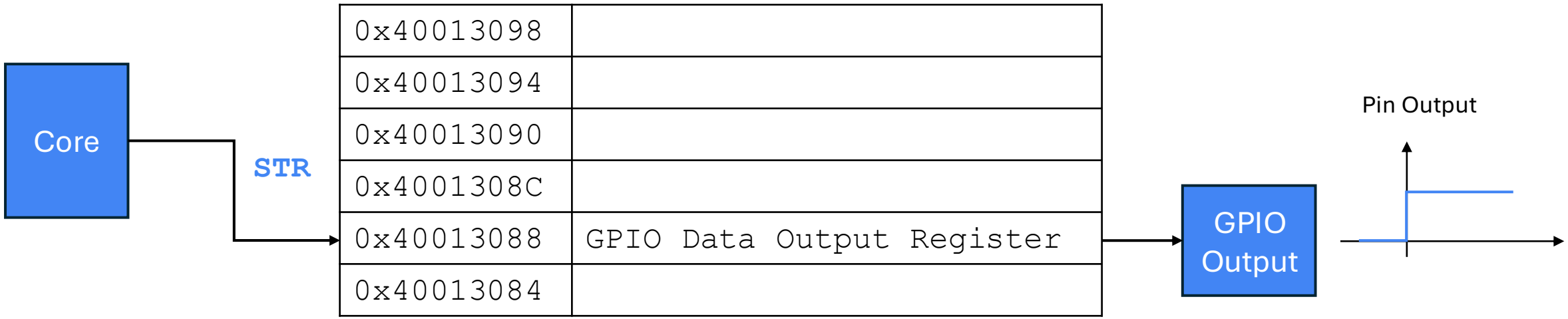
Memory-mapped I/O does not need any special instructions.

The memory and the I/O devices share the same address space.

Each peripheral register or data buffer is assigned to a memory address in the memory space of the microprocessor.

Basic Idea - Memory Mapped I/O

- A convenient way to interface with I/O devices
- Each device registers is assigned to a memory address in the address space of the microprocessor
- Use native CPU load/store instructions (LDR/STR)



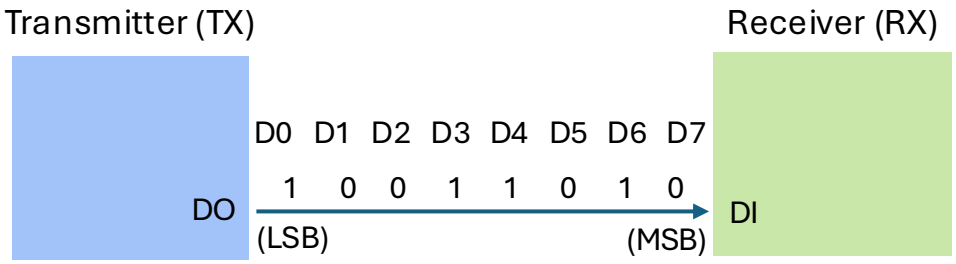
Bus Architectures

Data Bus Types

A ***data bus*** is a set of wires, supporting circuits and protocol, that provides transportation for data between two locations

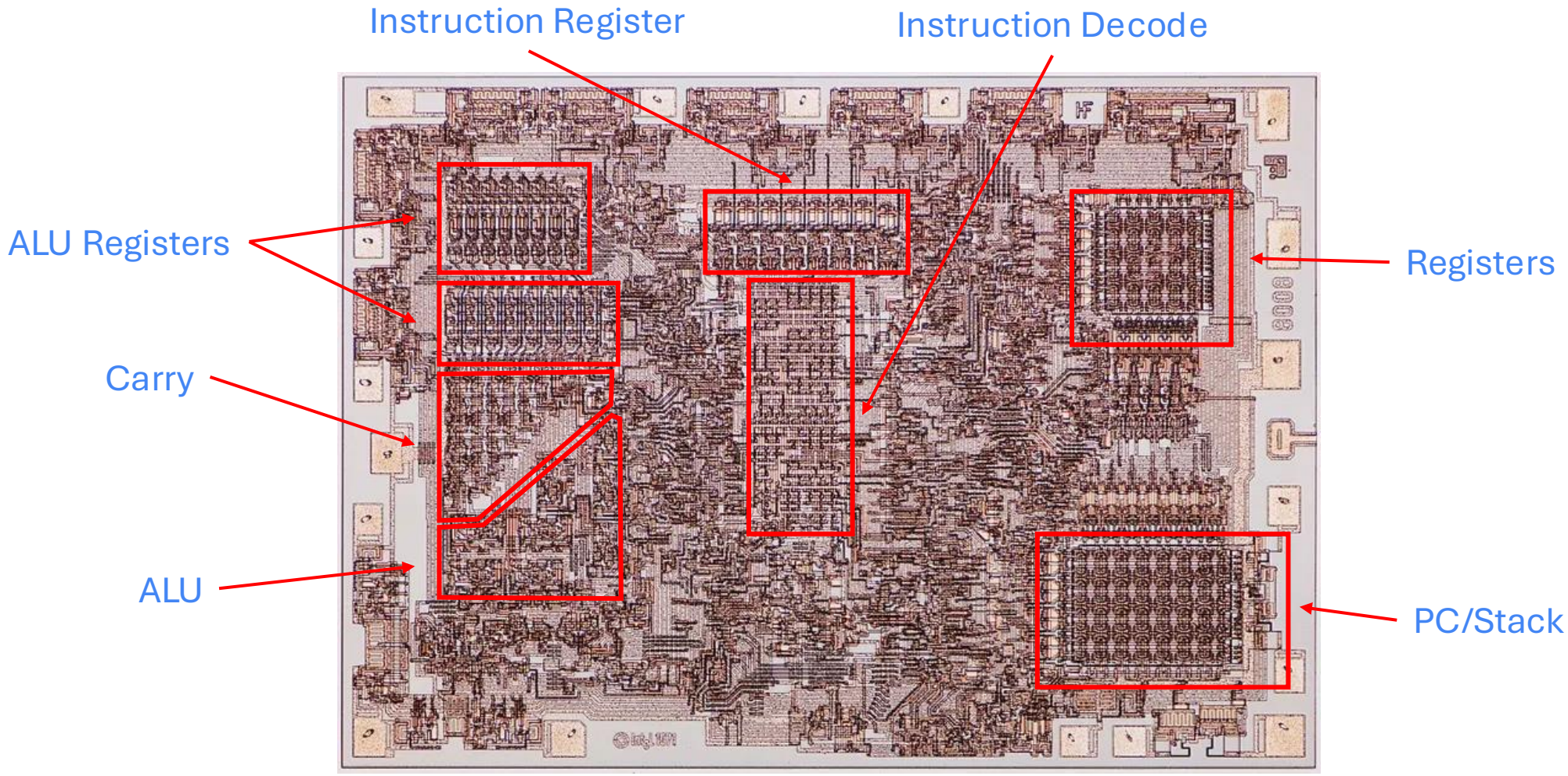
Parallel Interface Example

Serial Interface Example



Which type of bus is most used on chip?

Intel 8008 Microprocessor

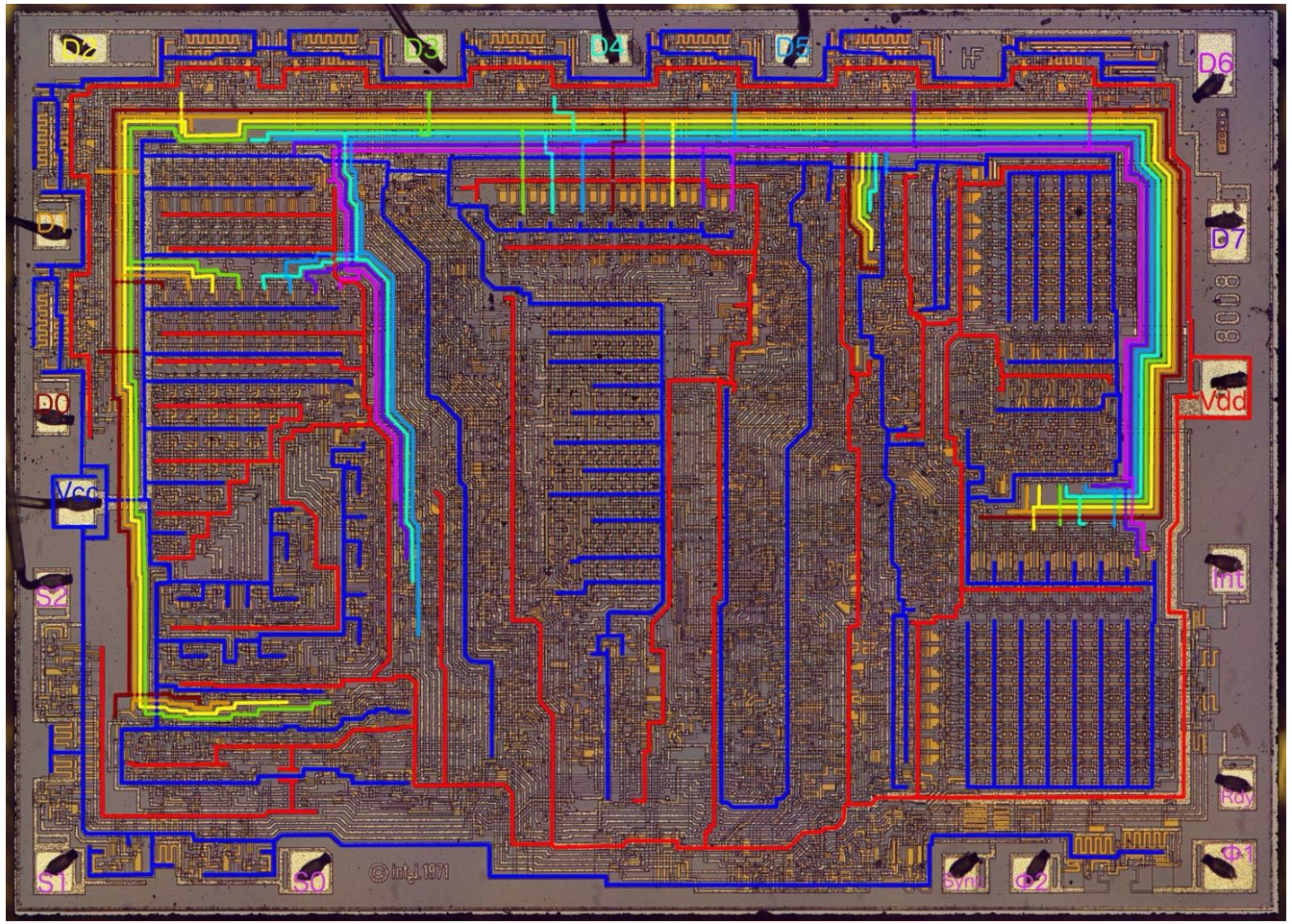


Which type of bus is most used on chip?

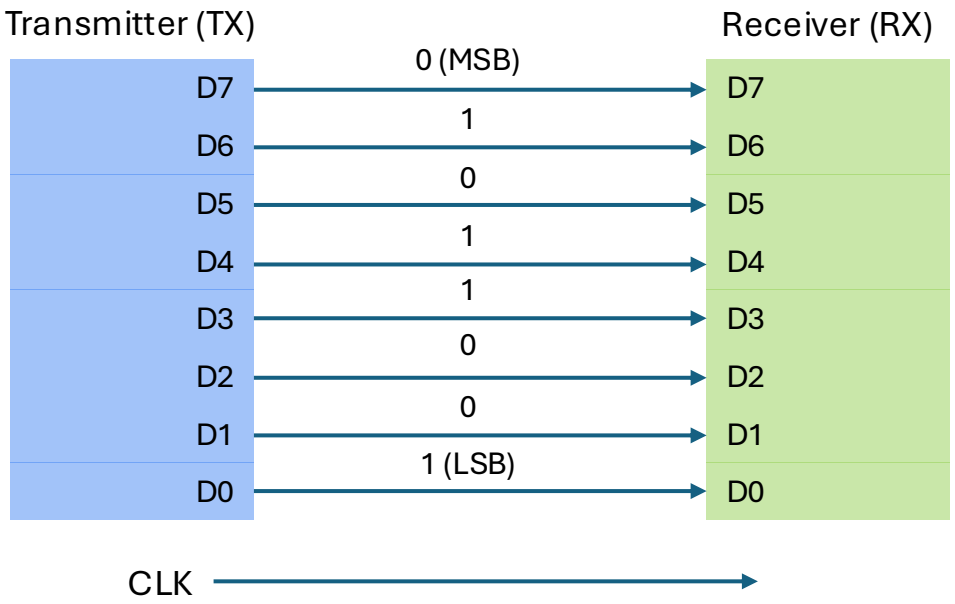
Intel 8008 Microprocessor

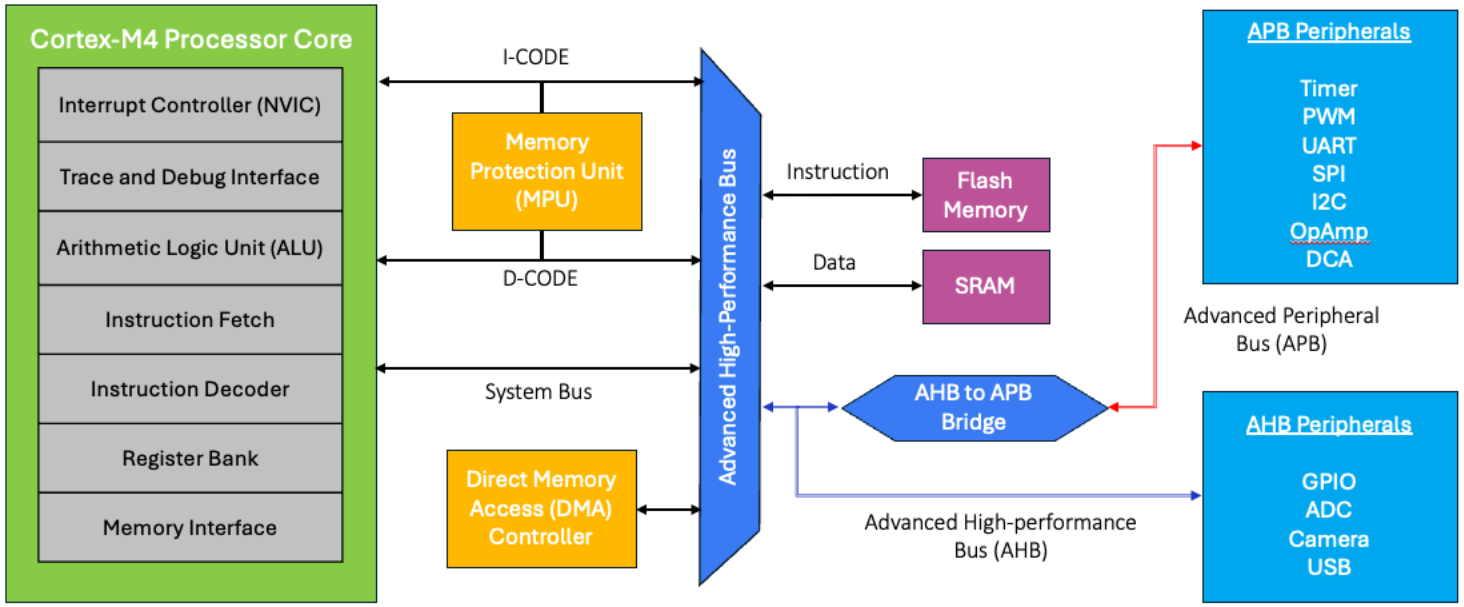
Power Bus: Red and Blue

8-bit Parallel Data Bus:
Shown in 8 rainbow colors



How do we send to individual peripherals?





