ECSE 324

Computer Organization

Software – Assemblers, Linkers, Compilers & Debuggers

Prof. Christophe Dubach

Original slides from: Prof. Derek Nowrouzezahrai

Outline

You've discussed the **behavior** of assembly instructions and the **operations** they can perform

- the process of *implementing* an algorithm using assembly instructions should be clear at this stage

We will discuss the *pragmatics* of how to **program** and **run** algorithms on a computing **platform**

We will discuss the *pragmatics* of how to **program** and **run** algorithms on a computing **platform**

- from assembly to machine instructions
- transitioning to higher-level languages
- execution and management of machine code

```
;--------------------------------------------------------------
; zstr_count:
; Counts a zero-terminated ASCII string to determine its size
; in:   eax = start address of the zero terminated string
; out:  ecx = count = the length of the string

zstr_count:                     ; Entry point
    mov  ecx, -1                ; Init the loop counter, pre-decrement
                                ;  to compensate for the increment
.loop:
    inc  ecx                    ; Add 1 to the loop counter
    cmp  byte [eax + ecx], 0    ; Compare the value at the string's
                                ;  [starting memory address Plus the
                                ;  loop offset], to zero
    jne  .loop                  ; If the memory value is not zero,
                                ;  then jump to the label called '.loop',
                                ;  otherwise continue to the next line
.done:
                                ; We don't do a final increment,
                                ;  because even though the count is base 1,
                                ;  we do not include the zero terminator in the
                                ;  string's length
    ret                         ; Return to the calling program
```

→

```
00000030 B9FFFFFFFF

00000035 41
00000036 803C0800

0000003A 75F9

0000003C C3
```

We will discuss the *pragmatics* of how to **program** and **run** algorithms on a computing **platform**

- from assembly to machine instructions
- transitioning to higher-level languages
- execution and management of machine code

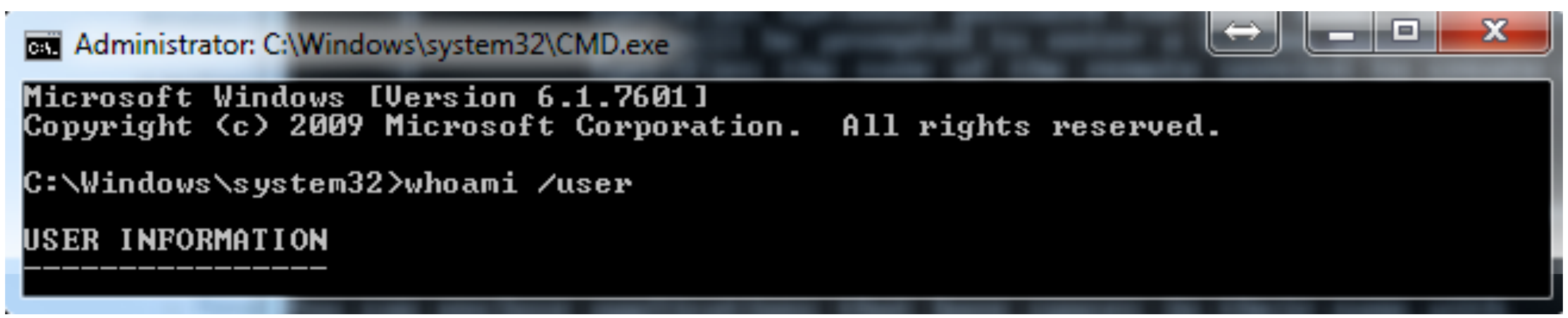
```
/* Length of a string. */

int len = strlen(str);

 /* Traverse a string */
 for(i = 0; i < len; i++){
```

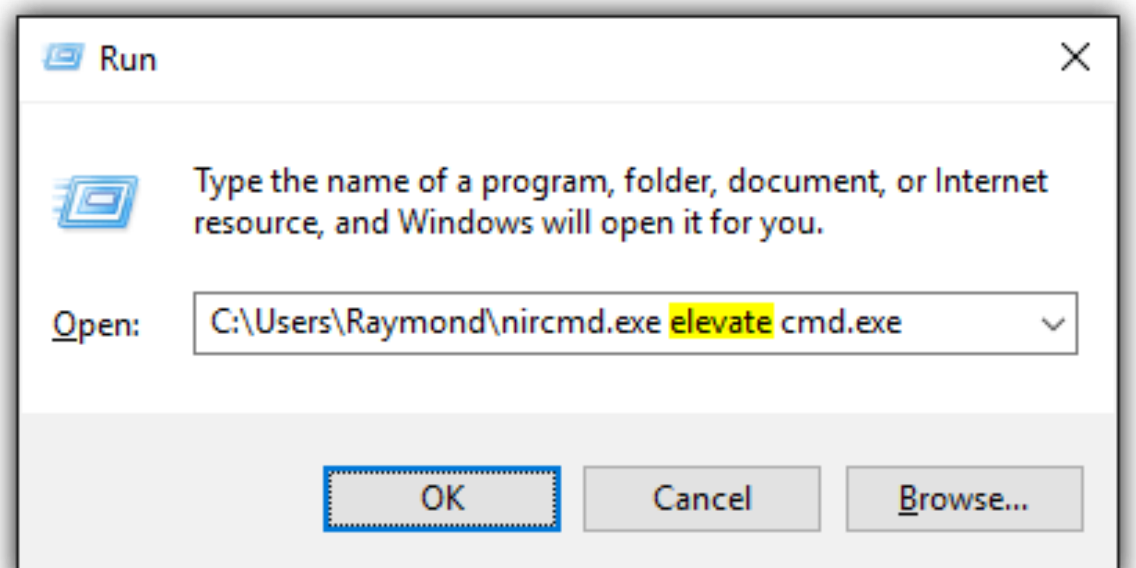
```
zstr_count:
    mov   ecx, -1

.loop:
    inc   ecx
    cmp   byte [eax + ecx], 0


    jne   .loop


.done:




    ret
```

```
00000030 B9FFFFFFFF



00000035 41
00000036 803C0800



0000003A 75F9









0000003C C3
```

We will discuss the *pragmatics* of how to **program** and **run** algorithms on a computing **platform**

- from assembly to machine instructions
- transitioning to higher-level languages
- execution and management of machine code





Assembly Language

Assembly is a convenient* **abstraction** designed for human **creation** and **consumption**

- computers don't naturally "speak" assembly

Before an algorithm, implemented in assembly, can be executed on a computer it must be:

- validated for correctness*
- converted to a form consumable by a computer
    - properly ordered machine code

Enter the Assembler

The assembler is a software tool that:

- verifies assembly code listings for validity, and
- converts valid **assembly opcodes** and **operands** into their associated **machine code values**
- computes a **memory layout** for the machine code

```
;--------------------------------------------------------------
; zstr_count:
; Counts a zero-terminated ASCII string to determine its size
; in:   eax = start address of the zero terminated string
; out:  ecx = count = the length of the string

zstr_count:                     ; Entry point
    mov  ecx, -1                ; Init the loop counter, pre-decrement
                                ;  to compensate for the increment
.loop:
    inc  ecx                    ; Add 1 to the loop counter
    cmp  byte [eax + ecx], 0    ; Compare the value at the string's
                                ;  [starting memory address Plus the
                                ;  loop offset], to zero
    jne  .loop                  ; If the memory value is not zero,
                                ;  then jump to the label called '.loop',
                                ;  otherwise continue to the next line
.done:
                                ; We don't do a final increment,
                                ;  because even though the count is base 1,
                                ;  we do not include the zero terminator in the
                                ;  string's length
    ret                         ; Return to the calling program
```

```
00000030 B9FFFFFFFF

00000035 41
00000036 803C0800

0000003A 75F9

0000003C C3
```

Assembling Source Code

The assembler accepts assembly source listings, stored in an input text file, as input…

```
;-----------------------------------------------------------
; zstr_count:
; Counts a zero-terminated ASCII string to determine its size
; in:   eax = start address of the zero terminated string
; out:  ecx = count = the length of the string

zstr_count:                    ; Entry point
    mov  ecx, -1               ; Init the loop counter, pre-decrement
                               ;  to compensate for the increment
.loop:
    inc  ecx                   ; Add 1 to the loop counter
    cmp  byte [eax + ecx], 0   ; Compare the value at the string's
                               ;  [starting memory address Plus the
                               ;  loop offset], to zero
    jne  .loop                 ; If the memory value is not zero,
                               ;  then jump to the label called '.loop',
                               ;  otherwise continue to the next line
.done:
                               ; We don't do a final increment,
                               ;  because even though the count is base 1,
                               ;  we do not include the zero terminator in the
                               ;  string's length
    ret                        ; Return to the calling program
```

```
00000030 B9FFFFFFFF

00000035 41
00000036 803C0800

0000003A 75F9

0000003C C3
```

Assembling Source Code

… recognizes individual assembly instruction mnemonics (or doesn't!)…

- interprets **addressing modes** and **data operands**

```
;------------------------------------------------------------
; zstr_count:
; Counts a zero-terminated ASCII string to determine its size
; in:   eax = start address of the zero terminated string
; out:  ecx = count = the length of the string

zstr_count:                     ; Entry point
    mov  ecx, -1                ; Init the loop counter, pre-decrement
                                ;  to compensate for the increment
.loop:
    inc  ecx                    ; Add 1 to the loop counter
    cmp  byte [eax + ecx], 0    ; Compare the value at the string's
                                ;  [starting memory address Plus the
                                ;  loop offset], to zero
    jne  .loop                  ; If the memory value is not zero,
                                ;  then jump to the label called '.loop',
                                ;  otherwise continue to the next line
.done:
                                ; We don't do a final increment,
                                ;  because even though the count is base 1,
                                ;  we do not include the zero terminator in the
                                ;  string's length
    ret                         ; Return to the calling program
```

```
00000030 B9FFFFFFFF

00000035 41
00000036 803C0800

0000003A 75F9

0000003C C3
```

Assembling Source Code

… converts them to their associated machine OP binary (or equivalent) codes…

```
;---------------------------------------------------------
; zstr_count:
; Counts a zero-terminated ASCII string to determine its size
; in:   eax = start address of the zero terminated string
; out:  ecx = count = the length of the string

zstr_count:                   ; Entry point
    mov   ecx, -1             ; Init the loop counter, pre-decrement
                              ;  to compensate for the increment
.loop:
    inc   ecx                 ; Add 1 to the loop counter
    cmp   byte [eax + ecx], 0 ; Compare the value at the string's
                              ;  [starting memory address Plus the
                              ;  loop offset], to zero
    jne   .loop               ; If the memory value is not zero,
                              ;  then jump to the label called '.loop',
                              ;  otherwise continue to the next line
.done:
                              ; We don't do a final increment,
                              ;  because even though the count is base 1,
                              ;  we do not include the zero terminator in the
                              ;  string's length
    ret                       ; Return to the calling program
```

```
00000030 B9FFFFFFFF
00000035 41
00000036 803C0800
0000003A 75F9
0000003C C3
```

Assembling Source Code

… lays out the OP codes in (relative) memory…

- usually in a sequential block of memory
- where do empty lines in the layout come from?

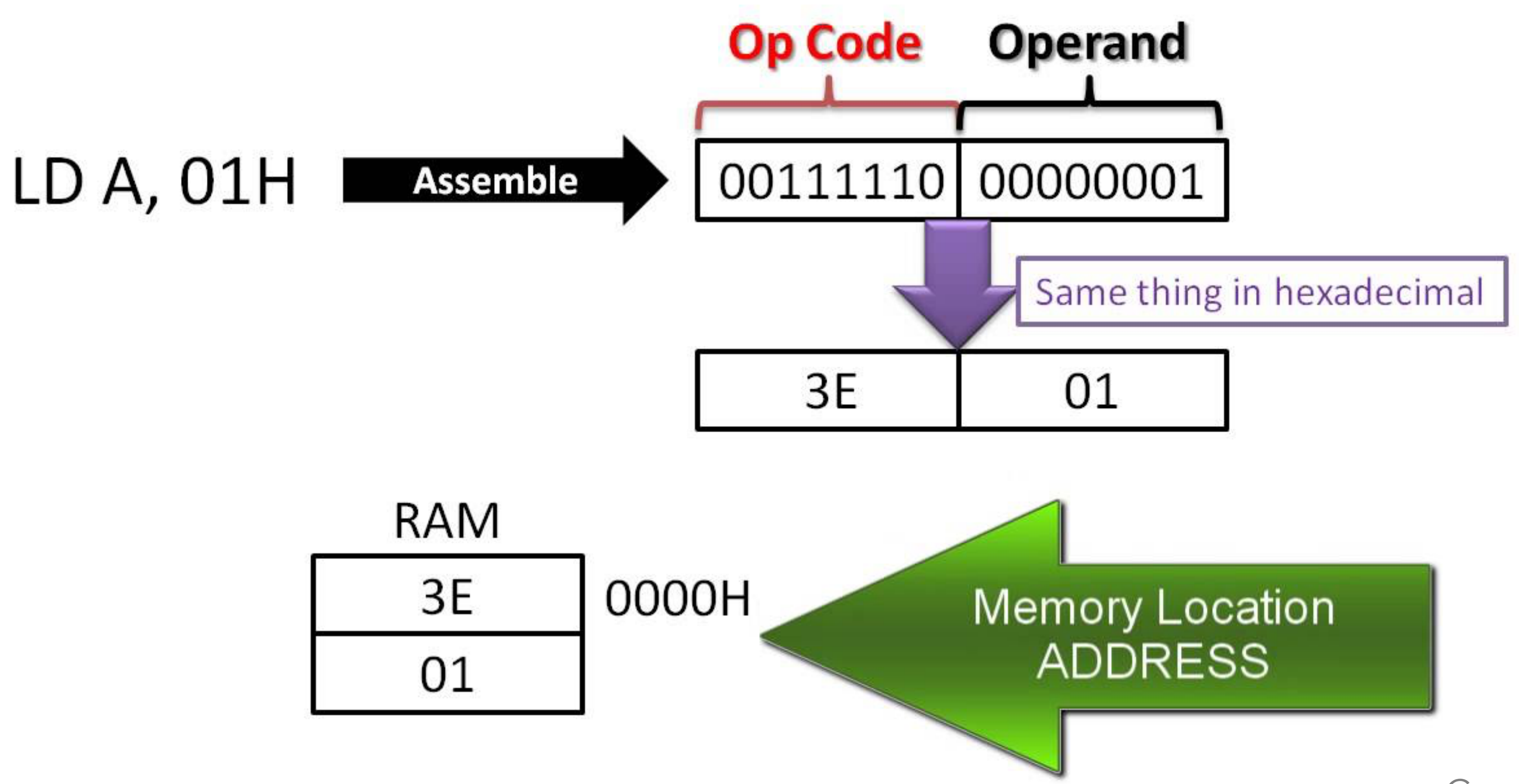
```
;------------------------------------------------------------
; zstr_count:
; Counts a zero-terminated ASCII string to determine its size
; in:   eax = start address of the zero terminated string
; out:  ecx = count = the length of the string

zstr_count:                   ; Entry point
    mov  ecx, -1              ; Init the loop counter, pre-decrement
                              ;  to compensate for the increment
.loop:
    inc  ecx                  ; Add 1 to the loop counter
    cmp  byte [eax + ecx], 0  ; Compare the value at the string's
                              ;  [starting memory address Plus the
                              ;  loop offset], to zero
    jne  .loop                ; If the memory value is not zero,
                              ;  then jump to the label called '.loop',
                              ;  otherwise continue to the next line
.done:
                              ; We don't do a final increment,
                              ;  because even though the count is base 1,
                              ;  we do not include the zero terminator in the
                              ;  string's length
    ret                       ; Return to the calling program
```

```
00000030  B9FFFFFFFF


00000035  41
00000036  803C0800


0000003A  75F9







0000003C  C3
```

OP Code Size

OP codes might not all occupy the same amount of memory! (it does for ARM but not for X86)

- varying number of data arguments
- compactness of addressing modes



Assembling Source Code

… recognizes **data directives** and **labels** …

- allocates and populates space appropriately
- populates *symbol table* with label names & locations

```
;-----------------------------------------------------------
; zstr_count:
; Counts a zero-terminated ASCII string to determine its size
; in:   eax = start address of the zero terminated string
; out:  ecx = count = the length of the string

zstr_count:                   ; Entry point
    mov  ecx, -1              ; Init the loop counter, pre-decrement
                              ;  to compensate for the increment
.loop:
    inc  ecx                  ; Add 1 to the loop counter
    cmp  byte [eax + ecx], 0  ; Compare the value at the string's
                              ;  [starting memory address Plus the
                              ;  loop offset], to zero
    jne  .loop                ; If the memory value is not zero,
                              ;  then jump to the label called '.loop',
                              ;  otherwise continue to the next line
.done:
                              ; We don't do a final increment,
                              ;  because even though the count is base 1,
                              ;  we do not include the zero terminator in the
                              ;  string's length
    ret                       ; Return to the calling program
```

→

```
00000030 B9FFFFFFFF
00000035 41
00000036 803C0800
0000003A 75F9
0000003C C3
```

Assembling Source Code

… searches and replaces symbolic entries with their associated values from the symbol table

The assembler outputs an **object program** to file

```
;------------------------------------------------------------
; zstr_count:
; Counts a zero-terminated ASCII string to determine its size
; in:   eax = start address of the zero terminated string
; out:  ecx = count = the length of the string

zstr_count:                       ; Entry point
    mov  ecx, -1                  ; Init the loop counter, pre-decrement
                                  ;  to compensate for the increment
.loop:
    inc  ecx                      ; Add 1 to the loop counter
    cmp  byte [eax + ecx], 0      ; Compare the value at the string's
                                  ;  [starting memory address Plus the
                                  ;  loop offset], to zero
    jne  .loop                    ; If the memory value is not zero,
                                  ;  then jump to the label called '.loop',
                                  ;  otherwise continue to the next line
.done:
                                  ; We don't do a final increment,
                                  ;  because even though the count is base 1,
                                  ;  we do not include the zero terminator in the
                                  ;  string's length
    ret                           ; Return to the calling program
```

in.asm

[*plain text*]

```
00000030 B9FFFFFFFF

00000035 41
00000036 803C0800

0000003A 75F9

0000003C C3
```

out.obj

[*binary*]

We say that an assembler *assembles* to object code

Early Assemblers: Pen, Paper & Books

Today, assemblers are programs that we execute on computers

- using computers to program computers

In the past, humans had to *manually* assemble their own code

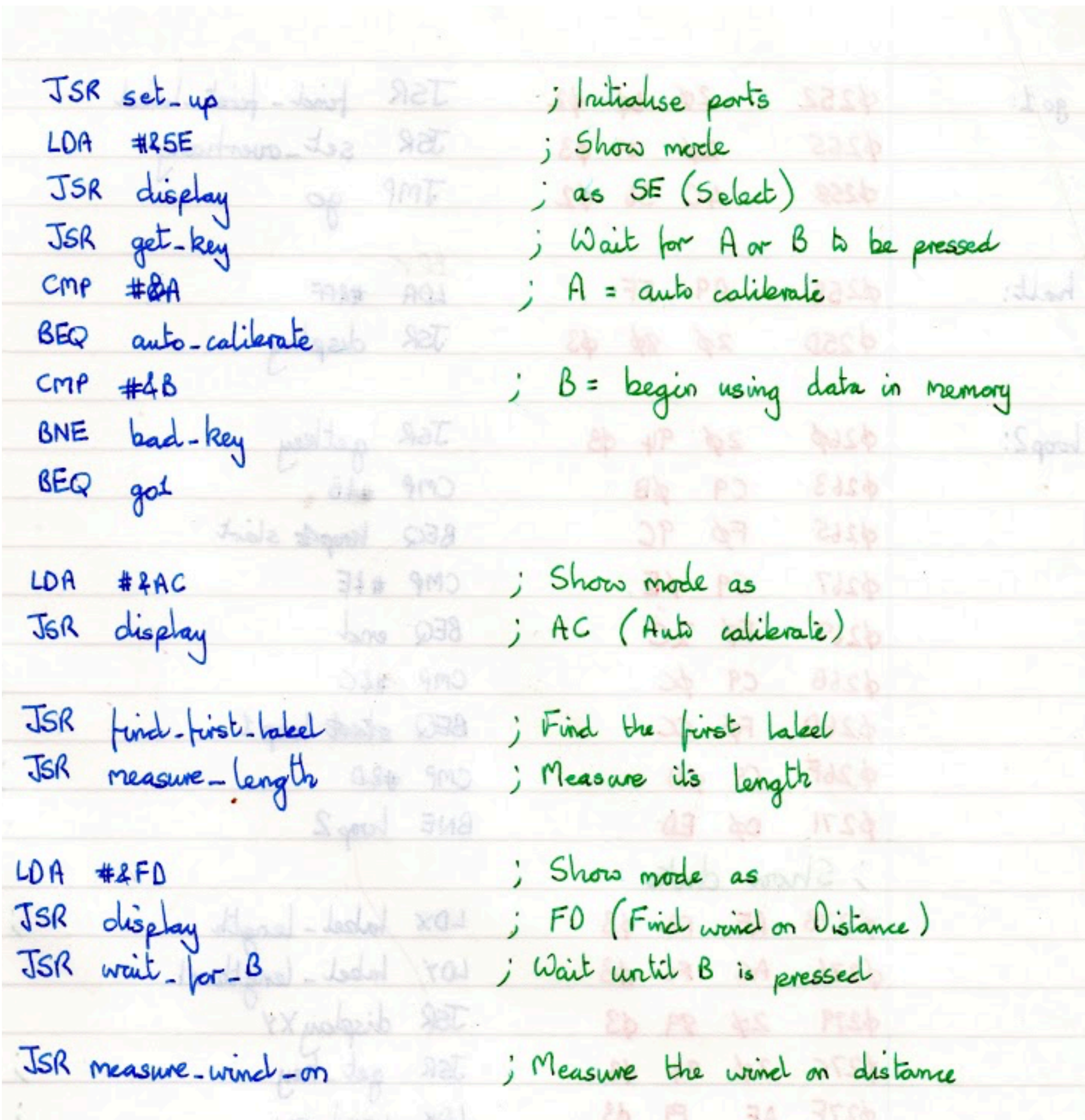
- working through this process can be helpful

