Name: The Answers

NOTE: These solutions are for use only for CMSC 611, Spring 2000 at UMBC.
Other uses prohibited!

CMSC 611 Midterm

Prof. Ethan Miller

April 4, 2000

This exam is 5 pages long and has 7 questions (plus 1 bonus).

This is an open book exam. You may use your notes and other references, but you may not share your reference material with others in the class. Your answers *must* be in your own words — copying verbatim from reference material will result in a 0 for that question. You must show all of your work — partial credit may be given for partially correct answers, while answers with no justification may not receive full points. You may use the back of the exam sheets if you need extra space.

Please read over the entire exam ***before*** starting.

IMPORTANT: please put your name at the top of ***each page***. Any page without a name on it will not be graded.

1. [10 points]

Briefly explain the difference between the VLIW and superscalar design approaches, listing two advantages for each approach. Which approach would you use for an processor to be used in a handheld device, and why?

Superscalar designs issue multiple instructions per cycle and choose their instructions from a “window” of available instructions. The range of instructions that can be issued in one cycle is fixed by the available functional units. VLIW, on the other hand, uses a long instruction “word” that contains “instructions” for each of the available functional units. If a functional unit isn’t needed in a particular long instruction, it’s wasted for that cycle.

Advantages for a VLIW design:

- Dependency analysis is done statically at compile time, drastically reducing hardware complexity and thus CPU size.
- VLIW processors can often run faster because they don’t check dependency.

Advantages for superscalar:

- Superscalar CPUs are backwards compatible with each other as well as with previous (non-superscalar) designs.
- Superscalar CPUs can take advantage of dynamic changes in ILP; for example, they can dynamically schedule the full use of functional units across branches. They can also dynamically resolve potential hazards; VLIW can’t do this as well.

I probably wouldn’t use either for a handheld device (they’re kind of overkill), but I’d choose VLIW if I had to choose one. While superscalar is far more flexible, VLIW would almost certainly consume less power and be less expensive. Since a handheld usually runs only a few programs (and isn’t often used for programming), the overhead of recompilation would be low.

2. [10 points]

Many processors issue multiple instructions per cycle to increase throughput. Why not build a processor that issues 32 instructions per cycle? What limits instruction-level parallelism in processors? List two distinct problems that limit ILP even if the number of available transistors is virtually unlimited.

The biggest limit to ILP is control hazards. The complexity in speculating through multiple levels of branches is extremely high. One or two levels of branches can be speculated across, but speculating across 3+ levels of branches is both complex and ineffective. The problem with all this speculation is that only one path is correct. Thus, speculating across 3 levels of branches means that 7/8 of the work is wasted. Speculating across 5 levels of branches wastes 31/32 of the work (only 1 of 32 paths is correct!), so it takes 64 functional units to produce the same benefit as one level of speculation with just 4 functional units. The difficulty gets exponentially worse, too.

A second limit to ILP is that it’s often difficult to find 32 (or 64, or more) independent instructions to execute at the same time, given the limited number of registers that the program can use. While there might be more parallelism available, it’s very difficult to see it because of register reuse and branches.

A third limit to ILP is instruction & data bandwidth to memory. More instructions means more memory accesses, and the CPU simply doesn’t have enough bandwidth to the outside world. Even if the number of transistors is high, the number of pins on a CPU is going to be limited by die size.

3. [25 points]

This question deals with the following DLX code. Assume that R4 and R5 are initialized to point to a[0] and b[0], respectively, before entering the loop.

```
            LD   F0, 0(R8)    ; R8 points to the variable z
            ANDI R2,R2,#0
            SUBD F2,F2,F2     ; set F2 = 0
loop:       LD   F4,0(R4)
            LD   F6,0(R5)
            MULD F4,F4,F6     ; F4 = F4 * F6
            ADDD F2,F2,F4
            ADDD F4,F4,F0
            SD   0(R4),F4
            ADDI R4,R4,#8
            ADDI R5,R5,#8
            ADDI R2,R2,#1
            SUBI R6,R2,#n     ; R6 = R2 - n
            BNEZ R6,loop
            NOP
            SD   0(R7),F2     ; R7 points to the variable x
```

The DLX processor running this code uses the standard 5-stage DLX pipeline with a single branch delay slot. The pipelined FP multiply unit requires 5 EX cycles, while the pipelined FP adder requires 4 EX cycles.

