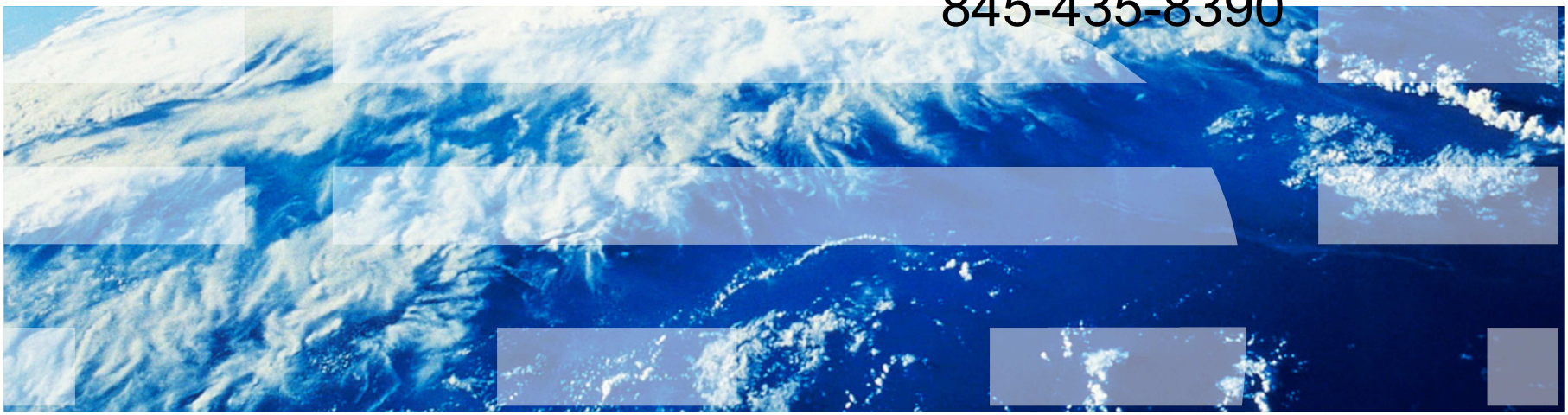
# z13 Vector Extension Facility (SIMD)

Session 16897
March 3, 2015

Jonathan Bradbury
Peter Relson
IBM Corporation
relson@us.ibm.com
845-435-8390



# Trademarks

**The following are trademarks of the International Business Machines Corporation in the United States and/or other countries.**

**IBM***
**IBM Logo***

* Registered trademarks of IBM Corporation

**The following are trademarks or registered trademarks of other companies.**

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.
IT Infrastructure Library is a registered trademark of the Central Computer and Telecommunications Agency which is now part of the Office of Government Commerce.
Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.
Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.
ITIL is a registered trademark, and a registered community trademark of the Office of Government Commerce, and is registered in the U.S. Patent and Trademark Office.
UNIX is a registered trademark of The Open Group in the United States and other countries.
Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.
Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.
Linear Tape-Open, LTO, the LTO Logo, Ultrium, and the Ultrium logo are trademarks of HP, IBM Corp. and Quantum in the U.S. and other countries.

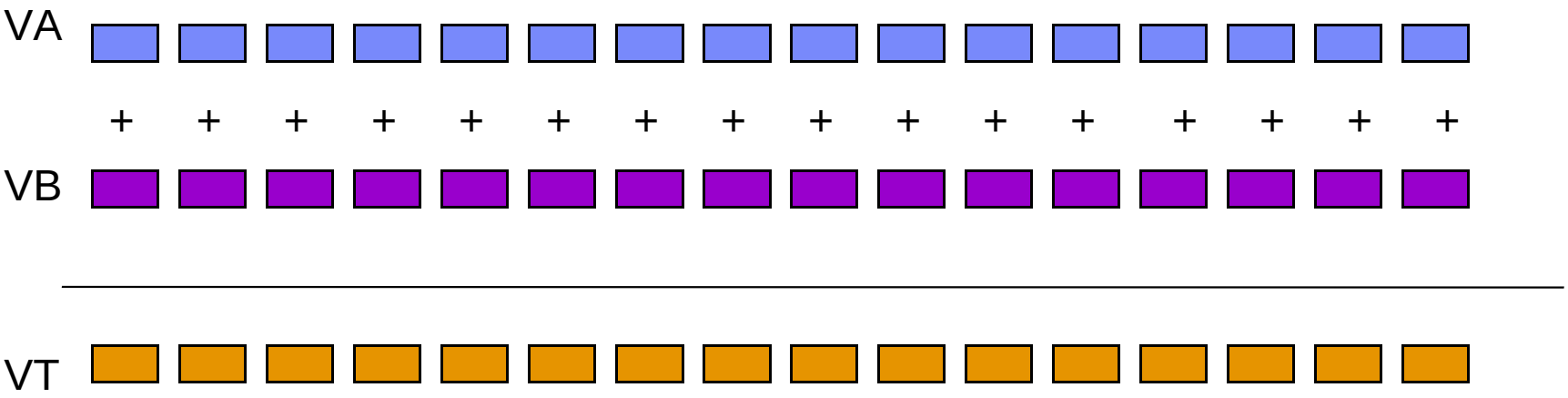
* All other products may be trademarks or registered trademarks of their respective companies.

**Notes**:
Performance is in Internal Throughput Rate (ITR) ratio based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput improvements equivalent to the performance ratios stated here.
IBM hardware products are manufactured from new parts, or new and serviceable used parts. Regardless, our warranty terms apply.
All customer examples cited or described in this presentation are presented as illustrations of the manner in which some customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics will vary depending on individual customer configurations and conditions.
This publication was produced in the United States. IBM may not offer the products, services or features discussed in this document in other countries, and the information may be subject to change without notice. Consult your local IBM business contact for information on the product or services available in your area.
All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.
Information about non-IBM products is obtained from the manufacturers of those products or their published announcements. IBM has not tested those products and cannot confirm the performance, compatibility, or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.
Prices subject to change without notice. Contact your IBM representative or Business Partner for the most current pricing in your geography.

# Agenda

- Vector/SIMD Overview
- Vector Register File
- z/OS Infrastructure and Application Considerations
- Software exploitation
- New Instructions
- String operations

# Vector/SIMD Overview




- SIMD – Single Instruction Multiple Data, also sometimes referred to as vector.

# Vector/SIMD Overview

- Each register contains multiple data elements of a fixed size. (Byte, Halfword, Word, Doubleword, Quadword)
- The collection of elements in a register is also called a vector.
- A single instruction will operate on all of the elements in the register.
- Most instructions have a non-destructive operand encoding (T=A+B vs. A=A+B)
- For most operations the CC is not set.  For a few instructions a summary condition code is used.

# SIMD theme

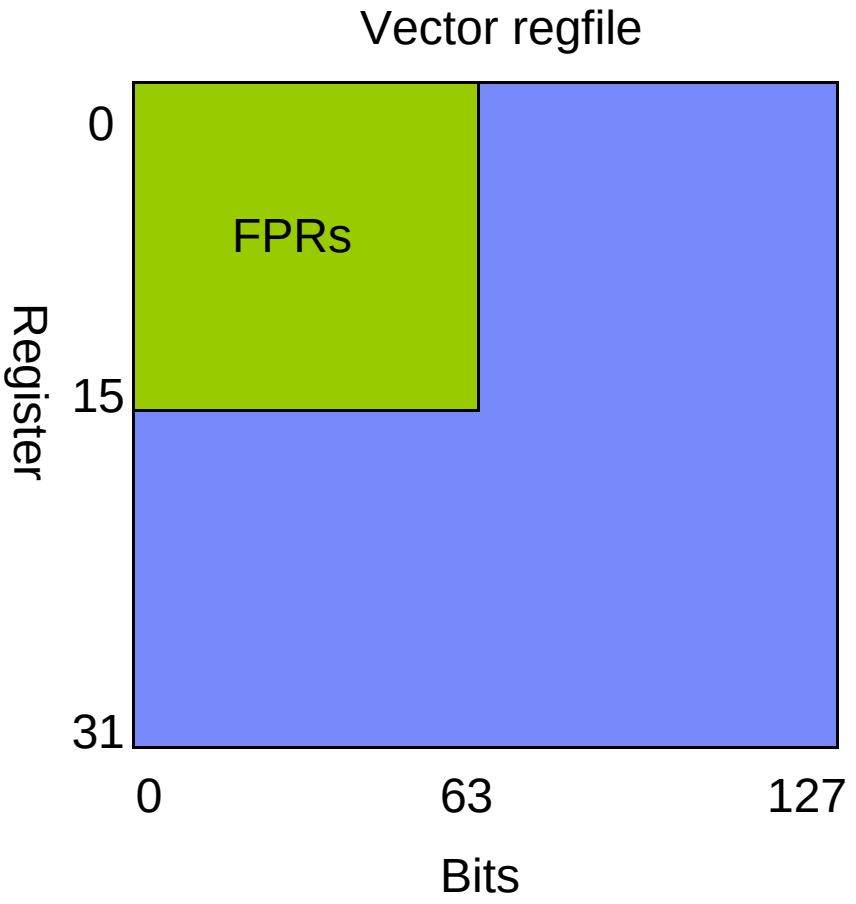
- Parallel
- Pipelined
- No Branching
  - CC captured within data result
  - Select instruction to use data result

# zSeries SIMD theme

- Efficient Loads/Stores
  - Can handle any byte alignment
  - Most efficient loads on 8 byte boundaries
  - String operations supported to basic block boundaries or to specific lengths
  - Gather and Scatter
- Floating-Point
  - Clean trap result to be suppressed
  - Scalar or Vector
  - Expandable to other data types (radix and lengths)

