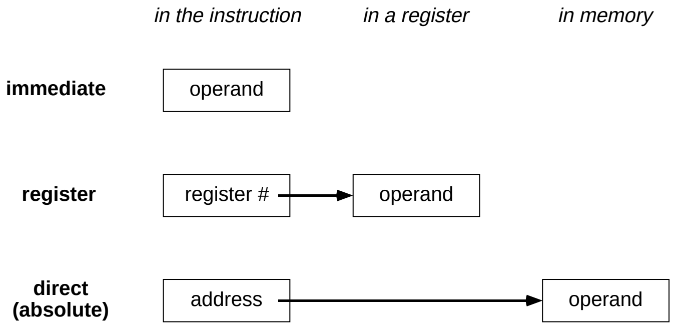
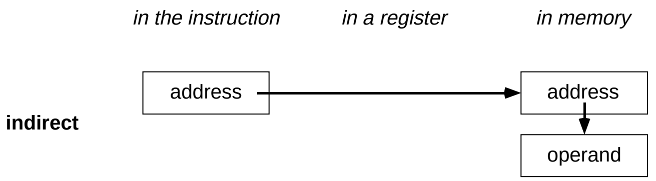
# Addressing Modes

- We have seen that instructions can, in general, refer to:
  - Immediate operands MOVE R1,#1
  - Operands in registers ADD R2,R3,R4
  - Operands in memory LOAD R3,100
- We can illustrate the three corresponding addressing modes as follows:

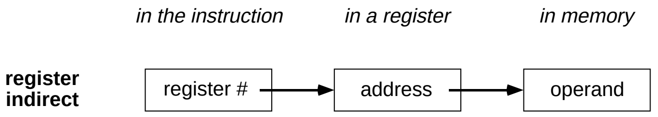


# Indirect and Register Indirect Addressing



- If you have an <u>address</u> stored in memory, you can access the memory location <u>pointed to by that address</u> using *indirect addressing* as shown above

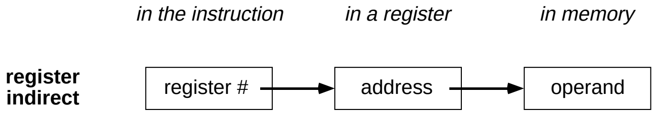
```
LOAD  R1,@100
```



- *Register indirect addressing*, shown above, is generally more efficient

```
LOAD  R1,@R2
```

# Implementing Pointers Using Register Indirect Addressing



- For example, consider the C code:

```
int x;      /* define a variable x */
int *px;    /* define a pointer (to x )*/
px = &x;    /* set px to point to x */
*px = 1;    /* store 1 in x via pointer */
```

- The assembly language translation, using register indirect addressing and a LOAD / STORE architecture, might be:

```
x   .reserve  4
    MOVE      R2,#x     ; R2 = px = &x
    MOVE      R3,#1
    STORE     @R2,R3    ; *px = 1
```

# Handling Arrays in Assembler (First Attempt)

- Consider the C code:

```
int a[100];    /* define an array*/
int i=40;      /* define an index */
a[i] = 50;     /* access the array*/
```

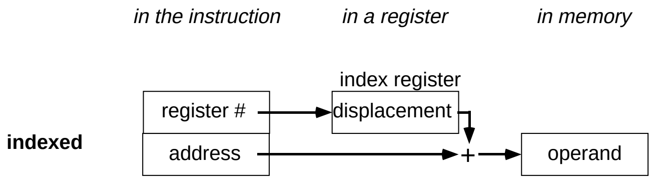
- In assembler, the variables could be defined in a .bss segment as follows, assuming an int takes 4 bytes:

```
a:  .reserve  100*4     ; int a[100]
i:  .word     40        ; int i = 40
```

- Then a[i] could be accessed as follows:

```
LOAD    R3,i        ; scale array index
MULT    R3,R3,#4    ;   (mult by 4)
MOVE    R4,#a       ; base of array
ADD     R3,R3,R4    ; address of a[i]
MOVE    R2,#50
STORE   @R3,R2      ; a[i]=50
```

