INTRODUCTION TO AVRASSEMBLY PROGRAMMING

Hoda Roodaki
hroodaki@kntu.ac.ir

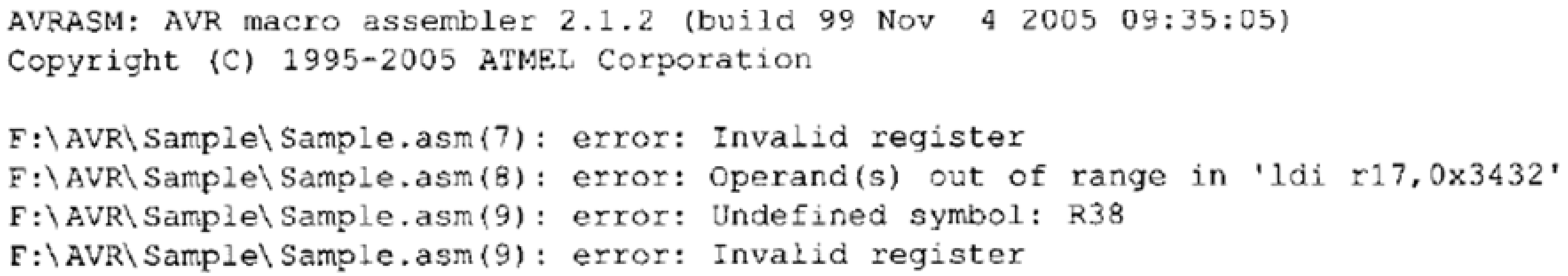
ASSEMBLING AN AVR PROGRAM

- How a basic form of an Assembly language program is created, assembled, and made ready to run?
  - We need a text editor
    - In the case of the AVR microcontrollers, we use the AtmegStudio, which has a text editor, assembler, simulator, and much more all in one software package.
      - It is an excellent development software that supports all the AVR chips and is free.
      - For assemblers, the source file has the extension "asm". The "asm" extension for the source file is used by an assembler in the next step.
      - The "asm" source file containing the program code created in step 1 is fed to the AVR assembler. The assembler produces
        - » an object file (The object file has the extension "obj")
        - » a hex file (the hex file extension is "hex")
        - » an eeprom file (the eeprom file has the extension "eep")
        - » a list file (the list file extension is "lst")
        - » a map file (the map file extension is "map")
      - After a successful link, the hex file is ready to be burned into the AVR's program ROM.

More about asm and object files

- The asm file is also called the *source* file and must have the "asm" extension.
  - This file is created with a text editor such as Windows Notepad.
  - The assembler converts the asm file's Assembly language instructions into machine language and provides the obj (object) file.
  - The object file is used as input to a simulator.
    - Before we can assemble a program to create a ready-to-run program, we must make sure that it is error free.
    - The AtmegStudio provides us error messages and we examine them to see the nature of syntax errors.
    - The assembler will not assemble the program until all the syntax errors are fixed.

More about asm and object files

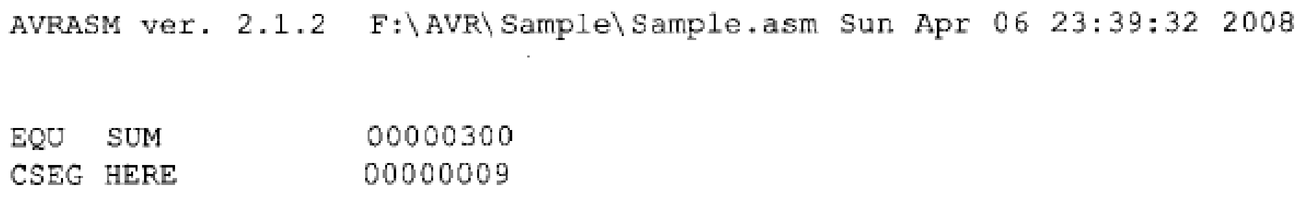


```
AVRASM: AVR macro assembler 2.1.2 (build 99 Nov  4 2005 09:35:05)
Copyright (C) 1995-2005 ATMEL Corporation

F:\AVR\Sample\Sample.asm(7): error: Invalid register
F:\AVR\Sample\Sample.asm(8): error: Operand(s) out of range in 'ldi r17,0x3432'
F:\AVR\Sample\Sample.asm(9): error: Undefined symbol: R38
F:\AVR\Sample\Sample.asm(9): error: Invalid register
```

“lst" and "map" files

- The map file shows the labels defined in the program together with their values.

