A C Simulator for the EnLight256

Michael D. Vose
Department of Electrical Engineering and Computer Science
University of Tennessee
Knoxville, TN 37996-3450
vose@eecs.utk.edu

Introduction

This document describes an open source simulator, written in C, for the EnLight256. It enables a user to write a C program containing embedded EnLight256 commands, compile and execute that program, and obtain the same results as if actual EnLight256 hardware was present.[1]

In addition, the simulator implements tracing and logging functionality which facilitates debugging application code. The simulator plays well with gdb based debuggers, being acquiescent in a debugging environment in which the application – not the simulator – is being debugged. Moreover, the simulator is instrumented to generate reports concerning (the simulated) hardware usage, and the expected execution time for an application running on (the simulated) hardware.

The simulator also contains methods both for modifying the functionality of the underlaying hardware being simulated, and for extending the API. That allows for variations (of the EnLight256) to be specified, and provides a sandbox for the exploration, development, and validation of algorithms which would be particularly well suited to future architectures in which matrix/vector operations complete in a single machine cycle.

The simulator was developed and tested on a gnu/linux system (Linux version 2.6.10-gentoo-r6) using gcc (3.3.2 20031218) on Intel hardware (Pentium(R) 4 CPU 1700MHz).

Overview

The simulator is comprised of two files, `machine.c` and `simulate.c`, both of which consist mostly of macros. The total compressed size (bzip2) is under 10KB. The file `machine.c` implements an abstract machine whose register structure coincides with that of the EnLight256. The file `simulate.c` (whose first line is `#include "machine.c"`) layers the EnLight API over the abstract machine and implements tracing, logging, and extension functionality (rudimentary tracing is implemented in `machine.c`).

To use the simulator, simply insert `#include "simulate.c"` as the first line in the application's C source file, and place both `machine.c` and `simulate.c` in the directory containing that source. No special compilation is required; use whatever would be appropriate for the application, as if EnLight's API were known

[1] Caveat: actual hardware was unavailable; the simulator was written based on LENSLET's documentation (EnLight256 Bit Exact Simulator User Manual Release 1.1). Validation of simulation results was with respect to the Matlab based simulator provided by LENSLET which runs only in a Microsoft Windows environment.

to the compiler (caveat: since EnLight uses polymorphic functions, whereas the simulator does not, a variation of the API is actually implemented; see the Appendix for both variations and extensions).

Simple examples of using the simulator are presented in the section on usage, including detailed information concerning the statistics, tracing, logging, and extension functionality. Source code is given in the appendix.

Abstract Machine Architecture

Types

This section describes the abstract architecture of the machine being simulated and its API. To streamline exposition, the following types will be used:

Short	=	$\{-128, \ldots, 127\}$	8-bit signed character (2s complement)
Long	=	$\{-32768, \ldots, 32767\}$	16-bit signed integer (2s complement)
Vector_s	=	$Short^{256}$	256 element vector of *Short*
Vector_l	=	$Long^{256}$	256 element vector of *Long*
Matrix	=	$Short^{256\times256}$	256×256 matrix of *Short*

Registers

The register set consists of three sorts:

1. Sixteen instances of *Vector_s* : $S_0 \ldots S_{15}$.
2. Eight instances of *Vector_l* : $L_0 \ldots L_7$, where for all i and j,

   the low-order byte of $L_i[j]$ is $S_{2i}[j]$, and the high-order byte of $L_i[j]$ is $S_{2i+1}[j]$.
3. Four instances of *Matrix* : $M_0 \ldots M_3$.

Semantics

For purposes of this document, a scalar is regarded as a bit vector containing n bits (either 8 or 16). Scalars are of three kinds: *unsigned, signed, fractional*. A scalar's kind is distinct from the *Short* or *Long* type which the scalar is an instance of; it is an attribute — in some ways similar to a *C* type cast — by which the semantics (i.e., mathematical value) of the scalar is determined.

Let scalar **b** have bits $b_0 \ldots b_{n-1}$. To say **b** is *unsigned* is to indicate it's semantics is

$$u = \sum_i b_i 2^i$$

To say **b** is *signed* is to indicate it's semantics is

$$s = s_n(u) = \begin{cases} u & \text{if } u < 2^{n-1} \\ u - 2^{n-1} & \text{if } u \geq 2^{n-1} \end{cases}$$

Note that the function inverse to s_n is $u_n(s) = (s + 2^n)$ modulo 2^n. To say **b** is *fractional* is to indicate it's semantics is

$$f_n(s) = 2^{1-n}s$$

Every (n bit) fractional f satisfies $-1 \leq f \leq 1 - 2^{1-n}$. Let signed scalers **b** and **b′** have semantics s and s' respectively. Note that

$$f_n(s)f_n(s') = 2^{2-2n}ss' = 2^{1-2n}(2ss') = f_{2n}(2ss')$$

It follows that if one regards **b** and **b′** as fractional, having semantics f and f' respectively, then

$$ff' = f_{2n}(2ss')$$

In particular, the product of fractional operands — interpreted as a fractional result — is represented by a bit string (of twice the length) which coincides with the representation of twice the product of their signed interpretations.

Because the abstract machine is precision limited, results too large for (the number of bits comprising) a data type must somehow be reduced/limited. Assuming x is a signed integer, language like "*x is hard-limited*" (to n bits) refers to $H_n(x)$ defined as

$$H_n(x) = \begin{cases} x & \text{if } x \in \{-2^{n-1}, \ldots, 2^{n-1} - 1\} \\ -2^{n-1} & \text{if } x < -2^{n-1} \\ 2^{n-1} - 1 & \text{if } x > 2^{n-1} - 1 \end{cases}$$

If $x = ff'$ is the fractional result of multiplying n-bit fractional operands, language like "*x is hard-limited to n bits*" refers to $f_n(H_n(\lceil 2^{n-1}x \rfloor))$, where $\lceil \cdot \rfloor$ denotes truncation (the result of rounding $\cdot$ towards zero). The signed scalar **b″** (with semantics s'') representing the hard-limited result satisfies

$$f_n(s'') = f_n(H_n(\lceil 2^{n-1}x \rfloor)) \iff s'' = H_n(\lceil 2^{1-n}ss' \rfloor)$$

The number n of bits in the hard-limited result is inferred from context; if the result is Short then n = 8, if the result is Long then n = 16.

Instructions

Implemented EnLight commands (including I/O) and their semantics are described below. Simulator instructions relating to tracing, logging, and extension functionality are covered in the section on usage.

Naming conventions

- s_s (or s'_s) refers to a source register and denotes the index i (or i') of register S_i (or $S_{i'}$).
- d_s (or d'_s) refers to a destination register and denotes the index i (or i') of register S_i (or $S_{i'}$).

- s_l (or s'_l) refers to a source register and denotes the index i (or i') of register L_i (or $L_{i'}$).
- d_l (or d'_l) refers to a destination register and denotes the index i (or i') of register L_i (or $L_{i'}$).
- s_m refers to a source register and denotes the index i of register M_i.
- d_m refers to a destination register and denotes the index i of register M_i.
- t_s refers to a temporary register and denotes the index i of register S_i.
- t_l refers to a temporary register and denotes the index i of register L_i.
- i_s denotes a Short immediate value.
- i_l denotes a Long immediate value.

Whenever any of the above (s_s, s_l, s_m, d_s, ...) appears as a subscript in a register name, the third subscript is omitted (hence S_{s_s} is abbreviated to S_s), since the third subscript carries redundant type information (the register name determines type). In the following descriptions, C operators applied to registers act component-wise. All arguments are numeric and specified by base ten integers (a fractional operand f is represented by the signed integer s such that $f = f_n(s)$, where n is determined from context).

APL_AND(s_s, s'_s, d_s)	: $S_d = S_s \,\&\, S_{s'}$	
APL_COPY(s_s, d_s)	: $S_d = S_s$	
APL_COPY16(s_l, d_l)	: $L_d = L_s$	
APL_LSL(s_s, d_s)	: $S_d = S_s$<<1	
APL_LSR(s_s, d_s)	: $S_d = S_s$>>1	unsigned logical shift
APL_NOT(s_s, d_s)	: $S_d = \tilde{}\,S_s$	
APL_OR(s_s, s'_s, d_s)	: $S_d = S_s \,\vert\, S_{s'}$	
APL_SADD(i_s, s_s, d_s)	: $S_d = i_s + S_s$	hard-limited
APL_SADDM(i_s, s_l, d_l)	: $L_d = i_s + L_s$	hard-limited
APL_SAND(i_s, s_s, d_s)	: $S_d = i_s \,\&\, S_s$	
APL_SCIA(i_s, s_s, d_s)	: $S_d[j] = S_s[j]$	j even
	; $S_d[j] = i_s + S_s[j]$	j odd, hard-limited
APL_SCIV(i_s, d_s)	: $S_d[j] = 0$	j even
	; $S_d[j] = i_s$	j odd
APL_SCOPY(i_s, d_s)	: $S_d = i_s$	
APL_SCRA(i_s, s_s, d_s)	: $S_d[j] = i_s + S_s[j]$	j even, hard-limited
	; $S_d[j] = S_s[j]$	j odd
APL_SCRV(i_s, d_s)	: $S_d[j] = i_s$	j even
	; $S_d[j] = 0$	j odd
APL_SHFT_D$(9, d_s)$	: $S_d = S_9$	
	; $S_9[j] = S_9[j+1]$	$j = 0, 1, \ldots$ where $S_9[256] = 0$

APL_SHFT_D2$(8, 9, d_s)$	: $S_d = S_9$	
	; $S_8[j] = S_8[j+1]$	$j = 0, 1, \ldots$ where $S_8[256] = S_9[0]$
	; $S_9[j] = S_9[j+1]$	$j = 0, 1, \ldots$ where $S_9[256] = 0$
APL_SHFT_TRF$(s_s, 8)$	: $S_8 = S_s$	
APL_SHFT_TRF$(s_s, 9)$	: $S_9 = S_s$	
APL_SHFT_U$(9, d_s)$	: $S_d = S_9$	
	; $S_9[j] = S_9[j-1]$	$j = 255, 254, \ldots$, where $S_9[-1] = 0$
APL_SHFT_U2$(8, 9, d_s)$	: $S_d = S_9$	
	; $S_9[j] = S_9[j-1]$	$j = 255, 254, \ldots$ where $S_9[-1] = S_8[255]$
	; $S_8[j] = S_8[j-1]$	$j = 255, 254, \ldots$ where $S_8[-1] = 0$
APL_SMUL(i_s, s_s, d_l)	: $L_d = i_s * S_s$	hard-limited, fractional operands
APL_SMUR(i_s, s_s, d_s)	: $S_d = i_s * S_s$	hard-limited, fractional operands
APL_SOR(i_s, s_s, d_s)	: $S_d = i_s \mid S_s$	
APL_SSUB(i_s, s_s, d_s)	: $S_d = i_s - S_s$	hard-limited
APL_SSUBM(i_s, s_l, d_l)	: $L_d = i_s - L_s$	hard-limited
APL_SXOR(i_s, s_s, d_s)	: $S_d = i_s \hat{} S_s$	
APL_VABS(s_s, d_s)	: $S_d = \mathrm{abs}(S_s)$	hard-limited
APL_VABS16(s_l, d_l)	: $L_d = \mathrm{abs}(L_s)$	hard-limited
APL_VADD(s_s, s'_s, d_s)	: $S_d = S_s + S_{s'}$	hard-limited
APL_VADD16(s_l, s'_l, d_l)	: $L_d = L_s + L_{s'}$	hard-limited
APL_VADDM(s_l, s'_s, d_l)	: $L_d = S_{s'} + L_s$	hard-limited
APL_VASL(s_s, d_s)	: $S_d = S_s$<<1	hard-limited
APL_VASL16(s_l, d_l)	: $L_d = L_s$<<1	hard-limited
APL_VASR(s_s, d_s)	: $S_d = S_s$>>1	signed arithmetic shift
APL_VASR16(s_l, d_l)	: $L_d = L_s$>>1	signed arithmetic shift
APL_VCCONJ(s_s, d_s)	: $S_d[j] = S_s[j]$	j even
	; $S_d[j] = -S_s[j]$	j odd, hard-limited
APL_VCCONJ16(s_l, d_l)	: $L_d[j] = L_s[j]$	j even
	; $L_d[j] = -L_s[j]$	j odd, hard-limited;
APL_VCMUL(s_s, s'_s, d_l)	: $L_d[j] = S_s[j] * S_{s'}[j+1]$	j even, hard-limited, fractional operands
	; $L_d[j] = S_s[j] * S_{s'}[j-1]$	j odd, hard-limited, fractional operands
APL_VCMUR(s_s, s'_s, d_s)	: $S_d[j] = S_s[j] * S_{s'}[j+1]$	j even, hard-limited, fractional operands
	; $S_d[j] = S_s[j] * S_{s'}[j-1]$	j odd, hard-limited, fractional operands
APL_VCOGE(s_s, s'_s, d_s)	: $S_d = ((S_s \geq S_{s'})?\ -1 : 0)$	
APL_VCOGE16(s_d, s'_d, d_d)	: $S_d = ((S_s \geq S_{s'})?\ -1 : 0)$	
APL_VCOMP(s_s, s'_s, d_s)	: $S_d = ((S_s == S_{s'})?\ 0 : ((S_s > S_{s'})?\ 127 : -128))$	
APL_VCOMP16(s_l, s'_l, d_l)	: $L_d = ((L_s == L_{s'})?\ 0 : ((L_s > L_{s'})?\ 32767 : -32768))$	

