The Internet of Things: Arduino and Assembly

Dr Joshua Ellul

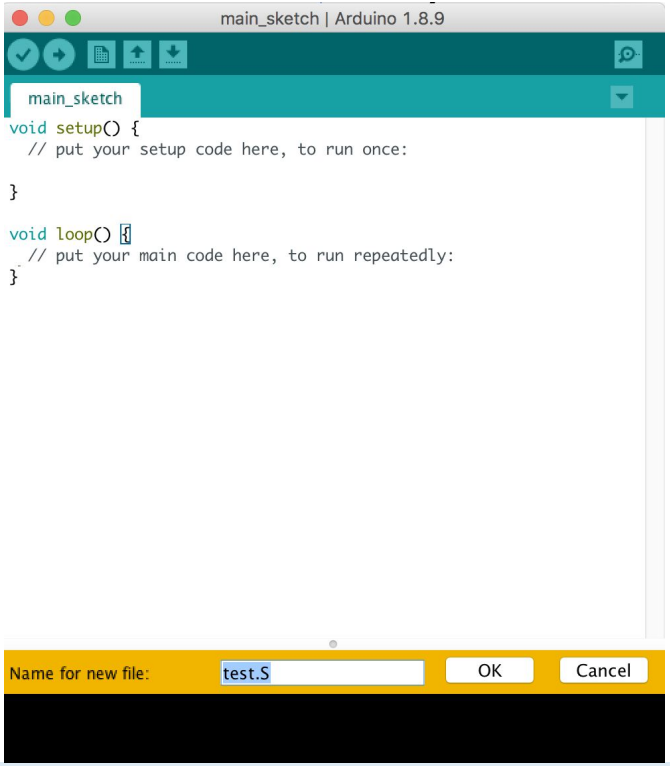
Assembly Docs

- AVR Instruction Set Manual
  - http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf

- Atmel AT1886: Mixing Assembly and C with AVRGCC
  - http://ww1.microchip.com/downloads/en/appnotes/doc42055.pdf

