```
/***************************************************************************
 * hwgp: Hardware implementation of Genetic Programming. Written in
 * Handel-C for compilation to a Xilinx FPGA
 *
 * Pete Martin 2001
 *
 ***************************************************************************
 * This file implements the following versions:
 *
 * -DHANDELC     for compling under handelc, both simulator and edif
 * -DDEBUG       for compliling for the handelc simulator
 *               if this is not defined but HANDELC is then it will compile
 *               for EDIF
 *
 * -DPOPSIZE     Overrides the default population size
 * -DMAXPROGLEN  Overrides the default maximum progam length
 * -DPROBLEM=xxx Set the problem to compile for
 *               XOR
 *               ANT
 ***************************************************************************/

#define VERSION "1.0"

/*#define PRESET*/

#ifndef HANDELC
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#else
#ifdef DEBUG

set clock = external "P35";

#else
/*
 * Include RC1000-PP support header file
 */
#define PP1000_BOARD_TYPE PP1000_V2_VIRTEX
#define PP1000_CLOCKRATE  10
#define PP1000_CLOCK    PP1000_MCLK
#define PP1000_32BIT_RAMS
#define PP1000_DIVIDE4
#include <pp1000.h>

#endif  /* DEBUG */
#endif  /* !HANDELC */

#include <stdlib.h>

#include <hwgph.h>

/* Define the internal population storage
* this is a subset of the full population that is held in external
* static RAM. These are used for internal operations.
* There are 2 separate sets of individuals in ram here.
* Each one is used for a different phase of the pipeline
* The working ram is arranged as WORDLEN bit words. The evaluation
* function will decode this into instruction sized chunks as needed.
* MPRAMs are used here, not because we want to access the rams twice in once
* clock cycle, but because this allows better routing performance.
*/
mpram {
    RAM Word ReadWriteA[MAXNODES];
    RAM Word ReadWriteB[MAXNODES];
} workingPop  [MAXPHASE*MAXPAR] WITHBLOCK;

mpram {
```

```
    RAM Bool  ReadWriteA[POPSIZE];
    RAM Bool  ReadWriteB[POPSIZE];
}populationControl WITHBLOCK;

mpram {
    RAM Word  ReadWriteA[POPSIZE];
    RAM Word  ReadWriteB[POPSIZE];
}populationFitness WITHBLOCK;

mpram {
    RAM Word  Read[POPSIZE];
    RAM Word  ReadWriteB[POPSIZE];
}populationLen WITHBLOCK;

/* Storage for the population control information
* This can be processed in parallel with the main population */
mpram {
    RAM Word   ReadWriteA[MAXPAR] ;
    RAM Word   ReadWriteB[MAXPAR] ;
} workingLen [MAXPHASE] WITHBLOCK;

mpram {
    RAM    PopIndex  ReadWriteA[MAXPAR];
    RAM    PopIndex  ReadWriteB[MAXPAR];
} workingprogNum[MAXPHASE] WITHBLOCK;

mpram {
    RAM Word      Read[MAXPAR];
    RAM Word      Write[MAXPAR];
}workingFitness      [MAXPHASE]  WITHBLOCK;


RAM Word      workingParentFitness[MAXPHASE] [MAXPAR] WITHBLOCK;
RAM Word      workingParentLen[MAXPHASE] [MAXPAR] WITHBLOCK;

UINT32        cycles; /* Count of cycles executed in main program */
Bool          running; /* Control the measurement of cycles */

#if 1
#if DEBUG
/* Provide some storage for sram if we are using the simulator */
ram UINT32  sram[4][POPSIZE*MAXNODES] WITHBLOCK;
#define RAMW (log2ceil(POPSIZE*MAXNODES))

macro proc PP1000WriteBank0(C_UINT addr, C_UINT val)
{
    sram[0][adju(addr,RAMW)]=val;
}

macro proc PP1000WriteBank1(C_UINT addr, C_UINT val)
{
    sram[1][adju(addr,RAMW)]=val;
}


macro proc PP1000ReadBank0(C_UINT val, C_UINT addr)
{
    val = sram[0][adju(addr,RAMW)];
}

macro proc PP1000ReadControl(x)
{
    x=1;
}

macro proc PP1000WriteStatus(x)
{

}

#endif


#if defined ANT
```

```
static UINT32 initialMap[32] = {
  0x0000000e,
  0x00000008,
  0x0e000008,
  0x21000008,
  0x21000008,
  0x00601f78,
  0x20001000,
  0x00101000,
  0x00101000,
  0x20101000,
  0x00100000,
  0x00001000,
  0x20001000,
  0x00101000,
  0x1c101000,
  0x00820000,
  0x00000000,
  0x00001000,
  0x01011000,
  0x08011000,
  0x00011000,
  0x00011000,
  0x04001000,
  0x00801000,
  0x00010f98,
  0x00010002,
  0x00010002,
  0x00007f02,
  0x00000082,
  0x00000080,
  0x0000003c,
  0x00000000,
};

/* The active maps for the evalProgs to use */
RAM UINT32    map[MAXPAR][GRIDX];

/* Init the maps from the initialMap */
macro proc initMap(C_UINT i)
{
    unsigned int p;
    Bool     done;

    p = 0;
    do {
        map[i][p]=initialMap[p];
        PAR {
            p++;
            done = (p == 31);
        }
    } while(!done);
}
#endif

/*
* Define some control variables
*/
#if !defined HANDELC
Bool tree    = 0;   /* If true then print program tree */
Bool newseed= 0;    /* If true then ANSI_C version will generate seed */
#endif
Bool stopon = 0;    /* If true then stop if 100% correct prog */

UINT8 nbreed, nxover, ncopy, nmutate, nrand;

Phase phBreed;
Phase phEval;
Phase phWriteback;
unsigned char good = 0;
int   dumpProg      = 0;
int   dumpFittest   = 0;
int   dumpLengths   = 0;
int   verbose       = 0;
```

```
int   dumpFitlen   = 0;

/*
 * Generic GP  function prototypes
 */
macro proc initPop();
macro proc writeBack();
//macro proc writeBackInd(C_UINT phase, C_UINT ind);
//macro proc readIn(C_UINT i, C_UINT ind);
void readIn(PopIndex, WorkIndex);
macro proc findBest(C_UPTR best);
macro proc RandomGen();
macro proc evalProg(C_UINT indx);
macro proc CycleCount();

/*
 * output channel
 */
#ifdef HANDELC
/*chanout stdout;*/
chanout Short logout;
macro proc dooutdata(C_UINT c, C_UINT v)
{
#if DEBUG
    logout ! (Short)c;
    logout ! (Short)v;
#else
    PP1000WriteStatus(c);
    PP1000WriteStatus(v);
#endif
}
#endif

/*********************************************************************
 * Random number generator for Handelc
 *
 * Uses a shift register with taps and feedback
 *********************************************************************/
#ifdef HANDELC
#define RANDWIDTH  32  /* Number of bits in the generator
                          This is greater than the biggest number
                          wanted from the generator*/
#endif

#define TAP1       0
#define TAP2       5
#define TAP3       9
#define TAP4       20
#define TAP5       30
#define TAP6       31




/* Channel for getting the seed */
#ifdef HANDELC
#ifdef SIMULATE
chanin  RandReg seedIn with {infile = "c:/tmp/seed.dat"};
chanout RandReg randout with {outfile = "c:/tmp/random.dat"};
#endif
#endif
#if defined HANDELC
/****************************************************************************
 * Name:     random
 *
 * Purpose:  Generate a pseudo-random number
 *
 * Inputs:   Nothing
 *
 * Returns:  Nothing
 *
 * Notes:    The number is stored in the static variable randReg
 *
 ****************************************************************************/
```

```
RandReg randReg;     /* The register used for generating the number */
macro proc random()
{
  static unsigned int BITW  oldbit0;
  unsigned int BITW bit0;
  par {
    bit0 = randReg[TAP1] ^
      randReg[TAP2] ^
      randReg[TAP3] ^
      randReg[TAP4] ^
