Last Time

- Memory design

Today

- Finite state machines:
  - Model of sequential behavior
  - As applied to control
- Lab 2 specification

Administrivia

- Project 1 due today
  - Reports by 5:00pm
- Homework 2 hand-in:
  - See Mark's announcement on blackboard
- Quiz 1:
  - Available for pickup now

Homework 1

- Pick up graded assignments from Mark any time after class
- Questions about solution set now?

Finite State Machines (FSMs)

Pure FSM form is composed of:

- A set of states
- A set of possible inputs (or events)
- A set of possible outputs
- A transition function:
  - Given the current state and an input: defines the output and the next state

Finite State Machines (FSMs)

States:

- Represent all possible “situations” that must be distinguished
- At any given time, the system is in exactly one of the states
- There is a finite number of these states

Finite State Machines (FSMs)

An example: our synchronous counter

- States: ?

Finite State Machines (FSMs)

An example: our synchronous counter

- States: the different combinations of the digits: 000, 001, 010, … 111

- Inputs: ?

Finite State Machines (FSMs)

An example: our synchronous counter

- Inputs:
  - Really only one: the event associated with the clock transitioning from high to low
  - We will call this “C”

- Outputs: ?

Finite State Machines (FSMs)

An example: our synchronous counter

- Outputs: same as the set of states

- Transition function: ?

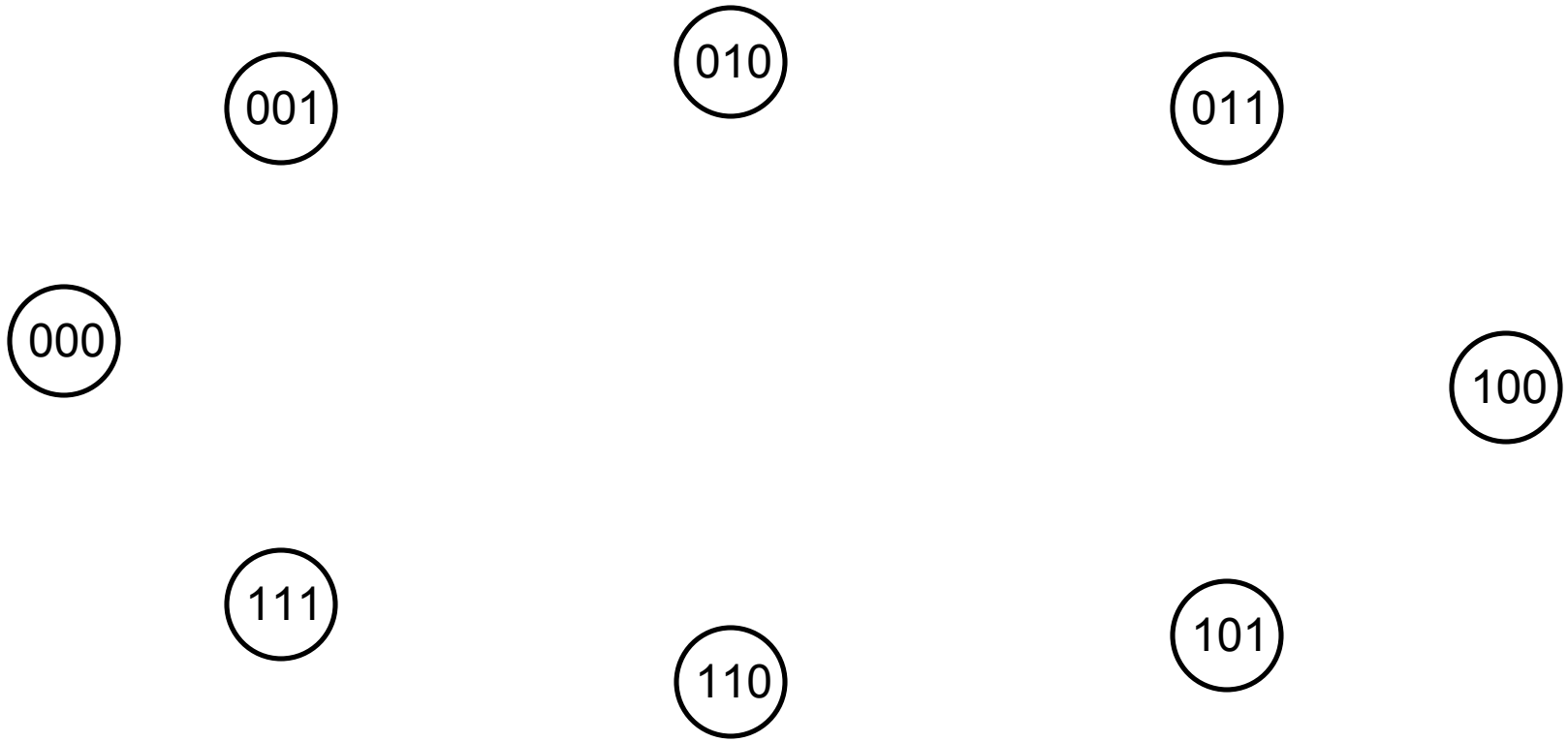
Finite State Machines (FSMs)

An example: our synchronous counter

- Transition function:
  - On the clock event, transition to the next state in the sequence

FSM Example: Synchronous Counter

A Graphical Representation:



