Block of ARCompact™ Instructions

```
 ....
0x00000848:
 [0x00000848] ext      r2,r9
 [0x0000084c] xor      r3,r12,r2
 [0x00000850] and      r3,r3,0xf
 [0x00000854] asl      r3,r3,0x3
 [0x00000858] and      r2,r2,0x7
 [0x0000085c] or       r3,r3,r2
 [0x00000860] asl      r4,r3,0x8
 [0x00000864] brcc.d   r10,r13,0x2c
 [0x00000868] or       r4,r4,r3
 ....
```

JIT Translated Block with Performance Model

```
extern CpuState cpu;              // global processor state
void BLK_0x00000848(void) {                                          1
  cpu.r[2] = (uint16_t)(cpu.r[9]);
  pipeline(0,cpu.avail[9],&(cpu.avail[2]),0x00000848,1,0);
  cpu.r[3] = cpu.r[12] ^ cpu.r[2];                                   2
  pipeline(cpu.avail[12],cpu.avail[2],&(cpu.avail[3]),0x0000084c,1,0);
  cpu.r[3] = cpu.r[3] & (uint32_t)15;
  pipeline(cpu.avail[3],0,&(cpu.avail[3]),0x00000850,1,0);
  cpu.r[3] = cpu.r[3] << ((sint8_t)3 & 0x1f);
  pipeline(cpu.avail[3],0,&(cpu.avail[3]),0x00000854,1,0);
  cpu.r[2] = cpu.r[2] & (uint32_t)7;
  pipeline(cpu.avail[2],0,&(cpu.avail[2]),0x00000858,1,0);
  cpu.r[3] = cpu.r[3] | cpu.r[2];
  pipeline(cpu.avail[3],cpu.avail[2],&(cpu.avail[3]),0x0000085c,1,0); 3
  cpu.r[4] = cpu.r[3] << ((sint8_t)8 & 0x1f);
  pipeline(cpu.avail[3],0,&(cpu.avail[4]),0x00000860,1,0);
  // compare and branch instruction with delay slot
  pipeline(cpu.avail[10],cpu.avail[13],&(ignore),0x00000864,1,0);
  if (cpu.r[10] >= cpu.r[13]) {
    cpu.pl[FE] = cpu.pl[ME] - 1; // branch penalty                    4
    fetch(0x0000086c);           // speculative fetch due to branch pred.
    cpu.auxr[BTA] = 0x00000890;  // set BTA register
    cpu.D = 1;                   // set delay slot bit
  } else {
    cpu.pc = 0x0000086c;                                             5
  }
  cpu.r[4] = cpu.r[4] | cpu.r[3];// delay slot instruction
  pipeline(cpu.avail[4],cpu.avail[3],&(cpu.avail[4]),0x00000868,1,0);
  if (cpu.D) {                   // branch was taken
    cpu.D = 0;                   // clear delay slot bit
    cpu.pc = cpu.auxr[BTA];      // set PC
  }
  cpu.cycles = cpu.pl[WB];       // set total cycle count at end of block
  return;
}
```

```
// pipeline stages
typedef enum {
  FE,     // fetch
  DE,     // decode
  EX,     // execute
  ME,     // memory
  WB,     // write back
  STAGES // 5 stages
} Stage;
```

```
// processor state
typedef struct {
  uint32_t pc;                                                       6
  uint32_t r[REGS];          // general purpose registers
  uint32_t auxr[AUXREGS];    // auxiliary registers
  char     L,Z,N,C,V,U,D,H;  // status flags (H...halt bit)
  uint64_t pl[STAGES];       // per stage cycle count                 7
  uint64_t avail[REGS];      // per register cycle count
  uint64_t cycles;           // total cycle count
  uint64_t ignore;           // used when insn. does not produce result
} CpuState;
```

Data Structures