Name Solution

Computer Architecture
EE 4720
Final Examination

10 May 2005, 15:00–17:00 CDT

Problem 1		(25 pts)
Problem 2		(15 pts)
Problem 3		(20 pts)
Problem 4		(20 pts)
Problem 5		(20 pts)
Exam Total		(100 pts)

Alias Resolution Time

Good Luck!

Problem 1: (25 pts) The `madd.s fd, fr, fs, ft` instruction writes floating-point register `fd` with (`fr` $\times$ `fs`) + `ft`. The `fr` field is in bits 25:21, the other fields are in their usual places. The instruction is used in the code below:

```
# With ordinary instructions:
 mul.s f1, f2, f3
 add.s f1, f1, f4

# With a madd.s instruction:
 madd.s f1, f2, f3, f4
```

(*a*) Add datapath connections (including any connections to the register file) to the implementation on the next page (also shown below) to implement the `madd.s` instruction. The code below should execute as shown (pay attention to `f4`).

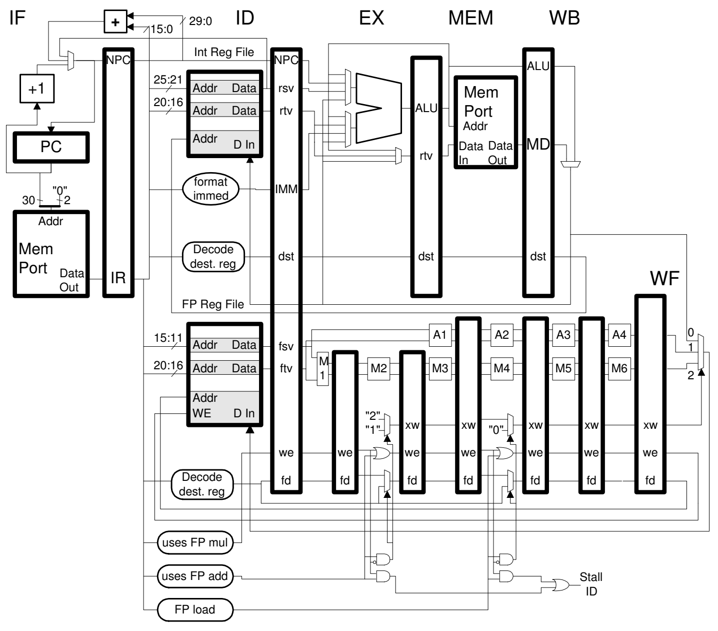
```
 madd.s f1, f2, f3, f4 IF ID M1 M2 M3 M4 M5 M6 A1 A2 A3 A4 WF
 lwc1 f4, 0(r2)           IF ID EX ME WF
 add.s  f5, f6, f7           IF ID A1 A2 A3 A4 WF
```

- Use the existing multiplier and adder.
- It should still be possible to execute ordinary floating point multiply and add instructions.

(*b*) Modify the logic so that `xw`, `we`, and `fd` work correctly for the `madd.s` instruction (and continue to work correctly for existing instructions).

(*c*) Without a `madd.s` instruction there is no possible structural hazard on `WF` in the implementation below for multiply because multiply is the longest latency instruction implemented. With `madd.s` that's no longer true; add control logic to detect this new structural hazard and generate the Stall ID signal.

USE NEXT PAGE FOR SOLUTION.



USE NEXT PAGE FOR SOLUTION.

Problem 1, continued:

- [✓] Datapath for `madd.s`, don't break `add.s` or `mul.s`.
- [✓] Code on previous page must run as shown.
- [✓] Modify `xw`, `we`, and `fd` for `madd.s`, don't break other instructions.
- [✓] Detect and handle new structural hazard when `mul.s` in ID.

Solution on next page.

