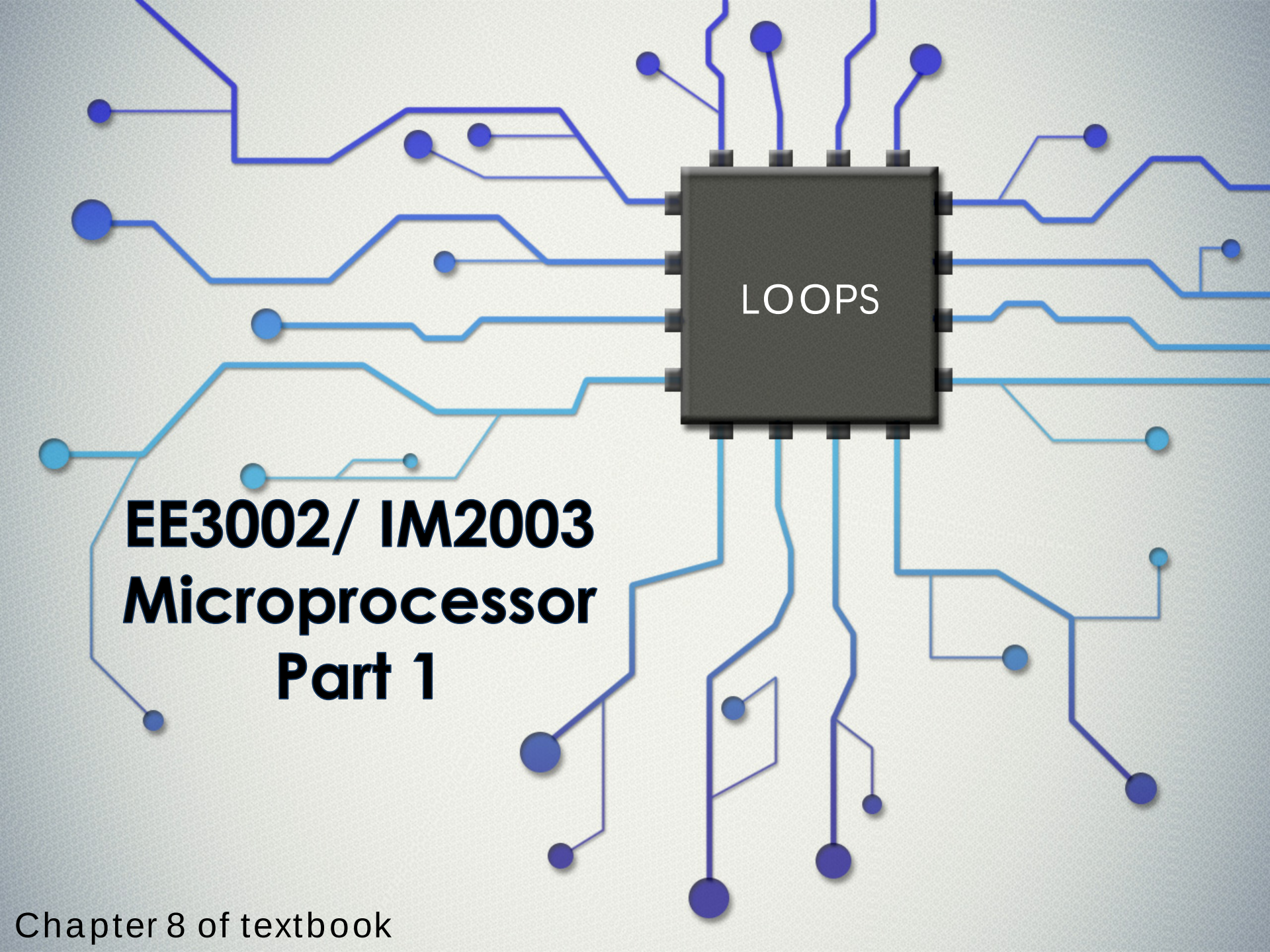


EE3002/ IM2003 Microprocessor Part 1

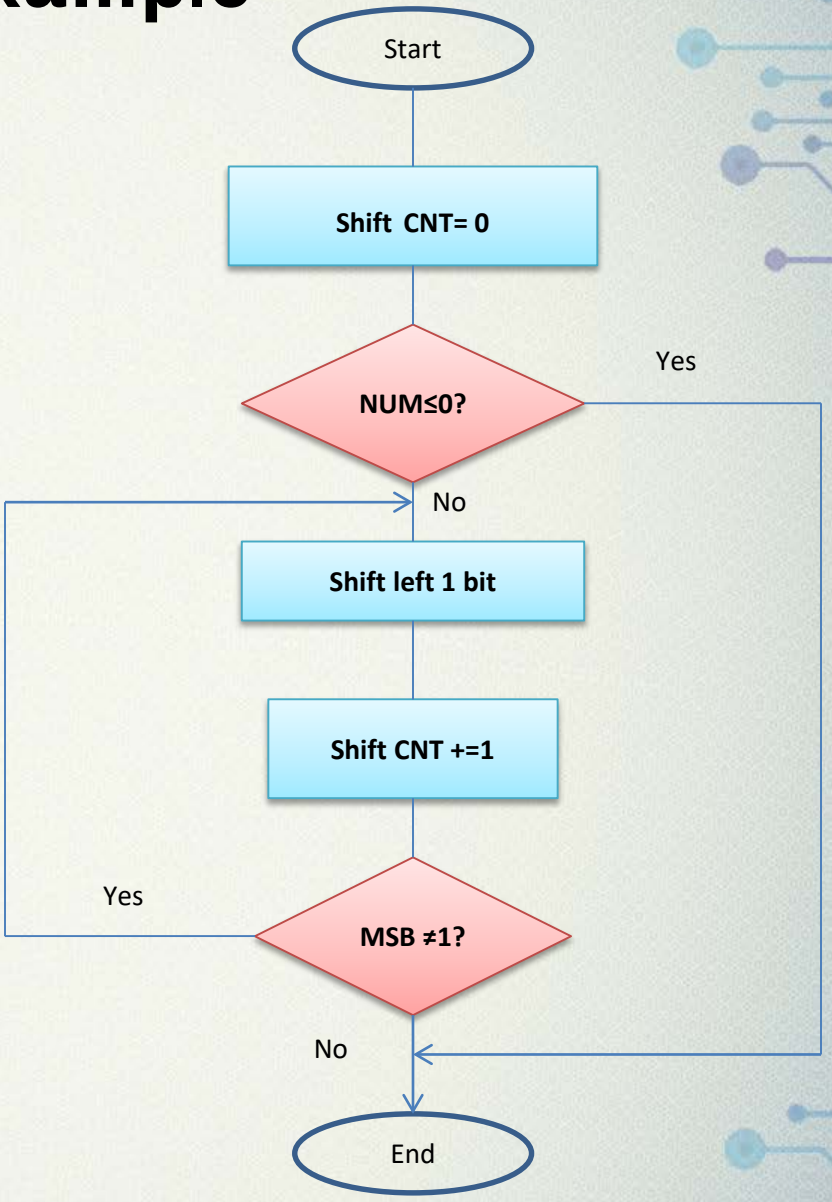
Chapter 8 of textbook

More on Loops

- We have encountered loops before. Now we will cover looping in more details.
- Looping actually interferes with the 3-stage pipeline in ARM architecture. This reduces the efficiency of the pipeline.
- The reason is simple, since it involves a conditional execution of a branching instruction, it is not possible to fetch the next instruction in advance.
- So unnecessary branching should be avoided for efficiency sake.

Loop Example

- Lets look at a program that count the number of leading zeroes in a register.
- The flow chart of the program is given on the right.



Program

```
;r3 contains the binary value to be examined
;r4 will contain shift count at the end
    MOV   r4, #0    ;clear shift count
    CMP   r3, #0    ;is the original value <=0
    BLE     finish    ;if yes, no leading zeroes
loop  MOVS  r3, r3, LSL #1 ;shift left one bit
    ADD   r4, r4, #1 ; increment shift counter
    BPL loop  ; condition code PL test  N flag clear
finish
```

While Loops

- While loops evaluate the loop condition before the loop body.

```
    B   Test
Loop  ...
    ....;    instructions
Test      ;    evaluate condition
    BNE   Loop
```

For Loops

- Example in C language

  for (j=0; j<10; j++) {instructions}

- In assembly

```
    MOV     r1, #0   ;j=0
Loop CMP    r1, #10 ;j<10?
    BGE     Done   ;if j >= 10, finish
    ...             ;instructions
    ADD     r1, r1, #1   ;j++
    B       Loop
Done
```

