Project 3 Adjustments

- Part 1 now due: April 11th (one week)
  - This is a “soft” deadline
- Part 2 now due: April 18th (two weeks)

(project 4 complexity will be adjusted accordingly)

Also: see discussion on D2L about types

Last Time

Digital to analog conversion

- Pulse-width modulation (PWM)
- Resistive networks

Analog to digital conversion

- Successive approximation
- Coding examples with the mega8

Project 3

Today: Input/Output

- I/O via polling
- Serial interfaces
- I/O with interrupts

Administrivia

- Homework 5 due today @5:00
- Project 2 demonstrations need to be completed by Tueday @3:30

Example: Stable Hovering

Lessons Learned from Lab 2

- Timing of sensory and control actions can be important
- Sensors and actuators are rarely ideal
  - Must account for this in our code
- Debugging can be a long process
  - Control the experiments
  - Implement and test in stages

Input/Output Systems

Processor needs to communicate with other devices:

- Receive signals from sensors
- Send commands to actuators
- Or both (e.g., disks, audio, video devices)

I/O Systems

Communication can happen in a variety of ways:

- Binary parallel signal (e.g., the interface that you used for your robot)
- Serial signals
- Analog

I/O Systems

Many devices are operating independently of the processor – except when communication happens

- We say that these devices are acting **asynchronously** of the processor
- The processor must have some way of knowing that something has changed with the device (e.g., that it is ready to send or receive information)

An Example: SICK Laser Range Finder

- Laser is scanned horizontally
- Using phase information, can infer the distance to the nearest obstacle (within a very narrow region)
- Spatial resolution: ~.5 degrees, 1 cm
- Can handle full 180 degrees at 20 Hz

I/O By Polling

One possible approach: the processor continually checks the state of the device:

```
do {
   x = PINB & 0x10;
}while(x == 0);
y = PINC …
```

I/O By Polling

What is wrong with this approach?

I/O By Polling

What is wrong with this approach?

- In embedded systems, we are typically managing many devices at once

I/O By Polling

- We can potentially be waiting for a long time before the state changes
  - We call this **busy waiting**
- The processor is wasting time that could be used to do other tasks

What is one way to solve this?

I/O By Polling: An Alternative

Alternative: do something while we are waiting

```
do {
   x = PINB & 0x10;
  <go do something else>
}while(x == 0);
y = PINC …
```

Serial Communication

- Communicate a set of bytes using a single signal line
- We do this by sending one bit at a time:
  - The value of the first bit determines the state of a signal line for a specified period of time
  - Then, the value of the 2nd bit is used
  - Etc.

Serial Communication

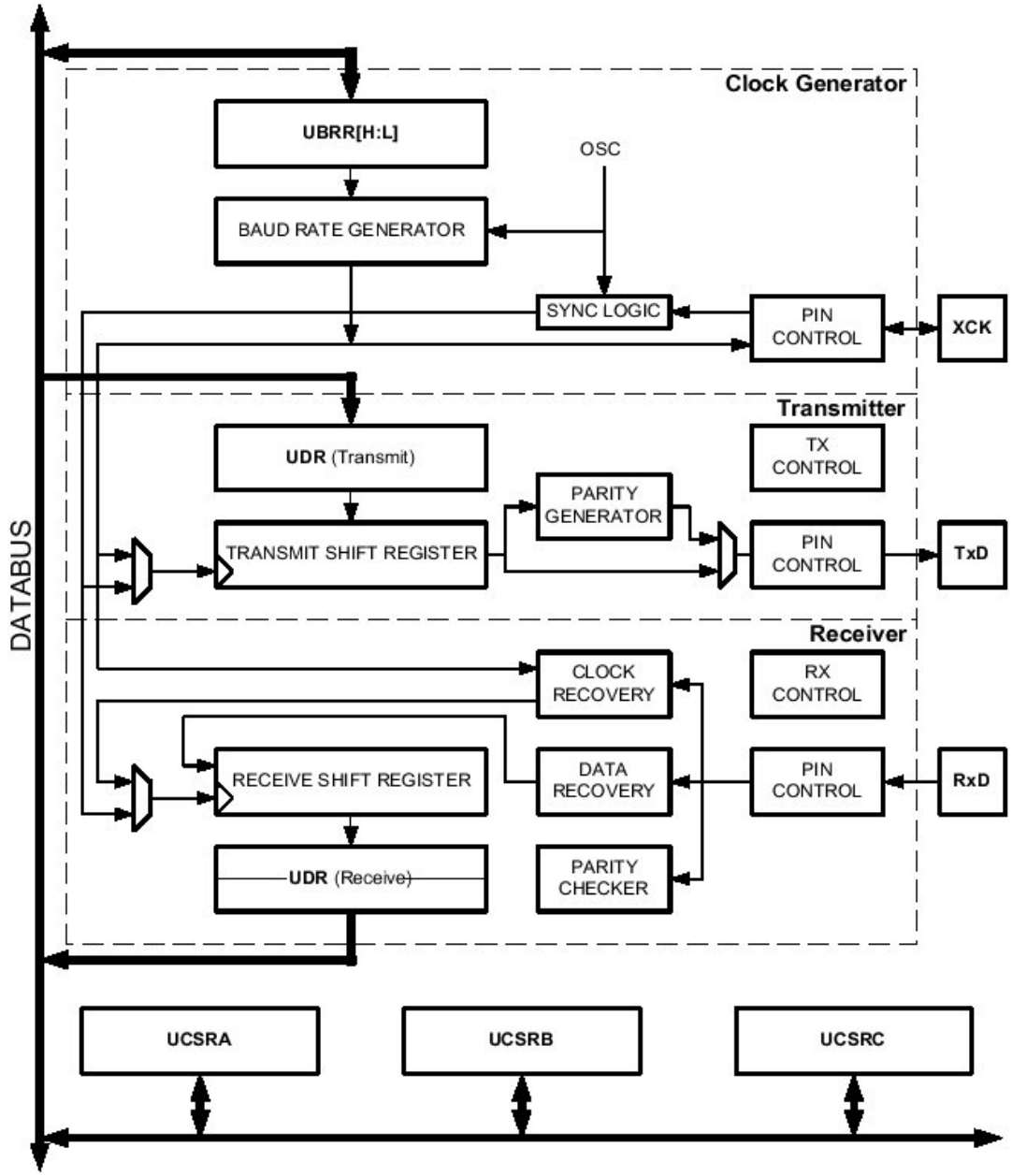
The sender and receiver must have some way of agreeing on when a specific bit is being sent

- Typically, each side has a clock to tell it when to write/read a bit
- In some cases, the sender will also send a clock signal (on a separate line)
- In other cases, the sender/receiver will first synchronize their clocks before transfer begins

Serial Communication

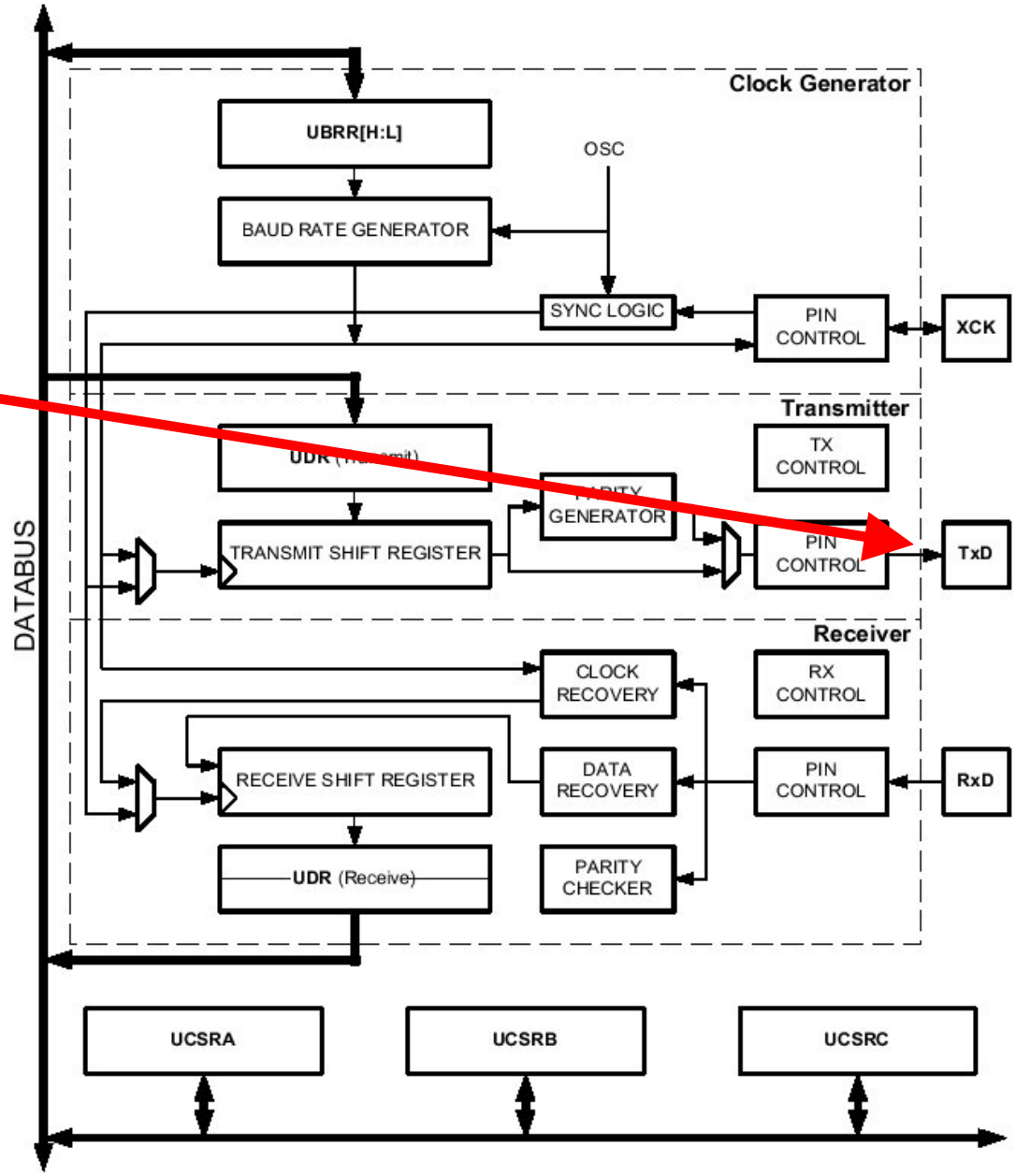
- Hardware implementations are very common:
  - Our mega 8 has a Universal, Asynchronous serial Receiver/Transmitter (UART)
  - Handles all of the bit-level manipulation
  - You only have to interact with it on the byte level

Mega8 UART


Clock Generator
UBRR[H:L]
OSC
BAUD RATE GENERATOR
SYNC LOGIC
PIN CONTROL
XCK
Transmitter
UDR (Transmit)
TX CONTROL
PARITY GENERATOR
TRANSMIT SHIFT REGISTER
PIN CONTROL
TxD
DATABUS
Receiver
CLOCK RECOVERY
RX CONTROL
RECEIVE SHIFT REGISTER
DATA RECOVERY
PIN CONTROL
RxD
UDR (Receive)
PARITY CHECKER
UCSRA
UCSRB
UCSRC

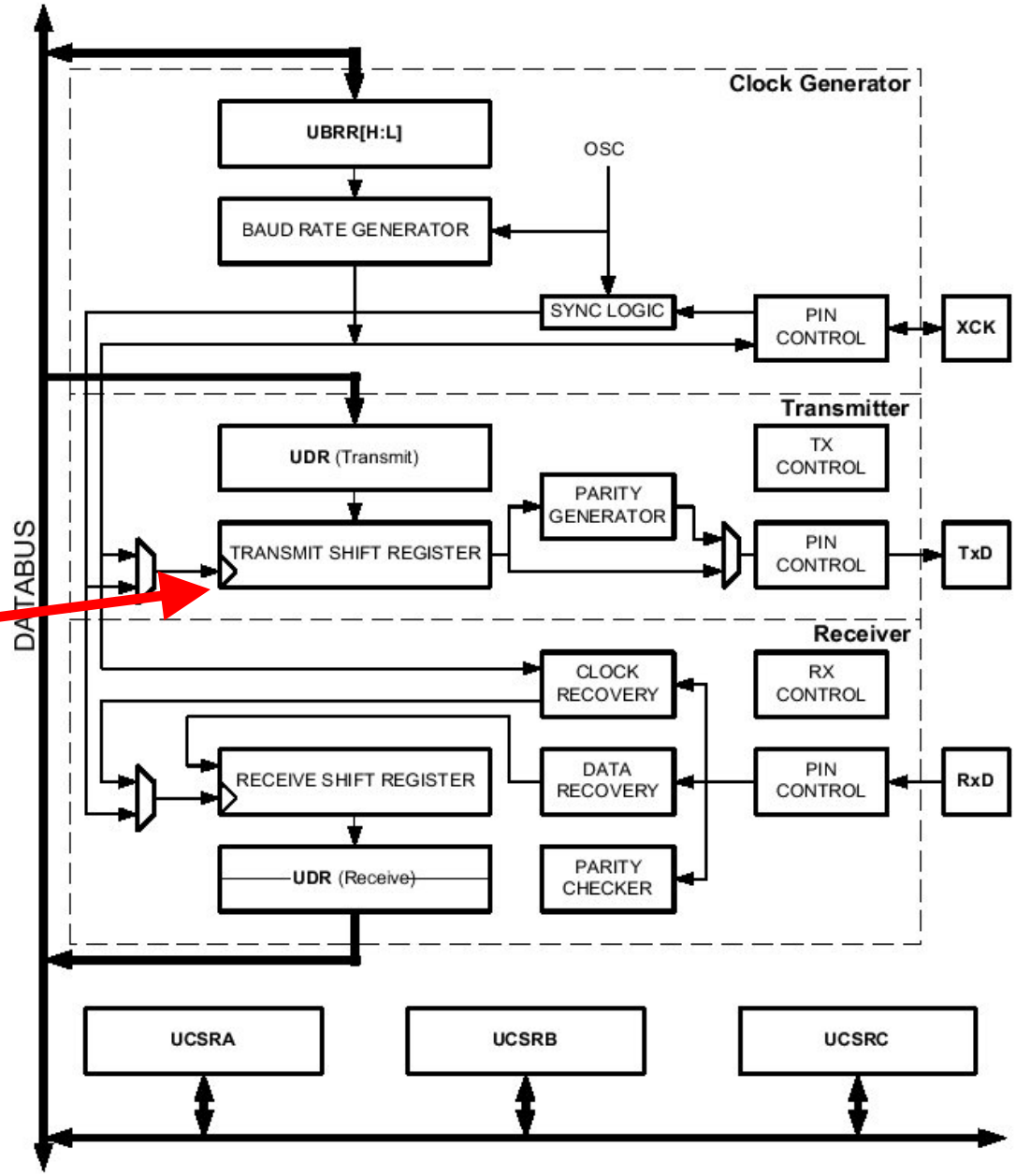
Mega8 UART

- Transmit pin (PD1)



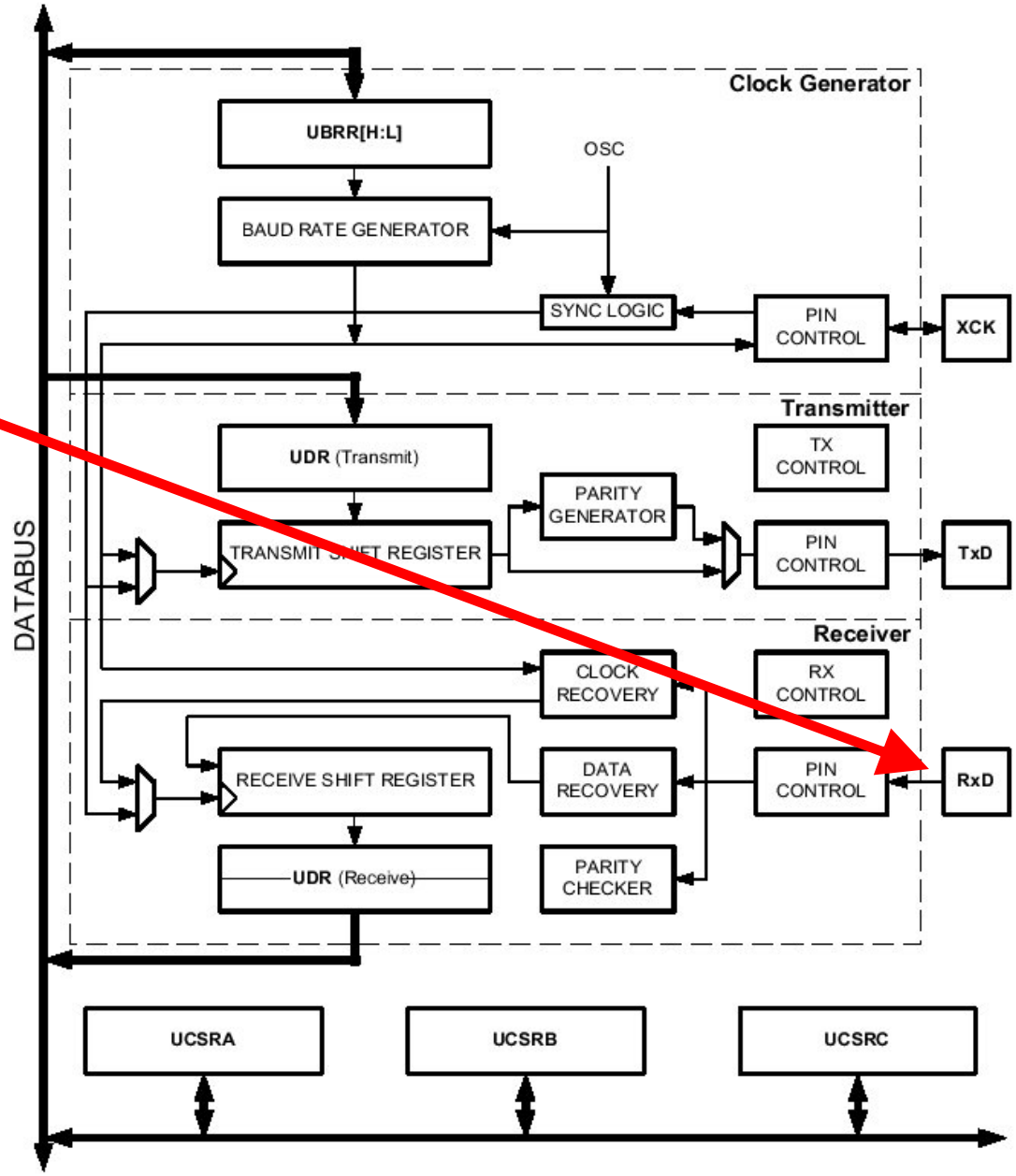
Mega8 UART

- Transmit pin (PD1)
- Transmit shift register



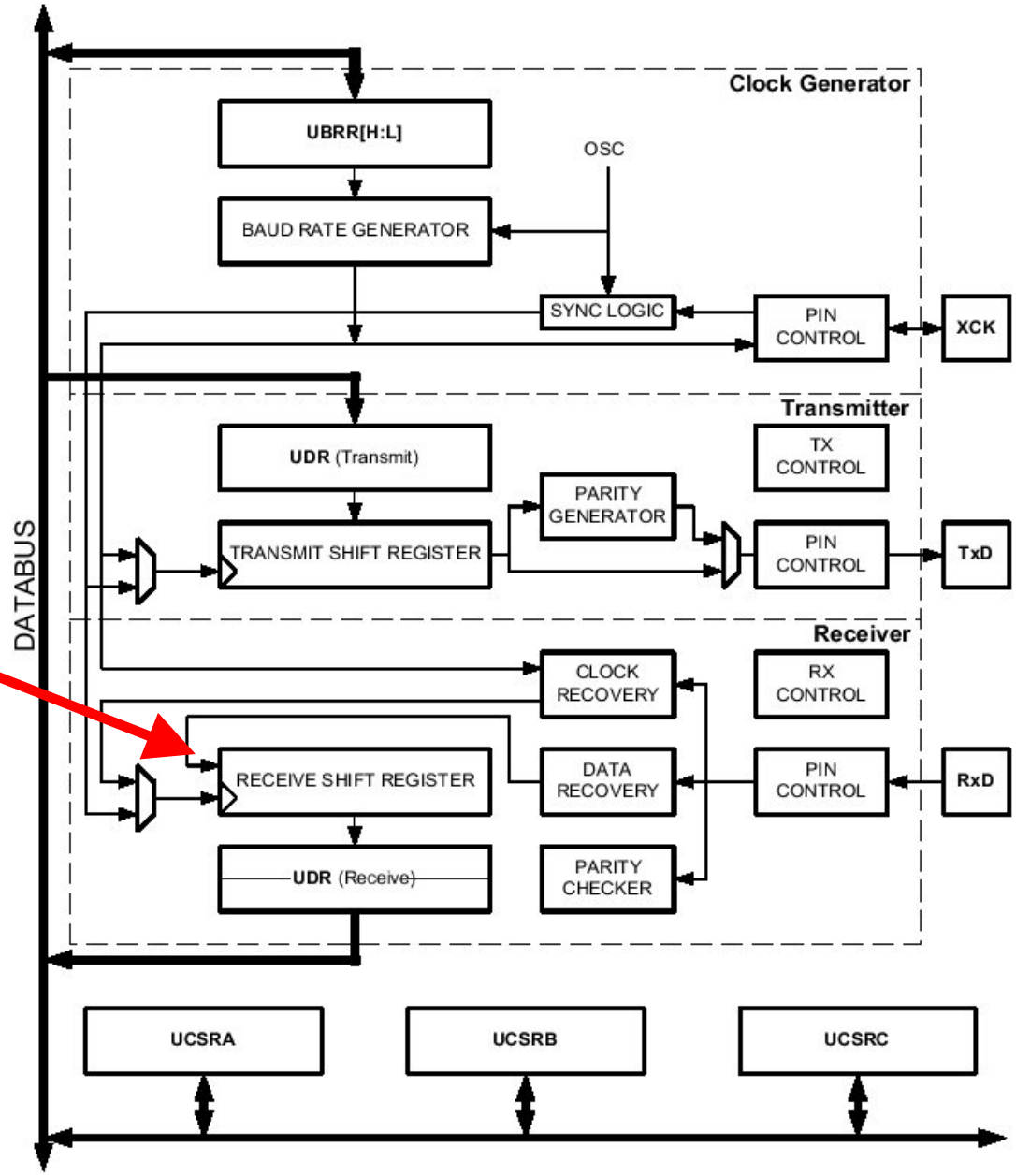
Mega8 UART

- Receive pin (PD0)



Mega8 UART

- Receive pin (PD0)
- Receive shift register



Mega8 UART C Interface

`ioinit()`: initialize the port

`getchar()`: receive a character

`kbhit()`: is there a character in the buffer?

`putchar()`: put a character out to the port

See the Atmel FAQ from the main class web page

Mega8 UART C Interface

`printf()`: formatted output

`scanf()`: formatted input

See the LibAvr documentation or the AVR C textbook

Serial I/O by Polling

```
int c;
while(1) {
  if(kbhit()) {
     // A character is available for reading
     c = getchar();
     <do something with the character>
  }
  <do something else while waiting>
}
```

I/O By Polling: An Alternative

Polling works great … but:

- We have to guarantee that our “something else” does not take too long (otherwise, we may miss the event)
- Depending on the device, “too long” may be very short

I/O by Polling

In practice, we typically reserve this polling approach for situations in which:

- We know the event is coming very soon
- We must respond to the event very quickly

(both are measured in nano- to micro-seconds)

An Alternative: Interrupts

- Hardware mechanism that allows some event to temporarily interrupt an ongoing task
- The processor then executes an **interrupt handler** (a small piece of code)
- Execution then continues with the original program

Some Sources of Interrupts (Mega8)

External:

- An input pin changes state
- The UART receives a byte on a serial input

Internal:

- A clock
- Processor reset
- The on-board analog-to-digital converter completes its conversion

Interrupts

There are many possible interrupts

- How do we know which one has occurred?
- How does the processor respond to a specific interrupt?

Interrupts

How do we know which interrupt has occurred?

- The mega8 hardware identifies each interrupt with a unique integer

How does the processor respond to a specific interrupt?

