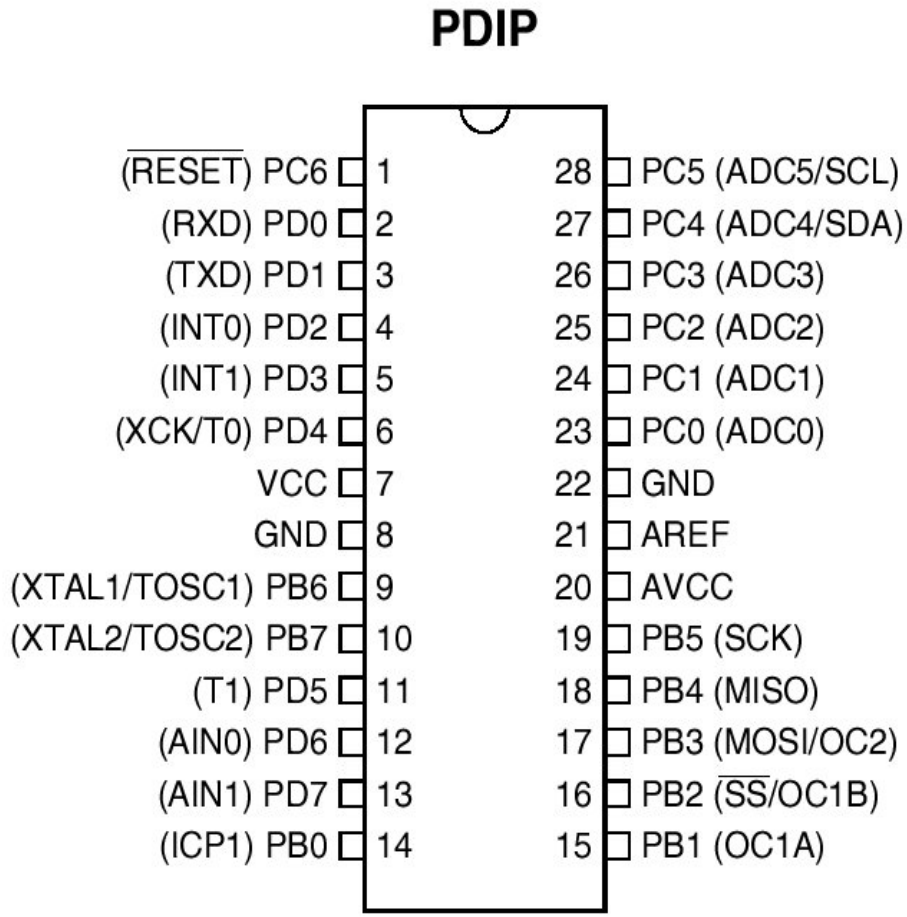
# Atmel Mega8 Basics

- Complete, stand-alone computer
- Ours is a 28-pin package
- Most pins:
  - Are used for input/output
  - How they are used is configurable



# Key Features

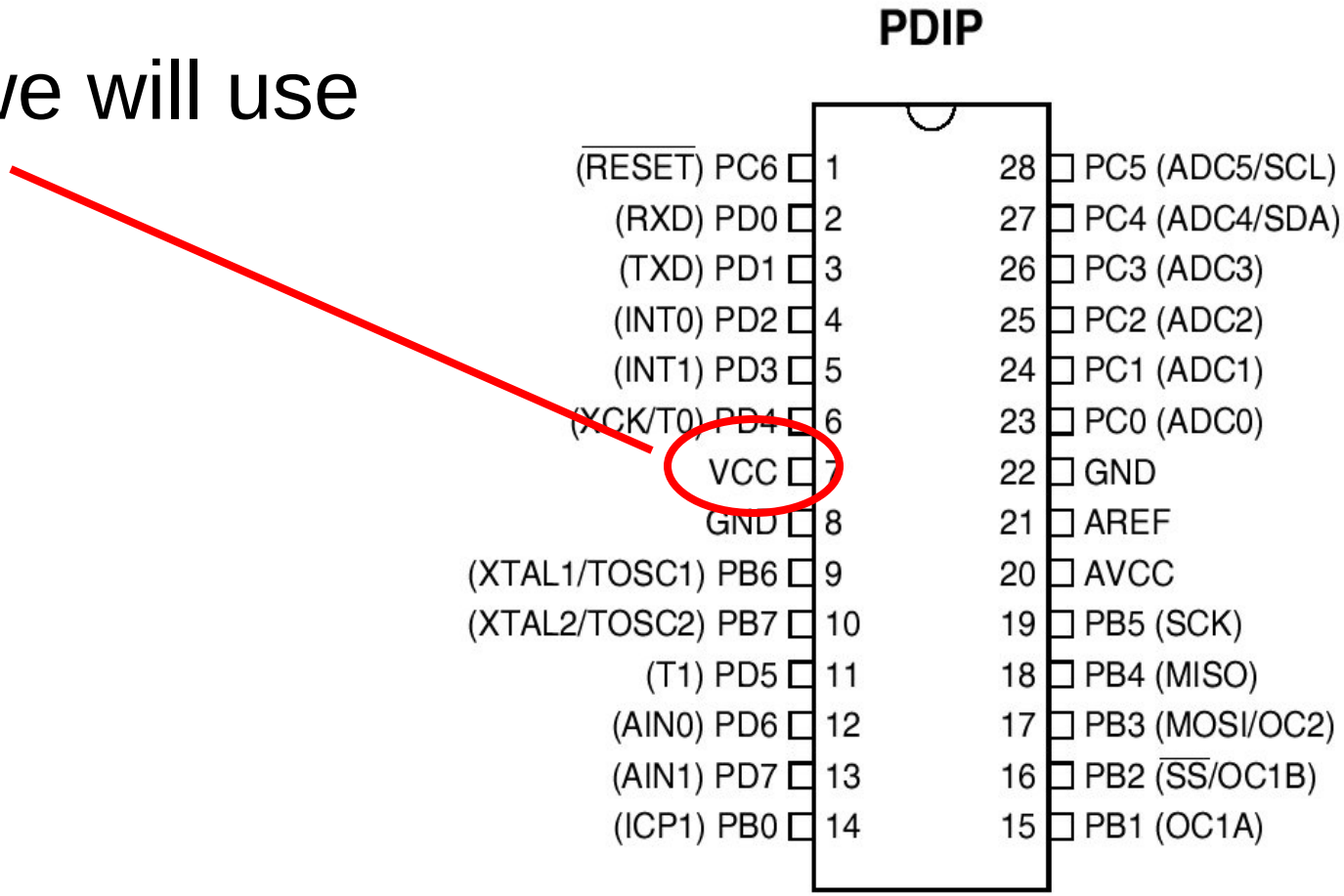
- Up to 16 MIPS (single cycle for most instructions)
- ~23 digital pins: configurable as inputs or outputs
- 6 channel, 10-bit analog-to-digital converter
- Serial communication support: RS232, SPI, I2C
- 3 counter/timers (2 8-bit; 1 16-bit)
- Internal/external interrupt support
- Brown-out detection
- Internal oscillator (1 MHz)
- Bootloader support
- Sleep mode
- Watchdog timer

# Interrupt Sources

- External pins: state change; falling/rising edge
- Timer/counters: when counter overflows
- Communication peripherals
- Brown out
- Analog to digital conversion complete

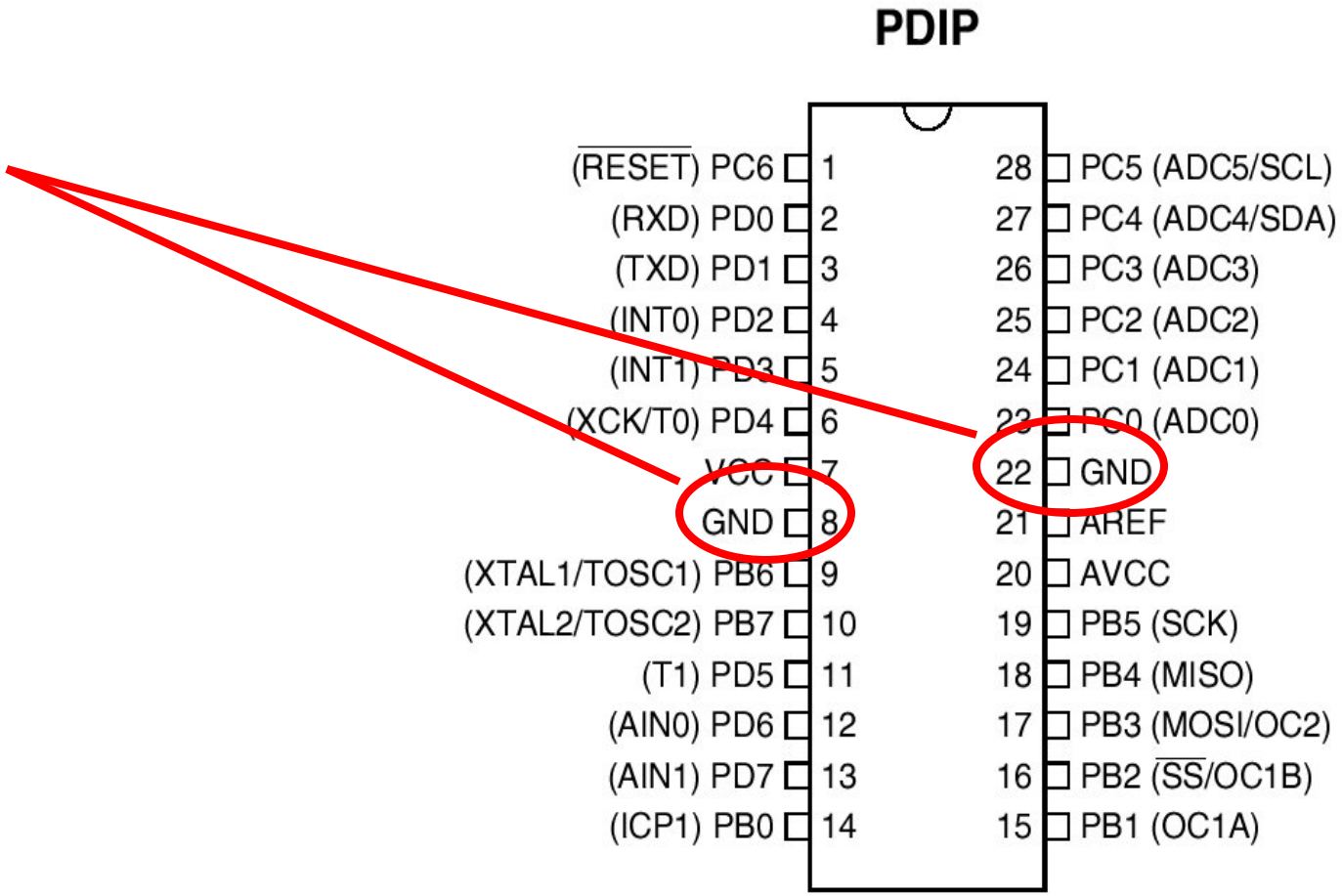
# Atmel Mega8 Basics

Power (we will use +5V)



# Atmel Mega8 Basics

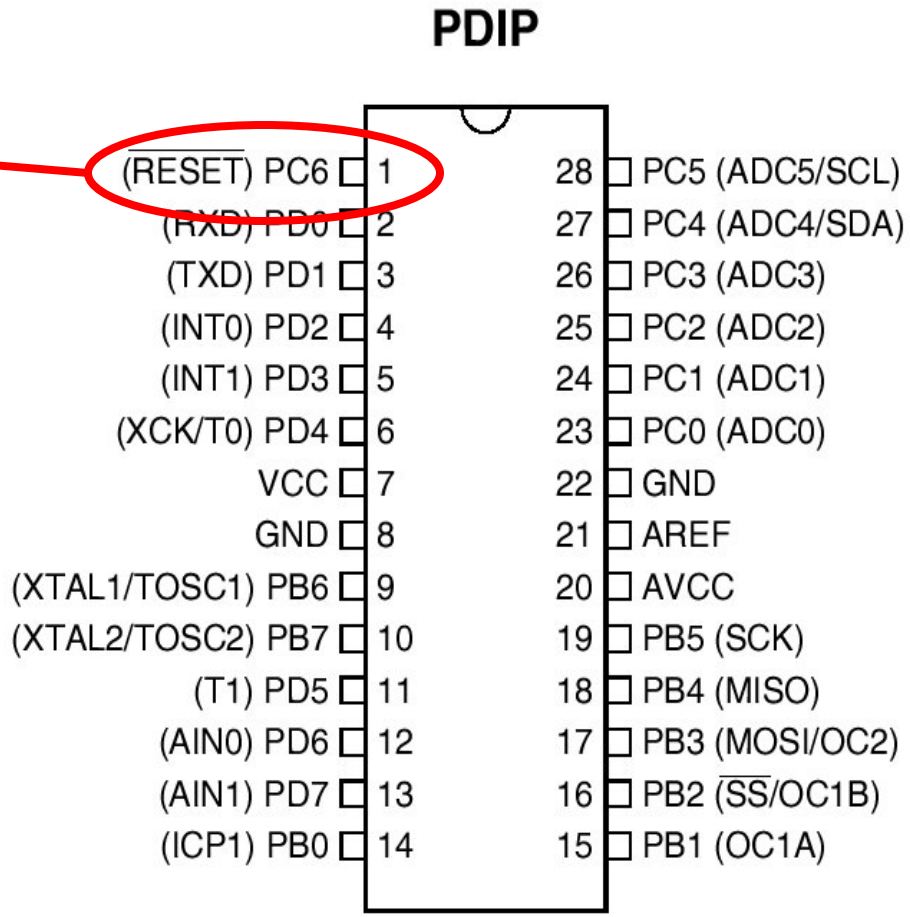
Ground



# Atmel Mega8 Basics

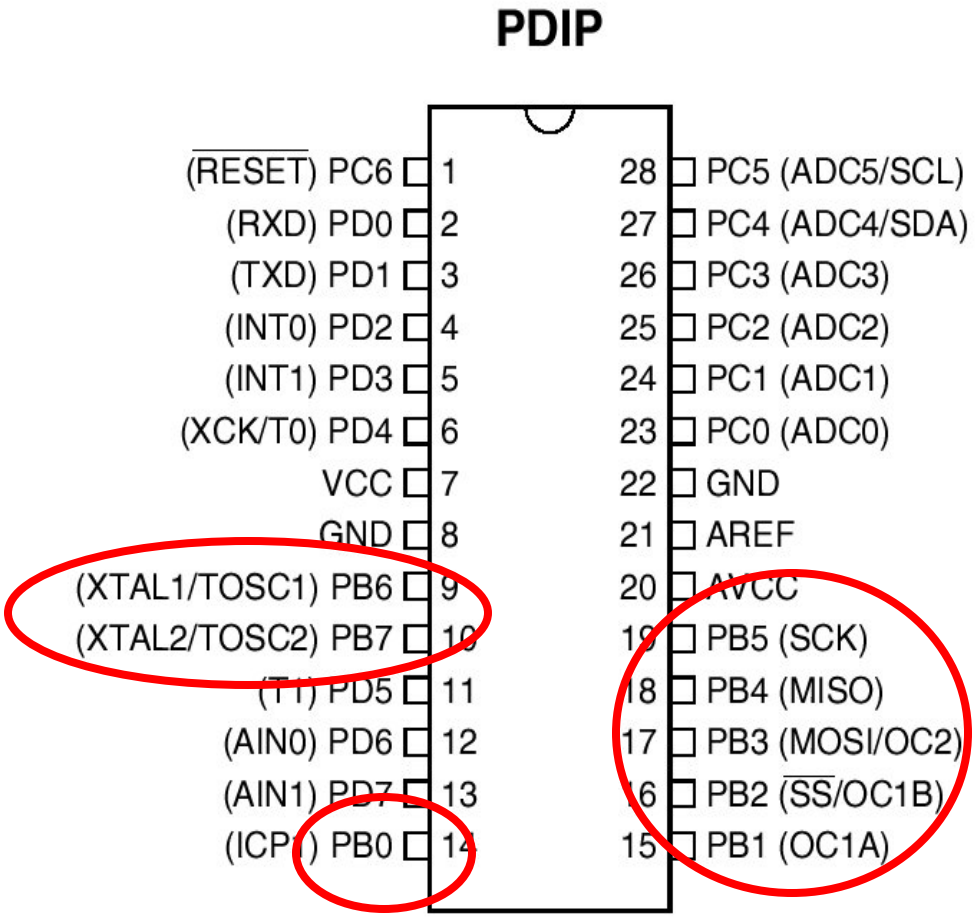
Reset

- Bring low to reset the processor
- In general, we will tie this pin to high through a pull-up resistor (10K ohm)



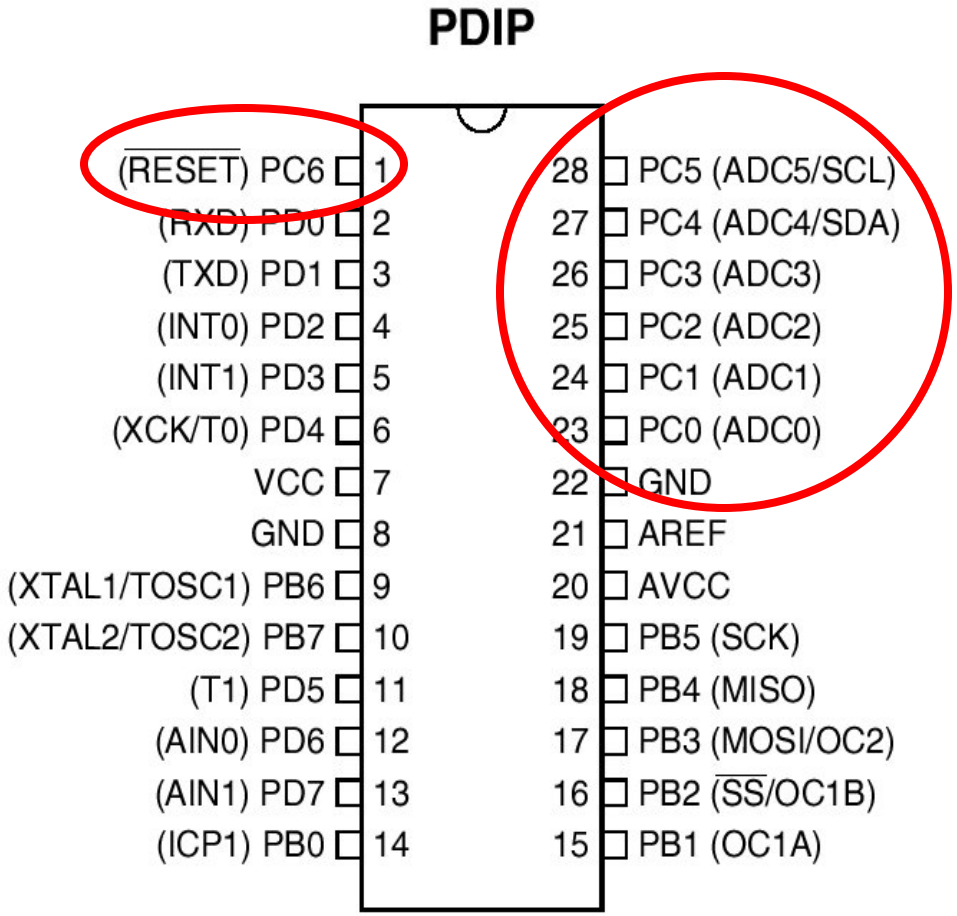
# Atmel Mega8 Basics

PORT B



# Atmel Mega8 Basics

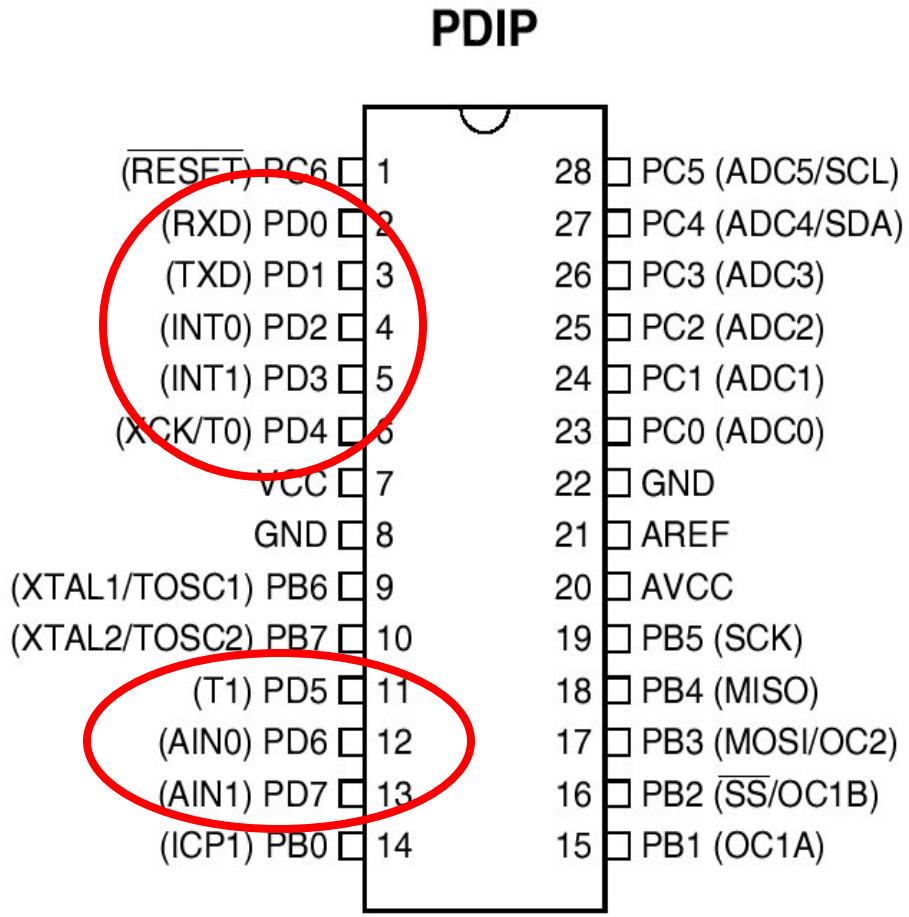
PORT C



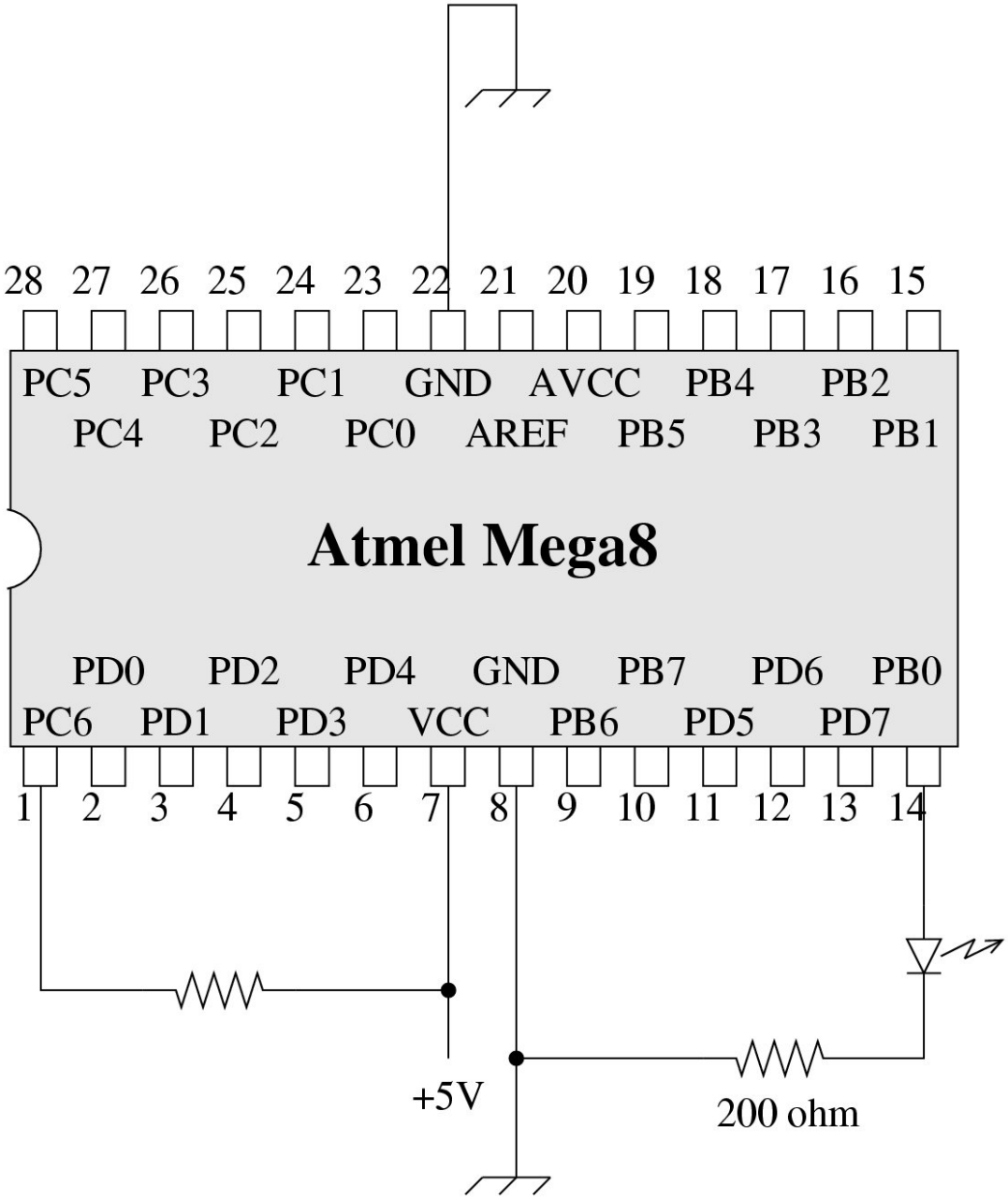
# Atmel Mega8 Basics

PORT D

(all 8 bits are available)



