

COMP-462

Embedded Systems

Lecture 2: I/O, Logic/Shift
Operations, Addressing modes,
Memory Operations, Subroutines,
Introduction to C

Agenda

□ Recap

- ❖ Embedded systems
- ❖ Product life cycle
- ❖ ARM programming

□ Outline

- ❖ Input/output
- ❖ Logical/shift operations
- ❖ Addressing modes, memory operations
- ❖ Stack and subroutines
- ❖ Introduction to C
 - o Structure of a C program
 - o Variables, expressions and assignments

ARM Assembly Language

□ Assembly format

| <i>Label</i> | <i>Opcode</i> | <i>Operands</i> | <i>Comment</i> |
|--------------|---------------|-----------------|-------------------------|
| init | MOV | R0, #100 | ; set table size |
| | BX | LR | |

□ Comments

- ❖ Comments should explain *why* or *how*
- ❖ Comments should *not* explain the opcode and its operands
- ❖ Comments are a major component of self-documenting code

Simple Addressing Modes

□ Second operand - $\langle op2 \rangle$

ADD Rd, Rn, $\langle op2 \rangle$

❖ Constant

○ **ADD Rd, Rn, #constant** ; **Rd = Rn+constant**

❖ Shift

○ **ADD R0, R1, LSL #4** ; **R0 = R0+(R1*16)**

○ **ADD R0, R1, R2, ASR #4** ; **R0 = R1+(R2/16)**

□ Memory accessed only with LDR STR

❖ Constant in ROM: **=Constant / [PC, #offs]**

❖ Variable on the stack: **[SP, #offs]**

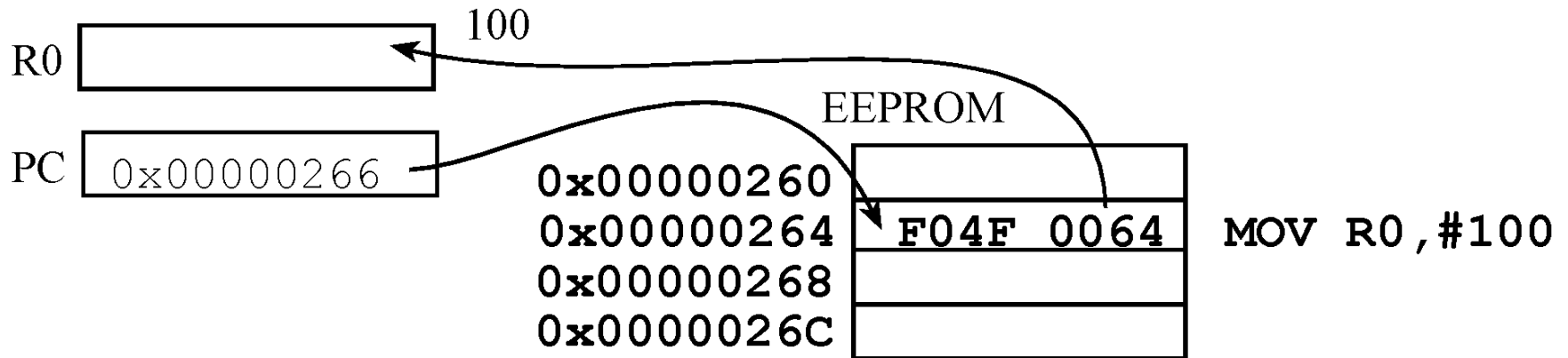
❖ Global variable in RAM: **[Rx]**

❖ I/O port: **[Rx]**

Addressing Modes

- Immediate addressing
 - ❖ Data is contained in the instruction

MOV R0, #100 ; R0=100, immediate addressing

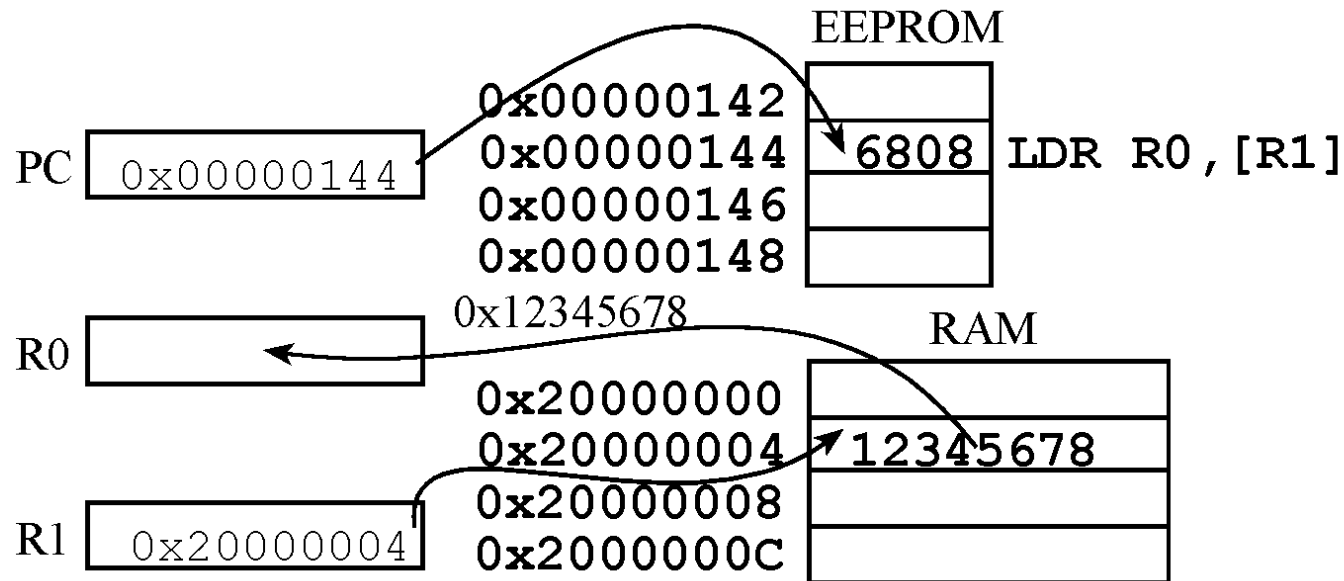


Addressing Modes

□ Indexed Addressing

- ❖ Address of the data in memory is in a register

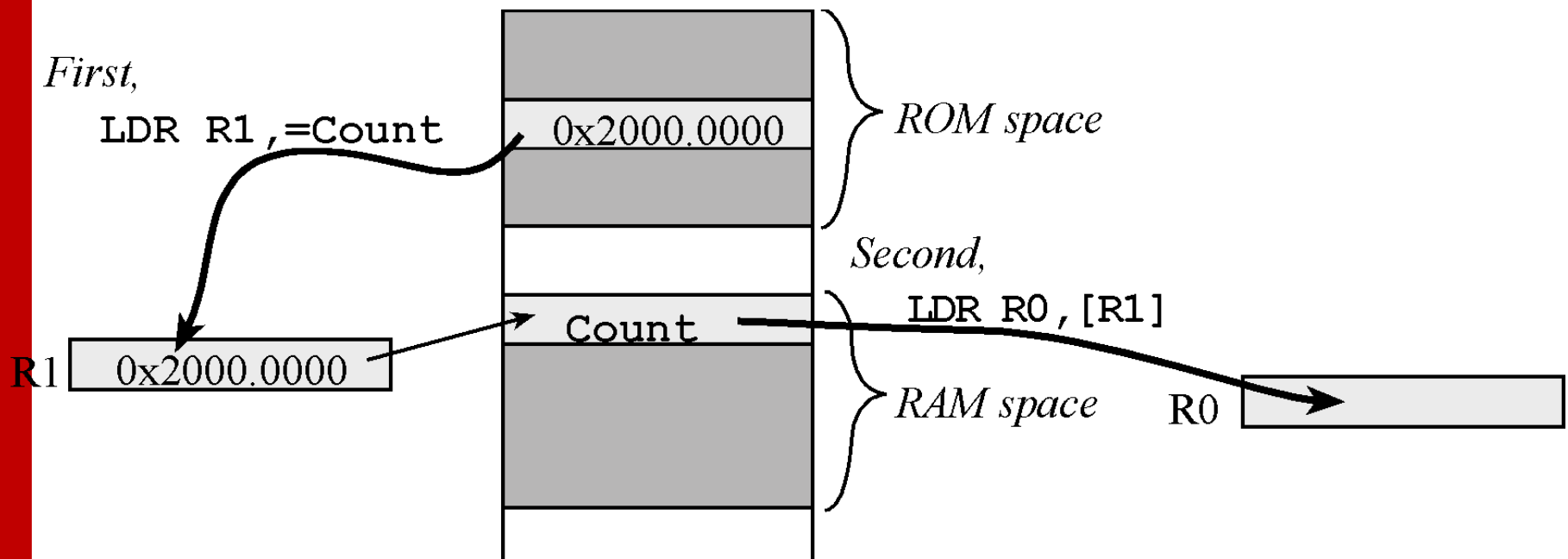
LDR R0, [R1] ; R0= value pointed to by R1