A set of states

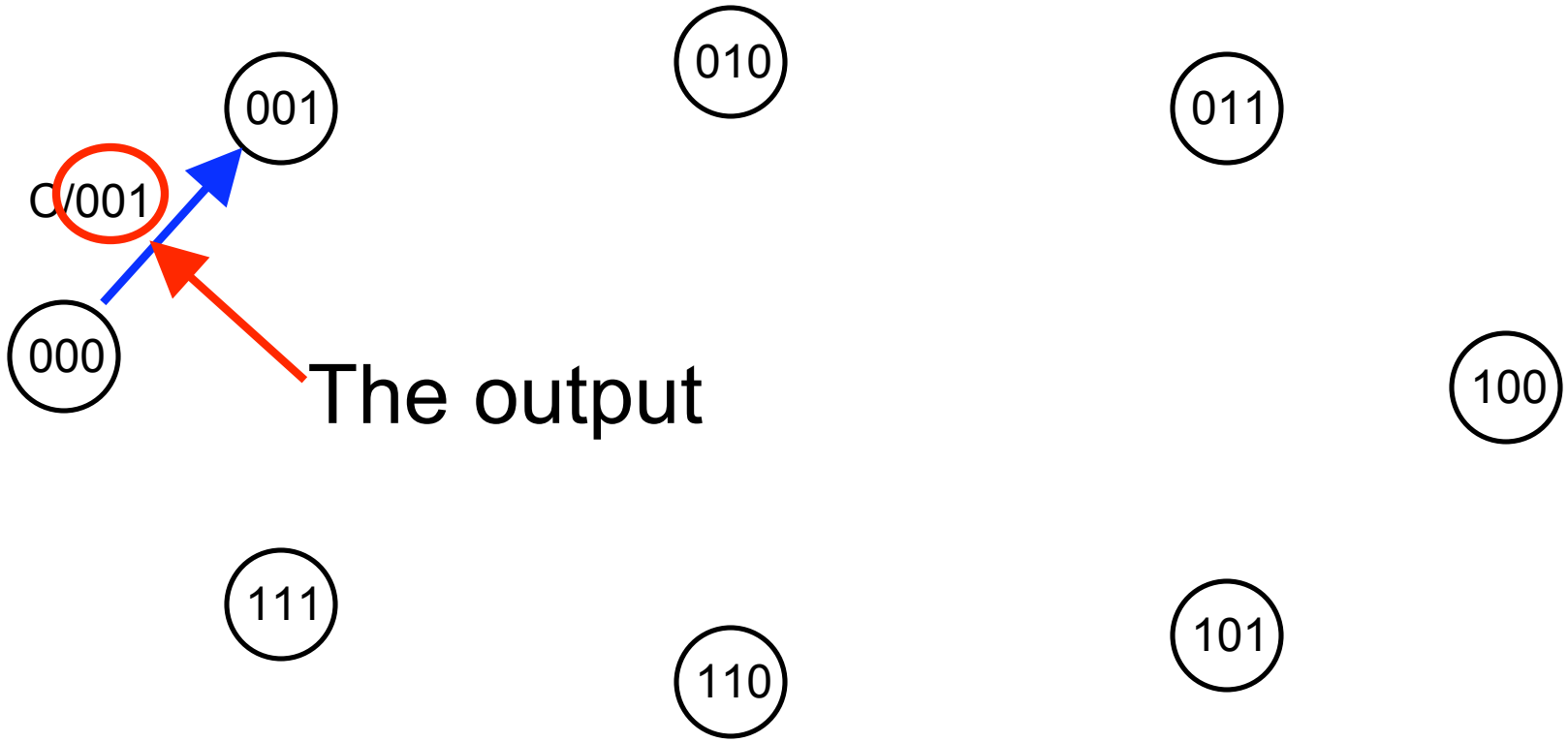
FSM Example: Synchronous Counter

FSM Example: Synchronous Counter

A transition

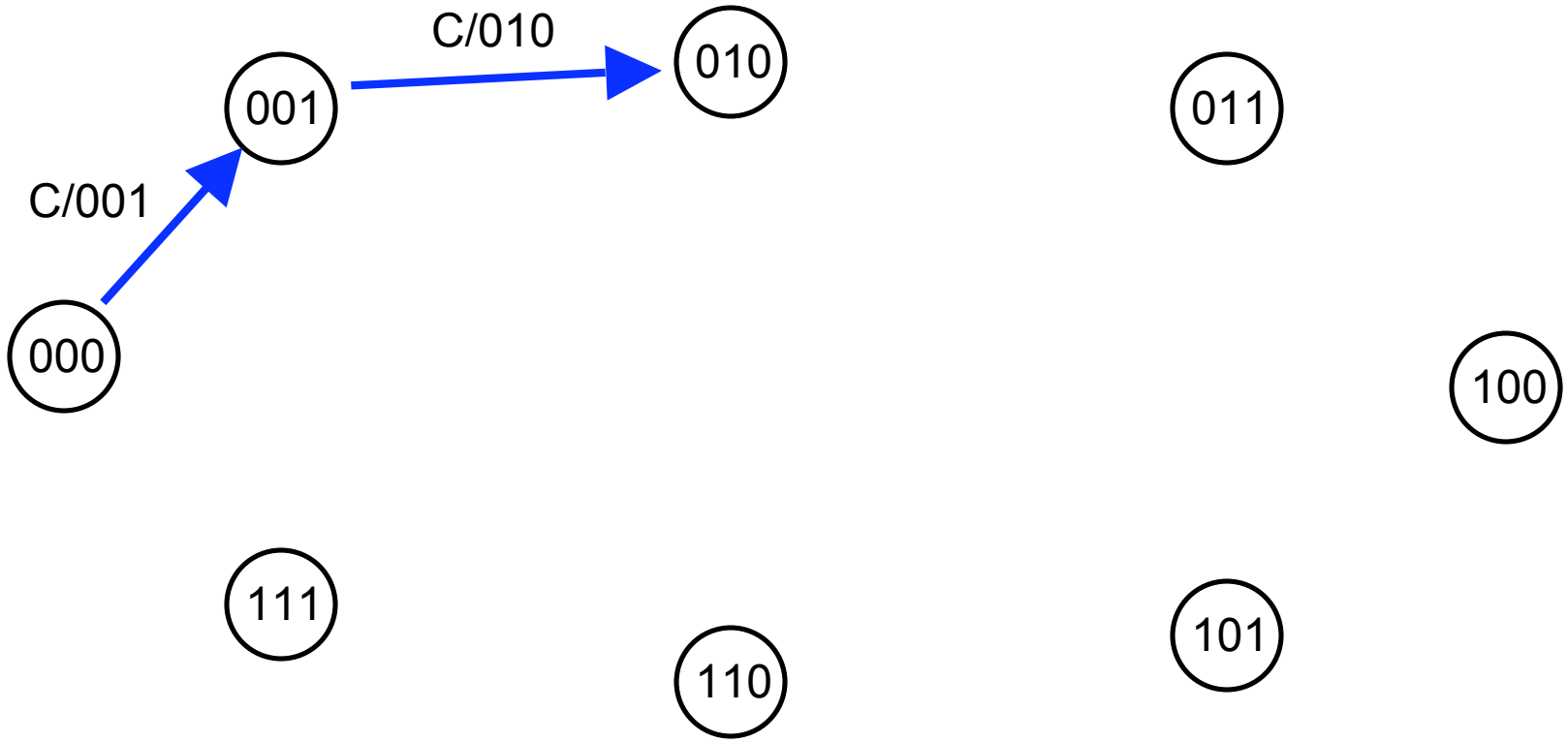
FSM Example: Synchronous Counter

A transition



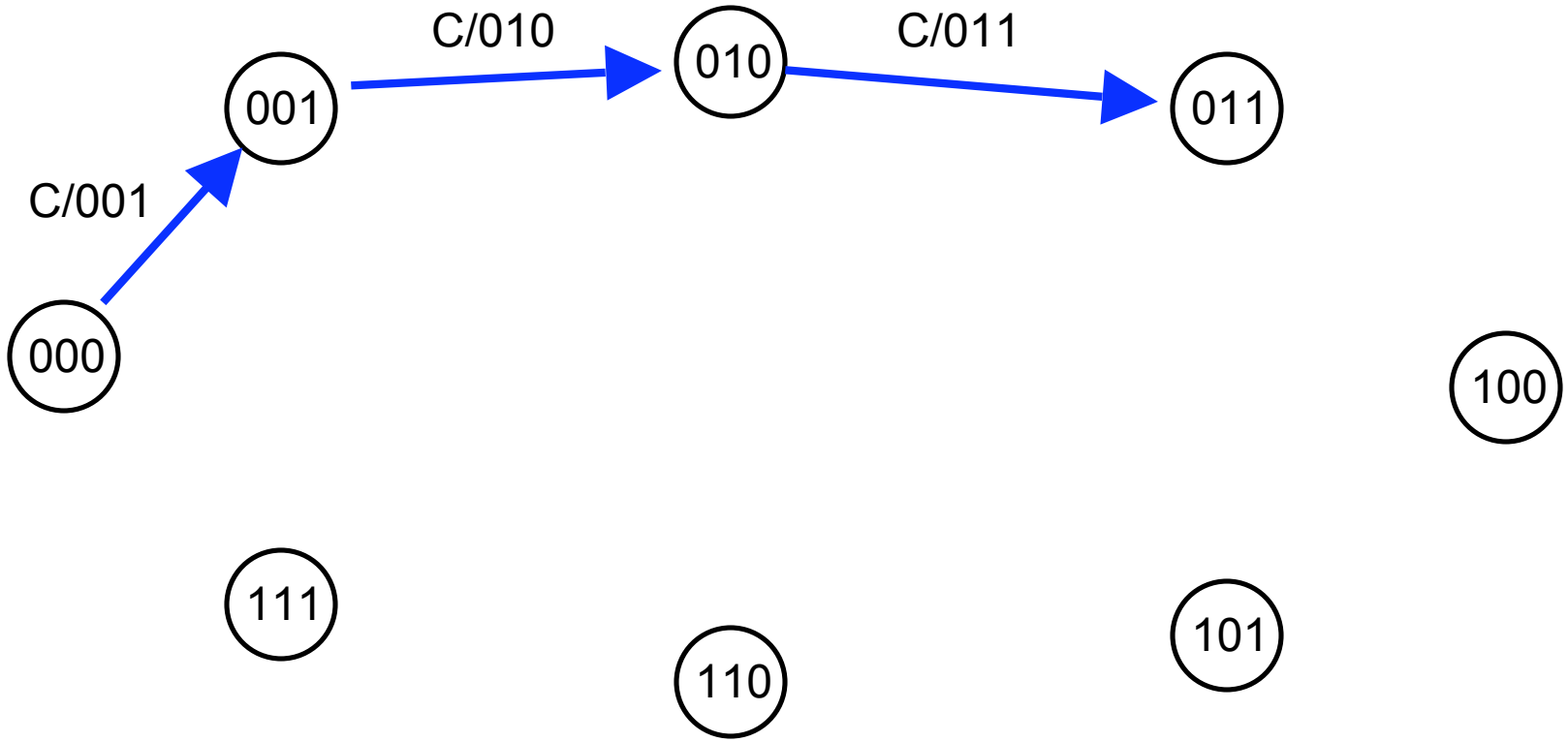
FSM Example: Synchronous Counter

The next transition



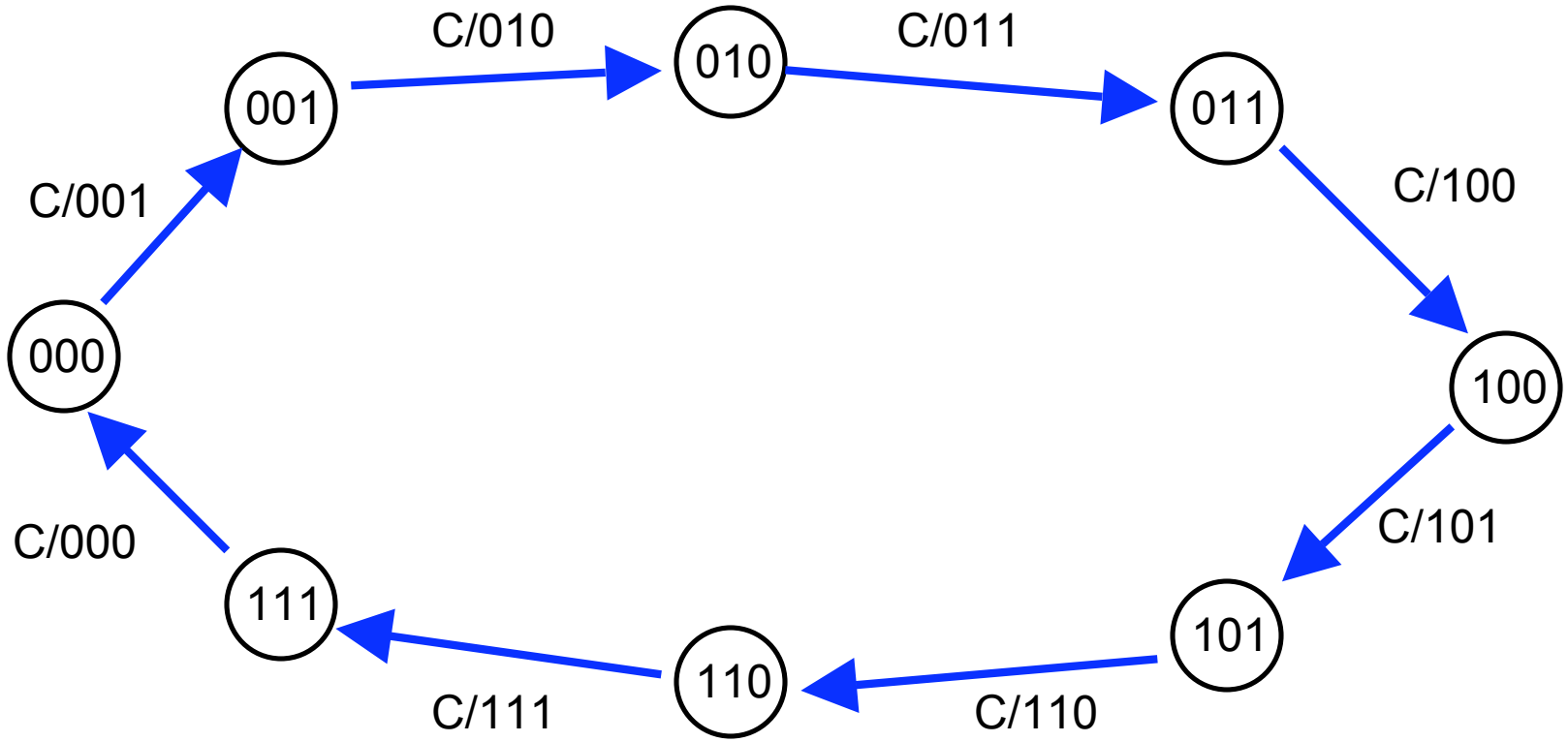
FSM Example: Synchronous Counter

The next transition

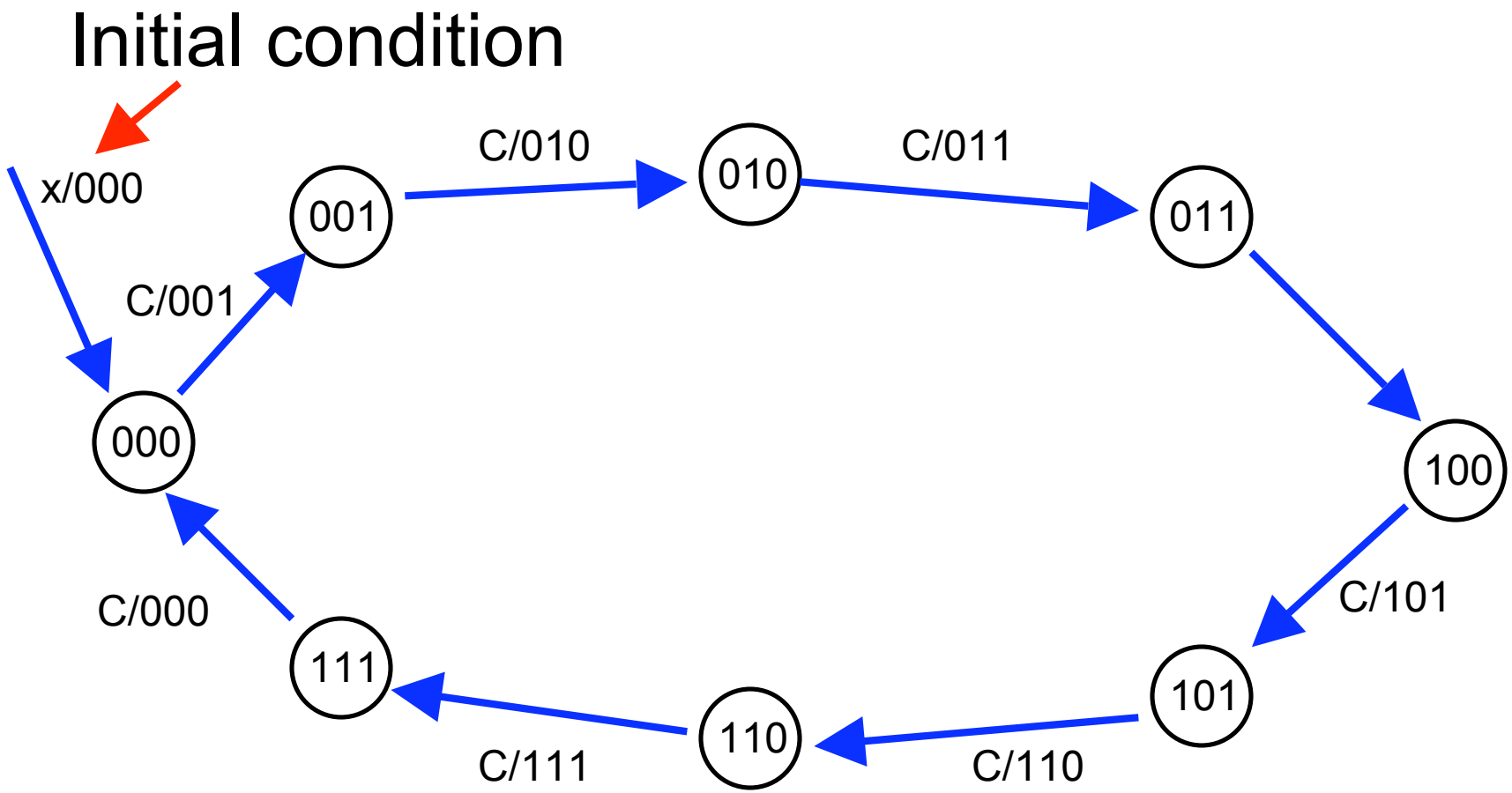


FSM Example: Synchronous Counter

The full transition set



FSM Example: Synchronous Counter



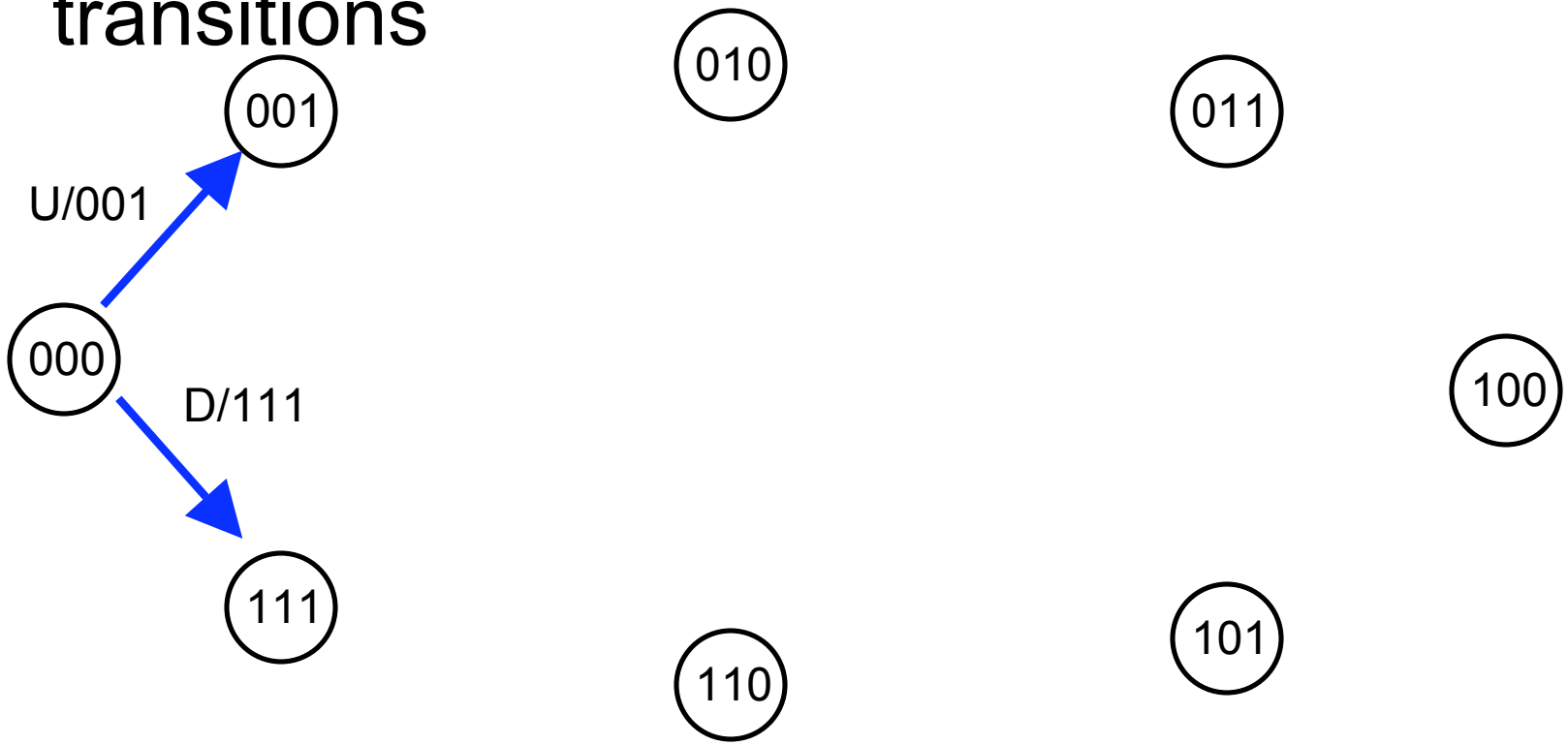
Example II: An Up/Down Counter

Suppose we have two events (instead of one): Up and down

- How does this change our state transition diagram?

