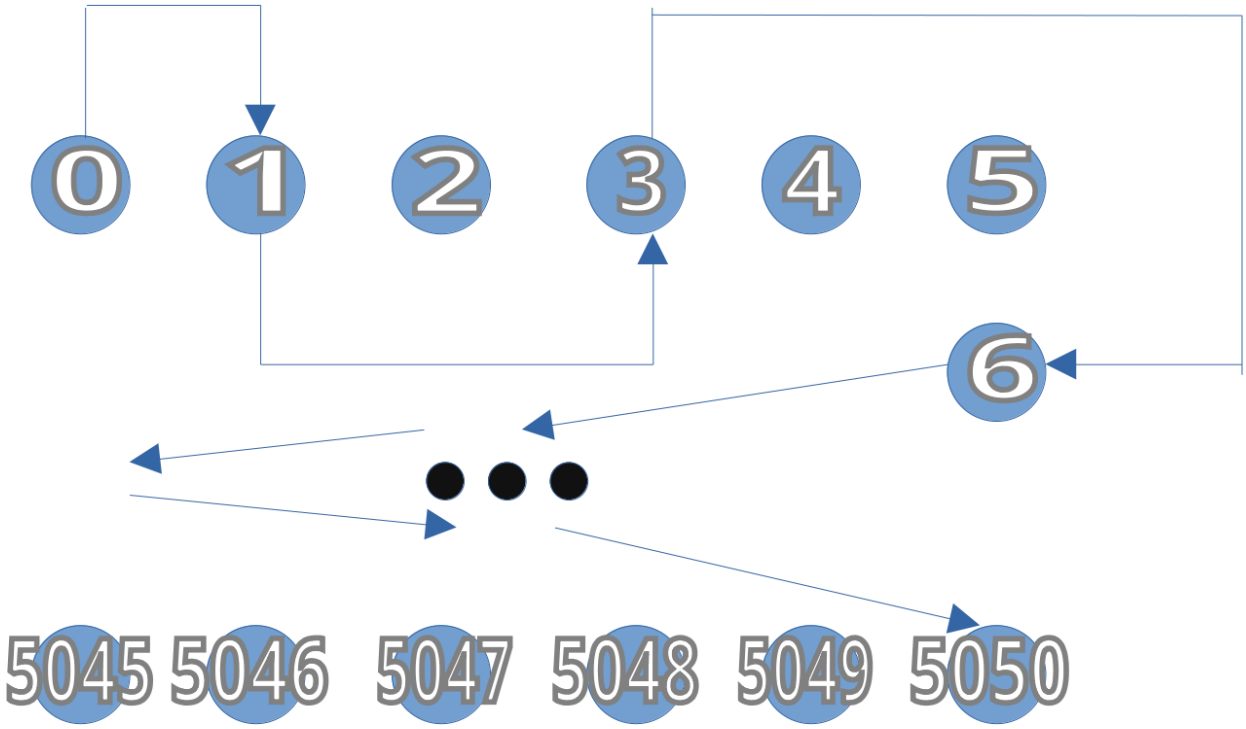
PA1实验报告

231300027朱士杭

必答题

程序是个状态机



1+2+...+100的程序状态机图如图所示，由于ubuntu下载的LibreOffice Draw过于垃圾，使用不太方便，所以画出来的图比较丑陋，但是能看明白就好。

这里的状态机其实主要是以程序的结果寄存器为例，省略了中间很多的状态以实现抽象（比如从内存中读取数据到寄存器当中，寄存器之间的数据传送），只保留了最终的结果作为状态，因此只有100个状态是有用的，其余状态都是被忽略的。

假设结果寄存器能够存储32位的值，那么在本图中其实总共应该有2^{32}个状态（但是实际上状态个数不是这个，因为实际的状态个数是需要算上总共具有的存储容量$2^{存储的总共位数个数}$）

理解基础设施

基于题目中给出的假设，可以进行一下计算：

总共编译500次NEMU,450次都是用于调试
·如果没有实现简易调试器：
排除一个bug需要花费30*20=600秒=10分钟才能排除一个bug
·如果实现简易调试器：
排除一个bug需要10分钟时间but可以获取并分析出更多的信息，隐式益处更大

