CMPSCI 201 – Fall 2006

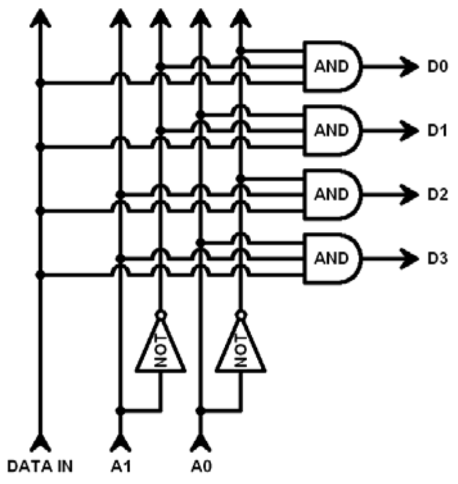
Midterm #2 – November 20, 2006

SOLUTION KEY

Professor William T. Verts

<1> 10 Points – Trace the following circuit, called a "demultiplexer", and show its outputs for all possible inputs.

INPUT			OUTPUT			
DATA	A1	A0	D3	D2	D1	D0
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0



Note that DATA IN is attached as an input to all four AND-gates, so when it is 0 all outputs must also be 0. You can only get non-0 outputs when DATA IN is 1. The AND-gates and NOT-gates are set up so that of the four possible behaviors of A1 and A0 only one of the AND-gates has its upper two bits equal to 1. Thus, the circuit always has *at most one output* equal to 1, corresponding to the "address" on the A1 and A0 lines.

<2> 5 Points – Short Answer – How would (much larger) demultiplexer circuits be used with 2-dimensional grids of CMOS flip-flops? What can they do that is necessary for the proper operation of a memory system? (It doesn't matter if the flip-flops are 6-transistor static cells or 1-transistor, 1-capacitor dynamic cells.)

Larger demultiplexers also have the characteristic that only at most one output will ever be 1. For N address lines ($A_{N-1} \ldots A_0$) there will be 2^N outputs, only one of which can ever be 1 at any time. **This makes the circuit perfect for driving the Word Lines in a memory grid, so that at most one row of memory bits is active at any time.**

<3> 10 Points – Write a complete ARM subroutine to evaluate the integer quadratic polynomial $y = 7n^2+4n-6$, where n is passed in through **R0** and the result is passed back through **R1**. Make certain that your subroutine is completely transparent with respect to any temporary registers that you might use. No solution should require more than ten lines of code at the very most (my solution is considerably shorter).

```
MUL   R1,R0,R0           ; R1 ← R0²            (N²)
RSB   R1,R1,R1,LSL #3    ; R1 ← 8×R1 – R1      (7N²)
ADD   R1,R1,R0,LSL #2    ; R1 ← R1 + 4×R0      (7N² + 4N)
SUB   R1,R1,#6           ; R1 ← R1 – 6         (7N² + 4N – 6)
MOV   PC,LR              ; Return from Subroutine
```

Since all calculations were performed in the return register, no temporary registers needed to be saved or restored.

<4> 25 Points – In a high-level language such as Pascal, I declare an array of 32-bit integers with the statement **`Var A : Array [1..4,-3..12] Of Integer ;`** where the first element of the array is at **`A[1,-3]`** and the last element of the array is at **`A[4,12]`**. There are four rows, indexed from 1 through 4, and sixteen columns, indexed from -3 through 12. In translating this array into ARM assembly language, I use the ARM directive **`A %`** ____ to allocate and initialize to zero all elements of the array (you put the number of bytes to allocate in the slot, such as **`A % 16`** to allocate 4 words of memory).

(A) How many array cells and how many bytes of memory need to be allocated?

4 rows × 16 columns = **64 array cells**, which is **256 bytes**.

(B) Write a ***minimal*** algebraic mathematical expression of the ***mapping function*** from array indices R and C (row and column) onto the ***row-major*** offset into a zero-based, 1-dimensional array of the correct number of elements (do <u>not</u> compute the byte-offset into physical memory). Your answer must be a polynomial *f* of the form *Offset* ← *f(R,C)*.

