

CprE 288

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

Instructors:

Dr. Phillip Jones

Dr. Zhao Zhang

Announcements

- Final Projects
 - Peer Review: Each person submit a Peer review of team members.
 - **Mandatory** Demo during you lab section next week (i.e. Deadweek).
- HW11: Not handed in or graded
- Final Exam:
 - **Morning Section: Monday 12/12 (9:45am): In class**
 - **Afternoon Section: Tuesday 12/13 (Noon) : In class**
- Reading for the next few weeks
 - Chapter 2.1-2.3, and 2.6.1-2.6.2
 - 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

C Control Statements

Loop statements:

While statement:

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

Do-While statement:

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

For statement:

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

DO-WHILE Loop

Example:

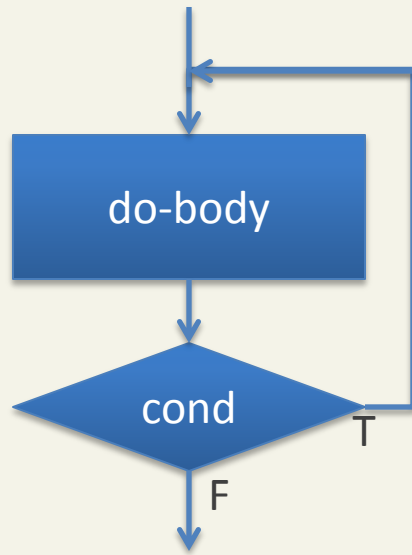
```
void strcpy (char *dst,  
            char *src)  
{  
    char ch;  
    do {  
        ch = *src++;  
        *dst++ = ch;  
    } while (ch);  
}
```

Do-While statement:

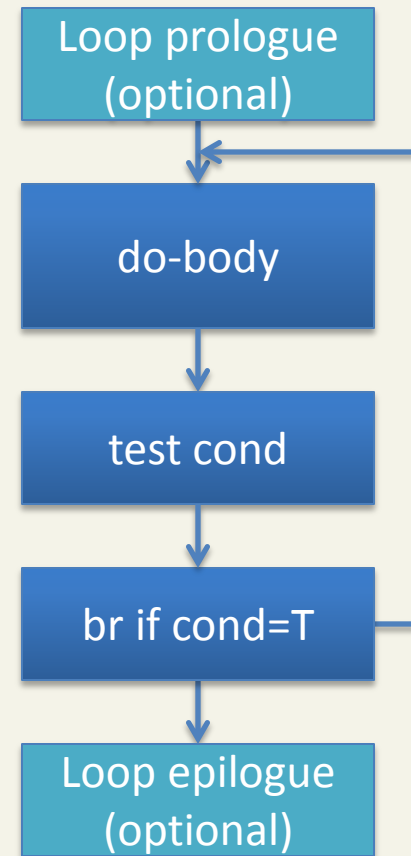
```
do  
{  
    loop-body  
} while (cond);
```

DO-WHILE Loop

Control and Data Flow Graph



Linear Code Layout



DO-WHILE Loop

```
; parameter: dst=>R0, src=>R1
```

```
; reg use: ch=>R2
```

```
strcpy:
```

```
    ;e.g. initialize local vars
```

```
loop:
```

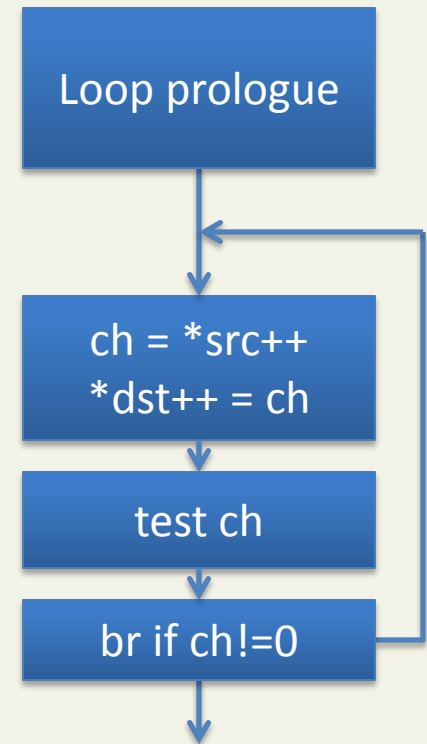
```
    LDRB R2, [R1], #1 ; get byte from src
```

```
    STRB R2, [R0], #1 ; store to dst
```

```
    CMP R2, #0
```

