Fast Conversion From UTF-8 with C++, DFAs, and SSE Intrinsics

Bob Steagall

C++Now 2018

Overview

- Some definitions
- What is UTF-8?
- What is a DFA?
- The KEWB converter
- Some performance measurements
- Caveat – I am not a Unicode expert

Terminology

- Code Unit
  - A single, indivisible, integral element of an encoded sequence of characters
  - A sequence of one or more code units specifies a **code point**
  - A code unit does not, by itself, identify any particular character or code point
  - The meaning of a particular code unit value is derived from a character encoding
  - In C++11, `char`, `uint8_t`, `wchar_t`, `char16_t`, and `char32_t` are commonly-used code unit types

Terminology

- Encoding
  - A method of representing a sequence of characters as a sequence of code unit sub-sequences
  - An encoding may be stateless or stateful
  - An encoding may be fixed width or variable width
  - An encoding may support bidirectional or random access decoding of code unit sequences
  - Common encodings include:
    - UTF-8, UTF-16, and UTF-32
    - ISO/IEC 8859 series of encodings, including ISO/IEC 8859-1
    - Windows code page 1252

Terminology

- Code Point
  - An integral value denoting an abstract character as defined by a character set.
  - A code point does not, by itself, identify any particular character
  - The meaning ascribed to a particular code point value is derived from a character set definition
  - In C++11, `char`, `wchar_t`, `char16_t`, and `char32_t` are commonly-used code point types

Terminology

- Character Set
  - A mapping of code point values to abstract characters
  - A character set need not provide a mapping for every possible code point value representable by the code point type
  - Common character sets include ASCII, Unicode, and Windows code page 1252
- Character
  - An element of written language, for example, a letter, number, or symbol
  - For our purposes, a character is identified by the combination of a character set and a code point value

Terminology

- ISO 10646
  - An international standard that defines the **Universal Character Set** (**UCS**)
  - UCS is a superset of all other character set standards
  - Assigns a position and name to every character
  - Guarantees lossless round-trip compatibility with other standards
- Basic Multilingual Plane
  - Each 2^{16} subset of code points, beginning at `U+0000`, is called a **plane**
  - The first such plane, `U+0000..U+FFFF`, is called the **BMP** or **Plane 0**
  - The most commonly-used characters from older standards appear in the BMP
  - Only the first 17 planes will be used; all code points lie in `U+000000..U+10FFFF`

Terminology

- Unicode
  - An international standard from the Unicode Consortium that corresponds to ISO 10646 at implementation level 3
  - All characters have the same names and positions as ISO 10646
  - Defines semantics associated with some subsets of characters
- Unicode Transformation Format (UTF)
  - UTF-8, UTF-16, UTF-32 are three standardized transformations that use 8-bit, 16-bit, and 32-bit code units, respectively
- ISO 10646 –vs– Unicode in a Nutshell
  - ISO 10646 is mostly a character set table with some definitions
  - Unicode specifies algorithms for rendering presentation forms of some scripts, sorting and comparison, handling bi-directional texts, and more

So What is UTF-8?

- A variable-length scheme for encoding code points
- Each code point is encoded by a sequence of 1-4 code units of an 8-bit unsigned integer type (`uint8_t` or `unsigned char`) (*bytes*, *octets*)
- The first byte in a sequence indicates the total length of the sequence
- ASCII characters are encoded as `0x00..0x7F`
- The first byte in a multibyte sequence always ranges from `0xC2..0xF4`
- Trailing bytes in a multibyte sequence always range from `0x80..0xBF`

UTF-8 Bit Layout

1: 0xxx.xxxx

U+0000..U+007F : **7** bits, leading byte < 0x80

2: 110x.xxxx 10xx.xxxx

U+0080..U+07FF : **11** bits, leading bytes 0xC2-0xDF

3: 1110.xxxx 10xx.xxxx 10xx.xxxx

U+0800..U+FFFF : **16** bits, leading bytes 0xE0-0xEF

4: 1111.0xxx 10xx.xxxx 10xx.xxxx 10xx.xxxx

U+010000..U+1FFFFF : **21** bits, leading bytes 0xF0-0xF4

trailing bytes are always 0x80..0xBF {1000.0000..1011.1111}

Overlong Sequences

- UTF-8 sequences must be as short as possible to encode a character
    - Security implications – e.g., bypassing substring checks

```
2: 1100.000x 10xx.xxxx
```

```
      Leading bytes 0xC0 and 0xC1 are invalid
```

```
3: 1110.0000 100x.xxxx 10xx.xxxx
```

```
      Leading byte 0xE0 followed by byte b1 ≤ 0x9F is invalid
```

```
4: 1111.0000 1000.xxxx 10xx.xxxx 10xx.xxxx
```

```
      Leading byte 0xF0 followed by byte b1 ≤ 0x8F is invalid
```

Overlong Example

- Consider the closing brace character }
  - Hex: `0x7D`
  - Binary: `0111 1101`

```
1: 0111.1101
```

`Valid ASCII leading byte`

```
2: 1100.0001 1011.1101
```

`Invalid sequence 0xC1,0xBD`

```
3: 1110.0000 1000.0001 1011.1101
```

`Invalid sequence 0xE0,0x81,0xBD`

Boundary Conditions

- Maximum code point `U+10FFFF` (17 planes of 2^{16} code points per plane)

- UTF-16 surrogates range `U+D800..U+DFFF`
  - Leading/High: `0xD800..0xDBFF`
  - Trailing/Low: `0xDC00..0xDFFF`

- Overlong sequences
  - 2-byte: leading `0xC0` or `0xC1`
  - 3-byte: leading `0xE0` followed by `b1 ≤ 0x9F`
  - 4-byte: leading `0xF0` followed by `b1 ≤ 0x8F`

Sample Converter

```
bool ReadCodePoint(char8_t const* pSrc, char32_t& cp) {
    char32_t    u1, u2, u3, u4, nu = 0;

    if ((u1 = *pSrc++) <= 0x7F) {
        cp = u1;  nu = 1;

    } else if ((u1 & 0xE0) == 0xC0) {
        u2 = *pSrc++;  nu = 2;
        cp = ((u1 & 0x1F) << 6) | (u2 & 0x3F);

    } else if ((u1 & 0xF0) == 0xE0) {
        u2 = *pSrc++;
        u3 = *pSrc++;  nu = 3;
        cp = ((u1 & 0x0F) << 12) | ((u2 & 0x3F) << 6) | (u3 & 0x3F);

    } else if ((u1 & 0xF8) == 0xF0) {
        u2 = *pSrc++;
        u3 = *pSrc++;
        u4 = *pSrc++;  nu = 4;
        cp = ((u1 & 0x07) << 18) | ((u2 & 0x3F) << 12) | ((u3 & 0x3F) << 6) | (u4 & 0x3F);
    }
    return Check(cp, nu);
}
```

What is a DFA?

- Deterministic Finite Automaton
  - Finite state machine that accepts/rejects strings of symbols
  - Recognizes *regular languages* – useful for pattern matching
- Defined by
  - Finite number of states
  - A finite set of input symbols
  - A transition function
  - A **start** state
  - One or more **accept** states

How Does a DFA Work?

- Given the current state and an input symbol (the *lookahead*), the transition function specifies the next state

- Beginning at the start state, symbols are consumed and state transitions occur until recognition halts

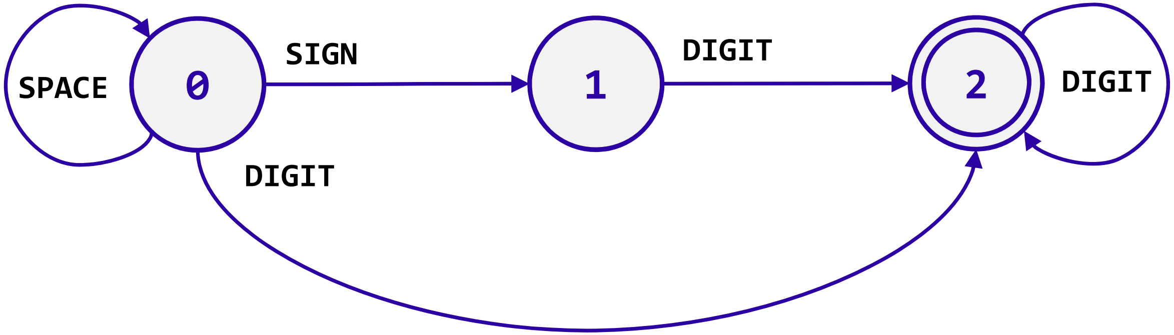
- Recognition halts when:
  - An accept state is reached – the string is *accepted*

    --or--

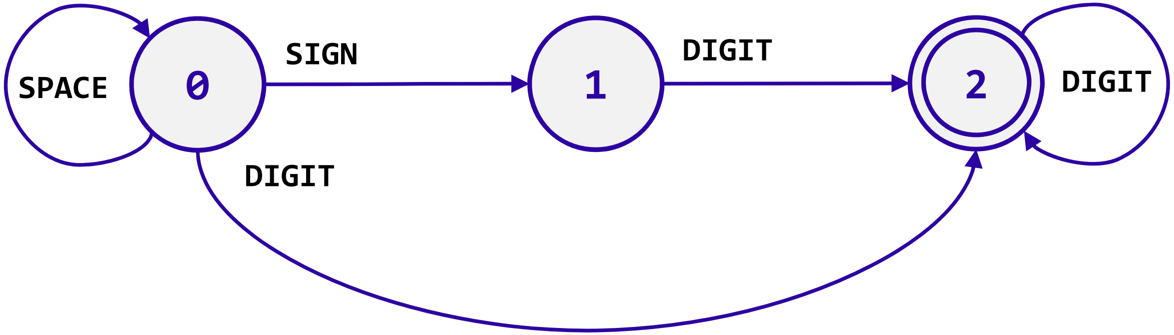
  - There is no transition leaving the state – the string is *rejected*

- DFAs are limited in the languages they can recognize
  - Can recognize simple regular expressions (*ee* `* + ? |`)
  - Cannot solve problems that require more than constant space, such as matching properly paired parentheses

An Example DFA – “[ ]*(+|-)?[0..9]+”



An Example DFA – “[ ]*(+|-)?[0..9]+”



State\Input	DIGIT	SIGN	SPACE	OTHER
0	2	1	0	R
1	2	R	R	R
2	2	A	A	A

Recognizing UTF-8

Boundary Conditions - Reminder

- Maximum code point `U+10FFFF` (17 planes of 2^16 code points)

- UTF-16 surrogates range `U+D800..U+DFFF`
  - Leading/High: `0xD800..0xDBFF`
  - Trailing/Low: `0xDC00..0xDFFF`

- Overlong sequences
  - 2-byte: leading `0xC0` or `0xC1`
  - 3-byte: leading `0xE0` followed by `b1 ≤ 0x9F`
  - 4-byte: leading `0xF0` followed by `b1 ≤ 0x8F`

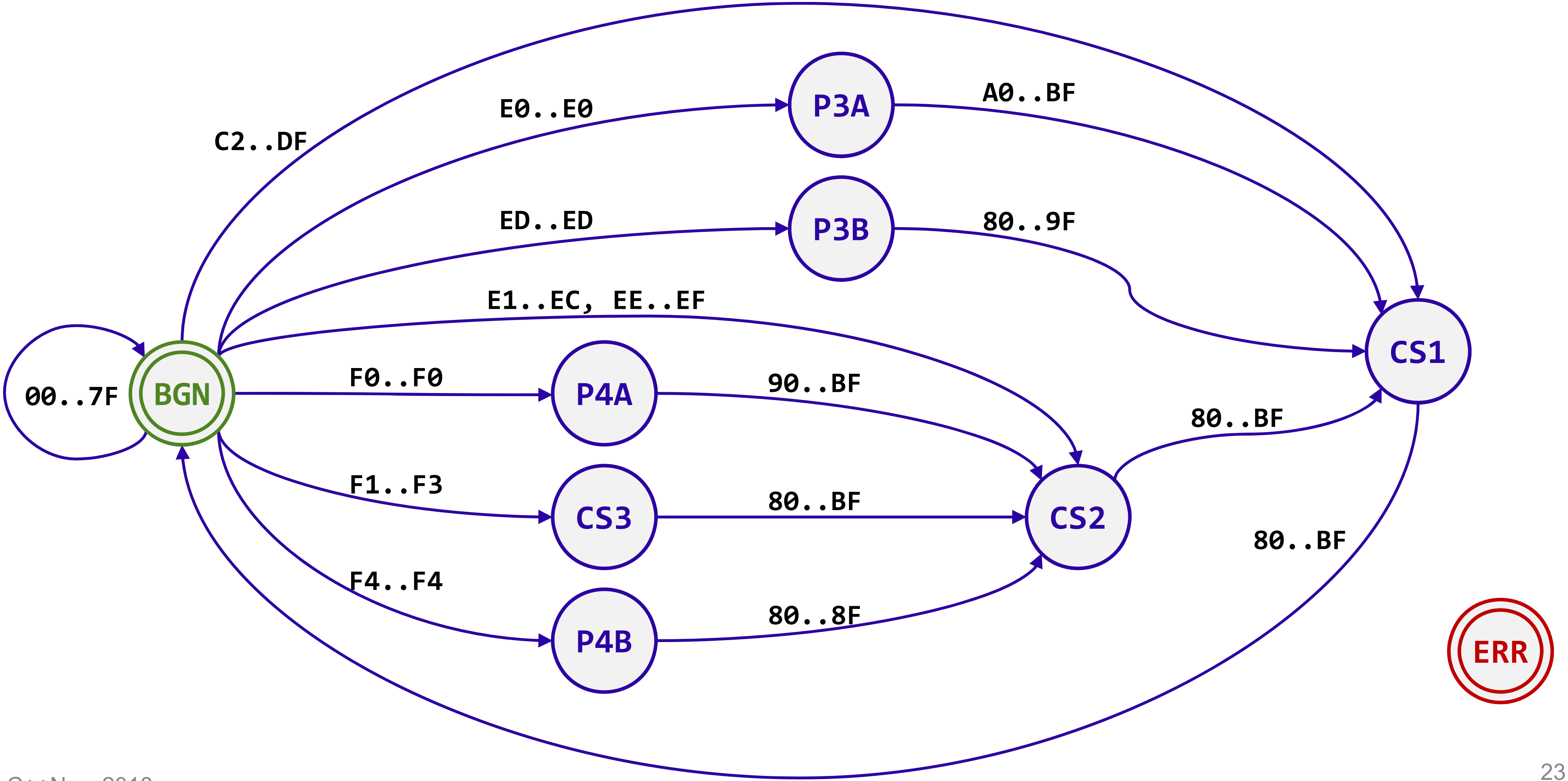
Finding the Transitions

```
CP-Hex          Code Point – Binary       UTF-8 Hex     UTF-8 Binary
======          ===================       =========     ============
0x00      0000 0000 0000 0000 0000 0000   00            0000 0000
0x7F      0000 0000 0000 0000 0111 1111   7F            0111 1111

                                          C0 80         1100 0000 1000 0000 Overlong
                                          C1 BF         1100 0001 1011 1111 Overlong

0x80      0000 0000 0000 0000 1000 0000   C2 80         1100 0010 1000 0000
0x7FF     0000 0000 0000 0111 1111 1111   DF BF         1101 1111 1011 1111

                                          E0 80 80      1110 0000 1000 0000 1000 0000 Overlong
                                          E0 9F BF      1110 0000 1001 1111 1011 1111 Overlong

0x800     0000 0000 0000 1000 1000 1000   E0 A0 80      1110 0000 1010 0000 1000 0000
0xD7FF    0000 0000 1101 0111 1111 1111   ED 9F BF      1110 1101 1001 1111 1011 1111

0xD800    0000 0000 1101 1000 0000 0000   ED A0 80      1110 1101 1010 0000 1000 0000 Surrogates
0xDFFF    0000 0000 1101 1111 1111 1111   ED BF BF      1110 1101 1011 1111 1011 1111
```

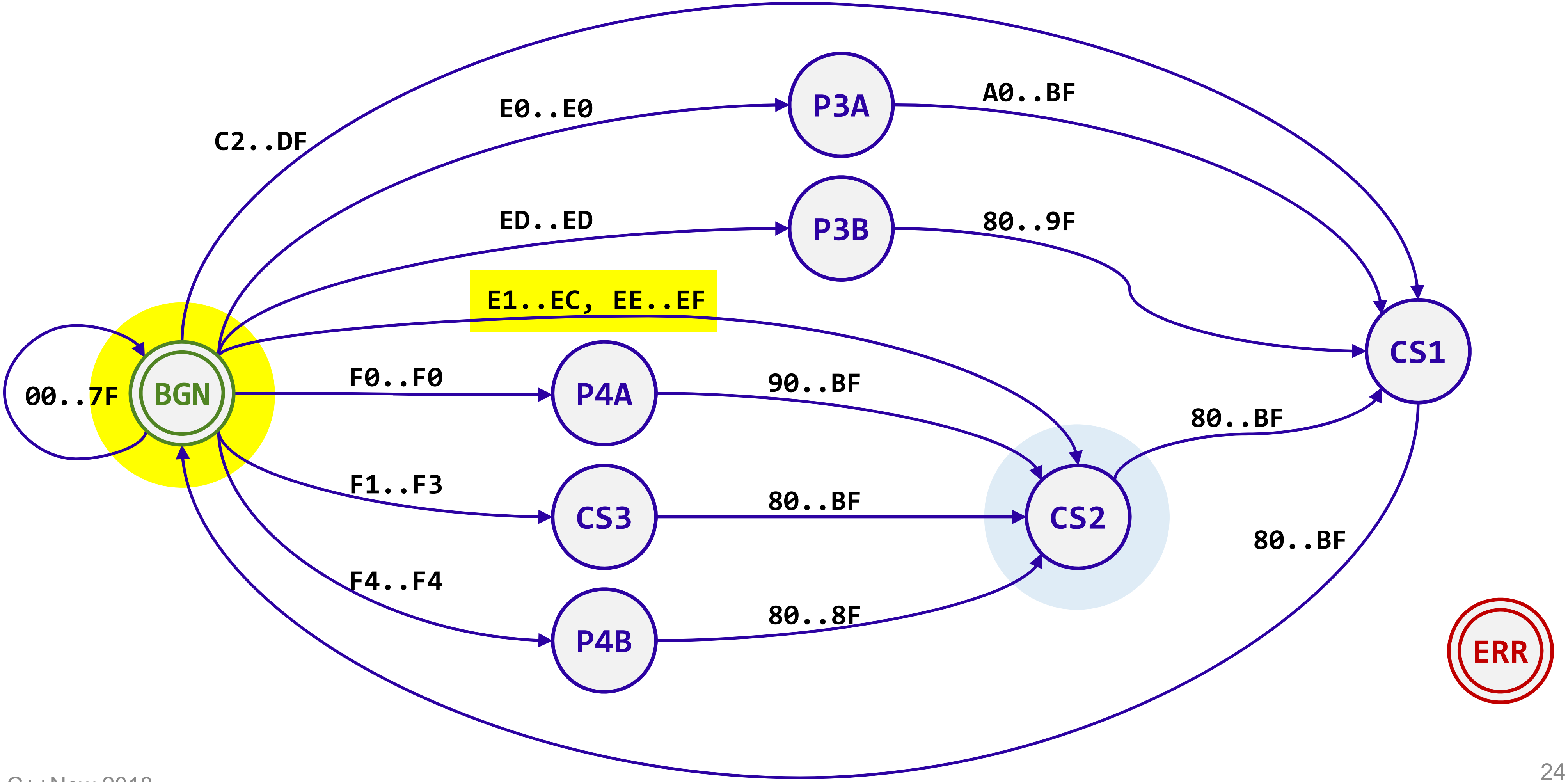
Finding the Transitions

```
CP-Hex          Code Point – Binary       UTF-8 Hex     UTF-8 Binary
======          ===================       =========     ============
0xD800    0000 0000 1101 1000 0000 0000   ED A0 80      1110 1101 1010 0000 1000 0000 Surrogates
0xDFFF    0000 0000 1101 1111 1111 1111   ED BF BF      1110 1101 1011 1111 1011 1111

0xE000    0000 0000 1110 0000 0000 0000   EE 80 80      1110 1110 1000 0000 1000 0000
0xFFFF    0000 0000 1111 1111 1111 1101   EF BF BF      1110 1111 1011 1111 1011 1101

                                          F0 80 80 80   1111 0000 1000 0000 1000 0000 1000 0000
                                          F0 8F BF BF   1111 0000 1000 1111 1011 1111 1011 1111
                                                                                        Overlong

0x10000   0000 0001 0000 0000 0000 0000   F0 90 80 80   1111 0000 1001 0000 1000 0000 1000 0000
0x10FFFF  0001 0000 1111 1111 1111 1111   F4 8F BF BF   1111 0100 1000 1111 1011 1111 1011 1111

0x110000  0001 0001 0000 0000 0000 0000   F4 90 80 80   1111 0100 1001 0000 1000 0000 1000 0000
                                                                                     Out-Of-Range
```

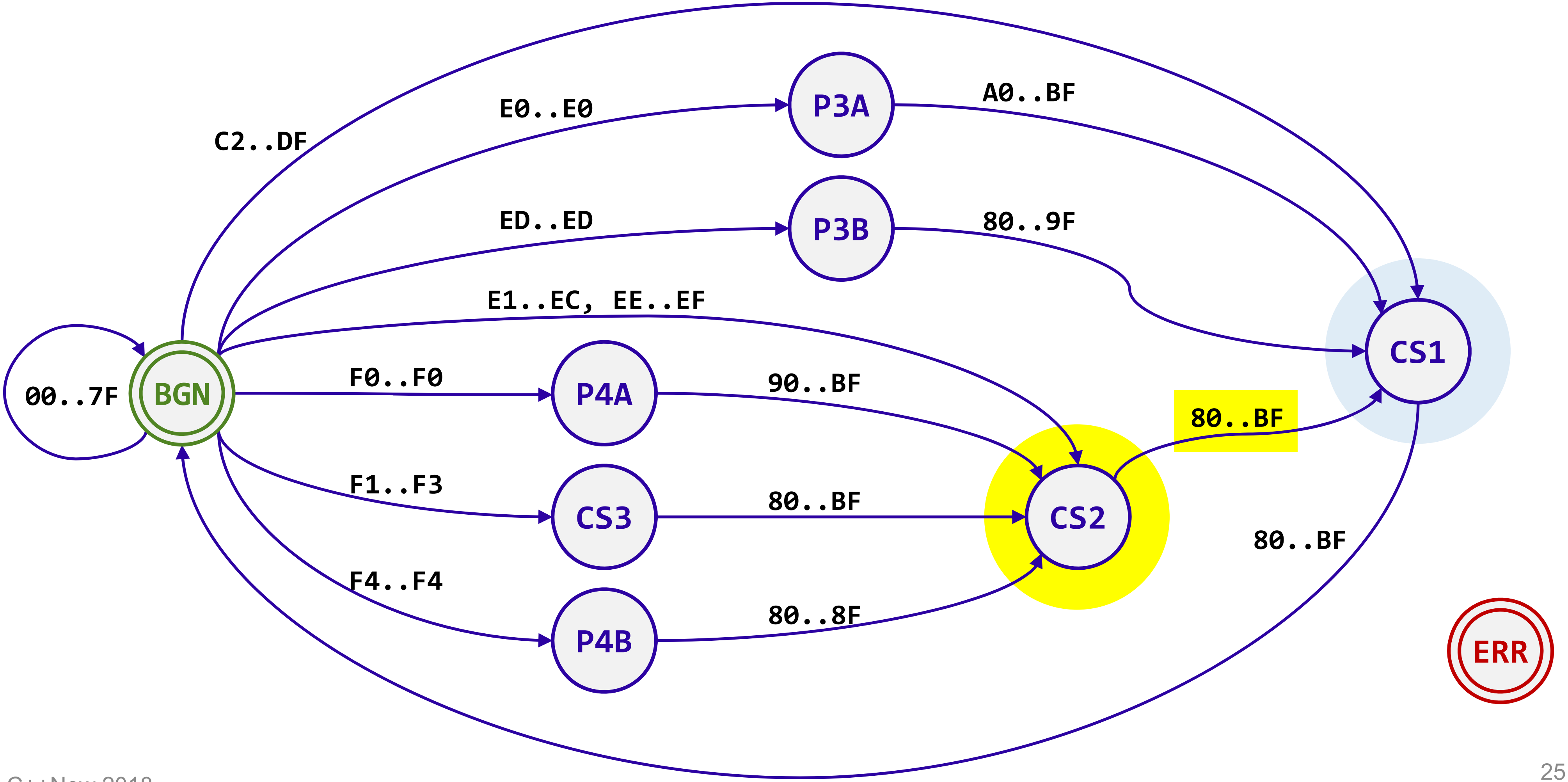
The DFA – Expressed in Octet Value Ranges



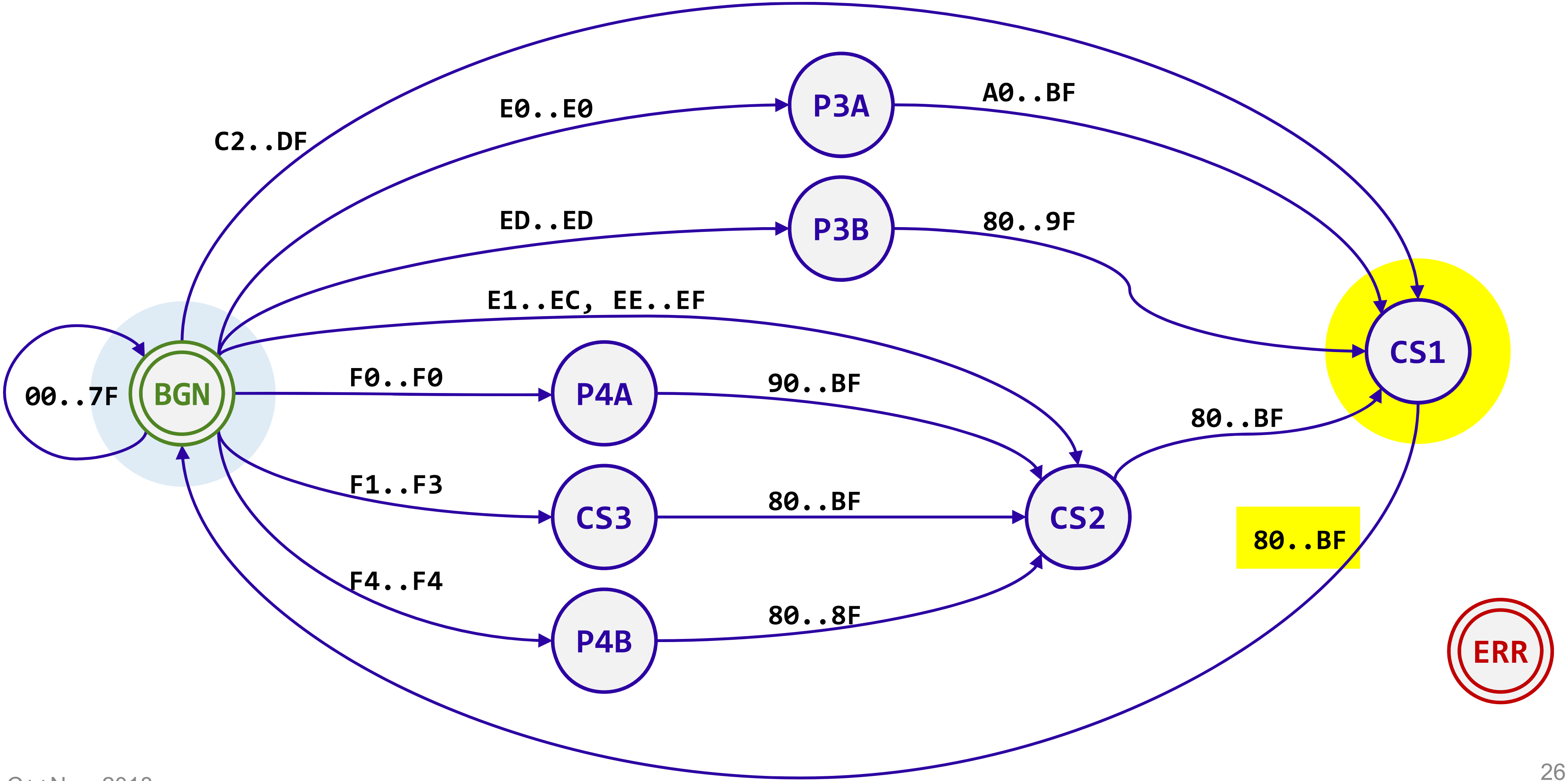
A Recognition Example – { .. E2 88 85 .. }



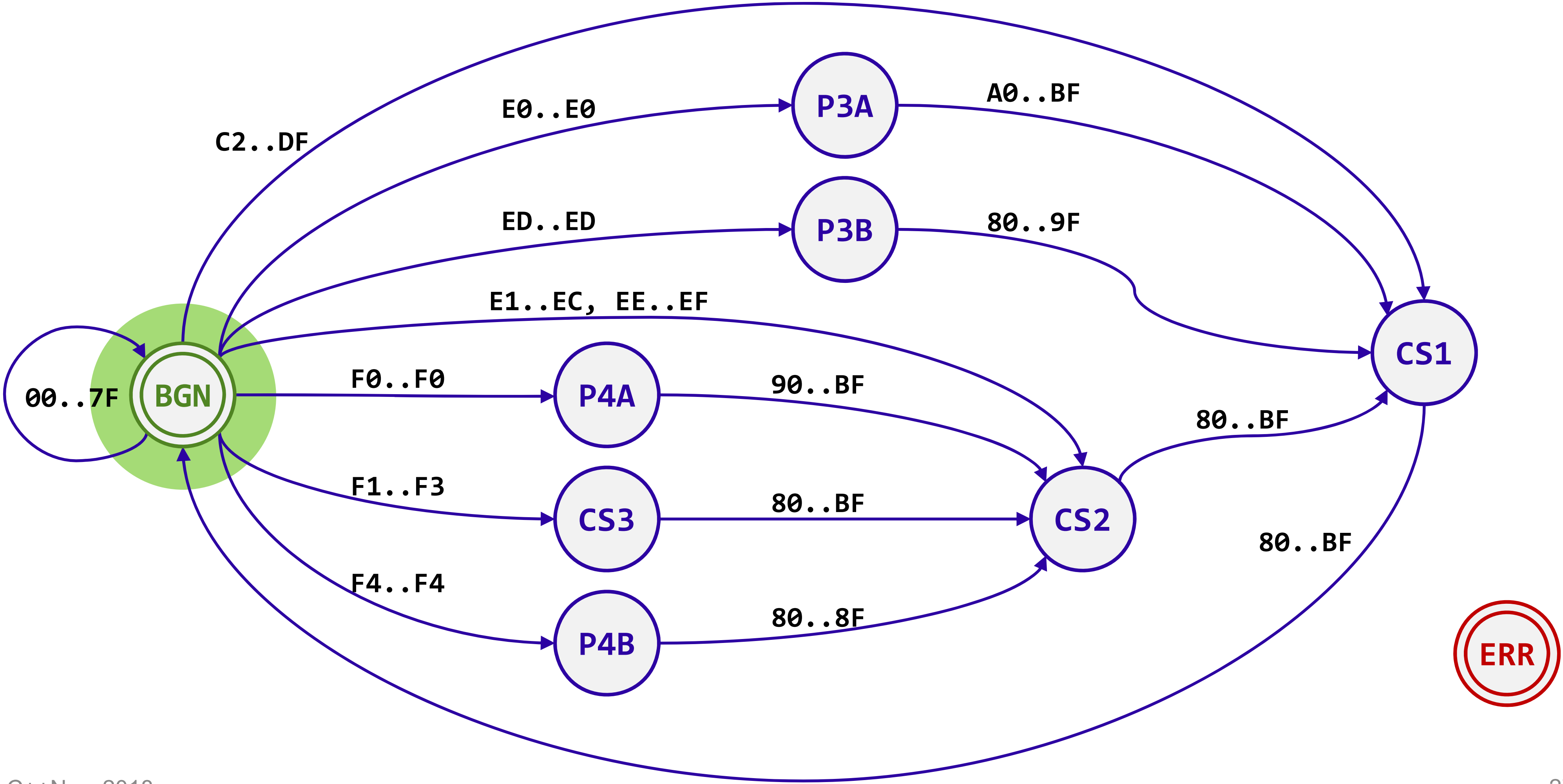
A Recognition Example – { .. E2 88 85 .. }



A Recognition Example – { .. E2 88 85 .. }



A Recognition Example – { .. E2 88 85 .. }



Converter Overview

Interface Assumptions

- Pointer arguments are non-null
- Input and output buffers exist
- Destination buffer is sized to receive output with no overflow
- Destination code points/units are little endian (LE)
- Using x64/x86 hardware and SSE2 instruction set available
- Destination code point buffer is aligned on `char32_t` boundary
- Destination code unit buffer aligned on `char16_t` boundary

Design Ideas/Principles/Goals

- Implement UTF-8 recognition and decoding using table-based DFA
  - Decode while recognizing
- Pre-compute as much as possible that contributes to performance
  - But also keep tables small
- Keep code as simple as possible
  - But also make code fast
  - Hide complexity of recognition in the DFA tables
- Try to be faster than the other guys!

