Week 3
Further into the MSP430

Hacettepe University

MSP430MtFaFbMc

- Mt: Memory Type
  - **C: ROM**
  - **F: Flash**
  - **P: OTP**
  - **E: EPROM (for developmental use. There are few of these.)**
- Fa, Fb: Family and Features
  - **10, 11: Basic**
  - **12, 13: Hardware UART**
  - **14: Hardware UART, Hardware Multiplier**
  - **31, 32: LCD Controller**
  - **33: LCD Controller, Hardware UART, Hardware Multiplier**
  - **41: LCD Controller**
  - **43: LCD Controller, Hardware UART**
  - **44: LCD Controller, Hardware UART, Hardware Multiplier**
- Mc: Memory Capacity
  - **0: 1kb ROM, 128b RAM**
  - **1: 2kb ROM, 128b RAM**
  - **2: 4kb ROM, 256b RAM**
  - **3: 8kb ROM, 256b RAM**
  - **4: 12kb ROM, 512b RAM**
  - **5: 16kb ROM, 512b RAM**
  - **6: 24kb ROM, 1kb RAM**
  - **7: 32kb ROM, 1kb RAM**
  - **8: 48kb ROM, 2kb RAM**
  - **9: 60kb ROM, 2kb RAM**

Example

- The MSP430F435 is a Flash memory device with an LCD controller, a hardware UART, 16 kb of code memory, and 512 bytes of RAM.

Microcontroller characteristics

- Integration: Able to implement a whole design onto a single chip.
- Cost: Are usually low-cost devices (a few $ each);
- Clock frequency: Compared with other devices (microprocessors and DSPs), MCUs use a low clock frequency:
  - MCUs today run up to 100 MHz/100 MIPS (Million Instructions Per Second).
- Power consumption: Low power (battery operation);
- Bits: 4 bits (older devices) to 32 bits devices;
- Memory: Limited available memory, usually less than 1 MByte;
- Input/Output (I/O): Low to high (8 to 150) pin-out count.

MSP430 main characteristics (1/3)

- Low power consumption:
  - 0.1 μA for RAM data retention;
  - 0.8 μA for real-time clock mode operation;
  - 250 μA/MIPS during active operation.

- Low operation voltage (from 1.8 V to 3.6 V);

- < 1 μs clock start-up;

- < 50 nA port leakage;

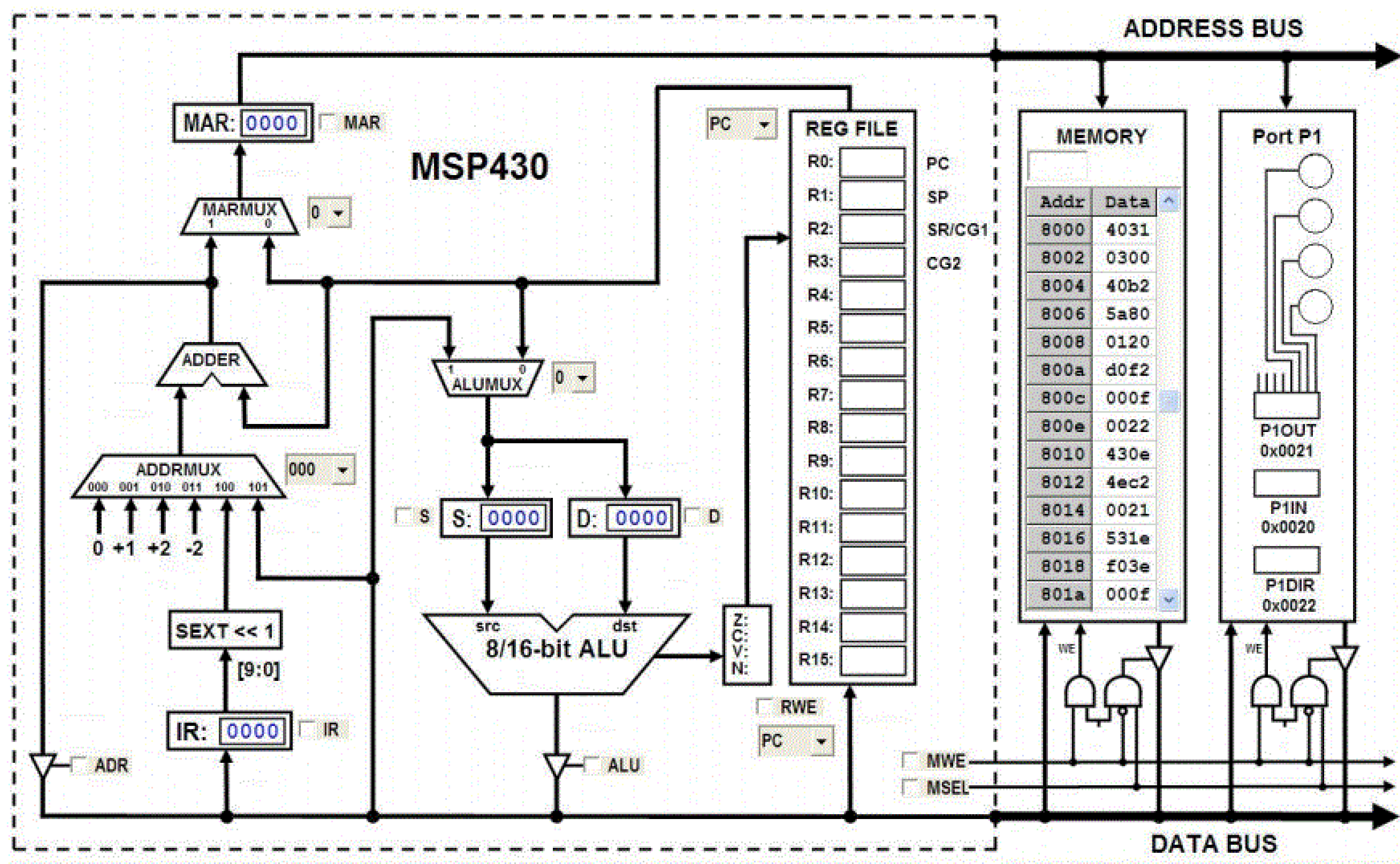
- Zero-power Brown-Out Reset (BOR).

MSP430 main characteristics (3/3)

- Flexibility:
  - Up to 256 kByte Flash;
  - Up to 100 pins;
  - USART, I2C, Timers;
  - LCD driver;
  - Embedded emulation;
  - And many more peripherals modules…

- Microcontroller performance:
  - Instruction processing on either bits, bytes or words
  - Reduced instructions set;
  - Compiler efficient;
  - Wide range of peripherals;
  - Flexible clock system.
- 1.8–3.6V operation

MSP430 Architecture



MACHINE VS. ASSEMBLY LANGUAGE

- **Machine Language Instructions**
  - **Sequence of zeros and ones understood by the CPU**
  - **Hard to read by humans**
  - **Consists of several fields**
  - **Opcode, source and destination fields, and an optional datum**
- **Assembly Language Instructions**
  - **A human understandable notation for machine language**
  - **One assembly instruction per machine language instruction**
  - **Consists of several fields**
  - **A mnemonic followed by zero or more operands**
- **Assembly Process**
  - **Converts an assembly language program into a machine language program**

Copyright 2009 Texas Instruments
All Rights Reserved

Machine vs Assembly language

Machine language	Assembly language
480D	`mov R8,R13`
5079 006B	`add.b #0x6B,R9`
23F1	`jnz 0x3E2`
1300	`reti`

Copyright 2009 Texas Instruments
All Rights Reserved

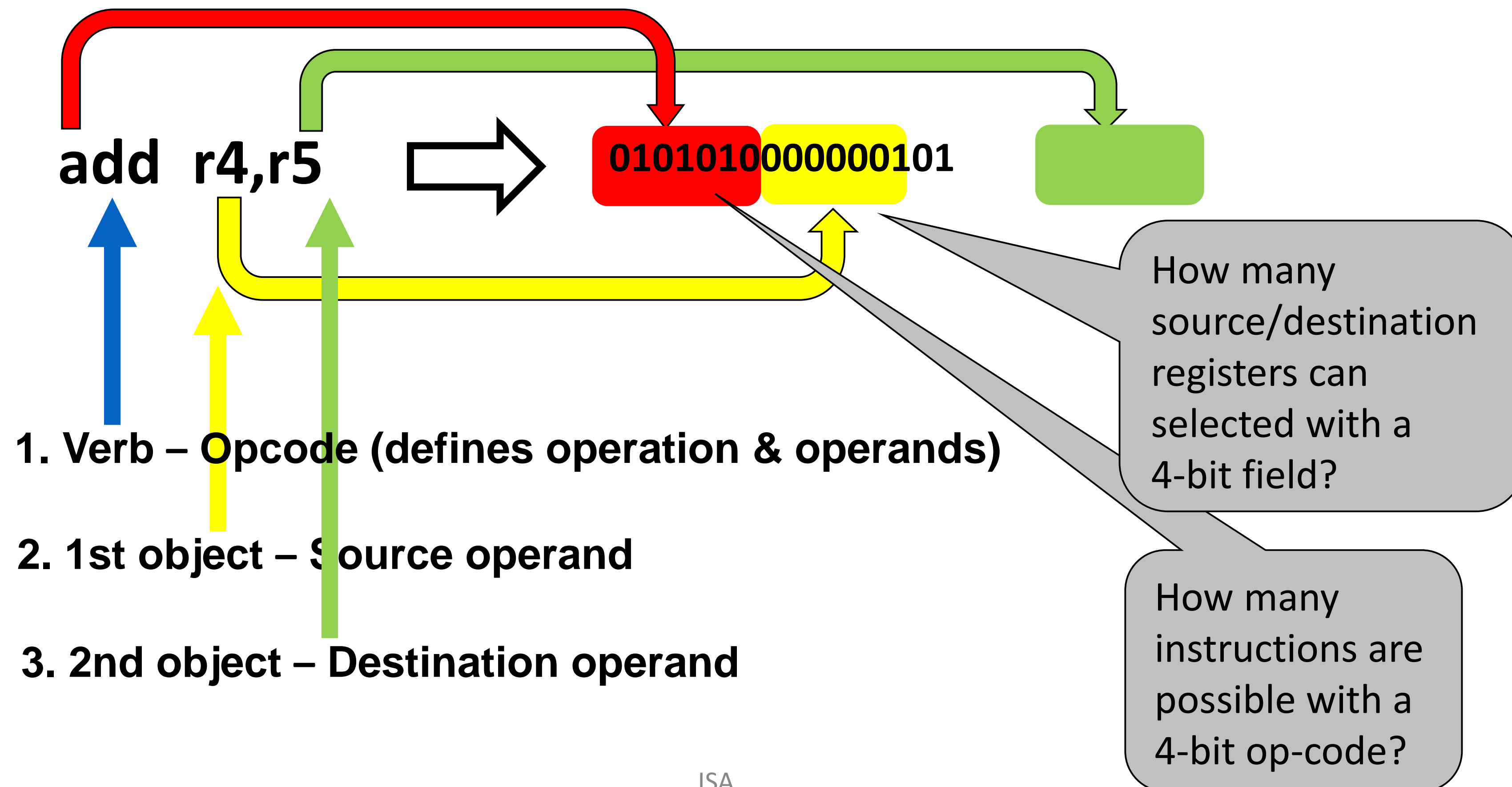
Computer Instructions

- Computer program consists of a sequence of instructions
  - instruction = verb + operand(s)
  - stored in memory as 1’s and 0’s
  - called machine code.
- Instructions are *fetched* from *memory*
  - The **program counter (PC)** holds the memory address of the next instruction (or operand).
  - The instruction is stored internal to the CPU in the **instruction register (IR)**.
- Programs execute sequentially through memory
  - **Execution order** is altered by changing the program counter **(PC)**.
  - A **computer clock** controls the speed and phases of instruction execution.

Machine vs Assembly Code

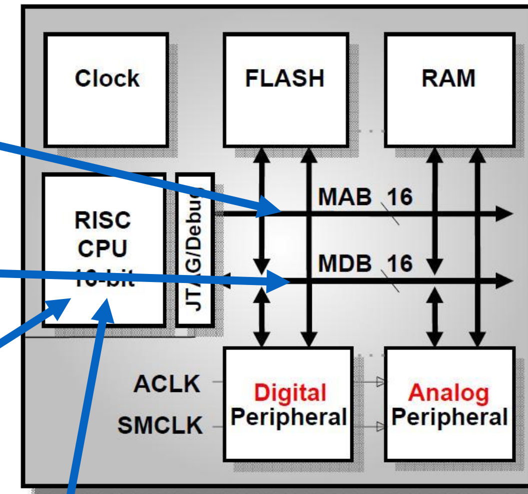
Machine Code		Assembly Code
0100000000110001		mov.w #0x0600,r1
0000011000000000		
0100000010110010		mov.w #0x5a1e,&0x0120
0101101000011110	Assembler ←	
0000000100100000		
0100001100001110		mov.w #0,r14
0101001101011110		add.b #1,r14
1111000001111110		and.b #0x0f,r14
0000000000001111		
0001001000110000	Disassembler →	push #0x000e
0000000000001110		
1000001110010001		sub.w #1,0(r1)
0000000000000000		
0010001111111101		jne $-4
0100000100111111		mov.w @r1+,r15

Anatomy of Machine Instruction



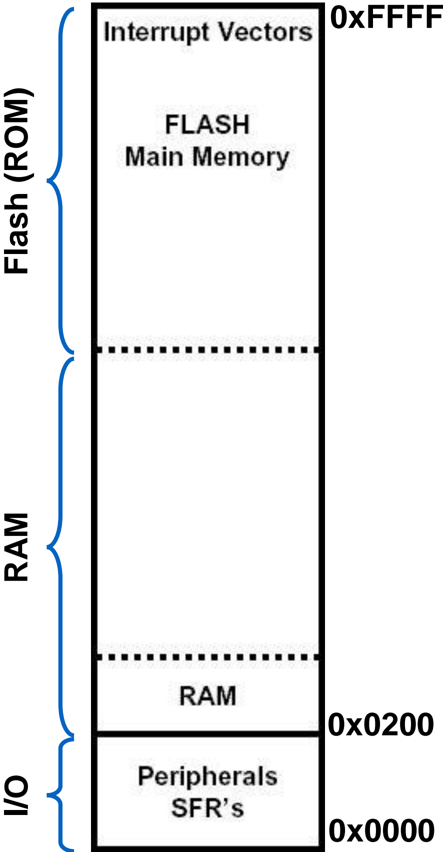
MSP430 Bus Architecture

- Memory Address Bus (uni-directional)
  - **Address Space** = number of possible memory locations (memory size)
- Memory Data Bus (bi-directional)
  - **Addressability** = # of bits stored in each memory location (8-bits).
  - Words are **always** addressed at an even address (**little endian**).
- Sixteen 16-bit registers
  - Program Counter (R0), Stack Pointer (R1), Status Register (R2), Constant Generator (R3), General Purpose Registers (R4-R15).
- 16-bit ALU (Arithmetic and Logic Unit)
  - Sets condition codes: Z, C, N, V
  - The master clock (MCLK) drives the CPU and ALU logic.



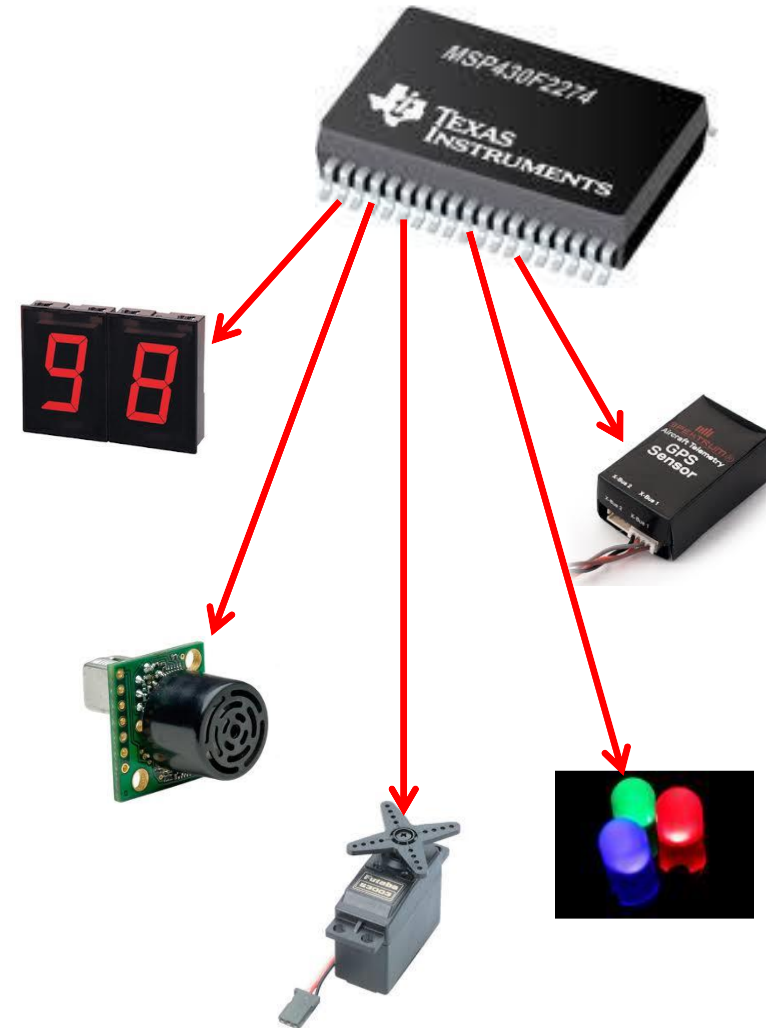
MSP430 Memory Architecture

- Memory
  - 64k byte addressable, address space (0x0000 - 0xFFFF)
  - Flash / ROM – Used for both code/data
    - Interrupt vectors - Upper 16 words
  - RAM (0x0200) – Volatile storage
- Input / Output
  - Get information in and out of the computer.
  - External devices attached to a computer are called peripherals.
  - Lower 512 bytes (0x0000 - 0x01FF) of address space
    - 16-bit peripherals (0x0100 - 0x01FF)
    - 8-bit peripherals (0x0010 - 0x00FF)
    - Special Function Registers – Lower 16 bytes



