Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

CS Basics
6) Bits and Branching

E. Benoist & C. Grothoff

Fall Term 2018-19

Computer Science Division

Bits and Branching

- Functions on bits
  - Shifts
  - Rotate
- Example : hexdump1
- Flags tests and branches
  - Comparisons
- Memory Addressing
- Conclusion

Functions on bits

Functions on bits

- **Binary boolean functions**[1]
  - Work on a Bit-by-Bit basis (similar to one of our exercises)
  - `AND`
    True if and only if both operands are true
  - `OR`
    True if and only if at least one of the operands is true
  - `XOR`
    True if and only if just one of the operands is true
- **Unary boolean function**
  - `NOT` change a value into its opposite.

[1] More on this in the course "Descrete mathematics"

Masking out bits

- **AND can be used to isolate bits**
  - We use and to set all `unwanted` bits to 0
- **Principle**
  - We use a bit mask: the interesting bits are 1, other are 0
  - We make a AND
  - In the result, all the uninteresting bits are removed, we can read interesting bits directly
- **Example**
  - We are interested in bits 0 and 3 in a byte (we start counting from the left with number 0)
  - The mask is 0000 1001 (noted 09H)
  - If we are interested by the bit 3 and 0 of register AL (8 bit general purpose register)
  - `and al,09h`
  - All the unitersting bits are 0
  - Values of bit 0 and 3 remain, the rest is erased

