Chapter 7

What are pseudo-ops

Directives or psuedo-ops give information to the

assembler.

- **Not executed by the program**
- **All directives start with a period "."**

Directive Description

.ORIG

Where to start in placing things in memory

.FILL

Declare a memory location (variable)

.BLKW

Declare a group of memory locations (array)

.STRINGZ

Declare a group of characters in memory (string)

.END

Tells assembly where your program source ends

Two Passes in the Assembler

First Pass

- **scan program file**
- **Find all labels and calculate the corresponding addresses - the symbol table**

Find the .ORIG statement,

which tells us the address of the first instruction.

- **Initialize Location Counter (LC), which keeps track of the**

current instruction.

2. For each non-empty line in the program:

a) If line contains a label, add label and LC to symbol table.

b) Increment LC.

– NOTE: If statement is .BLKW or .STRINGZ,

increment LC by the number of words allocated.

3. Stop when .END statement is reached.

NOTE: A line with only a comment is considered an empty line.

Second Pass

- **Convert instructions to machine language, using information from symbol table**

For each executable assembly language statement,

generate the corresponding machine language instruction.

– If operand is a label,

look up the address from the symbol table.

• Potential problems:

– Improper number or type of arguments

• ex: NOT R1,#7

ADD R1,R2

– Immediate argument too large

• ex: ADD R1,R2,#1023

– Address (associated with label) more than 256 from instruction

• can‟t use PC-relative addressing mode

Symbol table

In symbol table we keep the track of every statement
i.e after .orig 0x300 we know whether the

Opcode is valid or not. But if there is some label than that

Label is also added to the Symbol table and and LC is incremented.

Symbol table for Simple Multiply Program

Hex Address	Label / Instruction	Destination	Source1(Reg)	Source2(Reg)
	.ORIG x3000			
x3000	LD	R2	zero	
X3001	LD	R0	M0	
X3001	LD	R1	M1	
X3001	Loop BrZ Done			
X3001	Add	R2	R2	R0
X3001	Add	R1	R1	#-1
X3001	Br Loop			
X3001	Done St	R2	Result	
X3001	Halt			
X3001	Result .Fill 0x000			
X3001	Zero .Fill 0x000			
X3001	M0 .Fill 0x007			
X3001	M1 .Fill 0x003			
	.END			

LD, LDR, LEA, LDI

```
LEA R0, A      -- R0 has the address of A
LDI R2, C      -- R2 has value of which address C has
LDR R3, R0, 2  -- R3 has the value of C
               -- Because R0 has the address of A + 2 positions = C
```

CHAPTER 9

Subroutines

Blocks can be encoded as subroutines

A subroutine is a program fragment that:

– lives in user space

– performs a well-defined task

– is invoked (called) by a user program

– returns control to the calling program when finished

Parameters, Arguments and Returned Values

Arguments

– A value passed in to a subroutine (or trap) is called an argument.
– This is a value needed by the subroutine to do its job.
– Examples:
• In 2sComp routine, R0 is the number to be negated
• In PUTS routine, R0 is address of string to be printed.

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:
• In 2sComp routine, negated value is returned in R0.
• In GETC service routine, character read from the keyboard
is returned in R0.

Call by Reference and Call by Value

Call By reference means to pass the address of the variable to the function and than the function alters the value stored in that address.

Call by Value means to pass the value of the variable to the function and function process that value and returns the result.

