02155 Computer Architecture and Engineering Fall 2019

Assignment 1

Group members:
s184235 - Tala Azrak
s170724 - Kristian Van Kints

This report contains ”21” pages

7-10-2019

Contents

Problems

1 Exercise A1.1 - Textbook problems

1.1 Problem 2.34

Code "A1.1.s"

1.2 Problem 2.39.1

To tell weither this design is a good choice or not, we need to calcutate the speed-up. First we find the number of arithmetic instruction after reducing.
arithmetic Instructions Reduced =

$500 \cdot 10^6 - ((500 \cdot 10^6) \cdot \frac{100}{25}) = 375 \cdot 10^6$

Now we know that :

CPI arithmetic instructions = 1
CPI loadstore instructions = 10
CPI branch instructions = 3

arithmetic instructions = $500 \cdot 10^6$
loadstore instructions = $300 \cdot 10^6$
branch instructions = $100 \cdot 10^6$

arithmetic Instructions Reduced = $375 \cdot 10^6$
Total instructions = $500 \cdot 10^6 + 300 \cdot 10^6 + 100 \cdot 10^6 = 900 \cdot 10^6$
Total instructions reducing = $375 \cdot 10^6 + 300 \cdot 10^6 + 100 \cdot 10^6 = 775 \cdot 10^6$

So we can find the the cpu clock cycle using the following formula:

CPU clock cycles=Instructions for a program X average clock cycles per instruction:

CPU clock cycles = $500 \cdot 10^6 \cdot 1 + 300 \cdot 10^6 \cdot 10 + 100 \cdot 10^6 \cdot 3 = 380 \cdot 10^6$
CPU clock cycles after reducing = $375 \cdot 10^6 \cdot 1 + 300 \cdot 10^6 \cdot 10 + 100 \cdot 10^6 \cdot 3 = 367 \cdot 10^6$
Now we find the CPI

$$CPI = \frac{CPUclockCycles}{InstructionCount} = \frac{380 \cdot 10^6}{900 \cdot 10^6} = 4.22$$

$$CPIReduced = \frac{CPUclockCycles}{InstructionCount} = \frac{367 \cdot 10^6}{775 \cdot 10^6} = 4.74$$

Now we find the cpu by using the following formula
CPU = Instrucntion count X CPI X Clock cycle time
CPU = $900 \cdot 10^6 \cdot 4.22 \cdot 10^{-9} = 3.789 seconds$

CPU Reduced = $775 \cdot 10^6 \cdot 4.74 \cdot 10^{-9} = 4.04085 seconds$

And the final step is to find the speed -up, $\frac{CPU Redcued}{CPU} = 0.9399012584 = \underline{\underline{94\%}}$

These powerful arithmetic instructions would be "a good design choice" if the program have a lot of complicated tasks, since the advanced instructions would be used often and we would see a shorter execution time, however in programs where only tasks like simple calculations is needed the advanced instructions would extend the execution time of the program thereby being "a bad design choice".

1.3 Problem 2.39.2

Total instructions = $\frac{500}{2} \cdot 10^6 + 300 \cdot 10^6 + 100 \cdot 10^6 = 650 \cdot 10^6$
Total instructions reducing = $\frac{500}{10} \cdot 10^6 + 300 \cdot 10^6 + 100 \cdot 10^6 = 450 \cdot 10^6$

So we can find the the cpu clock cycle using the following formula:

CPU clock cycles=Instructions for a program X average clock cycles per instruction:

CPU clock cycles = $\frac{500}{2} \cdot 10^6 \cdot 1 + 300 \cdot 10^6 \cdot 10 + 100 \cdot 10^6 \cdot 3 = 355 \cdot 10^6$
CPU clock cycles after reducing = $\frac{500}{10} \cdot 10^6 \cdot 1 + 300 \cdot 10^6 \cdot 10 + 100 \cdot 10^6 \cdot 3 = 335 \cdot 10^6$