An AND instruction

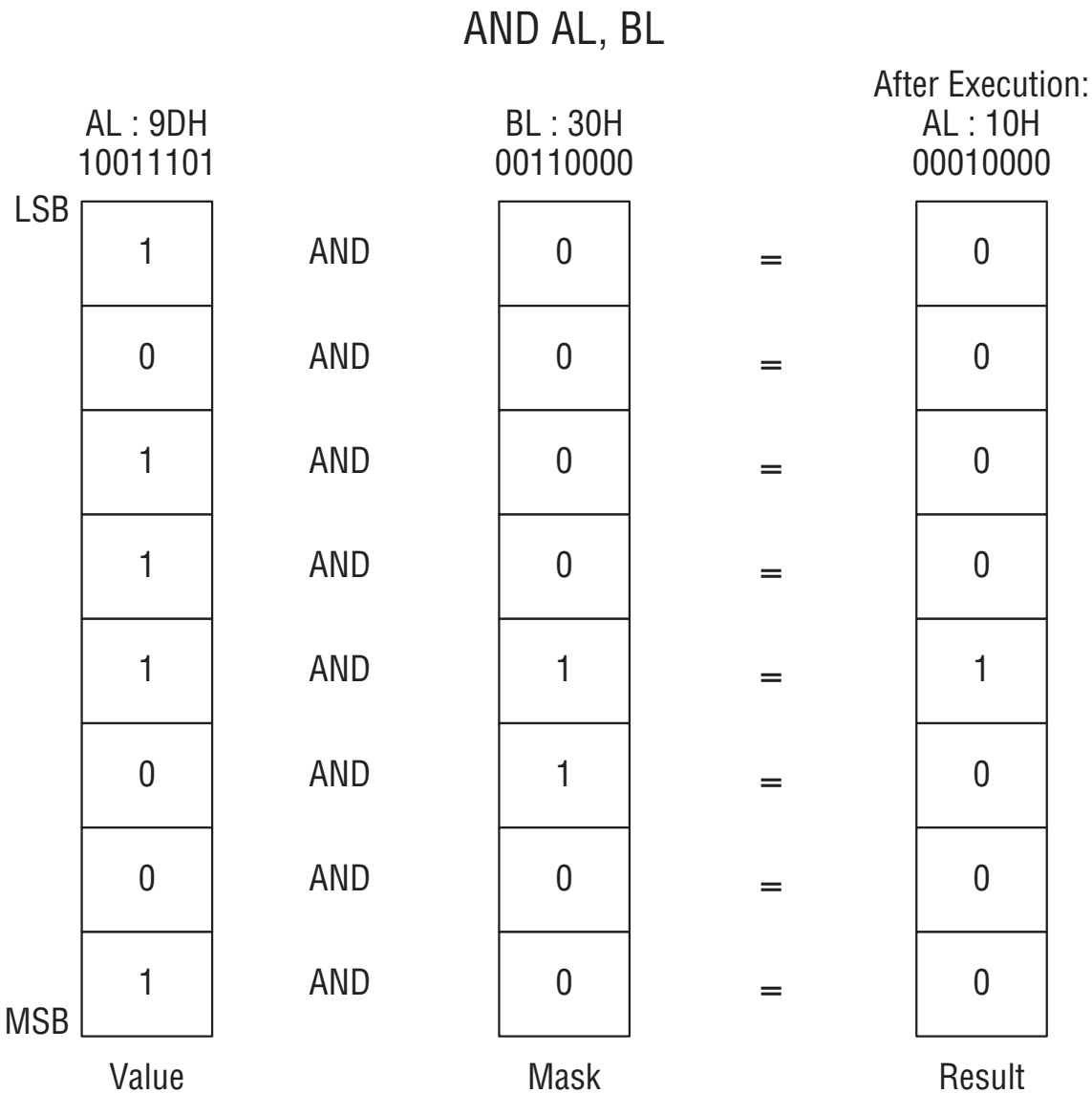

Figure 9-2: The anatomy of an AND instruction

Shifts

Shift bits

- **Shift bits to one side or the other**
  - `SHL` *SH*ifts bits to the *L*eft
  - `SHR` *SH*ifts bits to the *R*ight
- **Syntax**

```
shl <register/memory>, count
```

  The first operand : target of the operation
  The second operand : the number of bits by which to shift
- **Example**
  - AX contains B76FH =1011 0111 0110 1111B
  - `SHL AX,1`
  - AX becomes 6EDEH =0110 1110 1101 1110B
  - One bit has been "forgotten"
  - It has been transfered in the "Carry Flag"

Rotate

The Rotate instructions

- **To reuse the forgotten bits: Rotate**
  - `RCL`, `RCR`, `ROL` and `ROR` are rotate functions
- **Rotate Left** `ROL`
  - The bits reappear on the other side
  - 10110010
  - becomes 01100101

The rotate instruction

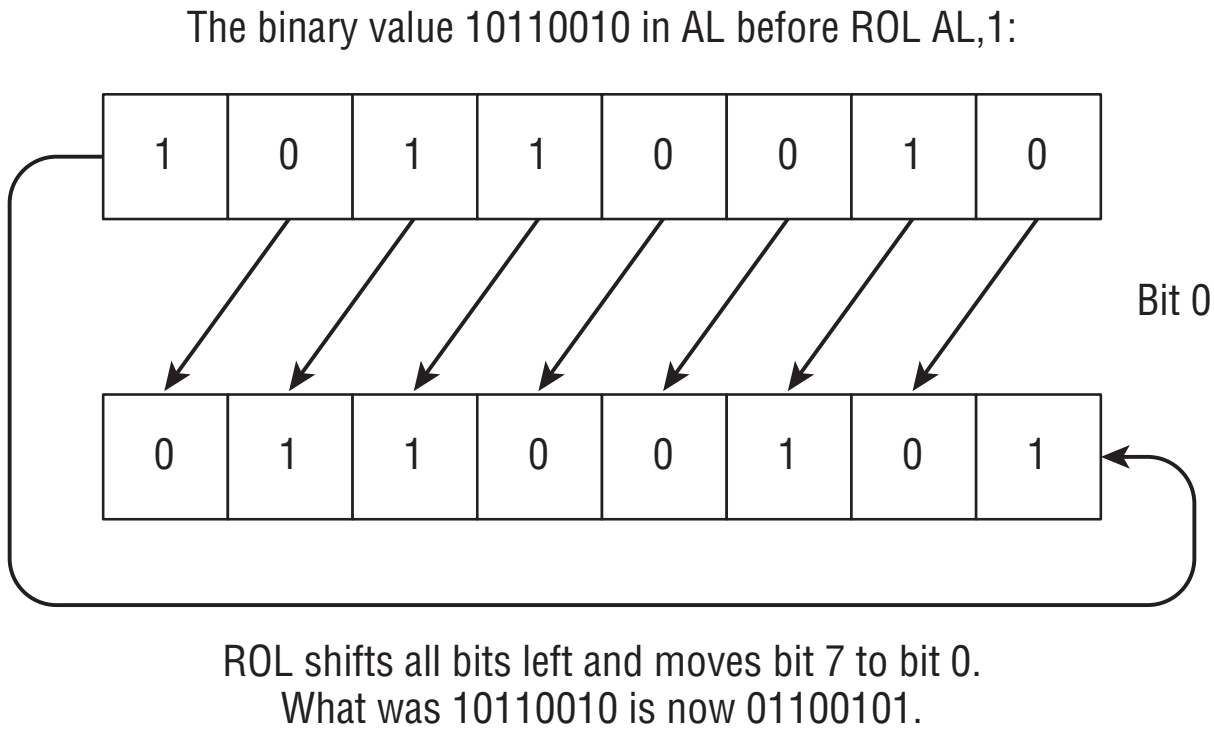

Figure 9-4: How the rotate instructions work

Rotate Carry

- **Functions `RCL` and `RCR`**
  - The forgotten bit goes in the carry
  - Then it is added on the other side

