**2024秋计算机体系结构**

**Project 01——设计并实现MIPS模拟器**

2024年9月19日

说明：这个项目（Project）涉及教材第一和第二章的内容，覆盖的知识点包括冯·诺依曼体系结构、计算机系统的层次结构、指令的翻译执行和指令系统等等。为了简化问题，我们仅考虑冯·诺依曼结构的CPU（控制器、运算器）和存储器。

# 用C语言描述主存

## 1.1 主存的结构和功能

主存一般被组织成按字节寻址的线性地址空间，并以字节为单位进行读写访问，即一次从主存中读出一个字节，或向主存中写入一个字节。当然，主存也可以按照半字（*2*个字节）、字（*4*个字节）、双字（*8*个字节）为单位访问。

为了提高访问效率，指令和数据一般会按地址对齐的方式存放在主存中。例如，按照这种方式，长度为*32*位（即*1*个字）的指令或数据在主存中的地址应能被*4*整除，若将其地址表示为二进制形式，地址的最低两位为*00*。

## 1.2 主存的C语言描述

* 主存的结构

描述主存结构的参考*C*源程序如下，这里我们将字节指针变量*mem*和整数变量*mem\_size*定义为全局变量。

**代码1 “主存”的创建和释放**

|  |
| --- |
| *#include <stdio.h>*  *#include <stdlib.h>*  *char \*mem; //* 主存起始地址指针  *int mem\_size=0x080000; //* 主存的大小  *void main(void)*  *{*  *mem = (char \*)malloc( sizeof(char) \* mem\_size );*  *if ( mem==NULL )*  *{*  *printf( “error: main memory allocation\n” );*  *exit( -1 );*  *}*  *free( mem );*  *}* |

这段代码的主要功能如下：首先，调用函数*malloc*申请一块大小为*mem\_size*个字节的存储空间。执行完函数*malloc*后，若字节指针变量*mem*的值为*NULL*，表示申请失败，应报错并退出；否则表示申请成功，*mem*的值就是这片空间的起始地址。

* 主存的功能

主存有两个功能：读访问，即读出存放在某个主存单元中的数据，以及写访问，即将数据写入主存中的特定单元。要实现这两个功能，必须明确以下信息：1）被读或写访问的主存单元的起始地址，地址一般是一个无符号整数；2）要读出或写入的数据类型，如有符号/无符号字节（*char/unsigned char*）、有符号/无符号半字（*short/unsigned short*）、有符号/无符号字（*int/unsigned int*）、单精度浮点（*float*）、双精度浮点（*double*）等。

代码2中的两个函数分别实现了将无符号字节变量val写入地址为*addr*的主存单元（*write\_mem\_ubyte*），和从地址为*addr*的主存单元读出一个无符号字节数（*read\_mem\_ubyte*）的功能。*mem*就是代码1中申请的存储空间的起始地址指针。

**代码2 无符号字节数的读出和写入**

|  |
| --- |
| *unsigned char read\_mem\_ubyte(long addr)*  *{*  *return \*((unsigned char \*)(mem+addr));*  *}*  *void write\_mem\_ubyte(long addr, unsigned char val)*  *{*  *\*((unsigned char \*)(mem+addr)) = val;*  *}* |

代码2的两个函数可被复制到代码1的*main( )*函数之前。

## 1.3 正确性测试

可以通过一段代码测试代码2两个函数的功能是否正确。代码3中的语句应被放在代码1中*free(mem)*语句之前。

**代码3 无符号字节数读出和写入的测试代码**

|  |
| --- |
| *unsigned char uh1, uh2;*  *unsigned long addr=256;*  *uh1 = 0x0f5;*  *write\_mem\_ubyte( addr, uh1 );*  *uh2 = read\_mem\_ubyte( addr );*  *print( “uh1 = %u, uh2 = %u\n”, uh1, uh2 );* |

