计算机体系结构

技术报告

姓名： 陈 洋

学号： 51255902034

专业：软件科学与技术

2023年1月6日

一、项目要求

本项目主要分为MIPS的反汇编（Disassembler）、非流水线模拟（Non-pipelined simulator）和流水线模拟（Pipelined simulator）三个部分。

1.1 反汇编（Disassembly）

反汇编器（Disassembler）可以实现将输入的MIPS二进制指令转化为等价的汇编代码。

支持的MIPS指令分为以下两类：

|  |  |
| --- | --- |
| **Category-1** | **Category-2** |
| \* J, JR, BEQ, BLTZ, BGTZ  \* BREAK  \* SW, LW  \* SLL, SRL, SRA  \* NOP | \* ADD, SUB  \* MUL  \* AND, NOR  \* SLT |

第一类指令遵循原MIPS指令集的编码规则，对第二类指令编码进行重定义：

31 30 26 25 21 20 16 15 11 10 6 5 0

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Imm 0  (1 bit) | Opcode  (5 bits) | rs  (5 bits) | rt  (5 bits) | rd  (5 bits) | 0  00000 | Function  (6 bits) |

1.2 非流水线模拟（Non-pipelined Simulation）

非流水线模拟器（Non-pipelined simulator）可以实现对MIPS指令的模拟，包含每个周期执行指令后寄存器和内存的状况，每条指令运行时间均为一个周期。

1.3流水线模拟（Pipelined Simulation）

流水线模拟器（Pipelined Simulator）可以实现对MIPS指令的流水线模拟，包含每个周期结束时处理器和内存的状态。

1.4主要项目要求

R1：反汇编器

R1.1：将两类MIPS二进制指令转化为汇编指令，在同一行同时输出原二进制指令（按照要求按位分隔）、对应指令地址和转化的汇编指令。

R1.2：将BREAK指令后的32位符号整数转化为十进制输出。

R2：非流水线模拟器

R2.1：输出周期序号、该周期执行的指令及指令地址。

R2.2：输出每周期结束时寄存器和内存的状态。

R3：流水线模拟器

R3.1：运用基本Scoreboard算法模拟流水线处理器。

R3.2：输出每周期处理器各部分（IF Unit、各Buffer和各Queue）的状态。

R3.3：输出每周期结束时寄存器和内存的状态。

二、项目概要设计

该项目主要分为反汇编（part1）、无流水线模拟（part2）、流水线模拟（part3）三个主要模块。其中，反汇编部分主要为解码功能（decode\_instruction），无流水线模拟部分主要为执行指令函数（execute\_instruction）、存储函数（store）和载入函数（load）。流水线模拟部分将指令执行过程分为五个阶段。此外，utils文件中包含辅助函数，main函数调用三部分函数进行整体模拟。整体架构设计见图3-1。

2.1 反汇编文件（part1.c）函数功能

2.1.1 print类函数（S1.1）

print相关函数的功能是按照utils.h文件中规定的格式输出汇编形式指令，其输入为一个字符串和一条需要反汇编的指令。主要有print\_rtype、print\_itype、print\_branch、print\_load、print\_store、print\_break和print\_nop函数。

2.1.2 write类函数

write类函数对输入的指令进行解析，判断指令类型，调用print类函数输出。

2.1.3 decode\_instruction函数

decode\_instruction函数根据指令的opcode部分进行粗分类，再调用write类函数解码。

2.2 非流水线模拟文件（part2.c）函数功能

2.2.1 execute类函数

execute\_rtype、execute\_itype等六条函数分别对六类opcode不同的指令进行运算或内存读取和存储操作。

2.2.2 内存操作函数

load和store函数根据地址分别进行对内存的读取和写入操作，中间需要调用check函数来检查是否对齐。这两个函数被execute\_load和execute\_store函数调用。

2.2.3 execute\_instruction函数

该函数根据不同的opcode分别调用六个execute类函数模拟执行指令。

2.3流水线模拟文件（part3.c）函数功能

流水线模拟部分主要有更新（update）、指令获取（IF）、指令发射（Issue）、执行（Exe）和写回（WB）等功能函数，由于要求输出每个周期结束时的状态，逆序执行这些函数。

Exe函数的功能是执行三个post buffer里指令；Exe函数将三个pre queue中符合要求的一条指令转移到对应的post buffer中；Issue函数应用Scoreboarding算法，将pre-issue buffer中的指令转移到对应的pre队列中；IF函数的功能是最多获取两条指令，放入pre-issue buffer中，update函数用来更新所有原件状态。

