LC-3, addressing modes

See P&P Appendices A and C: LC-3 ISA, TRAPS, Devices, Interrupts, Exceptions.

1. DATA IN REGISTER (RegFile[i] , IR , PC)

ADD	R2	R3		R1
0001	010	011	000	001
OP	DR	SR1		SR2

R2 <== R3 + R1 (register, register, register)

ADDi	R2	R3		#2
0001	010	011	1	00010
OP	DR	SR1		immed5

R2 <== R3 + IR[4:0] (register, register, immediate)

LEA	R2	myLoc
1110	010	0 0000 0001
		PCoffset9

R2 <== PC + IR[8:0] (register, PC, immediate)
(assembler computes **PCoffset9** from label)

2. MEMORY ADDRESS IN REGISTER (Regfile[i] , PC , IR)

LDR	R2	R3	#2
0110	010	011	000010
		BaseR	offset6

MAR <== R3 + IR[5:0]
R2 <== MDR
(register, base-offset)

LD	R2	myVar
1010	010	0 0000 0001
		PCoffset9

MAR <== PC + IR[8:0]
R2 <== MDR
(register, PC-relative)
(assembler calculates offset from label)

BR	z	myLoc
0000	010	0 0000 0001
	cc	PCoffset9

PC <== PC + IR[8:0] (PC-relative) (if Condition Code Z=1)

jmp		R2	
1100	000	010	000000

PC <== R2 (register)
(if **R7**, then aka "**RET**")

jsr		myFunc
0100	1	0000 0000000
		PCoffset11

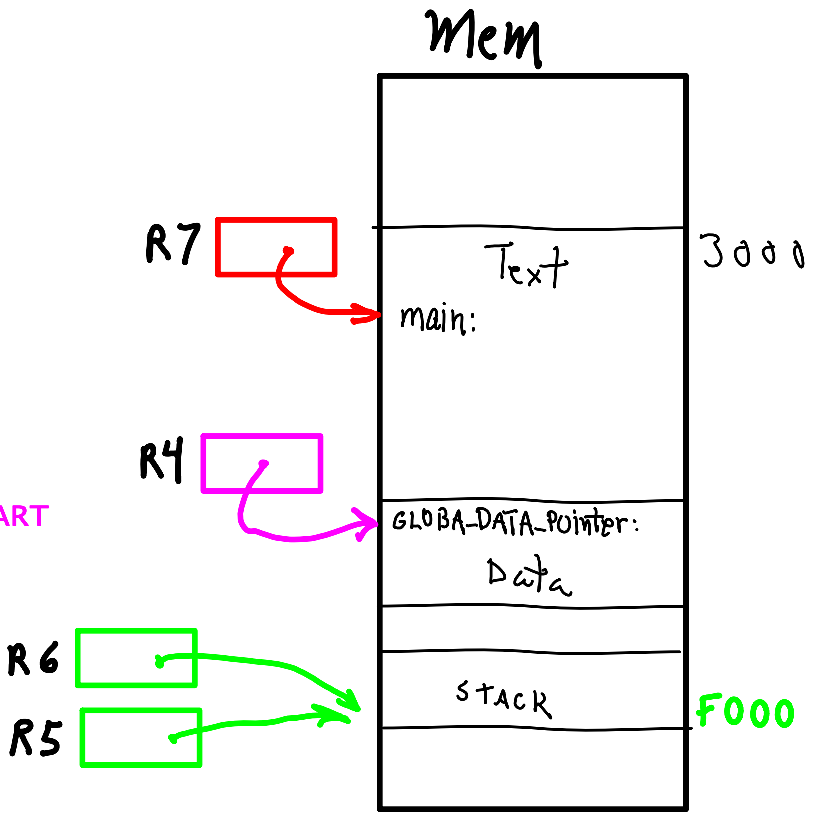
R7 <== PC
PC <== PC + IR[10:0]
(PC-relative)
(assembler calculates offset from label)

jsrr			R2	
0100	0	00	010	000000

R7 <== PC
PC <== R2
(register)

