Τμήμα Πληροφορικής
Πανεπιστήμιο Κύπρου

ΕΠΛ232 – Προγραμματιστικές Τεχνικές και Εργαλεία

Διάλεξη 18: Χαμηλού Επιπέδου Προγραμματισμός Ι

(Κεφάλαιο 25.1, KNK-2ED)

Δημήτρης Ζεϊναλιπούρ

http://www.cs.ucy.ac.cy/courses/EPL232

Περιεχόμενο Διάλεξης

- **Εισαγωγή**
- **Δυαδικοί Τελεστές** (Bitwise Operators)
  - Τελεστές Ολίσθησης (Shift): >>, <<
  - Τελεστές Συμπλήρωμα (~), AND (&), OR (|), XOR (^)
  - Χρήση Τελεστών για πρόσβαση σε bit
    - Ανάθεση, Διαγραφή και Δοκιμή Bit
  - Ονόματα σε Μάσκες
  - Παράδειγμα Κρυπτογράφησης XOR
- Δομές με Δυφία (Struct with bit-fields)
- Θέματα Ευθυγράμμισης Μνήμης με Δομές Διφυιών

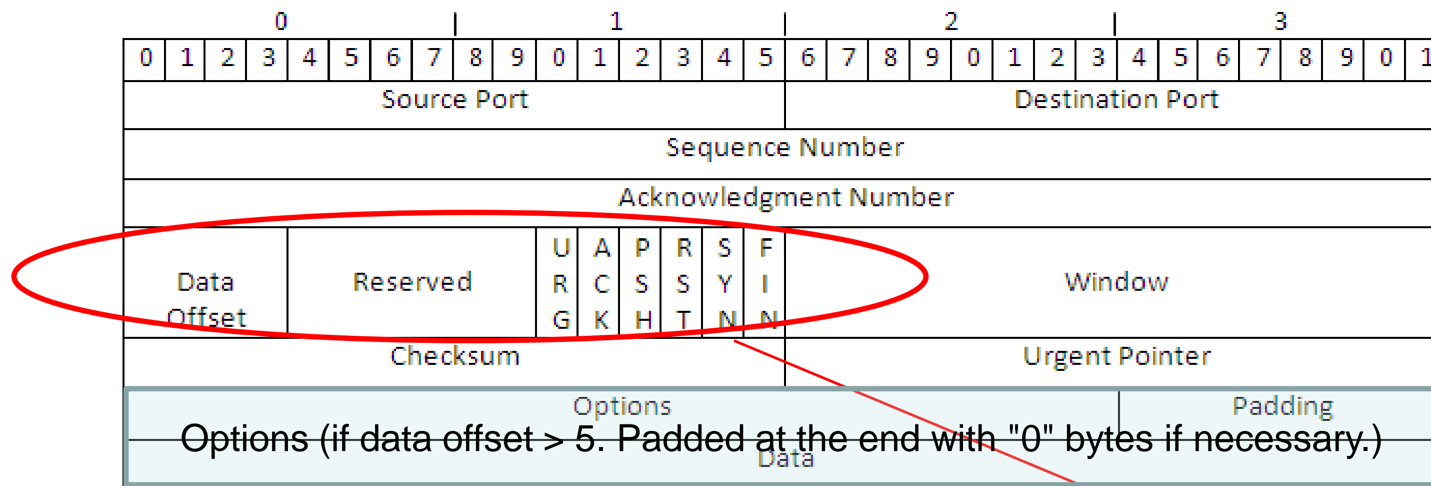
Διάλεξη #19

Προγραμματισμός με Bits (Εισαγωγή)

