中国科学技术大学
University of Science and Technology of China

计算系统概论A
Introduction to Computing Systems
(CS1002A.03)

Chapter 8
Data Structures

陈俊仕
cjuns@ustc.edu.cn
2023 Fall

计算机科学与技术学院
School of Computer Science and Technology

Outline

Outline

1. Review
2. **Subroutines**
3. Control Instructions for Subroutines
4. Memory Model for Program Execution
5. The Stack
6. Implementing Functions in C

Subroutines

- A subroutine is a program fragment that. . .
  - Resides in user space (*i.e*, not in OS)
  - Performs a well-defined task
  - Is invoked (called) multiple times by a user program
  - Returns control to the calling program when finished
- Virtues
  - Reuse code without re-typing it (and debugging it!)
  - Divide task into parts (or among multiple programmers)
  - Use vendor-supplied library of useful routines that one software engineer writes a program that requires such fragments and another software engineer writes the fragments.
    - math library
    - square root, sine, and arctangent, etc.
- In C language, called function; In other languages, called procedures, subroutines, methods ...

A simple illustration of a part of a program

```
; Service Routine for Keyboard Input
        .ORIG   x04A0           ;System call starting address

;START  ST      R7,SaveR7       ;Save the linkage back to the
;                               ;program?
        ST      R1,SaveR1       ;Save the values in the registers
        ST      R2,SaveR2       ;that are used so that they can
        ST      R3,SaveR3       ;be restored before RET

;Output Newline on CRT
        LD      R2,Newline
L1      LDI     R3,DSR          ;Check DDR—is it free?
        BRzp    L1              ;Loop until monitor is ready
        STI     R2,DDR          ;Move cursor to new clean line
;
;Output "Input a character"
        LEA     R1,Prompt       ;Prompt is starting address
                                ;of prompt string
Loop    LDR     R0,R1,#0        ;Get next prompt character
        BRzp    Input           ;Check for end of prompt string
L2      LDI     R3,DSR
        BRzp    L2
        STI     R0,DDR          ;Write next character of prompt
                                ;string
        ADD     R1,R1,#1        ;Increment prompt point
        BRnzp   Loop
```

```
Label   LDI     R3,DSR
        BRzp    Label
        STI     Reg,DDR
```

A simple illustration of a part of a program

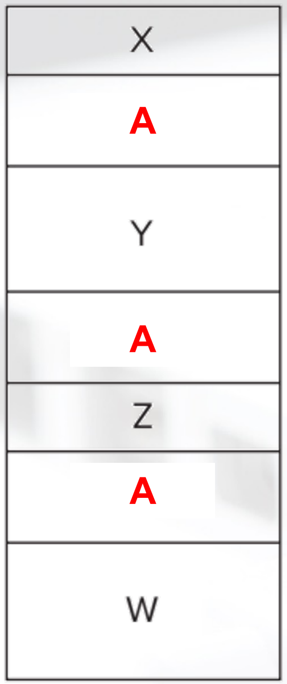
```
;Input a character from KB
Input  LDI     R3,KBSR       ;Has a character been typed?
       BRzp    Input
       LDI     R0,KBDR       ;Load it into R0

;Echo the character on CRT
L3     LDI     R3,DSR
       BRzp    L3
       STI     R0,DDR        ;Echo input character to the
                             ;monitor
;Output Newline on CRT
L4     LDI     R3,DSR        ;Check CRTDR—is it free?
       BRzp    L4
       STI     R2,DDR        ;Move cursor to new clean line

;Restore
       LD      R1,SaveR1     ;Service routine done, restore
       LD      R2,SaveR2     ;original values in registers.
       LD      R3,SaveR3     ;
;      LD      R7,SaveR7     ;Restore linkage back
;                            ;prior to RET?
       RET                   ;Return to calling program
```

```
Label  LDI     R3,DSR
       BRzp    Label
       STI     Reg,DDR
```

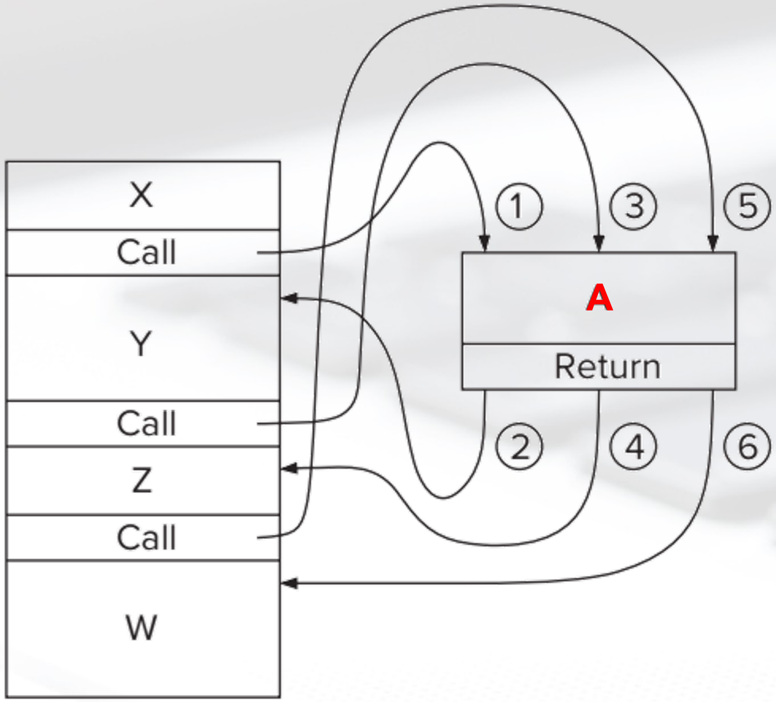
A simple illustration of a part of a program

```
;Memory for registers saved
;         SaveR7   .FILL x0000
          SaveR1   .FILL x0000
          SaveR2   .FILL x0000
          SaveR3   .FILL x0000

          DSR      .FILL xF3FC
          DDR      .FILL xF3FF
          KBSR     .FILL xF400
          KBDR     .FILL xF401
 ;
          Newline  .FILL x000A      ;ASCII code for newline
          Prompt   .STRINGZ "Input a character>"
                   .END
```

