微处理器系统设计

> Sep 5 23:55

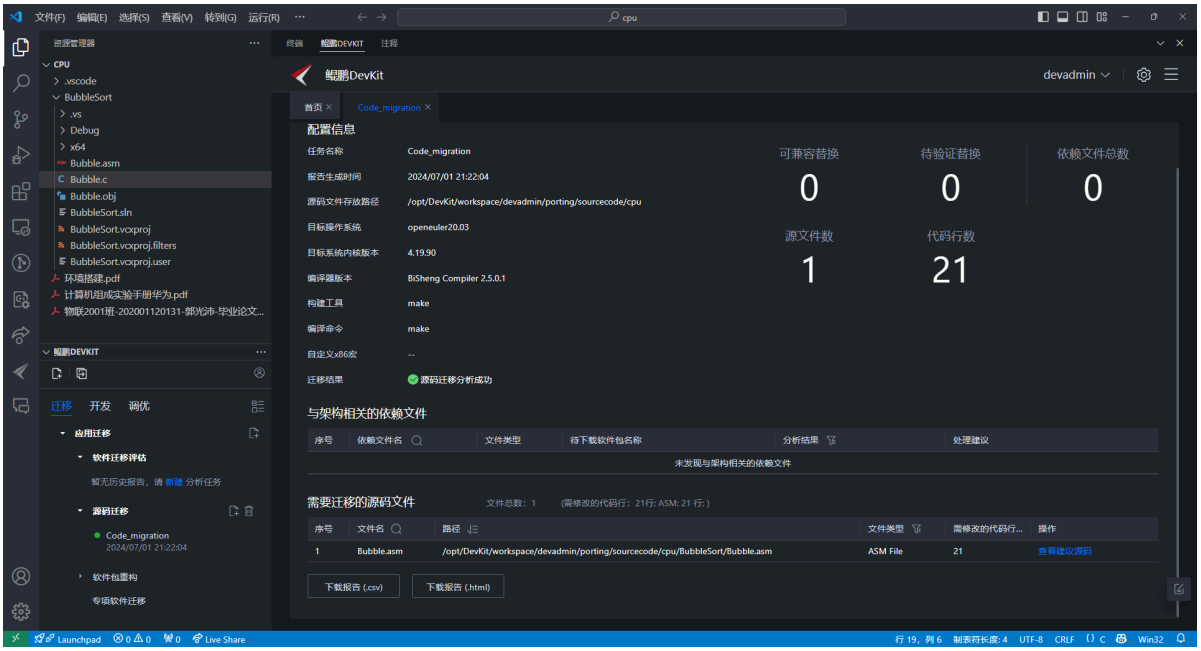
任务（可选）：

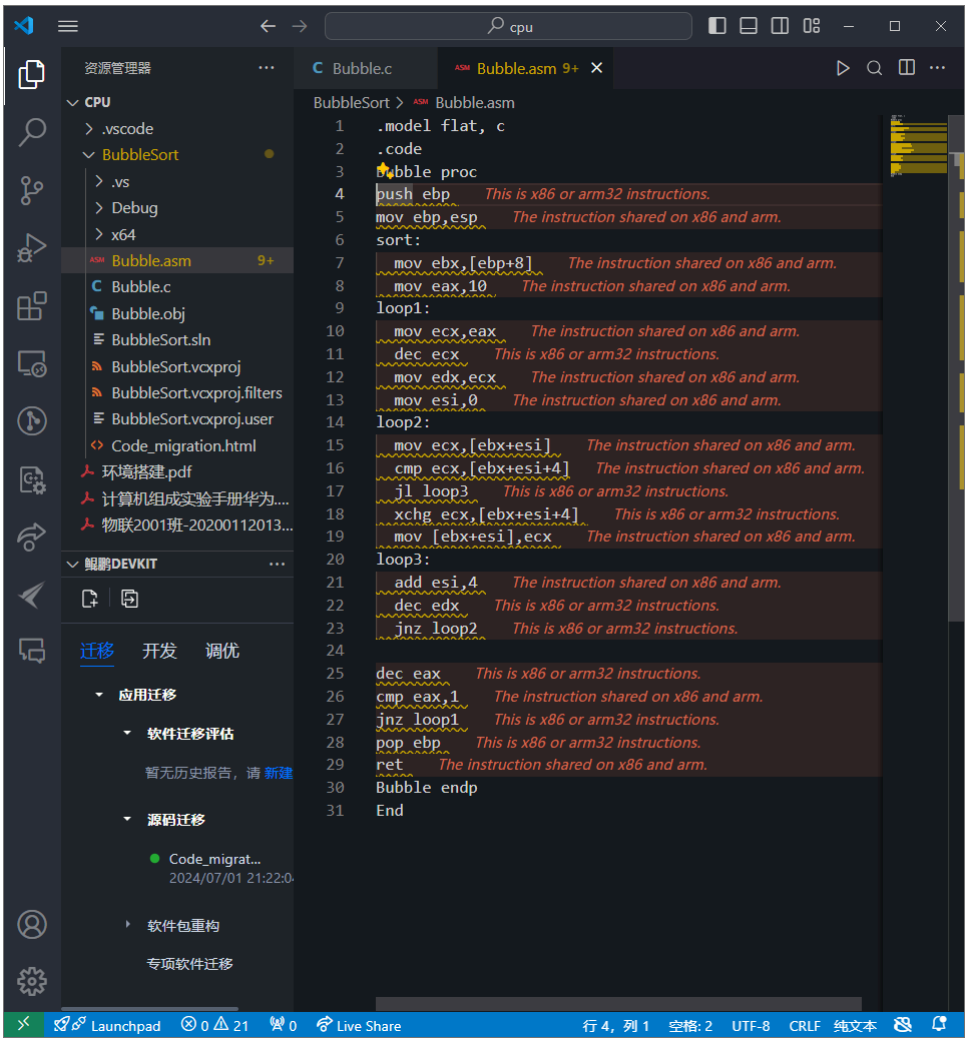
1. 华为鲲鹏 CPU 模拟环境搭建及程序移植实验（第 8 个实验任务，移植冒泡排序到鲲鹏 CPU上）；
2. 复现 * 同学乱序执行 CPU 的仿真实验；
3. 为乱序执行 CPU 增加分支预测功能并完成仿真实验（增加 cache）。

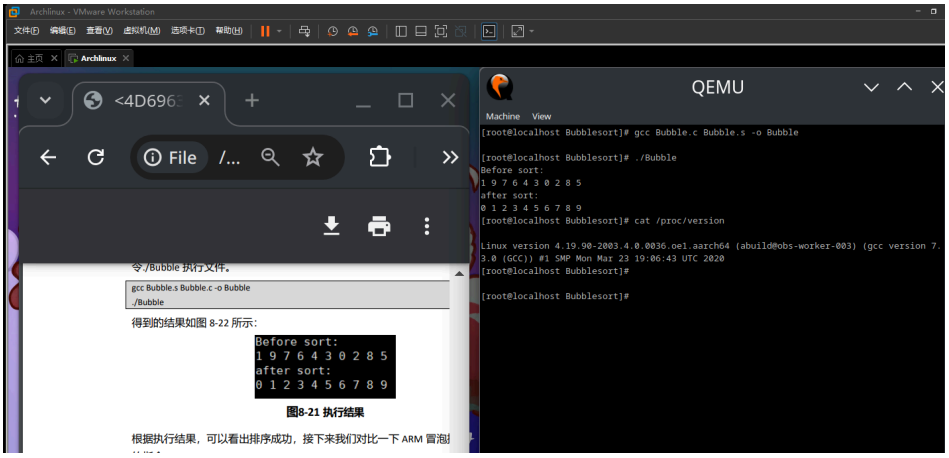
最低要求——任务1

> 2024/7/1 done

参考资料：openEuler







中等要求——任务2

> 任务理解：一生一芯 && 南大 PA0、PA1 指导环境搭建并补充基础知识，整体是一个地基的作用，而后需要自行阅读 riscv 手册设计一个 cpu，在 nemu 下仿真验证，该任务结束。

参考资料：

- 一生一芯
- b 站讲解视频
- verilator examples

预学习阶段

> 目标：学习一生一芯预学习部分以及「其他资料」里的南京大学 PA 实验先做一下，理解 nemu（NJU Emulator运行机理，把 dummy 样例跑起来即可。

NEMU

clone repo：

```
git clone -b ysyx-2204 git@github.com:OSCPU/ysyx-workbench.git ics2024
```

查看 README，通过 `bash init.sh <subproject-name>` 来初始化 nemu & abstract-machine，而后键入`make menuconfig`，在 ISA 选项中勾选 64-bit RISC-V architecture，生成配置文件。

为了顺利在 `/nemu` 下顺利编译需要预安装一系列依赖：

```
sudo pacman -Syu base-devel man-db gcc-doc gdb git readline sdl2 llvm clang
```

安装好依赖后，键入 `make` 编译，再键入 `make run` 运行，首次运行会发现直接报错退出，原因是在 `./src/monitor/monitor.c` 有 `asser(0)`，直接注释即可。