- Στις **προηγούμενες διαλέξεις** έχουμε γνωρίσει τη C ως μια **υψηλού επίπεδου γλώσσα**.
  - Οι δυνατότητες της σε αυτό το πλαίσιο ήταν **ανεξάρτητες αρχιτεκτονικής** (machine-independent), π.χ., χρήση sizeof(int) θα μας έδινε κάποια machine-independent υλοποίηση
  - Επίσης, **ΟΛΗ** η επεξεργασία **δεδομένων** στη **μνήμη**, **αρχεία**, κτλ. γινόταν σε **μονάδες** **Byte (B)**.
- Ωστόσο, αρκετά προγράμματα, χρειάζονται να εκτελούν λειτουργίες σε **μονάδες** **Bit (b)**, π.χ.,
  - Προγραμματισμός Συστημάτων (Λειτουργικά Συστήματα, Βάσεις Δεδομένων, Δίκτυα, κτλ.)
  - Προγράμματα/Βιβλιοθήκες κρυπτογράφησης, συμπίεσης, κτλ.
  - Προγράμματα Γραφικών (π.χ., μεταβολή συγκεκριμένων bits μιας εικόνας)
  - Προγράμματα που απαιτούν γρήγορη εκτέλεση και/η αποδοτική αξιοποίηση των διαθέσιμων πόρων του συστήματος.

Προγραμματισμός με Bits (Εφαρμογές στα Δίκτυα)

- **Εφαρμογή στα Δίκτυα:** Για παράδειγμα, θεωρήστε την ακόλουθη δομή (**TCP**), η οποία βρίσκεται κάτω από οποιαδήποτε επικοινωνία μας μέσω του WWW!
  - Κάθε πακέτο στο WWW χρησιμοποιεί το πρωτόκολλο: **HTTP** που βασίζεται σε **TCP** και **IP**.
  - Το Internet αποτελείται από πολλά αντίστοιχα πρωτοκόλλα.



Τα πεδία αυτά (flags), αποθηκεύονται σε bits για εξοικονόμηση χώρου στη μετάδοση ενός πακέτου.

Προγραμματισμός με Bits (Εφαρμογές στα Δίκτυα)

Απλοποιημένο παράδειγμα ενός **struct** το οποίο μπορεί να αναπαραστήσει τα δεδομένα του **TCP** σε ένα πρόγραμμα.

```
typedef struct {
    unsigned short source; // 2B
    unsigned short dest;   // 2B
    unsigned int seq;      // 4B
    unsigned int ack_seq;  // 4B
    unsigned short doff:4; //
    unsigned short res1:4; // 1B
    unsigned short res2:2; //
    unsigned short urg:1;  //
    unsigned short ack:1;  //
    unsigned short psh:1;  //
    unsigned short rst:1;  //
    unsigned short syn:1;  //
    unsigned short fin:1;   // 1B
    unsigned short window; // 2B
    unsigned short check;  // 2B
    unsigned short urg_ptr;// 2B
} __attribute__ ((__packed__))  TCP;
```

Υποδηλώνει 4 bits

sizeof(TCP) = 20 bytes!

Προγραμματισμός με Bits (Εφαρμογές σε Βάσεις)

- **Εφαρμογή στις Βάσεις:** Έστω ότι θέλετε να αποθηκεύσετε τις 25 απαντήσεις {Y|N} ερωτηματολογίων για 1 εκατομμύρια άτομα!
  - 25 απαντήσεις * **1 int** * 10^6 άτομα = **95,36 MB**
  - 25 απαντήσεις * **1 char** * 10^6 άτομα = **23,84 MB**
  - 25 απαντήσεις * **1 bit** * 10^6 άτομα = **2,98 MB!**
- Στις βάσεις δεδομένων, όπου ο **όγκος δεδομένων** είναι μεγάλος, γίνεται σημαντική η **εξοικονόμηση χώρου**
  - για αυτό η **χρήση bit πεδίων** αντί των **byte** πεδίων μπορεί να επιφέρει μεγάλη εξοικονόμηση σε εξειδικευμένες εφαρμογές.
- Σε αυτή την διάλεξη θα δούμε πως μπορούμε να **επεξεργαστούμε τα bits** από μια εφαρμογή για εφαρμογές σε Δίκτυα, Βάσεις Δεδομένων, Λ.Σ., κτλ.
  - Στόχος δεν είναι να εμβαθύνουμε πολύ, περισσότερο το ζητούμενο είναι να δείτε ότι μια τέτοια επεξεργασία είναι εφικτή

