湖南大學

HUNAN UNIVERSITY

课 程 名 称	：	专业综合设计
实验项目名称	：	一生一芯：riscv64 cpu设计报告
专 业 班 级	：	物联2001班
姓 名	：	郭光沛
学 号	：	202001120131
指 导 老 师	：	吴强
完 成 时 间	：	2023年9月5日

信息科学与工程学院

目录

1 环境搭建

1.1 系统选择

根据 ysyx 相关资料，其使用的环境各种依赖都是较新的版本。

于是这里使用的是 archlinux 下的 manjaro-gnome 发行版，使用 aur 作为下载软件镜像源，其提供的各种支持包几乎都是最新版本，适合该实验的环境搭建。

1.2 测试环境

1.2.1 nemu

获取“一生一芯”框架代码，使用了南京大学 PA 实验的环境 nemu：

```
git clone -b ysyx2204 git@github.com:OSCPU/ysyx-workbench.git ics2022
```

> 这里为了对接 PA0 讲义，项目目录设为与 PA0 相同

根据 PA0 讲义初始化各个目录后，框架结构如下：

```
ics2022
├── abstract-machine    # 抽象计算机
├── am-kernels          # 基于抽象计算机开发的应用程序
├── fceux-am            # 红白机模拟器
├── init.sh             # 初始化脚本
├── Makefile            # 用于工程打包提交
├── nemu                # NEMU
└── README.md
```

NEMU主要由4个模块构成: monitor、CPU、memory、设备，框架如下：

```
nemu
├── configs                     # 预先提供的一些配置文件
├── include                     # 存放全局使用的头文件
│   ├── common.h                # 公用的头文件
│   ├── config                  # 配置系统生成的头文件, 用于维护配置选项更新的时间戳
│   ├── cpu
│   │   ├── cpu.h
│   │   ├── decode.h            # 译码相关
│   │   ├── difftest.h
│   │   └── ifetch.h            # 取指相关
│   ├── debug.h                 # 一些方便调试用的宏
│   ├── device                  # 设备相关
│   ├── difftest-def.h
│   ├── generated
│   │   └── autoconf.h          # 配置系统生成的头文件, 用于根据配置信息定义相关的宏
│   ├── isa.h                   # ISA相关
│   ├── macro.h                 # 一些方便的宏定义
│   ├── memory                  # 访问内存相关
│   └── utils.h
├── Kconfig                     # 配置信息管理的规则
├── Makefile                    # Makefile构建脚本
├── README.md
├── resource                    # 一些辅助资源
├── scripts                     # Makefile构建脚本
```

```
│   ├── build.mk
│   ├── config.mk
│   ├── git.mk                  # git版本控制相关
│   └── native.mk
├── src                         # 源文件
│   ├── cpu
│   │   └── cpu-exec.c          # 指令执行的主循环
│   ├── device                  # 设备相关
│   ├── engine
│   │   └── interpreter         # 解释器的实现
│   ├── filelist.mk
│   ├── isa                     # ISA相关的实现
│   │   ├── mips32
│   │   ├── riscv32
│   │   ├── riscv64
│   │   └── x86
│   ├── memory                  # 内存访问的实现
│   ├── monitor
│   │   ├── monitor.c
│   │   └── sdb                 # 简易调试器
│   │       ├── expr.c          # 表达式求值的实现
│   │       ├── sdb.c           # 简易调试器的命令处理
│   │       └── watchpoint.c    # 监视点的实现
│   ├── nemu-main.c             # 你知道的...
│   └── utils                   # 一些公共的功能
│       ├── log.c               # 日志文件相关
│       ├── rand.c
│       ├── state.c
│       └── timer.c
└── tools                       # 一些工具
    ├── fixdep                  # 依赖修复, 配合配置系统进行使用
    ├── gen-expr
    ├── kconfig                 # 配置系统
    ├── kvm-diff
    ├── qemu-diff
    └── spike-diff
```

框架部分的说明在 PA1->RTFSC 部分

nemu 指令如下:

命令	格式	使用举例	说明
帮助(1)	help	help	打印命令的帮助信息
继续运行(1)	c	c	继续运行被暂停的程序
退出(1)	q	q	退出NEMU
单步执行	si [N]	si 10	让程序单步执行N条指令后暂停执行，当N没有给出时, 缺省为1
打印程序状态	info SUBCMD	info r	打印监视点信息
扫描内存(2)	x N EXPR	x 10 $esp	求出表达式EXPR的值, 将结果作为起始内存地址, 以十六进制形式输出连续的N个4字节
表达式求值	p EXPR	p $eax + 1	求出表达式EXPR的值, EXPR支持的运算请见调试中的表达式求值小节
设置监视点	w EXPR	w *0x2000	当表达式EXPR的值发生变化时, 暂停程序执行
删除监视点	d N	d 2	删除序号为N的监视点

第一次运行，会需要我们删除两行代码，使其正常运行：

PA1->RTFSC->准备第一个客户程序

```
  static void welcome() {
    Log("Trace: %s", MUXDEF(CONFIG_TRACE, ANSI_FMT("ON", ANSI_FG_GREEN),
ANSI_FMT("OFF", ANSI_FG_RED)));
    IFDEF(CONFIG_TRACE, Log("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"));
    Log("Build time: %s, %s", __TIME__, __DATE__);
    printf("Welcome to %s-NEMU!\n", ANSI_FMT(str(__GUEST_ISA__), ANSI_FG_YELLOW
ANSI_BG_RED));
    printf("For help, type \"help\"\n");
-   Log("Exercise: Please remove me in the source code and compile NEMU again.");
-   assert(0);
  }
```

1.2.2 verilator

根据实验文档，载入 npc 仓库

```
cd ysyx-workbench
bash init.sh npc
source ~/.bashrc
```

项目结构如下

```
ysyx-workbench/npc
├── csrc
│   └── main.cpp
├── Makefile
└── vsrc
    └── example.v
```

安装 verilator，verilator 会将 verilog 代码编译成静态库后再与 cpp/c 文件链接。

```
yay verilator
```

由于 arch 环境的不同，这里使用的 verilator 版本为 `Verilator 5.010 2023-04-30 rev UNKNOWN.REV`

尝试编写双控开关的代码：

`/npc/vsrc/example_DoupleControlSwitch.v`

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

`/npc/csrc/example_DoupleControlSwitch.cpp`

```
#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;
}
```

编译并运行

```
verilator --cc --exe --build -j 0 -Wall ./csrc/example_DoubleControlSwitch.cpp
./vsrc/example_DoubleControlSwitch.v
./obj_dir/Vexample_DoubleControlSwitch
```

```
 ~/M/riscv_cpudesigner/ics2022/npc  ysyx2204 ⇡5 !1 ?2  verilator --cc --exe --build -j 0 -Wall ./csrc/example_DoubleControlSwitch.cpp ./vsrc/example_DoubleControlSwitch.v
make: 进入目录“/home/noionion/MyPersonProject/riscv_cpudesigner/ics2022/npc/obj_dir”
g++  -I.  -MMD -I/usr/share/verilator/include -I/usr/share/verilator/include/vltstd -DVM_COVERAGE=0 -DVM_SC=0 -DVM_TRACE=0 -DVM_TRACE_FST=0 -DVM_TRACE_VCD=0 -faligned-new -fcf-protection=none -Wno-bool-operation -Wno-sign-compare -Wno-uninitialized -Wno-unused-but-set-variable -Wno-unused-parameter -Wno-unused-variable -Wno-shadow       -Os -c -o example_DoubleControlSwitch.o ../csrc/example_DoubleControlSwitch.cpp
g++ -Os  -I.  -MMD -I/usr/share/verilator/include -I/usr/share/verilator/include/vltstd -DVM_COVERAGE=0 -DVM_SC=0 -DVM_TRACE=0 -DVM_TRACE_FST=0 -DVM_TRACE_VCD=0 -faligned-new -fcf-protection=none -Wno-bool-operation -Wno-sign-compare -Wno-uninitialized -Wno-unused-but-set-variable -Wno-unused-parameter -Wno-unused-variable -Wno-shadow       -c -o verilated.o /usr/share/verilator/include/verilated.cpp
g++ -Os  -I.  -MMD -I/usr/share/verilator/include -I/usr/share/verilator/include/vltstd -DVM_COVERAGE=0 -DVM_SC=0 -DVM_TRACE=0 -DVM_TRACE_FST=0 -DVM_TRACE_VCD=0 -faligned-new -fcf-protection=none -Wno-bool-operation -Wno-sign-compare -Wno-uninitialized -Wno-unused-but-set-variable -Wno-unused-parameter -Wno-unused-variable -Wno-shadow       -c -o verilated_threads.o /usr/share/verilator/include/verilated_threads.cpp
/usr/bin/python3 /usr/share/verilator/bin/verilator_includer -DVL_INCLUDE_OPT=include Vexample_DoubleControlSwitch.cpp Vexample_DoubleControlSwitch___024root__DepSet_ha8c5a5d6__0.cpp Vexample_DoubleControlSwitch___024root__DepSet_h378d4e57__0.cpp Vexample_DoubleControlSwitch___024root__Slow.cpp Vexample_DoubleControlSwitch___024root__DepSet_ha8c5a5d6__0__Slow.cpp Vexample_DoubleControlSwitch___024root__DepSet_h378d4e57__0__Slow.cpp Vexample_DoubleControlSwitch__Syms.cpp > Vexample_DoubleControlSwitch__ALL.cpp
echo "" > Vexample_DoubleControlSwitch__ALL.verilator_deplist.tmp
g++ -Os  -I.  -MMD -I/usr/share/verilator/include -I/usr/share/verilator/include/vltstd -DVM_COVERAGE=0 -DVM_SC=0 -DVM_TRACE=0 -DVM_TRACE_FST=0 -DVM_TRACE_VCD=0 -faligned-new -fcf-protection=none -Wno-bool-operation -Wno-sign-compare -Wno-uninitialized -Wno-unused-but-set-variable -Wno-unused-parameter -Wno-unused-variable -Wno-shadow       -c -o Vexample_DoubleControlSwitch__ALL.o Vexample_DoubleControlSwitch__ALL.cpp
Archive ar -rcs Vexample_DoubleControlSwitch__ALL.a Vexample_DoubleControlSwitch__ALL.o
g++    example_DoubleControlSwitch.o verilated.o verilated_threads.o Vexample_DoubleControlSwitch__ALL.a    -pthread -lpthread -latomic   -o Vexample_DoubleControlSwitch
rm Vexample_DoubleControlSwitch__ALL.verilator_deplist.tmp
make: 离开目录“/home/noionion/MyPersonProject/riscv_cpudesigner/ics2022/npc/obj_dir”
a = 0, b = 0, f = 0
a = 0, b = 1, f = 1
a = 0, b = 0, f = 0
a = 0, b = 1, f = 1
a = 1, b = 0, f = 1
a = 1, b = 0, f = 1
a = 1, b = 0, f = 1
a = 0, b = 1, f = 1
a = 0, b = 0, f = 0
a = 0, b = 1, f = 1
a = 0, b = 1, f = 1
a = 0, b = 1, f = 1
a = 0, b = 0, f = 0
a = 1, b = 0, f = 1
a = 0, b = 0, f = 0
a = 1, b = 0, f = 1
a = 0, b = 1, f = 1
a = 1, b = 0, f = 1
a = 1, b = 1, f = 0
a = 1, b = 0, f = 1
a = 1, b = 1, f = 0
a = 0, b = 1, f = 1
a = 1, b = 1, f = 0
a = 0, b = 1, f = 1
a = 1, b = 0, f = 1
a = 0, b = 0, f = 0
a = 1, b = 0, f = 1
a = 1, b = 1, f = 0
a = 0, b = 0, f = 0
a = 0, b = 0, f = 0
a = 0, b = 1, f = 1
a = 1, b = 0, f = 1
a = 1, b = 0, f = 1
a = 0, b = 0, f = 0
a = 0, b = 0, f = 0
a = 1, b = 1, f = 0
a = 1, b = 1, f = 0
a = 0, b = 0, f = 0
a = 0, b = 1, f = 1
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 = 0, b = 1, f = 1
a = 1, b = 0, f = 1
a = 0, b = 0, f = 0
a = 0, b = 1, f = 1
a = 0, b = 1, f = 1
a = 1, b = 1, f = 0
a = 1, b = 1, f = 0
```

2 环境适配

2.1 nemu RTFSC

在 PA1 的实验中，会让我们在 `am-kernels/tests/cpu-tests` 文件夹下以如下指令运行 `dummy` 样例：

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

阅读相关的 makefile：`abstract-machine/scripts/platform/nemu.mk`

```
AM_SRCS := platform/nemu/trm.c \
           platform/nemu/ioe/ioe.c \
           platform/nemu/ioe/timer.c \
           platform/nemu/ioe/input.c \
           platform/nemu/ioe/gpu.c \
           platform/nemu/ioe/audio.c \
           platform/nemu/ioe/disk.c \
           platform/nemu/mpe.c

CFLAGS    += -fdata-sections -ffunction-sections
LDFLAGS   += -T $(AM_HOME)/scripts/linker.ld \
             --defsym=_pmem_start=0x80000000 --defsym=_entry_offset=0x0
LDFLAGS   += --gc-sections -e _start
NEMUFLAGS += -l $(shell dirname $(IMAGE).elf)/nemu-log.txt
NEMUFLAGS += -b

CFLAGS += -DMAINARGS=\"$(mainargs)\"
CFLAGS += -I$(AM_HOME)/am/src/platform/nemu/include
.PHONY: $(AM_HOME)/am/src/platform/nemu/trm.c

image: $(IMAGE).elf
    @$(OBJDUMP) -d $(IMAGE).elf > $(IMAGE).txt
    @echo + OBJCOPY "->" $(IMAGE_REL).bin
    @$(OBJCOPY) -S --set-section-flags .bss=alloc,contents -O binary $(IMAGE).elf
$(IMAGE).bin

verilate-build:
    make -C $(NEMU_HOME) clean app

run: verilate-build image
    $(MAKE) -C $(NEMU_HOME) ISA=$(ISA) run ARGS="$(NEMUFLAGS)" IMG=$(IMAGE).bin

gdb: verilate-build image
    $(MAKE) -C $(NEMU_HOME) ISA=$(ISA) gdb ARGS="$(NEMUFLAGS)" IMG=$(IMAGE).bin
```

可见最终还是在 nemu 中进行运行。这里的脚本只是构建 nemu 运行时载入的镜像。

在 nemu 文件夹下使用 `make menuconfig` 指令配置 ISA 为 `riscv64`， 运行 dummy 测试，看到我们需要关心的文件列表：

```
+ CC src/nemu-main.c
+ CC src/engine/interpreter/hostcall.c
+ CC src/engine/interpreter/init.c
+ CC src/isa/riscv64/inst.c
+ CC src/isa/riscv64/system/mmu.c
+ CC src/isa/riscv64/system/intr.c
```

```
+ CC src/isa/riscv64/reg.c
+ CC src/isa/riscv64/difftest/dut.c
+ CC src/isa/riscv64/logo.c
+ CC src/isa/riscv64/init.c
+ CC src/device/io/map.c
+ CC src/device/io/port-io.c
+ CC src/device/io/mmio.c
+ CC src/cpu/difftest/dut.c
+ CC src/cpu/difftest/ref.c
+ CC src/cpu/cpu-exec.c
+ CC src/monitor/monitor.c
+ CC src/monitor/sdb/watchpoint.c
+ CC src/monitor/sdb/sdb.c
+ CC src/monitor/sdb/expr.c
+ CC src/utils/timer.c
+ CC src/utils/state.c
+ CC src/utils/log.c
+ CC src/utils/rand.c
+ CC src/memory/vaddr.c
+ CC src/memory/paddr.c
+ CXX src/utils/disasm.cc
+ LD /home/noionion/MyPersonProject/riscv_cpudesigner/ics2022/nemu/build/riscv64-nemu-interpreter
```

查看 nemu 的调用过程：

```
nemu-main
    init_monitor // 初始化
        init_mem
            分配空间，定义起始位置
        init_isa
            // src/isa/riscv64/init.c
            // 根据上面make过程，可以确认是指向上诉唯一的init函数
        load_img
            // 这个image指的是测试程序，不是isa镜像
        init_sdb
            init_regex
            init_wp_pool

    engine_start // sdb 主循环
        sdb_mainloop
            'c': cmd_c
                // 注意以下的-1都是uint64_t类型，实际表示一个极大数
                cpu_exec(-1)
                    execute(-1)
                        // TODO: 以下应该就是嵌入npc要修改部分
                        exec_once
                            isa_exec_once
                                decode_exec
```

而运行 dummy 后进入 nemu，键入 `c` 运行程序后发现，decode_exec 函数是 PA1 中需要完成我们补全指令来通过 dummy.c 程序。不妨先试着完成一下 PA1 的 RTFM - 运行第一个C程序 部分。

根据提示我们逐步补全三个指令：`addi`、`jal`、`jalr`，提示如下：

```
// addi
```

```
(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.

// jal

(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.

// jalr

(nemu) c
invalid opcode(PC = 0x0000000080000014):
        67 80 00 00 13 01 01 ff ...
        00008067 ff010113...
There are two cases which will trigger this unexpected exception:
1. The instruction at PC = 0x0000000080000014 is not implemented.
2. Something is implemented incorrectly.
Find this PC(0x0000000080000014) in the disassembling result to distinguish which
case it is.
```