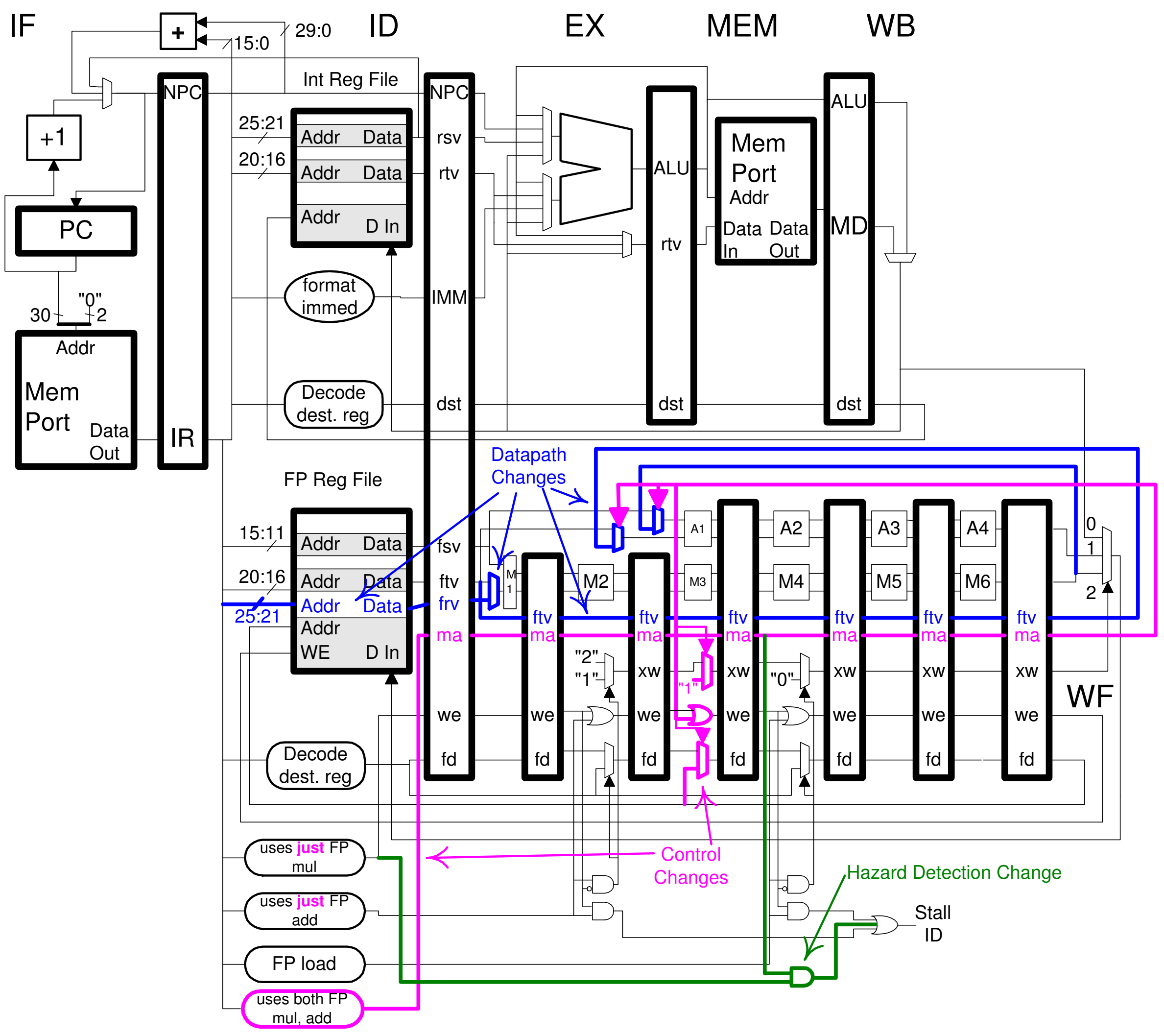
Solution shown below.

Part(a): Datapath changes are shown in blue. A third read port was added to the floating-point register file to retrieve the fr value (`frv`). For the `madd.s` instruction the `frv` is used for the multiply, a multiplexor is added in M1 to handle that. Register value `ftv` is sent down the pipeline in a new set of latches for use by the add in a second pass through the pipeline. The first add stage has new multiplexors to select `ftv` and multiply result from the WF stage.

