18-100 Lecture 19:
Intro to AVR Assembly Programming

James C. Hoe
Dept of ECE, CMU
March 26, 2015

Today's Goal: Get ready for Lab 9
Announcements: HW#7 due today
Midterm 2 next Tuesday!!
Handouts: Lab 9 (on Blackboard)
Atmel 8-bit AVR ATmega8 Databook (on Blackboard)
Atmel 8-bit AVR Instruction Set Manual (on Blackboard)

Computer System Abstraction Layers

18-290 18-240 18-220

- Applications
- Compilers
- OS
- Architecture (ISA)
- Microarchitecture
- Digital Design
- Circuits
- Devices/Physics

users and problems
prog. languages
resource virtualization
hw/sw interface
datapath
registers, ALU
digital logic
transistors, signals
atoms, electrons

To use an abstraction properly you must understand the limits of the abstraction
What is a Computer?

- **Computer, 2. a.** A calculating-machine; esp. an automatic electronic device for performing mathematical or logical operations; freq. with defining word prefixed, as analogue, digital, electronic computer.

--- Oxford English Dictionary

So what makes a computer a computer?

- **Processing**
- **Storage** (program and data)
- **I/O**

The fact that programs are stored in memory like data is very important.
ENIAC: “first” electronic digital computer
(Eckert and Mauchly, 1946)

- 18,000 vacuum tubes
- 30 ton, 80 by 8.5 feet
- 1900 additions per second
- 20 10-decimal-digit words
  (100-word core by 1952)
- programmed by 3000
  switches in the function
  table and plug-cables
  (became stored program in
  1948)

Your Computer: Atmel ATmega8

- ~$3.00 each
  - may be ~10K gates
  - clock up to 16MHz
  - 1KB Data SRAM (8-bit words)
  - 8KB Program Memory (Flash)

- BTW, a modern high-end CPU
  (e.g., Intel Xeon)
  - billions of transistors (10+
    cores)
  - many GHz (approaching 100
    GFLOPs/sec)
  - 10s of MB in just caches
Atmel ATmega8

State

I/O

Logic

Data Bus 8-bit

Flash Program Memory

Instruction Register

Instruction Decoder

Control Lines

Direct Addressing

Indirect Addressing

32 x 8 General Purpose Registers

ALU

Interrupt Unit

SPI Unit

Watchdog Timer

Analog Comparator

I/O Module 1

I/O Module 2

I/O Module n

Data RAM

EEPROM

I/O Lines

Page 9 Atmel 8-bit AVR ATmega8 Databook

Seeing the Big Picture

[images from Wikipedia]
Stored Program Architecture
[Burks, Goldstein, von Neumann, 1946]

◆ By far the most common architectural paradigm
◆ Memory holds both program and data
  - instructions and data in a linear memory array
  - instructions can be modified just like data
◆ Sequential instruction processing
  1. program counter (PC) identifies the current instruction
  2. instruction is fetched from memory
  3. instruction execution causes some state (e.g. memory) to be updated as a specific function of current state
  4. program counter is advanced (according to instruction)
  5. repeat

An Instruction Set Architecture

◆ Abstracting a processor/computer as
  - program visible state
    • memory, registers, program counters, etc.
  - set of instructions to modified state; each prescribes
    • which state elements are read as operands
    • which state elements are updated and to what new values
    • where is the next instruction
◆ Other details
  - instruction-to-binary encoding
  - data format and size
  - how to interface with the outside world?
  - protection and privileged operations
  - software conventions
**AVR** Program Visible State
(ones we care about for now)

<table>
<thead>
<tr>
<th>Program Memory</th>
<th>Data Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000</td>
<td>0x000</td>
</tr>
<tr>
<td>0x001</td>
<td>0x001</td>
</tr>
<tr>
<td>0x002</td>
<td>0xFFD</td>
</tr>
<tr>
<td>0x003</td>
<td>0xFFE</td>
</tr>
<tr>
<td>0xFFF</td>
<td>0xFFF</td>
</tr>
</tbody>
</table>

-4K (2^{12}) 16-bit words
- each instruction either 1 or 2 words

16-bit PC
8-b status

Register File
R0
R1
R2
R29
R30
R31

32 8-bit words

**AVR Instruction Example: ADD**

ADD – Add without Carry

**Description:**
A prose description of what ADD does

- Adds two registers without the C flag and places the result in the destination register Rd.