可以观察到键入 `q` 退出会时报错，怀疑 `exit()` 没有正常退出。从 error info 中看到 `xxx.mk` 有问题，进入 .mk 追踪，再 RTFSC 一下 `nemu-main.c`，自然去全局搜索 `is_exit_status_bad()`，在 `state.c` 中找到该函数，继续追溯至 `state.c` 修改 `cmd_q()`，添加如下代码，实现优雅退出。

```
  nemu_state.state = NEMU_QUIT;
```

```
bk@wukong ~/riscv-cpu/ics2024/nemu
pa0* $ make run                                                    [15:55:18]
+ CC src/monitor/sdb/sdb.c
+ LD /home/bk/riscv-cpu/ics2024/nemu/build/riscv64-nemu-interpreter
make[1]: 进入目录“/home/bk/riscv-cpu/ics2024”
make[1]: 离开目录“/home/bk/riscv-cpu/ics2024”
make[1]: 进入目录“/home/bk/riscv-cpu/ics2024”
make[1]: 离开目录“/home/bk/riscv-cpu/ics2024”
/home/bk/riscv-cpu/ics2024/nemu/build/riscv64-nemu-interpreter --log=/home/bk/riscv-cpu/ics2024/nemu/bui
ld/nemu-log.txt
[src/utils/log.c:30 init_log] Log is written to /home/bk/riscv-cpu/ics2024/nemu/build/nemu-log.txt
[src/memory/paddr.c:50 init_mem] physical memory area [0x80000000, 0x87ffffff]
[src/monitor/monitor.c:51 load_img] No image is given. Use the default build-in image.
[src/monitor/monitor.c:28 welcome] Trace: ON
[src/monitor/monitor.c:29 welcome] If trace is enabled, a log file will be generated to record the trace
. This may lead to a large log file. If it is not necessary, you can disable it in menuconfig
[src/monitor/monitor.c:32 welcome] Build time: 15:51:27, Aug 29 2024
Welcome to riscv64-NEMU!
For help, type "help"
(nemu) q

bk@wukong ~/riscv-cpu/ics2024/nemu
pa0* $                                                             [15:55:26]
```

Verilator

通过 `bash init.sh npc` 初始化 npc 文件夹，再通过 `yay -S verilator` 安装 verilator。

verilator 有两种使用方式：一种是查手册根据目标代码手动编译，直接使用 `--build` 参数；另一种是根据 `/verilator/examples` 的 `CMakeLists.txt && Makefile`（e.g. `make -j -C obj_dir/ -f Vexample_DoubleControlSwitch.mk Vexample_DoubleControlSwitch`）进行编译。

verilator 使用指南：https://soc.ustc.edu.cn/CECS/lab2/verilator/

接下来编写双控开关代码：

`/npc/vsrc/example_DoupleControlSwitch.v`:

```
module example_DoubleControlSwitch(
    input a,
    input b,
    output f
);
    assign f = a ^ b;
endmodule
```

注意 .v 文件最后需要添加一行空行，作为 EOF，否则会有 warning（尽管忽略也能正常仿真但还是不舒服），warning 解释如下：

EOFNEWLINE

Warns that a file does not end in a newline. POSIX defines that a line must end in a newline, as otherwise, for example **cat** with the file as an argument may produce undesirable results.

Repair by appending a newline to the end of the file.

Disabled by default as this is a code-style warning; it will simulate correctly.

Other tools with similar warnings: Verible's posix-eof, "File must end with a newline."

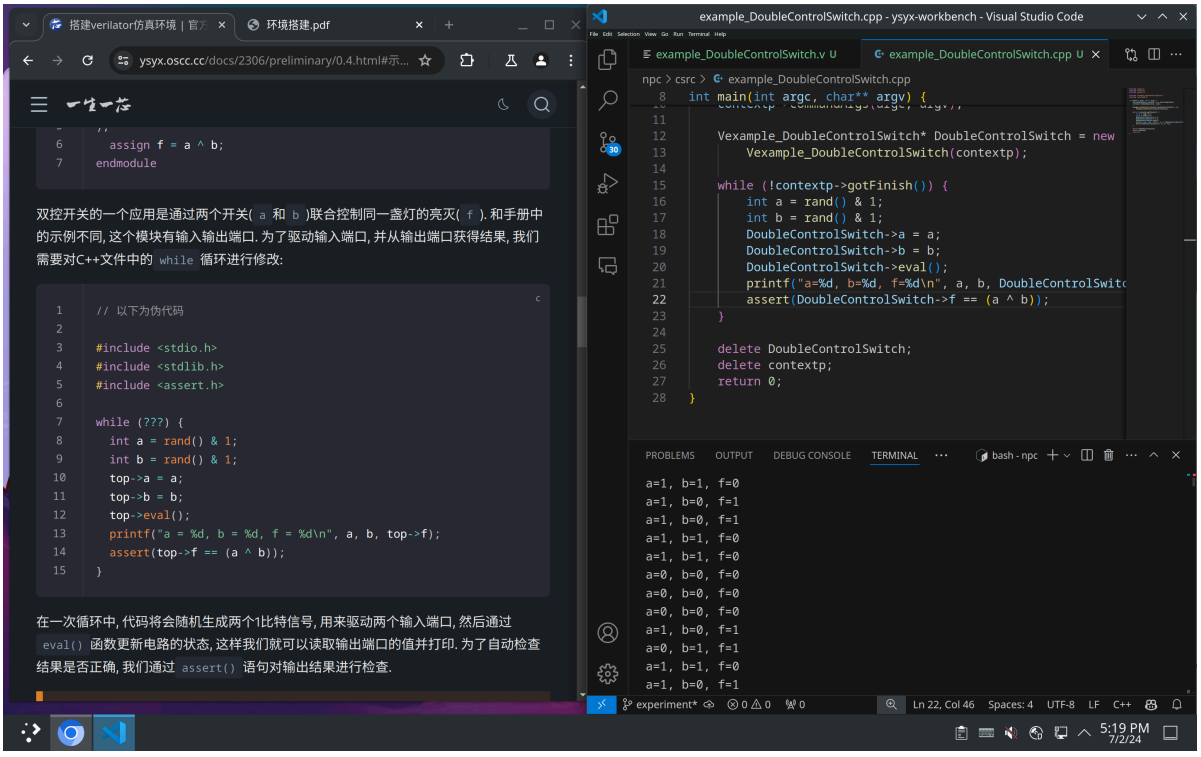
`/npc/csrc/example_DoupleControlSwitch.cpp`:

```
// 整体编码逻辑：添加 C library → 添加 verilator 转换的 C++ 类库 && verilated.h 主库 → 初始化模拟器上下文 → 声明 .v 中的模块 → 业务逻辑 → delete

#include <cstdlib>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include "Vexample_DoubleControlSwitch.h"
#include "verilated.h"

int main (int argc, char **argv) {
    VerilatedContext *contextp = new VerilatedContext;
    contextp->commandArgs(argc, argv);
    Vexample_DoubleControlSwitch* DoubleControlSwitch = new
        Vexample_DoubleControlSwitch(contextp);

    while (!contextp->gotFinish()) {
        int a = rand() & 1;
        int b = rand() & 1;
        DoubleControlSwitch->a = a;
        DoubleControlSwitch->b = b;
        DoubleControlSwitch->eval();
        printf("a=%d, b=%d, f=%d\n", a, b, DoubleControlSwitch->f);
        assert(DoubleControlSwitch->f == (a ^ b));
    }

    delete DoubleControlSwitch;
    delete contextp;
    return 0;
}
```

编写 `/npc/Makefile`：

```
VERILOG_MODULE=example_DoubleControlSwitch.v
CXX_MODULE=example_DoubleControlSwitch.cpp
VERILATOR_FLAGS=--cc --exe --build -j $(shell nproc) -Wall

all:
    @echo "Write this Makefile by your self."

sim:
    $(call git_commit, "sim RTL") # DO NOT REMOVE THIS LINE!!!
    verilator $(VERILATOR_FLAGS)  \
        ./csrc/$(CXX_MODULE) ./vsrc/$(VERILOG_MODULE) \

include ../Makefile
```

键入 `make sim` 成功仿真！



如果需要查看波形文件，需要在 `.cpp` 实现中添加 `dump()`，并给 `/npc/Makefile` VERILATOR_FLAGS 添加 `--trace`。

