Introduction to Microcontrollers

中山 大 学

数据科学与计算机学院

郭雪梅

Tel:39943108

Email:guoxuem@mail.sysu.edu.cn

Introduction to Embedded Systems

Arithmetic overflow, Branches, Control Structures, Abstraction & Refinement

Agenda

- Recap
  - Debugging
  - I/O
    - Switch and LED interfacing
  - C Programming
    - Random number generator, NOT gate in Keil
- Outline
  - Arithmetic Overflow
  - Conditional Branches
  - Conditional and Iterative Statements
    - `if, while, for` (In assembly and C)
  - Abstraction & Refinement
    - Device Driver

Condition Codes

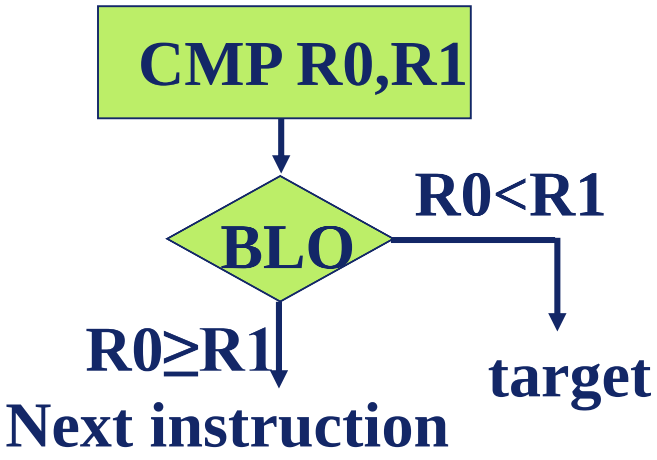
Bit	Name	Meaning after add or sub
N	negative	result is negative
Z	zero	result is zero
V	overflow	signed overflow
C	carry	unsigned overflow

- C set after an unsigned addition if the answer is wrong
- C cleared after an unsigned subtract if the answer is wrong
- V set after a signed addition or subtraction if the answer is wrong

Conditional Branch Instructions

- ❿ **Unsigned conditional branch**
  - follow `SUBS` `CMN` or `CMP`

```
BLO target    ; Branch if unsigned less than  (if C=0, same as BCC)
BLS target    ; Branch if unsigned less than or equal to (if C=0 or Z=1)
BHS target    ; Branch if unsigned greater than or equal to
                  (if C=1, same as BCS)
BHI target    ; Branch if unsigned greater than (if C=1 and Z=0)
```



Conditional Branch Instructions

❿ Signed conditional branch

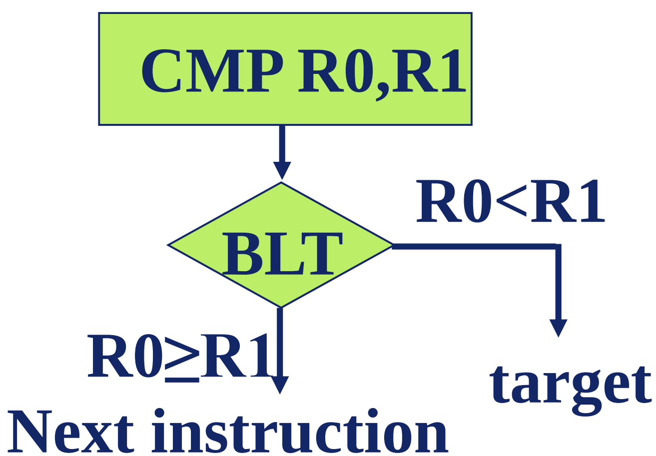
- follow `SUBS` `CMN` or `CMP`

`BLT target` ; if signed less than (if (~N&V | N&~V)=1, i.e. if N≠V)

`BGE target` ; if signed greater than or equal to (if (~N&V | N&~V)=0, i.e. if N=V)

`BGT target` ; if signed greater than (if (Z | ~N&V | N&~V)=0, i.e. if Z=0 and N=V)

`BLE target` ; if signed less than or equal to (if (Z | ~N&V | N&~V)=1, i.e. if Z=1 or N≠V)



Equality Test

Assembly code	C code

`LDR R2, =G ; R2 = &G`
`LDR R0, [R2] ; R0 = G`
`CMP R0, #7 ; is G == 7 ?`
`BNE next1 ; if not, skip`
`BL GEqual7 ; G == 7`
`next1`
	`uint32_t G;`
	`if(G == 7){`
	`GEqual7();`
	`}`

`LDR R2, =G ; R2 = &G`
`LDR R0, [R2] ; R0 = G`
`CMP R0, #7 ; is G != 7 ?`
`BEQ next2 ; if not, skip`
`BL GNotEqual7 ; G != 7`
`next2`
	`if(G != 7){`
	`GNotEqual7();`
	`}`

Program 5.8. Conditional structures that test for equality.

