软工计原联合项目

测试文档

NonExist 组
张钰晖，杨一滨，周正平

Contents

Chapter 1

文档说明

本文档是 NonExist 组软工计原联合项目的测试文档，主要描述了在实现 CPU 的过程中我们进行了哪些部分的测试。

测试文档将项目分成了以下部分：

1. **单指令测试**：在仅有流水线雏形时进行的单指令简单测试。
2. **功能测例**：自带 91 条测例，以及为 TLB 实现的 2 条测例。
3. **监控程序**：32 位 MIPS 监控程序。
4. **uCore 操作系统**：uCore 操作系统。

测试文档每个章节遵从以下介绍流程：

1. **简介概述**：详细介绍测试的功能。
2. **测试范畴**：测试能够覆盖的 CPU 部件，以及测试的要点。
3. **测试方法**：如何进行测试。
4. **测试结果**：测试结果是否正确。

希望本文档能给读者带来裨益。

Chapter 2

功能测例

2.1 简介概述

功能测例由汇编语言实现，主要用于测试 CPU 指令实现是否正确。

功能测例涵盖了 91 项测试，其中根据我们 CPU 最终完成情况，75 项是可测测例。

在此基础上，我们又增添了针对 TLB 操作指令 TLBWI 和 TLBWR 两条指令的测例，故总计 77 项测例。

功能测例由 MIPS 32 汇编语言编写。以下以第 1 个测试点 ADD 的流程为例，说明其程序结构。

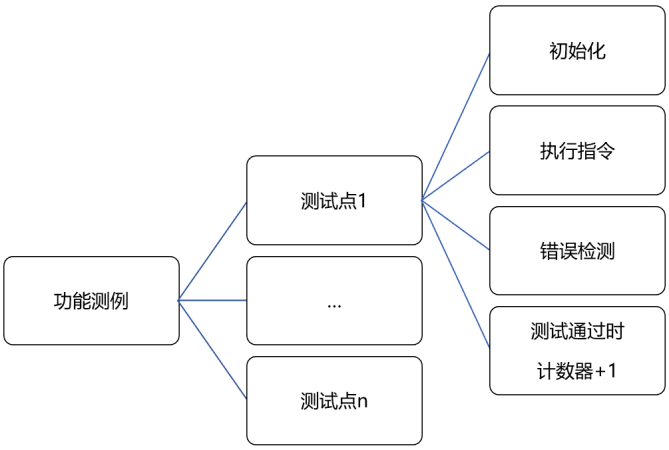

Figure 2.1: 功能测例程序框架

1. **主程序**：整个测试程序连续执行若干测试点，每个测试点结束后停顿 1s。

   例如，以下为主程序 start.S 中调用测试点 ADD 的代码片段：

```
inst_test:
    jal n1_add_test    #add
    nop
    jal wait_1s
    nop
```

2. **测试点**：每个测试点对应于一条指令。

   例如，n1_add_test 测试点的代码片段如下：

```
9   LEAF(n1_add_test)
10      lui a0, 0x100
11      li v0, 0x0
12  ###test inst
13      TEST_ADD(0x0480ff04, 0x40933204, 0x45143108)    # num1, num2 both random
14      TEST_ADD(0x2a19dd40, 0xa87971e0, 0xd2934f20)
...
211     TEST_ADD(0x25e5fad8, 0x00000000, 0x25e5fad8)    # num1 == 0 or num2 == 0
212     TEST_ADD(0x00000000, 0xdcaf5e62, 0xdcaf5e62)
...
262     TEST_ADD(0x00000000, 0x00000000, 0x00000000)    # num1 == 0 and num2 == 0
263 ###detect exception
264     bne v0, zero, inst_error
265     nop
266 ###score ++
267     addiu s3, s3, 1
268 ###output a0|s3
269 inst_error:
270     or t0, a0, s3
271     sw t0, 0(s1)
272     jr ra
273     nop
274 END(n1_add_test)
```

Listing 2.1: inst/n1_add.S

初始化　其中，第 10 行 a0 寄存器存储了测试功能点编号值，这里为 0x1；第 11 行 v0 寄存器存储的是例外检测，初始值为 0，如果程序出现例外会置为 0xFFFF0000。

指令执行　第 12 行到第 262 行为测试指令，分别测试了加数 1 和 2 都是随机数、加数 1 或 2 是 0、加数 1 和 2 都是 0 的情况。其中，宏 TEST_ADD 调用宏 ADD 执行指令，并判断目的寄存器 s0 的值是否与正确结果 s2 寄存器的值一致；宏 ADD 将加数 1 装载于寄存器 t0，加数 2 装载于寄存器 t1，加法结果位于寄存器 s0：

```
#define TEST_ADD(vs, vt, vd) \
    ADD(vs, vt); \
    li s2, vd; \
    bne s0, s2, inst_error; \
    nop
```

```
// ADD
#define ADD(v0, v1) \
    li t0, v0; \
    li t1, v1; \
    add s0, t0, t1
```