Now we find the CPI

$$CPI = \frac{CPU clock Cycles}{Instruction Count} = \frac{355 \cdot 10^6}{650 \cdot 10^6} = 5.46$$

$$CPI Reduced = \frac{CPU clock Cycles}{Instruction Count} = \frac{335 \cdot 10^6}{450 \cdot 10^6} = 7.43$$

Again we find the CPU time

CPU Reduced = $900 \cdot 10^6 \cdot 4.22 \cdot 380 \cdot 10^6 = 3.789 seconds$

CPU R2 = $650 \cdot 10^6 \cdot 5.46 \cdot 10^{-9} \cdot 1.1 = 3.9039 seconds$

CPU R10 = $450 \cdot 10^6 \cdot 7.43 \cdot 10^{-9} \cdot 1.1 = 3.67785 seconds$

$\frac{CPU Redcued}{CPU R2} = 0.9728732806 = \underline{\underline{97\%}}$

$\frac{CPU Redcued}{CPU R10} = 1.032668543 = \underline{\underline{103\%}}$

1.4 Problem 3.32

First of all we find out what A, B and C are in Binary. We start with A and we do the following

$0.39843750000 * 2 = 0.796875000$
$0.7968750000 * 2 = 1.593750000$
$0.5937500000 * 2 = 1.187500000$
$0.187500000 * 2 = 0.375000000$
$0.375000000 * 2 = 0.750000000$
$0.7500000000 * 2 = 1.500000000$
$0.500000000 * 2 = 1.000000000$

So it is 0110011 and now we shift it twice

$00110011 x2^0$ →0.110 0011 x $2^{(-1)}$ →1.1001100000x 2^2

B in binary $0.3437500000 * 2 = 0.6875000000$

$0.68750000000 * 2 = 1.375000000$
$0.375000000 * 2 = 0.750000000$
$0.750000000 * 2 = 1.500000000$
$0.500000000 * 2 = 1.000000000$
So it is 01011 and now we shift it twice

$01011x2^0$ →0.10110 x $2^{(-1)}$ →1.0110000000x 2^2
C in binary is 1.771 x 10^3 = 1771 =0110 1110 1011 so when it shifts it becomes $1.1011101011x2^{(10)}$

We Add A and B to each other

1.1001100000
1.0110000000
―――――
10.1111100000
Then we normalize A+B so it becomes $1.0111110000x2^{(-1)}$

now we can do (A+B)+C so we shift (A+B) so it has the same exponant as C

0000000000 10 11101011
1.1011101011

1.1011101100$x2^{(10)}$ Which is in binary 0110 1110 1100 and the final result is 1772

2 Exercise A1.2 - Exam problem (fall 2009)

2.1 a)

To find the CPI for the benchmark program, we use the following formula

$$CPI = \frac{CPUclockCycles}{InstructionCount} = \frac{\sum_{i=1}^{n}(CPI_i XC_i)}{InstructionCount}$$

$$CPI = \frac{8(0.2 \cdot 10^6) + 5(0.3 \cdot 10^6) + 30(0.1 \cdot 10^6) + 4(0.4 \cdot 10^6)}{10^6} = \underline{\underline{7.7}}$$

2.2 b)

Since we know that the clock rate is 200 MHz, we can use the following formula

$$CPUexecutiontimeforaprogram = \frac{CPUclockCyclesforaprogram}{clockRate}$$

$$CPUexecutiontime = \frac{7.7}{200} = \underline{\underline{0.038500seconds}}$$

2.3 c)

To find the speed up, we should find the cpi and cpu new

$$CPI_(new) = \frac{8(0.2 \cdot 10^6) + 5(0.3 \cdot 10^6) + 10(0.1 \cdot 10^6) + 4(0.4 \cdot 10^6)}{10^6} = \mathbf{5.7}$$

$$CPUexecutiontime_(new) = \frac{5.7}{200} \cdot 1.2 = \mathbf{0.034200\ seconds}$$

So the final speed up is

$$Speed - up = \frac{0.38500}{0.034200} = \underline{\underline{1.125730994}}$$

which in percent is 12.5730994 %

Lab Exercises

3 Exercise A1.3 - Integers product

Code "A1.3.s"

3.1 a)

myMult is a non-leaf procedure since it calls others procedures.

