Appendix B
AMPIRE Assembler

B.1. Assembly Code Format

The assembly code has the following format. Each part is optional and case-insensitive.

```
[label:]  [*][instruction]  [;comment]
```

The instruction set is shown in Table 3.1. Each register field is specified as R0 to R31. The immediate/offset field may be a decimal number, a hexdecimal number (with 'H' postfix), or a label. A number may also have the '#' prefix as shown in [Henn90], but that character is simply ignored by the assembler. By adding the '*' prefix to any instruction, a bad parity bit is generated for fault simulation.

B.2. Assembler Source Code

The C source code starts on the next page.

asm.c

```
 1:    /* AMPIRE Assembler */
 2:
 3:    /* INST_WIDTH must be less than or equal to 32 bits.  Even parity is used. */
 4:
 5:    #include <stdio.h>
 6:    #include <string.h>
 7:
 8:    #define OP_WIDTH        6       /* bits for opcode field */
 9:    #define REG_WIDTH       5       /* bits for register number */
10:    #define EXTRA_WIDTH     11      /* bits for "extra" field */
11:    #define ADDR_WIDTH      8       /* bits for address (for comment only) */
12:    #define ADDR_INC        4       /* amount of address increment */
13:
14:    #define IMM_WIDTH       REG_WIDTH + EXTRA_WIDTH
15:    #define OFFSET_WIDTH    2 * REG_WIDTH + IMM_WIDTH
16:    #define INST_WIDTH      OP_WIDTH + OFFSET_WIDTH
17:
18:    #define MAX_OPS         50      /* max number of opcodes */
19:    #define MAX_LINE        100     /* max number of characters per line */
20:    #define MAX_LABELS      50      /* max number of labels */
21:    #define MAX_LEN         20      /* max length of labels */
22:    #define SPACE           ' '
23:    #define EOL             '0
24:
25:    #define F_NONE          0       /* instruction format codes */
26:    #define F_ALU           1
27:    #define F_ALUI          2
28:    #define F_LHI           3
29:    #define F_JREG          4
30:    #define F_BRANCH        5
31:    #define F_OFFSET        6
32:    #define F_LOAD          7
33:    #define F_STORE         8
34:    #define F_DATA          9
35:
36:    #define T_ABS           0       /* absolute address type */
37:    #define T_REL           1       /* PC relative type */
38:
39:    char *opname[MAX_OPS];          /* opcode name database */
40:    int opnum[MAX_OPS];             /* opcode number database */
41:    int extra[MAX_OPS];             /* extra field database */
42:    char format[MAX_OPS];           /* instruction format codes */
43:    int opcount;                    /* number of opcodes */
44:
45:    char label[MAX_LABELS][MAX_LEN];        /* label name database */
46:    int addr[MAX_LABELS];           /* label address database */
47:    int label_count = 0;            /* number of labels */
48:
49:    int cur_addr;                   /* current address */
50:    int cur_line;                   /* current line in input file */
51:    char parity;                    /* used to create good/bad parity */
52:    FILE *infile, *outfile;
53:
54:    /* ============================================================ */
55:
56:    main (argc, argv)
57:      int argc;
58:      char *argv[];
59:    {
60:    extern char format[MAX_OPS];
61:    extern int label_count, cur_addr, cur_line;
62:    extern FILE *infile, *outfile;
63:
64:    char org_line[MAX_LINE], work_line[MAX_LINE];
65:    int pointer, opindex, rd, rs, rt, imm, offset, data;
66:
67:    if (argc != 2) {
68:            printf ("Usage: asm source_file   (output: inst.hex)0);
69:            exit (1);
70:    }
71:
```

```
 72:    define_opcodes ();                      /* define opcode database */
 73:
 74:    printf ("first pass ...0);
 75:    infile = fopen (argv[1], "r");          /* open input file */
 76:    if (infile == NULL) {
 77:            printf ("*** source file error ***0);
 78:            exit (1);
 79:    }
 80:
 81:    cur_line = 0;
 82:    cur_addr = 0;
 83:    while (get_line(org_line) != EOF) {     /* 1st pass -- scan for labels */
 84:            cur_line++;
 85:            strcpy (work_line, org_line);
 86:            filter_line (work_line);
 87:            pointer = 0;
 88:            opindex = get_opindex (work_line, &pointer);
 89:
 90:            if (opindex == -2) {
 91:                    add_label (work_line);  /* add to label database */
 92:                    opindex = get_opindex (work_line, &pointer);
 93:            }
 94:
 95:            if (opindex >= 0)
 96:                    cur_addr += ADDR_INC;   /* real instruction */
 97:    }   /* while */
 98:    fclose (infile);
 99:
100:    printf ("%d out of %d label database slots are used.0, label_count,
101:            MAX_LABELS);
102:    printf ("second pass ...0);
103:
104:    infile = fopen (argv[1], "r");          /* open input file */
105:    if (infile == NULL) {
106:            printf ("*** source file error ***0);
107:            exit (1);
108:    }
109:
110:    outfile = fopen ("inst.hex", "w");      /* open output file */
111:    if (outfile == NULL) {
112:            printf ("*** output file error ***0);
113:            exit (1);
114:    }
115:
116:    cur_line = 0;
117:    cur_addr = 0;
118:    while (get_line(org_line) != EOF) {     /* 2nd pass -- assemble codes */
119:            cur_line++;
