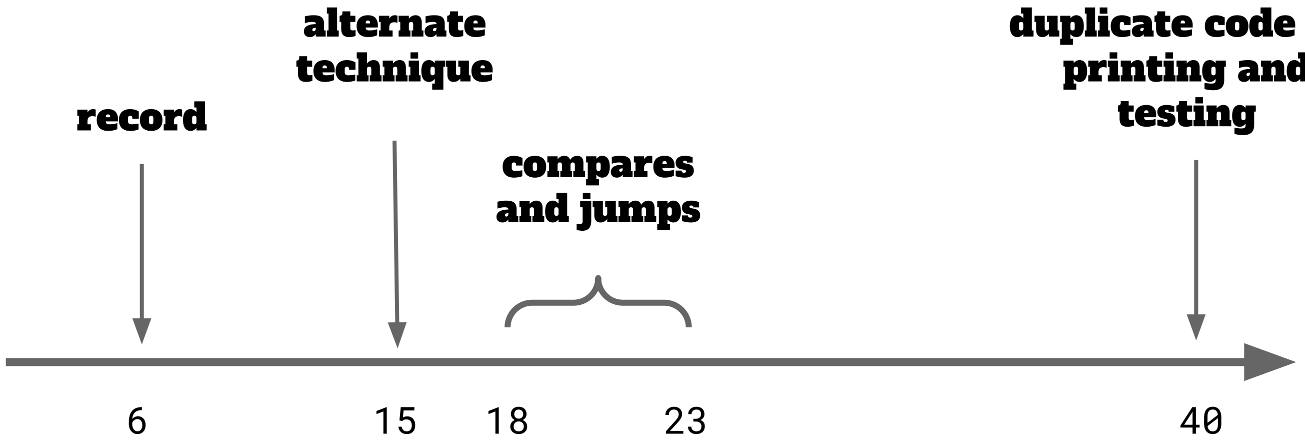
KEY - design

CSC 236

KEY program



lines of code written

Hundreds of lines

```
read c
if (c == '#')  {  }
.
if (c == '.')  { print c;  exit; }
.
if (c == 'A')  { print c; }
if (c == 'B')  { print c; }
.
if (c == 'a')  { c=c-20h; print c; }
if (c == 'b')  { c=c-20h; print c; }
.
.
.
```

- Unique test for each case
- It works!

Common

- Using compares
    - Common, obvious

```
if a ≤ char ≤ z
     char = char - 32
if A ≤ char ≤ Z
  or char == space
  or char == .
     print char
```

6 compares; 6 jumps

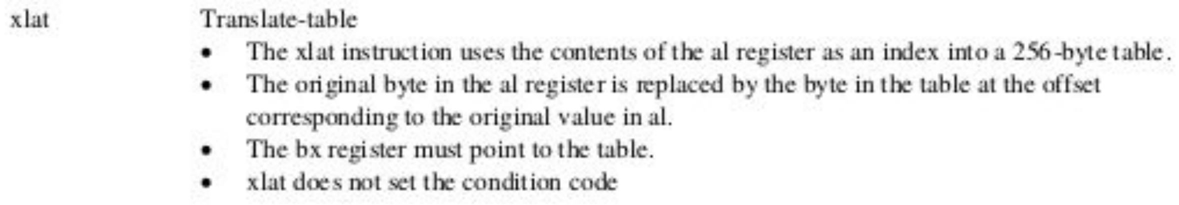
xlat

- Using compares
  - Common, obvious

```
if a ≤ char ≤ z
     char = char - 32
if A ≤ char ≤ Z
   or char == space
   or char == .
     print char
```

6 compares; 6 jumps

- xlat ⇒ Translate-table
  - al is index into a 256-byte table
  - al is replaced by the byte in the table at the offset
  - bx points to the table
  - Discussed pg 6-27