# Overlaid Vector/FPR register files

- Overlay the FPRs over the vector register file
- Bits 0:63 of SIMD registers 0-15 will correspond to FPRs 0-15
- When using an FPR, bits 64:127 of the corresponding vector register will become unpredictable



# Overlaid Vector/FPR register files

- CR0.45 still works in the same way for AFP
- New CR0.46 bit for enabling use of vector instructions
  - CR0.45 also needs to be set to 1
- New DXC (0xFE) if any access outside of traditional FPRs
  - System uses same process as for AFP registers

## Customer Control

- LOADxx parmlib member MACHMIG statement supports VEF option
  - Indicates that customer does not want applications to use VEF (usually in a migration scenario, trying to validate new hardware without necessarily getting exploitation of new hardware functionality)

# z/OS Support of Vector Extension Facility

## Application Control

- Bit CVTVEF indicates if an application may use VEF
- Application use automatically enabled upon first use for enabled, unlocked task
- IEAFP START: enable for subsequent use (such as by SRB)
- IEAFP STOPVECTOR: stop status saving of vector registers (when work unit no longer needs)
- IEAFP STOP (existing): stops status saving both of vector registers and AFPRs

# z/OS Support of Vector Extension Facility

## Application Considerations

- Be very aware that any use of a FPR will change all 16 bytes of the corresponding VR (this includes even LD)
- Linkage Convention (caller may assume across a call)
  - VRs 0 to 7 are volatile
  - For VRs 8 to 15: Bytes 0-7 are non-volatile, Bytes 8-15 are volatile
  - VRs 16 to 23 are non-volatile
  - VRs 24 to 31 are volatile
- As with FPRs, the first recovery routine gets control with time of error VRs, and any changes made prior to retry are reflected upon retry
- If VRs are in use by “Fork parent” they will be propagated to “Fork child”
- Disabled Interrupt Exits (DIEs) must not use VEF instructions

# z/OS Support of Vector Extension Facility

System structure

- New control block Extended Status Save Area (ESSA, mapped by IHAESSA) to hold vector register contents across status save (undispatch) and status restore (redispatch)
- Pointed to by task's STCB
- Used only when the work unit is known to be using VEF

# Exploitation of SIMD

- MASS and ATLAS libraries updated to exploit SIMD (Mathematical Accelerator Subsystem, Automatically Tuned Linear Algebra Software)
- z/OS Unix:
  - DBX supports reading and writing vector registers
  - Support added in PTRACE (BPX1PTR, BPX4PTR) to enable debuggers to read and write vector registers in debugged processes
- XL C/C++ and Cobol 5.2 use SIMD for string operations
  - C/C++ provides vector data type and vector facility built-ins
- XMLSS uses SIMD for character and string manipulation
- Java SDK 8 exploits SIMD

# z/OS Exploitation of SIMD (cont)

- LE – printf, scanf support for vector types
- PL/I – uses SIMD for SEARCH and VERIFY built-ins

# z/OS Exploitation of SIMD: simple example

```
#include <builtins.h>
int main() {
  vector signed int a = {-1, 2, -3, 4};
  vector signed int b = {-5, 6, -7, 8};
  vector signed int c, d;
  c = a + b;
  d = vec_abs(c);  //Generates VLP (VLPF)
  printf("d[0] = %d\n",d[0]);
  printf("d[1] = %d\n",d[1]);
  printf("d[2] = %d\n",d[2]);
  printf("d[3] = %d\n",d[3]);
  Return 0;
  }
```

>xlc **-qVECTOR -qARCH=11** a.c

# Vector Instructions supported

- Integer
  - 8b to 128b add, sub
  - 128b add with carry, subtract with carry
  - 8b to 64b min, max, avg, abs, compare
  - 8b to 32b multiply, multiply/add 4 - 32 x 32 multiply/adds
  - Logical ops, shifts,
  - Carryless Multiply (8b to 64b), Checksum (32b),
  - Memory accesses efficient with 8B alignment; minor penalties for other alignments if they cross a cache line
  - Gather by Step

# Vector Instructions supported

- String
  - Find 8b, 16b, 32b, equal or not equal with zero character end
  - Range compare
  - Find any equal
  - Isolate String
  - Load to block boundary, load/store with length to avoid access exceptions

- Floating-point - BFP Double Precision only 32 x 2 x 64b registers
  - 2 BFUs with an increase in architected registers
  - Exceptions suppress

# Vector Load Instructions

- VECTOR LOAD
  - VL $V_1$,$D_2$($X_2$,$B_2$)
  - Load 16 bytes from storage into $V_1$. **No alignment requirement**
- VECTOR LOAD AND REPLICATE
  - VLRP(B|H|F|G) $V_1$,$D_2$($X_2$,$B_2$),$M_3$
  - Load 1-8 bytes and replicate across all elements of $V_1$
- VECTOR LOAD ELEMENT
  - VLE(B|H|W|D) $V_1$,$D_2$($X_2$,$B_2$),$M_3$
  - The element sized second operand is placed into $V_1$ at index $M_3$
- VECTOR LOAD ELEMENT IMMEDIATE
  - VLEI(B|H|F|G) $V_1$,$I_2$,$M_3$
  - Places $I_2$ in $V_1$ at index $M_3$, leaves rest of vector unchanged
- VECTOR LOAD MULTIPLE
  - VLM $V_1$,$V_3$,$D_2$($B_2$),$M_4$
  - Up to 16 VRs loaded from storage

# Vector Load Instructions (cont)

- VECTOR LOAD TO BLOCK BOUNDARY
  - VLBB $V_1$,D2($X_2$,$B_2$),$M_3$
  - Loads up to 16 bytes into $V_1$ without crossing block boundary specified by $M_3$
- LOAD COUNT TO BLOCK BOUNDARY
  - LCBB $R_1$,$D_2$($X_2$,$B_2$),$M_3$
  - Loads $R_1$ with number of bytes that can be loaded with specified block size
- VECTOR LOAD WITH LENGTH
  - VLL $V_1$,$D_2$($B_2$),$R_3$
  - Loads the number of bytes specified in $R_3$ from storage into $V_1$
- VECTOR LOAD LOGICAL ELEMENT AND ZERO
  - VLLEZ (B|H|F|G) $V_1$,$D_2$($X_2$,$B_2$),$M_3$
  - Load element sized data from second operand address and place right justified in leftmost doubleword
- VECTOR GATHER ELEMENT
  - VGE (F|G) $V_1$,$D_2$($V_2$,$B_2$),$M_3$
  - Loads element from memory addressed by $B_2$+$V_2$($M_3$)+$D_2$

# Vector Store Instructions

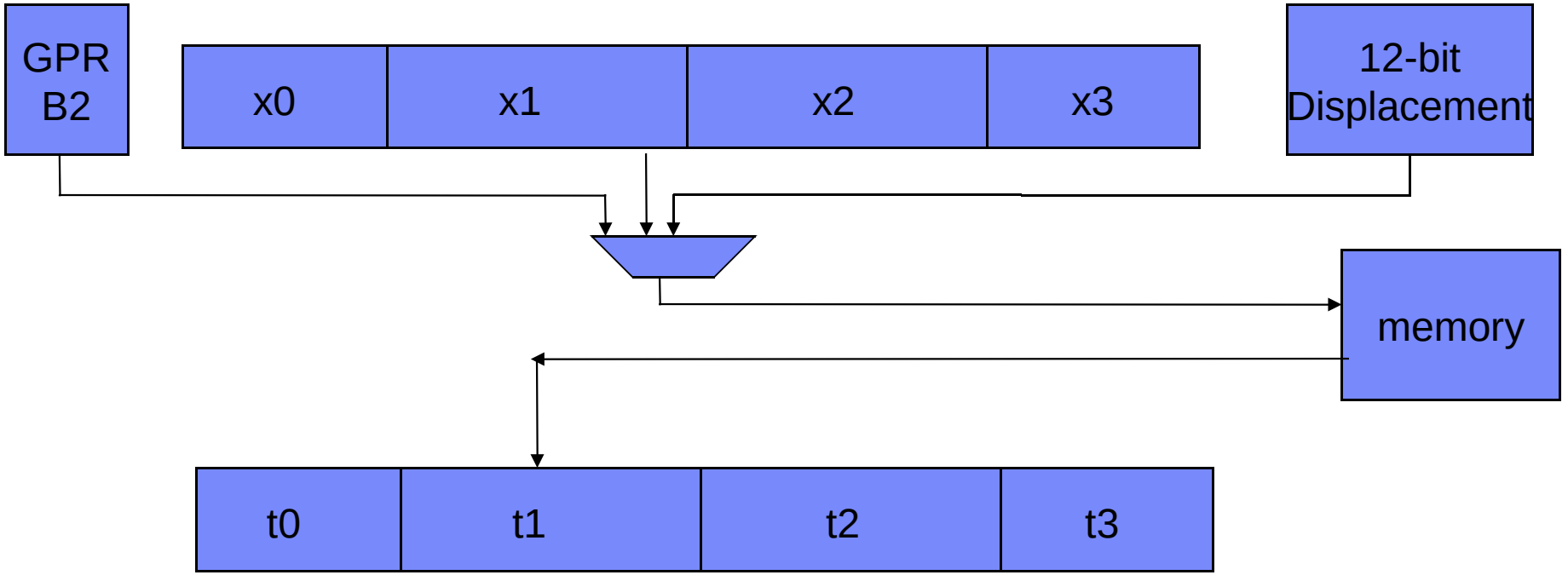
- VECTOR STORE
  - VST $V_1,D_2(X_2,B_2)$
  - Stores 16 bytes on byte boundary, no alignment required
- VECTOR STORE ELEMENT
  - VSTE(B|H|F|G) $V_1,D_2(X_2,B_2),M_3$
  - Stores element of $VR_1$ indexed by $M_3$ to second operand
- VECTOR STORE MULTIPLE
  - VSTM $V_1,V_3,D_2(B_2),M_4$
  - Stores range of up to 16 VRs to second operand location