Addressing Modes

□ PC Relative Addressing

- ❖ Address of data in **EEPROM** is indexed based upon the Program Counter



Memory Access Instructions

❑ Loading a register with a constant, address, or data

❖ **LDR** **Rd, =number**

❖ **LDR** **Rd, =label**

❑ **LDR** and **STR** used to load/store RAM using register-indexed addressing

❖ **Register [R0]**

❖ **Base address plus offset [R0, #16]**

Load/Store Instructions

□ General load/store instruction format

LDR{type} Rd, [Rn] ;load memory at [Rn] to Rd

STR{type} Rt, [Rn] ;store Rt to memory at [Rn]

LDR{type} Rd, [Rn, #n] ;load memory at [Rn+n] to Rd

STR{type} Rt, [Rn, #n] ;store Rt to memory [Rn+n]

LDR{type} Rd, [Rn, Rm, LSL #n] ;load [Rn+Rm<<n] to Rd

STR{type} Rt, [Rn, Rm, LSL #n] ;store Rt to [Rn+Rm<<n]

| <i>{type}</i> | <i>Data type</i> | <i>Meaning</i> | |
|---------------|--------------------------|--|--------------------------------|
| | 32-bit word | 0 to 4,294,967,295 or -2,147,483,648 to +2,147,483,647 | |
| B | Unsigned 8-bit byte | 0 to 255, | Zero pad to 32 bits on load |
| SB | Signed 8-bit byte | -128 to +127, | Sign extend to 32 bits on load |
| H | Unsigned 16-bit halfword | 0 to 65535, | Zero pad to 32 bits on load |
| SH | Signed 16-bit halfword | -32768 to +32767, | Sign extend to 32 bits on load |
| D | 64-bit data | Uses two registers | |

Logic Operations

| A Rn | B Operand2 | A&B AND | A B ORR | A^B EOR | A&(~B) BIC | A (~B) ORN |
|---------|---------------|------------|------------|------------|---------------|---------------|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 |

