北京郵電大學

计算机组成原理实验报告

题目：ARM 指令集系统

班　　级：2018211318 班

学　　号：2018210074

姓　　名：熊宇

学　　院：计算机学院

2020 年 6 月 3 日

目录

实验报告

一 实验内容和实验目的

此指令集以 ARM 指令集为基础进行开发，选取了较为常用的 35 条指令，并对指令编码方式进行了简化，形成了较为精简的指令集。

1、 指令长度

定长指令，长度为一个字 32 位

2、 操作数数量

按操作数数量可以分为三操作数、双操作数、单操作数、无操作数指令

3、 指令功能

按功能可以分为运算类、传送类、程序控制类和其他指令

4、 寻址方式

大多数指令的寻址方式为寄存器直接寻址和立即寻址，少数传送指令支持寄存器间接寻址、基址变址寻址和堆栈寻址，而跳转指令支持相对寻址

5、 条件执行

所有指令都支持条件执行

二 实验设备环境

Windows 环境 PC 机，Microsoft Visual Studio 2019集成化开发环境。

三 实验设计

1、 寄存器组织

（1）通用寄存器

```
R0 ~ R12
R13 (Stack Pointer, SP)
R14 (Link Register, LR)
R15 (Program Counter, PC)
```

（2）状态条件寄存器 CPSR

```
有四个状态位，分别为 Negative, Zero, Carry, Overflow
```

2、存储器组织

```
1M Byte 的物理地址
```

3、指令格式

```
cond:   条件码，若条件码代表的条件满足，则执行指令，否则不执行
optype: 将指令分为 4 类，分别进行编码
I:      标志位为 1 则代表立即数寻址，否则为寄存器寻址
op:     操作码
Rd:     目的寄存器
Rn, Rm: 源寄存器
```

（1）三操作数指令

```
| 31~28 | 27~26  | 25 | 24~21 | 20 | 19~16 | 15~12 | 11~0      |
| cond  | optype | 1  | op    | S  | Rd    | Rn    | imm_12    |
| cond  | optype | 0  | op    | S  | Rd    | Rn    | 0x00 | Rm |
```

（2）双操作数指令

```
| cond  | optype | 1  | op    | S  | Rd    | imm_16          |
| cond  | optype | 0  | op    | S  | Rd    | Rn    | 0x000     |
```

（3）单操作数指令

```
| cond  | optype | 1  | op    | S  | imm_20                |
| cond  | optype | 0  | op    | S  | Rd    | 0x0000          |
```

