Components of a Microprocessor
Components of a Microprocessor

• Memory:
  – Storage of data
  – Storage of a program

• Registers: small, fast memories
  – General purpose: store arbitrary data
  – Special purpose: used to control the processor
Special Purpose Registers
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, it is a single wire
• Many different components can be attached to the bus
• Any component can take input from 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 instruction decoder (in the microprocessor case)
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
• So: 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)

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

\[2^{M-1}\]

Andrew H. Fagg: Embedded Real-Time Systems: Microcontrollers
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)

<table>
<thead>
<tr>
<th>Address</th>
<th>Stored value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x32</td>
</tr>
<tr>
<td>1</td>
<td>0xF1</td>
</tr>
<tr>
<td>2</td>
<td>0x11</td>
</tr>
<tr>
<td>3</td>
<td>0x67</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>0x7B</td>
</tr>
</tbody>
</table>

$2^N - 1$
Memory Operations

Read

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

reads the value from the memory location referenced by ‘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
Last Time

• Flip-flops as 1-bit storage devices
• Microprocessor components
  – Random access memory
  – Program memory
  – Instruction decoder
  – Arithmetic logical unit
• Binary and hexadecimal number systems
Today

- Memory behavior
- Atmel mega8 microcontroller
- Assembly language (just a hint)
- Digital I/O with the Atmel mega8
Administrivia

• Homework 2 is out
  – Due on February 14th (one week)
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”)

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

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
  – (more about this later)
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:
  – Drive 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 element specified by A1, A0
Memory Timing Diagram

Q2

A1

A0

R/W

CS

CLK

D
Memory Timing Diagram

Q2

A1

A0

R/W

CS

CLK

Data bus not driven

D

Andrew H. Fagg: Embedded Real-Time Systems: Microcontrollers 33
Memory element 2 is initially in a high state
Memory Timing Diagram

What happens next?
Memory Timing Diagram

Chip is selected
Memory Timing Diagram

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

**Setup time:** all inputs must be valid during this time
Hold time: all inputs must continue to be valid
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

Q2
\hspace{1cm}
A1
\hspace{1cm}
A0
\hspace{1cm}
R/W
\hspace{1cm}
CS
\hspace{1cm}
CLK
\hspace{1cm}
D

What happens next?
Memory Timing Diagram II

On chip select – drive data bus from Q2
What happens now?
Memory Timing Diagram II

Q2

A1

A0

R/W

CS

CLK

Data bus returns to a non-driven state

D

Data bus returns to a non-driven state
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 quickly
An Example: the Atmel Mega8
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
Random Access Memory (RAM)

- 1 KByte in size

Atmel Mega8
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
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 \textit{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
Atmel Mega8

Status register

- Many machine instructions affect the state of this register
Some Mega8 Memory Operations

**LDS Rd, k**
- Load SRAM memory location k into register Rd
- $Rd \leftarrow (k)$

**STS Rd, k**
- Store value of Rd into SRAM location k
- $(k) \leftarrow Rd$

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: \( Rd \leftarrow Rd + Rr \)
- Also affects status register (zero, carry, etc.)

**ADC Rd, Rr**
- Add with carry
- \( Rd \leftarrow Rd + Rr + C \)
Add Two Register Values

ADD Rd, Rr

- Fetch register values
Add Two Register Values

**ADD Rd, Rr**
- Fetch register values
- ALU performs **ADD**

![Diagram of microcontroller components including ALU, registers, and control units.](image)
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
- Alters the status register

**TST Rd**
- Test for zero or minus
- Alters the status register
Some Mega8 Test Instructions

Modify the status register
Some Program Flow Instructions

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

**BRCS** \( k \)
- Branch if carry set
- If \( C==1 \) then \( PC \leftarrow PC + k + 1 \)
Atmel Mega8: Decoding Instructions

Results in a change to the program counter

- May be conditioned on the status register
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
```

.......

Andrew H. Fagg: Embedded Real-Time Systems: Microcontrollers 100
An Example

A C code snippet:

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

The Assembly:

1. Load the contents of memory location A into register 1
2. Load the contents of memory location B into register 2
3. Compare R2 (B) with R1
4. If B is greater than or equal to A, branch to address 3
5. Load the contents of memory location D into register 3
6. Add R1 (A) and R3 (D) into R3
7. ...
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
```

