Network Working Group
Request For Comments: 1936
Category: Informational

J. Touch
B. Parham
ISI

April 1996

Implementing the Internet Checksum in Hardware

Status of This Memo

This memo provides information for the Internet community. This memo does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

Abstract

This memo presents a techniques for efficiently implementing the Internet Checksum in hardware. It includes PLD code for programming a single, low cost part to perform checksumming at 1.26 Gbps.

Introduction

The Internet Checksum is used in various Internet protocols to check for data corruption in headers (e.g., IP) [4] and packet bodies (e.g, UDP, TCP) [5][6]. Efficient software implementation of this checksum has been addressed in previous RFCs [1][2][3][7].

Efficient software implementations of the Internet Checksum algorithm are often embedded in data copying operations ([1], Section 2). This copy operation is increasingly being performed by dedicated direct memory access (DMA) hardware. As a result, DMA hardware designs are beginning to incorporate dedicated hardware to compute the Internet Checksum during the data transfer.

This note presents the architecture of an efficient, pipelined Internet Checksum mechanism, suitable for inclusion in DMA hardware [8]. This design can be implemented in a relatively inexpensive programmable logic device (PLD) (1995 cost of $40), and is capable of supporting 1.26 Gbps transfer rates, at 26 ns per 32-bit word. Appendix A provides the pseudocode for such a device. This design has been implemented in the PC-ATOMIC host interface hardware [8]. We believe this design is of general use to the Internet community.

The remainder of this document is organized as follows:

Review of the Internet Checksum
One's Complement vs. Two's Complement Addition
Interfaces
Summary
Appendix A - PLD source code

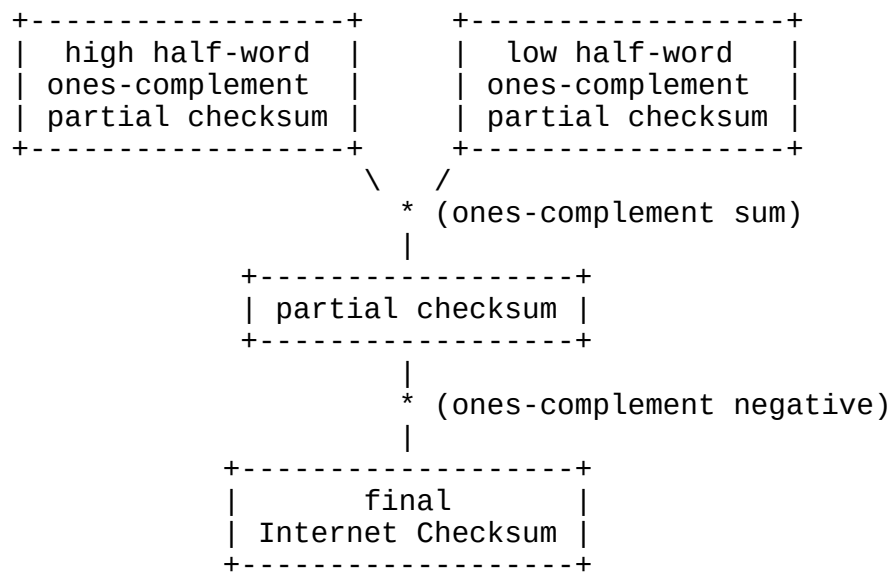
A Review of the Internet Checksum

The Internet Checksum is used for detecting corruption in a block of data [1]. It is initialized to zero, and computed as the complement of the ones-complement sum of the data, taken in 16-bit units. A subsequent checksum of the data and checksum together should generate a zero checksum if no errors are detected.

The checksum allows [1]:

- byte order "independence"
  reordered output is equivalent to reordered input
- 16-bit word-order independence
  reordering 16-bit words preserves the output
- incremental computation
- deferred carries
- parallel summation
  a result of deferred carries, incremental computation, and 16-bit word order independence

This note describes an implementation that computes two partial checksums in parallel, over the odd and even 16-bit half-words of 32-bit data. The result is a pair of partial checksums (odd and even), which can be combined, and the result inverted to generate the true Internet Checksum. This technique is related to the long-word parallel summation used in efficient software implementations [1].



One's Complement vs. Two's Complement Addition

The Internet Checksum is composed of a ones-complement lookahead adder and a bit-wise inverter. A ones-complement adder can be built either using twos-complement components, or natively.