# A First Circuit



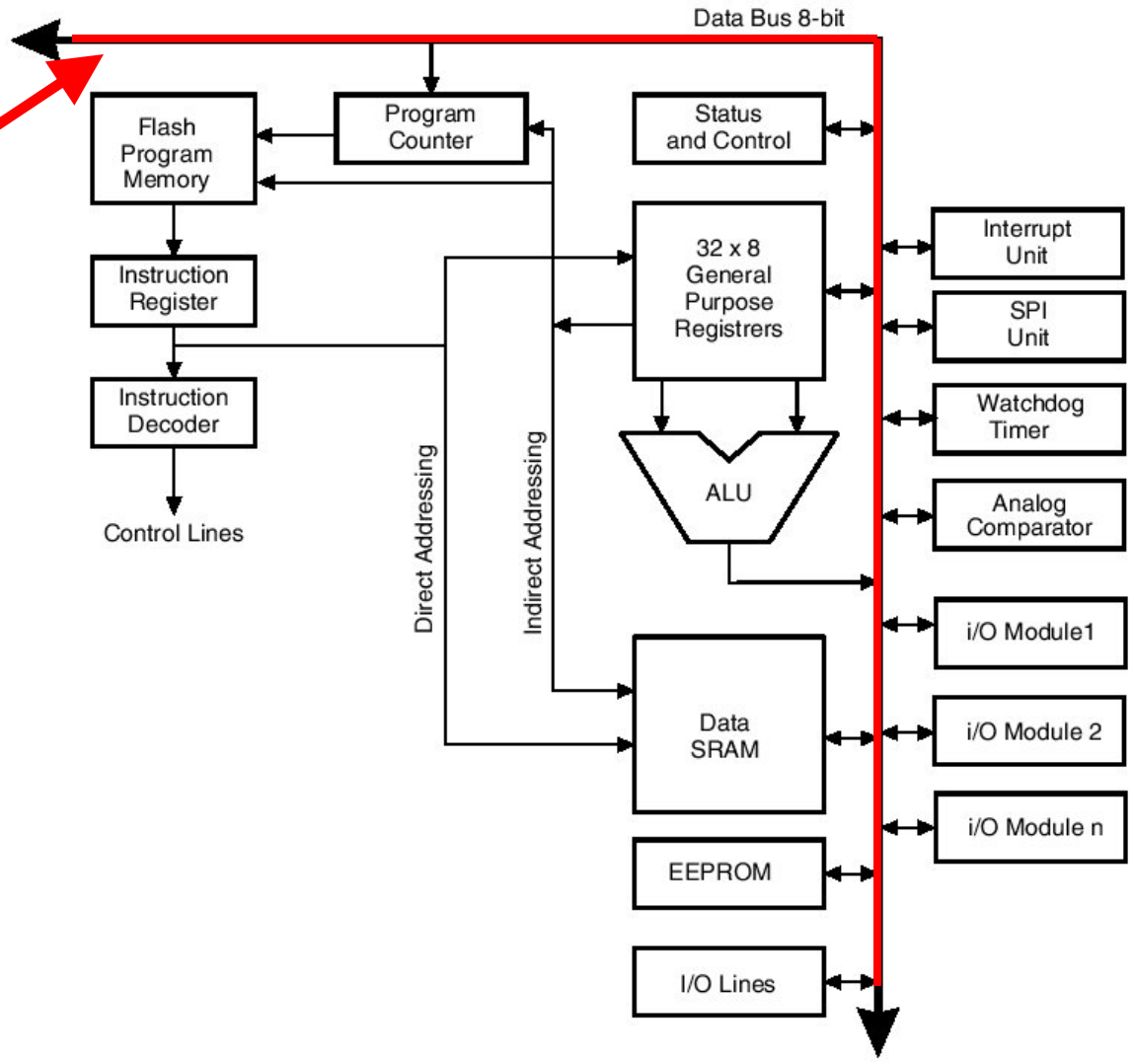
# Common Special-Purpose Registers

- Program counter
- Status register
- Instruction register
- Stack pointer
- Peripheral control is all done through registers

# Atmel Mega8

8-bit data bus

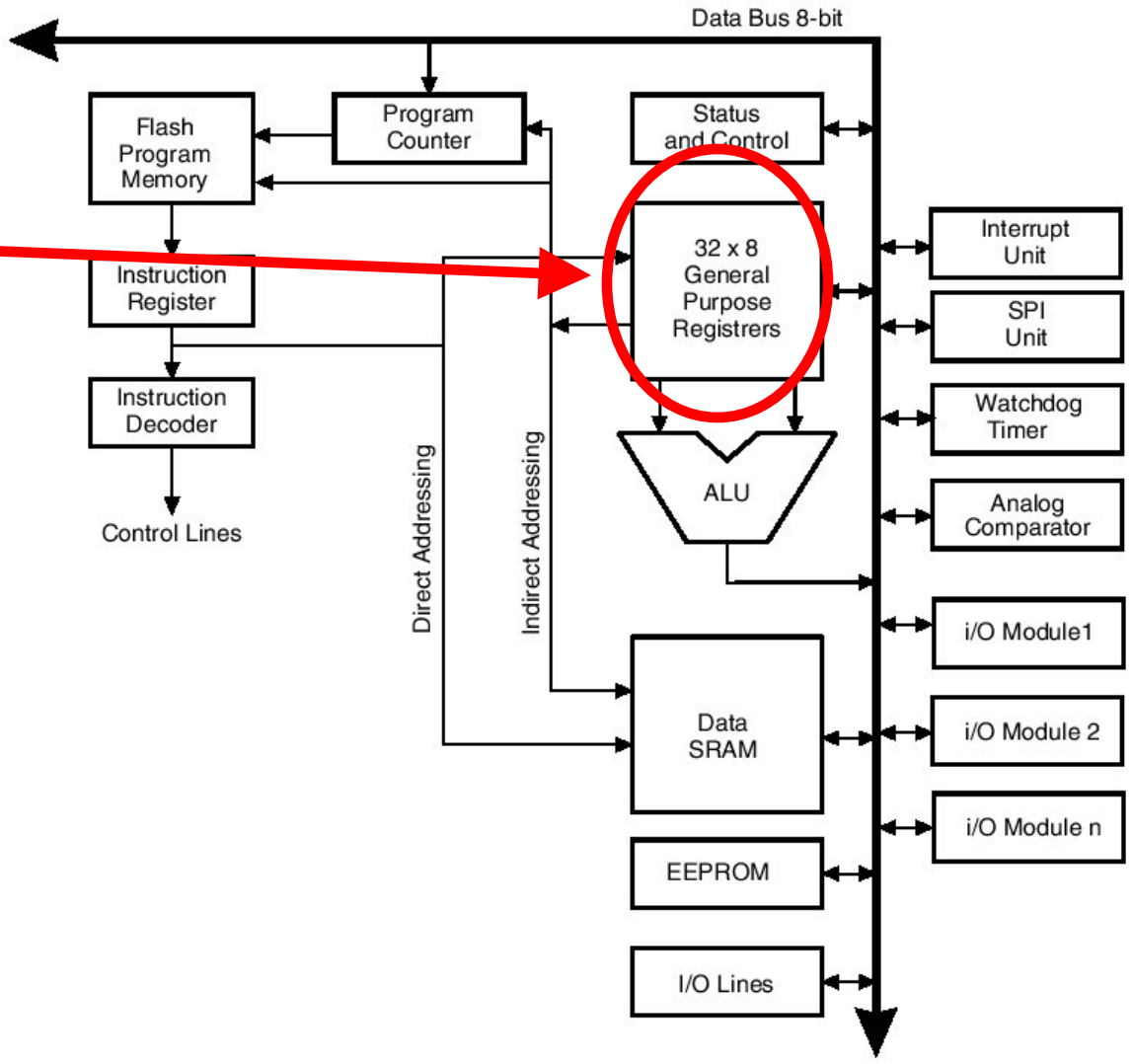
- Primary mechanism for data exchange



# Atmel Mega8

32 general purpose registers

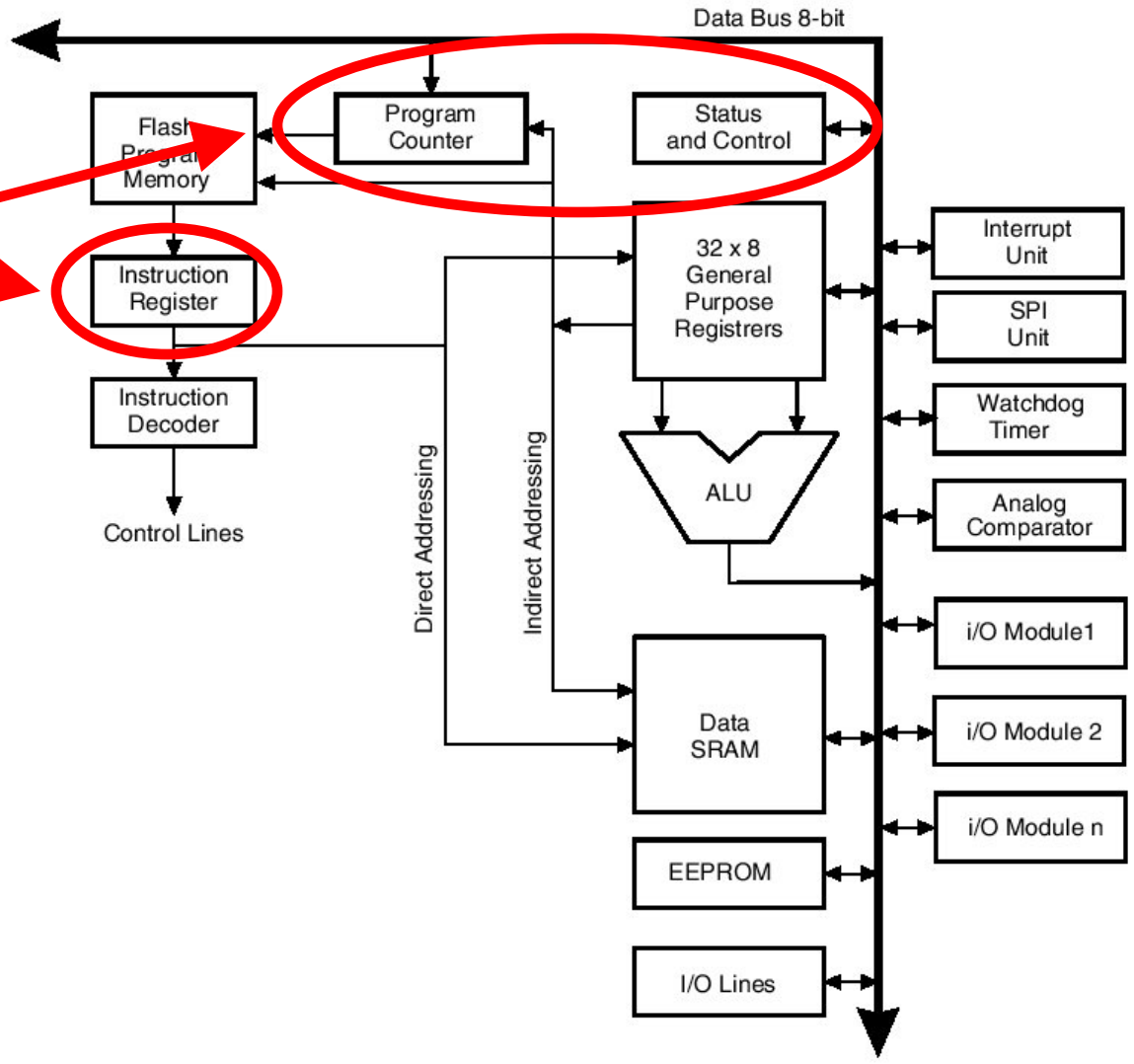
- 8 bits wide
- 3 pairs of registers can be combined to give us 16 bit registers



# Atmel Mega8

Special purpose registers

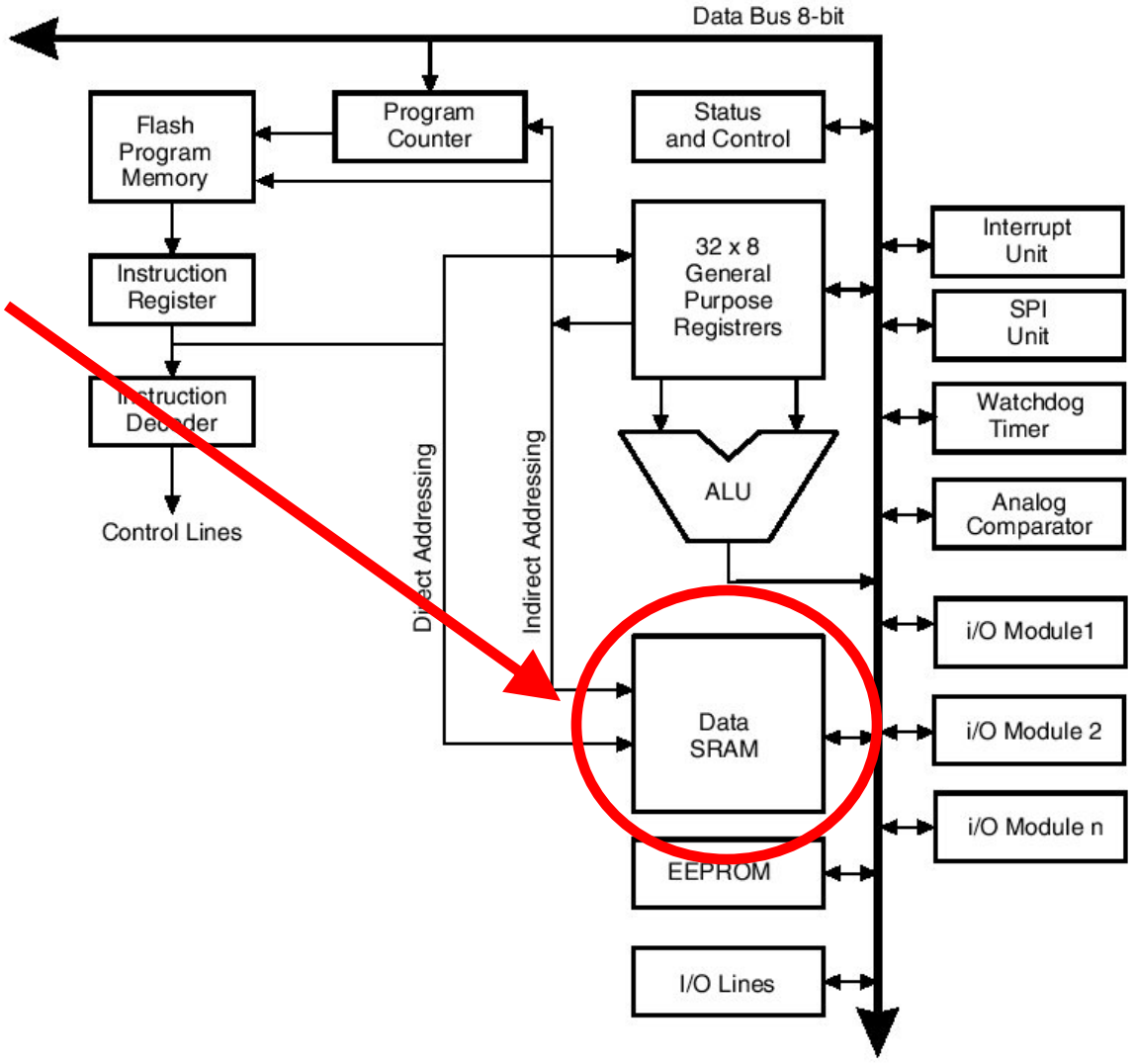
- Control of the internals of the processor



# Atmel Mega8

Random Access Memory (RAM)

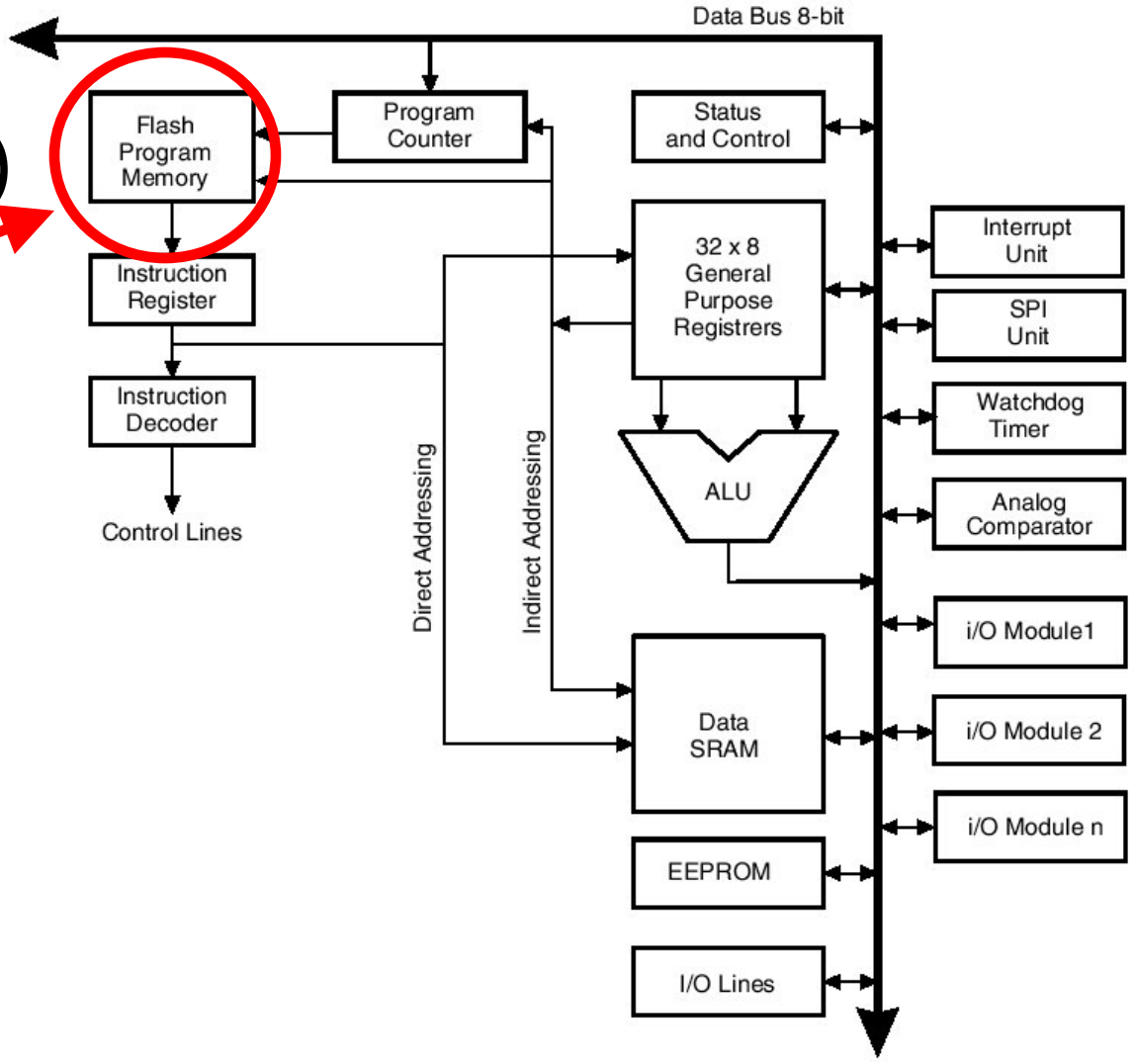
- 1 KByte in size
- Globals, heap and stack are stored here



# Atmel Mega8

Flash (EEPROM)

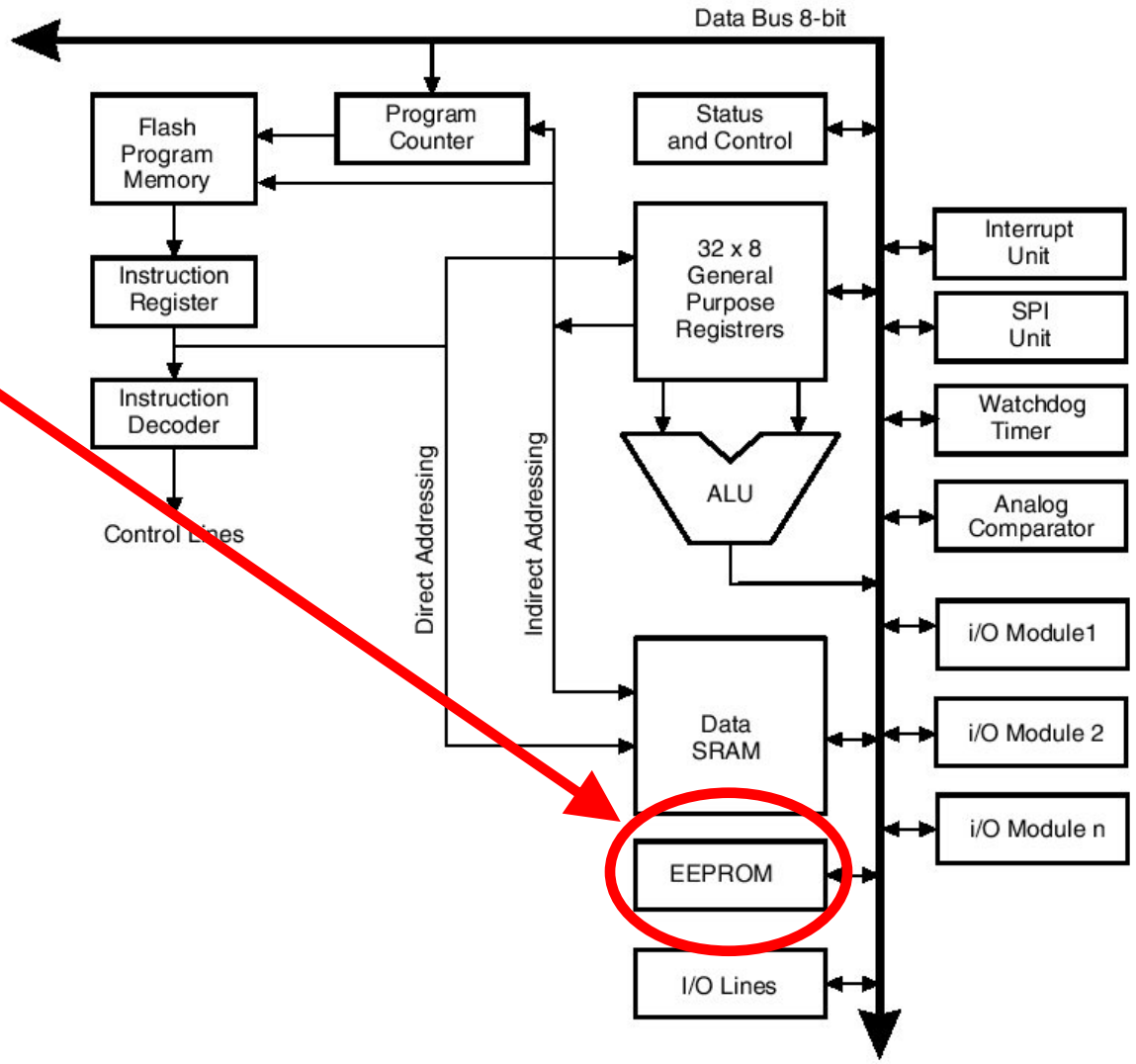
- Program storage
- 8 KByte in size
- 16 bit words