Unsigned Conditional Structures

Assembly code	C code

LDR R2, =G ; R2 = &G
LDR R0, [R2] ; R0 = G
CMP R0, #7 ; is G > 7?
BLS next1 ; if not, skip
BL GGreater7 ; G > 7
next1
	uint32_t G;
	if(G > 7){
	GGreater7();
	}

LDR R2, =G ; R2 = &G
LDR R0, [R2] ; R0 = G
CMP R0, #7 ; is G >= 7?
BLO next2 ; if not, skip
BL GGreaterEq7 ; G >= 7
next2
	if(G >= 7){
	GGreaterEq7();
	}

LDR R2, =G ; R2 = &G
LDR R0, [R2] ; R0 = G
CMP R0, #7 ; is G < 7?
BHS next3 ; if not, skip
BL GLess7 ; G < 7
next3
	if(G < 7){
	GLess7();
	}

LDR R2, =G ; R2 = &G
LDR R0, [R2] ; R0 = G
CMP R0, #7 ; is G <= 7?
BHI next4 ; if not, skip
BL GLessEq7 ; G <= 7
next4
	if(G <= 7){
	GLessEq7();
	}

Program 5.9. Unsigned conditional structures.

Signed Conditional Structures

Assembly code	C code

`LDR R2, =G ; R2 = &G`
`LDR R0, [R2] ; R0 = G`
`CMP R0, #7 ; is G > 7?`
`BLE next1 ; if not, skip`
`BL GGreater7 ; G > 7`
`next1`
	`int32_t G;`
	`if(G > 7){`
	`GGreater7();`
	`}`

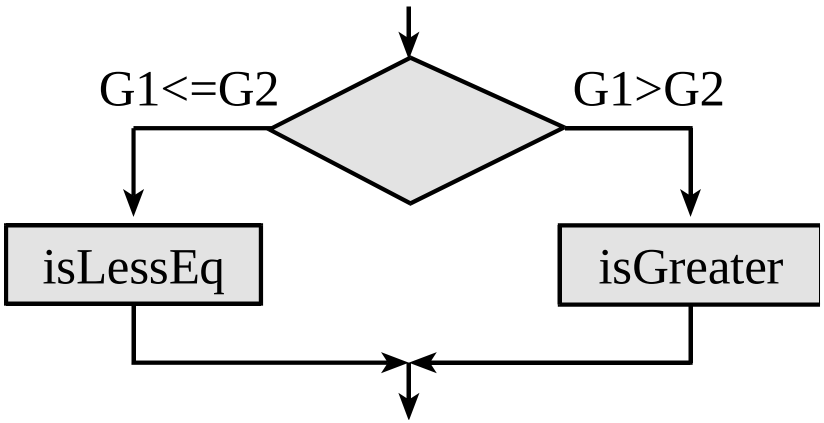
`LDR R2, =G ; R2 = &G`
`LDR R0, [R2] ; R0 = G`
`CMP R0, #7 ; is G >= 7?`
`BLT next2 ; if not, skip`
`BL GGreaterEq7 ; G >= 7`
`next2`
	`if(G >= 7){`
	`GGreaterEq7();`
	`}`

`LDR R2, =G ; R2 = &G`
`LDR R0, [R2] ; R0 = G`
`CMP R0, #7 ; is G < 7?`
`BGE next3 ; if not, skip`
`BL GLess7 ; G < 7`
`next3`
	`if(G < 7){`
	`GLess7();`
	`}`

`LDR R2, =G ; R2 = &G`
`LDR R0, [R2] ; R0 = G`
`CMP R0, #7 ; is G <= 7?`
`BGT next4 ; if not, skip`
`BL GLessEq7 ; G <= 7`
`next4`
	`if(G <= 7){`
	`GLessEq7();`
	`}`

Program 5.11. Signed conditional structures.

