SPRING 1993
COMPUTER SCIENCES DEPARTMENT
UNIVERSITY OF WISCONSIN—MADISON
PH. D. QUALIFYING EXAMINATION

Computer Architecture
Depth Examination

Monday, February 8, 1993
3:00 – 7:00 PM
113 Psychology

GENERAL INSTRUCTIONS:

1. Answer each question in a separate book.
2. Indicate on the cover of *each* book the area of the exam, your code number, and the question answered in that book. On *one* of your books list the numbers of *all* the questions answered. *Do not write your name on any answer book.*
3. Return all answer books in the folder provided. Additional answer books are available if needed.

SPECIFIC INSTRUCTIONS:

Answer *all* of the following *seven* questions. The questions are quite specific. If, however, some confusion should arise, be sure to state all your assumptions explicitly.

POLICY ON MISPRINTS AND AMBIGUITIES:

The Exam Committee tires to proofread the exam as carefully as possible. Nevertheless, the exam sometimes contains misprints and ambiguities. If you are convinced a problem has been stated incorrectly, mention this to the proctor. If necessary, the proctor can contact a representative of the area to resolve problems during the *first hour* of the exam. In any case, you should indicate your interpretation of the problem in your written answer. Your interpretation should be such that the problem is nontrivial.

(1) **Floating-Point Numbers**

Consider the IEEE 754 floating-point standard 32-bit, single-precision format. Recall that this format has a sign bit, 8 bits for a biased exponent (the bias is 127), and 23 bits for the fraction (mantissa). There is an implied leading bit (or hidden bit).

(a) What are denormalized numbers? How are they represented in this format?

(b) Show an arithmetic operation on normalized numbers that results in a denormalized number.

(c) Which of add, subtract, multiply, and divide, on normalized numbers, can result in a denormalized number?

(d) What are the implications of denormalized numbers for the hardware design of the floating-point unit? Why?

(2) **Vector Registers**

Several machines implement vector instructions. Some, including the Cray-1, use vector registers.

(a) What are vector registers? How are vector instructions implemented in machines *without* vector registers?

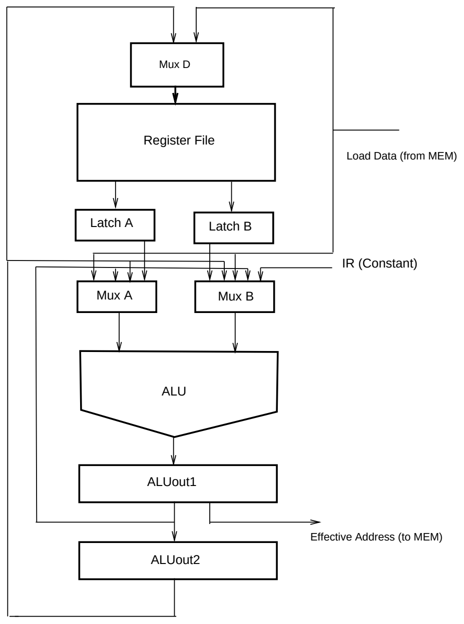
(b) How does the tradeoff between using and not using vector registers change as latency to main memory increases? Why?

(3) **Pipeline Hazards**

Consider the 5-stage DLX pipeline below (figure 6.9 from Hennessy and Patterson, but with load ports in muxes) Show how the following instruction sequence flows through the pipeline by filling in the table on the next page. **Be sure to hand in the table.**

format: opcode rd, rs1, rs2

ADD	r1, r2, r3
AND	r1, r1, r4
LOADW	r2, r1, #32
SUB	r4, r2, r1
OR	r5, r1, r2



For the IF, ID, EX, MEM, and WB fields, write in the opcode of the instruction currently in that stage. If a bubble is introduced by a stall, write ‘‘bubble’’.

For RA and RB, write in the register number (e.g., r2) that is being read on the A (B) port of the register file. Registers are read during the ID stage. Assume that registers are written (WB) before being read (ID).

For RD, write in the register number (e.g., r2) that is being written to the destination port of the register file. Registers are written during the WB stage.

For ALU, write the ALU operation (e.g., ADD, AND, etc.) occurring during that cycle.

For MUXA and MUXB, specify which multiplexor input should be selected: RA, RB, ALUout1, ALUout2, LoadData, IR.

For MUXD, specify which multiplexor input should be selected: ALUout2 or LoadData.

If the pipeline stalls in cycle *i*, mark the STALL field in cycle *i* to indicate that the *pipeline does not advance*. You should still fill in the other required fields. If the pipeline stalls, explain why in the **NOTES** section below the table.

If a field depends upon a previous or subsequent instruction (i.e., an instruction that is not a part of the 5 instruction sequence above), mark it with a ’-’. If the field is a "don’t care", then mark it with an X.