## 1.4 完整的程序实例

代码4是根据本节前面的分析得到的一个C程序实例。

**代码4 一个完整的C程序，实现了无符号字节数的读写和测试**

|  |
| --- |
| *#include <stdio.h>*  *#include <stdlib.h>*  *char \*mem; //* 主存起始地址指针  *int mem\_size=0x080000; //* 主存的大小  *unsigned char read\_mem\_ubyte(long addr)*  *{*  *return \*((unsigned char \*)(mem+addr));*  *}*  *void write\_mem\_ubyte(long addr, unsigned char val)*  *{*  *\*((unsigned char \*)(mem+addr)) = val;*  *}*  *void main(void)*  *{*  *unsigned char uh1, uh2;*  *unsigned long addr=256;*  *mem = (char \*)malloc( sizeof(char) \* mem\_size );*  *if ( mem==NULL )*  *{*  *printf( “error: main memory allocation\n” );*  *exit( -1 );*  *}*  *uh1 = 0x0f5;*  *write\_mem\_ubyte( addr, uh1 );*  *uh2 = read\_mem\_ubyte( addr );*  *print( “uh1 = %u, uh2 = %u\n”, uh1, uh2 );*  *free( mem );*  *}* |

同学们可以编译并执行该程序，查看结果。

* C程序的编译和执行

打开终端窗口，执行下面的命令即可将代码1中的C程序（假设文件名为*fast.c*）编译为二进制可执行程序（假设文件名为*fast*）。

|  |
| --- |
| *gcc -O3 fast.c -o fast* |

通过下面的命令可以执行程序fast。

|  |
| --- |
| *./fast* |

# MIPS的C语言描述

本节以MIPS为例，介绍如何用C语言描述一个指令集的功能。

## 2.1 MIPS指令格式

*MIPS*指令有以下三种格式：

* *R*型指令

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *31 26* | *25 21* | *20 16* | *15 11* | *10 0* |
| *OPCODE* | *RS1* | *RS2* | *RD* | *FUNC* |

它有*5*个字段，*OPCODE*和*FUNC*表示操作码，*RS1*和*RS2*表示源操作数，*RD*表示目的操作数。

**注意：所有R型指令的*OPCODE*字段都是0，操作类型由*FUNC*字段决定。**

* I型指令

|  |  |  |  |
| --- | --- | --- | --- |
| *31 26* | *25 21* | *20 16* | *15 0* |
| *OPCODE* | *RS1* | *RS2* | *IMM/UIMM* |

它有*4*个字段，*OPCODE*表示操作码；*RS1*表示源操作数；*IMM（UIMM）*表示有（无）符号立即数，用作源操作数；*RD*表示目的操作数。

**注意：究竟使用*IMM*还是*UIMM*，由*OPCODE*决定。**

* J型

|  |  |
| --- | --- |
| *31 26* | *25 0* |
| *OPCODE* | *OFFSET* |

它有*2*个字段，*OPCODE*表示操作码；*OFFSET*表示目标地址偏移量。

## 2.2 MIPS指令的读取

*MIPS*的每条指令都是*32*位，可以用一个*32*位无符号整数（*unsigned int*）表示其二进制编码。调用*read\_mem\_uword*函数（*该函数在作业4.1中完成*）就可以从存储器中读出一条*32*位*MIPS*指令，如代码5所示，这里*pc*表示该指令的地址。

**代码5 从主存中读出一条*MIPS*指令**

|  |
| --- |
| *long pc;*  *unsigned int insn = read\_mem\_uword( pc );* |

## 2.3 MIPS指令的译码

*MIPS*采用**固定字段译码**技术，即译码时不考虑指令类型，而是同时得到所有字段（*OPCODE*、*RS1*、*RS2*、*RD*、*IMM/UIMM*、*FUNC*、*OFFSET*）的值。指令执行时根据指令类型选择并使用相应的字段。

