CprE 288 – Introduction to Embedded Systems
ATmega128 Assembly Programming: Moving Data & Control Flow

Instructors:
Dr. Phillip Jones
Dr. Zhao Zhang

Announcements

• HW11: Due Friday 4/22 Midnight
• Everyone should now be assigned to a project team by end of this week
• Everyone assigned an SVN team repository by end of this week
• Projects
  – Lab attendance is mandatory
  – For each lab you miss you will lose 10 points on the project
    (See supplemental specification document, to be released)

C and Assembly Mixed Programming

- It's not productive to write large programs in assembly
- Mixed C/Assembly programming
  - Write the majority of code in C
  - Write time-critical code in assembly, or other code that has to be written in assembly
- Two forms of mixed C/Assembly
  - Inline assembly: include assembly code in .c file
  - Include assembly .S files in the project

We will use GCC assembly syntax and non-inline assembly code

Major Classes of Assembly Instructions

- Data Movement
  - Move data between registers
  - Move data in & out of SRAM
  - Different addressing modes
- Logic & Arithmetic
  - Addition, subtraction, etc.
  - AND, OR, bit shift, etc.
- Control Flow
  - Control which sections of code should be executed (e.g. in C "IF", "CASE", "WHILE", etc.
  - Typically the result of Logic & Arithmetic instructions help decide what path to take through the code.

C and Assembly Mixed Programming

```c
#include <avr/io.h>
#include <stdio.h>

cart ch1 = 0x30;
cart ch2 = 0x40;
int a = 0x1010;

void asm_func();

int main()
{
  asm_func();
}
```

C and Assembly Mixed Programming

```assembly
.global asm_func
.extern ch1 ch2 a

asm_func:
    ; Put our code here
    ... 
    ret
```
Major Classes of Assembly Instructions

• Data Movement
  – Move data between registers
  – Move data in & out of SRAM
  – Different addressing modes

• Logic & Arithmetic
  – Addition, subtraction, etc.
  – AND, OR, bit shift, etc.

• Control Flow
  – Control which sections of code should be executed (e.g. In C “IF”, “CASE”, “WHILE”, etc.
  – Typically the result of Logic & Arithmetic instructions help decided what path to take through the code.

Instructions to move data: Summary

• LDI Rd, K  Load Immediate Rd ← K  1 clk
• MOV Rd, Rr  Move Between Registers Rd ← Rr  1 clk
• LDS Rd, (k)  Load Direct Rd ← (k)  2 clks
  – Note: There is a ST version for each LD (except for LDI)
• LD Rd, Y  Load Indirect Rd ← (X)  2 clks
• LD Rd, Y+  Load Indirect & Post-Inc. Rd←(X), X← X + 1  2clks
• LDD Rd, Y+q  Load Indirect + offset. Rd←(X+q)  2 clks

LDI & MOV

• LDI Rd, K  Load Immediate Rd ← K  1 clk
• MOV Rd, Rr  Move Between Registers Rd ← Rr  1 clk
• Only need one clock cycle to execute
  – All parameters needed for execution available to ALU

Load Immediate

LDI: Load an 8-bit constant (limited to R16-R31)
Syntax:  LDI Rd, K
Operands:  16 ≤ d ≤ 31, 0 ≤ K ≤ 255
Operations:  Rd ← K, PC ← PC+1
Binary
Format
Cycles: 1
See 8-bit AVR Inst. Set Page 89
Question: Why limited to R16-R31?

Copy Register

MOV: Copy one register to another
Syntax:  MOV Rd, Rr
Operands:  0 ≤ d ≤ 31, 0 ≤ r ≤ 31
Operations:  Rd ← Rr, PC ← PC+1
Example:
    mov r16,r0 ; Copy r0 to r16

Copy Register Word

MOVW: Copy one register to another
Syntax (AVR):  MOVW Rd+1:Rd, Rr+1:Rr
Syntax (GCC):  MOVW Rd, Rr
Operands:  d=0,2,...,30, r=0,2,...,30
Operations:  Rd+1:Rd ← Rr+1:Rr, PC ← PC+1
Example:
    (AVR) movw r17:16, r1:r0
    (GCC) movw r16, r0
Load Immediate

```c
char a; // Assume a is at location 0xFFC0
...
```

```assembly
a = 0x10;
// Note: As a short cut the AVR compiler allows the
// programmer to use a C variable name within the
// assembly code. Be aware this gets translated into the
// memory location of the variable (sts a, r24 is the same
// as sts 0xFFC0, r24)
ldi r24, 0x10 ; Load imm 10
sts a, r24 ; Store to a
```