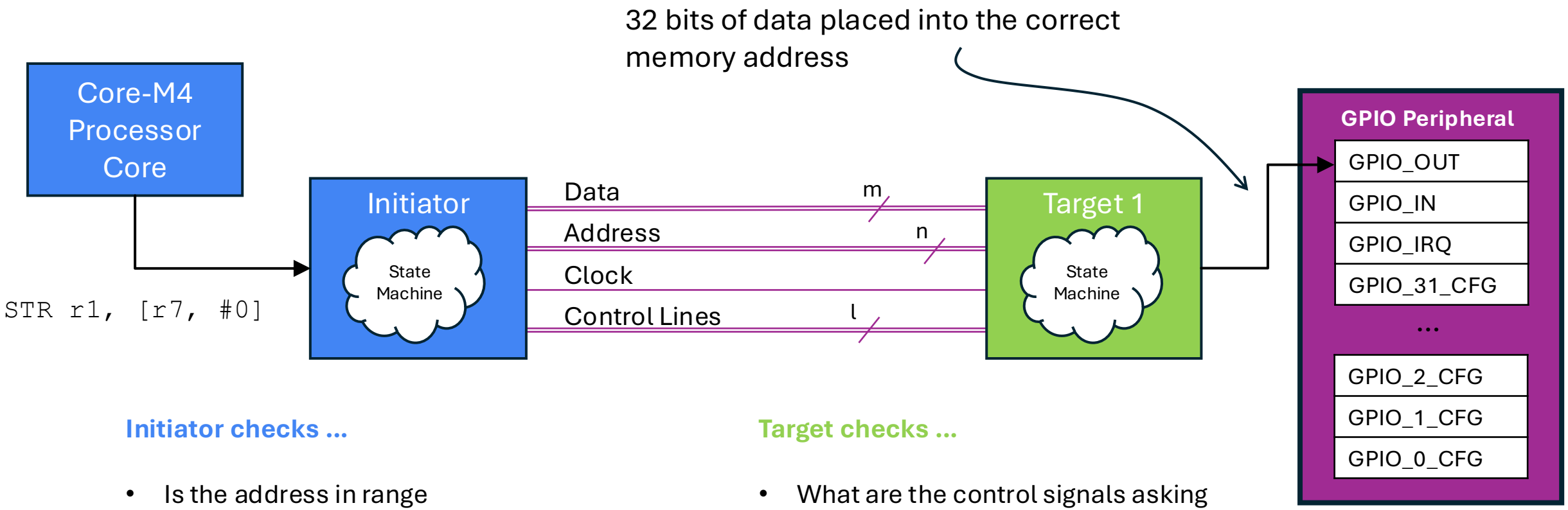
Bus Terminology

- A transaction is between an ***initiator*** and a ***target***
- An **Initiator** can start a transaction event and controls which devices receive and transmit data.
- A **Target** device can only transmit when commanded to by the Initiator.

Initiators and Targets share resources

- When not using a bus **Targets must not transmit**
- Tri-State drivers are used to “disconnect” bus
- Some wires are shared amongst devices while others are point-to-point connections

How do we use buses on the Cortex-M4?



Initiator checks ...

- Is the address in range
- Load or Store
- Control and timing signals are needed

Target checks ...

- What are the control signals asking for
- Receiving data → what to do with it?
- Transmitting data --> where do I get it from?

Advanced Microcontroller Bus Architecture (AMBA)

An on-chip communication standard for embedded microcontrollers

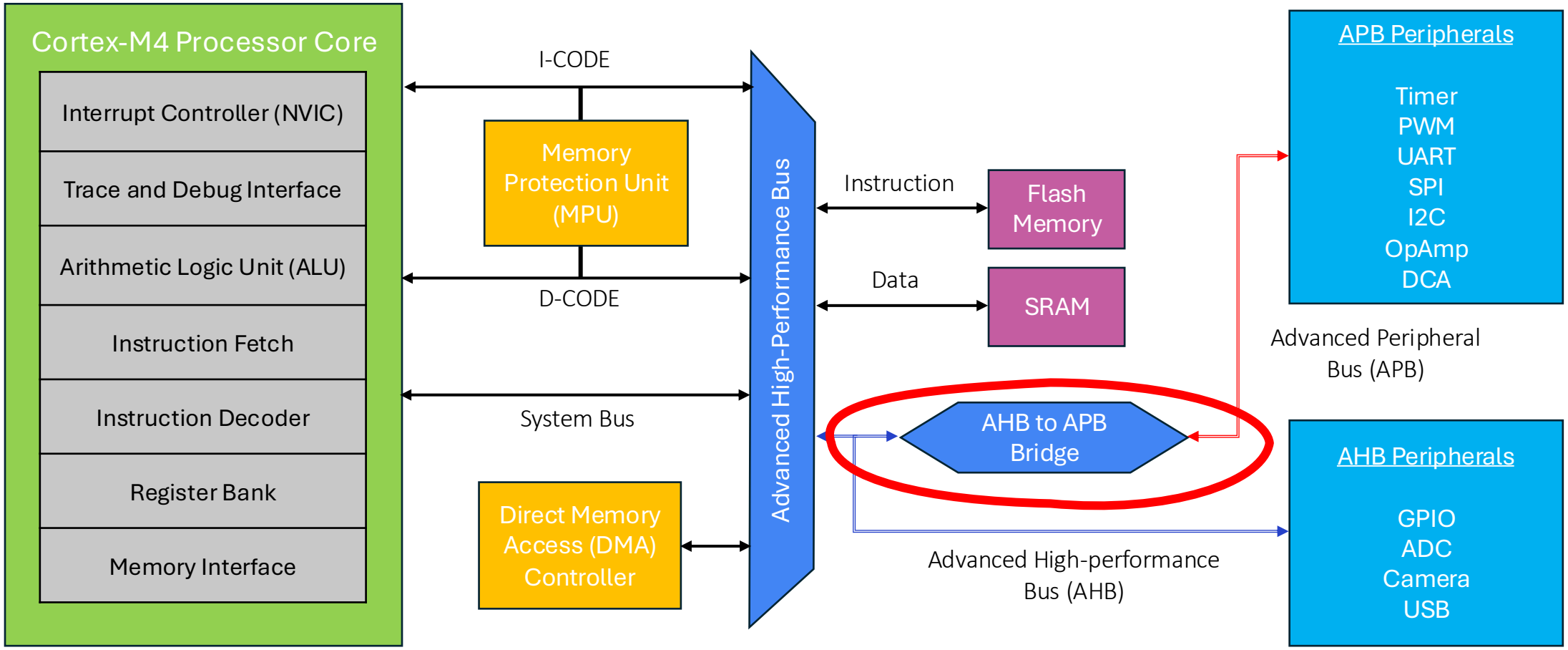
Advanced High-performance Bus (AHB)

- High performance
- Pipelined operation
- Burst transfer
- Multiple bus initiators
- Split transactions

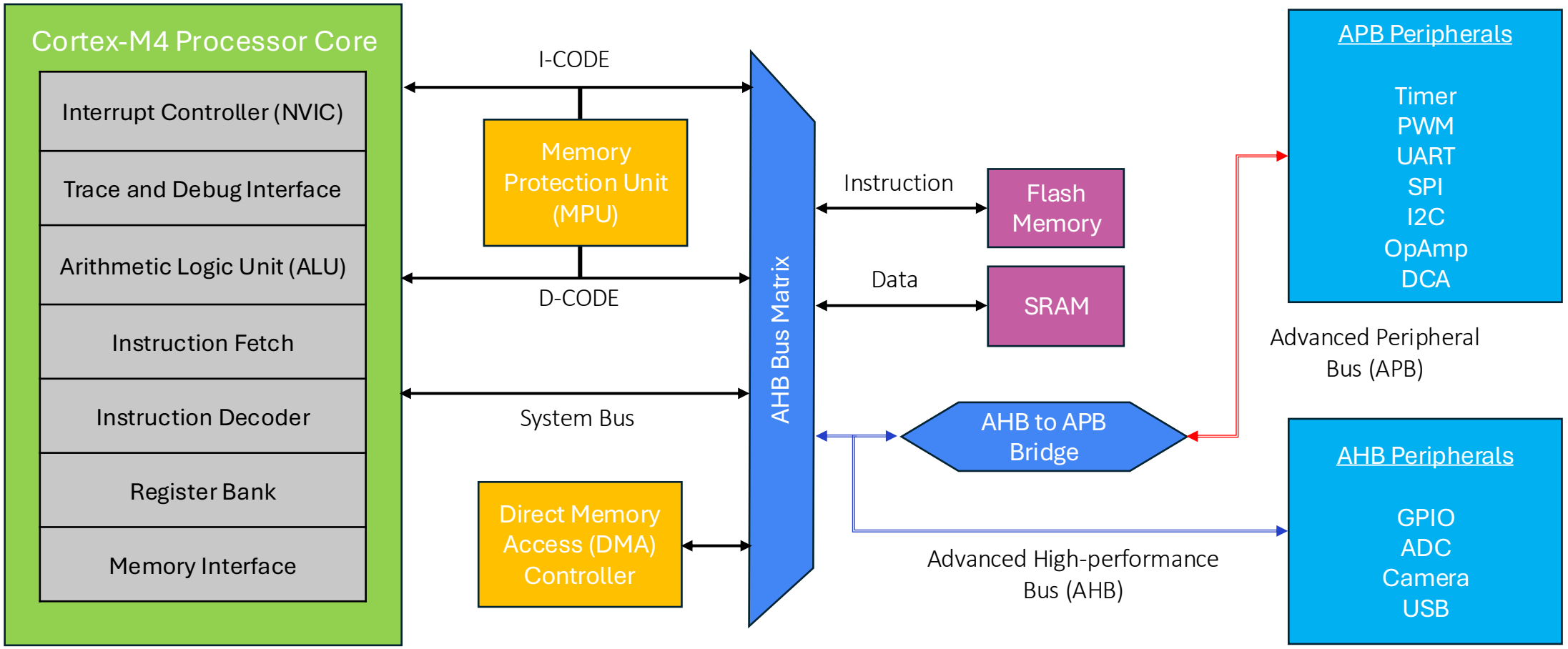
Advanced Peripheral Bus (APB)

- Low power
- Latched address/control
- Simple interface
- Suitable for many peripherals

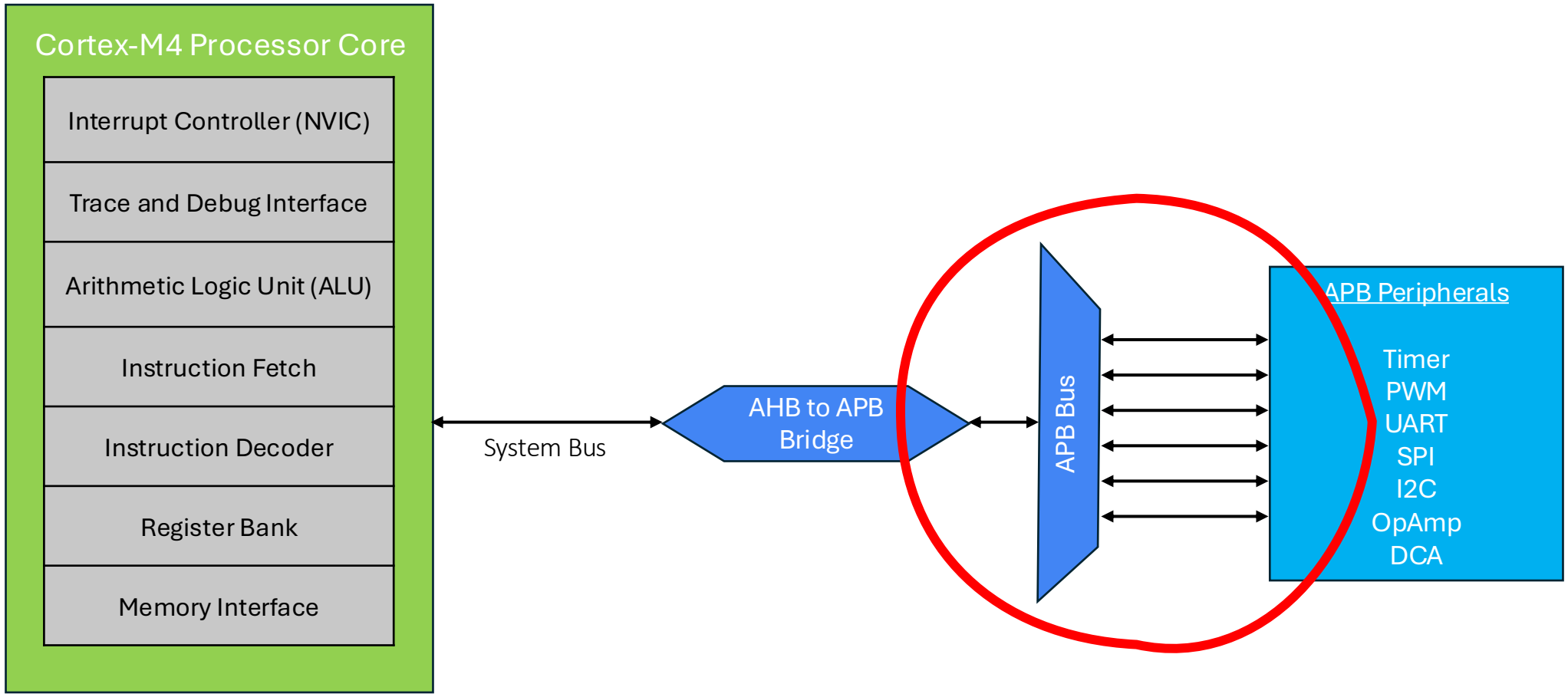
Advanced Microcontroller Bus Architecture (AMBA)



How do we interface with individual peripherals?



How do we interface with individual peripherals?



Advanced Microcontroller Bus Architecture (AMBA)