Student Number ________________________ **CA Qualifying Exam** **Question 4**

BE SURE TO HAND IN THIS PAGE

Cycle	IF	ID	EX	MEM	WB	RA	RB	RD	ALU	MUXA	MUXB	MUXD	STALL
i	ADD	-	-	-	-	-	-	-	-	-	-	-	-
i+1	AND	ADD	-	-	-	r2	r3	-	-	-	-	-	-
i+2													
i+3													
i+4													
i+5													
i+6													
i+7													
i+8													
i+9													
i+10													

NOTES:

(4) **ALU Design**

Background Superscalar processors take a sequential instruction stream and try to execute multiple instructions in parallel. Some superscalar processors issue multiple instructions in the same cycle only if the candidate instructions are of different types. For example, the IBM RS/6000 can issue one integer, one floating point, one ‘‘conditional’’, and one branch instruction in the same cycle. Other superscalar processors can issue two instructions of the same type if and only if they are independent. For example, some processors can execute two integer instructions in the same cycle if and only if there is no data dependence between the two instructions.

However, these restrictions limit the effective parallelism, and hence performance, that these processors can achieve. To improve performance, some recent superscalar processors can issue two (or more) instructions of the same type, even though they are dependent. For example, the TI SuperSparc can issue the following two instructions in the same cycle:

```
ADD    r1, r2, r3      i.e., r1 <-- r2 + r3
ADD    r4, r1, r5      i.e., r4 <-- r1 + r5
```

Note that the destination register of the first instruction (r1) is a source register for the second instruction.

The Problem In this problem, you will design an 8-bit arithmetic unit capable of:

1) Executing two independent add operations, or
2) Executing two dependent add operations where the first source operand of the second instruction is the result of the first instruction (as above).

The arithmetic unit has four 8-bit inputs (A1, A2, B1, B2), two 8-bit outputs (S1, S2), two carry-outs (CO1, CO2), and a 1-bit input control signal (ADEP) which indicates that a dependence exists. If the operations are independent, then (C01, S1) = A1 + B1 and (CO2, S2) = A2 + B2. If ADEP is asserted (equal 1) then the result of the first add operation (S1) should be used in place of input A2 to the second add operation. Note that S1 and CO1 must still be computed even if there is a dependence.

The components you have to use are:

(i) A fast carry-lookahead adder that takes two 8-bit inputs, a carry-in, and produces an 8-bit output and a carry-out. The delay from any input to any output is 8 gate delays.

(ii) An 8-bit wide 2:1 multiplexor that, based on a select input, selects one of two data inputs to pass to the output. The delay from any input to any output is 2 gate delays.

(iii) An 8-bit wide 3:2 carry save adder that takes three 8-bit inputs and reduces them to two 8-bit outputs. The delay from any input to any output is 2 gate delays.

(iv) Two-input AND, OR, NAND, and NOR gates, plus an inverter. The delay from any input to any output is 1 gate delay.

Design the arithmetic unit using these parts (you may use as many of each type as necessary). You will be graded first on correctness, second on minimizing the critical path, and finally on gate count. Be sure to state the critical path through your circuit.

(5) **Disk arrays.**
Assume you wish to implement a disk array of `n+c` disks that can *recover data* after the failure of `i` disks. Consider only one `n`-bit data word `x<n-1,0>` where one bit of `x` is stored on each of `n` disks and `c` check bits are stored on the other `c` disks.

(a) Let `i=1`. What is the *minimum* value for `c`? Why? Show how bit `x<0>` is recovered if its disk fails.

(b) Let `i=1`. Assume all disks are working and a user wishes to write just bit `x<0>`. What is the *minimum* number of disk writes your system must do? Why?

(c) Let `i=3`. Assume all disks are working and a user wishes to write just bit `x<0>`. What is the *minimum* number of disk writes your system must do? Why?

(6) **WAR and WAW hazards.**
Consider designing a machine that uses all kinds of aggressive, dynamic techniques to make its peak IPC (instructions per cycle) much greater than one. Your job is to design the general-purpose register file and associated logic so that instructions rarely stall due to write-after-read (WAR) and write-after-write (WAW) hazards on general purpose register accesses.

Discuss your design options and the tradeoffs between them.

(7) **Request Combining**
One of the most important characteristics of the NYU Ultracomputer is its ability to do *request combining*.

(a) What is request combining?

(b) Why do the Ultracomputer people consider request combining to be central to the design of a large-scale parallel computer?

The Thinking Machines CM-5 supports a limited form of request combining as a part of its control network. The control network is a simple binary tree (actually a 4-ary tree but ignore this detail). Requests to be combined are submitted into the (separate) control network. *All* processors must participate in a combining operation. Combining takes place as follows. *Each* processor submits a value to be combined into the network and then waits for its result. The network carries out the requisite operation, and generates the results for the individual processors. Results are returned to the processors only when they are guaranteed to be valid. That is, *all* processors have submitted their requests into the network, and sufficient time has past since the last request was submitted so that the network outputs are valid.

Compare the limited form of combining of the CM-5 to the more general approach of the NYU Ultracomputer.

(c) Discuss the pros and cons of the two approaches including: (i) hardware cost and complexity, and (ii) functionality and usability

(d) Under what conditions, workload, design constraints, etc., would you pick one over the other.