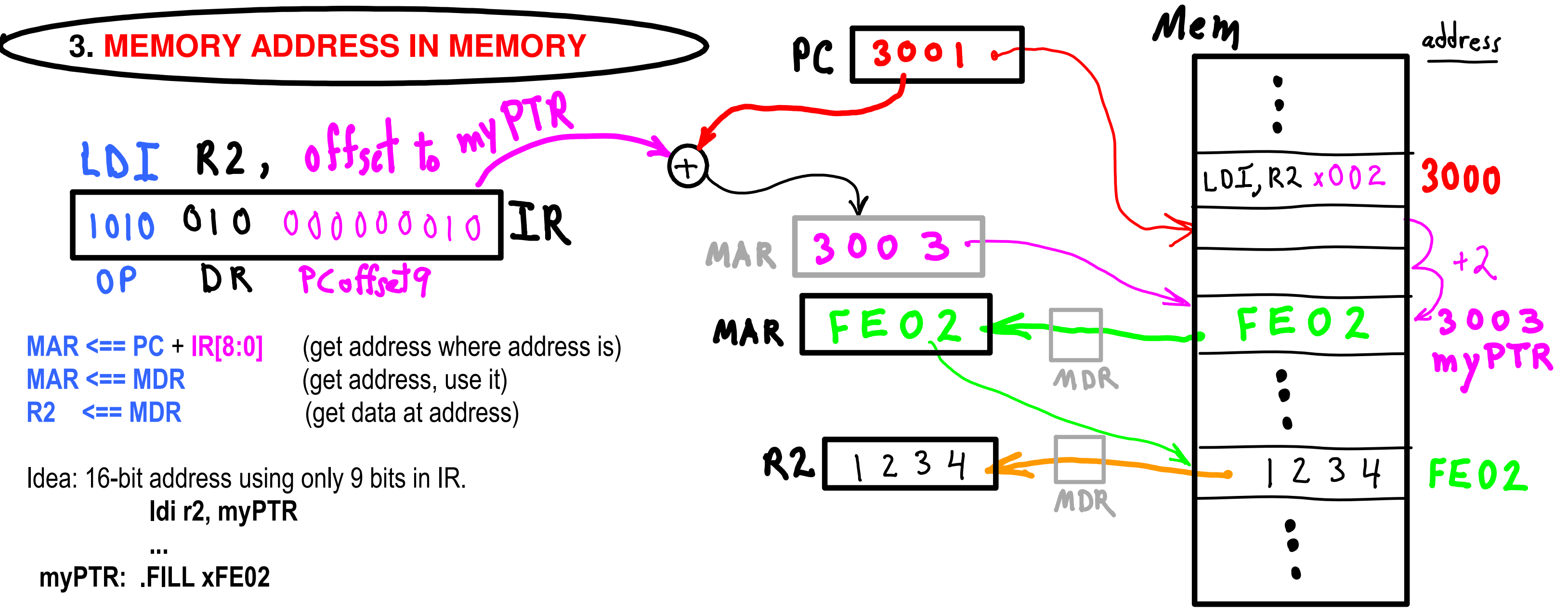
Typical usage (C compiler generated)

```
.orig x3000
INIT_CODE
LD   R6, STACK_POINTER
LD   R5, STACK_POINTER
LD   R4, GLOBAL_DATA_POINTER
LD   R7, GLOBAL_MAIN_POINTER
jsrr R7
HALT

STACK_POINTER       .FILL xF000
GLOBAL_DATA_POINTER .FILL GLOBAL_DATA_START
GLOBAL_MAIN_POINTER .FILL main

   ...

main:

   ...

func:

   ...

GLOBAL_DATA_POINTER:
       .FILL x1234
       .FILL x3000
       .FILL x0002
       .FILL func
```



```
;------ get data:

ADD R0, R4, #2
LDR R2, R0, #0

;----- jump to func's location:

ADD R0, R4, #3
JSRR R0
```

