Chapter 8 Data Structures

8.1 Subroutines

subroutines = **procedures** = **functions**

Target: To enable the programmer to write the code more efficiently.

8.1.1 Call/Return Mechanism

Call instruction: JSR, JSRR

The JSR(R) instruction does **two** things:

1. Loads PC, overwriting the incremented PC that was loaded during FETCH phase of the JSR(R) instruction.
2. Store the incremented PC into R7.

JSR: PCoffset11, bitd[10:0]

JSRR: BaseR

Return instruction: JMP R7

8.1.2 Saving and Restoring Registers

> **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.

Caller & Callee

A calls B $\rightarrow$ A: caller, B: callee

Caller save & Callee save

> How to decide to use which save?
>
> - The one who knows the register will be "polluted" or overwritten **should save the values**.
>
> How to decide the given codes is caller save or callee save?
>
> - In a relationship *A calls B*, :
>   - if the *save(LD) -> restore(ST)* happens in A, which is usually before & after the JSR(R) , it is a caller save.
>
>   e.g.: save R7
>   - if the *save(LD) -> restore(ST)* happens in B, which is usually at the beginning of the subroutine & before the JMP R7 , it is a callee save.

8.1.3 Library Routines

Chapter 8.1.4 in the textbook is recommend reading.

8.2 The Stack

8.2.1 ADT

An Abstract Data Type (ADT) is a data type that is organized in such a way that the specification on the **objects** and specification of the **operations** on the objects are separated from the **representation** of the objects and the **implementation** on the operations.

8.2.2 The defining notion of a stack

LIFO, that is the last thing you stored in the stack is the first thing you remove from it.

8.2.3 Two typical operations on stack

- **PUSH**: to store a value into the stack, it will be at the top of the stack
- **POP**: to get and remove the value of the top of the stack.

8.2.4 Implementation in Memmory

We use a ***stack pointer(SP)*** to keep track of the top of the stack. In LC-3, we use **R6** as a SP.

In LC-3, the stack **grows to zero**, which means that the stack will grows to lower location, for example, from x3FFF to x3FFFE, x3FFD and so on.

System Stack: x0000 - x2FFF, which can not be accessed by usual user.

So we can make our stack start and end from anywhere between **x3000 and xFCFF**. (xFD00 to xFFFF is used for I/O)

Usually, we choose xFCFF as the beginning of our user stack.

8.2.5 Push & Pop

To improve our efficiency, we **do not physically move** our data during pop operation.

So the basic push & pop operation is as follow:

PUSH

Suppose the value has been stored in $r0$ previously.

```
PUSH:   ADD r6, r6, #-1 ; Move the SP
        STR r0, r6, #0  ; Store the value
```

POP

Suppose the value is asked to store in $r0$.

```
POP:    LDR r0, r6, #0  ; Get the value
        ADD r6, r6, #1  ; Move the SP to remove the value
```

8.2.6 Overflow & Underflow

Overflow:

The stack has been **full** when trying inserting a new value. Only happens in PUSH.

Underflow:

The stack has been **empty** when trying removing a value. Only happens in POP.

RET instruction

It is the same as **JMP r7**;

Detect & Handle Overflow in PUSH

Suppose the Stack begins at x3FFF and its capacity is 5. And we handle Overflow by set $r5$ (make $r5$ be 1).

Key point: Judge if SP points to the lowest location.

```
PUSH:   AND r5, r5, #0  ; Initialize r5
        LD  r1, MAX     ; Initialize r1 with the opposite number of the lowest location
        ADD r2, r6, r1
        BRz Failure     ; Judge if SP points to the lowest location
        ADD r6, r6, #-1 ; No overflow, just do PUSH
        STR r0, r6, #0
        RET             ; Return to caller
Failure ADD r5, r5, #1  ; Set r5
        RET             ; Return to caller
MAX     .FILL   xC005   ; The opposite number of the lowest location
```

HINT:

The lowest location should be: x3FFB (x3FFF - #5), which is

```
0011 1111 1111 1011
```

Then we get its opposite number by calculating its 2's complement number:

```
1100 0000 0000 0101
```

which is xC005

Detect & Handle Underflow in POP

Suppose the Stack begins at x3FFF (its capacity has nothing to do with underflow). And we handle Overflow by set $r5$ (make $r5$ be 1).

Key point: Judge if SP points to the highest location.

```
POP:    AND r5, r5, #0  ; Initialize r5
        LD  r1, EMPTY   ; Initialize r1 with the opposite number of the highest
location
        ADD r2, r6, r1
        BRz Failure     ; Judge if SP points to the lowest location
        STR r0, r6, #0  ; No overflow, just do POP
        ADD r6, r6, #1
        RET             ; Return to caller
Failure ADD r5, r5, #1  ; Set r5
        RET             ; Return to caller
EMPTY   .FILL   xC000   ; The opposite number of the lowest location
```

HINT:

The highest location should be: x4000 (x3FFF + #1), which is

```
0100 0000 0000 0000
```

Then we get its opposite number by calculating its 2's complement number:

```
1100 0000 0000 0000
```

which is xC000