APL_VCRAI(s_s, d_s)	: $S_d[j] = 0$	j even
	; $S_d[j] = S_s[j] + S_s[j-1]$	j odd, hard-limited
APL_VCRAI16(s_l, d_l)	: $L_d[j] = 0$	j even
	; $L_d[j] = L_s[j] + L_s[j-1]$	j odd, hard-limited
APL_VCRSI(s_s, d_s)	: $S_d[j] = S_s[j] - S_s[j+1]$	j even, hard-limited
	; $S_d[j] = 0$	j odd
APL_VCRSI16(s_l, d_l)	: $L_d[j] = L_s[j] - L_s[j+1]$	j even, hard-limited
	; $L_d[j] = 0$	j odd
APL_VEIM(s_s, d_s)	: $S_d[j] = 0$	j even
	; $S_d[j] = S_s[j]$	j odd
APL_VERE(s_s, d_s)	: $S_d[j] = S_s[j]$	j even
	; $S_d[j] = 0$	j odd
APL_VMM$(s_m, 13, 12, i_s)$	; $S_{12} = (M_s\, S_{13})$<<i_s	matrix multiply and shift, fractional operands
APL_VMUJ(s_s, d_s)	: $S_d[j] = -S_s[j+1]$	j even, hard-limited
	; $S_d[j] = S_s[j-1]$	j odd
APL_VMUL(s_s, s'_s, d_l)	: $L_d = S_s * S'_s$	hard-limited, fractional operands
APL_VMUR(s_s, s'_s, d_s)	: $S_d = S_s * S'_s$	hard-limited, fractional operands
APL_VNEG(s_s, d_s)	: $S_d = -S_s$	hard-limited
APL_VNEG16(s_l, d_l)	: $L_d = -L_s$	hard-limited
APL_VRND(s_l, d_s)	: $S_d = L_s$>>8	
APL_VSIE(s_s, d_l)	: $L_d = S_s$	
APL_VSUB(s_s, s'_s, d_s)	: $S_d = S_{s'} - S_s$	hard-limited
APL_VSUB16(s_l, s'_l, d_l)	: $L_d = L_{s'} - L_s$	hard-limited
APL_XOR(s_s, s'_s, d_s)	: $S_d = S_{s'}$^S_s	

Macros

APL_VCLR(d_s)

APL_SCOPY($0, d_s$)

APL_VCOMUL(s_s, s'_s, d_s, t_s)

APL_VMUR(s_s, s'_s, t_s);
APL_VCRSI(t_s, t_s);
APL_VCMUR(s_s, s'_s, d_s);
APL_VCRAI(d_s, d_s);
APL_VADD(t_s, d_s, d_s)

APL_VCOMUL(s_s, s'_s, d_s, t_s)

APL_VMUR(s_s, s'_s, t_s);
APL_VCRSI(t_s, t_s);
APL_VCMUR(s_s, s'_s, d_s);
APL_VCRAI(d_s, d_s);
APL_VADD(t_s, d_s, d_s);

APL_VMAC(s_s, s'_s, d_l)

APL_VMUL($s_s, s'_s, 8$);
APL_VADD16($8, d_l, d_l$)

APL_VMACR(s_8, s'_8, d_8)

APL_VMUR($s_8, s'_8, 16$);
APL_VADD16($16, d_8, d_8$);

APL_VMUIM(s_s, s'_s, d_l)

APL_VCMUL(s_s, s'_s, d_l);
APL_VCRAI16(d_l, d_l)

APL_VMURE(s_s, s'_s, d_l)

APL_VMUL(s_s, s'_s, d_l);
APL_VCRSI16(d_l, d_l)

I/O Functions

`typedef signed char matrix_type[256][256];`		matrix type
`short *DVEC(`s_l`, short *d)`	`d` $= L_s$	write vector
`void DVSET(short *s,` d_l`)`	$L_d =$ `s`	load (initialize) vector
`signed char *SVEC(`s_s`, signed char *d)`	`d` $= S_s$	write vector
`void SVSET(signed char *s,` d_s`)`	$S_d =$ `s`	load (initialize) vector
`matrix_type *SMAT(`s_m`, matrix_type d)`	`d` $= M_s$	write matrix
`void SMSET(matrix_type s,` d_m`)`	$M_d =$ `s`	load (initialize) matrix

Printing

`void print_s(int k, char *s, char *f)` print `s` and display S_k with format `f`

`void print_l(int k, char *s, char *f)` print `s` and display L_k with format `f`

`void print_m(int k, int u, int v, int i, int j, char *s, char *f)`
print `s` and display $(M_k)_{i,j}$ with format `f`
for $i, j \in [u..u+i] \times [v..v+j]$

`void bits32(int i, char *s, int w)` print `s` with width `w` and display bits of i

Usage

This section illustrates using the simulator and introduces its reporting/logging features via examples. An attempt has been made to be reasonably complete, but unanswered questions should be decidable on the basis of source code (see the appendix).

Example 1: `hello.c`

```
#include "simulate.c"

signed char message[256];

int main()
{
  strcpy(message,"Hello World!");
  SVSET(message,0);
  print_s(0,"label","%c");
  return 0;
}
// gcc -o hello hello.c
```

```
> ./hello
V0 (8): label
  0:Hello World
 16:
 32:
 48:
 64:
 80:
 96:
112:
128:
144:
160:
176:
192:
208:
224:
240:
```

The header line `V0 (8): label` signifies that the contents of vector register 0 (8 bit) – i.e. S_0 – is displayed following the label `label`. After that is a listing of the vector's components (16 per line). Note how various include files are not needed in the program. That is because `#include "simulate.c"` already includes `<stdio.h>`, `<stdlib.h>`, `<unistd.h>`, `<math.h>`, and `<string.h>`.

Example 2: `multiply.c`

This program is considerably more complex; it accepts two 32-bit numbers (base 10) as command line arguments, and returns their product (the algorithm is *not* efficient; it simply exercises the simulator).

```
#include "simulate.c"

matrix_type matrix;

signed char *v0(unsigned long long x)
{
  static signed char v[256];  int i = 0;

  do v[i++] = x%16; while (x /= 16);
  do v[i++] = 0;    while (i < 256);
  return v;
}

unsigned long long convert(signed char *x)
{
  int i = 256;  unsigned long long a = 0;

  while (!x[i]) if (!(i--)) return 0;
  do a = 16*a + x[i]; while (i--);
  return a;
}

signed char *v1()
{
  static signed char v[256];  static int i = 256;

  if (i < 0) return v;
  while (i--) v[i] = ((7 == i%8)? 255: 0);  return v;
}

signed char *v2()
{
  static signed char v[256];  static int i = 256;

  if (i < 0) return v;
  while (i--) v[i] = ((7 == i)? 255: 0);  return v;
}

signed char *v3()
{
  static signed char v[256];  static int i = 256;

  if (i < 0) return v;
  while (i--) v[i] = 15;  return v;
}

void m0(int d)
```

```
{
  int i,j;

  for (i = 0; i < 2*d*d-d; i++){
    for (j = 0; j < 32*d; j++)
      matrix[i][j] = (signed char)((j%d == (d+1)*(i/d)-i)? 128: 0);
    while(j < 256) matrix[i][j++] = 0;
  }
  while (i < 256){ for (j = 0; j < 256; matrix[i][j++] = 0); i++; }
}

void embed(int i, int j, int d, int r)
{
  int k,l;

  APL_COPY(i,9);
  for (l = 0; l < r; l++){
    for (k = d; k--; ){
      if (l){
        APL_SHFT_U(9,i);
      } else {
        APL_SHFT_U(9,j);
      }
    }
    if (l){
      APL_VADD(j,9,j);
    } else {
      APL_VADD(i,9,j);
    }
  }
}

void normalize(int s, int t, int d)
{
  int j = 7, k;

  APL_SCOPY(0,t);
  if (!d) return;
  SVSET(v2(),15);
  SVSET(v3(),14);
 t:
  APL_AND(2*s,15,12);
  APL_AND(2*s+1,15,13);
  APL_AND(12,14,9);
  if (j > 0){
    for (k = j; k--; ){ APL_SHFT_D(9,10); }
  } else if (j < 0){
    for (k = -j; k--; ){ APL_SHFT_U(9,10); }
  }
  APL_VADD(t,9,t);
  if (!--d) return;
  APL_VASR16(6,5); APL_VASR16(5,6); APL_VASR16(6,5); APL_VASR16(5,6);

  APL_COPY16(s,4);
```

```
  for (k = 0; k < 8; k++){ APL_SHFT_D(9,10); }
  APL_COPY(9,2*s+1);
  APL_COPY(8,9);
  for (k = 0; k < 8; k++){ APL_SHFT_D(9,10); }
  APL_COPY(9,2*s);
  APL_VADD16(6,s,s);
  j--;
  goto t;
}

unsigned long long multiply(unsigned long long x, unsigned long long y)
{
  int d = 8;  signed char p[256];

  SVSET(v0(x),0);
  SVSET(v0(y),1);
  embed(0,2,d,32);
  APL_LSL(2,2);  APL_LSL(2,2);  APL_LSL(2,2);  APL_LSL(2,13);
  APL_VMM(0,13,12,0);
  APL_VASR(12,12);
  APL_SSUB(16,12,4);
  APL_SAND(15,4,4);
  embed(1,0,d,2*d-1);
  APL_VMUL(0,4,3);
  APL_VASR16(3,3);
  APL_COPY16(3,4);
  APL_SHFT_U(9,0);
  APL_COPY(9,1);
  APL_COPY(8,9);
  APL_SHFT_U(9,0);
  APL_COPY(9,0);
  APL_VADD16(0,3,4);
  APL_COPY16(4,3);
  APL_SHFT_U(9,0);
  APL_SHFT_U(9,0);
  APL_COPY(9,1);
  APL_COPY(8,9);
  APL_SHFT_U(9,0);
  APL_SHFT_U(9,0);
  APL_COPY(9,0);
  APL_VADD16(0,3,4);
  APL_COPY16(4,3);
  APL_SHFT_U(9,0);
  APL_SHFT_U(9,0);
  APL_SHFT_U(9,0);
  APL_SHFT_U(9,0);
  APL_COPY(9,1);
  APL_COPY(8,9);
  APL_SHFT_U(9,0);
  APL_SHFT_U(9,0);
  APL_SHFT_U(9,0);
  APL_SHFT_U(9,0);
  APL_COPY(9,0);
  APL_VADD16(0,3,4);
```

```
  SVSET(v1(),0);
  APL_AND(0,8,2);
  APL_AND(0,9,3);
  normalize(1,0,2*d*d-d);
  SVEC(0,p);
  return convert(p);
}

int main(int argc, char *argv[])
{
  int d = 8;  unsigned long long x,y,z;

  if (argc != 3){
    printf("usage: multiply x y\n");
    return -1;
  }
  sscanf(*++argv,"%llu",&x);  sscanf(*++argv,"%llu",&y);
  if (!((x < 4294967296ULL)&&(y < 4294967296ULL))){
    printf("numbers must not exceed 32 bits\n");
    return -1;
  }
  m0(d);
  SMSET(matrix,0);
  display_m(0);
  z = multiply(x,y);
  if (x*y != z)
    printf("(%llu) ERROR: %llu * %llu != %llu\n\n",x*y,x,y,z);
  else
    printf("\n%llu * %llu = %llu\n\n",x,y,z);

  return 0;
}
// gcc -Wall -gdwarf-2 -g3 -o multiply multiply.c -lm


> ./multiply 912869128 109247102

912869128 * 109247102 = 99728306739267056
```

Statistics

Including the line `Report(NULL,"main");` in the function `main`, following `z = multiply(x,y);` causes additional output to be produced. A report is generated summarizing instructions executed, estimated time taken to execute them (on actual hardware, with I/O time listed separately), register usage, loads (data transfer into the EnLight processor), and stores (data transfer out of the EnLight processor).

