Physics 120B: Lecture 11

Assembly Language and Arduino

Behind the C code (or sketch)

- C provides a somewhat human-readable interface
  - but it gets compiled into machine instruction set
  - ultimately just binary (or hex) instructions loaded into the ATMega program memory (flash)
  - even so, each instruction can be expressed in human terms
  - called "assembly language" or "machine code"
- Assembly instruction set is very low level
  - dealing with the processing of one data parcel (byte, usu.) at a time
  - a C command may break out into a handful of machine instructions

Viewing assembly produced by Arduino

- Look within the Arduino install directory:
  - On a Mac:
    - /Applications/Arduino.app/Contents/Resources/Java/

```
RXTXcomm.jar          lib/                  quaqua.jar
core.jar              libquaqua.jnilib      reference/
ecj.jar               libquaqua64.jnilib    revisions.txt
examples/             libraries/            tools/
hardware/             librxtxSerial.jnilib
jna.jar               pde.jar
```

  - we looked before in hardware/arduino/ for code details
  - in hardware/arduino/tools/avr/bin/ are some utilities

```
avarice*         avr-gcc*          avr-gprof*     avr-project*   ice-insight*
avr-addr2line*   avr-gcc-3.4.6*    avr-help*      avr-ranlib*    kill-avarice*
avr-ar*          avr-gcc-4.3.2*    avr-info*      avr-readelf*   libusb-config*
avr-as*          avr-gcc-select*   avr-ld*        avr-size*      make*
avr-c++*         avr-gccbug*       avr-man*       avr-strings*   simulavr*
avr-c++filt*     avr-gcov*         avr-nm*        avr-strip*     simulavr-disp*
avr-cpp*         avr-gdb*          avr-objcopy*   avrdude*       simulavr-vcd*
avr-g++*         avr-gdbtui*       avr-objdump*   ice-gdb*       start-avarice*
```


AVR, Dude?

- AVR is an 8-bit architecture developed by Atmel
  - http://en.wikipedia.org/wiki/Atmel_AVR
  - used by ATMega chips, on which Arduino is based
- Note in particular avr-objdump, avrdude
  - the latter mostly because it has a cool name (it can be used to shove machine code (.hex) onto chip)
    - DUDE means Downloader UploaDEr (a stretch)
- Running avr-objdump on .o or .elf files in your local Arduino/build/ directory disassembles code
  - the -d flag produces straight code
  - the -S flag intersperses with commented C-like code

Use .o or .elf?

- Can dump either stuff in the `.o` file or the `.elf` file
  - the `.o` file contains just the pieces you programmed
    - thus leaves out the code behind built-in functions
  - the `.elf` file contains the rest of the ATMega interface
  - so `.o` output will be smaller, but lack full context

Example: Simple Blink program

```
const int LED=13;

void setup()
{
  pinMode(LED,OUTPUT);
}

void loop()
{
  digitalWrite(LED,HIGH);
  delay(250);
  digitalWrite(LED,LOW);
  delay(500);
}
```

- Look how small it is, when written in high-level human terms!

Compiled, in build directory

- Compilation produces following in IDE message box:
  - Binary sketch size: 1,076 bytes (of a 30,720 byte maximum)
- Listing of build directory:

```
-rw-r--r--  1 tmurphy  tmurphy    239 Feb  3 08:42 simple_blink.cpp
-rw-r--r--  1 tmurphy  tmurphy   1062 Feb  3 08:42 simple_blink.cpp.d
-rw-r--r--  1 tmurphy  tmurphy     13 Feb  3 08:42 simple_blink.cpp.eep
-rwxr-xr-x  1 tmurphy  tmurphy  14061 Feb  3 08:42 simple_blink.cpp.elf*
-rw-r--r--  1 tmurphy  tmurphy   3049 Feb  3 08:42 simple_blink.cpp.hex
-rw-r--r--  1 tmurphy  tmurphy   3892 Feb  3 08:42 simple_blink.cpp.o
```

