Administrivia

- Homework 1 is posted & is due in 1 week
- Additional sequential logic readings have been posted on the schedule
Last Time

- D-type Flip-flops
- Sequential logic circuits
  - Shift register
  - Frequency divider
Today

- One more sequential logic example
- Microprocessor components (just enough to be dangerous)
Components of a Microprocessor
Components of a Microprocessor

• Memory:
  – Storage of data
  – Storage of a program
  – Either can be temporary or “permanent” storage

• Registers: small, fast memories
  – General purpose: store arbitrary data
  – Special purpose: used to control the processor
Components of a Microprocessor

• Instruction decoder:
  – Translates current program instruction into a set of control signals

• Arithmetic logical unit:
  – Performs both arithmetic and logical operations on data

• Input/output control modules
Components of a Microprocessor

- Many of these components must exchange data with one-another
- It is common to use a ‘bus’ for this exchange
Buses

• In the simplest form, a bus is a single wire
• Many different components can be attached to the bus
• Any component can take input from the bus or place information on the bus
Buses

• At most one component may write to the bus at any one time
• Which component is allowed to write is usually determined by the code that is currently executing
Collections of Bits

- 8 bits: a “byte”
- 4 bits: a “nybble”
- “words”: can be 8, 16, or 32 bits (depending on the processor)
Collections of Bits

• A data bus typically captures a set of bits simultaneously
• Need one wire for each of these bits
• In the Atmel Mega8: the data bus is 8-bits “wide”
• In your home machines: 32 or 64 bits
Memory

What are the essential components of a memory?
A Memory Abstraction

- We think of memory as an array of elements – each with its own address
- Each element contains a value
  - It is most common for the values to be 8-bits wide (so a byte)

\[
\begin{array}{cccc}
0x32 & 0xF1 & 0x11 & 0x67 \\
0 & 1 & 2 & 3 \\
\end{array}
\ldots
\]
A Memory Abstraction

- We think of memory as an array of elements – each with its own address
- Each element contains a value
  - It is most common for the values to be 8-bits wide (so a byte)

\[
\begin{array}{cccc}
0x32 & 0xF1 & 0x11 & 0x67 \\
0    & 1    & 2    & 3 \\
\end{array}
\]

\[2^{M-1}\]
Memory Operations

Read

\[ \text{foo}(A+5); \]

reads the value from the memory location referenced by the variable ‘A’ and adds the value to 5. The result is passed to a function called \( \text{foo}() \);
Memory Operations

Write

\[ A = 5; \]

writes the value 5 into the memory location referenced by ‘A’
Types of Memory

Random Access Memory (RAM)
- Computer can change state of this memory at any time
- Once power is lost, we lose the contents of the memory

- This will be our data storage on our microcontrollers
Types of Memory

Read Only Memory (ROM)

• Computer cannot arbitrarily change state of this memory

• When power is lost, the contents are maintained
Types of Memory

Erasable/Programmable ROM (EPROM)

• State can be changed under very specific conditions (usually not when connected to a computer)

• Our microcontrollers have an Electrically Erasable/Programmable ROM (EEPROM) for program storage
Example: A Read/Write Memory Module

Inputs:
• 2 Address bits: A0 and A1
• 1 “chip select” (CS) bit
• 1 read/write bit (1 = read; 0 = write)
• 1 clock signal (CLK)

Input or Output:
• Data bit (connected to the “data bus”)
A Read/Write Memory Module

- Address Bus
- Data Bus
- CS
- R/W
- CLK

Andrew H. Fagg: Embedded Real-Time Systems: Microcontrollers
A Read/Write Memory Module

Inputs or outputs

Our example:
- $M=2$
- $N=1$
Implementing A Read/Write Memory Module

With 2 address bits, how many memory elements can we address?

