GEODÆTISK INSTITUTS INTERNE RAPPORT NR. 14
THE DANISH GEODETIC INSTITUTE
INTERNAL REPORT NO. 14

Appendix 1.

USERS MANUAL FOR GPU
General Processing Unit
utilizing
Double Processing Unit

General Processing Unit designed for double precision floating point arithmetic after Wilkinson. The double precision orders are processed by the DPU.

K. Engsager

1983

ISBN 87 7450-048-1

GEODÆTISK INSTITUTS INTERNE RAPPORT NR. 14
THE DANISH GEODETIC INSTITUTE
INTERNAL REPORT NO. 14

Appendix 1.

USERS MANUAL FOR GPU
General Processing Unit
utilizing
Double Processing Unit

General Processing Unit designed for double
precision floating point arithmetic after Wilkinson.
The double precision orders are processed by the DPU.

K. Engsager

1983

ISBN 87 7450-048-1

References :

1 GPU reference manual. K. Engsager. Geodætisk Institut 1982.

2 DPU reefernce manual. K. Engsager. Geodætisk Institut 1982.

3 GPU driverproc, GPU blockprocedure, GPU execute.
K. Engsager. Geodætisk Institut 1982.

4 Algol 8. Users Guide. Part 2. RCSL 42 - i1278

5 Code procedures and the runtime organisaztion of ALGOL programs
RCSL 31 - D119.

6 RC Slang Assembler programming Guide. RCSL 42 - i0785.

Contents

Preface

The GPU is a reprogrammed RC8000/35 central unit. The GPU executes a piece of code (slang), which is pointed by a message send to it. Some procedures have been designed to handle the device in a standard way in algol programs. E.g. gpu_open gpu_close and gpu_error.

The block_procedure gpu_error wil handle any error situation concerning the gpu in such a way that the user will know whether it is overflow, time_out, bus-error, attempt to store in write protected area or attempt to execute instructions outside the pointed code area. If the installation has no GPU, the block-procedure will emulate the special gpu-instruction (in single presision only) using the escape facillity (this might be a little slow).

2. Declarations

ALGOL

The zone used to communicate with the GPU must be declared :

```
zone code(length, 1, gpu_error);
```

The name code is freely choosen among others.

The block_procedure gpu_error must be used because the GPU uses non standard status-bits in the answer.

NOT ALGOL

An area may be laid out to the code and the communication may be handled by the message - answer monitor calls.

3. Open

ALGOL

Use the procedure : gpu_open(z, gpu_name, code_area, code);

```
Call: gpu_open  (return value, boolean) on the file code_area is
----            the codesegment(s) given by the name code searched
                and loaded to the zone_buffer z. Then is the zone
                opened to the process gpu_name or gpu_dummy dependent
                of gpu_emulat. The procedure will load code of any
                length roomed in the zonebuffer.
                result := code found else false.
      z         (call and return value, zone) ready for open. At
                return ready for out_rec.
      gpu_name  (call value, long array). name of gpu_process.
      code_area (call value, long array). name of code file.
      code      (call value, short string). name of code to be
                searched for on last segment of code in long
                field 512.
```

Leading segments of code_area are skipped until the segment having
code_area(1) = long z(127) and code_area(2) = long z(128)
are skipped (inclusive).

Integer field 2 of codes on codearea must contain the length of the code in bytes. A zero is replaced by 512.

gpu_open will set the escape-mask used to emulate the gpu-instructions, when gpu does not exist.

A hard error during the actual execution of code will cause a break due to the fact that some code has been executed and the recovery will be very difficult if not impossible.

NOT ALGOL

The communication may be handled in a non standard way by the message - answer monitor calls. (see outrec).

4. Close

ALGOL

The communication by GPU is terminated by :

```
change_rec_6(code, 0); <* to ensure that no code is executed *>
close(code, true);
```

NOT ALGOL

If no zone has been declared and opened there is no zone to close.