MSP430 Ports

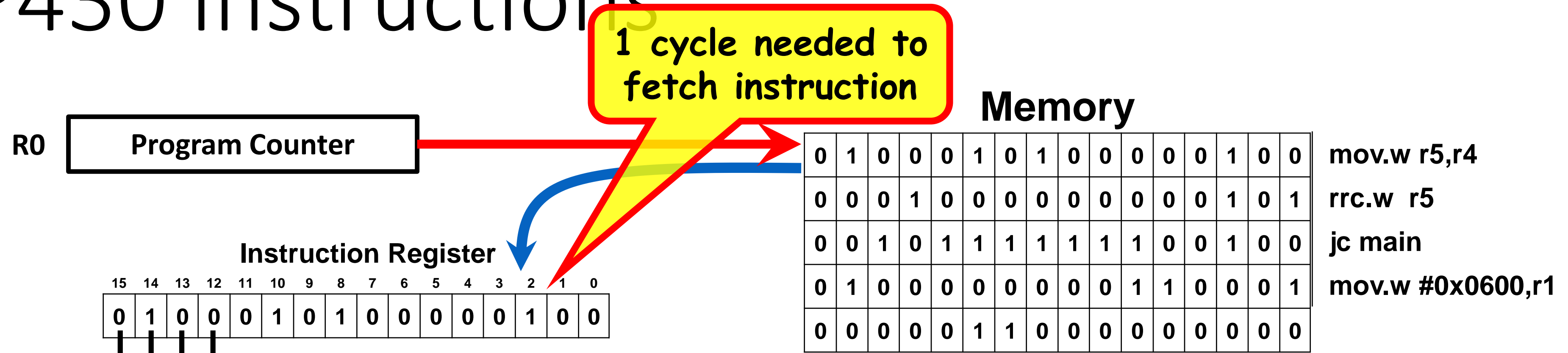
- Computer communicates with external world thru 8 bit memory locations called Ports.
  - Each Port bit is independently programmable for Input or Output.
  - Edge-selectable input interrupt capability (P1/P2 only) and programmable pull-up/pull-down resistors available.
- Port Registers
  - PxIN – read from port
  - PxOUT – write to port
  - PxDir – set port direction (input or output)

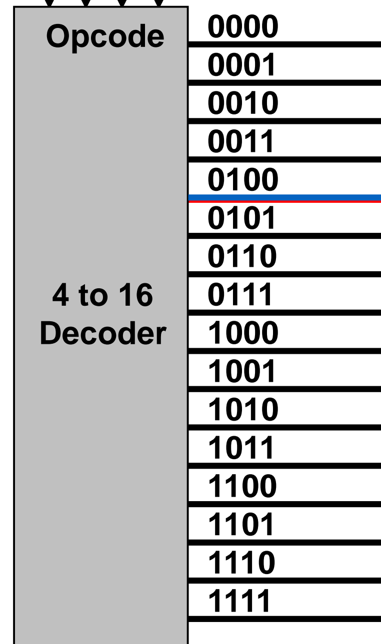


MSP430 Instructions

- The first 4-bits (nybble) of an instruction is called the **opcode** and specifies the instruction and format.
- The MSP430 ISA defines 27 instructions with **three instruction formats**: double operand, single operand, and jumps.
- Single and double operand instructions process **word** (16-bits) or **byte** (8-bit) data operations. (Default is word)
- **Orthogonal** instruction set – every instruction is usable with every addressing mode throughout the entire memory map.
- Includes high register count, no paging, stack processing, memory to memory operations, constant generator.

MSP430 Instructions

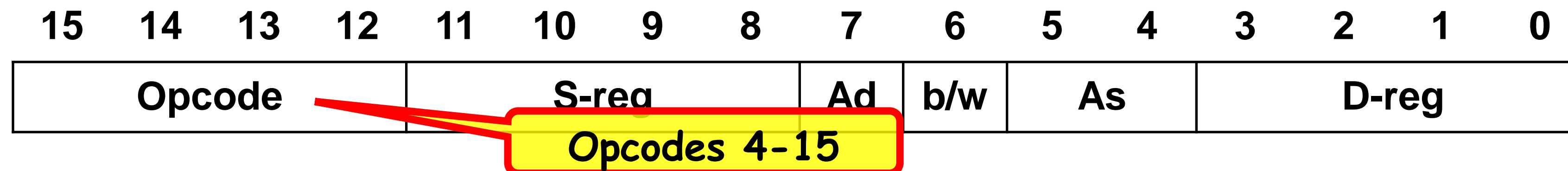




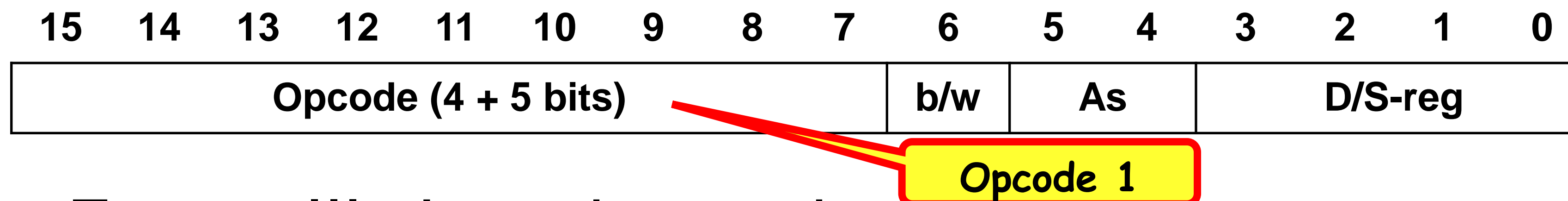
Opcode	Instruction	Format
0000	Undefined	Single Operand
0001	RCC, SWPB, RRA, SXT, PUSH, CALL, RETI	
0010	JNE, JEQ, JNC, JC	Jumps
0011	JN, JGE, JL, JMP	
0100	MOV	Double Operand
0101	ADD	
0110	ADDC	
0111	SUBC	
1000	SUB	
1001	CMP	
1010	DADD	
1011	BIT	
1100	BIC	
1101	BIS	
1110	XOR	
1111	AND	

MPS430 Instruction Formats

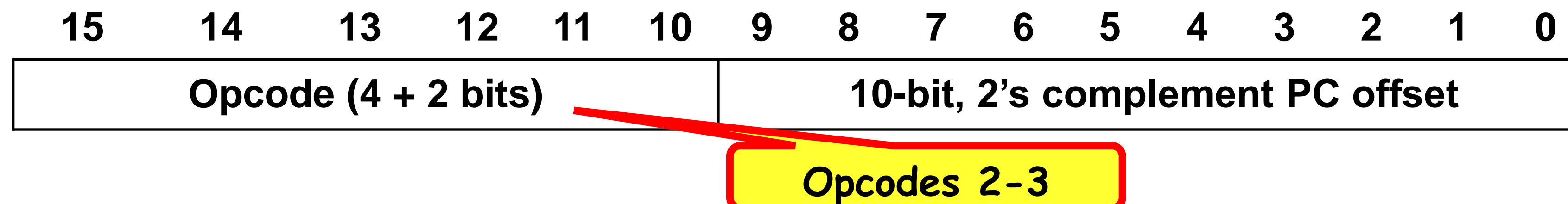
- Format I: Instructions with two operands:



- Format II: Instruction with one operand:



- Format III: Jump instructions:



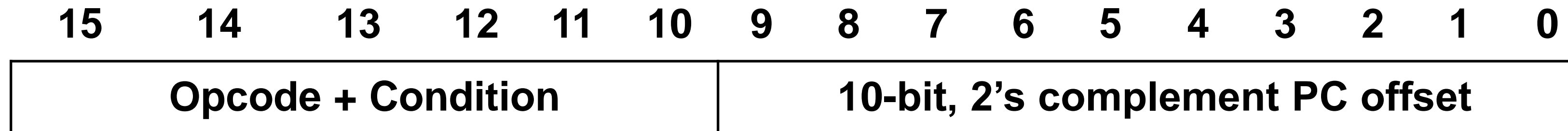
Format I: Double Operand

Mnemonic	Operation	Description
Arithmetic instructions		
ADD(.B or .W) src,dst	src+dst→dst	Add source to destination
ADDC(.B or .W) src,dst	src+dst+C→dst	Add source and carry to destination
DADD(.B or .W) src,dst	src+dst+C→dst (dec)	Decimal add source and carry to destination
SUB(.B or .W) src,dst	dst+.not.src+1→dst	Subtract source from destination
SUBC(.B or .W) src,dst	dst+.not.src+C→dst	Subtract source and not carry from destination
Logical and register control instructions		
AND(.B or .W) src,dst	src.and.dst→dst	AND source with destination
BIC(.B or .W) src,dst	.not.src.and.dst→dst	Clear bits in destination
BIS(.B or .W) src,dst	src.or.dst→dst	Set bits in destination
BIT(.B or .W) src,dst	src.and.dst	Test bits in destination
XOR(.B or .W) src,dst	src.xor.dst→dst	XOR source with destination
Data instructions		
CMP(.B or .W) src,dst	dst-src	Compare source to destination
MOV(.B or .W) src,dst	src→dst	Move source to destination

Format II: Single Operand

Mnemonic	Operation	Description
Logical and register control instructions		
`RRA(.B or .W) dst`	MSB→MSB→… LSB→C	Roll destination right
`RRC(.B or .W) dst`	C→MSB→…LSB→C	Roll destination right through carry
`SWPB(.W) dst`	Swap bytes	Swap bytes in destination
`SXT(.W) dst`	bit 7→bit 8…bit 15	Sign extend destination
`PUSH(.B or .W) src`	SP-2→SP, src→@SP	Push source on stack
Program flow control instructions		
`CALL dst`	SP-2→SP, PC+2→@SP dst→PC	Subroutine call to destination
`RETI`	@SP+→SR, @SP+→SP	Return from interrupt

Format III: Jump Instruction



- Jump instructions are used to direct program flow to another part of the program (by changing the PC).
- The condition on which a jump occurs depends on the Condition field consisting of 3 bits:
  - **JNZ/JNE** 000: jump if not equal (Z = 0)
  - **JZ/JEQ** 001: jump if equal (Z = 1)
  - **JNC/JLO** 010: jump if no carry (C = 0)
  - **JC/JHS** 011: jump if carry (C = 1)
  - **JN**100: jump if negative (N = 1)
  - **JGE** 101: jump if greater than or equal (N = V)
  - **JL** 110: jump if lower (N ≠ V)
  - **JMP** 111: unconditional jump

Different Machine Instructions for MSP430

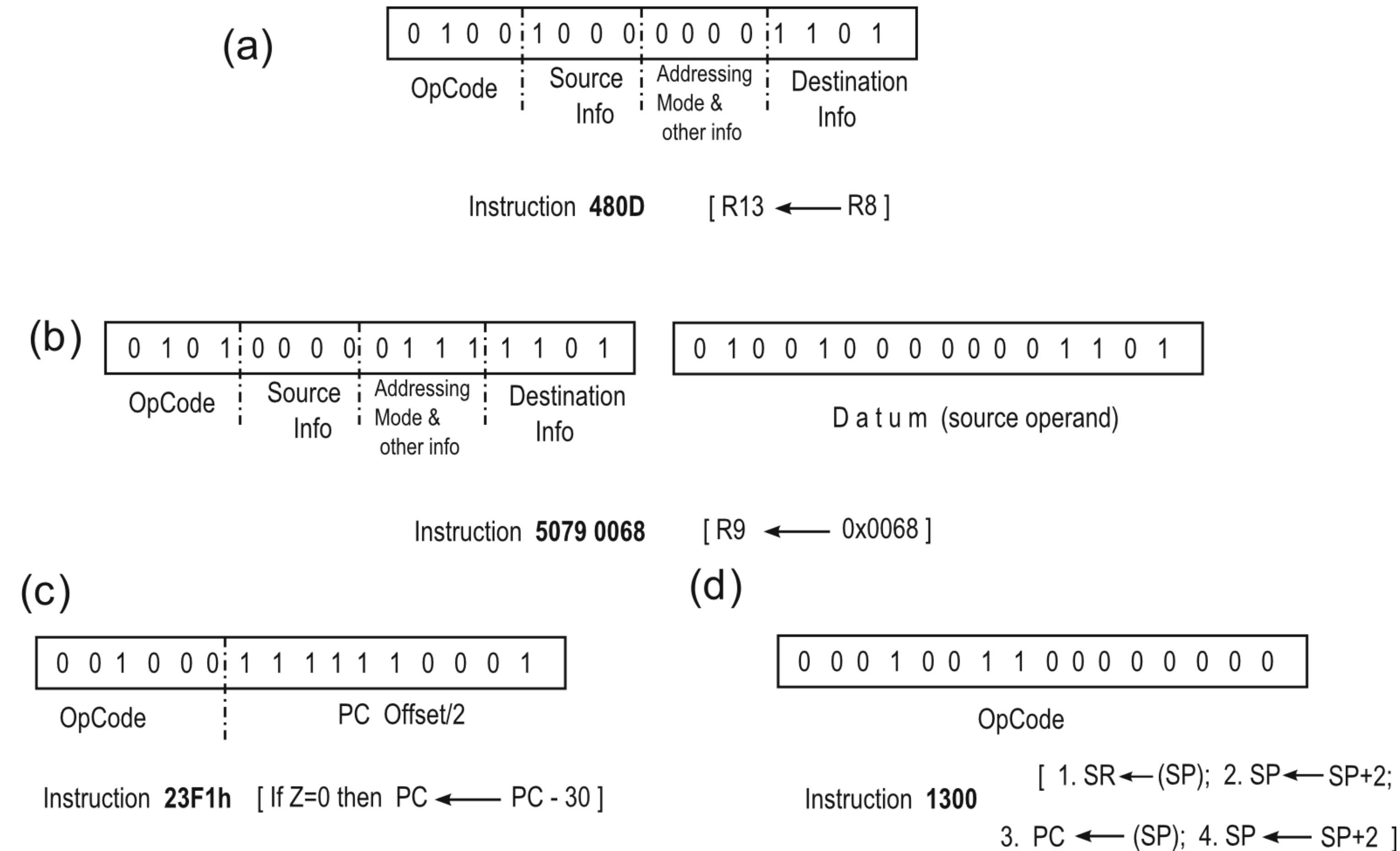


Fig. 3.23 Four machine language instructions for the MSP430

Copyright 2009 Texas Instruments
All Rights Reserved

Basic Assembly Process

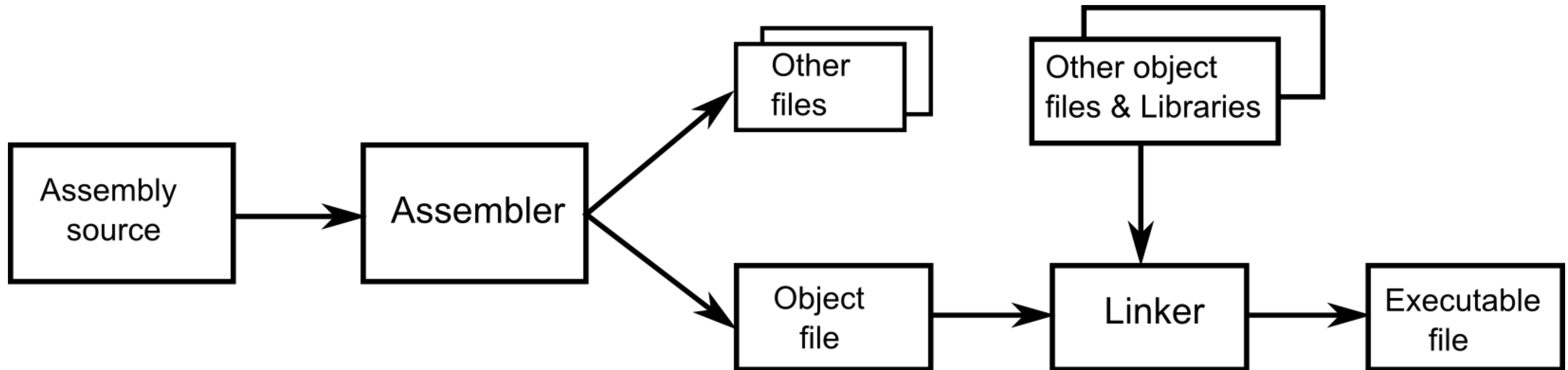


Fig. 3.24 Basic assembly process

Copyright 2009 Texas Instruments
All Rights Reserved

TOOLS FOR MSP430

Table 3.7 Programming and debugging tools for MSP430

Site	Simulator	C compiler	Assembler	Linker
CCS	X	X	X	X
IAR	X	X	X	X
mspgcc		X	X	X
naken	X		X	X
pds-430	X	X	X	
cdk4msp	X			
mcc-430		X		