  - note file size in bytes
  - .d file is list of header files
  - .eep is about EEPROM data
  - .o and .elf are compiled
  - .hex is what is sent to chip
    - note that the ASCII representation is at least 2× larger than binary version (e.g., 9C takes 2 bytes to write in ASCII, 1 byte in memory)

simple_blink.cpp

- Basically what's in the sketch, with *some* wrapping

```
#include "Arduino.h"
void setup();
void loop();
const int LED=13;

void setup()
{
  pinMode(LED,OUTPUT);
}

void loop()
{
  digitalWrite(LED,HIGH);
  delay(250);
  digitalWrite(LED,LOW);
  delay(500);
}
```

`avr-objdump -d` on `.o` file

```
simple_blink.cpp.o:     file format elf32-avr


Disassembly of section .text.loop:
  pgm        hex             cmd      arguments          ; comments
00000000 <loop>:
   0:    8d e0           ldi     r24, 0x0D       ; 13
   2:    61 e0           ldi     r22, 0x01       ; 1
   4:    0e 94 00 00     call    0       ; 0x0 <loop>
   8:    6a ef           ldi     r22, 0xFA       ; 250
   a:    70 e0           ldi     r23, 0x00       ; 0
   c:    80 e0           ldi     r24, 0x00       ; 0
   e:    90 e0           ldi     r25, 0x00       ; 0
  10:    0e 94 00 00     call    0       ; 0x0 <loop>
  14:    8d e0           ldi     r24, 0x0D       ; 13
  16:    60 e0           ldi     r22, 0x00       ; 0
  18:    0e 94 00 00     call    0       ; 0x0 <loop>
```

- Just the start of the 32-line file
- Entries are:
  - program memory address; hex command; assembly command, arguments, comments

`avr-objdump -S` on `.o` file

```
00000000 <loop>:
  pinMode(LED,OUTPUT);
}

void loop()
{
  digitalWrite(LED,HIGH);
   0:    8d e0           ldi     r24, 0x0D       ; 13
   2:    61 e0           ldi     r22, 0x01       ; 1
   4:    0e 94 00 00     call    0       ; 0x0 <loop>
  delay(250);
   8:    6a ef           ldi     r22, 0xFA       ; 250
   a:    70 e0           ldi     r23, 0x00       ; 0
   c:    80 e0           ldi     r24, 0x00       ; 0
   e:    90 e0           ldi     r25, 0x00       ; 0
  10:    0e 94 00 00     call    0       ; 0x0 <loop>
  digitalWrite(LED,LOW);
  14:    8d e0           ldi     r24, 0x0D       ; 13
  16:    60 e0           ldi     r22, 0x00       ; 0
  18:    0e 94 00 00     call    0       ; 0x0 <loop>
```

- Now has C code interspersed; 49 lines in file
  - but does not make sense on its own; `call` references wrong

`avr-objdump -d` on `.elf` file

```
00000100 <loop>:
 100:   8d e0           ldi     r24, 0x0D       ; 13
 102:   61 e0           ldi     r22, 0x01       ; 1
 104:   0e 94 b5 01     call    0x36a   ; 0x36a <digitalWrite>
 108:   6a ef           ldi     r22, 0xFA       ; 250
 10a:   70 e0           ldi     r23, 0x00       ; 0
 10c:   80 e0           ldi     r24, 0x00       ; 0
 10e:   90 e0           ldi     r25, 0x00       ; 0
 110:   0e 94 e2 00     call    0x1c4   ; 0x1c4 <delay>
 114:   8d e0           ldi     r24, 0x0D       ; 13
 116:   60 e0           ldi     r22, 0x00       ; 0
 118:   0e 94 b5 01     call    0x36a   ; 0x36a <digitalWrite>
```

- Now loop starts at memory location (program counter) 100 (hex)
  - and calls to other routines no longer just address 0
  - note useful comments for writes and delays
  - note also extensive use of registers r22, r24, etc.

