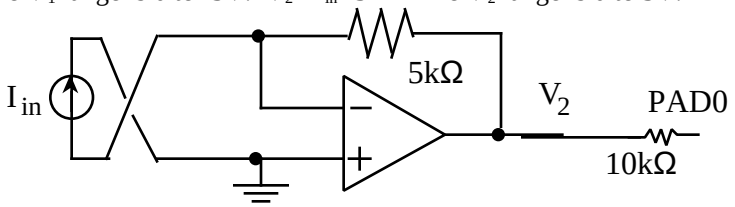
Jonathan W. Valvano May 16, 2000, 9 am to 12 noon

(25) Question 1. Because we are using an 8-bit ADC and a linear analog circuit, the measurement precision will also be 8 bits. To be linear, we need a linear current to voltage converter. Any op amp including the TLC2274 can be used.

Part a) The measurement resolution is 1000/256 = 3.9µA.

Part b) $V_1 = -I_{in} \bullet 5k\Omega$. The V_1 range is 0 to -5V. $V_2 = I_{in} \bullet 5k\Omega$. The V_2 range is 0 to 5V.



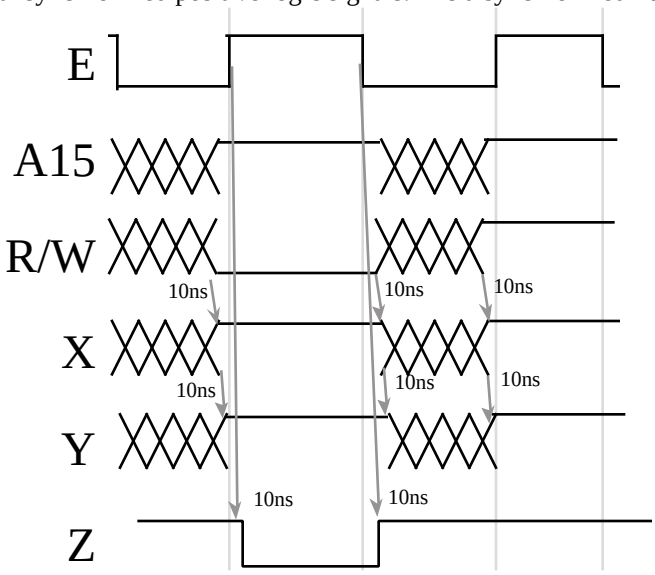
Part c) Show the ritual that initializes the ADC system (no interrupts, just simple one-time measurements)

```
void InitADC(void){
  ATDCTL2 = 0x80;  // Activate A/D
}
```

Part d) Note that 200*ADR0H will not overflow the 16-bit unsigned intermediate result (200*255=51000)

```
#define SCF 0x8000
unsigned short MeasureCurrent(void){ unsigned short current; // units in uA
  ATDCTL5 = 0;    // Start A/D channel 0
  while ((ATDSTAT & SCF) == 0){};  // if just ADR0H, could also spin on CCF0
  current=((200*ADR0H+25)/51);     // four good possibilities
  current=((50*(ADR0H+ADR1H+ADR2H+ADR3H)+25)/51);
  current=((125*ADR0H+16)/32);
  current=((125L*(long)(ADR0H+ADR1H+ADR2H+ADR3H)+64L)/128L);
  return(current); // result, 0 to 1000 uA
}
```

(10) Question 2. X and Y are unsynchronized positive logic signals. Z is a synchronized negative logic signal.



(15) Question 3. Part a) Write the ritual that initializes the system with a 1kHz 50% duty cycle waveform on PP0.

```
void InitPWM(void){
  PWCLK &= ~0x78; // CON01=0 Channel 0 and 1 are two separate 8-bit channels
  PWCLK |= 0x30;  // PCLKA=110 Clock A is divide by 64 or 8us or 125 kHz
  PWPOL &= ~0x10; // PCLK0=0 Clock A is clock source for Channel 0
  PWPOL |= 0x01;  // PPOL0=1 Channel 0 is high at the beginning
  PWEN  |= 0x01;  // enable PWM on channel 0
  PWCTL  = 0;     // CENTR=0, left aligned output
  PWPER0 = 124;   // period = 8us(PWPER0+1) = 1000us
  PWDTY0 = 62;    // duty cycle = (PWDTY0+1)/(PWPER0+1) is about 50%
}
```

Part b) Write a function that changes the duty cycle of the waveform on PP0.

```
void SetDuty(unsigned char duty){
  PWDTY0 = (duty*PWPER0)/256;    // duty cycle = (PWDTY0+1)/(PWPER0+1)
}
```

(20) Question 4. First, we calculate error

```
e(n) = xstar - x(n)
```

Next, we break the controller into proportional, integral, and derivative components

```
u(n) = p(n) + i(n) + d(n)
```

The proportional is straight forward, p=2.54•e = 127•e/50

```
p(n) = (127*e(n))/50
```

The integral term uses the discrete integration (123.2*1ms = 0.1232 sec = 77/625)

```
i(n) = i(n-1) + (77*e(n))/625
```

