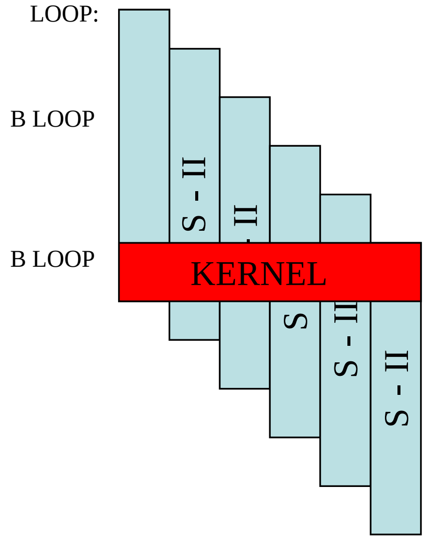
C64x+ Code savings

- SPLOOP Buffer
- Compact instructions
- C64x/C64x+ specifics

SPLOOP - Overview



How It Works:

1) **Loop pipeline copied to buffer in CPU core**
2) **SPLOOP Instruction provided with loop count and variables**
3) **Loop pipeline is executed out of SPLOOP buffer**

Fewer Accesses to Memory

- Saves Power
- Improves Performance

Reduces Code Size

- Up to 30% for loop based kernels
- More memory available for data

SPLOOP is fully Interruptible

- Improves performance
- also valid for multiple-assignment

SPLOOP – Terminology

A stage boundary is reached every ii cycles. The following terminology is used to describe specific stage boundaries.

First loading stage boundary: The first stage boundary after the **SPLOOP(D/W)** instruction.

Last loading stage boundary: The first stage boundary that occurs in parallel with or after the **SPKERNEL** instruction.

First kernel stage boundary: The same as the last loading stage boundary.

Last kernel stage boundary: The last stage boundary before the loop is only executing epilog instructions.

SPLOOP – Basic Example

```
for (i=0; i<val; i++)  {
        dest[i]=source[i];
        }
```

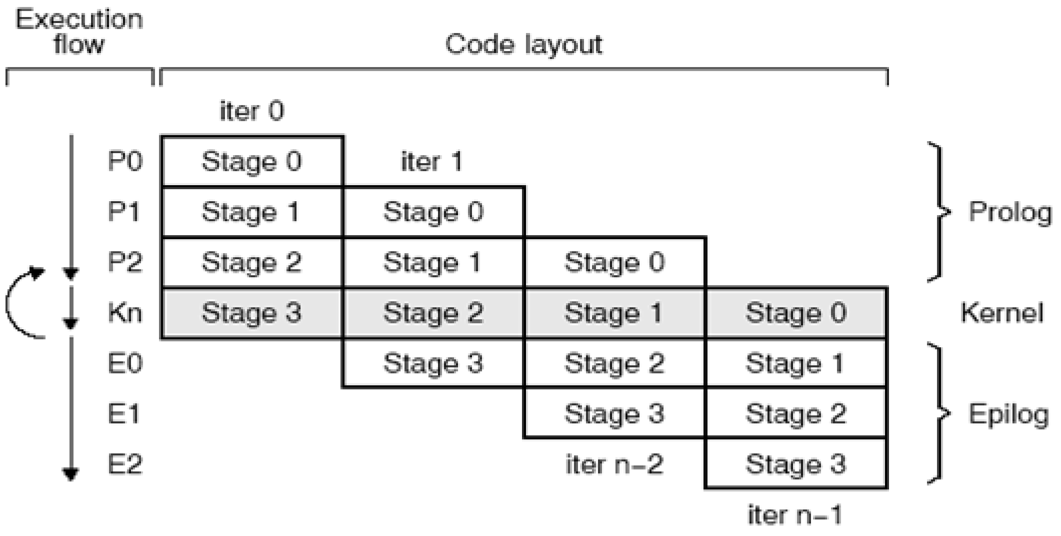
C-Code Copy Loop

```
        MVC           8,ILC          ;Do 8 loops
        NOP           3              ;4 cycle for ILC to load
        SPLOOP        1              ;Iteration interval is 1
        LDW           *A1++,A2       ;Load source
        NOP     4                    ;Wait for source to load
        MV      .L1X  A2,B2          ;Position data for write
        SPKERNEL      3,0            ;End loop and store value
||      STW           B2,*B0++
```

