

微处理器系统设计

Sep 5 23:55

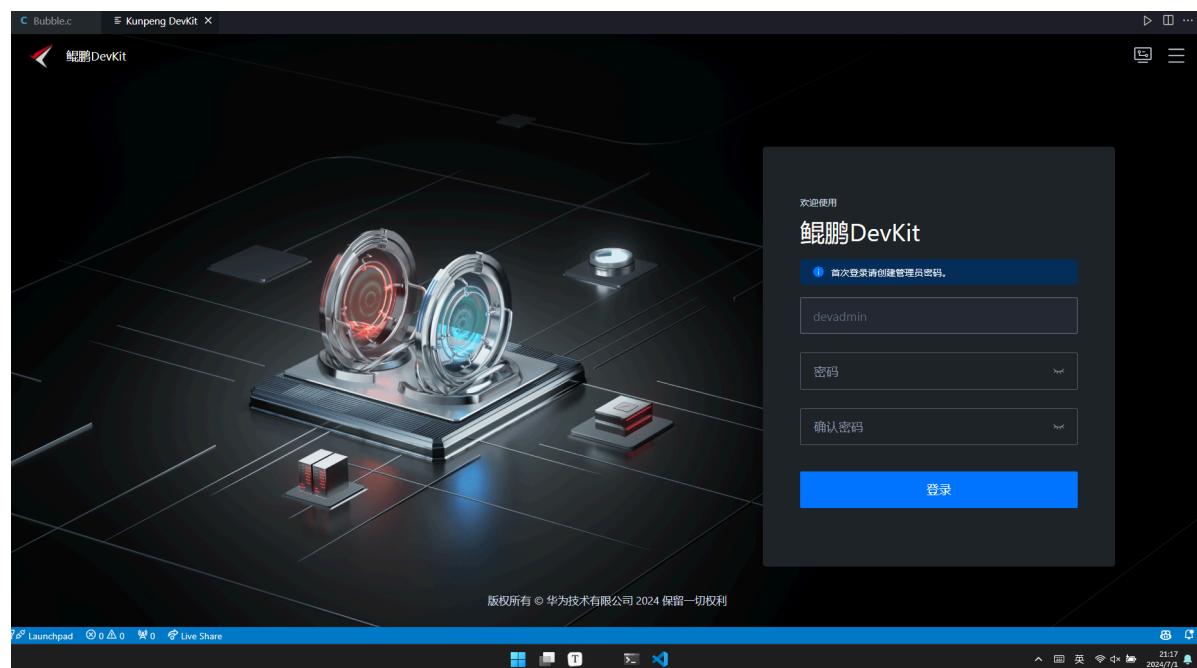
任务（可选）：

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](#)



A screenshot of the Kunpeng DevKit software interface, specifically the 'Code_migration' tab under the 'Code迁移' (Code Migration) section. The left sidebar shows a file tree with a 'BubbleSort' project selected, containing files like 'Bubble.asm', 'Bubble.c', and 'Bubble.obj'. The main pane displays the '配置信息' (Configuration Information) for the migration task, including the task name 'Code_migration', generation time '2024/07/01 21:22:04', source code path '/opt/DevKit/workspace/devadmin/porting/sourcecode/cpu', target system 'openeuler20.03', compiler 'BiSheng Compiler 2.5.0.1', build tool 'make', and analysis results showing 0 replacements, 0 verifications, and 0 dependencies. Below this, sections for '与架构相关的依赖文件' (Architecture-related dependency files) and '需要迁移的源码文件' (Source code files to be migrated) are shown, both currently empty. At the bottom, there are download buttons for '下载报告 (.csv)' and '下载报告 (.html)'.

```

CPU
  .vscode
  BubbleSort
    .vs
    Debug
    x64
    ASM Bubble.asm 9+
      BubbleSort > ASM Bubble.asm
      1 .model flat, c
      2 .code
      3 .Bubble proc
      4 push ebp This is x86 or arm32 instructions.
      5 mov ebp,esp The instruction shared on x86 and arm.
      6 sort:
      7 mov ebx,[ebp+8] The instruction shared on x86 and arm.
      8 mov eax,10 The instruction shared on x86 and arm.
      9 loop1:
      10 mov ecx,ecx The instruction shared on x86 and arm.
      11 dec ecx This is x86 or arm32 instructions.
      12 mov edx,ecx The instruction shared on x86 and arm.
      13 mov esi,0 The instruction shared on x86 and arm.
      14 loop2:
      15 mov ecx,[ebx+esi] The instruction shared on x86 and arm.
      16 cmp ecx,[ebx+esi+4] The instruction shared on x86 and arm.
      17 jl loop3 This is x86 or arm32 instructions.
      18 xchgb ecx,[ebx+esi+4] This is x86 or arm32 instructions.
      19 mov [ebx+esi],ecx The instruction shared on x86 and arm.
      20 loop3:
      21 add esi,4 The instruction shared on x86 and arm.
      22 dec edx This is x86 or arm32 instructions.
      23 jnz loop2 This is x86 or arm32 instructions.
      24
      25 dec eax This is x86 or arm32 instructions.
      26 cmp eax,1 The instruction shared on x86 and arm.
      27 jnz loop1 This is x86 or arm32 instructions.
      28 pop ebp This is x86 or arm32 instructions.
      29 ret The instruction shared on x86 and arm.
      Bubble endp
      End