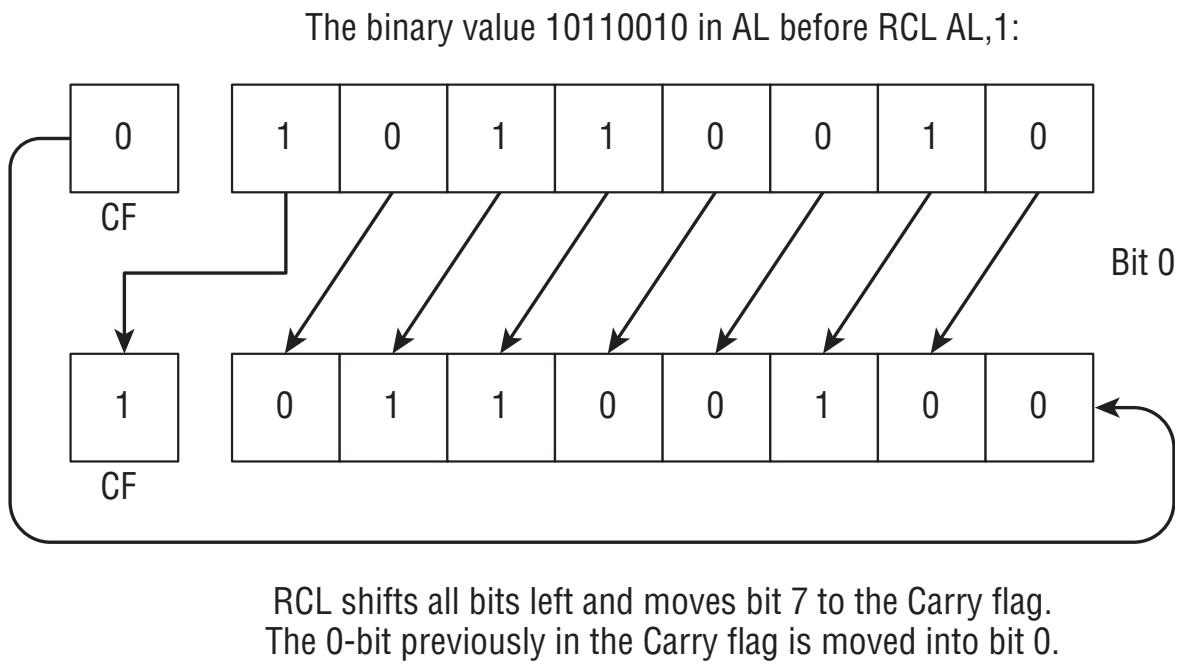

Figure 9-5: How the rotate through carry instructions work

Functions to set the Carry Flag

- **We can set the carry flag**
  - `CLC` Clear carry flag
  - `STC` Set carry flag
- **Can be useful to set the value before a rotate carry function**
  - The value can be set to 1
  - Or to 0

Example : hexdump1

Example Program

- **Display the content of a binary file**
  - Transform Bit and Bites into characters
  - Similar to a read only Hex editor
- **Standard flows**
  - Input file : standard input
  - Usage: `hexdump1 < file.bin`
  - Output on standard output
- **What it does**
  - Read 16 bytes at a time
  - Transform each byte in hex
  - Print out the 16 hexadecimal values separated by spaces

hexdump1 I

```
; .....
;  Run it this way:
;    hexdump1 < (input file)
;
SECTION .bss                    ; Section containing uninitialized data
        BUFFLEN equ 16          ; We read the file 16 bytes at a time
        Buff:   resb BUFFLEN    ; Text buffer itself
SECTION .data                   ; Section containing initialised data
        HexStr: db " 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00",10
        HEXLEN equ $-HexStr
        Digits: db "0123456789ABCDEF"
SECTION .text                   ; Section containing code
global  _start                  ; Linker needs this to find the entry point!
_start:
        nop                     ; This no-op keeps gdb happy...
; Read a buffer full of text from stdin:
Read:
        mov eax,3               ; Specify sys_read call
        mov ebx,0               ; Specify File Descriptor 0: Standard Input
        mov ecx,Buff            ; Pass offset of the buffer to read to
        mov edx,BUFFLEN         ; Pass number of bytes to read at one pass
        int 80h                 ; Call sys_read to fill the buffer
        mov ebp,eax             ; Save # of bytes read from file for later
        cmp eax,0               ; If eax=0, sys_read reached EOF on stdin
        je Done                 ; Jump If Equal (to 0, from compare)
  ....
```