      randReg[TAP5] ^
      randReg[TAP6];
    /* Shift by using all bits except top bit and adding bit 0 */
    randReg = randReg[RANDWIDTH-2:0]@oldbit0;
    oldbit0=bit0;
  }
}

macro proc RandomGen()
{
  while(1) {
    random();
  }
}

macro proc CycleCount()
{
  while(1) {
    if(running) {
        cycles++;
    } else {
        delay;
    }
  }
}

/***************************************************************************
 * Name:     randseed
 *
 * Purpose:  Seed the random number generator
 *
 * Inputs:   Nothing
 *
 * Returns:  Nothing
 *
 ***************************************************************************/
macro proc randseed()
{
#ifdef SIMULATE
  RandReg  seed;
  seedIn ? seed;
  randReg = seed;
#else
  PP1000RequestMemoryBank(1);
  PP1000ReadBank0(randReg,0);
  PP1000ReleaseMemoryBank(1);
#endif
}

/***************************************************************************
 * Name:     randXXXXX functions
 *
 * Purpose:  Generate random numbers of given widths and limits
 *
 * Inputs:   None
 *
 * Returns:  A random number of the correct width
 *
 * Notes:    Each function requires 1 clock cycle for the random() call
 *
 ***************************************************************************/

/* Return a value that is between 0 and MAXPROGLEN */
macro expr randPC() = 0@randReg[PROGW-1:0];
```

```
macro expr randLen() = 0@randReg[PROGW-1:0]|1;

macro proc randMethod(method)
{
#define mask1 0x1f
#define mask2 0xfc
#define mask3 (~mask1)
#define mask4 (~mask2)

  UINT8 v, v1, v2, v3, v4;

  v = adju(randReg,8);
  PAR {
    v1 = v&mask1;
    v2 = v&mask2;
    v3 = v&mask3;
    v4 = v&mask4;
    method = XOVER;   /* Default method */
  }

  /* This long winded code generates shallow logic as
     compared to a long if-then-else chain */

  /* Only override if we have the less probable cases */
  if(!v3&&v1)
    method = MUTATE;
  else
    delay;
  if(!v4&&v2)
    method =  COPY;
  else
    delay;
}

macro expr randPopIndex() = randReg[POPW-1:0];
#endif


macro proc PP1000WriteBank0func(C_UINT addr, C_UINT val)
{
    UINT32 v;

    do {
        PP1000WriteBank0(addr, val);
    PP1000ReadBank0(v,addr);
    } while(v != val);

}

macro proc PP1000WriteBank1func(C_UINT addr, C_UINT val)
{
    UINT32 v;

    do {
        PP1000WriteBank1(addr, val);
    PP1000ReadBank1(v,addr);
    } while(v != val);

}


macro proc mutate(C_UINT ind)
{
    Index           address;
    Phase           ph;

    ph = phBreed;
    /* Select an instruction to mutate */
    address = randPC() & adju(WorkingLenRead(ph,ind), IDXW);
    WorkingPopWrite(ph, ind, address, genNode());
#if 0
    printf("MASK = 0x%x\n", (FUNCMASK << (REGBITS+REGBITS)) | (REGMASK << REGBITS) | 
    REGMASK);
```

```
    printf("GETOPCODEMASK = 0x%x\n", FUNCMASK << (REGBITS+REGBITS));
    printf("GETEA1MASK = 0x%x\n", (((REGMASK << REGBITS))));
    printf("GETEA2MASK = 0x%x\n", (REGMASK));
#endif
}


/* Copy individual at work index w to work index w+1
* w is used directly and is always even (bit 0 == 0) and
w+1 is always odd (bit 0 == 1) */
macro proc copyInd(C_UINT w)
{
  Word    val;
  Index_1     count;
  Index       i;
  Phase       ph;
  Bool        done;

  PAR {
    i  = 0;
    count = 0;
    ph = phBreed;
  }

  do {
    WorkingPopRead(ph,w, i, C_ADDR val);
    WorkingPopWrite(ph,w | 1, i, val);
    PAR {
        count++;
        i++;
        done = (count == MAXNODES-1);
    }
  }while(!done);
}

/* Original saturating crossover */
#if defined TRUNCATE
#warning Truncate method selected
macro proc xover(C_UINT w)
{
  Word  val, val2;
  Word  len1;
  Word  len2;
  Word  newlen1, newlen2;
  PC      x1, x2;
  PC      i1, i2;
  Index  idx1, idx2;
  Index  count1, count2;
  Phase    ph;

  ph = phBreed;
  /*
   * Choose two crossover points at random and get the program lengths
   */


  PAR {
    x1 = randPC();
    len1 = WorkingLenRead(ph,w);
  }
  PAR {
    x2 = randPC();
    len2 = WorkingLenRead(ph,w | 1);
  }

    /*
    * Crudely adjust the crossover points so that they lie within the
    * individual
    */
  PAR {
#if defined BOOLOP
    x1 &= adju(len1, PROGW);
    x2 &= adju(len2, PROGW);
#else
```

```
    x1 %= adju(len1, PROGW);
    x2 %= adju(len2, PROGW);
#endif
}
  /*
  * calculate the starting points for crossover
  */
    PAR {
        i1 = x1;
        i2 = x2;
    }

  /*
  * Calculate the count of instructions for copying
  * count1 correspnds to individual 1
  */
  PAR {
    count1 = adju(len1, IDXW) - i1;
    count2 = adju(len2, IDXW) - i2;
  }
  /*
   * Copy until both counts have been exhausted
   */
  do {
    PAR {
        idx1 = i1;
        idx2 = i2;
    }
    PAR {
        WorkingPopRead(ph, w, idx1, C_ADDR val);
        WorkingPopRead(ph,w | 1, idx2, C_ADDR val2);
    }
    PAR {
        WorkingPopWrite(ph,w,idx1, val2);
        WorkingPopWrite(ph, w | 1, idx2, val);
    }
    /*
    * Calculate the counts, but don't let them go below zero
    * and calculate the next indexes, but don't let them go beyond MAXNODES
    */
    PAR {
        if(count1)
            count1--;
        else
            delay;
        if(count2)
            count2--;
        else
            delay;
        if(i1 != MAXPROGLEN-1)
            i1++;
        else
            delay;
        if(i2 != MAXPROGLEN-1)
            i2++;
        else
            delay;
    }
  }while( count1|| count2);
  /*
   * Adjust the lengths - broken into 2 bits to speedup the logic
   */
   PAR {
        newlen1 = adju(x1 + (adju(len2,PROGW)), WORDW);
        newlen2 = adju(x2 + (adju(len1,PROGW)), WORDW);
   }
   PAR {
        newlen1 = newlen1 - adju(x2, WORDW);
        newlen2 = newlen2 - adju(x1, WORDW);
   }
   //   printf("x1=%d x2=%d len1=%d, len2=%d, newlen1=%d, newlen2=%d\n",
   //     x1,x2,len1,len2,newlen1,newlen2);
#if defined BOOLOP
    /* Guard against zero length programs whcih are no good at all */
```

```
    while(!len1) {
      len1 = randLen();
    }

    while(!len2) {
      len2 = randLen();
    }
#endif

    /* Saturate the sizes to the maximum */
    if(newlen1>MAXNODES-1) newlen1=MAXNODES-1;
    if(newlen2>MAXNODES-1) newlen2=MAXNODES-1;

    PAR {
        WorkingLenWrite(ph, w)     =  newlen1;
        WorkingLenWrite(ph, w | 1) =  newlen2;
    }
}


#elif defined LIMITED
#warning Limiting crossover selected
/* New improved crossover */
macro proc xover(C_UINT w)
{
  Word  val, val2;
  Word  len1;
  Word  len2;
  Word  olap1, olap2;
  PC      x1, x2;
  PC      i1, i2;
  Index  idx1, idx2;
  Index  count1, count2;
  Phase    ph;
  Word   newlen1, newlen2;
  Bool   done1, done2;
  Bool   olapDone1, olapDone2;



  ph = phBreed;

  do {
    /*
     * Choose two crossover points at random and get the program lengths
     */
    PAR {
      x1 = randPC();
      len1 = WorkingLenRead(ph,w);
    }
    PAR {
      x2 = randPC();
      len2 = WorkingLenRead(ph,w | 1);
    }

    /*
     * Adjust the crossover points so that they lie within the
     * individual. This is modulus by remainder.
