```
void numberblks();
struct bblk *findblk(char *);
void setupcontrolflow();
void clearstatus();
void check_cf();
```

```
void classifyinst(short, itemarray, enum insttype *, int *, int *, int *);
void reclassifyinsts();
void makeinstitems(char *, short *, itemarray *, int);
void setupinstinfo(struct assemline *);
int readinfunc();
void dumpblk(FILE *, struct bblk *);
void dumpoutblks(FILE *, unsigned int, unsigned int);
void dumpblks(int, int);
void dumpfunc();
void dumpruleusage();
void dumpfunccounts();
void dumptotalcounts();
void dumpoptcounts();
```

```
void *alloc(unsigned int);
char *allocstring(char *);
void replacestring(char **, char *, char *);
void createmove(int, char *, char *, struct assemline *);
void assignlabel(struct bblk *, char *);
struct bblk *newblk(char *);
void freeblk(struct bblk *);
int inblist(struct blist *, struct bblk *);
void freeblist(struct blist *);
struct assemline *newline(char *);
void hookupline(struct bblk *, struct assemline *, struct assemline *);
void unhookline(struct assemline *);
struct assemline *insline(struct bblk *, struct assemline *, char *);
void delline(struct assemline *);
void freeline(struct assemline *);
void addtoblist(struct blist **, struct bblk *);
void sortblist(struct blist *);
void orderpreds();
void deleteblk(struct bblk *);
void unlinkblk(struct bblk *);
void delfrompreds_succs(struct bblk *);
void delfromsuccs_preds(struct bblk *);
struct bblk *delfromblist(struct blist **, struct bblk *);
struct loopnode *newloop();
void freeloops();
void dumploops(FILE *);
void dumploop(FILE *, struct loopnode *);
void incropt(enum opttype);
void quit(int);
```

```
// main include file for the opt program

#define NUMVARWORDS   3      /* number of words required to represent the
                                registers and variables on the SPARC     */
#define MAXFIELD     30      /* maximum characters in a field            */
#define MAXLINE      81      /* maximum characters in an assembly line   */
#define LOG2_INT      5      /* sizeof(int) = 32 = 2**5                  */
#define INT_REM      31      /* if (v & INT_REM) v is not an integer
                                multiple of sizeof(int)*4                */
#define ARGSIZE      20      /* maximum argument size for peephole rule  */
#define MAXRULES    100      /* maximum number of peephole rules         */

#define FALSE         0
#define TRUE          1

/* block status bits */
#define DONE          (1 << 0)

/* used to initialize bvect's */
#define binit() ((bvect) NULL)

/* used to allocate space for a basic block vector */
#define BALLOC  ((unsigned int *) malloc(sizeof(unsigned int)*bvectlen))

/*
 * block membership bit vector
 */
typedef unsigned int *bvect;

/*
 * variable state structure
 */
typedef unsigned int varstate[NUMVARWORDS];

/*
 * item array type
 */
typedef char **itemarray;

/* instruction types */
enum insttype {
   ARITH_INST,
   BRANCH_INST,
   CALL_INST,
   CMP_INST,
   CONV_INST,
   JUMP_INST,
   LOAD_INST,
   MOV_INST,
   RESTORE_INST,
   RETURN_INST,
   SAVE_INST,
   STORE_INST,
   COMMENT_LINE,
   DEFINE_LINE
};

/* optimization types */
enum opttype {
   REVERSE_BRANCHES,
   BRANCH_CHAINING,
   DEAD_ASG_ELIM,
   LOCAL_CSE,
   FILL_DELAY_SLOTS,
   CODE_MOTION,
   COPY_PROPAGATION,
   PEEPHOLE_OPT,
   REGISTER_ALLOCATION,
   UNREACHABLE_CODE_ELIM
```



```
};

#define TOC(t)  (t == BRANCH_INST || t == CALL_INST || t == JUMP_INST || \
                 t == RETURN_INST)

#define INST(t) (t != COMMENT_LINE && t != DEFINE_LINE)

/*
 * assembly line structure
 */
struct assemline {
   char *text;                  /* text of the assembly line               */
   struct assemline *next;      /* next assembly line                      */
   struct assemline *prev;      /* previous assembly line                  */
   enum insttype type;          /* type of assembly line                   */
   int instinfonum;             /* index into instruction information      */
   short numitems;              /* number of strings for instruction       */
   itemarray items;             /* list of strings for the line tokens     */
   varstate sets;               /* variables updated by the instruction    */
   varstate uses;               /* variables used by the instruction       */
   varstate deads;              /* variable values that are last used by
                                   the instruction                         */
   struct bblk *blk;            /* block containing the assembly line      */
};

/*
 * basic block structure
 */
struct bblk {
   char *label;                 /* label of basic block                     */
   unsigned short num;          /* basic block number                       */
   short loopnest;              /* loop nesting level                       */
   struct assemline *lines;     /* first line in a basic block              */
   struct assemline *lineend;   /* last line in a basic block               */
   struct blist *preds;         /* list of predecessors for a basic block   */
   struct blist *succs;         /* List of successors for a basic block.
                                   The first block in the list is the
                                   fall-through successor when the block
                                   ends with a conditional branch.          */
   struct bblk *up;             /* positionally previous basic block        */
   struct bblk *down;           /* positionally following basic block       */
   bvect dom;                   /* blocks that dominate this one            */
   struct loopnode *loop;       /* set if this block is a loop header       */
   varstate uses;               /* variables used before being set          */
   varstate defs;               /* variables set in this block before used  */
   varstate ins;                /* variables live entering the block        */
   varstate outs;               /* variables live leaving the block         */
   unsigned short status;       /* status field for the block               */
};

/*
 * basic block list structure
 */
struct blist {
   struct bblk *ptr;            /* pointer to block within the list         */
   struct blist *next;          /* pointer to the next blist element        */
   };

/*
 * loop information structure
 */
struct loopnode {
   struct loopnode *next;       /* pointer to next loop record              */
   struct bblk *header;         /* pointer to head block of loop            */
   struct bblk *preheader;      /* pointer to the preheader of the loop     */
   struct blist *blocks;        /* blocks in the loop                       */
   varstate invregs;            /* loop invariant variables                 */
   varstate sets;               /* variables updated in loop                */
   int anywrites;               /* any writes to memory?                    */
```

```
};

/*
 * instruction information
 */
struct instinfo {
   char *mneumonic;             /* mneumonic of instruction               */
   enum insttype type;          /* instruction class                      */
   int numargs;                 /* number of arguments                    */
   int numdstregs;              /* number of consecutive registers
                                   associated with the destination        */
   int numsrcregs;              /* number of consecutive registers
                                   associated with each source            */
   int setscc;                  /* condition codes set?                   */
   int datatype;                /* datatype of instruction                */
};

/*
 * variable information
 */
struct varinfo {
   char *name;                  /* variable name                          */
   short type;                  /* variable type                          */
   short indirect;              /* variable indirectly referenced?        */
};

/*
 * optimization information
 */
struct optinfo {
   int count;                   /* transformation count for opt phase     */
   int max;                     /* max transformations for opt phase      */
   char optchar;                /* character representing opt phase       */
   char *name;                  /* name of optimization                   */
};
```

```
#define MAXREGCHAR 5   /* maximum number of characters in a register       */
#define MAXREGS 64     /* maximum number of registers                      */
#define MAXVARS 32     /* maximum number of variables in a function        */
#define MAXVARLINE 500 /* maximum number of characters shown in a varstate */

/* variable types */
#define INT_TYPE    1
#define FLOAT_TYPE  2
#define DOUBLE_TYPE 3

void varinit(varstate);
int varcmp(varstate, varstate);
int varempty(varstate);
void unionvar(varstate, varstate, varstate);
void intervar(varstate, varstate, varstate);
void minusvar(varstate, varstate, varstate);
void varcopy(varstate, varstate);
int varcommon(varstate, varstate);
void delreg(char *, varstate, int);
void delvar(varstate, int);
int calcregpos(char *);
int isreg(char *);
void insreg(char *, varstate, int);
void insvar(varstate, int);
