Módulo 04
Aritmética de Punto Fijo
(Pt. 2)

Organización de Computadoras
Depto. Cs. e Ing. de la Comp.
Universidad Nacional del Sur

Copyright

Contenidos

- Clasificación de las operaciones
- Codificación decimal en binario (**BCD**)
- Representación **SM**
- Representación **RC**
- Representación **DRC**
- Operaciones de suma y de resta
- Detección de overflow
- Otras codificaciones

Suma en SM

- Para sumar dos números codificados en **SM** se aplica el siguiente algoritmo:
  - Sean $X = (X_{n-1} \ \dots \ X_1 X_0)$ e $Y = (Y_{n-1} \ \dots \ Y_1 Y_0)$ los números a ser sumados
  - Si $X_{n-1} = Y_{n-1}$ (números de igual signo), se suman las magnitudes $(X_{n-2} \dots X_1 X_0)$ e $(Y_{n-2} \dots Y_1 Y_0)$
  - En este caso el signo del resultado será X_{n-1}
  - Si hay acarreo en la posición del signo, entonces el resultado es inválido pues se produjo un overflow

Suma en SM

- Continúa:
  - En cambio, si $X_{n-1} \neq Y_{n-1}$ se deben comparar las magnitudes y en caso que $|X| < |Y|$, se procede a intercambiar los valores **X** e **Y**
  - Independientemente de que se haya intercambiado o no, el resultado buscado se obtiene restando la magnitud $(Y_{n-2} \ldots Y_1 Y_0)$ a $(X_{n-2} \ldots X_1 X_0)$
  - En este caso el signo del resultado también será X_{n-1} (nótese que será el signo correcto, es decir, el signo del número más grande en valor absoluto)

Ejemplo

- Asumida una base **r = 2** y una precisión **n = 8**, se desean realizar las siguientes sumas entre números codificados en **SM**:
  - Con operandos de un mismo signo:

$$\begin{array}{r} (0\ {}^{0}0011001)_2 = +(25)_{10} \\ +(0\ \ 0111110)_2 = +(62)_{10} \\ \hline (0\ \ 1010111)_2 = +(87)_{10} \end{array}$$

  - Con operandos de distintos signo:

$$\begin{array}{r} (0\ 1010100)_2 = +(\ 84)_{10} \\ +(1\ 1111000)_2 = -(120)_{10} \\ \hline \end{array} \qquad \begin{array}{r} (1\ {}^{0}1111000)_2 = -(120)_{10} \\ -(0\ \ 1010100)_2 = +(\ 84)_{10} \\ \hline (1\ \ 0100100)_2 = -(\ 36)_{10} \end{array}$$

Detección de overflow

- Al operar en signo-magnitud sólo se puede producir overflow al sumar números de igual signo
  - El mecanismo de detección del overflow consiste en inspeccionar el acarreo a la posición del signo: si hubo acarreo, se debe descartar el resultado por inválido
  - En contraste, no es posible provocar un overflow al sumar números de distinto signo
  - En ese caso el resultado siempre será más pequeño en valor absoluto que el mayor de los operandos, el cual ya era representable en esa precisión

Ejemplo

- Asumida una base **r = 2** y una precisión **n = 8**, se desean realizar las siguientes operaciones entre números codificados en **SM**:
  - Con operandos positivos:

$$\begin{array}{r} (0\ {}^{1}1010111)_2 = +(87)_{10} \\ +(0\ \ 0111110)_2 = +(62)_{10} \\ \hline (0\ \ 0010101)_2 = \quad \text{OV} \end{array}$$

  - Con operandos negativos:

$$\begin{array}{r} (1\ {}^{1}0010100)_2 = -(\ 20)_{10} \\ +(1\ \ 1111110)_2 = -(126)_{10} \\ \hline (1\ \ 0010010)_2 = \quad \text{OV} \end{array}$$