     */
    PAR {
      while(x1>=adju(len1,PROGW)) {
         x1 -= adju(len1,PROGW);
      }
      while(x2>=adju(len2,PROGW)) {
          x2 -= adju(len2,PROGW);
      }
    }

    /* Get size of new individuals */
    newlen1 = adju(x1, WORDW) + (len2 - adju(x2, WORDW));
    newlen2 = adju(x2, WORDW) + (len1 - adju(x1, WORDW));

    /* See if the newlens are bigger than MAXNODES-1 */
    olap1 = newlen1 & ~(MAXNODES-1);
```

```
    olap2 = newlen2 & ~(MAXNODES-1);
    olapDone1 = !(olap1 == 0);
    olapDone2 = !(olap2 == 0);
#if defined DUALXOVER
#warning DUAL Xover
  } while(olapDone1 || olapDone2);
#else
#warning SINGLE xover
  } while(olapDone1 && olapDone2);
#endif

  /*
   * calculate the starting points for crossover
   */
  PAR {
    i1 = x1;
    i2 = x2;
  }

  /*
   * Calculate the count of instructions for copying
   * count1 corresponds to individual 1
   */
  PAR {
    if(!olap1)
      count1 = adju(len2, IDXW) - x2;
    else
      count1 = 0;
    if(!olap2)
      count2 = adju(len1, IDXW) - x1;
    else
      count2 = 0;
    idx1 = x1;
    idx2 = x2;
  }
  /*
   * Copy until both counts have been exhausted
   */
  do {
    PAR {
        done1 = (count1 == 0);
        done2 = (count2 == 0);
    }
    PAR {
      WorkingPopRead(ph, w, idx1, C_ADDR val);
      WorkingPopRead(ph,w | 1, idx2, C_ADDR val2);
    }
    PAR {
      if(!done1) {
    WorkingPopWrite(ph,w,idx1, val2);
    count1--;
    idx1++;
      }
      if(!done2) {
    WorkingPopWrite(ph, w | 1, idx2, val);
    count2--;
    idx2++;
      }
      }
  }while( count1 || count2);

  PAR {
    if(!olap1)
      WorkingLenWrite(ph, w) =  newlen1;
    if(!olap2)
      WorkingLenWrite(ph, w | 1) =  newlen2;
    }
}

#else
#error Please specify TRUNCATE or LIMITED crossover
#endif
/*****************************************************************************
 * Name:     selection
```

```
 *
 * Purpose:  Selects MAXPAR individuals for breeding and copies them from
 *           external SDRAM into working ram
 *
 * Inputs:   Nothing
 *
 * Returns:  Nothing
 *
 * Notes:    Uses the phSelect to index into the required working memory
 *           No attempt is made to see if we overselect an individual.
 *           We only allow individuals that are not in workingPop
 *
 *       The selection is tournament and uses a tournament size of 2.
 *
 ***************************************************************************/
macro proc selection()
{
  WorkIndex    w;
  PopIndex     i, i1,i2;
  Word         fit1,fit2;
  Phase        ph;
  Word         fit;
  Word         len;

#if defined HANDELC && !defined DEBUG
    interface fastle(unsigned 1 A_LT_B)
        FastLt(Word A=fit1, Word B=fit2) with {busformat="B<I>"};
#endif

  PAR {
    w = 0;
    ph = phBreed;
  }

  do {
    /*
     * Find 2 individuals that are not the same and that are not in
     * workingPop
     */
    do {
      i1 = randPopIndex();
      fit1 = populationFitness.ReadWriteA[i1];
    } while(populationControl.ReadWriteA[i1]);

    do {
      i2 = randPopIndex();
      fit2 = populationFitness.ReadWriteA[i2];
    } while(populationControl.ReadWriteA[i2] || i1 == i2);
    /*
     * Choose the one with best fitness. 0 is best!
     */
    i = UnsignedLt(fit1, fit2, MAXFITW) ? i1 : i2;
    /*
     * Read it into workingPop and remember the fitness in the parent array
     */
    readIn(i,w);
    fit = adju(workingFitness[ph].Read[w], WORDW);
    len = adju(workingLen[ph].ReadWriteA[w], WORDW);
    workingParentFitness[ph][w] = fit;
    workingParentLen[ph][w] = len;
    w++;
  }while(LIMIT(w,MAXPAR));
}



/****************************************************************************
 * Name:     breed
 *
 * Purpose:  Performs the breeding operations on the working ram pointed to
 *           by phBreed
 *           Once this has finished, there will be MAXPAR new individuals
 *           to evaluate
 *
```

```
 * Inputs:   Nothing
 *
 * Returns:  Nothing
 *
 * Notes:    Since crossover and copy use 2 individuals, we arrange for
 *           all breed operations to operate on 2 individuals.
 *           This means that there are MAXWORK/2 breed operations.
 *           The individuals for breeding have already been placed in pairs
 *           in working memory, at addresses i, i+1
 ***************************************************************************/
macro proc breed()
{
  WorkIndex count1, count2;
  WorkIndex_1 loop;
  Method    method;

#if !defined PRESET
    /* Count1 and count2 get incremented twice in the do loop */
  PAR {
    count1 = 0;
    count2 = 0;
    loop   = 0;
  }

  do {
    /*
     * Select breed method
     */
    PAR {
        randMethod(C_ADDR method);
        count2++;    /* Now count 2 = count1+1 */
    }
    switch(method) {
        case MUTATE:
            mutate(count1);
            PAR {
                mutate(count2);
                nmutate++;
            }
            break;
        case XOVER:
            PAR {
                xover(count1);
                nxover++;
            }
            break;
        default:
            PAR {
                copyInd(count1);
                ncopy++;
            }
            break;
    }
    loop+=2;
  }while(!BITSET(loop, MAXPAR));
#endif
}

/***************************************************************************
 * Name:     replacement
 *
 * Purpose:  Replaces an individual with the newly evaluated individual.
 *
 * Inputs:   The PAR number
 *
 * Returns:  Nothing
 *
 ***************************************************************************/
macro proc replacement()
{
    /* Decide whether the evaluated individual will replace
     * the corresponding w'th parent */
    WorkIndex w;
    Phase     ph;
```

```
    Word      myFit;
    Word      parentFit;
    Bool      better;
    Bool      shorter;
    Word      myLen, parentLen;

#if defined HANDELC && !defined DEBUG
    interface fastle(unsigned 1 A_LE_B)
        FastLe(Word A=myFit, Word B=parentFit) with {busformat="B<I>"};
#endif

    PAR {
      w = 0;
      ph = phEval;
    }

    do {
        /* See if new individual is better than or the same as it's parent */
        PAR {
            myFit = workingFitness[ph].Read[w];
            parentFit = workingParentFitness[ph][w];
        myLen = WorkingLenRead(ph,w);
        parentLen = workingParentLen[ph][w];
        }
    //  printf("mylen = %d, parent = %d\n", myLen, parentLen);
        better = UnsignedLe(myFit,parentFit, MAXFITW);
    shorter = UnsignedLe(myLen, parentLen, 32);
        if(better) {
            delay;   /*
                      * Yes it is so leave the indicator set
                      * so it gets written back
                      */
        } else {
                    /*
                     * The new program is worse than the worst parent so discard it
                     * by clearing the control bit so it doesn't get written back
                    */
            populationControl.ReadWriteB[WorkingprogNumWrite(ph,w)] = 0;
        }
        w++;
    } while(LIMIT(w,MAXPAR));
}

/****************************************************************************
 * Name:     evaluate
 *
 * Purpose:  Evaluates the fitness of all individuals in working ram pointed
 *           to by phEval;
 *
 * Inputs:   Nothing
 *
 * Returns:  Nothing
 *
 ****************************************************************************/
macro proc evaluate()
{
  int w;

  FORPAR(w=0; w<MAXPAR;w++) {
    evalProg(w);
  }
}


/****************************************************************************
 * Name:     findBest
 *
 * Purpose:  Locates the best individual in the population
 *
 * Inputs:   Nothing
 *
 * Returns:  The individual number
 *
 ****************************************************************************/
```

```
macro proc findBest(C_UPTR best)
{
    PopIndex_1      count;
    Word            fit, bestf, len, shortest;

    PAR {
        bestf = -1;
    shortest = -1;
        count = 0;
    }

    do {
        fit = populationFitness.ReadWriteA[adju(count, POPW)];
    len = populationLen.Read[adju(count, POPW)];
        if(UnsignedLt(fit, bestf, MAXFITW) &&
       UnsignedLt(len, shortest, MAXFITW)){
            bestf = fit;
        shortest = len;
            C_PTR best  = adju(count, POPW);