□ Logic Instructions

AND{S} {Rd, } Rn, <op2> ; Rd=Rn&op2

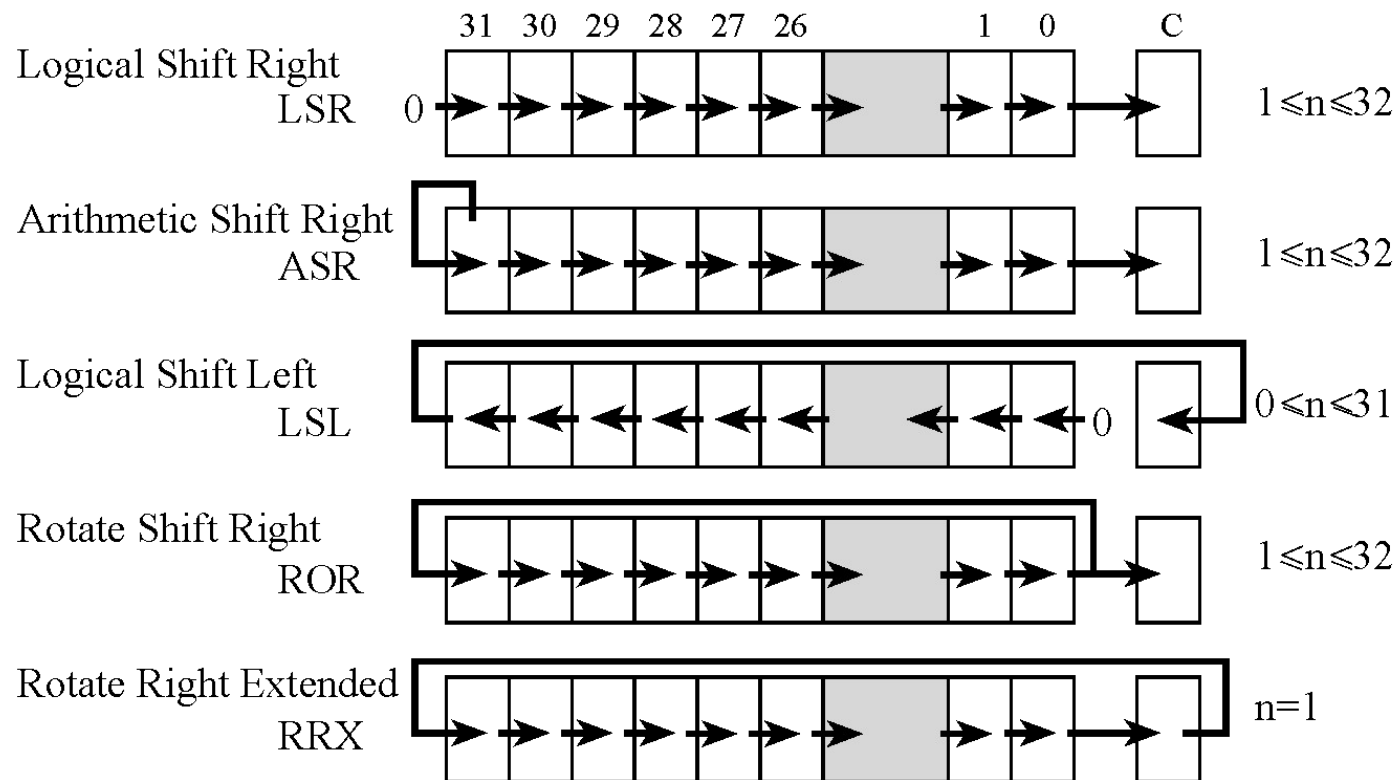
ORR{S} {Rd, } Rn, <op2> ; Rd=Rn|op2

EOR{S} {Rd, } Rn, <op2> ; Rd=Rn^op2

BIC{S} {Rd, } Rn, <op2> ; Rd=Rn&(~op2)

ORN{S} {Rd, } Rn, <op2> ; Rd=Rn|(~op2)

Shift Operations



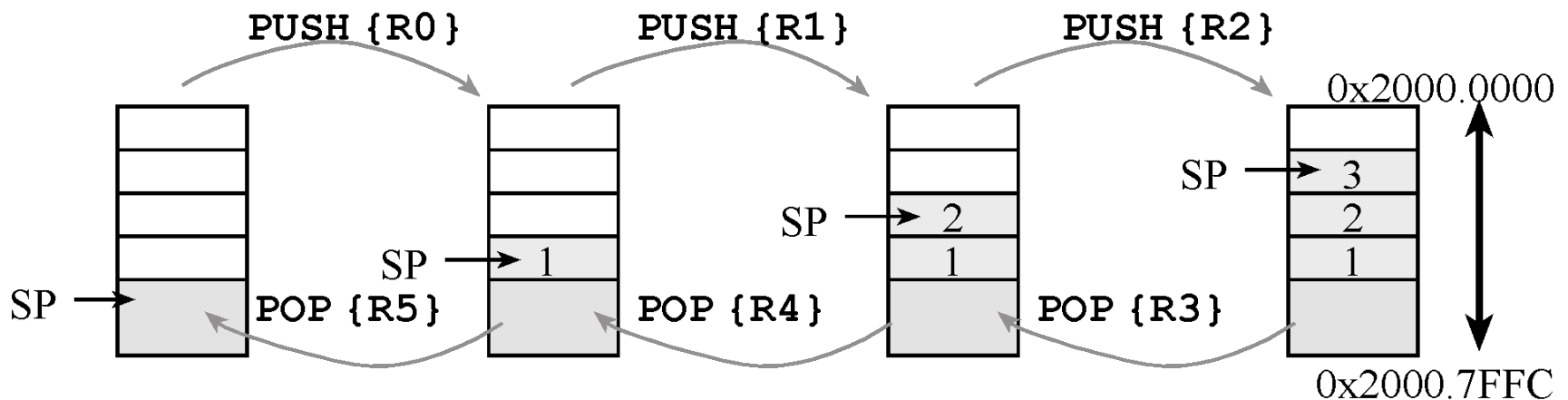
*Use the **ASR** instruction when manipulating signed numbers,
and use the **LSR** instruction when shifting unsigned numbers*

