Reasoning about `copyData`

Yet Another Account of a Proof of Correctness of an x86 Machine-Code Program

Shilpi Goel

ACL2 Seminar

19th October, 2015

Quick Background

- This talk is about machine-code program verification using the ACL2 `x86isa` books: `acl2/books/projects/x86isa`
  See documentation at:
  `http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____X86ISA`

- From `x86isa/README`:

  "These books contain the specification of x86 instruction set architecture (ISA); we characterize x86 machine instructions and model the instruction fetch, decode, and execute process using the ACL2 theorem-proving system. We use our x86 ISA specification to simulate and formally verify x86 machine-code programs."

- The book corresponding to this talk is:
  `x86isa/proofs/dataCopy/dataCopy.lisp`

This Talk

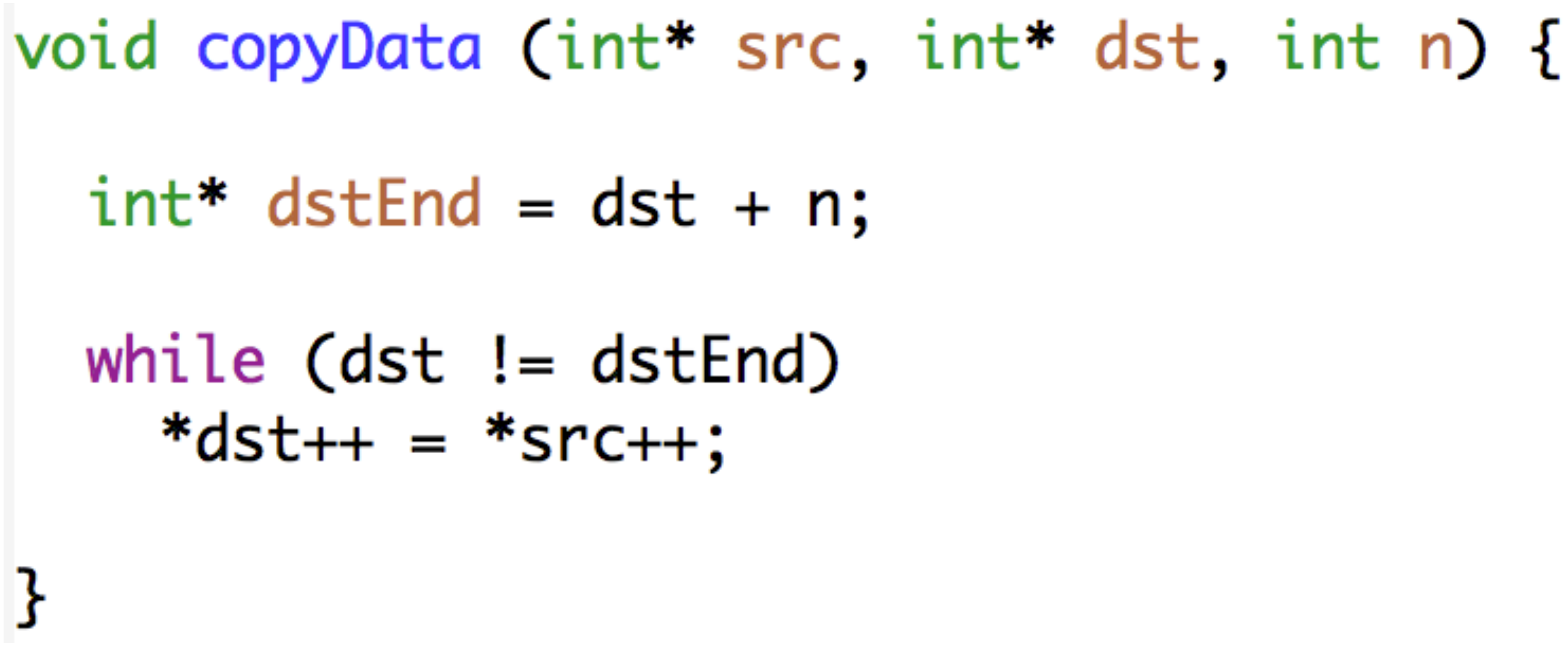
I'm going to walk through a naïve approach of reasoning about a simple x86 machine-code program — `copyData`.

Why?

1. This may help someone looking for **challenge programs** in ACL2 — consider using the `x86isa` books to verify a simple program!
2. Reasoning about memory regions (e.g., arrays) can be challenging and I want to share a **small success story**.
3. Though this naïve approach works well for a first attempt to verify a given program, I can definitely use **feedback**.

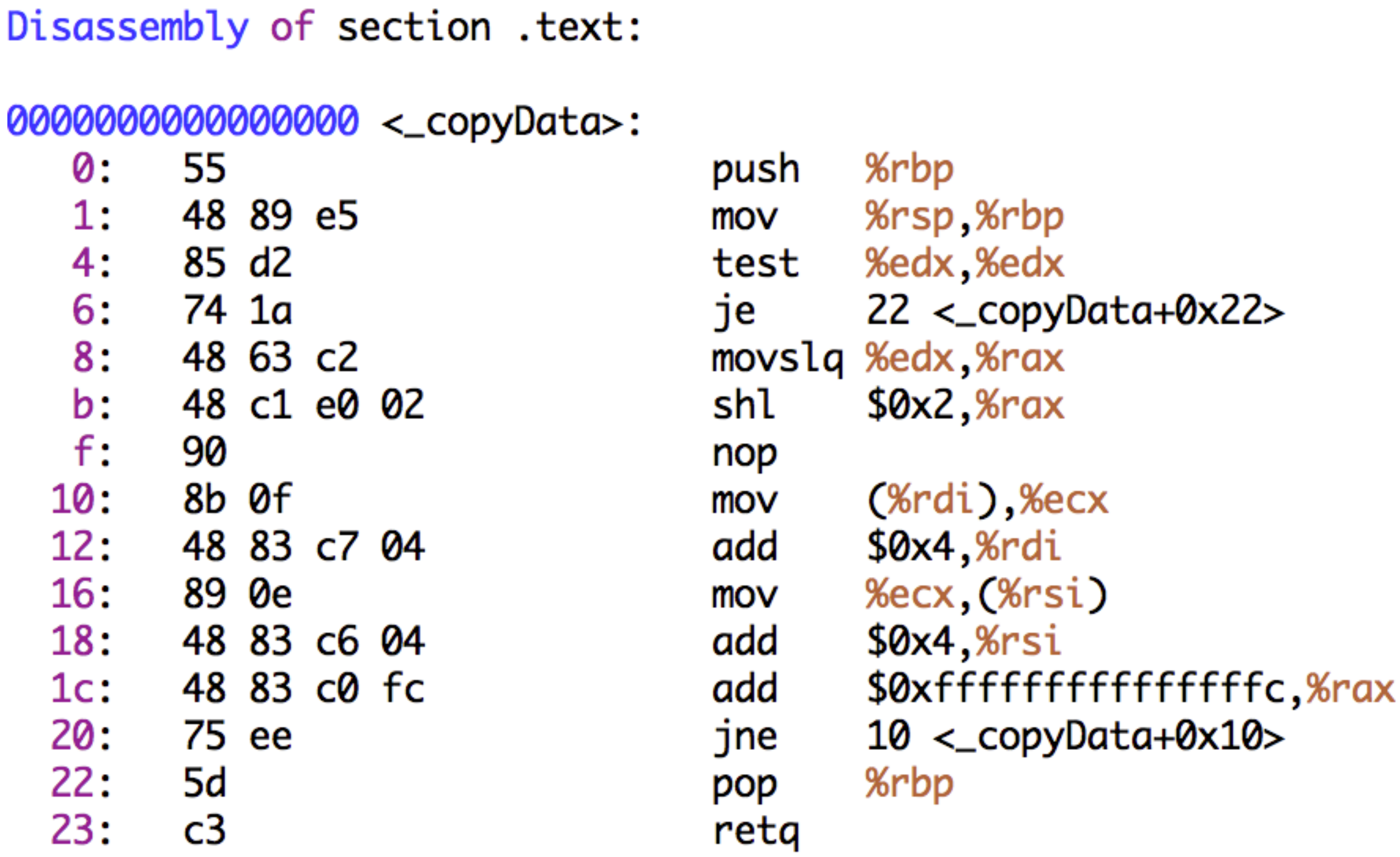
Note: This talk involves reading a lot of ACL2.

copyData Sub-Routine



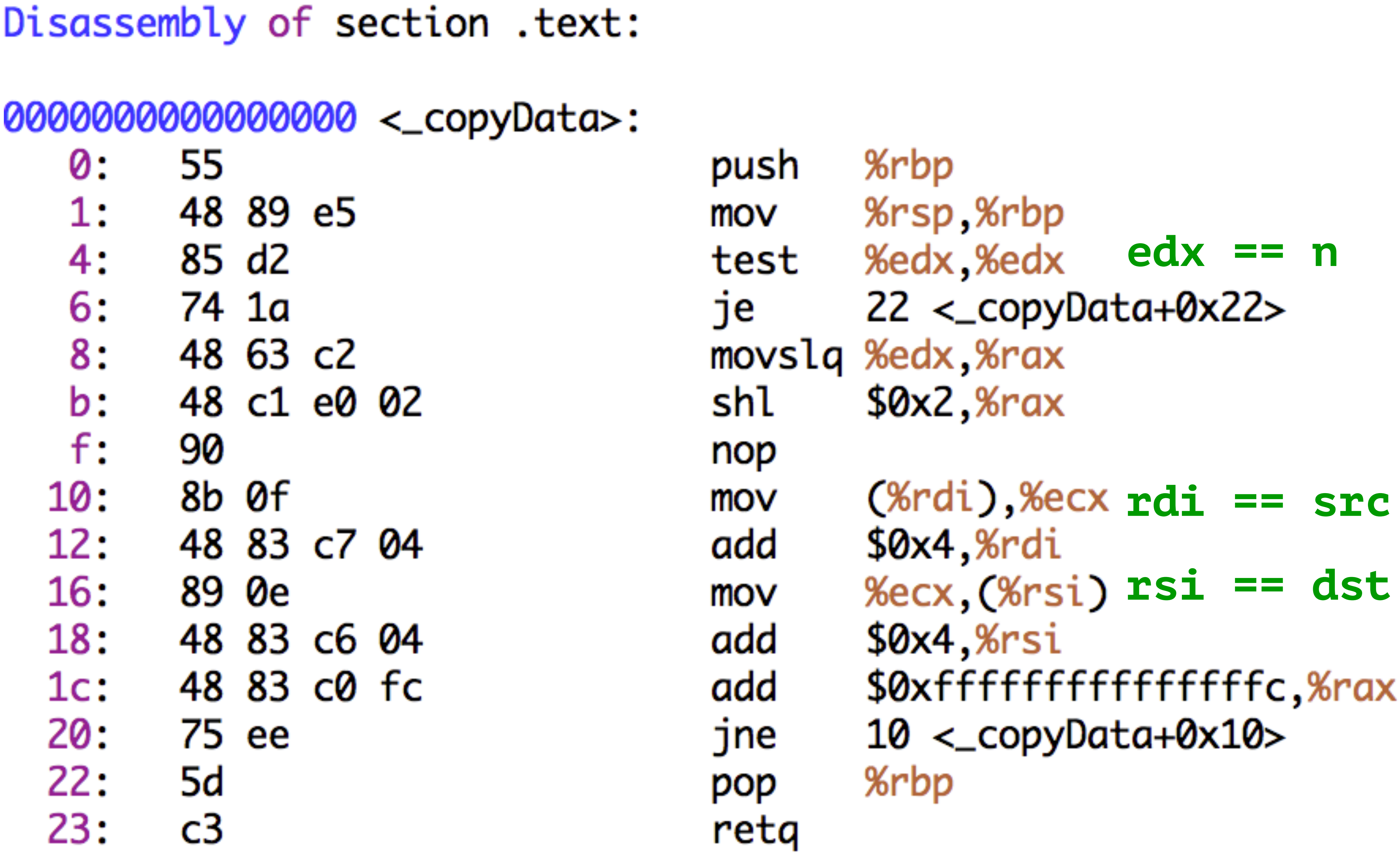
```
void copyData (int* src, int* dst, int n) {

  int* dstEnd = dst + n;

  while (dst != dstEnd)
    *dst++ = *src++;

}
```