Copyright 2009 Texas Instruments
All Rights Reserved

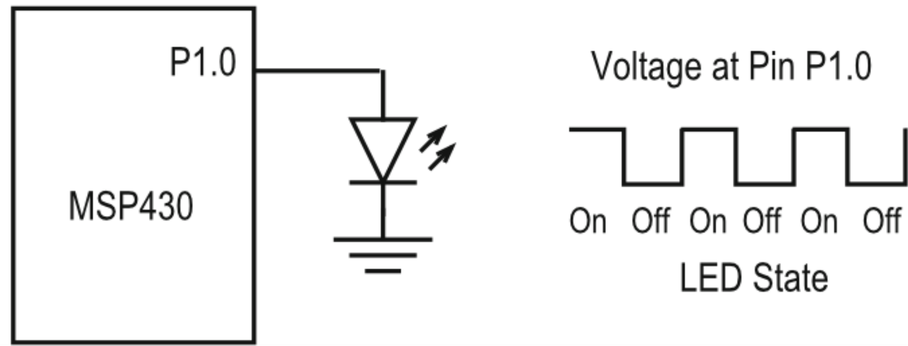


Fig. 4.1 Simplified hardware connection diagram for LED in board

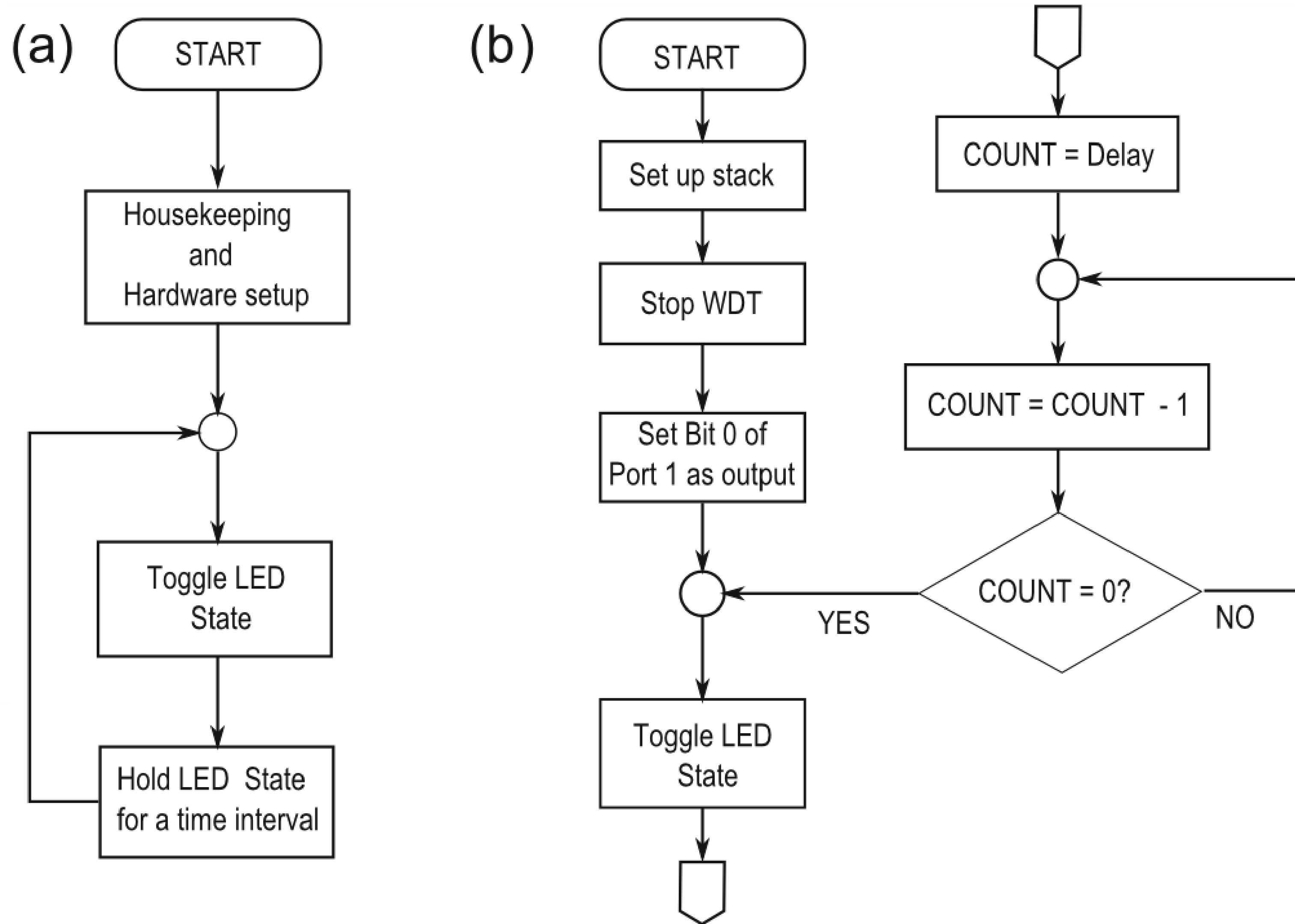


Fig. 4.2 Flow diagram for the blinking LED: **a** General Concept; **b** Expanded flow diagram

Assembler Coding Style

```
;   blinky.asm: Software Toggle P1.0
            .cdecls C,"msp430.h"                ; MSP430 C header
            .def    RESET

DELAY       .equ    0

            .bss    cnt,2                              riable

            .text                               ; begin code
C000: 4031 0280       mov.w   #0x0280,SP                ; init stack ptr
C004: 40B2 5A80 0120  mov.w   #WDTPW+WDTHOLD,&WDTCTL    ; stop WDT
C00a: DED2 0022       bis.b   #0x01,&P1DIR              ; set P1.0 output

C00e: E3D2 0021       xor.b   #0x01,&P1OUT
C012: 4380 41EC       mov.w   #DELAY,cnt

C016: 8390 41E8       sub.w   #1,cnt                    ; delay over?
C01a: 23FD              jnz   delaylp                   ; n
C01c: 3FF8            jmp     mainloop                  ; y, repeat

            .sect   ".reset"                    ; RESET vector
            .word   RESET                       ; start address
            .end
```

Put defines & variables here

Start executable code after .text directive

Put start label here

Listing 4.1: C Language Listing

```
1  #include  <msp430x12x.h>
2  void main(void)
3  { WDTCTL = WDTPW + WDTHOLD; /* Stop watchdog timer */
4    P1DIR |= 0x01;          /* Set P1.0 to output direction */
5    for (;;)
6    { unsigned int i;
7      P1OUT ^= 0x01;        /* Toggle P1.0 using ex-or */
8      i = 50000;            /* Delay */
9      do (i--);
10     while (i != 0);
11   }
12 }
```

Fig. 4.3 C language code for LED toggling

Listing 4.2: Binary Machine Language

```
1   0100000000110001 0000001100000000
2   0100000010110010 0101101010000000 0000000100100000
3   1101001111010010 0000000000100010
4   1110001111010010 0000000000100001
5   0100000000111111 1100001101010000
6   1000001100011111
7   0010001111111110
8   0011111111111001
```

Listing 4.3: Hex

```
4031 0300
40B2 5A80 0120
D3D2 0022
E3D2 0021
403F C350
831F
23FE
3FF9
```

Fig. 4.4 Executable machine language code for the example of Fig. 4.1

Listing 4.4: Assembly Version

```
1  mov.w   #0x300,SP
2  mov.w   #0x5A80,&0x0120
3  bis.b   #001,&0x0022
4  xor.b   #001,&0x0021
5  mov.w   #0xC350,R15
6  dec.w   R15
7  jnz     0x3FC
8  jmp     0x3F2
```

Hex Machine Lang.

```
4031 0300
40B2 5A80 0120
D3D2 0022
E3D2 0021
403F C350
831F
23FE
3FF9
```

Fig. 4.5 Assembly version for Fig. 4.4—Hex machine version shown for comparison

Listing 4.5: Assembly Code

```
1  ;Constants Declarations
2  #include   "msp430g2231.h"
3  LED        EQU     0x0001    ; LED at P1.0
4  DELAY      EQU     50000     ;
5  #define    COUNTER  R15
6  ;- - - - - - - - - - - - - - - - - - - - - -
7                 ORG 0F800h        ;Start Code
8  ;- - - - - - - - - - - - - - - - - - - - - -
9  RESET      mov.w   #300h, SP        ; Set stack
10 StopWDT    mov.w   #WDTPW+WDTHOLD,&WDTCTL
11                                     ; Stop WDT
12 SetupP1    bis.b   #001h,&P1DIR   ;P1.0 output
13     ;
14 Mainloop   xor.b   #LED,&P1OUT   ; Toggle P1.0
15 Wait       mov.w   #DELAY,COUNTER
16                          ;Load Delay to Counter
17 L1         dec.w   COUNTER           ; wait
18            jnz     L1           ; Delay over?
19            jmp     Mainloop          ; Again
```

Hex Code

```
4031 0300
40B2 5A80 0120

D3D2 0022

E3D2 0021
403F C350

831F
23FE
3FF9
```

Fig. 4.6 Assembly language code for Fig. 4.4

Fig. 4.7 Example of a list file line

F80A: D3D2 0022 SetUpP1 bis.b #001h,&P1DIR

Instruction Address | Machine lang. Instruction | Assembly Language Instruction

SetupP1 bis.b #001h, &P1DIR ; P1.0 Output

Label | Mnemonics | Operands | Comment

- Mnemonic src, dst or
- Mnemonic operand
- Mnemonic

- .b suffix represents byte size operand
- .w suffix represents word size operand

Directives

- Directives are for the assembler only!
- They do not translate into machine code or data to be loaded into microcontroller memory
- They serve to organize the program,
- Depends on the assembler ☹
- Some examples:
  - EQU and #define
    - LABEL EQU <Value or Expression>
    - #define <Symbolic Name> Value or expression or register
  - #include
    - #include “filename”
  - ORG

Directives

```
                                ORG       0F800h                   ;
F800 4031 0300       RESET      mov.w     #300h,SP                 ; Set stack
F804 40B2 5A80 0120  StopWDT    mov.w     #WDTPW+WDTHOLD,&WDTCTL   ; Stop WDT
F80A D3D2 0022                  bis.b     #LED,&P1DIR              ; P1.0 output
F80E ------                 - - - - -
     --                       - -
```

Information for watchdog timer control register

Fig. 4.8 Information for watchdog timer control register (WDTCTL) (*Courtesy of Texas Instruments, Inc.*)

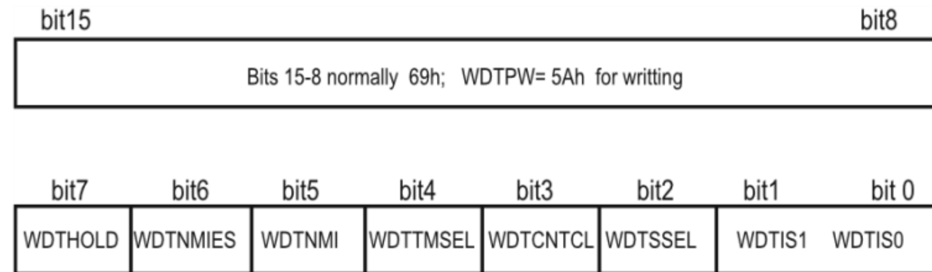


Table 4.1 Symbolic constants associated to watchdog timer control register (WDTCTL)

Name	Number	Comment
WTDCTL	0x0120	Register address
WTDPW	0x5A00	Required to make changes
WTDHOLD	0x0080	When set, stops WDT
WDTNMIES	0x0040	selects the interrupt edge for the NMI interrupt when WDTNMI = 1
WDTNMI	0x0020	Selects pin $\overline{RST}$/NMI function: 0 for Reset, 1 for NMI
WDTTMSEL	0x0010	WDT working mode: 0 for WDT, 1 for interval timer
WDTCNTCL	0x0008	Resets or clears the counter when set
WDTSSEL	0x0004	WDT clock source select: 0 for SMCLK, 1 for ACLK
WDTISx	(bits 1, 0)	WDT interval select. (Explanation below)

Mov #0x05A80, &0x0120

Information for watchdog timer control register

Fig. 4.8 Information for watchdog timer control register (WDTCTL) (*Courtesy of Texas Instruments, Inc.*)

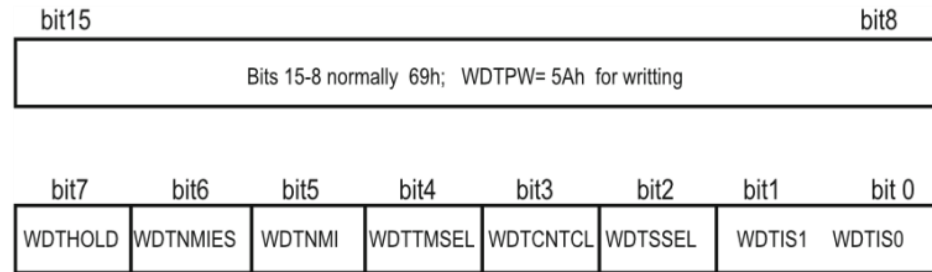


Table 4.2 Symbolic constants associated to watchdog timer control register (WDTCTL)

Name	b1-b0 value	Using bits	Time interval
WTDIS_0	0x00	None	Watchdog clock source/32768
WTDIS_1	0x01	WTDIS0	Watchdog clock source/8192
WTDIS_2	0x02	WTDIS1	Watchdog clock source/512
WTDIS_3	0x03	WTDIS1+WTDIS0	Watchdog clock source/64

```
mov #WDTPW+WDTTMSL+WDTCNTCL+WDTSSEL+WDTIS0,&WDTCTL
```

Labels

- Entry statement of main code or ISR (Interrupt Service Routine).
- Entry statement of subroutine
- Instruction to which a reference is made

```
MOV.W #Mainloop, R6 ;        R6=F80EH
MOV.W Mainloop, R6  ;        R6=E3D2H
CALL #Mainloop ;             CALL A SUB WITH 0XF80EH
```

Reset vector Allocation

```
;*****************************************************;
   MSP430G2xx1 Demo - Software Toggle P1.0
;
;Description: Toggle P1.0 by xor'ing P1.0 inside
;   of a software loop.
;     ACLK = n/a, MCLK = SMCLK = default DCO
;
;                 MSP430G2xx1
;             -----------------
;          /|\|              XIN|-
;           | |                 |
;           --|RST          XOUT|-
;             |                 |
;             |             P1.0|-->LED
;
;   Based on code written by D. Dang
;   Texas Instruments Inc.
;   October 2010
;   Built with IAR Embedded Workbench Version: 5.10
;****************************************************
#include  "msp430g2231.h"                 ; standard constants
LED           EQU     01h                 ;LED at pin P1.0
DELAY         EQU     50000
#define       COUNTER R15                 ; R15 as counter
;- - - - - - - - - - - - - - - - - - - - - - - - - - -
              ORG     0F800h              ; Program Reset
;- - - - - - - - - - - - - - - - - - - - - - - - - - -
RESET         mov.w   #0300h,SP          ; Initialize stack
StopWDT       mov.w   #WDTPW+WDTHOLD,&WDTCTL ; Stop WDT
SetupP1       bis.b   #001h,&P1DIR        ; P1.0 output
                                          ;
Mainloop      xor.b   #LED,&P1OUT         ; Toggle LED
Wait          mov.w   #DELAY,COUNTER      ; Delay to counter
L1            dec.w   COUNTER             ; Decrement counter
              jnz     L1                  ; Delay over?
              jmp     Mainloop            ; Again
                                          ;
;- - - - - - - - - - - - - - - - - - - - - - - -;
             Interrupt Vectors
;- - - - - - - - - - - - - - - - - - - - - - - - - -;
              ORG     0FFFEh              ; Address for
              DW      RESET               ; RESET Vector
              END
```

Documentation

Constants Declaration

Absolute directive

Executable Code

Reset vector allocation

ig. 4.11 An absolute IAR listing for blinking LED (*Courtesy Texas Instruments Inc.*)

ADDRESSING MODES

- **Addressing modes tell the CPU how to obtain the data needed to execute an instruction**
- The data may be
  - Explicitly supplied with the instruction
  - Stored in a CPU register
  - Stored at a memory location
  - Stored in an I/O device register
- Implicit Addressing Mode
  - Operand is implicit to the instruction

ADDRESSING MODES

- Immediate Addressing Mode
  - Syntax: #Number
- Register Addressing Mode
  - Syntax: Rn
- Indexed Addressing Mode
  - Syntax: X(Rn)
- Absolute or Direct Mode
  - Syntax: &X address is X
- Indirect Register Mode
  - Syntax: @Rn
- Direct Mode X -> address is X

Copyright 2009 Texas Instruments
All Rights Reserved

Addressing Modes

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode				S-reg				Ad	b/w	As		D-reg			

Address Mode	As/*Ad	Registers	Syntax
Register	*00	R0-R2, R4-R15	**Rn**
	*00	R3	**#0**
Symbolic	*01	R0	**address**
Indexed Register	*01	R1, R4-R15	**index(Rn)**
Absolute	*01	R2	**&address**
	01	R3	**#1**
Register Indirect	10	R0-R1,R4-R15	**@Rn**
	10	R2	**#4**
	10	R3	**#2**
Immediate	11	R0	**#number**
Indirect auto-inc	11	R1,R4-R15	**@Rn+**
	11	R2	**#8**
	11	R3	**#-1**

Addressing Modes

- The locations are specified using various *addressing modes*. There are seven of these in all but we look at only some of them.
- A single character denotes the mode in the operand.
  - immediate, #: The value itself (word or byte) is given and stored in the word following the instruction. This is also known as a *literal* value.
  - absolute, &: The address of a label in memory space is given and stored in the word following the instruction.
  - @ The address of a register in memory space is given and stored in the word following the instruction.
  - register, R: This specifies one of the 16 registers in the CPU.

Copyright 2009 Texas Instruments
All Rights Reserved

Immediate

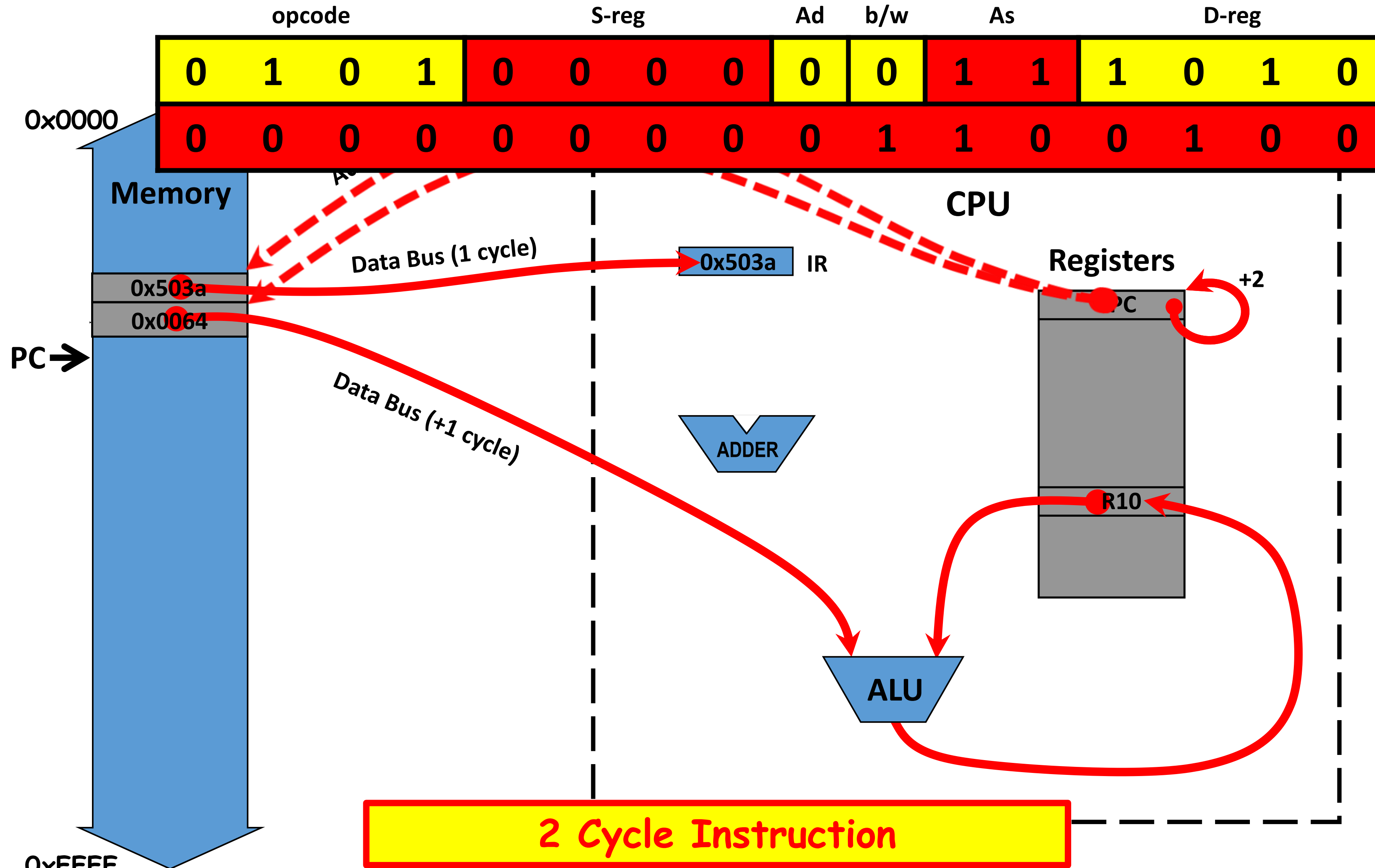
```
mov.b #00001000b,&0x0029  ; LED2 (P2.4) on ,
                          ; LED1 (P2.3) off
```

- A byte with *immediate value* (#) 00001000b is copied to the register at *address* (&) 0x0029.
- check the memory map you can confirm that this is the port P2 output register P2OUT.
- Fortunately the assembler allows us to use symbolic constants as in C, which are much clearer to understand. It substitutes their values from the header file:

```
mov.b #00001000b,& P2OUT ; LED2 (P2.4) on , LED1 (P2.3) off
```

- The header file includes a set of constants such as BIT3, which could be used instead of 00001000b.