In the code sample on the previous page a `lwc1` writes `f4`, in the illustrated implementation that's no problem. There would be incorrect results if an implementation did not read the `ft` register until the cycle before it was needed, when `madd.s` reaches M6. That's what "pay attention to f4" was warning about.

Control logic changes are shown in purple. A new logic block, $\boxed{\text{uses both PF mul, add}}$, detects multiply add instructions. The existing multiply- and add-unit detection blocks now (and perhaps always did) only assert their outputs if an instruction just does a multiply or just does an add. A `ma` (multiply-add) signal is sent down the pipeline and used to inject the product, `ftv`, `fd` (the destination register number), the output mux signal (`xw`), and write enable (`we`) into the A1 stage when `madd.s` reaches WF the first time. After that the `madd.s` looks like an ordinary FP add to the pipeline.

The logic to detect the new hazard appears in green. If a `madd.s` instruction is in M4 (it's first trip) and an ordinary multiply is in ID a stall signal is generated.



Problem 2: (15 pts) The execution of a MIPS program on a dynamically scheduled system using method 3 appears below. Complete the ID Register Map, Commit Register Map, and Physical Register File tables for registers `f2` and `f4`.

- The initial value of `f2` is 2.0, the `mul.d` writes 2.1, `ldc1` writes 2.2, and `sub.d` writes 2.3. The initial value of `f4` is 4.0, the `add.d` writes 4.1. Make up physical register numbers as needed.

[✓] As always, show table contents.

[✓] Show where registers are removed from and placed back in the free list.

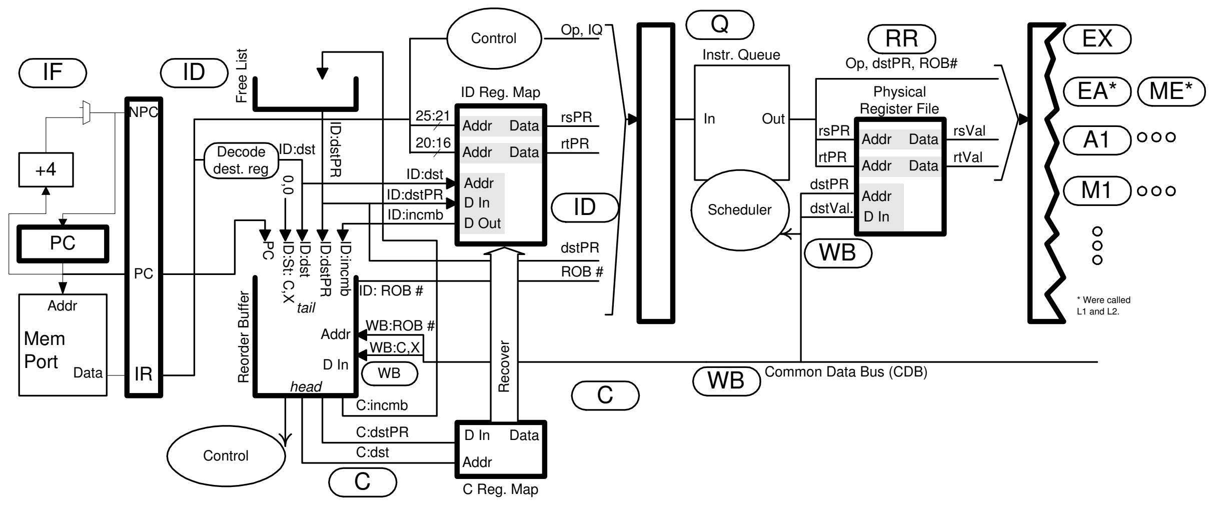
[✓] Show initial values.