        } else {
            delay;
        }
        count++;
    } while(!BITSET(count,POPSIZE));
}


macro proc donl()
{
#if !defined HANDELC
  printf("\n");
#endif
}

/****************************************************************************
 * Name:     printInstruction
 *
 * Purpose:  Prints the given program to the output channel
 *
 * Inputs:   Nothing
 *
 * Returns:  The individual number
 *
 ****************************************************************************/
macro proc printInstruction(C_UINT word)
{
  printf("0x%x ", word);
  dooutdata(FUNC,adju(GetOpcode(word),8));
  dooutdata(EA1,adju(GetEa1(word),8));
  dooutdata(EA2,adju(GetEa2(word),8));
  donl();
}

macro proc printProg(C_UINT idx)
{
  WorkIndex w;
  Index i;
  Word word;
  PopIndex prog;
  Index_1  len;
  Index_1 count;  /* We rely on the use of a carry into bit
             IDXW+1 to detect the end of the loop */

  PAR {
    w = 0;
    i = 0;
  }
  readIn(idx,w);

  prog = WorkingprogNumRead(phBreed,w);
  dooutdata(IND, adju(prog,8));
  donl();
  word = adju(WorkingFitnessRead(phBreed,w), WORDW);
  dooutdata(FIT, adju(word,8));
```

```
  donl();
  len = adju(WorkingLenRead(phBreed,w), IDXW+1);
  dooutdata(LEN, adju(len,8));
  donl();
  count = MAXNODES - len;

  while(!BITSET(count, MAXNODES)) {  /* Guard against zero length programs */
    WorkingPopRead(phBreed,w,i, C_ADDR word);
    i = i + 1;
    printInstruction(word);
    count++;
  }
#if defined ANT && !defined HANDELC && !defined PSIM
  {
    FILE    *df;

    if((df=fopen("trail.dat", "w")) == NULL ) {
      perror("Creating ant trail file");
      df = stdout;
    }
    for(i=0;i<len;i++) {
      WorkingPopRead(phBreed,w,i,C_ADDR word);
      printf("%d %d %d\n", GetOpcode(word), GetEa1(word), GetEa2(word));
      fprintf(df, "%d %d %d\n", GetOpcode(word), GetEa1(word), GetEa2(word));
    }
    if(df) fclose(df);
  }
#endif
}

macro proc printStats()
{
  dooutdata(NMUTATE, nmutate);
  dooutdata(NXOVER, nxover);
  dooutdata(NCOPY, ncopy);
}

macro proc printGen(C_UINT gen)
{
  dooutdata(GEN, gen);
}


void dumpResults()
{
    MainPopIndex   page0;
    MainPopIndex   page1;
    MainPopIndex   page2;
    MainPopIndex   addr;
    UINT32         val;
    PopIndex_1     idx;
    Word           fit, len;
    Bool           done;

    PP1000RequestMemoryBank(2);
    PAR {
        idx      = 0;
        page0    = 0;
        page1    = POPSIZE;
        page2    = POPSIZE*2;
    }
    addr = page0 | 1;
    PP1000WriteBank1func(addr, GENSIZE);
    addr = page0 | 2;
    PP1000WriteBank1func(addr, GENERATIONS);
    addr = page0 | 3;
    PP1000WriteBank1func(addr, MAXPAR);
    addr = page0 | 4;
    PP1000WriteBank1func(addr, MAXPROGLEN);
    addr = page0 | 5;
    PP1000WriteBank1func(addr, PROBLEMTYPE);
    addr = page0 | 6;
    PP1000WriteBank1func(addr, cycles);
    addr = page0 | 7;
```

```
    PP1000WriteBank1func(addr, POPSIZE);
    do {
        PAR {
            fit = populationFitness.ReadWriteA[adju(idx,POPW)];
            len = populationLen.Read[adju(idx,POPW)];
        }
        PAR {
            addr = page1 | adju(idx, SRAMW);
            val  = adju(fit,32);
        }
        PP1000WriteBank1func(addr, val);
        PAR {
            addr = page2 | adju(idx, SRAMW);
            val  = adju(len,32);
        }
        PP1000WriteBank1func(addr, val);
        done = (idx == POPSIZE-1);
        idx++;
    }while(!done);
    PP1000ReleaseMemoryBank(2);
}

void printLengths(Generation generation)
{
#if !defined HANDELC
  if(dumpLengths && generation % (POPSIZE/(MAXPAR)) == 0) {
    int i;

    for(i=0;i<POPSIZE;i++) {
      printf("%d ", populationLen.Read[i]);
    }
    printf("\n");
  }
#endif
}

/****************************************************************************
 * Name:     main
 *
 * Purpose:  The main function
 *
 * Inputs:   nothing
 *
 * Returns:  nothing
 *
 * Note:
 *
 ****************************************************************************/
MAINTYPE main MAINARGS
{
  /* Control variables to drive pipeline */
  Bool          doEval;
  Bool          selectionDone;
  Generation_w  generation;
  PopIndex      best;
#if defined HANDELC
  UINT8         ctrl;
#endif
  Bool evalDone;

  evalDone = 0;

#if !defined HANDELC && !defined PSIM
  if(argc >1) {
    int a = 1;
    do {
      if(strcmp(argv[a], "-s") == 0) {
        newseed = 1;
      }
      else if(strcmp(argv[a], "-p") == 0) {
    dumpProg = 1;
      }
      else if(strcmp(argv[a], "-f") == 0) {
    dumpFittest = 1;
```

```
      }
      else if(strcmp(argv[a], "-l") == 0) {
    dumpLengths = 1;
      }
      else if(strcmp(argv[a], "-v") == 0) {
    verbose = 1;
      }
      else if(strcmp(argv[a], "-t") == 0) {
    dumpFitlen = 1;
      } else {
    printf("Usage: %s [-s] [-p] [-f]\n", argv[0]);
    printf("\t-s set the random number seed from the clock\n");
    printf("\t-p dump the best program at the end\n");
    printf("\t-f dump the value of the fittest program once per generation\n");
    printf("\t-l dump the program lengths for each generation\n");
    printf("\t-v print verbose debug information\n");
    printf("\t-t print the lengths of programs that are 100%% fit\n");
    exit(1);
      }
      a++;
    } while(a < argc);
  }
  stopon=1;
#endif

  /* Wait for the host to tell us to start */
  PP1000ReadControl(ctrl);

  /* Initialise variables etc */
  PAR {
    phBreed     = 0;
    phEval      = 1;    /* Delayed by one pass so starts at 1 */
    doEval      = 0;
    generation  = 0;
    cycles      = 0;
    randseed();
  }

  PP1000RequestMemoryBank(1);
  PP1000WriteStatus(1);
  running = 1;
  PAR {

    /* Count cycles */
    CycleCount();

    /* Run the random number generator */
    RandomGen();


    SEQ {
      /* Build the initial population */
      initPop();   /* Uses phase 0 */
      printLengths(0);

      /*
       * Run the main pipeline. This will process MAXPAR individuals per pass
       * A generation is done when POPSIZE/MAXPAR individuals have been done
       * so the total number of passes is GENSIZE = GENERATONS * (POPSIZE/MAXPAR)
       */
        do {
             selectionDone = 0;
             PAR {
                    /* The selection block */
           SEQ {
             phWriteback = phBreed;   /* Write back this phase before we overwrite it */
             writeBack();
             selection();
             selectionDone = 1;
             breed();
             doEval=1;
           }
```

```
                  /* The eval block (only starts when we have done the 1st breed) */
                SEQ {
                    if(doEval) {
#if defined NOFITNESS
#warning No fitness selected
#else
              evaluate();
              evalDone=1;

#endif
              /*
               * Now wait until selection() has finished with the
               * global fitness and population and perform the replacement
               */
                        WAIT(selectionDone);
                        replacement();
                    } else delay;
                } /* SEQ for eval block */
            } /* PAR for main loop */
#if !defined HANDELC
         if(dumpFittest && generation % (POPSIZE/MAXPAR) == 0) {
           findBest(C_ADDR best);
           printf("%d\n", populationFitness.ReadWriteA[best]);
         }
         printLengths(generation);
#endif
            PAR {
            generation++;
                TOGGLE(phBreed);
                TOGGLE(phEval);
            }
        //          printGen(adju(generation,8));
      } while(!BITSET(generation, GENSIZE));
      phWriteback = phBreed;
      writeBack();    /* Write the last evaluated individuals */

      /* Tell the cycle counter to stop */
      running = 0;
#if !defined HANDELC
      if(dumpProg) {
    findBest(C_ADDR best);
    printProg(best);
      }
      if(dumpFitlen) {
    int f;
    int l;

    findBest(C_ADDR best);
    readIn(best,0);
    if((f=WorkingFitnessRead(phBreed,0)) == 0) {
      l = WorkingLenRead(phBreed,0);
      printf("%d\n", l);
    }
      }
#endif
      PP1000ReleaseMemoryBank(1);

      /* Write the lengths and fitnesses to the sram */
      dumpResults();

      /* Release all banks and signal that we have finished */
      PP1000WriteStatus(good);

      /*
       * This marks the end of the GP algorithm.
