

EECS 370
Exam 1 solution
Winter 2006
February 7, 2006

Name:

University of Michigan unickname:
(NOT your student ID number!)

Open book, open notes. No calculators, laptops, PDAs, cell phones, etc. This exam has 9 pages, 9 questions, and 100 points. Questions vary in difficulty; it is strongly recommended that you do not spend too much time on any one question.

The rules of the Honor Code of the University of Michigan College of Engineering apply for this exam.

Honor code pledge: I have neither given nor received aid on this examination, nor have I concealed any violations of the Honor Code.

Signature:

(examinations without a signed honor pledge will not be graded)

1. (22 points):

Circle TRUE or FALSE for each statement.

(a) **TRUE** FALSE

The ENIAC was the first working vacuum tube general purpose computer.

(b) TRUE **FALSE**

Load-store architectures are ISAs which include at least 3 distinct load and 3 distinct store instructions, which are used to load/store single bytes, half-words, and words.

(c) **TRUE** FALSE

The endianness of an ISA does not affect the format of byte-size data.

(d) TRUE **FALSE**

The propagation delay of a circuit measures the maximum time between changes in the output values.

(e) TRUE **FALSE**

Data is stored in static memories by charging and discharging capacitors with electrostatic charge.

(f) **TRUE** FALSE

Pseudo-instructions are ISA instructions which the assembler converts to other legal instructions.

(g) TRUE **FALSE**

When compiling an if/else control block from C to MIPS assembly, at least 4 branch instructions are required.

(h) TRUE **FALSE**

Object files are complete machine code programs that can be executed by a microprocessor.

(i) TRUE **FALSE**

Moore's law states that the density of transistor integration in a chip grows by 10 times every two years, that is, in 2008 engineers will be able to fit 10 times more transistors in a one inch-square die than they can fit today.

(j) **TRUE** FALSE

0001 0001 0001 0001 0001 0001 0001 0001 (binary) is a valid machine code MIPS instruction.

(k) TRUE **FALSE**

In the MIPS ISA, the C function below needs to save some arguments onto the stack:

```
int foobar(double apple[], double orange, int banana)
```

2. (14 points):

For each LC2K6 instruction or sequence of instructions below, in the blank provided, indicate which addressing mode is being used or emulated. The following conventions are used in the instruction sequences. Register 0 always contains zero. Register 1 is used as a temporary for operations that take several instructions to emulate. For each sequence, pick the ONE MOST SPECIFIC choice from the following list: Auto Increment, Base Plus Displacement, Direct, Double Indirect, Immediate, Indirect, PC relative, Register, Register Indirect, Tagged Indirect.

(a) **Answer: Direct**

lw 0 2 900

(b) **Answer: Register**

add 2 2 2

(c) **Answer: Base Plus Displacement**

lw 3 2 900

(d) **Answer: PC relative**

beq 0 0 900

(e) **Answer: Indirect**

lw 0 1 900

lw 1 2 0

(f) **Answer: Double Indirect**

lw 0 1 900

lw 1 1 0

lw 1 2 0

(g) **Answer: Register Indirect**

lw 3 2 0

3. (4 points):

Given the *initial* values in memory locations 224-227 (below left), which of the following MIPS code fragments will result in the *new* values for location 224-227 seen on the right?

Initial:	Mem[224]	0x00	New:	Mem[224]	0x25
	Mem[225]	0x25		Mem[225]	0x62
	Mem[226]	0x62		Mem[226]	0xFE
	Mem[227]	0xFE		Mem[227]	0xFE

(a)

```
lw    $4, 224($0)
lh    $5, 227($0)
sh    $5, 225($0)
sb    $4, 226($4)
```

(b)

```
lh    $4, 225($0)
lbu   $4, 224($0)
sb    $4, 227($4)
sw    $5, 226($4)
```

(c)

```
lui   $4, 227($0)
lh    $5, 225($0)
sb    $4, 226($0)
sh    $5, 224($4)
```

(d) CORRECT ANSWER

```
lb    $4, 227($0)
lh    $5, 225($0)
sh    $5, 226($4)
sb    $4, 226($0)
```

(e)

none of the above

4. (10 points)

Consider the following segment of MIPS assembly code, and convert the last instruction (**bgt**) into machine code: (you only need to convert the last instruction; do not worry about the incomplete ones with "...")

```
goal : add  ....  
      nor  ....  
      sll  ....  
      bgt $7 $11 goal
```

(a) Write the machine code for the instruction in hexadecimal format below: Show your work for partial credit.

The assembly translation of the **bgt** instruction is: (format: opcode rs rt rd)
slt \$11 \$7 \$1 bne \$1 \$0 -5
(bne \$1 \$0 -20 is also a valid answer)

The machine code in binary format is:

```
000000 01011 00111 00001 00000 101010  
000101 00001 00000 11111111 11111011
```

Hence the hex format is:

```
0x0167 082a  
0x1420 fffb
```

(when using -20:
0x0167 082a
0x1420 ffec)

(b) Count how many bits are set to 1 in the code you generated and write it below:
28 (26 with the alternate solution)

5. (8 points):

A combinational logic block CHECKER has two inputs A and B. It has two outputs GEQ_1 and ODD defined as follows:

$$\text{GEQ}_1 = 1 \text{ iff } A+B \geq 1$$

$$\text{ODD} = 1 \text{ iff } A+B \text{ is odd}$$

Draw truth tables for both GEQ_1 and ODD.

A	B	GEQ_1	ODD
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	0

6. (12 points):

Consider the following LC2K6 assembly language program. Assume all registers initially contain zero.

```

        lw      0 1 data
        lw      0 4 one
        lw      0 6 six
        add     0 6 5
        nand    0 0 2
        add     0 0 3
more    beq     0 1 done
        add     1 2 1
        add     3 4 3
        add     4 5 4
        add     5 6 5
        beq     7 7 more
done    halt
one     .fill   1
six     .fill   6
data    .fill   4
```

(a) When this is run, how many instructions are executed?

