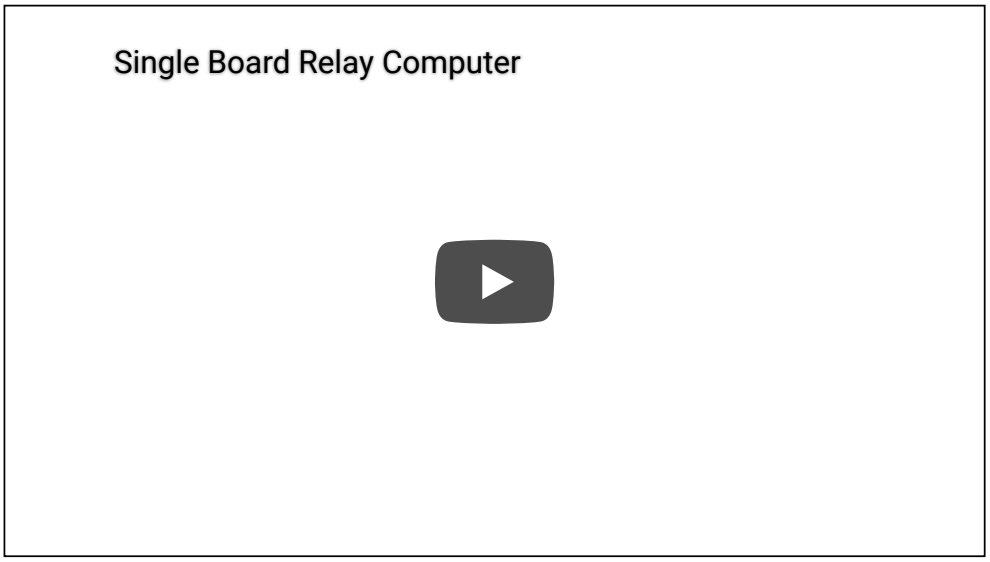
Example Code

Example programs

Overview

Here is a longer overview video featuring the third revision of the relay computer.

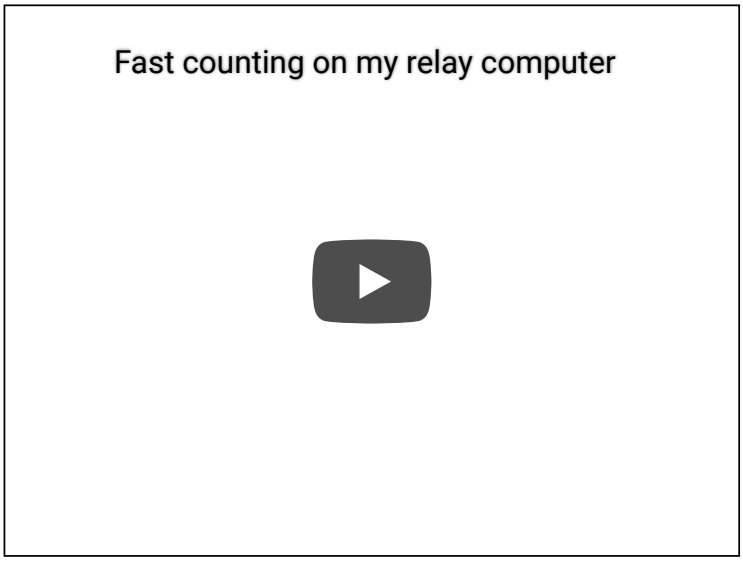


Single instruction counting

This just shows a single instruction loop using the increment and jump if not equal to zero (**incjne**) instruction.

```
                org     0x00
counter         skip    1

                org     0x10
start           incjne  counter, start
```