# Atmel Mega8

## EEPROM

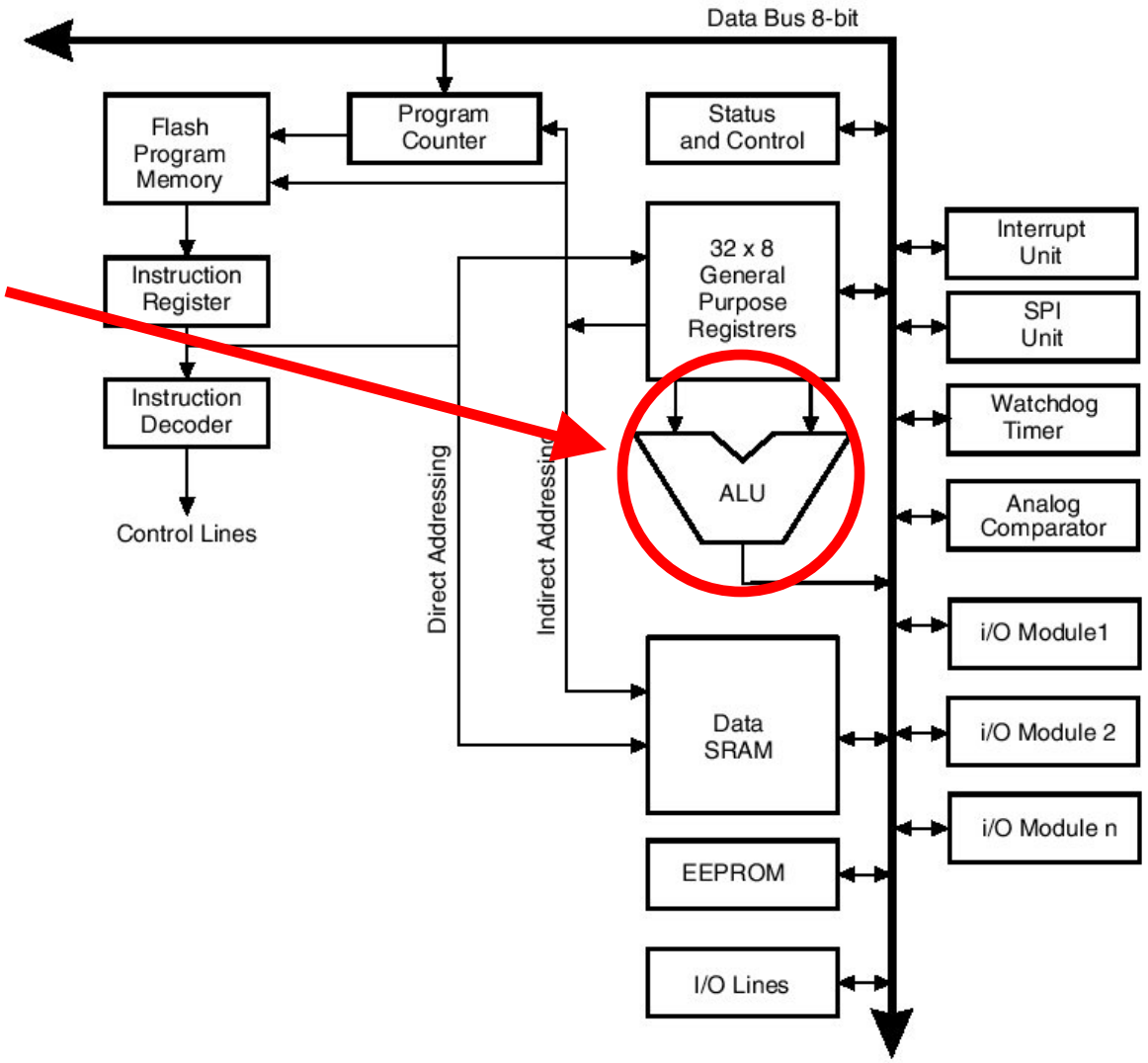
- Permanent data storage



# Atmel Mega8

Arithmetic Logical Unit

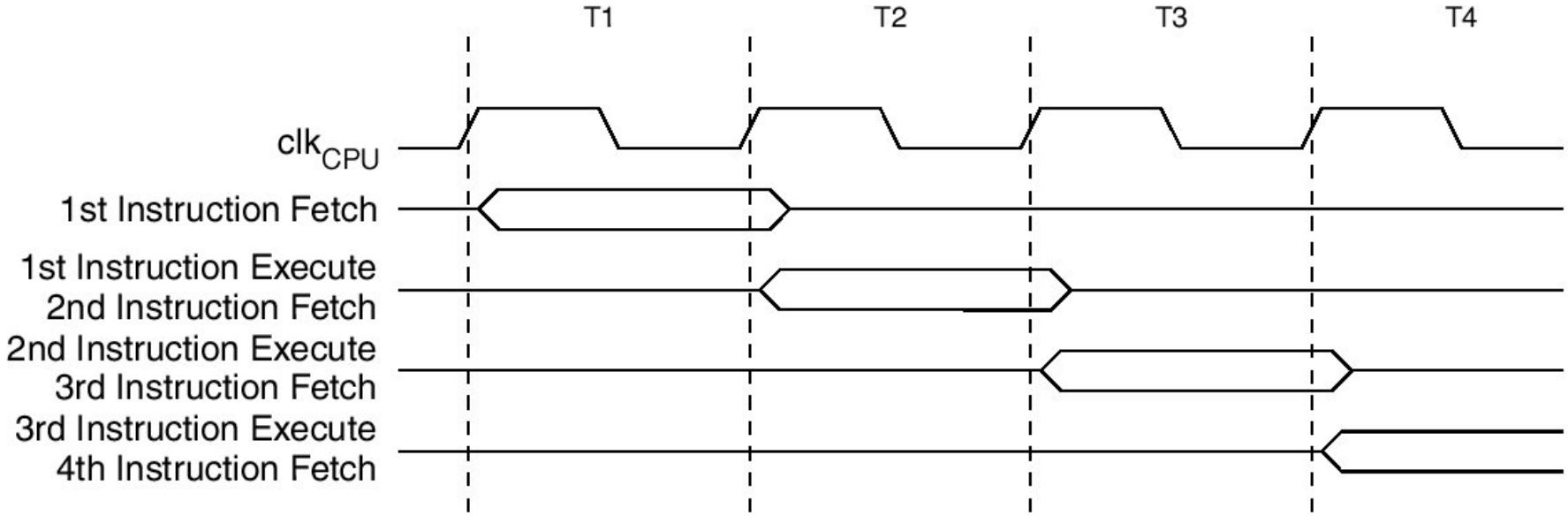
- Data inputs from registers
- Control inputs not shown (derived from instruction decoder)



# Processors in the Atmel Family

- Memory/program size
- Different numbers and types of I/O pins
- Custom support for other communication protocols (e.g., CANbus)

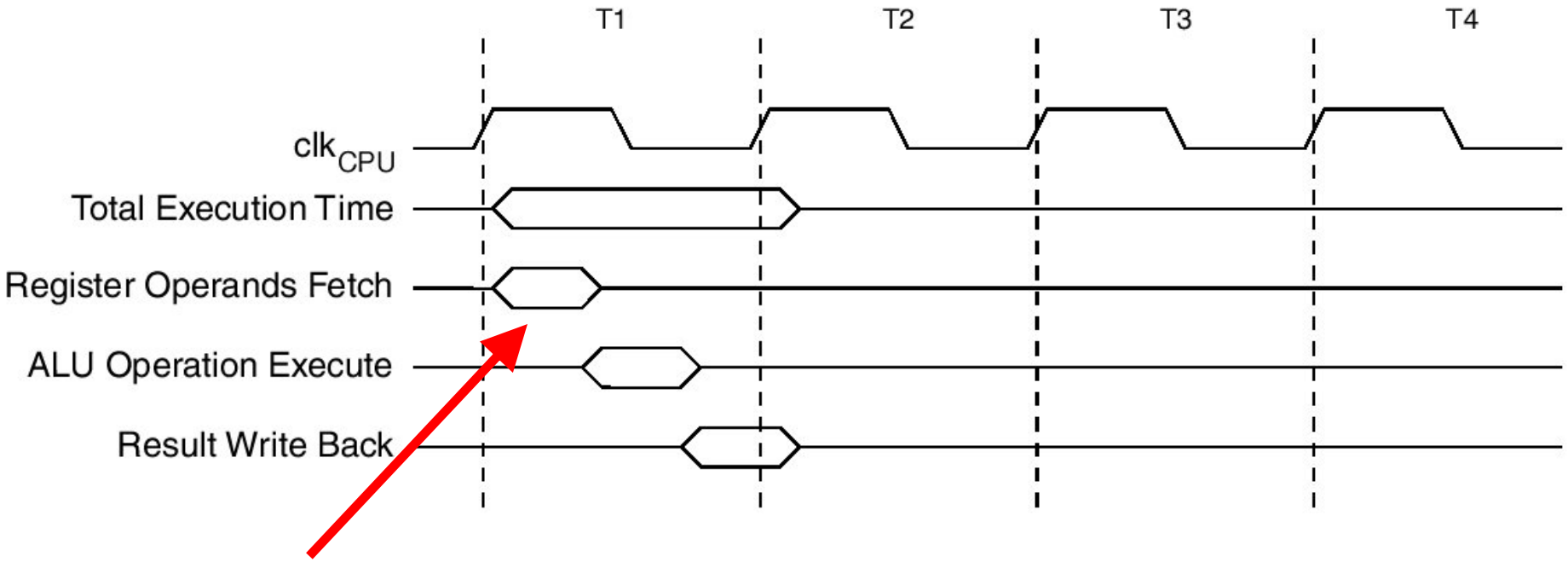
# Instruction Fetch/Execution Cycle




From Atmel Mega8 spec

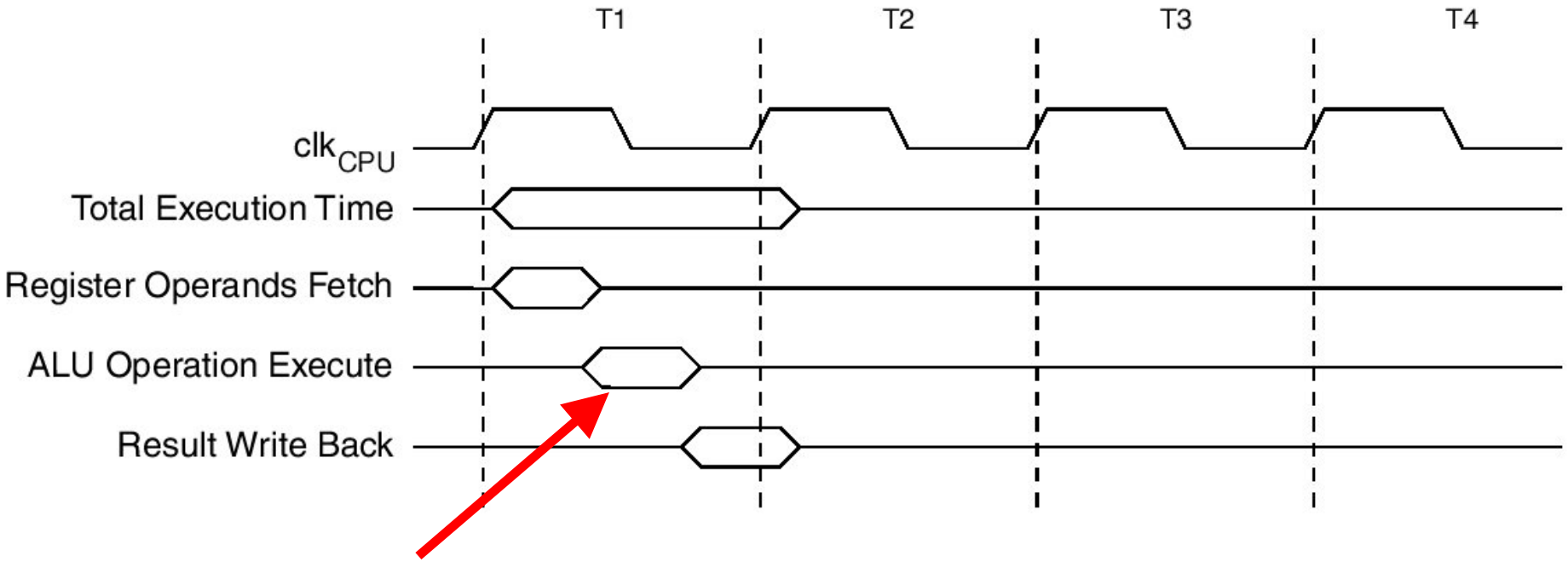
- While one instruction is being executed, the next is already being fetched from memory
- In many cases: each step happens on a single clock cycle

# Instruction Execution Cycle



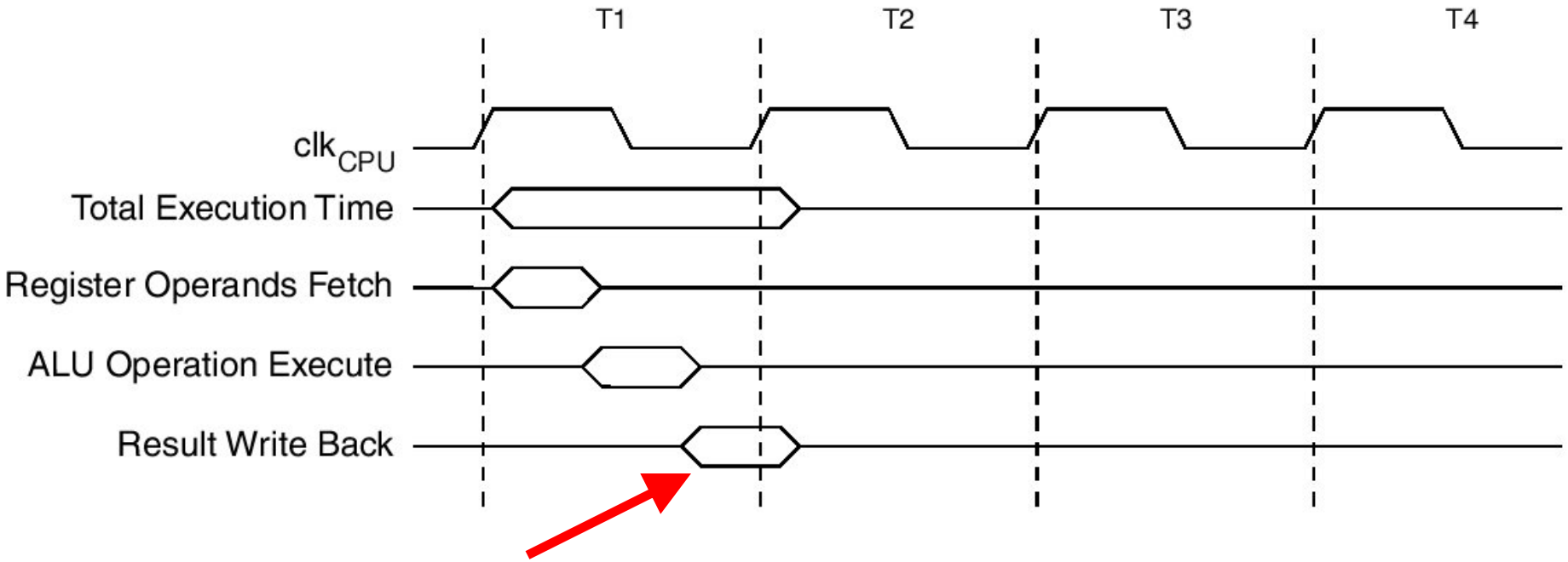

Address the registers and wait for the values to become available

# Instruction Execution Cycle




Perform the operation dictated by the instruction

# Instruction Execution Cycle



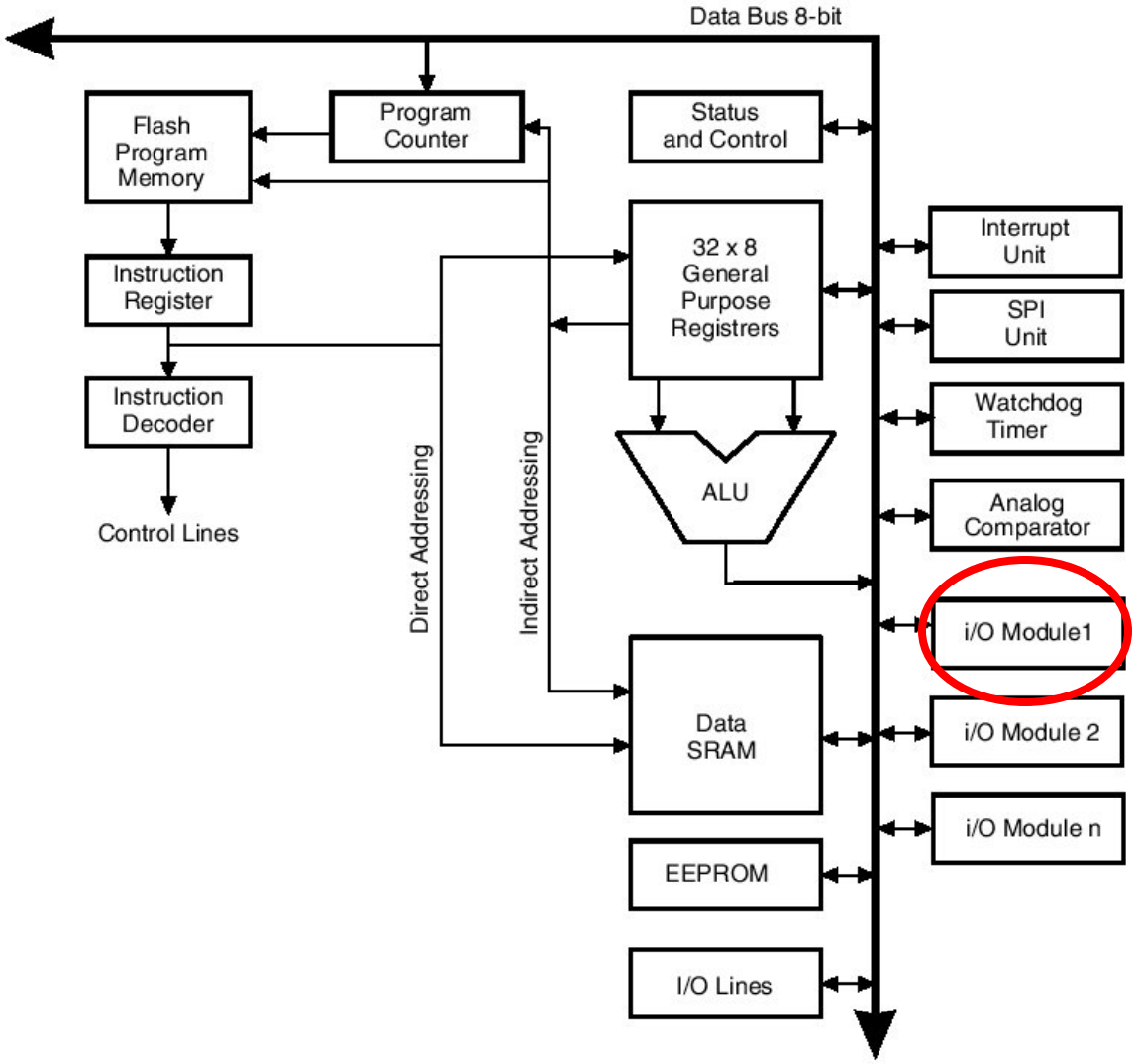

Result stored in destination register

Status register state changed

# Atmel Mega8

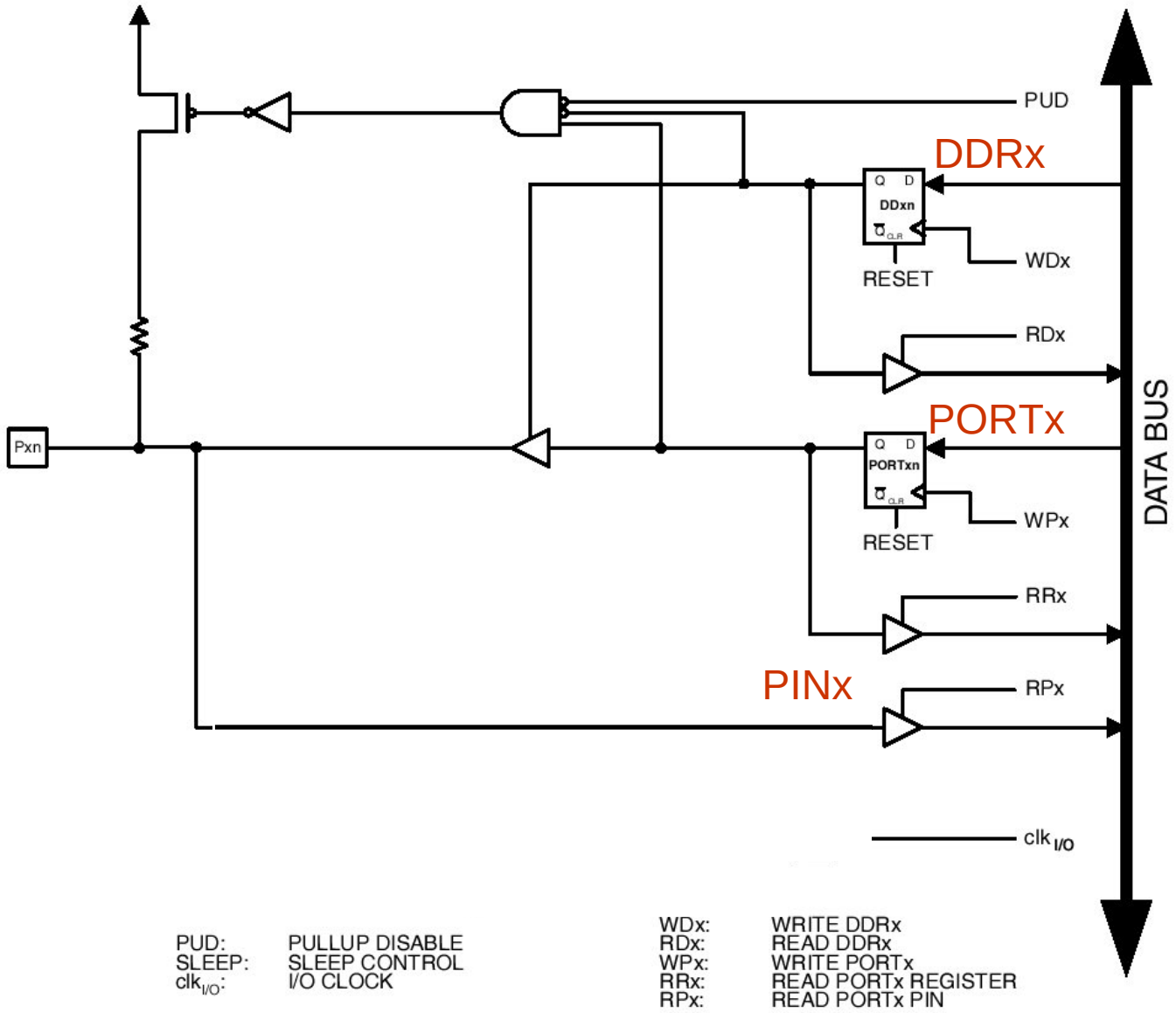
Control the pins through the I/O modules

- At the heart, these are registers … that are implemented using D flip-flops!