copyData Sub-Routine



```
Disassembly of section .text:

0000000000000000 <_copyData>:
   0:	55                   	push   %rbp
   1:	48 89 e5             	mov    %rsp,%rbp
   4:	85 d2                	test   %edx,%edx
   6:	74 1a                	je     22 <_copyData+0x22>
   8:	48 63 c2             	movslq %edx,%rax
   b:	48 c1 e0 02          	shl    $0x2,%rax
   f:	90                   	nop
  10:	8b 0f                	mov    (%rdi),%ecx
  12:	48 83 c7 04          	add    $0x4,%rdi
  16:	89 0e                	mov    %ecx,(%rsi)
  18:	48 83 c6 04          	add    $0x4,%rsi
  1c:	48 83 c0 fc          	add    $0xfffffffffffffffc,%rax
  20:	75 ee                	jne    10 <_copyData+0x10>
  22:	5d                   	pop    %rbp
  23:	c3                   	retq
```

copyData Sub-Routine



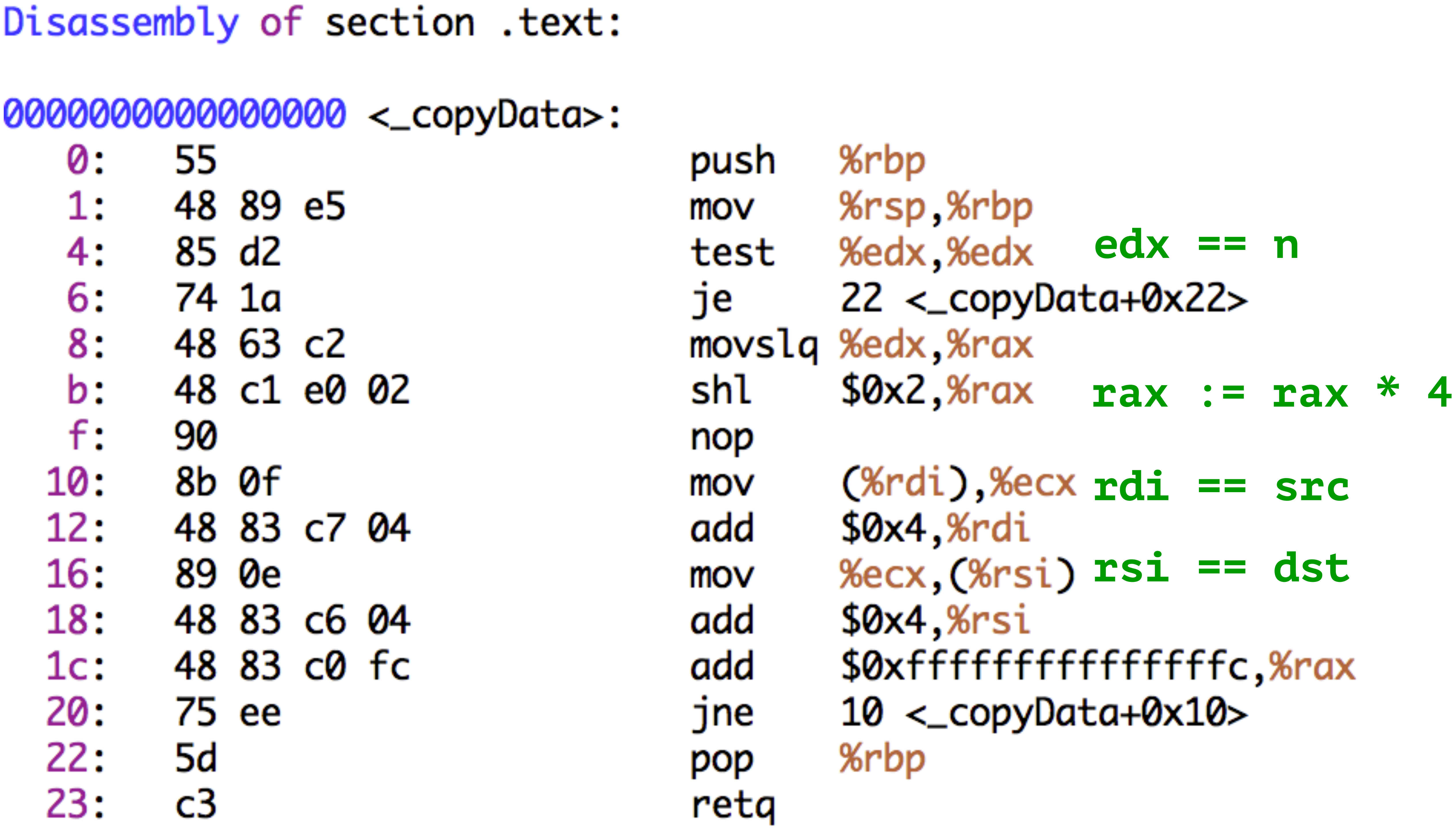
```
Disassembly of section .text:

0000000000000000 <_copyData>:
   0:	55                   	push   %rbp
   1:	48 89 e5             	mov    %rsp,%rbp
   4:	85 d2                	test   %edx,%edx
   6:	74 1a                	je     22 <_copyData+0x22>
   8:	48 63 c2             	movslq %edx,%rax
   b:	48 c1 e0 02          	shl    $0x2,%rax
   f:	90                   	nop
  10:	8b 0f                	mov    (%rdi),%ecx
  12:	48 83 c7 04          	add    $0x4,%rdi
  16:	89 0e                	mov    %ecx,(%rsi)
  18:	48 83 c6 04          	add    $0x4,%rsi
  1c:	48 83 c0 fc          	add    $0xfffffffffffffffc,%rax
  20:	75 ee                	jne    10 <_copyData+0x10>
  22:	5d                   	pop    %rbp
  23:	c3                   	retq
```

edx == n

rdi == src

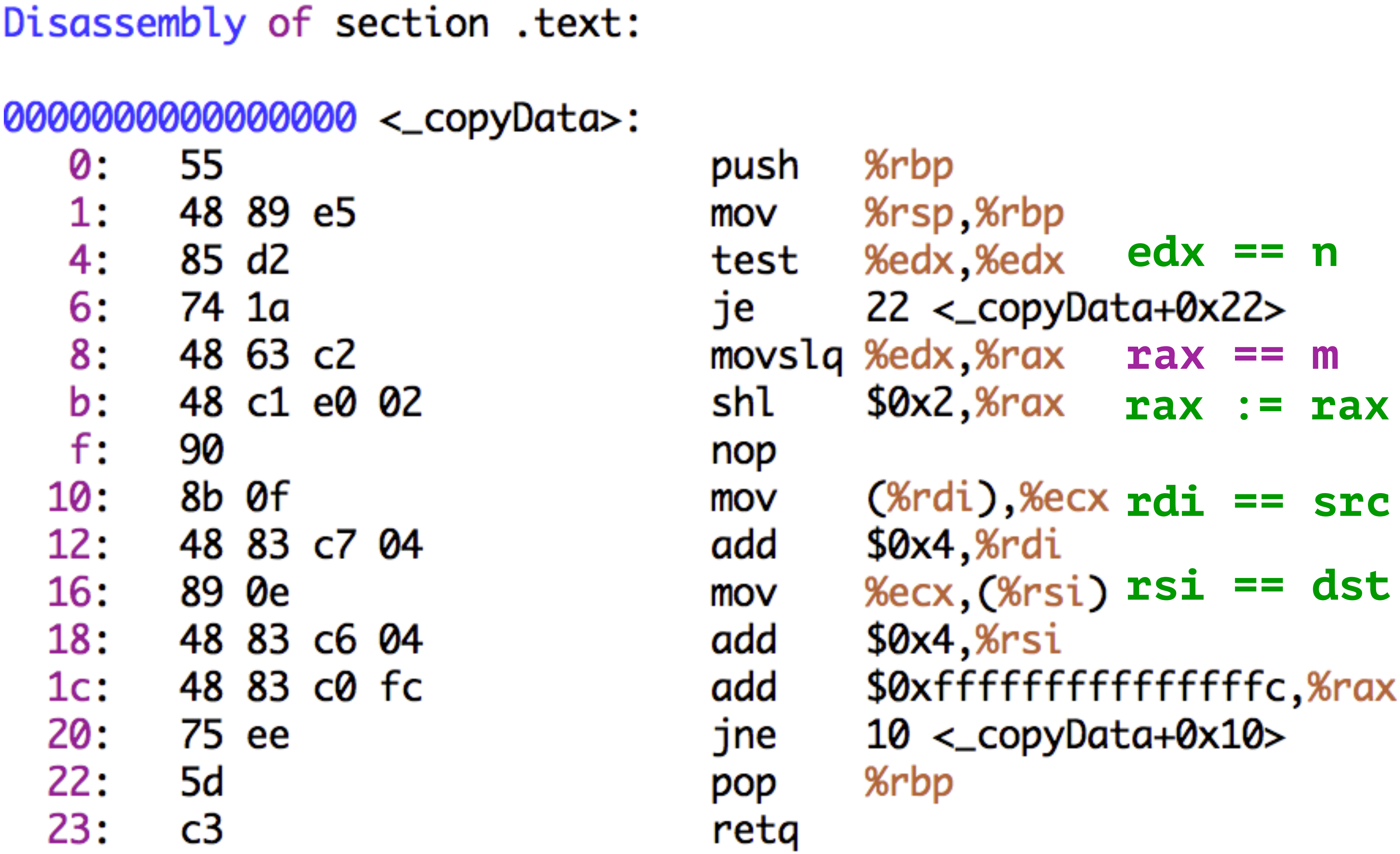
rsi == dst

copyData Sub-Routine



```
Disassembly of section .text:

0000000000000000 <_copyData>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   85 d2                   test   %edx,%edx      edx == n
   6:   74 1a                   je     22 <_copyData+0x22>
   8:   48 63 c2                movslq %edx,%rax
   b:   48 c1 e0 02             shl    $0x2,%rax      rax := rax * 4
   f:   90                      nop
  10:   8b 0f                   mov    (%rdi),%ecx    rdi == src
  12:   48 83 c7 04             add    $0x4,%rdi
  16:   89 0e                   mov    %ecx,(%rsi)    rsi == dst
  18:   48 83 c6 04             add    $0x4,%rsi
  1c:   48 83 c0 fc             add    $0xfffffffffffffffc,%rax
  20:   75 ee                   jne    10 <_copyData+0x10>
  22:   5d                      pop    %rbp
  23:   c3                      retq
```