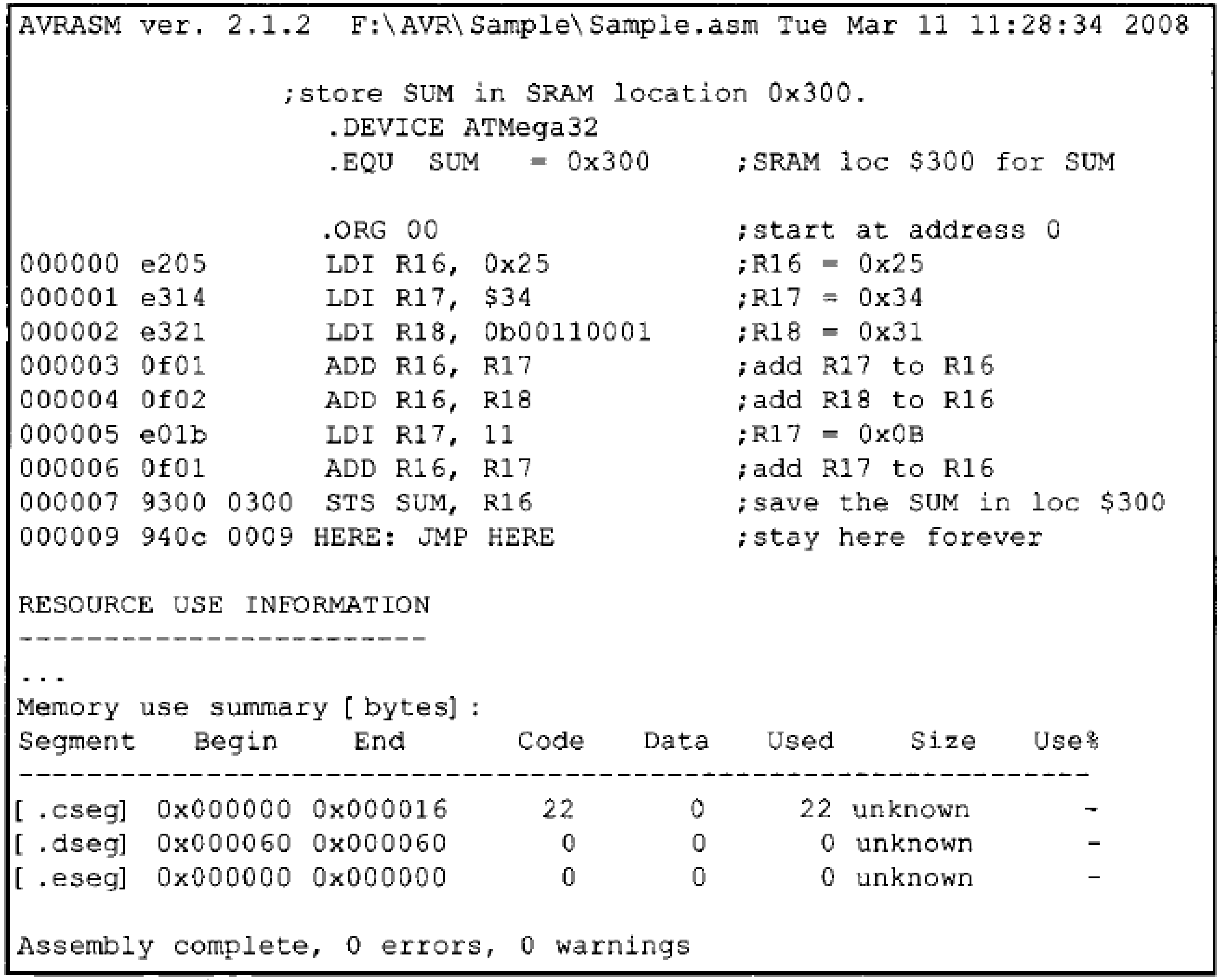


```
AVRASM ver. 2.1.2  F:\AVR\Sample\Sample.asm Sun Apr 06 23:39:32 2008


EQU  SUM          00000300
CSEG HERE         00000009
```

- The lst (list) file, which is optional, is very useful to the programmer.
  - The list shows the binary and source code.
  - It shows which instructions are used in the source code.
  - It shows the amount of memory the program uses.