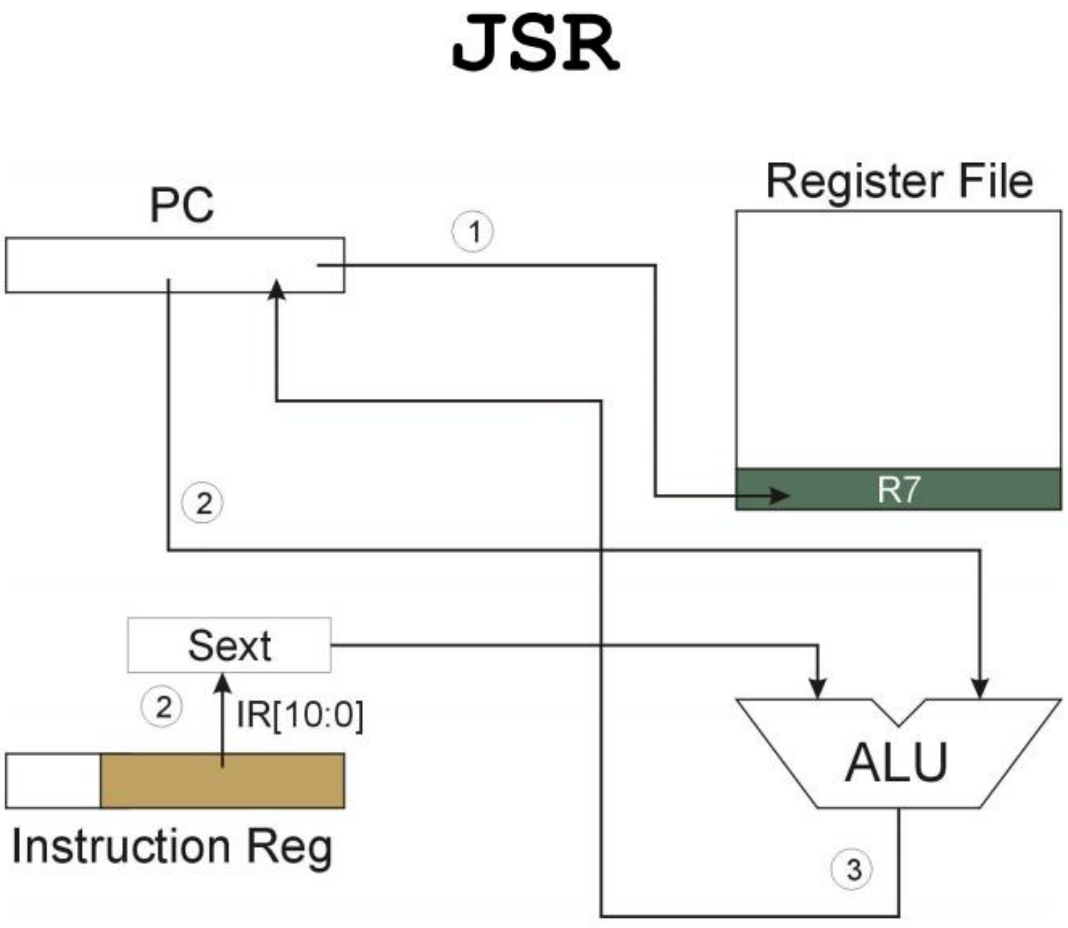
Calling a Subroutine in LC3

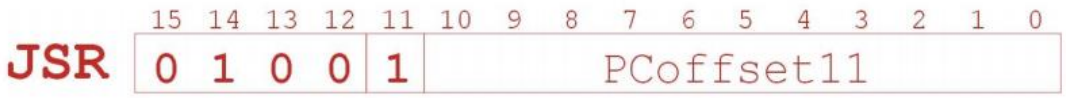
```
• 2sComp    NOT   R0, R0        ; flip bits
            ADD   R0, R0, #1    ; add one
            RET                 ; return to caller
```

• *To call from a program (within 1024 instructions):*

```
•; need to compute R4 = R1 - R3
            ADD   R0, R3, #0    ; copy R3 to R0
            JSR   2sComp        ; negate
            ADD   R4, R1, R0    ; add to R1
```

JSR

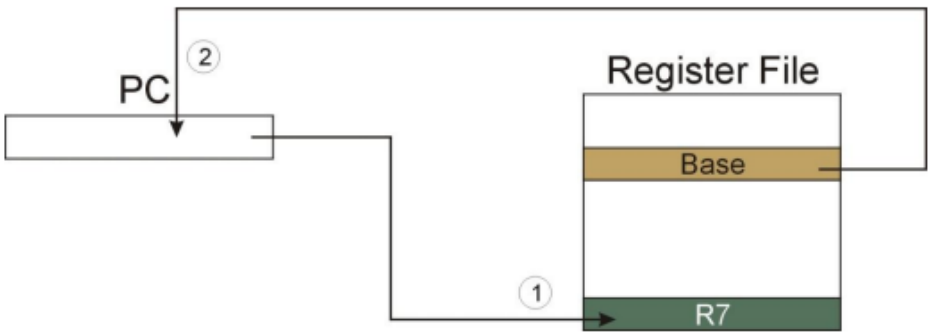


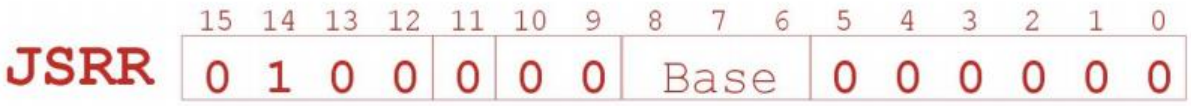


- Jumps to a location (like a branch but unconditional), and saves current PC (addr of next instruction) in R7.
  - saving the return address is called "linking"
  - target address is PC-relative **(PC + Sext(IR[10:0]))**
  - bit 11 specifies addressing mode
    - if =1, PC-relative: target address = **PC + Sext(IR[10:0])**
    - if =0, register: target address = contents of Register[**IR[8:6]**] **(JSRR)**

JSRR

JSRR





- Just like JSR, except Register addressing mode.
  - target address is Base Register
  - bit 11 specifies addressing mode

- What important feature does JSRR provide that JSR does not?