copyData Sub-Routine



```
Disassembly of section .text:

0000000000000000 <_copyData>:
   0:	55                   	push   %rbp
   1:	48 89 e5             	mov    %rsp,%rbp
   4:	85 d2                	test   %edx,%edx
   6:	74 1a                	je     22 <_copyData+0x22>
   8:	48 63 c2             	movslq %edx,%rax
   b:	48 c1 e0 02          	shl    $0x2,%rax
   f:	90                   	nop
  10:	8b 0f                	mov    (%rdi),%ecx
  12:	48 83 c7 04          	add    $0x4,%rdi
  16:	89 0e                	mov    %ecx,(%rsi)
  18:	48 83 c6 04          	add    $0x4,%rsi
  1c:	48 83 c0 fc          	add    $0xfffffffffffffffc,%rax
  20:	75 ee                	jne    10 <_copyData+0x10>
  22:	5d                   	pop    %rbp
  23:	c3                   	retq
```

edx == n

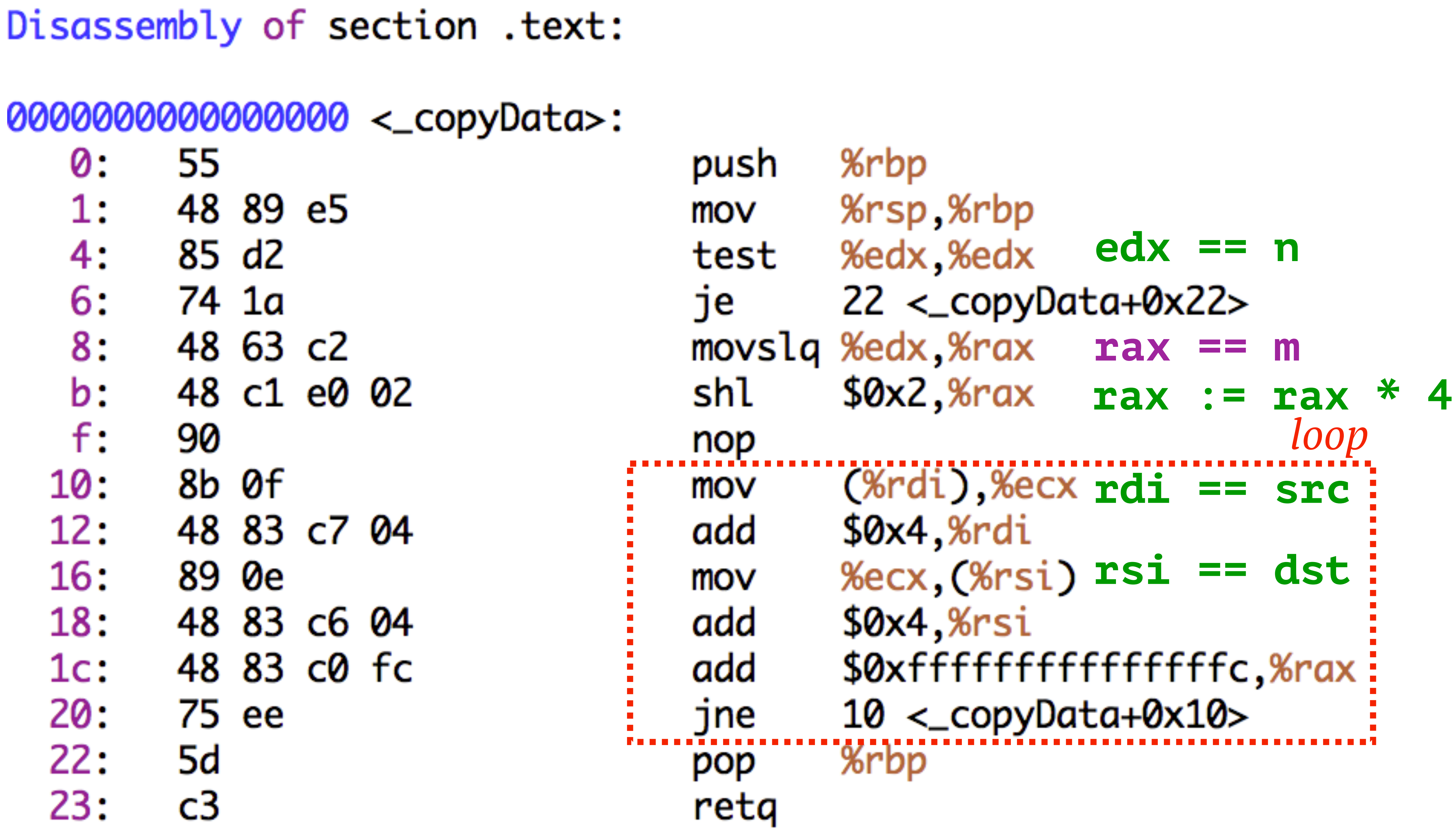
rax == m

rax := rax * 4

rdi == src

rsi == dst

copyData Sub-Routine



```
Disassembly of section .text:

0000000000000000 <_copyData>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   85 d2                   test   %edx,%edx
   6:   74 1a                   je     22 <_copyData+0x22>
   8:   48 63 c2                movslq %edx,%rax
   b:   48 c1 e0 02             shl    $0x2,%rax
   f:   90                      nop
  10:   8b 0f                   mov    (%rdi),%ecx
  12:   48 83 c7 04             add    $0x4,%rdi
  16:   89 0e                   mov    %ecx,(%rsi)
  18:   48 83 c6 04             add    $0x4,%rsi
  1c:   48 83 c0 fc             add    $0xfffffffffffffffc,%rax
  20:   75 ee                   jne    10 <_copyData+0x10>
  22:   5d                      pop    %rbp
  23:   c3                      retq
```

edx == n

rax == m

rax := rax * 4

loop

rdi == src

rsi == dst

Step 0: What Properties Do You Care About?

```
let
data = src[src-ptr to (src-ptr + m - 1)]
in x86
∧ <preconditions> ⇒

dst[dst-ptr to (dst-ptr + m - 1)]
in (x86-run (program-clk m) x86)
==
data

 ∧

src[src-ptr to (src-ptr + m - 1)]
in (x86-run (program-clk m) x86)
==
data
```

copy operation is successful

source is unmodified

Step 0: What Properties Do You Care About?

```
let
data = src[src-ptr to (src-ptr + m - 1)]
in x86
∧ <preconditions> ⇒

dst[dst-ptr to (dst-ptr + m - 1)]
in (x86-run (program-clk m) x86)
==
data

 ∧

src[src-ptr to (src-ptr + m - 1)]
in (x86-run (program-clk m) x86)
==
data
```

copy operation is successful

source is unmodified

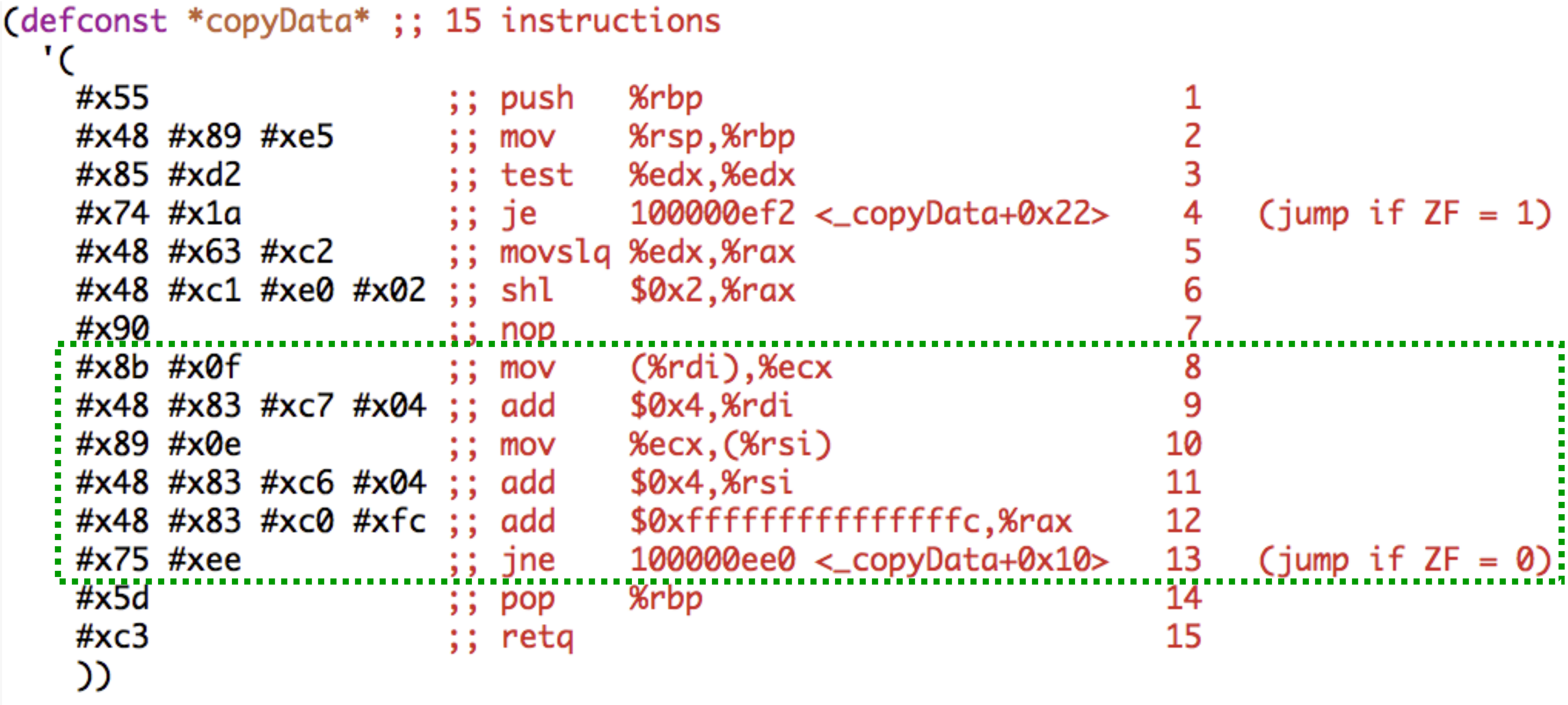
Step 1: Setup

Include `x86isa` + other helper books.

```
(in-package "X86ISA")

(include-book "programmer-level-memory-utils" :dir :proof-utils :ttags :all)
(include-book "centaur/bitops/ihs-extensions" :dir :system)

(local (include-book "centaur/bitops/signed-byte-p" :dir :system))
(local (include-book "arithmetic/top-with-meta" :dir :system))
```