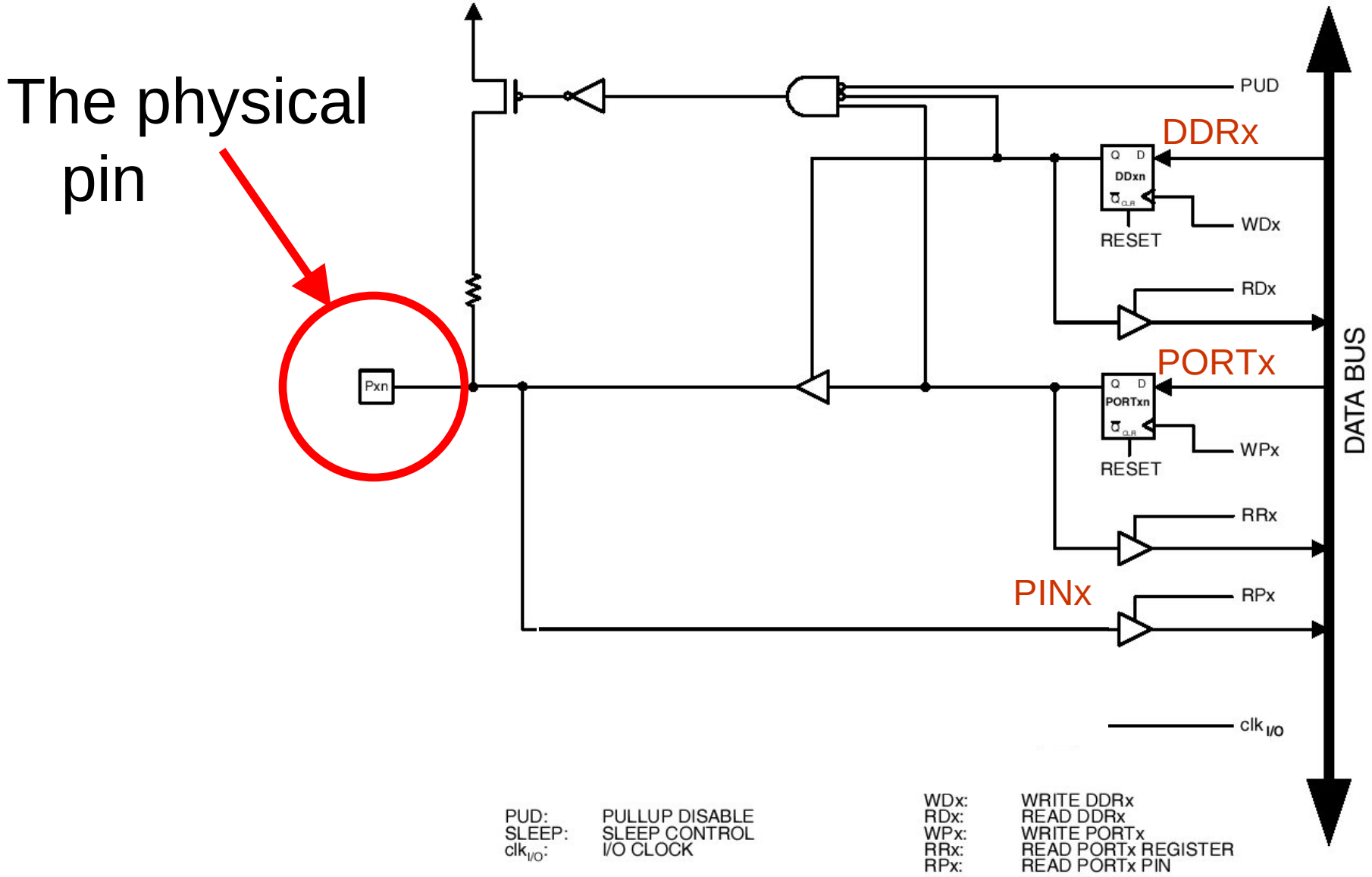


# I/O Pin Implementation

Single bit of PORT B



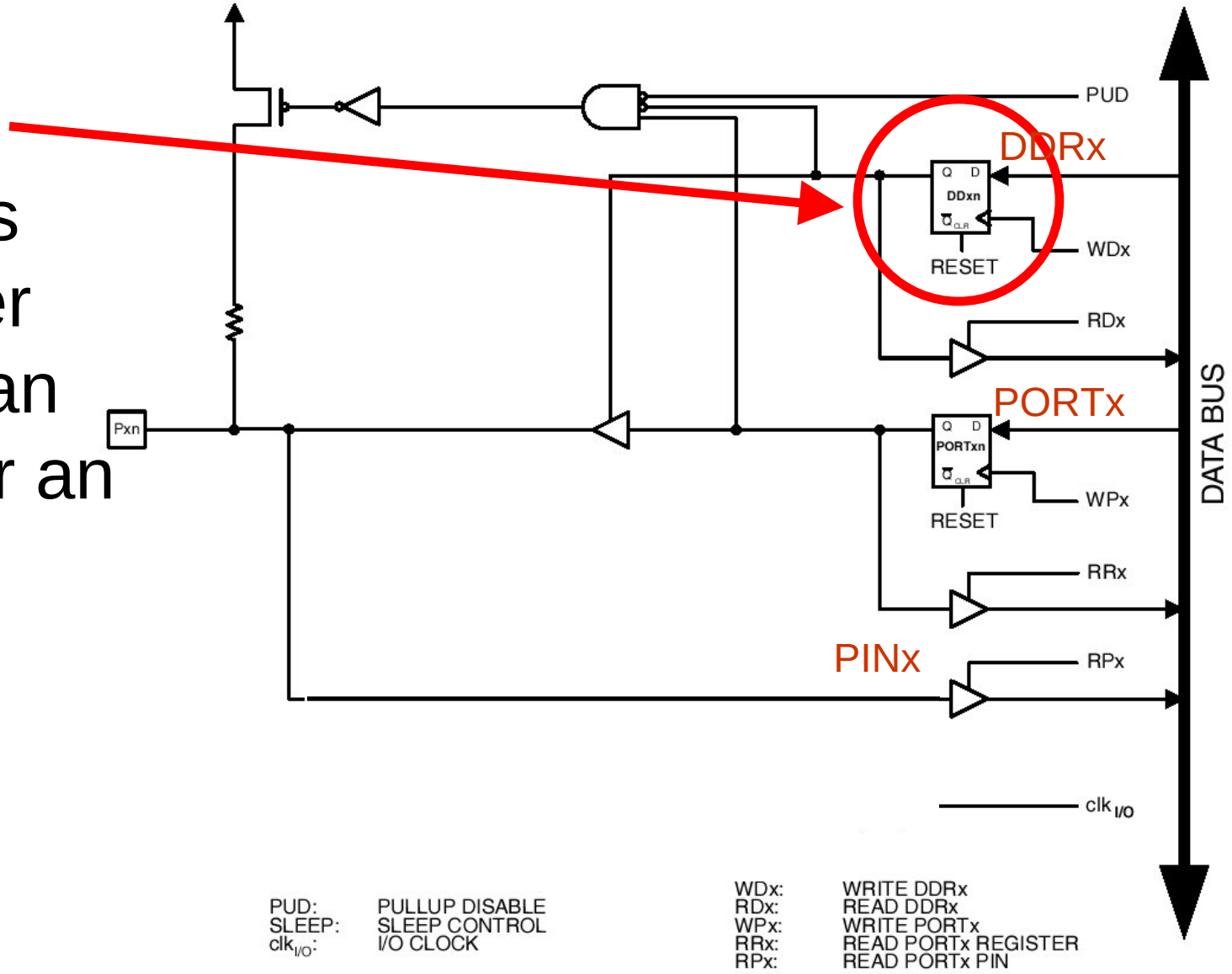
# I/O Pin Implementation



# I/O Pin Implementation

DDRB

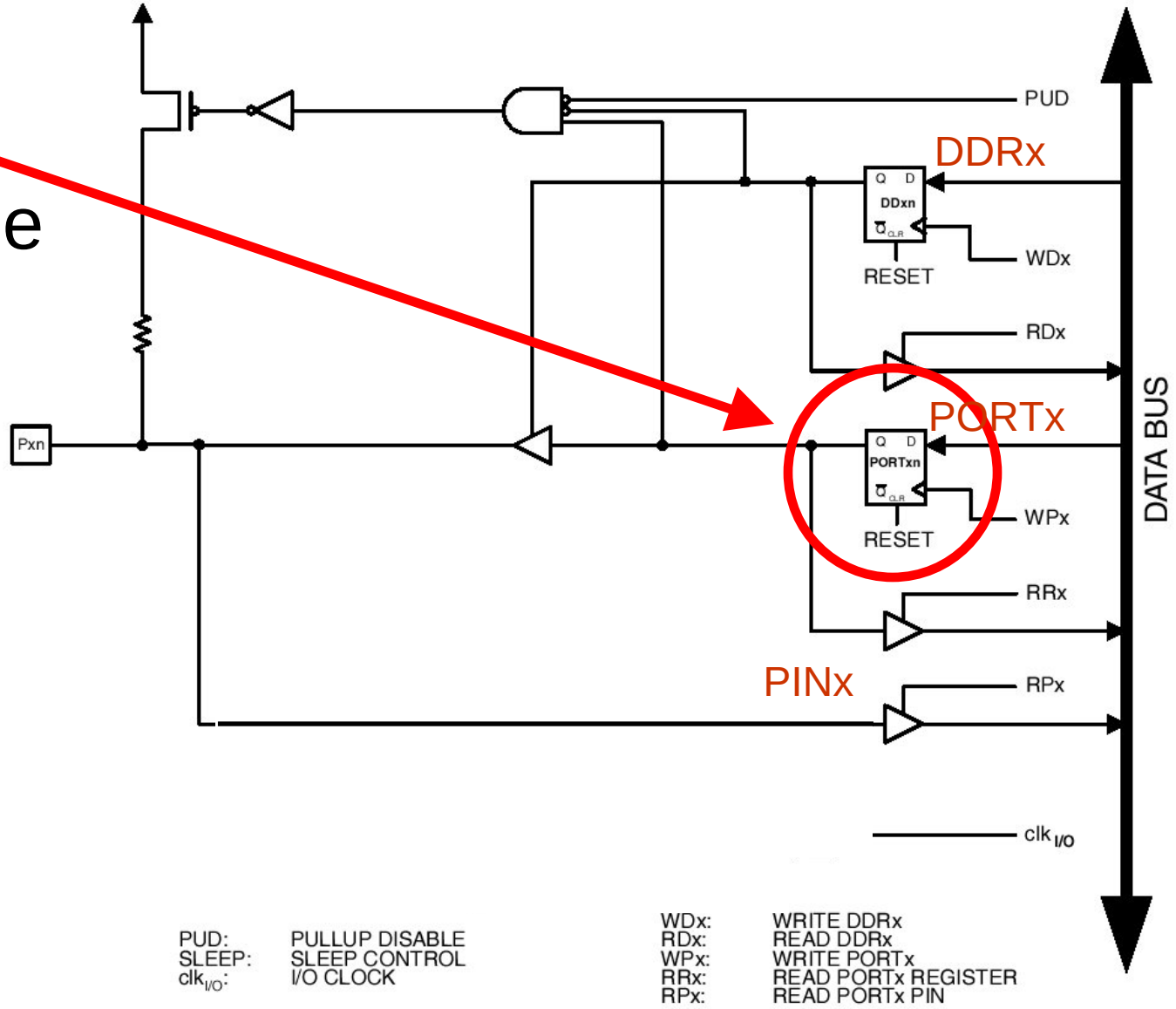
- Defines whether this is an input or an output



# I/O Pin Implementation

PORTB

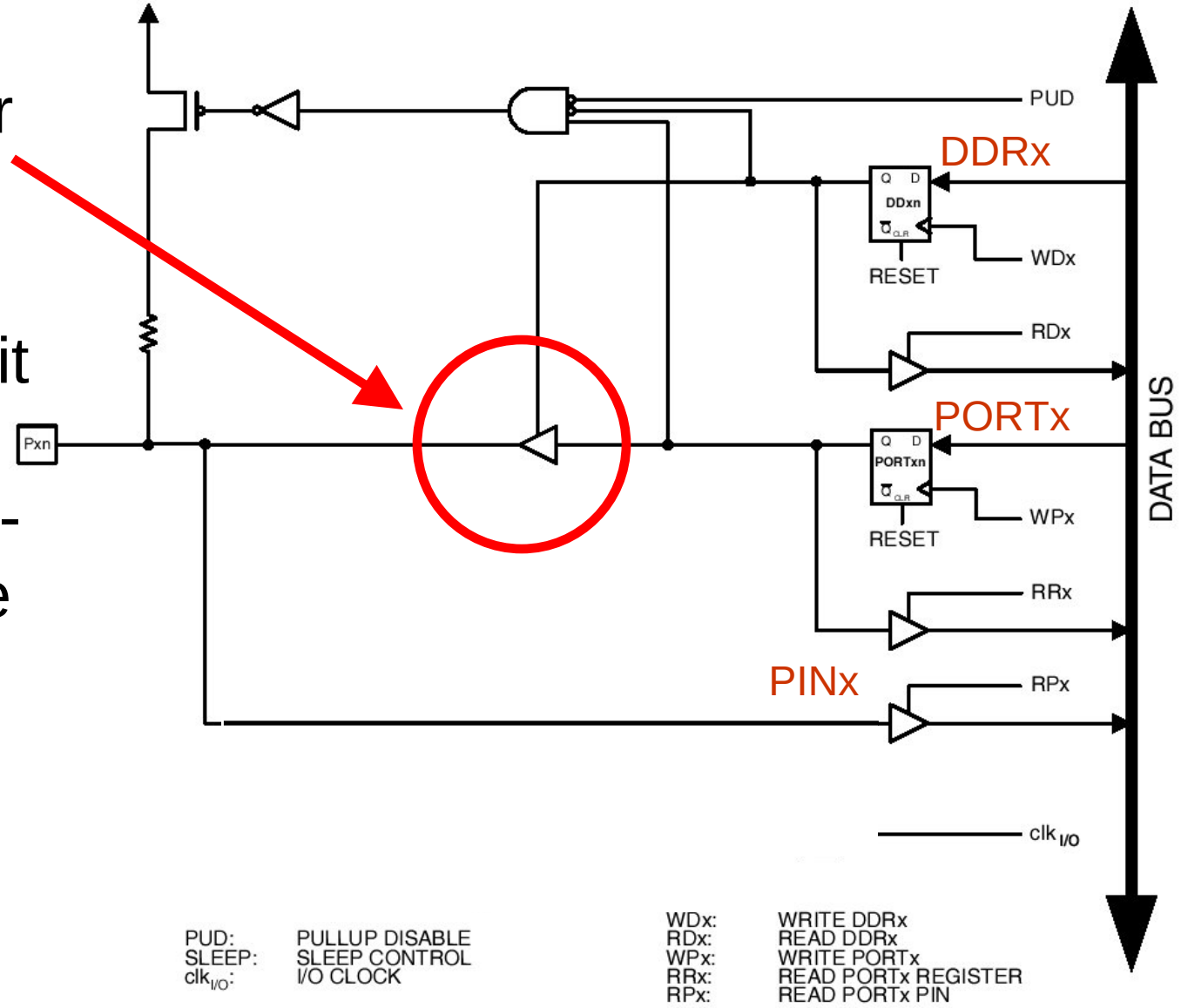
- Defines the value that is written out to the pin (if it is an output)