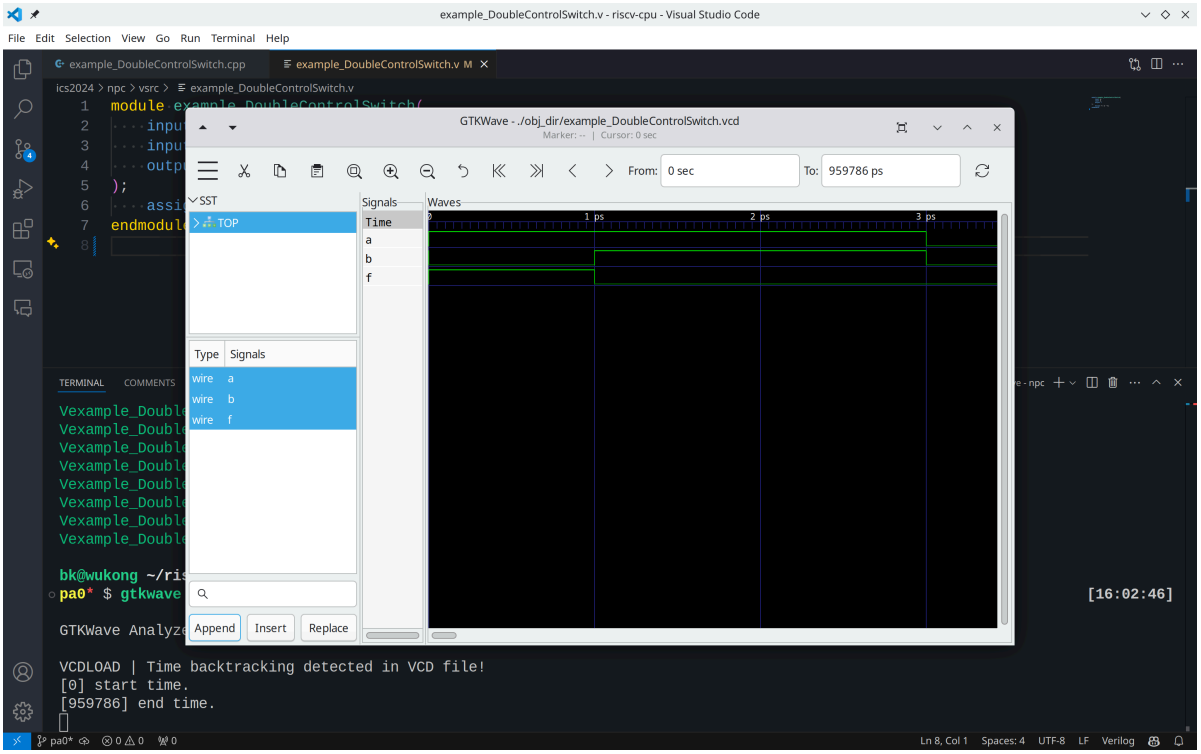
生成波形版本的 `/npc/csrc/example_DoupleControlSwitch.cpp`：

```
#include <cstdlib>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include "Vexample_DoubleControlSwitch.h"
#include "verilated.h"
#include "verilated_vcd_c.h" // 添加 vcd 库

int main (int argc, char **argv) {
    VerilatedContext *contextp = new VerilatedContext;
```

```
    contextp->commandArgs(argc, argv);
    Vexample_DoubleControlSwitch* DoubleControlSwitch = new
        Vexample_DoubleControlSwitch(contextp);

    // 初始化 vcd 对象，设置波形文件保存位置
    VerilatedVcdC* tfp = new VerilatedVcdC;
    contextp->traceEverOn(true);
    DoubleControlSwitch->trace(tfp, 0);
    tfp->open("./obj_dir/example_DoubleControlSwitch.vcd");

    while (!contextp->gotFinish()) {
        int a = rand() & 1;
        int b = rand() & 1;
        DoubleControlSwitch->a = a;
        DoubleControlSwitch->b = b;
        DoubleControlSwitch->eval();
        printf("a=%d, b=%d, f=%d\n", a, b, DoubleControlSwitch->f);

        // dump 波形文件
        tfp->dump(contextp->time());
        contextp->timeInc(1);

        assert(DoubleControlSwitch->f == (a ^ b));
    }

    delete DoubleControlSwitch;
    tfp->close();
    delete contextp;
    return 0;
}
```

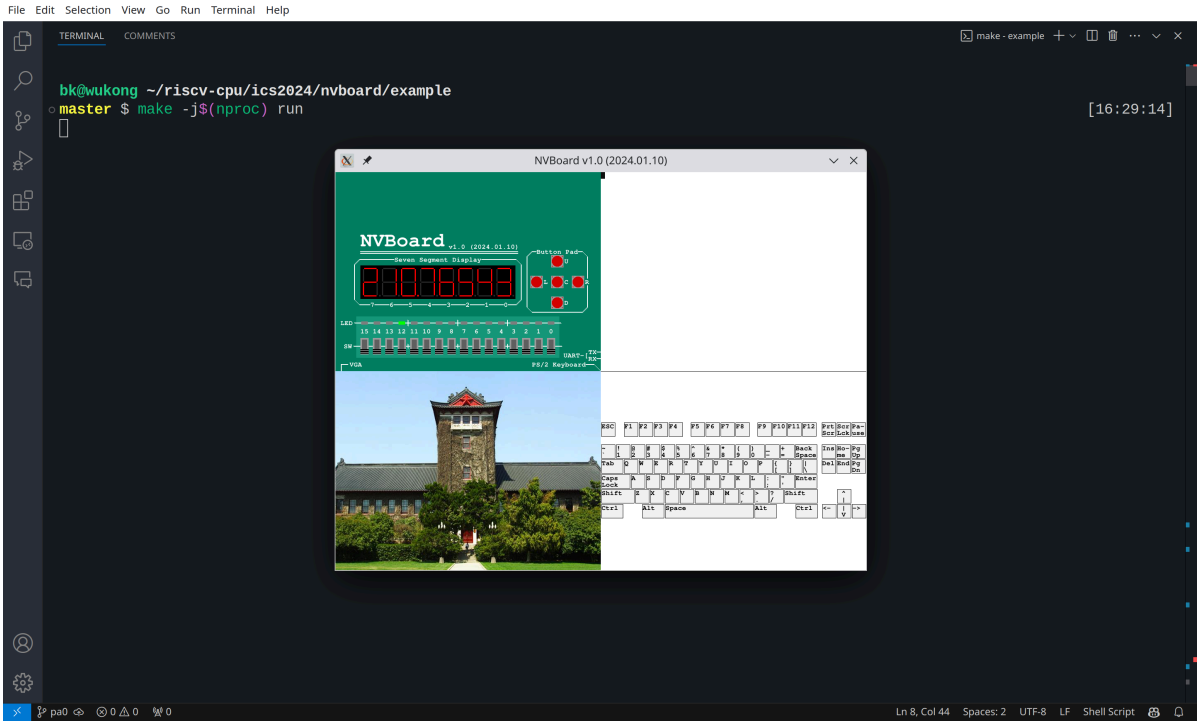
注意：

1. 若采用 pull src 手动编译 verilator 的方式，记得添加 ldd 的查找路径（若直接 yay 安装则可 pass）；
2. verilator 指令中，参数`--trace-fst` 和 `--trace` 分别对应 fst 函数和 vcd 函数；
3. 重新键入 `make sim`，运行 `./obj_dir/Vexample_DoubleControlSwitch` 后，即可在 `./obj_dir` 找到 `example_DoubleControlSwitch.vcd` 波形文件。



NVBoard

通过 `bash init.sh nvboard` 初始化 nvboard 文件夹，阅读 README 安装依赖，进入 `/nvboard/example` 目录，键入 `make -j$(nproc) run` 初体验什么是 NVBoard，记得先安装依赖 `sudo pacman -S sdl2_ttf`。



NVBoard(NJU Virtual Board)是南京大学开发的，用于教学的虚拟 FPGA 板卡项目, 可以在 RTL 仿真环境中提供一个虚拟板卡的界面。其代码形式如下：

```
#include <nvboard.h>

// ...
nvboard_bind_all_pins(&dut);
nvboard_init();

while (1) {
  // ...
  nvboard_update();
}

nvboard_quit();
```

如何引入 NVBoard 呢？在 Makefile 中：

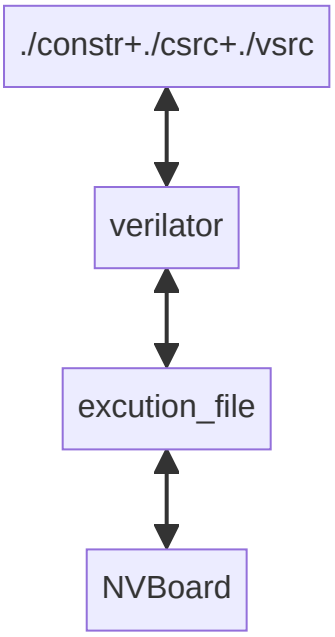
- 将生成的上述引脚绑定的C++文件加入源文件列表
- 将NVBoard的构建脚本包含进来 `include $(NVBOARD_HOME)/scripts/nvboard.mk`
- 通过 `make nvboard-archive` 生成 NVBoard 的库文件
- 在生成 verilator 仿真可执行文件(即(NVBOARD_ARCHIVE))将这个库文件加入链接过程，并添加链接选项`-lSDL2 -lSDL2_image`