The report is a printout of a statistics object which the simulator manages. There is a stack on which such objects can be manipulated explicitly, if the programmer wishes to do so, but there is also implicit manipulation which provides for both displaying statistics corresponding to a function call and for integrating (summing) the statistics for a function with that of it's descendents. The report for the above should be

```
-----------------------------------------------------
main                                                 Report name
-----------------------------------------------------
Operation                 Count                 Time  Operation, Number of executions, Time to execute those operations
APL_AND                     362           0.000002896
APL_COPY                    368           0.000002944
APL_COPY16                  122           0.000000976
APL_LSL                       4           0.000000032
APL_SAND                      1           0.000000008
APL_SCOPY                     1           0.000000008
APL_SHFT_D                 1932           0.000015456
APL_SHFT_U                 6718           0.000053744
APL_SSUB                      1           0.000000008
APL_VADD                    167           0.000001336
APL_VADD16                  122           0.000000976
APL_VASR                      1           0.000000008
APL_VASR16                  477           0.000003816
APL_VMM                       1           0.000000008
APL_VMUL                      1           0.000000008
SVEC                          1           0.000000064
SVSET                         5           0.000000320
SMSET                         1           0.000016384
                        ---------  --------------------
Total                     10285           0.000098992  Total operations executed, Total execution time
(      I/O)                               0.000016768  Time attributed to Input/Output
(immediate)                   4                       Number of operations containing immediate data

Register       Reads      Writes       Loads      Stores  Register, number of reads, writes, loads, stores
  0  (8)         140         411           2           1  Vector register S0 (8 bits / component)
  1  (8)           2         116           1           0  .
  2  (8)         155         163           0           0  .
  3  (8)         120         120           0           0  .
  4  (8)           2           2           0           0
  5  (8)           0           0           0           0
  6  (8)           0           0           0           0
  7  (8)           0           0           0           0
  8  (8)         123           0           0           0
  9  (8)        9062        8894           0           0
 10  (8)           0        8260           0           0
 11  (8)           0           0           0           0
 12  (8)         122         122           0           0
 13  (8)           1         121           0           0
 14  (8)         120           1           1           0
 15  (8)         240           1           1           0
           ----------  ----------  ----------  ----------
             10087       18211           5           1  Totals (for 8 bit vector registers)
           ----------  ----------  ----------  ----------
  0 (16)           3           0           0           0  Vector register L0 (16 bits / component)
  1 (16)         238         119           0           0  .
  2 (16)           0           0           0           0  .
  3 (16)           5           4           0           0  .
  4 (16)           2         123           0           0
  5 (16)         238         238           0           0
  6 (16)         357         238           0           0
  7 (16)           0           0           0           0
           ----------  ----------  ----------  ----------
               843         722           0           0  Totals (for 16 bit vector registers)
           ----------  ----------  ----------  ----------
             10930       18933           5           1  Totals combining both 8 and 16 bit vector registers

  0  (m)           1           1           1           0  Matrix register M0
  1  (m)           0           0           0           0  .
  2  (m)           0           0           0           0  .
  3  (m)           0           0           0           0  .
           ----------  ----------  ----------  ----------
                 1           1           1           0  Totals (for matrix registers)

-----------------------------------------------------
```

If, for example, one were interested in the statistics for the function `multiply` (in `multiply.c`) to be a separate report, simply put `Begin_Watch;` as the first instruction and `End_Watch("multiply");` as the last instruction in `multiply`. A report corresponding to multiply – which integrates the statistics of `multiply` with it's descendents – will be generated each time `multiply` is exited (the argument to `End_Watch` is simply a label to be inserted into the header of the report).

In case the descendent `normalize` (of `multiply`) should not have it's statistics integrated into the report for `multiply`, simply put `Suspend;` before, and `Resume;` after, the call to `normalize`.

Explicit Manipulation of Statistics Objects

The first argument to `Report(x,y)` is either a pointer to a statistics object (in which case the object pointed to is the basis for the report), or else `NULL` (in which case an implicitly managed default statistics object is the basis for the report; see the section above). Statistics concerning the simulation are accumulated by the default statistics object. The implicitly managed statistics make use of the stack and the operations described in the next paragraph. It is the programmer's responsibility to avoid conflicts – consult the source code – if conflicts are not desired.

The function `save_stats(x)` saves (the statistics contained in) the default statistics object to `x`. The function `restore_stats(x)` overwrites the default object with (the data contained in) `x`. The function `stat_name(x)` returns the name of `x` (a character string). The function `clear_stats(x)` initializes the statistics of `x` to 0. The function `combine_stats(x)` accumulates (adds) into `x` statistics from the default object provided `x` is not `NULL`, if `x` *is* `NULL` then `combine_stats` pretends its argument is the top of stack. The function `push_stats(x)` creates a statistics object with name `x`, initializes its data to match (the data of) the default object, pushes it on the stack, and returns top of stack (this is the only disciplined way to create and obtain a handle – pointer – to a statistics object). The function `pop_stats()` returns the top of stack, overwrites the default object with the top of stack, and pops the stack (if the stack is empty, it returns `NULL` and does nothing else).

Logging

The simulator can write a trace of the simulated execution to a log file. To initiate logging to file `x`, insert `Begin_Log("x");` at the point where logging should begin. To terminate logging, insert `End_Log;` (undefined behavior results if a `Begin_Log;` is not terminated with an `End_Log;`, or if logging intervals – the time interval beginning with execution of `Begin_Log;` and ending with execution of `End_Log;` – are nested or overlap).

For example, wrapping the statement `embed(1,0,d,2*d-1);` (occurring in the function `multiply`) with `Begin_Log("log");` and `End_Log;` produces the log file `log` having first three lines

```
299 APL_COPY rs1 ws9                Instruction count, Instruction name, I/O pattern
300 APL_SHFT_U rs9 *ws9 ws0
301 APL_SHFT_U rs9 *ws9 ws0
```

Here "I/O pattern" is a description of read/write activity; `rs`x indicates S_x was read, `ws`x indicates S_x was written, `rl`x indicates L_x was read, `wl`x indicates L_x was written, `rm`x indicates M_x was read, `wm`x indicates M_x was written, `ri`x indicates an immediate value of x was read. The arguments supplied to an instruction are recoverable from the I/O pattern *after* removing those elements of the pattern prefaced with an asterisk (in particular, instruction 301 is `APL_SHFT_U(9,0)`).

Logging prefaces commands with instruction numbers which are reset to 0 when a `Begin_Watch` is encountered. The instruction numbers are restored – as if the previous `Begin_Watch` had not existed – when a matching `End_Watch` is encountered.

Logging embeds data loaded by the functions `SVSET`, `DVSET`, and `SMSET` into the log file in a format which approximates that used by the printing functions (`print_s`, `print_l`, `print_m`). Comments can be embedded in a log file with the macro `COMMENT(x)` (where `x` is a character string). An instruction `x` can be passed to the log file without being executed via the macro `NoOp(x)`.

Matlab Files

If the file `log` is the result of logging a complete computation (i.e., the body of `main` was wrapped with `Begin_Log("log");` and `End_Log;`), then the utility program `l2matlab` will produce, from the log file, a collection of files for EnLight's Matlab simulator corresponding to the computation. For instance, the files corresponding to `multiply.c` produced by `l2matlab log` are

`log_0.dat`	Matlab data file (for the first load instruction; `multiply.c` contains six of them)
`log_1.dat`	Matlab data file (for the second load instruction)
`log_2.dat`	Matlab data file (for the third load instruction)
`log_3.dat`	Matlab data file (for the fourth load instruction)
`log_4.dat`	Matlab data file (for the fifth load instruction)
`log_5.dat`	Matlab data file (for the sixth load instruction)
`log_apl.m`	Matlab "m file" corresponding to the computation

The utility program `l2matlab` is listed in the appendix, together with the `bash` scripts and `sed` pattern files upon which it depends (the simulator is intended to be used on a gnu/linux system).

Tracing

The simulator has the ability to watch registers and invoke a user defined callback function whenever a register changes. The macro `Trace(x,y)` is how the callback function `y` is associated with register `x`. The register is specified as follows: `Short(`x`)` denotes S_x, `Long(`x`)` denotes L_x, `Matrix(`x`)` denotes M_x. Tracing of `x` is stopped by `Untrace(x)`.

If `y` is `NULL`, then rather than invoking a callback, changes to the contents of register `x` result in the new and old values being printed to `stdout`. This is referred to as *default tracing*. The macros `Trace_S`, `Trace_L`, `Trace_M`, and `Trace_All` start default tracing of $S_0, \ldots S_{15}$, $L_0, \ldots L_7$, $M_0, \ldots M_3$, and all registers (respectively). Their stop trace counterparts are `Untrace_S`, `Untrace_L`, `Untrace_M`, and `Untrace_All`.

Extension

The simulator facilitates user defined extensions to the API via the macros `Begin_Op(x,y,z,w)` and `End_Op(x)`. For example, there is no instruction to replace S_j with the component-wise logical shift of S_i by k bits, where $k > 0$ is a left shift and $k < 0$ is a right shift. Such an instruction may be created by defining a function which implements it and wrapping the function's body with `Begin_Op` and `End_Op`. One such implementation is

```
void APL_LS(int i, int j, int k) // source i, target j, shift k (+ ==> << ; - ==> >>)
{
  Begin_Op(0,APL_LS,8.,Read_Short(i);Write_Short(j);Read_Immediate(k));// number, name, time, i/o

  APL_COPY(i,18);
  if (k > 0)
    while (k--){ APL_LSL(18,18); }
  else if (k < 0)
    for (k = -k; k--; ){ APL_LSR(18,18); }
  APL_COPY(18,j);

  End_Op(0);
}
```

Now `APL_LS` can be used as if it were an instruction; its *definition* will not contribute to statistics, but its name (`APL_LS`) will be listed in statistical reports as if it were an instruction, its user defined time will contribute to expected execution time (the time of its body will not) and its user defined I/O pattern will contribute to register statistics (the I/O pattern of its body will not). Note how the implementation appears to use the nonexistent register S_{18}; the simulator implements the "invisible" registers S_{18}, S_{19}, L_9, and M_4 for defining instructions (hack the source if you want more).