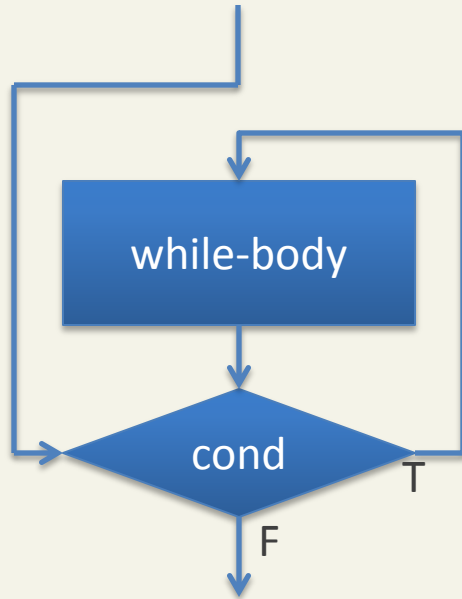
```
    BNE loop          ; ch!=0
```

```
    BX LR           ; return to caller
```

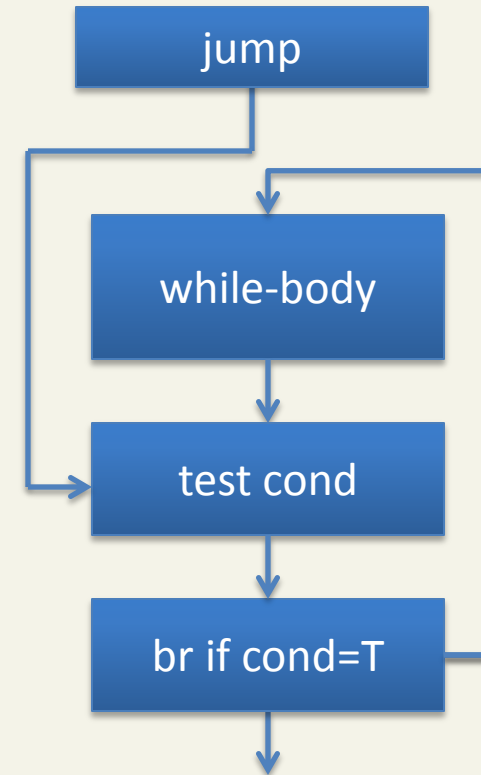


WHILE Loop

Control and Data Flow Graph



Linear Code Layout



(optional prologue and epilogue not shown)

WHILE Loop Example

strlen(): return the length of a C string

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

While statement:
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:

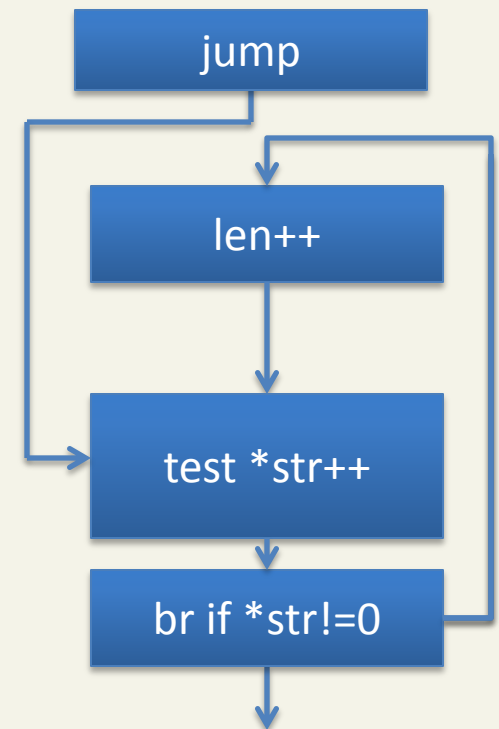
```
LDRB R2, [R0], #1 ; get byte from str
```

```
CMP R2, #0
```

```
BNE loop ; tmp_char != 0
```

```
MOV R0, R3 ; set return value
```

```
BX LR ; return to caller
```



FOR Loop

For statement:

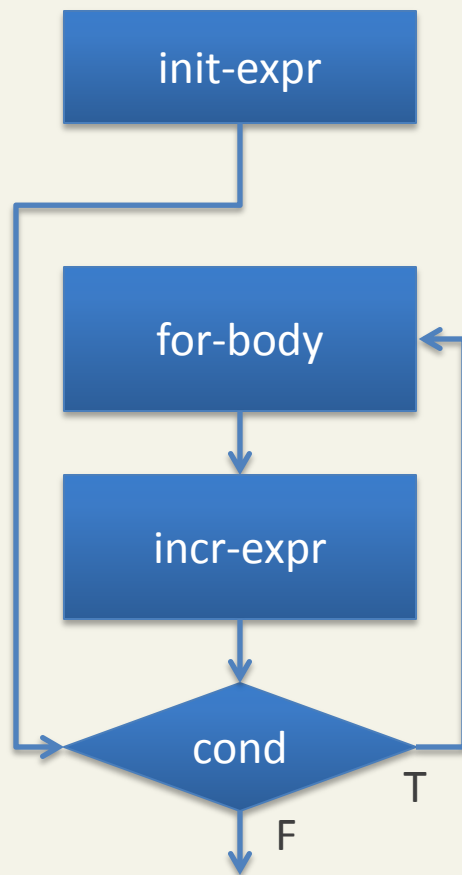
```
for (init-expr; cond-expr; incr-expr)  
{  
    loop-body;  
}
```

Example:

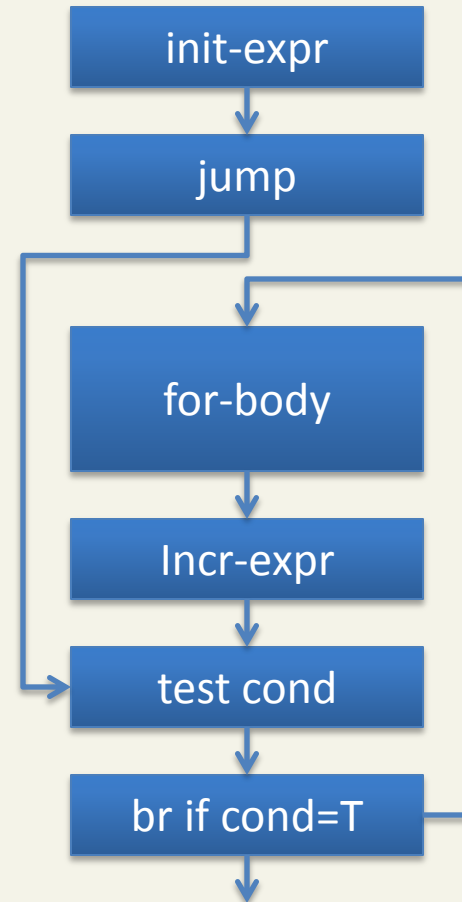
```
unsigned char checksum(unsigned char data[],  
                        int N)  
{  
    unsigned char checksum = 0;  
    for (int i=0; i < N; i++)  
    {  
        checksum ^= data[i];  
    }  
    return checksum;  
}
```

FOR Loop

Control and Data Flow Graph



Linear Code Layout



(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
```

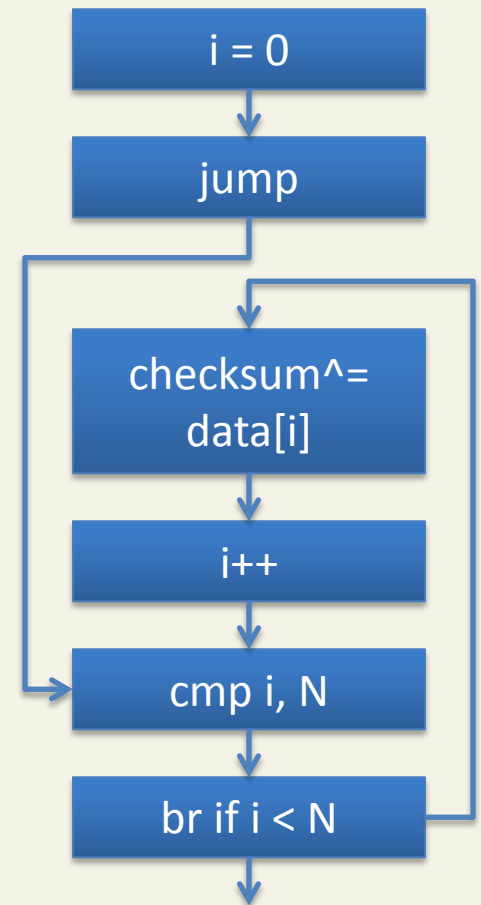
```
loop:
```