[a] What C code does the above loop implement?

```
for (i = 0; i < n; i++) {
  x += a[i] * b[i];
  a[i] = a[i] * b[i] + z;
}
```

[b] How many cycles (including stalls) will the above code require to do a single iteration of the loop?

FP multiply latency is 4 cycles, and FP add latency is 3 cycles. Stalls occur after LD (1), MULD (4), ADDD (3), ADDD (2 - forwarding to an SD instruction). Total cycles = 12 + 10 = 22 cycles per iteration.

[c] Rewrite the loop to eliminate all stalls and NOPs in the loop (don't worry about stalls in the code before `loop:`). All changes are legal as long as the loop still calculates the same result. You may assume that *n* is a multiple of 60. How much speedup do you gain over the original (unoptimized) code?

```
        LD   F0,0(R8)
        ANDI R2,R2,#0
        SUBD F2,F2,F2
        SUBD F18,F18,F18
loop:   LD   F4,0(R4)
        LD   F6,0(R5)
        LD   F14,8(R4)
        MULD F8,F4,F6
        LD   F16,8(R5)
        ADDD F2,F2,F18
        MULD F18,F14,F16
        ADDI R2,R2,#2
        ADDD F4,F8,F0
        ADDD F2,F2,F8
        SUBI R6,R2,#n
        ADDD F14,F18,F0
        ADDI R4,R4,#16
        ADDI R5,R5,#16
        SD   -16(R4),F4
        BNEZ R6,loop
        SD   -8(R4),F14
        ADDD F2,F2,F18
        SD   0(R7),F2
```

This loop calculates the same result, and requires just 17 cycles per two iterations, for a speedup of 44/17 = 2.59. Speedup can be further increased by unrolling the loop more times and amortizing the loop overhead over more instructions.

4. [30 points]

A load-store instruction set allows two “modifiers” usable on either loads or stores. The first modifier allows a second register to be specified; if this modifier is used, the effective address used for the memory operation is Ra + Rb + displacement. The second modifier is the update modifier, where Ra is updated with the effective address computed for the memory operation. The two modifiers may be combined. Examples of the instructions are:

```
LWX      R8,4(R2+R4)      ; Memory[R2+R4+4] => R8
LWU      R8,8(R2)         ; Memory[R2+8] => R8, R2+8 => R2
LWXU     R8,9(R2+R4)      ; Memory[R2+R4+9] => R8, R2+R4+9 => R2
```

[a] What changes would need to be made to the basic DLX hardware to support these changes to memory addressing? Would there need to be any modifications to the pipeline? Assume that each pipeline stage is “full” before the changes — there’s no time for additional operations in any stage unless they happen in parallel with operations that are already there. Also, assume that all ALUs have exactly two inputs and one output.

The DLX pipeline would need an additional write port for the register file to store the updated register value. It would would also need an additional read port so it could read three registers in a single cycle (two index registers and a value to store). The extended addressing modes would require an additional EX stage to calculate the .effective address — having a second ALU wouldn’t help because the result would take too long to calculate, extending the cycle time. This might require another ALU, though, to avoid structural hazards.

In addition, there would need to be more pipeline registers and different forwarding hardware, though the results being forwarded would be pretty similar in timing to those already present.

[b] For each modifier, show an example where using the modifier would save at least one instruction over the original DLX instruction set.

Extended		**Update**		**Extended Update**	
Before	**After**	**Before**	**After**	**Before**	**After**

`ADD R1,R2,R3`
`LW R8,8(R1)`
	`LWX R8,8(R2+R3)`	`LW R8,16(R2)`
		`ADD R2,R2,#16`
			`LWU R8,16(R2)`	`ADD R2,R3,R2`
				`LW R8,16(R2)`
				`ADD R2,R2,#16`
					`LWX R8,16(R3+R2)`

[c] Your measurements of benchmarks using the enhanced instruction set found the following frequencies and timings:

Instruction Type	Frequency	CPI
Integer ALU	60%	1
Branches	15%	2.1
Loads	15% (60% unmodified, 20% X, 15% U, 5% UX)	1.5
Stores	10% (60% unmodified, 20% X, 15% U, 5% UX)	1.1

The designers managed to squeeze the ISA modifications into the original 5 stage DLX pipeline, but they had to reduce the clock frequency from 750 MHz to 650 MHz to accommodate them. What are the relative speeds of the original and enhanced DLX? Assume that instruction CPI for each type didn’t change between the original and enhanced DLX; only the count for each instruction type changed.