The Call/Return Mechanism



(a) Without subroutines



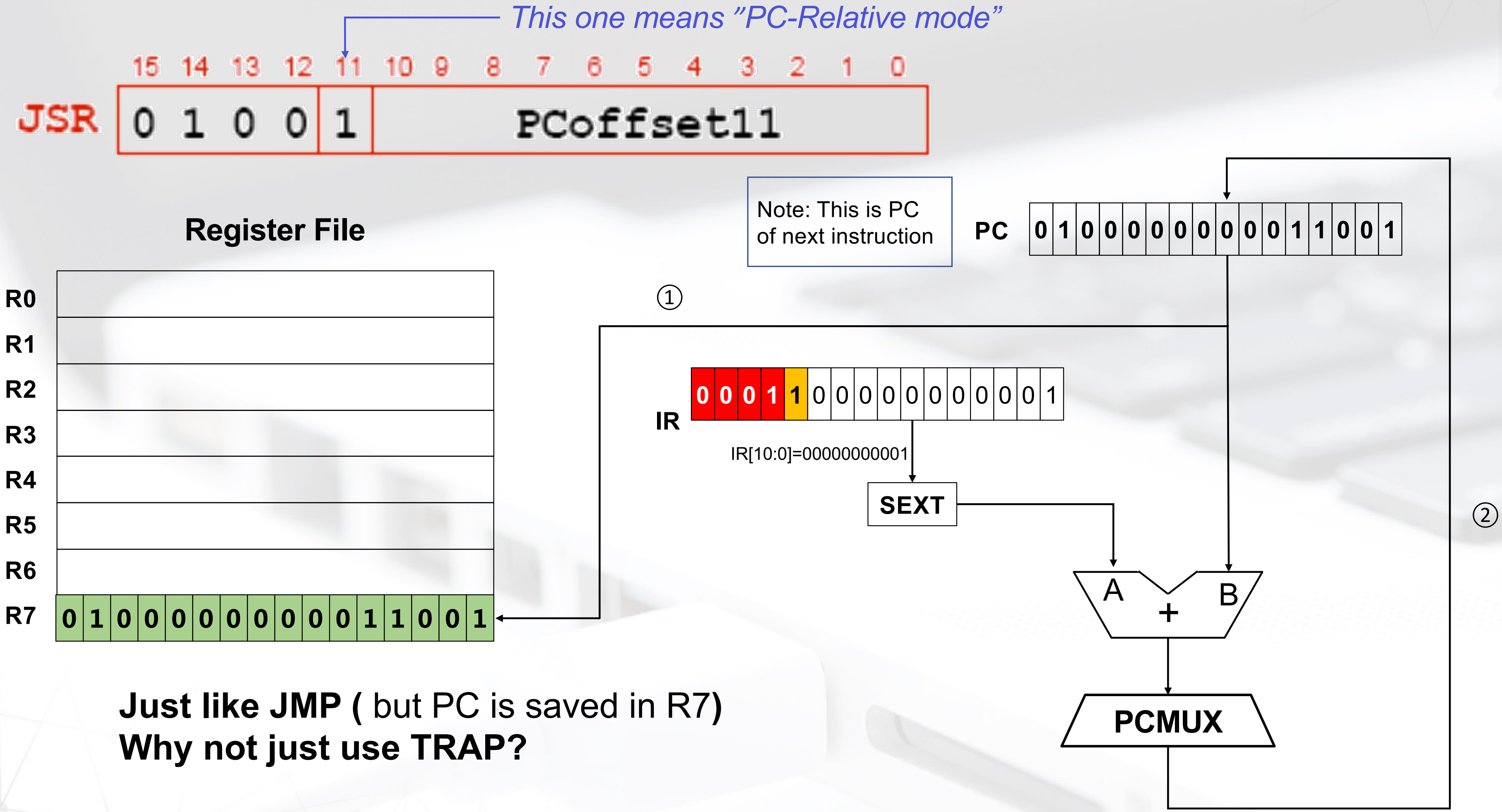
(b) With subroutines

Outline

Control Instructions for Subroutines

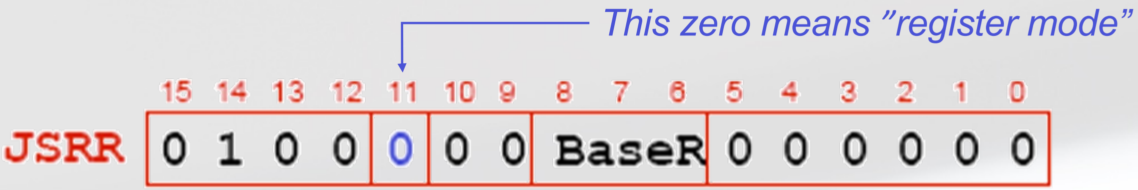
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BR	0	0	0	0	n	z	p	PCoffset9								
JSR	0	1	0	0	1	PCoffset11										
JSRR	0	1	0	0	0	0	0	BaseR			0	0	0	0	0	0
RTI	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
JMP	1	1	0	0	0	0	0	BaseR			0	0	0	0	0	0
RET	1	1	0	0	0	0	0	1	1	1	0	0	0	0	0	0
TRAP	1	1	1	1	0	0	0	0	TrapVector8							

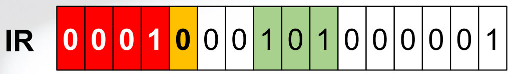
JSR (PC-Relative)

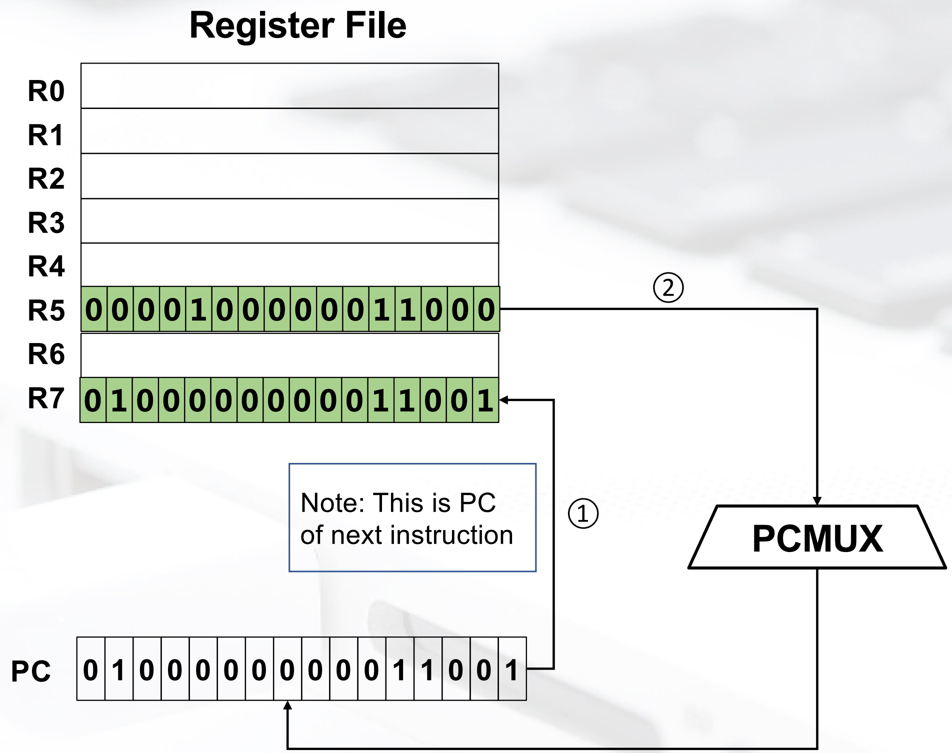


Just like JMP (but PC is saved in R7**)**
Why not just use TRAP?

JSRR (Register)







Virtues of JSRR?

RET instruction

RET – return instruction

- **How to return**
  - — Place address in R7 in PC, Return the execution to the last calling point.
- **PC ← (R7)**

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RET (JMP R7)	1	1	0	0	0	0	0	1	1	1	0	0	0	0	0	0

Example: Negate the value in R0

```
TwosComp    NOT    R0,R0        ;flip bits

            ADD    R0,R0,#1     ;add one

            RET                 ;return to caller
```

To call from a program

```
;need to compute R4 = R1-R3
            ADD    R0,R3,#0     ;copy R3 to R0
            JSR    TwosComp     ;negate
            ADD    R4,R1,R0     ;add to R1
            ...
```

Using Subroutines

- **Programmer must know**
  - **Address:** or at least a label that will be bound to its address
  - **Function:** what it does
    - NOTE: The programmer does not need to know *how* the subroutine works, but what changes are visible in the machine’s state after the routine has run
  - **Arguments:** what they are and where they are placed
  - **Return values:** what they are and where they are placed

Passing Information To Subroutines

Argument(s)

- **Value passed in to a subroutine is called an argument**
- **This is a value needed by the subroutine to do its job**
- **Examples**
  - — TwosComp: R0 is number to be negated
  - — OUT: R0 is character to be printed
  - — PUTS: R0 is *address* of string to be printed

How?

