计算机组成原理 ArchLab 实验报告

姓名：李孙木铎　**学号**：2023010304　**班级**：软件 31

2025 年 12 月 23 日

1 实验简介

本实验旨在深入理解流水线 Y86-64 处理器的设计与实现，并通过优化基准程序和处理器架构来最大化性能。实验分为三个部分：Part A 熟悉 Y86-64 汇编编程；Part B 扩展 SEQ 模拟器以支持新指令 `iaddq`；Part C 则是实验的核心，通过修改 `ncopy.ys` 和 `pipe-full.hcl` 来优化流水线性能（CPE）。

2 Part A: Y86-64 汇编指令编程

2.1 任务描述

Part A 要求实现三个简单的 Y86-64 程序，分别是：

- `sum.ys`: 迭代计算链表元素之和。
- `rsum.ys`: 递归计算链表元素之和。
- `copy.ys`: 将源内存块复制到目标内存块，并计算 checksum（异或和）。

2.2 实现思路

- 由于 Y86-64 指令集是一般 x86-64 的简化版本，也即 x86-64 的子集，因此可以参考 C 语言的实现逻辑，将其逐步翻译为汇编指令；
- 可以调用 gcc 的汇编工具，将原本的 C 代码编译为 x86-64 汇编代码，分析关键逻辑部分的汇编实现；
- 之后，结合 Y86-64 指令集的限制，进行相应的指令替换和逻辑调整，最终实现所需功能。

2.3 实现细节

2.3.1 sum.ys (迭代求和)

该程序实现了迭代版本的链表求和。

- **参数传递**：链表头指针 ls 通过 %rdi 传递。
- **循环逻辑**：使用 andq %rdi, %rdi 检查当前节点是否为空。若非空，则读取 (%rdi) 到临时寄存器并累加到 %rax，然后更新 %rdi 为 8(%rdi)（即 next 指针），直到链表结束。
- **栈空间与数据**：参考实验文档，将待测数据加载在数据段中，并按照规范合理调用主函数，分配栈空间。

实验结果：

2.3.2 rsum.ys (递归求和)

该程序实现了递归版本的链表求和。

- **保存现场**：由于是递归调用，必须将 val（当前节点值）保存在被调用者保存寄存器（如 %rbx）中，并在调用前将 %rbx 压栈保存。
- **递归逻辑**：
  1. 若 ls 为空，直接跳转至返回 0 。
  2. 否则，取出当前值存入 %rbx，加载 next 指针到 %rdi，调用 call rsum_list。
  3. 将两个值相加后存入返回值寄存器 %rax，并恢复现场（弹出栈中的 %rbx）。

2.3.3 copy.ys (块复制与 Checksum)

该程序实现了内存块复制及异或校验和计算。

- **参数**：src (%rdi), dest (%rsi), len (%rdx)。
- **常量加载**：由于 Y86-64 不支持直接的立即数加载，因此先使用 irmovq 将常量加载到闲置寄存器（如 %r8）中备用。
- **逻辑**：使用循环结构，每次从 src 读取一个字，写入 dest，并与 %rax 进行异或运算。之后，src 和 dest 指针增加 8，len 减 1。

2.4 实验结果

三个程序经 yas 编译器和 yis 模拟器后，输出结果如图 1所示：

```
thu@ubuntu:~/ArchLab/archlab-handout/sim/misc$ ./yas sum.ys
thu@ubuntu:~/ArchLab/archlab-handout/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
%r10:   0x0000000000000000      0x0000000000000c00

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

(a) sum.ys 运行结果

```
thu@ubuntu:~/ArchLab/archlab-handout/sim/misc$ ./yas rsum.ys
thu@ubuntu:~/ArchLab/archlab-handout/sim/misc$ ./yis rsum.yo
Stopped in 37 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:
0x01c0: 0x0000000000000000      0x0000000000000086
0x01c8: 0x0000000000000000      0x00000000000000b0
0x01d0: 0x0000000000000000      0x0000000000000086
0x01d8: 0x0000000000000000      0x000000000000000a
0x01e0: 0x0000000000000000      0x0000000000000086
0x01f0: 0x0000000000000000      0x000000000000005b
0x01f8: 0x0000000000000000      0x0000000000000013
```

(b) rsum.ys 运行结果