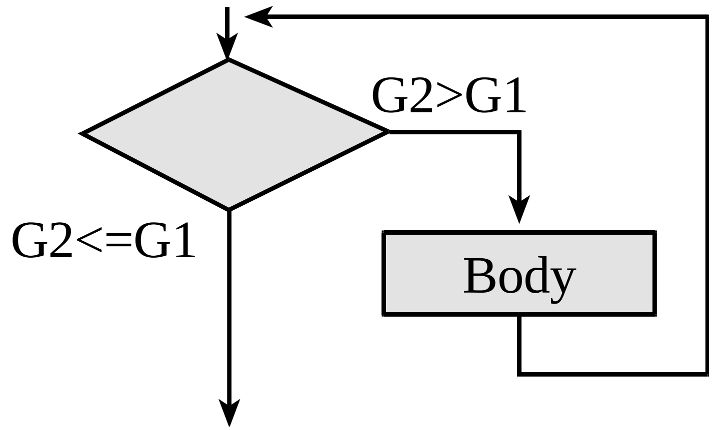
If-then-else




```
     LDR R2, =G1    ; R2 = &G1
     LDR R0, [R2]   ; R0 = G1
     LDR R2, =G2    ; R2 = &G2
     LDR R1, [R2]   ; R1 = G2
     CMP R0, R1     ; is G1 > G2 ?
     BHI high       ; if so, skip to high
low  BL  isLessEq   ; G1 <= G2
     B   next       ; unconditional
high BL  isGreater  ; G1 > G2
next
```

```
uint32_t G1,G2;
if(G1>G2){
  isGreater();
}
else{
  isLessEq();
}
```

While Loops

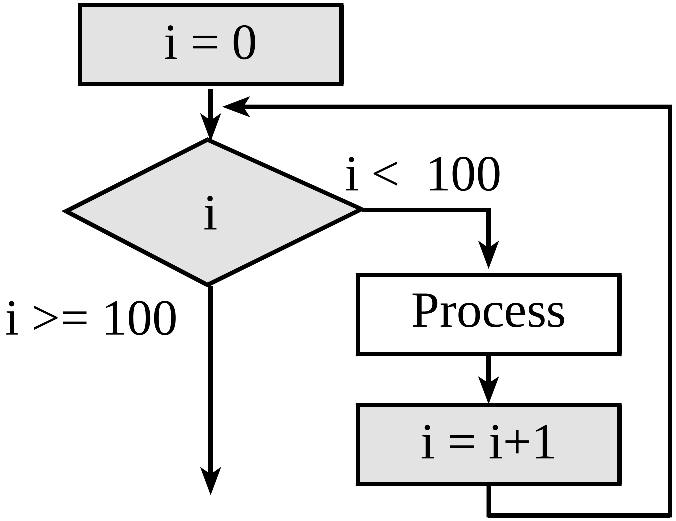



```
     LDR R4, =G1    ; R4 -> G1
     LDR R5, =G2    ; R5 -> G2
loop LDR R0, [R5]   ; R0 =  G2
     LDR R1, [R4]   ; R1 =  G1
     CMP R0, R1     ; is G2 <= G1?
     BLS next       ; if so, skip to
next
     BL  Body     ; body of the loop
     B   loop
next
```

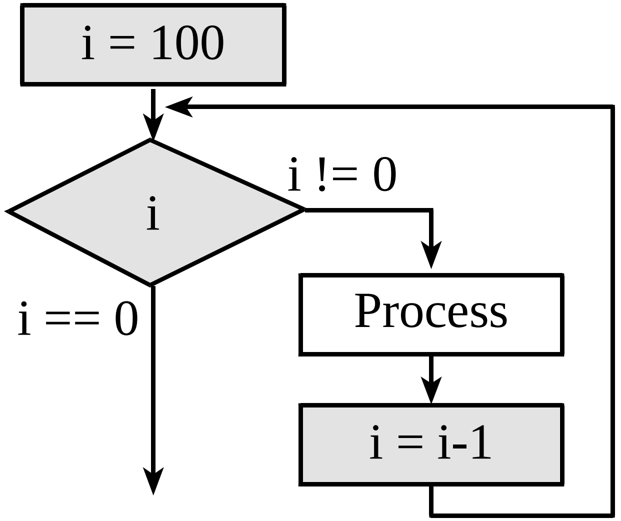
```
uint32_t G1,G2;
while(G2 > G1){
  Body();
}
```

For Loops

```
for(i=0; i<100; i++){
  Process();
}
```




```
for(i=100; i!=0; i--){
  Process();
}
```



For Loops

<pre>   MOV R4, #0   ; R4 = 0
loop CMP R4, #100   ; index >= 100?
     BHS done   ; if so, skip to
done
     BL Process   ; process
function*
     ADD R4, R4, #1 ; R4 = R4 + 1
     B   loop
done</pre>
	<pre>for(i=0; i<100; i++){
	  Process();
	}</pre>

Count up

<pre>   MOV R4, #100   ; R4 = 100
loop BL   Process   ; process
function
     SUBS R4, R4, #1 ; R4 = R4 - 1
     BNE loop
done</pre>
	<pre>for(i=100; i!=0; i--){
	  Process();
	}</pre>