Class Overview – Public Interface

```
class UtfUtils
{
  public:
    using char8_t   = unsigned char;
    using ptrdiff_t = std::ptrdiff_t;

  public:
    static bool         GetCodePoint(char8_t const* pSrc, char8_t const* pSrcEnd, char32_t& cdpt);

    static uint32_t     GetCodeUnits(char32_t cdpt, char8_t*& pDst);
    static uint32_t     GetCodeUnits(char32_t cdpt, char16_t*& pDst);

    //- Conversion to UTF-32/UTF-16.
    //
    static ptrdiff_t  BasicConvert(char8_t const* pSrc, char8_t const* pSrcEnd, char32_t* pDst);
    static ptrdiff_t  FastConvert(char8_t const* pSrc, char8_t const* pSrcEnd, char32_t* pDst);
    static ptrdiff_t  SseConvert(char8_t const* pSrc, char8_t const* pSrcEnd, char32_t* pDst);

    static ptrdiff_t  BasicConvert(char8_t const* pSrc, char8_t const* pSrcEnd, char16_t* pDst);
    static ptrdiff_t  FastConvert(char8_t const* pSrc, char8_t const* pSrcEnd, char16_t* pDst);
    static ptrdiff_t  SseConvert(char8_t const* pSrc, char8_t const* pSrcEnd, char16_t* pDst);
    ...
};
```

Class Overview – Private Interface – Internal Types

```
class UtfUtils
{
  private:
    enum CharClass : uint8_t
    {
        ILL = 0,    //- C0..C1, F5..FF  ILLEGAL octets that never occur in a valid UTF-8 sequence
                    //
        ASC = 1,    //- 00..7F          ASCII leading byte range
                    //
        CR1 = 2,    //- 80..8F          Continuation range 1
        CR2 = 3,    //- 90..9F          Continuation range 2
        CR3 = 4,    //- A0..BF          Continuation range 3
                    //
        L2A = 5,    //- C2..DF          Leading byte range A / 2-byte sequence
        L3A = 6,    //- E0              Leading byte range A / 3-byte sequence
        L3B = 7,    //- E1..EC, EE..EF  Leading byte range B / 3-byte sequence
        L3C = 8,    //- ED              Leading byte range C / 3-byte sequence
        L4A = 9,    //- F0              Leading byte range A / 4-byte sequence
        L4B = 10,   //- F1..F3          Leading byte range B / 4-byte sequence
        L4C = 11,   //- F4              Leading byte range C / 4-byte sequence
    };
    ...
};
```

Class Overview – Private Interface – Internal Types

```
class UtfUtils
{
  private:
    ...
    enum State : uint8_t
    {
        BGN = 0,    //- Start
        ERR = 12,   //- Invalid sequence
                    //
        CS1 = 24,   //- Continuation state 1
        CS2 = 36,   //- Continuation state 2
        CS3 = 48,   //- Continuation state 3
                    //
        P3A = 60,   //- Partial 3-byte sequence state A
        P3B = 72,   //- Partial 3-byte sequence state B
                    //
        P4A = 84,   //- Partial 4-byte sequence state A
        P4B = 96,   //- Partial 4-byte sequence state B
                    //
        END = BGN,  //- Start and End are the same state!
        err = ERR,  //- For readability in the state transition table
    };
    ...
};
```

Class Overview – Private Interface – Internal Types

```
class UtfUtils
{
  private:
    ...
    struct FirstUnitInfo
    {
        char8_t mFirstOctet;   //- Initial value of the code point based on first code unit
        State   mNextState;    //- The next state in the DFA based on the first code unit
    };

    struct alignas(2048) LookupTables           //- Requires 14 cache lines (896 bytes)
    {
        FirstUnitInfo   maFirstUnitTable[256];  //- First code unit info for all code units
        CharClass       maOctetCategory[256];   //- Character class of all code units
        State           maTransitions[108];     //- DFA transition table
    };

    ...
};
```

Class Overview – Private Interface – Key Members

```
class UtfUtils
{
  private:
    ...
    static  LookupTables const  smTables;

    ...
    //- Consume code units in DFA to compute a code point
    //
    static int32_t  Advance(char8_t const*& pSrc, char8_t const* pSrcEnd, char32_t& cdpt);

    //- Convert contiguous runs of ASCII code units in a SIMD way
    //
    static void     ConvertAsciiWithSse(char8_t const*& pSrc, char32_t*& pDst);

    //- Count low-order zero bits in a 32-bit integer
    //
    static int32_t  GetTrailingZeros(int32_t x);
};
```

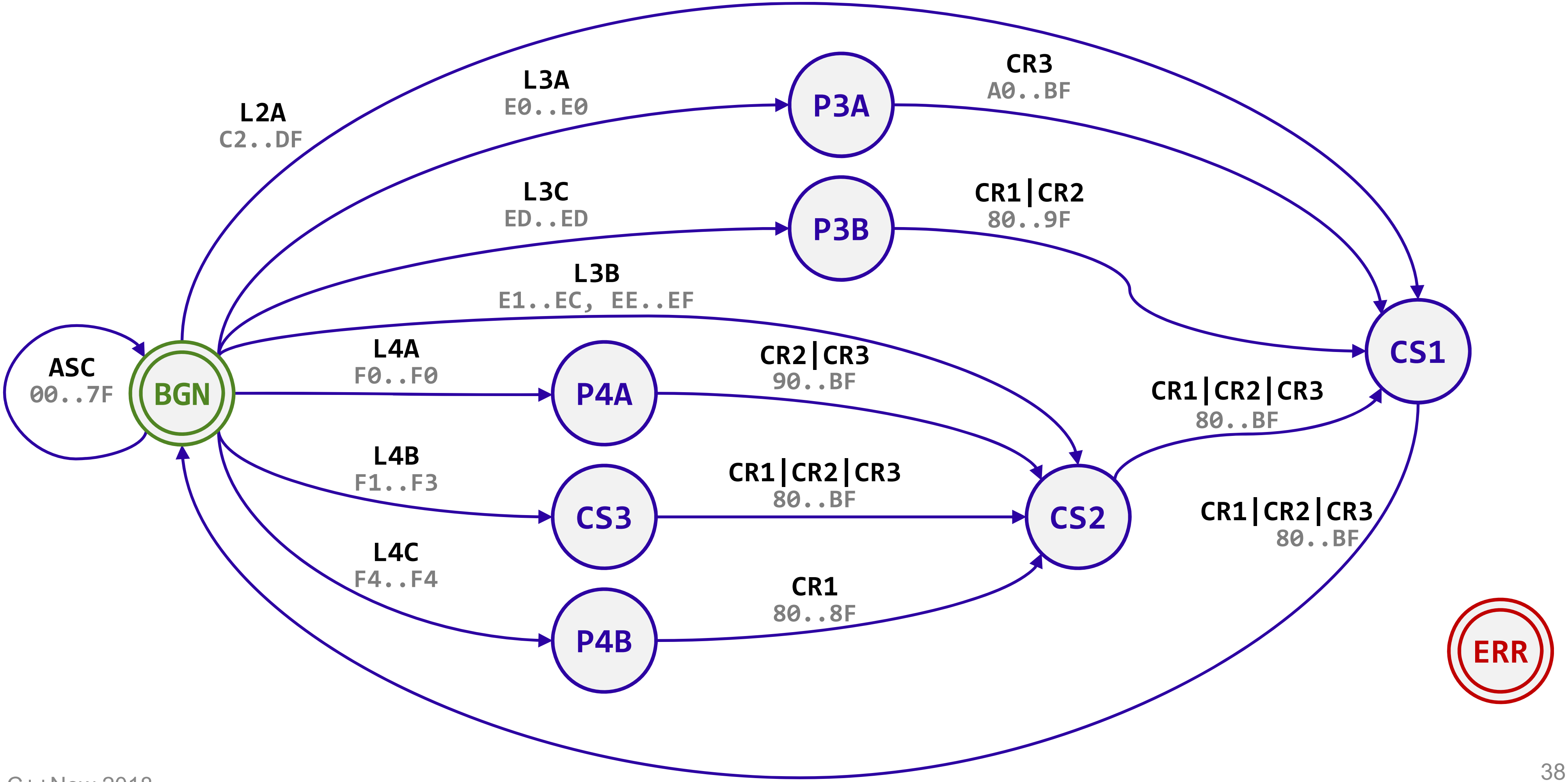
Mapping First Code Unit to Initial Values

```
//- The maFirstUnitTable member array.  This array maps the first code unit of a sequence to
//     1. a pre-masked value to start the computation of the resulting code point; and,
//     2. the next state in the DFA for this code unit.
{
//    CDPT  NEXT        HEXVAL
    ...
    { 0x21, BGN },   //- 0x21  !
    { 0x22, BGN },   //- 0x22  “
    { 0x23, BGN },   //- 0x23  #
    { 0x24, BGN },   //- 0x23  $
    ...
    { 0xC0, ERR },   //- 0xC0
    { 0xC1, ERR },   //- 0xC1
    { 0x02, CS1 },   //- 0xC2
    { 0x03, CS1 },   //- 0xC3
    ...
    { 0x00, P4A },   //- 0xF0
    { 0x01, CS3 },   //- 0xF1
    { 0x02, CS3 },   //- 0xF2
    { 0x03, CS3 },   //- 0xF3
    ...
};
```

Mapping an Octet to its Character Class

```
//- The maOctetCategory member array maps an input octet to a corresponding character class.
//   0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
//============================================================================================
{
    ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, //- 00..0F
    ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, //- 10..1F
    ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, //- 20..2F
    ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, //- 30..3F
                                                                                    //
    ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, //- 40..4F
    ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, //- 50..5F
    ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, //- 60..6F
    ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, ASC, //- 70..7F
                                                                                    //
    CR1, CR1, CR1, CR1, CR1, CR1, CR1, CR1, CR1, CR1, CR1, CR1, CR1, CR1, CR1, CR1, //- 80..8F
    CR2, CR2, CR2, CR2, CR2, CR2, CR2, CR2, CR2, CR2, CR2, CR2, CR2, CR2, CR2, CR2, //- 90..9F
    CR3, CR3, CR3, CR3, CR3, CR3, CR3, CR3, CR3, CR3, CR3, CR3, CR3, CR3, CR3, CR3, //- A0..AF
    CR3, CR3, CR3, CR3, CR3, CR3, CR3, CR3, CR3, CR3, CR3, CR3, CR3, CR3, CR3, CR3, //- B0..BF
                                                                                    //
    ILL, ILL, L2A, L2A, L2A, L2A, L2A, L2A, L2A, L2A, L2A, L2A, L2A, L2A, L2A, L2A, //- C0..CF
    L2A, L2A, L2A, L2A, L2A, L2A, L2A, L2A, L2A, L2A, L2A, L2A, L2A, L2A, L2A, L2A, //- D0..DF
    L3A, L3B, L3B, L3B, L3B, L3B, L3B, L3B, L3B, L3B, L3B, L3B, L3B, L3C, L3B, L3B, //- E0..EF
    L4A, L4B, L4B, L4B, L4C, ILL, ILL, ILL, ILL, ILL, ILL, ILL, ILL, ILL, ILL, ILL, //- F0..FF
},
```

The DFA – Expressed in Character Classes / Octet Value Ranges



DFA Transition Table

```
//- The maTransitions member array.  Given the current DFA state and an input octet,
//  get the next DFA state.
//
//  ILL  ASC  CR1  CR2  CR3  L2A  L3A  L3B  L3C  L4A  L4B  L4C  CLASS/STATE
//=========================================================================
{
    err, END, err, err, err, CS1, P3A, CS2, P3B, P4A, CS3, P4B,   //- BGN|END
    err, err, err, err, err, err, err, err, err, err, err, err,   //- ERR
                                                                  //
    err, err, END, END, END, err, err, err, err, err, err, err,   //- CS1
    err, err, CS1, CS1, CS1, err, err, err, err, err, err, err,   //- CS2
    err, err, CS2, CS2, CS2, err, err, err, err, err, err, err,   //- CS3
                                                                  //
    err, err, err, err, CS1, err, err, err, err, err, err, err,   //- P3A
    err, err, CS1, CS1, err, err, err, err, err, err, err, err,   //- P3B
                                                                  //
    err, err, err, CS2, CS2, err, err, err, err, err, err, err,   //- P4A
    err, err, CS2, err, err, err, err, err, err, err, err, err,   //- P4B
},

...
```

Basic Conversion Algorithm

The Basic Conversion Algorithm (UTF-8 to UTF-32)

```
KEWB_ALIGN_FN std::ptrdiff_t
UtfUtils::BasicConvert(char8_t const* pSrc, char8_t const* pSrcEnd, char32_t* pDst) noexcept
{
    char32_t*   pDstOrig = pDst;
    char32_t    cdpt;

    while (pSrc < pSrcEnd)
    {
        if (Advance(pSrc, pSrcEnd, cdpt) != ERR)
        {
            *pDst++ = cdpt;
        }
        else
        {
            return -1;
        }
    }
    return pDst - pDstOrig;
}
```

Converting a Single Code Point - Overview

```
KEWB_FORCE_INLINE int32_t
UtfUtils::Advance(char8_t const*& pSrc, char8_t const* pSrcEnd, char32_t& cdpt) noexcept
{
    FirstUnitInfo   info;   //- The descriptor for the first code unit
    char32_t        unit;   //- The current UTF-8 code unit
    int32_t         type;   //- The code unit's character class
    int32_t         curr;   //- The current DFA state

    info = smTables.maFirstUnitTable[*pSrc++];          //- Look up the first descriptor
    cdpt = info.mFirstOctet;                            //- Get the initial code point value
    curr = info.mNextState;                             //- Advance to the next state

    while (curr > ERR)                                  //- Loop over subsequent units
    {
        if (pSrc < pSrcEnd)
        {
            unit = *pSrc++;                             //- Cache the current code unit
            cdpt = (cdpt << 6) | (unit & 0x3F);         //- Adjust code point with continuation bits
            type = smTables.maOctetCategory[unit];      //- Look up the code unit's character class
            curr = smTables.maTransitions[curr + type]; //- Advance to the next state
        }
        else
        {
            return ERR;
        }
    }
    return curr;
}
```

Converting a Single Code Point

```
KEWB_FORCE_INLINE int32_t
UtfUtils::Advance(char8_t const*& pSrc, char8_t const* pSrcEnd, char32_t& cdpt) noexcept
{
    FirstUnitInfo   info;   //- The descriptor for the first code unit
    char32_t        unit;   //- The current UTF-8 code unit
    int32_t         type;   //- The code unit's character class
    int32_t         curr;   //- The current DFA state

    info = smTables.maFirstUnitTable[*pSrc++];      //- Look up the first code unit descriptor
    cdpt = info.mFirstOctet;                        //- Get the initial code point value
    curr = info.mNextState;                         //- Advance to the next state

    while (curr > ERR)                              //- Loop over subsequent code units
    {
        ...
    }
    return curr;
}
```

Converting a Single Code Point

```
KEWB_FORCE_INLINE int32_t
UtfUtils::Advance(char8_t const*& pSrc, char8_t const* pSrcEnd, char32_t& cdpt) noexcept
{
    FirstUnitInfo   info;   //- The descriptor for the first code unit
    char32_t        unit;   //- The current UTF-8 code unit
    int32_t         type;   //- The code unit's character class
    int32_t         curr;   //- The current DFA state

    info = smTables.maFirstUnitTable[*pSrc++];      //- Look up the first code unit descriptor
    cdpt = info.mFirstOctet;                        //- Get the initial code point value
    curr = info.mNextState;                         //- Advance to the next state

    while (curr > ERR)                              //- Loop over subsequent code units
    {
        ...
    }
    return curr;
}
```

Converting a Single Code Point

```
KEWB_FORCE_INLINE int32_t
UtfUtils::Advance(char8_t const*& pSrc, char8_t const* pSrcEnd, char32_t& cdpt) noexcept
{
    FirstUnitInfo   info;   //- The descriptor for the first code unit
    char32_t        unit;   //- The current UTF-8 code unit
    int32_t         type;   //- The code unit's character class
    int32_t         curr;   //- The current DFA state

    info = smTables.maFirstUnitTable[*pSrc++];      //- Look up the first code unit descriptor
    cdpt = info.mFirstOctet;                        //- Get the initial code point value
    curr = info.mNextState;                         //- Advance to the next state

    while (curr > ERR)                              //- Loop over subsequent code units
    {
        ...
    }
    return curr;
}
```

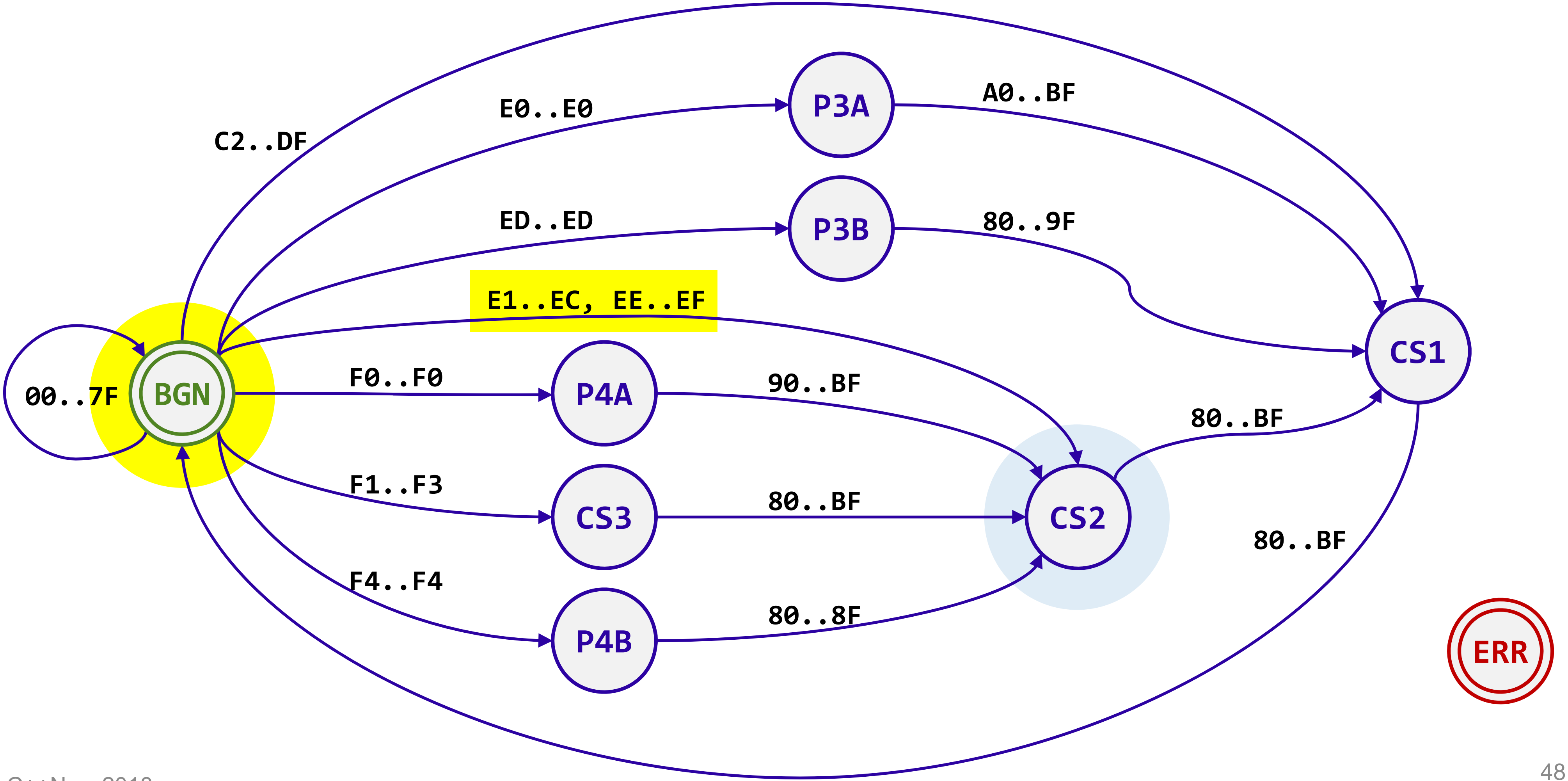
Converting a Single Code Point

```
KEWB_FORCE_INLINE int32_t
UtfUtils::Advance(char8_t const*& pSrc, char8_t const* pSrcEnd, char32_t& cdpt) noexcept
{
    FirstUnitInfo   info;   //- The descriptor for the first code unit
    char32_t        unit;   //- The current UTF-8 code unit
    int32_t         type;   //- The code unit's character class
    int32_t         curr;   //- The current DFA state

    info = smTables.maFirstUnitTable[*pSrc++];      //- Look up the first code unit descriptor
    cdpt = info.mFirstOctet;                        //- Get the initial code point value
    curr = info.mNextState;                         //- Advance to the next state

    while (curr > ERR)                              //- Loop over subsequent code units
    {
        ...
    }
    return curr;
}
```

A Decoding Example – { .. E2 88 85 .. }

```
0000 0000 0000 0000        cdpt
```

A Decoding Example – { .. E2 88 85 .. }



Converting a Single Code Point

```
KEWB_FORCE_INLINE int32_t
UtfUtils::Advance(char8_t const*& pSrc, char8_t const* pSrcEnd, char32_t& cdpt) noexcept
{
    FirstUnitInfo   info;   //- The descriptor for the first code unit
    char32_t        unit;   //- The current UTF-8 code unit
    int32_t         type;   //- The code unit's character class
    int32_t         curr;   //- The current DFA state

    info = smTables.maFirstUnitTable[*pSrc++];      //- Look up the first code unit descriptor
    cdpt = info.mFirstOctet;                        //- Get the initial code point value
    curr = info.mNextState;                         //- Advance to the next state

    while (curr > ERR)                              //- Loop over subsequent code units
    {
        ...
    }
    return curr;
}
```

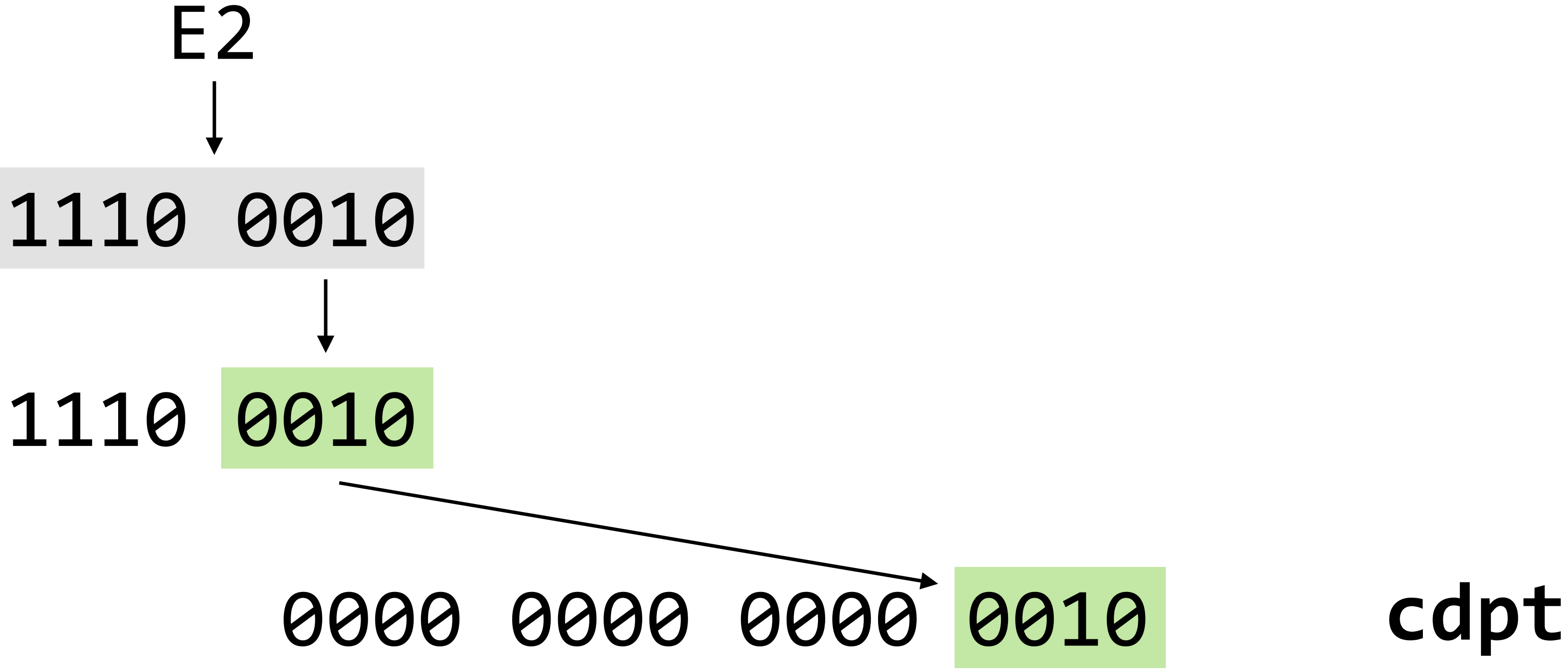
Converting a Single Code Point

```
KEWB_FORCE_INLINE int32_t
UtfUtils::Advance(char8_t const*& pSrc, char8_t const* pSrcEnd, char32_t& cdpt) noexcept
{
    FirstUnitInfo   info;   //- The descriptor for the first code unit
    char32_t        unit;   //- The current UTF-8 code unit
    int32_t         type;   //- The code unit's character class
    int32_t         curr;   //- The current DFA state

    info = smTables.maFirstUnitTable[*pSrc++];      //- Look up the first code unit descriptor
    cdpt = info.mFirstOctet;                        //- Get the initial code point value
    curr = info.mNextState;                         //- Advance to the next state

    while (curr > ERR)                              //- Loop over subsequent code units
    {
        ...
    }
    return curr;
}
```

Converting a Single Code Point

```
KEWB_FORCE_INLINE int32_t
UtfUtils::Advance(char8_t const*& pSrc, char8_t const* pSrcEnd, char32_t& cdpt) noexcept
{
    FirstUnitInfo   info;   //- The descriptor for the first code unit
    char32_t        unit;   //- The current UTF-8 code unit
    int32_t         type;   //- The code unit's character class
    int32_t         curr;   //- The current DFA state

    info = smTables.maFirstUnitTable[*pSrc++];      //- Look up the first code unit descriptor
    cdpt = info.mFirstOctet;                        //- Get the initial code point value
    curr = info.mNextState;                         //- Advance to the next state

    while (curr > ERR)                              //- Loop over subsequent code units
    {
        ...
    }
    return curr;
}
```

A Decoding Example – { .. E2 88 85 .. }



E2

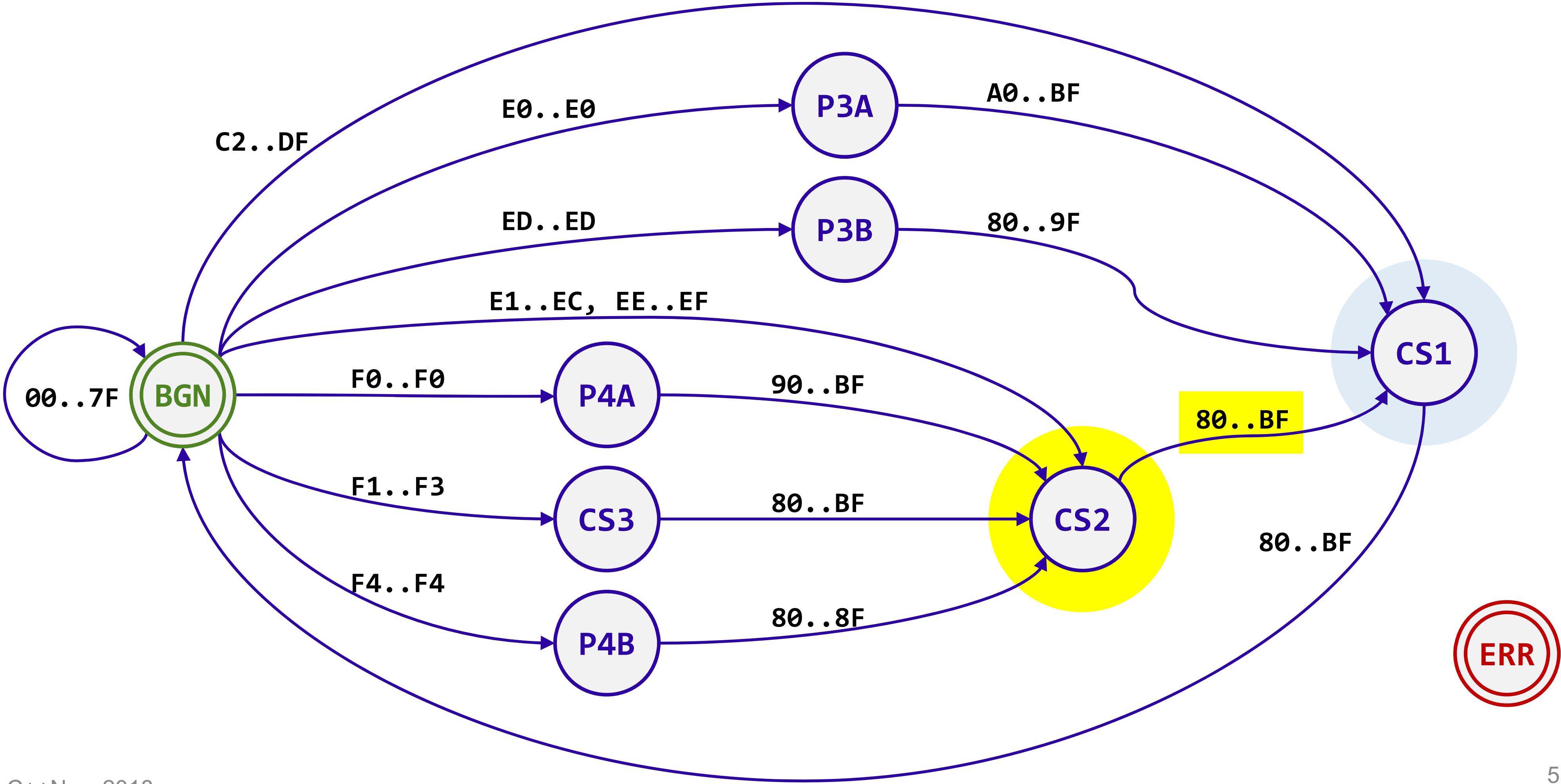
1110 0010

1110 0010

0000 0000 0000 0010

cdpt

A Decoding Example – { .. E2 88 85 .. }



Converting a Single Code Point

```
KEWB_FORCE_INLINE int32_t
UtfUtils::Advance(char8_t const*& pSrc, char8_t const* pSrcEnd, char32_t& cdpt) noexcept
{
    ...

    while (curr > ERR)                                  //- Loop over subsequent code units
    {
        if (pSrc < pSrcEnd)
        {
            unit = *pSrc++;                             //- Cache the current code unit
            cdpt = (cdpt << 6) | (unit & 0x3F);         //- Adjust code point with new bits
            type = smTables.maOctetCategory[unit];      //- Look up the code unit's char class
            curr = smTables.maTransitions[curr + type]; //- Advance to the next state
        }
        else
        {
            return ERR;
        }
    }
    return curr;
}
```

Converting a Single Code Point

```
KEWB_FORCE_INLINE int32_t
UtfUtils::Advance(char8_t const*& pSrc, char8_t const* pSrcEnd, char32_t& cdpt) noexcept
{
    ...

    while (curr > ERR)                                  //- Loop over subsequent code units
    {
        if (pSrc < pSrcEnd)
        {
            unit = *pSrc++;                             //- Cache the current code unit
            cdpt = (cdpt << 6) | (unit & 0x3F);         //- Adjust code point with new bits
            type = smTables.maOctetCategory[unit];      //- Look up the code unit's char class
            curr = smTables.maTransitions[curr + type]; //- Advance to the next state
        }
        else
        {
            return ERR;
        }
    }
    return curr;
}
```

Converting a Single Code Point

```
KEWB_FORCE_INLINE int32_t
UtfUtils::Advance(char8_t const*& pSrc, char8_t const* pSrcEnd, char32_t& cdpt) noexcept
{
    ...

    while (curr > ERR)                                  //- Loop over subsequent code units
    {
        if (pSrc < pSrcEnd)
        {
            unit = *pSrc++;                             //- Cache the current code unit
            cdpt = (cdpt << 6) | (unit & 0x3F);         //- Adjust code point with new bits
            type = smTables.maOctetCategory[unit];      //- Look up the code unit's char class
            curr = smTables.maTransitions[curr + type]; //- Advance to the next state
        }
        else
        {
            return ERR;
        }
    }
    return curr;
}
```

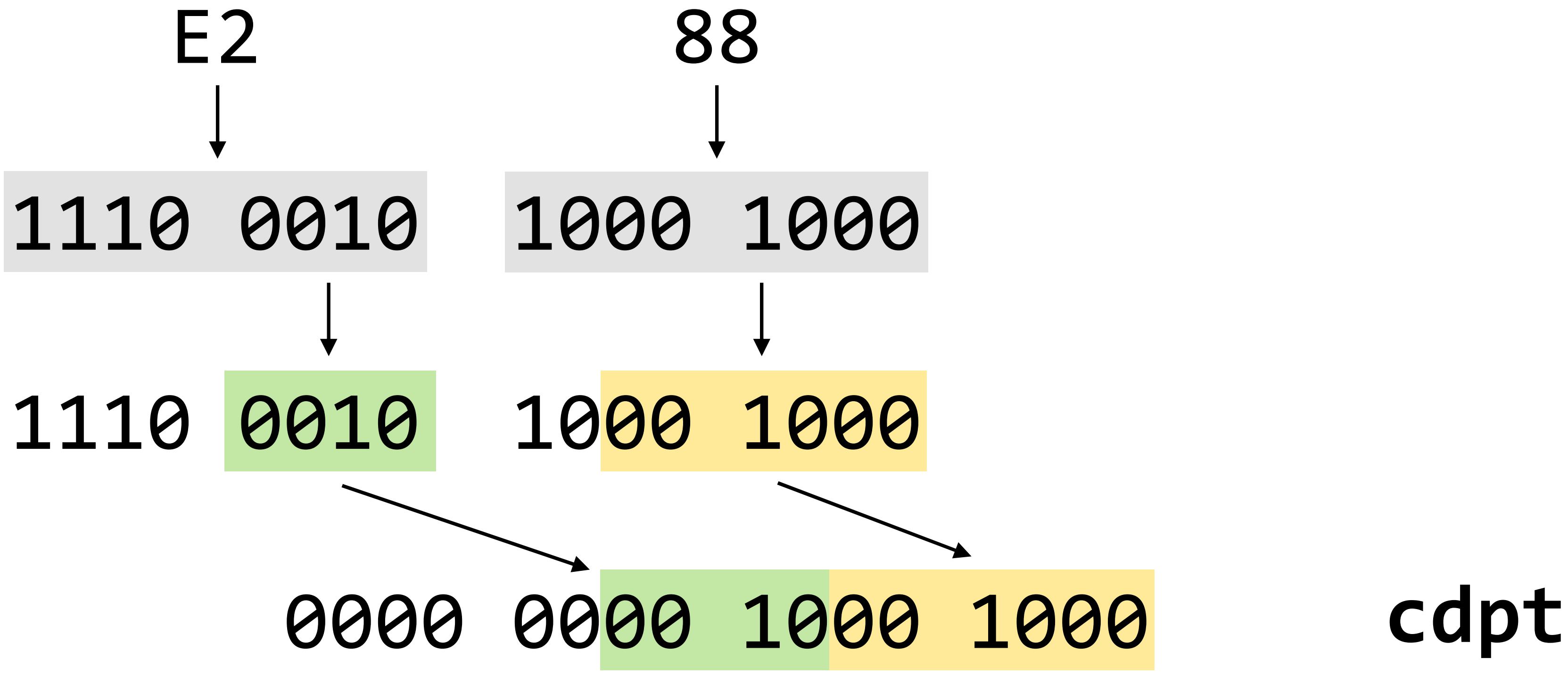
Converting a Single Code Point

```
KEWB_FORCE_INLINE int32_t
UtfUtils::Advance(char8_t const*& pSrc, char8_t const* pSrcEnd, char32_t& cdpt) noexcept
{
    ...

    while (curr > ERR)                                  //- Loop over subsequent code units
    {
        if (pSrc < pSrcEnd)
        {
            unit = *pSrc++;                             //- Cache the current code unit
            cdpt = (cdpt << 6) | (unit & 0x3F);         //- Adjust code point with new bits
            type = smTables.maOctetCategory[unit];      //- Look up the code unit's char class
            curr = smTables.maTransitions[curr + type]; //- Advance to the next state
        }
        else
        {
            return ERR;
        }
    }
    return curr;
}
```

Converting a Single Code Point

```
KEWB_FORCE_INLINE int32_t
UtfUtils::Advance(char8_t const*& pSrc, char8_t const* pSrcEnd, char32_t& cdpt) noexcept
{
    ...

    while (curr > ERR)                                  //- Loop over subsequent code units
    {
        if (pSrc < pSrcEnd)
        {
            unit = *pSrc++;                             //- Cache the current code unit
            cdpt = (cdpt << 6) | (unit & 0x3F);         //- Adjust code point with new bits
            type = smTables.maOctetCategory[unit];      //- Look up the code unit's char class
            curr = smTables.maTransitions[curr + type]; //- Advance to the next state
        }
        else
        {
            return ERR;
        }
    }
    return curr;
}
```