- **In registers (simple, fast, but limited number)**
- **In memory (many, but awkward, expensive)**
- **Both**

Getting Values From Subroutines

Return Values

- A value passed out of a subroutine is called a return value
- This is the value that you called the subroutine to compute
- Examples
  - TwosComp: negated value is returned in R0
  - GETC: character read from the keyboard is returned in R0

How?

- Registers, memory, or both
- Single return value in register most common

A problem when dealing with subroutines

- We have known that every time an instruction loads a value into a register, the value that was previously in that register is lost. Thus, we need to save the value in a register
  - if that value will be destroyed by some subsequent instruction, and
  - if we will need it after that subsequent instruction.
- Caution Using JSR, JSRR, and TRAP
  - You MUST save R7 if you call any other subroutine using JSR,JSRR or TRAP

Caution Using TRAPs (*"caller-save" in User code*)

```
                LEA    R3,BLOCK      ;Init. To first loc.
                LD     R6,ASCII      ;Char->digit template
                LD     R7,COUNT      ;Init. to 10
AGAIN           TRAP   x23           ;Get char
                ADD    R0,R0,R6      ;Convert to number
                STR    R0,R3,#0      ;Store number
                ADD    R3,R3,#1      ;Incr pointer
                ADD    R7,R7,-1      ;Decr counter
                BRp    AGAIN         ;More?
                BRnzp  NEXT_TASK
ASCII           .FILL xFFD0   ;Negative of x0023
COUNT           .FILL #10
BLOCK           .BLKW #10
```

Caution Using TRAPs (*"caller-save"* *in User code*)

```
                LEA    R3,BLOCK      ;Init. To first loc.
                LD     R6,ASCII      ;Char->digit template
                LD     R7,COUNT      ;Init. to 10
AGAIN           ST     R7,SaveR7
                TRAP   x23           ;Get char
                LD     R7,SaveR7
                ADD    R0,R0,R6      ;Convert to number
                STR    R0,R3,#0      ;Store number
                ADD    R3,R3,#1      ;Incr pointer
                ADD    R7,R7,-1      ;Decr counter
                BRp    AGAIN         ;More?
                BRnzp  NEXT_TASK
SaveR7          .BKLW 1
ASCII           .FILL xFFD0   ;Negative of x0023
COUNT           .FILL #10
BLOCK           .BLKW #10
```

Saving and Restoring Registers

- **Called routine** => *"callee-save"*
  - **Before start, save registers that will be altered**
    (except output regs)
  - **Before return, restore those same registers**
    (again, except output regs)
  - **Values are saved by storing them in memory**
- **Calling routine** => *"caller-save"*
  - **If register value needed later, save register destroyed by own instructions or by called routines (if known)**
    - Save R7 before TRAP
  - **Or avoid using those registers altogether**
- *LC-3: By convention,* ***callee-saved when possible***
  - **Other ISAs use a more efficient combination of caller- and callee-save**

Saving and Restore Registers

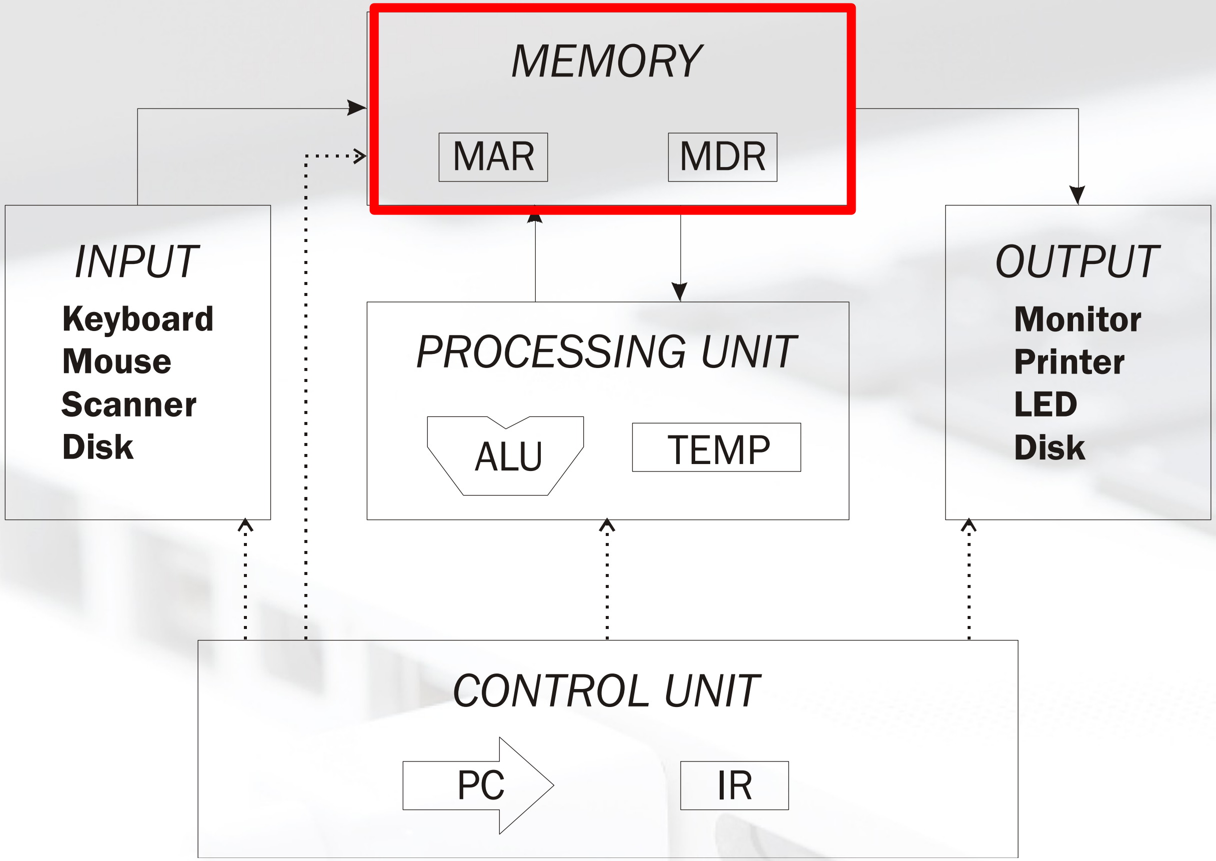
- Like service routines, must save and restore registers
  - Who saves what is part of the calling convention
- **Generally use "callee-save" strategy, except for return values**
  - Same as trap service routines
  - Save anything that subroutine alters internally that shouldn't be visible when the subroutine returns
  - Restore incoming arguments to original values (unless overwritten by return value)
- Remember
  - **You MUST save R7 if you call any other subroutine or trap**
  - Otherwise, you won't be able to return!