Count down

Registers to pass parameters

High level program	**Subroutine**
1) Sets Registers to contain inputs	
2) Calls subroutine	
	3) Sees the inputs in registers
	4) Performs the action of the subroutine
	5) Places the outputs in registers
6) Registers contain outputs	

Stack to pass parameters

High level program	Subroutine
1) Pushes inputs on the Stack 2) Calls subroutine	
	3) Sees the inputs on stack (pops) 4) Performs the action of the subroutine 5) Pushes outputs on the stack
6) Stack contain outputs (pop) 7) Balance stack	

Assembly Parameter Passing

- Parameters passed using registers/stack
- Parameter passing need not be done using only registers or only stack
  - Some inputs could come in registers and some on stack and outputs could be returned in registers and some on the stack
- Calling Convention
  - Application Binary Interface (ABI)
  - ARM: use registers for first 4 parameters, use stack beyond, return using R0
- Keep in mind that you may want your assembly subroutine to someday be callable from C or callable from someone else’s software

ARM Arch. Procedure Call Standard (AAPCS)

- ABI standard for all ARM architectures
- Use registers R0, R1, R2, and R3 to pass the first four input parameters (in order) into any function, C or assembly.
- We place the return parameter in Register R0.
- Functions can freely modify registers R0–R3 and R12. If a function needs to use R4 through R11, it is necessary to push their current register values onto the stack, use the register, and then pop the old value off the stack before returning.

Parameter-Passing: Registers

Caller

```
;--call a subroutine that
;uses registers for parameter passing
      MOV   R0,#7
      MOV   R1,#3
      BL    Exp
  ;; R2 becomes 7^3 = 343 (0x157)
```

Call by value

- Suggest changes to make it AAPCS

Callee

```
;---------Exp-----------
; Input: R0 and R1 have inputs XX an YY
; Output: R2 has the result XX raised to YY
; Comments: R1 and R2 and non-negative
;         Destroys input R1
XX          RN  0
YY          RN  1
Pow         RN  2
Exp
        ADDS   XX,#0
        BEQ    Zero
        ADDS   YY,#0        ; check if YY is zero
        BEQ    One
        MOV    pow, #1      ; Initial product is 1
More    MUL    pow,XX    ; multiply product with XX
        SUBS   YY,#1        ; Decrement YY
        BNE    More
        B      Retn       ; Done, so return
Zero    MOV    pow,#0    ; XX is 0 so result is 0
        B      Retn
One     MOV    pow,#1    ;  YY is 0 so result is 1
Retn    BX     LR
```

Return by value

Parameter-Passing: Stack

Caller

```
;-------- call a subroutine that
; uses stack for parameter passing
   MOV  R0,#12
     MOV  R1,#5
     MOV  R2,#22
     MOV  R3,#7
     MOV  R4,#18
     PUSH {R0-R4}
  ; Stack has 12,5,22,7 and 18
  ; (with 12 on top)
     BL  Max5
; Call Max5 to find the maximum
;of the five numbers
     POP {R5}
;; R5 has the max element (22)
```

Call by value

Callee

```
;---------Max5-----------
; Input: 5 signed numbers pushed on the stack
; Output: put only the maximum number on the stack
; Comments: The input numbers are removed from stack
numM  RN   1   ; current number
max    RN  2   ; maximum so far
count   RN  0   ; how many elements
Max5
        POP    {max}     ; get top element (top of stack)
                         ; store it in max
        MOV   count,#4  ; 4 more to go
Again  POP   {numM}       ; get next element
        CMP   numM,max
        BLT    Next
        MOV   max, numM  ; new numM is the max
Next   SUBS count,#1  ; one more checked
        BNE    Again
        PUSH  {max}       ; found max so push it on stack
        BX      LR
```

- Flexible style, but not AAPCS

Return by value

Parameter-Passing: Stack & Regs

Caller

```
;------call a subroutine that uses
;both stack and registers for
;parameter passing
    MOV  R0,#6 ; R0 elem count
    MOV  R1,#-14
      MOV  R2,#5
      MOV  R3,#32
      MOV  R4,#-7
      MOV  R5,#0
      MOV  R6,#-5
      PUSH {R4-R6} ; rest on stack
; R0 has element count
; R1-R3 have first 3 elements;
; remaining on Stack
      BL  MinMax
;; R0 has -14 and R1 has 32
;; upon return
```