SPLOOP implementation of Copy Loop

SPLOOP instruction flow of example

- Instructions loaded into loop buffer after SPLOOP command is encountered
- SPKERNEL indicates that the loop is finished loading
- Loop pipeline is executed out of SPLOOP buffer



SPLOOP – Example Vector Sum

```
C64x
_vecsum_c64:
              MVKL    .S2     cData,       B_cData        ; output pointer
              MVKL    .S2     bData,       B_bData        ;bData pointer
              MVKH    .S2     cData,       B_cData        ;
      ||      MVKL    .S1     aData,       A_aData        ;aData pointer

              B       .S2     loop_vs                     ;prolog start
      ||      MVK     .S1     nData,       A_count        ;LOOP counter

             SHRU    .S1     A_count,     3,      A_count ;set up loop count
      ||      MVK     .L2     1,           B_pr         ;prolog count enable
      ||      MVKH    .S2     bData,       B_bData

              ADD     .D1X    B_cData,     8,      A_cData ;create off pointer
      ||      SUB     .L1     A_count,     1,      A_count ;initialize loop
      count
      ||      SHL     .S2     B_pr,        30,     B_pr    ;set up prolog
      counter
      ||      MVKH    .S1     aData,       A_aData
      ****************************** PIPE LOOP KERNEL
loop_vs:
              BDEC    .S1     loop_vs,     A_count          ;branch
      ||      LDDW    .D2T2   *B_bData++,  B_d32:B_d10      ;load b i+0-3
      ||      LDDW    .D1T1   *A_aData++,  A_d32:A_d10      ;load a i+0-3
      ||      ADD2    .S2X    A_d54,       B_d54, B_s54     ;sum i+5, sum i+4
      ||      ADD2    .L1X    A_d10,       B_d10, A_s10     ;sum i+1, sum i+0

              LDDW    .D2T2   *B_bData++, B_d76:B_d54    ;load b i+4-7
      ||      LDDW    .D1T1   *A_aData++, A_d76:A_d54    ;load a i+4-7
      ||      ADD2    .L2X    B_d76,       A_d76, B_s76   ;sum i+6, sum i+7

              ADD2    .L1X    A_d32,       B_d32, A_s32  ;sum i+2, sum i+3
      ||      ADD     .S2     B_pr,        B_pr,  B_pr    ;1st stage pro
      ||[!B_pr]STDW   .D1T2   B_s76:B_s54,*A_cData++[2] ;store sums i+4-7
      ||[!B_pr]STDW   .D2T1   A_s32:A_s10,*B_cData++[2] ;store sums i+0-3
```

- C64x+

```
          MVK     .S2     nData/8-2,       B_count     ;LOOP counter

    SPLOOPD 3
   ||      MVC     .S2     B_count, ILC
   ||      ADDAB   .D1     DP, aData,        A_aData
   ||      ADDAB   .D2     DP, bData,        B_bData

           LDDW    .D2T2   *B_bData++, B_d32:B_d10      ;load b i+0-3
   ||      LDDW    .D1T1   *A_aData++, A_d32:A_d10      ;load a i+0-3

           LDDW    .D2T2   *B_bData++, B_d76:B_d54     ;load b i+4-7
   ||      LDDW    .D1T1   *A_aData++, A_d76:A_d54      ;load a i+4-7
```

Stage boundary

```
    SPMASK
   ||^     ADDAB    .D2    DP, cData,    B_cData
   ||^     ADDAB    .D1    DP, cData+8,  A_cData

           NOP     2
```

```
           ADD2    .L1X    A_d32,       B_d32, A_s32  ;sum i+2, sum i+3

           ADD2    .S2X    A_d54,       B_d54, B_s54  ;sum i+5, sum i+4
   ||      ADD2    .L1X    A_d10,       B_d10, A_s10  ;sum i+1,sum i+0

           ADD2    .L2X    B_d76,       A_d76, B_s76     ;sum i+6, sum i+7
```

```
    SPKERNEL 2, 0
   ||      STDW   .D1T2  B_s76:B_s54,*A_cData++[2]  ;store sums i+4-7
   ||      STDW   .D2T1  A_s32:A_s10, *B_cData++[2]  ;store sums i+0-3
```

- 24% fewer instructions
  - Elimination of instructions priming the loop that get re-embedded in kernel
- Code is sequential for one iteration of loop
- The loop kernel is automatically piped up with all instructions that have been executed since the SPLOOP instruction