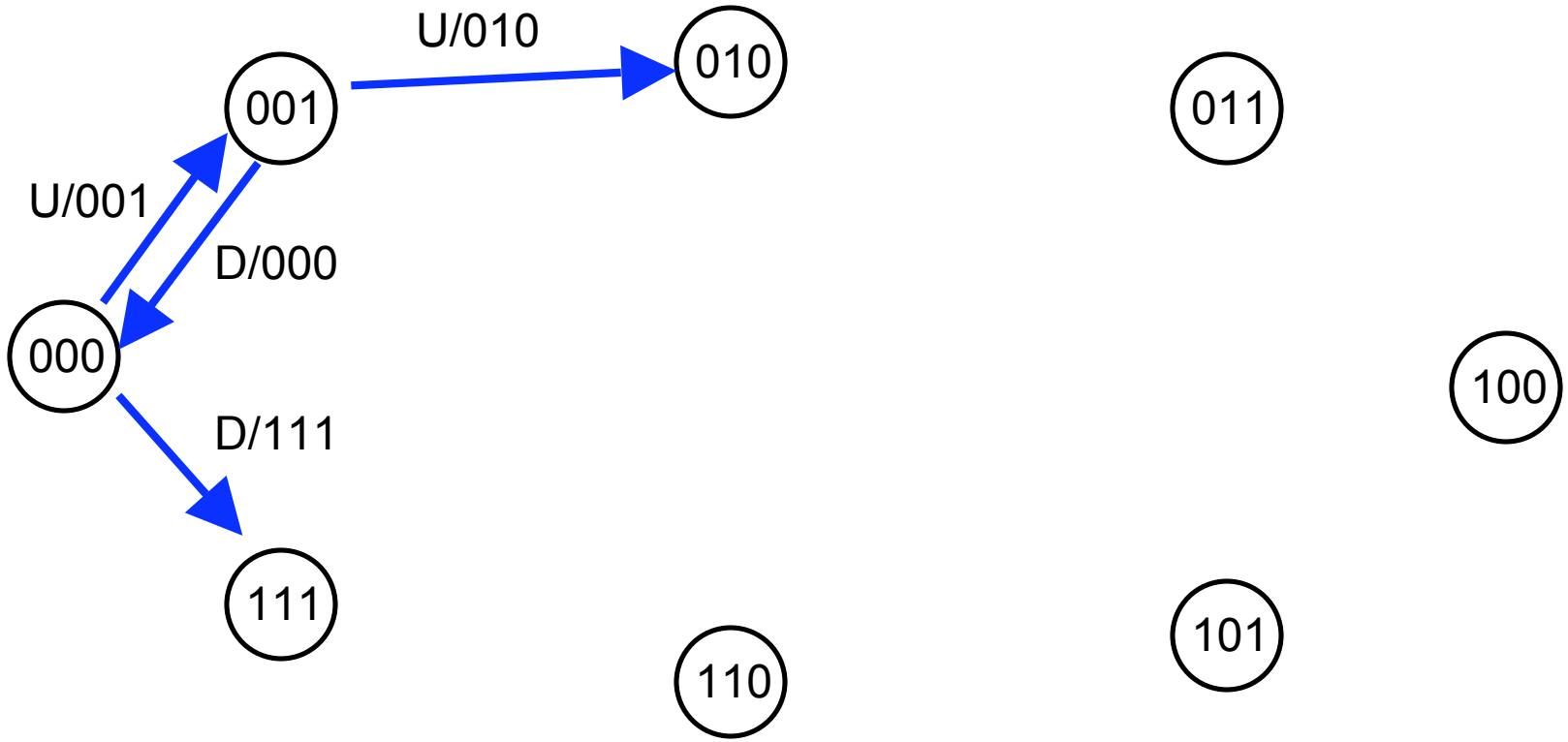
Example II: An Up/Down Counter

From state 000, there are now two possible transitions



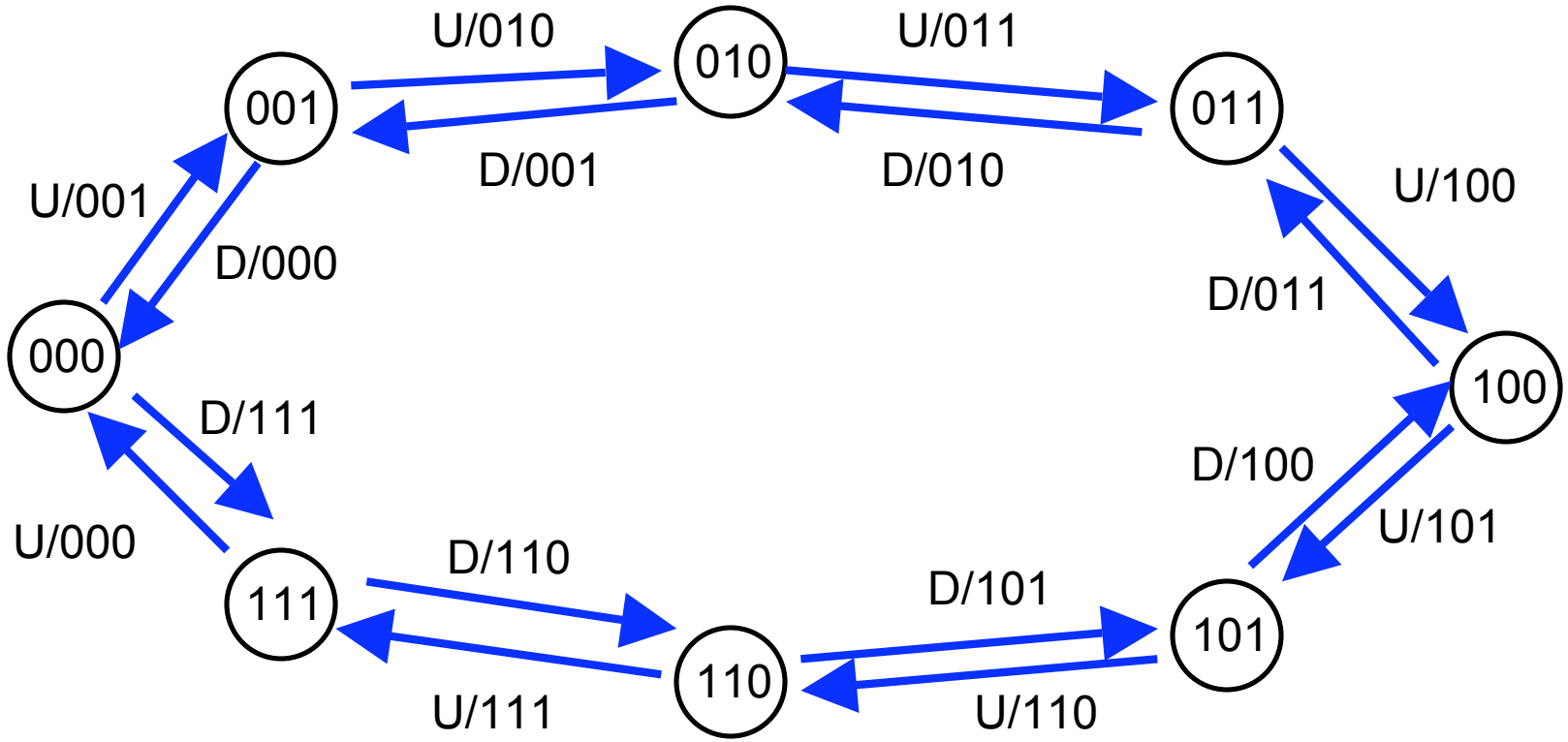
Example II: An Up/Down Counter

Likewise for state 001…



Example II: An Up/Down Counter

The full transition set



Last Time

Finite State Machines are defined by:

- A set of states
- A set of inputs (or events)
- A set of outputs
- A state transition function
- An output function

Today

- FSMs for control
- Implementing FSMs in code
- Lab 2

Administrivia

- HW 2 due today
- Lab 2 is now available
- HW 1 and quiz 1 are available for pick up

- Reading:
  - FSM web pages
  - Atmel Mega8 processor summary
  - ESP 9,10

FSMs and Control

How do we relate FSMs to Control?

- States are ?

FSMs and Control

How do we relate FSMs to Control?

- States are our memory of recent inputs

- Inputs are ?

FSMs and Control

How do we relate FSMs to Control?

- States are our memory of recent inputs
- Inputs are some processed representation of what the sensors are observing
- Outputs are ?

FSMs and Control

How do we relate FSMs to Control?

- States are our memory of recent inputs
- Inputs are some processed representation of what the sensors are observing
- Outputs are the control actions

FSMs: A Control Example

Suppose we have a vending machine:

- Accepts dimes and nickels
- Will dispense one of two things once $.20 has been entered: Jolt or Buzz Water
  - The “user” requests one of these by pressing a button
- Ignores select if < $.20 has been entered
- Immediately returns any coins above $.20

Vending Machine FSM

What are the states?

Vending Machine FSM

What are the states?

- $0
- $.05
- $.10
- $.15
- $.20

Vending Machine FSM

What are the inputs/events?

Vending Machine FSM

What are the inputs/events?

- Input nickel (N)
- Input dime (D)
- Select Jolt (J)
- Select Buzz Water (BW)

Vending Machine FSM

What are the outputs?

Vending Machine FSM

What are the outputs?

- Return nickel (RN)
- Return dime (RD)
- Dispense Jolt (DJ)
- Dispense Buzz Water (DBW)
- Nothing (Z)

Vending Machine Design

What is the initial state?

Vending Machine Design

What is the initial state?

- S = $0

Vending Machine Design

What can happen from S = $0?

Event	Next State	Output

Vending Machine Design

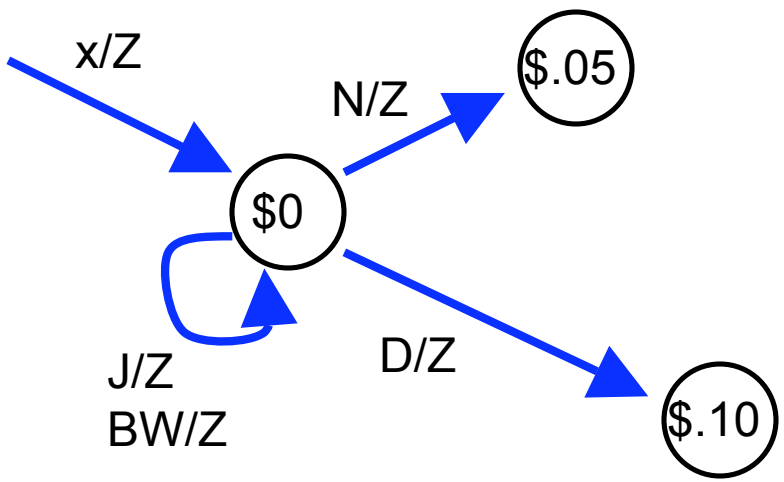
What can happen from S = $0?

What does this part of the diagram look like?

Event	Next State	Output
N	$.05	Z
D	$.10	Z
J	$0	Z
BW	$0	Z

Vending Machine Design

A piece of the state diagram:



Vending Machine Design

What can happen from S = $0.05?

Event	Next State	Output

Vending Machine Design

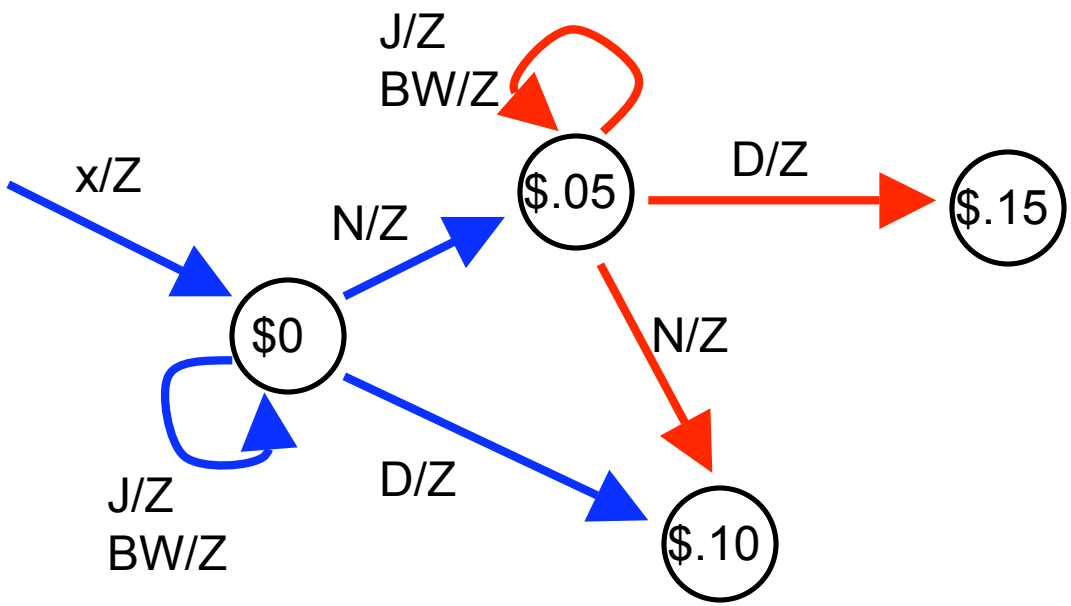
What can happen from S = $0.05?

What does the modified diagram look like?

Event	Next State	Output
N	$.10	Z
D	$.15	Z
J	$.05	Z
BW	$.05	Z

Vending Machine Design

A piece of the state diagram:



Vending Machine Design

What can happen from S = $0.10?

