CprE 288 – Introduction to Embedded Systems
Atmega128 Assembly Programming: Translating C Control Statements and Function Calls

Instructors:
Dr. Phillip Jones

Announcements

• Final Exam:
  – Morning Section: Tuesday 5/3 (9:45am): 75 minutes
  – Afternoon Section: Friday 5/6 (9:45am) : 75 minutes
• HW11: Due Friday 4/22 Midnight
• Project Demos: Mandatory Demo during you lab section next week (i.e. Deadweek).
• Projects
  – Lab attendance is mandatory
  – For each lab you miss you will lose 10 points on the project (See supplemental specification document)
  – Peer Review: Each team member must submit a Peer review of each of their project partners.

Major Classes of Assembly Instructions

• Data Movement
  – Move data between registers
  – Move data in & out of SRAM
  – Different addressing modes
• Logic & Arithmetic
  – Addition, subtraction, etc.
  – AND, OR, bit shift, etc.
• Control Flow
  – Control which sections of code should be executed (e.g. In C “if”, “CASE”, “WHILE”, etc.
  – Typically the result of Logic & Arithmetic instructions help decided what path to take through the code.

C Control Statements

Recall control statements in C
If statement
if (cond) if-body;
if (cond) if-body else else-body;

How to Evaluate a Condition

Evaluate a simple condition:
1. Have flags set in SREG
2. Branch is taken if certain flag or their combination is true
There are two possible outcomes for a branch: Taken or Not Taken

Example:
LDS r24, a
LDS r26, b
CP r24, r26 ; compare a, b and set flags
BRLT endif ; branch if a < b
endif:
Evaluate Condition

More details:

1. **What instructions set flags in SREG?**
   - Data operation: ADD r24, r22
   - Test: TST r24
   - Compare: CP r24, r22
     CPI r24, 0x0F

2. **Branch condition is evaluated based on the those flags**
   - May Z, N, V, S, C, H or their complement
   - May use a combination of them

---

**Example: How ADD Sets Flags**

ADD – Add two registers without carry

<table>
<thead>
<tr>
<th>I</th>
<th>T</th>
<th>H</th>
<th>S</th>
<th>V</th>
<th>N</th>
<th>Z</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

How does ADD affect the flags:

- N, Z: Set according to the result of ADD, negative or Zero
- V: Set if overflow happens
- S: Set if the actual result is negative, S=N⊕V
- C: Set if carry happens
- H: Set if half carry happens

---

**Example: How ADD Set Flags**

What are the flag values?

```
LDI r16, 0x10
LDI r17, 0x20
ADD r17, r16

LDI r16, 0xFF
LDI r17, 0x01
ADD r17, r16
```

---

**TST: Test a value**

TST – Test for Zero or Minus

<table>
<thead>
<tr>
<th>I</th>
<th>T</th>
<th>H</th>
<th>S</th>
<th>V</th>
<th>N</th>
<th>Z</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TST is a **pseudo instruction**

TST Rd ⇔ AND Rd, Rd

How does AND set the flags:

- N, Z: Set according to the result of AND
- V: Always set to 0
- S: S = N ⊕ V (same as N because V=0)
- I, T, H, C: Not affected

---

**Pseudo Instruction**

Pseudo instruction is not natively supported by the CPU, and not part of the instruction set

Assembler translates pseudo instructions into native ones before generating the binary code

---

**Conditional Branches**

Commonly used branches

- **BREQ**: Equal, signed or unsigned does matter
- **BRNE**: Not Equal, signed or unsigned does matter
- **BRLT**: Less Than, for signed type
- **BRGE**: Greater than or Equal, for signed type
- **BRLO**: Lower than, for unsigned type
- **BRSH**: Same or Higher than, for unsigned type
**CP and CPC: Compare Multiple Registers**

**CP**: Compare  
Syntax: CP Rd, Rr  
Operation: Rd-Rr, PC←PC+1

**CPC**: Compare with Carry  
Syntax: CPC Rd, Rr  
Operation: Rd-Rr-C, PC←PC+1

CP/CPC is like SUB/SBC but only affect the flags

---

**Exercises**

Exercises: Write a sequence of instructions

Branch to label if $a < b$, $a$ and $b$ are variables of “signed char” type

Branch to label if $a >= b$, $a$ and $b$ are variables of "unsigned char" type

Branch to label if $a == b$, $a$ and $b$ are “char” type variables

---

**Exercise**

extern int a, b;  
Branch to label if $a < b$

LDS r24, a  
LDS r25, a+1  
LDS r22, b  
LDS r23, b+1  
CP r24, r22  
CPC r25, r23  
BRLT label

