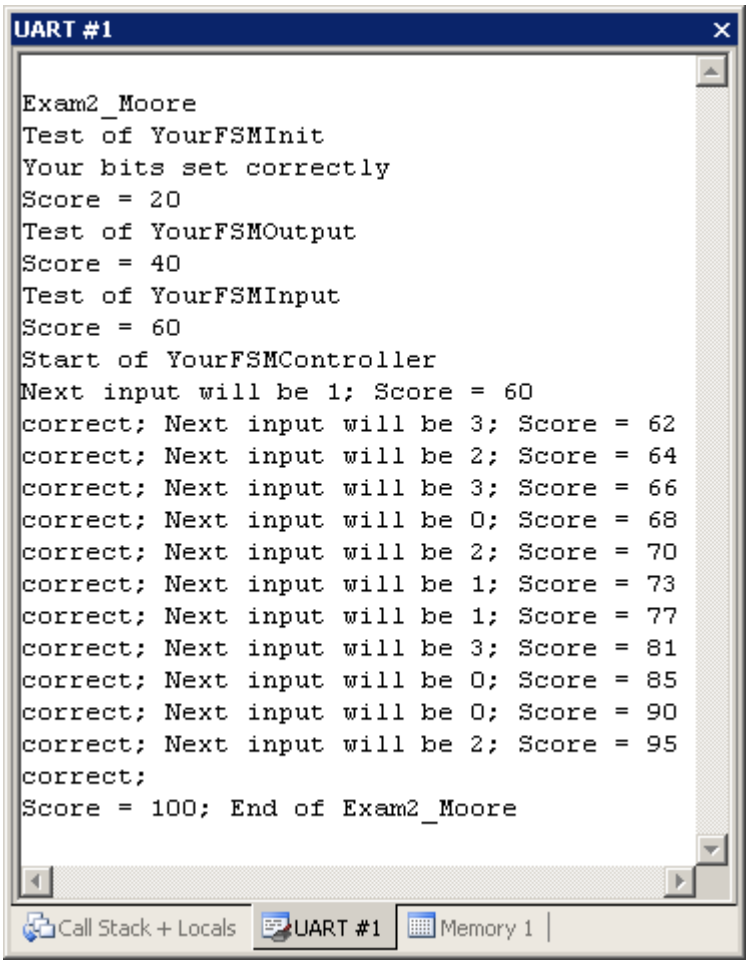
First:_____ Last:_____

Scoring The correct output values are shown in the figure on the right.
Your grade will be based both on the numerical results returned by your program and on your programming style. In particular, write code that is easy to understand, easy to debug, easy to change. Please employ good labels, pretty structure, and good comments.

Performance Score= Run by TA at the checkout		TA:

I promise to follow these rules

This is a closed book exam. You must develop the software solution using the **Keil uVision** simulator. You have 35 minutes, so allocate your time accordingly. You are allowed to bring only some pencils (no books, laptops, cell phones, hats, disks, CDs, or notes). You will have to leave other materials up front. Each person works alone (no groups). You have full access to **Keil uVision**, with the **Keil uVision** help. You may use the Window's calculator. You sit in front of a computer and edit/assemble/run/debug the programming assignment. You do NOT have access the book, internet or manuals. You may not take this paper, scratch paper, or rough drafts out of the room. You may not access your network drive or the internet. You are not allowed to discuss this exam with other EE319K students until Thursday.



The following activities occurring during the exam will be considered scholastic dishonesty:

1) running any program from the PC other than **Keil uVision**, or a calculator,
2) communicating with other students by any means about this exam until Thursday,
3) using material/equipment other than a pen/pencil.

Students caught cheating will be turned to the Dean of Students.

Signed: _____ October 31, 2012

Procedure

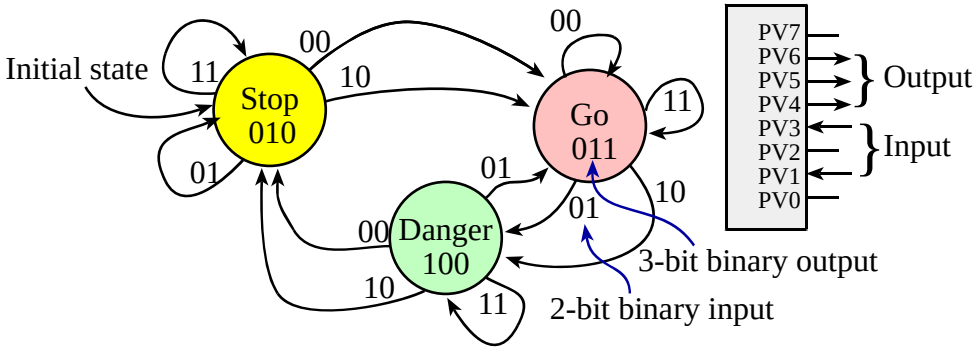
First, you will log onto the computer and download files from the web as instructed by the TAs.

