A.1

Effective CPI = × Clock cycles for category categories

=(0.485 ) (1.0 ) +(0.367 ) (1.4 ) + (0.107 ) ( ( 0.6) (2.0 )+ (1– 0. 6) ( 1.5) ) + (0.03 )+ ( 1.2) =

1.24

A.2

CPI =1.0 ×51.1%+1.4×35.0% +2.0×11.0%×60%+ 1.5×11.0% ×40% +1.2×2.8%=1.23

A.3

ALU instructions: (17.8% + 3.0% + 0.6% + 5.6% + 1.0% + 0.9% + 4.1%) = 33.0%

Load-stores: (9.4% + 2.4%) = 11.8%

Conditional branches: 1.0%

Jumps: 0% FP add: (8.6% + 6.2%) = 14.8%

Load-store FP: (16.5% + 11.6%) = 28.1%

Other FP: (1.4% + 0.4% + 0.4% + 0.8%) = 3.0%

CPI = 1.0 × 33.0% + 1.4 × 11.8% + 2.0 × 1.0% × 60% + 1.5 × 1.0% × 40% + 6.0 × 8.2% + 4.0 × 14.8% + 20 × 0.2% + 1.5 × 28.1% + 2.0 × 3.0% = 2.12

A.4

ALU instructions: (30.2% + 1.2% + 1.2% + 3.7% + 6.8% + 0.2% + 0.4% + 1.0% + 1.6%) = 46.3% Load-stores: (16.0% + 1.4%) = 17.4%

Conditional branches: 7.0%

Jumps: 0% FP add: (3.4% + 1.4%) = 4.8%

Load-store FP: (11.7% + 4.4%) = 16.1%

Other FP: (0.8% + 0.4% + 0.3%) = 1.5%

CPI = 1.0 × 46.3% + 1.4 × 17.4% + 2.0 × 7.0% × 60% + 1.5 × 7.0% × 40% + 6.0 × 6.4% + 4.0 × 4.8% + 20 × 0.4% + 1.5 × 16.1% + 2.0 × 1.5% = 1.76

A.5

将语句2中的工作量从1增加到2。它将语句3中的功从减法改为否定，可能是一种节省。以上说明，如果要获得最好的结果，编写优化编译器意味着要结合复杂的权衡分析能力来控制任何优化步骤。

A.6

A.7

a.

ex\_a\_7: DADD R1,R0,R0 ; R0 = 0, initialize i = 0

SW 7000(R0),R1 ; store i

loop: LD R1,7000(R0) ; get value of i

DSLL R2,R1,#3 ; R2 = word offset of B[i]

DADDI R3,R2,#3000 ; add base address of B to R2

LD R4,0(R3) ; load B[i]

LD R5,5000(R0) ; load C

DADD R6,R4,R5 ; B[i] + C

LD R1,7000(R0) ; get value of i

DSLL R2,R1,#3 ; R2 = word offset of A[i]

DADDI R7,R2,#1000 ; add base address of A to R2

SD 0(R7),R6 ; A[i] ← B[i] + C

LD R1,7000(R0) get value of i

DADDI R1,R1,#1 ; increment i

SD 7000(R0),R1 ; store i

LD R1,7000(R0) ; get value of i

DADDI R8,R1,#-101 ; is counter at 101?

BNEZ R8,loop ; if not 101, repeat

Instructions executed = 2 + (16 × 101) = 1618

Memory-data references executed = 0 + (8 × 101) = 808

Instruction bytes = 4 × 18 = 72

b.

ex\_b\_7: movq $0x0,%rax # rax = 0, initialize i = 0

movq $0x0,%rbp # base pointer = 0

movq %rax,0x1b58(%rbp) # store i to location 7000 ($1b58)

movq 0x1388(%rbp),%rdx # load C from 5000 ($1388)

loop: movq 0x1b58(%rbp),%rax # get value of i

mov %rax,%rbx # rbx gets copy of i

shl $0x3,%rbx # rbx now has i \* 8

movq 0x0bb8(%rbx),%rcx # load B[i] (3000 = $0bb8) to rcx

add %rdx,%cdx # B[i] + C

mov %rax,%rbx # rbx gets copy of i

shl $0x3,%rbx # rbx now has i \* 8

movq %rcx,0x03e8(%rbx) # A[i] ← B[i] + C

(base address of A is 1000)

movq 0x1b58(%rbp),%rax # get value of i

add $0x1,%rax # increment i

movq %rax,0x1b58(%rbp) # save i

cmpq $0x0065,%rax # is counter at 101 ($0065)?

jae loop # if not 101, repeat

Instructions executed = 3 + (13 × 101) = 1316

A.8

a.