RTFM 查阅问题所在位置

在我的PA中选择了riscv32的ISA指令体系结构，阅读RISCV32的ISA手册 "riscv-spec-20191213.pdf" 文档

riscv32有哪几种指令格式? Page15 -> Page29 & Page4

首先先查看目录，一般来说ISA手册在开头总得先大致介绍一下大体上有哪些指令格式吧，因此抛开引言啥的发现在Chapter2当中有介绍基本指令格式"Base Instruction Format"，因此猜想应该在这个目录下面会有基本的指令格式介绍，通过pdf的跳转功能，成功跳到该目录下，并且进行相应的查看，发现确实一共有4中最最基本的指令格式R/I/S/U型指令，当然往下确实还可以继续去细分但是没有什么必要了，之后还可以继续往下看，现在按需寻找即可

instructions using a 3-address format. Although a 2-address format would be possible, it would increase instruction count and lower efficiency. We wanted to avoid intermediate instruction sizes (such as Xtensa's 24-bit instructions) to simplify base hardware implementations, and once a 32-bit instruction size was adopted, it was straightforward to support 32 integer registers. A larger number of integer registers also helps performance on high-performance code, where there can be extensive use of loop unrolling, software pipelining, and cache tiling.

For these reasons, we chose a conventional size of 32 integer registers for the base ISA. Dynamic register usage tends to be dominated by a few frequently accessed registers, and regfile implementations can be optimized to reduce access energy for the frequently accessed registers [20]. The optional compressed 16-bit instruction format mostly only accesses 8 registers and hence can provide a dense instruction encoding, while additional instruction-set extensions could support a much larger register space (either flat or hierarchical) if desired.

For resource-constrained embedded applications, we have defined the RV32E subset, which only has 16 registers (Chapter 4).

2.2 Base Instruction Formats

In the base RV32I ISA, there are four core instruction formats (R/I/S/U), as shown in Figure 2.2. All are a fixed 32 bits in length and must be aligned on a four-byte boundary in memory. An instruction-address-misaligned exception is generated on a taken branch or unconditional jump if the target address is not four-byte aligned. This exception is reported on the branch or jump instruction, not on the target instruction. No instruction-address-misaligned exception is generated for a conditional branch that is not taken.

The alignment constraint for base ISA instructions is relaxed to a two-byte boundary when instruction extensions with 16-bit lengths or other odd multiples of 16-bit lengths are added (i.e., IALIGN=16).

Instruction-address-misaligned exceptions are reported on the branch or jump that would cause instruction misalignment to help debugging, and to simplify hardware design for systems with IALIGN=32, where these are the only places where misalignment can occur.

The behavior upon decoding a reserved instruction is UNSPECIFIED.

Some platforms may require that opcodes reserved for standard use raise an illegal-instruction exception. Other platforms may permit reserved opcode space be used for non-conforming extensions.

The RISC-V ISA keeps the source (*rs1* and *rs2*) and destination (*rd*) registers at the same position in all formats to simplify decoding. Except for the 5-bit immediates used in CSR instructions (Chapter 9), immediates are always sign-extended, and are generally packed towards the leftmost available bits in the instruction and have been allocated to reduce hardware complexity. In particular, the sign bit for all immediates is always in bit 31 of the instruction to speed sign-extension circuitry.

其实刚刚在找的过程中还漏了一个就是在Introduction引言部分会有一些基本的介绍，翻阅Chapter2之后我又重新回到Introduction部分发现确实还有一个地方也提到了，就是在Overview这个地方，通常总览部分也会介

绍一些核心的内容，比如基本的指令格式，所以不要轻易漏掉。

1.3 RISC-V ISA Overview

A RISC-V ISA is defined as a base integer ISA, which must be present in any implementation, plus optional extensions to the base ISA. The base integer ISAs are very similar to that of the early RISC processors except with no branch delay slots and with support for optional variable-length instruction encodings. A base is carefully restricted to a minimal set of instructions sufficient to provide a reasonable target for compilers, assemblers, linkers, and operating systems (with additional privileged operations), and so provides a convenient ISA and software toolchain "skeleton" around which more customized processor ISAs can be built.