```

Migration Assistant (左侧栏)

- 迁移
- 应用迁移
- 软件迁移评估
- 暂无历史报告, 请新建
- 源码迁移
- Code_migrat... 2024/07/01 21:22:0
- 软件包重构
- 专项软件迁移

Launchpad | Live Share | 行 4, 列 1 空格: 2 UTF-8 CRLF 纯文本

Archlinux VMWare Workstation (下方窗口)

命令行输出:

```

[root@localhost Bubblesort]# gcc Bubble.c Bubblesort -o Bubble
[root@localhost Bubblesort]# ./Bubble
Before sort:
1 9 7 6 4 3 0 2 8 5
after sort:
0 1 2 3 4 5 6 7 8 9
[root@localhost Bubblesort]# cat /proc/version
Linux version 4.19.90-2003.4.0.0036.oe1.aarch64 (abuild@obs-worker-003) (gcc version 7.3.0 (GCC) #1 SMP Mon Mar 23 19:06:43 UTC 2020
[root@localhost Bubblesort]#

```

得到的结果如图 8-22 所示:

根据执行结果, 可以看出排序成功, 接下来我们对比一下 ARM 冒泡

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
❯ pao* $ make run
+ 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, 0x8fffffff]
[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
❯ pao* $ |
```

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_DoubleControlSwitch.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_DoubleControlSwitch.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！