Copy Register & Copy Register Word

Make R2 = 0x10
- Recall: Cannot use LDI on R2
```assembly
ldi r24, 0x10 ; r24 = 0x10
mov r2, r24 ; r2 = r24
```

Make R5:R4 = 0x3020 using three instructions
```assembly
ldi r24, 0x10 ; r24 = 0x10
mov r2, r24 ; r2 = r24
movw r4, r24 ; r5:r4 = r25:r24
```

LDS (Load Direct from Data Space)

- LDS Rd, (k)  
  - Load Direct \( \text{Rd} \leftarrow (k) \)  2 clks
- Note:

```
LDS Rd, (k)
```

Load Direct

LDS: Load direct from storage space (data memory)

Syntax: LDS Rd, k
Operands: \( 0 \leq d \leq 31, 0 \leq k \leq 65,535 \)
Operations: \( \text{Rd} \leftarrow (k), \text{PC} \leftarrow \text{PC}+2 \)

Binary Format

Cycles: 2
See 8-bit AVR Inst. Set Page 90

Store Direct

STS: Store direct to storage space (data memory)

Syntax: STS k, Rr
Operands: \( 0 \leq r \leq 31, 0 \leq k \leq 65,535 \)
Operations: \( (k) \leftarrow \text{Rr}, \text{PC} \leftarrow \text{PC}+2 \)

Binary Format

Cycles: 2
See 8-bit AVR Inst. Set Page 113

Exercise

```c
int a; // assume a is located at 0xFD00
...
a = 0x2030;
```
Load and Store

```c
int a; // Assume variable a is at 0x0100
    a = 0x2030;
ldi r24, 0x30; // Store to lower half
sts 0x0100, r24;
ldi r24, 0x20; // Store to higher half
sts 0x0101, r24;
```

; Short-cut syntax, assembler knows &a is 0x0100
```c
ldi r24, 0x30; r24 = 0x30
sts 0x0100, r24;
```

; Short-cut syntax, assembler knows &a is 0x0100
```c
ldi r24, 0x20; r24 = 0x20
sts 0x0101, r24;
```

Load and Store

A full function example
```c
extern int a, b;
char myfunc()
{
    a = b;
}
```

Assembly code
```c
global myfunc
.extern a b
myfunc:
    lds r24, b
    lds r25, b+1
    sts a, r24
    sts a+1, r25
    ret
```

Another function example
```c
extern char a, b;
char myfunc()
{
    a = b;
}
```

Assembly code
```c
global myfunc
.extern a b
myfunc:
    lds r24, b
    sts a, r24
    ret
```

Exercise: LDI, LDS, STS, ADD

```c
char a; // at 0x0100
char b; // at 0x0101
int my_x; // at 0x0102
int my_y; // at 0x0104

a = 0x5;
b = 0x43;
my_x = 0x6070;
my_y = my_x;
b = b + a;
```

Exercise: LDI, LDS, STS, ADD

```c
; a = 0x5;
LDI R18, 0x5
```

```
```
Exercise: LDI, LDS, STS, ADD

; a = 0x5;
LDI R18, 0x5
STS 0x0100, R18

; same as
; STS a, R18

Exercise: LDI, LDS, STS, ADD

; a = 0x5;
LDI R18, 0x5
STS 0x0100, R18

; b = 0x43
LDI R19, 0x43
STS 0x0101, R19

; same as
; STS b, R19

Exercise: LDI, LDS, STS, ADD

; a = 0x5;
LDI R18, 0x5
STS 0x0100, R18

; b = 0x43
LDI R19, 0x43
STS 0x0101, R19

; my_x = 0x6070
LDI R20, 0x70
LDI R21, 0x60
Exercise: LDI, LDS, STS, ADD

; a = 0x5;
LDI R18, 0x5
STS 0x0100, R18
; b = 0x43
LDI R19, 0x43
STS 0x0101, R19
; my_x = 0x6070
LDI R20, 0x70
LDI R21, 0x60
STS 0x0102, R20
STS 0x0103, R21

Exercise: LDI, LDS, STS, ADD

; a = 0x5;
LDI R18, 0x5
STS 0x0100, R18
; b = 0x43
LDI R19, 0x43
STS 0x0101, R19
; my_x = 0x6070
LDI R20, 0x70
LDI R21, 0x60
STS 0x0102, R20
STS 0x0103, R21