# Vector Store Instructions (Cont)

- VECTOR STORE WITH LENGTH
  - VSTL $V_1$,$D_2$($B_2$),$R_3$
  - Stores the number of bytes specified by $R_3$ from $V_1$ into the second operand location
- VECTOR SCATTER ELEMENT
  - VSCE(F|G) $V_1$,$D_2$($V_2$,$B_2$),$M_3$
  - Stores element of $V_1$ indexed by $M_3$ to memory addressed by $B_2$+$V_2$($M_3$)+$D_2$

# Gather / Scatter by Step (new)

- Vector Gather Element 32b, 64b,
- Vector Scatter Element 32b, 64b



# Vector Move Ops

- VECTOR LOAD REGISTER
  - VLR $V_1$,$V_2$
  - Copies $V_2$ to $V_1$
- VECTOR LOAD GR FROM VR ELEMENT
  - VLGV(B|H|F|G) $R_1$,$V_3$,$D_2(B_2)$
  - Copies element indexed by $D_2(B_2)$ from $V_3$ into $R_1$
- VECTOR LOAD VR ELEMENT FROM GR
  - VLVG(B|H|F|G) $V_1$,$R_3$,$D_2(B_2)$
  - Copies element sized portion of $R_3$ to $V_1$ at element index specified by $D_2(B_2)$
- VECTOR LOAD VR FROM GR PAIR DISJOINT
  - VLVGP $V_1$,$R_2$,$R_3$
  - $V_1$ <- $R_2$ || $R_3$

# Vector Manipulation Ops

- Vector Permute VPERM $V_1,V_2,V_3,V_4$
  - $V_1[i] = (V_2 \,||\, V_3)[VR_4(i)]$ for i=0..16
- Vector Permute Doubleword Immediate (DW)
  VPDI $V_1, V_2, V_3, M_4$
- Vector Select (QW) - VSEL $V_1, V_2, V_3, V_4$
  $V_1 = V_2.x$ if $V_4.x=1$, else $V_3.x$ for x=0..127
- Vector Pack (modulo) (DW->W,W->HW,HW->B)
  VPK $V_1, V_2, V_3, M_4$
- Vector Pack Saturate (DW->W,W->HW,HW->B)
  VPKS $V_1, V_2, V_3, M_4, M_5$cc
- Vector Pack Logical Saturate (DW->W,W->HW,HW->B)
  VPKLS $V_1, V_2, V_3, M_4, M_5$cc
  - For Saturating Packs CC0-no saturation CC1- Saturation

# Vector Manipulation Ops

- Vector Unpack (Signed|Unsigned) High (W->DW, HW->W, B->HW)
  VUPH $V_1$, $V_2$,$M_3$; VUPLH $V_1$, $V_2$, M3
- Vector Unpack (Signed|Unsigned) Low (W->DW, HW->W, B->HW)
  VUPL $V_1$, $V_2$, M3; VUPLL $V_1$, $V_2$, M3
- Vector Replicate (DW, W, HW, B)
  VREP $V_1$, $V_3$, $I_2$, $M_4$
  $V_1$ <= replicate $V_3(I_2)$
- Vector Replicate Immediate (DW,W,HW,B)
  VREPI $V_1$, $I_2$, $M_3$
- Vector Merge High (DW,W,HW,B)
  VMRH $V_1$, $V_2$, $V_3$, $M_4$
- Vector Merge Low (DW,W,HW,B)
  VMRL $V_1$, $V_2$, $V_3$, $M_4$

# Vector Arithmetic Instructions

- Vector Add (modulo) (QW,DW,W,HW,B)
  VA $V_1$, $V_2$, $V_3$, $M_4$
  $V_1 = V_2 + V_3$
- Vector Add Compute Carries (QW,DW,W,HW,B)
  VACC $V_1$, $V_2$, $V_3$, $M_4$
  $V_1$ = carry out of $V_2 + V_3$
- Vector Subtract (modulo) (QW,DW,W,HW,B)
  VS $V_1$, $V_2$, $V_3$, $M_4$
  $V_1 = V_2 – V_3$
- Vector Subtract Compute Borrows (QW,DW,W,HW,B)
  VSCBI $V_1$, $V_2$, $V_3$, $M_4$
  $V_1$ = carry out of $V_2 + \sim V_3 + 1$
- Vector Add with Carry (QW)
  VAC $V_1$, $V_2$, $V_3$, $V_4$, $M_5$
  $V_1 <= V_2 + V_3 + 000\ V_4.127$

# Vector Arithmetic Instructions

- Vector Add with Carry Compute Carry (QW)
  VACCC $V_1$, $V_2$, $V_3$, $V_4$, $M_5$
  V1 <= carry out of V2 + V3 + 0000.V4.127
- Vector Subtract with Borrow Indication (QW)
  VSBI $V_1$, $V_2$, $V_3$, $V_4$, $M_5$
- Vector Subtract with Borrow Compute Borrow Indication (QW)
  VSBCBI V1, V2, B3, V4, M5

- Vector Sum Across Doubleword (4HW,2W)
  VSUMG $V_1$, $V_2$, $V_3$, $M_4$
- Vector Sum Across Quadword (4W, 2D)
  VSUMQ $V_1$, $V_2$, $V_3$, $M_4$
- Vector Sum Across Word (4B, 2HW) V
  SUM $V_1$, $V_2$, $V_3$, $M_4$

# Vector Arithmetic Instructions

- Vector Average (DW,W,HW,B)
  VAVG $V_1$, $V_2$, $V_3$, $M_4$
- Vector Average Logical (DW,W,HW,B)
  VAVGL $V_1$, $V_2$, $V_3$, $M_4$

- Vector Load Positive (DW, W, HW, B) absolute value
  VLP $V_1$, $V_2$, $M_3$
- Vector Load Complement (DW, W, HW, B) 2's complement
  VLC $V_1$, $V_2$, $M_3$

# Vector Multiply

- Vector Multiply Low (W,HW,B)  VML $V_1$, $V_2$, $V_3$, $M_4$
- Vector Multiply High (W,HW,B)  VMH $V_1$, $V_2$, $V_3$, $M_4$
- Vector Multiply High Logical (W,HW,B)  VMLH $V_1$, $V_2$, $V_3$, $M_4$
- Vector Multiply Even (W,HW,B)  VME $V_1$, $V_2$, $V_3$, $M_4$
- Vector Multiply Even Logical (W,HW,B)  VMLE $V_1$, $V_2$, $V_3$, $M_4$
- Vector Multiply Odd (W,HW,B)  VMO $V_1$, $V_2$, $V_3$, $M_4$
- Vector Multiply Odd Logical (W,HW,B) VMLO $V_1$, $V_2$, $V_3$, $M_4$

# Vector Multiply and Add

- All Multiply and Add instructions have non-destructive encodings
- Vector Multiply and Add Low (W,HW,B)
  VMAL $V_1$, $V_2$, $V_3$, $V_4$, $M_5$
- Vector Multiply and Add High (W,HW,B)
  VMAH $V_1$, $V_2$, $V_3$, $V_4$, $M_5$
- Vector Multiply and Add High Logical (W,HW,B)
  VMALH $V_1$, $V_2$, $V_3$, $V_4$, $M_5$
- Vector Multiply and Add Even (W,HW,B)
  VMAE $V_1$, $V_2$, $V_3$, $V_4$, $M_5$
- Vector Multiply and Add Even Logical (W,HW,B)
  VMALE $V_1$, $V_2$, $V_3$, $V_4$, $M_5$
- Vector Multiply and Add Odd (W,HW,B)
  VMAO $V_1$, $V_2$, $V_3$, $V_4$, $M_5$
- Vector Multiply and Add Odd Logical (W,HW,B)
  VMALO $V_1$, $V_2$, $V_3$, $V_4$, $M_5$

# Vector Bit Ops

- Vector AND (QW)  VN  $V_1$, $V_2$, $V_3$
  $V_1 = V_2$ & $V_3$
- Vector AND with compliment (QW) v1=v2 & ~v3  VNC
- Vector OR (QW)  VO
- Vector NOR (QW)
  - VNO v1,v2,v3
- Vector XOR (QW)  VX
- Vector Element Rotate Left (DW,W,HW,B)  VERLL, VERLLV
- Vector Element Shift Left (QW,DW,W,HW,B)  VESL, VESLV

# Vector Bit Ops

- Vector Shift Left by Byte (QW)  VSLB
- Vector Shift Left by bit  VSL
- Vector Element Shift Right Arithmetic (QW,DW,W,HW,B)
  VESRA, VESRAV
- Vector Element Shift Right Logical (QW,DW,W,HW,B)
  VESRL, VESRLV
- Vector Count Leading Zeros (DW,W,HW,B)  VCLZ
- Vector Count Trailing Zeros (DW,W,HW,B)  VCTZ
- Vector Population Count (B)  VPOPCT