- The processor stores an **interrupt table** in program memory

Mega8 Interrupt Table Implementation

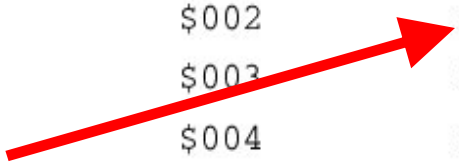
```
addressLabels Code                    Comments
$000          rjmp   RESET            ; Reset Handler
$001          rjmp   EXT_INT0         ; IRQ0 Handler
$002          rjmp   EXT_INT1         ; IRQ1 Handler
$003          rjmp   TIM2_COMP        ; Timer2 Compare Handler
$004          rjmp   TIM2_OVF         ; Timer2 Overflow Handler
$005          rjmp   TIM1_CAPT        ; Timer1 Capture Handler
$006          rjmp   TIM1_COMPA       ; Timer1 CompareA Handler
$007          rjmp   TIM1_COMPB       ; Timer1 CompareB Handler
$008          rjmp   TIM1_OVF         ; Timer1 Overflow Handler
$009          rjmp   TIM0_OVF         ; Timer0 Overflow Handler
$00a          rjmp   SPI_STC          ; SPI Transfer Complete Handler
$00b          rjmp   USART_RXC        ; USART RX Complete Handler
$00c          rjmp   USART_UDRE       ; UDR Empty Handler
$00d          rjmp   USART_TXC        ; USART TX Complete Handler
$00e          rjmp   ADC              ; ADC Conversion Complete Handler
$00f          rjmp   EE_RDY           ; EEPROM Ready Handler
$010          rjmp   ANA_COMP         ; Analog Comparator Handler
$011          rjmp   TWSI             ; Two-wire Serial Interface
```

Mega8 Interrupt Table Implementation

Address in the program memory

```
addressLabels Code                    Comments
$000          rjmp   RESET            ; Reset Handler
$001          rjmp   EXT_INT0         ; IRQ0 Handler
$002          rjmp   EXT_INT1         ; IRQ1 Handler
$003          rjmp   TIM2_COMP        ; Timer2 Compare Handler
$004          rjmp   TIM2_OVF         ; Timer2 Overflow Handler
$005          rjmp   TIM1_CAPT        ; Timer1 Capture Handler
$006          rjmp   TIM1_COMPA       ; Timer1 CompareA Handler
$007          rjmp   TIM1_COMPB       ; Timer1 CompareB Handler
$008          rjmp   TIM1_OVF         ; Timer1 Overflow Handler
$009          rjmp   TIM0_OVF         ; Timer0 Overflow Handler
$00a          rjmp   SPI_STC          ; SPI Transfer Complete Handler
$00b          rjmp   USART_RXC        ; USART RX Complete Handler
$00c          rjmp   USART_UDRE       ; UDR Empty Handler
$00d          rjmp   USART_TXC        ; USART TX Complete Handler
$00e          rjmp   ADC              ; ADC Conversion Complete Handler
$00f          rjmp   EE_RDY           ; EEPROM Ready Handler
$010          rjmp   ANA_COMP         ; Analog Comparator Handler
$011          rjmp   TWSI             ; Two-wire Serial Interface
```

Mega8 Interrupt Table Implementation



Change program counter to the location identified by "EXT_INT1"

```
addressLabels Code                    Comments
$000          rjmp   RESET            ; Reset Handler
$001          rjmp   EXT_INT0         ; IRQ0 Handler
$002          rjmp   EXT_INT1         ; IRQ1 Handler
$003          rjmp   TIM2_COMP        ; Timer2 Compare Handler
$004          rjmp   TIM2_OVF         ; Timer2 Overflow Handler
$005          rjmp   TIM1_CAPT        ; Timer1 Capture Handler
$006          rjmp   TIM1_COMPA       ; Timer1 CompareA Handler
$007          rjmp   TIM1_COMPB       ; Timer1 CompareB Handler
$008          rjmp   TIM1_OVF         ; Timer1 Overflow Handler
$009          rjmp   TIM0_OVF         ; Timer0 Overflow Handler
$00a          rjmp   SPI_STC          ; SPI Transfer Complete Handler
$00b          rjmp   USART_RXC        ; USART RX Complete Handler
$00c          rjmp   USART_UDRE       ; UDR Empty Handler
$00d          rjmp   USART_TXC        ; USART TX Complete Handler
$00e          rjmp   ADC              ; ADC Conversion Complete Handler
$00f          rjmp   EE_RDY           ; EEPROM Ready Handler
$010          rjmp   ANA_COMP         ; Analog Comparator Handler
$011          rjmp   TWSI             ; Two-wire Serial Interface
```

Interrupt Example

Suppose we are executing the "something else" code:

LDS R1 (A) ← **PC**

LDS R2 (B)

CP R2, R1

BRGE 3

LDS R3 (D)

ADD R3, R1

STS (D), R3

An Example

Suppose we are executing the "something else" code:

LDS R1 (A)

LDS R2 (B) ← **PC**

CP R2, R1

BRGE 3

LDS R3 (D)

ADD R3, R1

STS (D), R3

An Example

Suppose we are executing the "something else" code:

LDS R1 (A)

LDS R2 (B)

CP R2, R1 ← **PC**

BRGE 3

LDS R3 (D)

ADD R3, R1

STS (D), R3

An Example

An interrupt occurs (EXT_INT1):

LDS R1 (A)

LDS R2 (B)

CP R2, R1  **PC**

BRGE 3

LDS R3 (D)

ADD R3, R1

STS (D), R3

An Example

An interrupt occurs (EXT_INT1):

LDS R1 (A)

LDS R2 (B)

CP R2, R1

BRGE 3

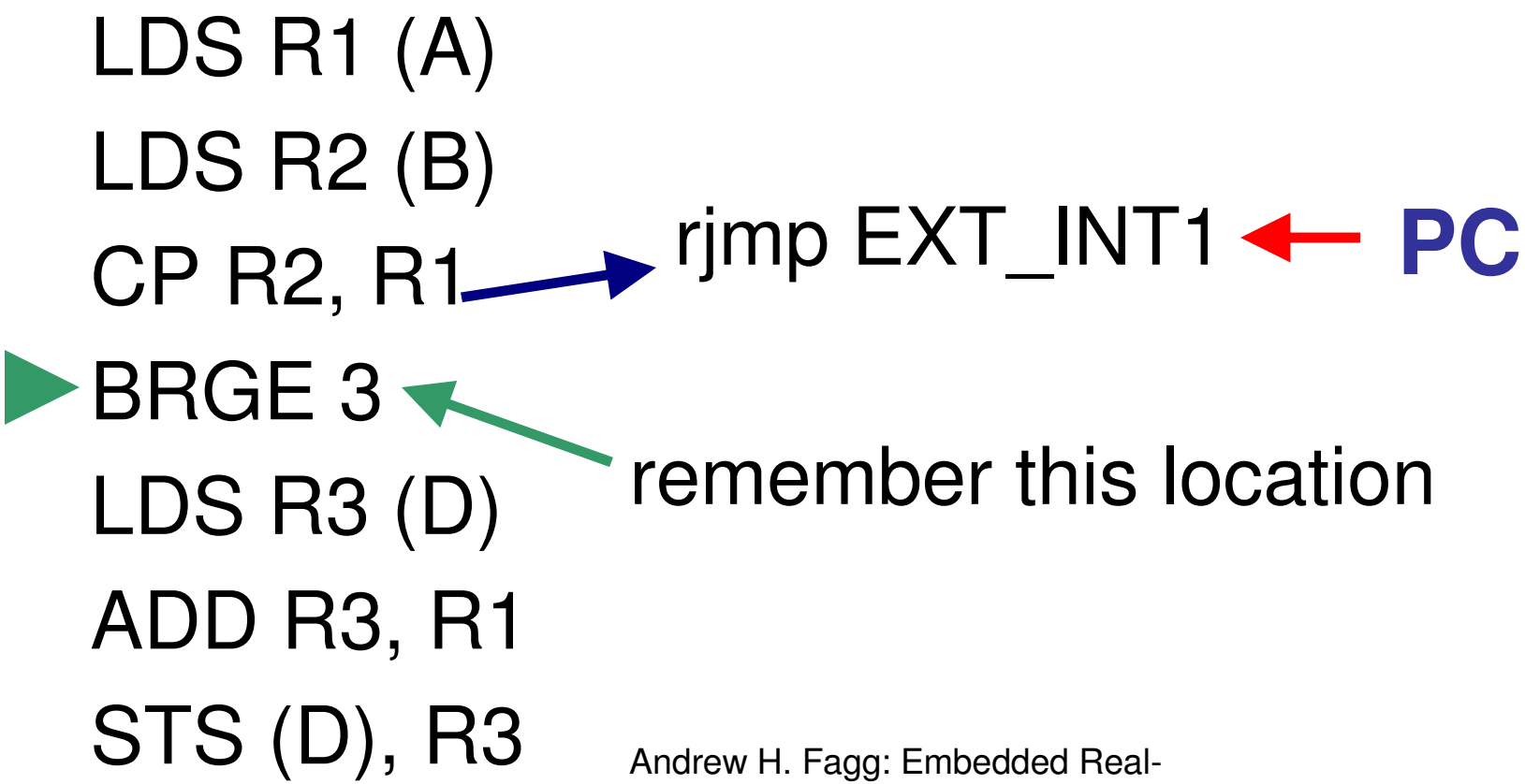
LDS R3 (D)

ADD R3, R1

STS (D), R3

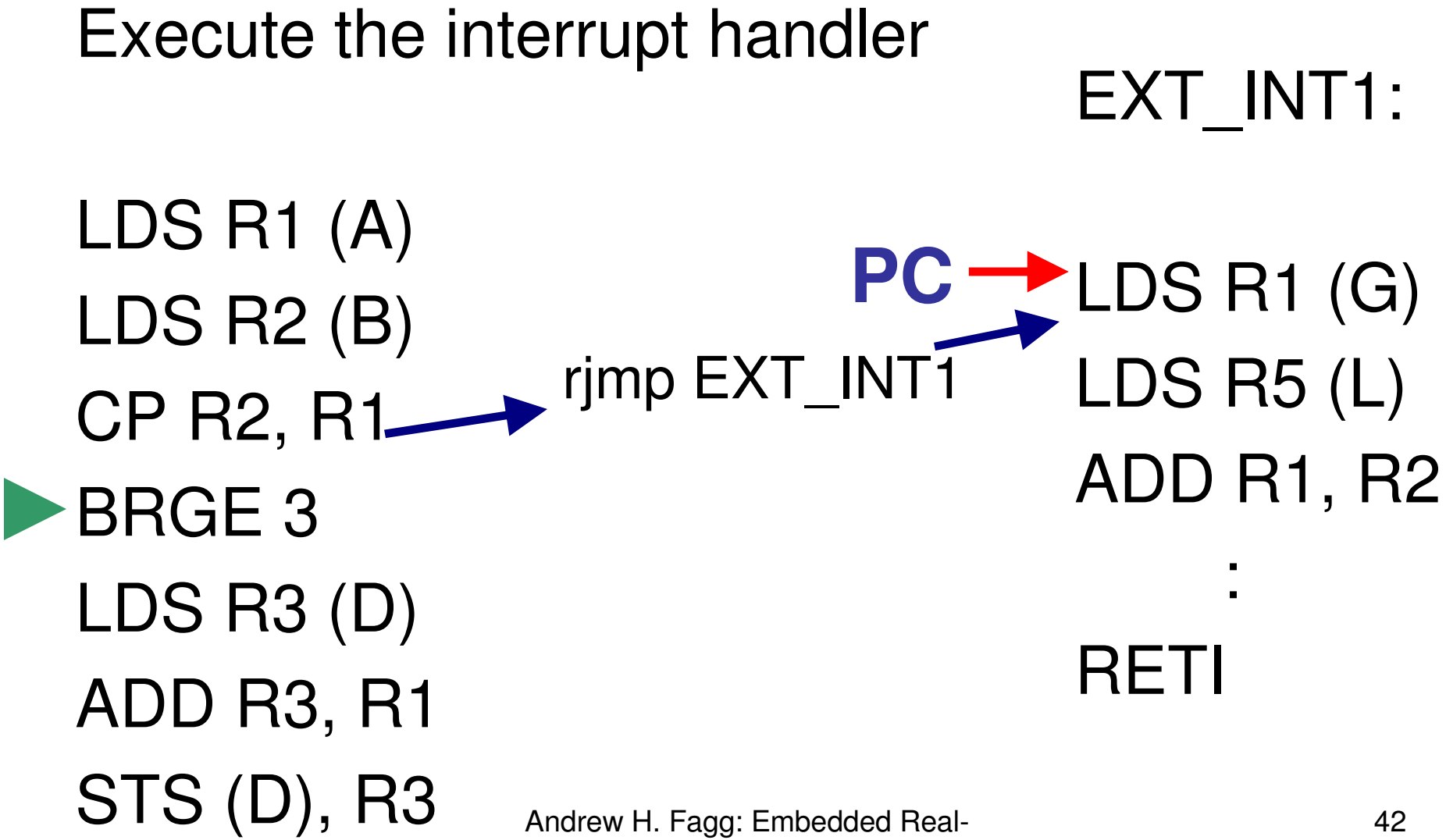
An Example

An interrupt occurs (EXT_INT1):



An Example

Execute the interrupt handler



An Example

Execute the interrupt handler

LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3

EXT_INT1:

LDS R1 (G)
PC → LDS R5 (L)
ADD R1, R2
:
RETI

An Example

Execute the interrupt handler

LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3

EXT_INT1:

LDS R1 (G)
LDS R5 (L)
PC → ADD R1, R2
:
RETI

An Example

Execute the interrupt handler

LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3

EXT_INT1:

LDS R1 (G)
LDS R5 (L)
ADD R1, R2
:
RETI

PC →

An Example

Return from interrupt

```
LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3
```

```
EXT_INT1:

LDS R1 (G)
LDS R5 (L)
ADD R1, R2
   :
RETI
```

PC →

An Example

Return from interrupt

LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3

EXT_INT1:

LDS R1 (G)
LDS R5 (L)
ADD R1, R2
:
RETI

An Example

Continue execution with original

```
LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)   <-- PC
ADD R3, R1
STS (D), R3
```

```
EXT_INT1:

LDS R1 (G)
LDS R5 (L)
ADD R1, R2
    :
RETI
```

An Example

Continue execution with original

```
LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3
```

PC

```
EXT_INT1:

LDS R1 (G)
LDS R5 (L)
ADD R1, R2
    :
RETI
```

Interrupt Routines

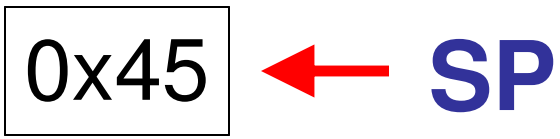
- Generally a very small number of instructions
  - We want a quick response so the processor can return to what it was originally doing
- Register use
  - If the interrupt routine makes use of registers, then it must restore their state before returning
  - We accomplish this through the use of a **stack**

The Stack

A hardware-supported data structure composed of:

- A block of memory
- A stack pointer (SP) that indicates the current top of the stack

The Stack (an example)

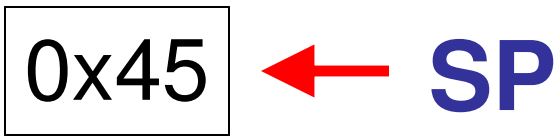


The Stack (an example)

Operation:

```
PUSH R1
```

(assume R1 contains 0x31)

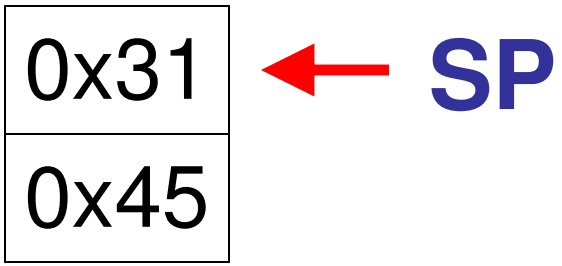


The Stack (an example)

Operation:

```
PUSH R1
```

(assume R1 contains 0x31)

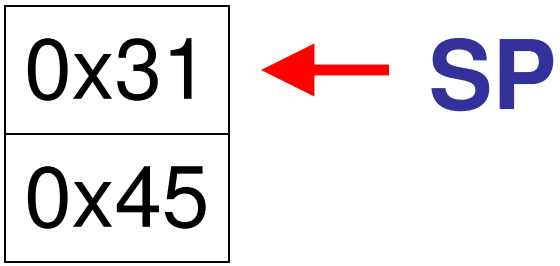


The Stack (an example)

Now perform:

```
PUSH R5
```

(assume R5 contains 0xF3)

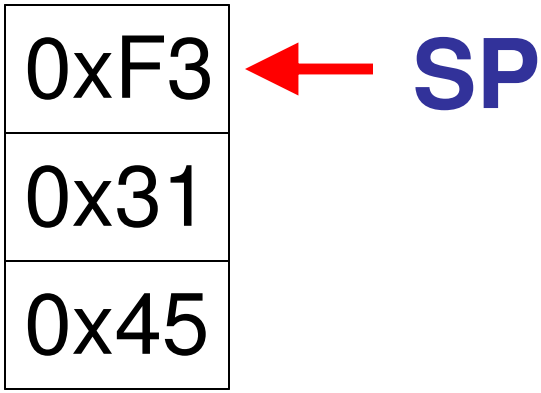


The Stack (an example)

Now perform:

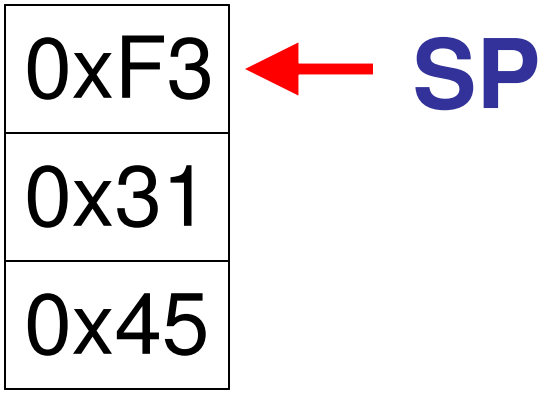
```
PUSH R5
```

(assume R5 contains 0xF3)



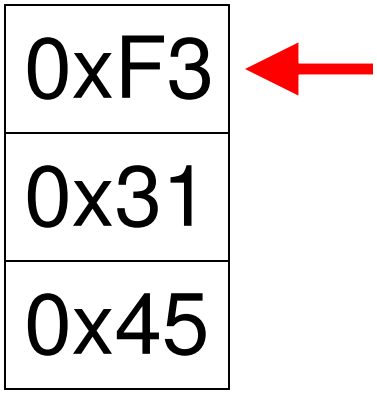
The Stack (an example)

The interrupt routine (or function) now performs its job …



The Stack (an example)

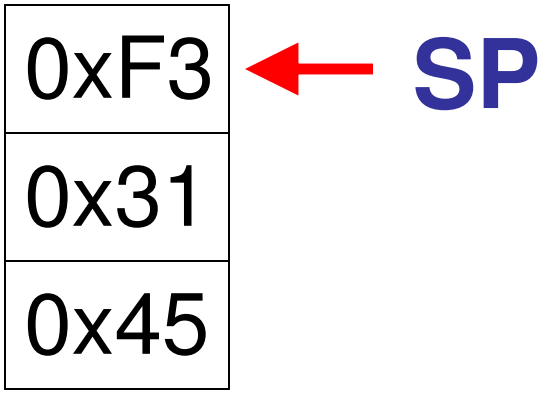
The interrupt routine (or function) now performs its job (changing R1 and R5)… and now restores the state of R5 and R1 …



The Stack (an example)

Now perform:

```
POP R5
```

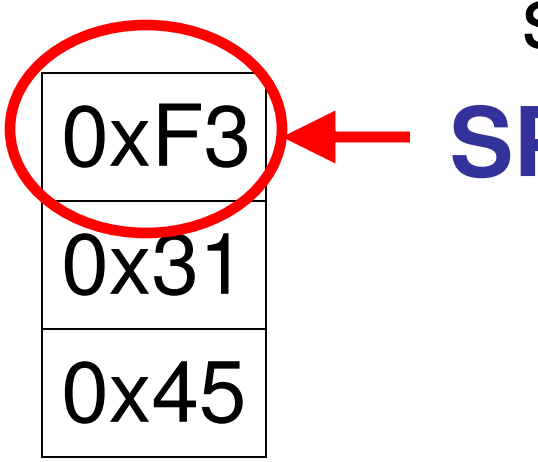


The Stack (an example)

Now perform:

```
POP R5
```

R5 now is set to the value that is on the top of the stack (0xF3) …

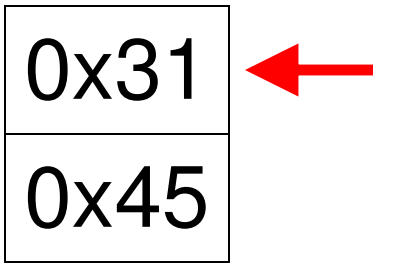


The Stack (an example)

Now perform:

```
POP R5
```

R5 now is set to the value that is on the top of the stack (0xF3) … and the stack pointer is incremented

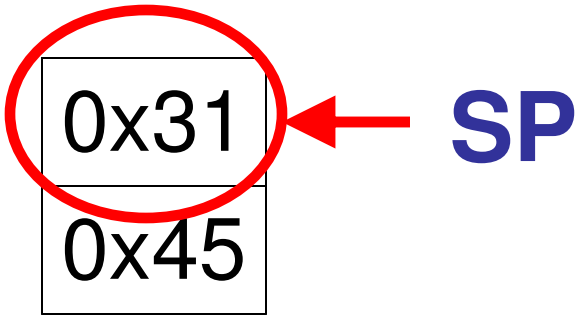


The Stack (an example)

Now perform:

```
POP R1
```

R1 receives the value on the top of the stack (0x31)

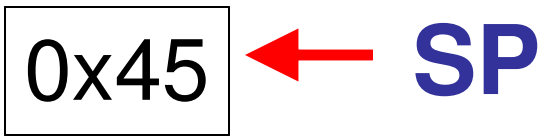


The Stack (an example)

Now perform:

```
POP R1
```

R1 receives the value on the top of the stack (0x31) and the SP is incremented



The Stack

In addition to the temporary storage of register values, the stack is also used to:

- Pass parameters to a function
- Store the return location for use after an interrupt or a function call
- Store the value of the status register

Stack Manipulation in the Mega8

In the Mega8 and with our gcc compiler:

- Stack manipulation is typically hidden from us
- This is true for functions as well as interrupt routines

Last Time

- I/O by polling
  - Can lead to wasted CPU time due to “busy waiting”
  - Can miss events if you don’t check for them often enough
- Interrupts
  - Temporarily stop what the processor is doing
  - Execute a small “interrupt handler” function
  - Return the processor to its original state and keep executing as if nothing else has happened

Last Time

Stack: location in memory for temporary storage

- Save register states
- Save return location (so we know where to come back after a function or an interrupt
- Pass function parameters

Today

- Interrupt handler example
  - Dealing with large volumes of incoming data
- Hardware timers and associated interrupts
  - Will allow us to precisely time the regular execution of certain interrupt handlers

Administrivia

- Project 3:
  - Soft deadline for part #1 (encoder processing) is due on Tuesday
- 4 Robots are working well now
  - Left/right turning asymmetry has been fixed

Back to Receiving Serial Data…

```
int c;
while(1) {
  if(kbhit()) {
     // A character is available for reading
     c = getchar();
     <do something with the character>
  }
  <do something else while waiting>
}
```

With this solution, how long can “something else” take?

Receiving Serial Data

How can we allow the “something else” to take a longer period of time?

Receiving Serial Data

How can we allow the “something else” to take a longer period of time?

- The UART implements a 1-byte buffer
- Let’s create a larger buffer…

Receiving Serial Data

Creating a larger buffer. This will be a globally-defined data structure composed of:

- N-byte memory space:

```
char buffer[BUF_SIZE];
```

- Integers that indicate the first element in the buffer and the number of elements:

```
int front, nchars;
```

Buffered Serial Data

Implementation:

- We will use an interrupt routine to transfer characters from the UART to the buffer as they become available

- Then, our main() function can remove the characters from the buffer

Interrupt Handler

```
// Called when the UART receives a byte
SIGNAL(SIG_UART_RECV) {
  // Handle the character in the UART buffer

  }
}
```

Interrupt Handler

```
// Called when the UART receives a byte
SIGNAL(SIG_UART_RECV) {
  // Handle the character in the UART buffer
  int c = getchar();

  if(nchars < BUF_SIZE) {
     buffer[(front+nchars)%BUF_SIZE] = c;
     nchars += 1;
  }
}
```

Reading Out Characters

```
// Called by a "main" program
// Get the next character from the
  circular buffer
int get_next_character() {
  int c;



  }
}
```

Reading Out Characters

```
// Called by a "main" program
// Get the next character from the circular buffer
int get_next_character() {
   int c;
   if(nchars == 0)
       return(-1); // Error
   else {
       // Pull out the next character
       c = buffer[front];

       // Update the state of the buffer
       --nchars;
       front = (front + 1)%BUF_SIZE;
       return(c);
   }
}
```

An Updated main()

```
int c;
while(1) {
  do {
     ????

  }while(???);
  <do something else while waiting>
}
```

An Updated main()

```
int c;
while(1) {
  do {
     c = get_next_character();
     if(c != -1)
          <do something with the character>
  }while(c != -1);

  <do something else while waiting>
}
```

Buffered Serial Data

This implementation captures the essence of what we want, but there are some subtle things that we must handle ….

Buffered Serial Data

Subtle issues:

- The reading side of the code must make sure that it does not allow the buffer to overflow
  - But at least we have BUF_SIZE times more time
- We have a shared data problem …

The Shared Data Problem

- Two independent segments of code that could access the same data structure at arbitrary times

- In our case, get_next_character() could be interrupted while it is manipulating the buffer
  - This can be very bad

Solving the Shared Data Problem

- There are segments of code that we want to execute without being interrupted

- We call these code segments **critical sections**

Solving the Shared Data Problem

There are a variety of techniques that are available:

- Clever coding
- Hardware: test-and-set instruction
- Semaphores: software layer above test-and-set
- Disabling interrupts

Disabling Interrupts

- How can we modify get_next_character()?

- The it is important that the critical section be as short as possible

Assume:

- serial_receive_enable(): enable interrupt flag
- serial_receive_disable(): clear (disable) interrupt flag

Modified get_next_character()

```
int get_next_character() {
   int c;
   serial_receive_disable();
   if(nchars == 0)
       serial_receive_enable();
       return(-1); // Error
   else {
       // Pull out the next character
       c = buffer[front];
       --nchars;
       front = (front + 1)%BUF_SIZE;
       serial_receive_enable();
       return(c);
   }
}
```

Initialization Details

```
main()
{
  nchars = 0;
  front = 0;

  // Enable UART receive interrupt
  serial_receive_enable();

  // Enable global interrups
  sei();
      :
```

Enabling/Disabling Interrupts

- Enabling/disabling interrupts allows us to ensure that a specific section of code (the critical section) cannot be interrupted
  - This allows for safe access to shared variables

- But: must not disable interrupts for a very long time

Last Time

- Interrupts in practice
- Serial data processing
- Data buffering
- Shared data problem

Today

- Timers/counters
- Generating regular interrupts
- Direct Memory Access (DMA)

Administrivia

- Should have part 1 of project 3 demonstrated today
- Homework 5 and project 2 grading done for Thursday

Counter/Timers in the Mega8

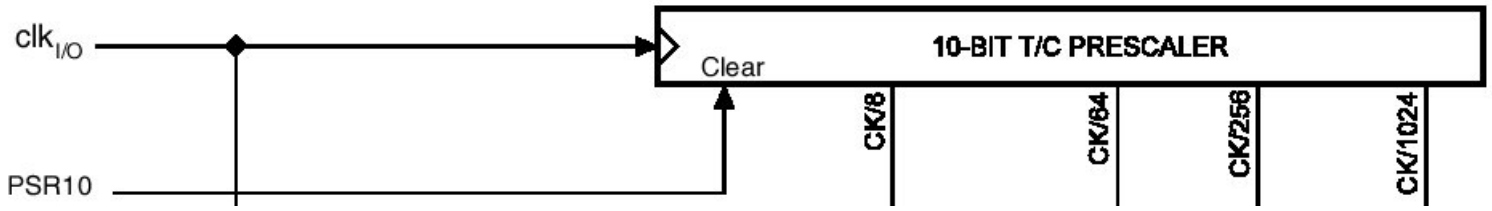
The mega8 incorporates three counter/timer devices. These can:

- Be used to count the number of events that have occurred (either external or internal)
- Act as a clock
- Trigger an interrupt after a specified number of events

Timer 0

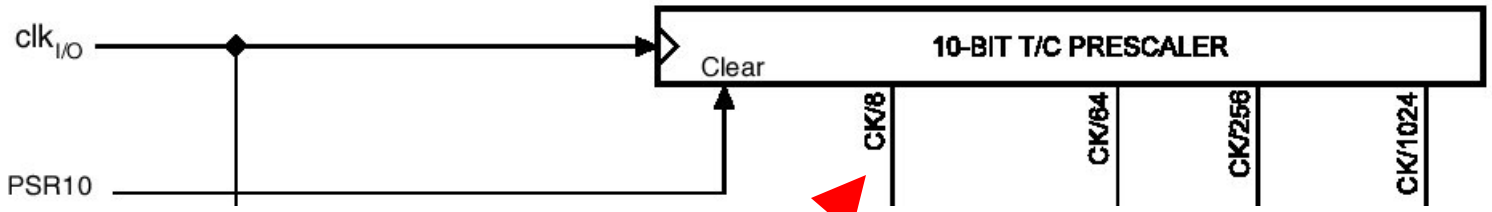
- Input source:
  - Pin T0 (PD4)
  - System clock
    - Potentially divided by a “prescaler”
- 8-bit counter
- When the counter turns over from 0xFF to 0x0, an interrupt can be generated

Timer 0 Implementation



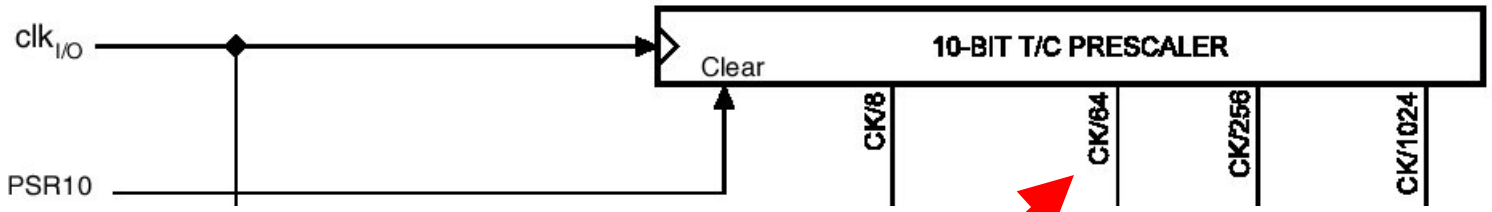
- Clock input to 10-bit counter
- Output bits: 3, 6, 8, and 10

