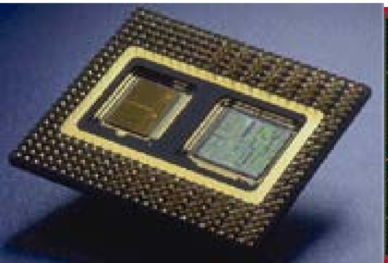
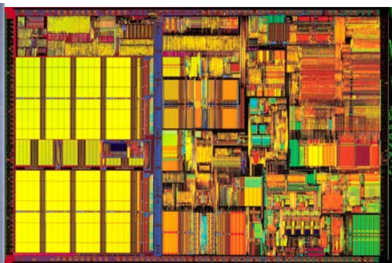
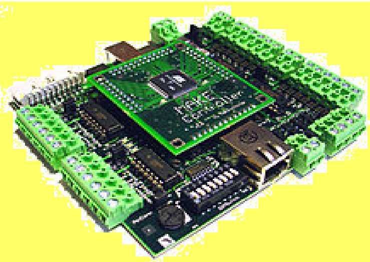
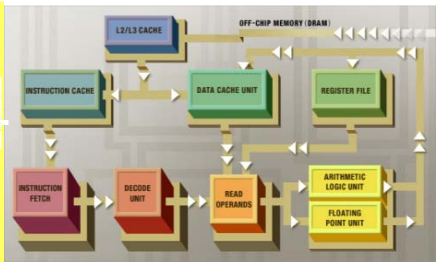
Lecture 8
Data Structures

Bryan Ackland
Department of Electrical and Computer Engineering
Stevens Institute of Technology
Hoboken, NJ 07030

Adapted from HCS12/9S12 An Introduction to Software and Hardware Interfacing Han-Way Huang, 2010

Data Structures

- A program consists of algorithms plus data structures
  - algorithm is sequence of operations required to produce result
  - data structures organize data to complement the algorithm
  - good data structures improve “transparency” of the code
- Programs we have written to-date have only operated on very small quantities of data
  - need data structures to manage complexity of total data space in a real application
- We will be examining:
  - arrays: index-able set of elements of same type
  - strings: sequence of characters terminated by a special character
  - stacks: first-in-last-out data structure

Arrays

- Arrays are index-able data structures made up of elements of same type and precision
- Arrays usually consist of a finite, predetermined number of elements
  - first element is often associated with index 0
  - e.g. we may want to create an 12 element array A of 16-bit signed integers. Each element in the array can be represented (conceptually) as $A[i] \;\; where\; 0 \le i \le 11$
  - a one-dimensional array is sometimes called a vector
- A two-dimensional array is an array of 1-D arrays
  - e.g. let B be a 6 element array of vectors, where each vector is itself an 10 element array of 8-bit integers.
  - B consists of 60 integers in all. Each element (integer) can be represented (conceptually) as $B[i][j] \;\; \text{where}\; 0 \le i \le 5, 0 \le j \le 9$
  - B[3][6] is an 8-bit integer
  - B[3] is an 10 element vector of 8-bit integers
  - a matrix is an example of a 2-D array

Declaring and Accessing Arrays

- Memory space for an array can be allocated using the DS and DC assembler directives, e.g:

  ABC: DS.B 8

  - allocates space for a 1-D array (vector) ABC of 8 elements (in this case bytes) without initializing the values in the array

  ABC: DC.B 13, 3, 4, 28, 19, 59, 100, 6

  - allocates space for a 1-D array (vector) ABC of 8 elements (in this case bytes) and also initializes the elements of the array

- Label ABC is the address of the first element (ABC[0])
- To access the element ABC[5]:

```
ldx     #ABC    ;load vector base address into X
ldaa    5, X    ;load contents of ABC[5] into A
```

- What if ABC was an array of 16-bit numbers?