int regexists(char *, varstate);
void setsuses(char *, enum insttype, int, itemarray, int, varstate, varstate,
              int, int);
int allocreg(short, varstate, char *);
char *varname(int);
void dumpvarstate(char *, varstate);
```

```
void binsert(bvect *, unsigned int);
void bdelete(bvect *, unsigned int);
void bunion(bvect *, bvect);
void binter(bvect *, bvect);
int bin(bvect, unsigned int);
void bcpy(bvect *, bvect);
int bequal(bvect, bvect);
bvect ball();
void bclear(bvect);
int bcnt(bvect);
bvect bnone();
void bfree(bvect);
void bdump(FILE *, bvect);
```

```
/*
 * perform control flow analysis functions
 */
#include <stdio.h>
#include <string.h>
#include "opt.h"
#include "flow.h"
#include "misc.h"

int numblks = 0;         /* number of basic blocks in the function */

/*
 * numberblks - number the basic blocks in the function
 */
void numberblks()
{
   int num;
   struct bblk *cblk;
   extern struct bblk *top;
   extern int bvectlen;

   /* number each basic block in the function */
   num = 0;
   for (cblk = top; cblk; cblk = cblk->down)
      cblk->num = ++num;
   num++;

   /* reset the length in integer words for the basic block bit vectors */
   bvectlen = num >> LOG2_INT;
   if (num & INT_REM)
      bvectlen++;

   /* set the number of blocks for the program */
   numblks = num-1;
}

/*
 * findblk - find the block with the specified label
 */
struct bblk *findblk(char *label)
{
   struct bblk *cblk;
   extern struct bblk *top;

   /* search for the block */
   for (cblk = top; cblk; cblk = cblk->down)
      if (cblk->label && strcmp(cblk->label, label) == 0)
         return cblk;

   /* could not find the block */
   return (struct bblk *) NULL;
}

/*
 * setupcontrolflow - setup the control flow within the function
 */
void setupcontrolflow()
{
   struct bblk *cblk, *tblk;
   extern struct bblk *top;

   /* for each basic block in the function */
   for (cblk = top; cblk; cblk = cblk->down) {

      /* if there are one or more instructions in the block */
      if (cblk->lineend) {

         /* if the last instruction is a jump, then establish the
            control flow between the current block and the target
```



```
            and continue with the next block */
         if (cblk->lineend->type == JUMP_INST) {
            if ((tblk = findblk(cblk->lineend->items[1]))) {
               addtoblist(&cblk->succs, tblk);
               addtoblist(&tblk->preds, cblk);
            }
            else {
               fprintf(stderr,
                       "setupcontrolflow - could not find block with label %s\n",
                        cblk->lineend->items[1]);
               quit(1);
            }
            continue;
         }

         /* if the last instruction is a branch, then establish the
            control flow between the current block and the target */
         else if (cblk->lineend->type == BRANCH_INST) {
            if ((tblk = findblk(cblk->lineend->items[1]))) {
               addtoblist(&cblk->succs, tblk);
               addtoblist(&tblk->preds, cblk);
            }
            else {
               fprintf(stderr,
                       "setupcontrolflow - could not find block with label %s\n",
                        cblk->lineend->items[1]);
               quit(1);
            }
         }

         /* if the last instruction is a return, then continue with
            the next block since a return has no successors */
         else if (cblk->lineend->prev &&
                  cblk->lineend->prev->type == RETURN_INST)
            continue;
      }

      /* establish control flow between the current block and the
         block following it since the current block could fall into it */
      addtoblist(&cblk->succs, cblk->down);
      if (cblk->down)
         addtoblist(&cblk->down->preds, cblk);
   }
   check_cf();
}

/*
 * clearstatus - clear the status field in the basic block indicating
 *               that it has not been visited yet
 */
void clearstatus()
{
    struct bblk *cblk;
    extern struct bblk *top;

    for (cblk=top; cblk; cblk=cblk->down)
        cblk->status &= ~DONE;
}

/*
 * check_cf - check the control flow for inconsistencies
 */
void check_cf()
{
   struct bblk *cblk, *tblk;
   struct blist *bptr;
   struct assemline *ptr;
   extern struct bblk *top;
```

```
   /* go through every block */
   numberblks();
   for (cblk = top; cblk; cblk = cblk->down) {

       /* check that up matches down */
       if (cblk->up && cblk->up->down != cblk) {
          fprintf(stderr, "check_cf - up does not match down at block %d\n",
                  cblk->up->num);
          quit(1);
       }

       /* check that down matches up */
       if (cblk->down && cblk->down->up != cblk) {
          fprintf(stderr, "check_cf - down does not match up at block %d\n",
                  cblk->down->num);
          quit(1);
       }

       /* check if all predecessors match */
       for (bptr = cblk->preds; bptr; bptr = bptr->next)
          if (!inblist(bptr->ptr->succs, cblk)) {
             fprintf(stderr, "check_cf - pred %d does not have %d as succ\n",
                     bptr->ptr->num, cblk->num);
             quit(1);
          }

       /* check if all successors match */
       for (bptr = cblk->succs; bptr; bptr = bptr->next)
          if (!inblist(bptr->ptr->preds, cblk)) {
             fprintf(stderr, "check_cf - succ %d does not have %d as pred\n",
                     bptr->ptr->num, cblk->num);
             quit(1);
          }

       /* check that if two successors, then the first successor is the
          fall-through successor */
       if (cblk->succs && cblk->succs->next &&
           cblk->succs->ptr != cblk->down) {
          fprintf(stderr,
                  "check_cf - first succ of branch should be fallthru\n");
          quit(1);
       }


       /* check if target of jump or branch instruction is to the
          correct block */
       if (cblk->lineend) {
          if (cblk->lineend->type == JUMP_INST &&
              !((tblk = cblk->succs->ptr) && tblk->label &&
                strcmp(cblk->lineend->items[1], tblk->label) == 0)) {
             fprintf(stderr,
                     "check_cf - target of jump in block %d incorrect\n",
                     cblk->num);
             quit(1);
          }
          else if (cblk->lineend->type == BRANCH_INST &&
                   !((tblk = cblk->succs->next->ptr) && tblk->label &&
                     strcmp(cblk->lineend->items[1], tblk->label) == 0)) {
             fprintf(stderr,
                     "check_cf - target of branch in block %d incorrect\n",
                     cblk->num);
             quit(1);
          }
       }

       /* check that assembly lines are appropriately linked */
       if (cblk->lines && cblk->lines->prev) {
          fprintf(stderr,
                  "check_cf - first line in %d should not have a prev line\n",
```



```
                  cblk->num);
          quit(1);
       }
       for (ptr = cblk->lines; ptr; ptr = ptr->next) {
          if (ptr->next && ptr->next->prev != ptr) {
             fprintf(stderr,
                     "check_cf - next does not match prev at block %d\n",
                     cblk->num);
             quit(1);
          }
          if (ptr->prev && ptr->prev->next != ptr) {
             fprintf(stderr,
                     "check_cf - prev does not match next at block %d\n",
                     cblk->num);
             quit(1);
          }
       }
       if (cblk->lineend && cblk->lineend->next) {
          fprintf(stderr, "last line in %d should not have a next line\n",
                  cblk->num);
          quit(1);
       }
   }
}
```

```
/*
 * input and output support functions
 */
#include <stdio.h>
#include <string.h>
#include "opt.h"
#include "analysis.h"
#include "misc.h"
#include "io.h"
#include "vars.h"
#include "vect.h"

#define MAXLOOPLEVELS 10

int totnuminsts;               /* total static instructions */
int totnummems;                /* total static memory references */
int totinsts[MAXLOOPLEVELS];   /* total static insts at each loop nest level */
int totmems[MAXLOOPLEVELS];    /* total static mems at each loop nest level */

/* insttypes are used to identify the type of each instruction */
struct instinfo insttypes[] = {
   {"add", ARITH_INST, 3, 1, 1, FALSE, INT_TYPE},
   {"addcc", ARITH_INST, 3, 1, 1, TRUE, INT_TYPE},
   {"and", ARITH_INST, 3, 1, 1, FALSE, INT_TYPE},
   {"andcc", ARITH_INST, 3, 1, 1, TRUE, INT_TYPE},
   {"andn", ARITH_INST, 3, 1, 1, FALSE, INT_TYPE},
   {"andncc", ARITH_INST, 3, 1, 1, TRUE, INT_TYPE},
   {"ba", JUMP_INST, 1, 0, 0, FALSE, 0},
   {"ba,a", JUMP_INST, 1, 0, 0, FALSE, 0},
   {"be", BRANCH_INST, 1, 0, 0, FALSE, INT_TYPE},
   {"bg", BRANCH_INST, 1, 0, 0, FALSE, INT_TYPE},
   {"bge", BRANCH_INST, 1, 0, 0, FALSE, INT_TYPE},
   {"bgeu", BRANCH_INST, 1, 0, 0, FALSE, INT_TYPE},
   {"bgu", BRANCH_INST, 1, 0, 0, FALSE, INT_TYPE},
   {"bl", BRANCH_INST, 1, 0, 0, FALSE, INT_TYPE},
   {"ble", BRANCH_INST, 1, 0, 0, FALSE, INT_TYPE},
   {"bleu", BRANCH_INST, 1, 0, 0, FALSE, INT_TYPE},
   {"blu", BRANCH_INST, 1, 0, 0, FALSE, INT_TYPE},
