z/Architects Gone Wild
Unusual Assembler Instructions

Jan Samohýl
CA Technologies

8/12/2011
Session 09748

Instructions Covered

- Checksum
- Update Tree
- Compare and Form Codeword
- Perform Locked Operation
- Load Pair Disjoint
- Population Count
- Store Clock Fast
- Monitor Call
- Translate One/Two to One/Two
- Search String Unicode
- Convert Unicode

Checksum

- Format: CKSM R_1,R_2

- Compute 32-bit checksum – unsigned “sum” of consecutive words in memory modulo 23^2
- The sum is calculated in one’s complement representation, i.e. negative numbers are simply negated
  - Contrast this with the standard two’s complement representation used on the architecture (negative number is negated +1)
  - To simulate one’s complement addition, after each word is added as usual, the carry of result is added to it

Checksum – continued

- The memory area is specified by register pair
  - R_2 contains the address (depends on addr. mode)
  - R_2 +1 contains the length (32 or 64 bit unsigned integer depending on addr. mode)
  - If length is not divisible by 4, the area is padded with 0 bytes
  - R_2, R_2 +1 are modified during the instruction
- The unsigned 32-bit result is added to low word of R_1 (so you need to clear it at the beginning)
- Returns CC=0 on success, CC=3 if interrupted

Checksum example

- Computing 16-bit TCP/IP checksum:

```
XR      R0,R0     R2,R3 -> 'HELLO WORLD'
CKSM    R0,R2     R0 = x'78DA7EAB', R3 = 0
LR      R2,R0     R2 = x'78DA7EAB'
SRDL    R2,16     x'000078DA7EAB0000'
ALR     R2,R3     R2 = x'7EAB78DA'
ALR     R2,R0     R2 = x'F785F785'
SRL     R2,16     R2 = x'0000F785'
```

- Another possible use would be to compute entry into hash table with prime size

Update Tree – introduction

- The purpose of Update Tree (UPT) instruction is to help implement a single algorithm, **multi-way merge sorting**
- The purpose of Compare and Form Codeword (CFC) instruction is to allow having keys of any length in this algorithm
- I will first explain the algorithm, and then how the instructions help in the implementation

Multiway Merge Sorting

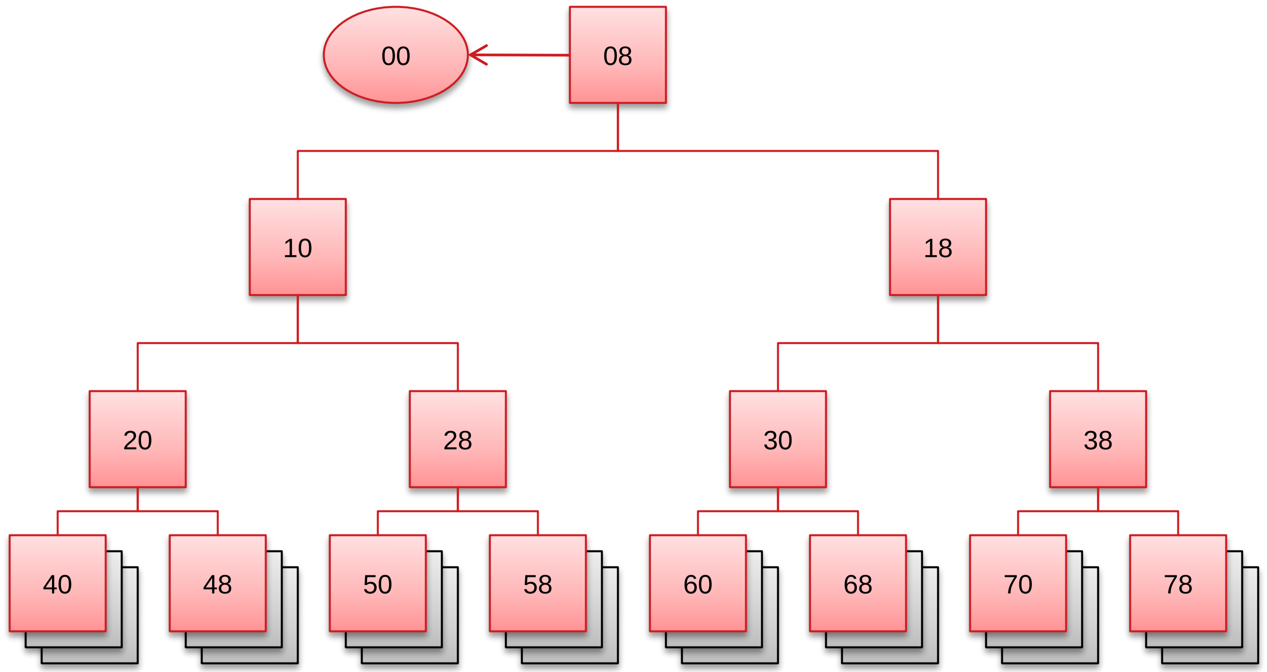
- We have several sorted input sequences, and we merge them into a single sequence, which will be sorted