Timer 0 Implementation



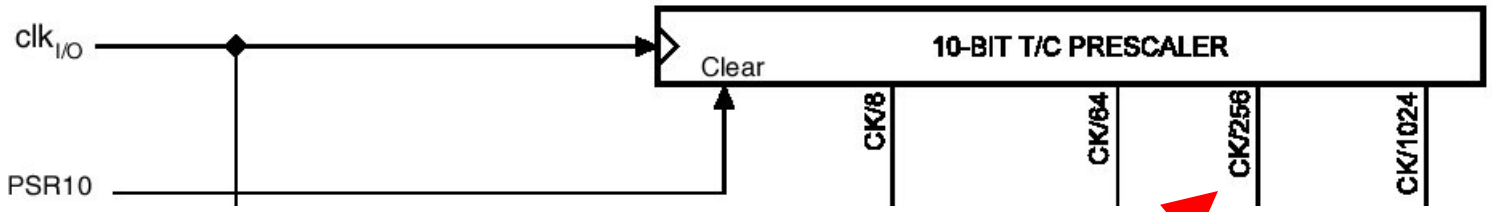
- Clock input to 10-bit counter
- Output bits: 3, 6, 8, and 10

Timer 0 Implementation



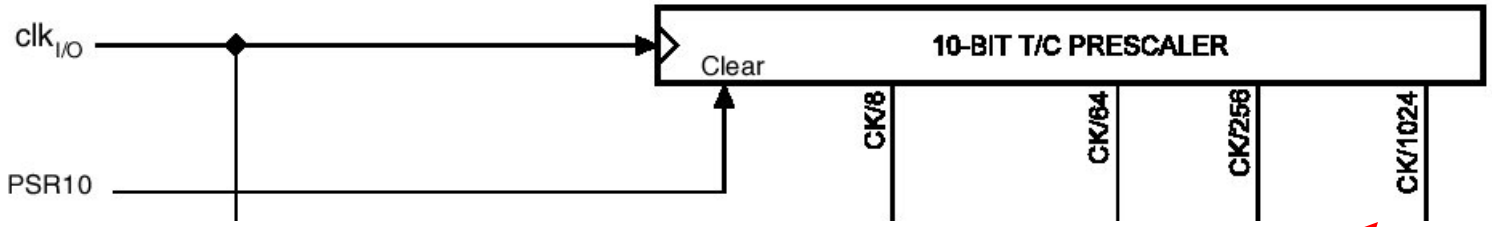
- Clock input to 10-bit counter
- Output bits: 3, 6, 8, and 10

Timer 0 Implementation



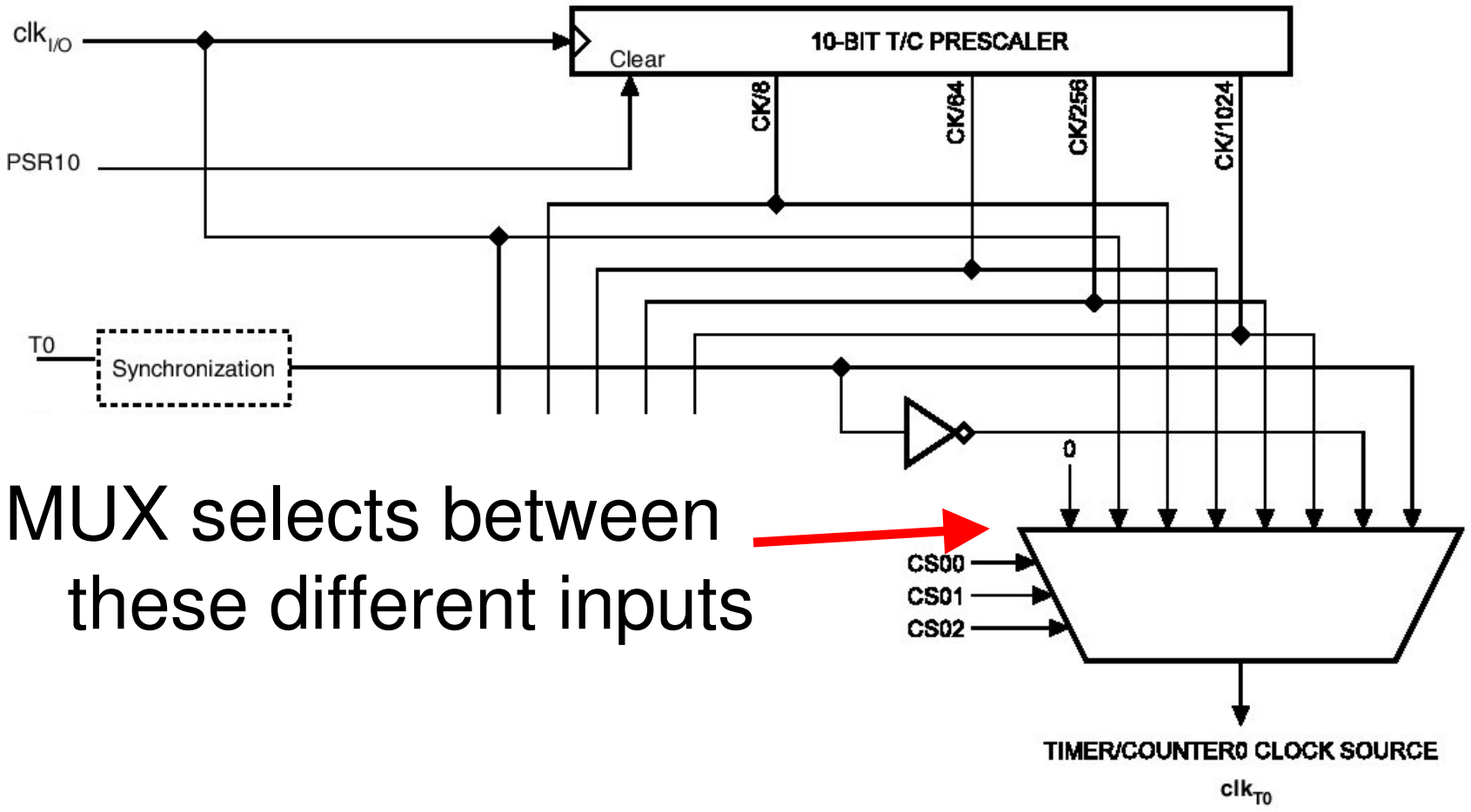
- Clock input to 10-bit counter
- Output bits: 3, 6, 8, and 10

Timer 0 Implementation



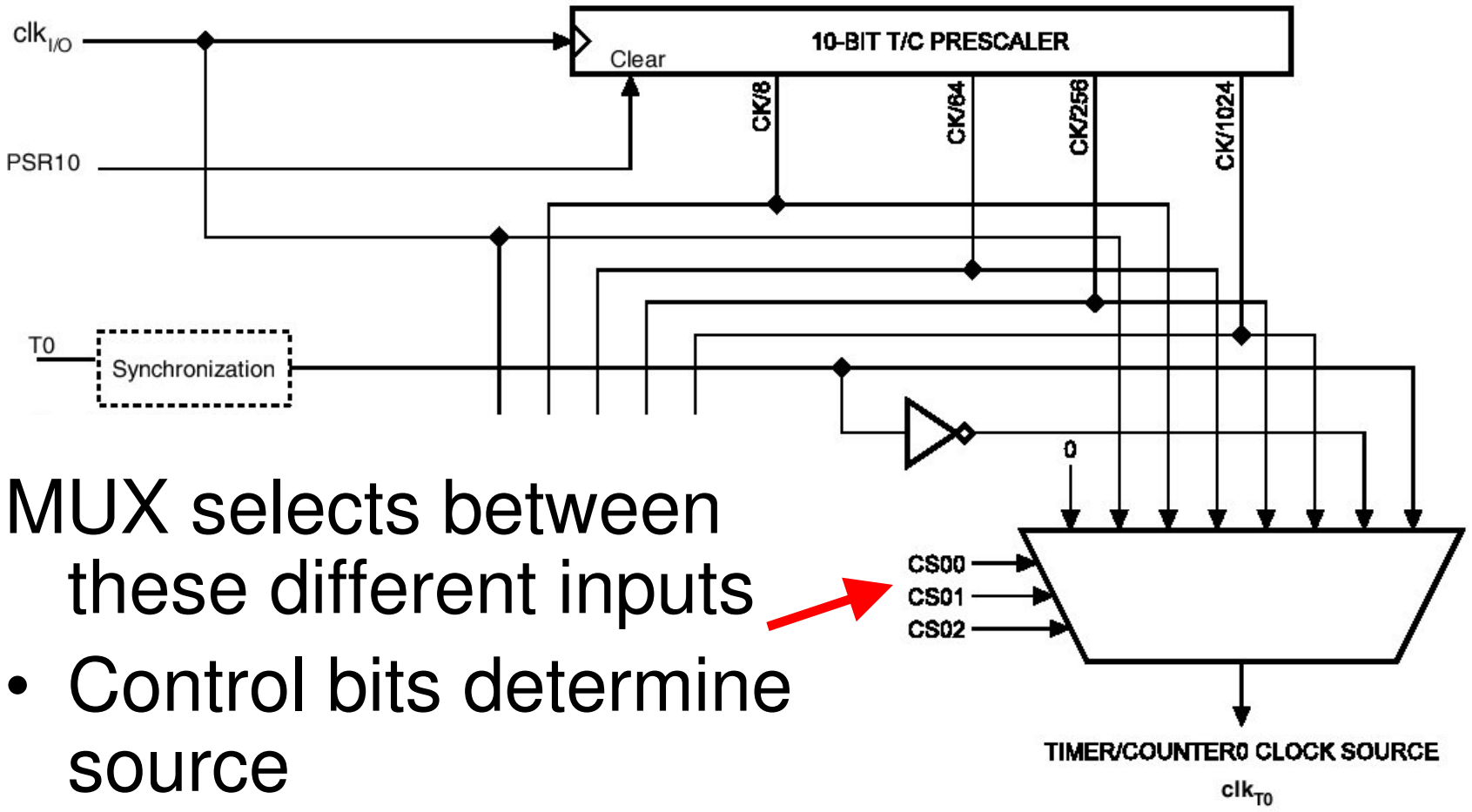
- Clock input to 10-bit counter
- Output bits: 3, 6, 8, and 10
  - These serve to divide the clock by the specified number of counts