（4）无操作数指令

```
| cond  | optype | 0  | op    | S  | 0x00000               |
```

4、指令功能

（1）运算类指令

```
AND, ORR,  EOR, BIC, MVN
ADD, ADC,  SUB, SBC
MUL, UDIV, SDIV
LSL, LSR,  ASR, ROR
SXTB, SXTH
```

（2）传送类指令

```
MOVW, MOVT, MOV
LDRB, LDRH, LDR
```

```
STRB, STRH, STR
PUSH, POP
```

（3）程序控制类指令

```
NOP
SWI
B, BL
CMP, CMN, TST, TEQ
```

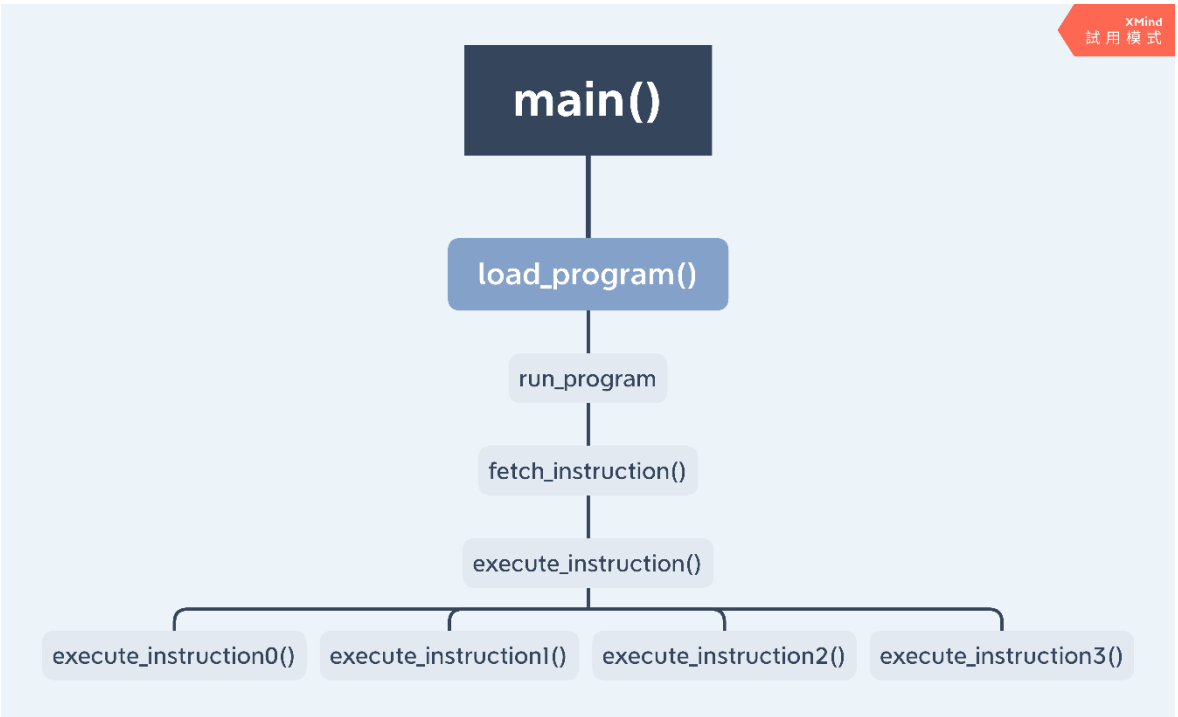
5、条件执行

cond	条件码	标志	含义
0000	EQ	Z == 1	Equal (==)
0001	NE	Z == 0	Not Equal (!=)
0010	CS	C == 1	Unsigned >=
0011	CC	C == 0	Unsigned <
0100	MI	N == 1	Negative
0101	PL	N == 0	Positive Or Zero
0110	VS	V == 1	Overflow
0111	VC	V == 0	No Overflow
1000	HI	(C == 1) && (Z == 0)	Unsigned >
1001	LS	(C == 0) || (Z == 1)	Unsigned <=
1010	GE	N == V	Signed >=
1011	LT	N != V	Signed <
1100	GT	(N == V) && (Z == 0)	Signed >
1101	LE	(N != V) || (Z == 1)	Signed <=
1110	AL	-	Always
1111	NV	-	Never

6、模块结构

（1）函数调用关系

```
main()
load_program()
      run_program()
                    fetch_instruction()
                       execute_instruction()
                           execute_instruction0()
                           execute_instruction1()
                           execute_instruction2()
                           execute_instruction3()
```



（2）文件结构关系

```
A、defs.h
数据类型定义
寄存器、内存变量的声明

B、main.c
`main()` 整个程序的入口
调用 `load_program()` 将程序装入内存，再调用 `run_program()` 运行程序

C、part1.c
`load_program()` 将指令的机器码装入内存
`run_program()`  执行装入内存的指令
调用 `fetch_instruction()` 和 `execute_instruction()` 执行指令

D、part2.c
`fetch_instruction()`    取指令
`execute_instruction()`  执行指令
将指令分为 4 大类，分别调用相应的执行函数执行
`exeucte_instruction0()`