5. Out_rec

ALGOL

After open, set_position, change_rec_6 or out_rec_6 a new block of code may be loaded to the gpu by a call of the procedure

```
          out_rec_6(code, length);
```

When the code is ready to be executed this may bee done in several ways :

```
   if change_wanted then change_rec_6(code, no_of_code_bytes);
   if pos_wanted then set_position(code, 0, 0) else
   if close_wanted then close(code, true) else
                    out_rec_6(code, length);
```

NOT ALGOL

When a codearea has been created then it must be sent to the GPU by a message ad then GPU must be waited for by a wait_answer.

The message buffer must contain the description of the codearea :

```
    mess +  0 :  next buffer
    mess +  2 :  previous buffer
    mess +  4 :  receiver or answer type
    mess +  6 :  sender
    mess +  8 :  operation = 5 < 12 <* write *>
    mess + 10 :  first code (see below)
    mess + 12 :  last code
```

GPU starts execution of code in word (first_code + 4)
++

The first 8 words are set by the monitor procedure.
first_code and last_code are absolute addresses of the codearea.

6. Block-procedure gpu_error

The block procedure will handle any error-situation from the GPU.

If the GPU is disconnected and gpu_emulat permits it will the code be emulated by the block procedure, - but in single precission. The runtime will increase drastically.

If the GPU is disconnected during an execution of a code send from a process the process is breaked with an alarm and a message is sent to the operator.

The status bits from the GPU are :

bit no.	value	description
0	1 < 23	intervention
1	1 < 22	illegal instruction
2	1 < 21	time out
3	1 < 20	bus error
4	1 < 19	read/write address fault
5	1 < 18	floating point overflow *)
6	1 < 17	floating point underflow **)
7	1 < 16	integer overflow *)
8	1 < 15	integer underflow *)
9	1 < 14	code address fault ***)
10-23		see ref. 1.

*) controlled by the statusbits of the senders process. e.g. integer underflow is only active when the senders process has set the status : integer exception active.

**) in case of floating point underflow the result is set to floating point zero and no exception is set up.

++++++++++++

***) the next instruction to be executed is fetched from an address which lies outside the limits code_low and code_top - or the codearea-limits code_low and code_top are not found inside the write-limits llim and ulim of the senders process.

7. Gpu_exec

The external procedure gpu_exec emulates the gpu by executing the code.

Gpu_exec is called from the blockprocedure if emulation is allowed, see gpu_emulat.

The procedure may be called directly itself during debugging of the code to the GPU.

The execution of the special instructions is a little slow because the instructions are executed through the escape routine.

integer procedure gpu_exec(code).

gpu_exec (return value, integer) the number of bytes processed.

code (call and return value, real array) the code to be emulated.

code 64 = gpu_red will not be emulated.

The execution is locked in not_blockfloating mode.

8. Gpu_emulat

Own integer control parameter used by the blockprocedure gpu_error.

The first two bits are controlbits and the last 22 are the number of emulations.

gpu_open uses the first 2 bits of gpu_emulat :
- = 3 : open is to gpu_dummy. i.e. forced emulation.
- <> 3 : open is to gpu_name. i.e. the GPU is used.

gpu_error uses the first 2 bits of gpu_emulat :
- = 0 : no emulation. i.e. break process.
- <> 0 : emulation. (unless the error situation is unrepairable).

9. Code

The GPU is a rebuilt RC 8000/35 which executes the normal RC 8000 instructions with some exceptions.

A sequence of instructions may be translated by slang into a code-file which may be input by an application program.

Example 1:

```
begin
  zone code(128, 1, std_error),
       gpu(128, 1, gpu_error);

  open(code, 4, <:codefile:>, 0);
  open(gpu , 0, <:gpul:>,     -2);

  in_rec_6(code, 512);
  out_rec_6(gpu, 512);
  to_from(gpu, code, 512);
  close(code, true);
  close(gpu,  true);

end;
```

A sequence of instructions could also be placed in a codeprocedure, which moves the sequence to a given zone.

The code may be placed after the gpu_exec segments and translated at the same time as that procedure (see the text gpu_exectx in the contract file gpu_proc_file.