EE 186

- Home
- Course Overview
- Course Schedule
- Assignments
- **Resources**
- Staff

ARM

ARMv7 Architecture Reference Manual

AMBA Advanced High-performance Bus (AHB) Protocol Specification

AMBA Advanced Peripheral Bus (APB) Protocol Specification

STM32

STM32 Reference Manual

STM32 Datasheet

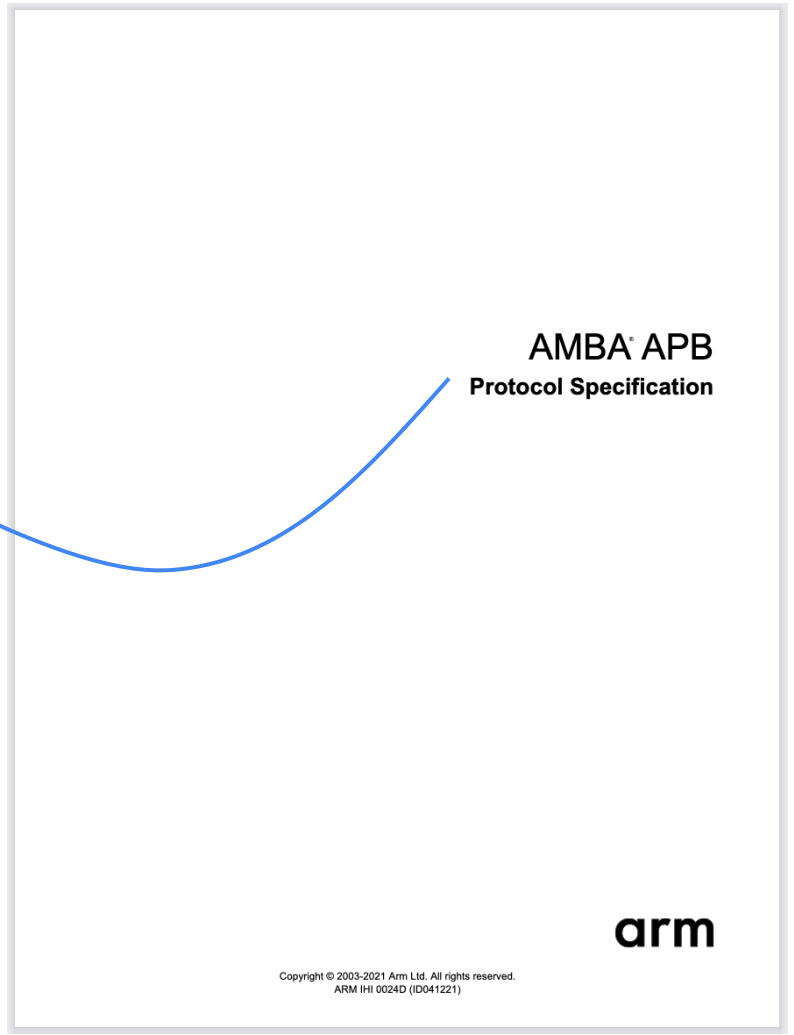
STM32 IDE

STM32 Videos

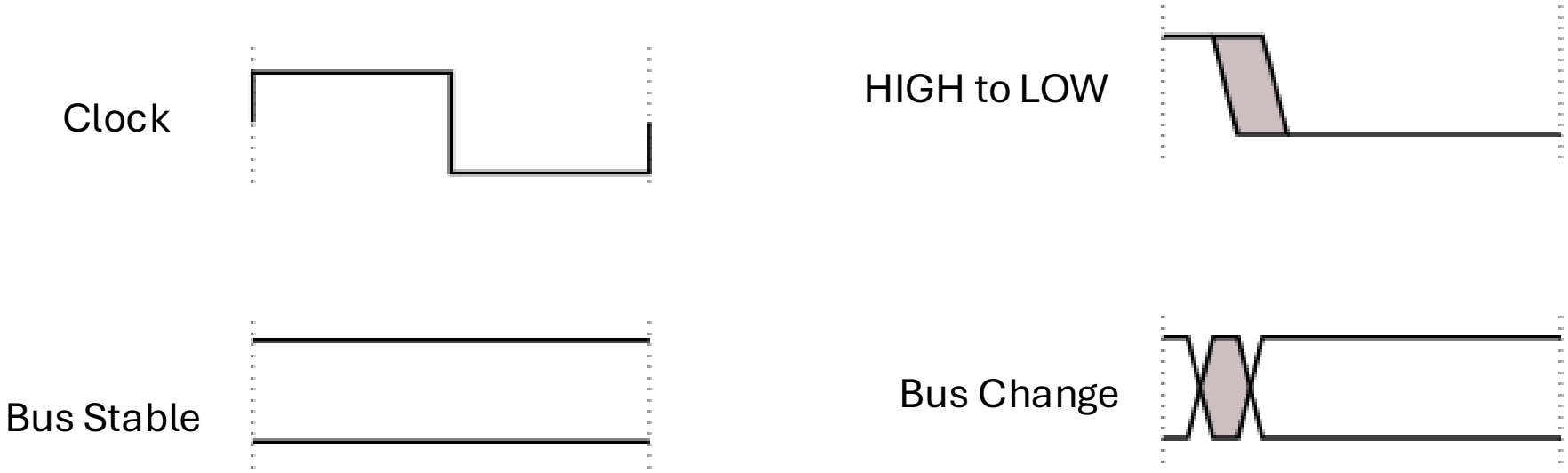
Nucleo-144 User Manual

Nucleo Board Pinout

Nucleo Development Board Website

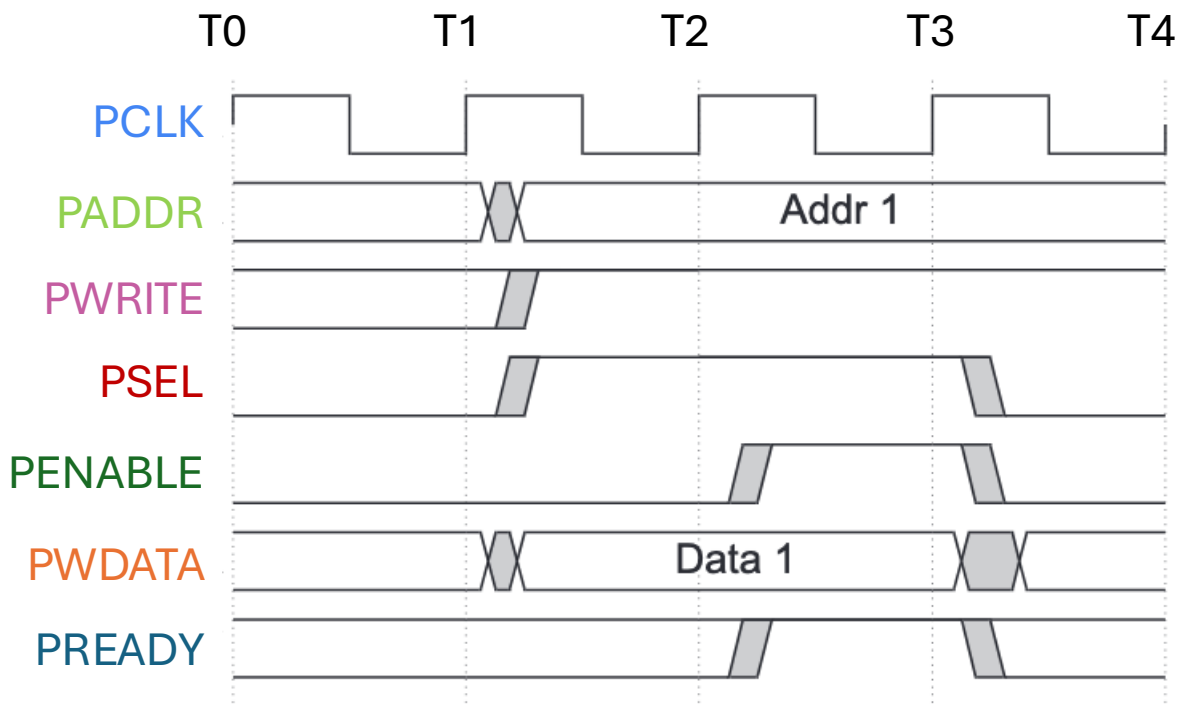


Some APB notation you might see

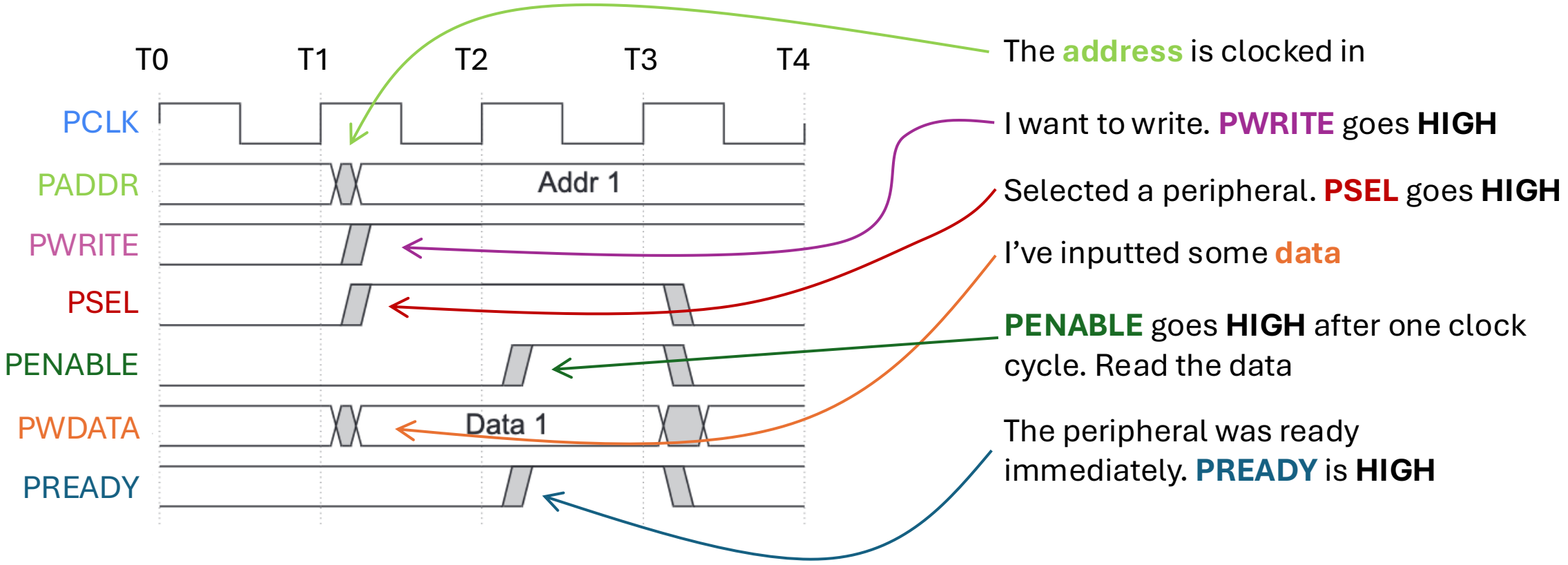


APB Bus Signals

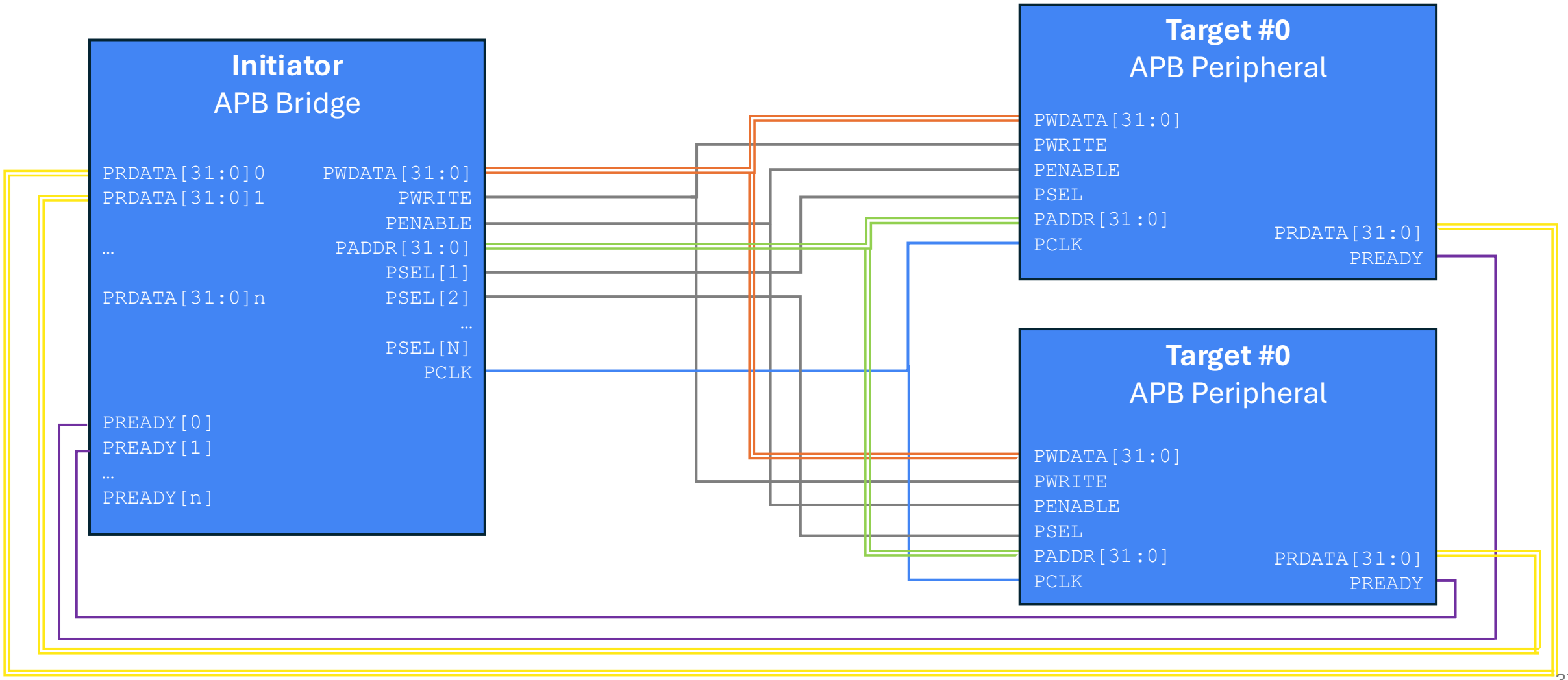
- PCLK – [1 bits]
  - Clock ~pos edge
- PADDR – [32 bits]
  - Address on the bus
  - Up to 32 bits but not always
- PWRITE – [1 bit]
  - 1 = Write
  - 0 = Read
- PWDATA – [32 bits]
  - Data written to the I/O device
- PSEL
  - Asserted if the current bus transaction is targeted to **this device**
  - Each target has a PSEL
- PENABLE
  - High during entire transaction other than the first cycle
- PREADY
  - Driven by the target
  - Indicates if the target is **ready** to do the transaction
  - Each target has a PREADY



What's happening here?



APB: One Initiator, Multiple Targets



Example: Driving Multiple Targets

Let’s assume we have one APB Initiator and two Target devices (D1 and D2)

- D1 is mapped to addresses `0x00001000 – 0x0000100F`
- D2 is mapped to addresses `0x00001010 – 0x0000101F`

The Cortex-M4 processor Core executes the following instructions:

```
STR R0, [R1, #4]
```

```
R0 = 0x20000000
R1 = 0x00001000
```