120:            strcpy (work_line, org_line);
121:            filter_line (work_line);
122:            pointer = 0;
123:            opindex = get_opindex (work_line, &pointer);
124:
125:            if (opindex == -2)              /* skip labels */
126:                    opindex = get_opindex (work_line, &pointer);
127:
128:            if (opindex >= 0) {             /* different instruction formats */
129:                rs = 0;
130:                rt = 0;
131:                rd = 0;
132:                imm = 0;
133:                offset = 0;
134:
135:                switch (format[opindex]) {
136:                    case F_NONE: {
137:                            break;
138:                    }
139:                    case F_ALU: {
140:                            rd = get_reg (work_line, &pointer);
141:                            rs = get_reg (work_line, &pointer);
142:                            rt = get_reg (work_line, &pointer);
```

```
143:                            break;
144:                    }
145:                    case F_ALUI: {
146:                            rt = get_reg (work_line, &pointer);
147:                            rs = get_reg (work_line, &pointer);
148:                            imm = get_num (work_line, &pointer, IMM_WIDTH,
149:                                    T_ABS);
150:                            break;
151:                    }
152:                    case F_LHI: {
153:                            rt = get_reg (work_line, &pointer);
154:                            imm = get_num (work_line, &pointer, IMM_WIDTH,
155:                                    T_ABS);
156:                            break;
157:                    }
158:                    case F_JREG: {
159:                            rs = get_reg (work_line, &pointer);
160:                            break;
161:                    }
162:                    case F_BRANCH: {
163:                            rs = get_reg (work_line, &pointer);
164:                            imm = get_num (work_line, &pointer, IMM_WIDTH,
165:                                    T_REL);
166:                            break;
167:                    }
168:                    case F_OFFSET: {
169:                            offset = get_num (work_line, &pointer, OFFSET_WIDTH,
170:                                    T_REL);
171:                            break;
172:                    }
173:                    case F_LOAD: {
174:                            rt = get_reg (work_line, &pointer);
175:                            imm = get_num (work_line, &pointer, IMM_WIDTH,
176:                                    T_ABS);
177:                            rs = get_reg (work_line, &pointer);
178:                            break;
179:                    }
180:                    case F_STORE: {
181:                            imm = get_num (work_line, &pointer, IMM_WIDTH,
182:                                    T_ABS);
183:                            rs = get_reg (work_line, &pointer);
184:                            rt = get_reg (work_line, &pointer);
185:                            break;
186:                    }
187:                    case F_DATA: {
188:                            data = get_num (work_line, &pointer, INST_WIDTH,
189:                                    T_ABS);
190:                            break;
191:                    }
192:                }   /* switch */
193:
194:                check_line (work_line, pointer);
195:            }   /* if */
196:
197:            print_line (org_line, opindex, rd, rs, rt, imm, offset, data);
198:    }   /* while */
199:
200:    fclose (outfile);
201:    fclose (infile);
202:    printf ("199d Words, or %d Bytes0, cur_addr/ADDR_INC, cur_addr);
203:    }   /* main */
204:
205:    /* ============================================================ */
206:
207:    get_line (line)         /* get one line from file, return length or EOF */
208:      char line[];
209:    {
210:    extern int cur_line;
211:    extern FILE *infile;
212:
213:    int index, letter;
```

```
214:
215:    for (index=0; (letter=getc(infile)) != EOF && letter != EOL; index++)
216:            line[index] = letter;
217:    line[index] = NULL;
218:
219:    if (index >= MAX_LINE) {
220:            printf ("(%d) Line Too Long:200s0, cur_line, line);
221:            exit (1);
222:    }
223:
224:    if (letter == EOF && index == 0)
225:            return (EOF);
226:    else
227:            return (index);
228:    }   /* function get_line */
229:
230:    /* ============================================================ */
231:
232:    /* Convert unneeded characters to spaces, uppercase to lowercase. */
233:
234:    filter_line (line)
235:      char line[];
236:    {
237:    int index = 0;
238:    char letter;
239:
240:    while ((letter=line[index]) != NULL) {
241:            if (letter=='' || letter==',' || letter=='(' || letter==')' ||
242:                letter=='#')
243:                    line[index] = SPACE;
244:            if (letter >= 'A' && letter <= 'Z')
245:                    line[index] = letter - 'A' + 'a';
246:            index++;
247:    }
248:    }   /* function filter_line */
249:
250:    /* ============================================================ */
251:
252:    /* If valid, return opcode index with pointer after the opcode.