The first argument to `Begin_Op(x,y,z,w)` must be an integer $0 \leq$ `x` < 128 not used by any other extension, and it must match the argument of the corresponding `End_Op(x)`. The second argument `y` is a name for the instruction (to be used in statistics, log files, etc.). The third argument `z` is the user defined time in nanoseconds (float format). The fourth argument `w` is the I/O pattern which defines the read/write activity, specified as follows (listed in argument order and semicolon separated):

`Read_Immediate(`i`)`	i is an immediate argument
`Read_Short(`i`)`	S_i is an argument which is read
`Write_Short(`i`)`	S_i is an argument which is written
`Read_Long(`i`)`	L_i is an argument which is read
`Write_Long(`i`)`	L_i is an argument which is written
`Read_Matrix(`i`)`	M_i is an argument which is read
`Write_Matrix(`i`)`	M_i an argument which is is written
`Read_Short_(`i`)`	S_i is read, but is *not* an argument
`Write_Short_(`i`)`	S_i is written, but is *not* an argument
`Read_Long_(`i`)`	L_i is read, but is *not* an argument
`Write_Long_(`i`)`	L_i is written, but is *not* an argument
`Read_Matrix_(`i`)`	M_i is read, but is *not* an argument
`Write_Matrix_(`i`)`	M_i is written, but is *not* an argument

Modification

The simulated machine can be modified by defining a new instruction as in the previous section, "removing" the instruction to be modified, and redefining it as the new instruction. For instance, if one desires to change `APL_LSL` so that it takes a third argument indicating the number of bits to shift, one might proceed as follows

```
void modified_APL_LSL(int i, int j, int k)
{
  Begin_Op(0,APL_LSL,8.,Read_Short(i);Write_Short(j);Read_Immediate(k));

  APL_COPY(i,18);
  if (k > 0) while (k--){ APL_LSL(18,18); }
  APL_COPY(18,j);

  End_Op(0);
}

#undef APL_LSL
#define APL_LSL(x,y,z) modified_APL_LSL(x,y,z)
```

Speed

Statistics, logging, tracing, and extension incur some amount of overhead. To eliminate that overhead (and the corresponding functionality), include `#define OPTIMIZE` before `#include "simulate.c"` in the application's source file. The timing for the second example when using `#define OPTIMIZE` (and compiling with `gcc -O3 -o multiply multiply.c -lm`) is

```
> time ./multiply 912869128 109247102

912869128 * 109247102 = 99728306739267056

real    0m0.057s
user    0m0.052s
sys     0m0.001s
```

This compares favorably with the Matlab simulator, it executes `log_apl.m` in approximately 25 seconds.

Cautions

- The simulator is *not* thread safe.
- It is assumed that users are familiar with C, gcc, and cpp's macro facilities.
- Most things involving the simulator are macros; arguments may be evaluated multiple times and what may appear to be a single statement may expand into many... program defensively!

Appendix

Variations

The simulator is not polymorphic, so EnLight's four-variable versions of `APL_SHFT_D` and `APL_SHFT_U` are implemented as `APL_SHFT_D2` and `APL_SHFT_U2`.

The functions which write registers (`DVEC`, `SVEC`, `SMAT`) differ from their Matlab counterparts which have no destination argument.

Extensions

Long (16 bit) versions of `APL_VCRAI` and `APL_VCRSI` are implemented as `APL_VCRAI16` and `APL_VCRSI16`.

Source

`machine.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include <string.h>

FILE     *LogFile;
int      Print = 1;

int       Du, Dv;
long      Z = 1, T, Op[128+1024];
long      R_m[ 4], W_m[4];
long      R_s[16], R_l[8], R_i;
long      W_s[16], W_l[8];
long      I_s[16], I_l[8], O_s[16], O_l[16], I_m[4], O_m[4];
long long I;
short     V[10][257], M[5][256][256], K[2] = {255<<8, 255};
short     V_[10][257], M_[5][256][256];

typedef void (*callback)();
callback     Callback[28];

#define STATE_SIZE (sizeof(V)+sizeof(M))

typedef unsigned char state[STATE_SIZE];

#define Save(x)       { memcpy(x,V,sizeof(V)); memcpy(x+sizeof(V),M,sizeof(M)); }
#define Save_l(x)       memcpy(V_[x],V[x],257*sizeof(short))
#define Save_s(x)       Save_l((x)/2)
#define Save_m(x)       memcpy(M_[x],M[x],65536*sizeof(short))
#define Restore(x)    { memcpy(V,(x),sizeof(V)); memcpy(M,x+sizeof(V),sizeof(M)); }
#define Restore_l(x)    memcpy(V[x],V_[x],257*sizeof(short))
#define Restore_s(x)    Restore_l((x)/2)
#define Restore_m(x)    memcpy(M[x],M_[x],65536*sizeof(short))

#define Short(x)      (x)
```

```
#define Long(x)       ((x)+16)
#define Matrix(x)     ((x)+24)
#define WatchReg(x,y) Callback[x] = (y)

#define unsigned(x)   ((unsigned short)(255&(x)))
#define short(x)      ((short)(x))
#define int(x)        ((int)(x))
#define extend(x)     short((signed char)(x))

#define R(i)          (((i)&1)? (V[(i)>>1][Du]>>8): extend(255&V[(i)>>1][Du]))
#define r(i,u)        (((i)&1)? (V[(i)>>1][u ]>>8): extend(255&V[(i)>>1][u ]))
#define Rl(i)         int(V[i][Du])

#define W(i,x)        V[(i)>>1][Du] = (V[(i)>>1][Du]&K[(i)&1])|(((i)&1)? (x)<<8: (x)&255)
#define w(i,u,x)      V[(i)>>1][u] = (V[(i)>>1][u]&K[(i)&1])|(((i)&1)? (x)<<8: (x)&255)
#define Wl(i,x)       V[i][Du] = (x)

#define diff_s(i)                                                 \
{                                                                 \
  for (Du = 256; Du--; )                                          \
    if ((255&(((i)&1)? (V[(i)>>1][Du]>>8): V[(i)>>1][Du])) !=     \
        (255&(((i)&1)? (V_[(i)>>1][Du]>>8): V_[(i)>>1][Du]))){    \
      Callback[Short(i)]();                                       \
      break;                                                      \
    }                                                             \
}

#define diff_l(i)                                         \
{                                                         \
  for (Du = 256; Du--; )                                  \
    if (V[i][Du] != V_[i][Du]){                           \
      Callback[Long(i)]();                                \
      break;                                              \
    }                                                     \
}

#define diff_m(x)                                         \
{                                                         \
  for (Du = 0; Du < 256; Du++)                            \
    for (Dv = 0; Dv < 256; Dv++){                         \
      if (M[x][Du][Dv] != M_[x][Du][Dv]){                 \
        Callback[Matrix(x)]();                            \
        Du = 256; break;                                  \
      }                                                   \
    }                                                     \
}

#define C(o,e)        for(Du = 256, I+=Z; Du      ; Du--, o, Du--, e)
#define D(x)          for(Du = 256, I+=Z; Du--    ; x               )
#define D2(x,y)       for(Du = 256, I+=Z; Du--    ; x, y            )
#define D3(x,y,z)     for(Du = 256, I+=Z; Du--    ; x, y, z         )
#define U2(x,y)       for(Du = 0  , I+=Z; Du < 256; x, y, Du++      )
#define U3(x,y,z)     for(Du = 0  , I+=Z; Du < 256; x, y, z, Du++   )

#define HL(e,b)       (((e) >= (1<<((b)-1)))? (1<<((b)-1))-1 : (((e) < -(1<<((b)-1)))? -(1<<((b)-1)): (e)))

#define H(e)          HL(e,8)
#define Hl(e)         HL(e,16)
#define SL(e,b)       HL(int(e)/(1<<(b)),b)
#define L(e)          SL(e,8)

#define ABS(x)        ((int(x) < 0)? - int(x): int(x))

#define abs(i,j)      D(  W(j,  H(  ABS(R(i))                              ) ) )
#define add(i,j,k)    D(  W(k,  H(       R(i) +  R(j)                      ) ) )
#define addi(i,j,k)   D(  W(k,  H(   short(i) +  R(j)                      ) ) )
#define and(i,j,k)    D(  W(k,           R(i) &  R(j)                        ) )
#define andi(i,j,k)   D(  W(k,            (i) &  R(j)                        ) )
```

```
#define asl(i,j)      D(  W(j,  H(       R(i) << 1                          ) ) )
#define asr(i,j)      D(  W(j,           R(i) /  2                            ) )
#define cmp(i,j,k)    D(  W(k, ((R( i)>R( j))?   127: ((R( i)<R( j))?   128: 0))) )
#define cp(i,j)       D(  W(j,           R(i)                                 ) )
#define absl(i,j)     D( Wl(j, Hl( ABS(Rl(i))                               ) ) )
#define addil(i,j,k)  D( Wl(k, Hl(     int(i) +  Rl(j)                      ) ) )
#define addl(i,j,k)   D( Wl(k, Hl(      Rl(i) +  Rl(j)                      ) ) )
#define addls(i,j,k)  D( Wl(k, Hl(      Rl(i) +  int( R(j))                 ) ) )
#define asll(i,j)     D( Wl(j, Hl(      Rl(i) << 1                          ) ) )
#define asrl(i,j)     D( Wl(j,          Rl(i) /  2                            ) )
#define cmpl(i,j,k)   D( Wl(k, ((Rl(i)>Rl(j))? 32767: ((Rl(i)<Rl(j))? 32768: 0))) )
#define cpl(i,j)      D( Wl(j,          Rl(i)                                 ) )
#define ex(i,j)       D( Wl(j,           R(i)                                 ) )
#define ge(i,j,k)     D(  W(k,         ((R(i) <  R(j))? 0: 255)               ) )
#define gel(i,j,k)    D( Wl(k,        ((Rl(i) <  Rl(j))? 0: 65535)            ) )
#define ldi(i,j)      D(  W(j,            i                                   ) )
#define lsl(i,j)      D(  W(j, ((unsigned char)((signed char)R(i))) << 1       ) )
#define lsr(i,j)      D(  W(j, unsigned(R(i)) >> 1                            ) )
#define mul(i,j,k)    D(  W(k,  L(  int(R(i)) *2* int(R(j))                 ) ) )
#define muli(i,j,k)   D( Wl(k, Hl(        (i) *2* int(R(j))                 ) ) )
#define mulis(i,j,k)  D(  W(k,  L(        (i) *2* int(R(j))                 ) ) )
#define mull(i,j,k)   D( Wl(k, Hl(       R(i) *2* int(R(j))                 ) ) )
#define neg(i,j)      D(  W(j,  H(     - R(i)                               ) ) )
#define negl(i,j)     D( Wl(j, Hl(     -Rl(i)                               ) ) )
#define not(i,j)      D(  W(j,          ~R(i)                                 ) )
#define or(i,j,k)     D(  W(k,           R(i) |  R(j)                         ) )
#define ori(i,j,k)    D(  W(k,       short(i) |  R(j)                         ) )
#define sub(i,j,k)    D(  W(k,  H(       R(j) -  R(i)                       ) ) )
#define subi(i,j,k)   D(  W(k,  H(   short(i) -  R(j)                       ) ) )
#define subil(i,j,k)  D( Wl(k, Hl(     int(i) -  Rl(j)                      ) ) )
#define subl(i,j,k)   D( Wl(k, Hl(      Rl(j) -  Rl(i)                      ) ) )
#define tr(i,j)       D(  W(j,  L(      Rl(i)                               ) ) )
#define xor(i,j,k)    D(  W(k,           R(i) ^  R(j)                         ) )
#define xori(i,j,k)   D(  W(k,            (i) ^  R(j)                         ) )

#define csu(i,j)      D2( W(j,R(i)), w(i,Du+1,R(i)) ); w(i,0,0)
#define csu2(i,j,k)   D3( W(k,R(j)), w(i,Du+1,R(i)) , w(j,Du+1,R(j)) ); w(i,0,0); w(j,0,r(i,256))

