CprE 288 – Introduction to Embedded Systems
ATmega128 Assembly Programming: Translating C Control Statements and Function Calls

Instructors:
Dr. Phillip Jones

Announcements

- HW11: Due Thursday 12/3 Midnight
- Project Demos: Mandatory Demo during your lab section next week (i.e. Deadweek).
- Projects
  - Lab attendance is mandatory
  - For each lab you miss you will loose 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 doesn’t matter
- **BRNE**: Not Equal, signed or unsigned doesn’t 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
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 $a \geq b$, $a$ and $b$ are variables of “unsigned char” type

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

Exercises

Exercise: Write a sequence of instructions

1. Branch to label if $a < b$
   
   LDS r24, a
   LDS r22, b
   CP r24, r22
   BRLT label

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

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 $m < n$
**CPI: Compare with Immediate**

**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)

---

**Translate If-Statement**

```
if (cond)  
  if-body;
Example:
if (ch < 0) {  
  ch = -ch;
  LDS r24, ch : load ch  
  TST r24 : test for zero or minus  
  BRGE endif ; skip if (ch<0) is false //check for complement
  NEG r24 ; ch = -ch
  STS ch, r24 ; save ch
endif: ...
```

---

**If-Statement: Structure**

**RJMP: Unconditional Branch**

**RJMP**: Relative jump, with a 12-bit relative address  
Syntax: RJMP k  
Condition: None  
Operands: -2K ≤ k < 2K  
Operation: PC←PC+k+1  
Binary format:  
```
1100    kkkk    kkkk    kkkk
```
**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:

```
1001 010k kkkk 110k
kkkk kkkk kkkk kkkk
```

---

**If-Else Statement**

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

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

---

**If-Else Statement: Structure**

Control and Data Flow Graph

```
cond
F
T
if-body
F-body
```

Linear Code Layout

```
test cond
br if cond=F
if-body
jump
else-body
```

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

```
CP r24, r22
CPC r25, r23
BRGE else
MOVW r20, r24
RJMP endif
endif: ...
else:
MOVW r20, r22
endif: ...
```

---

**Signed Type and Unsigned Type**

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

a, b are int
a, b are unsigned int

CP r24, r22
CPC r25, r23
BRGE else
... ...

---

**Conditional Branch: Encoding Example**

syntax: `BRLT k`
Condition: \(N \oplus V = 1\)

\(N, V: \text{Two’ s complement’ s negative and overflow}\)
Operands: \(-64\text{sk}s+63\)
Operation: if true \(PC \leftarrow PC+k+1\)
otherwise \(PC \leftarrow PC+1\)

Binary:

```
1111 00kk kkkk k100
```
Caveat: No BRLE and BRGT

Two types of if-conditions are trouble-free

\[
\begin{align*}
\text{C Assembly} \\
\text{if (a >= b)} & \quad \text{branch if a<b, use BRLT} \\
\text{if (a < b)} & \quad \text{branch if a\geq b, use BRGE}
\end{align*}
\]

What about

\[
\begin{align*}
\text{if (a > b)} & \quad \text{...} \\
\text{if (a <= b)} & \quad \text{...}
\end{align*}
\]

Caveat: No BRLE and BRGT

What do we do? \textit{Swap} the registers!

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

Caveat: No Swap within CPI

The swap trick doesn’t work with CPI

Case 1: if (ch > 10)

\[
\begin{align*}
\text{CPI 10, r24} \\
\text{BRLT endif}
\end{align*}
\]

Case 2: if (ch <= 10)

\[
\begin{align*}
\text{CPI 10, r24} \\
\text{BRGE endif}
\end{align*}
\]

Caveat: No Swap within CPI

We can increment the immediate value

\[
\begin{align*}
\text{C translated C Assembly} \\
\text{Case 1: if (ch > 10) \Leftrightarrow if (ch >= 11), branch if ch<11} \\
& \quad \text{CPI r24, 11} \\
& \quad \text{BRLT endif} \\
\text{Case 2: if (ch <= 10) \Leftrightarrow if (ch < 11), branch if ch\geq 11} \\
& \quad \text{CPI r24, 11} \\
& \quad \text{BRGE endif}
\end{align*}
\]

Complex Condition

if (ch >= 0 && ch <= 10)

\[
\begin{align*}
\text{LDS r24, ch; load ch} \\
\text{TST r24; test ch} \\
\text{BRLT else} \\
\text{CPI r24, 11; cmp ch, 11} \\
\text{BRGE else; if-body} \\
\text{... else-body} \\
\text{endif;}
\end{align*}
\]