A twos-complement implementation of a ones-complement adder requires either two twos-complement adders, or two cycles per add. The sum is performed, then the high-bit carry-out is propagated to the carry-in, and a second sum is performed. (ones-complement addition is {+1s} and twos-complement is {+2s})

```
a {+1s} b == (a {+2s} b) + carry(a {+2s} b)

e.g.,
        halfword16 a,b;
        word32 c;
        a {+1s} b == r
such that:
        c = a {+2s} b;                          # sum value
        r = (c & 0xFFFF) {+2s} (c >> 16);       # sum carry
```

Bits of a twos-complement lookahead adder are progressively more complex in carry lookahead. (OR the contents of each row, where terms are AND'd or XOR'd {^})

```
4-bit carry-lookahead 2's complement adder:
        a,b : input data
        p   : carry propagate, where pi = ai*bi = (ai)(bi)
        g   : carry generate, where gi = ai + bi

Out0 := a0 ^ b0 ^ ci

Out1 := a1 ^ b1 ^ (cip0     + g0)

Out2 := a2 ^ b2 ^ (cip0p1   + g0p1   + g1)

Out3 := a3 ^ b3 ^ (cip0p1p2 + g0p1p2 + g1p2 + g2)

Cout := cip0p1p2p3 + g0p1p2p3 + g1p2p3 + g2p3 + g3
```

The true ones-complement lookahead adder recognizes that the carry-wrap of the twos-complement addition is equivalent to a toroidal carry-lookahead. Bits of a ones-complement lookahead adder are all the same complexity, that of the high-bit of a twos-complement lookahead adder. Thus the ones-complement sum (and thus the Internet Checksum) is bit-position independent. We replace 'ci' with the 'co' expression and reduce. (OR terms in each row pair).

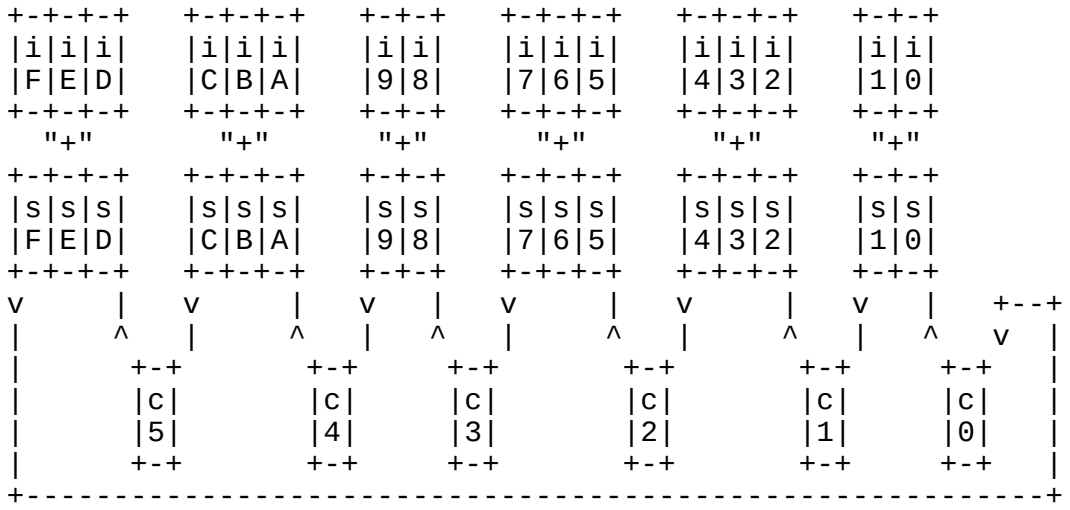
4-bit carry-lookahead 1's complement ring adder:

```
Out0 = a0 ^ b0 ^ (g3       + g2p3     + g1p2p3   + g0p1p2p3)

Out1 = a1 ^ b1 ^ (g3p0     + g2p3p0   + g1p2p3p0 + g0)

Out2 = a2 ^ b2 ^ (g3p0p1   + g2p3p0p1 + g1       + g0p1)

Out3 = a3 ^ b3 ^ (g3p0p1p2 + g2       + g1p2     + g0p1p2)
```

A hardware implementation can use this toroidal design directly, together with conventional twos-complement fast-adder internal components, to perform a pipelined ones-complement adder [8].

A VLSI implementation could use any full-lookahead adder, adapted to be toroidal and bit-equivalent, as above. In our PLD implementation, we implement the adders via 2- and 3-bit full-lookahead sub-components. The adder components are chained in a ring via carry bit registers. This relies on delayed carry-propagation to implement a carry pipeline between the fast-adder stages.

Full-lookahead adders in a toroidal pipeline



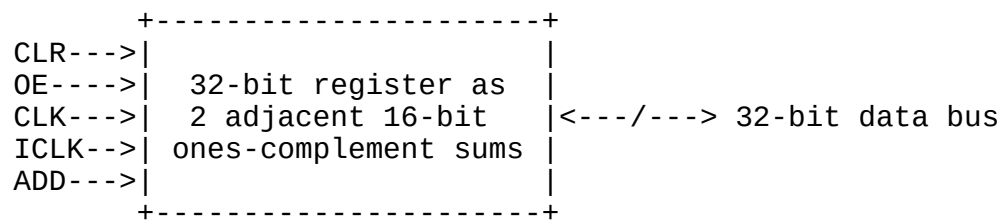
Implementation of fast-adders in PLD hardware is currently limited to 3-bits, because an i-bit adder requires 4+2^i product terms, and current PLDs support only 16 product terms. The resulting device takes at most 5 "idle" clock periods for the carries to propagate through the accumulation pipeline.