253:     * If comment or blank line, return -1.  If label, return -2.
254:     */
255:
256:    get_opindex (line, index)
257:      char line[];
258:      int *index;
259:    {
260:    extern char *opname[MAX_OPS];
261:    extern int opcount, cur_line;
262:    extern char parity;
263:
264:    int wpos = 0, opindex;
265:    char word[MAX_LINE];
266:
267:    while (line[*index] == SPACE)           /* find first non-space */
268:            (*index)++;
269:
270:    if (line[*index] == ';' || line[*index] == NULL)        /* comment/blank */
271:            return (-1);
272:
273:    parity = 0;
274:    if (line[*index] == '*') {              /* create bad parity */
275:            parity = 1;
276:            (*index)++;
277:    }
278:
279:    while (line[*index] != SPACE && line[*index] != NULL)   /* get 1st word */
280:            word[wpos++] = line[(*index)++];
281:    word[wpos] = NULL;
282:
283:    if (word[wpos-1] == ':')                /* label */
284:            return  (-2);
```

```
285:
286:    for (opindex=0; opindex<opcount; opindex++)     /* find matching opcode */
287:            if (strcmp(word, opname[opindex]) == 0)
288:                    break;
289:
290:    if (opindex == opcount) {                               /* no match */
291:            printf ("(%d) Invalid Opcode:201s0, cur_line, line);
292:            exit (1);
293:    }
294:
295:    return (opindex);
296:    }   /* function get_opindex */
297:
298:    /* ============================================================ */
299:
300:    add_label (line)               /* add label to database */
301:      char line[];
302:    {
303:    extern char label[MAX_LABELS][MAX_LEN];
304:    extern int addr[MAX_LABELS];
305:    extern int label_count, cur_line;
306:
307:    int wpos = 0, index, loop;
308:    char word[MAX_LINE];
309:
310:    if (label_count >= MAX_LABELS) {
311:            printf ("Label Database Full, %d Entries0, label_count);
312:            exit (1);
313:    }
314:
315:    index = 0;
316:    while (line[index] == SPACE)            /* find first non-space */
317:            index++;
318:
319:    while (line[index] != ':')              /* get label */
320:            word[wpos++] = line[index++];
321:    word[wpos] = NULL;
322:
323:    if (wpos >= MAX_LEN) {
324:            printf ("(%d) Label Too Long:201s0, cur_line, line);
325:            exit (1);
326:    }
327:
328:    for (loop=0; loop<label_count; loop++)  /* see if already defined */
329:            if (strcmp(word, label[loop]) == 0) {
330:                    printf ("(%d) Label Already Defined:201s0, cur_line, line);
331:                    exit (1);
332:            }
333:
334:    strcpy (label[label_count], word);      /* add it */
335:    addr[label_count++] = cur_addr;
336:    }   /* function add_label */
337:
338:    /* ============================================================ */
339:
340:    get_reg (line, index)           /* return register number */
341:      char line[];
342:      int *index;
343:    {
344:    extern int cur_line;
345:
346:    int wpos = 0, reg_num;
347:    char word[MAX_LINE];
348:
349:    while (line[*index] == SPACE)           /* skip spaces */
350:            (*index)++;
351:
352:    while (line[*index] != SPACE && line[*index] != NULL)   /* get one word */
353:            word[wpos++] = line[(*index)++];
354:    word[wpos] = NULL;
355:
```

```
356:    if (word[0] != 'r') {
357:            printf ("(%d) Register must start with 'r':202s0, cur_line, line);
358:            exit (1);
359:    }
360:
361:    if (sscanf (&word[1], "%d", &reg_num) < 1) {
362:            printf ("(%d) Invalid Register Format:202s0, cur_line, line);