错误检测　第 263-265 行通过寄存器 v0 判断测试过程中是否出现例外，正确情况下没有例外；如出现例外则跳转到 inst_error。

测试通过时计数器 +1 第 266-267 行，s3 寄存器存测试分值，测试通过一个功能点加 1 分；第 268-273 行为测试结束，将测试编号和测试分值输出显示并返回主程序。

可以看出，功能测例对指令集有着较高的覆盖率、较全面鲁棒的测试数据、较清晰的代码结构，并且可以自动验证执行的正确性，帮助开发者大量减少了自行编写测例的时间。

2.2 测试范畴

2.2.1 测试覆盖面

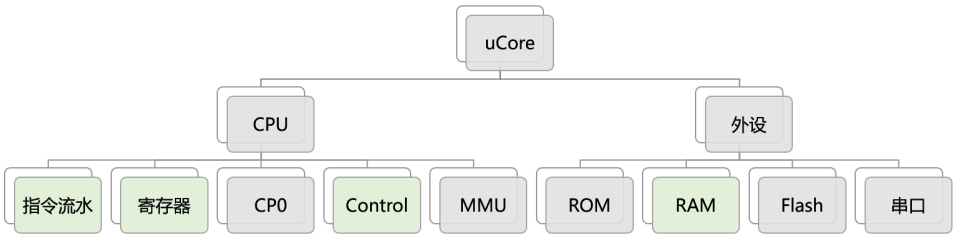

Figure 2.2: 功能测例测试范畴（绿 -可以测试；灰 -不能测试）

功能测例主要测试了 CPU 以下部件是否实现正确：

1. **五级流水线**：包括 IF、ID、EX、MEM、WB 共 5 个模块。
2. **寄存器**：包括寄存器堆（32 个通用寄存器）、HI/LO 寄存器共 2 个模块。
3. **Control**：包括 Control（流水线控制器）共 1 个模块。
4. **RAM**：包括 RAM（用作内存）共 1 个外部设备。

功能测例不能全面测试的 CPU 部件包括：

1. **CP0**：功能测例不能测试 TLB MISS、中断等异常，故不能全面测试 CP0。
2. **MMU**：功能测例不需要 TLB 进行地址映射，故不能全面测试 MMU。
3. **ROM**：功能测例不需要 BootLoader 进行引导，故不能测试 ROM。
4. **Flash**：功能测例烧录在 BaseRAM 中，不需要存入硬盘，故不能测试 Flash。
5. **串口**：功能测例不需要标准输入/输出，故不能测试串口。