`exeucte_instruction1()`
`exeucte_instruction2()`
`exeucte_instruction3()`

E、print.c
`input()`  暂停程序，显示提示信息
`print()`  将寄存器、内存状态显示在屏幕上
```

四 测试代码

```
#define _CRT_SECURE_NO_WARNINGS

# include <stdio.h>
# include <string.h>

int as[] = {
    0xe4600000,  // push r0
    0xe4610000,  // push r1
    0xe4620000,  // push r2
    0xe4630000,  // push r3
    0xe4640000,  // push r4
    0xe4650000,  // push r5
    0xe4660000,  // push r6
    0xe4670000,  // push r7

    0xea00f000,  // mov r0, array=0x0f000
    0xea010004,  // mov r1, 4
    0xea020002,  // mov r2, 2
    0xea030000,  // mov r3, 0
    0xea040001,  // mov r4, 1
    0xe2a40000,  // str r4, [r0, 0]
    0xec850001,  // add r5, r0, r1
    0xe2a45000,  // str r4, [r5, 0]

    0xe6800002,  //SWI #0x2

    //loop:
    0xe2830000,  // ldr r3, [r0, 0]
    0xe2865000,  // ldr r6, [r5, 0]
    0xec833006,  // add r3, r3, r6
    0xe0a35001,  // str r3, [r5, r1]
    0xec800001,  // add r0, r0, r1
    0xec855001,  // add r5, r5, r1
    0xee822001,  // add r2, r2, 1
    0xead20063,  // cmp r2, #0x63
    0xd62fffe0,  // ble loop(-0x20)

    0xe4460000,  // pop r6
    0xe4450000,  // pop r5
    0xe4440000,  // pop r4
    0xe4430000,  // pop r3
    0xe4420000,  // pop r2
```

```
        0xe4410000,  // pop r1
        0xe4400000,  // pop r0


    };

    int main() {
        FILE* fp = fopen("example", "wb");

        for (int i = 0; i < sizeof(as) / sizeof(int); ++i) {
            fwrite(as + i, sizeof(int), 1, fp);
            printf("%x", as[i]);
        }

        fclose(fp);

        return 0;
    }
```

