ATMEGA128 Architecture and Assembly Programming Intro

Instructors:
Dr. Phillip Jones
Announcements

• Exam 2 on April 7 (Thursday next week)
• This is the 2\textsuperscript{nd} week for lab 9
• Project starts next week, join a project team by this week
AVR ARCHITECTURE OVERVIEW
Why use assembly programming?

• Full access to hardware features
  – Compiler limits a programmers access to the hardware features that the compiler writer decided to implement

• Writing time critical portions of code
  – Allows tight control over what the CPU is doing on every clock cycle

• Debugging
  – It in not uncommon when trying to debug odd system behavior to have to look at disassembled code

Refs: Beginners Introduction to the Assembly Language of ATMEL-AVR-Microprocessors (Gerhard Schmidt)
Why learn the ATMEGA128 Hardware Architecture?

- Helps give intuition to why the assembly instructions were created the way they were
- Help understand what special feature may be available for you to make use of.
ATmega128 Architecture Overview

• 8 bit processor
  – size of bus is 8 bits
  – size of registers is 8 bits
• RISC architecture
• Harvard architecture
  – separate data and instruction memory
• 133 instructions
ATmega128 Block Diagram

CPU

Flash Program Memory

Instruction Register

Instruction Decoder

Control Lines

Program Counter

Status and Control

32 x 8 General Purpose Registers

ALU

Data Bus 8-bit

Interrupt Unit

SPI Unit

Watchdog Timer

Analog Comparator

I/O Module 1

I/O Module 2

I/O Module n

Data SRAM

EEPROM

I/O Lines

Direct Addressing

Indirect Addressing

http://class.ece.iastate.edu/cpre288
ATMEGA128: RISC CPU Architecture

• What is RISC?
  – Reduced Instruction Set Computing (with respect to CISC, Complex Instruction set Computing)

• Typical RISC
  – LD/Store based: ALU to memory transaction via registers
  – Most instruction are the same length
  – Typically many less instructions than a CISC architecture
  – Typically many more registers than CISC since Data must be moved into a register before it can be operated on
  – Low number of instruction typically makes hardware design simpler (as compared to CISC)

Ref: [http://www.seas.upenn.edu/~palsetia/cit595s07/RISCvsCISC.pdf](http://www.seas.upenn.edu/~palsetia/cit595s07/RISCvsCISC.pdf) (Diana Palsetia)
ATMEGA128: RISC vs CISC example

<table>
<thead>
<tr>
<th>Size / time</th>
<th>CISC</th>
<th>RISC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 byte, 1clk</td>
<td>mov R1, 10</td>
<td>mov R1, 0</td>
</tr>
<tr>
<td>1 byte, 1clk</td>
<td>mov R2 5</td>
<td>mov R2, 10</td>
</tr>
<tr>
<td>4 byte, 30 clk</td>
<td>mul R2, R1</td>
<td>mov R3, 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Begin: add R1, R2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>loop Begin</td>
</tr>
</tbody>
</table>

- **CISC**: Instructions often variable length and variable time
- **RISC**: Instruction typically constant length and time
  - Almost all ATMEGA128 instructs take 1 clk (a few take 2 clks)
  - Simpler hardware logic for decoding instructions (thus typically faster)

*Note: Many current RISC architectures do have a multiply (mul) instruction*
  - ATMGA128: 2 clock cycles for integer multiply

Ref: [http://www.seas.upenn.edu/~palsetia/cit595s07/RISCvsCISC.pdf](http://www.seas.upenn.edu/~palsetia/cit595s07/RISCvsCISC.pdf) (Diana Palsetia)
Most instructions are 16-bit or 32-bit
  – Takes one or two cycles to fetch

Simple two-stage pipeline
  – Most instructions take one or two cycles

Registers are 8-bit and addresses are 16-bit
• Program memory
  – Flash based: Program stays even if power turned off (non-volatile)
  – 16-bits wide, instructions are 16-bit (typical) or 32-bit wide.

