COMP2121: Microprocessors and Interfacing

AVR Assembly Programming (I)
Basic AVR Instructions

http://www.cse.unsw.edu.au/~cs2121

Lecturer: Hui Wu

Term 2, 2019

1

1

Contents

- Arithmetic and Logical Instructions
- Control instructions
- Sample AVR assembly programs

2

2

AVR Instruction Overview

- Load/store architecture
  - ❑ Only load/store instructions access memory
- At most two operands in each instruction
- Most instructions are two bytes long
- Some instructions are 4 bytes long
- Four Categories:
  - ❑ Arithmetic and logical instructions
  - ❑ Program control instructions
  - ❑ Data transfer instructions
  - ❑ Bit and bit test instructions

3

3

Selected Arithmetic and Logical Instructions

- Addition instructions: **add** (addition), **adc** (addition with carry), **inc** (increment)
- Subtraction instructions: **sub** (subtraction), **sbc** (subtraction with carry), **dec** (decrement)
- Multiplication instructions: **mul** (unsigned*unsigned) , **muls** (signed*signed), **mulsu** (signed*unsigned)
- Logical instructions: **and** (logical and), **or** (logical or), **eor** (logical exclusive or)
- Others: **clr** (clear register), **ser** (set register), **tst** (test for zero or negative), **com** (1's complement), **neg** (2's complement)
- Refer to the AVR Instruction Set for the complete list of instructions.

4

4

Selected Program Control Instructions

- Unconditional jump: jmp, rjmp, ijmp
- Subroutine call: rcall, icall, call
- Subroutine and interrupt return: ret, reti
- Conditional jump: breq, brne, brsh, brlo, brge, brlt, brvs, brvc, brie, brid
- Refer to the AVR Instruction Set for a complete list.

5

5

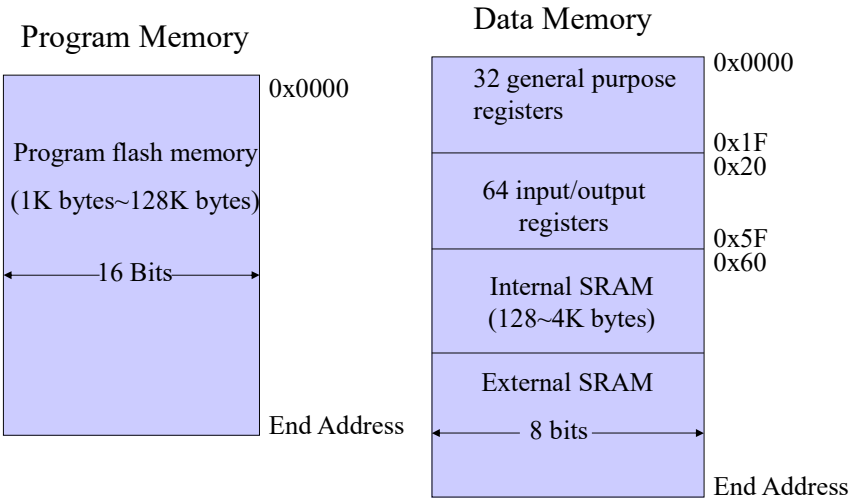
Selected Data Transfer Instructions

- Copy instructions: mov, movw
- Load instructions: ldi, ld, ldd, lds
- Store instructions: st, std, sts
- Load program memory: lpm
- I/O instructions: in, out
- Stack instructions: push, pop

6

6

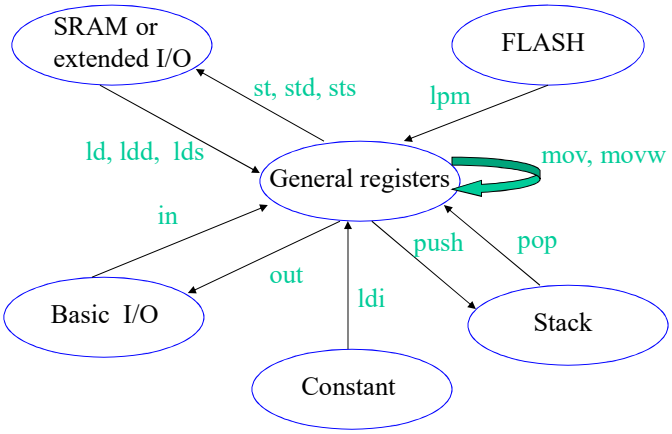
AVR Program Memory and Data Memory



7

7

A Global Picture of Data Transfer Instructions



8

8

Selected Shift and Bit-set Instructions

- Shift instructions: lsl, lsr, rol, ror, asr
- Bit-set Instructions: bset, bclr, sbi, cbi, bst, bld, sex, clx, nop, sleep, wdr, break
- Refer to the AVR Instruction Set for a complete list.