3. MEMORY ADDRESS IN MEMORY



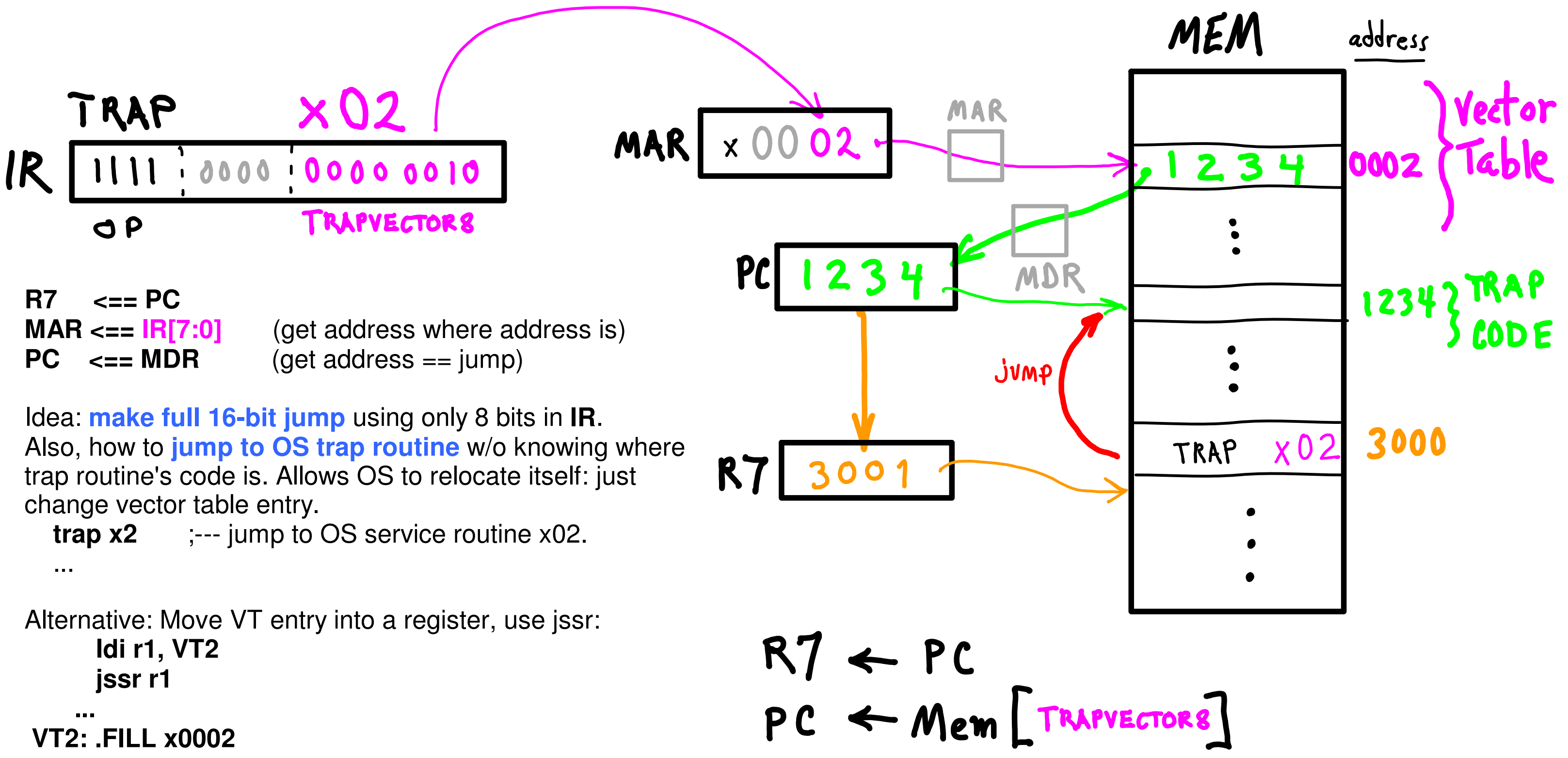
MAR <== PC + IR[8:0]   (get address where address is)
MAR <== MDR   (get address, use it)
R2 <== MDR   (get data at address)

Idea: 16-bit address using only 9 bits in IR.

```
        ldi r2, myPTR
        ...
myPTR:  .FILL xFE02
```

Alternative: Move myPTR into a register, use base-offset mode:

```
        ld r1, myPTR
        ldr r2, r1, 0
        ...
myPTR:  .FILL xFE02
```



R7 <== PC
MAR <== IR[7:0]   (get address where address is)
PC <== MDR   (get address == jump)

Idea: make full 16-bit jump using only 8 bits in **IR**.
Also, how to jump to OS trap routine w/o knowing where trap routine's code is. Allows OS to relocate itself: just change vector table entry.

```
    trap x2       ;--- jump to OS service routine x02.
    ...
```

Alternative: Move VT entry into a register, use jssr:

```
        ldi r1, VT2
        jssr r1
      ...
 VT2: .FILL x0002
```

Aside: Using what we had above to eliminate ldi, we could eliminate both LDI and TRAP instructions from the LC3's ISA: we would have two unused opcodes to play with.

