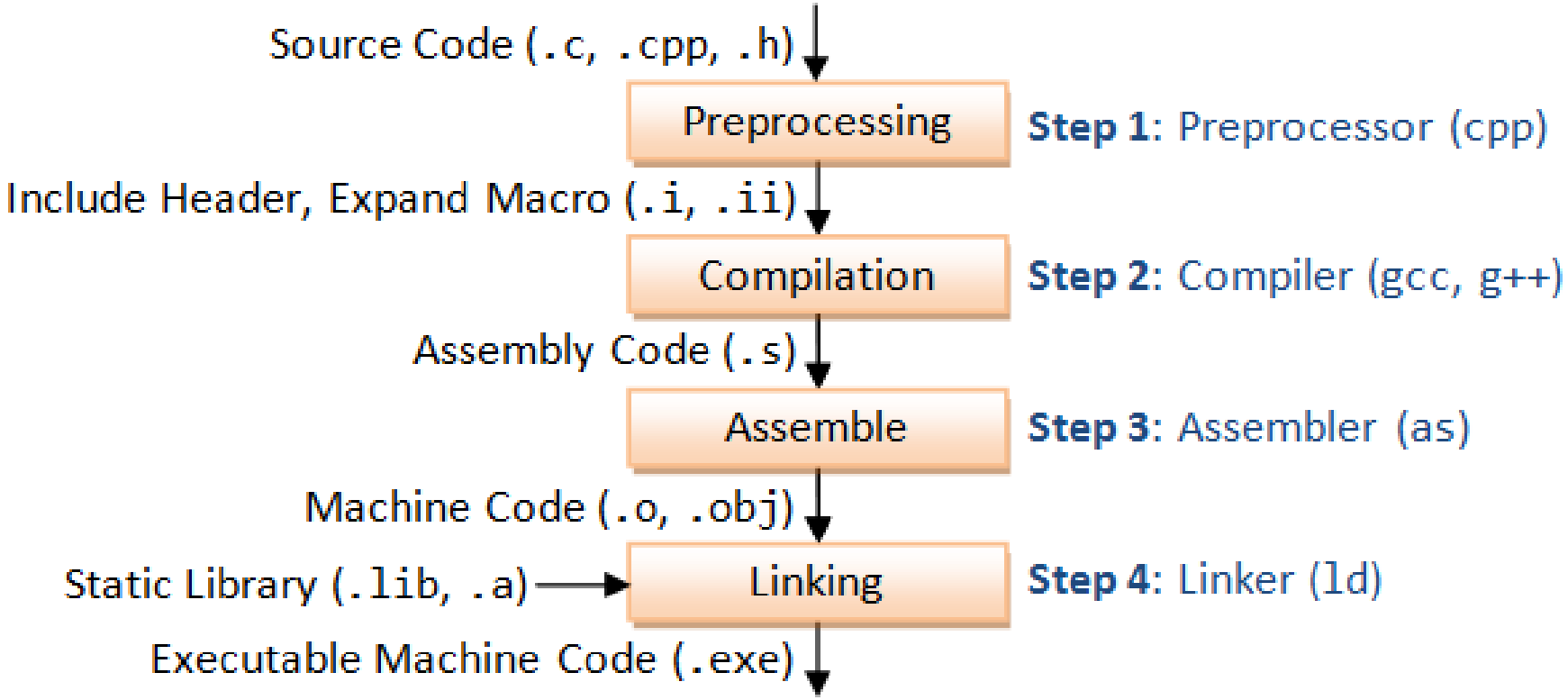
CSE 211: Introduction to Embedded Systems

Section 3

Contact Information

tasneem.awaad@eng.asu.edu.eg

Software Build Process



General Layout of Assembly

- {label} {instruction|directive|pseudo-instruction} {;comment}

Addressing Modes (Immediate Addressing)

- MOV
  - Format: **MOV Rn, Op2**
  - **Op2** can be register or #immediate
  - MOV R0,#0x25
- LDR Rn, =0x123