Euclid's algorithm

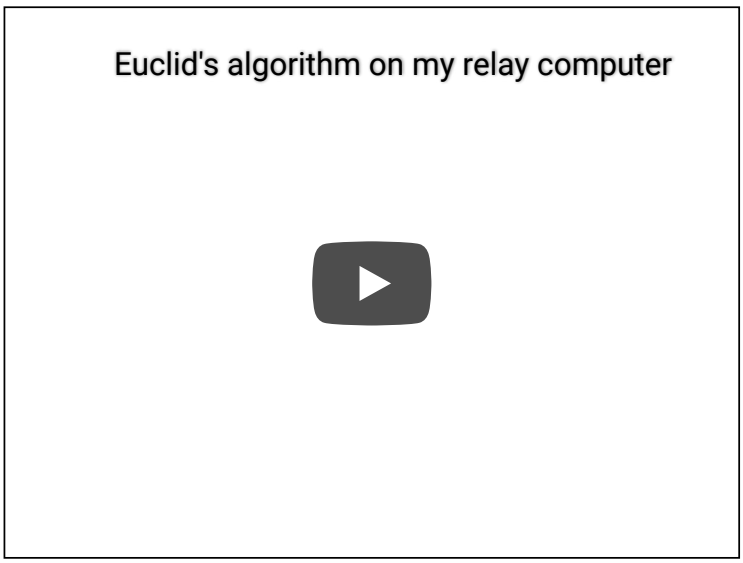
This program finds the greatest common divisor of two numbers using Euclid's algorithm from 300 BC.

```
; Euclid's algorithm using repeated subtraction

                org     0x00
a               skip    1               ; First number
b               skip    1               ; Second number
tmp             skip    1               ; Tmp variable

                org     0x10
euclid          st      #144, a         ; Initialize A
                st      #233, b         ; Initialize B
euclop          jeq     b, eucdon       ; Done ?
                st      a, tmp
                rsbto   b, tmp          ; A - B -> TMP
                jls     tmp, over       ; A <= B ?
                rsbto   b, a            ; A - B -> A
                jmp     euclop
over            rsbto   a, b            ; B - A -> B
                jmp     euclop
eucdon          halt
```

Fun fact: the longest run-time for Euclid's algorithm occurs when you try to find the GCD of two successive numbers from the Fibonacci sequence. When you do this, Euclid's algorithm runs through the Fibonacci sequence in reverse.



Bit-wise OR

This program computed bit-wise OR, which is an instruction missing from the relay computer

```
; Compute bitwise OR: Y = Y | X.
                bicto   x, y    ; Clear bits in y which are set in x
                addto   x, y    ; Add the bits which are set in x into y
```

Exclusive OR

This program computes Exclusive-OR, which is an instruction missing from the relay computer

```
; Compute Exclusive-OR: Y = Y ^ X.
; This can be computed as follows : Y + X - 2*(Y & X)
                st      y, tmp
                andto   x, tmp
                lsl     tmp
```

```
                addto   x, y
                rsbto   tmp, y
```

Multiply

This program multiplies two 8-bit numbers and produces a 16-bit result

```
        org     0x00
argx    skip    1
argy    skip    1
res_lo  skip    1       ; Result low
res_hi  skip    1       ; Result high
count   skip    1

        org     0x10
mul     st      #0, res_lo
        st      #0, res_hi
        st      #-8, count
loop    lsl     res_lo
        rol     res_hi
        lsl     argy
        jcc     skip
        addto   argx, res_lo
        adcto   #0, res_hi
skip    incjne  count, loop
mulrtn  jmp     0

; Try it..
        org     0x20
        st      #3, argx
        st      #5, argy
        jsr     mulrtn, mul
        halt
```

Divide

This program divides an 8-bit divisor into an 8-bit dividend and produces and 8-bit quotient and 8-bit remainder.

```
        org     0x00
quotient        skip    1
remainder       skip    1
dividend        skip    1
divisor skip    1
count   skip    1

        org     0x10
div     clr     remainder
        st      #-8, count
divlop  lsl     dividend                ; Shift dividend into remainder one
        rol     remainder               ; bit at a time...
        rsbto   divisor, remainder      ; Can we subtract divisor now?
        jcc     toomuch                 ; Branch if not..
        lslo    quotient                ; Shift a 1 into quotient
        incjne  count, divlop
        jmp     divrtn
toomuch addto   divisor, remainder      ; Restore..
        lsl     quotient                ; Shift a 0 into quotient
        incjne  count, divlop
divrtn  jmp     0

; Try it
        org     0x20
        st      #42, dividend
        st      #5, divisor
        jsr     divrtn, div
        halt
```