       * The machine will still be running the random number
       * generator and the cycle counter(s) however until
       * the machine is reset
       */
    } /* SEQ */
  } /* par */
  MAINRET;
}
```

```
/**********************************************************************
 * Name:     makeProg
 *
 * Purpose:  Make an individual program.
 *
 * Inputs:   The phase to write to (1st index into block select ram )
 *           The individual number within the phase (2nd index into ram )
 *
 * Returns:  Nothing
 *
 * Notes:    We generate the maximal number of instructions regardless
 *           of the actual length to simplify the logic. It is not an expensive
 *           operation in the general scheme of things.
 *
 **********************************************************************/
macro proc makeProg()
{
    PC         proglen;
    Proglen_1 icount;
    Index      i;
    Phase      zero;
    static PopIndex progNum = 0;

#if defined PRESET
#if defined ANT
    zero = 0;
    WorkingLen(zero,0) = 12;
    WorkingFitness(zero,0) = DEFAULT_FITNESS;
    WorkingprogNum(zero,0) = progNum;
    populationControl[progNum]= 1;
    progNum++;
    WorkingPopWrite(zero,0,0,0x07);
    WorkingPopWrite(zero,0,1,0x1c);
    WorkingPopWrite(zero,0,2,0x11);
    WorkingPopWrite(zero,0,3,0x07);
    WorkingPopWrite(zero,0,4,0x14);
    WorkingPopWrite(zero,0,5,0x12);
    WorkingPopWrite(zero,0,6,0x0b);
    WorkingPopWrite(zero,0,7,0xc);
    WorkingPopWrite(zero,0,8,0xc);
    WorkingPopWrite(zero,0,9,0xe);
    WorkingPopWrite(zero,0,10,0x19);
    WorkingPopWrite(zero,0,11,0xe);
#elif defined BOOLPARITY
    zero = 0;
    WorkingLenWrite(zero,0) = 8;
    WorkingFitnessWrite(zero,0) = DEFAULT_FITNESS;
    WorkingprogNumWrite(zero,0) = progNum;
    populationControl.ReadWriteA[progNum]= 1;
    progNum++;
    WorkingPopWrite(zero,0,0,0x33);
    WorkingPopWrite(zero,0,1,0x2f);
    WorkingPopWrite(zero,0,2,0x2f);
    WorkingPopWrite(zero,0,3,0x2f);
    WorkingPopWrite(zero,0,4,0x0);
    WorkingPopWrite(zero,0,5,0x2f);
    WorkingPopWrite(zero,0,6,0x18);
    WorkingPopWrite(zero,0,7,0x18);
#else
#error No preset data for this problem type
#endif
#else

    /*
    * Determine initial program length. We ensure that this is greater than 0
    */
    PAR {
      proglen = randLen();
      zero = 0;
    }
    /*
     * Initialise the program in block select memory
     * Dont try to optimise the number of nodes to be the length
```

```
     * as this will add extra logic.
     */
    PAR {
        WorkingLenWrite(zero,0)        = adju(proglen, WORDW);
        WorkingFitnessWrite(zero,0)    = DEFAULT_FITNESS;
    WorkingprogNumWrite(zero,0)    = progNum;
        i = 0;
    icount = 0;
        populationControl.ReadWriteB[progNum]= 1;  /* Signal that this should
                                           get written */
    progNum++;
    }

    do  {
        WorkingPopWrite(zero, 0, i, genNode());
        PAR {
            i++;
            icount++;
        }
    } while(!BITSET(icount, MAXPROGLEN));
#endif
}



/***************************************************************************
 * Name:     writeBackInd
 *
 * Purpose:  Writes an individual back to external SRAM from on-chip
 *           block select ram.
 *
 * Inputs:   The phase to use (1st index into ram arrays)
 *           The individual number (2nd index into ram arrays )
 *
 * Returns:  Nothing
 *
 * Note:     The address to write to is calculated by taking the
 *           individual number and multiplying by the program size.
 *           The instructions are packed into WORDLEN bit words to make memory
 *           access more efficient.
 *           The individual is not written back if it does not have it's
 *           control bit set.
 ***************************************************************************/
void writeBackInd(Phase phase, WorkIndex ind)
{
    MainPopIndex address, addr;
    Index     j,k;
    PopIndex  num;
    Word f,n, word;
    UINT32  val;
    Bool   done, done2;

    num = adju(WorkingprogNumRead(phase,ind), POPW);

    if(!populationControl.ReadWriteA[num]) {
        delay;
    } else {
        PAR {
            indexToAddr(num, address);
            j = 0;
            k = 0;
            done = 0;
        }
            SEQ {
                populationControl.ReadWriteA[num] =0;
        f = WorkingFitnessRead(phase, ind);
        n = WorkingLenRead(phase, ind);
                populationFitness.ReadWriteA[num] = f;
                populationLen.ReadWriteB[num] = n;
            }


            do  {
                SEQ {
```

```
                    word = WorkingPopGet(phase,ind,j);
                    val  = adju(word,BIT32);
                    addr = address | adju(j, SRAMW);
//                  delay;
#if defined HANDELC
                    PP1000WriteBank0func(addr , 0@word);
#else
                    PP1000WriteBank0func(addr , word);
#endif
                    PAR {
                        done2 = (k == MAXNODES-1);
                        j++;
                        k++;
                    }
                }
            } while(!done2);
    }
}

/**************************************************************************
 * Name:     readIn
 *
 * Purpose:  Reads an individual from external SRAM
 *
 * Inputs:   i   = The individual number in the main population
 *           ind = The individual in working ram
 *
 * Returns:  Nothing
 *
 * Note:     The address to write to is calculated by taking the
 *           individual number and multiplying by the program size
 *           The individual is marked as being in workingPop
 **************************************************************************/
void readIn(PopIndex i, WorkIndex ind)
{
    MainPopIndex  address;
    MainPopIndex  addr2;
    UINT32        val;
    Index_1       j;
    Word          f,n;
    Phase         ph;
    Bool          done;

    PAR {
        indexToAddr(i, address);
        j = 0;
        ph = phBreed;
        done = 0;
    }
        SEQ {
            populationControl.ReadWriteA[i]=1;
            f = populationFitness.ReadWriteB[i];
            n = populationLen.Read[i];
            WorkingFitnessWrite(ph, ind) = f;
            WorkingLenWrite(ph, ind) = n;
            WorkingprogNumWrite(ph, ind) = adju(i, POPW);
        }

        do {
            addr2 = address|adju(j,SRAMW);
                PP1000ReadBank0(val, addr2);
                WorkingPopWrite(ph,ind,adju(j, IDXW), adju(val, WORDW));
                PAR {
                    j++;
                    done = (j == MAXNODES-1);
                }
        } while(!done);
}

/**************************************************************************
 * Name:     initPop
 *
 * Purpose:  Initialise the population
 *
```

```
 * Inputs:   Nothing
 *
 * Returns:  Nothing
 *
 * Note:     Side effect is to construct the programs in external SRAM
 *           The programs are built in the block ram first then copied to the
 *           external SRAM.
 *           The fitness of the built programs is evaluated as part of this
 *           before they are written back so that we start with a full set
 *           of fitness cases
 *           Two of the block select rams are used for the different phases
 *           which ensures that we don't break the rule about reading/writing
 *           to a block select ram more than once per clock cycle.
