Податочни типови

Бинарен броен систем
Податочни типови
Локални променливи
Проследување на аргументи
Вредности со и без знак

Бинарен формат фиксна запирка (знак-магнитуда)

- Базата е 2, положбата на запирката во зборот е фиксна:

$$a_n a_{n-1} \cdots a_0 . a_{-1} \cdots a_{-m} \Rightarrow A = (-1)^{a_n} \sum_{i=-m}^{n-1} a_i 2^i$$

чекор на квантизација: $q = 2^{-m}$

опсег: $-(2^n - q) \leq A \leq 2^n - q$

Само дел од децималните вредности имаат конечна бинарна претстава!

негативност:

две 0, + и –

внимавај на битот за знак

бинарна в.	знак-маг.	неознач.
000	0	0
001	1	1
010	2	2
011	3	3
100	-0	4
101	-1	5
110	-2	6
111	-3	7

Бинарен формат фиксна запирка (комплемент на 1)

- Негативните броеви се добиваат со комплементирање на позитивните
  - собирањето се изведува исто како кај броеви без предзнак
  - негативност:
    - при собирањето битот за пренос се додава на сумата
    - две 0, + и –

бинарна в.	комп. 1	неознач.
000	0	0
001	1	1
010	2	2
011	3	3
111	-0	7
110	-1	6
101	-2	5
100	-3	4

Бинарен формат фиксна запирка (комплемент на 2)

- Негативните броеви се добиваат со комплементирање на позитивните и додавање 1
  - собирањето се изведува исто како кај броеви без предзнак
  - претворање (2): најди ја првата 1 од десно и комплементирај ги цифрите лево од неа

бинарна в.	комп. 2	неознач.
000	0	0
001	1	1
010	2	2
011	3	3
111	-1	7
110	-2	6
101	-3	5
100	-4	4

Бинарен формат фиксна запирка (комплемент на 2) (2)

- проширувањето се прави со дополнување со битот за знак од лево

  -3: 1101 → 1111 1101

- поместување во лево: додај 0 од десната страна
- поместување во десно: додај го битот за знак лево
- комплемент на 2 од најмалиот број е истиот број

Бинарен формат фиксна запирка (комплемент на 2) (3)

- За проверка на преполнување се користат битовите за пренос на позициите на битот за знак и битот лево од него. Ако и во двете позиции има иста вредност, резултатот е коректен. Битот за пренос се игнорира

Пример:

0111 (7)	0111 (7)
1101 (-3)	0011 (3)
0100 (4)	1010 (-6)
1111 прен.	0111 прен.

Бинарен формат фиксна запирка (комплемент на 2) (4)

- Множење: удвои ја прецизноста, игнорирај го битот за пренос надвор од двојната прецизност

```
   00000101 (5)
× 11111010 (−6)
          0
        101
          0
       101
      101
     101
    x01
  xx1          .
 xx11100010 (−30)
```

Неоптимално:

Сите собирања се 8 битни

Има 2x повеќе множења

Бинарен формат фиксна запирка (комплемент на 2) (5)

- Алгоритам “додај и помести”: додај го (4-битниот) парцијален производ на акумулираниот производ и помести во десно. Парцијалниот производ од битот за знак извади го од акумулираниот производ и помести во десно

  0101 (5)
  ×1010 (−6)
  0000 0000 (парцијален производ 1)
  0000 0000 (помести десно)
  0101 0000 (додај го парцијалниот производ 2)
  0010 1000 (помести десно)
  0010 1000 (додај го парцијалниот производ 3)
  0001 0100 (помести десно)
  1100 0100 (одземи го парц. производ од битот за знак)
  1110 0010 (помести десно) резултат −30

Бинарен формат подвижна запирка

- Типично се пакуваат како битови за знак, експонент и мантиса

$$a_n a_{e-1} \cdots a_0 a_{-1} \cdots a_{-m}$$

  - Најлевиот бит од мантисата е секогаш 1

$$M = \sum_{i=-1}^{-m} a_i 2^i \qquad 0{,}5 \le M < 1 \qquad -2^{e-1} \le E \le 2^{e-1}$$