# I/O Pin Implementation

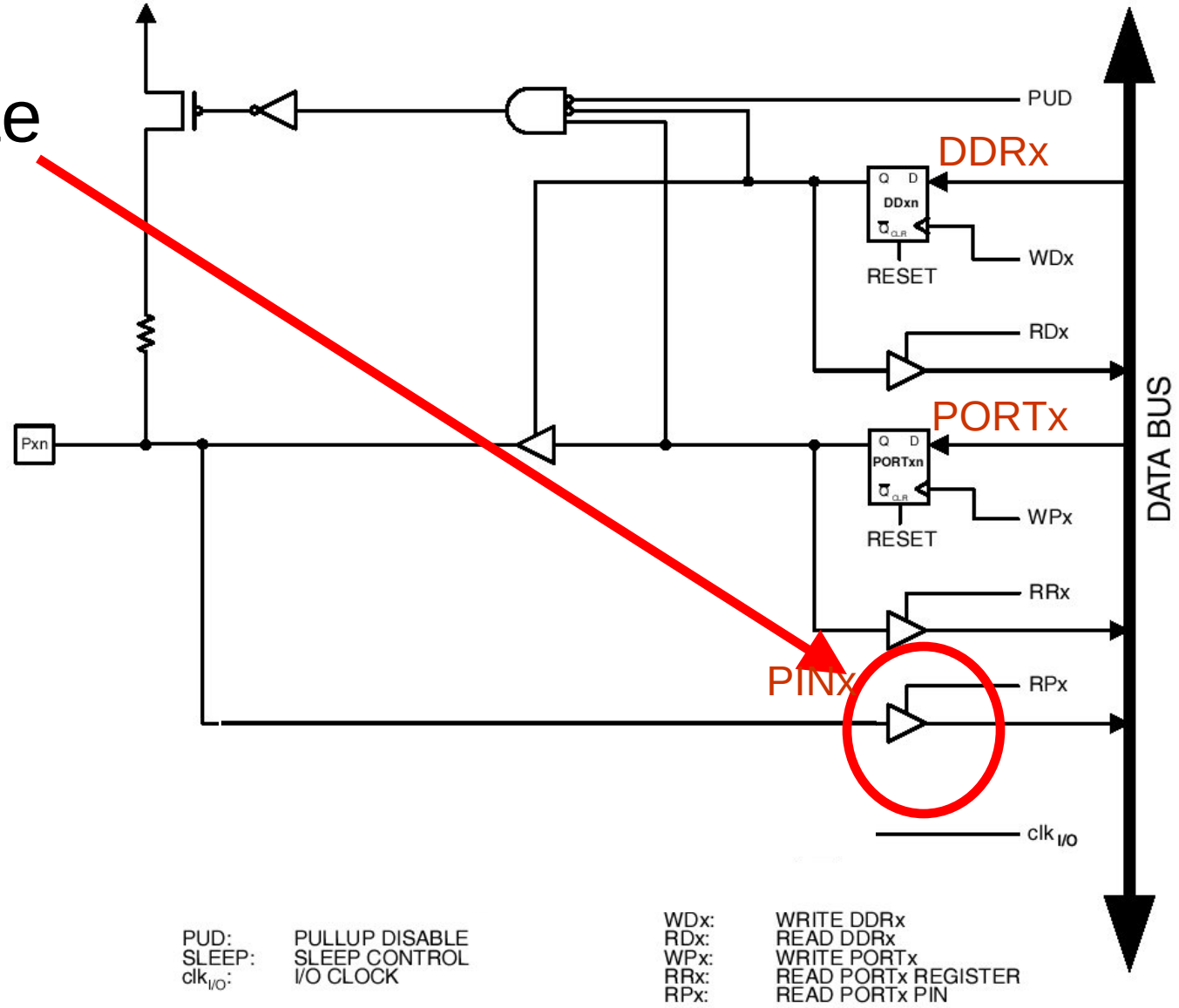
Tristate buffer

- When this pin is an output pin, it allows the PORTB flip-flop to drive the pin

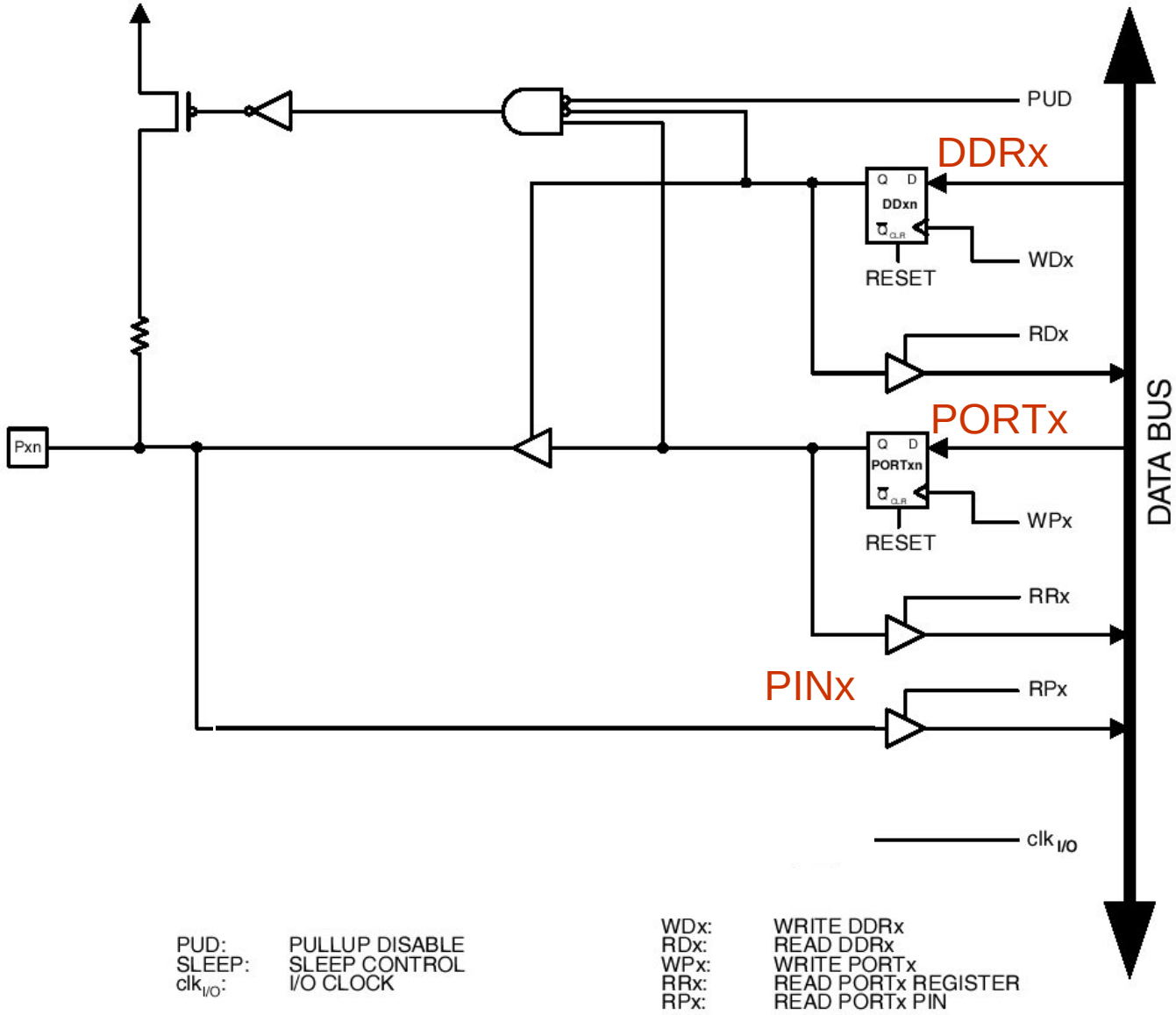


# I/O Pin Implementation

Input tri-state buffer

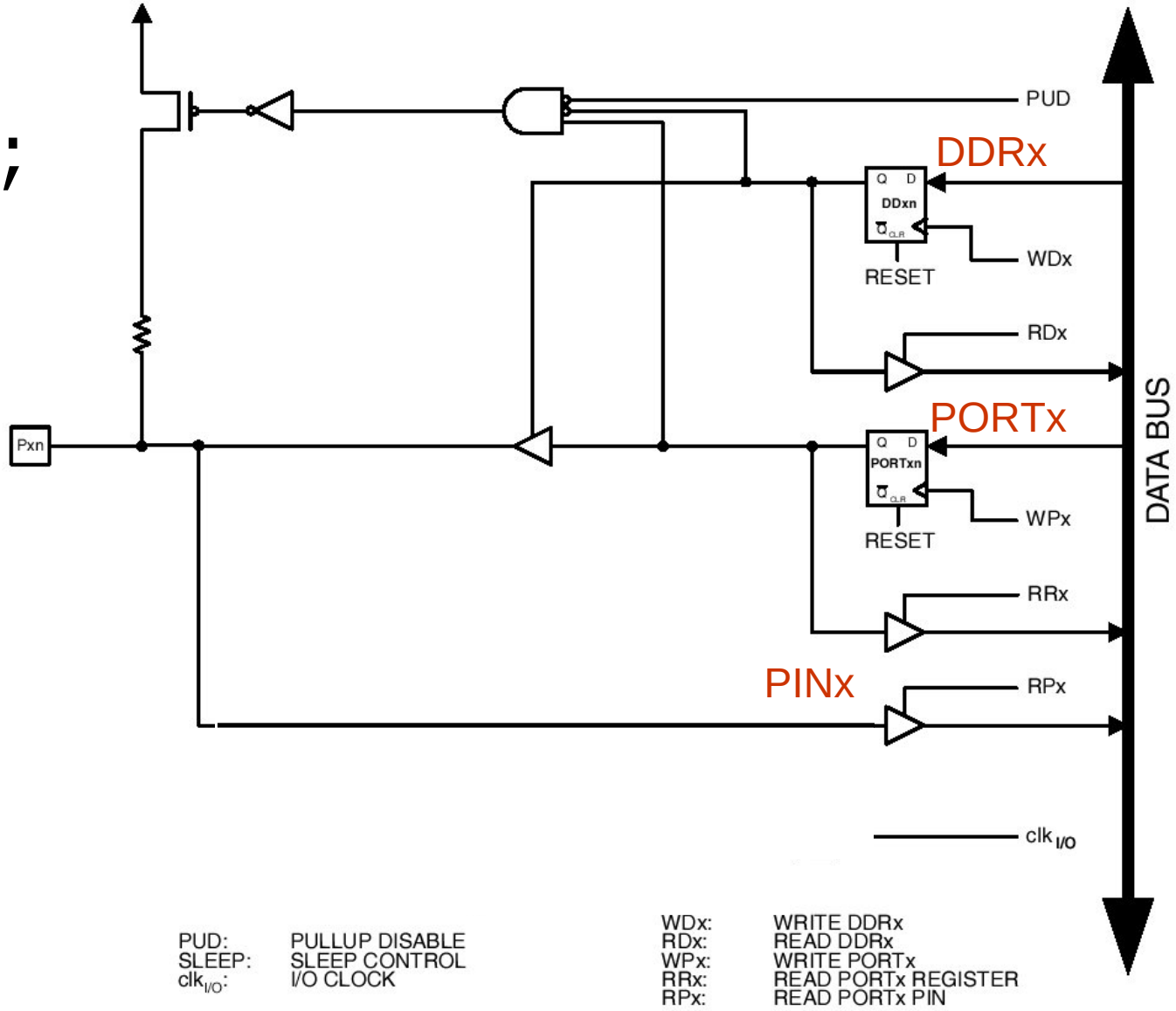


# I/O Pin Implementation



# I/O Pin Implementation

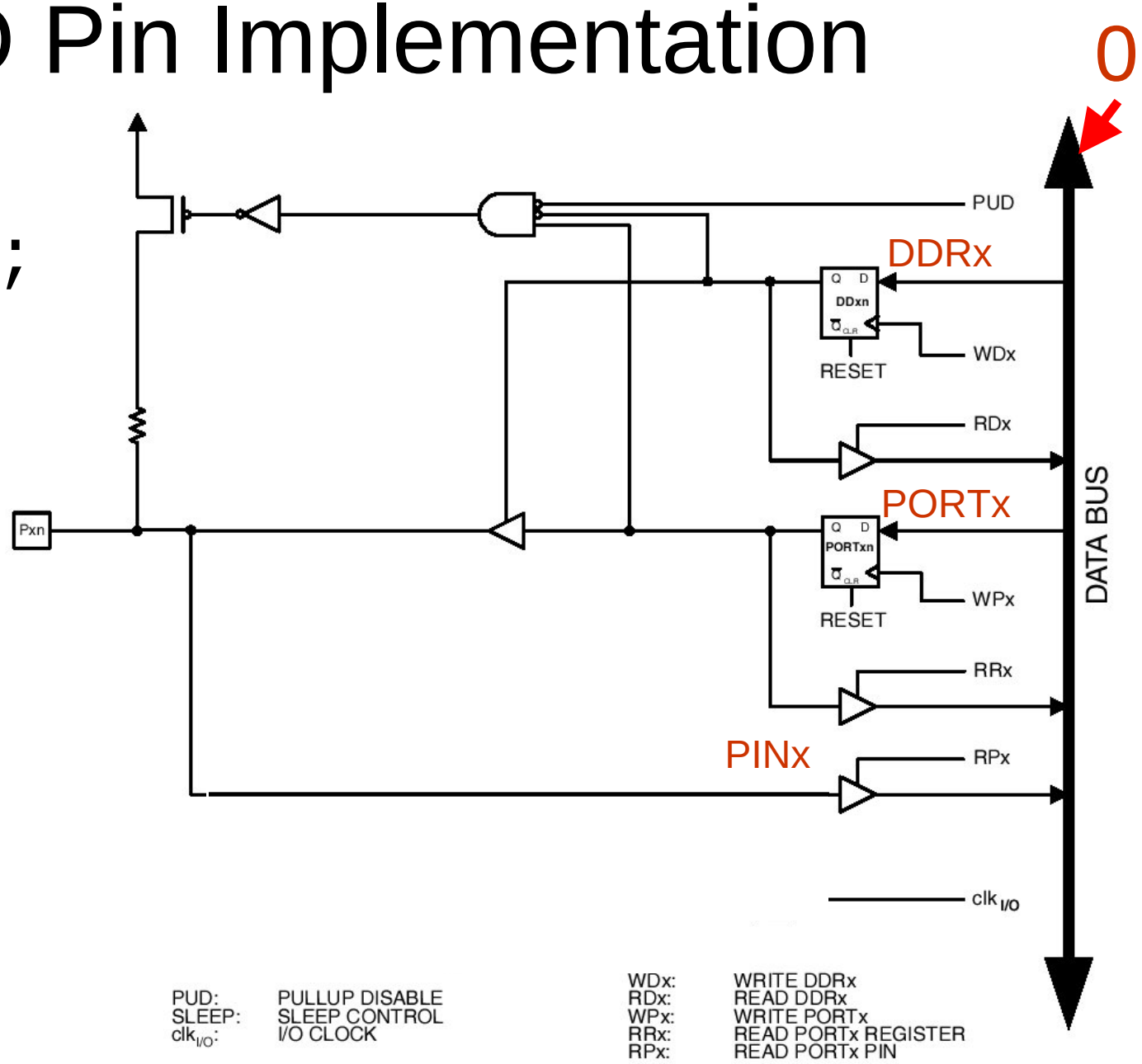
```
DDRB = 0;
```



# I/O Pin Implementation

```
DDRB = 0;
```

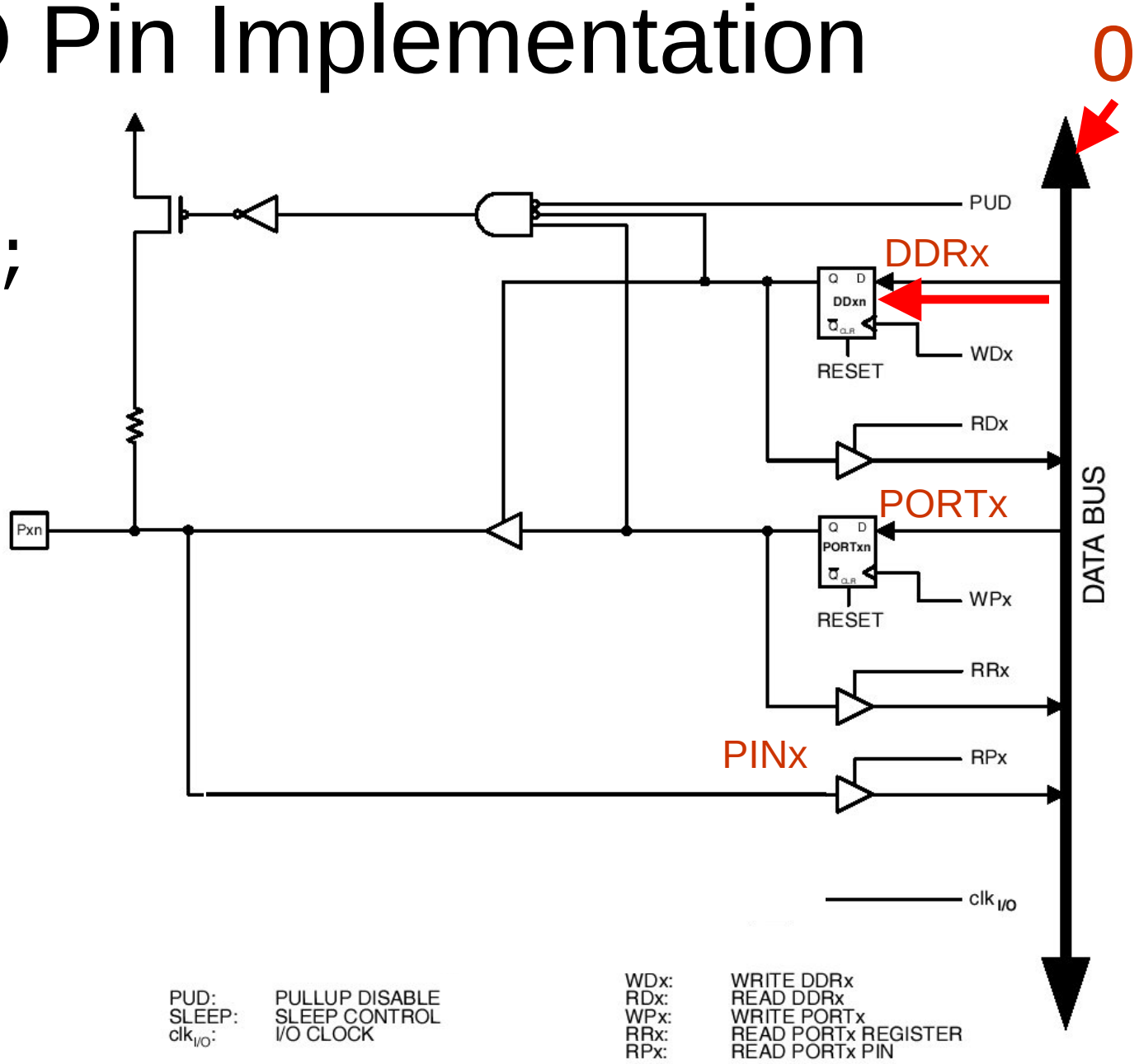
- “0” is written to the data bus



# I/O Pin Implementation

```
DDRB = 0;
```

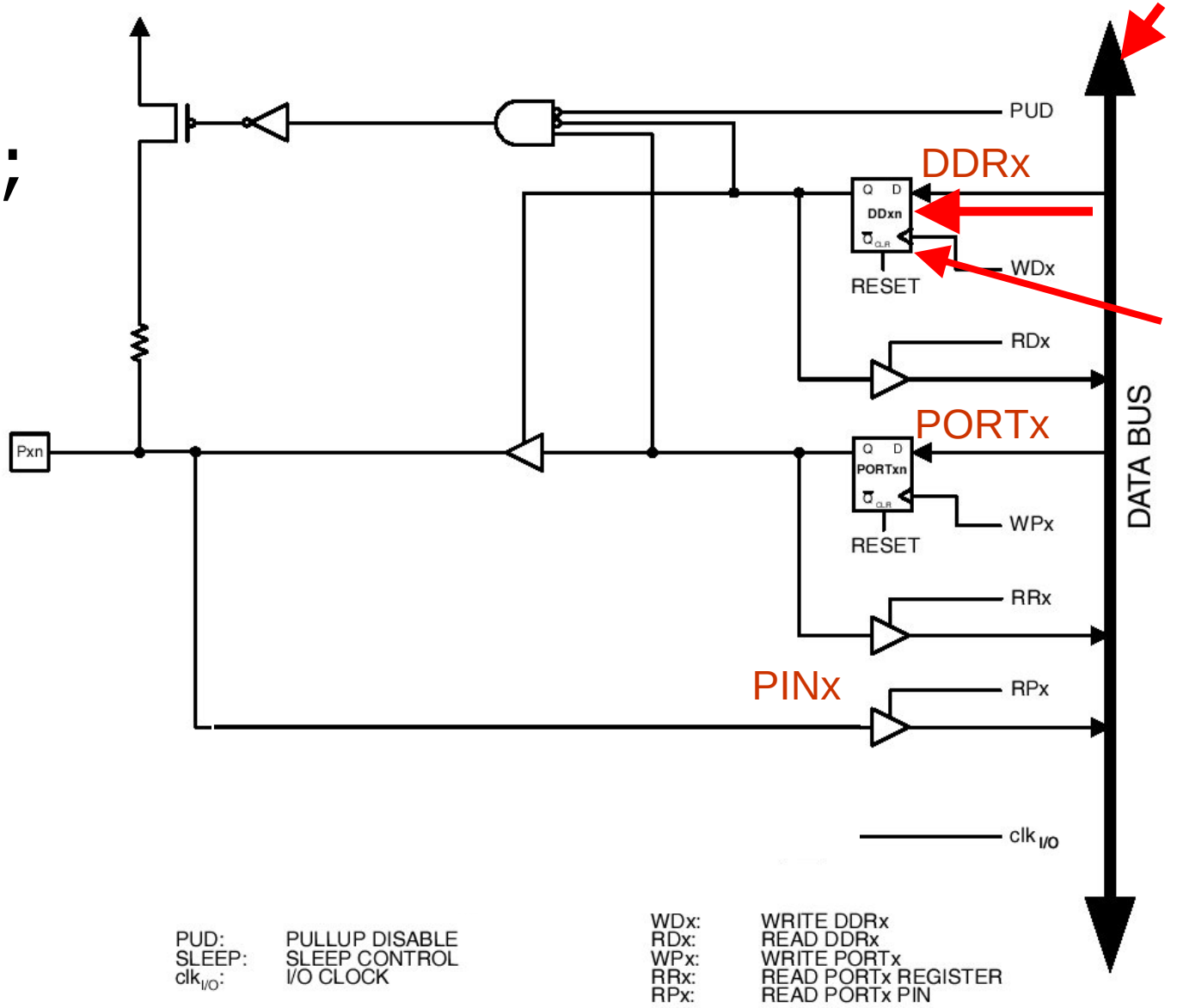
- “0” is written to the data bus
- This is input to the DDRB register



# I/O Pin Implementation

```
DDRB = 0;
```

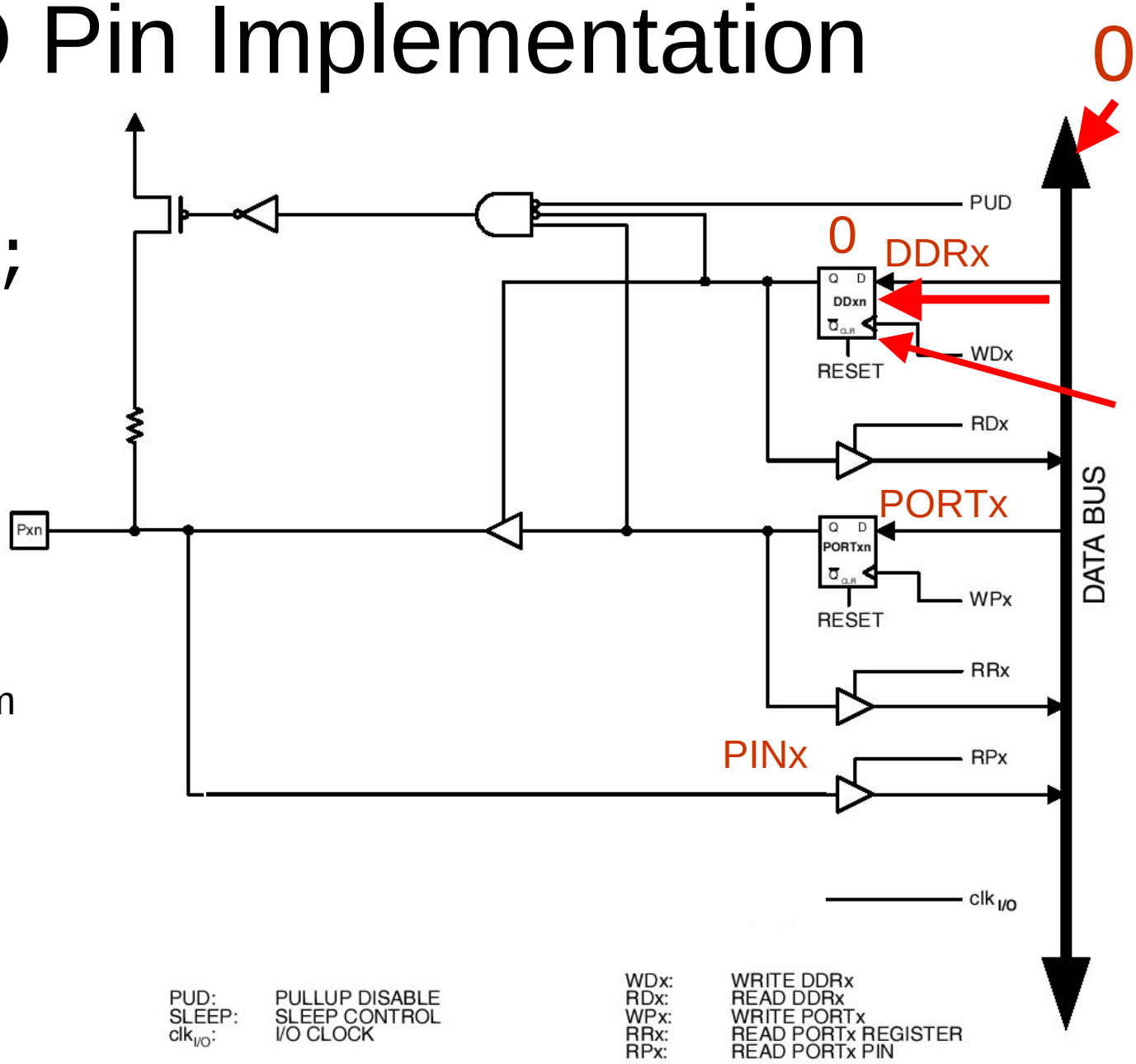
- “0” is written to the data bus
- This is input to the DDRB register
- WDB is clocked from high to low



# I/O Pin Implementation

```
DDRB = 0;
```

- “0” is written to the data bus
- This is input to the DDRB register
- WDB is clocked from high to low
- “0” is stored by the flip-flop

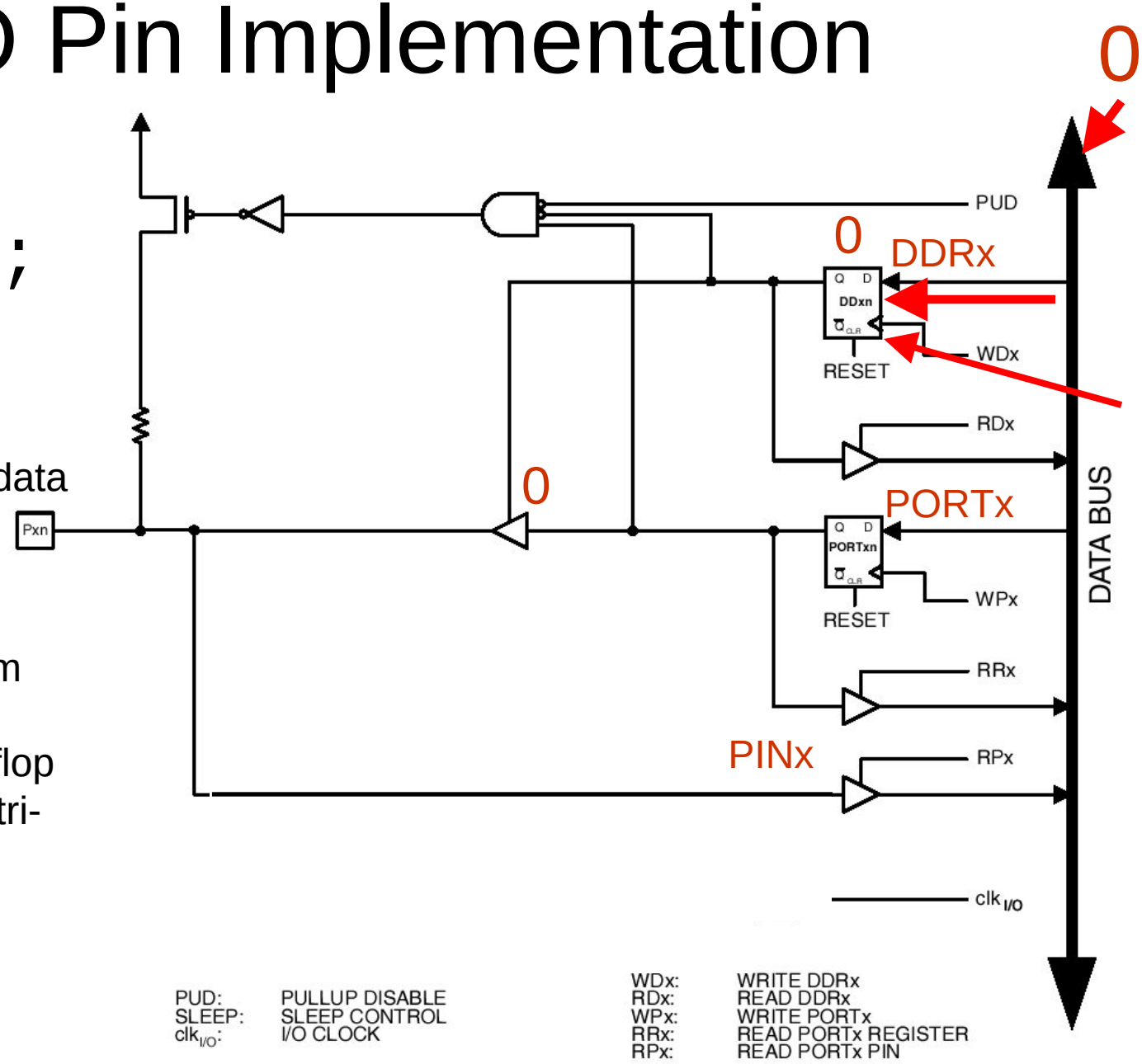


# I/O Pin Implementation

```
DDRB = 0;
```

- “0” is written to the data bus
- This is input to the DDRB register
- WDB is clocked from high to low
- “0” is stored by flip-flop
- Which turns off the tri-state buffer