363:            exit (1);
364:    }
365:
366:    if (reg_num < 0 || reg_num > (power2(REG_WIDTH)-1)) {
367:            printf ("(%d) Register Number Out of Range:202s0, cur_line, line);
368:            exit (1);
369:    }
370:
371:    return (reg_num);
372:    }   /* function get_reg */
373:
374:    /* ============================================================ */
375:
376:    get_num (line, index, maxbits, type)    /* return number */
377:      char line[];
378:      int *index, maxbits, type;
379:    {
380:    extern int cur_line;
381:
382:    int wpos = 0, num;
383:    char word[MAX_LINE];
384:
385:    while (line[*index] == SPACE)           /* skip spaces */
386:            (*index)++;
387:
388:    while (line[*index] != SPACE && line[*index] != NULL)   /* get one word */
389:            word[wpos++] = line[(*index)++];
390:    word[wpos] = NULL;
391:
392:    num = find_label (word);                /* find address, -1 if not found */
393:    if (num >= 0) {
394:            if (type == T_REL)
395:                    num = num - (cur_addr + ADDR_INC);      /* PC relative */
396:    }
397:    else
398:            if (line[(*index)-1] == 'h') {
399:                    if (sscanf (word, "%x", &num) < 1) {
400:                            printf ("(%d) Invalid Hexadecimal Format:202s0,
401:                                    cur_line, line);
402:                            exit (1);
403:                    }
404:            }
405:            else
406:                    if (sscanf (word, "%d", &num) < 1) {
407:                            printf ("(%d) Invalid Decimal Format:202s0,
408:                                    cur_line, line);
409:                            exit (1);
410:                    }
411:
412:    if (num < -(power2(maxbits-1)) || num > (power2(maxbits-1)-1)) {
413:            printf ("(%d) %d-bit 2's Complement Number Out of Range:202s0,
414:                    cur_line, maxbits, line);
415:            exit (1);
416:    }
417:
418:    if (num < 0)            /* 2's complement form */
419:            num = power2 (maxbits) + num;
420:    return (num);
421:    }   /* function get_num */
422:
423:    /* ============================================================ */
424:
425:    find_label (word)       /* return address, or -1 if not found */
426:      char word[];
```

```
427:    {
428:    extern char label[MAX_LABELS][MAX_LEN];
429:    extern int addr[MAX_LABELS];
430:    extern int label_count;
431:
432:    int loop;
433:
434:    for (loop=0; loop<label_count; loop++)          /* search */
435:            if (strcmp(word, label[loop]) == 0)
436:                    return (addr[loop]);
437:
438:    return (-1);    /* not found */
439:    }   /* function find_label */
440:
441:    /* ============================================================ */
442:
443:    check_line (line, index)        /* check the rest of the line */
444:      char line[];
445:      int index;
446:    {
447:    extern int cur_line;
448:
449:    while (line[index] == SPACE)            /* skip spaces */
450:            index++;
451:
452:    if (line[index] != ';' && line[index] != NULL) {
453:            printf ("(%d) Too Many Operands/Comments without Semicolon:203s0,
454:                    cur_line, line);
455:            exit (1);
456:            }
457:    }   /* function check_line */
458:
459:    /* ============================================================ */
460:
461:    /* To generate correct even parity, 'parity' should be 0 when this routine
462:     * is called.  The bad parity flag is set in the get_opindex routine.