   {"bne", BRANCH_INST, 1, 0, 0, FALSE, INT_TYPE},
   {"call", CALL_INST, 2, 0, 1, FALSE, 0},
   {"cmp", CMP_INST, 2, 0, 1, TRUE, INT_TYPE},
   {"fadds", ARITH_INST, 3, 1, 1, FALSE, FLOAT_TYPE},
   {"faddd", ARITH_INST, 3, 2, 2, FALSE, DOUBLE_TYPE},
   {"fbe", BRANCH_INST, 1, 0, 0, FALSE, DOUBLE_TYPE},
   {"fbg", BRANCH_INST, 1, 0, 0, FALSE, DOUBLE_TYPE},
   {"fbge", BRANCH_INST, 1, 0, 0, FALSE, DOUBLE_TYPE},
   {"fbl", BRANCH_INST, 1, 0, 0, FALSE, DOUBLE_TYPE},
   {"fble", BRANCH_INST, 1, 0, 0, FALSE, DOUBLE_TYPE},
   {"fbne", BRANCH_INST, 1, 0, 0, FALSE, DOUBLE_TYPE},
   {"fdivs", ARITH_INST, 3, 1, 1, FALSE, FLOAT_TYPE},
   {"fdivd", ARITH_INST, 3, 2, 2, FALSE, DOUBLE_TYPE},
   {"fdtoi", CONV_INST, 2, 1, 2, FALSE, 0},
   {"fdtos", CONV_INST, 2, 1, 2, FALSE, 0},
   {"fitos", CONV_INST, 2, 1, 1, FALSE, 0},
   {"fitod", CONV_INST, 2, 2, 1, FALSE, 0},
   {"fmovs", MOV_INST, 2, 1, 1, FALSE, FLOAT_TYPE},
   {"fmovd", MOV_INST, 2, 2, 2, FALSE, DOUBLE_TYPE},
   {"fmuls", ARITH_INST, 3, 1, 1, FALSE, FLOAT_TYPE},
   {"fmuld", ARITH_INST, 3, 2, 2, FALSE, DOUBLE_TYPE},
   {"fnegs", ARITH_INST, 2, 1, 1, FALSE, FLOAT_TYPE},
   {"fstod", CONV_INST, 2, 2, 1, FALSE, 0},
   {"fstoi", CONV_INST, 2, 1, 1, FALSE, 0},
   {"fsubs", ARITH_INST, 3, 1, 1, FALSE, FLOAT_TYPE},
   {"fsubd", ARITH_INST, 3, 2, 2, FALSE, DOUBLE_TYPE},
   {"ld", LOAD_INST, 2, 1, 1, FALSE, INT_TYPE},
   {"ldd", LOAD_INST, 2, 2, 1, FALSE, DOUBLE_TYPE},
   {"ldf", LOAD_INST, 2, 1, 1, FALSE, FLOAT_TYPE},
   {"ldsb", LOAD_INST, 2, 1, 1, FALSE, INT_TYPE},
   {"ldsh", LOAD_INST, 2, 1, 1, FALSE, INT_TYPE},
```



```
   {"ldub", LOAD_INST, 2, 1, 1, FALSE, INT_TYPE},
   {"lduh", LOAD_INST, 2, 1, 1, FALSE, INT_TYPE},
   {"mov", MOV_INST, 2, 1, 1, FALSE, INT_TYPE},
   {"or", ARITH_INST, 3, 1, 1, FALSE, INT_TYPE},
   {"orcc", ARITH_INST, 3, 1, 1, TRUE, INT_TYPE},
   {"orn", ARITH_INST, 3, 1, 1, FALSE, INT_TYPE},
   {"orncc", ARITH_INST, 3, 1, 1, TRUE, INT_TYPE},
   {"restore", RESTORE_INST, 0, 0, 0, FALSE, 0},
   {"restore", RESTORE_INST, 3, 1, 1, FALSE, 0},
   {"ret", RETURN_INST, 0, 0, 0, FALSE, 0},
   {"retl", RETURN_INST, 0, 0, 0, FALSE, 0},
   {"save", SAVE_INST, 3, 1, 1, FALSE, 0},
   {"sdiv", ARITH_INST, 3, 1, 1, FALSE, INT_TYPE},
   {"sdivcc", ARITH_INST, 3, 1, 1, TRUE, INT_TYPE},
   {"sethi", ARITH_INST, 2, 1, 1, FALSE, INT_TYPE},
   {"sll", ARITH_INST, 3, 1, 1, FALSE, INT_TYPE},
   {"smul", ARITH_INST, 3, 1, 1, FALSE, INT_TYPE},
   {"smulcc", ARITH_INST, 3, 1, 1, TRUE, INT_TYPE},
   {"sra", ARITH_INST, 3, 1, 1, FALSE, INT_TYPE},
   {"srl", ARITH_INST, 3, 1, 1, FALSE, INT_TYPE},
   {"st", STORE_INST, 2, 1, 1, FALSE, INT_TYPE},
   {"stb", STORE_INST, 2, 1, 1, FALSE, INT_TYPE},
   {"std", STORE_INST, 2, 1, 2, FALSE, DOUBLE_TYPE},
   {"stf", STORE_INST, 2, 1, 1, FALSE, FLOAT_TYPE},
   {"sth", STORE_INST, 2, 1, 1, FALSE, INT_TYPE},
   {"sub", ARITH_INST, 3, 1, 1, FALSE, INT_TYPE},
   {"subcc", ARITH_INST, 3, 1, 1, TRUE, INT_TYPE},
   {"umul", ARITH_INST, 3, 1, 1, FALSE, INT_TYPE},
   {"umulcc", ARITH_INST, 3, 1, 1, TRUE, INT_TYPE},
   {"udiv", ARITH_INST, 3, 1, 1, FALSE, INT_TYPE},
   {"udivcc", ARITH_INST, 3, 1, 1, TRUE, INT_TYPE},
   {"xor", ARITH_INST, 3, 1, 1, FALSE, INT_TYPE},
   {"xorcc", ARITH_INST, 3, 1, 1, TRUE, INT_TYPE},
   {"xnor", ARITH_INST, 3, 1, 1, FALSE, INT_TYPE},
   {"xnorcc", ARITH_INST, 3, 1, 1, TRUE, INT_TYPE},
   {"", 0, 0, 0, 0, 0, 0}
};

/* totopts are used to track the number of transformations for each phase */
struct optinfo totopts[] = {
   { 0, -1, 'B', "reverse branches" },
   { 0, -1, 'C', "branch chaining" },
   { 0, -1, 'D', "dead assignment elimination" },
   { 0, -1, 'E', "local cse" },
   { 0, -1, 'F', "fill delay slots" },
   { 0, -1, 'M', "loop-invariant code motion" },
   { 0, -1, 'O', "copy propagation" },
   { 0, -1, 'P', "peephole optimization" },
   { 0, -1, 'R', "register allocation" },
   { 0, -1, 'U', "unreachable code elimination" },
   { 0, 0, '\0', "" }
};



struct bblk *top = (struct bblk *) NULL;  /* top block in the function    */
struct bblk *bot = (struct bblk *) NULL;  /* end block in the function    */
char funcname[MAXFIELD];                  /* name of the current function */
short functype;                           /* type of the current function */
int numpeeprules;                         /* number of peephole rules     */
int numrulesapplied[MAXRULES];            /* count each rule is applied   */

/*
 * classifyinst - classify an instruction
 */
void classifyinst(short numitems, itemarray items, enum insttype *type,
                  int *instinfonum, int *numdstregs, int *numsrcregs)
{
   int i;
```

```
   /* search through the list of known instructions */
   for (i = 0; insttypes[i].mneumonic[0]; i++)
      if (numitems-1 == insttypes[i].numargs &&
          strcmp(insttypes[i].mneumonic, items[0]) == 0) {
         *type = insttypes[i].type;
         *instinfonum = i;
         *numdstregs = insttypes[i].numdstregs;
         *numsrcregs = insttypes[i].numsrcregs;
         return;
      }
   fprintf(stderr, "classifyinst - %s is unknown instruction\n", items[0]);
   quit(1);
}

/*
 * reclassifyinsts - reclassify the instructions
 */
void reclassifyinsts()
{
   struct bblk *cblk;
   struct assemline *ptr;
   extern struct bblk *top;

   for (cblk = top; cblk; cblk = cblk->down) {
      for (ptr = cblk->lines; ptr; ptr = ptr->next)
         if (INST(ptr->type))
            setupinstinfo(ptr);
   }
}

/*
 * makeinstitems - make the items associated with an instruction
 */
void makeinstitems(char *text, short *numitems, itemarray *items, int strip)
{
   int i, j, commas, tabs;
   char args[4][MAXFIELD], inst[MAXLINE];

   /* strip out the blanks */
   j = 0;
   for (i = 0; text[i]; i++)
      if (text[i] != ' ' || !strip)
         inst[j++] = text[i];
   inst[j] = '\0';

   /* replace the commas with tabs */
   commas = 0;
   tabs = 0;
   for (i = 0; inst[i]; i++)
      if (inst[i] == '\t')
         tabs++;
      else if (tabs == 2 && inst[i] == ',') {
         commas++;
         inst[i] = '\t';
      }

   /* setup the arguments in the instruction */
   if (commas == 2 &&
       sscanf(inst, "\t%s\t%s\t%s\t%s", args[0], args[1], args[2], args[3])
       == 4)
      *numitems = 4;
   else if (commas == 1 &&
            sscanf(inst, "\t%s\t%s\t%s", args[0], args[1], args[2]) == 3)
      *numitems = 3;
   else if (commas == 0 &&
            sscanf(inst, "\t%s\t%s", args[0], args[1]) == 2)
      *numitems = 2;
   else if (commas == 0 && sscanf(inst, "\t%s", args[0]) == 1)
```



```
      *numitems = 1;
   else if (commas == 0 && sscanf(text, "%s = %s", args[0], args[1]) == 2)
      *numitems = 2;
   *items = (itemarray) alloc(*numitems * sizeof(char *));
   for (i = 0; i < *numitems; i++)
      (*items)[i] = allocstring(args[i]);
}

/*
 * setupinstinfo - setup instruction information
 */
void setupinstinfo(struct assemline *ptr)
{
   int numdstregs, numsrcregs;

   makeinstitems(ptr->text, &ptr->numitems, &ptr->items, TRUE);
   classifyinst(ptr->numitems, ptr->items, &ptr->type, &ptr->instinfonum,
                &numdstregs, &numsrcregs);
   setsuses(ptr->text, ptr->type, ptr->instinfonum, ptr->items, ptr->numitems,
            ptr->sets, ptr->uses, numdstregs, numsrcregs);
}

/*
 * readinfunc - read in the function and store the instructions into basic
 *              blocks
 */
int readinfunc()
{
   int n;
   struct assemline *ptr;
   struct bblk *tblk;
   char line[MAXLINE], items[2][MAXFIELD];
   char *status;
   extern int numvars;
   extern struct varinfo vars[];

   /* print out a pending switch to the data segment */
   printf("\t.seg\t\"data\"\n");

   /* read in and dump out directives until read the text segment */
   while ((status = fgets(line, MAXLINE, stdin))) {
      printf("%s", line);
      if (strcmp(line, "\t.seg\t\"text\"\n") == 0)
         break;
   }
   if (!status)
      return FALSE;

   /* check if we are at the beginning of a function */
   fgets(line, MAXLINE, stdin);
   if (strcmp(line, "\t.align\t8\n") != 0) {
      fprintf(stderr, "expecting\n\t.align\t8\n and got\n%s", line);
      quit(1);
   }
   printf("%s", line);
   fgets(line, MAXLINE, stdin);
   if (strncmp(line, "\t.global", 8) != 0) {
      fprintf(stderr, "expecting\n\t.global\t%%s\n and got\n%s", line);
      quit(1);
   }
   printf("%s", line);
   fgets(line, MAXLINE, stdin);
   if (sscanf(line, "\t.proc\t%d", &n) != 1) {
      fprintf(stderr, "expecting\n\t.proc\n and got\n%s", line);
      quit(1);
   }
   printf("%s", line);
   fgets(line, MAXLINE, stdin);
   if (line[strlen(line)-2] != ':') {
```

```
      fprintf(stderr, "readinfunc - expecting label and got\n%s", line);
      quit(1);
   }

   /* setup function name and type */
   strcpy(funcname, line);
   funcname[strlen(funcname)-2] = '\0';
   switch (n) {
      case 0:
         functype = 0;
         break;

      case 6:
         functype = FLOAT_TYPE;
         break;

      case 7:
         functype = DOUBLE_TYPE;
         break;

      default:
         functype = INT_TYPE;
         break;
   }

   /* read in the instructions for the function */
   top = bot = newblk(funcname);
   numvars = 0;
