The `cmp` Instruction

- The `cmp` instruction is used to compare values in registers with each other and with literal values.
- `cmp` works by subtracting the operands (without storing any result, so it is not destructive like an add, for example) and possibly setting the flags based on the result.
  - If the values are equal, the zero flag is set
  - If the second operand is larger than the first, the sign flag is set – the result of the subtraction is negative
  - If the first operand is larger than he second, no flag is set.

126

The `cmp` Instruction

- We can use the cmp instruction as a low-level implementation of the relational operators (==, !=, >, <, etc) of a high-level language.
- For example
  - Suppose we have
    - If (x<y) then do instr1 else do instr2
  - This might translate into

```
mov al, x
mov bl, y
cmp al, bl
js instr1_label ; if x<y cmp x,y will set the sign flag
jmp instr2_label
```

127

The `cmp` Instruction

☐ Write Run Log ☐ Log Assembler Activity

Source Code | List File | Configuration | Tokens | Run Log

```
; Example showing how the flags are affected by the cmp instruction
mov al, 10
cmp al, 10
cmp al, 20
cmp al, 05
end
```

128

Interacting with the External Environment

- The Samphire simulator uses the IN and OUT instructions to get input from, and to send output to, a specified *Port*.
- Thus, for example
  – IN 00

  Gets input from Port 00 (this port is linked to the keyboard in Samphire) and places it into the AL register (by design).
- Input coming from the keyboard will be ascii data.
- Thus, if the user presses the ‘0’ key in response to the IN 00 instruction, the value 30 will be placed into the AL register.

129

ASCII Table

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Interacting with the External Environment

- To reflect the value inputted from the keyboard back onto the VDU:

```
;Reading from the keyboard and writing to the VDU

start:
   mov bl, c0 ; start address of the VDU

loop:
   in 00     ; get input from the keyboard and place it into the al register
   mov [bl], al ; move the value in al to memory at address given in the bl register
   inc bl ; increment bl to point to the next VDU location
   jz start; jump to start if the result of inc bl is 0 - i.e., an address outside of the VDU
   jmp loop
end
```

Interacting with the External Environment

- Converting ASCII to integer
  - If we wish to work with numbers rather than characters, we simply subtract the ASCII value for ‘0’ (i.e., 30 hex) from the numeric character read from the keyboard.
  - Thus, if we press ‘1’ in response to IN 00, 31 will be placed into the AL register.
    - 31 – 30 = 1, i.e., the integer corresponding to the ASCII character ‘1’

132

Interacting with the External Environment

- The following code will add two numbers together and will put the result into the VDU.
- Note that it is assumed that the user will only press numeric characters and that the result of the addition is <=9

133

Interacting with the External Environment

```
;Reading numeric characters, converting them to integer, adding them and displaying
;the result in the VDU

in 00     ; get input from the keyboard and place it into the al register
sub al, 30 ; convert to integer
mov [80], al   ; copy the al register value to the bl register via the memory
mov bl, [80]   ; location 80. Note this is not good practice because, in general,  we
               ; cannot be sure if the location 80 is not being used,

sub al, 30; convert to integer

in 00      ; input a second numeric value
sub al, 30 ; convert to integer
add al, bl ; add to the value in bl
add al, 30 ; convert back to an ascii value
mov [c0], al ; write to th VDU
end
```

134

Interacting with the External Environment

- Using the stack to copy values between registers.
  - Rather than choosing random memory locations as temporary places to stores values being copied between registers, it is better practice to use the stack.
  - push al will put the contents of al onto the stack
  - pop bl will place the value on the top of the stack into the bl register.
  - The stack will be explained in more detail later.
  - Thus, an updated version of the previous code is:

135

Interacting with the External Environment

```
;Reading numeric characters, converting them to integer, adding them and displaying
;the result in the VDU

in 00     ; get input from the keyboard and place it into the al register
sub al, 30 ; convert to integer
push al
pop bl  ; copy al to bl via the stack

sub al, 30; convert to integer

in 00      ; input a second numeric value
sub al, 30 ; convert to integer
add al, bl ; add to the value in bl
add al, 30 ; convert back to an ascii value
mov [c0], al ; write to th VDU
end
```

136

Interacting with the External Environment

- To read and write multi-digit numbers, we must perform some arithmetic:
  - Suppose we wish to read the numeric characters ‘2’ ’3’ as the integer 23, we could proceed as follows:
    - Read the first character
    - Convert it to an integer
    - Multiple it by 10
    - Read the second character
    - Convert it to an integer
    - Add the first integer to the second integer

137

Interacting with the External Environment

– Conversely to write the integer 23 as the characters '2' '3', we could proceed as follows:
  - Divide a copy of the integer by 10 – this is the number of 10s in the integer
  - Convert it to a character by adding 30
  - Write it to the VDU
  - Use the modulus operator to get the remainder after dividing the original integer by 10 – this is the number of units in the integer
  - Convert it a character by adding 30
  - Write it to the VDU

138

Interacting with the External Environment

```
;Reading and writing multi digit numbers

in 00     ; get input from the keyboard and place it into the al register
sub al, 30 ; convert to integer
mul al, 0a ; 0a is the hex value for decimal 10
push al
pop bl  ; copy al to bl via the stack

in 00      ; input a second numeric value
sub al, 30 ; convert to integer
add al, bl ; add to the value in bl. bl now has the multidigit number

; write the number to the VDU

push al
pop bl   ;put a copy of the number in bl

div al, 0a ; get the number of 10s in the integer
add al, 30 ;  convert to character
mov[c0], al ; write to VDU

mod bl, 0a ; get hte number of units in the integer
add bl, 30 ; convert to character
mov [c1], bl ; write to VDU

end
```

139