```
int main(int argc, char** argv) {
    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;
return 0;
}
```

a=1, b=1, f=0
a=1, b=0, f=1
a=1, b=0, f=1
a=1, b=1, f=0
a=1, b=1, f=0
a=0, b=0, f=0
a=0, b=0, f=0
a=0, b=0, f=0
a=1, b=0, f=1
a=0, b=1, f=1
a=1, b=1, f=0
a=1, b=0, f=1

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_DoubleControlSwitch.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);

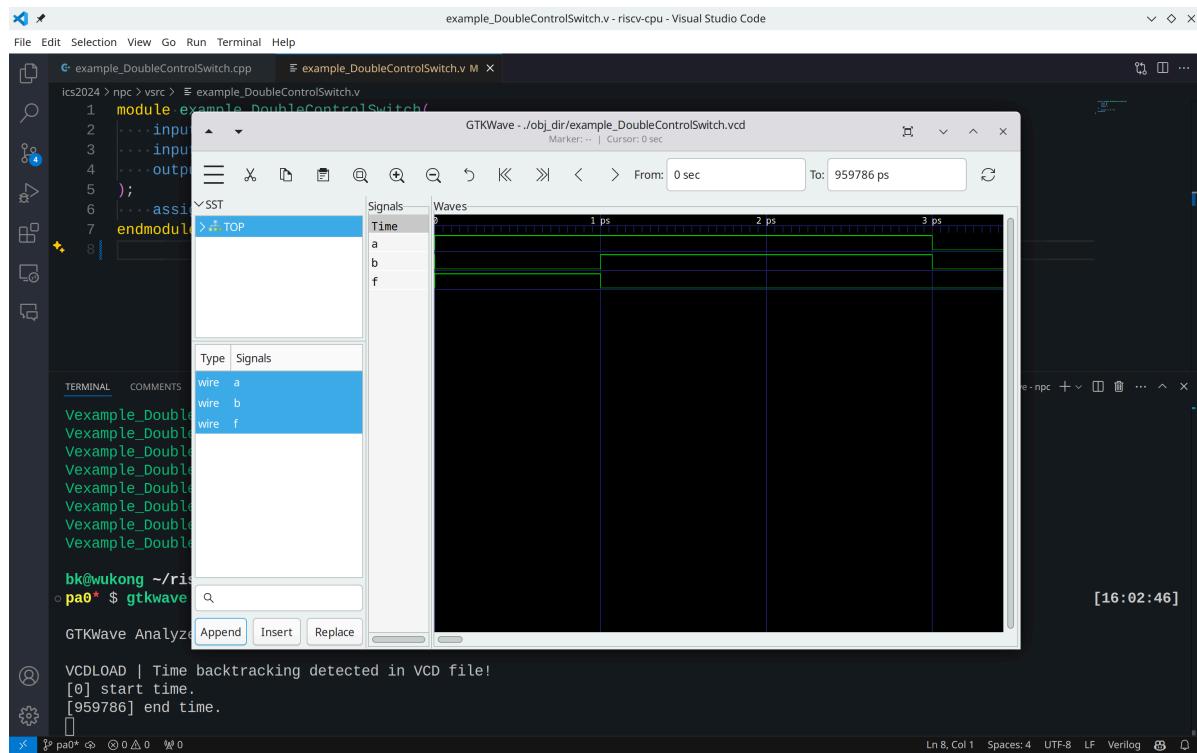
    assert(DoubleControlSwitch->f == (a ^ b));
}

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