3.2 b)

First we didn't use the stack and it worked fine, then we added the stack to make it act more like a real function reusing registers so they act like local variables.

3.3 c)

There will be overflow if the length of multiplicand and the multiplier is larger than 31bits that is used to represent all possible products sence the 32th bit is used for the sign.

4 Exercise A1.4 - Complex numbers product

Code "A1.4.s"

4.1 a)

complexMul is a non-leaf procedure since it calls others procedures.

4.2 b)

Yes, we did use the stack to avoid using many registers as global variables.

4.3 c)

The real part produce overflow if:
The length of the multiplicand plus the length of the multiplier is larger than 31 bits in one of the multiplications. A negative product is subtracted form positive and the result is negative, or A positive product is subtracted from a negative and the result is positive.

The imaginary part produce overflow if:
The length of the multiplicand plus the length of the multiplier is larger than 31 bits in one of the multiplications. The added products of the multiplications have both positive values and the result is negative, or the added products of the multiplication have both negative numbers and the result is positive.

5 Exercise A1.5 - RISC-V instruction for multiplication

5.1 a)

Code "A1.5.s"
Assuming that each instruction takes one clock cycles, we can find the number of clock cycles by counting the instructions
$CI_1 = 1220$
$CI_2 = 81$
Saved clock cycles : $CI_1 - CI_2 = 1220 - 81 = 1140$

6 Exercise A1.6 - Extra questions

6.1 a)

We would have had to use more of the other registers that RISC-V have available and properly use the stack even more. Beside having only one register as the result would mean even more complicated conditions for overflow if that had to be implemented.

6.2 b)

x0	zero	kept
x1	ra	kept
x2	sp	kept
x3	gp	kept
x4	tp	kept
x5-x7	t0-t2	not kept
x8	s0/fp	kept
x9	s1	kept
x10-x11	a0-a1	not kept
x12-x17	a2-a7	not kept
x18-x27	s2-s11	kept
x28-x31	t3-t6	not kept
f0-f7	ft0-ft7	not kept
f8-f9	fs0-fs1	kept
f10-f11	fa0-fa1	not kept
f12-f17	fa2-fa7	not kept
f18-f27	fs2-fs11	kept
f28-f31	ft8-ft11	kept

The hardware can't stop you from overwriting the register then you are coding the procedure, but this standard is meant to be followed when designing a program.

6.3 c)

Store each 32-bit values in memory next to each other. We achieve this by using the registers as HI and LO registers so that the most significant 32 bits are stored in HI registers while the least signficant 32 bits stored in LO registers. When shifting left in multiplication we save the most significant bit in the LO register, and when shifting right we store the least significant bit in the HI register.

