AVR Programming

Why Assembly

- To read and understand assembly code
  - We may need to modify some assembly code only if necessary.
  - But, we are not going to make assembly code from scratch.
- To observe important hardware components
  - Assembly elements reflect hardware components.
- To make high-level programming better fit with hardware
  - High-level coding is compiled into assembly elements.

Tools : IDE

- Integrated Development Environment
  - Chip/manufacture specific
  - Editing
  - Compilation and linking
  - Debugging (software and/or hardware)
  - Uploading
  - Documentation
- Otherwise
  - Text editor
  - Command line compilation
  - Debug as a mind game

Atmel Studio

- Coding and compilation
  - Case2.hello.c
- Debugging
  - Execution control
  - Memory
  - Register
  - I/O
  - Processor
  - Watch
- Documentation
  - Assembly language help
  - Hardware and software help

Assembly vs. C

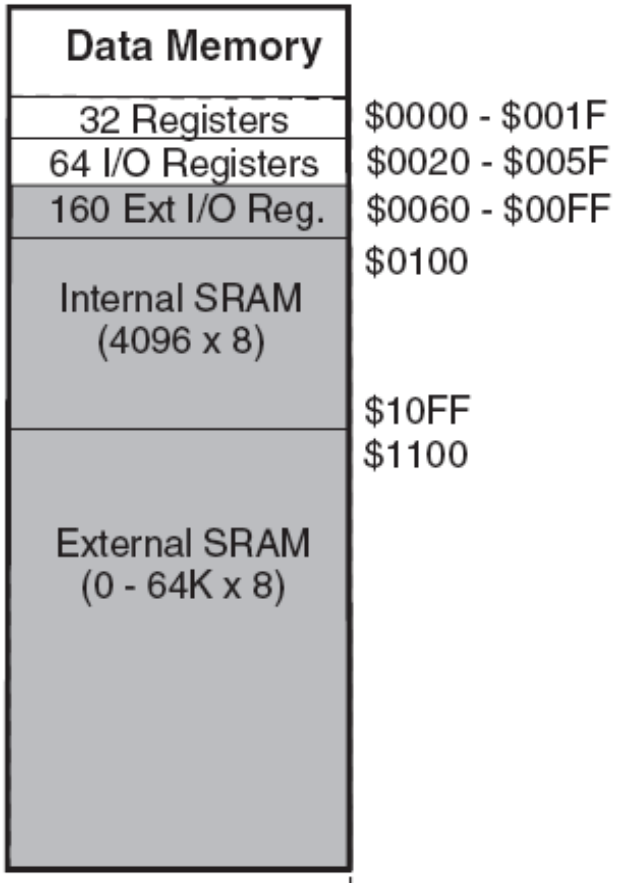
- Difference
  - No variable
  - Any value is either a constant or in a register or at a memory address
  - Any function is a code address
  - Hardware-specific routines must be followed
- Relation via compiler
  - Variable
  - Function

Assembly

- Hardware-specific
  - Register structure
  - I/O structure
  - Addressing
  - Code memory layout
  - Data memory layout
  - Key registers
- Language-specific
  - ISA
  - Directives
  - Expressions

Register Structure

- Also for I/O structure
- 32 general registers
  - Mem address : 0x00-0x1F
- 64 I/O registers
  - Mem address : 0x20-0x5F
  - I/O address : 0x00-0x3F
- 160 Extended I/O registers
  - Mem address : 0x60-0xFF



Key Registers

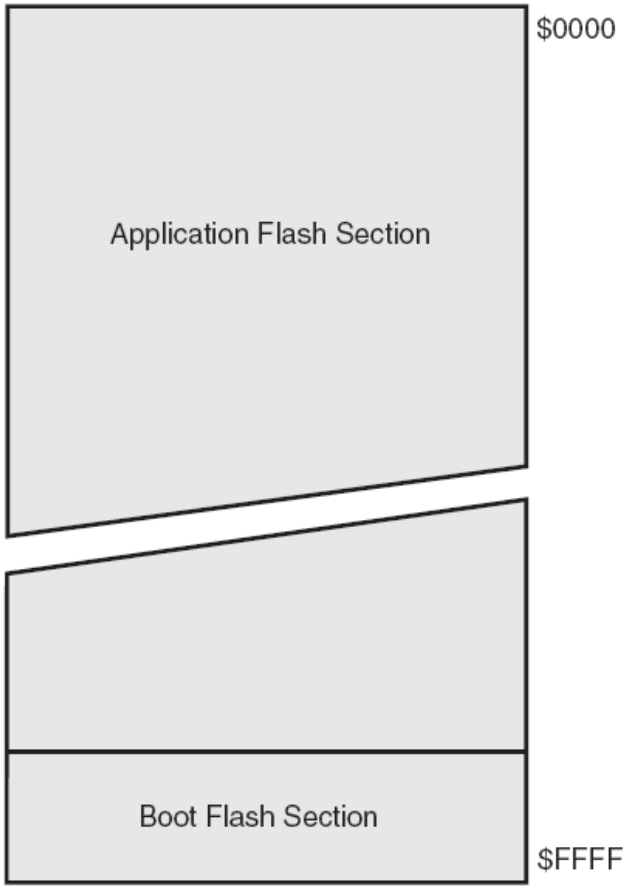
- Status register (SREG) : 1 byte
  - I/O address 0x3F, mem address 0x5F
- Stack pointer (SP) : 2 bytes
  - I/O address 0x3D-3E, mem address 0x5D-5E
- X register : R27:R26
- Y register : R29:R28
- Z register : R31:R30
- R0 : temporary register
- R1 : zero register