```

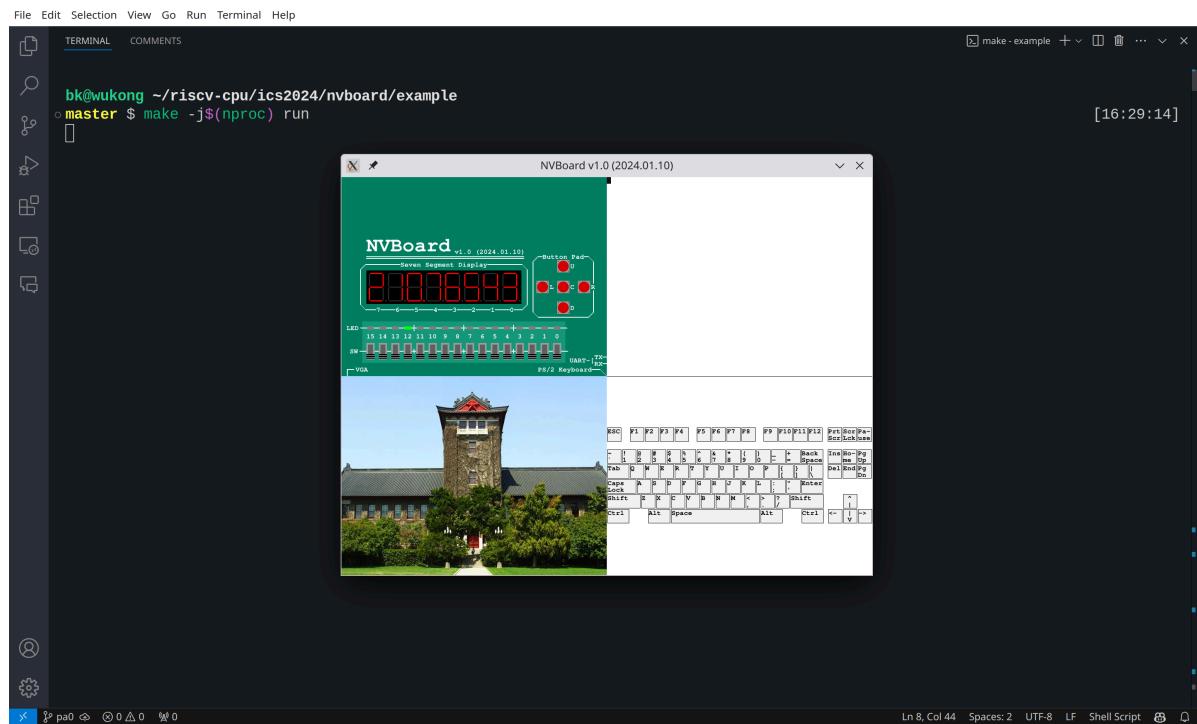
注意：

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 ./vsrsrc) -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 += $(INCFLAGS) -DTOP_NAME=""\\"$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

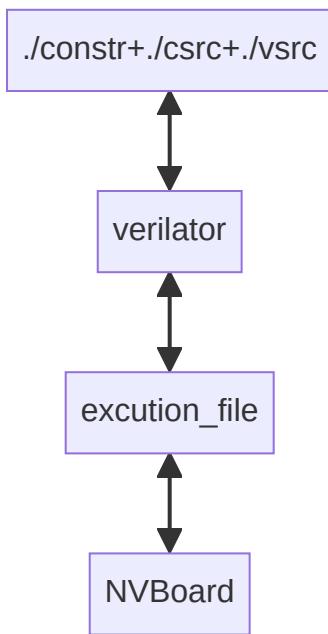
run: $(BIN)
    @$^

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.

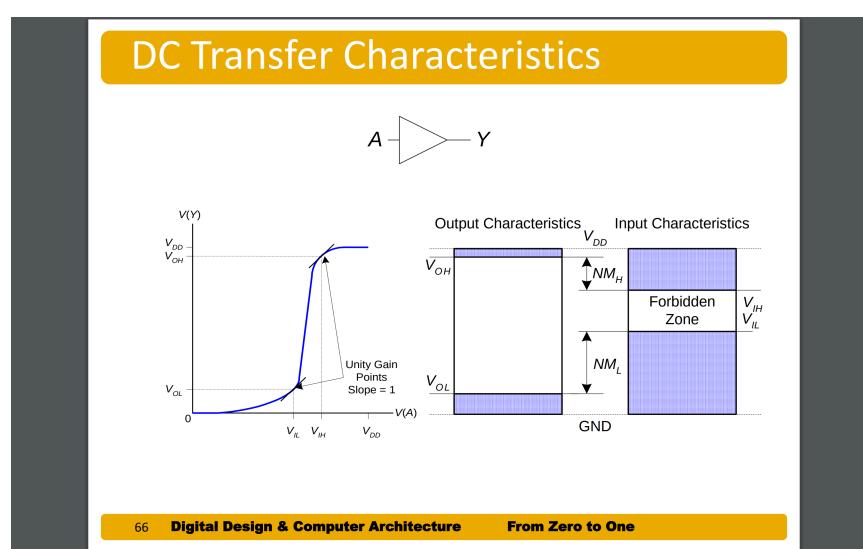
In procedural assignment, use **reg** type

`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)

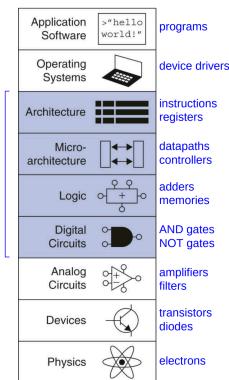


- 脑海中先有电路再有代码，verilog 本质上是硬件描述语言，真正的描述电路 = 实例化 + 连线

- 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.