Web site xxxxxx
user: xxxxxx
password: xxxxxx

Unzip the folder placing it in a temporary folder. You are not allowed to archive this exam. Within **Keil uVision** open these files, put your name on the first comment line of the file **Exam2.s**. Before writing any code, please assemble and run the system. You should get output like the figure above (but a much lower score). You may wish create backup versions of your program. If you wish to roll back to a previous version, simply open one of the backup versions.

My main program will call your subroutines multiple times, and will give your solution a performance score of 0 to 100. *You should not modify my main program or my example data.* When you have written your subroutines, you should run my main program, which will output the results to the **`UART#1`** window. After you are finished, raise your hand and wait for a TA. The TA will direct you on how to complete the submission formalities. The TA will run your program in front of you and record your performance score on your exam cover sheet. The scoring page will not be returned to you. The submission guidelines are at the end of the exam handout.

You will write three subroutines and one FSM controller. My main program will call your subroutines multiple times, giving you a grade on these subroutines. If these subroutines operate properly, my program will jump to your FSM controller. During each loop of your FSM controller, you will call my grader subroutine and points will be awarded. When my grader subroutine is done testing your FSM controller it will output a performance score of 0 to 100. *You should not modify my main program.* When you have written the first subroutine, you should run the system, which will output the results to the **`UART#1`** window. The inputs are on Port V bits 3 and 1, and the outputs are on Port V bits 6, 5, and 4. For example, if you are in the **Stop** state the output is 010_2. If the input is 10_2, then change to the **Go** state. The input/output values are shown in binary. The input/output testing will be performed by my grading software.



Part a) Your first subroutine, called **`YourFSMInit`**, will initialize the FSM. This subroutine should initialize the I/O port. Bits 6, 5 and 4 are the output; bits 3 and 1 are the input. All accesses to I/O port registers must be friendly. Bits 7, 2 and 0 are not part of the FSM and should not be changed.

Part b) Your second subroutine, called **`YourFSMOutput`**, will output to Port V bits 6, 5 and 4. All accesses to I/O port registers must be friendly. The value to output (0 to 7) is passed in Register R0. When writing to Port V you must write back the original values for the input pins, bits 3 and 1. E.g., if bit 3 is a 1, do not write a 0 to this bit. Bits 7, 2 and 0 are not part of the FSM and should not be changed. This is not a real I/O, so there is no bit-specific addressing allowed.

Part c) Your third subroutine, called **`YourFSMInput`**, will input from Port V bits 3 and 1. All accesses to I/O port registers must be friendly. The input from Port V (bits 3 and 1) is returned as a 2-bit number in Register R0.

```
if PV3=0 and PV1=0, then return Register R0 equal to 0
if PV3=0 and PV1=1, then return Register R0 equal to 1
```

```
if PV3=1 and PV1=0, then return Register R0 equal to 2
if PV3=1 and PV1=1, then return Register R0 equal to 3
```

Part d) Finally, you will write a controller for a Moore FSM. *Your controller should call your three subroutines.* All I/O accesses should be performed in a friendly manner. Your FSM controller does not execute **`BX LR`**. My grader program will stop your FSM controller if the direction bits on any of the two inputs or three outputs are incorrect. There are three parts to this FSM controller. Convert the Moore FSM graph to a linked data structure (a graph) and store it in EEPROM. You will need to save a pointer to the current state in memory because the **`MyGrader`** program may change the registers. The initial state is **Stop**. The proper sequence is output, input, and go to next state. Place your FSM controller software between **`BL MyGrader`** and **`B loop`**. Notice that the grader subroutine must be called before you execute the output, input, next sequence. Your FSM controller will follow this sequence.

```
;  put your graph data structure here
YourFSMController
     BL YourFSMInit ;your initialization
loop
     BL  MyGrader ;do not move or remove this line
; Part d) put your output-input-next engine here
; you must use a global variable for the state pointer
;the Grader will check outputs and make inputs happen
;1. output to PTV bits 6,4,3 (be friendly)
;2. input from PTV bits 3,1
;3. next


     B  loop    ;do not remove this line
```

Note that calling your subroutine in part (a) in part (b) will greatly reduce the amount of code you will need to write.