$$\begin{array}{l} 73, 51, 48, 31, 16 \\ 60, 37, 33, 32, 12 \\ 68, 63, 57, 34, 19 \end{array} \Rightarrow 73, 68, 63, 60, 57, 51, 48, 37, 34, 33, 32, \ldots$$

- If we start from sequences of length one, iteratively, we sort any sequence (in descending order)
- To merge the sequences, we pick the largest among the first elements in the sequences
- We use a special tree – see next slide for layout
  - Node of the tree will have 4 byte key and 4 byte value

Merge Tree Layout

- Example of tree node addresses (hex) for 8 sequences
- 7 internal nodes of the tree, in memory stored by rows, dummy root node first, then the root node
- Parent node address = (Node address / 16) * 8

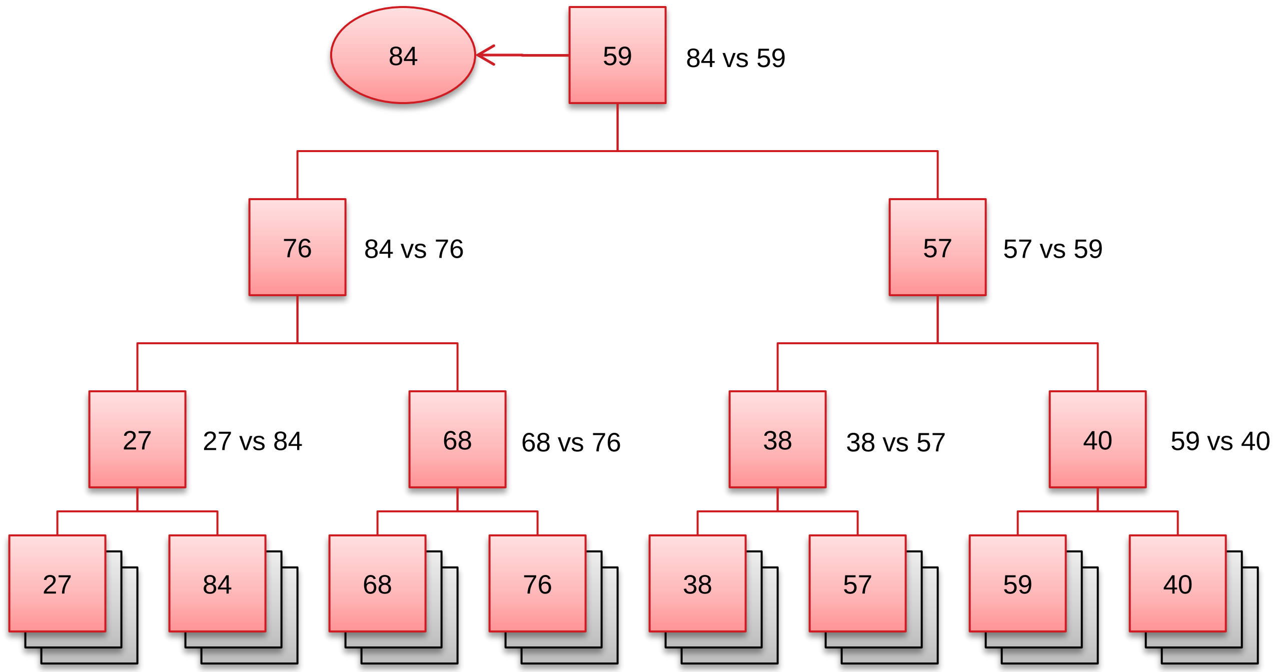


SHARE in Orlando 2011

Priming the Merging Tree

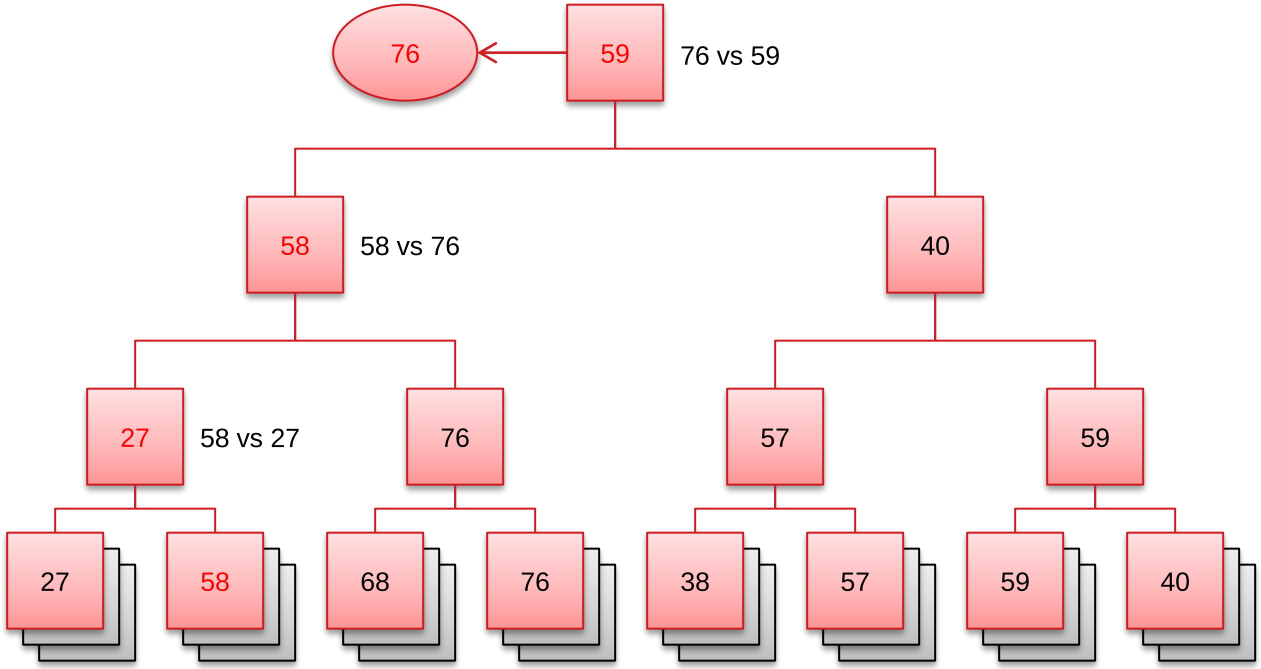
- Assume leaf nodes of the tree contain the first values in the sequences to be merged
- It’s like a tournament - we go from the bottom up, at each internal node we compare the two keys, we store the smaller key (“loser”) and value in the node, the larger key (“winner”) and value continues to the higher row
- So at the top (dummy node) the largest key will emerge, and we put the 2nd largest to the root node
- The update operation (see next) can be used to prime the tree too (put lowest possible values in internal nodes and add leaf nodes with UPT)

Priming the Merging Tree



Updating the Merging Tree

- Take the next element from the sequence where the previous winner originated
- Update elements on the path from the new element to the top by storing the “loser” and letting winner to go the top