Integer square root

```
; Integer square root

        org     0x00
num     skip    1       ; Find square root of this
result  skip    1       ; Result ends up here

; Subroutine
        org     0x10
sqrt    st      #0xFF, result
sqrt1   addto   #2, result
        rsbto   result, num
        jcs     sqrt1
        lsr     result
s_done  jmp     0

; Try it
        org     0x20
        st      #144, num
        jsr     s_done, sqrt
        halt
```

Subroutines

The relay computer supports subroutines with the jsr instruction. This instruction saves the next instruction address (the return address) in a specified memory location before jumping to the target address. The idea is to insert the return address into a jump instruction which is executed at the end of the subroutine. When the jump is executed, it will transfer control back to the instruction following the jsr instruction.

```
; Calling routine calling 'sub'
                jsr     subrtn, sub
; Next instruction to execute after call is complete

; Called subroutine
sub:            . . .           ; Do some work
subrtn:         jmp     0       ; Return back to caller.  Caller has to insert return address
                                ; into this jump instruction.
```

The above method is fast and directly supported, but does not allow recursion. If recursion is needed, then a stack needs to be implemented. Here is one way to do this:

```
; Calling routine calling 'dest'
                jsr     pushdata, dest
; Next instruction to execute after call is complete
```

```
; Called subroutine
dest:           jsr     pushrtn, push   ; Save return address on stack
                ; Do some work
                jmp     popj            ; Return from subroutine


; Helper subroutine which saves return address on stack and then
; jumps to the requested subroutine.
push:           st      pushdata, 0xff  ; Save return address in stack
; stack starts at 0xFF
                dec     push            ; Decrement stack pointer
pushrtn:        jmp     0               ; Jump to subroutine
pushdata:       nop                     ; Place to save return address


; Helper code which pops return address off stack and jumps to it
popj:           inc     push            ; Increment stack pointer
                st      push, readit    ; Insert stack pointer into following add instruction
                clr     return          ; Clear return, we're going to add to it
readit:         add     return, 0       ; Insert return address from stack into following jmp
return:         jmp     0               ; Jump to it
```

Pointers

Pointer registers can be implemented with self modified code:

```

; Memory copy subroutine

from_ptr        skip    1       ; Address to copy from
to_ptr          skip    1       ; Address to copy to
count           skip    1       ; No. bytes to copy
tmp             skip    1

; Copy..
memcpy          neg     count   ; Negate count so we can use incjne

; Read indirection from a pointer:
loop            st      from_ptr, get_it        ; Insert pointer into code
                clr     tmp                     ; Pre-clear desintation
get_it          add     tmp, 0                  ; Add target to tmp

; Write indirection to a pointer:
                st      to_ptr, put_it          ; Insert pointer into code
put_it          st      tmp, 0                  ; Store tmp to target

; Increment pointers
                inc     from_ptr
                inc     to_ptr
                incjne  count, loop             ; Loop if not done

memcpy_rtn      jmp     0                       ; Return from subroutine
```

Hello, world!

This program writes "Hello, world!" to the serial console.

```

        org     0x00

tmp     skip    1

msg     data    0x48
        data    0x65
        data    0x6C
        data    0x6C
        data    0x6F
        data    0x2C
        data    0x20
        data    0x57
        data    0x6F
        data    0x72
```

```
        data    0x6C
        data    0x64
        data    0x21
        data    0x0D
        data    0x0A
        data    0x00

        org     0x20

start   st      #msg, ptr       ; Point to message
loop    clr     tmp             ; Pre-clear
ptr     add     tmp, 0          ; Read from pointer
        jeq     tmp, done       ; Jump if end of message
        outc    tmp             ; Write character to serial
        inc     ptr             ; Increment pointer
        jmp     loop            ; Loop...
done    halt
```