`avr-objdump -S` on `.elf` file

```
void loop()
{
  digitalWrite(LED,HIGH);
 100:   8d e0           ldi     r24, 0x0D       ; 13
 102:   61 e0           ldi     r22, 0x01       ; 1
 104:   0e 94 b5 01     call    0x36a   ; 0x36a <digitalWrite>
  delay(250);
 108:   6a ef           ldi     r22, 0xFA       ; 250
 10a:   70 e0           ldi     r23, 0x00       ; 0
 10c:   80 e0           ldi     r24, 0x00       ; 0
 10e:   90 e0           ldi     r25, 0x00       ; 0
 110:   0e 94 e2 00     call    0x1c4   ; 0x1c4 <delay>
  digitalWrite(LED,LOW);
 114:   8d e0           ldi     r24, 0x0D       ; 13
 116:   60 e0           ldi     r22, 0x00       ; 0
 118:   0e 94 b5 01     call    0x36a   ; 0x36a <digitalWrite>
  delay(500);
 11c:   64 ef           ldi     r22, 0xF4       ; 244
 11e:   71 e0           ldi     r23, 0x01       ; 1
```

- Embedded C code
  - note 500 delay is 1×256 + 244 (0x01F4)

A look at .hex file

```
:100100008DE061E00E94B5016AEF70E080E090E070
:100110000E94E2008DE060E00E94B50164EF71E0B2
:100120000080E090E00E94E20008958DE061E00E948E
```

- Snippet of ASCII .hex file around sections displayed on previous four slides
  - first: how many bytes in line (2 hex characters/byte)
  - next, program counter for 1st instr. in line: 0100, 0110, 0120
  - then 00, then, instructions, like: 8DE0, 61E0, 0E94B501
    - just contents of assembly, in hex terms
  - checksum at end

```
100:   8d e0          ldi    r24, 0x0D       ; 13
102:   61 e0          ldi    r22, 0x01       ; 1
104:   0e 94 b5 01    call   0x36a   ; 0x36a <digitalWrite>
108:   6a ef          ldi    r22, 0xFA       ; 250
10a:   70 e0          ldi    r23, 0x00       ; 0
10c:   80 e0          ldi    r24, 0x00       ; 0
10e:   90 e0          ldi    r25, 0x00       ; 0
110:   0e 94 e2 00    call   0x1c4   ; 0x1c4 <delay>
```


Counting bytes

- The end of the hex file looks like:

```
:10042000D0E00E9480002097E1F30E940000F9CF05
:04043000F894FFCF6E
:00000001FF
```

- And the corresponding assembly:

```
 42a:   0e 94 00 00     call    0       ; 0x0 <__vectors>
 42e:   f9 cf           rjmp    .-14            ; 0x422 <main+0x10>
00000430 <_exit>:
 430:   f8 94           cli
00000432 <__stop_program>:
 432:   ff cf           rjmp    .-2             ; 0x432 <__stop_program>
```

  - last 4 bytes on penultimate line; note 04 leader (4 bytes)
    - normal (full) line has 16 bytes (hex 0x10)
    - 67 full-size lines is 1072 bytes, plus four at end → 1076 bytes
    - Recall: Binary sketch size: 1,076 bytes (of a 30,720 byte maximum)
- Last line in hex file likely a standard ending sequence

Great, but what does it *mean*?

- We've seen some patterns, and seen assembly code
  - but what do we make of it?
- See Chapter 32 of ATMega datasheet, pp. 537–539
  - or http://en.wikipedia.org/wiki/Atmel_AVR_instruction_set
- But won't learn without a lot of effort
- Some examples:
  - in the copied code, we really only saw LDI and CALL

Mnemonics	Operands	Description	Operation	Flags	#Clocks
LDI	Rd, K	Load Immediate	Rd ← K	None	1
CALL[1]	k	Direct Subroutine Call	PC ← k	None	4

  - LDI puts contents of byte K (2nd arg.) into register Rd (1st arg.)
  - CALL loads K (only arg.) into PC (program counter)
    - so next operation takes place there; saves place for call origin
  - note info on how many clock cycles are taken

