Hw 2

A.7 a.

```
        dadd   $t0, $zero, $zero     # i=0
        sd     $a3, 7000($t0)        # store i in a3
        addi   $t1, $zero, 100       # t0 =100
loop:   ld     $a3, 7000($t0)        # get i
        dslt   $t2, $t1, $a3         # if 100<i
        bne    $t2, $zero, EXIT      # when 100<i, EXIT
        dsll   $t2, $a3, 3           # t2 = i*8
        daddi  $t3, $t2, 3000        # t3 = B + i*4
        ld     $a1, 0($t2)           # t3 = B[i]
        ld     $a2, 5000($t0)        # a2 = C
        dadd   $t2, $a1, $a2         # t2 = B[i]+C
        daddi  $a0, $t2, 1000        # a0 = A + i*4
        sd     $t3, 0($a0)           # A[i]=B[i]+C
        daddi  $a3, $a3, 1           # i++
        sd     $a3, 7000($t0)        # store i
        j      Loop
EXIT:
```

instructions: 3 + 13 × 101 + 3 = 1319

memory-data references: 1 + 5 × 101 = 506

code size in bytes: 16 × 32 / 8 = 64

b.

```
        movq   $0x0, %rax                 # i=0
        movq   $0x0, %rbp                 # base pointer=0
        movq   %rax, 0x1b58(%rbp)         # store i
loop:   movq   0x1b58(%rbp), %rax         # get i
        cmpq   $0x65, %rax                # i = 101
        je     EXIT                       # if i=101, EXIT
        mov    %rax, %rbx                 # rbx = i
        shl    $0x3, %rbx                 # rbx = i*8
        movq   0xbb8(%rbx), %rcx          # B[i]
        movq   0x1388(%rbp), %rdx         # get C
        add    %rdx, %cdx                 # B[i]+C
        mov    %rax, %rbx                 # rbx = i
        shl    $0x3, %rbx                 # rbx = i*8
        movq   %rcx, 0x3e8(%rbx)          # A[i] <- B[i]+C
        mov    %rax, %rbx                 # rbx=i
        add    $0x1, %rbx                 # i++
        movq   %rbx, 0x1b58(%rbp)         # save i
        jmp    loop
```

instructions: 3 + 15 × 101 + 3 = 1521

memory-data references: 1 + 5 × 101 = 506

code size in bytes: 18 × 32 / 8 = 72

A.18 a.

Stack:	Accumulator:	Memory-memory:	Load-store:
Push B	Load B	Add A, B, C ○ *	Load R1, B ○
Push C	Add C *	Add B, A, C ○ *	Load R2, C ○
Add *	Store A ○	Sub D, $\underset{\Delta}{A}$, $\overset{\Delta}{\underset{\Delta}{B}}$	Add R3, R1, R2 ○ *
Pop A ○	Add C △ *		Store A, R3 ☆
Push A	Store B ○ ☆		Add R1, R3, R2 ○ *
Push C △	Sub A *		store B, R1 ☆
Add *	Negate	A = C+C	Sub R4, R3, R1 ○ *
Pop B ○	Store D ☆	D = A - B	Store D, R4 ☆
Push B △		D = D + A	
Push A △			
Sub *			
Pop D			

b. A value is loaded from memory after having been loaded once: △
stack, Accumulator, Memory-memory

The result of one instruction is passed to another instruction as an operand: ○
stack, Accumulator, Memory-memory, Load-store

storage within the processor: *
storage in memory: ☆

c.

	Stack	Accumulator	Memory-memory	Load-store
instruction bytes be fetched	16/8 × 12 = 24	16/8 × 8 = 16	16/8 × 3 = 6	16/8 × 8 = 16
data be transferred from/to memory	16/8 × 9 = 18	16/8 × 4 = 8	16/8 × 9 = 18	16/8 × 5 = 10
total memory traffic	24 + 18 = 42	16+8 = 24	6+18 = 24	16+10 = 26

so Accumulator and Memory-memory are most efficient.

d.

	Stack	Accumulator	Memory-memory	Load-store
instruction bytes be fetched	64/8 × 12 = 96	64/8 × 8 = 64	64/8 × 3 = 24	64/8 × 8 = 64
data be transferred from/to memory	64/8 × 9 = 72	64/8 × 4 = 32	64/8 × 9 = 72	64/8 × 5 = 40
total memory traffic	96+72 = 168	64 + 32 = 96	24+72 = 96	64 + 40 = 104