9

9

Add without Carry

- Syntax: add Rd, Rr
- Operands: Rd, Rr ∈{r0, r1, …, r31}
- Operation: Rd←Rd + Rr
- Flags affected: H, S, V, N, Z, C
- Encoding: 0000 11rd dddd rrrr
- Words: 1
- Cycles: 1
- Example: add r1, r2 ; Add r2 to r1
  add r28, r28 ; Add r28 to itself

10

10

Add with Carry (1/2)

- Syntax: adc Rd, Rr
- Operands: Rd, Rr ∈ {r0, r1, …, r31}
- Operation: Rd←Rd + Rr + C
- Flags affected: H, S, V, N, Z, C
- Encoding: 0001 11rd dddd rrrr
- Words: 1
- Cycles: 1
- Example: Add r1 : r0 to r3 : r2

```
add r2, r0      ; Add low byte
adc r3, r1      ; Add high byte
```

- Comments: adc is used in multi-byte addition.

11

11

Add with Carry (2/2)

- When adding two n byte integers, use ***add*** to add bytes 0, then use ***adc*** to add bytes 1, bytes 2, …, bytes n–1.
- Example: Assume that an integer x is stored in registers r5:r4:r3:r2, and an integer y is stored in registers r10:r9:r8:r7, where the left–most register stores the most significant bye and the right–most register stores the least significant byte.

```
add r7, r2      ; Add bytes 0
adc r8, r3      ; Add bytes 1
adc r9, r4      ; Add bytes 2
adc r10, r5     ; Add bytes 3
```

12

12

Increment

- Syntax: inc Rd
- Operands: Rd ∈{r0, r1, …, r31}
- Operation: Rd←Rd+1
- Flags affected: S, V, N, Z
- Encoding: 1001 010d dddd 1011
- Words: 1
- Cycles: 1
- Example:

```
clr r22          ; clear r22
loop: inc r22    ; Increment r22
      cpi r22, $4F ; compare r22 to $4F
      brne loop  ; Branch to loop if not equal
      nop        ; Continue (do nothing
```

13

13

Subtract without Carry

- Syntax: sub Rd, Rr
- Operands: Rd, Rr ∈{r0, r1, …, r31}
- Operation: Rd←Rd–Rr
- Flags affected: H, S, V, N, Z, C
- Encoding: 0001 10rd dddd rrrr
- Words: 1
- Cycles: 1
- Example:

```
sub r13, r12   ; Subtract r12 from r13
```

14

14

Subtract with Carry (1/2)

- Syntax: sbc Rd, Rr
- Operands: Rd, Rr ∈ {r0, r1, …, r31}
- Operation: Rd←Rd–Rr–C
- Flags affected: H, S, V, N, Z, C
- Encoding: 0000 10rd dddd rrrr
- Words: 1
- Cycles: 1
- Example: Subtract r1:r0 from r3:r2

```
sub r2, r0   ; Subtract low byte
sbc r3, r1   ; Subtract with carry high byte
```

Comments: sbc is used in multi-byte subtraction

15

15

Subtract with Carry (2/2)

- When subtracting one n byte integer from another n byte integer, use ***sub*** to subtract bytes 0, then use ***sbc*** to subtract bytes 1, bytes 2, …, bytes n–1.
- Example: Assume that an integer x is stored in registers r5:r4:r3:r2, and an integer y is stored in registers r10:r9:r8:r7, where the left–most register stores the most significant bye and the right–most register stores the least significant byte.

```
sub r7, r2    ; subtract bytes 0
sbc r8, r3    ; subtract bytes 1
sbc r9, r4    ; subtract bytes 2
sbc r10, r5   ; subtract bytes 3
```

16

16

Decrement

- Syntax: dec Rd
- Operands: Rd ∈{r0, r1, …, r31}
- Operation: Rd←Rd–1
- Flags affected: S, V, N, Z
- Encoding: 1001 010d dddd 1010
- Words: 1
- Cycles: 1
- Example:

```
ldi r17, $10   ; Load constant in r17
loop: add r1, r2    ; Add r2 to r1
      dec r17       ; Decrement r17
      brne loop;    ; Branch to loop if r17≠0
      nop           ; Continue (do nothing)
```