Inserting Assembly Code into C Sketch

- The Arduino interface provides a means to do this
  - via `asm()` command
- Can send digital values directly to port
- Why would you do this?
  - consider that `digitalWrite()` takes > 60 clock cycles
    - maybe you need faster action
    - maybe you need several pins to come on simultaneously
  - might need delays shorter than 1 μs
    - insert `nop` (no operation) commands, taking 1 cycle each
  - might need to squeeze code to fit into flash memory
    - direct low-level control without bells & whistles is more compact
- Why *wouldn't* you do this?
  - lose portability, harder to understand code, mistake prone

Direct Port Manipulation

- Can actually do this *without* going all the way to assembly language
  - see http://arduino.cc/en/Reference/PortManipulation
  - PORTD maps to pins 0–7 on Arduino
  - PORTB (0:5) maps to pins 8–13 on Arduino
  - PORTC (0:5) maps to analog pins 0–5
  - Each (D/B/C) has three registers to access; e.g., for port D:
    - DDRD: direction: 11010010 has pins 1, 4, 6, 7 as output
      - must keep pin 0 as input, pin 1 as output if Serial is used
    - PORTD: read/write values (can probe PORTD as well as set it)
    - PIND: read values (cannot set it)
  - So DDR replaces `pinMode()`
  - writing PORTD = B01010010 puts pins 6, 4, 1 HIGH at once

Example: Hard-coded Outputs

```
void setup()
{
  DDRD |= B00010010;
}

void loop()
{
  PORTD |= B00010000;
  delay(250);
  PORTD &= B11101111;
  delay(500);
}
```

- Serial-friendly, and sets pin 4 (D:4) as output
- Uses bitwise logic AND, OR, and NOT to set pin values
  - virtue of this is that it leaves other pin values undisturbed
- Sketch compiles to 676 bytes
  - compare to 1076 using Arduino commands

More Flexible Coding of Same

```
const int OUTBIT=4;

void setup()
{
  DDRD = B00000010 | (1 << OUTBIT);
}

void loop()
{
  PORTD |= (1 << OUTBIT);
  delay(250);
  PORTD &= ~(1 << OUTBIT);
  delay(500);
}
```

- Again sets port D to be Serial-friendly and pin 4 as output
- Still 676 bytes (no penalty for flexibility)
  - compiles to same actions, but now easier to modify
  - compiles to 474 bytes without delay functions
  - adding back `pinMode()` → 896 bytes
  - then restoring `digitalWrite()` → 1076 bytes

Resulting Assembly Code

0001 0010

```
DDRD = B00000010 | (1 << OUTPIN);
a6:    82 e1          ldi     r24, 0x12       ; 18
a8:    8a b9          out     0x0a, r24       ; 10

PORTD |= (1 << OUTPIN);
ac:    5c 9a          sbi     0x0b, 4 ; 11

PORTD &= ~(1 << OUTPIN);
ba:    5c 98          cbi     0x0b, 4 ; 11
```

- Tiny commands
  - load (LDI) B00010010 (0x12) into r24 (register 24)
  - write r24 out (OUT) to port 0x0a (see ATMega register summary)
  - set 4th bit (SBI) of register 0x0b (write HIGH to that pin)
  - clear 4th bit (CBI) of register 0x0b (write LOW to that pin)

What's with addresses 0x0a and 0x0b?

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
0x0B (0x2B)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	95
0x0A (0x2A)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	95
0x09 (0x29)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	95
0x08 (0x28)	PORTC	–	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0	94
0x07 (0x27)	DDRC	–	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0	94
0x06 (0x26)	PINC	–	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0	94
0x05 (0x25)	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	94
0x04 (0x24)	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	94
0x03 (0x23)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	94

- From the ATMega short datasheet
  - we see 0x0a is DDRD
  - and 0x0b is PORTD
  - 0x09 is PIND, if anyone cares (Port D input pin address)
- And the commands used in previous clip…

Mnemonics	Operands	Description	Operation	Flags	#Clocks
LDI	Rd, K	Load Immediate	Rd ← K	None	1
OUT	P, Rr	Out Port	P ← Rr	None	1
SBI	P,b	Set Bit in I/O Register	I/O(P,b) ← 1	None	2
CBI	P,b	Clear Bit in I/O Register	I/O(P,b) ← 0	None	2