三、项目详细设计

该项目的整体架构如图3-1的类图所示。

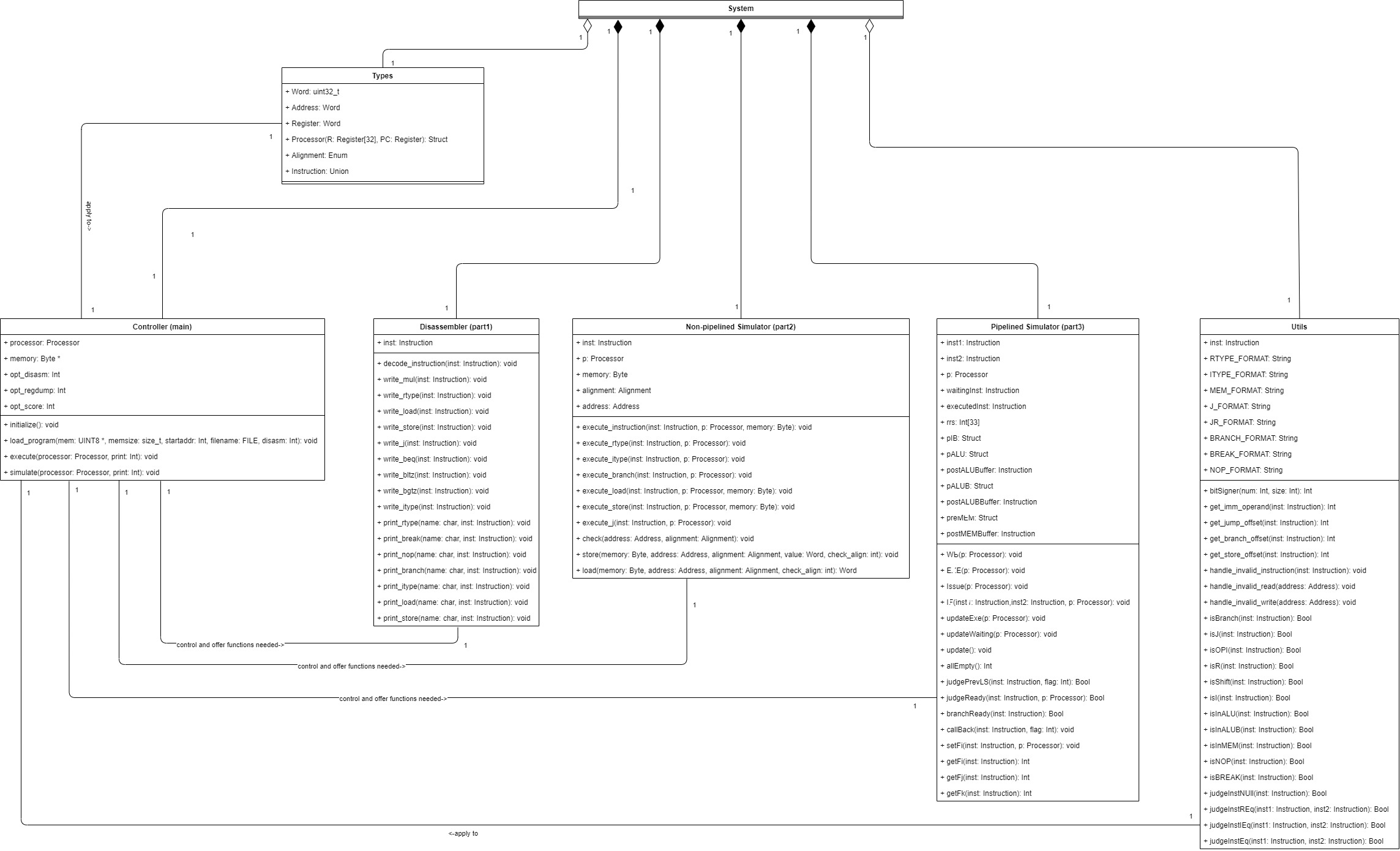


图3-1 项目架构设计类图

对于读入文件的每一行二进制指令转换为长整数类型，从低地址到高地址按字节依次存储到模拟内存的数组mem中。

利用联合体成员共用一块内存空间的特性，使用union来存储每条指令。在types.h中，定义联合体Instruction，包含opcode、itype、rtype、jtype四个struct，struct opcode中分为除26位的非opcode部分和6位的opcode部分，struct rtype将32位从低地址到高地址依次划分为6位的funct、5位的shamt、5位的rd、5位的rt、5位的rs和6位的opcode，struct itype将32位依次划分为16位的offset、5位的rt、5位的rs和6位的opcode，struct jtype将32位依次划分为26位的imm和6位的opcode。

由此，存储的指令inst可使用inst.opcode.opcode来获取其最高的6位，若其为rtype划分类型指令，可使用inst.rtype.shamt来获取6至10位。

同时，types.h文件中还定义了Word类型为uint32\_t，Address类型为Word，Register类型为Word，用struct Processor来模拟寄存器，包括一个包含32个Register类型元素的数组R和一个Register类型的PC。