17

17

Multiply Unsigned

- Syntax: mul Rd, Rr
- Operands: Rd, Rr ∈{r0, r1, …, r31}
- Operation: r1, r0←Rd*Rr (unsigned←unsigned * unsigned)
- Flags affected: Z, C
- Encoding: 1001 11rd dddd rrrr
- Words: 1
- Cycles: 2
- Example:

```
mul r6, r5    ; Multiply r6 and r5
mov r6, r1
mov r5, r0    ; Copy result back in r6 : r5
```

18

18

Multiply Signed

- Syntax: muls Rd, Rr
- Operands: Rd, Rr ∈{r16, r17, …, r31}
- Operation: r1, r0←Rd*Rr (signed←signed * signed)
- Flags affected: Z, C
- Encoding: 0000 0010 dddd rrrr
- Words: 1
- Cycles: 2
- Example:
```
mul r17, r16          ; Multiply r17 and r16
movw r17:r16, r1:r0   ; Copy result back to r17 : r16
```

19

19

Multiply Signed with Unsigned

- Syntax: mulsu Rd, Rr
- Operands: Rd, Rr ∈{r16, r17, …, r23}
- Operation: r1, r0←Rd*Rr (signed←signed * unsigned)
- Flags affected: Z, C
  C is set if bit 15 of the result is set; cleared otherwise.
- Encoding: 0000 0011 0ddd 0rrr
- Words: 1
- Cycles: 2

20

20

An Example of Using Multiply Instructions (1/3)

Multiply two unsigned 16-bit numbers stored in r23:r22 and r21:r20, respectively, and store the 32-bit result in r19:r18:r17:r16.

How to do it?

Let ah and al be the high byte and low byte, respectively, of the multiplicand and bh and bb the high byte and low byte, respectively, of the multiplier.

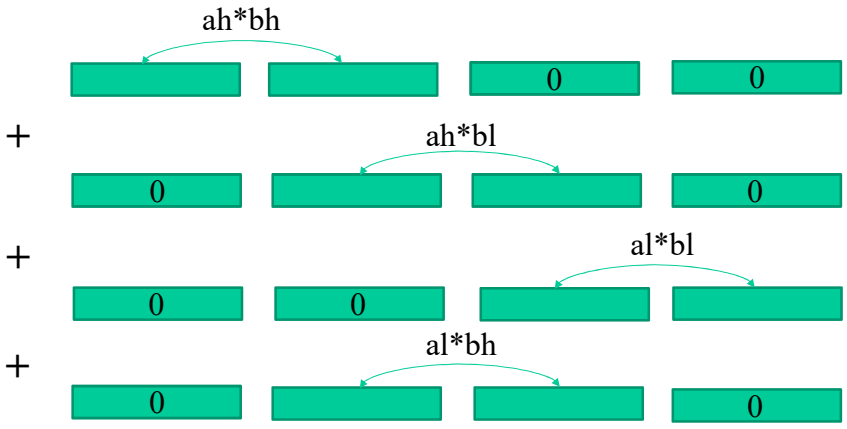
ah : al * bh : bl
= (ah* 2^8+ al) * (bh* 2^8+bl)
= ah*bh*2^{16} + al*bh* 2^8 + ah*bl*2^8 + al*bl

21

21

An Example of Using Multiply Instructions (2/3)

One procedure:



22

22

An Example of Using Multiply Instructions (3/3)

```
mul16x16_32:
    clr     r2                  ; r2=0
    mul     r23, r21            ; (unsigned) ah * (unsigned) bh
    movw    r19 : r18, r1 : r0  ; move the result of ah * bh to r19:r18
    mul     r22, r20            ; (unsigned) al * (unsigned) bl
    movw    r17 : r16, r1: r0   ; move the result of al * bl to r17:r16
    mul     r23,  r20           ; (unsigned) ah * (unsigned) bl
    add     r17, r0
    adc     r18, r1
    adc     r19, r2
    mul     r21, r22            ; (unsigned) bh * (unsigned) al
    add     r17, r0
    adc     r18, r1
    adc     r19, r2
    ret                         ; return to the caller
```