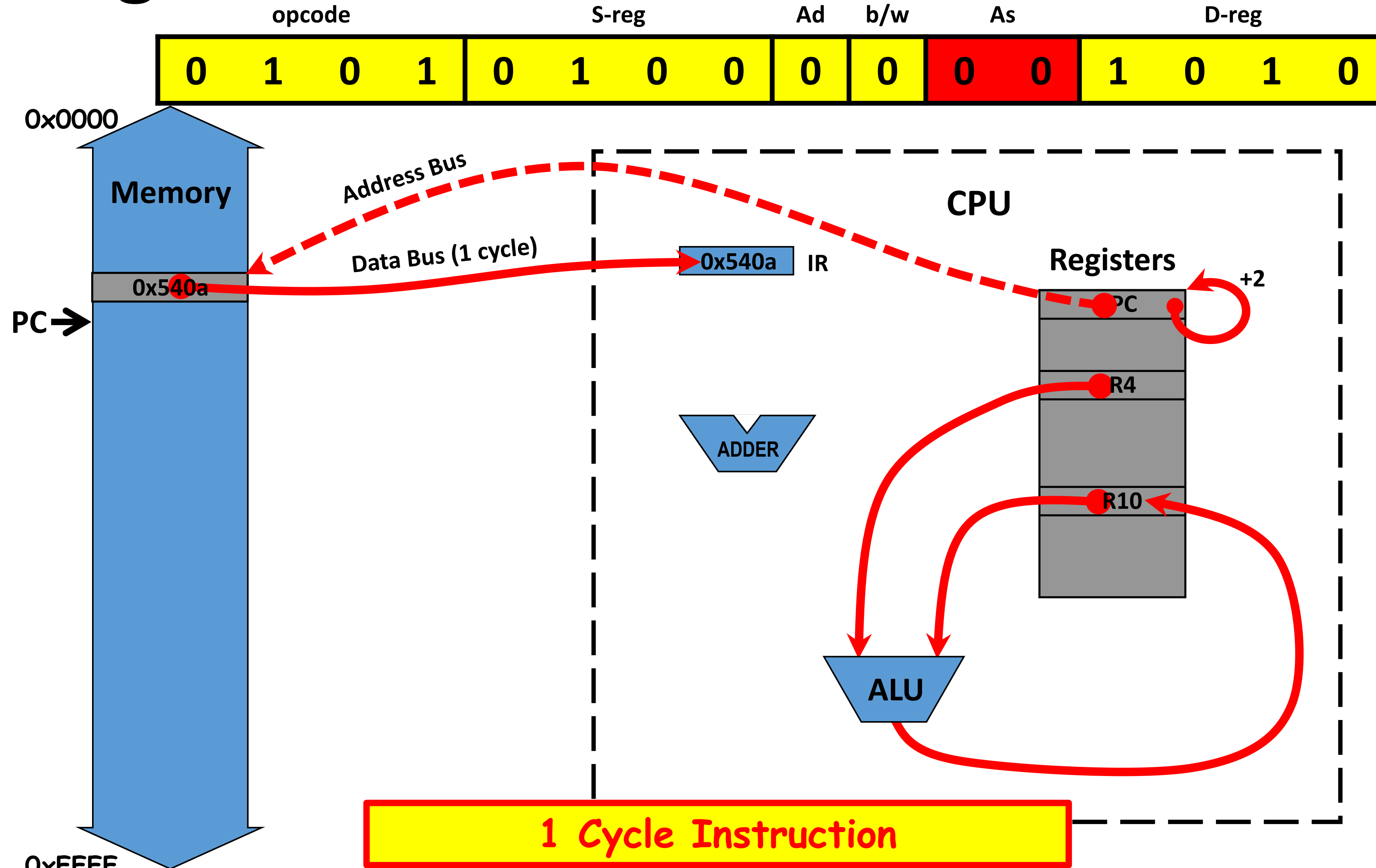
11 w/R0 = Immediate Mode



Register Addressing Mode

- mov.w R5 ,R6 *; move (copy) word from R5 to R6*
- The PC is incremented by 2 while the instruction is being fetched, before it is used as a source.
- As = 00

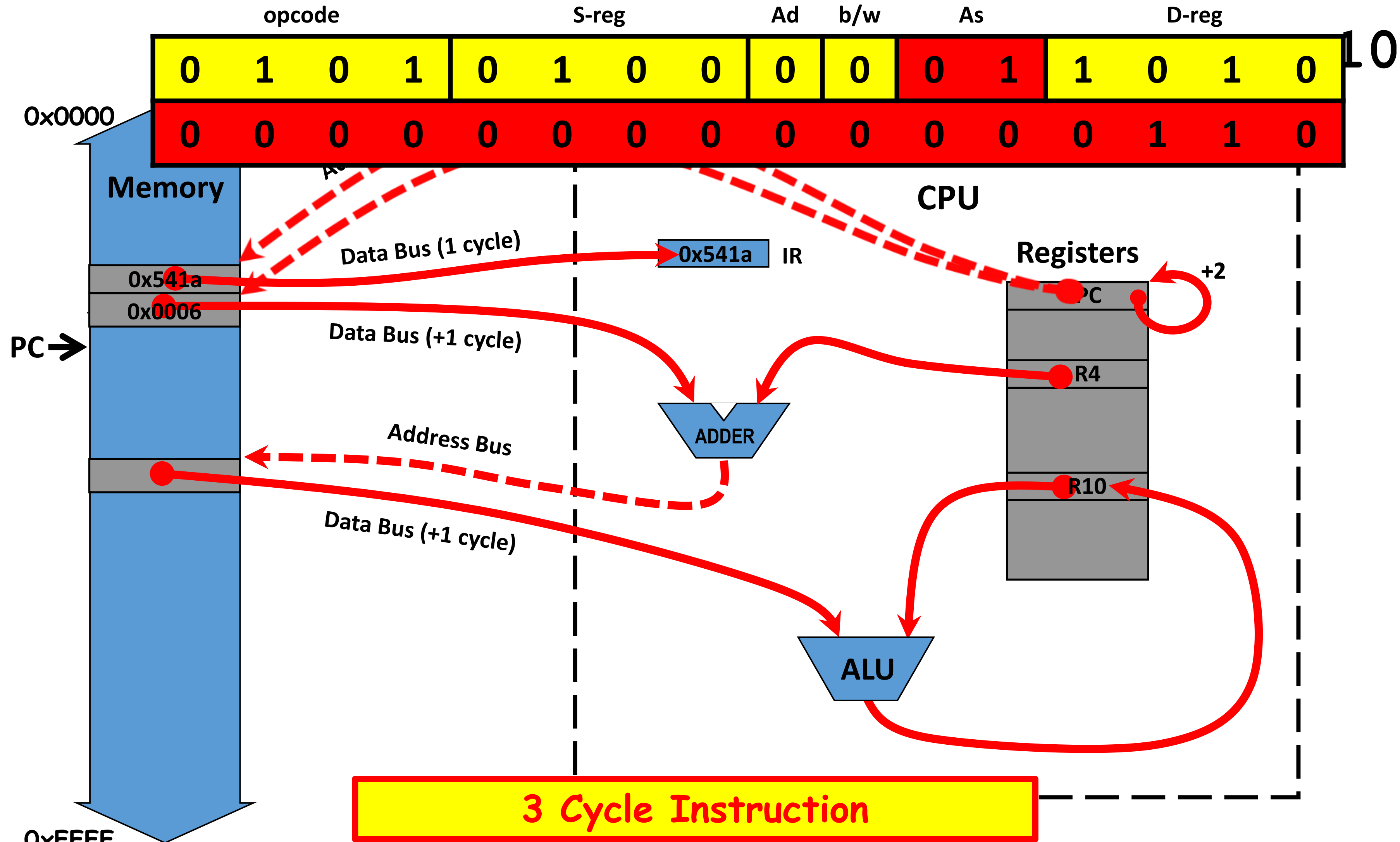
00 = Register Mode



Indexed Mode

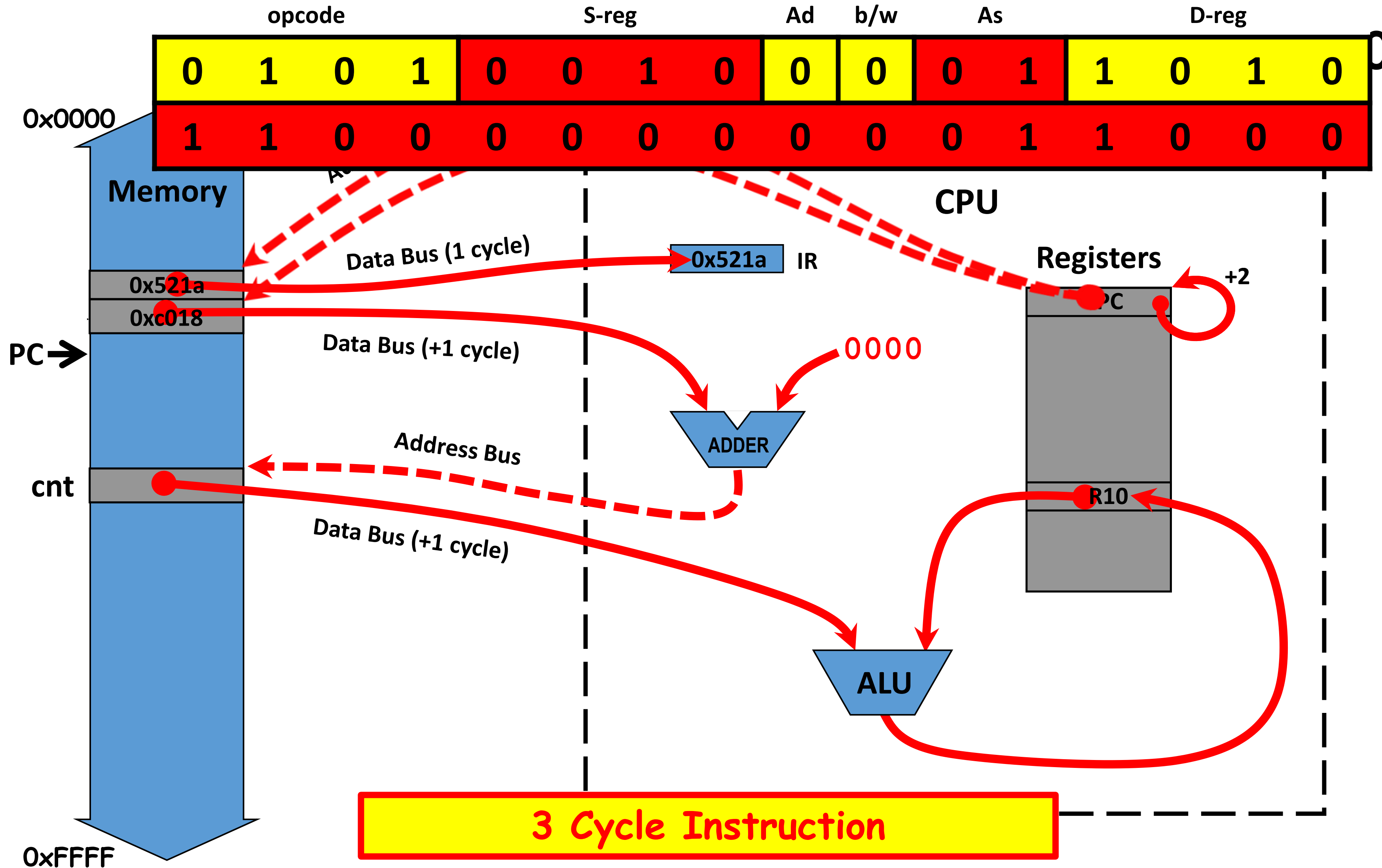
- mov.b 3(R5),R6 *; load byte from address 3+(R5) into R6*
  - Indexed addressing can be used for the source, destination, or both
  - R5 is used for the index here
  - As = 01

01 = Indexed Mode



3 Cycle Instruction

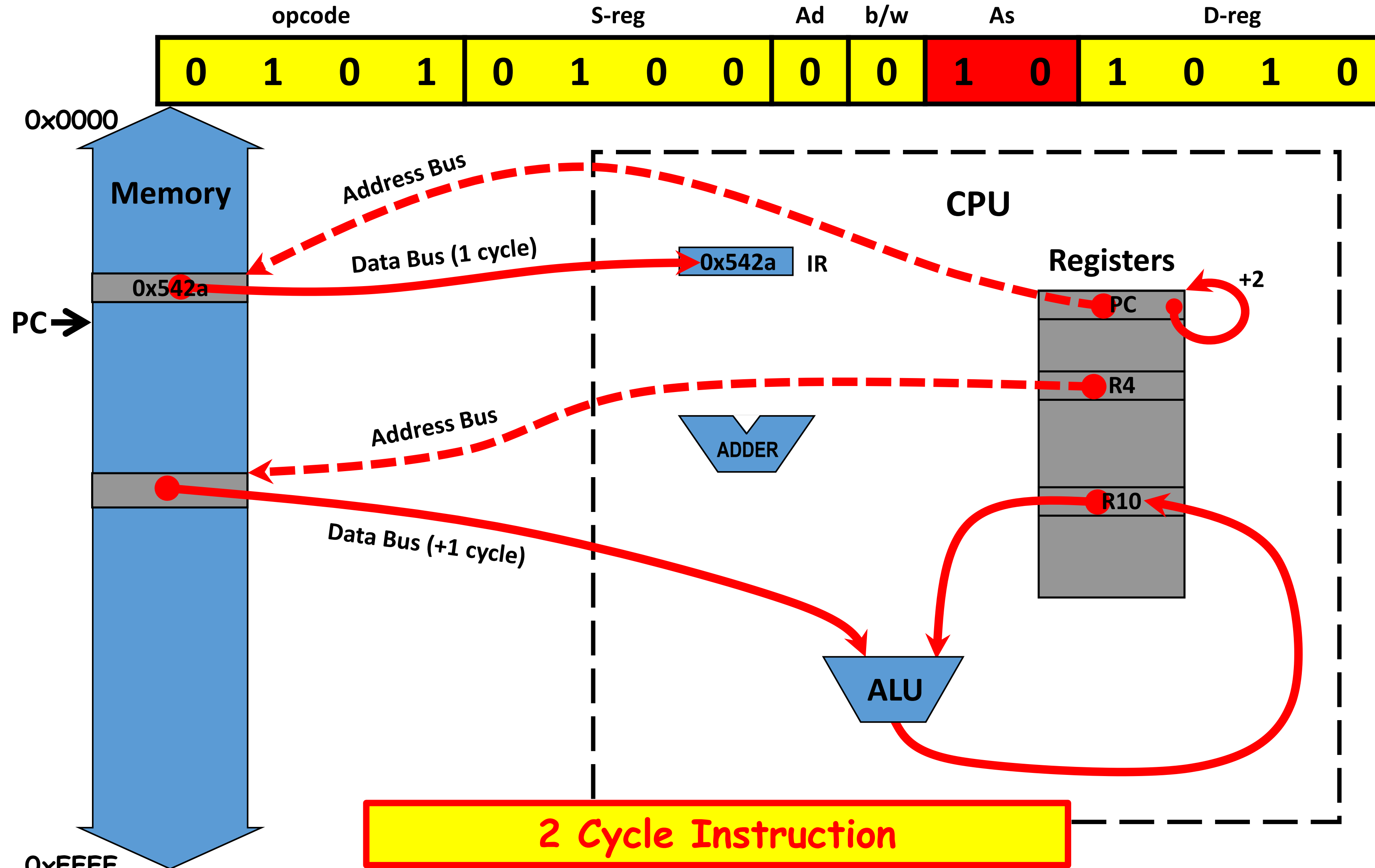
01 w/R2 = Absolute Mode



Indirect Register Mode

- mov.w @R5 ,R6 *; load word from address (R5)=4 into R6*
  - The address of the source is 4, the value in R5. Thus a word is loaded from address 4 into R6. The value in R5 is unchanged.
  - Indirect addressing cannot be used for the destination so indexed addressing must be used
    - **mov.w** R6 ,0(R5) *; store word from R6 into address 0+(R5)=4*

