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。

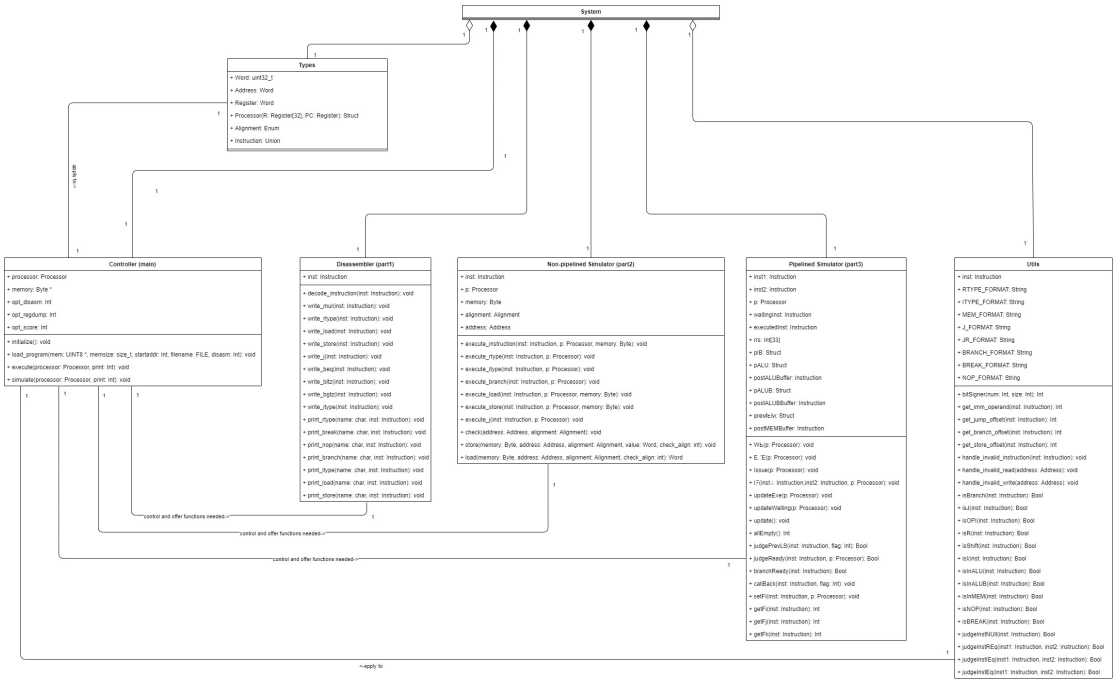

图 3-1 项目架构设计类图

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。

```
1  11000000000000001000000000000100
2  11000000000000010000000000000001
3  00000000010000010010000000100000
4  00000000100000100010100000100000
5  00000000010001010011000000100000
6  00000000100011000111000000100000
7  00000000001001010000100000100000
8  10101100000000001000000011111100
9  10001100000000001000000011111100
10 00000100001000000000000000001101
11 00000000000000001010100010000000
12 10001101010001010000000010111100
13 10001101010001000000000010111000
14 10001101010000110000000010110100
15 00000000000000001010110010000010
16 00011101011000000000000000000010
17 00000000101000011001100000100000
18 00001000000000000000000000100011
19 00000000100000110011000000100000
20 10101101010001100000000010110000
21 00000000001000100001000000100010
22 10101100000000001000000011111100
23 00001000000000000000000000011000
24 11000000000000001000000000000100
25 00000000000000000000000000001101
26 00000000000000000000000000000000
27 11111111111111111111111111111111
```

```
1   1 10000 00000 00001 00000 00000 000100  64  ADD R1, R0, #4
2   1 10000 00000 00010 00000 00000 000001  68  ADD R2, R0, #1
3   0 00000 00010 00001 00100 00000 100000  72  ADD R4, R2, R1
4   0 00000 00100 00010 00101 00000 100000  76  ADD R5, R4, R2
5   0 00000 00010 00101 00110 00000 100000  80  ADD R6, R2, R5
6   0 00000 00010 00110 00111 00000 100000  84  ADD R7, R2, R6
7   0 00000 00001 00101 00001 00000 100000  88  ADD R1, R1, R5
8   1 01011 00000 00001 00000 00011 111100  92  SW  R1, 252(R0)
9   1 00011 00000 00001 00000 00011 111100  96  LW  R1, 252(R0)
10  0 00001 00001 00000 00000 00000 001101  100 BLTZ    R1, #52
11  0 00000 00000 00001 01010 00010 000000  104 SLL R10, R1, #2
12  1 00011 01010 00101 00000 00010 111100  108 LW  R5, 188(R10)
13  1 00011 01010 00100 00000 00010 111000  112 LW  R4, 184(R10)
14  1 00011 01010 00011 00000 00010 110100  116 LW  R3, 180(R10)
15  0 00000 00000 00001 01011 00010 000010  120 SRL R11, R1, #2
16  0 00111 01011 00000 00000 00000 000010  124 BGTZ    R11, #8
17  0 00000 00101 00011 00110 00000 100000  128 ADD R6, R5, R3
18  0 00010 00000 00000 00000 00000 100011  132 J   #140
19  0 00000 00100 00011 00110 00000 100000  136 ADD R6, R4, R3
