第 4 章实验 - 优化 Y86-64 流水线处理器性能

1120231863　左逸龙　07112303

1 实验环境

本实验在 Windows 10 操作系统下通过 WSL2 完成，具体环境如下：

- **操作系统**：Windows 10 (Build 19045) + WSL2
- **Linux 发行版**：Ubuntu 24.04.1 LTS
- **内核版本**：6.6.87.2-microsoft-standard-WSL2

2 Part A：Y86-64 汇编程序设计

2.1 任务概述

Part A 要求编写三个 Y86-64 汇编程序，分别实现链表迭代求和、链表递归求和、块复制与异或校验功能。以下分别介绍这三个程序的实现以及测试结果。

2.2 `sum.ys`：链表迭代求和

`sum_list` 函数采用迭代方式遍历链表，累加每个节点的值并返回总和。链表节点结构为 `{val, next}`，每个字段占 8 字节。核心实现如下：

```
sum_list:
    pushq %rbx              # 保存被调用者保存寄存器
    xorq %rax, %rax         # 初始化返回值 sum = 0
    jmp test                # 跳转到循环条件判断

loop:
    mrmovq (%rdi), %rbx     # 读取当前节点的 val
    addq %rbx, %rax         # sum += val
    mrmovq 8(%rdi), %rdi    # rdi = rdi->next

test:
    andq %rdi, %rdi         # 测试 rdi 是否为 NULL
    jne loop                # 非空则继续循环

done:
    popq %rbx               # 恢复被调用者保存寄存器
    ret
```

程序运行结果如下图所示，返回值 `%rax = 0xcba` (即 $10 + 176 + 3072 = 3258$) 与预期一致。

```
(base) laonuo@DESKTOP-VARHBN1:/mnt/d/CSAPP-Experiments/Exp.4_lexue/sim/misc$ ./yis sum.yo
Stopped in 28 steps at PC = 0x13.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%rax:   0x0000000000000000      0x0000000000000cba
%rsp:   0x0000000000000000      0x0000000000000200

Changes to memory:
0x01f0: 0x0000000000000000      0x000000000000005b
0x01f8: 0x0000000000000000      0x0000000000000013
```

2.3 `rsum.ys`：链表递归求和

`rsum_list` 函数采用递归方式实现链表求和。递归的基例是空指针返回 0，递归步骤是当前节点值加上剩余链表的和。

核心实现如下：

```
1  rsum_list:
2      andq %rdi, %rdi          # 测试 rdi 是否为 NULL
3      je base_case             # 空指针跳转到基例
4      pushq %rbx               # 保存 %rbx
5      pushq %r12               # 保存 %r12
6      mrmovq (%rdi), %rbx      # 保存当前节点的 val
7      mrmovq 8(%rdi), %r12     # 获取 next 指针
8      rrmovq %r12, %rdi        # 设置递归调用参数
9      call rsum_list           # 递归调用
10     addq %rbx, %rax          # sum = val + rsum_list(next)
11     popq %r12                # 恢复寄存器
12     popq %rbx
13     ret
14
15 base_case:
16     xorq %rax, %rax          # 返回 0
17     ret
```

注意这里递归的实现需要使用额外的被调用者保存寄存器 (`%rbx` 和 `%r12`) 保存当前节点的值和 next 指针，以确保递归返回后能正确累加。运行结果如下图所示，返回值同样为 `0xcba`，与预期一致。

```
(base) laonuo@DESKTOP-VARHBN1:/mnt/d/CSAPP-Experiments/Exp.4_lexue/sim/misc$ ./yis rsum.yo
Stopped in 46 steps at PC = 0x13.  Status 'HLT', CC Z=0 S=0 O=0
Changes to registers:
%rax:   0x0000000000000000      0x0000000000000cba
%rsp:   0x0000000000000000      0x0000000000000200

Changes to memory:
0x01a8: 0x0000000000000000      0x000000000000008a
0x01b0: 0x0000000000000000      0x0000000000000038
0x01b8: 0x0000000000000000      0x00000000000000b0
0x01c0: 0x0000000000000000      0x000000000000008a
0x01c8: 0x0000000000000000      0x0000000000000028
0x01d0: 0x0000000000000000      0x000000000000000a
0x01d8: 0x0000000000000000      0x000000000000008a
0x01f0: 0x0000000000000000      0x000000000000005b
0x01f8: 0x0000000000000000      0x0000000000000013
```

2.4 `copy.ys`：块复制与异或校验

`copy_block` 函数将源数组复制到目标数组，同时计算所有元素的异或校验和。

核心实现如下：

```
1  copy_block:
2      pushq %rbx              # 保存被调用者保存寄存器
3      xorq %rax, %rax         # 初始化 result = 0
4      irmovq $8, %r8          # 常量 8 (指针步进)
5      irmovq $1, %r9          # 常量 1 (计数器递减)
6      jmp test
7
8  loop:
9      mrmovq (%rdi), %rbx     # 读取 src[i]
10     rmmovq %rbx, (%rsi)     # 写入 dst[i]
11     xorq %rbx, %rax         # result ^= src[i]
12     addq %r8, %rdi          # src++
13     addq %r8, %rsi          # dst++
14     subq %r9, %rdx          # len--
15
16 test:
17     andq %rdx, %rdx         # 测试 len > 0
18     jg loop                 # 正数则继续循环
19
20 done:
21     popq %rbx
22     ret
```

由于 Y86-64 缺少立即数加法指令 (Part B 将添加 `iaddq`)，因此这里需要使用寄存器 `%r8` 和 `%r9` 存储常量 8 和 1。运行结果如下图所示，异或校验和 `0x00a ^ 0x0b0 ^ 0xc00 = 0xcba` 验证正确，与预期一致。

```
(base) laonuo@DESKTOP-VARHBN1:/mnt/d/CSAPP-Experiments/Exp.4_lexue/sim/misc$ ./yis copy.yo
Stopped in 41 steps at PC = 0x13.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%rax:   0x0000000000000000      0x0000000000000cba
%rsp:   0x0000000000000000      0x0000000000000200
%rsi:   0x0000000000000000      0x0000000000000048
%rdi:   0x0000000000000000      0x0000000000000030
%r8:    0x0000000000000000      0x0000000000000008
%r9:    0x0000000000000000      0x0000000000000001

Changes to memory:
0x0030: 0x0000000000000111      0x000000000000000a
0x0038: 0x0000000000000222      0x00000000000000b0
0x0040: 0x0000000000000333      0x0000000000000c00
0x01f0: 0x0000000000000000      0x000000000000006f
0x01f8: 0x0000000000000000      0x0000000000000013
```

3 Part B：扩展 SEQ 处理器支持 `iaddq` 指令

3.1 任务概述

Part B 要求修改 `seq-full.hcl`，使 SEQ 处理器支持 `iaddq V, rB` 指令。该指令将立即数 V 加到寄存器 rB，并更新条件码。

3.2 `iaddq` 指令分析

- **指令格式**

`iaddq` 指令的编码格式如下图所示，共占 10 字节：

icode	ifun	rA	rB	V
C	0	F	rB	8 字节立即数
1 字节		1 字节		8 字节

其中 `rA` 字段固定为 `F` (表示不使用)，`rB` 指定目标寄存器。

- **各阶段行为**

阶段	操作
取指	`icode:ifun ← C:0`，`rA:rB ← M1[PC+1]`，`valC ← M8[PC+2]`，`valP ← PC+10`
译码	`valB ← R[rB]`
执行	`valE ← valB + valC`，设置条件码 (ZF, SF, OF)
访存	(无操作)
写回	`R[rB] ← valE`
更新 PC	`PC ← valP`

