СПбНИУ ИТМО

Курсовая работа

По дисциплине Организация ЭВМ и систем

Выполнил:

Манаков, группа 3121

2014

Цель работы

Разработка микропрограммного управления и схемы ЭВМ с архитектурой CISC и системой микрокоманд ***MCS51***. Исходными данными является программная модель на уровне ассемблера, перечень команд, выполняемых схемой и элементная база ***MaxPlus***.
Для функционального описания микропрограмм и моделирования могут быть использованы языки программирования, наиболее близким из которых является Си в системе ***BorlandC++***.
Схема проекта разрабатывается в системе ***MaxPlus*** и загружается в ПЛИС фирмы Алтера.
Верификация проекта выполняется в симуляторе ***MaxPlus***.
Для описания, визуального моделирования, кодирования и создания загрузочных файлов в проекте ***MaxPlus*** используется система ***BorlandC++***.

Команды ЭВМ

Список команд

- `DEC {Ri, @Rj, ad}`
- `ANL C, {bit, /bit}`
- `MOV A, {Ri, #d}`
- `JZ ref`

Описание команд

DEC

Данная операция уменьшает операнд на единицу. При применении этой операции к операнду, в котором находится `00h`, значение в операнде становится `0FFh`. Данная операция никак не влияет на флаги.

DEC Ri

C	AC	F0	RS1	RS0	OV		P

Размер команды (байт)	1
Количество циклов	1
Код команды	00011nnn
Операция	DEC Rn = Rn - 1
Пример	DEC R7

DEC @Rj

C	AC	F0	RS1	RS0	OV		P

Размер команды (байт)	1
Количество циклов	1
Код команды	0001011i
Операция	DEC (Rj) = (Rj) - 1
Пример	DEC @R1

DEC ad

C	AC	F0	RS1	RS0	OV		P

Размер команды (байт)	2
Количество циклов	1
Код команды	00010101 ad
Операция	DEC (direct) = (direct) - 1
Пример	DEC 35h

ANL

Данная операция выполняет побитовое логическое И между байтовыми или битовыми операндами и сохраняет результат в первом операторе.

ANL C, /bit

C	AC	F0	RS1	RS0	OV		P

Размер команды (байт)	2
Количество циклов	2
Код команды	10110000 bit
Операция	ANL C = C AND NOT (bit)
Пример	ANL C, /22h

ANL C, bit

C	AC	F0	RS1	RS0	OV		P

Размер команды (байт)	2
Количество циклов	2
Код команды	100000010 bit
Операция	ANL C = C AND (bit)
Пример	ANL C, 22h

MOV

Данная операция копирует данные второго операнда в первый операнд. Эта операция никак не влияет на исходные данные.

MOV A, Ri

C	AC	F0	RS1	RS0	OV		**P**

Размер команды (байт)	1
Количество циклов	1
Код команды	1110011i
Операция	MOV A = (Ri)
Пример	MOV A, @R1

MOV A, #d

C	AC	F0	RS1	RS0	OV		**P**

Размер команды (байт)	2
Количество циклов	1
Код команды	01110100 d
Операция	MOV A = d
Пример	MOV A, #0FFh

JZ

Данная операция передает контроль по адресу, если значение в аккумуляторе равно нулю. Иначе исполняется следующая за данной операцией инструкция. Эта операция не влияет на аккумулятор и флаги.