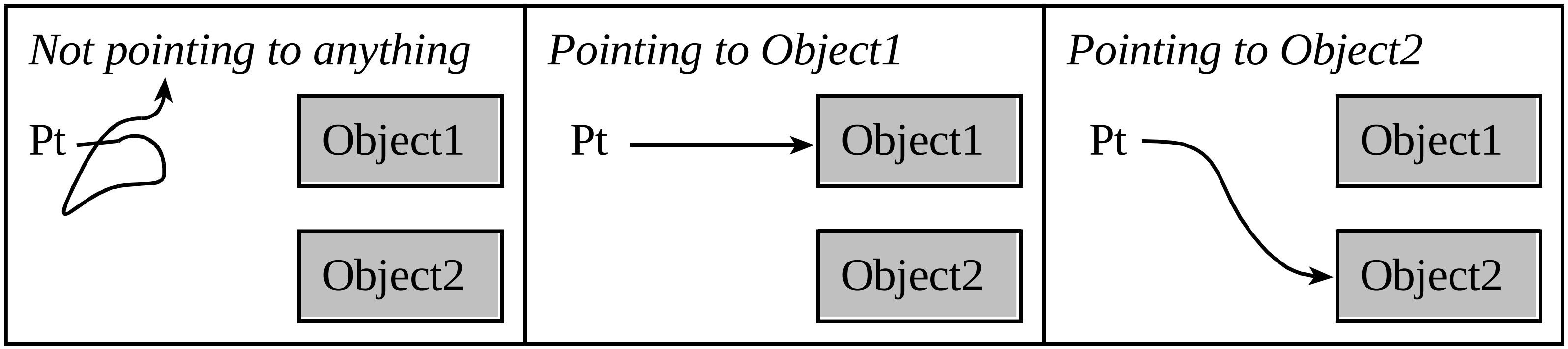
- Not AAPCS

Callee

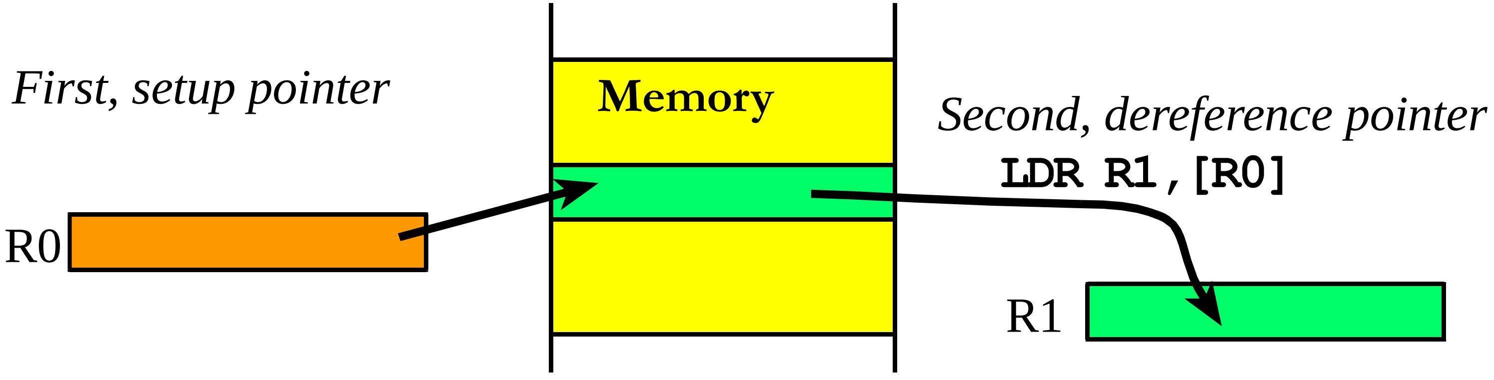
```
;---------MinMax-----------
; Input: N numbers reg+stack; N passed in R0
; Output: Return in R0 the min and R1 the max
; Comments: The input numbers are removed from stack
numMM RN 3; hold the current number in numMM
max    RN  1      ; hold maximum so far in max
min    RN  2
N      RN  0      ; how many elements
MinMax
       PUSH {R1-R3}    ; put all elements on stack
         CMP  N,#0              ; if N is zero nothing to do
         BEQ  DoneMM
         POP  {min}             ; pop top and set it
         MOV  max,min    ;  as the current min and max
loop   SUBS N,#1        ;  decrement and check
         BEQ   DoneMM
         POP   {numMM}
         CMP   numMM,max
         BLT   Chkmin
         MOV   max,numMM  ; new num is the max
Chkmin CMP numMM,min
       BGT   NextMM
         MOV   min,numMM  ; new num is the min
NextMM B   loop
DoneMM MOV R0,min       ; R0 has min
         BX  LR
```

Pointers

- **Pointers are addresses that refer to objects**
  - **Objects may be data, functions or other pointers**
  - **If a register or memory location contains an address we say it points into memory**