Variable Indexing

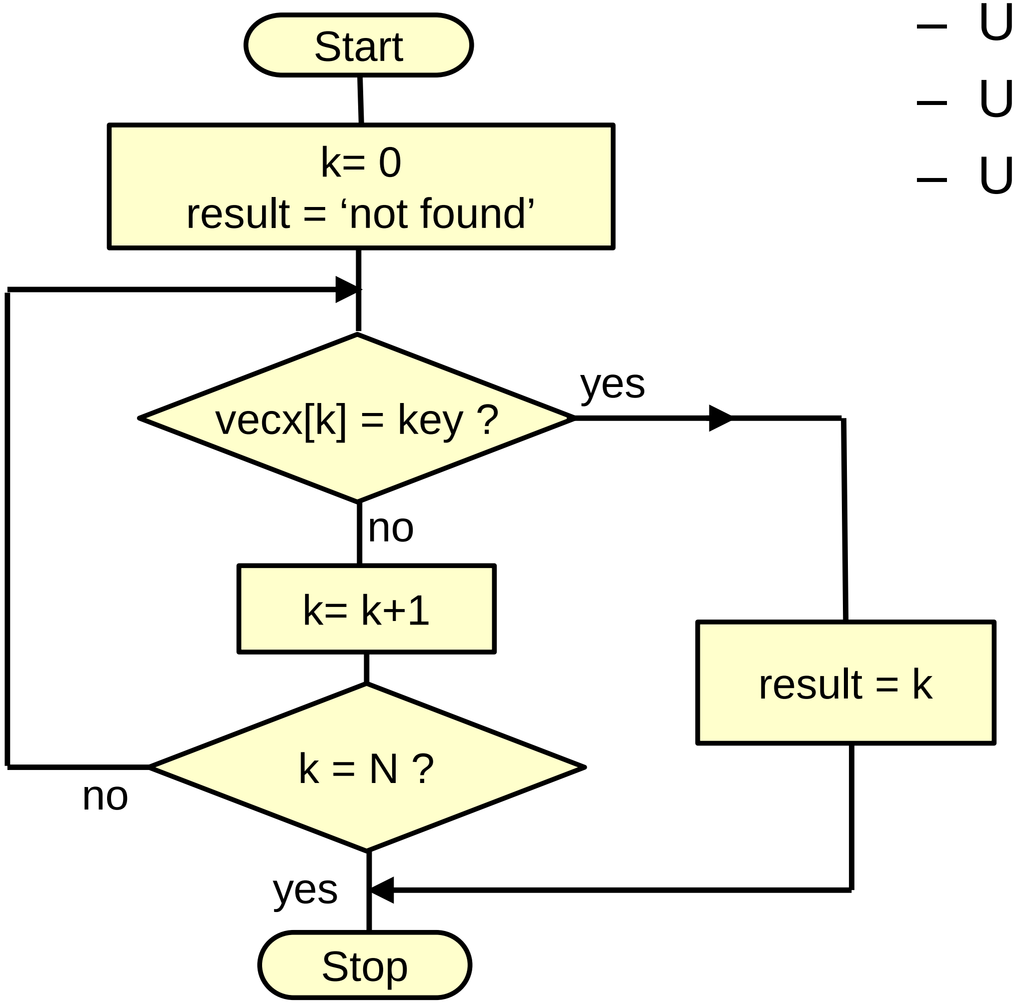
- What if the index into the array is a run-time variable
- For example, to access the element ABC[k], where k is an 8-bit value stored in memory location $1000:

```
ldx     #ABC    ; load vector base address into X
ldab    $1000   ; load value k into B
ldaa    B, X    ; load ABC[k] into A
```

- What if ABC was an array of 16 or 32-bit integers ?

Array Example: Sequential Access

- An array *vecx* consists of *N* 16-bit elements. Determine whether a particular 16-bit key is found in *vecx* and, if so, the index of its first occurrence.
  - Use Y to hold key
  - Use B to hold index k
  - Use X as pointer to array vecx



Array Example: Sequential Access (cont.)

```
N:          EQU     15                      ; length of array
notfound:   EQU     $FF                     ; $FF is code for "not found"
key:        EQU     190
            ORG     $800
result:     DS.B    1                       ; reserve a byte for result
vecx:       DC.W    13,15,320,980,42,86,130,319,430,4,190,20,18,55,30

            ORG     $1000
            clrb                            ; initialize index
            movb    #notfound, result       ; initialize search result
            ldy     #key                    ; key we're searching for
            ldx     #vecx                   ; set up pointer to array
loop:       tfr     B, A                    ; copy index to A
            lsla                            ; and multiply by 2 (byte offset)
            cpy     A, X                    ; compare key to array element
            beq     found
            incb                            ; increment index
            cmpb    #N                      ; are we at the end of the array?
            bne     loop                    ; no - continue
            bra     done                    ; yes – key not found
found:      stab    result                  ; store index of found key
done:       bgnd
```

Array Example: Random (indexed) Access

- An ordered array *vecq* consists of *N* unsigned 8-bit elements. The numbers are stored in increasing order. Use a binary search to determine whether a particular 8-bit key is found in *vecq* and, if so, the index of its occurrence.

