# Memory

CSE 1302

# Pseudo-operations

Pseudo-ops are commands to assembler

| | |
|---|---|
| .text | means “text section”, or instructions are next |
| .data | means “data section” |
| .byte | reserves data storage |

```
var:   .byte   10
```

reserves one byte, initializes it to 10, and makes var a label that is address of byte

# Example

```
byte rshift2(byte x) {
        return(x >> 2);
}
```

```
        .text                   ;code segment follows
        .global rshift2         ;tell linker about rshift2
rshift2:
        lsr r24                 ;do actual work
        lsr r24                 ;result is in r24
        ldi r25, 0              ;return value in r25:r24
        ret                     ;return
```

# Data Segment

```
byte x;          //x declared as single byte
```

In assembly looks like this:

```
        .data       ;data segment follows
x:      .byte       ;reserve one byte for x
```

As in C, no automatic initialization takes place, this is the programmer's responsibility!

# Example (2)

```
byte xRshift2() {//x declared elsewhere
        return(x >> 2);
}

xRshift2:
        ldi r30, lo8(x)     ;load addr of x into
        ldi r31, hi8(x)     ;      index reg Z
        ld r24, Z           ;load x into r24
        lsr r24             ;do actual work
        lsr r24             ;result is in r24
        mov r25, r1         ;r1 is normally zero
        ret                 ;return
```

# Cautions

- Assembly has no understanding of data type
  - Programmer must handle multi-byte data
  - No conversions, Load (ld) just copies bits in memory to same bits in register
- Addresses are 16 bits
  - Requires two registers (r31:r30) and two loads
    - lo8(x) gives low byte of x, hi8(x) gives high byte of x
    - Use Load Immediate (ldi), because x is the address

# Multi-byte Data Manipulation

- Use bits in SREG to save intermediate values
- *C* bit (carry) for addition, e.g,

  r9:r8 ← r9:r8 + var

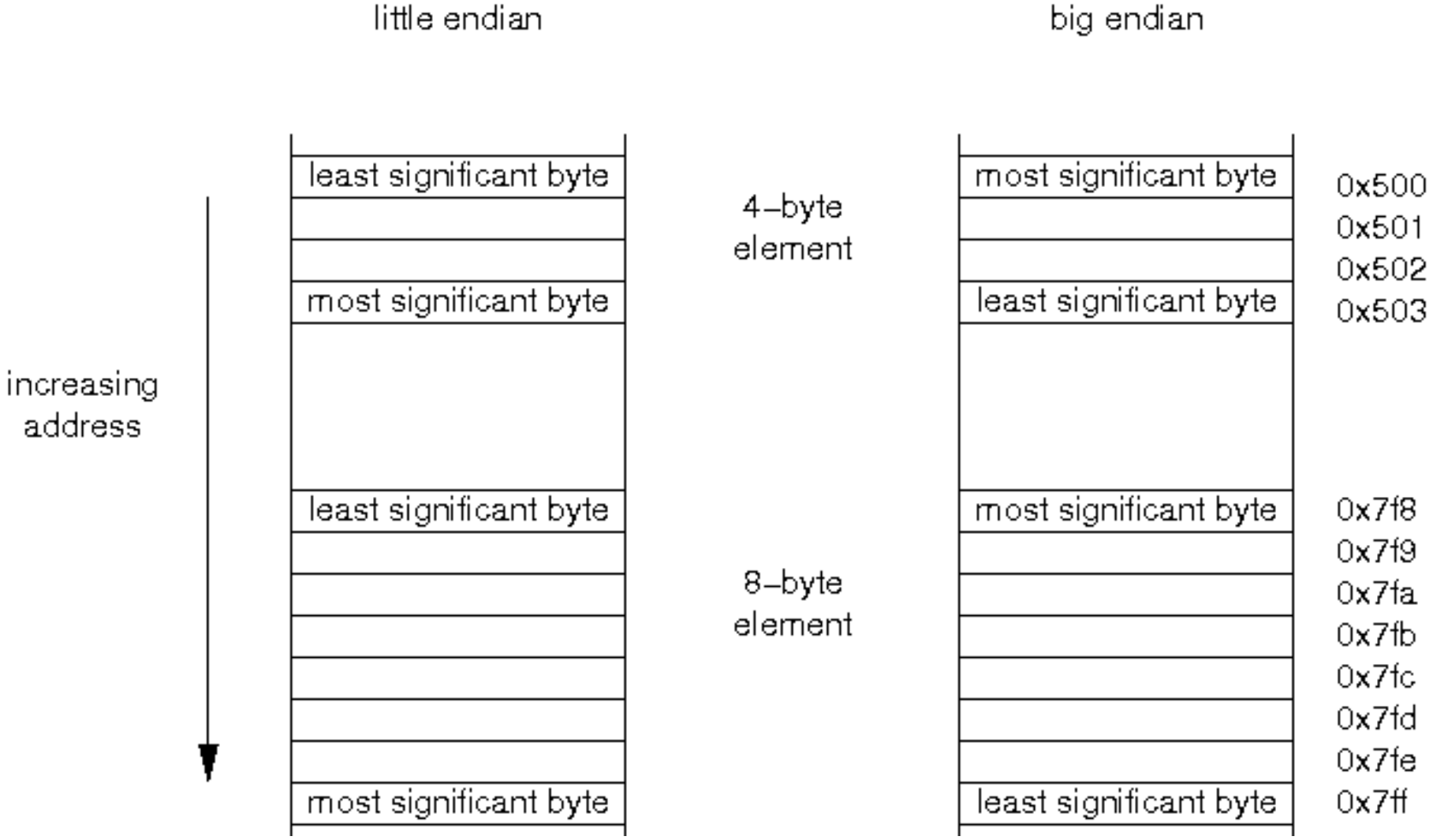
```
lds r4, (var)   ;load var into r5:r4
lds r5, (var+1)
add r8, r4      ;adds lsbits and puts carry in C
adc r9, r5      ;uses carry from prev. add
```