3.3 HCL 修改详解

`iaddq` 指令的行为与 `OPq` 类似 (执行算术运算并设置条件码)，但操作数来源不同：一个来自立即数 `valC`，另一个来自寄存器 `valB`。对 `seq-full.hcl` 的修改涉及以下部分：

```
## Fetch 阶段：添加为合法指令
bool instr_valid = icode in
    { INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
      IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ, IIADDQ };  # + IIADDQ

## Fetch 阶段：需要寄存器字节
bool need_regids =
    icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
               IIRMOVQ, IRMMOVQ, IMRMOVQ, IIADDQ };      # + IIADDQ

## Fetch 阶段：需要立即数
bool need_valC =
    icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ,
               IJXX, ICALL, IIADDQ };                    # + IIADDQ

## Decode 阶段：从 rB 读取
word srcB = [
    icode in { IOPQ, IRMMOVQ, IMRMOVQ, IIADDQ } : rB;   # + IIADDQ
    ...
];

## Decode 阶段：写回 rB
word dstE = [
    icode in { IRRMOVQ } && Cnd : rB;
    icode in { IIRMOVQ, IOPQ, IIADDQ } : rB;            # + IIADDQ
    ...
];

## Execute 阶段：ALU 输入 A 使用立即数
word aluA = [
    icode in { IRRMOVQ, IOPQ } : valA;
    icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IIADDQ } : valC;  # +
IIADDQ
    ...
];

## Execute 阶段：ALU 输入 B 使用寄存器值
word aluB = [
    icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
               IPUSHQ, IRET, IPOPQ, IIADDQ } : valB;    # + IIADDQ
    ...
];

## Execute 阶段：设置条件码
bool set_cc = icode in { IOPQ, IIADDQ };                # + IIADDQ
```

3.4 测试结果

ISA 检查通过，验证 `iaddq` 指令行为正确：

```
(base) laonuo@DESKTOP-VARHBN1:/mnt/d/CSAPP-Experiments/Exp.4_lexue/sim/seq$ ./ssim -t ../y86-code/asumi.yo | tail -12
32 instructions executed
Status = HLT
Condition Codes: Z=1 S=0 O=0
Changed Register State:
%rax:   0x0000000000000000      0x0000abcdabcdabcd
%rsp:   0x0000000000000000      0x0000000000000100
%rdi:   0x0000000000000000      0x0000000000000038
%r10:   0x0000000000000000      0x0000a000a000a000
Changed Memory State:
0x00f0: 0x0000000000000000      0x0000000000000055
0x00f8: 0x0000000000000000      0x0000000000000013
ISA Check Succeeds
```

y86-code 回归测试全部通过：

```
(base) laonuo@DESKTOP-VARHBN1:/mnt/d/CSAPP-Experiments/Exp.4_lexue/sim/y86-code$ make testssim | tail -18
Makefile:42: warning: ignoring prerequisites on suffix rule definition
Makefile:45: warning: ignoring prerequisites on suffix rule definition
Makefile:48: warning: ignoring prerequisites on suffix rule definition
Makefile:51: warning: ignoring prerequisites on suffix rule definition
grep "ISA Check" *.seq
asum.seq:ISA Check Succeeds
asumr.seq:ISA Check Succeeds
cjr.seq:ISA Check Succeeds
j-cc.seq:ISA Check Succeeds
poptest.seq:ISA Check Succeeds
prog1.seq:ISA Check Succeeds
prog2.seq:ISA Check Succeeds
prog3.seq:ISA Check Succeeds
prog4.seq:ISA Check Succeeds
prog5.seq:ISA Check Succeeds
prog6.seq:ISA Check Succeeds
prog7.seq:ISA Check Succeeds
prog8.seq:ISA Check Succeeds
pushquestion.seq:ISA Check Succeeds
pushtest.seq:ISA Check Succeeds
ret-hazard.seq:ISA Check Succeeds
rm asum.seq asumr.seq cjr.seq j-cc.seq poptest.seq pushquestion.seq pushtest.seq prog1.seq prog2.seq prog3.seq prog4.seq prog5.seq prog6.
seq prog7.seq prog8.seq ret-hazard.seq
```

ptest 测试套件 (含 iaddq 测试) 全部通过：

```
(base) laonuo@DESKTOP-VARHBN1:/mnt/d/CSAPP-Experiments/Exp.4_lexue/sim/ptest$ make SIM=../seq/ssim TFLAGS=-i
./optest.pl -s ../seq/ssim -i
Simulating with ../seq/ssim
  All 58 ISA Checks Succeed
./jtest.pl -s ../seq/ssim -i
Simulating with ../seq/ssim
  All 96 ISA Checks Succeed
./ctest.pl -s ../seq/ssim -i
Simulating with ../seq/ssim
  All 22 ISA Checks Succeed
./htest.pl -s ../seq/ssim -i
Simulating with ../seq/ssim
  All 756 ISA Checks Succeed
```

4 Part C：优化 ncopy 函数与 PIPE 处理器

4.1 任务概述

Part C 的目标是优化 `ncopy.ys` (复制数组并统计正数个数) 和 `pipe-full.hcl` (流水线处理器)，使平均 CPE 尽可能低，同时满足以下约束：

- `ncopy.ys` 代码不超过 1000 字节
- `pipe-full.hcl` 必须通过所有标准测试，保证正确无误
- `ncopy.ys` 必须通过 `./correctness.pl` 正确性测试

4.2 优化策略一：iaddq 指令

首先，我们可以将 Part B 实现的 `iaddq` 指令迁移到 PIPE 处理器中，替代 `irmovq + addq` 组合，以减少指令数量，例如：

```
# 优化前                          # 优化后
irmovq $8, %r10                   iaddq $8, %rdi
addq %r10, %rdi
```

这一优化可以使每次指针更新从 2 条指令减少为 1 条。

4.3 优化策略二：10× 循环展开

循环展开是减少循环控制开销的经典方法。通过测试，我发现代码大小与展开因子近似成正比 (8× ≈ 800 字节，12× ≈ 1200 字节)，结合题目要求的 1000 字节限制，因此选择了 10× 展开。

每个元素的处理模式如下：

```
L0:
    mrmovq (%rdi), %r8          # 从 src 读取元素
    rmmovq %r8, (%rsi)          # 写入 dst
    andq %r8, %r8               # 设置条件码
    jle L1                      # 非正数跳过计数
    iaddq $1, %rax              # count++
L1:
    mrmovq 8(%rdi), %r8         # 处理下一个元素
    rmmovq %r8, 8(%rsi)
    andq %r8, %r8
    jle L2
    iaddq $1, %rax
L2:
    # ... 重复至 L9 ...
```

循环尾部更新指针并判断是否继续：

```
    iaddq $80, %rdi             # src += 10
    iaddq $80, %rsi             # dst += 10
    iaddq $-10, %rdx            # len -= 10
    jge L0                      # len >= 0 继续循环
```

4.4 优化策略三：达夫设备 (Duff's Device) 优化+跳转树

循环展开后需处理 0–9 个剩余元素。一种简单的处理方法是写成一个独立的循环依次处理，虽然简单，但是效率低下；另外一种改进方法是采用**达夫设备 (Duff's Device) 优化**，将余数处理的循环也展开，并借助跳转表直接跳转到对应的余数处理，这能够在 `O(1)` 时间复杂度内找到跳转目标。

