Imperial College London
Department of Computing

Computer Systems (113) / Architecture (110)
Exercises – *Pentium Addressing Modes*

1) Declare the following "global" variables/arrays using data declaration directives:

```
int  k, n           // assume integers are 32-bit   e.g. 2's-complement values
char chr                 // assume characters are 16-bit e.g. unicode characters
int  mark[10]
char name[10]
double  mean[10]    // assume doubles are 64-bits.
                    // use directive dq or resq for declaring quadwords
```

and then translate the following sequence of assignments into Pentium `mov` instructions. *Remember the source and destination of a Pentium* `mov` *instruction cannot both be memory operands*. Use registers as you wish in questions 1, 2 and 3 - for example to hold array subscripts.

```
k = mark[n]
chr = name[n]
mean[k] = mean[n]
```

2) Using data declaration directives declare `p` as a "global" person object:

```
class  person {
       int  mark[10]
       char name[10]
       double mean[10]
}

person p  //  declare p as one global object of the total size of the fields
```

and then translate the following sequence of assignments into Pentium `mov` instructions. You need not declare `k`, `n` and `chr` again.

```
k = p.mark[n]
chr = p.name[n]
p.mean[k] = p.mean[n]
```

3) Now declare `q` as a global array of person objects (assume Person is defined as in question 2):

```
person q[10]
```

and then translate the following sequence of assignments into Pentium instructions.

```
k = q[k].mark[n]
chr = q[k].name[n]
```

You can assume that the following instruction is available:

```
imul dest, src       ; which performs dest = dest * src
```

Imperial College London
Department of Computing

Computer Systems (113) / Architecture (110)
Solutions – *Pentium Addressing Modes*

1

```
k    resd   1              ; or   k    dd 0
n    resd   1              ; or   n    dd 0
chr  resw   1              ; or   chr  dw 0
mark resd   10             ; or   mark times 10 dd 0
name resw   10             ; or   name times 10 dw 0
mean resq   10             ; or   mean times 10 dq 0

; k = mark[n]
mov  eax, [n]              ; this will translate to mov eax, [address_of_n]
mov  ebx, [mark+4*eax]
mov  [k], ebx              ; this will translate to mov [address_of_k], ebx

; chr = name[n]            ; we'll assume that n is already in eax
mov  bx, [name+2*eax]      ; note:  we use bx not ebx, since the element is 16-bit
mov  [chr], bx

; mean[k] = mean[n]        ; note: quadword mov's are not allowed
                           ; we'll need to copy the 1st doubleword then the 2nd
mov  ebx, [mean+8*eax]
mov  ecx, [k]
mov  [mean+8*ecx], ebx
mov  ebx, [mean+8*eax+4]   ; now let's copy the 2nd doubleword. the displacement is
                           ; mean+4 here
mov  [mean+8*ecx+4], ebx
```

2 `k`, `n`, `chr` are declared as in Q1 above

```
p      resb 140            ; we could declare this as p resw 70 or p resd 35
       ; note mark has byte offset 0, name is at offset 40, mean is at offset 60

; k = p.mark[n]
mov  eax, [n]
mov  ebx, [p+4*eax]        ; since mark starts at the beginning of p we don't need
                           ; an extra offset
mov  [k], ebx

; chr = p.name[n]
mov  bx, [p+40+2*eax]      ; name starts 40 bytes from the start. note: the
                           ; assembler will add p and 40 to give the true
                           ; displacement
mov  [chr],  bx

; p.mean[k] = p.mean[n] ;
mov  ebx, [p+60+8*eax]     ; mean starts at byte offset 60 from the start of p
mov  ecx, [k]
mov  [p+60+8*ecx], ebx
mov  ebx, [p+60+8*eax+4]    ; +4 because the 2nd doubleword is 4-bytes after
                            ; the first
mov  [p+60+8*ecx+4], ebx
```

3 `k`, `n`, `chr` are declared as in Q1 above

```
q      resb 1400           ; since each person is 140 bytes, we need 1400 bytes for
                           ; 10 person objects

; k = q[k].mark[n]
mov  ebx, [k]              ; let's compute the byte offset of element k from the
                           ; beginning q first
imul ebx, 140              ; multiple by 140 since each person is 140 bytes in size
mov  eax, [n]
mov  ecx, [q+ebx+4*eax]    ; now copy q[k].mark[n] to ecx
mov  [k], ecx

; chr = q[k].name[n]
mov  ebx, [k]
imul ebx, 140
mov  eax, [n]
mov  cx,  [q+40+ebx+2*eax] ; name is 40 bytes from the start of a person object
                           ; and char's are 16-bit (hence cx and not ecx)
mov  [chr], cx
```