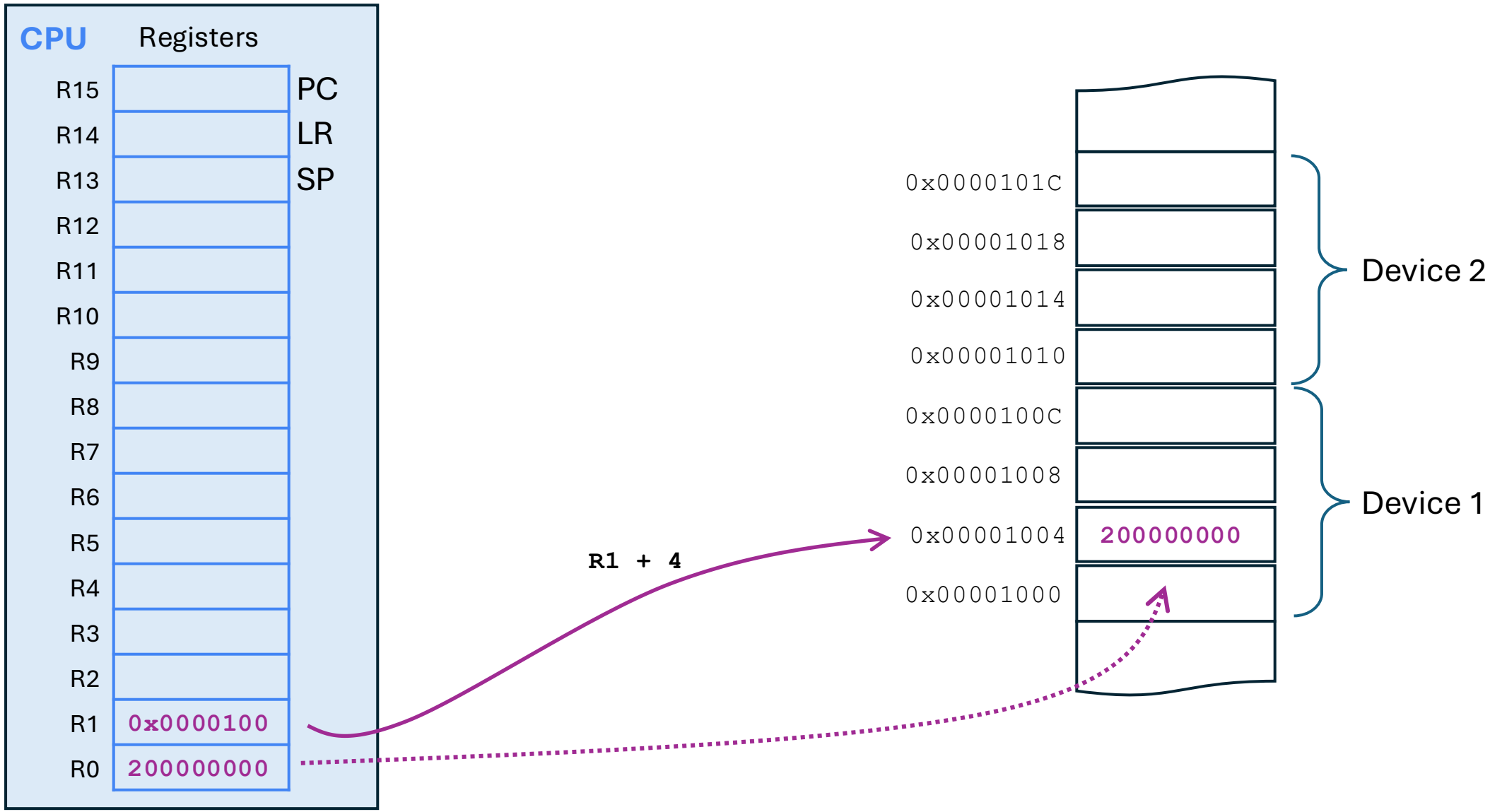
Example: Driving Multiple Targets

Execute:

```
STR R0, [R1, #4]
```

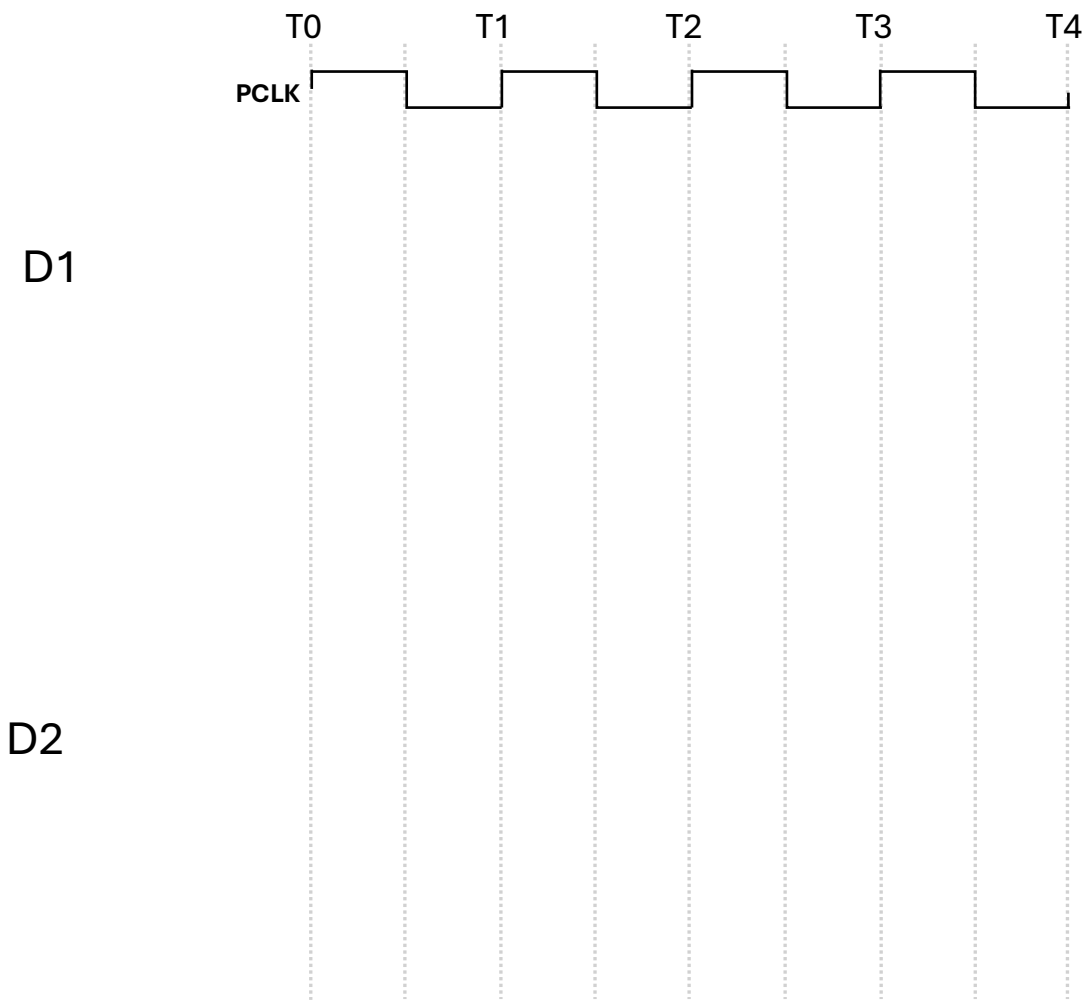
Given:

```
R0 = 0x00C0FFEE
R1 = 0x00001000
```



Group Exercise:

Show the signals for both devices



Let's assume we have one APB Initiator and two Target devices (D1 and D2)

- D1 is mapped to addresses `0x00001000 – 0x0000100F`
- D2 is mapped to addresses `0x00001010 – 0x0000101F`

The Cortex-M4 processor Core executes the following instructions:

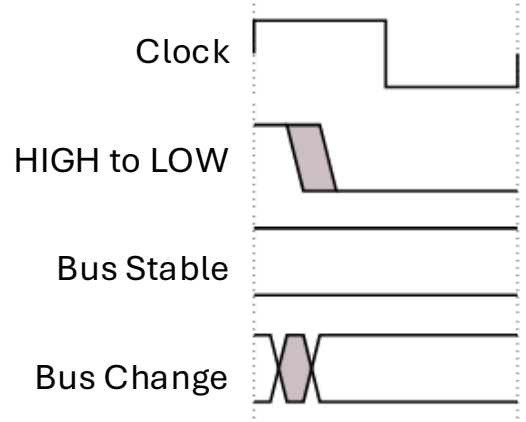
Execute:

```
STR R0, [R1, #4]
```

Given:

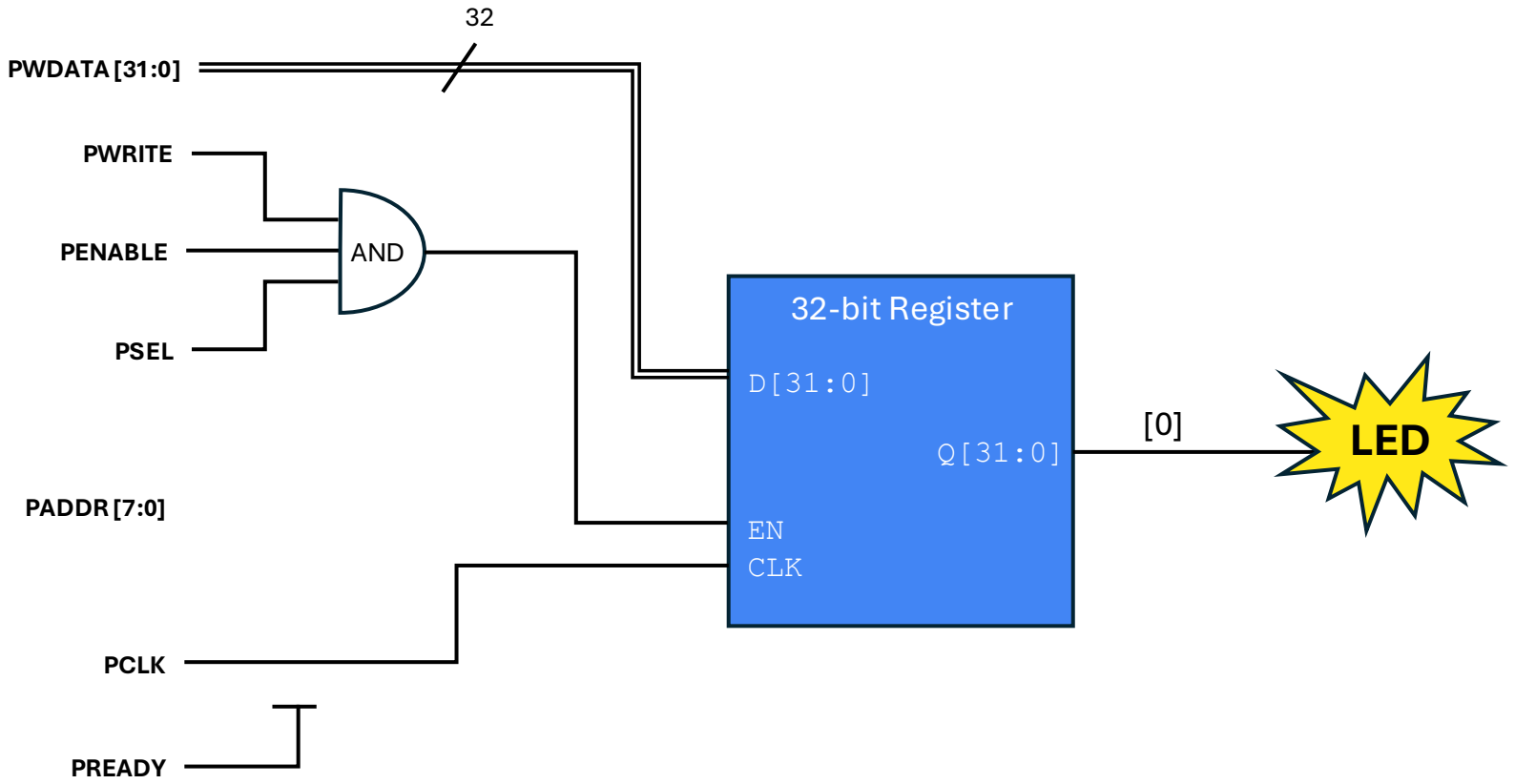
```
R0 = 0x00C0FFEE
R1 = 0x00001000
```

Remember →



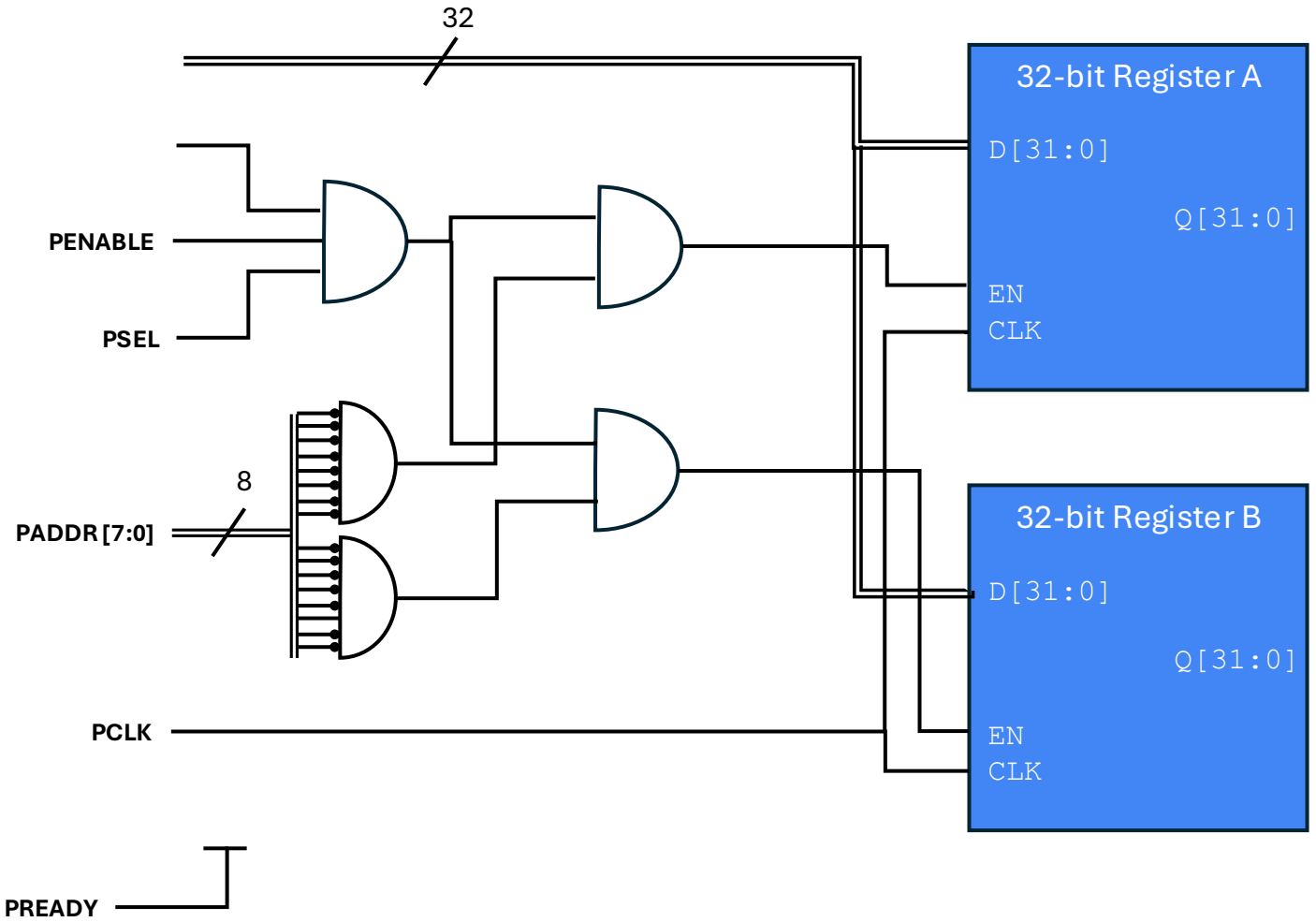
Example: How do we control an LED with the LSB of this register?