Although it is convenient to speak of *the* RISC-V ISA, RISC-V is actually a family of related ISAs, of which there are currently four base ISAs. Each base integer instruction set is characterized by the width of the integer registers and the corresponding size of the address space and by the number of integer registers. There are two primary base integer variants, RV32I and RV64I, described in Chapters 2 and 5, which provide 32-bit or 64-bit address spaces respectively. We use the term XLEN to refer to the width of an integer register in bits (either 32 or 64). Chapter 4 describes the RV32E subset variant of the RV32I base instruction set, which has been added to support small microcontrollers, and which has half the number of integer registers. Chapter 6 sketches a future RV128I variant of the base integer instruction set supporting a flat 128-bit address space (XLEN=128). The base integer instruction sets use a two's-complement representation for signed integer values.

LUI指令的行为是什么? Page19

SRLI is a logical right shift (zeros are shifted into the upper bits); and SRAI is an arithmetic right shift (the original sign bit is copied into the vacated upper bits).

31 — 12	11 — 7	6 — 0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	LUI
U-immediate[31:12]	dest	AUIPC

LUI (load upper immediate) is used to build 32-bit constants and uses the U-type format. LUI places the U-immediate value in the top 20 bits of the destination register *rd*, filling in the lowest 12 bits with zeros.

AUIPC (add upper immediate to pc) is used to build pc-relative addresses and uses the U-type format. AUIPC forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the address of the AUIPC instruction, then places the result in register *rd*.

LUI （load upper immediate）加载高位立即数，用于构建 32 位常量，并使用 U 类型格式(即无符号整数 Unsigned)。LUI 将 U 立即数放在目标寄存器 rd 的前 20 位中，用零填充最低的 12 位

LUI指令通常与其他指令（如ADDI）结合使用，以便在指令中生成更大的立即数。例如，如果我们想要生成一个32位的立即数0x12345678，可以先用LUI指令加载高20位，然后用ADDI指令加载低12位：

```
LUI x3, 0x12345
ADDI x3, x3, 0x678
```

从而实现向x3寄存器当中加载/存入0x12345678这个立即数

mstatus寄存器的结构是怎么样的? Page20 in "riscv-privileged-20211203.pdf"

3.1.6 Machine Status Registers (mstatus and mstatush)

The mstatus register is an MXLEN-bit read/write register formatted as shown in Figure 3.6 for RV32 and Figure 3.7 for RV64. The mstatus register keeps track of and controls the hart's current operating state. A restricted view of mstatus appears as the sstatus register in the S-level ISA.

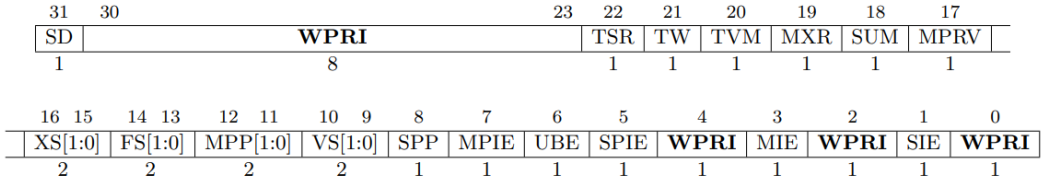


Figure 3.6: Machine-mode status register (mstatus) for RV32.

mstatus 寄存器是 RISC-V 架构中机器模式（M-mode）的一个重要控制状态寄存器（CSR），它用于管理和控制处理器的状态。这个寄存器包含了多个位字段，每个位字段都有特定的含义和功能。mstatus 寄存器跟踪并控制 HART 的当前运行状态。

status 寄存器还包含其他位字段，用于管理处理器的特权级别、虚拟化、调试模式等。通过读取和写入 mstatus 寄存器，可以控制和监控处理器的状态，例如启用或禁用中断、设置特权级别、处理异常等。这对于操作系统、异常处理程序和其他特权级别的软件非常重要。

总的来说，mstatus 寄存器是 RISC-V 架构中用于控制处理器状态的关键寄存器（拥有特权的寄存器），在异常处理、中断管理和特权级别管理中起着至关重要的作用。

shell命令

nemu/目录下的所有.c和.h和文件总共有多少行代码?

