AVR Cheatsheet (ELEC1601)

Abyan Majid

Win: `CTRL + F` * Mac: `CMD + F`
AVR simulator: https://jonopriestley.github.io/avrsim/

Concepts

Labels
A string that can be used to encode addresses of data or code (typically ends with a colon ":", and each label must be unique)

Directives
Words that are not translated into code (typically starts with a dot ".")

Instruction
An operation that controls the behaviour of the microcontroller's CPU.

X, Y, Z Pointers

- **Z:** Registers R31:R30
  - **Z+** post-increments value stored at registers R31:R30
  - **-Z** pre-decrements value stored at registers R31:R20
- **Y:** Registers R29:R28
  - **Y+** post-increments value stored at registers R29:R28
  - **-Y** pre-decrements value stored at registers R29:R28
- **X:** Registers R27:R26
  - **X+** post-increments value stored at registers R27:R26
  - **-X** pre-decrements value stored at registers R27:R26

Stack pointer (SP)
Pointer that references the top of the stack

Subroutine
A labelled section of code that performs a specific task (like functions)

Status registers

- **I (Global interrupt-flag) (Bit 7):** Set to `I=1` if the interrupts are allowed by the microcontroller. Otherwise, it is set to `I=0`
- **T (Test-flag) (Bit 6):** Reserved for test instructions and is not applicable in most cases
- **H (Half carry-flag) (Bit 5):** Set to `H=1` if there is a carry from the lower 4 bits to the higher 4 bits in an arithmetic operation done in binary. Otherwise, it is cleared `H=0`
- **S (Sign-flag) (Bit 4):** Set to `S=1` if the result of an operation is negative. Otherwise, it is cleared `S=0`
- **V (Overflow-flag) (Bit 3):** Set to `V=1` if an arithmetic operation results in a 2s complement overflow. Otherwise, it is cleared `V=0`
- **N (Negative-flag) (Bit 2):** Set to `N=1` if the result of an operation is negative. Otherwise, or if there is no arithmetic operation, it is cleared `N=0`
- **Z (Zero-flag) (Bit 1):** Set to `Z=1` if the result of an operation is `0`. Otherwise, it is cleared `Z=0`. It is often used to store the result of a boolean expression, therefore very useful for control flow!
- **C (Carry-flag) (Bit 0):** Set to `C=1` if the result of an arithmetic operation produces a carry of the most significant bit. Otherwise, it is cleared `C=0`

Directives

.end
Ends the program

.section .data
A mark that tells the assembler that text appearing afterwards are data definitions.

.section .text
A mark that tells the assembler of the end of data definitions and that every text afterwards is an instruction.

.byte
Defines numbers each sized 1 byte

.ascii
    Defines a string, where each character is 1 byte and is encoded in ASCII

.asciz
    Does what `.ascii` does but an extra byte of value `0x00` is appended to the end of the string.

.string
    synonym of `.asciz`

.space <b>, <n>
    Reserves a space of `<b>` bytes, each of which stores the value `<n>`

.space <b>
    Reserves a space of `<b>` bytes, each of which stores 0 by default.

Instructions

PUSH - Push register on stack

`PUSH <Rd>`
    Copies the value of `<Rd>` and stores it at the top of the stack, then post-decrements SP.

POP - Pop register from stack

`POP <Rd>`
    Pre-increments SP and stores into `<Rd>` and removes the value pointed by SP

MOV - Copy register

`MOV <Rd>, <Rr>
    Copies the value stored in `<Rr>` into `<Rd>` without clearing `<Rr>`

hi8 - Get 8 most significant bits

`hi8(<label>)`
    Returns the 8 most significant bits of the address of `<label>`

lo8 - Get 8 least significant bits

`lo8(<label>)`
    Returns the 8 least significant bits of the address of `<label>`

LDI - Load immediate

`LDI <Rd>, <q>`
    Loads into `<Rd>` an immediate value `<q>`

LDS - Load direct from data space

`LDS <Rd>, <label>`
    Loads into `<Rd>` one byte at address `<label>` from the data space

LD - Load indirect from data space

`LD <Rd>, <pointer>`
    Loads into `<Rd>` the value at an address in data space that is stored in the register pairs referenced by `<pointer>`, where `<pointer>` is either `X`, `Y`, or `Z`.

LDD - Load indirect from data space with displacement

`LDD <Rd>, <pointer+q>`
    Loads into `<Rd>` the value at `address + <q>` in data space, where `<q>` is an immediate value, and address is stored in the register pairs referenced by `<pointer>`, where `<pointer>` is either `X`, `Y`, or `Z`.

ST - Store indirect to data space

`ST <pointer>, <Rd>`
    Stores the value of `<Rd>` to an address in data space that is stored by the register pairs referenced by `<pointer>`, where `pointer` is either `X`, `Y`, or `Z`

STS - Store direct to data space

`STS <label>, <Rd>`
    Stores one byte from `<Rd>` into the data space at address `<label>`

STD - Store indirect to data space with displacement

`STD <pointer+q>, <Rd>`
    Stores the value of `<Rd>` to `address + <q>` in data space, where `<q>` is an immediate value, and address is stored in the register pairs referenced by `<pointer>`, where `pointer` is either `X`, `Y`, or `Z`

RET - Return from subroutine

`RET`
    Return from subroutine (exit subroutine)

CLR - Clear register

`CLR <Rd>`
    Clears register `<Rd>`

CALL - Call to a subroutine

`CALL <label>`
    Enter subroutine `<label>`

CP - Compare

`CP <Rd>, <Rr>`
    Compare the value in `<Rd>` with `<Rr>`. This instruction is often followed by a branch with some condition.

CPI - Compare with immediate

`CPI <Rd>, <q>`
    Compare the value in `<Rd>` with some immediate `<q>`. This instruction is often followed by a branch with some condition.