Appendix

7 Code A1.1

```
 # 02155, Assignment1, A1.1 Exercise 234
  .data
hello:
#  .asciiz "120"
#     .asciiz "-180"
  .asciiz "+300"
  .text
  .globl main

main:
  la a0, hello          # Load x10 with a string

  jal Convert           # Jump and link Convert

    mv a1, a0
  li a0, 1                 # Load instruction to print in binary form
  ecall
    mv a0, a1

  nop
  j end              # Jump to end
  nop
##############################################################################
Convert:
  addi sp, sp, -36          # adjust stack to make room for 8 items of the size 32 bit (8*4byte)
    sw ra, 0(sp)        # saves return-address
  sw t0, 4(sp)              # save temporary register t0 for use afterwards
  sw t1, 8(sp)              # save temporary register t1 for use afterwards
  sw t2, 12(sp)               # save temporary register t2 for use afterwards
  sw t3, 16(sp)               # save temporary register t3 for use afterwards
  sw t4, 20(sp)               # save temporary register t4 for use afterwards
  sw t5, 24(sp)               # save temporary register t5 for use afterwards
  sw t6, 28(sp)               # save temporary register t6 for use afterwards
    sw a1, 32(sp)       # 10

  li t0, 0            # initialize temporary register for byte
  li t1, 0            # initialize temporary register for temp byte
  li t2, 0          # initialize temporary register for sign
  li t3, 0x2D         # initialize temporary register for compare value(Ascii "-")
  li t4, 0x2B         # initialize temporary register for compare value(Ascii "+")
  li t5, 0x30         # initialize temporary register for compare value(hex 30)
  li t6, 0x40         # initialize temporary register for compare value(hex 40)
    li a1, 10

  jal TestSign          #

  jal TestByte             #

    jal Sign          #

    mv a0, t0
```

```
  lw ra, 0(sp)         # restores return-address
    lw t0, 4(sp)              # restore register t0 for caller
  lw t1, 8(sp)               # restore register t1 for caller
  lw t2, 12(sp)              # restore register t2 for caller
  lw t3, 16(sp)              # restore register t3 for caller
  lw t4, 20(sp)              # restore register t4 for caller
  lw t5, 24(sp)              # restore register t5 for caller
  lw t6, 28(sp)              # restore register t6 for caller
    lw a1, 32(sp)
  addi sp, sp, 36            # adjust stack to delete 8 items

jr ra                #jump to the restored return-address
################################################################################
TestSign:
  lb t1, 0(a0)         # Load first byte
    beq t1, t3, NumNeg       # If the loaded byte is equal to ascii "-"
    beq t1, t4, NumPos       # If the loaded byte is equal to ascii "+"
    jr ra
NumNeg:
  addi t2, t2, 1         # set t2 to 1 because the string is a negativ number

    addi a0, a0, 1         # Prepare byte
    lb t1, 0(a0)         # Load first byte
    jr ra
################################################################################
NumPos:

    addi a0, a0, 1         # Prepare for next byte
    lb t1, 0(a0)         # Load byte
    jr ra
################################################################################
TestByte:
  beq t1, x0, Exit
  blt t1, t5, Fail
    bge t1, t6, Fail

    mul t0, t0, a1
    sub t1, t1, t5
    add t0, t0, t1

    addi a0, a0, 1         # Prepare for next byte
    lb t1, 0(a0)         # Load byte

    j TestByte
################################################################################
Fail:
  addi a0, a0, -1
    j end
################################################################################
Sign:
  bne t2, x0, AddSign
    jr ra
AddSign:
  xori t0, t0, -1
  addi t0, t0, 1
    mv t2, x0
    j Sign
```

```
################################################################################
Exit:
  jr ra
################################################################################
end:
  nop
```

8 Code A1.3

```
       # 02155, Assignment1, Exercise A1.3
    .data
#aa:    .word 0x2           #initialize multiplicand
aa:   .word 0b11111111111111111111111111111110          #initialize multiplicand
bb:   .word 0x3             #initialize multiplier
pp:   .word 0                 #initialize product
    .text
    .globl main

main:
  lw a0, aa           # load multiplicand into a0   A
  lw a1, bb                 # load multiplier into a1     B

  jal myMult            # jump and link myMult

    mv a1, a0
  li a0, 1                  # load instruction to print in binary form
  ecall
    mv a0, a1

  nop
  j end               # jump to end
  nop
################################################################################
myMult:
  addi sp, sp, -28          # adjust stack to make room for 8 items of the size 32
    bit (8*4byte)
    sw ra, 0(sp)        # saves return-address
  sw t0, 4(sp)              # save temporary register t0 for use afterwards
  sw t1, 8(sp)              # save temporary register t1 for use afterwards
  sw t2, 12(sp)               # save temporary register t2 for use afterwards
  sw t3, 16(sp)               # save temporary register t3 for use afterwards
  sw t4, 20(sp)               # save temporary register t4 for use afterwards
  sw t5, 24(sp)               # save temporary register t5 for use afterwards


  li t0, 0            # initialize temporary register for sign_flag
  li t1, 0            # initialize temporary register for product
  li t2, 0          # initialize temporary register for
    test_on_least_significant_bit
  li t3, 0          # initialize temporary register for counter
  li t4, 1          # initialize temporary register for compare value 1
  li t5, 31         # initialize temporary register for compare value 31