hexdump1 II

```
; Set up the registers for the process buffer step:
        mov esi,Buff            ; Place address of file buffer into esi
        mov edi,HexStr          ; Place address of line string into edi
        xor ecx,ecx             ; Clear line string pointer to 0

; Go through the buffer and convert binary values to hex digits:
Scan:
        xor eax,eax             ; Clear eax to 0

; Here we calculate the offset into the line string, which is ecx X 3
        mov edx,ecx             ; Copy the pointer into line string into edx
;       shl edx,1               ; Multiply pointer by 2 using left shift
;       add edx,ecx             ; Complete the multiplication X3
        lea edx,[edx*2+edx]
; Get a character from the buffer and put it in both eax and ebx:
        mov al,byte [esi+ecx]   ; Put a byte from the input buffer into al
        mov ebx,eax             ; Duplicate the byte in bl for second nybble

; Look up low nybble character and insert it into the string:
        and al,0Fh                  ; Mask out all but the low nybble
        mov al,byte [Digits+eax]    ; Look up the char equivalent of nybble
        mov byte [HexStr+edx+2],al  ; Write the char equivalent to line string

; Look up high nybble character and insert it into the string:
        shr bl,4                    ; Shift high 4 bits of char into low 4 bits
        mov bl,byte [Digits+ebx] ; Look up char equivalent of nybble
        mov byte [HexStr+edx+1],bl  ; Write the char equivalent to line string
```

hexdump1 III

```
; Bump the buffer pointer to the next character and see if we re done:
        inc ecx             ; Increment line string pointer
        cmp ecx,ebp         ; Compare to the number of characters in the buffer
        jna Scan            ; Loop back if ecx is <= number of chars in buffer

; Write the line of hexadecimal values to stdout:
        mov eax,4                   ; Specify sys_write call
        mov ebx,1                   ; Specify File Descriptor 1: Standard output
        mov ecx,HexStr              ; Pass offset of line string
        mov edx,HEXLEN              ; Pass size of the line string
        int 80h                     ; Make kernel call to display line string
        jmp Read                    ; Loop back and load file buffer again

; All done! Let s end this party:
Done:
        mov eax,1                   ; Code for Exit Syscall
        mov ebx,0                   ; Return a code of zero
        int 80H                     ; Make kernel call
```