• Data Memory
  – SRAM based: Data disappears if power is turned off (volatile)
  – 8-bits wide: all data and registers are stored as 8-bit chunks.
• Registers:
  – 8-bits wide
  – Directly accessible by ALU

• Data Memory:
  – 8-bit wide
  – Must use a register to move to from the ALU
ATMEGA128: Memory Map organization

- Address layout:
  - First 32 rows (0 – 0x1F) are general registers
  - Next 64 rows (0x20-0x5F) are I/O registers
  - Next 160 rows (0x60-0xFF) are Extend I/O registers
  - Next 4096 rows (0x0100 – 0x10FF ) are Internal SRAM

http://class.ece.iastate.edu/cpre288
• 32 8-bit general purpose registers
  – Used for accessing SRAM
  – Used for storing function parameters
  – Used for instructions to execute operations on

• What is an 8-bit register.
  – Basically just 8 D-Flips connected together
• Register pairs R27:R26, R29:R28, R31:R30 can be used as 16-bit pointers (short versions of these registers are X, Y, Z)

• R16 – R31 may be used with 8-bit immediate values (e.g. LDI R17, 5)

• R24 – R31 may be used as 16-bit register pairs with 8-bit immediate values (e.g. ADIW R24, 10)
ATMEGA128: GP Registers

• 16-bit Datum stored across adjacent registers
  – Used for accessing SRAM
  – Used for storing function parameters
  – Used for instructions to execute operations on
• 32-bit stored across adjacent registers
STATUS REGISTER (SREG)
Describes the status of the CPU

**I: Global Interrupt Enable**, enable/disable interrupts to the CPU

**T: Bit Copy Storage**, for moving a single bit between registers

**H: Half Carry Flag**, To indicate a half carry, useful in BCD arithmetic
**S: Sign Bit**, Whether the *actual* result is negative or not with *signed* type operation

**V: Two’s Complement Overflow Flag**, whether a overflow happened or not with *signed operands*

**N: Negative Flag**, whether the result is negative or not with *signed operands*
Z: Zero flag, whether the result is zero or not
C: Carry Flag, whether a carry is generated for unsigned operands
H, S, N, V, Z, C bits are regarding the last arithmetic/logic operation
Designer’s Perspective

H: \( Rd3 \cdot Rr3 + Rr3 \cdot R3 + R3 \cdot Rd3 \)
Set if there was a carry from bit 3; cleared otherwise

S: \( N \oplus V \), For signed tests.

V: \( Rd7 \cdot Rr7 \cdot 7 + Rd7 \cdot Rr7 \cdot R7 \)
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: \( \overline{R7} \cdot \overline{R6} \cdot \overline{R5} \cdot R4 \cdot R3 \cdot R2 \cdot R1 \cdot \overline{R0} \)
Set if the result is $00; cleared otherwise.

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

---

Rd: The first register value
Rr: The second register value
R: The result register value of Rd + Rr

Example: R7 refers to the 7\textsuperscript{th} bit of the result.
Some intuitive explanations

- **N = R7**: The sign bit of two’s complement of the result
- **V bit**: Overflow happens if
  - Rd and Rr are positive and R is negative; or
  - Rd and Rr are negative and R is positive
- **S = N xor V**
  - No overflow: The N bit indeed tells if the result is negative
  - Overflow: The result is actually positive (S = 0) if it appears to be negative (N = 1), or negative (S = 1) if it appears to be positive (N = 0)

The N, V, S bits are meaningful if we interpret the operation as *signed* type
Some intuitive explanations (cont)

- **Z bit**: If all bits are zero, Z = 1
- **C bit**: carry happens if
  - Rd7 = 1, Rr7 = 1; or
  - Rd7 = 1, R7 = 0 (and Rr7 = 0); or
  - Rr7 = 1, R7 = 0 (and Rd7 = 0)
- **H bit**: Similar to C but based on bit 3 instead of bit 7

The C and H bits are meaningful if we interpret the operation as unsigned type

The Z bit is meaningful for both signed and unsigned type
Examples

• Let’s look at some examples
  – See if you can guess the value of the H, S, V, N, Z, and C flags
Add two operands: \( a + b \)

```
LDI R24, 0x18 ; load imme. a
LDI R22, 0x09 ; load imme. b
ADD R24, R22 ; a+b
```

If \( a = 24 \) (0b00011000), \( b = 9 \) (0b00001001), what are the values for those flags?

