计算机系统结构 - Arch Lab 实验报告

UNIkeEN

目录

1 实验目的

本 Lab 旨在通过对基准程序和流水线处理器做优化，提升其性能，以此学习五级流水线 Y86 处理器的设计与实现，从中认识代码与硬件相互作用对程序性能的影响。

任务目标 对 `sim/pipe/ncopy.ys` 做保留原有语义的修改，这段 Y86 汇编代码实现的是将长度为 `len` 的整数数组从 `src` 复制到 `dst`，并返回 `src` 中包含的正整数的数量。

1. `ncopy.ys` 函数需适用于任意数组大小，且在任意数组大小的测试集上都通过正确性认证。
2. `ncopy` 文件编译为 `yo` 文件后长度不能超过 1000 字节。
3. 可以修改 Y86 处理器的流水线硬件逻辑定义 `pipe-full.hcl` 以取得加速效果，修改后需要通过回归测试

2 思路与实现

基准程序的平均 CPE 为 **15.18**，我将通过以下的顺序与方式展开优化。

2.1 指令合并

整数运算 Y86 本身不包含立即数的整数运算指令，由实验文档可得，TA 为 Y86 增加了一条整数运算指令（`iaddq`），可以把立即数和寄存器变量做加法并回存到寄存器中。

由此，可以将以下此类代码

```
1    irmovq $1, %r10
2    addq %r10, %rax      # count++
```

合并为

```
1    iaddq $1, %rax       # count++
```

同理 `irmovq + subq` 也可优化为 `iaddq`（操作数为负数）。基准程序中共有三处可以进行合并优化。

比较条件码 `iaddq` 指令在完成整数运算的同时会设置条件码，故可将以下此类代码

```
1    iaddq $-1, %rdx
2    andq %rdx, %rdx
3    jl ...
```

合并为

```
    iaddq $-1, %rdx
    jl ...
```

经过以上两种优化后，平均 CPE 将降为 **11.81**，接下来的过程均将使用上述合并。

2.2 指令顺序替换

基准程序中存在数据冒险。如：

```
    mrmovq (%rdi), %r10 # read val from src...
    rmmovq %r10, (%rsi) # ...and store it to dst
```

第一条指令从内存中读出寄存器 %rdi 的值并存入 %r10，而下一条指令需要 %r10 的值作为源操作数。流水线（如无旁路）会将下一条指令暂停一个周期，导致执行阶段中插入一个气泡。

解决方案是调换指令顺序。将后续无相关性的指令提前，如下：

```
    mrmovq (%rdi), %r10 # read val from src...
    andq %r10, %r10
    rmmovq %r10, (%rsi) # ...and store it to dst
```

优化后平均 CPE 将降为 **11.57**，之后过程会尽可能调换指令顺序以避免气泡。

2.3 循环展开

循环展开通常可以提升程序的性能。自二路起分别测试循环展开，可知一定范围内展开次数越多，程序性能越好（在寄存器数量足够时，增加展开次数的代码只需简单堆叠，对每一次展开使用一个新的寄存器即可）。以下是八路循环展开中复制部分的实现：

```
    LoopEight:
    mrmovq (%rdi), %r8
    mrmovq 8(%rdi),%r9
    ...
    mrmovq 56(%rdi),%rcx
    iaddq $64, %rdi      # src+=8

    rmmovq %r8, (%rsi)
    rmmovq %r9,8(%rsi)
    ...
    rmmovq %rcx,56(%rsi)
    iaddq $64, %rsi      # dst+=8
```

由于八路循环展开分别使用的是不同的中间寄存器，此时无需担心数据冒险。

复制部分结束后，逐一检查中间寄存器存储值是否为正整数，若是，则 count 自增，若不是则进入下一项检查。检查部分代码如下：

```
CheckOne:
    andq %r8, %r8
    jle CheckTwo
    iaddq $1, %rax
CheckTwo:
    andq %r9, %r9
    jle CheckThree
    iaddq $1, %rax
CheckThree:
    ...
CheckEight:
    andq %rcx, %rcx
    jle Next
    iaddq $1, %rax
Next:                   # check for the next loop (len>=8?)
    iaddq $-8, %rdx
    jge LoopEight
```