#define addii(i,j,k)  C(  W(k,  H(  R(j) + short(i)  )  ),  W(k,     R(j)               ) )
#define addir(i,j,k)  C(  W(k,      R(j)               ),  W(k,  H( R(j)  + short(i) )   ) )
#define cnj(i,j)      C(  W(j,  H(       - R(i)      )  ),  W(j,     R(i)               ) )
#define cnjl(i,j)     C( Wl(j, Hl(       - Rl(i)     )  ), Wl(j,    Rl(i)               ) )
#define im(i,j)       C(  W(j,      R(i)               ),  W(j,     0                  ) )
#define ldii(i,j)     C(  W(j,      i                  ),  W(j,     0                  ) )
#define ldir(i,j)     C(  W(j,      0                  ),  W(j,     i                  ) )
#define mi(i,j)       C(  w(j,Du-1,H(     - R(i)      ) ),  w(j, Du+1,R(i)           ) )
#define re(i,j)       C(  W(j,      0                  ),  W(j,     R(i)               ) )
#define sube(i,j)     C(  W(j,      0                  ),  W(j,  H( R(i)  -  r(i,Du+1) ) ) )
#define subel(i,j)    C( Wl(j,      0                  ), Wl(j, Hl( Rl(i) - rl(i,Du+1) ) ) )
#define sumo(i,j)     C(  W(j,  H(  R(i) + r(i,Du-1)  ) ),  W(j,     0                  ) )
#define sumol(i,j)    C( Wl(j, Hl( Rl(i) + rl(i,Du-1) ) ), Wl(j,     0                  ) )
#define xp(i,j,k)     C(  W(k,  L(int(2)*R(i)*r(j,Du-1))),  W(k,  L(int(2)*R(i)*r(j,Du+1) ) ) )
#define xpl(i,j,k)    C( Wl(k, Hl(int(2)*R(i)*r(j,Du-1))), Wl(k, Hl(int(2)*R(i)*r(j,Du+1) ) ) )

#define csd(i,j)      w(i,256,0); U2( W(j,R(i)), W(i,r(i,Du+1)) )
#define csd2(i,j,k)   w(i,256,r(j,0)); w(j,256,0); U3( W(k,R(j)), W(i,r(i,Du+1)), W(j,r(j,Du+1)) )

#define mm(i,j,k,l)                                   \
{                                                     \
  for (I += Z, Dv = 256; Dv--; ){                     \
    for(T = 0, Du = 256; Du--; T += M[i][Dv][Du]*R(j)); \
    w(k,Dv,H(T>>(15-(l))));                           \
  }                                                   \
}

#define O2(o,x,y)     if ((x) == (y))              { o(x,16)    ; cp(16,y) ; I-=Z; }else{ o(x,y)    ;}
#define O2l(o,x,y)    if ((x) ==  y )              { o(x, 8)    ; cpl(8,y) ; I-=Z; }else{ o(x,y)    ;}
#define O3(o,x,y,z)   if(((x) == (z))||((y) == (z))){ o(x,y,16)  ; cp(16,z) ; I-=Z; }else{ o(x,y,z)  ;}
```

```
#define O3l(o,x,y,z)  if(((x) == (z))||((y) == (z))){ o(x,y, 8)  ; cpl(8,z) ; I-=Z; }else{ o(x,y,z)  ;}
#define O4(o,x,y,z,w) if(((y) == (z)))              { o(x,y,16,w); cp(16,z) ; I-=Z; }else{ o(x,y,z,w);}

unsigned char red(short x)
{
  return ((x)? 255: 0);
}

unsigned char green(short x)
{
  return ((x)? 255: 0);
}

unsigned char blue(short x)
{
  return ((x)? 255: 0);
}

int display_m(int k)
{
  int i,j; char n[16]; FILE *f;

  sprintf(n,"matrix%d.ppm",k);
  if (!(f = fopen(n,"w"))) return 0;
  fprintf(f,"P6\n# %s\n256 256\n255\n",n);
  for(i = 0; i < 256; i++)
    for(j = 0; j < 256; j++)
      fprintf(f,"%c%c%c",red(M[k][i][j]),green(M[k][i][j]),blue(M[k][i][j]));
  fclose(f);
  return 1;
}

void print_s(int k, char *s, char *f)
{
  int i,j;  if (!f) f = " %4d";

  if (Print){
    printf("V%d (8): %s\n",k,s);
    for(i = 0; i < 256; i+= 16){
      printf("%3d:",i);
      for(j = 0; j < 16; j++) printf(f,r(k,i+j));
      putchar('\n');
    }
    putchar('\n');
  } else if (LogFile){
    fprintf(LogFile,"\nV%d (8): %s\n",k,s);
    for(i = 0; i < 256; i+= 16){
      fprintf(LogFile," %3d:",i);
      for(j = 0; j < 16; j++) fprintf(LogFile," %4d",r(k,i+j));
      fputc('\n',LogFile);
    }
    fputc('\n',LogFile);
  }
}

void print_l(int k, char *s, char *f)
{
  int i,j;  if (!f) f = " %6d";

  if (Print){
    printf("V%d (16): %s\n",k,s);
    for(i = 0; i < 256; i+= 8){
      printf("%3d:",i);
      for(j = 0; j < 8; j++) printf(f,V[k][i+j]);
      putchar('\n');
    }
    putchar('\n');
  } else if (LogFile){
```

```
    fprintf(LogFile,"\nV%d (16): %s\n",k,s);
    for(i = 0; i < 256; i+= 8){
      fprintf(LogFile," %3d:",i);
      for(j = 0; j < 8; j++) fprintf(LogFile," %6d",V[k][i+j]);
      fputc('\n',LogFile);
    }
    fputc('\n',LogFile);
  }
}

void print_m(int k, int u, int v, int i, int j, char *s, char *f)
{
  int a,b;  if (!f) f = " %4d";

  if (Print){
    printf("M%d: %s\n",k,s);
    printf("    "); for(a = 0; a < j; a++) printf(" %4d",v+a); putchar('\n');
    for(a = 0; a < i; a++){
      printf("%3d:",u+a);
      for(b = 0; b < j; b++) printf(f,M[k][u+a][v+b]);
      putchar('\n');
    }
    putchar('\n');
  } else if (LogFile){
    fprintf(LogFile,"\nM%d: %s\n",k,s);
    fprintf(LogFile,"    "); for(a = 0; a < j; a++) fprintf(LogFile," %4d",v+a); fputc('\n',LogFile);
    for(a = 0; a < i; a++){
      fprintf(LogFile," %3d:",u+a);
      for(b = 0; b < j; b++) fprintf(LogFile," %4d",M[k][u+a][v+b]);
      fputc('\n',LogFile);
    }
    fputc('\n',LogFile);
  }
}

void bits32(int i, char *s, int w)
{
  unsigned h = 0, j = 1<<31;  char f[64];

  sprintf(f,"%c%ds %c10d: ",'%',w,'%'); printf(f,s,i);
  do printf("%c%s",((j&i)?'1':'0'),(((!(++h&3))? " ": "")); while(j >>= 1); putchar('\n');
}
```

simulate.c

```
#include "machine.c"

char *Def[1024], *Ops[66] = {"APL_AND",
                             "APL_COPY",
                             "APL_COPY16",
                             "APL_LSL",
                             "APL_LSR",
                             "APL_NOT",
                             "APL_OR",
                             "APL_SADD",
                             "APL_SADDM",
                             "APL_SAND",
                             "APL_SCIA",
                             "APL_SCIV",
                             "APL_SCOPY",
                             "APL_SCRA",
                             "APL_SCRV",
                             "APL_SHFT_D",
                             "APL_SHFT_D2",
                             "APL_SHFT_TRF",
                             "APL_SHFT_U",
                             "APL_SHFT_U2",
                             "APL_SMUL",
                             "APL_SMUR",
                             "APL_SOR",
                             "APL_SSUB",
                             "APL_SSUBM",
                             "APL_SXOR",
                             "APL_VABS",
                             "APL_VABS16",
                             "APL_VADD",
                             "APL_VADD16",
                             "APL_VADDM",
                             "APL_VASL",
                             "APL_VASL16",
                             "APL_VASR",
                             "APL_VASR16",
                             "APL_VCCONJ",
                             "APL_VCCONJ16",
                             "APL_VCMUL",
                             "APL_VCMUR",
                             "APL_VCOGE",
                             "APL_VCOGE16",
                             "APL_VCOMP",
                             "APL_VCOMP16",
                             "APL_VCRAI",
                             "APL_VCRAI16",
                             "APL_VCRSI",
                             "APL_VCRSI16",
                             "APL_VEIM",
                             "APL_VERE",
                             "APL_VMM",
                             "APL_VMUJ",
                             "APL_VMUL",
                             "APL_VMUR",
                             "APL_VNEG",
                             "APL_VNEG16",
                             "APL_VRND",
                             "APL_VSIE",
                             "APL_VSUB",
                             "APL_VSUB16",
                             "APL_XOR",
                             "DVEC",
                             "DVSET",
                             "SVEC",
                             "SVSET",
```

```
                            "SMAT",
                            "SMSET"};

int Noop;

double Time[1024+128] = {8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,
                         8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,
                         8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,8.,
                         128.,128.,64.,64.,16384.,16384};

#ifndef OPTIMIZE

#define Error(f,s) printf(f,s)

#define C0(i,j)            if  ((i) !=  (j))               Error("--> register constraint %s violated\n","C0")
#define C1(i,j)            if (((i) !=    8)||((j) !=  9)) Error("--> register constraint %s violated\n","C1")
#define C2(i)              if (((i) !=    8)&&((i) !=  9)) Error("--> register constraint %s violated\n","C2")
#define C3(i,j)            if (((i) !=   13)||((j) != 12)) Error("--> register constraint %s violated\n","C3")

#define CS(x) if (Z&&Callback[Short(x) ])
#define CL(x) if (Z&&Callback[Long(x)  ])
#define CM(x) if (Z&&Callback[Matrix(x)])

#define Log_i(x,t) if (Z&&LogFile) fprintf(LogFile," %si%d",t,x);
#define Log_s(x,t) if (Z&&LogFile) fprintf(LogFile," %ss%d",t,x);
#define Log_l(x,t) if (Z&&LogFile) fprintf(LogFile," %sl%d",t,x);
#define Log_m(x,t) if (Z&&LogFile) fprintf(LogFile," %sm%d",t,x);
#define Log_o(n)   if (Z&&LogFile) \
                     fprintf(LogFile,"\n%lld %s",I,(((n)<128)?Ops[n]:Def[(n)-128]));

#define WS(w,t) W_s[w]+=Z; CS(w) Save_s(w); Log_s(w,t);
#define WL(w,t) W_l[w]+=Z; CL(w) Save_l(w); Log_l(w,t);
#define WM(w,t) W_m[w]+=Z; CM(w) Save_m(w); Log_m(w,t);
#define RI(r,t) R_i   +=Z; Log_i(r,t);
#define RS(r,t) R_s[r]+=Z; Log_s(r,t);
#define RL(r,t) R_l[r]+=Z; Log_l(r,t);
#define RM(r,t) R_m[r]+=Z; Log_m(r,t);

#define S00(n,r0,w0)                    Log_o(n); Op[n]+=Z; RS( r0,"r");              WS( w0, "w")
#define S01(n,r0,w0)                    Log_o(n); Op[n]+=Z; RL( r0,"r");              WL( w0, "w")
#define S02(n,r0,r1,w0)                 Log_o(n); Op[n]+=Z; RS( r0,"r"); RS( r1,"r"); WS( w0, "w")
#define S03(n,w0)                       Log_o(n); Op[n]+=Z;                           WS( w0, "w")
#define S04(n,rw0,w1)      C0(rw0,9);   Log_o(n); Op[n]+=Z; RS(rw0,"r");              WS(rw0,"*w"); WS( w1,"w")
#define S05(n,rw0,rw1,w0)  C1(rw0,rw1); Log_o(n); Op[n]+=Z; RS(rw0,"r"); RS(rw1,"r"); WS(rw0,"*w"); WS(rw1,"*w"); WS(w0,"w")
#define S06(n,r0,r1,w0)                 Log_o(n); Op[n]+=Z; RL( r0,"r"); RL( r1,"r"); WL( w0, "w")
#define S07(n,r0,r1,w0)                 Log_o(n); Op[n]+=Z; RL( r0,"r"); RS( r1,"r"); WL( w0, "w")
#define S08(n,r0,r1,w0)                 Log_o(n); Op[n]+=Z; RS( r0,"r"); RS( r1,"r"); WL( w0, "w")