可能

b.

不可能

两个地址的指令编码为“00”到“01”。单地址指令可以用上两位编码为“11”，并使用“00000”到“11110”来区分31条单地址指令。然后，上面7位中的模式“11”和“11111”用于编码零地址指令，下面5位用于区分它们。我们只能编码其中的32种，而不是必须编码的35种;因此，这些指令编码是不可能的。

c.

在本部分中，我们已经像上面那样编码了三条双地址指令。此外，我们有24条零地址指令编码如下:在addr[9:5]字段中使用' 00000 '，在addr[4:0]字段中使用' 00000 '到' 10111 '。我们想要适应尽可能多的单一地址的指令。注意，我们不能利用上面7位中的“11”和“00000”的任何未使用的编码，因为我们需要使用整个addr[4:0]字段作为操作数的单个地址。因此，我们可以在addr[9:5]中使用' 00001 '到' 11111 '的编码作为单地址指令，并保存寄存器地址的最后5位。因为它包含31种模式，所以我们最多可以支持31条单地址指令。如果我们愿意，我们还可以增加8个额外的零地址指令。

A.9

a.

1) Stack: Push A // one address appears in the instruction, code size = 8 bits (opcode) + 64 bits (memory address) = 72 bits;

Push B // one address appears in the instruction, code size = 72 bits;

Add // zero address appears in the instruction, code size = 8 bits;

Pop C // one address appears in the instruction, code size = 72 bits;

Total code size = 72 + 72 + 8 + 72 = 224 bits.

2) Accumulator

Load A // one address appears in the instruction, code size = 8 bits (opcode) + 64 bits (memory address) = 72 bits;

Add B // one address appears in the instruction, code size = 72 bits; Store C // one address appears in the instruction, code size = 72 bits;

Total code size = 72 + 72 + 8 + 72 = 216 bits.

3) Register-memory

Load R1, A // two addresses appear in the instruction, code size = 8 bits (opcode) + 6 bits (register address) + 64 bits (memory address) = 78 bits;

Add R3, R1, B // three addresses appear in the instruction, code size = 8 bits (opcode) + 6 bits (register address) + 6 bits (register address) + 64 bits (memory address) = 84 bits;

Store R3,C // two addresses appear in the instruction, code size = 78 bits;

Total code size = 78 + 84 + 78 = 240 bits.

4) Register-register

Load R1, A // two addresses appear in the instruction, code size = 8 bits (opcode) + 6 bits (register address) + 64 bits (memory address) = 78 bits;

Load R2, B // two addresses appear in the instruction, code size = 78 bits;

Add R3, R1, R2 // three addresses appear in the instruction, code size = 8 bits (opcode) + 6 bits (register address) + 6 bits (register address) + 6 bits (register address) = 26 bits;

Store R3, C // two addresses appear in the instruction, code size = 78 bits;

Total code size = 78 + 78 + 26 + 78 = 260 bits.

b.

1) Stack

The total code size is 672/8 = 84 bytes. The number of bytes of data moved to or from memory is 576/8 = 72 bytes. There are 3 overhead instructions. The number of overhead data bytes is 24 bytes. A.10

2) Accumulator

The total code size is 576/8 = 72 bytes. The number of bytes of data moved to or from memory is 512/8 = 64 bytes. There is 1 overhead instruction. The number of overhead data bytes is 8 bytes.

3) Register-memory

The total code size is 564/8 = 71 bytes. The number of bytes of data moved to or from memory is 384/8 = 48 bytes. There is no overhead instruction. The number of overhead data bytes is 0 bytes.

4) Register-register

The total code size is 546/8 = 69 bytes. The number of bytes of data moved to or from memory is 384/8 = 48 bytes. There is no overhead instruction. The number of overhead data bytes is 0 bytes.