23

23

Bitwise AND

- Syntax: and Rd, Rr
- Operands: Rd, Rr ∈ {r0, r1, …, r31}
- Operation: Rd←Rr · Rd (Bitwise AND Rr and Rd)
- Flags affected: S, V, N, Z
- Encoding: 0010 00rd dddd rrrr
- Words: 1
- Cycles: 1
- Example:

```
ldi  r20, 0b00110101
ldi  r16, 1
and  r20, r16          ; r20=0b00000001
```

24

24

Bitwise OR

- Syntax: or Rd, Rr
- Operands: Rd, Rr ∈{r0, r1, …, r31}
- Operation: Rd←Rr v Rd (Bitwise OR Rr and Rd)
- Flags affected: S, V, N, Z
- Encoding: 0010 10rd dddd rrrr
- Words: 1
- Cycles: 1
- Example:

```
ldi r15, 0b11110000
ldi r16, 0b00001111
or r15, r16          ; Do bitwise or between registers
                     ; r15=0b11111111
```

25

25

Bitwise Exclusive-OR

- Syntax: eor Rd, Rr
- Operands: Rd, Rr ∈{r0, r1, …, r31}
- Operation: Rd←Rr ⊕ Rd (Bitwise exclusive OR Rr and Rd)
- Flags affected: S, V, N, Z
- Encoding: 0010 01rd dddd rrrr
- Words: 1
- Cycles: 1
- Example:

```
eor  r4,  r4      ; Clear r4
eor  r0, r22      ; Bitwise exclusive or between r0 and r22
                  ; If r0=0b10101011 and r22=0b01001000
                  ; then r0=0b11100011
```

26

26

Clear Bits in Register

- Syntax: cbr Rd, k
- Operands: Rd ∈ {r16, r17, …, r31} and 0 ≤ k ≤ 255
- Operation: Rd←Rd · ($FF-k) (Clear the bits specified by k)
- Flags affected: S, V, N, Z
- Encoding: 0111 wwww dddd wwww (wwwwwwww=$FF-k)
- Words: 1
- Cycles: 1
- Example:

```
cbr  r4, 0b11          ; Clear bits 0 and 1 of r4.
```

27

27

Compare without Carry

- Syntax: cp Rd, Rr
- Operands: Rd ∈ {r0, r1, …, r31}
- Operation: Rd–Rr (Rd is not changed)
- Flags affected: H, S, V, N, Z, C
- Encoding: 0001 01rd dddd rrrr
- Words: 1
- Cycles: 1
- Example:

```
        cp r4, r5          ; Compare r4 with r5
        brne noteq         ; Branch if r4 ≠ r5
        ...
noteq:  nop                ; Branch destination (do nothing)
```

- The only difference between ***cp*** and ***sub*** is that the result of ***cp*** is not stored.
- ***cp*** is used immediately before a branch instruction.

28

28

Compare with Carry (1/2)

- Syntax: cpc Rd, Rr
- Operands: Rd ∈{r0, r1, …, r31}
- Operation: Rd–Rr–C (Rd is not changed)
- Flags affected: H, S, V, N, Z, C
- Encoding: 0001 01rd dddd rrrr
- Words: 1
- Cycles: 1
- Example:

```
                        ; Compare r3:r2 with r1:r0
        cp r2, r0       ; Compare low bytes
        cpc r3, r1      ; Compare high bytes
        brne noteq      ; Branch if not equal
        ...
noteq:  nop             ; Branch destination (do nothing)
```

29

29

Compare with Carry (2/2)

- The only difference between ***cpc*** and ***sbc*** is that the result of cpc is not stored.
- When comparing two n–byte integers, use ***cp*** to compare bytes 0, and ***cpc*** to compare bytes 1, bytes 2, …, bytes n–1.
- Example: Assume that an integer x is stored in registers r5:r4:r3:r2, and an integer y is stored in registers r10:r9:r8:r7, where the left–most register stores the most significant bye and the right–most register stores the least significant byte.

```
        cp   r7, r2     ; compare bytes 0
        cpc r8, r3      ; compare bytes 1
        cpc r9, r4      ; compare bytes 2
        cpc r10, r5     ; compare bytes 3
        brne noteq      ; if x!=y, goto noteq
        …
noteq: inc r0
```

30

30

Compare with Immediate

- Syntax: cpi Rd, k
- Operands: Rd ∈{r16, r17, …, r31} and 0≤ k ≤ 255
- Operation: Rd – k (Rd is not changed)
- Flags affected: H, S, V, N, Z, C
- Encoding: 0011 kkkk dddd kkkk
- Words: 1
- Cycles: 1