   while ((status = fgets(line, MAXLINE, stdin))) {
      line[strlen(line)-1] = '\0';

      /* check if we are at the end of the function */
      if (strcmp(line, "\t.seg\t\"data\"") == 0) {

         /* clean up last empty block */
         if (!bot->lines) {
            tblk = bot;
            bot = bot->up;
            deleteblk(tblk);
         }
         return TRUE;
      }

      /* ignore nop instructions */
      if (strcmp(line, "\tnop") == 0)
         continue;

      /* store a comment line */
      else if (line[0] == '!') {
         ptr = insline(bot, (struct assemline *) NULL, line);
         ptr->type = COMMENT_LINE;
      }

      /* store a define line */
      else if (sscanf(line, "%s = %s", items[0], items[1]) == 2) {
         ptr = insline(bot, (struct assemline *) NULL, line);
         ptr->type = DEFINE_LINE;
         makeinstitems(ptr->text, &ptr->numitems, &ptr->items, 0);
         if (numvars == MAXVARS) {
            fprintf(stderr, "readinfunc - too many variables in %s\n",
                    funcname);
            quit(1);
         }
         vars[numvars].name = allocstring(ptr->items[0]);
         vars[numvars].type = 0;
         vars[numvars].indirect = FALSE;
         numvars++;
      }
```



```
      /* if a label, then start a new block */
      else if (line[0] == '.' && line[1] == 'L') {
         line[strlen(line)-1] = '\0';
         if (bot->lines || bot->label) {
            tblk = newblk(line);
            tblk->up = bot;
            bot->down = tblk;
            bot = tblk;
         }
         else
            assignlabel(bot, line);
      }

      /* else an instruction */
      else {
         ptr = insline(bot, (struct assemline *) NULL, line);
         setupinstinfo(ptr);
         if ((TOC(ptr->type) && ptr->type != RETURN_INST) ||
             ptr->type == RESTORE_INST) {
            tblk = newblk((char *) NULL);
            tblk->up = bot;
            bot->down = tblk;
            bot = tblk;
         }
      }
   }
   if (!status) {
      fprintf(stderr, "unexpected end of file after function\n");
      quit(1);
   }

   /* clean up last empty block */
   if (!bot->lines) {
      tblk = bot;
      bot = bot->up;
      deleteblk(tblk);
   }

   /* indicate that a function was read in */
   return TRUE;
}

/*
 * dumpblk - dumps the information for a block
 */
void dumpblk(FILE *fout, struct bblk *cblk)
{
   int i;
   struct blist *bptr;
   struct assemline *ptr;
   char vartext[MAXVARLINE], new[MAXLINE];
   extern int swa;
#define COMMENT_START 20
#define USES_START    (COMMENT_START+13)
#define DEADS_START   (USES_START+14)

   /* print out predecessor and successor information */
   if (!swa) {
      fprintf(fout, "!\n");
      fprintf(fout, "! block %d\n", cblk->num);
      fprintf(fout, "! preds:");
      for (bptr = cblk->preds; bptr; bptr = bptr->next)
         fprintf(fout, " %d", bptr->ptr->num);
      fprintf(fout, "\n");
      fprintf(fout, "! succs:");
      for (bptr = cblk->succs; bptr; bptr = bptr->next)
         fprintf(fout, " %d", bptr->ptr->num);
      fprintf(fout, "\n");
      fprintf(fout, "! doms:");
```

```
      bdump(fout, cblk->dom);
      fprintf(fout, "\n");
      fprintf(fout, "! ins=");
      dumpvarstate(vartext, cblk->ins);
      fprintf(fout, "%s\n", vartext);
      fprintf(fout, "! outs=");
      dumpvarstate(vartext, cblk->outs);
      fprintf(fout, "%s\n", vartext);
      fprintf(fout, "!\n");
   }

   /* print out label */
   if (cblk->label)
      fprintf(fout, "%s:\n", cblk->label);

   /* for each assembly line in the block */
   for (ptr = cblk->lines; ptr; ptr = ptr->next) {

      /* print out the text of the assembly line */
      fprintf(fout, "%s", ptr->text);

      /* if the line contains an instruction */
      if (INST(ptr->type) && !swa) {

         /* line up sets after the instruction */
         new[0] = '\0';
         for (i = 1; ptr->text[i]; i++)
            if (ptr->text[i] == '\t') {
               i++;
               break;
            }
         if (!ptr->text[i])
            fprintf(fout, "\t");
         i = strlen(&ptr->text[i]);
         for (; i < COMMENT_START; i++)
            strcat(new, " ");
         fprintf(fout, new);

         /* print out the sets */
         if (ptr->type == CALL_INST) {
            strcpy(vartext, "scratch");
            fprintf(fout, "! sets=%s", vartext);
         }
         else if (ptr->type == SAVE_INST || ptr->type == RESTORE_INST) {
            strcpy(vartext, "window");
            fprintf(fout, "! sets=%s", vartext);
         }
         else {
            dumpvarstate(vartext, ptr->sets);
            fprintf(fout, "! sets=%s", vartext);
         }

         /* line up uses after the sets */
         new[0] = '\0';
         for (i = COMMENT_START+strlen("sets=")+strlen(vartext);
              i < USES_START; i++)
            strcat(new, " ");
         fprintf(fout, new);

         /* print out the uses */
         dumpvarstate(vartext, ptr->uses);
         fprintf(fout, "uses=%s", vartext);

         /* line up deads after the uses */
         new[0] = '\0';
         for (i = USES_START+strlen("uses=")+strlen(vartext);
              i < DEADS_START; i++)
            strcat(new, " ");
         fprintf(fout, new);
```



```
         /* print out the deads */
         dumpvarstate(vartext, ptr->deads);
         fprintf(fout, "deads=%s", vartext);
      }

      /* print out the carriage return */
      fprintf(fout, "\n");
      if ((ptr->type == CALL_INST || ptr->type == BRANCH_INST ||
           ptr->type == RETURN_INST) && !ptr->next)
         fprintf(fout, "\tnop\n");
   }
}

/*
 * dumpoutblks - dumps a range of blocks to a file pointer
 */
void dumpoutblks(FILE *fout, unsigned int num1, unsigned int num2)
{
   struct bblk *cblk;
   void dumpblk(FILE *, struct bblk *);
   extern struct bblk *top;

   /* find the start block */
   for (cblk = top; cblk; cblk = cblk->down)
      if (cblk->num == num1)
         break;

   /* check if no more blocks */
   if (!cblk) {
      fprintf(fout, "no blocks in the range %d to %d\n", num1, num2);
      return;
   }

   /* print each block in the range */
   for (; cblk; cblk = cblk->down)
      if (cblk->num <= num2)
         dumpblk(fout, cblk);
      else
         break;
}

/*
 * dumpblks - diagnostic function to dump out a range of blocks
 */
void dumpblks(int num1, int num2)
{
   dumpoutblks(stderr, num1, num2);
}

/*
 * dumpfunc - write out the function to stdout
 */
void dumpfunc()
{
   struct bblk *cblk;
   extern int swa;

   /* dump out loop information */
   if (!swa)
      dumploops(stdout);

   /* order the preds in each block */
   orderpreds();

   /* print out each block in the program */
   for (cblk = top; cblk; cblk = cblk->down)
      dumpblk(stdout, cblk);
}
```

```
/*
 * dumpruleusage - dump out how many times each rule was applied
 */
void dumpruleusage()
{
   int i, n;

   /* print out which peephole optimization rules were applied */
   n = 0;
   for (i = 0; i < numpeeprules; i++)
      n += numrulesapplied[i];
   fprintf(stderr, "\nNumber of peeprules is %d, applied %d times.\n",
           numpeeprules, n);
   for (i = 0; i < numpeeprules; i++)
      if (numrulesapplied[i] > 0) {
         fprintf(stderr, "Peephole rule %d applied %d times.\n",
                 i+1, numrulesapplied[i]);
      }
}

/*
 * dumpfunccounts - dump out the number of instructions and memory references
 *                  for the function
 */
void dumpfunccounts()
{
   int i, numinsts, nummems;
   int insts[MAXLOOPLEVELS];
   int mems[MAXLOOPLEVELS];
   struct bblk *cblk;
   struct assemline *ptr;
   static int first = TRUE;

   numinsts = nummems = 0;
   for (i = 0; i < MAXLOOPLEVELS; i++)
      insts[i] = mems[i] = 0;
   for (cblk = top; cblk; cblk = cblk->down)
      for (ptr = cblk->lines; ptr; ptr = ptr->next)
         if (INST(ptr->type)) {
            numinsts++;
            insts[cblk->loopnest]++;
            if (ptr->type == LOAD_INST || ptr->type == STORE_INST) {
               nummems++;
               mems[cblk->loopnest]++;
            }
            if ((ptr->type == CALL_INST || ptr->type == BRANCH_INST ||
                 ptr->type == RETURN_INST) && !ptr->next) {
               numinsts++;
               insts[cblk->loopnest]++;
            }
         }
   if (first) {
      fprintf(stderr, "function     level instructions memory refs\n");
      fprintf(stderr, "------------ ----- ------------ ------------\n");
      first = FALSE;
   }
   fprintf(stderr, "%-12s total %12d %11d\n", funcname, numinsts, nummems);
   totnuminsts += numinsts;
   totnummems += nummems;
   if (insts[1] > 0)
      for (i = 0; i < MAXLOOPLEVELS && insts[i] > 0; i++) {
         totinsts[i] += insts[i];
         totmems[i] += mems[i];
         fprintf(stderr, "      %5d %12d %11d\n", i, insts[i], mems[i]);
      }
   else {
      totinsts[0] += insts[0];
      totmems[0] += mems[0];
```



```
   }
}

/*
 * dumptotalcounts - dump out the number of instructions and memory references
 *                   for the entire file
 */
void dumptotalcounts()
{
   int i;