10 = Indirect Register Mode



2 Cycle Instruction

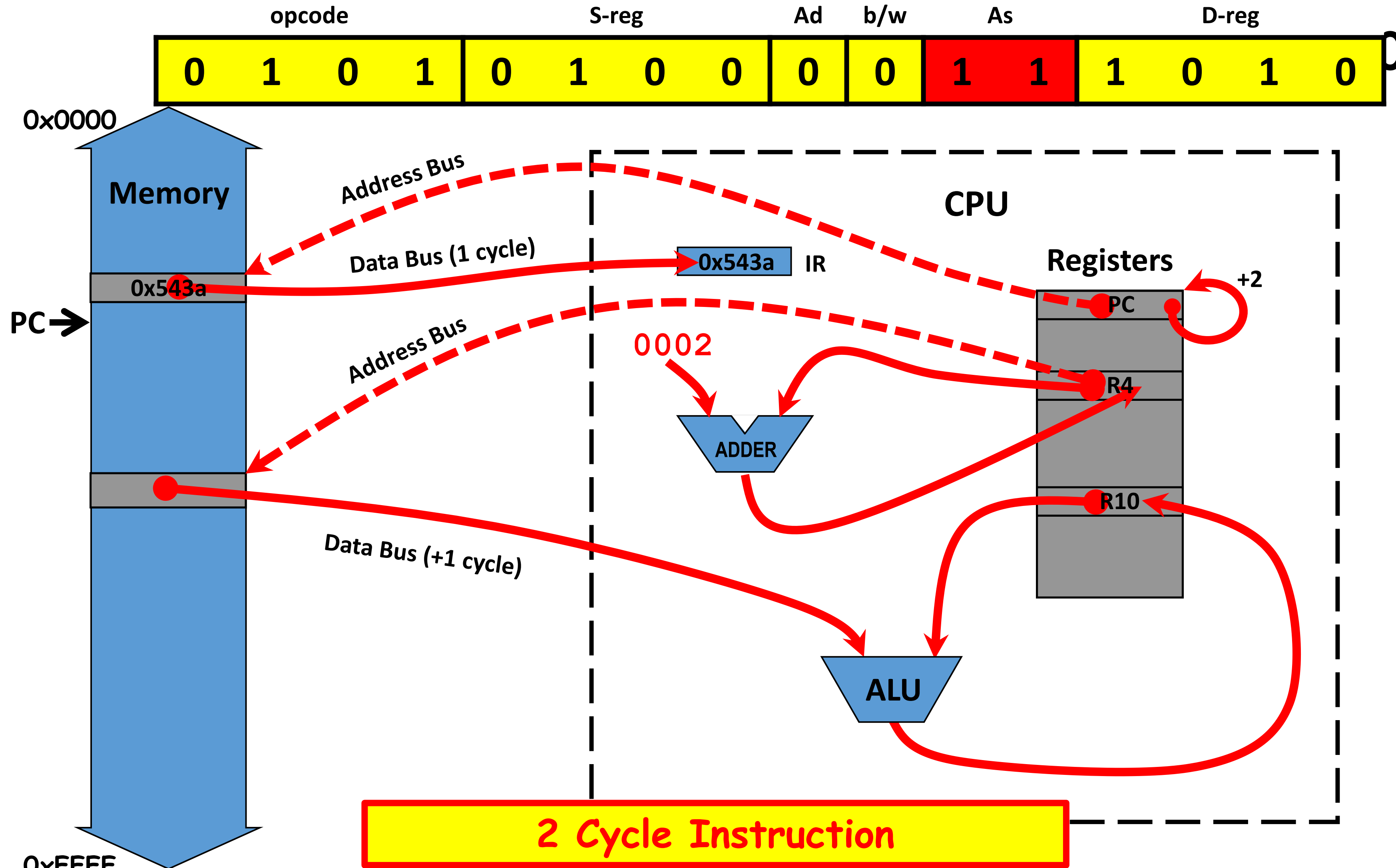
Indirect Autoincrement

- mov.w @PC+,R6 *; load immediate word into R6*
- PC is automatically incremented after the instruction is fetched and therefore points to the following word.
- The instruction loads this word into R6 and increments PC to point to the next word, which in this case is the next instruction. The overall effect is that the word that followed the original instruction has been loaded into R6.

Indirect Autoincrement Register Mode

- available only for the source
- mov.w @R5+,R6
- A word is loaded from address 4 into R6 and the value in R5 is incremented to 6 because a word (2 bytes) was fetched.
- Useful when stepping through an array or table, where expressions of the form *c++ are often used in C. Instead use:
  - **mov.w** R6 ,0(R5) *; store word from R6 into address 0+(R5)=4*
  - **incd.w** R5 *; R5 += 2*
  - For indirect register mode W(S) = 10.
  - For indirect autoincrement mode, W(S) = 11.

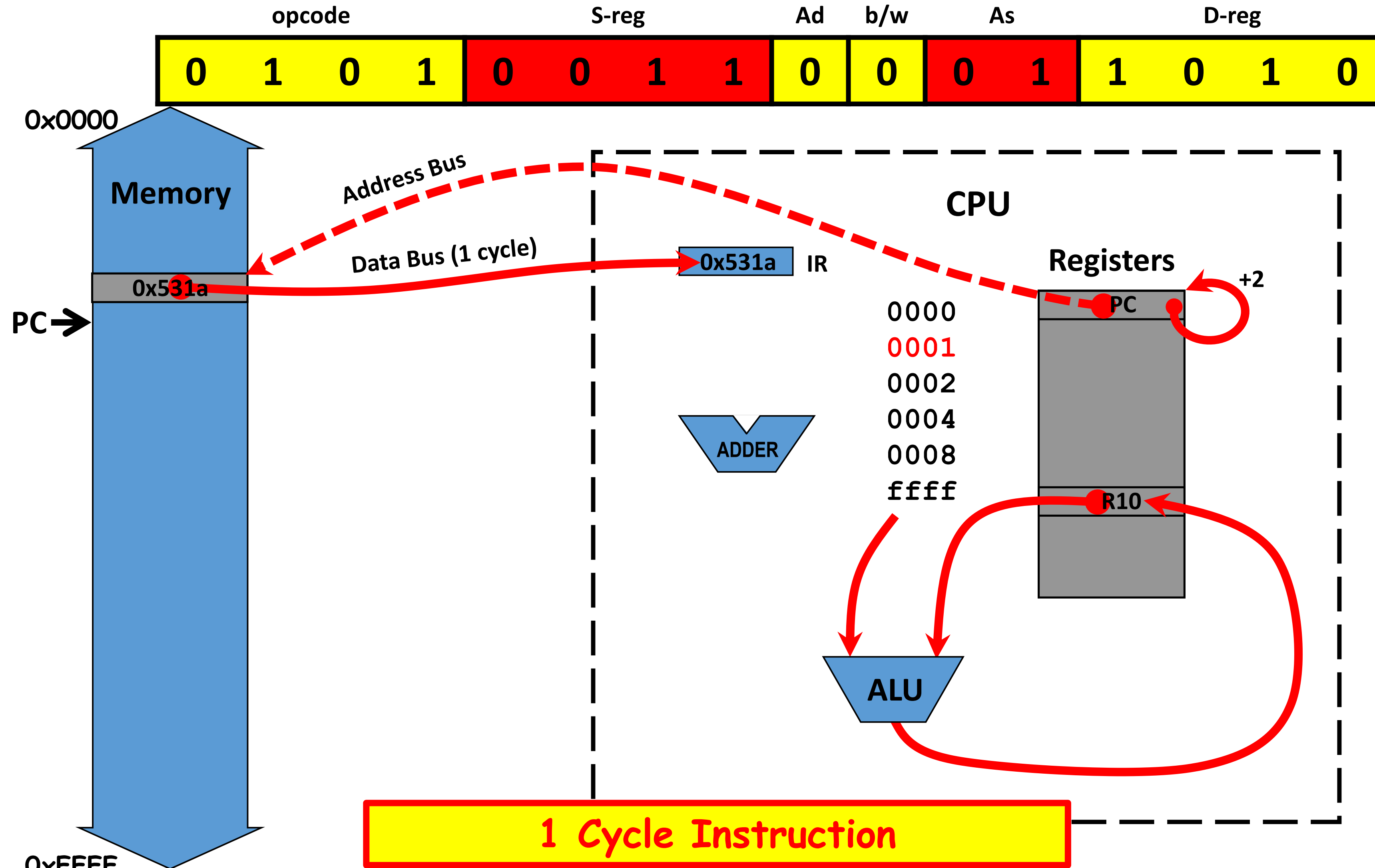
11 = Indirect Auto-increment Mode



Constant Generator

- To improve code efficiency, the MSP430 "hardwires" six register/addressing mode combinations to commonly used source values, eliminating the need to use a memory location for the immediate value:
  - #0 - R3 in register mode
  - #1 - R3 in indexed mode
  - #4 - R2 in indirect mode
  - #2 - R3 in indirect mode
  - #8 - R2 in indirect auto-increment mode
  - #-1 - R3 in indirect auto-increment mode

Constant Generator



1 Cycle Instruction

Addressing Modes (C, C++)

C, C++	Addressing Mode	Assembly
`register int x, y;` `y = x;`	Register	mov.w r4,r5
`int a, b, tab[100];` `a = tab[b];`	Indexed Register	mov.w tab(r4),r5
`int* cow = &tab[0];` `int cat = *cow;`	Indirect Register	mov.w @r6,r5
`int* cow = &tab[0];` `int cat = *cow++;`	Indirect Auto-increment	mov.w @r6+,r5
`int cat = 100;`	Immediate	mov.w #100,r5
`int cat = *(int*)100;`	Absolute	mov.w &100,r5
`extern int dog;` `int cat = dog;`	Symbolic	mov.w dog,r5

ADDRESSING MODE EXAMPLES

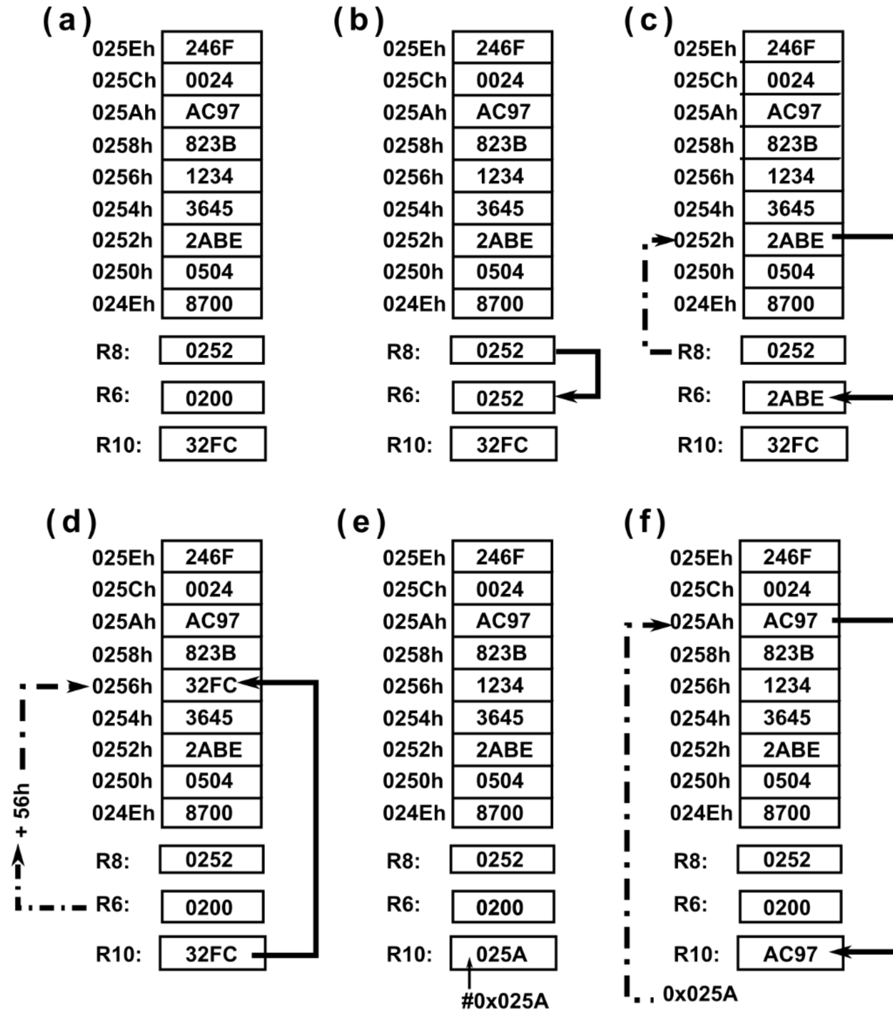


Fig. 3.33 Illustrating addressing modes. **a** Intial condition, **b** mov R8, R6, **c** mov @R8, R6, **d** mov R10, 56h(R6), **e** mov #0x025A, R10, **f** mov 0x025A, R10

Copyright 2009 Texas Instruments
All Rights Reserved

ADDRESSING MODE EXAMPLES

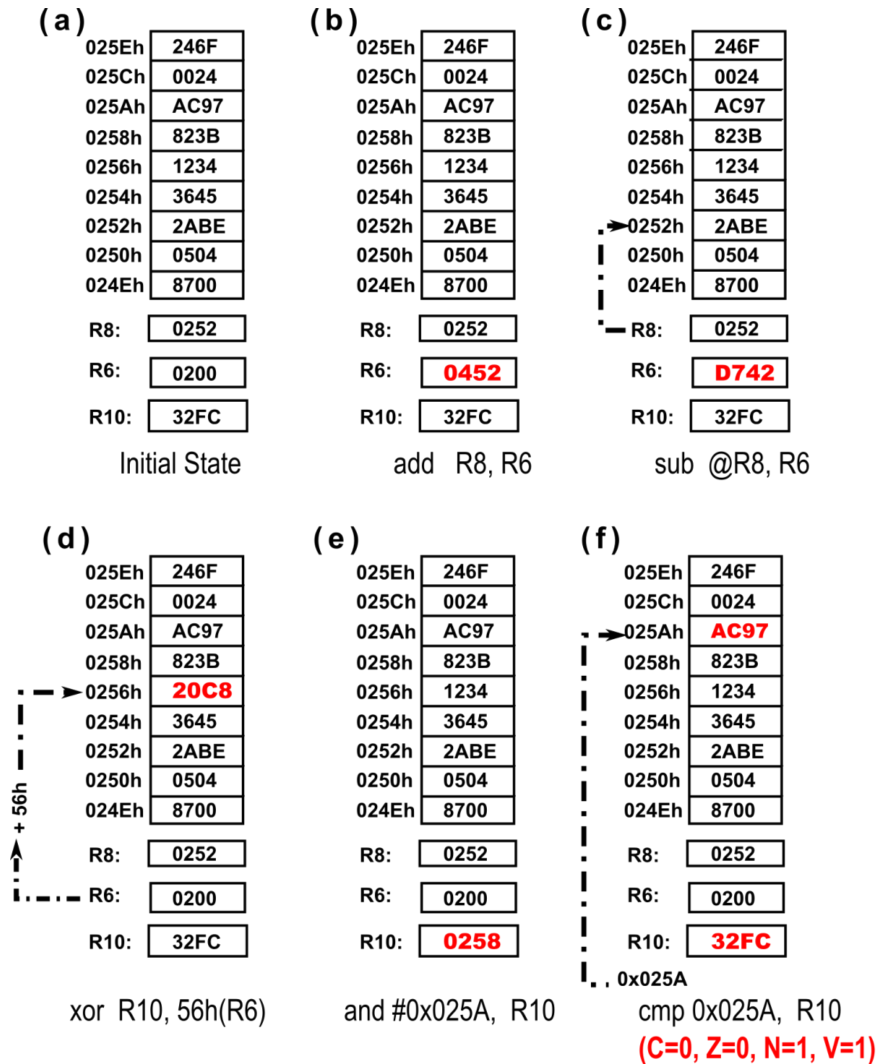


Fig. 3.34 Illustrating addressing modes with arithmetic-logic instructions

Copyright 2009 Texas Instruments
All Rights Reserved

Addressing Modes (C, C++)

C, C++	Addressing Mode	Assembly
`register int x, y;` `y = x;`	Register	`mov.w r4,r5`
`int a, b, tab[100];` `a = tab[b];`	Indexed Register	`mov.w tab(r4),r5`
`int* cow = &tab[0];` `int cat = *cow;`	Indirect Register	`mov.w @r6,r5`
`int* cow = &tab[0];` `int cat = *cow++;`	Indirect Auto-increment	`mov.w @r6+,r5`
`int cat = 100;`	Immediate	`mov.w #100,r5`
`int cat = *(int*)100;`	Absolute	`mov.w &100,r5`
`extern int dog;` `int cat = dog;`	Symbolic	`mov.w dog,r5`

Exercise 3.2

Match the C code on the left with the possible assembly code in the middle and the addressing mode on the right.

```
register int x, y;
extern int pig;
int cat, dog, table[100];
int* cow = &table[0];
```

1. y = x;
2. cat = table[dog];
3. cat = *cow;
4. dog = *cow++;
5. cat = 100;
6. cat = *(int*)100;
7. cat = pig;

a. mov.w @r8,r6
b. mov.w r4,r5
c. mov.w #100,r6
d. mov.w table(r7),r6
e. mov.w @r8+,r7
f. mov.w pig,r6
g. mov.w &100,r6

i. Absolute
ii. Indexed register
iii. Indirect auto-inc
iv. Indirect register
v. Immediate
vi. Register
vii. Symbolic

Core instructions

Table 4.4 Core MSP430 instructions

Type	Instruction	Description	V	N	Z	C
Data Transfer	`mov src,dest`	Loads destination with source	-	-	-	-[1]
	`push src`	Pushes source onto top of stack	-	-	-	-
	`swpb dest`	Swap bytes in destination word	-	-	-	-
Arithmetic	`add src,dest`	Adds source to destination	*	*	*	*
	`addc src,dest`	Adds source and carry to destination	*	*	*	*
	`sub src,dest`	Adds $\overline{source} + 1$ to destination (subtract source from destination)	*	*	*	*
	`subc src,dest`	Adds $\overline{source} + CF$ to destination (subtract with borrow)	*	*	*	*
	`dadd src,dest`	Adds source and carry to destination in Decimal (BCD) form[2]	*	*	*	*
	`cmp src,dest`	$dest - source$, but only affects flags[3]	*	*	*	*
	`sxt dest`	Sign extend LSB to 16-bit word	0	*	*	*

Core instructions

	`and src,dest`	"AND"'s source to destination bitwise	0	*	*	*
	`xor src,dest`	"XOR"'s source to destination bitwise	*	*	*	*
	`bit src,dest`	Like and, but only affects flags[4]	0	*	*	*
Logic and bit management	`bic src,dest`	Resets bits in destination	-	-	-	-
	`bis src,dest`	Sets bits in destination.	-	-	-	-
	`rra dest`	Roll bits to right arithmetically, i.e., $B_n \rightarrow B_(n-1) \ldots B_1 \rightarrow B_0 \rightarrow C$	0	*	*	*
	`rrc dest`	Roll destinations to right through Carry, $C \rightarrow B_n \rightarrow B_(n-1) \ldots B_1 \rightarrow B_0 \rightarrow C$	*	*	*	*
	`jz/jeq label`	Jump if zero/equal (Z = 1)	-	-	-	-
	`jnz/jne label`	Jump not zero/equal (Z = 0)	-	-	-	-
	`jc/jhe label`	Jump if carry (C = 1) – if higher or equal–- (≥, for unsigned numbers)	-	-	-	–
	`jnc/jlo label`	Jump if not carry (C = 0)– if lower,– - (<, for unsigned numbers)	-	-	-	–
Program Flow	`jn label`	Jump if negative (N = 1)	-	-	-	-
	`jge label`	Jump if $V = N$ (≥, for signed numbers)	-	-	-	-
	`jl label`	Jump if V≠N (if <, signed numbers)	-	-	-	-
	`jmp label`	Jump to label unconditionally	-	-	-	-
	`call dest`	Call subroutine at destination	-	-	-	-
	`reti`	Return from interrupt	-	-	-	-

[1]: For Flags: – means there is no effect; * there is an effect; "0", flag is reset.
[2]: Result is irrelevant if operands are not in format BCD
[3]: Used to compare numbers, usually followed by a conditional jump
[4]: Used to test if bits are set, usually followed by a conditional jump

Emulation

- The clear instruction clr.w or clr.b puts the value of the destination to 0. In many processors this is a distinct instruction but not in the MSP430: The assembler translates clr.b P2OUT to mov.b #0,P2OUT. You can see this in the Disassembly window of the debugger. This is an example of an emulated instruction.
- The program in assembly language writes to P2OUT before P2DIR, the opposite order from the program in C. This ensures that the correct values appear on the pins as soon as they are made into outputs.
- If the pins are switched to output first, the outputs initially are driven to the values that happen to be sitting in P2OUT
- It is perfectly legal to write to P2OUT while the pin is configured as an input: The value waits in a buffer until the pins are enabled for output.