Lookup table

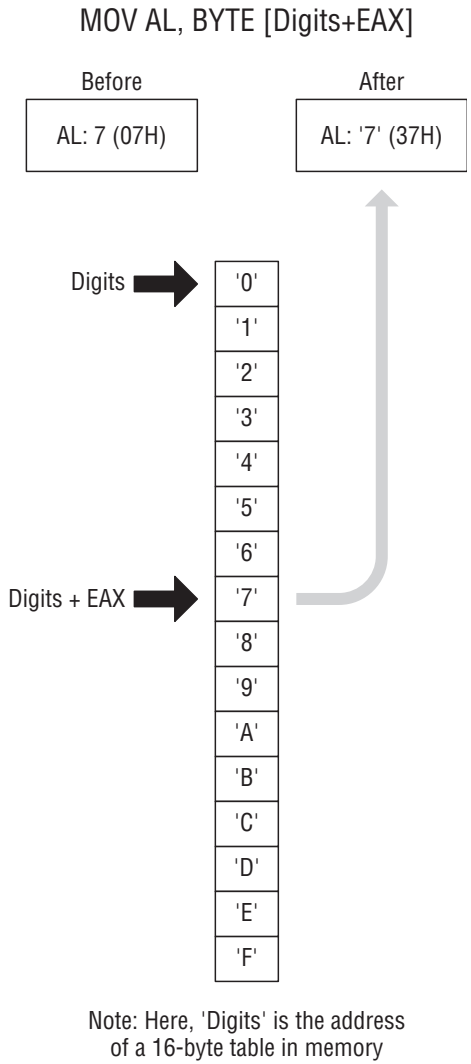

Figure 9-6: Using a lookup table

HexStr

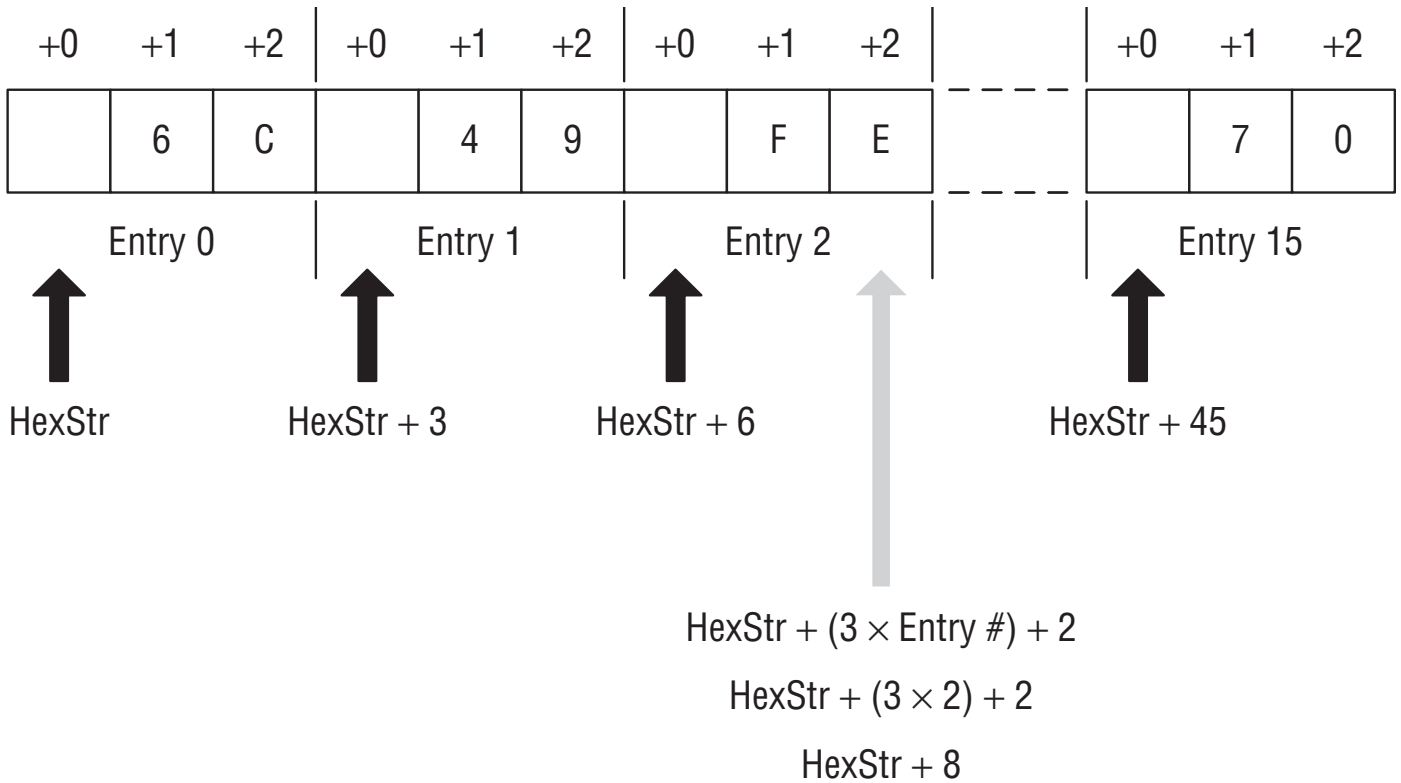

Figure 9-7: A table of 16 three-byte entries

Multiplying by shifting

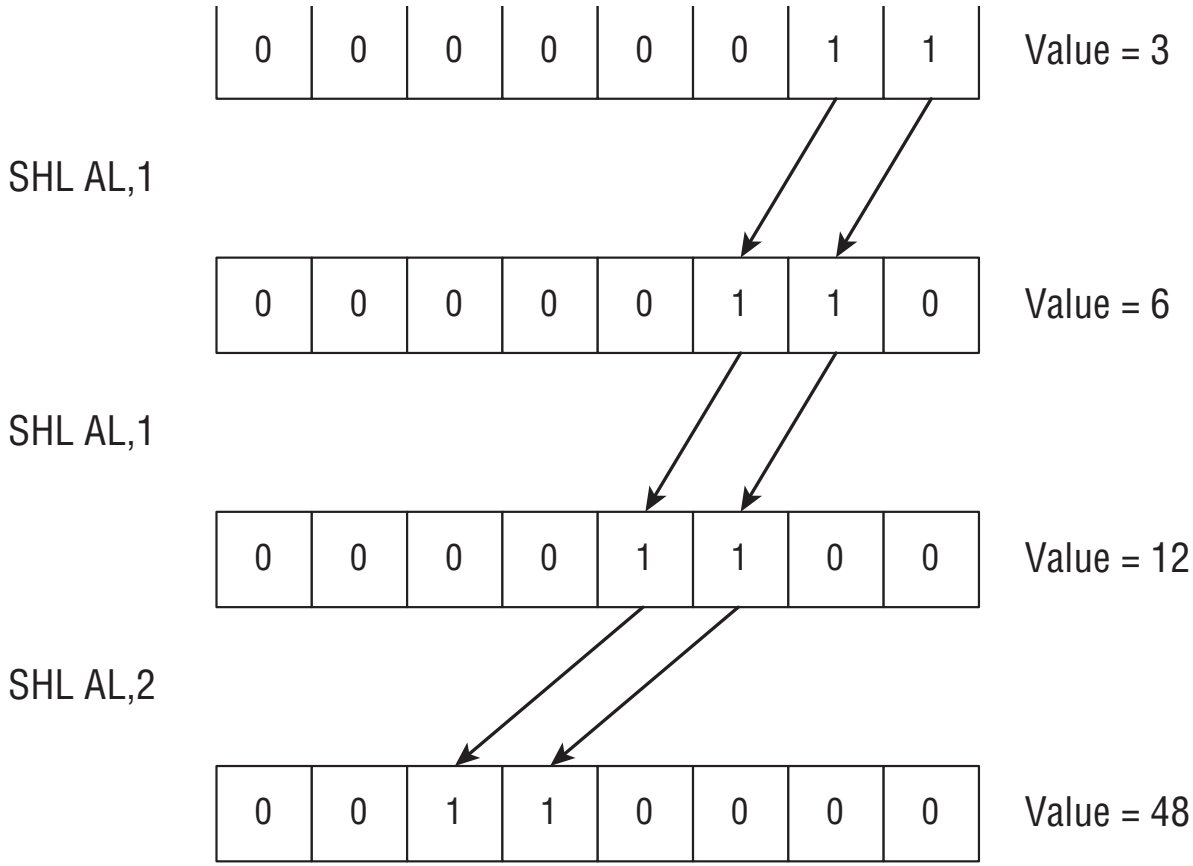

Figure 9-8: Multiplying by shifting

Multiply by shifting and adding

- **Multiply by two**

```
mov al,3
shl al,1
shl al,1
shl al,2
```

  AL is 48 = 3*2*2*4

- **Multiply by 3**

  Multiply x by two and add once x

```
mov rdx, rcx      ; copy the number
shl rdx, 1          ; multiply by two
add rdx, rcx      ; add x*2 and x
```

Multiply

- **Multiply by 7**

```
mov rdx, rcx
shl rdx,2          ; edx multiplied by 4
add rdx, rcx   ; edx is value times 5
add rdx, rcx   ; times 6
add rdx rcx   ; times 7
```

Flags tests and branches

Jumps

- **Unconditional jumps**
  - Go to an address in any case
  - `JMP`

```
jmp <label>
```

- **Conditional jumps**
  - Tests the value of one or more flags before jumping
  - otherwise it goes to the next instruction
- **Test the Zero Flag (ZF)**
  - `JNZ` Jump Non Zero (seen previousely)
  - `JZ` Jump if ZF is set

Jumps (example)

- **Jump and jump zero**

```
    mov word [Runningsum],0         ; Clear the ↘
    →running  total
    mov rcx, 17                              ; We ↘
    →will loop 17 times
WorkLookup:
    add word [Runningsum],3         ; Add 3 to the ↘
    →running total
    dec rcx                         ; substract one↘
    → to the counter
    jz SomewereElse                 ; If the ↘
    →counter is zero, we are done!
    jmp WorkLookup                  ;  We loop ↘
    →otherwise
```

Comparisons

Comparisons

- **Flags are used to comparate**
  - We use the instruction `CMP`
  - It comparates two operands
  - and sets the following flags: OF, SF, ZF, AF, FF