 ***************************************************************************/
macro proc initPop()
{
  PopIndex_1 idx;

  idx      = 0;

  do {
      makeProg();
      PAR {
        writeBackInd(0, 0);
        idx++;
      }
  } while(!BITSET(idx, POPSIZE));
}

/***************************************************************************
 * Name:     writeBack
 *
 * Purpose:  Writes all individuals in working ram pointed to by phWriteback
 *           out to external RAM, and clears the inwork bit for all
 *           individuals still in workingPop.
 *
 * Inputs:   Nothing
 *
 * Returns:  Nothing
 *
 ***************************************************************************/
macro proc writeBack()
{
    WorkIndex w;

    w = 0;
    do {
        writeBackInd(phWriteback, w);
        w++;
    } while(LIMIT(w,MAXPAR));
}


#if !defined HANDELC
/***************************************************************************
 * iso-c stubs for PP1000 and macro expression/procs
 **************************************************************************/
char * stoi(int v)
{
  static char b[10];
  sprintf(b, "%d", v);
  return b;
}

Word sram[4][POPSIZE*MAXNODES]; /* 4 banks */
Word READRAM(unsigned bank, unsigned long addr)
{
  return sram[bank][addr];
}

void WRITERAM(unsigned bank, unsigned addr, unsigned val)
{
  sram[bank][addr] = val;
}
```

```c
/*
 * Random number routines for a regular ANSI-C system
 */
FILE *randlog;

RandReg randReg;    /* The register used for generating the number */
RandReg   s_random_lfsr(void)
{
 unsigned int BITW bit0;
 RandReg  t1, t2, t3, t4, t5, t6;
 t1 = !!(randReg & (1LL<<TAP1));
 t2 = !!(randReg & (1LL<<TAP2));
 t3 = !!(randReg & (1LL<<TAP3));
 t4 = !!(randReg & (1LL<<TAP4));
 t5 = !!(randReg & (1LL<<TAP5));
 t6 = !!(randReg & (1LL<<TAP6));
 bit0 = t1^t2^t3^t4^t5^t6;
 randReg <<= 1;
 randReg |= bit0;
 nrand++;
 return randReg;
}

#ifdef LFSR_16
RandReg s_random(void)
{
  int i;

  for(i=0;i<15;i++) {
    s_random_lfsr();
  }
  return s_random_lfsr();
}
#elif defined CA_16
typedef  unsigned long Rbits;

#define  BMAX ((sizeof(Rbits)*8)-1)
#define  BIT_P1(b) (b==BMAX?0:(b+1))
#define  BIT_M1(b) (b==0?BMAX:(b-1))
#define  BIT(v,n)  (!!(v&(1<<n)))
#define  SBIT(v,n,b) (b ? (v|=(1<<n)):(v&=(~(1<<n))))
#define CABIT(ca,bit) ((BIT(ca,BIT_P1(bit))|BIT(ca,bit))^BIT(ca,BIT_M1(bit)))

RandReg s_random(void)
{
  static Rbits v1 = 0x8000;
  Rbits  v2;
  Rbits  r;
  int i;


  v2 = 0;

  for(i=0;i<=BMAX;i++) {
    int m;
    int e,w,x;
    e=BIT(v1,BIT_M1(i));
    w=BIT(v1,BIT_P1(i));
    x=BIT(v1,i);
    m = CABIT(v1,i);
    SBIT(v2,i,CABIT(v1,i));
  }
  r=v1;
  v1=v2;
  return r;
}
#elif defined CONGRNG
RandReg s_random()
{
  return rand();
}
#elif defined SEQRNG
RandReg s_random()
```

```
{
  return randReg++;
}
#elif defined SPARSE
RandReg s_random()
{
  static r = 2;
  r+=2;
  return r & 0xfffffff0;
}
#elif defined TRUERAND
RandReg s_random()
{
  static int fd = -1;
  unsigned long v;

  if(fd==-1) {
    fd = open("10megs-random.1", O_RDONLY);
    if(fd == -1) {
      perror("Opening random number file");
      exit(1);
    }
    lseek(fd, randReg%1000000, SEEK_SET);
  }
  read(fd, &v, sizeof v);
  return v;
}
#else
RandReg s_random()
{
  return s_random_lfsr();
}
#endif
void RandomGen(void)
{
}

void CycleCount(void)
{
}

void randseed(void)
{
#if !defined PSIM
    FILE *fp;
    time_t t;
    struct timeval tv;
    struct timezone tz;

    gettimeofday(&tv, &tz);
    t = time(NULL);
    t = tv.tv_usec + tv.tv_sec;
    if(newseed) {
        fp=fopen(RANDFILE, "w");
        if(fp) {
          fprintf(fp, "%lu", t);
          randReg = t;
        } else {
          randReg=t;
        }
    } else {
        fp=fopen(RANDFILE, "r");
        if(fp) {
          fscanf(fp, "%d", &randReg);
        } else {
          randReg = time(NULL);
        }
    }
    if(fp) fclose(fp);
    if(verbose && !tree) {
      printf("SEED=%d\n", randReg);
    }
    randlog=fopen(RANDLOG, "w");
    srand(randReg);
```

```
#else
    randReg = 260158;
#endif
}

PC randPC()
{
  return s_random() % MAXPROGLEN;
}

/* Return a non-zero program length */
PC randLen()
{
  PC r = 0;
  do {
    r = randPC();
  }while(r==0);
  return r;
}

void randMethod(Method *method)
{
  static const UINT8 mask1 = 0x1f;
  static const UINT8 mask2 = 0xfc;
  UINT8 mask3;
  UINT8 mask4;
  UINT8 v, v1, v2, v3, v4;

  PAR {
    v = s_random() & 0xff;
    mask3 = ~mask1;
    mask4 = ~mask2;
  }

  PAR {
    v1 = v&mask1;
    v2 = v&mask2;
    v3 = v&mask3;
    v4 = v&mask4;
  }
  if(!v3&&v1)
    *method = MUTATE;
  else if(!v4&&v2)
    *method = COPY;
  else
    *method = XOVER;
}

PopIndex randPopIndex()
{
  return s_random() % POPSIZE;
}

/****************************************************************************
 * Name:     decodeTerm
 *
 * Purpose:  Print a term for the ant problem
 *
 * Inputs:   The term number
 *
 * Returns:  a pointer to static data describing the term
 *
 ****************************************************************************/
#if !defined HANDELC
char * decodeTerm(int t)
{
  static char * termTab[] = {
    "Left",
    "Right",
    "Move",
    "Nop",
  };

  if(t<0 || t>3)
```

```
    return "Unknown";
  else
    return termTab[t];
}
#endif

void dooutdata(int v1, int v2)
{
  char * vs1 = "";
  char * vs2 = "";
  static  char buf[100];

  switch(v1) {
  case LEN:
    vs1 = "Length";
    vs2 = stoi(v2);
    break;
  case GEN:
    vs1 = "Generation";
    vs2 = stoi(v2);
    break;
  case IND:
    vs1 = "Individual";
    vs2 = stoi(v2);
    break;
#if XOR
  case FUNC:
    vs2 = "";
    switch(v2) {
    case AND:
      vs2="AND";
      break;
    case OR:
      vs2="OR";
      break;
    case NAND:
      vs2="NAND";
      break;
    case NOR:
      vs2="NOR";
      break;
    default:
      sprintf(buf, "Unkown Opcode");
      break;
    }
    vs1 = "FUNC";
    break;
 case EA1:
   vs1 = "Ea1";
   vs2 = stoi(v2);
   break;
 case EA2:
   vs1 = "Ea2";
   vs2 = stoi(v2);
   break;
 case K:
   vs1 = "K";
   vs2 = stoi(v2);
   break;
#endif
#if BOOLPARITY
  case FUNC:
    vs2 = "";
    switch(v2) {
    case AND:
      vs2="AND";
      break;
    case OR:
      vs2="OR";
      break;
    case NOR:
      vs2="NOR";
      break;
    case NAND:
```

```
      vs2="NAND";
      break;
    default:
      sprintf(buf, "Unkown Opcode");
      break;
    }
    vs1 = "FUNC";
    break;
 case EA1:
   vs1 = "R";
   vs2 = stoi(v2);
   break;
 case EA2:
   vs1 = "R";
   vs2 = stoi(v2);
   break;
#endif
#if ANT
  case FUNC:
    vs2 = "";
    switch(v2) {
    case IF_FOOD:
      vs2 = "IF_FOOD";
      break;
    case PROGN2:
      vs2 = "PROGN";
      break;
    default:
      sprintf(buf, "Unknown opcode %d\n", v2);
      vs2=buf;
      vs1="Func";
    }
    break;
 case EA1:
   vs1 = "";
   vs2 = decodeTerm(v2);
   break;
 case EA2:
   vs1 = "";
   vs2 = decodeTerm(v2);
   break;
#endif
 case FIT:
   vs1 = "Fitness";
   vs2 = stoi(v2);
   break;
  case INIT:
    vs1 = "InitPop";
    vs2 = stoi(v2);
    break;
  case NMUTATE:
    vs1="Mutate";
    vs2=stoi(v2);
    break;
  case NXOVER:
    vs1="Crossover";
    vs2=stoi(v2);
    break;
  case NCOPY:
    vs1="Copy";
    vs2=stoi(v2);
    break;
 default:
   sprintf(buf,"Unknown op (%d)", v1);
   vs1 = buf;
   vs2 = stoi(v2);
   break;
  }
  printf("%s %s ",vs1, vs2);
}

void     WorkingPopWrite(unsigned int a, unsigned int b, unsigned int c, unsigned int v)
{
  workingPop[a*b].ReadWriteA[c]=v;
```

```
}

void     WorkingPopRead(unsigned int a, unsigned int b, unsigned int c, unsigned int *  v)
{
  *v = workingPop[a*b].ReadWriteA[c];
}

Word     WorkingPopGet(unsigned int a, unsigned int b, unsigned int c)
{
  return workingPop[a*b].ReadWriteA[c];
}

#endif

/*************************************************************************
 * fitness functions
 *************************************************************************/
#if defined XOR
/***************************************************************************
 * Name:     evalProg
 *
 * Purpose:  Runs a program for an individual
 *
 * Inputs:   The phase index
 *           the population index (individual)
 *
 ***************************************************************************/
macro proc evalProg(C_UINT indx_in)
{
  Index       pc;    /* Index into words of instructions */
  Index_1     counter;
  Register    ea1, ea2,r;
  Bool        regs[MAXREGS];
  Register    i;
  Opcode      opcode;
  Bool        res;
  Word        fit;
  Bool        done;
  Word        word;
  WorkIndex   indx;
  Fcase       curFit;
  Bool        done;
  Phase       ph;



/*
* Per-run initialisation
*/

  WorkingFitnessWrite(phEval,indx) = DEFAULT_FITNESS;
  curFit = 0;

    do {
  /*
   * Per-fitness case initialisation.