   fprintf(stderr, "------------ ----- ------------ ------------\n");
   fprintf(stderr, "%-12s total %12d %11d\n", "program",
           totnuminsts, totnummems);
   if (totinsts[1] > 0)
      for (i = 0; i < MAXLOOPLEVELS && totinsts[i] > 0; i++) {
         fprintf(stderr, "      %5d %12d %11d\n",
                 i, totinsts[i], totmems[i]);
      }
}

/*
 * dumpoptcounts - dump out the number of transformations applied for
 *                 each optimization
 */
void dumpoptcounts()
{
   int i, n;

   fprintf(stderr, "\n");
   n = 0;
   for (i = 0; totopts[i].optchar; i++)
      if (totopts[i].count) {
         n += totopts[i].count;
         fprintf(stderr, "%3d transformations applied by %s phase.\n",
                 totopts[i].count, totopts[i].name);
      }
   fprintf(stderr, "---\n");
   fprintf(stderr, "%3d transformations applied by all optimization phases.\n",
           n);
}
```

```
/*
 * miscellaneous support functions
 */
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>
#include <string.h>
#include <setjmp.h>
#include "opt.h"
#include "io.h"
#include "misc.h"
#include "vars.h"
#include "vect.h"

/*
 * alloc - allocates space and checks the status of the allocation
 */
void *alloc(unsigned int bytes)
{
   void *ptr;

   if ((ptr = malloc(bytes)))
      return ptr;
   fprintf(stderr, "alloc: ran out of space\n");
   quit(1);

   /* should never reach here */
   return (void *) 0;
}

/*
 * allocstring - allocate space for a string and copy the string to that
 *               location
 */
char *allocstring(char *str)
{
   char *dst;

   dst = (char *) alloc(strlen(str)+1);
   strcpy(dst, str);
   return dst;
}

/*
 * replacestring - replace a dynamically allocated string
 */
void replacestring(char **s1, char *old, char *new)
{
   int i;
   char *p, *s, *d;
   char t[MAXFIELD];

   if (*s1) {
      p = strstr(*s1, old);
      for (s = *s1, d = t; s != p; *d++ = *s++)
         ;
      strcpy(d, new);
      d += strlen(new);
      for (i = 0; i < strlen(old); i++)
         s++;
      while ((*d++ = *s++))
         ;
      if (strlen(old) >= strlen(new))
         strcpy(*s1, t);
      else {
         free(*s1);
         *s1 = allocstring(t);
      }
      return;
```



```
   }
   fprintf(stderr, "replacestring - dst string not yet allocated\n");
   quit(1);
}

/*
 * createmove - create a move instruction
 */
void createmove(int type, char *src, char *dst, struct assemline *ptr)
{
   char mnem[MAXFIELD], line [MAXLINE];

   if (type == INT_TYPE)
      strcpy(mnem, "mov");
   else if (type == FLOAT_TYPE)
      strcpy(mnem, "fmovs");
   else if (type == DOUBLE_TYPE)
      strcpy(mnem, "fmovd");
   else {
      fprintf(stderr, "createmove - bad type %d\n", type);
      quit(1);
   }
   sprintf(line, "\t%s\t%s,%s", mnem, src, dst);
   free(ptr->text);
   ptr->text = allocstring(line);
   setupinstinfo(ptr);
}

/*
 * assignlabel - assigns a label to a basic block
 */
void assignlabel(struct bblk *cblk, char *label)
{
   /* assign label */
   if (cblk->label)
      free(cblk->label);
   if (label)
      cblk->label = allocstring(label);
   else
      cblk->label = (char *) NULL;
}

/*
 * newblk - allocate a new basic block
 */
struct bblk *newblk(char *label)
{
   struct bblk *tblk;

   /* allocate the space for the block */
   tblk = (struct bblk *) alloc(sizeof(struct bblk));

   /* initialize the fields of the block */
   tblk->label = (char *) NULL;
   assignlabel(tblk, label);
   tblk->num = 0;
   tblk->loopnest = 0;
   tblk->lines = (struct assemline *) NULL;
   tblk->lineend = (struct assemline *) NULL;
   tblk->preds = (struct blist *) NULL;
   tblk->succs = (struct blist *) NULL;
   tblk->up = (struct bblk *) NULL;
   tblk->down = (struct bblk *) NULL;
   tblk->dom = binit();
   tblk->loop = (struct loopnode *) NULL;
   varinit(tblk->uses);
   varinit(tblk->defs);
   varinit(tblk->ins);
   varinit(tblk->outs);
```

```
   tblk->status = 0;

   /* return the pointer to the block */
   return tblk;
}

/*
 * freeblk - frees up the space for a basic block
 */
void freeblk(struct bblk *cblk)
{
   struct assemline *ptr, *dptr;

   /* free label */
   if (cblk->label)
      free(cblk->label);

   /* free bit vectors */
   if (cblk->dom)
      bfree(cblk->dom);

   /* free blists */
   freeblist(cblk->preds);
   freeblist(cblk->succs);

   /* free assemlines */
   for (ptr = cblk->lines; ptr; ) {
      dptr = ptr;
      ptr = ptr->next;
      freeline(dptr);
   }
}

/*
 * inblist - check if a block is in a blist
 */
int inblist(struct blist *head, struct bblk *cblk)
{
   struct blist *bptr;

   for (bptr = head; bptr; bptr = bptr->next)
      if (bptr->ptr == cblk)
         return TRUE;
   return FALSE;
}

/*
 * freeblist - free up the space for a blist
 */
void freeblist(struct blist *head)
{
   struct blist *bptr, *dptr;

   for (bptr = head; bptr; ) {
      dptr = bptr;
      bptr = bptr->next;
      free(dptr);
   }
}

/*
 * newline - allocate a new assembly line
 */
struct assemline *newline(char *text)
{
   struct assemline *tline;

   /* allocate space for the assembly line */
   tline = (struct assemline *) alloc(sizeof(struct assemline));
```



```
   /* initialize the other fields of the assembly line */
   tline->text = allocstring(text);
   tline->next = tline->prev = (struct assemline *) NULL;
   varinit(tline->sets);
   varinit(tline->uses);
   varinit(tline->deads);
   tline->blk = (struct bblk *) NULL;

   /* return the pointer to the assembly line */
   return tline;
}

/*
 * hookupline - hook up the assembly line within the basic block
 */
void hookupline(struct bblk *cblk, struct assemline *ptr,
                struct assemline *line)
{
   /* hook this assembly line into the basic block before ptr */
   if (!ptr) {
      if (!cblk->lineend) {
         cblk->lines = cblk->lineend = line;
         line->prev = line->next = (struct assemline *) NULL;
      }
      else {
         cblk->lineend->next = line;
         line->prev = cblk->lineend;
         line->next = (struct assemline *) NULL;
         cblk->lineend = line;
      }
   }
   else {
      line->next = ptr;
      if (!(line->prev = ptr->prev))
         cblk->lines = line;
      else
         line->prev->next = line;
      ptr->prev = line;
   }
   line->blk = cblk;
}

/*
 * unhookline - unhook an assembly line from a basic block
 */
void unhookline(struct assemline *ptr)
{
   if (ptr->prev)
      ptr->prev->next = ptr->next;
   else
      ptr->blk->lines = ptr->next;
   if (ptr->next)
      ptr->next->prev = ptr->prev;
   else
      ptr->blk->lineend = ptr->prev;
   ptr->next = ptr->prev = (struct assemline *) NULL;
}

/*
 * insline - insert the line before the one in the argument
 */
struct assemline *insline(struct bblk *cblk, struct assemline *ptr, char *text)
{
   struct assemline *tline;

   /* allocate the assembly line */
   tline = newline(text);
```

```
   /* hook up the assembly line into the basic block */
   hookupline(cblk, ptr, tline);

   /* return the inserted assembly line */
   return tline;
}

/*
 * delline - delete the specified line in the basic block
 */
void delline(struct assemline *ptr)
{
   unhookline(ptr);
   freeline(ptr);
}

/*
 * freeline - deallocate an assembly line
 */
void freeline(struct assemline *ptr)
{
   int i;

   for (i = 0; i < ptr->numitems; i++)
      free(ptr->items[i]);
   free(ptr->text);
   free(ptr);
}

/*
 * addtoblist - add a basic block to a blist
 */
void addtoblist(struct blist **head, struct bblk *cblk)
{
   struct blist *bptr;

   /* first check that the basic block is not already in the blist */
   for (bptr = *head; bptr; bptr = bptr->next)
      if (bptr->ptr == cblk)
         return;

   /* allocate the space for the blist element */
   bptr = (struct blist *) alloc(sizeof(struct blist));

   /* link in the block at the head of the list */
   bptr->ptr = cblk;
   bptr->next = *head;
   *head = bptr;
}

/*
 * sortblist - sort the blocks in blist by the block number
 */
void sortblist(struct blist *head)
{
   struct blist *bptr, *bptr2;
   struct bblk *tblk;

   for (bptr = head; bptr; bptr = bptr->next)
      for (bptr2 = bptr->next; bptr2; bptr2 = bptr2->next)
         if (bptr->ptr->num > bptr2->ptr->num) {
            tblk = bptr->ptr;
            bptr->ptr = bptr2->ptr;
            bptr2->ptr = tblk;
         }
}

/*
 * orderpreds - make the predecessors of each block be in ascending order
```



```
 */
void orderpreds()
{
   struct bblk *cblk;
   extern struct bblk *top;

   for (cblk = top; cblk; cblk = cblk->down)
      sortblist(cblk->preds);
}

/*
 * deleteblk - delete a basic block from the list of basic blocks
 */
void deleteblk(struct bblk *cblk)
{
   extern struct bblk *bot;

   /* update bottom block if needed */
   if (cblk == bot)
      bot = cblk->up;

   /* unhook the "up" and "down" pointers */
   unlinkblk(cblk);

   /* unhook preds */
   delfrompreds_succs(cblk);

   /* unhook succs */
   delfromsuccs_preds(cblk);

   /* free up the memory */
   freeblk(cblk);
}

/*
 * unlinkblk - unhook a basic block from the list of basic blocks
 */
void unlinkblk(struct bblk *cblk)
{
   extern struct bblk *top;