Data Memory Layout

- 4Kx1 Internal SRAM
  - Mem address : 0x100-0x10FF
- 64Kx1 Extended SRAM
  - Mem address : 0x1100-0xFFFF
- Layout
  - Structured by the application
  - BSS for global variables
  - Heap for dynamic allocated variables
  - Stack for local variables

Code Memory Layout

- Boot flash section
  - For boot loader
- Application flash section
  - Interrupt vector
  - Configuration data
    - Peripheral initialization
    - Data memory initialization
  - Application code



Assembly Code

- Mnemonic instructions
- Labels
  - [label:] directive [operands] [Comment]
  - [label:] instruction [operands] [Comment]
- Comments
  - ; [Text]
- Directives
- Expressions and functions

ISA

- Addressing
- Instructions
- Case3.avr

Addressing

- Direct access
  - Register direct
  - I/O direct
  - Data direct
- Indirect data space access
  - Data indirect
  - Data indirect with displacement
  - Data indirect with pre-decrement
  - Data indirect with post-increment
- Program access
  - Direct program access
  - Indirect program access
  - Relative program access

Instructions

- Arithmetic and logic instructions
  - **ADD** (SUB), **SUBI** (no ADI), **ADIW**
  - AND, ANDI
  - **COM**, NEG
  - **SBR**, CBR
  - INC, DEC
  - **CLR**, SER
  - **MUL**
  - Flags in SREG (eg. what is V for ADD?)

Instructions

- Branch instructions
  - JMP, **RJMP**, **IJMP**
  - **CALL**, RCALL, ICALL
  - **RET**, **RETI**
  - CP, **CPI**
  - BRXX
  - CPSE, SBXX
  - Range of branch?

Instructions

- Data transfer instructions
  - **LD** (ST), **LDI** (no STI), **LDD** (STD)
  - MOV, **MOVW**
  - **IN**, OUT
  - PUSH, POP
  - Target address?

Instructions

- Bit and bit-test instructions
  - **SBI**, CBI (for I/O registers)
  - **BSET**, BCLR (for SREG)
  - **SEX**, CLX (for SREG's X flag)
  - Target registers?
- MCU control instruction
  - NOP
  - **SLEEP**

Example: Case 4

- s=foo(5);
- 5 is passed to foo as R24:25
- s is Y+1:2 in memory

```
0000007F  LDI R24,0x0A     Load immediate
00000080  LDI R25,0x00     Load immediate
00000081  RCALL PC-0x0032     Relative call subroutine, 4F
00000082  STD Y+2,R25      Store indirect with displacement
00000083  STD Y+1,R24      Store indirect with displacement
```

Example: Case 4

- for (i=0, sum=0; i<5; i++) sum += i;

```
00000058  STD Y+2,R1      Store indirect with displacement
00000059  STD Y+1,R1      Store indirect with displacement
0000005A  STD Y+4,R1      Store indirect with displacement
0000005B  STD Y+3,R1      Store indirect with displacement
0000005C  RJMP PC+0x000E      Relative jump, 6A
0000005D  LDD R18,Y+3     Load indirect with displacement
0000005E  LDD R19,Y+4     Load indirect with displacement
0000005F  LDD R24,Y+1     Load indirect with displacement
00000060  LDD R25,Y+2     Load indirect with displacement
00000061  ADD R24,R18     Add without carry
00000062  ADC R25,R19     Add with carry
00000063  STD Y+4,R25     Store indirect with displacement
00000064  STD Y+3,R24     Store indirect with displacement
00000065  LDD R24,Y+1     Load indirect with displacement
00000066  LDD R25,Y+2     Load indirect with displacement
00000067  ADIW R24,0x01       Add immediate to word
00000068  STD Y+2,R25     Store indirect with displacement
00000069  STD Y+1,R24     Store indirect with displacement
0000006A  LDD R24,Y+1     Load indirect with displacement
0000006B  LDD R25,Y+2     Load indirect with displacement
0000006C  CPI R24,0x05        Compare with immediate
0000006D  CPC R25,R1      Compare with carry
0000006E  BRLT PC-0x11        Branch if less than, signed, 5D
```

Example: Case 4

- for (i=0, sum=0; i<5; i++) sum += i;
- R1 is 0
- i is Y+1:2 in memory
- sum is Y+3:4 in memory
- R24:26 is temporary i in CPU
- R18:19 is temporary sum in CPU
- Y is the current stack top pointer
- Variables in C are referenced at an offset of Y in memory