Addressing Modes

- Load/Store memory
  - Register indirect Addressing Mode
  - PC relative Addressing Mode
  - PUSH and POP Register Addressing Mode

Addressing Modes

- Load/Store memory
  - Register indirect Addressing Mode
    - **Regular Register indirect**
      - LDR R7, [R5]
      - R5 unchanged, R7 = Mem[R5]
    - **With Immediate Offset:**
      - LDR R7, [R5, #4]
      - R5 unchanged, R7 = Mem[R5 + 4]
    - **With Register Offset:**
      - LDR R7, [R5, R4]
      - R5 Unchanged, R7 = Mem[R5 + R4]

Addressing Modes

- Load/Store memory
  - Register indirect Addressing Mode
    - **With Pre indexed Immediate Offset:**
      - LDR R7, [R5, #4]!
      - R5 = R5 + 4
      - R7 = Mem[R5]
    - **With Post indexed Immediate Offset:**
      - LDR R7, [R5], #4
      - R7 = Mem[R5]
      - R5 = R5 + 4
    - **With Shifted Register Offset:**
      - LDR R7, [R5, R4, LSL #2]
      - R5 Unchanged, R7 = Mem[R5 + R4<<2]

PC Relative Addressing

- PC-relative: An addressing mode where the effective address is calculated by its position relative to the current value of the program counter.
- PC-relative addressing is indexed addressing mode using the PC as the pointer.
    - LDR Rn, =1234567 to move any 32-bit value into a register
    - BL{cond} label ; branch to subroutine at label
    - It is used for branching, for calling functions
    - B Location ; jump to Location, using PC-relative addressing
    - LDR R1,[PC,#28]

Bitwise Operations

- Apply to any “integral” data type like char, short, int and long.
- Arguments are treated as bit vectors.
- Operations applied bitwise ->

Operator	Symbol
AND	&
OR	|
XOR	^
One’s complement	~
Shift left	<<
Shift right	>>

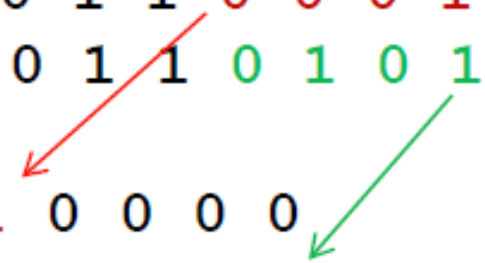
Bitwise Operations

```
C = A & B;
(AND)

C = A | B;
(OR)

C = A ^ B;
(XOR)

B = ~A;
(COMPLEMENT)
```

```
A   0 1 1 0 0 1 1 0
B   1 0 1 1 0 0 1 1
C   0 0 1 0 0 0 1 0

A   0 1 1 0 0 1 0 0
B   0 0 0 1 0 0 0 0
C   0 1 1 1 0 1 0 0

A   0 1 1 0 0 1 0 0
B   1 0 1 1 0 0 1 1
C   1 1 0 1 0 1 1 1

A   0 1 1 0 0 1 0 0
B   1 0 0 1 1 0 1 1
```

Bitwise Operations

```
B = A << 3;                A   1 0 1 0 1 1 0 1
(Left shift 3 bits)        B   0 1 1 0 1 0 0 0

B = A >> 2;                A   1 0 1 1 0 1 0 1
(Right shift 2 bits)       B   0 0 1 0 1 1 0 1

B = '1';                   B = 0 0 1 1 0 0 0 1 (ASCII 0x31)
C = '5';                   C = 0 0 1 1 0 1 0 1 (ASCII 0x35)
D = (B << 4) | (C & 0x0F);
        (B << 4)           = 0 0 0 1 0 0 0 0
      (C & 0x0F)           = 0 0 0 0 0 1 0 1
             D             = 0 0 0 1 0 1 0 1 (Packed BCD 0x15)
```



Set Bit (OR with 1)

- unsigned char X=0x07;
- X=X | (1 << 2);