Event	Next State	Output

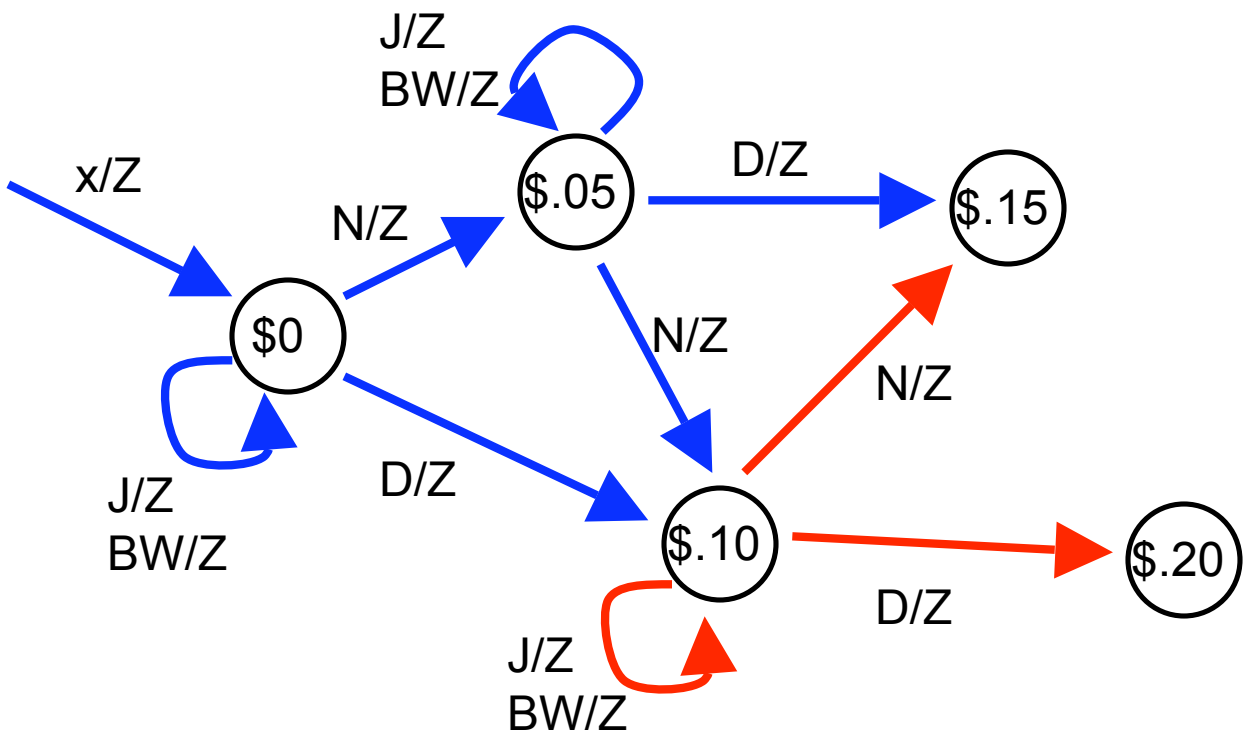
Vending Machine Design

What can happen from S = $0.10?

Event	Next State	Output
N	$.15	Z
D	$.20	Z
J	$.10	Z
BW	$.10	Z

Vending Machine Design

A piece of the state diagram:



Vending Machine Design

What can happen from S = $0.15?

Event	Next State	Output

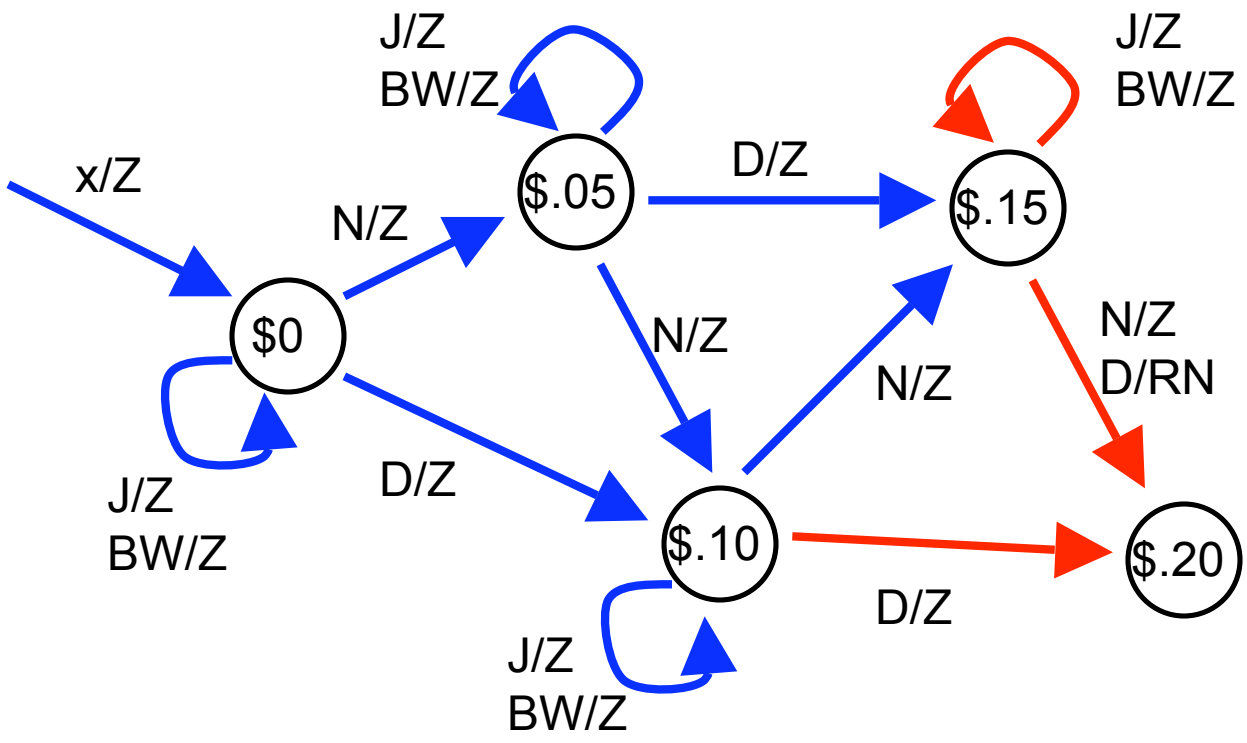
Vending Machine Design

What can happen from S = $0.15?

Event	Next State	Output
N	$.20	Z
D	$.20	RN
J	$.15	Z
BW	$.15	Z

Vending Machine Design

A piece of the state diagram:



Vending Machine Design

Finally: what can happen from S = $0.20?

Event	Next State	Output

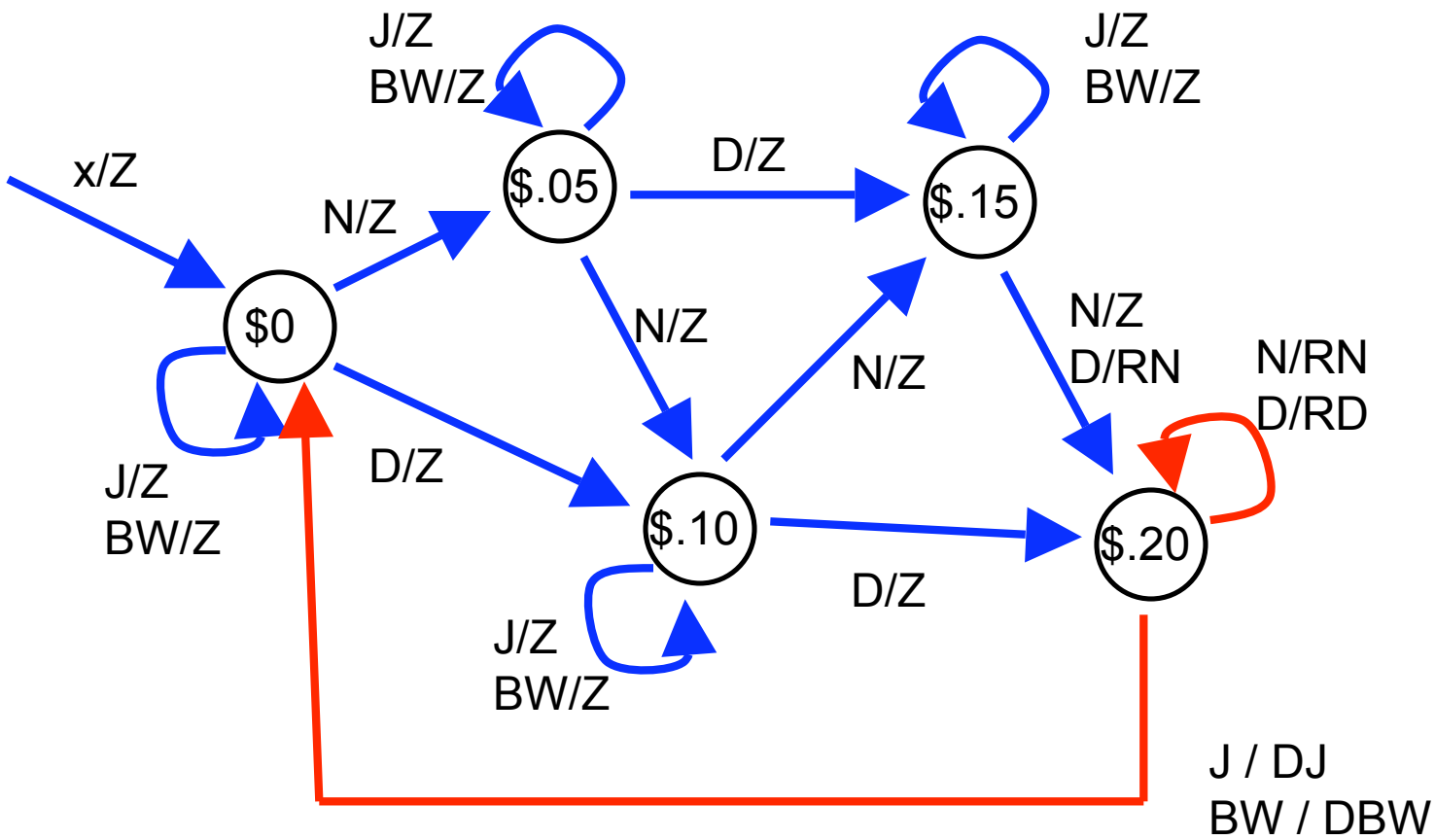
Vending Machine Design

Finally, what can happen from S = $0.20?

Event	Next State	Output
N	$.20	RN
D	$.20	RD
J	$0	DJ
BW	$0	DBW

Vending Machine Design

The complete state diagram:



Project 2: The Problem

Project 1:

- Implementation of a feedback control circuit (in digital logic) that orients and then moves toward a beacon

Project 2:

- Integrate this capability into a sequence of movements

Project 2: The Problem

Primary behavior of the robot:

- Phase 1:
  - Move toward beacon in front of the robot
  - Scan for another beacon on the left
  - When beacon is found, turn toward it
- Phase 2:
  - Move toward beacon in front
  - Scan for another beacon on the right
  - When beacon is found, turn toward it
- Repeat

Project 2: The Problem

An exception occurs if the robot loses sight of the forward beacon (no signal on either the left or the right sensor pair)

If in phase 1:

- Rotate turret to the right
- If a beacon is found, then turn the robot toward it and continue with phase 2
- Else stop moving

Project 2: The Problem

Exception handling

If in phase 2:

- Rotate turret to the left
- If a beacon is found, then turn the robot toward it and continue with phase 1
- Else stop moving

Project 2: Step 1

Design the FSM for this problem

- What are the states?
- What are the sensory signals?
- What are the inputs?
- What are the outputs?

Project 2: Step 2

Design the FSM for this problem

- What is the mapping from sensory signals to events?

Project 2: Step 3

Design the FSM for this problem

- What does the transition function look like?

Project 2: Step 4

Design the FSM for this problem

- What is the mapping from output to robot action?
- What must the robot do if no event occurs?

Project 2: Step 5

Implementation

- Write a C program that implements your FSM
- Burn this onto an Atmel mega8 processor
- Get it to work!

Next Time

- Another FSM control example
- Practicalities of Atmel processor programming
- Reading:
  - Lab 2 description
  - Atmel mega8 summary
  - The Atmel programming HOWTO

Implementing Finite State Machines

How would we implement an FSM with the logic components we have studied so far?

Implementing Finite State Machines

- Use flip-flops to represent the states:
  - Each **FSM state** is represented as a unique combination of flip-flop states
  - A clock signal is distributed to each flip-flop
- Use a set of binary input lines to represent the FSM inputs
  - We represent a **FSM input** with a unique combination of input bits

Implementing Finite State Machines

- The **transition function** is implemented as a logic function:
  - Combines the inputs with the current state
  - Produces the next state for each of the flip-flops
- **Output**:
  - Another logic function of the flip-flop states

Implementing FSMs in Code

Representing the current state:

- Introduce a “state” variable (an integer)
- Assign each state in your FSM a unique integer value
- For each distinct state, we will have a separate piece of code that implements the transition function

- (we can also use an enumerated data type instead of integers)

FSMs in C

```
int state = 0;   // Initial state
while(1) {
  <do some processing of the sensory inputs>
  switch(state) {
      case 0:
            <handle state 0>
            break;
      case 1:
            <handle state 1>
            break;
      case 2: …
  }
}
```

FSMs in C

```
int state = 0;   // Initial state
while(1) {
   <do some processing of the sensory inputs>
   switch(state) {
       case 0:
             <handle state 0>
             break;
       case 1:
             <handle state 1>
             break;
       case 2: …
   }
}
```

Variable declaration and initialization

FSMs in C

```
int state = 0;   // Initial state
while(1) {
   <do some processing of the sensory inputs>
   switch(state) {
       case 0:
             <handle state 0>
             break;
       case 1:
             <handle state 1>
             break;
       case 2: …
   }
}
```

A comment (use liberally)

FSMs in C

```
int state = 0;    // Initial state
while(1) {
   <do some processing of the sensory inputs>
   switch(state) {
       case 0:
             <handle state 0>
             break;
       case 1:
             <handle state 1>
             break;
       case 2: …
   }
}
```

Loop forever

FSMs in C

```
int state = 0;   // Initial state
while(1) {
  <do some processing of the sensory inputs>
   switch(state) {
       case 0:
             <handle state 0>
             break;
       case 1:
             <handle state 1>
             break;
       case 2: …
   }
}
```

"pseudo code": not really code, but indicates what is to be done

FSMs in C

```
int state = 0;   // Initial state
while(1) {
  <do some processing of the sensory inputs>
   switch(state) {
       case 0:
             <handle state 0>
             break;
       case 1:
             <handle state 1>
             break;
       case 2: …
   }
}
```

In this case: we will translate the current sensory inputs into a representation of an event (if one has happened)

FSMs in C

```
int state = 0;   // Initial state
while(1) {
  <do some processing of the sensory inputs>
  switch(state) {
     case 0:
            <handle state 0>
            break;
     case 1:
            <handle state 1>
            break;
     case 2: …
  }
}
```

Switch/case syntax allows us to cleanly perform many "if(x==y)" operations

FSMs in C

```
int state = 0;   // Initial state
while(1) {
   <do some processing of the sensory inputs>
   switch(state) {
      case 0:
            <handle state 0>
            break;
      case 1:
            <handle state 1>
            break;
      case 2: …
   }
}
```

If state==0, then execute the following code

FSMs in C

```
int state = 0;   // Initial state
while(1) {
   <do some processing of the sensory inputs>
   switch(state) {
       case 0:
            <handle state 0>
            break;
       case 1:
            <handle state 1>
            break;
       case 2: …
   }
}
```

This code can be as complex as necessary

FSMs in C

```
int state = 0;   // Initial state
while(1) {
   <do some processing of the sensory inputs>
   switch(state) {
       case 0:
             <handle state 0>
             break;
       case 1:
             <handle state 1>
             break;
       case 2: …
   }
}
```

break says to exit the switch (don't forget it or strange things will happen!)