General Function = (R–1) × (12 – (-3) + 1) + (C + 3) = (R–1) × 16 + C + 3
Minimal Function *f*(R,C) = 16 × R + C – 13

(C) Repeat part (B), but this time the minimal function must be for ***column-major*** memory layout.

General Function = (C – (-3)) × (4 – 1 + 1) + (R – 1) = (C+3) × 4 + R – 1
Minimal Function *f*(R,C) = 4 × C + R + 11

(D) Using the ***row-major*** formula you developed in part (B), write the correct ARM statements to load into register **`R0`** the contents of **`A[R,C]`** where **`R`** is in register **`R1`** and **`C`** is in register **`R2`**. You do not need to perform range checking on **`R`** and **`C`**. Compute the zero-based array index into register **`R3`**, and use the **`ADR`** pseudo-instruction to put the address of array **`A`** into register **`R5`**.

```
ADR   R5,A                   ; R5 ← Address of A
MOV   R3,R1,LSL #4           ; R3 ← R×16
ADD   R3,R3,R2               ; R3 ← R×16 + C
SUB   R3,R3,#13              ; R3 ← R×16 + C – 13
LDR   R0,[R5,R3,LSL #2]      ; R0 ← A[R3]
```

(E) Using row-major format, load into **`R0`** the contents of **`A[2,5]`** using as few ARM statements as possible. As in part (D), put the address of array **`A`** into **`R5`**.

Using the formula 16×R + C – 13, where R=2 and C=5, we get array offset 24, which is byte offset 96. Either of the following should work:

```
ADR   R5,A                   ADR   R5,A+96
LDR   R0,[R5,#96]            LDR   R0,[R5]
```

<5> 15 Points – In some hypothetical high-level programming language (Ada is actually pretty close) I can specify whether a parameter to a subroutine is call-by-value (`IN`), call-by-return (`OUT`), call-by-value-return (`IN OUT`), or call-by-reference (`VAR`). Here you see a subroutine definition using this form:

```
Subroutine Glop (IN  P1:Integer ;
                 VAR P2:Integer ;
                 OUT P3:Integer) ;
```

Translate each of the following high-level language calls into ARM assembly language, showing both the prolog (setup code before the call that pushes parameters onto the stack) and epilog (code after the call that pops parameters off of the stack). If the call cannot be made due to violations in how parameters are used, tell me which parameters are in error and why (i.e., "play compiler" and give me an appropriate error message).

(A) **`Call Glop (5, X, Y) ;`**

```
MOV  R0,#5          ; Push value 5 onto stack
STR  R0,[SP,#-4]!   ;
ADR  R0,X           ; Push address of X onto stack
STR  R0,[SP,#-4]!   ;
SUB  SP,SP,#4       ; Reserve one word on stack for Y
BL   Glop           ; CALL SUBROUTINE
LDR  R0,[SP],#4     ; Pop OUT parameter Y
STR  R0,Y           ;
ADD  SP,SP,#8       ; Discard value and var parameters
```

(B) **`Call Glop (X, 5, Y) ;`**

This cannot be done using the given methodology because 5 is a *value*, not a *variable* as required by `P2`. There is no address to pass in to `P2`. For this to be made to work at all, the value 5 would need to be stored in memory and the address of that location be passed to the routine. Incidentally, this is what is done in traditional FORTRAN: all constants are stored in memory locations, and all parameters are call-by-reference. This can lead to bizarre problems if a subroutine then changes a formal parameter with a constant passed in; that numeric constant now has a new value throughout the remainder of the program!

(C) **`Call Glop (X, Y, Z) ;`**

```
LDR  R0,X           ; Push the value of X onto stack
STR  R0,[SP,#-4]!   ;
ADR  R0,Y           ; Push address of Y onto stack
STR  R0,[SP,#-4]!   ;
SUB  SP,SP,#4       ; Reserve one word on stack for Z
BL   Glop           ; CALL SUBROUTINE
```

```
        LDR  R0,[SP],#4      ; Pop OUT parameter Z
        STR  R0,Z            ;
        ADD  SP,SP,#8        ; Discard value and var parameters
```