2.2.2 测试要点

功能测例的测试要点包括如下几个部分：

1. **算术运算指令**：共 12 条，测试加、减、乘等指令。
2. **逻辑运算指令**：共 14 条，测试与、或、移位等指令。

3. **分支跳转指令**：共 10 条，测试 B/J 型指令。

4. **HI/LO 寄存器指令**：共 4 条，测试对 HI/LO 寄存器的读写。

5. **CP0 相关指令**：共 4 条，包括 SYSCALL、ERET、读/写 CP0 等指令。这一部分也测试部分异常处理行为。

6. **访存指令**：共 8 条，包括 L/S 型指令。

7. **异常指令**：共 13 条，其中的指令会触发整型溢出异常、地址未对齐异常、发生于延迟槽的异常等。

8. **延迟槽指令**：共 10 条，其中的被测指令位于延迟槽中。

此外，笔者新增了 2 条针对 TLB 的功能测例：

1. **TLBWI**：测试指令 TLBWI。测试文件如下：

```
LEAF(n92_tlbwi_test)
    .set noreorder
    lui  a0, 0x100
    li  v0, 0x0
###test inst
    TEST_TLBWI(0x0, 0x4002, 0x4042, 0x0, 0x61754443, 0x000007a0)
    TEST_TLBWI(0x0, 0x4002, 0x4042, 0x1, 0x5c4fb45a, 0x00000aac)
    TEST_TLBWI(0x0, 0x4002, 0x4042, 0x2, 0x14908300, 0x00000ae8)
    TEST_TLBWI(0x0, 0x4002, 0x4042, 0x3, 0x516da739, 0x000000cc)
    TEST_TLBWI(0x0, 0x4002, 0x4042, 0x4, 0x85675a34, 0x00000510)
    TEST_TLBWI(0x0, 0x4002, 0x4042, 0x5, 0x0e4dac98, 0x00000040)
    TEST_TLBWI(0x0, 0x4002, 0x4042, 0x6, 0xd9c6eddb, 0x00000180)
    TEST_TLBWI(0x0, 0x4002, 0x4042, 0x7, 0x5753dd01, 0x00000ca0)
    TEST_TLBWI(0x0, 0x4002, 0x4042, 0x8, 0xe543b9f3, 0x0000031c)
    TEST_TLBWI(0x0, 0x4002, 0x4042, 0x9, 0x4726aca2, 0x00000cf8)
    TEST_TLBWI(0x0, 0x4002, 0x4042, 0xa, 0xb022040a, 0x00000800)
    TEST_TLBWI(0x0, 0x4002, 0x4042, 0xb, 0x5ca0fd00, 0x00000834)
    TEST_TLBWI(0x0, 0x4002, 0x4042, 0xc, 0x063ba000, 0x00000c64)
    TEST_TLBWI(0x0, 0x4002, 0x4042, 0xd, 0xc2268cfe, 0x000001e8)
    TEST_TLBWI(0x0, 0x4002, 0x4042, 0xe, 0x1611444c, 0x00000484)
    TEST_TLBWI(0x0, 0x4002, 0x4042, 0xf, 0x33cc6f2a, 0x000001dc)
###detect exception
    bne v0, zero, inst_error
    nop
###score ++
    addiu s3, s3, 1
###output a0|s3
inst_error:
    or t0, a0, s3
    sw t0, 0(s1)
    jr ra
    nop
END(n92_tlbwi_test)
```

Listing 2.2: n92_tlbwi.S

<TODO>: Please Check

宏 TEST_TLBWI 会依次将 CP0 的 Index 寄存器赋值为 0x0, 0x1, ..., 0xF，并依次调用 TLBWI 指令写 TLB 表项。最终，通过先向基址处写入数据再读出，判断其是否与原数据相等，作为测试通过的依据：

```
/* 92
 * TEST_TLBWI(ENTRYHI, ENTRYLO0, ENTRYLO1, INDEX, data, base_addr)
 */
#define TEST_TLBWI(entryhi, entrylo0, entrylo1, index, data, base_addr) \
    li s2, 0; \
    mtc0 s2, c0_entrylo0; \
    mtc0 s2, c0_entrylo1; \
    mtc0 s2, c0_entryhi; \
    li s2, 0; \
    mtc0 s2, c0_index; \
    tlbwi; \
    li s2, 1; \
    mtc0 s2, c0_index; \
    tlbwi; \
    li s2, 2; \
    mtc0 s2, c0_index; \
    tlbwi; \
    li s2, 3; \
    mtc0 s2, c0_index; \
    tlbwi; \
    li s2, 4; \
    mtc0 s2, c0_index; \
    tlbwi; \
    li s2, 5; \
    mtc0 s2, c0_index; \
    tlbwi; \
    li s2, 6; \
    mtc0 s2, c0_index; \
    tlbwi; \
    li s2, 7; \
    mtc0 s2, c0_index; \
    tlbwi; \
    li s2, 8; \
    mtc0 s2, c0_index; \
    tlbwi; \
    li s2, 9; \
    mtc0 s2, c0_index; \
    tlbwi; \
    li s2, 10; \
    mtc0 s2, c0_index; \
    tlbwi; \
    li s2, 11; \
    mtc0 s2, c0_index; \
    tlbwi; \
    li s2, 12; \
    mtc0 s2, c0_index; \
    tlbwi; \
```

```
    li s2, 13; \
    mtc0 s2, c0_index; \
    tlbwi; \
    li s2, 14; \
    mtc0 s2, c0_index; \
    tlbwi; \
    li s2, 15; \
    mtc0 s2, c0_index; \
    tlbwi; \
    li s2, entryhi; \
    mtc0 s2, c0_entryhi; \
    li s2, entrylo0; \
    mtc0 s2, c0_entrylo0; \
    li s2, entrylo1; \
    mtc0 s2, c0_entrylo1; \
    li s2, index; \
    mtc0 s2, c0_index; \
    tlbwi; \
    li t1, data; \
    li t0, base_addr; \
    sw t1, 0x0(t0); \
    lw s0, 0x0(t0); \
    li s2, data; \
    bne s0, s2, inst_error; \
    nop
```