Example: Case 5

```
  ... :
  c2: 0e 94 67 00    jmp  0xce       ; jump to <main> at 0xce.
  ... :
000000ce <main>:                     ; the stack pointer of <main> is at 0x10FF.
  ce: c7 ef          ldi  r28, 0xF7  ; move up the stack pointer to 0x10F7, because b[8] is at 0x10F8.
  d0: d0 e1          ldi  r29, 0x10  ;
  d2: de bf          out  0x3e, r29  ; store the new stack pointer.
  d4: cd bf          out  0x3d, r28  ;
  d6: 80 e0          ldi  r24, 0x00  ; a[] is at 0x0100
  d8: 91 e0          ldi  r25, 0x01  ; the address of a[] is loaded in r24:25
  da: bc 01          movw r22, r24   ; r22:23 are used as the second parameter of strcpy (a[])
  dc: ce 01          movw r24, r28   ; r24:25 are used as the first parameter of strcpy (b[])
  de: 01 96          adiw r24, 0x01  ;
  e0: 0e 94 76 00    call 0xec       ; call <strcpy> at 0xec.
  e4: 80 e0          ldi  r24, 0x00  ;
  e6: 90 e0          ldi  r25, 0x00  ;
  e8: 0c 94 7d 00    jmp  0xfa       ; jump to <_exit> at 0xfa.
000000ec <strcpy>:                   ; the stack pointer of <strcpy> is at 0x10F5.
  ec: fb 01          movw r30, r22   ; The second parameter (a[])is moved to Z.
  ee: dc 01          movw r26, r24   ; The first parameter (b[]) is moved to X.
  f0: 01 90          ld   r0, Z+     ; load a byte (char) at Z to r0
  f2: 0d 92          st   X+, r0     ; store r0 to X
  f4: 00 20          and  r0, r0     ;
  f6: e1 f7          brne .-8        ; loop until r0 is 0x00.
  f8: 08 95          ret             ;
000000fa <_exit>:
  fa: ff cf          rjmp .-2        ; jump to itself
```

Example: Case 5

```
__vectors:
vector     __vector_1
vector     __vector_2
.....
vector     __vector_35



__init:
clr __zero_reg__
out AVR_STATUS_ADDR, __zero_reg__
ldi r28,lo8(__stack)
ldi r29,hi8(__stack)
out AVR_STACK_POINTER_HI_ADDR, r29

__do_copy_data:
ldi r17, hi8(__data_end)
ldi r26, lo8(__data_start)
ldi r27, hi8(__data_start)
ldi r30, lo8(__data_load_start)
ldi r31, hi8(__data_load_start)
ldi r16, hh8(__data_load_start)
out AVR_RAMPZ_ADDR, r16
rjmp .L__do_copy_data_start
.L__do_copy_data_loop:
elpm r0, Z+
st  X+, r0
.L__do_copy_data_start:
cpi r26, lo8(__data_end)
cpc r27, r17
brne .L__do_copy_data_loop

XJMP main
```

```
__vectors:
jmp L0188  ; 0x0000
jmp L01A5  ; 0x0002
.....
jmp L01A5  ; 0x0044

.DW 0xBAAB ; 0x0046
.....
.DW 0x0081 ; 0x0187

__init:
clr r1     ; 0x0188
out 0x3F, r1     ; 0x0189
ldi r28, 0xF0    ; 0x018A
ldi r29, 0x10    ; 0x018B
out 0x3E, r29    ; 0x018C
out 0x3D, r28    ; 0x018D

__do_copy_data:
ldi r17, 0x1     ; 0x018E
ldi r26, 0x0     ; 0x018F
ldi r27, 0x1     ; 0x0190
ldi r30, low(L11C0*2)  ; 0x0191
ldi r31, high(L11C0*2) ; 0x0192
ldi r16, 0x0     ; 0x0193
out 0x3B, r16    ; 0x0194
rjmp L0198 ; 0x0195
L0196:
elpm r0, Z+      ; 0x0196
st  X+, r0 ; 0x0197
L0198:
cpi r26, 0xA     ; 0x0198
cpc r27, r17     ; 0x0199
brbc 1, L0196    ; 0x019A

jmp L0BF6  ; 0x01A3, to L0BF6
```

Directives

- Directives are NOT opcodes
- Directives are used to
  - Adjust the location of the program in memory
  - Define macros
  - Set data in memory
  - ...
- Directives are used for
  - Convenience
  - Compilation

Directives

- Code memory
  - .CSEG : define the start of a code Segment
  - .DB : define constant byte(s)
  - .DW : define constant word(s)
- Data memory
  - .DSEG : define the start of a data Segment
  - .BYTE : reserve byte(s) to a variable
- Others
  - .SET : set a symbol equal to an expression
  - .DEF : set a symbolic name on a register

Expressions and Functions

- Expressions and functions are constant and evaluated before compilation.
- Expressions can consist of operands, operators and functions.
- Functions
  - LOW(expression): the low byte
  - HIGH(expression): the second byte
  - EXP2(expression): 2 to the power
  - LOG2(expression): the integer part of log2

Example: Case 6

- Move configuration data from code memory to a table in data memory
  - How to use labels, comments, directives, expressions, and functions.
  - How to watch code and data memory
  - How to watch registers
  - How to debug code