双控开关 Makefile 示例：

```
TOPNAME = example_DoubleControlSwitch
NXDC_FILES = constr/example_DoubleControlSwitch.nxdc
INC_PATH ?=

VERILATOR = verilator
VERILATOR_CFLAGS += -MMD --build -cc  \
                -O3 --x-assign fast --x-initial fast --noassert

BUILD_DIR = ./build
OBJ_DIR = $(BUILD_DIR)/obj_dir
BIN = $(BUILD_DIR)/$(TOPNAME)

default: $(BIN)

$(shell mkdir -p $(BUILD_DIR))

# constraint file
SRC_AUTO_BIND = $(abspath $(BUILD_DIR)/auto_bind.cpp)
$(SRC_AUTO_BIND): $(NXDC_FILES)
    python3 $(NVBOARD_HOME)/scripts/auto_pin_bind.py $^ $@

# project source
VSRCS = $(shell find $(abspath ./vsrc) -name "*.v")
CSRCS = $(shell find $(abspath ./csrc) -name "*.c" -or -name "*.cc" -or -name
"*.cpp")
CSRCS += $(SRC_AUTO_BIND)

# rules for NVBoard
include $(NVBOARD_HOME)/scripts/nvboard.mk

# rules for verilator
```

```
INCFLAGS = $(addprefix -I, $(INC_PATH))
CXXFLAGS += $(INCFLAGS) -DTOP_NAME="\"V$(TOPNAME)\""

$(BIN): $(VSRCS) $(CSRCS) $(NVBOARD_ARCHIVE)
    @rm -rf $(OBJ_DIR)
    $(VERILATOR) $(VERILATOR_CFLAGS) \
        --top-module $(TOPNAME) $^ \
        --trace \
        $(addprefix -CFLAGS , $(CXXFLAGS)) $(addprefix -LDFLAGS , $(LDFLAGS)) \
        --Mdir $(OBJ_DIR) --exe -o $(abspath $(BIN))

all: default

run: $(BIN)
    @$^

clean:
    rm -rf $(BUILD_DIR)

.PHONY: default all clean run
```

截至目前的项目理解（实现与应用是两个方向）：



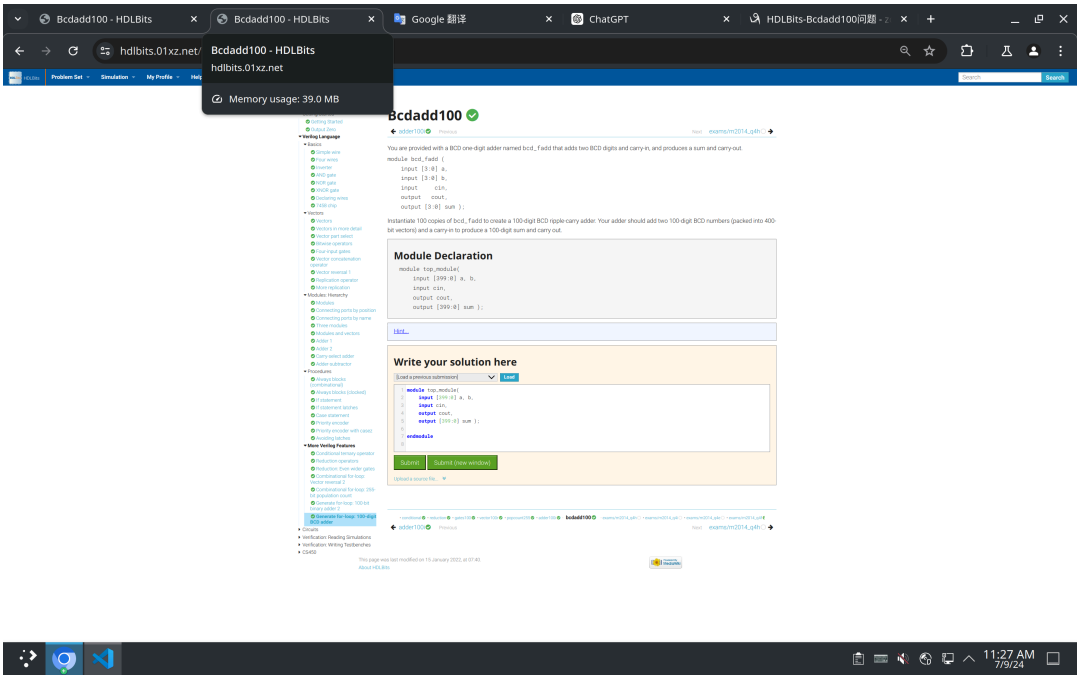
基础阶段

> 根据实验报告和 paper 继续复现（paper 是更宏观的概述，在中等要求的实验初期可以只看实验报告，paper 留作扩展阅读），方向是搭建一系列环境，而后根据 riscv 等相关资料（可以在 PA 导论找到）设计模块，通过一系列 `.v` 实现 riscv cpu 的设计之后，用 nemu 进行验证后该任务结束。

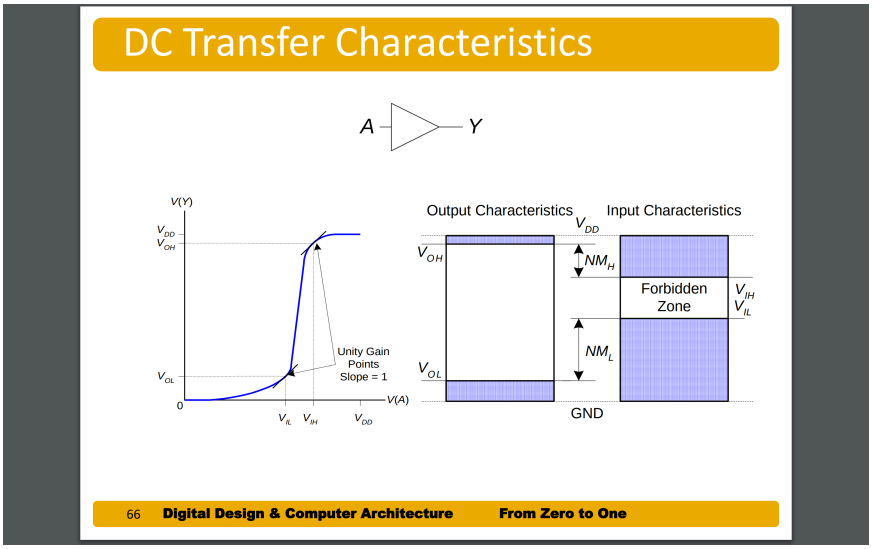
数电基础知识（DDCArv Chapter1-5）：

- 参考资料：
  - DDCArv
  - 实操网站

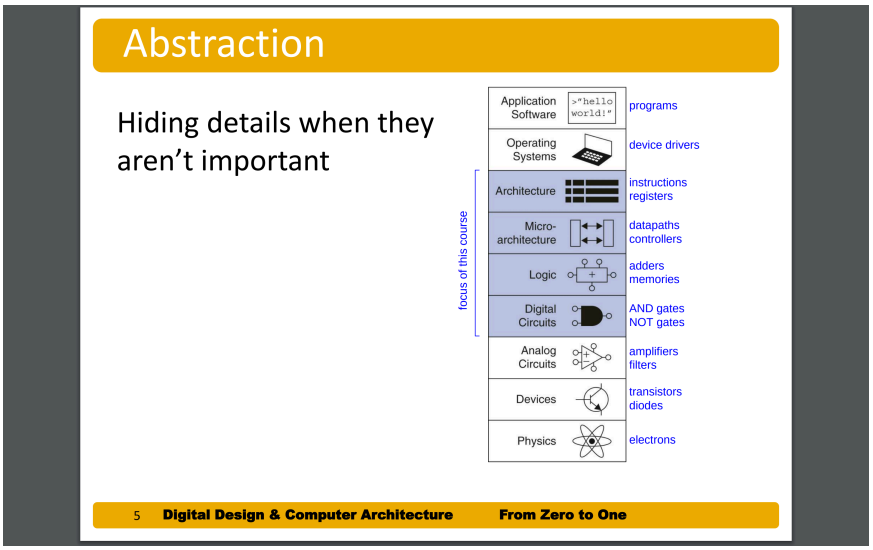
> In a combinational always block, use **blocking** assignments. In a clocked always block, use **non-blocking** assignments.
>
> In procedural assignment, use **reg** type
>
> `assign` 关键字主要用于定义连续赋值的组合逻辑输出和 `wire` 类型的信号赋值
>
> 锁存器 / 触发器存在于逻辑电路中，目的是为了记录电路状态，但容易进入未定义的状态，为避免出现这种问题，应该使语句尽可能完整，杜绝从 module 视角来看输出端口信号不发生任何变化的现象
>
> verilog 代码经过仿真器仿真，综合器转换为硬件实现，最后在硬件电路上运行



- 无法做到真正的连续，只能提供近似的离散值，误差是一定存在的（e.g. time latency）



- 脑海中**先有电路再有代码**，verilog 本质上是**硬件描述语言**，真正的描述电路 = 实例化 + 连线
  - e.g., Unlike a programming language, assign statements ("continuous assignments") **describe connections between things**, not the action of copying a value from one thing to another.