   /* relink a backward pointer to bypass the block to be deleted */
   if (cblk->down)
      cblk->down->up = cblk->up;

   /* set a forward pointer to bypass the block to be deleted */
   if (!cblk->up)
      top = cblk->down;
   else
      cblk->up->down = cblk->down;
}

/*
 * delfrompreds_succs - delete cblk from the successor list of all the
 *                      predecessors of cblk
 */
void delfrompreds_succs(struct bblk *cblk)
{
   struct blist *curpred;

   for (curpred=cblk->preds; curpred; curpred=curpred->next)
      if (!delfromblist(&(curpred->ptr->succs), cblk)) {
         fprintf(stderr, "delfrompreds_succs(), basic block not found.\n");
         quit(1);
      }
}

/*
 * delfromsuccs_preds - delete cblk from the predecessor list of all the
```

```
 *                      successors of cblk
 */
void delfromsuccs_preds(struct bblk *cblk)
{
    struct blist *cursucc;

    for (cursucc=cblk->succs; cursucc; cursucc=cursucc->next)
       if (!delfromblist(&(cursucc->ptr->preds), cblk)) {
          fprintf(stderr, "delfromsuccs_preds(), basic block not found.\n");
          quit(1);
       }
}

/*
 * delfromblist - deletes a block from a blist
 */
struct bblk *delfromblist(struct blist **head, struct bblk *cblk)
{
   struct bblk *tblk;
   struct blist *bptr, *pbptr;

   /* check if the basic block already has been allocated */
   tblk = (struct bblk *) NULL;
   pbptr = (struct blist *) NULL;
   for (bptr = *head; bptr; pbptr = bptr, bptr = bptr->next)
      if (bptr->ptr == cblk) {
         tblk = bptr->ptr;
         if (pbptr)
            pbptr->next = bptr->next;
         else
            *head = bptr->next;
         free(bptr);
         break;
      }
   return tblk;
}

/*
 * newloop - allocate a new loop
 */
struct loopnode *newloop()
{
   struct loopnode *loop;
   extern struct loopnode *loops;

   loop = (struct loopnode *) alloc(sizeof(struct loopnode));
   loop->header = (struct bblk *) NULL;
   loop->preheader = (struct bblk *) NULL;
   loop->blocks = (struct blist *) NULL;
   varinit(loop->invregs);
   varinit(loop->sets);
   loop->anywrites = 0;
   loop->next = loops;
   loops = loop;
   return loop;
}

/*
 * freeloops - free up the loop structures
 */
void freeloops()
{
   struct bblk *cblk;
   struct loopnode *lptr, *dlptr;
   extern struct bblk *top;
   extern struct loopnode *loops;

   /* clean up loop information associated with each basic block */
   for (cblk = top; cblk; cblk = cblk->down) {
```



```
      cblk->loop = (struct loopnode *) NULL;
      cblk->loopnest = 0;
      bfree(cblk->dom);
   }

   /* free up information associated with each loop */
   for (lptr = loops; lptr; ) {
      freeblist(lptr->blocks);
      dlptr = lptr;
      lptr = lptr->next;
      free(dlptr);
   }

   /* indicate there is currently no loop information calculated */
   loops = (struct loopnode *) NULL;
}

/*
 * dumploops - dumps the loops in the function to the specified file pointer
 */
void dumploops(FILE *fout)
{
   struct loopnode *lptr;
   void dumploop(FILE *, struct loopnode *);
   extern struct loopnode *loops;

   fprintf(fout, "! loops in function\n");
   for (lptr = loops; lptr; lptr = lptr->next)
      dumploop(fout, lptr);
   fprintf(fout, "\n");
}

/*
 * dumploop - dump a loop in the program to the specified file pointer
 */
void dumploop(FILE *fout, struct loopnode *lptr)
{
   struct blist *bptr;

   fprintf(fout, "!  loop: head = %d\n", lptr->header->num);
   fprintf(fout, "!     blocks =");
   sortblist(lptr->blocks);
   for (bptr = lptr->blocks; bptr; bptr = bptr->next)
      fprintf(fout, " %d", bptr->ptr->num);
   fprintf(fout, "\n");
}

/*
 * incropt - Increment the count of transformations for an optimization.
 *           Calls to this function should be placed immediately before
 *           a transformation for an optimization is about to be applied.
 *
 */
void incropt(enum opttype opt)
{
   extern int moreopts;
   extern jmp_buf my_env;
   extern struct optinfo totopts[];

   if (totopts[opt].count == totopts[opt].max) {
      moreopts = FALSE;
      longjmp(my_env, 1);
   }
   totopts[opt].count++;
   return;
}

/*
 * quit - exits the program
```

```
 */
void quit(int flag)
{
   exit(flag);
}
```

```
/*
 * main driver for the program
 */
#include <stdio.h>
#include <setjmp.h>
#include <ctype.h>
#include "opt.h"
#include "analysis.h"
#include "flow.h"
#include "io.h"
#include "misc.h"
#include "opts.h"

int swa = FALSE;      /* assembly with no extra comments   */
int swb = FALSE;      /* reverse branches flag             */
int swc = FALSE;      /* branch chaining flag              */
int swd = FALSE;      /* dead assignment elimination flag  */
int swe = FALSE;      /* local cse flag                    */
int swf = FALSE;      /* fill delay slots flag             */
int swm = FALSE;      /* loop-invariant code motion flag   */
int swo = FALSE;      /* copy propagation flag             */
int swp = FALSE;      /* peephole optimization flag        */
int swr = FALSE;      /* register allocation flag          */
int swu = FALSE;      /* unreachable code elimination flag */

int moreopts = TRUE;  /* should more optimizations be performed */
jmp_buf my_env;

/*
 * checkflags - check the optimization flags
 */
void checkflags(char *flags)
{
   int i, j;
   extern struct optinfo totopts[];

   /* check that compilation flags begin with a '-' */
   if (flags[0] != '-') {
      fprintf(stderr,
              "checkflags - optimization flags should begin with '-'\n");
      quit(1);
   }

   /* set all the appropriate flag variables */
   for (i = 1; flags[i]; i++) {
      switch (flags[i]) {
         case 'A':
            swa = TRUE;
            break;

         case 'B':
            swb = TRUE;
            break;

         case 'C':
            swc = TRUE;
            break;

         case 'D':
            swd = TRUE;
            break;

         case 'E':
            swe = TRUE;
            break;

         case 'F':
            swf = TRUE;
            break;
```



```
         case 'M':
            swm = TRUE;
            break;

         case 'O':
            swo = TRUE;
            break;

         case 'P':
            swp = TRUE;
            break;

         case 'R':
            swr = TRUE;
            break;

         case 'U':
            swu = TRUE;
            break;

         default:
            fprintf(stderr, "%c is an invalid optimization flag\n", flags[i]);
            quit(1);
      }

      /* set maximum number of transformations for the optimization if it
         is specfied */
      if (isdigit((int) flags[i+1])) {
         for (j = 0; totopts[j].optchar; j++)
            if (totopts[j].optchar == flags[i])
               break;
         if (!totopts[j].optchar) {
            fprintf(stderr, "flag %c not in totopts\n", flags[i]);
            quit(1);
         }
         sscanf(&flags[i+1], "%d", &totopts[j].max);
         for ( ; isdigit((int) flags[i+1]); i++)
            ;
      }
   }
}

/*
 * main - main function for the assembly optimizer
 */
int main(int argc, char *argv[])
{
   int changes, anychanges;
   char *flags = "-BCDEFMOPRU";

   /* read in peephole optimization rules */
   readinrules();

   /* check for flags */
   if (argc == 1)
      checkflags(flags);
   else if (argc == 2)
      checkflags(argv[1]);
   else {
      fprintf(stderr, "main - wrong number of arguments\n");
      quit(1);
   }

   /* process each function in the file */
   while (readinfunc()) {

      /* setup the control flow between the basic blocks */
      setupcontrolflow();
```

```
   /* setup to stop applying transformations */
   setjmp(my_env);
   if (moreopts) {

      /* perform branch optimizations */
      if (swc)
         remvbranchchains();
      if (swb)
         reversebranches();

      /* remove unreachable code */
      if (swu)
         unreachablecodeelim();
   }

   /* find the loops in the function */
   numberblks();
   findloops();

   if (moreopts) {

      /* allocate variables to registers */
      if (swr)
         regalloc(&changes);

      /* apply peephole optimization rules */
      calclivevars();
      calcdeadvars();
      if (swp) {
         applypeeprules(&changes);
         if (changes) {
            calclivevars();
            calcdeadvars();
         }
      }

      /* perform local common subexpression elimination */
      if (swe)
         localcse(&changes);

      /* perform copy propagation */
      if (swo) {
         calclivevars();
         calcdeadvars();
         localcopyprop(&changes);
      }

      /* calculate live variable information */
      calclivevars();

      /* remove dead assignments */
      if (swd)
         deadasgelim();

      /* apply peephole optimization rules */
      calclivevars();
      calcdeadvars();
      if (swp) {
         applypeeprules(&changes);
         if (changes) {
            calclivevars();
            calcdeadvars();
         }
      }

      /* repeatedly apply loop-invariant code motion, common subexpression
         elimination, copy propagation, and dead assignment elimination */
      do {
```



```