The Stack

- ❑ Stack is last-in-first-out (LIFO) storage
 - ❖ 32-bit data
- ❑ Stack pointer, SP or R13, points to top element of stack
- ❑ Stack pointer *decremented* as data placed on stack
- ❑ **PUSH** and **POP** instructions used to load and retrieve data

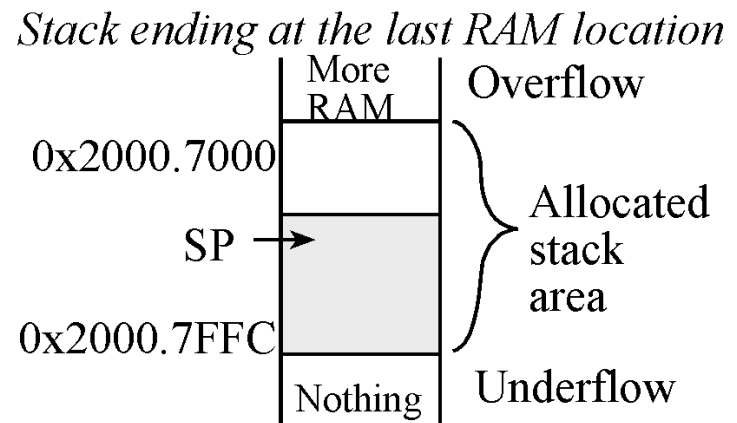
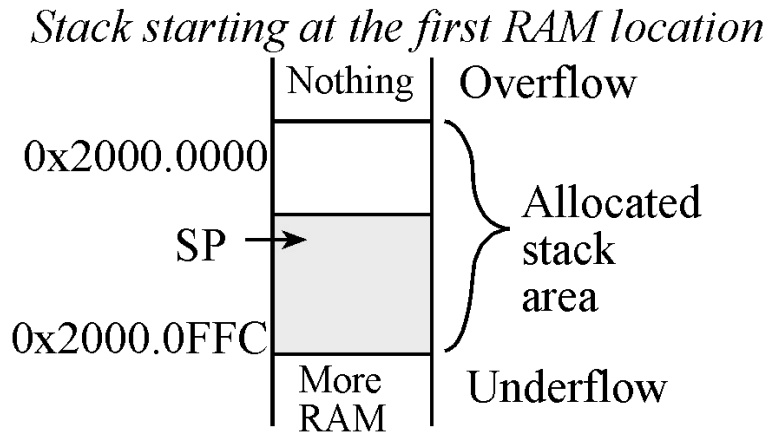
The Stack

- Stack is last-in-first-out (LIFO) storage
 - ❖ 32-bit data
- Stack pointer, SP or R13, points to top element of stack
- Stack pointer *decremented* as data placed on stack (*incremented* when data is removed)
- **PUSH** and **POP** instructions used to load and retrieve data



Stack Usage

□ Stack memory allocation



□ Rules for stack use

- ❖ Stack should always be balanced, i.e. functions should have an equal number of pushes and pops
- ❖ Stack accesses (push or pop) should not be performed outside the allocated area
- ❖ Stack reads and writes should not be performed within the free area

To set

The **or** operation to set bits 1 and 0 of a register.

The other six bits remain constant.