PA1

仅完成到 PA1 phase1

> 三个对调试有用的宏（`nemu/include/debug.h`）中定义
>
> - `Log()`是`printf()`的升级版, 专门用来输出调试信息, 同时还会输出使用`Log()`所在的源文件, 行号和函数. 当输出的调试信息过多的时候, 可以很方便地定位到代码中的相关位置
> - `Assert()`是`assert()`的升级版, 当测试条件为假时, 在assertion fail之前可以输出一些信息
> - `panic()`用于输出信息并结束程序, 相当于无条件的assertion fail

`/ics2024/nemu/src/monitor/sdb/sdb.c`:

```
***
#include "../include/memory/vaddr.h"
***
static int cmd_q(char *args) {
  nemu_state.state = NEMU_QUIT;
  return -1;
}

static int cmd_si(char *args) {
  int n = 1;
  if (args != NULL) {
    n = atoi(args);
  }
  Assert(n > 0, "Invalid si argument: %s\n", args);
  cpu_exec(n);
  return 0;
}

static int cmd_info(char *args) {
  if (args == NULL)  {
    printf("Usage: info r/w. Missing argument\n");
    return 0;
  }
  switch (*args) {
    case 'r':
      isa_reg_display();
      break;
```

```
    case 'w':
     // sdb_watchpoint_display();
      break;
    default:
      printf("Usage: info r/w. Invalid argument\n");
  }

  return 0;
}

static int cmd_x(char *args) {
  char *arg1 = strtok(NULL, " ");
  char *arg2 = strtok(NULL, " ");
  if (arg1 == NULL || arg2 == NULL) {
    printf("Usage: x N EXPR. Missing argument\n");
    return 0;
  }

  char *arg3 = strtok(NULL, " ");
  if (arg3 != NULL) {
    printf("Usage: x N EXPR. Too many arguments\n");
    return 0;
  }

  int n = atoi(arg1);
  Assert(n > 0, "Invalid x argument: %s\n", args);
  vaddr_t expr = strtol(arg2, NULL, 16);

  int i, j;
  for (i = 0; i < n;) {
    printf("%#018lx: ", expr);
    for (j = 0; i < n && j < 4; i++, j++) {
      word_t w = vaddr_read(expr, 8);
      expr += 8;
      printf("%#018lx ", w);
    }
    puts("");
  }

  return 0;
}

***

static struct {
  const char *name;
  const char *description;
  int (*handler) (char *);
} cmd_table [] = {
  { "help", "Display information about all supported commands", cmd_help },
  { "c", "Continue the execution of the program", cmd_c },
  { "q", "Exit NEMU", cmd_q },
  { "si", "Step execution", cmd_si },
  { "info", "Show program status", cmd_info },
  { "x", "Scan memory", cmd_x },
  { "p", "Expression evaluation", cmd_p },
```

```
  { "w", "Set watchpoint", cmd_w },
  { "d", "Delete watchpoint", cmd_d },
};
***
```

`/nemu/src/isa/$ISA/reg.c`:

```
void isa_reg_display() {
  int len = sizeof(regs) / sizeof(regs[0]);
  for (int i = 0; i < len; i++) {
    printf("%-15s0x%08lx%20ld\n", regs[i], cpu.gpr[i], cpu.gpr[i]);;
  }
}
```

PA1 代码参考：https://www.cnblogs.com/nosae/p/17045249.html

PA2

阅读资料：

- CPU架构之争：RISC的诞生与发展缩影
- See MIPS Run
- Computer Architecture A Quantitative Approach (5th edition)
  - power, cache, multicore processor, IS parallelism, vector architectures
- RISC-V 开放架构设计之道
  - 制程工艺先进，缩小栅极间的距离，减小晶粒面积，降低生成成本

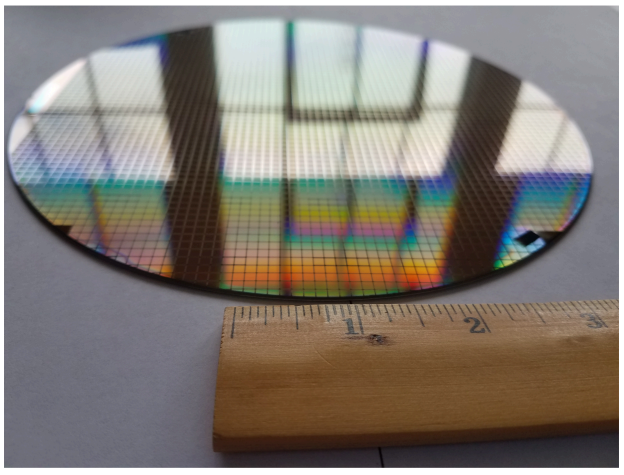


图 1.4: 由 SiFive 设计的直径 8 英寸的 RISC-V 芯片晶圆。它包含两类 RISC-V 芯片，均使用较旧较大的工艺制程。其中 FE310 芯片的尺寸为2.65 mm×2.72 mm，SiFive 测试芯片的尺寸为2.89 mm×2.72 mm。该晶圆包含总计 3172 个芯片，前者 1846 个，后者 1866 个。

运行第一个C程序

初始化 am-kernels 文件夹：

```
bash init.sh am-kernels
```

安装依赖：

```
sudo pacman -Syu riscv64-elf-gcc riscv64-linux-gnu-gcc
```

在 `am-kernels/tests/cpu-tests/` 目录下键入

```
make ARCH=riscv64-nemu ALL=dummy run
```

运行时报错:

```
ics2021* $ make -b ARCH=$ISA-nemu ALL=dummy run    [9:52:00]

Makefile.dummy:3: /Makefile: 没有那个文件或目录
make[1]: *** 没有规则可制作目标“/Makefile”。 停止。
test list [1 item(s)]: dummy
[          dummy] ***FAIL***
```

添加 `-n` 参数查看预执行指令，猜想是没有 `AM_HOME` 环境变量。

```
ics2021* $ make -n ARCH=riscv64-nemu ALL=dummy run

/bin/echo -e "NAME = dummy\nSRCS = tests/dummy.c\ninclude ${AM_HOME}/Makefile" > Makefile.dummy
if make -s -f Makefile.dummy ARCH=riscv64-nemu run; then \
        printf "[%14s] \033[1;32mPASS\033[0m\n" dummy >> .result; \
else \
        printf "[%14s] \033[1;31m***FAIL***\033[0m\n" dummy
```

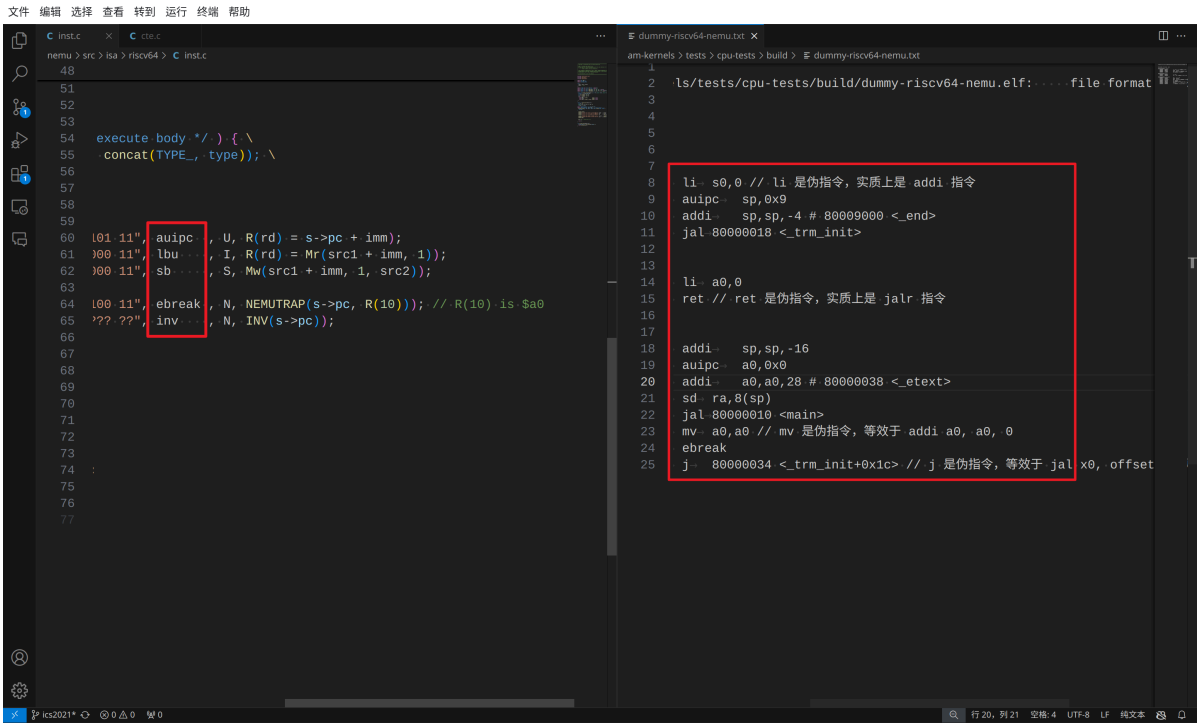
添加环境变量到 `~/.zshrc`:

```
export AM_HOME=/home/bk/riscv-cpu/ics2024/abstract-machine
```