但是在实际实现过程当中发现，由于 Y86-64 处理器不支持间接跳转，因此需要使用 `pushq` 和 `ret` 指令组合来实现跳转表的跳转，代码如下：

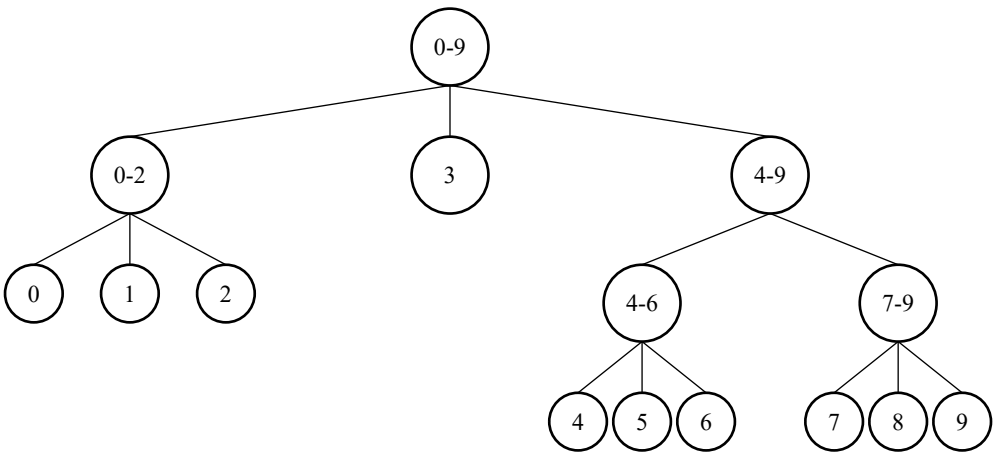
```
# 假设跳转表存储在 jump_table，余数在 %rdx (范围 0-9)
# 每个跳转地址占 8 字节

irmovq jump_table, %r8       # 跳转表基地址
addq %rdx, %rdx              # rdx *= 2
addq %rdx, %rdx              # rdx *= 2 (现在 rdx *= 4)
addq %rdx, %rdx              # rdx *= 8 (计算偏移量)
addq %rdx, %r8               # r8 = &jump_table[rdx]
mrmovq (%r8), %r8            # r8 = 目标地址
pushq %r8                    # 将目标地址压栈
ret                          # 弹出并跳转

.align 8
jump_table:
    .quad R0
    .quad R1
    .quad R2
    # ... R3 到 R9 ...
```

其中不仅存在 Load-Use 冒险，`ret` 也会产生 3 个周期的延迟，实际效率并没有想象中的高。事实上，跳转表更加适合余数范围较大的场景，而我们的余数范围较小，因此可以考虑采用**跳转树**的方法来优化跳转目标的查找。

4.4.1 跳转树结构



4.4.2 设计亮点

- **非平衡树设计**：以 3 为界而非 4 或 5，可以使小余数分支深度更浅。之所以要这样处理，是因为实际评测当中，小数组占多数，可以观察到小余数 (0-3) 的处理频率更高，且小数组中余数处理 CPE 权重高，因此优先优化小余数处理可显著降低平均 CPE。
- **Fall-through 结构**：余数处理代码采用“瀑布式”设计，例如 R9 直接 fall-through 到 R8，R8 到 R7，以此类推，避免多余跳转指令。这与达夫设备的设计思路类似，但更加灵活，可以根据实际分支情况调整结构。

```
R9:
    mrmovq 64(%rdi), %r8
    rmmovq %r8, 64(%rsi)
    andq %r8, %r8
    jle R8
    iaddq $1, %rax
R8:                              # 直接 fall-through 到 R8，避免跳转
    mrmovq 56(%rdi), %r8
    rmmovq %r8, 56(%rsi)
    # ... 以此类推 ...
```

4.5 优化策略四：加载转发 (Load Forwarding)

4.4.1 问题分析

主循环中 `mrmovq` 紧跟 `rmmovq` 的序列存在 Load-Use 冒险。标准 PIPE 处理器采用加载互锁 (Load Interlock) 解决，即暂停一个周期等待 `mrmovq` 完成内存访问。

```
mrmovq (%rdi), %r8      # Memory 阶段产生 valM
rmmovq %r8, (%rsi)      # Decode 阶段需要 %r8 → 冲突！
```

教材指出“Load-Use 冒险不能通过转发解决”，因为 `mrmovq` 在 Memory 阶段才会产生 `valM`，而 `rmmovq` 在 Decode 阶段需要用到 `valA`，因此无法通过转发解决。

但是通过实际分析后，我发现 `rmmovq` 虽然在 Decode 阶段读取了 `%r8`，但真正使用这个值是在 **Memory 阶段** (写入内存) 。当 `rmmovq` 到达 Execute 阶段时，前一条 `mrmovq` 也同时进入 Memory 阶段，此时 `m_valM` 可用。因此可在 Execute 阶段新增转发路径，用正确的 `m_valM` 覆盖错误的 `valA`。

4.5.2 HCL 修改

在 `pipe-full.hcl` 中新增 `e_valA` 信号，并放宽冒险检测条件：

```
## Execute 阶段：加载转发
word e_valA = [
    # rmmovq/pushq 且 srcA 冲突时，直接转发 m_valM
    E_icode in { IRMMOVQ, IPUSHQ } && E_srcA == M_dstM : m_valM;
    1 : E_valA;    # 默认使用原值
];

## 放宽冒险检测条件 (仅 srcB 冲突或 srcA 冲突且非 rmmovq/pushq 时才暂停)
bool F_stall =
    E_icode in { IMRMOVQ, IPOPQ } &&
    (
        E_dstM == d_srcB ||
        (E_dstM == d_srcA && !(D_icode in { IRMMOVQ, IPUSHQ }))
    ) ||
    IRET in { D_icode, E_icode, M_icode };
```

类似修改同样应用于 `D_stall`、`D_bubble`、`E_bubble`。

4.6 测试结果

PIPE 回归测试通过：

```
(base) laonuo@DESKTOP-VARHBN1:/mnt/d/CSAPP-Experiments/Exp.4_lexue/sim/y86-code$ make testpsim | tail -18
Makefile:42: warning: ignoring prerequisites on suffix rule definition
Makefile:45: warning: ignoring prerequisites on suffix rule definition
Makefile:48: warning: ignoring prerequisites on suffix rule definition
Makefile:51: warning: ignoring prerequisites on suffix rule definition
grep "ISA Check" *.pipe
asum.pipe:ISA Check Succeeds
asumr.pipe:ISA Check Succeeds
cjr.pipe:ISA Check Succeeds
cjr.pipe:ISA Check Succeeds
cjr.pipe:ISA Check Succeeds
cjr.pipe:ISA Check Succeeds
cjr.pipe:ISA Check Succeeds
cjr.pipe:ISA Check Succeeds
cjr.pipe:ISA Check Succeeds
cjr.pipe:ISA Check Succeeds
cjr.pipe:ISA Check Succeeds
cjr.pipe:ISA Check Succeeds
cjr.pipe:ISA Check Succeeds
cjr.pipe:ISA Check Succeeds
cjr.pipe:ISA Check Succeeds
cjr.pipe:ISA Check Succeeds
cjr.pipe:ISA Check Succeeds
j-cc.pipe:ISA Check Succeeds
poptest.pipe:ISA Check Succeeds
prog1.pipe:ISA Check Succeeds
prog2.pipe:ISA Check Succeeds
prog3.pipe:ISA Check Succeeds
prog4.pipe:ISA Check Succeeds
prog5.pipe:ISA Check Succeeds
prog6.pipe:ISA Check Succeeds
prog7.pipe:ISA Check Succeeds
prog8.pipe:ISA Check Succeeds
pushquestion.pipe:ISA Check Succeeds
pushtest.pipe:ISA Check Succeeds
ret-hazard.pipe:ISA Check Succeeds
rm asum.pipe asumr.pipe cjr.pipe j-cc.pipe poptest.pipe pushquestion.pipe pushtest.pipe prog1.pipe prog2.pipe prog3.pipe prog4.pipe prog
5.pipe prog6.pipe prog7.pipe prog8.pipe ret-hazard.pipe
```

ptest 测试套件通过 (含 iaddq)：

```
(base) laonuo@DESKTOP-VARHBN1:/mnt/d/CSAPP-Experiments/Exp.4_lexue/sim/ptest$ make SIM=../pipe/psim TFLAGS=-i
./optest.pl -s ../pipe/psim -i
Simulating with ../pipe/psim
  All 58 ISA Checks Succeed
./jtest.pl -s ../pipe/psim -i
Simulating with ../pipe/psim
  All 96 ISA Checks Succeed
./ctest.pl -s ../pipe/psim -i
Simulating with ../pipe/psim
  All 22 ISA Checks Succeed
./htest.pl -s ../pipe/psim -i
Simulating with ../pipe/psim
  All 756 ISA Checks Succeed
```

`ncopy.ys` 正确性测试与字节数通过：

```
(base) laonuo@DESKTOP-VARHBN1:/mnt/d/CSAPP-Experiments/Exp.4_lexue/sim/pipe$ ./correctness.pl | tail -1 && ./correctness.pl -p | tail -1 && ./check-len.pl < ncopy.yo
68/68 pass correctness test
68/68 pass correctness test
ncopy length = 997 bytes
```