Interfaces

The above device has been installed in a VL-Bus PC host interface card [8]. It has a hardware and software interface, defined as follows.

Hardware Interface

The Internet Checksum hardware appears as a single-port 32-bit register, with clock and control signals [8]:




```
CLR    = zero the register
OE     = write the register onto the data bus
CLK    = clock to cycle the pipeline operation
ICLK   = input data latch clock
ADD    = initiating an add of latched input data
```

CLR causes the contents of the checksum register and input latch to be zeroed. There is no explicit load; a CLR followed by a write of the load value to a dummy location is equivalent.

The OE causes the register to be written to the data bus, or tri-stated.

The CLK causes the pipeline to operate. If no new input data is latched to be added (via ICLK, ADD), a virtual "zero" is summed into the register, to permit the pipeline to empty.

The ICLK (transparently) latches the value on the data bus to be latched internally, to be summed into the accumulator on the next ADD signal. The ADD signal causes the latched input data (ICLK) to be accumulated into the checksum pipeline. ADD and ICLK are commonly tied together. One 32-bit data value can be latched and accumulated into the pipeline adder every 26-ns clock, assuming data is stable when the ADD/ICLK signal occurs.

The internal 32-bit register is organized as two 16-bit ones-complement sums, over the even and odd 16-bit words of the data stream. To compute the Internet Checksum from this quantity, ones-complement add the halves together, and invert the result.

Software Interface

The device is used as a memory-mapped register. The register is read by performing a read on its equivalent memory location.

The device is controlled via an external memory-mapped register. Bits in this control register clear the device (set/clear the CLR line), and enable and disable the device (set/clear the ADD line). The CLR line can alternatively be mapped to a memory write, e.g., such that reading the location is a non-destructive read of the checksum register, and a write of any value clears the checksum register. The enable/disable control must be stored in an external register.

The device is designed to operate in background during memory transfers (either DMA or programmed I/O). Once enabled, all transfers across that bus are summed into the checksum register. The checksum is available 5 clocks after the last enabled data accumulation. This delay is often hidden by memory access mechanisms and bus arbitration. If required, "stall" instructions can be executed for the appropriate delay.

For the following example, we assume that the device is located at CKSUMLOC. We assume that reading that location reads the checksum register, and writing any value to that location clears the register. The control register is located at CTLLOC, and the checksum enable/disable bit is CKSUMBIT, where 1 is enabled, and 0 is disabled. To perform a checksum, a programmer would clear the register, (optionally initialize the checksum), initiate a series of transfers, and use the result:

```
/******* initialization *******/
*(CTLLOC) &= ~((ctlsize)(CKSUMBIT));     /* disable sum */
(word32)(*(CKSUMLOC)) = 0;               /* clear reg   */
*(CTLLOC) |= CKSUMBIT;                   /* enable sum  */
{ (optional) write initial value to a dummy location }

/***** perform a transfer *****/
{ do one or more DMA or PIO transfers - read or write }

/***** gather the results *****/
*(CTLLOC) &= ~((ctlsize)(CKSUMBIT));     /* disable sum  */
sum = (word32)(*(CKSUMLOC));             /* read sum     */
sum = (sum & 0xFFFF) + (sum >> 16);      /* fold halves  */
sum = (sum & 0xFFFF) + (sum >> 16);      /* add in carry */
ipcksum = (halfword16)(~(sum & 0xFFFF)); /* 1's negative */
```

Summary

This note describes the design of a hardware Internet Checksum that can be implemented in an inexpensive PLD, achieving 1.26 Gbps. This design has been implemented in the PC-ATOMIC host interface hardware [8]. We believe this design is of general use to the Internet community.

Security Considerations

Security considerations are not addressed here. The Internet Checksum is not intended as a security measure.

Acknowledgements

The authors would like to thank the members of the "High-Performance Computing and Communications", notably Mike Carlton, and "Advanced Systems" Divisions at ISI for their assistance in the development of the hardware, and this memo.

References

[1] Braden, R., Borman, D., and Partridge, C., "Computing the Internet Checksum," Network Working Group RFC-1071, ISI, Cray Research, and BBN Labs, Sept. 1988.

[2] Mallory, T., and Kullberg, A., "Incremental Updating of the Internet Checksum," Network Working Group RFC-1141, BBN Comm., Jan. 1990.

[3] Plummer, W., "TCP Checksum Function Design," IEN-45, BBN, 1978, included as an appendix in RFC-1071.