```

```
  jal checkSign         #jump and link checkSign which handels signed multiplication

  jal testMultiplier        # jump and link testMultiplier

  jal TestSign

  lw ra, 0(sp)        # restores return-address
    lw t0, 4(sp)              # restore register t0 for caller
  lw t1, 8(sp)              # restore register t1 for caller
  lw t2, 12(sp)             # restore register t2 for caller
  lw t3, 16(sp)             # restore register t3 for caller
  lw t4, 20(sp)             # restore register t4 for caller
  lw t5, 24(sp)             # restore register t5 for caller

  addi sp, sp, 28           # adjust stack to delete 8 items

jr ra               #jump to the restored return-address
##################################################################################
checkSign:
  blt a0, x0,converta0    # convert the multiplicand to positive number
    blt a1, x0,converta1    # convert the multiplier to positive number
  xor t0, t0, t1        # xor the sign_flag_multiplicand and sign_flag_multiplier
    and store it in sign_flag
  li t1, 0            # initialize temporary register for product
    jr ra
##################################################################################
converta0:
  addi a0, a0, -1       # 2 instructions inverting to 2-compliment
  xori a0, a0, -1       #
  li t0, 1          # load the negative sign to t0
    j checkSign
##################################################################################

converta1:
  addi a1, a1, -1       # 2 instructions inverting to 2-compliment
  xori a1, a1, -1       #
  li t1, 1          # load the negative sign to t0
    j checkSign
##################################################################################
testMultiplier:
  andi t2, a1, 1        # load the last bit of the multipler into temp reg t3
  beq t2, t4, multiplier1   # If the least significant bit of the multiplier(t2) is
    equal 1(t4), go to multiplier1
  beq t0, t4, shiftSigned   # if signed (sign_flag(t0) is equal 1(t4)) shift for 31
    iterations
  jr ra

multiplier1:
  add t1, t1, a0        # add multiplicand(a0) to product(t1) and place the result
    in product register(t1)

##################################################################################
shiftSigned:
  slli a0, a0, 1        # shift the multiplicand register left 1 bit
  srli a1, a1, 1        # shift the multiplier register right 1 bit
  addi t3, t3, 1        # increment counter(t3)
  beq t3, t5, Exit      # if its 31th repetition end loop(counter(t3) equal 31(t5))
```

```
  j testMultiplier      # else jump to testMultiplier
Exit:
  jr ra
################################################################################
TestSign:
  beq t0, t4, Sign      # if the product should be negativ, make it(sign_flag(t0)
    equal 1(t4))
    mv a0, t1         # if no sign is needed move product(t1) to a0
    jr ra
################################################################################
Sign:
  xori t1, t1, -1       # 2 instructions getting 2-compliment of the product
  addi a0, t1, 1        # also stores the product in a0
    jr ra
################################################################################
end:

  nop
```

9 Code A1.4

```
# Course 02155 , Assignment 1, A1 template
  .data
aa: .word 2 # Re part of z
bb: .word -3 # Im part of z
cc: .word -4 # Re part of w
dd: .word -5 # Im part of w
img:.asciiz " + i"
  .text
  .globl main

main:
  lw a0 , aa      #a
  lw a1 , bb     #b
  lw a2 , cc     #c
  lw a3 , dd     #d
  jal complexMul # Multiply z and w

    mv a2, a0
    mv a3, a1

    mv a1, a2
  li a0, 1                # load instruction to print in binary form
  ecall

  la a1, img
  li a0, 4
  ecall

    mv a1, a3
    li a0, 1
    ecall