**Operation:**

(i) \( \text{Rd} \leftarrow \text{Rd} + \text{Rr} \)

**Syntax:**

ADD Rd,Rr

**Operands:**

\( 0 \leq d < 31, 0 \leq r < 31 \)

**Program Counter:**

PC \( \leftarrow \) PC + 1

**16-bit Opcode:**

| 0000 | 11rd | dddd | rrrr |

Note:

- 2 input 1 output function, but Rd is used as both src and dest
- what is this “carry”?

- “ADD” = 000011 in bit[15:10]
- d = bit[8], bit[7:4]
- r = bit[9], bit[3:0]
Other ALU Instructions

<table>
<thead>
<tr>
<th>Mnemonics</th>
<th>Operands</th>
<th>Description</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>R1, R2</td>
<td>Add two Registers</td>
<td>R1 = R1 + R2</td>
</tr>
<tr>
<td>ADC</td>
<td>R1, R2</td>
<td>Add with Carry two Registers</td>
<td>R1 = R1 + R2 + C</td>
</tr>
<tr>
<td>ADDW</td>
<td>R1, R2</td>
<td>Add Immediate to Word</td>
<td>R1 = R1 + R2</td>
</tr>
<tr>
<td>SUB</td>
<td>R1, R2</td>
<td>Subtract two Registers</td>
<td>R1 = R1 - R2</td>
</tr>
<tr>
<td>SUBC</td>
<td>R1, R2</td>
<td>Subtract with Carry two Registers</td>
<td>R1 = R1 - R2 - C</td>
</tr>
<tr>
<td>SBCH</td>
<td>R1, R2</td>
<td>Subtract with Carry from Reg</td>
<td>R1 = R1 - R2 - C</td>
</tr>
<tr>
<td>SUBW</td>
<td>R1, R2</td>
<td>Subtract Immediate from Word</td>
<td>R1 = R1 - R2 - R</td>
</tr>
<tr>
<td>AND</td>
<td>R1, R2</td>
<td>Logical AND Registers</td>
<td>R1 = R1 &amp; R2</td>
</tr>
<tr>
<td>ANDI</td>
<td>R1, R2</td>
<td>Logical AND Register and Constant</td>
<td>R1 = R1 &amp; R2</td>
</tr>
<tr>
<td>OR</td>
<td>R1, R2</td>
<td>Logical OR Registers</td>
<td>R1 = R1</td>
</tr>
<tr>
<td>ORI</td>
<td>R1, R2</td>
<td>Logical OR Register and Constant</td>
<td>R1 = R1</td>
</tr>
<tr>
<td>XOR</td>
<td>R1, R2</td>
<td>Exclusive OR Registers</td>
<td>R1 = R1 ^ R2</td>
</tr>
<tr>
<td>XORI</td>
<td>R1, R2</td>
<td>Exclusive OR Register and Constant</td>
<td>R1 = R1 ^ R2</td>
</tr>
<tr>
<td>COM</td>
<td>R1</td>
<td>One’s Complement</td>
<td>R1 = 0xFF - R1</td>
</tr>
<tr>
<td>NEG</td>
<td>R1</td>
<td>Negate in Register</td>
<td>R1 = -R1</td>
</tr>
<tr>
<td>CBR</td>
<td>R1</td>
<td>Clear Carry in Register</td>
<td>R1 = R1 &amp; 0x0</td>
</tr>
<tr>
<td>INC</td>
<td>R1</td>
<td>Increment</td>
<td>R1 = R1 + 1</td>
</tr>
<tr>
<td>DEC</td>
<td>R1</td>
<td>Decrement</td>
<td>R1 = R1 - 1</td>
</tr>
<tr>
<td>TST</td>
<td>R1</td>
<td>Test for Zero or Minus</td>
<td>R1 = R1 &amp; R6</td>
</tr>
<tr>
<td>CLR</td>
<td>R1</td>
<td>Clear Register</td>
<td>R1 = R1 &amp; 0x0</td>
</tr>
<tr>
<td>SBR</td>
<td>R1</td>
<td>Set Register</td>
<td>R1 = 0xFF</td>
</tr>
<tr>
<td>NUL</td>
<td>R1, R2</td>
<td>Multiply (unsigned)</td>
<td>R1: R2 = R1 * R2</td>
</tr>
<tr>
<td>NUL.S</td>
<td>R1, R2</td>
<td>Multiply Signed</td>
<td>R1: R2 = R1 * R2</td>
</tr>
<tr>
<td>NULSU</td>
<td>R1, R2</td>
<td>Multiply Signed with Signed</td>
<td>R1: R2 = R1 * R2</td>
</tr>
<tr>
<td>PMUL</td>
<td>R1, R2</td>
<td>Functional Multiply (unsigned)</td>
<td>R1: R2 = R1 * R2</td>
</tr>
<tr>
<td>PMUL.S</td>
<td>R1, R2</td>
<td>Functional Multiply Signed</td>
<td>R1: R2 = R1 * R2</td>
</tr>
<tr>
<td>PMULSU</td>
<td>R1, R2</td>
<td>Functional Multiply Signed with Signed</td>
<td>R1: R2 = R1 * R2</td>
</tr>
</tbody>
</table>