Emulation

Table 4.5 Emulated instructions in the MSP430

Type	Instruction	Description	Core Inst.
Data Transfer	pop dest	Loads destination from TOS	mov @SP+,dest
	adc dest	Add carry to destination	addc #0,dest
	dadc src,dest	Decimal add Carry to destination	addc #0,dest
	dec dest	Decrement destination	sub #1,dest
Arithmetic	decd dest	Decrement destination twice	sub #2,dest
	inc dest	Increment destination	add #1,dest
	incd dest	Increment destination twice	add #2,dest
	sbc dest	Subtract Carry from destination	subc #0,dest
	tst dest	Test destination	cmp #0,dest
	inv dest	Invert bits in destination	xor #0FFFFh,dest
	rla dest	Roll (shift) bits to left	add dest,dest
Logic and bit Management	rlc dest	Roll bits left through carry	addc dest,dest
	clr dest	Clear destination	mov #0,dest
	clrc	Clear carry flag	bic #1,SR
	clrz	Clear zero flag	bic #2,SR
	clrn	Clear negative flag	bic #4,SR
	setc	Clear carry flag	bis #1,SR
	setz	Clear zero flag	bis #2,SR
	setn	Clear negative flag	bis #4,SR
	br dest	Branch to destination	mov dest,PC
Program Flow	dint	Disable interrupts	bic #8,SR
	eint	Enable interrupts	bis #8,SR
	nop	no operation	mov R3,R3
	ret	Return from subroutine	mov @SP+,PC

Word and Byte Instructions

Instruction(s)	Register notation	After
`mov.w R5,R6`	R6 ← R5	R5 = 03DA, R6 = 03DA
`mov.b R5,R6`	LSB(R6) ← LSB(R5) MSB(R6) ← 00h	R5 = 03DA, R6 = 00DA
`mov @R5,R6`	R6 ← (R5)	R5 = 03DA, R6 = 2B40, [03DA] = 2B40
`mov.b @R5,R6`	MSB(R6)←0, LSB(R6)←(R5)	R5 = 03DA, R6 = 0040 [03DA] = 2B40
`mov @R5,0(R6)`	(R6) ← (R5)	R5 = 03DA, R6 = 0226 [03DA] = 2B40, [0226] = 2B40
`mov.b @R5,1(R6)`	Byte(R6 + 1) ← Byte(R5)	R5 = 03DA, R6 = 0226 [03DA] = 2B40, [0226] = 405A
Sequence:		
`mov @R5+,R6`	R6 ← (R5); R5 ← R5 + 2	R5 = 03DC, R6 = 2*B*40, [03DA] = 2B40, [03DC] = 4580
`mov @R5+,R15`	R15 ← (R5) R5 ← R5 + 2	R5 = 03DE, R15 = 4580, [03DA] = 2B40, [03DC] = 4580
Sequence:		
`mov.b @R5+,R6`	MSB(R6)←0, LSB(R6)←(R5); R5 ← R5 + 1	R5 = 03DB, R6 = 0040 [03DA] = 2B40
`mov.b @R5+,R15`	MSB(R15)←0, LSB(R15)←(R5); R5 ← R5 + 1	R5 = 03DC, R15 = 002B [03DA] = 2B40
`mov.b R6,&0227`	(0227)←LSB(R6)	R6 = 0226, [0226] = 265A

R5 = 03DAh, R6 = 0226h, R15 = BAF4h, [03DAh] = 2B40h, [03DCh] = 4580h and [0226] = F35Ah.

Constant Generators

Operands in register mode requires less memory and faster processing times

- R3 for immediate values 0,1,2 and -1 (0xFFFF)
- R2 for immediate values 4 and 8 and absolute value 0

Non-constant generation		Constant generation	
Assembly Instruction	Machine language	Assembly Instruction	Machine language
`mov.w #0x300,SP`	4031 0300	`bis.b #001,& 0x0022`	D3D2 0022
`mov.w #0x5A8,& 0x0120`	40B2 5A80 0120	`xor.b #001,& 0x0021`	E3D2 0021
`mov.w #0xC350,R15`	403F C350	`sub.w #001,R15`	831F

Types of Instructions

- Data Transfer Instructions
  - Copy data from a source to a destination
- Arithmetic-logic Instructions
  - Perform arithmetic and/or logic operations on operands
- Program Control Instructions
  - Modify the default flow of execution in a program

Copyright 2009 Texas Instruments
All Rights Reserved

Table 4.6 Valid Operators in expressions listed by precedence order

Group	Operator	Meaning
1	+	Unary plus symbol
	−	Unary minus symbol
	~	1s complement
		Logical NOT
2	*	Multiplication
	/	Division
	%	Modulo
3	+	Addition
	−	Subtraction
4	<<	Shift left
	>>	Shift right
5	<	Less than
	<=	Less than or equal to
	>	Greater than
	>=	Greater than or equal to
6	=	Equal to
	!=	Not equal to
7	&	AND
	^	XOR
	‖	OR

Data Transfer Instructions

- Copy data from a source to a destination *destination* ← *source*
- Do not affect flags
- Included Instructions:
  - Data transfer: MOVE
  - Data exchange: SWAP
  - Stack manipulation: PUSH & POP
- Treat I/O locations like memory
  - Memory-mapped I/O
  - Examples:

```
MOV R8,R3 ; Copies the contents of R8 into R3
MOV (0xF348),R5 ; Copies into R5 the word at address F348h
PUSH R7 ; Copies onto the top of the stack the contents of R7
```

Copyright 2009 Texas Instruments
All Rights Reserved

Arithmetic Logic Ops

- Perform arithmetic and/or logic operations on data
  - *destination* ← (*DestinationOperand* ☐ *SourceOperand*)
- Flags affected according to operation result
- Included Instructions:
  - Arithmetic: ADD, SUB
  - Compare and test: CMP, TEST
  - Bitwise logic: AND, OR, XOR, NOT
  - Bit Displacement: SHIFT, ROTATE
  - Examples:

```
ADD R7,R5 ; Places on R5 the sum of the contents of R5 and R7
AND #05AD,R6 ; Places on R6 the bitwise result of anding the contents of R6 and the value 05ADh
ROTL R3 ; Rotates the contents of register R3 one position to the left
```

Copyright 2009 Texas Instruments
All Rights Reserved

Arithmetic Logic Ops

Table 4.7 Arithmetic instructions for the MSP430

Core Instructions			
Mnemonics & Operands	Description	Flags	Comments
`add src,dest`	dest ← src + dest	Normal	Add src to dest
`addc src,dest`	dest ← src + dest + C	Normal	Add with Carry
`dadd src,dest`	BCD algorithm used in dest ← src + dest + C	Special[a]	Decimal version of `addc`. Data in BCD format
`sub src,dest`	dest ← dest + `.NOT.`src + 1	Normal	Subtract src from dest. ($dest \leftarrow dest - src$)[b]
`subc src,dest` `sbb src,dest`	dest ← dest + `.NOT.`src + C	Normal	Subtract with borrow ($dest \leftarrow dest - src + C$)[b]
`cmp src,dest`	dest + `.NOT.`src + 1	Normal	Only affect flags.
`sxt dest`	MSB ← FFh × (Bit 7)	Special[c]	Word operand only. Signed LSB extended to 16 bit.

Emulated			
Mnemonics & Operands	Description	Emulated Instruction	Comments
`adc dest`	dest ← dest + carry	`addc #0,dest`	Add carry to dest.
`dadc dest`	BCD version for `adc`	`dadd #0,dest`	
`inc dest`	dest ← dest + 1	`add #1,dest`	
`incd dest`	dest ← dest + 2	`add #2,dest`	
`dec dest`	dest ← dest − 1	`sub #1,dest`	
`decd dest`	dest ← dest − 2	`sub #2,dest`	
`tst dest`	dest ← dest − 0	`cmp #0,dest`	To test for sign or zero

[a] C = 1 if result > 99 for bytes or result > 9999 for words; V is undefined

[b] Borrow needed if C = 0; Borrow = $\overline{Carry}$

[c] C = NOTZ, V = 0

Arithmetic Logic Ops

R5 = 35DA, R6 = EF26, R7 = 5469, R8 = 0268,
[0268] = 364A, [026A] = 2FD1, [03BC] = 1087

dummy

Instruction	Operation	Results	Flage C	Z	N	V
Addition:						
`add R5,R6` or						
`add.w R5,R6`	35DAh+EF26h=12500h	R6=2500	1	0	0	0
`add.b #0x26,R5`	26h + 0DAh = 100h	R5=0000	1	1	0	0
Decimal addition:						
A. Assuming carry C=0.						
`dadd.b #0x96,R6`	0 + 96 + 26 = 122	R6=0022	1	0	0	X
B. Assuming carry C=1.						
`dadd R7,&0x03BC` or						
`dadd.w R7,&0x03BC`	1+5469+1087 = 6557	[03BC]=6557	0	0	0	X
Instruction	Operation	Results	C	Z	N	V
Subtraction, which actually uses two's complement addition						
`sub 2(R8),R6` or						
`sub.w 2(R8),R6`	EF26h+D02Eh+1h = 19F55h	R6=9F55	1	0	1	0
`sub.b #67,R5`	0DAh + 0BCh+1h = 197h	R5=0097	1	0	1	0
Sign extention						
`sxt R5`	Bit7=1: MSB(R5)←FFh	R5=FFDA	1	0	1	0
`sxt R6`	Bit7=0: MSB(R6)←FFh	R6=0026	0	0	0	0
Compare						
`cmp R6,R7` or						
`cmp.w R6,R7`	5469h+10D9h+1 = 6543h	No change	0	0	0	0

Working with Bits

Bitwise operations work directly on bits

Table 3.4 Logic properties and applications

0.AND.X = 0; 1.AND.X = X	To **clear** specific bits in destination, the binary expression of the source has 0 at the bit positions to clear and 1 elsewhere (Fig. 3.25a)
0.OR.X = X; 1.OR.X = 1	To **set** specific bits in destination, the binary expression of the source has 1 at the bit positions to set and 0 elsewhere (Fig. 3.25b)
0.XOR.X = X; 1.XOR.X = $\overline{X}$	To **toggle or invert** specific bits in destination, the binary expression of the source has 1 at the bit positions to invert and 0 elsewhere (Fig. 3.25c)

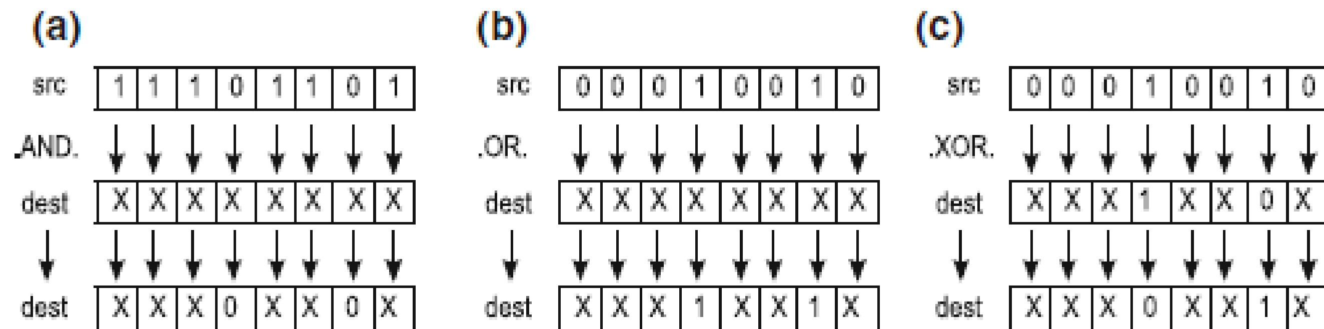


Fig. 3.25 Using logic properties to work with bits 1 and 5 only

Copyright 2009 Texas Instruments
All Rights Reserved

Logic and register control core instructions

Table 4.8 Logic and register control core instructions for the MSP430

Core Instructions			
Mnemonics & Operands	**Description**	**Flags**	**Comments**
`and src,dest`	dest ← src.AND.dest	Normal	Bitwise AND
`xor src,dest`	dest ← src.XOR.dest	See Note *	Bitwise XOR
`bic src,dest`	dest← (.NOT.src).AND.dest	Not affected	Clear bits in dest with mask src.
`bis src,dest`	dest ← src.OR.dest	Not affected	Set bits in dest with mask src.
`bit src,dest`	src.AND.dest	Normal	Test bits in dest with mask src. Only affects flags
`rra dest`	$b_n \rightarrow b_{n-1} \rightarrow \cdots$ $\cdots \rightarrow b_0 \rightarrow C$	C←LSB	Roll dest right arithmetically.
`rrc dest`	$C_{old} \rightarrow b_n \rightarrow \cdots$ $\cdots \rightarrow b_0 \rightarrow C_{new}$	C←LSB	Rotate dest right logically through C.

Emulated Instructions			
Mnemonics & Operands	**Description**	**Emulated Instruction**	**Comments**
`inv dest`	bit(h)←.NOT.bit(h)	`xor #0xFFFF,dest`	Inverts bits in dest.
`rla dest`	$C \leftarrow b_n \leftarrow \cdots$ $\cdots \leftarrow b_0 \leftarrow 0$	`add dest,dest`	Roll dest left.
`rlc dest`	$C_{new} \leftarrow b_n \leftarrow \cdots$ $\cdots \leftarrow b_0 \leftarrow C_{old}$	`addc dest,dest`	Rotate dest left through Carry.
`clr dest`	dest ← 0	`mov #0,dest`	Clears destination.
`clrc`	C ← 0	`bic #1,SR`	Clears Carry flag.
`clrn`	N ← 0	`bic #4,SR`	Clears Sign flag.
`clrz`	Z ← 0	`bic #2,SR`	Clears Zero flag.
`setc`	C ← 1	`bis #1,SR`	Sets Carry flag.
`setn`	N ← 1	`bis #4,SR`	Sets Sign flag.
`setz`	Z ← 1	`bis #2,SR`	Sets Zero flag.

C = NOT(Z), N reflects MSB, V = 1 if both operands are negative

Logic and register control core instructions

Example 4.8 *Assume contents of the registers and memory* **before** *any instruction as*

R12 = 25A3h = 0010010110100011, R15 = 8B94h = 1000101110010100,

[25A5h] = 6Ch = 01101100

(a) **AND** *and* **BIT TEST**

Instructions: `and R15,R12` *or* `and.w R15,R12` *and* `bit R15,R12` *or* `bit.w R15,R12`

Operation:
0010 0101 1010 0011 (R12) AND
1000 1011 1001 0100 (R15) =
0000 0001 1000 0000

Flags: C = 1Z = 0N = 0V = 0

`and R15,R12` *yields*
R12 = 0180 *but*
`bit R15,R12` *leaves* R12 *unchanged*

Instructions: `and.b 2(R12),R15` *and* `bit.b 2(R12),R15`

Operation:
0110 1100 (Memory) AND
1001 0100 (LowByteR15) =
0000 0100 (new Low Byte R15)

Flags: C = 1Z = 0N = 0V = 0

`and.b 2(R12),R15` *yields*
R15 = 0004, *but*
`bit.b 2(R12),R15` *leaves* R15 *unchanged*.

(b) **BIT CLEAR (BIC)**

Instruction: `bis R15,R12` *or* `bis.w R15,R12`

Operation:
0010 0101 1010 0011 (R12) AND
0111 0100 0110 1011 ($\overline{R15}$) =
0010 0100 0010 0011 (new R12)

New Contents: R12 = 2423
Flags: not affected

Instruction: `bic.b 2(R12),R15`

Operation:
1001 0011 ($\overline{Memory}$) AND
1001 0100 (LowByteR15) =
1001 0000 (new Low Byte R15)

New Contents: R15 = 0090
Flags: not affected

(c) **BIT SET (BIS)**

Instruction: `bis R15,R12` *or* `bis.w R15,R12`

Operation:
1000 1011 1001 0100 (R15) OR
0010 0101 1010 0011 (R12) =
1010 1111 1011 0111

New Contents: R12 = AFB7
Flags: not affected.