Suma en DRC

- Para sumar dos números en complemento a la base disminuida se usa el siguiente algoritmo:
  - Sean **X** e **Y** los números a ser sumados y sean **S = (S$_{n-1}$ ... S$_1$S$_0$)** los dígitos del resultado y **C = (C$_n$C$_{n-1}$ ... C$_1$)** los acarreos que se generen
  - La suma preliminar se obtiene sumando la totalidad de los dígitos, signo incluido
  - Si **X$_{n-1}$ = Y$_{n-1}$ = 0** (ambos operandos positivos), **C$_n$** es necesariamente **0**, y el resultado preliminar es el definitivo, pero si **S$_{n-1}$ = 1**, se produjo overflow

Suma en DRC

- Continúa:
  - Si $X_{n-1} \neq Y_{n-1}$ (operandos de distinto signo), se estudia el acarreo de salida (C_n)
  - Si $C_n = 0$, el resultado preliminar es el definitivo
  - Si $C_n = 1$, se descarta el acarreo e incrementando en **1** al resultado preliminar se obtiene el definitivo
  - Finalmente, si $X_{n-1} = Y_{n-1} = 1$ (ambos operandos negativos), C_n es necesariamente **1**, y el resultado definitivo se obtiene incrementando en **1** al resultado preliminar, pero si $S_{n-1} = 0$, se produjo overflow

Ejemplo

- Asumida una base $r = 2$ y una precisión $n = 8$, se desean realizar las siguientes operaciones entre números codificados en **DRC**:
  - Con operandos positivos:

$$\begin{array}{r} (^{0}00101001)_2 = +(41)_{10} \\ +(\ 00100110)_2 = +(38)_{10} \\ \hline (\ 01001111)_2 = +(79)_{10} \end{array}$$

  - Como era de esperar, $C_n = 0$. Por otra parte, como $S_{n-1} = 0$, el resultado preliminar es el definitivo:

$$(01001111)_2 = +(79)_{10}$$

Ejemplo

- Continúa:
  - Con operandos de distinto signo:

$$\begin{array}{r} ({}^{1}11010110)_2 = -(41)_{10} \\ +(\ 01100010)_2 = +(98)_{10} \\ \hline (\ 00111000)_2 = +(56)_{10} \end{array}$$

  - En esta oportunidad, $C_n = 1$. Se descarta el acarreo de salida, pero se debe incrementar en **1** el resultado preliminar para obtener el definitivo:

$$(00111001)_2 = +(57)_{10}$$

Detección de overflow

- Al operar en complemento a la base disminuida sólo se puede producir overflow al sumar números de igual signo
  - El mecanismo de detección del overflow consiste en inspeccionar el bit de signo del resultado: en caso de no ser el esperado, se produjo un overflow
  - Esta detección equivale a computar $C_n \oplus C_{n-1}$. Verificar que está expresión cubre todos los casos
  - Al igual que en las restantes representaciones, no es posible provocar un overflow al sumar números de distinto signo

Ejemplo

- Asumida una base $r = 2$ y una precisión $n = 8$, se desea realizar la siguiente operación entre números codificados en **DRC**:
  - Con operandos negativos:

$$
\begin{array}{r}
(^{1}10110111)_2 = -(72)_{10} \\
+(\ 11000000)_2 = -(63)_{10} \\
\hline
(\ 01110111)_2 = \quad \text{OV}
\end{array}
$$

  - Como era de esperar, $C_n = 1$. No obstante, como el signo del resultado es incorrecto, se produjo overflow
  - Nótese que $C_n \oplus C_{n-1} = 1$, pues $C_n = 1$ y $C_{n-1} = 0$

Suma en RC

- Para sumar dos números en complemento a la base se usa el siguiente algoritmo:
  - Sean **X** e **Y** los números a ser sumados, sean **$S = (S_{n-1} \dots S_1 S_0)$** los dígitos del resultado y **$C = (C_n C_{n-1} \dots C_1)$** los acarreos que se generen
  - La suma definitiva se obtiene sumando la totalidad de los dígitos, signo incluido
  - Si **$X_{n-1} = Y_{n-1} = 0$** (ambos operandos positivos), **C_n** es necesariamente **0**, pero si **$S_{n-1} = 1$** (el signo del resultado es incorrecto), se produjo overflow