---

**Exercise**

extern unsigned long m, n;  
Branch to label if $m < n$

---

4/19/2016
CPI: Compare with Immediate

Syntax: CPI Rd, K
Operands: 16 ≤ d ≤ 31, 0 ≤ K ≤ 255
Operations: Rd - K, PC ← PC + 1

Branch to label if r24 ≥ 10 (signed type)
CPI r24, 10
BRGE label

Caveat: Instructions with Immediate

Recall all instructions we have learned that use an 8-bit immediate value:
LDI, SUBI, SBCI, ANDI, ORI, CPI, ADIW, SBIW

Constraint for LDI, SUBI, SBCI, ANDI, ORI, CPI
General format: OP Rd, K
Operands: 16 ≤ d ≤ 31, 0 ≤ K ≤ 255
In other words, they only work on R16-R31
Reason: 4-bit OP, 4-bit d, and 8-bit K

Constraint for ADIW, SBIW
They only work on R24, R26, R28 and R30 (d is 2-bit)

Caveat: Instructions with Immediate

Recall all instructions we have learned that use an 8-bit immediate value:
LDI, SUBI, SBCI, ANDI, ORI, CPI, ADIW, SBIW

Constraint for LDI, SUBI, SBCI, ANDI, ORI, CPI
General format: OP Rd, K
Operands: 16 ≤ d ≤ 31, 0 ≤ K ≤ 255
In other words, they only work on R16-R31
Reason: 4-bit OP, 4-bit d, and 8-bit K

Constraint for ADIW, SBIW
They only work on R24, R26, R28 and R30 (d is 2-bit)

Translate If-Statement

if (cond)
  if-body;
Example:
  if (ch < 0) {
    ch = -ch;
  }
  LDS r24, ch ; load ch
  CPI r24, 0 ; compare
  BRGE endif ; take branch if complement true
  NEG r24 ; ch = -ch
  STS ch, r24 ; save ch
endif: ...

If-Statement: Structure

if (ch < 0) // C code is testing for less than
ch = -ch;
  LDS r24, ch
  CPI r24, 0
  BRGE endif ; Assemble test for complement
  NEG r24 ; complement
  STS ch, r24
endif: ...

RJMP: Unconditional Branch

Syntax: RJMP k
Condition: None
Operands: -2K ≤ k < 2K
Operation: PC ← PC + k + 1
Binary format:
**JMP: Unconditional Branch**

**JMP** – Jump, with a 22-bit absolute address

Syntax: **JMP k**
- Condition: None
- Operands: \(0 \leq k < 4M\)
- Operation: \(PC \leftarrow k\)

```
Binary:

<table>
<thead>
<tr>
<th>1001</th>
<th>010k</th>
<th>kkkk</th>
<th>110k</th>
</tr>
</thead>
<tbody>
<tr>
<td>kkkk</td>
<td>kkkk</td>
<td>kkkk</td>
<td>kkkk</td>
</tr>
</tbody>
</table>
```

**IF-Else Statement**

if (cond)
  if-body
else
  else-body;

Example:
extern int max, a, b;
if (a < b)
  max = b;
else
  max = a;

**If-Else Statement: Structure**

Control and Data Flow Graph

Linear Code Layout

```
\begin{align*}
&\text{cond} & \text{test cond} & \text{br if cond=F} & \text{if-body} & \text{jump} & \text{else-body} \\
&\text{F} & & & \text{If-body} & & \text{else-body} \\
&\text{T} & \text{if-body} & \text{jump} & \text{else-body} & \\
&\text{else-body} & & & & & \\
\end{align*}
```

; assume a in r25:r22, b in r23:r22, max in r20:r21 (all signed)

```
CP \quad r24, \quad r22 \\
CPC \quad r25, \quad r23 \\
BRGE \quad \text{else} \\
MOVW \quad r20, \quad r22 \\
RJMP \quad \text{endif} \\
\text{else:} \quad MOVW \quad r20, \quad r24 \\
\text{endif:} \quad \ldots
```