Count down loops

- In cases when a count down loop can be used instead of a count up loop, it should be used.
- A CMP instruction can be saved.

```
    MOV     r1, #10    ;j = 10
Loop  ...
    ...                ;instructions
    SUBS    r1, r1, #1 ;j = j-1
    BNE     Loop       ;if j= 0, finish
Done
```

Example 1

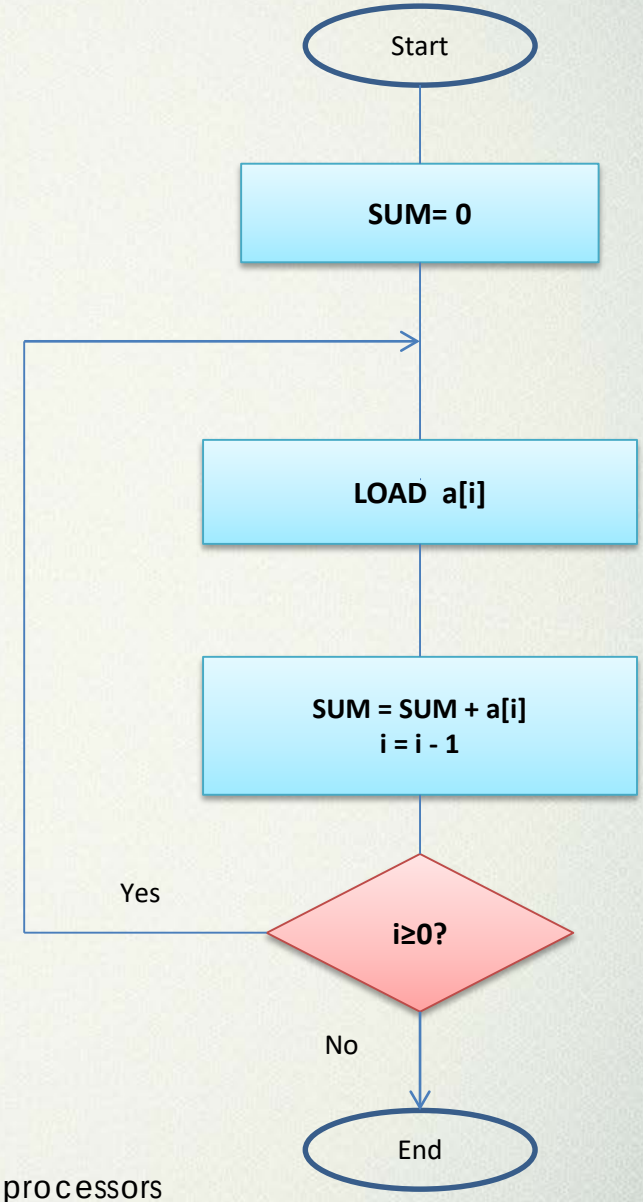
- Translate the following C code to assembly

```
for (i=0; i<8; i++) {
 a[i] = b[7-i]
}
```

```
        AREA        Prog8b, CODE, READONLY
SRAM_BASE   EQU   0x40000000
        ENTRY
        MOV       r0, #0              ;i
        ADR r1, arrayb        ;load address of array b
        MOV       r2, #SRAM_BASE ; a[i] starts here
Loop      CMP       r0, #8              ;i = 8?
        BGE   done
        RSB  r3, r0, #7         ;index = 7-i
        LDRB      r5, [r1, r3]        ;load b[7-i]
        STRB r5, [r2, r0]       ;store into a[i]
        ADDr0,r0, #1            ;i++
        B     Loop
done      B     done
        ALIGN
arrayb    DCB 0xA, 0x9, 0x8, 0x7, 0x6, 0x5, 0x4, 0x3
        END
```

Summation Example

- Lets look at a program that sum six 32-bit integers.
- The flow chart of the program is given on the right.



```
AREA Prog8c, CODE, READONLY
ENTRY
    MOVr0, #0          ;sum =0
    MOVr1, #5          ;# of elements -1
    ADR r2, arraya     ;load start of array
Loop  LDR r3, [ r2,r1, LSL #2] ;load value from memory
          ADD      r0, r3, r0      ;sum += a[i]
          SUBS     r1, r1, #1      ;i=i-1
          BGE Loop           ;loop only if i>= 0
done  B     done
          ALIGN
arraya  DCD    -1, -2, -3, -4, -5, -6
          END
```

Do… While Loops

- Structure as follows:

```
LOOP  ....        ; loop body
      ....        ; evaluate condition
      BNE   LOOP
EXIT  ....
```

More on Flags

- Flags are based on the results of comparisons or ALU operations if the S suffix is added.
- Flags can be used to control loops.
- Flags can also be used to control execution of instructions !!!