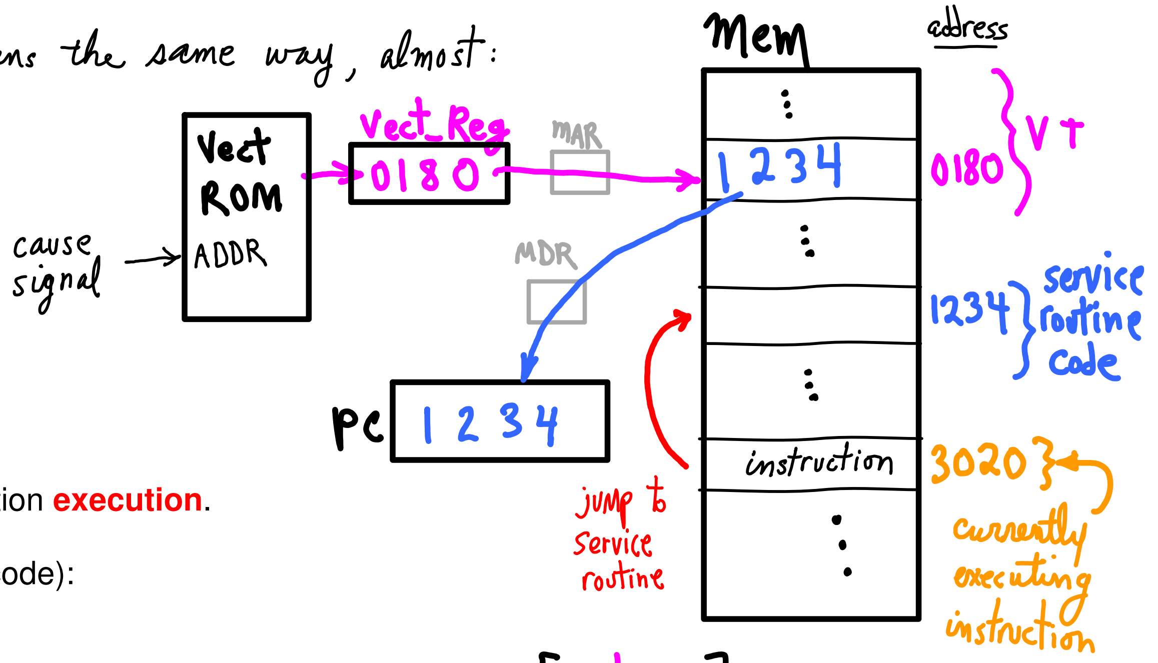
Exceptions
Interrupts

Yet another **address-in-memory** mechanism.
Just like **TRAP**, but not an instruction.

Something goes wrong: jump to OS routine   (exception)
I/O device sends a signal: jump to OS routine   (interrupt)

The jump happens the same way, almost:

MAR <== VECT_REG
PC <== MDR



EXCEPTIONS
---- **detected during** instruction **execution**.
Eg., "**illegal opcode**"
detected in **state-32** (decode):
VECT_REG <== **x0100**.

INTERRUPTS
---- **generated by device** interrupt logic
---- detected in **state-18** (fetch)
Eg., a **keyboard event:**
VECT_REG <== **x0180**

PC ← Mem [vect_reg]

(How To jump back?)

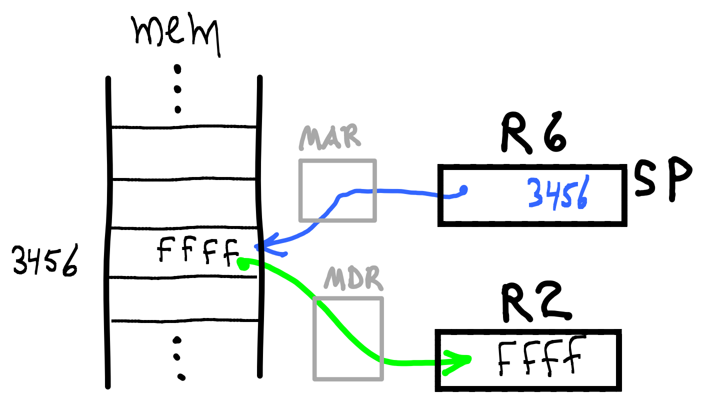
LC3 Controller States,
13: opcode exception
44: privilege exception
49: interrupt

But,
more needs to be done: Save currently executing code's state!