五 实验总结

通过这次实验，我对 ARM 指令集相对于 x86 指令集的区别有了深入的理解和掌握。在后台模拟 ARM 指令集每条指令的实现过程中，我对于 CPU 的工作流程和总线冲突有了更深层次的理解；之前在课堂上模糊不清的指令执行步骤也通过本次实验逐一弄懂。

特别地，在书写 ARM 指令集带进位的加减法——即 ADC 、 SBC 实现中关于更新 C 、 V 标志位相关代码时，结合Reference ，我对溢出有了更本质的认识，也了解到在计算 C 、 V 时应当使用 64 位进行运算而非我们指令的数据类型 32 位。在软中断指令 SWI 的实现过程中，明了保护现场与恢复现场、利用中断矢量表跳转到中断服务程序的步骤和过程。

在本次虚拟机大作业中，对于软件和硬件的逻辑等价性有了基于实际层次上的体会和认同。

六 系统使用说明书

用作测试的汇编程序对应的机器码已经写入 `example` 文件，详见四 测试代码。

在 Windows 环境下直接运行 `ARM.exe` 即可，程序会自动读取 `example` 中的指令机器码，译码并执行。

执行过程中的寄存器状态、内存状态、将要执行的指令会显示在屏幕上。

七 源代码及可执行文件

源代码见附页。
可执行文件见附件。

defs.h

```
# ifndef __DEFS_H__
# define __DEFS_H__

# include <stdint.h>

/* type definition */

typedef uint32_t word;
typedef uint16_t hword;
typedef uint8_t  byte;

/* registers and memory definition */

# define SP R[13]               /* Stack Pointer */
# define LR R[14]               /* Link Register */
# define PC R[15]               /* Program Counter */

# define MEMORY_SIZE (1<<20)     /* 1MB */

extern word R[16];              /* general purpose registers */
extern word CPSR;                     /* Current Program Status
Register */
extern word IR;                 /* Instruction Register */
extern word AR;
extern word DR;

extern word EA;                 /* Interruption Allowed */
extern word ER;                 /* Interruption Response */
extern word QUERY[5];

extern byte M[MEMORY_SIZE];      /* main memory */
                                /*  0x00000 ~ 0x0ffff belongs to
operating system
                                 *  0x10000 ~ 0xfffff belongs to
users */


//#define M[100] = 0x0100 00F0     /* push ac */
/* mov ac, 0 */
/* disp 中断了，成功 */
/* pop ac */
/* ei = 0 */
/* pop pc */


/* opcode and cond for decoding machine code */
```

```
enum CONDITION {     /* arm cond */
    EQ, NE, CS, CC,  /* 0000 ~ 0011 */
    MI, PL, VS, VC,  /* 0100 ~ 0111 */
    HI, LS, GE, LT,  /* 1000 ~ 1011 */
    GT, LE, AL, NV   /* 1100 ~ 1111 */
};

enum OP3 {    /* triple operand instructions */
    AND, ORR,  EOR,  BIC,
    ADD, ADC,  SUB,  SBC,
    MUL, UDIV, SDIV,
    LSL, LSR,  ASR,  ROR
};

enum OP0 {    /* zero and some triple operand instructions */
    LDRB, STRB,
    LDRH, STRH,
    LDR,  STR,
    NOP
};

enum OP2 {    /* single operand instructions */
    MOV,  MOVW, MOVT, MVN,
    SXTB, SXTH,
    CMP,  CMN,  TST,  TEQ
};

enum OP1 {    /* double operand instructions */
    BL,  B,
    POP, PUSH,
    SWI
};

# endif
```

part1.h

```
# ifndef __PART1_H__
# define __PART1_H__

# include "defs.h"

word load_program (const char*);    /* return the start_address
*/
int  run_program  (word);           /* return 0 if no error or
error code */
```

```
# endif
```

part2.h

```
# ifndef __PART2_H__
# define __PART2_H__

void fetch_instruction();
void execute_instruction();

# endif
```