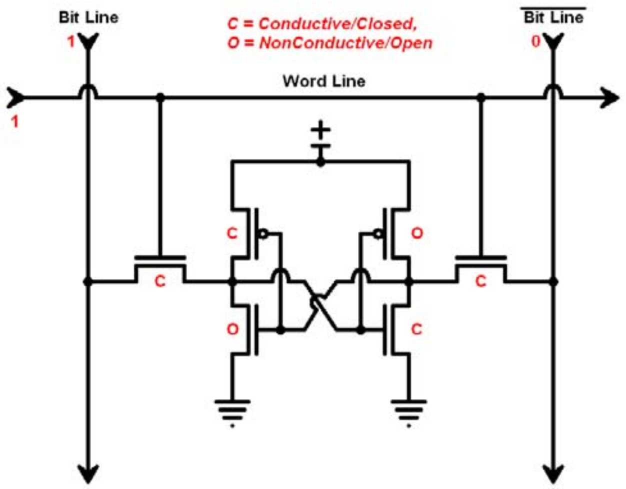
<6> 10 Points – Using the same definition of subroutine **Glop** as in problem 5, show how the framework of the subroutine itself is written in ARM assembly code. In the entry code of your subroutine you must save register **LR**, register **IP**, two 32-bit words of local storage, register **R0**, and register **R1** onto the stack, in that order. You must also set the value of the **IP** register to **SP** as soon as the old value of **IP** is saved. In the exit code you must restore the registers, discard the local storage, restore **IP**, and return. Fill in the blank below with the correct stack offset to parameter **P1**.

```
Glop                                   ; Entry code
        STR  LR,[SP,#-4]!              ; Push LR
        STR  IP,[SP,#-4]!              ; Push IP
        MOV  IP,SP                     ; Establish IP
        SUB  SP,SP,#8                  ; Reserve 2 Words
        STR  R0,[SP,#-4]!              ; Push R0
        STR  R1,[SP,#-4]!              ; Push R1

        ;----------------------------------;
        ;  "Do useful work"
        ;
             LDR  R0,[IP,#16]          ; Grab P1 off stack
        ;
        ;----------------------------------;

                                       ; Exit code
        LDR  R1,[SP],#4                ; Pop R1
        LDR  R0,[SP],#4                ; Pop R0
        ADD  SP,SP,#8                  ; Discard 2 Words
        LDR  IP,[SP],#4                ; Pop IP
        LDR  PC,[SP],#4                ; Pop and Return
```

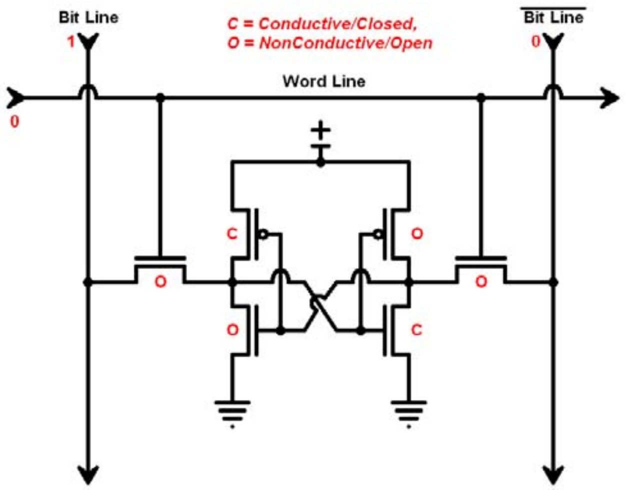
At the point labeled "Do Useful Work" the stack configuration is as follows:

SP+40	*earlier junk*	*IP+24*
SP+36	*earlier junk*	*IP+20*
SP+32	**Value for P1**	**IP+16**
SP+28	**Address for P2**	**IP+12**
SP+24	**Slot for P3**	**IP+8**
SP+20	**Return Address**	**IP+4**
SP+16	**Old IP**	**IP+0**
SP+12	**Local Word #1**	**IP-4**
SP+8	**Local Word #2**	**IP-8**
SP+4	**Saved R0**	**IP-12**
SP+0	**Saved R1**	**IP-16**