Step 1: Initialize variables min and max to 0 and N-1 respectively

Step 2: If max < min then stop. No element matches key

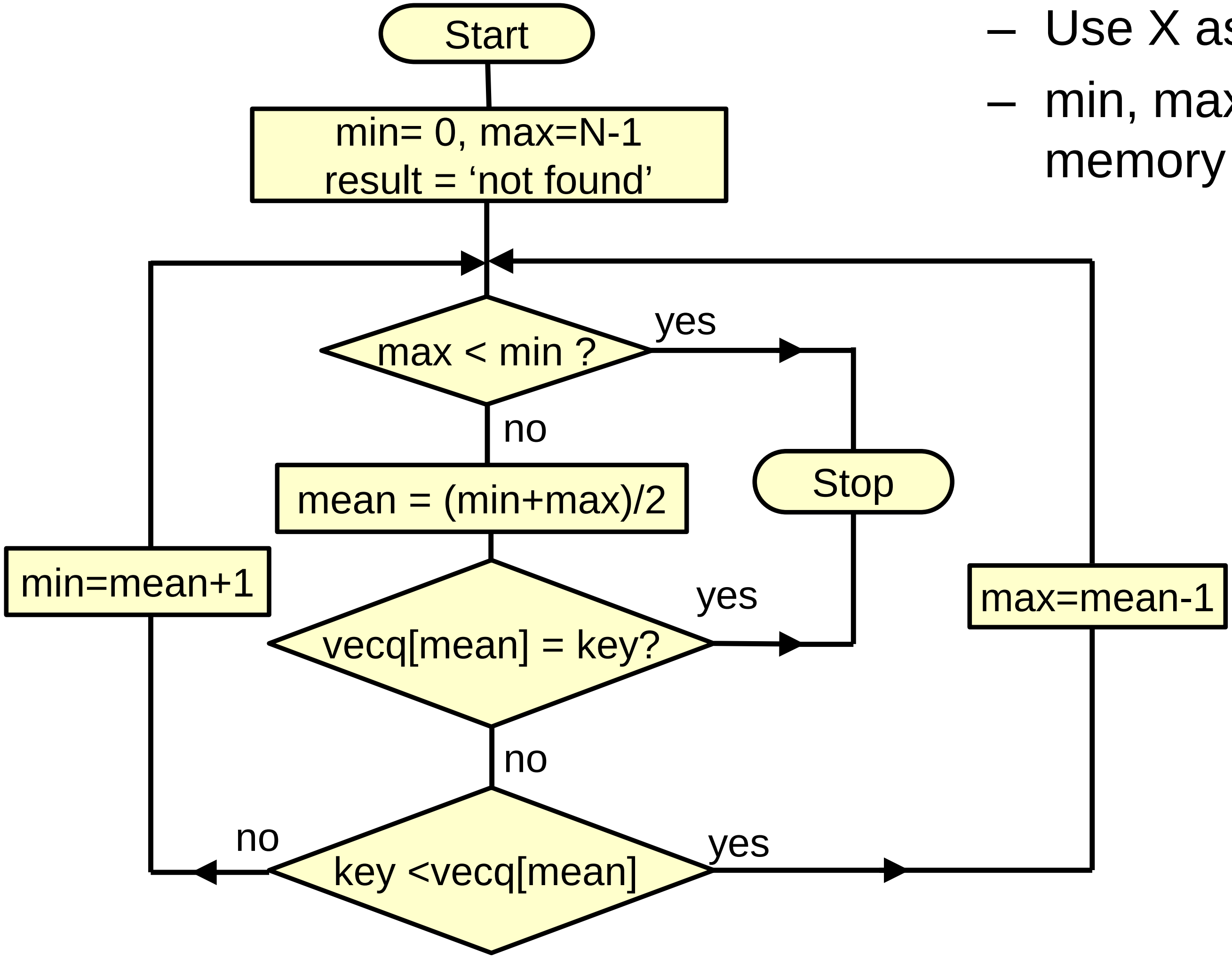
Step 3: Let mean = (min+max)/2

Step 4: If key = vecq[mean], then key is found, exit

Step 5: If key < vecq[mean] set max to (mean-1), go to step 2

Step 6: If key > vecq[mean] set min to (mean+1), go to step 2

Array Example: Random (indexed) Access



- Use B to hold mean
- Use X as pointer to array vecq
- min, max & result stored in memory

Random (indexed) Access (cont.)

```
N:          EQU     30                          ;length of array
key:        EQU     67                          ;key we're searching for
notfound:   EQU     $FF
            ORG     $800
min:        DS.B    1                           ;minimum index value
max:        DS.B    1                           ;maximum index value
result:     DS.B    1                           ;reserve a byte for index result
vecq:       DC.B    1,3,6,9,11,20,30,45,48,60,61,63,64,65,67
            DC.B    69,72,74,76,79,80,83,85,88,90,110,113,114,120,123

            ORG     $4000
            clr     min                         ;initialize min to 0
            movb    #N-1, max                   ;initialize max to N-1
            movb    #notfound, result           ;initialize result to 'not found'
            ldx     #vecq                       ;use X as pointer to array
```

Random (indexed) Access (cont.)

```
loop:   ldab    min
        cmpb    max
        bhi     knf         ;if min>max, then key not found
        addb    max         ;compute mean index
        lsrb                ;B=mean = (min+max)/2
        ldaa    b,x         ;get copy of vecq[mean]
        cmpa    #key        ;compare to key
        beq     found
        bhi     lower
upper:  incb
        stab    min         ;set min=mean+1
        bra     loop
lower:  decb
        stab    max         ;set max=mean-1
        bra     loop
found:  stab    result      ;result = current mean (index)
knf:    bgnd
        END
```

Strings

- A string is a data structure use to hold a sequence of characters
- Each character is represented using its 8-bit ascii code

LS Hex Digit

MS Hex Digit

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	DS	RS	US
2		!	“	#	$	%	&	‘	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Structure of Strings