[4] Postel, Jon, "Internet Protocol," Network Working Group RFC-791/STD-5, ISI, Sept. 1981.

[5] Postel, Jon, "User Datagram Protocol," Network Working Group RFC-768/STD-6, ISI, Aug. 1980.

[6] Postel, Jon, "Transmission Control Protocol," Network Working Group RFC-793/STD-7, ISI, Sept. 1981.

[7] Rijsinghani, A., "Computation of the Internet Checksum via Incremental Update," Network Working Group RFC-1624, Digital Equipment Corp., May 1994.

[8] Touch, J., "PC-ATOMIC", ISI Tech. Report. SR-95-407, June 1995.

Authors' Addresses

Joe Touch
University of Southern California/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292-6695
USA
Phone: +1 310-822-1511 x151
Fax: +1 310-823-6714
URL: http://www.isi.edu/~touch
EMail: touch@isi.edu

Bruce Parham
University of Southern California/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292-6695
USA
Phone: +1 310-822-1511 x101
Fax: +1 310-823-6714
EMail: bparham@isi.edu

Appendix A: PLD source code

The following is the PLD source code for an AMD MACH-435 PLD. The MACH-435 is composed of 8 22V10-equivalent PLD blocks, connected by a configurable internal matrix.

---- (PLD source code follows) ----

```
TITLE    PC-ATOMIC IP Sum Accelerator - 1-clock 2- and 3-bit 26 ns version
PATTERN  ip_sum
REVISION 1.01
AUTHOR   J. Touch & B. Parham
COMPANY  USC/ISI
DATE     06/21/94

CHIP    ip_sum          MACH435

; accumulates in 1 clock (1 level of logic)
;
; resources allocated to reduce fitting time
;
; uses an input register "dl" to latch the data bus values on rising edge
; accumulates a hi/lo ones-complement sum in register "q"
; the input and output are accessed via bidirectional pins "dq"
;
; uses 2 groups of 6 carry bit registers "cy"
;
; use 3-bit full-adders with carry lookahead (settles in 6 clocks)
; group 16 bits as      [000102 030405 0607 080910 111213 1415]
;                       [161718 192021 2223 242526 272829 3031]
;
; locking the pins down speeds up fitting and is designed to force
; 4-bit components into single "segments" of the PLD.
; we could have indicated the same thing via:
;       GROUP MACH_SEG_A        dq[6..0]
;       GROUP MACH_SEG_B        dq[14..8]
;       GROUP MACH_SEG_C        dq[22..16]
;       GROUP MACH_SEG_D        dq[30..24]

;
; control pins:
;
PIN     20      clk             ; adder clock
PIN     62      ip_add          ; add current data to sum
PIN     83      ip_sum_ena      ; output current sum
PIN     41      ip_clr          ; clear current sum
PIN 23  ip_dclk                 ; input data latch (tied to clk, or not)
```

```
;
; dq are data bus pins
; dl is the input register
;
PIN     [9..3]          dq[6..0] IPAIR dl[6..0]         ; IO port
PIN     [18..12]        dq[14..8] IPAIR dl[14..8]       ; IO port
PIN     [30..24]        dq[22..16] IPAIR dl[22..16]     ; IO port
PIN     [39..33]        dq[30..24] IPAIR dl[30..24]     ; IO port
PIN     ?       dq[31,23,15,7] IPAIR dl[31,23,15,7]     ; IO port

;
; q  is the partial checksum register
; dl is the input register
; dq are the data bus pins
;
NODE    ?       q[31..0] OPAIR dq[31..0]        ; internal data in reg
NODE    ?       dl[31..0] REG                   ; input reg

;
; cy are the carry register bits
;
NODE    ?       cy[31,29,26,23,21,18,15,13,10,7,5,2] REG
                                                ;1-bit internal carry bits

EQUATIONS

;
; .trst is the tri-state control, 0 means these are always inputs
;
ip_add.trst             = 0
ip_clr.trst             = 0
ip_sum_ena.trst         = 0

;
; grab data to the input register on every clock (irrelevant if invalid)
;
dl[31..0].clkf          = ip_dclk       ; grab data all the time
        ; don't use setf, rstf, or trst for dl
        ; we want dl to map to input registers, not internal cells