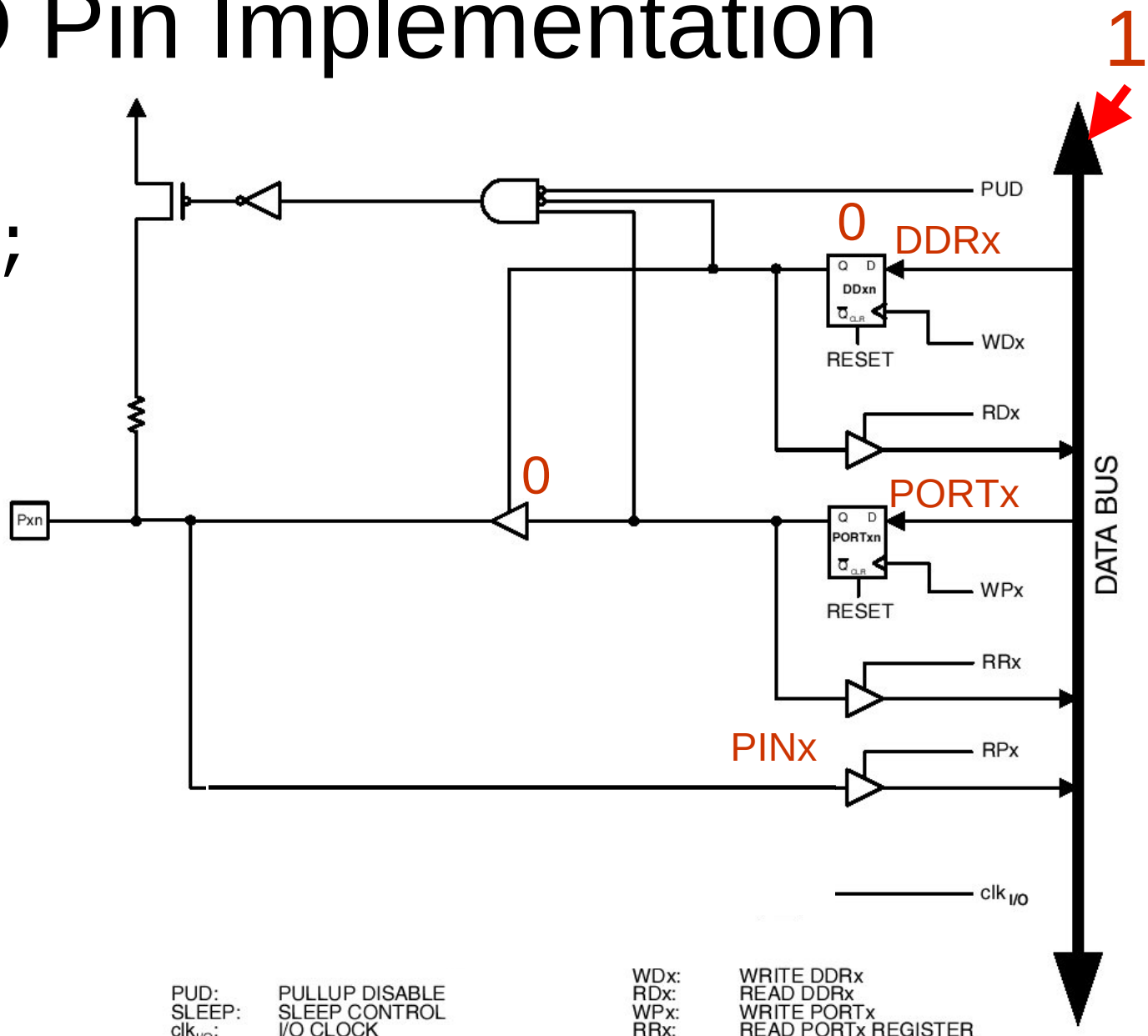
-> this is an input pin



# I/O Pin Implementation

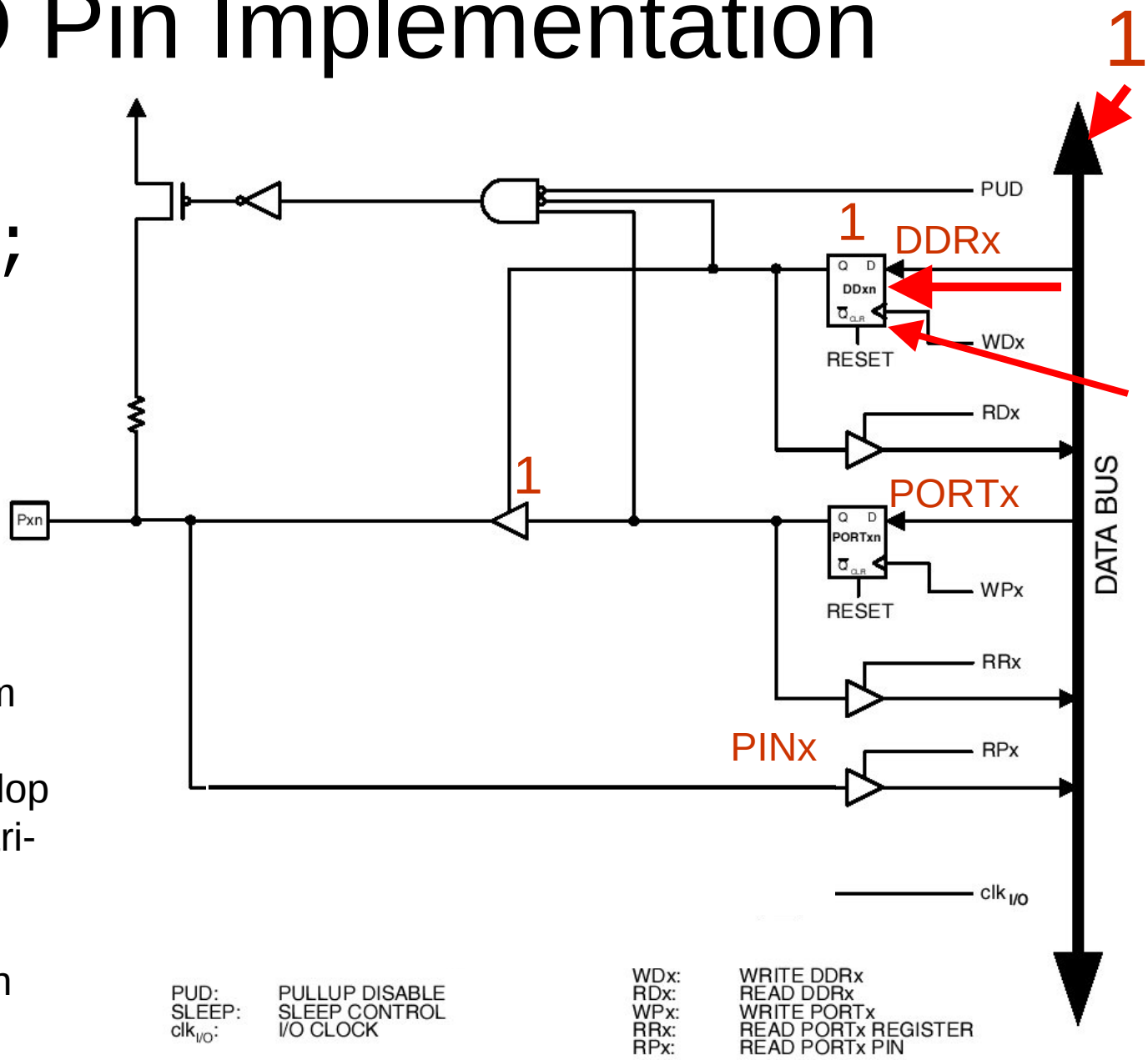
```
DDRB = 1;
```

- “1” is written to the data bus



# I/O Pin Implementation

```
DDRB = 1;
```

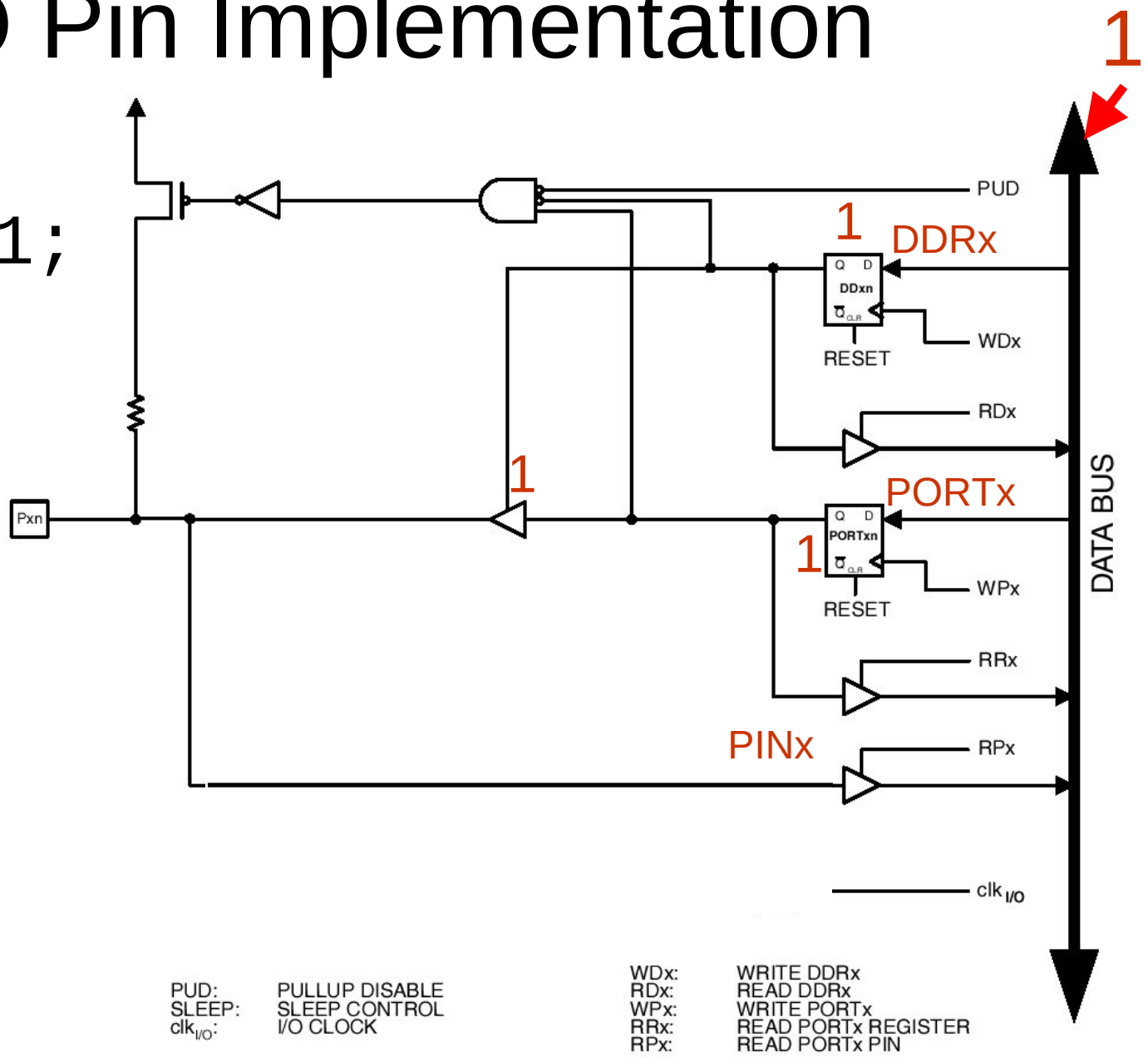



- “1” is written to the data bus
- This is input to the DDRB register
- WDB is clocked from high to low
- “1” is stored by flip-flop
- Which turns on the tri-state buffer

-> this is an output pin

# I/O Pin Implementation

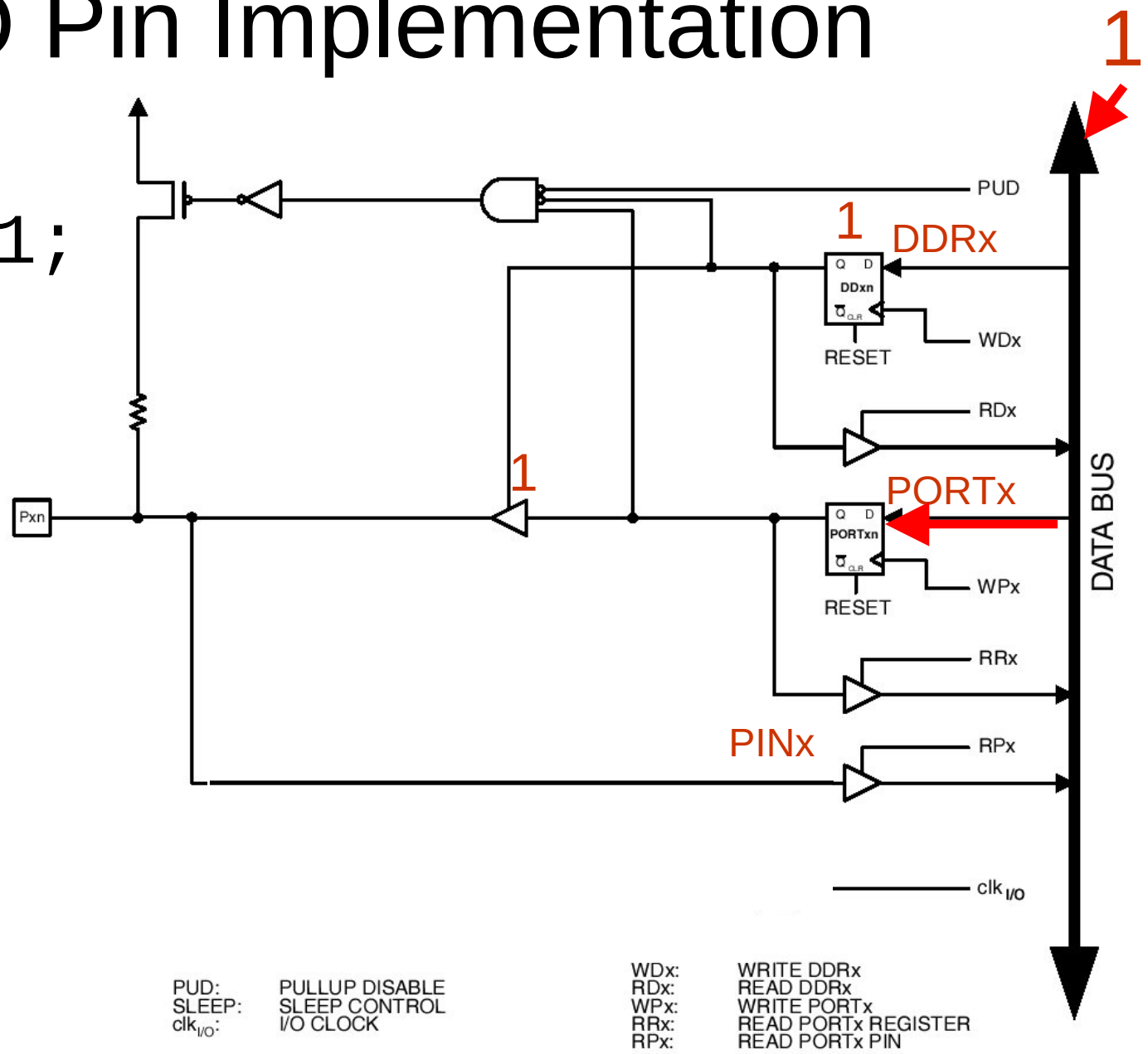
```
PORTB = 1;
```



# I/O Pin Implementation

```
PORTB = 1;
```

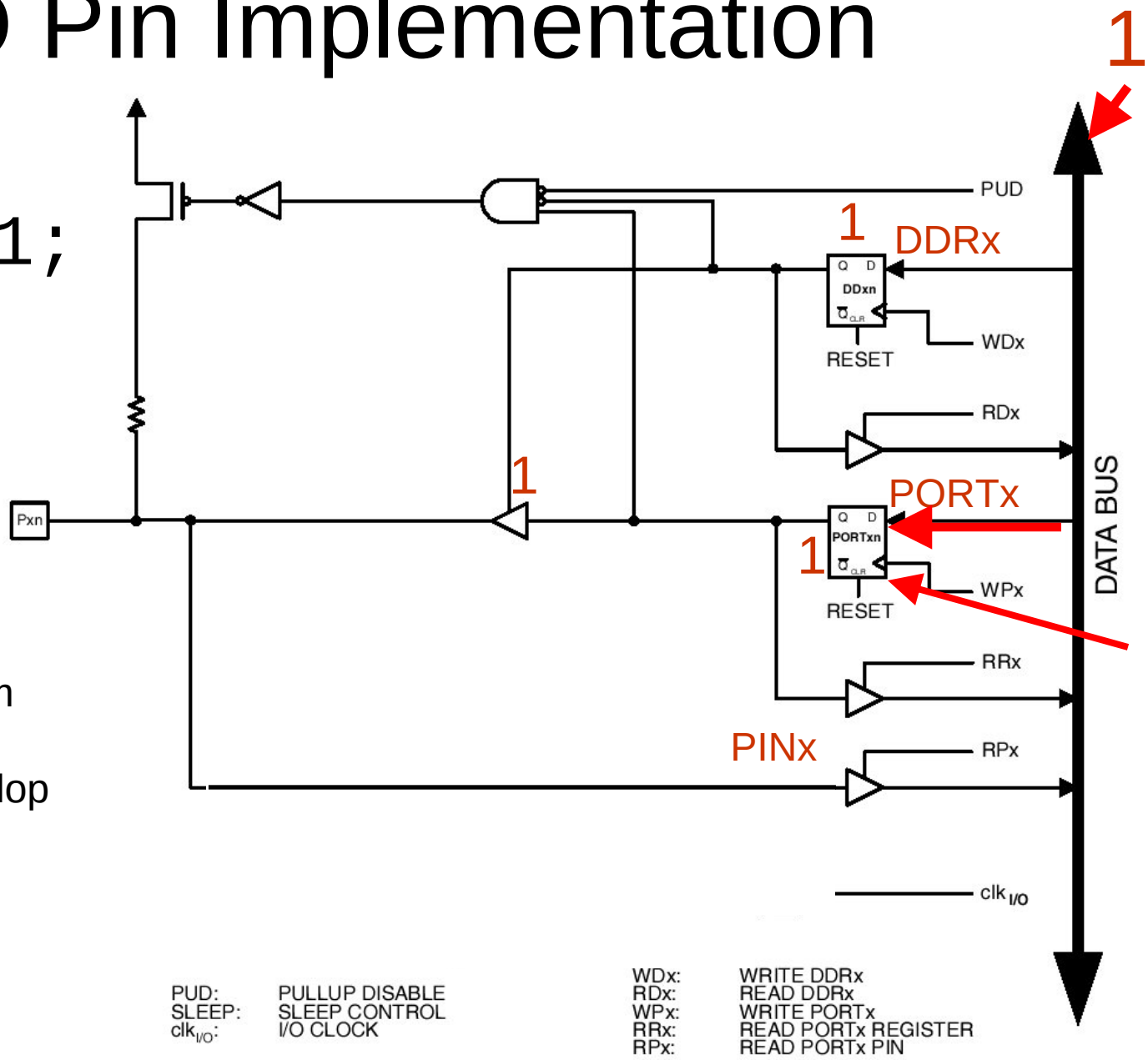
- “1” is written to the data bus
- This is input to the PORTB register



# I/O Pin Implementation

```
PORTB = 1;
```

- “1” is written to the data bus
- This is input to the PORTB register
- WPB is clocked from high to low
- “1” is stored by flip-flop

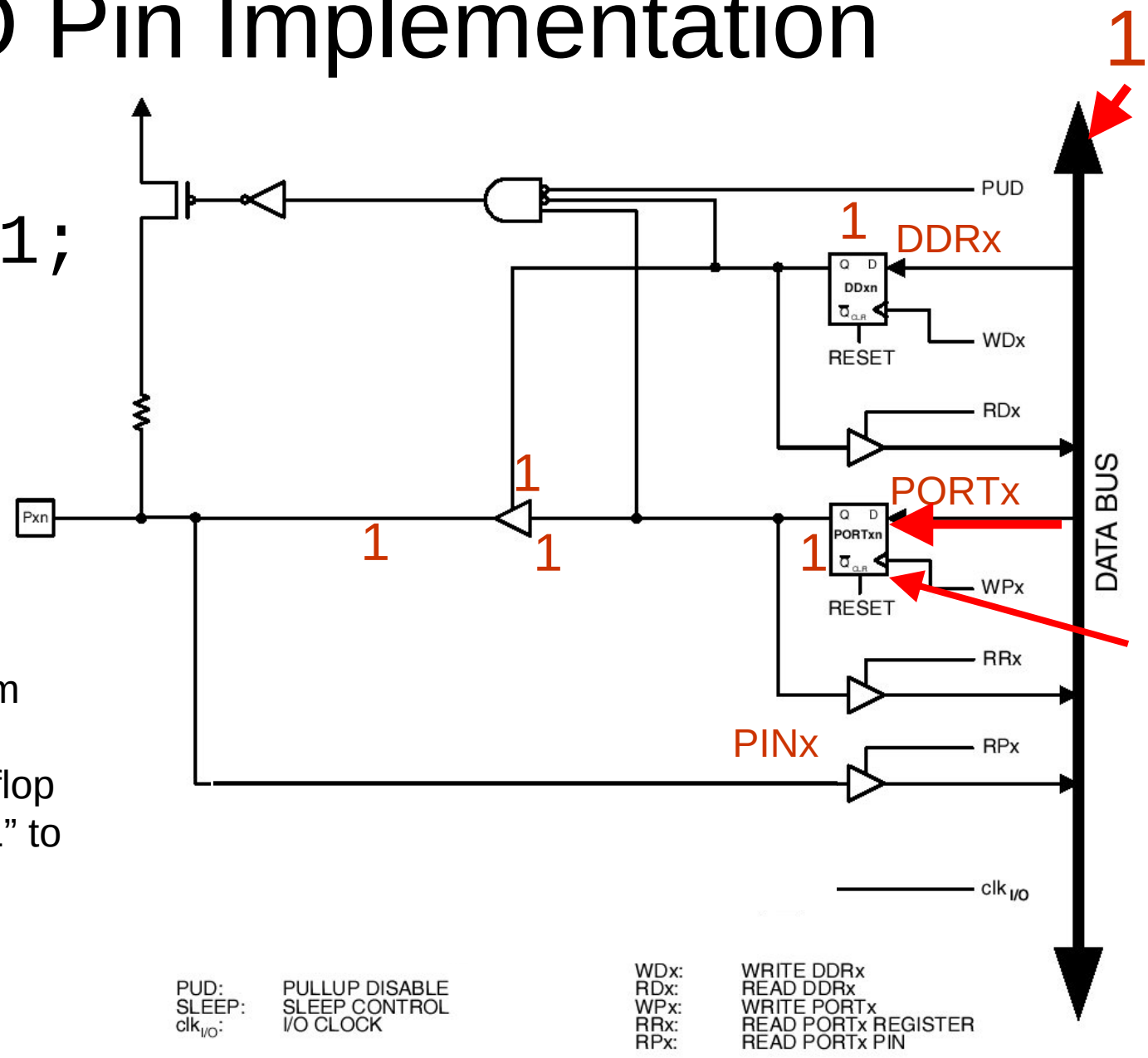


# I/O Pin Implementation

```
PORTB = 1;
```

- “1” is written to the data bus
- This is input to the PORTB register
- WPB is clocked from high to low
- “1” is stored by flip-flop
- Which provides a “1” to the tri-state buffer

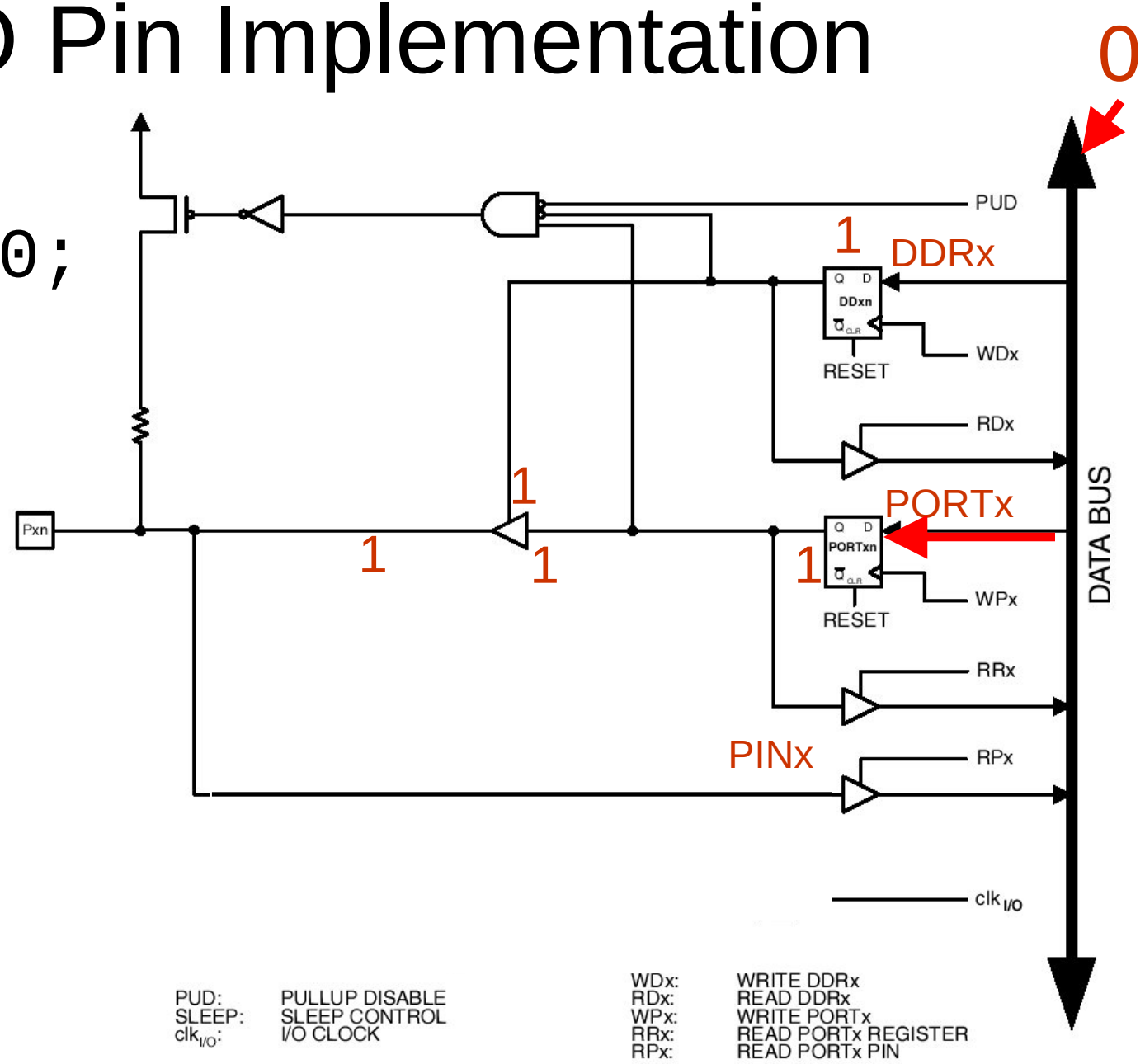
-> output a “1”



# I/O Pin Implementation

```
PORTB = 0;
```

- “0” is written to the data bus

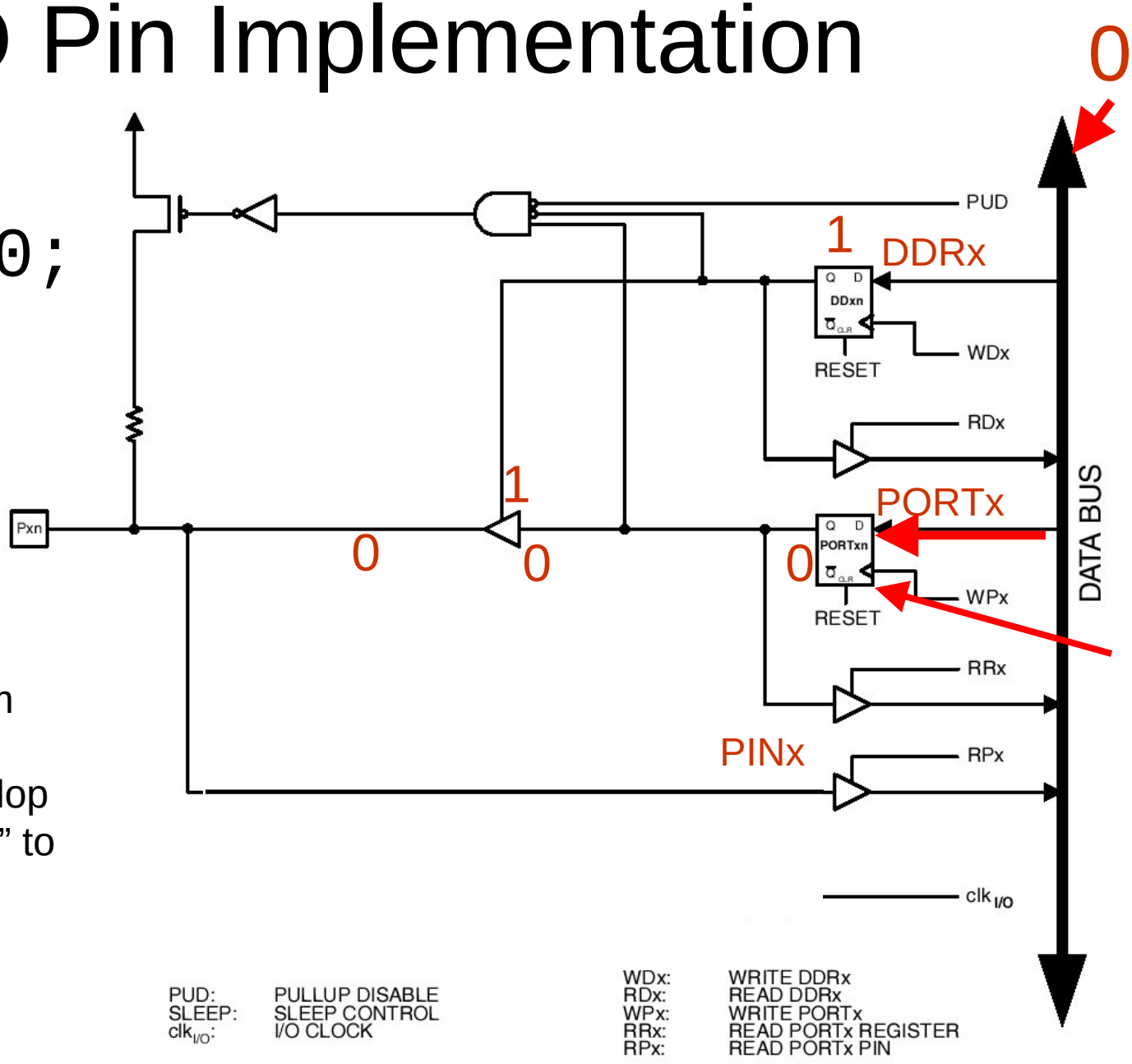


# I/O Pin Implementation

```
PORTB = 0;
```

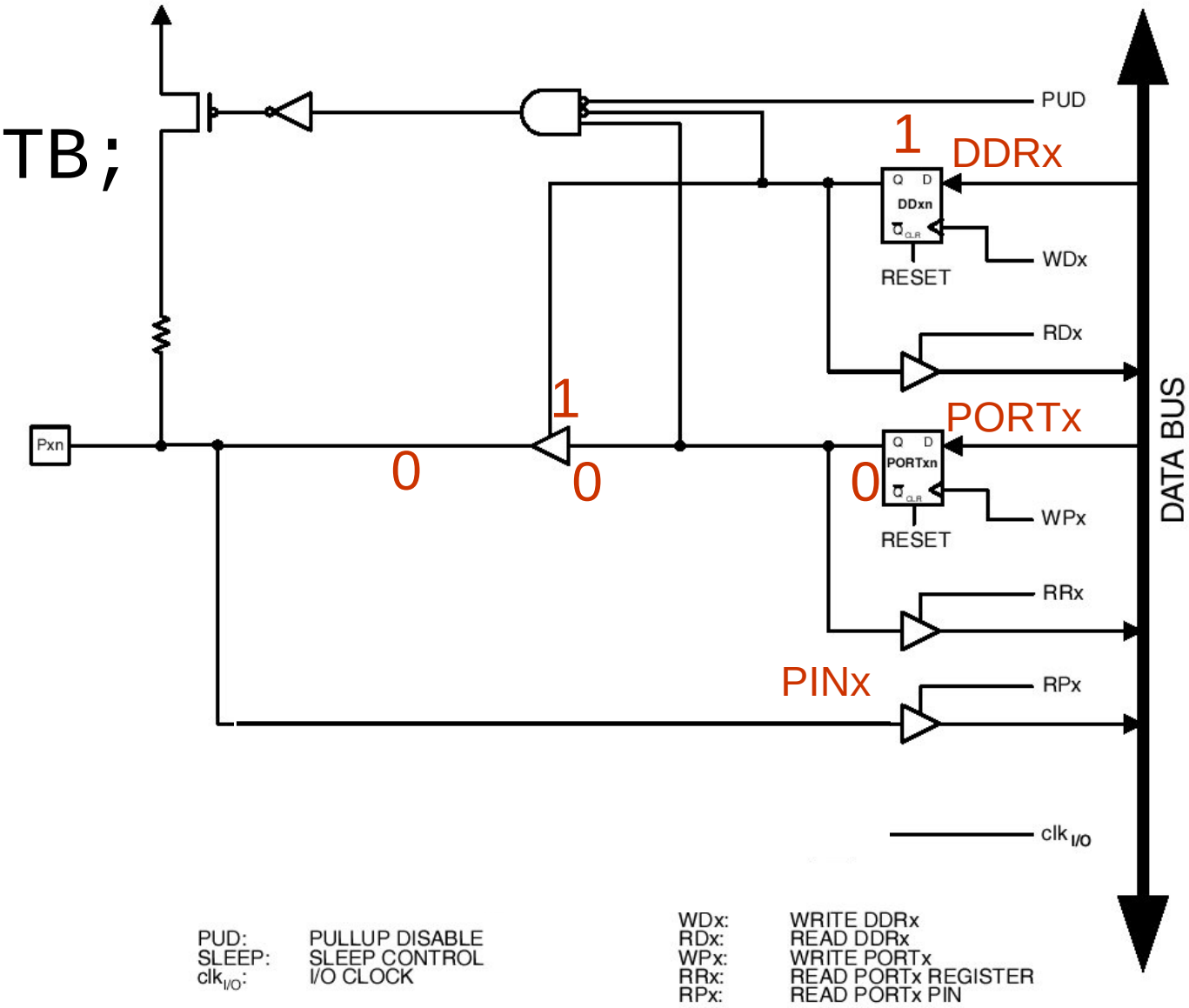
- “0” is written to the data bus
- This is input to the PORTB register
- WPB is clocked from high to low
- “0” is stored by flip-flop
- Which provides a “0” to the tri-state buffer