A Decoding Example – { .. E2 88 85 .. }



E2

88

1110 0010

1000 1000

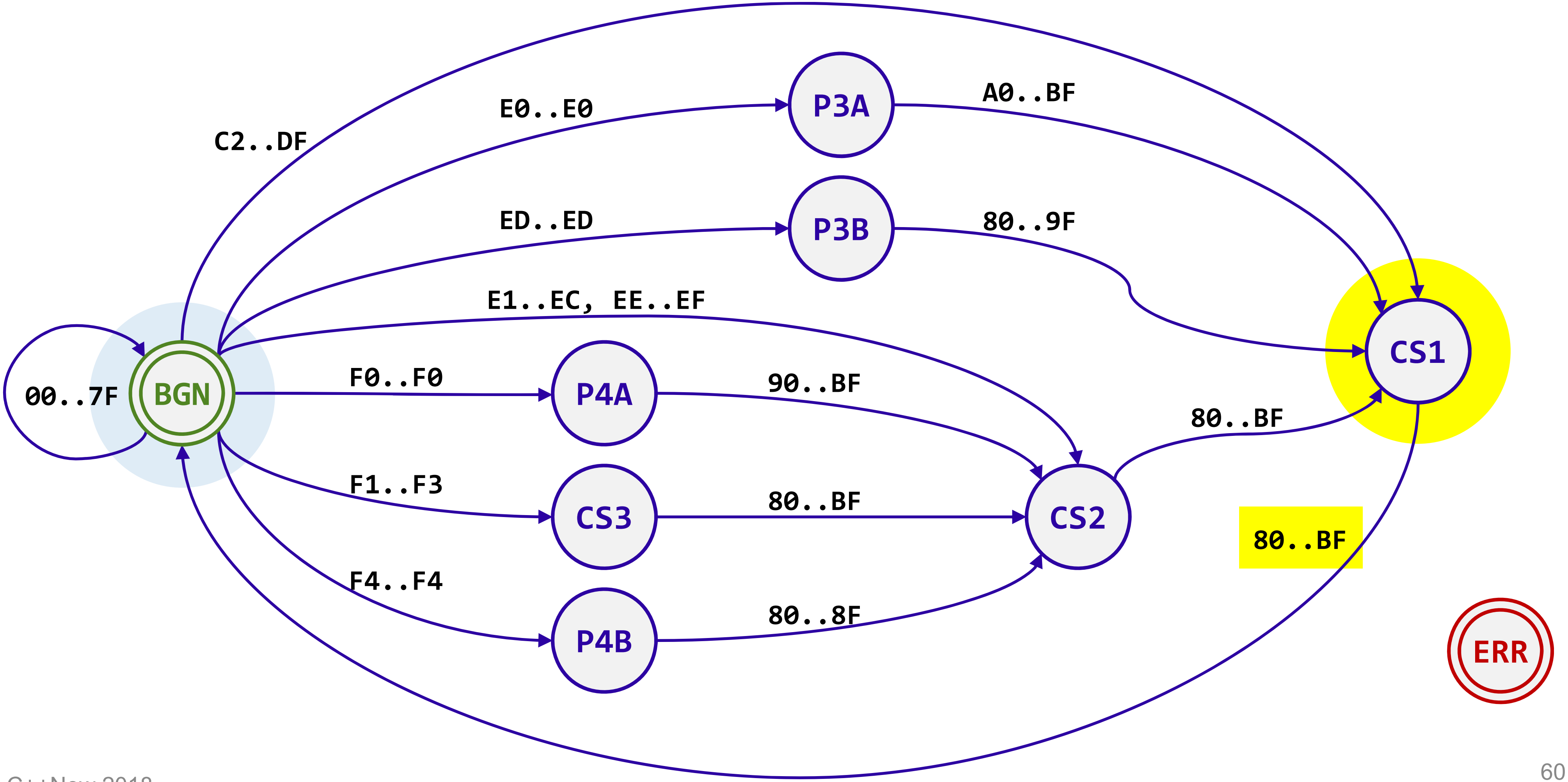
1110 0010

1000 1000

0000 0000 1000 1000

cdpt

A Decoding Example – { .. E2 88 85 .. }



Converting a Single Code Point

```
KEWB_FORCE_INLINE int32_t
UtfUtils::Advance(char8_t const*& pSrc, char8_t const* pSrcEnd, char32_t& cdpt) noexcept
{
    ...

    while (curr > ERR)                                  //- Loop over subsequent code units
    {
        if (pSrc < pSrcEnd)
        {
            unit = *pSrc++;                             //- Cache the current code unit
            cdpt = (cdpt << 6) | (unit & 0x3F);         //- Adjust code point with new bits
            type = smTables.maOctetCategory[unit];      //- Look up the code unit's char class
            curr = smTables.maTransitions[curr + type]; //- Advance to the next state
        }
        else
        {
            return ERR;
        }
    }
    return curr;
}
```

Converting a Single Code Point

```
KEWB_FORCE_INLINE int32_t
UtfUtils::Advance(char8_t const*& pSrc, char8_t const* pSrcEnd, char32_t& cdpt) noexcept
{
    ...

    while (curr > ERR)                                  //- Loop over subsequent code units
    {
        if (pSrc < pSrcEnd)
        {
            unit = *pSrc++;                             //- Cache the current code unit
            cdpt = (cdpt << 6) | (unit & 0x3F);         //- Adjust code point with new bits
            type = smTables.maOctetCategory[unit];      //- Look up the code unit's char class
            curr = smTables.maTransitions[curr + type]; //- Advance to the next state
        }
        else
        {
            return ERR;
        }
    }
    return curr;
}
```

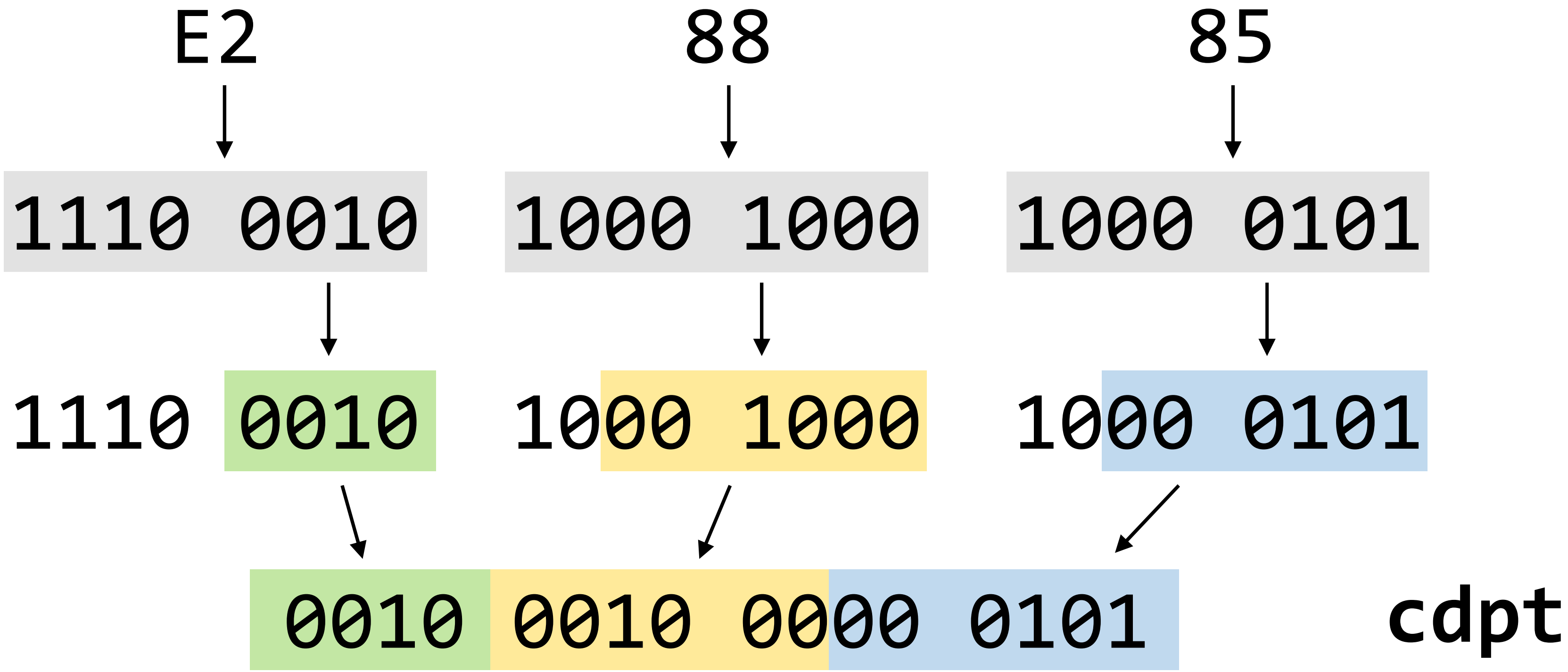
Converting a Single Code Point

```
KEWB_FORCE_INLINE int32_t
UtfUtils::Advance(char8_t const*& pSrc, char8_t const* pSrcEnd, char32_t& cdpt) noexcept
{
    ...

    while (curr > ERR)                                  //- Loop over subsequent code units
    {
        if (pSrc < pSrcEnd)
        {
            unit = *pSrc++;                             //- Cache the current code unit
            cdpt = (cdpt << 6) | (unit & 0x3F);         //- Adjust code point with new bits
            type = smTables.maOctetCategory[unit];      //- Look up the code unit's char class
            curr = smTables.maTransitions[curr + type]; //- Advance to the next state
        }
        else
        {
            return ERR;
        }
    }
    return curr;
}
```

Converting a Single Code Point

```
KEWB_FORCE_INLINE int32_t
UtfUtils::Advance(char8_t const*& pSrc, char8_t const* pSrcEnd, char32_t& cdpt) noexcept
{
    ...

    while (curr > ERR)                                  //- Loop over subsequent code units
    {
        if (pSrc < pSrcEnd)
        {
            unit = *pSrc++;                             //- Cache the current code unit
            cdpt = (cdpt << 6) | (unit & 0x3F);         //- Adjust code point with new bits
            type = smTables.maOctetCategory[unit];      //- Look up the code unit's char class
            curr = smTables.maTransitions[curr + type]; //- Advance to the next state
        }
        else
        {
            return ERR;
        }
    }
    return curr;
}
```

A Decoding Example – { .. E2 88 85 .. }



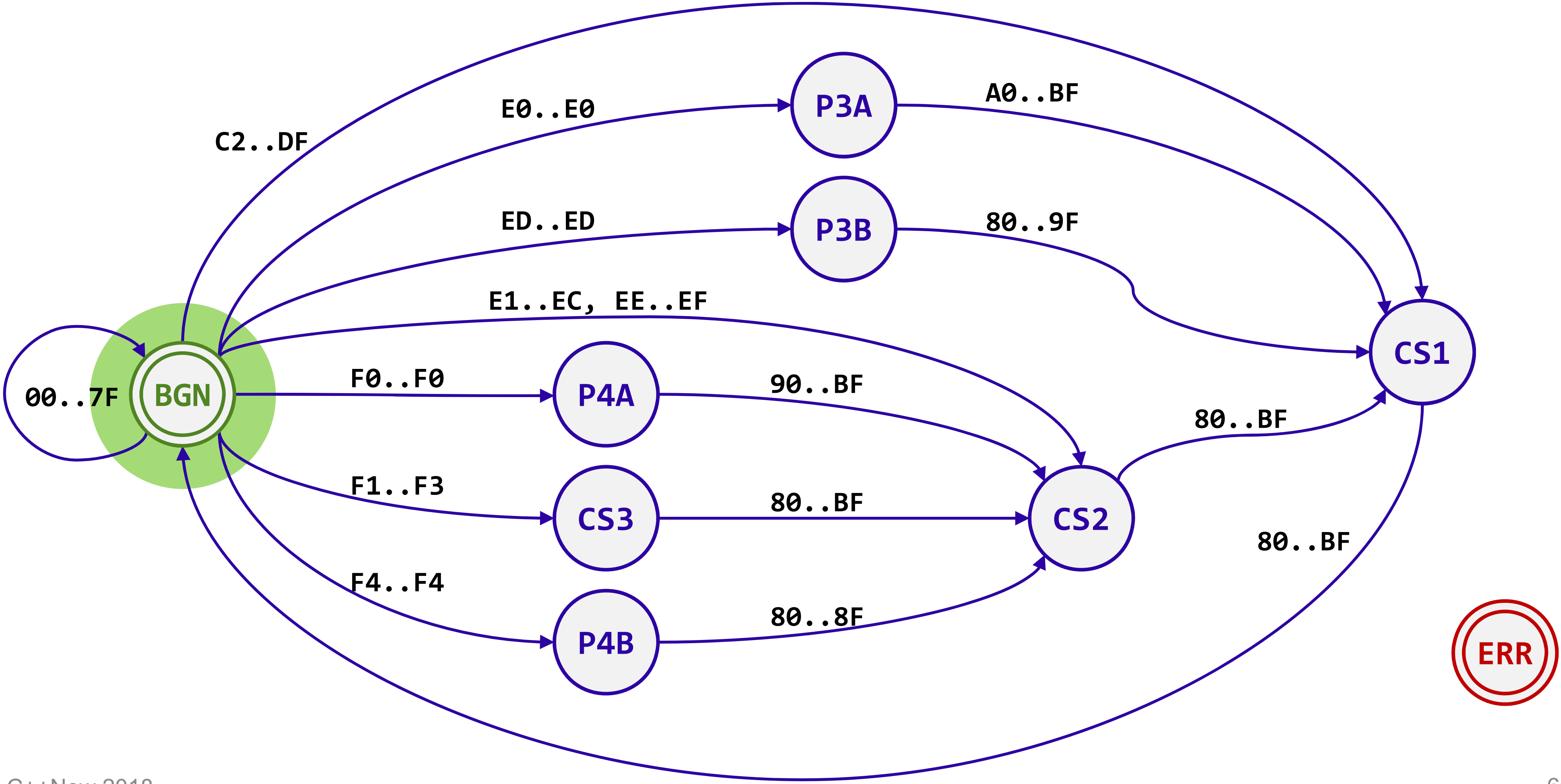
E2 88 85

1110 0010 1000 1000 1000 0101

1110 0010 1000 1000 1000 0101

0010 0010 0000 0101 **cdpt**

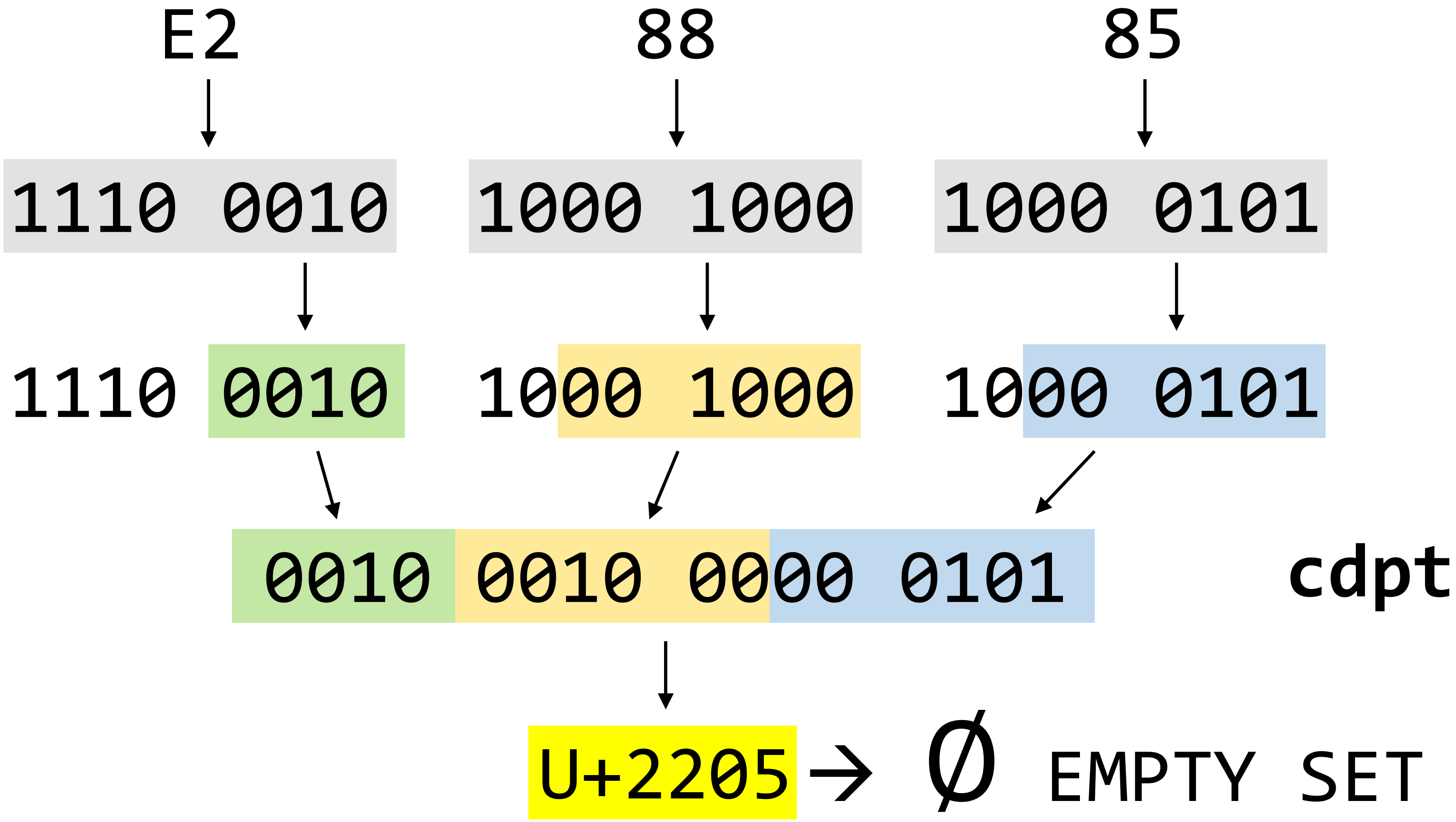
A Decoding Example – { .. E2 88 85 .. }



Converting a Single Code Point

```
KEWB_FORCE_INLINE int32_t
UtfUtils::Advance(char8_t const*& pSrc, char8_t const* pSrcEnd, char32_t& cdpt) noexcept
{
    ...

    while (curr > ERR)                                  //- Loop over subsequent code units
    {
        if (pSrc < pSrcEnd)
        {
            unit = *pSrc++;                             //- Cache the current code unit
            cdpt = (cdpt << 6) | (unit & 0x3F);         //- Adjust code point with new bits
            type = smTables.maOctetCategory[unit];      //- Look up the code unit's char class
            curr = smTables.maTransitions[curr + type]; //- Advance to the next state
        }
        else
        {
            return ERR;
        }
    }
    return curr;
}
```

A Decoding Example – { .. E2 88 85 .. }



E2 88 85

1110 0010 1000 1000 1000 0101

1110 0010 1000 1000 1000 0101

0010 0010 0000 0101

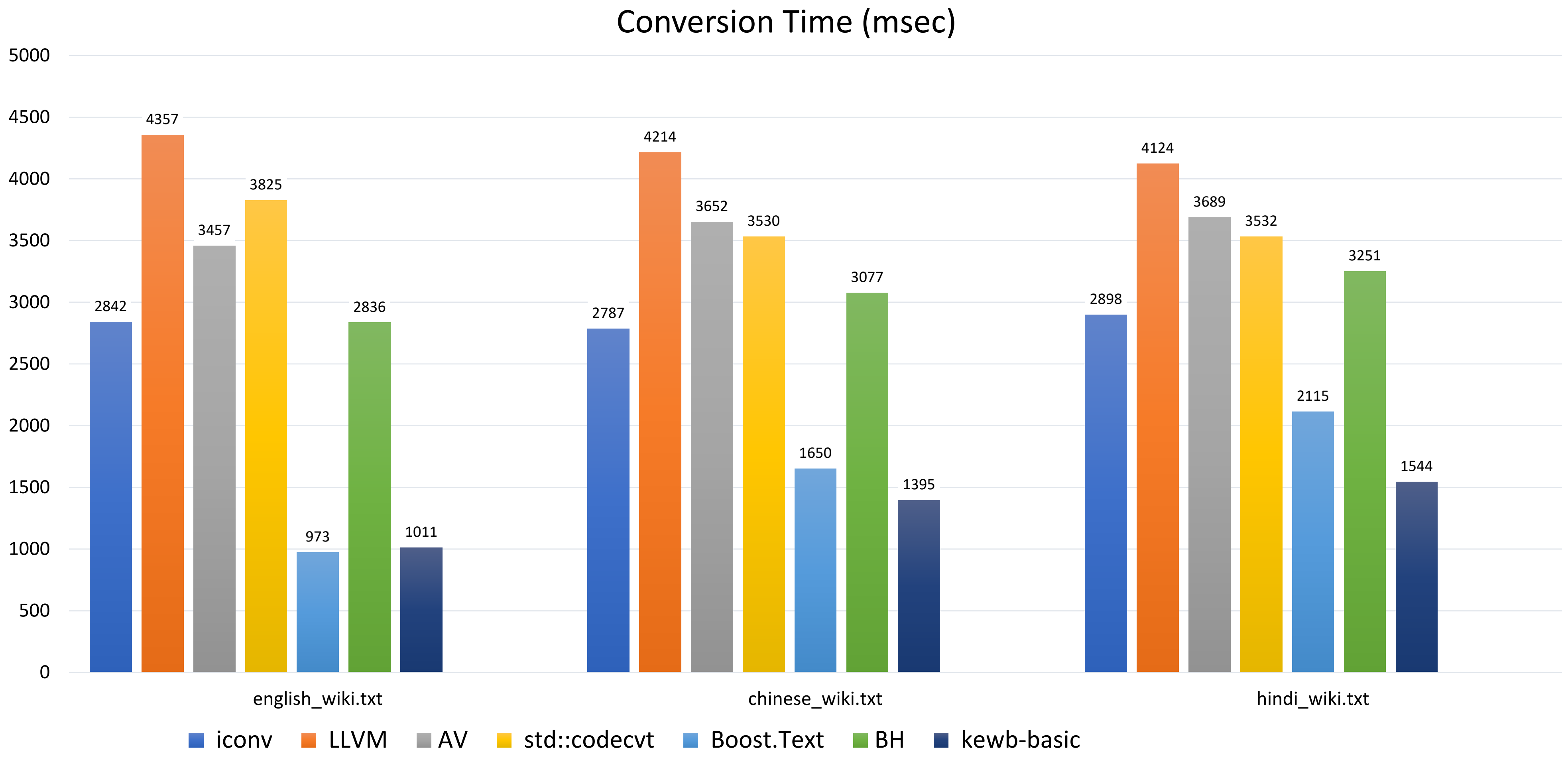
cdpt

U+2205 → ∅ EMPTY SET

The Basic Conversion Algorithm (UTF-8 to UTF-32)

```
KEWB_ALIGN_FN std::ptrdiff_t
UtfUtils::BasicConvert(char8_t const* pSrc, char8_t const* pSrcEnd, char32_t* pDst) noexcept
{
    char32_t*   pDstOrig = pDst;
    char32_t    cdpt;

    while (pSrc < pSrcEnd)
    {
        if (Advance(pSrc, pSrcEnd, cdpt) != ERR)
        {
            *pDst++ = cdpt;
        }
        else
        {
            return -1;
        }
    }
    return pDst - pDstOrig;
}
```

Basic Conversion Performance Overview (Linux/GCC)



Optimizing for ASCII

The Basic Conversion Algorithm (UTF-8 to UTF-32)

```
KEWB_ALIGN_FN std::ptrdiff_t
UtfUtils::BasicConvert(char8_t const* pSrc, char8_t const* pSrcEnd, char32_t* pDst) noexcept
{
    char32_t*   pDstOrig = pDst;
    char32_t    cdpt;

    while (pSrc < pSrcEnd)
    {
        if (Advance(pSrc, pSrcEnd, cdpt) != ERR)
        {
            *pDst++ = cdpt;
        }
        else
        {
            return -1;
        }
    }
    return pDst - pDstOrig;
}
```

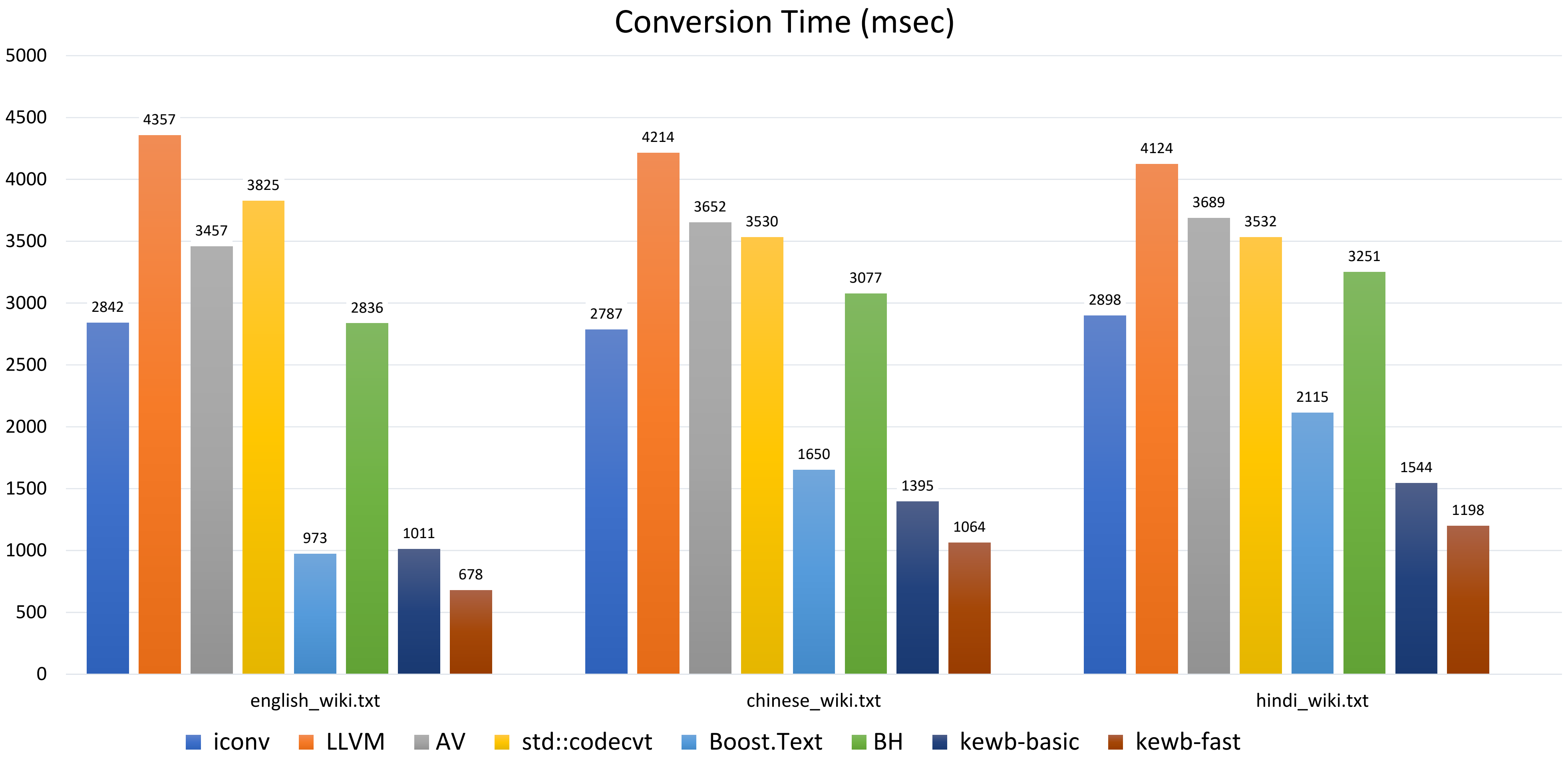
Converting a Single Code Point

```
KEWB_FORCE_INLINE int32_t
UtfUtils::Advance(char8_t const*& pSrc, char8_t const* pSrcEnd, char32_t& cdpt) noexcept
{
    FirstUnitInfo   info;   //- The descriptor for the first code unit
    char32_t        unit;   //- The current UTF-8 code unit
    int32_t         type;   //- The code unit's character class
    int32_t         curr;   //- The current DFA state

    info = smTables.maFirstUnitTable[*pSrc++];      //- Look up the first code unit descriptor
    cdpt = info.mFirstOctet;                        //- Get the initial code point value
    curr = info.mNextState;                         //- Get the second state

    while (curr > ERR)                              //- Loop over subsequent code units
    {
        ...
    }
    return curr;
}
```

The ASCII-Optimized Conversion Algorithm (UTF-8 to UTF-32)

```
KEWB_ALIGN_FN std::ptrdiff_t
UtfUtils::FastConvert(char8_t const* pSrc, char8_t const* pSrcEnd, char32_t* pDst) noexcept
{
    char32_t*   pDstOrig = pDst;
    char32_t    cdpt;

    while (pSrc < pSrcEnd)
    {
        if (*pSrc < 0x80)
        {
            *pDst++ = *pSrc++;
        }
        else
        {
            if (Advance(pSrc, pSrcEnd, cdpt) != ERR)
            {
                *pDst++ = cdpt;
            }
            else
            {
                return -1;
            }
        }
    }
    return pDst - pDstOrig;
}
```

ASCII-Optimized Conversion Performance Overview (Linux/GCC)



Optimizing for ASCII with SSE

The ASCII-Optimized Conversion Algorithm (UTF-8 to UTF-32)

```
KEWB_ALIGN_FN std::ptrdiff_t
UtfUtils::FastConvert(char8_t const* pSrc, char8_t const* pSrcEnd, char32_t* pDst) noexcept
{
    char32_t*   pDstOrig = pDst;
    char32_t    cdpt;

    while (pSrc < pSrcEnd)
    {
        if (*pSrc < 0x80)
        {
            *pDst++ = *pSrc++;
        }
        else
        {
            if (Advance(pSrc, pSrcEnd, cdpt) != ERR)
            {
                *pDst++ = cdpt;
            }
            else
            {
                return -1;
            }
        }
    }
    return pDst - pDstOrig;
}
```

The SSE-Optimized Conversion Algorithm (UTF-8 to UTF-32)

```
KEWB_ALIGN_FN std::ptrdiff_t
UtfUtils::SseConvert(char8_t const* pSrc, char8_t const* pSrcEnd, char32_t* pDst) noexcept
{
    char32_t*   pDstOrig = pDst;
    char32_t    cdpt;

    while (pSrc < (pSrcEnd - sizeof(__m128i)))
    {
        if (*pSrc < 0x80)
        {
            ConvertAsciiWithSse(pSrc, pDst);
        }
        else
        {
            if (Advance(pSrc, pSrcEnd, cdpt) != ERR)
            {
                *pDst++ = cdpt;
            }
            else
            {
                return -1;
            }
        }
    }
    ...
}
```

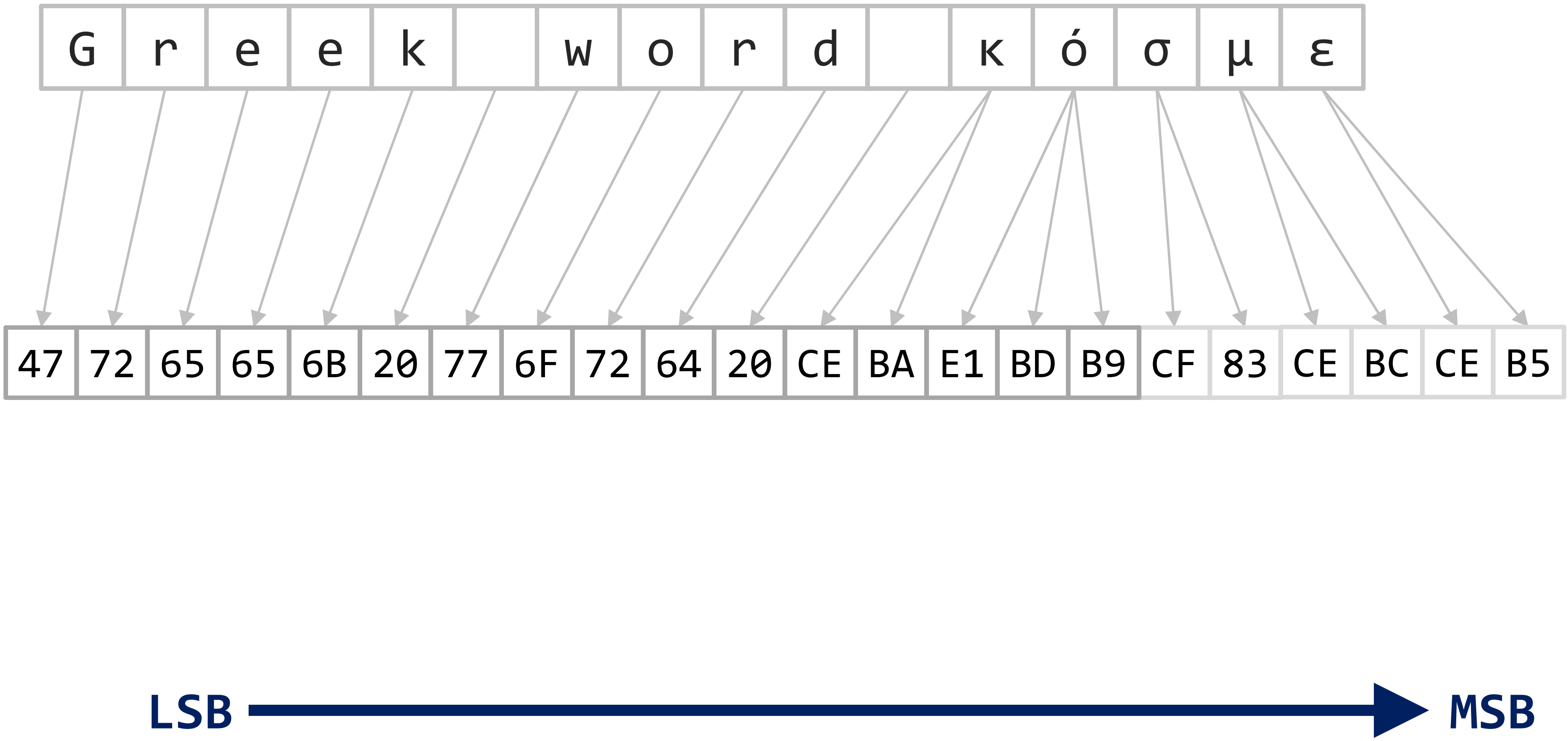
The SSE-Optimized Conversion Algorithm (UTF-8 to UTF-32)

```
KEWB_ALIGN_FN std::ptrdiff_t
UtfUtils::SseConvert(char8_t const* pSrc, char8_t const* pSrcEnd, char32_t* pDst) noexcept
{

    ...

    while (pSrc < pSrcEnd)
    {
        if (*pSrc < 0x80)
        {
            *pDst++ = *pSrc++;
        }
        else
        {
            if (Advance(pSrc, pSrcEnd, cdpt) != ERR)
            {
                *pDst++ = cdpt;
            }
            else
            {
                return -1;
            }
        }
    }
    return pDst - pDstOrig;
}
```