        ; besides, input registers don't need setf, rstf, or trst

;
; control of the checksum register
;
dq[31..0].clkf          = clk                   ; clk clocks everything
dq[31..0].setf          = gnd                   ; never preset registers
dq[31..0].rstf          = ip_clr                ; clear on reset
dq[31..0].trst          = ip_sum_ena            ; ena outputs sum - read
```

```
;
; control for the carry register
;
cy[31,29,26,23,21,18,15,13,10,7,5,2].clkf       = clk
cy[31,29,26,23,21,18,15,13,10,7,5,2].setf       = gnd    ; never preset
cy[31,29,26,23,21,18,15,13,10,7,5,2].rstf       = ip_clr ; clear on reset

;
; INPUT DATA LATCH
; nothing fancy here - grab all inputs when ip_add signal is high
; i.e., grab data in input register
;
dl[31..0]             := dq[31..0]

;
; COMBINATORIAL ADDER
;
; built as a series of 2-bit and 3-bit (carry-lookahead) full-adders
; with carries sent to the carry register "pipeline"
;
; sum[n] are sum bits
; cy[m] are carry bits
; ":+:" is XOR

;
; SUM[0] = (A0 :+: B0 :+: CARRY_IN)
;
; CY[0] = ((A0 * B0) + ((A0 :+: B0) * CARRY_IN))
;
; actually, the latter can be rewritten as
;
; CY[0] = ((A0 * B0) + ((A0 + B0) * CARRY_IN))
;
; because the XOR won't be invalidated by the AND case, since the
; result is always 1 from the first term then anyway
; this helps reduce the number of XOR terms required, which are
; a limited resource in PLDs
;
```

```
; SUM THE LOW-ORDER WORD
;

;
; the first 5 bits [0..4] of the low-order word
;
q[0]    := (q[0] :+: (ip_add * dl[0]) :+: cy[15])

q[1]    := (q[1] :+: (ip_add * dl[1]) :+:
        ((ip_add *
                (q[0] * dl[0] +
                dl[0] * cy[15])) +
        (q[0] * cy[15])))

q[2]    := (q[2] :+: (ip_add * dl[2]) :+:
        ((ip_add *
                (q[1] * dl[1] +
                 q[1] *  q[0] *  dl[0] +
                dl[1] *  q[0] *  dl[0] +
                 q[1] * dl[0] * cy[15] +
                dl[1] * dl[0] * cy[15] +
                dl[1] *  q[0] * cy[15])) +
        (q[1] * q[0] * cy[15])))

cy[2]   := ((ip_add *
                (q[2] * dl[2] +
                 q[2] *  q[1] * dl[1] +
                dl[2] *  q[1] * dl[1] +
                 q[2] *  q[1] *  q[0] *  dl[0] +
                 q[2] * dl[1] *  q[0] *  dl[0] +
                dl[2] *  q[1] *  q[0] *  dl[0] +
                dl[2] * dl[1] *  q[0] *  dl[0] +
                 q[2] *  q[1] * dl[0] * cy[15] +
                 q[2] * dl[1] *  q[0] * cy[15] +
                 q[2] * dl[1] * dl[0] * cy[15] +
                dl[2] *  q[1] *  q[0] * cy[15] +
                dl[2] *  q[1] * dl[0] * cy[15] +
                dl[2] * dl[1] *  q[0] * cy[15] +
                dl[2] * dl[1] * dl[0] * cy[15])) +
        (q[2] * q[1] * q[0] * cy[15]))

q[3]    := (q[3] :+: (ip_add * dl[3]) :+: cy[2])

q[4]    := (q[4] :+: (ip_add * dl[4]) :+:
        ((ip_add *
                (q[3] * dl[3] +
                dl[3] * cy[2])) +
        (q[3] * cy[2])))
```

```
;
; the next 3 bits [5..7] of the low-order word
;
q[5]    := (q[5] :+: (ip_add * dl[5]) :+:
        ((ip_add *
                (q[4] * dl[4] +
                 q[4] *  q[3] * dl[3] +
                dl[4] *  q[3] * dl[3] +
                 q[4] * dl[3] * cy[2] +
                dl[4] * dl[3] * cy[2] +
                dl[4] *  q[3] * cy[2])) +
        (q[4] * q[3] * cy[2])))

cy[5]   := ((ip_add * (
                 q[5] * dl[5] +
                 q[5] *  q[4] * dl[4] +
                dl[5] *  q[4] * dl[4] +
                 q[5] *  q[4] *  q[3] * dl[3] +
                 q[5] * dl[4] *  q[3] * dl[3] +
                dl[5] *  q[4] *  q[3] * dl[3] +
                dl[5] * dl[4] *  q[3] * dl[3] +
                 q[5] *  q[4] * dl[3] * cy[2] +
                 q[5] * dl[4] *  q[3] * cy[2] +