检查完毕和第一次循环前，通过 iaddq + jge 判断 len 是否大于等于循环展开的次数 8，若是则进入下一次循环，若不是则跳转至剩余部分。

剩余部分对剩余小于 8 个元素进行复制和计数。首先恢复由上述过 iaddq 指令已变为负数的 len。如果剩余元素为 0 则结束，不为 0 则继续复制。

```
RemainStart:
    iaddq $8, %rdx
    jg LoopRemain
    ret
LoopRemain:
    iaddq $-1, %rdx
    ...
Npos:
    andq %rdx,%rdx
    jg LoopRemain       # if so, goto Loop:
```

如果在 LoopRemain 中直接复用基准程序的循环代码，优化后平均 CPE 为 **8.53**。

2.4 剩余部分决策树

观察 benchmark.pl 的输出，发现此时对平均 CPE 的影响主要来自于数组大小小于 8 的情况，接下来对剩余部分进行优化。

可以根据剩余部分的个数建立决策树，通过多次 iaddq 算得剩余部分的 len。考虑到数组越小 CPE 越大，建立的决策树左大右小以尽可能优化平均 CPE。

循环展开 8 路，剩余部分个数 0-7 共八种情况，设计决策树如下：

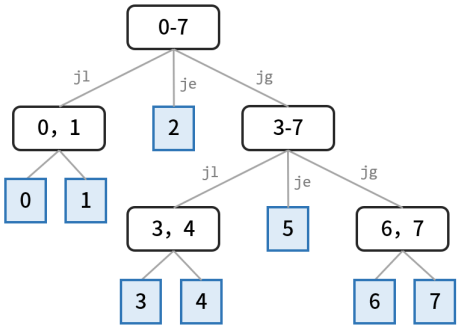


图 1: 决策树

按递减顺序，为剩余元素分别建立复制与检查计数的代码段（第一段对第七个剩余元素操作，第二段对第六个，以此类推），决策树得到 len 后直接跳转到对应代码段开始，具体实现如下：

```
RemainStart:
    iaddq $6, %rdx
    jl Branch0_1
    je Remain2          # len=2, go to Remain2
    jg Branch3_7
Branch0_1:
    iaddq $1, %rdx
    je Remain1          # len=1, go to Remain1
    ret                 # len=0, return
Branch3_7:
    iaddq $-3, %rdx
    je Remain5          # len=5, go to Remain5
    jg Branch6_7
    iaddq $1, %rdx
    je Remain4          # len=4, go to Remain4
    jne Remain3         # len=3, go to Remain3
Branch6_7:
    iaddq $-2, %rdx
```

```
    jne Remain6         # len=6, go to Remain6
Remain7:
    mrmovq 48(%rdi),%r8
    andq %r8,%r8        # *(src+6)>0?
    rmmovq %r8,48(%rsi)
Remain6:
    mrmovq 40(%rdi),%r8
    jle Remain6s        # *(src+6)<=0, skip count+1
    iaddq $1,%rax
Remain6s:
    rmmovq %r8,40(%rsi)
    andq %r8,%r8
...
Remain1:
    mrmovq (%rdi),%r8
    jle Remain1s
    iaddq $1,%rax
Remain1s:
    rmmovq %r8,(%rsi)
    andq %r8,%r8
    jle Done
    iaddq $1,%rax
```

注：在 Remain7 中通过 andq 判断 %r8 后，其对应的条件转移指令并不紧随其后，而是位于 Remain6 中（见注释）。同理，Remain6s 中 andq 对应的条件转移指令位于 Remain5 中，之后亦然。只有 Remain1s 中 andq 对应的条件转移指令紧随其后，这样的设计是为了避免数据冒险。