“lst" and "map" files



```
AVRASM ver. 2.1.2  F:\AVR\Sample\Sample.asm Tue Mar 11 11:28:34 2008

                  ;store SUM in SRAM location 0x300.
                     .DEVICE ATMega32
                     .EQU  SUM   = 0x300      ;SRAM loc $300 for SUM

                     .ORG 00                  ;start at address 0
000000 e205          LDI R16, 0x25            ;R16 = 0x25
000001 e314          LDI R17, $34             ;R17 = 0x34
000002 e321          LDI R18, 0b00110001      ;R18 = 0x31
000003 0f01          ADD R16, R17             ;add R17 to R16
000004 0f02          ADD R16, R18             ;add R18 to R16
000005 e01b          LDI R17, 11              ;R17 = 0x0B
000006 0f01          ADD R16, R17             ;add R17 to R16
000007 9300 0300     STS SUM, R16             ;save the SUM in loc $300
000009 940c 0009 HERE: JMP HERE               ;stay here forever

RESOURCE USE INFORMATION
------------------------
...
Memory use summary [bytes]:
Segment    Begin    End       Code   Data   Used    Size   Use%
---------------------------------------------------------------
[.cseg] 0x000000 0x000016      22      0     22 unknown     -
[.dseg] 0x000060 0x000060       0      0      0 unknown     -
[.eseg] 0x000000 0x000000       0      0      0 unknown     -

Assembly complete, 0 errors, 0 warnings
```

“lst" and "map" files

- Many assemblers assume that the list file is not wanted unless you indicate that you want to produce it.
- The programmer uses the list and map files to ensure correct system design.

THE PROGRAM COUNTER IN THE AVR

- The most important register in the AVR microcontroller is the PC (program counter).
- The program counter is used by the CPU to point to the address of the next instruction to be executed.
- As the CPU fetches the opcode from the program ROM, the program counter is incremented automatically to point to the next instruction.
  - The wider the program counter, the more memory locations a CPU can access.
    - That means that a 14-bit program counter can access a maximum of 16K (2^{14} = 16K) program memory locations.

THE PROGRAM COUNTER IN THE AVR

- In AVR microcontrollers each code memory location is 2 bytes wide.
  - For example, in ATmega32, whose code memory is 32K bytes, the memory is organized as 16Kx 16.
  - Its program counter is 14 bits wide (2^{14} = 16K memory locations).
  - The ATmega64 has a 15-bit program counter, so its code memory has 32K locations (2^{15}=32K), with each location containing 2 bytes (32K x 2 bytes= 64K bytes).

ROM memory map in the AVR family

- Some family members have only a few kilobytes of on-chip ROM and some, such as the ATmega128, have 128K of ROM.
  - The point to remember is that no member of the AVR family can access more than 4M words of opcode because the program counter in the AVR can be a maximum of 22 bits wide (000000 to $3FFFFF address range).
- It must be noted that while the first location of program ROM inside the AVR has the address of 000000, the last location can be different depending on the size of the ROM on the chip.

Example

Find the ROM memory address of each of the following AVR chips:
(a) ATtiny25 with 2 KB
(b) ATmega16 with 16 KB
(c) ATmega64 with 64 KB

Solution:

(a) With 2K bytes of on-chip ROM memory, we have 2048 bytes (2 × 1024 = 2048). As each address location in AVR is 2 bytes, its Flash has 1024 locations (2048 / 2 = 1024). This maps to address locations of 0000 to $03FF. Notice that 0 is always the first location.
(b) With 16K bytes of on-chip ROM memory, we have 16,384 bytes (16 × 1024 = 16,384), and 8192 locations (16384 / 2 = 8192), which gives 0000–$1FFF.
(c) With 64K we have 65,535 bytes (64 × 1024 = 65,535), and 32,768 locations. Converting 32,768 to hex, we get $8000; therefore, the memory space is 0000 to $7FFF.