Load the contents of memory location B into register 2
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
- ........
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:

```
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
Atmel Mega8 Basics

Power (we will use +5V)
Atmel Mega8 Basics

Ground

(PDIP)

(RESET) PC6 □ 1
(RXD) PD0 □ 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) □ 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

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
(RXD) PD0 □ 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
(ICP) PB0 □ 14

28 □ PC5 (ADC5/SCL)
27 □ PC4 (ADC4/SDA)
26 □ PC3 (ADC3)
25 □ PC2 (ADC2)
24 □ PC1 (ADC1)
23 □ PC0 (ADC0)
22 □ GND
21 □ AREF
20 □ AVCC
19 □ PB5 (SCK)
18 □ PB4 (MISO)
17 □ PB3 (MOSI/OC2)
16 □ PB2 (SS/OC1B)
15 □ PB1 (OC1A)
Atmel Mega8 Basics

PORT C

(RESET) PC6 □ 1
(RXD) PD0 □ 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

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

Control the pins through the I/O modules

- At the heart, these are registers ... that are implemented using D flip-flops!
I/O Pin Implementation

Single bit of PORT B
I/O Pin Implementation

The physical pin
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

![Diagram of I/O pin implementation with tristate buffer](image)
I/O Pin Implementation

Input tri-state buffer
Last Time

- Memory behavior
- Microprocessor components
- Manipulating the state of pins
  - Registers: DDRx, PORTx, and PINx
Today

• Homework 1 solution set has been posted
• Connecting C code to the I/O pins
• Bit Masking
• Getting into the hardware
  – Compiling and downloading code

• On Thursday: come ready with winavr and AVRstudio installed on your laptops
I/O Pin Implementation

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
I/O Pin Implementation

\[ \text{DDRB} = 0; \]

DDRx  PORTx  PINx

PUD: PULLUP DISABLE
SLEEP: SLEEP CONTROL
clk\text{\_IO}: I/O CLOCK

WDx: WRITE DDRx
RDx: READ DDRx
WPx: WRITE PORTx
RRx: READ PORTx REGISTER
RPx: READ PORTx PIN
I/O Pin Implementation

DDRB = 0;

- "0" is written to the data bus
I/O Pin Implementation

DDRB = 0;

- "0" is written to the data bus
- This is input to the DDRB register
I/O Pin Implementation

DDRB = 0;

• “0” is written to the data bus
• This is input to the DDRB register
• WDB is clocked from high to low
I/O Pin Implementation

$$DDRB = 0;$$

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

- PUD: PULLUP DISABLE
- SLEEP: SLEEP CONTROL
- clk_{IO}: I/O CLOCK
- WDx: WRITE DDRx
- RDx: READ DDRx
- WPx: WRITE PORTx
- RRx: READ PORTx REGISTER
- RPx: READ PORTx PIN
I/O Pin Implementation

DDRB = 0;

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

-> this is an input pin
I/O Pin Implementation

DDRB = 1;

- “1” is written to the data bus
$\text{DDRB} = 1$;

- “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

$\rightarrow$ this is an output pin
I/O Pin Implementation

PORTB = 1;

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

- "1" is written to the data bus
- This is input to the PORTB register
I/O Pin Implementation

PORTB = 1;

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

PORTB = 1;

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

-> output a “1”
PORTB = 0;

- “0” is written to the data bus

```
PUD: PULLUP DISABLE
SLEEP: SLEEP CONTROL
clk\(_{\text{IO}}\): I/O CLOCK
WDx: WRITE DDRx
RDx: READ DDRx
WPx: WRITE PORTx
RRx: READ PORTx REGISTER
RPx: READ PORTx PIN
```
I/O Pin Implementation

PORTB = 0;

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

-> output a "0"
I/O Pin Implementation

foo = PORTB;

PUD: PULLUP DISABLE
SLEEP: SLEEP CONTROL
clkIO: I/O CLOCK
WDx: WRITE DDRx
RDx: READ DDRx
WRx: WRITE PORTx
RRx: READ PORTx REGISTER
RPx: READ PORTx PIN
I/O Pin Implementation

foo = PORTB;

- RPB is set high

```c
foo = PORTB;
```

- RPB is set high
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
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

\[
\begin{align*}
01011110 & \quad A \\
10011011 & \quad B \\
\hline
? & \quad C = A \& B
\end{align*}
\]
Bit-Wise Operators

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

C = A & B
Bit-Wise Operators

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

A
B

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

\[
\begin{array}{c}
010111110 \\
10011011 \\
\hline
10
\end{array}
\]

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>01011110</td>
<td>10011011</td>
<td>00011010</td>
</tr>
</tbody>
</table>

\[ C = A \& B \]
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 1?

\[
A = A \& \ 0x\text{FB};
\]

or

\[
A = A \& \sim 4;
\]
I/O Pin Implementation

Single bit of PORT B
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

main() {
    DDRB = 7;   // Set port B pins 0, 1, and 2 as outputs

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

```c
main() {
    DDRB = 7;  // Set port B pins 0, 1, and 2 as outputs

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

What does this program do?
A Second Program

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

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

Flashes LED on PB1 at 1 Hz
on PB0: 0.5 Hz
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>
Last Time(s)

- Bit manipulation: pin hardware to code
- Bit masking
- Project 1
Today

• A bit more on bit masking
• Homework 1 discussion
• Serial communication

• Project 1 due in one week
More Bit Masking

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

• How do we express this in code?
Bit Masking

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

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

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

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

  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 = command_to_robot; // A value between 0 and 7

    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

```c
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; // A value between 0 and 7

    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

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;
}
```
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

```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;
}
```

Read the value from the port
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;
}
```

“Mask out” all bits except B6 and B7
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;
}
```

Right shift the result by 6 bits – so the value of B6 and B7 are now in bits 0 and 1 of “outval”
A Note About the C/Atmel Book

The book uses C syntax that looks like this:

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

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

```c
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

• We will work through the details on Thursday. Before then:
  – 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