Update Tree

- Format: UPT

- Uses registers 0-5 as follows:
  - R0: Key portion of the current “winner”
  - R1: Value portion of the current “winner”
  - R2 & R3: Node contents if key was equal
  - R4: Address of the tree (the dummy node above root)
  - R5: Index of the start (or current) node
- The tree is a structure described above:
  - Each node has 8 bytes, 4 byte key and 4 byte value
  - Address R4 and index R5 must be doubleword aligned
- These regs. are modified during the operation

Update Tree operation

- Move to parent node by setting R5 := (R5 / 16) * 8
  - If R5=0, finish with CC=1
  - If R0 negative, finish with CC=3
- Current node address = R4 + R5
- Compare R0 to the key in the current node
  - If equal, store in R2 and R3 the key and value in the current node, finish with CC=0
  - If R0 lower, swap R0 and key in the node and swap R1 and value in the node
- Repeat this process

Codeword

- To extend the previous algorithm with keys of any (fixed) size, we use codewords
    - The construction depends whether we sort in ascending or descending order
- Codeword is 4-byte value, formed as a result of lexicographic comparison of two keys (halfword-wise):
    - Higher halfword is the index to the uncompared key portions
    - Lower halfword is a
        - first lexicographically different halfword of the lower operand (to merge in descending order)
        - complement of the first lexicographically different halfword of the higher operand (to merge in ascending order)

Codeword example

• Low key	x’1122 3344 5566’
• High key	x’1122 3344 5577’
• Codeword for descending order	x’00065566’
• Codeword for ascending order	x’0006AA99’

• Low key	x’00DE ADBA BE34’
• High key	x’00DE ADBE EF36’
• Codeword for descending order	x’0004ADBA’
• Codeword for ascending order	x’00045241’

Compare and Form Codeword

- Format: CFC $D_2(B_2)$

- Compares two memory areas by consecutive halfwords
- Base addresses of the areas are in R1 and R3 (both halfword aligned)
- R2 is used as index to the halfword being compared
- Index limit is low 15 bits of effective address $D_2(B_2)$, it defines the length of the memory areas
- Lowest bit of effective address $D_2(B_2)$ is operation control
  - Determines if we are constructing codeword for ascending or descending sort
  - Assume 0, i.e. ascending, for the explanation

Compare and Form Codeword

- If R2 is larger than index limit, CC=0 and finish
- Compare halfwords at addresses R1+R2 and R3+R2
- Increment R2 by 2 (index to next halfword)
- If halfwords are equal, repeat from the beginning
- Shift R2 left by 16 bits (to make room)
- If 1st halfword is lower than 2nd, put complement of the 2nd halfword into low halfword of R2, CC=1
- If 1st halfword is higher than 2nd, put complement of the 1st halfword into low halfword of R2, swap R1 and R3, CC=2
- Resulting R2 is the "codeword", R1 points to lower key

Using codewords

- In the above tree update operations, we can replace the tree keys with codewords resulting from the compare between the “loser” and “winner”
- Now consider:
  - All the codewords along the path from where the previous winner originated to the top of the tree were created by comparison to this previous winner
  - The codeword for the new element is created by comparing it to the previous winner, and we introduce the new element to the node where the previous winner originated
- So, during the update operation, there are 3 keys in play: key of the last winner **W**, key of the new element **N** and key of the node we currently process **C**

Using codewords during Update Tree

- We know W <= C and W <= N (that’s why W is the winner)
- At each node, we compare codeword C_{WC} between W and C and codeword C_{WN} between W and N, 3 possibilites:
  - N differs sooner from W and C (because N > C), then the first halfword of C_{WC} is higher than first halfword of C_{WN}; so C wins over N
  - N differs at the same halfword as W and C; then the second halfwords of C_{WC} and C_{WN} determine the correct winner
  - N differs later from W and C (i.e. shares longer equal prefix with C than with W); then $C_{WC} = C_{WN}$ and update stops; we use CFC to compare C to N and store codeword C_{CN} into the node and continue
- During update, at each node the new key N can be different!

