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

Instructors:
Dr. Phillip Jones
Dr. Zhao Zhang
Announcements

• Exam 3: Friday 7/6 (last day of class)

• Final Projects
  – Projects: Mandatory Demos Friday, 7/6 (11am – 3pm)
  – Reminder, Lab attendance is mandatory: -10 points from final project for each lab session you miss

• Quiz 7 (Thursday, 7/3): ARM Procedure Call Standard readings.

• Reading for the next few weeks
  – **Textbook**: Chapter 2.1-2.3, and 2.6.1-2.6.2
  – **Textbook**: Chapter 4.1 – 4.3
  – **Assemble ARM instruction set manual**:  
    • Preface, Chapter 3, Chapter 4
  – **ARM Procedure Call Standard**:  
    • Sections: 5, 7.1.1, 7.2.
Major Classes of Assembly Instructions

- **Data Movement**
  - Move data between registers
  - Move data in & out of Memory
  - Different addressing modes

- **Logic & Arithmetic**
  - Addition, subtraction, etc.
  - AND, ORR, EOR (Exclusive OR), bit shift, etc.

- **Control Flow**
  - Control which sections of code should be executed (e.g. In C “IF”, “CASE”, “WHILE”, etc.
  - Function Calls
Loop statements:

**While statement:**

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

**Do-While statement:**

```c
do {
   loop-body
} while (cond);
```

**For statement:**

```c
for (init-expr; cond-expr; incr-expr) {
   loop-body;
}
```
DO-WHILE Loop

Example:

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

Do-While statement:
```
do
{
    loop-body
}
while (cond);
```
**DO-WHILE Loop**

**Control and Data Flow Graph**

- do-body
- cond
  - T
  - F

**Linear Code Layout**

1. Loop prologue (optional)
2. do-body
3. test cond
4. br if cond=T
5. Loop epilogue (optional)
DO-WHILE Loop

; parameter: dst=>R0, src=>R1
; reg use: ch=>R2

strcpy:
   ;e.g. initialize local vars

loop:
   LDRB R2, [R1], #1 ; get bbyte from src
   STRB R2, [R0], #1 ; store to dst
   CMP R2, #0
   BNE loop ; ch!=0
   BX LR ; return to caller
WHILE Loop

Control and Data Flow Graph

Linear Code Layout

Jump

While-body

Test cond

Br if cond=T

(optional prologue and epilogue not shown)
**WHILE Loop Example**

`strlen()`: return the length of a C string

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

*While statement:*

```c
while (cond)
{
    loop-body;
}
```
WHILE Loop

; parameter: str=> R0, return string length=> R0
; reg use: len=>R3, tmp_char=>r2

strcpy:
    ANDS  R3, #0 ; len = 0 (prologue)

B test

loop:
    ADDS R3, 1

test:
    LDRB  R2, [R0], #1 ; get byte from str
    CMP  R2, #0
    BNE  loop ; tmp_char!=0
    MOV  R0, R3 ; set return value
    BX  LR ; return to caller
FOR Loop

For statement:
```
for (init-expr; cond-expr; incr-expr)
{
    loop-body;
}
```

Example:
```
unsigned char checksum(unsigned char data[],
                        int N)
{
    unsigned char checksum = 0;
    for (int i=0; i < N; 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=>R0, N=>R1, return value=>R0
; reg use: checksum=>R2, i=>R3, temp_data=>R4

checksum:

    PUSH R4 ; preserve is non-volatile
    ANDS R2, #0 ; checksum = 0
    ANDS R3, #0 ; i = 0
    B cond

loop:

    LDRB R4, [R0], #1 ; load data[i]
    EOR R2, R4 ; checksum ^= data[i]
    ADDS R3, #1 ; i++

cond:

    CMP R3, R1 ; cmp i, n
    BLT loop ; br if i < n

    POP R4 ; preserve is non-volatile
    MOV R0, R2 ; set return value
    BX LR ; return to caller
Loop Optimization: Example

; parameter: data=>R0, N=>R1, return value=>R0
; reg use: checksum=>R2, temp_data=>R3

checksum:
  ANDS R2, #0 ;checksum = 0
  ANDS R1, R1 ;set Status flag

B cond
loop:
  LDRB R3, [R0], #1 ;load data[i]
  EOR R2, R3 ;checksum^=data[i]
  SUBS R1, #1 ;N--

cond:
  BNE loop ;br if N ! = 0

MOV R0, R2 ;set return value
BX LR ;return to caller

One less instruction in loop: 4 vs. 5, for long running loops can save much time!
Another example:

```c
int data[]; // at location 0x1000_A000

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

; parameter: N=>R0
; reg use: data=>R1, R2=>0, i=>R3,
clear_data:

    MOVW R1, #0x1000 ; data global
    MOVT R1, #0xA000 ;
    ANDS R2, #0 ; R2 = 0
    ANDS R3, #0 ; i = 0
    B cond

loop:

    STR R2, [R1], #4 ; clear data[i]
    ADDS R3, #1 ; i++

cond:

    CMP R3, R0 ; cmp i, N
    BLT loop ; br if i<N

BX LR ; return to caller

i = 0

jump

data[i] = 0

i++

cmp i, N

br if i < N
Optimized version

; parameter: N=>R0
; reg use: data=>R1, R2=>0,
checksum:

    MOVW R1, #0x1000 ; data global
    MOVT R1, #0xA000 ;
    ANDS R2, #0 ;R2 = 0
    ANDS R0, R0 ;set Status flag
    B cond

loop:
    STR R2, [R1], #4 ;clear data[i]
    SUBS R0, #1 ; N--

cond:

    BNE loop ;br if N !=0

    BX LR ; return to caller

One less instruction in loop: 3 vs. 4, for long running loops can save much time!
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*
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 Diagram]

- Event occurs here
- Exception processing
- Exception return (optional)
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
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 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

<table>
<thead>
<tr>
<th>Jump table word</th>
<th>Code for ISR 1</th>
<th>Code for ISR 2</th>
<th>Code for ISR 3</th>
<th>Code for ISR n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>JMP ISR1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>JMP ISR2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>JMP ISR3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(n-1)</td>
<td>JMP ISRn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
AVR Interrupt Vector Table

How to make the exceptional control flow happen?

Select Signal

Current PC + 1
Current PC + 2
Branch Target

Next PC

0x0000 + (Interrupt number – 1)

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

Interrupt has happened and interrupt is enabled

Instruction memory uses word address: one word is two bytes
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$^{(2)}$</th>
<th>Source</th>
<th>Interrupt Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$0000^{(1)}$</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()
AVR Interrupt Vector Table

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
.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.