The instructions must follow some simple rules :

1) The first instruction to be executed must be in byte 4 : code_low + 4.
2) The instructions must be within the given limits code_low = first of buffer and code_high = code_low + (length_of_buffer).
3) No privileged instructions may be executed except the special GPU-instructions.
4) The first word must contain the length in bytes (multiple of 512) when gpu_open is used.

An easy way of composing a code is to use the gpu_declaration_block and follow the instructions given in page 19 ff.

10. Description of the gpu-instructions

```
the doubleword x (value real) is named : dwd(addr),
                                          where addr = absaddr(x)
modus_sign is set by gpu_init.
modus_blfl is set by gpu_init.
```

gpu_add (= 30)

```
ar := if addr = 0 then
_       (if modus_sign = 0 then (AR + (dwd(w) con 0.0))
_                              else (AR - (dwd(w) con 0.0)) ) else
_       (if modus_sign = 0 then (AR + (dwd(w) con dwd(w+addr)) )
_                              else (AR - (dwd(w) con dwd(w+addr)) ) );
```

gpu_arm (= 59)

```
AR := AR * dwd(addr);
```

gpu_cmv_a (= 47)

```
if w0 > 0 then
begin
  real x;
  x := dwd(addr);
  if x <> 0.0 then
  begin
    repeat
      AR     := + ( x * dwd(w_pre) ) + dwd(w);
      dwd(w) := AR;
      w_pre  := wpre + 4;
      w      := w    + 4;
      w0     := w0   - 1;
    until w0 = 0;
  end;
end;
```