Anti-reset windup will limit the value of the `i(n)` to be between min < `i(n)` < max

The derivative term uses the discrete differentiation (0.00125/(6•1ms) = 0.208333 sec = 5/24)

```
d(n) = (5*(e(n)+ 3*e(n-1) -3*e(n-2) -e(n-3)))/24
```

The entire system must be bounded -500 < `u(n)` < 500

(20) Question 5. The key to designing a good instruction is to include operations that every OS needs, allowing for the widest range of OS configurations. It is important to execute the previously critical sections as an atomic sequence. These instructions basically implement the flowcharts shown in the book Figure 5.9. We don't want to perform too much of the OS code (e.g., blocking sequence, the structure of the TCB) because then the instructions could not be used for very many OS applications.

Part a) We need indirect indexed, but some other formats may be needed by other programmers

extended addressing mode

```
wait $0800  // semaphore at absolute address $0800
signal $0800
```

indexed addressing mode

```
wait 0,x    // x points to the semaphore
signal 0,x
```

indirect indexed addressing mode

```
wait [2,x]    // x+2 points to the address of the semaphore
signal [2,x]
```

Part b) The use of the software interrupt to make OS calls is very important. Notice that the flexibility is maintained by issuing a software interrupt, and letting the OS software perform the blocking function. `addr` is the effective address of the 16-bit semaphore. Here is the best answer

```
wait addr
        (addr) => value          // read 16-bit semaphore (atomic Read-Modify-Write)
        value-1 => value         // decrement semaphore
        value => (addr)          // write changed 16-bit value into memory
        if(value<0) {            // change-test sequence atomic
                push registers   // execute software interrupt, call OS (like SWI)
                I=1              // continues to be atomic
                ($FFDC)=>PC      // software interrupt vector
        }
signal addr
        (addr) => value          // read 16-bit semaphore (atomic Read-Modify-Write)
        value+1 => value         // increment semaphore
        value => (addr)          // write changed 16-bit value into memory
        if(value<=0) {           // change-test sequence atomic
                push registers   // execute software interrupt, call OS (like SWI)
                I=1              // continues to be atomic
                ($FFDA)=>PC      // software interrupt vector
        }
```

Here is another answer that doesn't use software interrupts. `addr` is the effective address of the 16-bit semaphore, and `branch` is the target location to jump to if the semaphore operation is successful. Assembly examples:

extended addressing mode

```
wait $0800,ok    // semaphore at absolute address $0800, branch if success
signal $0800,ok
```

indexed addressing mode

```
wait 0,x,ok    // semaphore pointed to by register X, branch if success
signal 0,x,ok
```

Pseudo code:

```
wait addr,branch
      (addr) => value          // read 16-bit semaphore (atomic Read-Modify-Write)
      value-1 => value         // decrement semaphore
      value => (addr)          // write changed 16-bit value into memory
      if(value>=0) branch=>PC  // branch if wait operation was successful
signal addr,branch
      (addr) => value          // read 16-bit semaphore (atomic Read-Modify-Write)
      value+1 => value         // increment semaphore
      value => (addr)          // write changed 16-bit value into memory
      if(value>0) branch=>PC   // branch if signal operation requires no more OS functions
```

Here is a third very simple, but complete answer (like the `inc` and `dec` instructions). Assembly examples:

extended addressing mode

```
wait $0800     // semaphore at absolute address $0800, V=0 if success
signal $0800
```

indexed addressing mode

```
wait 0,x     // semaphore pointed to by register X, V=0 if success
signal 0,x
```

Pseudo code:

```
wait addr
      (addr) => value          // read 16-bit semaphore (atomic Read-Modify-Write)
      value-1 => value         // decrement semaphore
      value => (addr)          // write changed 16-bit value into memory
      if(value>=0)
            V=0                // wait operation was successful
      else
            V=1                // wait operation was unsuccessful, need to block
signal addr
      (addr) => value          // read 16-bit semaphore (atomic Read-Modify-Write)
      value+1 => value         // increment semaphore
      value => (addr)          // write changed 16-bit value into memory
      if(value>0)
            V=0                // signal operation was successful, requires no more OS functions
      else
            V=1                // signal operation was unsuccessful, need to wakeup a thread
```

Part c) Similar to program 5.10 in the book. Again the software interrupt handlers will implement the blocking and unlocking of threads.

```
void Wait(short *semaphore){
asm("wait [2,x]\n"); }    // decrement, and software interrupt if blocked
void Signal(short *semaphore){
asm("signal [2,x]\n"); }    // increment, and software interrupt if a thread needed to wakeup
```

(10) Question 6. The 6812 is running in expanded mode with 512K of extended data page RAM.

Part a) The page number goes in the DPAGE register and the offset is added to $7000.

```
DPAGE = $19 ;
*((char *)( $7126 )) = 0;
```

Part b) The physical memory address of this byte is `$19126`

Part c) The CSD will handle it. The CSD will activate for the extended data RAM at physical address $0F000, but not for accesses to the internal EEPROM at $0F000?