Microprocessors
Review: Components of a Microprocessor

What are they?
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: temporarily 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: add, subtract, multiply, AND, OR …

• 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
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

Andrew H. Fagg: Embedded Real-Time Systems: Microcontrollers
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
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)
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)
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
Atmel Instructions
Some Mega2560 Memory Operations

LDS Rd, k
- Load SRAM memory location k into register Rd
- Rd ← (k)

STS Rd, k
- Store value of Rd into SRAM location k
- (k) ← 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**
- Add Rd and Rr (these 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, Rr: multiply (signed)
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
```

……….
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;
}
```

Compare the contents of register 2 with those of register 1

This results in a change to the status register

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;
}
```

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

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;
}
```

Branch if greater than or equal to
will 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
```

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
```

if not true
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:

```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;
}
```

Store the value in register 3 back to memory location D

The Assembly:

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

......
Take-Aways

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

- Many different component work together to execute a single instruction
- One line of C code typically translates into a sequence of several instructions
- In the Teensy, most instructions are executed in a single clock cycle

The high-level view is important here: you won’t be compiling programs on exams
An Example

#include "oulib.h"

volatile uint8_t a = 10;

int main (void)
{
    a = a+5;

    while(1) {
        delay_ms(++a);
    }
}
0000013c <main>:

volatile uint8_t a = 10;

int main (void)
{
    a = a+5;
    13c:  80 91 00 02  lds  r24, 0x0200
    140:  8b 5f         subi  r24, 0xFB         ; 251
    142:  80 93 00 02  sts  0x0200, r24

    while(1) {
        delay_ms(++a);
        146:  80 91 00 02  lds  r24, 0x0200
        14a:  8f 5f         subi  r24, 0xFF         ; 255
        14c:  80 93 00 02  sts  0x0200, r24
        150:  80 91 00 02  lds  r24, 0x0200
        154:  90 e0         ldi  r25, 0x00         ; 0
        156:  0e 94 ae 00  call  0x15c    ; 0x15c <delay_ms>
        15a:  f5 cf         rjmp   .-22           ; 0x146 <main+0xa>
0000013c <main>:
volatile uint8_t a = 10;

int main (void)
{
    a = a+5;
    while(1) {
        delay_ms( ++a );
    }
}

Location in program memory

Andrew H. Fagg: Embedded Real-Time Systems: Microprocessors
0000013c <main>:
volatile uint8_t a = 10;

int main (void)
{
    a = a+5;
    13c:  80 91 00 02       lds   r24, 0x0200
    140:  8b 5f             subi  r24, 0xFB         ; 251
    142:  80 93 00 02       sts   0x0200, r24

    while(1) {
        delay_ms(++a);
        146:  80 91 00 02       lds   r24, 0x0200
        14a:  8f 5f             subi  r24, 0xFF         ; 255
        14c:  80 93 00 02       sts   0x0200, r24
        150:  80 91 00 02       lds   r24, 0x0200
        154:  90 e0             ldi   r25, 0x00         ; 0
        156:  0e 94 ae 00       call  0x15c   ; 0x15c <delay_ms>
        15a:  f5 cf             rjmp   .-22           ; 0x146 <main+0xa>

volatile uint8_t a = 10;

int main (void)
{
    a = a+5;
    while(1) {
        delay_ms(++a);
        lds r24, 0x0200
        subi r24, 0xFB ; 251
        sts 0x0200, r24
        lds r24, 0x0200
        subi r24, 0xFF ; 255
        sts 0x0200, r24
        lds r24, 0x0200
        ldi r25, 0x00 ; 0
        call 0x15c ; 0x15c <delay_ms>
        rjmp .-22 ; 0x146 <main+0xa>
0000013c <main>:

volatile uint8_t a = 10;

int main (void)
{
    a = a+5;
    13c: 80 91 00 02 lds r24, 0x0200
    140: 8b 5f subi r24, 0xFB ; 251
    142: 80 93 00 02 sts 0x0200, r24

    while(1) {
        delay_ms(++a);
        146: 80 91 00 02 lds r24, 0x0200
        14a: 8f 5f subi r24, 0xFF ; 255
        14c: 80 93 00 02 sts 0x0200, r24
        150: 80 91 00 02 lds r24, 0x0200
        154: 90 e0 ldi r25, 0x00 ; 0
        156: 0e 94 ae 00 call 0x15c ; 0x15c <delay_ms>
        15a: f5 cf rjmp .-22 ; 0x146 <main+0xa>