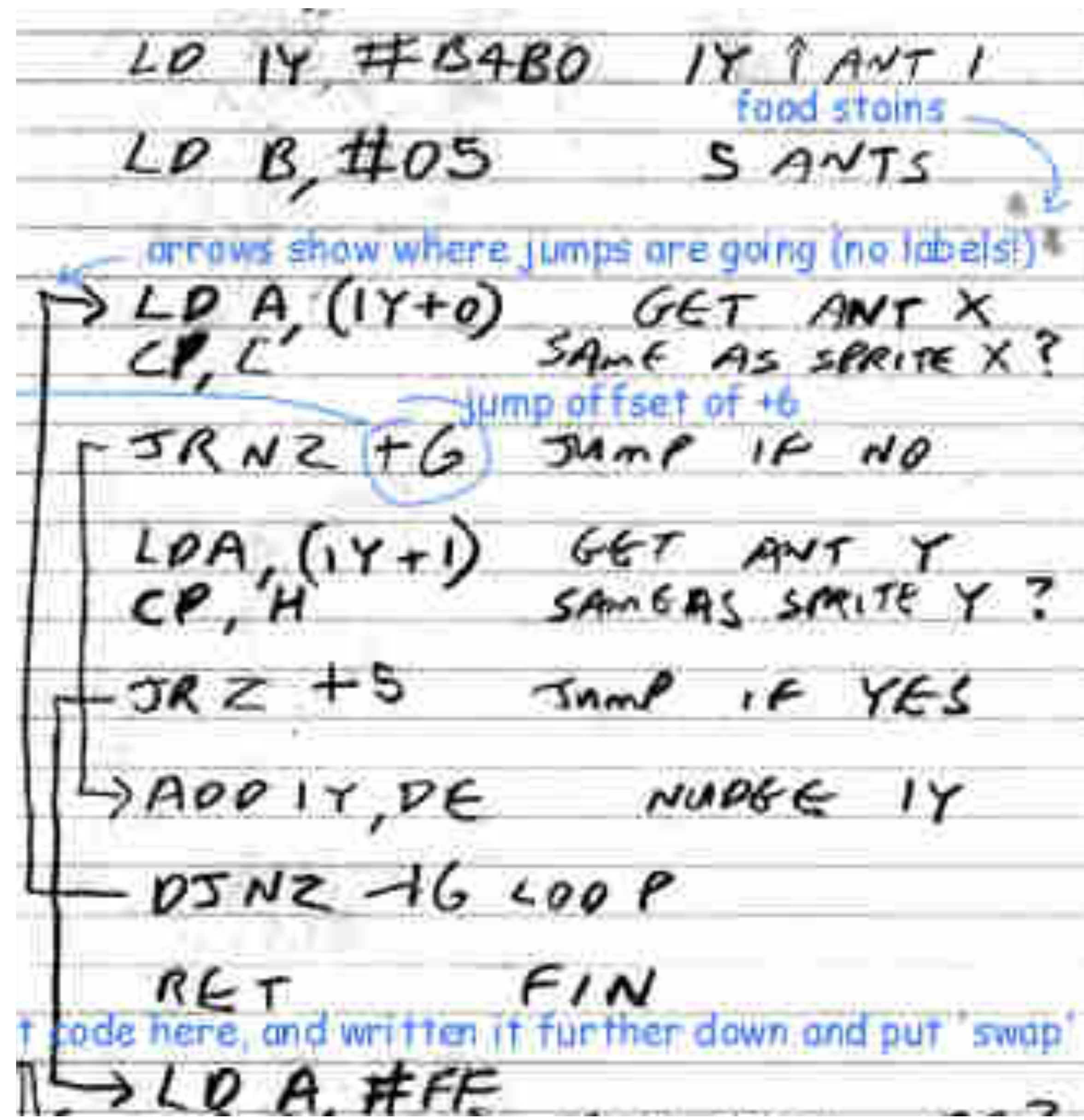
Early Assemblers: Pen, Paper & Books

Type your assembly code in a text editor

Early Assemblers: Pen, Paper & Books

~~Type~~ Write your assembly code ~~in a text editor~~ on paper





Early Assemblers: Pen, Paper & Books

Sequentially replace assembler mnemonics (and data/addressing operands) with their binary machine OP codes

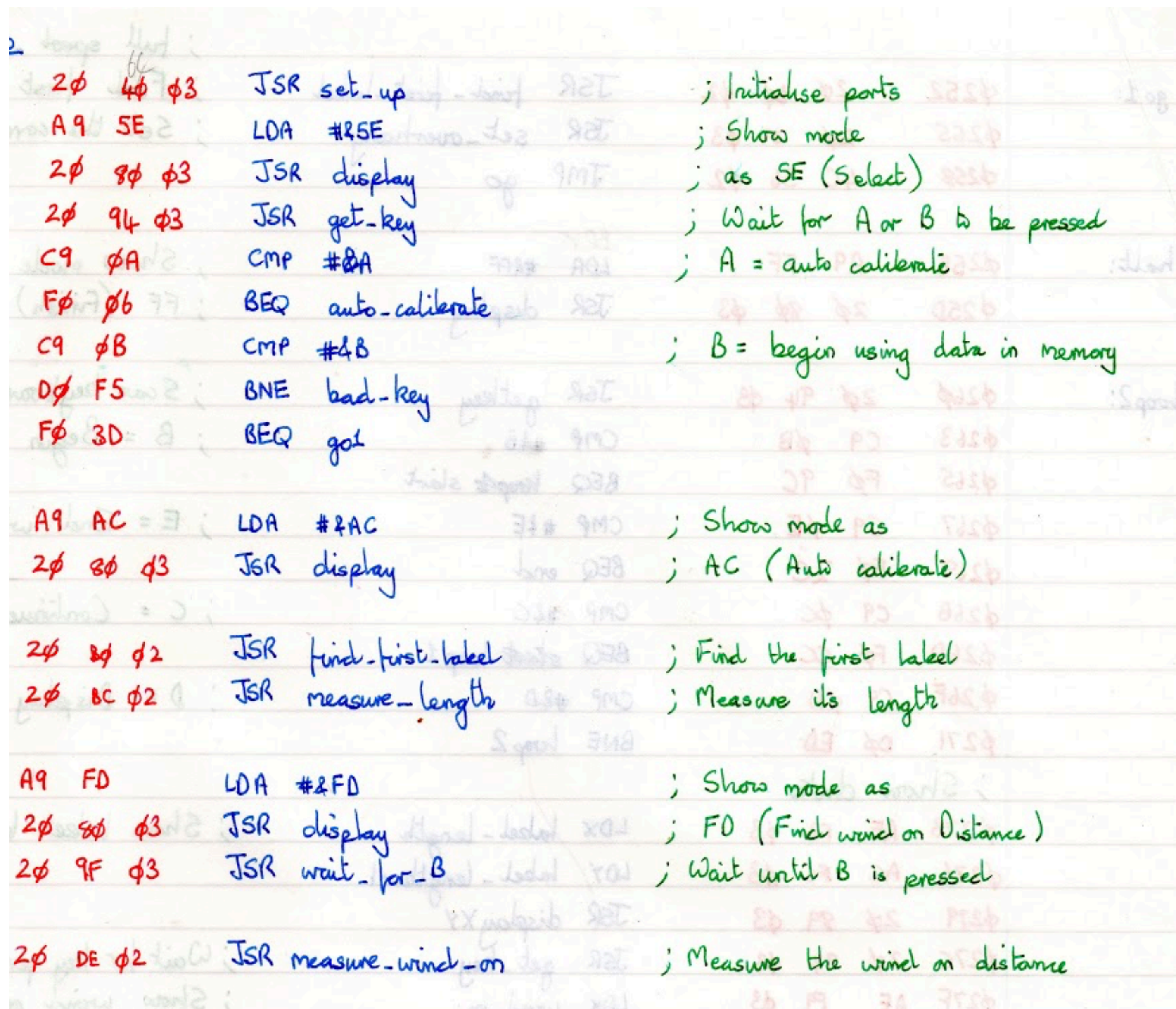
- How? **R**ead **T**he **M**anual…

ADDLW **Add Literal and W**

Syntax:	[*label*] ADDLW k
Operands:	$0 \leq k \leq 255$
Operation:	$(W) + k \rightarrow W$
Status Affected:	C, DC, Z
Encoding:	11 | 111x | kkkk | kkkk
Description:	The contents of the W register are added to the eight bit literal 'k' and the result is placed in the W register.

Early Assemblers: Pen, Paper & Books

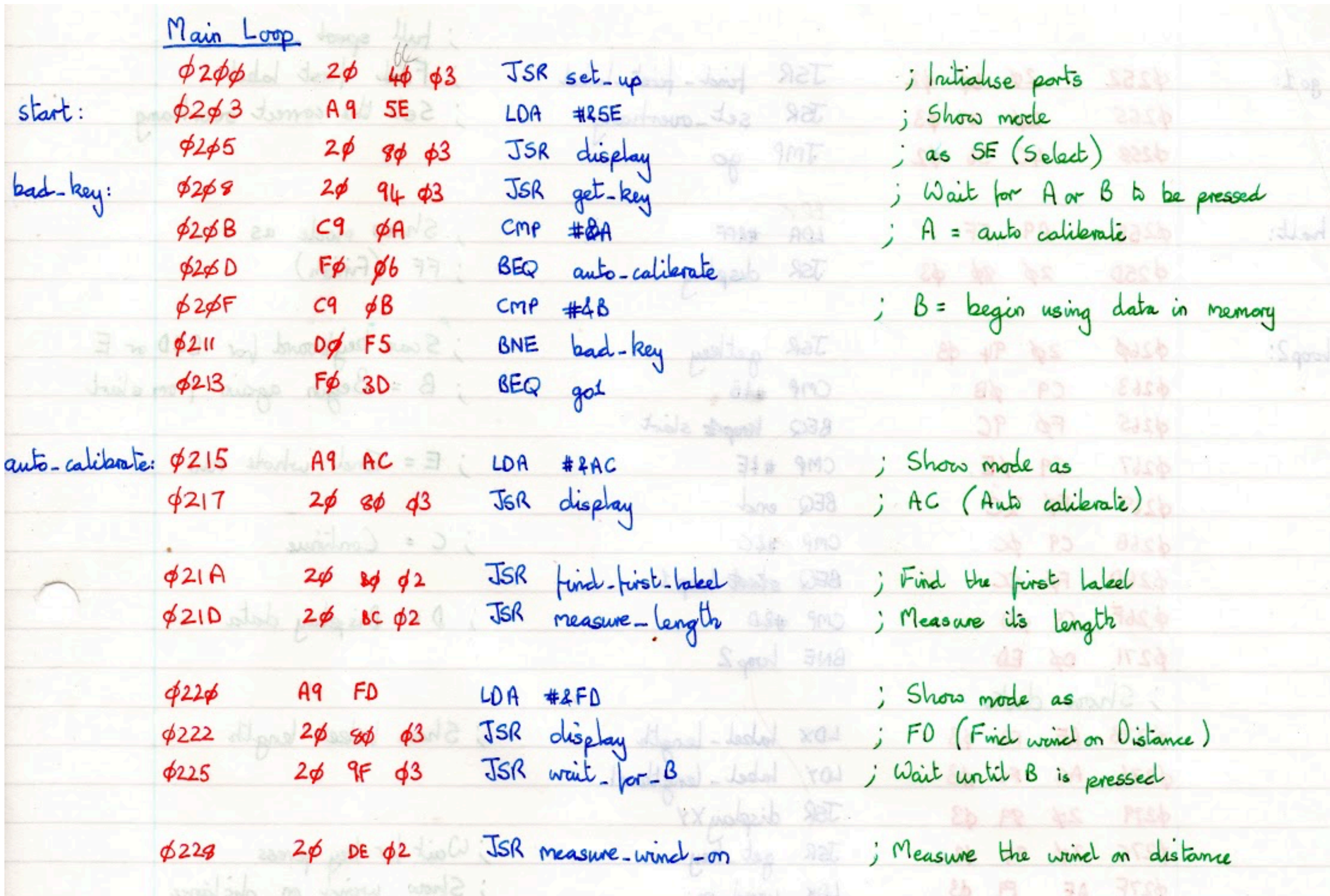
Sequentially replace assembler mnemonics (and data/addressing operands) with their binary machine OP codes



```
20 40 03    JSR set_up             ; Initialise ports
A9 5E       LDA #&5E               ; Show mode
20 80 03    JSR display            ; as SE (Select)
20 94 03    JSR get_key            ; Wait for A or B to be pressed
C9 0A       CMP #&0A               ; A = auto calibrate
F0 06       BEQ auto_calibrate
C9 0B       CMP #&0B               ; B = begin using data in memory
D0 F5       BNE bad_key
F0 3D       BEQ go1

A9 AC       LDA #&AC               ; Show mode as
20 80 03    JSR display            ; AC (Auto calibrate)

20 80 02    JSR find_first_label   ; Find the first label
20 BC 02    JSR measure_length     ; Measure its length

A9 FD       LDA #&FD               ; Show mode as
20 80 03    JSR display            ; FD (Find wind on Distance)
20 9F 03    JSR wait_for_B         ; Wait until B is pressed

20 DE 02    JSR measure_wind_on    ; Measure the wind on distance
```

Early Assemblers: Pen, Paper & Books

Perform (manual) relative memory layout



Two-pass Assemblers

An important question arises during assembly, when substituting values from the symbol table:

- what happens if we encounter a label/name without an existing symbol table entry (a **forward reference**)?

```
.loop1:

    inc ecx
    cmp byte [eax + ecx]

    jne .loop2          ; we don't yet have a value for the label loop2

.loop2:

    ret
```

- what's the problem here? how would you solve it?

Two-pass Assemblers

Two-pass assemblers solve this problem by:

1. making an initial pass: converting mnemonics and building the symbol table **when you can**

```
.loop1:

    inc ecx
    cmp byte [eax + ecx]

    jne .loop2

.loop2:

    ret
```

1. making an initial pass: converting mnemonics and building the symbol table **when you can**

Object Program Memory Map

Memory Address	OP Code/Data
0x00	
0x01	
0x02	
0x03	

Symbol Table

Symbol Name	Symbol Value

```
.loop1:

    inc ecx
    cmp byte [eax + ecx]

    jne .loop2

.loop2:

    ret
```

1. making an initial pass: converting mnemonics and building the symbol table **when you can**

Object Program Memory Map

Memory Address	OP Code/Data
0x00	
0x01	
0x02	
0x03	

```
.loop1:

    inc ecx
    cmp byte [eax + ecx]

    jne .loop2

.loop2:

    ret
```

Symbol Table

Symbol Name	Symbol Value
loop1	

1. making an initial pass: converting mnemonics and building the symbol table **when you can**

Object Program Memory Map

Memory Address	OP Code/Data
0x00	
0x01	
0x02	
0x03	

```
.loop1:

    inc ecx
    cmp byte [eax + ecx]

    jne .loop2

.loop2:

    ret
```

Symbol Table

Symbol Name	Symbol Value
loop1	0x00

1. making an initial pass: converting mnemonics and building the symbol table **when you can**

Object Program Memory Map

Memory Address	OP Code/Data
0x00	
0x01	
0x02	
0x03	

Symbol Table

Symbol Name	Symbol Value
loop1	0x00

```
.loop1:

    inc ecx
    cmp byte [eax + ecx]

    jne .loop2

.loop2:

    ret
```

assume/given:

inc is a 1-byte instruction:

- 4-bit OP code
- 4-bit operand code

1. making an initial pass: converting mnemonics and building the symbol table **when you can**

Object Program Memory Map

Memory Address	OP Code/Data
0x00	3A
0x01	
0x02	
0x03	

Symbol Table

Symbol Name	Symbol Value
loop1	0x00

```
.loop1:

    inc ecx
    cmp byte [eax + ecx]

    jne .loop2

.loop2:

    ret
```

assume/given:

inc is a 1-byte instruction:

- 4-bit OP code
- 4-bit operand code

1. making an initial pass: converting mnemonics and building the symbol table **when you can**

Object Program Memory Map

Memory Address	OP Code/Data
0x00	3A
0x01	
0x02	
0x03	

Symbol Table

Symbol Name	Symbol Value
loop1	0x00

```
.loop1:

    inc ecx
    cmp byte [eax + ecx]

    jne .loop2

.loop2:

    ret
```

assume/given:

cmp is a 1-byte instruction

1. making an initial pass: converting mnemonics and building the symbol table **when you can**

Object Program Memory Map

Memory Address	OP Code/Data
0x00	3A
0x01	7F
0x02	
0x03	

```
.loop1:

    inc ecx
    cmp byte [eax + ecx]

    jne .loop2

.loop2:

    ret
```

Symbol Table

Symbol Name	Symbol Value
loop1	0x00

assume/given:

cmp is a 1-byte instruction

1. making an initial pass: converting mnemonics and building the symbol table **when you can**

Object Program Memory Map

Memory Address	OP Code/Data
0x00	3A
0x01	7F
0x02	
0x03	

```
.loop1:

    inc ecx
    cmp byte [eax + ecx]

    jne .loop2

.loop2:

    ret
```

Symbol Table

Symbol Name	Symbol Value
loop1	0x00

assume/given:

jne is a 1-byte instruction:

- 4-bit OP code (F)
- 4-bit operand

1. making an initial pass: converting mnemonics and building the symbol table **when you can**

Object Program Memory Map

Memory Address	OP Code/Data
0x00	3A
0x01	7F
0x02	F?
0x03	

```
.loop1:

    inc ecx
    cmp byte [eax + ecx]

    jne .loop2

.loop2:

    ret
```

Symbol Table

Symbol Name	Symbol Value
loop1	0x00

assume/given:

jne is a 1-byte instruction:

- 4-bit OP code (F)
- 4-bit operand

1. making an initial pass: converting mnemonics and building the symbol table **when you can**

Object Program Memory Map

Memory Address	OP Code/Data
0x00	3A
0x01	7F
0x02	F?
0x03	

Symbol Table

Symbol Name	Symbol Value
loop1	0x00

```
.loop1:

    inc ecx
    cmp byte [eax + ecx]

    jne .loop2

.loop2:

    ret
```

assume/given:

jne is a 1-byte instruction:

- 4-bit OP code (F)
- 4-bit operand

1. making an initial pass: converting mnemonics and building the symbol table **when you can**

Object Program Memory Map

Memory Address	OP Code/Data
0x00	3A
0x01	7F
0x02	F?
0x03	

Symbol Table

Symbol Name	Symbol Value
loop1	0x00
loop2	

```
.loop1:

    inc ecx
    cmp byte [eax + ecx]

    jne .loop2

.loop2:

    ret
```

assume/given:

jne is a 1-byte instruction:

- 4-bit OP code (F)
- 4-bit operand

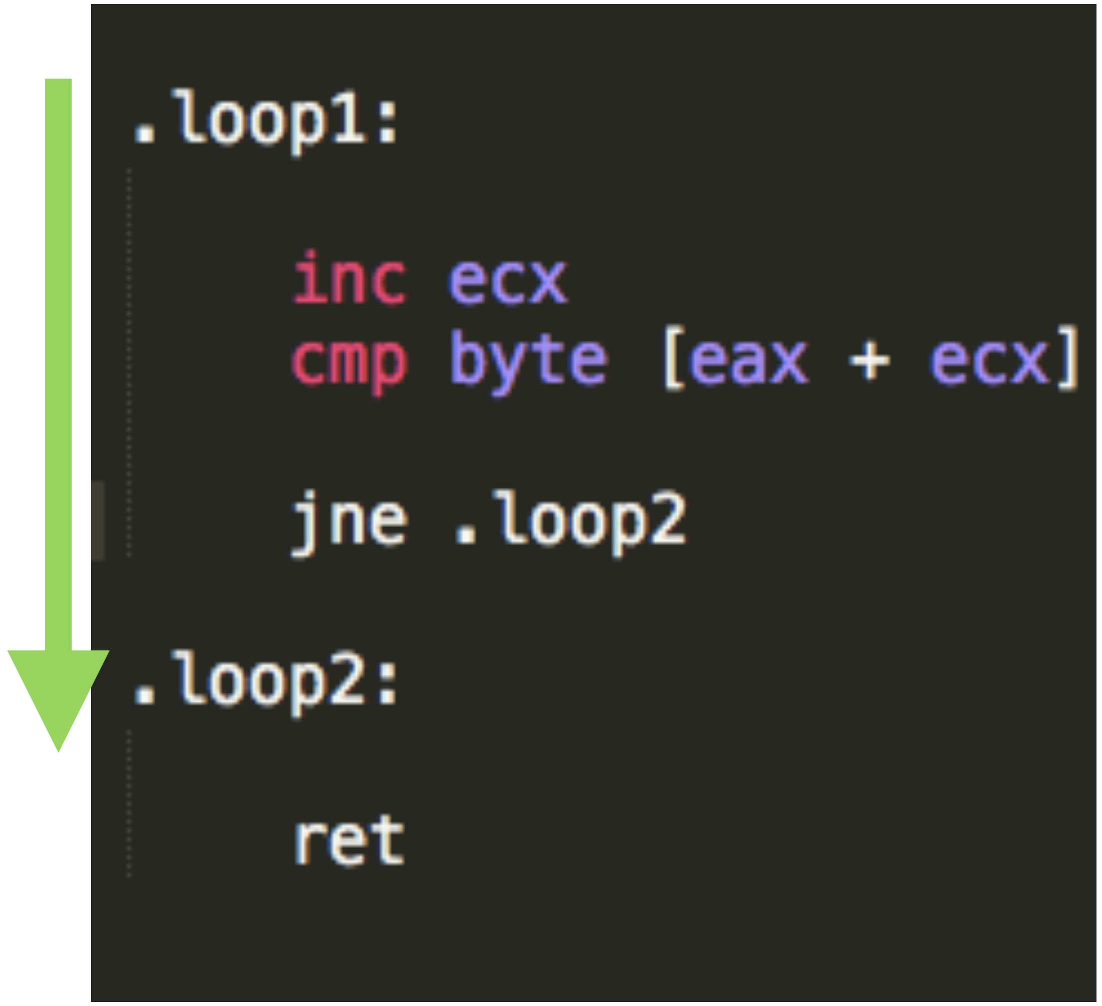
1. making an initial pass: converting mnemonics and building the symbol table **when you can**

Object Program Memory Map

Memory Address	OP Code/Data
0x00	3A
0x01	7F
0x02	F?
0x03	

Symbol Table

Symbol Name	Symbol Value
loop1	0x00
loop2	0x03



assume/given:

jne is a 1-byte instruction:

- 4-bit OP code (F)
- 4-bit operand

1. making an initial pass: converting mnemonics and building the symbol table **when you can**

Object Program Memory Map

Memory Address	OP Code/Data
0x00	3A
0x01	7F
0x02	F?
0x03	

Symbol Table

Symbol Name	Symbol Value
loop1	0x00
loop2	0x03

```
.loop1:

    inc ecx
    cmp byte [eax + ecx]

    jne .loop2

.loop2:

    ret
```

assume/given:

ret is a 1-byte instruction:

- 8-bit OP code

1. making an initial pass: converting mnemonics and building the symbol table **when you can**

Object Program Memory Map

Memory Address	OP Code/Data
0x00	3A
0x01	7F
0x02	F?
0x03	DD

Symbol Table

Symbol Name	Symbol Value
loop1	0x00
loop2	0x03

```
.loop1:

    inc ecx
    cmp byte [eax + ecx]

    jne .loop2

.loop2:

    ret
```

assume/given:

ret is a 1-byte instruction:

- 8-bit OP code

Two-pass Assemblers

Two-pass assemblers solve this problem by:

1. making an initial pass: converting mnemonics and building the symbol table **when you can**

2. make a final pass filling in missing references

```
.loop1:

    inc ecx
    cmp byte [eax + ecx]

    jne .loop2

.loop2:

    ret
```

2. make a final pass filling in missing references

Object Program Memory Map

Memory Address	OP Code/Data
0x00	3A
0x01	7F
0x02	F?
0x03	DD

Symbol Table

Symbol Name	Symbol Value
loop1	0x00
loop2	0x03

```
.loop1:

    inc ecx
    cmp byte [eax + ecx]

    jne .loop2

.loop2:

    ret
```

Missing reference?

- No

2. make a final pass filling in missing references

Object Program Memory Map

Memory Address	OP Code/Data
0x00	3A
0x01	7F
0x02	F?
0x03	DD

Symbol Table

Symbol Name	Symbol Value
loop1	0x00
loop2	0x03

```
.loop1:

    inc ecx
    cmp byte [eax + ecx]

    jne .loop2

.loop2:

    ret
```

Missing reference?

- No

2. make a final pass filling in missing references

Object Program Memory Map

Memory Address	OP Code/Data
0x00	3A
0x01	7F
0x02	F?
0x03	DD

```
.loop1:

    inc ecx
    cmp byte [eax + ecx]

    jne .loop2

.loop2:

    ret
```

Symbol Table

Symbol Name	Symbol Value
loop1	0x00
loop2	0x03

Missing reference?

- Yes!
- Find & replace value from symbol table

2. make a final pass filling in missing references

Object Program Memory Map

Memory Address	OP Code/Data
0x00	3A
0x01	7F
0x02	F3
0x03	DD

Symbol Table

Symbol Name	Symbol Value
loop1	0x00
loop2	0x03

```
.loop1:

    inc ecx
    cmp byte [eax + ecx]

    jne .loop2

.loop2:

    ret
```

Missing reference?