ROM memory map in the AVR family

- Where the AVR wakes up when it is powered up?
- At what address does the CPU wake up when power is applied?
- For AVR microcontrollers the microcontroller wakes up at memory address 0000 when it is powered up.
  - When the AVR is powered up, the PC (program counter) has the value of 00000 in it.
  - This means that it expects the first opcode to be stored at ROM address $00000.
  - For this reason, in the AVR system, the first opcode must be burned into memory location $00000 of program ROM because this is where it looks for the first instruction when it is booted.
  - We achieve this by using the .ORG statement in the source program. Next we discuss the step-by-step action of the program counter in fetching and executing a sample program.

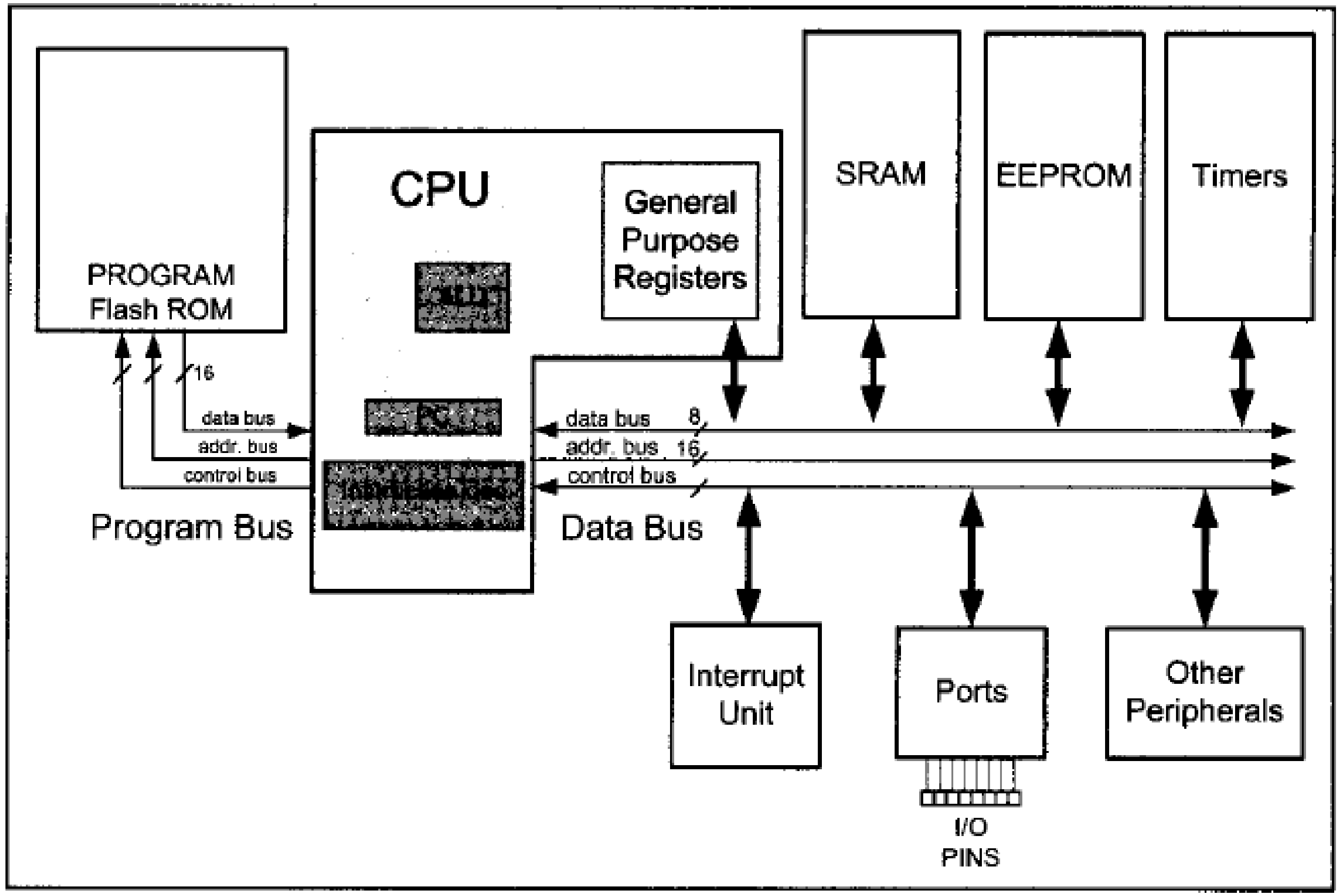
Harvard architecture in the AVR

- AVR uses Harvard architecture
  - there are separate buses for the code and the data memory.
    - The Program Bus provides access to the Program ROM.
    - The Data Bus is used for bringing data to the CPU.

