

```

1  long_t compute_alu(alu_t op, long_t argA, long_t argB)
2  {
3      switch (op) {
4          case A_ADD:
5              return argB + argA;
6          case A_SUB:
7              return argB - argA;
8          case A_AND:
9              return argB & argA;
10         case A_XOR:
11             return argB ^ argA;
12         case A_NONE: /* act as default */
13             default:
14                 return 0;
15     }
16 }
17
18 /*
19  * compute_cc: modify condition codes according to operations
20  * args
21  *   op: operations (A_ADD, A_SUB, A_AND, A_XOR)
22  *   argA: the first argument
23  *   argB: the second argument
24  *   val: the result of operation on argA and argB
25  *
26  * return
27  *   PACK_CC: the final condition codes
28  */
29 cc_t compute_cc(alu_t op, long_t argA, long_t argB, long_t val)
30 {
31     bool_t zero = FALSE;
32     bool_t sign = FALSE;
33     bool_t ovf = FALSE;
34
35     zero = val == 0;
36     sign = val < 0;
37     switch (op) {
38         case A_ADD:
39             if (argA > 0 && argB > 0)
40                 ovf = val < 0;
41             else if (argA < 0 && argB < 0)
42                 ovf = val >= 0;
43             else
44                 ovf = 0;

```

```

45         break;
46     case A_SUB:
47         if (-argA > 0 && argB > 0)
48             ovf = val < 0;
49         else if (-argA < 0 && argB < 0)
50             ovf = val >= 0;
51         else
52             ovf = 0;
53         break;
54     case A_AND: case A_XOR:
55         break;
56     case A_NONE:
57         zero = FALSE;
58         sign = FALSE;
59         break;
60     }
61     return PACK_CC(zero,sign,ovf);
62 }
63
64 /*
65  * cond_doit: whether do (mov or jmp) it?
66  * args
67  *     PACK_CC: the current condition codes
68  *     cond: conditions (C_YES, C_LE, C_L, C_E, C_NE, C_GE, C_G)
69  *
70  * return
71  *     TRUE: do it
72  *     FALSE: not do it
73  */
74 bool_t cond_doit(cc_t cc, cond_t cond)
75 {
76     switch (cond) {
77     case C_YES:
78         return TRUE;
79     case C_LE:
80         return (GET_SF(cc) ^ GET_OF(cc)) | GET_ZF(cc);
81     case C_L:
82         return GET_SF(cc) ^ GET_OF(cc);
83         break;
84     case C_E:
85         return GET_ZF(cc);
86     case C_NE:
87         return !GET_ZF(cc);
88     case C_GE:

```

```

89         return !(GET_SF(cc) ^ GET_OF(cc));
90     case C_G:
91         return !(GET_SF(cc) ^ GET_OF(cc)) & !GET_ZF(cc);
92     default:
93         return FALSE;
94     }
95 }
96
97 /*
98  * nexti: execute single instruction and return status.
99  * args
100  *     sim: the y86 image with PC, register and memory
101  *
102  * return
103  *     STAT_AOK: continue
104  *     STAT_HLT: halt
105  *     STAT_ADR: invalid instruction address
106  *     STAT_INS: invalid instruction, register id, data address, stack
address, ...
107  */
108 stat_t nexti(y86sim_t *sim)
109 {
110     byte_t codefun = 0;
111     itype_t icode;
112     alu_t ifun;
113     long_t next_pc = sim->pc;
114
115     regid_t regA = REG_NONE, regB = REG_NONE;
116     long_t imm;
117
118     /* get code and function (1 byte) */
119     if (!get_byte_val(sim->m, next_pc, &codefun)) {
120         err_print("PC = 0x%x, Invalid instruction address", sim->pc);
121         return STAT_ADR;
122     }
123     icode = GET_ICODE(codefun);
124     ifun = GET_FUN(codefun);
125     next_pc++;
126
127     /* get registers if needed (1 byte) */
128     switch (icode) {
129         case I_RRMOVL: case I_IRMOVL: case I_RMMOVL: case I_MRMOVL: case I_ALU:
case I_POPL: case I_PUSHL:
130             if (!get_byte_val(sim->m, next_pc, &codefun)) {

```

```

131         err_print("PC = 0x%x, Invalid instruction address", sim->pc);
132         return STAT_ADR;
133     }
134     regA = GET_REGA(codefun);
135     regB = GET_REGB(codefun);
136     next_pc++;
137     break;
138     default:
139         break;
140 }
141
142 /* get immediate if needed (4 bytes) */
143 switch (icode) {
144     case I_IRMOVL: case I_RRMOVL: case I_MRMOVL: case I_JMP: case I_CALL:
145         if (!get_long_val(sim->m, next_pc, &imm)) {
146             err_print("PC = 0x%x, Invalid instruction address", sim->pc);
147             return STAT_ADR;
148         }
149         next_pc += 0x4;
150         break;
151     default:
152         break;
153 }
154
155 /* execute the instruction */
156 switch (icode) {
157     case I_HALT: /* 0:0 */
158         return STAT_HLT;
159         break;
160
161     case I_NOP: /* 1:0 */
162         sim->pc = next_pc;
163         break;
164
165     case I_RRMOVL: /* 2:x regA:regB */
166         sim->pc = next_pc;
167         if (!cond_doit(sim->cc, ifun))
168             break;
169         set_reg_val(sim->r, regB, get_reg_val(sim->r, regA));
170         break;
171
172     case I_IRMOVL: /* 3:0 F:regB imm */
173         if (regA != 0xF) {
174             err_print("PC = 0x%x, Invalid instruction address", sim->pc);