Direct Assembly in Sketch

```
void setup()
{
  asm("ldi\tr24, 0x12\n\t" "out\t0x0a, r24\n\t");
  // could replace with asm("sbi\t0x0a,4\n\t");
}

void loop()
{
  asm("sbi\t0x0b, 4\n\t");
  delay(250);
  asm("cbi\t0x0b, 4\n\t");
  delay(500);
}
```

- Use if you're really feeling black-belt…
  - note use of tabs (`\t`), and each instruction ending (`\n\t`)
  - can gang several instructions into same `asm()` command
  - no advantage in this program over PORTD approach (in fact, far less intelligible), but illustrates method (and actually works!)

Packing command into hex

```
a6:   82 e1          ldi     r24, 0x12       ; 18
a8:   8a b9          out     0x0a, r24       ; 10
ac:   5c 9a          sbi     0x0b, 4         ; 11
ba:   5c 98          cbi     0x0b, 4         ; 11
```

- The human-readable form gets packed into hex code
- Prescription varies by command, found in instruction set reference (link from course website); for LDI:

Operation:

(i) Rd ← K

	Syntax:	**Operands:**	**Program Counter:**
(i)	LDI Rd,K	$16 \le d \le 31, 0 \le K \le 255$	PC ← PC + 1

16-bit Opcode:

1110	KKKK	dddd	KKKK

  - r24 → d = 24, which is 8 off minimum of 16, so dddd → 1000
  - K = 0x12 = 0001 0010
  - 1110 0001 1000 0010 = E 1 8 2 → 82 E1, as in line `a6` above

More Examples

```
a8:   8a b9          out     0x0a, r24       ; 10
```

Operation:

(i) I/O(A) ← Rr

	Syntax:	**Operands:**	**Program Counter:**
(i)	OUT A,Rr	$0 \le r \le 31, 0 \le A \le 63$	PC ← PC + 1

16-bit Opcode:

1011	1AAr	rrrr	AAAA

- OUT command
  - r = 24 = 0x18 = 0001 1000, or 1 1000 split to r rrrr
  - A = 0x0a = 0000 1010, or 00 1010 split to AA AAAA
  - so get 1011 1001 1000 1010 = B 9 8 A → 8A B9

One More Example

```
ac:    5c 9a             sbi      0x0b, 4          ; 11
```

Operation:

(i) I/O(A,b) ← 1

	Syntax:	Operands:	Program Counter:
(i)	SBI A,b	$0 \le A \le 31$, $0 \le b \le 7$	PC ← PC + 1

16-bit Opcode:

1001	1010	AAAA	Abbb

- SBI command
  - A = 0x0b = 0000 1011 → 0101 1 when split to AAAA A
  - b = 4 = 100
  - so have 1001 1010 0101 1100 = 9 A 5 C → 5C 9A

Language Reference

Mnemonics	Operands	Description	Operation	Flags	#Clocks	#Clocks XMEGA
		Arithmetic and Logic Instructions				
ADD	Rd, Rr	Add without Carry	Rd ← Rd + Rr	Z,C,N,V,S,H	1	
ADC	Rd, Rr	Add with Carry	Rd ← Rd + Rr + C	Z,C,N,V,S,H	1	
ADIW(1)	Rd, K	Add Immediate to Word	Rd ← Rd + 1:Rd + K	Z,C,N,V,S	2	
SUB	Rd, Rr	Subtract without Carry	Rd ← Rd - Rr	Z,C,N,V,S,H	1	
SUBI	Rd, K	Subtract Immediate	Rd ← Rd - K	Z,C,N,V,S,H	1	
SBC	Rd, Rr	Subtract with Carry	Rd ← Rd - Rr - C	Z,C,N,V,S,H	1	
SBCI	Rd, K	Subtract Immediate with Carry	Rd ← Rd - K - C	Z,C,N,V,S,H	1	
SBIW(1)	Rd, K	Subtract Immediate from Word	Rd + 1:Rd ← Rd + 1:Rd - K	Z,C,N,V,S	2	
AND	Rd, Rr	Logical AND	Rd ← Rd • Rr	Z,N,V,S	1	

- First portion of 3 page instruction set (119 cmds.)
  - 29 arithmetic and logic; 38 branch; 20 data transfer; 28 bit and bit-test; 4 MCU control
- Flags store results from operation, like:
  - was result zero (Z)?, was there a carry (C)?, result negative (N)?, and more

Example from Instruction Reference

ADC – Add with Carry

Description:

Adds two registers and the contents of the C Flag and places the result in the destination register Rd.