Abstraction

Hiding details when they aren't important



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

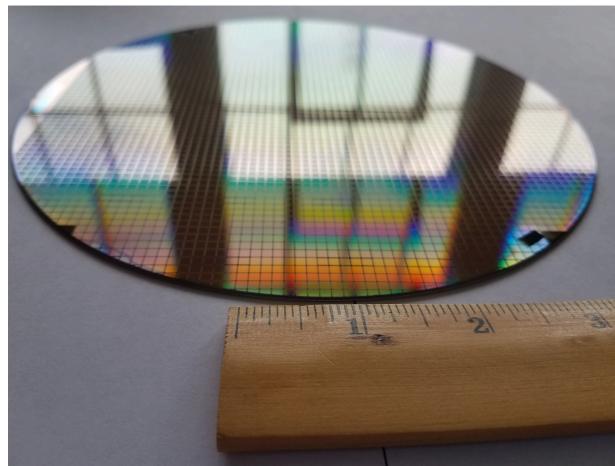


图 1.4: 由 SiFive 设计的直径 8 英寸的 RISC-V 芯片晶圆。它包含两类 RISC-V 芯片，均使用较旧较大的工艺制程。其中 FE310 芯片的尺寸为 2.65 mm × 2.72 mm，SiFive 测试芯片的尺寸为 2.89 mm × 2.72 mm。该晶圆包含总计 3172 个芯片，前者 1846 个，后者 1866 个。

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\nninclude ${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 Klib archive [riscv64-nemu]  
+ CC /usr/bin/riscv64-unknown-elf-gcc  
+ AR > build/lib/riscv64-nemu.a  
# Building Klib archive [riscv64-nemu]  
+ LD > build/dummy-riscv64-nemu.elf  
# Creating image [riscv64-nemu]  
+ OBJCOPY -g build/dummy-riscv64-nemu.bin  
[src/utils/log.c:38 init_log] Log is written to /home/bk/riscv-cpu/ics2024/am-kernels/tests/cpu-tests/build/nemu-log.txt  
[src/memory/paddr.c:59 init_mem] physical memory area [0x80000000, 0x87fffff]  
[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  
We need to RISC-V NEMU  
For help type "help"  
(nemu) r  
invalid opcode(PC = 0x0000000000000000):  
13 04 00 00 17 91 00 00 ...  
00000413 000009117...  
There are two cases which will trigger this unexpected exception:  
1. The instruction at PC = 0x0000000000000000 is not implemented.  
2. Something is implemented incorrectly.  
Find this PC(0x0000000000000000) in the disassembling result to distinguish which case it is.  
If it is the first case, see  
  
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 = 0x0000000000000000  
[src/cpu/cpu-exec.c:88 statistic] host time spent = 165 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`：

```
inst.c      c inst.c
nemu > src > riscv64 > c inst.c
48
51
52
53
54     execute_body /* ) { \
55         concat(TYPE_, type)); \
56
57
58
59
60 l01_11", auipc , U, R(rd) = s->pc + imm);
61 )00_11", lbu   , I, R(rd) = Mr(src1 + imm, 1);
62 )00_11", sb    , S, Mw(src1 + imm, 1, src2));
63
64 l00_11", ebreak , N, NEMUTRAP(s->pc, R(10)); // R(10).is-$a0
65 )??_??", inv   , N, INV(s->pc));
66
67
68
69
70
71
72
73
74 :
75
76
77
```

```
dummy-riscv64-nemu.txt
1
2     ls/tests/cpu-tests/build/dummy-riscv64-nemu.elf:.... file format
3
4
5
6
7
8 li s0,0 // li 是伪指令, 实质上是 addi 指令
9 auipc sp,0x9
10 addi sp,sp,-4 # 80000900 <_end>
11 jal 80000018 <_trm_init>
12
13
14 li a0,0
15 ret // ret 是伪指令, 实质上是 jalr 指令
16
17 addi sp,sp,-16
18 auipc a0,0x9
19 addi a0,a0,28 # 80000038 <_etext>
20 sd ra,8(sp)
21 jal 80000010 <main>
22 mv a0,a0 // mv 是伪指令, 等效于 addi a0, a0, 0
23 ebreak
24 j 80000034 <_trm_init+0x1c> // j 是伪指令, 等效于 jal x0, offset
```