- Example:

```
        cpi r19, 30        ; Compare r19 with 30
        brne noteq         ; Branch if r19 ≠ 30
        ...
noteq:  nop                ; Branch destination (do nothing)
```

31

31

Test for Zero or Minus

- Syntax: tst Rd
- Operands: Rd ∈{r0, r1, …, r31}
- Operation: Rd←Rd · Rd
- Flags affected: S, V, N, Z
- Encoding: 0010 00dd dddd dddd
- Words: 1
- Cycles: 1
- Example:

```
        tst r0             ; Test r0
        breq zero          ; Branch if r0=0
        ...
zero:   nop                ; Branch destination (do nothing)
```

32

32

One's Complement

- Syntax: com Rd
- Operands: Rd ∈ {r0, r1, …, r31}
- Operation: Rd←$FF – Rd
- Flags affected: S, V, N, Z
- Encoding: 1001 010d dddd 0000
- Words: 1
- Cycles: 1
- Example:

```
        com r4          ; Take one's complement of r4
        breq zero       ; Branch if zero
        ...
zero:   nop             ; Branch destination (do nothing)
```

33

33

Two's Complement

- Syntax: neg Rd
- Operands: Rd ∈ {r0, r1, …, r31}
- Operation: Rd←$00 – Rd (The value of $80 is left unchanged)
- Flags affected: H, S, V, N, Z, C

  H: R3 + *Rd3*

  Set if there is a borrow from bit 3; cleared otherwise
- Encoding: 1001 010d dddd 0001
- Words: 1
- Cycles: 1
- Example:

```
        sub r11,r0      ;Subtract r0 from r11
        brpl positive   ;Branch if result positive
        neg r11         ;Take two's complement of r11
positive: nop           ;Branch destination (do nothing)
```

34

34

Sign Extension (1/4)

- Remember that negative numbers in computers are represented by their 2's complements.
- How to extend a binary number of m bits to an equivalent binary number of m+n bits?

Example 1: $x = (0100)_2 = 4$

$x = (0000\ 0100)_2$

$= (0000\ 0000\ 0100)_2$

In general, if the number is a positive number, add n 0's to the left of the binary number. This procedure is called sign extension.

35

35

Sign Extension (2/4)

Example 2: $x = (1100)_2 = -4$

$x = (1111\ 1100)_2$

$= (1111\ 1111\ 1100)_2$

In general, if the number is a negative number, add n 1's to its left. This procedure is called sign extension.

36

36

Sign Extension (3/4)

How to add two binary numbers of different lengths?

- Sign-extend the shorter number such that it has the same length as the longer number, and then add both numbers.

Example 3: $x = (11010100)_2 = -44$,

$y = (0100)_2 = 4$

x+y=?

Since y is a positive number, $y=(00000100)_2$.

$x+y = (11010100)_2 + (00000100)_2$

$= (11011000)_2 = -40$

37

37

Sign Extension (4/4)

Example 4: $x = (11010100)_2 = -44$,

$y = (1100)_2 = -4$

x+y=?

Since y is a negative number, $y=(11111100)_2$.

$x+y = (11010100)_2 + (11111100)_2$

$= (11010000)_2 = -48$

38

38

AVR Assembly Programming: Example 1 (1/3)

The following AVR assembly program implements division by using repeated subtractions. The dividend is stored in in the register r18 and r17, where r18 stores the most significant byte and r17 stores the least significant byte. The divisor is stored in r19. The quotient is stored in r21 and r20 and the remainder is stored in r17.

```
.include  "m2560def.inc"    ; Include definition file for ATmega2560
.equ dividend=2121          ; Define dividend to be 2121
.def dividend_high=r18      ; Define dividend_high to be r18
.def dividend_low=r17
.def divisor=r19
.def quotient_low=r20
.def quotient_high=r21
.def zero=r16
```

39

39

AVR Assembly Programming: Example 1 (2/3)

```
.cseg                       ; Define a code segment
.org  0x100                 ; Set the starting address of code segment to 0x100
ldi dividend_low, low(dividend)     ; load r17 with  the low byte of the dividend
ldi dividend_high, high(dividend)   ; load r18 with the high byte of the dividend
ldi divisor, 45                     ; load r19 with 45
clr quotient_low            ; quotient_low=0
clr quotient_high           ; quotient_high=0
clr zero                    ; zero=0
```