Benchmark 结果：

```
(base) laonuo@DESKTOP-VARHBN1:/mnt/d/CSAPP-Experiments/Exp.4_lexue/sim/pipe$ ./benchmark.pl | tail -2
Average CPE     7.46
Score   60.0/60.0
```

优化阶段	CPE
原始版本	15.18
+ iaddq	11.46
+ 10× 展开	9.18
+ 分治搜索余数处理	7.88
+ 加载转发	**7.46**

最终 CPE 为 **7.46**，达到满分标准 (< 7.5) 。

5 实验心得

本次实验围绕 Y86-64 处理器的设计与优化展开，主要收获集中在以下两个技术点：

5.1 跳转树优化与设计

循环展开是经典的优化手段，但展开后的余数处理往往被忽视。本实验中，通过对达夫设备进行改进，将跳转表替换为跳转树。更关键的是，通过分析评测数据分布 (小数组权重高) ，采用非平衡设计优先优化小余数情况，进一步提升了平均 CPE 表现。这一思路体现了“面向评测优化”的工程实践方法。

5.2 加载转发的突破

教材明确指出 Load-Use 冒险不能通过转发解决，需要使用加载互锁。本实验中，通过分析 `mrmovq` + `rmmovq` 序列的特殊性——`rmmovq` 在 Memory 阶段

才真正需要数据——发现可以在 Execute 阶段新增转发路径，绕过加载互锁的限制。这一优化将每个元素的处理周期减少 1，进一步降低了 CPE。

这一经历提示，在工程实践中，教材结论往往基于一般情况，针对特定场景进行深入分析可能发现突破口。

6 附录：源代码

源代码文件也一并提交至附件。

6.1 `sum.ys`

```
1  .pos 0
2  # sum.ys - Iteratively sum linked list elements
3  # 姓名：左逸龙
4  # 学号：1120231863
5      irmovq stack, %rsp
6      call main
7      halt
8
9  .align 8
10 ele1:
11     .quad 0x00a
12     .quad ele2
13 ele2:
14     .quad 0x0b0
15     .quad ele3
16 ele3:
17     .quad 0xc00
18     .quad 0
19
20 main:
21     irmovq ele1, %rdi
22     call sum_list
23     ret
24
25 sum_list:
26     pushq %rbx
27     xorq %rax, %rax
28     jmp test
29
30 loop:
31     mrmovq (%rdi), %rbx
32     addq %rbx, %rax
33     mrmovq 8(%rdi), %rdi
34
35 test:
36     andq %rdi, %rdi
37     jne loop
38
```

```
done:
    popq %rbx
    ret

.pos 0x200
stack:
```

6.2 `rsum.ys`

```
.pos 0
# rsum.ys - Recursively sum linked list elements
# 姓名：左逸龙
# 学号：1120231863
    irmovq stack, %rsp
    call main
    halt

.align 8
ele1:
    .quad 0x00a
    .quad ele2
ele2:
    .quad 0x0b0
    .quad ele3
ele3:
    .quad 0xc00
    .quad 0

main:
    irmovq ele1, %rdi
    call rsum_list
    ret

rsum_list:
    andq %rdi, %rdi
    je base_case
    pushq %rbx
    pushq %r12
    mrmovq (%rdi), %rbx
    mrmovq 8(%rdi), %r12
    rrmovq %r12, %rdi
    call rsum_list
    addq %rbx, %rax
    popq %r12
    popq %rbx
    ret

base_case:
    xorq %rax, %rax
    ret

```

```
.pos 0x200
stack:
```

6.3 `copy.ys`

```
.pos 0
# copy.ys - Copy block and return XOR checksum
# 姓名：左逸龙
# 学号：1120231863
    irmovq stack, %rsp
    call main
    halt

.align 8
src:
    .quad 0x00a
    .quad 0x0b0
    .quad 0xc00
dest:
    .quad 0x111
    .quad 0x222
    .quad 0x333

main:
    irmovq src, %rdi
    irmovq dest, %rsi
    irmovq $3, %rdx
    call copy_block
    ret

copy_block:
    pushq %rbx
    xorq %rax, %rax
    irmovq $8, %r8
    irmovq $1, %r9
    jmp test

loop:
    mrmovq (%rdi), %rbx
    rmmovq %rbx, (%rsi)
    xorq %rbx, %rax
    addq %r8, %rdi
    addq %r8, %rsi
    subq %r9, %rdx

test:
    andq %rdx, %rdx
    jg loop

done:
    popq %rbx
```

```
47      ret
48
49  .pos 0x200
50  stack:
```