Subroutine Template

```
SUB_NAME
        ;Register Saving
        ST R0, SUB_R0
        ST R1, SUB_R1
        …
        ST R6, SUB_R6
        ST R7, SUB_R7;Return address

        ;***Code***

        ;Register Restoring
        LD R0, SUB_R0
        LD R1, SUB_R1
        …
        LD R6, SUB_R6
        LD R7, SUB_R7            ;Return address
        RET
```

Outline

Review: Memory in Von Neumann Model



Review: Using Memory

- Memory
  - Just a big “array”
  - “Indexed” by address
  - Accessed with loads and stores instructions
- LD/LDR/LDI
  - Read a word out of memory
  - Use different addressing mode
- ST/STR/STI
  - Place a word in memory
  - Use different addressing mode

Memory

Address	Value
x0000	x00A0
x0001	x5007
x0002	x0201
x0003	x0203
x0004	x3002
	. . .
xFFFC	x5007
xFFFD	x0201
xFFFE	x0203
xFFFF	x3002

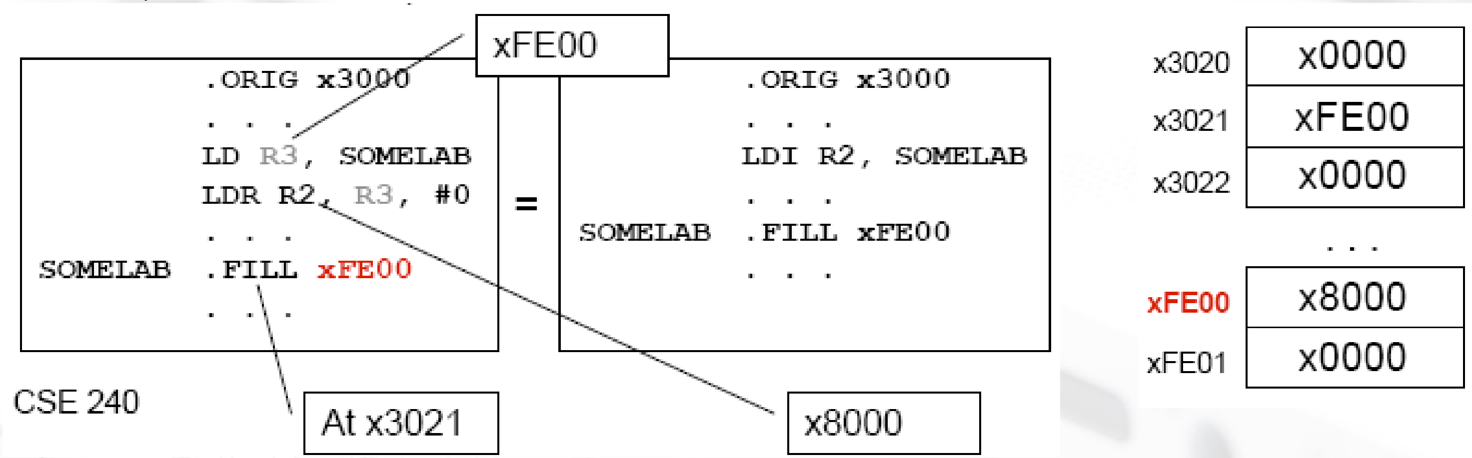
Review: Using Memory

Problem

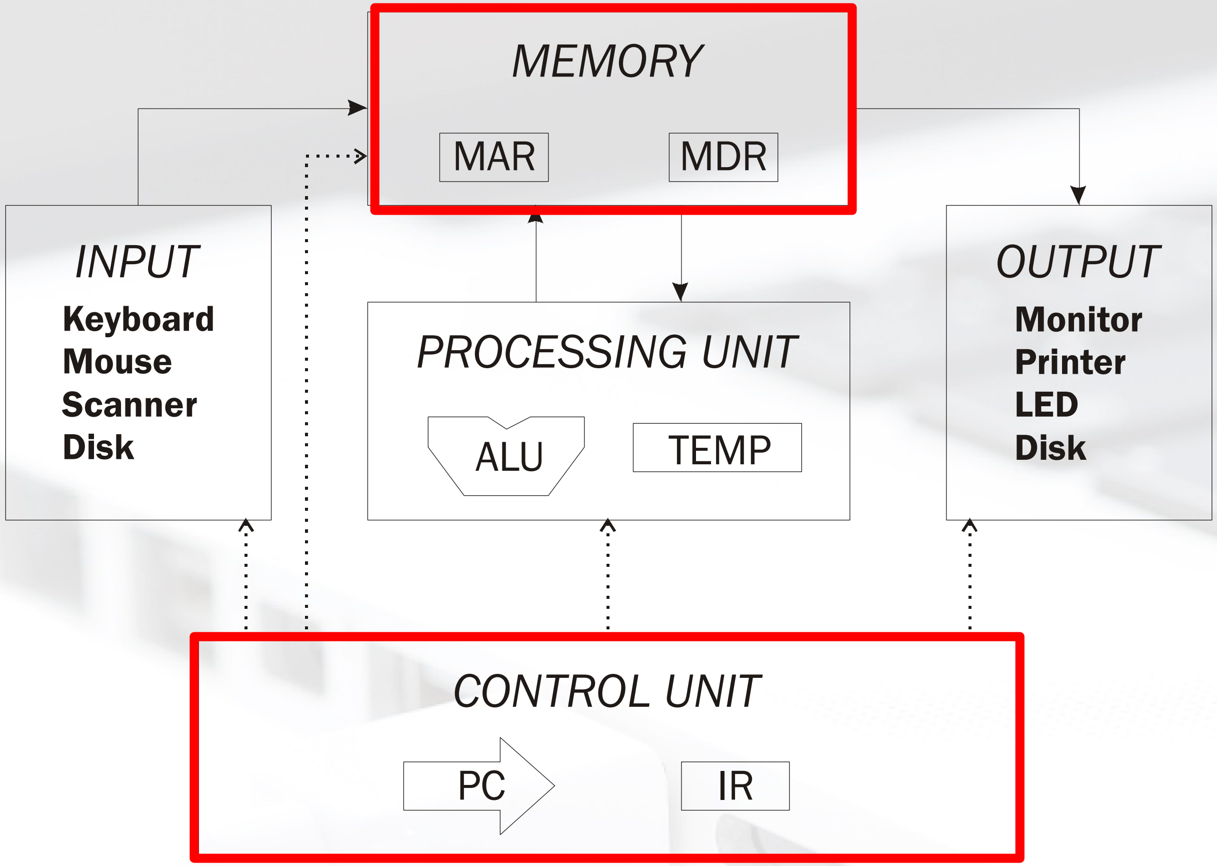
- What if the memory you want to access is far away?
- LD/ST won't work (PC-relative)
- LDR/STR won't work alone (need to get address in register)

Solution: LDI/STI

- Place *address* of far away value nearby
- Load address, then load/store from that



Memory Model for Function Calls



Problem

- **How do we allocate memory during the execution of a program written in C?**
  - Programs need memory for code and data such as instructions, global and local variables, etc.
  - Modern programming practices encourage many (reusable) functions, callable from anywhere.
  - Some memory can be statically allocated, since the size and type is known at compile time.
  - Some memory must be allocated dynamically, size and type is unknown at compile time.