20  1 01011 01010 00110 00000 00010 110000  140 SW  R6, 176(R10)
21  0 00000 00001 00010 00001 00000 100010  144 SUB R1, R1, R2
22  1 01011 00000 00001 00000 00011 111100  148 SW  R1, 252(R0)
23  0 00010 00000 00000 00000 00000 011000  152 J   #96
24  1 10000 00000 00001 00000 00000 000100  156 ADD R1, R0, #4
25  0 00000 00000 00000 00000 00000 001101  160 BREAK
26  00000000000000000000000000000000    164 0
27  11111111111111111111111111111111    168 -1
```

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。

```
--------------------
Cycle:1

IF Unit:
    Waiting Instruction:
    Executed Instruction:
Pre-Issue Buffer:
    Entry 0:[ADD    R1, R0, #4]
    Entry 1:[ADD    R2, R0, #1]
    Entry 2:
    Entry 3:
Pre-ALU Queue:
    Entry 0:
    Entry 1:
Post-ALU Buffer:
Pre-ALUB Queue:
    Entry 0:
    Entry 1:
Post-ALUB Buffer:
Pre-MEM Queue:
    Entry 0:
    Entry 1:
Post-MEM Buffer:

Registers
R00:    0   0   0   0   0   0   0   0
R08:    0   0   0   0   0   0   0   0
R16:    0   0   0   0   0   0   0   0
R24:    0   0   0   0   0   0   0   0

Data
164:    0   -1  0   1   2   1   1   0
196:    5   -5  6   1   1   1   1   1
228:    1   1   1   1   1   1   1   0
```

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。

```
.
├── Makefile
├── README.md
├── part1.c
├── part2.c
├── part3.c
├── mips.c
├── mips.h
├── main.c
├── main.h
├── MIPSsim
├── mipscode
│   ├── code
│   │   └── sample.input
│   ├── out
│   │   ├── disasm.output
│   │   ├── score.output
│   │   └── trace.output
│   └── ref
│       ├── sample_disasm.solution
│       ├── sample_score.solution
│       └── sample_trace.solution
├── sample.txt
├── output.txt
├── simulation.txt
├── types.h
├── utils.c
└── utils.h
```

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）

```
tinyoo@tinyoo-virtual-machine:~/Desktop/MIPS simulator/MIPS$ make
gcc -g -Wpedantic -Wall -Wextra -Werror -std=c89 -o MIPSsim utils.c part1.c part
2.c part3.c main.c
sample_disasm TEST PASSED!
single_disasm TEST PASSED!
---------Disassembly Tests Complete---------
sample_execute TEST PASSED!
single_execute TEST PASSED!
-----------Execute Tests Complete-----------
sample_simulate TEST PASSED!
single_simulate TEST PASSED!
-----------Simulation Tests Complete-----------
=============All tests finished=============
tinyoo@tinyoo-virtual-machine:~/Desktop/MIPS simulator/MIPS$
```