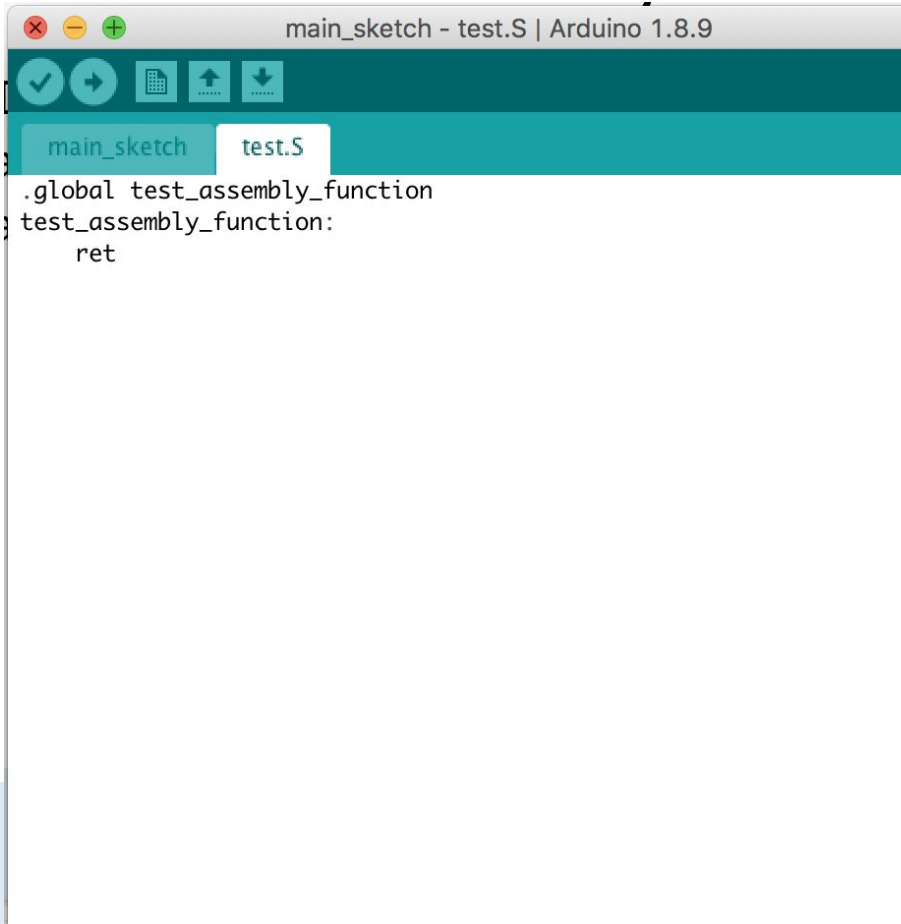
Getting started with Assembly

- In Arduino IDE:
  - Create a new sketch

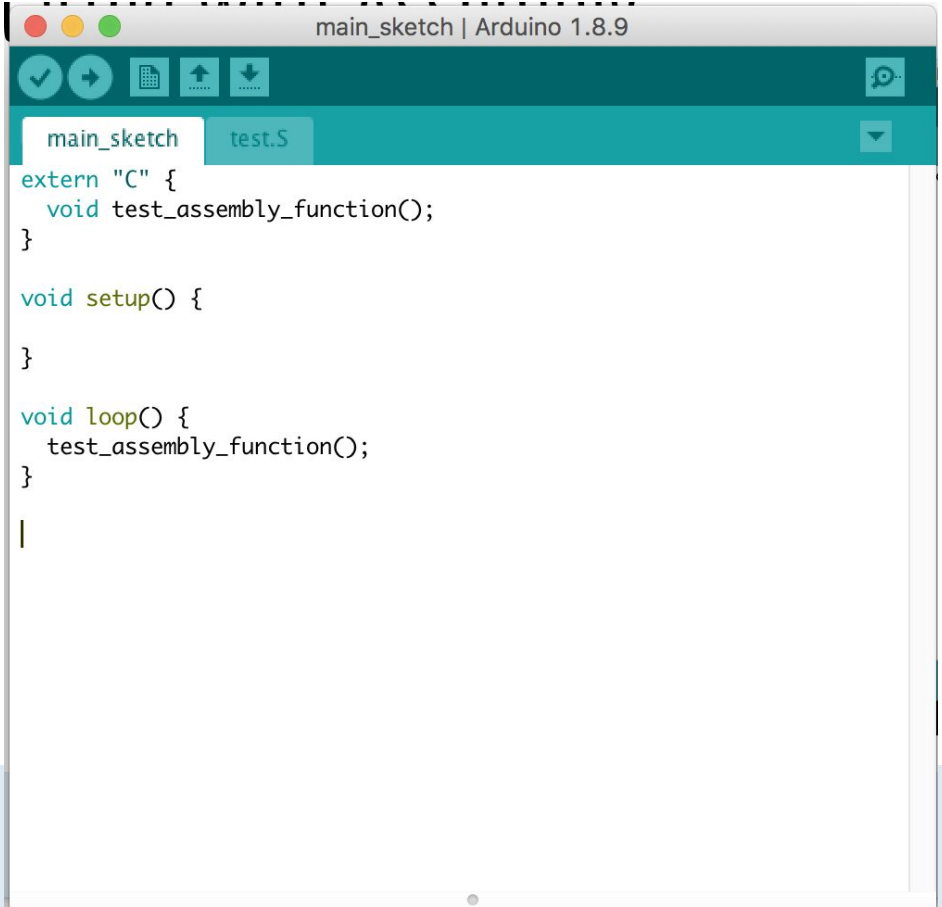


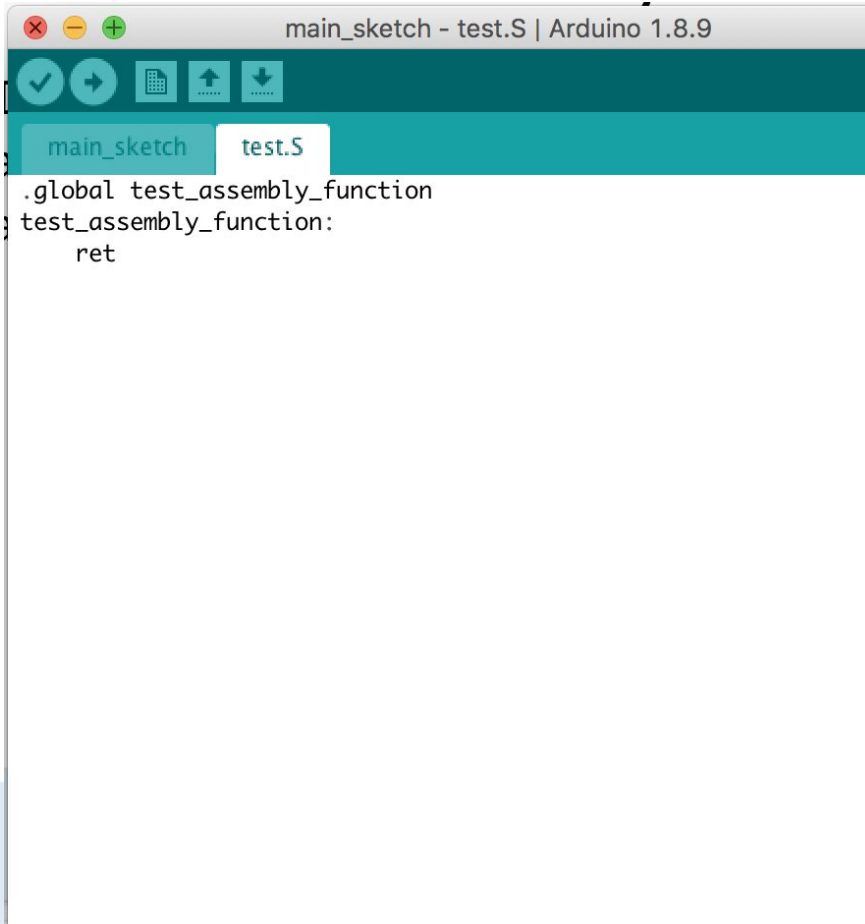
Getting started with Assembly

- In Arduino IDE:
  - Create a new sketch
  - Create a new tab (down arrow > new tab)
    - With a ".S" extension, e.g. test.S



Getting started with Assembly





Getting started with Assembly

Change the code:
- to blink every second
- and then call the assembly test function

AVR Instruction Set

- 32 General Purpose 8-bit registers (R0 to R32)

AVR Instruction Set

- 32 General Purpose 8-bit registers (R0 to R32)
- Special purpose registers:
    - PC

AVR Instruction Set

- 32 General Purpose 8-bit registers (R0 to R32)
- Special purpose registers:
  - PC
  - SP

AVR Instruction Set

- 32 General Purpose 8-bit registers (R0 to R32)
- Special purpose registers:
  - PC
  - SP
  - SREG

Status Register (SREG)

SREG	Status Register
C	Carry Flag
Z	Zero Flag
N	Negative Flag
V	Two's complement overflow indicator
S	N ⊕ V, for signed tests
H	Half Carry Flag
T	Transfer bit used by BLD and BST instructions
I	Global Interrupt Enable/Disable Flag

AVR Instruction Set

- 32 General Purpose 8-bit registers
  (R0 to R32)
- Special purpose registers:
  - PC
  - SP
  - SREG
- X, Y and Z are names given to register pairs: R27:R26, R29:R28 and R31:R30
  - 16-bit pointer registers
  - Able to point to 16-bit SRAM locations
  - Or into locations in program memory (Z can only do this)
  - Higher Byte:Lower Byte >> <Reg>H:<Reg>L

ldi

- Load Immediate
- Loads an 8-bit constant directly to register 16 to 31
- Page 115 of instruction set reference (at time of writing)

- ldi r16, 255 ; load 255 into r16

ldi

- Load Immediate
- Loads an 8-bit constant directly to register 16 to 31
- *Page 115 of instruction set reference (*at time of writing)

- ldi r16, 255 ; load 255 into r16

73. LDI – Load Immediate

73.1. Description

Loads an 8-bit constant directly to register 16 to 31.

