Project 1: Optimizing the Performance of a Pipelined Processor

520021910863, Heda Chen, marcythm@sjtu.edu.cn
520030910287, Xinming Shu, Xinming_Shu@sjtu.edu.cn

April 23, 2022

1 Introduction

This project consists of three parts, which are listed below:

1. In Part A, we need to translate three functions in `sim/misc/examples.c` which are written in C into Y86 assembly program.
2. In Part B, the requirement is to extend the implementation of the given sequential Y86 processor to support the `iaddl` instruction.
3. In Part C, the requirement is to optimize both the given pipelined Y86 processor and `ncopy.ys` to improve the performance of the assembly program. We tried several approaches, such as loop unrolling and rearrange the order of instructions to avoid hazards, and finally achieved a CPE of 9.17.

Additionally, all of the three parts have passed the tests given, and all assembly programs are annotated in detail.

In the process of finishing the project,

1. Chen first wrote all the code, then Shu did the same independently, checked Chen's code for correctness and discussed some details.
2. Shu wrote the main part of the report, Chen revised some details and reformatted the report in LaTeX.

2 Experiments

2.1 Part A

2.1.1 Analysis

This part is a 'warm-up' session of this project. In this part, we need to translate three functions in `sim/misc/examples.c` which are written in C into Y86 assembly program. The key points and core techniques used are listed as follows:

1. Follow the Y86 calling conventions, e.g. take care to save the value of callee-saved registers (`%ebx, %ebp`) in function, pass function arguments using `%edi, %esi, %edx...`.
2. Use stack properly to save caller-saved registers when calling a subroutine.
3. Mimic the functionality of C, divide the code based on functionality with sufficient and clear labels.

For more details, see the comments in the code section.

2.1.2 Code

1. Core part of `sum.ys`

```
main:
    irmovl ele1, %edi    # following the calling conventions, use %rdi
    call   sum_list      # (here %edi) to store the first argument
    ret
sum_list:
    irmovl 0, %eax       # initialize `sum`: `sum = 0`
    jmp    check_condition # jump into loop, and check loop condition first: `while (ls)`
L3:
    mrmovl (%edi), %ecx  # load the value of current element: `tmp = *(ls->val)` (because
                         # in Y86 the addition can only be performed between registers)
    addl   %ecx, %eax    # add the value to `sum`: `sum += tmp`
    mrmovl 4(%edi), %edi # move the pointer to the next element: `ls = ls->next`
check_condition:
    andl   %edi, %edi    # check if current pointer is 0: `ls == NULL`
    jne    L3            # if not, jump to L3
    ret
```

2. Core part of `rsum.ys`

```
main:
    irmovl ele1, %edi    # following the calling conventions, use %rdi
    call   rsum_list     # (here %edi) to store the first argument
    ret
rsum_list:
    andl   %edi, %edi    # check if current pointer is 0 (the end of the list): `if (!ls)`
    je     L3            # if so, terminate the recursion: `return 0`
    pushl  %edi          # save the pointer of current element
    mrmovl 4(%edi), %edi # move pointer to the next element
    call   rsum_list     # calculate the sum of the list starting from the next element
    popl   %edi          # restore the pointer of current element
    mrmovl (%edi), %ecx  # load the value of current element: `tmp = *(ls->val)` (because
                         # in Y86 the addition can only be performed between registers)
    addl   %ecx, %eax    # add the value of current element to get the sum of current list
    ret
L3:
    irmovl 0, %eax       # when terminate the recursion, set counter to 0: `sum = 0`
    ret
```

3. Core part of `copy.ys`

```
main:
    irmovl src, %edi     # following the calling conventions, use %rdi,
    irmovl dest, %esi    # %rsi and %rdx (here %edi, %esi and %edx) to
    irmovl 3, %edx       # store the first three arguments
    call   copy_block
    ret
copy_block:
    pushl  %ebx          # save %ebx
    irmovl 0, %eax       # initialize checksum: `result = 0`
    jmp    check_condition # jump into loop, and check loop condition first: `while (ls)`
L3:
```

```
    mrmovl (%edi), %ebx  # load current value in source block: `val = *src`
    rmmovl %ebx, (%esi)  # store the value into dest block: `*dest = val`
    xorl   %ebx, %eax    # add current value into checksum: `result ^= val`
    irmovl 4, %ecx       # use %ecx as a constant 4
    addl   %ecx, %edi    # move to next value in source block: `src++`
    addl   %ecx, %esi    # move to next value in dest block: `dest++`
    irmovl 1, %ecx       # use %ecx as a constant 1
    subl   %ecx, %edx    # decrease `len`: `len--`
check_condition:
    andl   %edx, %edx    # check if `len` is greater than zero: `while (len > 0)`
    jg     L3            # if not, continue to loop
    popl   %ebx          # restore %ebx
    ret
```

2.1.3 Evaluation

1. For `sum.ys` simulating `sum_list()`, as the figure represents, the return value `%eax` is `0xcba`, while none of the callee-saved registers are changed.