(d) **XOR**

Instruction: `xor.b #0x75,R15`

Operation:
0111 0101 (0x75) XOR
1001 0100 (LowByteR15) =
1110 0001

New Contents: R15 = 00E1
Flags: C = 1Z = 0N = 1V = 0

Copyright 2009 Texas Instruments
All Rights Reserved

Logic and register control core instructions

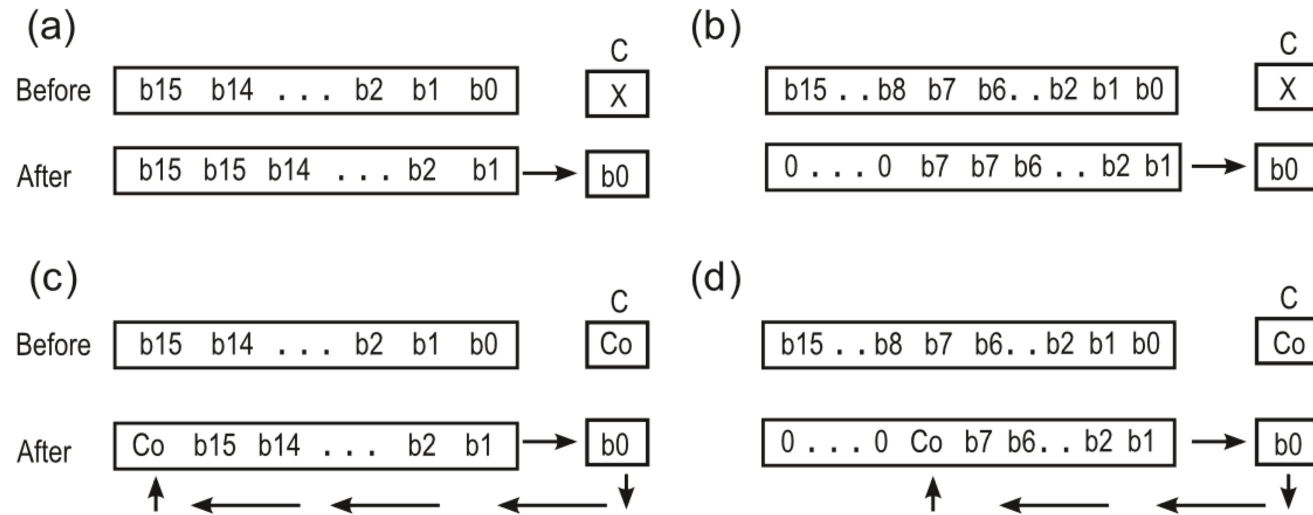


Fig. 4.15 Right arithmetic rolling (shifting): **a** `rra.w dest` and **b** `rra.b dest`; Right rotation through carry: **c** `rrc.w dest` and **b** `rrc.b dest`

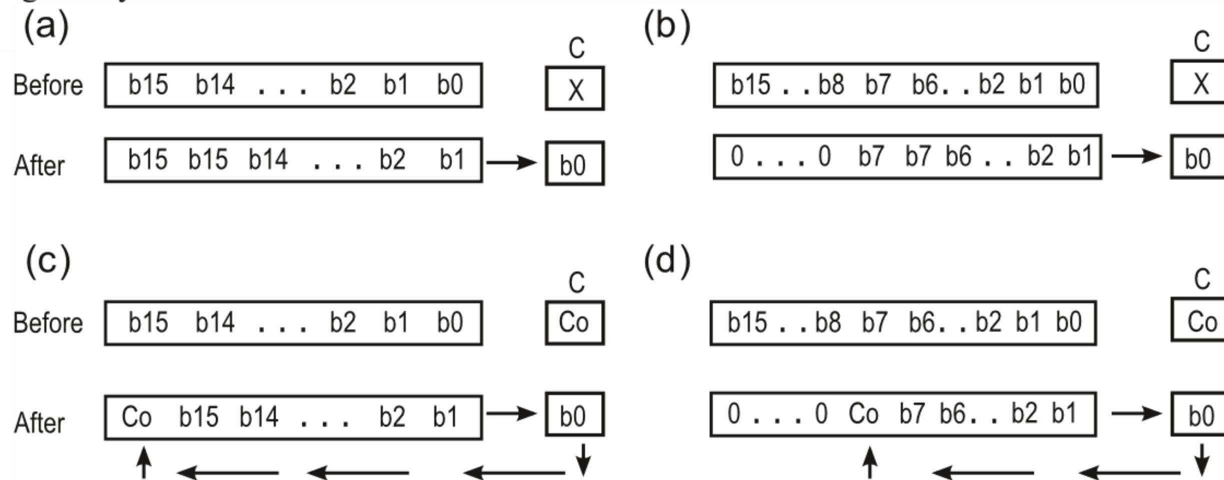


Fig. 4.15 Right arithmetic rolling (shifting): **a** `rra.w dest` and **b** `rra.b dest`; Right rotation through carry: **c** `rrc.w dest` and **b** `rrc.b dest`

Logic and register control core instructions

Example 4.9 *Each case below is independent of others. For all cases, contents of register* R5 *before instruction is:* R5 = 8EF5 = 1000 1110 1111 0101

(a) **Right shift/rotations**

Instruction: `rra R5` *or* `rra.w R5`

Operation: 1000 1110 1111 0101 $\xrightarrow{rra}$ 1100 0111 0111 1010 (LSB 1 ⇒ C)
New Contents: R5 = C77A *Flags:* C = 1Z = 0N = 1V = 0

Instruction: `rra.b R5`

Operation: 1000 1110 1111 0101 $\xrightarrow{rra}$ $\overbrace{00000000}^{Higher\,Byte}$ $\overbrace{11111010}^{rra\,here}$ (LSB 1 ⇒ C)
New Contents: R5 = 00FA *Flags:* C = 1Z = 0N = 1V = 0

Instructions: `clc` *followed by* `rrc R5` *or* `rrc.w R5`

Operation: C = 0*and then* 1000 1110 1111 0101 $\xrightarrow{rra}$ 0100 0111 0111 1010 1 ⇒ CF
New Contents: R5 = 477A *Flags:* C = 1 Z = 0 N = 0 V = 0

(b) **Left Shifts/Rotations**

Instruction: `rla R5` *or* `rla.w R5`, *equivalent to* `add R5,R5`.

*Operation:*1000 1110 1111 0101 *yields* C←1 0001 1101 1110 1010
New Contents: R5 = 1DEA *Flags:* C = 1 Z = 0 N = 0 V = 1

Instruction: `rla.b R5`, *equivalent to* `add.b R5,R5`

Operation: 1000 1110 1111 0101 ⇒ 0000 0000 1110 1010, C←1
New Contents: R5 = 00EA *Flags:* C = 1Z = 0N = 1V = 0

Instructions: `setc` *followed by* `rlc R5` *or* `rlc.w R5`, *equivalent to* `addc R5,R5`.

Operation: C = 1 *and then* 1000 1110 1111 0101 ⇒ 0001 1101 1110 1011 with 1 ⇒ C
New Contents: R5 = 1DEB *Flags:*C = 1Z = 0N = 0V = 1

PROGRAM CONTROL INSTR.

- Modify the default flow of execution in a program
  - *PC ← NewAddress*
- Do not affect flags
- Included Instructions:
- Unconditional Jump: Always change the PC
  - Conditional Jump: Change the PC if condition is true
  - Subroutine Calls and Returns: Transfer control from main to subroutines, returning to the calling point
- Examples:

```
JMP #F345h ; Loads PC with the address 0xF345 so program execution continues there
JZ #F345 ; Loads PC with the address 0xF345 if the Zero Flag is set
CALL Sub1 ; Saves PC onto the stack and loads PC with address Sub1. When special instruction
RET
```

Copyright 2009 Texas Instruments
All Rights Reserved

CONDITIONAL

Conditional Jump Instructions enable decision making in programs

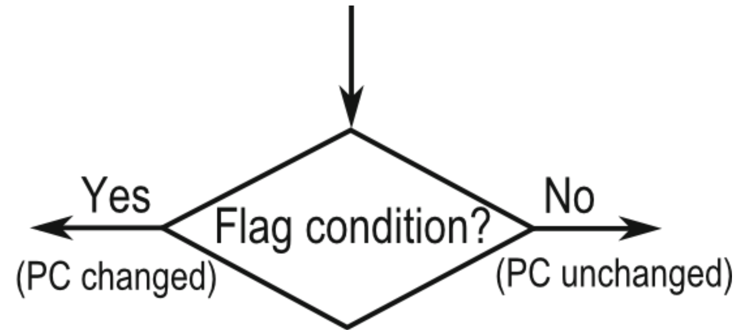


Fig. 3.26 Decision symbol associated to conditional jumps

Table 3.5 Conditional jumps

Mnemonics	Meaning	Mnemonics	Meaning
`jz`	Jump if zero (Z = 1)	`jn`	Jump if negative (N = 1)
`jnz`	Jump if not zero (Z = 0)	`jp`	Jump if positive (N = 0)
`jc`	Jump if carry (C = 1)	`jv`	Jump if overflow (V = 1)
`jnc`	Jump if no carry (C = 0)	`jnv`	Jump if not overflow (V = 0)

Copyright 2009 Texas Instruments
All Rights Reserved

CONDITIONAL

Table 4.9 Program flow instructions for the MSP430

Core Instructions		
Mnemonics & Operands	Description	Comments
`call dest`	Push PC and PC ← dest	Subroutine Call
`jmp label`	PC ← label	Unconditional jump (goto)
`jc label`	If C = 1, then PC ← label	"Jump if carry"
(or `jhs label`)		"Jump if higher than or same as"
`jnc label`	If C = 0, then PC ← label	"Jump if no carry"
(or `jlo label`)		"Jump if lower than"
`jge label`	If N = V, then PC ← label	"Jump if greater than or equal to"
`jl label`	If N ≠ V, then PC ← label	"Jump if less than"
`jn label`	If N = 1, then PC ← #label	"Jump if negative"
`jnz label`	If Z = 0, then PC ← label	"Jump if not zero"
(or `jne label`)		"Jump if not equal"
`jz label`	If Z = 1, then PC ← label	"Jump if zero"
(or `jeq label`)		"Jump if equal"
`reti`	Pops SR and then Pops PC	Return from interrupt

Emulated Instructions		
Mnemonics & Operands	Description	Emulated Instruction
`br dest`	Branch (go) to label	`mov dest,PC`
`dint`	Disable Interrupts (GIE = 0 in SR)	`bic #8, SR`
`eint`	Enable Interrupts (GIE = 1 in SR)	`bis #8, SR`
`nop`	No operation	`mov R3, R3`
`ret`	Return from subroutine	`mov @SP+, PC`

Copyright 2009 Texas Instruments
All Rights Reserved

CONDITIONAL

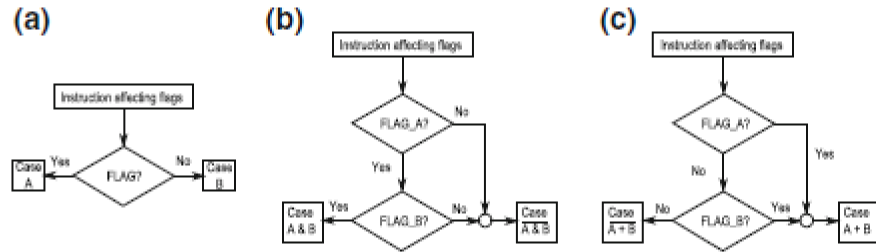


Fig. 4.18 Illustration of conditional jump operation: **a** Single Flag/condition; **b** AND-compound statement; **c** OR-compound statement

Table 4.10 Comparing A and B by A-B

Case	Unsigned numbers	Signed numbers
$A = B$	jeq	jeq
$A \neq B$	jne	jne
$A \geq B$	jhs	jge
$A < B$	jlo	jl

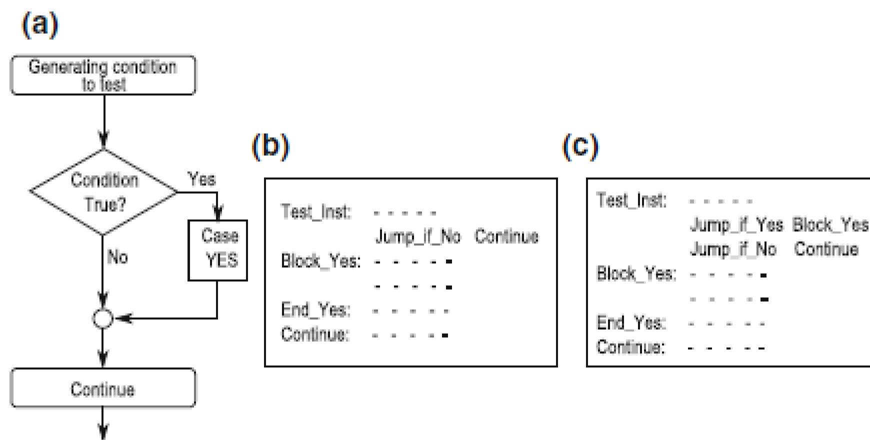


Fig. 4.19 IF-Structure **a** Flowchart, **b** and **c** Assembly code examples

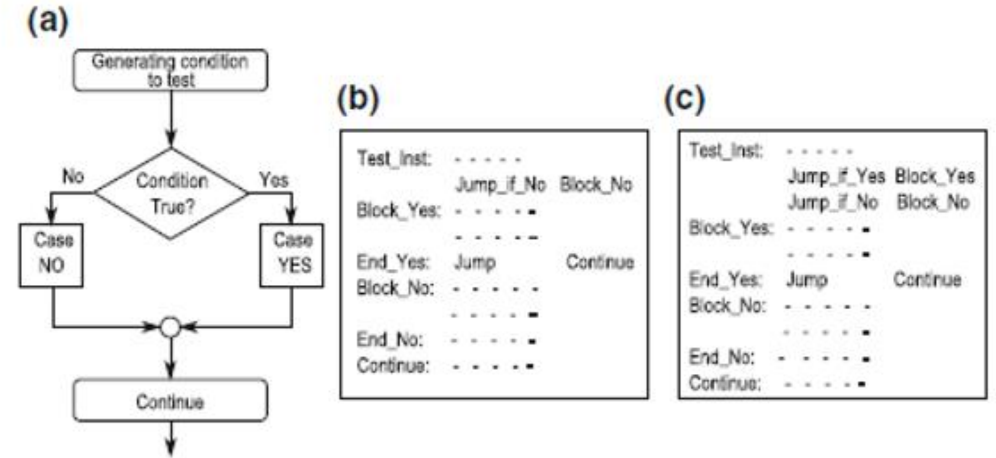


Fig. 4.20 IF-ELSE Structure **a** Flow chart, **b** and **c** Assembly code examples

The pseudo code in the left column may be programmed by the instructions of the right column:

LOOP

Correspondence between some flowcharts constructs and register transfer notation (RTN)

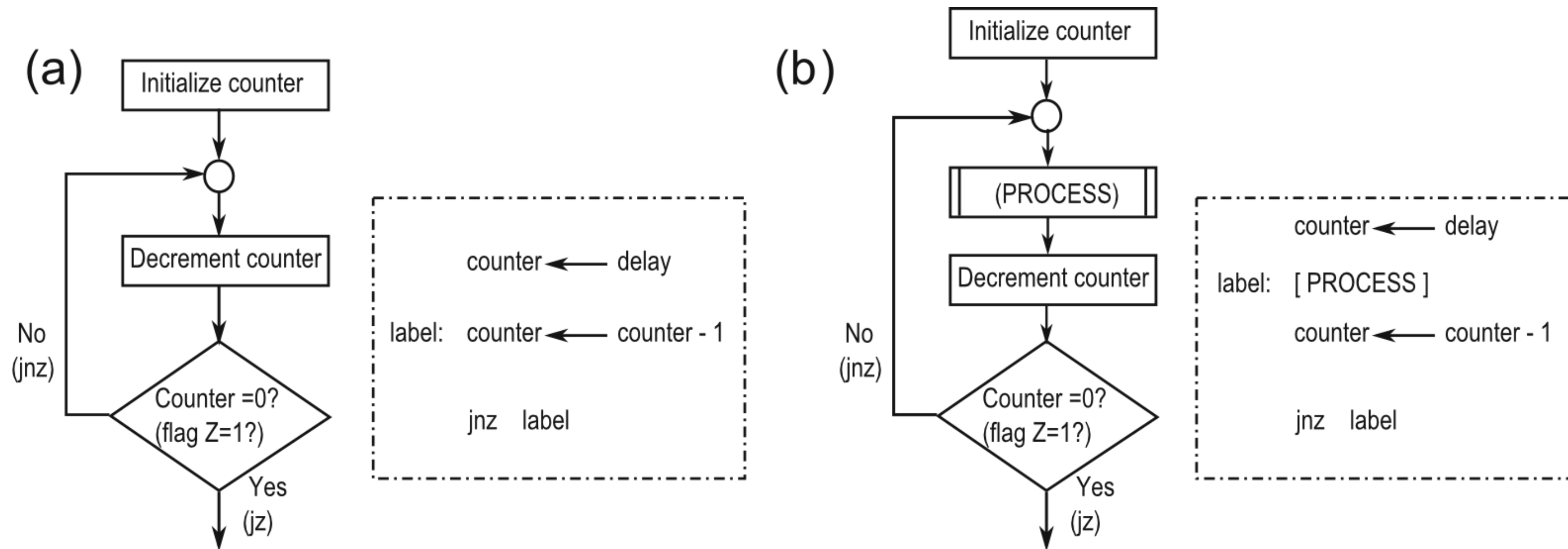


Fig. 3.27 Flow diagram and instruction skeleton associated to (**a**) delay loops and (**b**) iteration loops

LOOP

The pseudo code in the left column may be programmed by the instructions of th right column:

Step 1. Initialize
pointer = NUMBERS,
SUM = 0,
COUNTER = 2.
	`Step1: mov #NUMBERS,R8`
	`mov #0,R9`
	`mov #2,R10`

Step 2. Read number & increment pointer.	`Read: mov @R8+,R11`
Step 3. If number is not multiple of 4, go to step 2.	`Step3: tst #3,R11` `jnz Read`
Step 4. Add to SUM	`Step4: add R11,R9`
Step 5. Decrement COUNTER.	`Step5: dec R10`
Step 6. If COUNTER $\neq$ 0 go to step 2.	`Step6: jnz Read`
Step 7. Store Result.	`Step7: mov R9,&RESULT`

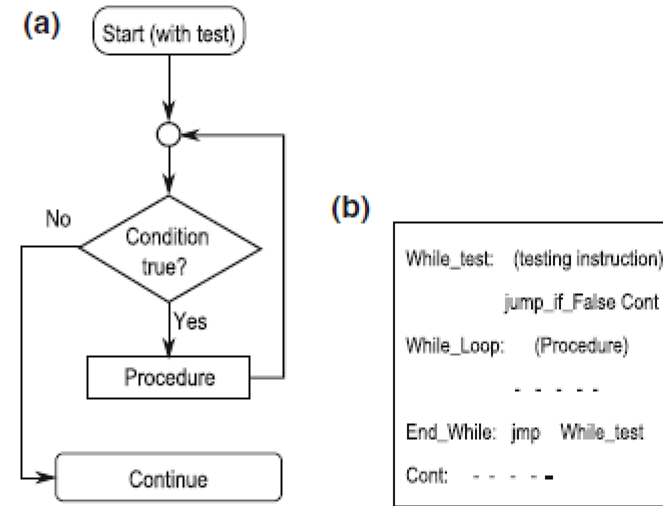


Fig. 4.22 WHILE-Loop structure: **a** Generic pseudo code; **b** Flowchart; **c** Assembly structure