6.4 seq-full.hcl

```
1   #/* $begin seq-all-hcl */
2   ####################################################################
3   #  HCL Description of Control for Single Cycle Y86-64 Processor SEQ
#
4   #  Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2010       #
5   ####################################################################
6
7   ## Your task is to implement the iaddq instruction
8   ## The file contains a declaration of the icodes
9   ## for iaddq (IIADDQ)
10  ## Your job is to add the rest of the logic to make it work
11
12
####################################################################
13  # 姓名：左逸龙
14  # 学号：1120231863
15  # Instruction: iaddq V, rB
16  #   Fetch:      icode:ifun <- C:0, rA:rB <- M[PC+1], valC <- M[PC+2],
valP <- PC+10
17  #   Decode:     valB <- R[rB]
18  #   Execute:    valE <- valB + valC, condition codes updated
19  #   Memory:     (no access)
20  #   Write:      R[rB] <- valE
21  #   PC:         valP
22
####################################################################
23
24  ####################################################################
25  #    C Include's.  Don't alter these                               #
26  ####################################################################
27
28  quote '#include <stdio.h>'
29  quote '#include "isa.h"'
30  quote '#include "sim.h"'
31  quote 'int sim_main(int argc, char *argv[]);'
32  quote 'word_t gen_pc(){return 0;}'
33  quote 'int main(int argc, char *argv[])'
34  quote '  {plusmode=0;return sim_main(argc,argv);}'
35
36  ####################################################################
37  #    Declarations.  Do not change/remove/delete any of these       #
38  ####################################################################
39
40  ##### Symbolic representation of Y86-64 Instruction Codes
```

```
#############
wordsig INOP  'I_NOP'
wordsig IHALT 'I_HALT'
wordsig IRRMOVQ 'I_RRMOVQ'
wordsig IIRMOVQ 'I_IRMOVQ'
wordsig IRMMOVQ 'I_RMMOVQ'
wordsig IMRMOVQ 'I_MRMOVQ'
wordsig IOPQ  'I_ALU'
wordsig IJXX  'I_JMP'
wordsig ICALL 'I_CALL'
wordsig IRET  'I_RET'
wordsig IPUSHQ  'I_PUSHQ'
wordsig IPOPQ 'I_POPQ'
# Instruction code for iaddq instruction
wordsig IIADDQ  'I_IADDQ'

##### Symbolic represenations of Y86-64 function codes
#####
wordsig FNONE    'F_NONE'        # Default function code

##### Symbolic representation of Y86-64 Registers referenced
explicitly #####
wordsig RRSP     'REG_RSP'      # Stack Pointer
wordsig RNONE    'REG_NONE'     # Special value indicating "no
register"

##### ALU Functions referenced explicitly
#####
wordsig ALUADD  'A_ADD'   # ALU should add its arguments

##### Possible instruction status values
#####
wordsig SAOK  'STAT_AOK'  # Normal execution
wordsig SADR  'STAT_ADR'  # Invalid memory address
wordsig SINS  'STAT_INS'  # Invalid instruction
wordsig SHLT  'STAT_HLT'  # Halt instruction encountered

##### Signals that can be referenced by control logic
####################

##### Fetch stage inputs    #####
wordsig pc 'pc'       # Program counter
##### Fetch stage computations    #####
wordsig imem_icode 'imem_icode'   # icode field from instruction
memory
wordsig imem_ifun  'imem_ifun'    # ifun field from instruction
memory
wordsig icode   'icode'   # Instruction control code
wordsig ifun    'ifun'    # Instruction function
wordsig rA    'ra'      # rA field from instruction
wordsig rB    'rb'      # rB field from instruction
wordsig valC    'valc'    # Constant from instruction
wordsig valP    'valp'    # Address of following instruction
```

```
85 boolsig imem_error 'imem_error'   # Error signal from instruction
memory
86 boolsig instr_valid 'instr_valid' # Is fetched instruction valid?
87
88 ##### Decode stage computations   #####
89 wordsig valA  'vala'      # Value from register A port
90 wordsig valB  'valb'      # Value from register B port
91
92 ##### Execute stage computations  #####
93 wordsig valE  'vale'      # Value computed by ALU
94 boolsig Cnd 'cond'      # Branch test
95
96 ##### Memory stage computations   #####
97 wordsig valM  'valm'      # Value read from memory
98 boolsig dmem_error 'dmem_error'   # Error signal from data memory
99
100
101 ################################################################
102 #    Control Signal Definitions.                               #
103 ################################################################
104
105 ################ Fetch Stage     ###################################
106
107 # Determine instruction code
108 word icode = [
109   imem_error: INOP;
110   1: imem_icode;    # Default: get from instruction memory
111 ];
112
113 # Determine instruction function
114 word ifun = [
115   imem_error: FNONE;
116   1: imem_ifun;   # Default: get from instruction memory
117 ];
118
119 bool instr_valid = icode in
120   { INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
121         IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ, IIADDQ };
122
123 # Does fetched instruction require a regid byte?
124 bool need_regids =
125   icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
126         IIRMOVQ, IRMMOVQ, IMRMOVQ, IIADDQ };
127
128 # Does fetched instruction require a constant word?
129 bool need_valC =
130   icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX, ICALL, IIADDQ };
131
132 ################ Decode Stage    ###################################
133
134 ## What register should be used as the A source?
135 word srcA = [
136   icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ  } : rA;
```

```
137   icode in { IPOPQ, IRET } : RRSP;
138   1 : RNONE; # Don't need register
139 ];
140
141 ## What register should be used as the B source?
142 word srcB = [
143   icode in { IOPQ, IRMMOVQ, IMRMOVQ, IIADDQ } : rB;
144   icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
145   1 : RNONE;  # Don't need register
146 ];
147
148 ## What register should be used as the E destination?
149 word dstE = [
150   icode in { IRRMOVQ } && Cnd : rB;
151   icode in { IIRMOVQ, IOPQ, IIADDQ } : rB;
152   icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
153   1 : RNONE;  # Don't write any register
154 ];
155
156 ## What register should be used as the M destination?
157 word dstM = [
158   icode in { IMRMOVQ, IPOPQ } : rA;
159   1 : RNONE;  # Don't write any register
160 ];
161
162 ################ Execute Stage   ###################################
163
164 ## Select input A to ALU
165 word aluA = [
166   icode in { IRRMOVQ, IOPQ } : valA;
167   icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IIADDQ } : valC;
168   icode in { ICALL, IPUSHQ } : -8;
169   icode in { IRET, IPOPQ } : 8;
170   # Other instructions don't need ALU
171 ];
172
173 ## Select input B to ALU
174 word aluB = [
175   icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
176            IPUSHQ, IRET, IPOPQ, IIADDQ } : valB;
177   icode in { IRRMOVQ, IIRMOVQ } : 0;
178   # Other instructions don't need ALU
179 ];
180
181 ## Set the ALU function
182 word alufun = [
183   icode == IOPQ : ifun;
184   1 : ALUADD;
185 ];
186
187 ## Should the condition codes be updated?
188 bool set_cc = icode in { IOPQ, IIADDQ };
189
```

```
################ Memory Stage    ###################################

## Set read control signal
bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET };

## Set write control signal
bool mem_write = icode in { IRMMOVQ, IPUSHQ, ICALL };

## Select memory address
word mem_addr = [
  icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;
  icode in { IPOPQ, IRET } : valA;
  # Other instructions don't need address
];

## Select memory input data
word mem_data = [
  # Value from register
  icode in { IRMMOVQ, IPUSHQ } : valA;
  # Return PC
  icode == ICALL : valP;
  # Default: Don't write anything
];

## Determine instruction status
word Stat = [
  imem_error || dmem_error : SADR;
  !instr_valid: SINS;
  icode == IHALT : SHLT;
  1 : SAOK;
];

################ Program Counter Update ############################

## What address should instruction be fetched at

word new_pc = [
  # Call.  Use instruction constant
  icode == ICALL : valC;
  # Taken branch.  Use instruction constant
  icode == IJXX && Cnd : valC;
  # Completion of RET instruction.  Use value from stack
  icode == IRET : valM;
  # Default: Use incremented PC
  1 : valP;
];
#/* $end seq-all-hcl */
```

6.5 ncopy.ys

```
#/* $begin ncopy-ys */
##################################################################
# ncopy.ys - Copy a src block of len words to dst.
# Return the number of positive words (>0) contained in src.
#
# 姓名：左逸龙
# 学号：1120231863
#
# ======================== 优化说明 ========================
#
# 1. 加载转发 (Load Forwarding) - 配合 pipe-full.hcl 的修改
#    - 问题：标准 PIPE 中，mrmovq 后紧跟 rmmovq 使用同一寄存器会产生
#      1 个周期的气泡 (load-use 冒险)
#    - 解决：在 pipe-full.hcl 中修改 e_valA 信号，当 Execute 阶段的
#      rmmovq 需要的数据正好由 Memory 阶段的 mrmovq 产生时，直接转发
#      m_valM，避免暂停
#    - 效果：每个元素节省 1 个周期
#
# 2. 10x 循环展开 (Loop Unrolling)
#    - 思路：每次循环处理 10 个元素，减少循环控制开销
#    - 展开因子选择：10x 在代码大小 (<1000字节) 和性能之间取得平衡
#    - 每个元素的处理模式：
#      mrmovq offset(%rdi), %r8  # 从 src 读取
#      rmmovq %r8, offset(%rsi)  # 写入 dst (加载转发消除了气泡)
#      andq %r8, %r8             # 设置条件码
#      jle NextLabel             # 如果 <= 0 跳过计数
#      iaddq $1, %rax            # count++
#
# 3. iaddq 指令
#    - 使用 iaddq 替代 irmovq + addq 组合，减少指令数量
#    - 例如：iaddq $80, %rdi 代替 irmovq $80, %r10 + addq %r10, %rdi
#
# 4. 达夫设备 (Duff's Device) 优化+跳转树
#    - 问题：循环展开后需要处理 0-9 个剩余元素
#    - 思路：使用达夫设备 (Duff's Device) 优化+跳转树减少平均分支次数
#    - 结构：先与 3 比较分为 [0-2], [3], [4-9] 三组
#            再递归细分各子区间
#    - 优势：平均只需 2-3 次比较即可定位到正确的处理入口，相比跳转表，跳转树的效
率更高
#
##################################################################
# Do not modify this portion
# Function prologue.
# %rdi = src, %rsi = dst, %rdx = len
ncopy:

##################################################################
# You can modify this portion

