CprE 288
Translating C Control Statements and Function Calls, AVR Interrupt Processing

Instructors:
Dr. Phillip Jones
Dr. Zhao Zhang
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.
DO-WHILE Loop

do
   \textit{do-body};
while (\textit{cond});

Example:

\begin{verbatim}
void strcpy (char *dst, char *src)
{
    char ch;
    do {
        ch = *src++;
        *dst++ = ch;
    } while (ch);
}
\end{verbatim}
DO-WHILE Loop

Control and Data Flow Graph

Linear Code Layout

Loop prologue (optional)

do-body

test cond

br if cond=T

Loop epilogue (optional)
DO-WHILE Loop

; parameter: dst=>R25:R24, src=>R23:R22
; reg use: dst=>Z-reg, src=>X-reg
strcpy:
    movw r30, r24
    movw r26, r22
loop:
    ld  r20, X+  // byte from src
    st  Z+, r20  //store to dst
    tst  r20
    brne loop
    ret
WHILE Loop

Control and Data Flow Graph

Linear Code Layout

(sequential code layout where:
- `while-body` is the body of the loop,
- `test cond` checks the condition,
- `br if cond=T` branches to the body if the condition is true,
- `jump` represents the jump to the end of the loop)

(optional prologue and epilogue not shown)
strlen(): return the length of a C string

int strlen(char *str)
{
    int len = 0;
    while (*str++)
        len++;
    return len;
}
WHILE Loop

; parameter: str=>r25:r24
; reg use: str=>Z-reg, len=>r25:r24, ch=>r22

strcpy:
    movw r30, r24
    clr r24                   ; len = 0
    clr r25
    rjmp test

loop:
    adiw r24, 1

test:
    ld r22, Z+
    tst r22
    brne loop
    ret
FOR Loop

for (init-exp; cond-exp; incr-exp)
  for-body;

Example:
unsigned char checksum(unsigned char data[],
    int N)
{
    unsigned char checksum = 0;
    for (int i=N; i>=0; i--)
        checksum ^= data[i];
    return checksum;
}
FOR Loop