We are assuming the APB only gets the lowest 8-bits of the address



Example: Multiple registers in one peripheral

We are assuming the APB only gets the lowest 8-bits of the address

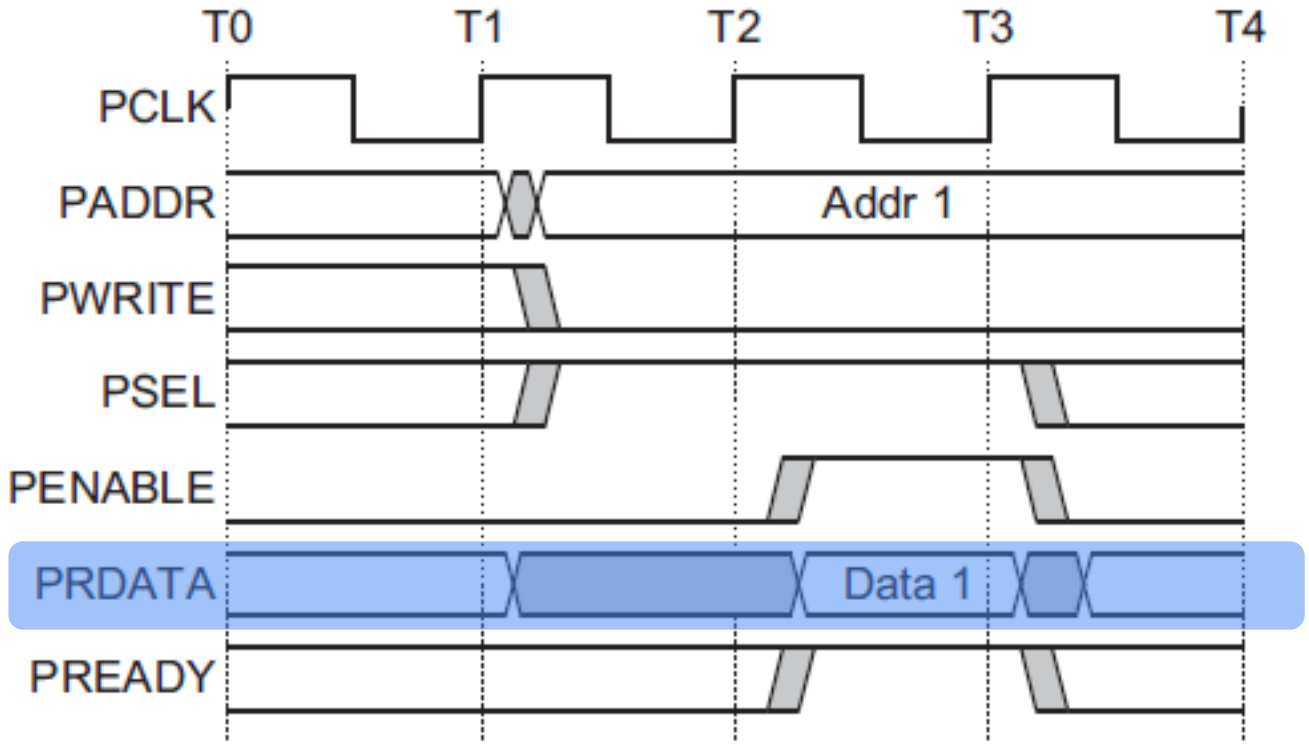


Register A should be written at address **0x00001000**

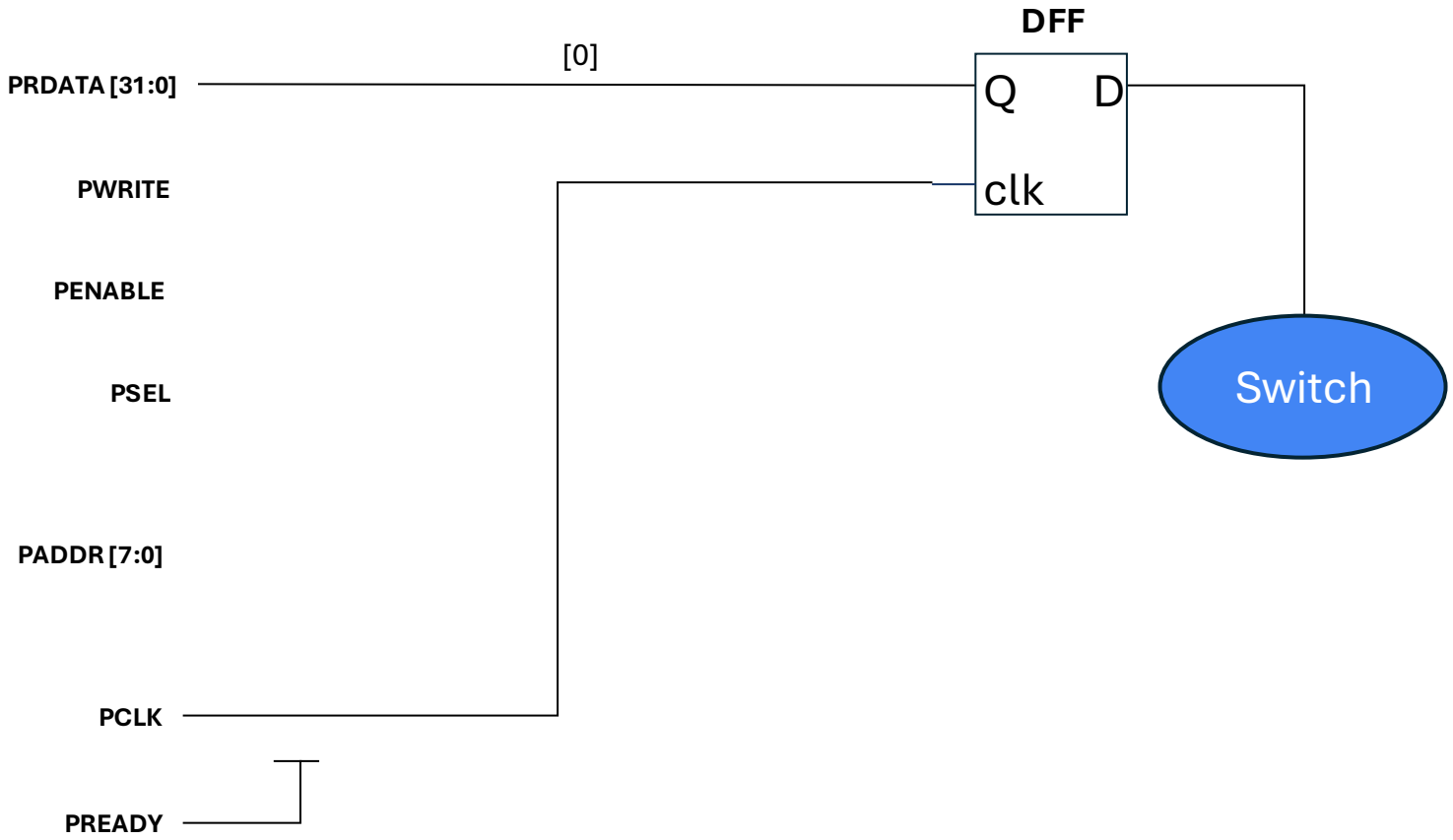
Register B should be written at address **0x00001004**

What about APB reads?

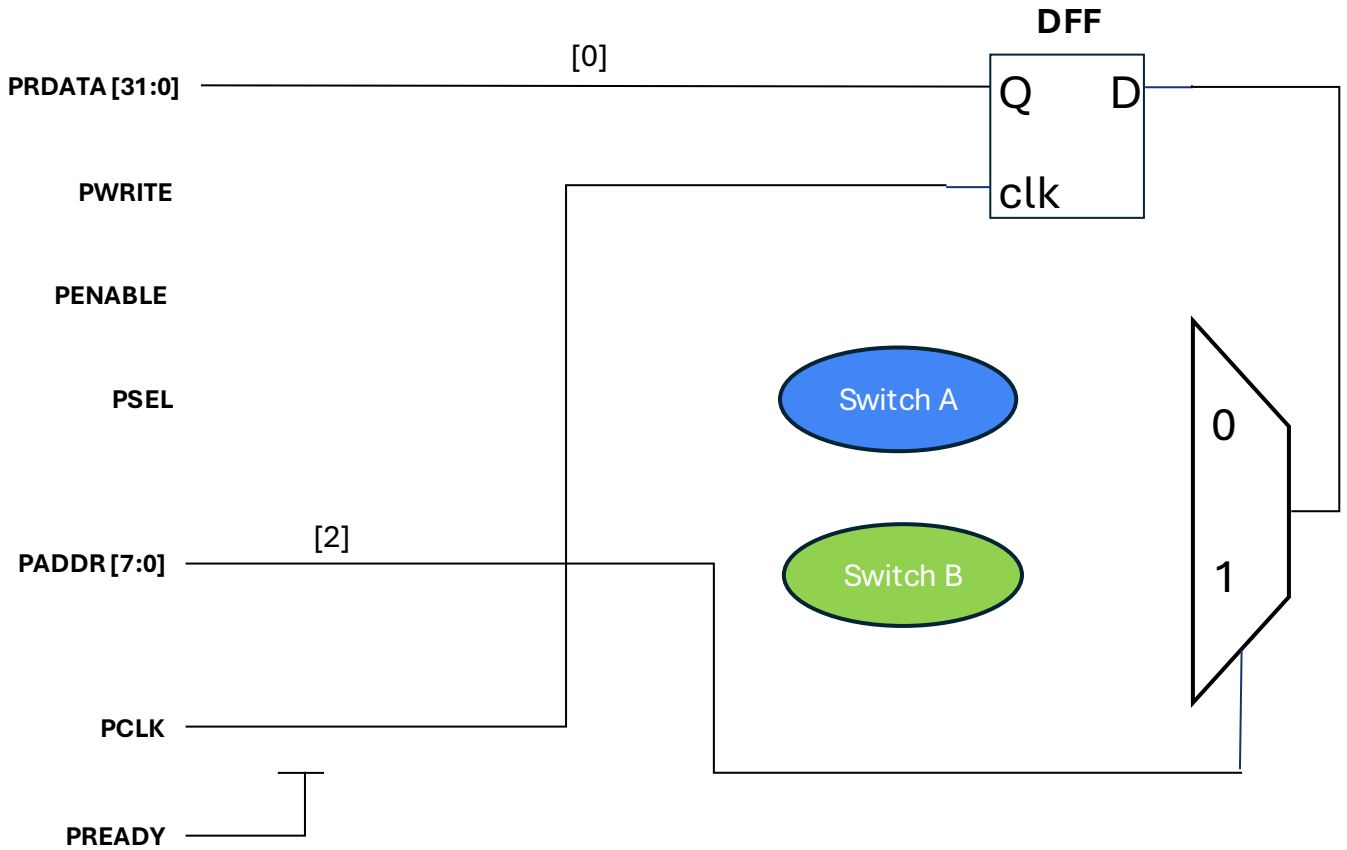
Each target device has its own read data (PRDATA) bus!



Example: Reading data from a switch



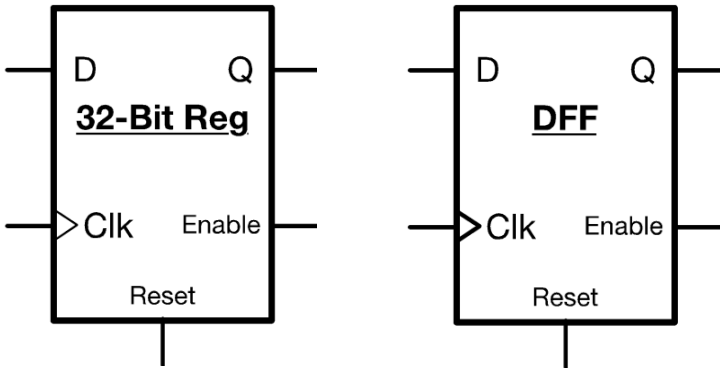
Example: Reading from a select switch



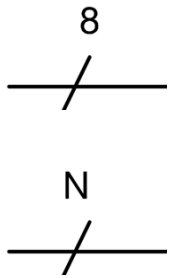
If **0x00001000** read from **A**
If **0x00001004** read from **B**

Some Standard Notation

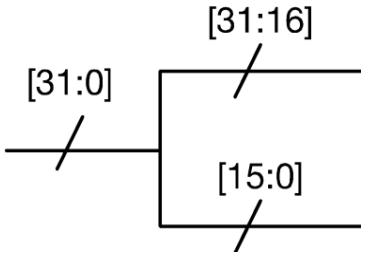
Expand DFF and Registers



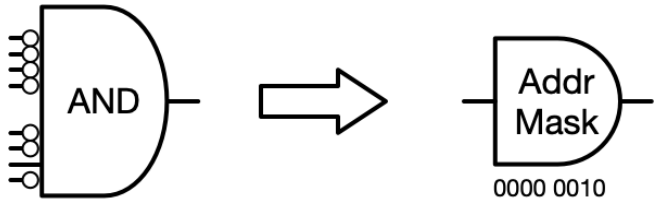
Label the number of lines in the bus



Bus concatenation and data labeling



Selecting Address Lines



VDD = logical ‘1’

GND = logical ‘0’

Group Exercise:

All reads read from register, all writes write…

We are assuming the APB only gets the lowest 8-bits of the address

PRDATA [31:0]

PWDATA [31:0]

PWRITE

PENABLE

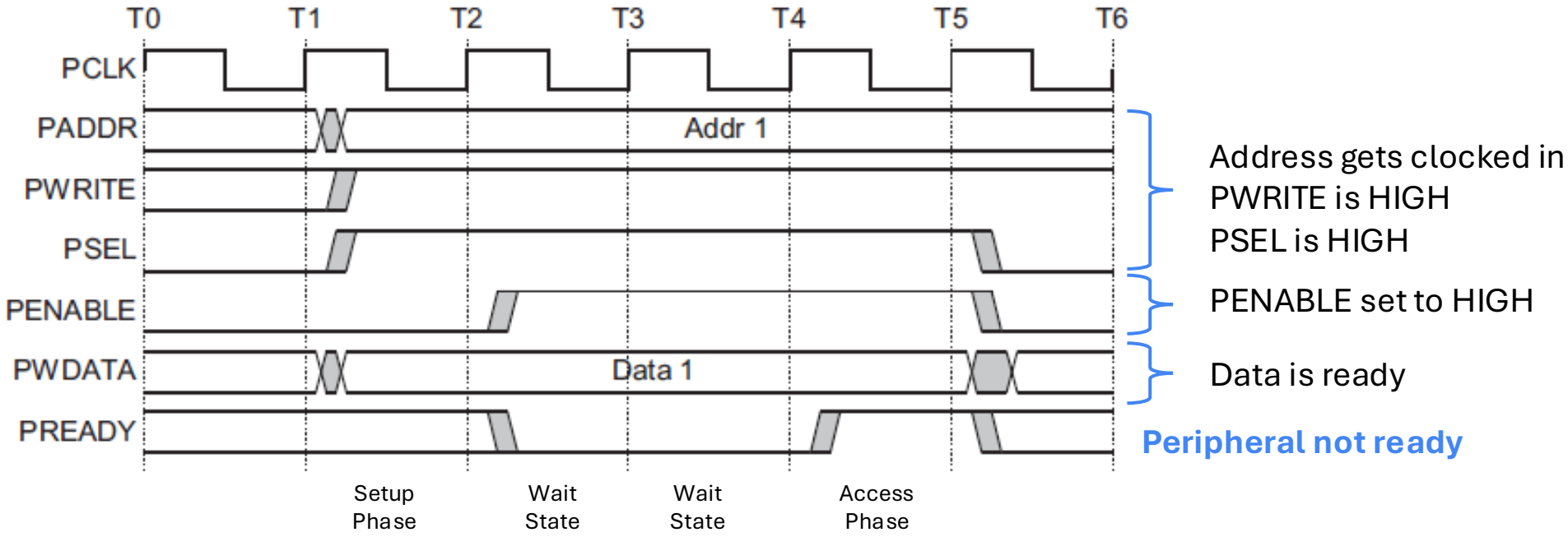
PSEL

PADDR [7:0]

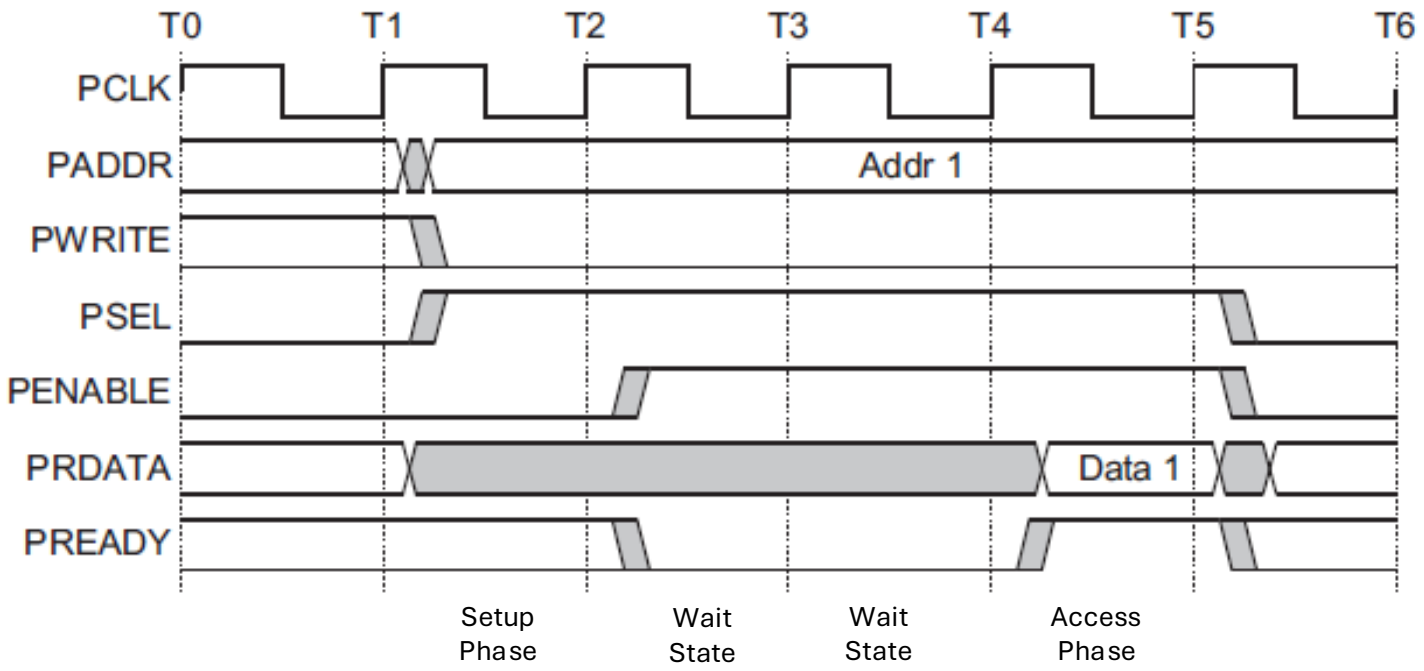
PCLK

PREADY

A **write** transfer with wait states



A **read** transfer with wait states



APB State Machine

IDLE

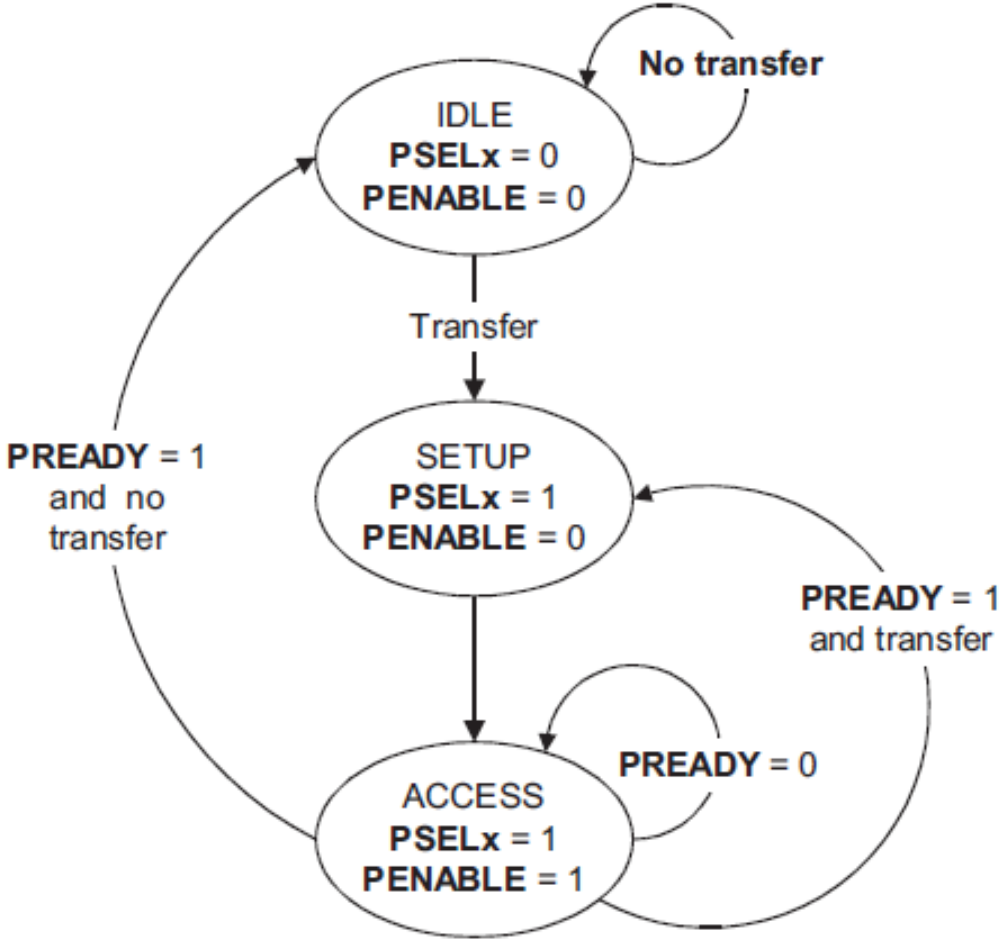
- The default APB state

SETUP

- When a transfer is required
- PSELx is asserted
- Only one clock cycle

ACCESS

- PENABLE is asserted
- Addr, write, select, and write data remain stable
- Stay if PREADY = 0 (LOW)
- Go to IDLE if PREADY = 1 (HIGH)
- Go to SETUP if PREADY = 1 and more data is pending