FSMs in C

```
int state = 0;    // Initial state
while(1) {
   <do some processing of the sensory inputs>
   switch(state) {
       case 0:
             <handle state 0>
             break;
       case 1:
             <handle state 1>
             break;
       case 2: …
   }
}
```

If state==1, then …

FSMs in C

```
int state = 0;    // Initial state
while(1) {
   <do some processing of the sensory inputs>
   switch(state) {
       case 0:
             <handle state 0>
             break;
       case 1:
             <handle state 1>
             break;
       case 2: …
   }
}
```

End of the **switch** block

FSMs in C

```
int state = 0;    // Initial state
while(1) {
   <do some processing of the sensory inputs>
   switch(state) {
       case 0:
             <handle state 0>
             break;
       case 1:
             <handle state 1>
             break;
       case 2: …
   }
}
```

End of the while block

Last Time

- Finite state machines
- FSM implementations in C

Today

- More about FSMs in C
- Programming the Atmel Mega8
- Computer Architecture

Administrivia

- Lab 2:
  - Available on the website
  - Get started this week!
- Lab in general:
  - Make sure you clean up your workspace before you leave
  - Put used batteries back on the charger – not in the “charged” battery boxes

Administrivia

- Midterm:
  - 3 weeks from today
  - Will be allowed 1 double-sided page of notes
  - More details in 2 weeks
- Class Thursday:
  - Discussion of homework 2 (and any remaining questions about homework 1)
  - A couple more FSM examples (last opportunity before midterm)

FSMs in C

```
int state = 0;   // Initial state
while(1) {
  <do some processing of the sensory inputs>
  switch(state) {
      case 0:
            <handle state 0>
            break;
      case 1:
            <handle state 1>
            break;
      case 2: …
  }
}
```

FSMs in C (some other possibilities)

```
int state = 0;   // Initial state
while(1) {
  <do some processing of the sensory inputs>
  switch(state) {
      case 0:
            <handle state 0>
            break;
         :
      default:
            <handle default case>
            break;
  }
  <do some low-level control>
}
```

FSMs in C (some other possibilities)

```
int state = 0;    // Initial state
while(1) {
   <do some processing of the sensory inputs>
   switch(state) {
       case 0:
             <handle state 0>
             break;
          :
       default:
             <handle default case>
             break;
   }
   <do some low-level control>
}
```

Matches any state (if we reach this point)

FSMs in C (some other possibilities)

```
int state = 0;    // Initial state
while(1) {
   <do some processing of the sensory inputs>
   switch(state) {
       case 0:
             <handle state 0>
             break;
          :
       default:
             <handle default case>
             break;
   }
  <do some low-level control>
}
```

(possibly) alter some control outputs (e.g., steering direction)

FSMs in C: Processing for Individual States

```
case STATE_10cents:
   // $.10 has already been deposited
   switch(event) {
        case EVENT_NICKEL:   // Nickel
                state = STATE_15cents;  // Transition to $.15
                break;
        case EVENT_DIME:   // Dime
                state = STATE_20cents;  // Transition to $.2
                break;
        case EVENT_JOLT:   // Select Jolt
        case EVENT_BUZZ:   // Select Buzzwater
                display_NOT_ENOUGH();
                break;

        case EVENT_NONE:   // No event
                break;           // Do nothing

   };
   break;
```

FSMs in C: Processing for Individual States

```
case 2:
   // $.10 has already been deposited
   switch(event) {
        case 0:    // Nickel
                state = 3;   // Transition to $.15
                break;
        case 1:    // Dime
                state = 4;   // Transition to $.2
                break;
        case 2:    // Select Jolt
        case 3:    // Select Buzzwater
        case 4:    // No event
                break;        // Do nothing

   };
   break;
```

Another integer

FSMs in C: Processing for Individual States

```
case 2:
   // $.10 has already been deposited
   switch(event) {
       case 0:    // Nickel
              state = 3;  // Transition to $.15
              break;
       case 1:    // Dime
              state = 4;  // Transition to $.2
              break;
       case 2:    // Select Jolt
       case 3:    // Select Buzzwater
       case 4:    // No event
              break;        // Do nothing

   };
   break;
```

A nickel has been received

FSMs in C: Processing for Individual States

```
case 2:
   // $.10 has already been deposited
   switch(event) {
       case 0:   // Nickel
                 state = 3;  // Transition to $.15
                 break;
       case 1:   // Dime
                 state = 4;  // Transition to $.2
                 break;
       case 2:   // Select Jolt
       case 3:   // Select Buzzwater
       case 4:   // No event
                 break;        // Do nothing

   };
   break;
```

Change state for next iteration of the while() loop

FSMs in C: Processing for Individual States

```
case 2:
   // $.10 has already been deposited
   switch(event) {
       case 0:   // Nickel
                 state = 3;  // Transition to $.15
                 break;
       case 1:   // Dime
                 state = 4;  // Transition to $.2
                 break;
       case 2:   // Select Jolt
       case 3:   // Select Buzzwater
       case 4:   // No event
                 break;        // Do nothing

   };
   break;
```

If any of these match, then execute the following code (which does nothing in this example)

A Note on “Style” in C

- The numbers that we assigned to the different states are arbitrary (and at first glance, hard to interpret)
- Instead, we can define constant strings that have some meaning

- Replace: 0, 1, 2, 3, 4, 5
- With: STATE_00, STATE_05, STATE_10, STATE_15, STATE_20

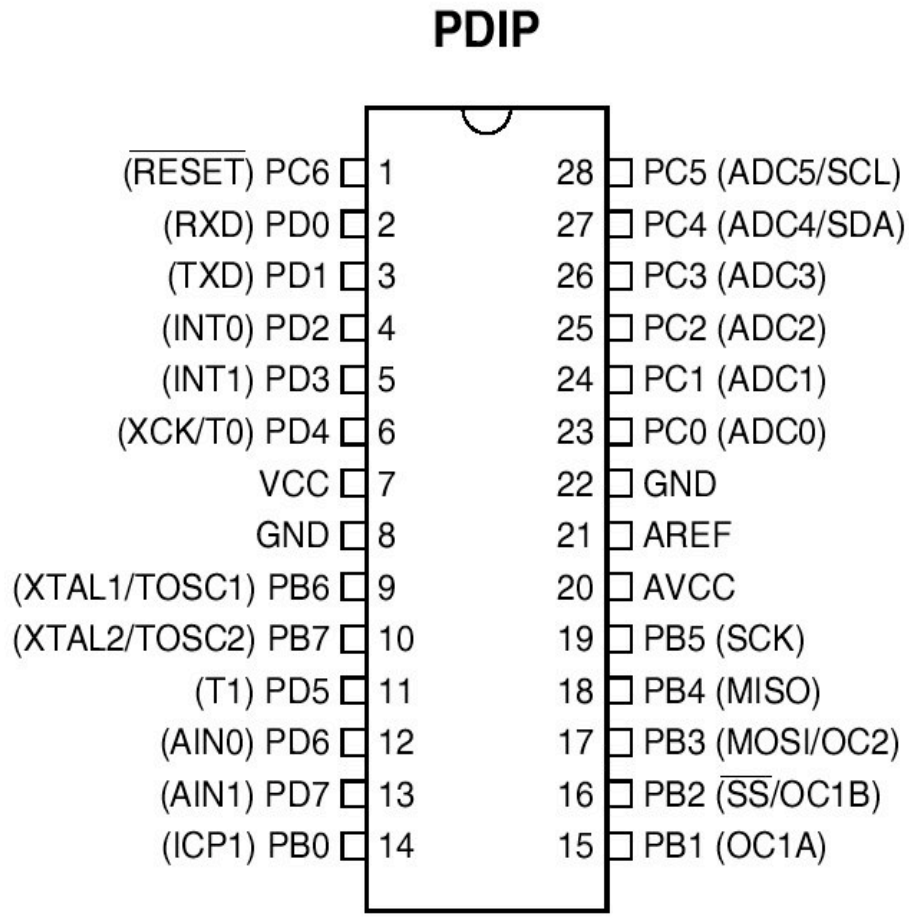
A Note on “Style” in C

In C, this is done by adding some definitions to the beginning of your program (either in the .c file or the .h file):

```
#define STATE_00   0
#define STATE_05   1
#define STATE_10   2
#define STATE_15   3
#define STATE_20   4
```

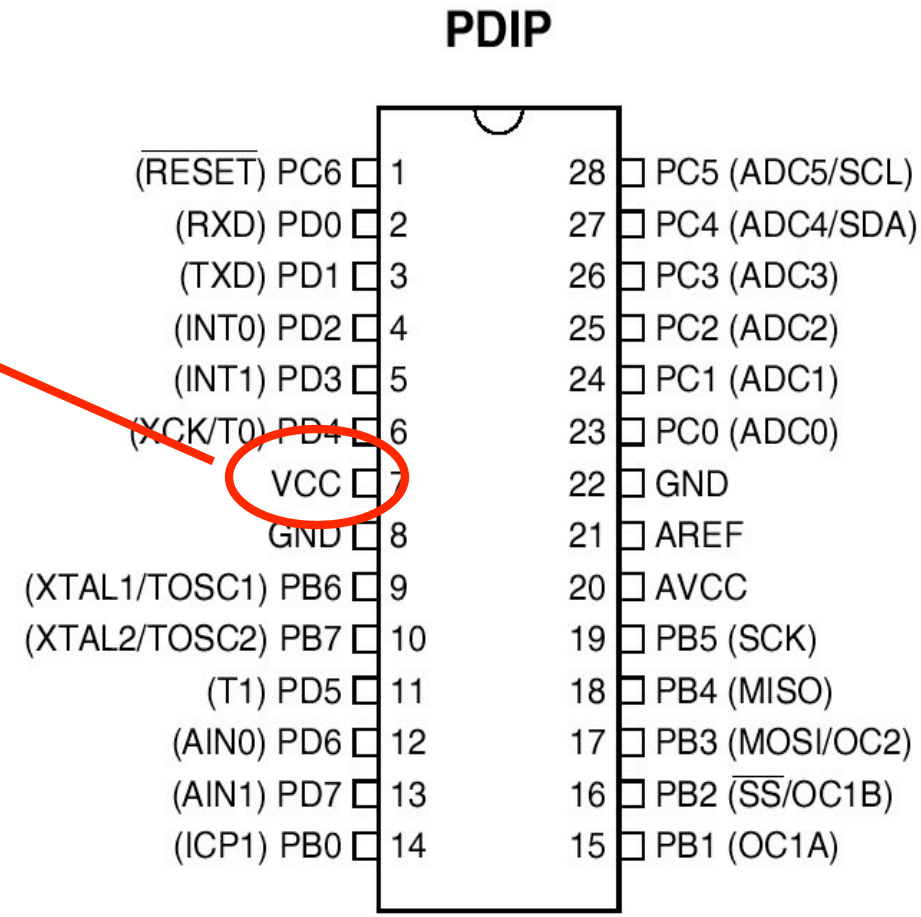
Atmel Mega8 Basics

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



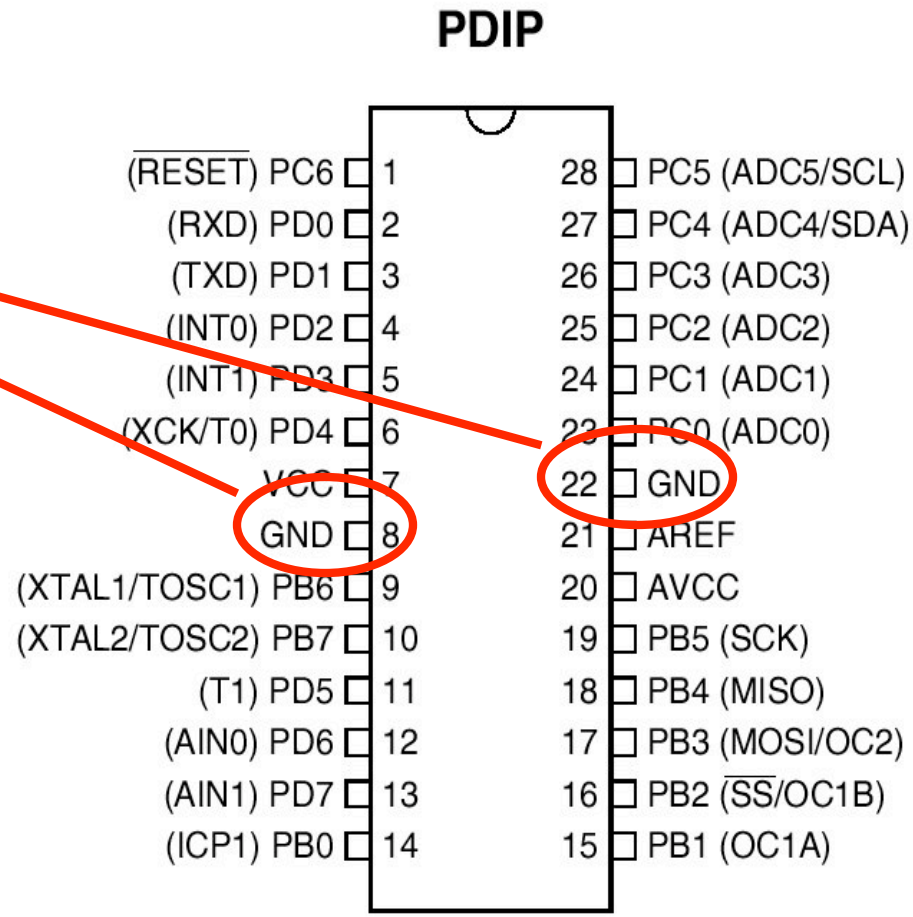
Atmel Mega8 Basics

Power (we will use +5V)



