Overview

- Data transfer instructions in AVR
- Sample AVR assembly programs using data transfer instructions

Motivations

- How to transfer data between two registers?
- How to transfer data between memory and a register?
- How to transfer a constant to a register?

Selected Data Transfer Instructions

- mov, movw
- ldi, ld, ldd, lds
- st, stc
- lpm
- in, out
- Push, pop
- Refer to the main textbook and AVR Instruction Set for a complete list.
**Copy Register**

- **Syntax:** `mov Rd, Rr`
- **Operands:** $Rd, Rr \in \{r0, r1, \ldots, r31\}$
- **Operation:** $Rd \leftarrow Rr$
- **Flag affected:** None
- **Encoding:** 0010 11rd dddd rrrr
- **Words:** 1
- **Cycles:** 1
- **Example:**
  ```assembly```
mov r1, r0 ; Copy r0 to r1```
```assembly```

**Copy Register Pair**

- **Syntax:** `movw Rd+1:Rd, Rr+1:Rr`
- **Operands:** $d, r \in \{0, 2, \ldots, 28, 30\}$
- **Operation:** $Rd+1:Rd \leftarrow Rr+1:Rr$
- **Flag affected:** None
- **Encoding:** 0000 0001 dddd rrrr
- **Words:** 1
- **Cycles:** 1
- **Example:**
  ```assembly```
  movw r21:r20, r1:r0 ; Copy r1:r0 to r21:r20```
  ```assembly```

**Load Immediate**

- **Syntax:** `ldi Rd, k`
- **Operands:** $Rd \in \{r16, r17, \ldots, r31\}$ and $0 \leq k \leq 255$
- **Operation:** $Rd \leftarrow k$
- **Flag affected:** None
- **Encoding:** 1110 kkkk dddd kkkk
- **Words:** 1
- **Cycles:** 1
- **Example:**
  ```assembly```
  ldi r16, $42 ; Load $42 to r16```
  ```assembly```

**Load Indirect**

- **Syntax:** `ld Rd, v`
- **Operands:** $Rd \in \{r0, r1, \ldots, r31\}$ and $v \in \{x, x+, -x, y, y+, -y, z, z+, -z\}$
- **Operation:**
  1. $Rd \leftarrow (v)$ if $v \in \{x, y, z\}$
  2. $x \leftarrow x-1$ and $Rd \leftarrow (x)$ if $v =-x$
  3. $y \leftarrow y-1$ and $Rd \leftarrow (y)$ if $v =-y$
  4. $z \leftarrow z-1$ and $Rd \leftarrow (z)$ if $v =-z$
  5. $Rd \leftarrow (x)$ and $x \leftarrow x+1$ if $v =x+$
  6. $Rd \leftarrow (y)$ and $y \leftarrow y+1$ if $v =y+$
  7. $Rd \leftarrow (z)$ and $z \leftarrow z+1$ if $v =z+$
- **Flag affected:** None
- **Encoding:** Depends on $v$. Refer to AVR Instruction Set for details
- **Words:** 1
- **Cycles:** 2
- **Comments:** Post-inc and pre-dec are used to load contiguous data.
Load Indirect (Cont.)

- Example: 4-byte integer addition.

```assembly
def loop_counter = r20
.equ loop_bound = 4
.dseg
  int1: .byte 4 ; Allocate 4 bytes to the first integer;
  int2: .byte 4 ; Allocate 4 bytes to the second integer;
  int3: .byte 4 ; Allocate 4 bytes to store the result
.cseg
  ldi r26, low(int1) ; Load low byte of the address of the 1st int
  ldi r27, high(int1) ; Load high byte of the address of the 2nd int
  ldi r28, low(int2)
  ldi r29, high(int2)
  ldi r30, low(int3)
  ldi r31, high(int3)
  clr loop_counter ; loop_counter=0
```

Load Indirect (Cont.)

- Example: 4-byte integer addition.

```assembly
loop:  ld r0, x+ ; Load the next byte of the 1st int
       ld r1, y+ ; Load the next byte of the 2nd int
       cpi loop_counter, 1 ; Least significant byte?
       breq first_byte
       adc r1, r0 ; Add two bytes with carry
       jmp store
first_byte:  add r1, r0 ; Add two least significant bytes
             store:  st z+, r1 ; Store the result
                    cpi loop_counter, loop_bound ; End of loop?
                    brlt loop
                    ret
```

Load Indirect with Displacement

- Syntax: ldd Rd, v
- Operands: Rd ∈ {r0, r1, ..., r31} and v ∈ {y+q, z+q}
- Operation: Rd ← (v)
- Flag affected: None
- Encoding: Depends on v. Refer to AVR Instruction Set for details
- Words: 1
- Cycles: 2
- Example: clr r31 ; Clear Z high byte
  ldi r30, $60 ; Set Z low byte to $60
  ld r0, Z+ ; Load r0 with data space loc. $60(Z post inc)
  ld r1, Z ; Load r1 with data space loc. $61
  ldi r30, $63 ; Set Z low byte to $63
  ld r2, Z ; Load r2 with data space loc. $63
  ld r3, -Z ; Load r3 with data space loc. $62(Z pre dec)
  ldd r4, Z+2 ; Load r4 with data space loc. $64
- Comments: ldd is used to load an element of a structure.