```
thu@ubuntu:~/ArchLab/archlab-handout/sim/misc$ ./yas copy.ys
thu@ubuntu:~/ArchLab/archlab-handout/sim/misc$ ./yis copy.yo
Stopped in 42 steps at PC = 0x13.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%rax:   0x0000000000000000      0x0000000000000cba
%rcx:   0x0000000000000000      0x0000000000000cba
%rsp:   0x0000000000000000      0x0000000000000200
%rsi:   0x0000000000000000      0x0000000000000048
%rdi:   0x0000000000000000      0x0000000000000030
%r8:    0x0000000000000000      0x0000000000000001
%r9:    0x0000000000000000      0x0000000000000008

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

(c) copy.ys 运行结果

图 1: Part A 三个程序的运行结果

可见，最终 %rax 中的值为 0x0000000000000cba，即链表元素之和（或内存块校验和）；且对内存块复制的结果，目的内存区的数据与源内存区一致，均符合预期。

3 Part B: 在 SEQ 处理器中实现 iaddq 指令

3.1 任务描述

Part B 要求在顺序处理器（SEQ）的硬件描述语言 `seq-full.hcl` 中实现 `iaddq V, rB` 指令。该指令的功能是 `R[rB] ← R[rB] + V`。由于 Y86-64 指令集中没有直接支持立即数加法的指令，因此实现该指令后，可以减少 `irmovq` 指令的使用频率，从而提升程序性能。

3.2 iaddq 指令逻辑分析

参考 CSAPP 练习题 4.3，`iaddq` 指令总长度为 10 字节，编码格式如下：

- **指令编码：** `C 0`（1 字节）;
- **寄存器编码：** `F rB`（1 字节，`rA` 字段无效，填 `F`）;
- **立即数：** `V`（8 字节，64 位有符号整数）.

各阶段的计算逻辑如表 1所示：

表 1: SEQ 处理器中 iaddq 指令在各个阶段的计算逻辑

Stage	**iaddq V, rB**
Fetch	icode:ifun $\leftarrow M_1$[PC]
	rA:rB $\leftarrow M_1$[PC + 1]
	valC $\leftarrow M_8$[PC + 2]
	valP $\leftarrow$ PC + 10
Decode	valB $\leftarrow R$[rB]
Execute	valE $\leftarrow$ valB + valC
	Set CC
Memory	
Write Back	R[rB] $\leftarrow$ valE
PC Update	PC $\leftarrow$ valP

3.3 HCL 实现细节

在 seq-full.hcl 中，我添加了 IIADDQ 到相应的控制信号集合中：

```
# Fetch Stage
bool instr_valid = icode in { INOP, IHALT, ..., IIADDQ };
bool need_regids = icode in { IRRMOVQ, ..., IIADDQ };
bool need_valC   = icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IIADDQ };

# Decode Stage
srcB = [
    icode in { IOPQ, IRMMOVQ, IMRMOVQ, IIADDQ } : rB;
    ...
];
dstE = [
    icode in { IIRMOVQ, IOPQ, IIADDQ } : rB;
    ...
];

# Execute Stage
aluA = [
    icode in { IIADDQ, IIRMOVQ, IMRMOVQ, IRMMOVQ } : valC;
    ...
];
aluB = [
    icode in { IIRMOWQ, IMRMOVQ, IOPQ, IIADDQ } : valB;
    ...
];
set_cc = icode in { IOPQ, IIADDQ };
```