Motivation

■ **Why is memory allocation important? Why not just use a memory manager?**

- **Allocation affects the performance and memory usage of every C, C++, Java program.**
- **Current systems do not have enough registers to store everything that is required.**
- **Memory management is too slow and cumbersome to solve the problem.**
- **Static allocation of memory resources is too inflexible and inefficient, as we will see.**

Goals

- **What do we care about?**
  - Fast program execution
  - Efficient memory usage
  - Avoid memory fragmentation
  - Maintain data locality
  - Allow recursive calls
  - Support parallel execution
  - Minimize resource allocation
  - Memory should never be allocated for functions that are not executed.

Scope: Local vs. Global

- **A variable's declaration assists the compiler in managing the storage of that variable.**
- **In C, a variable's declaration conveys three pieces of information to the compiler:**
  - ***the variable's identifier and its type***
    - — The first two of these, identifier and type, the C compiler gets explicitly from the variable's declaration.
  - ***the variable's scope———***
    - — The third piece, scope, the compiler infers from the position of the declaration within the code. The scope of a variable is the region of the program in which the variable is "alive" and accessible.
    - — The good news is that in C, there are only two basic types of scope for a variable. Either the variable is ***global*** to the entire program, or it is ***local***, **or private**, to a particular block of code.

A C program that demonstrates nested scope.

```
#include <stdio.h>

int globalVar = 2; // This variable is a global variable

int main(void)
{
    int localVar = 3; // This variable is local to main

    printf("globalVar = %d, localVar = %d\n", globalVar, localVar);

    // Creating a new sub-block within main
    {
       int localVar = 4; // This local to the sub-block within main

       printf("globalVar = %d, localVar = %d\n", globalVar, localVar);
    }

    printf("globalVar = %d, localVar = %d\n", globalVar, localVar);
}
```

C allows this: as long as the different variables sharing the same name are declared in separate blocks.

If we compile and execute this code, the output generated looks as follows:

```
globalVar = 2, localVar = 3
globalVar = 2, localVar = 4
globalVar = 2, localVar = 3
```

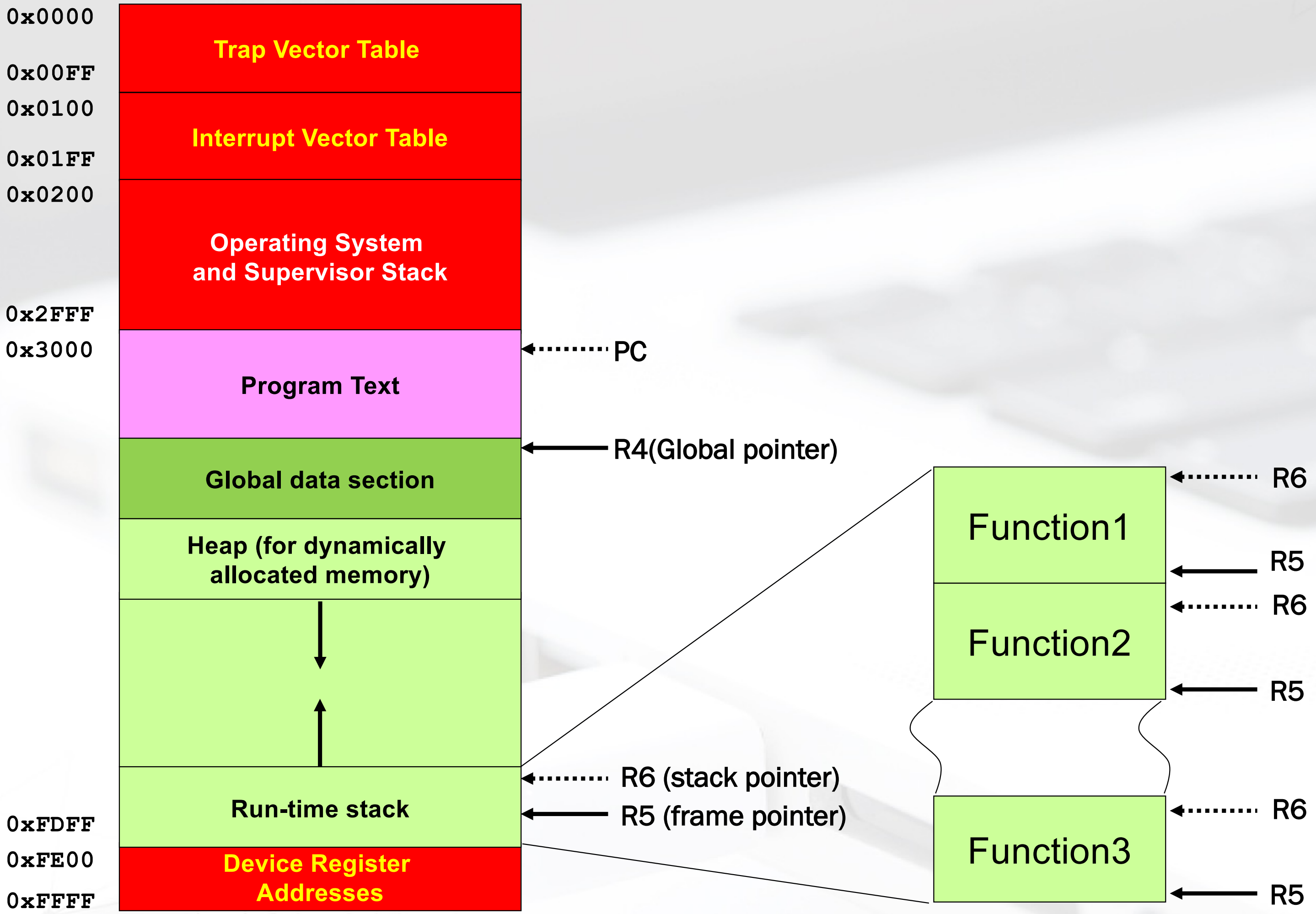
Initialization of Variables

```
double width;
double pType = 9.44;
double mass = 6.34E2;
double verySmallAmount = 9.1094E-31;
double veryLargeAmount = 7.334553E102;
int average = 12;
int windChillIndex = -21;
int unknownValue;
int mysteryAmount;
bool flag = false;
char car = 'A';     // single quotes specify a single ASCII character
char number = '4'; // single quotes specify a single ASCII characte
```

What initial value will a variable have if it has no initializer? In C, by default,

- Local variables start with an undefined value. That is, local variables have garbage values in them, unless we explicitly initialize them in our code. It is standard coding practice to explicitly initialize local variables within their declarations.
- Global variables, in contrast, are initialized to 0.