Condition codes

Field Mnemonic	Condition Code Flags	Meaning
EQ	Z set (Z == 1)	Equal
NE	Z clear (Z == 0)	Not equal
CS/HS	C set (C == 1)	Unsigned ≥
CC/LO	C clear (C == 0)	Unsigned <
MI	N set (N == 1)	Negative
PL	N clear (N == 0)	Positive or zero
VS	V set (V == 1)	Overflow
VC	V clear (V == 0)	No overflow
HI	C set and Z clear (C==1 && Z==0)	Unsigned >
LS	C clear or Z set (C==0 || Z==1)	Unsigned ≤
GE	N = V (N==V)	Signed ≥
LT	N ≠ V (N !=V)	Signed <
GT	Z clear and N = V (Z==0 && N==V)	Signed >
LE	Z set or N ≠ V (Z ==1 || N !=V)	Signed ≤
AL	Always	Default

Example

- Suppose that you need to compare 2 signed numbers in 2's complement form, with 0xFF000000 in r0 and 0xFFFFFFFF in r1. If you want to branch to some code only if the first number was less than the second, you might have something like

```
CMP     r0, r1      ;r0<r1?
BLT     algor
```

In this case, the branch would be taken as r0 is less r1. If both numbers are UNSIGNED, then BCC should be used instead.

Conditional Execution

- Branches should be reduced for efficiency sake.
- Removing a branch operation will not only improve execution time but also reduces code size.
- Conditional execution provides this capability.

Example

- Suppose you have to test a string for the presence of either a “!” or a “?” character. In C language, it can be written as

```
if (char == ‘!’ || char == ‘?’)
 found++;
```

- Assume char in r0 and found in r1

```
TEQ     r0, #’!’     ;test if equal using EOR
TEQNE r0, #’?’     ;if equal Z flag will be set
ADDEQ    r1, r1, #1 ;if Z set, increment r1
```

Final Example(GCD)

- The Greatest Common Divisor algorithm by Euclid is presented as follows.

```
while (a != b) {   /* a and b positive nos */
 if (a>b) a = a – b;
 else b = b – a;
 }
```

- E.g. a = 18, b = 6

  first pass : a = 12, b = 6

  second pass : a = 6, b = 6 (answer = 6)

Assembly Program 1

- Assume r0 contain ***a*** and r1 contain ***b***

```
gcd   CMP  r0, r1      ; a>b?
      BEQ   end        ; if a = b we're done
      BLT   less       ; a<b branches
      SUB   r0, r0, r1 ; a = a-b
      B   gcd          ; loop again
less
      SUB   r1, r1, r0 ;b = b –a
      B   gcd
```

Better Assembly Program

```
gcd   CMP     r0, r1
      SUBGT   r0, r0, r1
      SUBLT r1, r1, r0
      BNE     gcd
; no of branches reduced from 4 to 1!!!
```

Straight-line Coding

- If the number of times a loop is executed is small (<10 times), it is much more efficient to repeat the codes (straight-line coding) instead of using loops. In other words, straight line coding will be much faster.
- However the price paid is more memory used due to the increase code length.

Sum 6 Numbers Straight-line Coding

```
ADR     r2, arraya    ;load start of array
    LDR     r0, [ r2], #4          ;load first value from memory
LDR     r3, [ r2], #4          ; load 2nd value from memory
ADD     r0, r3, r0
LDR     r3, [r2], #4;load 3rd value from memory
ADD     r0, r3, r0    ;
LDR     r3, [ r2], #4          ; load 4th value from memory
ADD     r0, r3, r0
LDR     r3, [r2], #4;load 5th value from memory
ADD     r0, r3, r0    ;
LDR     r3, [r2], #4;load 6th value from memory
ADD     r0, r3, r0    ;
```

Summary

- Looping in various styles
- Conditional execution of instructions
- Straight-line coding for speed