# Vector Bit Ops

- Vector Population Count (B)
  VPOPCT
- Vector Rotate and Insert Under Mask (DW,W,HW,B)
  VERIM
  - Rotate amt in i-text. Mask VR passed in to select the bits to insert
- Vector Generate Byte Mask Immediate
  VGBM $V_1$, $I_2$
  - Sets bytes of $V_1$ to all ones or zeros depending on each bit of $I_2$ field
- Vector Generate Mask (DW, W, HW,B)
  VGM $V_1$, $I_2$, $I_3$, $M_4$
  - Generates a mask giving a start and stop bit position

# Vector Compare Ops

- Vector Compare Equal (DW,W,HW,B)
- Vector Compare Greater (DW,W,HW,B)
- Vector Compare Greater Logical (DW,W,HW,B)
  - Above 3 Compare Ops set the CC in the following way
    - CC0 – All compares have true result
    - CC1 – Mixed results
    - CC3 – All compares have false result
  - Allows for shortcut code

# Vector Compare Ops

- Vector Compare Element (DW,W,HW,B)
- Vector Compare Element Logical (DW,W,HW,B)
  - Above 2 Compare ops compare a single element right justified in leftmost doubleword
  - CC is set as it would be for a GPR compare
- Vector Minimum (DW,W,HW,B)
- Vector Minimum Logical (DW,W,HW,B)
- Vector Maximum (DW,W,HW,B)
- Vector Maximum Logical (DW,W,HW,B)

# Misc Vector Ops

- Vector Checksum (W)
- Vector Galois Field Multiply Sum (D,W,HW,B)
- Vector Galois Field Multiply Sum and Accumulate (D,W,HW,B)
  - Used for CRC

# Vector Floating Point Extensions

- Double precision binary floating point only for zNew
- Both single element and vector modes
- Single opcode for single element and vector instruction.  Mask bit to determine which to use.  Precision is also in the mask field but only double precision is allowed at this time.

# IEEE Floating Point Exceptions

- For vector operations all IEEE trapping exceptions will suppress. (How POWER and Intel work)
  - New PIC and VXC code to specify which exception and the lowest indexed element to take the exception.
  - VXC is just an overlay of the DXC in fixed storage and FPC register
- All non-trapping exceptions will work as defined in chapter 19 with the exceptions from each element being ORed into the flags.

# Floating Point Instructions

- VECTOR FP ADD
- VECTOR FP COMPARE EQUAL
- VECTOR FP COMPARE GREATER
- VECTOR FP COMPARE GREATER EQUAL
- VECTOR FP CONVERT FROM FIXED 64-BIT
- VECTOR FP CONVERT FROM LOGICAL 64-BIT
- VECTOR FB CONVERT TO FIXED 64-BIT
- VECTOR FP CONVERT TO LOGICAL 64-BIT
- VECTOR FP DIVIDE

# Floating Point Instructions

- VECTOR LOAD FP INTEGER
- VECTOR LOAD LENGTHENED (Short->Long)
- VECTOR LOAD ROUNDED (Long->Short)
- VECTOR FP MULTIPLY
- VECTOR FP MULTIPLY AND ADD
- VECTOR FP MULTIPLY AND SUBTRACT
- VECTOR FP PERFORM SIGN OPERATION
  - Single instruction to compliment, force positive, or force negative
- VECTOR FP SQUARE ROOT
- VECTOR FP SUBTRACT
- VECTOR FP TEST DATA CLASS IMMEDIATE

# New Program Interrupts

- New Data Exception for use of Vector registers when the facility is not enabled
  - DXC 0xFE

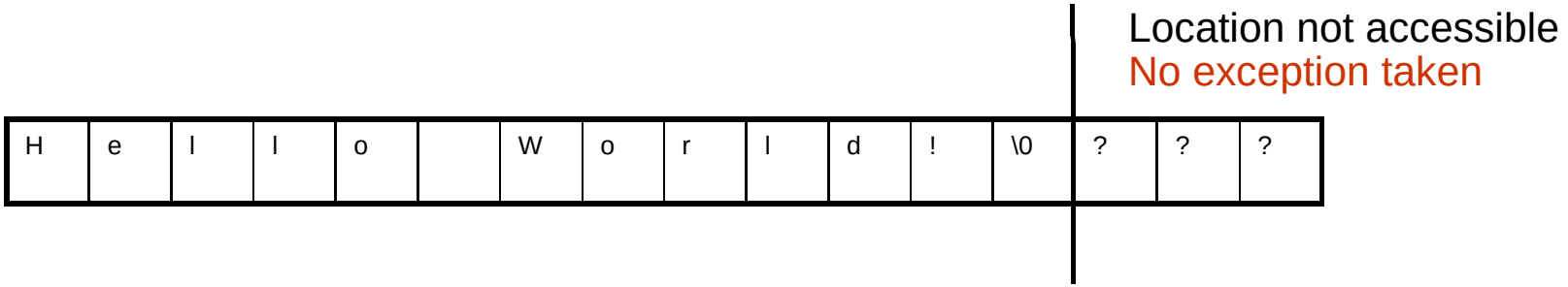
- New program interrupt for Vector Data Exception
  - PIC 0x001B
  - Will always suppress
  - Will store a VXC in the FPC and prefix in the same location as the DXC
  - VXC will contain an interruption reason as well as the index of the leftmost element that took the exception

# String Acceleration Challenges

- Strings can start on any byte/halfword/word boundary
- C-Style strings may end near a page crossing
  - Not known until comparison is done
  - If more bytes are read unwarranted exceptions might occur
- Java strings may also end near the end of a page (although you know when this will happen)
- Substring search is a difficult problem

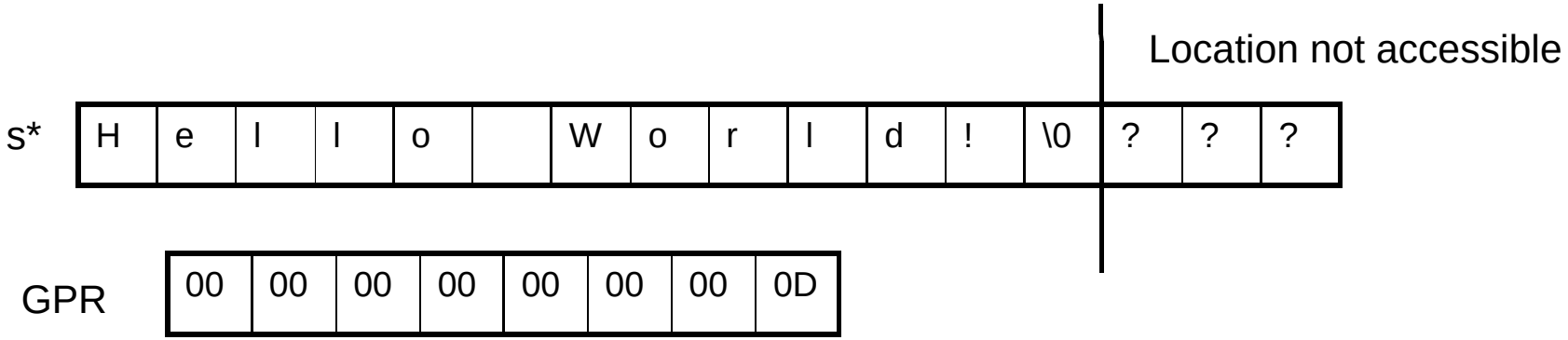
# Load to Block Boundary Instruction

VLBB – Loads a VR with as many bytes as it can without crossing a block boundary.
Immediate field to specify block size (64B,128B,256B,2K,4K,etc)



# Load Count to Block Boundary

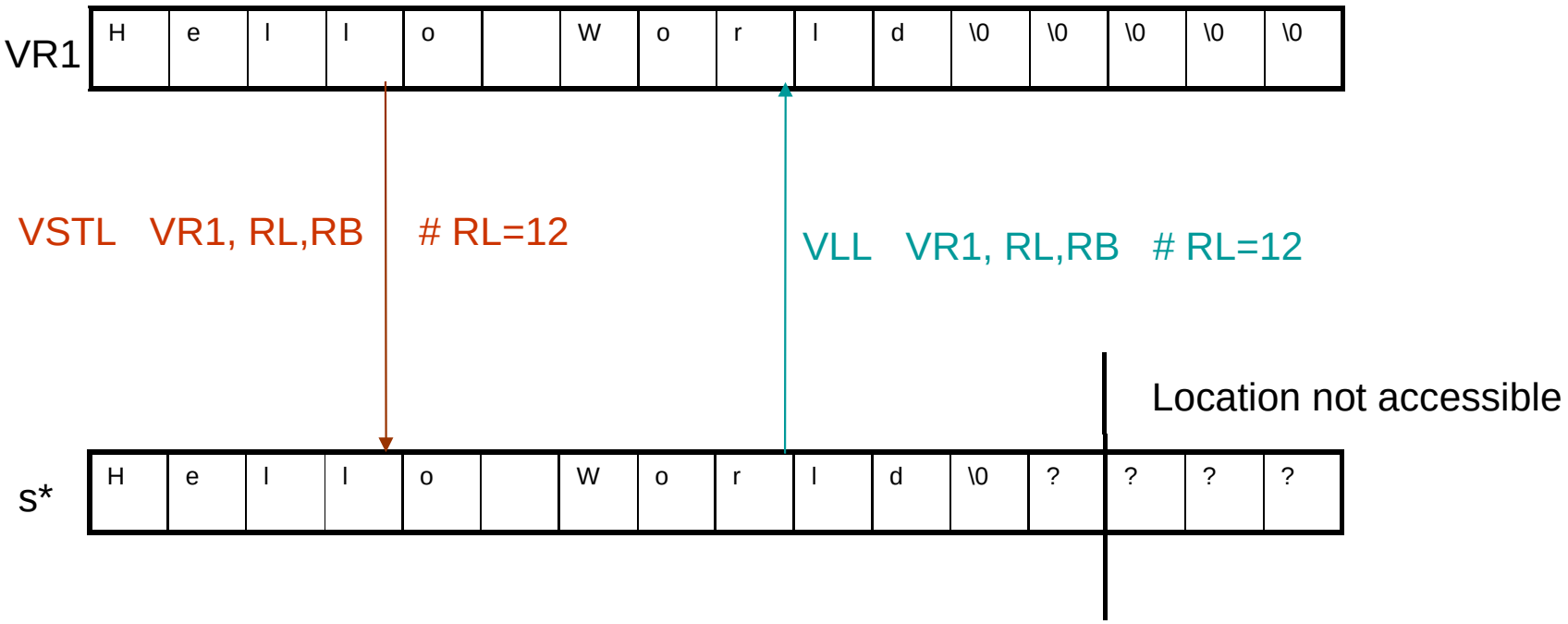
LCBB – Determines the number of bytes loaded from a D(X,B) address.
Stores result in a GPR
Sets condition code indicating full or partial vector load.