Memory Model in the LC-3



Allocating Space for Variables

- **There are two regions of memory in which declared variables in C are allocated storage:**
  - ***the global data section:*** **Variables that are global are allocated storage in the global data section.**
  - ***the run-time stack:*** **Local variables are allocated storage on the run-time stack.**

A C program that performs a simple network rate calculation

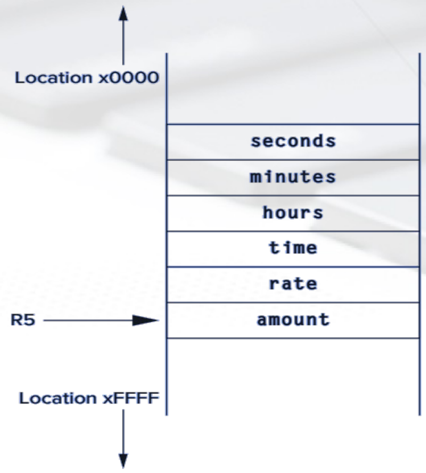
```c
#include <stdio.h>
int main(void)
{
    int amount;             // The number of bytes to be transferred
    int rate;               // The average network transfer rate
    int time;               // The time, in seconds, for the transfer
    int hours;              // The number of hours for the transfer
    int minutes;            // The number of mins for the transfer
    int seconds;            // The number of secs for the transfer

// Get input: number of bytes and network transfer rate
    printf("How many bytes of data to be transferred? ");
    scanf("%d", &amount);
    printf("What is the transfer rate (in bytes/sec)? ");
    scanf("%d", &rate);

// Calculate total time in seconds
    time = amount / rate;

// Convert time into hours, minutes, seconds
    hours = time / 3600;                  // 3600 seconds in an hour
    minutes = (time % 3600) / 60;         // 60 seconds in a minute
    seconds = ((time % 3600) % 60);       // remainder is seconds

// Output results
    printf("Time : %dh %dm %ds\n", hours, minutes, seconds);
}
```

Identifier	Type	Location (as an offset)	Scope	Other info...
amount	int	0	main	...
hours	int	−3	main	...
minutes	int	−4	main	...
rate	int	−1	main	...
seconds	int	−5	main	...
time	int	−2	main	...

The compiler's symbol table when it compiles the code

■ The stack frame from function main of the code

- This function has five local variables.
- R5 is the frame pointer and points to the first local variable.



Outline

Stack: An Abstract Data Type

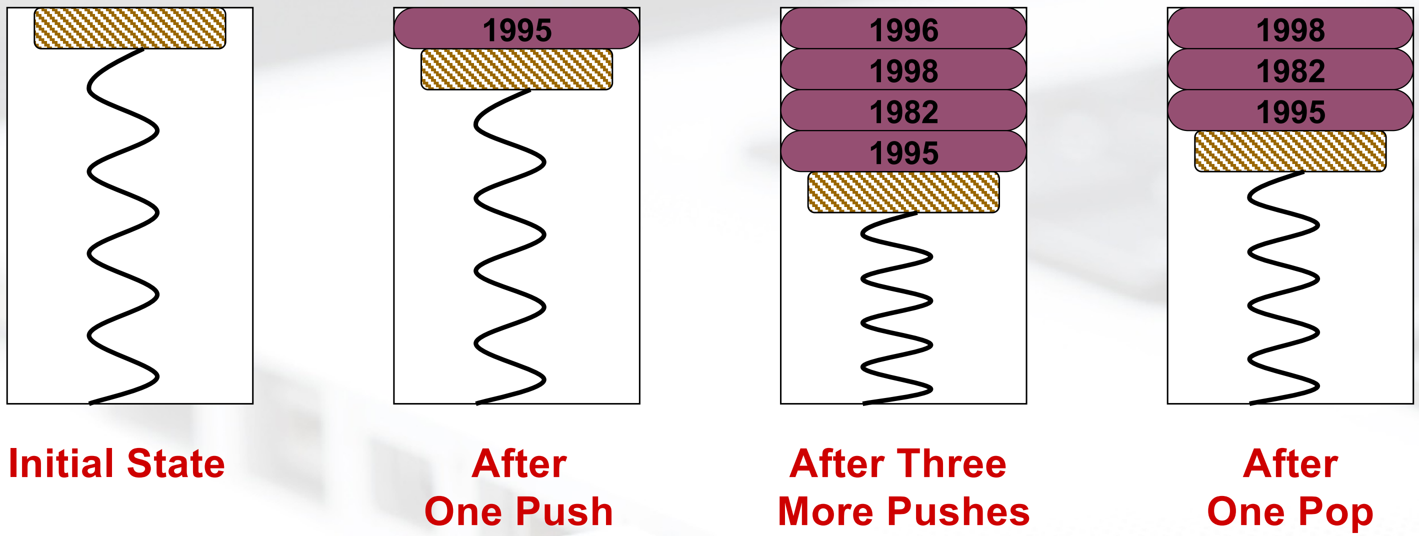
- **An important abstraction that you will encounter in many applications.**
- **The fundamental model for execution of C, Java, Fortran, and many other languages.**
- **We will describe two uses of the stack:**
  - **Evaluating arithmetic expressions**
    - — Store intermediate results on stack instead of in registers
  - **Function calls**
    - — Store parameters, return values, return address, dynamic link
    - — Interrupt-Driven I/O
    - — Store processor state for currently executing program

Stack Data Structure

- A LIFO (last-in first-out) storage structure
  - The first thing you put in is the last thing you take out
  - The last thing you put in is the first thing you take out
  - This means of access is what defines a stack, not the specific implementation.
- Two main operations
  - PUSH: add an item to the stack
  - POP: remove an item from the stack
- Error conditions:
  - Underflow (try to pop from empty stack)
  - Overflow (try to push onto full stack)
- A register (eg. R6) holds address of top of stack (TOS)

A Physical Stack

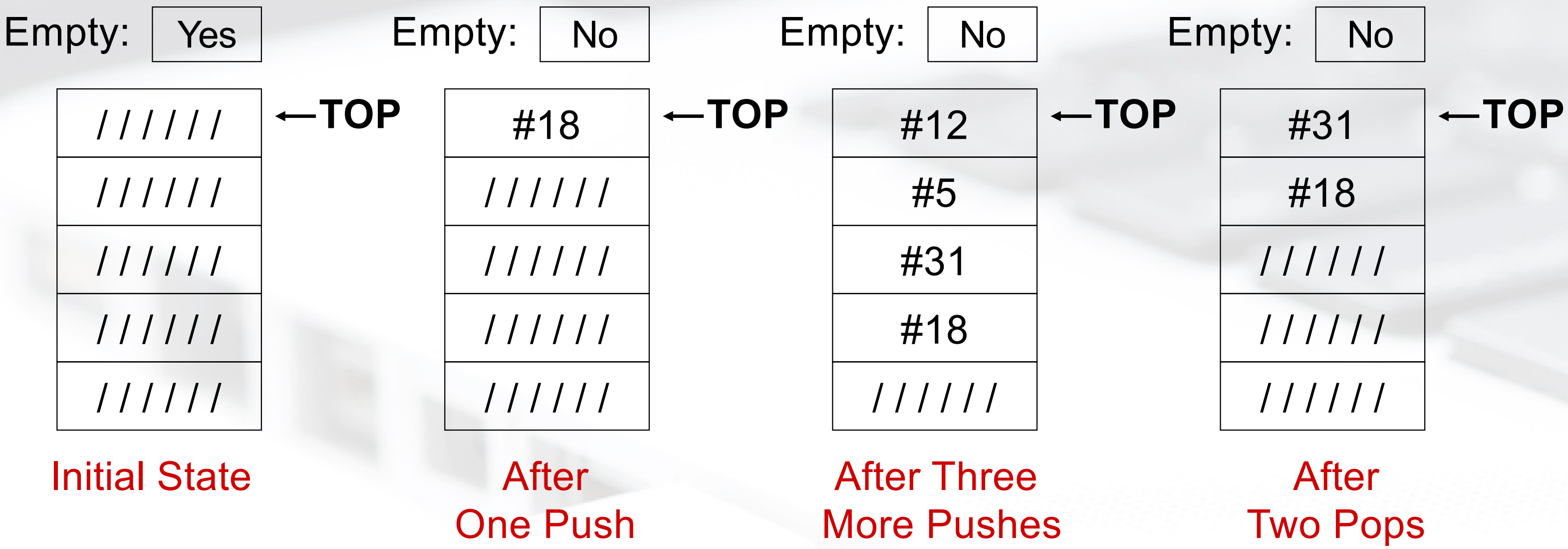
- Coin holder



Last quarter in is the first quarter out (LIFO)