<u>Submission Guidelines</u>:

- Log onto Blackboard and submit your Exam2.s source file into the Exam2 field. Be careful because only one submission will be allowed.

Memory access instructions

```
LDR    Rd, [Rn]        ; load 32-bit number at [Rn] to Rd
LDR    Rd, [Rn,#off]   ; load 32-bit number at [Rn+off] to Rd
LDR    Rd, =value      ; set Rd equal to any 32-bit value (PC rel)
LDRH   Rd, [Rn]        ; load unsigned 16-bit at [Rn] to Rd
LDRH   Rd, [Rn,#off]   ; load unsigned 16-bit at [Rn+off] to Rd
LDRSH  Rd, [Rn]        ; load signed 16-bit at [Rn] to Rd
LDRSH  Rd, [Rn,#off]   ; load signed 16-bit at [Rn+off] to Rd
LDRB   Rd, [Rn]        ; load unsigned 8-bit at [Rn] to Rd
LDRB   Rd, [Rn,#off]   ; load unsigned 8-bit at [Rn+off] to Rd
LDRSB  Rd, [Rn]        ; load signed 8-bit at [Rn] to Rd
LDRSB  Rd, [Rn,#off]   ; load signed 8-bit at [Rn+off] to Rd
STR    Rt, [Rn]        ; store 32-bit Rt to [Rn]
STR    Rt, [Rn,#off]   ; store 32-bit Rt to [Rn+off]
STRH   Rt, [Rn]        ; store least sig. 16-bit Rt to [Rn]
STRH   Rt, [Rn,#off]   ; store least sig. 16-bit Rt to [Rn+off]
STRB   Rt, [Rn]        ; store least sig. 8-bit Rt to [Rn]
STRB   Rt, [Rn,#off]   ; store least sig. 8-bit Rt to [Rn+off]
PUSH   {Rt}            ; push 32-bit Rt onto stack
POP    {Rd}            ; pop 32-bit number from stack into Rd
ADR    Rd, label       ; set Rd equal to the address at label
MOV{S} Rd, <op2>       ; set Rd equal to op2
MOV    Rd, #im16       ; set Rd equal to im16, im16 is 0 to 65535
MVN{S} Rd, <op2>       ; set Rd equal to -op2
```

Branch instructions

```
B    label   ; branch to label    Always
BEQ  label   ; branch if Z == 1   Equal
BNE  label   ; branch if Z == 0   Not equal
BCS  label   ; branch if C == 1   Higher or same, unsigned ≥
BHS  label   ; branch if C == 1   Higher or same, unsigned ≥
BCC  label   ; branch if C == 0   Lower, unsigned <
BLO  label   ; branch if C == 0   Lower, unsigned <
BMI  label   ; branch if N == 1   Negative
BPL  label   ; branch if N == 0   Positive or zero
BVS  label   ; branch if V == 1   Overflow
BVC  label   ; branch if V == 0   No overflow
BHI  label   ; branch if C==1 and Z==0  Higher, unsigned >
BLS  label   ; branch if C==0 or  Z==1  Lower or same, unsigned ≤
BGE  label   ; branch if N == V   Greater than or equal, signed ≥
BLT  label   ; branch if N != V   Less than, signed <
BGT  label   ; branch if Z==0 and N==V  Greater than, signed >
BLE  label   ; branch if Z==1 and N!=V  Less than or equal, signed ≤
BX   Rm      ; branch indirect to location specified by Rm
BL   label   ; branch to subroutine at label
BLX  Rm      ; branch to subroutine indirect specified by Rm
```

Interrupt instructions

```
CPSIE  I               ; enable interrupts  (I=0)
CPSID  I               ; disable interrupts (I=1)
```