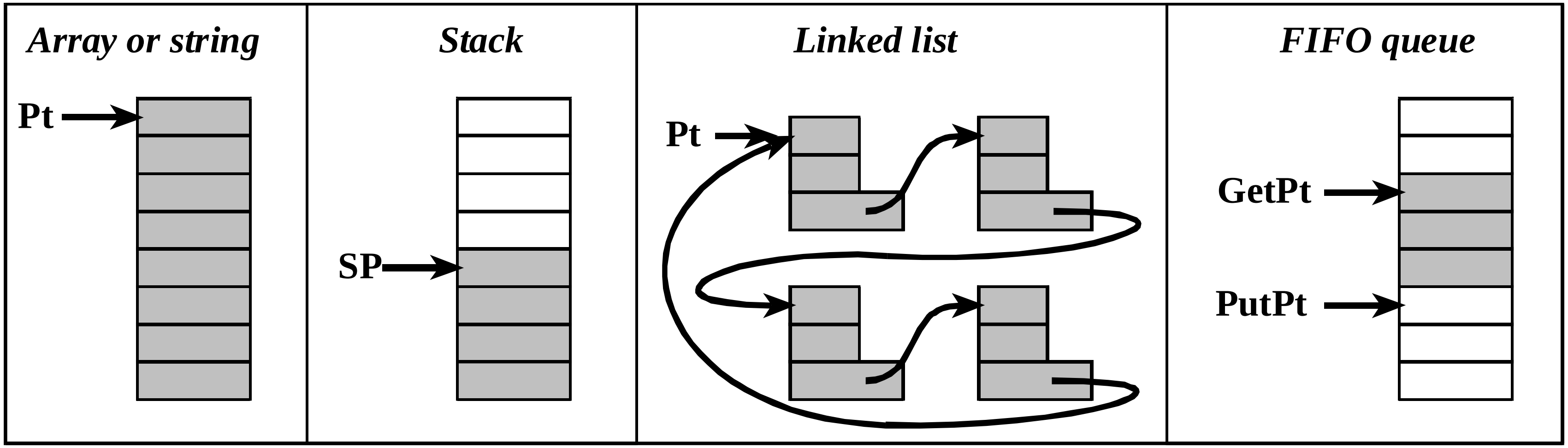


- **Use of indexed addressing mode**



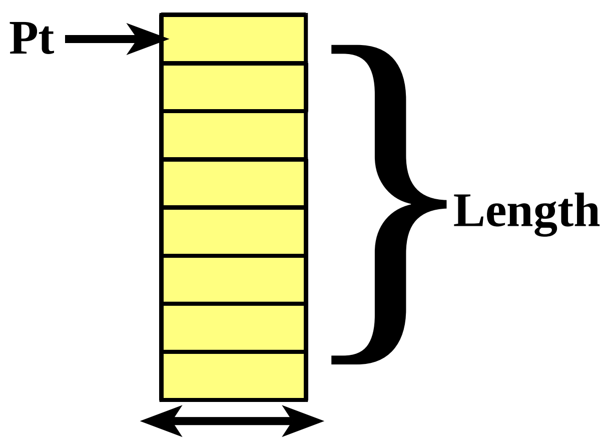
Data Structures

⑩ **Pointers are the basis for data structures**



Data structure { Organization of data
Functions to facilitate access

Arrays



- Random access
- Sequential access
- An array
  - equal precision and
  - allows random access.
- The precision is the size of each element. (n)
- The length is the number of elements (fixed or variable).
- The origin is the index of the first element.
  - zero-origin indexing.

`int32_t Buffer[100]` { Data in consecutive memory; Access: `Buffer[i]`

Array Declaration

- In assembly

```
        AREA  Data
A       SPACE 400
        AREA  |.text|
Prime   DCW 1,2,3,5,7,11,13
```

- In C

```
uint32_t A[100];
const uint16_t Prime[] =
   {1,2,3,5,7,11,13};
```

- Length of the array?
  - One simple mechanism saves the length of the array as the first element

In C:

```
const int8_t
   Data[5]={4,0x05,0x06,0x0A,0x09};
const int16_t
   Powers[6]={5,1,10,100,1000,10000};
Filter int32_t Filter[5]=
{4,0xAABBCCDD,0x00,0xFFFFFFFF,-0x01}
```

In assembly:

```
Data    DCB 4,0x05,0x06,0x0A,0x09
Powers  DCW 5,1,10,100,1000,10000
Filter  DCD 4,0xAABBCCDD, 0x00
        DCD 0xFFFFFFFF,-0x01
```