463:     */
464:
465:    print_line (org_line, opindex, rd, rs, rt, imm, offset, data)
466:      char org_line[];
467:      int opindex, rd, rs, rt, imm, offset, data;
468:    {
469:    extern int opnum[MAX_OPS];
470:    extern int extra[MAX_OPS];
471:    extern char format[MAX_OPS];
472:    extern int cur_addr;
473:    extern char parity;
474:    extern FILE *outfile;
475:
476:    unsigned int inst;
477:    int loop, inst_digits, addr_digits;
478:
479:    inst_digits = (INST_WIDTH + 1) / 4;     /* number of hex digits */
480:    if ((INST_WIDTH + 1) % 4 != 0)
481:            inst_digits++;
482:    addr_digits = ADDR_WIDTH / 4;
483:    if (ADDR_WIDTH % 4 != 0)
484:            addr_digits++;
485:
486:    if (opindex >= 0) {
487:        if (format[opindex] == F_DATA)
488:            inst = data;
489:        else
490:            inst = opnum[opindex] * power2 (OFFSET_WIDTH) +
491:                    rs * power2 (REG_WIDTH + IMM_WIDTH) +
492:                    rt * power2 (IMM_WIDTH) +
493:                    rd * power2 (EXTRA_WIDTH) +
494:                    extra[opindex] + imm + offset;
495:
496:        for (loop=0; loop<INST_WIDTH; loop++)       /* find parity bit */
497:            if (inst & power2(loop))
```

```
498:                    parity ^= 1;                    /* XOR */
499:
500:        if (INST_WIDTH < 32) {
501:            inst += parity * power2 (INST_WIDTH);
502:            fprintf (outfile, "%.*X", inst_digits, inst);
503:        }
504:        else {
505:            fprintf (outfile, "%d", parity);
506:            fprintf (outfile, "%.*X", inst_digits-1, inst);
507:        }
508:
509:        fprintf (outfile, " // ");
510:        fprintf (outfile, "%.*X", addr_digits, cur_addr);
511:        cur_addr += ADDR_INC;
512:    }
513:    else {                                          /* no instruction */
514:            for (loop=0; loop<inst_digits; loop++)
515:                    putc (SPACE, outfile);
516:            fprintf (outfile, " // ");
517:            for (loop=0; loop<addr_digits; loop++)
518:                    putc (SPACE, outfile);
519:    }
520:
521:    fprintf (outfile, "%s0, org_line);
522:    }   /* function print_line */
523:
524:    /* ============================================================ */
525:
526:    power2 (exp)            /* return power of 2 */
527:      int exp;
528:    {
529:    int result = 1, loop;
530:
531:    for (loop=0; loop<exp; loop++)
532:            result = result * 2;
533:
534:    return (result);
535:    }   /* function power2 */
536:
537:    /* ============================================================ */
538:
539:    /* instruction format code:
540:     *      NONE    opcode
541:     *      ALU     opcode rd, rs, rt
542:     *      ALUI    opcode rt, rs, imm
543:     *      LHI     opcode rt, imm
544:     *      JREG    opcode rs
545:     *      BRANCH  opcode rs, imm
546:     *      OFFSET  opcode offset
547:     *      LOAD    opcode rt, imm(rs)
548:     *      STORE   opcode imm(rs), rt
549:     *      DATA    opcode data_word
550:     */
551:
552:    define_opcodes ()
553:    {
554:    extern char *opname[MAX_OPS];
555:    extern int opnum[MAX_OPS];
556:    extern int extra[MAX_OPS];
557:    extern char format[MAX_OPS];
558:    extern int opcount;
559:
560:    int x = 0;
561:
562:    opname[x]="j";          opnum[x]=2;     extra[x]=0;     format[x++]=F_OFFSET;
563:    opname[x]="jal";        opnum[x]=3;     extra[x]=0;     format[x++]=F_OFFSET;
564:    opname[x]="beqz";       opnum[x]=4;     extra[x]=0;     format[x++]=F_BRANCH;