# Memory Organization

- On an AVR processor, there are multiple memories
  - Program memory
  - Data memory
- Data memory is byte addressable (i.e., each byte in the memory has a unique address)
- For multi-byte data elements, the address of the element is the lowest address the element occupies (e.g., for a 16-bit integer occupying 0x500 and 0x501, the address of the integer is 0x500)
- Program memory is 16-bit word addressable (the size of an AVR instruction)

# Little Endian vs. Big Endian

- The “endian”-ness of a processor defines the ordering it uses for multi-byte primitive data elements (e.g., 16-bit int on AVR).
- Little endian $\Rightarrow$ the least significant byte (LSB) goes in the lowest address, “littlest end first”
- Big endian $\Rightarrow$ the most significant byte (MSB) goes in the lowest address, “biggest end first”
- AVR is a little endian machine


little endian
big endian
least significant byte
most significant byte
most significant byte
least significant byte
4–byte
element
0x500
0x501
0x502
0x503
increasing
address
least significant byte
most significant byte
8–byte
element
0x7f8
0x7f9
0x7fa
0x7fb
0x7fc
0x7fd
0x7fe
0x7ff
most significant byte
least significant byte

# Primitive Elements vs. Collections

- Note: endianness only applies to primitive elements (e.g., integers, floats, etc.), not collections of elements

- A “string” is an array of characters, and therefore is not impacted by the endianness of the machine.  The first character is in the lowest address.