# Load/Store with Length

Provide a means to load or store a partial vector with a specified number of consecutive bytes
Will not take an exception on bytes not loaded or stored

Length and base address in GPRs



# String Search Instructions

- FIND ELEMENT NOT EQUAL OR ZERO
  - VFENEZ VR1,VR2,VR3
  - Compares VR2 to VR3 from left to right for byte/halfword/word inequality or 0
  - Stores index of leftmost miscompare in VR1 or 16 if equal
  - Optionally sets condition code to 0 if zero is found, 1 if VR2<VR3, 2 if VR2 > VR3, and 3 if equal
- Find ELEMENT Not Equal
  - Same as VFENEZ except no zero compare only CC1-3 are set.

# VFENEZ Details

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| V2 | H | e | l | l | o | | W | o | r | l | d | ! | \0 | \0 | \0 | \0 |
| | = | = | = | = | = | = | = | = | = | = | = | = | Z | | | |
| V3 | H | e | l | l | o | | W | o | r | l | d | ! | \0 | \0 | \0 | \0 |
| V1 | \0 | \0 | \0 | \0 | \0 | \0 | \0 | 12 | \0 | \0 | \0 | \0 | \0 | \0 | \0 | \0 |

Set CC=0

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| V2 | H | e | l | l | o | | W | o | r | l | d | ! | \0 | \0 | \0 | \0 |
| | = | = | = | = | = | = | ≠ | | | | | | | | | |
| V3 | H | e | l | l | o | | S | u | n | ! | \0 | \0 | \0 | \0 | \0 | \0 |
| V1 | \0 | \0 | \0 | \0 | \0 | \0 | \0 | 6 | \0 | \0 | \0 | \0 | \0 | \0 | \0 | \0 |

Set CC=2

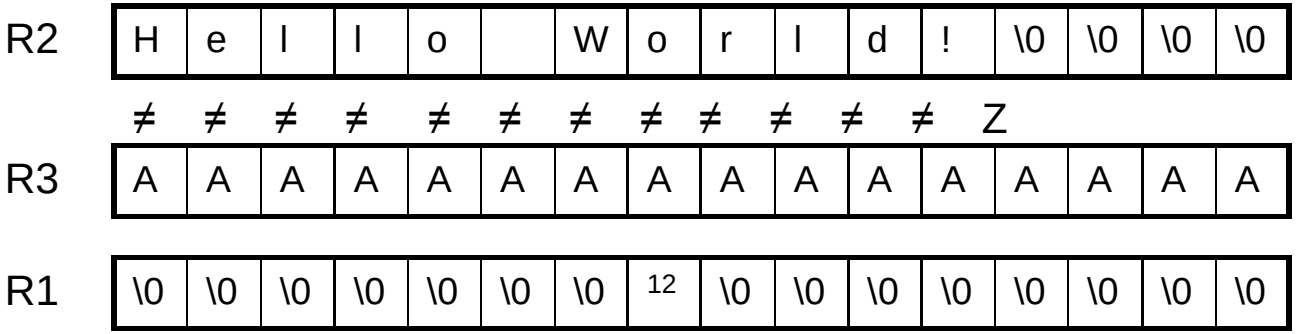
| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| V2 | H | e | l | l | o | | W | o | r | l | d | ! | | H | o | w |
| | = | = | = | = | = | = | = | = | = | = | = | = | = | = | = | = |
| V3 | H | e | l | l | o | | W | o | r | l | d | ! | | H | o | w |
| V1 | \0 | \0 | \0 | \0 | \0 | \0 | \0 | 16 | \0 | \0 | \0 | \0 | \0 | \0 | \0 | \0 |

Set CC=3

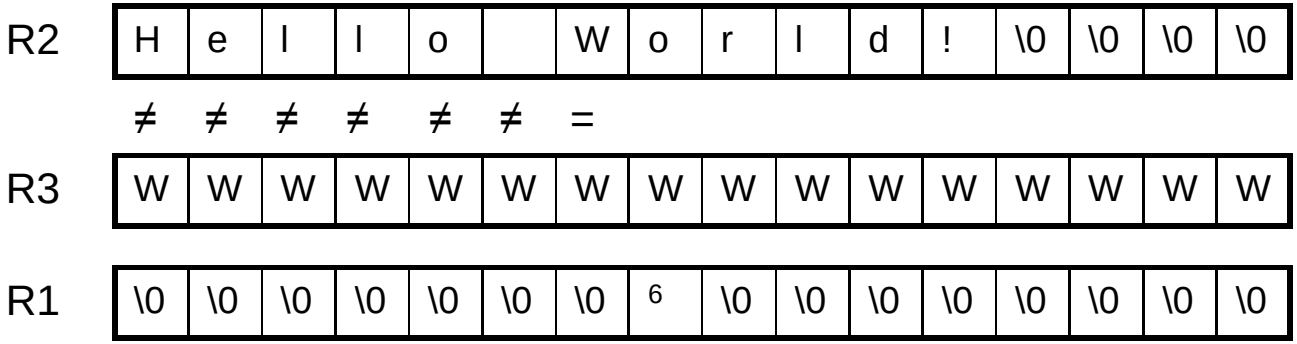
# FIND ELEMENT EQUAL

- VECTOR FIND ELEMENT EQUAL OR ZERO
  - VFEEZ VR1,VR2,VR3
  - Compares VR2 to VR3 from left to right for byte/halfword/word equality or 0
  - Stores index of leftmost equality in VR1 or 16 if none equal
  - Optionally sets CC-0 if zero found, CC-1 if bytes are equal, CC-3 if no bytes are equal
- Find Byte/Halfword Equal
  - Same as VFEEZ except no zero compare, only CC1,3 are set

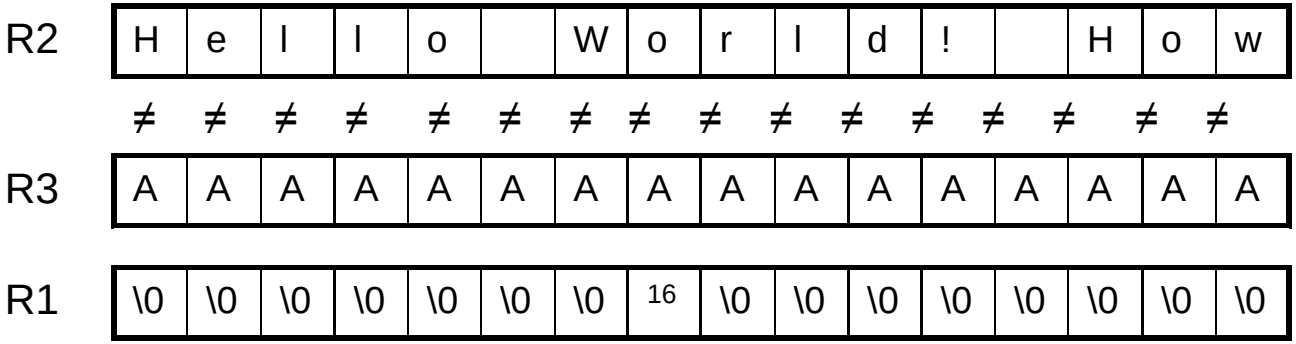
# VFEEZ Details




Set CC=0



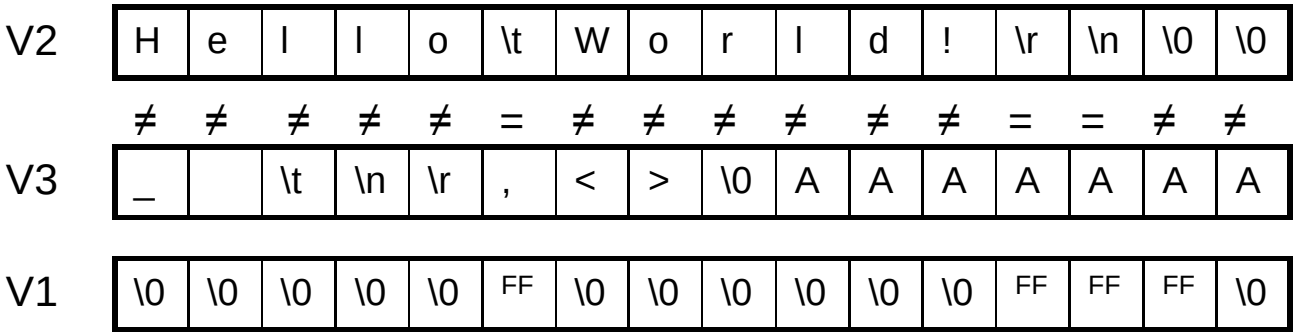

Set CC=1




Set CC=3

# Other String Instructions

Equal Any: Set Condition code if any byte/halfword/word in $V_2$ equals any byte/halfword/word in $V_3$