- Yes!
- Find & replace value from symbol table

2. make a final pass filling in missing references

Object Program Memory Map

Memory Address	OP Code/Data
0x00	3A
0x01	7F
0x02	F3
0x03	DD

Symbol Table

Symbol Name	Symbol Value
loop1	0x00
loop2	0x03

```
.loop1:

    inc ecx
    cmp byte [eax + ecx]

    jne .loop2

.loop2:

    ret
```

Missing reference?

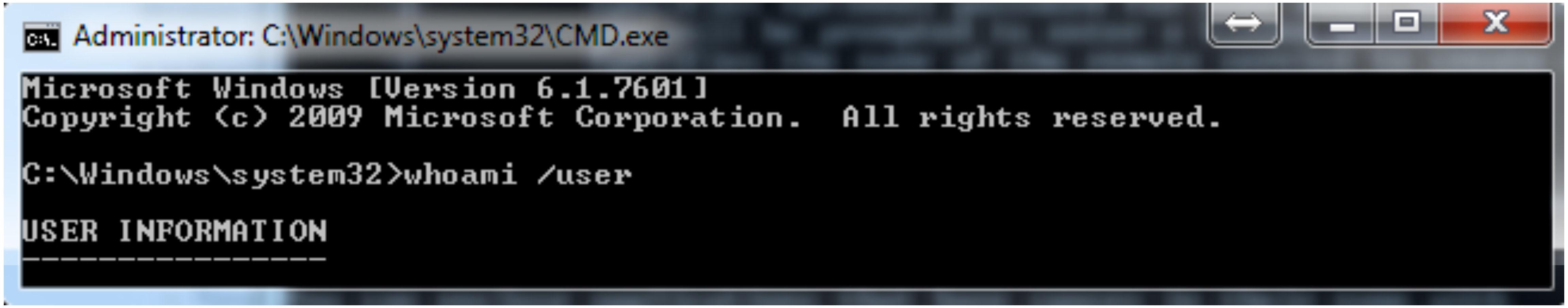
- No

Loading & Executing Object Programs

Once an object program binary memory layout is generated, we can execute our assembled program

How?

- by invoking a *loader program*



Loader Program

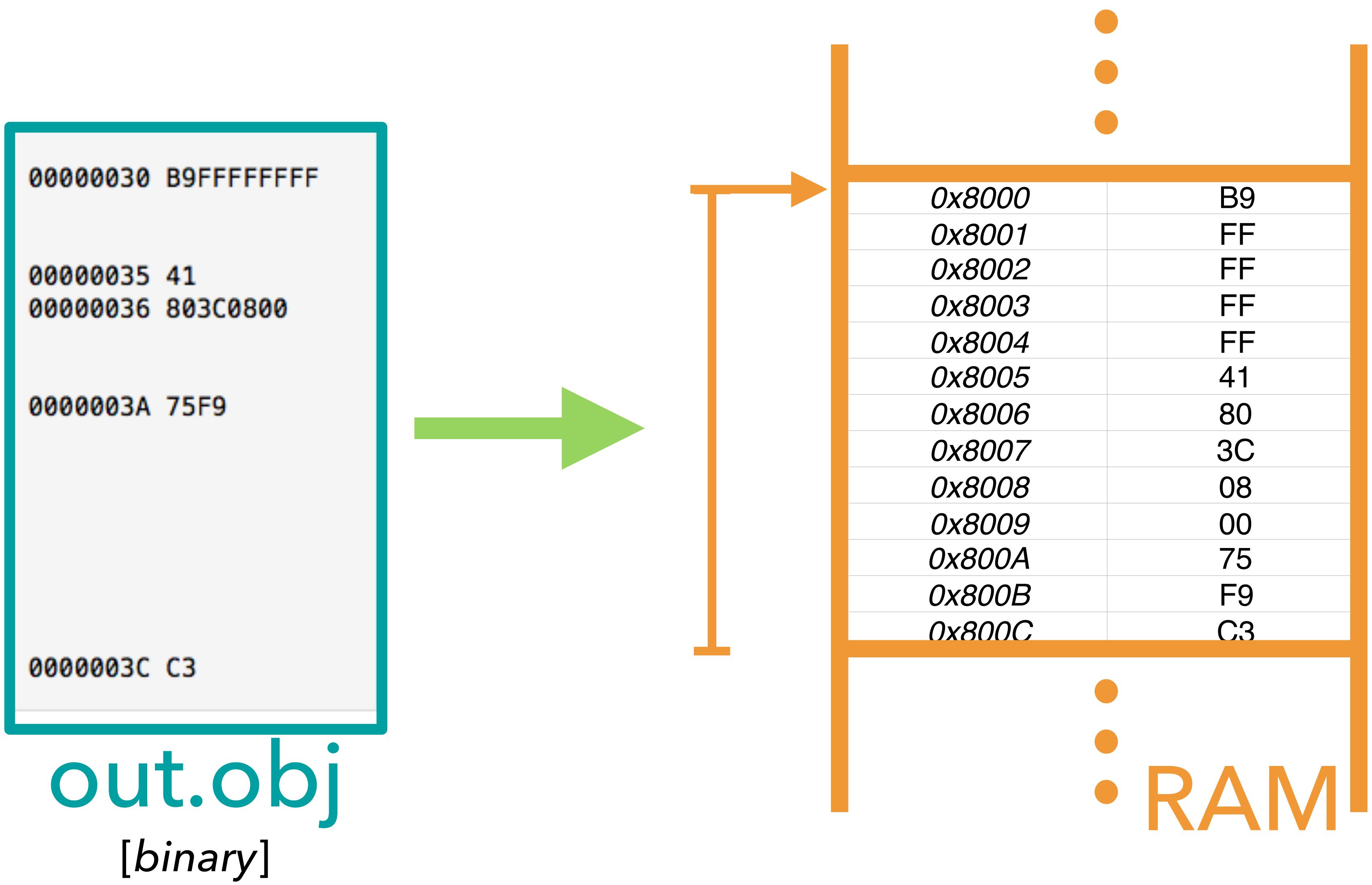
The loader program has 3 responsibilities:

1. load object program's contents from file into memory
2. jump to starting address to execute program
3. recover memory after program execution

Loader Program

The loader program has 3 responsibilities:

1. load object program's contents from file into memory
   - user identifies file via, e.g., command-line/GUI/etc.
   - loader needs to know: start address & program length



Symbol Table

Symbol Name	Symbol Value
START	0x00
loop1	0x00
loop2	0x03

Loader Program

The loader program has 3 responsibilities:

2. jump to starting address
    - i.e., sets the program counter to the absolute start point
    - i.e., executes the first instruction of the object program

```
; loader program logic ...

; load object program
; file contents into memory

; parse START address
; from symbol table

; compute absolute address

jmp STARTabs
```

Program Counter
0x8000

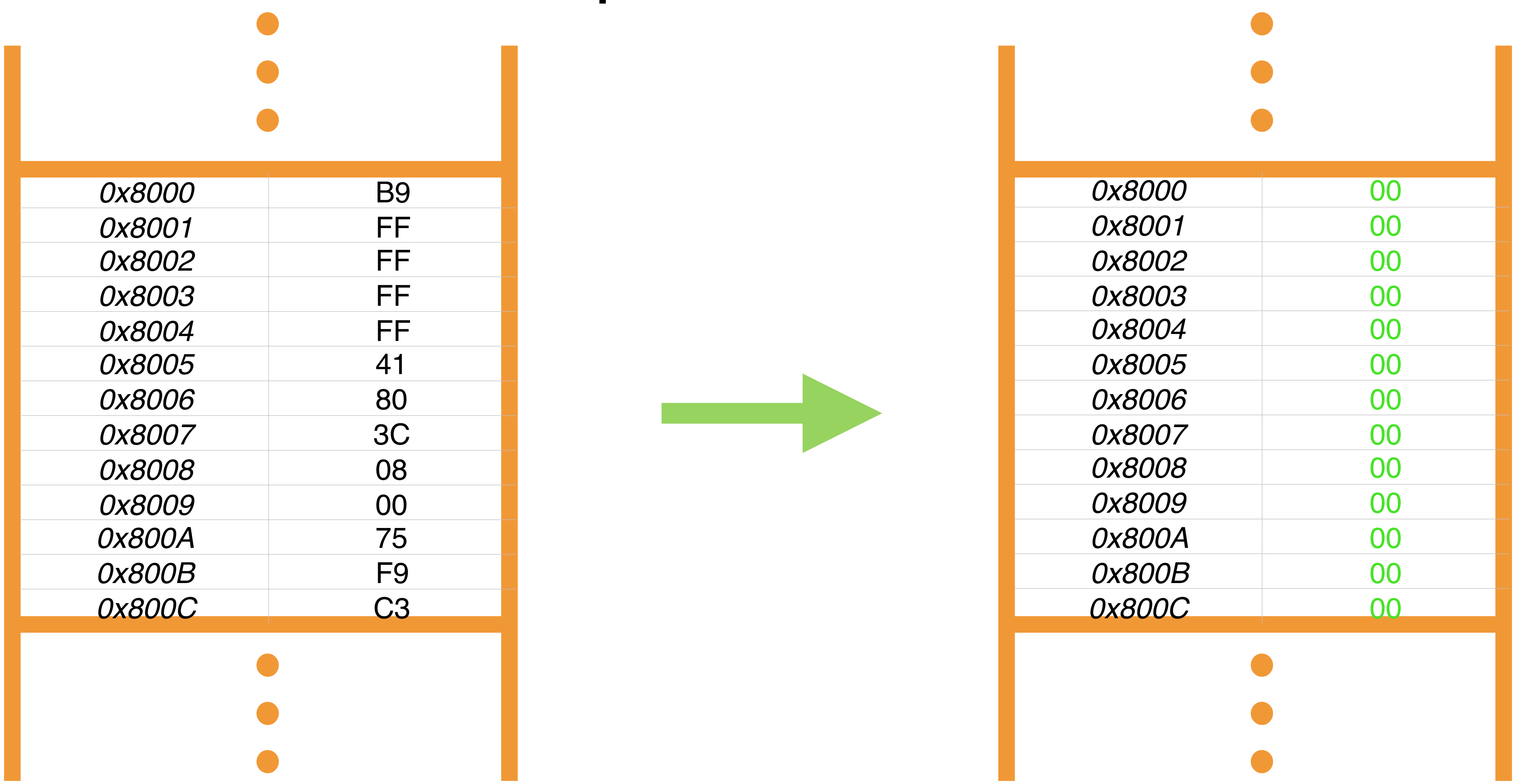
0x8000	B9
0x8001	FF
0x8002	FF
0x8003	FF
0x8004	FF
0x8005	41
0x8006	80
0x8007	3C
0x8008	08
0x8009	00
0x800A	75
0x800B	F9
0x800C	C3

Loader Program

The loader program has 3 responsibilities:

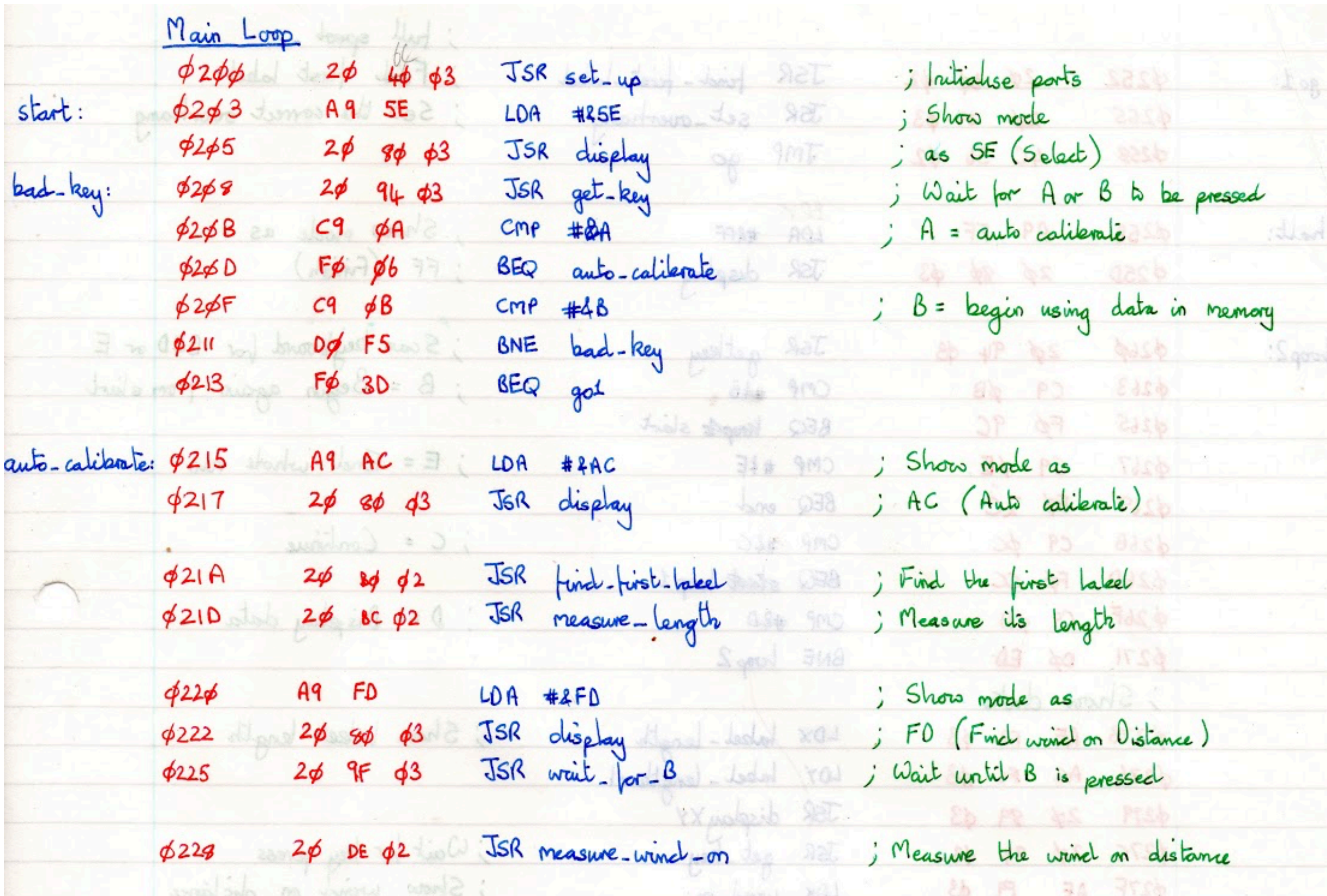
3. recover memory after program execution

- program termination follows a predefined protocol
- loader cleans up* and returns control to user



Early Assemblers: Pen, Paper & Books

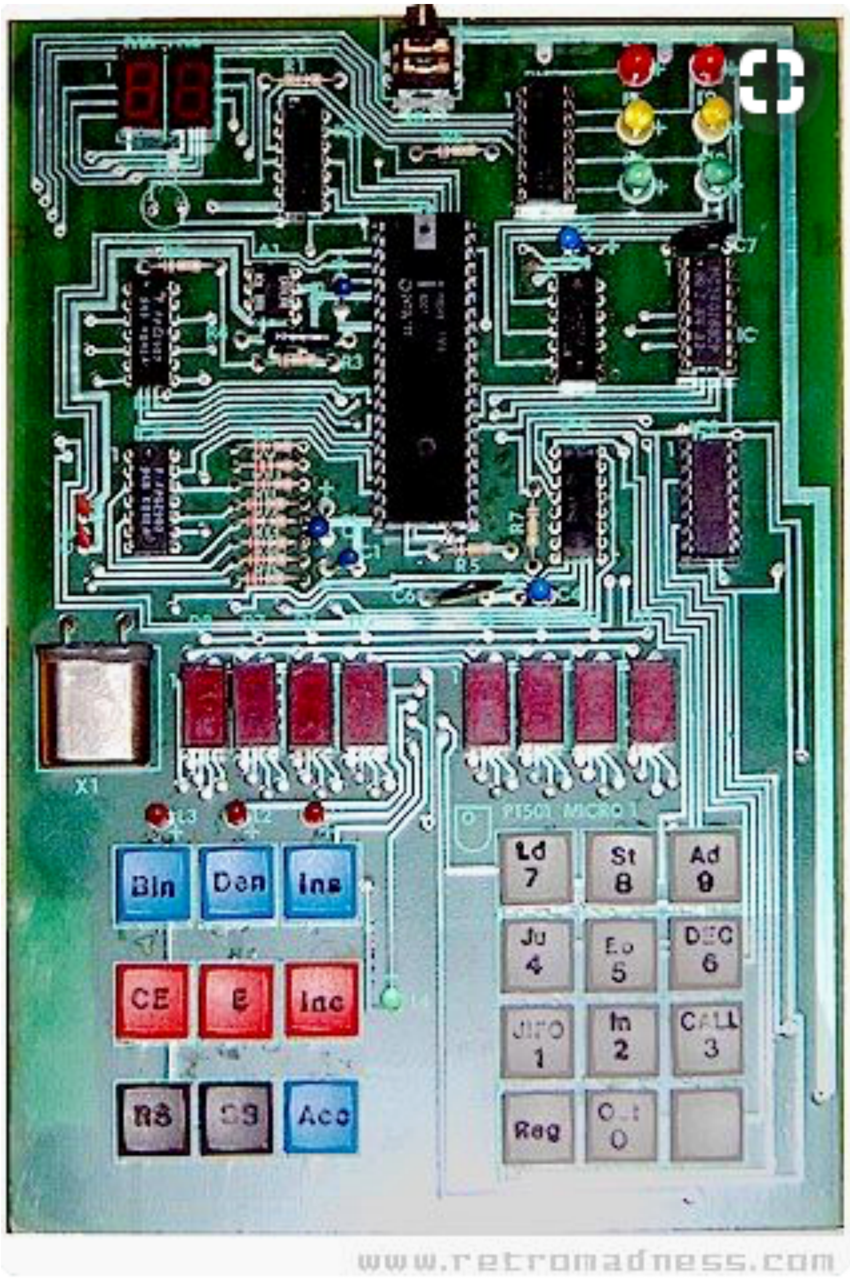
Convert assembly **to** binary **in** memory layout

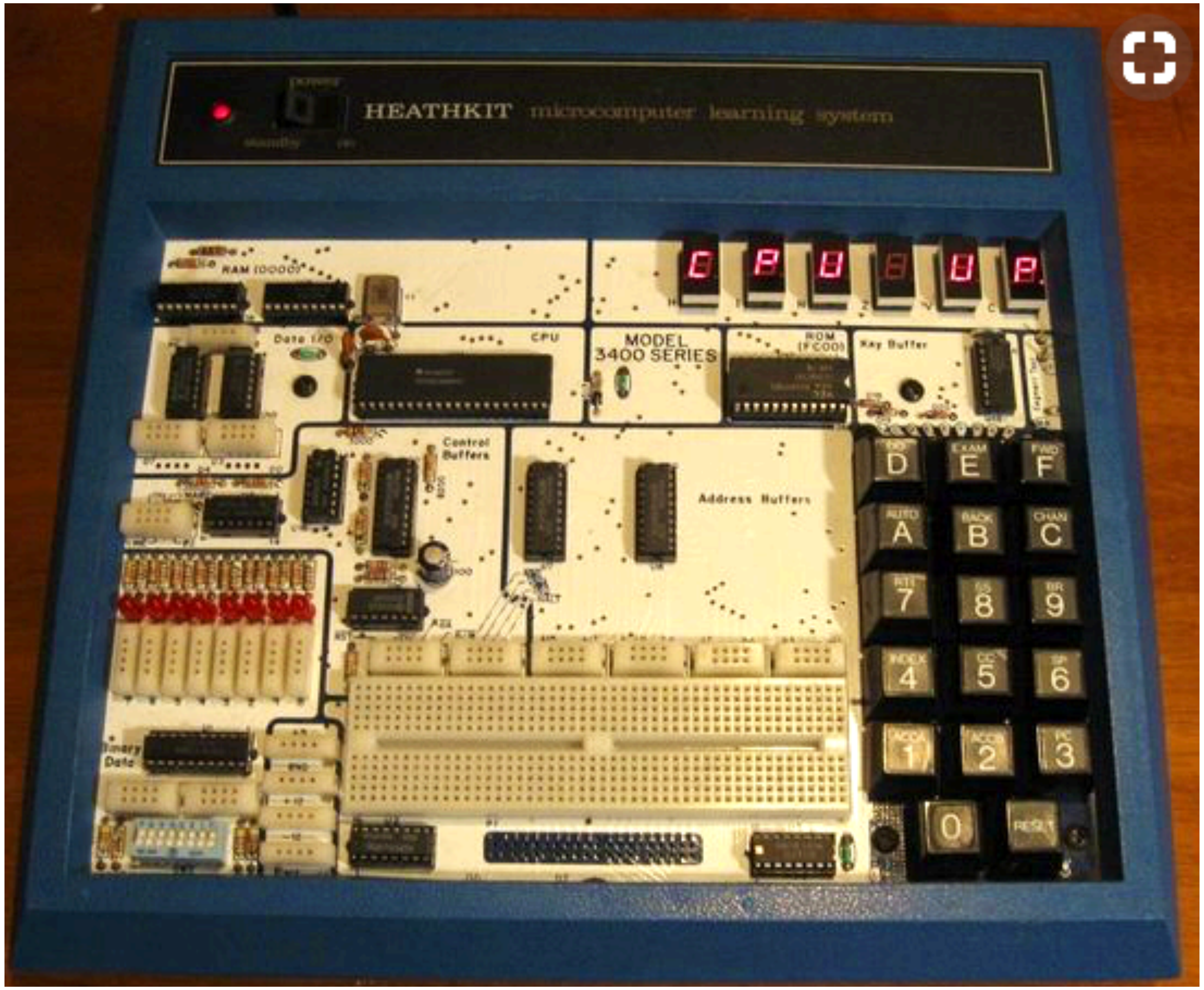


Early Loaders: Keypads & Fingers

Early loader "interfaces" were rudimentary

- many of these loaders weren't even implemented in software!