<7> 10 Points – In the following static RAM bit, the Bit Line on the left is 1 (positive) and its complement on the right is 0 (grounded). When the Word Line is brought to 1 (positive), some transistors will conduct, acting like closed switches, and some will be cut off, acting like open switches. Next to each transistor in the diagram below write "C" for conductive and "O" for open:

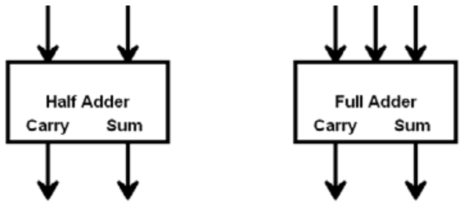


After the flip-flop settles, the Word Line is now brought to 0 (grounded). Show the conductive and non-conductive transistors in the diagram below:



When the Word line is 1, the input values established on the Bit Lines sets the flip-flop appropriately. Once the flip-flop settles, turning off the Word line simply *isolates* the flip-flop from the Bit Lines; it is "remembering" without asserting its value anywhere else. By making both Bit Line inputs "3-state" (essentially not connected to any external data source), opening the Word Line again causes the flip-flop to read out its value, putting on the Bit Lines what it is remembering.

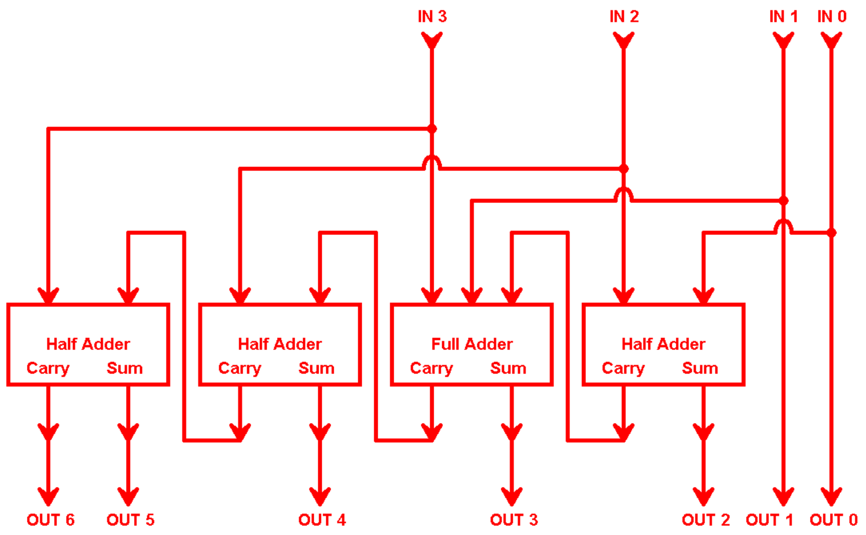
<8> 15 Points – Draw a circuit with four input bits that multiplies the 4-bit unsigned input value by 5, in hardware. Because the input value is between 0 and 15, the output will be between 0 and 75, so the result will require 7 bits of output. Your circuit must use nothing but full-adders and half-adders, oriented as shown below.



To understand this problem you need only look at the binary value of our constant multiplier, 5. In binary, 5 is 101, which means that 5×N is 4×N + N. Multiplying a number by 4 is equivalent to shifting it left by two bits, which can be done in hardware by simply moving the input wires two bits to the left. Since N is four bits wide, 4×N is N with two zeroes appended to the right. Those two numbers are what need to be added:

			`In3`	`In2`	`In1`	`In0`
`+`	`In3`	`In2`	`In1`	`In0`	`0`	`0`
`Out6`	`Out5`	`Out4`	`Out3`	`Out2`	`Out1`	`Out0`

An adder-chain will do the job, but only a very little hardware is needed. In most cases, you are adding two bits together, requiring half-adders, but only in one case will you also need to accommodate an *additional* carry, requiring a full-adder (`In3 + In1 + carry` from adding `In2` and `In0`). More full-adders would be necessary if the input operand were wider than 4 bits.


IN 3
IN 2
IN 1
IN 0
Half Adder
Carry
Sum
Half Adder
Carry
Sum
Full Adder
Carry
Sum
Half Adder
Carry
Sum
OUT 6
OUT 5
OUT 4
OUT 3
OUT 2
OUT 1
OUT 0