由此，可以使用p -> R[inst.rtype.rs]来获取rtype类型指令inst的rs部分代表的寄存器所存储的值，其余类似操作进行相似的处理。

此外，定义内存大小为1024\*1024（1 MByte）。

3.1 反汇编（part1.c）（S1）

读取文件中的指令，若读到的指令instruction代表BREAK指令，即

((instruction & 0xFF) == 0x0d) && (((instruction >> 24) & 0xFF) == 0x00)，

那么该指令后的每一行二进制表示32位整数，直接输出原二进制、地址和表示的十进制数。

在读到BREAK指令及其之前，对于每一行输入的指令进行解码操作，按格式输出原二

进制指令和地址，并调用part1.c中的decode\_instruction函数输出对应的汇编代码，原输入文件和得到输出汇编文件比对如图3-2所示。

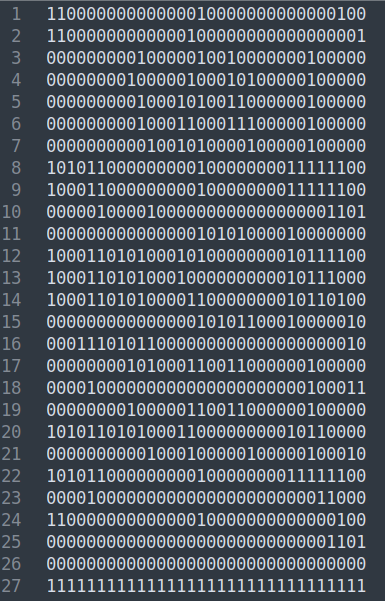
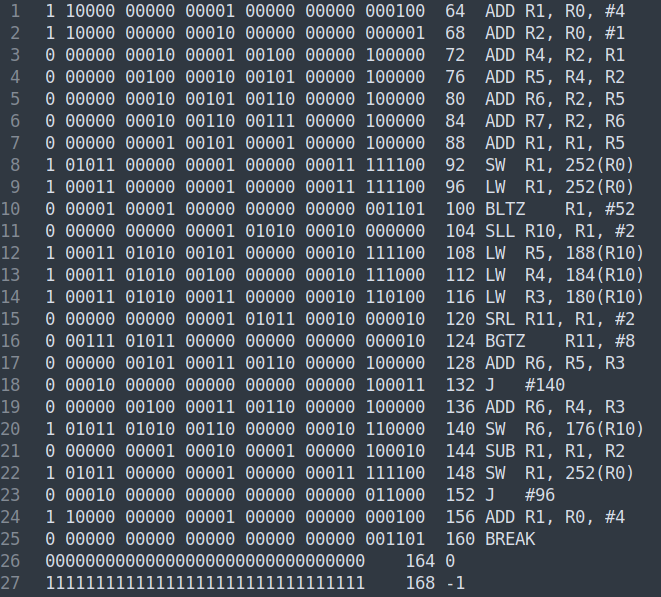
 

图3-2 输入二进制文件（左）与输出汇编形式（右）比对

3.1.1 decode\_instruction函数（S1.1）

decode\_instruction函数根据每条指令的opcode判断调用哪条write类函数。例如，若指令inst为011100 00011 00100 00101 00000 000010，判断其opcode为0x1c，是MUL指令，则调用write\_mul(inst, index)，其中，index为选取输出格式的检索。

3.1.2 write类函数（S1.2）

由decode\_instruction函数调用的write类函数主要通过指令的具体其它部分来判断指令类型，调用print类函数输出转换的汇编指令。由于decode\_instruction函数中已经对opcode进行了粗分类，write类函数中可省去对opcode的判断。例如，write\_load函数只需调用print\_load函数，而write\_rtype函数需要根据指令的funct进行进一步判断，获取指令类型后将其传输给print类函数进行输出。

对于write\_rtype函数，若funct部分为0x08，表明其为JR操作，将字符串“JR”传输

进print\_rtype函数，若shamt部分为0x00且funct部分为0x20，则表明其为ADD操作，将字符串“ADD”传输进print\_rtype操作。

void write\_rtype(Instruction inst, int index) {

……

else if ( inst.rtype.funct == 0x08 ) print\_rtype("JR", inst, index);

else {

if ( inst.rtype.shamt == 0 ) {

/\* add \*/

if ( inst.rtype.funct == 0x20 ) print\_rtype("ADD", inst, index);

/\* and \*/

else if ( inst.rtype.funct == 0x24 ) print\_rtype("AND", inst, index);

……

else handle\_invalid\_instruction(inst);

}

……

else handle\_invalid\_instruction(inst);

}

}

3.1.3 print类函数（S1.3）

在utils.h文件中定义了各指令的输出形式如下：

#define RTYPE\_FORMAT "%s\tR%d, R%d, R%d\n"

#define ITYPE\_FORMAT "%s\tR%d, R%d, #%d\n"

#define MEM\_FORMAT "%s\tR%d, %d(R%d)\n"

#define UTYPE\_FORMAT "%s\tR%d, #%d\n"

#define J\_FORMAT "J\t#%d\n"

#define JR\_FORMAT "JR\tR%d\n"

#define BRANCH\_FORMAT "%s\tR%d, R%d, %d\n"

#define BREAK\_FORMAT "BREAK\n"

#define NOP\_FORMAT "\n"