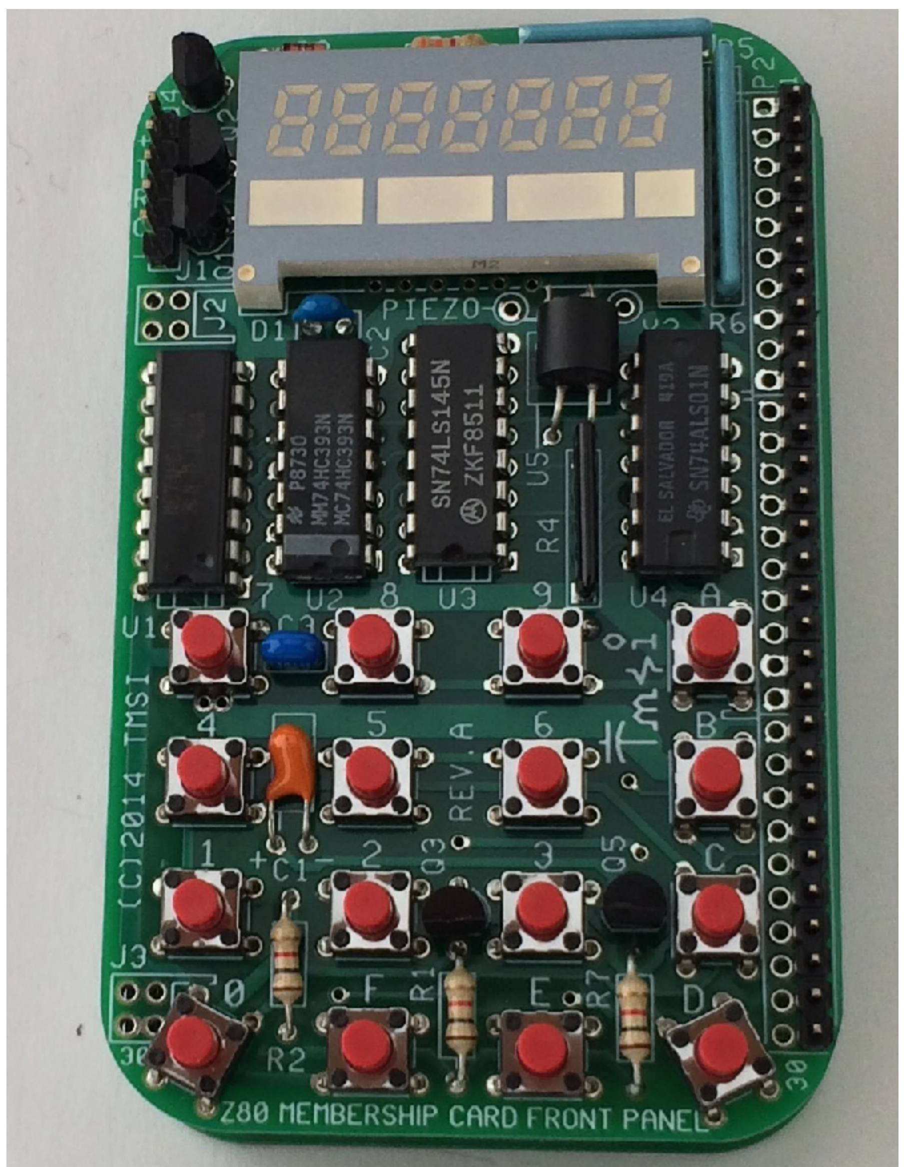


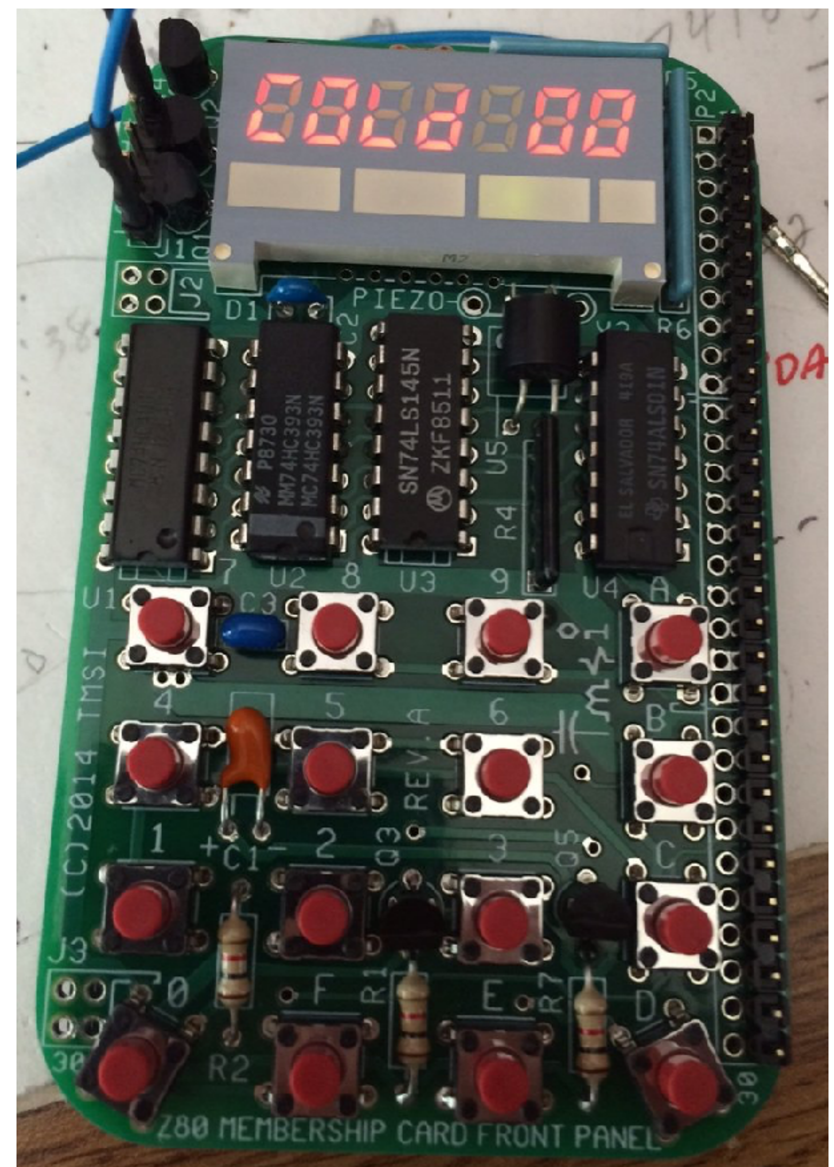


Early Loaders: Keypads & Fingers

Early loader "interfaces" were rudimentary

- many of these loaders weren't even implemented in software!

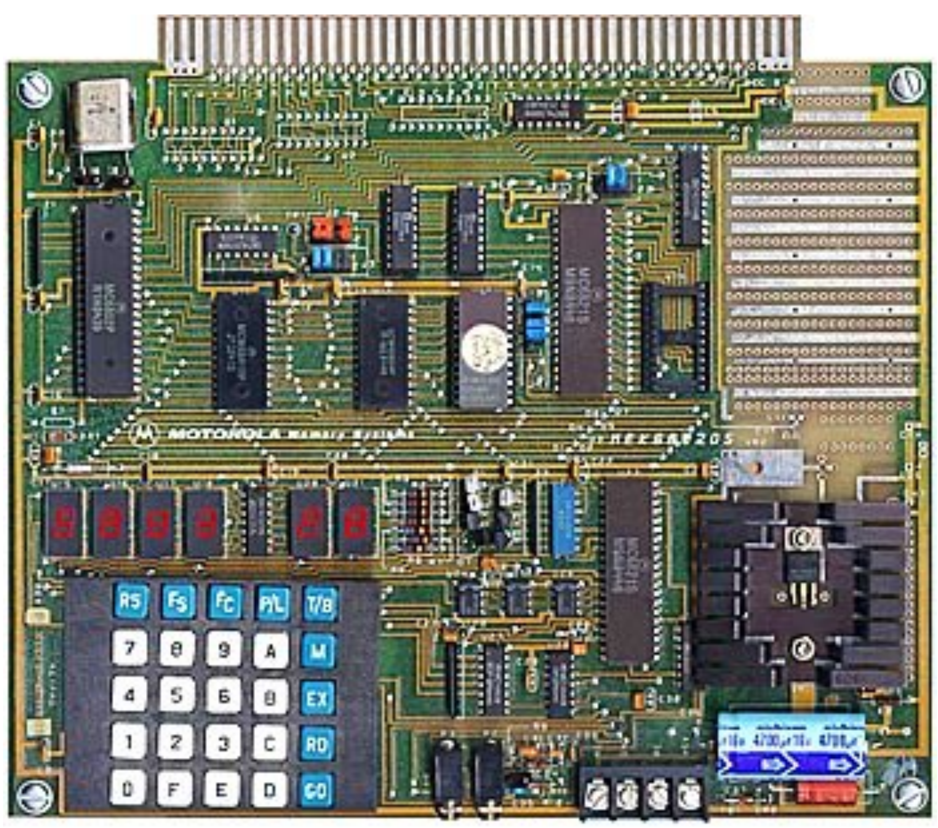


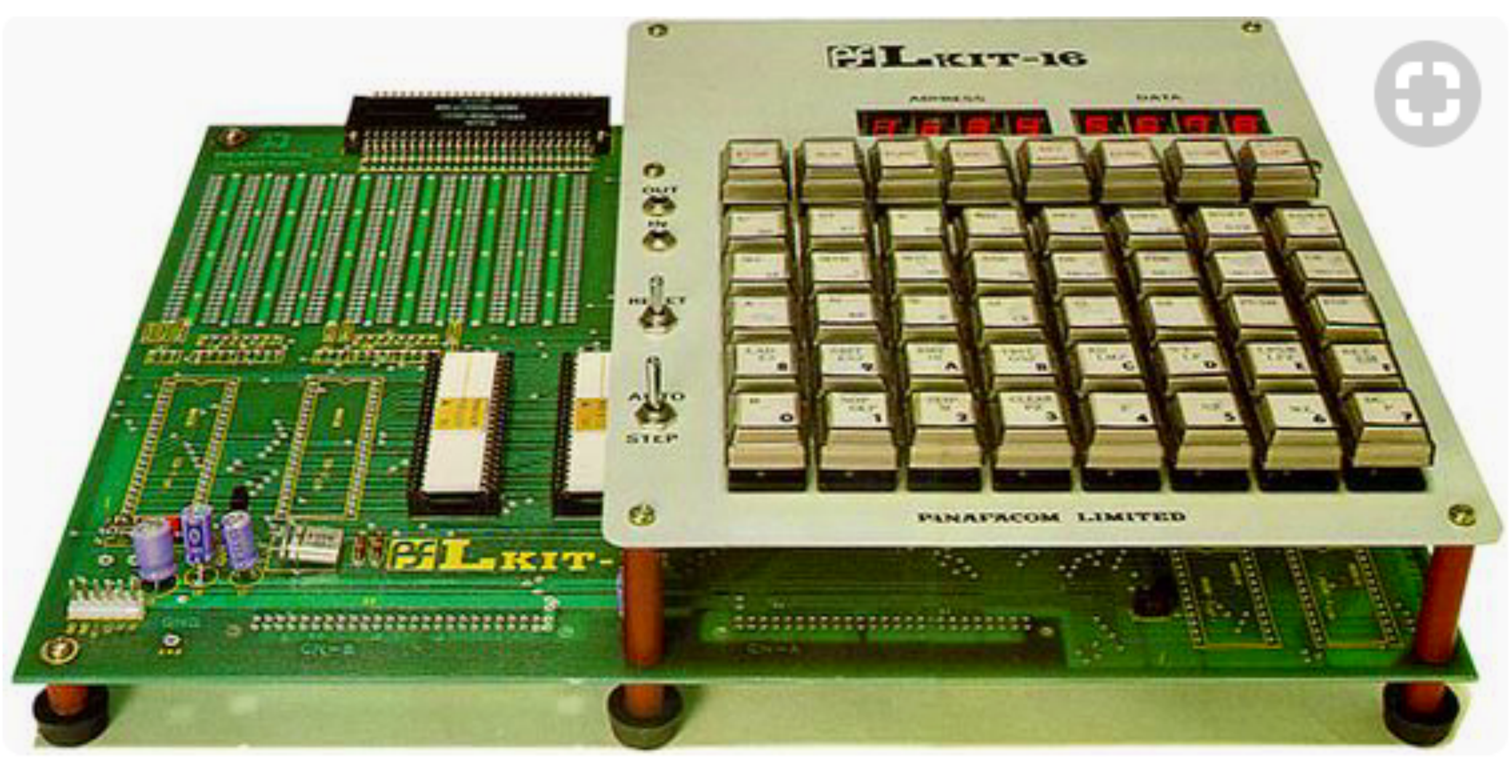


Early Loaders: Keypads & Fingers

Early loader "interfaces" were rudimentary

- many of these loaders weren't even implemented in software!
- user exposed to a simple, calculator-like keypad
  - entered address offsets manually
  - populated data* manually
  - set PC manually





Low- & high-level code interaction

Multi-source Object File Generation

So far, we assumed assemblers expected one source file & generated the object program file

```
;---------------------------------------------------------------
; zstr_count:
; Counts a zero-terminated ASCII string to determine its size
; in:   eax = start address of the zero terminated string
; out:  ecx = count = the length of the string

zstr_count:                   ; Entry point
    mov  ecx, -1              ; Init the loop counter, pre-decrement
                              ;  to compensate for the increment
.loop:
    inc  ecx                  ; Add 1 to the loop counter
    cmp  byte [eax + ecx], 0  ; Compare the value at the string's
                              ;  [starting memory address Plus the
                              ;  loop offset], to zero
    jne  .loop                ; If the memory value is not zero,
                              ;  then jump to the label called '.loop',
                              ;  otherwise continue to the next line
.done:
                              ; We don't do a final increment,
                              ;  because even though the count is base 1,
                              ;  we do not include the zero terminator in the
                              ;  string's length
    ret                       ; Return to the calling program
```

in.asm

[*plain text*]

```
00000030 B9FFFFFFFF

00000035 41
00000036 803C0800

0000003A 75F9

0000003C C3
```

out.obj

[*binary*]

For small programs, this suffices, but why shouldn't we try to fit everything in a single *main.asm*?

Multi-source Object File Generation

Ideally, we want the flexibility* to split our code up across files

```
zstr_count:
    mov  ecx, -1

.loop:
    inc  ecx
    cmp  byte [eax + ecx], 0


    jne  .loop

.done:




    ret
```

in0.asm

```
zstr_count:
    mov  ecx, -1

.loop:
    inc  ecx
    cmp  byte [eax + ecx], 0


    jne  .loop

.done:




    ret
```

in1.asm

```
zstr_count:
    mov  ecx, -1

.loop:
    inc  ecx
    cmp  byte [eax + ecx], 0


    jne  .loop

.done:




    ret
```

in2.asm

```
zstr_count:
    mov  ecx, -1

.loop:
    inc  ecx
    cmp  byte [eax + ecx], 0


    jne  .loop

.done:




    ret
```

in3.asm

→

```
00000030 B9FFFFFFFF


00000035 41
00000036 803C0800


0000003A 75F9






0000003C C3
```

out.obj

[*binary*]

Multi-source Object File Generation

Here’s a better example:

```
zstr_count:
    mov   ecx, -1

.loop:
    inc   ecx
    cmp   byte [eax + ecx], 0


    jne   .loop


.done:




    ret
```

gfx.asm

```
zstr_count:
    mov   ecx, -1

.loop:
    inc   ecx
    cmp   byte [eax + ecx], 0


    jne   .loop


.done:




    ret
```

snd.asm

```
zstr_count:
    mov   ecx, -1

.loop:
    inc   ecx
    cmp   byte [eax + ecx], 0


    jne   .loop


.done:




    ret
```

ai.asm

```
zstr_count:
    mov   ecx, -1

.loop:
    inc   ecx
    cmp   byte [eax + ecx], 0


    jne   .loop


.done:




    ret
```

net.asm

```
00000030 B9FFFFFFFF


00000035 41
00000036 803C0800


0000003A 75F9







0000003C C3
```

game.obj

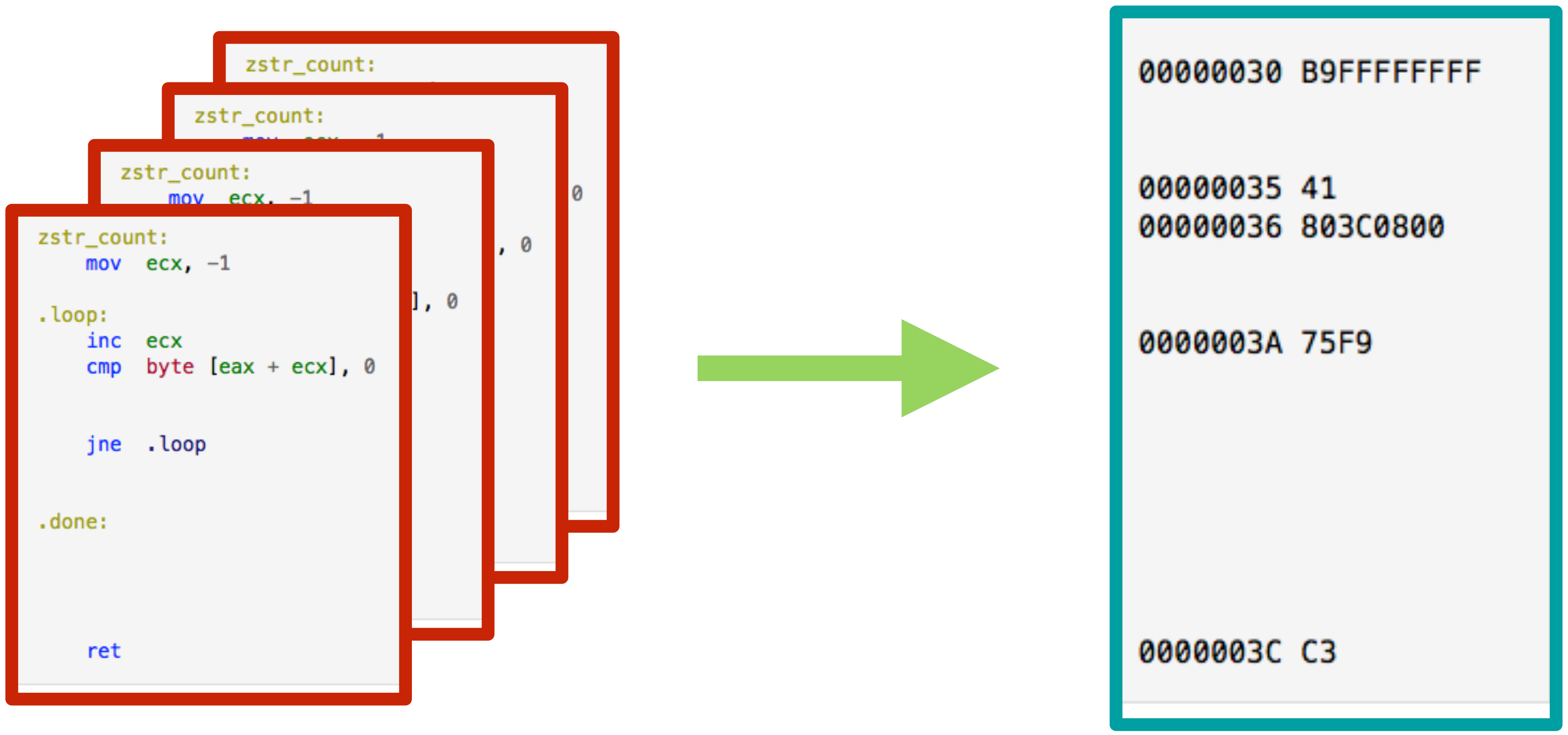
[binary]

Some good reasons: specialization/modularity, team work

Multi-source Object File Generation

Does the previous two-pass assembler algorithm work in this multi-input scenario?

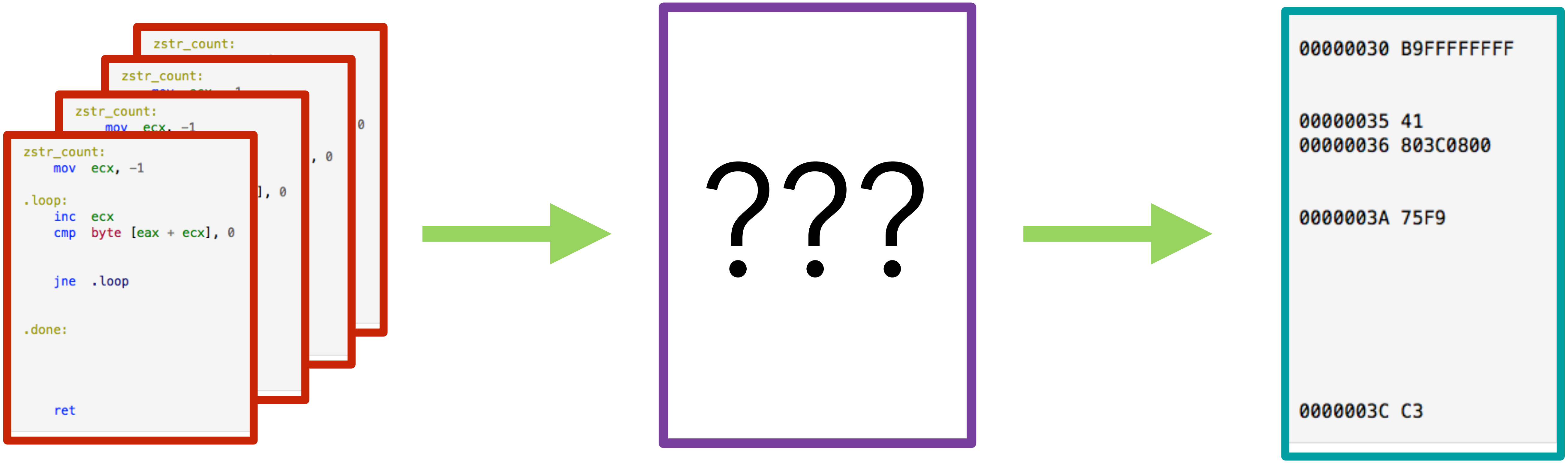
- where does it break down?



Enter the Linker

To solve this problem, we need to introduce another tool: the **linker**

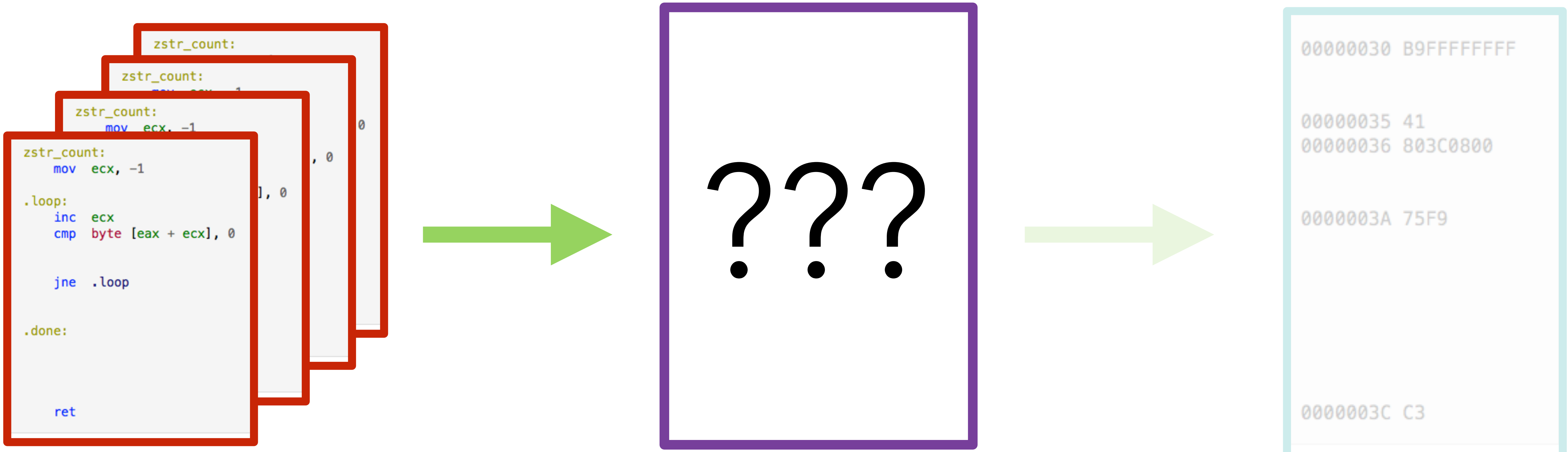
- a linker works *in tandem* with an assembler



Enter the Linker

How does the output of the assembler (i.e., the input to the linker) need to change?