Operation:

(i) Rd ← K

	Syntax:	Operands:	Program Counter:
(i)	LDI Rd,K	16 ≤ d ≤ 31, 0 ≤ K ≤ 255	PC ← PC + 1

16-bit Opcode:

1110	KKKK	dddd	KKKK

73.2. Status Register (SREG) and Boolean Formula

I	T	H	S	V	N	Z	C
–	–	–	–	–	–	–	–

Example:

```
clr r31 ; Clear Z high byte
ldi r30,$F0 ; Set Z low byte to $F0
lpm ; Load constant from Program
; memory pointed to by Z
```

Words	1 (2 bytes)
Cycles	1

mov

- Copy register
- Copy one register’s value into another
- *Page 123 of instruction set reference

- mov r16, r0 ; copy value in r0 to r16

mov

- Copy register
- Copy one register's value into another
- *Page 123 of instruction set reference

- mov r16, r0 ; copy value in r0 to r16

79. MOV – Copy Register

79.1. Description

This instruction makes a copy of one register into another. The source register Rr is left unchanged, while the destination register Rd is loaded with a copy of Rr.

Operation:

(i) Rd ← Rr

	Syntax:	Operands:	Program Counter:
(i)	MOV Rd,Rr	0 ≤ d ≤ 31, 0 ≤ r ≤ 31	PC ← PC + 1

16-bit Opcode:

0010	11rd	dddd	rrrr

79.2. Status Register (SREG) and Boolean Formula

I	T	H	S	V	N	Z	C
–	–	–	–	–	–	–	–

Example:

```
mov r16,r0 ; Copy r0 to r16
call check ; Call subroutine
...
check: cpi r16,$11 ; Compare r16 to $11
...
ret ; Return from subroutine
```

Words	1 (2 bytes)
Cycles	1

clr

- Clear register
- Clears a register's value (zeros it out)
- Page 54 of instruction set reference

- clr r18 ; clear r18

clr

- Clear register
- Clears a register's value (zeros it out)
- *Page 71 of instruction set reference

- clr r18 ; clear r18

43. CLR – Clear Register

43.1. Description

Clears a register. This instruction performs an Exclusive OR between a register and itself. This will clear all bits in the register.

Operation:

(i) Rd ← Rd ⊕ Rd

	Syntax:	Operands:	Program Counter:
(i)	CLR Rd	0 ≤ d ≤ 31	PC ← PC + 1

16-bit Opcode: (see EOR Rd,Rd)

0010	01dd	dddd	dddd

43.2. Status Register (SREG) and Boolean Formula

I	T	H	S	V	N	Z	C
–	–	–	0	0	0	1	–

S 0
Cleared.

V 0
Cleared.

N 0
Cleared.

Z 1
Set.

ser

- Set register (set all bits to 1)
- Sets all bits in a register to 1
- *Page 163 of instruction set reference

- set r18 ; set r18 to 0xFF

ser

- Set register (set all bits to 1)
- Sets all bits in a register to 1
- *Page 163 of instruction set reference

- set r18 ; set r18 to 0xFF

110. SER – Set all Bits in Register

110.1. Description

Loads $FF directly to register Rd.

Operation:

(i) Rd ← $FF

Syntax:	Operands:	Program Counter:
(i) SER Rd	16 ≤ d ≤ 31	PC ← PC + 1

16-bit Opcode:

1110	1111	dddd	1111

110.2. Status Register (SREG) and Boolean Formula

I	T	H	S	V	N	Z	C
–	–	–	–	–	–	–	–

Example:

```
clr r16 ; Clear r16
ser r17 ; Set r17
out $18,r16 ; Write zeros to Port B
nop ; Delay (do nothing)
out $18,r17 ; Write ones to Port B
```

Words	1 (2 bytes)
Cycles	1

ret

- Return from Subroutine
- Sets the PC to the value currently on the Stack (SP)
- Also, removes the top 16-bits from the stack

- *Page 139

ret

- Return from Subroutine
- Sets the PC to the value currently on the Stack (SP)
- Also, removes the top 16-bits from the stack

- *Page 139

92.1. Description

Returns from subroutine. The return address is loaded from the STACK. The Stack Pointer uses a pre-increment scheme during RET.

