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

\[
\text{do}
\quad \text{do-body;}
\quad \text{while (cond);}
\]

Example:

```c
void strcpy (char *dst, char *src)
{
    char ch;
    do {
        ch = *src++;
        *dst++ = ch;
    } while (ch);
}
```

DO-WHILE Loop

; parameter: dst=>R25:R24, src=>R23:R22
; reg use: dst=>X-reg, src=>X-reg

strcpy:

\[
\text{movw r30, r24}
\text{movw r26, r22}
\]

loop:

\[
\text{ld r20, X+ // byte from src}
\text{st Z4, r20 //store to dst}
\text{tst r20}
\text{brne loop}
\text{ret}
\]

WHILE Loop

; parameter: dst=>R25:R24, src=>R23:R22
; reg use: dst=>X-reg, src=>X-reg

strcpy:

\[
\text{movw r30, r24}
\text{movw r26, r22}
\]

loop:

\[
\text{ld r20, X+ // byte from src}
\text{st Z4, r20 //store to dst}
\text{tst r20}
\text{brne loop}
\text{ret}
\]

```c
while (cond)
{
    do-body
    test cond
    br if cond-T
    jump
}
```
WHILE Loop Example

```c
strlen(): return the length of a C string
int strlen(char *str)
{
    int len = 0;
    while (*str++)
        len++;
    return len;
}
```

WHILE Loop

```c
; 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

```c
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

```c
; 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

```c
; 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 r26 ; r25 = 0
cpc r27, 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--
  clr r27
  rjmp cond_check ; jump to condition

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
cond_check:
  brne loop ; repeat if N != 0

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  ; i++
  adiw r26, 1
  cpc r27, r25
  brlt for_loop ; br if i<n
  ret

FOR Loop

Another example
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

  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

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

---

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 subtype 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

---

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 |

---

Exception 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
AVR Interrupt Vector Table

How to make the exceptional control flow happen?

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

Current PC

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

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()

AVR Interrupt Vector Table

To write an assembly ISR, create a .S file in the project, and write function as follows

#include <avr/io.h>

; Input capture ISR on Timer/Counter 1, channel A
.global 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

Interrupt Service Routine

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.