BCLR - Bit clear in SREG

`BCLR <bit@SREG>`

Clears the `<bit@SREG>`-th status register (i.e. status = 0), where `<bit@SREG>` is the bit with which the status register is denoted in the SREG file.

BCLR - Bit set in SREG

`BCLR <bit@SREG>`

Set the `<bit@SREG>`-th status register (i.e. status = 1), where `<bit@SREG>` is the bit with which the status register is denoted in the SREG file.

BRBC - Branch if bit in SREG is cleared

`BRBC <bit@SREG>`

Branch to `<label>` iff the `<bit@SREG>`-th status register is cleared (flag = 0), where `<bit@SREG>` is the bit with which the status register is denoted in the SREG file

BRBS - Branch if bit in SREG is set

`BRBS <bit@SREG>, <label>`

Branch to `<label>` iff the `<bit@SREG>`-th status register is set (flag = 1), where `<bit@SREG>` is the bit with which the status register is denoted in the SREG file

BRCC - Branch if carry is cleared

`BRCC <label>`

Branch to `<label>` iff the carry flag is cleared (C = 1), often used after an arithmetic operation.

BRCS - Branch if carry is set

`BRCS <label>`

Branch to `<label>` iff the carry flag is set (C = 1), often used after an arithmetic operation.

BRHC - Branch if half carry flag is cleared

`BRHC <label>`

Branch to `<label>` if the half carry flag is cleared (H = 0)

BRHS - Branch if half carry flag is set

`BRHS <label>`

Branch to `<label>` if the half carry flag is set (H = 1)

BRID - Branch if global interrupt is disabled

`BRID <label>`

Branch to `<label>` if global interrupts are disabled (I = 0)

BRIE - Branch if global interrupt is enabled

`BRIE <label>`

Branch to `<label>` if global interrupts are enabled (I = 1)

BRMI - Branch if minus

`BRMI <label>`

Branch to `<label>` iff the negative flag is set (N = 1)

BRPL - Branch if plus

`BRPL <label>`

Branch to `<label>` iff the negative flag is cleared (N = 0)

BRTC - Branch if test flag is cleared

`BRTC <label>`

Branch to `<label>` iff the test flag is cleared (T = 0)

BRVC - Branch if overflow is cleared

`BRVC <label>`

Branch to `<label>` iff the overflow flag is cleared (V = 0)

BRVS - Branch if overflow is set

`BRVS <label>`

Branch to `<label>` iff the overflow flag is set (V = 1)

BRTS - Branch if test flag is set

`BRTS <label>`

Branch to `<label>` iff the test flag is set (T = 1)

BREQ - Branch if equal

`BREQ <label>`

Branch to `<label>` iff `Rd = Rr`. In terms of status registers, it branches if the zero flag is set (Z = 1).

BRNE - Branch if not equal

`BRNE <label>`

Branch to `<label>` iff `Rd ≠ Rr`. In terms of status registers, it branches if the zero flag is cleared (Z = 0).

BRGE - Branch if greater or equal (Signed)

`BRGE <label>`

Branch to `<label>` iff `Rd ≥ Rr` (SIGNED). In terms of status registers, it branches if the sign flag is cleared (S = 0).

BRLT - Branch if less than (Signed)

`BRLT <label>`

Branch to `<label>` iff `Rd < Rr` (SIGNED). In terms of status registers, it branches if the carry flag is set (S = 1)

BRSH - Branch if same or higher (Unsigned)

`*BRSH <label>*`

Branch to `<label>` iff `Rd ≥ Rr` (UNSIGNED). In terms of status registers, it branches if the carry flag is cleared (C = 0).

BRLO - Branch if lower (Unsigned)

`*BRLO <label>*`

Branch to `<label>` iff `Rd < Rr` (UNSIGNED). In terms of status registers, it branches if the carry flag is set (S = 1)

INC - Increment value in register

`*INC <Rd>*`

Increments value in register `<Rd>` i.e. `Rd = Rd + 1`

Snippets for Invoking Subroutine

(a) Pass parameters and return result via registers

Calling Program

```
LDS R25, x      ; First param
LDS R24, y      ; Second param
CALL subroutine
MOV Y+, R25     ; Store result in
MOV Y+, R24     ; address in Y
```

Invoked Subroutine

```
subroutine:
        ...
        MOV R5, R25     ; first param
        MOV R4, R24     ; second param
        ...
        LDI R25, 0xFF   ; result
        LDI R24, 0xFF   ; result
        RET
```

(b) Pass parameters and return result via memory/data space

Calling Program

```
...
STS p1, R25     ; Set first param
STS p2, R24     ; Set second param
CALL subroutine
LDI R28, lo8(result)
LDI R29, hi8(result)
LD R24, Y       ; Get result
LDD R25, Y+1
...
```

Invoked Subroutine

```
subroutine:
        ...
        LDS R19, p1  ; Get first param
        LDS R18, p2  ; Get second param
        ...
        ...
        ...
        LDI R28, lo8(result)
        LDI R29, hi8(result)
        ST  Y, R24   ; Set result
        STD Y+1, R25
        RET
```

(c) Pass parameters and return result via the stack

Calling Program	Invoked Subroutine
... PUSH R1 ; Result space PUSH R19 ; Second param PUSH R18 ; First param CALL subroutine POP R0 ; discard POP R0 ; discard POP R20 ; Get result ...	subroutine: IN R31, 0x3E ; Z <- SP IN R30, 0x3D ... LDD R18, Z + 3 ; Get first p. LDD R19, Z + 4 ; Get second p. ... ... STD Z + 5, R20 ; Set result OUT 0x3E, R31 ; Optional! OUT 0x3D, R30 ; SP <- Z RET