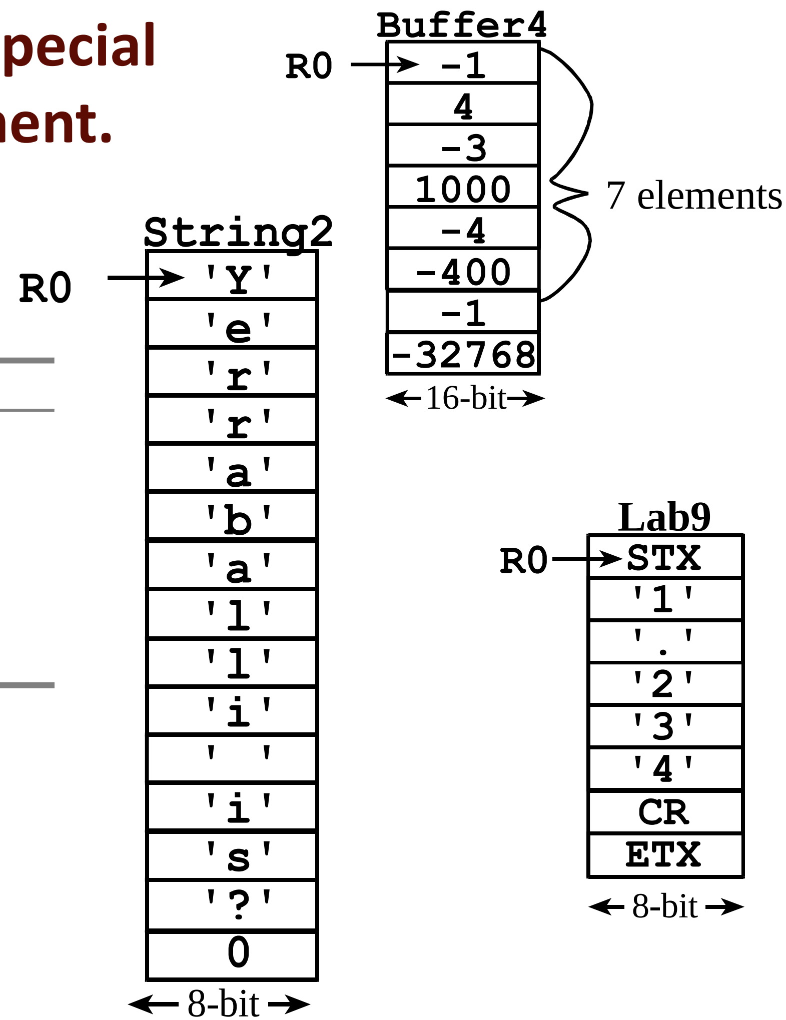
Split declaration on multiple lines if needed

… Arrays

⑩ Length of the Array:

- Alternative mechanism stores a special termination code as the last element.

ASCII	C	code	name
NUL	\0	0x00	null
ETX	\x03	0x03	end of text
EOT	\x04	0x04	end of transmission
FF	\f	0x0C	form feed
CR	\r	0x0D	carriage return
ETB	\x17	0x17	end of transmission block



Array Access

- In general, let `n` be the precision of a zero-origin indexed array in elements.
  - n=1 (8-bit elements)
  - n=2 (16-bit elements)
  - n=4 (32-bit elements)
- If `I` is the index and
- `Base` is the base address of the array,
- then the address of the element at `I` is

```
Base+n*I
```

- In C

```
d = Prime[4];
```

- In assembly

```
LDR   R0,=Prime
LDRH  R1,[R0,#8]
```

Array Access

❑ Indexed access

❖ In C

```
uint32_t i;
sum = 0;
for(i=0; i<7; i++) {
   sum += Prime[i];
}
```

uint16_t

❖ In assembly

```
   LDR R0,=Prime
   MOV R1,#0      ; ofs = 0
   MOV R3,#0      ; sum = 0
lp LDRH R2,[R0,R1] ;Prime[i]
   ADD R3,R3,R2  ; sum
   ADD R1,R1,#2  ; ofs += 2
   CMP R1,#14    ; ofs <= 14?
   BLO lp
```

❑ Pointer access

❖ In C

```
uint16_t *p;
sum = 0;
for(p = Prime;
     p != &Prime[7]; p++){
   sum += *p;
}
```

❖ In assembly

```
   LDR R0,=Prime  ; p = Prime
   ADD R1,R0,#14 ; &Prime[7]
   MOV R3,#0      ; sum = 0
lp LDRH R2,[R0]   ; *p
   ADD R3,R3,R2   ; sum += ..
   ADD R0,R0,#2   ; p++
   CMP R0,R1
   BNE lp
```

Example 6.2

- ***Statement***: Design an exponential function, $y = 10^x$, with a 32-bit output.
- ***Solution***: Since the output is less than 4,294,967,295, the input must be between 0 and 9. One simple solution is to employ a constant word array, as shown below. Each element is 32 bits. In assembly, we define a word constant using DCD, making sure in exists in ROM.