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>result</th>
<th>Z</th>
<th>C</th>
<th>H</th>
<th>N</th>
<th>V</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>9</td>
<td>33</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
a = 24 (0x18), b = 9 (0x09), result is 33 (0x21)

- **Z = 0: Result Not Zero**
  - The result 33 is not zero, no matter as signed or unsigned type

- **N = 0: Result Not Negative as signed type**
  - Look at the 7th bit of the result
  - The sign bit of 33 (0b00100001) is zero

- **V = 0: Operation has No Overflow as signed type**
  - 33 is a 7-bit value

- **S = 0: Actual result Not Negative as signed type**
  - The actual result is 33 for 24+9
  - $S = N \oplus V$

- **C = 0: No Carry**, or no overflow as unsigned type
  - The result 33 is an 8-bit value
Add two operands: \( a + b \)

```plaintext
LDI R24, 250 ; load \( a \)

LDI R22, 10 ; load \( b \)

ADD R24, R22 ; \( a+b \)
```

If \( a = 250 \) (0b11111010), \( b = 10 \) (0b00001010), what are the values for those flags?

<table>
<thead>
<tr>
<th></th>
<th>( a )</th>
<th>( b )</th>
<th>result</th>
<th>Z</th>
<th>C</th>
<th>H</th>
<th>N</th>
<th>V</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>250</td>
<td>10</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-6</td>
<td>10</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: Overflow as unsigned operation
The Stack Pointer Register: pointing to the top of the stack.

- Stack pointer is implemented as two 8-bit I/O registers
- SPH:SPL (most significant: least significant byte)
- Example for setting the SP register to top of stack

```asm
.DEF MyPreferredRegister = R16
.DEF RAMEND = $10FF
LDI MyPreferredRegister, HIGH(RAMEND) ; Upper byte
OUT SPH,MyPreferredRegister ; to stack pointer
LDI MyPreferredRegister, LOW(RAMEND) ; Lower byte
OUT SPL,MyPreferredRegister ; to stack pointer
```
Using Stack Pointer Register

– Place value onto the stack (Push)
  • Remember Stack starts at the highest address and grows downward. Thus “push” decrements SP.

```bash
PUSH R16 ; Throw that value in R16 on top of the stack
```

– Remove value from the stack (Pop). “pop” increments SP, i.e. makes the stake smaller

```bash
POP R16 ; Read value from the top of the stack, place in R16
```
Special Purpose Registers

Extension to General Purpose Registers

– Stack pointer
– Ports A, B, C, D, E, F, G, ...
– Registers related to interrupt
– And more ...
ATmega128 I/O Ports

I/O port registers are in the I/O spaces

- They also have their own memory addresses
- They can be directly accessed using a memory address

http://class.ece.iastate.edu/cpre288
Summary of AVR Registers

1. **GPRs: R0-R31**
   - R26/R27, R28/R29, R30/R31 are **X, Y, Z registers**

2. **Status register SREG**
   - H, S, N, V, Z, C bits

3. **Stack pointer SP**

4. **Special purpose registers SPRs**
   - SP is a SPR

http://class.ece.iastate.edu/cpre288
ATmega128 Memory Address

GPRs R0-R31: addresses 0x0000-0x001F
   – Directly accessed by ALU instructions and by memory instructions

I/O registers (space): 0x0020-0x005F
   – Directly accessed by IN/OUT instructions and by memory instructions

Extended I/O registers: 0x0060-0x00FF
   – Directly accessed by memory instructions only

Normal memory: 0x0100 above
   – Directly accessed by memory instructions only
Example AVR Assembly

```assembly
int a;
a = a + 10;

LDS R24, a    ; Load a's lower 8-bit
LDS R25, a+1  ; Load a's upper 8-bit
ADIW R24, 10  ; R24/R25 ← R24/R25+10
STS a, R24    ; save a's upper half
STS a+1, R25  ; save a's lower half
```
Example AVR Assembly

if (a > 0)

...  

CLR R1 ; R1 ← 0  
CP R1, R24 ; compare lower half  
CPC R1, R25 ; compare higher half  
BRGE else1 ; branch if greater than  
; or equal  

http://class.ece.iastate.edu/cpre288
How to Study Assembly

1. Get to know CPU registers
   Memorize some rules for usage

2. Know basic types of Instruction
   Memory load and store
   Arithmetic/Logic
   Compare and branch
3. Translate C statements
   - Memory accesses
   - Simple arithmetic statements
   - If statement
   - Loop statements

4. Translate C functions
   - Function Linkage
   - Making a function call
5. **Interrupt System**

- Principle of interrupt and exception
- Interrupt vector table
- Saving and restoring context
Challenges

Challenges in learning Assembly

– Must understand how CPU works cycle by cycle
– Have to memorize some notations before fully understanding them