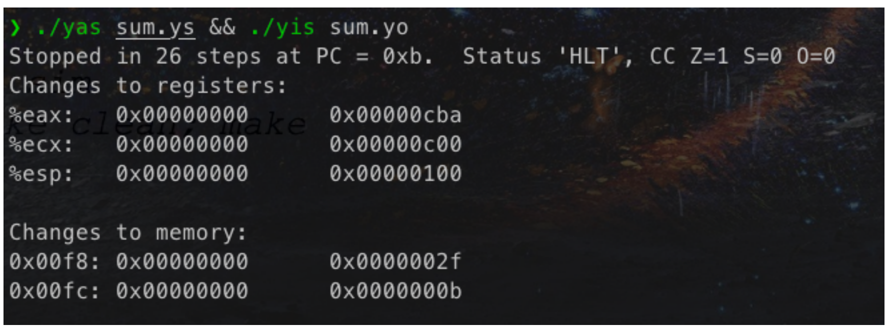

Figure 1: the test result of sum.ys

2. For `rsum.ys` simulating `rsum_list()`, as the figure represents, the return value `%eax` is `0xcba`, while none of the callee-saved registers are changed.

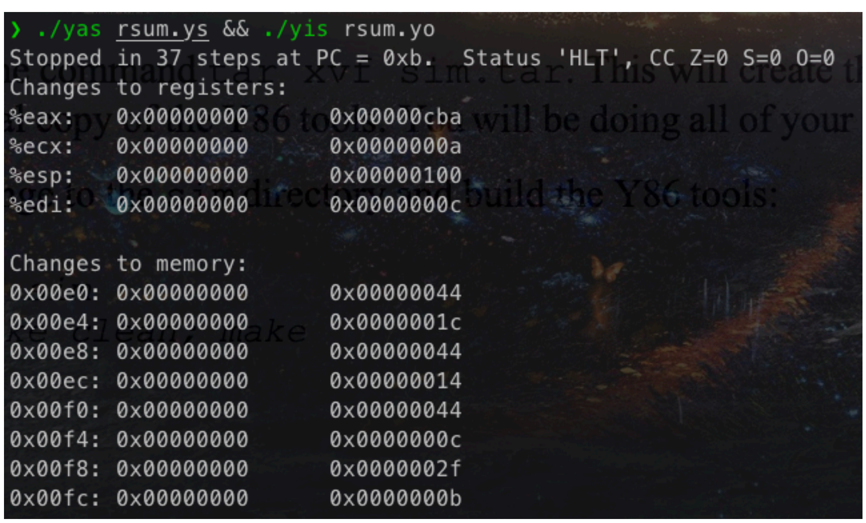

Figure 2: the test result of rsum.ys

3. For `copy.ys` simulating `copy_block()`, as the figure represents, all of the data are correctly copied, andthe returned checksum `%eax` is `0xcba`, while none of the callee-saved registers are changed.

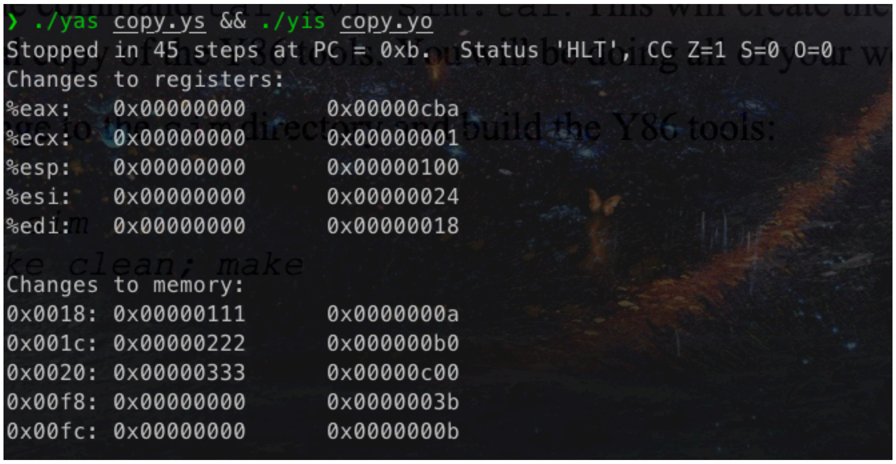

Figure 3: the test result of copy.ys

2.2 Part B

2.2.1 Analysis

In this part, we should extend the sequential Y86 processor by modifying `seq-full.hcl` to support `iaddl` instruction. The key points are listed as follows:

1. Understand the processor's logic and take a good command of the syntax of HCL.
2. Determine which signals should be modified to implement `iaddl` instruction.