Προγραμματισμός με Bits (Συζήτηση)

- Κάποια από αυτά που θα δούμε σε αυτή την ενότητα στηρίζονται στο πως είναι τα δεδομένα **αποθηκευμένα στην μνήμη.**
  - Συνεπώς, οι τεχνικές αυτές ενδέχεται να δυσχεράνουν την **μεταφερσιμότητα** του κώδικα σας, εάν δεν προγραμματιστούν ορθά (το παράδειγμα του TCP έπρεπε να είχε `#if…elif..endif`)
  - Βεβαιωθείτε ότι θα χρησιμοποιήσετε τις τεχνικές αυτές μόνο όπου **αντιλαμβάνεστε** πλήρως την **χρήση τους.**
- Επίσης, περιορίστε την χρήση τους σε **μερικές ενότητες** του **κώδικα σας** (π.χ., με επιλεκτικό compile #ifdef) παρά σε **όλο τον κώδικα.**
- Σε κάθε περίπτωση, αργά ή γρήγορα θα **χρειαστείτε** τους τελεστές αυτούς στην **πορεία των σπουδών** σας ή την **επαγγελματική σας πορεία**.
  - Π.χ., επέκταση υφιστάμενου κώδικα, μεταφορά βιβλιοθηκών κτλ.

Τελεστές Ολίσθησης (Bitwise Shift Operators)

- Η C (και όλες οι πραγματικές γλώσσες) παρέχει **έξι (6) δυαδικούς** τελεστές (`>>`, `<<`, `&`, `~`, `^`, `|`), οι οποίοι εφαρμόζονται πάνω σε **ακέραιες τιμές (μόνο!)** σε επίπεδο **bit.**
- Οι τελεστές ολίσθησης μετακινούν τα bits σε ένα ακέραιο στα δεξιά ή στα αριστερά:
  - **`val << offset`** (Αριστερή Ολίσθηση)
  - **`val >> offset`** (Δεξιά Ολίσθηση)
  - Όπου **`offset`** είναι θετικός ακέραιος μέσα στο εύρος ορισμού του val.
    - Π.χ., εάν `val` είναι `unsigned short` τότε offset είναι μέχρι 16 εναλλακτικά κάποιοι μεταγλωττιστές προειδοποιούν `warning: left shift count >= width of type`

Τελεστές Ολίσθησης (Bitwise Shift Operators)

- Παράδειγμα εφαρμογής των δυο τελεστών στην ακέραια τιμή 13:

```
unsigned short i, j;

i = 13;
  /* i is now 13 (binary 0000000000001101) */

j = i << 2; // δηλ., floor(13 * 4) = 52!
  /* j is now 52 (binary 0000000000110100) */

j = i >> 2; // δηλ., floor(13/4)= f(3.25)= 3!
  /* j is now  3 (binary 0000000000000011) */
```

Τα μηδενικά προστίθενται αυτόματα στην αρχή ή τέλος!

Τελεστές Ολίσθησης (Bitwise Shift Operators)

- Το αποτέλεσμα του `i << j` είναι αυτό όπου τα bits στο `i` μετακινούνται αριστερά κατά `j` θέσεις.
  - Για κάθε bit που γίνεται “shifted off” στο αριστερό άκρο του `i`, ένα zero bit εισάγεται στα δεξιά (στο LSB)

```
  j = i << 2;
/* j is now 52 (binary 0000000000110100) */
```

- Το αποτέλεσμα του `i >> j` είναι αυτό όπου τα bits στο `i` μετακινούνται δεξιά κατά `j` θέσεις.
  - Εάν `i` είναι **unsigned ή εάν η τιμή του `i`** είναι **μη-αρνητική**, μηδενικά προστίθενται στα αριστερά (στο MSb).
  - Εάν `i` είναι αρνητικό, το αποτέλεσμα εξαρτάται από την υλοποίηση!

- **Για λόγους μεταφερσιμότητας, χρησιμοποιείτε το shifting ΜΟΝΟ πάνω σε `unsigned` ακέραιους!**

Τελεστές Ολίσθησης (Bitwise Shift Operators)

Αποδοτικός Πολλαπλασιασμός / Διαίρεση με δυνάμεις του 2!

