Patt-Ch8 Data Structures

> *Data Structure = Abstract Data Types, ADT*
>
> Data Type:
>
> - 数据对象集 Objects
> - 数据集合相关的操作集 Operations
>
> Abstract:
>
> - 与存放数据的机器无关
> - 与数据的物理存储结构无关
> - 与实现操作的算法和编程语言无关

8.1 Subroutines (Functions)

8.1.1 JSR/JSRR

> Jump to SubRoutine

JSR
JSRR

Jump to Subroutine

Assembler Formats

JSR LABEL
JSRR BaseR

Encoding

JSR

15–12	11	10–0
0100	1	PCoffset11

JSRR

15–12	11	10–9	8–6	5–0
0100	0	00	BaseR	000000

Examples

```
JSR   QUEUE  ; Put the address of the instruction following JSR into R7;
             ; Jump to QUEUE.
JSRR  R3     ; Put the address of the instruction following JSRR into R7;
             ; Jump to the address contained in R3.
```

Brief Description

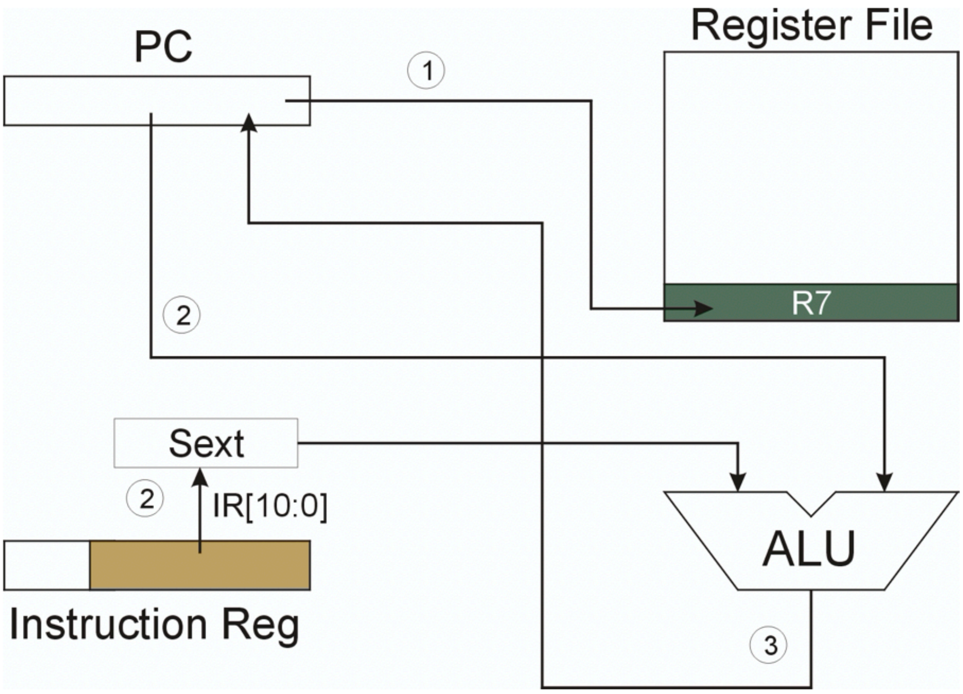
- R7 <- incremented PC ***linkage back to the calling routine***
- PC <- the address of the first instruction of the subroutine
  - JSRR: base register
  - JSR: PC + offset (sign-extending bits [10:0] and adding this value to the incremented PC)

Description

First, the incremented PC is saved in a temporary location. Then the PC is loaded with the address of the first instruction of the subroutine, which will cause an unconditional jump to that address after the current instruction completes execution. The address of the subroutine is obtained from the base register (if bit [11] is 0), or the address is computed by sign-extending bits [10:0] and adding this value to the incremented PC (if bit [11] is 1). Finally, R7 is loaded with the value stored in the temporary location. This is the linkage back to the calling routine.

data path

JSR



NOTE: PC has already been incremented
during instruction fetch stage.