Friendly software modifies just the bits that need to be.

```
GPIO_PORTD_DIR_R |= 0x03; // PD1,PD0 outputs
```

Assembly:

```
LDR  R0,=GPIO_PORTD_DIR_R
LDR  R1,[R0]          ; read previous value
ORR  R1,R1,#0x03      ; set bits 0 and 1
STR  R1,[R0]          ; update
```

| | | | | | | | | |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|-------------------|
| c ₇ | c ₆ | c ₅ | c ₄ | c ₃ | c ₂ | c ₁ | c ₀ | value of R1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0x03 constant |
| c ₇ | c ₆ | c ₅ | c ₄ | c ₃ | c ₂ | 1 | 1 | result of the ORR |

To toggle

The **exclusive or** operation can also be used to toggle bits.

EXA-1: Write a program to toggle PD7. Assume all register configuration all done to make PD7 an output pin.

C program:

```
GPIO_PORTD_DATA_R ^= 0x80; /* toggle PD7 */
```

Assembly:

```
LDR  R0, =GPIO_PORTD_DATA_R
LDR  R1, [R0]          ; read port D
EOR  R1, R1, #0x80     ; toggle bit 7
STR  R1, [R0]          ; update
```

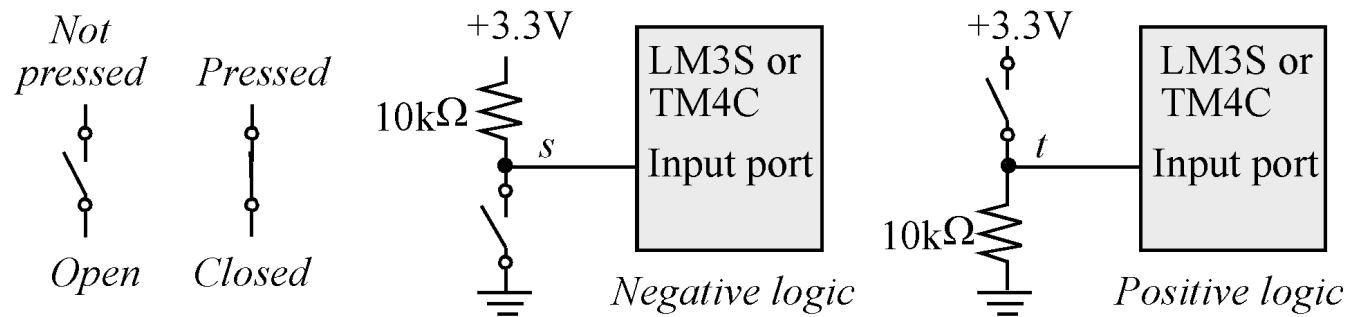
| | | | | | | | |
|------------|-------|-------|-------|-------|-------|-------|-------|
| b_7 | b_6 | b_5 | b_4 | b_3 | b_2 | b_1 | b_0 |
| <u>1</u> | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $\sim b_7$ | b_6 | b_5 | b_4 | b_3 | b_2 | b_1 | b_0 |

value of R1

0x80 constant

result of the EOR

Switch Interfacing



EXA-2: Write an assembly program to read a switch and store it in a variable in the memory called “Pressed”.

The **and** operation to extract, or *mask*, individual bits:

```
Pressed = GPIO_PORTA_DATA_R & 0x40;
//true if PA6 switch pressed
```

Assembly:

```
LDR  R0,=GPIO_PORTA_DATA_R
LDR  R1,[R0]      ; read port A
AND  R1,R1,#0x40  ; clear all bits except bit 6
LDR  R0,=Pressed  ; update variable
STR  R1,[R0]      ; true iff switch pressed
```

| | | | | | | | |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| a ₇ | a ₆ | a ₅ | a ₄ | a ₃ | a ₂ | a ₁ | a ₀ |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | a ₆ | 0 | 0 | 0 | 0 | 0 | 0 |

value of R1
 0x40 constant
 result of the AND

Shift Example

EXA-3: Write an assembly program to combine unsigned 4-bit High and Low component into 8-bit Result variable.

