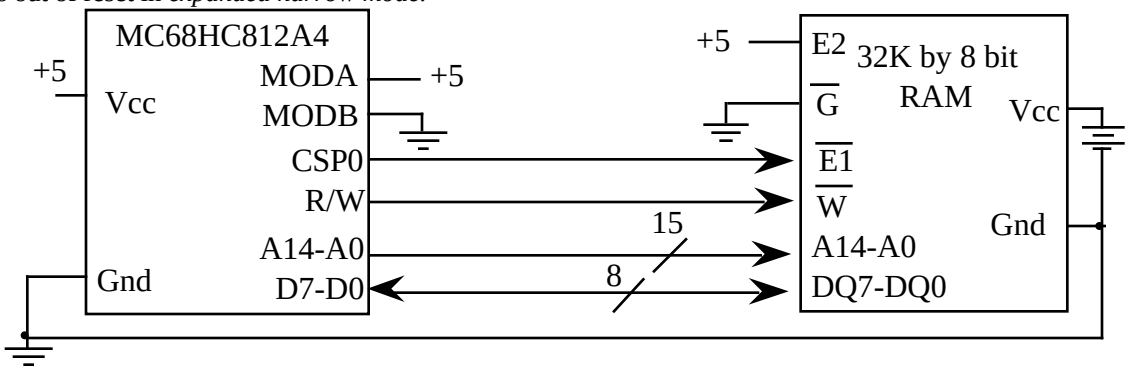
Jonathan W. Valvano First:_____ Last:_____
April 18, 2001, 9:00am-9:50am

This is an open book, open notes exam. You must put your answers on these pages only, you can use the back. You have 50 minutes, so please allocate your time accordingly. ***Please read the entire quiz before starting.***

(30) Question 1. You need more program space, but don't want the complexity of using EPROM (which requires external programming) or EEPROM (which also requires special hardware configuration for programming.) Instead, you choose to use a battery-backed SRAM. You will use 4 NiCad AA batteries to supply +5V power the RAM. You will use expanded narrow mode, which automatically puts the internal EEPROM at $1000 to $1FFF, and activates your CSP0 with 3 cycle stretches. Using CSP0 will automatically place your RAM at $8000 to $FFFF without requiring any software initialization. The jumper settings, MODA=1 and MODB=0, make the computer comes out of reset in *expanded narrow mode.*



Part a) Except for the number of address lines and the specific timing values, you may assume your 32k RAM is identical to the 60L64 8k RAM shown in section 9.7.2. You may also assume t_{AVQV} equals t_{E1LQV}. Assuming you will be running with 3 cycle stretches, what is the maximum (slowest) access time t_{AVQV} that can be used?

Part b) Again, assume 3 cycle stretches. What is the maximum (slowest) setup time t_{DVWH} that can be used?

(35) Question 2. Starting with the original Lab 17 files, you will develop a **Sleep** OS primitive. A thread will put itself to sleep for a fixed amount of time in **msec** when it doesn't need to run any more at this point. The basic idea is the thread calls `OS_Sleep` specifying how long it wishes to remain dormant. E.g., executing

```
OS_Sleep(2000);
```

will cause this thread not to be run in the next 2000 ms. At the end of the sleep interval, it is not guaranteed to run at that exact time, but rather its status is made active again, allowing it to be run. This concept is similar to the preemption-points added for cooperative multitasking. If fact, a preemption-point could be implemented simply by calling `OS_Sleep(0);` You may assume an unsigned 16-bit value called `SleepCounter` has been added to each TCB. When the `SleepCounter` is zero, the thread can be run. A nonzero value specifies the time in ms until the thread awakes. Also, you may assume the `OS_AddThread` function will initialize this value to 0. You will need an additional periodic interrupt process using output compare 0 that implements the sleep waiting.

Part a) Show the implementation of the `OS_Sleep` function. Calling this function will set the `SleepCounter` value in the TCB and cause a preemption (this thread stops running). The prototype is

```
void OS_Sleep(unsigned short delay);
```

Part b) Edit the existing `threadSwitch` into the new system, which will not run any thread with a nonzero `SleepCounter`. You may assume there is at least one thread that never blocks or sleeps.

```
void threadSwitch(void){ // do most of the work here


  RunPt=RunPt->Next;


  PORTJ=RunPt->Id;     /* PortJ shows which thread is running */
}
#pragma interrupt_handler threadSwitchISR()
void threadSwitchISR(void){
asm(" ldx _RunPt\n"
    " sts 2,x");
  threadSwitch();
  TC3 = TCNT+TimeSlice;  // Thread runs for a unit of time
  TFLG1 = 0x08;          // acknowledge by clearing TC3F
asm(" ldx _RunPt\n"
    " lds 2,x");
}
unsigned int NumThread=0; // number of threads
```

Part c) A lot of the OC0 software is given. You will complete the output compare 0 ISR. Once every ms, this ISR will decrement the `SleepCounter` for all threads with a nonzero `SleepCounter`.

```
void SleepInit(void){ // TCNT runs at 125 ns
  TIOS  |= 0x01;      // enable output compare 0
  TMSK1 |= 0x01;      // arm OC0
  TC0 = TCNT + 50;    // first one, right away
  TFLG1  = 0x01;      // initially clear C0F
}
#pragma interrupt_handler OC0Handler
void OC0Handler(void){
  TC0 = TC0 + 8000;   // interrupt every 1 ms
  TFLG = 0x01;        // acknowledge interrupt by clearing C0F
```

```
}
extern void OC0Handler();
#pragma abs_address:0xffee
void (*OC0_interrupt_vector[])() = { OC0Handler};
#pragma end_abs_address
```

(35) Question 3. The Analog Devices DAC8043 12-bit DAC is interfaced to the 6812. The details can be found on pages 405-407 and 632-633 in the book. The DAC output range is -5 to +5 V. The DAC uses offset binary, converting 0 to 4095 into -5 to +5V. E.g., a binary value of 2048 is converted to 0 V. Also, a binary value of 1638 is converted to -1 V. Other values can be found in Table 11.14 on page 633.
Part a) Assuming a low noise environment, what is the DAC resolution, ΔV?

Part b) You will develop a 16-bit signed binary fixed-point format to represent the analog signals. Let $\Delta=2^n$ be the binary fixed-point constant value. Choose the largest n such that $\Delta V \geq \Delta$.

Part c) Explain your format showing how -1 V would be stored in your software.

Part d) Develop an equation that converts a signed binary fixed-point number, label it `binaryData`, into a 12-bit offset binary value, label it `dacData`, which can be sent to the DAC. In particular, this equation should convert your answer in part c) into 1638. Also, this equation should convert 0 into 2048.

Part e) The following is essentially program 7.20 found on page 407. Add the C implementation of your equation.

```
#define SPIF 0x80
void DACout(short binaryData){
unsigned short dacData;
unsigned char dummy;
```

```
  SP0DR = 0x00FF&(dacData>>8); // msbyte
  while((SP0SR&SPIF)==0);      // gadfly wait
  dummy = SP0DR;               // clear SPIF
  SP0DR = 0x00FF&dacData;      // lsbyte
  while((SP0SR&SPIF)==0);      // gadfly wait
  dummy = SP0DR;               // clear SPIF
  PORTS &= ~0x80;              // PS7=LD=0
  PORTS |= 0x80; }             // PS7=LD=1
```