```

```

175         return STAT_ADR;
176     }
177     set_reg_val(sim->r,regB,imm);
178     sim->pc = next_pc;
179     break;
180
181     case I_RMMOVL: /* 4:0 regA:regB imm */
182     {
183         long_t regB_cont = get_reg_val(sim->r,regB);
184         if(!set_long_val(sim->m, imm+regB_cont, get_reg_val(sim->r,regA))) {
185             err_print("PC = 0x%x, Invalid data address 0x%x",
186                     sim->pc,regB_cont+imm);
187             return STAT_ADR;
188         }
189         sim->pc = next_pc;
190         break;
191     }
192
193     case I_MRMOVL: /* 5:0 regB:regA imm */
194     {
195         long_t cont;
196         if(!get_long_val(sim->m,get_reg_val(sim->r,regB)+imm,&cont)) {
197             err_print("PC = 0x%x, Invalid data address 0x%x",
198                     sim->pc,get_reg_val(sim->r,regB)+imm);
199             return STAT_ADR;
200         }
201         set_reg_val(sim->r,regA,cont);
202         sim->pc = next_pc;
203         break;
204     }
205
206     case I_ALU: /* 6:x regA:regB */
207     {
208         long_t regA_cont = get_reg_val(sim->r,regA),
209             regB_cont = get_reg_val(sim->r,regB);
210         long_t res = compute_alu(ifun,regA_cont,regB_cont);
211         sim->cc = compute_cc(ifun,regA_cont,regB_cont,res);
212         set_reg_val(sim->r,regB,res);
213         sim->pc = next_pc;
214         break;
215     }
216
217     case I_JMP: /* 7:x imm */
218         if(cond_doit(sim->cc,ifun))

```

```

219         sim->pc = imm;
220     else
221         sim->pc = next_pc;
222     break;
223
224     case I_CALL: /* 8:x imm */
225         if(imm < 0 || imm > sim->m->len) {
226             err_print("WTFatCALL %x",sim->pc);
227             return STAT_ADR;
228         }
229         else {
230             sim->pc = imm;
231             if(!push_long_val(sim,next_pc)) {
232                 err_print("PC = 0x%x, Invalid stack address 0x%x",
233                     sim->pc,get_reg_val(sim->r,REG_ESP));
234                 return STAT_ADR;
235             }
236         }
237     break;
238
239     case I_RET: /* 9:0 */
240         if(!pop_long_val(sim,&(sim->pc))) {
241             err_print("PC = 0x%x, Invalid stack address 0x%x",
242                 sim->pc,get_reg_val(sim->r,REG_ESP));
243             return STAT_ADR;
244         }
245     break;
246
247     case I_PUSHL: /* A:0 regA:F */
248         if(!push_long_val(sim,get_reg_val(sim->r,regA))) {
249             err_print("PC = 0x%x, Invalid stack address 0x%x",
250                 sim->pc,get_reg_val(sim->r,REG_ESP));
251             return STAT_ADR;
252         }
253     sim->pc = next_pc;
254     break;
255
256     case I_POPL: /* B:0 regA:F */
257     {
258         sim->pc = next_pc;
259         if(regB != 0xF) {
260             err_print("PC = 0x%x, Invalid instruction %.2x", sim->pc, codefun);
261             return STAT_INS;
262         }

```

```

263     long_t val;
264     if(!pop_long_val(sim,&val)) {
265         err_print("PC = 0x%x, Invalid stack address 0x%x",
266                 sim->pc, get_reg_val(sim->r, REG_ESP));
267         return STAT_ADR;
268     }
269     set_reg_val(sim->r, regA, val);
270     break;
271 }
272
273 default:
274     err_print("PC = 0x%x, Invalid instruction %.2x", sim->pc, codefun);
275     return STAT_INS;
276 }
277
278 return STAT_AOK;
279 }
280
281 void usage(char *pname)
282 {
283     printf("Usage: %s file.bin [max_steps]\n", pname);
284     exit(0);
285 }
286
287 int main(int argc, char *argv[])
288 {
289     FILE *binfile;
290     int max_steps = MAX_STEP;
291     y86sim_t *sim;
292     mem_t *saver, *savem;
293     int step = 0;
294     stat_t e = STAT_AOK;
295
296     if (argc < 2 || argc > 3)
297         usage(argv[0]);
298
299     /* set max steps */
300     if (argc > 2)
301         max_steps = atoi(argv[2]);
302
303     /* load binary file to memory */
304     if (strcmp(argv[1]+(strlen(argv[1])-4), ".bin"))
305         usage(argv[0]); /* only support *.bin file */
306

```

```

307     binfile = fopen(argv[1], "rb");
308     if (!binfile) {
309         err_print("Can't open binary file '%s'", argv[1]);
310         exit(1);
311     }
312
313     sim = new_y86sim(MEM_SIZE);
314     if (load_binfile(sim->m, binfile) < 0) {
315         err_print("Failed to load binary file '%s'", argv[1]);
316         free_y86sim(sim);
317         exit(1);
318     }
319     fclose(binfile);
320
321     /* save initial register and memory stat */
322     saver = dup_reg(sim->r);
323     savem = dup_mem(sim->m);
324
325     /* execute binary code step-by-step */
326     for (step = 0; step < max_steps && e == STAT_AOK; step++)
327         e = nexti(sim);
328
329     /* print final stat of y86sim */
330     printf("Stopped in %d steps at PC = 0x%x. Status '%s', CC %s\n",
331           step, sim->pc, stat_name(e), cc_name(sim->cc));
332
333     printf("Changes to registers:\n");
334     diff_reg(saver, sim->r, stdout);
335
336     printf("\nChanges to memory:\n");
337     diff_mem(savem, sim->m, stdout);
338
339     free_y86sim(sim);
340     free_reg(saver);
341     free_mem(savem);
342
343     return 0;
344 }

```