Perform Locked Operation

- Format: PLO $R_1,D_2(B_2),R_3,D_4(B_4)$

- Acquires a lock specified by address in R_1, then performs operation specified in R0, and releases the lock
- The operations possible are various “compare and swaps”, “compare and loads”, “compare and swap and stores” (various sizes etc.)
  - 14 pages of description in Principles of Operation

Never Perform Locked Operation

- I would not recommend using this instruction at all
- It has a fatal flaw – the lock
  - The lock is not synchronized with the other instructions of the same name (such as Compare and Swap), so you cannot combine them
  - And acquiring the lock is almost same as doing CS
- PLO also requires complex parameter lists for most operations
- Use the CS, CDS or CSST instructions instead

Load Pair Disjoint

- Format: LPD R_3,D_1(B_1),D_2(B_2)
- Register R_3 is loaded from address D_1(B_1), and register R_3+1 is loaded from address D_2(B_2), **interlocked**
  - R_3 specifies an even-odd register (32-bit) pair
  - The access is usually interlocked, i.e. no other processor can modify one of the locations fetched – if access was interlocked, CC=0 is set, if not, CC=3 is set
  - This is new instruction on z196
- There is also LPDG, which operates with 64-bit registers
- Use case: test a lock and retrieve a value at the same time

Population Count

- Format: POPCNT R_1,R_2

- Count the number of bits in each of the 8 bytes of (64-bit) register R_2 and place the count into the corresponding byte of R_1
  - Sets CC=0 if result is all 0, CC=1 otherwise
  - This is new instruction on z196
- Sadly, this instruction obsoleted many elegant (or not) algorithms to do this by hand
- Useful for operations with bitmaps (such as computing size of a set given by bitmap)

Population Count example

- To compute total number of bits of R15 = x'FEDCBA9876543210' into R8, do this (uses R9 as a work register):

```
POPCNT      R8,R15        R8=x'0705050305030301'
AHHLR       R8,R8,R8      R8=x'0C08080405030301'
SLLG        R9,R8,16      R9=x'0804050303010000'
ALGR        R8,R9         R8=x'140C0D0708040301'
SLLG        R9,R8,8       R9=x'0C0D070804030100'
ALGR        R8,R9         R8=x'2019140F0C070401'
SRLG        R8,R8,56      R8=x'0000000000000020'
```

Store Clock Fast

- Format: STCKF $D_2(B_2)$

- Like STCK (stores internal 8-byte TOD clock at the specified address), but “fast”
- Fast means, in this context, that the other processors are not serialized.
  - With STCKF, it can happen that you get earlier time on processor where this instruction executed demonstrably later.
  - With STCK, causality of the universe is preserved.
- To be future-proof use STCKE, which is also serialized
  - And uses 128-bit extended TOD clock

Monitor Call

- Format: MC $D_1(B_1),I_2$
- This instruction will invoke monitor event program interrupt (code 0x40)
  - During interrupt, the effective 64-bit address $D_1(B_1)$ (monitor code) is stored at real location 176, and the byte of immediate value I_2 (monitor class) is stored at real location 149
  - The interrupt can be masked with 16 monitor-mask bits in CR8, which correspond to value of low 4 bits in monitor class (0 in mask bit means MC will be no-op)

Monitor Call extended

- On z196, this instruction can do hardware counting instead of an interrupt
  - If the bit of 16-bit enhanced-monitor-mask in CR8 corresponding to monitor class is 1, MC won't cause interrupt, but will instead increment a counter in the table in memory
- The index to the counter in the table is the monitor code
- The table address and number of entries are described at real address 264
- Effectively, the counters are 48-bit (but the organization of the table is a bit unusual)

Translate One/Two to One/Two (TROO, TROT, TRTO, TRTT)

- Format: TRxx $R_1,R_2[,M_3]$
- These 4 ultimate translation instructions:
  - Read single or double byte characters from source location
  - Translate them according to table to single or double byte characters, prematurely ending if a specified testing character after translation is encountered
  - Store the translated characters at another location
- This is an extended version of extended versions of TR and TRT