How could we implement each memory element?
Implementing A Read/Write Memory Module

With 2 address bits, how many memory elements can we address?
• 4 1-bit elements

How could we implement each memory element?
• With a D flip-flop
Memory Module Specification

“chip select” signal:

• Allows us to have multiple devices (e.g., memory modules) that can write to the bus

• But: only one device will ever be selected at one time
Memory Module Specification

When chip select is low:
• No memory elements change state
• The memory does not drive the data bus
Memory Module Specification

When chip select is high:

• If R/W is high:
  – The memory drives the data bus with the value that is stored in the element specified by A1, A0

• If R/W is low:
  – Store the value that is on the data bus in the memory element specified by A1, A0
Memory Timing Diagram

Q2

A1

A0

R/W

CS

CLK

D

Andrew H. Fagg: Embedded Real-Time Systems: Microcontrollers
Memory Timing Diagram

Q2

A1

A0

R/W

CS

CLK

Data bus not driven

D
Memory Timing Diagram

Memory element 2 is initially in a high state

Note: memory elements 0, 1, & 3 not shown
Memory Timing Diagram

What happens next?
Memory Timing Diagram

Q2

A1

A0

R/W

CS

CLK

D

Chip is selected
Memory Timing Diagram

Q2

A1

A0

R/W

CS

CLK

D

Address memory element 2
Memory Timing Diagram

- Specify a write operation
- Data bus is driven low (by another device)
Memory Timing Diagram

Clock goes low
Memory Timing Diagram

Memory element 2 changes state to low
Memory Timing Diagram II

Q2

A1

A0

R/W

CS

CLK

D
Memory Timing Diagram II

Q2
- __________
- 
A1
- __
- 
A0
- __________
- 
R/W
- __________
- 
CS
- __
- 
CLK
- 
D
- __________

Data bus is not driven
Memory Timing Diagram II

What happens next?
Memory Timing Diagram II

Q2

A1

A0

R/W

CS

CLK

D

On chip select – drive data bus from Q2
What happens now?
Data bus returns to a non-driven state

Memory Timing Diagram II
Memory Summary

• Many independent storage elements
• Elements are typically organized into 8-bit bytes
• Each byte has its own address
• The value of each byte can be read
• In RAM: the value can also be changed
One More Bus Note

Many devices on the bus. However, at a given time:

• There is exactly one device that is the “writer”
• There is exactly one that is the “reader”
Atmel Mega8 Architecture
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
Atmel Mega8

Random Access Memory (RAM)
- 1 KByte in size

Note: in high-end processors, RAM is a separate component
Atmel Mega8

Flash (EEPROM)

- Program storage
- 8 KByte in size

Andrew H. Fagg: Embedded Real-Time Systems: Microcontrollers
Atmel Mega8

Flash (EEPROM)

- In this and many microcontrollers, program and data storage is separate
- Not the case in our general purpose computers
Atmel Mega8

EEPROM

- Permanent data storage
Atmel Mega8

Arithmetic Logical Unit
- Data inputs from registers
- Control inputs not shown (derived from instruction decoder)
Machine-Level Programs

Machine-level programs are stored as sequences of atomic machine instructions

• Stored in program memory
• Execution is generally sequential (instructions are executed in order)
• But – with occasional “jumps” to other locations in memory
Types of Instructions

- Memory operations: transfer data values between memory and the internal registers
- Mathematical operations: ADD, SUBTRACT, MULT, AND, etc.
- Tests: value == 0, value > 0, etc.
- Program flow: jump to a new location, jump conditionally (e.g., if the last test was true)
Atmel Mega8: Decoding Instructions

Program counter

- Address of currently executing instruction
Atmel Mega8: Decoding Instructions

Instruction register
- Stores the machine-level instruction currently being executed
Atmel Mega8

Instruction decoder
- Translates current instruction into control signals for the rest of the processor
Some Mega8 Memory Operations

\textbf{LDS Rd, k}

- Load SRAM memory location \( k \) into register \( \text{Rd} \)
- \( \text{Rd} \leftarrow (k) \)

\textbf{STS Rd, k}

- Store value of \( \text{Rd} \) into SRAM location \( k \)
- \( (k) \leftarrow \text{Rd} \)

We refer to this as “Assembly Language”
Load SRAM Value to Register

LDS Rd, k
Store Register Value to SRAM

STS Rd, k
Some Mega8 Arithmetic and Logical Instructions

**ADD Rd, Rr**
- Rd and Rr are registers
- Operation: Rd <- Rd + Rr

**ADC Rd, Rr**
- Add with carry
- Rd <- Rd + Rr + C
Add Two Register Values

ADD Rd, Rr

- Fetch register values
Add Two Register Values

**ADD Rd, Rr**
- Fetch register values
- ALU performs ADD
Add Two Register Values

ADD Rd, Rr

- Fetch register values
- ALU performs ADD
- Result is written back to register via the data bus
Some Mega8 Arithmetic and Logical Instructions