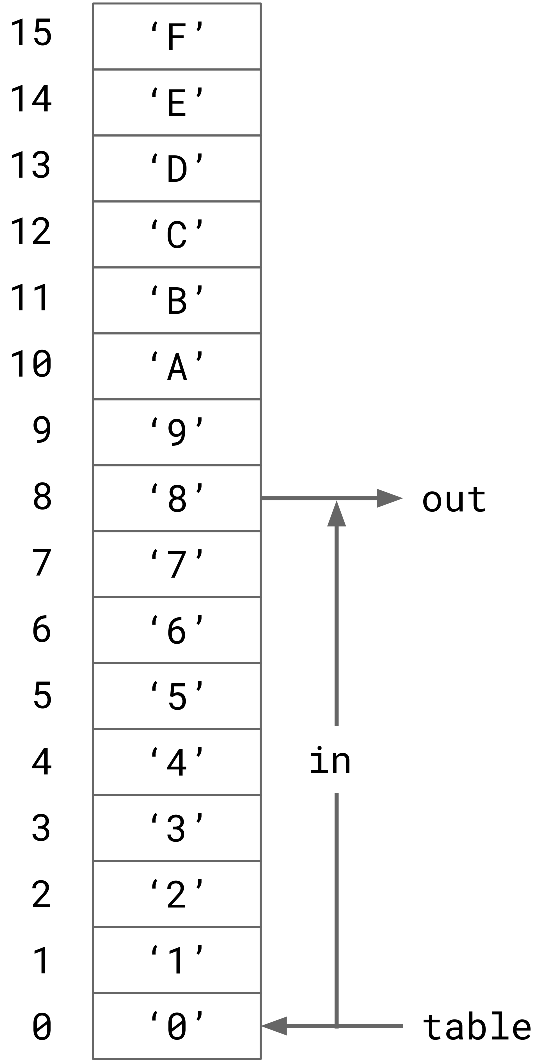


xlat Translate-table

- The xlat instruction uses the contents of the al register as an index into a 256-byte table.
- The original byte in the al register is replaced by the byte in the table at the offset corresponding to the original value in al.
- The bx register must point to the table.
- xlat does not set the condition code

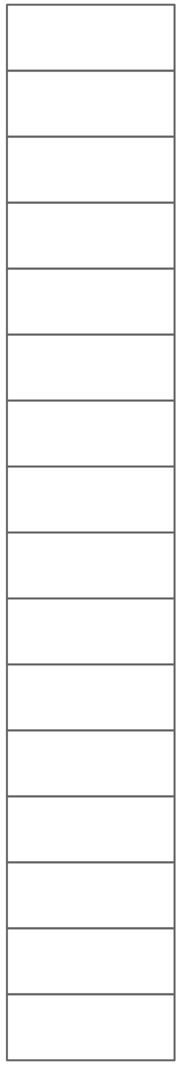
Translation table

- Example
  - `out = table[in]`
  - Char “in” is index into table
  - out = *(table + in)



Table

- 128 entries
  - al is index
  - Range 0 to 255
  - Why not 256 entries?
- ASCII
  - Range 0 to 127
  - Need error check (<128)
  - Not necessary in KEY

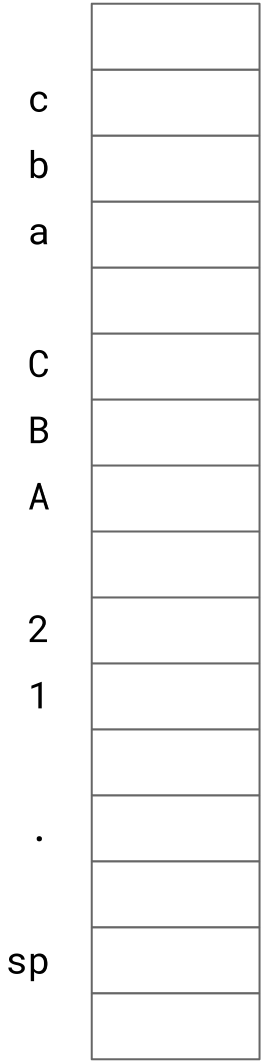


KEY with xlat

- Build the table
- What goes in the table?