  - Опсег на апсолутни вредности

$$\frac{1}{2} 2^{-2^{e-1}} \le |A| \le \left(1 - 2^{-m}\right) 2^{2^{e-1}}$$

Бинарен формат подвижна запирка

- Компјутерско претставување (стандард IEEE 754)

	знак	експонент	офсет на екс.	мантиса	вкупно
single	1	8	(127)	23	32
double	1	11	(1023)	52	64

  - Вредностите на експонентот 0 и сите 1 се резервирани за специјална намена. Опсегот на вредности е [-126, 127] и [-1022, 1023] соодветно
  - Мантисата: најлевиот бит е 1 и не се запишува

π: 3.14159265358979323846264338327795 точна вредност

3.141592741012573242187 5 запишана вредност

11.00100100001111110110 11

0 | 10000000 | 10010010000111111011011

знак | екс.128→1 | мантиса: левата единица се подразбира

Податочни типови (1)

- Податочниот тип определува
  - како вредностите за даден податок се сместуваат во меморијата,
  - множеството вредности за податокот (зависи од платформата и од преведувачот), и
  - операциите што може да се извршат со или врз неговите вредности.
- Поделба
  - Стандардни и изведени (корисничко дефинирани, композитни)
    - Стандардни: int, char, float, double, …
    - Изведени: набројувачки, полиња, записи итн.
  - Едноставни и сложени
- Операции:
  - Аритметички, логички и релациски

Податочни типови (2)

- целобројни вредности
  - char,int, short, long, unsigned, ...
  - Ограничување на множеството вредности

Податочен тип	Големина	Вредности
char	8-bit byte	unsigned 0 ÷ 255 signed -128 ÷ 127
short	16-bit halfword	unsigned 0 ÷ 65535 signed -32768 ÷ 32767
Int	32-bit word	unsigned 0 ÷ 4294967296 signed -2147483648 ÷ 2147483647
long	64-bit double word	unsigned 0 ÷ 18446744073709551616 signed -… ÷ …

Податочни типови (3)

- реални вредности
  - float, double, ...
  - Ограничување на множеството вредности
    - Големина на броевите и точност на броевите.

Податочен тип	**float**	**double**
големина	32-bit word	64-bit double word
# of bits in mantissa:	24 (1 sign bit)	53 (1 sign bit)
# of decimal digits of precision:	6	15
smallest such that 1.0+DBL_EPSILON != 1.0:	1e-7	1e-16
max decimal exponent:	38	308
min decimal exponent:	-38	-308

Податочни типови (4)

- знаковни вредности
  - елементи на ASCII кодната шема
  - знаците се задаваат со ‘a’, ‘!’, …
  - низите од знаци со “abc”, “a”, “\n”, …

Константи

- Нумерички константи
  - Целобројни константи
    - Децимални константи: 1, 0, 2, 234
    - Октални константи: 012
    - Хексадецимални константи: 0x23
  - Реални константи
    - Децимална и експоненцијална нотација
    - Броевите со (.) и (e) се третираат како double
    - float се означуваат со F после бројот: 3.5F, 1e-7F
  - long се означуваат со L после бројот: 890L е long int, 12.0L е long double

Хардверска поддршка

- Инструкции за вчитување и запишување кај ARM архитектурите:

Архитектура	Инструкција	Функција
Pre-ARMv4	LDRB STRB LDR STR	load an unsigned 8-bit value store a signed or unsigned 8-bit value load a signed or unsigned 32-bit value store a signed or unsigned 32-bit value
ARMv4	LDRSB LDRH LDRSH STRH	load a signed 8-bit value load an unsigned 16-bit value load a signed 16-bit value store a signed or unsigned 16-bit value
ARMv5	LDRD STRD	load a signed or unsigned 64-bit value store a signed or unsigned 64-bit value

  - При вчитување на 8-битна или 16-битна вредност во 32-битен регистар автоматски се прави проширување со битот за знак
  - При запишување на 8-битна или 16-битна вредност од регистар автоматски зе земаат само најниските битови
  - cast операцијата не одзема време

Хардверска поддршка (2)

- Кај преведувачите *armcc* и *gcc* :