The process of determining which signals should be modified is as follows:

1. In Fetch Stage, `IIADDL` should be added into `instr_valid`, `need_regid` and `need_valC` because it is a valid instruction, and also need `regid` and `valC`.
2. In Decode Stage, when icode is `IIADDL`, `srcB` is from `rB` since the second operand of `iaddl` is a register, `dstE` (where the result from ALU is passed towards) is `rB` since `iaddl imm, rB` means `rB += imm` (`rB` is updated).
3. In Execute Stage, add `IIADDL` ino the choices region of `set_cc` since `iaddl` operation involves ALU operation which will set conditional codes.
4. In Execute Stage, when `icode` is `IIADDL`, `aluA` (the first op) is `valC` (the immediate in the instruction) since `iaddl imm, rB` means the first op is `imm` (`valC`).
5. In Execute Stage, when `icode` is `IIADDL`, `aluB` (the second op) is `valB` (the value of the second register that is read) for the same reason above.

2.2.2 Code

Here only the modified parts of the code are listed.

```
bool instr_valid = icode in
  { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
         IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL, IIADDL };

```

```
# Does fetched instruction require a regid byte?
bool need_regids =
  icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
          IIRMOVL, IRMMOVL, IMRMOVL, IIADDL };

# Does fetched instruction require a constant word?
bool need_valC =
  icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL, IIADDL };

## What register should be used as the B source?
int srcB = [
  icode in { IOPL, IRMMOVL, IMRMOVL, IIADDL } : rB;
  icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
  1 : RNONE; # Don't need register
];

## What register should be used as the E destination?
int dstE = [
  icode in { IRRMOVL } && Cnd : rB;
  icode in { IIRMOVL, IOPL, IIADDL} : rB;
  icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
  1 : RNONE; # Don't write any register
];

## Select input A to ALU
int aluA = [
       icode in { IRRMOVL, IOPL } : valA;
       icode in { IIRMOVL, IRMMOVL, IMRMOVL, IIADDL } : valC;
       icode in { ICALL, IPUSHL } : -4;
       icode in { IRET, IPOPL } : 4;
       # Other instructions don't need ALU
];

## Select input B to ALU
int aluB = [
       icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
                    IPUSHL, IRET, IPOPL, IIADDL } : valB;
       icode in { IRRMOVL, IIRMOVL } : 0;
       # Other instructions don't need ALU
];

## Should the condition codes be updated?
bool set_cc = icode in { IOPL, IIADDL };
```

2.2.3 Evaluation

1. First, we test out solution on the Y86 program `asumi.ys`. Here the result is omitted because it takes too much space and is not good for the layout of the report.
2. Then we retest the solution using the benchmark programs. Here the result is omitted because of the same reason as above.
3. Finally we test the implementation of `iaddl` with regression tests. As the figure shows, our implementation passes all the ISA checks.

2.3 Part C

2.3.1 Analysis

In this part, we were asked to speed up the program `ncopy.ys` as much as possible by optimizing both the program itself and the pipelined Y86 processor's implementation. The key points and core techniques are listed as follows:

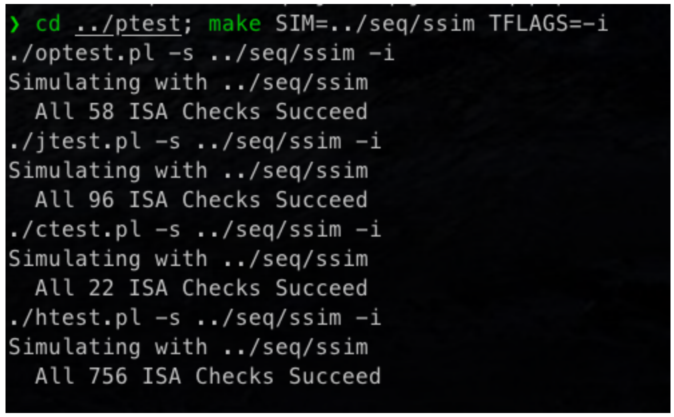

Figure 4: Part B Regression test

1. Modify `pipe-full.hcl` to support `iaddl` instruction. This part is similar to Part B, so I will not go into details.
2. Use `iaddl` instruction to replace the load-add case, e.g. replace `irmovl 1, %edi; addl %edi, %eax` with `iaddl 1, %eax`.
3. Perform loop unrolling to reduce the overhead of frequently modifying and comparing the subscript in the loop. Here we choose 7-way loop unrolling, as this maximizes the exploitation of the binary search technique used later.
4. Avoid load-and-use hazards. When a load-and-use hazard occurs, it will waste a clock cycle, which can be avoided and utilized by reordering instructions. Furthermore, we use two registers (`%esi, %edi`) to store the variable `val` in turn, loading them separately ahead of time.
5. Use binary search for remaining elements. For large inputs, it's better to unroll the loops for more ways. However, for small inputs, it is important to choose a good way to handle the remaining elements. For parts smaller than the number of ways, the easiest way is to write another loop for them. But there is another approach that performs better, that is, jumping to different positions for different number of remaining elements. Since Y86 does not support relative jump instructions, we use binary search to get the correct jump destination for each case.
   Additionally, here we choose to implement a binary search of depth 2, thus the maximum distinguishable number is $2^2 + 2^1 + 2^0 = 7$, so we use 7-way loop unrolling as above.

For more details, see the comments in the code section.

2.3.2 Code

1. 7-way loop unrolling part

```
##################################################################
# You can modify this portion
  xorl %eax,%eax  # count = 0;
  iaddl -6, %edx  # len <= 6?
  jle S06         # if so, goto S06

L0:
  mrmovl (%ebx), %esi  # tmp0 = src[0]
  mrmovl 4(%ebx), %edi # tmp1 = src[1]
  andl %esi, %esi      # if tmp0 <= 0
  rmmovl %esi, (%ecx)  # dst[0] = tmp0
  jle L1               # skip increasing counter
  iaddl 1, %eax        # else counter++
```

```
L1:
  mrmovl 8(%ebx), %esi # tmp0 = src[2]
  andl %edi, %edi      # if tmp1 <= 0
  rmmovl %edi, 4(%ecx) # dst[1] = tmp1
  jle L2               # skip increasing counter
  iaddl 1, %eax        # else counter++
L2:
  mrmovl 12(%ebx), %edi # tmp1 = src[3]
  andl %esi, %esi      # if tmp0 <= 0
  rmmovl %esi, 8(%ecx) # dst[2] = tmp0
  jle L3               # skip increasing counter
  iaddl 1, %eax        # else counter++
L3:
  mrmovl 16(%ebx), %esi # tmp0 = src[4]
  andl %edi, %edi      # if tmp1 <= 0
  rmmovl %edi, 12(%ecx) # dst[3] = tmp1
  jle L4               # skip increasing counter
  iaddl 1, %eax        # else counter++
L4:
  mrmovl 20(%ebx), %edi # tmp1 = src[5]
  andl %esi, %esi      # if tmp0 <= 0
  rmmovl %esi, 16(%ecx) # dst[4] = tmp0
  jle L5               # skip increasing counter
  iaddl 1, %eax        # else counter++
L5:
  mrmovl 24(%ebx), %esi # tmp0 = src[6]
  andl %edi, %edi      # if tmp1 <= 0
  rmmovl %edi, 20(%ecx) # dst[5] = tmp1
  jle L6               # skip increasing counter
  iaddl 1, %eax        # else counter++
L6:
  andl %esi, %esi      # if tmp0 <= 0
  rmmovl %esi, 24(%ecx) # dst[6] = tmp0
  jle ChkCond          # skip increasing counter
  iaddl 1, %eax        # else counter++
ChkCond:
  iaddl 28, %ebx       # src += 7 * sizeof(Byte)
  iaddl 28, %ecx       # dst += 7 * sizeof(Byte)
  iaddl -7, %edx       # len -= 7
  jg L0                # if len > 0, goto L0
```

2. Binary search for finding the number of remaing elements

```
S06:                   # len in (x-6: [0, 6] -> [-6, 0])
  iaddl 3, %edx        # (x-6: [0, 6] -> [-6, 0]) => (x-3: [0, 6] -> [-3, 3])
  jg S46               # len-3 > 0, len in [4, 6]
  je R3                # len-3 == 0, len = 3
S02:                   # len in (x-3: [0, 2] -> [-3, -1])
  iaddl 2, %edx        # (x-3: [0, 2] -> [-3, -1]) => (x-1: [0, 2] -> [-1, 1])
  jl R0                # len-1 < 0, len = 0
  je R1                # len-1 == 0, len = 1
  jmp R2               # len-1 > 0, len = 2
S46:                   # len in (x-3: [4, 6] -> [1, 3])
  iaddl -2, %edx       # (x-3: [4, 6] -> [1, 3]) => (x-5: [4, 6] -> [-1, 1])
  jl R4                # len-5 < 0, len = 4
  je R5                # len-5 == 0, len = 5
```