执行 `source ~/.zshrc` 后再次键入 `make ARCH=riscv64-nemu ALL=dummy run` 成功执行:

```
ics2021* $ make ARCH=riscv64-nemu ALL=dummy run                                    [10:11:30]

# Building dummy-run [riscv64-nemu]
# Building am-archive [riscv64-nemu]
+ CC src/platform/nemu/trm.c
+ AR -> build/am-riscv64-nemu.a
# Building klib-archive [riscv64-nemu]
+ LD -> build/dummy-riscv64-nemu.elf
# Creating image [riscv64-nemu]
+ OBJCOPY -> build/dummy-riscv64-nemu.bin
[src/utils/log.c:30 init_log] Log is written to /home/bk/riscv-cpu/ics2024/am-kernels/tests/cpu-tests/build/nemu-log.txt
[src/memory/paddr.c:50 init_mem] physical memory area [0x80000000, 0x87ffffff]
[src/monitor/monitor.c:61 load_img] The image is /home/bk/riscv-cpu/ics2024/am-kernels/tests/cpu-tests/build/dummy-riscv64-nemu.bin, size = 57
[src/monitor/monitor.c:28 welcome] Trace: ON
[src/monitor/monitor.c:29 welcome] If trace is enabled, a log file will be generated to record the trace. This may lead to a large log file. If it is not necessary, you can disable it in menuconfig
[src/monitor/monitor.c:32 welcome] Build time: 10:04:30, Sep  3 2024
Welcome to riscv64-NEMU!
For help, type "help"
(nemu) c
invalid opcode(PC = 0x0000000080000000):
        13 04 00 00 17 91 00 00 ...
        00000413 00009117...
There are two cases which will trigger this unexpected exception:
1. The instruction at PC = 0x0000000080000000 is not implemented.
2. Something is implemented incorrectly.
Find this PC(0x0000000080000000) in the disassembling result to distinguish which case it is.

If it is the first case, see
       _                         __  __                         _
      (_)                       |  \/  |                       | |
  _ __ _ ___  ___ ________   __ | \  / | __ _ _ __  _   _  __ _| |
 | '__| / __|/ __|______\ \ / / | |\/| |/ _` | '_ \| | | |/ _` | |
 | |  | \__ \ (__        \ V /  | |  | | (_| | | | | |_| | (_| | |
 |_|  |_|___/\___|        \_/   |_|  |_|\__,_|_| |_|\__,_|\__,_|_|

for more details.

If it is the second case, remember:
* The machine is always right!
* Every line of untested code is always wrong!

[src/cpu/cpu-exec.c:120 cpu_exec] nemu: ABORT at pc = 0x0000000080000000
[src/cpu/cpu-exec.c:88 statistic] host time spent = 155 us
[src/cpu/cpu-exec.c:89 statistic] total guest instructions = 1
[src/cpu/cpu-exec.c:90 statistic] simulation frequency = 6,451 inst/s
(nemu) q
test list [1 item(s)]: dummy
[          dummy] PASS
```

在 `/ics2024/am-kernels/tests/cpu-tests/build/dummy-riscv64-nemu.txt` 查看汇编结果（即需要实现的指令），通过对比 `inst.c` 中已实现的指令，发现需要补全的指令是 `addi`、`jal`、`jalr`、`sd`:



查阅 RISC-V 手册以及实验指导，理解如何添加对应指令。需添加的指令格式：

RV32I Base Instruction Set

imm[31:12]			rd	0110111	LUI
imm[31:12]			rd	0010111	AUIPC
imm[20|10:1|11|19:12]			rd	1101111	JAL
imm[11:0]	rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type

RV64I Base Instruction Set (in addition to RV32I)

imm[11:0]		rs1	110	rd	0000011	LWU
imm[11:0]		rs1	011	rd	0000011	LD
imm[11:5]	rs2	rs1	011	imm[4:0]	0100011	SD
000000	shamt	rs1	001	rd	0010011	SLLI
000000	shamt	rs1	101	rd	0010011	SRLI
010000	shamt	rs1	101	rd	0010011	SRAI
imm[11:0]		rs1	000	rd	0011011	ADDIW
0000000	shamt	rs1	001	rd	0011011	SLLIW
0000000	shamt	rs1	101	rd	0011011	SRLIW
0100000	shamt	rs1	101	rd	0011011	SRAIW
0000000	rs2	rs1	000	rd	0111011	ADDW
0100000	rs2	rs1	000	rd	0111011	SUBW
0000000	rs2	rs1	001	rd	0111011	SLLW
0000000	rs2	rs1	101	rd	0111011	SRLW
0100000	rs2	rs1	101	rd	0111011	SRAW

根据指令格式判断指令类别：

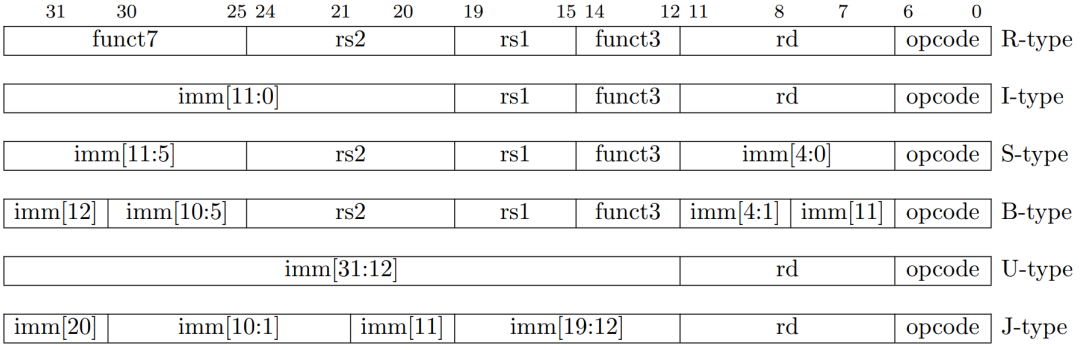


Figure 2.3: RISC-V base instruction formats showing immediate variants.

`addi` 指令功能：

Integer Register-Immediate Instructions

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

ADDI adds the sign-extended 12-bit immediate to register *rs1*. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result. ADDI *rd, rs1, 0* is used to implement the MV *rd, rs1* assembler pseudoinstruction.

`jal`、`jalr` 指令功能：

Unconditional Jumps

The jump and link (JAL) instruction uses the J-type format, where the J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the address of the jump instruction to form the jump target address. Jumps can therefore target a ±1 MiB range. JAL stores the address of the instruction following the jump (`pc`+4) into register *rd*. The standard software calling convention uses `x1` as the return address register and `x5` as an alternate link register.

The alternate link register supports calling millicode routines (e.g., those to save and restore registers in compressed code) while preserving the regular return address register. The register `x5` *was chosen as the alternate link register as it maps to a temporary in the standard calling convention, and has an encoding that is only one bit different than the regular link register.*

Plain unconditional jumps (assembler pseudoinstruction J) are encoded as a JAL with *rd*=`x0`.

31	30 21	20	19 12	11 7	6 0
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
1	10	1	8	5	7
	offset[20:1]			dest	JAL

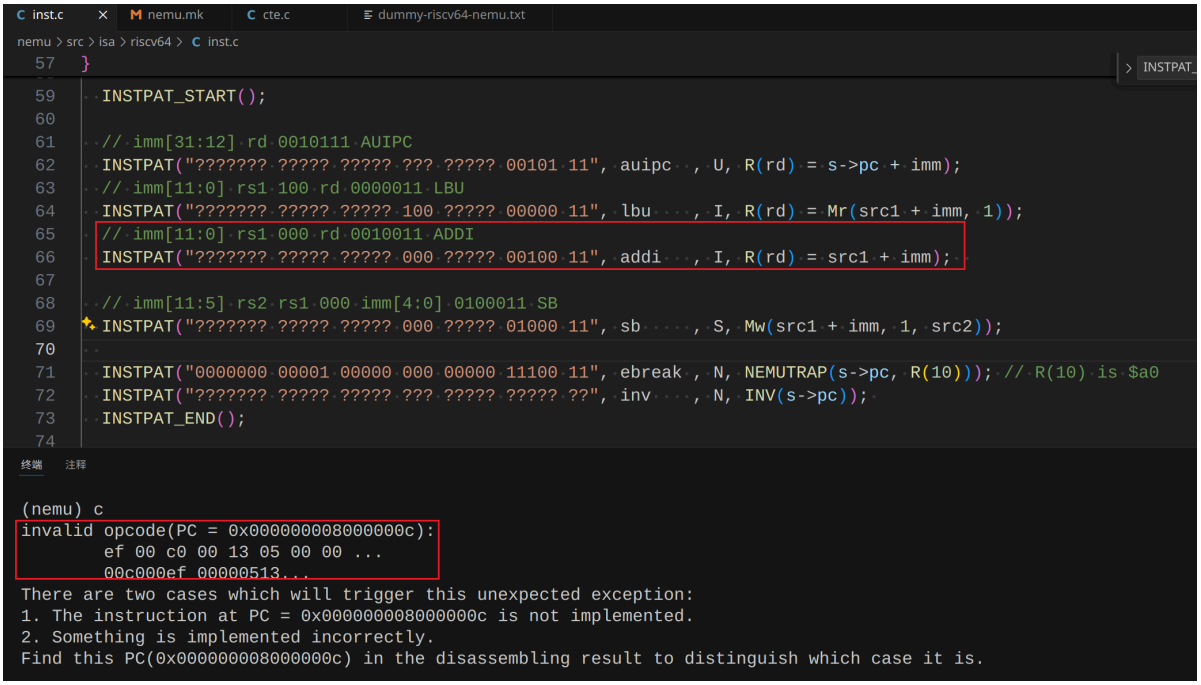
The indirect jump instruction JALR (jump and link register) uses the I-type encoding. The target address is obtained by adding the sign-extended 12-bit I-immediate to the register *rs1*, then setting the least-significant bit of the result to zero. The address of the instruction following the jump (`pc`+4) is written to register *rd*. Register `x0` can be used as the destination if the result is not required.