# Little Endian 32-bit Long Integer

```
long val  =  0x12345678;   /* &val is 0x5000 */
```

| addr | byte |
|---|---|
| 0x5000 | 0x78 |
| 0x5001 | 0x56 |
| 0x5002 | 0x34 |
| 0x5003 | 0x12 |

# More Addressing Modes

- Register addressing – operand is in register

add Rd, Rr  Rd ← Rd + Rr

source operand

destination operand

operation (op code)

$0 \leq d \leq 31, 0 \leq r \leq 31$

- Immediate addressing – operand is explicitly present in code

subi Rd, 10 $\quad\quad$ Rd ← Rd – 10

↑

"immediate" operand

- Constant values use C notation:
  - Default base is 10
  - Hex uses 0x notation ($10_{16}$ is written 0x10)
  - Negative constants are allowed, e.g., -12

- Direct addressing – memory address of operand is explicit in code

lds Rd, (k) Rd ← M[k]

$0 \leq d \leq 31, 0 \leq k \leq 65{,}535$

- Note: typically use symbols (variable names) instead of explicit *k*
  - Assembler translates into actual address
  - Or linker if specified in another file

sts (var), Rr M[var] ← Rr

- Indirect addressing – memory address is stored in register

st X, Rr  M[X] ← Rr

- Address register can be X, Y, or Z
  - X is (r27:r26), Y is (r29:r28), and Z is (r31:r30)

- There are also post-increment and pre-decrement versions

st X+, Rr  M[X] ← Rr, X ← X+1
st -X, Rr  X ← X-1, M[X] ← Rr

# Quiz Time

- Go to Canvas and answer the single question on Quiz 10A

True or False:

`.text` is an example of a pseudo-op.

# Array indexing

- If the array is declared as follows:

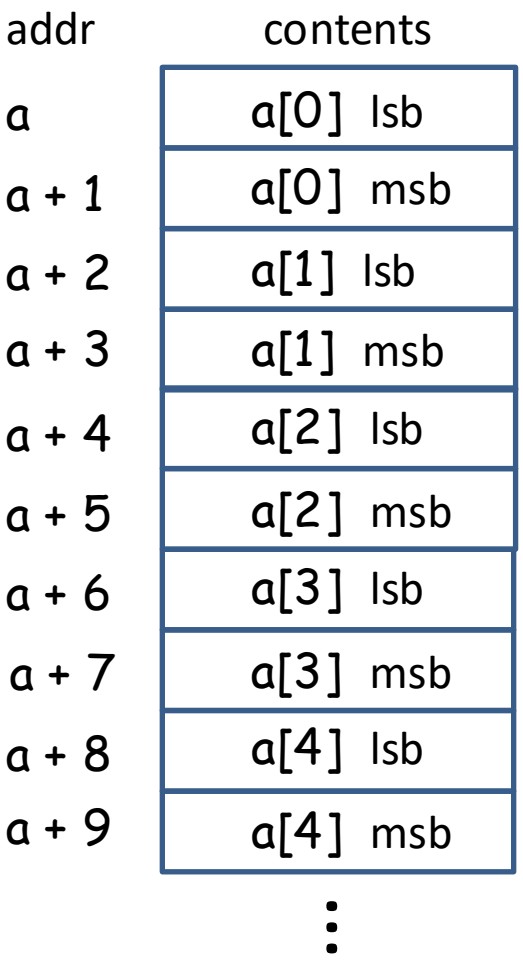
  int a[10];

- And I wish to read a[3] in assembly language
- Use Z (r31:r30) as index register

```
ldi r30,lo8(a)     ;use ldi for a pointer, lo8 and hi8 are macros
ldi r31,hi8(a)
ldi r16,3          ;put index value in a register
lsl r16            ;every int takes 2 addresses so multiply index by 2
add r30,r16
adc r31,r1         ;r1 is always zero in compiled C code
ld r18,Z+          ;actually do the load (in two instructions)
ld r19,Z
```