3. Unrolling of remaining loops

```
R6:
  mrmovl 20(%ebx), %esi # tmp = src[6]
  andl %esi, %esi      # if tmp <= 0
  rmmovl %esi, 20(%ecx) # dst[6] = tmp
```

```
  jle R5               # skip increasing counter
  iaddl 1, %eax        # else counter++
R5:
  mrmovl 16(%ebx), %esi # tmp = src[5]
  andl %esi, %esi      # if tmp <= 0
  rmmovl %esi, 16(%ecx) # dst[5] = tmp
  jle R4               # skip increasing counter
  iaddl 1, %eax        # else counter++
R4:
  mrmovl 12(%ebx), %esi # tmp = src[4]
  andl %esi, %esi      # if tmp <= 0
  rmmovl %esi, 12(%ecx) # dst[4] = tmp
  jle R3               # skip increasing counter
  iaddl 1, %eax        # else counter++
R3:
  mrmovl 8(%ebx), %esi # tmp = src[3]
  andl %esi, %esi      # if tmp <= 0
  rmmovl %esi, 8(%ecx) # dst[3] = tmp
  jle R2               # skip increasing counter
  iaddl 1, %eax        # else counter++
R2:
  mrmovl 4(%ebx), %esi # tmp = src[2]
  andl %esi, %esi      # if tmp <= 0
  rmmovl %esi, 4(%ecx) # dst[2] = tmp
  jle R1               # skip increasing counter
  iaddl 1, %eax        # else counter++
R1:
  mrmovl (%ebx), %esi  # tmp = src[1]
  andl %esi, %esi      # if tmp <= 0
  rmmovl %esi, (%ecx)  # dst[1] = tmp
  jle R0               # skip increasing counter
  iaddl 1, %eax        # else counter++
R0:
```

2.3.3 Evaluation

1. Run the given `correctness.pl` script to check correctness:

```
> ./correctness.pl
Simulating with instruction set simulator yis
  ncopy
0      OK
1      OK
2      OK
3      OK
# 61 lines omitted
64     OK
128    OK
192    OK
256    OK
68/68 pass correctness test
```

2. Run `benchmark.pl` script to check performance, and get a CPE of 9.17:

```
> ./benchmark.pl
  ncopy
0      39
1      47     47.00
2      58     29.00
3      53     17.67
# 59 lines omitted
62     422    6.81
63     424    6.73
64     435    6.80
```

```
Average CPE   9.17
Score  60.0/60.0
```

3 Conclusion

3.1 Problems

Several obstacles we encountered are as follows:

1. Most of the information available on the Internet is about the newer 64-bit Y86, which has some minor differences from the 32-bit Y86 used in this experiment, causing some troubles. For example, when Chen was writing some sample programs to help him understand Y86 ISA, the assembler kept reporting an confusing error `Invalid line`. After checking the program several times, he realized that he should change `xxxq` to `xxxl`, as the older CS:APP books used.

2. The assembler gives very little information about syntax error, which is hardly helpful for debugging. Chen often spends quite a long time, looking for where exactly the syntax error is. The example is the same as above.

3. When Chen started trying Part C, he first attempted to replace the load-add case with the `iaddl` instruction. Surprisingly, the benchmark script gives a CPE of 2 at this point (of course, the correctness script gave the positive result). Chen checked for a long time and even suspected that there was a problem with the benchmark script (and actually rewrote it). Finally he found out that the `iaddl` instruction was not implemented in `pipe-full.hcl`. After migrating the implementation from Part B, the CPE was back to normal level. (In fact this question is still unsolved: why could that version pass the correctness test? Is there any bug?)

3.2 Achievements

1. First of all, the most remarkable advantage of our solution should be the readability of the code. In every assembly program, almost every instruction is followed by a comment explaining what it does. (For HCL files, the original comments were clear enough, so we didn't add comment to them.)

2. Also, the good code readability is benificial for collaboration with partner. We use GitHub for code synchronization, and discussed details thoroughly when writing the report.

3. With respect to the performance of our solution, we have optimized the Part C code to maximize the use of the above techniques under limited conditions (depth = 2), achieving the CPE of 9.17. In fact Chen also investigated how to implement branch prediction, but did not attempt it after discovering that loop unrolling was sufficient enough to obtain a low CPE because of the hassle and lack of time.

4. In this experiment we successfully applied what we have learned in class and gained a deeper understanding.