Timer 0 Implementation



MUX selects between these different inputs

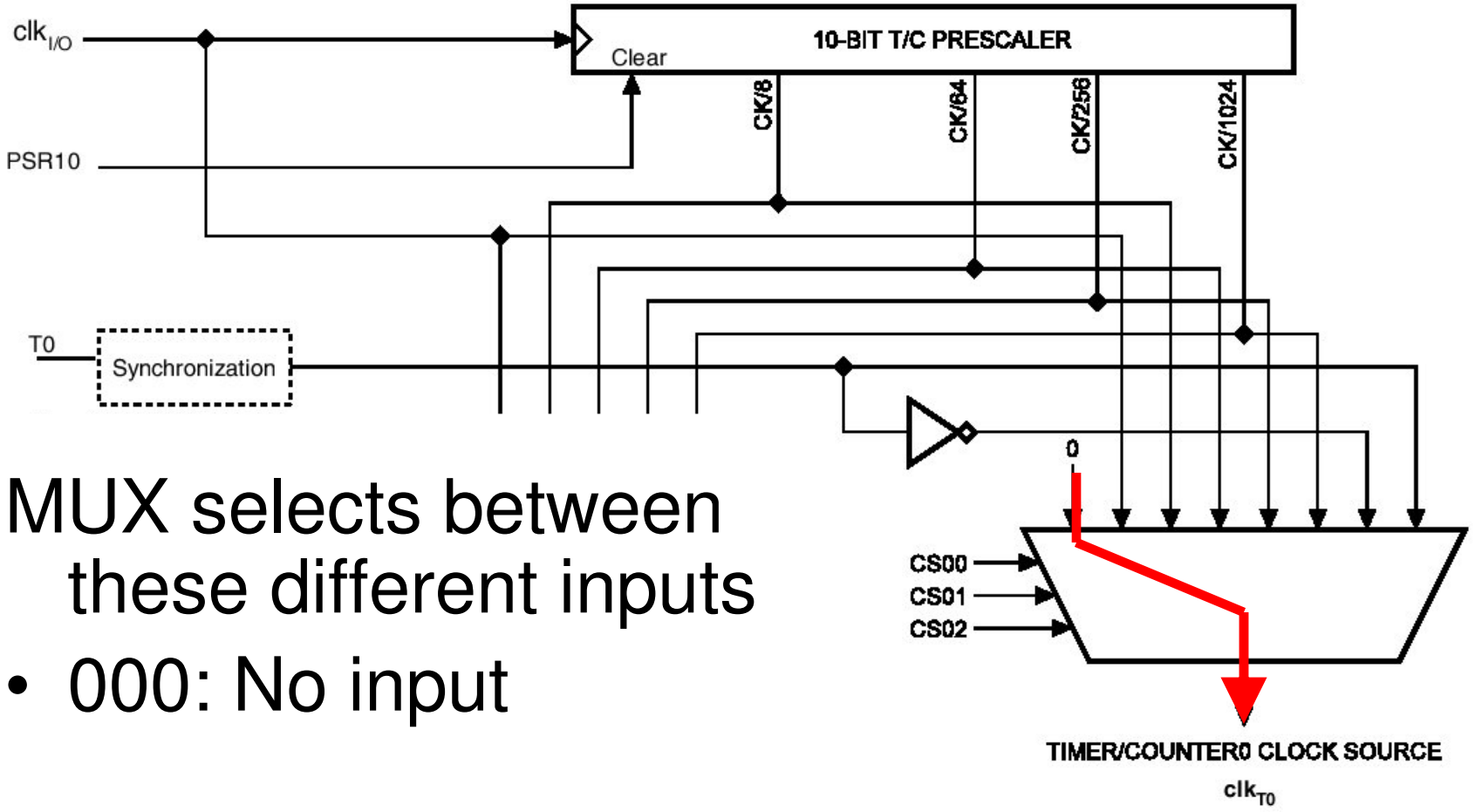
Timer 0 Implementation



MUX selects between these different inputs

- Control bits determine source

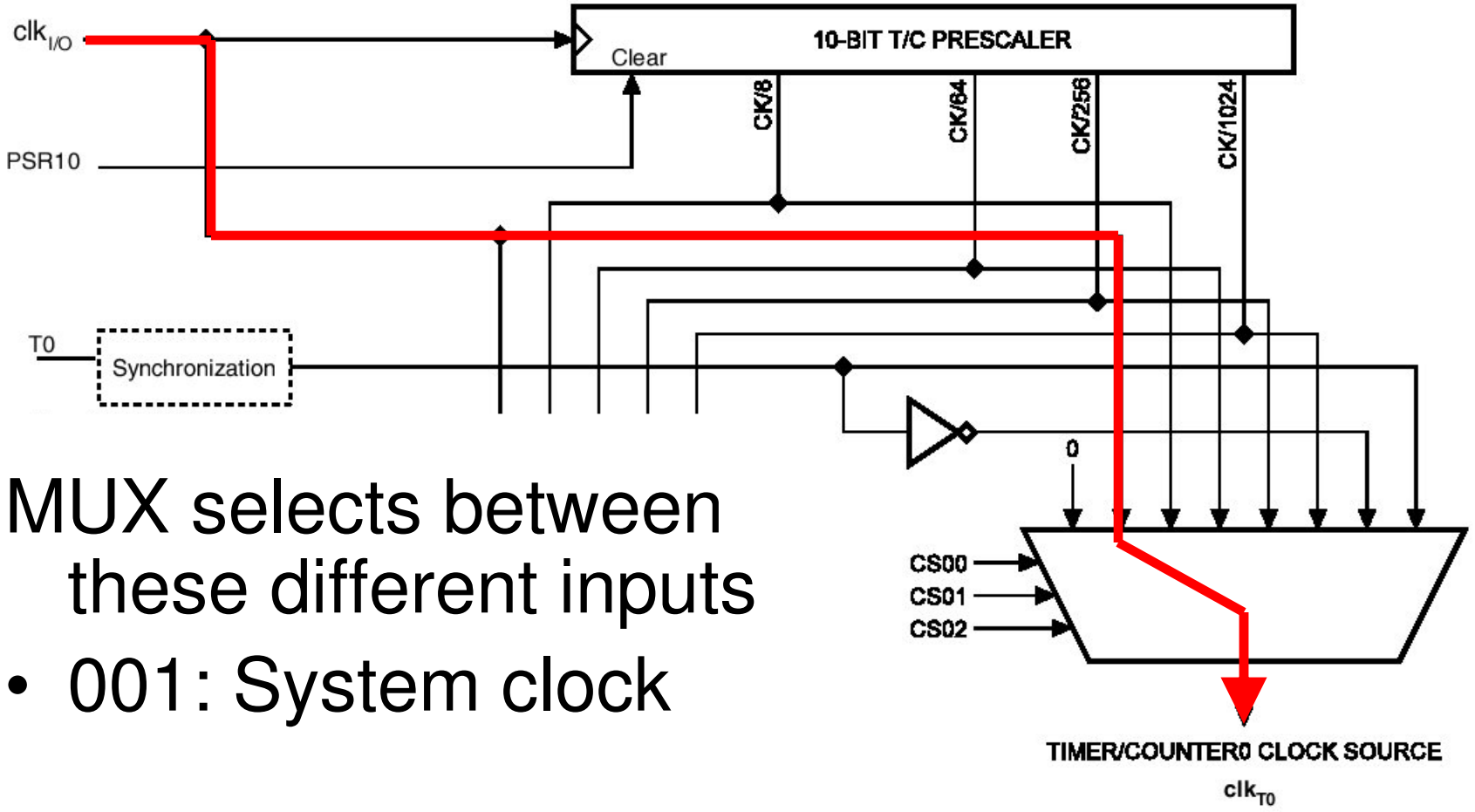
Timer 0 Implementation



MUX selects between these different inputs

- 000: No input

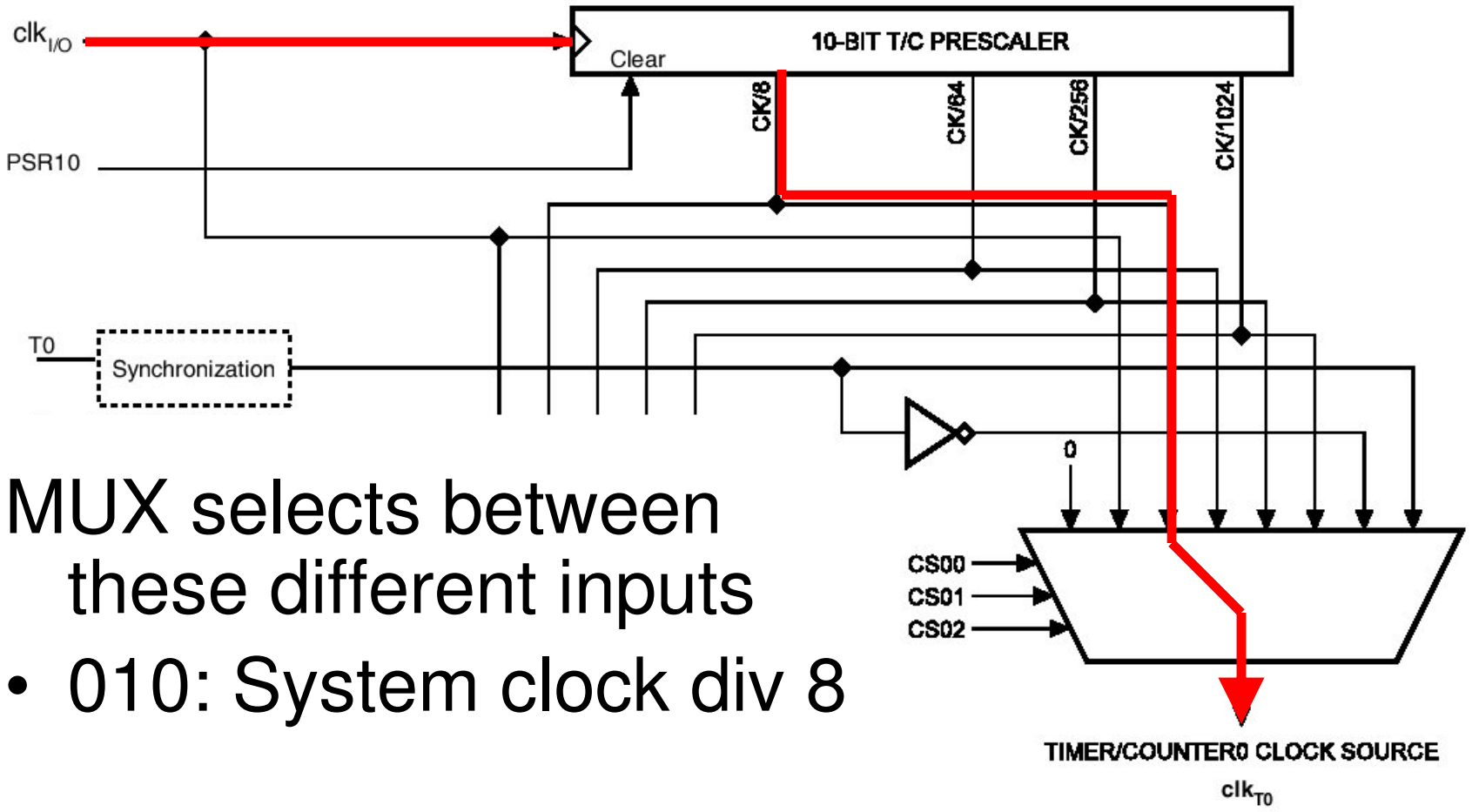
Timer 0 Implementation



MUX selects between these different inputs

- 001: System clock

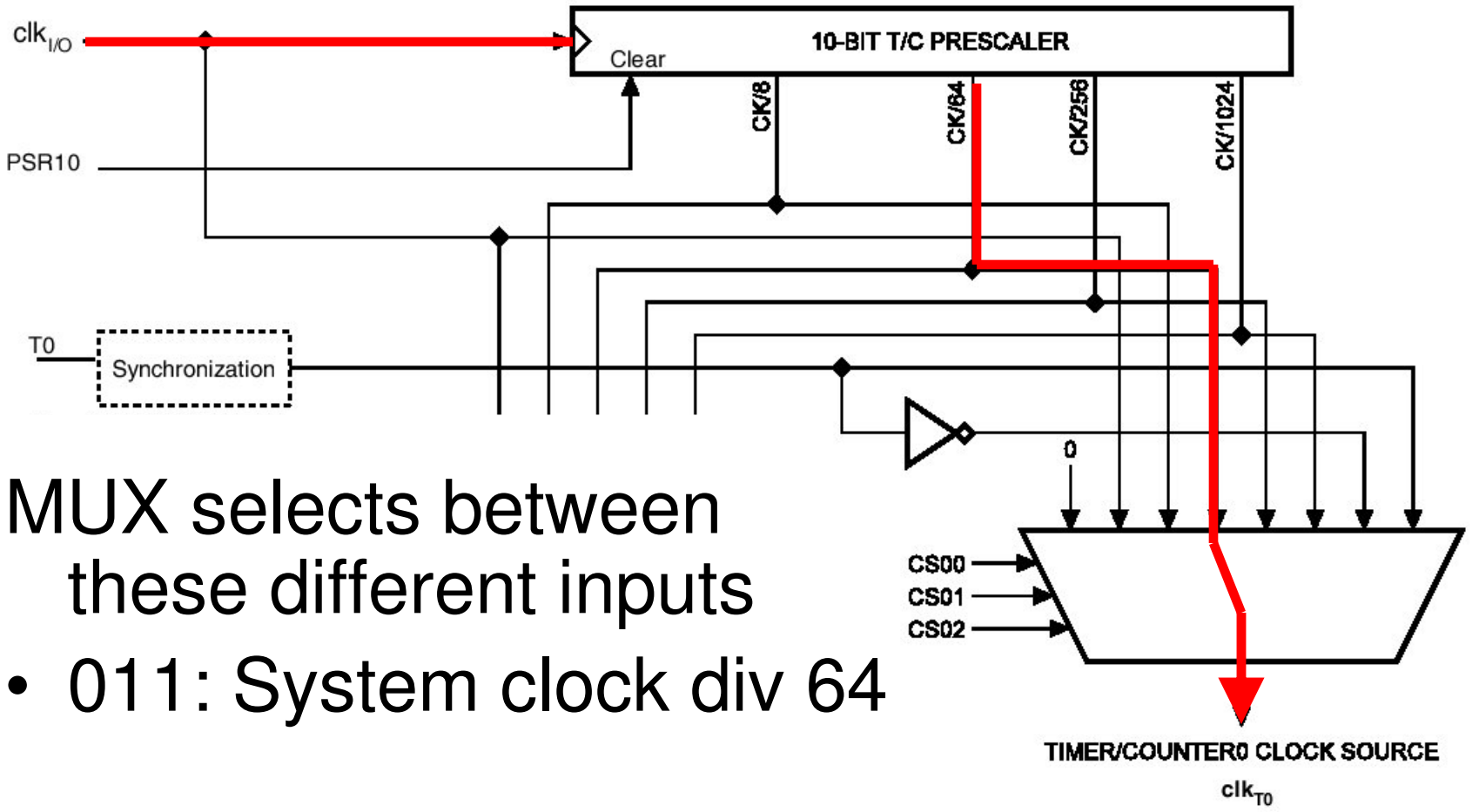
Timer 0 Implementation



MUX selects between these different inputs

- 010: System clock div 8

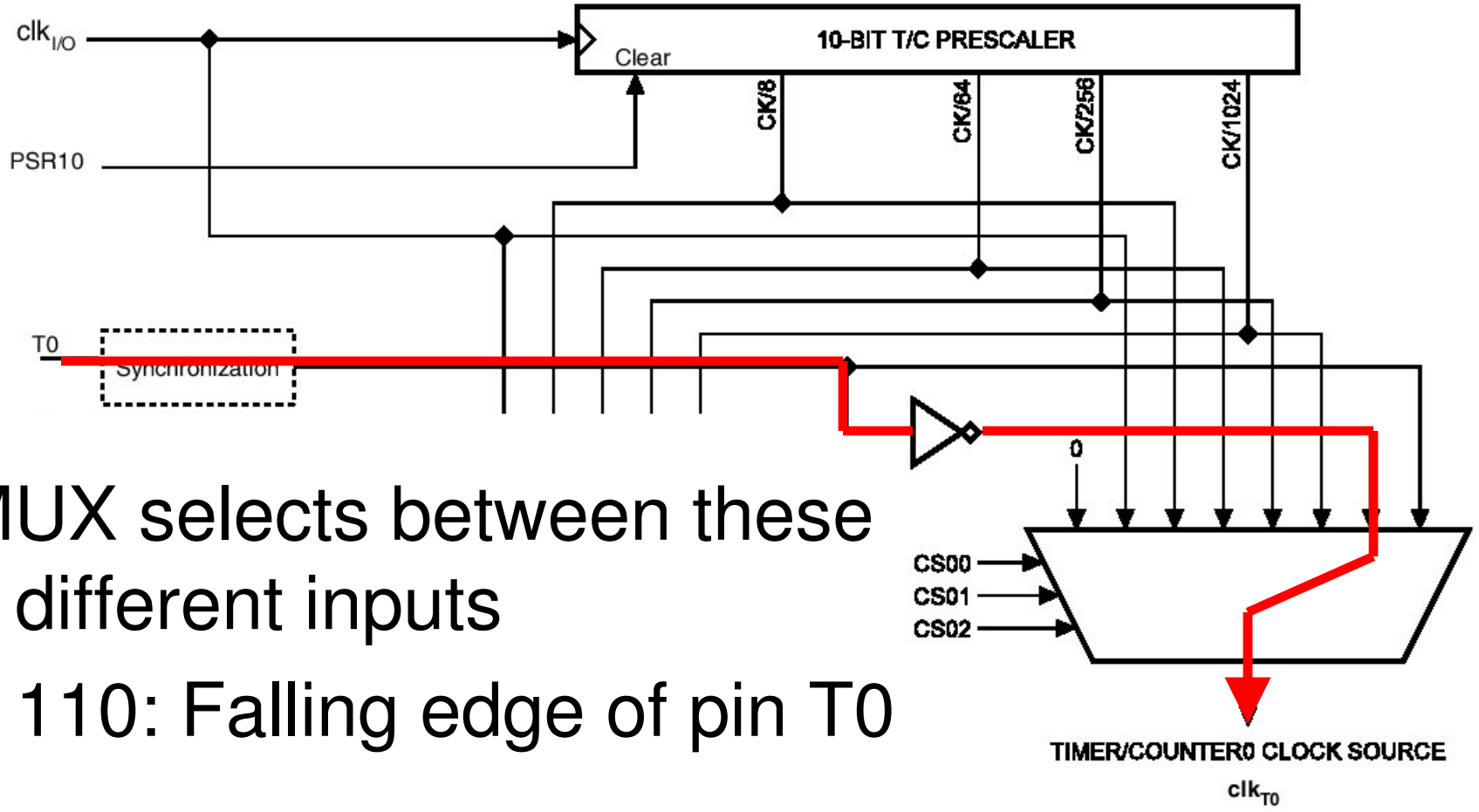
Timer 0 Implementation



MUX selects between these different inputs

- 011: System clock div 64

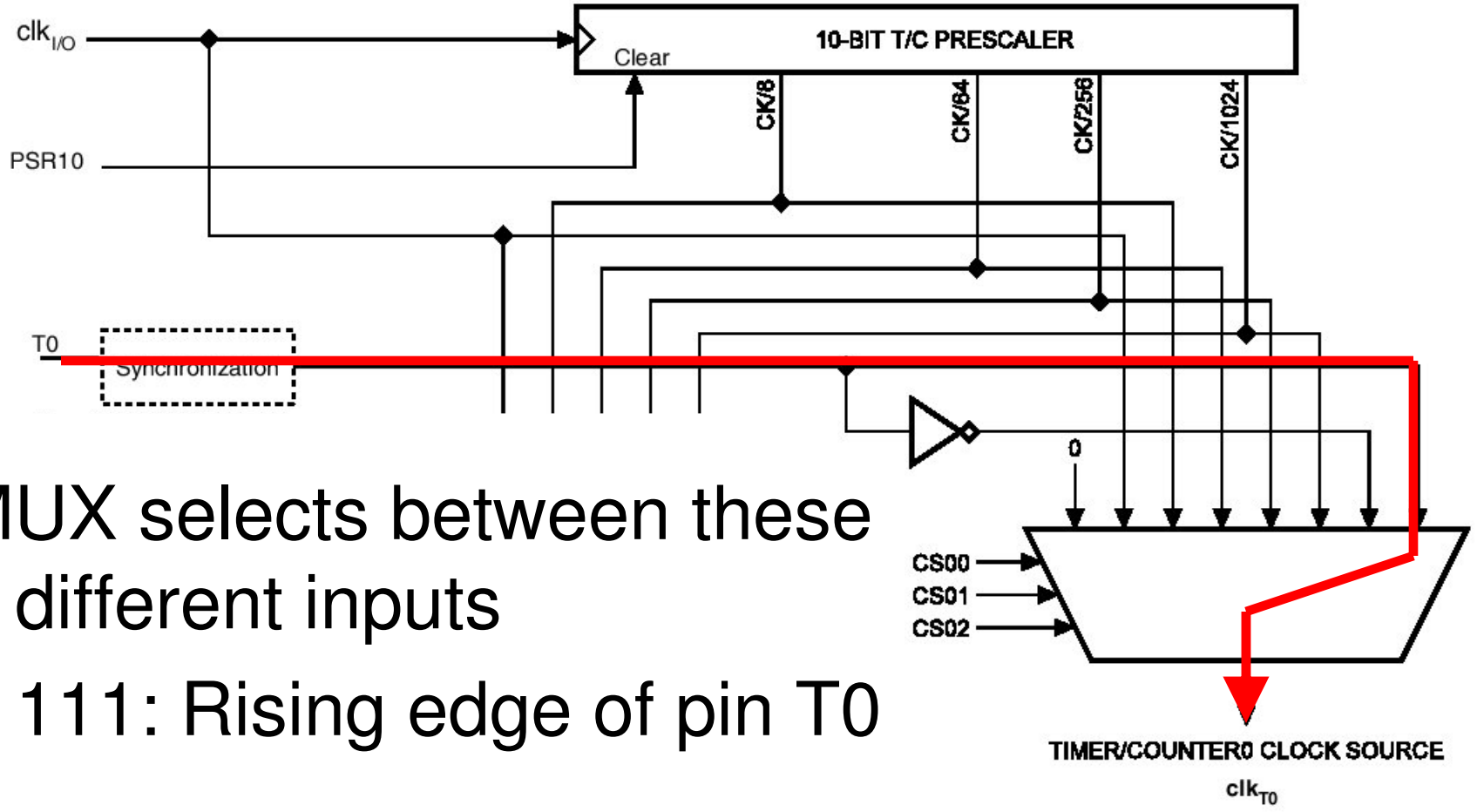
Timer 0 Implementation



MUX selects between these different inputs

- 110: Falling edge of pin T0

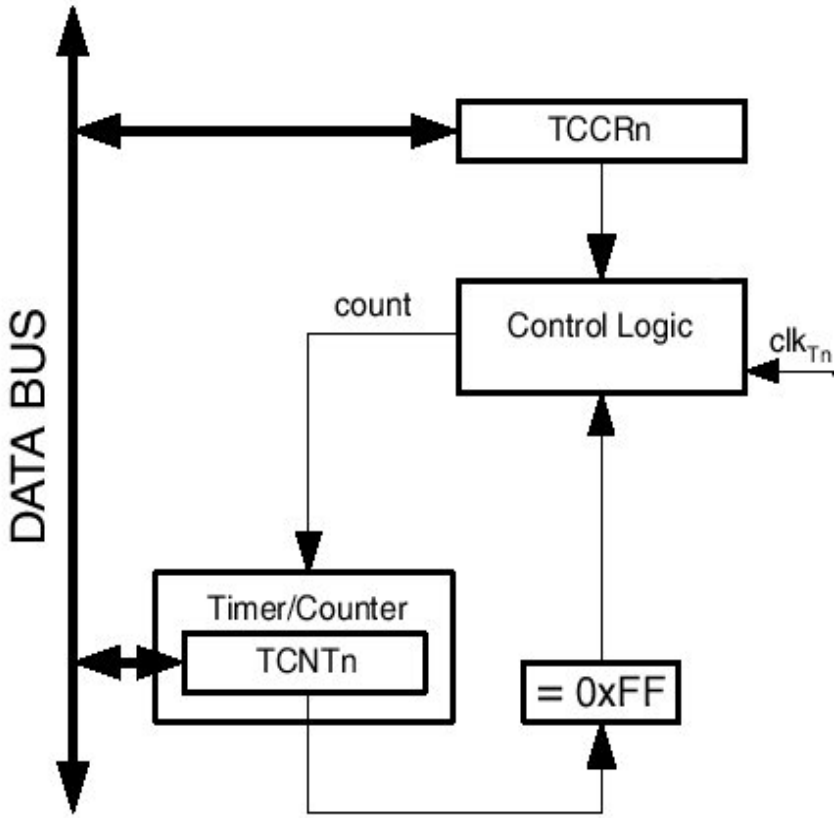
Timer 0 Implementation



MUX selects between these different inputs

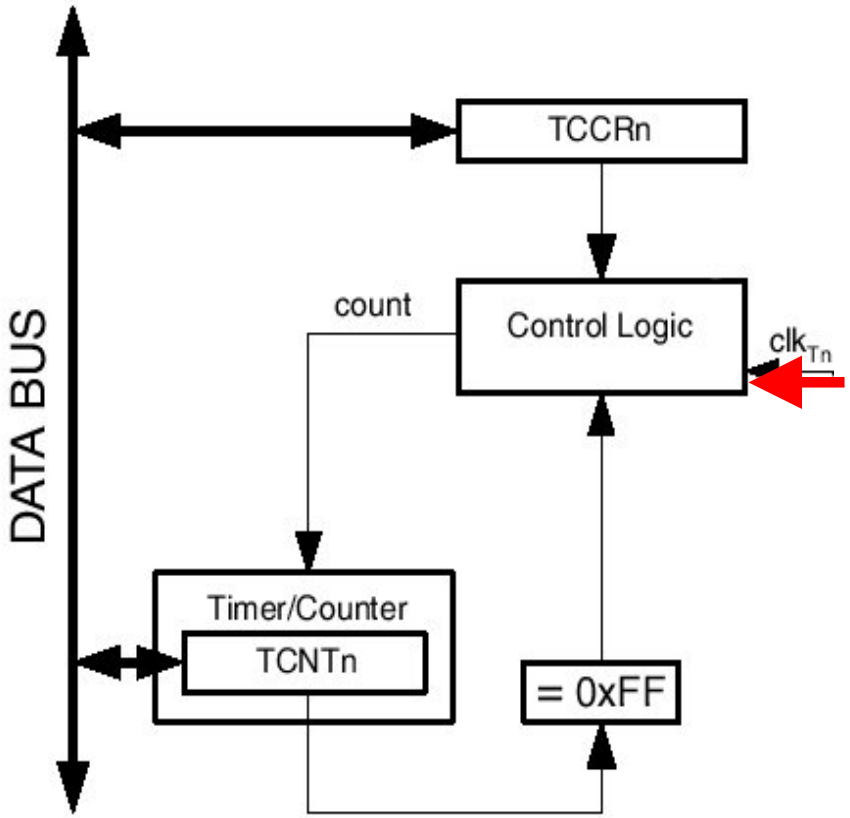
- 111: Rising edge of pin T0

Timer 0



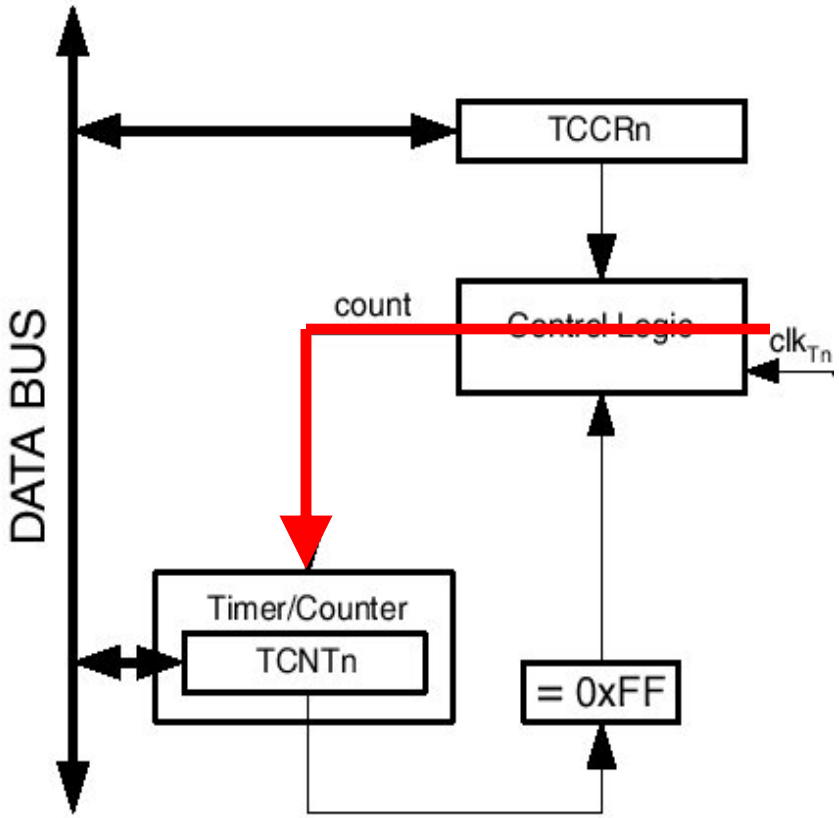
- TCNT0: 8-bit counter (a register)
- TCCR0: control register

Timer 0



- Clock source from previous slide

Timer 0



- Increment counter on every low-to-high transition

Timer 0 Example

Suppose:

- 16MHz clock
- Prescaler of 1024
- We wait for the timer to count from 0 to 156

How long does this take?

Timer 0 Example

$$delay = \frac{1024 * 156}{16{,}000{,}000} = 9948\ \mu s \approx 10\ ms$$

Timer 0 Code Example

```
timer0_config(TIMER0_PRE_1024);   // Prescale by 1024

timer0_set(0);       // Set the timer to 0

// Do something else for a while
while(timer0_read() < 156) {
};

// Break out at ~10 ms
```

See Atmel FAQ for example code

Timer 0 Example

Advantage over delay_ms():

- Can do other things while waiting
- Timing is much more precise
  - We no longer rely on a specific number of instructions to be executed

Timer 0 Example

Disadvantage:

- “something else” cannot take very much time

What is the solution?

Timer 0 Interrupt

What is the solution?

- Use interrupts!
- We can configure the timer to generate an interrupt every time the timer's counter rolls over from 0xFF to 0x00

Timer 0 Example II

Suppose:

- 16MHz clock
- Prescaler of 1024

How often is the interrupt generated?

Timer 0 Example II

$$interval = \frac{1024 * 256}{16{,}000{,}000} = 16.384\ ms$$

How many counts do we need so that we toggle the state of PB0 every second?

Timer 0 Example II

How many counts do we need so that we toggle the state of PB0 every second?

$$counts = \frac{1000\ ms}{16.384\ ms} = 61.0352$$

We will assume 61 is close enough.

Example II: Interrupt Routine

```
SIGNAL(SIG_OVERFLOW0) {
    ++counter;
    if(counter == 61) {
        // Toggle output state every 61st interrupt:
        //  This means: on for ~1 second and then off for ~1 sec
        PORTB ^= 1;
        counter = 0;
    };
};
```

See Atmel FAQ for example code

Example II: Initialization

```
// Initialize counter
counter = 0;

// Interrupt occurs every (1024*256)/16000000 = .016384 seconds
timer0_config(TIMER0_PRE_1024);

// Enable the timer interrupt
timer0_enable();

// Enable global interrupts
sei();

 while(1) {
   // Do something else
 };
```

Timer 0 with Interrupts

This solution is particularly nice:

- "something else" does not have to worry about timing at all
  - PB0 state is altered asynchronously

- Note that we can still have the shared data problem (but not in this example)

Other Timers

Timer 1:

- 16 bit counter

Timer 2:

- 8 bit counter

Next Topic: Information Encoding

We have talked about various forms of information encoding:

- Analog: use voltage to encode a value
- Parallel digital
- Serial digital

Next Topic: Information Encoding

An alternative: pulse-width modulation (PWM)

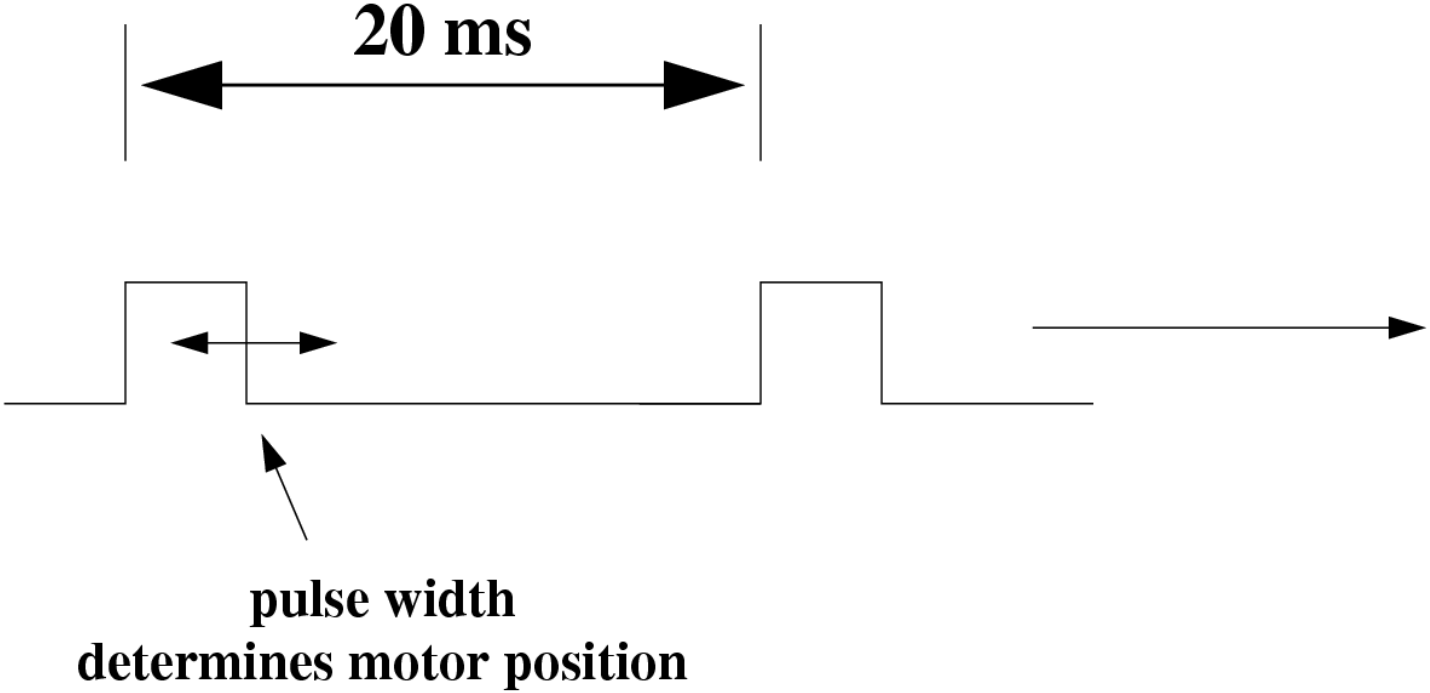
- Information is encoded in the time between the rising and falling edge of a pulse

PWM Example:

RC Servo Motors

- 3 pins: power (red), ground (black), and command signal (white)
- Signal pin expects a PWM signal

PWM Example



Internal circuit translates pulse width into a goal position:

- 0.5 ms: 0 degrees
- 1.5 ms: 180 degrees

RC Servo Motors

- Internal potentiometer measures the current orientation of the shaft
- Uses a **Position Servo Controller**: the difference between current and commanded shaft position determines shaft velocity.
- Mechanical stops limit the range of motion
  - These stops can be removed for unlimited rotation

PWM Example II: Controlling LED Brightness

What is the relationship of current flow through an LED and the rate of photon emission?

Controlling LED Brightness

What is the relationship of current flow through an LED and the rate of photon emission?

- They are linearly related (essentially)

Controlling LED Brightness

Suppose we pulse an LED for a given period of time with a digital signal: what is the relationship between pulse width and number of photons emitted?

Controlling LED Brightness

Suppose we pulse an LED for a given period of time with a digital signal: what is the relationship between pulse width and number of photons emitted?

- Again: they are linearly related (essentially)

- If the period is short enough, then the human eye will not be able to detect the flashes

Controlling LED Brightness

We need:

- To produce a periodic behavior, and
- A way to specify the pulse width (or the duty cycle)

How do we implement this in code?

Controlling LED Brightness

How do we implement this in code?

One way:

- Interrupt routine increments an 8-bit counter
- When the counter is 0, turn the LED on
- When the counter reaches some “duration”, turn the LED off

Last Time

- Interrupts
- Timers
- Generating regular interrupts
- PWM control

Today

- Interrupt subtleties
- DC motor control
- Direct Memory Access (DMA)

Administrivia

- Project 3 due on Tuesday
- New Atmel programmers are on-line.
  - See Atmel FAQ for details on how to use them
  - You will need a different adapter between the programmer and your circuit (but your circuit does not need to change)
- Schedule has been updated
  - See readings for coming weeks

Interrupt Challenge I: Shared Data and Compiler Optimizations

- Compilers (including ours) will often optimize code in order to minimize execution time
- These optimizations often pose no problems, but can be problematic in the face of interrupts and shared data

Shared Data and Compiler Optimizations

For example:

```
A = A + 1;
C = B * A
```

Will result in ‘A’ being fetched from memory once (into a general-purpose register) – even though ‘A’ is used twice

Shared Data and Compiler Optimizations

Now consider:

```
while(1) {
   PORTB = A;
}
```

What does the compiler do with this?

Shared Data and Compiler Optimizations

The compiler will assume that ‘A’ never changes.

This will result in code that looks something like this:

```
R1 = A;  // Fetch value of A into register 1
while(1) {
    PORTB = R1;
}
```

The compiler only fetches A from memory once!

Shared Data and Compiler Optimizations

This optimization is generally fine – but consider the following interrupt routine:

```
SIGNAL(SIG_OVERFLOW0){
  A = PIND;
}
```

- The global variable ‘A’ is being changed!
- The compiler has no way to anticipate this

Shared Data and Compiler Optimizations

The fix: the programmer must tell the compiler that it is not allowed to assume that a memory location is not changing

- This is accomplished when we declare the global variable:

```
volatile uint8_t A;
```

Back to Our Interrupt Implementation …

```
volatile uint8_t counter, duration;

SIGNAL(SIG_OVERFLOW0) {
  ++counter;
  if(counter == 0)
     PORTB |= 1;
  if(counter >= duration)
     PORTB &= ~1;
}
```

Initialization Details

- Set up timer
- Enable interrupts
- Set duration in some way
  - In this case, we will slowly increase it

What does this implementation look like?

Initialization

```
int main(void) {
   DDRB = 0xFF;
   PORTB = 0;

   // Initialize counter
   counter = 0;
   duration = 0;

   // Interrupt configuration
   timer0_config(TIMER0_NOPRE);  // No prescaler
   // Enable the timer interrupt
   timer0_enable();
   // Enable global interrupts
   sei();
                  :
```

PWM Implementation

What is the resolution (how long is one increment of “duration”)?

PWM Implementation

What is the resolution (how long is one increment of “duration”)?

- The timer0 counter (8 bits) expires every 256 clock cycles

$$t = \frac{256}{16000000} = 16\ \mu s$$

(assuming a 16MHz clock)

PWM Implementation

What is the period of the pulse?

PWM Implementation

What is the period of the pulse?

- The 8-bit counter (of the interrupt) expires every 256 interrupts

$$t = \frac{256 * 256}{16000000} = 4.096\ ms$$

Doing “Something Else”

```
        :
  unsigned int i;
  while(1) {
     for(i = 0; i < 256; ++i)
         duration = i;
         delay_ms(50);
     };
  };
}
```

Interrupts and Timers

Timing can often involve a cascade of multiple counters:

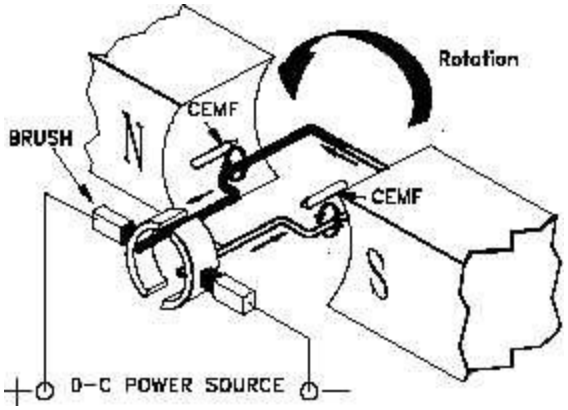
- Prescalar (1 … 1024)
- Timer0 (256)
- Counter within an interrupt routine (any)

Each counter implements a frequency division

DC Motors

- Current (ideally) is proportional to the torque produced by the motor
- Direction of current flow determines torque direction

How can a digital input control torque magnitude?



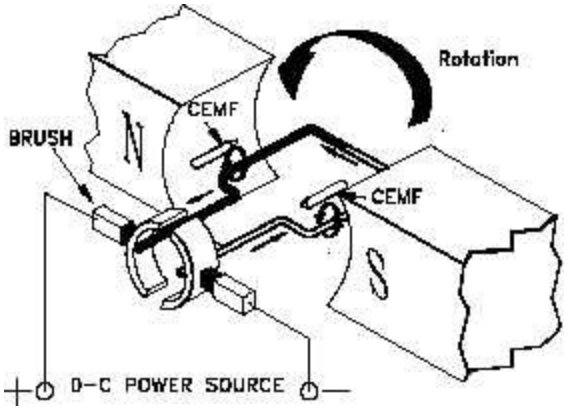
www.tpub.com

www.pcgadgets.com

LEDs to DC Motors

How can a digital input control torque magnitude?

- Use PWM!



www.tpub.com

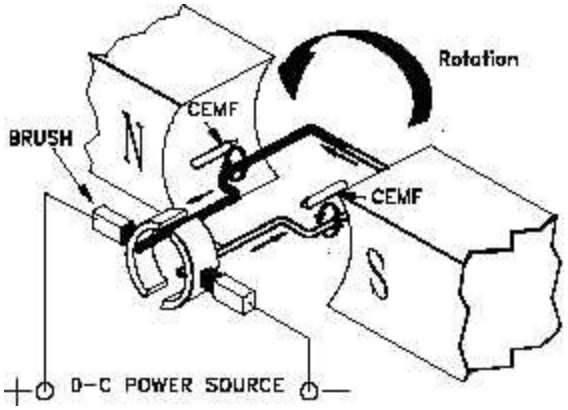
How do we handle torque direction?

www.pcgadgets.com

LEDs to DC Motors

How do we handle torque direction?