Recap: APB Signals

- Documentation describes all APB signals
- References on class webpage

Table 2-1 APB signal descriptions

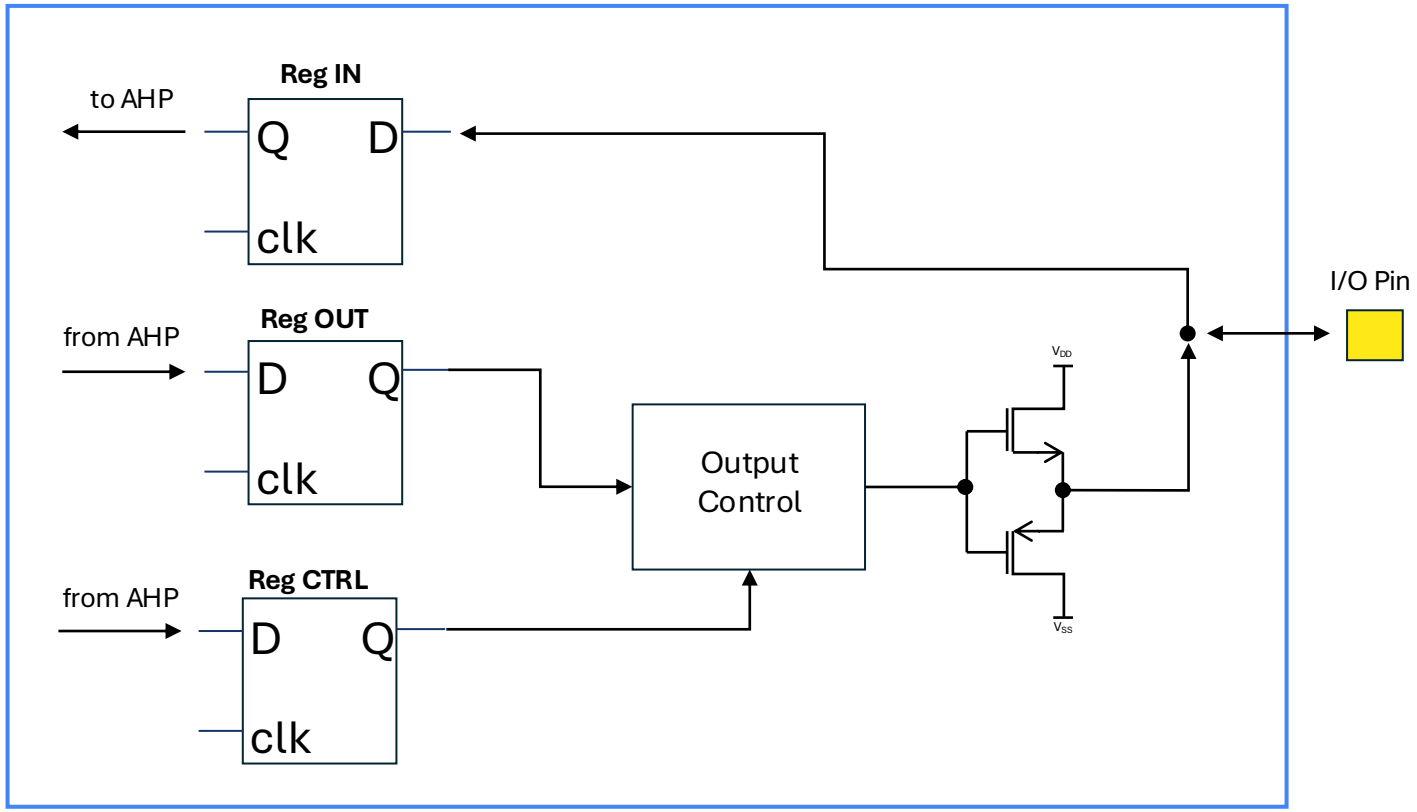
Signal	Source	Description
PCLK	Clock source	Clock. The rising edge of **PCLK** times all transfers on the APB.
PRESETn	System bus equivalent	Reset. The APB reset signal is active LOW. This signal is normally connected directly to the system bus reset signal.
PADDR	APB bridge	Address. This is the APB address bus. It can be up to 32 bits wide and is driven by the peripheral bus bridge unit.
PPROT	APB bridge	Protection type. This signal indicates the normal, privileged, or secure protection level of the transaction and whether the transaction is a data access or an instruction access.
PSELx	APB bridge	Select. The APB bridge unit generates this signal to each peripheral bus slave. It indicates that the slave device is selected and that a data transfer is required. There is a **PSELx** signal for each slave.
PENABLE	APB bridge	Enable. This signal indicates the second and subsequent cycles of an APB transfer.
PWRITE	APB bridge	Direction. This signal indicates an APB write access when HIGH and an APB read access when LOW.
PWDATA	APB bridge	Write data. This bus is driven by the peripheral bus bridge unit during write cycles when **PWRITE** is HIGH. This bus can be up to 32 bits wide.
PSTRB	APB bridge	Write strobes. This signal indicates which byte lanes to update during a write transfer. There is one write strobe for each eight bits of the write data bus. Therefore, **PSTRB[n]** corresponds to **PWDATA[(8n + 7):(8n)]**. Write strobes must not be active during a read transfer.
PREADY	Slave interface	Ready. The slave uses this signal to extend an APB transfer.
PRDATA	Slave interface	Read Data. The selected slave drives this bus during read cycles when **PWRITE** is LOW. This bus can be up to 32-bits wide.
PSLVERR	Slave interface	This signal indicates a transfer failure. APB peripherals are not required to support the **PSLVERR** pin. This is true for both existing and new APB peripheral designs. Where a peripheral does not include this pin then the appropriate input to the APB bridge is tied LOW.

Recap: MMIO and APB

- Memory Mapped I/0
  - A way to interface with peripherals from the processor core
  - Peripherals and external devices are presented as memory and can be hardware registers or “virtual registers” via state machines
  - Use **load** and **store** instructions to access MMIO
  - The AHB and AHP transport data to-and-from the core and peripherals
- Advanced Peripheral Bus (APB)
  - Use APB to access MMIO
  - Bus structure uses address and data lines as well as control lines
  - Initiator sets PENABLE high when data on the bus is ready
  - Each Target device has its own read data bus (PRDATA) and sets PREADY high when data is ready

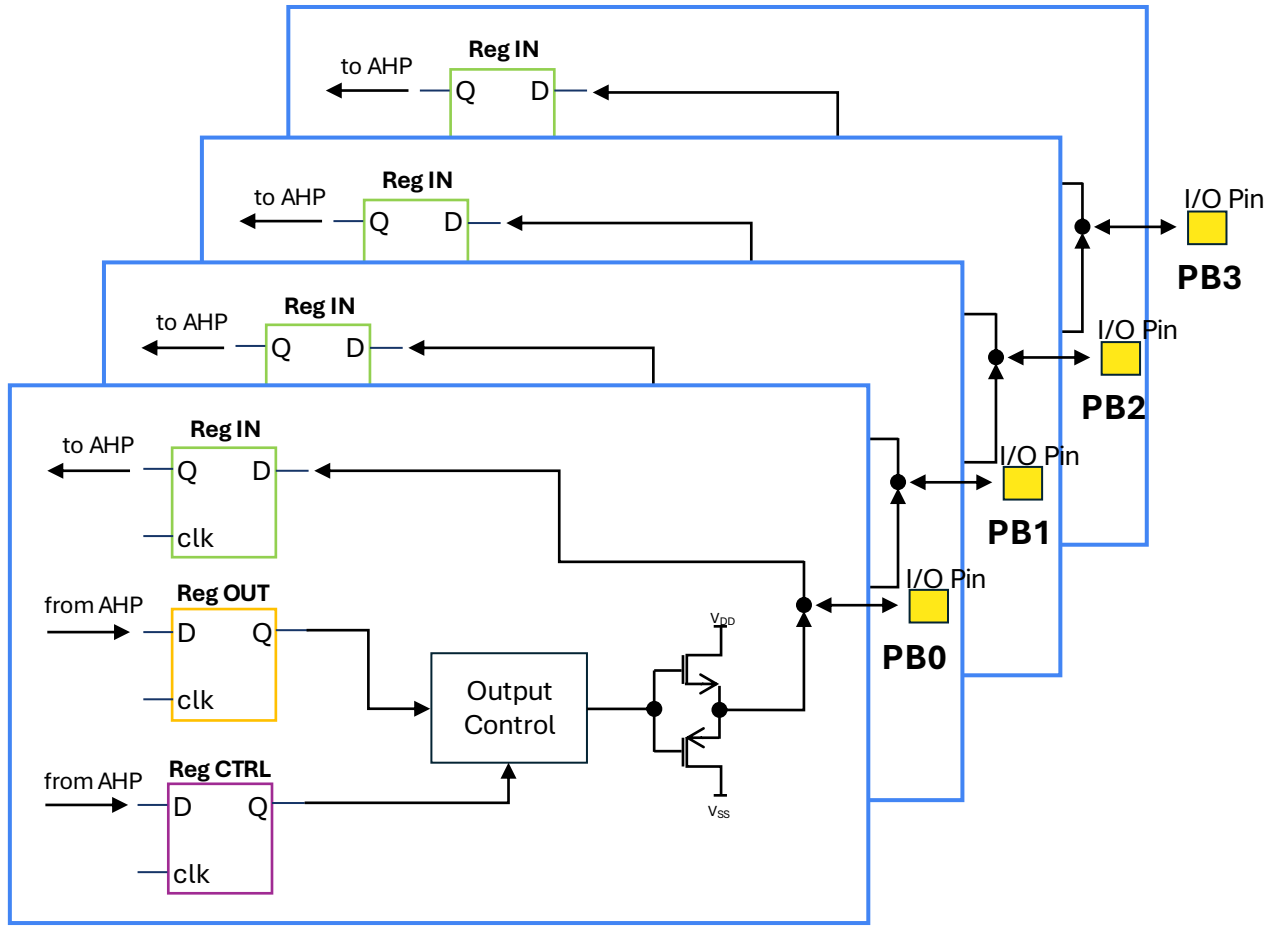
A deeper dive into GPIO

GPIO: General Purpose Input/Output

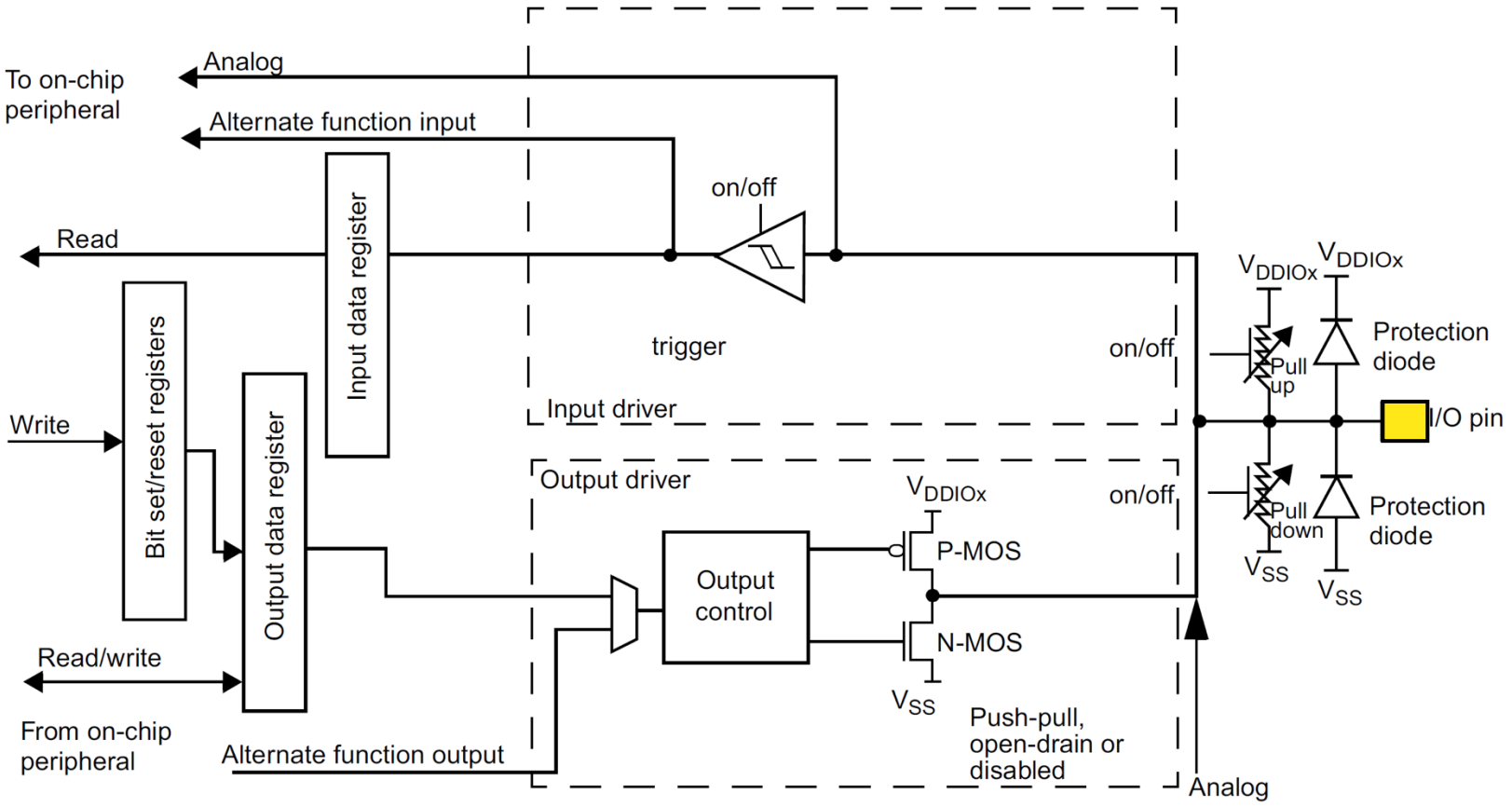


GPIO Port

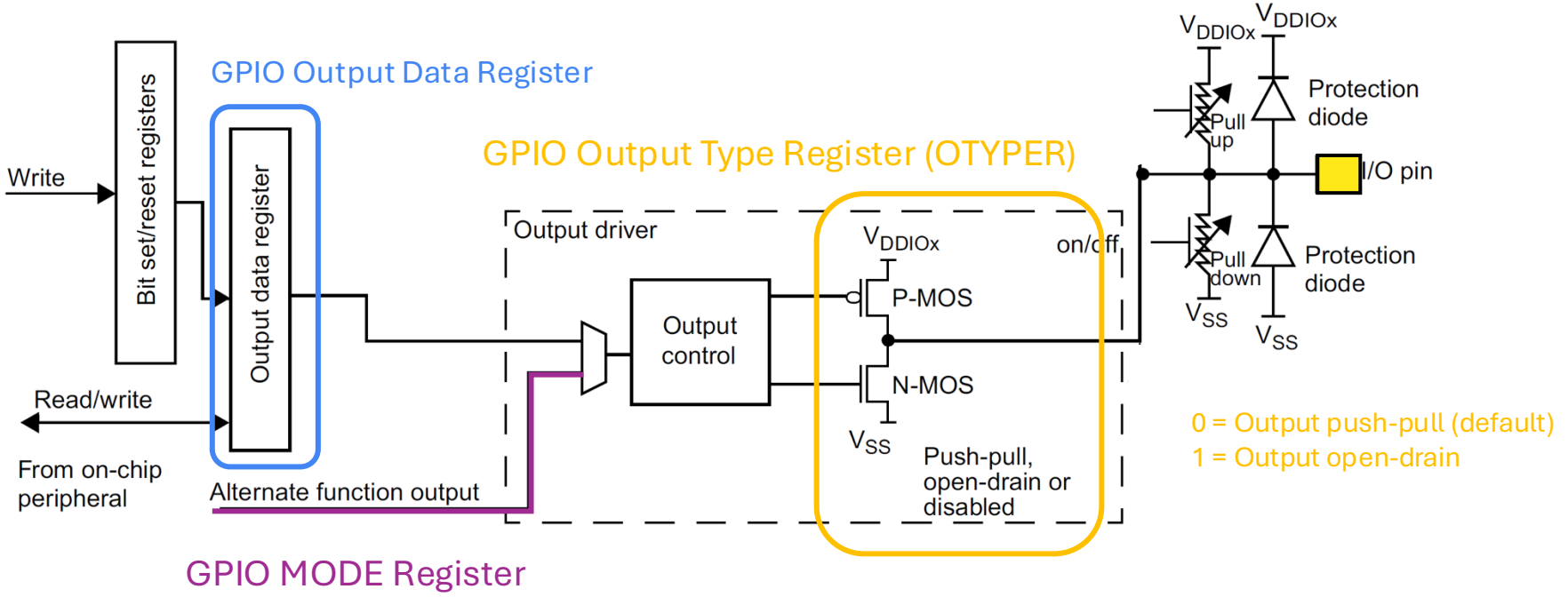
Address	Registers	
	Port B - Input	
0x033088	1	0 1 0
	Port B - Output	
0x03308C	0	0 1 0
	Port B - Control	
0x033080	0	0 1 1