**NEG Rd**: take the two’s complement of Rd

**AND Rd, Rr**: bit-wise AND with a register

**ANDI Rd, K**: bit-wise AND with a constant

**EOR Rd, Rr**: bit-wise XOR

**INC Rd**: increment Rd

**MUL Rd, Rr**: multiply Rd and Rr (unsigned)

**MULS Rd, Rd**: multiply (signed)
Some Mega8 Test Instructions

**CP Rd, Rr**
- Compare Rd with Rr

**TST Rd**
- Test for if register Rd is zero or a negative number
Some Program Flow Instructions

**RJMP** $k$
- Change the program counter by $k+1$
- $PC \leftarrow PC + k + 1$

**BRGE** $k$
- Branch if greater than or equal to
- If last compare was greater than or equal to, then $PC \leftarrow PC + k + 1$
Connecting Assembly Language to C

• Our C compiler is responsible for translating our code into Assembly Language

• Today, we rarely program in Assembly Language
  – Embedded systems are a common exception
  – Also: it is useful in some cases to view the assembly code generated by the compiler
An Example

A C code snippet:

```c
if(B < A) {
    D += A;
}
```
An Example

A C code snippet:

```c
if(B < A) {
    D += A;
}
```

The Assembly:

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

........
An Example

A C code snippet:

```c
if(B < A) {
    D += A;
}
```

Load the contents of memory location A into register 1

The Assembly:

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

........
An Example

A C code snippet:

```c
if(B < A) {
    D += A;
}
```

The Assembly:

1. Load the contents of memory location B into register 2
2. Load the contents of memory location A into register 1
3. Compare register 2 to register 1
4. Branch if greater than or equal to register 1
5. Load the contents of memory location D into register 3
6. Add register 3 to register 1
7. Store the result in memory location D
8. Repeat

PC
An Example

A C code snippet:

```c
if(B < A) {
    D += A;
}
```

The Assembly:

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

Compare the contents of register 2 with those of register 1

This results in a change to the status register
An Example

A C code snippet:

```c
if(B < A) {
    D += A;
}
```

Branch If Greater Than or Equal To:
jump ahead 3 instructions if true

The Assembly:

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

PC
An Example

A C code snippet:

```c
if(B < A) {
    D += A;
}
```

The Assembly:

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

Branch if greater than or equal to will jump ahead 3 instructions if true
An Example

A C code snippet:

```c
if(B < A) {
    D += A;
}
```

The Assembly:

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

Not true: execute the next instruction
An Example

A C code snippet:

```c
if(B < A) {
    D += A;
}
```

The Assembly:

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

Load the contents of memory location D into register 3
An Example

A C code snippet:

```c
if(B < A) {
    D += A;
}
```

Add the values in registers 1 and 3 and store the result in register 3

The Assembly:

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

PC
An Example

A C code snippet:

```c
if(B < A) {
    D += A;
}
```

The Assembly:

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

Store the value in register 3 back to memory location D.
Summary

Instructions are the “atomic” actions that are taken by the processor

• One line of C code typically translates to a sequence of several instructions

• In the mega 8, most instructions are executed in a single clock cycle

The high-level view is important here: don’t worry about the details of specific instructions
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