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	0	0	0	0	1 / 0	1	1

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	0	0	0	0	1	0	0

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	0	0	0	0	1	1	1

Clear Bit (AND with Zero)

- unsigned char Y=0x07;
- Y=Y & ~ (1 << 2);

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
Y	0	0	0	0	0	1 / 0	1	1

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
~ (1 << 2)	1	1	1	1	1	0	1	1

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	0	0	0	0	0	1	1

Sheet 3

- Embedded systems always require the user to manipulate bits in registers or variables. Given an integer variable a, write two code fragments in C. The first should set bit 3 of a. The second should clear bit 3 of a. In both cases, the remaining bits should be unmodified.

Answer

First fragment:

```
a |= 1<<3;
```

Second fragment:

```
a &= ~ 1<<3;
```

Sheet 3

- Develop a sequence of instructions that sets the rightmost four bits of R3, clears the leftmost three bits of R3, and inverts bit positions 7,8 and 9 of R3. Assuming that R3 is 16-bit register.

Answer

- ORR R3, R3, #0x000F
- AND R3, R3, #0x1FFF
- EOR R3, R3, #0x0380

Sheet 3

- When does the LR have to be pushed on the stack?

Answer

- When a function calls another function, the LR is saved in the stack to avoid losing the return address of the caller function.

Sheet 3

- Show the SP value and the content of stack after executing this instruction PUSH {R4, R6-R8} assuming the SP initially equals 0x2000.1000 and R4=1, R6=2, R7=3, R8=4. What will be the values of the registers R0-R4 after executing this instruction POP{R0-R3}?

Answer

0x20000FF0	1
0x20000FF4	2
0x20000FF8	3
0x20000FFC	4

R0=1, R1=2, R2=3, R3=4

Sheet 3

- Explain how does the return from subroutine work in these two functions?

```
Function PUSH {R4,LR}          Function2
        ;stuff                         ;stuff
        POP  {R4,PC}                   BX LR
```

Answer

Function PUSH	Function2
push{lr} is putting the return address, in the link register, onto the stack when the subroutine is called. pop{pc} is fetching that return address from the stack and putting it into the program counter, thus returning control back to the place the subroutine was called from.	At the end of the subroutine, the BX LR instruction will retrieve the return address from the LR register, returning the program to the place from which the subroutine was called. More precisely, it returns to the instruction immediately after the instruction that performed the subroutine call.

Sheet 3

- Write a complete ARM assembly program for the procedure func2. The procedure func2 calculates this C expression ((X+Y)>>3) – Z and stores its value in R0. Assume X, Y, Z are 32-bit signed numbers. X, Y, Z are defined in the memory as shown

```
        AREA    mydata, DATA, READONLY
X       DCD     -20
Y       DCD     -60
Z       DCD     -20
```

Answer

- Answer:

- func2
- LDR R0, =X
- LDR R1, =Y
- LDR R2, [R0]
- LDR R3, [R1]
- ADD R0, R2, R3
- ASR R0, R0, #3
- LDR R4, =Z
- LDR R5, [R4]
- SUB R0, R0, R5
- BX LR

Sheet 3

- Write a complete ARM assembly program that calls the procedure func1 which in turn calls a procedure func2. The procedure func2 is defined in Q8 of Sheet 3.

Answer

```
Reset_Handler
                BL func1
DeadLoop        B DeadLoop
func1
                PUSH {LR}
                BL func2
                POP {LR}
                BX LR
```

Syntax of Arithmetic Instructions

- op{cond}{S} Rd, Rn, Operand2
  - op -> operation such as ADD
  - cond (EQ,NE, GT,..etc.)-> is an optional condition code
  - S -> is an optional suffix. If S is specified, the condition code flags are updated on the result of the operation

Startup File

- A startup file is a piece of code written in assembly or C language that executes before the main() function of our embedded application. It performs various initialization steps by setting up the hardware of the microcontroller so that the user application can run. Therefore, a startup file always runs before the main() code of our embedded application.

- The startup file performs various initializations and contains code for interrupt vector routines.