gpu_cmv_s (= 31)

```
if w0 > 0 then
begin
  real x;
  x := dwd(addr);
  if x <> 0.0 then
  begin
    repeat
      AR     := - ( x * dwd(w_pre) ) + dwd(w);
      dwd(w) := AR;
      w_pre  := w_pre + 4;
      w      := w     + 4;
      w0     := w0    - 1;
    until w0 = 0;
  end;
end;
```

```
gpu_init    (= 63)
++++++++++++++++

begin
  integer md;
  md := word(addr);
  modus_sign := (md shift (-22)) extract 1;
  comment 0: add,  1: sub  ;
  modus_blfl := (md shift (-21)) extract 1;
  comment  0: normalizing mode,  1:  fix_point_mode with block_exp;
  _         see chapter 12;
  block_exp := md extract 12;
  if block_exp > 2047 then block_exp := ((-1) shift 12) + block_exp;
  if md shift (-23) extract 1 = 1 then AR := 0.0;
end;


gpu_inv     (= 60)
++++++++++++++++

begin
  comment the content of w_pre is destroied;
  dwd(w) := 1 / dwd(addr);
end;


gpu_mla     (= 58)
++++++++++++++++

if w0 > 0 then
begin
  repeat
    AR := if modus_sign = 0 then
    _      (AR + dwd(w_pre) * dwd(w)) else
    _      (AR - dwd(W_pre) * dwd(w));
    w_pre := w_pre + 4;
    w     := w     + 4;
    w0    := w0    - 1;
  until w0 = 0;
end;


gpu_red     (= 62)
++++++++++++++++

begin
  comment cholesky-reduction of a datamatic block of columns.

  The instruction must use w3 as regw and w1 as regx and the
  address must point to address c5 in the working area :
            gpured  w3  x1   c5.
  w1 need not be zero as it is subtracted from the address.

  In block_floating mode c6 and b8 is pointing the exponents
  which should be subtracted from the unreduced elements exponent
  to make the element a block_floating number with exponent equal
  zero (see chapter 13).
```

```
in normal mode c6 and b8 is dummy.

working area :
 b0   =  base of working area :  CAT_I1U + nlu_base
 c4   = b0 +  2  : AUTORED
 c3   = b0 +  4  : LR = last_reduced column
 b7   = b0 +  6  : (R_max + 1) Helmert block_red_limit.
 n2   = b0 +  8  : SZU = Saved Zeroes in Unreduced coulmn
 c1   = b0 + 10  : CAT_SZR + nlr_base, base of Saved Zeroes of
                   Reduced columns.
 c0   = b0 + 12  : CAT_I1R + nlr_base, index of first nonzero
                   element in reduced column.
 n11  = b0 + 14  : 4 * (R_MAX + 1) = 2 * word(b7)
 b5   = b0 + 16  : tail_displacement.
 c5   = b0 + 18  : accu_mode :
                   accumode(2) =
                          if block_floating mode then 1 else 0.
                   accu_mode(12:23) = if pos_accumulation then 0
                                      else ((-1) extract 12)
                                           <* neg_accumulation *>.
 c6   = b0 + 20  : block_exp_r + nlr_base, base of catalog on
                   exponents of reduced columns to be subtracted
                   from the exponent of the element under reduc-
                   tion to make it be a block_floating number.
 n13  = b0 + 22  : block_exp_u , to be subtracted from the
                   exponent of the element under reduction.
 b8   = b0 + 24  : block_exp_u + nlu_base, base of catalog on
                   exponents of unreduced columns to be e.t.c.
 b6   = b0 + 26  : exp_lim, max acceptable loss of binals.
 b4   = b0 + 28  : status + nlu_base, base of catalog of status-
                   area of unreduced elements.
 n0   = b0 + 30  : U, index of unreduced column.
 b1   = b0 + 32  : cat_szu + nlu_base, base of catalog of saved
                   zeroes of unreduced columns
 c2   = b0 + 34  : FR, first index of reduced column in datamatic
                   block.
 b3   = b0 + 36  : LU, last index of unreduced column in datamatic
                   block.
 b2   = b0 + 38  : FU, first index of unreduced column in data-
                   matic block.

Only the content of c5 is changed at return.
++++++++++++++++++++++++++++++++++++++++++

The b-working_locations refer to unreduced block.
The c-working_locations refer to reduced block.
The n-working_locations are pure working_locations used by the
microprogram.

;

<* check the registers used in call of bf *>
if reg_x <> w1 or reg_pre <> w2 or reg_w <> w3 then illop;
```

```
for col := fu step 2 until lu do
begin

  if auto_red then lr := col;
  szu := nlu.cat_szu.col * 2;
  if szu <= (R_max + 1) then
  begin

    if bl_fl_mode then exp_u := nlu.cat_exp_u.col;

    for row := fr step 1 until lr do
    begin

      szr := nlu.cat_szr.row * 2;
      if szr < szu then szr := szu;

      if szr < (R_max + 1) then
      begin

        szr := szr + szr;
        reg_pre := nlr.cat_ilr + szr;
        reg_w   := nlu.cat_ilu + szr;

        ful_red := row*2 < (rmax+1)*2;
        w0 := (if ful_red then (2*row - szr)
              else ((R_max+1)*2 - szr) ) // 4;

        wl := (1 shift 23) add
              ((accu_mode extract 1) shift 22) add
              (((accu_mode shift (-12)) extract 1) shift 21);
        gpu_init(wl);

        gpu_mla(w0,w);
        dia := reg_pre = reg_w;

        if bl_fl_mode then red_exp := nlu.cat_exp_u.row + exp_u
        else red_exp := 0;

        u_f := nlu.cat_ilu.row + row + row + tail_disp;
        x_t := nlu.u_f;
        if sign_extend(x_t extract 12) > -2048 or
        _  sign_extend(x_t extract 12) - red_exp > -2048 then
        begin
          u_f := u_f - tail_disp;
          x_h := nlu.u_f;
          x_h := ((x_h shift (-12)) shift 12) add
          _  ((sign_extend(x_h extract 12) - red_exp) extract 12);
          if accu_mode extract 12 = 0 then gpu_add(x_h, x_t)
          _  else gpu_sub(x_h, x_t);
        end
        else
        _    x_h := 0.0;
```

```
          if full_red then
          begin
            if dia then
            begin
              wl := 0;
              gpu_init(wl);
              gpu_str(w, 0);

              if reg_pre <= 0 then
              begin
                s_f := status + 2 * col;
                nlu.s_f := (real <::> add
                           (if reg_pre = 0 then 4 else 3)) shift 24;
              end
              else
              begin
                exp_loss := (sign_extend(x_h extract 12)) -
                            (sign_extend(reg_w extract 12));
                s_f := status + col + col;
                nlu.s_f := ((real <::> add
                           (if exp_loss >= exp_lim then 3 else 1)
                           ) shift 24) add exp_loss;
                if exp_loss < exp_lim then
                begin
                  gpu_sqrt(reg_w);
                  wl := (if bl_fl_mode then 1 else 0) shift 21;
                  gpu_init(wl);
                  gpu_inv(reg_w, nlu.ilu+col+col);
                end
                else
                begin
                  u_f := nlu.ilu + col + col;
                  nlu.u_f := 0.0;
                end;
              end;
            end
            else
            begin
              gpu_arm(reg_pre);
              u_f := nlu.ilu + col + col;
              gpu_str(nlu.u_f, 0);
            end;

          end
          else
          begin
            wl := 0;
            gpu_init(wl); <* normal mode *>
            u_f := nlu.cat_ilu.col + col + col;
            gpu_str(nlu.u_f, tail_disp);
          end;
        end row-loop;
      end col-loop
    end;
  end;
end;
```

```
gpu_sqrt    (=  1)
++++++++++++++++

begin
  comment w_pre, w := sqrt( dwd(addr) );
  w_pre con w := sqrt( dwd(addr) );
end;


gpu_stop    (=  0)
++++++++++++++++

begin
  comment stop execution of user program in gpu and return
_         status to host and user;
end;


gpu_str     (= 61)
++++++++++++++++

begin
  comment if the effective address is zero then store AR in dwd(w)
          else store AR as double floating point number in
          dwd(w) con dwd(w+addr).
  AR is unchanged;

  if addr = 0 then
  dwd(w) := float_rounded_to_36_bit(
_           if -, bl_fl_mode then normalize(AR)
_                            else bl_fl_norm(AR))
  else
  dwd(w) con dwd(w+addr) := float_rounded_to_71_bit(
_           if -, bl_fl_mode then normalize(AR)
_                            else bl_fl_norm(AR));
end;


gpu_sub     (= 31)
++++++++++++++++

AR := if addr = 0 then
_      (if modus_sign = 0 then (AR - (dwd(w) con 0.0))
_                          else (AR + (dwd(w) con 0.0)) ) else
_      (if modus_sign = 0 then (AR - (dwd(w) con dwd(w+addr)) )
_                          else (AR + (dwd(w) con dwd(w+addr)) ) );


ks          (= 51)
++++++++++++++++

no_op;
```