- +5V to north 0V to south
- 0V to north +5V to south



www.tpub.com

How would we implement this?

www.pcgadgets.com

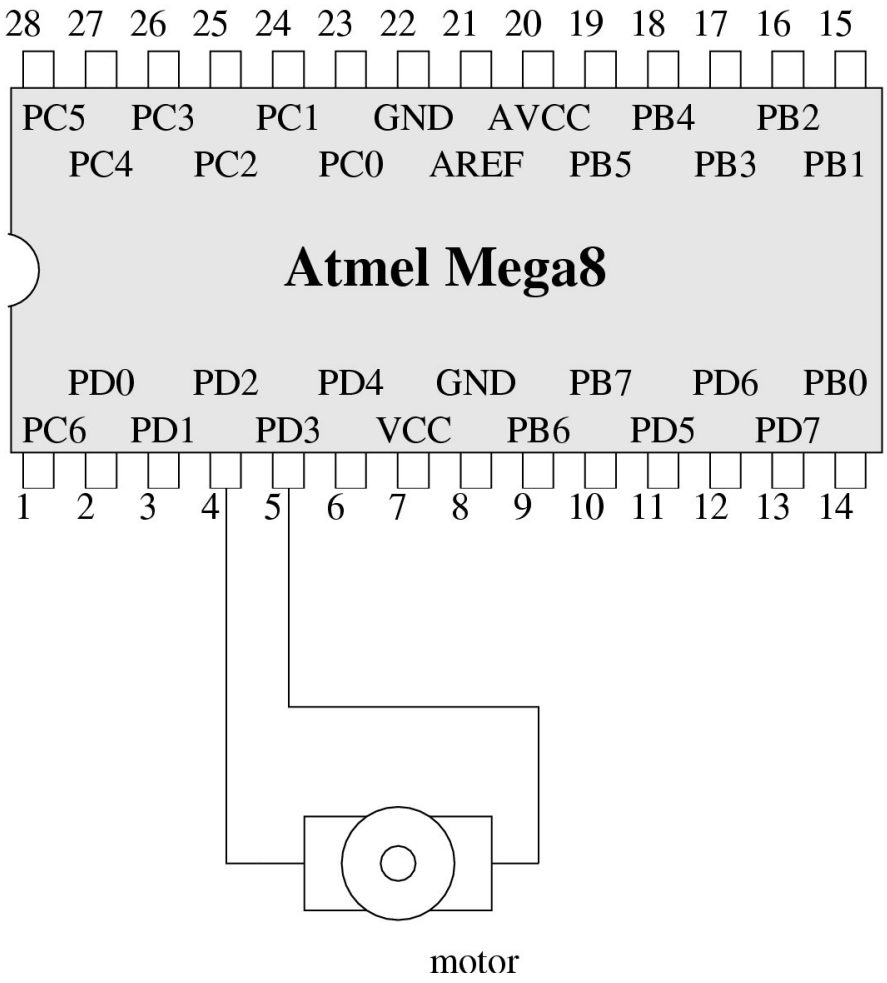
DC Motor Control

One possibility…

- Connect motor directly to the I/O pins

Two directions:

- PD2: 1; PD3: 0
- PD2: 0; PD3: 1

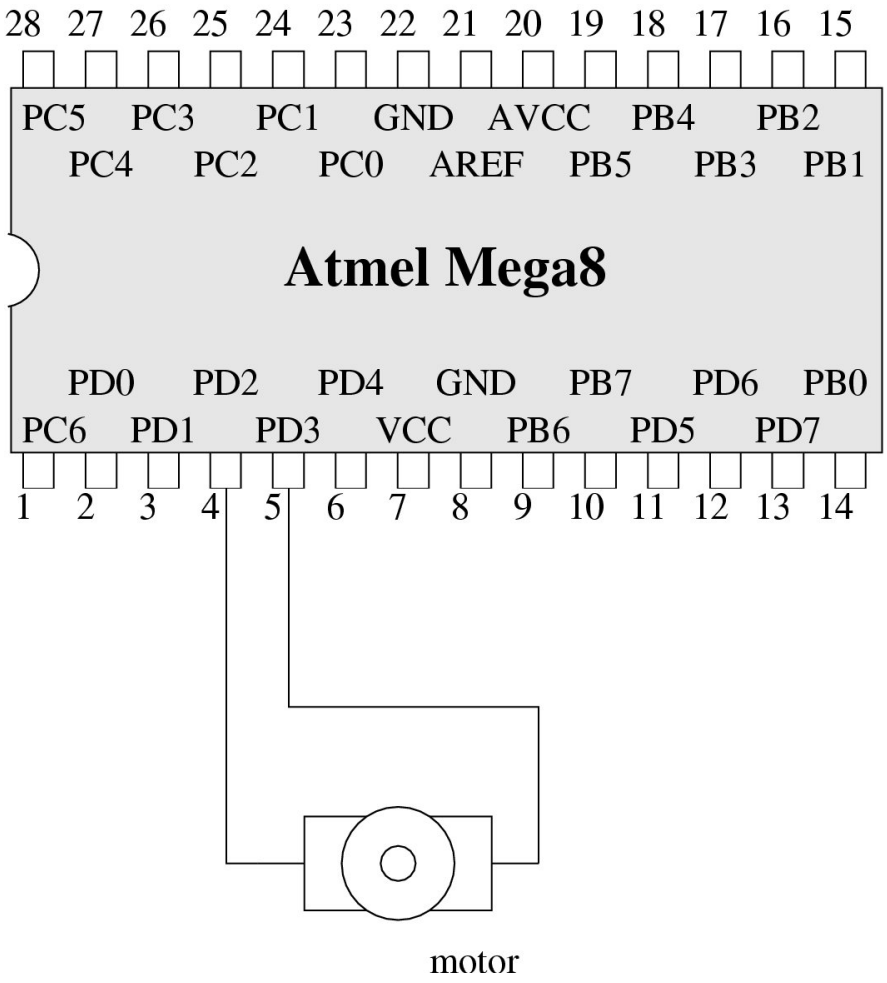


DC Motor Control

One possibility…

- Connect motor directly to the I/O pins

What is wrong with this implementation?

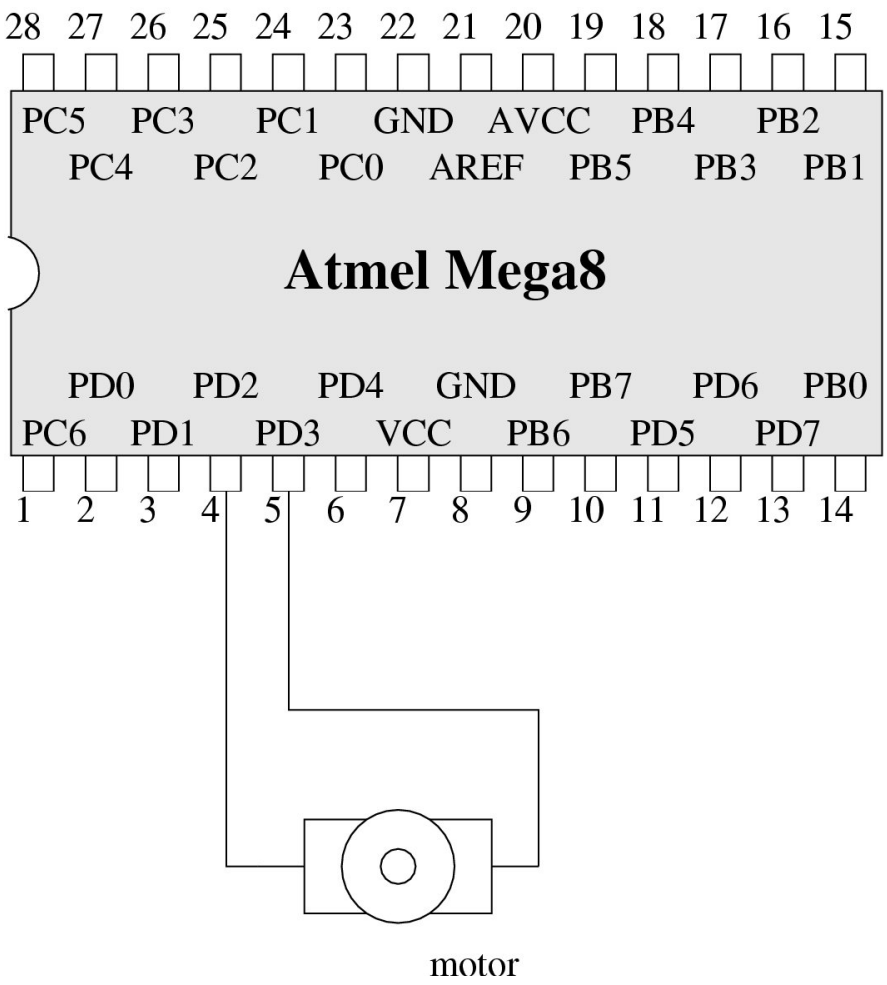


DC Motor Control

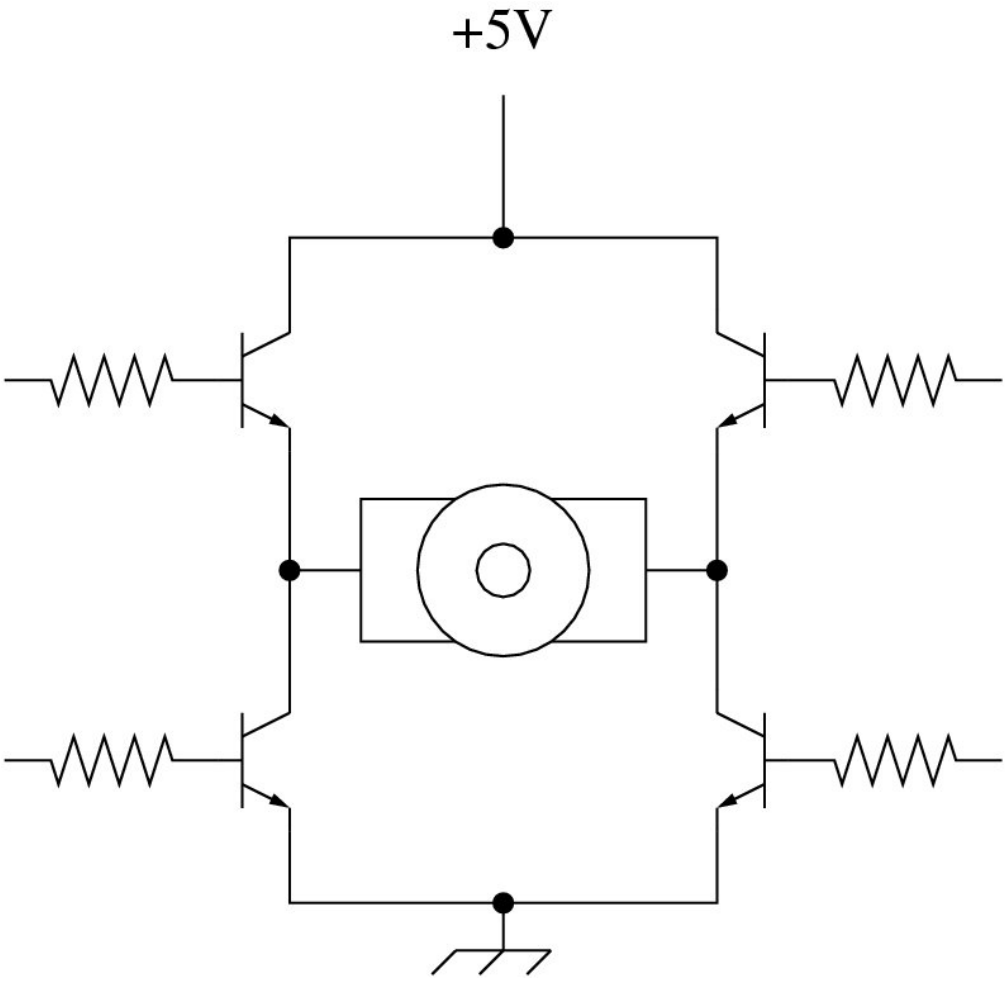
What is wrong with this implementation?

- Our I/O pins can source/sink at most 20 mA of current
- This is not very much when it comes to motors…

How do we fix this?

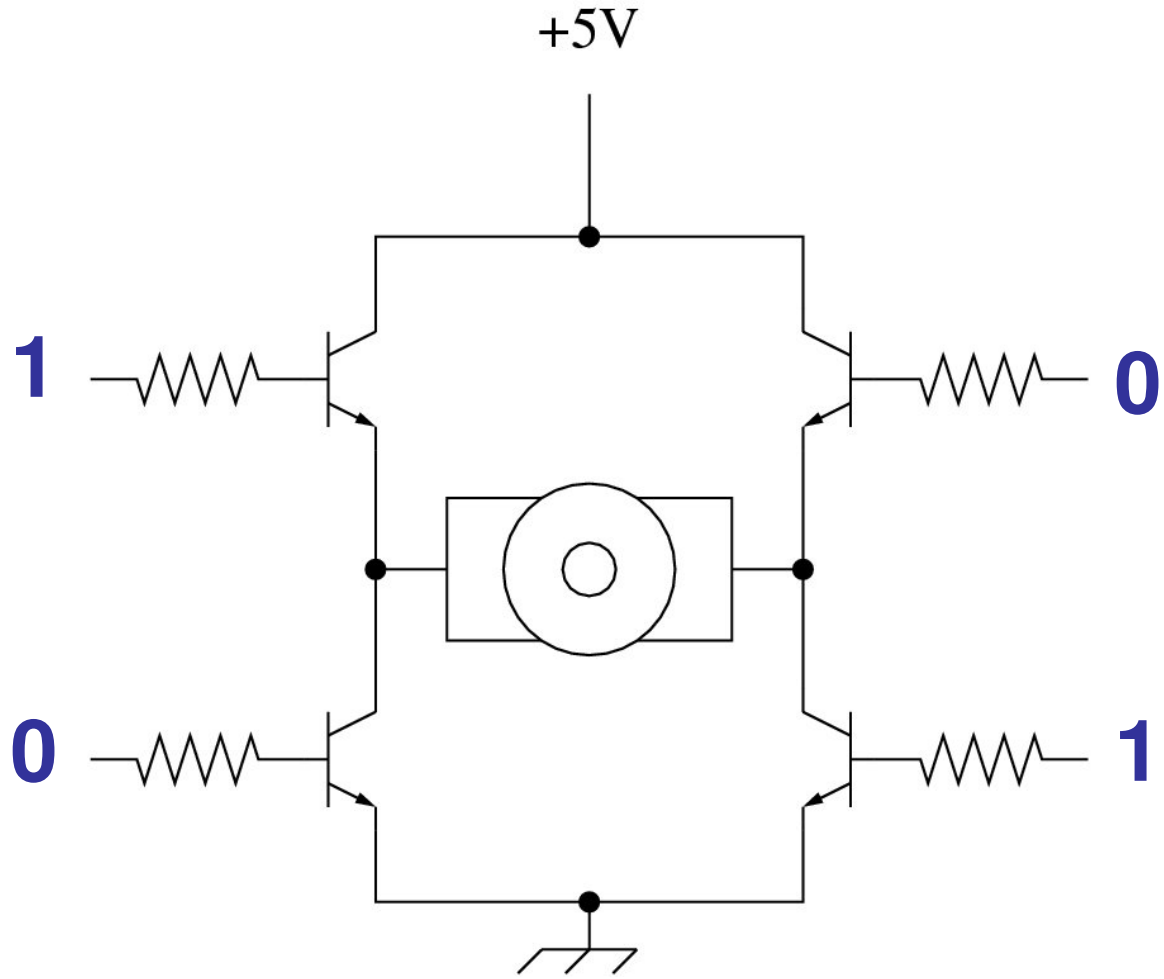


Simple H-Bridge



Simple H-Bridge

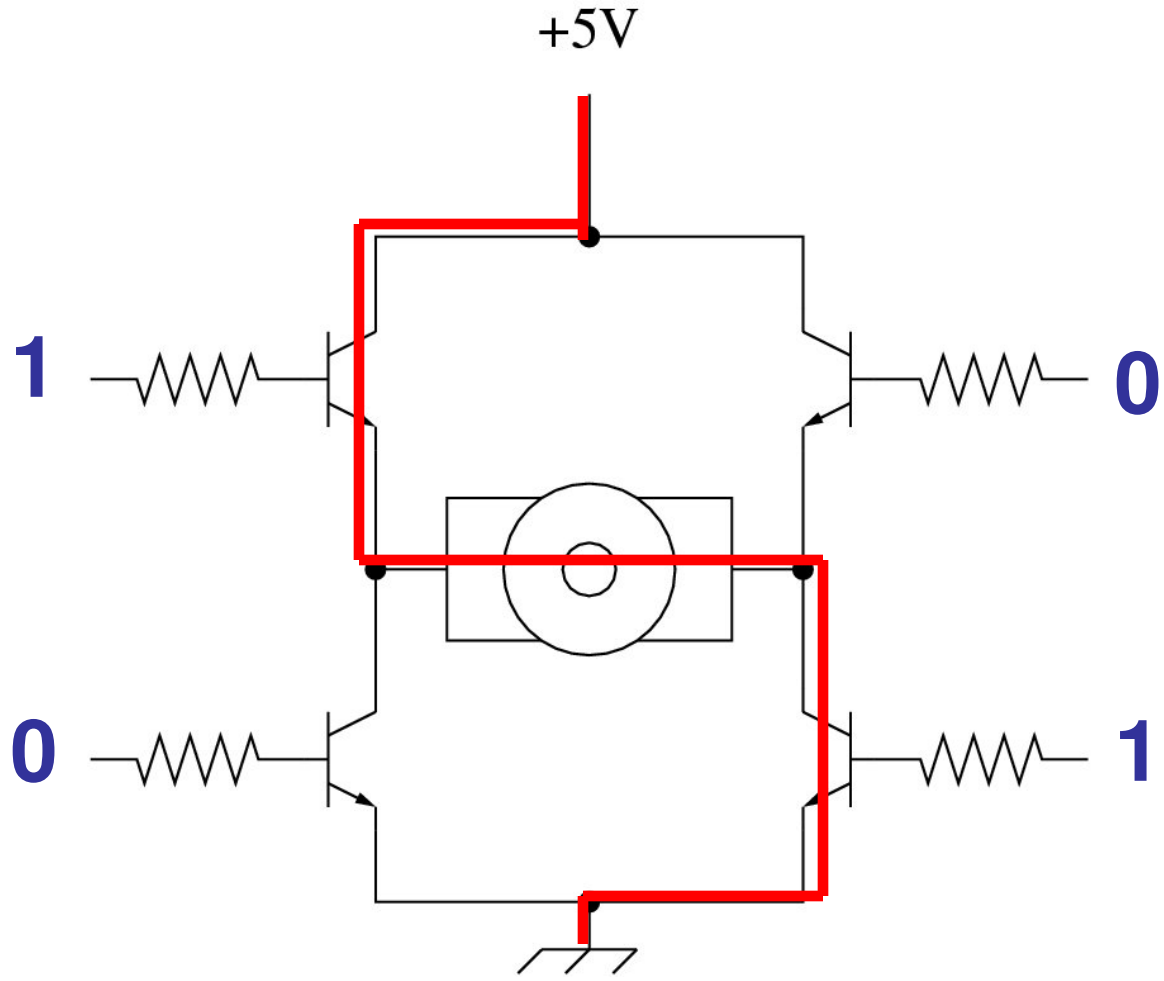
What happens with these inputs?



Simple H-Bridge

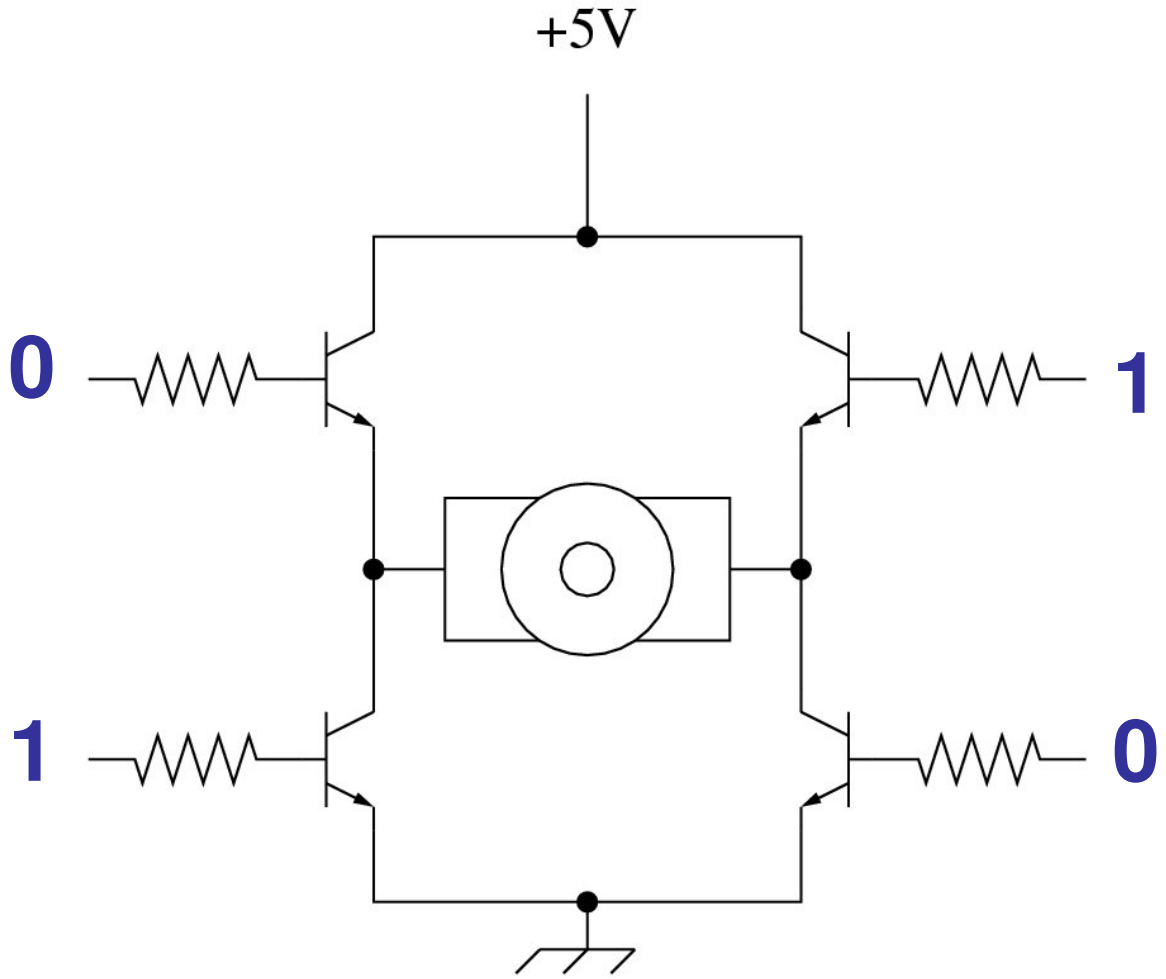
What happens with these inputs?

- Motor turns in one direction



Simple H-Bridge

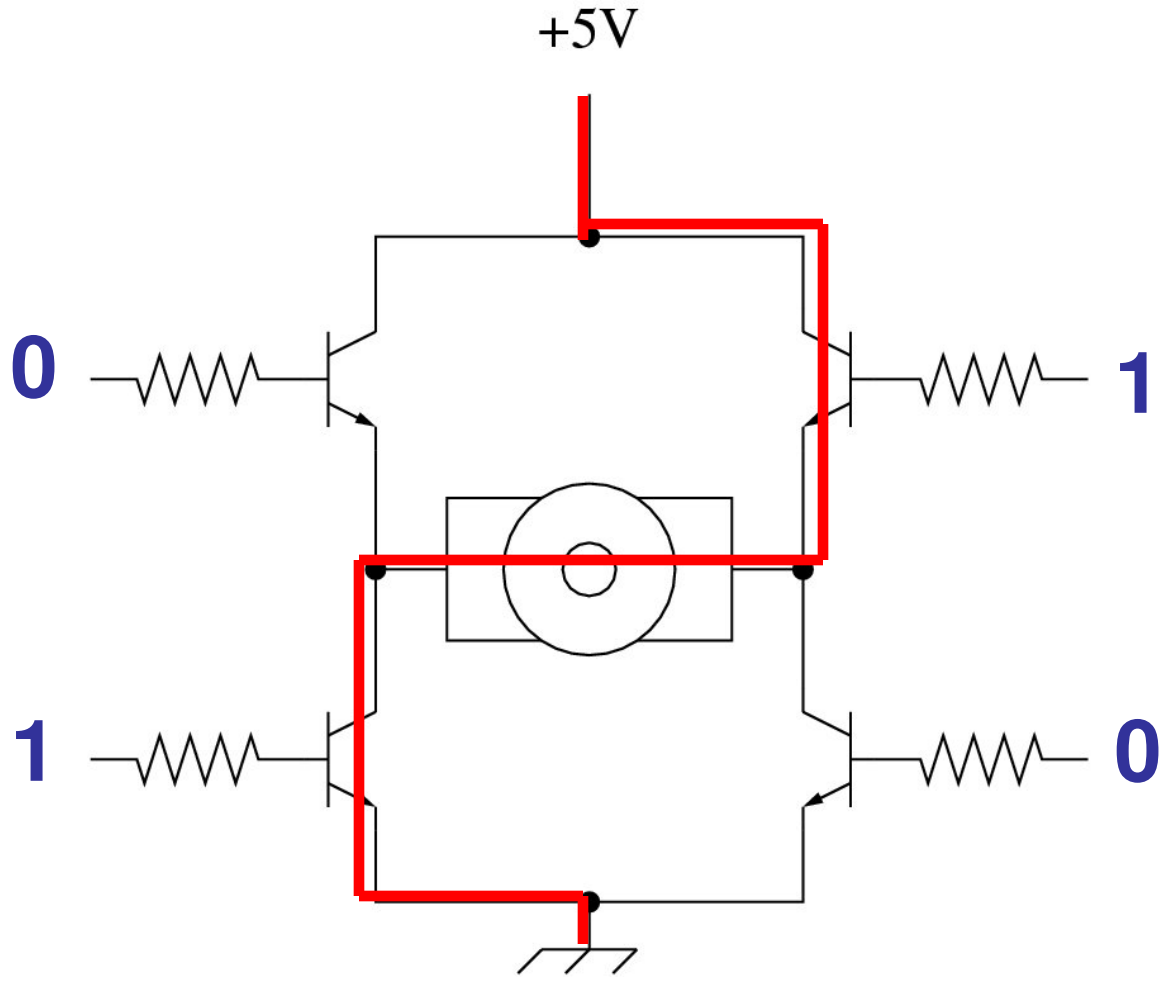
How about these inputs?



Simple H-Bridge

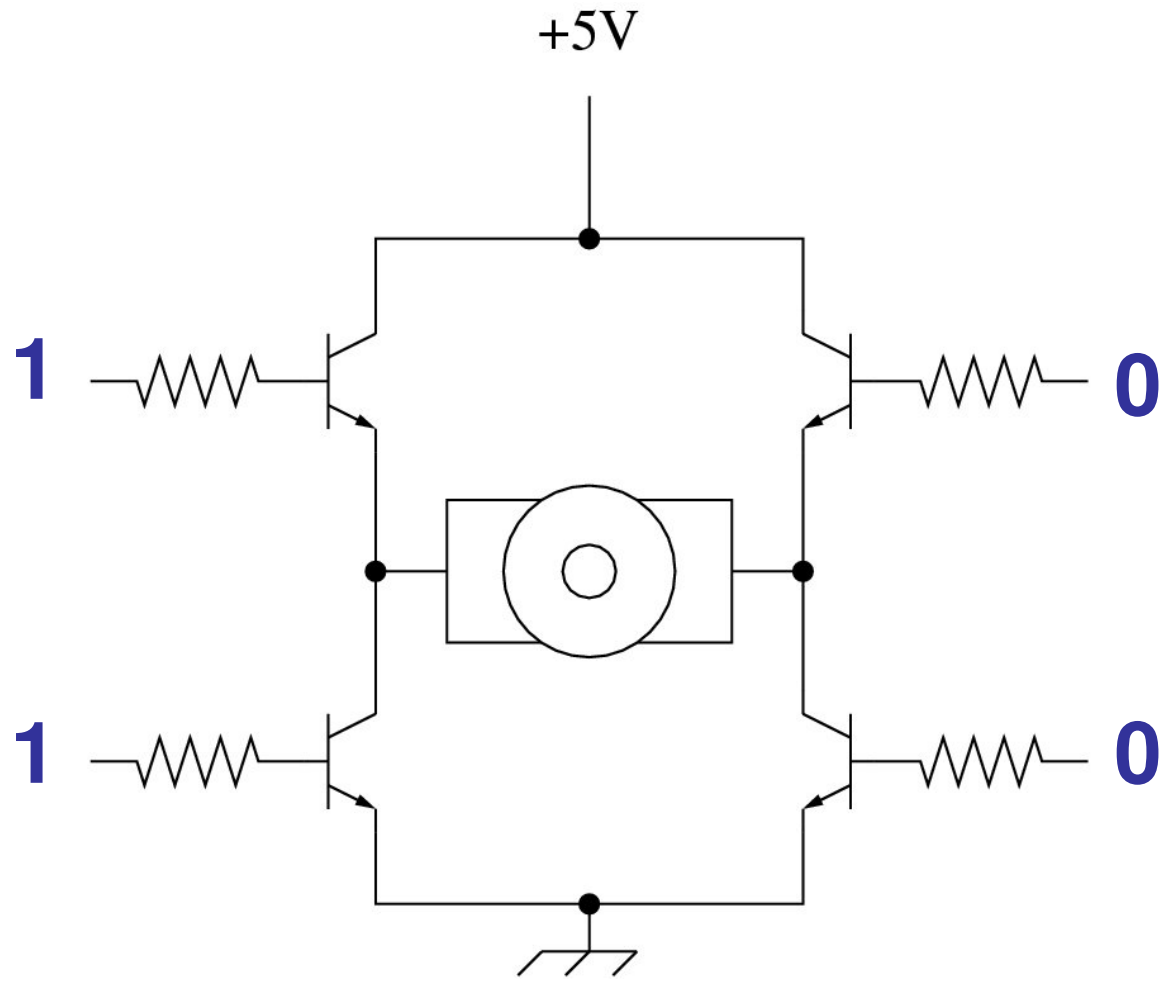
What happens with these inputs?