Step 1: Setup

Introduce the program.



```
(defconst *copyData* ;; 15 instructions
  '(
    #x55                ;; push   %rbp                                   1
    #x48 #x89 #xe5      ;; mov    %rsp,%rbp                              2
    #x85 #xd2           ;; test   %edx,%edx                              3
    #x74 #x1a           ;; je     100000ef2 <_copyData+0x22>             4   (jump if ZF = 1)
    #x48 #x63 #xc2      ;; movslq %edx,%rax                              5
    #x48 #xc1 #xe0 #x02 ;; shl    $0x2,%rax                              6
    #x90                ;; nop                                           7
    #x8b #x0f           ;; mov    (%rdi),%ecx                            8
    #x48 #x83 #xc7 #x04 ;; add    $0x4,%rdi                              9
    #x89 #x0e           ;; mov    %ecx,(%rsi)                           10
    #x48 #x83 #xc6 #x04 ;; add    $0x4,%rsi                             11
    #x48 #x83 #xc0 #xfc ;; add    $0xfffffffffffffffc,%rax              12
    #x75 #xee           ;; jne    100000ee0 <_copyData+0x10>            13   (jump if ZF = 0)
    #x5d                ;; pop    %rbp                                  14
    #xc3                ;; retq                                         15
    ))
```

Step 2: Define Clock Functions

Showing only the loop clock function here...

```
(defun loop-clk-base () 6)
(defun loop-clk-recur () 6)

(defun loop-clk (m)
  (if (signed-byte-p 64 m)
      (let ((new-m (loghead 64 (+ #xfffffffffffffffc m))))
        (if (<= m 4)
            (loop-clk-base)
          (clk+ (loop-clk-recur) (loop-clk new-m))))
    0))
```

Step 3: Define Abstractions

Source Array:

```
(defun-nx source-bytes (i src-ptr x86)
  (mv-nth 1 (rb (create-canonical-address-list
                 i
                 (+ (- i) src-ptr))
                :x x86)))
```

Read `i` bytes from addresses:
`(src-ptr - i), (src-ptr - i + 1), … , (src-ptr - 1).`

Step 3: Define Abstractions

Source Array:

```
(defun-nx source-bytes (i src-ptr x86)
  (mv-nth 1 (rb (create-canonical-address-list
                 i
                 (+ (- i) src-ptr))
                :x x86)))
```

Read `i` bytes from addresses:
`(src-ptr - i), (src-ptr - i + 1), ... , (src-ptr - 1).`

Later, I'll talk about why this definition doesn't do the "natural" thing, i.e., read `i` bytes from `src-ptr` to `(src-ptr + i - 1)`.

Spoiler:
It's a "I-like-it-that-way" decision, not so much a technical one.

Step 3: Define Abstractions

Destination Array: same kind of definition as `source-bytes`

```
(defun-nx destination-bytes (j dst-ptr x86)
  (mv-nth 1 (rb (create-canonical-address-list
                 j
                 (+ (- j) dst-ptr))
                :x x86)))
```

Read `j` bytes from addresses:
`(dst-ptr - j), (dst-ptr - j + 1), ... , (dst-ptr - 1).`

Step 5: Effect Theorems

What's the effect of the loop on the x86 state?

```
(defthm effects-copyData-loop
  (implies
   (loop-preconditions k m addr src-ptr dst-ptr x86)
   (equal (x86-run (loop-clk m) x86)
          ???)))
```

Step 4: Figure Out the Pre-Conditions

I think that this is the hardest step of them all.

Here, we need to think about the loop invariant too.

Step 4: Figure Out the Pre-Conditions

I think that this is the hardest step of them all.
Here, we need to think about the loop invariant too.

Let's recall how the `copyData` loop works.

In every iteration:

1. 4 bytes from the `src` are copied to the `dst`.
2. `src-ptr` and `dst-ptr` are incremented by 4.
3. Number of bytes to be copied (`m`) is decremented by 4 (using wrap-around addition).
4. If `m` is zero, we jump out of the loop. Otherwise, we iterate.

Step 4: Figure Out the Pre-Conditions

Important:
Every iteration of the loop modifies a different set of memory locations.

Step 4: Figure Out the Pre-Conditions

Important:
Every iteration of the loop modifies a different set of memory locations.

m:
number of bytes to be copied
decreases by 4 in every iteration

Step 4: Figure Out the Pre-Conditions

Important:
Every iteration of the loop modifies a different set of memory locations.

m:
number of bytes to be copied
decreases by 4 in every iteration

k:
number of bytes already copied
increases by 4 in every iteration

(m + k):
Remains constant in every iteration

Step 4: Figure Out the Pre-Conditions

Important:
Every iteration of the loop modifies a different set of memory locations.

m:
number of bytes to be copied
decreases by 4 in every iteration

k:
number of bytes already copied
increases by 4 in every iteration

Initial value: `0`

(m + k):
Remains constant in every iteration

Step 4: Figure Out the Pre-Conditions

```
;; Initial x86 state is well-formed.
(x86p x86)
(xr :programmer-level-mode 0 x86)
(equal (xr :ms 0 x86) nil)
(equal (xr :fault 0 x86) nil)
```

Step 4: Figure Out the Pre-Conditions

```
;; Initial x86 state is well-formed.
(x86p x86)
(xr :programmer-level-mode 0 x86)
(equal (xr :ms 0 x86) nil)
(equal (xr :fault 0 x86) nil)

;; For convenience, name some parts of the state.
(equal (xr :rgf *rdi* x86) src-ptr)
(equal (xr :rgf *rsi* x86) dst-ptr)
;; m = Number of bytes to be copied
(equal (xr :rgf *rax* x86) m)
```

Step 4: Figure Out the Pre-Conditions

```
;; Initial x86 state is well-formed.
(x86p x86)
(xr :programmer-level-mode 0 x86)
(equal (xr :ms 0 x86) nil)
(equal (xr :fault 0 x86) nil)

;; For convenience, name some parts of the state.
(equal (xr :rgf *rdi* x86) src-ptr)
(equal (xr :rgf *rsi* x86) dst-ptr)
;; m = Number of bytes to be copied
(equal (xr :rgf *rax* x86) m)

(unsigned-byte-p 33 m)
(equal (mod m 4) 0)
(posp m)
;; k = Number of bytes already copied
(unsigned-byte-p 33 k)
(equal (mod k 4) 0)
(unsigned-byte-p 33 (+ m k))
```

Step 4: Figure Out the Pre-Conditions

```
;; Program is located at address “addr”.
(program-at
 (create-canonical-address-list (len *copyData*) addr)
 *copyData* x86)

;; Poised to execute first instruction of the loop:
(equal addr (+ -16 (xr :rip 0 x86)))

;; All program addresses are canonical.
(canonical-address-p addr)
(canonical-address-p (+ (len *copyData*) addr))
```

Step 4: Figure Out the Pre-Conditions

```
;; All the destination addresses are canonical.
(canonical-address-p (+ (- k) dst-ptr))
(canonical-address-p (+ m dst-ptr))

;; All the source addresses are canonical.
(canonical-address-p (+ (- k) src-ptr))
(canonical-address-p (+ m src-ptr))
```

Step 4: Figure Out the Pre-Conditions

```
;; Memory locations of interest are disjoint.

(disjoint-p ;; Program addresses and destination addresses
 (create-canonical-address-list (len *copyData*) addr)
 (create-canonical-address-list (+ m k)
                                (+ (- k) dst-addr)))
```

Step 4: Figure Out the Pre-Conditions

```
;; Memory locations of interest are disjoint.

(disjoint-p ;; Program addresses and destination addresses
 (create-canonical-address-list (len *copyData*) addr)
 (create-canonical-address-list (+ m k)
                                (+ (- k) dst-addr)))

(disjoint-p ;; Return addresses and destination addresses
 (create-canonical-address-list 8
                                (+ 8 (xr :rgf *rsp* x86)))
 (create-canonical-address-list (+ m k)
                                (+ (- k) dst-addr)))
```

Step 4: Figure Out the Pre-Conditions

```
;; Memory locations of interest are disjoint.

(disjoint-p ;; Program addresses and destination addresses
 (create-canonical-address-list (len *copyData*) addr)
 (create-canonical-address-list (+ m k)
                                (+ (- k) dst-addr)))

(disjoint-p ;; Return addresses and destination addresses
 (create-canonical-address-list 8
                                (+ 8 (xr :rgf *rsp* x86)))
 (create-canonical-address-list (+ m k)
                                (+ (- k) dst-addr)))

(disjoint-p ;; Source addresses and destination addresses
 (create-canonical-address-list (+ m k)
                                (+ (- k) src-addr))
 (create-canonical-address-list (+ m k)
                                (+ (- k) dst-addr)))
```

Step 4: Figure Out the Pre-Conditions

```
;; Values copied in the previous iterations
;; of the loop are unaltered.

;; If k > 0:

;; dst[(dst-ptr - k) to (dst-ptr - 1)]  ==
;; src[(src-ptr - k) to (src-ptr - 1)]

;; If k == 0: trivially true.

(equal (destination-bytes k dst-ptr x86)
       (source-bytes      k src-ptr x86))
```