Basic Structure of an I/O Port (Single Bit)



Basic Structure of an I/O Port Bit : Output



Enable Clock

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	RNG EN	Res.	AESEN
													rw		rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	ADCEN	OTGFS EN	Res.	Res.	Res.	Res.	GPIOH EN	GPIOG EN	GPIOF EN	GPIOE EN	GPIOD EN	GPIOC EN	GPIOB EN	GPIOA EN
		rw	rw					rw	rw	rw	rw	rw	rw	rw	rw

Bit 1 **GPIOBEN:** IO port B clock enable

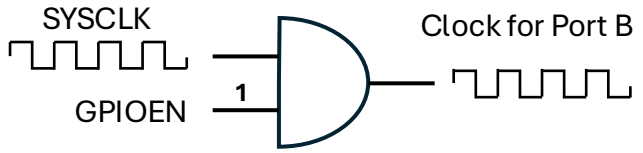
Set and cleared by software.

0: IO port B clock disabled

1: IO port B clock enabled

```
# define RCC+AHB2ENR_GPIOBEN ((uint32_t)0x0000002U

RCC->AHB2ENR |= RCC_AHB2ENR_GPIOEN;
```



GPIO Mode Register (MODER)

32 bits (16 pins, 2 bits per pin)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODE15[1:0]		MODE14[1:0]		MODE13[1:0]		MODE12[1:0]		MODE11[1:0]		MODE10[1:0]		MODE9[1:0]		MODE8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODE7[1:0]		MODE6[1:0]		MODE5[1:0]		MODE4[1:0]		MODE3[1:0]		MODE2[1:0]		MODE1[1:0]		MODE0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

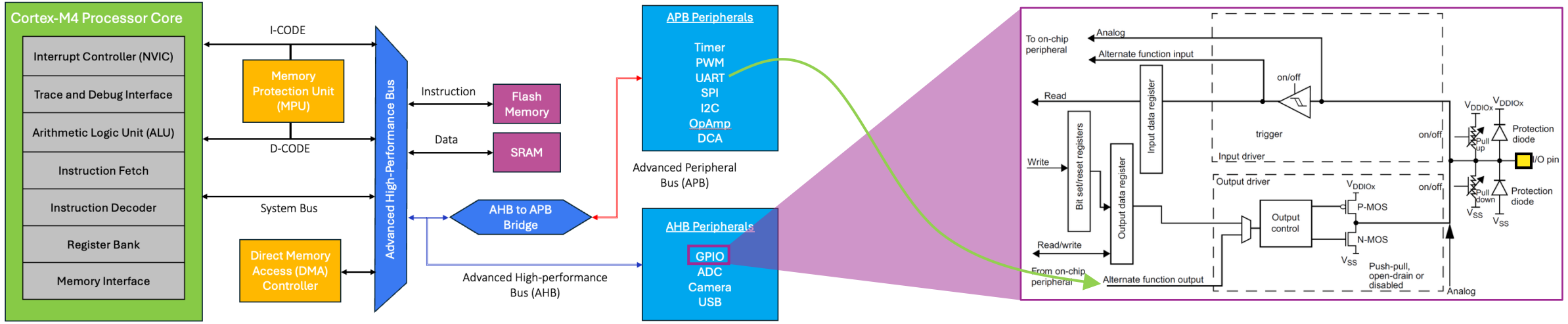
Pin 2 Pin 1 Pin 0

Bits 2y+1:2y **MODEy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O mode.
00: Input mode
01: General purpose output mode
10: Alternate function mode
11: Analog mode (reset state)

```
GPIOB→MODER &= ~(3UL<<4);     // Clear bits 4 and 5 for Pin 2
GPIOB→MODER |= 1UL<<4;        // Set bit 4, set Pin 2 as output
```

Example: Alternative Functions

Alternative functions could be UART, SPI, I2C, etc.



GPIO Output Type Register (OTYPE)

16 bits reserved, 16 data bits, 1 bit for each pin

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

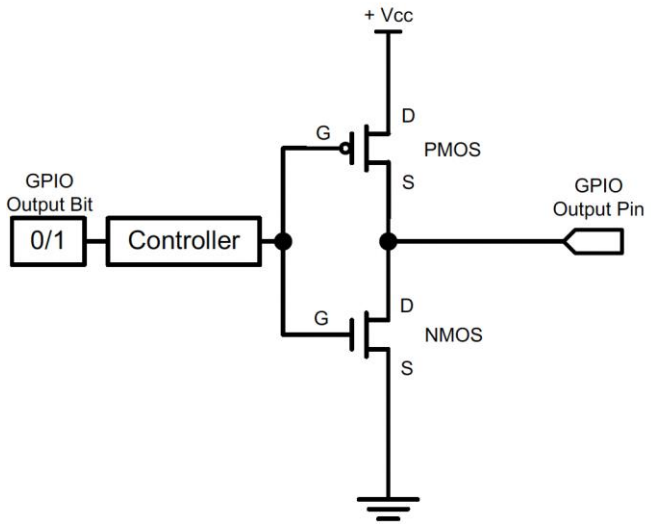
Bits 15:0 **OTy:** Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O output type.

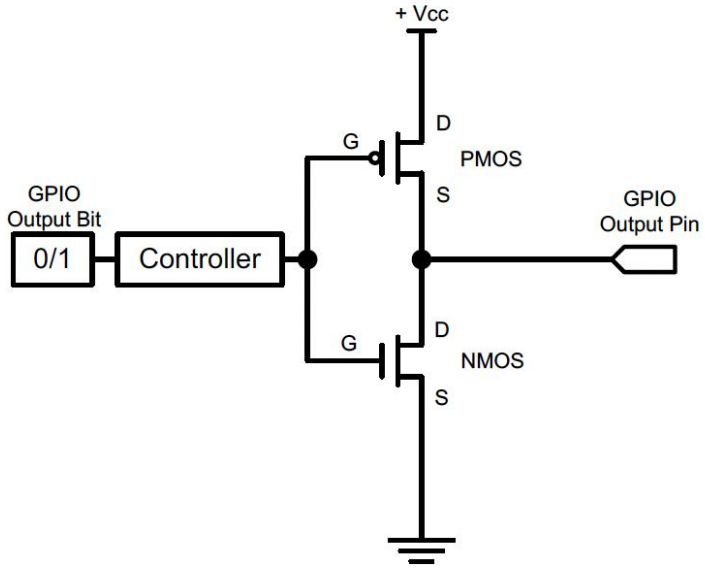
0: Output push-pull (reset state)

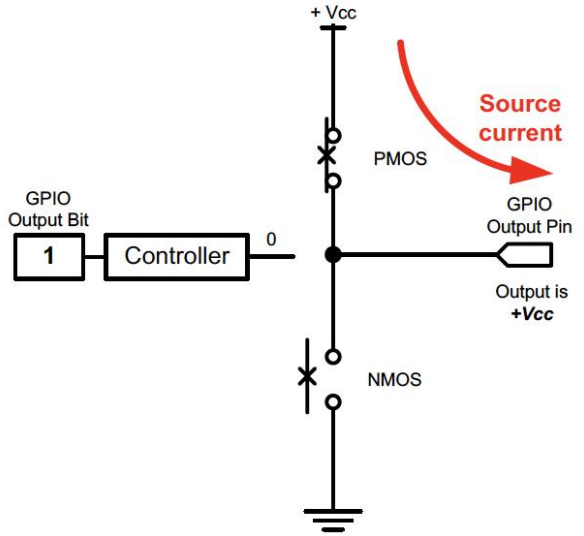
1: Output open-drain

```
GPIOB→OTYPE &= ~(1UL<<2);     // Clear bit 2
```

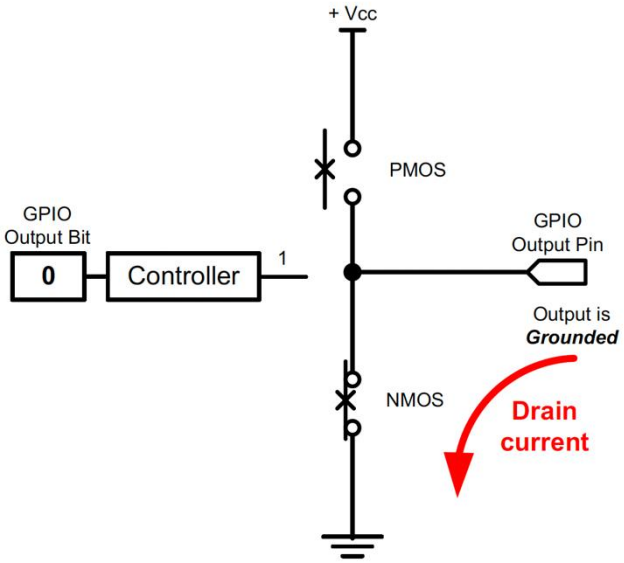


GPIO Output: Push-pull



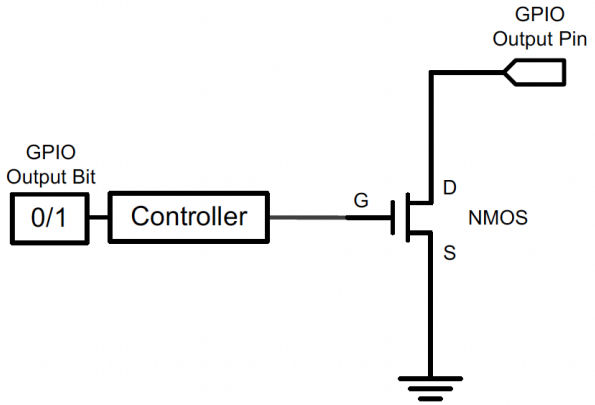


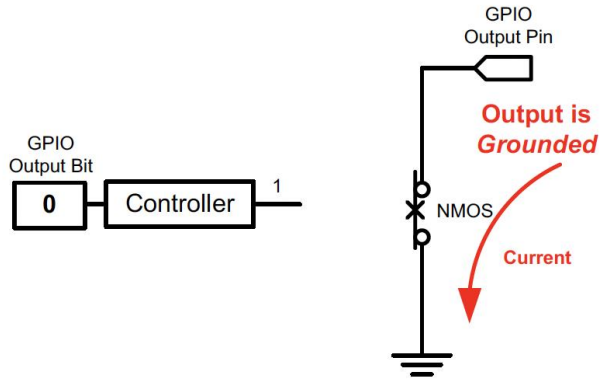
GPIO Output = 1
Source current to external circuit



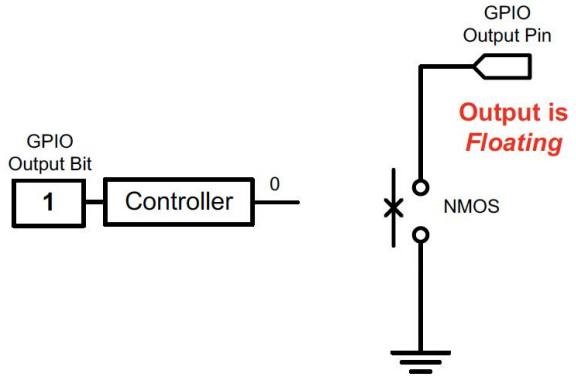
GPIO Output = 0
Drain current from external circuit

GPIO: Open Drain





GPIO Output =0
Drain current from external circuit



GPIO Output = 1
GPIO Pin has high-impedance to external circuit

GPIO Output Speed

Configure the GPIO output speed

Output Speed:

- Speed of rising and falling
- Four speeds – Low, Medium, Fast, High

Tradeoffs:

- Higher GPIO speed increases EMI noise and power consumption
- Configure based on peripheral speed
  - Low speed for toggling LEDs
  - High speed for SPI

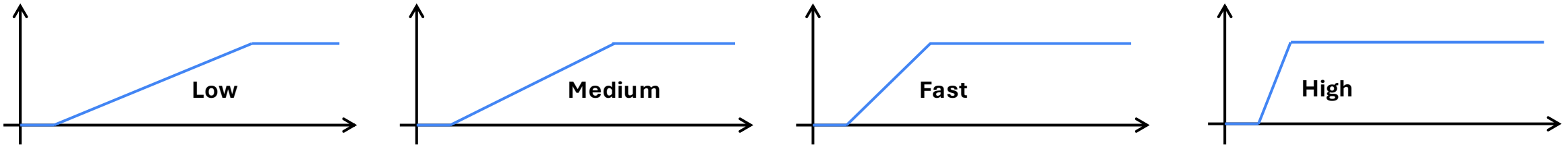
8.4.3 GPIO port output speed register (GPIOx_OSPEEDR) (x = A to I)

Address offset: 0x08

Reset value: 0x0C00 0000 (for port A)

Reset value: 0x0000 0000 (for the other ports)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
OSPEED15 [1:0]		OSPEED14 [1:0]		OSPEED13 [1:0]		OSPEED12 [1:0]		OSPEED11 [1:0]		OSPEED10 [1:0]		OSPEED9 [1:0]		OSPEED8 [1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OSPEED7 [1:0]		OSPEED6 [1:0]		OSPEED5 [1:0]		OSPEED4 [1:0]		OSPEED3 [1:0]		OSPEED2 [1:0]		OSPEED1 [1:0]		OSPEED0 [1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw



GPIO Output Data Register (ODR)

16 bits reserved, 16 data bits, 1 bit for each pin

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OD15	OD14	OD13	OD12	OD11	OD10	OD9	OD8	OD7	OD6	OD5	OD4	OD3	OD2	OD1	OD0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

```
GPIOB→ODR |= 1UL << 2;          \\ Set bit 2
```