Logical instructions

```
AND{S} {Rd,} Rn, <op2> ; Rd=Rn&op2    (op2 is 32 bits)
ORR{S} {Rd,} Rn, <op2> ; Rd=Rn|op2    (op2 is 32 bits)
EOR{S} {Rd,} Rn, <op2> ; Rd=Rn^op2    (op2 is 32 bits)
BIC{S} {Rd,} Rn, <op2> ; Rd=Rn&(~op2) (op2 is 32 bits)
ORN{S} {Rd,} Rn, <op2> ; Rd=Rn|(~op2) (op2 is 32 bits)
LSR{S} Rd, Rm, Rs      ; logical shift right Rd=Rm>>Rs  (unsigned)
LSR{S} Rd, Rm, #n      ; logical shift right Rd=Rm>>n   (unsigned)
ASR{S} Rd, Rm, Rs      ; arithmetic shift right Rd=Rm>>Rs (signed)
```

```
ASR{S} Rd, Rm, #n      ; arithmetic shift right Rd=Rm>>n  (signed)
LSL{S} Rd, Rm, Rs      ; shift left Rd=Rm<<Rs (signed, unsigned)
LSL{S} Rd, Rm, #n      ; shift left Rd=Rm<<n  (signed, unsigned)
```

Arithmetic instructions

```
ADD{S} {Rd,} Rn, <op2> ; Rd = Rn + op2
ADD{S} {Rd,} Rn, #im12 ; Rd = Rn + im12, im12 is 0 to 4095
SUB{S} {Rd,} Rn, <op2> ; Rd = Rn - op2
SUB{S} {Rd,} Rn, #im12 ; Rd = Rn - im12, im12 is 0 to 4095
RSB{S} {Rd,} Rn, <op2> ; Rd = op2 - Rn
RSB{S} {Rd,} Rn, #im12 ; Rd = im12 - Rn
CMP    Rn, <op2>       ; Rn - op2      sets the NZVC bits
CMN    Rn, <op2>       ; Rn - (-op2)   sets the NZVC bits
MUL{S} {Rd,} Rn, Rm    ; Rd = Rn * Rm       signed or unsigned
MLA    Rd, Rn, Rm, Ra  ; Rd = Ra + Rn*Rm    signed or unsigned
MLS    Rd, Rn, Rm, Ra  ; Rd = Ra - Rn*Rm    signed or unsigned
UDIV   {Rd,} Rn, Rm    ; Rd = Rn/Rm         unsigned
SDIV   {Rd,} Rn, Rm    ; Rd = Rn/Rm         signed
```

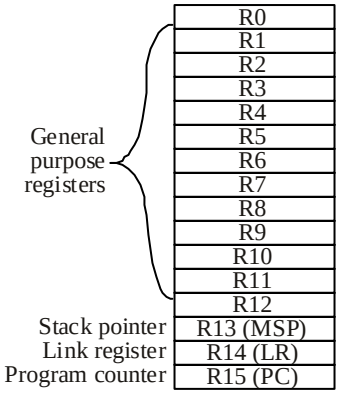
Notes `Ra Rd Rm Rn Rt represent 32-bit registers`

```
value   any 32-bit value: signed, unsigned, or address
{S}     if S is present, instruction will set condition codes
#im12   any value from 0 to 4095
#im16   any value from 0 to 65535
{Rd,}   if Rd is present Rd is destination, otherwise Rn
#n      any value from 0 to 31
#off    any value from -255 to 4095
label   any address within the ROM of the microcontroller
op2     the value generated by <op2>
```

Examples of flexible operand **<op2>** creating the 32-bit number. E.g., **Rd = Rn+op2**

```
ADD Rd, Rn, Rm          ; op2 = Rm
ADD Rd, Rn, Rm, LSL #n  ; op2 = Rm<<n  Rm is signed, unsigned
ADD Rd, Rn, Rm, LSR #n  ; op2 = Rm>>n  Rm is unsigned
ADD Rd, Rn, Rm, ASR #n  ; op2 = Rm>>n  Rm is signed
ADD Rd, Rn, #constant   ; op2 = constant, where X and Y are hexadecimal digits:
```

- produced by shifting an 8-bit unsigned value left by any number of bits
- in the form **0x00XY00XY**
- in the form **0xXY00XY00**
- in the form **0xXYXYXYXY**



Condition code bits

N negative
Z zero
V signed overflow
C carry or unsigned overflow