Step 4: Figure Out the Pre-Conditions

```
;; All the stack addresses are canonical.
(canonical-address-p (xr :rgf *rsp* x86))
(canonical-address-p (+ 8 (xr :rgf *rsp* x86)))

;; Return address of the copyData is canonical.
(canonical-address-p
 (logext
  64
  (combine-bytes
   (mv-nth 1
           (rb (create-canonical-address-list
                8 (+ 8 (xr :rgf *rsp* x86)))
               :r x86))))))
```

Step 5: Effect Theorems

```
(defthm effects-copyData-loop
  (implies
   (loop-preconditions k m addr src-ptr dst-ptr x86)
   (equal (x86-run (loop-clk m) x86)
          ???)))
```

Step 5: Effect Theorems

```
  (defthm effects-copyData-loop
    (implies
     (loop-preconditions k m addr src-ptr dst-ptr x86)
     (equal (x86-run (loop-clk m) x86)
            ???)))


(defthmd effects-copyData-loop-base
  (implies
   (and (equal m 4)
        (loop-preconditions k m addr src-ptr dst-ptr x86))
   (equal (x86-run (loop-clk-base) x86)
          ???)))
```

Step 5: Effect Theorems

```
  (defthm effects-copyData-loop
    (implies
     (loop-preconditions k m addr src-ptr dst-ptr x86)
     (equal (x86-run (loop-clk m) x86)
            ???)))


(defthmd effects-copyData-loop-base
  (implies
   (and (equal m 4)
        (loop-preconditions k m addr src-ptr dst-ptr x86))
   (equal (x86-run (loop-clk-base) x86)
          ???)))

(defthmd effects-copyData-loop-recur
  (implies
   (and (< 4 m)
        (loop-preconditions k m addr src-ptr dst-ptr x86))
   (equal (x86-run (loop-clk-recur) x86)
          ???)))
```

Step 5: Effect Theorems: Loop's Last Iteration

```
(defthmd effects-copyData-loop-base
  (implies
   (and (equal m 4)
        (loop-preconditions k m addr src-ptr dst-ptr x86))
   (equal (x86-run (loop-clk-base) x86)
          (XW
           :RGF *RAX* 0
           ....
           (MV-NTH
            1
            (WB
             (CREATE-ADDR-BYTES-ALIST
              (CREATE-CANONICAL-ADDRESS-LIST
               4 DST-PTR)
              (MV-NTH 1
                      (RB
                       (CREATE-CANONICAL-ADDRESS-LIST
                        4 SRC-PTR)
                       :X X86))))
            X86)))))
```

Step 5: Effect Theorems: dst in the Last Iteration

```
(defthm loop-base-destination-bytes-projection

  ;; dst[(+ -k dst-ptr) to (dst-ptr + 3)]
  ;;   in (x86-run (loop-clk-base) x86) ==

  ;; src[(+ -k src-ptr) to (src-ptr + 3)]
  ;;   in x86

  (implies
   (and (loop-preconditions k m addr src-ptr dst-ptr x86)
        (equal m 4))
   (equal (destination-bytes (+ 4 k) (+ 4 dst-ptr)
                                (x86-run (loop-clk-base) x86))
          (source-bytes (+ 4 k) (+ 4 src-ptr) x86))))
```

Step 5: Effect Theorems: A Loop Iteration (not the last)

```
(defthmd effects-copyData-loop-recur
  (implies
   (and (< 4 m)
        (loop-preconditions k m addr src-ptr dst-ptr x86))
   (equal
    (x86-run (loop-clk-recur) x86)
    (XW
     :RGF *RAX*
     (LOGHEAD 64 (+ #XFFFFFFFFFFFFFFFC M))
     ...
     (MV-NTH
      1
      (WB
       (CREATE-ADDR-BYTES-ALIST
        (CREATE-CANONICAL-ADDRESS-LIST 4 DST-PTR)
        (MV-NTH 1 (RB (CREATE-CANONICAL-ADDRESS-LIST
                       4 SRC-PTR)
                      :X X86)))
       X86))))))
```

Step 5: Effect Theorems: dst in an Iteration (not the last)

```
(defthm loop-recur-destination-bytes-projection

  ;; dst[(+ -k dst-ptr) to (dst-ptr + 3)]
  ;;   in (x86-run (loop-clk-recur) x86) ==

  ;; src[(+ -k src-ptr) to (src-ptr + 3)]
  ;;   in x86

  (implies
   (and (< 4 m)
        (loop-preconditions k m addr src-ptr dst-ptr x86))
   (equal (destination-bytes (+ 4 k) (+ 4 dst-ptr)
                             (x86-run (loop-clk-recur) x86))
          (source-bytes (+ 4 k) (+ 4 src-ptr) x86))))
```

Step 5: Effect Theorems

Characterizing the state after the loop has run to completion:

```
(defthm effects-copyData-loop
  (implies
   (loop-preconditions k m ptr src-ptr dst-ptr x86)
   (equal (x86-run (loop-clk m) x86)
          (loop-state k m src-ptr dst-ptr x86))))
```

I like to think about x86 states, not clocks.

Also, induction scheme suggested by `loop-state` is more suitable than the one by `loop-clk`.

Step 5: Effect Theorems

```
(defun-nx loop-state (k m src-ptr dst-ptr x86)
  (if (signed-byte-p 64 m)
      (if (<= m 4)
          (x86-run (loop-clk-base) x86)
         (b* ((new-m
                 (loghead 64 (+ #xfffffffffffffffc m)))
                (new-k (+ 4 k))
                (new-src-ptr (+ 4 src-ptr))
                (new-dst-ptr (+ 4 dst-ptr))
                (x86 (x86-run (loop-clk-recur) x86)))
               (loop-state new-k new-m
                           new-src-ptr new-dst-ptr
                           x86)))
    x86))
```

Step 5: Effect Theorems: Proving `effects-copyData-loop`

Induction Scheme:

```
(AND (IMPLIES (NOT (SIGNED-BYTE-P 64 M))
              (:P ADDR DST-ADDR K M SRC-ADDR X86))

     (IMPLIES (AND (SIGNED-BYTE-P 64 M)
                   (< 4 M)
                   (:P ADDR
                       (+ 4 DST-ADDR)
                       (+ 4 K)
                       (LOGHEAD 64 (+ 18446744073709551612 M))
                       (+ 4 SRC-ADDR)
                       (X86-RUN (LOOP-CLK-RECUR) X86)))
              (:P ADDR DST-ADDR K M SRC-ADDR X86))

     (IMPLIES (AND (SIGNED-BYTE-P 64 M) (<= M 4))
              (:P ADDR DST-ADDR K M SRC-ADDR X86)))
```

Step 5: Effect Theorems: Proving `effects-copyData-loop`

```
Subgoal *1/3
(IMPLIES (NOT (SIGNED-BYTE-P 64 M))
         (IMPLIES
          (LOOP-PRECONDITIONS K M ADDR SRC-ADDR DST-ADDR X86)
          (EQUAL (X86-RUN (LOOP-CLK M) X86)
                 (LOOP-STATE K M SRC-ADDR DST-ADDR X86))))
```

Step 5: Effect Theorems: Proving `effects-copyData-loop`

```
Subgoal *1/2
(IMPLIES
 (AND
  (SIGNED-BYTE-P 64 M)
  (< 4 M)
  (IMPLIES
   (LOOP-PRECONDITIONS (+ 4 K)
                       (LOGHEAD 64 (+ 18446744073709551612 M))
                       ADDR
                       (+ 4 SRC-ADDR)
                       (+ 4 DST-ADDR)
                       (X86-RUN (LOOP-CLK-RECUR) X86))
   (EQUAL (X86-RUN (LOOP-CLK (LOGHEAD 64 (+ 18446744073709551612 M)))
                   (X86-RUN (LOOP-CLK-RECUR) X86))
          (LOOP-STATE (+ 4 K)
                      (LOGHEAD 64 (+ 18446744073709551612 M))
                      (+ 4 SRC-ADDR)
                      (+ 4 DST-ADDR)
                      (X86-RUN (LOOP-CLK-RECUR) X86)))))
 (IMPLIES
  (LOOP-PRECONDITIONS K M ADDR SRC-ADDR DST-ADDR X86)
  (EQUAL (X86-RUN (LOOP-CLK M) X86)
         (LOOP-STATE K M SRC-ADDR DST-ADDR X86))))
```

Step 5: Effect Theorems: Proving `effects-copyData-loop`

To discharge Subgoal `*1/2`:

```
(defthm loop-recur-implies-loop-preconditions
  (implies
   (and (< 4 m)
        (loop-preconditions k m addr src-ptr dst-ptr x86))
   (loop-preconditions (+ 4 k)
                       (loghead 64 (+ #xfffffffffffffffc m))
                       addr
                       (+ 4 src-ptr)
                       (+ 4 dst-ptr)
                       (x86-run (loop-clk-recur) x86))))
```

Step 5: Effect Theorems: Proving `effects-copyData-loop`

```
Subgoal *1/1
(IMPLIES (AND (SIGNED-BYTE-P 64 M) (<= M 4))
         (IMPLIES
          (LOOP-PRECONDITIONS K M ADDR SRC-ADDR DST-ADDR X86)
          (EQUAL (X86-RUN (LOOP-CLK M) X86)
                 (LOOP-STATE K M SRC-ADDR DST-ADDR X86))))
```

Step 5: Effect Theorems

Characterizing the state after the loop has run to completion:

```
(defthm effects-copyData-loop
  (implies
   (loop-preconditions k m ptr src-ptr dst-ptr x86)
   (equal (x86-run (loop-clk m) x86)
          (loop-state k m src-ptr dst-ptr x86))))
```

Q.E.D.