```

```
49    # ========== 主循环：10x 展开 ==========
50    # 首先将 len 减 10，判断是否能进入主循环
51    iaddq $-10, %rdx
52    jl Rem      # len < 10，跳转到余数处理
53
54  L0:
55    # 处理第 0 个元素
56    mrmovq (%rdi), %r8
57    rmmovq %r8, (%rsi)
58    andq %r8, %r8
59    jle L1
60    iaddq $1, %rax
61  L1:
62    # 处理第 1 个元素
63    mrmovq 8(%rdi), %r8
64    rmmovq %r8, 8(%rsi)
65    andq %r8, %r8
66    jle L2
67    iaddq $1, %rax
68  L2:
69    # 处理第 2 个元素
70    mrmovq 16(%rdi), %r8
71    rmmovq %r8, 16(%rsi)
72    andq %r8, %r8
73    jle L3
74    iaddq $1, %rax
75  L3:
76    # 处理第 3 个元素
77    mrmovq 24(%rdi), %r8
78    rmmovq %r8, 24(%rsi)
79    andq %r8, %r8
80    jle L4
81    iaddq $1, %rax
82  L4:
83    # 处理第 4 个元素
84    mrmovq 32(%rdi), %r8
85    rmmovq %r8, 32(%rsi)
86    andq %r8, %r8
87    jle L5
88    iaddq $1, %rax
89  L5:
90    # 处理第 5 个元素
91    mrmovq 40(%rdi), %r8
92    rmmovq %r8, 40(%rsi)
93    andq %r8, %r8
94    jle L6
95    iaddq $1, %rax
96  L6:
97    # 处理第 6 个元素
98    mrmovq 48(%rdi), %r8
99    rmmovq %r8, 48(%rsi)
100   andq %r8, %r8
101   jle L7
```

```
102   iaddq $1, %rax
103 L7:
104   # 处理第 7 个元素
105   mrmovq 56(%rdi), %r8
106   rmmovq %r8, 56(%rsi)
107   andq %r8, %r8
108   jle L8
109   iaddq $1, %rax
110 L8:
111   # 处理第 8 个元素
112   mrmovq 64(%rdi), %r8
113   rmmovq %r8, 64(%rsi)
114   andq %r8, %r8
115   jle L9
116   iaddq $1, %rax
117 L9:
118   # 处理第 9 个元素
119   mrmovq 72(%rdi), %r8
120   rmmovq %r8, 72(%rsi)
121   andq %r8, %r8
122   jle Nxt
123   iaddq $1, %rax
124 Nxt:
125   # 更新指针，准备下一轮迭代
126   iaddq $80, %rdi   # src += 10
127   iaddq $80, %rsi   # dst += 10
128   iaddq $-10, %rdx  # len -= 10
129   jge L0      # 如果 len >= 0，继续循环
130
131   # ========== 余数处理：二叉搜索 ==========
132   # 此时 rdx 在 [-10, -1]，表示剩余 0-9 个元素
133 Rem:
134   iaddq $7, %rdx    # rdx = len - 3 (因为之前减了10，现在+7相当于原始
len-3)
135   jl R02      # len < 3 → 处理 0-2 个元素
136   jg R49      # len > 3 → 处理 4-9 个元素
137         # len = 3 → 直接 fall-through 到 R3
138
139   # ---------- 处理 3, 2, 1 个元素（fall-through 结构）----------
140 R3:
141   mrmovq 16(%rdi), %r8
142   rmmovq %r8, 16(%rsi)
143   andq %r8, %r8
144   jle R2
145   iaddq $1, %rax
146 R2:
147   mrmovq 8(%rdi), %r8
148   rmmovq %r8, 8(%rsi)
149   andq %r8, %r8
150   jle R1
151   iaddq $1, %rax
152 R1:
153   mrmovq (%rdi), %r8
```

```
154   rmmovq %r8, (%rsi)
155   andq %r8, %r8
156   jle Done
157   iaddq $1, %rax
158   ret     # 直接返回, 节省 jmp Done 的 8 字节
159
160   # ---------- 处理 4-9 个元素 ----------
161 R49:
162   iaddq $-4, %rdx   # rdx = len - 7
163   jl R46      # len < 7 → 处理 4-6 个元素
164   je R7     # len = 7
165   iaddq $-1, %rdx
166   je R8     # len = 8
167         # len = 9 → fall-through
168
169   # 9-4 个元素的处理 (fall-through 到 R3)
170 R9:
171   mrmovq 64(%rdi), %r8
172   rmmovq %r8, 64(%rsi)
173   andq %r8, %r8
174   jle R8
175   iaddq $1, %rax
176 R8:
177   mrmovq 56(%rdi), %r8
178   rmmovq %r8, 56(%rsi)
179   andq %r8, %r8
180   jle R7
181   iaddq $1, %rax
182 R7:
183   mrmovq 48(%rdi), %r8
184   rmmovq %r8, 48(%rsi)
185   andq %r8, %r8
186   jle R6
187   iaddq $1, %rax
188 R6:
189   mrmovq 40(%rdi), %r8
190   rmmovq %r8, 40(%rsi)
191   andq %r8, %r8
192   jle R5
193   iaddq $1, %rax
194 R5:
195   mrmovq 32(%rdi), %r8
196   rmmovq %r8, 32(%rsi)
197   andq %r8, %r8
198   jle R4
199   iaddq $1, %rax
200 R4:
201   mrmovq 24(%rdi), %r8
202   rmmovq %r8, 24(%rsi)
203   andq %r8, %r8
204   jle R3
205   iaddq $1, %rax
206   jmp R3      # 继续处理剩余的 3 个元素
```

```
207
208   # ---------- 处理 4-6 个元素的分支入口 ----------
209 R46:
210   iaddq $2, %rdx    # rdx = len - 5
211   jl R4     # len = 4
212   je R5     # len = 5
213   jmp R6      # len = 6
214
215   # ---------- 处理 0-2 个元素的分支入口 ----------
216 R02:
217   iaddq $2, %rdx    # rdx = len - 1
218   jl Done     # len = 0
219   je R1     # len = 1
220   jmp R2      # len = 2
221
222 ##################################################################
223 # Do not modify the following section of code
224 # Function epilogue.
225 Done:
226   ret
227 ##################################################################
228 # Keep the following label at the end of your function
229 End:
230 #/* $end ncopy-ys */
```