A.10

增加的原因有:

1.可以更自由地使用使用寄存器的编译技术，例如循环展开、公共子表达式消除和避免名称依赖。

2. 可以保存要传递给子程序的值的更多位置。

3.减少了存储和重新加载值的需要。

不增加的原因包括:

1.表示寄存器名需要更多的位，从而增加了指令的总体大小或减少了指令中其他字段的大小。

2. 在发生异常时要保存的CPU状态更多。

3.增加了芯片面积和电力消耗。

A.11

40 bytes in total

At least 44 bytes

On a 64-bit processor 48 bytes

A.12

A.13

A.14

A.15

A.16

A.17

A.18

Accumulator architecture code:

Load B ;Acc ← B

Add C ;Acc ← Acc + C

Store A ;Mem[A] ← Acc

Add C ;Acc ← "A” + C

Store B ;Mem[B] ← Acc Negate ;Acc ← − Acc

Add A ;Acc ← “− B” + A

Store D ;Mem[D] ← Acc

Memory-memory architecture code:

Add A, B, C ;Mem[A] ← Mem[B] + Mem[C]

Add B, A, C ;Mem[B] ← Mem[A] + Mem[C]

Sub D, A, B ;Mem[D] ← Mem[A] − Mem[B]

Stack architecture code: (TOS is top of stack, NTTOS is the next to the top of stack, and \* is the initial contents of TOS)

Push B ;TOS ← Mem[B], NTTOS ← \*

Push C ;TOS ← Mem[C], NTTOS ← TOS

Add ;TOS ← TOS + NTTOS, NTTOS ← \*

Pop A ;Mem[A] ← TOS, TOS ← \*

Push A ;TOS ← Mem[A], NTTOS ← \*

Push C ;TOS ← Mem[C], NTTOS ← TOS

Add ;TOS ← TOS + NTTOS, NTTOS ← \*

Pop B ;Mem[B] ← TOS, TOS ← \*

Push B ;TOS ← Mem[B], NTTOS ← \*

Push A ;TOS ← Mem[A], NTTOS ← TOS

Sub ;TOS ← TOS − NTTOS, NTTOS ← \*

Pop D ;Mem[D] ← TOS, TOS ← \*

Load-store architecture code:

Load R1,B ;R1 ← Mem[B]

Load R2,C ;R2 ← Mem[C]

Add R3,R1,R2 ;R3 ← R1 + R2 = B + C

Add R1,R3,R2 ;R1 ← R3 + R2 = A + C

Sub R4,R3,R1 ;R4 ← R3 − R1 = A − B

Store A,R3 ;Mem[A] ← R3

Store B,R1 ;Mem[B] ← R1

Store D,R4 ;Mem[D] ← R4

A.19

A.20

a.

ALU length = 16 bits

Data-reference length = 16 × 0.304 + 24 × (0.669 – 0.304) + 32 × (1.0 – 0.669) = 24.2 bits Branch-type length = 16 × 0.001 + 24 × (0.852 – 0.001) + 32 × (0.985 – 0.852) = 25.3 bits Average length = 16 × 0.47 + 24.2 × 0.38 + 25.3 × 0.15 = 20.5 bits

b.

c.

The per-instruction bytes fetched is (16 bits/instr)(0.47) + (40 bits/instr)(0.38 + 0.15) = 3.59 bytes/instr.

A.21

A.22

a.

|  |
| --- |
| 43 4F 4D 50 55 54 45 52  C O M P U T E R |

b.

|  |
| --- |
| 45 52 55 54 4D 50 43 4F  E R U T M P C O |

c.

4F4D, 5055, and 5455. Other misaligned 2-byte words would contain data from outside the given 64 bits.

d.

52 55 54 4D, 55 54 4D 50, and 54 4D 50 43. Other misaligned 4-byte words would contain data from outside the given 64 bits.

A.23

1)对于桌面，我们关心整数和浮点单位的速度，但不太关心功耗。我们还会考虑机器代码和运行在其上的软件(如x86代码)的兼容性。应当使ISA能够处理各种各样的应用，但商品定价对专门职能的资源投资造成限制。

2)对于服务器来说，数据库应用程序是典型的，因此ISA应该以整数计算为目标，并支持高吞吐量、多用户和虚拟化。ISA设计可以强调整数操作和内存操作。可增加动态电压和频率调整，降低能耗。

3)云计算ISAs应该为虚拟化提供支持。整数操作和内存管理将是重要的。对于大型数据中心来说，节省电力和便于系统监控的设施将是非常重要的。用户数据的安全性和加密非常重要，因此ISAs可以为这些问题提供支持。

4)功耗是嵌入式计算的一个大问题。价格也很重要。嵌入式ISAs可为具有特殊指令和/或寄存器的传感和控制设备、模拟到数字和数字到模拟转换器提供设施，以及接口到专门的、应用特定的功能。ISA可能省略浮点数，虚拟内存，缓存，或其他性能优化，以节省能源或成本。