用位操作和移位操作就可以得到以上所有字段的值，下面的代码6给出译码一条*MIPS*指令的*C*语句。

**代码6 *MIPS*指令的译码**

|  |
| --- |
| *int OPCODE = (insn >> 26) & 0x3f;*  *int RS1 = (insn >> 21) & 0x1f;*  *int RS2 = (insn >> 16) & 0x1f;*  *int RD = (insn >> 11) & 0x1f;*  *int IMM* *= (int)((short)(insn & 0xffff));* *// 这个表达式实现了符号扩展*  *unsigned int UIMM = (unsigned int)(insn & 0xffff);* *// 这个表达式实现了0扩展*  *int FUNC = insn & 0x7ff;*  *int OFFSET = insn & 0x3ffffff;* |

# MIPS指令的功能描述

## 3.1 声明OPCODE和FUNC字段的值

*MIPS*指令*OPCODE*和*FUNC*字段的值可以定义为枚举类型，如代码3所示，请注意阅读代码中的注释。

可以将代码6保存为文件名为*fast.h*的头文件，在*fast.c*中增加下面一行引用。

|  |
| --- |
| *#include “fast.h”* |

**代码6 指令操作码和FUNC字段的定义**

|  |
| --- |
| *enum opcodes {*  */\* All R-Type Insns are SPECIAL insns: Opcode = 0 \*/*  *OP\_SPECIAL = 0,*  */\* I-Type Insns. Max range: 1 - 63 \*/*  */\* I-Type ALU Insns. \*/*  *OP\_ADDI, OP\_ADDUI, …,*  */\* I-Type Branch and Jump Insns. \*/*  *OP\_BEQZ, OP\_BNEZ, …,*  */\* I-Type Load and Store Insns. \*/*  *OP\_LB, OP\_LBU, …,*  */\* J-Type Insns. \*/*  *OP\_J, …,*  */\* !!! Do \*NOT\* remove this entry !!!*  *This is used to delimit the end of the opcode array, so we can check*  *to see if you try to define more that 64 opcodes!*  *\*/*  *NUM\_OPCODES*  *};*  */\* MIPS only has 6 bits for the opcode, which isn't enough to specify*  *all of the implemented instructions. To work around this, several*  *instructions share the same "OP\_SPECIAL" opcode. To differentiate*  *these instructions, we use the 11 bit function field. The func values*  *for these instructions are defined here.*  *Max function range: 0 - 2047*  *\*/*  *enum functions {*  *OP\_ADD = 0, OP\_ADDU, …,*  */\* !!! Do \*NOT\* remove this entry !!!*  *This is used to delimit the end of the function array, so we can check*  *to see if you try to define more that 2048 functions!*  *\*/*  *NUM\_FUNCS*  *};* |

## 3.2 MIPS指令的功能

每条MIPS指令的功能都可以用一个函数实现，函数格式可以为：

|  |
| --- |
| *long INSN\_指令名(long pc)* |

这里的指令名就是代码3中的定义的枚举类型*opcodes*和*functions*的值。代码7给出了指令*ADD*、*ADDU*、*ADDI*、*ADDUI、NOP*的参考实现。可以看出，每条指令有两个功能：