# Memory Layout

```
ldi r30,lo8(a)
ldi r31,hi8(a)
ldi r16,3
lsl r16
add r30,r16
adc r31,r1
ld r18,Z+
ld r19,Z
```




register contents

# Memory Layout

```
ldi r30,lo8(a)
ldi r31,hi8(a)
ldi r16,3
lsl r16
add r30,r16
adc r31,r1
ld r18,Z+
ld r19,Z
```

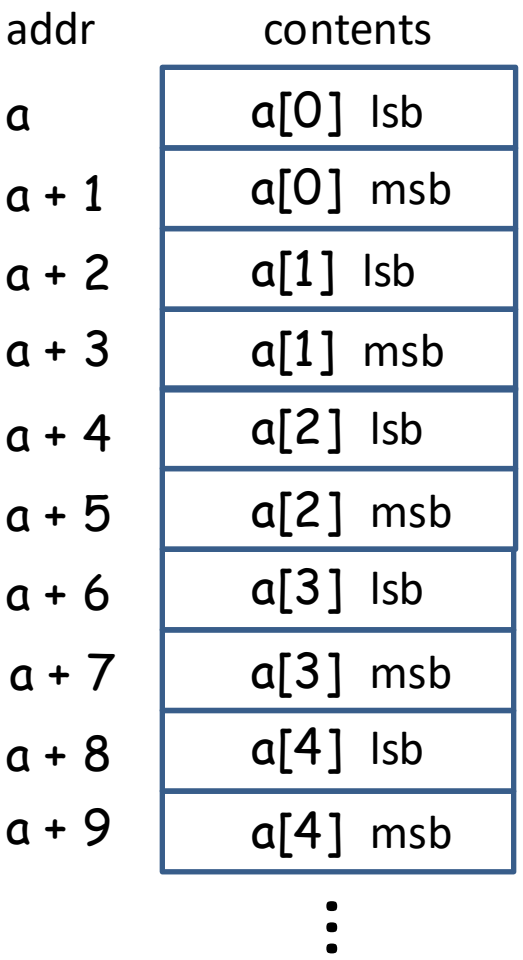
| addr | contents |
|---|---|
| a | a[0] lsb |
| a + 1 | a[0] msb |
| a + 2 | a[1] lsb |
| a + 3 | a[1] msb |
| a + 4 | a[2] lsb |
| a + 5 | a[2] msb |
| a + 6 | a[3] lsb |
| a + 7 | a[3] msb |
| a + 8 | a[4] lsb |
| a + 9 | a[4] msb |
| ⋮ | |

register contents

Z(r31:r30) addr of a

# Memory Layout

```
ldi r30,lo8(a)
ldi r31,hi8(a)
ldi r16,3
lsl r16
add r30,r16
adc r31,r1
ld r18,Z+
ld r19,Z
```




register contents

| | |
|---|---|
| Z(r31:r30) | addr of a |
| r16 | 3 |

# Memory Layout

```
ldi r30,lo8(a)
ldi r31,hi8(a)
ldi r16,3
lsl r16
add r30,r16
adc r31,r1
ld r18,Z+
ld r19,Z
```

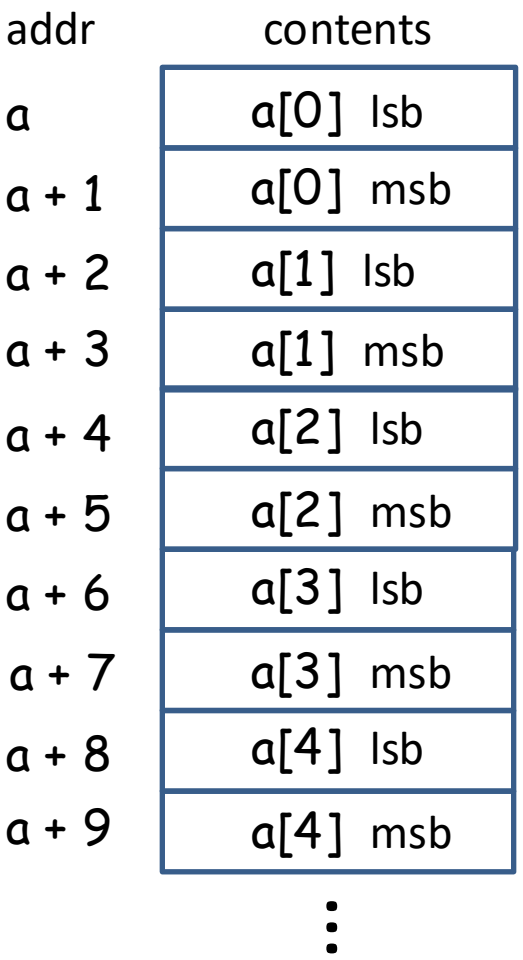
| addr | contents |
|---|---|
| a | a[0] lsb |
| a + 1 | a[0] msb |
| a + 2 | a[1] lsb |
| a + 3 | a[1] msb |
| a + 4 | a[2] lsb |
| a + 5 | a[2] msb |
| a + 6 | a[3] lsb |
| a + 7 | a[3] msb |
| a + 8 | a[4] lsb |
| a + 9 | a[4] msb |