565:    opname[x]="bnez";       opnum[x]=5;     extra[x]=0;     format[x++]=F_BRANCH;
566:    opname[x]="addui";      opnum[x]=9;     extra[x]=0;     format[x++]=F_ALUI;
567:    opname[x]="subui";      opnum[x]=11;    extra[x]=0;     format[x++]=F_ALUI;
568:    opname[x]="andi";       opnum[x]=12;    extra[x]=0;     format[x++]=F_ALUI;
```

```
569:    opname[x]="ori";        opnum[x]=13;    extra[x]=0;     format[x++]=F_ALUI;
570:    opname[x]="xori";       opnum[x]=14;    extra[x]=0;     format[x++]=F_ALUI;
571:    opname[x]="lhi";        opnum[x]=15;    extra[x]=0;     format[x++]=F_LHI;
572:    opname[x]="trap";       opnum[x]=17;    extra[x]=0;     format[x++]=F_OFFSET;
573:    opname[x]="jr";         opnum[x]=18;    extra[x]=0;     format[x++]=F_JREG;
574:    opname[x]="jalr";       opnum[x]=19;    extra[x]=0;     format[x++]=F_JREG;
575:    opname[x]="slli";       opnum[x]=20;    extra[x]=0;     format[x++]=F_ALUI;
576:    opname[x]="srli";       opnum[x]=22;    extra[x]=0;     format[x++]=F_ALUI;
577:    opname[x]="srai";       opnum[x]=23;    extra[x]=0;     format[x++]=F_ALUI;
578:    opname[x]="seqi";       opnum[x]=24;    extra[x]=0;     format[x++]=F_ALUI;
579:    opname[x]="snei";       opnum[x]=25;    extra[x]=0;     format[x++]=F_ALUI;
580:    opname[x]="slti";       opnum[x]=26;    extra[x]=0;     format[x++]=F_ALUI;
581:    opname[x]="sgti";       opnum[x]=27;    extra[x]=0;     format[x++]=F_ALUI;
582:    opname[x]="slei";       opnum[x]=28;    extra[x]=0;     format[x++]=F_ALUI;
583:    opname[x]="sgei";       opnum[x]=29;    extra[x]=0;     format[x++]=F_ALUI;
584:    opname[x]="lw";         opnum[x]=35;    extra[x]=0;     format[x++]=F_LOAD;
585:    opname[x]="sw";         opnum[x]=43;    extra[x]=0;     format[x++]=F_STORE;
586:    opname[x]="adduif";     opnum[x]=49;    extra[x]=0;     format[x++]=F_ALUI;
587:    opname[x]="jrf";        opnum[x]=50;    extra[x]=0;     format[x++]=F_JREG;
588:    opname[x]="swf";        opnum[x]=51;    extra[x]=0;     format[x++]=F_STORE;
589:
590:    opname[x]="nop";        opnum[x]=0;     extra[x]=0;     format[x++]=F_NONE;
591:    opname[x]="sll";        opnum[x]=0;     extra[x]=4;     format[x++]=F_ALU;
592:    opname[x]="srl";        opnum[x]=0;     extra[x]=6;     format[x++]=F_ALU;
593:    opname[x]="sra";        opnum[x]=0;     extra[x]=7;     format[x++]=F_ALU;
594:    opname[x]="adduf";      opnum[x]=0;     extra[x]=25;    format[x++]=F_ALU;
595:    opname[x]="addu";       opnum[x]=0;     extra[x]=33;    format[x++]=F_ALU;
596:    opname[x]="subu";       opnum[x]=0;     extra[x]=35;    format[x++]=F_ALU;
597:    opname[x]="and";        opnum[x]=0;     extra[x]=36;    format[x++]=F_ALU;
598:    opname[x]="or";         opnum[x]=0;     extra[x]=37;    format[x++]=F_ALU;
599:    opname[x]="xor";        opnum[x]=0;     extra[x]=38;    format[x++]=F_ALU;
600:    opname[x]="seq";        opnum[x]=0;     extra[x]=40;    format[x++]=F_ALU;
601:    opname[x]="sne";        opnum[x]=0;     extra[x]=41;    format[x++]=F_ALU;
602:    opname[x]="slt";        opnum[x]=0;     extra[x]=42;    format[x++]=F_ALU;
603:    opname[x]="sgt";        opnum[x]=0;     extra[x]=43;    format[x++]=F_ALU;
604:    opname[x]="sle";        opnum[x]=0;     extra[x]=44;    format[x++]=F_ALU;
605:    opname[x]="sge";        opnum[x]=0;     extra[x]=45;    format[x++]=F_ALU;
606:
607:    opname[x]=".word";      opnum[x]=0;     extra[x]=0;     format[x++]=F_DATA;
608:
609:    opcount = x;
610:    printf ("%d out of %d opcode database slots are used.0, x, MAX_OPS);
611:    }   /* function define_data */
```