11. Description of the gpu_declaration_block

The gpu_declaration_block "gpu_names" is copied into the code by the directive :

```
p.<:gpunames:>
```

and the code must be supplemented by an end-directive to the gpu_declarations_block :

```
e.
```

before the last end-directive.

The names are :

```
u0   =   w0
u1   =   w1  ;   u5  =  x1
u2   =   w2  ;   u6  =  x2
u3   =   w3  ;   u7  =  x3

u8   =   indirect
u9   =   relative
u10  =   indirect and relative

u11  =   gpu_add
u12  =   gpu_arm
u13  =   gpu_cmv_a

u14  =   gpu_cmv_s
u15  =   gpu_init
u16  =   gpu_inv

u17  =   gpu_mla
u18  =   gpu_red
u19  =   gpu_sqrt

u20  =   gpu_stop
u21  =   gpu_str
u22  =   gpu_sub
```

12. Construction of the special gpu_instructions.

An instruction consists of an operation and an adress.

To ensure a correct address calculation a special construction must be used at least by relative addresses :

```
   a)      (:uc + uw + ux + ua:)<12 + abs_addr

   b)      (:uc + uw + ux + ua:)<12 + (:(:addr.:)<12:)>12

   c)      a0 = addr.
           h.
           uc + uw + ux + ua, a0
           w.

   where uc is instruction (11 <= c <= 22)
         uw is word        ( 0 <= w <=  3)
         ux is index       ( 5 <= x <=  7 or ux = 0)
         ua is addressmode ( 8 <= a <= 10)
```

Note that the address must not be negative and less than 4095 else the command part is destroyed.