Store Indirect

- Syntax: st v, Rr
- Operands: Rr ∈ {r0, r1, ..., r31} and v ∈ {x, x+, -x, y, y+, -y, z, z+, -z}
- Operation:
  (i) (v)←Rr if v ∈ {x, y, z}
  (ii) x ← x-1 and (x)←Rr if v = x
       y ← y-1 and (y)←Rr if v = y
       z ← z-1 and (z)←Rr if v = z
  (iii) (x)←Rr and x ← x+1 if v = x+
       (y)←Rr and y ← y+1 if v = y+
       (z)←Rr and z ← z+1 if v = z+
- Flag affected: None
- Encoding: Depends on v. Refer to AVR Instruction Set for details
- Words: 1
- Cycles: 2
- Comments: Post-inc and pre-dec are used to store contiguous data.
**Store Indirect with Displacement**

- **Syntax:** std v, Rr
- **Operands:** Rd∈{r0, r1, ..., r31} and v∈{y+q, z+q}
- **Operation:** (v)←Rr
- **Flag affected:** None
- **Encoding:** Depends on v. Refer to AVR Instruction Set for details
- **Words:** 1
- **Cycles:** 2
- **Example:**
  ```asm
  clr r29 ; Clear Y high byte
  ldi r28, $60 ; Set Y low byte to $60
  st Y+, r0 ; Store r0 in data space loc. $60(Y post inc)
  st Y, r1 ; Store r1 in data space loc. $61
  ldi r28, $63 ; Set Y low byte to $63
  st Y, r2 ; Store r2 in data space loc. $63
  st -Y, r3 ; Store r3 in data space loc. $62 (Y pre dec)
  std Y+2, r4 ; Store r4 in data space loc. $64
  ...
  ```
- **Comments:** std is used to store an element of a structure.

**Load Program Memory**

- **Syntax:**
  - (i) LPM
  - (ii) LPM Rd, Z
  - (iii) LPM Rd, Z+
- **Operands:** Rd
- **Operations:**
  - (i) R0←(Z)
  - (ii) Rd←(Z)
  - (iii) Rd←(Z)
- **Flag affected:** None
- **Encoding:**
  - (i) 1001 0101 1100 1000
  - (ii) 1001 000d dddd 0100
  - (iii) 1001 000d dddd 0101
- **Words:** 1
- **Cycles:** 3
- **Comments:** Z contains the byte address while the flash memory uses word addressing. Therefore, the word address must be converted into byte address before having access to data on flash memory.

**Load Program Memory (Cont.)**

- **Example**
  ```asm
  ldi zh, high(Table_1<<1) ; Initialize Z pointer
  ldi zl, low(Table_1<<1)
  lpm r16, z+ ; r16=0x76
  lpm r17, z ; r17=0x58
  ...
  Table_1: .dw 0x5876
  ...
  ```
- **Comments:** Table_1<<1 converts word address into byte address

**Load Program Memory (Cont.)**

- **Example**
  ```asm
  ldi zh, high(Table_1<<1) ; Initialize Z pointer
  ldi zl, low(Table_1<<1)
  lpm r16, z+ ; r16=0x76
  lpm r17, z ; r17=0x58
  ...
  Table_1: .dw 0x5876
  ...
  ```
- **Comments:** Table_1<<1 converts word address into byte address
Load an I/O Location to Register

- Syntax: `in Rd, A`
- Operands: `Rd ∈ {r0, r1, …, r31}` and `0 ≤ A ≤ 63`
- Operation: `Rd ← I/O (A)`
  - Loads one byte from the location `A` in the I/O Space (Ports, Timers, Configuration registers etc.) into register `Rd` in the register file.
- Flag affected: None
- Encoding: `1011 0AA dddd AAAA`
- Words: 1
- Cycles: 1
- Example:
  ```
  in r25, $16 ; Read Port B
  cpi r25, 4 ; Compare read value to constant
  breq exit ; Branch if r25=4
  ...
  exit: nop ; Branch destination (do nothing)
  ```

Store Register to an I/O Location

- Syntax: `out A, Rr`
- Operands: `Rr ∈ {r0, r1, …, r31}` and `0 ≤ A ≤ 63`
- Operation: `I/O (A) ← Rr`
  - Store the byte in register `Rr` to the I/O location (register).
- Flag affected: None
- Encoding: `1011 1AA rrr rrrr AAAA`
- Words: 1
- Cycles: 1
- Example:
  ```
  clr r16 ; Clear r16
  ser r17 ; Set r17 to $ff
  out $18, r16 ; Write zeros to Port B
  nop ; Wait (do nothing)
  out $18, r17 ; Write ones to Port B
  ```

Push Register on Stack

- Syntax: `push Rr`
- Operands: `Rr ∈ {r0, r1, …, r31}`
- Operation: `(SP) ← Rr
  `SP ← SP –1`
- Flag affected: None
- Encoding: `1001 001 dddd 1111`
- Words: 1
- Cycles: 2
- Example:
  ```
  call routine ; Call subroutine
  ...
  routine: push r14 ; Save r14 on the stack
            push r13 ; Save r13 on the stack
  ...
  pop r13 ; Restore r13
  pop r14 ; Restore r14
  ret ; Return from subroutine
  ```

Pop Register from Stack

- Syntax: `pop Rr`
- Operands: `Rr ∈ {r0, r1, …, r31}`
- Operation: `Rr ← (SP)
  `SP ← SP +1`
- Flag affected: None
- Encoding: `1000 000 dddd 1111`
- Words: 1
- Cycles: 2
- Example:
  ```
  call routine ; Call subroutine
  ...
  routine: push r14 ; Save r14 on the stack
            push r13 ; Save r13 on the stack
  ...
  pop r13 ; Restore r13
  pop r14 ; Restore r14
  ret ; Return from subroutine
  ```