```
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
```

```
B cond
```

```
loop:
```

```
LDRB R3, [R0], #1 ;load data[i]
```

```
EOR R2, R3 ;checksum^=data[i]
```

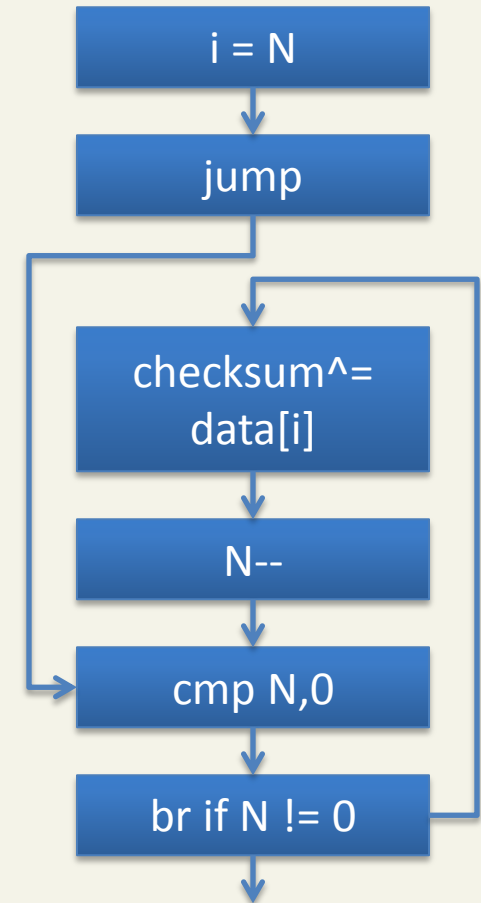
```
SUBS R1, #1 ;N--
```

```
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!

FOR Loop

Another example:

```
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,  
checksum:
```

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

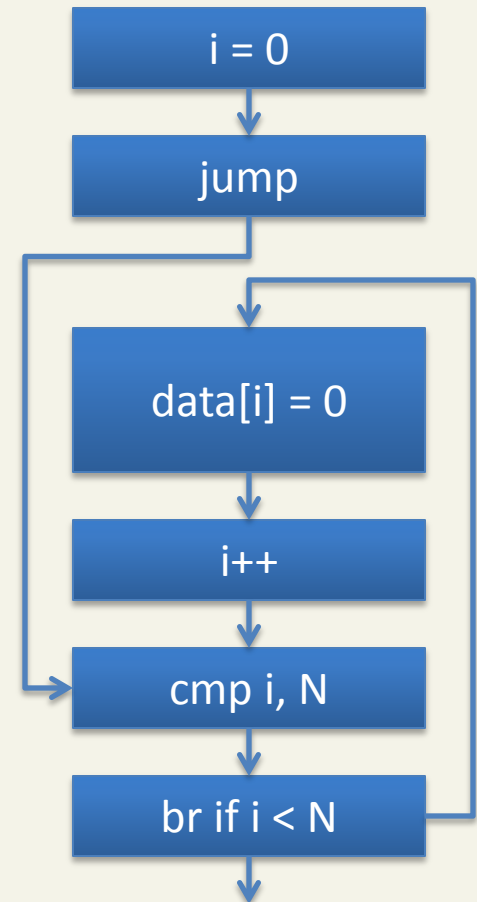
```
loop:
```

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

```
cond:
```

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

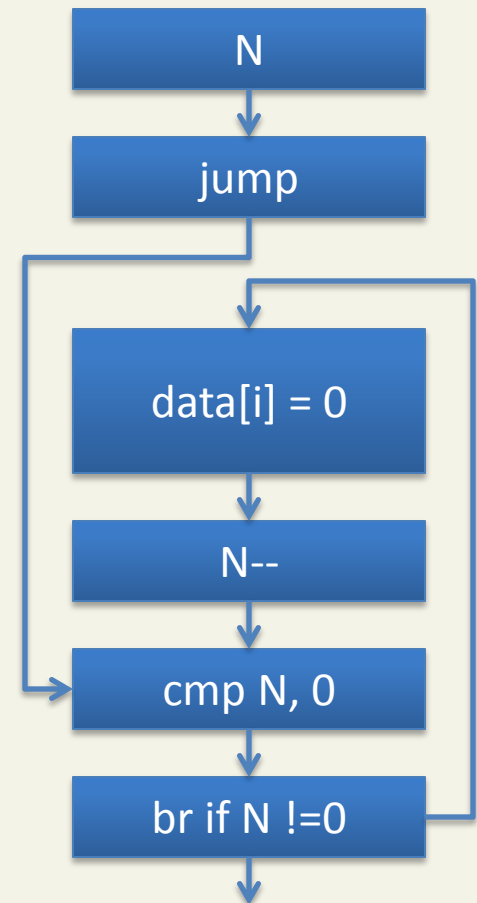
```
    BX LR ;return to caller
```



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
    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!