            anychanges = FALSE;
            if (swm) {
               codemotion(&anychanges);
               if (swe) {
                  changes = FALSE;
                  localcse(&changes);
                  if (swo) {
                     calclivevars();
                     calcdeadvars();
                     localcopyprop(&changes);
                     if (swd) {
                        calclivevars();
                        deadasgelim();
                     }
                  }
                  if (changes)
                     anychanges = TRUE;
               }
               if (anychanges) {
                  calclivevars();
                  calcdeadvars();
                  changes = FALSE;
                  if (swp) {
                     applypeeprules(&changes);
                     if (changes) {
                        calclivevars();
                        calcdeadvars();
                     }
                  }
               }
            }
         } while (anychanges);

         /* fill delay slots */
         if (swf)
            filldelayslots();
      }

      /* dump out the assembly code */
      dumpfunc();

      /* dump out counts */
      dumpfunccounts();
   }

   /* dump out total counts */
   dumptotalcounts();

   /* dump out number of transformations for each phase */
   dumpoptcounts();

   /* dump out rule usage */
   dumpruleusage();

   /* successfully terminate execution */
   return 0;
}
```

```
/*
 * register and local variable analysis functions
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "opt.h"
#include "misc.h"
#include "vars.h"


/*
 * regstring provides a mapping from a register position to the assembly
 * string representing the register
 */
char *regstring[] = {
  "%g0", "%g1", "%g2", "%g3", "%g4", "%g5", "%g6", "%g7",
  "%o0", "%o1", "%o2", "%o3", "%o4", "%o5", "%sp", "%o7",
  "%l0", "%l1", "%l2", "%l3", "%l4", "%l5", "%l6", "%l7",
  "%i0", "%i1", "%i2", "%i3", "%i4", "%i5", "%fp", "%i7"
};

struct varinfo vars[MAXVARS];   /* variable information */
int numvars;                    /* number of variables  */

/*
 * varinit - initialize a variable state
 */
void varinit(varstate v)
{
   int i;

   for (i = 0; i < NUMVARWORDS; i++)
      v[i] = 0;
}

/*
 * varcmp - returns TRUE if two variable states are identical
 */
int varcmp(varstate v1, varstate v2)
{
   int i;

   for (i = 0; i < NUMVARWORDS; i++)
      if (v1[i] != v2[i])
         return FALSE;
   return TRUE;
}

/*
 * varempty - returns TRUE if a variable state is empty
 */
int varempty(varstate v)
{
   int i;

   for (i = 0; i < NUMVARWORDS; i++)
      if (v[i] != 0)
         return FALSE;
   return TRUE;
}

/*
 * unionvar - union two varstates together and store in a third
 */
void unionvar(varstate vd, varstate v1, varstate v2)
{
   int i;
```



```
   for (i = 0; i < NUMVARWORDS; i++)
      vd[i] = v1[i] | v2[i];
}

/*
 * intervar - intersect two varstates together and store in a third
 */
void intervar(varstate vd, varstate v1, varstate v2)
{
   int i;

   for (i = 0; i < NUMVARWORDS; i++)
      vd[i] = v1[i] & v2[i];
}

/*
 * minusvar - subtract one varstate from another and store in a third
 */
void minusvar(varstate vd, varstate v1, varstate v2)
{
   int i;

   for (i = 0; i < NUMVARWORDS; i++)
      vd[i] = v1[i] & ~v2[i];
}

/*
 * varcopy - copy one varstate to another
 */
void varcopy(varstate vd, varstate vs)
{
   int i;

   for (i = 0; i < NUMVARWORDS; i++)
      vd[i] = vs[i];
}

/*
 * varcommon - compare if two varstates have any variables in common
 */
int varcommon(varstate v1, varstate v2)
{
   int i;

   for (i = 0; i < NUMVARWORDS; i++)
      if (v1[i] & v2[i])
         return TRUE;
   return FALSE;
}

/*
 * delreg - delete a register from a variable state
 */
void delreg(char *regstr, varstate vars, int numreg)
{
   int regnum, i;

   /* determine the starting index associated with the register */
   regnum = calcregpos(regstr);
   if (regnum == -1)
      return;

   /* remove the bit(s) associated with the register(s) */
   for (i = 0; i < numreg; i++)
      if (regnum+i < 32)
         vars[0] &= ~(1 << (regnum+i));
      else
         vars[1] &= ~(1 << ((regnum+i) - 32));
```

```
}

/*
 * delvar − delete a bit associated with a memory variable
 */
void delvar(varstate v, int pos)
{
   v[2] &= ~(1 << pos);
}

/*
 * calcregpos − calculates the register position of a register
 */
int calcregpos(char *regtext)
{
   int num;
   char type;

   if (strcmp(regtext, "%fp") == 0)
      return 30;
   else if (strcmp(regtext, "%sp") == 0)
      return 14;
   else if (sscanf(regtext, "%%%c%d", &type, &num) == 2) {
      if (type == 'g')
         return num;
      else if (type == 'o')
         return num+8;
      else if (type == 'l')
         return num+16;
      else if (type == 'i')
         return num+24;
      else if (type == 'f')
         return num+32;
   }
   return −1;
}

/*
 * isreg − determines if the string is a register
 */
int isreg(char *regtext)
{
   return calcregpos(regtext) != −1;
}

/*
 * insreg − set a bit associated with a register
 */
void insreg(char *reg, varstate v, int numreg)
{
   int num, i;

   /* calculate the starting bit position associated with the register */
   num = calcregpos(reg);
   if (num == −1)
      return;
   if (numreg > 1 && !(*reg == '%' && *(reg+1) == 'f' &&
       isdigit((int) *(reg+2))))
      numreg = 1;

   /* set the bit(s) associated with the register(s) */
   for (i = 0; i < numreg; i++)
      if (num+i < 32)
         v[0] |= (1 << (num+i));
      else
         v[1] |= (1 << ((num+i)−32));
}

/*
```



```
 * insvar − set a bit associated with a memory variable
 */
void insvar(varstate v, int pos)
{
   v[2] |= (1 << pos);
}

/*
 * regexists − checks if a register is in a register state
 */
int regexists(char *reg, varstate v)
{
   int num;

   num = calcregpos(reg);
   if (num < 32)
      return v[0] & (1 << num);
   else
      return v[1] & (1 << (num−32));
}

/*
 * setsuses − set the bits associated with the registers set and used in an
 *            instruction
 */
void setsuses(char *text, enum insttype type, int instinfonum, itemarray items,
              int numitems, varstate sets, varstate uses, int numdstregs,
              int numsrcregs)
{
   int i, j;
   char c, *p;
   char fields[4][MAXFIELD], tmp[MAXLINE];
   extern int numvars;
   extern short functype;
   extern struct instinfo insttypes[];

   /* initialize the sets and uses fields */
   varinit(sets);
   varinit(uses);

   /* insert the register sets and uses depending on the type of the
      instruction */
   if (!INST(type))
      return;
   switch (type) {
      case ARITH_INST:
      case CONV_INST:
      case MOV_INST:
         switch (numitems) {
            case 4:
               insreg(items[1], uses, numsrcregs);
               insreg(items[2], uses, numsrcregs);
               insreg(items[3], sets, numdstregs);
               break;

            case 3:
               insreg(items[1], uses, numsrcregs);
               insreg(items[2], sets, numdstregs);
               break;

            default:
               fprintf(stderr,
                       "setsuses − should not have %d args for ARITH_INST\n",
                       numitems − 1);
               quit(1);
         }
         break;

      case BRANCH_INST:
```

```
         break;

      case CALL_INST:

         /* should set all of the scratch registers since the called
            function could overwrite all of them */
         sets[0] = 0x0000FFFE;
         sets[1] = 0xFFFFFFFF;

         /* set the uses to be all of the arguments registers that are
            passed to the called function */
         j = atoi(items[2]);
         for (i = 0; i < j; i++)
            uses[0] |= (1 << (i+8));
         break;

      case CMP_INST:
         insreg(items[1], uses, numsrcregs);
         insreg(items[2], uses, numsrcregs);
         break;

      case JUMP_INST:
         break;

      case LOAD_INST:
      case STORE_INST:
         if (type == LOAD_INST)
            strcpy(tmp, items[1]);
         else
            strcpy(tmp, items[2]);
         j = 0;
         for (i = 1; tmp[i]; i++)
            if (tmp[i] == '+' || tmp[i] == ']')
               break;
            else
               fields[0][j++] = tmp[i];
         fields[0][j] = '\0';
         j = 0;
         if (tmp[i] == '+')
            for (i++; tmp[i]; i++) {
               if (tmp[i] == ']')
                  break;
               else
                  fields[1][j++] = tmp[i];
            }
         fields[1][j] = '\0';
         if (!tmp[i]) {
            fprintf(stderr, "setsuses - could not find ] in load %s\n", text);
            quit(1);
         }
         insreg(fields[0], uses, numsrcregs);
         insreg(fields[1], uses, numsrcregs);
         if (type == LOAD_INST)
            insreg(items[2], sets, numdstregs);
         else
            insreg(items[1], uses, numsrcregs);
         break;

      case RESTORE_INST:
      case SAVE_INST:
         if (numitems == 4) {
            insreg(items[1], uses, numsrcregs);
            insreg(items[2], uses, numsrcregs);