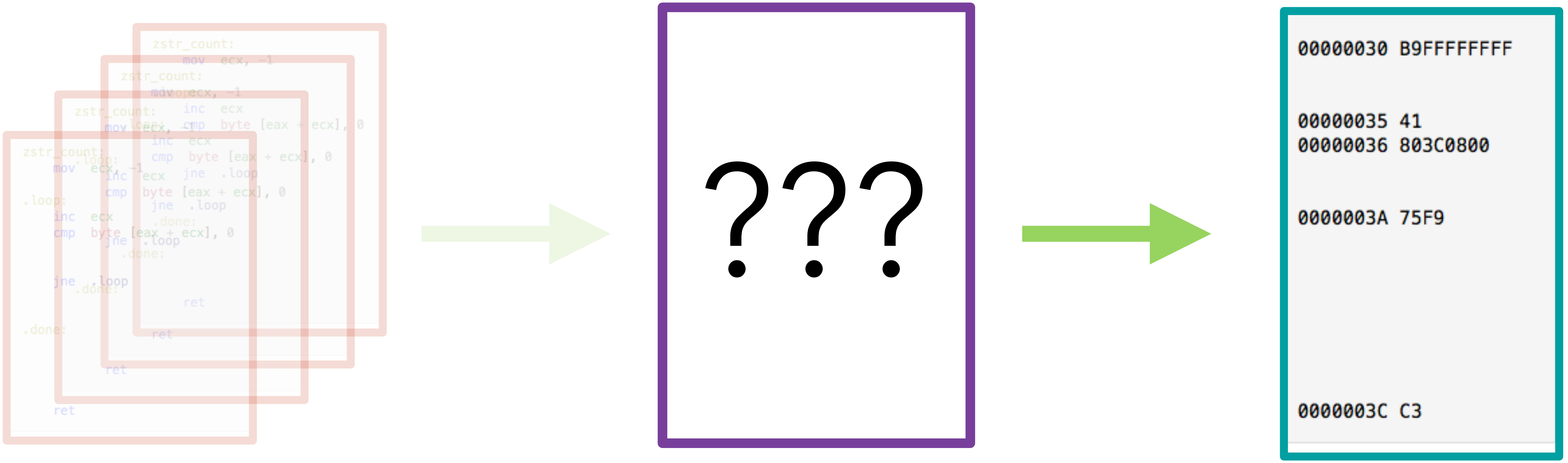
How does the linker process this output to generate the final object program?



Enter the Linker

How does the output of the assembler (i.e., the input to the linker) need to change?

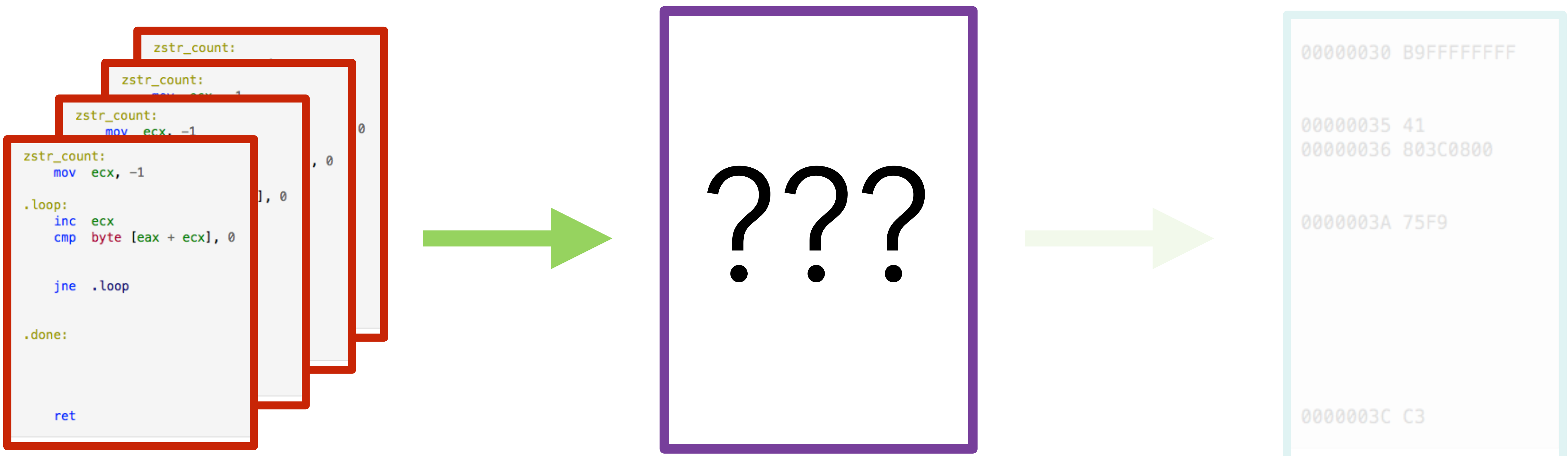
How does the linker process this output to generate the final object program?



Assembling Multiple Source Files

First, we assemble source files **separately**

- unlike the 1-source file case, the assembler may come across external references that are in another source file



Assembling Multiple Source Files

We need to deal with the fact that external references may not be resolved during a first (or second) pass through any single source file

```
.loop:
    mov eax, -1
    inc eax

    call someExternalFunction

    jmp loop

    ret
```

gfx.asm

```
.someExternalFunction:
    cmp byte [eax + ecx]
    mov eax, ecx
    ret
```

snd.asm

Assembling Multiple Source Files

So, now, an assembler has more responsibilities:

- follow the original two-pass process to generate:
  - memory mapped binary object content
  - an exportable symbol table
  - a list of **externally unresolved references**

```
.loop:
    mov eax, -1
    inc eax

    call someExternalFunction

    jmp loop

    ret
```

gfx.asm

- memory mapped binary object content
- an exportable symbol table
- a list of **externally unresolved references**

Object Program Memory Map

Memory Address	OP Code/Data
0x00	
0x03	
0x04	
0x06	
0x08	

```
.loop:
    mov eax, -1
    inc eax

    call someExternalFunction

    jmp loop

    ret
```

gfx.asm

Symbol Table

Symbol Name	Symbol Value

External References

External Reference Name

- memory mapped binary object content
- an exportable symbol table
- a list of **externally unresolved references**

Object Program Memory Map

Memory Address	OP Code/Data
0x00	
0x03	
0x04	
0x06	
0x08	

```
.loop:
    mov eax, -1
    inc eax

    call someExternalFunction

    jmp loop

    ret
```

Symbol Table

Symbol Name	Symbol Value
loop	

External References

External Reference Name

- memory mapped binary object content
- an exportable symbol table
- a list of **externally unresolved references**

Object Program Memory Map

Memory Address	OP Code/Data
0x00	
0x03	
0x04	
0x06	
0x08	

```
.loop:
    mov eax, -1
    inc eax

    call someExternalFunction

    jmp loop

    ret
```

Symbol Table

Symbol Name	Symbol Value
loop	0x00

External References

External Reference Name

- memory mapped binary object content
- an exportable symbol table
- a list of **externally unresolved references**

Object Program Memory Map

Memory Address	OP Code/Data
0x00	CDFFFF
0x03	
0x04	
0x06	
0x08	

```
.loop:
    mov eax, -1
    inc eax

    call someExternalFunction

    jmp loop

    ret
```

Symbol Table

Symbol Name	Symbol Value
loop	0x00

External References

External Reference Name

- memory mapped binary object content
- an exportable symbol table
- a list of **externally unresolved references**

Object Program Memory Map

Memory Address	OP Code/Data
0x00	CDFFFF
0x03	
0x04	
0x06	
0x08	

```
.loop:
    mov eax, -1
    inc eax

    call someExternalFunction

    jmp loop

    ret
```

Symbol Table

Symbol Name	Symbol Value
loop	0x00

External References

External Reference Name

- memory mapped binary object content
- an exportable symbol table
- a list of **externally unresolved references**

Object Program Memory Map

Memory Address	OP Code/Data
0x00	CDFFFF
0x03	DE
0x04	
0x06	
0x08	

```
.loop:
    mov eax, -1
    inc eax

    call someExternalFunction

    jmp loop

    ret
```

Symbol Table

Symbol Name	Symbol Value
loop	0x00

External References

External Reference Name

- memory mapped binary object content
- an exportable symbol table
- a list of **externally unresolved references**

Object Program Memory Map

Memory Address	OP Code/Data
0x00	CDFFFF
0x03	DE
0x04	FA??
0x06	
0x08	

```
.loop:
    mov eax, -1
    inc eax

    call someExternalFunction

    jmp loop

    ret
```

Symbol Table

Symbol Name	Symbol Value
loop	0x00

External References

External Reference Name

- memory mapped binary object content
- an exportable symbol table
- a list of **externally unresolved references**

Object Program Memory Map

Memory Address	OP Code/Data
0x00	CDFFFF
0x03	DE
0x04	FA??
0x06	
0x08	

```
.loop:
    mov eax, -1
    inc eax

    call someExternalFunction

    jmp loop

    ret
```

Symbol Table

Symbol Name	Symbol Value
loop	0x00

External References

External Reference Name
someExternalFunction

- memory mapped binary object content
- an exportable symbol table
- a list of **externally unresolved references**

Object Program Memory Map

Memory Address	OP Code/Data
0x00	CDFFFF
0x03	DE
0x04	FA??
0x06	
0x08	

```
.loop:
    mov eax, -1
    inc eax

    call someExternalFunction

    jmp loop

    ret
```

Symbol Table

Symbol Name	Symbol Value
loop	0x00

External References

External Reference Name
someExternalFunction

- memory mapped binary object content
- an exportable symbol table
- a list of **externally unresolved references**

Object Program Memory Map

Memory Address	OP Code/Data
0x00	CDFFFF
0x03	DE
0x04	FA??
0x06	FB00
0x08	

```
.loop:
    mov eax, -1
    inc eax

    call someExternalFunction

    jmp loop

    ret
```

Symbol Table

Symbol Name	Symbol Value
loop	0x00

External References

External Reference Name
someExternalFunction

- memory mapped binary object content
- an exportable symbol table
- a list of **externally unresolved references**

Object Program Memory Map

Memory Address	OP Code/Data
0x00	CDFFFF
0x03	DE
0x04	FA??
0x06	FB00
0x08	

```
.loop:
    mov eax, -1
    inc eax

    call someExternalFunction

    jmp loop

    ret
```

Symbol Table

Symbol Name	Symbol Value
loop	0x00

External References

External Reference Name
someExternalFunction

- memory mapped binary object content
- an exportable symbol table
- a list of **externally unresolved references**

Object Program Memory Map

Memory Address	OP Code/Data
0x00	CDFFFF
0x03	DE
0x04	FA??
0x06	FB00
0x08	80

```
.loop:
    mov eax, -1
    inc eax

    call someExternalFunction

    jmp loop

    ret
```

Symbol Table

Symbol Name	Symbol Value
loop	0x00

External References

External Reference Name
someExternalFunction

- memory mapped binary object content
- an exportable symbol table
- a list of **externally unresolved references**

Memory	OP Code/
0x00	CDFFFF
0x03	DE
0x04	FA??
0x06	FB00
0x08	80

Symbol Name	Symbol Value
loop	0x00

External Names
someExternalFunction

gfx.obj

[*binary*]

```
.loop:
    mov eax, -1
    inc eax

    call someExternalFunction

    jmp loop

    ret
```

- memory mapped binary object content
- an exportable symbol table
- a list of **externally unresolved references**

Object Program Memory

Memory Address	OP Code/Data
0x00	
0x03	
0x05	

```
.someExternalFunction:
    cmp byte [eax + ecx]
    mov eax, ecx
    ret
```

snd.asm

Symbol Table

Symbol Name	Symbol Value

External References

External Reference Name

- memory mapped binary object content
- an exportable symbol table
- a list of **externally unresolved references**

Object Program Memory

Memory Address	OP Code/Data
0x00	
0x03	
0x05	

```
.someExternalFunction:
    cmp byte [eax + ecx]
    mov eax, ecx
    ret
```

Symbol Table

Symbol Name	Symbol Value
someExternalFunction	

External References

External Reference Name

- memory mapped binary object content
- an exportable symbol table
- a list of **externally unresolved references**

Object Program Memory

Memory Address	OP Code/Data
0x00	
0x03	
0x05	

```
.someExternalFunction:
    cmp byte [eax + ecx]
    mov eax, ecx
    ret
```

Symbol Table

Symbol Name	Symbol Value
someExternalFunction	0x00

External References

External Reference Name

- memory mapped binary object content
- an exportable symbol table
- a list of **externally unresolved references**

Object Program Memory

Memory Address	OP Code/Data
0x00	
0x03	
0x05	

```
.someExternalFunction:
    cmp byte [eax + ecx]
    mov eax, ecx
    ret
```

Symbol Table

Symbol Name	Symbol Value
someExternalFunction	0x00

External References

External Reference Name

- memory mapped binary object content
- an exportable symbol table
- a list of **externally unresolved references**

Object Program Memory

Memory Address	OP Code/Data
0x00	75038A
0x03	
0x05	

```
.someExternalFunction:
    cmp byte [eax + ecx]
    mov eax, ecx
    ret
```

Symbol Table

Symbol Name	Symbol Value
someExternalFunction	0x00

External References

External Reference Name

- memory mapped binary object content
- an exportable symbol table
- a list of **externally unresolved references**

Object Program Memory

Memory Address	OP Code/Data
0x00	75038A
0x03	
0x05	

```
.someExternalFunction:
    cmp byte [eax + ecx]
    mov eax, ecx
    ret
```

Symbol Table

Symbol Name	Symbol Value
someExternalFunction	0x00

External References

External Reference Name

- memory mapped binary object content
- an exportable symbol table
- a list of **externally unresolved references**

Object Program Memory

Memory Address	OP Code/Data
0x00	75038A
0x03	CD7C
0x05	

```
.someExternalFunction:
    cmp byte [eax + ecx]
    mov eax, ecx
    ret
```

Symbol Table

Symbol Name	Symbol Value
someExternalFunction	0x00

External References

External Reference Name

- memory mapped binary object content
- an exportable symbol table
- a list of **externally unresolved references**

Object Program Memory

Memory Address	OP Code/Data
0x00	75038A
0x03	CD7C
0x05	

```
.someExternalFunction:
    cmp byte [eax + ecx]
    mov eax, ecx
    ret
```

Symbol Table

Symbol Name	Symbol Value
someExternalFunction	0x00

External References

External Reference Name

- memory mapped binary object content
- an exportable symbol table
- a list of **externally unresolved references**

Object Program Memory

Memory Address	OP Code/Data
0x00	75038A
0x03	CD7C
0x05	80

```
.someExternalFunction:
    cmp byte [eax + ecx]
    mov eax, ecx
    ret
```

Symbol Table

Symbol Name	Symbol Value
someExternalFunction	0x00

External References

External Reference Name

- memory mapped binary object content
- an exportable symbol table
- a list of **externally unresolved references**

Memory Address	OP Code/ Data
0x00	75038A
0x03	CD7C
0x05	80

Symbol Name	Symbol Value
someExternalFunction	0x00

External Names

snd.obj

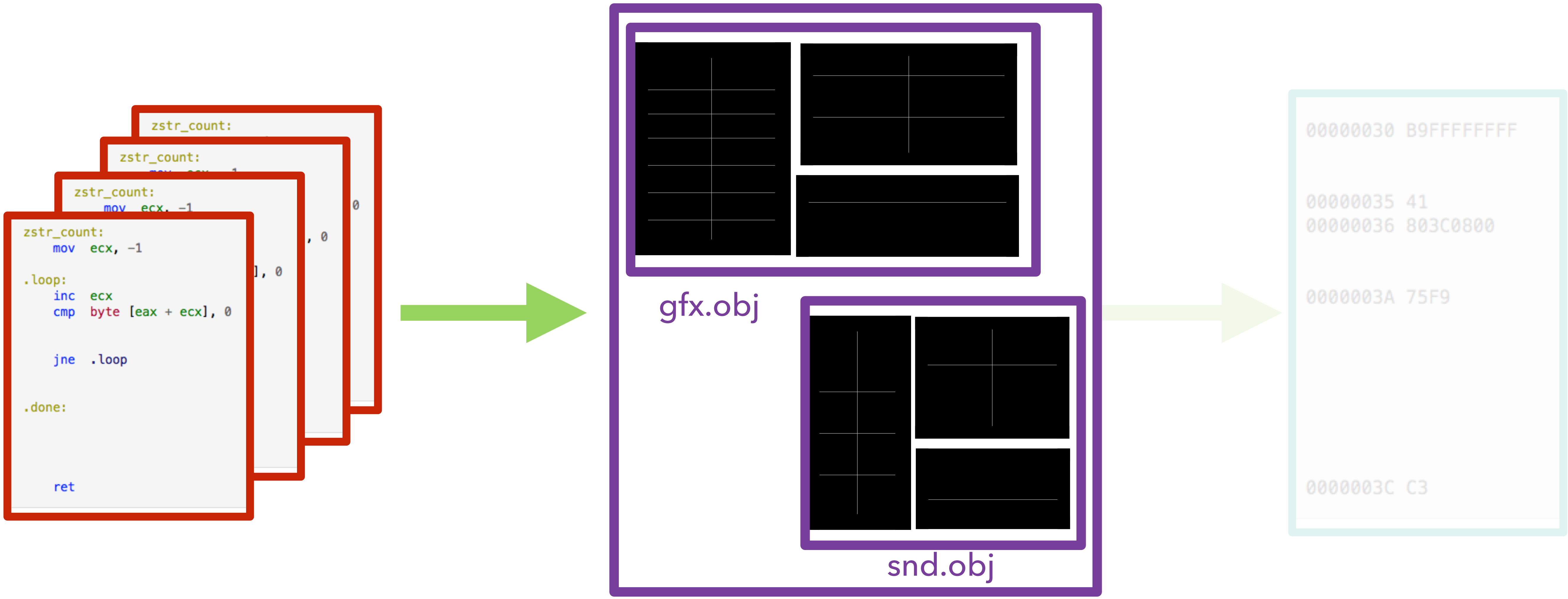
[binary]

```
.someExternalFunction:
    cmp byte [eax + ecx]
    mov eax, ecx
    ret
```

Assembling Multiple Source Files

After separately assembling each source file, we forward individual **object files** to the linker

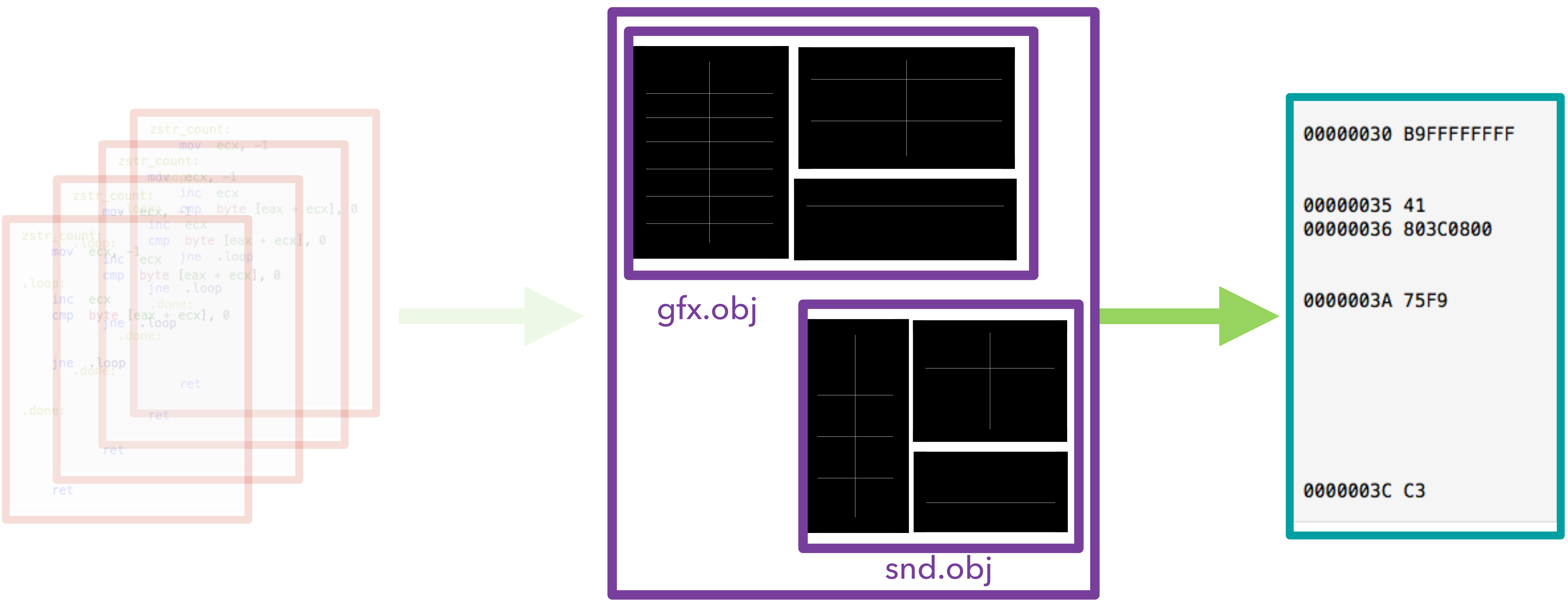
- each one stores (potentially incomplete) memory maps, symbol tables, and external references



Enter the Linker

Can you guess what the linker does with these?

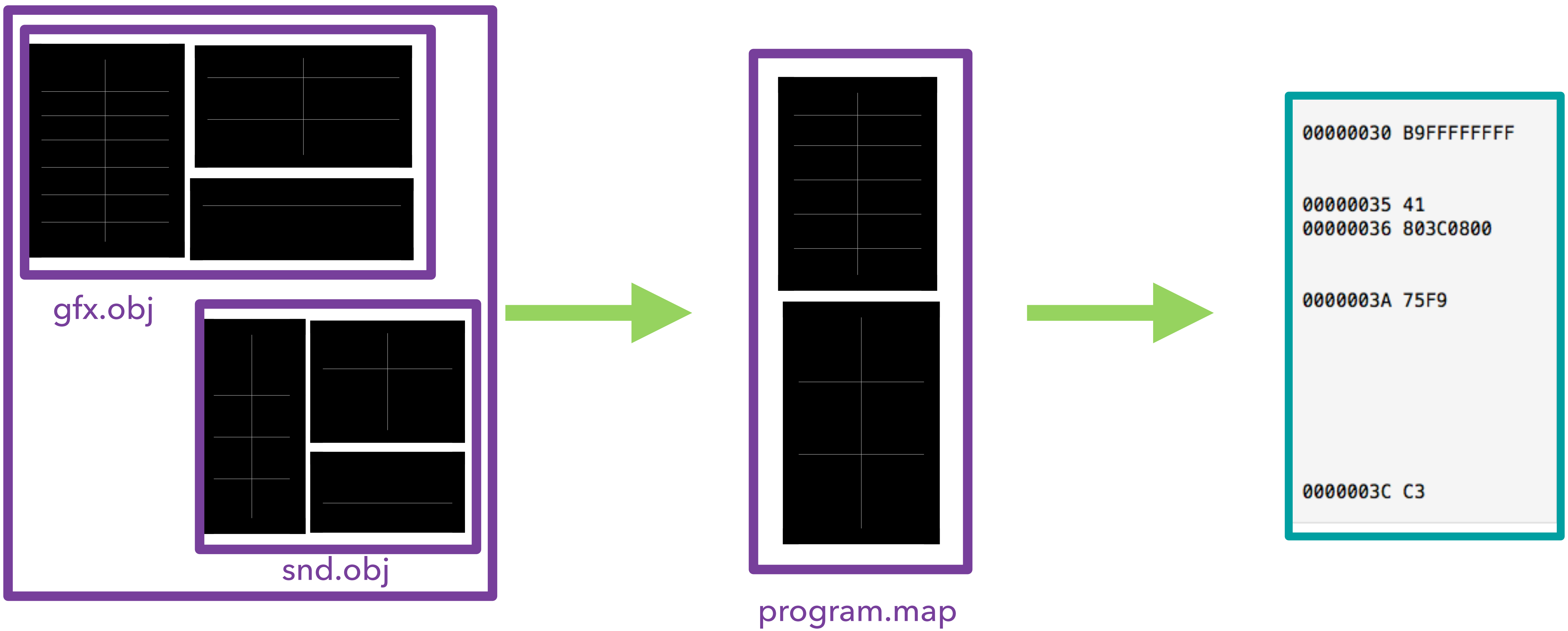
- any missing references **across object files** need to be resolved



Enter the Linker

Can you guess what the linker does with these?