Range Compare: $V_2$ – String to search

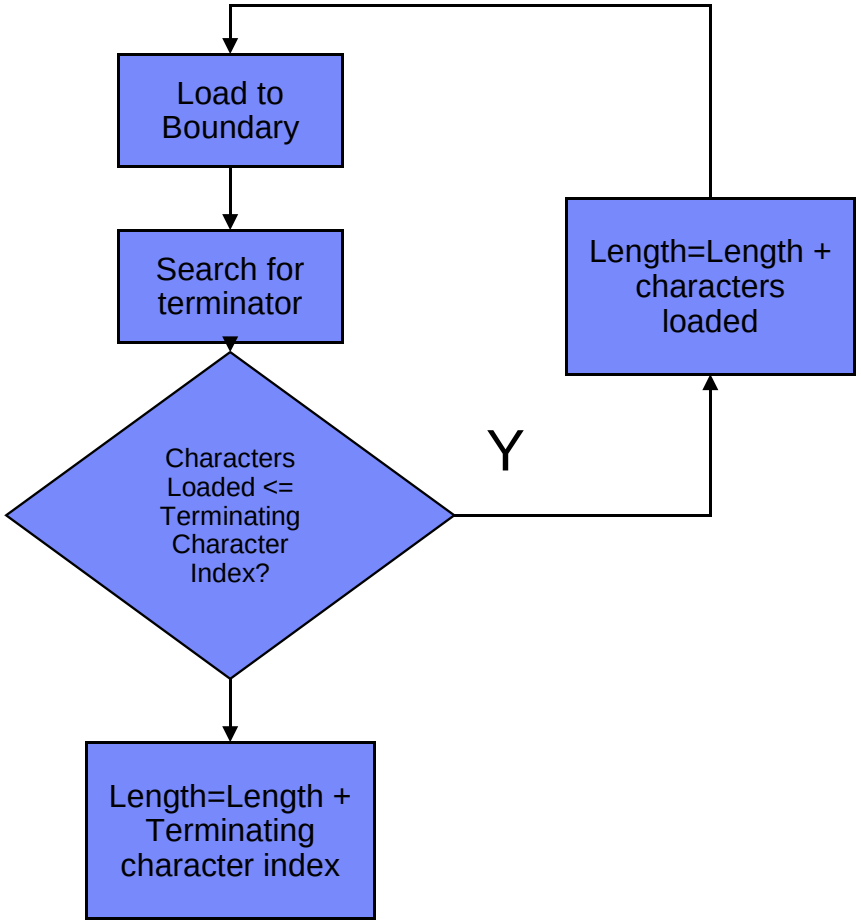
$V_3$ – 1-8 pairs of characters to search for

$V_4$ – control vector to specify >,<,= for each character

# Range Compare

| | | H | e | l | l | o | | W | o | r | l | d | ! | \0 | ?? | ?? | ?? |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | ge | T | T | T | T | T | F | T | T | T | T | T | F | F | | | |
| Z | Le | T | F | F | F | F | T | T | F | F | F | F | F | T | | | |
| a | ge | F | T | T | T | T | F | F | T | T | T | T | F | F | | | |
| z | le | T | T | T | T | T | T | T | T | T | T | T | T | T | | | |
| 0 | ge | T | T | T | T | T | F | T | T | T | T | T | T | F | | | |
| 9 | le | F | F | F | F | F | T | F | F | F | F | F | F | T | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| RT | 0 | FF | FF | FF | FF | FF | 00 | FF | FF | FF | FF | FF | 00 | 00 | ?? | ?? | ?? |
| RT | 1 | 00 | 00 | 00 | 00 | 00 | 00 | 05 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |

# strlen()



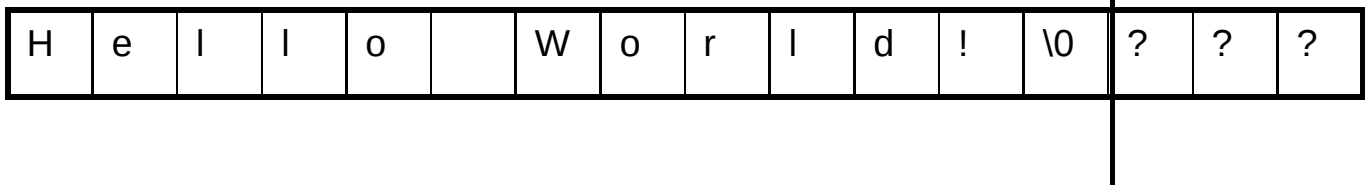
## size_t strlen(const char* s) with LBB

- R2 - @ of string, R1 will contain length

```
 XGRK R1,R1,R1          Zero out running index
LOOP: VLBB   V16,0(R1,R2),6 Load up to 16 bytes
 LCBB R3,0(R1,R2),6    Find how many bytes were loaded (GLEN)
 ALGRK   R1,R1,R3      Increment length by bytes loaded
 VFENEBZ V17,V16,V16   Look for 0 byte
 VLVGB   R4,V17,7(R0)  Extract index to gpr (16-no match) (GPOS)
 CLGR R3, R4           If GLEN <= GPOS have more to search
 BRNH LOOP
 SLGRK   R1,R1,R3      Subtract off amount loaded
 ALGRK   R1,R1,R4      Add amount to the zero that was found
```

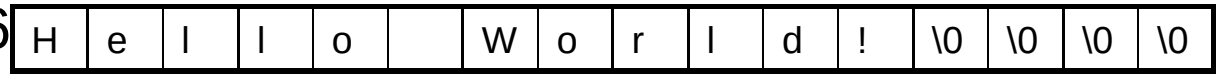
Memory at 0x6FF3 = STR@

Location not accessible




```
VLBB   V16,0(R1,STR@),6
```

V16




```
LCBB        R3,0(R1,STR@),6     R3 = 13
ALGRK       R1,R1,R3            R1 = 13
VFENEBZ     V17,V16,V16         V17=0x000000000000000C….
VLVGB       R4,V17,7(0)         R4=12
CGR         R3,R4
BRNH        LOOP
SLGRK       R1,R1,R3
ALGRK       R1,R1,R4
```

# Memory at 0x6FF6 = STR@

Page accessible

| H | e | l | l | o | | W | o | r | l | d | ! | \0 | ? | ? | ? |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

```
VLBB    V16,0(R1,STR@),6
```

V16

| H | e | l | l | o | | W | o | r | l | \0 | \0 | \0 | \0 | \0 | \0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

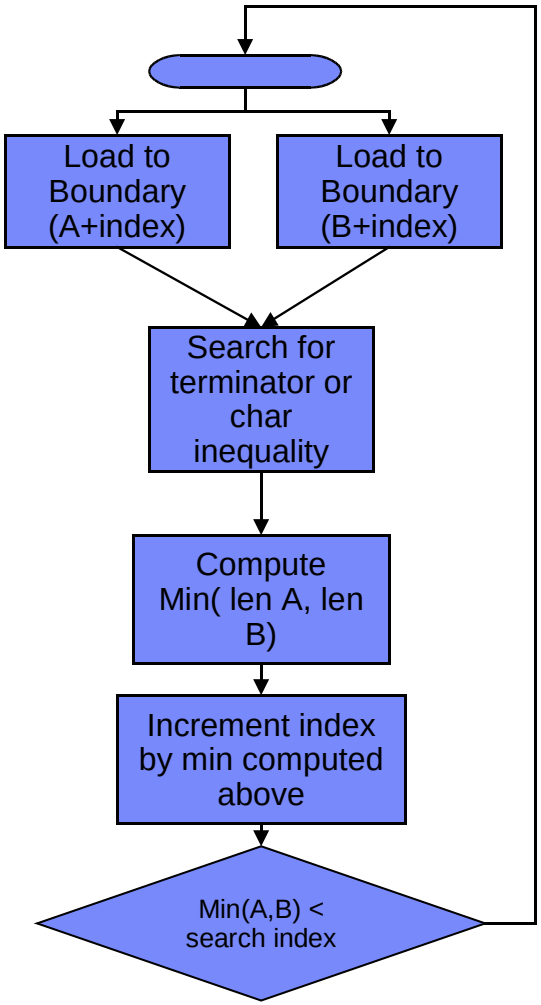
```
LCBB    R3,0(R1,STR@),6          R3 = 10
ALGRK   R1,R1,R3                 R1 = 10
VFENEBZ V17,V16,V16              V2=0x0000000000000000A….
VLVGB   R4,V17,7(0)              R4 = 10
CGR     R3,R4                    10 <= 10?
BRNH    LOOP                     Taken
VLBB    V16,0(R1,STR@),6
```

V16

| d | ! | \0 | \0 | \0 | \0 | \0 | \0 | \0 | \0 | \0 | \0 | \0 | \0 | \0 | \0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

```
LCBB    R3,0(R1,STR@),6          R3=16
ALGRK   R1,R1,R3                 R1=26
VFENEBZ V17,V16,V16
VLGBRI  R4,V17,7(0)              R4=2
CGR     R3,R4                    16 <= 2?
BRNH    LOOP                     Not Taken
SLGRK   R1,R1,R3                 R1=10
ALGRK   R1,R1,R4                 R1=12
```