期间注意自己需要完成 J-type 指令的立即数组装，以及 dnpc 与 snpc 的区别。修改 nemu/src/isa/riscv64/inst.c 部分代码如下：

```
  // ...

  enum {
    TYPE_I, TYPE_U, TYPE_S,
    TYPE_N, // none
+   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 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)

  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;
    int rs1 = BITS(i, 19, 15);
    int rs2 = BITS(i, 24, 20);
    *rd     = BITS(i, 11, 7);
    switch (type) {
      case TYPE_I: src1R();          immI(); break;
      case TYPE_U:                   immU(); break;
      case TYPE_S: src1R(); src2R(); immS(); break;
+     case TYPE_J:                   immJ(); break;
    }
  }

  // ...

    INSTPAT_START();
    INSTPAT("??????? ????? ????? ??? ????? 00101 11", auipc  , U, R(rd) = s->pc +
imm);
    INSTPAT("??????? ????? ????? 011 ????? 00000 11", ld     , I, R(rd) = Mr(src1 +
imm, 8));
    INSTPAT("??????? ????? ????? 011 ????? 01000 11", sd     , S, Mw(src1 + imm, 8,
src2));

    INSTPAT("0000000 00001 00000 000 00000 11100 11", ebreak , N, NEMUTRAP(s->pc,
R(10))); // R(10) is $a0
+   INSTPAT("??????? ????? ????? 000 ????? 00100 11", addi   , I, R(rd) = src1 +
imm);
+   INSTPAT("??????? ????? ????? ??? ????? 11011 11", jal    , J, R(rd) = s->pc +
4, s->dnpc = s->pc + imm);
+   INSTPAT("??????? ????? ????? 000 ????? 11001 11", jalr   , I, R(rd) = s->pc +
4, s->dnpc = (src1 + imm) & ~1);

    INSTPAT("??????? ????? ????? ??? ????? ????? ??", inv    , N, INV(s->pc));

    INSTPAT_END();
```

完成后测试，如若成功则输出的提示如下：

```
(nemu) c
[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 = 542 us
[src/cpu/cpu-exec.c:89 statistic] total guest instructions = 13
[src/cpu/cpu-exec.c:90 statistic] simulation frequency = 23,985 inst/s
(nemu) q
 dummy
[         dummy] PASS!
```

跳过每次输入 c 才能运行程序：通过批处理模式运行NEMU

通过源码：`nemu/src/monitor/monitor.c` 可知，在NEMU的启动参数中加入`-b`参数可以令其为批处理模式，修改makefile文件加入参数：

```
# abstract-machine/scripts/platform/nemu.mk

NEMUFLAGS += -b
```

2.2 add riscv64_npc

> “一生一芯”讲义中，并不是要我们以 nemu 作为自己设计的 cpu 测试用的虚拟环境。为了方便，魔改 nemu 的底层来进行测试会更加合理。

根据 支持RV64IM的单周期NPC 和 `nemu/src/isa/filelist.mk` 中的说明：

```
INC_PATH += $(NEMU_HOME)/src/isa/$(GUEST_ISA)/include
DIRS-y += src/isa/$(GUEST_ISA)
```

可以在 `nemu/src/isa/` 下新建 `riscv64-npc` 来指定新的路径，并构建相应的 defftest 和 DPI-C 接口来完成目标。

为了适配该路径，我们需要修改 `nemu/Kconfig`，写入 riscv64-npc 这一选项（因为不使用spike作为REF，故这里不管下面两个选项）：

```
  mainmenu "NEMU Configuration Menu"

  choice
    prompt "Base ISA"
    default ISA_riscv32
  config ISA_x86
    bool "x86"
  config ISA_mips32
    bool "mips32"
  config ISA_riscv32
    bool "riscv32"
  config ISA_riscv64
    bool "riscv64"
  config ISA_loongarch32r
    bool "loongarch32r"
+ config ISA_riscv64_npc
+   bool "riscv64-npc"
  endchoice

  config ISA
    string
    default "x86" if ISA_x86
    default "mips32" if ISA_mips32
    default "riscv32" if ISA_riscv32
    default "riscv64" if ISA_riscv64
    default "loongarch32r" if ISA_loongarch32r
+   default "riscv64-npc" if ISA_riscv64_npc
    default "none"

  config ISA64
-   depends on ISA_riscv64
+   depends on ISA_riscv64 || ISA_riscv64_npc
    bool
    default y
```

并为difftest程序中各个宏定义部分添加相应配置：

```
nemu/include/difftest-def.h
+ #elif defined(CONFIG_ISA_riscv64_npc)
+ # define DIFFTEST_REG_SIZE (sizeof(uint64_t) * 33) // GPRs + pc
```

```
nemu/src/monitor/monitor.c
-     MUXDEF(CONFIG_ISA_riscv64, "riscv64", "bad")))) "-pc-linux-gnu"
+     MUXDEF(CONFIG_ISA_riscv64, "riscv64",
+     MUXDEF(CONFIG_ISA_riscv64_npc, "riscv64", "bad"))))) "-pc-linux-gnu"

nemu/tools/qemu-diff/include/isa.h
+ #elif defined(CONFIG_ISA_riscv64_npc)
+ #define ISA_QEMU_BIN "qemu-system-riscv64"
+ #define ISA_QEMU_ARGS
  // ...
+ #elif defined(CONFIG_ISA_riscv64_npc)
+     uint64_t gpr[32];
+     uint64_t fpr[32];
+     uint64_t pc;
```

接下来就可以开始整个`riscv64-npc`文件夹了。新建`nemu/src/isa/riscv64_npc`，将`riscv64`文件夹中的目录文件复制到`riscv64_npc`中，修改`nemu/src/isa/riscv64_npc/include/isa-def.h`为

```
#ifndef __ISA_RISCV64_NPC_H__
#define __ISA_RISCV64_NPC_H__

#include <common.h>

typedef struct {
  word_t gpr[32];
  vaddr_t pc;
} riscv64_npc_CPU_state;

// decode
typedef struct {
  union {
    uint32_t val;
  } inst;
} riscv64_npc_ISADecodeInfo;

#define isa_mmu_check(vaddr, len, type) (MMU_DIRECT)

#endif
```

重新在 nemu 文件夹下`make menuconfig`，选用 ISA 为 riscv64-npc 后测试dummy样例能否正常运行。

2.3 让nemu引用verilog

之前说到，verilator 会将 verilog 代码编译成静态库。在`riscv64_npc`新建文件夹`verilated`，将 cpp 文件置于其目录下，在其中新建`vsrc`文件夹存放我们的 verilog 文件。现在`riscv64_npc`文件夹结构如下：

```
.
├── difftest
│   └── dut.c
├── include
│   ├── isa-def.h
│   └── verilated_cpu.h
├── init.c
├── inst.c
├── local-include
```

```
│   └── reg.h
├── logo.c
├── reg.c
├── system
│   ├── intr.c
│   └── mmu.c
└── verilated
    ├── verilated_cpu.cpp
    └── vsrc
        └── cpu.v
```

在 cpu.v 中写入如下内容：

```
module cpu {
    input clk
};
    always @(*) begin
        $display("%d", clk);
    end
endmodule;
```

在 verilated_cpu.cpp 中写入如下内容：

```
#include "./vsrc/obj_dir/Vcpu.h"
#include <cstdio>
#include <assert.h>
#include <fstream>
#include <iostream>
using namespace std;

Vcpu* vcpu = NULL;
VerilatedContext* vcontext = NULL;

extern "C" {
    void vcpu_init() {
        if (vcpu) delete vcpu;
        if (vcontext) delete vcontext;

        vcontext = new VerilatedContext();
        vcpu = new Vcpu{vcontext};
        vcpu->clk = 0;
        vcpu->eval();
    }

    void vcpu_do_cycle() {
        vcpu->clk = 1;
        vcpu->eval();
        vcpu->clk = 0;
        vcpu->eval();
    }
}
```

由于 nemu 中主要以 c 进行编程，同一函数名在 gcc 与 g++ 编译产生的符号并不相同，故需要加上 `extern "C"` 才能让 c 代码能调用 c++ 函数

在 verilated_cpu.h 加上函数定义：

```
#ifdef _cplusplus
extern "C"{
#endif

void vcpu_init();
void vcpu_do_cycle();

#ifdef _cplusplus
}
#endif
```

现在在 verilated/vsrc 文件中使用 verilator 进行编译，会得到一系列 .h .o .a 等文件。在 inst.c 文件中调用它，改写整个文件：

```
#include "local-include/reg.h"
#include <cpu/cpu.h>
#include <cpu/ifetch.h>
#include <cpu/decode.h>
#include <verilated_cpu.h>
#include <utils.h>

int isa_exec_once(Decode *s) {
  s->isa.inst.val = 0; // 这里要等pc接口实现
  vcpu_do_cycle();
  s->dnpc = 0; // 同上
  return 0;
}
```

再次测试 dummy 文件夹，发现我们的 verilated/obj_dir 文件夹中的内容并未编译进目标中。

查找编译的 makefile 文件：`nemu/scripts/build.mk`，我们将整个 verilater 编译加入编译目标中：

```
// ...

# Modified to adapt verilated cpu

VERILATED_DIR = $(NEMU_HOME)/src/isa/riscv64_npc/verilated
VSRC_DIR = $(VERILATED_DIR)/vsrc
VERILATED_OBJDIR = $(VSRC_DIR)/obj_dir

VERILATED_SRCS = $(addprefix src/isa/riscv64_npc/verilated/, $(notdir $(shell find
$(VERILATED_DIR) -maxdepth 1 -name "*.cpp")))

.IGNORE: verilate
verilate: $(VSRCS)
    rm -rf $(VERILATED_OBJDIR)
    cd $(VSRC_DIR) && verilator --cc --build --timing --exe cpu.v

OBJS += $(wildcard $(VERILATED_OBJDIR)/*.o)
OBJS += $(VERILATED_SRCS:%.cpp=$(OBJ_DIR)/%.o)

$(OBJ_DIR)/%.o: %.cpp
    @echo + CXX $<
    @mkdir -p $(dir $@)
    @$(CXX) $(CFLAGS) $(CXXFLAGS) -faligned-new -c -o $@ $<
```

```
    $(call call_fixdep, $(@:.o=.d), $@)

# Some convenient rules

.PHONY: app clean

app: verilate $(BINARY)

$(BINARY): $(OBJS) $(ARCHIVES)
    @echo + LD $@
    @$(LD) -o $@ $(OBJS) $(LDFLAGS) $(ARCHIVES) $(LIBS)

clean:
    rm -rf $(VERILATED_OBJDIR)
    rm -rf $(BUILD_DIR)
```

这里需要 `--exe` 参数是为了将 verilator 中的一些源码头文件编译到 obj_dir 文件夹中，尽管它会出现如

```
/usr/bin/ld: /usr/lib/gcc/x86_64-pc-linux-
gnu/13.1.1/../../../../lib/Scrt1.o: in function `_start':
(.text+0x1b): undefined reference to `main'
collect2: 错误: ld 返回 1
make[4]: *** [Vcpu.mk:61: Vcpu] 错误 1
```

这样的报错，告诉我们 verilator 的链接目标没有 main 函数，没关系，反正我们只要它的 verilated.o 等文件

再次编译，我们可以看到编译完成，nemu 连续输出 01 串。现在我们已经能把 verilog 代码添加入 nemu 并运行它。

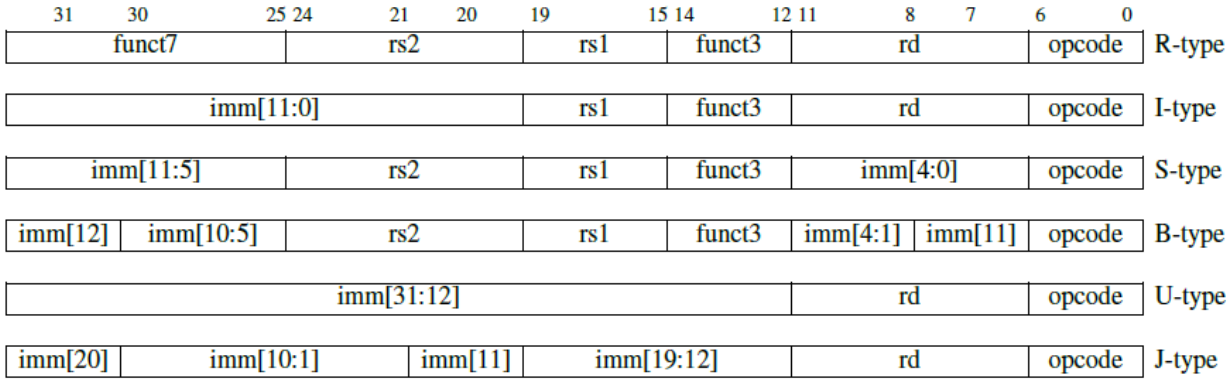
3 riscv64 单周期cpu 设计

3.1 riscv64指令解析

3.1.1 指令格式和指令表

这里需要完成的是 riscv64 指令集的 IM 部分，查找 riscv 的英文文档和中科大的中文说明（部分中文内容翻译有误）。可以知道 riscv64 的指令类型如下：

- 用于寄存器-寄存器操作的 R 类型指令
- 用于短立即数和访存 load 操作的 I 型指令
- 用于访存 store 操作的 S 型指令
- 用于条件跳转操作的 B 类型指令
- 用于长立即数的 U 型指令
- 用于无条件跳转的 J 型指令



riscv 架构很主要的特点：指令中固定提供三个寄存器操作数，要读写的寄存器的标识符总是在同一位置，意味着在解码指令之前，就可以先开始访问寄存器，而不必像 x86 需先解析操作；符号位总是在指令中最高位。这意味着可能成为关键路径的立即数符号扩展，可以在指令解码之前进行。

RV32I 有 31 寄存器加上一个值恒为 0 的 x0 寄存器：

31 … 0	
x0 / zero	Hardwired zero
x1 / ra	Return address
x2 / sp	Stack pointer
x3 / gp	Global pointer
x4 / tp	Thread pointer
x5 / t0	Temporary
x6 / t1	Temporary
x7 / t2	Temporary
x8 / s0 / fp	Saved register, frame pointer
x9 / s1	Saved register
x10 / a0	Function argument, return value
x11 / a1	Function argument, return value
x12 / a2	Function argument
x13 / a3	Function argument
x14 / a4	Function argument
x15 / a5	Function argument
x16 / a6	Function argument
x17 / a7	Function argument
x18 / s2	Saved register
x19 / s3	Saved register
x20 / s4	Saved register
x21 / s5	Saved register
x22 / s6	Saved register
x23 / s7	Saved register
x24 / s8	Saved register
x25 / s9	Saved register
x26 / s10	Saved register
x27 / s11	Saved register
x28 / t3	Temporary
x29 / t4	Temporary
x30 / t5	Temporary
x31 / t6	Temporary

32

31 … 0
pc

32

而我们需要完成的指令集如下，

RV I 指令为最基础的指令集：

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	Rsltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and
0000 pred	succ	00000	000	00000	0001111	I fence
0000 0000	0000	00000	001	00000	0001111	I fence.i
000000000000		00000	00	00000	1110011	I ecall
000000000000		00000	000	00000	1110011	I ebreak
csr		rs1	001	rd	1110011	I csrrw
csr		rs1	010	rd	1110011	I csrrs
csr		rs1	011	rd	1110011	I csrrc
csr		zimm	101	rd	1110011	I csrrwi
csr		zimm	110	rd	1110011	I cssrrsi
csr		zimm	111	rd	1110011	I csrrci

不需要完成 fence、ecall 和 csr 指令

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]			rs1	110	rd	0000011	I lwu
imm[11:0]			rs1	011	rd	0000011	I ld
imm[11:5]		rs2	rs1	011	imm[4:0]	0100011	S sd
0000000		shamt	rs1	001	rd	0010011	I slli
0000000		shamt	rs1	101	rd	0010011	I srli
0100000		shamt	rs1	101	rd	0010011	I srai
imm[11:0]			rs1	000	rd	0011011	I addiw
0000000		shamt	rs1	001	rd	0011011	I slliw
0000000		shamt	rs1	101	rd	0011011	I srliw
0100000		shamt	rs1	101	rd	0011011	I sraiw
0000000		rs2	rs1	000	rd	0111011	R addw
0100000		rs2	rs1	000	rd	0111011	R subw
0000000		rs2	rs1	001	rd	0111011	R sllw
0000000		rs2	rs1	101	rd	0111011	R srlw
0100000		rs2	rs1	101	rd	0111011	R sraw

RV M 扩展则是乘除法相关：

31 25	24 20	19 15	14 12	11 7	6 0	
0000001	rs2	rs1	000	rd	0110011	R mul
0000001	rs2	rs1	001	rd	0110011	R mulh
0000001	rs2	rs1	010	rd	0110011	R mulhsu
0000001	rs2	rs1	011	rd	0110011	R mulhu
0000001	rs2	rs1	100	rd	0110011	R div
0000001	rs2	rs1	101	rd	0110011	R divu
0000001	rs2	rs1	110	rd	0110011	R rem
0000001	rs2	rs1	111	rd	0110011	R remu

RV64M Satndard Extension (in addition to RV32M)