  - Basically a substraction, where the result is thrown away
  - Just the flags remain

```
cmp <op1>, <op2>    ; sets OF, SF, ZF, AF, ↘
→PF and CF
                                ; Equivalent to
                                ; Result = <op1> - ↘
                                →<op2>
                                ; sub <op1>,<op2>
                                ; Where result is ↘
                                →forgotten
```

Comparisons (Cont.)

- **Provides the following testings**
  - `Equal` (ZF is set),
  - `Not equal` (ZF is not set),
  - `Greater than` (ZF is not set, SF is not set),
  - `Less than` (ZF is not set, SF is set),
  - `Greater than or equal to` (ZF is set or SF is not set),
  - and `Less than or equal to`
- **Equality testing jumps**
  - `JE`: Jump if Equal, `JNE`: Jump not equal
- **Jumps for Singed Values**
  - Greater than / less than
  - `JG`: Jump if Greater; `JGE`: Jump if Greater or Equal
  - `JNG`, `JNGE`
  - `JL`, `JLE`, `JNL`,`JNLE`
- **Jumps for Unsigned Values**
  - Above / below
  - `JA`: Jump is Above; `JAE`: Jump is Above or Equal
  - `JNA`, `JNAE`
  - `JB`, `JBE`, `JNB`, `JNBE`

Test one bit

- **We are interested in testing one bits**
  - `TEST` operation tests elements bitwise
  - It executes a `AND`, sets the flags and discards the result

```
test <operand>, <bit mask>
test ax, 08h  ; Bit 3 in binary is 0000 ↘
→1000B
```

  AX is not changed (unlike with AND)
- **If bit 3 is a 1**
  - Zero Flag is 0
- **If bit 3 is a 0**
  - Zero Flag is 1

Bit Test

- **One can test a bit with** `BT`
  - The bit is copied in the Carry Flag CF

```
bt <value containing bit>, <bit number>
```

- **Testing**
  Test the value of CF `JC` (Jump if Carry) and `JNC` (Jump if Not Carry)

```
bt eax, 4                ; Test bit 4 of AX
jnc quit                  ; We are done if bit 4 ↘
→= 0
```