Bubble sort

This example shows the use of pointers.

```
; Bubble sort

        org     0x00
count   skip    1
flag    skip    1
tmp     skip    1
tmp1    skip    1
tmp2    skip    1

; Some numbers to sort..
dstart  data    5
        data    1
        data    10
        data    12
        data    3
        data    20
        data    4
        data    8
dend

        org     0x20

sort    st      #-(dend-dstart-1), count        ; Number of items...
        clr     flag
        st      #dstart, ptr                    ; Set pointers
        st      #dstart+1, ptr1
loop
; Read items
        clr     tmp
ptr     add     tmp, 0
        clr     tmp1
ptr1    add     tmp1, 0
; Compare them
        st      tmp, tmp2
        rsbto   tmp1, tmp2
        jls     tmp2, noswap    ; Branch if already in order
; Swap items
        st      ptr, ptr2       ; Copy pointers
        st      ptr1, ptr3
ptr2    st      tmp1, 0
ptr3    st      tmp, 0
; Set flag to indicate we did something
        inc     flag
noswap  inc     ptr             ; Advance pointers
        inc     ptr1
        incjne  count, loop     ; loop
        jne     flag, sort      ; Repeat until sorted
        halt
```

LFSR Random number generator

```
                org     0
acc             data    1
scratch         skip    1
lut             data    9
count           data    0x80

                org     5
loop            jsr     done, rng
                incjne  count, loop
                halt

                org     0x10
rng             st      acc, scratch
                rol     scratch
                rol     scratch
                rol     scratch
                andto   #0x3, scratch
                jeq     scratch, appendone
                dec     scratch
                jeq     scratch, appendzero
                dec     scratch
                jeq     scratch, appendzero

appendone       lslo    acc
                jmp     done

appendzero      lsl     acc
done            jmp     0
```

Simon-like memory game

```
; Memory game

                org     0x00
pat_len         skip    1                       ; Pattern length
rng             skip    1                       ; Random number generator
count           skip    1                       ; Counter
delay_count     skip    1
tmp             skip    1
tmp1            skip    1

; Start

done            org     0x0f    ; Stop just before start so player can
                                ; hit run button to play again.
                halt
start           st      #0xfd, pat_len          ; Initial length
main_loop       jsr     show_rtn, show_pat      ; Show pattern
                jsr     read_rtn, read_pat      ; Read pattern from user
                jcc     done                    ; We fail
                dec     pat_len                 ; Increase length
                jmp     main_loop

; Show pattern to player

show_pat        st      #1, rng
                st      pat_len, count
show_loop       st      rng, tmp
                rol     tmp
                rol     tmp
                rol     tmp
                andto   #3, tmp
                st      #1, tmp1

                jeq     tmp, tdone
tloop           lsl     tmp1
                dec     tmp
                jne     tmp, tloop

tdone           out     tmp1
                jsr     delay_rtn, delay
                out     #0
                jsr     rng_rtn, rng_step
                incjne  count, show_loop
show_rtn        jmp     0
```

```
; Delay

delay           st      #0xFA, delay_count
delay_loop      incjne  delay_count, delay_loop
delay_rtn       jmp     0

; Random number generator: rng = rng*49 + 47 = rng * 32 + rng * 16 + rng + 47

rng_step        st      rng, tmp
                lsl     tmp
                lsl     tmp
                lsl     tmp
                lsl     tmp
                addto   tmp, rng
                lsl     tmp
                addto   tmp, rng
                addto   #47, rng
rng_rtn         jmp     0

; Read pattern from player, verifying along the way

read_pat        st      #1, rng
                st      pat_len, count
read_loop
                inwait  tmp
                jeq     tmp, read_loop
                out     tmp
                st      #0xff, tmp1
cvt_loop        inc     tmp1
                lsr     tmp
                jcc     cvt_loop
                st      rng, tmp
                rol     tmp
                rol     tmp
                rol     tmp
                andto   #3, tmp
                rsbto   tmp1, tmp
                jne     tmp, fail
                out     #0
                jsr     rng_rtn, rng_step
unpress         in      tmp
                jne     tmp, unpress
                incjne  count, read_loop
                stc
read_rtn        jmp     0

; Flash all LEDs if player makes a mistake
fail            out     #15
                jsr     delay_rtn, delay
                jsr     delay_rtn, delay
                out     #0
                clc
                jmp     read_rtn
```