Not the same as TRAP.
For **TRAP**, currently executing code,
---- **knows** a jump is occurring;
---- **can SAVE** its own STATE beforehand;
---- **knows** its **CC state could change**: does not **BR** immediately after **TRAP**.

Before we explain saving state, let's see Stack Addressing.

STACK OPERATIONS



I. Access top item in stack.

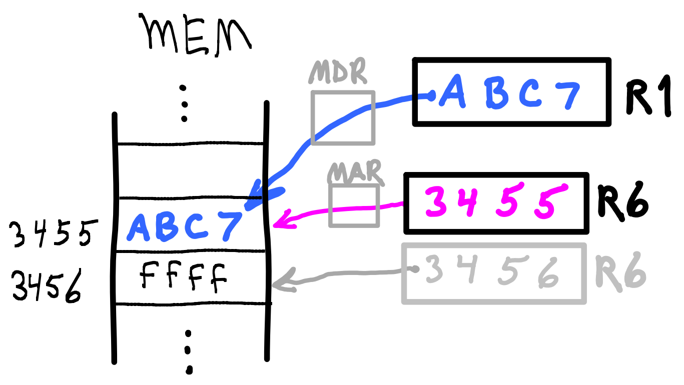
R2 ← mem[R6]

LDR R2, R6, #0

MAR <== R6
R2 <== MDR

Stack Pointer (**SP**) is **R6**

II. Put new item on top of stack: PUSH

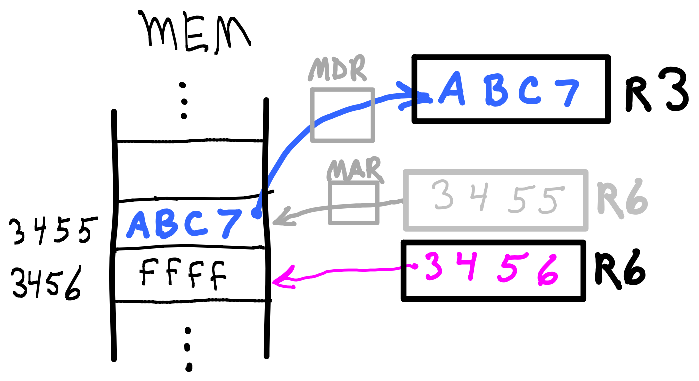


PUSH R1

ADD R6, R6, #-1 — 1. R6--
STR R1, R6, #0 — 2. MEM [R6] ← R1

R6 <== R6 - 1
MAR <== R6 + IR[5:0]
MDR <== R1

III. Remove item from top of stack: POP



PoP R3

LDR R3, R6, #0 — 1. R3 ← MEM[R6]
ADD R6, R6, #1 — 2. R6++

MAR <== R6 + IR[5:0]
R3 <== MDR
R6 <== R6 - 1

$\boxed{\text{Saving State}}$

We need to restart currently executing code in its Same execution state (PSR, PC, SP, RegFile)

When an **exception/interrupt occurs**

---- **PSR altered** immediately, before the next instruction is fetched.

---- **PC altered**, i.e., a jump.