Податочен тип	имплементација
char	unsigned 8-bit byte
short	signed 16-bit halfword
int	signed 32-bit word
long	signed 32-bit word
long long	signed 64-bit double word

  - Кај постарите верзии на ARM процесорите вчитувањето на unsigned char беше хардверски поддржано
  - Проблематично при портирање
    - char бројач неможе да е негативен (постојано исполнет услов i>=0)

Алоцирање на регистрите

- Преведувачот се обидува локалните променливи да ги чува во регистрите на процесорот
- Доколку не се преклопуваат ќе алоцира исти регистри за различни локални променливи
- Ако не е можно сите да се сместат во регистрите ги става на стекот (во меморијата)
- За поголема ефикасност
  - се намалува бројот на локални променливи
  - најчесто користените се сместуваат во регистрите
  - бројот на локални променливи во јамките треба да е што помал
- На преведувачот може да му се сугерира кои променливи да ги смести во регистри со клучниот збор register, но обично сугестиите се помалку оптимални од изборот на преведувачот

Локални променливи

- 8-битни и 16-битни променливи кај 32-битен процесор:
  - + штеди меморија (?)
  - - ја успорува работата

```
int checksum_v1(int *data)
{
   char i;
   int sum = 0;
   for (i = 0; i < 64; i++)
   {
      sum += data[i];
   }
   return sum;
}
```

```
checksum_v1
        MOV r2,r0               ; r2 = data
        MOV r0,#0               ; sum = 0
        MOV r1,#0               ; i = 0
checksum_v1_loop
        LDR r3,[r2,r1,LSL #2]   ; r3 = data[i]
        ADD r1,r1,#1            ; r1 = i+1
        AND r1,r1,#0xff         ; i = (char)r1
        CMP r1,#0x40            ; compare i, 64
        ADD r0,r3,r0            ; sum += r3
        BCC checksum_v1_loop ; if (i<64) loop
        MOV pc,r14              ; return sum
```

Локални променливи (2)

- Ефикасен код:
  armcc compiler

```
int checksum_v2(int *data)
{
   int i;
   int sum = 0;
   for (i = 0; i < 64; i++)
   {
      sum += data[i];
   }
   return sum;
}
```

```
checksum_v2
        MOV r2,r0                ; r2 = data
        MOV r0,#0                ; sum = 0
        MOV r1,#0                ; i = 0
checksum_v2_loop
        LDR r3,[r2,r1,LSL #2]    ; r3 = data[i]
        ADD r1,r1,#1             ; r1 = i+1

        CMP r1,#0x40             ; compare i, 64
        ADD r0,r3,r0             ; sum += r3
        BCC checksum_v2_loop ; if (i<64) loop
        MOV pc,r14               ; return sum
```

Локални променливи (3)

- 8-битни и 16-битни променливи кај 32-битен процесор:
  IAR compiler

```
int checksum_v1(int *data)
{
  char i;
  int sum = 0;
  for (i = 0; i < 64; i++)
  {
    sum += data[i];
  }
  return sum;
}
```

```
checksum_v1:
        MOVS  R1, R0            ; R1=data
        MOV   R3, #0x0
        MOVS  R0, R3            ; int sum = 0
        MOV   R3, #0x0
        MOVS  R2, R3            ; i = 0
checksum_v1_0:
        ANDS  R2, R2, #0xFF
        CMP   R2, #0x40         ; compare i, 64
        BGE   checksum_v1_1     ; if(i>=64) goto out
        LDR   R3, [R1,#+0]      ; R3 = *data
        ADDS  R0, R3, R0        ; sum+=R3
        ADDS  R1, R1, #0x4      ; data++
        ADDS  R2, R2, #0x1      ; i++
        B     checksum_v2_0     ; loop
checksum_v1_1:
        BX LR                   ; return
```

Локални променливи (4)