print.h

```
# ifndef __PRINT_H__
# define __PRINT_H__

void input();
void print();

# endif
```

part1.c

```
#define _CRT_SECURE_NO_WARNINGS

# include <stdio.h>
# include <string.h>
# include <stdlib.h>
# include <stdbool.h>

# include "defs.h"
# include "print.h"
# include "part2.h"


word load_program(const char* exec_file) {  /* done */

    FILE* fp = fopen(exec_file, "rb");
    if(!fp) {
```

```
                printf("error: executable file \"%s\" does not exist\n",
exec_file);
                exit(1);
            }

            /* move the exec_file to vm memory */
            word start_address = 0x20000;

            for( word address = start_address; !feof(fp) ; address +=
sizeof(word)) {
                fread(M + address, sizeof(word), 1, fp);
            }

            fclose(fp);

            return start_address;
        }

        int run_program(word start_address) { /* done */

            PC = start_address;
            SP = 0x01000;
            EA = 0;
            ER = 0;

            while(1) {  /* 指令周期 */

                print();
                input();

                EA = 0;            //开中断
                ER = 0;
                fetch_instruction();
                execute_instruction();
            }

            return 0;
        }
```

part2.c

```
        #define _CRT_SECURE_NO_WARNINGS

        # include <stdio.h>
        # include <string.h>
        # include <stdlib.h>
        # include <windows.h>
```

```
# include <stdbool.h>

# include "defs.h"

#define MSB(x) ((x >> 31) & 0x1)

/* registers and memory definition */

word R[16];                    /* general purpose registers */
word CPSR;                     /* Current Program Status Register
*/

word IR;                       /* Instruction Register */
word AR;
word DR;
byte M[MEMORY_SIZE];            /* main memory */
word EA;                  /* Interruption Allowed */
word ER;                  /* Interruption Response */
word QUERY[5] = { 0x2000, 0x3000, 0x300a, 0x5000, 0x6000 };

/* buffer for CPSR */

static byte N, Z, C, V;

void fetch_instruction() {   /* done */
    AR = PC;
    IR = DR = *(word*)(M + AR);
    PC += 4;
}

/* 是否满足指令执行的条件 */
static bool condition_passed() {  /* done */

    bool ans;
    byte cond       = ((IR >> 28) & 0xf);    /*  4 bit */

    /* 状态寄存器译码 */
    byte N = ((CPSR >> 31) & 0x1);
    byte Z = ((CPSR >> 30) & 0x1);
    byte C = ((CPSR >> 29) & 0x1);
    byte V = ((CPSR >> 28) & 0x1);

    /* 判断执行条件 */
    switch(cond) {
        case EQ: ans = (Z == 1); break;
        case NE: ans = (Z == 0); break;
        case CS: ans = (C == 1); break;
        case CC: ans = (C == 0); break;
        case MI: ans = (N == 1); break;
```

```
                case PL: ans = (N == 0); break;
                case VS: ans = (V == 1); break;
                case VC: ans = (V == 0); break;
                case HI: ans = ((C == 1) && (Z == 0)); break;
                case LS: ans = ((C == 0) || (Z == 1)); break;
                case GE: ans = (N == V); break;
                case LT: ans = (N != V); break;
                case GT: ans = ((N == V) && (Z == 0)); break;
                case LE: ans = ((N != V) || (Z == 1)); break;
                case AL: ans = true;  break;
                case NV: ans = false; break;
            }

            return ans;
        }

        static void execute_instruction3() {
            /* decode instruction */
            byte I          = ((IR >> 25) & 0x1);    /*  1 bit */
            byte op         = ((IR >> 21) & 0xf);    /*  4 bit */
            byte Rd         = ((IR >> 16) & 0xf);    /*  4 bit */
            byte Rn         = ((IR >> 12) & 0xf);    /*  4 bit */
            hword shifter_operand = (IR & 0xfff);    /* 12 bit */

            /* for and, add */
            word first_operand  = R[Rn];
            word    second_operand    =    (I    ?    shifter_operand    :
R[shifter_operand & 0xf]);    /* imm_12 or register */

            /* for lsl */
            byte shift = shifter_operand & 0x1f;

            /* execute instruction */
            switch(op) {
                case AND: {
                    R[Rd] = first_operand & second_operand;
                    N = MSB(R[Rd]);
                    Z = !R[Rd];
                    C = 0;
                    /* V unchanged */
                    break;
                }
                case ORR: {
                    R[Rd] = first_operand | second_operand;
                    N = MSB(R[Rd]);
                    Z = !R[Rd];
                    C = 0;
                    /* V unchanged */
                    break;
```

```
                    }
                    case EOR: {
                        R[Rd] = first_operand ^ second_operand;
                        N = MSB(R[Rd]);
                        Z = !R[Rd];
                        C = 0;
                        /* V unchanged */
                        break;
                    }
                    case BIC: {
                        R[Rd] = first_operand & (~ second_operand);
                        N = MSB(R[Rd]);
                        Z = !R[Rd];
                        C = 0;
                        /* V unchanged */
                        break;
                    }

                    case ADD: {
                        R[Rd] = first_operand + second_operand;
                        N = MSB(R[Rd]);
                        Z = !R[Rd];
                        C        =        !(((uint64_t)first_operand        +
(uint64_t)second_operand)==(uint64_t)(R[Rd]));
                        V        =        !(((int64_t)first_operand        +
(int64_t)second_operand)==((int64_t)R[Rd]));
                        break;
                    }
                    case ADC: {
                        R[Rd] = first_operand + second_operand + C;
                        N = MSB(R[Rd]);
                        Z = !R[Rd];
                        C        =        !(((uint64_t)first_operand        +
(uint64_t)second_operand + (uint64_t)C)==(uint64_t)(R[Rd]));