6.6 pipe-full.hcl

```
1   #/* $begin pipe-all-hcl */
2   ###############################################################################
3   #    HCL Description of Control for Pipelined Y86-64 Processor     #
4   #    Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2014     #
5   ###############################################################################
6
7   ## Your task is to implement the iaddq instruction
8   ## The file contains a declaration of the icodes
9   ## for iaddq (IIADDQ)
10  ## Your job is to add the rest of the logic to make it work
11
12
################################################################################
13  # 姓名：左逸龙
14  # 学号：1120231863
15  #
16  # ======================== 实现说明 ========================
17  #
18  # 1. iaddq 指令实现
19  #
20  # iaddq V, rB: 将立即数 V 加到寄存器 rB，同时设置条件码
21  #
22  # 各阶段行为：
```

```
23 #   Fetch:   icode:ifun <- C:0
24 #            rA:rB <- M[PC+1]  (rA = F, 表示不使用)
25 #            valC <- M[PC+2]   (8字节立即数)
26 #            valP <- PC + 10
27 #   Decode:  valB <- R[rB]     (读取目标寄存器的当前值)
28 #   Execute: valE <- valB + valC, 设置 CC (条件码)
29 #   Memory:  (无操作)
30 #   Write:   R[rB] <- valE     (写回结果)
31 #   PC:      PC <- valP
32 #
33 # HCL 修改点:
34 #   - instr_valid: 添加 IIADDQ
35 #   - need_regids: 添加 IIADDQ (需要读取 rB)
36 #   - need_valC:   添加 IIADDQ (需要立即数)
37 #   - d_srcB:      添加 IIADDQ (从 rB 读取)
38 #   - d_dstE:      添加 IIADDQ (写入 rB)
39 #   - aluA:        添加 IIADDQ -> valC
40 #   - aluB:        添加 IIADDQ -> valB
41 #   - set_cc:      添加 IIADDQ (需要设置条件码)
42 #
43 # 2. 加载转发优化 (Load Forwarding)
44 #
45 # 问题背景:
46 #   在标准 PIPE 处理器中, 以下代码会产生 1 个周期的暂停:
47 #     mrmovq (%rdi), %r8   # 从内存加载到 r8
48 #     rmmovq %r8, (%rsi)   # 立即使用 r8 -> load-use 冒险!
49 #
50 #   原因: mrmovq 在 Memory 阶段产生 valM, 而 rmmovq 在 Decode
51 #   阶段就需要读取 r8。即使有转发, 当 rmmovq 进入 Execute 时,
52 #   mrmovq 刚离开 Memory 阶段, 数据无法及时转发。
53 #
54 # 解决方案:
55 #   观察到 rmmovq 实际上在 Memory 阶段才需要 valA (用于写内存),
56 #   而此时 mrmovq 已经完成 Memory 阶段, m_valM 可用。
57 #   因此可以将 m_valM 直接转发到 Execute 阶段的 e_valA。
58 #
59 # 实现修改:
60 #   1. e_valA 信号: 当 Execute 阶段是 rmmovq/pushq 且其 srcA
61 #      与 Memory 阶段的 dstM 匹配时, 使用 m_valM
62 #
63 #   2. 冒险检测条件 (F_stall, D_stall, D_bubble, E_bubble):
64 #      放宽 load-use 冒险的检测条件, 当下一条指令是 rmmovq/pushq
65 #      且冲突发生在 srcA 时, 不产生暂停 (因为可以转发)
66 #      条件变为: 只有当 dstM == srcB, 或者
67 #      (dstM == srcA 且 下一条指令不是 rmmovq/pushq) 时才暂停
68 #
69
##################################################################
70 ##################################################################
71 #    C Include's.  Don't alter these                             #
72 ##################################################################
```

```
73
74 quote '#include <stdio.h>'
75 quote '#include "isa.h"'
76 quote '#include "pipeline.h"'
77 quote '#include "stages.h"'
78 quote '#include "sim.h"'
79 quote 'int sim_main(int argc, char *argv[]);'
80 quote 'int main(int argc, char *argv[]){return sim_main(argc,argv);}'
81
82 ##################################################################
83 #    Declarations.  Do not change/remove/delete any of these       #
84 ##################################################################
85
86 ##### Symbolic representation of Y86-64 Instruction Codes
#############
87 wordsig INOP  'I_NOP'
88 wordsig IHALT 'I_HALT'
89 wordsig IRRMOVQ 'I_RRMOVQ'
90 wordsig IIRMOVQ 'I_IRMOVQ'
91 wordsig IRMMOVQ 'I_RMMOVQ'
92 wordsig IMRMOVQ 'I_MRMOVQ'
93 wordsig IOPQ  'I_ALU'
94 wordsig IJXX  'I_JMP'
95 wordsig ICALL 'I_CALL'
96 wordsig IRET  'I_RET'
97 wordsig IPUSHQ  'I_PUSHQ'
98 wordsig IPOPQ 'I_POPQ'
99 # Instruction code for iaddq instruction
100 wordsig IIADDQ  'I_IADDQ'
101
102 ##### Symbolic represenations of Y86-64 function codes
#####
103 wordsig FNONE    'F_NONE'        # Default function code
104
105 ##### Symbolic representation of Y86-64 Registers referenced
#####
106 wordsig RRSP     'REG_RSP'           # Stack Pointer
107 wordsig RNONE    'REG_NONE'          # Special value indicating "no
register"
108
109 ##### ALU Functions referenced explicitly ##########################
110 wordsig ALUADD  'A_ADD'        # ALU should add its arguments
111
112 ##### Possible instruction status values                      #####
113 wordsig SBUB  'STAT_BUB'  # Bubble in stage
114 wordsig SAOK  'STAT_AOK'  # Normal execution
115 wordsig SADR  'STAT_ADR'  # Invalid memory address
116 wordsig SINS  'STAT_INS'  # Invalid instruction
117 wordsig SHLT  'STAT_HLT'  # Halt instruction encountered
118
119 ##### Signals that can be referenced by control logic ##############
120
121 ##### Pipeline Register F ##########################################
```

```
122
123 wordsig F_predPC 'pc_curr->pc'       # Predicted value of PC
124
125 ##### Intermediate Values in Fetch Stage ############################
126
127 wordsig imem_icode  'imem_icode'      # icode field from instruction
memory
128 wordsig imem_ifun   'imem_ifun'       # ifun  field from instruction
memory
129 wordsig f_icode 'if_id_next->icode'  # (Possibly modified)
instruction code
130 wordsig f_ifun  'if_id_next->ifun'   # Fetched instruction function
131 wordsig f_valC  'if_id_next->valc'   # Constant data of fetched
instruction
132 wordsig f_valP  'if_id_next->valp'   # Address of following
instruction
133 boolsig imem_error 'imem_error'      # Error signal from instruction
memory
134 boolsig instr_valid 'instr_valid'    # Is fetched instruction valid?
135
136 ##### Pipeline Register D ########################################
137 wordsig D_icode 'if_id_curr->icode'   # Instruction code
138 wordsig D_rA 'if_id_curr->ra'      # rA field from instruction
139 wordsig D_rB 'if_id_curr->rb'      # rB field from instruction
140 wordsig D_valP 'if_id_curr->valp'     # Incremented PC
141
142 ##### Intermediate Values in Decode Stage  ########################
143
144 wordsig d_srcA   'id_ex_next->srca'  # srcA from decoded instruction
145 wordsig d_srcB   'id_ex_next->srcb'  # srcB from decoded instruction
146 wordsig d_rvalA 'd_regvala'      # valA read from register file
147 wordsig d_rvalB 'd_regvalb'      # valB read from register file
148
149 ##### Pipeline Register E ########################################
150 wordsig E_icode 'id_ex_curr->icode'   # Instruction code
151 wordsig E_ifun  'id_ex_curr->ifun'    # Instruction function
152 wordsig E_valC  'id_ex_curr->valc'    # Constant data
153 wordsig E_srcA  'id_ex_curr->srca'    # Source A register ID
154 wordsig E_valA  'id_ex_curr->vala'    # Source A value
155 wordsig E_srcB  'id_ex_curr->srcb'    # Source B register ID
156 wordsig E_valB  'id_ex_curr->valb'    # Source B value
157 wordsig E_dstE 'id_ex_curr->deste'    # Destination E register ID
158 wordsig E_dstM 'id_ex_curr->destm'    # Destination M register ID
159
160 ##### Intermediate Values in Execute Stage ########################
161 wordsig e_valE 'ex_mem_next->vale'  # valE generated by ALU
162 boolsig e_Cnd 'ex_mem_next->takebranch' # Does condition hold?
163 wordsig e_dstE 'ex_mem_next->deste'      # dstE (possibly modified to
be RNONE)
164
165 ##### Pipeline Register M                  ########################
166 wordsig M_stat 'ex_mem_curr->status'     # Instruction status
167 wordsig M_icode 'ex_mem_curr->icode'  # Instruction code
```

```
168 wordsig M_ifun  'ex_mem_curr->ifun' # Instruction function
169 wordsig M_valA  'ex_mem_curr->vala'      # Source A value
170 wordsig M_dstE 'ex_mem_curr->deste' # Destination E register ID
171 wordsig M_valE  'ex_mem_curr->vale'      # ALU E value
172 wordsig M_dstM 'ex_mem_curr->destm' # Destination M register ID
173 boolsig M_Cnd 'ex_mem_curr->takebranch' # Condition flag
174 boolsig dmem_error 'dmem_error'         # Error signal from
instruction memory
175
176 ##### Intermediate Values in Memory Stage ##########################
177 wordsig m_valM 'mem_wb_next->valm'  # valM generated by memory
178 wordsig m_stat 'mem_wb_next->status'  # stat (possibly modified to be
SADR)
179
180 ##### Pipeline Register W ##########################################
181 wordsig W_stat 'mem_wb_curr->status'     # Instruction status
182 wordsig W_icode 'mem_wb_curr->icode'  # Instruction code
183 wordsig W_dstE 'mem_wb_curr->deste' # Destination E register ID
184 wordsig W_valE  'mem_wb_curr->vale'      # ALU E value
185 wordsig W_dstM 'mem_wb_curr->destm' # Destination M register ID
186 wordsig W_valM  'mem_wb_curr->valm' # Memory M value
187
188 ####################################################################
189 #    Control Signal Definitions.                                   #
190 ####################################################################
191
192 ################ Fetch Stage     ###################################
193
194 ## What address should instruction be fetched at
195 word f_pc = [
196   # Mispredicted branch.  Fetch at incremented PC
197   M_icode == IJXX && !M_Cnd : M_valA;
198   # Completion of RET instruction
199   W_icode == IRET : W_valM;
200   # Default: Use predicted value of PC
201   1 : F_predPC;
202 ];
203
204 ## Determine icode of fetched instruction
205 word f_icode = [
206   imem_error : INOP;
207   1: imem_icode;
208 ];
209
210 # Determine ifun
211 word f_ifun = [
212   imem_error : FNONE;
213   1: imem_ifun;
214 ];
215
216 # Is instruction valid?
217 bool instr_valid = f_icode in
218   { INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
```

```
219     IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ, IIADDQ };
220
221 # Determine status code for fetched instruction
222 word f_stat = [
223   imem_error: SADR;
224   !instr_valid : SINS;
225   f_icode == IHALT : SHLT;
226   1 : SAOK;
227 ];
228
229 # Does fetched instruction require a regid byte?
230 bool need_regids =
231   f_icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
232          IIRMOVQ, IRMMOVQ, IMRMOVQ, IIADDQ };
233
234 # Does fetched instruction require a constant word?
235 bool need_valC =
236   f_icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX, ICALL, IIADDQ };
237
238 # Predict next value of PC
239 word f_predPC = [
240   f_icode in { IJXX, ICALL } : f_valC;
241   1 : f_valP;
242 ];
243
244 ################ Decode Stage ######################################
245
246
247 ## What register should be used as the A source?
248 word d_srcA = [
249   D_icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ  } : D_rA;
250   D_icode in { IPOPQ, IRET } : RRSP;
251   1 : RNONE; # Don't need register
252 ];
253
254 ## What register should be used as the B source?
255 word d_srcB = [
256   D_icode in { IOPQ, IRMMOVQ, IMRMOVQ, IIADDQ } : D_rB;
257   D_icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
258   1 : RNONE;  # Don't need register
259 ];
260
261 ## What register should be used as the E destination?
262 word d_dstE = [
263   D_icode in { IRRMOVQ, IIRMOVQ, IOPQ, IIADDQ } : D_rB;
264   D_icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
265   1 : RNONE;  # Don't write any register
266 ];
267
268 ## What register should be used as the M destination?
269 word d_dstM = [
270   D_icode in { IMRMOVQ, IPOPQ } : D_rA;
271   1 : RNONE;  # Don't write any register
```

```
];

## What should be the A value?
## Forward into decode stage for valA
word d_valA = [
  D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
  d_srcA == e_dstE : e_valE;    # Forward valE from execute
  d_srcA == M_dstM : m_valM;    # Forward valM from memory
  d_srcA == M_dstE : M_valE;    # Forward valE from memory
  d_srcA == W_dstM : W_valM;    # Forward valM from write back
  d_srcA == W_dstE : W_valE;    # Forward valE from write back
  1 : d_rvalA;  # Use value read from register file
];

word d_valB = [
  d_srcB == e_dstE : e_valE;    # Forward valE from execute
  d_srcB == M_dstM : m_valM;    # Forward valM from memory
  d_srcB == M_dstE : M_valE;    # Forward valE from memory
  d_srcB == W_dstM : W_valM;    # Forward valM from write back
  d_srcB == W_dstE : W_valE;    # Forward valE from write back
  1 : d_rvalB;  # Use value read from register file
];

################ Execute Stage ######################################

## Select input A to ALU
word aluA = [
  E_icode in { IRRMOVQ, IOPQ } : E_valA;
  E_icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IIADDQ } : E_valC;
  E_icode in { ICALL, IPUSHQ } : -8;
  E_icode in { IRET, IPOPQ } : 8;
  # Other instructions don't need ALU
];

## Select input B to ALU
word aluB = [
  E_icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
         IPUSHQ, IRET, IPOPQ, IIADDQ } : E_valB;
  E_icode in { IRRMOVQ, IIRMOVQ } : 0;
  # Other instructions don't need ALU
];

## Set the ALU function
word alufun = [
  E_icode == IOPQ : E_ifun;
  1 : ALUADD;
];

## Should the condition codes be updated?
bool set_cc = E_icode in { IOPQ, IIADDQ } &&
  # State changes only during normal operation
  !m_stat in { SADR, SINS, SHLT } && !W_stat in { SADR, SINS, SHLT };