Exercise: LDI, LDS, STS, ADD

; b = b + a
LDS R0, 0x0100
LDS R1, 0x0101
ADD R1, R0

Exercise: LDI, LDS, STS, ADD

; a = 0x5;
LDI R18, 0x5
LDS R0, 0x0100
LDI R1, 0x0101
; b = b + a
LDS R0, 0x0100
LDS R1, 0x0101
ADD R1, R0
Exercise: LDI, LDS, STS, ADD

; a = 0x5;
LDI R18, 0x5
STS 0x0100, R18
; b = 0x43
LDI R19, 0x43
STS 0x0101, R19
; my_x = 0x6070
LDI R20, 0x70
LDI R21, 0x60
STS 0x0102, R20
STS 0x0103, R21
; my_y = my_x
STS 0x0104, R20
STS 0x0105, R21

LD (Load Indirect from Data Space)

• LD Rd, X  \[ \text{Load Indirect } R_d \leftarrow (X) \]

X, Y, Z Registers

Three indirect address (pointer) registers: X, Y, and Z

X \( \Rightarrow \) R27:R26
Y \( \Rightarrow \) R29:R28
Z \( \Rightarrow \) R31:R30

Use the GPR names to manipulate the pointers
Use the X, Y, Z names to dereference

Example: Load Using a Pointer

char *str;
char ch;

ch = *str;

How many loads do we have to use?

Steps:
1. Load the contents of str (it contains an address)
2. Load the contents of the dereferenced address (i.e. *str)
3. Store to ch the contents of the dereferenced address

Exercise: LDI, LDS, STS, LD

char *str; // at F000
char ch; // at FC00

str = 0xFA00;
ch = *str;

Exercise: LDI, LDS, STS, LD

char *str; // at F000
char ch; // at FC00

str = 0xFA00;
ch = *str;
// Assign str
LDI R16, 0x00;

X, Y, Z Registers

Three indirect address (pointer) registers: X, Y, and Z

X \( \Rightarrow \) R27:R26
Y \( \Rightarrow \) R29:R28
Z \( \Rightarrow \) R31:R30

Use the GPR names to manipulate the pointers
Use the X, Y, Z names to dereference

Example: Load Using a Pointer

char *str;
char ch;

ch = *str;

How many loads do we have to use?

Steps:
1. Load the contents of str (it contains an address)
2. Load the contents of the dereferenced address (i.e. *str)
3. Store to ch the contents of the dereferenced address

Exercise: LDI, LDS, STS, LD

char *str; // at F000
char ch; // at FC00

str = 0xFA00;
ch = *str;
// Assign str
LDI R16, 0x00;
Exercise: LDI, LDS, STS, LD

char *str; // at FD00
char ch; // at FC00

str = 0xFA00;
ch = *str;
// Assign str
LDI R16, 0x00;
LDI R17, 0xFF;

// Assign ch
LDI R30, 0x00;
LDS R31, 0xFF;

// Load contents of str
LDS R30, 0xFD00;
LD R18, Z;
Exercise: LDI, LDS, STS, LD

```c
char *str; // at FD00
char ch;   // at FC00

str = 0xFA00;
ch = *str;
```

Example of Encoding

**LDD Y+q** (Load indirect with Displacement)

- LDD Rd, Y+q  Load Indirect + offset Rd ← (Y+q)  2 clks

```
<table>
<thead>
<tr>
<th>Y or Z REGISTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
</tr>
<tr>
<td>16:0</td>
</tr>
</tbody>
</table>

• What is the significance of q being 6-bits in size?

Example: Load Using a Pointer

```c
char ch = *str; // str at FD00
// Assumes str is not initialized
// Using short cut syntax
// Warning can be confusing

; Use the Z register (Rr31:r30)
lds r30, str ; Load str, lo8
lds r31, str+1 ; Load str, hi8
ld r24, Z ; Load *str
sts ch, r24 ; store to a
```

Exercise: LDI, LDS, STS, LD, LDD

```c
int *pInt; // at FD00
int a;     // at FC00

pInt = 0xFA00;
a = *pInt;
```

Example: Encoding

**Syntax (using Z):** LDD Rd, Z+q
Operands: 0 ≤ d ≤ 31, 0 ≤ q ≤ 63
Operations: Rd ← (Z+q), PC ← PC+1

Binary Format

| 10-q | qqqqd | dddd | qqqq |

Cycle: 2
See 8-bit AVR Inst. Set Page 88
Note: Unique coding for X, Y, and Z
Where is the index of Z (R31:R30)?

Exercise: LDI, LDS, STS, LD, LDD