NOP指令使用NOP\_FORMAT，BREAK指令使用BREAK\_FORMAT，JR指令使用JR\_FORMAT，J指令使用J\_FORMAT，BLTZ、BGTZ指令使用UTYPE\_FORMAT，LW和SW指令使用MEM\_FORMAT，ADD（i）、SUB（i）、MUL（i）、AND（i）、NOR（i）、SLT、BEQ、SLL、SRL、SRA等指令使用ITYPE\_FORMAT，ADD、AND、SUB、SLT、MUL、NOR等指令使用RTYPE\_FORMAT。

print类函数主要有print\_rtype、print\_itype、print\_branch、print\_load、print\_store、

print\_break和print\_nop函数。输入为一条指令inst和一个字符串name。name传入的是该指令的指令名，通过解析指令的rt、rd、funct等部分获取应该使用的输出形式。

例如，print\_itype函数为：

void print\_itype(char \*name UNUSED, Instruction inst UNUSED, int index) {

if (index == 1)

printf(ITYPE\_FORMAT, name, inst.itype.rt, inst.itype.rs, bitSigner(inst.itype.offset,16));

else if (index == 2)

printf(ITYPE\_FORMAT2,name,inst.itype.rt,inst.itype.rs, bitSigner(inst.itype.offset,16));

}

由于两个项目要求的输出格式不同，此处对每一种输出形式定义了两种输出样式，用index来检索。当decode\_instruction和write类函数根据opcode、funct等判断该指令为itype类型时，会调用print\_itype函数，当指令为11000000000000100000000000000101，opcode部分110000为0x30，表示其为一个重定义的ADD指令，调用print\_itype("ADD", inst, index)：

/\* addi \*/

case(0x30):

{

print\_itype("ADD", inst, index);

break;

}

此时，传入print\_itype中的name为“ADD”，进入函数后，按照ITYPE\_FORMAT格式输出，ITYPE\_FORMAT的定义为"%s\tR%d, R%d, #%d\n"，%s将被替代为ADD，第一个%d被替代为指令的rt部分00010即2，第二个%d被替代为指令的rs部分00000即0，第三个%d被替代为bitSigner(inst.itype.offset,16)，即offset部分00000000101进行符号位扩展后得到的整数5，那么最终输出的汇编指令为ADD R2, R0, #5。

3.2 非流水线模拟（part2.c）（S2）

非流水线模拟依次读取指令进行操作。该部分的主要函数是execute\_instruction函数。将指令粗分为rtype、itype、branch、j、load和store六类，分别进行操作。

3.2.1 execute\_instruction函数（S2.1）

对于execute\_instruction函数的输入指令inst、处理器指针p以及内存指针memory，首先实行switch(inst.opcode.opcode)来判断指令类型。若opcode部分为0x00（ADD、AND、SUB、SLL、SRL、SRA、SLT、NOR、BREAK、JR、NOP）或0x1c（MUL），执行execute\_rtype(inst, p)；若opcode部分为0x21（MULI）、0x30（ADDI）、0x31（SUBI）、0x32（ANDI）、0x33（NORI）或0x35（SLTI），执行execute\_itype(inst, p)；若为0x2b（SW），执行execute\_store(inst, p, memory)；若为0x23（LW），执行execute\_load(inst, p, memory)，若为0x04（BEQ）、0x01（BLTZ）或0x07（BGTZ），执行execute\_branchexecute\_branch(inst, p)；若为0x02（J），执行execute\_j(inst, p)；若为未定义指令，执行handle\_invalid\_instruction(inst)报错并终止程序。

3.2.1 execute类函数（S2.2）

execute\_rtype函数处理ADD、MUL、SRL、SUB、SLL、NOP、SLT、SRA、AND、NOR、JR和BREAK指令。通过switch(inst.rtype.funct)，判断具体指令类型，并进行计算，操作时注意将寄存器中的值转化为32位整数。操作结束后，PC直接加四，指向下一条指令。例如，若funct部分为0x20，表明其为ADD操作指令，通过处理器指针p获取涉及操作的寄存器，将rs和rt部分表示的寄存器中存储的值相加，结果存储到rd编号的寄存器中，进行的操作为：

p -> R[inst.rtype.rd] = (sWord)p -> R[inst.rtype.rs] + (sWord)p -> R[inst.rtype.rt];

execute\_itype函数处理重定义的ADD、SUB、MUL、AND、NOR和SLT操作，计算后同样PC直接加四指向下一条指令，在调取指令offset部分计算时需要注意符号扩展，例如这里重定义的ADD操作为：

p -> R[inst.itype.rt] = (sWord)(p -> R[inst.itype.rs] + (signed)bitSigner(inst.itype.offset, 16));

execute\_branch函数与前两个函数类似，这里区分不同branch操作只需读取opcode部分进行判断。需要注意的是，若指令判断后决定branch，PC需加上offset表示的跳转大小，同时仍需加四。例如，BEQ指令需要进行的操作为：

if ( (signed)(p -> R[inst.itype.rs]) == (signed)(p -> R[inst.itype.rt]) )

branchaddr += ((sHalf)get\_branch\_offset(inst)+4);

else

branchaddr += 4;

break;

execute\_j函数处理J指令，PC直接指向目标地址：

nextPC = ((((p->PC+4)>>28) & 0xf) << 28) | (signed)(bitSigner(get\_jump\_offset(inst), 32));

p -> PC = nextPC;

execute\_load和execute\_store函数涉及对内存的操作，需要分别调用load和store函数，执行完成后PC同样加四，指向下一条指令。execute\_load函数通过

address = (sWord)p -> R[inst.itype.rs] + bitSigner(inst.itype.offset, 16);

word = load(memory, address, LENGTH\_WORD, 0);