ROM width in the AVR

- Each location of the address space holds two bytes (a word).
- If we have 16 address lines, this will give us 2^{16} locations, which is 64K of memory location with an address map of 0000-FFFFH.

Harvard architecture in the AVR



Harvard architecture in the AVR

- In the Program Bus
  - The data bus is 16 bits wide
  - The address bus is as wide as the PC register to enable the CPU to address the entire Program ROM.
- In the Data Bus
  - The data bus is 8 bits wide
    - The CPU can access one byte of data at a time
  - The address bus is 16 bits wide.
    - Thus the data memory space can be up to 64K bytes

Example

- Data memory space and how to use the STS and LDS instructions.
  - Execute the "LDS Rn,k" instruction,
  - CPU puts k on the address bus of the Data Memory,
  - Receives data through the data bus.
    - For example, to execute "LDS R20, 0x90'',
    - The CPU puts 0x90 on the address bus.
    - The location $90 is in the SRAM.
    - The SRAM puts the contents of location $90 on the data bus.
    - The CPU gets the contents of location $90 through the data bus and puts it in R20.

Example

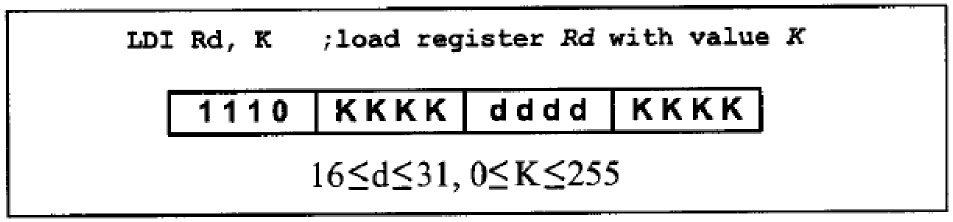
- The "STS k,Rn" instruction is executed similarly.
  - The CPU puts k on the address bus and the contents of Rn on the data bus.
  - The unit whose address is on the address bus receives the contents of data bus.
    - For example, to execute the "STS $100,R30" instruction the CPU puts the contents of R30 on the data bus and $100 on the address bus.
    - Because $100 is bigger than $60, the address belongs to SRAM; thus SRAM gets the contents of the data bus and puts it in location $100 of the SRAM.

Little endian vs. big endian

- Little Endian
  - The low byte goes to the low memory location
  - The high byte goes to the high memory address
- Big Endian
  - The high byte goes to the low address
  - The low byte goes to the high address
- The placing of the code in the AVR ROM
  - Little Endian

Instruction size of the AVR

- The AVR instructions
  - either 2-byte or 4-byte.
  - Almost all the instructions in the AVR are 2-byte instructions.
    - The exceptions are STS, JMP, and a few others.
      - LDI instruction formation
      - The LDI is a 2-byte (16-bit) instruction.
        - » The first 4 bits are set aside for the opcode
        - » The second and the fourth 4 bits are used for the value of 00 to $FF
        - » The third 4 bits present the destination register.