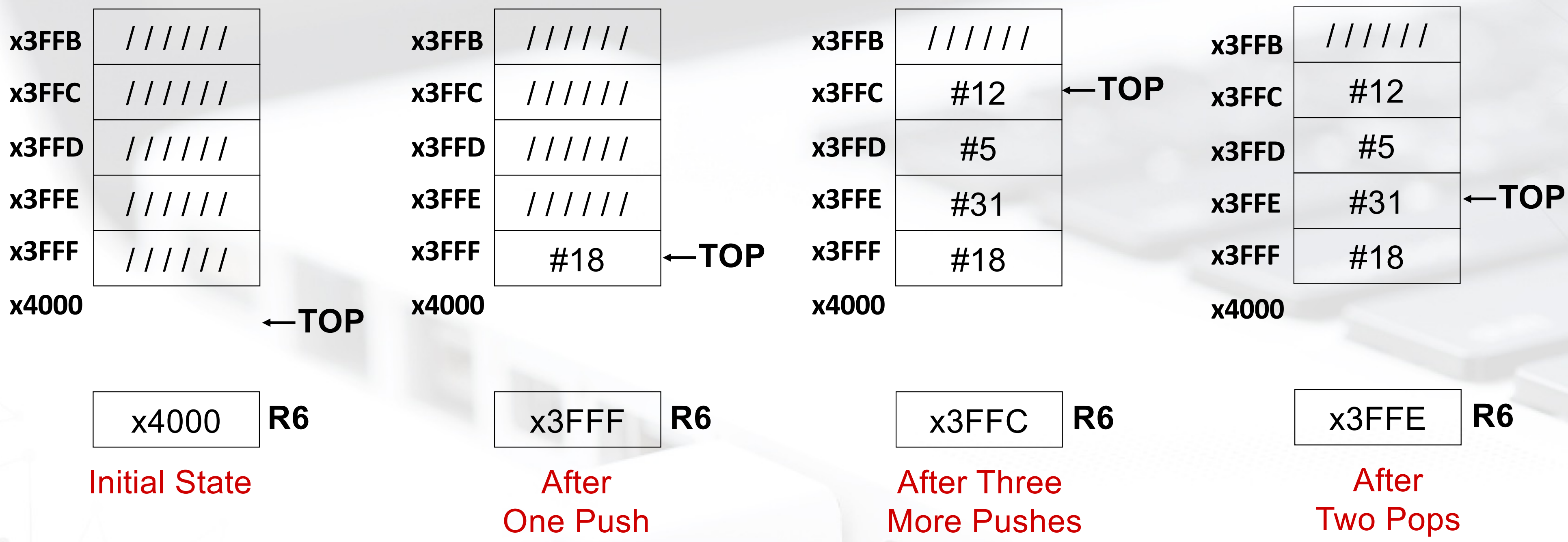
A Hardware Stack Implementation

- Data items move between registers



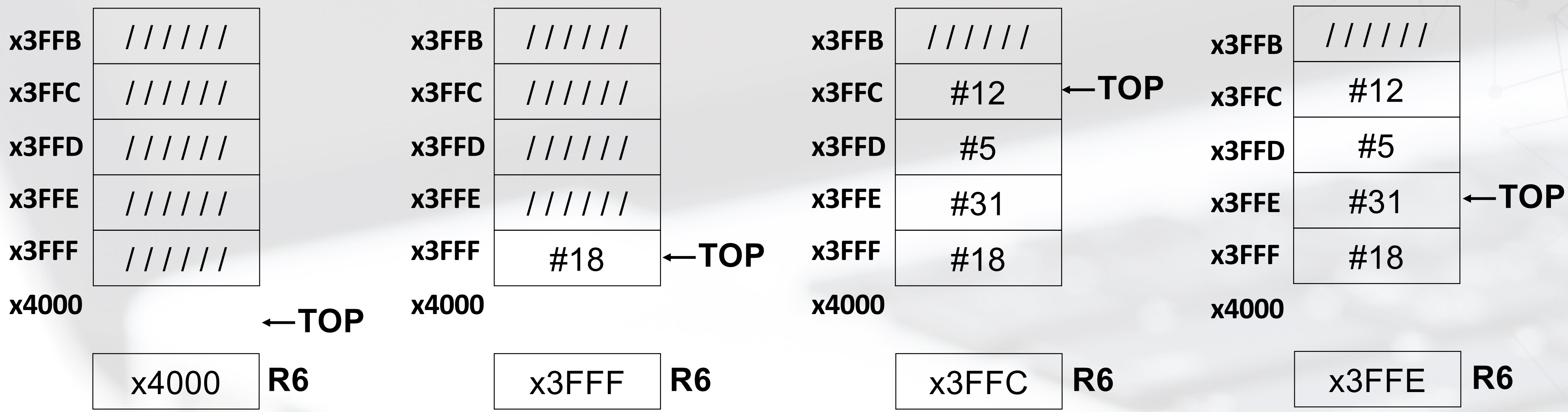
A Software Stack Implementation

- Data items don't move in memory, just our idea about where TOP of the stack is



By convention, R6 holds the Top of Stack (TOS) Pointer (SP)

Basic Push and Pop Code




```
PUSH
        ADD R6, R6, #-1           ; increment stack ptr
        STR R0, R6, #0            ; store data(R0) to TOS
POP
        LDR R0, R6, #0            ; load data(R0) from TOS
        ADD R6, R6, #1            ; decrement stack ptr
```

- **Note:** Stacks can grow in either direction (toward higher address or toward lower addresses)

Pop with Underflow Detection

- If we try to pop too many items off the stack, an **underflow** condition occurs.
  - Check for underflow by checking TOS before removing data.
  - Return status code in R5 (0 for success, 1 for underflow)

```
POP    LD   R1, EMPTY
       ADD R2, R6, R1        ; Compare stack pointer
       BRz UNDER             ; with x3FFF
       LDR R0, R6, #0         ; The actual ‘pop’
       ADD R6, R6, #1        ; Adjust stack pointer
       AND R5, R5, #0        ; Success: return R5 = 0
       RET
UNDER AND R5, R5, #0        ; Underflow: return R5 = 1
       ADD R5, R5, #1
       RET
EMPTY .FILL xC000           ; EMPTY = -x4000
```