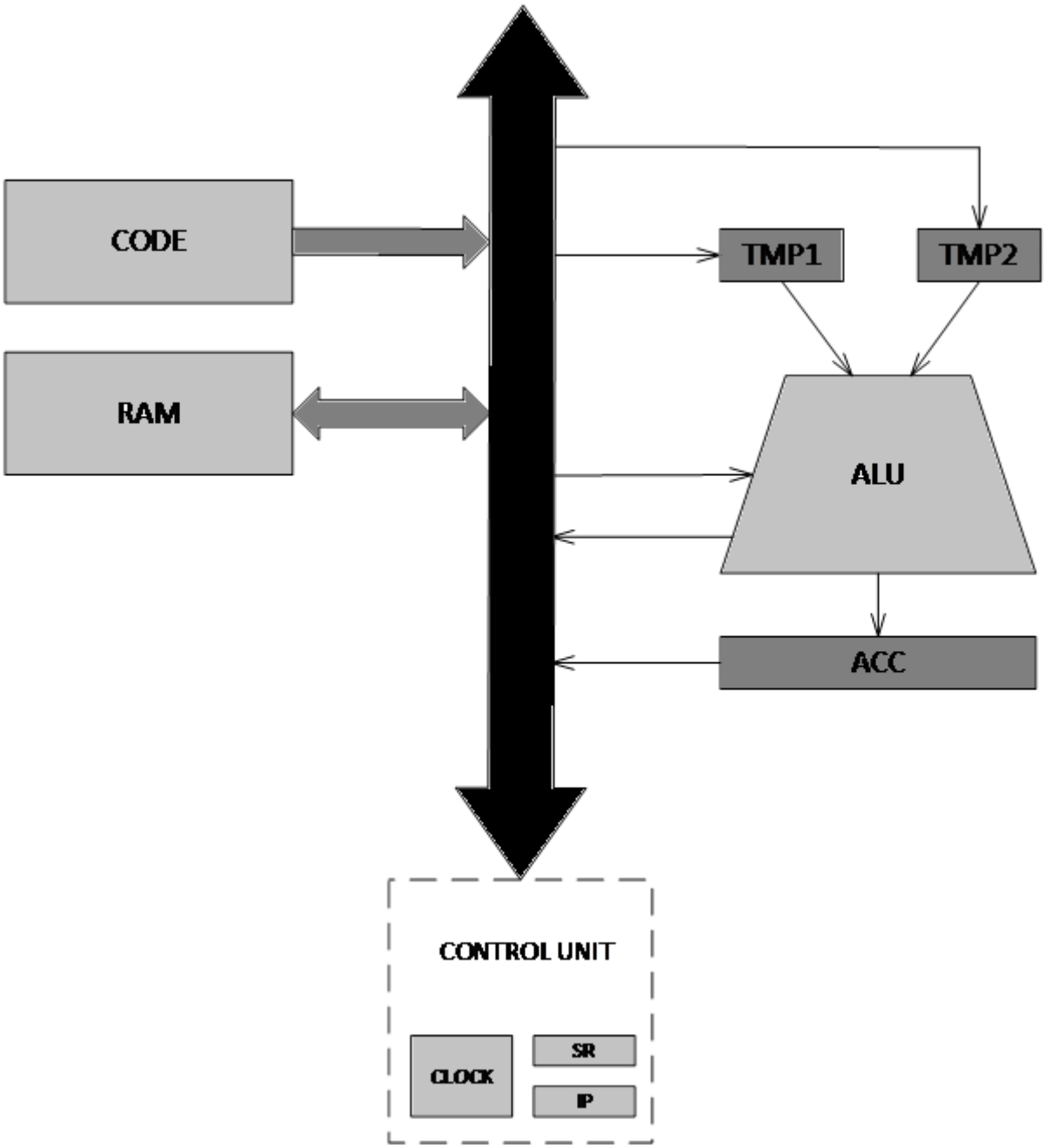
JZ rel

C	AC	F0	RS1	RS0	OV		P

Размер команды (байт)	2
Количество циклов	2
Код команды	01100000 rel
Операция	JZ PC = PC + 2 IF A = 0 PC = PC + rel
Пример	JZ LABEL

Структура ЭВМ

Структурная схема ЭВМ выбирается для определения необходимых ресурсов памяти и проектирования функциональной микропрограммы для команд теста.



Реализация эмулятора *MCS51*

Для реализации используется язык программирования Си и компилятор GCC.
В нашем случае понадобится реализовать еще одну команду для того чтобы была возможность использовать некоторые команды из задания.

MOV Rn, #immediate

C	AC	F0	RS1	RS0	OV		P

Данная операция записывает в регистр *Rn* значение *immediate*.

Размер команды (байт)	1
Количество циклов	1
Код команды	11111nnn
Операция	MOV Rn = immediate
Пример	MOV R5, #0h

Устройство и принцип работы эмулятора

Эмулятор состоит только из необходимых компонент, многие регистры, команды, внутреннее устройство самой архитектуры не реализованы, так как в этом нет необходимости. Нам необходимо только чтобы внешне нельзя было отличить работу эмулятора от работы микроконтроллера.
Программа хранит в себе значения всех необходимых нам регистров, ячеек памяти.
В нашем случае нам необходимы следующие регистры:

- Аккумулятор
- Счетчик команд
- R-регистры
- RSW-регистр

Также нам необходимо несколько видов памяти:

- ROM (память с программой)
- RAM (память для данных)
- Битовая память

Для простоты реализации битовые регистры были отделены от внутренней памяти и выделены в отдельную память.
Инструкция микроконтроллера описывается специальной структурой, которая хранит в себе *имя команды*, которое является ничем иным как *enum*, два поля *arg1* и *arg2*, которые отвечают за аргументы команды.
Данная структура введена для простого создания программ, нет необходимости вычислять программу где-то на бумаге или писать специальный код для преобразования мнемонический код команды в двоичный код. Также отпадает необходимость писать код для распознавания команды самим эмулируемым контроллером. Мы избавляемся от ненужной работы — сначала закодировать команду руками, затем ее раскодировать.
Все регистры, ячейки памяти имеют размер в один байт. Есть исключения, например, регистр состояний, память с программой, однобитовые регистры. Но эти исключения обусловлены удобством написания эмулятора. Нет необходимости усложнять эмулятор.
Может показаться, что эмулятор очень большой, на самом деле нет. Большую часть кода занимает формирование и форматирование вывода результата работы программы.
Команды эмулируемой программы заносятся в исходный код эмулятора.
Основной принцип работы строится на том, что у нас есть основная функция, которая занимается исполнением инструкции. Под исполнением инструкции понимается выполнение соответствующего данной инструкции функции.
Эмулируемая программа исполняется до тех пор, пока не будет достигнута последняя команда.

```c
#include <stdio.h>
#include <stdint.h>
/* === DATA ================================================================ */
#define DATA_MEMORY_SIZE 16
#define BIT_MEMORY_SIZE 8
#define R_COUNT 8

typedef enum {
    DEC_R_DIRECT,
    DEC_R_INDIRECT,
    DEC_DIRECT,

    ANL_C_BIT,
    ANL_C_NOT_BIT,


    MOV_R_N_IMMEDIATE,
    MOV_R_INDIRECT,
    MOV_DIRECT,

    JZ_REL
} InstructionName;

typedef struct {
    InstructionName name;
    uint8_t arg1;
    uint8_t arg2;
} Instruction;

Instruction program_memory[] = {

    // MOV_DIRECT test
    { MOV_DIRECT,        0x0A       }, // 0

    // MOV_R_N_IMMEDIATE and DEC_R_DIRECT test
    { MOV_R_N_IMMEDIATE, 0x00, 0xFF }, // 1
    { DEC_R_DIRECT,      0x00       }, // 2

    // MOV_R_N_IMMEDIATE and DEC_R_INDIRECT test
    { MOV_R_N_IMMEDIATE, 0x01, 0x00 }, // 3
    { DEC_R_INDIRECT,    0x01       }, // 4
    { DEC_R_INDIRECT,    0x01       }, // 5

    // ANL_C_NOT_BIT_TEST
    { ANL_C_NOT_BIT,     0x00       }, // 6

    // ANL_C_BIT_TEST
    { ANL_C_BIT,         0x00       }, // 7

    // JZ test
    { MOV_DIRECT,        0x00       }, // 8
    { JZ_REL,            0x02       }, // 9
    { MOV_R_N_IMMEDIATE, 0x02, 0x01 }, // 10
    { MOV_R_N_IMMEDIATE, 0x03, 0x01 }  // 11

};

uint8_t data_memory[DATA_MEMORY_SIZE];
uint8_t bit_memory[BIT_MEMORY_SIZE];
uint8_t r[R_COUNT];
uint8_t ic;
uint8_t acc;

struct {
    uint8_t carry;
```

```
    uint8_t parity;
} psw;
/* === INSTRUCTIONS ======================================================= */
void dec_r_direct(uint8_t arg) {
    r[arg]--;
    ic++;
}

void dec_r_indirect(uint8_t arg) {
    data_memory[r[arg]]--;
    ic++;
}

void dec_direct(uint8_t arg) {
    data_memory[arg]--;
    ic++;
}
void anl_c_bit(uint8_t arg) {
    psw.carry &= bit_memory[arg];
    ic++;
}
void anl_c_not_bit(uint8_t arg) {
    psw.carry &= !bit_memory[arg];
    ic++;
}
void mov_r_n_immediate(uint8_t r_i, uint8_t value) {
    r[r_i] = value;
    ic++;
}
void mov_r_indirect(uint8_t arg) {
    acc = data_memory[r[arg]];
    psw.parity = (acc % 2) == 0;
    ic++;
}
void mov_direct(uint8_t arg) {
    acc = arg;
    psw.parity = (acc % 2) == 0;
    ic++;
}
void jz_rel(uint8_t arg) {
    if (acc == 0)
        ic += arg;
    else
        ic++;
}
void execute_instruction(Instruction instruction) {
    switch(instruction.name) {
        case DEC_R_DIRECT:
        dec_r_direct(instruction.arg1);
        break;

        case DEC_R_INDIRECT:
        dec_r_indirect(instruction.arg1);
        break;

        case DEC_DIRECT:
        dec_direct(instruction.arg1);
        break;

        case ANL_C_BIT:
        anl_c_bit(instruction.arg1);
        break;

        case ANL_C_NOT_BIT:
        anl_c_not_bit(instruction.arg1);
```

```
            break;

            case MOV_R_N_IMMEDIATE:
            mov_r_n_immediate(instruction.arg1, instruction.arg2);
            break;

            case MOV_R_INDIRECT:
            mov_r_indirect(instruction.arg1);
            break;

            case MOV_DIRECT:
            mov_direct(instruction.arg1);
            break;

            case JZ_REL:
            jz_rel(instruction.arg1);
            break;

            default:
            printf("Unknown instruction");
        }
    }

    /* === CONTROL UNIT CONTROL ================================================ */

    void init_control_unit() {
        ic  = 0;
        acc = 0;
        psw.carry = 1;
        for (int i = 0; i < DATA_MEMORY_SIZE; i++) {
            data_memory[i] = 0x00;
        }
        for (int i = 0; i < R_COUNT; i++) {
            r[i] = 0x00;
        }
        for (int i = 0; i < BIT_MEMORY_SIZE; i++) {
            bit_memory[i] = 0;
        }
    }
    void execute_next_instruction() {
        Instruction instruction = program_memory[ic];
        execute_instruction(instruction);
    }
    void execute_program() {
        int program_length = sizeof(program_memory) / sizeof(Instruction);
        while (ic < program_length) {
            execute_next_instruction();
        }
    }
    void run() {
        while(1) {
            execute_program();
            ic = 0;
        }
    }

    /* === LOGGING ============================================================= */
    void print_registers_state() {
        printf("REGISTERS\n");
        printf("ACC: %d\n", acc);
        printf("PSW: C %d | P %d\n", psw.carry, psw.parity);
        printf("IC:  %d\n", ic);
    }
    void print_r_registers_state() {
        printf("R REGISTERS\n");
```

```
    for (int i = 0; i < R_COUNT; i++) {
        printf("%-3u ", r[i]);
    }
    printf("\n");
}
void print_bit_memory_state() {
    printf("BIT MEMORY\n");
    for (int i = 0; i < BIT_MEMORY_SIZE; i++) {
        printf("%u ", bit_memory[i]);
    }
    printf("\n");
}
void print_data_memory_state(int row_display_count) {
    printf("MEMORY\n");
    int columns = DATA_MEMORY_SIZE / row_display_count;
    for (int i = 0; i < row_display_count; i++) {
        for (int j = 0; j < columns; j++) {
            printf("%-6u", data_memory[i * columns + j]);
        }
        printf("\n");
    }
}
void print_state() {
    printf("\n\n");
    print_registers_state();

    printf("\n\n");
    print_r_registers_state();

    printf("\n\n");
    print_bit_memory_state();

    printf("\n\n");
    print_data_memory_state(4);
}
void print_instruction(Instruction instruction) {
   switch(instruction.name) {
        case DEC_R_DIRECT:
        printf("%-5s R%u", "DEC", instruction.arg1);
        break;

        case DEC_R_INDIRECT:
        printf("%-5s @R%u", "DEC", instruction.arg1);
        break;

        case DEC_DIRECT:
        printf("%-5s %Xh", "DEC", instruction.arg1);
        break;

        case ANL_C_BIT:
        printf("%-5s C, %Xh", "ANL", instruction.arg1);
        break;

        case ANL_C_NOT_BIT:
        printf("%-5s C, /%Xh", "ANL", instruction.arg1);
        break;

        case MOV_R_N_IMMEDIATE:
        printf("%-5s R%u, #%Xh", "MOV", instruction.arg1, instruction.arg2);
        break;

        case MOV_R_INDIRECT:
        printf("%-5s A, @R%u", "MOV", instruction.arg1);
        break;
```

```
        case MOV_DIRECT:
        printf("%-5s A, #%Xh", "MOV", instruction.arg1);
        break;

        case JZ_REL:
        printf("%-5s %u", "JZ", instruction.arg1);
        break;

        default:
        printf("Unknown instruction");
    }
}
void print_program() {
    int program_length = sizeof(program_memory) / sizeof(Instruction);
    for (int i = 0; i < program_length; i++) {
        printf("%3d | ", i);
        print_instruction(program_memory[i]);
        printf("\n");
    }
    printf("\n");
}
/* ================================================================== */
int main() {
    init_control_unit();
    run();

    return 0;
}
```