- Strings are stored in consecutive memory (byte) locations
  - one character per byte
- A string is always terminated by a NULL ($00) character
  - strings are set up for sequential access
  - NULL character lets us know when we've reached the end
  - compare to arrays (know the array bounds and can use random access)
- In C, we might say: char str[] = "Hello, world"
  - when string is allocated, C compiler automatically adds NULL at end
- In assembly, the NULL must be explicitly added
  - e.g:

```
ORG    $800
DC.B   "Hello, world",0
```

$800	$801	$802	$803	$804	$805	$806	$807	$808	$809	$80A	$80B	$80C
$48	$65	$6C	$6C	$6F	$2C	$20	$77	$6F	$72	$6C	$64	$00
H	*e*	*l*	*l*	*o*	*,*		*w*	*o*	*r*	*l*	*d*	*NULL*

Strings Example:

- Convert an 8-bit unsigned number into its decimal ascii string suitable for sending to a printer. Suppress leading zeros.
- Solution:
  - up to 4 bytes are needed to represent result (including NULL)
  - divide by 100, then divide remainder by 10

```
          ORG     $5000
data:     DC.B    217
out_str:  ds.b    4              ; reserve 4 bytes for result

          ORG     $4000
          ldy     #out_str       ; Y is pointer to output string
          ldab    data           ; number to be converted into D = A:B
          clra
          ldx     #100
          idiv                   ; [D]/[X] → X, remainder D
          exg     X, D           ; Quotient into B
```

Strings Example (cont.)

```
        tstb                 ; check for zero
        beq     tens         ; suppress leading zero
        addb    #$30         ; convert remainder to ascii
        stab    1,Y+         ; store hundreds digit
tens:   tfr     X, D         ; restore remainder
        ldx     #10
        idiv                 ; determine tens digit
        exg     X, D         ; quotient into B
        tstb                 ; check for zero
        bne     skip         ; may need to suppress
        cmpy    #out_str     ; was hundreds zero suppressed?
        beq     units        ; suppress leading zero
skip:   addb    #$30         ; convert to ascii
        stab    1,Y+         ; store tens digit
units:  tfr     X, D         ; restore remainder
        addb    #$30         ; convert remainder to ascii
        stab    1,Y+         ; store units digit
        clr     0,Y          ; terminate with NULL
```