Converting ASCII Character Runs - Overview

```
KEWB_FORCE_INLINE void
UtfUtils::ConvertAsciiWithSse(char8_t const*& pSrc, char32_t*& pDst) noexcept
{
    __m128i     chunk, half, qrtr, zero;            //- SSE “registers”
    int32_t     mask, incr;                         //- ASCII bit mask and advancement

    zero  = _mm_set1_epi8(0);                       //- Zero out the interleave register
    chunk = _mm_loadu_si128((__m128i const*) pSrc); //- Load a register with 8-bit values
    mask  = _mm_movemask_epi8(chunk);               //- Find the octets with high bit set

    half = _mm_unpacklo_epi8(chunk, zero);          //- Unpack bytes 0-7 into 16-bit words
    qrtr = _mm_unpacklo_epi16(half, zero);          //- Unpack words 0-3 into 32-bit dwords
    _mm_storeu_si128((__m128i*) pDst, qrtr);        //- Write to memory
    qrtr = _mm_unpackhi_epi16(half, zero);          //- Unpack words 4-7 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 4), qrtr);  //- Write to memory

    half = _mm_unpackhi_epi8(chunk, zero);          //- Unpack bytes 8-15 into 16-bit words
    qrtr = _mm_unpacklo_epi16(half, zero);          //- Unpack words 8-11 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 8), qrtr);  //- Write to memory
    qrtr = _mm_unpackhi_epi16(half, zero);          //- Unpack words 12-15 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 12), qrtr); //- Write to memory

    //- If no bits were set in the mask, then all 16 code units were ASCII.
    //
    if (mask == 0)
    {
        pSrc += 16;
        pDst += 16;
    }

    //- Otherwise, the number of trailing (low-order) zero bits in the mask is
    //  the number of ASCII code units starting from the lowest byte address.
    else
    {
        incr  = GetTrailingZeros(mask);
        pSrc += incr;
        pDst += incr;
    }
}
```

Converting ASCII Character Runs – SSE Example



G	r	e	e	k		w	o	r	d		κ	ό	σ	μ	ε

47	72	65	65	6B	20	77	6F	72	64	20	CE	BA	E1	BD	B9	CF	83	CE	BC	CE	B5

Converting ASCII Character Runs

```
KEWB_FORCE_INLINE void
UtfUtils::ConvertAsciiWithSse(char8_t const*& pSrc, char32_t*& pDst) noexcept
{
    __m128i     chunk, half, qrtr, zero;            //- SSE “registers”
    int32_t     mask, incr;                         //- ASCII bit mask and advancement

    zero  = _mm_set1_epi8(0);                       //- Zero out the interleave register
    chunk = _mm_loadu_si128((__m128i const*) pSrc); //- Load a register with 8-bit values
    mask  = _mm_movemask_epi8(chunk);               //- Find octets with high bit set

    half = _mm_unpacklo_epi8(chunk, zero);          //- Unpack bytes 0-7 into 16-bit words
    qrtr = _mm_unpacklo_epi16(half, zero);          //- Unpack words 0-3 into 32-bit dwords
    _mm_storeu_si128((__m128i*) pDst, qrtr);        //- Write to memory
    qrtr = _mm_unpackhi_epi16(half, zero);          //- Unpack words 4-7 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 4), qrtr);  //- Write to memory

    ...
}
```

Converting ASCII Character Runs

```
KEWB_FORCE_INLINE void
UtfUtils::ConvertAsciiWithSse(char8_t const*& pSrc, char32_t*& pDst) noexcept
{
    __m128i     chunk, half, qrtr, zero;                 //- SSE “registers”
    int32_t     mask, incr;                              //- ASCII bit mask and advancement

    zero  = _mm_set1_epi8(0);                            //- Zero out the interleave register
    chunk = _mm_loadu_si128((__m128i const*) pSrc);      //- Load a register with 8-bit values
    mask  = _mm_movemask_epi8(chunk);                    //- Find octets with high bit set

    half = _mm_unpacklo_epi8(chunk, zero);               //- Unpack bytes 0-7 into 16-bit words
    qrtr = _mm_unpacklo_epi16(half, zero);               //- Unpack words 0-3 into 32-bit dwords
    _mm_storeu_si128((__m128i*) pDst, qrtr);             //- Write to memory
    qrtr = _mm_unpackhi_epi16(half, zero);               //- Unpack words 4-7 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 4), qrtr);       //- Write to memory

    ...
}
```

Converting ASCII Character Runs

```
KEWB_FORCE_INLINE void
UtfUtils::ConvertAsciiWithSse(char8_t const*& pSrc, char32_t*& pDst) noexcept
{
    __m128i     chunk, half, qrtr, zero;                //- SSE “registers”
    int32_t     mask, incr;                             //- ASCII bit mask and advancement

    zero  = _mm_set1_epi8(0);                           //- Zero out the interleave register
    chunk = _mm_loadu_si128((__m128i const*) pSrc);     //- Load a register with 8-bit values
    mask  = _mm_movemask_epi8(chunk);                   //- Find octets with high bit set

    half = _mm_unpacklo_epi8(chunk, zero);              //- Unpack bytes 0-7 into 16-bit words
    qrtr = _mm_unpacklo_epi16(half, zero);              //- Unpack words 0-3 into 32-bit dwords
    _mm_storeu_si128((__m128i*) pDst, qrtr);            //- Write to memory
    qrtr = _mm_unpackhi_epi16(half, zero);              //- Unpack words 4-7 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 4), qrtr);      //- Write to memory

    ...
}
```

Converting ASCII Character Runs – SSE Example

```
zero = _mm_set1_epi8(0)
```

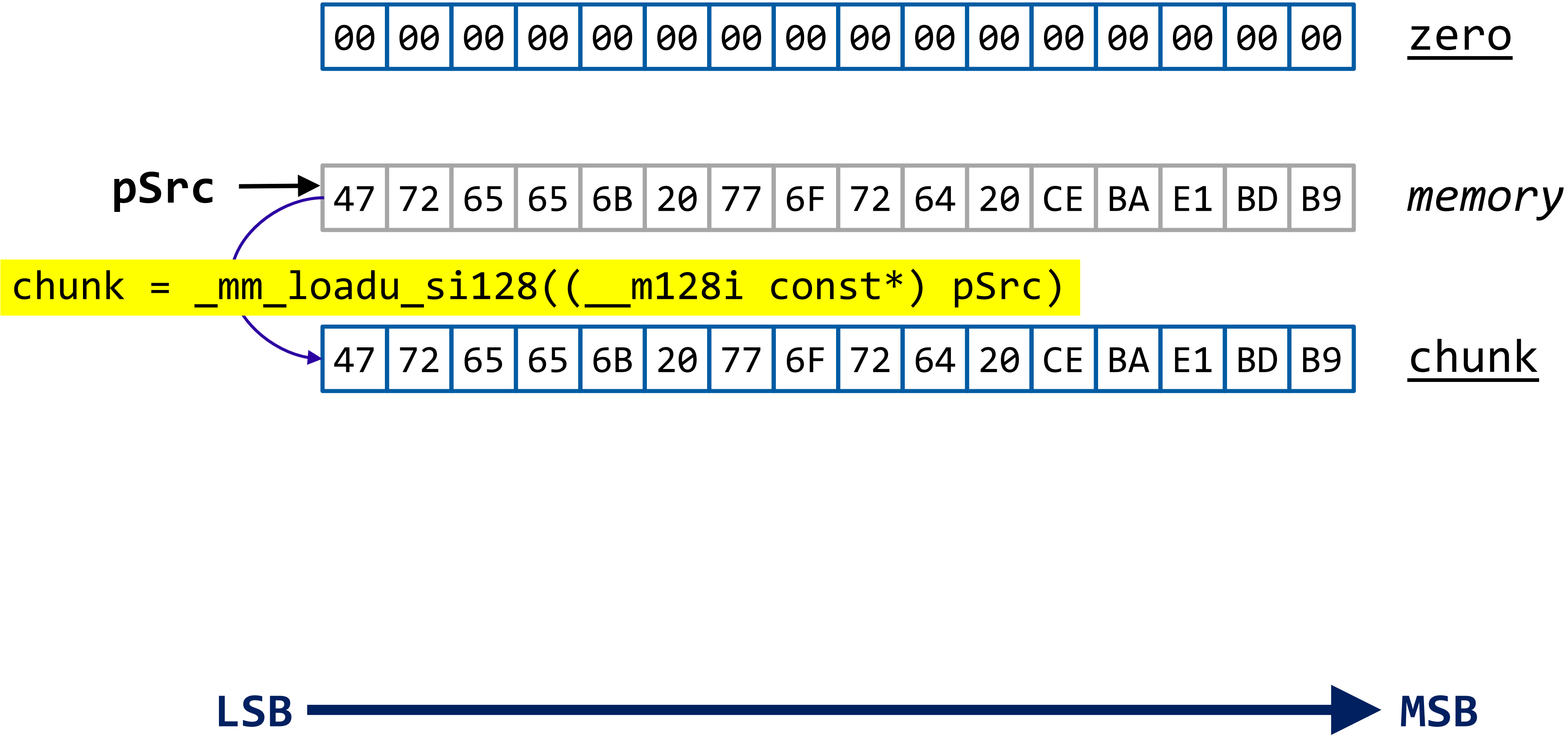
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	zero

LSB → **MSB**

Converting ASCII Character Runs

```
KEWB_FORCE_INLINE void
UtfUtils::ConvertAsciiWithSse(char8_t const*& pSrc, char32_t*& pDst) noexcept
{
    __m128i     chunk, half, qrtr, zero;            //- SSE “registers”
    int32_t     mask, incr;                         //- ASCII bit mask and advancement

    zero  = _mm_set1_epi8(0);                       //- Zero out the interleave register
    chunk = _mm_loadu_si128((__m128i const*) pSrc); //- Load a register with 8-bit values
    mask  = _mm_movemask_epi8(chunk);               //- Find octets with high bit set

    half = _mm_unpacklo_epi8(chunk, zero);          //- Unpack bytes 0-7 into 16-bit words
    qrtr = _mm_unpacklo_epi16(half, zero);          //- Unpack words 0-3 into 32-bit dwords
    _mm_storeu_si128((__m128i*) pDst, qrtr);        //- Write to memory
    qrtr = _mm_unpackhi_epi16(half, zero);          //- Unpack words 4-7 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 4), qrtr);  //- Write to memory

    ...
}
```

Converting ASCII Character Runs – SSE Example



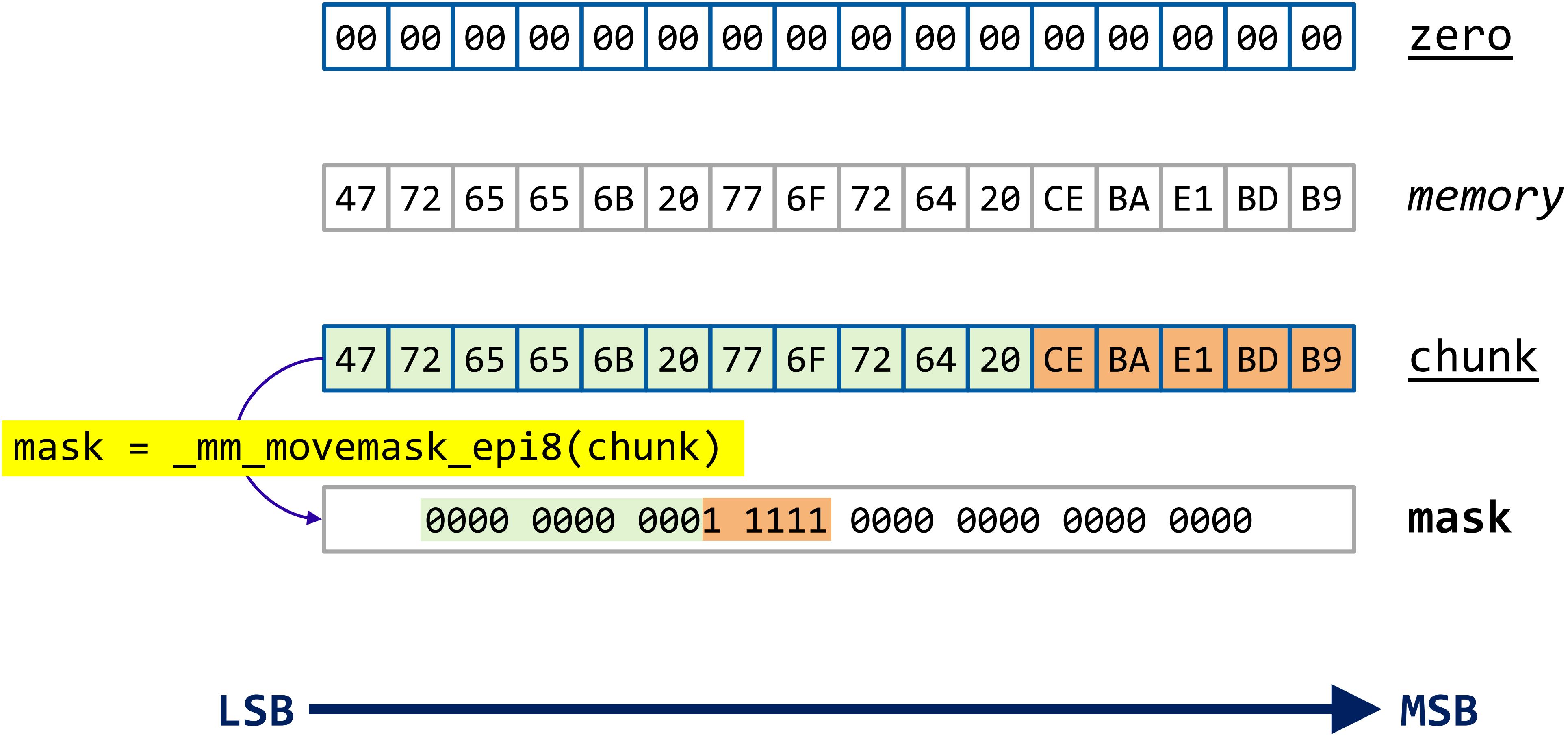
	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	zero
pSrc	47	72	65	65	6B	20	77	6F	72	64	20	CE	BA	E1	BD	B9	*memory*
	47	72	65	65	6B	20	77	6F	72	64	20	CE	BA	E1	BD	B9	chunk

```
chunk = _mm_loadu_si128((__m128i const*) pSrc)
```

Converting ASCII Character Runs

```
KEWB_FORCE_INLINE void
UtfUtils::ConvertAsciiWithSse(char8_t const*& pSrc, char32_t*& pDst) noexcept
{
    __m128i     chunk, half, qrtr, zero;                //- SSE “registers”
    int32_t     mask, incr;                             //- ASCII bit mask and advancement

    zero  = _mm_set1_epi8(0);                           //- Zero out the interleave register
    chunk = _mm_loadu_si128((__m128i const*) pSrc);     //- Load a register with 8-bit values
    mask  = _mm_movemask_epi8(chunk);                   //- Find octets with high bit set

    half = _mm_unpacklo_epi8(chunk, zero);              //- Unpack bytes 0-7 into 16-bit words
    qrtr = _mm_unpacklo_epi16(half, zero);              //- Unpack words 0-3 into 32-bit dwords
    _mm_storeu_si128((__m128i*) pDst, qrtr);            //- Write to memory
    qrtr = _mm_unpackhi_epi16(half, zero);              //- Unpack words 4-7 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 4), qrtr);      //- Write to memory

    ...
}
```

Converting ASCII Character Runs – SSE Example



00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	zero
47	72	65	65	6B	20	77	6F	72	64	20	CE	BA	E1	BD	B9	*memory*
47	72	65	65	6B	20	77	6F	72	64	20	CE	BA	E1	BD	B9	chunk

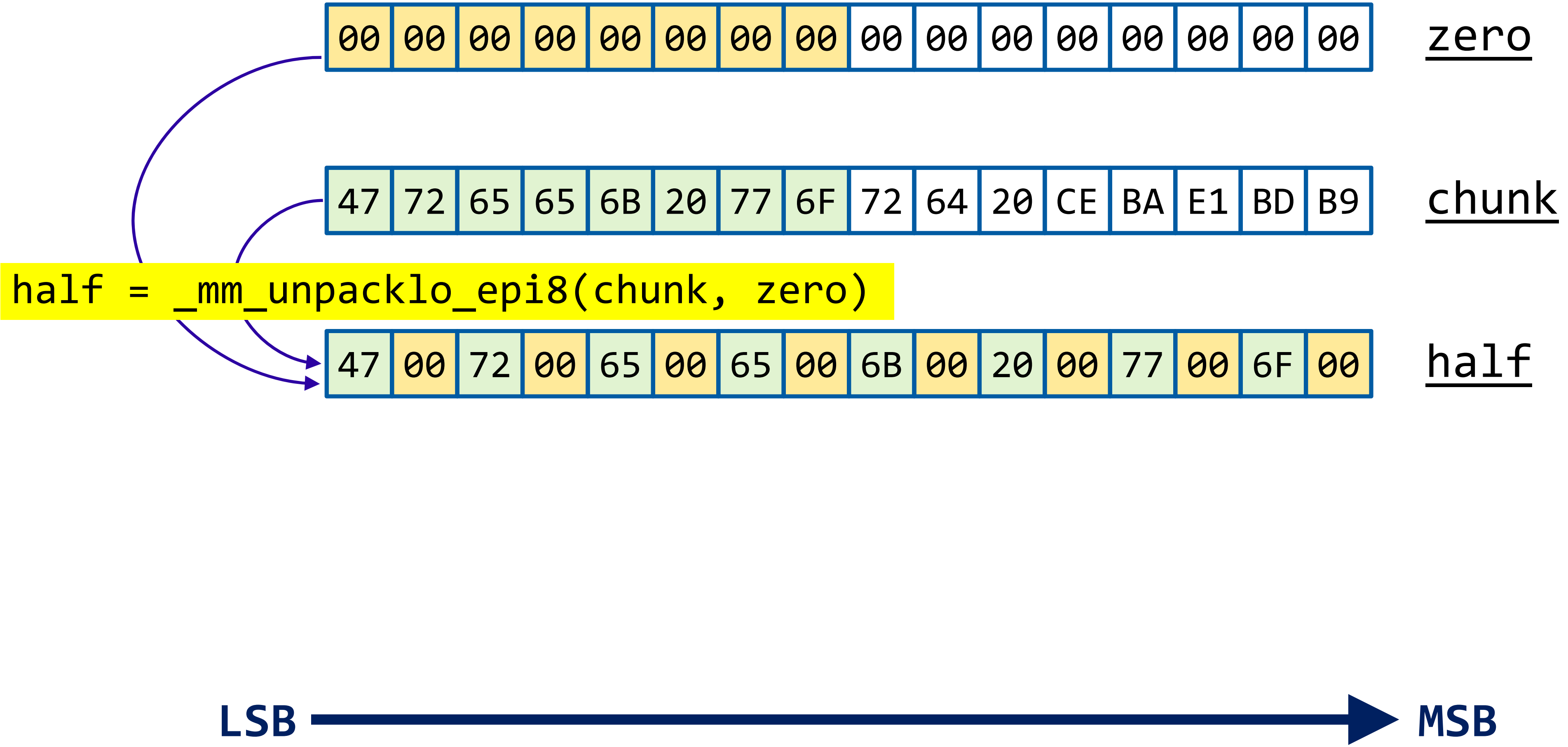
```
mask = _mm_movemask_epi8(chunk)
```

0000 0000 0001 1111 0000 0000 0000 0000 **mask**

Converting ASCII Character Runs

```
KEWB_FORCE_INLINE void
UtfUtils::ConvertAsciiWithSse(char8_t const*& pSrc, char32_t*& pDst) noexcept
{
    __m128i     chunk, half, qrtr, zero;            //- SSE “registers”
    int32_t     mask, incr;                         //- ASCII bit mask and advancement

    zero  = _mm_set1_epi8(0);                       //- Zero out the interleave register
    chunk = _mm_loadu_si128((__m128i const*) pSrc); //- Load a register with 8-bit values
    mask  = _mm_movemask_epi8(chunk);               //- Find octets with high bit set

    half = _mm_unpacklo_epi8(chunk, zero);          //- Unpack bytes 0-7 into 16-bit words
    qrtr = _mm_unpacklo_epi16(half, zero);          //- Unpack words 0-3 into 32-bit dwords
    _mm_storeu_si128((__m128i*) pDst, qrtr);        //- Write to memory
    qrtr = _mm_unpackhi_epi16(half, zero);          //- Unpack words 4-7 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 4), qrtr);  //- Write to memory

    ...
}
```

Converting ASCII Character Runs – SSE Example



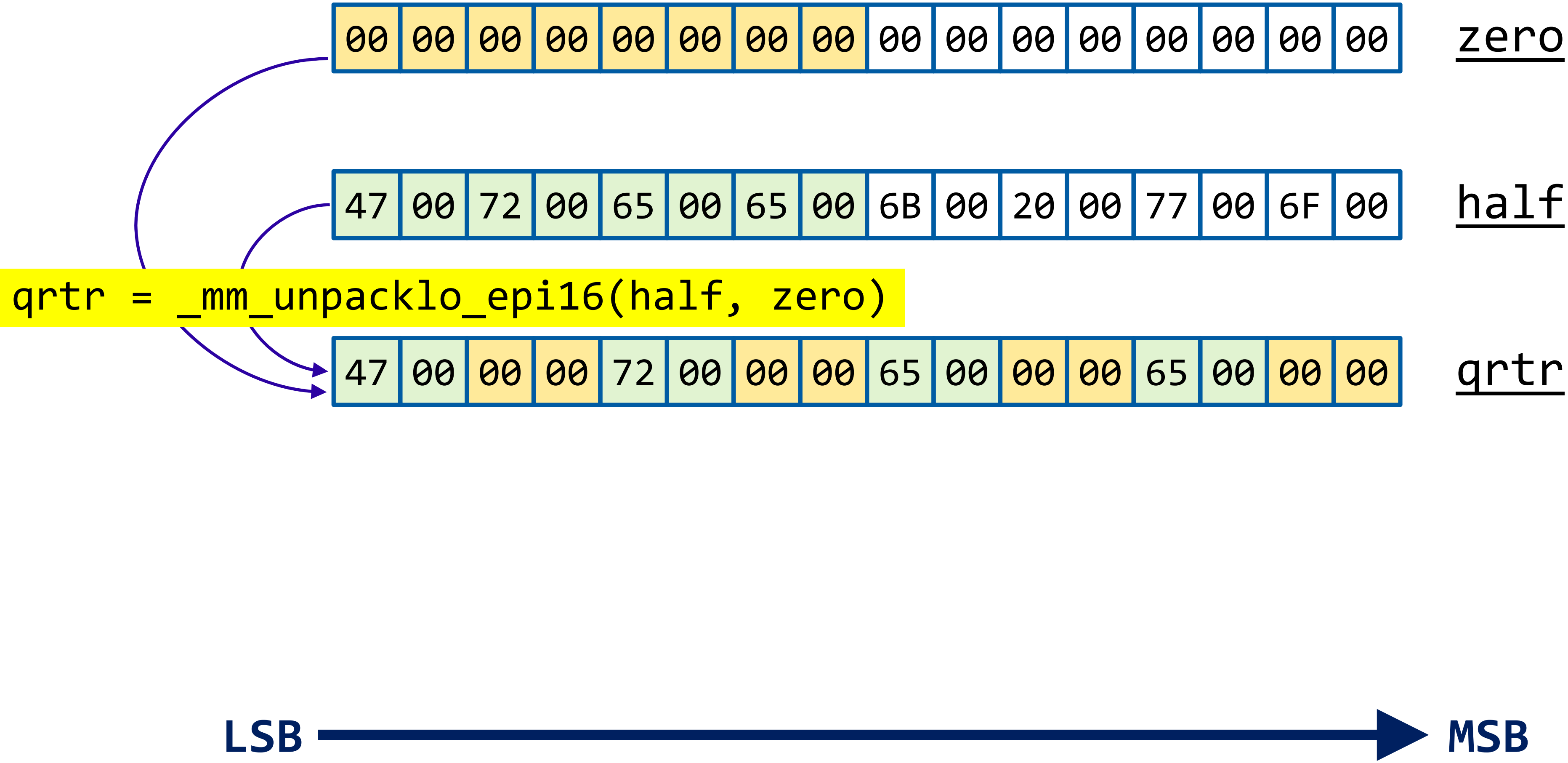
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	zero
47	72	65	65	6B	20	77	6F	72	64	20	CE	BA	E1	BD	B9	chunk
47	00	72	00	65	00	65	00	6B	00	20	00	77	00	6F	00	half

```
half = _mm_unpacklo_epi8(chunk, zero)
```

Converting ASCII Character Runs

```
KEWB_FORCE_INLINE void
UtfUtils::ConvertAsciiWithSse(char8_t const*& pSrc, char32_t*& pDst) noexcept
{
    __m128i     chunk, half, qrtr, zero;            //- SSE “registers”
    int32_t     mask, incr;                         //- ASCII bit mask and advancement

    zero  = _mm_set1_epi8(0);                       //- Zero out the interleave register
    chunk = _mm_loadu_si128((__m128i const*) pSrc); //- Load a register with 8-bit values
    mask  = _mm_movemask_epi8(chunk);               //- Find octets with high bit set

    half = _mm_unpacklo_epi8(chunk, zero);          //- Unpack bytes 0-7 into 16-bit words
    qrtr = _mm_unpacklo_epi16(half, zero);          //- Unpack words 0-3 into 32-bit dwords
    _mm_storeu_si128((__m128i*) pDst, qrtr);        //- Write to memory
    qrtr = _mm_unpackhi_epi16(half, zero);          //- Unpack words 4-7 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 4), qrtr);  //- Write to memory

    ...
}
```

Converting ASCII Character Runs – SSE Example



00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	zero
47	00	72	00	65	00	65	00	6B	00	20	00	77	00	6F	00	half
47	00	00	00	72	00	00	00	65	00	00	00	65	00	00	00	qrtr

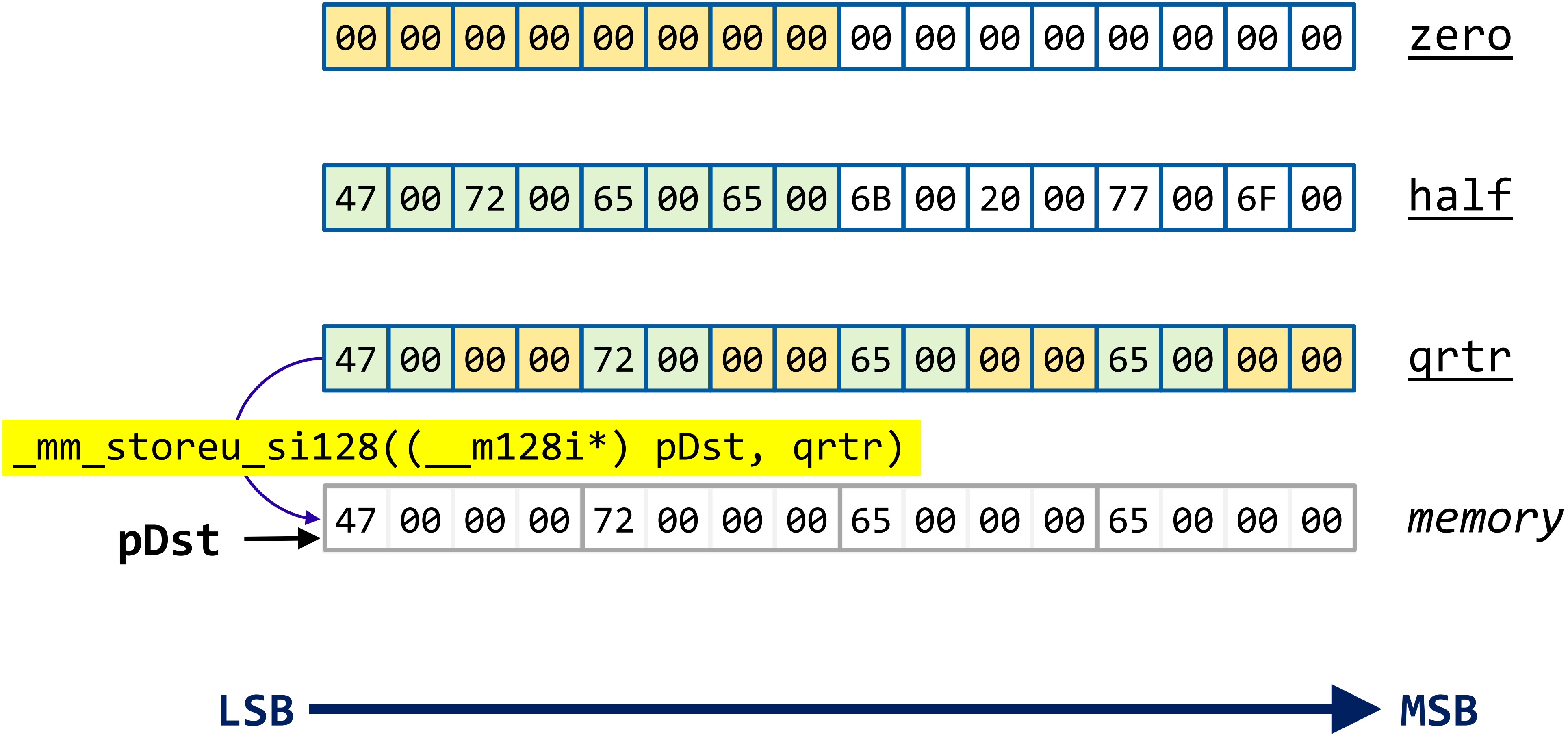
```
qrtr = _mm_unpacklo_epi16(half, zero)
```

LSB → MSB

Converting ASCII Character Runs

```
KEWB_FORCE_INLINE void
UtfUtils::ConvertAsciiWithSse(char8_t const*& pSrc, char32_t*& pDst) noexcept
{
    __m128i     chunk, half, qrtr, zero;            //- SSE “registers”
    int32_t     mask, incr;                         //- ASCII bit mask and advancement

    zero  = _mm_set1_epi8(0);                       //- Zero out the interleave register
    chunk = _mm_loadu_si128((__m128i const*) pSrc); //- Load a register with 8-bit values
    mask  = _mm_movemask_epi8(chunk);               //- Find octets with high bit set

    half = _mm_unpacklo_epi8(chunk, zero);          //- Unpack bytes 0-7 into 16-bit words
    qrtr = _mm_unpacklo_epi16(half, zero);          //- Unpack words 0-3 into 32-bit dwords
    _mm_storeu_si128((__m128i*) pDst, qrtr);        //- Write to memory
    qrtr = _mm_unpackhi_epi16(half, zero);          //- Unpack words 4-7 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 4), qrtr);  //- Write to memory

    ...
}
```

Converting ASCII Character Runs – SSE Example



00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	zero
47	00	72	00	65	00	65	00	6B	00	20	00	77	00	6F	00	half
47	00	00	00	72	00	00	00	65	00	00	00	65	00	00	00	qrtr
47	00	00	00	72	00	00	00	65	00	00	00	65	00	00	00	*memory*

```
_mm_storeu_si128((__m128i*) pDst, qrtr)
```

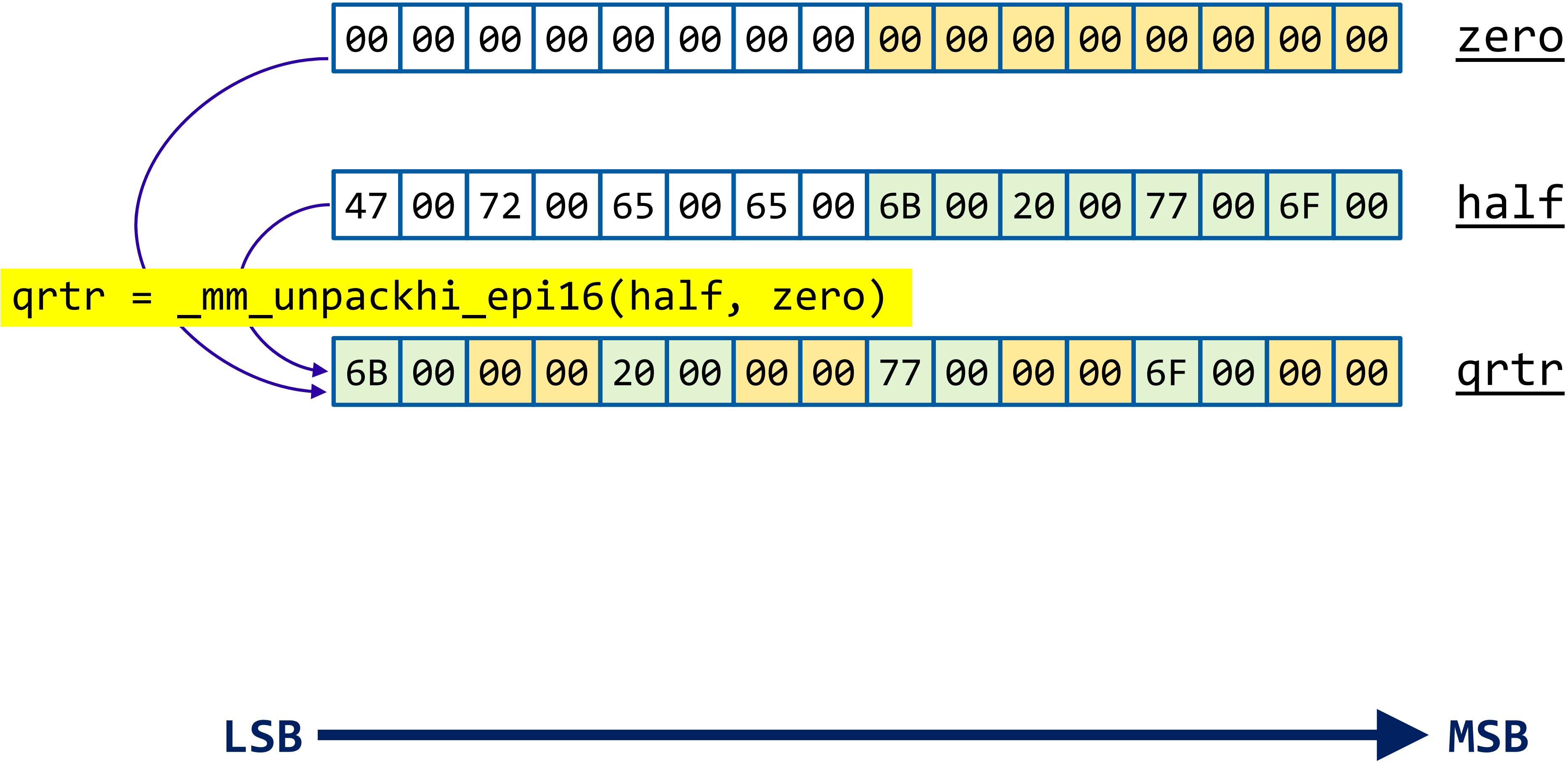
pDst →

LSB → **MSB**

Converting ASCII Character Runs

```
KEWB_FORCE_INLINE void
UtfUtils::ConvertAsciiWithSse(char8_t const*& pSrc, char32_t*& pDst) noexcept
{
    __m128i     chunk, half, qrtr, zero;            //- SSE “registers”
    int32_t     mask, incr;                         //- ASCII bit mask and advancement

    zero  = _mm_set1_epi8(0);                       //- Zero out the interleave register
    chunk = _mm_loadu_si128((__m128i const*) pSrc); //- Load a register with 8-bit values
    mask  = _mm_movemask_epi8(chunk);               //- Find octets with high bit set

    half = _mm_unpacklo_epi8(chunk, zero);          //- Unpack bytes 0-7 into 16-bit words
    qrtr = _mm_unpacklo_epi16(half, zero);          //- Unpack words 0-3 into 32-bit dwords
    _mm_storeu_si128((__m128i*) pDst, qrtr);        //- Write to memory
    qrtr = _mm_unpackhi_epi16(half, zero);          //- Unpack words 4-7 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 4), qrtr);  //- Write to memory

    ...
}
```