将以上决策树配合八路循环展开，优化后平均 CPE 可以达到 **7.90**。

修改循环展开为十路（寄存器数量仍然足够，故不需要复用寄存器并处理冒险），修改决策树如下后平均 CPE 可以达到 **7.81**。

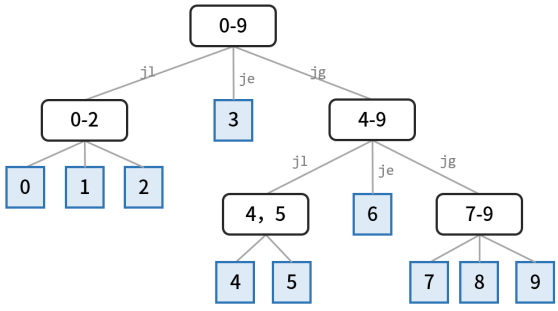


图 2: 修改后决策树

2.5 分支预测

根据分支预测的相关知识，在有多个分支指令时，可以将跳转可能性较大的分支指令放在前面。如下面这段代码

```
    jl Branch0_2
    je Remain3
    jg Branch4_9
```

可以优化为

```
    jg Branch4_9
    jl Branch0_2
    je Remain3
```

原程序中有两处此类优化，优化后平均 CPE 可以达到 **7.79**。

2.6 其他优化

本次实验初始时 %rax 为 0，ncopy 部分不再需要使用 xorq 为 count 赋初值。且基准程序该部分中对 len 是否为 0 的比较和后续代码重复，删除之。

原始代码:

```
ncopy:
    xorq %rax,%rax
    andq %rdx,%rdx
    jg Start
    ret
Start:
    iaddq $-10, %rdx
    jl RemainStart
```

修改后:

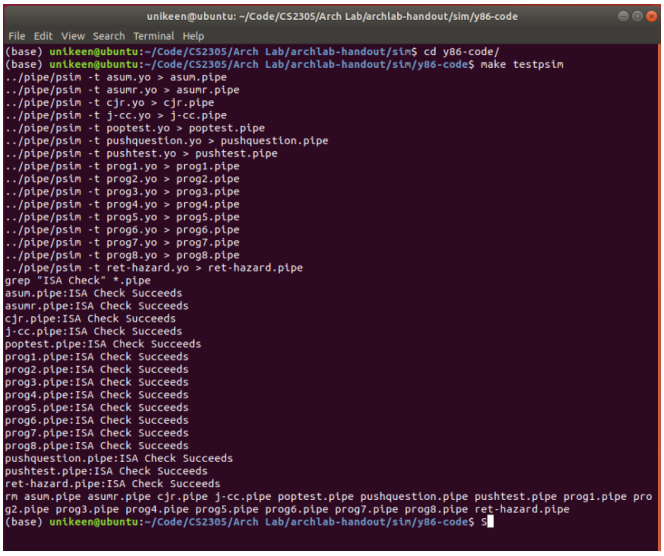
```
ncopy:
    iaddq $-10, %rdx
    jl RemainStart
```