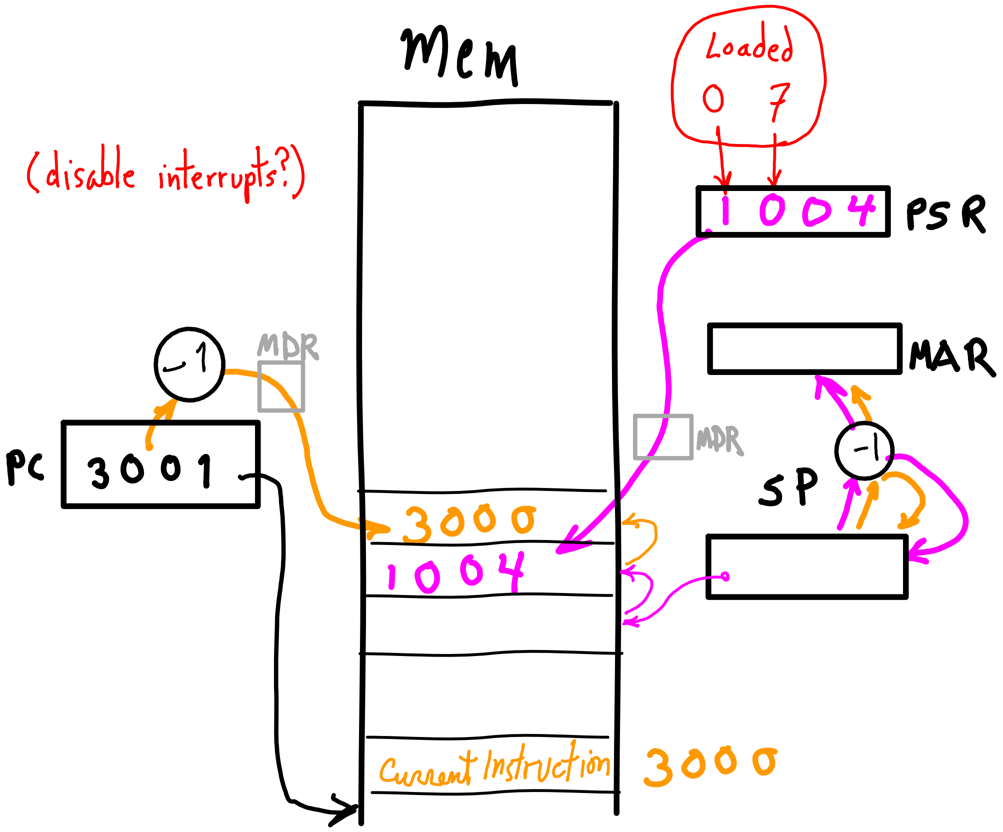
PC could go to **R7**, but what about **nested execeptions/interrupts**?

---- **SP (R6) altered** to push state, it needs to be saved.

---- **Regs** can be **saved by service routine** code.

===> **Hardware**, not instruction execution, **must save state**!

```
49 INT
   MDR         <= PSR          1.
   PSR[10:8] <= IntPriority   1.   (disable interrupts?)
   PSR[15]    <= 0             1.
 <PSR[15] == 1?> save SP

37, 41 push PSR
   SP    <= SP-1  }
   MAR <= SP-1   } 1.
   Mem <= MDR    }
43, 47, 48 push PC
   MDR <= PC-1   }
   SP    <= SP-1  }
   MAR <= SP-1   } 2.
   Mem <= MDR    }
50, 52, 54 jump
   MAR <= Vector
   MDR <= Mem
   PC    <= MDR
```



ALSO, if PSR[15] == 1, must save SP, and switch to SUPER's STACK. See R6 save/restore hardware near ALU.