The solution shown below, there is nothing interesting about this problem.

```
 # Cycle            0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
 mul.d f2, f4, f6   IF   ID   Q    RR   M1   M2   M3   M4   WF   C
 add.d f4, f2, f10       IF   ID   Q                   RR   A1   A2   A3   WF   C
 ldc1 f2,0(r1)                IF   ID   Q    EA   ME   WF                            C
 sub.d f2, f2, f8                  IF   ID   Q    RR   A1   A2   A3   WF                  C

 # Cycle            0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
ID Register Map

 # Solution
 F2       12             9         71   99
 F4       18                  51

 # Cycle            0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
Commit Register Map

 # Solution
 F2       12                                                     9                   71   99
 F4       18                                                                    51

 # Cycle            0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
Physical Register File

 # Solution
 12          2.0                                                 ]

 18          4.0                                                                ]

  9                      [                                  2.1                      ]

 51                           [                                            4.1

 71                                [                   2.2                                ]

 99                                     [                             2.3

 # Cycle            0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
```

HARDWARE SHOWN ON NEXT PAGE.

Problem 2, continued: Dynamically scheduled processor shown for reference.



Problem 3: (20 pts) The code below runs on three systems, one using a bimodal predictor (B) with a 2^{10}-entry BHT, a local predictor (L) with a 2^{10}-entry BHT and a 16-bit local history, and a global predictor (G) with a 16-bit GHR. The outcomes of B2 are shown but not B1's so **pay attention to where B1's outcomes are located**.

```
LOOP:
SHORT:
# B1 Iterates 10 times
B1: bne r3,r4 SHORT
addi r4, r4,1
...
B2: beq r1, r2 SKIP     N    N    T    N    T    N    N    T    N    T ...
...
SKIP:
j LOOP
nop
```

(*a*) Find the prediction accuracy of each predictor on each branch after warmup. Briefly explain.

✓ Accuracy predicting B1 on B: 90%

✓ Accuracy predicting B1 on L: 100%

✓ Accuracy predicting B1 on G: 100%

✓ Accuracy predicting B2 on B: 60%

✓ Accuracy predicting B2 on L: 100%

✓ Accuracy predicting B2 on G: 80%

(*b*) Determine the number of table entries used by branch B2 on each predictor:

✓ Entries for B2 on B: 1 BHT.

✓ Entries for B2 on L: 1 BHT + 5 PHT.

✓ Entries for B2 on G: 1 BHT + 2 PHT.

(*c*) Suppose the history sizes were reduced.

✓ What is the smallest local history size for which the accuracy on B2 will be unchanged (from the previous answer) when using the local predictor. Briefly explain.

Four bits. With three bits a local history of NTN would appear before both a T and N outcome so the PHT entry counter will mis-predict the T, the N, or even both. With four bits all outcomes can be distinguished.

✓ What is the smallest global history size for which the accuracy on B2 will be unchanged when using the global predictor. Briefly explain.

Eleven bits. For the global history size of 16 used above the GHR holds (global predictor can "see") just one previous outcome of B2. At 11 bits the GHR still holds that previous outcome, at 10 bits is does not and so prediction accuracy will be reduced.

Branch B1 is a simple ten-iteration loop, so there will be nine taken outcomes followed by a not-taken. The diagrams below show the counter values, predictions, and prediction outcomes using a bimodal predictor on branches B1 and B2. Prediction accuracy is computed over a range of cycles that will repeat. For B1 that is cycles 5-10, for B2, that is 0-5 (and 5-10).