0000001	rs2	rs1	000	rd	0111011	R mulw
0000001	rs2	rs1	100	rd	0111011	R divw
0000001	rs2	rs1	101	rd	0111011	R divuw
0000001	rs2	rs1	110	rd	0111011	R remw
0000001	rs2	rs1	111	rd	0111011	R remuw

3.1.2 指令特点归纳

首先从 opcode 入手，可以将指令区分为以下几种：

- R 类型指令有：64位的 0110011（RV32I、RV32M） 和32 位宽的 0111011（RV64I、RV64M），可以全部归类为 ALU 指令
- I 类型有：
  - 64位的 0010011（RV32I、RV32M） 和32 位宽的 0011011（RV64I、RV64M），可以归类为 ALU 指令
  - 内存读取指令 0000011，归类为 MEM 指令
  - ecall 和 ebreak 指令 1110011，归类为 ENV 指令
  - jalr 跳转指令 1100111，归类到 JUMP 指令
- S 类型有：内存写入指令 0100011，可以归类为 MEM 指令
- B 类型有：分支逻辑跳转指令 1100011，可以归类为 JUMP 指令
- U 类型有：AUIPC 0010111 和 LUI 0110111，可以归类为 Uimm 操作指令

- J 类型有：JAL 跳转指令 1101111,可以归类为 JUMP 指令

按 opcode 作为分类 12 位的控制线，则可以进一步分类各个模块组的控制线：

```
assign alu_en           = R_alu64   | R_alu32   | I_alu64   | I_alu32;
assign mem_en           = I_memload | S_memstore;
assign env_en           = I_env;
assign logic_jump_en    = I_jalr    | B_branch  | J_jal;
assign uimm_op_en       = U_auipc   | U_lui;
```

以及 imm 的选择控制：

```
wire I = I_alu64 | I_alu32 | I_memload | I_env | I_jalr;
wire S = S_memstore;
wire B = B_branch;
wire U = U_auipc | U_lui;
wire J = J_jal;
```

然后从 funct3 进行更具体的分类

> ALU 部分的分类较为复杂，需要 funct7 一起判断，最后再进行特点分析

对于内存读写模块，读写操作已经由操作数分开。而 riscv 的特点是最高位为符号表示位（即 `funct3[2]`），可以发现无论读写，都有以下特点：

funct3[1:0]	读写长度
00	8
01	16
10	32
11	64

对于跳转模块，可以发现分支跳转的 funct3 各个位有如下表示（这里的最高位就不是符号位了：

funct3[2]	funct3[1]	funct3[0]
等价关系运算/顺序关系运算	有符号/无符号	是否结果取反

故而本要有6次的判断，在总结之后我们只需要3次的判断即可：

原判断：

```
wire equal = (bus1 == bus2);
wire sless = ($signed(bus1) < $signed(bus2));
wire uless = ($unsigned(bus1) < $unsigned(bus2));

wire jump_en = (funct3 == 3'b000) ?  equal :
               (funct3 == 3'b001) ? ~equal :
               (funct3 == 3'b100) ?  sless :
               (funct3 == 3'b101) ? ~sless :
               (funct3 == 3'b001) ?  uless :
               (funct3 == 3'b001) ? ~uless ;
```

总结后：

```
wire equal = (bus1 == bus2);
wire sless = ($signed(bus1) < $signed(bus2));
wire uless = ($unsigned(bus1) < $unsigned(bus2));

wire less       = (funct3[1])   ? uless         : sless     ;
wire result_tmp = (funct3[2])   ? less          : equal     ;
wire result     = (funct3[0])   ? ~result_tmp   : result_tmp;
```

对于 imm，只有 ENV 指令使用其进行分类。

最后是较为复杂的 ALU 部分，其分类我们采用 R type 和 I type、32 和 64、funct3、funct7 四部分进行分类：

	0110011		**0111011**		**0010011**	**0011011**
funct7[0] →		1		1		
funct3 ↓			32			32
000	add/sub	mul	addw/subw	mulw		
001	sll	mulh	sllw		slli	slliw
010	slt	mulhsu				
011	sltu	mulhu				
100	xor	div		divw		
101	srl/sra	divu	srlw/sraw	divuw	srli/srai	
110	or	rem		remw		
111	and	remu		remuw		

根据 riscv 的指令说明，立即数只需要在开始时对第二个操作数根据 I/R 类型进行选择，而 64/32 位指令只需计算的开始对两个操作数进行截断扩展、计算结尾进行截断扩展。这里主要需要分辨的应该是 18 种指令。

注意到 funct7[0] 可以对 RV I 和 RV M 指令进行初步的区分，我们将 ALU 分为两部分

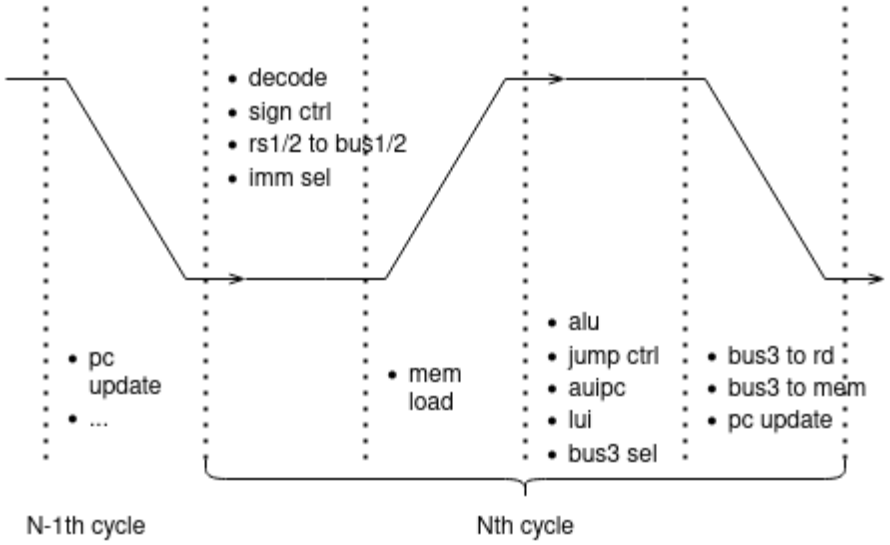
接着根据 funct3 ，可以各自作区分：

funct3	**rv i**	**rv m**
000	add/sub	mul
001	sll	mulh
010	slt	mulhsu
011	sltu	mulhu
100	xor	div
101	srl/sra	divu
110	or	rem
111	and	remu

最后是区分 add/sub 和 srl/sra，观察可发现 funct7[5] 恰好可以区分这一位（注意由于没有 subi，故这里 imm [5] 与 funct7[5] 不等效；而 srli/srai 则能以 imm[5] 区分，与 funct7[5] 等效）

3.2 架构设计

3.2.1 单周期行为预期



在不考虑乘除法周期的情况下，单周期内可以使用上图逻辑：

1. 在上一周期的下降沿时，pc 更新，此时我们可以在地址更新后立即取指令（由于 c 语言模拟的特殊性可以不考虑实际内存读写需要时钟）
2. 取指令过后对指令的拆分译码、读取寄存器两个源操作数我们均可以以组合逻辑完成（实际情况寄存器其实也应该在时钟上下跳变沿进行读写，这里为了保证内存读取前能让 bus1 得到寄存器中的操作数，只能这么模拟）
3. 此时一般而言，alu、跳转控制等等组合逻辑电路也应该会在此过程完成，但考虑到中途总线数据、pc值可能发生变化，而最终的结果会在下降沿才会写入寄存器，所以我们在这里可以直接认为它在下降沿前运行即可
4. 上升沿前总线控制会根据控制线分配总线 bus1、bus2 和 imm 的输入
5. 上升沿时，bus1 和 imm 已经稳定，若涉及内存读取操作则此时将读取内存到 bus3 上
6. 下降沿前，所有总线稳定，计算稳定
7. 下降沿将 bus3 结果按写权限写寄存器、内存和更新 pc

如若考虑实际情况，设计应使用多周期

1. 在上一周期的下降沿，更新pc
2. 第一个上升沿，从内存中取出指令
3. 第一个下降沿前，组合逻辑完成指令拆分、译码控制等操作
4. 第一个下降沿时，读取寄存器到总线 bus1 和 bus2，imm 此时应已稳定
5. 第二个上升沿前，bus1 稳定
6. 第二个上升沿时，若涉及内存读取则取内存
7. 第二个下降沿前，若不考虑实际情况下的长周期运算则运算稳定
8. 第二个下降沿时，根据写权限写寄存器、内存和更新 pc

所以单线至少需要**两个时钟周期**才能在实际情况下完成正常的单指令周期

网上一些设计说不采用和计组书上写的一样的前半个收起写 reg，后半个周期读 reg，是因为需要用下降沿将一个时钟周期分开，那么 ID 阶段的时延就不够用了，无法提高到很高的频率，然后将一条指令分配到了 5 个时钟周期中。但这里认为 ID 阶段的时延其实只不过 4 次门操作的时延，实际情况远小于读写内存带来的时延消耗，将其抬高频率也不会降低单条指令总时延

更合理的解释应该是，高频率多周期来处理一条指令，每个周期干的事越短，多流水线的覆盖率就越高，经过几个周期的启动后也能达到一周期一指令的速度，此时短周期的收益就特别高

不过具体的流水线控制可能还有一些其他的情况，比如长周期运算等需要中断其他流水线，可以参考 https://github.com/Yawei-Ding/ysyx_riscv64_cpu

目前我们需要的完成度并不用这么高，时间上也没那么充裕，所以流水线啥的就以后有空再来吧

3.2.2 整体设计

整体将根据上述单周期预期行为进行建模，主要区分为三个部分

1. 取指令、指令拆分、译码控制三模块作为第一部分，产生各个具体指令运算模块的控制信号
2. ENV、ALU、MEM、JUMP、UIMM_OP 五个具体功能分类模块作为第二部分，对指令进行具体的 cpu 状态控制和数学逻辑运算。为了编程方便，ALU 根据 RV I/RV M 分为 ALU_I 和 ALU_M，JUMP 根据指令类型分为 JAL、BRANCH、JAL，UIMM_OP 根据指令分为 AUIPC 和 LUI
3. 总线控制部分，这里有四条总线，记为用于输入的 bus1、bus2、imm 和用于输出的 bus3（为了编程方便，imm 的控制归于第一部分），以及寄存器读写控制

文件列表如下：

```
├── alu.v
├── const.v
├── cpu.v
├── dpidef.v
├── env.v
├── fetch.v
├── gprs.v
├── inst_split.v
├── logic_jump.v
├── memory.v
├── sign_imm_ctrl.v
└── uimm_op.v
```

整体管理：

- `const.v` 用于标记例如 opcode 等的常量标记（详见 指令特点归纳）
- `dpidef.v` 用于管理与 c 代码交互的函数（verilator 的 DPI-C 机制）
- `cpu.v` 连接其他部分，并包含第三部分的总线管理（bus1、bus2、bus3）和寄存器读写控制，和跳转指令的控制

第一部分：

- `fetch.v` 用于 pc 更新和取指令
- `inst_split.v` 对 fetch 得到的指令进行切割，得到 opcode、funct3 以及 R 类型指令需要的 funct7 和其他 5 种类型指令需要的 5 种不同长度、不同组装方式的立即数
- `sign_imm_ctrl.v` 对 opcode 进行判断，区分 12 种类型指令信号和 5 种运算控制模块的控制信号，并分配 imm 总线

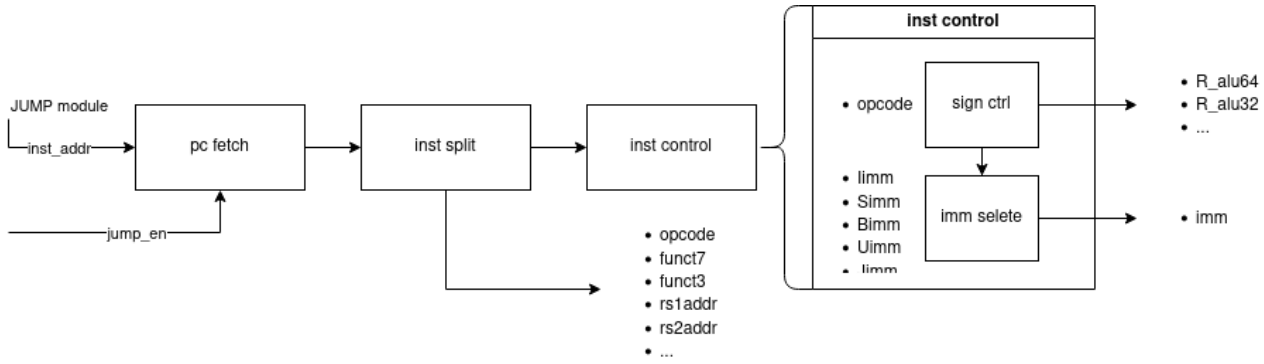
第二部分：

- `env.v` 包含 ecall 和 ebreak 指令的处理，含 module ENV 单个模块
- `alu.v` 包含计算类的指令，含 module ALU 和 module ALU_M_ext，其中 cpu 连线仅需连接 module ALU（内部已实例化 module ALU_M_ext）
- `memory.v` 包含内存读写指令，含 module Memory 单个模块
- `logic_jump.v` 包含三种 pc 跳转指令，按类型区分为 module JALR、module Branch 和 module JAL 三个模块
- `Uimm_op.v` 包含两种 U 类型指令，按指令区分为 module AUIPC 和 module LUI 两个模块

为了对接 nemu 的指令实现检测，该部分的每个模块都具有一条名为 debug_inst_analy 的线路，汇总至 `fetch.v` 中进行判断。在下降沿更新指令时，会检测这一周期的指令是否有被实现（即经过第二部分其中的某一个模块）。若 debug_inst_analy 为 0 则说明指令未被实现，跳入 nemu 提供的 INV(pc) 指令接口来中断测试程序并报错。

3.3 模块设计

3.3.1 取指令、指令拆分、译码控制

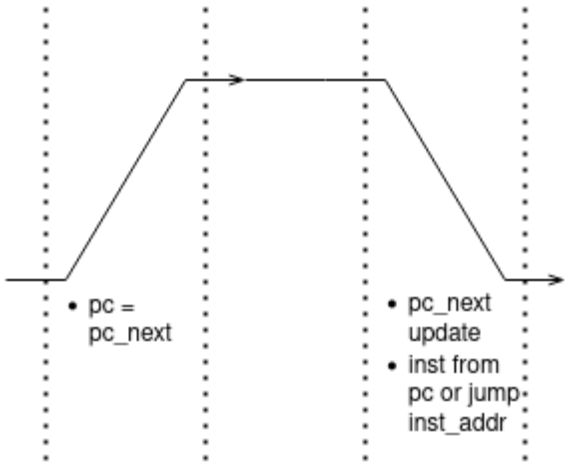


在 cpu IF 阶段，我们主要区分为 pc 自更新和取指令。常规而言，在周期结束的时钟下降沿，pc 进行更新，在多周期 cpu 下第一个周期再进行取指令，这样会有一个良好的先后区分。

然而目前我们设计的单周期 cpu 这个时间却是不确定的，我们无法保证下降沿到上升沿期间取指令会在什么时候执行（也许在 pc 还没更新就取完指令了），因此 pc 的更新必须提前。

初步的设计是，我们定义两个 pc 状态寄存器 pc_reg 和 pc_next_reg ，pc_next_reg 在下降沿更新，而 pc_reg 在上升沿更新为 pc_next_reg。这样在下降沿以 pc_reg 为地址取指令时我们便能保证 pc_reg 取出的是下一条指令。

然而这又带来了另一个问题，在跳转指令中 pc 的新地址是在上升沿到下降沿这一时间才被确定，而 pc_reg 已经更新为pc_next_reg （即 pc_reg + 4），而正确的 pc_next_reg 在下降沿才更新，此时取指令就发生了错误。故而在这种情况下应该以传来的地址作为取指令依据，而非 pc_reg （听起来有些不合常理，但是逻辑上才能保证取指令正确），而此时 pc_next_reg 应更新为传入的地址 +4。




```
module Fetch (
    input clk,
    input jump_en,
    input [63:0] inst_addr,
    output reg [31:0] inst,
    output reg [63:0] pc_reg
);
    reg [63:0] pc_next_reg;

    initial begin
        pc_reg = PC_RESET;
        pc_next_reg = PC_RESET;
    end
```

```
    always @(pc_next_reg) begin
        pc_reg <= pc_next_reg;
    end

    always @(negedge clk) begin
        if (jump_en) begin
            pc_next_reg <= inst_addr + 4;
            inst <= readword(inst_addr);
        end
        else begin
            pc_next_reg <= pc_reg + 4;
            inst <= readword(pc_reg);
        end
    end
endmodule;
```

取出指令后，我们可以根据六种指令格式对指令进行初步分割（这里无需判断，无非是使用与否）

```
    assign opcode = inst[6:0];
    assign funct3 = inst[14:12];
    assign funct7 = inst[31:25];
    assign rs1addr = inst[19:15];
    assign rs2addr = inst[24:20];
    assign rdaddr = inst[11:7];
    assign Iimm = inst[31:20];
    assign Simm = {inst[31:25], inst[11:7]};
    assign Bimm = {inst[31], inst[7], inst[30:25], inst[11:8], 1'b0};
    assign Uimm = {inst[31:12], 12'b0};
    assign Jimm = {inst[31], inst[19:12], inst[20], inst[30:21], 1'b0};
```