Assembly Programming 101

- Break down high-level program constructs into a sequence of elemental operations

- E.g. High-level Code

  \[ f = (g + h) - (i + j) \]

- Assembly Code
  - suppose \( g, h, i, j \) are in \( r15, r16, r17, r18 \) and do not need to be preserved
    
    \[
    \begin{align*}
    \text{add } & r15, r16 ; r15 = g+h \\
    \text{add } & r17, r18 ; r17 = i+j \\
    \text{sub } & r15, r17 ; r15 = f
    \end{align*}
    \]

  What if we do want to preserve \( r15 \sim r18 \)?
General Instruction Classes

- Arithmetic and logical operations
  - fetch operands from specified locations
  - compute a result as a function of the operands
  - store result to a specified location
  - update PC to the next sequential instruction
- Data movement operations
  - fetch operands from specified locations
  - store operand values to specified locations
  - update PC to the next sequential instruction
- Control flow operations
  - fetch operands from specified locations
  - compute a branch condition and a target address
  - if “branch condition is true” then $PC \leftarrow$ target address
  - else $PC \leftarrow$ next seq. instruction

Move “Immediate” to Register

LDI – Load Immediate

Description:
Loads an 8 bit constant directly to register 18 to 31.

Operation:
(i) $Rd \leftarrow K$

Syntax:
LDI Rd, K

Operands: $16 \leq d \leq 31, 0 \leq K \leq 255$

Program Counter:
$PC \leftarrow PC + 1$

16-bit Opcode:

Note:
$Rd$ can only be r16~r31 because in order to give you an 8-bit immediate, there are only 4 bits left to specify $Rd$. 
Move Register to Register (Copy)

MOV – Copy Register

Description:
The instruction makes a copy of one register into another. The source register Rr is left unchanged, while the destination register Rd is loaded with a copy of Rr.