Memory Addressing

Protected Mode Memory Addressing

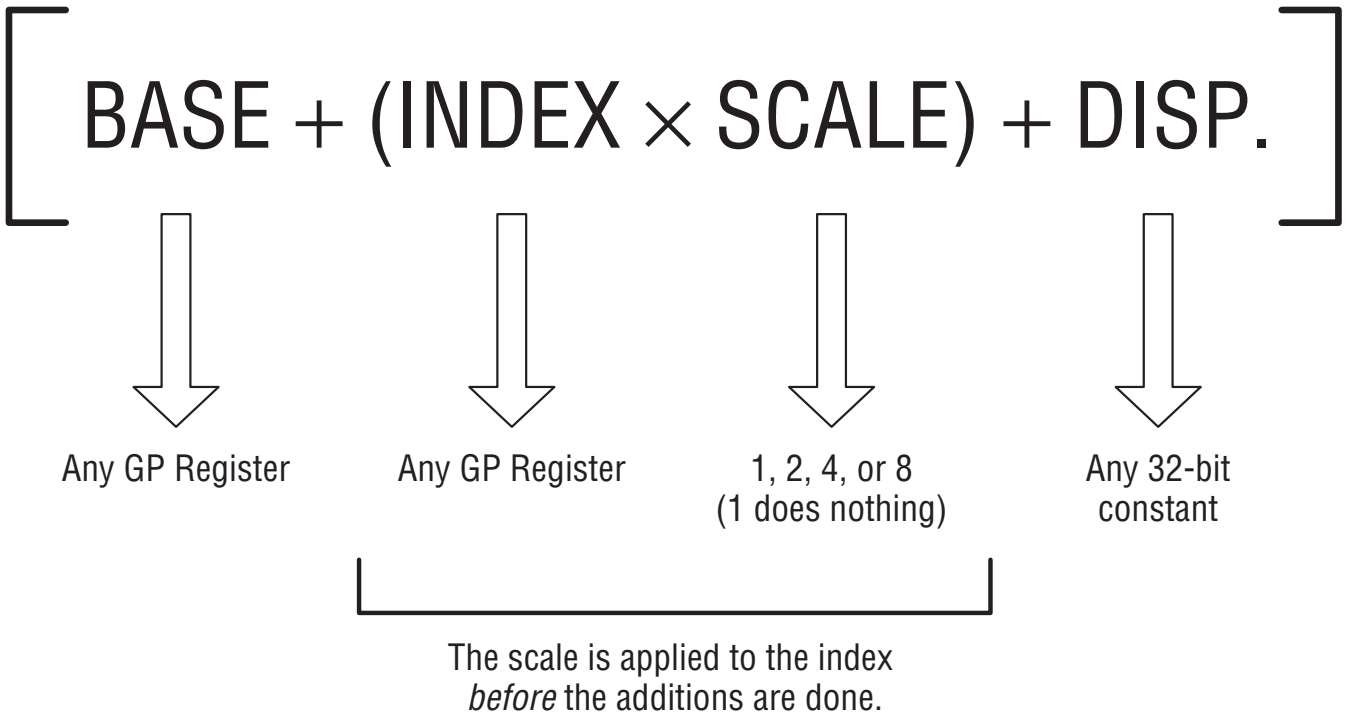

Figure 9-9: Protected mode memory addressing

Protected Mode Memory addressing

- **Effective Address Calculations**
  - The address used to read or write in memory
- **Base scheme**
  - The quantity inside a register contains the address

```
mov   [eax], 42     ; Moves 42 into the ↘
→memory which address is in eax
```

- **Displacements**
  - Symbolic address defined in .bss or .data

```
mov eax, [HexStr]           ; HexStr is a ↘
→label pointing somewhere in the memory
mov ebx, [HexStr + 3]   ; Displacement of↘
→ 3 bytes starting at HexStr
```

Memory addressing (Cont.)

- **Base + Displacement Addressing**
  -

```
mov byte [HexStr+edx+2], al   ; Copy one ↘
→byte stored in al into memory
```

  HexStr+2 is computed at compilation,
  so we have: Base (edx) and displacement (HexStr)
- **Base + Index Addressing**
  - Address is calculated based on two registers, one is the base (stable), the other the index (incremented)

```
sub byte [ebp+ecx], 20h      ; substract ↘
→20h from one byte in memory
```

Loop visiting memory

```
; Set up the registers for the process buffer ↘
→step:
   mov ecx,esi   ; Place the number of bytes read↘
   → into ecx
   mov ebp,Buff              ; Place address of ↘
   →buffer into ebp
   dec ebp                             ; Adjust count ↘
   →to offset
; Visit buffer, convert lowercase to uppercase ↘
→chars:
Scan:
   cmp byte [ebp+ecx],61h   ; Test input char ↘
   →against lowercase 'a'
   jb Next   ; If below 'a' in ASCII, not ↘
   →lowercase
   cmp byte [ebp+ecx],7Ah          ; Test input ↘
   →char against lowercase 'z'
   ja Next   ; If above 'z' in ASCII, not ↘
   →lowercase
```

Memory Addressing (Cont.)