When **exception/interrupt** routine **COMPLETES**

--- **RESTORE Regs**, done in service routine

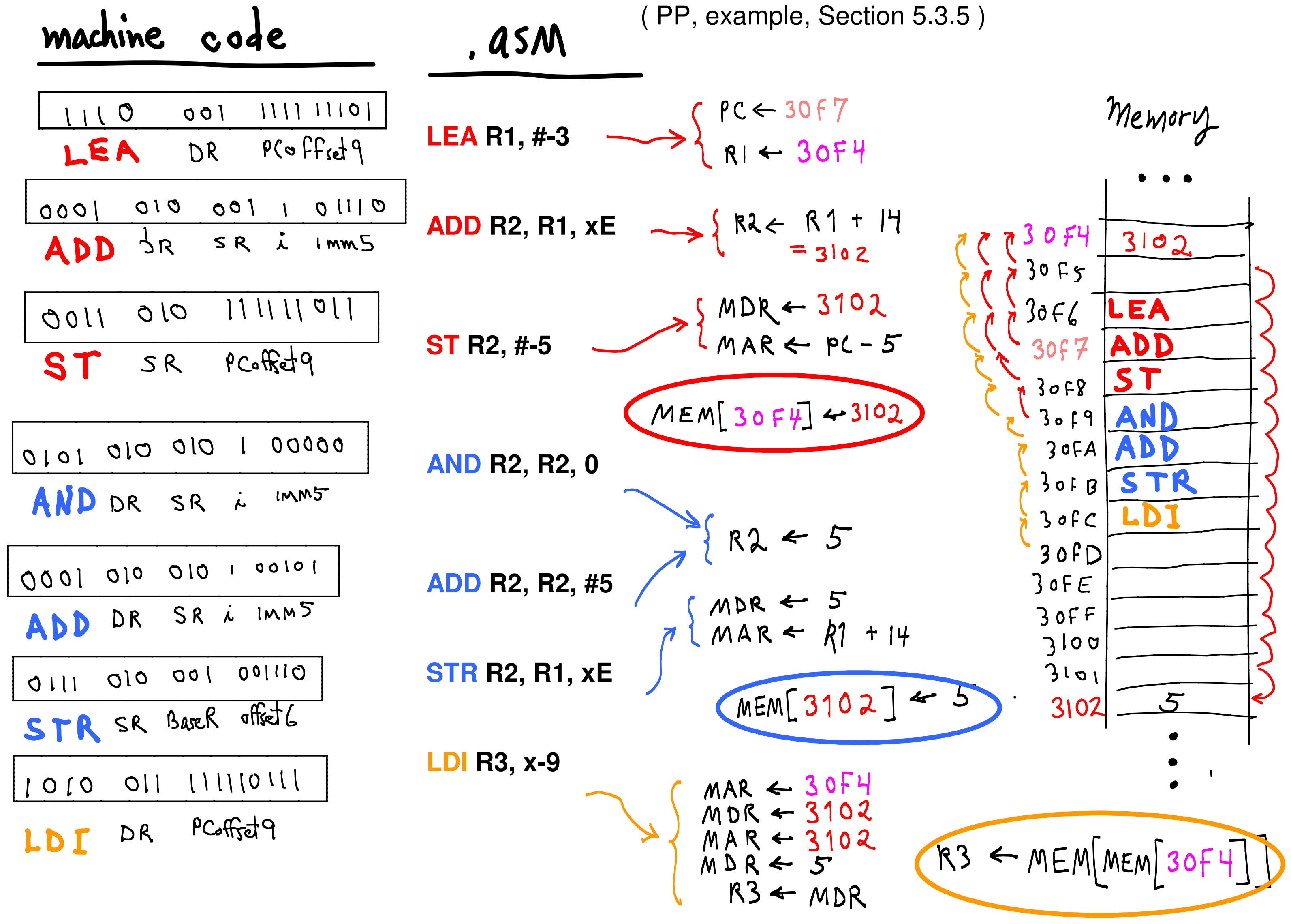
--- **RESTORE PC, PSR**: the **RTI** instruction,

PC <== POP
PSR <== POP

---- **RESTORE SP**, see **R6** save/restore hardware

```
8 RTI
   MAR  <= SP

36, 38, 39 pop PC
   MDR <= Mem   }
   PC    <= MDR  }
   SP    <= SP+1 } Pop PC
   MAR <= SP+1  }

40, 42, 34 pop PSR
   MDR <= Mem   }
   PSR <= MDR   } Pop PSR
   SP    <= SP+1 }
 <PSR[15] == 1?> (restore SP)
```