Listing 2.3: inst_test.h

2. **TLBWR**：测试指令 TLBWR。测试文件如下：

```
LEAF(n93_tlbwr_test)
    .set noreorder
    lui  a0, 0x100
    li  v0, 0x0
###test inst
    TEST_TLBWR(0x0, 0x4002, 0x4042, 0x61754443, 0x000007a0)
    TEST_TLBWR(0x0, 0x4002, 0x4042, 0x5c4fb45a, 0x00000aac)
    TEST_TLBWR(0x0, 0x4002, 0x4042, 0x14908300, 0x00000ae8)
    TEST_TLBWR(0x0, 0x4002, 0x4042, 0x516da739, 0x000000cc)
    TEST_TLBWR(0x0, 0x4002, 0x4042, 0x85675a34, 0x00000510)
    TEST_TLBWR(0x0, 0x4002, 0x4042, 0x0e4dac98, 0x00000040)
    TEST_TLBWR(0x0, 0x4002, 0x4042, 0xd9c6eddb, 0x00000180)
    TEST_TLBWR(0x0, 0x4002, 0x4042, 0x5753dd01, 0x00000ca0)
    TEST_TLBWR(0x0, 0x4002, 0x4042, 0xe543b9f3, 0x0000031c)
    TEST_TLBWR(0x0, 0x4002, 0x4042, 0x4726aca2, 0x00000cf8)
    TEST_TLBWR(0x0, 0x4002, 0x4042, 0xb022040a, 0x00000800)
    TEST_TLBWR(0x0, 0x4002, 0x4042, 0x5ca0fd00, 0x00000834)
    TEST_TLBWR(0x0, 0x4002, 0x4042, 0x063ba000, 0x00000c64)
    TEST_TLBWR(0x0, 0x4002, 0x4042, 0xc2268cfe, 0x000001e8)
    TEST_TLBWR(0x0, 0x4002, 0x4042, 0x1611444c, 0x00000484)
    TEST_TLBWR(0x0, 0x4002, 0x4042, 0x33cc6f2a, 0x000001dc)
###detect exception
    bne v0, zero, inst_error
```

```
    nop
###score ++
    addiu s3, s3, 1
###output a0|s3
inst_error:
    or t0, a0, s3
    sw t0, 0(s1)
    jr ra
    nop
END(n93_tlbwr_test)
```

Listing 2.4: n93_tlbwr.S

<TODO>: Please Check

宏 TEST_TLBWR 会首先调用 TLBWI 指令共 15 次以构造所有 TLB 表项；然后调用 TLBWR 指令写入其中一个表项。最终，通过先向基址处写入数据再读出，判断其是否与原数据相等，作为测试通过的依据：

```
/* 93
 * TEST_TLBWR(ENTRYHI, ENTRYLO0, ENTRYLO1, data, base_addr)
 */
#define TEST_TLBWR(entryhi, entrylo0, entrylo1, data, base_addr) \
    li s2, 0; \
    mtc0 s2, c0_entrylo0; \
    mtc0 s2, c0_entrylo1; \
    mtc0 s2, c0_entryhi; \
    li s2, 0; \
    mtc0 s2, c0_index; \
    tlbwi; \
    li s2, 1; \
    mtc0 s2, c0_index; \
    tlbwi; \
    li s2, 2; \
    mtc0 s2, c0_index; \
    tlbwi; \
    li s2, 3; \
    mtc0 s2, c0_index; \
    tlbwi; \
    li s2, 4; \
    mtc0 s2, c0_index; \
    tlbwi; \
    li s2, 5; \
    mtc0 s2, c0_index; \
    tlbwi; \
    li s2, 6; \
    mtc0 s2, c0_index; \
    tlbwi; \
    li s2, 7; \
    mtc0 s2, c0_index; \
    tlbwi; \
    li s2, 8; \
    mtc0 s2, c0_index; \
```

```
tlbwi; \
li s2, 9; \
mtc0 s2, c0_index; \
tlbwi; \
li s2, 10; \
mtc0 s2, c0_index; \
tlbwi; \
li s2, 11; \
mtc0 s2, c0_index; \
tlbwi; \
li s2, 12; \
mtc0 s2, c0_index; \
tlbwi; \
li s2, 13; \
mtc0 s2, c0_index; \
tlbwi; \
li s2, 14; \
mtc0 s2, c0_index; \
tlbwi; \
li s2, 15; \
mtc0 s2, c0_index; \
tlbwi; \
li s2, entryhi; \
mtc0 s2, c0_entryhi; \
li s2, entrylo0; \
mtc0 s2, c0_entrylo0; \
li s2, entrylo1; \
mtc0 s2, c0_entrylo1; \
tlbwr; \
li t1, data; \
li t0, base_addr; \
sw t1, 0x0(t0); \
lw s0, 0x0(t0); \
li s2, data; \
bne s0, s2, inst_error; \
nop
```

2.3 测试方式

2.3.1 仿真阶段

仿真阶段的主要目的在于首先借助 EDA 工具（Vivado）方便的仿真调试功能，找出显而易见、较为简单的 bug，以减轻接下来在硬件上测试调试的负担。这一阶段仅测试 CPU 位于 FPGA 内部的核心逻辑是否正确，而所有外围设备（包括 RAM）均不能测试，需使用自己模拟实现的 RAM 模块。

模拟的 RAM 模块需首先将功能测例编译后的二进制文件转换成 Verilog 可接受的 ASC-II 文本文件，而后在 initial 语句中导入该文件，将 CPU 指令计数器置为 0x80000000，开始仿真运行。

```
initial $readmemh ("ram.data", data_ram);
```