8.1.2 JMP/RET

JMP — Jump
RET — Return from Subroutine

Assembler Formats

JMP BaseR
RET

Encoding

JMP

15–12	11–9	8–6	5–0
1100	000	BaseR	000000

RET

15–12	11–9	8–6	5–0
1100	000	111	000000

Operation

```
PC = BaseR;
```

Description

The program unconditionally jumps to the location specified by the contents of the base register. Bits [8:6] identify the base register.

Examples

JMP R2 ; PC ← R2
RET ; PC ← R7

Note

The RET instruction is a special case of the JMP instruction, normally used in the return from a subroutine. The PC is loaded with the contents of R7, which contains the linkage back to the instruction following the subroutine call instruction.

8.1.3 Save and Restore

- **Why we need saving & restoring?**

Every time an instruction loads a value into a register, the value that was previously in the register is **lost**

- **When we need saving & restoring?**

The value will be destroyed by some subsequent instruction **and** we need it after that subsequent instruction.

- **2 Kinds**

    - Caller save, save&restore happen in A
    - Callee save, save &restore happen in B

> A call B, A is caller, B is callee

```
; MAIN
    JSR A

; Subroutine A
A   ...
    ST R7, SAVER7          ; Caller save
    JSR B
    LD R7, SAVER7
    ...
    RET
    SAVER7 .BLKW 1

; Nest-Subroutine B
B   ST R1, SAVER1          ; Callee save
    ...
    LD R1, SAVER1
    RET
    SAVER1 .BLKW 1
```

How about Recursion?

- Absolutely we can't use `ST` and `LD`, because we can't return MAIN function
- **stack frame**: replace `ST R1, Save1` with `PUSH`, and `LD R1, Save1` with `POP`

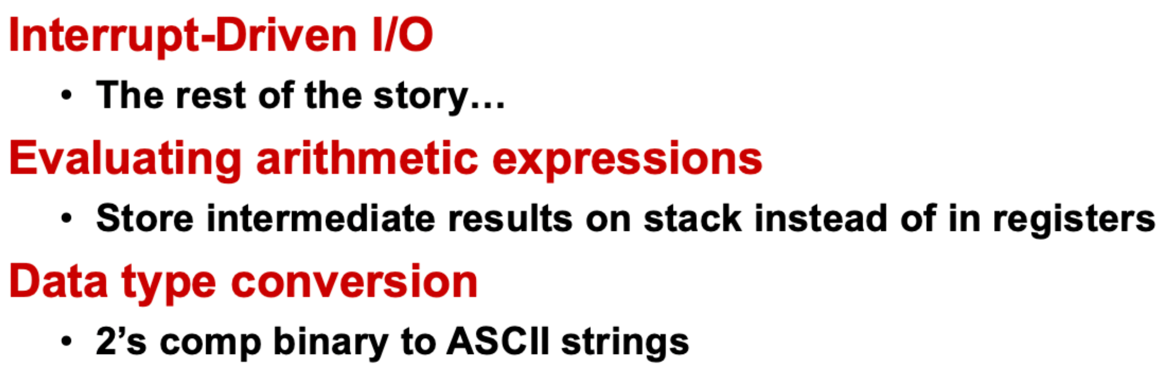
8.1.4 Library Routines

Recommend Reading

8.2 Stack

8.2.1 Stack: Overview

Use Example



Feature

Last In First Out, or ***LIFO***

2 Basic Operations

- PUSH
- POP

8.2.2 A Software Implementation

By convention(按照惯例), **R6** holds the **Top of Stack (TOS) pointer** (both for OS or for User, depending on which is executed now)