C program:

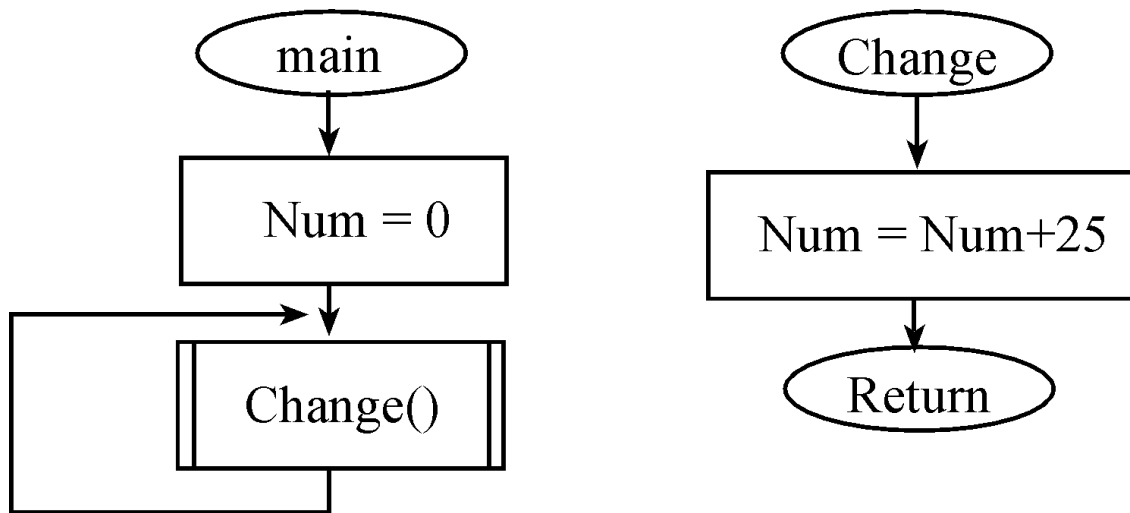
```
Result = (High<<4)|Low;
```

Assembly:

```
LDR  R0,=High
LDR  R1,[R0]    ; read value of High
LSL  R1,R1,#4    ; shift into position
LDR  R0,=Low
LDR  R2,[R0]    ; read value of Low
ORR  R1,R1,R2    ; combine the two parts
LDR  R0,=Result
STR  R1,[R0]    ; save the answer
```

| | | | | | | | | |
|-------|-------|-------|-------|-------|-------|-------|-------|--------------------------------------|
| 0 | 0 | 0 | 0 | h_3 | h_2 | h_1 | h_0 | value of High in R1 |
| h_3 | h_2 | h_1 | h_0 | 0 | 0 | 0 | 0 | after last LSL |
| 0 | 0 | 0 | 0 | l_3 | l_2 | l_1 | l_0 | value of Low in R2 |
| h_3 | h_2 | h_1 | h_0 | l_3 | l_2 | l_1 | l_0 | result of the ORR instruction |

Functions

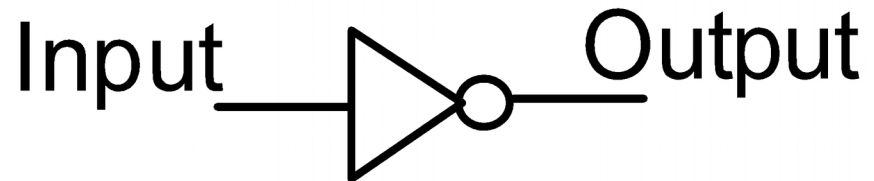


| | | | | |
|--------|-----|-----------|--------------------|--------------------|
| Change | LDR | R1,=Num | ; 5) R1 = &Num | unsigned long Num; |
| | LDR | R0,[R1] | ; 6) R0 = Num | void Change(void){ |
| | ADD | R0,R0,#25 | ; 7) R0 = Num+25 | Num = Num+25; |
| | STR | R0,[R1] | ; 8) Num = Num+25 | } |
| | BX | LR | ; 9) return | void main(void){ |
| main | LDR | R1,=Num | ; 1) R1 = &Num | Num = 0; |
| | MOV | R0,#0 | ; 2) R0 = 0 | while(1){ |
| | STR | R0,[R1] | ; 3) Num = 0 | Change(); |
| loop | BL | Change | ; 4) function call | } |
| | B | loop | ; 10) repeat | } |