#define S09(n,rm,r0,w0,ri) C3(r0,w0);   Log_o(n); Op[n]+=Z; RM( rm,"r"); RS( r0,"r"); WS( w0, "w"); RI( ri,"r");
#define S10(n,r0,w0)                    Log_o(n); Op[n]+=Z; RL( r0,"r");              WS( w0, "w")
#define S11(n,r0,w0)                    Log_o(n); Op[n]+=Z; RS( r0,"r");              WL( w0, "w")
#define S12(n,r0,w0)       C2(w0);      Log_o(n); Op[n]+=Z; RS( r0,"r");              WS( w0, "w")
#define S13(n,ri,r0,w0)                 Log_o(n); Op[n]+=Z; RI( ri,"r"); RS( r0,"r"); WS( w0, "w")
#define S14(n,ri,r0,w0)                 Log_o(n); Op[n]+=Z; RI( ri,"r"); RL( r0,"r"); WL( w0, "w")
#define S15(n,ri,w0)                    Log_o(n); Op[n]+=Z; RI( ri,"r");              WS( w0, "w")
#define S16(n,ri,r1,w0)                 Log_o(n); Op[n]+=Z; RI( ri,"r"); RS( r1,"r"); WL( w0, "w")

#define D00(r0,w0)         CS( w0) diff_s( w0)
#define D01(r0,w0)         CL( w0) diff_l( w0)
#define D02(r0,r1,w0)      CS( w0) diff_s( w0)
#define D03(w0)            CS( w0) diff_s( w0)
#define D04(rw0,w1)        CS(rw0) diff_s(rw0); CS(w1 ) diff_s( w1)
#define D05(rw0,rw1,w0)    CS(rw0) diff_s(rw0); CS(rw1) diff_s(rw1); CS(w0) diff_s(w0)
#define D06(r0,r1,w0)      CL( w0) diff_l( w0)
#define D07(r0,r1,w0)      CL( w0) diff_l( w0)
#define D08(r0,r1,w0)      CL( w0) diff_l( w0)
#define D09(rm,r0,w0)      CS( w0) diff_s( w0)
#define D10(r0,w0)         CS( w0) diff_s( w0)
```

```
#define D11(r0,w0)         CL( w0) diff_l( w0)
#define D12(r0,w0)         CS( w0) diff_s( w0)
#define EX(x)              if (!Noop){ x; }

#else

#undef diff_s
#undef diff_l
#undef diff_m
#define diff_s(i)
#define diff_l(i)
#define diff_m(i)

#define CS(x)
#define CL(x)
#define CM(x)
#define Log_i(x,t)
#define Log_s(x,t)
#define Log_l(x,t)
#define Log_m(x,t)
#define Log_o(n)
#define WS(w,t)
#define WL(w,t)
#define WM(w,t)
#define RI(r,t)
#define RS(r,t)
#define RL(r,t)
#define RM(r,t)
#define S00(n,r0,w0)
#define S01(n,r0,w0)
#define S02(n,r0,r1,w0)
#define S03(n,w0)
#define S04(n,rw0,w1)
#define S05(n,rw0,rw1,w0)
#define S06(n,r0,r1,w0)
#define S07(n,r0,r1,w0)
#define S08(n,r0,r1,w0)
#define S09(n,rm,r0,w0,ri)
#define S10(n,r0,w0)
#define S11(n,r0,w0)
#define S12(n,r0,w0)
#define S13(n,ri,r0,w0)
#define S14(n,ri,r0,w0)
#define S15(n,ri,w0)
#define S16(n,ri,r1,w0)
#define D00(r0,w0)
#define D01(r0,w0)
#define D02(r0,r1,w0)
#define D03(w0)
#define D04(rw0,w1)
#define D05(rw0,rw1,w0)
#define D06(r0,r1,w0)
#define D07(r0,r1,w0)
#define D08(r0,r1,w0)
#define D09(rm,r0,w0)
#define D10(r0,w0)
#define D11(r0,w0)
#define D12(r0,w0)

#define EX(x) x

#endif

#define APL_AND(i,j,k)     { S02( 0,i,j,k);   EX(and(i,j,k));     D02(i,j,k); }
#define APL_COPY(i,j)      { S00( 1,i,j);     EX(cp(i,j));        D00(i,j);   }
#define APL_COPY16(i,j)    { S01( 2,i,j);     EX(cpl(i,j));       D01(i,j);   }
#define APL_LSL(i,j)       { S00( 3,i,j);     EX(lsl(i,j));       D00(i,j);   }
#define APL_LSR(i,j)       { S00( 4,i,j);     EX(lsr(i,j));       D00(i,j);   }
```

```
#define APL_NOT(i,j)       { S00( 5,i,j);     EX(not(i,j));       D00(i,j);   }
#define APL_OR(i,j,k)      { S02( 6,i,j,k);   EX(or(i,j,k));      D02(i,j,k); }
#define APL_SADD(i,j,k)    { S13( 7,i,j,k);   EX(addi(i,j,k));    D00(j,k);   }
#define APL_SADDM(i,j,k)   { S14( 8,i,j,k);   EX(addil(i,j,k));   D01(j,k);   }
#define APL_SAND(i,j,k)    { S13( 9,i,j,k);   EX(andi(i,j,k));    D00(j,k);   }
#define APL_SCIA(i,j,k)    { S13(10,i,j,k);   EX(addii(i,j,k));   D00(j,k);   }
#define APL_SCIV(i,j)      { S03(11,j);       EX(ldii(i,j));      D03(j);     }
#define APL_SCOPY(i,j)     { S15(12,i,j);     EX(ldi(i,j));       D03(j);     }
#define APL_SCRA(i,j,k)    { S13(13,i,j,k);   EX(addir(i,j,k));   D00(j,k);   }
#define APL_SCRV(i,j)      { S15(14,i,j);     EX(ldir(i,j));      D03(j);     }
#define APL_SHFT_D(i,j)    { S04(15,i,j);     EX(csd(i,j));       D04(i,j);   }
#define APL_SHFT_D2(i,j,k) { S05(16,i,j,k);   EX(csd2(i,j,k));    D05(i,j,k); }
#define APL_SHFT_TRF(i,j)  { S12(17,i,j);     EX(cp(i,j));        D12(i,j);   }
#define APL_SHFT_U(i,j)    { S04(18,i,j);     EX(csu(i,j));       D04(i,j);   }
#define APL_SHFT_U2(i,j,k) { S05(19,i,j,k);   EX(csu2(i,j,k));    D05(i,j,k); }
#define APL_SMUL(i,j,k)    { S16(20,i,j,k);   EX(muli(i,j,k));    D01(j,k);   }
#define APL_SMUR(i,j,k)    { S13(21,i,j,k);   EX(mulis(i,j,k));   D00(j,k);   }
#define APL_SOR(i,j,k)     { S13(22,i,j,k);   EX(ori(i,j,k));     D00(j,k);   }
#define APL_SSUB(i,j,k)    { S13(23,i,j,k);   EX(subi(i,j,k));    D00(j,k);   }
#define APL_SSUBM(i,j,k)   { S14(24,i,j,k);   EX(subil(i,j,k));   D01(j,k);   }
#define APL_SXOR(i,j,k)    { S13(25,i,j,k);   EX(xori(i,j,k));    D00(j,k);   }
#define APL_VABS(i,j)      { S00(26,i,j);     EX(abs(i,j));       D00(i,j);   }
#define APL_VABS16(i,j)    { S01(27,i,j);     EX(absl(i,j));      D01(i,j);   }
#define APL_VADD(i,j,k)    { S02(28,i,j,k);   EX(add(i,j,k));     D02(i,j,k); }
#define APL_VADD16(i,j,k)  { S06(29,i,j,k);   EX(addl(i,j,k));    D06(i,j,k); }
#define APL_VADDM(i,j,k)   { S07(30,i,j,k);   EX(addls(i,j,k));   D07(i,j,k); }
#define APL_VASL(i,j)      { S00(31,i,j);     EX(asl(i,j));       D00(i,j);   }
#define APL_VASL16(i,j)    { S01(32,i,j);     EX(asll(i,j));      D01(i,j);   }
#define APL_VASR(i,j)      { S00(33,i,j);     EX(asr(i,j));       D00(i,j);   }
#define APL_VASR16(i,j)    { S01(34,i,j);     EX(asrl(i,j));      D01(i,j);   }
#define APL_VCCONJ(i,j)    { S00(35,i,j);     EX(cnj(i,j));       D00(i,j);   }
#define APL_VCCONJ16(i,j)  { S01(36,i,j);     EX(cnjl(i,j));      D01(i,j);   }
#define APL_VCMUL(i,j,k)   { S08(37,i,j,k);   EX(O3l(xpl,i,j,k)); D08(i,j,k); }
#define APL_VCMUR(i,j,k)   { S02(38,i,j,k);   EX(O3(xp,i,j,k));   D02(i,j,k); }
#define APL_VCOGE(i,j,k)   { S02(39,i,j,k);   EX(ge(i,j,k));      D02(i,j,k); }
#define APL_VCOGE16(i,j,k) { S06(40,i,j,k);   EX(gel(i,j,k));     D06(i,j,k); }
#define APL_VCOMP(i,j,k)   { S02(41,i,j,k);   EX(cmp(i,j,k));     D02(i,j,k); }
#define APL_VCOMP16(i,j,k) { S06(42,i,j,k);   EX(cmpl(i,j,k));    D06(i,j,k); }
#define APL_VCRAI(i,j)     { S00(43,i,j);     EX(O2(sumo,i,j));   D00(i,j);   }
#define APL_VCRAI16(i,j)   { S01(44,i,j);     EX(O2l(sumol,i,j)); D01(i,j);   }
#define APL_VCRSI(i,j)     { S00(45,i,j);     EX(O2(sube,i,j));   D00(i,j);   }
#define APL_VCRSI16(i,j)   { S01(46,i,j);     EX(O2l(subel,i,j)); D01(i,j);   }
#define APL_VEIM(i,j)      { S00(47,i,j);     EX(im(i,j));        D00(i,j);   }
#define APL_VERE(i,j)      { S00(48,i,j);     EX(re(i,j));        D00(i,j);   }
#define APL_VMM(i,j,k,l)   { S09(49,i,j,k,l); EX(O4(mm,i,j,k,l)); D09(i,j,k); }
#define APL_VMUJ(i,j)      { S00(50,i,j);     EX(mi(i,j));        D00(i,j);   }
#define APL_VMUL(i,j,k)    { S08(51,i,j,k);   EX(mull(i,j,k));    D08(i,j,k); }
#define APL_VMUR(i,j,k)    { S02(52,i,j,k);   EX(mul(i,j,k));     D02(i,j,k); }
#define APL_VNEG(i,j)      { S00(53,i,j);     EX(neg(i,j));       D00(i,j);   }
#define APL_VNEG16(i,j)    { S01(54,i,j);     EX(negl(i,j));      D01(i,j);   }
#define APL_VRND(i,j)      { S10(55,i,j);     EX(tr(i,j));        D10(i,j);   }
#define APL_VSIE(i,j)      { S11(56,i,j);     EX(ex(i,j));        D11(i,j);   }
#define APL_VSUB(i,j,k)    { S02(57,i,j,k);   EX(sub(i,j,k));     D02(i,j,k); }
#define APL_VSUB16(i,j,k)  { S06(58,i,j,k);   EX(subl(i,j,k));    D06(i,j,k); }
#define APL_XOR(i,j,k)     { S02(59,i,j,k);   EX(xor(i,j,k));     D02(i,j,k); }

#define APL_VCLR(i)           APL_SCOPY(0,i)

#define APL_VCOMUL(i,j,k,l) { APL_VMUR(i,j,l); \
                              APL_VCRSI(l,l);  \
                              APL_VCMUR(i,j,k);\
                              APL_VCRAI(k,k);  \
                              APL_VADD(l,k,k); }

#define APL_VMAC(i,j,k)     { APL_VMUL(i,j,8);   \
                              APL_VADD16(8,k,k); }