获取需要内存地址，并取出该地址中存储的内容，随后通过

p -> R[inst.itype.rt] = (sWord)word;

将读取的内容装在到目标寄存器中。

execute\_store函数通过

address = (sWord)p -> R[inst.itype.rs] + (sWord)get\_store\_offset(inst);

store(memory, address, LENGTH\_WORD, (sWord)p -> R[inst.itype.rt], 0);

获取将要存储数据的内存地址，再通过store函数将指定寄存器中的值存储到目标内存地址中。

3.2.3 内存操作函数load和store（S2.3）

load和store函数获取指定地址和存储内容的类型（大小），首先判断是否对齐以及地址是否超出最大内存空间，由于此项目实现的LOAD和STORE指令均针对字，即仅实现了LW和SW，输入类型在实际操作中没有使用到。

store函数要获取要存储的值value，将value按字节存储到用数组memory模拟的内存中：

memory[address] = (value & 0xff);

memory[address + 1] = ((value >> 8) & 0xff);

memory[address + 2] = ((value >> 16) & 0xff);

memory[address + 3] = ((value >> 24) & 0xff);

load函数从读入地址开始获取四个字节，作为需要载入到寄存器中的值：

Word data UNUSED = 0;

data |= memory[address];

data |= (memory[address + 1] << 8);

data |= (memory[address + 2] << 16);

data |= (memory[address + 3] << 24);

3.3流水线模拟（part3.c）（S3）

该部分每周期的输出格式形如图3-3。由于输出每周期各执行块和内存的状况为周期结束时的情况，在实现时倒序操作，并利用函数update，在一个执行流程结束后再更新队列长度等参数。

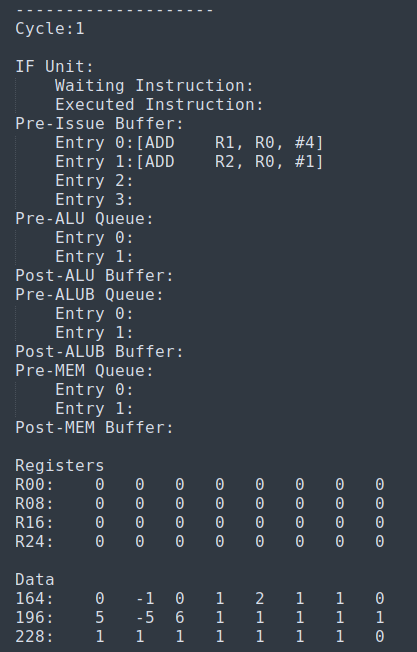


图3-3周期结束时各部件状态输出形式

这里使用结构体来模拟Pre-Issue Buffer和三个pre队列，每个结构体中都包含指令数组、数组长度（len）和数组暂时长度（tlen）三个变量。数组长度在执行完处理器各个组件后一次性更新，数组暂时长度用于记录执行过程中数组长度的变化，在最后赋值给数组长度。使用数组rrs来记录寄存器占用情况，若有指令正在对寄存器i进行写操作，那么rrs[i]=1，反之rrs[i]=0，在指令执行完成后将其写入寄存器的rrs值置零。

3.3.1 WB函数（S3.1）

WB函数的输入为处理器指针p，它用来处理Post-ALU Buffer、Post-ALUB Buffer和Post-MEM Buffer中的指令。

给定一张表，若post buffer中存在非NOP指令，执行该指令并在其后清空该buffer。执行指令时调用part2.c部分的execute类函数，由于在execute类函数中自动将PC加四，这里将PC减四归位。同时，记录下该指令写入的寄存器Fi，在更新完这一阶段状态后再在数组rrs中解除其占用状况。

3.3.2 Exe函数（S3.2）

Exe函数的作用是将三个pre队列中的指令分别放入三个post缓存中。

如果Pre-MEM Queue中的指令为SW指令，不做任何操作。

对于Pre-ALU Queue中的指令，如果Post-ALU Buffer为空，将队列中第一条指令放入post缓存，更新Pre-ALU Queue结构体的参数tlen（减少1），len是队列的实际长度，为了防止发生碰撞，将更新的队列大小暂存tlen中，在整张表执行这一轮执行完成后统一更新参数len。

对于Pre-ALUB Queue中的指令，需要执行两个周期的时间，因此利用变量cc来记录。执行的时间，只有当cc等于2时才将pre队列中的指令pop进post缓存中。

对于Pre-MEM Queue中的指令，除最初判断的SW指令，操作和Pre-ALU Queue相同。

3.3.3 Issue函数（S3.3）

Issue函数的功能是运用Scoreboarding算法将Pre-Issue Buffer中的指令放入三个pre队列中。为了方便叙述，这里称Pre-Issue Buffer为PIB。