40

40

AVR Assembly Programming: Example 1 (3/3)

```
loop:  cp dividend_low, divisor              ; Compare low bytes
       cpc dividend_high, zero               ; Compare high bytes
       brlo done                             ; If dividend<divisor, go to done.
       sub dividend_low, divisor             ; dividend_low=dividend_low-divisor
       sbc dividend_high, zero               ; dividend_high=dividend_high-C
       adiw quotient_high:quotient_low, 1    ; Increase the quotient by 1.
       rjmp loop
done: rjmp done                              ; Loop forever here
```

41

41

ARV Assembly Programming: Example 2 (1/5)

Convert the following C program into an AVR assembly program:

```
unsigned  int i, sum;
void main()
 {
   sum=0;
   for (i =1; i<100; i++)
       sum=sum+i;
 }
```

42

42

ARV Assembly Programming: Example 2 (2/5)

We can use general registers to store i and sum.

Question: How many general registers are needed to store i and sum, respectively?

Answer: One general register for i and two general registers for sum. Why?

- The maximum unsigned number stored in one AVR general register is 11111111_b $= 2^8 - 1 = 255$.
- The maximum unsigned number stored in two AVR general registers is 11111111 $11111111_b = 2^{16}-1 = 65535$.

We use r10 to store i and r12:r11 to store sum.

ARV Assembly Programming: Example 2 (3/5)

To understand the structure of for loop, we convert the C program into an equivalent program:

```
unsigned  int i, sum;
void main()
 {
   sum=0;
   i=1;
   forloop: sum=sum+i;
   i=i+1;
   if ( i<100)
      goto forloop;
 }
```

ARV Assembly Programming: Example 2 (4/5)

```
unsigned  int i, sum;
   void main()
    {


      sum=0;


      i=1;
```



```
.include "m2560def.inc" ; Include definition file for Mega64
.def  zero=r0        ; Define i to be r0
.def  i=r16          ; Define i to be r16
.def  sum_h=r12      ; Define sum_h to be r12
.def  sum_l=r11      ; Define sum_l to be r11
.cseg
clr zero       ; zero is used to sign-extend i
clr sum_h
clr sum_l
ldi i, 1       ; Note that i must be in r16-r31
```

45

45

ARV Assembly Programming: Example 2 (5/5)

```
    forloop: sum=sum+i;


    i=i+1;


    if ( i<100)
       goto forloop;
}
```

```
forloop: add sum_l, i    ; Add bytes 0
adc sum_h, zero          ; Add bytes 1

inc i            ; Increment loop counter


cpi i, 100
brlo forloop
loopforever: rjmp loopforever
```

46

46

Jump

- Syntax: jmp k
- Operands: $0 \le k < 4M$
- Operation: PC←k
- Flag affected: None
- Encoding: 1001 010k kkkk 110k
  kkkk kkkk kkkk kkkk
- Words: 2
- Cycles: 3
- Example:

```
        mov r1, r0      ; Copy r0 to r1
        jmp farplc      ; Unconditional jump
        ...
farplc: inc r20         ; Jump destination
```

47

47

Relative Jump

- Syntax: rjmp k
- Operands: $-2K \le k < 2K$
- Operation: PC←PC+k+1
- Flag affected: None
- Encoding: 1100 kkkk kkkk kkkk
- Words: 1
- Cycles: 2
- Example:

```
        cpi  r16, $42   ; Compare r16 to $42
        brne error      ; Branch to error if r16 ≠ $42
        rjmp ok         ; jump to ok
error:  add  r16, r17   ; Add r17 to r16
        inc  r16        ; Increment r16
ok:     mov r2, r20     ; Jump destination
```

48

48

Indirect Jump (1/2)

- Syntax: ijmp
- Operation:

```
(i)  PC←Z(15:0) Devices with 16 bits PC, 128K bytes
program memory maximum.
(ii) PC(15:0)←Z(15:0)Devices with 22 bits PC, 8M bytes
program memory maximum.
     PC(21:16) <- 0
```

- Flag affected: None
- Encoding: 1001 0100 0000 1001
- Words: 1
- Cycles: 2

49

49

Indirect Jump (2/2)