Answer: 32

(b) What is the final value for all registers? For partial credit, show all values assigned to each register in order from left to right, and neatly cross out a value when you write a new value for a register.

reg[0] = (initially zero, never changes)

reg[1] = 4 ~~3~~ ~~2~~ ~~1~~ 0

reg[2] = -1

reg[3] = 0 1 8 27 64

reg[4] = 1 7 19 37 61

reg[5] = 6 12 18 24 30

reg[6] = 6

reg[7] = (initially zero, never changes)

(c) Assume this is used as a function (like the project 1 multiplier), with a single parameter “x” placed at the .fill labeled “data”. The result is the final value in register 3 at the halt. What function of x does this code compute?

Answer: x^3

7. (10 points):

Consider an architecture with the following properties:

- byte addressable
- char occupies 1 byte and needs 1 byte alignment
- short occupies 2 bytes and needs 2 byte alignment
- int occupies 4 bytes and needs 4 byte alignment
- float occupies 4 bytes and needs 4 byte alignment
- double occupies 8 bytes and needs 8 byte alignment

The following global declarations are made:

```
struct s {
    char name[81];
    float precision;
    double latitude;
    double longitude;
    short type;
    double altitude;
};
```

```
char tags[5];
int i;
struct s data[10];
```

Assume the variable `tags` starts at address 0.

(a) How much memory is required (in bytes) for the global variables assuming NO optimization? For partial credit, show the starting address of each variable and the offset (from the start of the structure) of each field.

Answer: Structure `s`: Field offsets: `name=0`, `precision=84`, `latitude=88`, `longitude=96`, `type=104`, `altitude=112`; total structure size = $112+8 = 120$ bytes. Global variable addresses: `tags=0`, `i=8`, `data=16`; size of `data` = $120*10 = 1200$ bytes, total memory = $16+1200 = 1216$ bytes.

(b) How much memory is required (in bytes) for the global variables assuming variables are reordered to use storage as efficiently as possible? (same rules as part (a) for partial credit)

Answer: (one of several optimal reorderings) Structure `s`: field offsets: `latitude=0`, `longitude=8`, `altitude=16`, `precision=24`, `type=28`, `name=30`, Total structure size = $30+81 = 111$; this pads to 112 (next multiple of 8). Global variable addresses: `data=0`, `i=1120`, `tags=1124`; size of `tags` = 5 bytes, total memory = $1124+5 = 1129$ bytes.

8. (10 points):

The following MIPS assembly function contains a number of bugs and inefficiencies related to register saving. In particular, some registers may not have been saved that should have been, some registers may have been saved unnecessarily, some registers may be saved or restored at the wrong place on the stack, and space may not be allocated or deallocated from the stack correctly. The instructions related to register saving are all marked with a * in the code below; you may assume that the rest of the function (instructions without a *) is all correct. Fix the code by correctly following standard MIPS calling conventions. Cross out any unnecessary instructions, fill in any missing instructions, and correct any instructions with errors. Note that in the MIPS architecture \$si and \$ti denote callee- and caller- saved registers, respectively.
function:

```
addi $sp, $sp, 16      addi $sp, $sp, -12
sw $ra, 8($sp)
sw $t1, 4($sp)          sw $s2, 4($sp)
sw $s1, 0($sp)
```

```
lw $s1, 0($a0)
lbu $s2, 4($a0)
add $t1, $s1, $s2
slt $t1, $zero, $t1
beq $zero, skip
```

```
addi $sp, $sp, 16      addi $sp, $sp, -4
sw $ra, 8($sp)
sw $s1, 4($sp)
sw $t1, 0($sp)
```

```
jal foobar
lw $t1, 8($sp)          lw $t1, 0($sp)
lw $s1, 4($sp)
lw $ra, 0($sp)
addi $sp, $sp, -16     addi $sp, $sp, 4
```

```
add $t1, $s2, $t1
skip: sw $t1, 8($a0)
```

```
lw $s1, 8($sp)          lw $s1, 0($sp)
lw $t1, 4($sp)          lw $s2, 4($sp)
lw $ra, 0($sp)          lw $ra, 8($sp)
addi $sp, $sp, -16     addi $sp, $sp, 12
```

```
jr $ra
```

9. (10 points):

You are a chief architect at Stacks Inc. - a company specializing in production of small stack-based processors. Stack-based processors contain a hardware stack instead of the register file and can only access data at the top or top-1 position of the stack. You are asked to design a new ISA for the next generation PowerStack processor. Addresses, data, and instructions are all 8 bits long on this processor. The processor has the following instructions:

```
pop          // removes and discards the top of the stack

add          // pops the top and top-1 values,
            // adds them and pushes result on the stack

pushi IMM    // pushes IMM on top of the stack

beqz IMM     // if top entry is 0 then start
            // executing instructions at PC+1+IMM

lw IMM      // load word at address IMM and push it on the stack

lwt IMM     // pop the top value, add IMM to it and push
            // value at the resulting address on the stack
```

(a). What is the maximum bit-width of the IMM field in PowerStack assembly?

5 bits

(b) How many more instructions can you add to PowerStack ISA to increase its capabilities but preserve the size of the immediate field?

2

(c) If the immediate value (IMM) is signed, what is the range of values it can encode?

-16 to 15

(d) If the immediate value (IMM) is unsigned, what is the range of values it can encode?

0 to 31

(e) What addressing mode do the following PowerStack instructions use?

(i) lw: **Direct**

(ii) lwt: **Base+Displacement**

(iii) beqz: **PC-relative**