Design example

EXA-4: Design a not-gate using the Tiva Launchpad. Steps are as follows:

- a) Overall function, specifications
- b) Data flow (test)
- c) Flowchart (test)
- d) Software (test)
- c) Simulation, prototype (test)
- e) Build (test)



Design

- ☐ Function table (PE0 input, PE1 output)
- ☐ How to test (switch input, LED output)
- ☐ Draw a data flow graph
- ☐ Draw a flowchart
- ☐ Write pseudo code
- ☐ Write assembly
- ☐ Simulate

Process

❑ Solution in NOTGate

Start Activate clock for port

Enable PE1, PE0 pins

Make PE0 (switch) pin an input

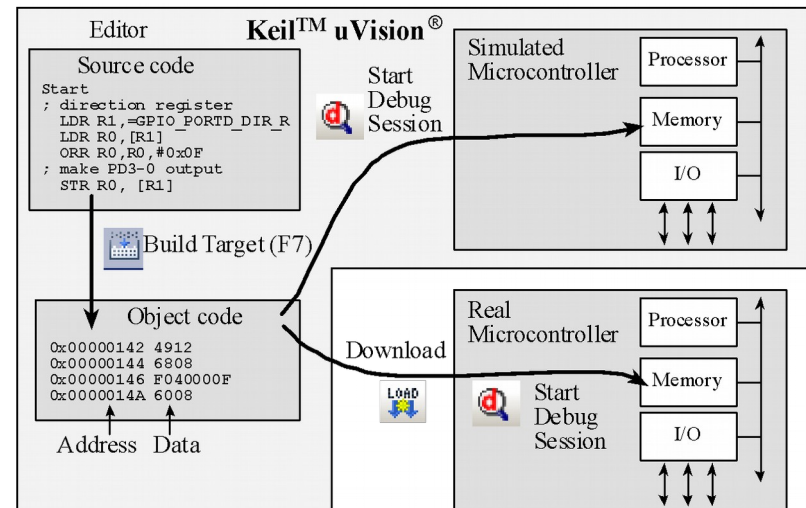
Make PE1 (LED) pin an output

Loop Read input from Switch (0 or 1)

PE1 = NOT(PE0)

Write output to LED

branch Loop



❑ Build it, run it, test it.


Subroutines

```
;-----Rand100-----  
; Return R0=a random number between  
; 1 and 100. Call Random and then divide  
; the generated number by 100  
; return the remainder+1
```

Rand100

```
    PUSH {LR}  ; SAVE Link  
    BL        Random  
;R0 is a 32-bit random number  
    LDR R1,=100  
    BL Divide  
    ADD    R0,R3,#1  
    POP {LR} ;Restore Link back  
    BX LR
```

POP {PC}



```
;-----Divide-----  
; find the unsigned quotient and remainder  
; Inputs:  dividend in R0  
;          divisor in R1  
; Outputs: quotient in R2  
;          remainder in R3  
;dividend = divisor*quotient + remainder
```

Divide

```
    UDIV R2,R0,R1  ;R2=R0/R1,R2 is quotient  
    MUL  R3,R2,R1  ;R3=(R0/R1)*R1  
    SUB  R3,R0,R3   ;R3=R0-R3  
                        ;R3 is remainder of R0/R1  
    BX   LR         ;return
```

```
ALIGN  
END
```

One function calls another, so LR must be saved

Reset, Subroutines and Stack

- ☐ A **Reset** occurs immediately after power is applied and when the reset signal is asserted (Reset button pressed)
- ☐ The Stack Pointer, SP (R13) is initialized at **Reset** to the 32-bit value at location 0 (Default: 0x20000408)
- ☐ The Program Counter, PC (R15) is initialized at **Reset** to the 32-bit value at location 4 (Reset Vector)
- ☐ The Link Register (R14) is initialized at Reset to 0xFFFFFFFF
- ☐ Thumb bit is set at *Reset*
- ☐ Processor automatically saves return address in LR when a subroutine call is invoked.
- ☐ User can push and pull multiple registers on or from the **Stack** at subroutine entry and before subroutine return.