                 q[5] * dl[4] * dl[3] * cy[2] +
                dl[5] *  q[4] *  q[3] * cy[2] +
                dl[5] *  q[4] * dl[3] * cy[2] +
                dl[5] * dl[4] *  q[3] * cy[2] +
                dl[5] * dl[4] * dl[3] * cy[2])) +
        (q[5] * q[4] * q[3] * cy[2]))

q[6]    := (q[6] :+: (ip_add * dl[6]) :+: cy[5])

q[7]    := (q[7] :+: (ip_add * dl[7]) :+:
        ((ip_add *
                (q[6] * dl[6] +
                dl[6] * cy[5])) +
        (q[6] * cy[5])))

cy[7]   := ((ip_add *
                (q[7] * dl[7] +
                 q[7] *  q[6] * dl[6] +
                dl[7] *  q[6] * dl[6] +
                 q[7] * dl[6] * cy[5] +
                dl[7] * dl[6] * cy[5] +
                dl[7] *  q[6] * cy[5])) +
        (q[7] * q[6] * cy[5]))
```

```
;
; the next 5 bits [8..12] of the low-order word
;
q[8]    := (q[8] :+: (ip_add * dl[8]) :+: cy[7])

q[9]    := (q[9] :+: (ip_add * dl[9]) :+:
        ((ip_add *
                (q[8] * dl[8] +
                dl[8] * cy[7])) +
        (q[8] * cy[7])))

q[10]   := (q[10] :+: (ip_add * dl[10]) :+:
        ((ip_add *
                (q[9] * dl[9] +
                 q[9] *  q[8] * dl[8] +
                dl[9] *  q[8] * dl[8] +
                 q[9] * dl[8] * cy[7] +
                dl[9] * dl[8] * cy[7] +
                dl[9] *  q[8] * cy[7])) +
        (q[9] * q[8] * cy[7])))

cy[10]  := ((ip_add *
                (q[10] * dl[10] +
                 q[10] *  q[9] * dl[9] +
                dl[10] *  q[9] * dl[9] +
                 q[10] *  q[9] *  q[8] * dl[8] +
                 q[10] * dl[9] *  q[8] * dl[8] +
                dl[10] *  q[9] *  q[8] * dl[8] +
                dl[10] * dl[9] *  q[8] * dl[8] +
                 q[10] *  q[9] * dl[8] * cy[7] +
                 q[10] * dl[9] *  q[8] * cy[7] +
                 q[10] * dl[9] * dl[8] * cy[7] +
                dl[10] *  q[9] *  q[8] * cy[7] +
                dl[10] *  q[9] * dl[8] * cy[7] +
                dl[10] * dl[9] *  q[8] * cy[7] +
                dl[10] * dl[9] * dl[8] * cy[7])) +
        (q[10] * q[9] * q[8] * cy[7]))

q[11]   := (q[11] :+: (ip_add * dl[11]) :+: cy[10])

q[12]   := (q[12] :+: (ip_add * dl[12]) :+:
        ((ip_add *
                (q[11] * dl[11] +
                dl[11] * cy[10])) +
        (q[11] * cy[10])))
```

```
;
; the final 3 bits [13..15] of the low-order word
;
q[13]   := (q[13] :+: (ip_add * dl[13]) :+:
        ((ip_add *
                (q[12] * dl[12] +
                 q[12] *  q[11] * dl[11] +
                dl[12] *  q[11] * dl[11] +
                 q[12] * dl[11] * cy[10] +
                dl[12] * dl[11] * cy[10] +
                dl[12] *  q[11] * cy[10])) +
        (q[12] * q[11] * cy[10])))

cy[13]  := ((ip_add * (
                 q[13] * dl[13] +
                 q[13] *  q[12] * dl[12] +
                dl[13] *  q[12] * dl[12] +
                 q[13] *  q[12] *  q[11] * dl[11] +
                 q[13] * dl[12] *  q[11] * dl[11] +
                dl[13] *  q[12] *  q[11] * dl[11] +
                dl[13] * dl[12] *  q[11] * dl[11] +
                 q[13] *  q[12] * dl[11] * cy[10] +
                 q[13] * dl[12] *  q[11] * cy[10] +
                 q[13] * dl[12] * dl[11] * cy[10] +
                dl[13] *  q[12] *  q[11] * cy[10] +
                dl[13] *  q[12] * dl[11] * cy[10] +
                dl[13] * dl[12] *  q[11] * cy[10] +
                dl[13] * dl[12] * dl[11] * cy[10])) +
        (q[13] * q[12] *  q[11] * cy[10]))

q[14]   := (q[14] :+: (ip_add * dl[14]) :+: cy[13])

q[15]   := (q[15] :+: (ip_add * dl[15]) :+:
        ((ip_add *
                (q[14] * dl[14] +
                dl[14] * cy[13])) +
        (q[14] * cy[13])))

cy[15]  := ((ip_add *
                (q[15] * dl[15] +