**Conditional Branch: Encoding Example**

```
syntax: \quad \text{BRLT k} \\
Condition: N \oplus V = 1 \\
N, V: \text{Two’}s \text{ complement’}s \text{ negative and overflow} \\
Operands: \quad -64\text{SkS}+63 \\
Operation: \quad \text{if true } PC \leftarrow PC+k+1 \\
\quad \text{otherwise } PC \leftarrow PC+1 \\
Binary:

<table>
<thead>
<tr>
<th>1111</th>
<th>00kk</th>
<th>kkkk</th>
<th>k100</th>
</tr>
</thead>
</table>
```

**Signed Type and Unsigned Type**

if \((a < b)\) ... else ...

a, b are \textbf{int} \quad a, b are \textbf{unsigned int}

CP \quad r24, \quad r22 \\
CPC \quad r25, \quad r23 \\
\textbf{BRGE} \quad \textbf{else} \\
\ldots \quad \ldots
Caveat: No BRLE and BRGT

Two types of if-conditions are trouble-free

C Assembly
if (a >= b) branch if a<b, use BRLT
if (a < b) branch if a≥b, use BRGE

What about
if (a > b) ...
if (a <= b) ...

Caveat: No BRLE and BRGT

\[
\text{C} \quad \text{Assembly}
\]

\[
\begin{align*}
\text{if (a > b)} & \rightarrow \text{branch if a <= b} \\
& \quad \text{CP r24, r22} \\
& \quad \text{CPC r25, r23} \\
& \quad \text{BRLT endif}
\end{align*}
\]

\[
\begin{align*}
\text{if (a <= b)} & \rightarrow \text{branch if a > b} \\
& \quad \text{CP r24, r22} \\
& \quad \text{CPC r25, r23} \\
& \quad \text{BRGE endif}
\end{align*}
\]

Problem: no BRLE and BRGT in AVR assembly!

Caveat: No Swap within CPI

What do we do? Swap the registers!

C Translated C Assembly

\[
\begin{align*}
\text{if (a > b) } \leftrightarrow \text{if (b < a), branch if b} & = a \\
& \quad \text{CP r22, r24} \\
& \quad \text{CPC r23, r25} \\
& \quad \text{BRLT endif}
\end{align*}
\]

\[
\begin{align*}
\text{if (a <= b) } \leftrightarrow \text{if (b > a), branch if b} & = a \\
& \quad \text{CP r22, r24} \\
& \quad \text{CPC r23, r25} \\
& \quad \text{BRGE endif}
\end{align*}
\]

Complex Condition

\[
\begin{align*}
\text{if (ch >= 0 \&\& ch <= 10)} & \\
& \text{LDS r24, ch ; load ch} \\
& \text{TST r24 ; test ch} \\
& \text{BRLT else} \\
& \text{CPI r24, 11 ; cmp ch, 11} \\
& \text{BRGE else ; if-body} \\
& \text{... ; else-body} \\
& \text{endif ; else-body}
\end{align*}
\]

Recall Lazy Evaluation
Complex Condition

```c
if (ch >= 0 || ch <= 10)
    LDS r24, ch ; load ch
    TST r24, ch ; test ch
    CPI r24, 11 ; cmp ch, 11
    BRGE if_body
    BRGE else
else:
    ... ; else-body
endif;
```

Another form of Lazy Evaluation

Function Call Convention

What are the issues with function call?
- Pass parameters
- Jump to the callee
- Use local storage in the stack
- Share registers between caller and callee
- Return to the caller
- Get the return value

We will study the AVR-GCC call convention
- It's NOT part of the instruction set architecture
- Must follow it in C/assembly programming or to use gcc library function

AVR-GCC Call Convention: Parameters and Return Value

Function parameters
- R25:R24, R23:R22, ..., R9:R8
- All aligned to start in even-numbered register
  i.e. char will take two registers (use the even one)
- A long type uses two pairs
- Extra parameters go to stack

Function return values
- 8-bit in r24 (with r25 cleared to zero), or
- 16-bit in R25:R24, or
- 32-bit in R25-R22, or
- 64-bit in R25-R18

AVR-GCC Call Convention: Register Usage

How to share registers between caller and callee?