```

```
    mv a0, a2
    mv a1, a3

  nop
  j end # Jump to end of program
  nop
#############################################################################################
complexMul:
  addi sp, sp, -60          # adjust stack to make room for 8 items of the size 32
    bit (8*4byte)
    sw ra, 0(sp)        # saves return-address
    sw a0, 4(sp)
    sw a1, 8(sp)
    sw a2, 12(sp)
    sw a3, 16(sp)
    sw s0, 20(sp)
    sw s1, 24(sp)
    sw s2, 28(sp)
    sw s3, 32(sp)
  sw t0, 36(sp)               # save temporary register t0 for use afterwards
  sw t1, 40(sp)               # save temporary register t1 for use afterwards
  sw t2, 44(sp)               # save temporary register t2 for use afterwards
  sw t3, 48(sp)               # save temporary register t3 for use afterwards
  sw t4, 52(sp)               # save temporary register t4 for use afterwards
  sw t5, 56(sp)               # save temporary register t5 for use afterwards

    lw a0, 4(sp)
    lw a1, 12(sp)
    jal myMult
    sw a0, 20(sp)

    lw a0, 8(sp)
    lw a1, 16(sp)
    jal myMult
    sw a0, 24(sp)

    lw a0, 4(sp)
    lw a1, 16(sp)
    jal myMult
    sw a0, 28(sp)

    lw a0, 8(sp)
    lw a1, 12(sp)
    jal myMult
    sw a0, 32(sp)

    lw a0, 20(sp)
    lw a1, 24(sp)
    sub a0, a0, a1

    lw a1, 28(sp)
    lw a2, 32(sp)
    add a1, a1, a2

  lw ra, 0(sp)        # restores return-address

    lw a2, 12(sp)
    lw a3, 16(sp)
```

```
    lw s0, 20(sp)
    lw s1, 24(sp)
    lw s2, 28(sp)       # restore register s2 for caller
    lw s3, 32(sp)       # restore register s3 for caller
    lw t0, 36(sp)               # restore register t0 for caller
  lw t1, 40(sp)             # restore register t1 for caller
  lw t2, 44(sp)             # restore register t2 for caller
  lw t3, 48(sp)             # restore register t3 for caller
  lw t4, 52(sp)             # restore register t4 for caller
  lw t5, 56(sp)             # restore register t5 for caller
  addi sp, sp, 60           # adjust stack to delete 8 items

jr ra
##################################################################################
myMult:
  addi sp, sp, -4           # adjust stack to make room for 8 items of the size 32 bit (8*4byte)
    sw ra, 0(sp)        # saves return-address
  li t0, 0            # initialize temporary register for sign_flag
  li t1, 0            # initialize temporary register for product
  li t2, 0          # initialize temporary register for test_on_least_significant_bit
  li t3, 0          # initialize temporary register for counter
  li t4, 1          # initialize temporary register for compare value 1
  li t5, 31         # initialize temporary register for compare value 31

  jal checkSign         #jump and link checkSign which handels signed multiplication

  jal testMultiplier        # jump and link testMultiplier

  jal TestSign

  lw ra, 0(sp)        # restores return-address
  addi sp, sp, 4            # adjust stack to delete 8 items

jr ra               #jump to the restored return-address
##################################################################################
checkSign:
  blt a0, x0,converta0    # convert the multiplicand to positive number
    blt a1, x0,converta1    # convert the multiplier to positive number
  xor t0, t0, t1        # xor the sign_flag_multiplicand and sign_flag_multiplier and store it in sign_flag
  li t1, 0            # initialize temporary register for product
    jr ra
##################################################################################
converta0:
  addi a0, a0, -1       # 2 instructions inverting to 2-compliment
  xori a0, a0, -1       #
  li t0, 1          # load the negative sign to t0
    j checkSign
##################################################################################
converta1:
  addi a1, a1, -1       # 2 instructions inverting to 2-compliment
  xori a1, a1, -1       #
  li t1, 1          # load the negative sign to t0
    j checkSign
##################################################################################
testMultiplier:
```

```
  andi t2, a1, 1        # load the last bit of the multipler into temp reg t3
  beq t2, t4, multiplier1   # If the least significant bit of the multiplier(t2) is
    equal 1(t4), go to multiplier1

  j shiftSigned       # shift for 31 iterations
  jr ra
##################################################################################
multiplier1:
  add t1, t1, a0        # add multiplicand(a0) to product(t1) and place the result
    in product register(t1)


##################################################################################
shiftSigned:
  slli a0, a0, 1        # shift the multiplicand register left 1 bit
  srli a1, a1, 1        # shift the multiplier register right 1 bit
  addi t3, t3, 1        # increment counter(t3)
  beq t3, t5, Exit      # if its 31th repetition end loop(counter(t3) equal 31(t5))
  j testMultiplier      # else jump to testMultiplier
##################################################################################
Exit:
  jr ra
##################################################################################
TestSign:
  beq t0, t4, Sign      # if the product should be negativ, make it(sign_flag(t0)
    equal 1(t4))
    mv a0, t1         # if no sign is needed move product(t1) to a0
    jr ra
##################################################################################
Sign:
  xori t1, t1, -1       # 2 instructions getting 2-compliment of the product
  addi a0, t1, 1        # also stores the product in a0
    jr ra
##################################################################################
end:
  nop
```