Control and Data Flow Graph

```
init-expr

for-body

incr-expr

cond

F

T
```

Linear Code Layout

```
init-expr

jump

for-body

Incr-expr

test cond

br if cond=T

(optional prologue and epilogue not shown)
FOR Loop

; parameter: data=>r25:r24, N=>r23:r22
; reg use: data=>Z-reg, checksum=>r24, i=>r27:r26, ch=>r20

checksum:

movw r30, r24 ; Z-reg = data
clr r24 ; checksum = 0
clr r25 ; r25 = 0
clr r26 ; i = 0
clr r27
rjmp cond
FOR Loop

; parameter: data=>r25:r24, N=>r23:r22
; reg use: data=>Z-reg, checksum=>r24, i=>r27:r26,
; ch=>r20

loop:
    ld  r20, Z+ ; load data[i]
    eor r24, r20 ; checksum ^= ...
    adiw r26, 1 ; i++

cond:
    cp  r26, r22 ; cmp i, n
    cpc r27, r23
    brlt loop ; br if i<n
    ret
Loop Optimization: Example

; parameter: data=>r25:r24, N=>r23:r22
; reg use: data=>Z-reg, checksum=>r24, i=>r27:r26,
checksum:
    movw r30, r24          ; Z-reg = data
    movw r26, r22          ; r27:r26 = N
    clr r24                ; checksum = 0
    clr r25                ; r25 = 0
    cp r26, r1             ; cmp r27:r26 to 0
    cpc r27, r1            ; to set the Z-flag
    rjmp cond_check

loop:
    ld r20, Z+             ; load data[i]
    eor r24, r20           ; checksum ^= data[i]
    sbiw r26, 1            ; N--

cond_check:
    brne loop              ; repeat if N != 0

*Four instructions in the loop body instead of six, 7 cycles vs 9 cycles*
extern int data[];

// clear the first n elements of data[]
void clear_data(int n)
{
    for (int i = 0; i < n; i++)
        data[i] = 0;
}
FOR Loop Example

; parameter: n=>r25:r24
; reg use: i=>r26:r27,
;       &data[i]=>r31:r30 (Z-reg)

clear_data:
   ldi   r30, lo8(data)       ; Z-reg = data
   ldi   r31, hi8(data)
   clr   r26                 ; i = 0
   clr   r27
   rjmp  cond_check          ; jump to condition
FOR Loop Example

\begin{verbatim}
for_loop:
    st  Z+, r1                        ; data[i] = 0
    st  Z+, r1
    adiw r26, 1                       ; i++

cond_check:
    cp   r26, r24                    ; cmp i, n
    cpc  r27, r25
    brlt for_loop                    ; br if i<n
    ret
\end{verbatim}
FOR Loop Example: Optimized Version

; n=>r25:r24, &data[i]=>r31:r30 (Z-reg)
clear_data:
  ldi r30, lo8(data) ; Z-reg = data
  ldi r31, hi8(data)
cp r24, r1 ; test condition for 1st time
cpc r25, r1
rjmp cond_check ; jump to condition
for_loop:
  st Z+, r1 ; data[i] = 0
  st Z+, r1
  sbiw r24, 1 ; n--
cond_check:
  brne for_loop ; br if n!=0
ret
AVR Interrupt Processing

1. Exceptional Control Flow
2. Connecting interrupt source and ISR: Vector Table
3. Writing ISR functions

ISR: Interrupt Service Routine

*Interrupt processing will NOT be covered in Exam 3*
Exceptional Control Flow

Exception events in *general* processors:
- Internal sources: Arithmetic overflow, memory violation, and others
- External sources: Timer expirations, input capture, output compare, and others

AVR: All are called interrupts, exception handler is called ISR
Exceptional Control Flow

Need to do the following

- Stop the foreground execution
- Establish a running environment for ISR
- Find and run the corresponding ISR
- Resume the foreground execution
Interrupt Principle

What computation is correct?
- If the program state at the end is what we want to see
- That includes registers and memory contents that programmers may perceive

What is computation?
- It’s a transition sequence of a finite state machine leading to the desired state, and
- The next state is a function of the current state (a sub-type of Moore Machine)

How do we stop (and then resume) a finite state machine?
- Restore state including PC, GPRs, SREG, Stack, and any other important state information
Interrupt Principle

State of a program execution
- Registers: PC, R0-R31, SREG, SP, others
- Static data (global and state variables)
- Stack data (local variables, linkage, temp. variables)

The next state is a function of the current state during a computation phase
Interrupt Principle

Registers:
Save and restore all registers to be changed

Data Segment:
Only change ISR-private variable and shared variables
Do not change other part of data memory

Stack Segment:
Create its one own stack frames
Do not change what’s already here in stack
Restore stack top before exiting
**AVR Interrupt Vector Table**

**Interrupt number:** unique identifier number of each event

**Interrupt Jump Table:** Each entry is a jump instruction that jumps to an Interrupt Service Routine (ISR)

In AVR, the table starts at 0x0000 by default
AVR Interrupt Vector Table

How to make the exceptional control flow happen?

Next PC

Select Signal

Current PC + 1

Current PC + 2

Branch Target

0x0000 + (Interrupt number – 1)

Note: Instruction memory uses word address
one word is two bytes

Interrupt has happened and interrupt is enabled

Instruction has happened and interrupt is enabled
AVR Interrupt Vector Table

Vector Table is actually different: Each entry stores the address to an ISR
  – For example, Motorola MPC555 uses exception vector table

However, AVR uses Jump Table but calls it Vector Table
# AVR Interrupt Vector Table

<table>
<thead>
<tr>
<th>Vector No.</th>
<th>Program Address$^{(1)}$</th>
<th>Source</th>
<th>Interrupt Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$0000$</td>
<td>RESET</td>
<td>External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset</td>
</tr>
<tr>
<td>2</td>
<td>$0002$</td>
<td>INT0</td>
<td>External Interrupt Request 0</td>
</tr>
<tr>
<td>3</td>
<td>$0004$</td>
<td>INT1</td>
<td>External Interrupt Request 1</td>
</tr>
<tr>
<td>4</td>
<td>$0006$</td>
<td>INT2</td>
<td>External Interrupt Request 2</td>
</tr>
<tr>
<td>5</td>
<td>$0008$</td>
<td>INT3</td>
<td>External Interrupt Request 3</td>
</tr>
<tr>
<td>6</td>
<td>$000A$</td>
<td>INT4</td>
<td>External Interrupt Request 4</td>
</tr>
<tr>
<td>7</td>
<td>$000C$</td>
<td>INT5</td>
<td>External Interrupt Request 5</td>
</tr>
<tr>
<td>8</td>
<td>$000E$</td>
<td>INT6</td>
<td>External Interrupt Request 6</td>
</tr>
<tr>
<td>9</td>
<td>$0010$</td>
<td>INT7</td>
<td>External Interrupt Request 7</td>
</tr>
<tr>
<td>10</td>
<td>$0012$</td>
<td>TIMER2 COMP</td>
<td>Timer/Counter2 Compare Match</td>
</tr>
<tr>
<td>11</td>
<td>$0014$</td>
<td>TIMER2 OVF</td>
<td>Timer/Counter2 Overflow</td>
</tr>
<tr>
<td>12</td>
<td>$0016$</td>
<td>TIMER1 CAPT</td>
<td>Timer/Counter1 Capture Event</td>
</tr>
<tr>
<td>13</td>
<td>$0018$</td>
<td>TIMER1 COMPA</td>
<td>Timer/Counter1 Compare Match A</td>
</tr>
<tr>
<td>14</td>
<td>$001A$</td>
<td>TIMER1 COMPB</td>
<td>Timer/Counter1 Compare Match B</td>
</tr>
</tbody>
</table>

35 interrupt sources in total; see page 60 of ATmega128 data sheet
The AVR Interrupt Vector Table has 35 entries, starting from 1
- By default, GCC fills all entries with a default ISR
- The default ISR resets the program execution

If you declare an ISR for an interrupt source:
- GCC fills the associated entry with “JMP your_ISR”

Example: ISR (TIMER1_CAPT_vect)
- The C function name is TIMER1_CAPT_vect()
- Entry 12 (address 0x0016) of the table is filled with the starting address of TIMER1_CAPT_vect()
To write an assembly ISR, create a .S file in the project, and write function as follows