Listing 2.5: ram.v 的 initial 块

通过阅读功能测例的代码我们可知，19 号寄存器的数值存放了功能测例的通过条数。因此，只需在仿真中观察该寄存器的值是否随着每个测试点的结束自增 1 即可。

2.3.2 硬件阶段

硬件阶段的主要目的在于在真实开发板上运行功能测例，加入对 RAM 部分的测试，并找出 CPU 可能存在的时序漏洞。

在这一阶段，需将功能测例定义的七段数码管地址通过 MMU 映射至开发板的七段数码管，将编译后的功能测例烧入 BaseRAM 中，则七段数码管显示的即为测试点的通过个数。

```
#define LED_ADDR  0xbfd0f000    // LED virtual address
#define NUM_ADDR  0xbfd0f010    // 7-segments digital display virtual address
```

Listing 2.6: start.S 中定义的 LED 与七段数码管地址

2.4 测试结果

77 条功能测例全部通过，见以下表格（其中硬件测试时钟频率为 25MHz，“P”-通过，“N”-无需实现）。

序号	测试程序	功能测试点	仿真测试	硬件测试
1	ADD	执行 ADD 指令是否产生正确的运算结果（未测试整型溢出例外的情况）	P	P
2	ADDI	执行 ADDI 指令是否产生正确的运算结果（未测试整型溢出例外的情况）	P	P
3	ADDU	执行 ADDU 指令是否产生正确的运算结果	P	P
4	ADDIU	执行 ADDIU 指令是否产生正确的运算结果	P	P
5	SUB	执行 SUB 指令是否产生正确的运算结果（未测试整型溢出例外的情况）	P	P
6	SUBU	执行 SUBU 指令是否产生正确的运算结果	P	P
7	SLT	执行 SLT 指令是否产生正确的运算结果	P	P
8	SLTI	执行 SLTI 指令是否产生正确的运算结果	P	P
9	SLTU	执行 SLTU 指令是否产生正确的运算结果	P	P
10	SLTIU	执行 SLTIU 指令是否产生正确的运算结果	P	P
11	DIV	执行 DIV 指令是否产生正确的运算结果	N	N
12	DIVU	执行 DIVU 指令是否产生正确的运算结果	N	N
13	MULT	执行 MULT 指令是否产生正确的运算结果	P	P
14	MULTU	执行 MULTU 指令是否产生正确的运算结果	P	P

Table 2.1: 算术运算指令（共 12(0xC) 条）

序号	测试程序	功能测试点	仿真测试	硬件测试
15	AND	执行 AND 指令是否产生正确的运算结果	P	P

16	ANDI	执行 ANDI 指令是否产生正确的运算结果	P	P
17	LUI	执行 LUI 指令是否产生正确的运算结果	P	P
18	NOR	执行 NOR 指令是否产生正确的运算结果	P	P
19	OR	执行 OR 指令是否产生正确的运算结果	P	P
20	ORI	执行 ORI 指令是否产生正确的运算结果	P	P
21	XOR	执行 XOR 指令是否产生正确的运算结果	P	P
22	XORI	执行 XORI 指令是否产生正确的运算结果	P	P
23	SLLV	执行 SLLV 指令是否产生正确的移位结果	P	P
24	SLL	执行 SLL 指令是否产生正确的移位结果	P	P
25	SRAV	执行 SRAV 指令是否产生正确的移位结果	P	P
26	SRA	执行 SRA 指令是否产生正确的移位结果	P	P
27	SRLV	执行 SRLV 指令是否产生正确的移位结果	P	P
28	SRL	执行 SRL 指令是否产生正确的移位结果	P	P

Table 2.2: 逻辑运算指令（共 14(0xE) 条）

序号	测试程序	功能测试点	仿真测试	硬件测试
29	BEQ	执行 BEQ 指令是否产生正确的判断和跳转结果（延迟槽指令为 nop，未测试延迟槽）	P	P
30	BNE	执行 BNE 指令是否产生正确的判断和跳转结果（延迟槽指令为 nop，未测试延迟槽）	P	P
31	BGEZ	执行 BGEZ 指令是否产生正确的判断和跳转结果（延迟槽指令为 nop，未测试延迟槽）	P	P
32	BGTZ	执行 BGTZ 指令是否产生正确的判断和跳转结果（延迟槽指令为 nop，未测试延迟槽）	P	P
33	BLEZ	执行 BLEZ 指令是否产生正确的判断和跳转结果（延迟槽指令为 nop，未测试延迟槽）	P	P
34	BLTZ	执行 BLTZ 指令是否产生正确的判断和跳转结果（延迟槽指令为 nop，未测试延迟槽）	P	P
35	BGEZAL	执行 BGEZAL 指令是否产生正确的判断、跳转和链接结果（延迟槽指令为 nop，未测试延迟槽）	N	N
36	BLTZAL	执行 BLTZAL 指令是否产生正确的判断、跳转和链接结果（延迟槽指令为 nop，未测试延迟槽）	N	N
37	J	执行 J 指令是否产生正确的跳转结果（延迟槽指令为 nop，未测试延迟槽）	P	P
38	JAL	执行 JAL 指令是否产生正确的跳转和链接结果（延迟槽指令为 nop，未测试延迟槽）	P	P
39	JR	执行 JR 指令是否产生正确的跳转结果（延迟槽指令为 nop，未测试延迟槽）	P	P
40	JALR	执行 JALR 指令是否产生正确的跳转和链接结果（延迟槽指令为 nop，未测试延迟槽）	P	P