此后进入信号控制模块，在这一模块我们确认接下来应该是哪一个模块进入工作：

首先进行信号的区分：

```
    assign R_alu64          = (opcode == R_ALU64);
    assign R_alu32          = (opcode == R_ALU32);
    assign I_alu64          = (opcode == I_ALU64);
    assign I_alu32          = (opcode == I_ALU32);
    assign I_memload        = (opcode == I_MEMLOAD);
    assign I_env            = (opcode == I_ENV);
    assign I_jalr           = (opcode == I_JALR);
    assign S_memstore       = (opcode == S_MEMSTORE);
    assign B_branch         = (opcode == B_BRANCH);
    assign U_auipc          = (opcode == U_AUIPC);
    assign U_lui            = (opcode == U_LUI);
    assign J_jal            = (opcode == J_JAL);
```

上述诸如 R_ALU64 为常量

而后判断是哪些部件进行工作：

```
    assign alu_en           = R_alu64   | R_alu32   | I_alu64   | I_alu32;
    assign mem_en           = I_memload | S_memstore;
    assign env_en           = I_env;
    assign logic_jump_en    = I_jalr    | B_branch  | J_jal;
    assign uimm_op_en       = U_auipc   | U_lui;
```

为了方便，立即数选择的总线控制我也在这一部分实现：

```
    wire I = I_alu64 | I_alu32 | I_memload | I_env | I_jalr;
    wire S = S_memstore;
    wire B = B_branch;
    wire U = U_auipc | U_lui;
    wire J = J_jal;

    wire [4:0] type_sel = {I, S, B, U, J};

    parameter caseI = 5'b10000;
    parameter caseS = 5'b01000;
    parameter caseB = 5'b00100;
    parameter caseU = 5'b00010;
    parameter caseJ = 5'b00001;

    always @(*) begin
        case (type_sel)
            caseI: imm = {{(52){Iimm[11]}}, Iimm};
            caseS: imm = {{(52){Simm[11]}}, Simm};
            caseB: imm = {{(51){Bimm[12]}}, Bimm};
            caseU: imm = {{(32){Uimm[31]}}, Uimm};
            caseJ: imm = {{(43){Jimm[20]}}, Jimm};
            default: imm = 64'b0;
        endcase
    end
```

之后指令将被区分到 5 个部分进行运算等操作。

3.3.2 运算模块与存储模块

GPRS 通用寄存器组：

如前文所说，需要在上升沿前就完成 bus1 和 bus2 的数据输出，故寄存器在上升沿前就需要以组合逻辑直接输出。写入我们仍旧在下降沿进行。

```
module Gprs (
    input clk,
    input [4:0] rs1addr, rs2addr, rdaddr,
    input [2:0] RRW,
    input [63:0] rd,
    output [63:0] rs1, rs2,
    output [63:0] debug_gprs[32]
);
    reg [63:0] gprs[32];
    initial begin
        gprs[0] = 64'b0;
    end

    // The register values need to be fetched before memory load (posedge clk).
    assign rs1 = (RRW[2]) ? gprs[rs1addr] : 0;
    assign rs2 = (RRW[1]) ? gprs[rs2addr] : 0;

    always @(negedge clk) begin
        if (RRW[0] && rdaddr != 0) begin
            gprs[rdaddr] <= rd;
        end
```

```
    end
endmodule;
```

需要注意 0 寄存器只读。

ENV 环境调用和环境断点指令：

这一部分暂时无法从硬件角度实现，它涉及异常跳转等。但是 nemu 实现了一条特殊的 `nemutrap` 指令, 用于指示客户程序的结束，具体地，在 RISC-V 中，nemu 选择了 `ebreak` 指令来作为 `nemutrap` 指令。故这里实际上是实现 ebreak 的 DPI-C 接口，DPI-C 接口如下：

```
#include <cpu/cpu.h>
void ebreak(uint64_t addr, uint64_t gpr10) {NEMUTRAP(addr, gpr10); }
```

而 ENV 模块只需判断 ebreak 并调用即可。

Memory 内存读写模块：

这里使用常规的上升沿读取下降沿写入的涉及即可，DPI-C 接口设计如下：

```
#define CREATE_DPI(type, name) \
    type read##name(uint64_t addr)              {return vaddr_read(addr,
sizeof(type)); } \
    void write##name(uint64_t addr, type value) {vaddr_write(addr, sizeof(type),
value); }

#include <memory/vaddr.h>
CREATE_DPI(uint8_t, byte)
CREATE_DPI(uint16_t, half)
CREATE_DPI(uint32_t, word)
CREATE_DPI(uint64_t, dword)
```

模块设计如下：

```
module Memory (
    input clk, read_en, write_en,
    input [2:0] funct3,
    input [63:0] bus1, bus2, imm,
    output [63:0] mem_out
);
    reg [63:0] mem_read;

    wire [63:0] addr = bus1 + imm;

    always @(posedge clk) begin
        if (~read_en) begin
            mem_read = 64'b0;
        end
        else begin
            case (funct3[1:0])
                MEMFLAG_TYPE_BYTE   : mem_read = {56'b0 , readbyte(addr)};
                MEMFLAG_TYPE_HALF   : mem_read = {48'b0 , readhalf(addr)};
                MEMFLAG_TYPE_WORD   : mem_read = {32'b0 , readword(addr)};
                MEMFLAG_TYPE_DWORD  : mem_read = readdword(addr);
            endcase
        end
    end
```

```
    always @(negedge clk) begin
        if (~write_en) begin
        end
        else begin
            case (funct3[1:0])
                MEMFLAG_TYPE_BYTE   : writebyte(addr, bus2[7:0]);
                MEMFLAG_TYPE_HALF   : writehalf(addr, bus2[15:0]);
                MEMFLAG_TYPE_WORD   : writeword(addr, bus2[31:0]);
                MEMFLAG_TYPE_DWORD  : writedword(addr, bus2);
            endcase
        end
    end

    assign mem_out = (funct3[MEMINDEX_UNSIGNED])        ? mem_read :
                     (funct3[1:0] == MEMFLAG_TYPE_BYTE) ? {{(56){mem_read[7]}},
mem_read[7:0]}   :
                     (funct3[1:0] == MEMFLAG_TYPE_HALF) ? {{(48){mem_read[15]}},
mem_read[15:0]} :
                     (funct3[1:0] == MEMFLAG_TYPE_WORD) ? {{(32){mem_read[31]}},
mem_read[31:0]} :
                                                          mem_read
                ;
endmodule;
```

由于读取还区分有无符号，故需要在输出前进行截断扩展。

JUMP 跳转模块：

跳转模块具有三种类型的指令，其中 jalr 和 jal 都是无条件跳转，B 类型指令则需要条件判断，在前面指令特点归纳时也已经说明其规律。

故输出到 pc 的控制线 jump_en 可如下设置：

```
    wire jump_en = J_jal | I_jalr | B_jump_en;
```

其中 B_jump_en 由 Branch 模块判断得到，J_jal 和 I_jalr 在前面信号控制就可得到。

输出的地址则由信号判断来自于哪个跳转部件：

```
    wire [63:0] inst_addr = (J_jal)      ? jal_addr_out      :
                            (I_jalr)     ? jalr_addr_out     :
                            (B_jump_en)  ? branch_addr_out   :
                                           64'b0             ;
```

具体的模块设计这里就不必要详细说明了。需要注意的是 pc_reg 会在上升沿就更新，故在指令周期中上升沿前的 pc 为当前指令地址，上升沿后为当前指令地址 +4。而 bus3 在下降沿才会写入寄存器，故输入的 pc 应当以 pc+4 看待。例如 jal 指令：（jal_out 输出到 bus3，inst_addr 将输出到 pc-fetch 模块）

```
jal_out = pc;
inst_addr = pc - 4 + imm;
```

ALU 运算模块：

判断逻辑也已经在前面特点分析中讲过，模块设计如下：

RV M 扩展部分

```
module ALU_M_ext (
```

```
    input en,
    input [2:0] funct3,
    input [63:0] operand1, operand2,
    output reg [63:0] alu_M_Out
);
    parameter ALU_MUL       = 3'b000;
    parameter ALU_MULH      = 3'b001;
    parameter ALU_MULHSU    = 3'b010;
    parameter ALU_MULHU     = 3'b011;
    parameter ALU_DIV       = 3'b100;
    parameter ALU_DIVU      = 3'b101;
    parameter ALU_REM       = 3'b110;
    parameter ALU_REMU      = 3'b111;

    // arithmetic all result
    wire [127:0] result_mulss = $signed(operand1)   * $signed(operand2);
    wire [127:0] result_mulsu = $signed(operand1)   * $unsigned(operand2);
    wire [127:0] result_muluu = $unsigned(operand1) * $unsigned(operand2);

    wire [63:0] ans_mul     = result_mulss[63:0];
    wire [63:0] ans_mulh    = result_mulss[127:64];
    wire [63:0] ans_mulhsu  = result_mulsu[127:64];
    wire [63:0] ans_mulhu   = result_muluu[127:64];
    wire [63:0] ans_div     = $signed(operand1) / $signed(operand2);
    wire [63:0] ans_divu    = $unsigned(operand1) / $unsigned(operand2);
    wire [63:0] ans_rem     = $signed(operand1) % $signed(operand2);
    wire [63:0] ans_remu    = $unsigned(operand1) % $unsigned(operand2);

    // case result
    always @(*) begin
        if (~en) alu_M_Out = 64'b0;
        else begin
            case (funct3)
                ALU_MUL: alu_M_Out = ans_mul;
                ALU_MULH: alu_M_Out = ans_mulh;
                ALU_MULHSU: alu_M_Out = ans_mulhsu;
                ALU_MULHU: alu_M_Out = ans_mulhu;
                ALU_DIV: alu_M_Out = ans_div;
                ALU_DIVU: alu_M_Out = ans_divu;
                ALU_REM: alu_M_Out = ans_rem;
                ALU_REMU: alu_M_Out = ans_remu;
            endcase
        end
    end
endmodule;
```

RV I 及 M 扩展的整合：

```
module ALU (
    input en,
    input [1:0] mode, // {Rtype/Itype, 64/32}
    input [2:0] funct3,
    input [6:0] funct7,
    input [63:0] bus1, bus2, imm,
    output [63:0] alu_out
);
```

```
    wire [63:0] operand1 = (mode[0]) ? {{(32){bus1[31]}}, bus1[31:0]}   :
                                       bus1                             ;
    wire [63:0] operand2 = (mode[1]) ? imm                              :
                           (mode[0]) ? {{(32){bus2[31]}}, bus2[31:0]}   :
                                       bus2                             ;

    reg [7:0] debug_inst_alu_I;
    wire [7:0] debug_inst_alu_M_ext;

    reg [63:0] result_I64;
    wire [63:0] result_M_ext64;

    parameter ALU_ADD   = 3'b000;
    parameter ALU_SHL   = 3'b001;
    parameter ALU_SLT   = 3'b010;
    parameter ALU_SLTU  = 3'b011;
    parameter ALU_XOR   = 3'b100;
    parameter ALU_SHR   = 3'b101;
    parameter ALU_OR    = 3'b110;
    parameter ALU_AND   = 3'b111;

    // arithmetic all result
    wire [5:0]  shift_amt   = (mode[0]) ? {1'b0, operand2[4:0]} : operand2[5:0];

    wire [63:0] ans_add     = operand1 + operand2;
    wire [63:0] ans_sub     = operand1 - operand2;
    wire [63:0] ans_shl     = operand1 << shift_amt;
    wire [63:0] ans_slt     = ($signed(operand1) < $signed(operand2)) ? 64'b1 :
64'b0;
    wire [63:0] ans_sltu    = ($unsigned(operand1) < $unsigned(operand2)) ? 64'b1 :
64'b0;
    wire [63:0] ans_xor     = operand1 ^ operand2;
    wire [63:0] ans_srl64   = operand1 >> shift_amt;
    wire [63:0] ans_srl32   = {32'b0, operand1[31:0]} >> shift_amt;
    wire [63:0] ans_sra     = $signed(operand1) >>> shift_amt;
    wire [63:0] ans_or      = operand1 | operand2;
    wire [63:0] ans_and     = operand1 & operand2;

    ALU_M_ext alu_m_ext (
        .en(funct7[ALUINDEX_M_EXT] & ~mode[1]), .funct3(funct3),
.operand1(operand1), .operand2(operand2), .alu_M_Out(result_M_ext64)
    );

    // case result
    always @(*) begin
        if (~en) result_I64 = 64'b0;
        else begin
            case (funct3)
                ALU_ADD: result_I64 = (funct7[ALUINDEX_OP_MODIFIER] & ~mode[1]) ?
ans_sub : ans_add;
                ALU_SHL: result_I64 = ans_shl;
                ALU_SLT: result_I64 = ans_slt;
                ALU_SLTU: result_I64 = ans_sltu;
                ALU_XOR: result_I64 = ans_xor;
                ALU_SHR: result_I64 = (funct7[ALUINDEX_OP_MODIFIER]) ? ans_sra :
((mode[0]) ? ans_srl32 : ans_srl64);
```

```
                ALU_OR : result_I64 = ans_or;
                ALU_AND: result_I64 = ans_and;
            endcase
        end
    end

    wire [63:0] result_I32      = {{(32){result_I64[31]}}, result_I64[31:0]};
    wire [63:0] result_M_ext32  = {{(32){result_M_ext64[31]}},
result_M_ext64[31:0]};

    wire [63:0] result_I        = (mode[0])                             ?
result_I32        : result_I64;
    wire [63:0] result_M_ext    = (mode[0])                             ?
result_M_ext32    : result_M_ext64;
    assign alu_out              = (funct7[ALUINDEX_M_EXT] & ~mode[1])   ?
result_M_ext      : result_I;
endmodule;
```

ALU 需要考虑的部分较多。首先是需要区分操作数是有无符号，第二个操作数是否是 imm

而计算时需要考虑 32 位和 64 位的移位运算并不相同。

> SLLW, SRLW, and SRAW are RV64I-only instructions that are analogously defined but operate on 32-bit values and produce signed 32-bit results. The shift amount is given by rs2[4:0].

以及 add/sub 的区分为 `funct7[ALUINDEX_OP_MODIFIER] & ~mode[1]`（没有 subi 但有 addi）

srl/sra 则为 `funct7[ALUINDEX_OP_MODIFIER]`（既有 srli 也有 srai，且判断位相同）

最后除了按照有无符号位进行截断扩展选择外，还要注意 RV I 与 RV M 的区分。（M 扩展中没有 I 类型指令）

Uimm op 长立即数操作模块：

这部分按按指令实现即可

3.3.3 总线控制

立即数的控制在前面已经提到。寄存器的读写权限根据指令总结如下：

```
    wire reg_to_bus1_en = R_alu32   | R_alu64    | I_alu32    | I_alu64  |
                          I_memload | S_memstore |
                          I_jalr    | B_branch   ;
    wire reg_to_bus2_en = R_alu32   | R_alu64    | S_memstore | B_branch ;
    wire bus3_to_reg_en = R_alu32   | R_alu64    | I_alu32    | I_alu64  |
                          I_memload |
                          I_jalr    | J_jal      |
                          U_auipc   | U_lui      ;
```

bus1 和 bus2 的来源只有寄存器，故不需要太多区分

```
    wire [63:0] bus1 = (reg_to_bus1_en)  ? rs1            : 64'b0;
    wire [63:0] bus2 = (reg_to_bus2_en)  ? rs2            : 64'b0;
```

bus3 的来源多样，需要根据指令进行选择

```
wire [63:0] bus3 = (alu_en)          ? alu_out        :
                   (I_memload)       ? mem_out        :
                   (J_jal)           ? jal_out        :
                   (I_jalr)          ? jalr_out       :
                   (U_auipc)         ? auipc_out      :
                   (U_lui)           ? lui_out        :
                                       64'b0          ;
```