- Το **x << y** μας δίνει ένα πολύ αποδοτικό τελεστή για **πολλαπλασιασμό** του **x** με **2^y**!
  - Π.χ., 13 * 4 = 52 είναι το ίδιο με 13 << 2 = 52
  - `/* j is now 52 (binary 0000000000110100) */`

- Αντίστοιχα, το **x >> y** μας δίνει ένα αποδοτικό τελεστή για **διαίρεση** του **x** με **2^y**!
  - Π.χ., 13 / 4 = 3 είναι το ίδιο με 13 >> 2 = 3
  - `/* j is now  3 (binary 0000000000000011) */`

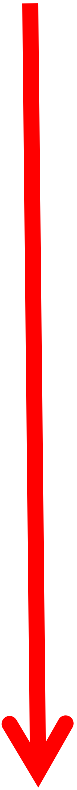
Τελεστές Ολίσθησης (Bitwise Shift Operators)

Άλλες Επισημάνσεις

- Οι **τελεσταίοι** x, y (x << y), εάν και συνήθως τύπου `int`, μπορεί να είναι και `char`.
  - **unsigned char** ή **char** μας δίνει 8 bits
- Το **αποτέλεσμα** θα φέρει σε κάθε περίπτωση τον τύπο του **αριστερού μέλους**
- Προσοχή πρέπει να δίνεται στις **προτεραιότητες** των τελεστών:
  - `i << 2 + 1` σημαίνει **`i << (2 + 1)`**, όχι **`(i << 2) + 1`**
  - Εφόσον οι δυαδικοί τελεστές έχουν **χαμηλότερη προτεραιότητα από τους αριθμητικούς τελεστές** (δες επόμενη διαφάνεια)

Τελεστές Ολίσθησης (Bitwise Shift Operators)

Operator	Description	Associativity
()	Parentheses (function call) (see Note 1)	left-to-right
[]	Brackets (array subscript)	
.	Member selection via object name	
->	Member selection via pointer	
++ --	Postfix increment/decrement (see Note 2)	
++ --	Prefix increment/decrement	right-to-left
+ -	Unary plus/minus	
! ~	Logical negation/bitwise complement	
(*type*)	Cast (change *type*)	
*	Dereference	
&	Address	
sizeof	Determine size in bytes	
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <=	Relational less than/less than or equal to	left-to-right
> >=	Relational greater than/greater than or equal to	
== !=	Relational is equal to/is not equal to	left-to-right



http://www.difranco.net/cop2220/op-prec.htm

Επιπλέον Δυαδικοί Τελεστές { ~, &, ^, | }

- **Προτεραιότητες:**

Ψηλότερη: **~**

>> <<

&

^

Χαμηλότερη: **|**

- Παραδείγματα:

`i & ~j | k` **σημαίνει** `(i & (~j)) | k`

`i ^ j & ~k` **σημαίνει** `i ^ (j & (~k))`

- Κάνοντας ορθολογιστική χρήση των παρενθέσεων θα μας σώσει από πολλά λογικά λάθη.

Επιπλέον Δυαδικοί Τελεστές { ~, &, ^, | }

- Υπάρχουν τέσσερις επιπρόσθετοι τελεστές:
  - ~ **Δυαδικό Συμπλήρωμα** (bitwise complement)
  - & **Δυαδική Σύζευξη** (bitwise AND)
  - | **Δυαδική Διάζευξη** (bitwise inclusive *OR)*
  - ^ **Δυαδική Αποκλειστική Διάζευξη** (bitwise X*OR)*
- Ο τελεστής **~ είναι μοναδιαίος**, ενώ οι **υπόλοιποι είναι δυαδικοί**.
  - Οι τελεστές αυτοί εφαρμόζουν τις **γνωστές Boolean** πράξεις απευθείας πάνω στη **δυαδική ακολουθία** του τυπου (**ακεραίου ή χαρακτήρα)**
    - Διαφορετικό από &&, || που είναι λογικοί τελεστές σύγκρισης
    - Θυμηθείτε ότι το XOR επιστρέφει 1 εάν το ένα εκ' των δυο bit είναι 1 αλλά **όχι και τα δυο**.