AVR Interrupt Processing (Not updated for ARM)

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

ISR: Interrupt Service Routine

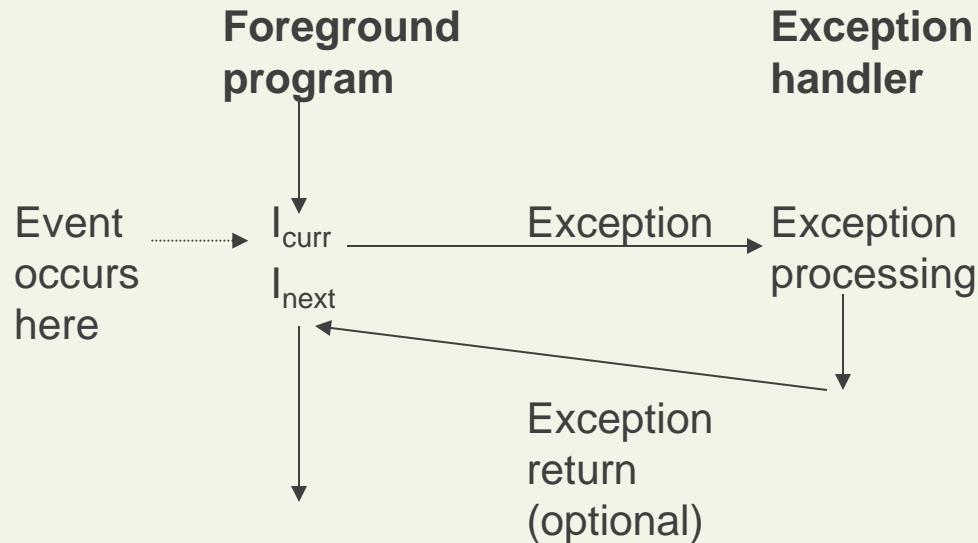
*Interrupt processing will NOT be covered in
Exam 3*

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:

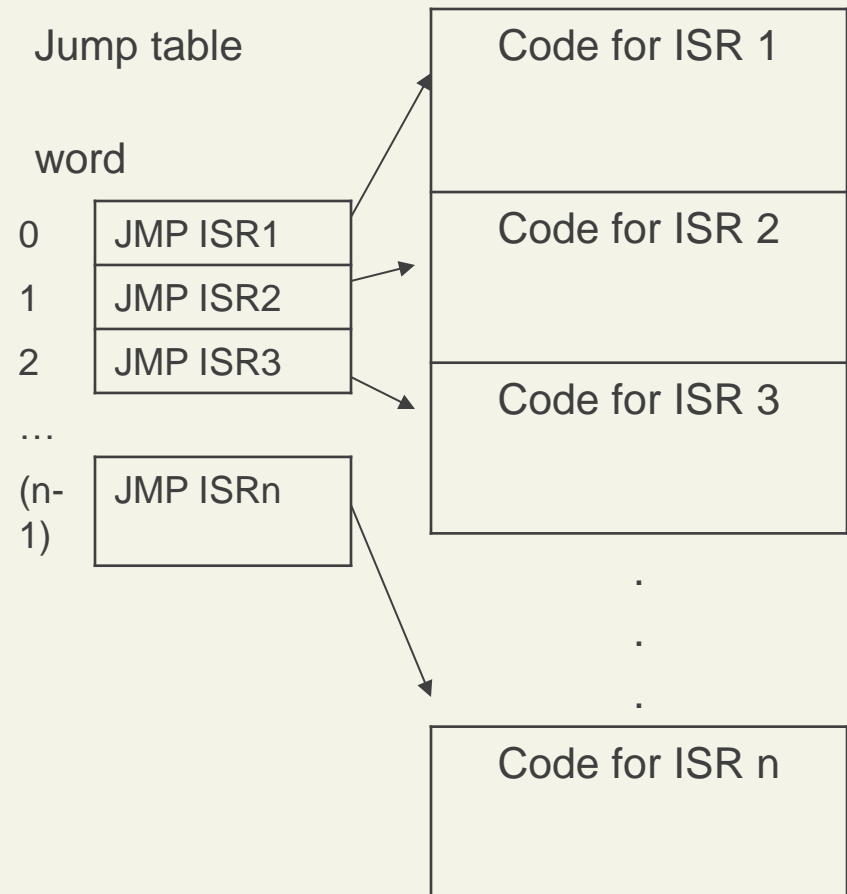
- Create its one own stack frames
- Do not change what's already here in stack
- Restore stack top before exiting