Suma en RC

- Continúa:
  - Si $X_{n-1} \neq Y_{n-1}$ (operandos de distinto signo), en caso de generarse acarreo de salida, se descarta
  - Finalmente, si $X_{n-1} = Y_{n-1} = 1$ (ambos operandos negativos), C_n es necesariamente 1, este acarreo también se descarta, pero si $S_{n-1} = 0$ (el signo del resultado es incorrecto), se produjo overflow

Ejemplo

- Asumida una base $r = 2$ y una precisión $n = 8$, se desean realizar las siguientes sumas entre números codificados en **RC**:
  - Con operandos positivos:

$$\begin{array}{r} (^{0}00010110)_2 = +(22)_{10} \\ +(\ 00010010)_2 = +(18)_{10} \\ \hline (\ 00101000)_2 = +(40)_{10} \end{array}$$

  - Como era de esperar, $C_n = 0$. A su vez, considerando que $S_{n-1} = 0$, el resultado es el correcto pues no se produjo overflow

Ejemplo

- Continúa:
  - Con operandos de distinto signo:

$$
\begin{array}{r}
(^{1}11010111)_2 = -(41)_{10} \\
+(\ 01100010)_2 = +(98)_{10} \\
\hline
(\ 00111001)_2 = +(57)_{10}
\end{array}
$$

  - En esta oportunidad, $C_n = 1$. Se descarta el acarreo de salida y como no se puede producir overflow con operandos de distinto signo el resultado obtenido es necesariamente correcto

Detección de overflow

- Al operar en complemento a la base sólo se puede producir overflow al sumar números de igual signo
  - El mecanismo de detección del overflow es análogo al anterior: se inspecciona el bit de signo del resultado y si no es el esperado es porque se produjo overflow
  - Esta detección equivale a computar $C_n \oplus C_{n-1}$
  - Recordemos que al igual que en las restantes representaciones, no es posible provocar overflow al sumar números de distinto signo

Ejemplo

- Asumida una base $r = 2$ y una precisión $n = 8$, se desea realizar la siguiente operación entre números codificados en **RC**:
  - Con operandos negativos:

$$
\begin{array}{r}
({}^{1}10000000)_2 = -(128)_{10} \\
+(\ 11111111)_2 = -(\ \ 1)_{10} \\
\hline
(\ 01111111)_2 = \quad \text{OV}
\end{array}
$$

  - Como era de esperar, $C_n = 1$. No obstante, el signo del resultado es incorrecto, se produjo overflow
  - Nótese que $C_n \oplus C_{n-1} = 1$, pues $C_n = 1$ y $C_{n-1} = 0$