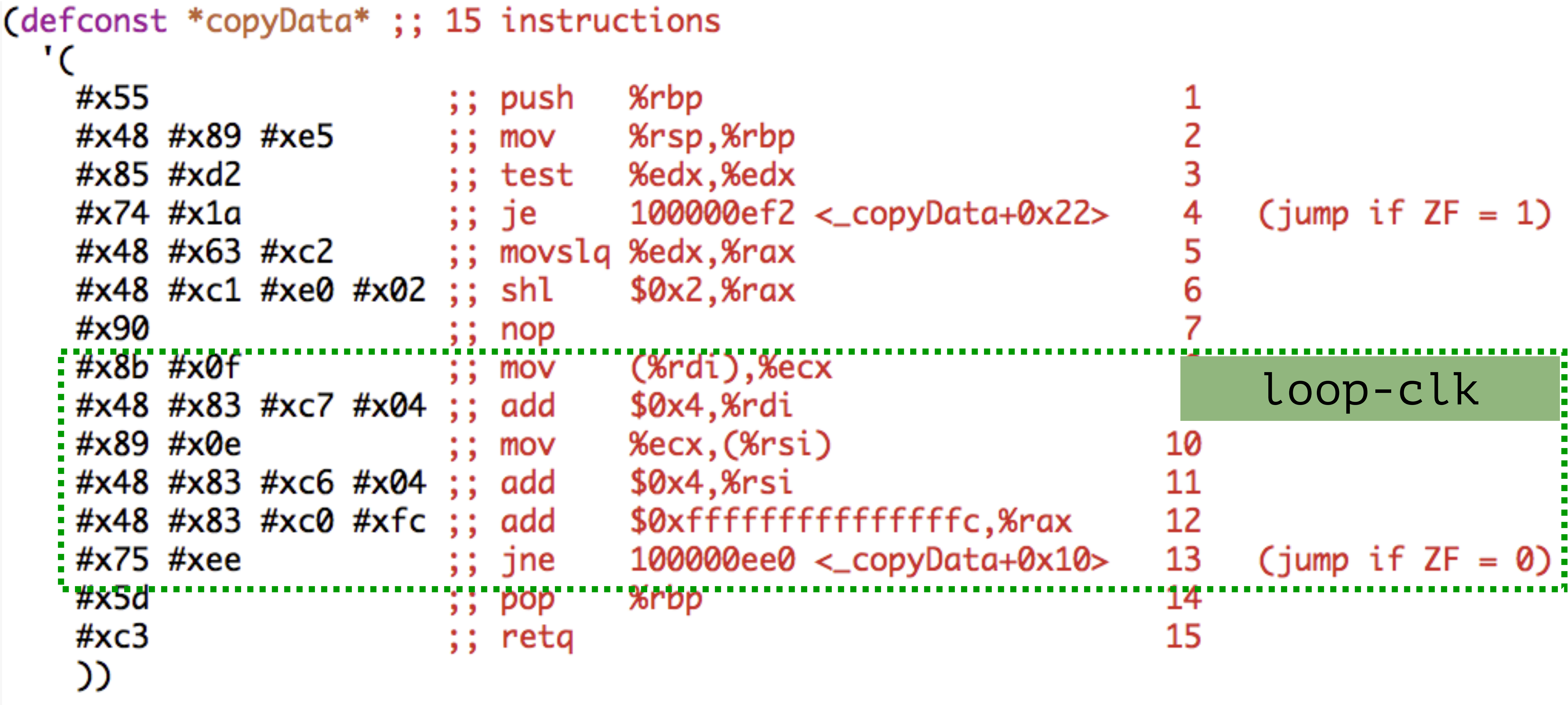
Step 5: Effect Theorems

```
(defthmd destination-array-and-loop-state

  ;; dst[(+ -k dst-ptr) to (dst-ptr + m - 1)]
  ;;   in (loop-state k m src-ptr dst-ptr x86) ==

  ;; src[(+ -k src-ptr) to (src-ptr + m - 1)]
  ;;   in x86

  (implies
   (and (loop-preconditions k m addr src-ptr dst-ptr x86)
        (natp k))
   (equal
    (destination-bytes
     (+ k m)
     (+ m dst-ptr)
     (loop-state k m src-ptr dst-ptr x86))
    (source-bytes (+ k m) (+ m src-ptr) x86))))
```

Step 5: Effect Theorems

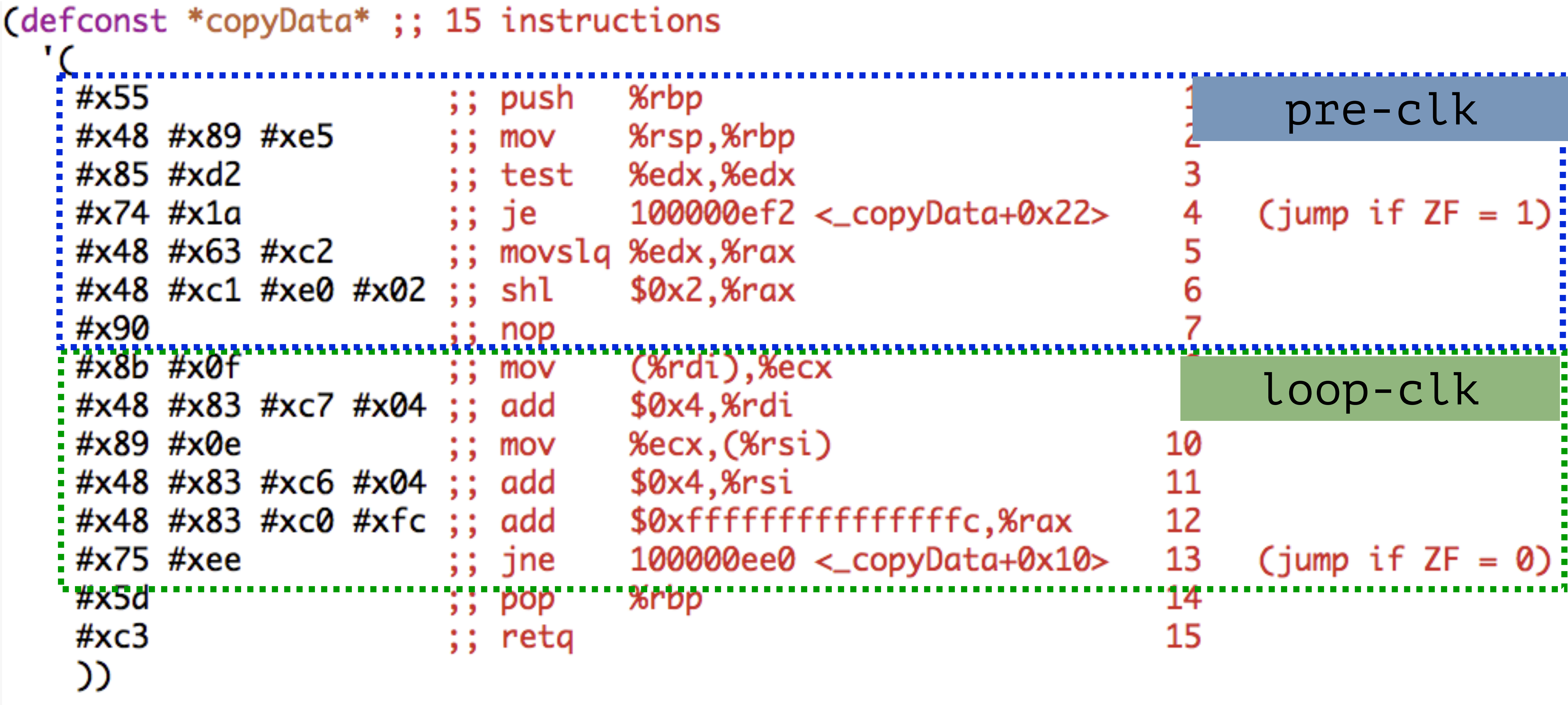
```
(defthm destination-array-and-x86-state-after-loop

  ;; dst[(+ -k dst-ptr) to (dst-ptr + m - 1)]
  ;;   in (x86-run (loop-clk m) x86) ==

  ;; src[(+ -k src-ptr) to (src-ptr + m - 1)]
  ;;   in x86

  (implies
   (and (loop-preconditions k m addr src-ptr dst-ptr x86)
        (natp k))
   (equal
    (destination-bytes
     (+ k m)
     (+ m dst-ptr)
     (x86-run (loop-clk m) x86))
    (source-bytes (+ k m) (+ m src-ptr) x86))))
```

Step 6: Composition and Other Final Touches



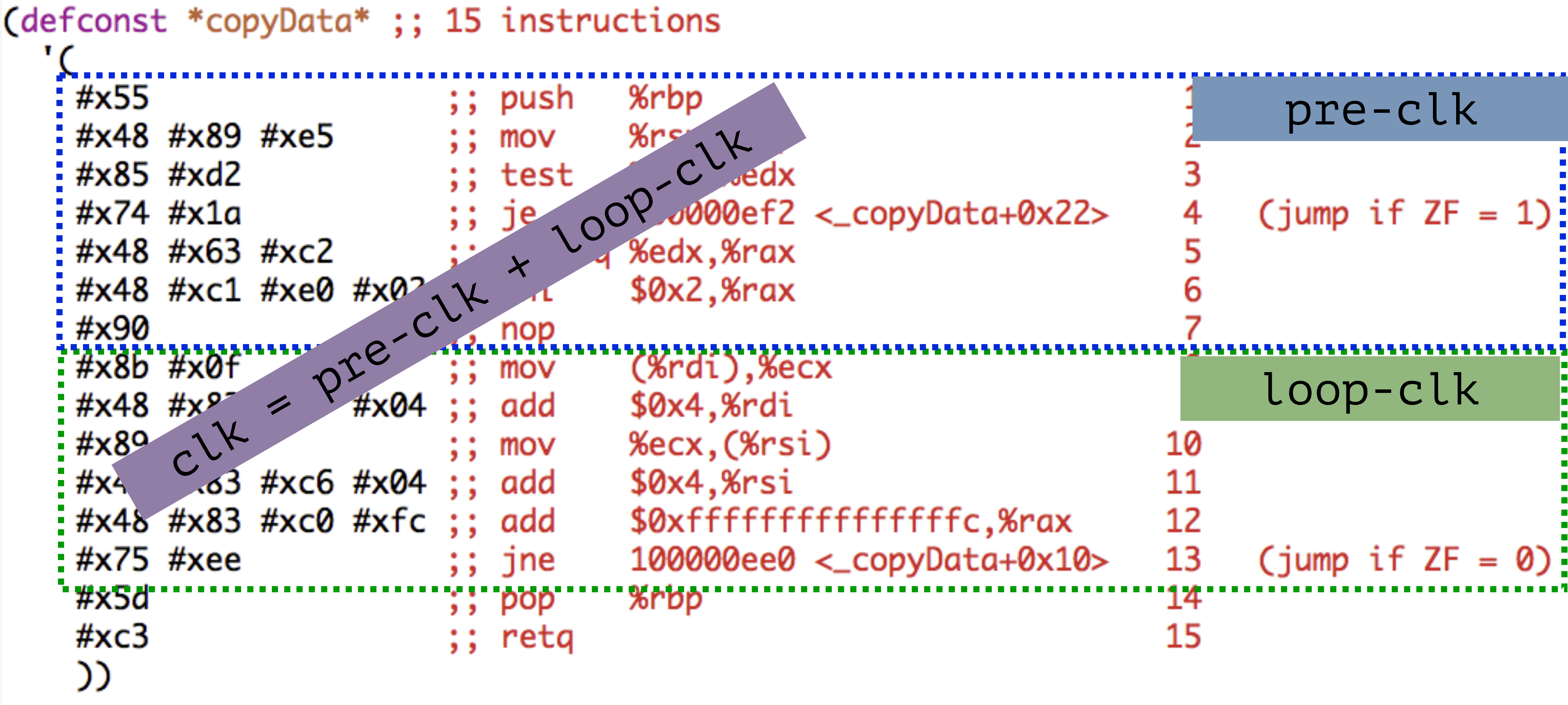
```
(defconst *copyData* ;; 15 instructions
  '(
    #x55                ;; push   %rbp                             1
    #x48 #x89 #xe5      ;; mov    %rsp,%rbp                        2
    #x85 #xd2           ;; test   %edx,%edx                        3
    #x74 #x1a           ;; je     100000ef2 <_copyData+0x22>       4   (jump if ZF = 1)
    #x48 #x63 #xc2      ;; movslq %edx,%rax                        5
    #x48 #xc1 #xe0 #x02 ;; shl    $0x2,%rax                        6
    #x90                ;; nop                                     7
    #x8b #x0f           ;; mov    (%rdi),%ecx
    #x48 #x83 #xc7 #x04 ;; add    $0x4,%rdi
    #x89 #x0e           ;; mov    %ecx,(%rsi)                     10
    #x48 #x83 #xc6 #x04 ;; add    $0x4,%rsi                       11
    #x48 #x83 #xc0 #xfc ;; add    $0xfffffffffffffffc,%rax        12
    #x75 #xee           ;; jne    100000ee0 <_copyData+0x10>      13   (jump if ZF = 0)
    #x5d                ;; pop    %rbp                            14
    #xc3                ;; retq                                   15
    ))
```