- Example:

```
        clr r10                 ; Clear r10
        ldi r20, 2              ; Load jump table offset
        ldi r30, low(Lab<<1)    ; High byte of the starting address (base) of jump table
        ldi r31, high(Lab<<1)   ; Low byte of the starting address (base) of jump table
        add r30, r20
        adc r31, r10            ; Base + offset is the address of the jump table entry
        lpm r0, Z+              ; Load low byte of the the jump table entry
        lpm r1, Z               ; Load high byte of the jump table entry
        movw r31:r30, r1:r0     ; Set the pointer register Z to point the target instruction
        ijmp                    ; Jump to the target instruction
        …
Lab:    .dw jt_l0               ; The first entry of the jump table
        .dw jt_l1               ; The second entry of the jump table
        …
jt_l0:  nop
jt_l1:  nop
        …
```

50

50

Branch If Equal

- Syntax: breq k
- Operands: $-64 \leq k < 63$
- Operation: If Rd = Rr (Z = 1) then PC ← PC + k + 1, else PC ← PC + 1
- Flag affected: None
- Encoding: 1111 00kk kkkk k001
- Words: 1
- Cycles: 1 if condition is false
  2 if conditional is true
- Example:

```
        cp r1, r0       ; Compare registers r1 and r0
        breq  equal     ; Branch if registers equal
        ...
equal:  nop             ; Branch destination (do nothing)
```

51

51

Branch If Same or Higher (Unsigned)

- Syntax: brsh k
- Operands: $-64 \leq k < 63$
- Operation: if Rd ≥ Rr (unsigned comparison) then PC ← PC + k + 1, else PC ← PC + 1
- Flag affected: none
- Encoding: 1111 01kk kkkk k000
- Words: 1
- Cycles: 1 if condition is false
  2 if conditional is true
- Example:

```
        subi r26, $56       ; subtract $56 from r26
        brsh test           ; branch if r26 ≥ $56
        ...
Test:   nop                 ; branch destination
        …
```

52

52

Branch If Lower (Unsigned)

- Syntax: brlo k
- Operands: -64 ≤ k < 63
- Operation: If Rd < Rr (unsigned comparison) then PC ← PC + k + 1, else PC ← PC + 1
- Flag affected: None
- Encoding: 1111 00kk kkkk k000
- Words: 1
- Cycles: 1 if condition is false
  2 if conditional is true
- Example:

```
        eor  r19, r19          ; Clear r19
loop:   inc r19                ; Increase r19
        ...
        cpi  r19, $10          ; Compare r19 with $10
        brlo loop              ; Branch if r19 < $10 (unsigned)
        nop                    ; Exit from loop (do nothing)
```

53

53

Branch If Less Than (Signed)

- Syntax: brlt k
- Operands: -64 ≤ k < 63
- Operation: If Rd < Rr (signed comparison) then PC ← PC + k + 1, else PC ← PC + 1
- Flag affected: None
- Encoding: 1111 00kk kkkk k100
- Words: 1
- Cycles: 1 if condition is false
  2 if conditional is true
- Example:

```
        cp r16, r1          ; Compare r16 to r1
        brlt less           ; Branch if r16 < r1 (signed)
        ...
less:   nop                 ; Branch destination (do nothing)
```

54

54

Branch If Greater or Equal (Signed)

- Syntax: brge k
- Operands: -64 ≤ k < 63
- Operation: If Rd ≥ Rr (signed comparison) then PC ← PC + k + 1, else PC ← PC + 1
- Flag affected: None
- Encoding: 1111 01kk kkkk k100
- Words: 1
- Cycles: 1 if condition is false
  2 if conditional is true
- Example:

```
        cp    r11, r12      ; Compare registers r11 and r12
        brge  greateq       ; Branch if r11 ≥ r12 (signed)
        ...
greateq: nop                ; Branch destination (do nothing)
```

55

55

Branch If Overflow Set

- Syntax: brvs k
- Operands: -64 ≤ k < 63
- Operation: If V=1 then PC ← PC + k + 1, else PC ← PC + 1
- Flag affected: None
- Encoding: 1111 00kk kkkk k011
- Words: 1
- Cycles: 1 if condition is false
  2 if conditional is true
- Example:

```
        add r3, r4          ; Add r4 to r3
        brvs overfl         ; Branch if overflow
        ...
overfl: nop                 ; Branch destination (do nothing)
```