SPLOOP - Architecture

SPLOOP Hardware Support

The basic hardware support for the SPLOOP operation is:

- Loop buffer
- Loop buffer count register (LBC)
- Inner loop count register (ILC)
- Reload inner loop count register (RILC)
- Task state register (TSR)
- Interrupt task state register (ITSR)

Instructions

SPLOOP, SPLOOPD and SPLOOPW – invoke the loop buffer mechanism
• each clear the loop buffer count register (LBC), load the iteration interval (ii) and start the LBC counting.
• must be the first instruction of the execute packet

SPKERNEL and SPKERNELR – mark the end of the software pipelined loop

SPMASK and SPMASKR – prevent instruction from being loaded into the loop buffer or block execution of instructions already in the loop buffer

NOTE: SP instructions don't consume execution units.

SPLOOP – Piping the loop buffer

SPLOOP 3

```
             instr.D1
||           instr.S1          cycle 1

             instr.D1
||           instr.L2
||           instr.M1          cycle 2

             instr.S2
||           instr.M2          cycle 3

             instr.S2
||           instr.M1
||           instr.M2          cycle 4

             instr.S1
||           instr.L1
||           instr.M2
||           instr.D2          cycle 5

             instr.L1
||           instr.M1
||           instr.D1
|| SPKERNEL                    cycle 6
```

Execute

Execute

Execute

3 execute packets

KERNEL table

S1	S2	L1	L2	M1	M2	D1	D2
S1	S2	L1	L2	M1	M2	D1	D2
S1	S2	L1	L2	M1	M2	D1	D2

ILC = [illegible]

LBC = [illegible]/0

[illegible]

"SPLOOP body length" of 6

"Instructions of each stage populate KERNEL table with different color"

SPLOOPD – Differences

The unconditional SPLOOPD is used when the loop is known to execute for a minimum number of loop iterations. The required minimum of number of iterations is a function of the II as shown below:

II	Minimum Number of Loop Iterations
1	4
2	2
3	2
>=4	1

When using SPLOOPD the ILC register must be loaded with a value that is biased to compensate for the required minimum number of loop iterations.

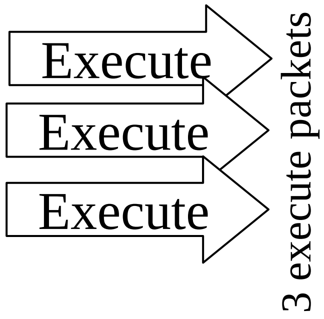
Differences SPLOOP vs. SPLOOPD:

- Initial termination condition test is always false;
- ILC decrement is disabled for the first 3 cycles
- The loop must execute at least 1 iteration.
- Stage boundary termination condition is forced to false
- Loop cannot be interrupted for the first 3 cycles of the loop.

SPLOOPD

SPLOOPD 3

```
              instr.D1
||            instr.S1        cycle 1

              instr.D1
||            instr.L2
||            instr.M1        cycle 2

              instr.S2
||            instr.M2        cycle 3

              instr.S2
||            instr.M1
||            instr.M2        cycle 4

              instr.S1
||            instr.L1
||            instr.M2
||            instr.D2        cycle 5

              instr.L1
||            instr.M1
||            instr.D1
|| SPKERNEL                   cycle 6
```



KERNEL table

S1	S2	L1	L2	M1	M2	D1	D2
S1	S2	L1	L2	M1	M2	D1	D2
S1	S2	L1	L2	M1	M2	D1	D2

ILC = [illegible]   LBC = [illegible]/0

Execute kernel
Drain kernel (epilog)
[illegible] loop buffer (prolog)

5 iterations with SPLOOPD needs an adjustment of the ILC
II = 3 means ILC value needs to be 3 (5 minus 2).

The adjustment value is defined in the table shown on the previous slide.

SPLOOPW – Differences

The **SPLOOPW** instruction is used to initiate a loop buffer operation when the total number of loops required in not known in advance. E.g. while loops.

Differences SPLOOP vs. SPLOOPW:
• This instruction executes unconditionally and cannot be predicated
• The SPLOOPW instruction invokes the loop buffer mechanism. The testing of the termination
•condition is delayed for four cycles.
• The **SPLOOPW** instruction cannot be used in a nested SPLOOP operation.
• When the **SPLOOPW** instruction is used to initiate a loop buffer operation, the
epilog is skipped when the loop terminates.

