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 ...
```

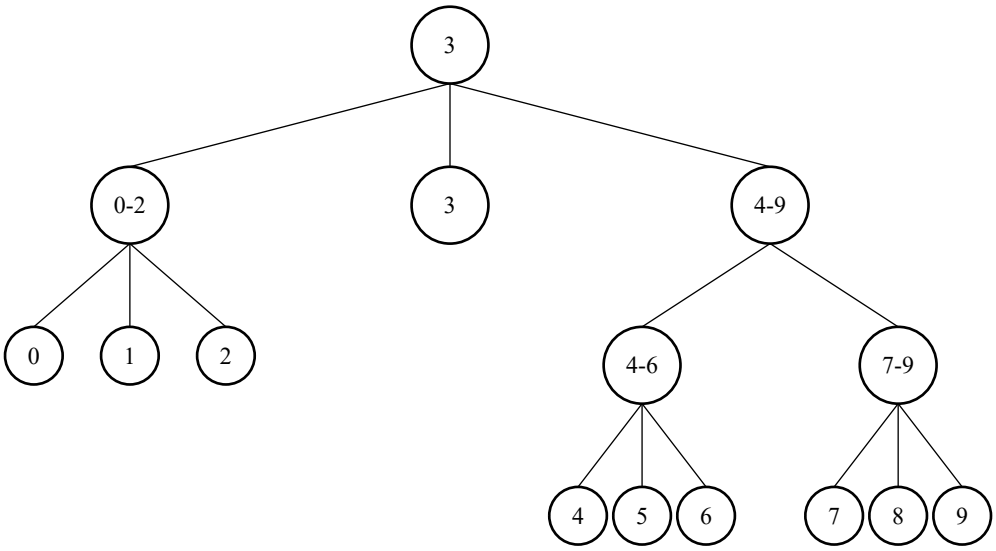
循环尾部更新指针并判断是否继续：

```
    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：分治搜索余数处理

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，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。通过采用分治搜索结构，我们可以减少平均分支次数，从而降低因分支预测失败而导致的性能损失。

4.4.1 分治搜索结构



此时最坏情况下需要 3 次比较，假设每种情况的概率相同，则平均需要 2.5 次比较，相较于线性处理，分支次数显著降低。

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，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，R9 直接 fall-through 到 R8，R8 到 R7，以此类推直到 R1，避免多余跳转指令：

```
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
```

```
9       rmmovq %r8, 56(%rsi)
10      # ... 依此类推 ...
```

4.5 优化策略四：加载转发（Load Forwarding）

4.5.1 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。

```
1   mrmovq (%rdi), %r8      # Memory 阶段产生 valM
2   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”，因为当 `rmmovq` 在 Decode 阶段时，`mrmovq` 刚进入 Memory 阶段，数据尚未就绪。

4.5.2 突破点

分析发现 `rmmovq` 实际在 **Memory 阶段**才使用 `valA`（写入内存），而非 Execute 阶段。当 `rmmovq` 到达 Execute 阶段时，前一条 `mrmovq` 已完成 Memory 阶段，`m_valM` 可用。因此可在 Execute 阶段新增转发路径。

4.5.3 HCL 修改

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

```
1   ## Execute 阶段: 加载转发
2   word e_valA = [
3       # rmmovq/pushq 且 srcA 冲突时, 直接转发 m_valM
4       E_icode in { IRMMOVQ, IPUSHQ } && E_srcA == M_dstM : m_valM;
5       1 : E_valA;     # 默认使用原值
6   ];
7
8   ## 放宽冒险检测条件 (仅 srcB 冲突或 srcA 冲突且非 rmmovq/pushq 时才暂停)
9   bool F_stall =
10      E_icode in { IMRMOVQ, IPOPQ } &&
11      (
12          E_dstM == d_srcB ||
13          (E_dstM == d_srcA && !(D_icode in { IRMMOVQ, IPUSHQ }))
14      ) ||
15      IRET in { D_icode, E_icode, M_icode };
```

类似修改应用于 `D_stall`、`D_bubble`、`E_bubble`。

4.6 代码大小优化

初始实现代码为 1005 字节，超出限制。将 `jmp Done` 替换为 `ret` 节省 8 字节（`jmp` 占 9 字节，`ret` 占 1 字节），最终代码大小为 **997 字节**。

4.7 测试结果

PIPE 回归测试通过：

`[占位符：report_7.png - PIPE 回归测试截图]`

ptest 测试套件通过（含 iaddq）：

`[占位符：report_8.png - ptest 测试截图]`

正确性测试通过（68/68）：

`[占位符：report_9.png - correctness.pl 测试截图]`

Benchmark 结果：

`[占位符：report_10.png - benchmark.pl 结果截图]`

4.8 优化效果汇总

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

最终 CPE 为 **7.46**，优于官方文档提到的最佳结果（7.48），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 分治搜索余数处理

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，通过将余数处理从线性结构改为分治搜索结构，平均分支次数从 5 次降低至 2–3 次。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，是实现 CPE < 7.5 的关键。

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

6.1 sum.ys

```
.pos 0
# sum.ys - Iteratively 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 sum_list
    ret

sum_list:
    pushq %rbx
    xorq %rax, %rax
    jmp test

loop:
    mrmovq (%rdi), %rbx
    addq %rbx, %rax
    mrmovq 8(%rdi), %rdi

test:
    andq %rdi, %rdi
    jne loop

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
    ret

.pos 0x200
stack:
```

6.4 seq-full.hcl

完整源码见 `sim/seq/seq-full.hcl`，主要修改点已在 Part B 部分详细说明。

6.5 ncopy.ys

完整源码见 `sim/pipe/ncopy.ys`，主要优化策略已在 Part C 部分详细说明。

6.6 pipe-full.hcl

完整源码见 `sim/pipe/pipe-full.hcl`，主要包括 `iaddq` 指令支持和加载转发优化，已在 Part B 和 Part C 部分详细说明。