                        V        =        !(((int64_t)first_operand        +
(int64_t)second_operand + (uint64_t)C)==((int64_t)R[Rd]));
                        break;
                    }
                    case SUB: {
                        R[Rd] = first_operand - second_operand;
                        N = MSB(R[Rd]);
                        Z = !R[Rd];
                        C        =        !(((uint64_t)first_operand        -
(uint64_t)second_operand)==(uint64_t)(R[Rd]));
                        V        =        !(((int64_t)first_operand        -
(int64_t)second_operand)==((int64_t)R[Rd]));
                        break;
                    }
                    case SBC: {
```

```
                        R[Rd] = first_operand - second_operand + C - 1;
                        N = MSB(R[Rd]);
                        Z = !R[Rd];
                        C        =        !(((uint64_t)first_operand        -
(uint64_t)second_operand + (uint64_t)C - 1)==(uint64_t)(R[Rd]));
                        V        =        !(((int64_t)first_operand        -
(int64_t)second_operand + (uint64_t)C - 1)==((int64_t)R[Rd]));
                        break;
                    }
                    case MUL: {
                        R[Rd] = first_operand * second_operand;
                        N = MSB(R[Rd]);
                        Z = !R[Rd];
                        /* C V unchanged */
                        break;
                    }
                    case UDIV: {
                        if( second_operand == 0x0)
                        {
                            fprintf(stderr,   "error:   divisor   cannot   be
zero\n");
                            R[Rd] = 0;
                        }
                        else
                            R[Rd] = first_operand / second_operand;
                        /* N Z C V unchanged */
                        break;
                    }
                    case SDIV: {
                        if( second_operand == 0x0)
                        {
                            fprintf(stderr,   "error:   divisor   cannot   be
zero\n");
                            R[Rd] = 0;
                        }
                        else
                            R[Rd]      =      (int32_t)(first_operand)      /
(int32_t)(second_operand);
                        /* N Z C V unchanged */
                        break;
                    }

                    case LSL: {
                        R[Rd] = first_operand << shift;
                        N = MSB(R[Rd]);
                        Z = !R[Rd];
                        C = MSB(first_operand << (shift - 1));     /* Last
digit removed */
                        /* V unchanged */
```

```
                    break;
                }
                case LSR: {
                    R[Rd] = first_operand >> shift;
                    N = MSB(R[Rd]);
                    Z = !R[Rd];
                    C = (first_operand >> (shift - 1)) & 0x1;    /* Last
digit removed */
                    /* V unchanged */
                    break;
                }
                case ASR: {
                    R[Rd] = ((int32_t)(first_operand)) >> shift;
                    N = MSB(R[Rd]);
                    Z = !R[Rd];
                    C = (first_operand >> (shift - 1)) & 0x1;    /* Last
digit removed */
                    /* V unchanged */
                    break;
                }
                case ROR: {
                    R[Rd] = (first_operand >> shift) | (first_operand <<
(32 - shift));
                    N = MSB(R[Rd]);
                    Z = !R[Rd];
                    C = ((first_operand >> (shift - 1)) | (first_operand
<< (32 - (shift - 1)))) & 0x1;    /* Last digit removed */
                    /* V unchanged */
                    break;
                }

                default: exit(2); break;

            }
        }

        static void execute_instruction0() {
            /* decode instruction */
            byte I         = ((IR >> 25) & 0x1);    /*  1 bit */
            byte op        = ((IR >> 21) & 0xf);    /*  4 bit */
            byte Rd        = ((IR >> 16) & 0xf);    /*  4 bit */
            byte Rn        = ((IR >> 12) & 0xf);    /*  4 bit */
            hword shifter_operand = (IR & 0xfff);    /* 12 bit */

            /* for ldr */
            word base  = R[Rn];
            word offset = (I ? shifter_operand : R[shifter_operand &
0xf]);    /* imm_12 or register */
```

```
                switch(op) {
                    case LDRB: {
                        AR = base + offset;
                        R[Rd] = DR = *(byte*)(M + AR);
                        break;
                    }
                    case STRB: {
                        AR = base + offset;
                        *(byte*)(M + AR) = DR = (byte)R[Rd];
                        break;
                    }
                    case LDRH: {
                        AR = base + offset;
                        R[Rd] = DR = *(hword*)(M + AR);
                        break;
                    }
                    case STRH: {
                        AR = base + offset;
                        *(hword*)(M + AR) = DR = (hword)R[Rd];
                        break;
                    }
                    case LDR: {
                        AR = base + offset;
                        R[Rd] = DR = *(word*)(M + AR);
                        break;
                    }
                    case STR: {
                        AR = base + offset;
                        *(word*)(M + AR) = DR = (word)R[Rd];
                        break;
                    }

                    case NOP: break;

                    default: exit(2); break;
                }

            }

            static void execute_instruction2() {

                /* decode instruction */
                byte I          = ((IR >> 25) & 0x1);    /*  1 bit */
                byte op         = ((IR >> 21) & 0xf);    /*  4 bit */
                byte Rd         = ((IR >> 16) & 0xf);    /*  4 bit */

                /* for mov and sxt */