	Operation:		
(i)	Rd ← Rd + Rr + C		

	Syntax:	Operands:	Program Counter:
(i)	ADC Rd,Rr	0 ≤ d ≤ 31, 0 ≤ r ≤ 31	PC ← PC + 1

16-bit Opcode:

0001	11rd	dddd	rrrr

Status Register (SREG) Boolean Formula:

I	T	H	S	V	N	Z	C
–	–	⇔	⇔	⇔	⇔	⇔	⇔

First half of page for add with carry
Note use of C status bit

ADC, Continued

H: $Rd3 \bullet Rr3 + Rr3 \bullet \overline{R3} + \overline{R3} \bullet Rd3$
Set if there was a carry from bit 3; cleared otherwise

S: $N \oplus V$, For signed tests.

V: $Rd7 \bullet Rr7 \bullet \overline{R7} + \overline{Rd7} \bullet \overline{Rr7} \bullet R7$
Set if two's complement overflow resulted from the operation; cleared otherwise.

N: R7
Set if MSB of the result is set; cleared otherwise.

Z: $\overline{R7} \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0}$
Set if the result is $00; cleared otherwise.

C: $Rd7 \bullet Rr7 + Rr7 \bullet \overline{R7} + \overline{R7} \bullet Rd7$
Set if there was carry from the MSB of the result; cleared otherwise.

R (Result) equals Rd after the operation.

Example:

```
                ; Add R1:R0 to R3:R2
    add   r2,r0 ; Add low byte
    adc   r3,r1 ; Add with carry high byte
```

Words: 1 (2 bytes)
Cycles: 1

Example code: delay function

```
delay(2000);
  ac:   60 ed          ldi    r22, 0xD0      ; 208
  ae:   77 e0          ldi    r23, 0x07      ; 7
  b0:   80 e0          ldi    r24, 0x00      ; 0
  b2:   90 e0          ldi    r25, 0x00      ; 0
  b4:   0e 94 ac 00    call   0x158   ; 0x158 <delay>
```

- Want to wait for 2000 ms
- Load registers 22..25 with 2000
  - 0×2^{24} 0×2^{16} 7×2^{8} $208\times2^{0} = 2000$
- Call program memory location 0x158
  - first store address of next instruction (0xb8) in STACK
  - set program counter (PC) to 0x158
  - next instruction will be at program address 0x158
  - return from routine will hit program at location 0xb8

Delay Function

- Has 81 lines of assembly code
  - many instructions repeated in loops
  - uses commands MOVW, IN, CLI, LDS, SBIS, RJMP, CPI, BREQ, ADDIW, ADC, MOV, EOR, ADD, LDI, BRNE, SUB, SBC, SUBI, SBCI, BRCS, CP, CPC, RET
  - essentially loads a counter with how many milliseconds
  - and another counter with 1000
  - rifles through a microsecond (16 clock cycles), decrementing microsecond counter (down from 1000)
  - when 1k counter reaches zero, 1 ms elapsed, decrement ms counter
  - after each decrement, check if zero and return if so

Announcements

- Project proposals due Friday, 2/07
- Tracker check-off, turn in code by 2/11 or 2/12
- Will move to new lab schedule next week
  - fill out Doodle poll if you haven't and want a say
  - partners can both fill out poll, so not underrepresented
- Lectures will terminate after this week
- Let's plan "midterm" for Wednesday, 2/19
  - will give example of some simple task you are to do in Arduino, and you write down C-code on blank paper that would successfully compile and perform the desired task