10 Code A1.5

```
# Course 02155 , Assignment 1, A1 template
  .data
aa: .word 2 # Re part of z
bb: .word -3 # Im part of z
cc: .word -4 # Re part of w
dd: .word -5 # Im part of w
img:.asciiz " + i"
  .text
  .globl main

main:
  lw a0 , aa      #a
  lw a1 , bb     #b
  lw a2 , cc     #c
```

```
  lw a3 , dd     #d
  jal complexMul # Multiply z and w

    mv a2, a0
    mv a3, a1

    mv a1, a2
  li a0, 1                 # load instruction to print in binary form
  ecall

  la a1, img
  li a0, 4
  ecall

    mv a1, a3
     li a0, 1
     ecall

    mv a0, a2
    mv a1, a3

  nop
  j end # Jump to end of program
  nop
##########################################################################################
complexMul:
  addi sp, sp, -60          # adjust stack to make room for 8 items of the size 32
    bit (8*4byte)
    sw ra, 0(sp)          # saves return-address
    sw a0, 4(sp)
    sw a1, 8(sp)
    sw a2, 12(sp)
    sw a3, 16(sp)
    sw s0, 20(sp)
    sw s1, 24(sp)
    sw s2, 28(sp)
    sw s3, 32(sp)
  sw t0, 36(sp)                # save temporary register t0 for use afterwards
  sw t1, 40(sp)                # save temporary register t1 for use afterwards
  sw t2, 44(sp)                # save temporary register t2 for use afterwards
  sw t3, 48(sp)                # save temporary register t3 for use afterwards
  sw t4, 52(sp)                # save temporary register t4 for use afterwards
  sw t5, 56(sp)                # save temporary register t5 for use afterwards

    lw a0, 4(sp)
    lw a1, 12(sp)
    mul a0, a0, a1
    sw a0, 20(sp)

    lw a0, 8(sp)
    lw a1, 16(sp)
    mul a0, a0, a1
    sw a0, 24(sp)

    lw a0, 4(sp)
    lw a1, 16(sp)
    mul a0, a0, a1
    sw a0, 28(sp)
```

```
    lw a0, 8(sp)
    lw a1, 12(sp)
    mul a0, a0, a1
    sw a0, 32(sp)

    lw a0, 20(sp)
    lw a1, 24(sp)
    sub a0, a0, a1

    lw a1, 28(sp)
    lw a2, 32(sp)
    add a1, a1, a2

  lw ra, 0(sp)        # restores return-address

    lw a2, 12(sp)
    lw a3, 16(sp)
    lw s0, 20(sp)
    lw s1, 24(sp)
    lw s2, 28(sp)       # restore register s2 for caller
    lw s3, 32(sp)       # restore register s3 for caller
    lw t0, 36(sp)               # restore register t0 for caller
  lw t1, 40(sp)             # restore register t1 for caller
  lw t2, 44(sp)             # restore register t2 for caller
  lw t3, 48(sp)             # restore register t3 for caller
  lw t4, 52(sp)             # restore register t4 for caller
  lw t5, 56(sp)             # restore register t5 for caller
  addi sp, sp, 60           # adjust stack to delete 8 items

jr ra
#####################################################################################
myMult:
  addi sp, sp, -4           # adjust stack to make room for 8 items of the size 32
    bit (8*4byte)
    sw ra, 0(sp)        # saves return-address
  li t0, 0            # initialize temporary register for sign_flag
  li t1, 0            # initialize temporary register for product
  li t2, 0          # initialize temporary register for
    test_on_least_significant_bit
  li t3, 0          # initialize temporary register for counter