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]		rd	1100111	JALR	
imm[12:10:5]	rs2	rs1	000	imm[4:1][11]	BEQ
imm[12:10:5]	rs2	rs1	001	imm[4:1][11]	BNE
imm[12:10:5]	rs2	rs1	100	imm[4:1][11]	BLT
imm[12:10:5]	rs2	rs1	101	imm[4:1][11]	BGE
imm[12:10:5]	rs2	rs1	110	imm[4:1][11]	BLTU
imm[12:10:5]	rs2	rs1	111	imm[4:1][11]	BGEU
imm[11:0]		rs1	000	rd	0000011
imm[11:0]		rs1	001	rd	0000011
imm[11:0]		rs1	010	rd	0000011
imm[11:0]		rs1	100	rd	0000011
imm[11:0]		rs1	101	rd	0000011
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011
imm[11:0]		rs1	000	rd	0010011
imm[11:0]		rs1	010	rd	0010011
imm[11:0]		rs1	011	rd	0010011
imm[11:0]		rs1	100	rd	0010011
imm[11:0]		rs1	110	rd	0010011
imm[11:0]		rs1	111	rd	0010011
00000000	shamt	rs1	001	rd	0010011
00000000	shamt	rs1	101	rd	0010011
01000000	shamt	rs1	101	rd	0010011
00000000	rs2	rs1	000	rd	0110011
01000000	rs2	rs1	000	rd	0110011
00000000	rs2	rs1	001	rd	0110011
00000000	rs2	rs1	010	rd	0110011
00000000	rs2	rs1	011	rd	0110011
00000000	rs2	rs1	100	rd	0110011
00000000	rs2	rs1	101	rd	0110011
01000000	rs2	rs1	101	rd	0110011
00000000	rs2	rs1	110	rd	0110011
00000000	rs2	rs1	111	rd	0110011
fm	pred	succ	rs1	000	rd
000000000000			00000	000	00000
000000000001			00000	000	00000
					1110011
					1110011
					EBREAK

Volume I: RISC-V Unprivileged ISA V20191213

131

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	011	imm[4:0]	opcode
-----------	-----	-----	-----	----------	--------

RV64I Base Instruction Set (in addition to RV32I)					
imm[11:0]	rs1	110	rd	00000011	LWU
imm[11:0]	rs1	011	rd	00000011	LD
imm[11:5]	rs2	rs1	011	imm[4:0]	0100011
000000	shamt	rs1	001	rd	0010011
000000	shamt	rs1	101	rd	0010011
010000	shamt	rs1	101	rd	0010011
000000	imm[11:0]	rs1	000	rd	0011011
00000000	shamt	rs1	001	rd	0011011
00000000	shamt	rs1	101	rd	0011011
01000000	shamt	rs1	101	rd	0011011
00000000	rs2	rs1	000	rd	0111011
01000000	rs2	rs1	000	rd	0111011
00000000	rs2	rs1	001	rd	0111011
01000000	rs2	rs1	101	rd	0111011
00000000	rs2	rs1	101	rd	0111011
01000000	rs2	rs1	101	rd	0111011
00000000	rs2	rs1	101	rd	0111011
01000000	rs2	rs1	101	rd	0111011
00000000	rs2	rs1	110	rd	0111011
01000000	rs2	rs1	111	rd	0111011
00000000	shamt	rs1	001	rd	0111011
00000000	shamt	rs1	101	rd	0111011
01000000	shamt	rs1	101	rd	0111011
00000000	imm[11:0]	rs1	000	rd	0111011
000000000000			00000	000	00000
000000000001			00000	000	00000
					1110011
					1110011
					SRRAW

根据指令格式判断指令类别：

31	30	25 24	21	20	19	15	14	12 11	8	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
	imm[12]	imm[10:5]	rs2		rs1		funct3	imm[4:1]	imm[11]		opcode		B-type
	imm[31:12]							rd		opcode			U-type
	imm[20]	imm[10:1]	imm[11]	imm[19:12]				rd		opcode			J-type

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。

The screenshot shows a terminal window with several tabs: `inst.c`, `nemu.mk`, `cte.c`, and `dummy-riscv64-nemu.txt`. The `inst.c` tab contains assembly code for various RISC-V instructions, including `addi`. The `dummy-riscv64-nemu.txt` tab shows the output of the emulator, which includes an error message:

```
(nemu) c
invalid opcode(PC = 0x000000008000000c):
    ef 00 c0 00 13 05 00 00 ...
    00c000ef 00000513...
```

There are two cases which will trigger this unexpected exception:
1. The instruction at PC = 0x000000008000000c is not implemented.
2. Something is implemented incorrectly.
Find this PC(0x000000008000000c) in the disassembling result to distinguish which case it is.

