計算機組織與 RISC-V CPU 設計 · 第 26 章

驗證與模擬: 組譯器、Testbench 與除錯實錄

Python 組譯器 / 自檢測試程式 / testbench 設計 / 波形與追蹤除錯 / 迴歸測試自動化

摘要

「沒有驗證過的設計等於不存在。」本章介紹本教材 CPU 的完整驗證流程: 先寫一個約 260 行的 Python 組譯器把 RV32I 組語轉成 `$readmemh` 十六進位檔; 再設計自檢 (self-checking) 測試程式——程式自己計算校驗和並把成功/失敗魔數寫進記憶體;testbench 只需輪詢該位址即可自動判定 PASS/FAIL。最後以一次真實的除錯過程為例 (管線版迴圈加總算錯), 示範如何用逐週期追蹤 testbench 定位到「距離 3 RAW 危障」的根因。全流程以 shell 腳本自動化, 一鍵完成組譯、雙版本模擬與結果比對。

目錄

1 驗證策略總覽

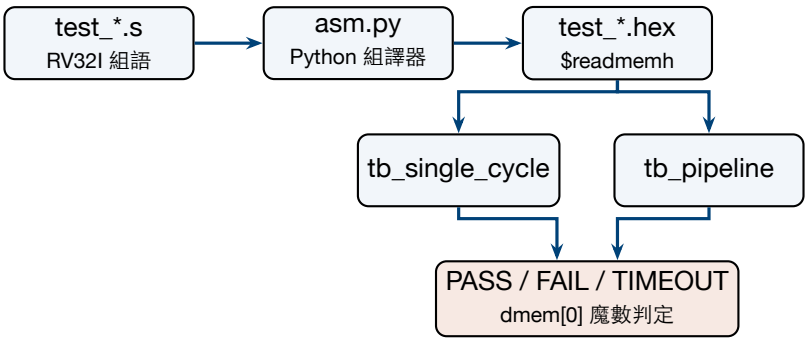

圖 1 驗證流程: 同一份程式跑兩種微架構, 結果必須一致

同一份測試程式在單週期與管線兩個實作上執行——兩者 ISA 相同、微架構不同, 結果必須完全一致。這正是第 1 章「架構 vs. 組織」的實證: 架構是合約, 組織是實作。

2 Python 組譯器 asm.py

工具鏈裡最實用的一環。支援 RV32I 全部 40 條指令、標籤與常用偽指令, 輸出每行一字的十六進位檔:

```
def enc_r(f7, rs2, rs1, f3, rd, op):
    return (f7<<25)|(rs2<<20)|(rs1<<15)|(f3<<12)|(rd<<7)|op

def enc_b(imm, rs2, rs1, f3, op):        # B 型立即值切片
    i = imm & 0x1FFF
    return ((i>>12&1)<<31)|((i>>5&0x3F)<<25)|(rs2<<20)|(rs1<<15)| \
           (f3<<12)|((i>>1&0xF)<<8)|((i>>11&1)<<7)|op
```

程式 1 asm.py: 編碼核心 (節錄)

兩遍掃描: 第一遍收集標籤位址, 第二遍展開偽指令並編碼。`li` 依常數大小展開成 `addi` 或 `lui+addi`(注意低 12 位為負時上半部要 +1 補償——第 18 章介紹過的細節, 組譯器裡必須真的寫對)。

3 自檢測試程式

```
# 慣例: x28 累積校驗和, 每步驟後與期望值比對
    addi x5, x0, 100
    addi x6, x0, -3
    add  x28, x5, x6      # x28 = 97
    addi x29, x0, 97      # 期望值
    bne  x28, x29, fail   # 不符立即跳 fail
    # ... 共 15 個檢查點: 算邏/移位/比較/載入儲存/分支/跳躍 ...
pass:
    lui  x30, 0xC0DE6     # 0xC0DE600D = 成功魔數
    addi x30, x30, 0x00D
    sw   x30, 0(x0)       # 寫入 dmem[0]
    j    pass             # 停在此處
fail:
```

```
    lui  x30, 0xDEADB     # 0xDEADBEEF = 失敗魔數
    ...
```

程式 2　test_basic.s 的自檢骨架

自檢設計要領

- 期望值在程式內硬編碼 (手算或用參考模擬器產生), 不依賴 testbench 逐條核對。
- 每個檢查點涵蓋一類指令行為: 符號延伸、無號比較、位元組/半字存取、taken/not-taken 分支……
- 失敗立刻分岔到 fail, 魔數還能編碼「錯在第幾個檢查點」(把檢查點編號寫進 dmem[4])。
- 測試程式對微架構零假設——才能同時跑單週期與管線版。

4　Testbench 設計

```
module tb_pipeline;
    reg clk = 0, rst_n = 0;
    always #5 clk = ~clk;                    // 100 MHz
    top_pipeline #(.HEXFILE(`HEX)) dut (.clk(clk), .rst_n(rst_n));

    integer i;
    initial begin
        $dumpfile("wave.vcd"); $dumpvars(0, tb_pipeline);
        repeat (4) @(posedge clk); rst_n = 1;
        for (i = 0; i < 20000; i = i + 1) begin
            @(posedge clk);
            if (dut.u_dmem.mem[0] == 32'hC0DE600D) begin
                $display("PASS (%0d cycles)", i); $finish;
            end
            if (dut.u_dmem.mem[0] == 32'hDEADBEEF) begin
                $display("FAIL checkpoint=%0d", dut.u_dmem.mem[1]);
                $fatal;
            end
        end
        $display("TIMEOUT"); $fatal;
    end
endmodule
```