Operation:

	Operation:	Comment:
(i)	PC(15:0) ← STACK	Devices with 16-bit PC, 128KB Program memory maximum.
(ii)	PC(21:0) ← STACK	Devices with 22-bit PC, 8MB Program memory maximum.

	Syntax:	Operands:	Program Counter:	Stack:
(i)	RET	None	See Operation	SP ← SP + 2, (2 bytes, 16 bits)
(ii)	RET	None	See Operation	SP ← SP + 3, (3 bytes, 22 bits)

16-bit Opcode:

1001	0101	0000	1000

92.2. Status Register (SREG) and Boolean Formula

I	T	H	S	V	N	Z	C
–	–	–	–	–	–	–	–

Example:

```
 call routine ; Call subroutine
 ....
routine: push r14 ; Save r14 on the Stack
 ...
 pop r14 ; Restore r14
 ret ; Return from subroutine
```

Words	1 (2 bytes)
Cycles	4 devices with 16-bit PC
	5 devices with 22-bit PC

Working with Registers

- Mapping names to registers
  var1 = 16 ;name var1 can be used instead of R16
- (avrgcc does not allow names to start with r)

- For pointer access use R26 to R31
- 16-bit-counters are best performed in R25:R24
- Accessing single bits in registers is best done in R16 to R23

Run-time Environment Calling Convention

- How are parameters and return values actually passed under the hood in C? https://gcc.gnu.org/wiki/avr-gcc
- See calling convention section
- Return value:
  - If a single returned byte, then it is returned in R24

Run-time Environment Calling Convention

- How are parameters and return values actually passed under the hood in C? https://gcc.gnu.org/wiki/avr-gcc
- See calling convention section
- Retu
  - **Calling Convention**
    - An argument is passed either completely in registers or completely in memory.
    - To find the register where a function argument is passed, initialize the register number R_n with R26 and follow this procedure:
      1. If the argument size is an odd number of bytes, round up the size to the next even number.
      2. Subtract the rounded size from the register number R_n.
      3. If the new R_n is at least R8 and the size of the object is non-zero, then the low-byte of the argument is passed in R_n. Subsequent bytes of the argument are passed in the subsequent registers, i.e. in increasing register numbers.
      4. If the new register number R_n is smaller than R8 or the size of the argument is zero, the argument will be passed in memory.
      5. If the current argument is passed in memory, stop the procedure: All subsequent arguments will also be passed in memory.
      6. If there are arguments left, goto 1. and proceed with the next argument.
    - Return values with a size of 1 byte up to and including a size of 8 bytes will be returned in registers. Return values whose size is outside that range will be returned in memory.
    - If a return value cannot be returned in registers, the caller will allocate stack space and pass the address as implicit first pointer argument to the callee. The callee will put the return value into the space provided by the caller.
    - If the return value of a function is returned in registers, the same registers are used as if the value was the first parameter of a non-varargs function. For example, an 8-bit value is returned in R24 and an 32-bit value is returned R22...R25.
    - Arguments of varargs functions are passed on the stack. This applies even to the named arguments.

    For example, suppose a function with the following prototype:

    ```
    int func (char a, long b);
    ```

    then

    - a will be passed in R24.
    - b will be passed in R20, R21, R22 and R23 with the LSB in R20 and the MSB in R23.
    - The result is returned in R24 (LSB) and R25 (MSB).