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：

The screenshot shows a terminal window with tabs: `inst.c`, `nemu.mk`, and `cte.c`. The `inst.c` tab contains C code for decoding and executing RISC-V instructions. A specific section for J-Type instructions is highlighted:

```
enum {
    TYPE_I, TYPE_U, TYPE_S,
    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) // .imm[11:0].rs1.fun
#define immU() do { *imm = SEXT(BITS(i, 31, 12), 20) << 12; } while(0) // .imm[31:12].rd.op
#define immS() do { *imm = (SEXT(BITS(i, 31, 25), 7) << 5) | BITS(i, 11, 7); } while(0) // .imm[11:5].rs2.rs
#define immJ() do { *imm = ((SETX(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|15)
```

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

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.vval;
@@ -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("???????? ?????? ?????? ??? ?????? 0010111", 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("00000000 00001 000 000 00000 11100 11", ebreak , N, NEMUTRAP(s->pc, R(10))); // R(10) is $a0
- INSTPAT("???????? ?????? ?????? ??? ???, inv   , N, INV(s->pc));
+ INSTPAT("???????? ?????? ?????? ??? ?????? 0010111", auipc , U, R(rd) = s->pc + imm); // imm[31:12] rd 0010111 AUIP
+ INSTPAT("???????? ?????? ?????? ??? ?????? 1100111", jal    , J, R(rd) = s->pc + 4; s->dmpc = s->pc + imm); // imm[20:10:1|11|19:12] rd 1
+ INSTPAT("???????? ?????? ?????? 000 ?????? 1100011", jalr   , I, R(rd) = s->pc + 4; s->dmpc = (src1 + imm) & -1); // imm[11:0] rs1 000 rd 110001
+ INSTPAT("???????? ?????? ?????? 100 ?????? 00000 11", lbu    , I, R(rd) = Mr(src1 + imm, 1)); // imm[11:0] rs1 100 rd 000000
+ INSTPAT("???????? ?????? ?????? 000 ?????? 00100 11", addi   , I, R(rd) = src1 + imm); // imm[11:0] rs1 000 rd 001000
+ INSTPAT("???????? ?????? ?????? 000 ?????? 01000 11", sb     , S, Mw(src1 + imm, 1, src2)); // imm[11:5] rs2 rs1 000 011 imm[11:0]
+ INSTPAT("00000000 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
```

```

nemu> sccs > platform > scripts > platform > nemu.mk
...
11 AM_SRCS := platform/nemu/trm.c \
12          platform/nemu/10e/timer.c \
13          platform/nemu/10e/input.c \
14          platform/nemu/10e/gpu.c \
15          platform/nemu/10e/disk.c \
16          platform/nemu/npe.c
17
18 CFLAGS += -fdata-sections -ffunction-sections
19 LDFLAGS += -T $(AM_HOME)/scripts/linker.ld \
20            -Wl,-defsym=_mem_start=0x90000000 -Wl,-defsym=_entry_offset=0
21 LDFLAGS += --o-sections -e start
22 NEMUFLAGS += -l $(shell dirname $(IMAGE).elf)/nemu-log.txt -b
23
24 CFLAGS += -OMAINARGSS=$(mainargs)
25 CFLAGS += -I$(AM_HOME)/am/src/platform/nemu/include
26 .PHONY: $(AM_HOME)/am/src/platform/nemu/trm.c
27
28 image: $(IMAGE).elf > $(IMAGE).txt
29 $(OBJS) + $(OBCOPY) "+^" $(IMAGE_REL).bin
30 $(OBCOPY) -S $(OBCOPY) -5 --set-section-flags bss=alloc,contents -0 bin
31
32 run: image
33 $(NAME) -C $(NEMU_HOME) ISA=$(ISA) run ARGS=$(NEMUFLAGS) IM
34
35 gdb: image
36 $(NAME) -C $(NEMU_HOME) ISA=$(ISA) gdb ARGS=$(NEMUFLAGS) IM
37
38

```