-> output a “0”



# I/O Pin Implementation

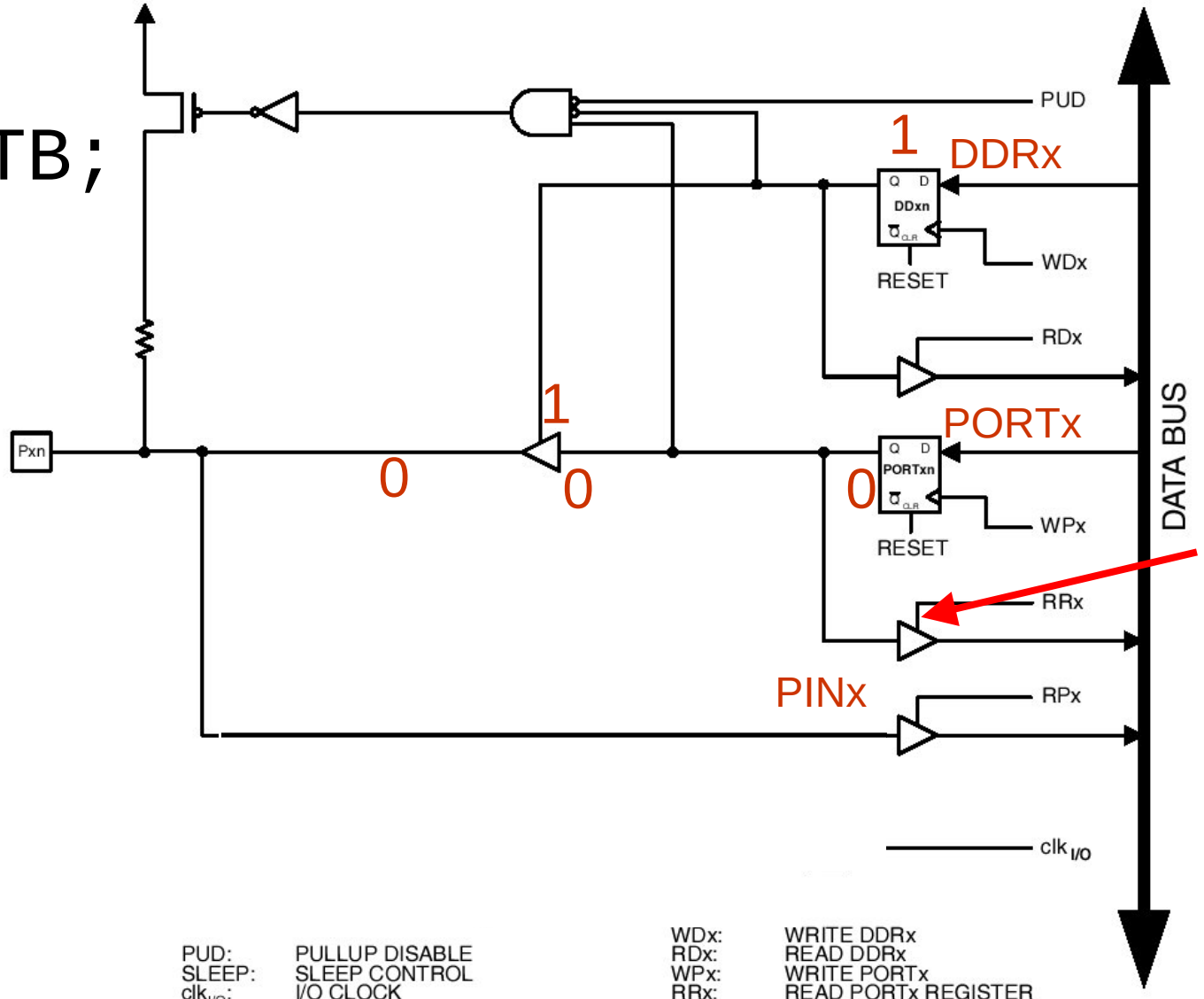
```
foo = PORTB;
```



# I/O Pin Implementation

```
foo = PORTB;
```

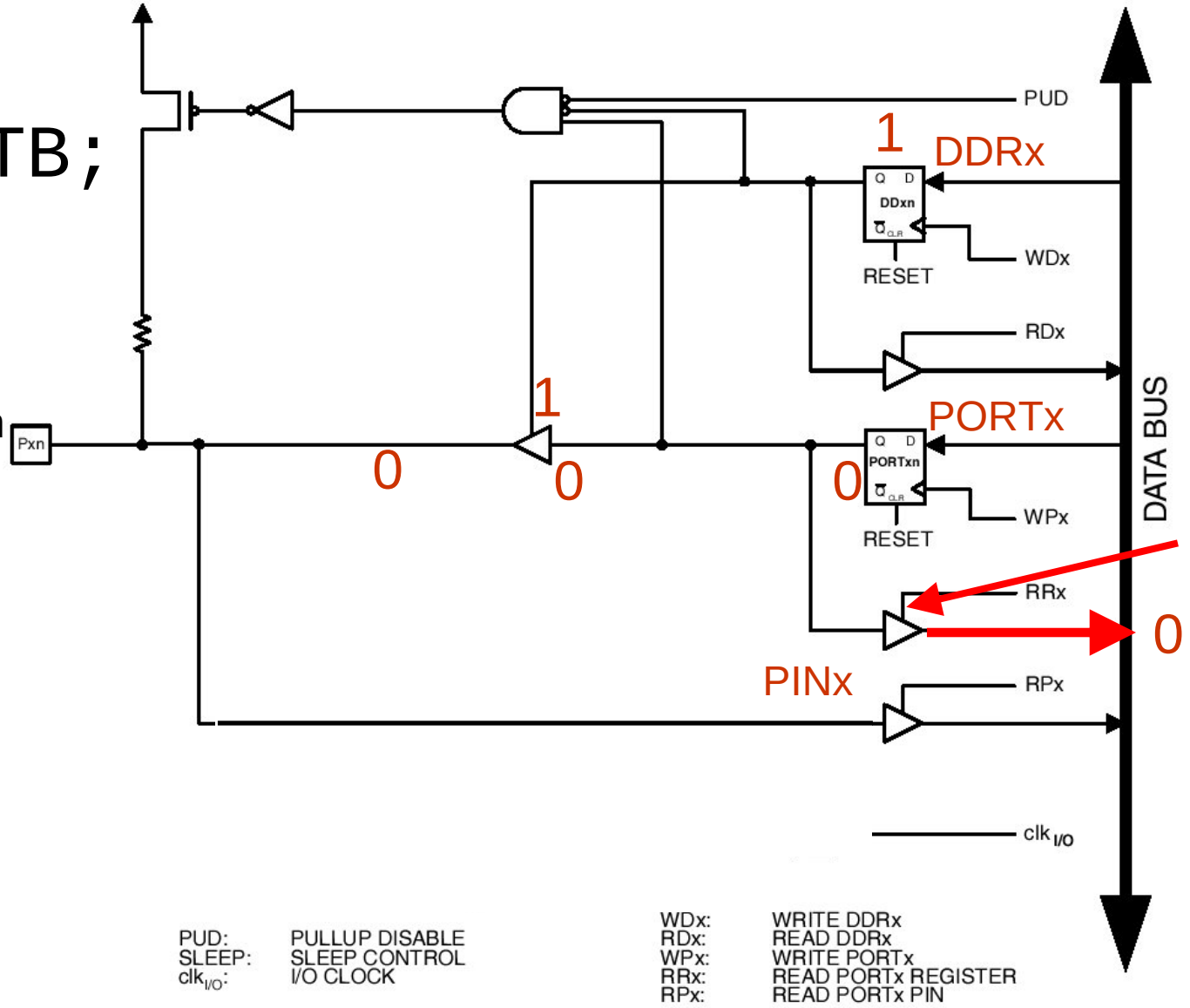
- RPB is set high



# I/O Pin Implementation

```
foo = PORTB;
```

- RPB is clocked from high to low
- “0” is written to the data bus

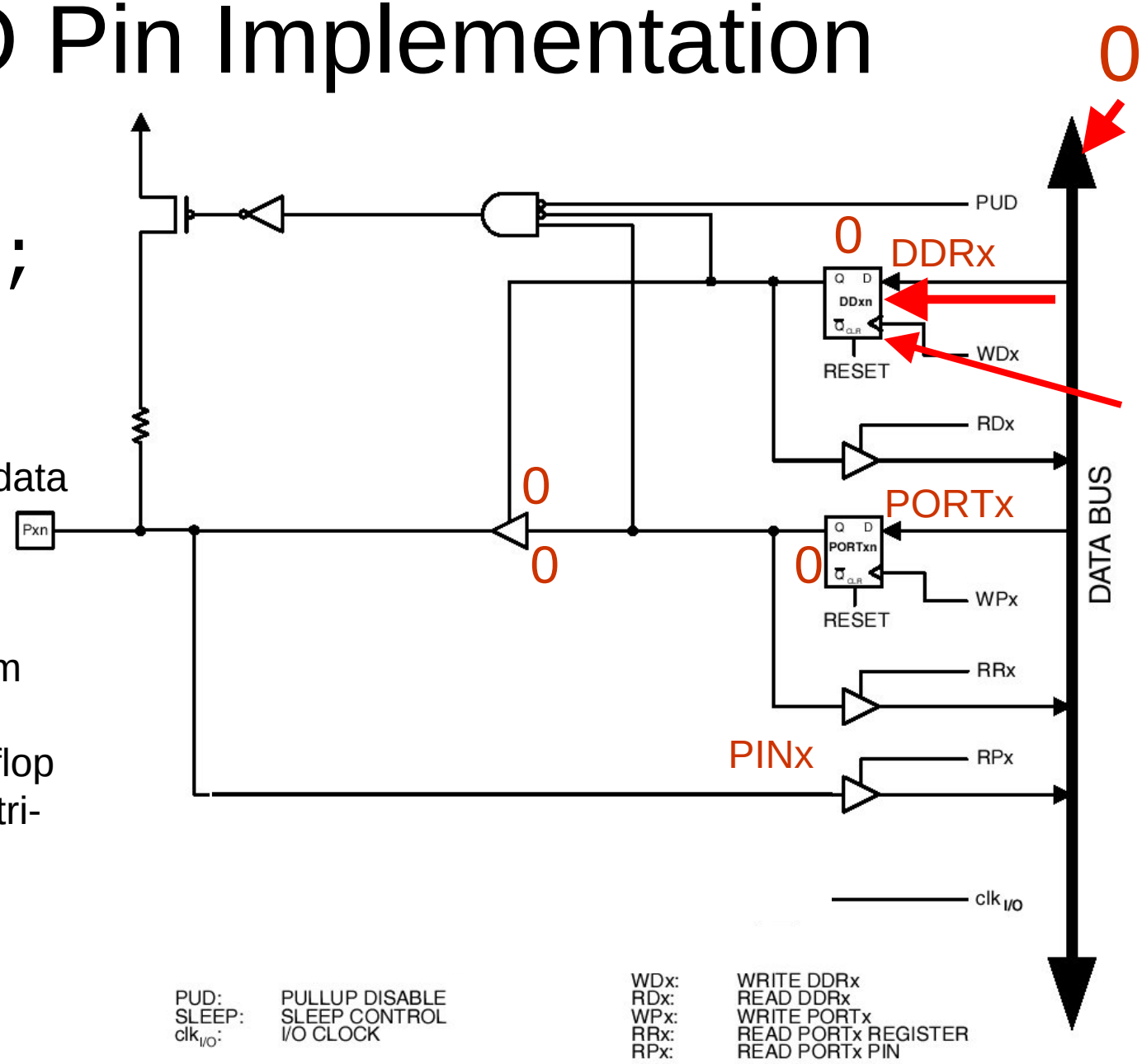


# I/O Pin Implementation

```
DDRB = 0;
```

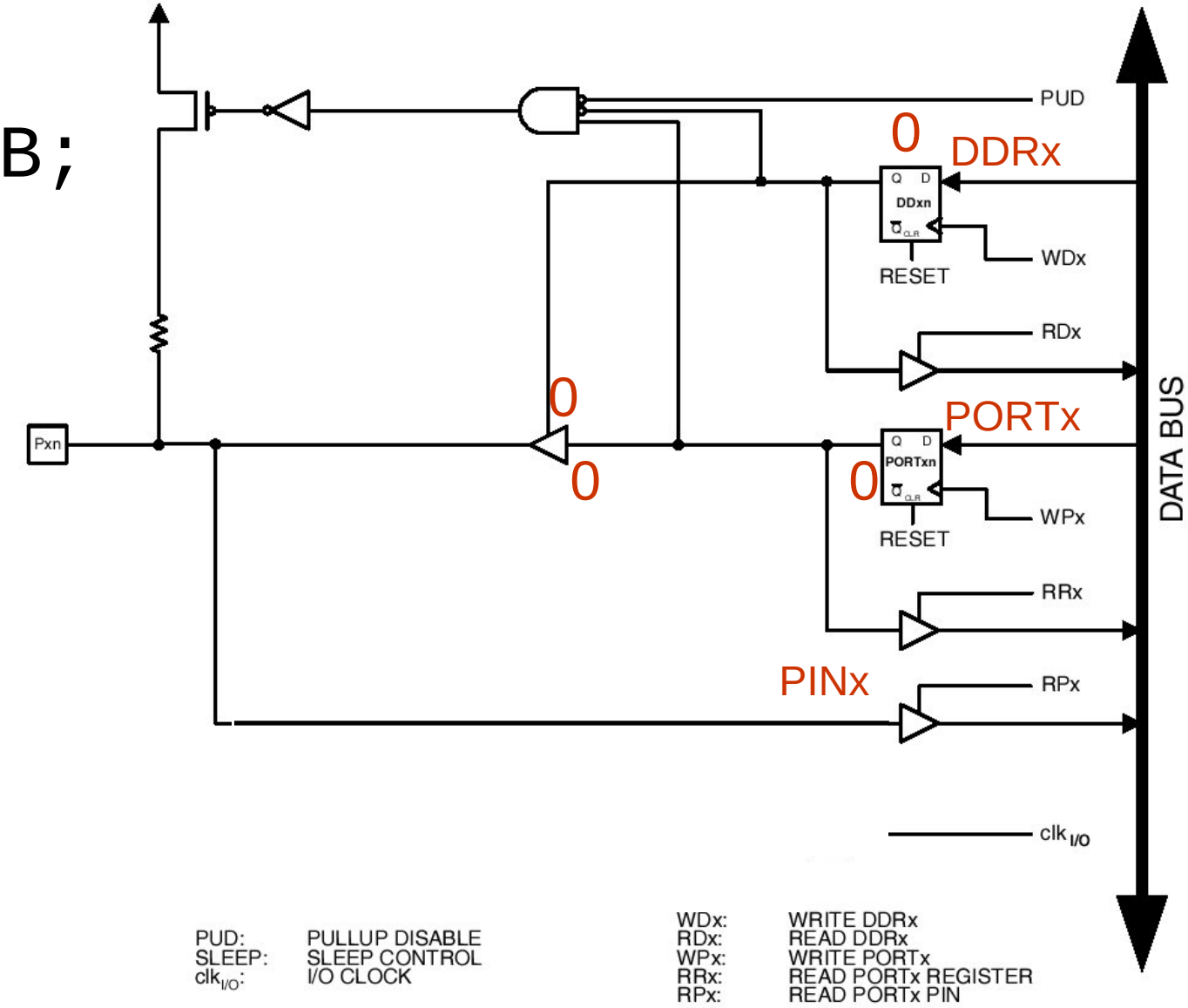
- “0” is written to the data bus
- This is input to the DDRB register
- WDB is clocked from high to low
- “0” is stored by flip-flop
- Which turns off the tri-state buffer

-> this is an input pin



# I/O Pin Implementation

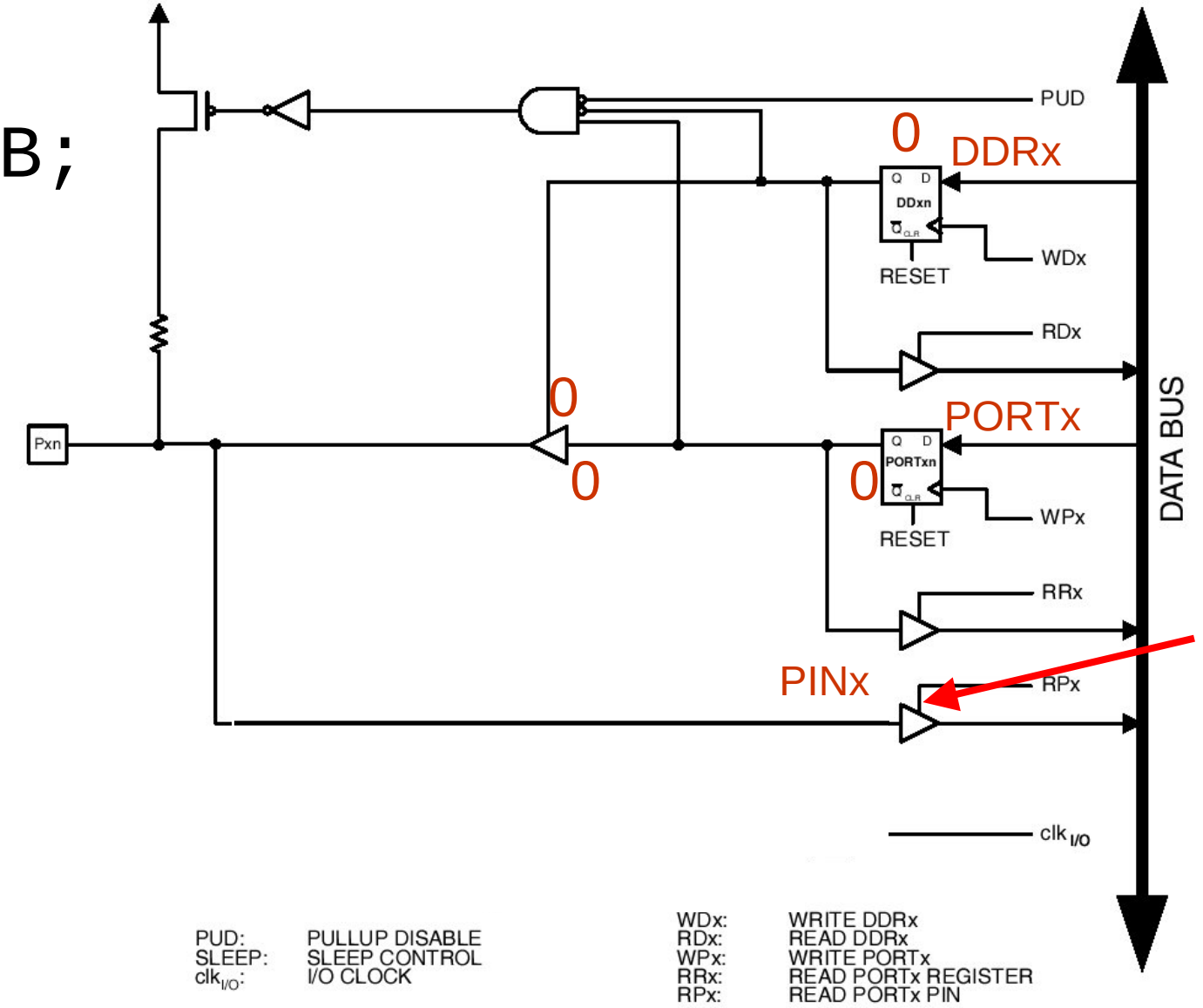
```
foo = PINB;
```



# I/O Pin Implementation

```
foo = PINB;
```

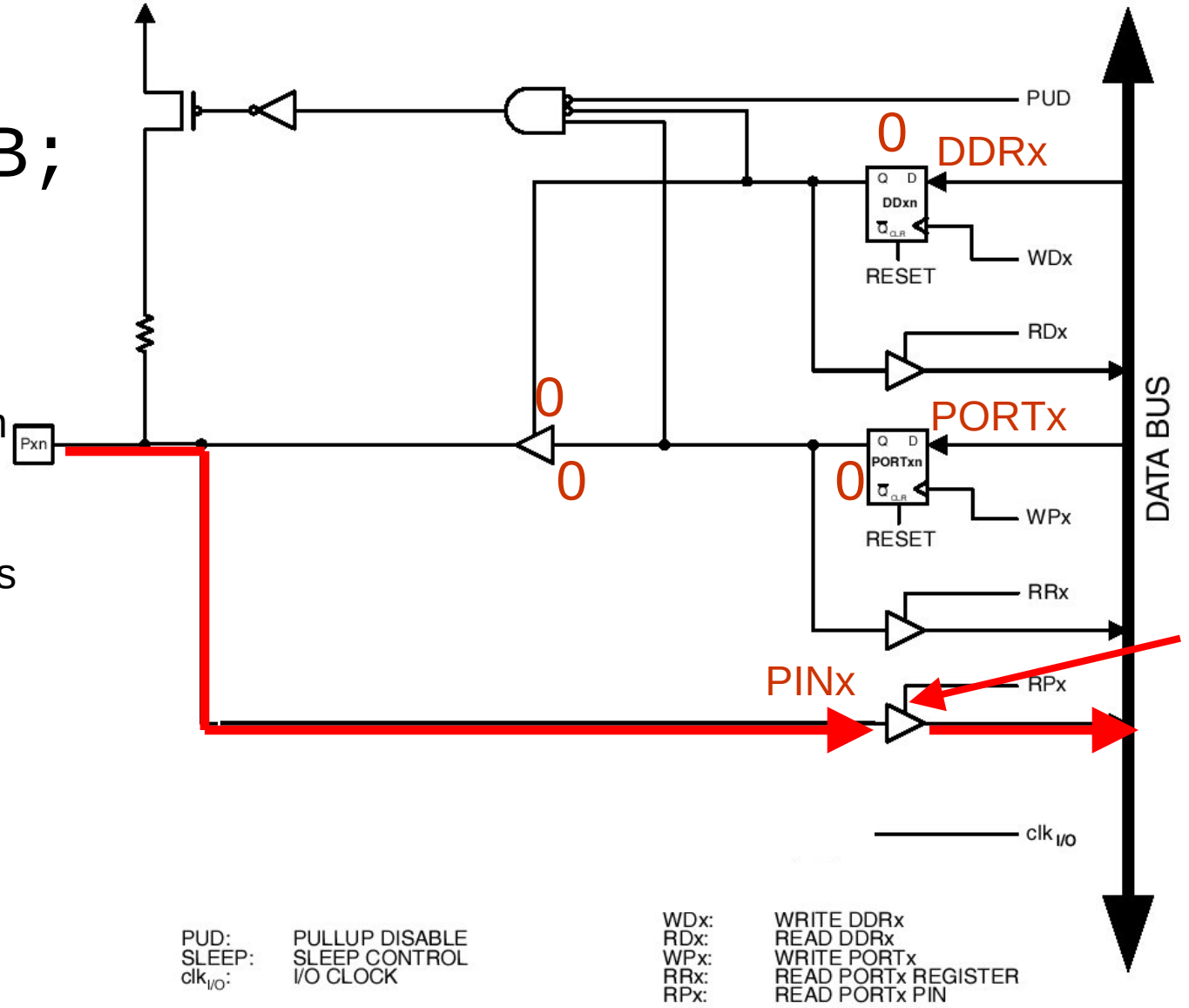
- RPB is set high



# I/O Pin Implementation

```
foo = PINB;
```

- RPB is clocked from high to low
- The pin state is copied to the data bus



# Bit Manipulation

PORTB is a register

- Controls the value that is output by the set of port B pins
- But – all of the pins are controlled by this single register (which is 8 bits wide)