Converting ASCII Character Runs – SSE Example



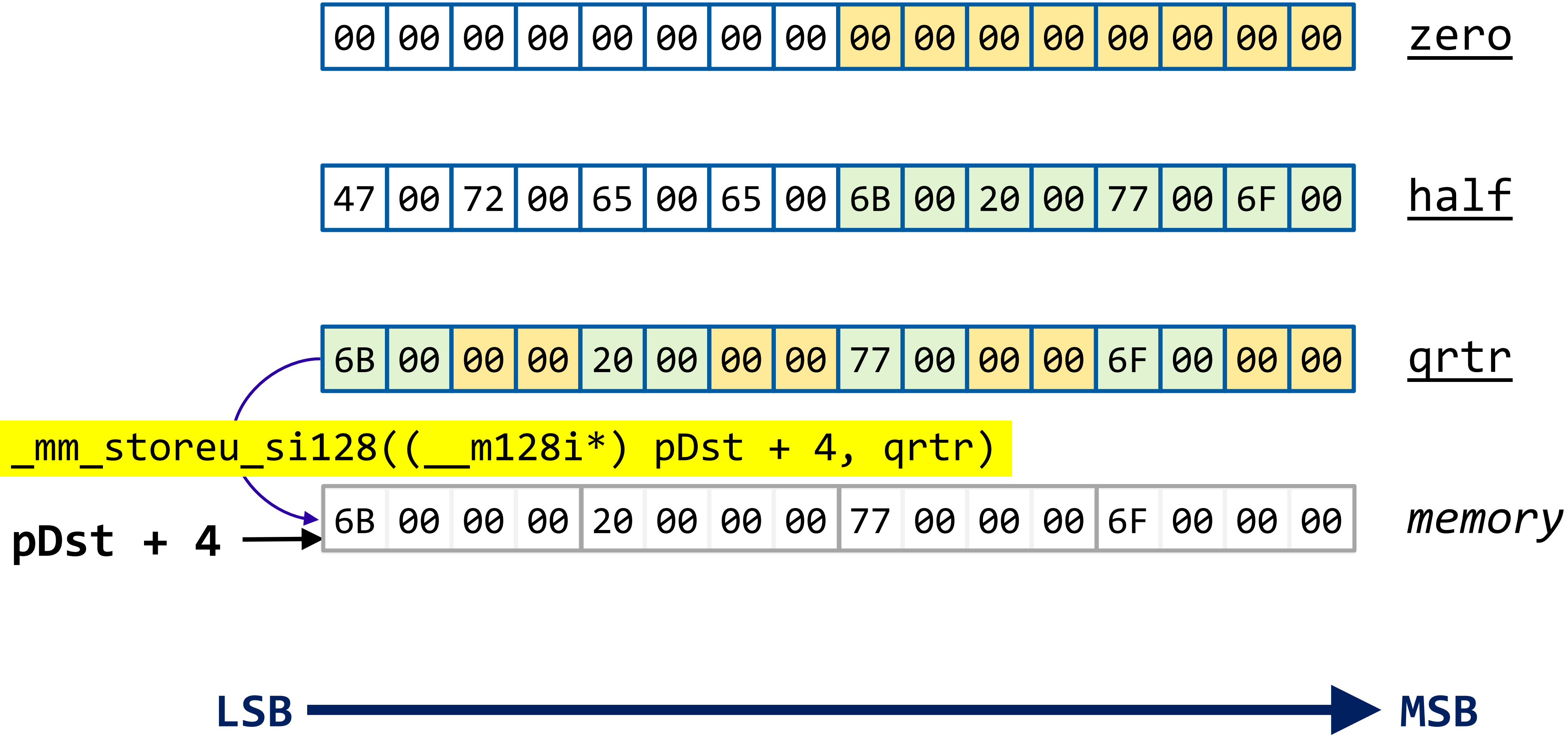
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	zero
47	00	72	00	65	00	65	00	6B	00	20	00	77	00	6F	00	half
6B	00	00	00	20	00	00	00	77	00	00	00	6F	00	00	00	qrtr

```
qrtr = _mm_unpackhi_epi16(half, zero)
```

Converting ASCII Character Runs

```
KEWB_FORCE_INLINE void
UtfUtils::ConvertAsciiWithSse(char8_t const*& pSrc, char32_t*& pDst) noexcept
{
    __m128i     chunk, half, qrtr, zero;            //- SSE “registers”
    int32_t     mask, incr;                         //- ASCII bit mask and advancement

    zero  = _mm_set1_epi8(0);                       //- Zero out the interleave register
    chunk = _mm_loadu_si128((__m128i const*) pSrc); //- Load a register with 8-bit values
    mask  = _mm_movemask_epi8(chunk);               //- Find octets with high bit set

    half = _mm_unpacklo_epi8(chunk, zero);          //- Unpack bytes 0-7 into 16-bit words
    qrtr = _mm_unpacklo_epi16(half, zero);          //- Unpack words 0-3 into 32-bit dwords
    _mm_storeu_si128((__m128i*) pDst, qrtr);        //- Write to memory
    qrtr = _mm_unpackhi_epi16(half, zero);          //- Unpack words 4-7 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 4), qrtr);  //- Write to memory

    ...
}
```

Converting ASCII Character Runs – SSE Example



00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	zero
47	00	72	00	65	00	65	00	6B	00	20	00	77	00	6F	00	half
6B	00	00	00	20	00	00	00	77	00	00	00	6F	00	00	00	qrtr
6B	00	00	00	20	00	00	00	77	00	00	00	6F	00	00	00	*memory*

```
_mm_storeu_si128((__m128i*) pDst + 4, qrtr)
```

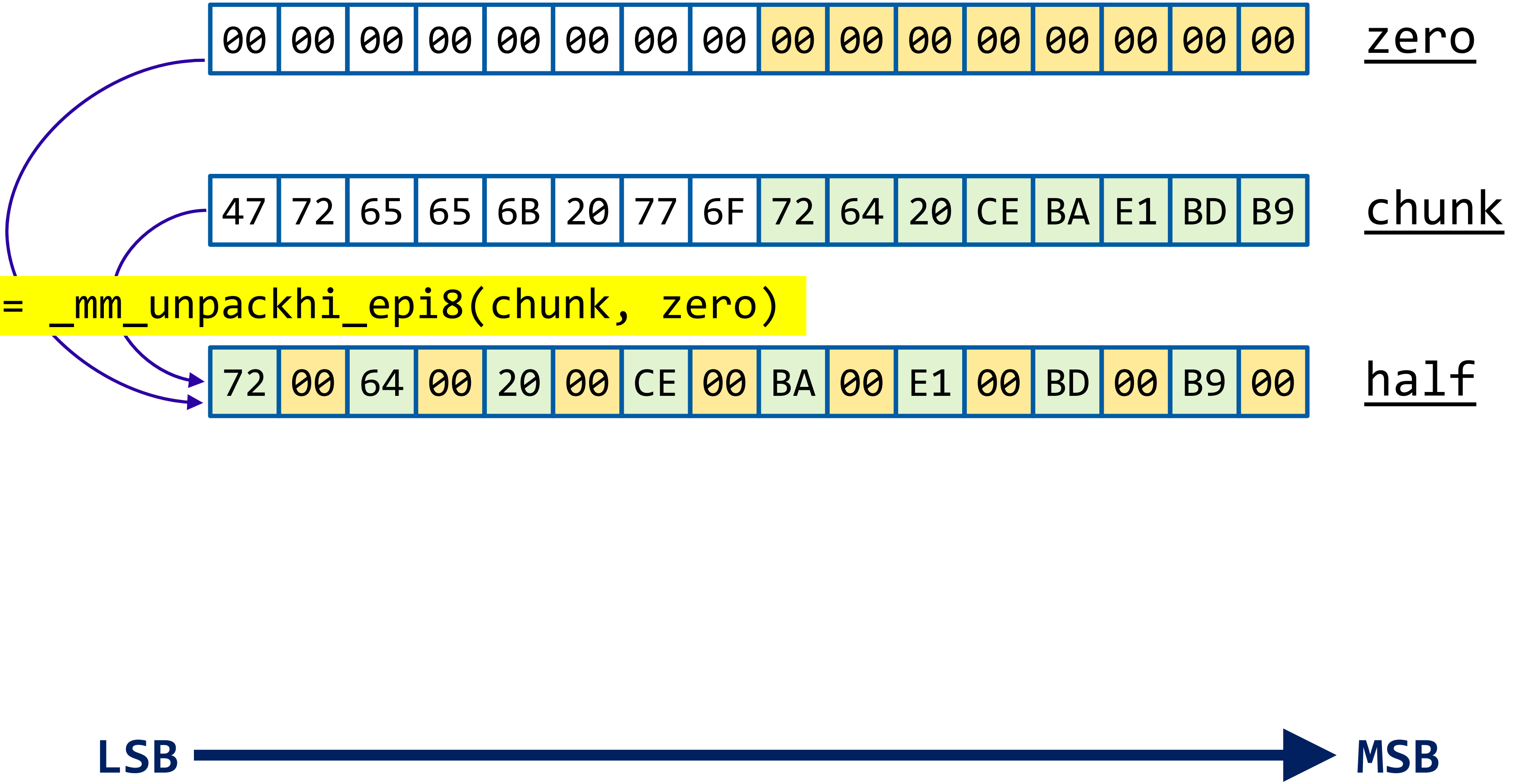
pDst + 4 →

LSB → **MSB**

Converting ASCII Character Runs

```
KEWB_FORCE_INLINE void
UtfUtils::ConvertAsciiWithSse(char8_t const*& pSrc, char32_t*& pDst) noexcept
{
    ...

    half = _mm_unpacklo_epi8(chunk, zero);          //- Unpack bytes 0-7 into 16-bit words
    qrtr = _mm_unpacklo_epi16(half, zero);          //- Unpack words 0-3 into 32-bit dwords
    _mm_storeu_si128((__m128i*) pDst, qrtr);        //- Write to memory
    qrtr = _mm_unpackhi_epi16(half, zero);          //- Unpack words 4-7 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 4), qrtr);  //- Write to memory

    half = _mm_unpackhi_epi8(chunk, zero);          //- Unpack bytes 8-15 into 16-bit words
    qrtr = _mm_unpacklo_epi16(half, zero);          //- Unpack words 8-11 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 8), qrtr);  //- Write to memory
    qrtr = _mm_unpackhi_epi16(half, zero);          //- Unpack words 12-15 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 12), qrtr); //- Write to memory

    ...
}
```

Converting ASCII Character Runs – SSE Example



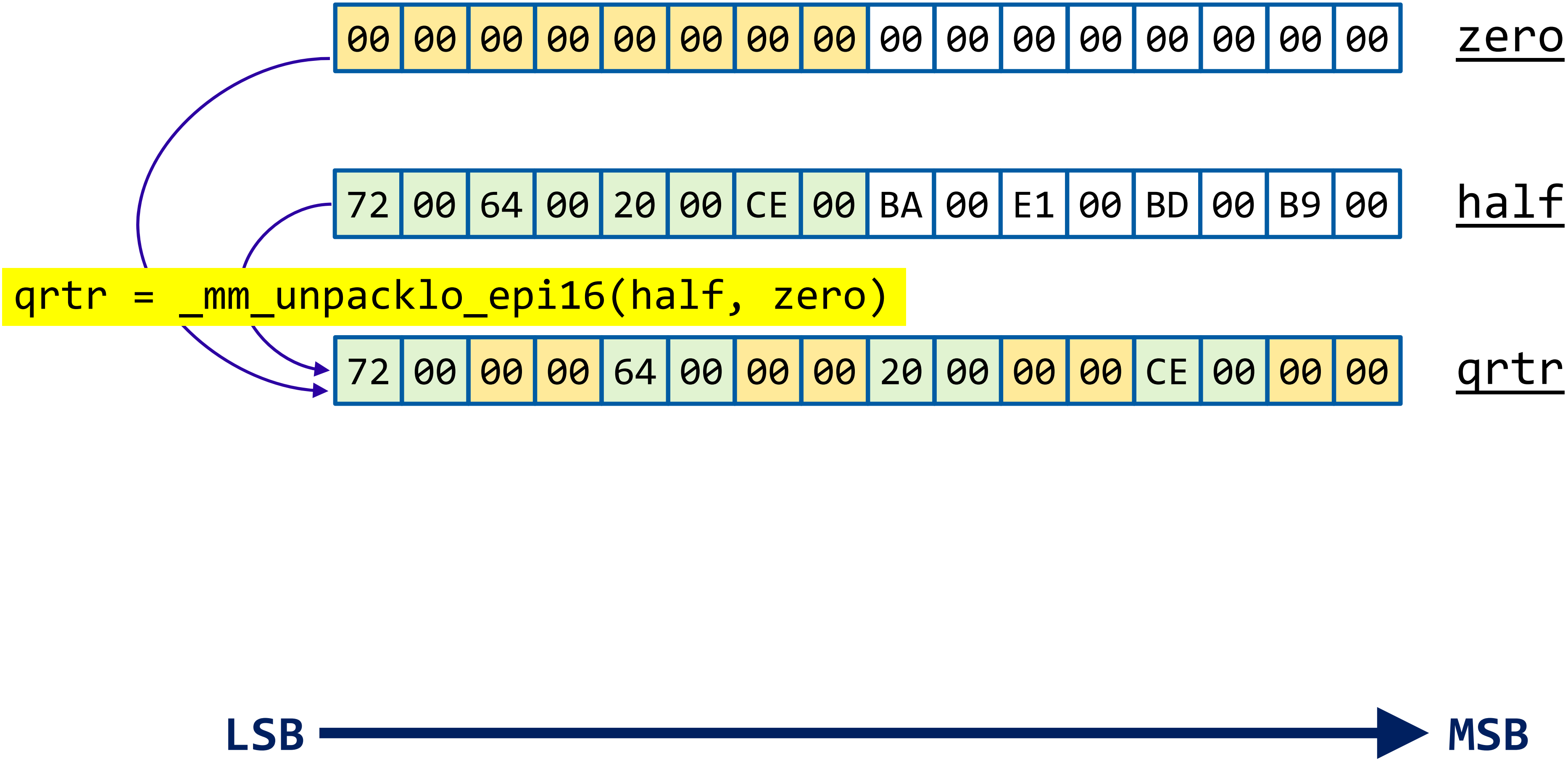
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	zero
47	72	65	65	6B	20	77	6F	72	64	20	CE	BA	E1	BD	B9	chunk
72	00	64	00	20	00	CE	00	BA	00	E1	00	BD	00	B9	00	half

```
half = _mm_unpackhi_epi8(chunk, zero)
```

Converting ASCII Character Runs

```
KEWB_FORCE_INLINE void
UtfUtils::ConvertAsciiWithSse(char8_t const*& pSrc, char32_t*& pDst) noexcept
{
    ...

    half = _mm_unpacklo_epi8(chunk, zero);          //- Unpack bytes 0-7 into 16-bit words
    qrtr = _mm_unpacklo_epi16(half, zero);          //- Unpack words 0-3 into 32-bit dwords
    _mm_storeu_si128((__m128i*) pDst, qrtr);        //- Write to memory
    qrtr = _mm_unpackhi_epi16(half, zero);          //- Unpack words 4-7 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 4), qrtr);  //- Write to memory

    half = _mm_unpackhi_epi8(chunk, zero);          //- Unpack bytes 8-15 into 16-bit words
    qrtr = _mm_unpacklo_epi16(half, zero);          //- Unpack words 8-11 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 8), qrtr);  //- Write to memory
    qrtr = _mm_unpackhi_epi16(half, zero);          //- Unpack words 12-15 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 12), qrtr); //- Write to memory

    ...
}
```

Converting ASCII Character Runs – SSE Example



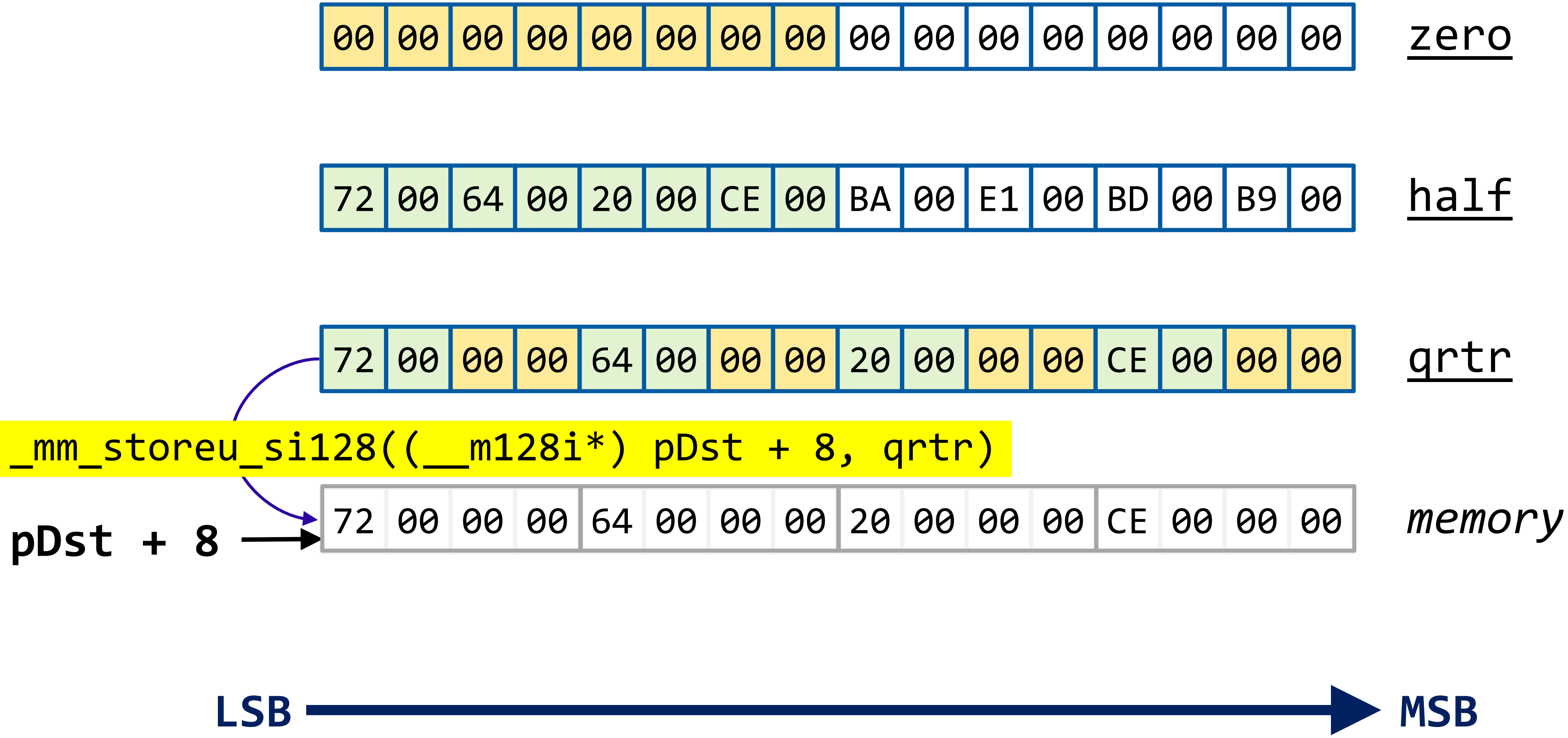
```
qrtr = _mm_unpacklo_epi16(half, zero)
```

00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	zero
72	00	64	00	20	00	CE	00	BA	00	E1	00	BD	00	B9	00	half
72	00	00	00	64	00	00	00	20	00	00	00	CE	00	00	00	qrtr

Converting ASCII Character Runs

```
KEWB_FORCE_INLINE void
UtfUtils::ConvertAsciiWithSse(char8_t const*& pSrc, char32_t*& pDst) noexcept
{
    ...

    half = _mm_unpacklo_epi8(chunk, zero);          //- Unpack bytes 0-7 into 16-bit words
    qrtr = _mm_unpacklo_epi16(half, zero);          //- Unpack words 0-3 into 32-bit dwords
    _mm_storeu_si128((__m128i*) pDst, qrtr);        //- Write to memory
    qrtr = _mm_unpackhi_epi16(half, zero);          //- Unpack words 4-7 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 4), qrtr);  //- Write to memory

    half = _mm_unpackhi_epi8(chunk, zero);          //- Unpack bytes 8-15 into 16-bit words
    qrtr = _mm_unpacklo_epi16(half, zero);          //- Unpack words 8-11 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 8), qrtr);  //- Write to memory
    qrtr = _mm_unpackhi_epi16(half, zero);          //- Unpack words 12-15 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 12), qrtr); //- Write to memory

    ...
}
```

Converting ASCII Character Runs – SSE Example



00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	zero
72	00	64	00	20	00	CE	00	BA	00	E1	00	BD	00	B9	00	half
72	00	00	00	64	00	00	00	20	00	00	00	CE	00	00	00	qrtr
72	00	00	00	64	00	00	00	20	00	00	00	CE	00	00	00	*memory*

```
_mm_storeu_si128((__m128i*) pDst + 8, qrtr)
```

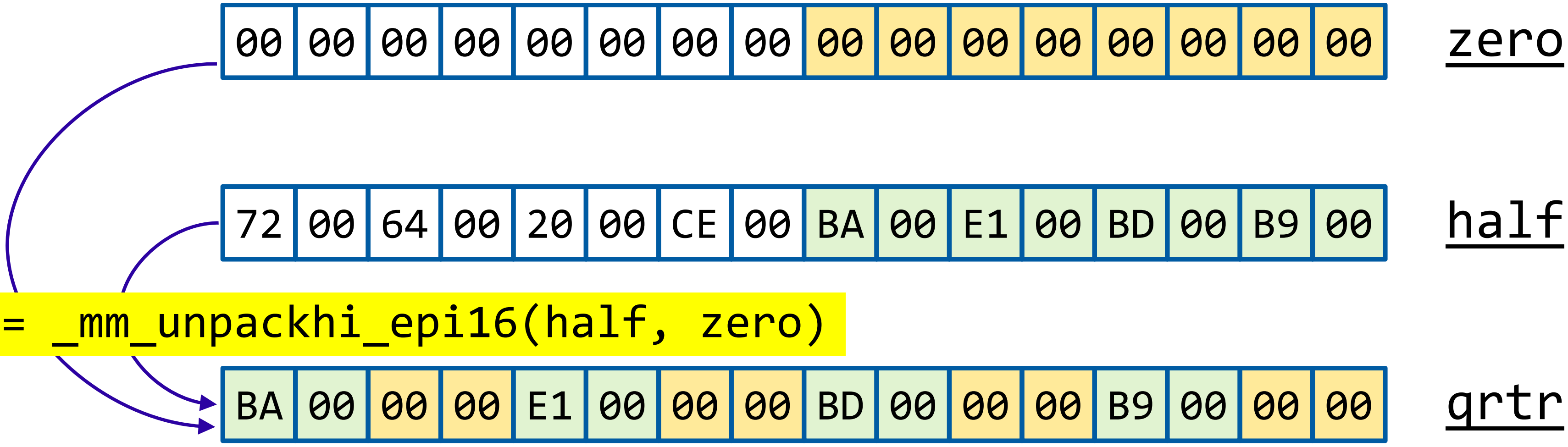
pDst + 8 →

LSB → **MSB**

Converting ASCII Character Runs

```
KEWB_FORCE_INLINE void
UtfUtils::ConvertAsciiWithSse(char8_t const*& pSrc, char32_t*& pDst) noexcept
{
    ...

    half = _mm_unpacklo_epi8(chunk, zero);          //- Unpack bytes 0-7 into 16-bit words
    qrtr = _mm_unpacklo_epi16(half, zero);          //- Unpack words 0-3 into 32-bit dwords
    _mm_storeu_si128((__m128i*) pDst, qrtr);        //- Write to memory
    qrtr = _mm_unpackhi_epi16(half, zero);          //- Unpack words 4-7 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 4), qrtr);  //- Write to memory

    half = _mm_unpackhi_epi8(chunk, zero);          //- Unpack bytes 8-15 into 16-bit words
    qrtr = _mm_unpacklo_epi16(half, zero);          //- Unpack words 8-11 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 8), qrtr);  //- Write to memory
    qrtr = _mm_unpackhi_epi16(half, zero);          //- Unpack words 12-15 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 12), qrtr); //- Write to memory

    ...
}
```

Converting ASCII Character Runs – SSE Example



00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	zero
72	00	64	00	20	00	CE	00	BA	00	E1	00	BD	00	B9	00	half
BA	00	00	00	E1	00	00	00	BD	00	00	00	B9	00	00	00	qrtr

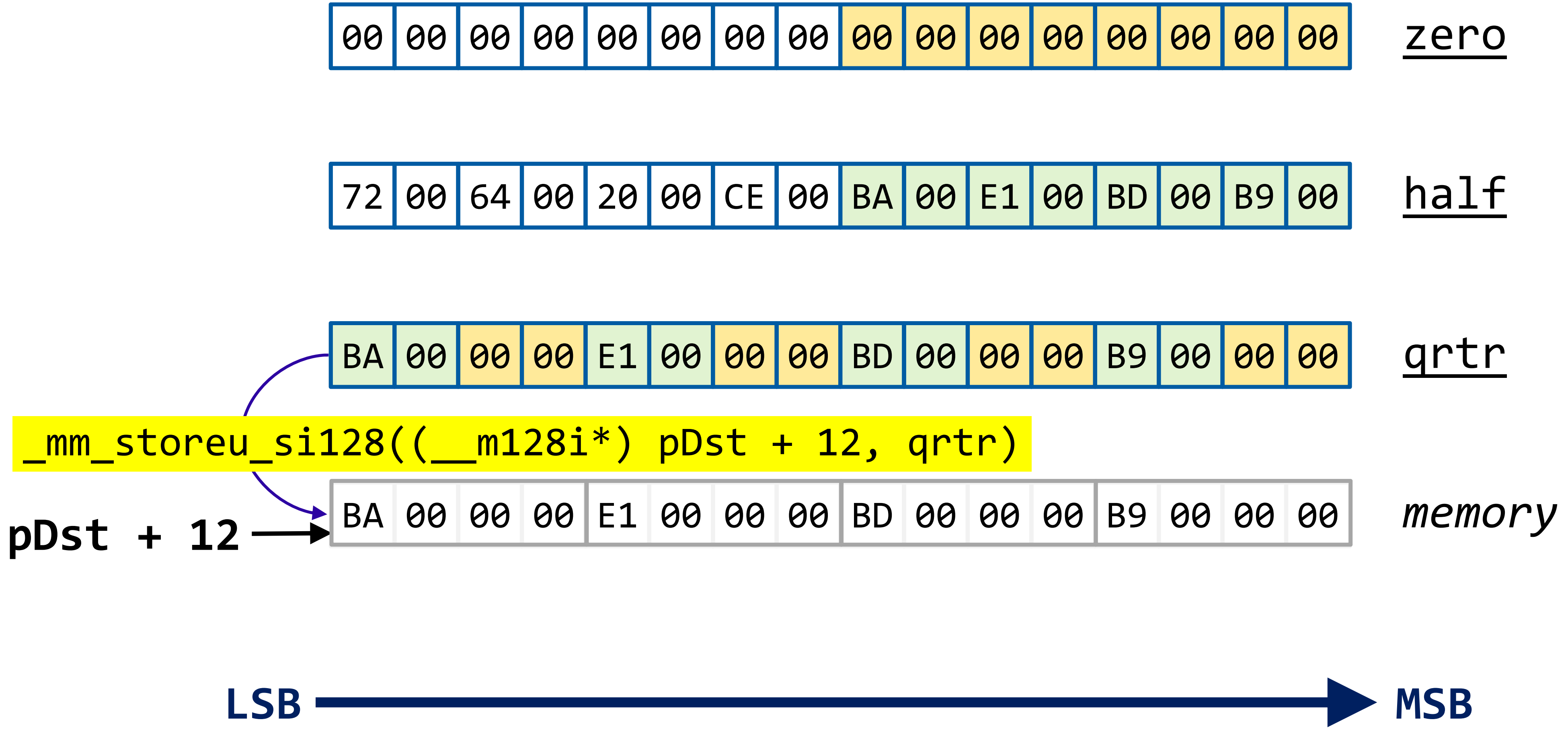
```
qrtr = _mm_unpackhi_epi16(half, zero)
```

LSB → MSB

Converting ASCII Character Runs

```
KEWB_FORCE_INLINE void
UtfUtils::ConvertAsciiWithSse(char8_t const*& pSrc, char32_t*& pDst) noexcept
{
    ...

    half = _mm_unpacklo_epi8(chunk, zero);          //- Unpack bytes 0-7 into 16-bit words
    qrtr = _mm_unpacklo_epi16(half, zero);          //- Unpack words 0-3 into 32-bit dwords
    _mm_storeu_si128((__m128i*) pDst, qrtr);        //- Write to memory
    qrtr = _mm_unpackhi_epi16(half, zero);          //- Unpack words 4-7 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 4), qrtr);  //- Write to memory

    half = _mm_unpackhi_epi8(chunk, zero);          //- Unpack bytes 8-15 into 16-bit words
    qrtr = _mm_unpacklo_epi16(half, zero);          //- Unpack words 8-11 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 8), qrtr);  //- Write to memory
    qrtr = _mm_unpackhi_epi16(half, zero);          //- Unpack words 12-15 into 32-bit dwords
    _mm_storeu_si128((__m128i*) (pDst + 12), qrtr); //- Write to memory

    ...
}
```

Converting ASCII Character Runs – SSE Example



00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	zero
72	00	64	00	20	00	CE	00	BA	00	E1	00	BD	00	B9	00	half
BA	00	00	00	E1	00	00	00	BD	00	00	00	B9	00	00	00	qrtr
BA	00	00	00	E1	00	00	00	BD	00	00	00	B9	00	00	00	*memory*

```
_mm_storeu_si128((__m128i*) pDst + 12, qrtr)
```

pDst + 12

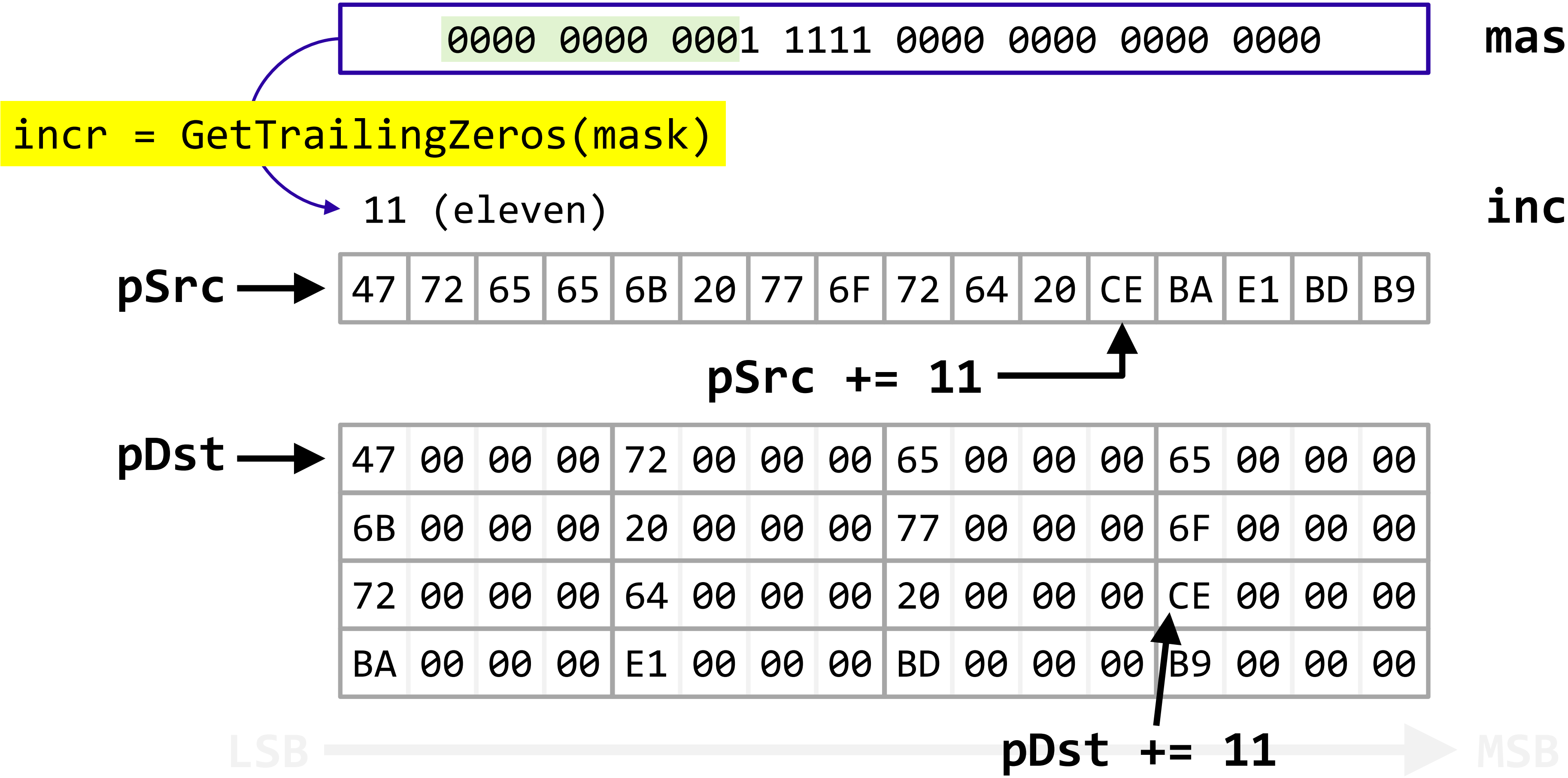
Converting ASCII Character Runs – SSE Code

```
KEWB_FORCE_INLINE void
UtfUtils::ConvertAsciiWithSse(char8_t const*& pSrc, char32_t*& pDst) noexcept
{
    ...
    //- If no bits were set in the mask, then all 16 code units were ASCII.
    //
    if (mask == 0)
    {
        pSrc += 16;
        pDst += 16;
    }

    //- Otherwise, the number of trailing (low-order) zero bits in the mask is
    //  the number of ASCII code units.
    //
    else
    {
        incr  = GetTrailingZeros(mask);
        pSrc += incr;
        pDst += incr;
    }
}
```