String Append Example:

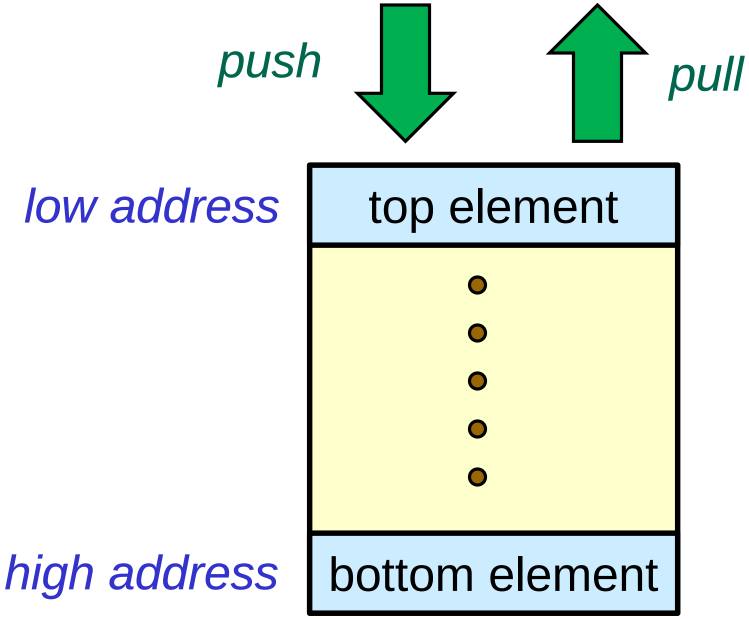
- Append *string2* to the end of *string1*

```
                ORG     $800
string1:        DC.B    "Happy  Birthday ",0
                ORG     $900
string2:        DC.B    "George",0

                ORG     $4000
                ldx     #string1        ; X points to string2
                ldy     #string2        ; Y points to string1
again:          ldaa    1,X+            ; test for NULL & increment pointer
                bne     again           ; reached end yet?
                decx                    ; set pointer back to NULL character
copy_loop:      ldaa    1,y+            ; get one character from string1
                staa    1,x+            ; add to end of string2
                bne     copy_loop       ; at end of string1 yet?
                bgnd
```

Stack

- Stack is a last-in-first-out (LIFO) data structure.
  - stack is a dynamic data structure – has a variable size
  - stack grows when new elements are added to the top of the stack
  - stack shrinks when existing elements are removed from top of stack