Operation:

(i) \( Rd \leftarrow Rr \)

Syntax: Operands: Program Counter:

(i) \( MOV \ Rd,Rr \) \( 0 \leq d \leq 31, \ 0 \leq r \leq 31 \) \( PC \leftarrow PC + 1 \)

16-bit Opcode:

\[ \text{xxxx} \]

We wait until next time to see “load” (i.e., move memory to register) and “store” (i.e., move register to memory)

Assembly Programming 102

- Break down high-level program constructs into a sequence of elemental operations

- E.g. High-level Code

\[ f = ( g + h ) - ( i + j ) \]

- Assembly Code

- suppose \( g, h, i, j \) are in r15, r16, r17, r18 and should be preserved; put result \( f \) in r19; assume r20 is “free”
Control Flow Instructions

Control Flow Graph:

- C-Code

```c
{ code block A }
if X==Y then
    { code block B }
else
    { code block C }
{ code block D }
```

Assembly Code (linearized):

- `if X==Y goto code B`
- `goto code C`
- `code D`

These things are called basic blocks.

Control Flow: Jump!

RJMP – Relative Jump

Description:
Relative jump to an address within PC - 2K + 1 and PC + 2K (words). For AVR microcontrollers with Program memory not exceeding 4K words (8K bytes) this instruction can address the entire memory from every address location. See also JMP.

- **Operation:**
  - PC ← PC + k + 1

- **Syntax:**
  - RJMP k
- **Operands:**
  - -2K ≤ k < 2K
- **Program Counter:**
  - PC ← PC + k + 1

16-bit Opcode:

```
1100  kkkk  kkkk  kkkk
```

Note: Jump target is specified as an offset from PC+1, but, fortunately, in assembly programs, you can specify the label of an “absolute” target instruction and the assembler will figure out the offset. (example later)
That is enough for Lab 9

I/O

- In Lab 9, you will tie PINB (as input) to dip-switches and PORTD (as output) to LEDs
- The instruction “in Rx, PINB” will load the value at PINB into Rx
- “out PORTD, Rx” will copy the contents of Rx to output PORTD (and hold)

- They are the only I/O operations you need know about
- Please do not feel free to experiment . . . .

When you fiddle with I/O, it is no longer an abstraction. Very real bad things can happen!!
Lab 9 Starter Code

```
.equ PINB=0x03
.equ DDRB=0x04
.equ PORTB=0x05
.equ PIND=0x09
.equ DDRD=0x0a
.equ PORTD=0x0b

.org 0x0000
entry:
    ldi r16,0xFF
    out DDRD,r16
    ldi r16,0x00
    out DDRB,r16
    ldi r16,0xff
    out PORTB,r16

main:
    in r16,PINB
    mov r17,r16
    andi r16,0x0f
    lsr r17
    lsr r17
    lsr r17
    add r16,r17
    out PORTD,r16
    rjmp main
```

DO NOT change the above!!

- figure out what the example does
- try it out for real on the board
- modify “compute stuff” to do what Lab 9 asks for
- demo your program on the board

---

Now back to the regularly scheduled program
Control Flow: Branch?

**BREQ – Branch if Equal**

**Description:**
Conditional relative branch. Tests the Zero Flag (Z) and branches relatively to PC if Z is set.
This instruction branches relatively to PC in either direction (PC - 63 ≤ destination ≤ PC + 64).
The parameter k is the offset from PC and is represented in two’s complement form.

\[(Z = 1) \text{ then } PC \leftarrow PC + k + 1, \text{ else } PC \leftarrow PC + 1\]

**Syntax:**
(i) BREQ k

**Operands:**
-64 ≤ k ≤ +63

**Program Counter:**
- PC ← PC + k + 1
- PC ← PC + 1, if condition is false

**16-bit Opcode:**

| 1111 | 00kk | kkkk | k001 |

**Note:**
- Like in RJMP, a branch target is also PC-relative
- (Z=1) is the branch condition. What the heck is Z?

---

**Status Register**

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>T</td>
<td>H</td>
<td>S</td>
<td>V</td>
<td>N</td>
<td>Z</td>
<td>C</td>
</tr>
</tbody>
</table>

Page 29 Atmel 8-bit AVR Instruction Set Manual

don’t worry about 7~4

- ‘V’, ‘N’, ‘Z’, ‘C’ are arithmetic flags automatically updated after each ALU-class instructions
  - Z: set if the last result was zero,
  - N: set if the last result was negative (2’s complement)
  - V: set if the last op caused an overflow (2’s comp)
  - C: set if the last op caused a carry (unsigned)