4 测试与gui接入

4.1 cpu-tests

完成了 cpu 的设计，可以使用 am-kernels/tests/cpu-tests 文件夹中的样例进行一键回归测试：

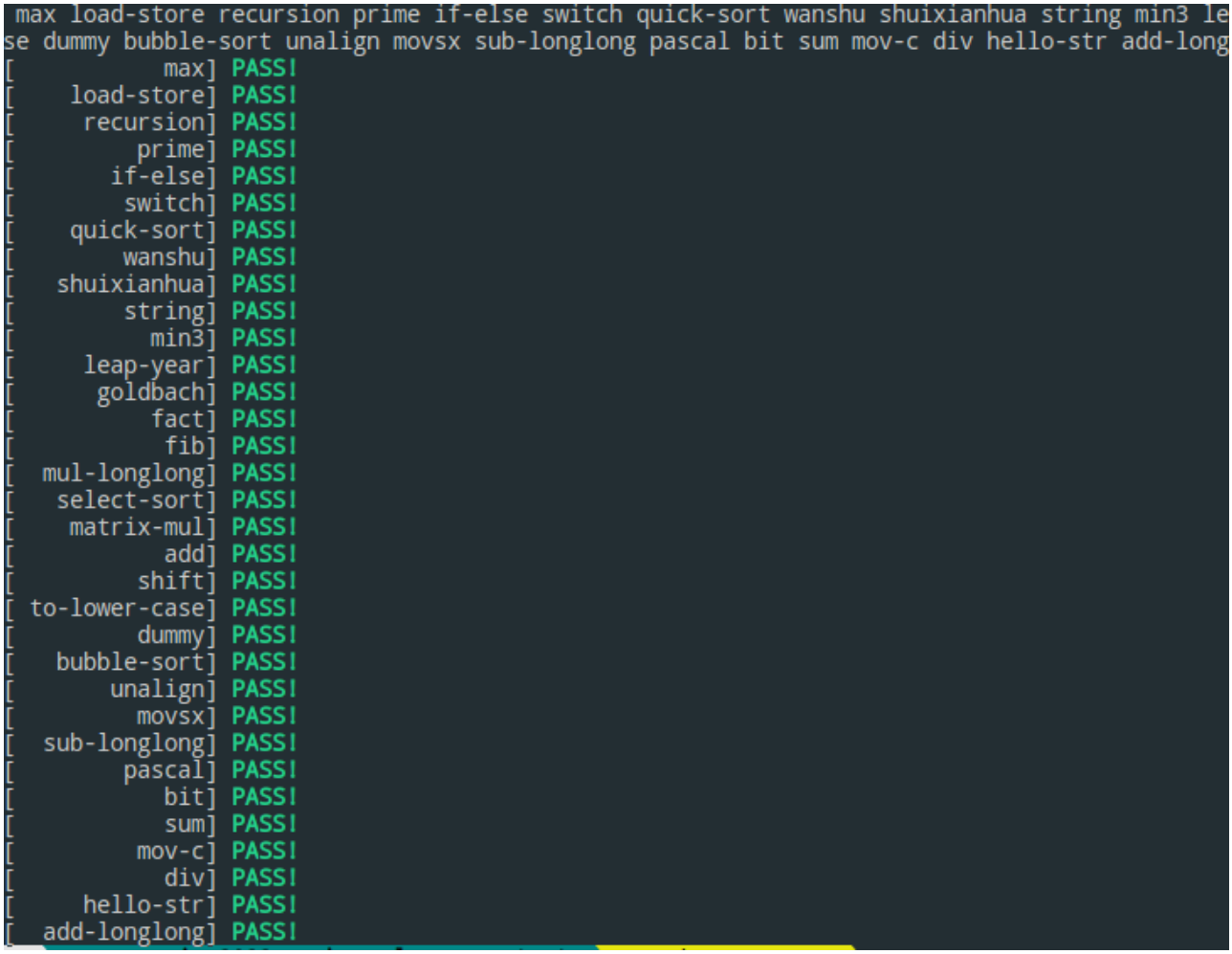
```
make -C $NEMU_HOME/../am-kernels/tests/cpu-tests ARCH=riscv64-nemu run
```

初次测试，如果 cpu 设计没有问题的话除了 hello-str 和 string 两个样例无法通过外，其余都能正确运行。

这两个测试样例需要开启 make menuconfig 中的 device 外设，并补全 abstract-machine/klib/src/stdio.c、abstract-machine/klib/src/stdlib.c、abstract-machine/klib/src/string.c 三个 kernel lib 库。

这部分代码可以参考 c 原生的代码实现，这里并不列出（上千行）

完成后再次测试，可以得到如下全 pass 的测试结果：



4.2 typing-game 与 红白机模拟器

完成 typing-game 和运行红白机模拟器需要补全键盘输入和 vga 显示：

`abstract-machine/am/src/platform/nemu/ioe/input.c`

```
#include <am.h>
#include <nemu.h>

#define KEYDOWN_MASK 0x8000

void __am_input_keybrd(AM_INPUT_KEYBRD_T *kbd) {
  int code = inl(KBD_ADDR);
  kbd->keydown = (code & KEYDOWN_MASK ? true : false);
  kbd->keycode = code & ~KEYDOWN_MASK;
}
```

`abstract-machine/am/src/platform/nemu/ioe/gpu.c`

```
#include <am.h>
#include <nemu.h>

#define W    400  // only support 400*300*32
#define H    300  // only support 400*300*32
#define SYNC_ADDR (VGACTL_ADDR + 4)

void __am_gpu_init() {
  int i;
  int w = W;
  int h = H;
  uint32_t *fb = (uint32_t *)(uintptr_t)FB_ADDR;
  for (i = 0; i < w * h; i ++){
    fb[i] = i;
  }
  outl(SYNC_ADDR, 1);
}

void __am_gpu_config(AM_GPU_CONFIG_T *cfg) {
  *cfg = (AM_GPU_CONFIG_T) {
    .present = true, .has_accel = false,
    .width = W, .height = H,
    .vmemsz = W*H*sizeof(uint32_t)
  };
}

void __am_gpu_fbdraw(AM_GPU_FBDRAW_T *ctl) {

  uint32_t *fb = (uint32_t *)(uintptr_t)FB_ADDR;
  uint32_t *pixels = (uint32_t*)(ctl->pixels);
  int x = ctl->x, y = ctl->y;
  int w = ctl->w, h = ctl->h;
  for(int j = 0; j< h; j++){
    for(int i = 0; i< w; i++){
      fb[(y+j)*W+(x+i)] = *(pixels+j*w+i);
    }
  }
```

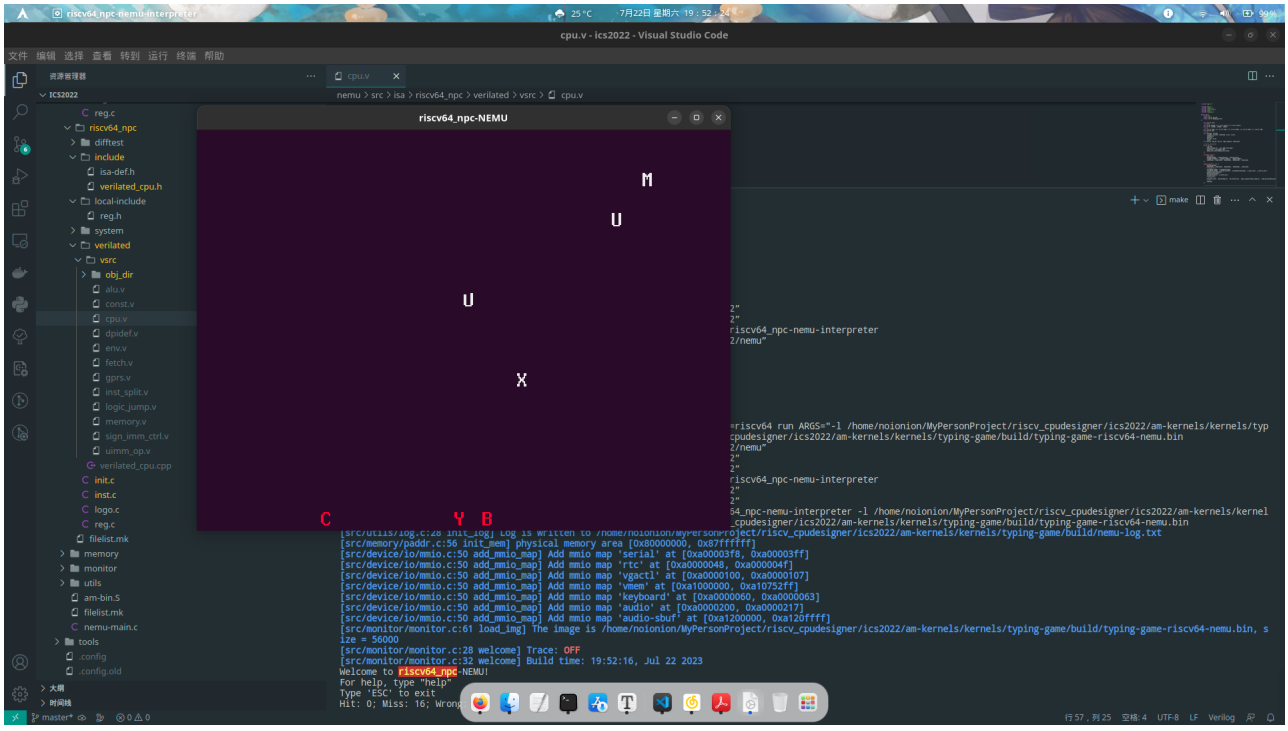
```
  if (ctl->sync) {
    outl(SYNC_ADDR, 1);
  }
}

void __am_gpu_status(AM_GPU_STATUS_T *status) {
  status->ready = true;
}
```

`nemu/src/device/vga.c` TODO 处

```
void vga_update_screen() {
  // TODO: call `update_screen()` when the sync register is non-zero,
  // then zero out the sync register
  if (vgactl_port_base[1]) {
    IFDEF(CONFIG_VGA_SHOW_SCREEN, update_screen());
    vgactl_port_base[1] = 0;
  }
}
```

运行 typing-game:

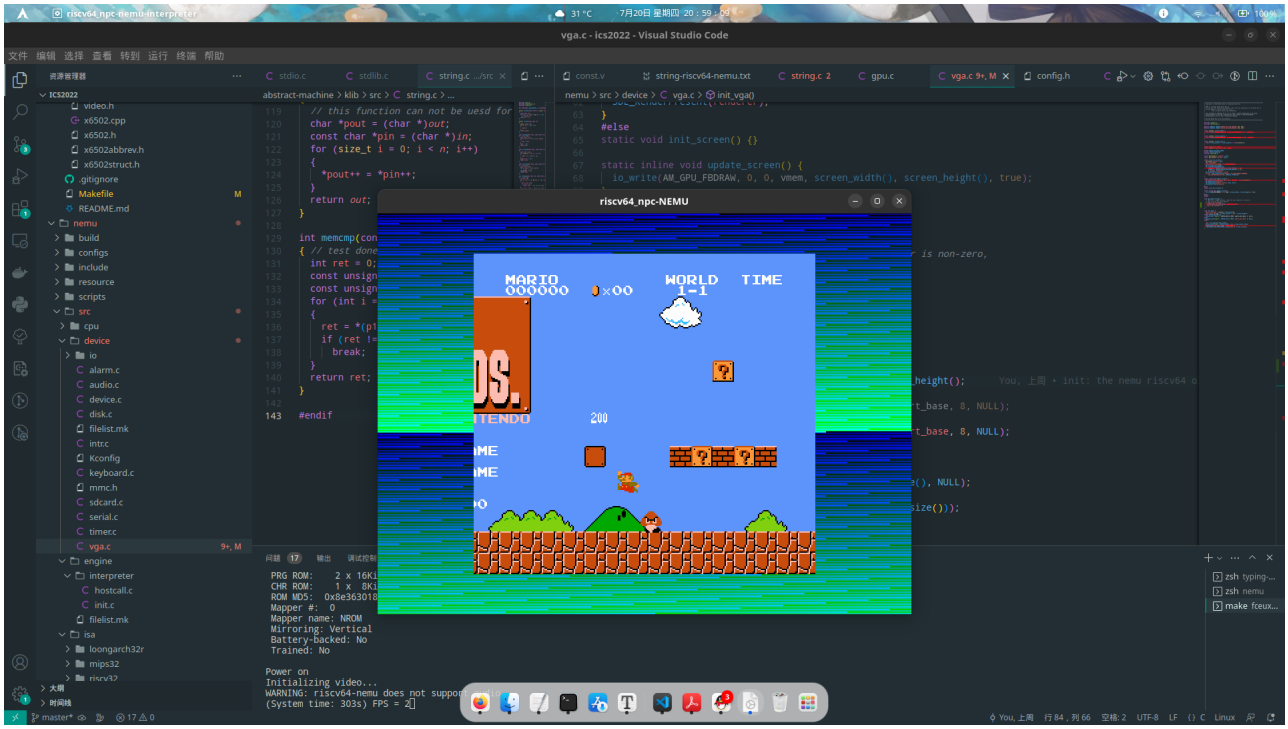
```
make -C $NEMU_HOME/../am-kernels/kernels/typing-game ARCH=riscv64-nemu run
```



运行红白机模拟器 超级马里奥：

```
make -C $NEMU_HOME/../fceux-am ARCH=riscv64-nemu run mainargs=mario
```

(虽然经过三层模拟后只剩下 0-2 帧左右的，不过勉强能玩)

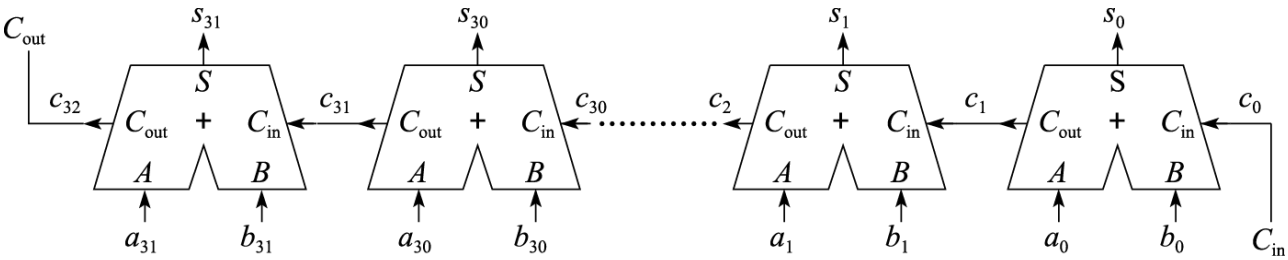


5 A 阶段部分内容（高速加乘法器设计）

5.1 64 位高速加法器设计

5.1.1 行波进位加法器

构建N位带进位加法器的最简单的方法是将N个一位全加器逐个串接起来。



这种串行的进位传递方式与人们日常演算十进制加法时采用的进位方式原理一样，非常直观。但是，这种加法器的电路中每一位的数据相加都必须等待低位的进位产生之后才能完成，即进位在各级之间是顺序传递的。对于一个64位的高性能通用CPU来说，在良好的流水线切分下，每级流水的延迟应控制在20级门以内，所以64位行波进位加法器高达129级门的延迟太长了。

5.1.2 先行进位加法器

为了改进行波进位加法器延迟随位数增加增长过快的缺点，人们提出了先行进位加法器的电路结构。其主要思想是先并行地计算每一位的进位，由于每一位的进位已经提前算出，这样计算每一个的结果只需要将本地和与进位相加即可。

并行进位逻辑如下：

设 gi = ai & bi，pi = ai | bi，则 进位 ci+1 计算逻辑为 ci+1 = gi | pi & ci。则一个 4 位先行进位加法器逻辑如下：

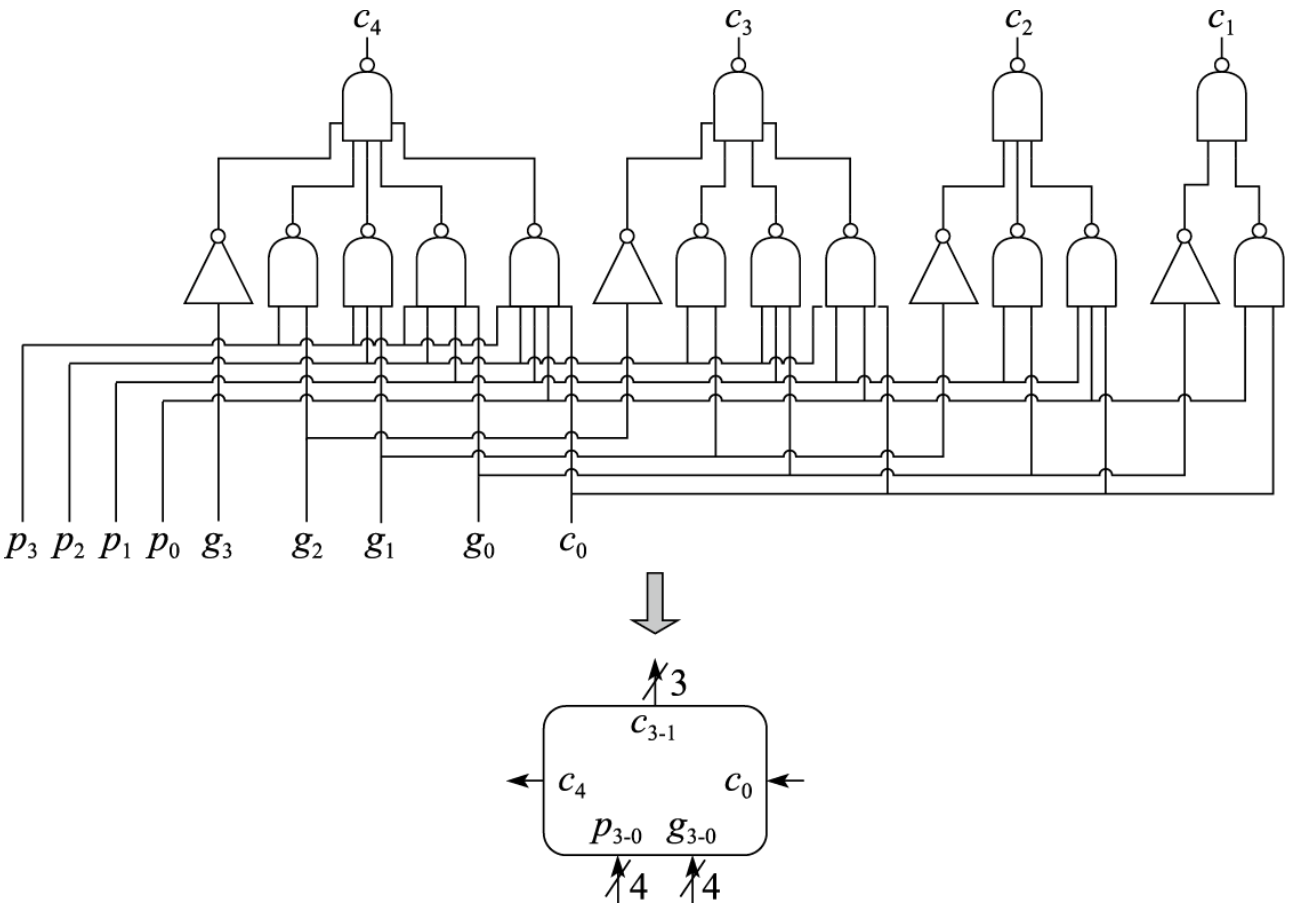
$$c_1 = g_0 \mid p_0 \,\&\, c_0$$

$$c_2 = g_1 \mid p_1 \,\&\, g_0 \mid p_1 \,\&\, p_0 \,\&\, c_0$$

$$c_3 = g_2 \mid p_2 \,\&\, g_1 \mid p_2 \,\&\, p_1 \,\&\, g_0 \mid p_2 \,\&\, p_1 \,\&\, p_0 \,\&\, c_0$$