We make the assumption that loads and stores that are either U or X become a load/store and *one* ALU instruction. Loads and stores that are UX become a load/store and *two* ALU instructions. Suppose that we originally had a program with 1000 instructions (in the enhanced version). This would include 600 ALU, 150 branch, 90 regular loads, 60 regular stores, 30 LWX, 22.5 LWU, 7.5 LWUX, 20 SWX, 15 SWU, 5 SWUX. Overall CPI is (0.60*1) + (0.15*2.1) + (0.15*1.5) + (0.10*1.1) = 1.25. Thus, it would take 1250 cycles to execute our hypothetical “program”, and it would do so at 650 MHz -> total time is (1250/650) = 1.92 us.

The original version of the CPU would take “longer” to execute the U, X, and UX instructions because it would add ALU instructions to the mix. Rather than just 600 ALU instructions, there would be 600 + 30 + 22.5 + (7.5*2) + 20 + 15 + (5*2) = 712.5, accounting for the added ALU instructions necessary to replace the extended and update instructions. This would increase execution cycles by (712.5-600) * 1 = 112.5 cycles, resulting in a total of 1362.5 cycles. At 750 MHz, this would require (1362.5/750) = 1.82 us.

The unmodified CPU (without X, U, UX instructions) is 1.92/1.82 = 1.05x faster.

5. [10 points]

Modern microprocessors have highly pipelined arithmetic units and can issue 4 or more instructions per cycle. Under these conditions, what benefits would vector instructions offer? Why not simply issue more scalar instructions instead?

- Vector instructions reduce the instruction bandwidth necessary to keep the CPU supplied with instructions. Reducing memory bandwidth should result in better performance.
- Vector instructions allow the dependency calculations to occur once for a sequence of computations, rather than having them occur once per instruction as with scalar instructions.
- Vector instructions can, theoretically, achieve very high levels of parallelism by chaining together four or more vector instructions without need to have that many ports in the register file. Instead, buffers in the vector functional units can allow many vector instructions to operate on a stream of values fetched just once from the register file. This is difficult to do with scalar functional units.
- Vector instructions replace loops in scalar code with single instructions. This reduces the branch count and thus reduces both control overhead and control hazards, reducing execution time further.

6. [10 points]

In a heavily-loaded system (one with many users), a (2,2) branch prediction buffer with 512 entries performs as well as a (4,2) branch prediction buffer with 16K entries (an entry is a single prediction for a particular combination of branch history and location). Give at least two reasons why the larger prediction buffer doesn't result in performance improvement. Assume that the processors are identical except for the branch prediction buffers.

- As noted in the text, looking at a longer history doesn't necessarily get you any more accuracy. Moreover, it would reduce the size of the number of unique branches in the BPB, particularly if individual branches tended to do the same thing each time.
- When there are lots of context switches (as in a heavily loaded system), the BPB would take time to "come up to speed" on the current program's branch pattern. If the time slice is relatively short, the larger BPB wouldn't necessarily get "filled" in one time slice, so a smaller buffer would do just as well.

7. [5 points]

A large national lab wants to run their parallel programs faster, and would like to purchase a high performance computer. They have a program that is 98% parallelizable — 98% of the execution time can be parallelized, reducing the execution time by the number of computers working on the problem. The lab can buy either 16 computers running at 1 GHz or 50 computers running at 500 MHz for the same price. Which configuration is faster?

The speedup over a single 1 GHz CPU for the 1 GHz configuration is 1/((0.98/16) + (0.02)) = 12.31. The speedup over a single 500 MHz processor for the 500 MHz configuration is 1/((0.98/50)+(0.02)) = 25.25. However, the 1 GHz machines are intrinsically twice as fast as the 500 MHz machines. The total speedup for the 1 GHz configuration over a single 500 MHz machine is thus 12.31*2 = 24.62. The 500 MHz configuration is thus slightly faster by a factor of 25.25/24.62 = 1.026.