Callee-save/Non-volatile: R2-R17, R28-R29
- Caller may use them for free, callee must keep their old values

Callee-save/Volatile: R18-R27, R30-R31
- Callee may use them for free, caller must save their old values if needed

Fixed registers
- R0: Temporary register used by gcc (no need to save)
- R1: Should be zero

AVR-GCC Call Convention

Key data structure: The Stack

- Saves the return address
- Holds local variables
- Pass extra part of parameters and return value (registers are used first in gcc AVR call convention)
**Hardware Support**

What the processor supports

- **RCALL, CALL**: Function call
- **RET**: Function return
- **PUSH, POP**: Stack operations

---

**Function Call: Example**

```plaintext
main:...;
   RCALL myfunc;
   ...;
myfunc:...;
   prologue;
   ...
   function body;
   ...
   epilogue;
   RET;
   return;
```

---

**Exercise**

```plaintext
int a, b, c;
void my_func() {
   ...
   c = max(a, b);
   ...
} 
int max(int a, int b) {
   if(a>b)
      return b;
   else
      return a;
}
```

---

**Function Call and Return**

**RCALL: Relative Call to Subroutine**

- **RCALL k**: k is 12-bit signed value
- **Operation**: \( PC \leftarrow PC + k + 1 \)  => Make the jump
- **Stack**: \( STACK \leftarrow PC + 1 \)  => Save the return PC
- **SP**: \( SP \leftarrow SP - 2 \)
- **Latency**: 3 cycles

**CALL: Long Call to Subroutine, 20-bit offset**

- **RCALL** can cover 4K-word range (ATmega128 has 64K-word or 128KB programming memory)

---

**Function Call: Example**

```plaintext
max:
   ; a=(r25:r24), b=(r23:r22), return value in (r25:r24)
   CP r24, r22; compare a, b
   OPC r25, r23;
   BRGE endif; branch if a>b
   MOVW r24, r22; move b to (r25:r24)
endif:
   RET;
```
Function Call and Return

RET: Return from subroutine

RET
Operation: PC ← STACK ; Restore return PC
SP ← SP+2 ; from the stack
Latency: 4 cycles

Stack Usage

SP register: Stack Pointer register

Stack Register

SP is two I/O registers
; initialize the SP to the highend of RAM
LDI r16, lo8(RAMEND)
OUT SPL, r16
LDI r16, hi8(RAMEND)
OUT SPH, r16
The I/O addresses are 0x3D and 0x3E on ATmega128 (to use IN/OUT)
The memory addresses are 0x5D and 0x5E (to use LDS/STS)

PUSH and POP

PUSH: Push register into stack
Syntax: PUSH Rr
Operations: STACK ← Rr
SP ← SP-1
PC ← PC+1
Latency: 2 cycles

AVR-GCC Call Convention: Register Usage

How to share registers between caller and callee?

Callee-save/Non-volatile: R2-R17, R28-R29
Caller may use them for free, callee must keep their old values

Caller-save/Volatile: R18-R27, R30-R31
Callee may use them for free, caller must save their old values if needed

Fixed registers
– R0: Temporary register used by gcc (no need to save)
– R1: Should be zero
AVR-GCC Call Convention

Example

```
int add2(int a, int b) {
    return a+b;
}

int add3(int a, int b, int c) {
    return add2(add2(a, b), c);
}

int main() {
    extern int sum, a, b, c;
    ...
    sum = add3(a, b, c);
    ...
}
```

Example

```
add2() is a leaf function, not need to use stack if you avoid using callee-save registers

add2:
; a=>r25:r24, b=>r23:r22
ADD r24, r22 ; add lower half
ADC r25, r23 ; add upper half
RET
```

```
add3:
; a=>r25:r24, b=>r23:r22, c=>r21:r20
PUSH r21 ; save c to stack
PUSH r20
RCALL add2 ; add2(a, b)
POP r22 ; restore c to r23:r22
POP r23
RCALL add2 ; add2(add2(a, b), c)
RET
```

Question: Why save c?

How main() calls add3: Assume for some reason, R29:R28 and R31:R30 must be preserved across the function all

main:
PUSH r31 ; save r31:r30
PUSH r30
LDS r64, a ; load a
LDS r25, a+1
LDS r22, b ; load b
LDS r23, b+1
LDS r20, c ; load c
LDS r21, c+1
RCALL add3 ; call add3(a, b, c)
STS sum, r24 ; save result to sum
STS sum+1, r25
POP r30 ; restore r31:r30
POP r31
```

Question: Is it not necessary to push/pop R29:R28?