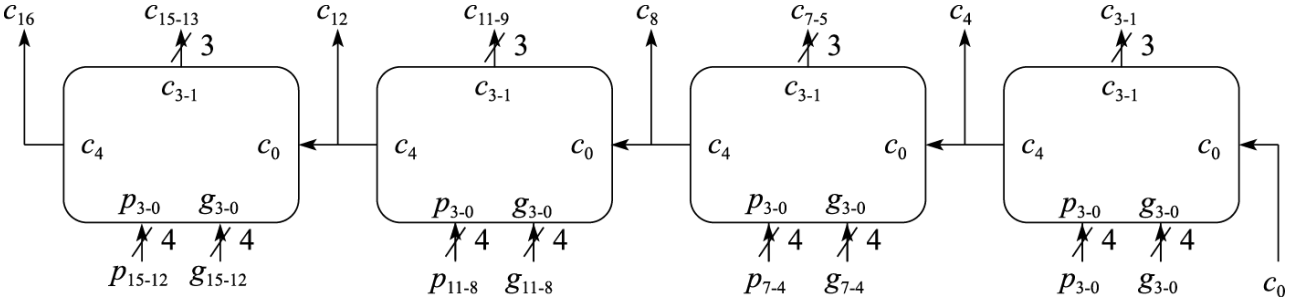
$$c_4 = g_3 \mid p_3 \,\&\, g_2 \mid p_3 \,\&\, p_2 \,\&\, g_1 \mid p_3 \,\&\, p_2 \,\&\, p_1 \,\&\, g_0 \mid p_3 \,\&\, p_2 \,\&\, p_1 \,\&\, p_0 \,\&\, c_0$$

采用先行进位逻辑，产生第4位的进位输出只需要2级门延迟，而之前介绍的行波进位逻辑则需要8级门延迟，先行进位逻辑的延迟显著地优于行波进位逻辑。



块内并行、块间串行逻辑：

理论上可以把上述并行进位方法扩展成更多位的情况，但那需要很多输入的逻辑门，在实现上是不现实的。实现更多位的加法器时通常采用分块的进位方法，将加法器分为若干个相同位数的块，块内通过先行进位的方法计算进位，块间通过行波进位的方法传递进位。



但是这样的进位延迟也是较大的。

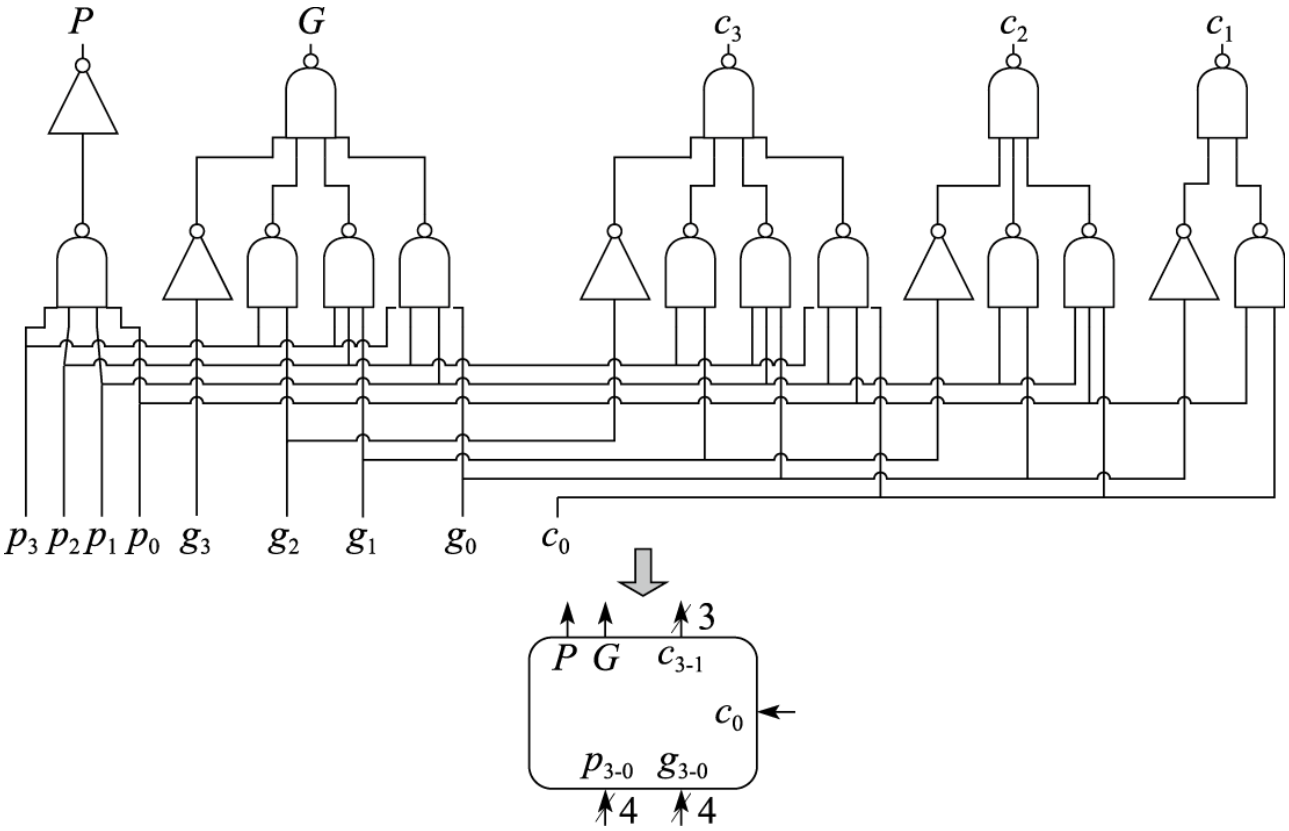
块内并行、块间并行逻辑：

为了进一步提升加法器的速度，可以在块间也采用先行进位的方法，即块内并行、块间也并行的进位实现方式。与前面类似，对于块内进位，定义其的进位生成因子为g和进位传递因子为p，对于块间的进位传递，定义其进位生成因子为G和块间进位传递因子为P，则其表达式如下：

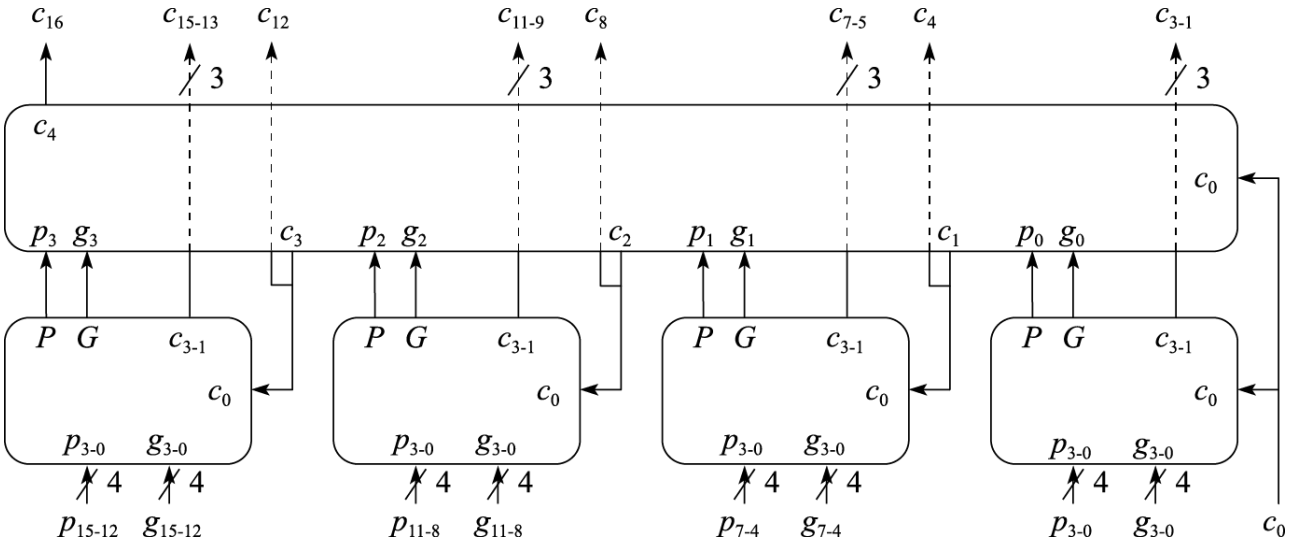
$$P = p_3 \ \& \ p_2 \ \& \ p_1 \ \& \ p_0$$

$$G = g_3 \mid p_3 \ \& \ g_2 \mid p_3 \ \& \ p_2 \ \& \ g_1 \mid p_3 \ \& \ p_2 \ \& \ p_1 \ \& \ g_0$$

上面的表达式可以解释为，当G为1时表示本块有进位输出生成，当P为1时表示当本块有进位输入时该进位可以传播至该块的进位输出。



例如下面所示的 16 位加法器结构：



可以看出，从pi和gi生成下层各块的P、G需要2级门延迟，上层根据自身pi和gi输入生成进位输出c1~c3需要2级门延迟，下层各块从c0输入至生成进位输出c1~c3也需要2级门延迟。所以整体来看，从pi和gi生成进位c1~c16最长的路径也只需要6级门延迟，这比前面介绍的块内并行但块间串行的电路结构更快。而且进一步分析可知，块间并行的电路结构中，最大的与非门的扇入为4，而前面分析块间串行电路结构延迟时，那个电路中最大的与非门的扇入为5。

5.1.3 具体实现

从上面的块间并行结构可以看到，64 位加法器可以由 4 个16位加法器再进行一次 4 位先行进位加法得到，于是可以构建出如下结构

```
    wire [15:0] P4, G4;
    add4_PG add4_PG_63_60 (.g(g[63:60]), .p(p[63:60]), .c0(c[60]), .c(c[63:61]),
.P(P4[15]), .G(G4[15]));
    add4_PG add4_PG_59_56 (.g(g[59:56]), .p(p[59:56]), .c0(c[56]), .c(c[59:57]),
.P(P4[14]), .G(G4[14]));
    add4_PG add4_PG_55_52 (.g(g[55:52]), .p(p[55:52]), .c0(c[52]), .c(c[55:53]),
.P(P4[13]), .G(G4[13]));
    add4_PG add4_PG_51_48 (.g(g[51:48]), .p(p[51:48]), .c0(c[48]), .c(c[51:49]),
.P(P4[12]), .G(G4[12]));

    add4_PG add4_PG_47_44 (.g(g[47:44]), .p(p[47:44]), .c0(c[44]), .c(c[47:45]),
.P(P4[11]), .G(G4[11]));
    add4_PG add4_PG_43_40 (.g(g[43:40]), .p(p[43:40]), .c0(c[40]), .c(c[43:41]),
.P(P4[10]), .G(G4[10]));
    add4_PG add4_PG_39_36 (.g(g[39:36]), .p(p[39:36]), .c0(c[36]), .c(c[39:37]),
.P(P4[9 ]), .G(G4[9 ]));
    add4_PG add4_PG_35_32 (.g(g[35:32]), .p(p[35:32]), .c0(c[32]), .c(c[35:33]),
.P(P4[8 ]), .G(G4[8 ]));

    add4_PG add4_PG_31_28 (.g(g[31:28]), .p(p[31:28]), .c0(c[28]), .c(c[31:29]),
.P(P4[7 ]), .G(G4[7 ]));
    add4_PG add4_PG_27_24 (.g(g[27:24]), .p(p[27:24]), .c0(c[24]), .c(c[27:25]),
.P(P4[6 ]), .G(G4[6 ]));
    add4_PG add4_PG_23_20 (.g(g[23:20]), .p(p[23:20]), .c0(c[20]), .c(c[23:21]),
.P(P4[5 ]), .G(G4[5 ]));
    add4_PG add4_PG_19_16 (.g(g[19:16]), .p(p[19:16]), .c0(c[16]), .c(c[19:17]),
.P(P4[4 ]), .G(G4[4 ]));

    add4_PG add4_PG_15_12 (.g(g[15:12]), .p(p[15:12]), .c0(c[12]), .c(c[15:13]),
.P(P4[3 ]), .G(G4[3 ]));
    add4_PG add4_PG_11_08 (.g(g[11:8 ]), .p(p[11:8 ]), .c0(c[8 ]), .c(c[11:9 ]),
.P(P4[2 ]), .G(G4[2 ]));
    add4_PG add4_PG_07_04 (.g(g[7 :4 ]), .p(p[7 :4 ]), .c0(c[4 ]), .c(c[7 :5 ]),
.P(P4[1 ]), .G(G4[1 ]));
    add4_PG add4_PG_03_00 (.g(g[3 :0 ]), .p(p[3 :0 ]), .c0(c[0 ]), .c(c[3 :1 ]),
.P(P4[0 ]), .G(G4[0 ]));

    // 4 个 4位先行进位加法器:
    wire [3:0] P16, G16;
    add4_PG add4_PG_63_48 (.g(G4[15:12]), .p(P4[15:12]), .c0(c[48]), .c({c[60],
c[56], c[52]}), .P(P16[3]), .G(G16[3]));
    add4_PG add4_PG_47_32 (.g(G4[11:8 ]), .p(P4[11:8 ]), .c0(c[32]), .c({c[44],
c[40], c[36]}), .P(P16[2]), .G(G16[2]));
    add4_PG add4_PG_31_16 (.g(G4[7 :4 ]), .p(P4[7 :4 ]), .c0(c[16]), .c({c[28],
c[24], c[20]}), .P(P16[1]), .G(G16[1]));
```

```
    add4_PG add4_PG_15_00 (.g(G4[3 :0 ]), .p(P4[3 :0 ]), .c0(c[0 ]), .c({c[12], c[8
], c[4 ]}), .P(P16[0]), .G(G16[0]));

    // 1 个 4位先行进位加法器，这个不需要输出P、G
    add4    add4_63_00    (.g(G16[3:0]), .p(P16[3:0]), .c0(c[0 ]), .c({c[64],
c[48], c[32], c[16]}));

    // 求结果
    assign carry  = c[64];
    assign result = (~operand1 & ~operand2 & c[63:0]) | (~operand1 & operand2 &
~c[63:0]) | operand1 & ~operand2 & ~c[63:0] | (operand1 & operand2 & c[63:0]);
```

其中先行进位加法器设计如下：

```
// 先行进位加法器，16 + 4
// 4位加，并行，延时约 2 级门
module add4_PG (
    input [3:0] g, p,   // gi = ai & bi, pi = ai | bi. i in [3:0]
    input c0,
    output [2:0] c,     // c[3:1]
    output P, G
);
    assign c[0] = g[0] | (p[0] & c0);
    assign c[1] = g[1] | (p[1] & g[0]) | (p[1] & p[0] & c0);
    assign c[2] = g[2] | (p[2] & g[1]) | (p[2] & p[1] & g[0]) | (p[2] & p[1] & p[0]
& c0);

    assign P = p[3] & p[2] & p[1] & p[0];
    assign G = g[3] | (p[3] & g[2]) | (p[3] & p[2] & g[1]) | (p[3] & p[2] & p[1] &
g[0]);
endmodule;

module add4 (
    input [3:0] g, p,   // gi = ai & bi, pi = ai | bi. i in [3:0]
    input c0,
    output [3:0] c      // c[4:1]
);
    assign c[0] = g[0] | (p[0] & c0);
    assign c[1] = g[1] | (p[1] & g[0]) | (p[1] & p[0] & c0);
    assign c[2] = g[2] | (p[2] & g[1]) | (p[2] & p[1] & g[0]) | (p[2] & p[1] & p[0]
& c0);
    assign c[3] = g[3] | (p[3] & g[2]) | (p[3] & p[2] & g[1]) | (p[3] & p[2] & p[1]
& g[0]) | (p[3] & p[2] & p[1] & p[0] & c0);
endmodule;
```

经检验计算结果无误。

5.2 128 位高速乘法器设计

5.2.1 booth 乘法器

Booth乘法器由英国的Booth夫妇提出。按照最简单的补码乘法算法，需要特地挑出第N个部分积，并使用补码减法操作（由于最高位的符号位计算），这就需要实现一个额外的状态机来控制，增加了硬件设计复杂度。因此他们对补码乘法公式进行变换，试图找到更适合于硬件实现的算法。