   * All done in 1 cycle for Handelc
   * a) Zero the register set
   * b) copy the input parameters
   * c) Set up control variables
   */
  PAR {
    i=0;
    FORPAR (r=2; r < MAXREGS; r++ ) {
      regs[r] = 0;
    }
    regs[0] = BIT0(curFit);
    regs[1] = BIT1(curFit);
    res     = BIT0(curFit) ^ BIT1(curFit);
    pc      = 1;
    counter = MAXNODES - adju(WorkingLenRead(phEval,indx),IDXW_1);
    fit     = WorkingFitnessRead(phEval,indx);
    done    = 0;
    word    = WorkingPopGet(phEval, indx, 0);
```

```
    indx     = indx_in;
  }

    do {
      /* Decode the instruction and maintain the counters */
      PAR {
        counter++;
        ea1 = GetEa1(word);
        ea2 = GetEa2(word);
        opcode = GetOpcode(word);
        word = WorkingPopGet(phEval, indx, pc);
        pc++;
        done = (counter == MAXNODES-1);
      }

      switch(opcode) {
      case AND:
        regs[ea1] &= (regs[ea2]);
        break;
      case OR:
        regs[ea1] |= regs[ea2];
        break;
      case NAND:
        regs[ea1]  = !(regs[ea1] & regs[ea2]);
        break;
      case NOR:
        regs[ea1] = !(regs[ea1] | regs[ea2]);
        break;
      }  /* switch */
    } while(!done);
    /*
     * Calculate the new raw fitness
     */
    if(regs[r0] == res ) {
      fit--;
      WorkingFitnessWrite(phEval,indx) = fit;
    } else {
      delay;
    }

    PAR {
        done = (curFit == MAXFITNESS-1);
        curFit++;
    }
    } while(!done);
}
#endif


#if defined ANT
/**************************************************************************
 * Name:     evalProg
 *
 * Purpose:  Runs a program for an individual
 *
 * Inputs:   The population index (individual)
 *
 * Returns:  Nothing
 *
 **************************************************************************/
macro proc evalProg(C_UINT indx)
{
  Index       pc;
  Index       len;
  Index_1     counter;
  Opcode      opcode;
  Direction   dir;
  Pos         x;
  Pos         y;
  Pos         ax, ay;
  Food        food;
  Food        uneaten;
  Time        timeval;
  Register    ea1, ea2;
```

```
  Register    eas[2];
  Bool        flags[2];
  Bool        e;
  Bool        foodHere;
  Bool        timeleft;
  Bool        foodleft;
  Word        word;
  Phase       ph;


  /*
   * Per run initialisation
   */
  initMap(indx);

  /*
   * Initialisation.
   * All done in 1 cycle for Handelc
   * a) Zero the register set
   * b) copy the input parameters
   * c) Set up control variables
   */
  PAR {
    ph         = phEval;
    food       = MAXFOOD - FOOD;      /* Start all food to collect */
    uneaten    = FOOD;
    dir        = EAST;         /* Pointing to the right */
    x          = 0;            /* At the first cell */
    y          = 0;
    timeval    = 0;
    pc         = 0;
    len        = adju(WorkingLenRead(phEval,indx),IDXW);
  }
  do {
      /* Get the 1st word and set up counter. pc is set to 1 for the next get */
      PAR {
        counter = MAXNODES - adju(len, IDXW+1);
    pc = 1;
        word = WorkingPopGet(ph,indx,0);
      }
    do {
      /* Decode the instruction */
      PAR {
    ea1 = GetEa1(word);
    ea2 = GetEa2(word);
    opcode = GetOpcode(word);
    ax=x;
    ay=y;
    word = WorkingPopGet(ph,indx,pc);
    pc++;
    counter++;
      }

      switch(opcode) {
      case IF_FOOD:
    switch(dir) {
    case EAST:  ax++; break;
    case WEST:  ax--; break;
    case NORTH: ay--; break;
    case SOUTH: ay++; break;
    default: delay;   break;
    } /* Switch  dir */
    /* Now check the cell 'ahead' */
    if(MAP(map, indx, adju((ax&31), 6), ay&31)) {
      PAR {
        eas[0] = ea1;
        flags[0]=1;
        flags[1]=0;
      }
    } else {
      PAR {
        eas[0] = ea2;
        flags[0]=1;
        flags[1]=0;
```

```
      }
    }
    break;  /*if_food */
      case PROGN2:
    PAR {
      eas[0]=ea1;
      eas[1]=ea2;
      flags[0]=1;
      flags[1]=1;
    }
    break;
      }  /* switch */
      /* Now execute the DOTERM proc as many times as needed */
      e = 0;
      do {
    if(flags[e]) {
      PAR {
        timeval = (timeval==MAXTIME) ? timeval : timeval+1;
        switch(eas[e]) {
        case LEFT:
          dir = (dir-1) & 3; break;
        case RIGHT:
          dir = (dir+1) & 3; break;
        case MOVE:
          switch(dir) {
          case EAST:  x++; break;
          case WEST:  x--; break;
          case SOUTH: y++; break;
          case NORTH: y--; break;
          default: delay;  break;
          }
#if !defined HANDELC
          PAR {
        y &= GRIDMASK;
        x &= GRIDMASK;
          }
#endif
          foodHere = MAP(map, indx, adju(x,6),y);
          if(foodHere && !BITSET(food, MAXFOOD)) {
        PAR {
          food++;
          uneaten--;
          CLRBIT(map, indx, adju(x,6), y);
        }
          }
          break;
        default: delay; break;
        } /* switch eas */
      } /* PAR */
    } else {
      delay;
    }
    TOGGLE(e);
      }while(e);
    } while(!BITSET(counter, MAXNODES));
    PAR {
      foodleft  = !(food==MAXFOOD);
      timeleft  = !(timeval==MAXTIME);
    }
  } while(timeleft && foodleft);
  /*
   * Calculate the new raw fitness
   */

  WorkingFitnessWrite(ph, indx) = uneaten;
#if !defined HANDELC
  if(dumpProg && uneaten == 0) {
      int i;
      printf("100%% prognum %d\n", workingprogNum[phEval].ReadWriteA[indx]);
      printf("-------------------------\n");
      printf("len=%d\n", len);
      for(i=0;i<len;i++) {
       WorkingPopRead(phEval, indx, i, C_ADDR word);
       printf("%d %d %d\n", GetOpcode(word), GetEa1(word), GetEa2(word));
```

```
      }
      printf("--------------------------\n");
  }
#endif
}
#endif  /* ANT */


#if defined BOOL11MUX
/****************************************************************************
 * Name:     evalProg
 *
 * Purpose:  Runs a program for an individual
 *
 * Inputs:   The phase index
 *           the population index (individual)
 *
 * Returns:  1 if we found 100% fit program
 *           0 otherwise
 *
 ****************************************************************************/
macro proc evalProg(C_UINT indx)
{
  Index       pc;   /* Index into words of instructions */
  Index       len;
  Index_1     counter;
  Register    ea1, ea2,r;
  Bool        regs[MAXREGS];
  Register    i;
  Opcode      opcode;
  Word        word;
  Bool        res;
  Word        fit;
  unsigned int curFit;
  unsigned r2;

  fit = DEFAULT_FITNESS;
  curFit = 0;
  do {
    /*
     * Initialisation.