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	0	dest	JALR

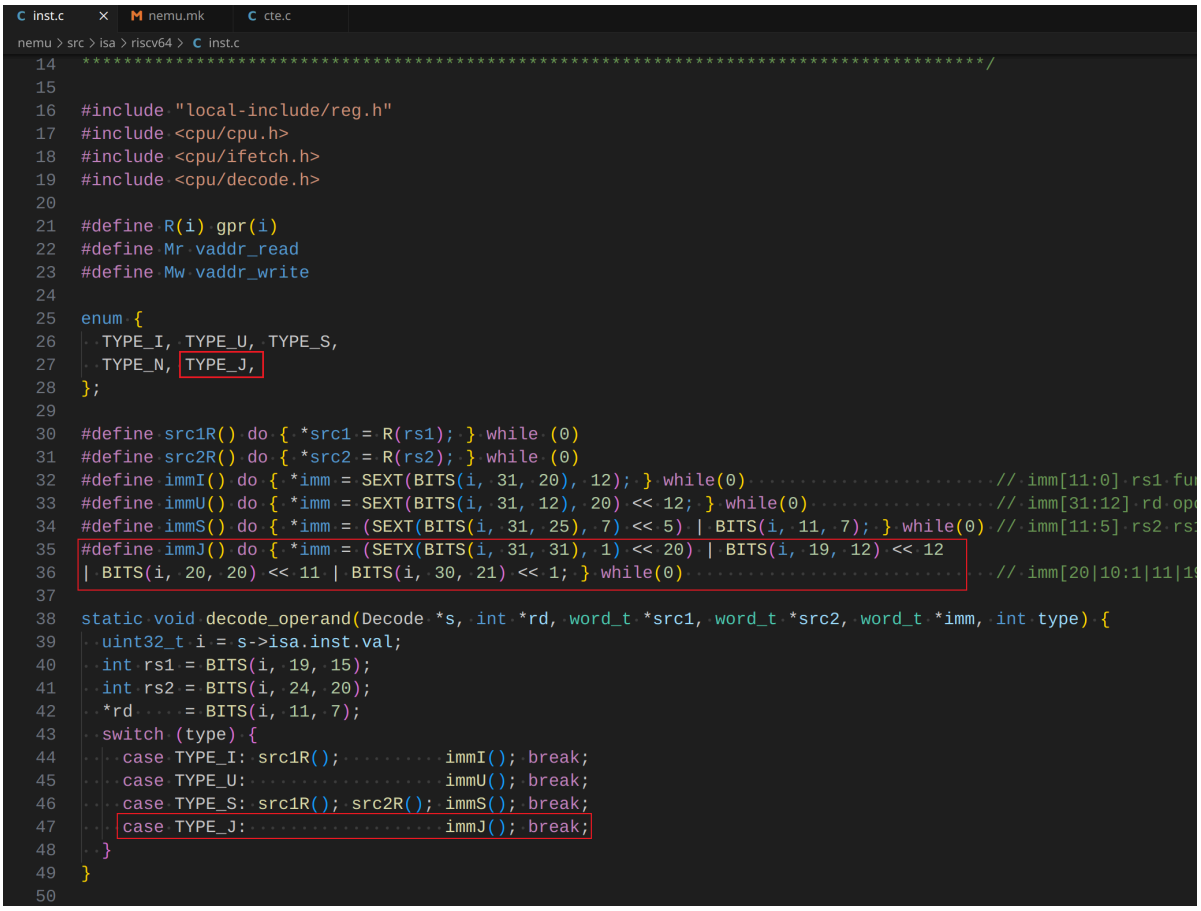
`sd` 指令功能：

The LW instruction loads a 32-bit value from memory and sign-extends this to 64 bits before storing it in register *rd* for RV64I. The LWU instruction, on the other hand, zero-extends the 32-bit value from memory for RV64I. LH and LHU are defined analogously for 16-bit values, as are LB and LBU for 8-bit values. The SD, SW, SH, and SB instructions store 64-bit, 32-bit, 16-bit, and 8-bit values from the low bits of register *rs2* to memory respectively.

添加 `addi` 指令之后，再次键入 `make ARCH=riscv64-nemu ALL=dummy run` 可以观察到 invalid opcode 已经发生了变化，说明指令继续往下执行了。



接下来如法炮制添加 `jal`、`jalr`、`sd` 指令，需要注意的是 `jal` 指令属于 J-Type，所以需要先补充 J-Type 类型的实现：

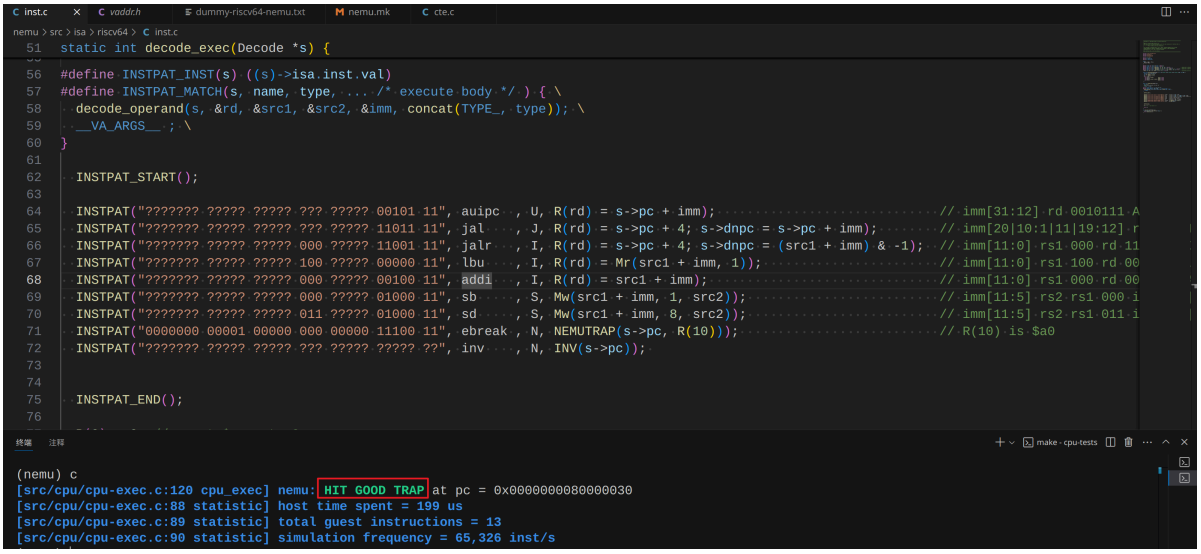


在实现 `jal`、`jalr` INSTPAT 时需要注意，当指令为跳转指令时，应当使用 `s->dnpc` 来更新 PC：

> 有了静态指令和动态指令这两个概念之后, 我们就可以说明`snpc`和`dnpc`的区别了: `snpc`是下一条静态指令, 而`dnpc`是下一条动态指令. 对于顺序执行的指令, 它们的`snpc`和`dnpc`是一样的; 但对于**跳转指令, `snpc`和`dnpc`就会有所不同, `dnpc`应该指向跳转目标的指令. 显然, 我们应该使用`s->dnpc`来更新PC**, 并且在指令执行的过程中正确地维护`s->dnpc`.