若Pre-MEM Queue中的第一条指令为SW指令，直接调用part2中的execute\_instruction函数，并将其从Pre-MEM Queue中pop掉。这源于SW指令不会传入post buffer，因此在该函数内执行。

对于PIB中的每一条指令，如果该指令所有需要读取的寄存器都已就位（在rrs中为0），且其对应的pre队列未满，将其pop到pre队列中，并将其写入寄存器的rrs值置为1。如果存在写入寄存器未就绪，跳过该指令判断PIB中的下一条指令，同时，若该未就绪指令为LW操作，也需设置其Fi的rrs值为1，因为其可能需要修改写入寄存器的值被后续指令使用。

3.3.4 IF函数（S3.4）

该函数用于每次最多获取两条指令，存入Pre-Issue Buffer。

首先需要判断PIB空余的位置数。若PIB已满，该周期不获取新指令；若有一空，获取一条指令；若有不少于两个空位，获取两条新指令。

在获取一条新指令时，若其非BREAK、Branch指令或J指令，直接填入PIB，PC加四；若为J指令，直接执行；若为Branch，填入Waiting Instruction。

在获取两条新指令时，需分别判断。若为普通指令，皆填入PIB；若第一条为普通指令，第二条为Branch指令，第一条填入PIB，第二条填入Waiting Instruction；若第一条为普通指令，第二条为J指令，第一条填入PIB，第二条直接执行。若第一条为J指令，该周期不再读取第二条指令，直接执行J操作；若第一条为Branch指令，该周期不再读取第二条指令，将该Branch指令填入Waiting Instruction；若第二条是BREAK指令，第一条填入PIB，但执行BREAK操作，终止读入；若第一条是BREAK指令，如果此时尚有指令在队列和buffer中未执行完成，BREAK操作填入Waiting Instruction，等其之前指令全部完成再终止程序。

3.3.5 updateExe函数（S3.5）

该函数用于修改Executed Instruction中的值，若指令是已经执行完成的状态，则清空。

3.3.6 updateWaiting函数（S3.6）

该函数用于修改Waiting Instruction中的指令，若指令可以开始执行，执行它并将其转入Executed Instruction，Waiting Instruction清空。

3.3.7 update函数（S3.7）

该函数用于一次更新队列大小、数组rrs值变化等。

四、项目测试

4.1 Makefile文件及解释

该项目的Makefile文件为：

SOURCES := utils.c part1.c part2.c part3.c main.c

HEADERS := types.h utils.h main.h

ASM\_TESTS := sample single

all: MIPSsim part1 part2 part3

@echo "=============All tests finished============="

.PHONY: part1 part2 part3 %\_disasm %\_simulate

MIPSsim: $(SOURCES) $(HEADERS) out

gcc -g -Wpedantic -Wall -Wextra -Werror -std=c89 -o $@ $(SOURCES)

out:

@mkdir -p ./mipscode/out

# Part 1 Tests

part1: MIPSsim $(addsuffix \_disasm, $(ASM\_TESTS))

@echo "---------Disassembly Tests Complete---------"

%\_disasm: mipscode/code/%.input mipscode/ref/%\_disasm.solution MIPSsim

@./MIPSsim -d $< > mipscode/out/disasm.output

@diff -b -w -B $(word 2, $^) mipscode/out/disasm.output && echo "$@ TEST PASSED!" || echo "$@ TEST FAILED!"

# Part 2 Tests

part2: MIPSsim $(addsuffix \_execute, $(ASM\_TESTS))

@echo "-----------Execute Tests Complete-----------"

%\_execute: mipscode/code/%.input mipscode/ref/%\_trace.solution MIPSsim

@./MIPSsim -t $< > mipscode/out/trace.output

@diff -b -w -B $(word 2, $^) mipscode/out/trace.output && echo "$@ TEST PASSED!" || echo "$@ TEST FAILED!"

# Part 3 Tests

part3: MIPSsim $(addsuffix \_simulate, $(ASM\_TESTS))

@echo "-----------Simulation Tests Complete-----------"

%\_simulate: mipscode/code/%.input mipscode/ref/%\_score.solution MIPSsim

@./MIPSsim -r $< > mipscode/out/score.output

@diff -b $(word 2, $^) mipscode/out/score.output && echo "$@ TEST PASSED!" || echo "$@ TEST FAILED!"

clean:

rm -f MIPSsim

rm -rf mipscode/out

-d，-t，-r分表表示输出反汇编文件、非流水线处理器模拟文件和流水线处理器模拟文件。

4.2 测试方法及结果

将原二进制指令文件存储在mipscode/code文件夹下，参考正确模拟文件放于mipscode/ref文件夹下，按特定方式命名，程序输出文件存储在mipscode/out文件夹下。其具体命名方式可见以下项目代码结构图4-1。