Step 6: Composition and Other Final Touches



```
(defconst *copyData* ;; 15 instructions
  '(
   #x55                ;; push   %rbp                                  1
   #x48 #x89 #xe5      ;; mov    %rsp,%rbp                             2
   #x85 #xd2           ;; test   %edx,%edx                             3
   #x74 #x1a           ;; je     100000ef2 <_copyData+0x22>            4   (jump if ZF = 1)
   #x48 #x63 #xc2      ;; movslq %edx,%rax                             5
   #x48 #xc1 #xe0 #x02 ;; shl    $0x2,%rax                             6
   #x90                ;; nop                                          7
   #x8b #x0f           ;; mov    (%rdi),%ecx
   #x48 #x83 #xc7 #x04 ;; add    $0x4,%rdi
   #x89 #x0e           ;; mov    %ecx,(%rsi)                          10
   #x48 #x83 #xc6 #x04 ;; add    $0x4,%rsi                            11
   #x48 #x83 #xc0 #xfc ;; add    $0xfffffffffffffffc,%rax             12
   #x75 #xee           ;; jne    100000ee0 <_copyData+0x10>           13   (jump if ZF = 0)
   #x5d                ;; pop    %rbp                                 14
   #xc3                ;; retq                                        15
   ))
```

Step 6: Composition and Other Final Touches



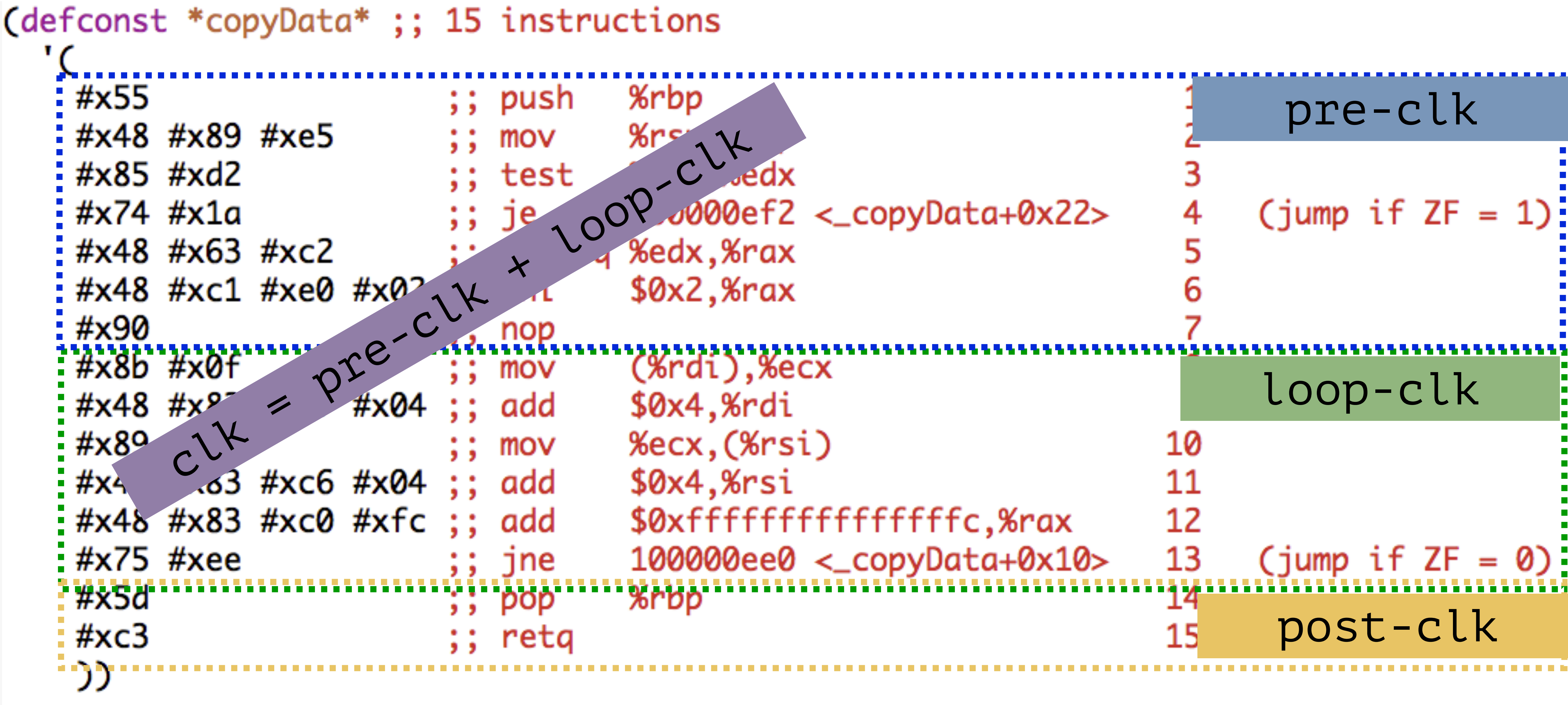
```
(defconst *copyData* ;; 15 instructions
  '(
    #x55                 ;; push   %rbp                                1
    #x48 #x89 #xe5       ;; mov    %rs                                 2
    #x85 #xd2            ;; test       edx                             3
    #x74 #x1a            ;; je      0000ef2 <_copyData+0x22>           4   (jump if ZF = 1)
    #x48 #x63 #xc2       ;;       q %edx,%rax                          5
    #x48 #xc1 #xe0 #x0         l   $0x2,%rax                           6
    #x90                    nop                                        7
    #x8b #x0f            ;; mov    (%rdi),%ecx
    #x48 #x8         #x04 ;; add   $0x4,%rdi
    #x89                 ;; mov    %ecx,(%rsi)                         10
    #x4   83 #xc6 #x04   ;; add    $0x4,%rsi                           11
    #x48 #x83 #xc0 #xfc  ;; add    $0xfffffffffffffffc,%rax            12
    #x75 #xee            ;; jne    100000ee0 <_copyData+0x10>          13  (jump if ZF = 0)
    #x5d                 ;; pop    %rbp                                14
    #xc3                 ;; retq                                       15
    ))
```

pre-clk

loop-clk

clk = pre-clk + loop-clk

Step 6: Composition and Other Final Touches



```
(defconst *copyData* ;; 15 instructions
  '(
    #x55                    ;; push   %rbp
    #x48 #x89 #xe5          ;; mov    %rs
    #x85 #xd2               ;; test         edx                            3
    #x74 #x1a               ;; je         0000ef2 <_copyData+0x22>         4   (jump if ZF = 1)
    #x48 #x63 #xc2          ;;         q %edx,%rax                         5
    #x48 #xc1 #xe0 #x0      ;;          $0x2,%rax                          6
    #x90                    ;;  nop                                        7
    #x8b #x0f               ;; mov    (%rdi),%ecx
    #x48 #x8         #x04   ;; add    $0x4,%rdi
    #x89                    ;; mov    %ecx,(%rsi)                         10
    #x4    83 #xc6 #x04     ;; add    $0x4,%rsi                           11
    #x48 #x83 #xc0 #xfc     ;; add    $0xfffffffffffffffc,%rax            12
    #x75 #xee               ;; jne    100000ee0 <_copyData+0x10>          13   (jump if ZF = 0)
    #x5d                    ;; pop    %rbp                                14
    #xc3                    ;; retq                                       15
    ))
```