Atmel Mega8 Basics

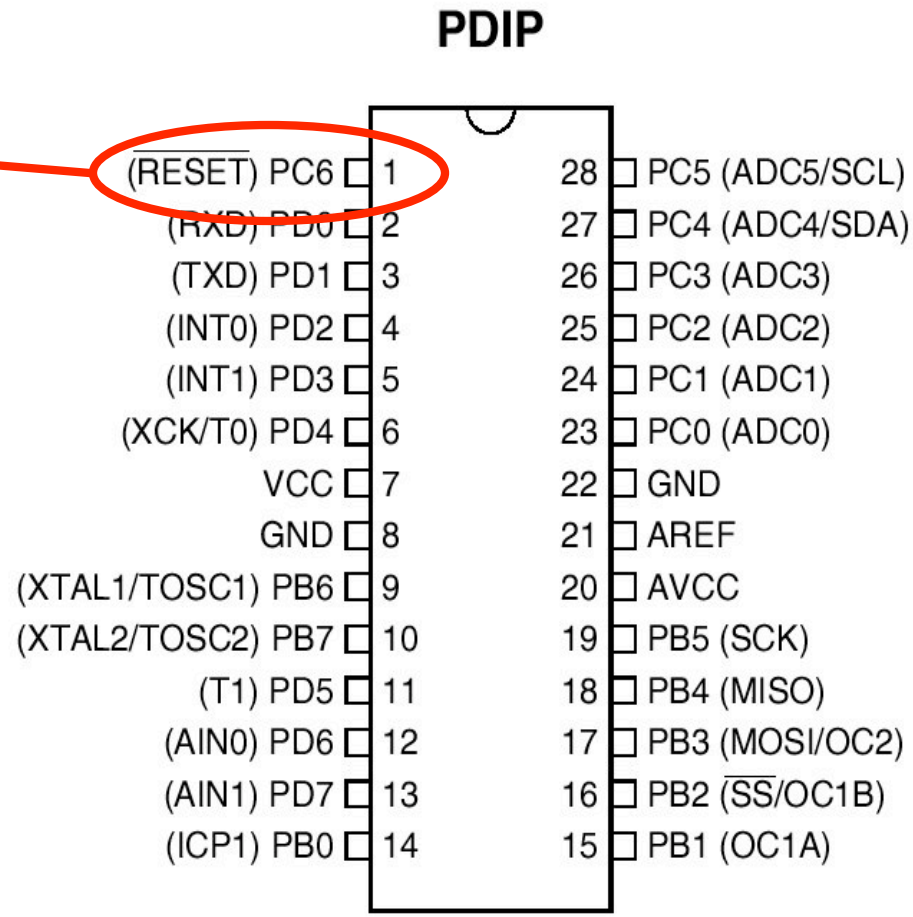
Ground



Atmel Mega8 Basics

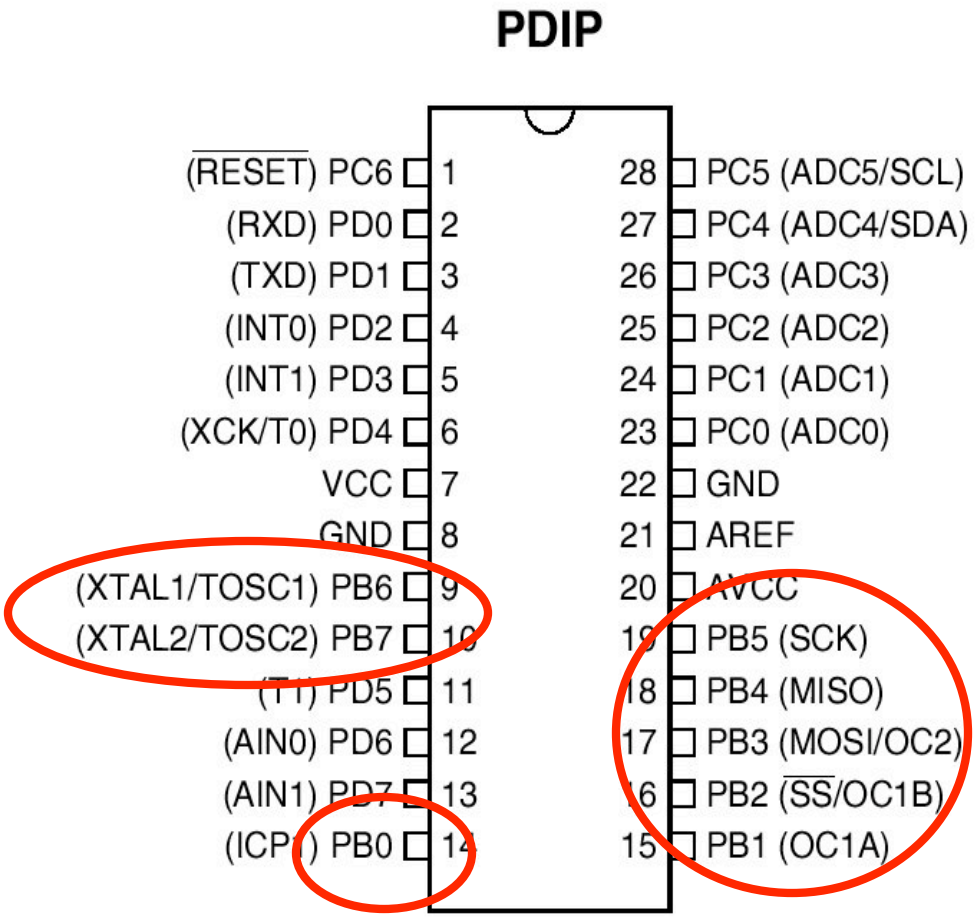
Reset

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



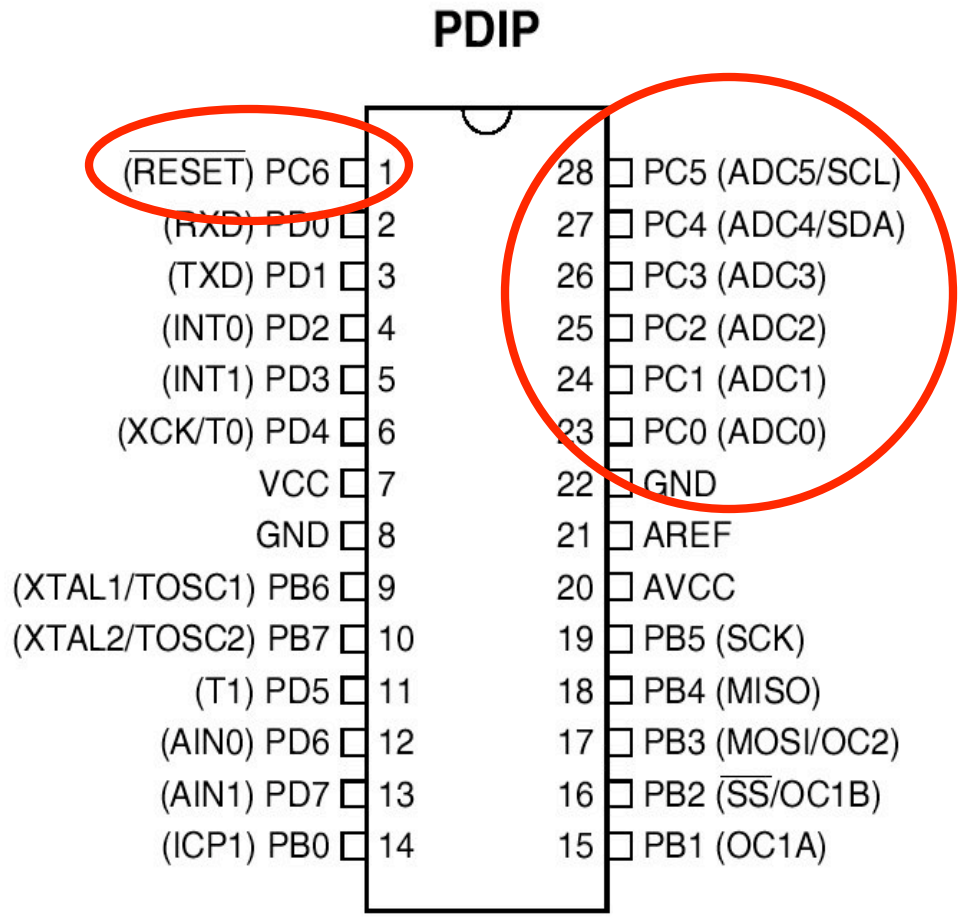
Atmel Mega8 Basics

PORT B



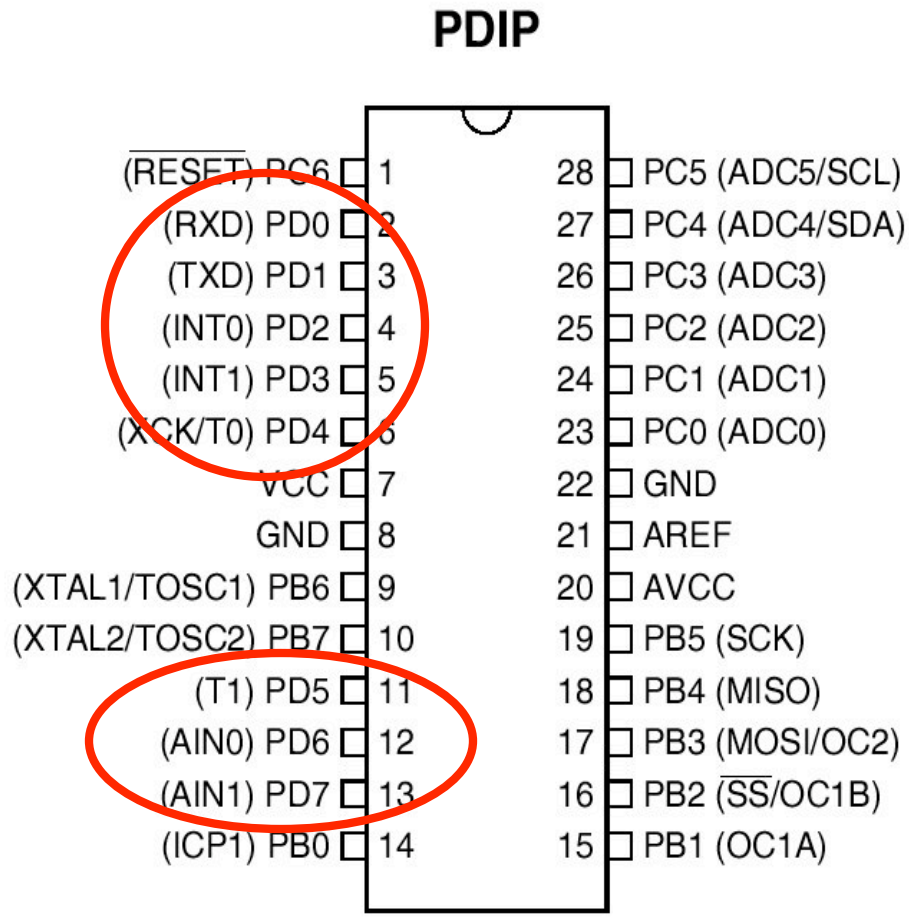
Atmel Mega8 Basics

PORT C

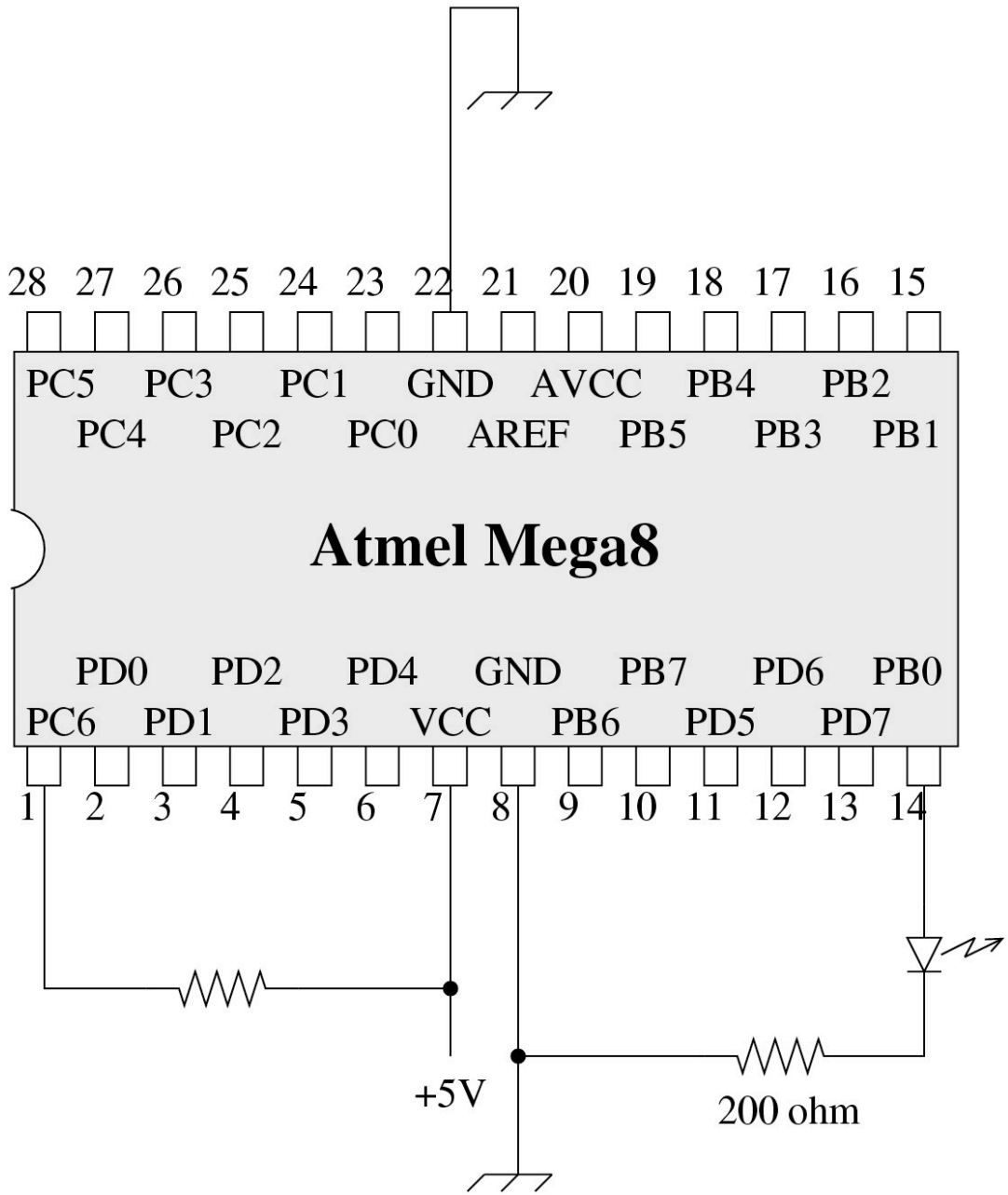


Atmel Mega8 Basics

PORT D



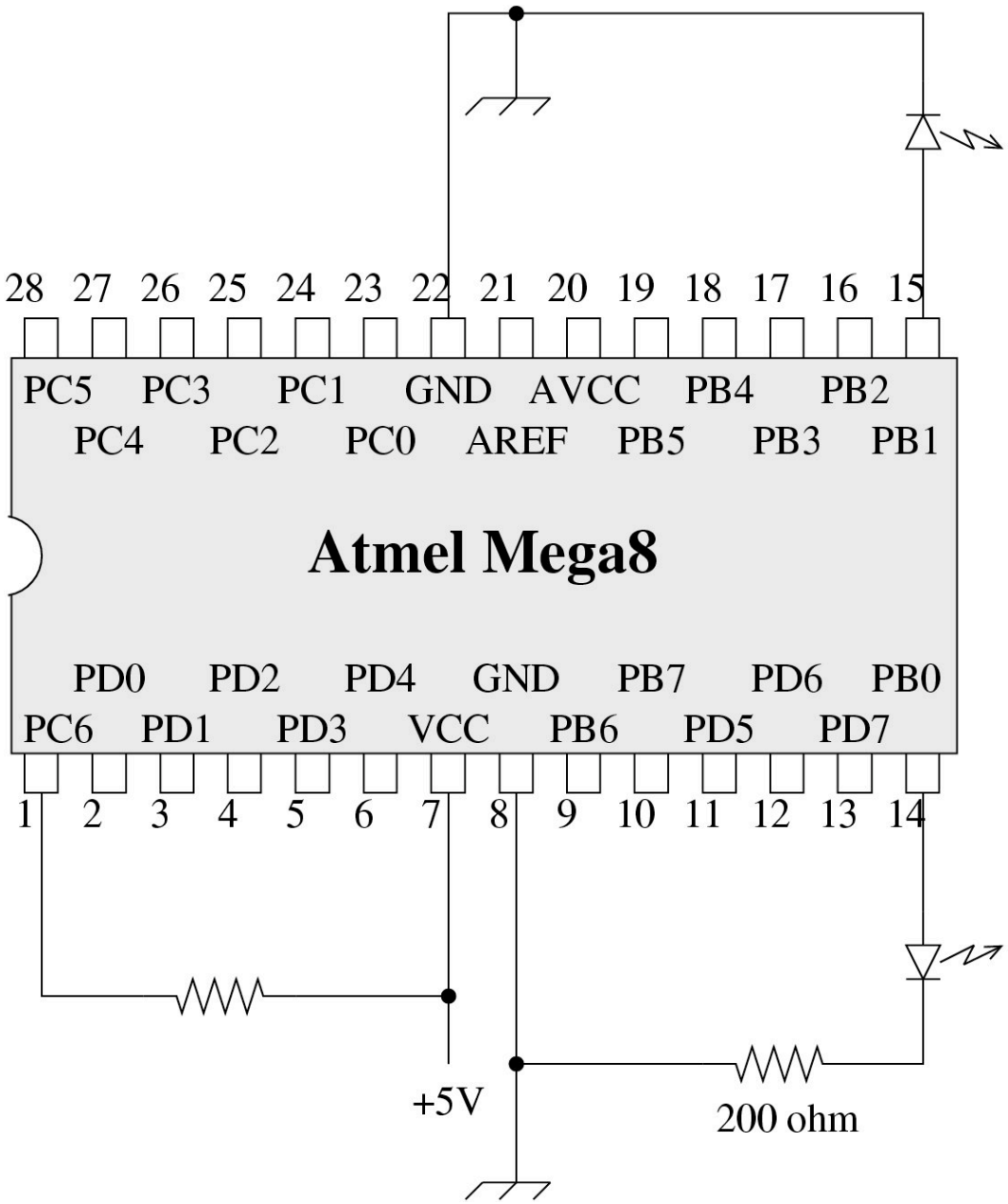
A First Circuit



A First Program

Flash the LEDs at a regular interval

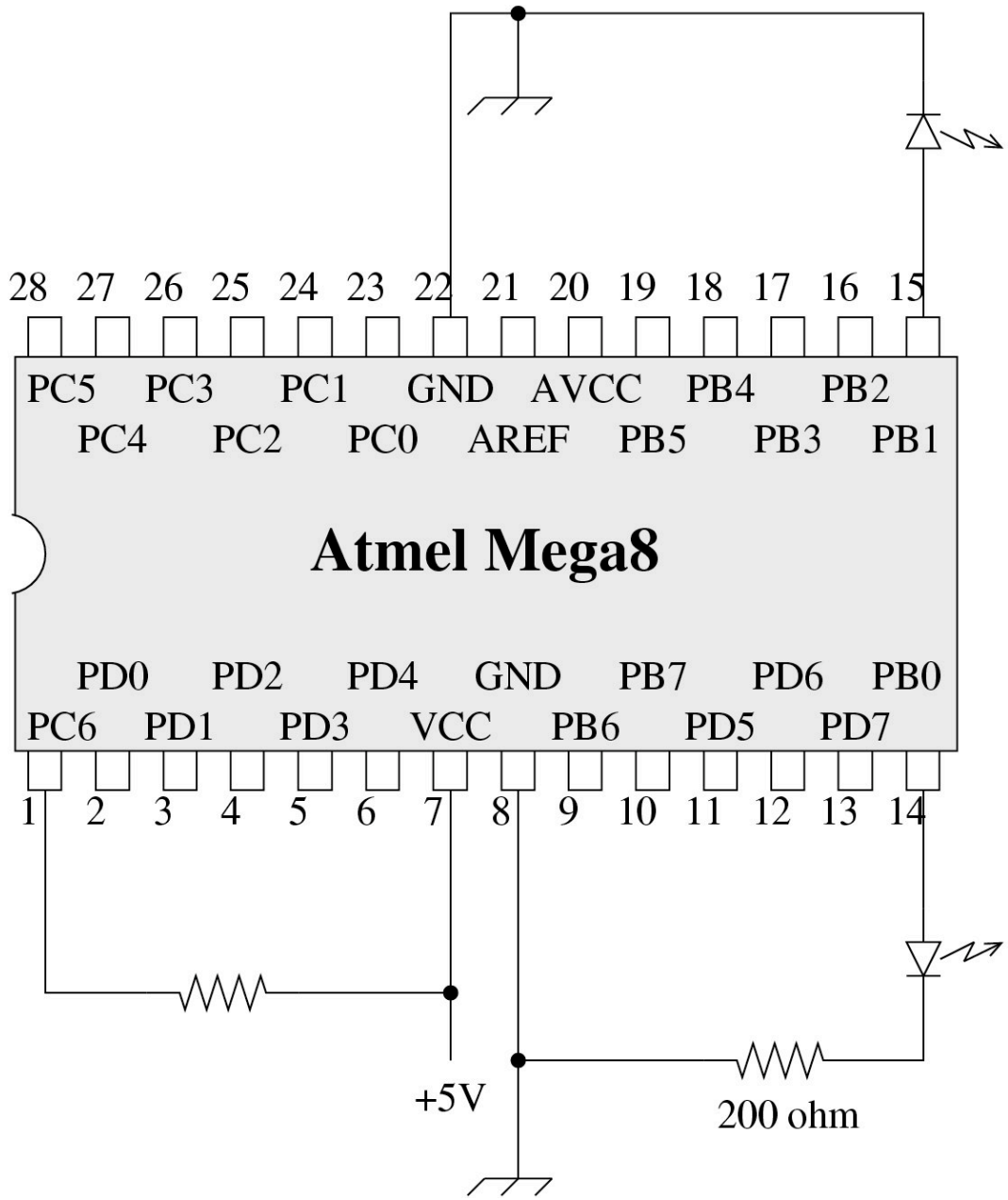
- How do we do this?



A First Program

How do we flash the LEDs at a regular interval?

- We toggle the state of PB0



A First 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);
   }
}
```

A First 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);
  }
}
```

A predefined “variable” that controls whether the port B pins are digital inputs or outputs (more on this later)

A First 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);
   }
}
```

Loop forever

A First 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);
   }
}
```

Another predefined variable: the value being written to port B

A First 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);
   }
}
```

Change the value being written to port B

A First 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);
   }
}
```

Bit-wise XOR operator

A First 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);
   }
}
```

Program pauses for 500 msec. This function is defined elsewhere in this sample program.

A First 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);
   }
}
```

What does this program do?

A First 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 2 Hz
on PB0: 1 Hz

Last Time

- FSM: abstract and in C
- Atmel basics

Today

- Atmel essentials:
  - Digital I/O
  - Care and feeding
  - Basic circuits
  - Cross-compiler use
  - Downloading cod
- FSM examples
- HW 2

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 in Practice

- Suppose you have connected your 3 robot control lines to B3, B4, and B5.
- Suppose also that you have connected your 2 turret control lines to B6 and B7

- Our robot control lines are specified as numbers 0 … 6
- How do we change the state of B3…B5 without changing the turret command?

Bit Masking

```
main() {
   DDRB = 0xF8;   // Set pins B3, B4, B5, B6, B7 as outputs

        :
        :

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

   val = command_to_robot;

   PORTB = (PORTB & 0xC7)          // Set the current B3-B5 to 0s