Επιπλέον Δυαδικοί Τελεστές { ~, &, ^, | }

- Παραδείγματα χρήσης των ~, &, ^, και | :

```
unsigned short i, j, k;
i = 21;
   /* i is now    21 (binary 0000000000010101) */
j = 56;
   /* j is now    56 (binary 0000000000111000) */
k = ~i; // flip all bits
   /* k is now 65514 (binary 1111111111101010) */
k = i & j; // εύρεση κοινών bits …
   /* k is now    16 (binary 0000000000010000) */
k = i | j; // το ένα ή το άλλο (ή και τα δυο)
   /* k is now    61 (binary 0000000000111101) */

k = i ^ j; // είτε το ένα ή το άλλο (όχι και τα δυο)
   /* k is now    45 (binary 0000000000101101) */
```

Επιπλέον Δυαδικοί Τελεστές { ~, &, ^, | }

- Ας δούμε κάποιες **απλές** πρακτικές χρήσεις των τελεστών αυτών
  - Σε λίγο θα δούμε κάποιες συχνές ρουτίνες

Παράδειγμα (Συμπλήρωμα)

- Ο τελεστής **~** μπορεί να χρησιμοποιηθεί για να κάνει ακόμη και χαμηλού επιπέδου προγράμματα **μεταφέρσιμα**
  - π.χ., Θέλουμε ένα **flag vector**, αρχικοποιημένο σε «1» για όλες τις θέσεις: `unsigned int a = ~0;`
  - π.χ., Θέλουμε ένα **flag vector**, αρχικοποιημένο σε «1» για όλες τις θέσεις εκτός τις τελευταίες πέντε θέσεις.
  - `unsigned int a = 0x1F;` // Δεκαεξαδική αναπαράσταση, θυμηθείτε ότι $0x00_{16} = 0000\ 0000_2$, $0xFF_{16} = 1111\ 1111_2$ ($1F_{16} = 0001\ 1111_2$)
  - `unsigned int a = ~0x1F;` ($1110\ 0000_2$)

Σύνθετοι Δυαδικοί Τελεστές {&=, ^=, |=}

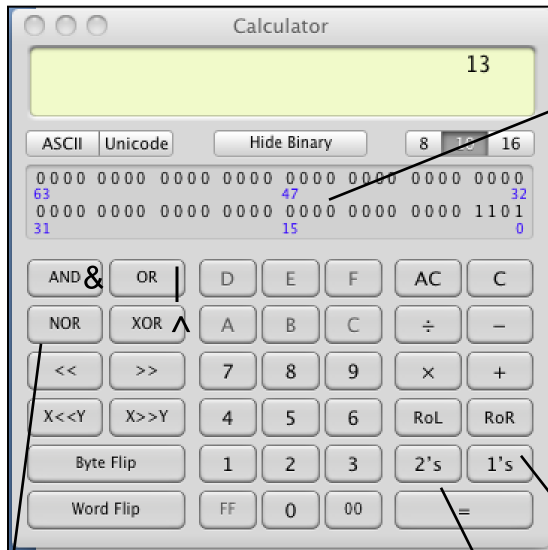
- Οι **σύνθετοι τελεστές ανάθεσης (compound assignment operators)** &=, ^=, και |= ανταποκρίνονται στους δυαδικούς τελεστές &, ^, και |:

```
i = 21;
   /* i is now 21 (binary 0000000000010101) */
j = 56;
   /* j is now 56 (binary 0000000000111000) */
i &= j; // άρα αποφεύγω k = i & j, αλλά χάνω το i
   /* i is now 16 (binary 0000000000010000) */
i ^= j; // Χρησιμοποιείται το νέο i ...
   /* i is now 40 (binary 0000000000101000) */
i |= j;
   /* i is now 56 (binary 0000000000111000) */
```