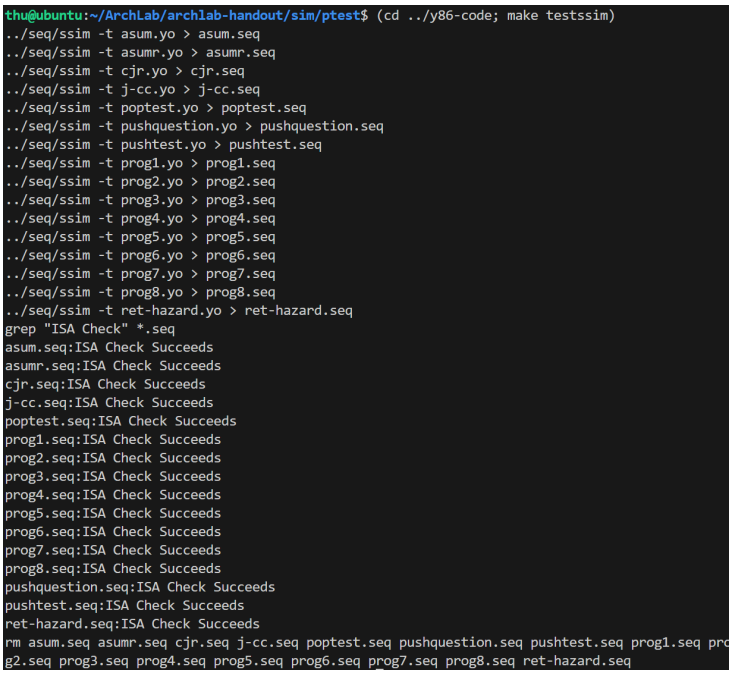
3.4 测试验证

使用工具测试 iaddq 指令的正确性，并验证整个处理器的功能完整性。测试结果如图 2所示：

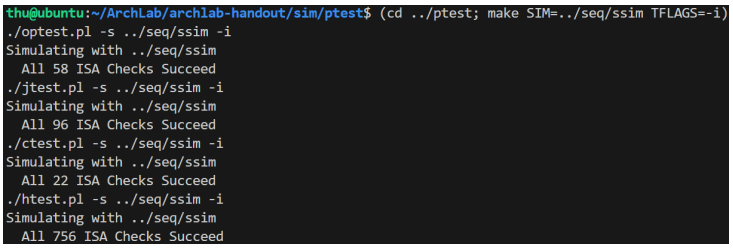
4 Part C: 流水线处理器设计与代码性能优化

4.1 任务描述

Part C 的目标首先是修改 pipe-full.hcl，使流水线处理器支持 iaddq 指令；其次是优化 ncopy.ys ，使其在复制内存块并统计正数个数时的性能（CPE, Cycles Per Element）最优。目标 CPE 为 7.50 以下。



(a) 验证标准指令集完整性



(b) 验证 iaddq 指令正确性

图 2: Part B iaddq 指令测试结果

4.2 优化策略分析

为了达到极致的性能，我主要采用了以下几种优化手段：

4.2.1 循环展开 (Loop Unrolling)

循环展开可以在一个循环中处理多个元素，提高寄存器利用率，进而提高指令级并行性，减少分支跳转带来的控制冒险。经分析可知，对 15 个寄存器，一共有 11 个寄存器可以用于并行进行内存加载和存储操作，因此最多可以展开到 11 路。但考虑后续的余数处理，选择 8 路或 9 路展开更为合适。

4.2.2 三叉树余数处理 (Ternary Tree Remainder Handling)

循环展开后，剩余元素（0-8 个）的处理成为短数组性能的瓶颈。简单的线性查找效率太低。由于条件跳转指令可以基于比较结果跳转到三组不同的标签，因此可以设计一棵三叉搜索树来快速定位剩余长度的处理入口。（选择 9 路展开，对三叉树结构更为

自然；如果选择 8 路展开，则需要构建深度为 3 的二叉树，复杂度更高。）基于 iaddq 判断余数大小，设计三叉搜索树：

- 首先判断 len < 3，若是则跳转到处理 0-2 个元素的标签；再判断 len < 6，若是则跳转到处理 3-5 个元素的标签；否则跳转到处理 6-8 个元素的标签；
- 对每个标签内，继续使用 iaddq 判断具体的剩余数量，并跳转到对应的单个处理块。
- 对处理部分，采用倒序阶梯式处理，降低跳转指令的数量。

4.2.3 预加载技术 (Pre-loading)

- **问题**：由于加载-使用冒险，在加载指令 mrmovq 后，紧接着使用该数据的指令（如 andq）会引入一个气泡。
- **解决**：可以使用一个不影响条件码的指令（如 je）来填补这个气泡，从而保持流水线的连续性。在余数处理的过程中，将所有的 mrmovq 指令都放在跳转指令之前，也即进行数据的预加载，就可以有效减少气泡的产生。
- **效果**：只应用三叉树和循环展开，CPE 可降至约 7.65；加入预加载后，CPE 进一步降至约 7.49，达到满分要求。

4.2.4 瀑布流处理 (Waterfall Structure)

对于剩余元素的具体处理，构建了一个紧凑的阶梯式处理结构：每个处理块（RK）在处理完第 K 个元素后，会加载第 K-1 个元素到寄存器，之后代码会自然执行到下一个处理块（Handle K-1）。这个设计减少了跳转指令的使用，同时预防了加载-使用冒险。