程式 3　tb_pipeline.v(核心)

三個要點:(1) 魔數輪詢讓 testbench 與測試程式解耦;(2) 逾時保護——PC 跑飛或死鎖時不會無限模擬;(3) `$dumpvars` 產生 VCD 波形, 供 GTKWave 檢視。

5　除錯實錄: 一個真實的管線 bug

管線版首次模擬, 迴圈加總 $1+2+\cdots+10$ 應得 55, 實得 **1**。除錯過程:

1. 縮小範圍: 寫最小重現組語 `test_dbg.s`(只留迴圈), 確認錯誤仍在——排除其他指令干擾。

2. 逐週期追蹤: 專用 `tb_trace.v` 每週期印出五級的 PC、指令、前遞選擇與寫回值:

```
$display("C%0d IF=%h ID=%h EX=%h(a=%h b=%h) WB rd=%0d wd=%h",
         cyc, if_pc, ifid_instr, idex_pc, ex_a, ex_b,
         memwb_rd, wb_data);
```

3. 定位: 第 9 週期,`add x5,x5,x6` 在 ID 讀 x5 時, 前一次迴圈寫 x5 的指令恰在同週期 **WB** ——距離 3, 兩條前遞路徑都不涵蓋, 讀到舊值。

4. 修正:regfile 加入寫穿旁路 (第 25 章 5.3 節); 重跑全部測試確認通過, 且單週期版不受影響 (BYPASS=0)。

經驗法則

管線 bug 幾乎都表現為「某指令讀到舊值/錯值」。與其盯波形海撈, 先問: 這個值的生產者和消費者相距幾條指令? 當時各在哪一級?——把時空關係畫出來, 九成的前遞/停頓 bug 一眼可見。

6 迴歸自動化 run_tests.sh

```
for prog in sw/*.s; do
    python3 tools/asm.py "$prog" -o build/prog.hex
    for cpu in single_cycle pipeline; do
        iverilog -g2012 -DHEX='"build/prog.hex"' \
            -o build/sim tb/tb_$cpu.v rtl/*.v
        vvp build/sim > build/run.log 2>&1
        grep -q PASS build/run.log && echo "  $cpu: PASS" \
                                   || { echo "  $cpu: FAIL"; exit 1; }
    done
done
```

程式 4 run_tests.sh(骨架)

最終迴歸結果

測試程式	單週期	五級管線
test_basic.s(81 條, 全指令類別)	PASS(98 週期)	PASS(133 週期)
test_hazard.s(危障壓力)	PASS	PASS

任何 RTL 修改後重跑一次腳本,30 秒內知道有沒有弄壞東西——這就是迴歸測試的價值。

7 再往前走

這顆 CPU 是起點而非終點, 可依本教材前半部的理論逐步升級:

- **CSR** 與例外 (第 21 章): 加入 mstatus/mtvec/mepc, 支援 ecall 與非法指令陷阱。
- **M** 擴展 (第 20 章): 多週期乘除法器, 管線需加入 EX 級佔用停頓。
- 快取 (第 4 章): 直接映射 I-cache/D-cache, 體驗未命中停頓與寫回策略。

- 分支預測 (第 12 章):2 位元飽和計數器 +BTB, 把分支代價從 2 拍壓到接近 0。
- 正式驗證: 接上 riscv-formal 或跑官方 riscv-arch-test 合規測試。

8 本章重點回顧

本章要點

- 驗證流程: 組譯器 → 自檢程式 → 魔數輪詢 testbench → 迴歸腳本, 環環解耦。
- 自檢程式把「對不對」的判斷放進程式本身,testbench 保持極簡且與微架構無關。
- 除錯方法論: 最小重現 → 逐週期追蹤 → 分析生產者/消費者的時空距離 → 修正後全量迴歸。
- 同一程式在兩種微架構上結果一致, 實證了「ISA 是合約」。

9 練習

1. 為 asm.py 增加 `blt`/`bge` 的偽指令對偶 `bgt`/`ble`(交換運算元實作), 並寫測試驗證。
2. 擴充自檢慣例: 失敗時把檢查點編號寫入 dmem[4], 修改 testbench 印出「第幾關失敗」。
3. 寫一個測試程式專門驗證 `lb`/`lbu`/`lh`/`lhu` 對同一字的四種讀法, 期望值先手算。
4. 用 GTKWave 打開 wave.vcd, 找出 test_hazard.s 中載入使用停頓發生的週期, 截圖標注氣泡。
5. 估算: 若把 test_basic.s 換成 1000 條指令的隨機測試, 單週期與管線的週期數各約多少?(用第 25 章的停頓統計推算)

參考資料

[1] S. Williams, *Icarus Verilog*, `steveicarus.github.io/iverilog/`.

[2] C. Wolf, *riscv-formal: Formal Verification Framework for RISC-V*, GitHub.

[3] RISC-V International, *riscv-arch-test* 合規測試套件, GitHub.

[4] 本教材原始碼:`verilog/tools/asm.py`、`verilog/tb/`、`verilog/run_tests.sh`。