- Motor turns in the other direction!



Simple H-Bridge

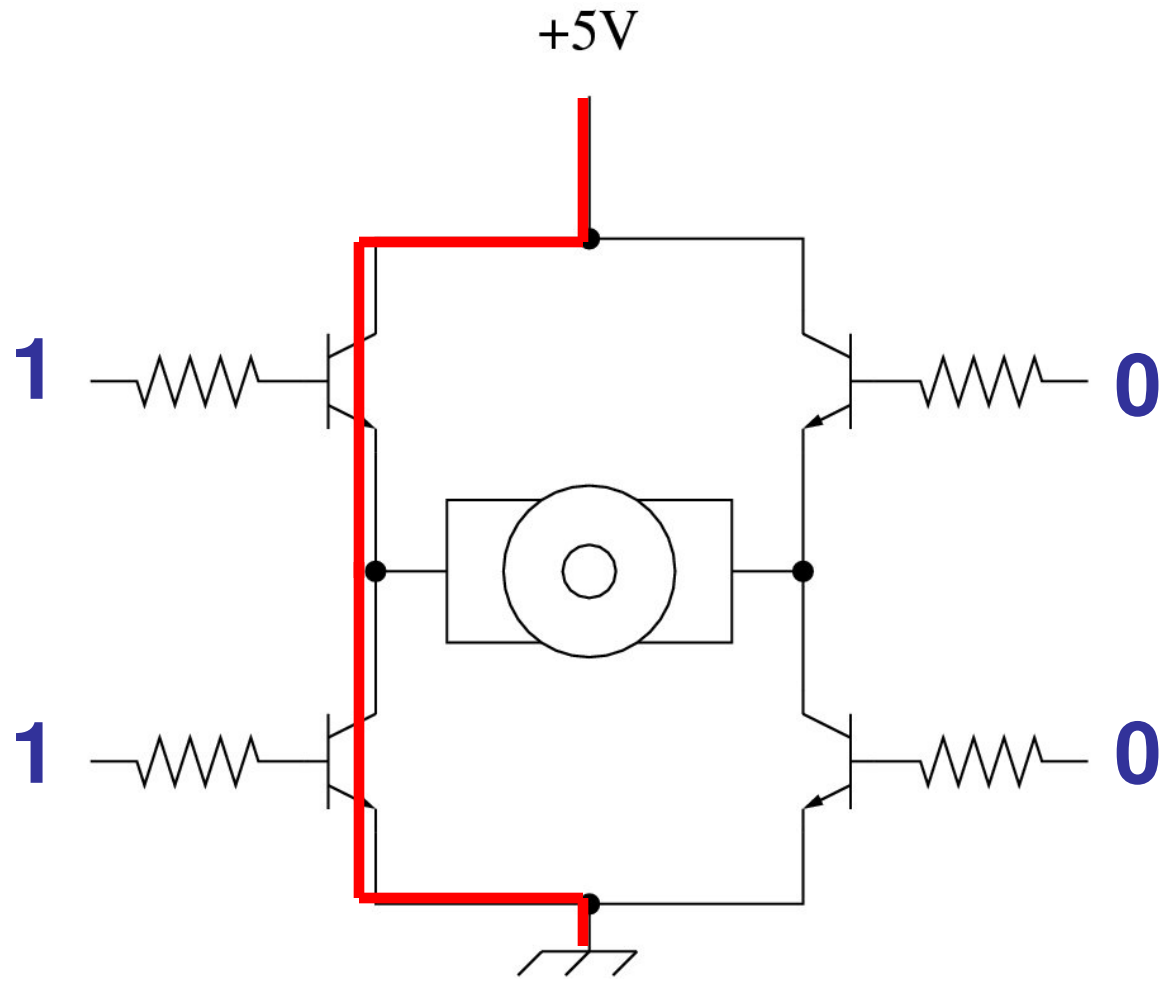
How about these inputs?



Simple H-Bridge

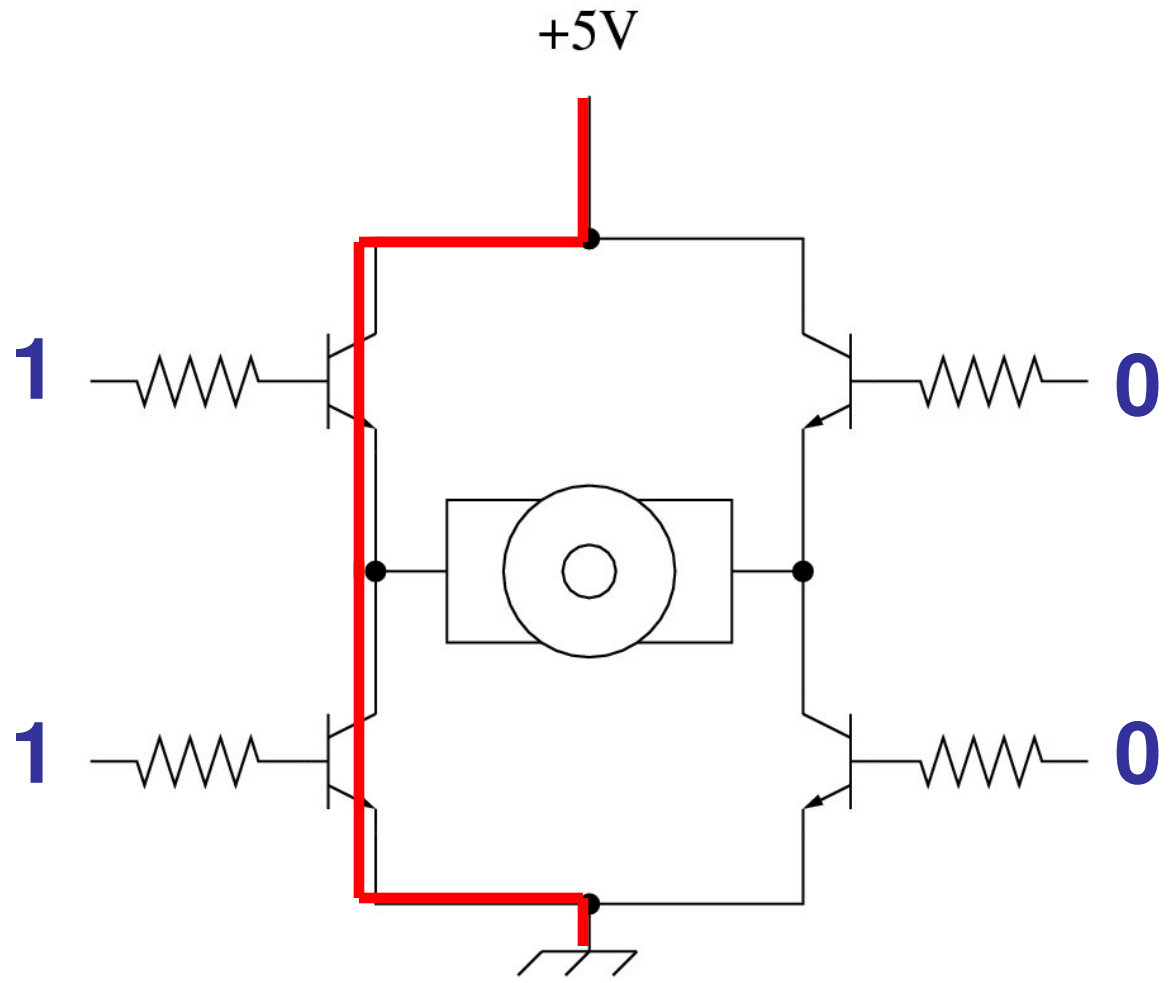
What happens with these inputs?

- We short power to ground
- … very bad



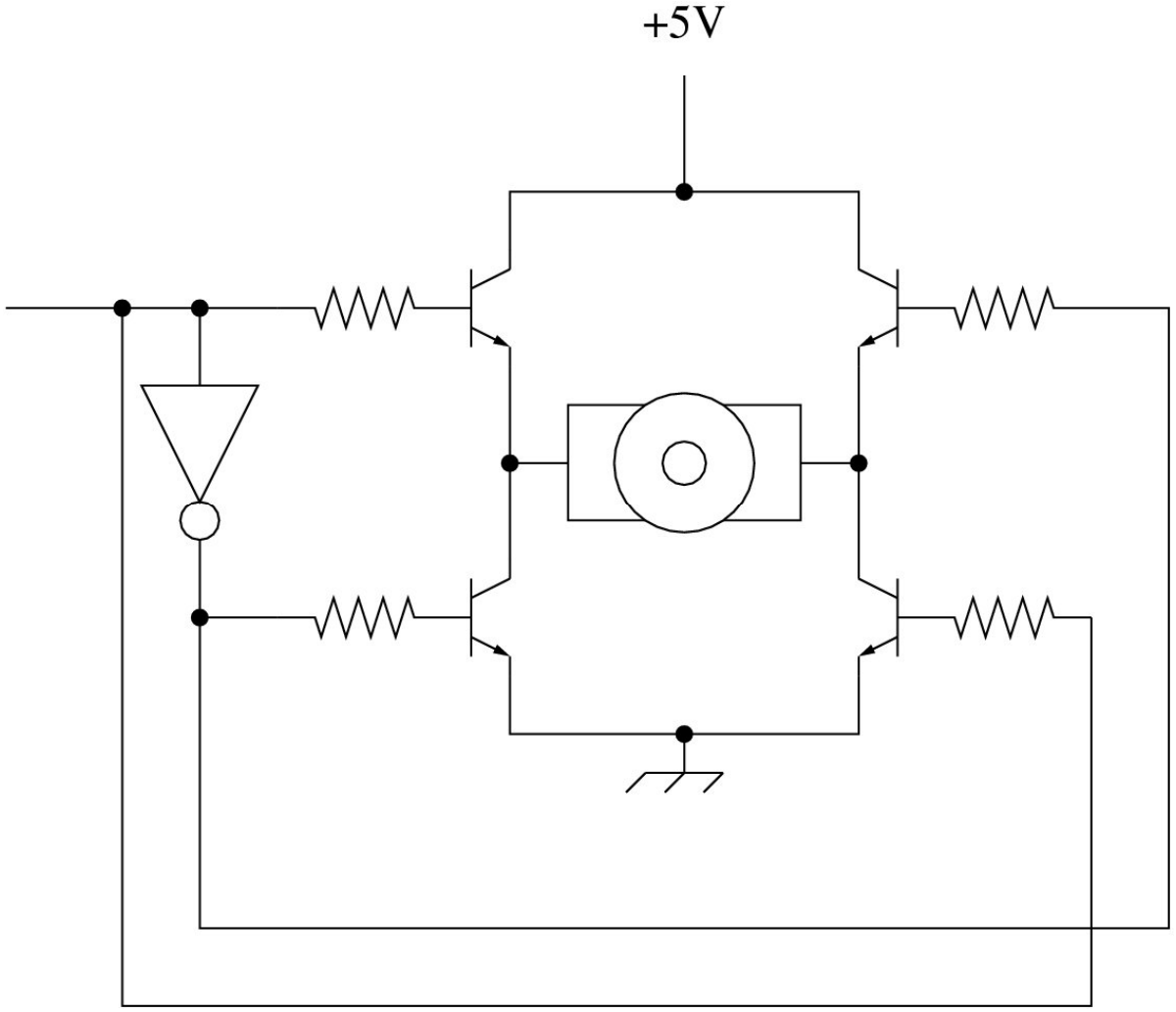
Simple H-Bridge

How can we prevent a processor from accidentally producing this case?



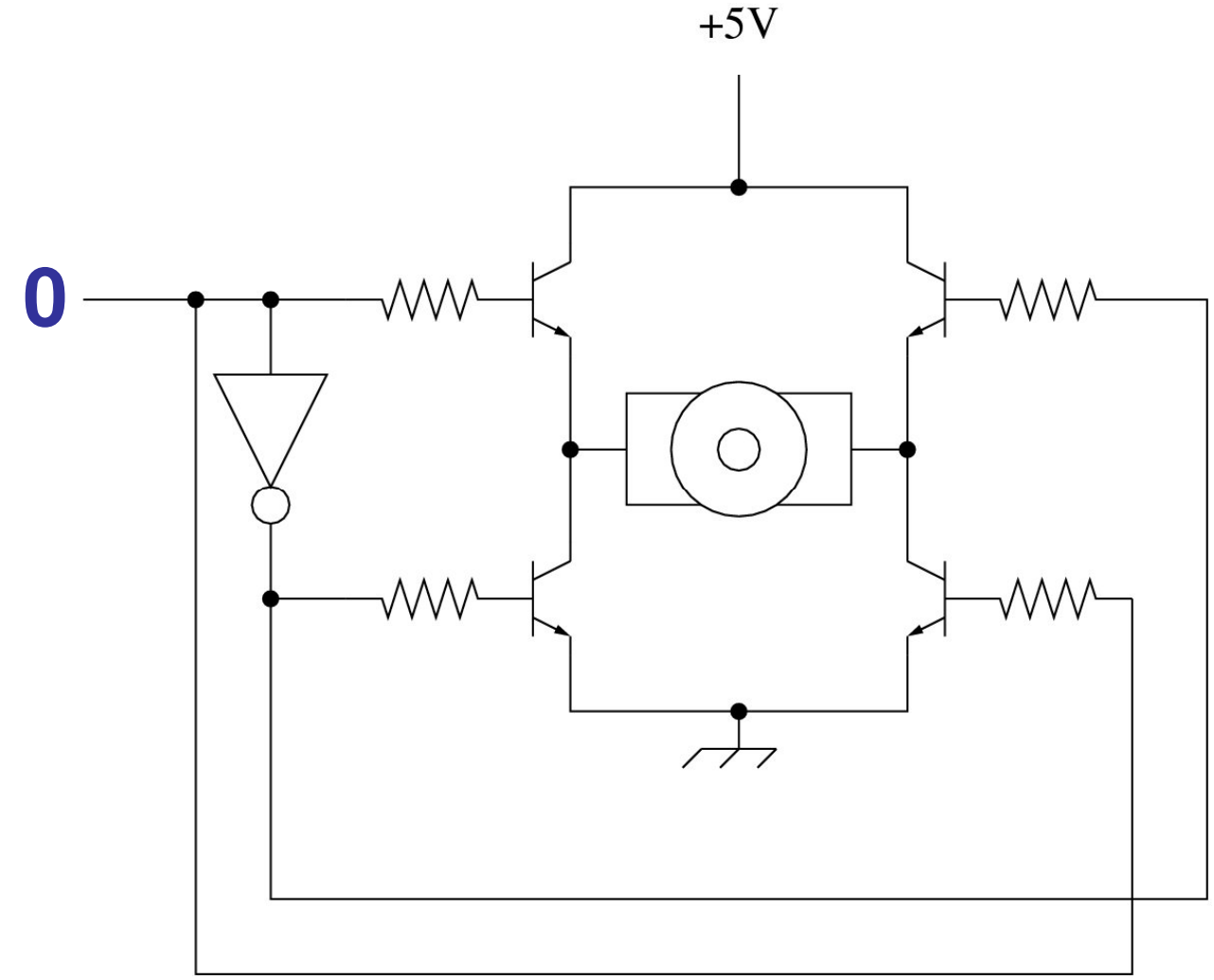
Modified H-Bridge

We introduce a little logic to ensure the short never occurs



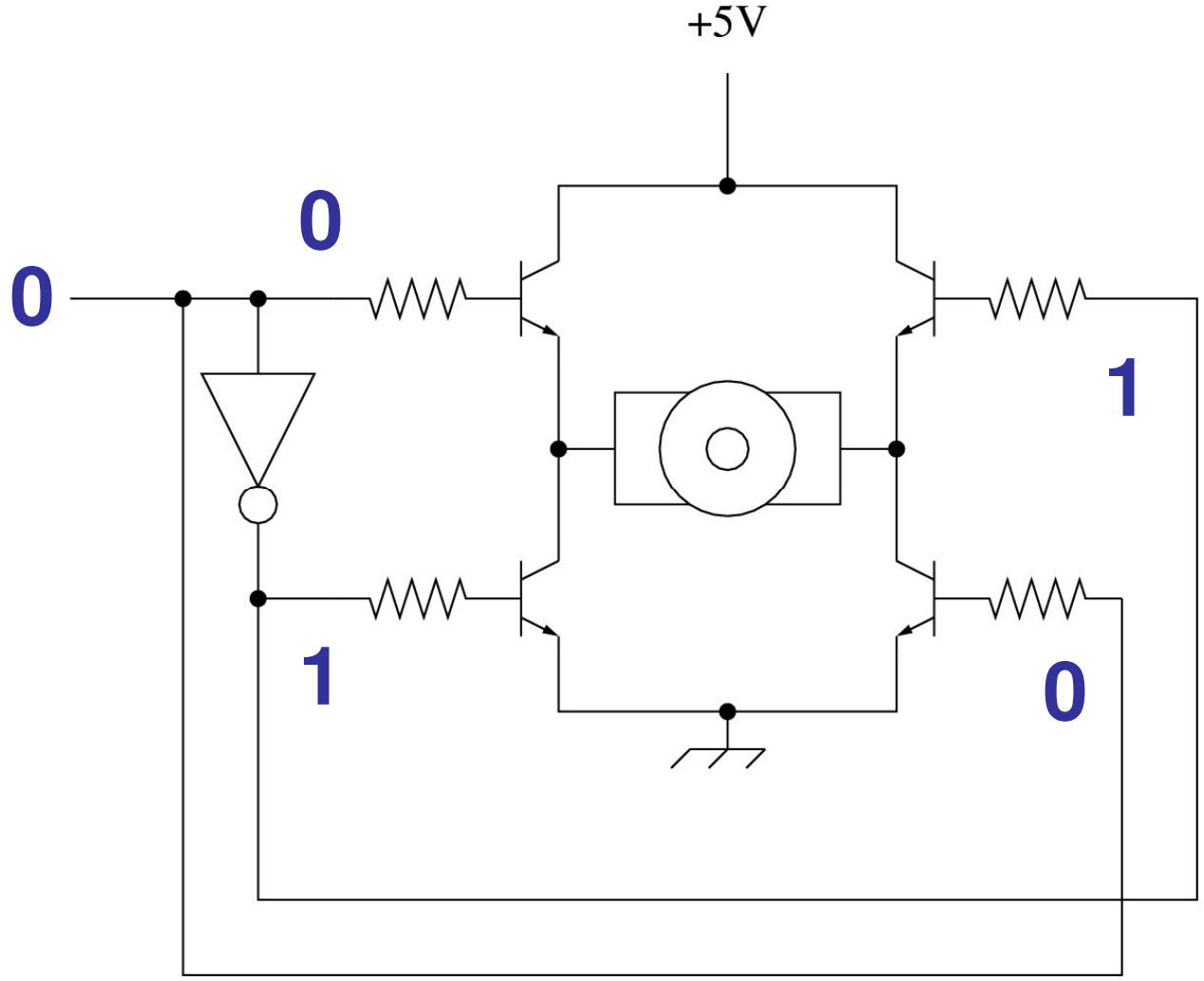
Modified H-Bridge

What happens with this input?



Modified H-Bridge

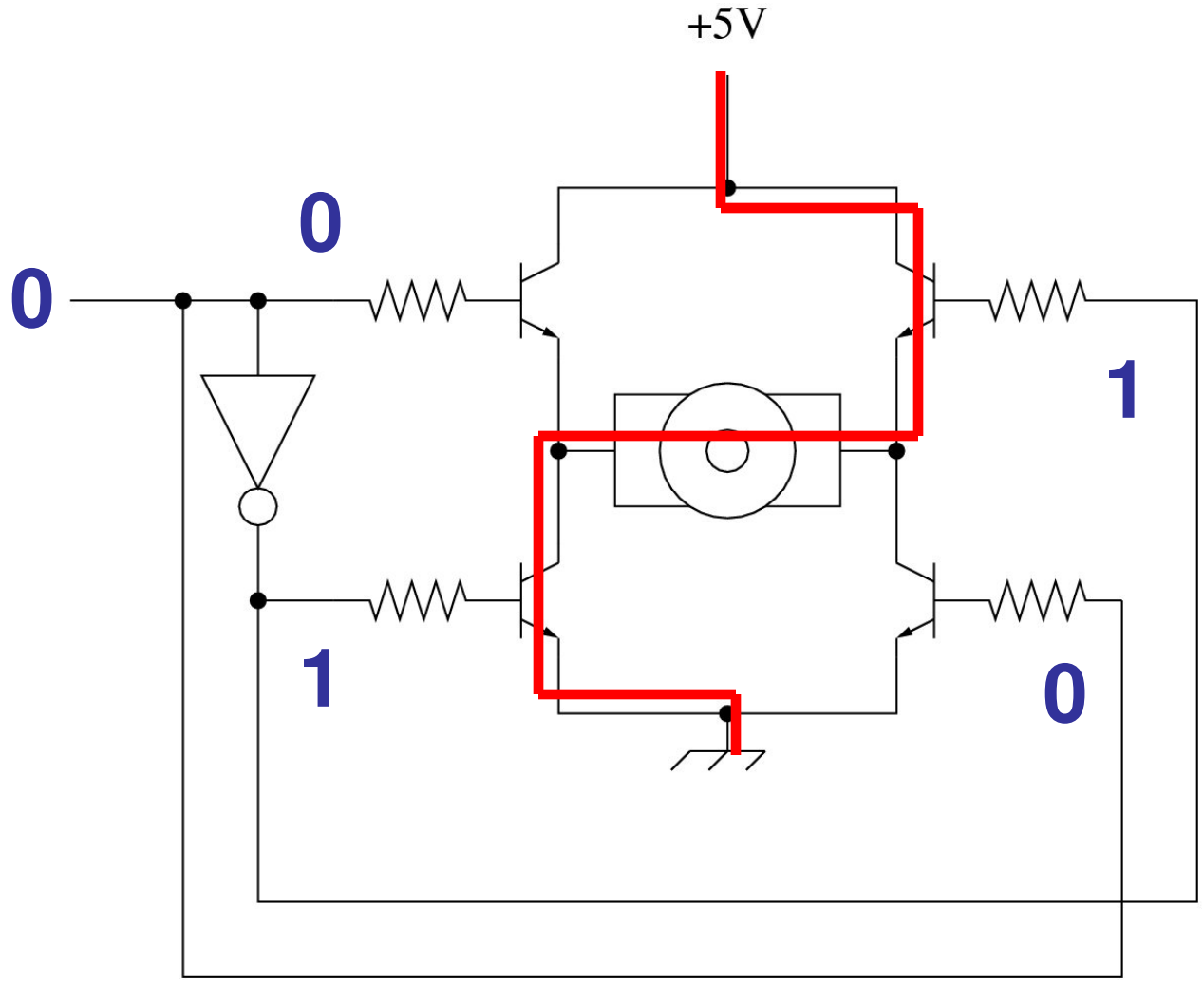
What happens with this input?



Modified H-Bridge

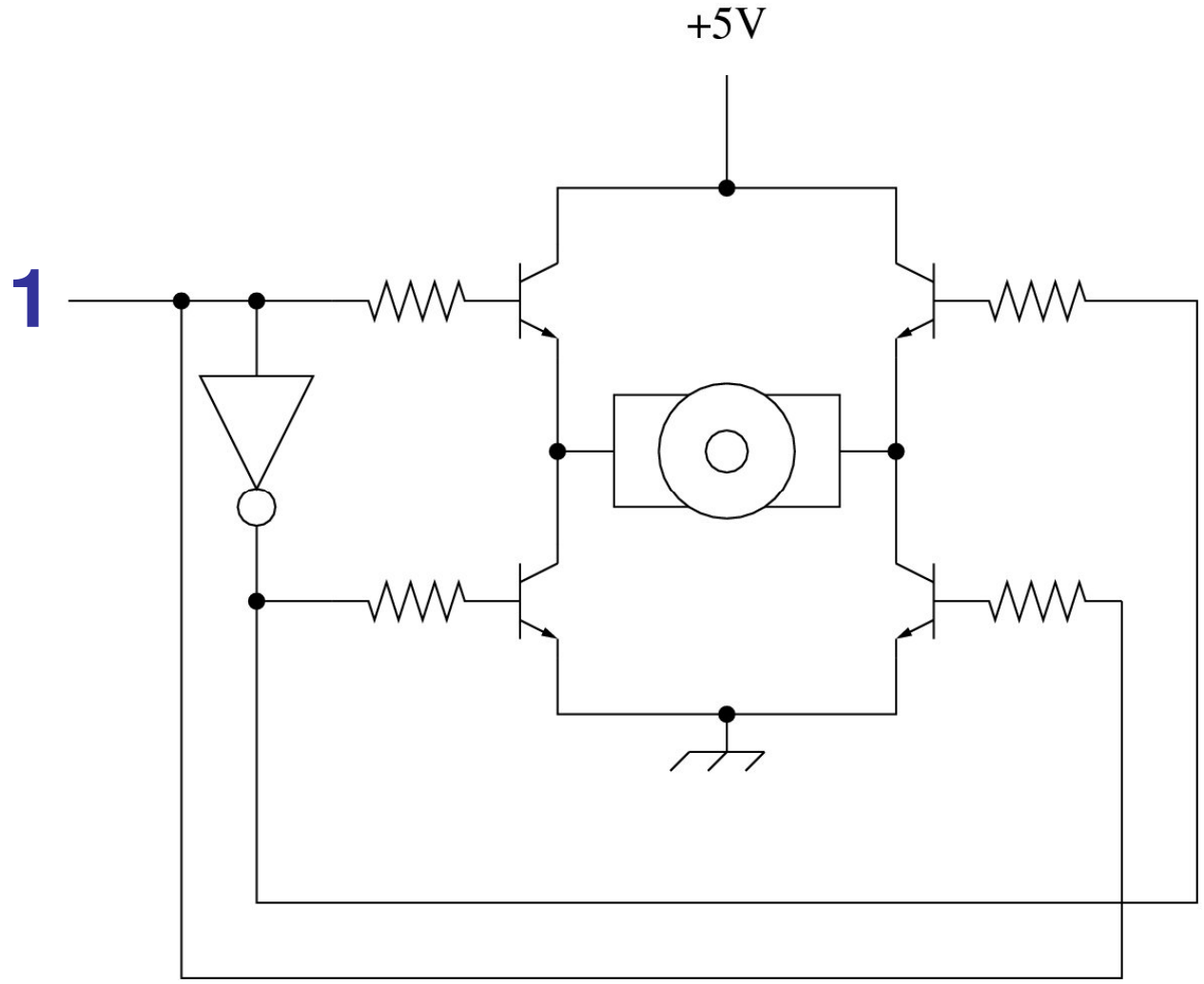
What happens with this input?