Finding the Trailing Zero-Bit Count

```
#if defined KEWB_PLATFORM_LINUX  && (defined KEWB_COMPILER_CLANG  ||  defined KEWB_COMPILER_GCC)

    KEWB_FORCE_INLINE int32_t
    UtfUtils::GetTrailingZeros(int32_t x) noexcept
    {
        return  __builtin_ctz((unsigned int) x);
    }

#elif defined KEWB_PLATFORM_WINDOWS  &&  defined KEWB_COMPILER_MSVC

    KEWB_FORCE_INLINE int32_t
    UtfUtils::GetTrailingZeros(int32_t x) noexcept
    {
        unsigned long   indx;
        _BitScanForward(&indx, (unsigned long) x);
        return (int32_t) indx;
    }

#endif
```

Converting ASCII Character Runs – SSE Example



```
0000 0000 0001 1111 0000 0000 0000 0000
```

mask

`incr = GetTrailingZeros(mask)`

`11 (eleven)`

incr

pSrc

47	72	65	65	6B	20	77	6F	72	64	20	CE	BA	E1	BD	B9

pSrc += 11

pDst

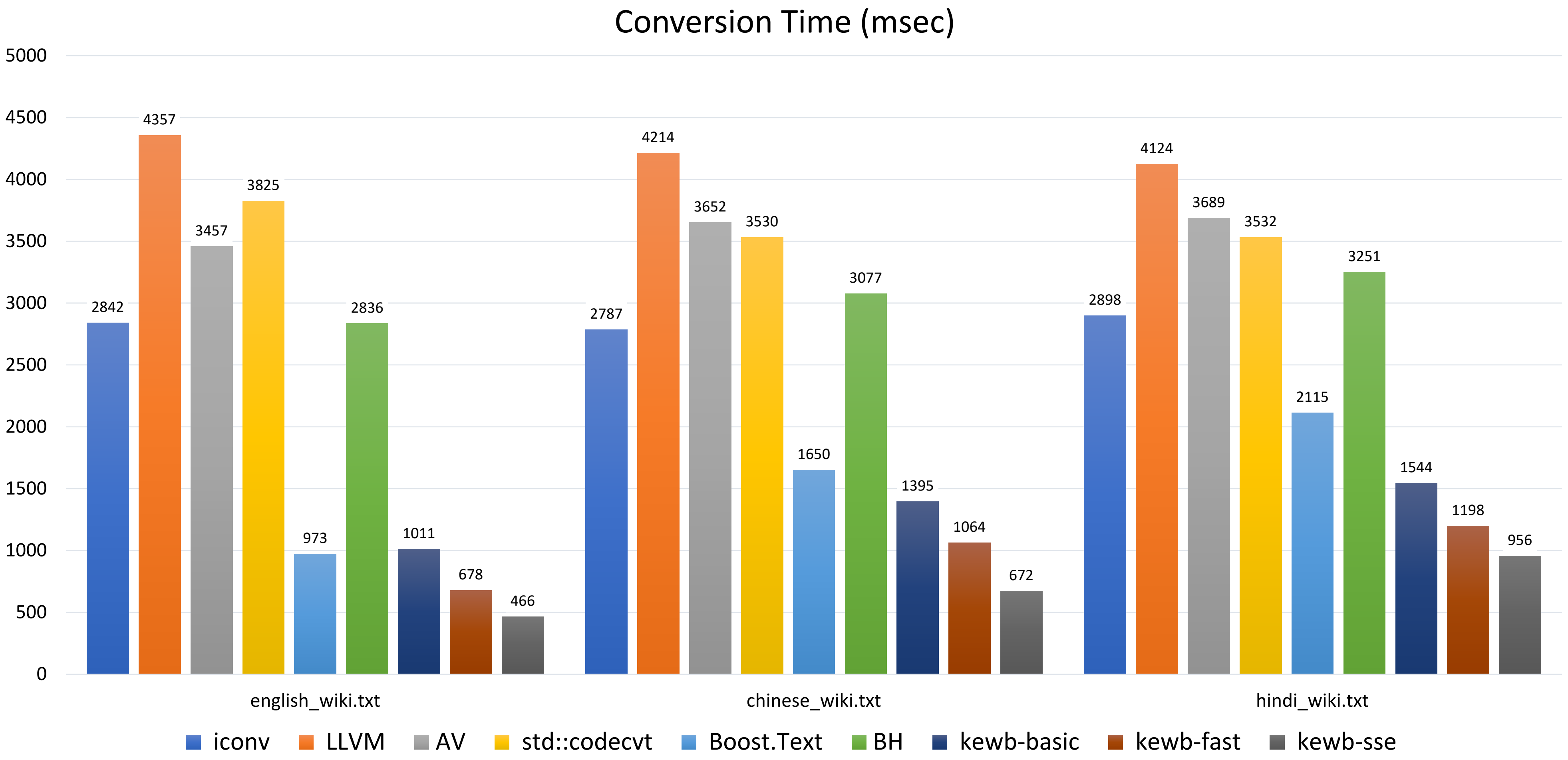
47	00	00	00	72	00	00	00	65	00	00	00	65	00	00	00
6B	00	00	00	20	00	00	00	77	00	00	00	6F	00	00	00
72	00	00	00	64	00	00	00	20	00	00	00	CE	00	00	00
BA	00	00	00	E1	00	00	00	BD	00	00	00	B9	00	00	00

pDst += 11

The SSE-Optimized Conversion Algorithm (UTF-8 to UTF-32)

```
KEWB_ALIGN_FN std::ptrdiff_t
UtfUtils::SseConvert(char8_t const* pSrc, char8_t const* pSrcEnd, char32_t* pDst) noexcept
{
    char32_t*   pDstOrig = pDst;
    char32_t    cdpt;

    while (pSrc < (pSrcEnd - sizeof(__m128i)))
    {
        if (*pSrc < 0x80)
        {
            ConvertAsciiWithSse(pSrc, pDst);
        }
        else
        {
            if (Advance(pSrc, pSrcEnd, cdpt) != ERR)
            {
                *pDst++ = cdpt;
            }
            else
            {
                return -1;
            }
        }
    }
    ...
}
```

SSE-Optimized Conversion Performance Overview (Linux/GCC)



Testing and Benchmarks

Testing Methodology – Platforms

- Ubuntu 18.04 VM on Windows 10 / Core i7 3740 / 2.7 GHz / 16GB RAM
  - **GCC 7.2**, all code compiled with `-O3 -march=westmere`
  - **Clang 5.0.1**, all code compiled with `-O3 -march=westmere`

- Windows 10 / Core i7 / 2.7 GHz / 16GB RAM
  - **Visual Studio 15.4.4**, all code compiled with `/O2 /Ob2 /Oi /Ot`

Testing Methodology – Input Data

- Nine input files
  - `english_wiki.txt`
  - `chinese_wiki.txt`
  - `hindi_wiki.txt`
  - `portuguese_wiki.txt`
  - `russian_wiki.txt`
  - `swedish_wiki.txt`

  } Taken directly from wikipedia.org

  - `stress_test_0.txt` – 100K ASCII code points (100K code units)
  - `stress_test_1.txt` – 100K Chinese code points (300K code units)
  - `stress_test_2.txt` – 50K Chinese code points interleaved with 50K ASCII code points (200K code units)

Testing Methodology – Reference Libraries

- iconv – GNU libiconv, used here as the “gold standard”
- LLVM – UTF conversion functions from the LLVM distribution
- AV – UTF-8 to UTF-32 conversion by Alexey Vatchenko
- `std::codecvt` – Standard library’s UTF conversion
- BH – Alternative DFA-based conversion by Bjoern Hoehrmann
- Boost.Text – Iterator-based interface to UTF conversion by Zach Laine
- win32-mbtowc – `MultiByteToWideChar()` from Win32 API

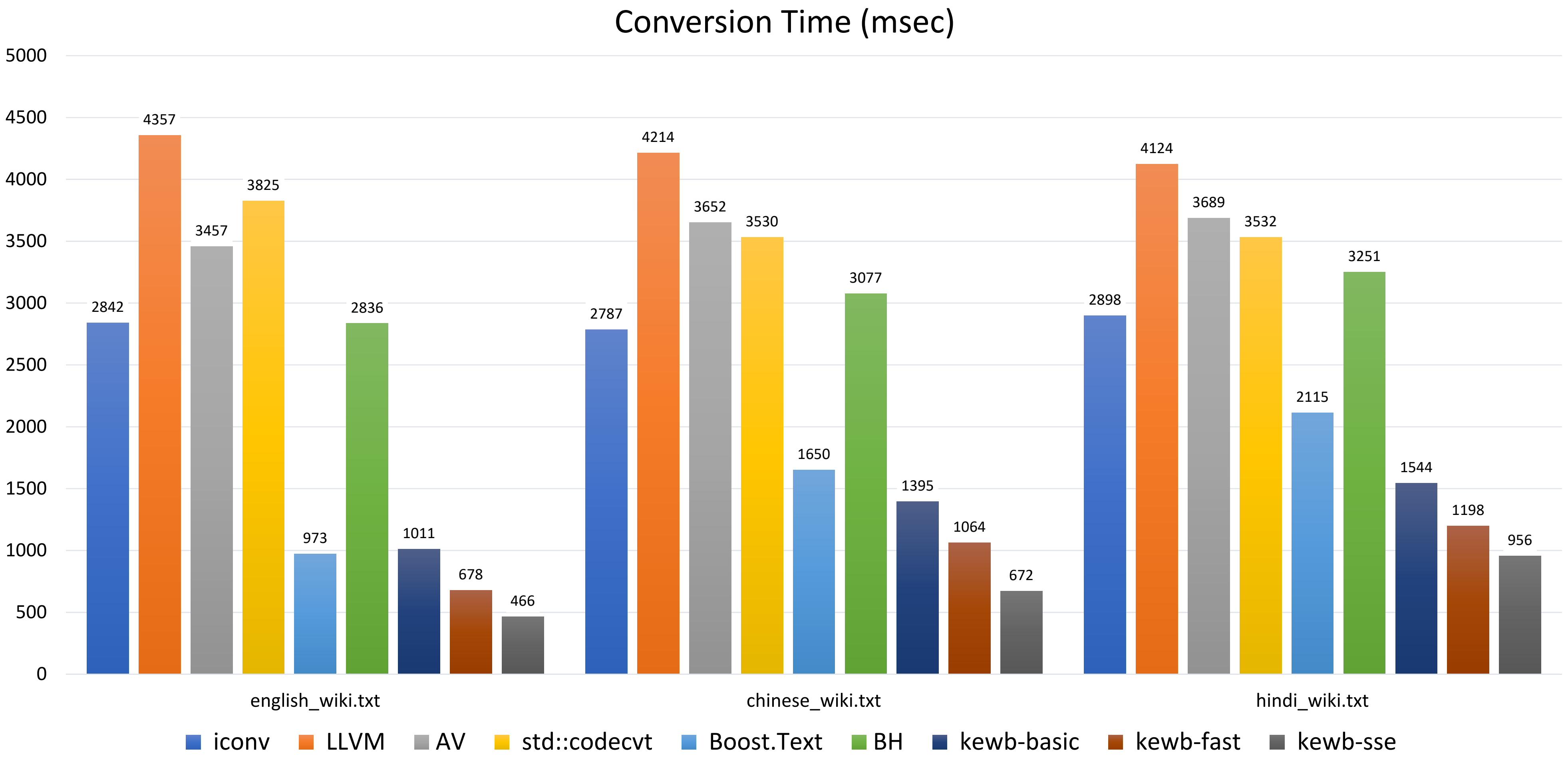
Testing Methodology - Timings

- Timings for each file were obtained by:
  1. Reading the input file
  2. Creating an oversized output buffer
  3. Starting the timer
  4. Entering the timing loop
  5. Performing conversion of the input buffer multiple times
     - The number of repetitions was such that 1GB of input text was processed
  6. Exiting the timing loop
  7. Stopping the timer
  8. Collecting and collating results
- To pass, a library’s result had to agree with `iconv()`

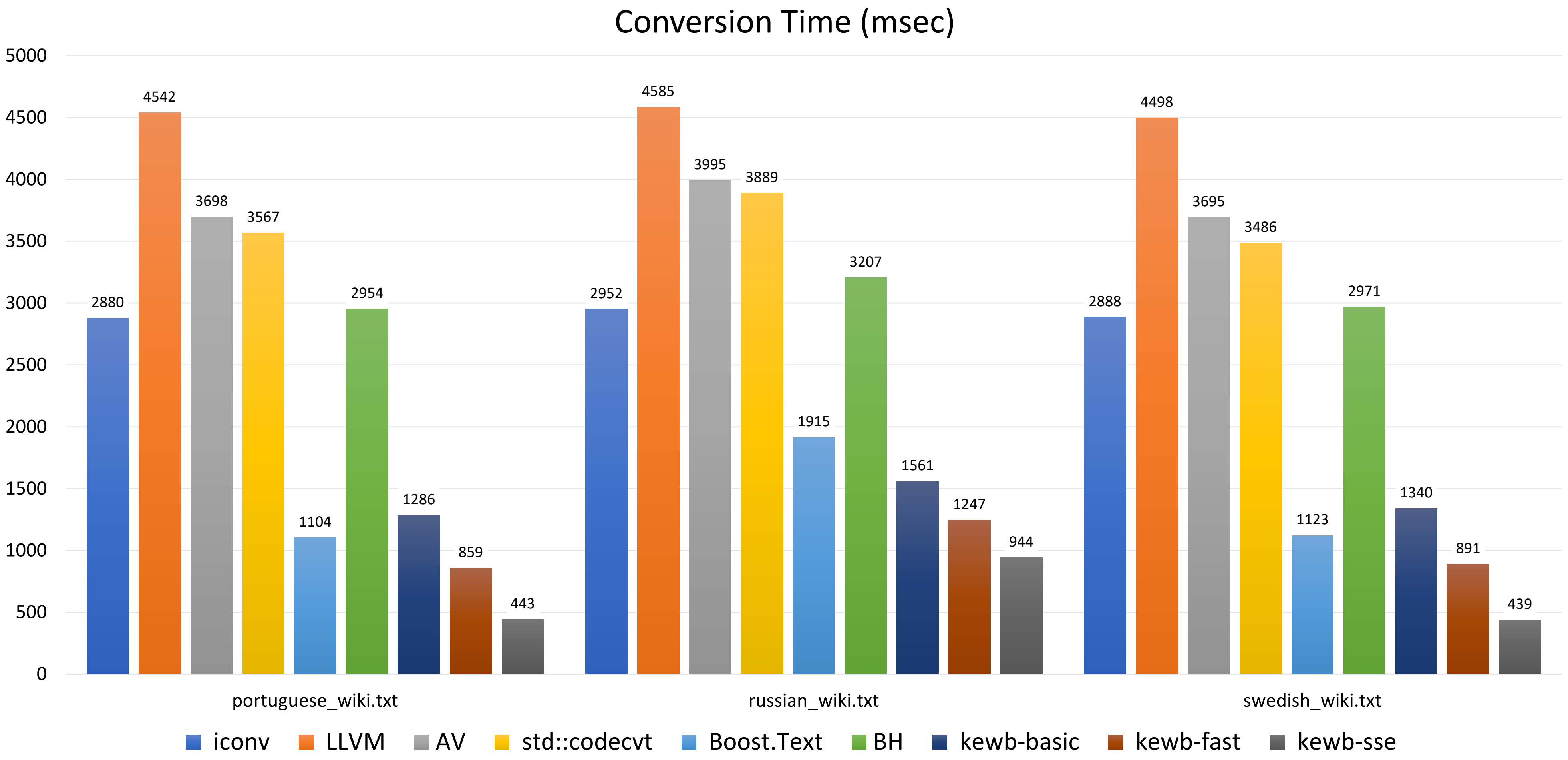
Benchmark Results

GCC 7.2 – Ubuntu 18.04 VM – Core i7

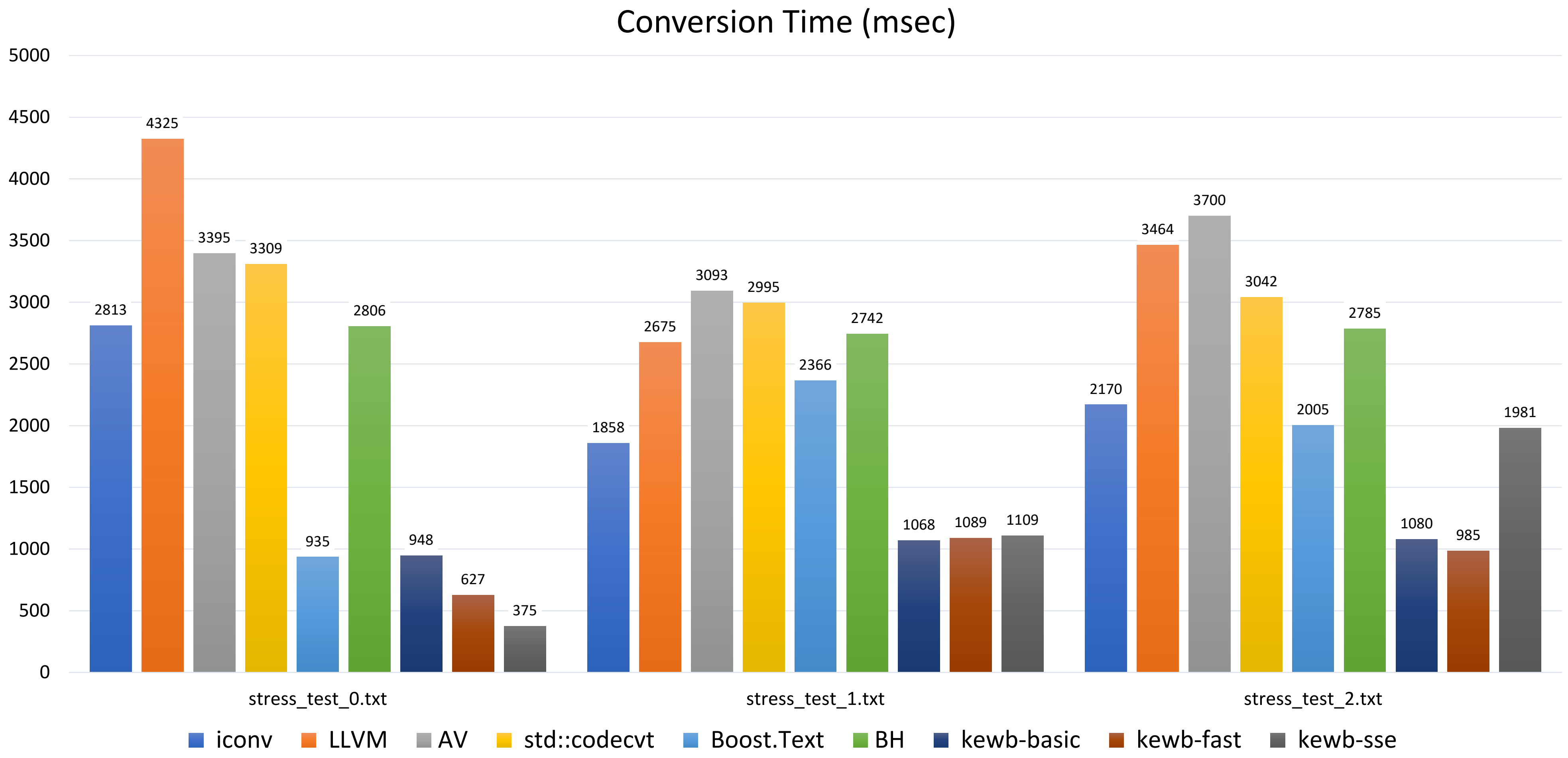
GCC 7.2 – Ubuntu 18.04 VM – Core i7 – UTF-32



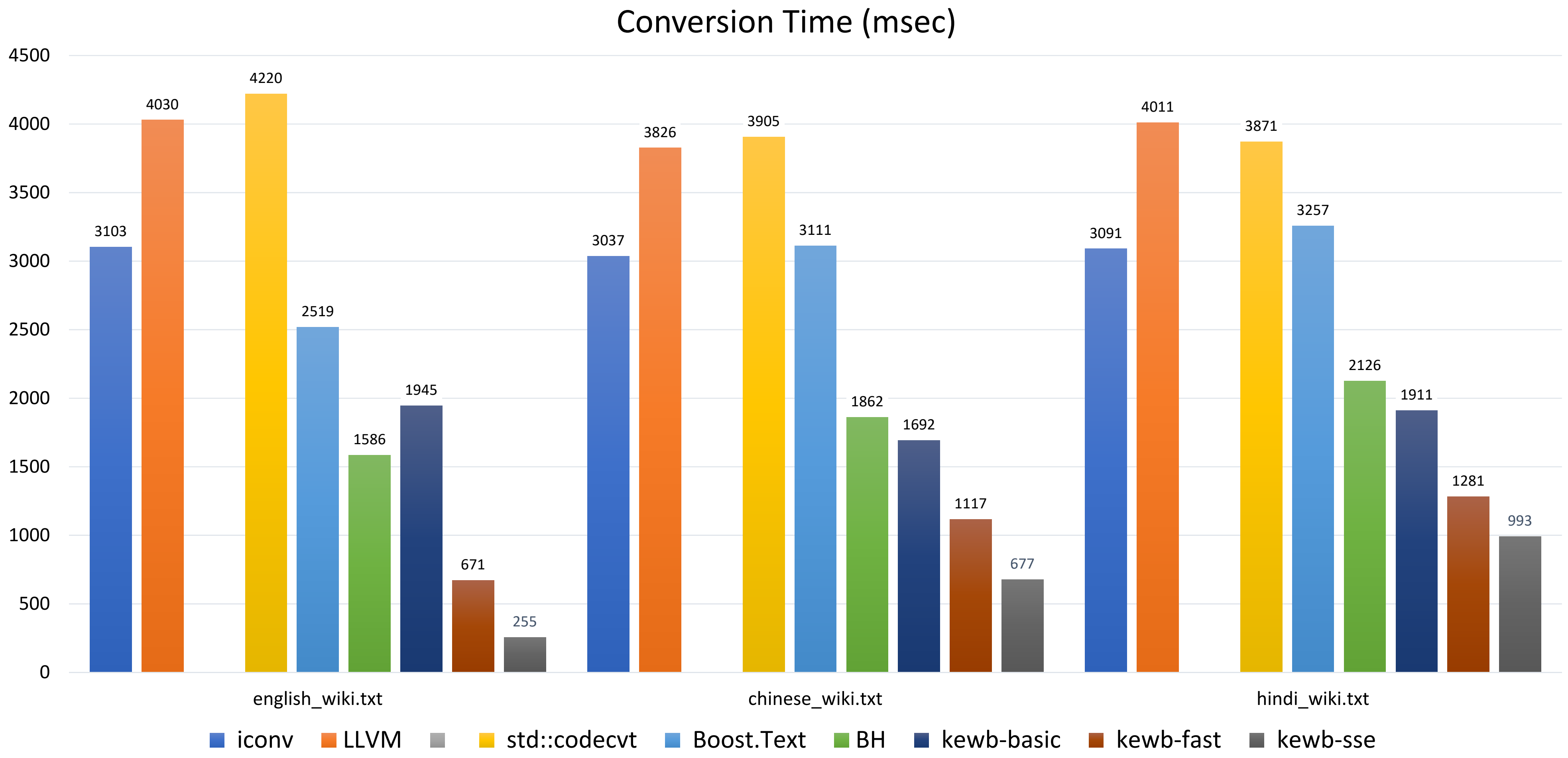
GCC 7.2 – Ubuntu 18.04 VM – Core i7 – UTF-32



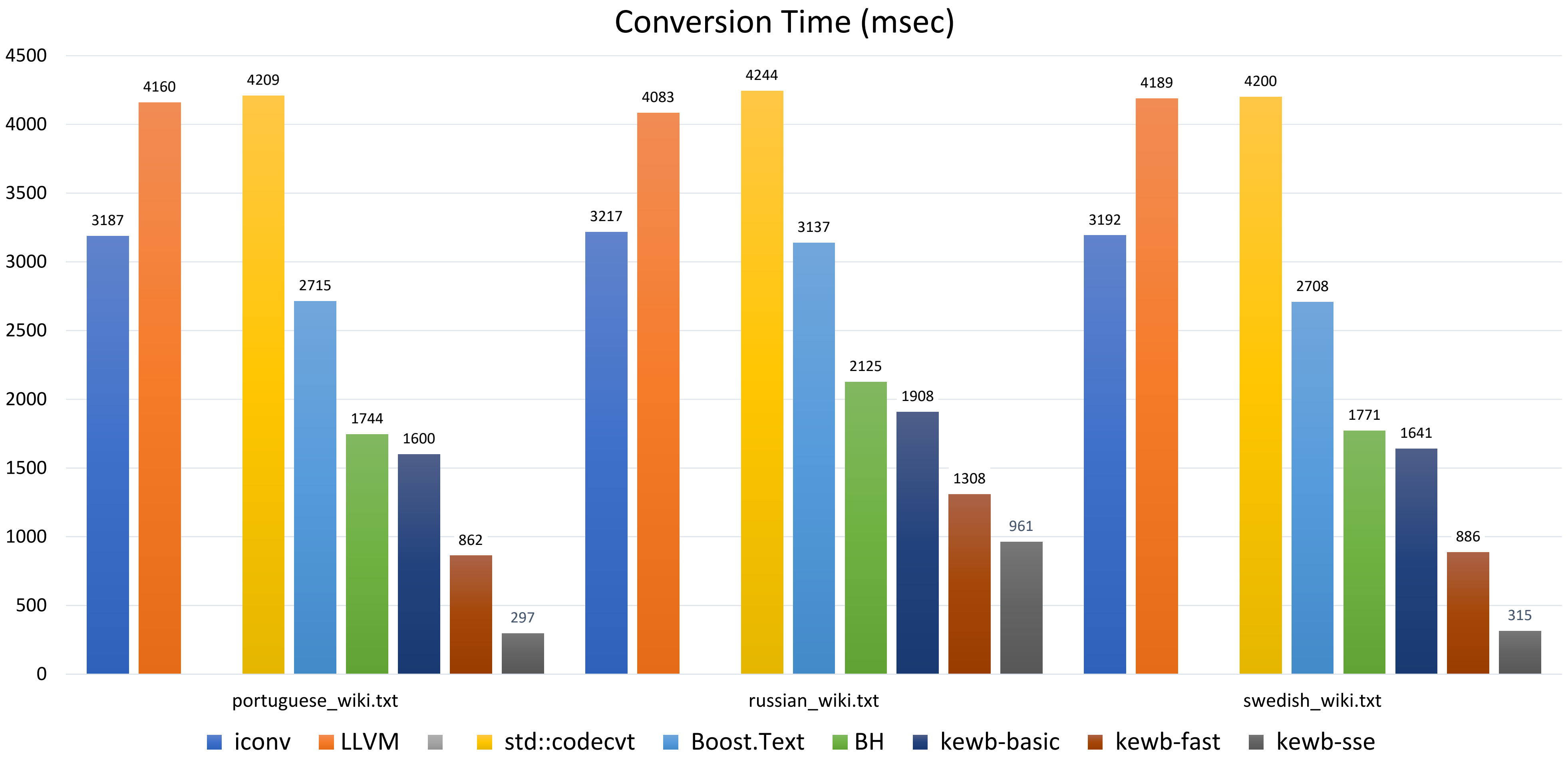
GCC 7.2 – Ubuntu 18.04 VM – Core i7 – UTF-32



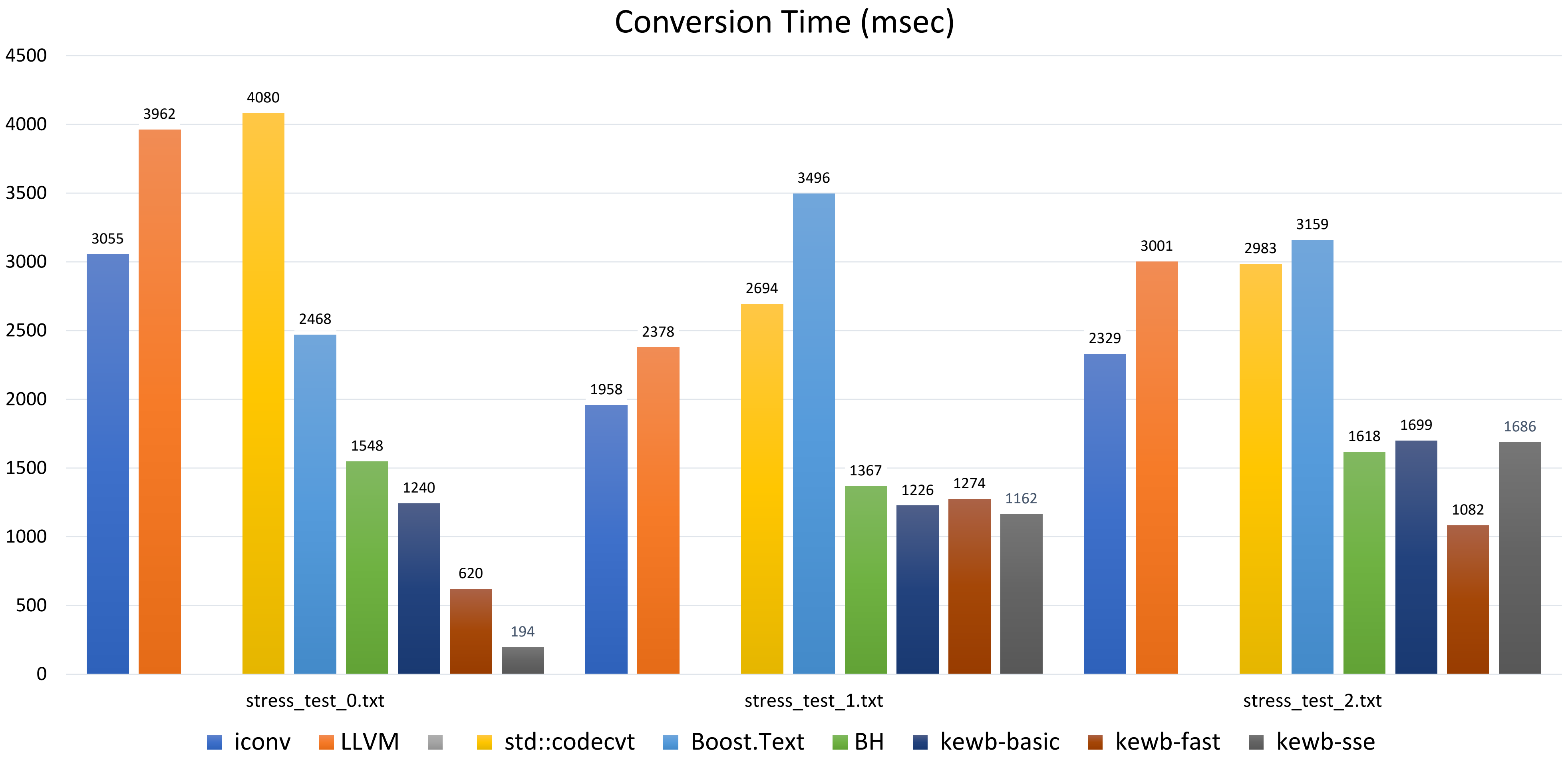
GCC 7.2 – Ubuntu 18.04 VM – Core i7 – UTF-16



GCC 7.2 – Ubuntu 18.04 VM – Core i7 – UTF-16



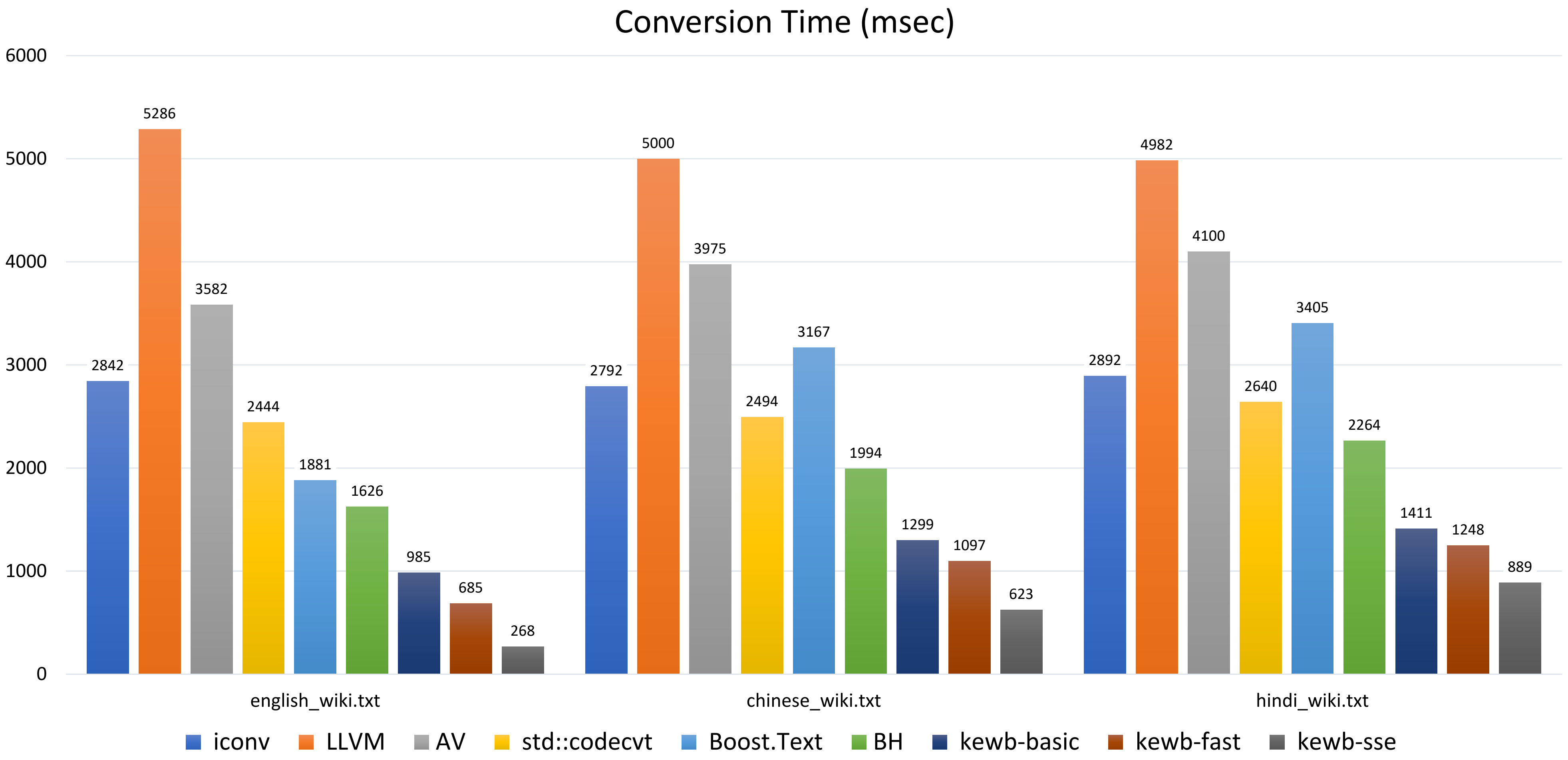
GCC 7.2 – Ubuntu 18.04 VM – Core i7 – UTF-16



