ECE253, Fall 2001, Final Examination, S. Davies and A. Moshovos

There are 5 questions in this exam. A summary of the 68000 instruction set is attached to this document. Plan your time carefully as some questions may require more time than others. ***Only answers given in the examination booklet will be considered during grading.*** *Good luck!*

1) Assembly – Memory Accesses and Simple Operations

(10 Marks) For a 68000-based computer system assume that memory is initialized as follows:

```
            org    $30000
            dc.b   $04,$ff
            dc.w   $abcd
     FOO    dc.l   $0f2e3d4c
            dc.b   $00,$01,$f0,$a1
```

Show the values of the registers listed on the rightmost column for each of the following instructions. Assume that all instructions execute in the order shown and that initially all registers are zero.

```
         INSTRUCTION              SHOW THIS REGISTER
1.  move.l   #$30000, a0          a0
2.  move.b   $30000, d1           d1
3.  lsr.b    #1,d1                d1
4.  move.b   3(a0), d2            d2
5.  move.w   (a0,d1), d3          d3
6.  move.l   FOO, d4              d4
7.  move.l   #FOO, a0             a0
8.  move.l   (a0)+, d1            d1
9.  move.w   (a0)+, d2            d2
10. move.b   (a0), d1             d1
```

2) Assembly – Calling Conventions:

(**15 Marks**) The following code shows two routines CALLER and CALLEE. The CALLER routine is calling the CALLEE. The code is missing the instructions to save and restore registers. Assume that the calling convention is as follows: register d0 is used for the return value if any, registers d1 through d3 and a0 through a3 are callee-saved and the rest are caller saved. Insert all necessary instructions around the call and in the CALLEE procedure to preserve all necessary register values. Write the four pieces of code on your exam notebook, labeling them as requested. For full marks, save only the registers that are necessary.

```
CALLER:
      movem.l     d2/a2-a3,-(a7)
      move.l      4(a7), a2
      move.l      8(a7), a3
      move.l      12(a7),d6
      adda.l      (a2),a3
      move.l      (a3), d2
      add.l       d2,d6
      lrl.l       #2,d6
      FILL IN SAVE/RESTORE CODE. LABEL THIS CODE AS "CALLER: BEFORE
      CALL".
      move.l      -(a7), a2
      move.l      -(a7),a3
      bsr         CALLEE
      add.l       #8, a7
      FILL IN SAVE/RESTORE CODE. LABEL THIS CODE AS "CALLER: AFTER
      CALL".
      add.l       d0,d6
      or.l        d2,d6
      move.l      (a2),d2
      add.l       d2,d6
      move.l      d6, d0
      movem.l     (a7)+, d2/a2-a3
      rts

CALLEE:
      FILL IN SAVE/RESTORE CODE. LABEL THIS CODE AS "CALLEE: PROLOGUE".
      move.l      4(a7), d2
      move.l      8(a7), d6
      add.l       d2,d3
      move.l      $40000, a2
      move.l      $50000, a6
      add.l       (a2,d2),a2
      add.l       (a2,d6),d6
      clr.l       d0
loop: add.b       (a6)+, d0
      dbf         d6, loop
      FILL IN SAVE/RESTORE CODE. LABEL THIS CODE AS "CALLEE: EPILOGUE".
      rts
```

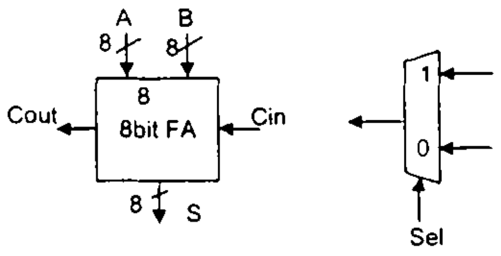
3) Logic Design / Adders and Multimedia Instructions

(20 Marks) The 68k instruction set was extended to include multimedia oriented instructions. These instructions operate on the existing 32-bit registers treating them as a collection of four individual bytes. For example, the "`add4b d0,d1`" instruction adds the contents of registers d0 and d1 treating this as four independent adds as follows:

	Byte 3	Byte 2	Byte 1	Byte 0
D0	`$0f`	`$01`	`$02`	`$ff`
	+	+	+	+
D1	`$01`	`$02`	`$05`	`$01`
	=	=	=	=
Result in D1	`$10`	`$03`	`$07`	`$00`

Notice that all four additions are done independently and do not affect each other. For example, while the addition of the bytes labeled "byte 0" overflows this does not affect the values next to them. In this question, you will be designing the adder to support both the regular add.l and the new add4b.

a) **(10 Marks)** Starting from a 32-bit ripple-carry adder convert it so that it supports two modes of operation: 32-bit wide addition or four independent additions, each 8-bits. The four independent additions are performed as shown above. The mode of operation is to be controlled by an input line WIDTH. When WIDTH is zero, the circuit behaves like a 32-bit adder, otherwise it behaves as four independent 8-bit adders as required to implement the operation described above. **You may use muxes but no other gates.** In your examination booklet draw the circuit using the following symbols:



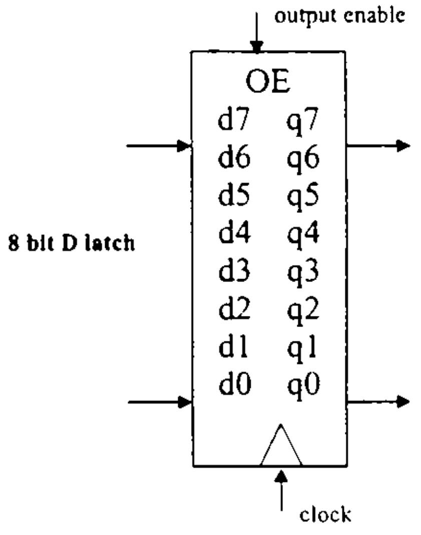
b) **(10 marks)** In some cases it is convenient to have an adder that implements saturating arithmetic. Such an adder saturates to the maximum possible unsigned integer if the result would overflow. A conventional adder simply wraps around 0 as we explained in class. For example, assuming 8-bit addition adding \$fe and \$03 would produce \$01 with a conventional adder and \$ff with a saturating one. Starting from a 32-bit ripple-carry adder and using MUXES, implement a 32-bit saturating adder. Assume that the adder will be always used to do unsigned 32-bit integer arithmetic.

4) Interrupt Vector Finite State Machine

(25 Marks) ASPIC Corporation has developed a new 8 bit processor with a 16 bit address bus (A0 through A15) that mimics the 68000's vectored interrupt capability. Just as for the 68000, an interrupt is requested by asserting the INTR line (active high in this case). A short time later the processor will then assert the IACK line (also active high). When the IACK line is high, the interrupting device should place the 8 bit interrupt vector number on the data bus and maintain that output until IACK is de-asserted. Otherwise its outputs should not load the data bus (i.e., they should be high-Z). The interrupting device should deassert INTR within one clock cycle of receiving the IACK.

ASPIC's processor provides a synchronous bus interface with 8 data lines (D0 through D7). To write to a location in memory, it first drives the address lines with a bit pattern matching the desired location and also places on the data lines the desired 8-bit value. After a 100ns delay it then sets the WR line high indicating that the address and data bus values are valid and may be loaded into the memory byte. The address and data bus values are maintained until WR is de-asserted 50ns later. There is a separate RD line that is asserted on reads but we do not need this for this task.

a) **(10 marks)** Using an 8 bit D-latch, AND and NOT gates provide a memory mapped register to store the interrupt vector in the I/O interface. The latch is equivalent to eight D flip flops, each tied to the same clock input with the output of each D F/F driving a tri-state buffer and all the tri-state buffers tied to the same output enable line. The latch should be loaded when a write is done to location $7FFF. The latch outputs will be driven by a signal VECCTL to be created in part (b). Connect both latch inputs and outputs so as to provide the ability to write the vector and have it accessed during the IACK cycle. Your circuit should use address lines A0-A15, data bus lines D0-D7, WR and VECCTL as input and/or output as appropriate.



b) **(15 marks)** Formally develop a synchronous state machine to control the circuit in part (a). The state machine is responsible for driving the INTR line once an external signal DEVINT, indicating the external peripheral wants action, has been active. The DEVINT signal will eventually de-activate. It should respond to IACK as described above, generating the appropriate pattern of signals INTR and VECCTL. Your development should include a state diagram and a state table. Explain what each state is meant to do. Implement the state machine using D flip flops and optimized combinational logic. Your finite state machine should generate one interrupt request for each time DEVINT transitions from 0 (inactive) to 1 (active).