```

```
## Generate valA in execute stage
## Load forwarding: forward valM from memory stage for rmmovq/pushq
word e_valA = [
  E_icode in { IRMMOVQ, IPUSHQ } && E_srcA == M_dstM : m_valM;
  1 : E_valA;    # Pass valA through stage
];

## Set dstE to RNONE in event of not-taken conditional move
word e_dstE = [
  E_icode == IRRMOVQ && !e_Cnd : RNONE;
  1 : E_dstE;
];

################ Memory Stage ######################################

## Select memory address
word mem_addr = [
  M_icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : M_valE;
  M_icode in { IPOPQ, IRET } : M_valA;
  # Other instructions don't need address
];

## Set read control signal
bool mem_read = M_icode in { IMRMOVQ, IPOPQ, IRET };

## Set write control signal
bool mem_write = M_icode in { IRMMOVQ, IPUSHQ, ICALL };

#/* $begin pipe-m_stat-hcl */
## Update the status
word m_stat = [
  dmem_error : SADR;
  1 : M_stat;
];
#/* $end pipe-m_stat-hcl */

## Set E port register ID
word w_dstE = W_dstE;

## Set E port value
word w_valE = W_valE;

## Set M port register ID
word w_dstM = W_dstM;

## Set M port value
word w_valM = W_valM;

## Update processor status
word Stat = [
  W_stat == SBUB : SAOK;
  1 : W_stat;
];
```

```
378
379################ Pipeline Register Control ########################
380
381# Should I stall or inject a bubble into Pipeline Register F?
382# At most one of these can be true.
383bool F_bubble = 0;
384bool F_stall =
385  # Conditions for a load/use hazard (relaxed for load forwarding)
386  E_icode in { IMRMOVQ, IPOPQ } &&
387  (
388    E_dstM == d_srcB ||
389    (E_dstM == d_srcA && !(D_icode in { IRMMOVQ, IPUSHQ }))
390  ) ||
391  # Stalling at fetch while ret passes through pipeline
392  IRET in { D_icode, E_icode, M_icode };
393
394# Should I stall or inject a bubble into Pipeline Register D?
395# At most one of these can be true.
396bool D_stall =
397  # Conditions for a load/use hazard (relaxed for load forwarding)
398  E_icode in { IMRMOVQ, IPOPQ } &&
399  (
400    E_dstM == d_srcB ||
401    (E_dstM == d_srcA && !(D_icode in { IRMMOVQ, IPUSHQ }))
402  );
403
404bool D_bubble =
405  # Mispredicted branch
406  (E_icode == IJXX && !e_Cnd) ||
407  # Stalling at fetch while ret passes through pipeline
408  # but not condition for a load/use hazard (relaxed for load
forwarding)
409  !(
410    E_icode in { IMRMOVQ, IPOPQ } &&
411    (
412      E_dstM == d_srcB ||
413      (E_dstM == d_srcA && !(D_icode in { IRMMOVQ, IPUSHQ }))
414    )
415  ) &&
416  IRET in { D_icode, E_icode, M_icode };
417
418# Should I stall or inject a bubble into Pipeline Register E?
419# At most one of these can be true.
420bool E_stall = 0;
421bool E_bubble =
422  # Mispredicted branch
423  (E_icode == IJXX && !e_Cnd) ||
424  # Conditions for a load/use hazard (relaxed for load forwarding)
425  E_icode in { IMRMOVQ, IPOPQ } &&
426  (
427    E_dstM == d_srcB ||
428    (E_dstM == d_srcA && !(D_icode in { IRMMOVQ, IPUSHQ }))
429  );
```

```
430
431# Should I stall or inject a bubble into Pipeline Register M?
432# At most one of these can be true.
433bool M_stall = 0;
434# Start injecting bubbles as soon as exception passes through memory
stage
435bool M_bubble = m_stat in { SADR, SINS, SHLT } || W_stat in { SADR,
SINS, SHLT };
436
437# Should I stall or inject a bubble into Pipeline Register W?
438bool W_stall = W_stat in { SADR, SINS, SHLT };
439bool W_bubble = 0;
440#/* $end pipe-all-hcl */
```