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 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.
Recall control statements in C

If statement

if (cond) if-body;

if (cond) if-body else else-body;
C Control Statements

Loop statements:

```c
while (cond) loop-body;

do loop-body
while (cond);

for (init-expr; cond-expr; incr-expr) loop-body;
```
How to Evaluate a Condition

Evaluate a simple condition:

1. Have flags set in SREG
2. Branch is \textbf{taken} if certain flag or their combination is true

There are two possible outcomes for a branch: \textbf{Taken} or \textbf{Not Taken}

Example:

\begin{verbatim}
LDS r24, a
LDS r26, b
CP r24, r26 ; compare a, b and set flags
BRLT endif ; branch if a < b
endif:
\end{verbatim}
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, \( S = N \oplus V \)
- **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>0</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:** \( S = N \oplus V \) (same as N because \( V=0 \))
- **I, T, H, C:** Not affected
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
CP and CPC: Compare Multiple Registers

**CP**: Compare

Syntax: CP Rd, Rr

Operation: \( \text{Rd-Rr}, \text{PC} \leftarrow \text{PC+1} \)

**CPC**: Compare with Carry

Syntax: CPC Rd, Rr

Operation: \( \text{Rd-Rr-C}, \text{PC} \leftarrow \text{PC+1} \)

CP/CPC is like SUB/SBC but only affect the flags
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
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 |
extern unsigned long m, n;
Branch to label if m < n
CPI: Compare with Immediate

Syntax: CPI Rd, K
Operands: $16 \leq d \leq 31$, $0 \leq K \leq 255$
Operations: Rd-K, PC$\leftarrow$PC+1

Branch to label if r24 $\geq$ 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: \( 16 \leq d \leq 31 \), \( 0 \leq K \leq 255 \)

In other words, they only work on \( R_{16-31} \)

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)
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

Control and Data Flow Graph

Linear Code Layout

cond

F

T

if-body

test cond

br if cond=F

if-body
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
STS   ch,   r24
endif: ...

if-body
br if cond=F
  test cond
RJMP: Unconditional Branch

RJMP: Relative jump, with a 12-bit relative address

Syntax: RJMP k
Condition: None
Operands: \(-2K \leq k < 2K\)
Operation: \(PC \leftarrow PC + k + 1\)
Binary format: 

\[
\begin{array}{cccc}
1100 & kkkk & kkkk & kkkk \\
\end{array}
\]
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

```
cond
  └── F
    ├── T
    │   └── if-body
    │       └── else-body
```

Linear Code Layout

```
test cond
  └── br if cond=F
      └── If-body
          └── jump
              └── else-body
```
If-Else Statement: Structure

; assume a in r25:r24, b in r23:r22, max in r20:r21 (all signed)

**CP** r24, r22
**CPC** r25, r23
**BRGE** else
**MOVW** r20, r22
**RJMP** endif

else: **MOVW** r20, r24
endif: …

test cond
  ↓
  br if cond=F
  ↓
  If-body
  ↓
  jump
  ↓
  else-body
  ↓
 endif
Signed Type and Unsigned Type

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

a, b are **int**

| CP     | r24,   | r22   |
| CPC    | r25,   | r23   |
| **BRGE** | else  |

... 

a, b are **unsigned int**

| CP     | r24,   | r22   |
| CPC    | r25,   | r23   |
| **BRSH** | else  |

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

\[
\begin{array}{cccc}
1111 & 00kk & kkkk & k100 \\
\end{array}
\]
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

C      Assembly
If (a > b), branch if a <= b?
    CP       r24,   r22
    CPC      r25,   r23
    BRLE     endif

if (a <= b), branch if a > b?
    CP       r24,   r22
    CPC      r25,   r23
    BRGT     endif

Problem: no BRLE and BRGT in AVR assembly!
Caveat: No BRLE and BRGT

What do we do? Swap the registers!

C         Translated C         Assembly

If (a > b) ⇔ if (b < a),    branch if b >= a
              CP r22, r24
              CPC r23, r25
              BRGE endif

if (a <= b) ⇔ if (b >= a), branch if b < a
              CP r22, r24
              CPC r23, r25
              BRLT endif
The swap trick doesn’t work with CPI
Case 1: if (ch > 10)
   CPI 10, r24
   BRLT endif
Case 2: if (ch <= 10)
   CPI 10, r24
   BRGE endif
We can increment the immediate value

C              translated C              Assembly

Case 1: if (ch > 10) ⇔ if (ch >= 11), branch if ch<11

\[ \text{CPI } r24, 11 \]
\[ \text{BRLT } \text{endif} \]