- Motor turns in one direction



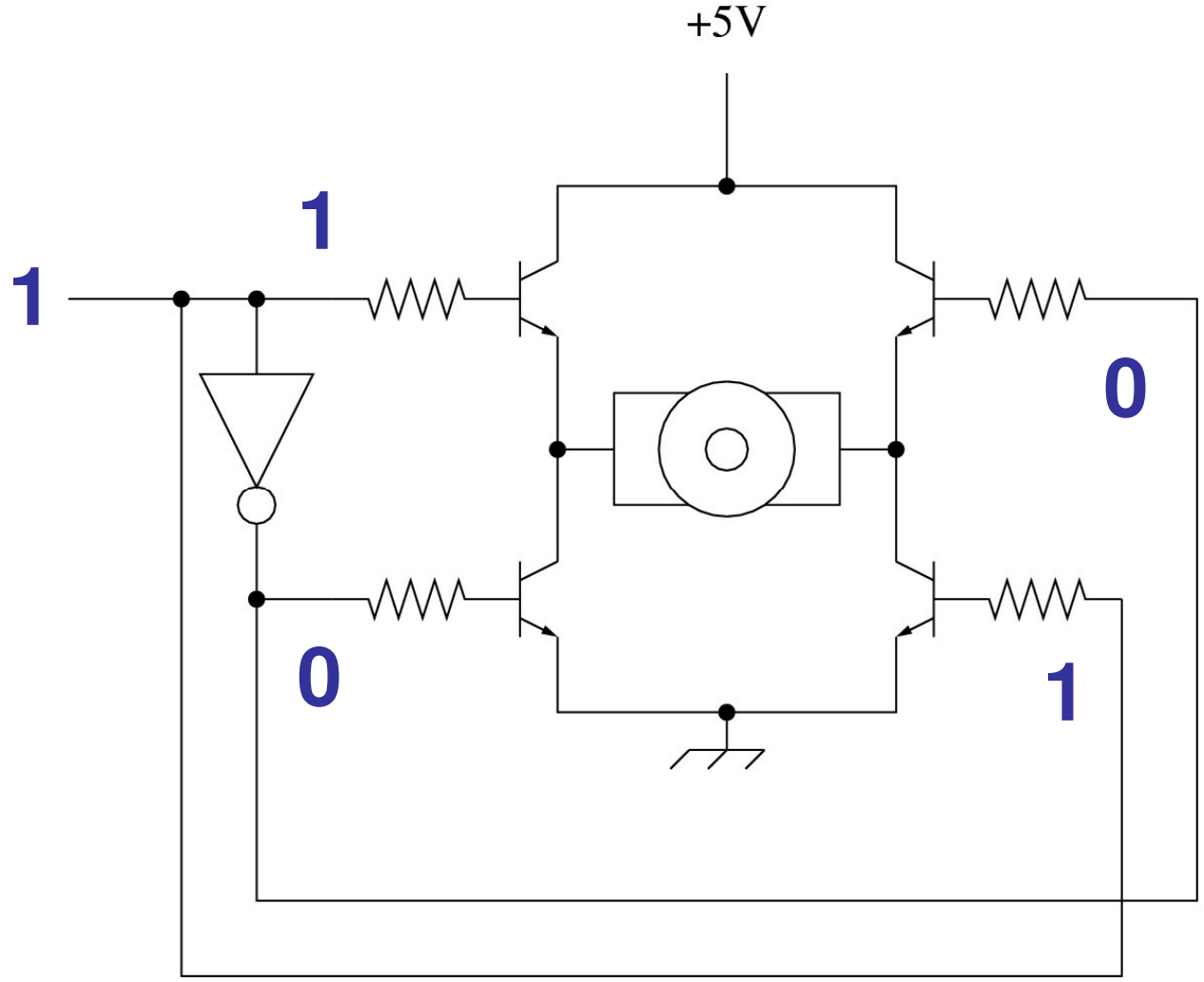
Modified H-Bridge

How about this input?



Modified H-Bridge

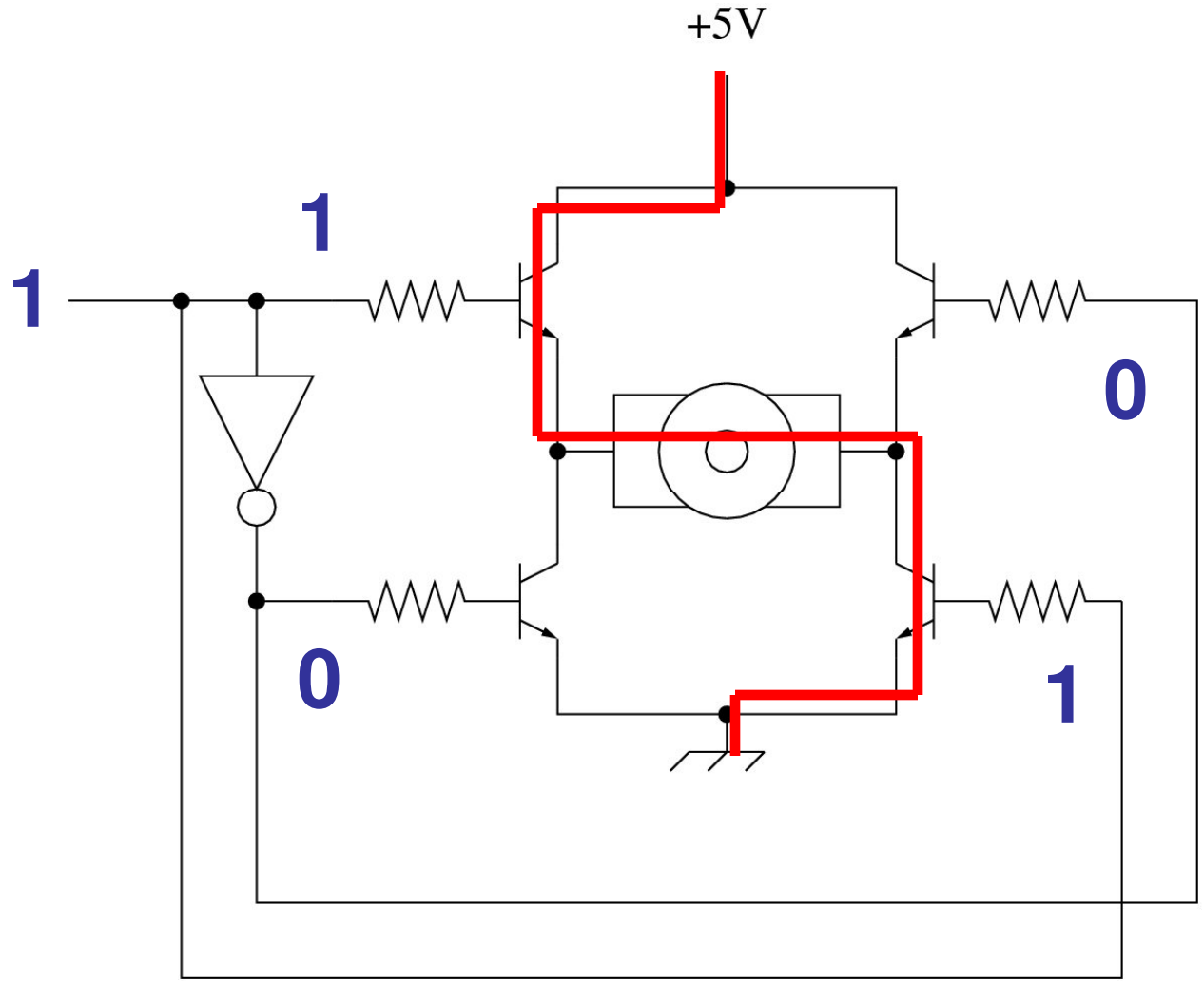
What happens with this input?



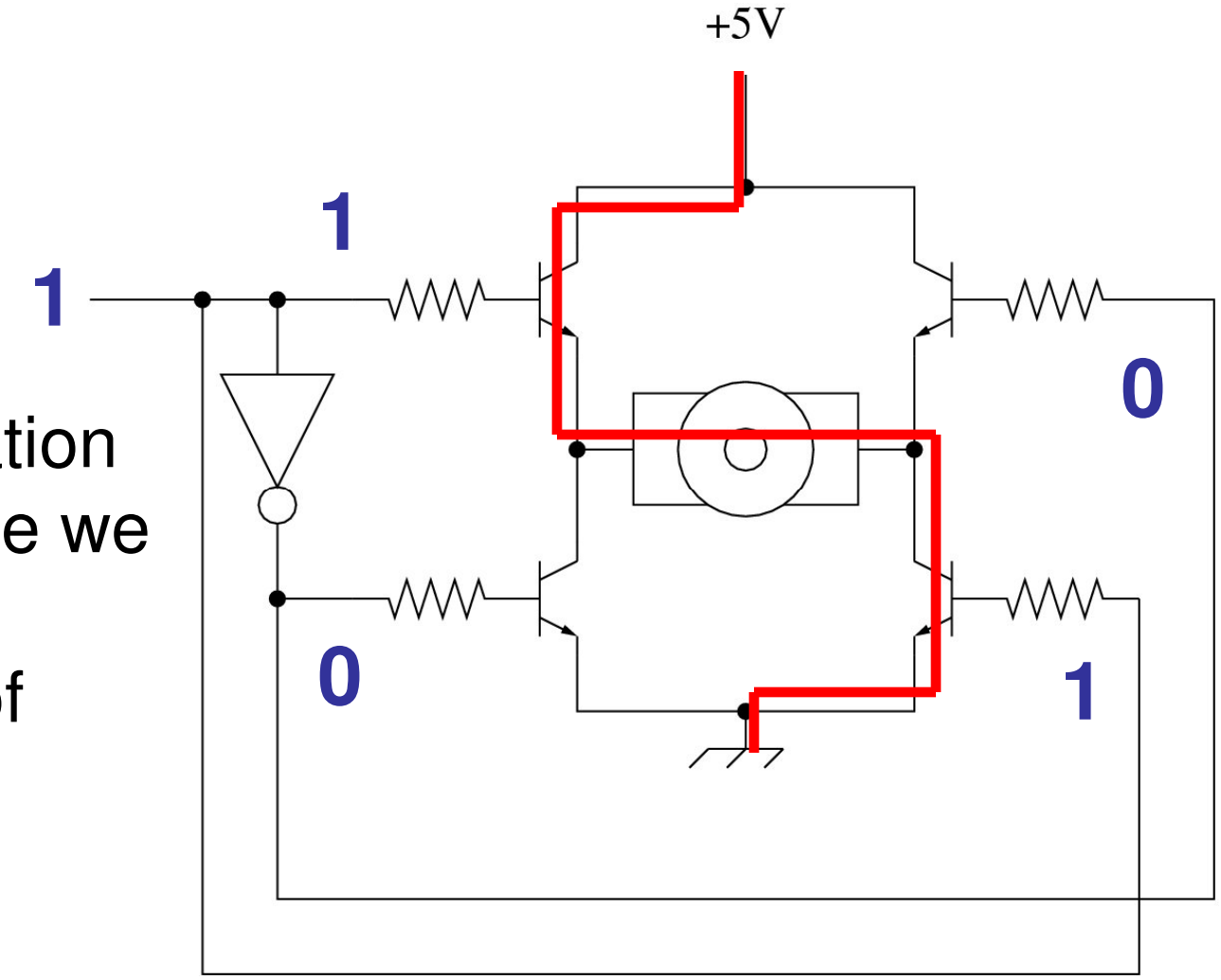
Modified H-Bridge

How about this input?

- Motor turns in the other direction



Modified H-Bridge



This implementation is nice because we only need one **direction** bit of control

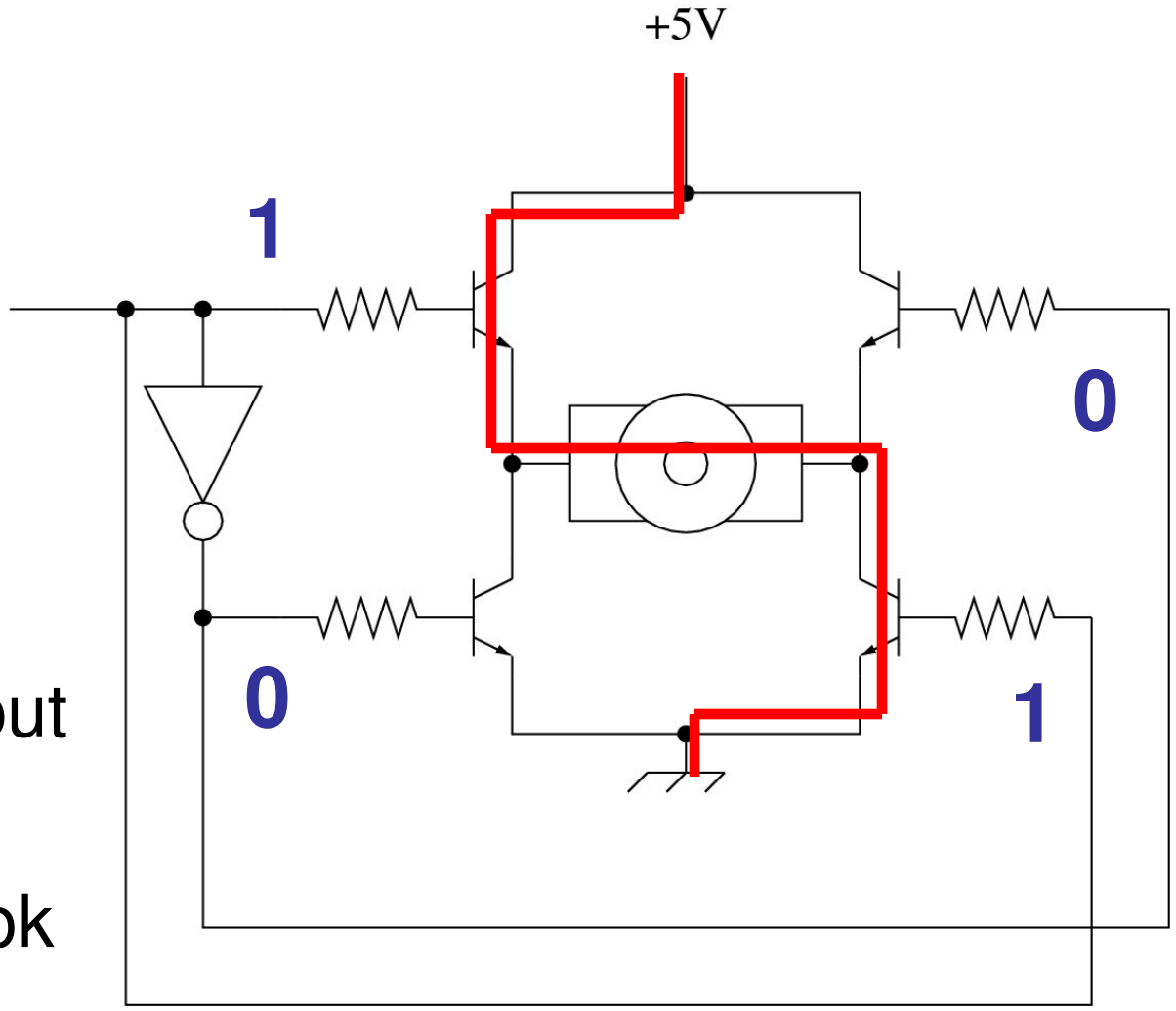
- What are we missing?

Modified H-Bridge

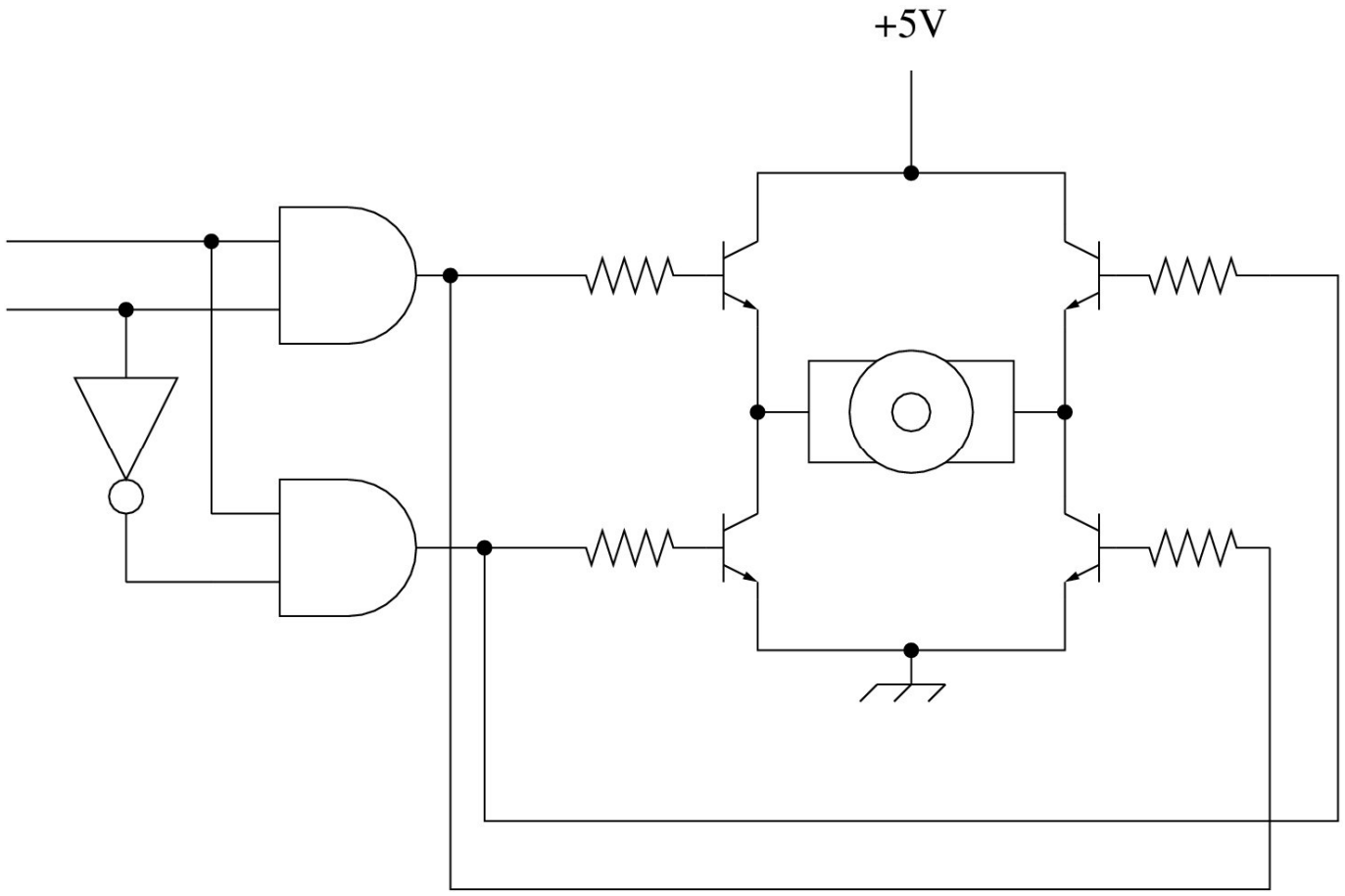
What are we missing?

- Control of torque magnitude
- Let's introduce a second PWM input

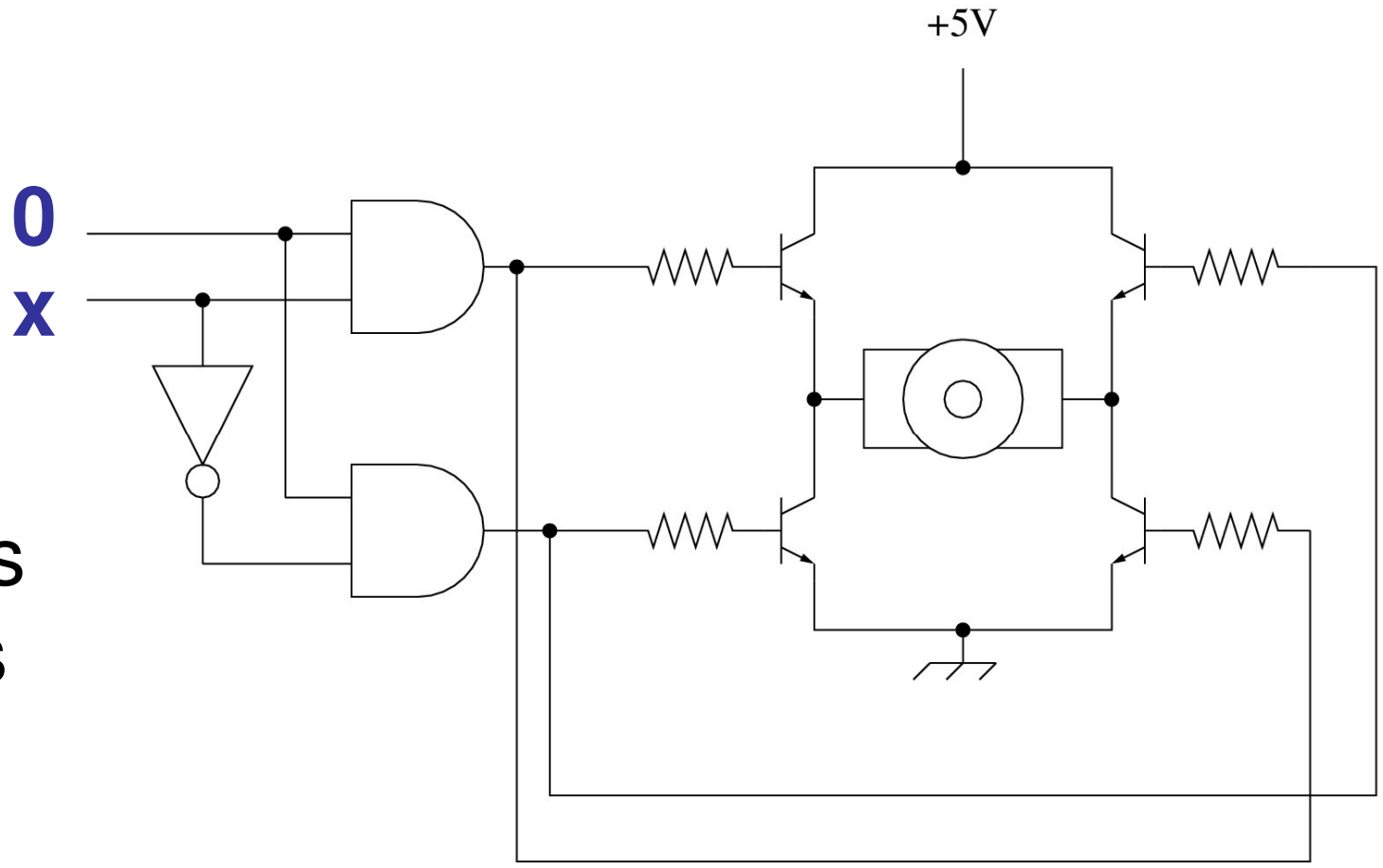
What would this look like?



PWM and Direction Control



PWM and Direction Control

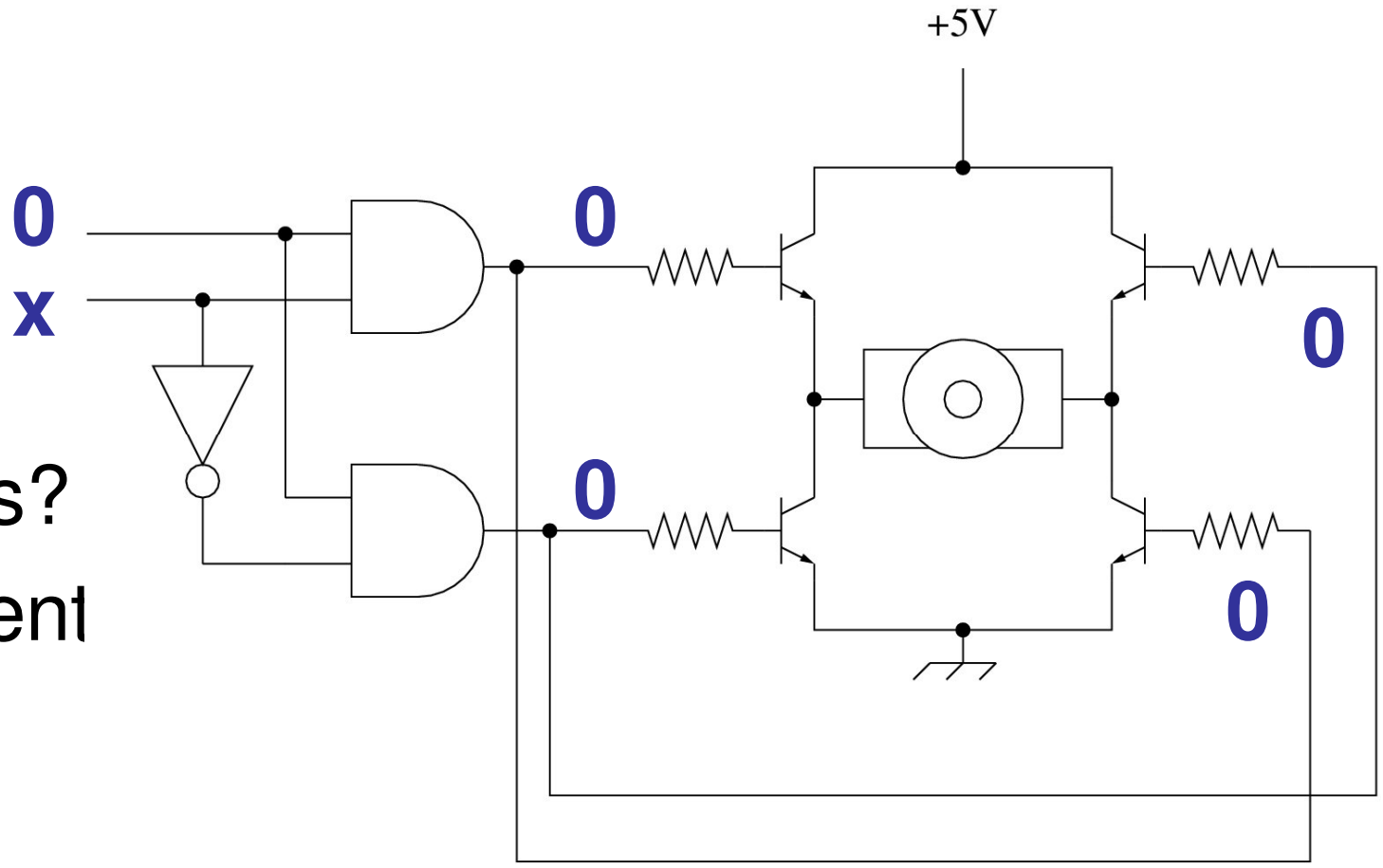


What happens with this input?