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

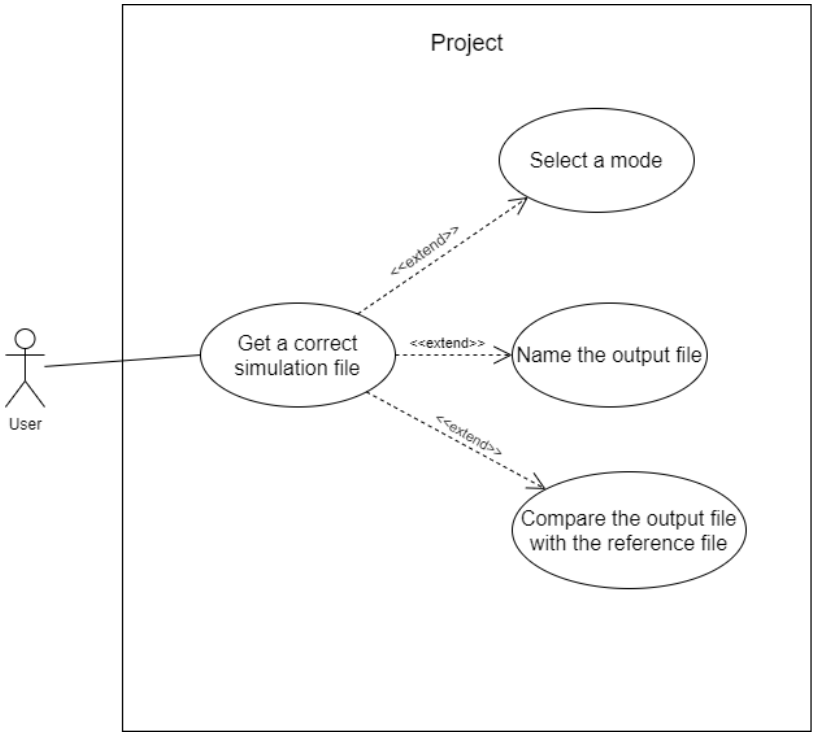

图 4-3 Use Case Diagram