Table 2.3: 分支跳转指令（共 10(0xA) 条）

序号	测试程序	功能测试点	仿真测试	硬件测试
41	MFHI	执行 MTHI 指令是否正确地将寄存器值写入 HI 寄存器，执行 MFHI 指令是否正确地读出 HI 寄存器的值到寄存器	P	P
42	MFLO	执行 MTLO 指令是否正确地将寄存器值写入 LO 寄存器，执行 MFLO 指令是否正确地读出 LO 寄存器的值到寄存器	P	P
43	MTHI	执行 MTHI 指令是否正确地将寄存器值写入 HI 寄存器，执行 MFHI 指令是否正确地读出 HI 寄存器的值到寄存器	P	P
44	MTLO	执行 MTLO 指令是否正确地将寄存器值写入 HI 寄存器，执行 MFLO 指令是否正确地读出 HI 寄存器的值到寄存器	P	P

Table 2.4: HILO 寄存器指令（共 4(0x4) 条）

序号	测试程序	功能测试点	仿真测试	硬件测试
45	BREAK	执行 BREAK 指令是否正确地产生断点例外	N	N
46	SYSCALL	执行 SYSCALL 指令是否正确地产生系统调用例外	P	P
55	ERET	执行 ERET 指令是否正确地从中断、例外处理程序返回	P	P
56	MFC0	执行 MTC0 指令是否正确地将寄存器值写入目的 CP0 寄存器，执行 MFC0 指令是否正确地读出源 CP0 寄存器的值到寄存器	P	P
57	MTC0	执行 MTC0 指令是否正确地将寄存器值写入目的 CP0 寄存器，执行 MFC0 指令是否正确地读出源 CP0 寄存器的值到寄存器	P	P

Table 2.5: CP0 指令（共 4(0x4) 条）

序号	测试程序	功能测试点	仿真测试	硬件测试
47	LB	结合 SW 指令，执行 LB 指令是否产生正确的访存结果	P	P
48	LBU	结合 SW 指令，执行 LBU 指令是否产生正确的访存结果	P	P

49	LH	结合 SW 指令，执行 LH 指令是否产生正确的访存结果	P	P
50	LHU	结合 SW 指令，执行 LHU 指令是否产生正确的访存结果	P	P
51	LW	结合 SW 指令，执行 LW 指令是否产生正确的访存结果	P	P
52	SB	结合 LW 指令，执行 SB 指令是否产生正确的访存结果	P	P
53	SH	结合 LW 指令，执行 SH 指令是否产生正确的访存结果	P	P
54	SW	结合 LW 指令，执行 SW 指令是否产生正确的访存结果	P	P

Table 2.6: 访存指令（共 8(0x8) 条）

序号	测试程序	功能测试点	仿真测试	硬件测试
58	ADD_EX	测试 ADD 指令整型溢出例外	P	P
59	ADDI_EX	测试 ADDI 指令整型溢出例外	P	P
60	SUB_EX	测试 SUB 指令整型溢出例外	P	P
61	LH_EX	测试 LH 指令访存地址非对齐例外	N	N
62	LHU_EX	测试 LHU 指令访存地址非对齐例外	N	N
63	LW_EX	测试 LW 指令访存地址非对齐例外	N	N
64	SH_EX	测试 SH 指令访存地址非对齐例外	N	N
65	SW_EX	测试 SW 指令访存地址非对齐例外	N	N
66	ERET_EX	测试取指地址非对齐例外	N	N
67	RESERVED_-INSTRUCTION_EX	测试保留指令例外	N	N
80	BEQ_EX_DS	测试延迟槽例外	P	P
81	BNE_EX_DS	测试延迟槽例外	P	P
82	BGEZ_EX_DS	测试延迟槽例外	P	P
83	BGTZ_EX_DS	测试延迟槽例外	P	P
84	BLEZ_EX_DS	测试延迟槽例外	P	P
85	BLTZ_EX_DS	测试延迟槽例外	P	P
86	BGEZAL_EX_DS	测试延迟槽例外	N	N
87	BLTZAL_EX_DS	测试延迟槽例外	N	N
88	J_EX_DS	测试延迟槽例外	P	P
89	JAL_EX_DS	测试延迟槽例外	P	P
90	JR_EX_DS	测试延迟槽例外	P	P
91	JALR_EX_DS	测试延迟槽例外	P	P