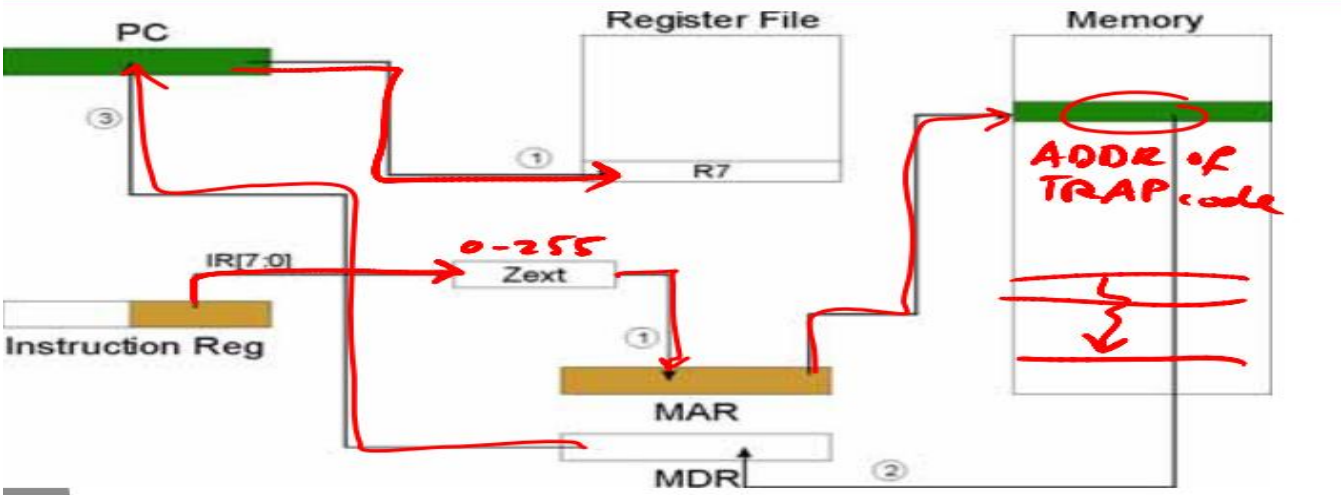
Saving Without Stack

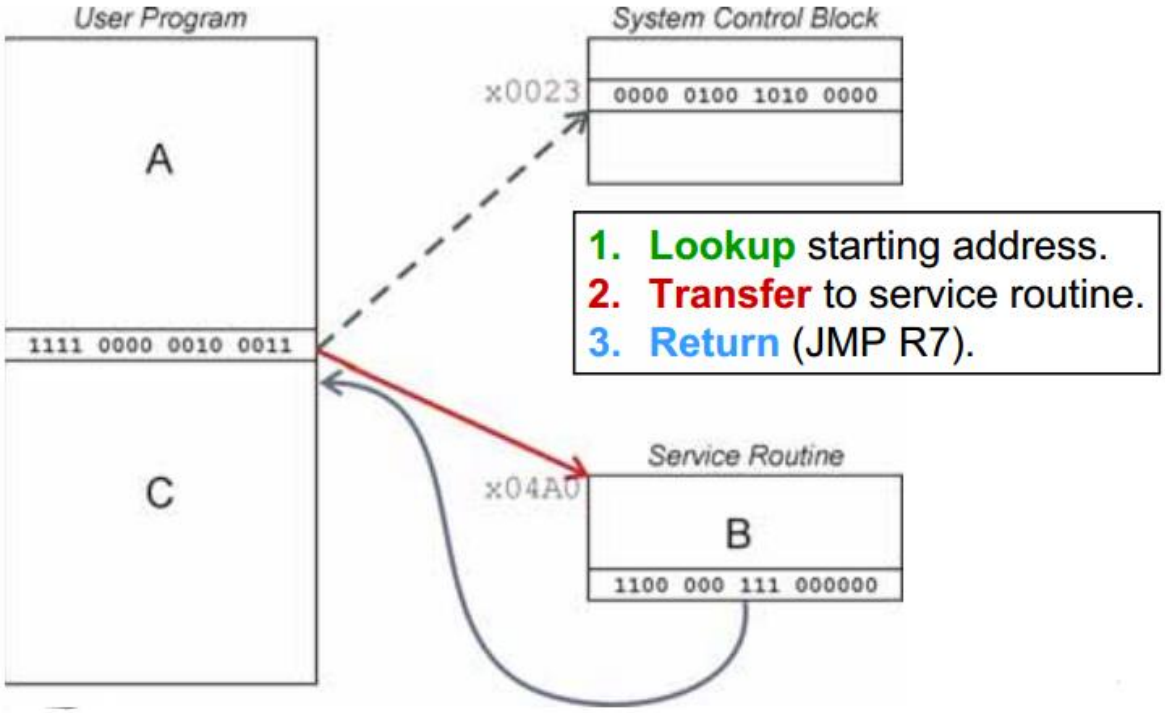
In order to save without stack we will make an array in which we dump our all registers values as well as we can pass values to the functions through it .

Trap

TRAP instruction

- **Used by program to transfer control to operating system**
- **8-bit trap vector names one of the 256 service routines**




User Program
A
1111 0000 0010 0011
C
System Control Block
x0023
0000 0100 1010 0000
1. Lookup starting address.
2. Transfer to service routine.
3. Return (JMP R7).
Service Routine
x04A0
B
1100 000 111 000000