                 q[15] *  q[14] * dl[14] +
                dl[15] *  q[14] * dl[14] +
                 q[15] * dl[14] * cy[13] +
                dl[15] * dl[14] * cy[13] +
                dl[15] *  q[14] * cy[13])) +
        (q[15] * q[14] * cy[13]))
```

```
; SUM THE HIGH-ORDER WORD
;

;
; the first 5 bits [16..20] of the high-order word
;
q[16]   := (q[16] :+: (ip_add * dl[16]) :+: cy[31])

q[17]   := (q[17] :+: (ip_add * dl[17]) :+:
        ((ip_add *
                (q[16] * dl[16] +
                dl[16] * cy[31])) +
        (q[16] * cy[31])))

q[18]   := (q[18] :+: (ip_add * dl[18]) :+:
        ((ip_add *
                (q[17] * dl[17] +
                 q[17] *  q[16] * dl[16] +
                dl[17] *  q[16] * dl[16] +
                 q[17] * dl[16] * cy[31] +
                dl[17] * dl[16] * cy[31] +
                dl[17] *  q[16] * cy[31])) +
        (q[17] * q[16] * cy[31])))

cy[18]  := ((ip_add *
                (q[18] * dl[18] +
                 q[18] *  q[17] * dl[17] +
                dl[18] *  q[17] * dl[17] +
                 q[18] *  q[17] *  q[16] * dl[16] +
                 q[18] * dl[17] *  q[16] * dl[16] +
                dl[18] *  q[17] *  q[16] * dl[16] +
                dl[18] * dl[17] *  q[16] * dl[16] +
                 q[18] *  q[17] * dl[16] * cy[31] +
                 q[18] * dl[17] *  q[16] * cy[31] +
                 q[18] * dl[17] * dl[16] * cy[31] +
                dl[18] *  q[17] *  q[16] * cy[31] +
                dl[18] *  q[17] * dl[16] * cy[31] +
                dl[18] * dl[17] *  q[16] * cy[31] +
                dl[18] * dl[17] * dl[16] * cy[31])) +
        (q[18] * q[17] * q[16] * cy[31]))

q[19]   := (q[19] :+: (ip_add * dl[19]) :+: cy[18])

q[20]   := (q[20] :+: (ip_add * dl[20]) :+:
        ((ip_add *
                (q[19] * dl[19] +
                dl[19] * cy[18])) +
        (q[19] * cy[18])))
```

```
;
; the next 3 bits [21..23] of the high-order word
;
q[21]   := (q[21] :+: (ip_add * dl[21]) :+:
        ((ip_add *
                (q[20] * dl[20] +
                 q[20] *  q[19] * dl[19] +
                dl[20] *  q[19] * dl[19] +
                 q[20] * dl[19] * cy[18] +
                dl[20] * dl[19] * cy[18] +
                dl[20] *  q[19] * cy[18])) +
        (q[20] * q[19] * cy[18])))

cy[21]  := ((ip_add * (
                 q[21] * dl[21] +
                 q[21] *  q[20] * dl[20] +
                dl[21] *  q[20] * dl[20] +
                 q[21] *  q[20] *  q[19] * dl[19] +
                 q[21] * dl[20] *  q[19] * dl[19] +
                dl[21] *  q[20] *  q[19] * dl[19] +
                dl[21] * dl[20] *  q[19] * dl[19] +
                 q[21] *  q[20] * dl[19] * cy[18] +
                 q[21] * dl[20] *  q[19] * cy[18] +
                 q[21] * dl[20] * dl[19] * cy[18] +
                dl[21] *  q[20] *  q[19] * cy[18] +
                dl[21] *  q[20] * dl[19] * cy[18] +
                dl[21] * dl[20] *  q[19] * cy[18] +
                dl[21] * dl[20] * dl[19] * cy[18])) +
        (q[21] * q[20] * q[19] * cy[18]))

q[22]   := (q[22] :+: (ip_add * dl[22]) :+: cy[21])

q[23]   := (q[23] :+: (ip_add * dl[23]) :+:
        ((ip_add *
                (q[22] * dl[22] +
                dl[22] * cy[21])) +
        (q[22] * cy[21])))

cy[23]  := ((ip_add *
                (q[23] * dl[23] +
                 q[23] *  q[22] * dl[22] +
                dl[23] *  q[22] * dl[22] +
                 q[23] * dl[22] * cy[21] +
                dl[23] * dl[22] * cy[21] +
                dl[23] *  q[22] * cy[21])) +
        (q[23] * q[22] * cy[21]))
```

```
;
; the next 5 bits [24..28] of the high-order word
;
q[24]   := (q[24] :+: (ip_add * dl[24]) :+: cy[23])

q[25]   := (q[25] :+: (ip_add * dl[25]) :+:
        ((ip_add *
                (q[24] * dl[24] +