Case 2: if (ch <= 10) ⇔ if (ch < 11), branch if ch≥11

\[ \text{CPI } r24, 11 \]
\[ \text{BRGE } \text{endif} \]
Complex Condition

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDS r24, ch</td>
<td>load ch</td>
</tr>
<tr>
<td>TST r24</td>
<td>test ch</td>
</tr>
<tr>
<td><strong>BRLT</strong></td>
<td>else</td>
</tr>
<tr>
<td>CPI r24, 11</td>
<td>cmp ch, 11</td>
</tr>
<tr>
<td>BRGE else</td>
<td>if-body</td>
</tr>
<tr>
<td>...</td>
<td>; if-body</td>
</tr>
<tr>
<td>else:</td>
<td>; else-body</td>
</tr>
<tr>
<td>...</td>
<td>; else-body</td>
</tr>
</tbody>
</table>

Recall **Lazy Evaluation**
Complex Condition

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDS r24, ch</td>
<td>load ch</td>
</tr>
<tr>
<td>TST r24</td>
<td>test ch</td>
</tr>
<tr>
<td>BRGE if_body</td>
<td>if_body</td>
</tr>
<tr>
<td>CPI r24, 11</td>
<td>cmp ch, 11</td>
</tr>
<tr>
<td>BRGE else</td>
<td>else</td>
</tr>
</tbody>
</table>

if_body:

... ; if-body

else:

... ; else-body

endif:

Another form of Lazy Evaluation
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

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>R0</th>
<th>R8 (P8)</th>
<th>R16 (P4)</th>
<th>R24 (P0, V1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1  (Zero)</td>
<td>R9 (P8)</td>
<td>R17 (P4)</td>
<td>R25 (P0, V0)</td>
</tr>
<tr>
<td>R2</td>
<td>R10 (P7)</td>
<td>R18 (P3, V7)</td>
<td>R26</td>
</tr>
<tr>
<td>R3</td>
<td>R11 (P7)</td>
<td>R19 (P3, V6)</td>
<td>R27</td>
</tr>
<tr>
<td>R4</td>
<td>R12 (P6)</td>
<td>R20 (P2, V5)</td>
<td>R28</td>
</tr>
<tr>
<td>R5</td>
<td>R13 (P6)</td>
<td>R21 (P2, V4)</td>
<td>R29</td>
</tr>
<tr>
<td>R6</td>
<td>R14 (P5)</td>
<td>R22 (P1, V3)</td>
<td>R30</td>
</tr>
<tr>
<td>R7</td>
<td>R15 (P5)</td>
<td>R23 (P1, V2)</td>
<td>R31</td>
</tr>
</tbody>
</table>

- **Non-volatile**: Callee save and restore if used
- **Volatile**: Caller save and restore if used
- **Fixed**

P: Parameter  
V: Result
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
main:…

RCALL myfunc

…

myfunc: …

…

…

RET

; other code

; call myfunc

; return to here

; prologue

; function body

; epilogure

; return
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: Example

max:
; a=>(r25:r24), b=>(r23:r22), return value in (r25:r24)
CP r24, r22 ; compare a, b
CPC r25, r23;
BRGE endif ; branch if a>=b
MOVW r24, r22 ; move b to (r25:r24)
endif:

RET;
Function Call and Return

**RCALL: Relative Call to Subroutine**

RCALL \( k \); \( k \) is 12-bit signed value

Operation:
- \( PC \leftarrow PC + k + 1 \) \( \Rightarrow \) Make the jump
- \( STACK \leftarrow PC + 1 \) \( \Rightarrow \) Save the return PC
- \( SP \leftarrow SP - 2 \)

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)
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

Before RCALL

High end
SP
Low end
Other
Data
Data Memory

After RCALL

ret addr.
Other
Data
Data Memory

main
RCALL ...
...
next addr.
myfunc
ret
Program Memory

50
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**: Push register into stack

Syntax: PUSH Rr

Operations:  
- STACK ← Rr  
- SP ← SP - 1  
- PC ← PC + 1  

Latency: 2 cycles
**POP:** Push register into stack

Syntax: POP Rr

Operations:  
- \( Rr \leftarrow \text{STACK} \)
- \( \text{SP} \leftarrow \text{SP} + 1 \)
- \( \text{PC} \leftarrow \text{PC} + 1 \)

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

Call-saved/Callee-save/Non-volatile

Call-used/Caller-save/Volatile

Fixed

P: Parameter
V: Result
Example

```c
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 r24, 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?