                word src_operand = ( I ? (IR & 0xffff) : R[IR & 0xf]);    /*
imm_16 or register */
```

```
                /* for cmp */
                word result;
                word first_operand  = R[Rd];
                word second_operand = src_operand;

                switch(op) {
                    case MOV:  /* the same as movw */
                    case MOVW: R[Rd]  = src_operand;          break;
                    case MOVT: R[Rd] |= (src_operand << 0x10); break;
                    case MVN:  R[Rd] = ~src_operand;          break;

                    case SXTB: R[Rd] = (int32_t)(int8_t )src_operand; break;
                    case SXTH: R[Rd] = (int32_t)(int16_t)src_operand; break;

                    case CMP: {
                        result = first_operand - second_operand;
                        N = MSB(result);
                        Z = !result;
                        C = (first_operand >= second_operand);
                        V   =   (~(MSB(result)   ^   MSB(second_operand)))   &
(MSB(first_operand) ^ MSB(second_operand));
                        break;
                    }
                    case CMN: {
                        result = first_operand + second_operand;
                        N = MSB(result);
                        Z = !result;
                        C   =   (result   <   first_operand)   ||   (result   <
second_operand);
                        V = (MSB(result) ^ MSB(first_operand)) & (MSB(result)
^ MSB(second_operand));
                        break;
                    }
                    case TST: {
                        result = first_operand & second_operand;
                        N = MSB(result);
                        Z = !result;
                        C = 0;
                        /* V unchanged */
                        break;
                    }
                    case TEQ: {
                        result = first_operand ^ second_operand;
                        N = MSB(result);
                        Z = !result;
                        C = 0;
                        /* V unchanged */
                        break;
```

```
                    }


                    default: exit(2); break;
                }
            }

            static void execute_instruction1() {

                /* decode instruction */
                byte op = ((IR >> 21) & 0xf);    /*  4 bit */
                byte Rd = ((IR >> 16) & 0xf);    /*  4 bit */

                /* for b */
                word imm_20 = (IR & 0xfffff);    /* 20 bit */
                int32_t offset = ((imm_20 & 0x80000) ? (imm_20 | 0xfff00000) :
imm_20 ); /* sign extension */

                switch(op) {
                    case BL: LR = PC;     /* no break; */ /* pc +=4 when
fetching instruction */
                    case B:  PC = PC - 4 + offset; break;     /* offset is
relative to current instruction */

                    case POP: {
                        AR = SP;
                        DR = *(word*)(M + AR);
                        R[Rd] = DR;
                        SP = SP + 4;
                        break;
                    }
                    case PUSH: {
                        SP = SP - 4;
                        AR = SP;
                        DR = R[Rd];
                        *(word*)(M + AR) = (word)DR;
                        break;
                    }
                    case SWI: {
                        *(word*)(M + 0x300a) = (word)0xe46c0000;//push R12;保
护现场
                        *(word*)(M + 0x300e) = (word)0xea0c0000;//mov R12, #0;
是设备服务功能，假设让 ac=0
                                                //disp 中断了，成功；模型机机器指
令 disp
                        *(word*)(M + 0x3012) = (word)0xe44c0000;//pop R12; 恢
复现场
                        //printf("准备写入 pop pc\n");
```

```
                        *(word*)(M + 0x3016) = (word)0xe44f0000;//pop pc；回复
现场，返回原程序断点
                        //printf("已经写入 pop pc，机器码为%x\n", M[0x3016]);
                        system("cls");
                        printf("An interrupt requestn had been made, interrupt
number is %x\n", imm_20);
                        ER = 1;
                        printf("Interruption response is in progress\n");
                        getchar();
                        //push pc
                        SP = SP - 4;
                        AR = SP;
                        DR = PC;
                        *(word*)(M + AR) = (word)DR;
                        PC = QUERY[imm_20];
                        printf("The  interrupt  response  ends.  Access  to
interrupted service\n");
                        printf("The interrupted service ends. Go back\n");
                        printf("The interrupt is successful\n");
                        getchar();
                        break;
                    }

                    default: exit(2); break;
                }
            }

            void execute_instruction() {

                if(!condition_passed())
                    return;

                /* buffer for CPSR */
                N = ((CPSR >> 31) & 0x1);
                Z = ((CPSR >> 30) & 0x1);
                C = ((CPSR >> 29) & 0x1);
                V = ((CPSR >> 28) & 0x1);

                byte optype = ((IR >> 26) & 0x3);    /*  2 bit */
                byte S      = ((IR >> 20) & 0x1);    /*  1 bit */

                /* switch */
                switch(optype) {
                    case  3:  execute_instruction3();  break;       /*  triple
operand instructions */
                    case  2:  execute_instruction2();  break;       /*  double
operand instructions */
                    case  1:  execute_instruction1();  break;       /*  single
operand instructions */
```

```
                    case 0: execute_instruction0(); break;    /* zero operand
and some triple operand instructoins */
                }

                /* renew CPSR */
                if(S) {
                    CPSR = ((N << 31) | (Z << 30) | (C << 29) | (V << 28));
                }
            }
```