4.3 代码实现概览

以下是 ncopy.ys 的核心代码片段：

```
Loop:
    # 9-way Unrolling with "Sandwich" Optimization
    mrmovq (%rdi), %r8
    # ... (Load all 9 values) ...
    mrmovq 64(%rdi), %rbx

    rmmovq %r8, (%rsi)   # Store Val 0
    # ... (Store all 9 values) ...
    rmmovq %rbx, 72(%rsi) # Store Val

Npos0:
```

```
    andq %r8, %r8        # val0 <= 0?
    jle Npos1            # if so, goto Npos0:
    iaddq $1, %rax       # count++
Npos1:
    # ... (Repeat for all 9 values) ...
Npos9:
    # Loop update
    iaddq $72, %rdi          # src += 8 * 10
    iaddq $72, %rsi          # dst += 8 * 10
    iaddq $-9, %rdx          # len -= 10
    jge Loop                 # if so, goto Loop:

# Remainder Handling with Pre-loading
Remainder:
    iaddq $6, %rdx           # <0 (-9..-7 -> 剩0..2)
    jl Rem_0_2

Rem_3_8: # rdx now ranges from 0 to 5
    iaddq $-3, %rdx          # <0 (0..2 -> 剩3..5)
    jl Rem_3_5

Rem_6_8: # rdx now ranges from 0 to 2
    iaddq $-1, %rdx          # rdx now ranges from -1 to 1
    mrmovq 40(%rdi), %r13
    jl R6                    # <0 (剩6)
    mrmovq 48(%rdi), %r14
    je R7                    # =0 (剩7)
    mrmovq 56(%rdi), %rbx
    jg R8                    # >0 (剩8)
# ... (Similar handling for Rem_3_5 and Rem_0_2) ...

R8: # handle 8th remaining element
    andq %rbx, %rbx
    rmmovq %rbx, 56(%rsi)
    mrmovq 48(%rdi), %r14
    jle R7
    iaddq $1, %rax
# ... (Similar handling for R7, R6, R5, R4, R3, R2, R1) ...
```

4.4 pipe-full.hcl 的修改

为了支持 ncopy.ys 中的优化，需在 pipe-full.hcl 中实现了 iaddq 指令。各流水线阶段的实现逻辑与关键 HCL 信号如表 2所示：

表 2: 流水线模式下 iaddq 指令的各阶段实现逻辑与 HCL 信号

Stage	**Operation Logic**	**Key HCL Signals**
Fetch	icode:ifun $\leftarrow M_1$[PC]	`instr_valid = 1`
	rA:rB $\leftarrow M_1$[PC + 1]	`need_regids = 1`
	valC $\leftarrow M_8$[PC + 2]	`need_valC = 1`
	valP $\leftarrow$ PC + 10	
Decode	valB $\leftarrow R$[rB]	`d_srcB = D_rB`
	dstE $\leftarrow$ rB	`d_dstE = D_rB`
Execute	valE $\leftarrow$ valB + valC	`aluA = E_valC`
	Set CC	`aluB = E_valB`
		`alufun = ALUADD`
		`set_cc = 1`
Memory	*No Operation*	`mem_read = 0`
		`mem_write = 0`
Write Back	R[dstE] $\leftarrow$ valE	(*Handled by pipeline*)

4.5 实验结果

编译构建 PIPE 处理器，经过正确性测试和性能测试，最终结果如图 4.5所示：且程序编译后字节长度为 967 字节，符合实验要求。

5 总结

- 通过 Y86-64 汇编编程，更深入地理解了汇编语言与底层计算机体系结构的关系，同时熟悉了 Y86-64 指令集的特点与限制。
- 通过在 SEQ 处理器中实现 `iaddq` 指令，理解了处理器实现指令集的基本流程与关键技术，并了解了硬件描述语言 HCL 的使用。

```
thu@ubuntu:~/ArchLab/archlab-handout/sim/pipe$ ./correctness.pl
57      OK
58      OK
59      OK
60      OK
61      OK
62      OK
63      OK
64      OK
128     OK
192     OK
256     OK
68/68 pass correctness test
```

(a) 正确性测试结果

```
thu@ubuntu:~/ArchLab/archlab-handout/sim/pipe$ ./benchmark.pl
55      345     6.27
56      356     6.36
57      360     6.32
58      371     6.40
59      379     6.42
60      380     6.33
61      388     6.36
62      399     6.44
63      400     6.35
64      400     6.25
Average CPE     7.49
Score   60.0/60.0
```

(b) 性能测试结果

图 3: Part C 正确性与性能测试结果

- 通过在 PIPE 处理器中实现 `iaddq` 指令，理解了 PIPE 处理器流水线设计的特点和原理，并掌握了流水线处理器中的指令实现方法。
- 通过对 `ncopy.ys` 的优化，深入理解了流水线处理器中的性能瓶颈与优化策略，掌握了循环展开、指令调度、预加载等底层优化技术。