Recall \textit{Lazy Evaluation}
Complex Condition

if (ch >= 0 || ch <= 10)

  LDS r24, ch ; load ch
  TST r24 ; test ch
  BRGE if_body
  CPI r24, 11 ; cmp ch, 11
  BRGE else
if_body:
  ...
else:
  ...
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

Function and Stack

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

**main**: …

```
RCALL myfunc ; other code
```

```
... ; prologue
... ; function body
... ; epilogue
RET ; return
```

**myfunc**: …

```
... ; prologue
... ; function body
... ; epilogue
RET ; return
```

**Exercise**

```c
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: Example

```
my_func:
    ... ; more instructions
    c = max(a, b)
LDS r24, a ; 1st parameter of max
LDS r25, a+1
LDS r22, b ; 2nd parameter of max
LDS r23, b+1;
RCALL max
STS c, r24 ; save return results
STS c+1 r25;
    ... ; more instructions
```

---

### Function Call and Return

#### RCALL: Relative Call to Subroutine

RCALL k; k is 12-bit signed value

Operation: PC ← PC+k+1 => Make the jump

```
STACK ← PC+1 => Save the return PC
SP ← 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 and Return

RET: Return from subroutine
RET
Operation: \( PC \leftarrow \text{STACK} \) ; Restore return PC
SP \( \leftarrow \text{SP+2} \) ; from the stack
Latency: 4 cycles

Stack Usage

SP register: Stack Pointer register

PUSH and POP

PUSH: Push register into stack
Syntax: PUSH Rr
Operations: \( \text{STACK} \leftarrow Rr \)
SP \( \leftarrow \text{SP-1} \)
PC \( \leftarrow \text{PC+1} \)
Latency: 2 cycles

PUSH and POP

POP: Push register into stack
Syntax: POP Rr
Operations: \( Rr \leftarrow \text{STACK} \)
SP \( \leftarrow \text{SP+1} \)
PC \( \leftarrow \text{PC+1} \)
Latency: 2 cycles

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)

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Call-saved/Callee-save/Non-volatile</th>
<th>Call-used/Caller-save/Volatile</th>
<th>Fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>R10 (P1)</td>
<td>R16 (P6)</td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td>R11 (P1)</td>
<td>R17 (P6)</td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td>R12 (P2)</td>
<td>R18 (P7)</td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td>R13 (P2)</td>
<td>R19 (P7)</td>
<td></td>
</tr>
<tr>
<td>R4</td>
<td>R14 (P3)</td>
<td>R20 (P7)</td>
<td>R15 (P5)</td>
</tr>
<tr>
<td>R5</td>
<td>R15 (P3)</td>
<td>R21 (P7)</td>
<td>R14 (P5)</td>
</tr>
<tr>
<td>R6</td>
<td>R16 (P4)</td>
<td>R22 (P7)</td>
<td>R13 (P5)</td>
</tr>
<tr>
<td>R7</td>
<td>R17 (P4)</td>
<td>R23 (P7)</td>
<td>R12 (P5)</td>
</tr>
<tr>
<td>R8</td>
<td>R18 (P5)</td>
<td>R24 (P7)</td>
<td>R11 (P5)</td>
</tr>
<tr>
<td>R9</td>
<td>R19 (P5)</td>
<td>R25 (P7)</td>
<td>R10 (P5)</td>
</tr>
<tr>
<td>R10</td>
<td>R20 (P6)</td>
<td>R26 (P7)</td>
<td>R9 (P5)</td>
</tr>
<tr>
<td>R11</td>
<td>R21 (P6)</td>
<td>R27 (P7)</td>
<td>R8 (P5)</td>
</tr>
<tr>
<td>R12</td>
<td>R22 (P6)</td>
<td>R28 (P7)</td>
<td>R7 (P5)</td>
</tr>
<tr>
<td>R13</td>
<td>R23 (P6)</td>
<td>R29 (P7)</td>
<td>R6 (P5)</td>
</tr>
<tr>
<td>R14</td>
<td>R24 (P6)</td>
<td>R30 (P7)</td>
<td>R5 (P5)</td>
</tr>
<tr>
<td>R15</td>
<td>R25 (P6)</td>
<td>R31 (P7)</td>
<td>R4 (P5)</td>
</tr>
</tbody>
</table>

**Call-saved/Callee-save/Non-volatile**
- P: Parameter
- V: Result

### Example

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

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

**add3:**
```assembly
; 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:**
```assembly
PUSH r31 ; save r31:r30
PUSH r30
LDS r31, 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?