Booth一位乘变换的公式推导如下：

$$
\begin{aligned}
&(-y_7 \times 2^7 + y_6 \times 2 \ldots\ldots 6 + \ldots + y_1 \times 2^1 + y_0 \times 2^0) \\
=\ &(-y_7 \times 2^7 + (y_6 \times 2^7 - y_6 \times 2^6) + (y_5 \times 2^6 - y_5 \times 2^5) + \ldots + \\
&(y_1 \times 2^2 - y_1 \times 2^1) + (y_0 \times 2^1 - y_0 \times 2^0) + (0 \times 2^0)) \\
=\ &(y_6 - y_7) \times 2^7 + (y_5 - y_6) \times 2^6 + \ldots + (y_0 - y_1) \times 2^1 + (y_{-1} - y_0) \times 2^0
\end{aligned}
$$

经过变换，公式变得更加规整，不再需要专门对最后一次部分积采用补码减法，更适合硬件实现。这个新公式被称为Booth一位乘算法。

在Booth一位乘算法中，为了计算N位的补码乘法，依然需要N-1次加法。而数据宽度较大的补码加法器面积大、电路延迟长，限制了硬件乘法器的计算速度，因此重新对补码乘法公式进行变换，得到Booth两位乘算法：

$$
\begin{aligned}
&(-y_7 \times 2^7 + y_6 \times 2^6 + \ldots + y_1 \times 2^1 + y_0 \times 2^0) \\
=\ &(-2 \times y_7 \times 2^6 + y_6 \times 2^6 + (y_5 \times 2^6 - 2 \times y_5 \times 2^4) + \ldots \\
&+ y_1 \times 2^2 - 2 \times y_1 \times 2^0) + y_0 \times 2^0 + y_{-1} \times 2^0) \\
=\ &(y_5 + y_6 - 2y_7) \times 2^6 + (y_3 + y_4 - 2y_5) \times 2^4 + \ldots + (y_{-1} + y_0 - 2y_1) \times 2^0
\end{aligned}
$$

根据Booth两位乘算法，需要每次扫描3位的乘数，并在每次累加完成后，将被乘数和乘数移2位。根据算法公式，可以推导出操作的方式

y_{i+1}	y_i	y_{i-1}	操作
0	0	0	不需要加（+0）
0	0	1	补码加X（$+[X]_{补}$）
0	1	0	补码加X（$+[X]_{补}$）
0	1	1	补码加2X（$+[X]_{补}$左移）
1	0	0	补码减2X（$-[X]_{补}$左移）
1	0	1	补码减X（$-[X]_{补}$）
1	1	0	补码减X（$-[X]_{补}$）
1	1	1	不需要加（+0）

如果使用Booth两位乘算法，计算N位的补码乘法时，只需要N/2-1次加法，如果使用移位加策略，则需要N/2个时钟周期来完成计算。龙芯处理器就采用了Booth两位乘算法来实现硬件补码乘法器，大多数现代处理器也均采用该算法。

经 Booth 两位乘法后，由于考虑到符号位运算，我们需要把 64 位的两个操作数扩展到 66 位。但是即使采用了Booth两位乘算法，使用移位加策略来完成一个64位的乘法操作也需要 33 个时钟周期，并且不支持流水操作，即第一条乘法全部完成之后才能开始计算下一条。故而 Booth 两位乘运算后产生的 33 个部分积，可以使用华莱士树进行运算。

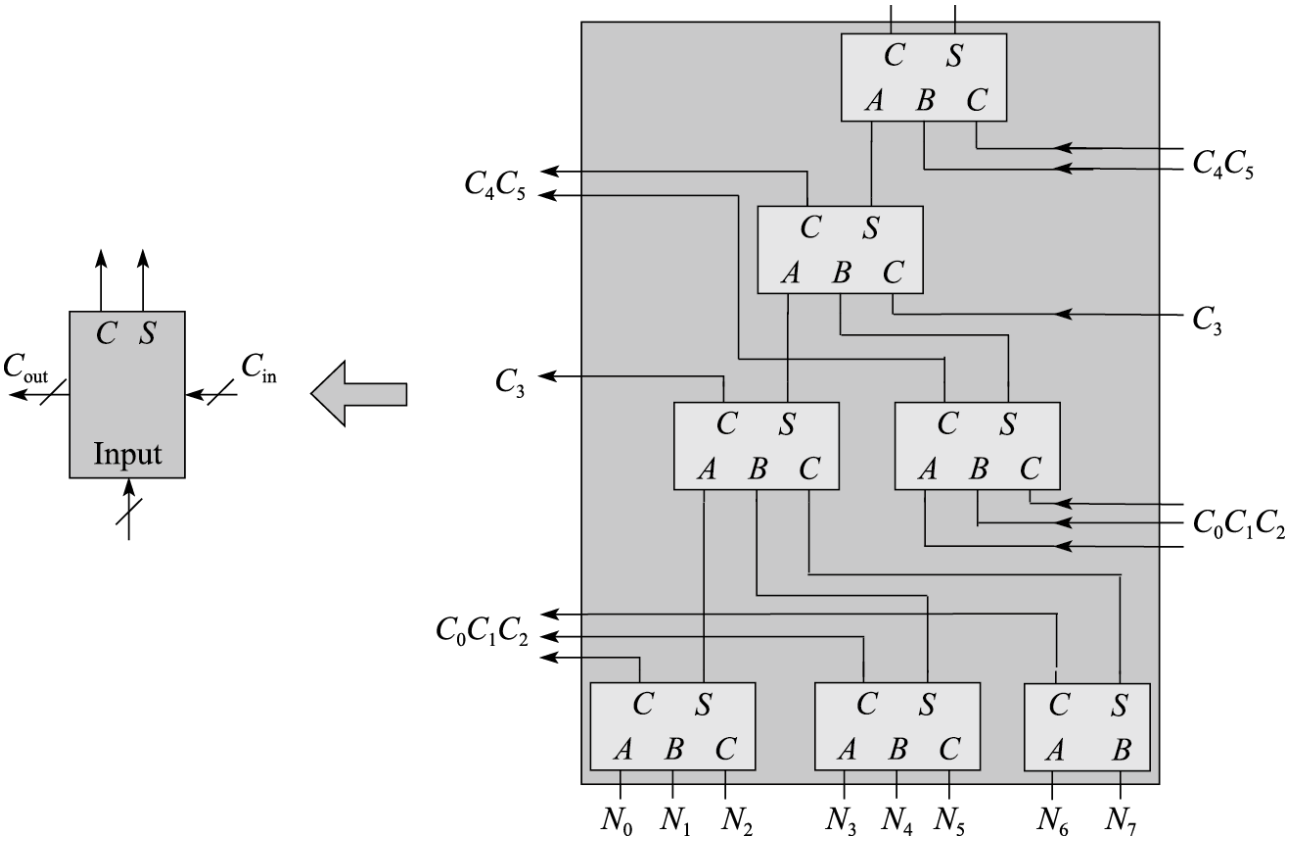
5.2.2 华莱士树

以64位数据的乘法为例，共有32个部分积，如果按照二叉树方式来搭建加法结构，第一拍执行16个加法，第二拍执行8个加法，以此类推，就可以在5个时钟周期内结束运算。这个设计还支持流水式操作：当上一条乘法指令到达第二级，此时第一级的16个加法器已经空闲，可以用来服务下一条乘法指令了。

这种设计的硬件开销非常大，其中128位宽度的加法器就需要31个，而用于锁存中间结果的触发器更是接近4000个。但华莱士树（Wallace Tree）结构可以大幅降低多个数相加的硬件开销和延迟。华莱士树使用全加器进行架构，4个数相加的华莱士树需要两层全加器，当前位的进位信号在第一层产生，并接到了下一位的第二层，这意味着 Cout 与 Cin 无关。全加器的S输出需要3级门延迟，而C输出需要2级门延迟，因此不论参与加法的数据宽度是多少位，将4个数相加转换为两个数相加最多只需要6级门延迟，最后把这两个数加起来还需要一个加法器。整套逻辑需要一个加法器的面积，再加上两倍数据宽度个全加器的面积。如果不使用华莱士树，而是先将四个数捉对相加，再把结果相加，计算的延迟就是两倍的加法器延迟，面积则是3倍的加法器面积。对于64位或者更宽的加法器，它的延迟肯定是远远超过6级门的，面积也比64个全加器要大得多。

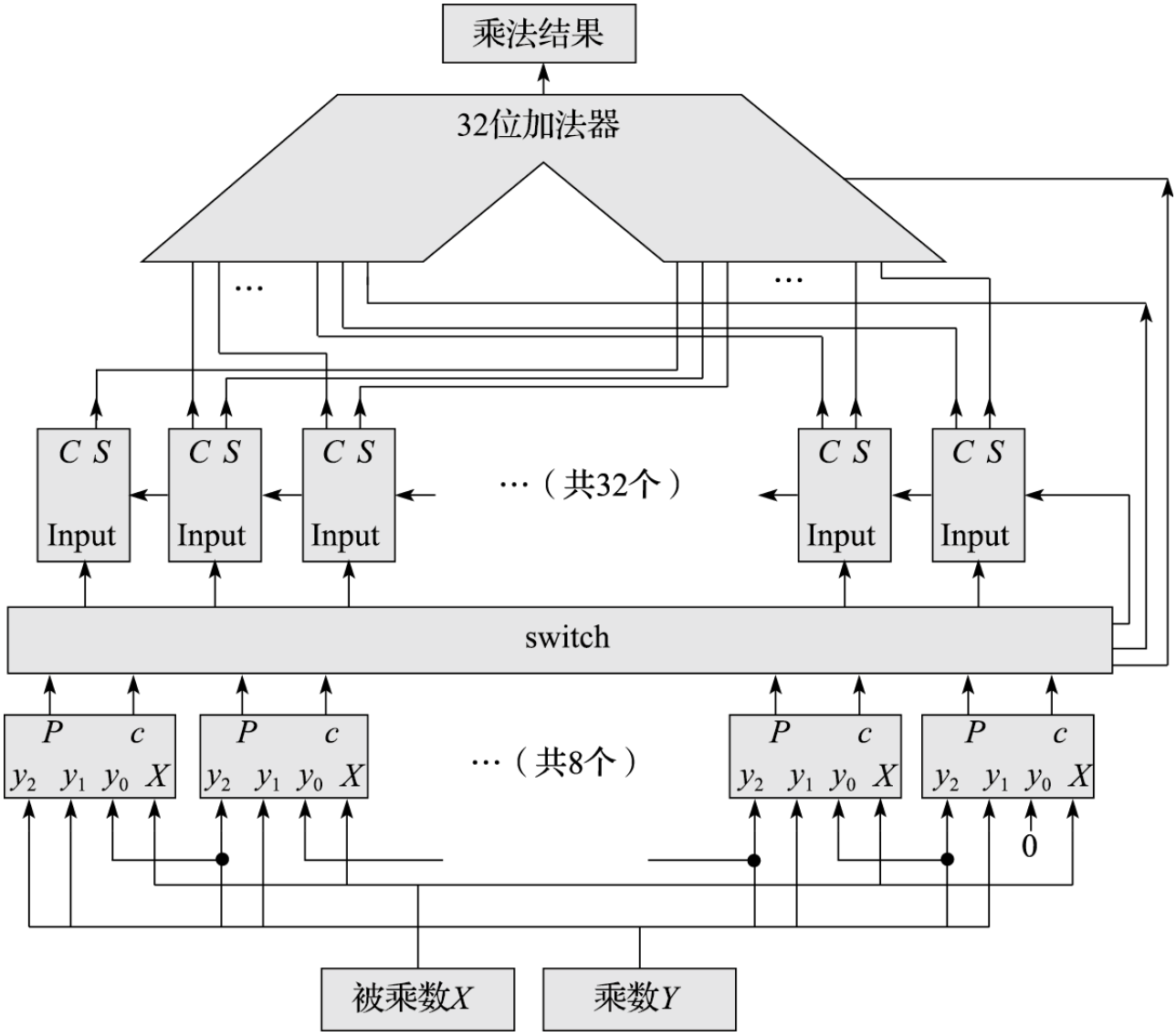
因此使用华莱士树进行多个数相加可以明显地降低计算延迟，数据宽度越宽，其效果越明显。通过本节后续的介绍可以归纳出，使用华莱士树进行M个N位数相加，可以大致降低延迟logN倍，而每一层华莱士树包含的全加器个数为⌊2M’/3⌋（M’是当前层次要加的数字个数）。

8个数相加的一位华莱士树如下所示：



从图中可以看出，通过华莱士树可以用4级全加器即12级门的延迟把8个数转换成两个数相加。华莱士树的精髓在于：通过连线实现进位传递，从而避免了复杂的进位传递逻辑。不过需要指出的是，在华莱士树中，每一级全加器生成本地和以及向高位的进位，因此在每一级华莱士树生成的结果中，凡是由全加器的进位生成的部分连接到下一级时要连接到下一级的高位。

为了构成一个16位定点补码乘法器，需要使用8个Booth编码器，外加32个8个数相加的一位华莱士树，再加上一个32位加法器。

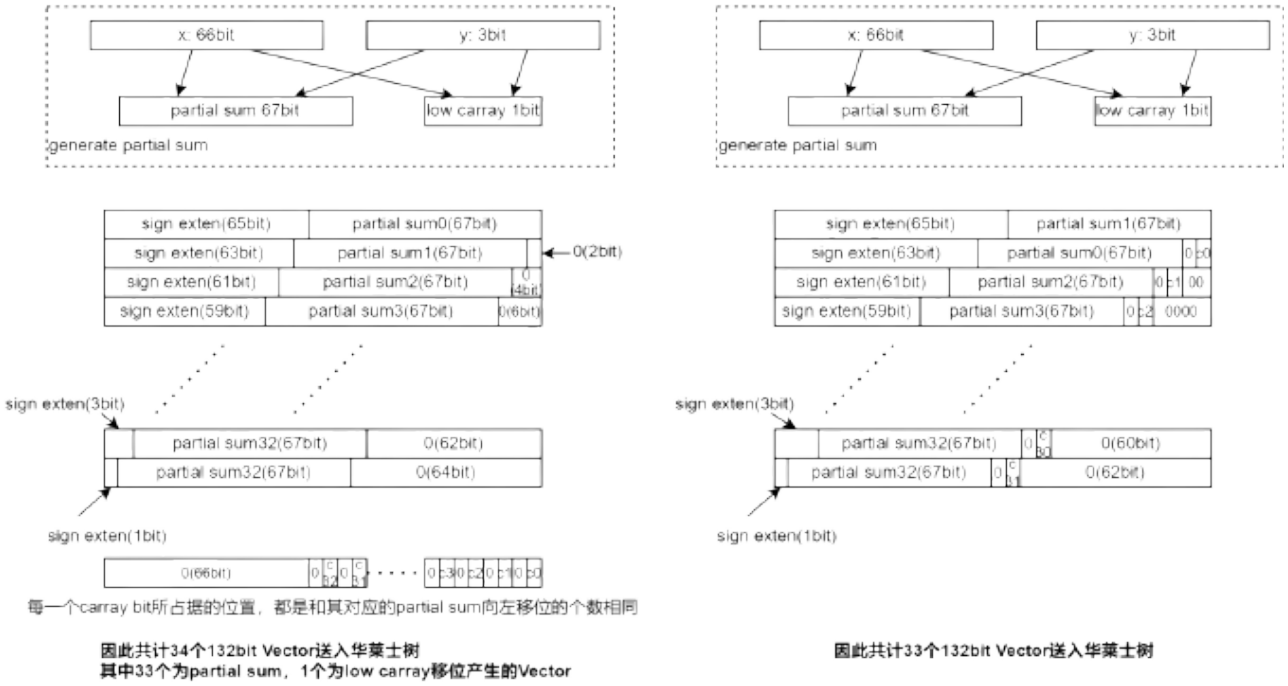


5.2.3 具体设计及优化

华莱士树这一步，我们需要对 33 个部分积进行加和，而每一次加和都会产生进位。如果每一步都进行进位运算的话，对于取−[X]补和−2[X]补的情况，需要进行取反再+1。我们可以想象的是，对于每一个部分积的生成时，都会生成一个加法器。在本例中就是生成 33 个 (66+1)bit 的加法器。这对于延迟和面积来说，都是无法接受的。而且+1带来的延迟是很高的，意味每个加法器内部都有66个bit的串行延迟。

如果将进位放在一起，那则是变成 34 个数进行加和，要多增加一个加法器

booth编码产生的partial sum，需要再左移对应的bit，才算生成完毕最终的booth编码。这就给了我们一个可乘之机：如果每个booth编码产生的partial sum[i]不进行+1，而是将其放在partial sum[i+1]的低位上(反正本身也要补零，0+1一定是1)，这样就不会引入额外的partial sum了。对比如下（右侧是现在的优化）：