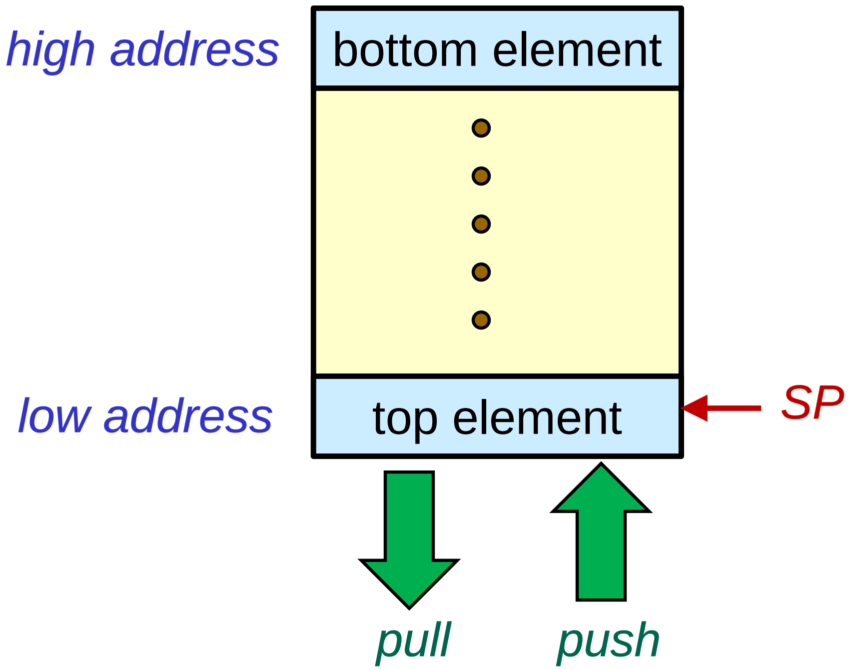


- The processor can add a new item to the stack by performing a push operation
- And remove an item from the stack using a pull (or pop) operation
- The stack is usually placed in a reserved area of RAM
  - usually at a high physical address
  - usually grows from high address down to low address

Stack

- We normally draw (think of) the stack as a data structure that grows downwards

- Stack pointer (SP) is a special register that points to the element on "the top" (lowest physical address) of the stack

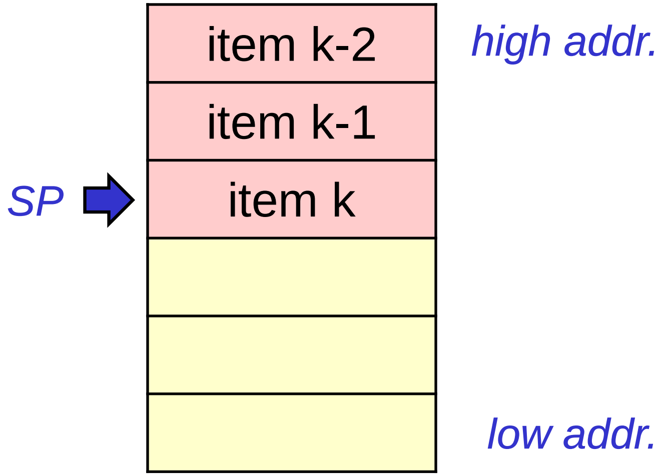


- When data is added (PUSH) or removed (PULL) the SP moves to reflect this change

- The SP can be used as an index register to access any data stored on the stack

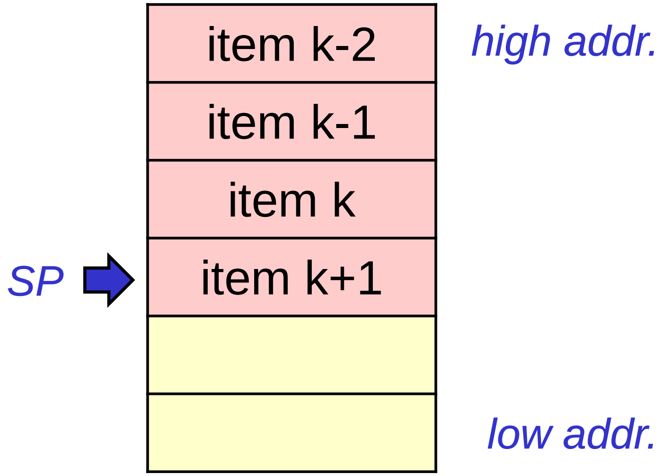
Stack PUSH

- On HCS12 (and most microprocessors), stack grows down from high addresses to lower addresses



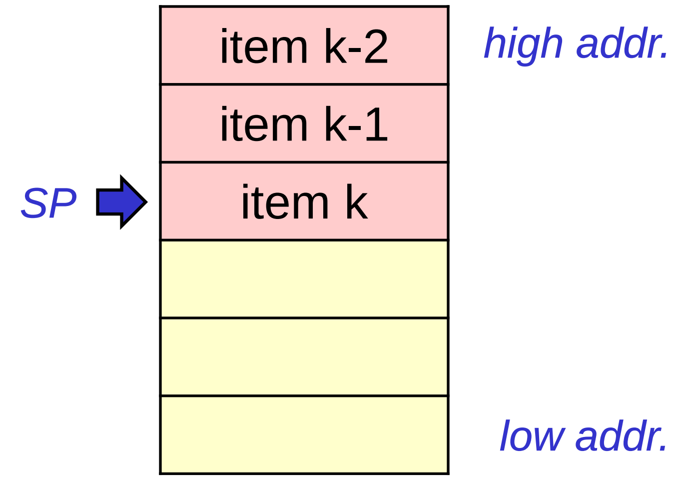
- Stack Pointer (SP) usually points to last element added