     * All done in 1 cycle for Handel-C
     * a) Zero the register set
     * b) copy the input parameters
     * c) Set up control variables
     */
    PAR {
      i=0;
      FORPAR (r=13; r < MAXREGS; r++ ) {
    regs[r] = 0;
      }
      FORPAR(r2=1; r2 < 13; r2++ ) {
    regs[r2] = !!(curFit & (1<<r2));
      }
      regs[0]=0;
      res = !!((curFit>>3) & (1<<(curFit & 7)));
           //      res     = !!(curFit[10:3]&(1<<curFit[2:0]));
      /*      printf("Cur = 0x%x data = 0x%x Address = 0x%x result = 0x%x\n", curFit,
          curFit>>3, curFit & 7, res);*/
      pc      = 0;
      counter = MAXNODES - adju(WorkingLenRead(phEval,indx),IDXW_1);

    }

    do {
      /* Decode the instruction and maintain the counters */
      PAR {
        counter++;
    ea1 = GetEa1(WorkingPopGet(phEval,indx,pc));
    ea2 = GetEa2(WorkingPopGet(phEval,indx,pc));
    opcode = GetOpcode(WorkingPopGet(phEval,indx,pc));
        pc++;
      }
```

```
      switch(opcode) {
      case AND:
    regs[ea1] &= (regs[ea2]);
    break;
      case OR:
    regs[ea1] |= regs[ea2];
    break;
      case NOT:
        regs[ea1]   = !(regs[ea1]);
    break;
      case IF:
    if(regs[ea1]) {
      PAR {
        pc++;
        counter++;
      }
    }else{
      delay;
    }
    break;
      }  /* switch */
    } while(!BITSET(counter, MAXNODES));
    /*
     * Calculate the new raw fitness
     */
    if(regs[r0] == res ) {
      fit--;
    } else {
      delay;
    }
    curFit++;
  } while(curFit != MAXFITNESS);
  WorkingFitnessWrite(phEval,indx) = fit;
}
#endif

#if defined BOOLPARITY
/***************************************************************************
 * Name:     evalProg
 *
 * Purpose:  Runs a program for an individual
 *
 * Inputs:   The phase index
 *           the population index (individual)
 *
 * Returns:  1 if we found 100% fit program
 *           0 otherwise
 *
 ***************************************************************************/
macro proc evalProg(C_UINT indx)
{
  Index       pc;   /* Index into words of instructions */
  Index_1     counter;
  Register    ea1, ea2,r;
  Bool        regs[MAXREGS];
  Register    i;
  Opcode      opcode;
  Bool        res;
  Bool        done;
  Bool        doneFit;
  Word        fit;
  Word curFit;
  Word tmpFit;
  unsigned r2;
  Word bits;
  Word  b;
  Phase       ph;

  PAR {
    fit = DEFAULT_FITNESS;
    ph  = phEval;
    curFit = 0;
  }
  do {
```

```
    /*
     * Initialisation.
     * All done in 1 cycle for Handel-C
     * a) Zero the register set
     * b) copy the input parameters
     * c) Set up control variables
     */
    PAR {
      i=0;
      FORPAR (r=PARITYBITS; r < MAXREGS; r++ ) {
    regs[r] = 0;
      }
      FORPAR(r2=0; r2 < PARITYBITS; r2++ ) {
    regs[r2] = !!(curFit & (1<<r2));
      }
#if 0
      for(i=0;i<MAXREGS;i++) {
    printf("curFit = 0x%x, reg %d = %d\n", curFit, i, regs[i]);
      }
#endif
      bits = 0;
      tmpFit = curFit;
      b=0;
      pc      = 0;
      counter = MAXNODES - adju(WorkingLenRead(ph,indx),IDXW_1);
    }  /* PAR */


/* Do the calculation of the parity in parallel with the fitness eval */
   PAR {

    do {
        if(tmpFit & 0x1) {
            bits++;
        } else {
            delay;
        }
        PAR {
            tmpFit >>= 1;
            done = (b==PARITYBITS-1);
            b++;
#if defined HANDELC
            res = bits[0];
#else
            res = bits & 1;
#endif
      }
    }while(!done);



    do {
      /* Decode the instruction and maintain the counters */
      /*PAR*/ {
        counter++;
    ea1 = GetEa1(WorkingPopGet(ph,indx,pc));
    ea2 = GetEa2(WorkingPopGet(ph,indx,pc));
    opcode = GetOpcode(WorkingPopGet(ph,indx,pc));
    //      printf("PC %d: I=0x%x, op = 0x%x, ea1 = 0x%x, ea2 = 0x%x\n", pc, WorkingPopGet
    (ph,indx,pc), opcode, ea1, ea2);
        pc++;
      }

      switch(opcode) {
      case AND:
    regs[ea1] &= (regs[ea2]);
    break;
      case OR:
    regs[ea1] |= regs[ea2];
    break;
#if defined NOXOR
#warning Using and,or,nand, nor
      case NAND:
    regs[ea1]  = !(regs[ea1]& regs[ea2]);
```

```
    break;
      case NOR:
    regs[ea1] = !(regs[ea1] | regs[ea2]);
    break;
#else
#warning Using and,or,not, xor
      case NAND:
    regs[ea1]  = !regs[ea2];
    break;
      case NOR:
    regs[ea1] = (regs[ea1] ^ regs[ea2]);
    break;
#endif
      }  /* switch */
    } while(!BITSET(counter, MAXNODES));
    /*
     * Calculate the new raw fitness
     */
   } /* PAR */
  if(regs[r0] == res ) {
    fit--;
  } else {
    delay;
  }
  curFit++;
  doneFit = (curFit == MAXFITNESS);
 } while(!doneFit);

  WorkingFitnessWrite(ph,indx) = fit;
}
#endif

#if defined PSIM
int _start()
{
  static char *argv[] = {"lgpc",NULL};
  return main(1, argv);
}

int printf(const char * fmt, ...)
{
  return 0;
}

int fprintf(FILE *f, const char * fmt, ...)
{
  return 0;
}

int sprintf(char *s, const char * fmt, ...)
{
  return 0;
}

int rand(void)
{
  int v;
  return v++;
}
#endif
#else

ram unsigned int 32 array[32];
void main(void)
{
    unsigned int 5 count;
    unsigned int 16 loop;
    count = 0;
    loop  = 0;
    PP1000RequestMemoryBank(1);

  /* Initialise */
    do {
        array[count]=adju(count,32);
```

```
        PP1000WriteBank0func(adju(count,21), array[count]);
        count++;
    } while(count);

    do {
    /* Modify */
    count = 0;
    do{
        unsigned int 32 val;
        PP1000ReadBank0(val, adju(count,21));
        array[count] = val;
        val <<= 1;
        array[count] = val;
        PP1000WriteBank0func(adju(count,21),array[count]);
        count++;
    } while(count);
    loop++;
    } while(loop);
    PP1000ReleaseMemoryBank(1);
    PP1000WriteStatus(1);
}
#endif
```