Table

- Printable
- Non-printable



Table

- Printable
    - A-Z ⇒ self
    -
    -
    -
- Non-printable
    -

c	
b	
a	
C	C
B	B
A	A
2	
1	
.	
sp	

Table

- Printable
  - A-Z ⇒ self
  - a-z ⇒ to upper
  - <space> ⇒ <space>
  - . ⇒ .
- Non-printable
  -

c	C
b	B
a	A
C	C
B	B
A	A
2	
1	
.	.
sp	sp

Table

- Printable
  - A-Z ⇒ self
  - a-z ⇒ to upper
  - <space> ⇒ <space>
  - . ⇒ .
- Non-printable
  - Doesn’t matter
    use any value you want

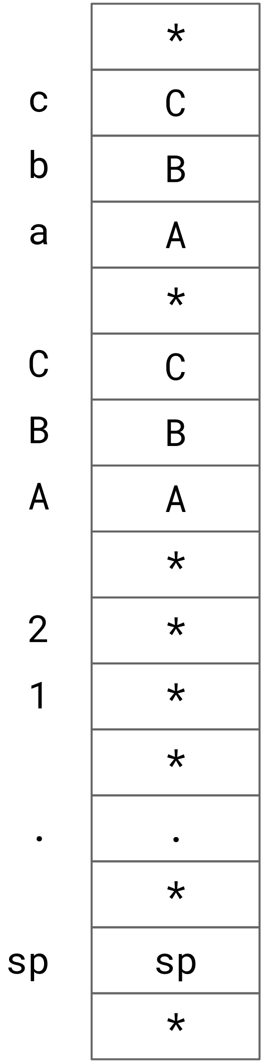
	*
c	C
b	B
a	A
	*
C	C
B	B
A	A
	*
2	*
1	*
	*
.	.
	*
sp	sp
	*

Data

- Create table in data segment

```
        .data
table   db '*…*  *…*.*…*A…Z*…*A…Z*…*',
```

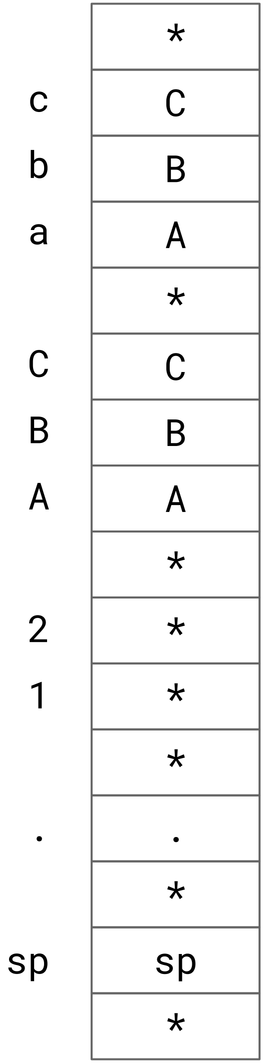
<space>



	*
c	C
b	B
a	A
	*
C	C
B	B
A	A
	*
2	*
1	*
	*
.	.
	*
sp	sp
	*

xlat

- xlat
  - al -- holds index
  - bx -- holds table base
  - Output char returned in al



Prepare

- xlat
  - al -- holds index
  - bx -- holds table base
  - Output char returned in al
- Outside of loop
  - Initialize bx
  - `mov  bx,offset table`

	*
c	C
b	B
a	A
	*
C	C
B	B
A	A
	*
2	*
1	*
	*
.	.
	*
sp	sp
	*

xlat

- xlat
  - al -- holds index
  - bx -- holds table base
  - Output char returned in al
- Loop
  - Read char into al (int 21h)
    - Returned in al (yeah!)
  - Execute xlat
    - `xlat`
    - No dest,source
  - If al != ‘*’ ⇒ print al
  - If al == ‘.’ ⇒ exit