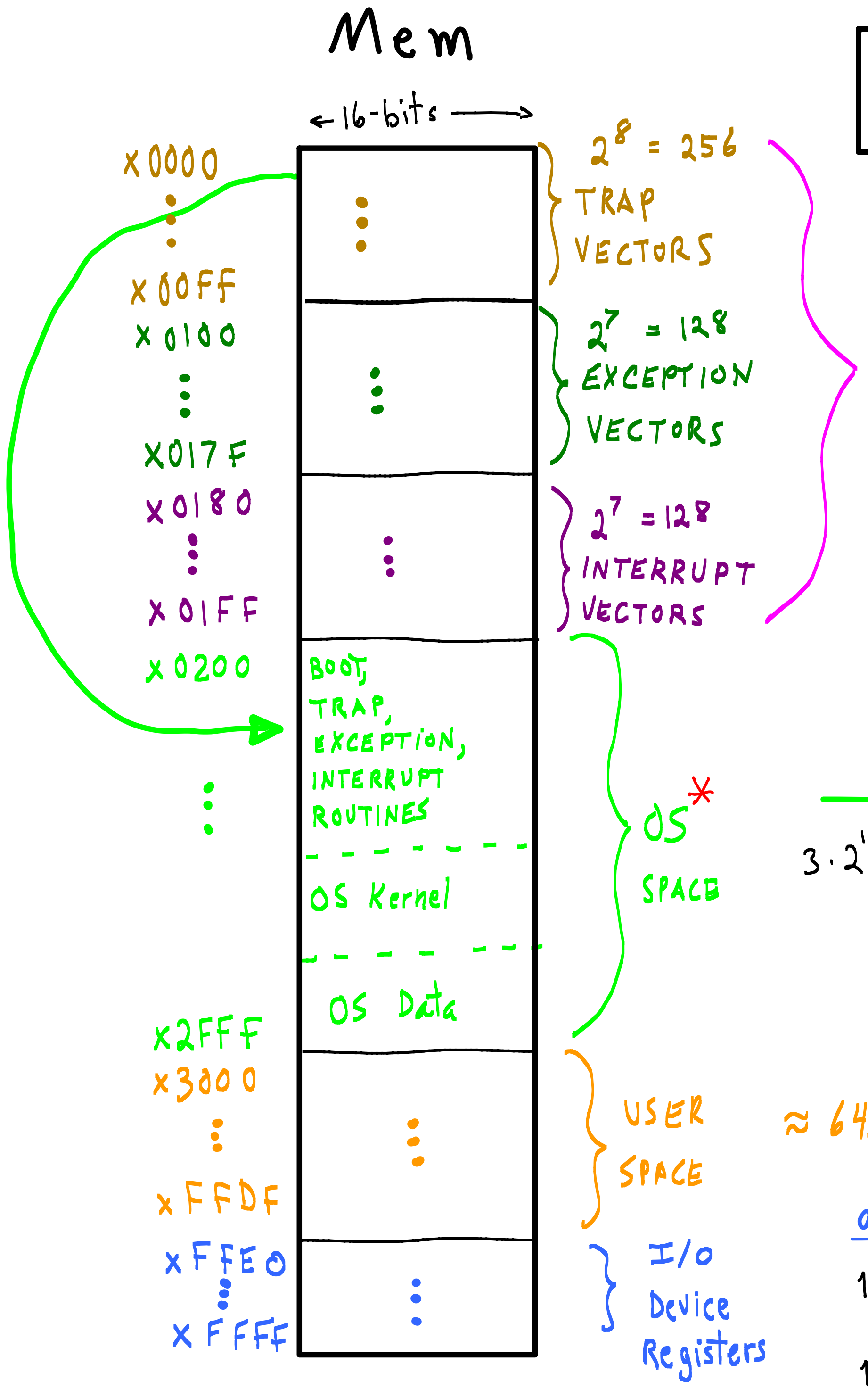
(PP, example, Section 5.3.5)



```
LEA R1, #-3
ADD R2, R1, xE
ST R2, #-5
AND R2, R2, 0
ADD R2, R2, #5
STR R2, R1, xE
LDI R3, x-9
```

;-- R1	<== &pointer	R1 gets (address of pointer variable)
;-- R2	<== &data	R2 gets (address of pointer variable + 14) == (address of data variable)
;-- pointer	<== &data	pointer variable gets (R2, address of data variable)
;-- R2	<== 0	data calculation into R2
;-- R2	<== 5	data calculation into R2
;-- data	<== 5	MEM[(R1, address of pointer variable) + 14] gets data, R2
;-- R3	<== data	R3 gets data from MEM via de-referencing pointer variable.

LC3 Memory Map



Vector Table, hardware defined

VT space

$x200 = 2 \times 16^2 = 2 \times (2^4)^2 = 2^9 = \frac{1}{2}k$

OS space

$3 \cdot 2^{12} = 3 \cdot (2^2) \cdot 2^{10} = 12k$ w/ VT

$\approx 64k - 12k = 53k$

device address range

```
1111 1111 1110 0000    FFE0
.... ...  ...  . . .    . . .
1111 1111 1111 1111    FFFF
<------------->  <--->
                  ↖ 5 bits → 32 registers
```

If these bits, addrBus[15:5], are all 1's, reference is to I/O device register, not memory.