296370 total

总计共有296360行代码，使用的shell命令如下（注意以下shell命令是在ICS2024PA目录下执行的，三种都是可以的选其中一个就行，具体的区别待会会说）：

```
find nemu/ -type f \( -name "*.c" -o -name "*.h" \) | xargs wc -l

find nemu/ -type f \( -name "*.c" -o -name "*.h" \) -exec wc -l {} +


find nemu/ -type f \( -name "*.c" -o -name "*.h" -o -name ".*.c" -o -name
".*.h" \) -exec wc -l {} +
```

对于第一行的命令来说（这个命令会输出所有.c和.h文件的总行数）：

1. find nemu/：在nemu/目录下查找文件。
2. -type f：只查找文件（不包括目录）。
3. (-name ".*c" -o -name* ".h")：查找所有以.c或.h结尾的文件。-o是逻辑“或”操作，(和)用于分组。
4. |：管道符号，将前一个命令的输出作为下一个命令的输入。
5. xargs wc -l：xargs命令将接收到的文件列表作为参数传递给wc -l，wc -l用于统计行数。

第二行命令其实和第一行命令没有什么区别，不再赘述了

第三行命令是考虑到了可能会有隐藏文件的情况，要统计包括隐藏文件（以.开头的文件）在内的所有.c和.h文件，需要添加.*的通配符*"..c"和".*.h"

和框架代码相比, 你在PA1中编写了多少行代码?

由于目前pa0分支中记录的正好是做PA1之前的状态, 通过比较PA1和PA0之间的不同就可以显示出自己在PA1当中编写了多少代码/做了多少的改动

使用git diff命令来统计两个分支之间的差异。可以使用--patch（或-p）选项来查看具体的代码差异，或者使用--stat选项来获取一个统计摘要都可以。在这里我们选择使用stat参数来进行统计：

```
edzee3000@edzee3000-ASUS-TUF-Gaming-F16-FX607JV-FX607JV:~/ICS2024/ICS2024PA/nemu
$ git diff --stat pa0..pa1
 .gitignore                            |   1 +
 Makefile                              |   2 +-
 am-kernels                            |   1 -
 nemu/Kconfig                          |  17 +++
 nemu/src/cpu/cpu-exec.c               |  23 ++-
 nemu/src/engine/interpreter/init.c    |   2 +-
 nemu/src/isa/riscv32/reg.c            |  29 ++++
 nemu/src/monitor/monitor.c            |   4 +-
 nemu/src/monitor/sdb/expr.c           | 295 +++++++++++++++++++++++++++++++++----
 nemu/src/monitor/sdb/sdb.c            | 147 ++++++++++++++++--
 nemu/src/monitor/sdb/watchpoint.c     | 126 +++++++++++++++-
 nemu/src/nemu-main.c                  |   4 +-
 nemu/tools/gen-expr/gen-expr.c        |  93 +++++++++---
 13 files changed, 669 insertions(+), 75 deletions(-)
edzee3000@edzee3000-ASUS-TUF-Gaming-F16-FX607JV-FX607JV:~/ICS2024/ICS2024PA/nemu
```

先使用--stat参数来进行摘要统计可以发现在不同的文件下面进行了多少改动，+表示增加内容，-表示减少内容，但是总感觉不够我们的需求，因此我们在stat基础上使用numstat表示行数统计：

```
edzee3000@edzee3000-ASUS-TUF-Gaming-F16-FX607JV-FX607JV:~/ICS2024/ICS2024PA/nemu
$ git diff --numstat pa0..pa1
1       0       .gitignore
1       1       Makefile
0       1       am-kernels
17      0       nemu/Kconfig
18      5       nemu/src/cpu/cpu-exec.c
1       1       nemu/src/engine/interpreter/init.c
29      0       nemu/src/isa/riscv32/reg.c
2       2       nemu/src/monitor/monitor.c
268     27      nemu/src/monitor/sdb/expr.c
134     13      nemu/src/monitor/sdb/sdb.c
121     5       nemu/src/monitor/sdb/watchpoint.c
2       2       nemu/src/nemu-main.c
75      18      nemu/tools/gen-expr/gen-expr.c
edzee3000@edzee3000-ASUS-TUF-Gaming-F16-FX607JV-FX607JV:~/ICS2024/ICS2024PA/nemu
$ git diff --numstat pa0..pa1 | awk '{total += $1 + $2} END {print "Total lines
added/removed: " total}'
Total lines added/removed: 744
```