Тестовая микропрограмма

Для того чтобы проверить наш эмулятор, напишем небольшую тестовую программу, которая проверяет все реализованные нами инструкции.

```
1.  MOV A, #0Ah
2.  MOV R0, #0FFh
3.  MOV A, @R0
4.  DEC @R0
5.  MOV R1, #0h
6.  DEC @R1
7.  DEC @R1
8.  ANL C, /0h
9.  ANL C, 0h
10. MOV A, #0h
11. JZ #02h
12. MOV R2, #01h
13. MOV R3, #01h
14.
```

Результат работы программы

```
REGISTERS:          ACC: 0
                    PSW: C 0 | P 1
                    IC:  12

R REGISTERS:        254 0   0   1   0   0   0   0

BIT MEMORY:         0 0 0 0 0 0 0 0

MEMORY:             254   0     0     0
                    0     0     0     0
                    0     0     0     0
                    0     0     0     0
```

```
 0 | MOV   A, #Ah
 1 | MOV   R0, #FFh
 2 | DEC   R0
 3 | MOV   R1, #0h
 4 | DEC   @R1
 5 | DEC   @R1
 6 | ANL   C, /0h
 7 | ANL   C, 0h
 8 | MOV   A, #0h
 9 | JZ    2
10 | MOV   R2, #1h
11 | MOV   R3, #1h
```

Реализация схемы

Данный этап работы состоит из следующих частей:

1. Генерация кода на языке *Verilog*
2. Синтез схемы из *Verilog* при помощи специальных программных средств, например, MAX+Plus II

Для того чтобы реализовать схему можно использовать язык описания аппаратуры *Verilog HDL (Verilog Hardware Description Language)*. Так как наш эмулятор написать на *C*, то нам необходимо перевести код из *C* в код на *Verilog*.
Для перевода кода можно воспользоваться ресурсом *http://www.c-to-verilog.com*.
После того как мы получили код на Verilog, мы его используем для создания схемы в *САПР*.
Далее полученная схема может быть реализована и использована по назначению.

Анализ схемы

Исходя из структуры схемы, возможно проанализировать компоненты схемы, сколько и каких элементов у нас будет.
В качестве триггеров для реализации регистров и памяти будут использоваться D-триггеры.
D-триггер состоит и 4 логических элементов *И*. Каждый элемент *И* состоит и 4 транзисторов, если мы используем комплементарные транзисторы.
Так как у нас все регистры однобайтовые, кроме регистра команд, который состоит из двух байтов, то можно вычислить количество необходимых триггеров и транзисторов для регистров.
Для реализации однобайтового регистра нам необходимо 8 D-триггеров, что есть 32 транзистора.
Так как у нас 4 однобайтовых регистра, то нам потребуется 32 D-триггера или 128 транзисторов.
При реализации двухбайтового регистра потребуется 16 D-триггеров, что есть 64 транзистора.
Если сложить все однобитовые регистры, R-регистры и ячейки памяти, то их количество в сумме равняется 25 однобайтовым регистрам. Следовательно, для их реализации потребуется 200 D-триггеров или 800 транзисторов.
Также нам надо учесть память для программы. Будем считать, что она равняется 128 байтам, то есть в нее может поместиться 64 инструкции.
Для реализации памяти для кода нам потребуется 1024 D-триггера или 4096 транзисторов.
В итоге получается, что нам потребуется **5088** транзистора для реализации данной схемы.

Вывод по работе

В ходе данной работы были получены базовые знания в разработке системы на кристалле. Также была глубже изучена сама архитектура *8051* и ее набор инструкций.

Был реализован эмулятор микропроцессора на *C*, который затем использовался для генерации кода на *Verilog*, который в свою очередь использовался для автоматического проектирования схемы.

Программа на *C* была протестирована при помощи специальной тестовой микропрограммы. В ходе выполнения тестовой программы не было обнаружено ошибок.

В самом конце был произведен приблизительный подсчет требуемых транзисторов для реализации схемы. Данный анализ помогает нам приблизительно представить затраты на элементы, размеры конечной схемы.