- Each has corresponding branch instructions
  - BREQ/BRNE, BRLRMI/BRPL, BRVS/BRVC, BRCS/BRCC

- E.g.,
  - after “SUB Rx, Ry” or “CP Rx, Ry”, Z is set if Rx==Ry
  - BREQ taken if Rx==Ry, BRNE taken if Rx!=Ry

Page 11 Atmel 8-bit AVR ATmega8 Databook
Assembly Programming 201

- E.g. High-level Code
  ```c
  if (i == j) then
    e = g
  else
    e = h
  f = e
  ```

- Assembly Code
  - suppose e, f, g, h, i, j are in r_e, r_f, r_g, r_h, r_i, r_j
  ```assembly
  cp r_i, r_j ; set status flags
  brne L1 ; if i!=j skip to L1 (else)
  mov r_e, r_g ; e gets g
  rjmp L2 ; skip to L2 (join)
  L1: mov r_e, r_h ; e gets h
  L2: mov r_f, r_e ; f gets e
  ```

Assembly Programming 202

- E.g. High-level Code
  ```c
  i=0; j=10;
  while (i != j) {
    i++;
  }
  ```

- Give it try. Pick your own free registers
Useful ALU Instructions

- ADD Rd, Rr — Add registers  \( Rd \leftarrow Rd + Rr \)
- ADC Rd, Rr — Add registers w. carry  \( Rd \leftarrow Rd + Rr + C \)
- SUB Rd, Rr — Subtract registers  \( Rd \leftarrow Rd - Rr \)
- AND Rd, Rr — AND registers  \( Rd \leftarrow Rd \cdot Rr \)
- OR Rd, Rr — OR registers  \( Rd \leftarrow Rd | Rr \)
- INC Rd — Increment register  \( Rd \leftarrow Rd + 1 \)
- DEC Rd — Decrement register  \( Rd \leftarrow Rd - 1 \)
- LSL Rd — Left shift register  \( Rd \leftarrow Rd \ll 1 \)
- LSR Rd — Right shift register  \( Rd \leftarrow Rd \gg 1 \)
- ASR Rd — Right shift register (sign-extend)  \( Rd \leftarrow Rd \gg 1 \)
- ADIW Rd, k — 16-bit add register-immediate  \( R(d+1):Rd = R(d+1):Rd + k \)

Useful Data Movement Instructions

- LDI Rd,K — Load Immediate  \( Rd \leftarrow K \)
- LDS Rd,k — Load from SRAM  \( Rd \leftarrow (k) \)
- LD Rd,X — Load register indirect  \( Rd \leftarrow (X) \)
- STS k,Rr — Store data to SRAM  \( (k) \leftarrow Rr \)
- ST X,Rr — Store register indirect  \( (X) \leftarrow Rr \)
- IN Rd,P — Read from port  \( Rd \leftarrow P \)
- OUT P,Rr — Write to port  \( P \leftarrow Rr \)
Useful Control Flow Instructions

- **RJMP** \( k \) — Jump to \( k \), where \( k \) is a memory address (label)
- **CP** \( Rd,Rr \) — Subtract \( Rd \) by \( Rr \) and set status flag but does not update \( Rd \)
- **BREQ** \( k \) — Branch to \( k \) if Z is set
  (branch if \( Rd=\)\( Rr \) following \( CP \) \( Rd \), \( Rr \))
- **BRNE** \( k \) — Branch to \( k \) if Z is clear
  (branch if \( Rd\neq Rr \) following \( CP \) \( Rd \), \( Rr \))
- **BRMI** \( k \) — Branch to \( k \) if S is set
  (branch if \( Rd<\)\( Rr \) following \( CP \) \( Rd \), \( Rr \))
- **BRPL** \( k \) — Branch to \( k \) if S is clear
  (branch if \( Rd\geq Rr \) following \( CP \) \( Rd \), \( Rr \))