    Store r24 to memory location 0x200

Andrew H. Fagg: Embedded Real-Time Systems: Microprocessors
0000013c <main>:

volatile uint8_t a = 10;

int main (void)
{
    a = a+5;
    13c:  80 91 00 02 lds r24, 0x0200
    140:  8b 5f   subi r24, 0xFB  ; 251
    142:  80 93 00 02 sts 0x0200, r24

    while(1) {
        delay_ms(++a);
        146:  80 91 00 02 lds r24, 0x0200
        14a:  8f 5f   subi r24, 0xFF  ; 255
        14c:  80 93 00 02 sts 0x0200, r24
        150:  80 91 00 02 lds r24, 0x0200
        154:  90 e0   ldi r25, 0x00  ; 0
        156:  0e 94 ae 00 call 0x15c  ; 0x15c <delay_ms>
        15a:  f5 cf   rjmp -.22        ; 0x146 <main+0xa>
    }
}
0000013c <main>:

volatile uint8_t a = 10;

int main (void)
{
    a = a+5;
    while(1) {
        delay_ms(++a);
        lds r24, 0x0200
        subi r24, 0xFB ; 251
        sts 0x0200, r24
        lds r24, 0x0200
        subi r24, 0xFF ; 255
        sts 0x0200, r24
        lds r24, 0x0200
        ldi r25, 0x00 ; 0
        call 0x15c ; 0x15c <delay_ms>
        rjmp -22 ; 0x146 <main+0xa>
    }
}
0000013c <main>: 

volatile uint8_t a = 10;

int main (void)
{
    a = a+5;
    13c:  80 91 00 02  lds  r24, 0x0200
    140:  8b 5f  subi  r24, 0xFB    ; 251
    142:  80 93 00 02  sts  0x0200, r24

    while(1) {
        delay_ms(++a);
        146:  80 91 00 02  lds  r24, 0x0200
        14a:  8f 5f  subi  r24, 0xFF    ; 255
        14c:  80 93 00 02  sts  0x0200, r24
        150:  80 91 00 02  lds  r24, 0x0200
        154:  90 e0  ldi  r25, 0x00    ; 0
        156:  0e 94 ae 00  call  0x15c    ; 0x15c <delay_ms>
        15a:  f5 cf  rjmp  .-22        ; 0x146 <main+0xa>

    Store r24 to memory location 0x200
0000013c <main>:

volatile uint8_t a = 10;

int main (void)
{
    a = a+5;
    a = a+5;
    while(1) {
        delay_ms(++a);
        delay_ms(++a);
    }
}

0000013c <main>:

volatile uint8_t a = 10;

int main (void)
{
    a = a+5;
    while(1) {
        delay_ms(++a);
    }
}
Compiled Result

0000013c <main>:

volatile uint8_t a = 10;

int main (void)
{
    a = a+5;

    while(1) {
        delay_ms(++a);
    }

    Call delay_ms()

    a = a+5;

    while(1) {
        delay_ms(++a);
    }

    Call delay_ms()

    a = a+5;

    while(1) {
        delay_ms(++a);
    }

    Call delay_ms()

    a = a+5;

    while(1) {
        delay_ms(++a);
    }

    Call delay_ms()
volatile uint8_t a = 10;

int main (void)
{
    a = a+5;

    while(1) {
        delay_ms(++a);

        lds   r24, 0x0200
        subi  r24, 0xFB   ; 251
        sts   0x0200, r24

        lds   r24, 0x0200
        subi  r24, 0xFF   ; 255
        sts   0x0200, r24

        lds   r24, 0x0200
        ldi   r25, 0x00   ; 0
        call  0x15c       ; 0x15c <delay_ms>
        rjmp  .-22        ; 0x146 <main+0xa>
    }
}