```c
int *pInt; // at FD00
int a;     // at FC00

pInt = 0xFA00;
a = *pInt;
```

```c
int *pInt; // at FD00
int a;     // at FC00

pInt = 0xFA00;
a = *pInt;
```
Exercise: LDI, LDS, STS, LD, LDD

```c
int *pInt; // at FD00
int a; // at FC00

pInt = 0xFA00;
// Assign pInt
LDI R16, 0x00;
LDI R17, 0xFA;
STS 0xFD00, R16;
STS 0xFD01, R17;
// Load contents of pInt
LDS R30, 0xFD00;
LDS R31, 0xFD01;
// Load contents of dereferenced address
LD R18, 0x70;
```

Exercise: LDI, LDS, STS, LD, LDD

```c
int *pInt; // at FD00
int a; // at FC00

pInt = 0xFA00;
a = *pInt;
// Assign pInt
LDI R16, 0x00;
LDI R17, 0xFA;
STS 0xFD00, R16;
STS 0xFD01, R17;
// Load contents of pInt
LDS R30, 0xFD00;
LDS R31, 0xFD01;
// Load contents of dereferenced address
LD R18, Z;
```
Exercise: LDI, LDS, STS, LD, LDD

```c
int *pInt; // at FD00
int a; // at FC00
pInt = 0xFA00;
a = *pInt; // Assign pInt
LDI R16, 0x00;
LDI R17, 0xFA;
STS 0xFD00, R16;
STS 0xFD01, R17;
// Load contents of pInt
LDS R30, 0xFD00;
LDS R31, 0xFD01;
// Load contents of dereferenced address
LD R18, Z;
LDD R19, Z+1;

STS 0xFC00, R18; // Store to a
STS 0xFC01, R19; // Store upper byte to a
```

Example: Load Using a Pointer

```c
int a = *pInt; // Assumes pInt is not initialized
// Using short cut syntax
// Warning this syntax can be confusing
; Use the Z register (r31:r30)
lds r30, pInt; // Load pInt
lds r31, pInt+1;
lr r24, Z; // Store (*pInt)
Ldd r25, Z+1;
sts a, r24; // Store to a
sts a+1, r25;
```

X, Y, Z Registers

Three formats for loading indirect using X

- **LD Rd, X**: 
  - X: Unchanged
  - Rd ← (X), X ← X+1
- **LD Rd, -X**: 
  - X: Pre decrement
  - X ← X-1, Rd ← (X)

Rd can be any of R0-R31
Latency: 2 clks

Three formats for storing indirect using X

- **ST X, Rd**: 
  - X: Unchanged
  - (X) ← Rd, X ← X+1
- **ST X+, Rd**: 
  - X: Post increment
  - (X) ← Rd, X ← X+1
- **ST –X, Rd**: 
  - X: Pre decrement
  - X ← X-1, (X) ← Rd

Rd can be any of R0-R31
Latency: 2 clks
**X, Y, Z Registers**

**Four formats** using for loading indirect using Y or Z (Z as example)

- **LD Rd, Z**  
  Z: Unchanged
- **LD Rd, Z+**  
  Z: Post increment
- **LD Rd, -Z**  
  Z: Pre decrement
- **LDD Rd, Z+q**  
  Z: unchanged

Rd can be any of R0-R31

q is from 0 to 63 (6-bit)

**LD X+ (LoaD indirect & post increment)**

- **LD Rd, X+** Load Indirect & Post-Inc.  
  \[ Rd \leftarrow (X), X \leftarrow X + 1 \]  
  2 clks

**Array Access**

**extern int A[], B[];**  
//A at FD20, B at FC60  
\[ A[0] = B[0]; \]

// **Short cut syntax**

First initialize X and Z registers:  
RegX=A, RegZ=B

1. **ldi r26, 0x20**  
   RegX = A
2. **ldi r27, 0xFD**  
   ; RegX = A
3. **ldi r30, 0x60**  
   ; RegZ = B
4. **ldi r31, 0xFC**  
   ; RegZ = B

Recall, array names are address constants

Note: lo8 and hi8 are gcc assembly macros

Then, load B[0] and store to A[0]

1. **ld r24, Z+**  
   r25:r24 = B[0]
2. **st X+, r24**  
   A[0] = r25:r24
3. **st X+, r25**  

The whole array can be copied if the code continues
Array Access

If we want to copy the arrays backwards, set up X and Z appropriately.