优化后 booth 运算逻辑如下：

```
module booth #(
    parameter INXY_W = 66,
    parameter PSUM_W = INXY_W*2,
    parameter PSUM_N = INXY_W/2
) (
    input  [INXY_W-1:0] x,
    input  [INXY_W-1:0] y,
    output [PSUM_W-1:0] psum [PSUM_N-1:0]
);
    logic [INXY_W:0] psum_raw [PSUM_N-1:0];
    logic            clow_raw [PSUM_N-1:0];

    booth_sel #(.WIDTH(INXY_W)) b_sel_0(.x(x), .sel({y[1:0], 1'b0}),
.psum(psum_raw[0]), .carry(clow_raw[0]));
    assign psum[0] = {{(INXY_W-1){psum_raw[0][INXY_W]}}, psum_raw[0]};
    for (genvar i = 1; i < PSUM_N; i++) begin
        booth_sel #(.WIDTH(INXY_W)) b_sel_(.x(x), .sel(y[2*i+1:2*i-1]),
.psum(psum_raw[i]), .carry(clow_raw[i]));
        assign psum[i] = {{(INXY_W-1-2*i){psum_raw[i][INXY_W]}}, psum_raw[i], 1'b0,
clow_raw[i-1], {(2*i-2){1'b0}}};
    end
endmodule;

module booth_sel #(
    parameter WIDTH = 32
) (
    input  [WIDTH-1:0] x,
    input  [2      :0] sel,
    output [WIDTH  :0] psum,
    output             carry
);
    wire sel_neg  =  sel[2] & (sel[1] ^ sel[0]);
    wire sel_pos  = ~sel[2] & (sel[1] ^ sel[0]);
    wire sel_dneg =  sel[2] & ~sel[1] & ~sel[0];
```

```
    wire sel_dpos = ~sel[2] &  sel[1] &  sel[0];

    assign psum  = sel_neg  ? ~{x[WIDTH-1], x} :
                   sel_pos  ?  {x[WIDTH-1], x} :
                   sel_dneg ? ~{x, 1'b0}       :
                   sel_dpos ?  {x, 1'b0}       :
                               {(WIDTH+1){1'b0}};
    assign carry = sel[2] & ~(sel[1] & sel[0]);
endmodule;
```

另外在华莱士树中的 csa 选择中，我们不止有 3:2 压缩器一种选择，而还有 4:2 压缩。采用 4:2 压缩可以使延迟进一步减少

如果纯采用 3:2 压缩器对 33 组部分积进行加和，则需要经过 8 层压缩才能得到最后的两组操作数。而引入了 4:2 压缩器后则只需要 5 层（其中三层需要 4:2 压缩）。虽然在面积上会大幅增加，不过都用华莱士树了那还考虑啥啊。

不过因为 33 个部分积的低位是有一堆 0 的，也不需要每一层堆满全加器（节省面积一点的 verilog 就会变得非常复杂），华莱士树部分代码如下：

```
`include "csa.v"

module wallace_33 (
    input  [131:0] in  [32:0],
    output [127:0] out [1 :0]
);
    // level 1
    // 33p split by (444444 333) -> 18p
    // totle: 9
    wire [132:0] s_row1  [8:0];
    wire [132:0] c_row1  [8:0];
    wire [132:0] co_row1 [8:0];
        // 1.0
        assign c_row1[0][0] = 1'b0;
        half_adder      comp1_0_0 (.a(in[0][0]), .b(in[1][0]),
                                   .s(s_row1[0][0]), .cout(c_row1[0][1]));
        assign s_row1[0][1] = in[0][1];     assign c_row1[0][2] = 1'b0;
        compressor_3to2 comp1_0_2 (.a(in[0][2]), .b(in[1][2]), .cin(in[2][2]),
                                   .s(s_row1[0][2]), .cout(c_row1[0][3]));
        half_adder      comp1_0_3 (.a(in[0][3]), .b(in[1][3]),
                                   .s(s_row1[0][3]), .cout(c_row1[0][4]));
        compressor_4to2 comp1_0_4 (.a(in[0][4]), .b(in[1][4]), .c(in[2][4]),
.d(in[3][4]),      .cin(1'b0), .s(s_row1[0][4]),   .co(c_row1[0][5]),
.cout(co_row1[0][5]));
        compressor_4to2 comp1_0_5 (.a(in[0][5]), .b(in[1][5]), .c(in[2][5]),
.d(1'b0), .cin(co_row1[0][5]), .s(s_row1[0][5]),   .co(c_row1[0][6]),
.cout(co_row1[0][6]));
        for(genvar i = 6; i < 132; i++) begin   // booth 有扩展符号
            compressor_4to2 comp1_0_ (.a(in[0][i]), .b(in[1][i]), .c(in[2][i]),
.d(in[3][i]), .cin(co_row1[0][i]), .s(s_row1[0][i]), .co(c_row1[0][i+1]),
.cout(co_row1[0][i+1]));
        end
        // 1.1 - 1.5
        for (genvar i = 1; i < 6; i++) begin
            localparam start    = 8*i-2   ;
            localparam index0   = 4*i;
            localparam index1   = 4*i+1;
```

```
            localparam index2   = 4*i+2;
            localparam index3   = 4*i+3;

            assign c_row1[i][start] = 1'b0;
            assign s_row1[i][start] = in[index0][start];        assign c_row1[i]
[start+1] = 1'b0;
            assign s_row1[i][start+1] = 1'b0;                   assign c_row1[i]
[start+2] = 1'b0;
            half_adder      comp1_i_2 (.a(in[index0][start+2]), .b(in[index1]
[start+2]), .s(s_row1[i][start+2]), .cout(c_row1[i][start+3]));
            assign s_row1[i][start+3] = in[index0][start+3];    assign c_row1[i]
[start+4] = 1'b0;
            compressor_3to2 comp1_i_4 (.a(in[index0][start+4]), .b(in[index1]
[start+4]), .cin(in[index2][start+4]), .s(s_row1[i][start+4]), .cout(c_row1[i]
[start+5]));
            half_adder      comp1_i_5 (.a(in[index0][start+5]), .b(in[index1]
[start+5]), .s(s_row1[i][start+5]), .cout(c_row1[i][start+6]));
            for (genvar j = start+6; j < 132; j++) begin
                compressor_4to2 comp1_i_ (.a(in[index0][j]), .b(in[index1][j]),
.c(in[index2][j]), .d(in[index3][j]), .cin(co_row1[i][j]), .s(s_row1[i][j]),
.co(c_row1[i][j+1]), .cout(co_row1[i][j+1]));
            end
        end
        // 1.6 - 1.8
        for (genvar i = 0; i < 3; i++) begin
            localparam start    = 46+6*i;
            localparam index0   = 24+3*i;
            localparam index1   = index0+1;
            localparam index2   = index0+2;

            assign c_row1[i+6][start] = 1'b0;
            assign s_row1[i+6][start] = in[index0][start];        assign
c_row1[i+6][start+1] = 1'b0;
            assign s_row1[i+6][start+1] = 1'b0;                   assign
c_row1[i+6][start+2] = 1'b0;
            half_adder      comp1_i3_2 (.a(in[index0][start+2]), .b(in[index1]
[start+2]), .s(s_row1[i+6][start+2]), .cout(c_row1[i+6][start+3]));
            assign s_row1[i+6][start+3] = in[index0][start+3];    assign
c_row1[i+6][start+4] = 1'b0;
            compressor_3to2 comp1_i3_4 (.a(in[index0][start+4]), .b(in[index1]
[start+4]), .cin(in[index2][start+4]), .s(s_row1[i+6][start+4]), .cout(c_row1[i+6]
[start+5]));
            half_adder      comp1_i3_5 (.a(in[index0][start+5]), .b(in[index1]
[start+5]), .s(s_row1[i+6][start+5]), .cout(c_row1[i+6][start+6]));
            for (genvar j = start+6; j < 132; j++) begin
                compressor_3to2 comp1_i3_ (.a(in[index0][j]), .b(in[index1][j]),
.cin(in[index2][j]), .s(s_row1[i+6][j]), .cout(c_row1[i+6][j+1]));
            end
        end

    // level 2
    // 18p split by (333333) -> 12p
    // totle: 6
    wire [132:0] s_row2  [5:0];
    wire [132:0] c_row2  [5:0];
        // 2.0
```

```
        for (genvar i = 0; i < 6; i++) begin
            half_adder      comp2_0_ (.a(s_row1[0][i]), .b(c_row1[0][i]),
.s(s_row2[0][i]), .cout(c_row2[0][i+1]));
        end
        for (genvar i = 6; i < 132; i++) begin
            compressor_3to2 comp2_0_ (.a(s_row1[0][i]), .b(c_row1[0][i]),
.cin(s_row1[1][i]), .s(s_row2[0][i]), .cout(c_row2[0][i+1]));
        end
        // 2.1
        for (genvar i = 6; i < 14; i++) begin
            assign s_row2[1][i] = c_row1[1][i];
        end
        for (genvar i = 14; i < 132; i++) begin
            compressor_3to2 comp2_1_ (.a(c_row1[1][i]), .b(s_row1[2][i]),
.cin(c_row1[2][i]), .s(s_row2[1][i]), .cout(c_row2[1][i+1]));
        end
        // 2.2
        for (genvar i = 22; i < 30; i++) begin
            half_adder      comp2_2_ (.a(s_row1[3][i]), .b(c_row1[3][i]),
.s(s_row2[2][i]), .cout(c_row2[2][i+1]));
        end
        for (genvar i = 30; i < 132; i++) begin
            compressor_3to2 comp2_2_ (.a(s_row1[3][i]), .b(c_row1[3][i]),
.cin(s_row1[4][i]), .s(s_row2[2][i]), .cout(c_row2[2][i+1]));
        end
        // 2.3
        for (genvar i = 30; i < 38; i++) begin
            assign s_row2[3][i] = c_row1[4][i];
        end
        for (genvar i = 38; i < 132; i++) begin
            compressor_3to2 comp2_3_ (.a(c_row1[4][i]), .b(s_row1[5][i]),
.cin(c_row1[5][i]), .s(s_row2[3][i]), .cout(c_row2[3][i+1]));
        end
        // 2.4
        for (genvar i = 46; i < 52; i++) begin
            half_adder      comp2_4_ (.a(s_row1[6][i]), .b(c_row1[6][i]),
.s(s_row2[4][i]), .cout(c_row2[4][i+1]));
        end
        for (genvar i = 52; i < 132; i++) begin
            compressor_3to2 comp2_4_ (.a(s_row1[6][i]), .b(c_row1[6][i]),
.cin(s_row1[7][i]), .s(s_row2[4][i]), .cout(c_row2[4][i+1]));
        end
        // 2.5
        for (genvar i = 52; i < 58; i++) begin
            assign s_row2[5][i] = c_row1[7][i];
        end
        for (genvar i = 58; i < 132; i++) begin
            compressor_3to2 comp2_5_ (.a(c_row1[7][i]), .b(s_row1[8][i]),
.cin(c_row1[8][i]), .s(s_row2[5][i]), .cout(c_row2[5][i+1]));
        end

    // level 3
    // 12p split by (3333) -> 8p
    // totle: 4
    wire [132:0] s_row3  [3:0];
    wire [132:0] c_row3  [3:0];
```

```
        for (genvar i = 0; i < 6; i++) begin
            half_adder      comp3_0_ (.a(s_row2[0][i]), .b(c_row2[0][i]),
.s(s_row3[0][i]), .cout(c_row3[0][i+1]));
        end
        for (genvar i = 6; i < 132; i++) begin
            compressor_3to2 comp3_0_ (.a(s_row2[0][i]), .b(c_row2[0][i]),
.cin(s_row2[1][i]), .s(s_row3[0][i]), .cout(c_row3[0][i+1]));
        end
        // 3.1
        for (genvar i = 6; i < 22; i++) begin
            assign s_row3[1][i] = c_row2[1][i];
        end
        for (genvar i = 22; i < 132; i++) begin
            compressor_3to2 comp3_1_ (.a(c_row2[1][i]), .b(s_row2[2][i]),
.cin(c_row2[2][i]), .s(s_row3[1][i]), .cout(c_row3[1][i+1]));
        end
        // 3.2
        for (genvar i = 30; i < 46; i++) begin
            half_adder      comp3_2_ (.a(s_row2[3][i]), .b(c_row2[3][i]),
.s(s_row3[2][i]), .cout(c_row3[2][i+1]));
        end
        for (genvar i = 46; i < 132; i++) begin
            compressor_3to2 comp3_2_ (.a(s_row2[3][i]), .b(c_row2[3][i]),
.cin(s_row2[4][i]), .s(s_row3[2][i]), .cout(c_row3[2][i+1]));
        end
        // 3.3
        for (genvar i = 46; i < 52; i++) begin
            assign s_row3[3][i] = c_row2[4][i];
        end
        for (genvar i = 52; i < 132; i++) begin
            compressor_3to2 comp3_3_ (.a(c_row2[4][i]), .b(s_row2[5][i]),
.cin(c_row2[5][i]), .s(s_row3[3][i]), .cout(c_row3[3][i+1]));
        end

    // level 4
    // 8p split by (44) -> 4p
    // totle: 2
    wire [132:0] s_row4  [1:0];
    wire [132:0] c_row4  [1:0];
    wire [132:0] co_row4  [1:0];
        // 4.0
        for (genvar i = 0; i < 6; i++) begin
            half_adder      comp4_0_ (.a(s_row3[0][i]), .b(c_row3[0][i]),
.s(s_row4[0][i]), .cout(c_row4[0][i+1]));
        end
        for (genvar i = 6; i < 132; i++) begin
            compressor_4to2 comp4_0_ (.a(s_row3[0][i]), .b(c_row3[0][i]),
.c(s_row3[1][i]), .d(c_row3[1][i]), .cin(co_row4[0][i]), .s(s_row4[0][i]),
.co(c_row4[0][i+1]), .cout(co_row4[0][i+1]));
        end
        // 4.1
        for (genvar i = 30; i < 46; i++) begin
            half_adder      comp4_1_ (.a(s_row3[2][i]), .b(c_row3[2][i]),
.s(s_row4[1][i]), .cout(c_row4[1][i+1]));
        end
        for (genvar i = 46; i < 132; i++) begin
```

```
            compressor_4to2 comp4_0_ (.a(s_row3[2][i]), .b(c_row3[2][i]),
.c(s_row3[3][i]), .d(c_row3[3][i]), .cin(co_row4[1][i]), .s(s_row4[1][i]),
.co(c_row4[1][i+1]), .cout(co_row4[1][i+1]));
        end

    // level 5
    // 4p -> 2p
    // total: 1
    wire [132:0] s_row5;
    wire [132:0] c_row5;
    wire [132:0] co_row5;
        for (genvar i = 0; i < 30; i++) begin
            half_adder      comp5_ (.a(s_row4[0][i]), .b(c_row4[0][i]),
.s(s_row5[i]), .cout(c_row5[i+1]));
        end
        for (genvar i = 30; i < 132; i++) begin
            compressor_4to2 comp5_ (.a(s_row4[0][i]), .b(c_row4[0][i]),
.c(s_row4[1][i]), .d(c_row4[1][i]), .cin(co_row5[i]), .s(s_row5[i]),
.co(c_row5[i+1]), .cout(co_row5[i+1]));
        end

    // result
    assign out[0] = s_row5[127:0];
    assign out[1] = c_row5[127:0];
endmodule;
```

3:2 压缩器就是常规的全加器，而 4:2 压缩器其实是 5:3 压缩，优化布线后约为三个异或门的延迟：

```
module half_adder (
    input a, b,
    output s, cout
);
    assign cout = a & b;
    assign s    = a ^ b;
endmodule;

module compressor_3to2 (
    input  a, b, cin,
    output s, cout
);
    assign cout = (a & b) | (a & cin) | (b & cin);
    assign s    = a ^ b ^ cin;
endmodule;

module compressor_4to2 (
    input  a, b, c, d, cin,
    output s, co, cout
);
    wire tmp1 = (a & b) | (c & d);
    wire tmp2 = (a ^ b) ^ (c ^ d);

    assign cout = (a | b) & (c | d);
    assign co   = ~(~tmp1 | tmp2) | (tmp2 & cin);
    assign s    = tmp2 ^ cin;
endmodule;
```

而最终的两个 128 位加法其实就是做两次 64 位加法（考虑低 64 位的进位也要运算完才能给高位，故而即使设计 128 位高速加法其实也是两个 64 位加法作串行运算）

最后的乘法器如下所示：

```
`include "booth.v"
`include "wallace.v"
`include "add64.v"

module mul128 (
    input  [63:0] operand1, operand2,
    input         sign_x, sign_y,
    output [63:0] result_h, result_l
);
    logic [65 :0] x = sign_x ? {{(2){operand1[63]}}, operand1} : {2'b0, operand1};
    logic [65 :0] y = sign_y ? {{(2){operand1[63]}}, operand2} : {2'b0, operand2};
    logic [131:0] psum [32:0];
    logic [127:0] treeout [1:0];
    wire add_h64_c, add_l64_c;

    booth #(.INXY_W(66)) u_booth(.x(x), .y(y), .psum(psum));

    wallace_33 wallace (.in(psum), .out(treeout));

    add64 add_l64 (.operand1(treeout[0][63:0]), .operand2(treeout[1][63:0]),
.c0(1'b0), .result(result_l), .carry(add_l64_c));

    add64 add_h64 (.operand1(treeout[0][127:64]), .operand2(treeout[1][127:64]),
.c0(add_l64_c), .result(result_h), .carry(add_h64_c));
endmodule;
```

按照现在的大部分处理器的设计，恰好 booth 两位乘为乘法运算第一个周期，华莱士树为运算第二个周期，两次 64 位加法为三四周期，共计 4 周期完成预期的流水线设计。