Table 2.7: 异常指令（共 13(0xD) 条）

序号	测试程序	功能测试点	仿真测试	硬件测试
68	BEQ_DS	测试延迟槽	P	P
69	BNE_DS	测试延迟槽	P	P
70	BGEZ_DS	测试延迟槽	P	P
71	BGTZ_DS	测试延迟槽	P	P
72	BLEZ_DS	测试延迟槽	P	P
73	BLTZ_DS	测试延迟槽	P	P
74	BGEZAL_DS	测试延迟槽	N	N
75	BLTZAL_DS	测试延迟槽	N	N
76	J_DS	测试延迟槽	P	P
77	JAL_DS	测试延迟槽	P	P
78	JR_DS	测试延迟槽	P	P
79	JALR_DS	测试延迟槽	P	P

Table 2.8: 延迟槽指令（共 10(0xA) 条）

2.5 问题与解决

在测试中我们发现，功能测例会通过 load 指令写入自身的代码区，导致自身逻辑遭到破坏。

例如，下面的第一条测试 LB 指令的代码，会将数据写入 VA (0x800244e8 + 0x00002174) = 0x8002665C 处：

```
...
# *(0x800244e8 + 0x00002174) = 0xe83814f0
TEST_LB(0xe83814f0, 0x800244e8, 0x00002174, 0x00002174, 0xffffffff0)
TEST_LB(0xbb62f9ba, 0x80021408, 0x00003c40, 0x00003c40, 0xffffffba)
TEST_LB(0x9eb52b80, 0x8002d46c, 0x000002ae, 0x000002ac, 0xffffffb5)
...
```

Listing 2.7: n47_lb.S 中会破坏自身程序的访存指令

然而，由于功能测例的初始地址在 VA 0x80000000 处，当测试范围为整个功能测例的所有代码时，会导致该地址处实际属于功能测例自身的代码段：

```
...
80026654:   10000005    b   8002666c <n40_jalr_test+0x954c>
80026658:   00000000    nop
8002665c:   00000021    move    zero,zero
80026660:   00000021    move    zero,zero
80026664:   3c12d65a    lui s2,0xd65a
...
```

Listing 2.8: 功能测例在 objdump 后的代码片段

从而，要想通过全部功能测例，有以下 2 种解决方案：

1. **使用 2 块 RAM**：同时使用 BaseRAM 和 ExtRAM，分别用于存储指令和数据。仿真测试显示这样可以通过所有功能测例而不会触发上述 bug。

2. **分段测试：**将功能测例逐段进行测试（其余部分注释掉），这样功能测例代码量减少，不会蔓延到被写入的内存区域。最终笔者得以用这样的方式通过全部功能测例。

Chapter 3

32 位监控程序

3.1 简介概述

32 位监控程序[1]相当于一个微型操作系统，其内核使用 MIPS 32 汇编语言编写（位于 monitor/kern/目录），客户端终端应用程序使用 Python 语言编写（term/term.py 文件）。

监控程序主要用于小范围的系统测试，将指令集中的 23 条指令综合使用，以确保其作为一个整体的正确性。

相比功能测例，监控程序增加了对 BootLoader、中断以及串口的需求，因此对外设的测试覆盖面大大扩展。此外，它还支持 TLB 的测试，但需要增加 2 条 TLB 指令（TLBP、TLBR）。我们因此没有采用监控程序进行 TLB 的测试。

监控程序的主要流程如下：

1. **Boot 阶段**：硬件初始化时置 PC 值为 VA 0xBFC00000，读取 ROM 中存储的 BootLoader，将 Flash 中存储的监控程序拷贝至 RAM。拷贝结束后跳转至启动入口 VA 0x80000000。

2. **系统初始化**：系统初始化阶段暂停中断处理，而后对 CP0 的 Status、Cause、Ebase 寄存器进行初始化，最后进入正常中断模式。

3. **启动完毕**：启动 shell 主线程，向串口打印启动信息。

   用户在终端可以执行以下命令来验证监控程序正确运行（其中的参数均为 32 位，使用小端序）：

   - **R**：`R`
     按照 $1 至 $30 的顺序返回用户空间寄存器值。
   - **D**：`D <addr> <num>`
     按照小端序返回指定地址连续 num 字节。
   - **A**：`A <addr> <num> <content>`
     在指定地址写入 content。约定 content 有 num 字节，并且 num 为 4 的倍数。
   - **G**：`G <addr>`
     执行指定地址的用户程序。ra 传递正常退出地址。