优化后平均 CPE 可以达到 **7.50**，达到满分水平。

3 运行截图

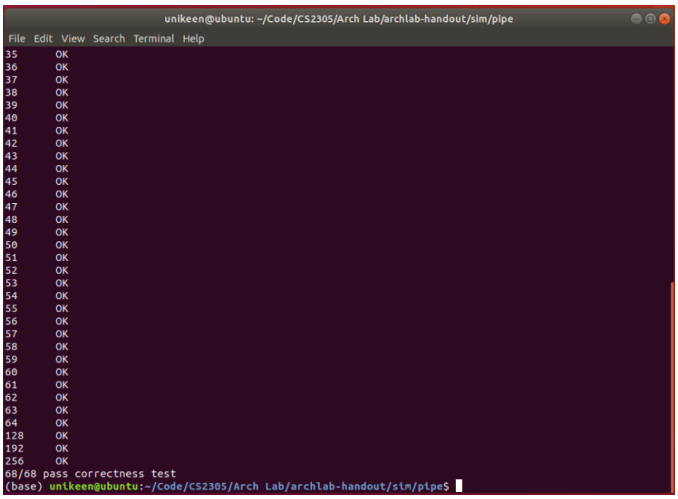
3.1 可行性测试

实验手册所要求的可行性测试截图如下，结果均为通过：



(a) 基准测试

(b) 回归测试



(c) 正确性测试

图 3: 可行性测试

3.2 CPE 得分

最终平均 CPE 为 **7.50**，执行 `benchmark.pl` 的截图如下：

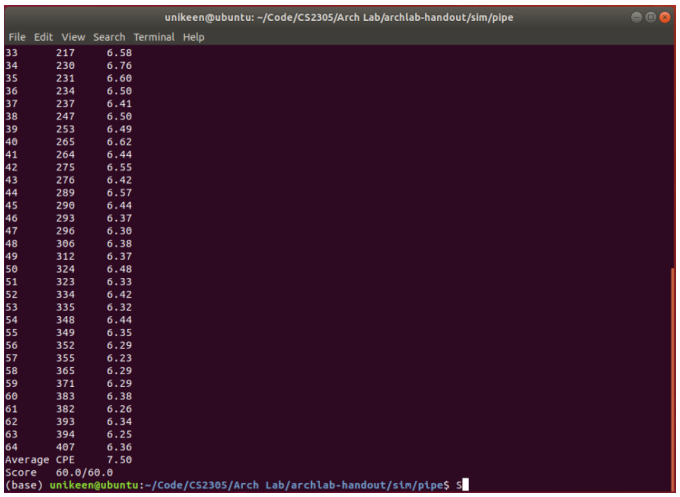


图 4: 平均 CPE 得分

3.3 长度检查

实验手册要求 `ncopy` 的汇编版本不得超过 1000 字节，执行相关命令得到最终长度为 998 字节，符合实验要求。截图如下：

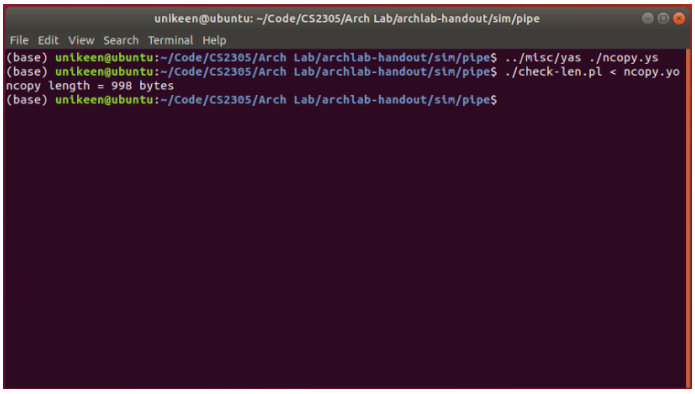


图 5: 长度检查

附录 A 全部代码

A.1 ncopy.ys 中的代码

```
1 #/* $begin ncopy-ys */
2 ##############################################################################
3 # ncopy.ys - Copy a src block of len words to dst.
```

```
# Return the number of positive words (>0) contained in src.
#
# UNIkeEN
#
# Describe how and why you modified the baseline code.
#
##################################################################
# Do not modify this portion
# Function prologue.
# %rdi = src, %rsi = dst, %rdx = len
ncopy:

##################################################################
# You can modify this portion
    iaddq $-10, %rdx
    jl RemainStart
LoopTen:
    mrmovq (%rdi), %r8
    mrmovq 8(%rdi),%r9
    mrmovq 16(%rdi),%r10
    mrmovq 24(%rdi),%r11
    mrmovq 32(%rdi),%r12
    mrmovq 40(%rdi),%r13
    mrmovq 48(%rdi),%r14
    mrmovq 56(%rdi),%rcx
    mrmovq 64(%rdi),%rbx
    mrmovq 72(%rdi),%rbp
    iaddq $80, %rdi     # src+=10
    rmmovq %r8, (%rsi)
    rmmovq %r9,8(%rsi)
    rmmovq %r10,16(%rsi)
    rmmovq %r11,24(%rsi)
    rmmovq %r12,32(%rsi)
    rmmovq %r13,40(%rsi)
    rmmovq %r14,48(%rsi)
    rmmovq %rcx,56(%rsi)
    rmmovq %rbx,64(%rsi)
    rmmovq %rbp,72(%rsi)
    iaddq $80, %rsi     # dst+=10
```

```
43 CheckOne:
44     andq %r8, %r8
45     jle CheckTwo
46     iaddq $1, %rax
47 CheckTwo:
48     andq %r9, %r9
49     jle CheckThree
50     iaddq $1, %rax
51 CheckThree:
52     andq %r10, %r10
53     jle CheckFour
54     iaddq $1, %rax
55 CheckFour:
56     andq %r11, %r11
57     jle CheckFive
58     iaddq $1, %rax
59 CheckFive:
60     andq %r12, %r12
61     jle CheckSix
62     iaddq $1, %rax
63 CheckSix:
64     andq %r13, %r13
65     jle CheckSeven
66     iaddq $1, %rax
67 CheckSeven:
68     andq %r14, %r14
69     jle CheckEight
70     iaddq $1, %rax
71 CheckEight:
72     andq %rcx, %rcx
73     jle CheckNine
74     iaddq $1, %rax
75 CheckNine:
76     andq %rbx, %rbx
77     jle CheckTen
