ELE 375 / COS 471 Midterm
Fall, 2002
Prof. Martonosi

Question	Score
1	/15
2	/15
3	/10
4	/35
5	/25
Total	**/ 100**

Please write your answers clearly in the space provided. For full credit and/or to get partial credit, show your work.

Name: ____________________SOLUTIONS!!!______________________

Honor code:

____

____

____

____

____

____

1) (15 points) Arithmetic. The PAW instruction set (as well as its commercially-widespread ancestor ARM) include both a "carry" bit and an "overflow" bit. Describe the difference between the functionality of these two bits and give an example of an arithmetic operation that would lead to them being set to different values.

The overflow bit is set when there is a signed overflow. That is, when the carry into the most-significant bit is not equal to the carry out of most-significant bit. Alternatively, this is when the result of an operation is too large to be represented in the given number system.

From the PAW reference manual, the carry bit is set when:

- *the addition produces a carry (for unsigned numbers this is overflow, but for signed numbers, it may not be overflow.)*
- *or when subtraction does not produce a borrow. (This is non-intuitive but is equivalent to saying that if the second operand were negated and added to the first operand, there would be a carry out of that addition.)*

Example:

```
  1111.....1111 (-1)
+ 0000.....0001 (1)
=
 1 0000....0000  (0)
```

This produces a carry but no overflow.

2) (15 points) The notion of "big-endian" or "little-endian" memory addressing is something that must be stated as part of the architecture specification. Distinguish *architecture* from *implementation* and give two reasons why endian-ness is considered an attribute of the *architecture*, rather than a characteristic of individual microprocessor *implementations*.

<u>Architecture</u> is an abstraction layer; it is a specification of instruction semantics, addressing modes, registers, instruction set, and indeed, anything programmers need to know to make a binary machine language program work correctly.

While architectures are typically long-lasting, and in class I use 30 years as a concrete example, it is not quite true that architectures "last 30 years and no more".

<u>Implementation</u>: a particular instance of an architecture. There can be several different implementations of an architecture that differ in terms of cost, speed, size, technology. But they all must adhere to the architecture specification.

7 points for correct definitions.

Endianness is part of architecture because… (any two of these, or anything else reasonable) got you 8 points)

- *Compiler/linker needs to be able to layout instructions in code memory without incorrectly positioning bytes*
- *Need to be able to fetch/decode instructions without incorrectly flipping bytes*
- *When loading/storing data, we need to agree on an ordering of bytes for sending things out to memory*
- *Software compatibility issues*

3) (10 points) The design team for a simple, single-issue processor is choosing between a pipelined or non-pipelined implementation. Here are some design parameters for the two possibilities:

Parameter	Pipelined Version	Non-Pipelined Version
Clock Rate	50MHz	15 MHz
CPI for ALU instructions	1	1
CPI for Control instructions	2	1
CPI for Memory instructions	2	1

For a very long-running program with 60% ALU instructions, 10% control instructions and 30% memory instructions, use the CPU performance equation to demonstrate which design will be faster and by how much.

Assume there are X instructions where X is very large (so we don't need to worry about pipeline fill/drain…)

*CPU Time = #Instructions * CPI * Clock Cycle Time*

For Pipelined Version:
*CPU Time = X (0.6*1 + 0.1*2 + 0.3*2) * (1/50,000,000)*
*= 1.4X / 50,000,000 = 2.8 * 10^{-8} X seconds*

CPU Time for Non-Pipelined Version:
*= X (0.6*1 + 0.1*1 + 0.3*1) / 15,000,000*
*= 1.0X / 15,000,000 = 6.67 *10^{-8} X seconds*

When you compute the ratio of which is faster, the X's will cancel out and you are left with:

The pipelined version is faster than the non-pipelined version by 6.67-2.8 / 6.67 = 138% faster

OR (also acceptable):
The pipelined version is 6.67 / 2.8 = 2.38 times faster.

4) (35 points) For the pipeline below:

IF	ID	RF	EX	MEM1	MEM2	WB

IF: Instruction fetch.
ID: Decode instruction.
RF: Fetch register operands. (Assume this occurs in the last half of the cycle.)
EX: Perform arithmetic operations and effective address calculations, branch comparisons, and branch target address calculations.
MEM1: Begin data memory access
MEM2: Complete data memory access.
WB: Write back results (Assume this occurs in the first half of the cycle.)

a) (8 points) When determining possible bypassing needs for an arithmetic instruction i, which previous instructions (i-1, i-2…) need to be considered?

Arithmetic instruction "i" at the beginning of EX ➔

i-1 → completed EX
i-2 → completed M1
i-3 → completed M2
i-4 → completed WB

Therefore, values computed by i-1 can be passed to i with EX/EX bypass; i-2 hasn't completed any valid data transfer from memory therefore, it cannot bypass any useful values; i-3 completed memory access and required values can be transferred to i with M2/EX bypass, i-4 was at WB when i was at RF, therefore I had read values modified by i-4 directly from register file without any bypass requirement. Consequently, the required bypasses are for instructions <u>i-1</u> and <u>i-3</u>.

Describing using the pipeline diagram:

```
i-4:  IF | ID | RF | EX | M1 | M2 | WB
i-3:       IF | ID | RF | EX | M1 | M2 | WB
i-2:            IF | ID | RF | EX | M1 | M2 | WB
i-1:                 IF | ID | RF | EX | M1 | M2 | WB
i  :                      IF | ID | RF | EX | M1 | M2 | WB
```

b) (8 points) How many load delay slots would you need in order to completely hide the delay of a load-use hazard? Why?