Using Assembly Functionality in C

- Create an assembly file (.S)
- Define your assembly function using the template:

```
.global <fnname>
<fnname>:
    ; code
    ret
```

Using Assembly Functionality in C

- Create an assembly file (.S)
- Define your assembly function using the template:

```
.global <fnname>
<fnname>:
    ; code
    ret
```

- Tell the C compiler about the assembly function by using the following template:

```
extern "C" {
  void <fnname>();
  ...
}
```

Using Assembly Functionality in C

- Create an assembly file (.S)
- Define your assembly function using the template:

```
.global <fnname>
<fnname>:
    ; code
    ret
```

- Tell the C compiler about the assembly function by using the following template:
- Call the assembly function from the C code

```
extern "C" {
  void <fnname>();
  ...
}
```

Putting it all together… outputting an assembly return value to the serial port

- Create assembly file returnchar.S
- Define the return_char function as follows:

```
.global return_char
return_char:
toret = 24
    ldi toret, 'a'
    ret
```

Putting it all together… outputting an assembly return value to the serial port

- Create assembly file returnchar.S
- Define the return_char function as follows:

```
.global return_char
return_char:
toret = 24
    ldi toret, 'a'
    ret
```

- Define the function prototype as follows:

```
extern "C" {
  ...
  void return_char();
  ...
}
```

Putting it all together… outputting an assembly return value to the serial port

- Create assembly file returnchar.S

```
.global return_char
return_char:
toret = 24
    ldi toret, 'a'
    ret
```

- Define the return_char function as follows:

- Define the function prototype as follows:

```
extern "C" {
  ...
  uint8_t return_char();
  ...
}
```

- Now, write code that returns the file over the serial port (as done before in class)

Alter return_char

```
.global return_char
return_char:
toret = 24
tmp = 25
    ldi tmp, 'b'
    mov toret, tmp
    ret
```

Alter return_char

```
.global return_char
return_char:
toret = 24
tmp = 25
    ldi tmp, 'b'
    mov toret, tmp
    ret
```

What is this code doing?
Use the instruction set sheet!

And again.. Alter return_char

```
.global return_char
return_char:
toret = 24
tmp = 25
    ldi tmp, ‘b’
    mov toret, tmp
    clr toret
    ret
```

And again.. Alter return_char

```
.global return_char
return_char:
toret = 24
tmp = 25
    ldi tmp, ‘b’
    mov toret, tmp
    clr toret
    ret
```

What is this code doing?
Use the instruction set sheet!

Integer Formats

- Positive Whole Numbers (+ve Integers)
  - 8-Bit: 0x0 to 0xFF
  - char and byte (on this platform and others) 0x0 to 0xFF
  - Prefer
    - uint8_t for unsigned 8 bits: 0 to 255
    - int8_t for signed 8 bits: -128 to 127

Integer Formats

- Positive Whole Numbers (+ve Integers)
  - 8-Bit: 0x0 to 0xFF
    - char and byte (on this platform and others) 0x0 to 0xFF
    - Prefer
      - uint8_t for unsigned 8 bits: 0 to 255
      - int8_t for signed 8 bits: -128 to 127
  - 16-Bit: 0x0 to 0xFFFF
    - int (on this platform)
    - Prefer
      - uint16_t for unsigned 16 bits: 0 to 65,535
      - int16_t for signed 16 bits: -32,768 to 32,767

Integer Formats

- Positive Whole Numbers (+ve Integers) cont.
  - 32-Bit: 0x0 to 0xFFFFFFFF
    - long int (on this platform)
    - Prefer
      - uint32_t for unsigned 32 bits: 0 to 4,294,967,295
      - int32_t for signed 32 bits:

  - 64-Bit: 0x0 to 0xFFFFFFFFFFFFFFFF
    - long long (on this platform)
    - Prefer
      - uint64_t for unsigned 64 bits: 0 to 18,446,744,073,709,551,615
      - int64_t for signed 64 bits: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
      -
    -

Returning 16-Bit Values from a Function

- uint16_t funcName();