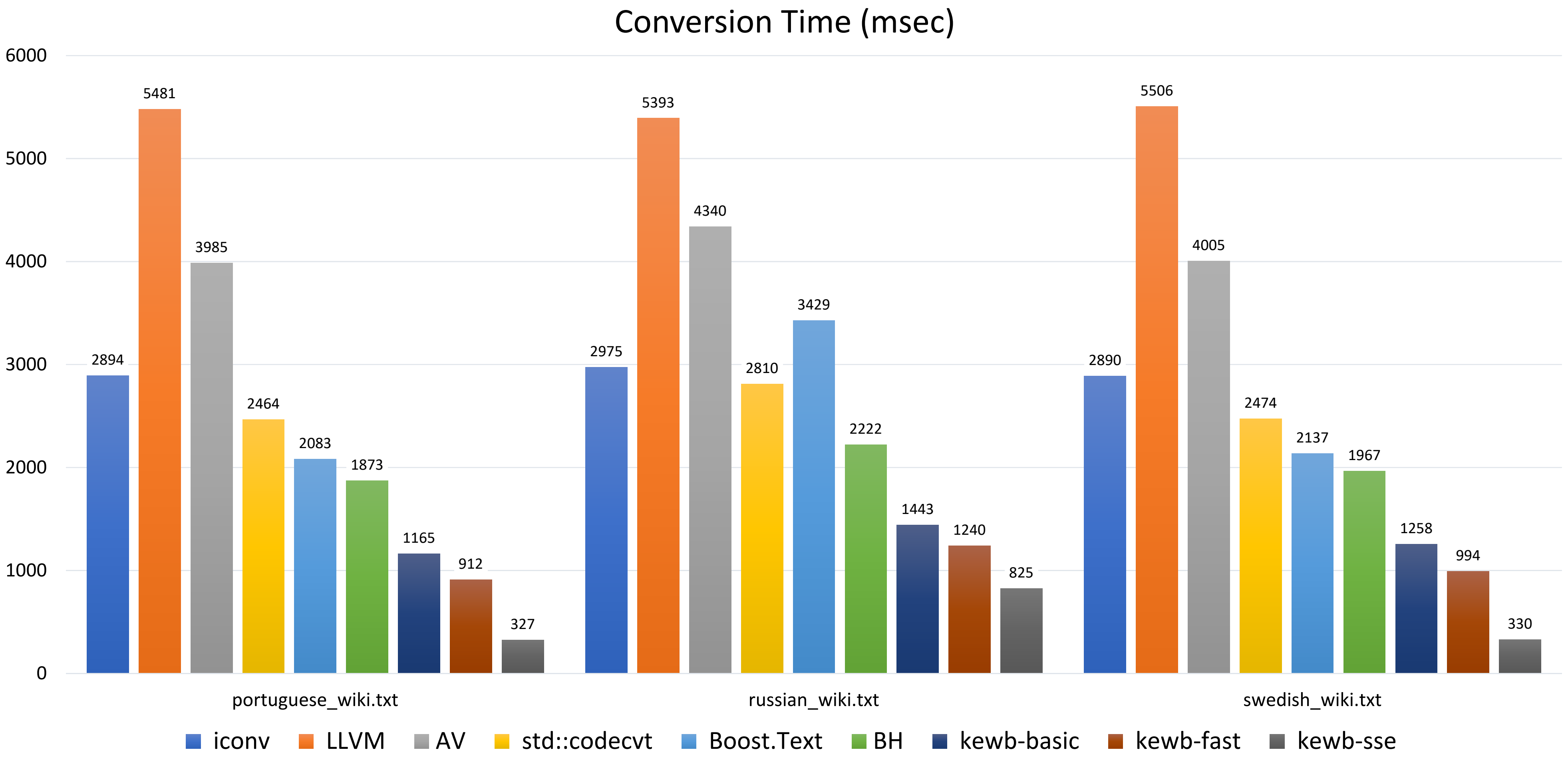
Benchmark Results

Clang 5.0.1 – Ubuntu 18.04 VM – Core i7

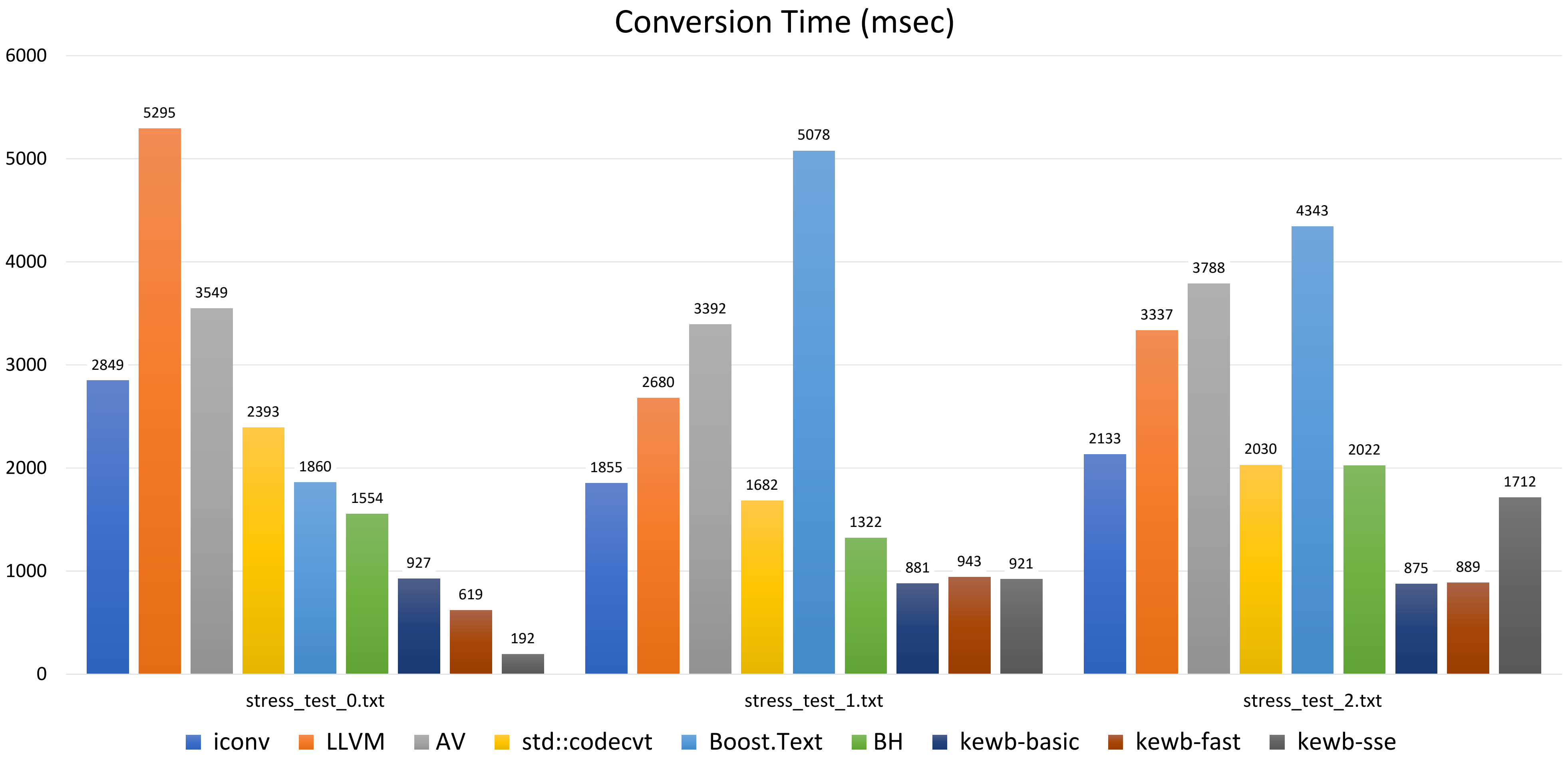
Clang 5.0.1 – Ubuntu 18.04 VM – Core i7 – UTF-32



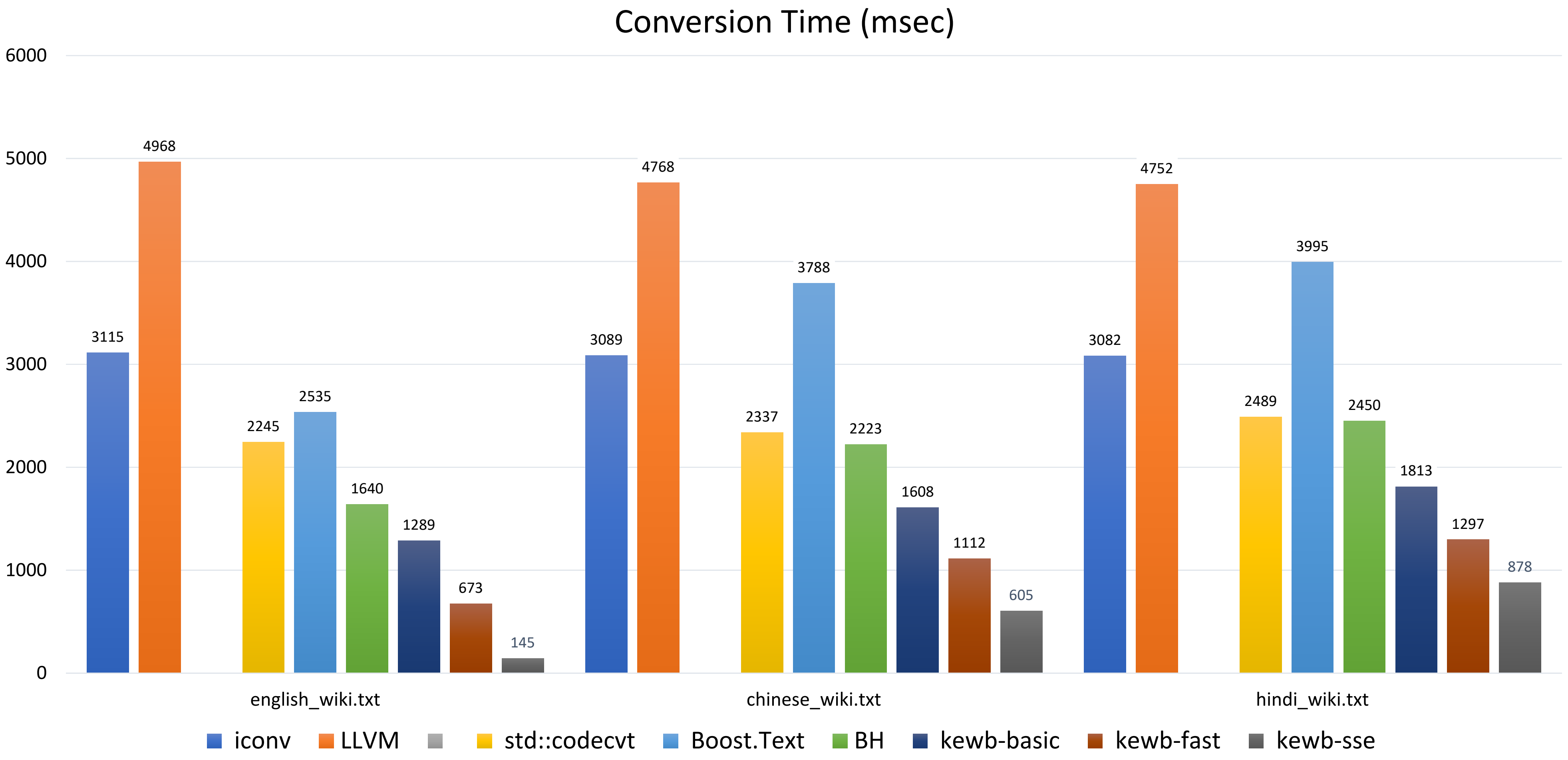
Clang 5.0.1 – Ubuntu 18.04 VM – Core i7 – UTF-32



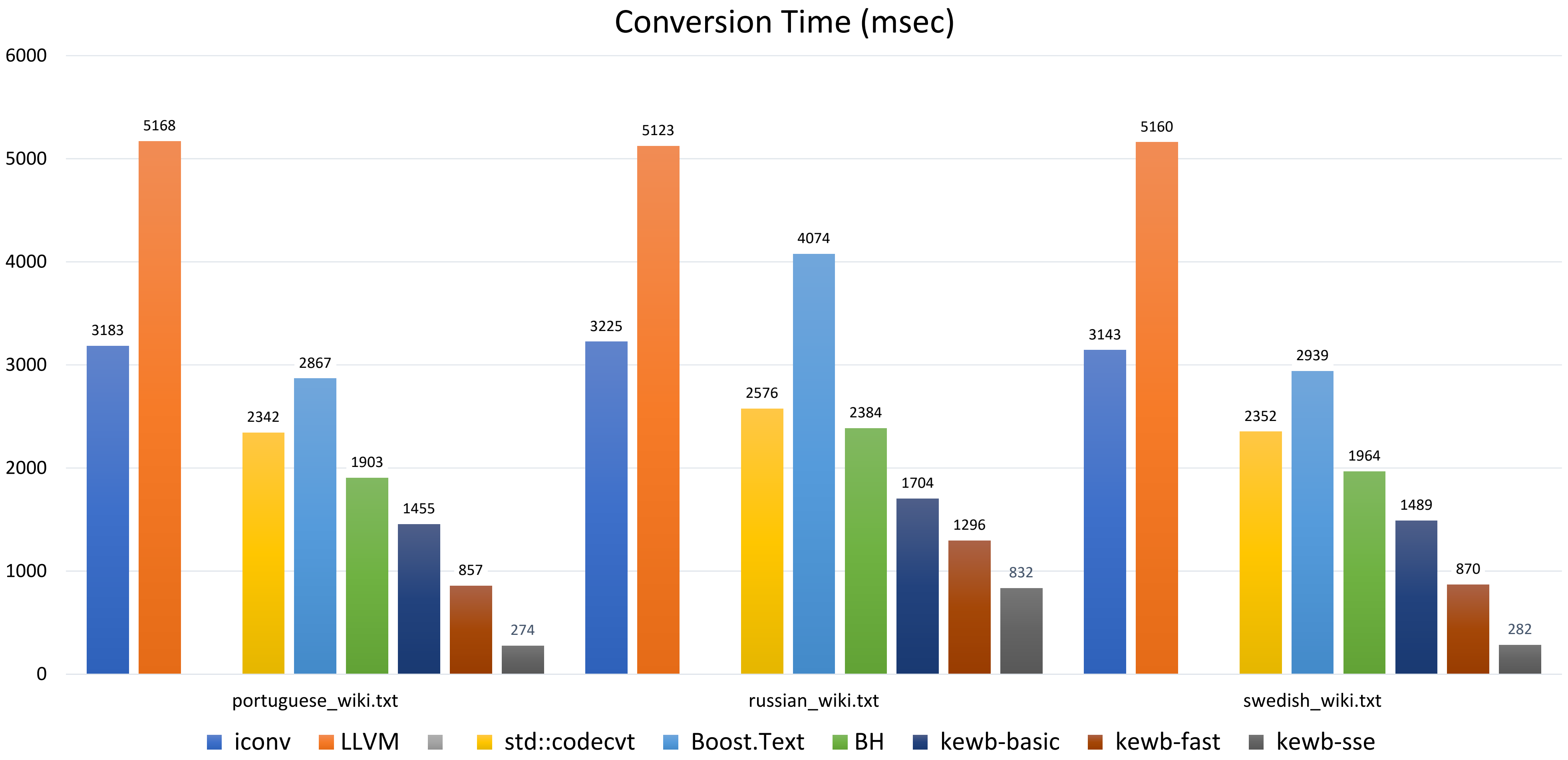
Clang 5.0.1 – Ubuntu 18.04 VM – Core i7 – UTF-32



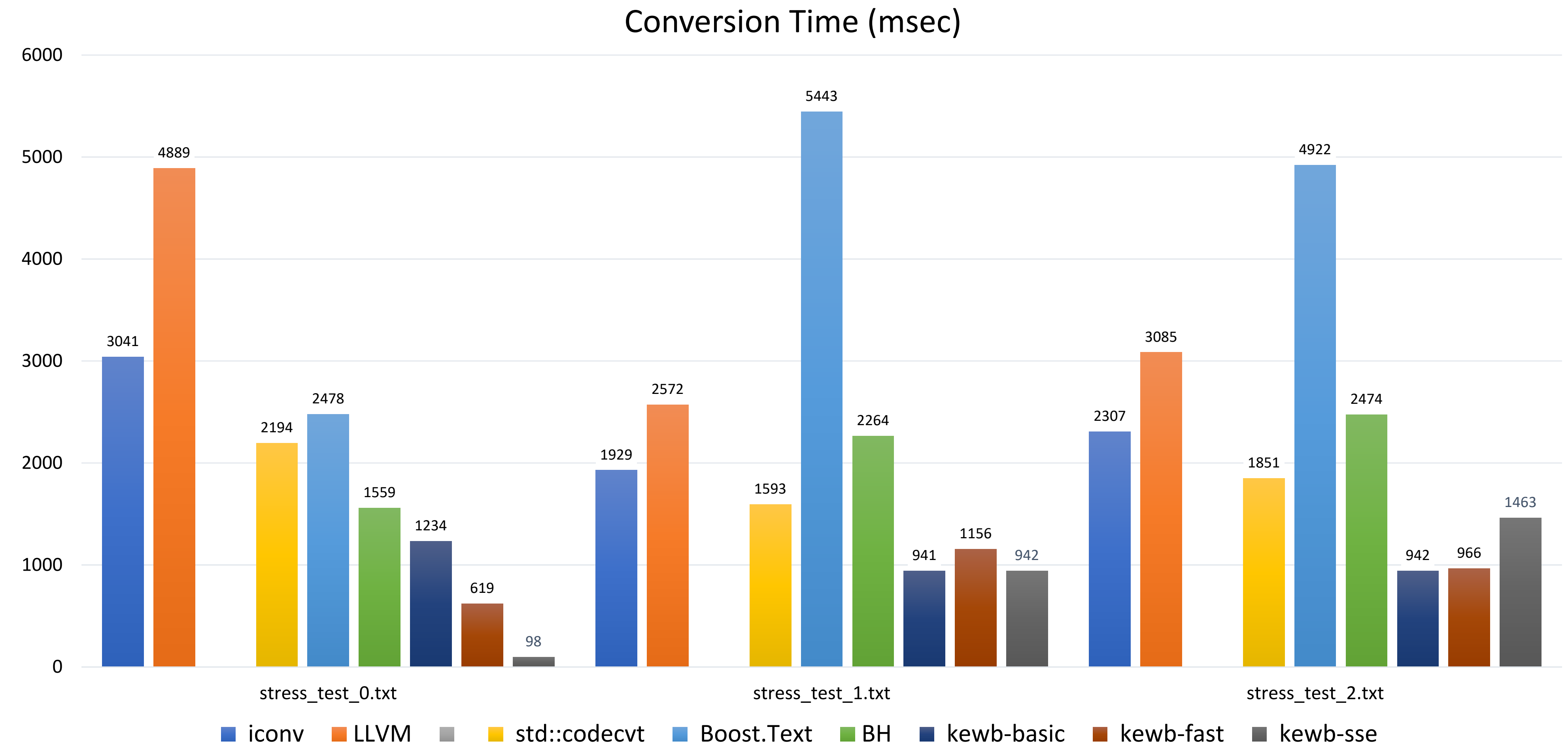
Clang 5.0.1 – Ubuntu 18.04 VM – Core i7 – UTF-16



Clang 5.0.1 – Ubuntu 18.04 VM – Core i7 – UTF-16



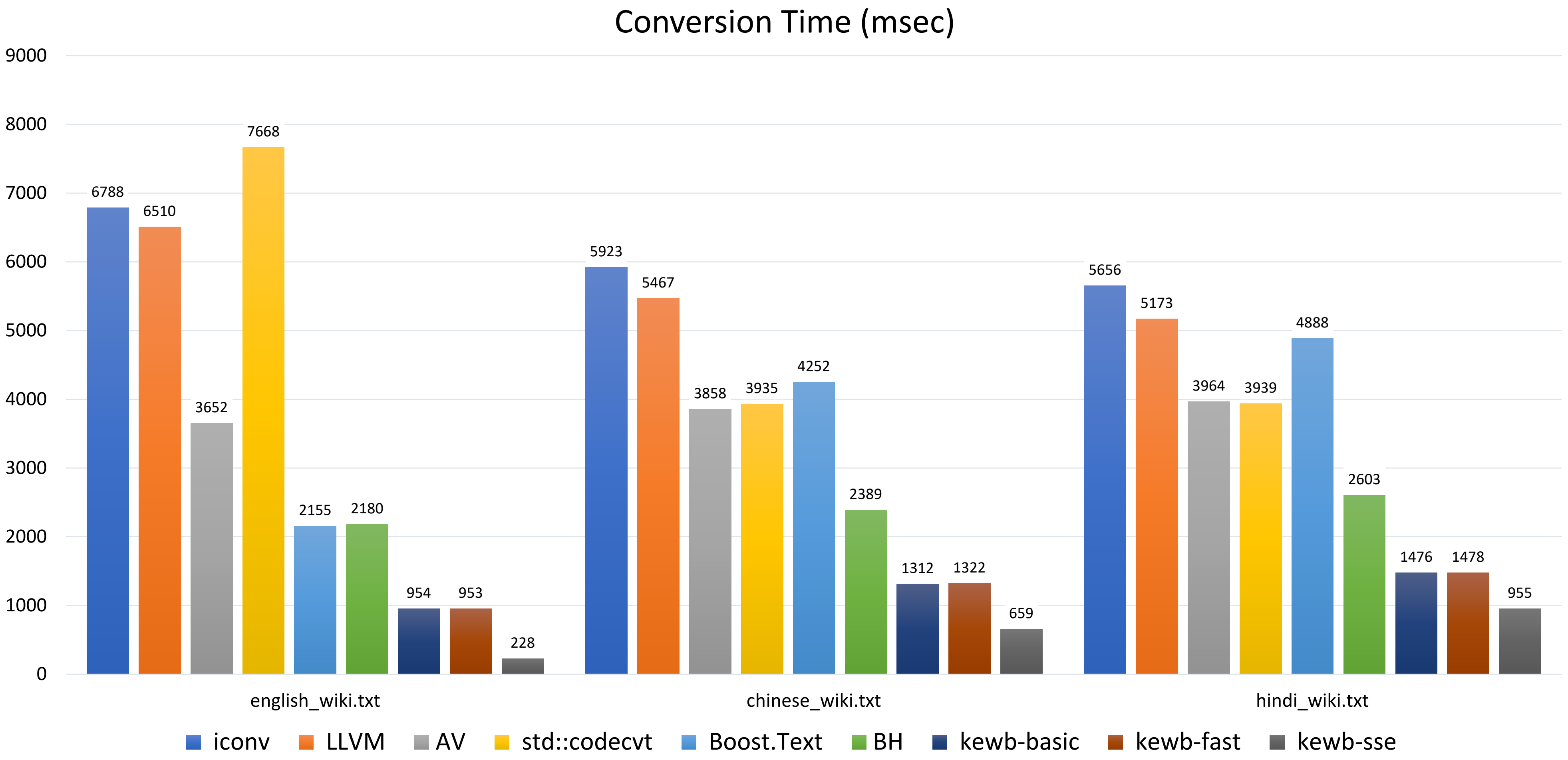
Clang 5.0.1 – Ubuntu 18.04 VM – Core i7 – UTF-16