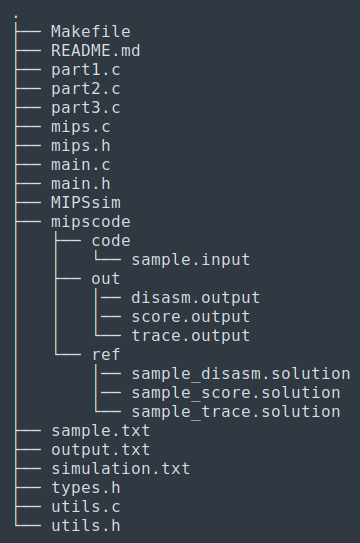


图4-1 项目代码结构

直接在终端键入make可一次执行三项模拟工作，并自动完成比对。由于规定的格式有稍许差别，使用-b -w -B来忽略空格、空行的差别。若显示TEST PASSED，说明执行成功且比对一致，如图4-2所示。（T1）

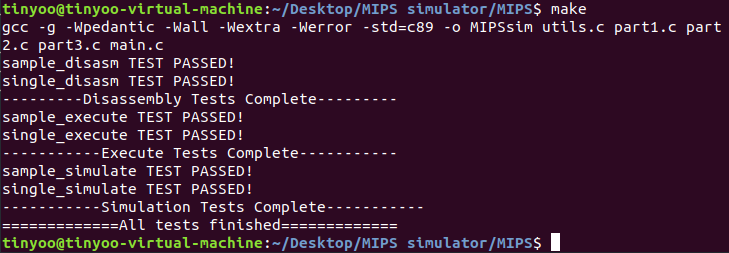


图4-2 make结果

也可在终端输入make进行编译，再键入

./MIPSsim sample.txt -r &> output.txt

其中，sample.txt是输入的二进制指令文件，output.txt表示将输出存储在文件output.txt中，-r表示输出流水线处理器模拟文件，也可换成-d输出转换的汇编文件，或-t输出非流水线处理器模拟文件。