#define APL_VMACR(i,j,k)    { APL_VMUR(i,j,16);   \
```

```
                              APL_VADD16(16,k,k); }


#define APL_VMUIM(i,j,k)    { APL_VCMUL(i,j,k); \
                              APL_VCRAI16(k,k); }

#define APL_VMURE(i,j,k)    { APL_VMUL(i,j,k);  \
                              APL_VCRSI16(k,k); }


short *DVEC(int i, short *v)
{
  int j = 256;

  Log_o(60); RL(i,"r"); Op[60]+=Z; O_l[i]+=Z; I+=Z;
  while(j--) v[j] = V[i][j];
  return v;
}

void DVSET(short *d, int i)
{
  int j = 256;

  Log_o(61); WL(i,"w"); Op[61]+=Z; I_l[i]+=Z; I+=Z;
  while(j--) V[i][j] = d[j];
  CL(i) diff_l(i);
  if (LogFile){
    j = Print; Print = 0;
    print_l(i,"DVSET",NULL);
    Print = j;
  }
}

signed char *SVEC(int i, signed char *v)
{
  Log_o(62); RS(i,"r"); Op[62]+=Z; O_s[i]+=Z; I+=Z;
  for (Du = 256; Du--; v[Du] = R(i));
  return v;
}

void SVSET(signed char *d, int i)
{
  int j;

  Log_o(63); WS(i,"w"); Op[63]+=Z; I_s[i]+=Z; I+=Z;
  for (Du = 256; Du--; W(i,(short)d[Du]));
  CS(i) diff_s(i);
  if (LogFile){
    j = Print; Print = 0;
    print_s(i,"SVSET",NULL);
    Print = j;
  }
}

typedef signed char matrix_type[256][256];

matrix_type *SMAT(int k, matrix_type m)
{
  int i,j;

  Log_o(64); RM(k,"r"); Op[64]+=Z; O_m[k]+=Z; I+=Z;
  for (i = 0; i < 256; i++) for (j = 0; j < 256; j++) m[i][j] = M[k][i][j];
  return (matrix_type *)m;
}

void SMSET(matrix_type m, int k)
{
  int i,j;
```

```
  Log_o(65); WM(k,"w"); Op[65]+=Z; I_m[k]+=Z; I+=Z;
  for (i = 0; i < 256; i++) for (j = 0; j < 256; j++) M[k][i][j] = m[i][j];
  CM(k) diff_m(k);
  if (LogFile){
    j = Print; Print = 0;
    print_m(k,0,0,256,256,"SMSET",NULL);
    Print = j;
  }
}

#ifndef OPTIMIZE

typedef struct
{
  long      Op[128+1024];
  long      R_m[ 4], W_m[4];
  long      R_s[16], R_l[8], R_i;
  long      W_s[16], W_l[8];
  long      I_s[16], I_l[8], O_s[16], O_l[16], I_m[4], O_m[4];
  long long I;
} stats;

typedef struct tag
{
  char       *t;
  stats       s;
  struct tag *n;
} stat_stack;

stat_stack *StatStackTop;

char *stat_name(stat_stack *q)
{
  return q->t;
}

void clear_stats(stat_stack *q)
{
  int i;

  if (q){
    memcpy(q->s.Op    ,0,sizeof(Op ));
    memcpy(q->s.R_m   ,0,sizeof(R_m));
    memcpy(q->s.W_m   ,0,sizeof(W_m));
    memcpy(&(q->s.R_i),0,sizeof(R_i));
    memcpy(q->s.R_s   ,0,sizeof(R_s));
    memcpy(q->s.R_l   ,0,sizeof(R_l));
    memcpy(q->s.W_s   ,0,sizeof(W_s));
    memcpy(q->s.W_l   ,0,sizeof(W_l));
    memcpy(q->s.I_s   ,0,sizeof(I_s));
    memcpy(q->s.I_l   ,0,sizeof(I_l));
    memcpy(q->s.O_s   ,0,sizeof(O_s));
    memcpy(q->s.O_l   ,0,sizeof(O_l));
    memcpy(q->s.I_m   ,0,sizeof(I_m));
    memcpy(q->s.O_m   ,0,sizeof(O_m));
    memcpy(&(q->s.I)  ,0,sizeof(I  ));
  } else {
    for (I = i = 0; i < 128+1024; i++){
      Op[i] = 0;
      if (i < 16){
        I_s[i] = O_s[i] = R_i = R_s[i] = W_s[i] = 0;
        if (i < 8){
          I_l[i] = O_l[i] = R_l[i] = W_l[i] = 0;
          if (i < 4){
            I_m[i] = O_m[i] = R_m[i] = W_m[i] = 0;
          }
        }
```

```
      }
    }
  }
}

void combine_stats(stat_stack *q)
{
  int i;

  if (q||(q = StatStackTop)){
    q->s.I += I;
    for (i = 0; i < 128+1024; i++){
      q->s.Op[i] += Op[i];
      if (i < 16){
        q->s.I_s[i] += I_s[i];
        q->s.O_s[i] += O_s[i];
        q->s.R_i    += R_i;
        q->s.R_s[i] += R_s[i];
        q->s.W_s[i] += W_s[i];
        if (i < 8){
          q->s.I_l[i] += I_l[i];
          q->s.O_l[i] += O_l[i];
          q->s.R_l[i] += R_l[i];
          q->s.W_l[i] += W_l[i];
          if (i < 4){
            q->s.I_m[i] += I_m[i];
            q->s.O_m[i] += O_m[i];
            q->s.R_m[i] += R_m[i] ;
            q->s.W_m[i] += W_m[i] ;
          }
        }
      }
    }
  }
}

void save_stats(stat_stack *q)
{
  memcpy(q->s.Op    ,Op  ,sizeof(Op ));
  memcpy(q->s.R_m   ,R_m ,sizeof(R_m));
  memcpy(q->s.W_m   ,W_m ,sizeof(W_m));
  memcpy(&(q->s.R_i),&R_i,sizeof(R_i));
  memcpy(q->s.R_s   ,R_s ,sizeof(R_s));
  memcpy(q->s.R_l   ,R_l ,sizeof(R_l));
  memcpy(q->s.W_s   ,W_s ,sizeof(W_s));
  memcpy(q->s.W_l   ,W_l ,sizeof(W_l));
  memcpy(q->s.I_s   ,I_s ,sizeof(I_s));
  memcpy(q->s.I_l   ,I_l ,sizeof(I_l));
  memcpy(q->s.O_s   ,O_s ,sizeof(O_s));
  memcpy(q->s.O_l   ,O_l ,sizeof(O_l));
  memcpy(q->s.I_m   ,I_m ,sizeof(I_m));
  memcpy(q->s.O_m   ,O_m ,sizeof(O_m));
  memcpy(&(q->s.I)  ,&I  ,sizeof(I  ));
}

void restore_stats(stat_stack *q)
{
  memcpy( Op ,q->s.Op    ,sizeof(Op ));
  memcpy( R_m,q->s.R_m   ,sizeof(R_m));
  memcpy( W_m,q->s.W_m   ,sizeof(W_m));
  memcpy(&R_i,&(q->s.R_i),sizeof(R_i));
  memcpy( R_s,q->s.R_s   ,sizeof(R_s));
  memcpy( R_l,q->s.R_l   ,sizeof(R_l));
  memcpy( W_s,q->s.W_s   ,sizeof(W_s));
  memcpy( W_l,q->s.W_l   ,sizeof(W_l));
  memcpy( I_s,q->s.I_s   ,sizeof(I_s));
  memcpy( I_l,q->s.I_l   ,sizeof(I_l));
  memcpy( O_s,q->s.O_s   ,sizeof(O_s));
```

```
  memcpy( O_l,q->s.O_l   ,sizeof(O_l));
  memcpy( I_m,q->s.I_m   ,sizeof(I_m));
  memcpy( O_m,q->s.O_m   ,sizeof(O_m));
  memcpy( &I ,&(q->s.I)  ,sizeof(I  ));
}

void push_stats(char *s)
{
  stat_stack *p = StatStackTop;

  StatStackTop    = (stat_stack *)malloc(sizeof(stat_stack));
  StatStackTop->n = p;
  StatStackTop->t = s;
  save_stats(StatStackTop);
}

stat_stack *pop_stats()
{
  stat_stack *p = StatStackTop;

  if (!p) return NULL;
  StatStackTop = p->n;
  restore_stats(p);
  return p;
}

void Report(stat_stack *q, char *st)
{
  int i;  long mr,mw,rs,ws,rl,wl,ls,ss,ll,sl,lm,sm;  double r,s,t;  stat_stack *p;

  if (q){
    push_stats(NULL); p = pop_stats();
    restore_stats(q);
  }
  r = s = t = mr = mw = rs = ws = rl = wl = ls = ss = ll = sl = lm = sm = 0;
  printf("\n-------------------------------------------------------\n");
  if (q ) printf("%s",stat_name(q));
  if (st) printf("%s",st);
  printf("\n-------------------------------------------------------\n");
  if (I){
    printf("Operation                  Count                    Time\n");
    for (i = 0; i < 1024+128; i++){
      s += (t = ((Op[i])? Op[i]*Time[i]: 0.)/1e9);
      if (i > 59) r += t;
      if (Op[i]) printf("%-15s %16ld %23.9lf\n",((i<128)?Ops[i]:Def[i-128]),Op[i],t);
    }
  }
  printf("                       ----------  ----------------------\n");
  printf("Total                  %10lld %23.9lf\n",I,s);
  printf("(      I/O)                       %23.9lf\n",r);
  printf("(immediate)            %10ld\n\n",R_i);
  for (i = 0; i < 16; i++){
    rs += R_s[i]; ws += W_s[i]; ls += I_s[i]; ss += O_s[i];
    if (i < 8){
      rl += R_l[i];  wl += W_l[i]; ll += I_l[i]; sl += O_l[i];
    }
    if (i < 4){
      mr += R_m[i];  mw += W_m[i]; lm += I_m[i]; sm += O_m[i];
    }
  }
  printf("Register       Reads      Writes       Loads      Stores\n");
  for (i = 0; i < 16; i++){
    printf("%3d  (8)  %10ld  %10ld  %10ld  %10ld\n",i,R_s[i],W_s[i],I_s[i],O_s[i]);
  }
  printf("          ----------  ----------  ----------  ---------- \n");
  printf("          %10ld  %10ld  %10ld  %10ld\n",rs,ws,ls,ss);
  printf("          ----------  ----------  ----------  ---------- \n");
  for (i = 0; i < 8; i++){
```

```
    printf("%3d (16)  %10ld  %10ld  %10ld  %10ld\n",i,R_l[i],W_l[i],I_l[i],O_l[i]);
  }
  printf("          ----------  ----------  ----------  ---------- \n");
  printf("          %10ld  %10ld  %10ld  %10ld\n",rl,wl,ll,sl);
  printf("          ----------  ----------  ----------  ---------- \n");
  printf("          %10ld  %10ld  %10ld  %10ld\n\n",rs+rl,ws+wl,ls+ll,ss+sl);
  for (i = 0; i < 4; i++){
    printf("%3d  (m)  %10ld  %10ld  %10ld  %10ld\n",i,R_m[i],W_m[i],I_m[i],O_m[i]);
  }
  printf("          ----------  ----------  ----------  ---------- \n");
  printf("          %10ld  %10ld  %10ld  %10ld\n",mr,mw,lm,sm);
  printf("\n-------------------------------------------------------\n");
  if (q){
    restore_stats(p);
    free(p);
  }
}

#define DEFAULT_CB_PRINT0(x,y,z)                                    \
          sprintf(s,"%s (%lld) @ %d",z,I-1,Du);                     \
          print_##x(y,s,NULL);                                      \

#define DEFAULT_CB_PRINT1(x,y,z)                                    \
          sprintf(s,"%s (%lld) @ %d,%d",z,I-1,Du,Dv);               \
          print_##x(y,Du,Dv,8,8,s,NULL)

#define DEFAULT_CB(x,y,z) void default_cb##x##y()                   \
{                                                                   \
  static state State;  char s[64];                                  \
                                                                    \
  Save(State);                                                      \
  putchar('\n');                                                    \
  DEFAULT_CB_PRINT##z(x,y,"after");                                 \
  Restore_##x(y);                                                   \
  DEFAULT_CB_PRINT##z(x,y,"before");                                \
  Restore(State);                                                   \
}