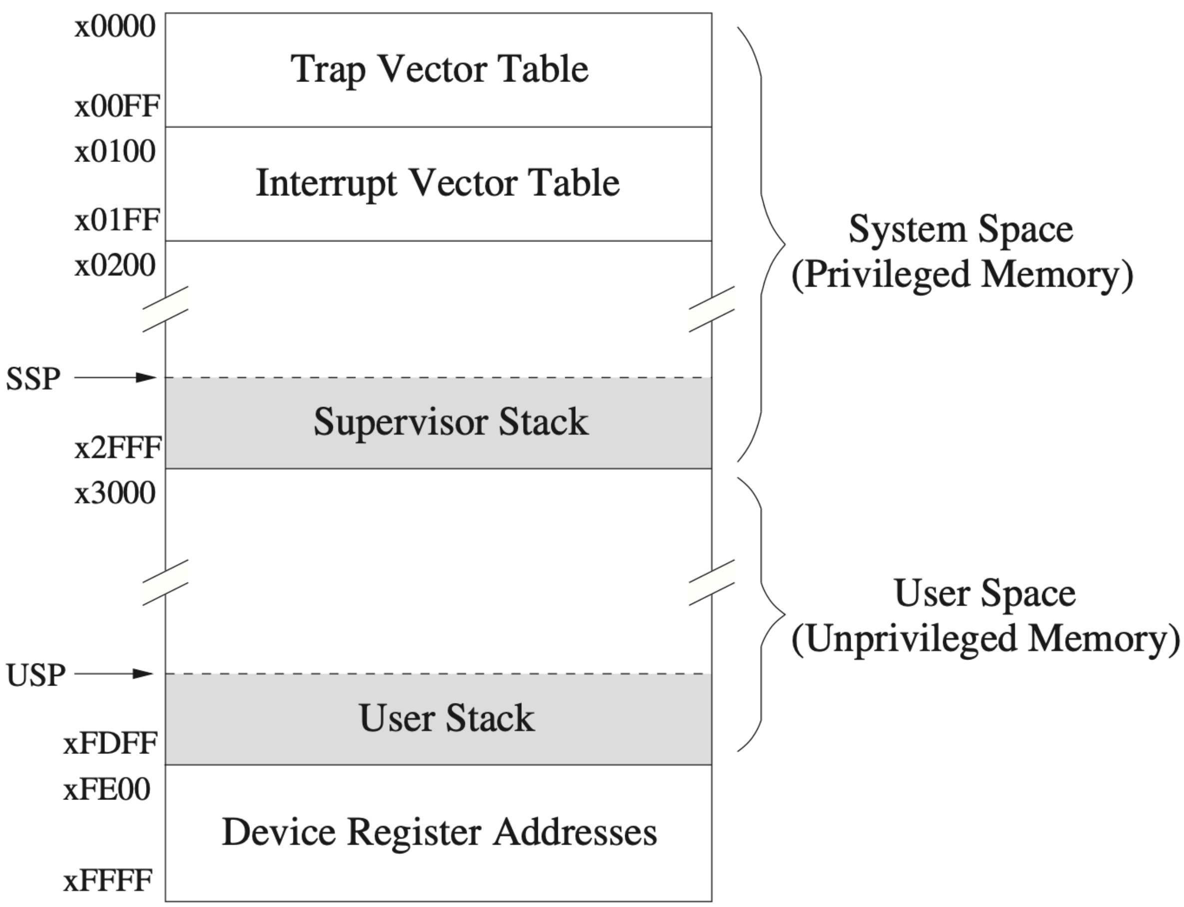


Figure A.1 **Memory map of the LC-3**

- **stack pointer(SP)** : keep track of the **Top of Stack**(TOS)
  - LC-3 use R6 as SP
  - R6 serves as SSP(Supervisor Stack Pointer) or USP(User Stack Pointer), depending on either OS or user's program is executing now
- **Supervisor Stack**
  - Start from `x2FFF`
  - cannot be accessed by usual user
- **User Stack**
  - usually start from `xFDFF`
- **Distinction : Global, Stack and Heap**
  - Global(全局变量区), 与code存放在一起, 从`x3000`开始放，运行时空间不变
  - User Stack, 往上长， 从`xFDFF` 开始放
  - Heap(堆)，malloc分配，free收回，从Global以下开始放