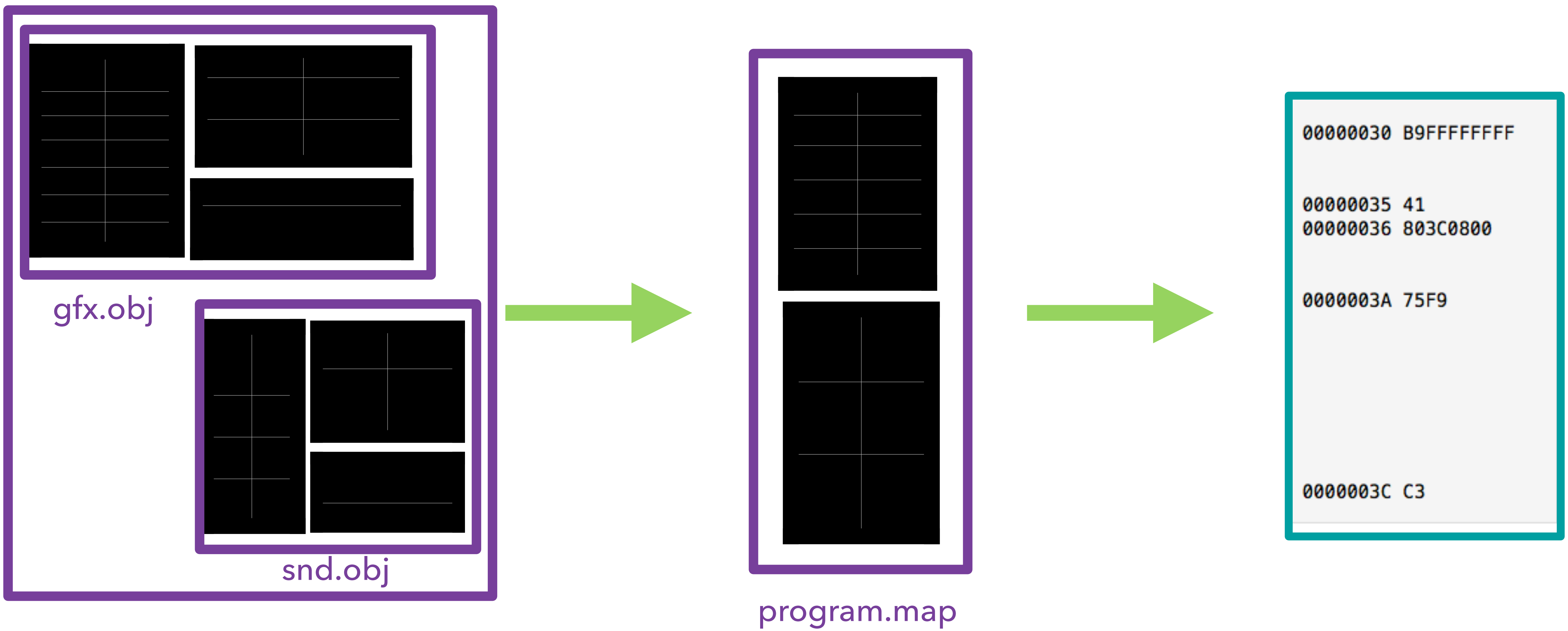
- first, we combine the object binaries into a single sequential memory map



Enter the Linker

Can you guess what the linker does with these?

- we'll eventually need a *globally consistent* (relative) memory **map** *across* individual object memory maps



Enter the Linker

Then, the linker identifies missing references…

snd.obj

Memory Address	OP Code/ Data
0x00	75038A
0x03	CD7C
0x05	80

Symbol Name	Symbol Value
someExternalFunction	0x00

External Names

gfx.obj

Memory	OP Code/
0x00	CDFFFF
0x03	DE
0x04	FA??
0x06	FB00
0x08	80

Symbol Name	Symbol Value
loop	0x00

External Names
someExternalFunction

Enter the Linker

Then, the linker identifies missing references…

Memory Address	OP Code/ Data
0x00	75038A
0x03	CD7C
0x05	80

Symbol Name	Symbol Value
someExternalFunction	0x00

External Names

snd.obj

Memory	OP Code/
0x00	CDFFFF
0x03	DE
0x04	FA??
0x06	FB00
0x08	80

Symbol Name	Symbol Value
loop	0x00

External Names
someExternalFunction

gfx.obj

Enter the Linker

... searches other symbol tables for it...

Memory Address	OP Code/ Data
0x00	75038A
0x03	CD7C
0x05	80

Symbol Name	Symbol Value
someExternalFunction	0x00

External Names

snd.obj

Memory	OP Code/
0x00	CDFFFF
0x03	DE
0x04	FA??
0x06	FB00
0x08	80

Symbol Name	Symbol Value
loop	0x00

External Names
someExternalFunction

gfx.obj

Enter the Linker

... and replaces it with the mapped (offset) address

offset = 3F

Memory Address	OP Code/ Data
0x00	75038A
0x03	CD7C
0x05	80

Symbol Name	Symbol Value
someExternalFunction	0x00

External Names

snd.obj

- after processing, any remaining missing external references are reported as errors

Memory	OP Code/
0x00	CDFFFF
0x03	DE
0x04	FA3F
0x06	FB00
0x08	80

Symbol Name	Symbol Value
loop	0x00

External Names
someExternalFunction

gfx.obj

Modularity

One benefit of using separate source files is the ability to group different functional units together

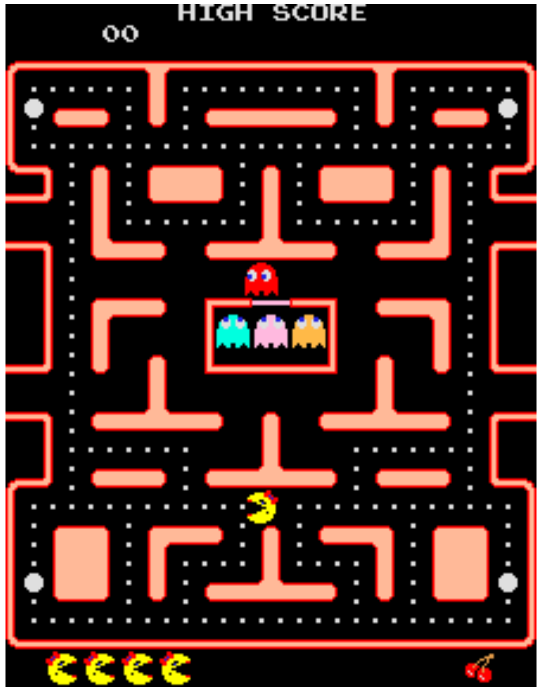
- often, many different applications can benefit from similar (if not identical) functionalities
    - advanced math routines
    - I/O and file processing routines
    - image and sound processing routines
    - networking routines
- here, it would be unfortunate/foolish to have to reinvent the wheel every time

Modularity

In the previous example, we assumed that every intermediate object binary is consumed

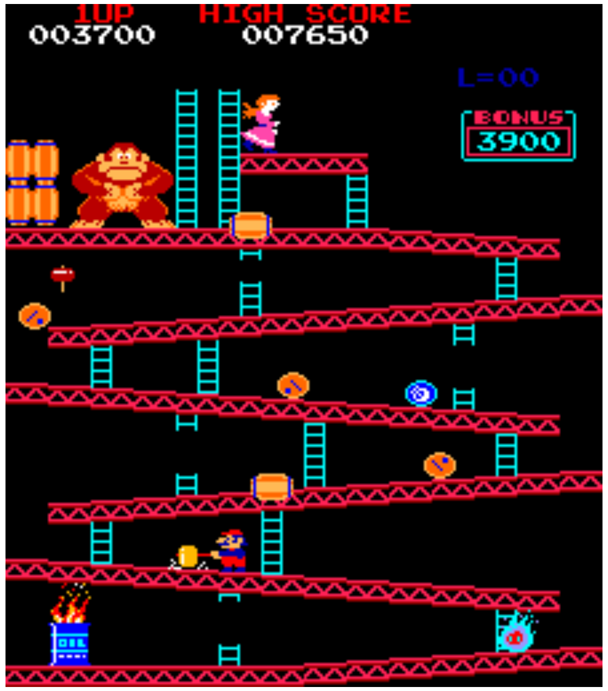
- once, and
- only for the object program that is being linked

For example, imaging writing **two** games:



pacman.obj

[binary]



dkong.obj

[binary]

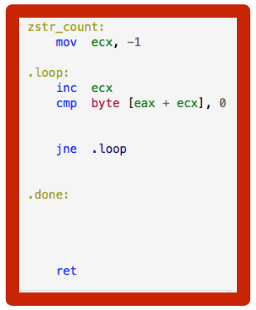
Modularity

You can imagine that these two games could have a significant overlap in their functionality:

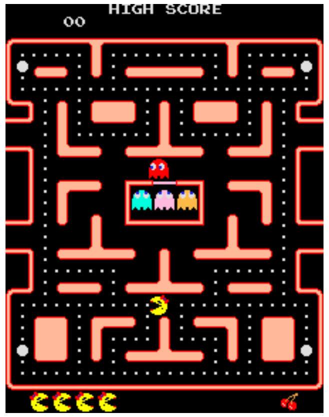
- graphics processing
- sound processing
- keypad processing

In fact, perhaps they only *differ* in their game logic

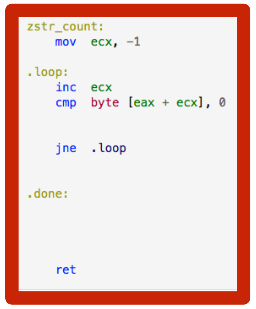
- here, it would be ideal to **reuse** code **across** programs



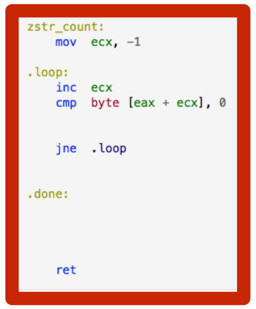
pacman.asm



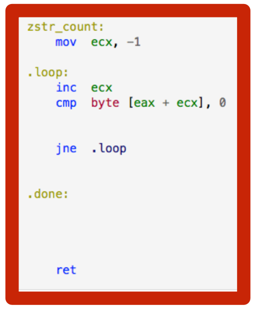
pacman.obj
[binary]



gfx.asm



snd.asm



key.asm

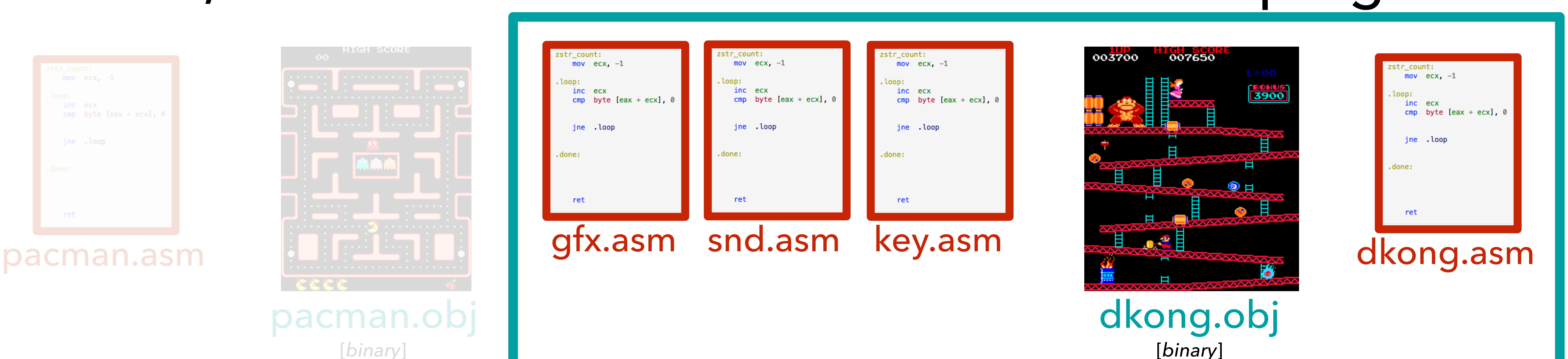
Modularity

You can imagine that these two games could have a significant overlap in their functionality:

- graphics processing
- sound processing
- keypad processing

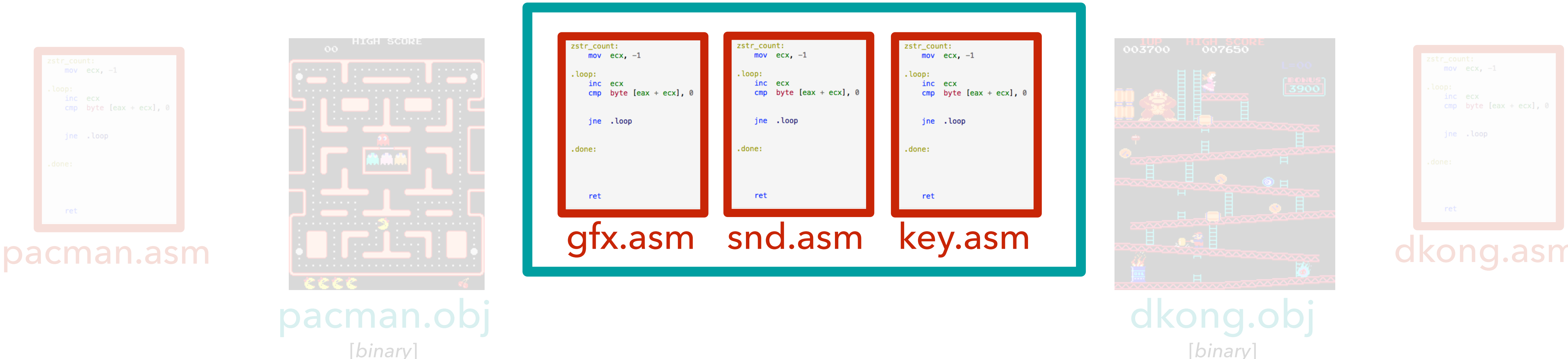
In fact, perhaps they only *differ* in their game logic

- here, it would be ideal to **reuse** code **across** programs

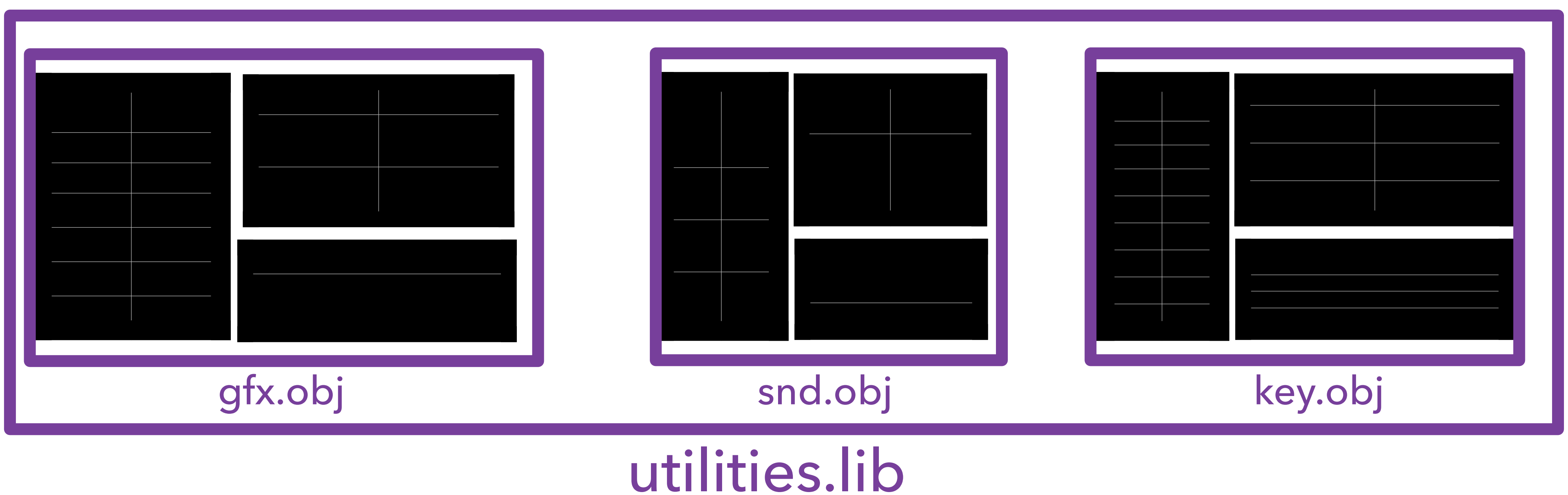


Libraries

We can package these shared routines into a **library**



Specifically, assemble + export: memory map, symbol and external reference tables for each object file



High-level Programming Languages

Assembly language coding requires a thorough understanding of the underlying CPU architecture

- pros:
  - understanding the implications of executed code
  - ability to fine-tune low-level behavior
- cons:
  - iteration time
  - barrier to entry
  - propensity for human error

High-level Programming Languages

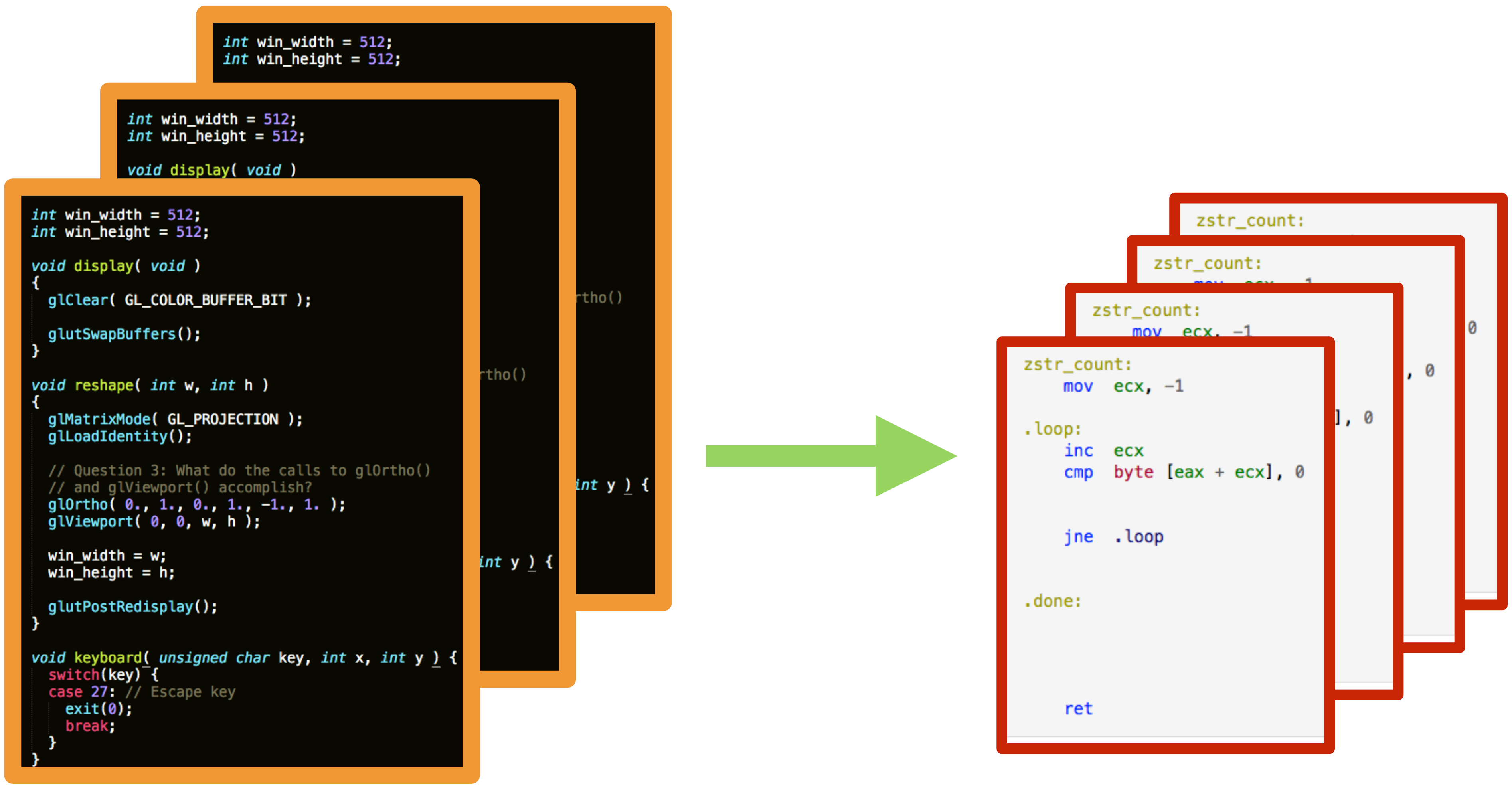
High-level programming languages reduce the need for such architecture-specific knowledge

- reduces the cost of cross-platform development
- decreases iteration time
- can simplify the design of larger, more complicated algorithms

The Compiler

A **compiler** converts a high-level source file (or file**s**) from the high-level language to assembly language

- after which, it invokes the *assembler* to generate object files

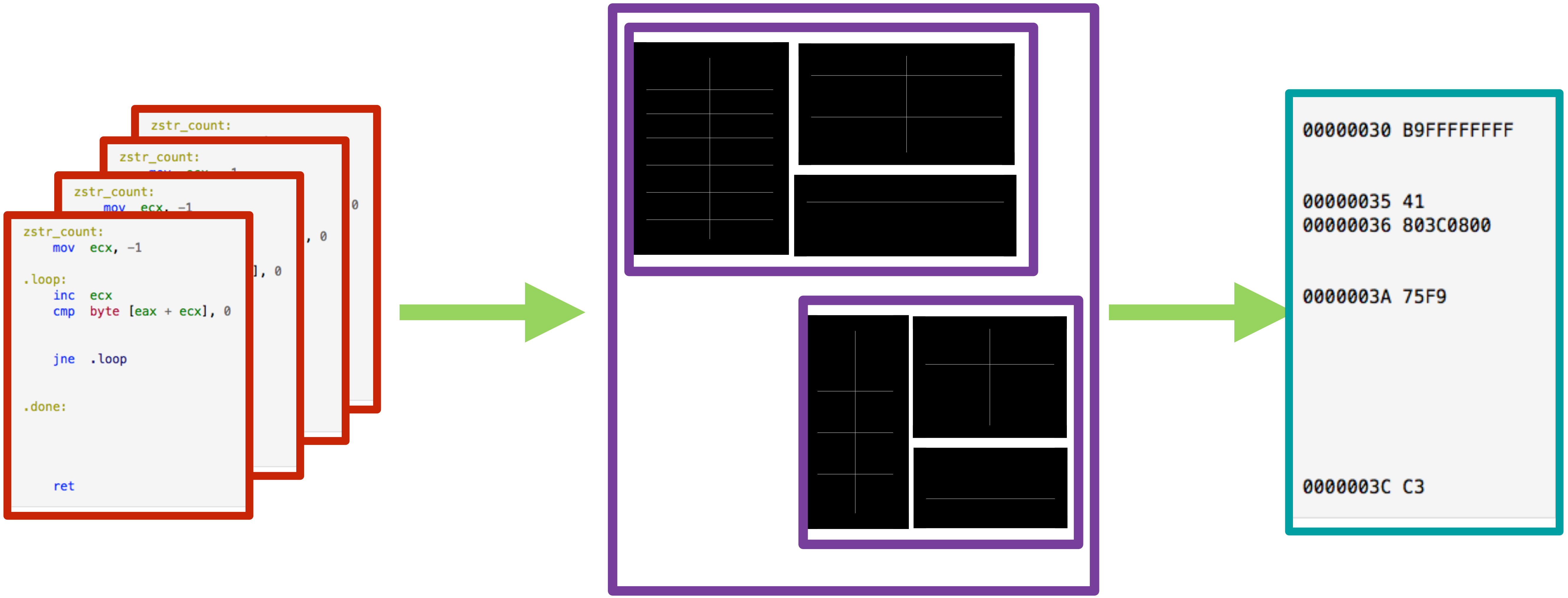


code.lang

The Compiler

A **compiler** converts a high-level source file (or file**s**) from the high-level language to assembly language

- after which, it invokes the *assembler* to generate object files
- after which, the linker generates the final object program



High-level Programming

High-level language compilers typically allow code to be split up across files/modules, too

- external references need to be explicitly identified for use inside a separate module*
- compiler will resolve names, external variable and subroutine addresses, at *compile-time*
    - except for subroutines & variables in external libraries
    - these are resolved at *link-time*

Compilers can also handle tedious tasks, like stack frame management for subroutines

- language features may add type and range checking, etc.