# strcmp()

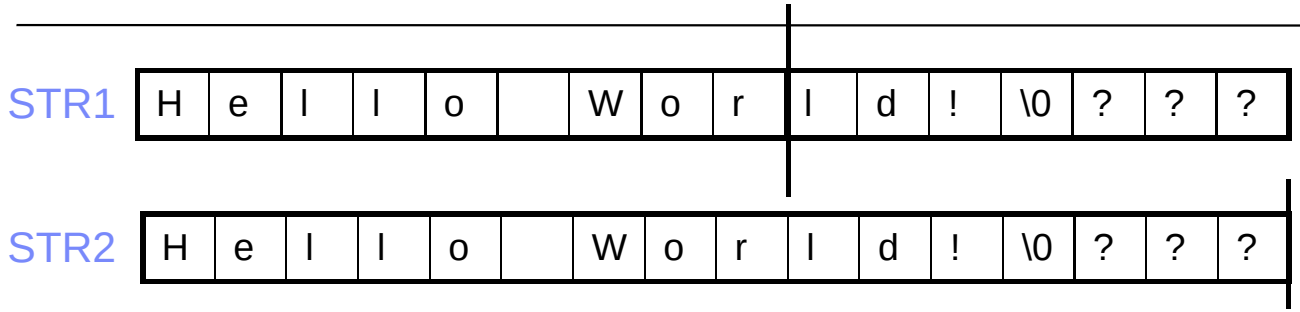


# int strcmp(const char* s1, const char* s2) using LBB

- R1 – STR1@, R2 – STR2@, R3 – Return value

```
  XGR  R4,R4

LOOP: VLBB    V16,0(R4,R1),6 Load s1
  VLBB V17,0(R4,R2),6 Load s2
  VFENEBZ V18,V16,V17    Compare strings
  VLVGB   R3,V18,7(0)    Extract index to gpr (16-no match)
  LCBB R5,0(R4,R1),6     Get load byte count
  LCBB R6,0(R4,R2),6
  CLGR R5,R6
  LOCGRH  R5,R6          Compute minimum of bytes loaded
  ALGRK   R4,R4,R5       Inc smallest load amt
  CLGR R5,R3             See if miscompare in portion loaded
  BRNH LOOP
*
  SLGRK   R4,R4,R5
  ALGRK   R4,R4,R3
  LB   R3,0(R4,R1)
  LB   R5,0(R4,R2)
  SRK  R3,R3,R5
```




```
LBB     V1,0(RX,STR1),4k
```

| H | e | l | l | o | | W | o | r | \0 | \0 | \0 | \0 | \0 | \0 | \0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

```
LBB     V2,0(RX,STR2),4k
```

| H | e | l | l | o | | W | o | r | l | d | ! | \0 | \0 | \0 | \0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

```
FENEBZ  V3,V1,V2
VLVGB   G3,V3, 7(0)     G3=9
LCBB    G1,0(RX,STR1),4k        G1=9
LCBB    G2,0(RX,STR2),4k        G2=16
CGR     G1,G2
LOCGR   G1,G2
AGR     RX,RX,G1        RX=9
CGR     G1,G3
BRLE    LOOP            ->TAKEN
LBB     V1,0(RX,STR1),4k
```

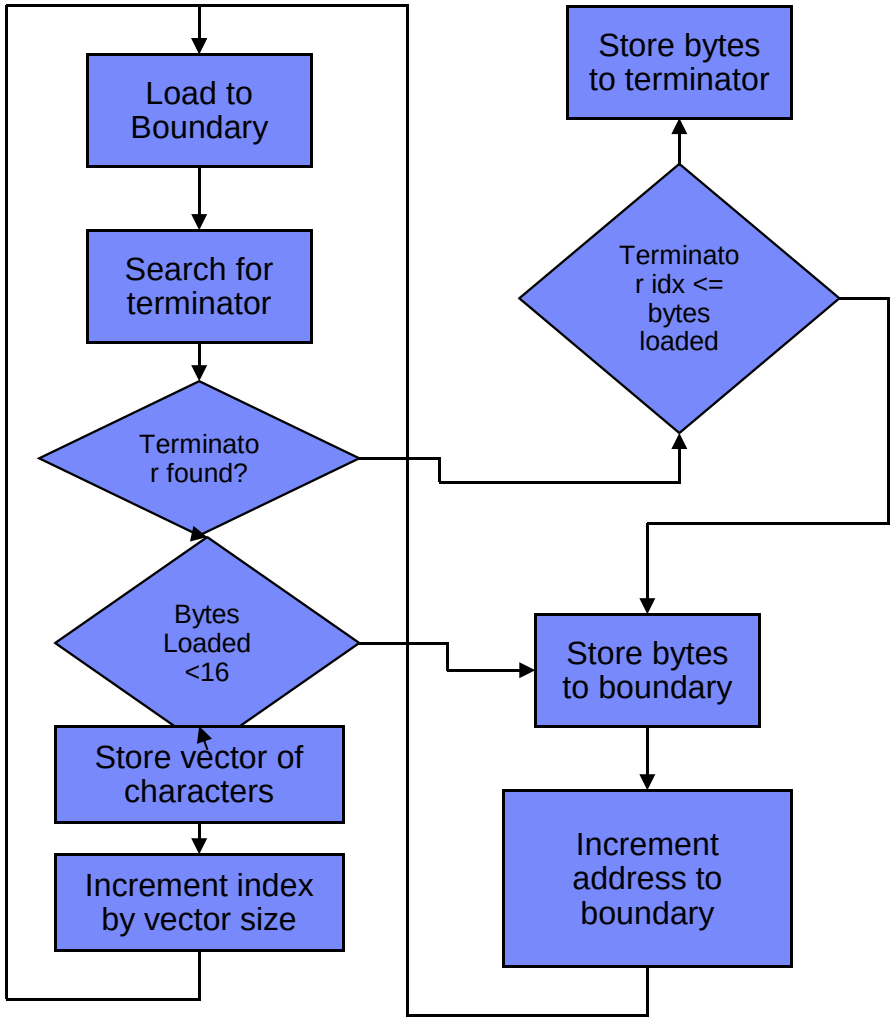
| l | d | ! | \0 | \0 | \0 | \0 | \0 | \0 | \0 | \0 | \0 | \0 | \0 | \0 | \0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

```
LBB     V2,0(RX,STR2),4k
```

| l | d | ! | \0 | \0 | \0 | \0 | \0 | \0 | \0 | \0 | \0 | \0 | \0 | \0 | \0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

```
FENEBZ  V3,V1,V2
VLVGB   G3,V3,7(0)      G3=3
LCBB    G1,0(RX,STR1),4k        G1=16
LCBB    G2,0(RX,STR2),4k        G2=16
…
BRLE    LOOP            -> Not Taken
…
```

# strcpy()



# strcpy()

- RB1 – STR1@, RB2 – STR2@

```
  XGR  RX,RX,RX
LOOP: LBB VSTR,0(RX,RB1)
  VFENEBZ  VPOS,VSTR,VSTR
  LCBB GLEN,0(RX,RB1)
  VLVGB    GPOS,VPOS,7(0) Extract index to gpr (16-no match)
  BRZ  EOS                EOS if 0 found
  CGHI GLEN,X'10'
  BRNE EOP                End of page, predicted NT
  VST  VSTR,0(RX,RB2)
  LA   RX,16(,RX)
  BRU  LOOP
EOP:  AGR RCUR,RX,RB2
  VSTL VSTR,0(GLEN,RCUR)  Store GLEN bytes
  AGR  RX,GLEN,RX
  BRU  LOOP               Start up on new page
EOS:  CMP GLEN,GPOS
  BRLE EOP
  AGR  RCUR,RX,RB2
  VSTL VSTR,0(GPOS,RCUR)  Store what's left
```

# strcpy() using LBB with LCBB setting cc

- R1 – dest@, R2 – src@

```
  XGRK R3,R3,R3
LOOP: VLBB    V16,0(R3,R2),6 Load in string 1
  LCBB R4,0(R3,R2),6          Find out how much was loaded
  BRNZ EOB                    Did we hit a 4k boundary
  VFENEBZSV17,V16,V16         Search for EOS
  BRZ  EOS                   Data Dependant
  VST  V16,0(R3,R1)           Write out 16-bytes of data
  AGHIK   R3,R3,16            Increment address by 16
  BRU  LOOP
EOB:  VFBNEZ  V17,V16,V16     Search for EOS
  VLVGB   R5,V17,7(0)         Extract index
  CLGR R5,R4                 GPOS<=GLEN -> EOS
  BRNH EOS2
  ALGRK   R5,R1,R3            Compute current Base Address
  VSTL V16,R4,0(R5)           Store out number of bytes to end of block
  ALGRK   R3,R4,R3            Increment index by number of bytes copied
  BRU  LOOP                  Start up on new page
EOS:  VLVGB   R5,V17,7(0)     Extract index to gpr (16-no match)
EOS2: ALGRK   R4,R1,R3
  VSTL V16,R5,0(R4)
```