AVR Interrupt Vector Table

Interrupt number: unique identifier number of each event

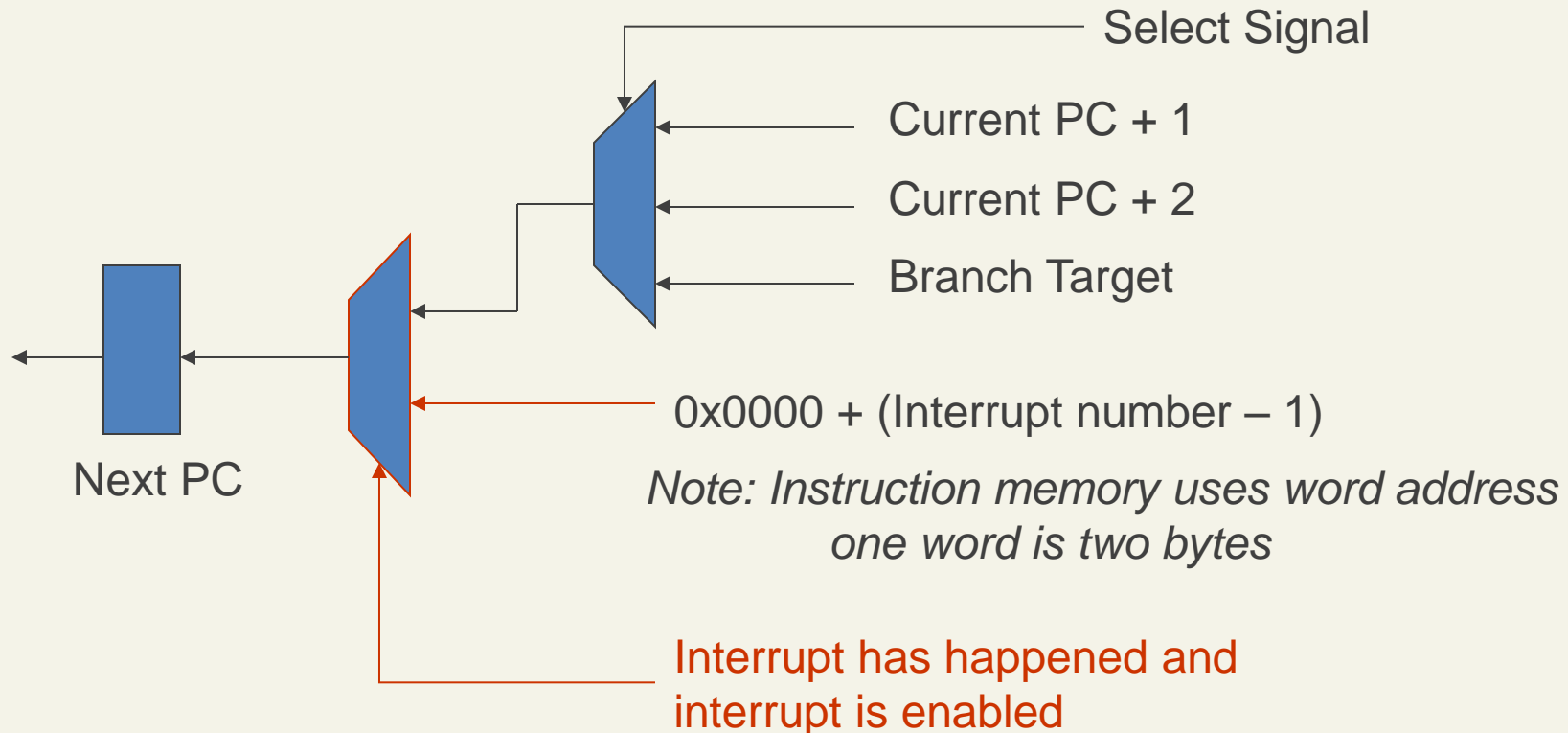
Interrupt Jump Table: Each entry is a jump instruction that jumps to an **Interrupt Service Routine (ISR)**

In AVR, the table starts at 0x0000 by default



AVR Interrupt Vector Table

How to make the exceptional control flow happen?



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

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

35 interrupt sources in total; see page 60 of ATmega128 data sheet

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.