经过行数统计之后可以显示在不同分支中所修改的总代码行数

倘若将比较分支代码修改行数添加到Makefile文件当中去，则可以通过make count命令进行输出代码修改行数的信息，如图所示，在自己的Makefile当中进行如下的修改（虽然之前的手册有说尽量不要去修改Makefile文件，但是管他呢，只是增添了一些东西而已，不会有什么大问题的）

```
    -@while (test -e .git/index.lock); do sleep 0.1; done
    -@(echo "> $(1)" && echo $(STUID) $(STUNAME) && uname -a && uptime) | git commit -F - $(GITFLAGS)
    -@sync
endef

_default:
    @echo "Please run 'make' under subprojects."

submit:
    git gc
    STUID=$(STUID) STUNAME=$(STUNAME) bash -c "$$(curl -s http://why.ink:8080/static/submit.sh)"

count:
    @echo "PA1和PA0相比修改的代码行数为:"
    @git diff --numstat pa0..pa1
    @git diff --numstat pa0..pa1 | awk '{total += $$1 + $$2} END {print "Total lines added/removed: " total}'

.PHONY: default submit
```

```
edzee3000@edzee3000-ASUS-TUF-Gaming-F16-FX607JV-FX607JV:~/ICS2024/ICS2024PA$ mak
e count
PA1和PA0相比修改的代码行数为:
1       0       .gitignore
1       1       Makefile
0       1       am-kernels
17      0       nemu/Kconfig
18      5       nemu/src/cpu/cpu-exec.c
1       1       nemu/src/engine/interpreter/init.c
29      0       nemu/src/isa/riscv32/reg.c
2       2       nemu/src/monitor/monitor.c
268     27      nemu/src/monitor/sdb/expr.c
134     13      nemu/src/monitor/sdb/sdb.c
121     5       nemu/src/monitor/sdb/watchpoint.c
2       2       nemu/src/nemu-main.c
75      18      nemu/tools/gen-expr/gen-expr.c
Total lines added/removed: 744
```

除去空行之外, nemu/目录下的所有.c和.h文件总共有多少行代码?

一共有259704非空行代码，使用的shell命令如下

```
find nemu/ -type f \( -name "*.c" -o -name "*.h" \) -exec grep -vE '^ *$'
{} + | wc -l
```

相比于之前统计代码行数的shell命令，这里的shell命令只是多了grep -vE '^ *$'的命令

主要是使用 grep 命令排除空行。其中的 `-v` 选项可以使 grep 显示不匹配的行，`-E` 允许使用扩展正则表达式，`"^ *$"`匹配完全为空或只包含空白字符的行。然后将grep之后的命令通过管道符传给word count统计line行数最后打印出结果就为259704

RTFM nemu/scripters/build.mk文件中CFLAGS变量的gcc编译选项 -Wall和-Werror

请解释gcc中的-Wall和-Werror有什么作用

在 gcc 编译器中，`-Wall` 和 `-Werror` 是两个常用的编译选项，它们用于控制编译过程中的警告信息。

对于`-Wall`选项，Wall 是 “Wall of warnings” 的缩写，意为“警告墙”。这个选项告诉 gcc 打开几乎所有的警告信息。这些警告信息包括了编译器能检测到的潜在问题，比如未初始化的变量、可能的数组越界、未使用的变量等。 使用 -Wall 可以帮助开发者发现代码中可能存在的问题，从而提高代码质量。它不会影响编译过程的成功与否，即使出现了警告，编译仍然会继续进行。

对于`-Werror`选项， Werror将所有警告转化为错误。这意味着，如果编译器发出了任何警告，编译过程会失败，并且不会生成可执行文件。（比如未初始化的变量、可能的数组越界、未使用的变量等这些编译过程其实是不太影响代码运行的但是仍然会显示错误，属于更加严格的） 这个选项通常用于确保代码的高标准，特别是在开发过程中。通过将警告视为错误，开发者被迫解决所有潜在的问题，而不是忽视它们。 在团队开发或代码审查中，`-Werror` 可以帮助保持代码的一致性和可靠性。

为什么要使用-Wall和-Werror?

一方面当然可以提高代码质量，帮助开发者注意到可能的问题，并在代码审查和测试之前解决它们，避免潜在的bug，这些警告指示了可能导致运行时错误或未定义行为的代码模式；另一方面是便于我们之后维护项目，代码中警告较少或没有警告，可以减少维护时的难度，因为潜在的问题已经被解决。

在 nemu 项目或以后其他需要高可靠性的软件项目中，使用 -Wall 和 -Werror 可以帮助我们编写更健壮、更可靠的代码。这些选项是许多项目中最佳实践的一部分，尤其是在安全性和稳定性至关重要的系统中显得更加有必要了。