Returning 16-Bit Values from a Function

- uint16_t funcName();
- E.g.:

```
.global return_16bit
return_16bit:
toret1 = 24
toret2 = 25
k = 0x1122
    ldi toret2, hi8(k)
    ldi toret1, lo8(k)
    ret
```

Returning 16-Bit Values from a Function

- uint16_t funcName();
- E.g.:

```
.global return_16bit
return_16bit:
toret1 = 24
toret2 = 25
k = 0x1122
    ldi toret2, hi8(k)
    ldi toret1, lo8(k)
    ret
```

- R24: LSB
- R25: MSB

Converting an Integer to String

- itoa defined in <stdlib.h>
  char* itoa(int val, char* s, int radix)

- val: int (16-bit)
- s: pointer to memory area large enough to hold the conversion
- radix: the base system to use (i.e. 10 for decimal)

- s will contain the integer
- (returns the pointer passed as s)

Converting an Integer to String

```
extern "C" {
  //...
  uint16_t return_16bit();
  //...
}

void setup() {
  //...
  Serial.begin(9600);
}

void loop() {
  uint16_t i16;
  char str[6];
  i16 = return_16bit();
  itoa(i16, str, 10);
  Serial.println(str);
  delay(1000);
}
```

Bitwise Operators

ori/or: Bitwise or

```
.global return_16bit
return_16bit:
_retLSB = 24
_retMSB = 25
op3 = 26
  clr _retLSB
  clr _retMSB
  clr op3
  ori _retLSB, 0x0F
  ori _retMSB, 0xAA
  ori op3, 0xE0
  or _retLSB, op3
  ret
```

ori/or: Bitwise and

```
.global return_16bit
return_16bit:
_retLSB = 24
_retMSB = 25
op3 = 26
     clr _retLSB
     clr _retMSB
     clr op3
     ori _retLSB, 0xFF
     ori _retMSB, 0xFF
     ori op3, 0x55
     and _retLSB, op3
     andi _retMSB, 0xEE
     ret
```

sbr/cbr: set/clear bits in register

```
.global return_16bit
return_16bit:
_retLSB = 24
_retMSB = 25
     clr _retLSB
     clr _retMSB
     sbr _retLSB, 0x33
     ori _retMSB, 0xFF
     cbr _retMSB, 0b10101010
     ret
```

eor: Exclusive Or

```
.global return_16bit
return_16bit:
_retLSB = 24
_retMSB = 25
     clr _retLSB
     clr _retMSB
     sbr _retLSB, 0b00111100
     ori _retMSB, 0b00100100
     eor _retLSB, _retMSB
     ret
```

lsl/lsr: Logical Shift Left/Right

```
.global return_16bit
return_16bit:
_retLSB = 24
_retMSB = 25
     ldi _retLSB, 11
     ldi _retMSB, 14
     lsl _retLSB
     lsr _retMSB
     ret
```

swap: Swap Nibbles

```
.global return_16bit
return_16bit:
_retLSB = 24
_retMSB = 25
     ldi _retLSB, 0xCD
     ldi _retMSB, 0xAB
     swap _retLSB
     swap _retMSB
     ret
```

Arithmetic

add/adc: Add without/with Carry

```
.global return_16bit
return_16bit:
_retLSB = 24
_retMSB = 25
opLSB = 26
opMSB = 27
     ldi _retMSB, 0x01
     ldi _retLSB, 0xFF
     ldi opMSB, 0
     ldi opLSB, 1
     add _retLSB, opLSB
     adc _retMSB, opMSB
     ret
```

sub/sbc: Subtract without/with Carry

```
.global return_16bit
return_16bit:
_retLSB = 24
_retMSB = 25
opLSB = 26
opMSB = 27
     ldi _retMSB, 0x02
     ldi _retLSB, 0xFE
     ldi opMSB, 0
     ldi opLSB, 0xFF
     sub _retLSB, opLSB
     sbc _retMSB, opMSB
     ret
```

movw: Copy Register Word

- Movw
  - Copy register word
  - Make a copy of a register pair’s value into another register pair
  - Page 124 of instruction set reference

movw: Copy Register Word

- Movw
  - Copy register word
  - Make a copy of a register pair's value into another register pair
  - Page 124 of instruction set reference

```
; copy value in r17:r16 into r1:r0
movw r16, r0
```