Example: Set Port B, Pin 2 (PB.2) to HIGH

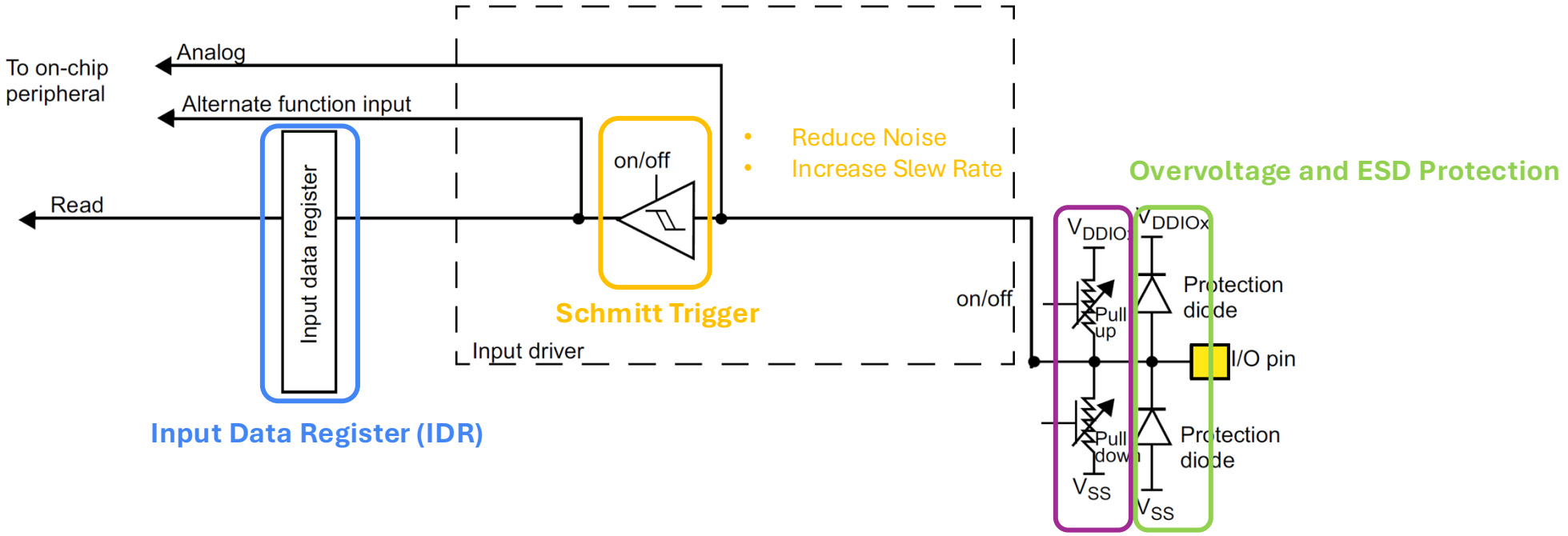
```
RCC->AHB2ENR |= RCC_AHB2ENR_GPIOEN;   \\ Enable clock of Port B

GPIO->MODER &= ~(3UL<<4);             \\ Clear mode bits
GPIO->MODER |= 1UL<<4;                \\ Set mode to output

GPIO->OTYPE &= ~(1UL<<2)              \\ Select push-pull output

GPIO->ODR |= 1UL << 2;                \\ Output logical “1” to drive PB. high
```

Basic Structure of an I/O Port Bit : Output



GPIO Pull-up/Pull-down Register (PUPDR)

00 = No pull-up, pull-down

10 = Pull-down

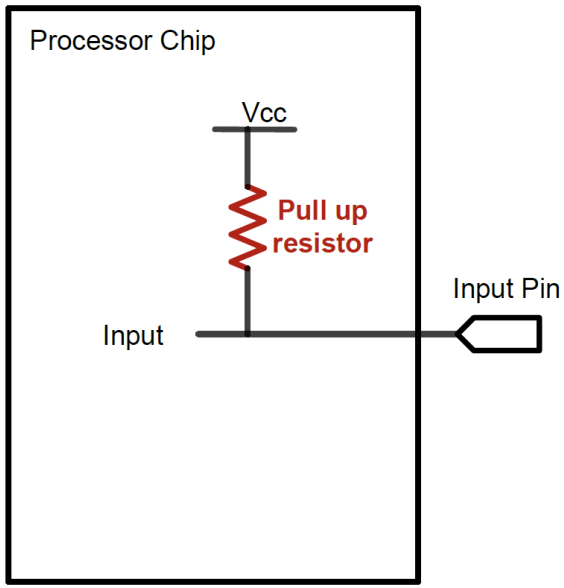
01 = Pull-up

11 = Reserved

GPIO Input: Pull up and Pull Down

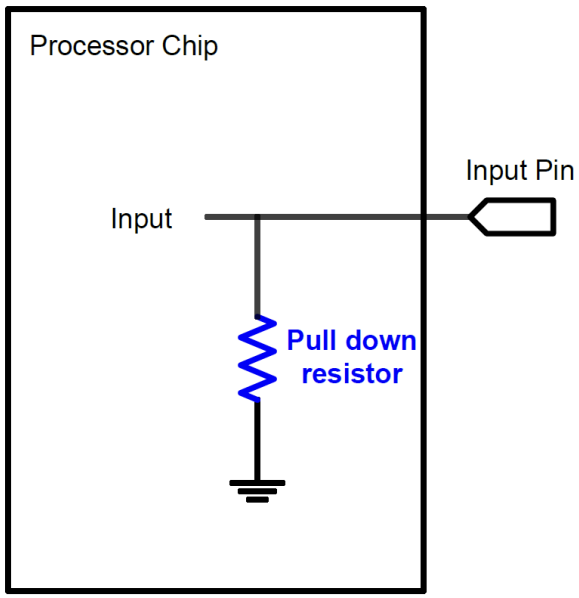
Digital input can have three states: HIGH, LOW, and High-Impedance

Pull-up



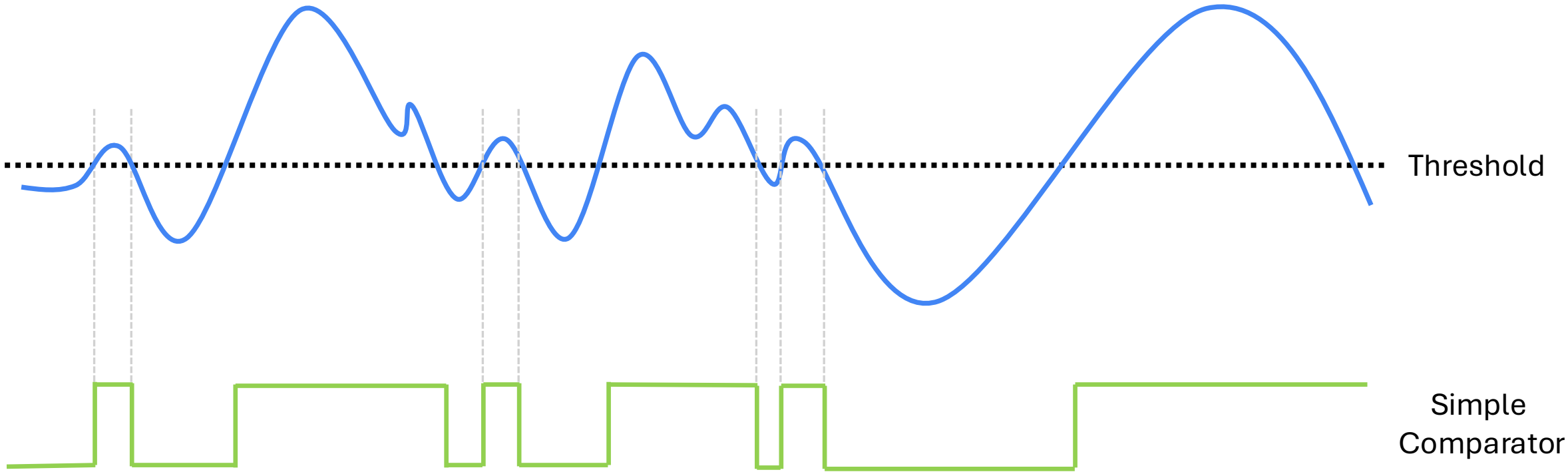
If the external input is HiZ, the input is read as HIGH

Pull-down

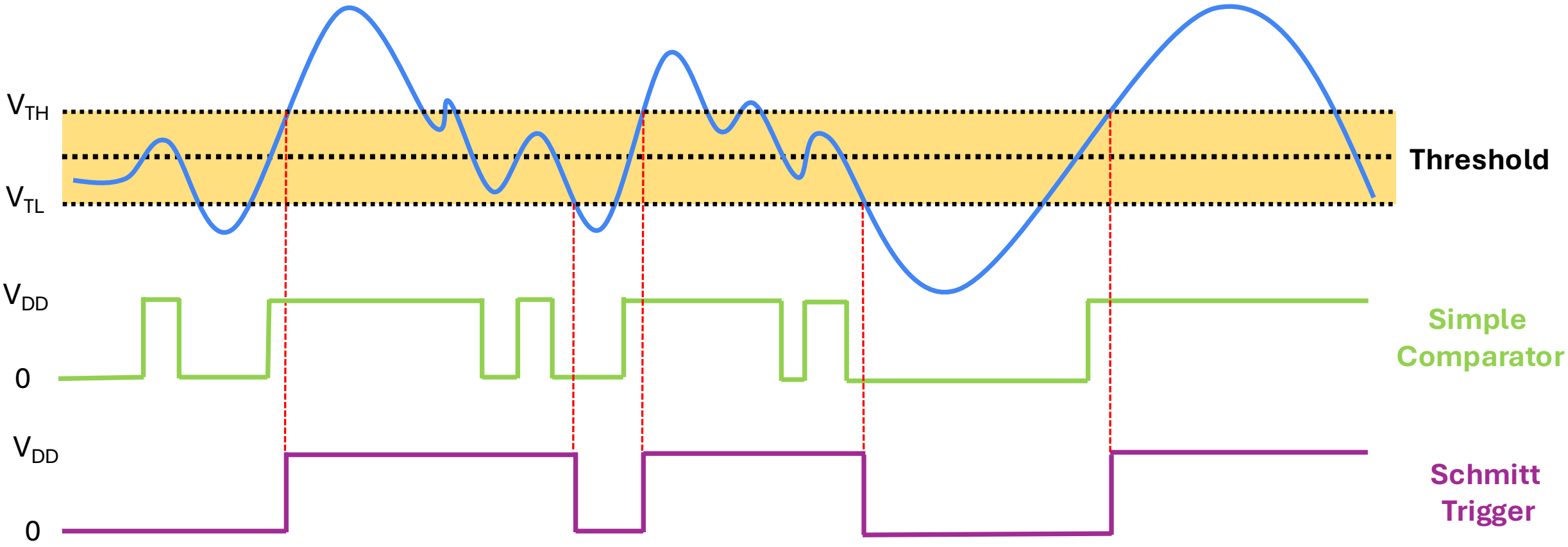


If the external input is HiZ, the input is read as LOW

Noisy Digital Signals

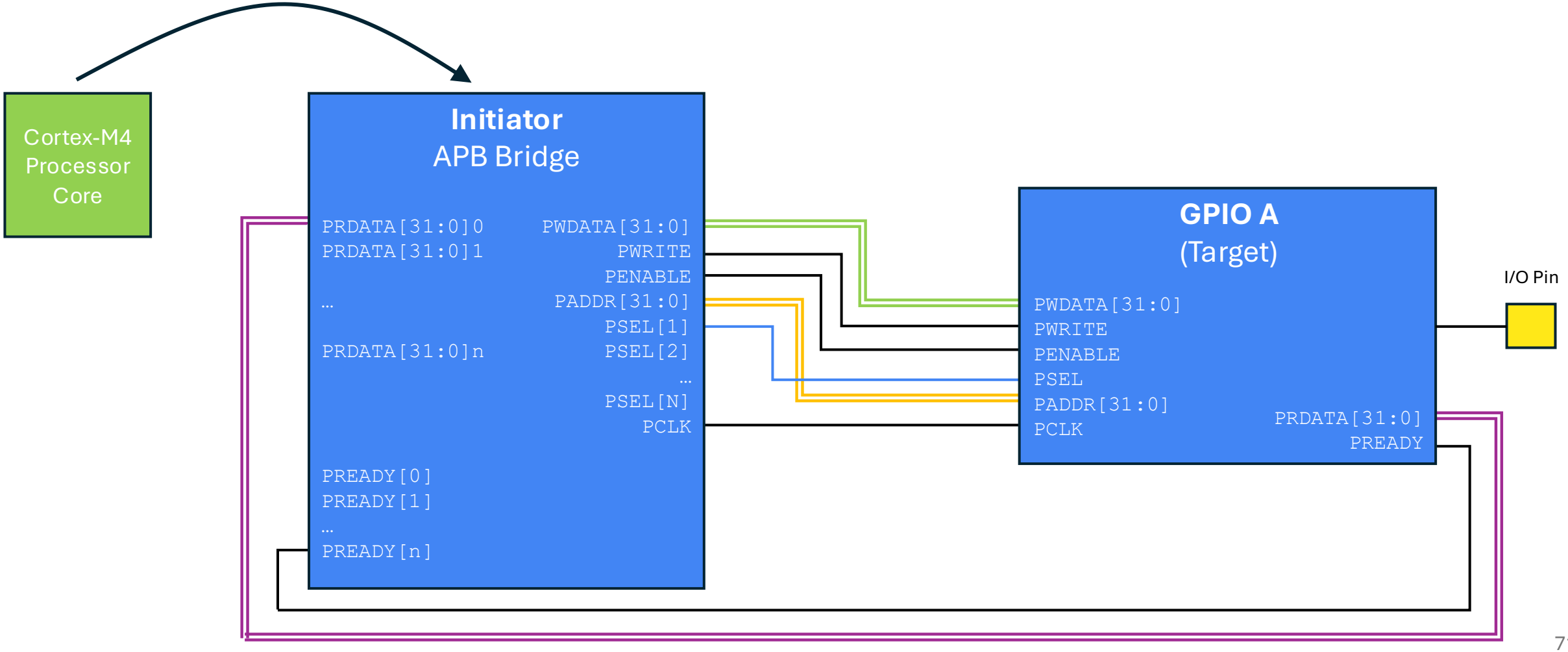


Schmitt Trigger



Example: Create an APB GPIO Peripheral

We are going to assume that GPIO is part of the APB rather than AHB



Group Exercise:

- Wire this ABP peripheral to the ABP Bus to enable full GPIO read/write functionality
- *Assume everything is ready in one clock cycle*
- *Assume GPIO_IN is updating every clock cycle*

PWDATA[31:0] →

PRDATA[31:0] ←

PWRITE →

PENABLE →

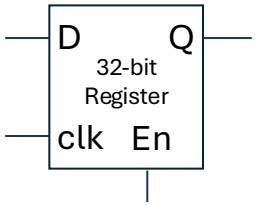
PSEL →

PADDR[31:0] →

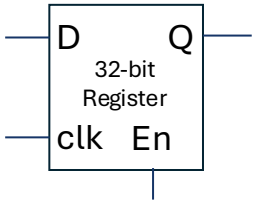
PCLK →

PREADY ←

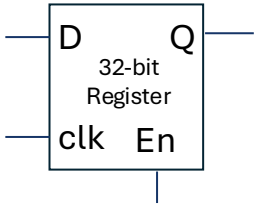
GPIO_IN

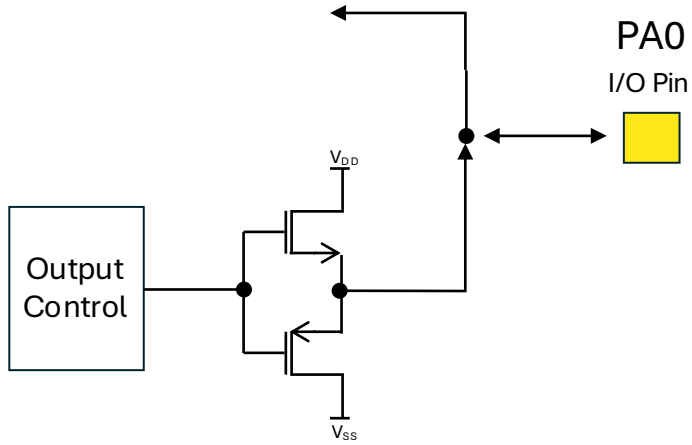


GPIO_ODR



GPIO_MODER





Recap

GPIO – General Purpose Input / Output

- On chip hardware that controls the configurations of “pins”
- Configure to output or capture input

MMIO – Memory Mapped Input / Output

- A means to interface with peripherals (including GPIO) from the processor core using Loads and Stores
- Use “memory mapped” registers to configure GPIO pins
- Use “memory mapped” registers to read and write from pin

Acknowledgements

- These lecture slides were adapted from the University of Michigan's EECS 373 course (many thanks to Alanson Sample!)

- These lecture slides also leverage materials from the following books:
  - Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C
    - Yifeng Zhu
  - The Definitive Guide to ARM Cortex – M3 and Cortex – M4 Processors
    - Joseph Yiu
  - Introduction to ARM Cortex – M Microcontrollers, Embedded Systems
    - Jonathan W. Valvano