# Handling Arrays in Assembler Using Indexed Addressing



- In indexed addressing, the instruction specifies a base address, and an index register specifies a displacement
  - Both are added together (by CPU) to produce the effective address

    LOAD R1,myarray[R2]

- Array access using indexed addressing

```
LOAD    R3,i        ; assumes an int
MULT    R3,R3,#4    ;   is 4 bytes wide
MOVE    R2,#50
STORE   a[R3],R2    ; a[i]=50
```

- Read Section 5.2, skipping 5.2.3 — 5.2.6

# Example — Working with Pointers

The C code:

```
int a[10], b[10];     /* store in memory */
int *ptra, *ptrb;     /* store in registers */

for (i=0 ; i<10 ; i++)          /* use indexed */
   a[i] = i;                    /* addressing */
```

The assembler code:

```
             .bss
a:           .reserve     10*4
b:           .reserve     10*4

             .text
             MOVE         R1,#0
test1:       BRLT         R1,#10,for1
             JUMP         endfor1
for1:        MPY          R2,R1,#4
             STORE        a[R2],R1
             ADD          R1,R1,#1
             JUMP         test1
endfor1:
```

# Example — Working with Pointers (cont.)

The C code:

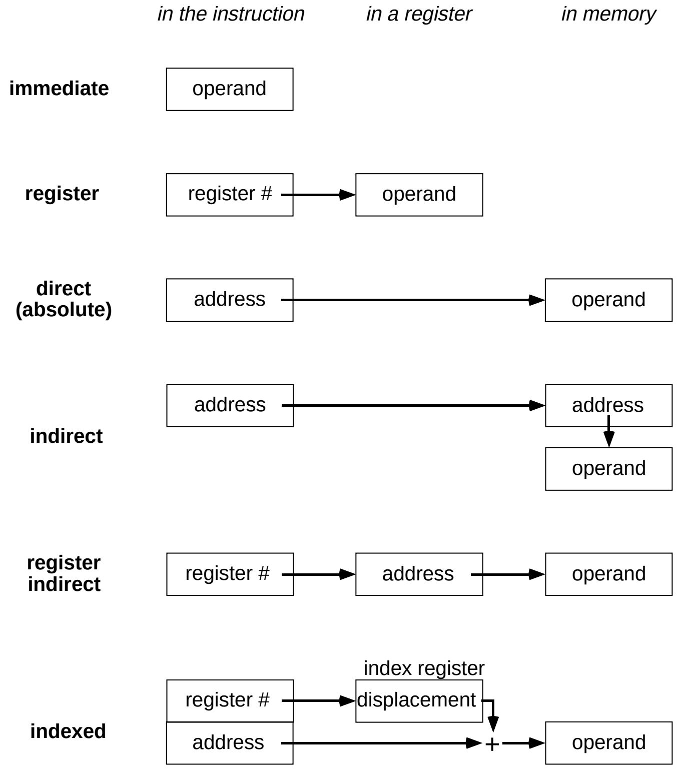
```
ptra = &a[0];
ptrb = &b[0];
for (i=1 ; i<=10 ; i++)       /* use register */
{                             /* indirect */
   *ptrb = *ptra;             /* addressing */
   ptra++; ptrb++
 }
```

The assembler code:

```
             MOVE        R2,#a   ; R2 = ptra
             MOVE        R3,#b   ; R3 = ptrb

             MOVE        R1,#1   ; R1 = i
test2:       BRLE        R1,#10,for2
             JUMP        endfor2
for2:        STORE       @R3,@R2
             ADD         R2,R2,#4
             ADD         R3,R3,#4
             ADD         R1,R1,#1
             JUMP        test2
endfor2:
```

# Common Addressing Modes



# Homework #4 — Due 10/26/98 (Part 3)

3. Consider the following sequence of instructions. For each instruction, tell me what it does (i.e., loads R3 with the value 100, loads R3 from memory location 100, etc.).

```
.equate   start   200
.equate   x       24

          LOAD    R1,#x
          LOAD    R2,x
          LOAD    R3,x*4
          LOAD    R4,start[R1]
          LOAD    R5,@R1
```

(This is the last question on Homework #4)