Translate One/Two to One/Two usage

- R_1 specifies address of the source data
- R_1+1 specifies length of the source data (up to 2GB in AMODE 31)
- R_2 specifies address of the destination (where to store the translation)
- R0 specifies testing character (low byte for TROO and TRTO, low halfword for TRTO and TRTT)
- R1 specifies address of translation table (must be either 4K page-boundary or doubleword aligned, depends on processor)
- Depending on processor, if M_3=1, then testing character is not used

Translate One/Two to One/Two usage

- Table of translation tables:

Instruction	Source character size	Destination & testing character size	Translation table size
TROO	1	1	256
TROT	1	2	512
TRTO	2	1	65536
TRTT	2	2	131072

- As usual for complex instructions, these change the contents of the registers as one would expect
- Sets CC=0 if entire operand was processed, CC=1 if equal testing character was encountered, CC=3 if not entire operand processed (can just restart in that case to finish)

Search String Unicode

- Format: SRSTU R_1,R_2
- Search a specified double-byte character in a given memory area
  - The memory area starts at address in R_2 and ends at address R_1 (R_1 points to first byte behind it)
  - The character is in low halfword of R0, high halfword 0
- Returns:
  - CC=1 if the character was found, its address in R_1
  - CC=2 if the character was not found
  - CC=3 if string wasn't fully searched, R_2 is updated with the new address, so you can just restart the instruction

Quick Field Guide to Unicode

- Character values from 0 – 10FFFF
- Almost all languages, ~110 000 assigned characters
- UTF-32 encoding: 4 bytes per character, direct value (best for internal use, not storage)
- UTF-16 encoding: mostly 2 bytes per character, codes above 10000 are represented as pair of values D800-DBFF and DC00-DFFF (surrogate pair)
- UTF-8 encoding: 1-4 bytes per character (higher values are longer), the 1-byte codes are same as 7-bit ASCII, it’s possible to find the start of the character from random byte in the stream

Convert Unicode

- Format: CUnn $R_1,R_2[,M_3]$
- Convert from Unicode encoding to Unicode encoding - between UTF-8, UTF-16, UTF-32
  - Processes characters from the source field and copies them to destination field
  - Both R_1 and R_2 denote a register pair; each pair determines starting address and length of area in storage, R_2 source and R_1 destination (similar to MVCL, except length is 32-bit)
  - Some variants have optional M3 parameter, which, if set to 1, will cause well-formedness checking of the source encoding
- z/OS has CSRUNIC macro, which can apparently call some of these instructions

Convert Unicode variations

Instruction	Source encoding	Destination encoding	Well-formedness checking
CU12 (CUTFU)	UTF-8	UTF-16	Yes
CU14	UTF-8	UTF-32	Yes
CU21 (CUUTF)	UTF-16	UTF-8	Yes
CU24	UTF-16	UTF-32	Yes
CU41	UTF-32	UTF-8	No
CU42	UTF-32	UTF-16	No

CC=0 if the entire source operand was converted
CC=1 if end of destination operand was reached
CC=2 if either well-formedness check fails or character is invalid
CC=3 if not entire source operand was processed yet (can be restarted)
As usual, the registers are updated with where the operation was at the time

Find Leftmost One (bonus)

- Format: FLOGR R_1,R_2
- Find the bit position of leftmost one bit in R_2 (64-bit) and store into register pair R_1, R_1+1:
  - If one bit is found in R_2 (R_2 nonzero)
    - R_1 will get its index, as 64-bit number, counting from 0 from left
    - R_1+1 will get contents of R_2 with the found one bit set to zero
    - CC = 2
  - If no one bit is found in R_2 (R_2 is zero)
    - R_1 = 64, R_1+1 = 0, CC = 0
- Again, this is useful for operations with bitmaps

The End

- Reference: z/Architecture Principles of Operation, 9th ed.
- Thanks for your attention
- Questions?