<table>
<thead>
<tr>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(RESET) PC6</td>
</tr>
<tr>
<td>2</td>
<td>(RXD) PD0</td>
</tr>
<tr>
<td>3</td>
<td>(TXD) PD1</td>
</tr>
<tr>
<td>4</td>
<td>(INT0) PD2</td>
</tr>
<tr>
<td>5</td>
<td>(INT1) PD3</td>
</tr>
<tr>
<td>6</td>
<td>(XCK/T0) PD4</td>
</tr>
<tr>
<td>7</td>
<td>VCC</td>
</tr>
<tr>
<td>8</td>
<td>GND</td>
</tr>
<tr>
<td>9</td>
<td>(XTAL1/TOSC1) PB6</td>
</tr>
<tr>
<td>10</td>
<td>(XTAL2/TOSC2) PB7</td>
</tr>
<tr>
<td>11</td>
<td>(T1) PD5</td>
</tr>
<tr>
<td>12</td>
<td>(AIN0) PD6</td>
</tr>
<tr>
<td>13</td>
<td>(AIN1) PD7</td>
</tr>
<tr>
<td>14</td>
<td>(ICP1) PB0</td>
</tr>
<tr>
<td>15</td>
<td>PB1 (OC1A)</td>
</tr>
<tr>
<td>16</td>
<td>PB2 (SS/OC1B)</td>
</tr>
<tr>
<td>17</td>
<td>PB3 (MOSI/OC2)</td>
</tr>
<tr>
<td>18</td>
<td>PB4 (MISO)</td>
</tr>
<tr>
<td>19</td>
<td>PB5 (SCK)</td>
</tr>
<tr>
<td>20</td>
<td>AVCC</td>
</tr>
<tr>
<td>21</td>
<td>AREF</td>
</tr>
<tr>
<td>22</td>
<td>GND</td>
</tr>
<tr>
<td>23</td>
<td>PC0 (ADC0)</td>
</tr>
<tr>
<td>24</td>
<td>PC1 (ADC1)</td>
</tr>
<tr>
<td>25</td>
<td>PC2 (ADC2)</td>
</tr>
<tr>
<td>26</td>
<td>PC3 (ADC3)</td>
</tr>
<tr>
<td>27</td>
<td>PC4 (ADC4/SDA)</td>
</tr>
<tr>
<td>28</td>
<td>PC5 (ADC5/SCL)</td>
</tr>
</tbody>
</table>
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

(RESET) PC6 1  28  □ PC5 (ADC5/SCL)
(RXD) PD0 2  27  □ PC4 (ADC4/SDA)
(TXD) PD1 3  26  □ PC3 (ADC3)
(INT0) PD2 4  25  □ PC2 (ADC2)
(INT1) PD3 5  24  □ PC1 (ADC1)
(XCK/T0) PD4 6  23  □ PC0 (ADC0)
VCC 7  22  □ GND
GND 8  21  □ AREF
(XTAL1/TOSC1) PB6 9  20  □ AVCC
(XTAL2/TOSC2) PB7 10  19  □ PB5 (SCK)
(T1) PD5 11  18  □ PB4 (MISO)
(AIN0) PD6 12  17  □ PB3 (MOSI/OC2)
(AIN1) PD7 13  16  □ PB2 (SS/OC1B)
(ICP) PB0 14  15  □ PB1 (OC1A)
Atmel Mega8 Basics

PORT C

(RESET) PC6 1
(RXD) PB0 2
(TXD) PD1 3
(INT0) PD2 4
(INT1) PD3 5
(XCK/T0) PD4 6
VCC 7
GND 8
XTAL1/TOSC1 PB6 9
XTAL2/TOSC2 PB7 10
(T1) PD5 11
(AIN0) PD6 12
(AIN1) PD7 13
(ICP1) PB0 14
PC5 (ADC5/SCL) 28
PC4 (ADC4/SDA) 27
PC3 (ADC3) 26
PC2 (ADC2) 25
PC1 (ADC1) 24
PC0 (ADC0) 23
GND 22
AREF 21
AVCC 20
PB5 (SCK) 19
PB4 (MISO) 18
PB3 (MOSI/OC2) 17
PB2 (SS/OC1B) 16
PB1 (OC1A) 15
Atmel Mega8 Basics

PORT D
(all 8 bits are available)
A First Circuit
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

PUD: PULLUP DISABLE
SLEEP: SLEEP CONTROL
clk\textsubscript{V0}: I/O CLOCK
WDx: WRITE DDR\textsubscript{x}
RDx: READ DDR\textsubscript{x}
WPx: WRITE PORT\textsubscript{x}
RRx: READ PORT\textsubscript{x} REGISTER
RPx: READ PORT\textsubscript{x} PIN
I/O Pin Implementation