print.c

```
            #define _CRT_SECURE_NO_WARNINGS

            # include <stdio.h>
            # include <string.h>
            # include <stdint.h>
            # include <stdbool.h>
            # include <stdlib.h>

            # include "defs.h"

            void input() {

                printf("Press Enter to execute next instruction.\n");
                getchar();

            }

            static char* AS_CONDITION [] = {
                "EQ", "NE", "CS", "CC",
                "MI", "PL", "VS", "VC",
                "HI", "LS", "GE", "LT",
                "GT", "LE", "",   "NV"
            };

            static char* AS_OP3 [] = {
                "AND", "ORR",  "EOR",  "BIC",
                "ADD", "ADC",  "SUB",  "SBC",
                "MUL", "UDIV", "SDIV",
                "LSL", "LSR",  "ASR",  "ROR"
            };

            static char* AS_OP0 [] = {
                "LDRB", "STRB",
                "LDRH", "STRH",
                "LDR",  "STR",
```

```
    "NOP"
};

static char* AS_OP2 [] = {
    "MOV",  "MOVW", "MOVT", "MVN",
    "SXTB", "SXTH",
    "CMP",  "CMN",  "TST",  "TEQ"
};

static char* AS_OP1 [] = {
    "BL",  "B",
    "POP", "PUSH",
    "SWI"
};

static char** AS_OP [] = {
    AS_OP0, AS_OP1, AS_OP2, AS_OP3
};

static void disas_instruction(char as[], word ir) {

    int len = 0;

    byte cond   = ((ir >> 28) & 0xf);
    byte optype = ((ir >> 26) & 0x3);
    byte I      = ((ir >> 25) & 0x1);
    byte op     = ((ir >> 21) & 0xf);
    byte S      = ((ir >> 20) & 0x1);

    bool suffix_S = (S && (optype == 3));

    len = sprintf(as, "%s%s%s ",
           AS_OP[optype][op],      /* ADD */
           AS_CONDITION[cond],     /* EQ  */
           (suffix_S ? "S" : "") ); /* S   */

    byte  Rd = ((ir >> 16) & 0xf);
    byte  Rn = ((ir >> 12) & 0xf);
    byte  Rm = (ir & 0xf);

    word imm_5  = (ir & 0x1f);
    word imm_12 = (ir & 0xfff);
    word imm_16 = (ir & 0xffff);
    word imm_20 = (ir & 0xfffff);

    switch(optype) {
        case 0: {
            if(op == NOP)       break;
```

```
                        if(!I)   len += sprintf(as+len, "R%-2d, [R%-2d, R%-
2d]", Rd, Rn, Rm);
                        else       len += sprintf(as+len, "R%-2d, [R%-2d,
#0x%x]", Rd, Rn, imm_12);
                        break;
                    }
                    case 3: {
                        if(!I)  len += sprintf(as+len, "R%-2d, R%-2d, R%-2d",
Rd, Rn, Rm);
                        else    len += sprintf(as+len, "R%-2d, R%-2d, #0x%x",
Rd, Rn, imm_12);
                        break;
                    }
                    case 2: {
                        if(!I)   len += sprintf(as+len, "R%-2d, R%-2d", Rd,
Rn);
                        else     len += sprintf(as+len, "R%-2d, #0x%x", Rd,
imm_16);
                        break;
                    }
                    case 1: {
                        if(!I)  len += sprintf(as+len, "R%-2d", Rd);
                        else    len += sprintf(as+len, "#0x%x", imm_20);
                        break;
                    }
                }

            }

            void print() {

                char as[3][64];

                /* windows */
                system("cls");

                disas_instruction(as[0],  *(word*)(M+PC-4));         /*  the
instruction just executed */
                disas_instruction(as[1],  *(word*)(M+PC    ));        /*  the
instruction to be executed */
                disas_instruction(as[2], *(word*)(M+PC+4));

                /* show registers */
                printf("Registers\n");
                printf("R0  :0x%08x         R1  :0x%08x         R2  :0x%08x
R3 :0x%08x\n", R[0],  R[1],  R[2],  R[3]);
                printf("R4  :0x%08x         R5  :0x%08x         R6  :0x%08x
R7 :0x%08x\n", R[4],  R[5],  R[6],  R[7]);
```

```
                printf("R8   :0x%08x          R9   :0x%08x          R10:0x%08x
R11:0x%08x\n", R[8],  R[9],  R[10], R[11]);
                printf("R12:0x%08x          SP   :0x%08x          LR   :0x%08x
PC :0x%08x\n", R[12], SP ,   LR,    PC   );
                printf("CPSR: 0x%08x\n", CPSR);
                printf("\n");

                /* show memory recently visited */
                int32_t center_address = AR / 4 * 4;
                int32_t first_address = center_address - 32;

                printf("Memory\n");
                for(int i = 0; i < 4; ++i) {
                    for(int j = 0; j < 4; ++j) {
                        int32_t ad = first_address + (i*4 + j) * 4;
                        if(ad > 0)
                            printf("0x%05x: 0x%08x%s", ad, *(word*)(M + ad),
((j==3) ? "\n" : "    " ));
                    }
                }
                printf("\n");

                /* show stack */
                printf("Stack\n");
                printf("      0x%05x 0x%08x\n", SP+12, *(word*)(M+SP+12));
                printf("      0x%05x 0x%08x\n", SP+8 , *(word*)(M+SP+8 ));
                printf("      0x%05x 0x%08x\n", SP+4 , *(word*)(M+SP+4 ));
                printf("SP -> 0x%05x 0x%08x\n", SP   , *(word*)(M+SP   ));
                printf("\n");

                /* show instructions */
                printf("Instructions\n");
                printf("       0x%05x 0x%08x %s\n", PC-4, *(word*)(M+PC-4),
as[0]);
                printf("PC -> 0x%05x 0x%08x %s\n", PC,    *(word*)(M+PC  ),
as[1]);
                printf("       0x%05x 0x%08x %s\n", PC+4, *(word*)(M+PC+4),
as[2]);
                printf("\n");

            }
```

main.c

```
            #define _CRT_SECURE_NO_WARNINGS

            # include <stdio.h>
```

```c
# include <string.h>
# include <stdlib.h>

# include "defs.h"
# include "part1.h"



int main(int argc, char ** argv) {  /* done */


    int exit_code;

    if(argc == 1 || argc == 2) {
        const char* exec_file;
        word start_address;

        exec_file = (argc == 1 ? "example" : argv[1]);
        start_address = load_program(exec_file);
        exit_code = run_program(start_address);
    }
    else {
        printf("error: multiple input files not supported\n");
        exit(1);
    }

    return exit_code;
}
```