0x00000134	1
0x00000138	10
0x0000013C	100
0x00000140	1,000
0x00000144	10,000
0x00000148	100,000
0x0000014C	1,000,000
0x00000150	10,000,000
0x00000154	100,000,000
0x00000158	1,000,000,000

```
const uint32_t Powers[10]   =

{1,10,100,1000,10000,100000,1000000,10000000,100000000,1000000000};
```

…Solution

C

```
const uint32_t Powers[10]
  ={1,10,100,1000,10000,
    100000,1000000,10000000,
    100000000,1000000000};

uint32_t power(uint32_t x){
  return Powers[x];
}
```

Powers[0]	1
Powers[1]	10
Powers[2]	100
Powers[3]	1,000
Powers[4]	10,000
Powers[5]	100,000
Powers[6]	1,000,000
Powers[7]	10,000,000
Powers[8]	100,000,000
Powers[9]	1,000,000,000

32-bit

Assembly

```
 AREA
   |.text|,CODE,READONLY,ALIGN=2
Powers DCD  1, 10, 100, 1000,
   10000
       DCD  100000, 1000000,
   10000000
       DCD  100000000, 1000000000
;------power----
; Input: R0=x
; Output: R0=10^x
power  LSL R0, R0, #2    ; x = x*4
       LDR R1, =Powers   ; R1=
   &Powers
       LDR R0, [R1, R0]  ;
   y=Powers[x]
       BX  LR
```

Base + Offset

Strings

Problem: ***Write software to output an ASCII string to an output device.***

Solution: **Because the length of the string may be too long to place all the ASCII characters into the registers at the same time, *call by reference* parameter passing will be used. With call by reference, a pointer to the string will be passed. The function `OutString`, will output the string data to the output device.**

The function `OutChar` will be developed later. For now all we need to know is that it outputs a single ASCII character to the serial port.

...Solution

```
Hello DCB "Hello world\n\r",0

;Input: R0 points to string
UART_OutString
     PUSH {R4, LR}
     MOV  R4, R0
loop LDRB R0, [R4]
     CMP  R0, #0   ;done?
     BEQ  done     ;0 sentinel
     BL   UART_OutChar ;print
     ADD  R4, #1   ;next
     B    loop
done POP  {R4, PC}

Start
     LDR R0,=Hello
     BL  UART_OutString
```

```
const uint8_t Hello[]= "Hello world\n\r";

void UART_OutString(uint8_t *pt){
  while(*pt){
    UART_OutChar(*pt);
    pt++;
  }
}
void main(void){
  UART_Init();
  UART_OutString("Hello World");
  while(1){};
}
```

Call by reference

Think about how this is insecure. What might happen if a malicious program called this?

Functional Debugging

Instrumentation: dump into array without filtering

Assume PortA and PortB have strategic 8-bit information.

`SIZE EQU 20`	`#define SIZE 20`
`ABuf SPACE SIZE`	`uint8_t ABuf[SIZE];`
`BBuf SPACE SIZE`	`uint8_t BBuf[SIZE];`
`Cnt SPACE 4`	`uint32_t Cnt;`

`Cnt` **will be used to index into the buffers**

`Cnt` **must be set to index the first element, before the debugging begins.**

```
Cnt = 0;
```

```
LDR R0,=Cnt
MOV R1,#0
STR R1,[R0]
```

... Functional Debugging

The debugging instrument saves the strategic Port information into respective Buffers

```
Save
  PUSH {R0-R4,LR}
  LDR  R0,=Cnt      ;R0 = &Cnt
  LDR  R1,[R0]      ;R1 = Cnt
  CMP  R1,#SIZE
  BHS  done         ;full?
  LDR  R3,=GPIO_PORTA_DATA_R
  LDRB R3,[R3]      ;R3 is PortA
  LDR  R2,=ABuf
  STRB R3,[R2,R1] ;save PortA
  LDR  R3,=GPIO_PORTB_DATA_R
  LDRB R3,[R3]      ;R3 is PortB
  LDR  R2,=BBuf
  STRB R3,[R2,R1] ;save PortB
  ADD  R1,#1
  STR  R1,[R0]      ;save Cnt
done
  POP {R0-R4,PC}
```

```
void Save(void){
 if(Cnt < SIZE){
  ABuf[Cnt]=GPIO_PORTA_DATA_R;
  BBuf[Cnt]=GPIO_PORTB_DATA_R;
  Cnt++;
 }
}
```

Debugging instruments save/restore all registers

Next, you add `BL Save` statements at strategic places within the system.

Use the debugger to display the results after program is done