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** | DECRn = 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** | ANLC = 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** | ANLC = 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** | MOVA = (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** | MOVA = 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** | JZPC = PC + 2IF 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** | MOVRn = 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.

#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

**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 *С* в код на *Verilog*.

Для перевода кода можно воспользоваться ресурсом [*http://www.c-to-verilog.com*](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 *С* 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.