- **Index x scale + displacement addressing**
  - When the data is an array
  - scale = size of one data
  - If we read 4 bytes at a time, we visit only addresses separated by 4 bytes

```
            mov ecx, 0
MyLoop:
        add ecx,4
        mov edx, [Digits + ecx]    ; 4 bytes ↘
        →in memory are transfered          at↘
        → once
        .....
        jnz MyLoop                          ; ↘
        →end of loop
```

How address scaling works

We have a label `DDTable` on a table of double words (dd)

[DDTable + ECX * 4]

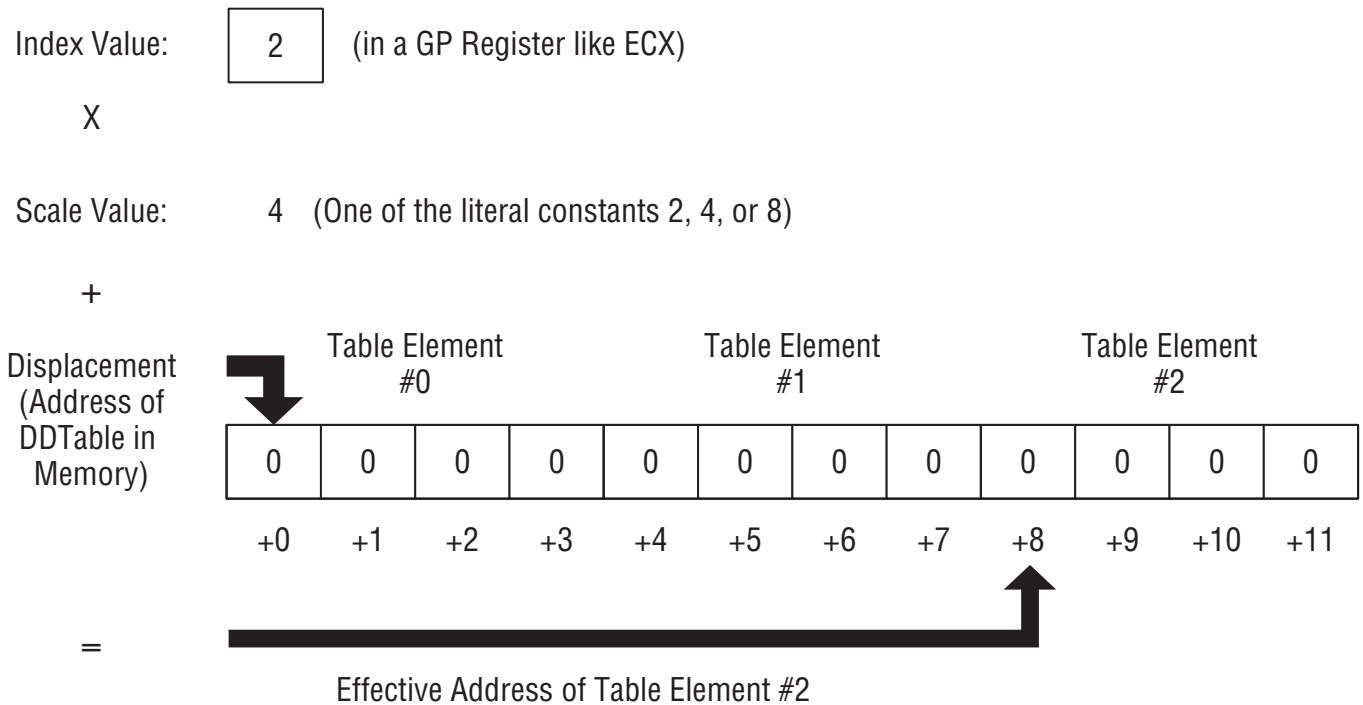

Figure 9-10: How address scaling works

Load Effective Address LEA

- **It is sometime useful to store an effective address in a register**
  - To access it later

```
lea ebx, [ScaleValue+ecx*4]      ; stores↘
→ the address in ebx
```

- **LEA does not store the value, only the address!**
  - Can be missued to do some calculation (not on address)
  - The two following codes are equivalent (they multiply edx by 3)

```
mov edx, ecx      ; copy the number
shl edx, 1          ; multiply by two
add edx, ecx      ; add x*2 and x
```

And

```
lea edx, [edx*2+edx]    ; Multiply edx by ↘
→3
```

Conclusion

Conclusion

- **Manipulating bits**
  - Manipulate data as arrays of bits
  - Using boolean functions (AND, OR, XOR or NOT)
  - Using shifts (or rotate)
- **Example: hexdump1**
  - Transforms binary input into a readable file
  - All values are written in hexadecimal
- **Jumps**
  - Lots of different jumps
  - Depend on testing
    On Flags (Zero, Carry, ...)
    On comparisons of numbers ($>$, $<$, $\geq$, *leq*)
- **Memory addressing**
  - Must respect schemes
  - Is very convenient to access memory

Bibliography

- This course corresponds to the chapter *9* of the course book: **Assembly Language Step by Step (3rd Edition)**