5) Assembly

(30 marks) An engineering team has developed a transmitter that encodes data (Hamming code) so that single bit errors due to line noise may be corrected. The transmitter circuit accepts 4 bits in parallel as input, generates three parity bits (p1, p2, p3) and transmits the resulting 7 bits serially. The parity bits are generated as follows:

$$p1=b0+b1+b2 \qquad p2=b1+b2+b3 \qquad p3=b0+b1+b3$$

where d0-d3 are the 4 input data bits and the addition is modulo 2 (single bit binary addition, i.e., XOR). The circuit outputs bits in the following order: b0, b1, b2, b3, p1, p2, p3. Stop and start bits are not required as both transmitter and receiver have access to the same serial bit clock.

In this question you will write, in parts b and c, an assembly program that decodes the data after they are received. The program reads a byte from a memory location RECVB. The byte contains the received bits in the following order:

bit	7	6	5	4	3	2	1	0
	0	P3	P1	P0	B3	B2	B1	B0

Here capital letters have been used to denote received values that may contain errors. The program places in the lower 4-bits of d0 the corrected received bits (c3,c2,c1,c0), where any single bit errors in the raw received serial input data (B3,B2,B1,B0) have been corrected in (c3,c2,c1,c0). The Hamming decoder program should calculate the parity on reception using the received bits as in:

$$p1'=B0+B1+B2 \qquad p2'=B1+B2+B3 \qquad p3'=B0+B1+B3$$

If the reception was without error then p1'=P1, p2'=P2 and p3'=P2 or, equivalently s1=p1'+P1=0, s2=p2'+P2=0 and s3=p3'+P3=0 where the additions are single bit modulo 2 (XOR). However, a single error in a data bit or a parity bit would make one or more of these sums non-zero. By examining s1-s3 (and assuming only one bit can be in error) one can determine which if any of the data bits are erroneous.

a) (5 Marks) Do this examination then reproduce and fill in the following truth table in your examination booklet where a 1 in an output column (Oi) indicates the corresponding data bit is in error. If a combination corresponds to more than one bit error then Oi can be "don't care".

s3	s2	s1	O3	O2	O1	O0
0	0	0				
0	0	1				
0	1	0				
0	1	1				
1	0	0				
1	0	1				
1	1	0				
1	1	1				

b) (10 Marks) Provide the 68k assembly program that calculates s3,s2,s1 and stores them as the three low order bits of d2. Assume that the received data is in register d1.

c) (15 Marks) Provide a program that takes the output of part b and returns the corrected four data bits in the low four bits of d0. Use the order shown above (b3,b2,b1,b0) for the result in d0. Assume that the original received data is in register d1.