* 完成计算，如上面四条指令就是加法运算；
* 修改*pc*，使*pc*指向下一条要执行的指令。显然，除了分支指令和跳转指令外，其他指令都会将*pc*加*4*。

**代码7 四条整数加法指令的功能描述**

|  |
| --- |
| *long INSN\_ADD(long pc)*  *{*  *int \*rd = r + RD;* *// R-型指令的目的寄存器为RD*  *int rs1 = get\_int( RS1 );* *// R-型指令的源操作数1为RS1，将其视作有符号数*  *int rs2 = get\_int( RS2 );* *// R-型指令的源操作数2为RS2，将其视作有符号数*  *\*rd = rs1 + rs2; // 可以直接将计算结果写回目的寄存器*  *return pc+4; // pc的值加4，指向顺序的下一条指令*  *}*  *long INSN\_ADDU(long pc)*  *{*  *unsigned int \*rd = r + RD;* *// I-型指令的目的寄存器为RD*  *unsigned int rs1 = get\_uint( RS1 );* *// I-型指令的源操作数1为RS1（无符号数）*  *unsigned int rs2 = get\_uint( RS2 );* *// I-型指令的源操作数2为RS2（无符号数）*  *\*rd = rs1 + rs2;*  *// 可以直接将计算结果写回目的寄存器*  *return pc+4;*   *// pc的值加4，指向顺序的下一条指令*  *}*  *long INSN\_ADDI(long pc)*  *{*  *int \*rd = r + RS2;* *// I-型指令的目的寄存器为RS2*  *int rs = get\_int( RS1 );* *// I-型指令的源操作数1为RS1，将其视作有符号数*  *\*rd = rs + IMM; // I-型指令的源操作数2为有符号数IMM*  *return pc+4; // pc的值加4，指向顺序的下一条指令*  *}*  *long INSN\_ADDUI(long pc)*  *{*  *unsigned int \*rd = r + RS2;* *// I-型指令的目的寄存器为RS2*  *unsigned int rs = get\_int( RS1 );* *// I-型指令的源操作数1为RS1（无符号数）*  *\*rd = rs + UIMM;*   *// I-型指令的源操作数2为无符号数UIMM*  *return pc+4;*   *// pc的值加4，指向顺序的下一条指令*  *}*  *long INSN\_NOP(long pc)*  *{*  *return pc+4;*  *}* |

## 3.3 MIPS程序的执行

MIPS程序的执行过程可以简单地描述为：

* 第一步，设置第一条指令的地址，即变量*pc*的初值；
* 第二步，判断*pc*是否为*0*，若*pc*为*0*则结束；
* 第三步，读取、译码并执行第一条指令；指令执行后，*pc*将指向下一条指令；然后转第二步。

我们可以用代码8中的函数*Execution*实现这一过程。这样，在main函数中调用*Execution*函数就可以执行一段*MIPS*程序了。

**代码8 Execution函数的C代码**

|  |
| --- |
| *void Execution(void)*  *{*  *pc = 0x1000; // 将程序的入口地址设为0x1000*  *for ( ; pc; )*  *{*  *unsigned int insn = read\_mem\_uword( pc ); // 读第一条指令*  *// 指令译码*  *if ( OPCODE ) // 若不是R型指令*  *{*  *switch ( OPCODE )*  *{*  *case OP\_ADDI: pc = INSN\_ADDI( pc ); break;*  *case OP\_ADDUI: pc = INSN\_ADDUI( pc ); break;*  *case OP\_NOP: pc = INSN\_NOP( pc ); break;*  *case OP\_JR: pc = INSN\_JR( pc ); break;*  *// 增加其他I型、J型指令的实现*  *default: printf( “error: unimplemented instruction\n” ); exit( -1 );*  *} // end switch( OPCODE )*  *} // end if ( OPCODE )*  *else*  *{*  *switch ( FUNC )*  *{*  *case OP\_ADD: pc = INSN\_ADD( pc ); break;*  *case OP\_ADDU: pc = INSN\_ADDU( pc ); break;*  *// 增加其他R型指令的实现*  *default: printf( “error: unimplemented instruction\n” ); exit( -1 );*  *} // end switch(FUNC)*  *} // end else*  *} // end for*  *} // end Execution()* |

## 3.5 测试所实现的指令功能

可以采用下面的方法测试自己实现的*MIPS*指令的功能是否正确。以*R*型指令*ADD R1, R2, R3*为例，这条指令的各字段如下：

*OPCODE = 0，RS1 = 00010，RS2 = 00011，RD = 00001，FUNC = 0*

因此，它的二进制编码为*000000 00010 00011 00001 00000000000*，用十六进制表示就是*0x00430800*。

我们还要使用的另一条测试指令是*JR R0*。由于*MIPS*的*R0*寄存器始终为*0*，执行这条指令，将把*pc*的值置为*0*，而pc为*0*正是*Execution*函数中*for*循环的结束条件。换句话说，这条指令可以作为程序的最后一条指令，表示程序结束。这条指令的十六进制编码为*0x8c000000*。代码9是指令*JR*功能的*C*语言描述。

