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
• In a microprocessor, which component is allowed to write is usually determined by the code that is currently executing
Atmel Mega2560 Architecture

Andrew H. Fagg: Embedded Real-Time Systems: Microcontrollers
Atmel Mega2560

8-bit data bus

- Primary mechanism for data exchange
Atmel Mega2560

32 general purpose registers
• 8 bits wide
• 3 pairs of registers can be combined to give us 16 bit registers
Atmel Mega2560

Special purpose registers

- Control of the internals of the processor
Atmel Mega2560

Random Access Memory (RAM)
- 8 KByte in size
Random Access Memory (RAM)
- 8 KByte in size

Note: in high-end processors, RAM is a separate component
Atmel Mega2560

Flash (EEPROM)

- Program storage
- 256 KByte in size

Andrew H. Fagg: Embedded Real-Time Systems: Microcontrollers
Atmel Mega2560

Flash (EEPROM)

- In this and many microcontrollers, program and data storage is separate
- Not the case in our general purpose computers
Atmel Mega2560

EEPROM

- Permanent data storage
Atmel Mega2560

Arithmetic Logical Unit

- Data inputs from registers
- Control inputs not shown (derived from instruction decoder)
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 Mega2560 (and 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)

<table>
<thead>
<tr>
<th>0x32</th>
<th>0xF1</th>
<th>0x11</th>
<th>0x67</th>
<th>......</th>
<th>0x7B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2^{M-1}</td>
<td></td>
</tr>
</tbody>
</table>
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>0x32</td>
<td>0x32</td>
</tr>
<tr>
<td>0xF1</td>
<td>0xF1</td>
</tr>
<tr>
<td>0x11</td>
<td>0x11</td>
</tr>
<tr>
<td>0x67</td>
<td>0x67</td>
</tr>
<tr>
<td>......</td>
<td>......</td>
</tr>
<tr>
<td>0x7B</td>
<td>0x7B</td>
</tr>
</tbody>
</table>

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

Read

\texttt{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 \texttt{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
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)
Mega2560: Decoding Instructions

- Program counter
  - Address of currently executing instruction
Mega2560: Decoding Instructions

Instruction register
- Stores the machine-level instruction currently being executed
Atmel Mega2560

Instruction decoder

- Translates current instruction into control signals for the rest of the processor
Some Mega2560 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 Mega2560 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 Mega2560 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 $k+1$
- $PC <- PC + k + 1$

**BRGE k**
- Branch if greater than or equal to
- If last compare was greater than or equal to, then $PC <- PC + k + 1$
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:

```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
```

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

A C code snippet:

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

Load the contents of memory location B into register 2

The Assembly:

```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
```

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;
}
```

Not true: execute the next instruction

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
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
```

........
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

PC
The Important Stuff

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 2560, 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
Mega2560
Atmel Mega2560

Pins are organized into 8-bit “Ports”:

- A, B, C ... L
  - But no “I”
Digital Input/Output

• Each port has three registers that control its behavior.

• For port B, they are:
  – DDRB: data direction register B
  – PORTB: port output register B
  – PINB: port input B
A First Circuit
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

0 1 0 1 1 1 1 0  \quad A

1 0 0 1 1 0 1 1  \quad B

?  \quad C = A \& B
Bit-Wise Operators

\[ 01011110 \quad A \]
\[ 10011011 \quad B \]

C = A & B
Bit-Wise Operators

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

0 C = A \& B
Bit-Wise Operators

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

\[ \begin{array}{l}
  01011110 \quad A \\
  10011011 \quad B \\
  \hline \\
  00011010 \quad C = A \& B \\
\end{array} \]
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?

$$A = A | 4;$$
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?

$$A = A \& \ 0xFB;$$

or

$$A = A \& \sim 4;$$
Bit Shifting

```c
uint8_t A = 0x5A;
uint8_t B = A << 2;
uint8_t C = A >> 5;
```

What are the values of B and C?
What mathematical operations have we performed?
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 PC0
A First Program

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

    while(1) {
        PORTC = PORTC | 0x1;
        delay_ms(500);
        PORTC = PORTC & ~0x1;
        delay_ms(500);
    }
}
```
main() {
    DDRC = 1;  // Set port C pin 0 as an output

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

```c
main() {
    DDRC = 3;   // Set port C pins 0, and 1 as outputs

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

What does this program do?
A Second Program

```c
main() {
    DDRC = 3;   // Set port C pins 0, and 1 as outputs

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

**Flashes LED on PC1 at 1 Hz**

**on PC0: 0.5 Hz**
Port-Related Registers

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

<table>
<thead>
<tr>
<th>Port</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 undisturbed

• 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 = ??? // Fill this in
}
```
Bit Masking

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

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

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

    PORTB = (PORTB & ~0x38) | ((val & 0x7)<<3);  // OR with new values (shifted to fit within B3-B5
}
Reading the Digital State of Pins

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

• How do we configure the port?
• How do we read the pins?
• How do we translate their values into an integer of 0 .. 3?
Reading the Digital State of Pins

```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 = ??? // Read the input value of B

    outval = ??? // Translate to a value 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;
}
```
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