To hide the delay of load-use hazard, we must be able to bypass the acquired value from the memory to the beginning of EX stage. Therefore, referring to part (a), if the using instruction is i^{th} instruction, the producing instruction should be at least 3 instructions ahead as i-3, to provide M2 → EX bypass. Or, there should be <u>2 delay slots</u>. Once again, referring to the pipeline diagram:

```
ld :          IF | ID | RF | EX | M1 | M2 | WB
use:               IF | ID | RF | -- | -- | EX | M1 | M2 | WB
```

c) (19 points) A hot-shot young chip designer points out that arithmetic instructions (which don't need the MEM stages) could complete sooner if they were allowed to write-back after the EX stage. That is, load and store instructions would use the full pipeline given above, while simple arithmetic instructions (as well as control instructions) would use the shorter pipeline shown below:

IF	ID	RF	EX	WB

Name at least 3 issues the ID phase must address in order for such a pipeline to work properly. Give concrete examples to illustrate the issues.

```
Arithmetic and Control:   IF | ID | RF | EX | WB
load / store          :   IF | ID | RF | EX | M1 | M2 | WB
```

1) Prevent WAW data hazards:
<u>ex:</u>

```
ldw R1, 0(R5) :   IF | ID | RF | EX | M1 | M2 | WB
add R1, R2, R3:        IF | ID | RF | EX | WB
```

R1 is written by both the load and add instructions and due to instruction sequence, the final value should depend on the add instruction, but if not avoided by control logic, load writes the R1 value later in time

2) Prevent structural hazards:
<u>ex:</u>

```
ldw R1, 0(R5) :   IF | ID | RF | EX | M1 | M2 | WB
and R5, R0, R8:        IF | ID | RF | EX | WB
add R1, R2, R3:             IF | ID | RF | EX | WB
```

There will be a conflict on resources as both instructions try to write to register file, through a single write port.

3) Additional control logic / bypass for arithmetic instructions to skip MEM stages:
<u>ex:</u>

arithmetic instruction: control should bypass the mem stages and should choose the correct input from the EX/M1 latch to write to register file

load/store instruction: control should enable the dataflow through mem stages as in the original pipeline

4) Maintenance of precise exceptions under out of order writes:
<u>ex:</u>

```
ldw R1, 0(R5) :   IF | ID | RF | EX | M1 | M2 | WB
and R5, R2, R8:        IF | ID | RF | EX | WB
```

Precise exceptions require the machine to complete the instructions before the instruction that caused the exception and to flush the ones following. After the exception is handled, the preserved machine state should restart execution from the exception pointer.

In the above example, in case of an exception during WB of ldw, precise exception requirement suggests flushing of pipe, handling the exception and restarting the execution from lw. However, as long as out of order completion is permitted, and instruction has already written the source operand of ldw (R5) and when the execution resumes, ldw fetches from wrong memory location. Hence, this is not a WAW or WAR hazard. (not a WAW because ldw and add write to different registers; Regarding WAR: In normal execution ldw reads the correct R5 during it RF (or bypassed) while and hasn't yet updated it.) Therefore, ID phase should assign the control signals such that the written values are buffered until all previously issued instructions complete.

5) (25 points) Instruction Set Design & Hardware/Software Interface In your first homework assignment, you solved a programming assignment involving a single-instruction computer (SIC) whose only instruction was "sbn" (Subtract and branch if negative). If you were in charge of writing a compiler to compile C code to the SIC instruction set, explain how you would handle all aspects of a C language procedure call. (List the key bits of functionality a compiler/processor must deal with regarding a procedure call, and for each, say how one would do it with this SIC.)

Recall that for the instruction:

```
sbn a,b,c
```

the functionality is:

```
Mem[a] = Mem[a] – Mem[b];
if (Mem[a] < 0) go to c;
```

Key steps in a procedure call normally include:
- Pass parameters via appropriate registers or on stack. (May involve adjusting a stack pointer)
- Save away any caller-saved registers
- Store the return address someplace save (register or on stack)
- Jump to the callee
- Save away any callee-saved registers
- Perform the procedure
- Place return value (if any) in appropriate spot
- Return to callee by restoring return address and jumping to it.

You got about 10 points for stating a general list along these lines. The next ten points were for starting to talk about things specific to this "SIC" instruction set. Namely, with this instruction set:
- There are no general-purpose registers, so no need to save away callee- or caller-saved registers… You may need to copy around variables in memory though, depending on how you assumed the two procedures coordinated their use of memory.
- Regarding parameter passing and the stack: Most solutions worked for a fixed point in memory presumed to be the "stack" location. This works as long as each procedure has its own safe spot in memory and nobody else ever writes over it. It does not work for recursion.
- Since c is a constant, you can only jump to fixed addresses, not those stored in memory. This is a big constraint on procedures. Kudos to those who noticed this. With any "normal" solution, you will end up with a fixed return value (something the compiler places there for c) which means that you can only call this procedure from one spot since you can only return back to one spot.
- There are ways to get to multiple return points, by setting what is effectively a big "case/switch" statement so that the compiler could statically lay out several return point pc options depending on some value set at call time.

You got about 20 points for clearly & completely laying out a basic implementation that worked for single call point, single return point, no recursion… The remaining 5 points were for noticing and suggesting ways to address problems like recursion, managing the stack, etc.

The actual coding was in many ways secondary to your observations about what this instruction set could or couldn't do. But for example, if you assumed (ok) that you had one memory address d initialized to the value one, then you could use building blocks like the ones below to load, store, change control. Add is the opposite of subtract, multiply is repeated addition, and so forth.

Call:
sbn a, a, +1
sbn a, d, proccall

Return:
sbn, a, a, +1
sbn a, d, returnaddr

Save val in Memory[a]:
Sbn a, a, +1
Sbn val, 0, +1