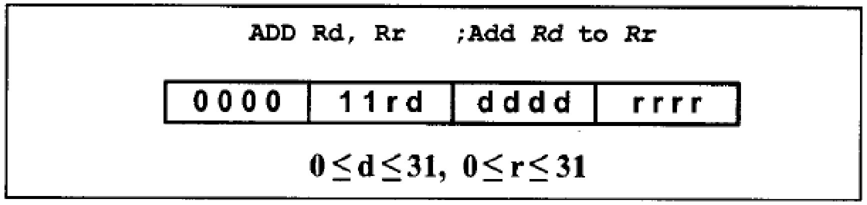


```
LDI Rd, K    ;load register Rd with value K
```

1110	KKKK	dddd	KKKK

$16 \le d \le 31,\ 0 \le K \le 255$

Instruction size of the AVR

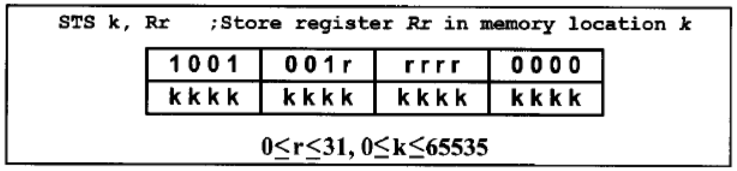


ADD Rd, Rr ;Add *Rd* to *Rr*

0000	11rd	dddd	rrrr

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

- The AVR instructions
  - ADD instruction formation
    - A 2-byte (16-bit) instruction.
      - The first 6 bits are set aside for the opcode,
      - The other 10 bits represent the source and the destination registers.
  - STS instruction formation
    - A 4-byte (32-bit) instruction.
      - The first 16 bits are set aside for the opcode and the address of the source,
      - The other 16 bits are used for the address of the destination.



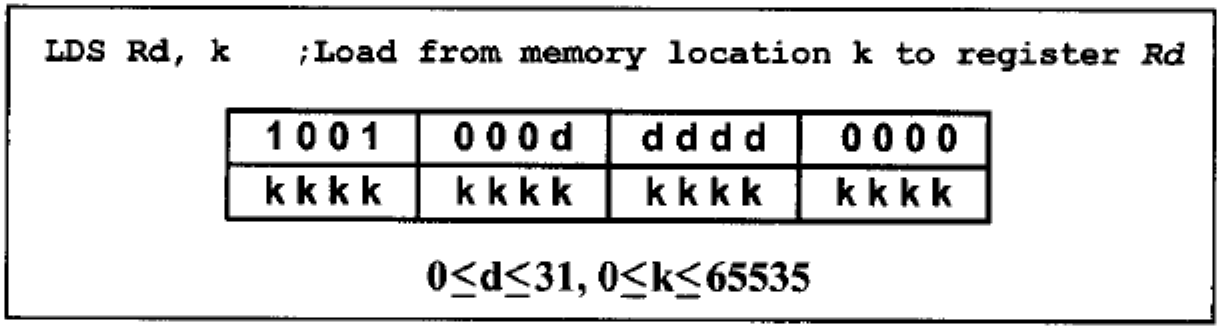
STS k, Rr ;Store register *Rr* in memory location *k*

1001	001r	rrrr	0000
kkkk	kkkk	kkkk	kkkk

$0 \leq r \leq 31,\ 0 \leq k \leq 65535$

Instruction size of the AVR

- The AVR instructions
  - LDS instruction formation
    - A 4-byte (32-bit) instruction.
      - The first 16 bits are set aside for the opcode and the destination register.
      - The other 16 bits are used for the address of the source memory location.



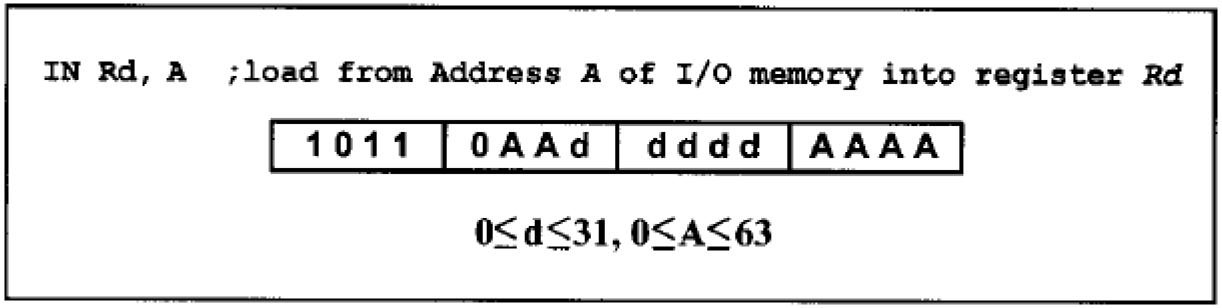
```
LDS Rd, k    ;Load from memory location k to register Rd
```

1001	000d	dddd	0000
kkkk	kkkk	kkkk	kkkk

$0 \leq d \leq 31, 0 \leq k \leq 65535$

Instruction size of the AVR

- The AVR instructions
  - IN instruction formation
    - A 2-byte (16-bit) instruction.
      - The first 5 bits are set aside for the opcode.
      - The other 11 bits are used for the address of the source memory location, and destination register.