PWM and Direction Control

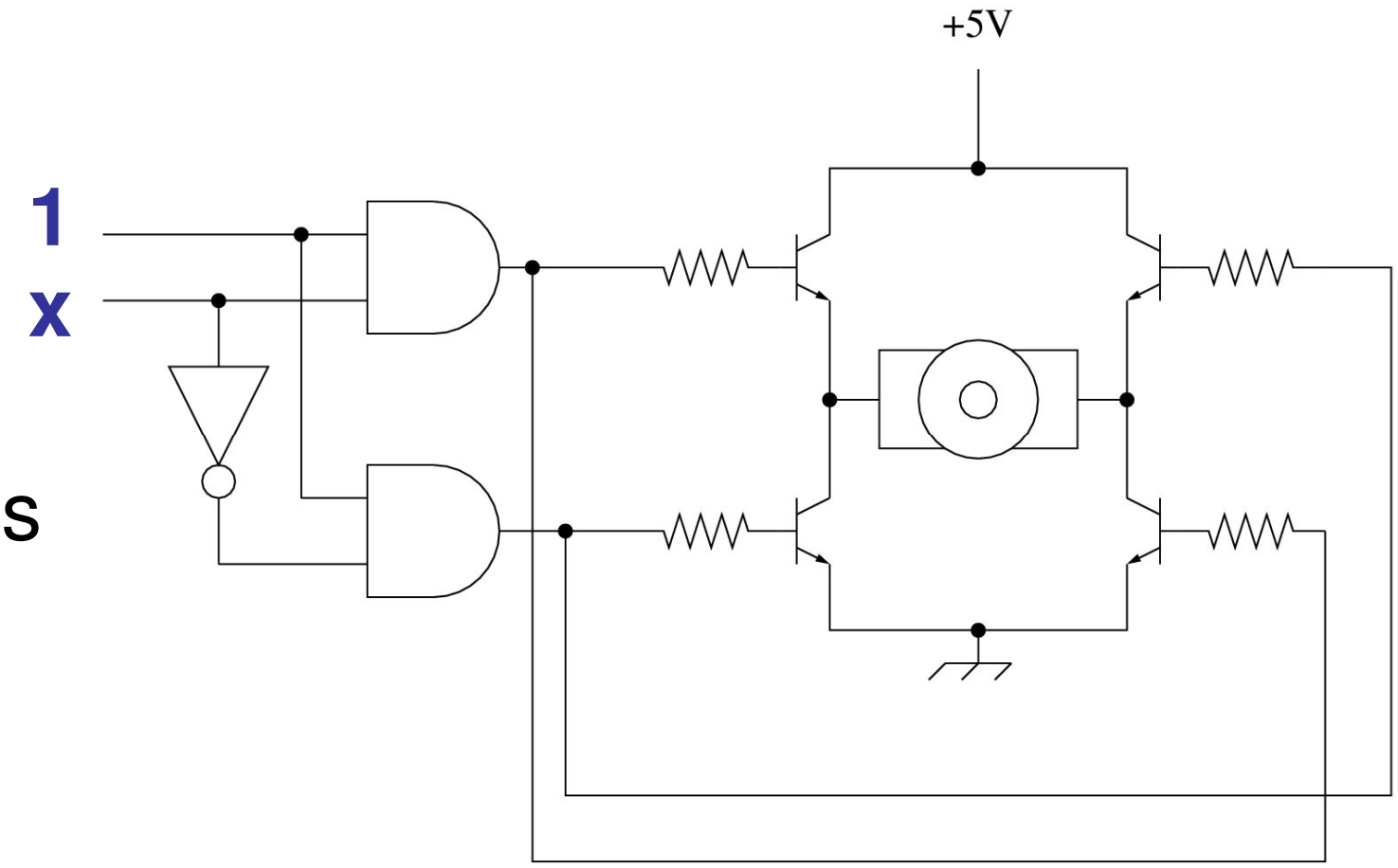
What happens?

- No current flow

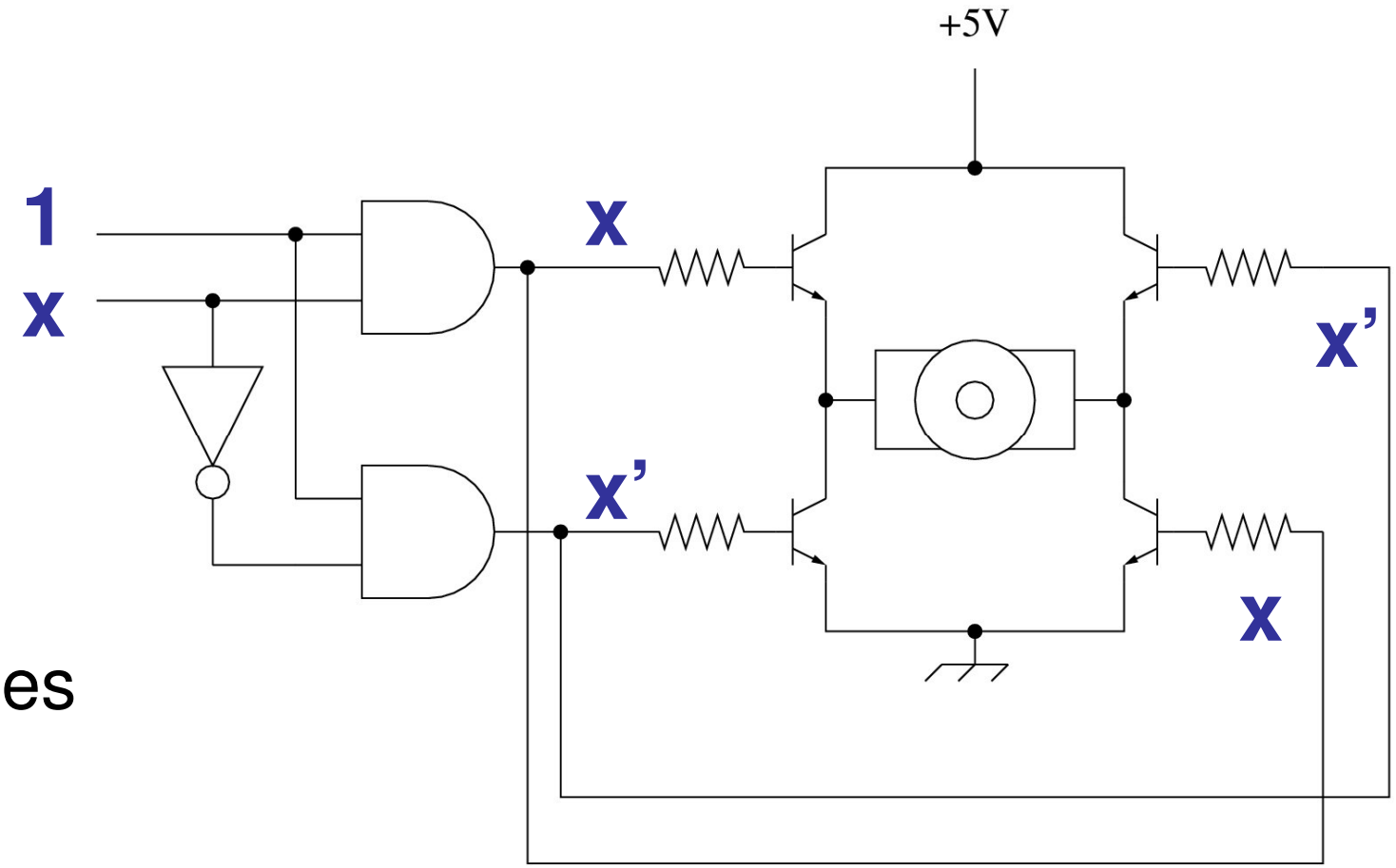


PWM and Direction Control

What happens now?



PWM and Direction Control

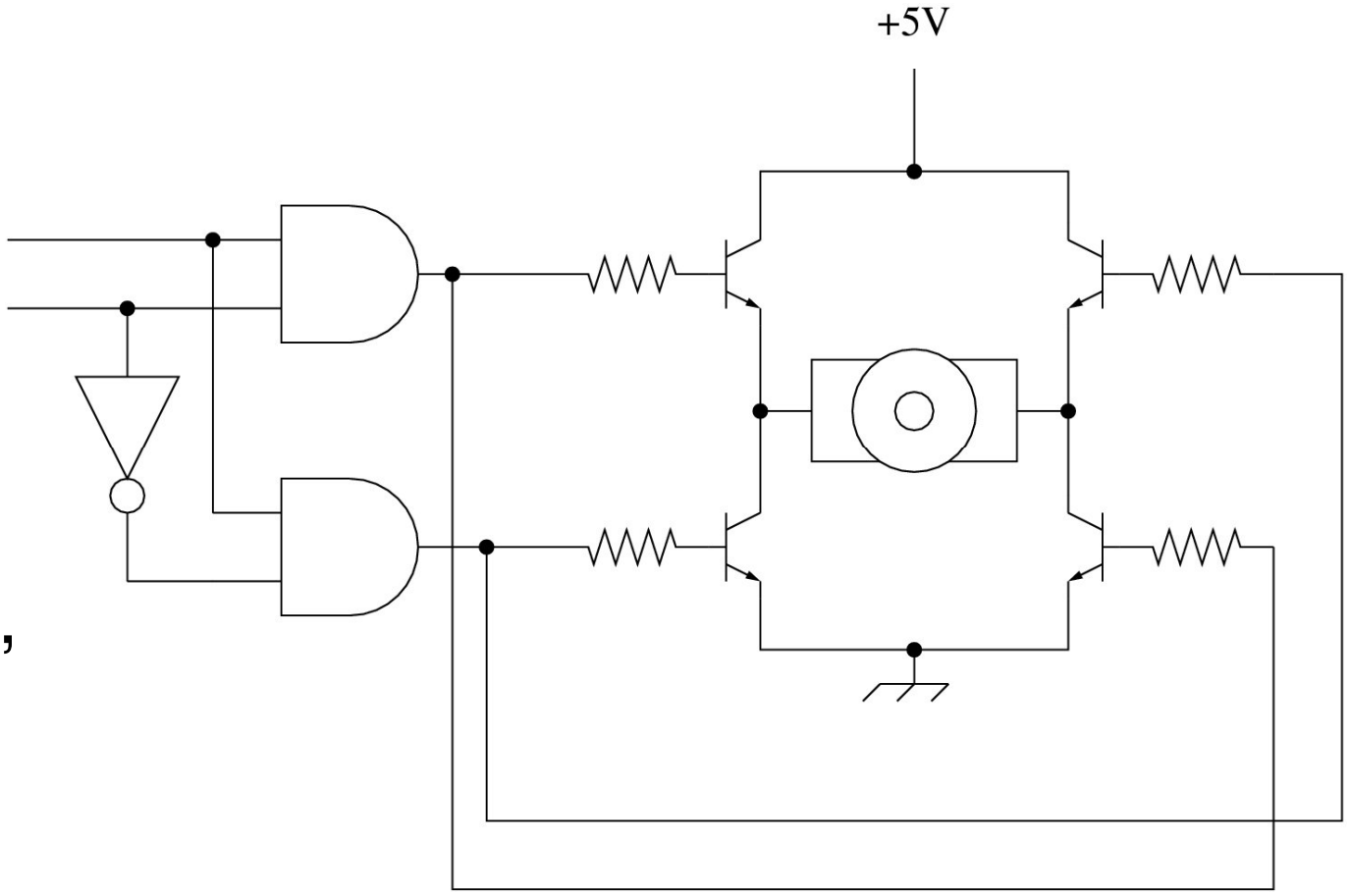


What happens now?

- 'x' determines motor direction

PWM and Direction Control

Direction



With the PWM input, we can control the magnitude of torque

Flow of Data in I/O

Back to our serial interrupt handler example…

- How does the data flow through the processor?

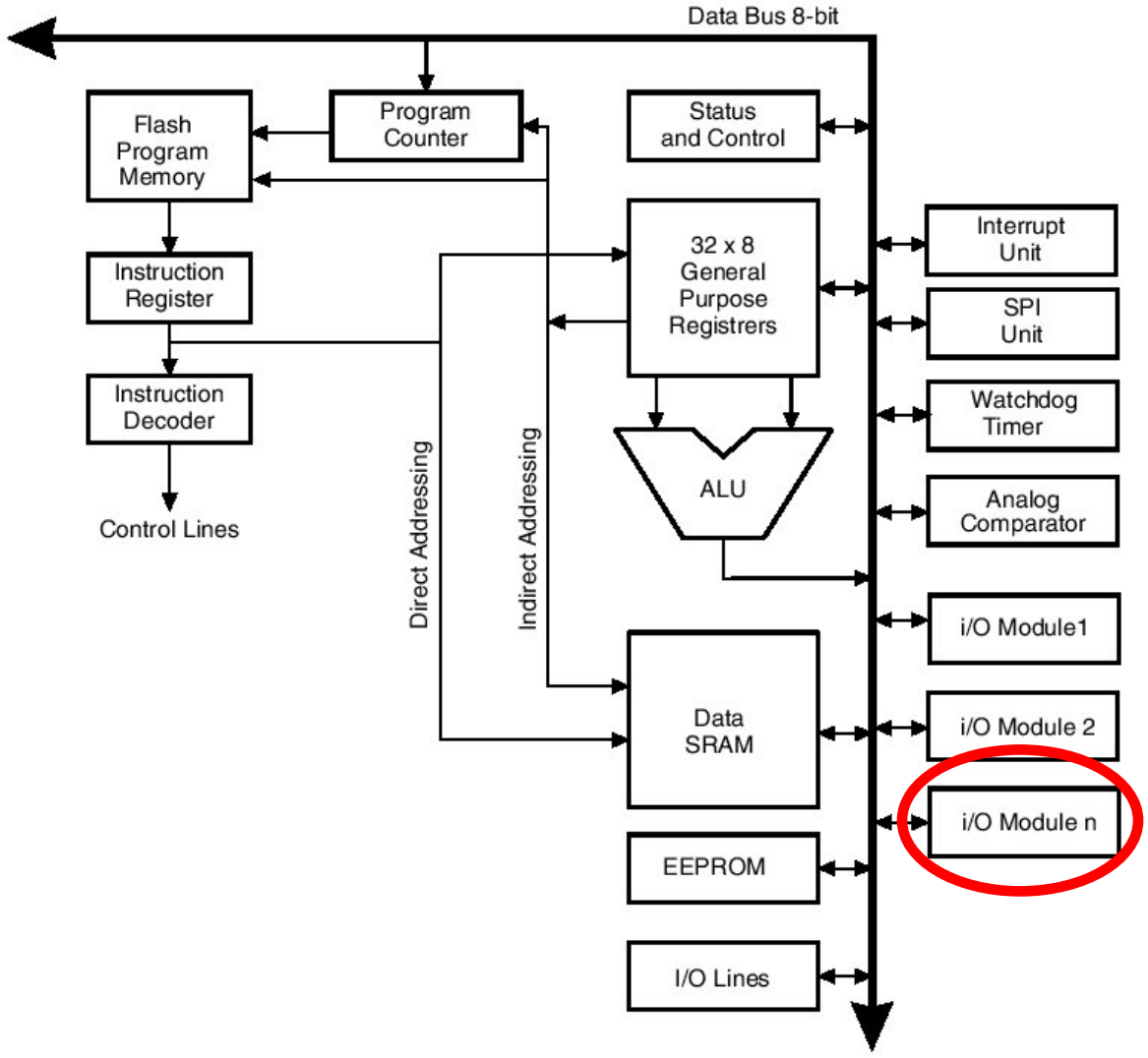
Interrupt Handler

```
SIGNAL(SIG_UART_RECV) {
  // Handle the character in the UART buffer
  int c = getchar();

  if(nchars < BUF_SIZE) {
     buffer[(front+nchars)%BUF_SIZE] = c;
     nchars += 1;
  }
}
```

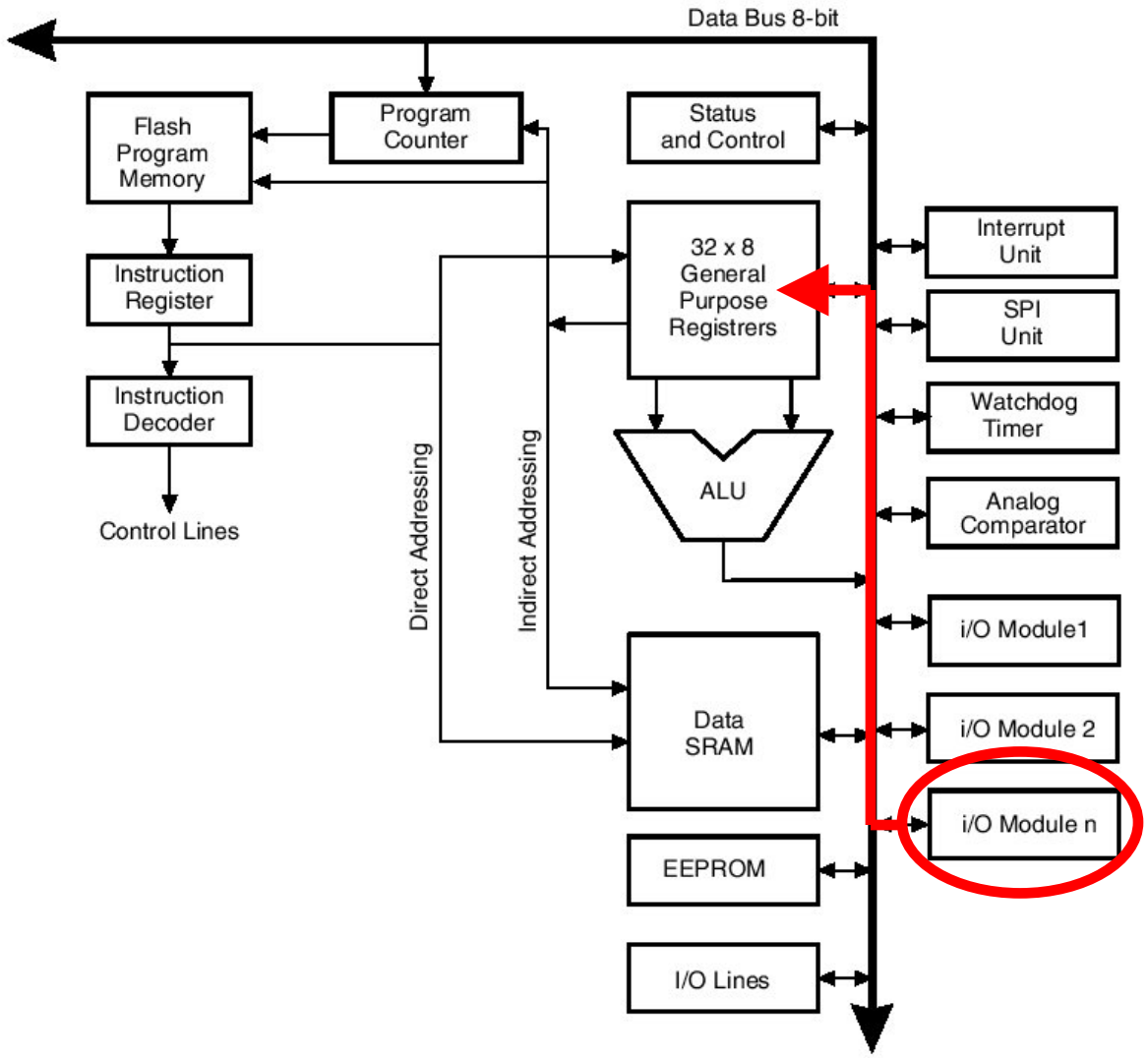
Data Flow on Each Interrupt

Byte arrives at serial device



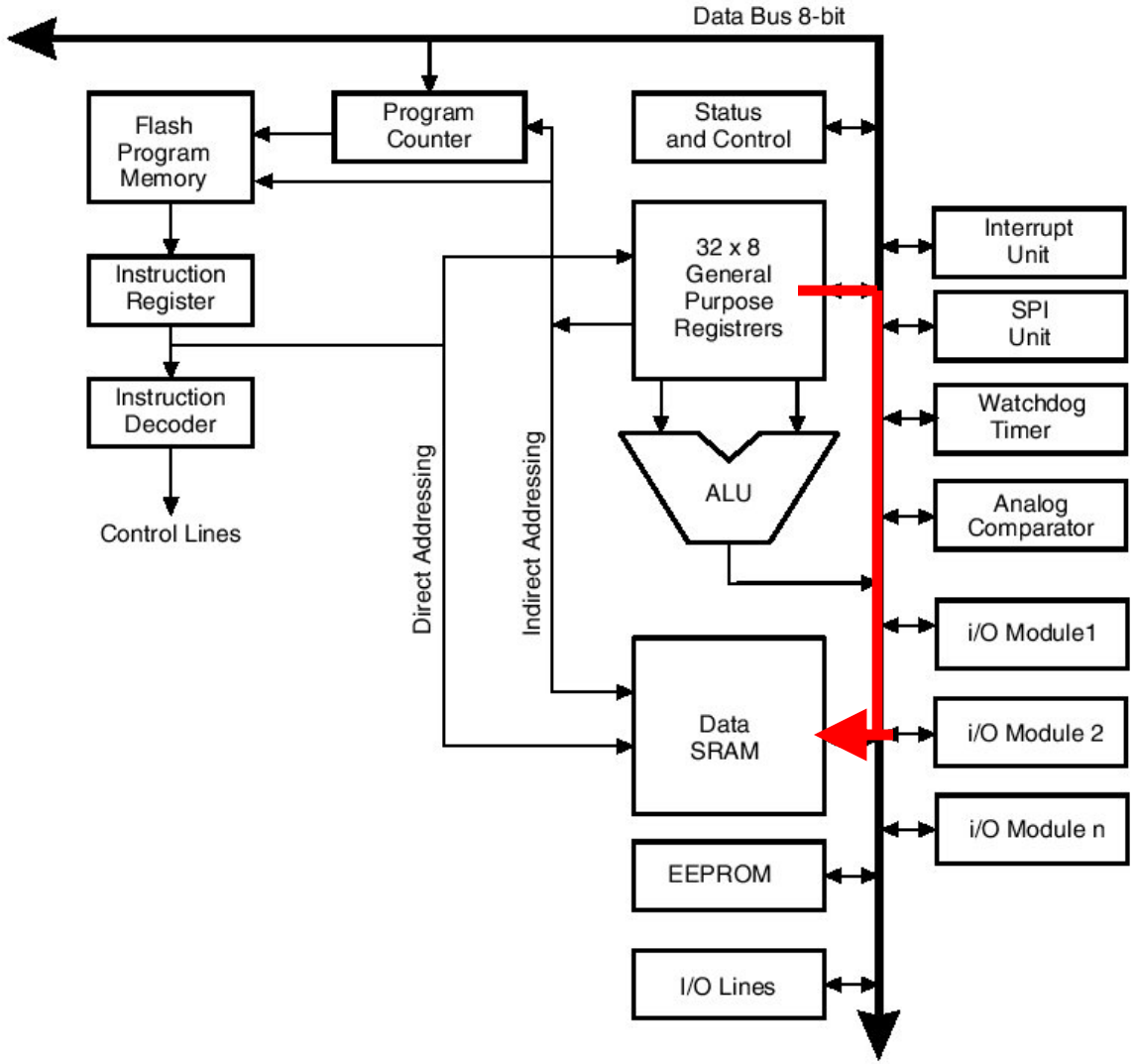
Data Flow on Each Interrupt

Interrupt routine loads byte into a register



Data Flow on Each Interrupt

Interrupt routine then writes byte out to buffer in RAM



Flow of Data in I/O

With each transfer:

- The byte value moves from the device to a register
- And then moves from the register to RAM

This is OK when we have very little data to move

- But: when there is a lot of data, we can waste a lot of CPU time in this double transfer

Moving a Lot of Data

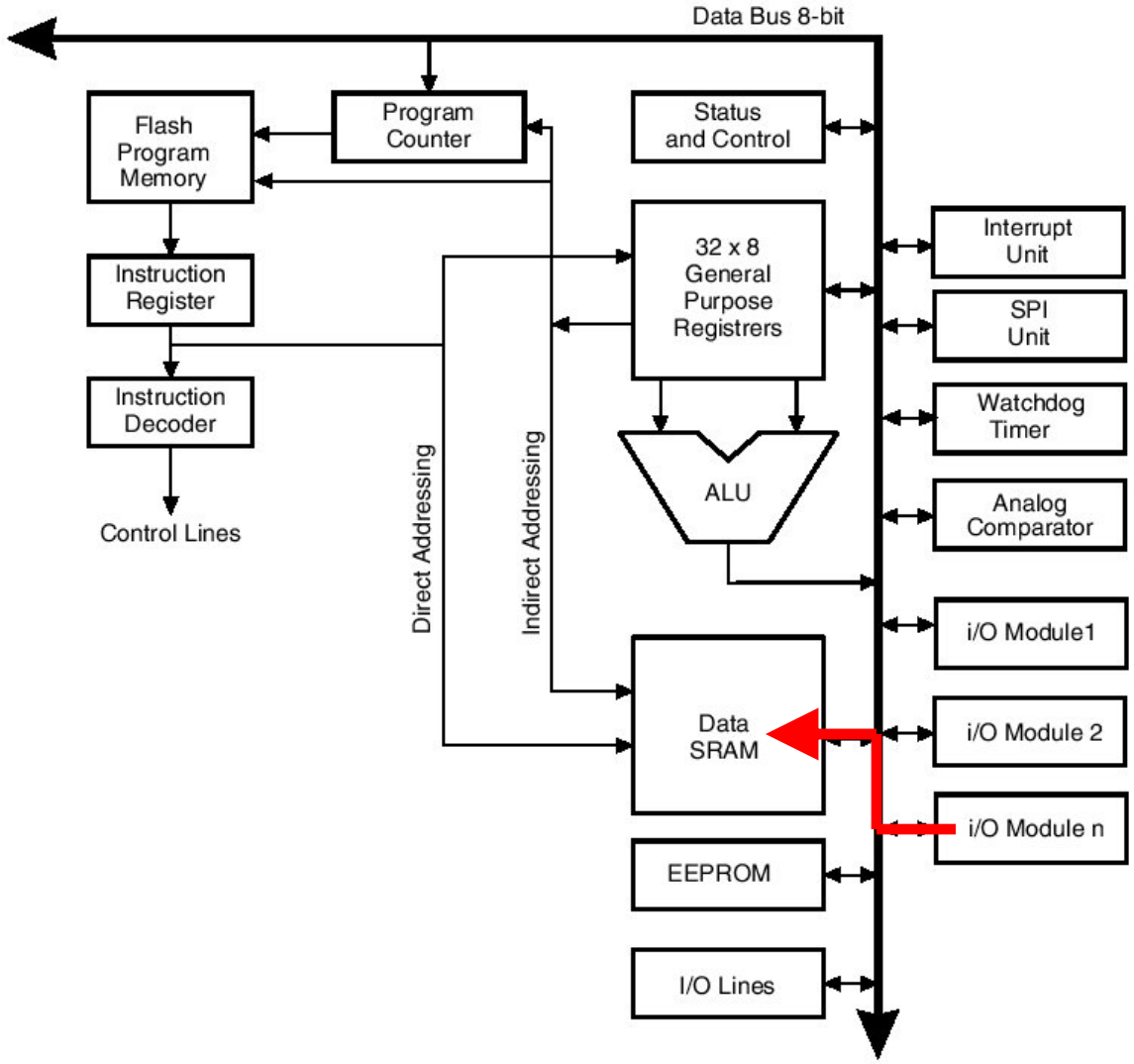
Direct memory access:

- CPU gives control of the data bus to the device itself
- Device generates the address and read/write signals
- Once transfer is complete, CPU takes control back

Data Flow During DMA

Device writes data directly into RAM

- Many bytes are transferred at a time



Data Flow During DMA

- This data flow technique is common in video, audio, and disk transfers
- Enables the CPU to perform some operations in parallel

- Note: the mega8 itself does not support DMA (but your home computer does)

Next Time

- Device communication
- Project 4