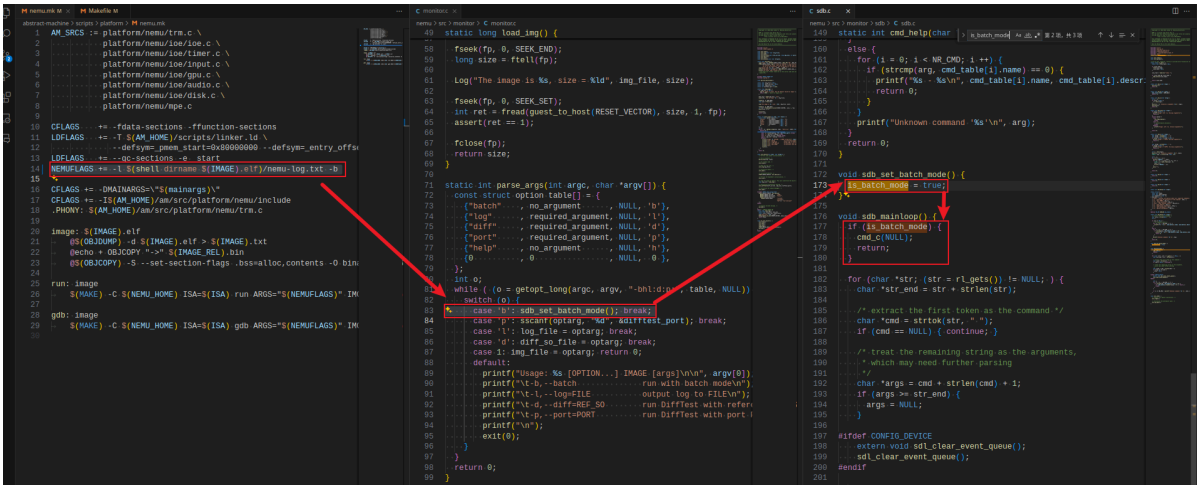
```
diff --git a/nemu/src/isa/riscv32/inst.c b/nemu/src/isa/riscv32/inst.c
index cb0c44e..0261636 100644
--- a/nemu/src/isa/riscv32/inst.c
+++ b/nemu/src/isa/riscv32/inst.c
@@ -24,14 +24,16 @@

 enum {
   TYPE_I, TYPE_U, TYPE_S,
-  TYPE_N, // none
+  TYPE_N, TYPE_J,
 };

 #define src1R() do { *src1 = R(rs1); } while (0)
 #define src2R() do { *src2 = R(rs2); } while (0)
-#define immI() do { *imm = SEXT(BITS(i, 31, 20), 12); } while(0)
-#define immU() do { *imm = SEXT(BITS(i, 31, 12), 20) << 12; } while(0)
-#define immS() do { *imm = (SEXT(BITS(i, 31, 25), 7) << 5) | BITS(i, 11, 7); } while(0)
+#define immI() do { *imm = SEXT(BITS(i, 31, 20), 12); } while(0)                     // imm[11:0] rs1 funct3 rd opcode I-type
+#define immU() do { *imm = SEXT(BITS(i, 31, 12), 20) << 12; } while(0)               // imm[31:12] rd opcode U-type
+#define immS() do { *imm = (SEXT(BITS(i, 31, 25), 7) << 5) | BITS(i, 11, 7); } while(0) // imm[11:5] rs2 rs1 funct3 imm[4:0] opcode S-type
+#define immJ() do { *imm = (SEXT(BITS(i, 31, 31), 1) << 20) | BITS(i, 19, 12) << 12 \
+| BITS(i, 20, 20) << 11 | BITS(i, 30, 21) << 1; } while(0)                         // imm[20|10:1|11|19:12] rd opcode J-type

 static void decode_operand(Decode *s, int *rd, word_t *src1, word_t *src2, word_t *imm, int type) {
   uint32_t i = s->isa.inst.val;
@@ -42,6 +44,7 @@ static void decode_operand(Decode *s, int *rd, word_t *src1, word_t *src2, word_
     case TYPE_I: src1R();          immI(); break;
     case TYPE_U:                   immU(); break;
     case TYPE_S: src1R(); src2R(); immS(); break;
+    case TYPE_J:                   immJ(); break;
   }
 }

@@ -57,12 +60,18 @@ static int decode_exec(Decode *s) {

   INSTPAT_START();
-  INSTPAT("??????? ????? ????? ??? ????? 00101 11", auipc  , U, R(rd) = s->pc + imm);
-  INSTPAT("??????? ????? ????? 100 ????? 00000 11", lbu    , I, R(rd) = Mr(src1 + imm, 1));
-  INSTPAT("??????? ????? ????? 000 ????? 01000 11", sb     , S, Mw(src1 + imm, 1, src2));

-  INSTPAT("0000000 00001 00000 000 00000 11100 11", ebreak , N, NEMUTRAP(s->pc, R(10))); // R(10) is $a0
-  INSTPAT("??????? ????? ????? ??? ????? ????? ??", inv    , N, INV(s->pc));
+  INSTPAT("??????? ????? ????? ??? ????? 00101 11", auipc  , U, R(rd) = s->pc + imm);                           // imm[31:12] rd 0010111 AUIPC
+  INSTPAT("??????? ????? ????? ??? ????? 11011 11", jal    , J, R(rd) = s->pc + 4; s->dnpc = s->pc + imm);     // imm[20|10:1|11|19:12] rd 1101111 JAL
+  INSTPAT("??????? ????? ????? 000 ????? 11001 11", jalr   , I, R(rd) = s->pc + 4; s->dnpc = (src1 + imm) & -1); // imm[11:0] rs1 000 rd 1100111 JALR
+  INSTPAT("??????? ????? ????? 100 ????? 00000 11", lbu    , I, R(rd) = Mr(src1 + imm, 1));                    // imm[11:0] rs1 100 rd 0000011 LBU
+  INSTPAT("??????? ????? ????? 000 ????? 00100 11", addi   , I, R(rd) = src1 + imm);                           // imm[11:0] rs1 000 rd 0010011 ADDI
+  INSTPAT("??????? ????? ????? 000 ????? 01000 11", sb     , S, Mw(src1 + imm, 1, src2));                      // imm[11:5] rs2 rs1 000 imm[4:0] 0100011 SB
+  INSTPAT("??????? ????? ????? 011 ????? 01000 11", sd     , S, Mw(src1 + imm, 8, src2));                      // imm[11:5] rs2 rs1 011 imm[4:0] 0100011 SD
+  INSTPAT("0000000 00001 00000 000 00000 11100 11", ebreak , N, NEMUTRAP(s->pc, R(10)));                       // R(10) is $a0
+  INSTPAT("??????? ????? ????? ??? ????? ????? ??", inv    , N, INV(s->pc));
+
+
   INSTPAT_END();
```

成功 HIT GOOD TRAP！



在 `abstract-machine/scripts/platform/nemu.mk` 添加 `-b` 参数进入批处理模式，避免键入 c 才能执行程序的麻烦。原理是因为键入 `-b` 参数后，会在 `monitor.c` 中执行 `sdb_set_batch_mode()`，将参数 `is_batch_mode` 设为 true，从而在 `sdb_mainloop()` 中执行 `cmd_c(NULL)`：

```
NEMUFLAGS += -l $(shell dirname $(IMAGE).elf)/nemu-log.txt -b
```



修改后的执行效果：

```
ics2021* $ make ARCH=riscv64-nemu ALL=dummy run
# Building dummy-run [riscv64-nemu]
# Building am-archive [riscv64-nemu]
+ CC src/platform/nemu/trm.c
+ AR -> build/am-riscv64-nemu.a
# Building klib-archive [riscv64-nemu]
+ LD -> build/dummy-riscv64-nemu.elf
# Creating image [riscv64-nemu]
+ OBJCOPY -> build/dummy-riscv64-nemu.bin
[src/utils/log.c:30 init_log] Log is written to /home/bk/riscv-cpu/ics2024/am-kernels/tests/cpu-tests/build/nemu-log.txt
[src/memory/paddr.c:50 init_mem] physical memory area [0x80000000, 0x87ffffff]
[src/monitor/monitor.c:61 load_img] The image is /home/bk/riscv-cpu/ics2024/am-kernels/tests/cpu-tests/build/dummy-riscv64-nemu.bin, size = 57
[src/monitor/monitor.c:28 welcome] Trace: ON
[src/monitor/monitor.c:29 welcome] If trace is enabled, a log file will be generated to record the trace. This may lead to a large log file. If it is not necessary, you can disable it in menuconfig
[src/monitor/monitor.c:32 welcome] Build time: 12:09:05, Sep  5 2024
Welcome to riscv64-NEMU!
For help, type "help"
[src/cpu/cpu-exec.c:120 cpu_exec] nemu: HIT GOOD TRAP at pc = 0x0000000080000030
[src/cpu/cpu-exec.c:88 statistic] host time spent = 53 us
[src/cpu/cpu-exec.c:89 statistic] total guest instructions = 13
[src/cpu/cpu-exec.c:90 statistic] simulation frequency = 245,283 inst/s
test list [1 item(s)]: dummy
[          dummy] PASS
```

遵照 tutorial，尝试以以其他 .c 作为测试文件，继续完善 `inst.c`。但其他测试文件基本都涉及到对 R-Type 指令的完善，可 R-Type 的完善并不成功，遂暂时跳过完善 `inst.c`，勉勉强强算是完成了 PA2 phase1。

> **温馨提示**
>
> PA2阶段1到此结束.

~~最高要求——任务3~~

~~尝试进阶，添加 cache / 分支预测~~

~~时间来不及玩不了~~

实验随记

挖掘漏洞的本质是发现计算机运行程序时进入了未定义状态，或者在已定义状态下遇到了不完全处理。现在就可以有两种视角来看待程序，一种是静态（代码本身），另一种是动态（状态跃迁）。

冯诺依曼体系结构核心：存储程序，程序控制

大家在学习的过程中会遇到各种各样的问题，伴随而来的可能还会有相当程度的挫败感和焦虑感

当我们去做难度大且有意义的事情时，不好解决的问题必然是客观存在的，当然解决问题的方法也是一定存在的。

有代码的地方就有基础设施，代码不是一次性用品，需要持续性维护，为了方便、快捷，必须构建基础设施（Makefile、git 等）保证代码的可用性。

有 bug 很正常（占比可能是 90% 及以上），记住**机器永远是对的**，耐心去调就行了（我们可是学习过计算机工作原理的人）！

有趣的东西：

- RTFM, STFW, RTFSC
- Unix 哲学最能体现 Linux 和 Windows 的区别: **编程创造**

事实上, 计算机系统的工作只做两件事情:

- make things work
- make things work better