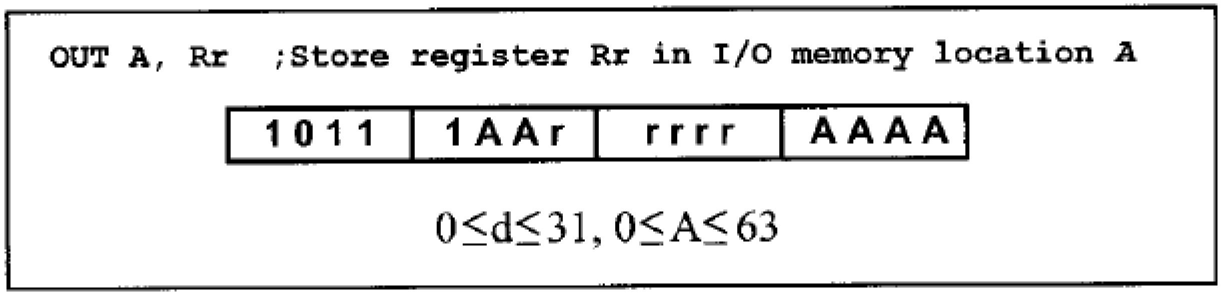
```
IN Rd, A  ;load from Address A of I/O memory into register Rd
```

1011	0AAd	dddd	AAAA

$0 \leq d \leq 31, 0 \leq A \leq 63$

Instruction size of the AVR

- The AVR instructions
  - OUT instruction formation
    - A 2-byte (16-bit) instruction.
      - The first 5 bits are set aside for the opcode.
      - The other 11 bits are used for the address of the source memory location and destination register.



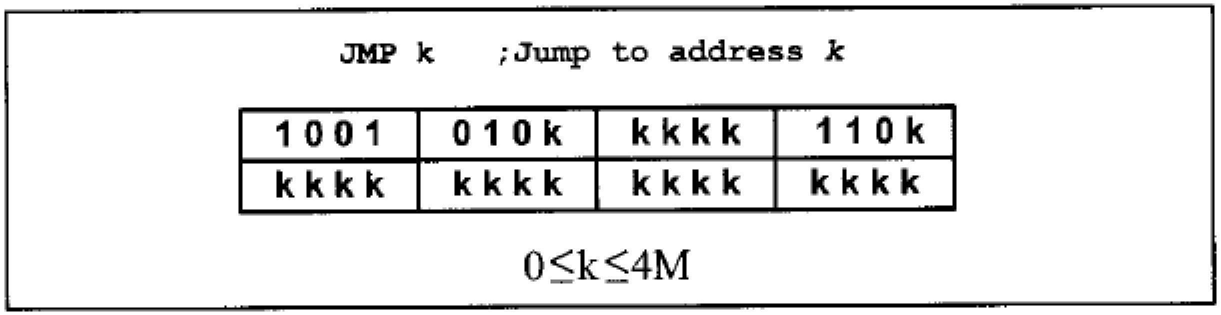
OUT A, Rr ;Store register Rr in I/O memory location A

1011	1AAr	rrrr	AAAA

$0 \leq d \leq 31, 0 \leq A \leq 63$

Instruction size of the AVR

- The AVR instructions
  - JMP instruction formation
    - A 4-byte (32-bit) instruction.
      - Only 10 bits are set aside for the opcode,
      - The rest (22 bits) are used for the target address of the JMP.
        - » The 22-bit address gives us 4M of address space; so, it can address all of the ROM space



```
JMP k    ;Jump to address k
```

1001	010k	kkkk	110k
kkkk	kkkk	kkkk	kkkk

$0 \leq k \leq 4M$