The physical pin

PINx

PORTx

DDRx

PUD

WDx

RDx

WPx

RRx

RPx

PUD: PULLUP DISABLE
SLEEP: SLEEP CONTROL
clkVO: I/O CLOCK

WDx: WRITE DDRx
RDx: READ DDRx
WPx: WRITE PORTx
RRx: READ PORTx REGISTER
RPx: READ PORTx PIN

DATA BUS

clkVO
I/O Pin Implementation

DDRB
- Defines whether this is an input or an output
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

Tristate buffer

- When this pin is an output pin, it allows the PORTB flip-flop to drive the pin
Input tri-state buffer
I/O Pin Implementation

$\text{DDRB} = 0$;
I/O Pin Implementation

$\text{DDRB} = 0;$

• “0” is written to the data bus
I/O Pin Implementation

$\text{DDRB} = 0$;

- "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

**Diagram:**

- **DDRx**
- **PORTx**
- **PINx**
- **clikvo**
- **WDX**
- **RDX**
- **Wpx**
- **Rpx**
- **PUD**
- **SLEEP**
- **RDx**
- **Read DDRx**
- **Write DDRx**
- **Read PORTx**
- **Read PORTx register**
- **Read PORTx pin**
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

$$\text{DDRB} = 1;$$

- "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;

PUD: PULLUP DISABLE
SLEEP: SLEEP CONTROL
clk\textsubscript{V0}: I/O CLOCK
WDx: WRITE DDRx
RDx: READ DDRx
WPx: WRITE PORTx
RRx: READ PORTx REGISTER
RPx: READ PORTx PIN
PORTB = 1;

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

```
PORTB = 1;
```

```
• “1” is written to the data bus
• This is input to the PORTB register
```
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
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”
foo = PORTB;
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

```c
foo = PINB;
```
foo = PINB;

- RPB is set high
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?

\[ C = A \& B; \]
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

\[ \begin{array}{cccccccc}
0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\
\end{array} \]

\[ \begin{array}{cccccccc}
1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
\end{array} \]

\[ \begin{array}{cccccccc}
? \\
\end{array} \]

\[ C = A \& B \]
Bit-Wise Operators

\[
\begin{array}{c}
0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & A \\
1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & B \\
\hline
C = A \& B
\end{array}
\]
Bit-Wise Operators

0 1 0 1 1 1 0 A
1 0 0 1 1 0 1 1 B

0 C = A & B
Bit-Wise Operators

0 1 0 1 1 1 1 0  A

1 0 0 1 1 0 1 1  B

1 0  C = A & B
Bit-Wise Operators

\[ 01011110 \quad A \]
\[ 10011011 \quad B \]
\[ 00011010 \quad C = A \& B \]
Bit-Wise Operators

Other Operators:

• OR:  |
• XOR: ^
• NOT: ~
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?

$$A = A \mid 4;$$
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 0?

$$A = A \& 0xFFB;$$

or

$$A = A \& \sim 4;$$
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
A First Program

```c
main() {
    DDRB = 1;       // Set port B pin 0 as an output

    while(1) {
        PORTB = PORTB ^ 0x1;       // XOR bit 0 with 1
        delay_ms(500);             // Pause for 500 msec
    }
}
```
main() {
    DDRB = 3;   // Set port B pins 0, and 1 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

```c
main() {
    DDRB = 3; // Set port B pins 0, and 1 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
### Port-Related Registers

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

<table>
<thead>
<tr>
<th></th>
<th>Directional control</th>
<th>Writing</th>
<th>Reading</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port B</td>
<td>DDRB</td>
<td>PORTB</td>
<td>PINB</td>
</tr>
<tr>
<td>Port C</td>
<td>DDRC</td>
<td>PORTC</td>
<td>PINC</td>
</tr>
<tr>
<td>Port D</td>
<td>DDRD</td>
<td>PORTD</td>
<td>PIND</td>
</tr>
</tbody>
</table>
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)
And: we want to leave the other bits undisrupted

• How do we express this in code?
Bit Masking

```c
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) | ((val & 0x7)<<3); // Set the current B3-B5 to 0s 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

```c
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;
}
```
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
Compiling and Downloading Code

Preparing to program:

• See the Atmel HowTo (pointer from the schedule page)
• Windoze: Install AVR Studio and WinAVR
• OS X: Install OSX-AVR
  – We will use ‘make’ for compiling and downloading
• Linux: Install binutils, avr-gcc, avr-libc, and avrdude
  – Same as OS X