(20) Question 1. We usually try to divide last. \(\frac{x}{100}\) returns 0, 1, or 2. So \(\frac{(x/100) \times 49}{100}\) will be 0, 49, or 98.

\[ y = \frac{(49 \times x + 51 \times y)}{100}; \] // compiler will use 16-bit intermediate values

(20) Question 2. There is a potential overflow. If \(x\) is greater than 1337, then \(49 \times x\) will be greater than 65535. There is a similar problem with \(51 \times y\). If \(y\) is greater than 1285, then \(51 \times y\) will be greater than 65535. The solution is to use 32-bit intermediate calculations. In C, we cast the operations to \text{long}.

\[ y = \frac{(49 \times (\text{long}) x + 51 \times (\text{long}) y)}{100}; \]

The trouble with this is speed. All calculations take 32-bit inputs and generate 32-bit results. In assembly, we can perform a 16-bit by 16-bit multiply that generates a 32-bit product. The additions will be full 32-bit. But, the divide will be 32 bit dividend by 16-bit divisor yielding a 16 bit quotient. See the \text{emacs} and \text{edivs} instructions. This assembly implementation is \text{much} faster than the compiled version.

\begin{verbatim}
xn   ds    2      ; 12-bit raw data
yn   ds    2      ; 12-bit filter output
do.words 49,51 ; filter coefficients (16-bit)
sum  ds    4      ; intermediate sum=49*xn+51*yn (32-bit)
TCShandler
  movb  #$20,TFLG1 ; ack
  ldd   TC5
  addd  #8333
  std   TC5       ; fs=240 Hz
  movw  #0,sum    ; sum=0 (32-bit)
  movw  #0,sum+2
  ldx   #xn       ; pointer to xn,yn
  ldy   #coef     ; pointer to 49,51
  emacs  sum      ; sum = 49*xn
  emacs  sum      ; sum = 49*xn + 51*yn
  ldy   sum       ; Y:D = 49*xn + 51*yn
  ldd   sum+2
  ldx   #100
  edivs          ; Y=(49*xn+51*yn)/100
  sty   yn
  rti
\end{verbatim}

(20) Question 3. You can use the simple TCNT method to measure execution speed

```c
unsigned short errorcount; // number of lost data points
#pragma interrupt_handler TC5handler()
void TC5handler(void){
  unsigned short start,delay;
  TFLG1=0x20;      // ack C5F
  TC5=TC5+1000;    // fs=1000Hz
  start=TCNT;
  ProcessDAS();
  delay=TCNT-start;
  if((delay>1000)&&(errorcount<65535U))errorcount++
}
void ritual(void) {
  asm(" sei");     // make atomic
  errorcount=0;
  TIOS|=0x20;       // enable OCS
  TSCR|=0x80;       // enable
  TMSK2=0x33;       // 1 us clock
  TMSK1|=0x20;      // Arm output compare 5
  TFLG1=0x20;       // Initially clear C5F
  TC5=TCNT+1000;
  asm(" cli"); }
```
(40) **Question 4.** Select the gain to match the 5 V range of the ADC

<table>
<thead>
<tr>
<th>Minimum Voltage (mV)</th>
<th>Maximum Voltage (mV)</th>
<th>Resolution (mV)</th>
<th>Precision</th>
<th>Desired Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25.5</td>
<td>0.1</td>
<td>256</td>
<td>196</td>
</tr>
<tr>
<td>25.5</td>
<td>255</td>
<td>1</td>
<td>230</td>
<td>19.6</td>
</tr>
<tr>
<td>255</td>
<td>2550</td>
<td>10</td>
<td>230</td>
<td>1.96</td>
</tr>
</tbody>
</table>

(20) Part a) All three circuits use noninverting gain. The TLC2274 chip has 4 op amps and is powered by +5, and all resistors are 1% metal film.

\[
R_3 = \frac{R_1 \cdot R_2}{R_1 + R_2}
\]

\[
V_{out} = (1 + \frac{R_2}{R_1})V_{in}
\]

gain = 1.96

<table>
<thead>
<tr>
<th>Circuit</th>
<th>R1</th>
<th>R2</th>
<th>3</th>
<th>Vout</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAD0</td>
<td>10kΩ</td>
<td>9.6kΩ</td>
<td>4.9kΩ</td>
<td>gain = 1.96</td>
</tr>
<tr>
<td>PAD1</td>
<td>10kΩ</td>
<td>9.4kΩ</td>
<td>9.6kΩ</td>
<td>gain = 19.6</td>
</tr>
<tr>
<td>PAD2</td>
<td>5kΩ</td>
<td>975kΩ</td>
<td>975kΩ</td>
<td>gain = 196</td>
</tr>
</tbody>
</table>

(10) Part b) The ritual simply enables the ADC port.

```c
void InitializeDVM(void){
    ATDCTL2 = 0x80; // activate ADC
}
```

(10) Part c) The function uses MULT mode to sequentially measure all three channels. If the result is $FF$, then the gain of that channel is too high.

```c
void MeasureDVM(void){
    ATDCTL5 = 0x10; // start a scanned sequence of 4 samples on channels 0,1,2,3
    while((ATDSTAT&0x8000)==0){}; // wait for completion
    if(ADR2<255){
        voltage = ADR2;
        exponent = -1; // range 0 to 25mV
    } else if(ADR1<255){
        voltage = ADR1;
        exponent = 0; // range 25 to 255mV
    } else {
        voltage = ADR0;
        exponent = 1; // range 255 to 2550mV
    }
```