- In code, we need to be able to manipulate the pins individually

# Bit-Wise Operators

If A and B are bytes, what does this code mean?

```
C = A & B;
```

The corresponding bits of A and B are ANDed together

# Bit-Wise Operators

If A and B are bytes, what does this code mean?

```
C = A & B;
```

# Bit-Wise Operators

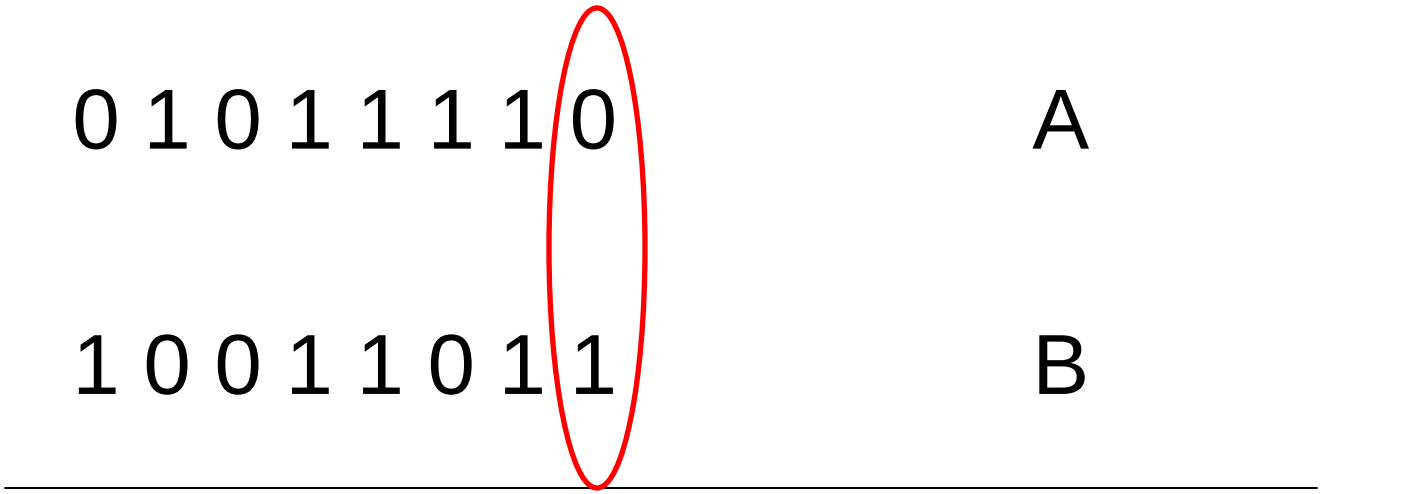
0 1 0 1 1 1 1 0 A

1 0 0 1 1 0 1 1 B

---

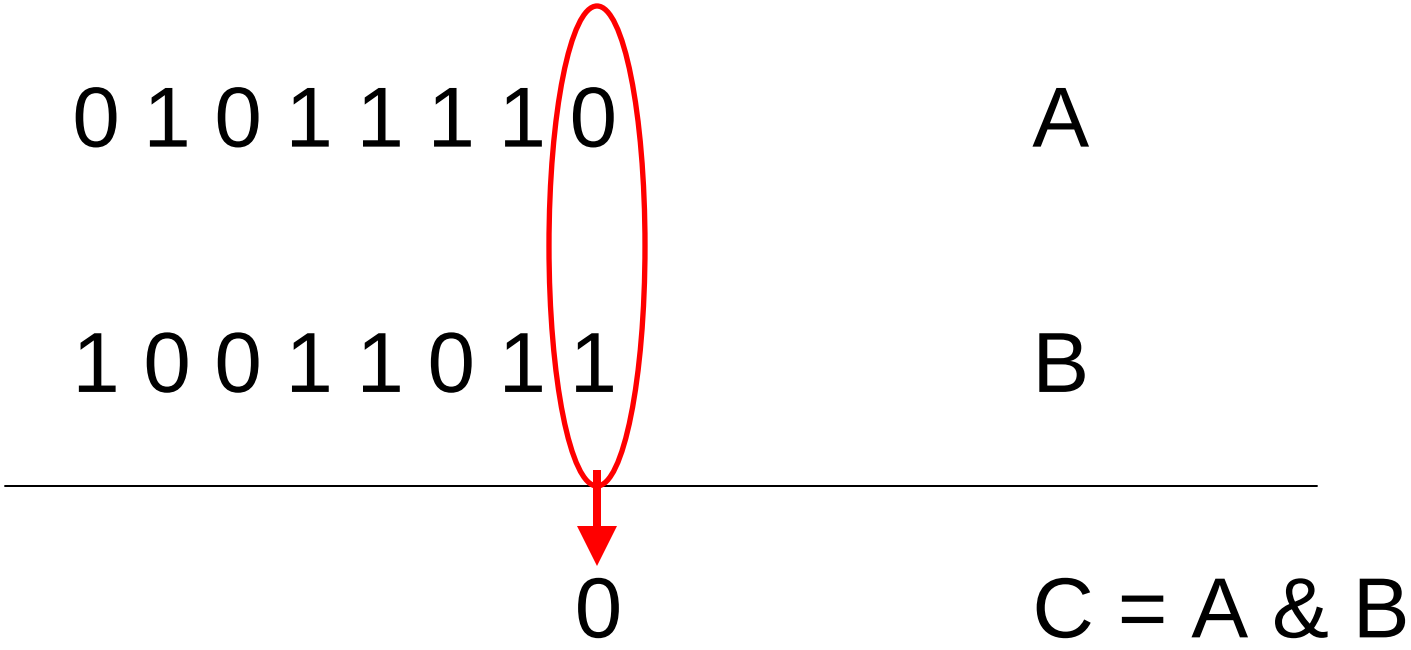
? C = A & B

# Bit-Wise Operators

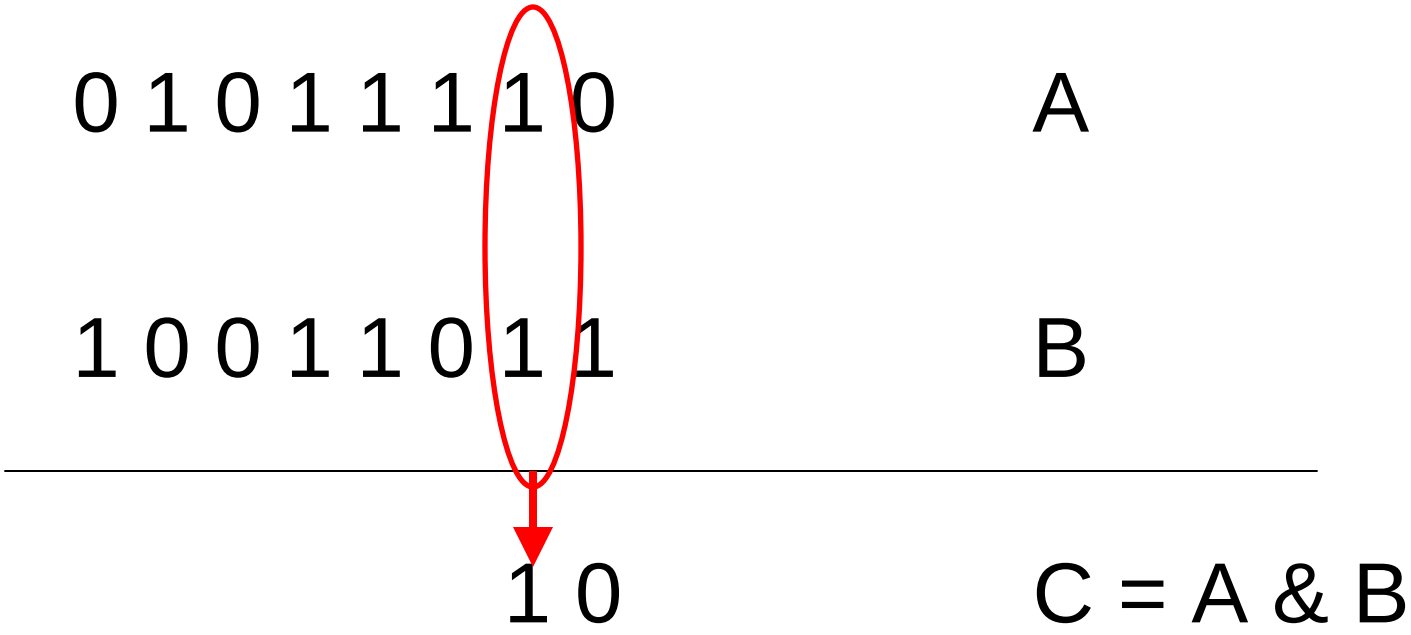



C = A & B

# Bit-Wise Operators



# Bit-Wise Operators



# Bit-Wise Operators

0 1 0 1 1 1 1 0 A

1 0 0 1 1 0 1 1 B

---

0 0 0 1 1 0 1 0 C = A & B

# Bit-Wise Operators

Other Operators:

- OR: |
- XOR: ^

# Bit Manipulation

Given a byte A, how do we set bit 2 (counting from 0) of A to 1?

# Bit Manipulation

Given a byte A, how do we set bit 2 (counting from 0) of A to 1?

```
A = A | 4;
```

# Bit Manipulation

Given a byte A, how do we set bit 2 (counting from 0) of A to 0?

# Bit Manipulation

Given a byte A, how do we set bit 2 (counting from 0) of A to 1?

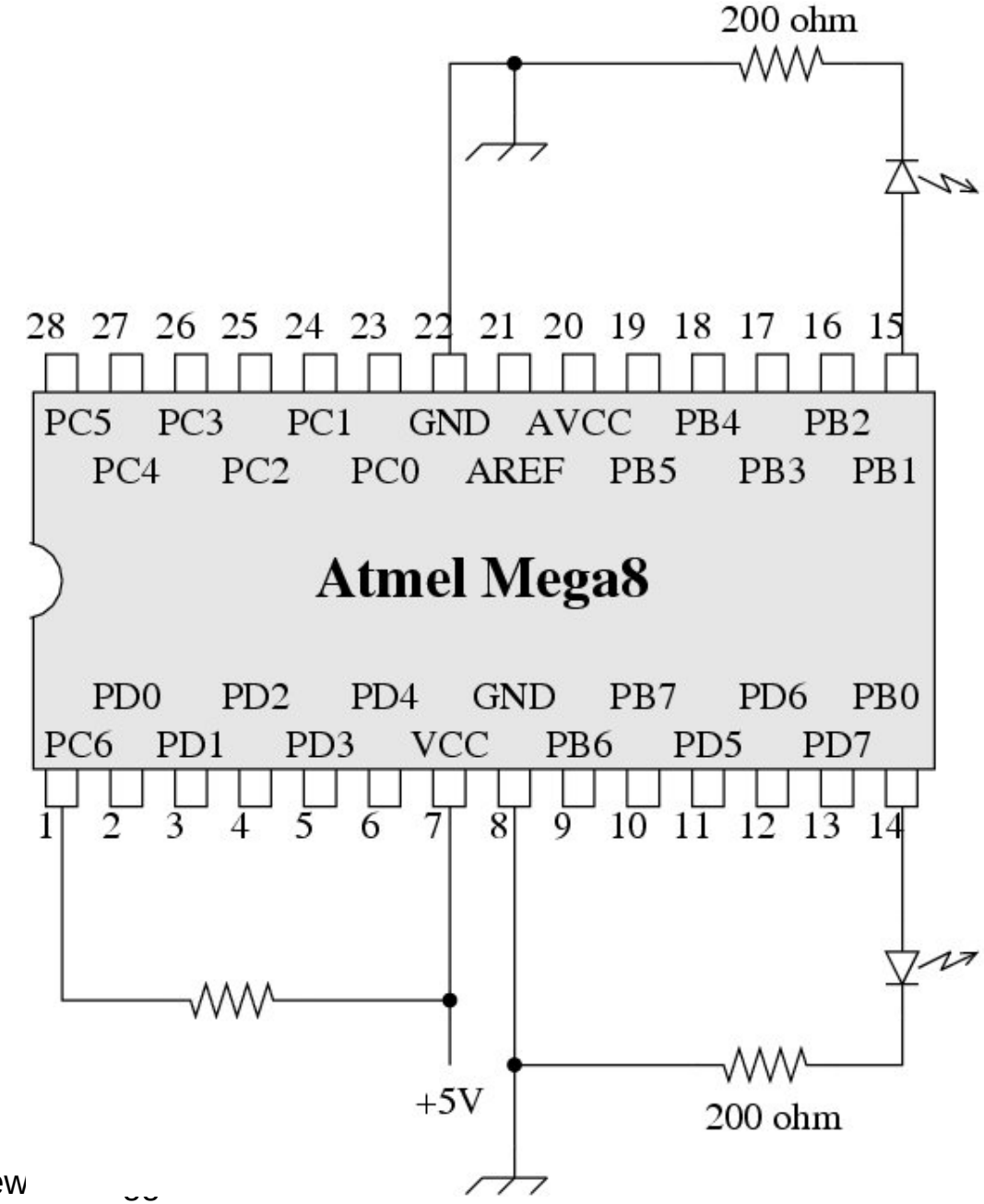
```
A = A & 0xFB;
```

```
A &= ~4;
```

# A First Program

Flash the LEDs at a regular interval

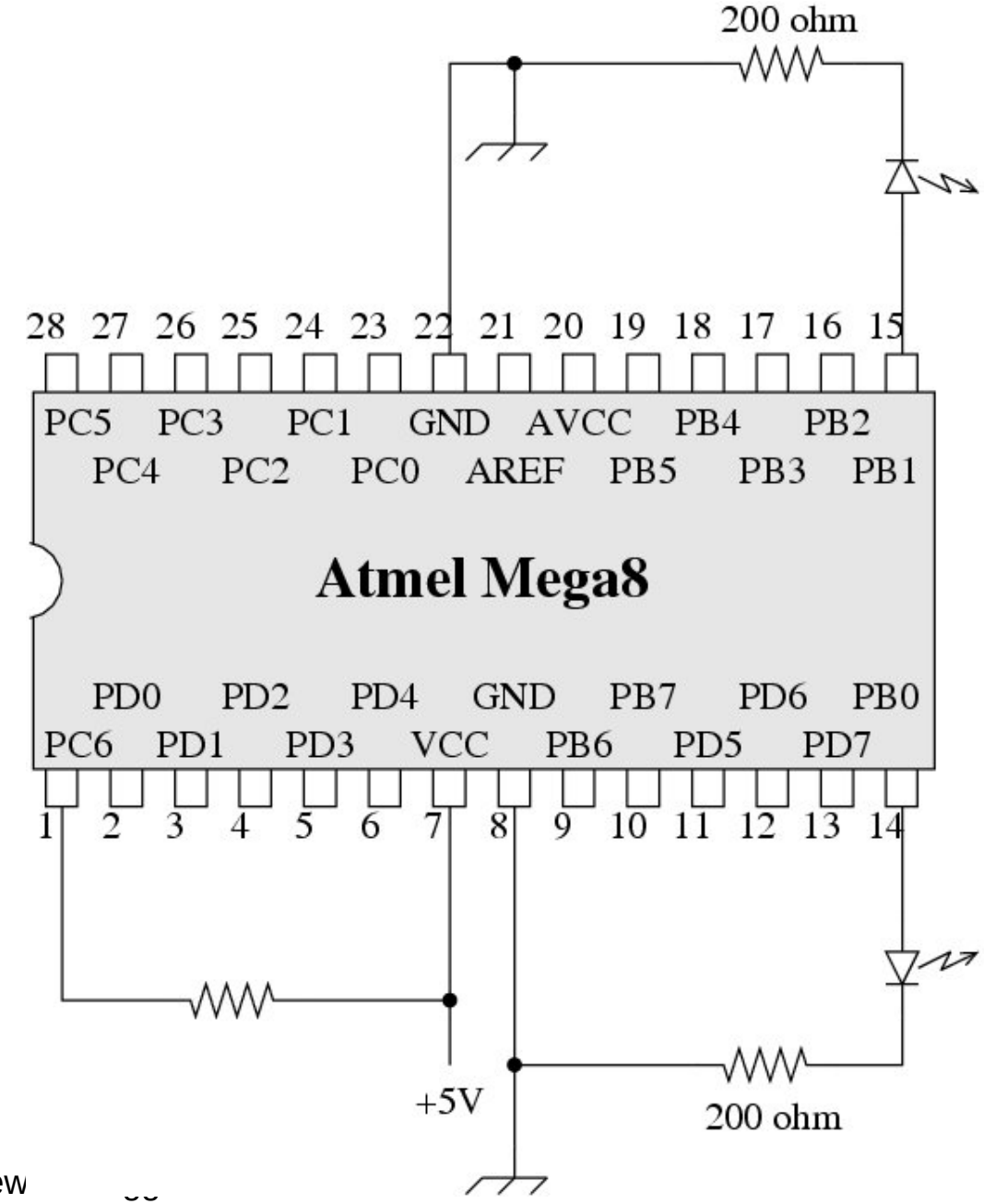
- How do we do this?



# A First Program

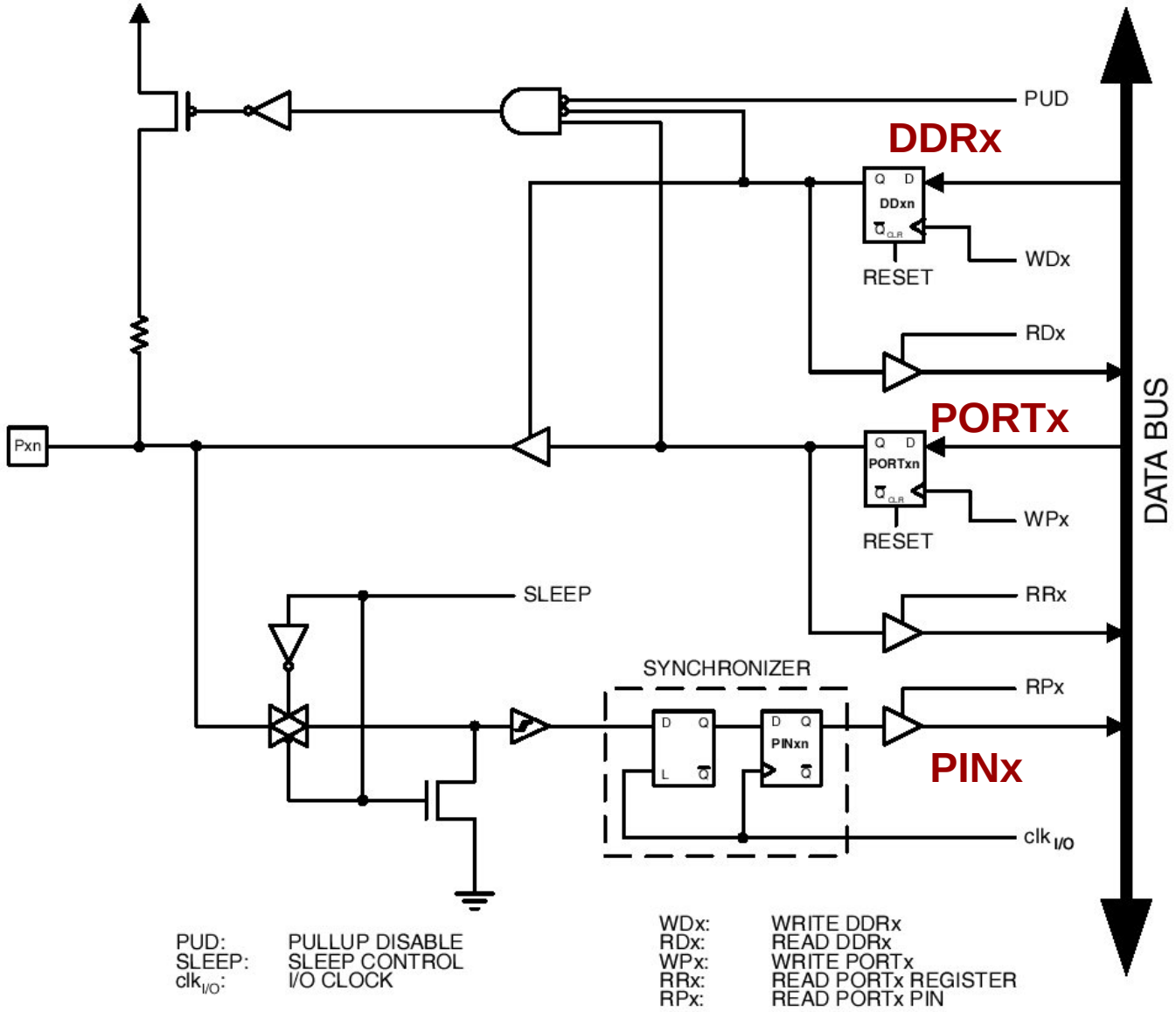
How do we flash the LED at a regular interval?

- We toggle the state of PB0



# I/O Pin Implementation

Single bit of PORT B



# A First Program

```
main() {
   DDRB = 0x1;   // Pin 0 to output

   while(1) {
       PORTB = PORTB ^ 0x1;   // XOR bit 0 with 1
       delay_ms(500);         // Pause for 500 msec
       }
}
```

# A Second Program

```
main() {
   DDRB = 0x3;   // Set all port B pins as outputs

   while(1) {
       PORTB = PORTB ^ 0x1;   // XOR bit 0 with 1
       delay_ms(500);         // Pause for 500 msec
       PORTB = PORTB ^ 0x2;   // XOR bit 1 with 1
       delay_ms(250);
       PORTB = PORTB ^ 0x2;   // XOR bit 1 with 1
       delay_ms(250);
   }
}
```

**What does this program do?**

# A Second Program

```
main() {
   DDRB = 0xFF;   // Set all port B pins as outputs

   while(1) {
       PORTB = PORTB ^ 0x1;   // XOR bit 0 with 1
       delay_ms(500);         // Pause for 500 msec
       PORTB = PORTB ^ 0x2;   // XOR bit 1 with 1
       delay_ms(250);
       PORTB = PORTB ^ 0x2;   // XOR bit 1 with 1
       delay_ms(250);
   }
}
```

**Flashes LED on PB1 at 1 Hz**
**on PB0: 0.5 Hz**

# More Bit Masking

- Suppose we have a 3-bit number (so values 0 … 7)
- Suppose we want to set the state of B3, B4, and B5 with this number (B3 is the least significant bit)

- How do we express this in code?

# Bit Masking

```
main() {
   DDRB = 0x38;   // Set pins B3, B4, B5  as outputs

        :
        :

  uint8_t val;  // A short is 8-bits wide

   val = command_to_robot;   // A value between 0 and 7

   PORTB = (PORTB & ~0x38)      // Set the current B3-B5 to 0s
       | ((val & 0x7)<<3);      // OR with new values (shifted
                                //  to fit within B3-B5
}
```

# Reading the Digital State of Pins

Given: we want to read the state of PB6 and PB7 and obtain a value of 0 … 3

- How do we configure the port?
- How do we read the pins?
- How do we translate their values into an integer of 0 .. 3?

# Reading the Digital State of Pins

```
main() {
   DDRB = 0x38;   // Set pins B3, B4, B5 as outputs
                  //   All others are inputs (suppose we care
                  //   about bits B6 and B7 only (so a 2-bit
                  //    number)
        :
        :

  unsigned short val, outval;  // A short is 8-bits wide

   val = PINB;

   outval = (val & 0xC0) >> 6;
}
```

# Port-Related Registers

The set of C-accessible register for controlling digital I/O:

| | Directional control | Writing | Reading |
|---|---|---|---|
| Port B | DDRB | PORTB | PINB |
| Port C | DDRC | PORTC | PINC |
| Port D | DDRD | PORTD | PIND |

# A Note About the C/Atmel Book

The book uses C syntax that looks like this:

```
PORTA.0 = 0;      // Set bit 0 to 0
```

This syntax is not available with our C compiler. Instead, you will need to use:

```
PORTA &= 0xFE;
```

or

```
PORTA &= ~1;
```

or

```
PORTA = PORTA & ~1;
```

# Putting It All Together

- Program development:
  - On your own laptop
  - We will use a C “crosscompiler” (avr-gcc and other tools) to generate code on your laptop for the mega8 processor
- Program download:
  - We will use “in circuit programming”: you will be able to program the chip without removing it from your circuit