            insreg(items[3], sets, numsrcregs);
         }
         break;

      case RETURN_INST:
         if (functype == INT_TYPE)
```



```
            insreg("%i0", uses, 1);
         else if (functype == FLOAT_TYPE)
            insreg("%f0", uses, 1);
         else if (functype == DOUBLE_TYPE)
            insreg("%f0", uses, 2);
         break;

      case COMMENT_LINE:
      case DEFINE_LINE:
         break;
   }

   /* check if a variable has had its address taken, determine
      the type of the variable, and update the sets and uses
      of the load or store */
   for (i = 0; i < numvars; i++)
      for (j = 1; j < numitems; j++)
         if ((p = strstr(items[j], vars[i].name))) {
            if ((c = *(p+strlen(vars[i].name))) == ']') {
               vars[i].type = insttypes[instinfonum].datatype;
               if (type == LOAD_INST)
                  insvar(uses, i);
               else if (type == STORE_INST)
                  insvar(sets, i);
               else {
                  fprintf(stderr, "setuses - memref not in load or store\n");
                  quit(1);
               }
            }
            else if (!isalnum((int) c) && c != '_')
               vars[i].indirect = TRUE;
         }

}

/*
 * allocreg - allocate a register of a given type
 */
int allocreg(short type, varstate r, char *reg)
{
   int i;
   int noallocate = 0xc000c000;  /* don't allocate %sp, %o7, %fp, %i7 */

   /* find the first available allocable register of a given type,
      copy the string associated with the register, and indicate
      that an available register was found */
   if (type == INT_TYPE) {
      for (i = 1; i < 32; i++)
         if (!(r[0] & (1 << i)) && !(noallocate & (1 << i))) {
            strcpy(reg, regstring[i]);
            return TRUE;
         }
      return FALSE;
   }
   else if (type == FLOAT_TYPE) {
      for (i = 0; i < 32; i++)
         if (!(r[1] & (1 << i))) {
            sprintf(reg, "%%f%d", i);
            return TRUE;
         }
      return FALSE;
   }
   else if (type == DOUBLE_TYPE) {
      for (i = 0; i < 32; i += 2)
         if (!(r[1] & (1 << i)) && !(r[1] & (1 << (i+1)))) {
            sprintf(reg, "%%f%d", i);
            return TRUE;
         }
      return FALSE;
```

```
   }
   else {
      fprintf(stderr, "allocreg - invalid register type %d\n", type);
      quit(1);
   }

   /* should never reach here */
   return FALSE;
}

/*
 * varname - return the variable name associated with the position of
 *           of the variable
 */
char *varname(int pos)
{
   return vars[pos].name;
}

/*
 * dumpvarstate - dump the variables that are in the variable state variable
 */
void dumpvarstate(char *out, varstate v)
{
   int i, j;
   char new[MAXVARLINE];

   /* dump the lower word first */
   out[0] = '\0';
   new[0] = '\0';
   for (i = 0; i < 32; i++)
      if (v[0] & (1 << i)) {
         sprintf(new, "%s:", regstring[i]);
         strcat(out, new);
      }

   /* dump the middle word next */
   new[0] = '\0';
   for (i = 0; i < MAXREGS-32; i++)
      if (v[1] & (1 << i)) {
         sprintf(new, "%%f%d:", i);
         strcat(out, new);
      }

   /* dump the upper word next */
   new[0] = '\0';
   for (i = 0; i < numvars; i++)
      if (v[2] & (1 << i)) {
         sprintf(new, "%s:", varname(i));
         j = 0;
         if (new[j] == '.') {
            j++;
            while (isdigit((int) new[j++]))
               ;
         }
         else {
            fprintf(stderr, "dumpvarstate - illegal variable name %s\n", new);
            quit(1);
         }
         strcat(out, &new[j]);
      }
}
```

```
/*
 * block bit vector manipulation functions
 */
#include <stdlib.h>
#include <stdio.h>
#include "opt.h"
#include "vect.h"

int bvectlen;   /* length of a basic block vector in integers */

/*
 * binsert - add an item to a basic block vector
 */
void binsert(bvect *sptr, unsigned int r)
{
   unsigned int *ptr, *end;

   /* if the bit vector has not yet been allocated */
   if (!*sptr) {

      /* allocate the space and initialize all of the words to zero */
      *sptr = (bvect) (ptr = BALLOC);
      for (end = ptr + bvectlen; ptr != end; *ptr++ = 0)
         ;
   }

   /* set the bit that corresponds to the basic block number */
   (*sptr)[r >> LOG2_INT] |= (((unsigned) 1) << (r & INT_REM));
}

/*
 * bdelete - delete an item from a basic block vector
 */
void bdelete(bvect *sptr, unsigned int r)
{
   if (*sptr)
      (*sptr)[r >> LOG2_INT] &= ~(((unsigned) 1) << (r & INT_REM));
}

/*
 * bunion - computes the union of two bvects
 */
void bunion(bvect *result, bvect vec)
{
   unsigned int *ptr1, *ptr2, *end;

   /* if the source vector is allocated */
   if ((ptr2 = (unsigned int *) vec)) {

      /* if the result vector is allocated, then union in all of the
         words of the source vector */
      if ((ptr1 = (unsigned int *) *result))
         for (end = ptr1 + bvectlen; ptr1 != end; *ptr1++ |= *ptr2++)
            ;

      /* else allocate space for the result vector and copy the source
         vector to it */
      else {
         *result = (bvect) (ptr1 = BALLOC);
         for (end = ptr1 + bvectlen; ptr1 != end; *ptr1++ = *ptr2++)
            ;
      }
   }
}

/*
 * binter - computes the intersection of two bvects
 */
void binter(bvect *result, bvect vec)
```



```
{
   unsigned int *ptr1, *ptr2, *end;

   /* if the result vector and the source vector has been allocated */
   if ((ptr1 = (unsigned int *) *result) && (ptr2 = (unsigned int *) vec))

      /* union in all of the words of the source vector */
      for (end = ptr1 + bvectlen; ptr1 != end; *ptr1++ &= *ptr2++)
         ;

   /* else clear the result vector if it has been allocated */
   else if (*result)
      bclear(*result);
}

/*
 * bin - indicate if a basic block is in a bit vector
 */
int bin(bvect ptr, unsigned int r)
{
   /* The bvect has to be allocated and the appropriate word in the
      vector has to have the appropriate bit set. */
   return (ptr && ptr[r >> LOG2_INT] & (((unsigned) 1) << (r & INT_REM)));
}

/*
 * bcpy - copy a bvect list to another one
 */
void bcpy(bvect *dst, bvect src)
{
   register unsigned int *ptr1, *ptr2, *end;

   /* If the src vector is allocated */
   if (src) {

      /* if the dst is not allocated, then we have to allocate
         space for it */
      if (!*dst)
         *dst = (bvect) BALLOC;

      /* copy the src vector to the dst vector */
      end = (ptr1 = (unsigned int *) *dst) + bvectlen;
      ptr2 = (unsigned int *) src;
      while (ptr1 != end)
         *ptr1++ = *ptr2++;
   }

   /* if the dst is allocated, then just clear it */
   else if (*dst)
      bclear(*dst);
}

/*
 * bequal - compare two bvect lists to see if all bits are equal
 */
int bequal(bvect a, bvect b)
{
   unsigned int *ptr1, *ptr2, *end;

   /* if both bvect have been allocated */
   if (a && b) {

      /* compare each word, if any differ then the two are not equal */
      end = (ptr1 = (unsigned int *) a) + bvectlen;
      ptr2 = (unsigned int *) b;
      while (ptr1 != end)
         if (*ptr1++ != *ptr2++)
            return FALSE ;
      return TRUE ;
```

```
   }

   /* if one exists, check that it is empty.  If neither
      exist, then they are equal. */
   if (a)
      end = (ptr1 = (unsigned int *) a) + bvectlen;
   else if (b)
      end = (ptr1 = (unsigned int *) b) + bvectlen;
   else
      return TRUE;
   while (ptr1 != end)
      if (*ptr1++)
         return FALSE ;
   return TRUE ;
}

/*
 * ball - initialize a bvect list containing all basic blocks
 */
bvect ball()
{
   unsigned int *ptr, *end;

   /* allocate the vector */
   end = (ptr = BALLOC) + bvectlen;

   /* set all bits of each word of the vector */
   do {
      *(--end) = -1;
   }
   while (ptr != end);

   /* return the vector */
   return((bvect) ptr);
}

/*
 * bclear - reinitialize a bvect list to contain no blocks
 */
void bclear(bvect sptr)
{
   unsigned int *ptr, *end;

   /* clear all bits in each word of the vector */
   end = (ptr = (unsigned int *) sptr) + bvectlen;
   do {
      *(--end) = 0;
   }
   while (ptr != end);
}

/*
 * bcnt - determine the number of bits set in a bvect list
 */
int bcnt(bvect sptr)
{
   int i, cnt;
   unsigned int *ptr, *end;

   if (!sptr)
      return 0;
   cnt = 0;
   for (end = (ptr = (unsigned int *) sptr) + bvectlen; ptr != end; ptr++)
      for (i = 0; i < sizeof(unsigned int)*8; i++)
         if (*ptr & (1 << i))
            cnt++;
   return cnt;
}
```



```
/*
 * bnone - initialize a bvect list containing no basic blocks
 */
bvect bnone()
{
   bvect sptr;
   void bclear(bvect);

   /* initialize the space for the vector */
   sptr = BALLOC;

   /* clear out all bits in the vector */
   bclear(sptr);

   /* return the vector */
   return(sptr);
}

/*
 * bfree - free up a bvect list
 */
void bfree(bvect ptr)
{
   free(ptr);
}

/*
 * bdump - dump out a bvect list in a readable fashion
 */
void bdump(FILE *fout, bvect ptr)
{
   int i;
   int bin(bvect, unsigned int);
   extern int numblks;

   for (i = 1; i <= numblks; i++)
      if (bin(ptr, i))
         fprintf(fout, " %d", i);
}
```