- Ефикасен код:
  IAR compiler

```
int checksum_v2(int *data)
{
  int i;
  int sum = 0;
  for (i = 0; i < 64; i++)
  {
    sum += data[i];
  }
  return sum;
}
```

```
checksum_v2:
        MOVS  R1, R0            ; R1=data
        MOV   R3, #0x0
        MOVS  R0, R3            ; int sum = 0
        MOV   R3, #0x0
        MOVS  R2, R3            ; i = 0
checksum_v2_0:

        CMP   R2, #0x40         ; compare i, 64
        BGE   checksum_v2_1     ; if(i>=64) goto out
        LDR   R3, [R1,#+0]      ; R3 = *data
        ADDS  R0, R3, R0        ; sum+=R3
        ADDS  R1, R1, #0x4      ; data++
        ADDS  R2, R2, #0x1      ; i++
        B     checksum_v2_0     ; loop
checksum_v2_1:
        BX LR                   ; return
```

Локални променливи (5)

- 8-битни и 16-битни променливи кај 32-битен процесор:

```
short checksum_v3(short *data)
{
  unsigned int i;
  short sum = 0;
  for (i = 0; i < 64; i++)
  {
    /* sum += data[i]; */
    sum = (short)(sum + data[i]);
  }
  return sum;
}
```

```
checksum_v3
        MOV R2,R0           ; R2 = data
        MOV R0,#0           ; sum = 0
        MOV R1,#0           ; i = 0
checksum_v3_loop
        ADD R3,R2,R1,LSL #1 ; R3 = &data[i]
        LDRH R3,[R3,#0]     ; R3 = data[i]
        ADD R1,R1,#1        ; i++
        CMP R1,#0x40        ; compare i, 64
        ADD R0,R3,R0        ; R0 = sum + R3
        MOV R0,R0,LSL #16
        MOV R0,R0,ASR #16 ; sum = (short)R0
        BCC checksum_v3_loop ; if (i<64) goto loop
        MOV pc, R14         ; return sum
```

Локални променливи (6)

- Ефикасен код:

  armcc compiler

```
short checksum_v4(short *data)
{
  unsigned int i;
  int sum=0;
  for (i = 0; i < 64; i++)
  {
    /*sum += *data; data++;*/
    sum += *(data++);
  }
  return (short)sum;
}
```

```
checksum_v4
        MOV R2, #0              ; sum = 0
        MOV R1, #0              ; i = 0

checksum_v4_loop
        LDRSH R3, [R0],#2       ; R3 = *(data++)
        ADD R1, R1, #1          ; i++
        CMP R1, #0x40           ; compare i, 64
        ADD R2, R3, R2          ; sum += R3

        BCC checksum_v4_loop;if(sum<64) goto loop
        MOV R0, R2, LSL #16
        MOV R0,R0,ASR #16 ; r0 = (short)sum
        MOV pc, R14             ; return r0
```

Локални променливи (7)

- Ефикасен код*:
  IAR compiler

```
short checksum_v4(short *data)
{
  unsigned int i;
  int sum=0;
  for (i = 0; i < 64; i++)
  {
    /*sum += *data; data++;*/
    sum += *(data++);
  }
  return (short)sum;
}
```

```
checksum_v4:
        MOVS  R1, R0            ; R1=data
        MOV   R3, #0x0
        MOVS  R0, R3            ; int sum = 0
        MOV   R3, #0x0
        MOVS  R2, R3            ; i = 0
checksum_v4_0:
        CMP   R2, #0x40         ; compare i, 64
        BCS   checksum_v4_1     ; if(i==64) goto out
        LDRSH R3, [R1,#+0]      ; R3 = *data
        ADDS  R0, R3, R0        ; sum+=R3
        ADDS  R1, R1, #0x2      ; data++
        ADDS  R2, R2, #0x1      ; i++
        B     checksum_v4_0     ; loop
checksum_v4_1:
        MOV   R0, R0, LSL #16
        MOVS  R0,R0,ASR #16     ; R0 = (short)sum
        BX LR                   ; return
```

* IAR comp. во двата случаи дава ист код

Проследување на аргументи

- 8-битни и 16-битни променливи кај 32-битен процесор:
  armcc compiler

```
short add_v1(short a, short b)
{
  return a + (b>>1);
}
```

```
add_v1
        ADD R0,R0,R1,ASR #1 ; R0 = (int)a+((int)b >> 1)
        MOV R0,R0,LSL #16
        MOV R0,R0,ASR #16   ; R0 = (short)R0
        MOV pc,R14                  ; return R0
```