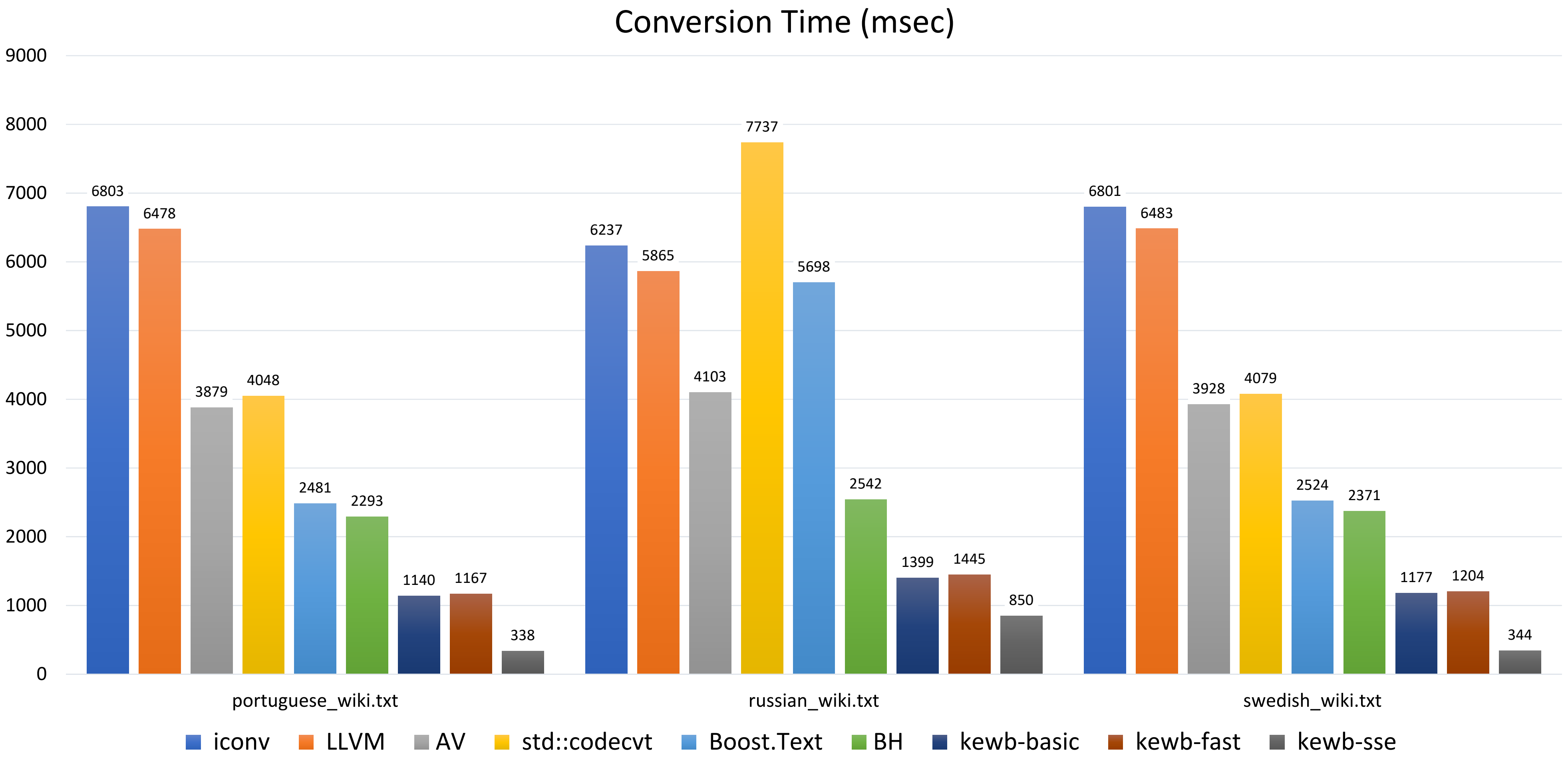
Benchmark Results

VS 2017 – Windows 10 – Core i7

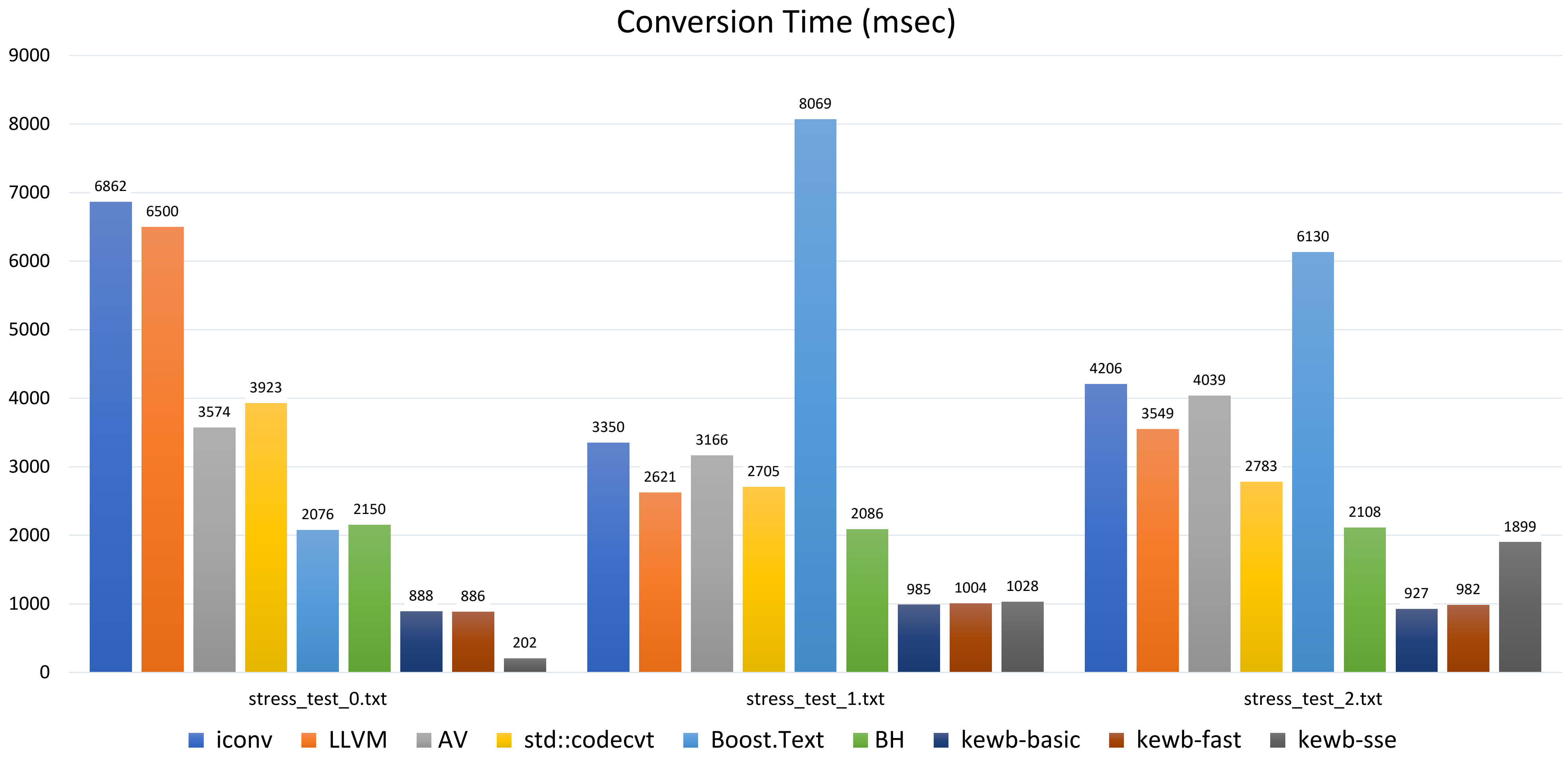
VS 2017 – Windows 10 – Core i7 – UTF-32



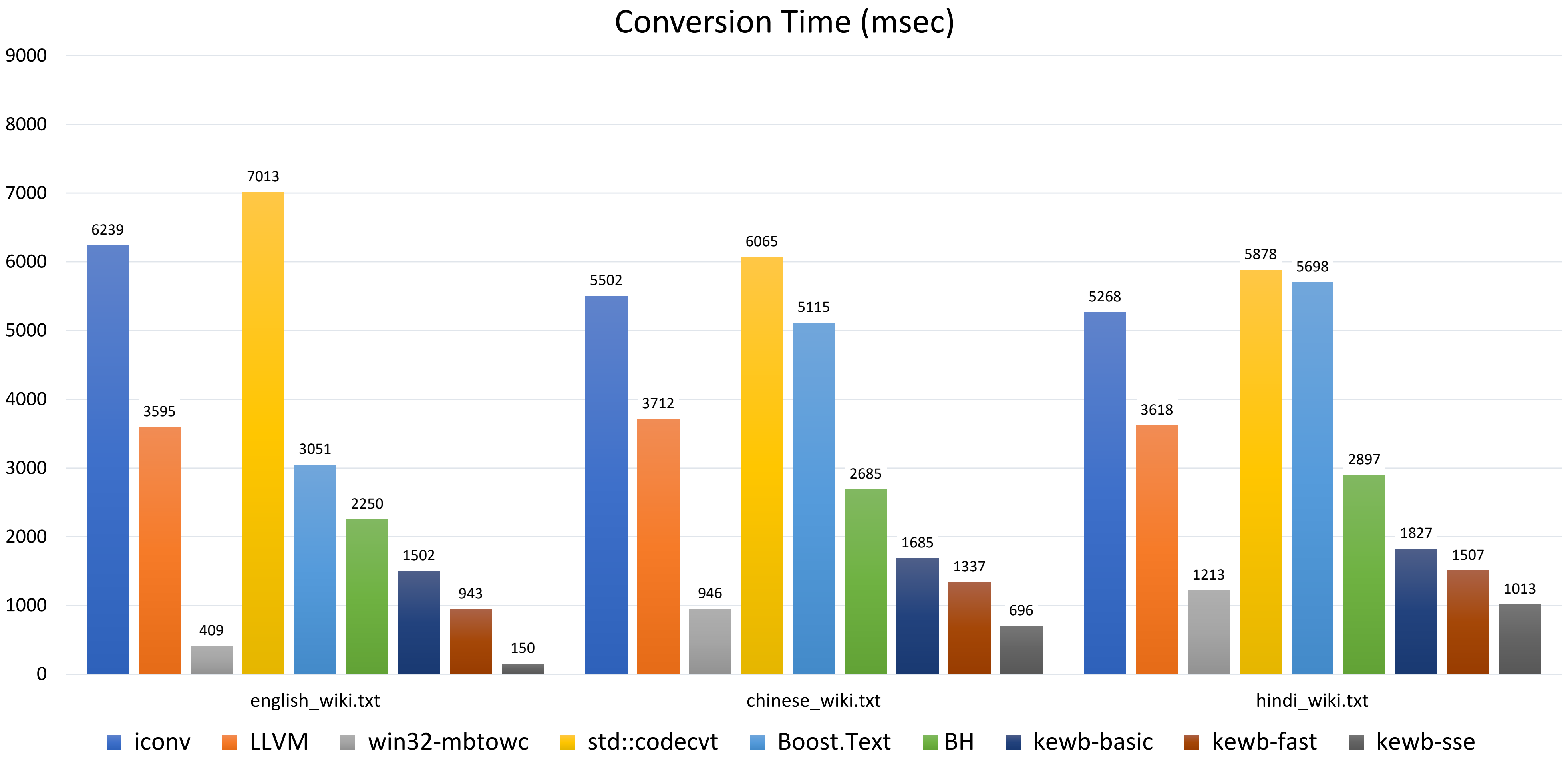
VS 2017 – Windows 10 – Core i7 – UTF-32



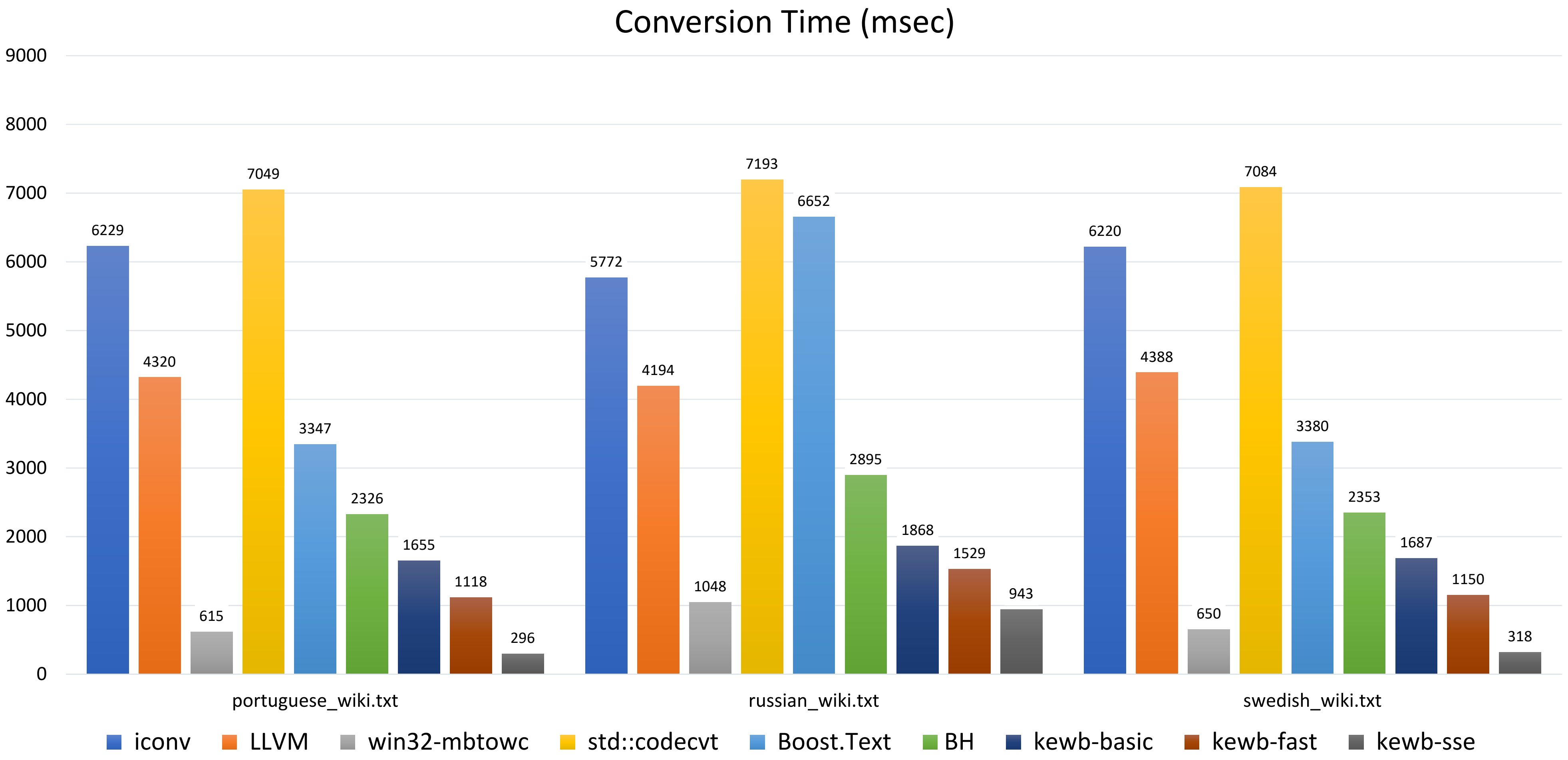
VS 2017 – Windows 10 – Core i7 – UTF-32



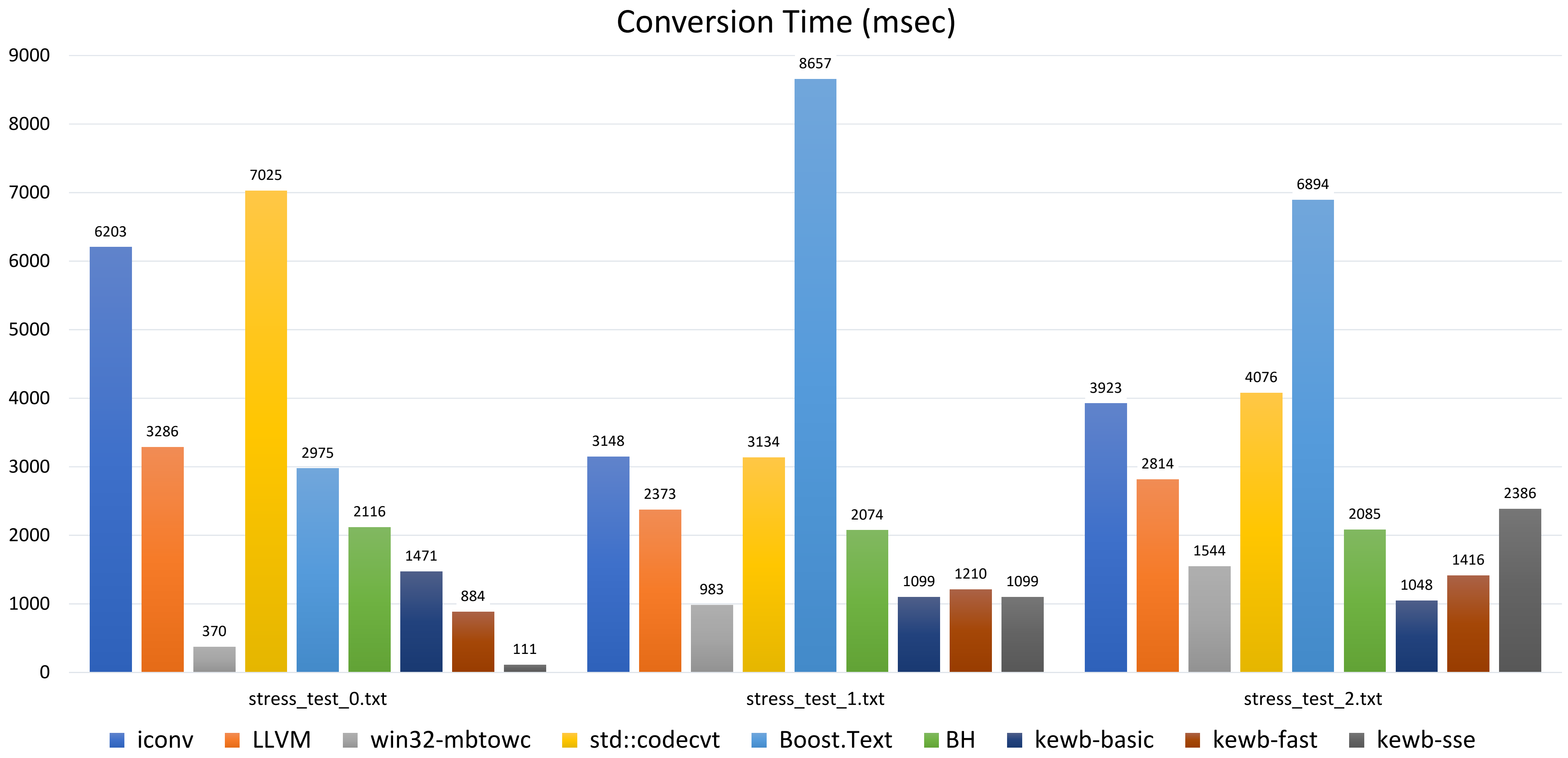
VS 2017 – Windows 10 – Core i7 – UTF-16



VS 2017 – Windows 10 – Core i7 – UTF-16



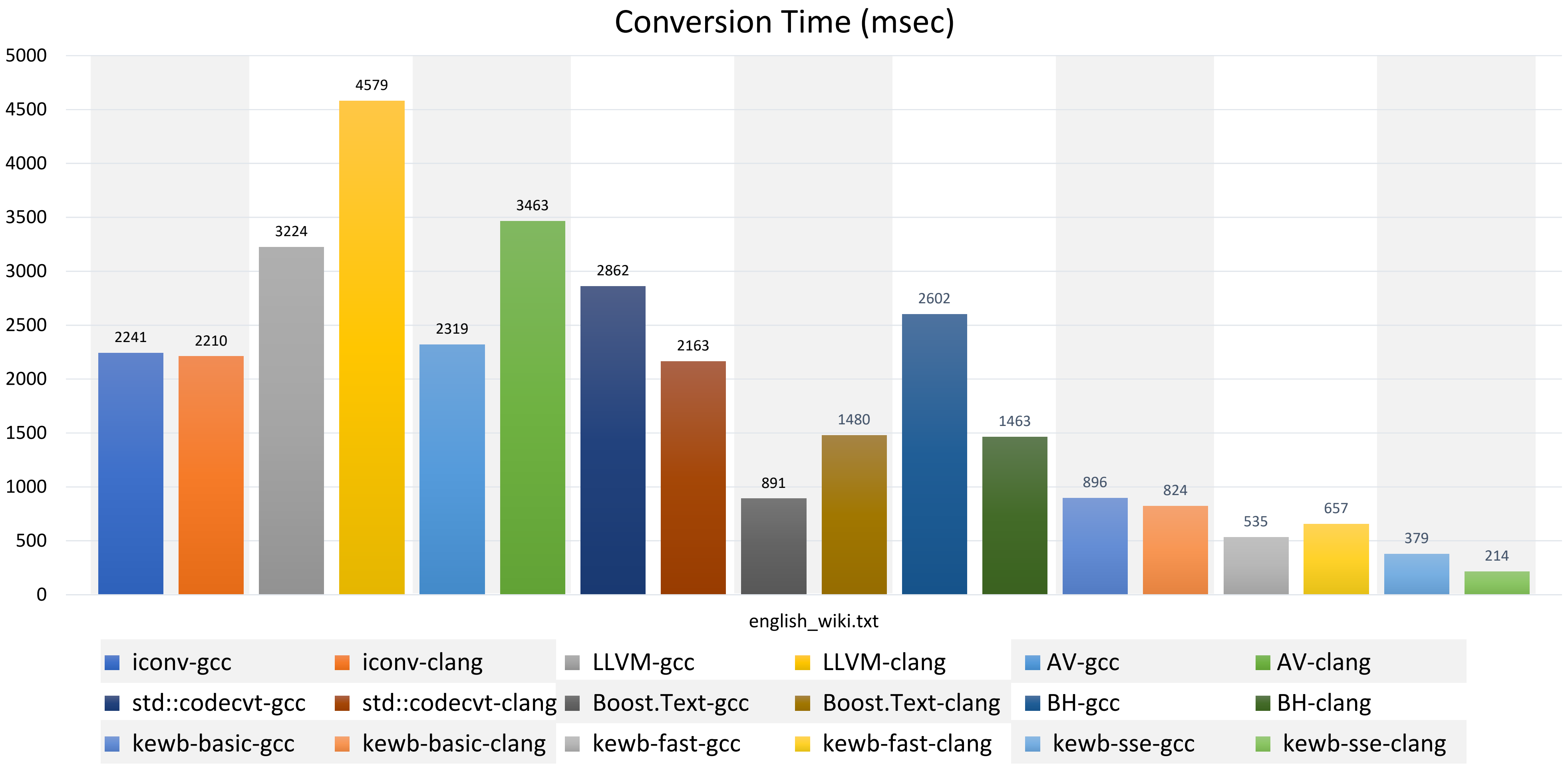
VS 2017 – Windows 10 – Core i7 – UTF-16



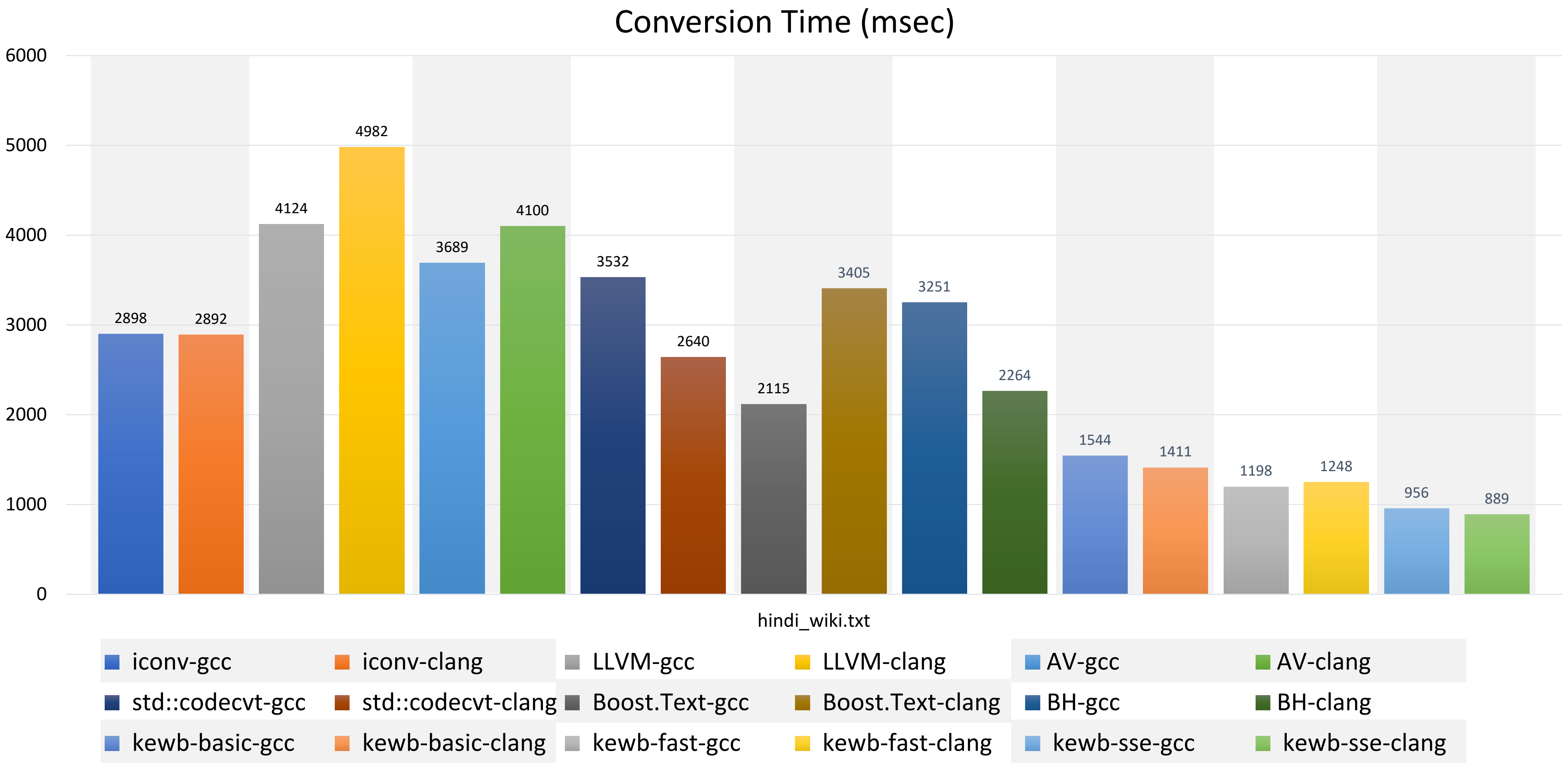
Benchmark Results

GCC 7.2/Clang 5.0.1 – Ubuntu 18.04 VM – Core i7

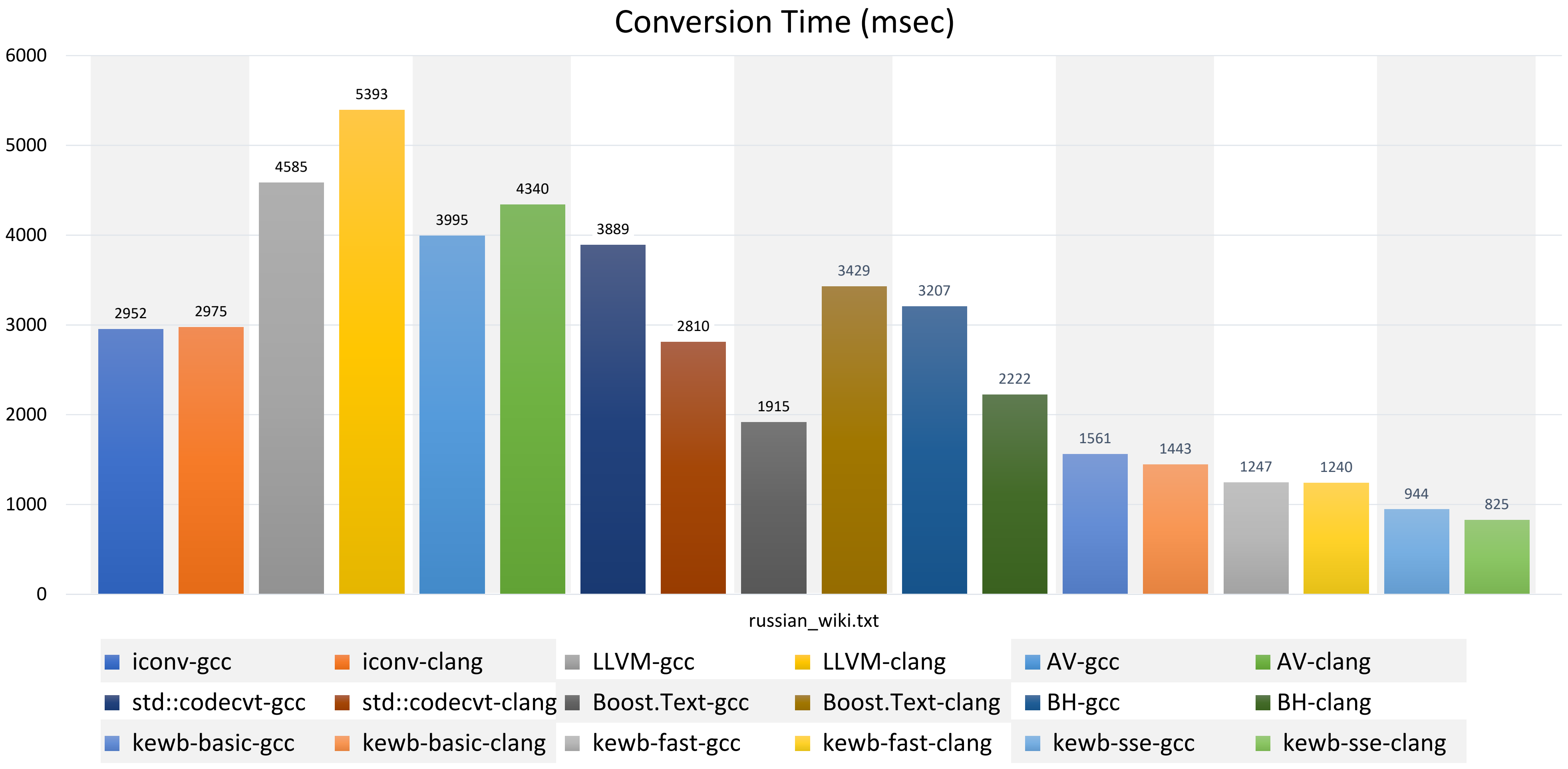
GCC 7.2/Clang 5.0.1 – Ubuntu 18.04 VM – Core i7 – UTF-32



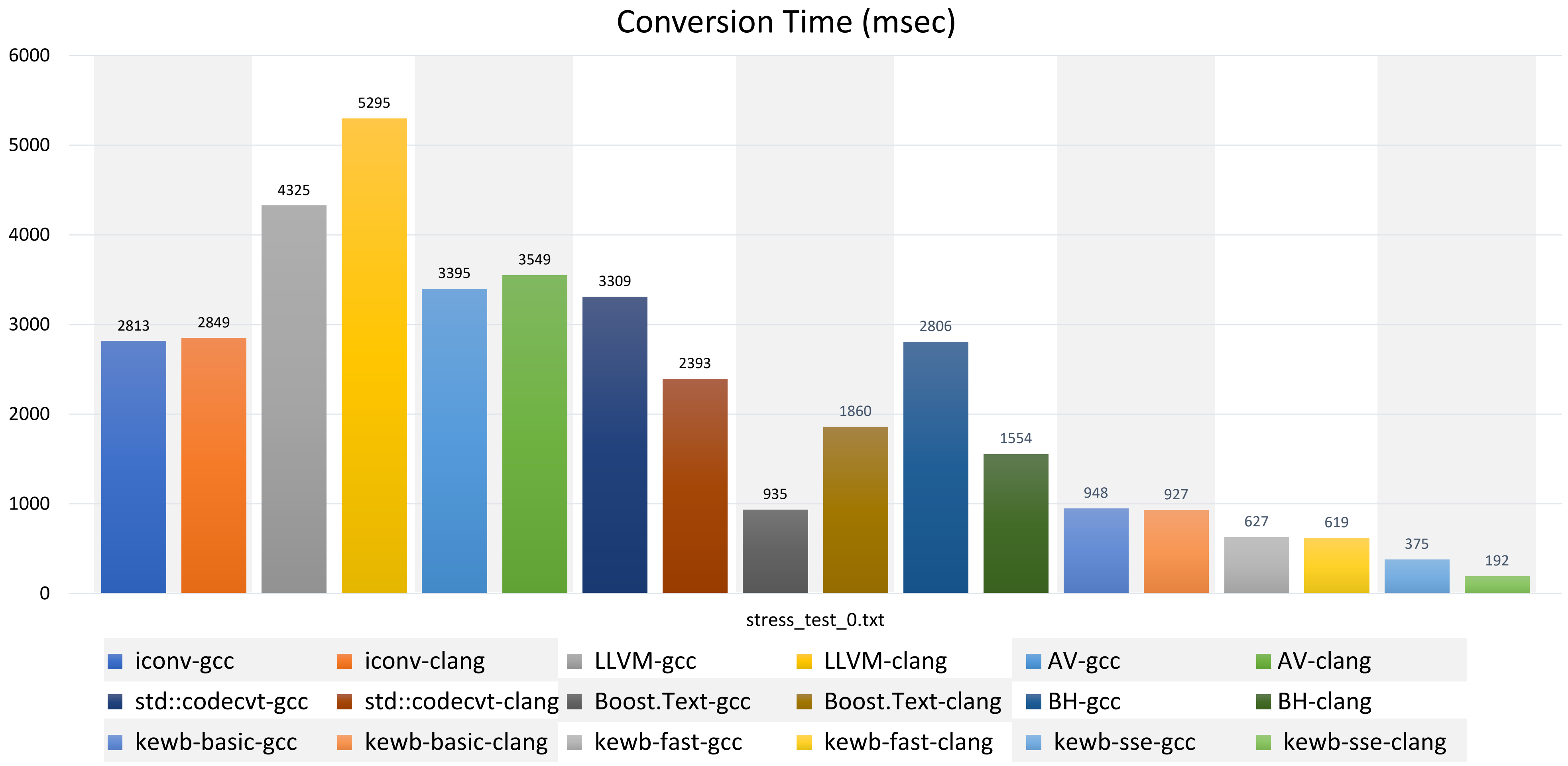
GCC 7.2/Clang 5.0.1 – Ubuntu 18.04 VM – Core i7 – UTF-32



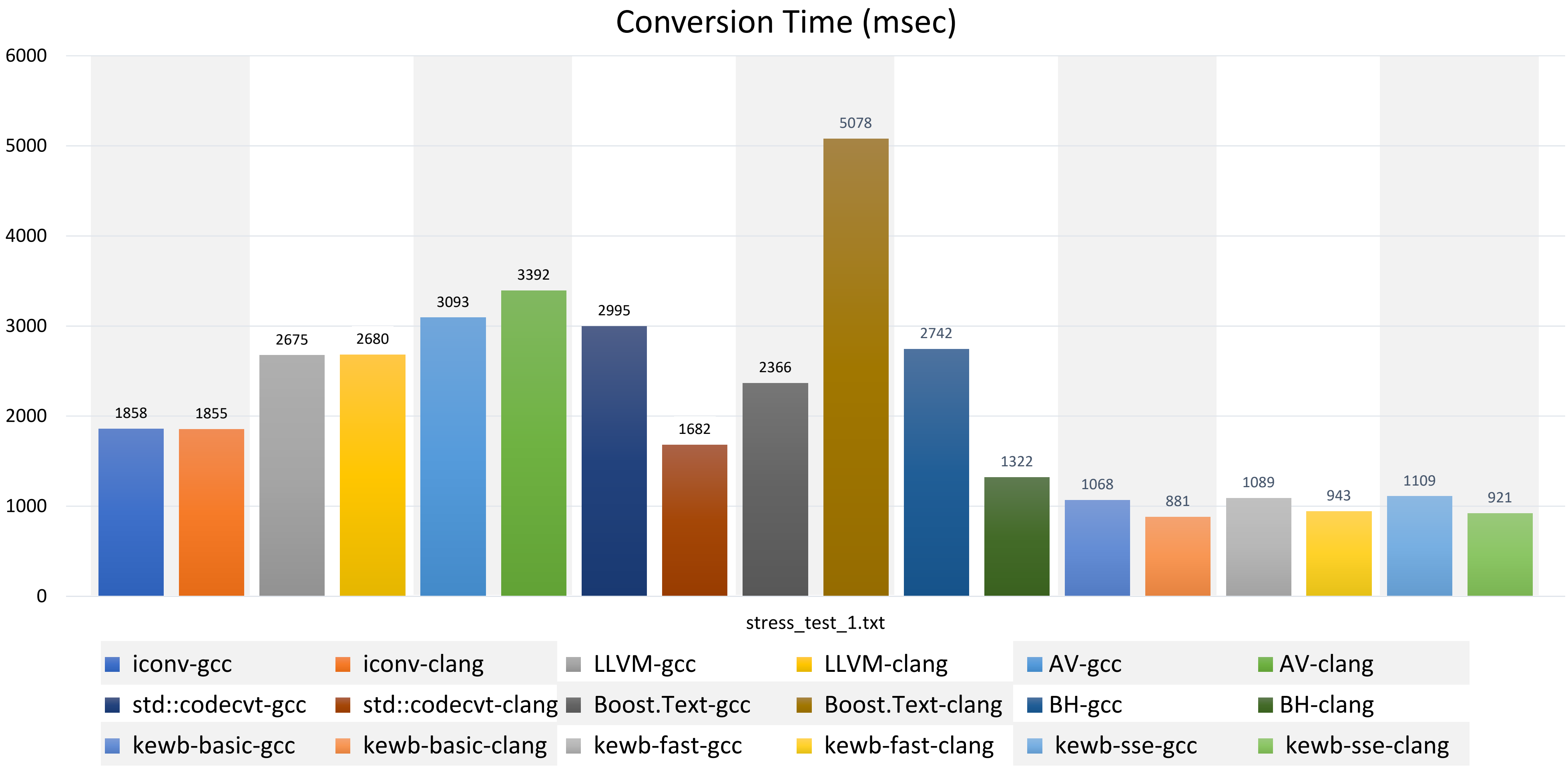
GCC 7.2/Clang 5.0.1 – Ubuntu 18.04 VM – Core i7 – UTF-32



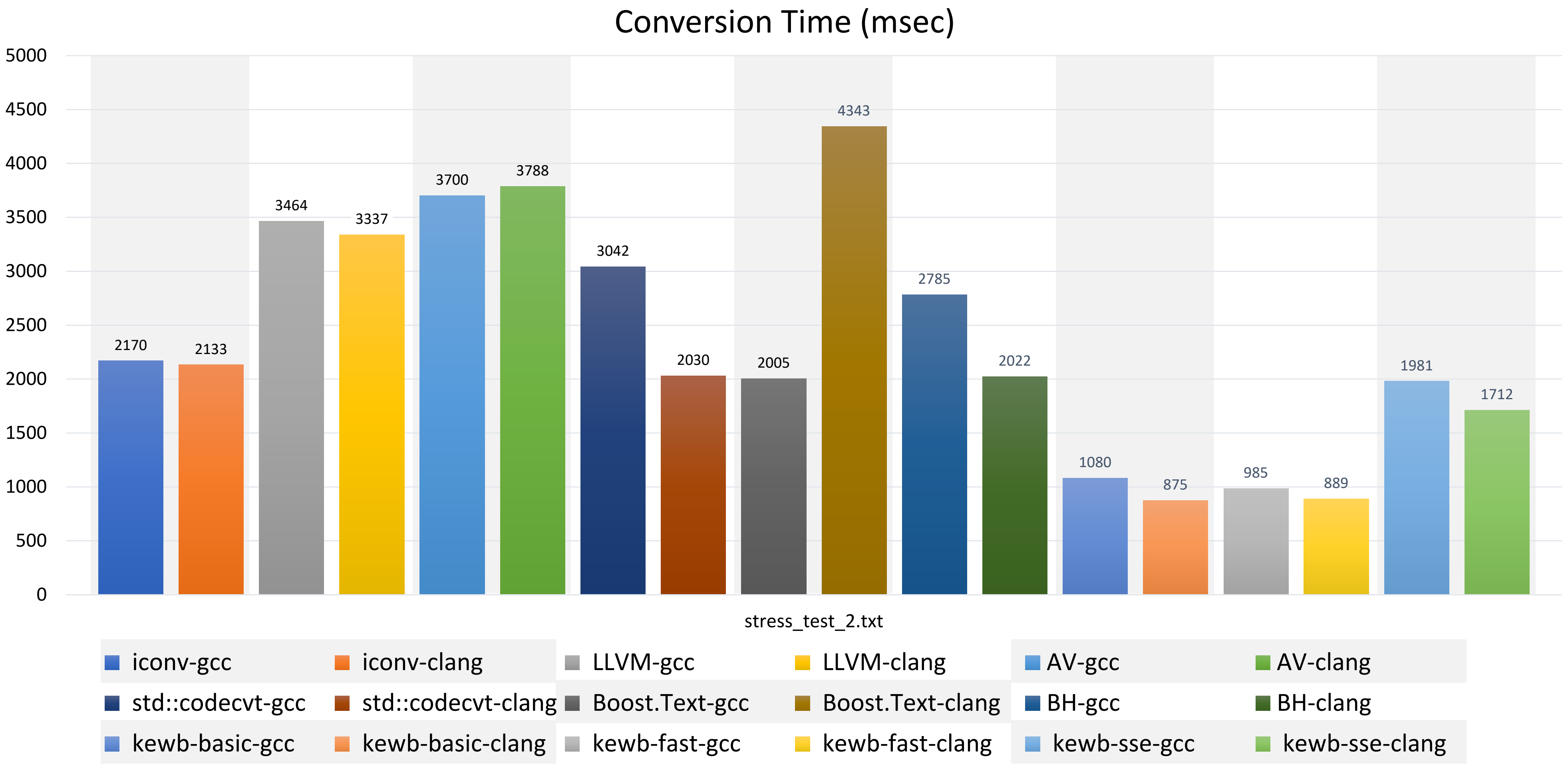
GCC 7.2/Clang 5.0.1 – Ubuntu 18.04 VM – Core i7 – UTF-32



GCC 7.2/Clang 5.0.1 – Ubuntu 18.04 VM – Core i7 – UTF-32



GCC 7.2/Clang 5.0.1 – Ubuntu 18.04 VM – Core i7 – UTF-32



Summary

Some Thoughts On Re-Use

- Error handling is intentionally limited
- Interface is intentionally small
- Actually two different table-based `Advance()` algorithms
  - *Big table* (876 bytes / 14 cache lines) and *small table* (380 bytes / 6 cache lines)
  - This talk covers the *big table* version of `Advance()`
- How to re-use
  - As library
  - Cut-and-paste

Caveats / Dragons

- Only a trivial mechanism for reporting errors
- No checking is done for null pointer arguments
- Assumes that the input and output pointers refer to buffers that exist
- Assumes that the destination buffer is appropriately sized to receive output with no overflow
- x64/x86 heritage means little endian decoding only

Future Directions / To-Do

- Provide conversions to little-endian and big-endian representations
- Provide a `Validate()` member function to check and measure length
  - IOW, a `strlen()` that validates and returns code point count
- Provide member function templates that take iterators
  - Input/Output iterators can be used with non-error-handling Basic and ASCII-optimized algorithms
  - Forward iterators required for error-handling Basic and ASCII-optimized
  - Pointers and RandomAccess iterators referring to contiguous storage can be used by all three alogrithms

Future Directions / To-Do

- Provide four-argument versions of the conversion functions that specify the output range
  - Error checking for out-of-bounds writes to the output buffer
  - Pointers and RandomAccess{contiguous}

- Provide meaningful error **reporting**
  - Type of error, and where it occurred

- Provide some common error **recovery** strategies, such as
  - Stop and return/throw immediately
  - Skip defective ranges of code units
  - Replace defective ranges of code units

To-Do: Error Handling (Basic Conversion Algorithm)

```
KEWB_ALIGN_FN std::ptrdiff_t
UtfUtils::BasicConvert(char8_t const* pSrc, char8_t const* pSrcEnd, char32_t* pDst)
{
    char32_t*   pDstOrig = pDst;
    char32_t    cdpt;

    while (pSrc < pSrcEnd)
    {
        if (Advance(pSrc, pSrcEnd, cdpt) != ERR)
        {
            *pDst++ = cdpt;
        }
        else
        {
            ImplementErrorHandlingStrategyHere(pSrc, pSrcEnd, pDst, state);
            return -1;
        }
    }

    return pDst - pDstOrig;
}
```

To-Do: Error Handling (Basic Conversion Algorithm / 4-Arg)

```
KEWB_ALIGN_FN std::ptrdiff_t
UtfUtils::BasicConvert
(char8_t const* pSrc, char8_t const* pSrcEnd, char32_t* pDst, char32_t* pDstEnd)
{
    char32_t*   pDstOrig = pDst;
    char32_t    cdpt;

    while (pSrc < pSrcEnd)
    {
        if (Advance(pSrc, pSrcEnd, cdpt) != ERR)
        {
            *pDst++ = cdpt;
        }
        else
        {
            ImplementErrorHandlingStrategyHere(pSrc, pSrcEnd, pDst, pDstEnd, state);
            return -1;
        }
    }

    return pDst - pDstOrig;
}
```

Summary

- Sometimes it pays to re-examine the algorithms and data structures used to solve a problem
- Don’t try too hard to outsmart the compiler – it is already very smart
- Build benchmarks and test, test, test, and then test some more
  - With multiple compilers
  - On multiple operating systems
  - On multiple hardware platforms
- Savor your victories!

References

- http://unicode.org/
  The Unicode Consortium
- http://www.cl.cam.ac.uk/~mgk25/unicode.html
  Markus Kuhn, *UTF-8 and Unicode FAQ for Unix/Linux*
- http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html
  ISO 10646:2017, *Universal Coded Character Set (UCS)*
- http://bjoern.hoehrmann.de/utf-8/decoder/dfa/
  Bjoern Hoehrmann, *Flexible and Economical UTF-8 Decoder*
- https://tools.ietf.org/html/rfc3629
  RFC-3629, *UTF-8 a transformation format of ISO 10646*
- https://en.wikipedia.org/wiki/UTF-8
  Wikipedia, *UTF-8*
- http://utf8everywhere.org
  UTF-8 Everywhere Manifesto
- https://github.com/tahonermann/text_view
  Tom Honermann's *text_view* GitHub repository

Questions?

Thank You for Attending!

Talk: https://github.com/BobSteagall/CppNow2018

Code: https://github.com/BobSteagall/utf_utils

Blog: https://bobsteagall.com