[1]文档地址：https://wyl8899.gitbooks.io/road-to-neptunus/content/；
项目地址：https://git.net9.org/wyl8899/Neptunus

- **T**：`T <num>`
  查看 index=num 的 TLB 项，依次回传 ENTRYHI, ENTRYLO0, ENTRYLO1 共 12 字节。

3.2 测试范畴

3.2.1 测试覆盖面

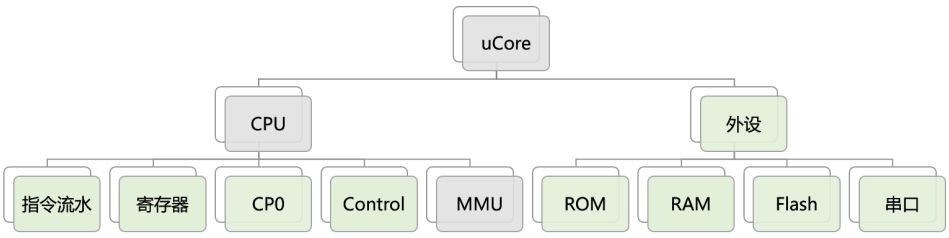

Figure 3.1: 32 位监控程序测试范畴（绿 -可以测试；灰 -不能测试）

32 位监控程序主要测试了 CPU 的以下部件是否实现正确：

1. **五级流水线**：包括 IF、ID、EX、MEM、WB 共 5 个模块。
2. **寄存器**：包括寄存器堆（32 个通用寄存器）、HI/LO 寄存器共 2 个模块。
3. **CP0**：包括 CP0 共 1 个模块。
4. **Control**：包括 Control（流水线控制器）共 1 个模块。
5. **ROM**：包括 ROM（存储 BootLoader）共 1 个外部设备。
6. **RAM**：包括 RAM（用作内存）共 1 个外部设备。
7. **Flash**：包括 Flash（存储监控程序）共 1 个外部设备。
8. **串口**：包括串口（用于与用户终端交互）共 1 个外部设备。

32 位监控程序不能全面测试的 CPU 部件包括：

1. **MMU**：我们没有打开监控程序的 TLB 选项，故不能全面测试 MMU。

3.2.2 测试要点

除去功能测例已经较为完整地测试过了的指令集合之外，32 位监控程序的测试要点集中于访存（包括对外部设备的时序调度）与异常处理方面：

1. **ROM 与 BootLoader**：能正确地读取 ROM，并运行 BootLoader 完成引导过程。
2. **RAM**：能正确实现内存的读写操作。
3. **Flash**：能正确地读取 Flash 中存储的监控程序，以完成 BootLoader 阶段的拷贝任务。
4. **串口**：能正常地读写串口，以实现用户终端的标准输入、输出。
5. **异常处理**：能正确处理硬件中断（包括时钟中断、串口中断）、Syscall 异常。

3.3 测试方式

32 位监控程序的测试分为如下步骤：

1. **系统初始化**

   (a) 将 BootLoader 编译后写入 ROM 对应的 Verilog 文件中。

   (b) 将监控程序编译后烧写入 Flash 的地址 0x0 处。

   (c) 将 CPU 的.bit 设计文件烧写入开发板 FPGA 中。

   (d) 点击开发板上 Reset 开关启动系统。

2. **启动客户端（监控程序终端）**

   (a) 将开发板的 USB 接口连接至 PC 端，以建立串口通信。

   (b) 在 PC 端（Ubuntu 16.04 x64 系统）监控程序的 term/目录下运行 `python term.py <pipe_path>`即可启动用户终端。

   (c) 可以输入 R、D、A、G、T 共 5 种指令以验证监控程序已正确运行。

3.4 测试结果

<TODO>：截图
经测试，监控程序可以正常启动，但是运行结果与预期不符。经检测，这是因为其软件实现存在 bug。

3.5 问题与解决

<TODO>：建议简单描述一下这里出现的 bug
例如下面这些：

1. **Flash 地址总线的一个 bug：**http://47.94.142.165:8088/gitlab/PRJ11_NonExist/CPU/commit/920973a3a1a28f027a85e275d703c3eec8cce6df

2. **Control 模块中的异常处理入口设置：**http://47.94.142.165:8088/gitlab/PRJ11_NonExist/CPU/commit/35971df28621af7772590fdef9ffc4a80d20ba18

3. **增加 ExtRAM：**http://47.94.142.165:8088/gitlab/PRJ11_NonExist/CPU/commit/260859a2e6f35c

4. **串口接收端的一个 bug：**http://47.94.142.165:8088/gitlab/PRJ11_NonExist/CPU/commit/200a69e2bebbc24fb5d47f18f098c29b096652b9

Chapter 4

uCore 操作系统