When a negative relative address is used then is it recommended to use construction c with ua = if indirect then u10 else u9).

13. Block_floating mode.

Block_floating mode for the gpu is a kind of fixed_point arithmetic.

The instruction gpu_init delevers the modus bits and if needed the block_floating exponent.

Block_floating mode is set by : modus := 1 shift 21;
Block_floating_exponent is taken from : modus extract 12;

In block_floating mode will the floating point stored by gpu_inv and gpu_str be stored with an exponent equal to or bigger than the block_floating_exponent, -
and all arithmetic on AR is followed with
no normalization shifts at all !!!
****

IT IS RECOMMENDED TO USE ZERO AS BLOCK_FLOATING_EXPONENT
++

Examples :
block_floating_exponent = 0;

number	is represented (octal)	with exponent
0.5	2000.0000.0000	0
1.0	2000.0000.0000	1
0.25	1000.0000.0000	0
2.0	2000.0000.0000	2
0.125	0400.0000.0000	0

It may then in block_floating_mode be nescessary to scale all numbers with a known size by subtraction or addition to the expo_nent of the number.

In solution of normal equations this looks like :

```
A transposed = AT
obs-eq :     A * X = B
norm-eq :    AT * A * X = AT * B
scale_factors : diagonal matrix S  (n*n)
transformation :
          AT * A * X = AT * B   from right and left by S
        S*AT * A*S * X = S*AT * B*S
             N     * X = S*AT * B*S
scale_factor to right-hand-side : R.
                 N * X = S*AT * B*S*R * (1/R)
                 N * X =      W * (1/R)
triangulated    NR * X = WR * (1/R)
back_solution        X = (NR**(-1)) * (WR * (1/R))
```

14. Examples.

example 1 : code placed in bs_area gpu_code
=========== product_sum result stored in a result area

```
gpu_code = slang

s. a20, b0, w.

    0                    ; fill (evt. length)
    0                    ; fill (evt. check_sum)

   jl.        b0.        ; goto code

a1: 1 < 23               ; modus = clear AR
a2: 0                    ; addr a
a3: 0                    ; addr b
a4: 0                    ; repetitions
a5: 0                    ; str addr
a6: 0                    ; displacement

a20=a1.
h.
b0: 63<6 + 2<2 , a20 ; clear AR
w.
   dl. w2     a3.        ; load addr
   rl. w0     a4.        ; load repetitions
h.
58<6 + 2<4 + 0<2, 0  ; mla w2 ;
w.
   rl. w3     a5.        ; w3 := str addr
a19=a6.
h.
61<6 + 3<4 + 3<2, a19; str. w3 (a6.)
w.
   0                     ; gpu_stop
i.
e.

; tail is missing. (see ref. 5).
```