Push with Overflow Detection

- If we try to push too many items onto the stack, an **overflow** condition occurs.
  - Check for underflow by checking TOS before adding data.
  - Return status code in R5 (0 for success, 1 for overflow)

```
PUSH   LD  R1, FULL
       ADD R2, R6, R1        ; Compare stack pointer
       BRz OVER              ; with x4004
       ADD R6, R6, #-1       ; Adjust stack pointer
       STR R0, R6, #0        ; The actual 'push'
       AND R5, R5, #0        ; Success: return R5 = 0
       RET
OVER   AND R5, R5, #0
       ADD R5, R5, #1        ; Overflow: return R5 = 1
       RET
FULL   .FILL xC005           ; FULL = -x3FFB
```

The final code for PUSH & POP in LC-3 - 1

```
POP     ST R2,Save2          ; save, needed by POP
        ST R1,Save1          ; save, needed by POP
        LD  R1, EMPTY        ; EMPTY contains -x3FFF
        ADD R2, R6, R1       ; Compare stack pointer with x3FFF
        BRz Fail_exit        ; Branch if stack empty
        LDR R0, R6, #0       ; The actual 'pop'
        ADD R6, R6, #1       ; Adjust stack pointer
        RET
EMPTY   .FILL xC000          ; EMPTY = -x4000
PUSH    ST R2,Save2          ; save, needed by PUSH
        ST R1,Save1          ; save, needed by PUSH
        LD  R1, FULL
        ADD R2, R6, R1       ; Compare stack pointer
        BRz Fail_exit               ; with x4004
        ADD R6, R6, #-1      ; Adjust stack pointer
        STR R0, R6, #0       ; The actual 'push'
        RET
FULL    .FILL xC005          ; FULL = -x3FFB
```

The final code for PUSH & POP in LC-3 - 1

```
POP     ST R2,Save2          ; save, needed by POP
        ST R1,Save1          ; save, needed by POP
        LD  R1, EMPTY        ; EMPTY contains -x3FFF
        ADD R2, R6, R1       ; Compare stack pointer with x3FFF
        BRz Fail_exit        ; Branch if stack empty
        LDR R0, R6, #0       ; The actual 'pop'
        ADD R6, R6, #1       ; Adjust stack pointer
        BRnzp Success_exit
EMPTY   .FILL xC000          ; EMPTY = -x4000
PUSH    ST R2,Save2          ; save, needed by PUSH
        ST R1,Save1          ; save, needed by PUSH
        LD  R1, FULL
        ADD R2, R6, R1       ; Compare stack pointer
        BRz Fail_exit               ; with x4004
        ADD R6, R6, #-1      ; Adjust stack pointer
        STR R0, R6, #0       ; The actual 'push'
        BRnzp Success_exit
FULL    .FILL xC005          ; FULL = -x3FFB
```

The final code for PUSH & POP in LC-3 - 2

```
Save1           .FILL x0000
Save2           .FILL x0000
Success_exit    LD   R1, Save1   ;Restore reg values
                LD   R2, Save2   ;
                AND R5, R5, #0  ; Success: return R5 = 0
                RET
;
Fail_exit       LD   R1, Save1   ;Restore reg values
                LD   R2, Save2
                AND R5, R5, #0
                ADD R5, R5, #1  ; Overflow: return R5 = 1
                RET
```

Arithmetic Using a Stack

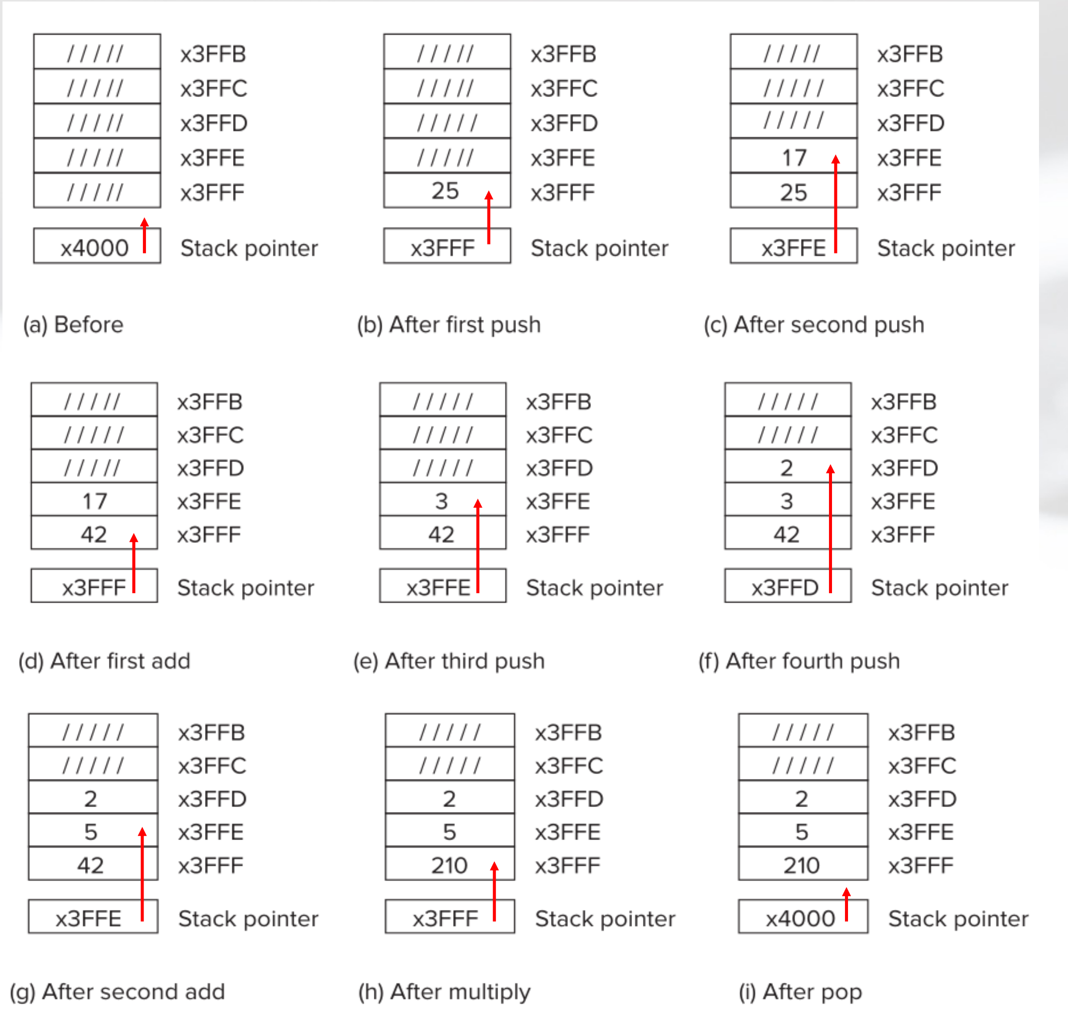
- Instead of registers, some ISA's use a stack for source and destination operations: a **zero-address** machine.
  - Example: ADD instruction pops two numbers from the stack, adds them, and pushes the result to the stack.

    ADD vs. ADD R0,R1,R2

- Evaluating (A+B)•(C+D) using a stack:

```
push A
push B
ADD
push C
push D
ADD
MULTIPLY
pop result
```

(25+17) x (3+2)



/////
x3FFB
x3FFC
x3FFD
x3FFE
x3FFF
17
25
42
3
2
5
210
x4000
x3FFF
x3FFE
x3FFD
Stack pointer
(a) Before
(b) After first push
(c) After second push
(d) After first add
(e) After third push
(f) After fourth push
(g) After second add
(h) After multiply
(i) After pop