```c
#include <avr/io.h>

; Input capture ISR on Timer/Counter 1, channel A
.gglobal TIMER1_COMPA_vect
TIMER1_COMPA_vect:
    ... ; my assembly code, many lines here
    RETI
```

Use the right vector name to declare your assembly function. GCC won’t report if the name is wrong.
General procedure of writing an ISR in assembly

1. Push ALL registers that could be changed into the stack
2. Interrupt processing
3. Pop all saved registers from the stack
4. Return from interrupt
Interrupt Service Routine

When Interrupt happens, the CPU let the current instruction finish, and then

1. Clear the I flag in SREG (to disable interrupt)
2. Push PC into the stack

Instruction **RETI** (Return From Interrupt) does the reverse:

1. Pop PC from the stack
2. Set the I flag in SREG (to enable interrupt)
ISR Example: Count IC Events

.global TIMER1_CAPT_vect
TIMER1_CAPT_vect:
PUSH r1 ; save r1
IN r1, 0x3F ; load SREG
PUSH r1 ; push SREG

LDS r1, n; load n
INC r1 ; n++
STS n, r1; store n

POP r1 ; pop SREG
OUT 0x3F, r1 ; restore SREG
POP r1 ; restore r1
RETI ; return from interrupt
ISR: What Registers to Save?

An ISR can be written in C and it calls other C functions (which may only change GPRs and SREG)

What GPRs should be saved before an ISR written in C starts execution, for correctness and maximum efficiency? (No more, no less.)

A. Caller-save (volatile) and fixed registers: R0-R1, R18-R27 and R30-R31, or
B. Callee-save (non-volatile) registers: R2-R17 and R28-R29, or
C. All GPRs: R0-R31?

The answer is A.