register contents

| | |
|---|---|
| Z(r31:r30) | addr of a |
| r16 | 3 |
| r16 | 6 |

# Memory Layout

```
ldi r30,lo8(a)
ldi r31,hi8(a)
ldi r16,3
lsl r16
add r30,r16
adc r31,r1
ld r18,Z+
ld r19,Z
```




register contents

| | |
|---|---|
| Z(r31:r30) | addr of a |
| r16 | 3 |
| r16 | 6 |
| Z(r31:r30) | addr of a[3] |

# Memory Layout

```
ldi r30,lo8(a)
ldi r31,hi8(a)
ldi r16,3
lsl r16
add r30,r16
adc r31,r1
ld r18,Z+
ld r19,Z
```

| addr | contents |
|---|---|
| a | a[0] lsb |
| a + 1 | a[0] msb |
| a + 2 | a[1] lsb |
| a + 3 | a[1] msb |
| a + 4 | a[2] lsb |
| a + 5 | a[2] msb |
| a + 6 | a[3] lsb |
| a + 7 | a[3] msb |
| a + 8 | a[4] lsb |
| a + 9 | a[4] msb |

register contents

| | |
|---|---|
| Z(r31:r30) | addr of a |
| r16 | 3 |
| r16 | 6 |
| Z(r31:r30) | addr of a[3] |
| r19:r18 | a[3] |

# Call by Value vs. Call by Reference

- In C, when we make the following invocation:

```
foo(bar);
```

  what is actually passed to `foo`?

- Two options:
  - Call by value – `bar` is evaluated and passed to `foo`
  - Call reference – the address of `bar` is passed to `foo`
- C is call by value – to manually make it call by reference, do the following:

```
foo(&bar);
```

# Really? Always?

- Arrays in C act a bit different in practice
  - E.g., `int a[10];`
  - The symbol `a` is synonymous with `&a[0]`
  - I.e., the name of an array can be used as the address of the first element
  - So, `foo(a)` is the same as `foo(&a[0])`, which is effectively call by reference

# Call by Value in Assembly

- We can easily implement call by value in AVR assembly language
  - E.g., `foo(bar);  // bar is an int`
  - In the calling routine, evaluate `bar` (including if it is a more complicated expression)
  - Put the value of `bar` into r25:r24
  - Call `foo`
  - Use the value in r25:r24 within `foo`

# Call by Reference in Assembly

- Call by reference in AVR assembly language
  - In calling routine, put address of `bar` in r25:r24
    - This is effectively `foo(&bar);`
  - In `foo`
    - Move r25:r24 to pointer register (e.g., X)

      ```
      mov r27, r25
      mov r26, r24
      ```

    - Load value of `bar` into registers

      ```
      ld r24, X+
      ld r25, X
      ```

    - r25:r24 now contains value of `bar`

# Rest of the Semester Logistics

- Today is last of new material for the semester
- Studio 10 is Apr 13
  - Assignment 9, Quiz 9B are due Apr 13
- Lecture Apr 15 is review for Exam 3
- Assignment 10 is due Apr 20
  - Quiz 10B also due Apr 20
  - No studio that day
- Exam 3 is Wednesday, Apr 22