Go back to top of while() loop
Example II

#include "oulib.h"

volatile uint16_t a = 10;

int main (void)
{
    a = a+5;

    while(1) {
        delay_ms(++a);
    }
}
#include "oulib.h"

volatile uint16_t a = 10;

int main (void)
{
    a = a+5;

    while(1) {
        delay_ms(++a);
    };
}
000013c <main>:
volatile uint16_t a = 10;
int main (void)
{
    a = a+5;

    while(1) {
        delay_ms(++a);
    }

    80 91 00 02  lds    r24, 0x0200
    90 91 01 02  lds    r25, 0x0201
    05 96       adiw   r24, 0x05    ; 5
    90 93 01 02  sts    0x0201, r25
    80 93 00 02  sts    0x0200, r24

    80 91 00 02  lds    r24, 0x0200
    90 91 01 02  lds    r25, 0x0201
    01 96       adiw   r24, 0x01    ; 1
    90 93 01 02  sts    0x0201, r25
    80 93 00 02  sts    0x0200, r24
    80 91 00 02  lds    r24, 0x0200
    90 91 01 02  lds    r25, 0x0201
    0e 94 b7 00  call    0x16e    ; 0x16e <delay_ms>
    f0 cf       rjmp    .-32    ; 0x14e <main+0x12>
0000013c <main>:
volatile uint16_t a = 10;
int main (void)
{
    a = a+5;
    13c:  80 91 00 02  lds    r24, 0x0200
    140: 90 91 01 02  lds    r25, 0x0201
    144: 05 96      adiw   r24, 0x05          ; 5
    146: 90 93 01 02  sts    0x0201, r25
    14a: 80 93 00 02  sts    0x0200, r24

    while(1) {
        delay_ms(++a);
        14e: 80 91 00 02  lds    r24, 0x0200
        152: 90 91 01 02  lds    r25, 0x0201
        156: 01 96      adiw   r24, 0x01          ; 1
        158: 90 93 01 02  sts    0x0201, r25
        15c: 80 93 00 02  sts    0x0200, r24
        160: 80 91 00 02  lds    r24, 0x0200
        164: 90 91 01 02  lds    r25, 0x0201
        168: 0e 94 b7 00  call 0x16e      ; 0x16e <delay_ms>
        16c: f0 cf      rjmp   .-32            ; 0x14e <main+0x12>

Compiled Result
Load memory locations 0x201, 0x200 to r25, r24
Compiled Result

Add 5 to r25, r24

Andrew H. Fagg: Embedded Real-Time Systems: Microprocessors
0000013c <main>:
volatile uint16_t a = 10;
int main (void)
{
    a = a+5;
13c:  80 91 00 02  lds  r24, 0x0200
140:  90 91 01 02  lds  r25, 0x0201
144:  05 96  adiw  r24, 0x05  ; 5
146:  90 93 01 02  sts  0x0201, r25
14a:  80 93 00 02  sts  0x0200, r24

    while(1) {
        delay_ms(++a);
14e:  80 91 00 02  lds  r24, 0x0200
152:  90 91 01 02  lds  r25, 0x0201
156:  01 96  adiw  r24, 0x01  ; 1
158:  90 93 01 02  sts  0x0201, r25
15c:  80 93 00 02  sts  0x0200, r24
160:  80 91 00 02  lds  r24, 0x0200
164:  90 91 01 02  lds  r25, 0x0201
168:  0e 94 b7 00  call  0x16e  ; 0x16e <delay_ms>
16c:  f0 cf  rjmp  .-32  ; 0x14e <main+0x12>

Andrew H. Fagg: Embedded Real-Time Systems: Microprocessors 68
0000013c <main>:
volatile uint16_t a = 10;
int main (void)
{
    a = a+5;
}

while(1) {
    delay_ms(++a);
}

Store r25, r24 to memory locations 0x201, 0x200

We have doubled the number of memory operations!
Take-Home Message I

We want to carefully choose our data types

- Smaller variables are handled more efficiently
- But: we need to make sure that the results of the math that we do with these variables fits in the size that we have chosen
  – Intermediate values must fit, too!
Take-Home Message II

- A line of C code usually translates into a sequence of atomic instructions.
- Most instructions are executed in one cycle of the system clock.
- For a given instruction, many different components work together to make that instruction happen:
  - Program counter, instruction register and decoder, general and special purpose registers, memory, ALU, etc.
Take-Home Message III

• You should know what these different components are and what they do at an abstract level
• You don’t need to know the details of the assembly language or how these details relate to specific lines of C code