8.2.3 Basic Push and Pop Code

Note :For our implementation, stack **grows downward**

(when item added, TOS moves closer to 0)

```
; Push
          ADD   R6, R6, #-1     ; increment stack ptr
          STR   R0, R6, #0      ; store data

; Pop
          LDR   R0, R6, #0      ; load data from TOS
          ADD   R6, R6, #1      ; decrement stack ptr
```

8.2.4 Pop with Underflow Detection

```
POP       AND R5, R5, #0  ; R5 <- success
          LD  R1, EMPTY   ; EMPTY = -x4000
          ADD R2, R6, R1  ; Compare stack ptr with x4000(EMPTY)
          BRz UNDERFLOW

          LDR R0, R6, #0
          ADD R6, R6, #1
          RET
UNDERFLOW ADD, R5, R5, #1 ; R5 <- failure
          RET
EMPTY     .FILL xC000
```

8.2.5 Push with Overflow Detection

```
PUSH      AND R5, R5, #0  ; R5 <- success
          LD  R1, FULL    ; FULL = -x3FFB
          ADD R2, R1, R6  ; Compare stack ptr with x3FFFB(MAX)
          BRz OVERFLOW

          ADD R6, R6, #-1
          STR R0, R6, #0
          RET
OVERFLOW  ADD, R5, R5, #1 ; R5 <- failure
          RET
FULL      .FILL xC005
```

8.2.6 The Complete Picture

```
;
; Subroutines for carrying out the PUSH and POP functions.  This
; program works with a stack consisting of memory locations x3FFF
; through x3FFB.  R6 is the stack pointer.
;
POP             AND     R5,R5,#0        ; R5 <-- success
                ST      R1,Save1        ; Save registers that
                ST      R2,Save2        ; are needed by POP
                LD      R1,EMPTY        ; EMPTY contains -x4000
                ADD     R2,R6,R1        ; Compare stack pointer to x4000
                BRz     fail_exit       ; Branch if stack is empty
;
                LDR     R0,R6,#0        ; The actual "pop"
                ADD     R6,R6,#1        ; Adjust stack pointer
                BRnzp   success_exit
;
PUSH            AND     R5,R5,#0
                ST      R1,Save1        ; Save registers that
                ST      R2,Save2        ; are needed by PUSH
                LD      R1,FULL         ; FULL contains -x3FFB
                ADD     R2,R6,R1        ; Compare stack pointer to x3FFB
                BRz     fail_exit       ; Branch if stack is full
;
                ADD     R6,R6,#-1       ; Adjust stack pointer
                STR     R0,R6,#0        ; The actual "push"
success_exit    LD      R2,Save2        ; Restore original
                LD      R1,Save1        ; register values
                RET
;
fail_exit       LD      R2,Save2        ; Restore original
                LD      R1,Save1        ; register values
                ADD     R5,R5,#1        ; R5 <-- failure
                RET
;
EMPTY           .FILL   xC000           ; EMPTY contains -x4000
FULL            .FILL   xC005           ; FULL contains  -x3FFB
Save1           .FILL   x0000
Save2           .FILL   x0000
```

Figure 8.11 The stack protocol.

8.3 Queue

> Fisrt In First Out, or ***FIFO***

均摊成本。使用线性队列rear == MAX时，将所有元素同时前移到空间的开头

8.4 Char Strings

8.5 Ordered Lists

> All the elements in the list are arranged according to some orders

Two Basic Operations

- Access
- Update

Two Realization

- Array
- Linked List

8.5.1 Realization of Array

- Access easily: Binary Search
- Update **slowly**: Insertion Sort

8.5.1 Realization of Linked List

- Access slowly
- Update easily

8.6 Array

2D & 3D Array

calculate the address

`A[i,j] = BASE + n[(i * sizeJ) + j]`

`A[i,j,k] = BASE + n[(i * sizeJ * sizeK) + (j * sizeK) + k]`

See Ch16

8.6 Recursion

8.6.1 Save and Restore Mechanism

```
ST  R7, SaveR7
LD  R7, SaveR7
SAveR7  .BLKW   #1
```

Static Save&Restore cannot work in recursion function!

Dynamic Structure: Stack