**代码9 JR指令功能的实现函数**

|  |
| --- |
| *long INSN\_JR(long pc)*  *{*  *return get\_int( R1 );*  *}* |

我们将这两条指令的十六进制编码写入地址为*0x1000*和*0x1004*的字存储单元，然后将*pc*的初值设为*0x1000*，就可以执行这段仅有两条指令的代码了。代码10给出了实现这一功能的*C*程序。

**代码10 ADD指令的功能测试代码**

|  |
| --- |
| *int i1 = 16, i2 = -35, i3;*  *put\_int( 2, i1 ); // R2 = 16*  *put\_int( 3, i2 ); // R3 = -35*  *write\_mem\_uword( 0x1000, 0x00430800 ); // mem[0x1000] = ADD R1, R2, R3*  *write\_mem\_uword( 0x1004, 0x8c000000 ); // mem[0x1004] = JR R0*  *Execution(); // execution above two MIPS instructions*  *i3 = get\_int( 1 );*  *printf( “r1 = %d\n”, i3 ); // output the value of R1* |

# 实验

## 4.1 主存

请同学们仿照代码2编写访问主存，读写有符号字节（*read\_mem\_byte*）、有符号/无符号半字（*read\_mem\_halfword/read\_mem\_uhalfwor*）、有符号/无符号字（*read\_mem\_word/ read\_mem\_uword*）、单精度浮点（*read\_mem\_float*）、双精度浮点（r*ead\_mem\_double*）等类型数据的函数。并仿照代码3编写程序，测试所编写函数的正确性。

要求：将所编写的代码添加到代码1中，编译并执行。

## 4.2 寄存器文件

请完成的对主存的结构和功能的描述，用*C*语言描述通用寄存器文件。通用寄存器文件含有32个32位的通用寄存器，每个通用寄存器可以被独立访问。

下面给出一个实现了通用寄存器文件的参考C程序。红色为与通用寄存器文件有关的代码。请同学们自行阅读并理解这部分代码，并进行正确性测试，发现并改正其中可能存在错误。

|  |
| --- |
| *#include <stdio.h>*  *#include <stdlib.h>*  *char \*mem; //* 主存起始地址指针  *int mem\_size=0x080000; //* 主存的大小  *int ireg\_size = 32; //* 通用寄存器文件的大小  *int \*r; //* 通用寄存器文件起始地址指针  *unsigned char read\_mem\_ubyte(long addr)*  *{*  *return \*((unsigned char \*)(mem+addr));*  *}*  *void write\_mem\_ubyte(long addr, unsigned char val)*  *{*  *\*((unsigned char \*)(mem+addr)) = val;*  *}*  *int get\_int(long addr)*  *{*  *return \*((int \*)(r+addr));*  *}*  *void put\_int(long addr, int val)*  *{*  *\*((int \*)(r+addr)) = val;*  *}*  *int get\_unit(long addr)*  *{*  *return \*((unsigned int\*)(r+addr));*  *}*  *void put\_unit(long addr, unsigned int val)*  *{*  *\*((unsigned int \*)(r+addr)) = val;*  *}*  *void main(void)*  *{*  *unsigned char uh1, uh2;*  *unsigned long addr=256;*  *mem = (char \*)malloc( sizeof(char) \* mem\_size );*  *if ( mem==NULL )*  *{*  *printf( “error: main memory allocation\n” );*  *exit( -1 );*  *}*  *r = (int \*)malloc( sizeof(int) \* ireg\_size );*  *if ( r = NULL )*  *{*  *printf( “error: int. Register file allocation\n” );*  *exit( -1 );*  *}*  *uh1 = 0x0f5;*  *write\_mem\_ubyte( addr, uh1 );*  *uh2 = read\_mem\_ubyte( addr );*  *print( “uh1 = %u, uh2 = %u\n”, uh1, uh2 );*  *free( mem );*  *free( r );*  *}* |