68000 Instruction Set

Instruction	Description	Assembler Syntax	Data Size	Condition Codes X N Z V C
ABCD	Add BCD with extend	Dx,Dy	B--	• U • U •
		-(Ax),-(Ay)		
ADD	ADD binary	Dn,< ea >	BWL	• • • • •
		< ea >,Dn		
ADDA	ADD binary to An	< ea >,An	-WL	- - - - - -
ADDI	ADD Immediate	#x,< ea >	BWL	• • • • •
ADDQ	ADD 3-bit immediate	#,< ea >	BWL	• • • • •
ADDX	ADD eXtended	Dy,Dx	BWL	• • • • •
		-(Ay),-(Ax)		
AND	Bit-wise AND	< ea >,Dn	BWL	- • • 0 0
		Dn,< ea >		
ANDI	Bit-wise AND with Immediate	#< data>,< ea >	BWL	- • • 0 0
ASL	Arithmetic Shift Left	#,Dy	BWL	• • • • •
		Dx,Dy		
		< ea >		
ASR	Arithmetic Shift Right	...	BWL	• • • • •
Bcc	Conditional Branch	Bcc.S < label>	BW-	- - - - - -
		Bcc.W < label>		
BCHG	Test a Bit and ChanGe	Dn,< ea >	B-L	- - • - -
		#< data>,< ea >		
BCLR	Test a Bit and CLEAR	...	B-L	- - • - -
BSET	Test a Bit and SET	...	B-L	- - • - -
BSR	Branch to SubRoutine	BSR.S < label>	BW-	- - - - - -
		BSR.W < label>		
BTST	Bit TeST	Dn,< ea >	B-L	- - • - -
		#< data>,< ea >		
CHK	CHecK Dn Against Bounds	< ea >,Dn	-W-	- • U U U
CLR	CLEAR	< ea >	BWL	- 0 1 0 0
CMP	CoMPare	< ea >,Dn	BWL	- • • • •
CMPA	CoMPare Address	< ea >,An	-WL	- • • • •
CMPI	CoMPare Immediate	#< data>,< ea >	BWL	- • • • •
CMPM	CoMPare Memory	(Ay)+,(Ax)+	BWL	- • • • •
DBcc	Looping Instruction	DBcc Dn,< label>	-W-	- - - - - -
DIVS	DIVide Signed	< ea >,Dn	-W-	- • • • 0
DIVU	DIVide Unsigned	< ea >,Dn	-W-	- • • • 0
EOR	Exclusive OR	Dn,< ea >	BWL	- • • 0 0
EORI	Exclusive OR Immediate	#< data>,< ea >	BWL	- • • 0 0
EXG	Exchange any two registers	Rx,Ry	--L	- - - - - -
EXT	Sign EXTend	Dn	-WL	- • • 0 0
ILLEGAL	ILLEGAL-Instruction Exception	ILLEGAL		- - - - - -
JMP	JuMP to Affective Address	< ea >		- - - - - -
JSR	Jump to SubRoutine	< ea >		- - - - - -
LEA	Load Effective Address	< ea >,An	--L	- - - - - -
LINK	Allocate Stack Frame	An,#< displacement>		- - - - - -
LSL	Logical Shift Left	Dx,Dy	BWL	• • • 0 •
		#,Dy		
		< ea >		
LSR	Logical Shift Right	...	BWL	• • • 0 •
MOVE	Between Effective Addresses	< ea >,< ea >	BWL	- • • 0 0
MOVE	To CCR	< ea >,CCR	-W-	I I I I I
MOVE	To SR	< ea >,SR	-W-	I I I I I
MOVE	From SR	SR,< ea >	W-	- - - - - -
MOVE	USP to/from Address Register	USP,An	--L	- - - - - -
		An,USP		
MOVEA	MOVE Address	< ea >,An	-WL	- - - - - -
MOVEM	MOVE Multiple	,< ea >	-WL	- - - - - -
		< ea >,		
MOVEP	MOVE Peripheral	Dn,x(An)	-WL	- - - - - -
		x(An),Dn		
MOVEQ	MOVE 8-bit immediate	#<-128.+127>,Dn	--L	- • • 0 0
MULS	MULtiply Signed	< ea >,Dn	-W-	- • • 0 0
MULU	MULtiply Unsigned	< ea >,Dn	-W-	- • • 0 0