Проследување на аргументи (2)

- 8-битни и 16-битни променливи кај 32-битен процесор:
  gcc compiler

```
short add_v1(short a, short b)
{
  return a + (b>>1);
}
```

```
add_v1_gcc
        MOV    R0, R0, LSL #16
        MOV    R1, R1, LSL #16
        MOV    R1, R1, ASR #17          ; R1 = (int)b >> 1
        ADD    R1, R1, R0, ASR #16      ; R1 += (int)a
        MOV    R1, R1, LSL #16
        MOV    R0, R1, ASR #16          ; R0 = (short)R1
        MOV    pc, LR                   ; return R0
```

Проследување на аргументи (3)

- 8-битни и 16-битни променливи кај 32-битен процесор:
  IAR compiler

```
short add_v1(short a, short b)
{
  return a + (b>>1);
}
```

```
add_v1:
        MOV    R0, R0, LSL #16
        MOVS   R0, R0, ASR #16        ; R0 = (int)a
        MOV    R1, R1, LSL #16
        MOVS   R1, R1, ASR #16        ; R1 = (int)b
        ADDS   R0, R0, R1, ASR #1     ; R0 += b >> 1
        MOV    R0, R0, LSL #16
        MOV    R0, R0, ASR #16        ; R0 = (short)R0
        BX     LR                     ; return R0
```

Вредности со или без знак?

- Операциите собирање, одземање и множење се изведуваат еднакво брзо за целобројни вредности со и без знак
- Кај делењето кое се изведува со поместување постои разлика:
  - -3 >> 1 = -2
  - -3 / 2 = -1
  - За правилно делење треба: (x<0) ? ((x+1) >> 1): (x >> 1)

```
int average_v1(int a, int b)
{
   return (a+b)/2;
}
```

```
average_v1
        ADD R0, R0, R1              ; R0 = a + b
        ADD R0, R0, R0, LSR #31     ; if (R0<0) r0++
        MOV R0, R0, ASR #1          ; R0 = R0 >> 1
        MOV pc, R14                 ; return R0
```

Вредности со или без знак? (2)

- IAR compiler
  - без оптимизација и со ниско ниво на оптимизација повикува процедура за 32-битно делење
  - со средно ниво на оптимизација се генерира следниот код

```
int average_v1(int a, int b)
{
  return (a+b)/2;
}
```

```
average_v1:
    ADD R0, R1, R0              ; R0 = a + b
    ADD R0, R0, R0, LSR #31     ; if (R0<0) r0++
    MOV R0, R0, ASR #1          ; R0 = R0 >> 1
    BX  LR                      ; return R0
```

```
int average_v2(unsigned int a,
unsigned int b)
{
  return (a+b)/2;
}
```

```
average_v2:
    ADD R0, R1, R0              ; R0 = a + b
    MOV R0, R0, LSR #1          ; R0 = R0 >> 1
    BX  LR                      ; return R0
```

Резиме

- Локални променливи
  - најчесто се чуваат во регистрите;
  - доколку не е неопходна аритметика со помалку битови, користи променливи со големина на регистрите
    - на 32-битна платформа int наместо char или short
- За забрзување на делењето, ако е возможно, користи променливи без знак (unsigned)
- Полиња и глобални променливи
  - најчесто се чуваат во меморијата;
  - користи променливи со оптимална големина во поглед на меморијата;
  - избегнувај адресирање кое користи адресни модови кои не се хардверски поддржани;

Резиме (2)

- Користи експлицитно кастирање при запишување од полиња и глобални променливи во локални променливи и обратно
  - при копирање во и од регистрите кастирањето е хардверско и не троши додатно време
  - со тоа се појаснува дека користиш “мали” промеливи за чување на податоците и “големи” променливи за забрзување на операциите
  - вклучи го предупредувањето за имплицитно кастирање
- Избегнувај кастирање во изрази
  - кастирањето во изразите троши додатно време
- Аргументи на функции
  - користи променливи со големина на регистрите за избегнување на додатни каст операции
- Бинарните формати дел од реалните броеви ги претставуваат приближно
  - не користи егзактно споредување на реални вредности