56

56

Branch If Overflow Clear

- Syntax: brvc k
- Operands: -64 ≤ k < 63
- Operation: If V=0 then PC ← PC + k + 1, else PC ← PC + 1
- Flag affected: None
- Encoding: 1111 01kk kkkk k011
- Words: 1
- Cycles: 1 if condition is false
  2 if conditional is true
- Example:

```
        add r3, r4          ; Add r4 to r3
        brvs noover         ; Branch if no overflow
        ...
noover: nop                 ; Branch destination (do nothing)
```

57

57

Branch if Global Interrupt is Enabled

- Syntax: brie k
- Operands: -64 ≤ k < 63
- Operation: If I=1 then PC ← PC + k + 1, else PC ← PC + 1
- Flag affected: None
- Encoding: 1111 00kk kkkk k111
- Words: 1
- Cycles: 1 if condition is false
  2 if conditional is true
- Example:

```
        brvs inten          ; Branch if the global interrupt is enabled
        ...
inten:  nop                 ; Branch destination (do nothing)
```

58

58

Branch if Global Interrupt is Disabled

- Syntax: brid k
- Operands: $-64 \leq k < 63$
- Operation: If I=0 then PC ← PC + k + 1, else PC ← PC + 1
- Flag affected: None
- Encoding: 1111 00kk kkkk k111
- Words: 1
- Cycles: 1 if condition is false
  2 if conditional is true
- Example:

```
        brid intdis        ; Branch if the global interrupt is enabled
        ...
intdis: nop                ; Branch destination (do nothing)
```

59

59

Signed Branching vs. Unsigned Branching (1/2)

Consider the following AVR assembly code:

```
        ldi  r16, low(-1)
        ldi  r17, high(-1)
        ldi  r20, low(0x500)
        ldi  r21, high(0x500)
        cp   r16, r20
        cpc  r17, r21
        brge comp2121
        clr r0
        rjmp end
comp2121: ldi r0, 1
    end:  nop
```

What is the value in register r0 after the above sequence of code is executed?

60

60

Signed Branching vs. Unsigned Branching (2/2)

Consider the following AVR assembly code:

```
            ldi  r16, low(-1)
            ldi  r17, high(-1)
            ldi  r20, low(0x500)
            ldi  r21, high(0x500)
            cp   r16, r20
            cpc  r17, r21
            brsh comp2121
            clr r0
            rjmp end
comp2121:   ldi r0, 1
    end:    nop
```

What is the value in register r0 after the above sequence of code is executed?

61

61

AVR Assembly Programming: Example 3 (1/4)

Convert the following C program into an AVR assembly program:

```
int x, z;      /* Assume that the size of a signed integer is two bytes */
short int y;  /* Assume that the size of a signed short integer is one byte */
void main()
 {
   …
   if (x<y)
       z=x-y;
    else
       z=x+y;
 }
```

62

62

AVR Assembly Programming: Example 3 (2/4)

```
int x, z;

short int y;
```

```
.include "m2560def.inc"  ; Include definition
file for Mega64
.def  x_h=r12    ; Define x_h to be r12
.def  x_l=r11    ; Define x_l to be r11
 def  z_h=r14    ; Define z_h to be r14
.def  z_l=r13    ; Define z_l to be r13

.def  y=r16      ; Define y to be r16
.def  temp=r17   ; Define temp to be r0
                 ; temp is used to sign-extend y
```

63

63

AVR Assembly Programming: Example 3 (3/4)

```
void main()
  {

   …

   if (x<y)
```

```
.cseg
cpi y,  0          ; Sign extend y
brlt  negative     ; Signed-extended y is stored
clr temp           ; in the register pair temp:y
rjmp next
negative: ldi temp, $FF
mov z_l, x_l       ; Copy x to z
mov z_h, x_h
cp x_l, y          ; Compare x with y
cpc x_h, temp
brge  elsepart
```

64

64

AVR Assembly Programming: Example 3 (4/4)

```
        z=x-y;



    else
        z=x+y;

}
```

```
sub z_l, y
sbc z_h, temp
rjmp end;

elsepart:
add z_l, y
adc z_l, temp
end: rjmp end;   ; Loop forever here
```

65

65

Reading Material

1. AVR Instruction Set (http://www.cse.unsw.edu.au/~cs2121/AVR/AVR-Instruction-Set.pdf)
2. AVR Assembler User Guide (http://www.cse.unsw.edu.au/~cs2121/AVR/AVR-Assembler-Guide.pdf)

66

66