# size_t strcspn(char* str, const char* stopset)

```
R1 = *str, R2 = *stopset, R0=return value

XGR   R3,R3
VZERO V16              zero out V16 (Uses VGBM)
VLBB  V17,0(R3,R2),6 Load stopset to 4k boundary
LCBB  R4,0(R3,R2),6
BRNZ  SPECIAL          Special case of page Xing. Predicted NT
VFENEBZS  V18,V17,V17 Look for Null
BRNZ  LONGSS           No string terminator, special case long stopset.
 Predicted NT
VLVGB R4,V18,7(0)     Extract index of first null
VREPB V18,V17,0(0)    Replicate first character across all characters
VLL   V17,R4,0(R2)    Load only up to null and zero out other elements
VCEQB V16,V16,V17     Create mask of null chars by comparing with 0
VSEL  V17,V17,V18,V16 Fill in null characters with first character
XGR   R3,R3           Reset string pointer to start
MORESTR:
VLBB  V16,0(R3,R1),6 Load str into V16 to 4k boundary
LCBB  R5,0(R3,R1),6  Get count of bytes loaded
VFAEBZ    V16,V16,V17,4  Search for stopset char or zero ZS=1, RT=1,
 CC=0
VLVGB R4,V16,7(0)    Extract index
ALGR  R3,R5          Increment Index
CLGR  R5,R4          See if miscompare before last byte loaded
BRNH  MORESTR
SLGR  R3,R5          Subtract off bytes loaded
ALGRK R0,R3,R4       add back in index to character found
```

# size_t strspn(char* str, const char* startset)

```
R1 = *str, R2 = *startset R0 = return value

XGR   R3,R3
VZERO V16              zero out V16
VLBB  V17,0(R3,R2),6   Load startset
LCBB  R4,0(R3,R2),6
BRNZ  SPECIAL          Special case of page Xing. Predicted NT
VFENEBZS  V18,V17,V17  Look for Null
BRNZ  LONGSS           No string terminator, special case long stopset.
                       Predicted NT
VLVGB R4,V18,7(0)      Extract index of first null
VREPB V18,V17,0(0)     Replicate first character across all characters
VLL   V17,R4,0(R2)     Load only up to null and zero out other elements
VCEQB V16,V16,V17      Create mask of null chars by comparing with 0
VSEL  V17,V17,V18,V16  Fill in null characters with first character
XGR   R3,R3            Reset string pointer to start
MORESTR:
VLBB  V16,0(R3,R1),6   Load str into V16
LCBB  R4,0(R3,R1),6    Get count of bytes loaded
VFAEBZV16,V16,V17,X'C' Search for first character not in startset or zero
                        IN=1,RT=1,ZS=1,CC=0
VLVGB R5,V16,7(0)      Extract index
ALGR  R3,R4            Add bytes loaded to index
CLGR  R4,R5
BRNH  MORESTR
SLGR  R3,R4            Subtract off bytes loaded
ALGRK R0,R3,R5         Add in bytes to a char not in startset
```

# char* strchr(char* s, int c)

- R2 - @ of string, R3 = *c, R1 will ptr to char

```
  XGRK R1,R1,R1        Zero out running index
  VLREP   V17,0(R3)
LOOP: VLBB    V16,0(R1,R2),6 Load up to 16 bytes
  LCBB R3,0(R1,R2),6  Find how many bytes were loaded
  ALGRK   R1,R1,R3    Increment length by bytes loaded
  VFEEEBZ V18,V16,V17 Look for 0 byte or character
  VLVGB   R4,V18,7(R0)   Extract index to gpr (16-no match)
  CLGR R3, R4      If GLEN <= GPOS have more to search
  BRNH LOOP
  SLGRK   R1,R1,R3    Subtract off amount loaded
  ALGRK   R1,R1,R4    Add amount to the zero that was found
  ALGRK   R2,R2,R1    Add offset to char to pointer
```

# bool isAlpha(char* str)  with EBCDIC characters

```
R1 = *str

XGR   R2,R2
XGR   R0,R0             Return value = 0 (false)
VLV16,RANGECHAR         Load range characters
VLV17,RANGECTL          Load Range Controls
MORESTR:
VLBB  V18,0(R2,R1),6    Load string
LCBB  R2,0(R2,R1),6
VSTRCZB   V19,V18,V16,V17,X'4'   Search ranges and for zero. RT=1
                                 (index), IN=0
VLVGB R3,V19,7(0)
ALGR  R2,R3
CLGR  R2,R3
BRNH  MORESTR         No string terminator, no non-matching characters
VFENEZBS  V19,V18,V18
LOCHIZ    R0,1           Set Return Value to True

RANGECHAR:
DC    X'81899199a2a9c1c9d1d9e2e900000000'
RANGECTL:
DC    X'a0c0a0c0a0c0a0c0a0c0a0c000000000' GE,LE
```

# Java String.compareTo(String str) (equiv to strcmp)

- R1 – STR1@, R2 – STR1_Leng R3 – STR2@, R4 – STR2_LENG R0 – Return value

```
 CGR R2,R4            Check to see if strings are same length
 BNE MISCMP
 XGR R0,R0
 SLLG   R2,R2,1(0)    Convert Halfword unicode characters to
                       bytes
LOOP:   VLL V16,R2,0(R1) Load str1
 VLL V17,R2,0(R3)     Load str2 (Same length for both)
 VFENEHSV18,V16,V17   Compare strings
 BRC 6,MISCMP
 SLGFI  R2,X'10'      Subtract 16 for next load length
 BRNM   LOOP
 BRU    DONE
MISCMP: LOCGHIL   R0,-1
 LOCGHIHR0,1
DONE:
```

# Java String.indexOf(char)

- R1 – STR1@, R2 – STR1_Leng R3 – STR2@, R4 – STR2_LENG R0 – Return value

```
 XGR R0,R0
 SLLG   R2,R2,1(0)    Convert Halfword unicode characters to
                       bytes
LOOP:   VLL V16,R2,0(R1) Load str1
 VLL V17,R2,0(R3)     Load str2 (Same length for both)
 VFENEHS V18,V16,V17  Compare strings
 BRC 6,MISCMP
 SLGFI  R2,X'10'      Subtract 16 for next load length
 BRNM   LOOP
 BRU    DONE
MISCMP: LOCGHIL   R0,-1
 LOCGHIH R0,1
DONE:
```

## Java String(String original)  (strcpy())

- R0 – dest@, R2 – src@, R1 – src_leng

```
SLLG    R1,R1,1(0)
LGR     R3,R1
MVCL    R0,R1
```

# void toUpper(char* s)

```
R1 = *str

   XGR  R2,R2
   XGR  R0,R0            Return value = 0 (false)
   VL   V16,RANGECHAR    Load range characters
   VL   V17,RANGECTL     Load Range Controls
   VREPIB   V20,X'40'    Create constant to add to make upper
MORESTR:   VLBB V18,0(R2,R1),6   Load string
   VSTRCZB  V19,V18,V16,V17,X'4'  Search for lower case and for zero. RT=1 (bit-
 vector), IN=0
   VAB  V21,V18,V20      Add in offset to make characters uppercase
   VSEL V21,V21,V18,V19  Replace lowercase characters with uppercase computed
 above
   VFENEZBS V22,V18,V18  Find an EOS  (could be moved to special case if VSTRC
                         had different CC settings)
   BRZ  EOS
   LCBB  R3,0(R2,R1),6
   BRNZ  EOB
   VST  V21,0(R2,R1)     Store out full 16 bytes of updated chars
   ALGHI R2,X'10'
   BRU  MORESTR          No string terminator or block crossing
EOB:   VLVGB    R4,v22,7(0)  Extract index
   CGR  R4,R3
   BRNH EOS2:
   VSTL  v22,R3,(R2,R1)
   ALGR  R2,R3
   BRU  MORESTR
EOS:   VLVGB    R4,V22,7(0)  Extract length
EOS2:  VSTL v22,R4,(R2,R1)
RANGECHAR:
   DC   X'81899199a2a900000000000000000000'
RANGECTL:
   DC   X'a0c0a0c0a0c000000000000000000000' GE,LE
```

# C string library

function

## strcspn

<cstring>

```
size_t strcspn ( const char * str1, const char * str2 );
```

**Get span until character in string**

Scans *str1* for the first occurrence of any of the characters that are part of *str2*, returning the number of characters of *str1* read before this first occurrence.

The search includes the terminating null-characters, so the function will return the length of *str1* if none of the characters of *str2* are found in *str1*.

# size_t strcspn(char* str, const char* stopset)

```
R1 = *str, R2 = *stopset, R0=return value

XGR    R3,R3
VZERO  V16      zero out V16 (Uses VGBM)
VLBB   V17,0(R3,R2),6  Load stopset to 4k boundary
LCBB   R4,0(R3,R2),6
BRNZ   SPECIAL      Special case of page Xing. Predicted NT
VFENEBZS V18,V17,V17   Look for Null
BRNZ   LONGSS       No string terminator, special case long stopset.
                    Predicted NT
VLVGB  R4,V18,7(0) Extract index of first null
VREPB  V18,V17,0(0)    Replicate first character across all characters
VLL    V17,R4,0(R2)    Load only up to null and zero out other elements
VCEQB  V16,V16,V17 Create mask of null chars by comparing with 0
VSEL   V17,V17,V18,V16 Fill in null characters with first character
XGR    R3,R3       Reset string pointer to start
MORESTR:
VLBB   V16,0(R3,R1),6  Load str into V16 to 4k boundary
LCBB   R5,0(R3,R1),6   Get count of bytes loaded
VFAEBZ V16,V16,V17,4   Search for stopset char or zero ZS=1, RT=1, CC=0
VLVGB  R4,V16,7(0) Extract index
ALGR   R3,R5       Increment Index
CLGR   R5,R4       See if miscompare before last byte loaded
BRNH   MORESTR
SLGR   R3,R5       Subtract off bytes loaded
ALGRK  R0,R3,R4    add back in index to character found
```

- In this case you could now replace everything from the VFENEBZS to the XGR with simply:

- VISTR  V17,V17
- BRNZ  LONGSS

- It saves you from having to do an extra compare to see if you are at the end of a string or not.

# Related Sessions

- Session 16618: What's New in Enterprise PL/I and C/C++

# Questions?