end of example 1.
================

example 2 : a code_procedure moves the code to a zone
===========

```
gpuinit = set 1
scope login gpuinit

(
if 9.yes
(gpuinit = slang names.yes list.yes xref.yes entry.no
gpuinit)
if 9.no
(gpuinit = slang entry.no
gpuinit)
lookup gpuinit
end
)


b.
d.

p.<:fpnames:>

l.

b. g1, e5
w.

s. a20, b20, j30, f20, g6

h.

g0 = 0                               ; no of extrn.

e5:

g1:     g2,    g2       ; rel of last point, rel of last abs word

j13:     g0+13,  0    ; RS entry 13, last used
j30:     g0+30,  0    ;      -    30, save stack ref, saved w3
j17:     g0+17,  0    ;      -    17, index alarm
j6:      g0+ 6,  0    ;      -     6, end register expression

g2 = k-2-g1
w.
e0:     g0
        0
        s3,  date
        s4,  time
e4:
```

```
w.

a1 = e5 + 6    ; modus
a2 = a1 + 2    ; addr a
a3 = a2 + 2    ; addr b
a4 = a3 + 2    ; repetitions
a5 = a4 + 2    ; str addr
a6 = a5 + 2    ; displacement


; code for check and init of code in zone z,
e1:
        rl. w2    (j13.)      ; get lastused
        ds. w3    (j30.)      ; save stackref and alarm addr

; check params
        dl  w1   x2+8         ;
        rl  w3   x1+h3        ; record base
        rl  w1   x1+h3+4      ; record length
        sl  w1     b1-1       ; if record length < code length
        jl. w3    (j17.)      ; then goto index alarm


; init code
f1:     al. w1     e5.-2      ; first of code
        ws  w3     2          ;  w3 := rel addr of zone buf
        rs. w3     f1.        ;
        al  w1   x1+6         ; first to store
        dl. w0     f2.        ; jump instr
        am.       (f1.)       ; zone displ
        ds  w0   x1           ;

; move code
f0:     al  w1   x1+4         ; increase pointer
        dl  w0   x1           ; load code
        am.       (f4.)       ; displ of zone
        ds  w0   x1           ; store in zone
        sh. w1     g3.        ;
        jl.        f0.

; return to RS
        dl. w3   (      j30.) ; restore stackp and return
        jl.      (      j6.)  ;   end reg expression

; work for code loader
f1:     0
f2:     0
f3:      jl.     f4           ; entry instr

c.      a6 - 1 - f3
       aw       0, r. (:a6 - f3:) > 1
z.
```

```
b0:
f4 = b0 - e5 - 4

a20 = a1.
h.
     63<6 + 2<2 ,   a20    ; clear AR
w.
     dl. w2       a3.      ; load addr
     rl. w0       a4.      ; load rep
     58<18+ 2<16 + 0       ; mla w2
     rl. w3       a5.      ; w3 := str addr
a19 = a6.
h.
     61<6 + 3<4 + 3<2, a19 ; str. w3 (a19.)
w.


w.
         aw           0, r. 2

g3:

c. g3 - g1 -506
  m.code on segm too long
z.

c. 502 - g3 + g1, j1 -1, r. 252 -(:g3-g1:)>1  z.

<:gpuinit<0><0><0>:>       ; alarm text

i.
e.       ; end of slang segm



g0:g1:    1                ; first tail, last tail
       0,0,0,0             ; name
       1<23 + e1 - e5      ; entry point for nllinitdpu
       2<18 + 8<12,0       ; spec bool proc, zone
       4<12 + e0 - e5      ; code proc start of extnr list
       1<12 + 0            ; 1 code segm, 0 bytes in core

d.

p.<:insertproc:>
e.
e.
e.

end of example 2.
================
```

```
example 3 : call of gpu.
===========

real procedure scalar_product(a, b, length, res_tail_addr);
value                                length, res_tail_addr;
real array                     a, b;
integer                              length, res_tail_addr;
<* if res_tail_addr equals zero then is the result rounded
_  to a normal real else is the result rounded to a double
_  real result *>
begin
  zone code(128, 1, std_error),
  _    gpu(128, 1, gpu_error);
  real res;
  integer array field word;

  comment a procedure abs_addr(x), where x is type general
  _       is called. the result is an integer giving the
  _       absolute adress of th simple variable x or the
  _       absolute adress of the first element of an array
  _       variable x;

  open(gpu, 0, <:gpul:>, -3);
  out_rec_6(gpu, 512);

  if examplel then
  begin
    open(code, 4, <:gpucode:>, 0);
    in_rec_6(code, 512);
    to_from(gpu, code);
    close(code, true);
  end
  else <* example 2 *>
    gpu_init(gpu);

  word          := 2;
  gpu.word(4)   := abs_addr(a);
  gpu.word(5)   := abs_addr(b);
  gpu.word(6)   := n; <* length of vector *>
  gpu.word(7)   := abs_addr(res);
  _                <* tail displacement = *>
  gpu.word(8)   := if res_tail_addr = 0 then 0 else
  _                res_tail_addr - gpu.word(7);

  close(gpu, true);

  scalar_product := res;

end scalar product;

end of example 3.
=================
```