Compiler Optimizations

"Premature optimization is the root of all evil."

– Donald Knuth

Compiled assembler code makes no guarantee on size nor computational efficiency

- for a long time, the battle between hand-optimized assembler & compiler-generated assembler waged on

Automatic compiler optimization strategies work well

- platform-dependent and independent optimizations possible, even on heterogeneous compute architectures!
- active research area

Composed Software Solutions

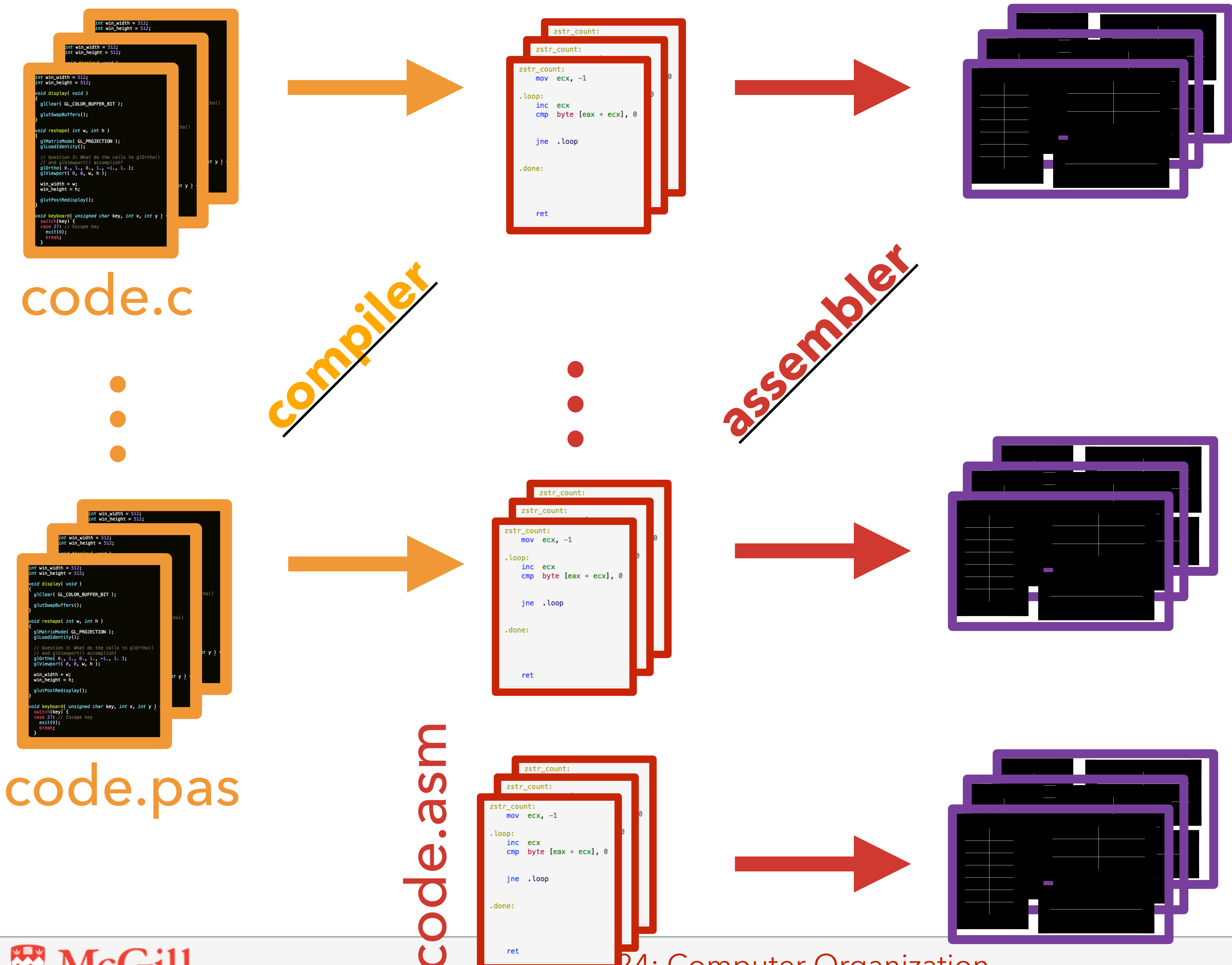
Multi-language Software Development

- Compilers convert high-level source to assembler
- Assemblers assemble source files to object data
- Linkers combine assembled object data into the final object program machine data
  - linkers may also draw from pre-assembled & packaged library binary object data archives

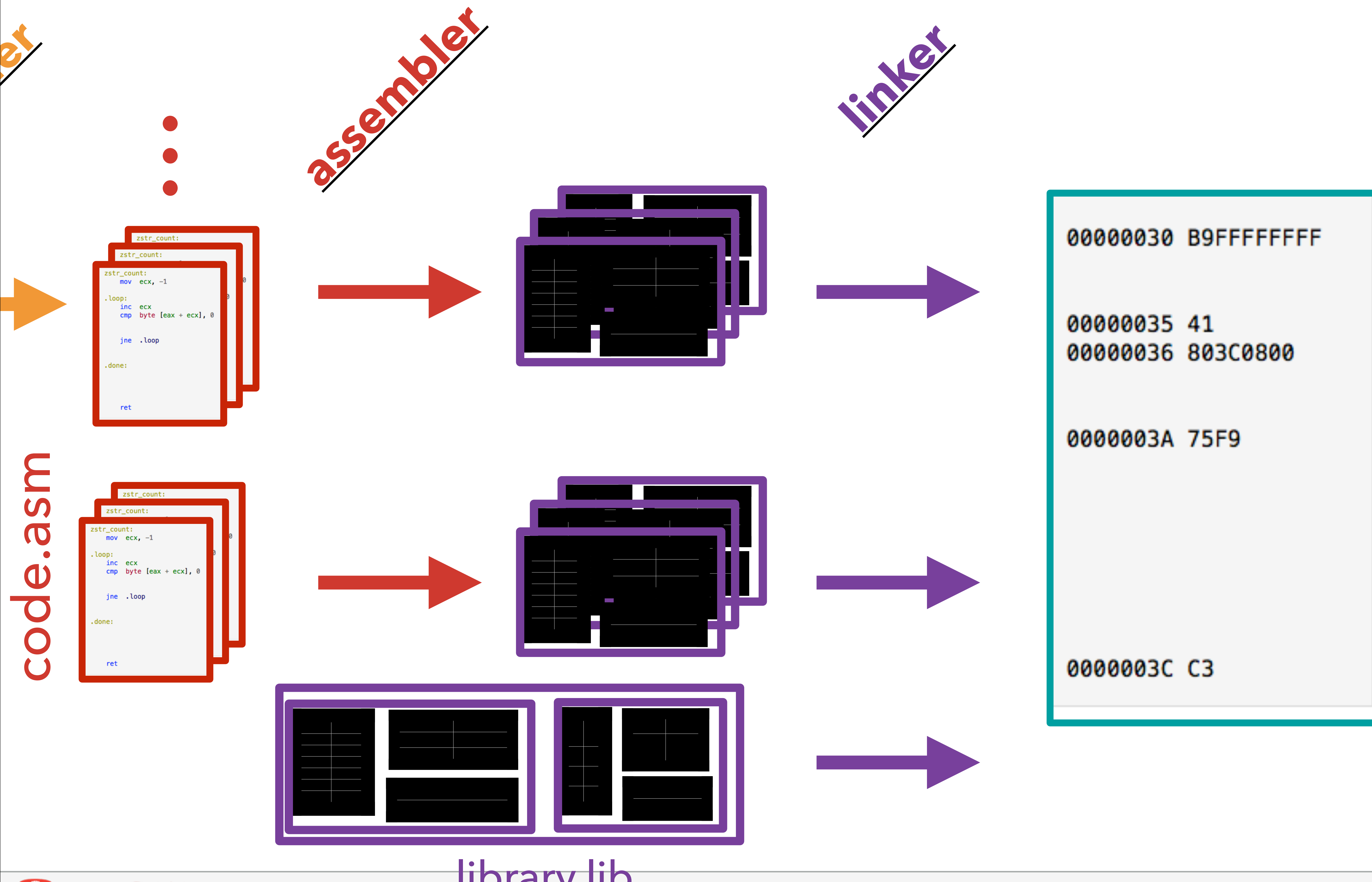
Conceptually, nothing prevents us from:

- mixing & matching high-level + assembler source
- using *many* different high-level languages

Multi-language Software Development



Multi-language Software Development



Multi-language Software Development

High-level languages can call assembler routines **and vice-versa**

- when calling assembler from high-level languages:
    - assembler code needs to respect the same subroutine calling conventions as the high-level language
    - high-level languages can "access" lower-level control
- here, relying on separate assembler source listings can sometimes become cumbersome
    - many compilers support *low-lever inlining* facilities

Inlined Assembler Code

```
int main(void) {

    int time = get_time();

    /* Add 10 and 20 and store result into register %eax */
    __asm__ ( "movl $10, %eax;"
              "movl $20, %ebx;"
              "addl %ebx, %eax;"
    );

    int result = -1;

      __asm__ (
        "int $0x80"           /* OS interrupt request */
        : "=a" (result),      /* return result in eax ("a") */
          "+c" (time),        /* pass time in ecx ("c") */
        : "a"  (0x180)        /* pass system call number
                                 in eax ("a") */
        : "memory", "cc"      /* notify compiler that
                                 memory and condition codes
                                 have changed */
    );

    return result + 10;
}
```

Inlined Assembler Code

```c
int main(void) {

    int time = get_time();

    /* Add 10 and 20 and store result into register %eax */
    __asm__ ( "movl $10, %eax;"
              "movl $20, %ebx;"
              "addl %ebx, %eax;"
    );

    int result = -1;

     __asm__ (
       "int $0x80"          /* OS interrupt request */
       : "=a" (result),     /* return result in eax ("a") */
         "+c" (time),       /* pass time in ecx ("c") */
       : "a"  (0x180)       /* pass system call number
                               in eax ("a") */
       : "memory", "cc"     /* notify compiler that
                               memory and condition codes
                               have changed */
    );

    return result + 10;
}
```

Inlined Assembler Code

```c
int main(void) {
    int time = get_time();

    /* Add 10 and 20 and store result into register %eax */
    __asm__ ( "movl $10, %eax;"
              "movl $20, %ebx;"
              "addl %ebx, %eax;"
    );

    int result = -1;

      __asm__ (
        "int $0x80"           /* OS interrupt request */
        : "=a" (result),      /* return result in eax ("a") */
          "+c" (time),        /* pass time in ecx ("c") */
        : "a"  (0x180)        /* pass system call number
                                 in eax ("a") */
        : "memory", "cc"      /* notify compiler that
                                 memory and condition codes
                                 have changed */
    );

    return result + 10;
}
```

Multi-language Software Development

High-level languages can call assembler routines **and vice-versa**

- when calling high-level routines from assembler:
    - need to match compiler-implemented subroutine calling mechanism/convention
        - pre- and post-conditions must match assembly- and calling-language conventions
            - stack and/or heap
            - condition bits/flags
            - register post-conditions

Debugging Strategies & Tools

Imagine:

- you've implemented your algorithm
- you've worked through compile errors (and warnings)
- you've worked through link errors
- you run your code and… it doesn't work
  - unexpected ("incorrect") output
  - program crash
  - infinite loop
  - etc.

How do you debug your problem?

Debugging Strategies & Tools

List of some debugging strategies/techniques:

- print statements
  - printing tags to highlight <u>execution flow</u>
    - *loop index variables & branch conditions*
  - printing final and intermediate <u>variable values</u>
- assertion statements
- unit tests
  - important to test both valid and invalid conditions

*Each of these strategies requires (re-)building & (re-)running your application**

Enter the Debugger

The **debugger** is a software tool that allows you to debug your application **while it runs**

- a more active way to track down and solve bugs
- debuggers sophisticate the process of bug tracking beyond earlier passive, build-dependent strategies

Concretely, a debugger allows you to:

- stop the execution of your program at any point
- examine (and modify!) the contents of registers, variables, and memory at this point
- resume execution until another point of interest

Enter the Debugger

To expose this advanced debugging functionality, debuggers leverage two key facilities:

- augmented build-generated object data
  - exposed through (advanced) software development tools
- execution-level control
  - exposed through (advanced) OS & HW facilities

Debugger – Debug Builds

Modern development toolchains (i.e., cross-compilers, compilers, assemblers, linkers) allow:

- mapping high-level code to its associated compiled/generated assembly code
- embedding object binaries with debug meta-data
    - explicit function and variable sizes and layout info
    - source-matched function and variable names

Debug builds are, as a result:

- less efficient* and less compact

Debugger – OS & HW Facilities

The ability of stopping, resuming, and modifying machine code and memory *during execution* requires more than just advanced dev tools

The OS and underlying HW platform must allow the disruption of normal execution protocols

- for example, the program counter is no longer the sole driving force of what gets executed next

A special interrupt-based HW feature, called *trace mode*, is exposed to the OS (who, in turn, exposes it to the debugger) to allow runtime debugging

Debugger — Trace Mode

When processors run in *trace mode*, they fire an interrupt **after the execution of each instruction**

- the OS exposes an associated interrupt-handler
- control flow is then relinquished to the debugger
    - the user can now execute debugger commands to:
        - view and edit memory (including variables)
        - view and edit registers and control flags
    - this interrupt is disabled during debugging
    - a *return-from-interrupt* is posted once the user commands regular execution flow continuation (which subsequently re-enables the interrupt)

Debugger — Breakpoints

The compiler/assembler and debugger allow source- and instruction-level ***breakpoints*** to inserted in code

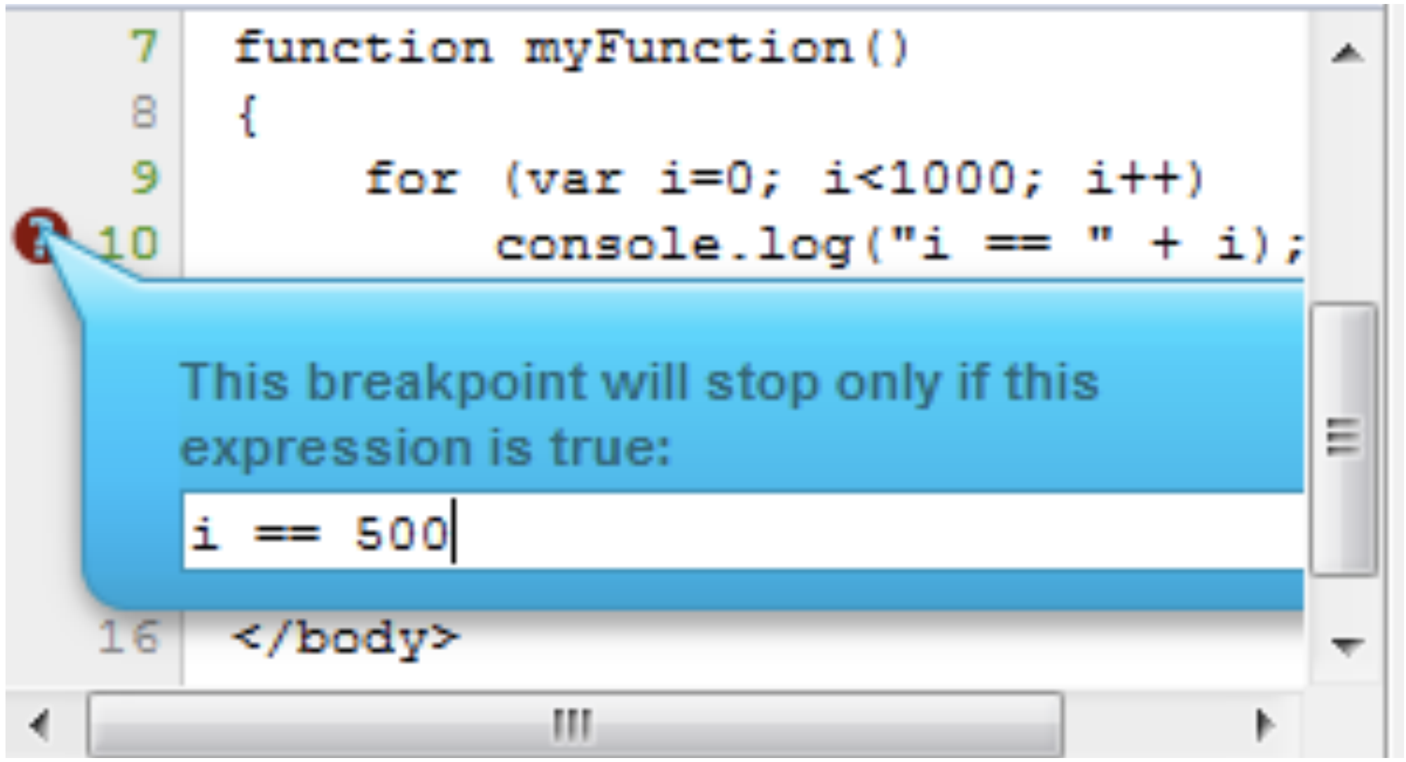
- a similar interrupt-based facility is signaled upon the execution of an instruction at a breakpoint
- control flow is once again relinquished to the debugger

Advanced development tools will allow for complex *conditional breakpoints* to be defined throughout the code

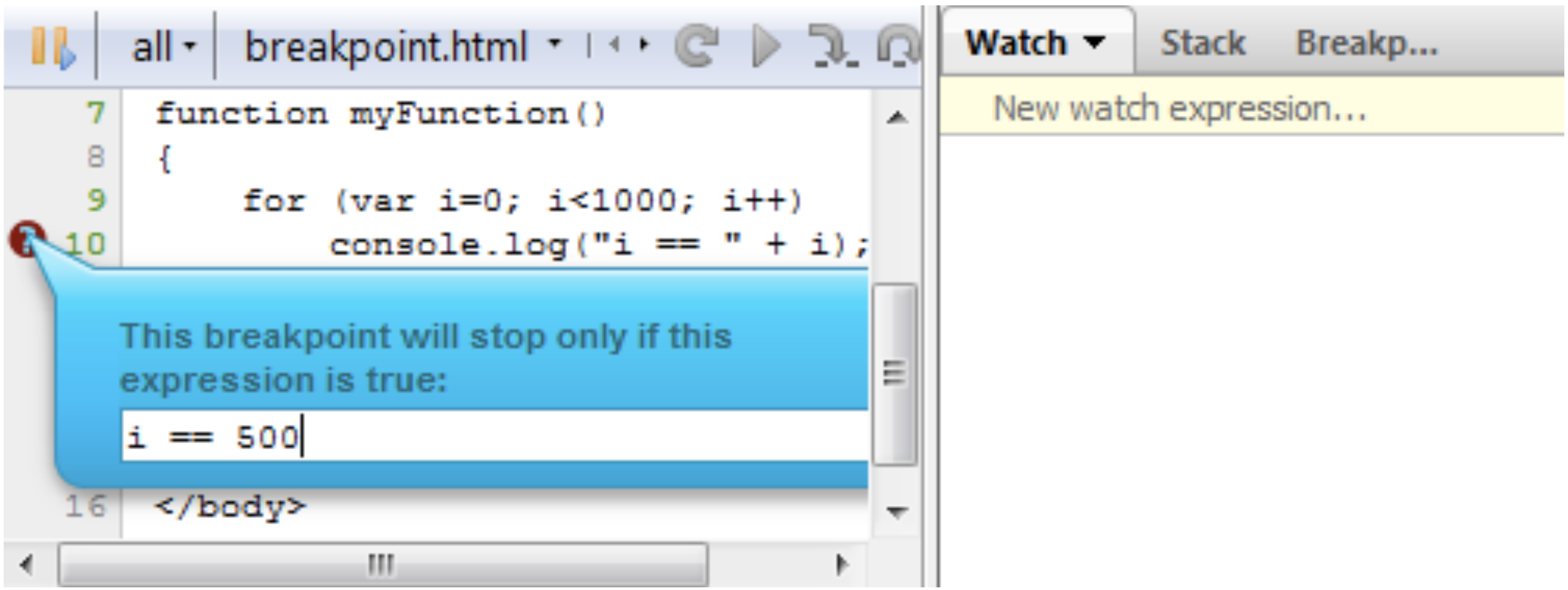
Debugger – Breakpoints

```
private void button1_Click(object sender, EventArgs e)
{
    int LetterCount = 0;
    string strText = "Debugging";
    string letter;

    for (int i = 0; i < strText.Length; i++)
    {
        letter = strText.Substring(1, 1);

        if (letter == "g")
        {
            LetterCount++;
        }
    }

    textBox1.Text = "g appears " + LetterCount + " times";
}
```

Debugger – Breakpoints



Debugger – Breakpoints



State-of-the-art Debuggers

A major differentiating technology between mature and immature software- and hardware-platforms is the **quality** and **capabilities** of their development toolchains

- not just the compilers, assemblers and linkers
- debuggers play a large role here*

Debugger development has remained an open area of applied research

- accommodating for more complex platforms
- more advanced debugging facilities*

The Operating System

At a high-level, the OS is responsible for:

- coordinating the execution of (potentially many) user-land applications
- managing the resources exposed to users
  - (equitable?) sharing of HW resources
    - managing memory and I/O requests
  - providing the illusion* of parallel execution
    - hiding latency from dependencies outside the processor (e.g., RAM, HD, etc.)

The *loader* is a component* of the OS

The Boot-strapping Process

What happens when you boot up your laptop?