- A PUSH (data) operation adds new data to the stack. It does this by decrementing the stack pointer and then storing the new data at the location indexed by the SP

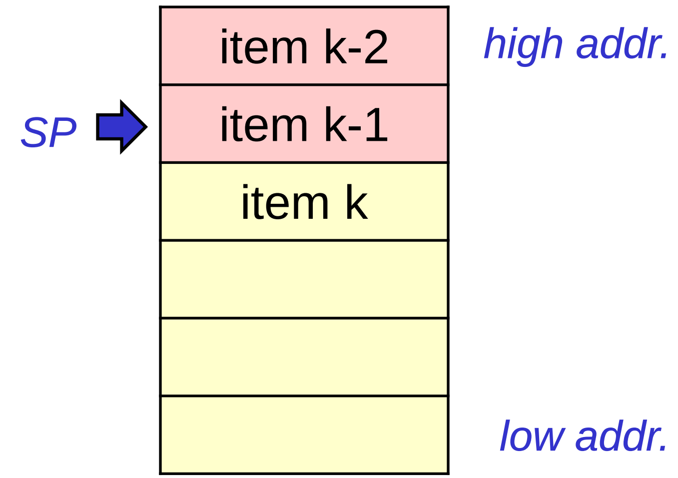


- SP will be decremented by either one or two depending on whether data is 8-bit or 16-bit

Stack PULL



- A PULL operation effectively removes data from the stack. It does this by loading data (to a register) from the memory location currently indexed by the SP and incrementing the SP



- SP will be incremented by either one or two depending on whether *pull'd* data is 8-bit or 16-bit
- After a PULL operation, the *pull'd* data will still be in memory but it is effectively removed from the stack because it is beyond the current value of the SP

Stack Instructions

Mnemonic	Function	Equivalent Instruction
psha	push A onto the stack	staa 1, –SP
pshb	push B onto the stack	staa 1, –SP
pshc	push CCR onto the stack	none
pshd	push D onto the stack	std 2, –SP
pshx	push X onto the stack	stx 2, –SP
pshy	push Y onto the stack	sty 2, –SP

Mnemonic	Function	Equivalent Instruction
pula	pull A from the stack	ldaa 1, SP+
pulb	pull B from the stack	ldaa 1, SP+
pulc	pull CCR from the stack	none
puld	pull D from the stack	ldd 2, SP+
pulx	pull X from the stack	ldx 2, SP+
puly	pull Y from the stack	ldy 2, SP+

Stack Implementation

- Stack is used to hold temporary data
- Stack is used to hold return address of subroutine call
- Stack can also be used to hold local variables
  - allows for dynamic allocation/release of memory space
  - # variables limited only by size of stack allocation region
  - stack data can be randomly accessed using SP as an index register
- Limited scope of access provides some data protection
- Stack hazards include:
  - overflow: pushing too much data on stack so that SP points to a location outside stack allocation region
  - underflow: pulling more data from the stack than had been previously pushed on to the stack.
- On Axiom CML-12C32 Development Board (used in lab), the stack is located in memory block $0E00 - $0E7F

Stack Example:

- What will be the contents of the stack after the execution of the following instructions?

```
lds     #$6000
ldaa    #$20
psha
ldab    #$40
pshb
ldx     #$1234
pula
pshx
pshx
puly
```

Stack as Temporary Storage

- Suppose in the middle of some algorithm, we need to divide data in D by 100 using **idiv** $(X \leftarrow [D] \div [X])$. But suppose also that we have some valuable data in register X.
  - We need to use register X, but we don't to lose the data in X.
- We could set up a special named memory location to temporarily store the data and then retrieve it after the divide:

```
stx     temp_x
ldx     #100
idiv
tfr     X, D
ldx     temp_x
```

requires us to deliberately allocate a named space and hold it available throughout the entire period of program execution

- Alternatively, we could just temporarily store it on the stack

```
pshx
ldx     #100
idiv
tfr     X, D
pulx
```

allows us to temporarily allocate space (on stack) and then release it when no longer needed – more efficient use of memory space