随后，可键入

diff -b output.txt simulation.txt

来比较输出文件和正确的参考输出文件simulation.txt。

4.3 图表

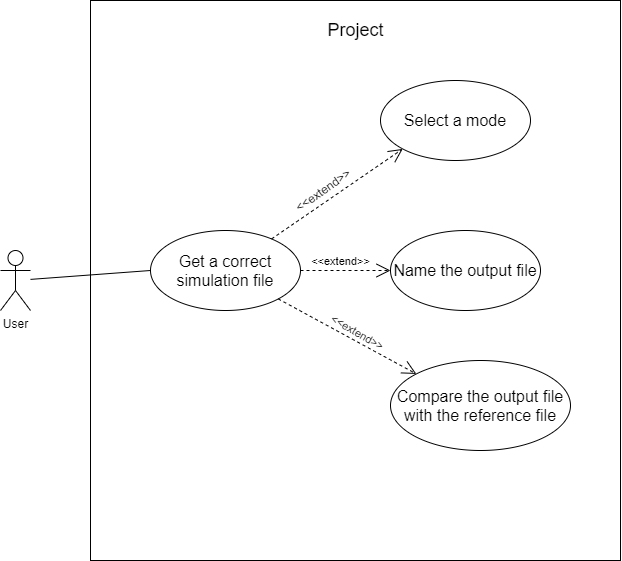


图4-3 Use Case Diagram

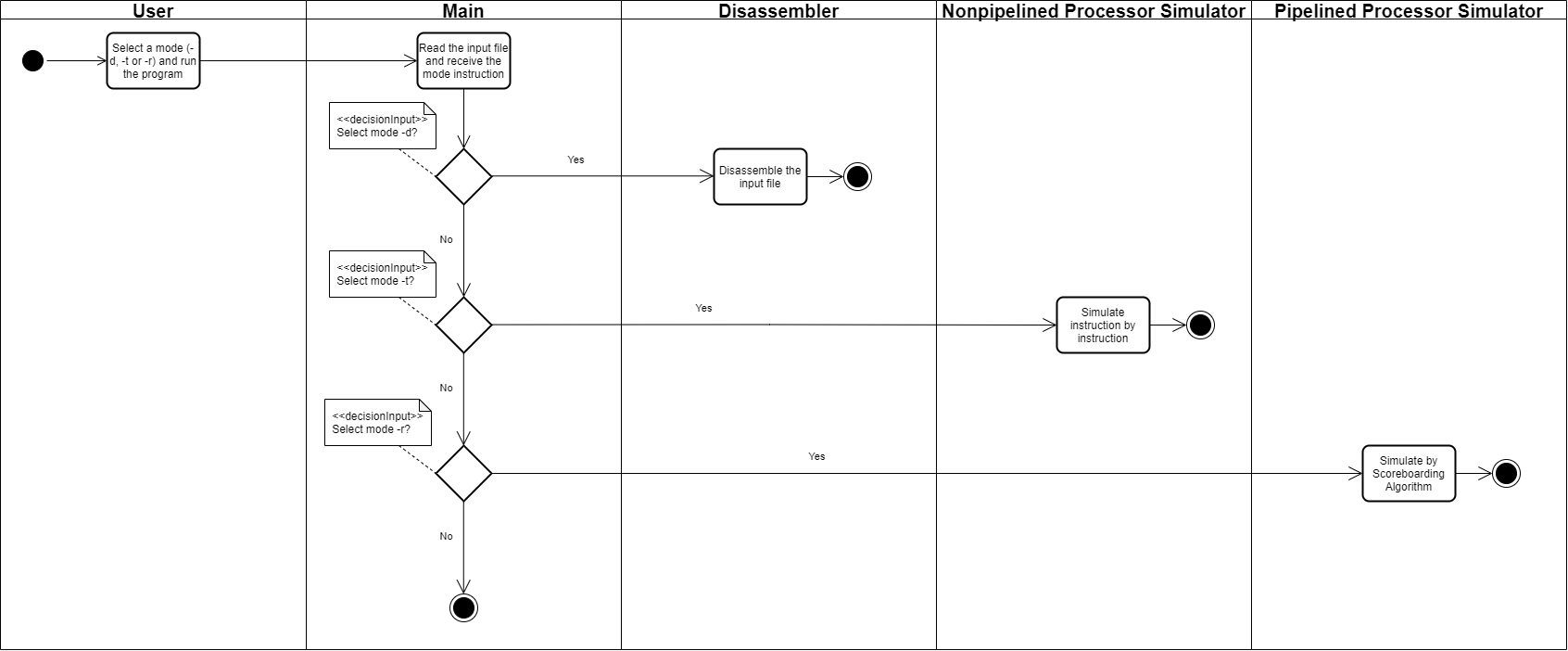


图4-4 Activity Diagram

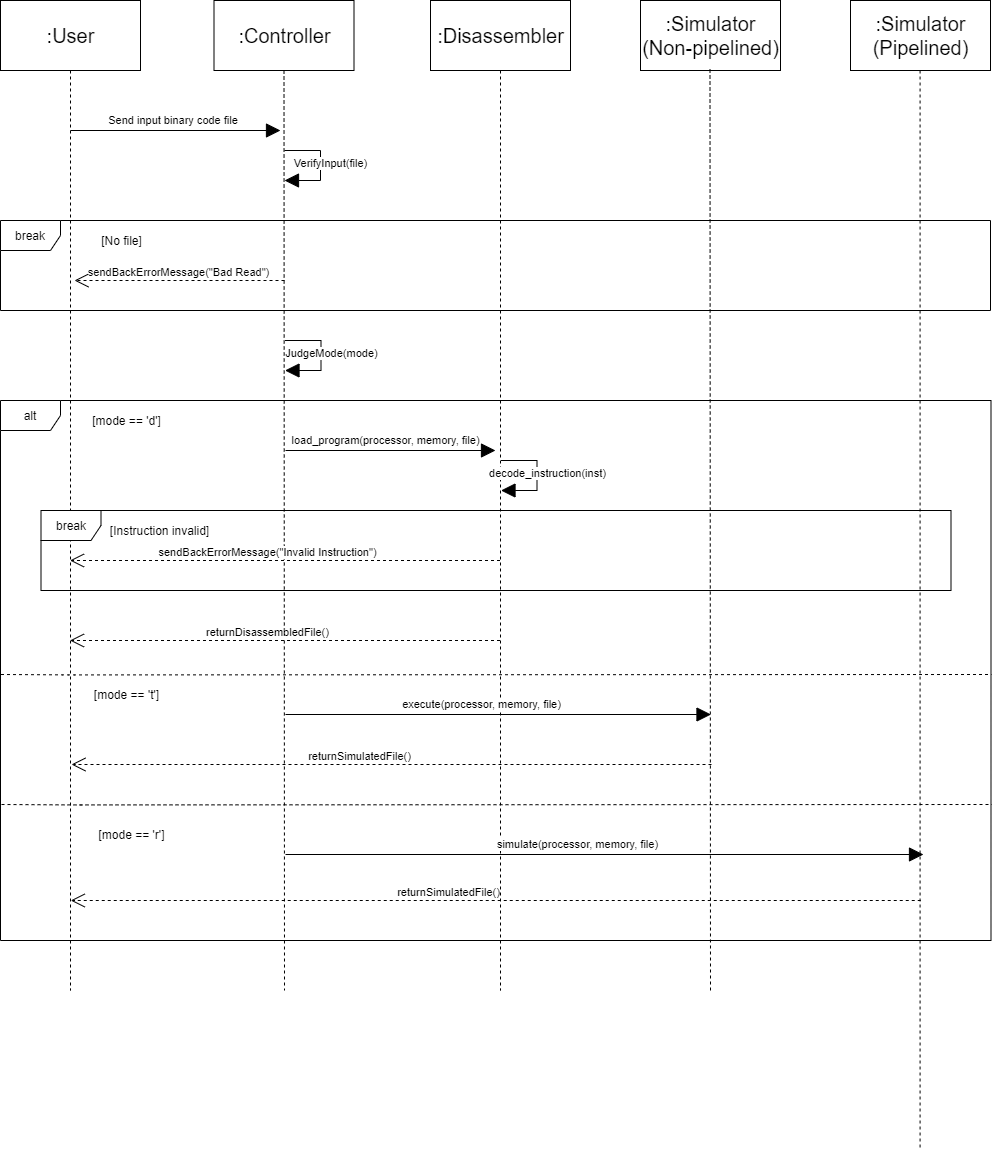


图4-5 Sequence Diagram