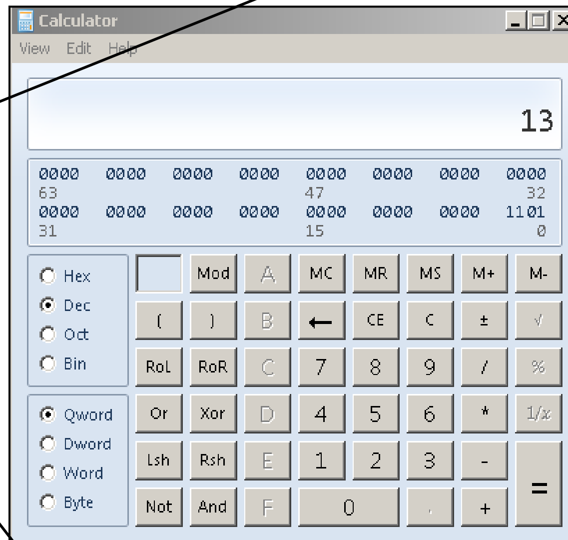
Υπολογιστική για Προγραμματιστές

- Ένα σημαντικότατο εργαλείο για ένα προγραμματιστή είναι μια εξειδικευμένη **αριθμομηχανή** για **προγραμματιστές**!

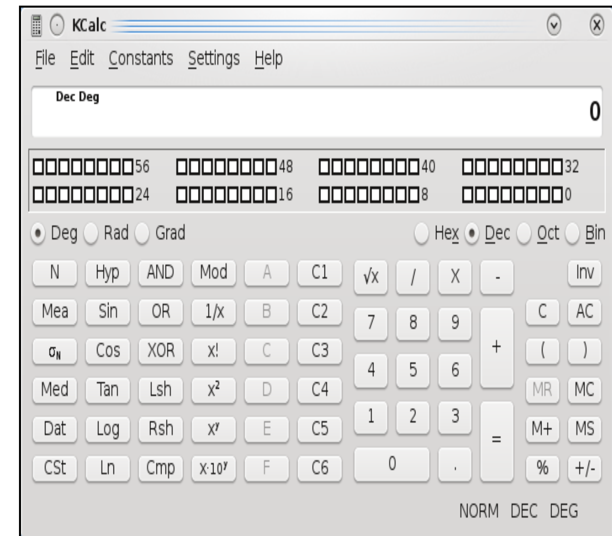
Bit viewer



Calculator (MacOSX) Programmer View



Calculator (Win7) Programmer View



Kcalc (Linux)

NOR = ~(P | Q)
True if both inputs are false

~b+1 *~b (complement)*

Συχνές Δυαδικές Πράξεις

- Οι δυαδικοί τελεστές μπορούν να χρησιμο-ποιηθούν για να εξάγουμε ή μεταβάλουμε επιλεκτικά κάποια δεδομένα
- **Συχνές Δυαδικές Πράξεις:**
  - **Ανάθεση** ενός bit (Setting)
  - **Μηδενισμός** ενός bit (Clearing)
  - **Δοκιμή** ενός bit (Testing)
- **Υποθέσεις (χωρίς απώλεια της γενικότητας)**:
  - `i` είναι 16-bit `unsigned short` μεταβλητή. `i=0x0000`
    - Εάν θέλαμε 32-bit, τότε `unsigned int`, δηλ., `i=0x00000000`
  - Το αριστερότερο— ή ***most significant***—bit θα το ονομάσουμε bit 15 και το least significant ως bit 0.

Συχνές Δυαδικές Πράξεις (Setting a bit)

- **Ανάθεση bit (i |= mask)**

Ο ευκολότερος τρόπος για να τεθεί το bit 4 (πέμπτο από δεξιά) του i είναι να γίνει OR η τιμή του i με **σταθερή τιμή** $0x0010_{16}$ $(0000\ 0000\ 0001\ 0000_2)$, π.χ.,

```
i |= 0x0010;     /* i is now ???? ???? ???1 ???? */
```