       | ((val & 0x7))<<3);        // OR with new values
   (shifted
                                   //  to fit within B3-B5
}
```

Bit Masking

```
main() {
  DDRB = 0xF8;    // Set pins B3, B4, B5, B6, B7 as outputs

       :
       :

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

   val = command_to_robot;

   PORTB = (PORTB & 0xC7)          // Set the current B3-B5 to 0s
       | ((val & 0x7))<<3);        // OR with new values
   (shifted
                                   //  to fit within B3-B5)
}
```

B3-B7 are outputs; all others are still inputs (could be different depending on how other pins are used)

Bit Masking

```
main() {
   DDRB = 0xF8;    // Set pins B3, B4, B5, B6, B7 as outputs

        :
        :

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

   val = foobar;

   PORTB = (PORTB & 0xC7)          // Set the current B3-B5 to 0s
       | ((val & 0x7))<<3);        // OR with new values
   (shifted
                                   //  to fit within B3-B5
}
```

"Mask out" the current values of pins B3-B5 (leave everything else intact)

Bit Masking

```
main() {
   DDRB = 0xF8;    // Set pins B3, B4, B5, B6, B7 as outputs

        :
        :

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

   val = foobar;

   PORTB = (PORTB & 0xC7)          // Set the current B3-B5 to 0s
       | ((val & 0x7))<<3);        // OR with new values
   (shifted

                                   //  to fit within B3-B5
}
```

Substitute an arbitrary value into these bits

Bit Masking

```
main() {
   DDRB = 0xF8;    // Set pins B3, B4, B5, B6, B7 as outputs

        :
        :

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

   val = foobar;

   PORTB = (PORTB & 0xC7)          // Set the current B3-B5 to 0s
        | ((val & 0x7))<<3);       // OR with new values
   (shifted

                                   //  to fit within B3-B5
}
```

And use the result to change the output state of port B

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;
}
```

B6 and B7 are configured as inputs

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;
}
```

Read the value from the port

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;
}
```

"Mask out" all bits except B6 and B7

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;
}
```

Right shift the result by 6 bits – so the value of B6 and B7 are now in bits 0 and 1 of “outval”

Port-related “Variables”

The set of C-accessible variables (actually called registers) for controlling digital I/O:

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

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

Physical Interface for Programming

AVR ISP

Physical Interface for Programming

AVR ISP

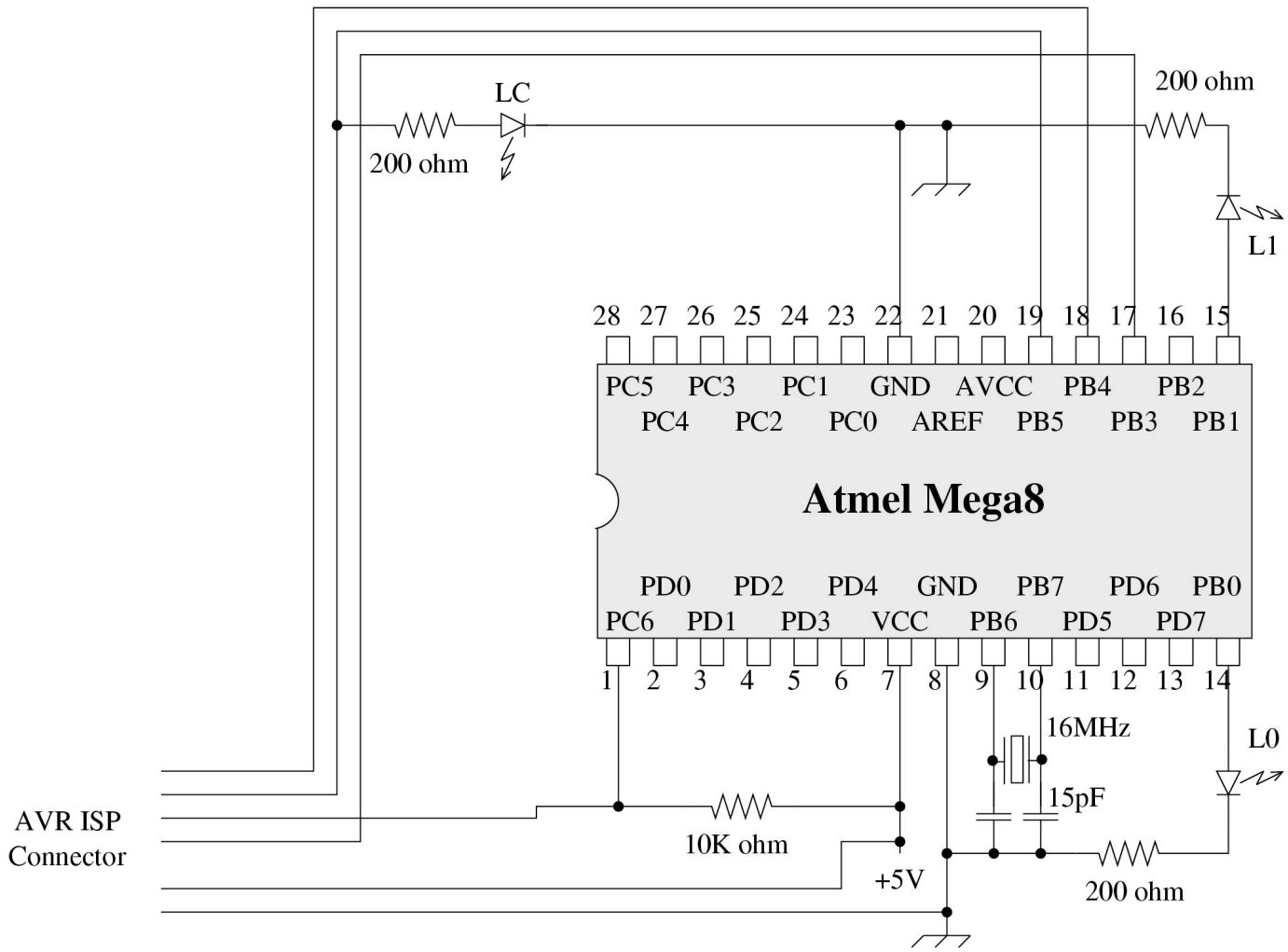
Serial
connection to
your computer

Physical Interface for Programming

AVR ISP

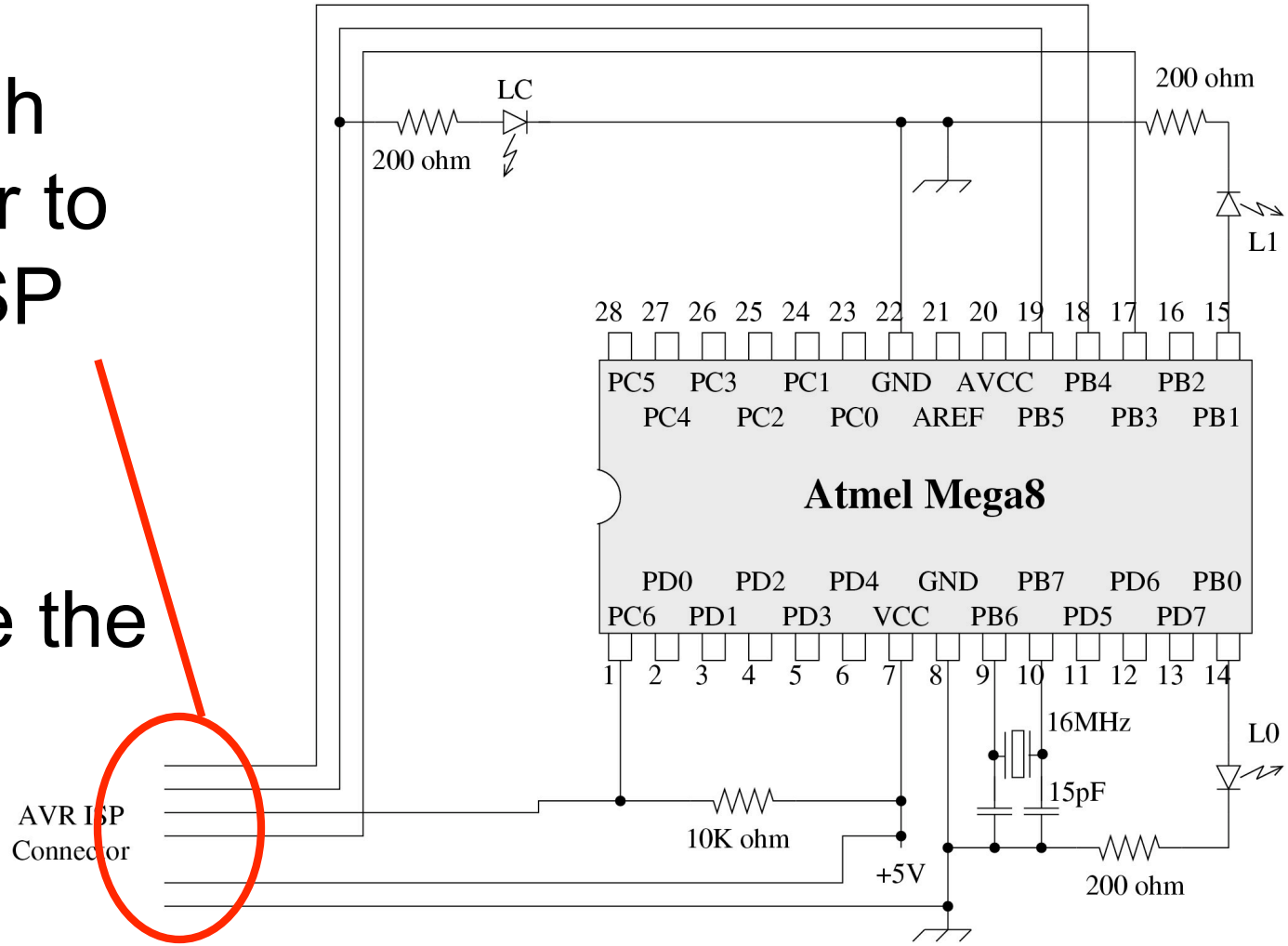
Header connection will connect to your circuit (through an adapter)

A More Complicated Circuit



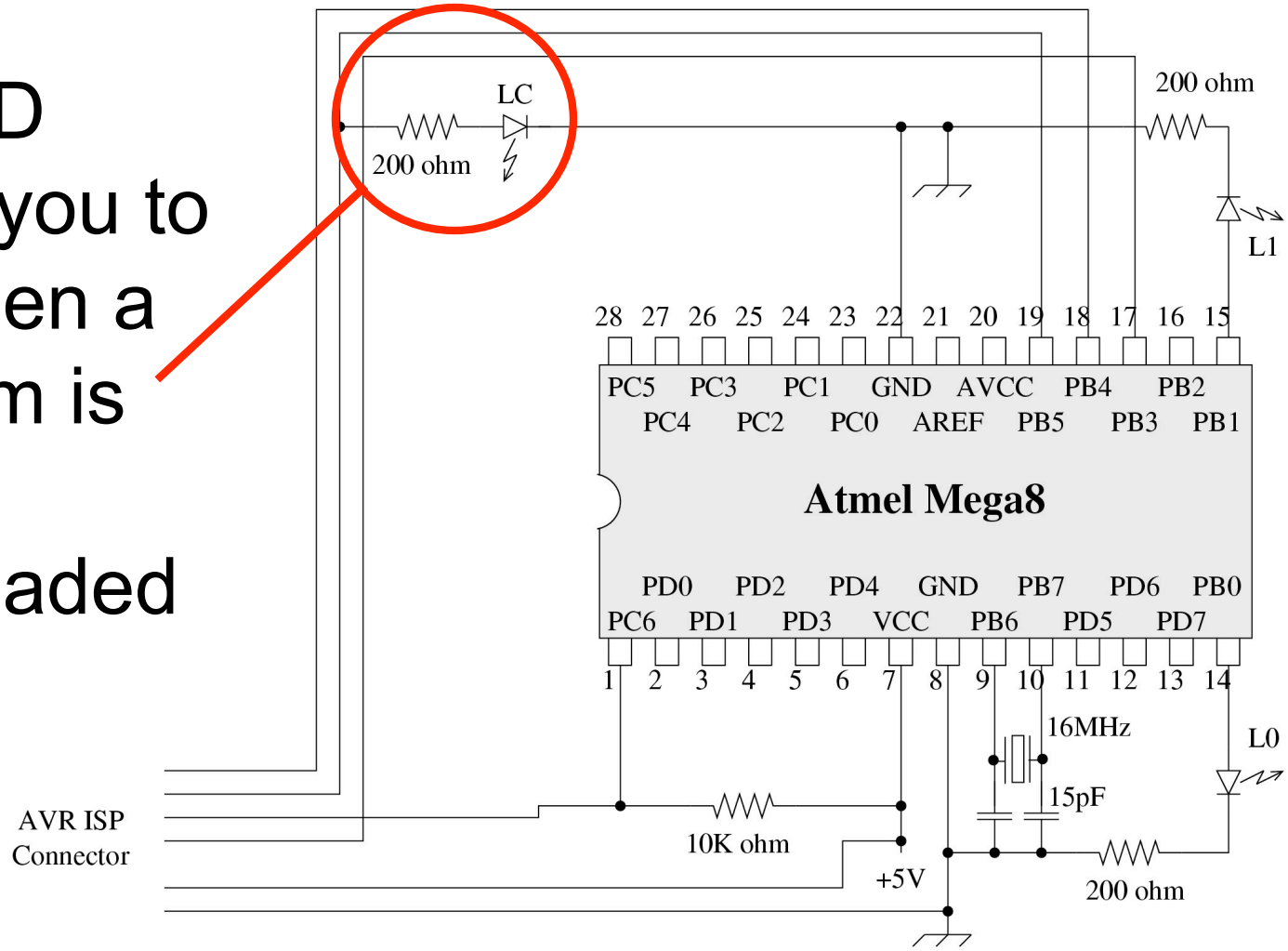
A More Complicated Circuit

- Through adapter to AVR ISP
- Do not reverse the pins!



A More Complicated Circuit

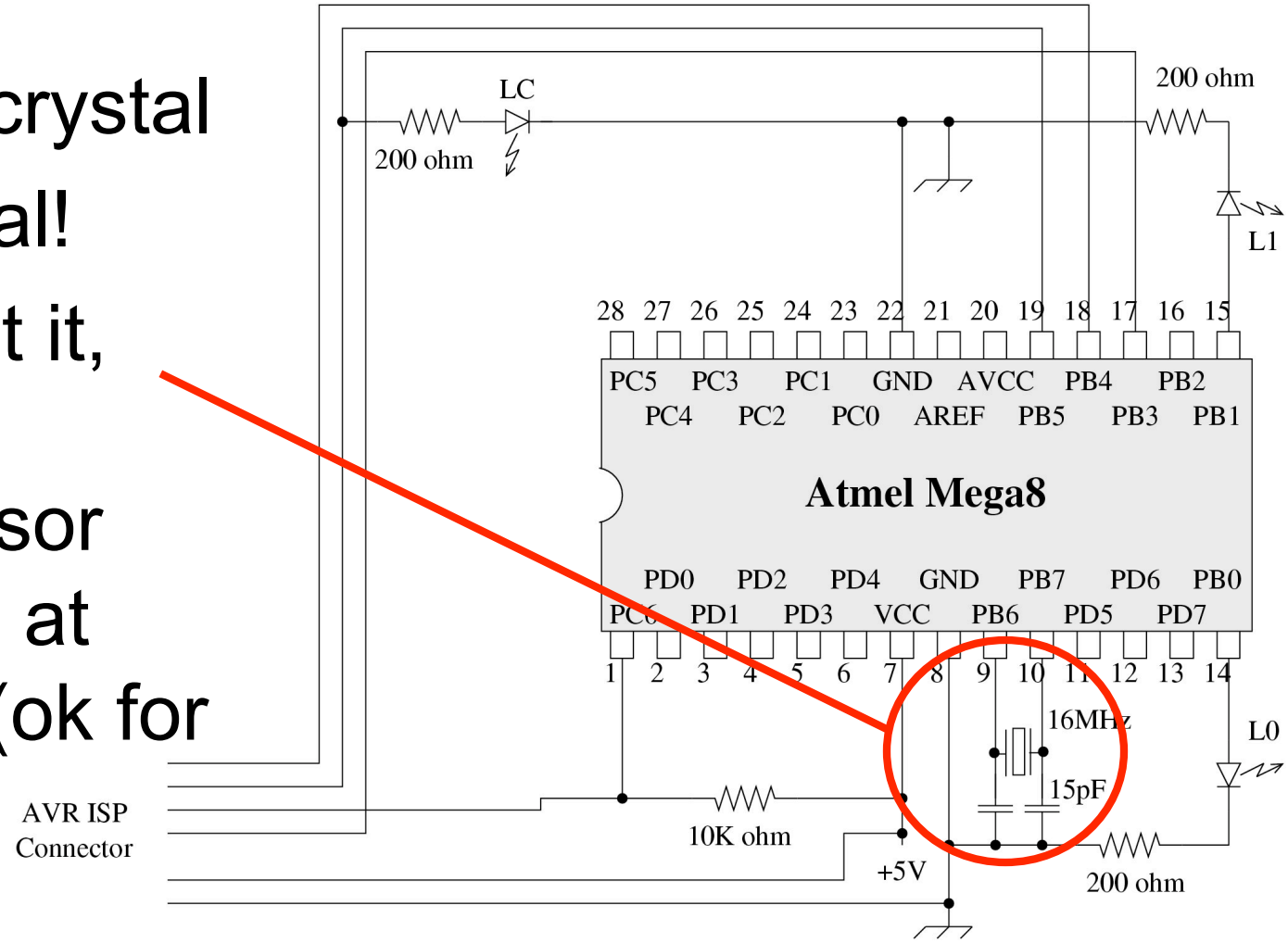
Extra LED allows you to see when a program is being downloaded



A More Complicated Circuit

16 MHz crystal

- Optional!
- Without it, your processor will run at 1MHz (ok for now)



Compiling and Downloading Code

- Create a “makefile” and a C source file in some personal directory (start with copies from the “Atmel HOWTO” part of the web page
- Change the TARGET line in makefile to match the name of your source file
- Change your source file

Compiling and Downloading Code

- From the windoze menu: select “Start” and then “run”
- Type “cmd”. This will bring up a “shell” interface (a command line)
- “cd” (change directory) to your directory
- Type “make”. Deal with any errors or warnings (both are there for a reason)
- Type “make program”

Compiling and Downloading Code

- Once the chip is programmed, the AVR ISP will automatically reset the processor; starting your program

Hints

- Use LEDs to show status information (e.g., the state of your FSM)
- Have one LED blink in some unique way at the beginning of your program
- Go slow:
  - Implement and test incrementally
  - Insert plenty of pauses into your code (use delay_ms())

Final Notes

- You will need some additional circuitry to program your processor
- There are a few more details in the code

-> See the examples posted on the net

Next Time

- Procedure for programming and downloading to the Mega8
- Inside the Mega8