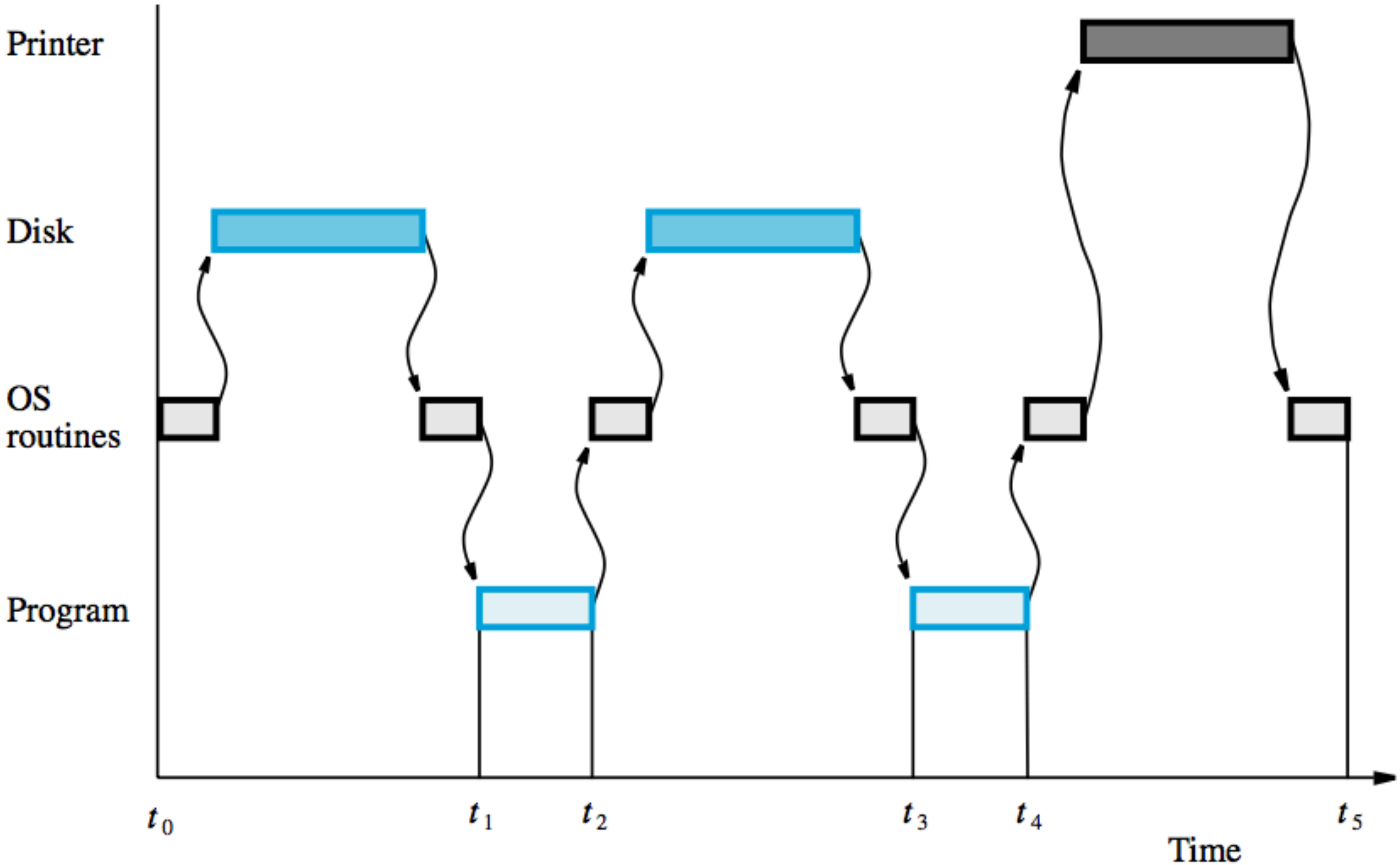
- **B**asic **I**nput-**O**utput **S**ystem (BIOS) runs
    - initializes the system and sets the PC at a pre-determined starting point in memory
        - the *bootloader*
- Bootloader most-likely *boots* your OS
- Sophisticated OSes are huge; during OS boot:
    - control of resources gradually relinquished to the OS (i.e., *daemons* are deployed at this point)
    - OS progressively loaded until user code is allowed to run (OS is "in charge", at this stage)

Life as an Application on an OS

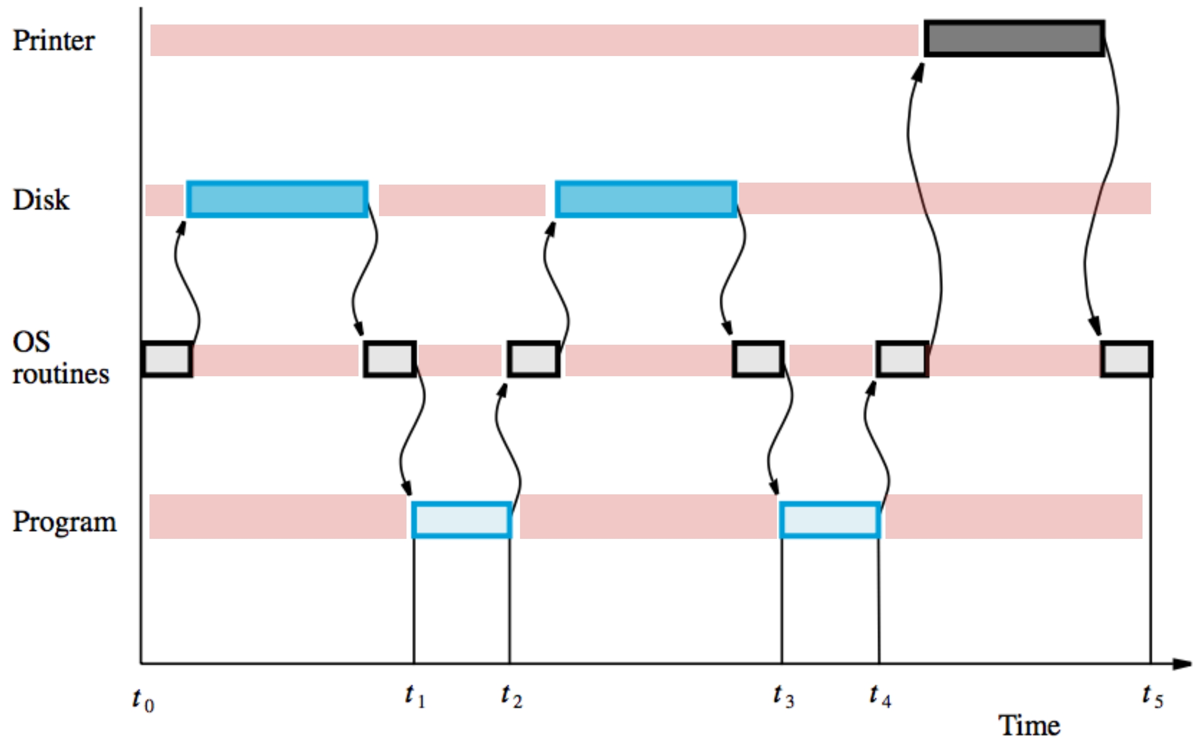
Life is hard for an application running on an OS

- you're allowed only *direct* access to a limited subset of the resources on the platform
- the OS decides when and how to dole out:
  - CPU processing access; this is is *time-shared* between applications
  - access to external resources (e.g., peripherals, disk); managed using *request-based mechanisms*

Life as an Application on an OS



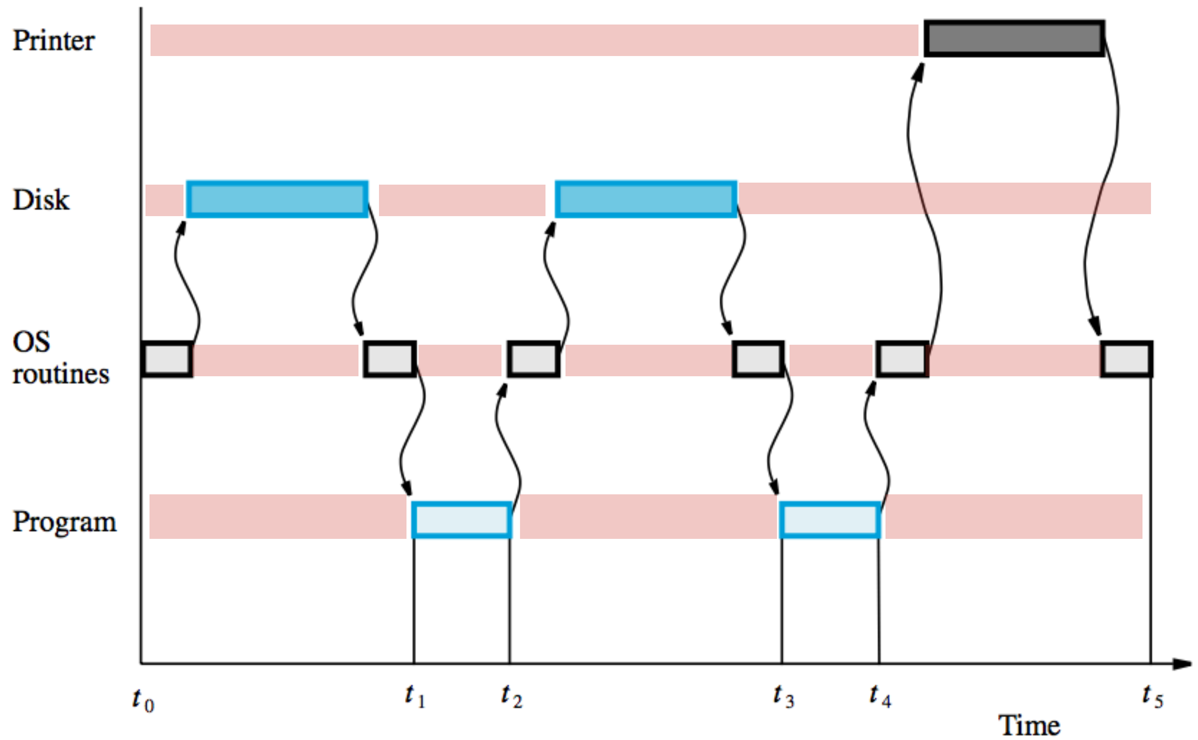
Life as an Application on an OS



In this example, it's clear that system resources are not managed to their full potential

- CPU is not at 100% utilization
- I/O devices are not at 100% utilization

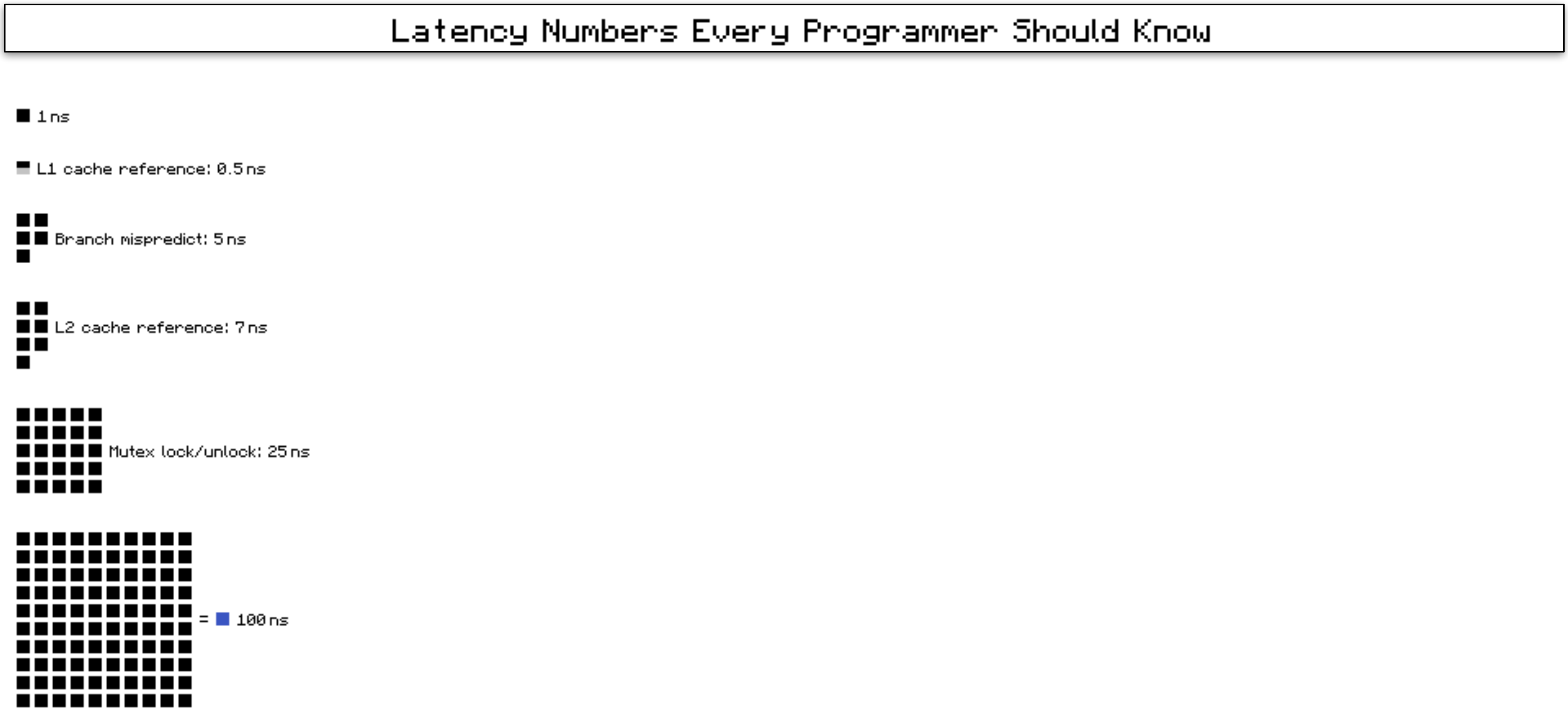
Life as an Application on an OS



Multitasking (a.k.a. multiprogramming) OSes better manage these inefficiencies by *scheduling resource utilization **across** applications*

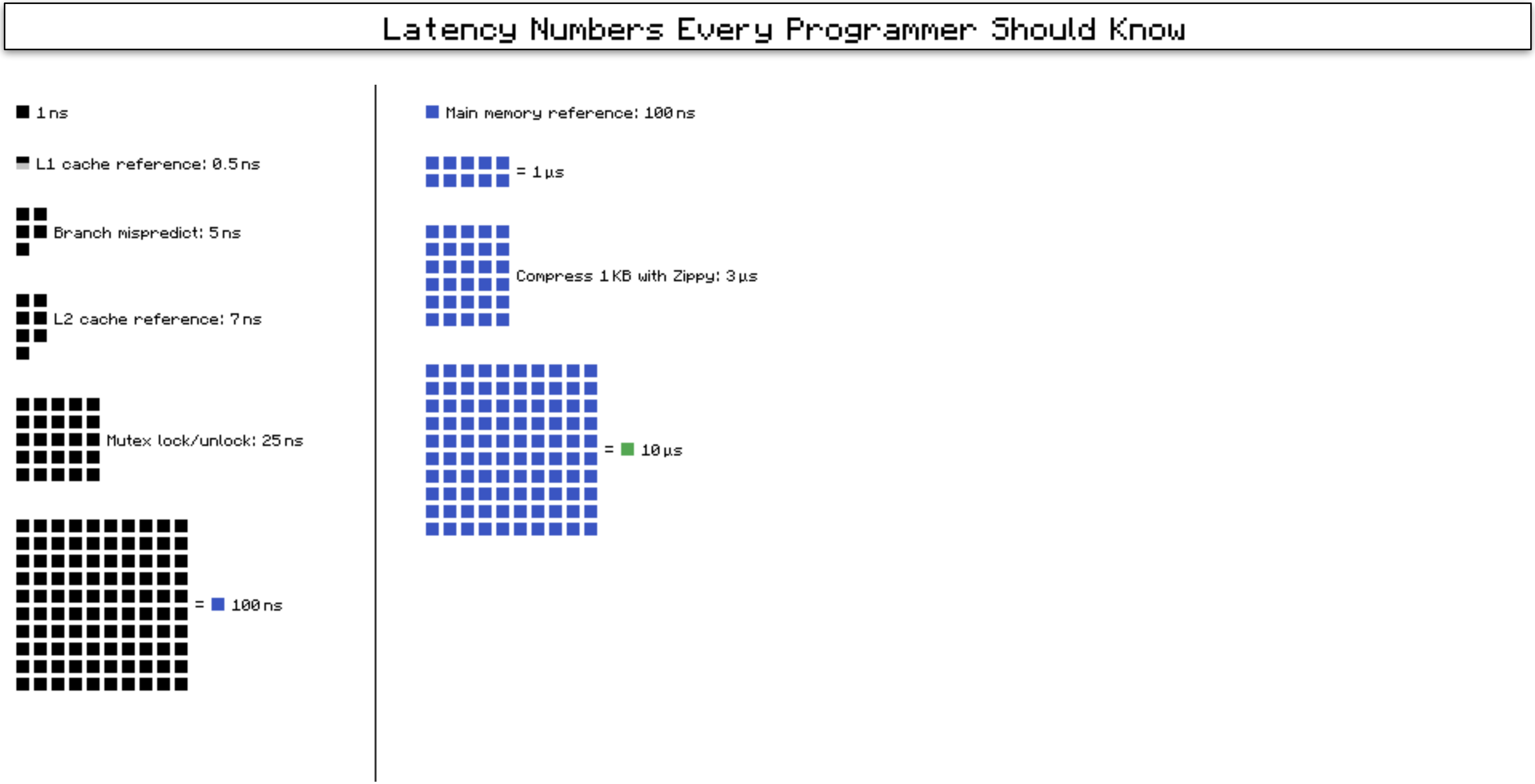
- latency-hiding can happen across scales

Life as an Application on an OS



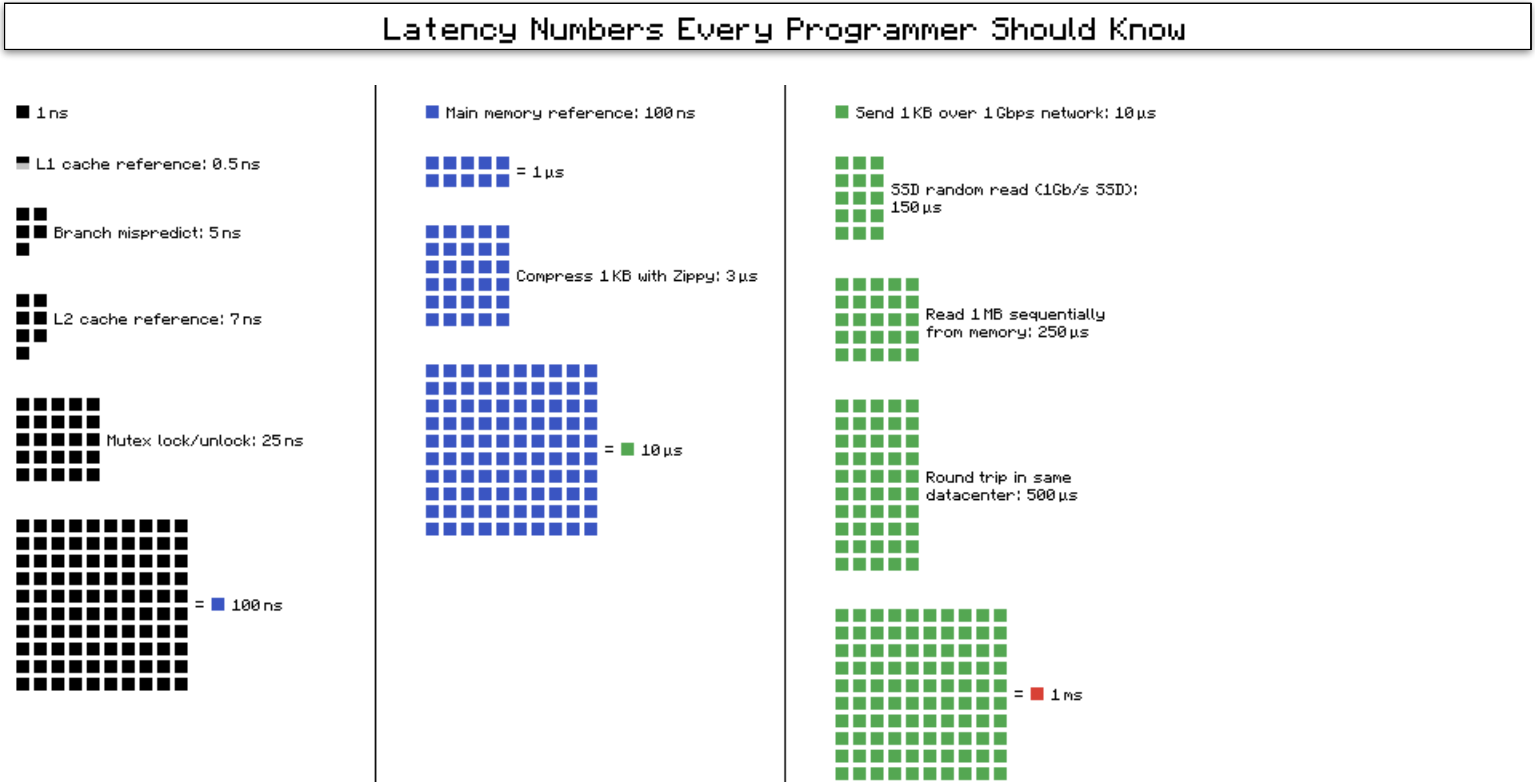
Source: https://gist.github.com/2841832

Life as an Application on an OS



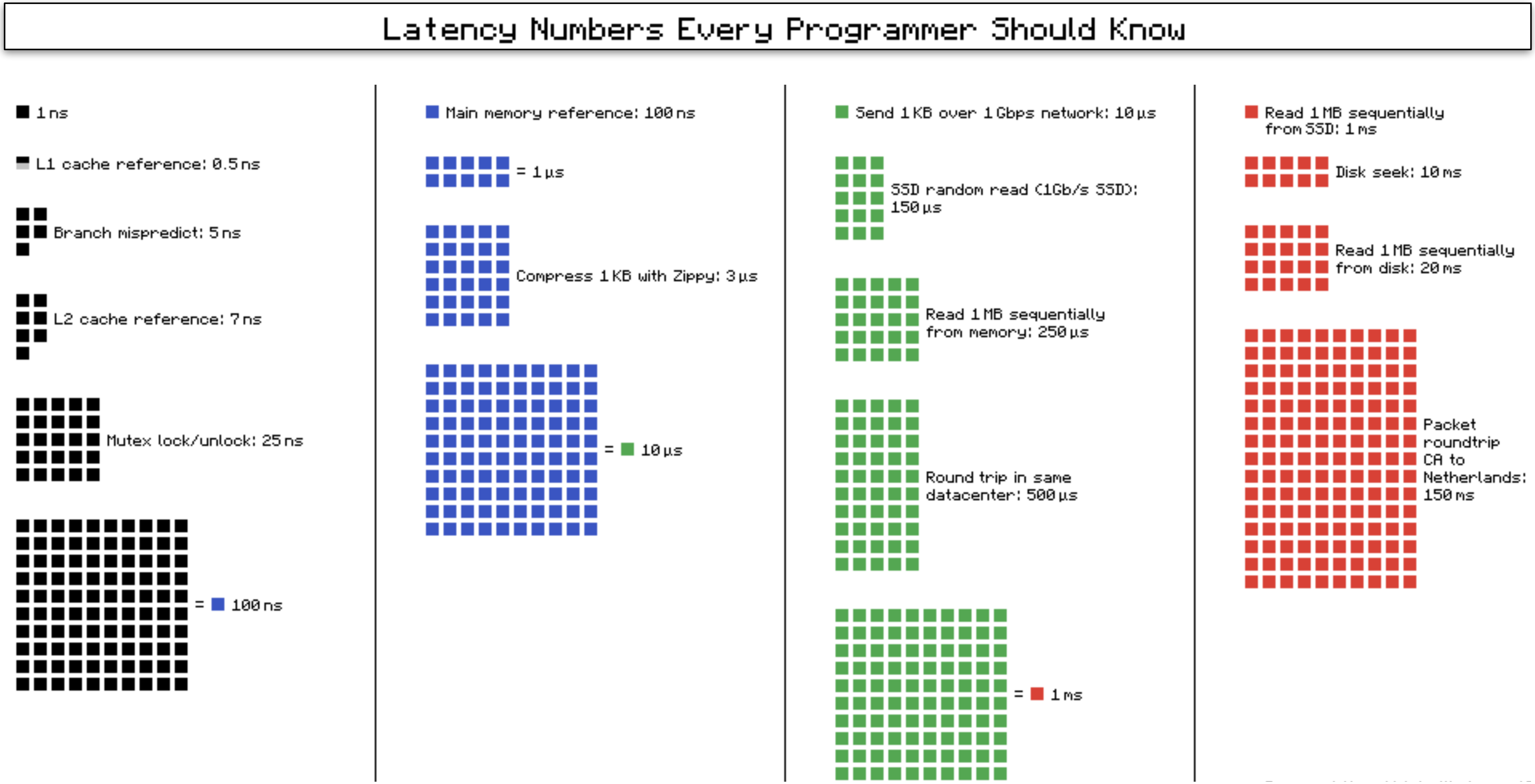
Source: https://gist.github.com/2841832

Life as an Application on an OS



Source: https://gist.github.com/2841832

Life as an Application on an OS



Source: https://gist.github.com/2841832

Conclusion

A hardware platform is only as useful as the software that is implemented on it

- enabling “good” software is just as important (or more important?) as enabling “good” hardware

The development toolchain is an important piece of this ecosystem

- interaction of low- and high-level languages
- interaction across abstraction layers
    - HW – OS – User applications