SPLOOPW – Piping the loop buffer

[A1] **SPLOOP 3**

```
           instr.D1
||         instr.S1          cycle 1

           instr.D1
||         instr.L2
||         instr.M1          cycle 2

           instr.S2
||         instr.M2          cycle 3

           instr.S2
||         instr.M1
||         instr.M2          cycle 4

           instr.S1
||         instr.L1
||         instr.M2
||         instr.D2          cycle 5

           instr.L1
||         instr.M1
||         instr.D1
|| SPKERNEL                  cycle 6
```

Execute

Execute

Execute

3 execute packets

KERNEL table

S1	S2	L1	L2	M1	M2	D1	D2
S1	S2	L1	L2	M1	M2	D1	D2
S1	S2	L1	L2	M1	M2	D1	D2

A1 = 4

LBC = 1/0

Finished loop buffer (prolog)

Execute kernel

Nested Loop

When the **SPLOOP** instruction is predicated, it indicates that the loop is a nested loop using the SPLOOP reload capability.

The SPMASKR /SPKERNELR instruction controls the reload point for nested loops.

The contents of the reload inner loop count register (RILC) is copied to ILC when either a SPKERNELR or a SPMASKR instruction is executed with the predication condition true.

The branch is used in a nested loop to place the PC back at the address of the execute packet after the SPKERNEL instruction.

Nested Loop

```
        [A1] SPLOOP 1

                instr.D1
    ||          instr.S1        cycle 1

                instr.D2
    ||          instr.L2
    ||          instr.M1        cycle 2


    SPKERNELR

outer_loop:
    [A1] BNOP.S2 outer_loop, 4

                SUB.L1 A1,1A1
    ||          instr.M2
```

KERNEL table

S1	S2	L1	L2	M1	M2	D1	D2

Execute kernel

A1 = 0   ILC = 0   RILC = 4

Reload kernel

Epilog

Finished

SPMASK

```
SPLOOP 1

                    instr.D1
||                  instr.S1        cycle 1

SPMASK D2
||                  instr.D2
||                  instr.L2
||                  instr.M1        cycle 2

                    instr.D2
||                  instr.L1
||                  instr.M2
||                  instr.S2        cycle 3

SPKERNEL

SPMASK
||^                 instr.S2
||^                 instr.L1

                    instr.D1
||                  instr.L2
||                  instr.M1
```

KERNEL table

Execute kernel

S1	S2	L1	L2	M1	M2	D1	D2

ILC = 4

Instruction not loaded into buffer but executed

Epilog

Finished

Interrupts

If an interrupt occurs while a software pipeline is executing out of the loop buffer, the loop will pipe down by executing an epilog and then service the interrupt. The interrupt return address stored in the IRP or NRP is the address of the execute packet containing the SPLOOP instruction.

The TSR (Task State Register) is copied into the ITSR (for interrupts) or NTSR (for NMI or exeptions) with the SPLX bit set to 1. On return from the interrupt with the ITSR or NTSR copied back into the TSR with the SPLX bit set to 1, execution is resumed at the address of the SPLOOP(D) instruction, and the loop is piped back up by executing a prolog.

ILC and RILC need to be saved/restored by Interrupt Service Routine

Breakpoints

The SPLOOP mechanism supports the placement of breakpoints, either hardware or software, at any execute packet within the SPLOOP/SPKERNEL body.

In case a HW Breakpoint with a count value greater than 1 is placed no other breakpoints may be set in the SPLOOP.

SPLOOP - Architecture

Loop Buffer
The loop buffer is used to store the instructions that comprise the loop and information describing the sequence that the instructions were added to the buffer and the state (active or inactive) of each instruction.
The loop buffer has enough storage for up to 14 execute packets.
Maximum loop body is 48 cycles.

Loop Buffer Count Register (LBC)
A loop buffer count register (LBC) is maintained as an index into the loop buffer. It is cleared to 0 when an **SPLOOP, SPLOOPD** or **SPLOOPW** instruction is encountered and is incremented by 1 at the end of each cycle. When LBC becomes equal to the iteration interval (II) specified by the **SPLOOP, SPLOOPD** or **SPLOOPW** instruction, then a stage boundary has been reached and LBC is reset to 0 and the inner loop count register (ILC) is decremented.
There are two LBCs to support overlapped nested loops. LBC is not a user-visible register.