Jump

```
fn:
     nop
     nop
     nop
skip:
     jmp fn
```

Call

```
callee:
     clr 24
     ret

caller:
     call callee
     ret
```

...

com/neg: One's and Two's Complement

```
.global return_16bit
return_16bit:
_retLSB = 24
_retMSB = 25
     ldi _retLSB, 0x33
     ldi _retMSB, 1
     com _retLSB
     neg _retMSB
     ret
```

asr: Arithmetic Shift Right

```
.global return_16bit
return_16bit:
_retLSB = 24
_retMSB = 25
     ldi _retLSB, 0xFE ;-2
     ldi _retMSB, 0xFE ;-2
     asr _retLSB
     lsl _retMSB
     ret
```

rol: Rotate Left through Carry

```
.global return_16bit
return_16bit:
_retLSB = 24
_retMSB = 25
     ldi _retLSB, 128
     ldi _retMSB, 0
     lsl _retLSB
     rol _retMSB
     ret
```

ror: Rotate Right through Carry

```
.global return_16bit
return_16bit:
_retLSB = 24
_retMSB = 25
     ldi _retLSB, 0
     ldi _retMSB, 1
     lsr _retMSB
     ror _retLSB
     ret
```

mul: Multiply Unsigned

Takes in two 8-bit values (from registers), multiplies them and stores the result in r1:r0 (16-bit)

```
.global return_16bit
return_16bit:
_regop1 = 24
_retLSB = 24
_regop2 = 25
_retMSB = 25
     ldi _regop1, 50
     ldi _regop2, 12
     mul _regop1, _regop2
     movw _retLSB, r0
     ret
```

Back to mul

- Causes incorrect behaviour
- From: https://gcc.gnu.org/wiki/avr-gcc
  - "R1 always contains zero. During an insn the content might be destroyed, e.g. by a MUL instruction that uses R0/R1 as implicit output register. **If an insn destroys R1, the insn must restore R1 to zero afterwards**"

Restoring r1

```
.global return_16bit
return_16bit:
_regop1 = 24
_retLSB = 24
_regop2 = 25
_retMSB = 25
      ldi _regop1, 50
      ldi _regop2, 12
      mul _regop1, _regop2
      movw _retLSB, r0
      clr r1
      ret
```

muls/mulsu: Multiply Signed/Signed with Unsigned

- Same as mul but:
  - muls <signed>, <signed>
  - mulsu <signed>, <unsigned>
    - Only works with r16 to r23

	Syntax:	Operands:	Program Counter:
(i)	MULSU Rd,Rr	$16 \le d \le 23$, $16 \le r \le 23$	PC ← PC + 1

rjmp

```
fn:
    nop
    nop
    nop
skip:
    rjmp fn
```

rcall: relative call

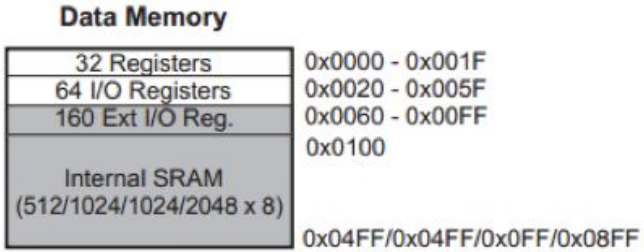
```
callee:
    clr 24
    ret

caller:
    rcall callee
    ret
```

Pointer-Registers

- X, Y, Z
  - 16-bit pointer registers
  - Mapped on R27:R26, R29:R28 and R31:R30 <MSB>:<LSB>
  - ld: load indirect from data space to register
  - st: store indirect from from register to data space

ATmega328P Data Memory Layout

Figure 7-3. Data Memory Map



ATmega328P Registers Memory Layout

Figure 6-2. AVR CPU General Purpose Working Registers