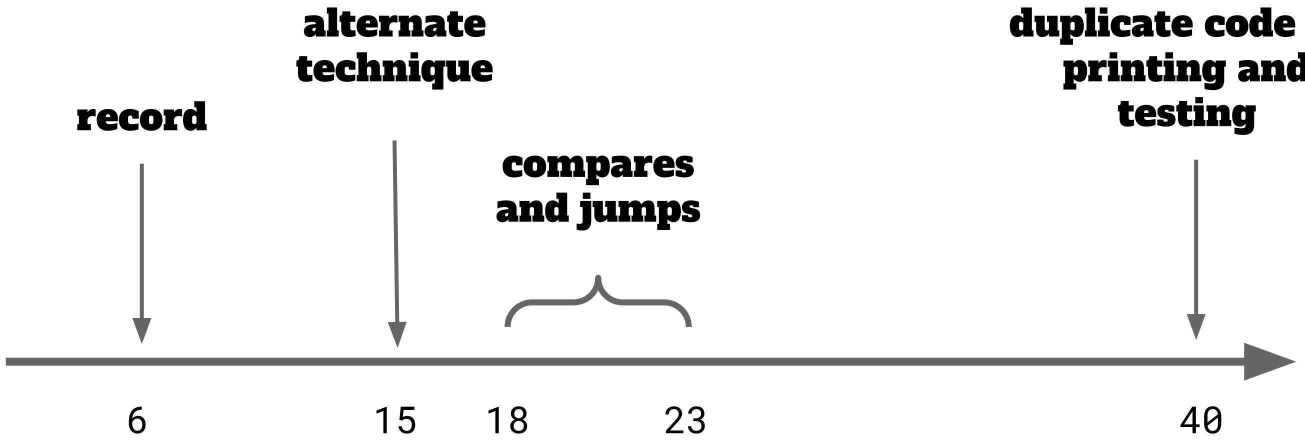
	*
c	C
b	B
a	A
	*
C	C
B	B
A	A
	*
2	*
1	*
	*
.	.
	*
sp	sp
	*

xlat

Code	Instructions
initialize	3
read char (int 21h)	2
translate	1
if char != ‘*’	2
write char (int 21h)	3
if char != ‘.’ loop	2
terminate (int 21h)	2
Total	15

	*
c	C
b	B
a	A
	*
C	C
B	B
A	A
	*
2	*
1	*
	*
.	.
	*
sp	sp
	*

Record program



lines of code written

New idea

- Only 1 int 21h
  - How?
  - Still need three different system calls
    - Read
    - Write
    - Terminate
- No compares
  - Still need to handle flow of control

Load code from table

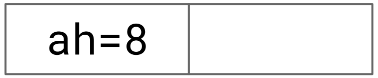
- Three int 21h
  - Hard coded in instruction
  - But it is a register
  - Instead load from table

Concept

- Not actual code
- Like a big finite state machine

Concept

- Set ah (for int 21h) to 8
  - Read char



Concept

- Set ah (for int 21h) to 8
  - Read char
- Issue int 21h

`ah=8`	`al='a'`

Concept

- Set ah (for int 21h) to 8
    - Read char
- Issue int 21h
- Use ah & al to get outchar

ah=8	al='a'

Concept

- Set al (for int 21h) to 8
  - Read char
- Issue int 21h
- Use ah & al to get outchar
  - `out = table[ah][al]`
- But also get next code (ah) from table
  - `out, code = table[ah][al]`
  - Put out in dl
  - Put code in ah

`ah=8`	`al='a'`

ah=8

	out	code
‘a’	‘A’	2
‘A’	‘A’	2
‘.’	‘.’	2

ah=8

	out	code
‘a’	‘A’	2
‘A’	‘A’	2
‘.’	‘.’	2

ah=2

	out	code
‘a’	‘A’	8
‘A’	‘A’	8
‘.’	?	?

ah=8

	out	code
‘a’	‘A’	2
‘A’	‘A’	2
‘.’	‘.’	2

ah=2

	out	code
‘a’	‘A’	8
‘A’	‘A’	8
‘.’	‘.’	4C

ah=8

	out	code
‘a’	‘A’	2
‘A’	‘A’	2
‘1’	?	?
‘.’	‘.’	2

ah=2

	out	code
‘a’	‘A’	8
‘A’	‘A’	8
‘1’	?	?
‘.’	‘.’	4C

ah=8

	out	code
‘a’	‘A’	2
‘A’	‘A’	2
‘1’	*	8
‘.’	‘.’	2

Right back to reading a char.

ah=2

	out	code
‘a’	‘A’	8
‘A’	‘A’	8
‘1’	*	*
‘.’	‘.’	4C

ah=8

	out	code
‘a’	‘A’	2
‘A’	‘A’	2
‘1’	*	8
‘.’	‘.’	2

Doesn’t matter.

ah=2

	out	code
‘a’	‘A’	8
‘A’	‘A’	8
‘1’	*	*
‘.’	‘.’	4C

Never reach this state.