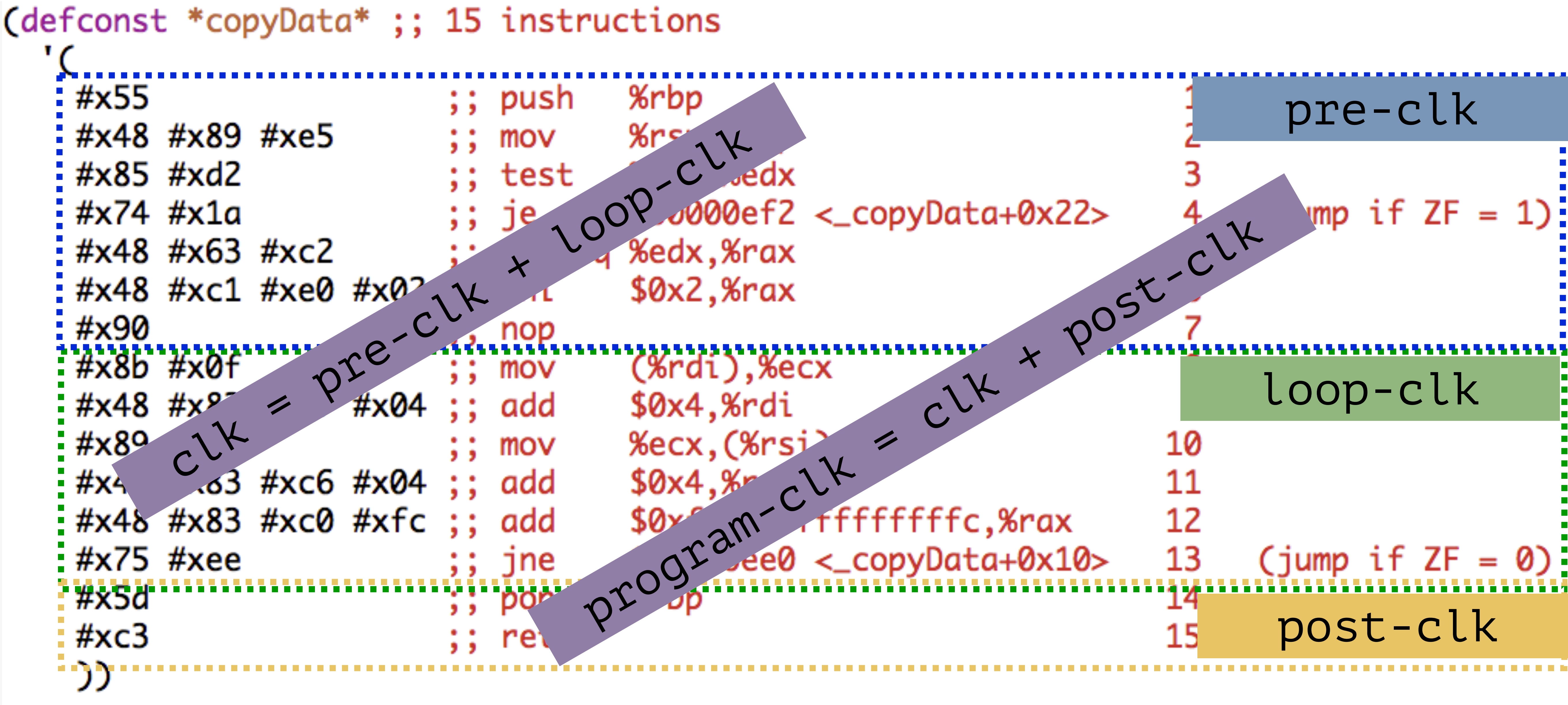
pre-clk

loop-clk

post-clk

clk = pre-clk + loop-clk

Step 6: Composition and Other Final Touches



```
(defconst *copyData* ;; 15 instructions
  '(
    #x55                     ;; push   %rbp                               1
    #x48 #x89 #xe5           ;; mov    %rs                                2
    #x85 #xd2                ;; test     edx                              3
    #x74 #x1a                ;; je      0000ef2 <_copyData+0x22>         4      mp if ZF = 1)
    #x48 #x63 #xc2           ;;       q %edx,%rax
    #x48 #xc1 #xe0 #x0              $0x2,%rax
    #x90                        nop                                      7
    #x8b #x0f                ;; mov    (%rdi),%ecx
    #x48 #x8          #x04   ;; add    $0x4,%rdi
    #x89                     ;; mov    %ecx,(%rsi                        10
    #x4    83 #xc6 #x04      ;; add    $0x4,%r                           11
    #x48 #x83 #xc0 #xfc      ;; add    $0x        ffffffffc,%rax        12
    #x75 #xee                ;; jne        ee0 <_copyData+0x10>          13   (jump if ZF = 0)
    #x5d                     ;; po          bp                           14
    #xc3                     ;; re                                       15
    ))
```

pre-clk

loop-clk

post-clk

clk = pre-clk + loop-clk

program-clk = clk + post-clk

Step 6: Composition and Other Final Touches

```
(defthm preconditions-implies-loop-preconditions
  (implies
   (and (preconditions n addr x86)
        (not (zp n))
        (equal m (ash n 2)))
   (loop-preconditions
    0 m addr
    (xr :rgf *rdi* x86) ;; src-ptr
    (xr :rgf *rsi* x86) ;; dst-ptr
    (x86-run (pre-clk n) x86))))
```

Step 6: Composition and Other Final Touches

```
(defthm preconditions-implies-loop-preconditions
  (implies
   (and (preconditions n addr x86)
        (not (zp n))
        (equal m (ash n 2)))
   (loop-preconditions
    0 m addr
    (xr :rgf *rdi* x86) ;; src-ptr
    (xr :rgf *rsi* x86) ;; dst-ptr
    (x86-run (pre-clk n) x86))))
```

Step 6: Composition and Other Final Touches

```
(defthm preconditions-implies-loop-preconditions
  (implies
   (and (preconditions n addr x86)
        (not (zp n))
        (equal m (ash n 2)))
   (loop-preconditions
    0 m addr
    (xr :rgf *rdi* x86) ;; src-ptr
    (xr :rgf *rsi* x86) ;; dst-ptr
    (x86-run (pre-clk n) x86))))
```

`loop-preconditions` are the post-conditions for the 7 instructions preceding the loop.

Step 6: Composition and Other Final Touches

By transitivity:

```
(defthm clk-copies-m-bytes-from-source-to-destination
  (implies
   (and (preconditions n addr x86)
        (not (zp n))
        (equal m (ash n 2)))
   (equal
    (destination-bytes
     m
     (+ m (xr :rgf *rsi* x86))
     (x86-run (clk n) x86))
    (source-bytes m (+ m (xr :rgf *rdi* x86)) x86))))
```

Step 6: Composition and Other Final Touches

And do more compositions to get the final theorem about a successful copy:

```
(defthm destination-array-is-a-copy-of-the-source-array
  (implies
   (and (preconditions n addr x86)
        (equal m (ash n 2)))
   (equal
    (destination-bytes
     m
     (+ m (xr :rgf *rsi* x86))
     (x86-run (program-clk n) x86))
    (source-bytes
     m
     (+ m (xr :rgf *rdi* x86))
     x86))))
```

Conclusion

And… we’re done. Whew.

Conclusion

And… we're done. Whew.

Wait. Where's the specification function of this program?

`copyData` is a "state-modification" program. I didn't choose to write an explicit specification function.

Conclusion

And… we're done. Whew.

Wait. Where's the specification function of this program?

`copyData` is a "state-modification" program. I didn't choose to write an explicit specification function.

Verification of other programs that do some computation (e.g., a factorial program) would add at least another step to this process — namely, writing formal specifications.

Reasoning about `copyData`

Yet Another Account of a Proof of Correctness of an x86 Machine-Code Program

Shilpi Goel

ACL2 Seminar

19th October, 2015

BTW… My Proposed Dissertation Project

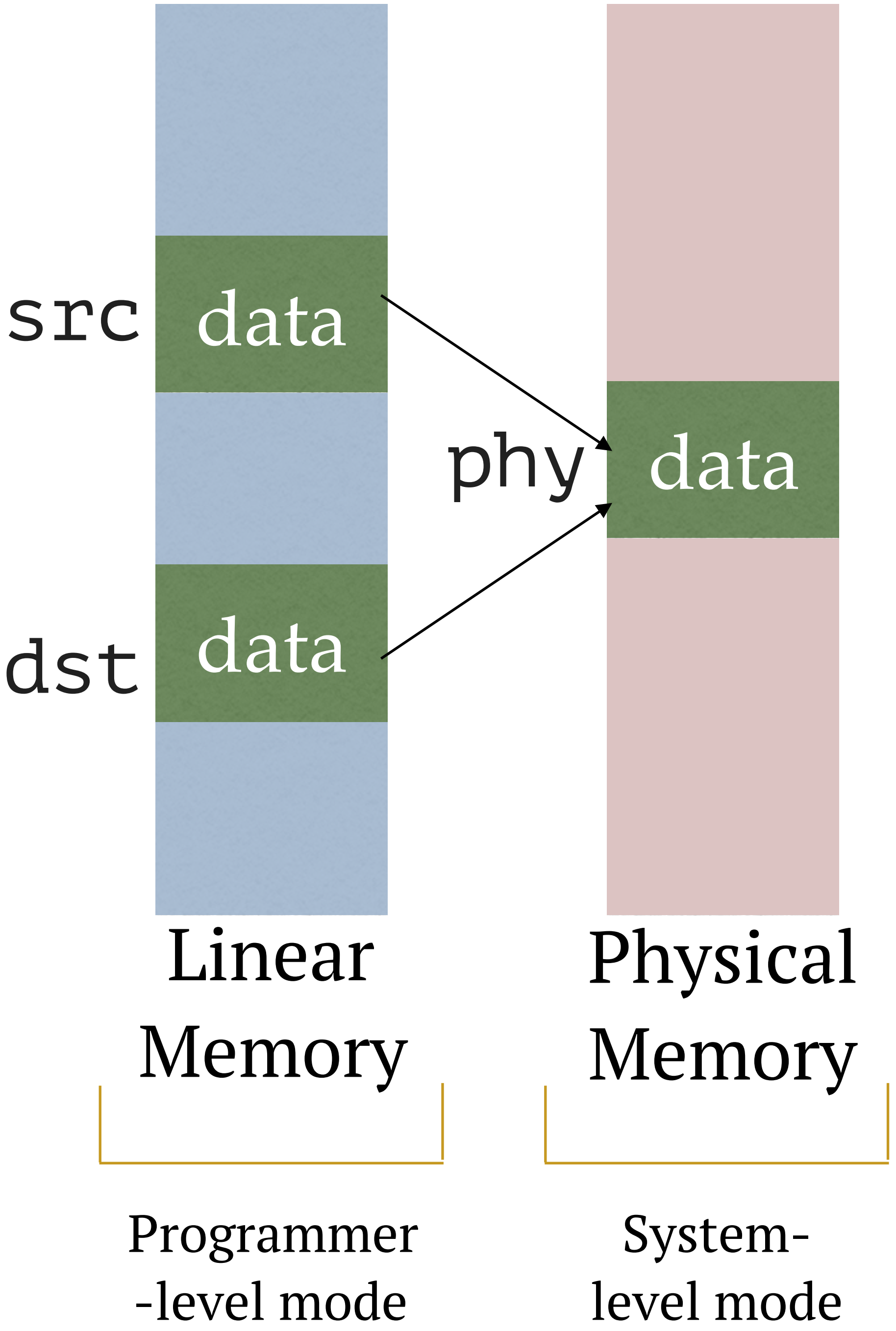
Formal Analysis of an Optimized Data-Copy Program

Specification:
Copy `data` from linear memory location `src` to disjoint linear memory location `dst`.

Verification Objective:
After a successful copy, `src` and `dst` contain `data`.

Implementation:
Include the *copy-on-write* technique:
`src` and `dst` can be mapped to the same physical memory location `phy`.

- System calls
- Page mapping
- Privileges
- Context Switches



BTW… My Proposed Dissertation Project

Formal Analysis of an Optimized Data-Copy Program

Specification:
Copy `data` from linear memory location `src` to disjoint linear memory location `dst`.

Verification Objective:
After a successful copy, `src` and `dst` contain `data`.

Implementation:
Include the *copy-on-write* technique:
`src` and `dst` can be mapped to the same physical memory location `phy`.

- System calls
- Page mapping
- Privileges
- Context Switches