Análisis

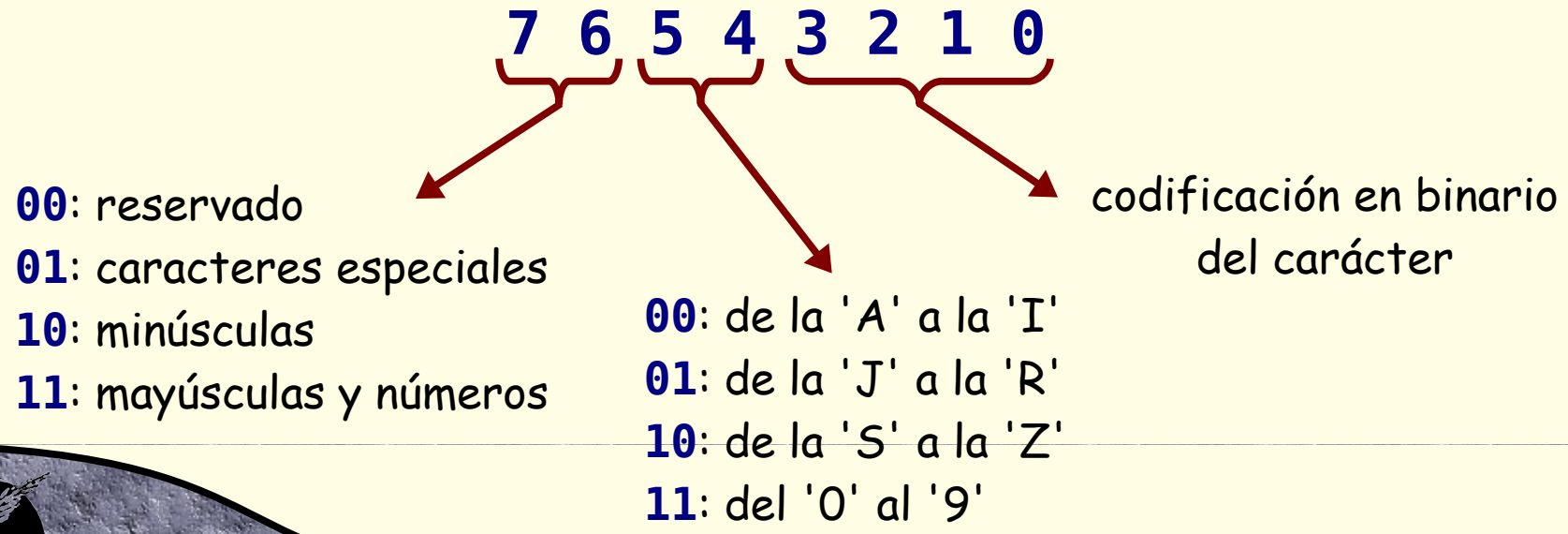
- Para completar el análisis comparativo de las tres representaciones alternativas nos resta estudiar la eficiencia de sus implementaciones:
  - Para **SM**, debe considerarse que para implementar una suma se debe llevar adelante una comparación de magnitudes; esta comparación se implementa a nivel de hardware analizando el signo de la diferencia entre las magnitudes
  - Es decir, toda vez que se opere en **SM** a veces se suma y otras veces se resta dos veces (lo que implica que hay que disponer de ambos circuitos en el hardware)

Análisis

- Continúa:
  - Para **DRC** sucede algo análogo: en el peor caso (esto es, cuando se produce acarreo de salida), para sumar dos magnitudes se debe realizar también dos operaciones sucesivas de suma
  - **RC** se destaca por ser la representación más eficiente, para realizar una suma simplemente se lleva adelante la suma propiamente dicha
  - Finalmente, en todos los casos se puede implementar la operación de resta usando el hardware asignado a la suma al combinarlo con la complementación

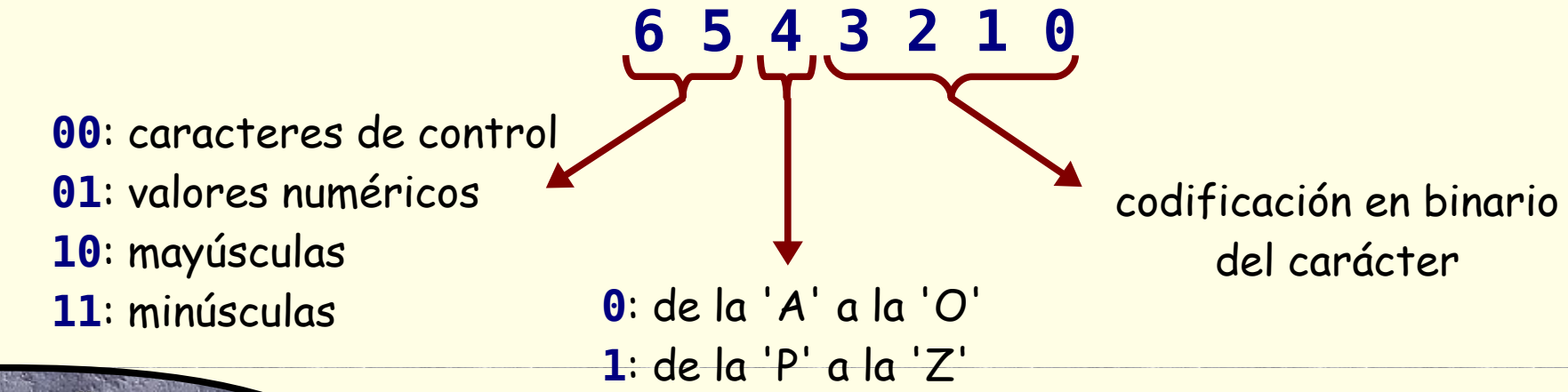
Codificación EBCDIC

- El código **EBCDIC** es una extensión del código **BCD** diseñado por **IBM** en la década del '60
  - Codifica directamente caracteres de texto, usando 8 bits por cada carácter
  - Los bits se organizan de la siguiente manera:



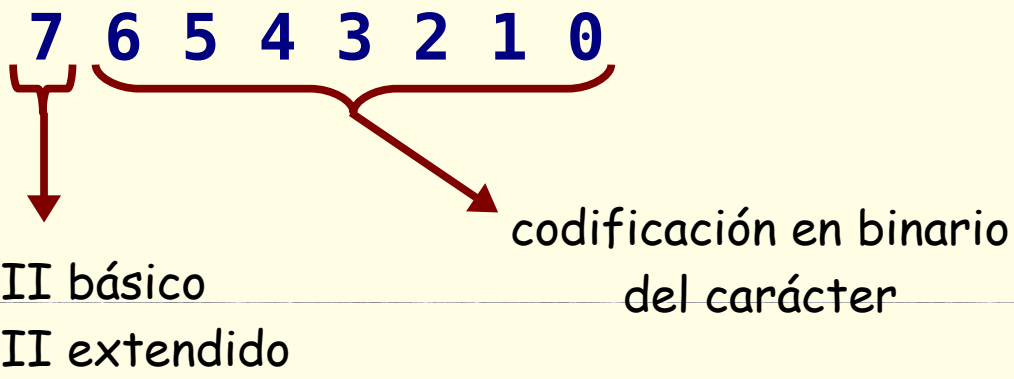
Codificación ASCII

- El código **ASCII** es un estándar también creado en los 60' específicamente para los americanos
  - Codifica cada carácter usando 7 bits, lo que permite 128 combinaciones (razón por la cual no contempla, entre otros, vocales acentuadas ni la letra eñe)
  - Los bits se organizan de la siguiente manera:



Codificación ASCII extendido

- El código **ASCII** extendido es una extensión del código **ASCII** original
  - Con el objeto de codificar algunos de los caracteres faltantes se incorpora un bit más por carácter
  - Existen múltiples extensiones (desde **ISO 8859-1** a **8859-15**), una para cada región o zona del planeta
  - Se organiza de la siguiente manera:



Codificación UNICODE

- El código **UNICODE** fue propuesto para unificar las codificaciones incompatibles existentes al momento de su concepción

  - Al tratarse de un estándar reciente, el comité que lo diseño tuvo la chance de corregir los inconvenientes identificados en los restantes códigos
  - Actualmente cubre 149.813 caracteres
  - Usa un esquema de codificación extensible, por lo que se puede seguir agregando nuevos caracteres
  - La última versión (**15.1**) es de septiembre del 2023

UTF-8

- La codificación **UTF-8** es una de las maneras que existe codificar el código **UNICODE**
  - La codificación se compone de una cantidad variable de bloques de 8 bits
  - Como objetivo de diseño se desea maximizar la compatibilidad con el código **ASCII**
  - Por esta razón, las primeras 128 codificaciones coinciden con el mapeo del código **ASCII**
  - De ahí en adelante entra en acción el esquema extensible, agregando nuevos bloques de 8 bits a medida que vayan siendo requeridos

UTF-16 y UTF-32

- Las codificaciones **UTF-16** y **UTF-32** aparecen como alternativas a la codificación **UTF-8**
  - **UTF-16** sólo codifica una porción del código **UNICODE** (¡solamente 1.112.064 caracteres!)
  - Hace uso de una cantidad variable de bloques de 16 bits
  - **UTF-32** codifica el mismo subconjunto que **UTF-16**
  - La principal diferencia es que hace uso de exactamente un único bloque de 32 bits.

¿Preguntas?