78     iaddq $1, %rax
79 CheckTen:
80     andq %rbp, %rbp
81     jle Next
```

```
82      iaddq $1, %rax
83  Next:
84      iaddq $-10, %rdx
85      jge LoopTen
86
87  RemainStart:
88      iaddq $7, %rdx
89      jg Branch4_9
90      jl Branch0_2
91      je Remain3
92  Branch0_2:
93      iaddq $2, %rdx
94      je Remain1
95      iaddq $-1, %rdx
96      je Remain2
97      ret
98  Branch4_9:
99      iaddq $-3, %rdx
100     jg Branch7_9
101     je Remain6
102     iaddq $1, %rdx
103     je Remain5
104     jne Remain4
105 Branch7_9:
106     iaddq $-2, %rdx
107     jl Remain7
108     je Remain8
109 Remain9:
110     mrmovq 64(%rdi),%r8
111     andq %r8,%r8
112     rmmovq %r8,64(%rsi)
113 Remain8:
114     mrmovq 56(%rdi),%r8
115     jle Remain8s
116     iaddq $1,%rax
117 Remain8s:
118     rmmovq %r8,56(%rsi)
119     andq %r8,%r8
120 Remain7:
```

```
121     mrmovq 48(%rdi),%r8
122     jle Remain7s
123     iaddq $1,%rax
124 Remain7s:
125     rmmovq %r8,48(%rsi)
126     andq %r8,%r8
127 Remain6:
128     mrmovq 40(%rdi),%r8
129     jle Remain6s
130     iaddq $1,%rax
131 Remain6s:
132     rmmovq %r8,40(%rsi)
133     andq %r8,%r8
134 Remain5:
135     mrmovq 32(%rdi),%r8
136     jle Remain5s
137     iaddq $1,%rax
138 Remain5s:
139     rmmovq %r8,32(%rsi)
140     andq %r8,%r8
141 Remain4:
142     mrmovq 24(%rdi),%r8
143     jle Remain4s
144     iaddq $1,%rax
145 Remain4s:
146     rmmovq %r8,24(%rsi)
147     andq %r8,%r8
148 Remain3:
149     mrmovq 16(%rdi),%r8
150     jle Remain3s
151     iaddq $1,%rax
152 Remain3s:
153     rmmovq %r8,16(%rsi)
154     andq %r8,%r8
155 Remain2:
156     mrmovq 8(%rdi),%r8
157     jle Remain2s
158     iaddq $1,%rax
159 Remain2s:
```

```
160     rmmovq %r8,8(%rsi)
161     andq %r8,%r8
162 Remain1:
163     mrmovq (%rdi),%r8
164     jle Remain1s
165     iaddq $1,%rax
166 Remain1s:
167     rmmovq %r8,(%rsi)
168     andq %r8,%r8
169     jle Done
170     iaddq $1,%rax
171 ##################################################################
172 # Do not modify the following section of code
173 # Function epilogue.
174 Done:
175     ret
176 ##################################################################
177 # Keep the following label at the end of your function
178 End:
179 #/* $end ncopy-ys */
```

A.2 pipe-full.hcl 中的代码

硬件代码相对于 TA 增加了 `iaddq` 后的版本，没有做进一步修改，在此省略。