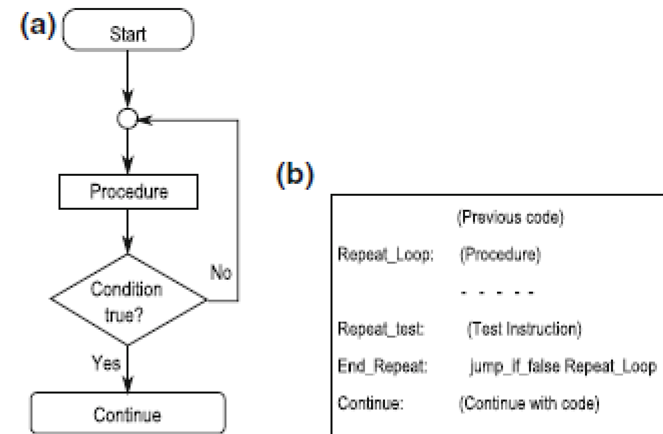


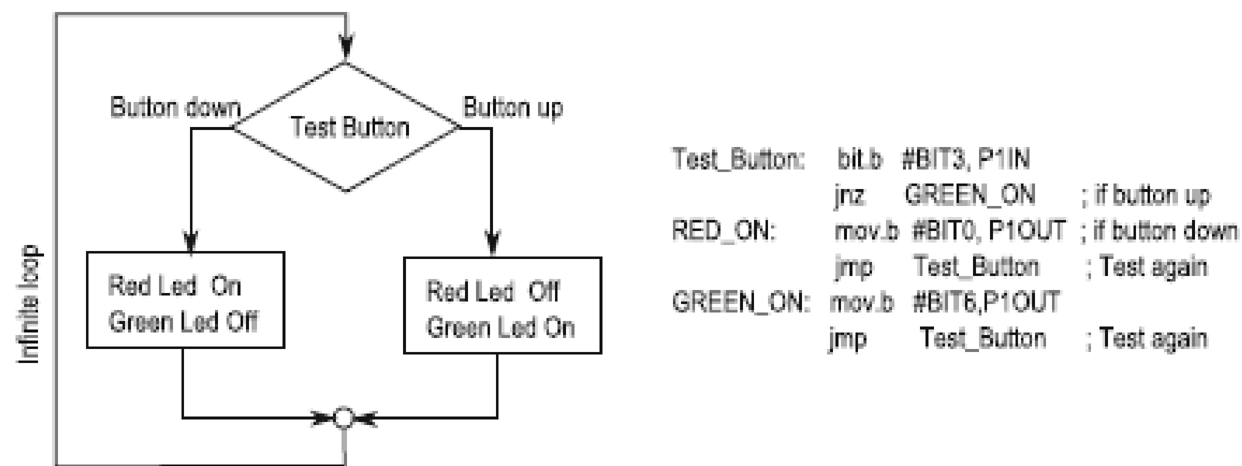
Fig. 4.23 REPEAT-UNTIL-Loop structure: **a** Generic pseudo code; **b** Flowchart; **c** Assembly structure

Copyright 2009 Texas Instruments
All Rights Reserved

LOOP

Example 4.14 A polling example

Let us illustrate polling with an example: a red LED and a green LED are driven by pins P1.0 and P1.6 of port 1, respectively. An LED is on if the output at the pin is high. A pushbutton is connected at pin P1.3, provoking a low voltage when down and a high voltage when up. The objective of the code is to turn on the red LED with the green LED off while the button is kept down, and conversely when it is up. Figure 4.24 illustrates the flow chart and code for the infinite loop of the main code. This is only part of the complete source program.




```
Test_Button:  bit.b  #BIT3, P1IN
              jnz    GREEN_ON      ; if button up
RED_ON:       mov.b  #BIT0, P1OUT  ; if button down
              jmp    Test_Button   ; Test again
GREEN_ON:     mov.b  #BIT6,P1OUT
              jmp    Test_Button   ; Test again
```

Fig. 4.24 Polling a button down with LEDs: flowchart and code

STACK

- A portion of memory used to temporarily store data
  - Access through special register Stack Pointer (SP)
  - Last-in-First-out (LIFO) operation
  - Stack contents is volatile
- Stack Operations
  - PUSH: Places data on top of the stack
  - POP (or pull) : Retrieves data from the top of the stack
  - Other instructions and events modifying the stack
- Invoking and returning from a subroutine call
- Responding and returning from an interrupt event

Copyright 2009 Texas Instruments
All Rights Reserved

STACK

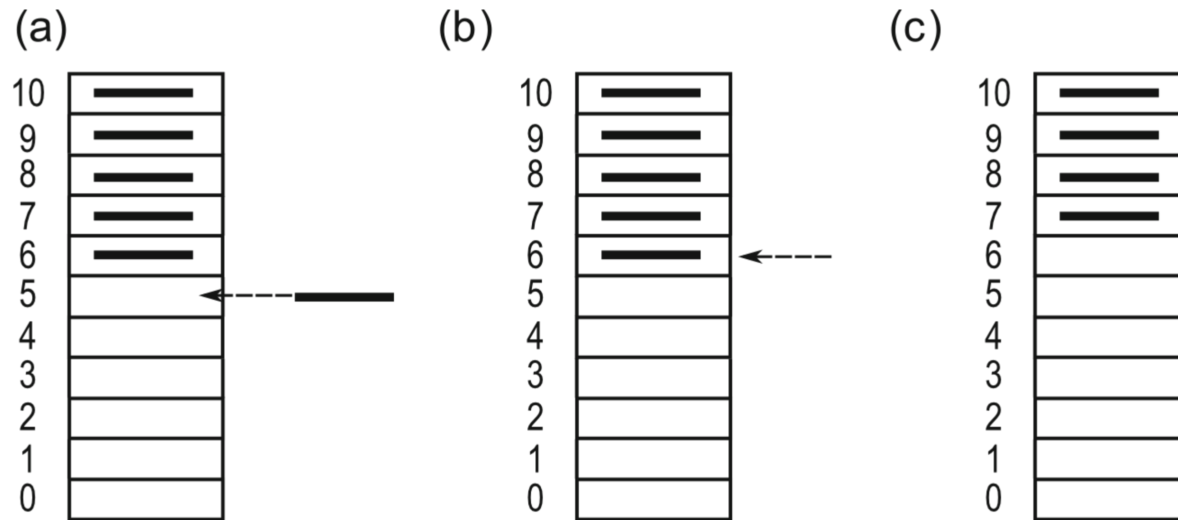


Fig. 3.28 Illustrating the stack operation. **a** TOS = 5 for pushing, **b** TOS = 6 for pulling, **c** After pulling, new TOS = 7

Copyright 2009 Texas Instruments
All Rights Reserved

STACK

(a) The stack is empty after the processor has been reset. The stack pointer should contain the address of the last word used, but there is none yet. It must therefore be initialized to the first address beyond the top of RAM, 0x0280 in the F2013. The hardware of the processor does *not* do this itself. For programs written in C, the compiler initializes the stack automatically as part of the startup code, which runs silently before the program starts, but *you must initialize SP yourself in assembly language*. This was shown in the section "Automatic Control: Use of Subroutines" on page 99.

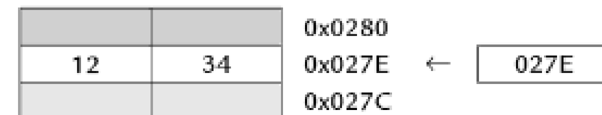
(b) The word 0x1234 has been added or *pushed* on to the stack. (The details of the instructions are explained later; recall that # means an immediate or literal value.) The value of SP is first decreased by 2 so that it points to the new location on the stack, then the value is copied to this address. This is called *predecrement addressing*.

(c) The byte 0x56 has been written to the stack. It goes into the lower byte of the next word, whose upper byte is wasted. A further word of 0x789A is written afterward and SP now points to this value.

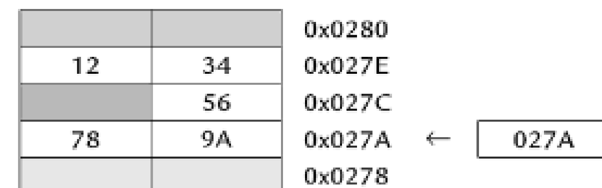
(d) A word has been removed, pulled or *popped* from the stack into the register R15. This retrieves the most recently added value, 0x789A, and copies it into R15. The stack pointer is increased by 2 afterward (postincrement addressing) and now points to the previously added value, the byte 0x56. Effectively the word 0x789A has been removed from the stack because it is now at a lower address than SP. The value remains in RAM until further values are pushed onto the stack, which grows downward and overwrites the old contents.

(a) Stack after initialization.

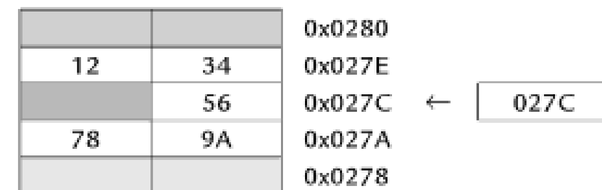
(b) Stack after `push.w #0x1234`.



(c) Stack after `push.b #0x56` followed by `push.w #0x789A`.



(d) Stack after `pop.w R15`.



STACK

- Push
  - Update the stack pointer to point to the new TOS
  - Copy the operand to the new TOS
- Pop or Pull
  - Copy the contents in the actual TOS to the destination
  - Update the stack pointer to point to the new TOS
  - Example: PUSH R9 and POP R9 (assume SP = 027Eh)

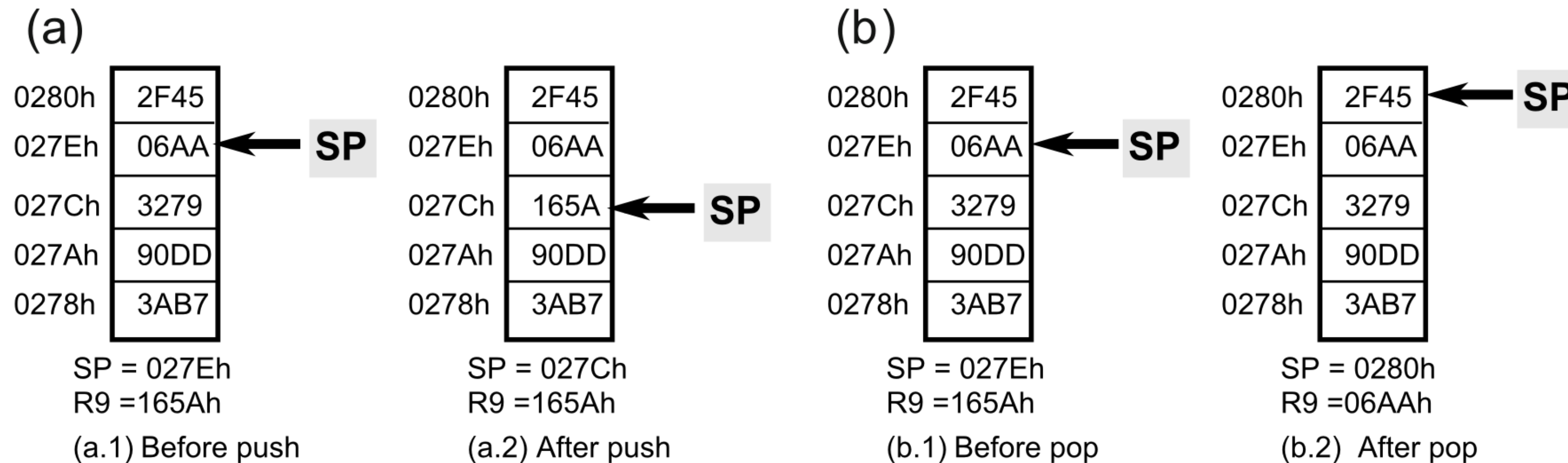


Fig. 3.29 The stack, SP register. **a** Push operation, **b** Pop operation

Copyright 2009 Texas Instruments
All Rights Reserved

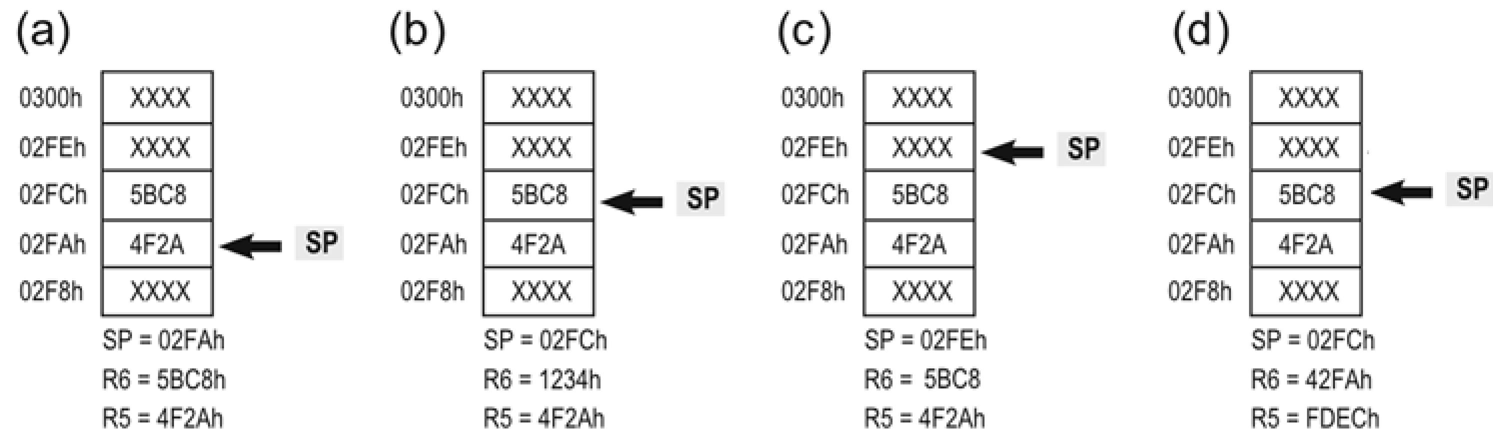


Fig. 4.13 Illustration of **push** and **pop** in Example 4.5. **a** After push operations, **b** Just before pop R6 in correct sequence, **c** After pop R6 in correct sequence, **d** After pop R6 in incorrect sequence

Correct Sequence:

```
FIRST:  push R6          ;save R6
        push R5
        mov #1234h,R6
        mov #0xFEDC,R5
        pop R5           ;recover R5
LAST:   pop  R6          ;recover R6
```

Incorrect Sequence:

```
FIRST:  push R6          ;save R6
        push R5          ;save R5
        mov #1234h,R6
        mov #0xFEDC,R5
LAST:   pop  R6          ;recover R6
```

SUBROUTINE

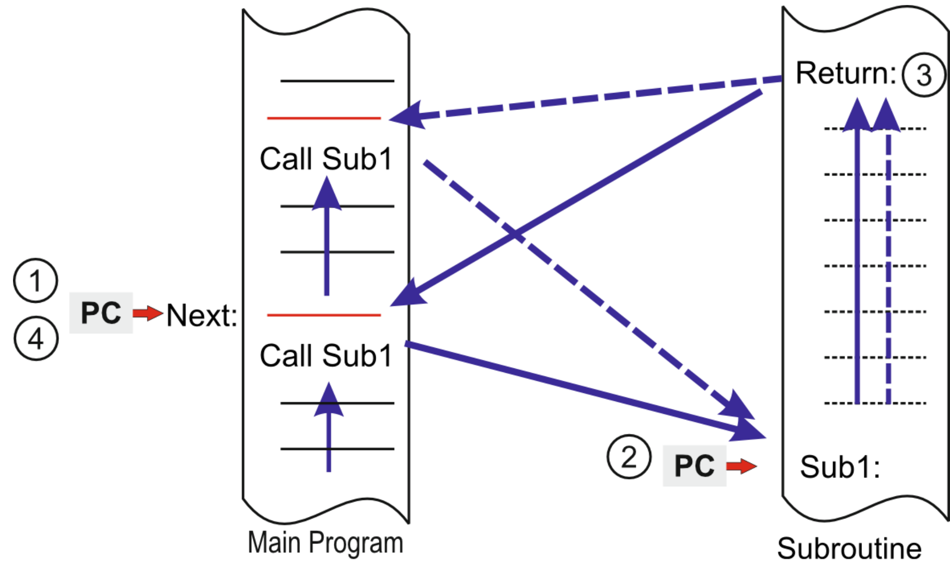


Fig. 3.31 Invoking a subroutine

1.Function Call
Saves PC onto stack
2.Function Execution

PC loaded with function address
3.Executing the Return

Restore the PC from the stack
4.Back at Main Program

Continue at instruction after “CALL”

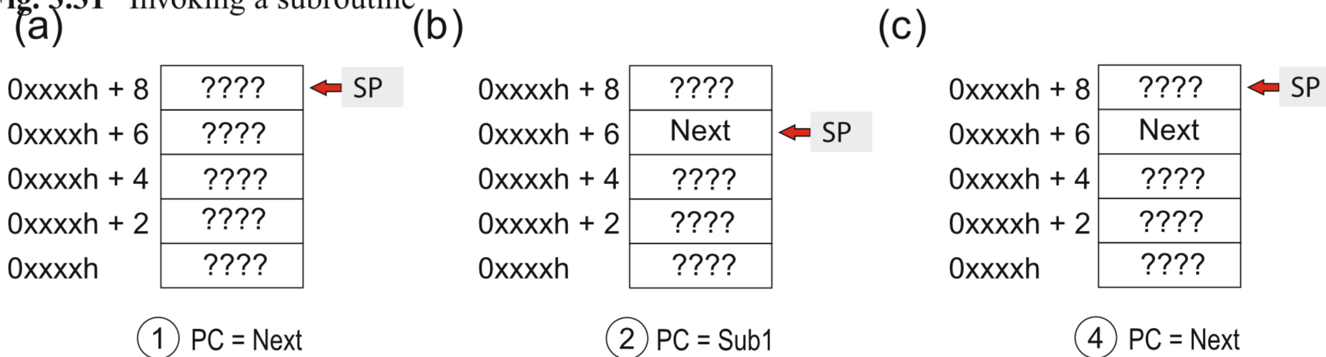


Fig. 3.32 Stack pointer use in the invocation and return of subroutines. **a** Stack before call, **b** Stack after call, **c** Stack after return

Copyright 2009 Texas Instruments
All Rights Reserved