SPLOOP - Architecture

Inner Loop Count Register (ILC)
The inner loop count register (ILC) is used as a down counter to determine when the SPLOOP is complete when the SPLOOP is initiated by either a **SPLOOP** or **SPLOOPD** instruction. When the loop is initiated using a **SPLOOPW** instruction, the ILC is not used to determine when the SPLOOP is complete. It is decremented once each time a stage boundary is encountered; that is, whenever the loop buffer count register (LBC) becomes equal to the iteration interval (ii).
There is a 4 cycle latency between when ILC is loaded and when its contents are available for use. When used with the **SPLOOP** instruction, it should be loaded 4 cycles before the **SPLOOP** instruction is encountered.

NOTE: ILC must be loaded explicitly using the **MVC** instruction.

SPLOOP - Architecture

Reload Inner Loop Count Register (RILC)
The reload inner loop count register (RILC) is used for resetting the inner loop count register (ILC) for the next invocation of a nested inner loop. There is a 4 cycle latency between when RILC is loaded with the **MVC** instructions and when the value loaded to RILC is available for use. RILC must be loaded explicitly using the **MVC** instruction.

Task State Register (TSR) and Interrupt Task State Register (ITSR)
The SPLX bit in the task state register (TSR) indicates whether an SPLOOP is currently executing or not executing.
When an interrupt occurs, the contents of TSR (including the SPLX bit) is copied to the interrupt task state register (ITSR).

SPLOOP - Architecture

SPLOOP Reload Inner Loop Count Register (RILC)
Predicated **SPLOOP** or **SPLOOPD** instructions used in conjunction with a **SPMASKR** or **SPKERNELR** instruction use the SPLOOP reload inner loop count register (RILC), as the iteration count value to be written
to the SPLOOP inner loop count register (ILC) in the cycle before the reload operation begins.

SPLOOP – Lab exercises

1. SPLOOP Debugging exercise: Single Step
2. SPLOOP Building the Piped Loop Kernel from SPLOOP asm code. (Reverse engineering, refresh c64x instruction pipelining)
3. SPLOOP Code Optimization exercise: How to write SPLOOP optimized code (tbd)
    - Analyze compiler feedback
    - Too long loops -> break loop apart.
    - Too complex -> Simplify loop

SPLOOP – Exercise 1

NOTE:

Single stepping into SPLOOP can only be done in non–real–time interrupt mode. If in real–time interrupt mode, a single step of the SPLOOP instruction will step over the entire SPLOOP operation.

SPLOOP – Exercise 2

```
  SPLOOP 2

L2:    ; PIPED LOOP KERNEL

          ADD      .D1    1,A2,A3
||        MPYLI   .M2X    B2,A6,B5:B4

          ADD      .L2    2,B2,B2
||        ADD      .L1    2,A2,A2
||        MPYLI   .M1     A3,A6,A5:A4

          NOP           2
          ADD     .L2    B4,B4,B7
          ADD     .S1    A4,A4,A3

          AND     .L1X   A6,B7,A0
||        AND     .S1    A6,A3,A1


          SPKERNEL
||        STDW    .D1T1   A1:A0,*A7++
```

SPLOOP – Exercise 2

Unit\cycle	0	2	4	6	8
.L1		ADD	ADD	ADD	ADD
.L2		ADD	ADD	ADD	ADD
.S1				ADD	ADD
.S2					
.M1		MPYLI	MPYLI	MPYLI	MPYLI
.M2					
.D1					STDW
.D2					

Unit\cycle	1	3	5	7	9
.L1				AND	AND
.L2			ADD	ADD	ADD
.S1				AND	AND
.S2					
.M1					
.M2	MPYLI	MPYLI	MPYLI	MPYLI	MPYLI
.D1	ADD	ADD	ADD	ADD	ADD
.D2					

Prolog & Kernel (in blue)

SPLOOP – Exercise 2

Unit\cycle	8	10	12	14	16
.L1	ADD	ADD			
.L2	ADD	ADD			
.S1	ADD	ADD	ADD	ADD	
.S2					
.M1	MPYLI	MPYLI			
.M2					
.D1	STDW	STDW	STDW	STDW	STDW
.D2					

Unit\cycle	9	11	13	15	17
.L1	AND	AND	AND	AND	
.L2	ADD	ADD	ADD		
.S1	AND	AND	AND	AND	
.S2					
.M1					
.M2	MPYLI				
.D1	ADD				
.D2					

Epilog & Kernel (in blue)