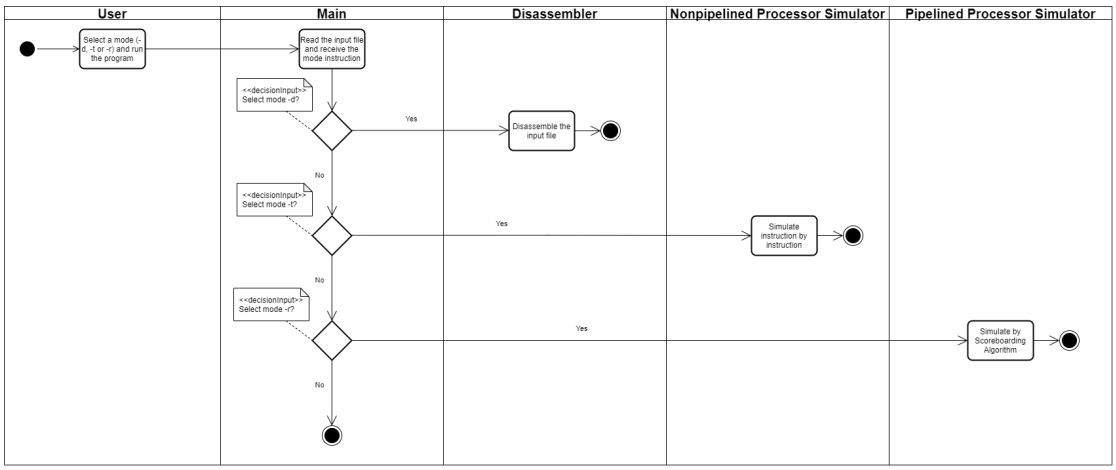

图 4-4 Activity Diagram

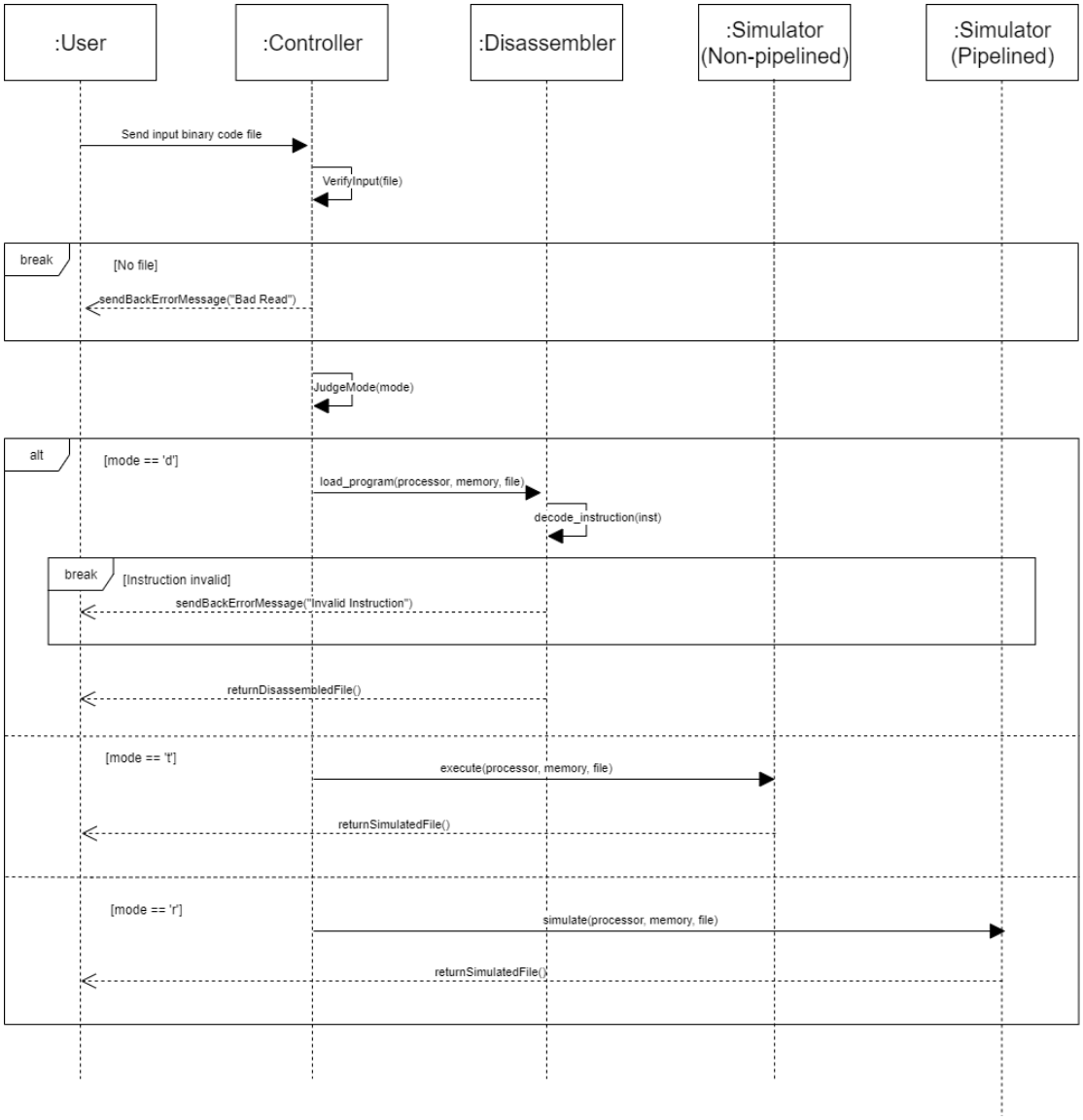

图 4-5 Sequence Diagram