NBCD	Negate BCD	< ea >	B--	• U • U •
NEG	NEGate	< ea >	BWL	• • • • •
NEGX	NEGate with eXtend	< ea >	BWL	• • • • •
NOP	No OPeration	NOP		- - - - -
NOT	Form one's complement	< ea >	BWL	- • • 0 0
OR	Bit-wise OR	< ea >,Dn	BWL	- • • 0 0
		Dn,< ea >		
ORI	Bit-wise OR with Immediate	#< data>,< ea >	BWL	- • • 0 0
PEA	Push Effective Address	< ea >	--L	- - - - -
RESET	RESET all external devices	RESET		- - - - -
ROL	ROtate Left	#,Dy	BWL	- • • 0 •
		Dx,Dy		
		< ea >		
ROR	ROtate Right	...	BWL	- • • 0 •
ROXL	ROtate Left with eXtend	...	BWL	• • • 0 •
ROXR	ROtate Right with eXtend	...	BWL	• • • 0 •
RTE	ReTurn from Exception	RTE		I I I I I
RTR	ReTurn and Restore	RTR		I I I I I
RTS	ReTurn from Subroutine	RTS		- - - - -
SBCD	Subtract BCD with eXtend	Dx,Dy	B--	• U • U •
		-(Ax),-(Ay)		
Scc	Set to -1 if True, 0 if False	< ea >	B--	- - - - -
STOP	Enable & wait for interrupts	#< data>		I I I I I
SUB	SUBtract binary	Dn,< ea >	BWL	• • • • •
		< ea >,Dn		
SUBA	SUBtract binary from An	< ea >,An	-WL	- - - - -
SUBI	SUBtract Immediate	#x,< ea >	BWL	• • • • •
SUBQ	SUBtract 3-bit immediate	#< data>,< ea >	BWL	• • • • •
SUBX	SUBtract eXtended	Dy,Dx	BWL	• • • • •
		-(Ay),-(Ax)		
SWAP	SWAP words of Dn	Dn	-W-	- • • 0 0
TAS	Test & Set MSB & Set N/Z-bits	< ea >	B--	- • • 0 0
TRAP	Execute TRAP Exception	#< vector>		- - - - -
TRAPV	TRAPV Exception if V-bit Set	TRAPV		- - - - -
TST	TeST for negative or zero	< ea >	BWL	- • • 0 0
UNLK	Deallocate Stack Frame	An		- - - - -

Symbol	Meaning
•	Set according to result of operation
-	Not affected
0	Cleared
1	Set
U	Outcome (state after operation) undefined
I	Set by immediate data

< ea >	Effective Address Operand
< data>	Immediate data
< label>	Assembler label
< vector>	TRAP instruction Exception vector (0-15)
< rg.lst>	MOVEM instruction register specification list
< displ.>	LINK instruction negative displacement
...	Same as previous instruction

Addressing Modes	Syntax
Data Register Direct	Dn
Address Register Direct	An
Address Register Indirect	(An)
Address Register Indirect with Post-Increment	(An)+
Address Register Indirect with Pre-Decrement	-(An)
Address Register Indirect with Displacement	w(An)
Address Register Indirect with Index	b(An,Rx)
Absolute Short	w
Absolute Long	l
Program Counter with Displacement	w(PC)
Program Counter with Index	b(PC Rx)
Immediate	#x
Status Register	SR
Condition Code Register	CCR

Legend

```
   Dn    Data Register        (n is 0-7)
   An    Address Register     (n is 0-7)
   b     08-bit constant
   w     16-bit constant
   l     32-bit constant
   x     8-, 16-, 32-bit constant
   Rx    Index Register Specification, one of:
            Dn.W  Low 16 bits of Data Register
            Dn.L  All 32 bits of Data Register
            An.W  Low 16 bits of Address Register
            An.L  All 32 bits of Address Register
```

Condition Codes for Bcc, DBcc and Scc Instructions.

Condition Codes set after CMP D0,D1 Instruction.

Relationship	Unsigned	Signed
D1 < D0	CS - Carry Bit Set	LT - Less Than
D1 <= D0	LS - Lower or Same	LE - Less than or Equal
D1 = D0	EQ - Equal (Z-bit Set)	EQ - Equal (Z-bit Set)
D1 != D0	NE - Not Equal (Z-bit Clear)	NE - Not Equal (Z-bit Clear)
D1 > D0	HI - HIgher than	GT - Greater Than
D1 >= D0	CC - Carry Bit Clear	GE - Greater than or Equal
	PL - PLus (N-bit Clear)	MI - Minus (N-bit Set)
	VC - V-bit Clear (No Overflow)	VS - V-bit Set (Overflow)
	RA - BRanch Always	
DBcc Only -	F - Never Terminate (DBRA is an alternate to DBF)	
	T - Always Terminate	
Scc Only -	SF - Never Set	
	ST - Always Set	