```

nemu> ses > monitor > c_monitors.c
...
40 static long load_img() {
41     rseek(fp, 0, SEEK_SET);
42     long size = ftell(fp);
43
44     Log("The image is %s, size = %ld", img_file, size);
45     if (size < 0) {
46         perror("read error");
47         return -1;
48     }
49     if (fclose(fp) != 0) {
50         perror("close error");
51         return -1;
52     }
53     return size;
54 }
55
56 static int parse_argv(int argc, char *argv[]) {
57     const struct option table[] = {
58         {"batch", no_argument, NULL, 'b'},
59         {"log", required_argument, NULL, 'l'},
60         {"diff", required_argument, NULL, 'd'},
61         {"ifile", required_argument, NULL, 'i'},
62         {"help", no_argument, NULL, 'h'},
63         {"port", required_argument, NULL, 'p'},
64         {"t", 0, 0, "trace"}, {"NULL", 0, 0, 0}
65     };
66
67     int opt;
68     while ((opt = getopt_long(argc, argv, "-bhlp:i:p:t:", table, NULL)) != -1) {
69         switch (opt) {
70             case 'b':
71                 adb_set_batch_mode(); break;
72             case 'l':
73                 p->sscanf(optarg, "%d", &difftest_port); break;
74             case 'd':
75                 log_file = optarg; break;
76             case 'i':
77                 img_file = optarg; return 0;
78             default:
79                 printf("Usage: %s [OPTION]... IMAGE [args]\n", argv[0]);
80                 printf("  -b, --batch           run in batch mode\n");
81                 printf("  -l, --logFILE        output log to FILE\n");
82                 printf("  -d, --diff=REF SO   run DiffTest with reference\n");
83                 printf("  -i, --imgFILE=PORT  run DiffTest with port\n");
84                 printf("\n");
85                 exit(0);
86         }
87     }
88     return 0;
89 }
90

```

```

nemu> ses > monitor > c_sdb.c
...
140 static int cmd_help(char *cmd, mode_t mode) {
141     for (i = 0, i < NR_CMD, i++) {
142         if (strcmp(cmd, cmd_table[i].name) == 0) {
143             printf("%s: %s\n", cmd_table[i].name, cmd_table[i].desc);
144             return 0;
145         }
146     }
147     printf("Unknown command '%s'\n", arg);
148
149 }
150
151 void sdb_set_batch_mode() {
152     if (!sdb_batchloop())
153         if (sdb_batchloop())
154             cmd = NULL;
155
156     return;
157 }
158
159 for (char *str; (str = rl_gets()) != NULL; ) {
160     char *str_end = str + strlen(str);
161
162     /* extract the first token as the command */
163     char *cmd = strtok(str, " ");
164     if (cmd == NULL) { continue; }
165
166     /* treat the remaining string as the arguments,
167      * which may need further parsing
168      */
169     char *args = cmd + strlen(cmd) + 1;
170     if (args == str_end) {
171         args = NULL;
172     }
173
174 #ifdef CONFIG_DEVICE
175     extern void sdb_clear_event_queue();
176     sdb_clear_event_queue();
177 #endif
178 }
179
200#endif

```

修改后的执行效果：

```

ics2024$ make ARCH=riscv64-nemu ALL=dummy run
# Building dummy-run [riscv64-nemu]
# Building am-archive [riscv64-nemu]
+ CC src/platform/nemu/trm.c
AR riscv64-nemu.a src/platform/nemu.a
Building klib-archive [riscv64-nemu]
+ LD -r build/dummy-riscv64-nemu.elf
# Creating image [riscv64-nemu]
+ OBJCOPY -O binary build/dummy-riscv64-nemu.bin
[src/utils/log.c:38 init_log] Log is written to /home/bk/riscv-cpu/ics2024/am-kernels/tests/cpu-tests/build/nemu-log.txt
[src/memory/paddr.c:58 init_mem] physical memory area [0x80000000, 0x87fffff]
[src/monitor/monitor.c:65 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:29 welcome] If trace is enabled, a long 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 = 0x0000000000000030
[src/cpu/cpu-exec.c:98 statistic] host time spent = 83 us
[src/cpu/cpu-exec.c:99 statistic] total guest instructions = 13
[src/cpu/cpu-exec.c:99 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。



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