DEFAULT_CB(s,0,0)
DEFAULT_CB(s,1,0)
DEFAULT_CB(s,2,0)
DEFAULT_CB(s,3,0)
DEFAULT_CB(s,4,0)
DEFAULT_CB(s,5,0)
DEFAULT_CB(s,6,0)
DEFAULT_CB(s,7,0)
DEFAULT_CB(s,8,0)
DEFAULT_CB(s,9,0)
DEFAULT_CB(s,10,0)
DEFAULT_CB(s,11,0)
DEFAULT_CB(s,12,0)
DEFAULT_CB(s,13,0)
DEFAULT_CB(s,14,0)
DEFAULT_CB(s,15,0)
DEFAULT_CB(l,0,0)
DEFAULT_CB(l,1,0)
DEFAULT_CB(l,2,0)
DEFAULT_CB(l,3,0)
DEFAULT_CB(l,4,0)
DEFAULT_CB(l,5,0)
DEFAULT_CB(l,6,0)
DEFAULT_CB(l,7,0)
DEFAULT_CB(m,0,1)
DEFAULT_CB(m,1,1)
DEFAULT_CB(m,2,1)
DEFAULT_CB(m,3,1)

#define COMMENT(c)        if (LogFile) fprintf(LogFile,"%s",c);
```

```
#define PASTE(x,y)        x##y
#define PASTE3(x,y,z)     x##y##z
#define APPLY(x,y,z)      x(y,z)
#define APPLY3(w,x,y,z)   w(x,y,z)
#define TShort(x)         s
#define TLong(x)          l
#define TMatrix(x)        m
#define VShort(x)         x
#define VLong(x)          x
#define VMatrix(x)        x


#define NoOp(x)           { Noop = 1; x; Noop = 0; }
#define Begin_Op(x,y,z,w) int ZZ = 1, ZZZ = Z; Op[128+x]+=Z; if (!Def[x]) Def[x] = #y; Time[128+x] = z; \
                          Log_o(128+x); label##x: w; if (!ZZ){ return; } Z = 0

#define End_Op(x)         I += (Z=ZZZ); ZZ = 0; goto label##x;
#define Begin_Watch       { push_stats(NULL); clear_stats(NULL); COMMENT("\nBegin_Watch"); }
#define End_Watch(x)      { Report(NULL,x); combine_stats(NULL); free(pop_stats()); COMMENT("\nEnd_Watch"); }
#define Suspend           Z = 0
#define Resume            Z = 1
#define Read_Immediate(i) if (ZZ){ RI(i,"r"); }
#define Read_Short(i)     if (ZZ){ RS(i,"r"); }
#define Write_Short(i)    if (ZZ){ WS(i,"w"); } else { CS(i) diff_s(i); }
#define Read_Long(i)      if (ZZ){ RL(i,"r"); }
#define Write_Long(i)     if (ZZ){ WL(i,"w"); } else { CL(i) diff_l(i); }
#define Read_Matrix(i)    if (ZZ){ RM(i,"r"); }
#define Write_Matrix(i)   if (ZZ){ WM(i,"w"); } else { CM(i) diff_m(i); }
#define Read_Short_(i)    if (ZZ){ RS(i,"*r"); }
#define Write_Short_(i)   if (ZZ){ WS(i,"*w"); } else { CS(i) diff_s(i); }
#define Read_Long_(i)     if (ZZ){ RL(i,"*r"); }
#define Write_Long_(i)    if (ZZ){ WL(i,"*w"); } else { CL(i) diff_l(i); }
#define Read_Matrix_(i)   if (ZZ){ RM(i,"*r"); }
#define Write_Matrix_(i)  if (ZZ){ WM(i,"*w"); } else { CM(i) diff_m(i); }
#define Trace(x,y)        { if (y) WatchReg(x,y); else APPLY(WatchReg,x,APPLY3(PASTE3,default_cb,T##x,V##x)); }
#define Untrace(x)        WatchReg(x,NULL)

#define Trace_S           Trace(Short( 0),NULL); \
                          Trace(Short( 1),NULL); \
                          Trace(Short( 2),NULL); \
                          Trace(Short( 3),NULL); \
                          Trace(Short( 4),NULL); \
                          Trace(Short( 5),NULL); \
                          Trace(Short( 6),NULL); \
                          Trace(Short( 7),NULL); \
                          Trace(Short( 8),NULL); \
                          Trace(Short( 9),NULL); \
                          Trace(Short(10),NULL); \
                          Trace(Short(11),NULL); \
                          Trace(Short(12),NULL); \
                          Trace(Short(13),NULL); \
                          Trace(Short(14),NULL); \
                          Trace(Short(15),NULL)

#define Untrace_S         Untrace(Short( 0)); \
                          Untrace(Short( 1)); \
                          Untrace(Short( 2)); \
                          Untrace(Short( 3)); \
                          Untrace(Short( 4)); \
                          Untrace(Short( 5)); \
                          Untrace(Short( 6)); \
                          Untrace(Short( 7)); \
                          Untrace(Short( 8)); \
                          Untrace(Short( 9)); \
                          Untrace(Short(10)); \
                          Untrace(Short(11)); \
                          Untrace(Short(12)); \
```

```
                        Untrace(Short(13)); \
                        Untrace(Short(14)); \
                        Untrace(Short(15))

#define Trace_L         Trace(Long(0),NULL); \
                        Trace(Long(1),NULL); \
                        Trace(Long(2),NULL); \
                        Trace(Long(3),NULL); \
                        Trace(Long(4),NULL); \
                        Trace(Long(5),NULL); \
                        Trace(Long(6),NULL); \
                        Trace(Long(7),NULL)

#define Untrace_L       Untrace(Long(0)); \
                        Untrace(Long(1)); \
                        Untrace(Long(2)); \
                        Untrace(Long(3)); \
                        Untrace(Long(4)); \
                        Untrace(Long(5)); \
                        Untrace(Long(6)); \
                        Untrace(Long(7))

#define Trace_M         Trace(Matrix(0),NULL); \
                        Trace(Matrix(1),NULL); \
                        Trace(Matrix(2),NULL); \
                        Trace(Matrix(3),NULL)

#define Untrace_M       Untrace(Matrix(0)); \
                        Untrace(Matrix(1)); \
                        Untrace(Matrix(2)); \
                        Untrace(Matrix(3))

#define Trace_All       Trace_S; Trace_L; Trace_M
#define Untrace_All     Untrace_S; Untrace_L; Untrace_M

#define Begin_Log(x)    { if (!(LogFile = fopen(x,"w"))) Error("\ncan't open logfile %s\n",x); }
#define End_Log         if (LogFile){ fprintf(LogFile,"\n"); fclose(LogFile); }

#else

#define stat_name(x)
#define clear_stats(x)
#define combine_stats(x)
#define save_stats(x)
#define restore_stats(x)
#define push_stats(x)
#define pop_stats()
#define Report(x,y)
#define COMMENT(c)
#define NoOp(x)
#define Begin_Op(x,y,z,w)
#define End_Op(x)
#define Begin_Watch
#define End_Watch(x)
#define Suspend
#define Resume
#define Read_Short(i)
#define Write_Short(i)
#define Read_Long(i)
#define Write_Long(i)
#define Read_Matrix(i)
#define Write_Matrix(i)
#define Read_Short_(i)
#define Write_Short_(i)
#define Read_Long_(i)
#define Write_Long_(i)
#define Read_Matrix_(i)
#define Write_Matrix_(i)
```

```
#define Trace(x,y)
#define Trace_S
#define Trace_L
#define Trace_M
#define Trace_All
#define Untrace_S
#define Untrace_L
#define Untrace_M
#define Untrace_All
#define Untrace(x,y)
#define Begin_Log(x)
#define End_Log

#endif
```

l2matlab.c

```
#include <stdio.h>
#include <stdlib.h>

void error(char *s)
{
  printf("%s\n",s);
  exit(-1);
}

#define WHITE(p) ((p) < 33)

char *next(char *p)
{
  while ( WHITE(*p)) if (!p) return NULL; else p++;
  while (!WHITE(*p)) p++;
  return p;
}

char Buf[1<<16];

void make_data_files(FILE *f, FILE *o, char *n)
{
  int h = 0, i,j,k; char *p, buf[1024], name[1024];  FILE *g;

  while (fgets(p=Buf,1<<16,f)){
    if (!(p = next(p))) goto c;
    if (!(p = next(p))) goto c;
    if (p-Buf < 5)      goto c;
    if ((p[-3] == 'S')&&(p[-2] == 'E')&&(p[-1] == 'T')){
      sprintf(name,"%s_%d",n,h++);
      if (!(g = fopen(name,"w"))){
        printf("Could not write file ");
        error(name);
      }
      fprintf(o,"load %s.dat\n",name);
      switch (p[-4]){
      case 'M':
        fgets(Buf,1<<16,f); fgets(Buf,1<<16,f);
        for (i = 256; i--; ){
          fgets(Buf,1<<16,f);
          for(p = Buf, j = 256; j--; ){
            if (!(p = next(p))) error("log format error\n");
            sscanf(p,"%d",&k);
            fprintf(g,"%5d",k);
          }
          fprintf(g,"\n");
        }
        break;
      case 'V':
        fgets(Buf,1<<16,f);
        for (i = 16; i--; ){
          fgets(Buf,1<<16,f);
          for(p = Buf, j = 16; j--; ){
            if (!(p = next(p))) error("log format error\n");
            sscanf(p,"%d",&k);
            fprintf(g,"%5d",k);
          }
        }
        fprintf(g,"\n");
        break;
      default:
        error("log format error\n");
      }
      fclose(g);
      sprintf(buf,"sed -f l2doz %s > %s.dat",name,name); system(buf);
      sprintf(buf,"rm %s",name); system(buf);
```

```
    }
  c: continue;
  }
}

void make_apl_file(FILE *f, FILE *g, char *n)
{
  int h = 0; char *p,*q, name[1024];

  while (fgets(p=Buf,1<<16,f)){
    while (*p != '(') if (!*p) goto c; else p++;
    if (p-Buf < 5)             goto c;
    if ((p[-3] == 'S')&&(p[-2] == 'E')&&(p[-1] == 'T')){
      while (*p != ')') if (!*p) goto c; else p++;
      *(p++) = ',';
      sprintf(name,"%s_%d",n,h++);
      for (q = name; *q; *(p++) = *(q++));
      *(p++) = ')';
      *(p++) = '\n';
      *p = 0;
    }
  c: fprintf(g,"%s",Buf);
  }
}

int main(int argc, char *argv[])
{
  char buf[1024]; FILE *f = fopen(*++argv,"r"), *g;

  if (!f){
    printf("Could not open log file ");
    error(*argv);
  }
  sprintf(buf,"%s_apl",*argv);
  if (!(g = fopen(buf,"w"))){
    printf("Could not write file ");
    error(buf);
  }
  make_data_files(f,g,*argv); fclose(f);
  sprintf(buf,"sed -f l2apl %s > apl",*argv); system(buf);
  if (!(f = fopen("apl","r"))) error("Could not open apl file");
  make_apl_file(f,g,*argv);
  fclose(f); fclose(g);
  sprintf(buf,"sed -f l2doz %s_apl > %s_apl.m",*argv,*argv); system(buf);
  sprintf(buf,"rm apl %s_apl",*argv); system(buf);
  return 0;
}
// gcc -Wall -gdwarf-2 -g3 -o l2matlab l2matlab.c -lm
```

l2apl

```
/^$\|^[^0123456789]/d
s/^\([0123456789]* \)\([^ ]*\) \(.*\)$/\2(\3)/g
s/(\*[^ ]*/(/g
s/ \*[^ ]*/ /g
s/ [ ]*/,/g
s/(,/(/
s/,$/)/
s/ri\|rs\|rl\|rm\|ws\|wl\|wm//g
```

l2doz

```
s/$/\r/
```