```assembly
#define N 100 ; assume array has 100 elems
ldi r26, lo8(A+2*N); RegX = &A[N]
ldi r27, hi8(A+2*N);
ldi r30, lo8(B+2*N); RegZ = &B[N]
ldi r31, hi8(B+2*N);

then do the following

ld r25, -Z; r25:r24 = B[N-1]
ld r24, -Z;

and repeat
```

Major Classes of Assembly Instructions

- **Data Movement**
  - Move data between registers
  - Move data in & out of SRAM
  - Different addressing modes

- **Logic & Arithmetic**
  - Addition, subtraction, etc.
  - AND, OR, bit shift, etc.

- **Control Flow**
  - Control which sections of code should be executed (e.g. In C "IF", "CASE", "WHILE", etc.
  - Typically the result of Logic & Arithmetic instructions help decided what path to take through the code.

Add without Carry

ADD: Add two registers without carry

**Syntax:** ADD Rd, Rr

**Operands:** 0 ≤ d ≤ 31, 0 ≤ r ≤ 31

**Operations:** Rd ← Rd+Rr, PC ← PC+1

**Binary Format**

```
0000
```

**SREG**

```
0000
```

Add with Carry

ADC: Add two registers with carry

**Syntax:** ADC Rd, Rr

**Operands:** 0 ≤ d ≤ 31, 0 ≤ r ≤ 31

**Operations:** Rd ← Rd+Rr+C, PC ← PC+1

**Binary Format**

```
0001
```

**SREG**

```
0000
```

Arithmetic Instruction

```c
int a, b;
...
a = a + b;
```
### Arithmetic Instruction

```
lds r18, a    ; load a
lds r19, a+1 ;
lds r24, b   ; load b
lds r25, b+1 ;
add r24, r18 ; add lower half
adc r25, r19 ; add higher half
sts a+1, r25 ; store a.byte1
sts a, r24   ; store a.byte0
```

### Subtract Immediate

#### SUBI: Subtract a register and a constant

**Syntax:** 
```
SUBI Rd, K
```

**Operands:** 
16 ≤ d ≤ 31, 0 ≤ K ≤ 255

**Operations:** 
Rd ← Rd-K, PC ← PC+1

**Binary Format**
```
0101   KKKK
       ddd1   KKKK
```

### Subtract Immediate with Carry

#### SBCI: Add two registers with carry

**Syntax:** 
```
SBCI Rd, K
```

**Operands:** 
16 ≤ d ≤ 31, 0 ≤ K ≤ 255

**Operations:** 
Rd ← Rd-K-C, PC ← PC+1

**Binary Format**
```
0100   KKKK
       ddd1   KKKK
```

### Arithmetic Instruction

```
char a;
...
a += 0x01;
```

**Challenge:** There are no "ADDI" and "ADIC"?

How to write the assembly code?
Logical AND

AND: Logical AND of two registers

Syntax: \texttt{AND \textit{Rd}, \textit{Rr}}

SREG

<table>
<thead>
<tr>
<th>I</th>
<th>T</th>
<th>H</th>
<th>S</th>
<th>V</th>
<th>N</th>
<th>Z</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Example:

and r2,r3 ; Bitwise and r2 and r3, result in r2
ldi r16,1 ; Set bitmask 0000 0001 in r16
and r2,r16 ; Isolate bit 0 in r2

Logical AND with Immediate

ANDI: Logical AND of a register and a constant

Syntax: \texttt{ANDI \textit{Rd}, \textit{K}} (16\leq r\leq 31, 0\leq K\leq 255)

SREG

<table>
<thead>
<tr>
<th>I</th>
<th>T</th>
<th>H</th>
<th>S</th>
<th>V</th>
<th>N</th>
<th>Z</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Example:

andi r17,$0F ; Clear upper nibble of r17
andi r18,$10 ; Isolate bit 4 in r18
andi r19,$AA ; Clear odd bits of r19

Multiply Unsigned

MUL: Multiply unsigned two registers

Syntax: \texttt{MUL \textit{Rd}, \textit{Rr}}

Operation: R1:R0 $\leftarrow$ Rd $\times$ Rr (unsigned)

SREG

<table>
<thead>
<tr>
<th>I</th>
<th>T</th>
<th>H</th>
<th>S</th>
<th>V</th>
<th>N</th>
<th>Z</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Example:

mul r5,r4 ; Multiply unsigned r5 and r4
movw r4,r0 ; Copy result back in r5:r4

Major Classes of Assembly Instructions

- **Data Movement**
  - Move data between registers
  - Move data in & out of SRAM
  - Different addressing modes
- **Logic & Arithmetic**
  - Addition, subtraction, etc.
  - AND, OR, bit shift, etc.
- **Control Flow**
  - Control which sections of code should be executed (e.g. In C “IF”, “CASE”, “WHILE”, etc.
  - Typically the result of Logic & Arithmetic instructions help decided what path to take through the code (i.e. they set flags)