  li t4, 1          # initialize temporary register for compare value 1
  li t5, 31         # initialize temporary register for compare value 31

  jal checkSign         #jump and link checkSign which handels signed multiplication

  jal testMultiplier        # jump and link testMultiplier

  jal TestSign

  lw ra, 0(sp)        # restores return-address
  addi sp, sp, 4            # adjust stack to delete 8 items

jr ra                #jump to the restored return-address
#####################################################################################
checkSign:
  blt a0, x0,converta0    # convert the multiplicand to positive number
    blt a1, x0,converta1    # convert the multiplier to positive number
```

```
  xor t0, t0, t1        # xor the sign_flag_multiplicand and sign_flag_multiplier
    and store it in sign_flag
  li t1, 0              # initialize temporary register for product
    jr ra
##########################################################################################
converta0:
  addi a0, a0, -1       # 2 instructions inverting to 2-compliment
  xori a0, a0, -1       #
  li t0, 1            # load the negative sign to t0
    j checkSign
##########################################################################################
converta1:
  addi a1, a1, -1       # 2 instructions inverting to 2-compliment
  xori a1, a1, -1       #
  li t1, 1            # load the negative sign to t0
    j checkSign
##########################################################################################
testMultiplier:
  andi t2, a1, 1        # load the last bit of the multipler into temp reg t3
  beq t2, t4, multiplier1   # If the least significant bit of the multiplier(t2) is
    equal 1(t4), go to multiplier1

  j shiftSigned         # shift for 31 iterations
  jr ra
##########################################################################################
multiplier1:
  add t1, t1, a0        # add multiplicand(a0) to product(t1) and place the result
    in product register(t1)


##########################################################################################
shiftSigned:
  slli a0, a0, 1        # shift the multiplicand register left 1 bit
  srli a1, a1, 1        # shift the multiplier register right 1 bit
  addi t3, t3, 1        # increment counter(t3)
  beq t3, t5, Exit      # if its 31th repetition end loop(counter(t3) equal 31(t5))
  j testMultiplier      # else jump to testMultiplier
Exit:
  jr ra
##########################################################################################

TestSign:
  beq t0, t4, Sign      # if the product should be negativ, make it(sign_flag(t0)
    equal 1(t4))
    mv a0, t1           # if no sign is needed move product(t1) to a0
    jr ra
##########################################################################################
Sign:
  xori t1, t1, -1       # 2 instructions getting 2-compliment of the product
  addi a0, t1, 1        # also stores the product in a0
    jr ra
##########################################################################################
end:
  nop
```