Επομένως χρησιμοποιώ δυνάμεις του 2:

```
0x8000(32768), 0x4000(16384),    0x2000(8192),  0x1000(4096),
0x800(2048),   0x400(1024),      0x200(512),    0x100(256),
0x80(128),     0x40(64),         0x20(32),      0x10(16),
0x8(8),        0x4(4),           0x2(2),        0x1(1)
```

- **Δημιουργία Μάσκας (1<<j)**
  - Εάν η θέση του bit που θέλουμε να μεταβάλουμε βρίσκεται σε **μεταβλητή j**, μπορούμε πρώτα να **φτιάξουμε μια μάσκα** και μετά την **ανάθεση με το |=**

```
i |= 1 << j;    /* sets bit j */
```

    - Π.χ., εάν j=3, τότε 1 << j είναι το 0x0008. $(0000\ 0000\ 0000\ 1000_2)$

Συχνές Δυαδικές Πράξεις (Testing a bit)

- ***Δοκιμή ενός bit* `(i & mask)`**
  - Μια έκφραση `if` δοκιμάζει κατά πόσο το **bit 4** (πέμπτο από δεξιά) του `i` έχει τεθεί:

```
if (i & 0x0010) …    /* tests bit #4 */
```

    - Αντίστοιχα με πριν, μια έκφραση που ελέγχει κατά πόσο το **bit `j`** του `unsigned int` **i** έχει τεθεί:

```
if (i & (1 << j)) …    /* tests bit j */
```

  - Μια έκφραση δοκιμάζει κατά πόσο τα **τελευταία 4 bit** του i έχουν τεθεί.

```
if (i & (1 | (1<<1) | (1<<2) | (1<<3))
```

ή καλύτερα

```
if (i & 15) ή (i & 0xF) /* 15=...01111 */
```

Συχνές Δυαδικές Πράξεις (Clearing a bit)

- **Μηδενισμός bit** `(i &= ~mask)`
- Για να μηδενίσουμε το **πέμπτο bit** του `i` απαιτεί μια μάσκα (mask) με ένα 0 bit στη θέση 4 και 1 bits στο υπόλοιπο:

```
Αρχικοποίηση i

i = 0x00ff;  /* π.χ. αρχικοποίηση i:0000 0000 1111 1111 */
i &= ~0x0010; /* Η μάσκα: ~0x0010 = 1111 1111 1110 1111 */
                                   &--------------------
/* Τελική Τιμή του i:               0000 0000 1110 1111 */
```

- **Αντίστοιχα** με το **παράδειγμα της ανάθεσης**, για να μηδενίσουμε ένα **αυθαίρετο bit** του οποίου η θέση αποθηκεύεται σε μεταβλητή **j**:

```
i &= ~(1 << j);      /* μηδενίζει το bit j */
```

Ονόματα σε Μάσκες

- Είναι **ευκολότερο**, και πιο **κατανοητό** εάν αποφεύγονται οι **hard coded μάσκες**.
- Ειδικότερα, είναι πιο ξεκάθαρο εάν χρησιμοποιούνται **ονόματα στις μάσκες**
- Υποθέστε για παράδειγμα ότι τα bits **0, 1**, and **2** ενός ακεραίου αντιστοιχούν στα **χρώματα blue, green, και red**, αντίστοιχα, τότε **δηλώνουμε**:

```
#define BLUE  1
#define GREEN 2
#define RED   4
```

*** οι σταθερές είναι δυνάμεις του 2 (όπως δείξαμε πριν)**

Ονόματα σε Μάσκες

- Ανάθεση, Μηδενισμός και Δόκιμη του `BLUE` bit:

```
i |= BLUE;            /* θέτει BLUE bit    */
if (i & BLUE) …       /* δοκιμάζει BLUE bit  */
i &= ~BLUE;           /* μηδενίζει BLUE bit */
```

- Ορισμός Πολλαπλών bits ταυτόχρονα::

```
i |= BLUE | GREEN;        /* θέτει BLUE και GREEN */
i &= ~(BLUE | GREEN);     /* μηδεν. BLUE και GREEN */
if(i & (BLUE | GREEN))    /* δοκιμ. BLUE και GREEN */
```