                dl[24] * cy[23])) +
        (q[24] * cy[23])))

q[26]   := (q[26] :+: (ip_add * dl[26]) :+:
        ((ip_add *
                (q[25] * dl[25] +
                 q[25] *  q[24] * dl[24] +
                dl[25] *  q[24] * dl[24] +
                 q[25] * dl[24] * cy[23] +
                dl[25] * dl[24] * cy[23] +
                dl[25] *  q[24] * cy[23])) +
        (q[25] * q[24] * cy[23])))

cy[26]  := ((ip_add *
                (q[26] * dl[26] +
                 q[26] *  q[25] * dl[25] +
                dl[26] *  q[25] * dl[25] +
                 q[26] *  q[25] *  q[24] * dl[24] +
                 q[26] * dl[25] *  q[24] * dl[24] +
                dl[26] *  q[25] *  q[24] * dl[24] +
                dl[26] * dl[25] *  q[24] * dl[24] +
                 q[26] *  q[25] * dl[24] * cy[23] +
                 q[26] * dl[25] *  q[24] * cy[23] +
                 q[26] * dl[25] * dl[24] * cy[23] +
                dl[26] *  q[25] *  q[24] * cy[23] +
                dl[26] *  q[25] * dl[24] * cy[23] +
                dl[26] * dl[25] *  q[24] * cy[23] +
                dl[26] * dl[25] * dl[24] * cy[23])) +
        (q[26] * q[25] * q[24] * cy[23]))

q[27]   := (q[27] :+: (ip_add * dl[27]) :+: cy[26])

q[28]   := (q[28] :+: (ip_add * dl[28]) :+:
        ((ip_add *
                (q[27] * dl[27] +
                dl[27] * cy[26])) +
        (q[27] * cy[26])))
```

```
;
; the final 3 bits [29..31] of the high-order word
;
q[29]   := (q[29] :+: (ip_add * dl[29]) :+:
        ((ip_add *
                (q[28] * dl[28] +
                 q[28] *  q[27] * dl[27] +
                dl[28] *  q[27] * dl[27] +
                 q[28] * dl[27] * cy[26] +
                dl[28] * dl[27] * cy[26] +
                dl[28] *  q[27] * cy[26])) +
        (q[28] * q[27] * cy[26])))

cy[29]  := ((ip_add * (
                 q[29] * dl[29] +
                 q[29] *  q[28] * dl[28] +
                dl[29] *  q[28] * dl[28] +
                 q[29] *  q[28] *  q[27] * dl[27] +
                 q[29] * dl[28] *  q[27] * dl[27] +
                dl[29] *  q[28] *  q[27] * dl[27] +
                dl[29] * dl[28] *  q[27] * dl[27] +
                 q[29] *  q[28] * dl[27] * cy[26] +
                 q[29] * dl[28] *  q[27] * cy[26] +
                 q[29] * dl[28] * dl[27] * cy[26] +
                dl[29] *  q[28] *  q[27] * cy[26] +
                dl[29] *  q[28] * dl[27] * cy[26] +
                dl[29] * dl[28] *  q[27] * cy[26] +
                dl[29] * dl[28] * dl[27] * cy[26])) +
        (q[29] * q[28] * q[27] * cy[26]))

q[30]   := (q[30] :+: (ip_add * dl[30]) :+: cy[29])

q[31]   := (q[31] :+: (ip_add * dl[31]) :+:
        ((ip_add *
                (q[30] * dl[30] +
                dl[30] * cy[29])) +
        (q[30] * cy[29])))

cy[31]  := ((ip_add *
                (q[31] * dl[31] +
                 q[31] *  q[30] * dl[30] +
                dl[31] *  q[30] * dl[30] +
                 q[31] * dl[30] * cy[29] +
                dl[31] * dl[30] * cy[29] +
                dl[31] *  q[30] * cy[29])) +
        (q[31] * q[30] * cy[29]))
```

```
;
; output nodes onto output pins (pending enable..)
;
dq[0]   := {q[0]}
dq[1]   := {q[1]}
dq[2]   := {q[2]}
dq[3]   := {q[3]}
dq[4]   := {q[4]}
dq[5]   := {q[5]}
dq[6]   := {q[6]}
dq[7]   := {q[7]}
dq[8]   := {q[8]}
dq[9]   := {q[9]}
dq[10]  := {q[10]}
dq[11]  := {q[11]}
dq[12]  := {q[12]}
dq[13]  := {q[13]}
dq[14]  := {q[14]}
dq[15]  := {q[15]}

dq[16]  := {q[16]}
dq[17]  := {q[17]}
dq[18]  := {q[18]}
dq[19]  := {q[19]}
dq[20]  := {q[20]}
dq[21]  := {q[21]}
dq[22]  := {q[22]}
dq[23]  := {q[23]}
dq[24]  := {q[24]}
dq[25]  := {q[25]}
dq[26]  := {q[26]}
dq[27]  := {q[27]}
dq[28]  := {q[28]}
dq[29]  := {q[29]}
dq[30]  := {q[30]}
dq[31]  := {q[31]}

;
; end.
;
```