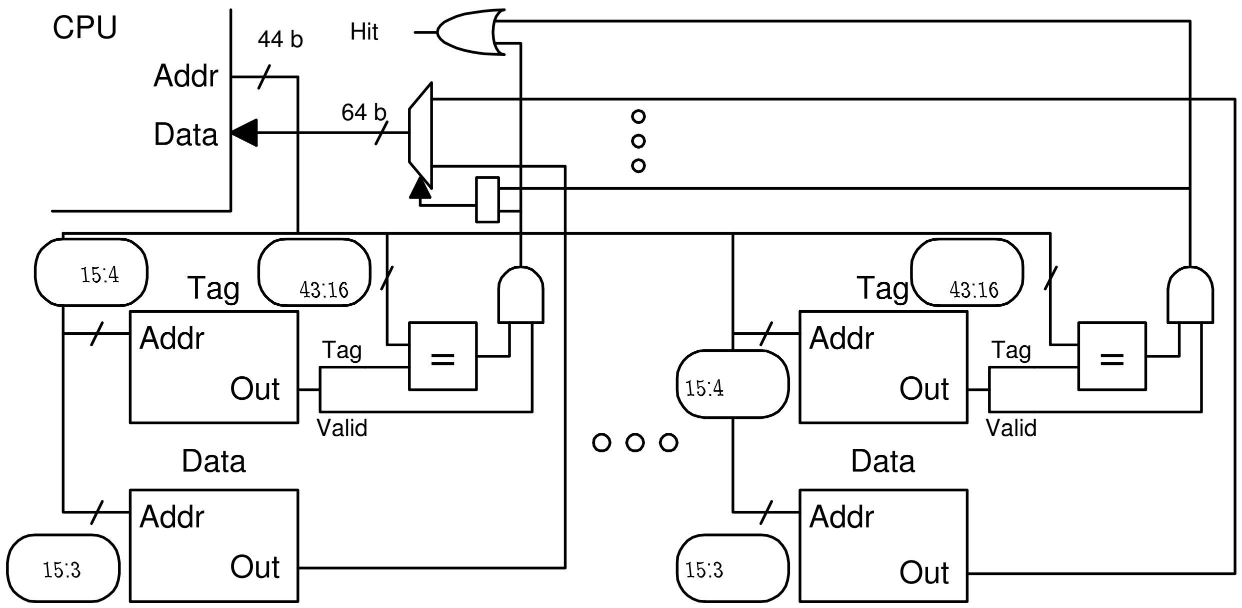
```
# Analysis for computing bimodal prediction accuracy on B1.

     # B1 Iteration  0   1    2    3    4    5    6    7    8    9      10
     # Counter       0  1   2   3    3     2   3    3    3    3       2
     # Prediction     n   n   t   t      t   t    t    t    t    t      t
     # Mispredict       x   x              x                          x
B1: bne r3,r4 SHORT    T   T   T    T ... N   T    T    T    T ...   N
addi r4, r4,1
...


# Analysis for computing bimodal prediction accuracy on B2.

     # B2 Iteration   0   1    2    3    4    5    6    7    8    9     10
        # Counter     0   0    0    1    0    1    0    0    1    0    1
     # Prediction      n   n    n    n     n   n    n    n    n    n    n
     # Mispredict                  x         x              x         x
B2: beq r1, r2 SKIP     N    N    T    N    T    N    N    T    N    T ...
```

The local predictor will predict both with 100% accuracy because their patterns are each less than 16 outcomes. The global predictor will predict B1 with 100% accuracy because the GHR will hold more than the last 9 iterations of B1, which is enough to predict with 100% accuracy. For B2, the GHR will hold only one previous outcome of B2 (the other 15 bits are outcomes of B1 which do not help in predicting B2). Therefore two PHT entries are used with the global predictor predicting B2, one for the previous outcome being taken and one for the previous outcome being not taken. After warmup the not-taken entry will predict taken (because that's the outcome 2 out of 3 times it is used, per iteration) and be wrong once. The taken entry will predict not taken and always be right. Therefore the prediction accuracy will be 80%.

Problem 4: The diagram below is for a 256-kiB (2^{18} byte) 4-way set-associative cache with a line size of 16 characters on a system with 8-bit characters. (20 pts)

(a) Answer the following, formulæ are fine as long as they consist of grade-time constants.

✓ Fill in the blanks in the diagram. *Pay attention to the width of the CPU address port.*



✓ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

	Tag	Index	Offset	
Address:				
	43 16	15 4	3 3	2 0

✓ Memory Needed to Implement (Indicate Unit!!):

It's the cache capacity plus $4 \times 2^{16-4}\,(44 - 16 + 1)$ bits.

✓ Show the bit categorization for a direct mapped cache with the same capacity and line size.

	Tag	Index	Offset	
Address:				
	43 18	17 4	3 3	2 0

(*b*) The code below runs on the same cache as the first part of this problem. Initially the cache is empty; consider only accesses to the array.

✓ What is the hit ratio running the code below?

On the first `j` iteration the hit ratio is 0.5 because the line size is only 16 characters and a data element is 8 characters. The number of `i` iterations is large enough so that the array does not fit in the cache and so on the second `j` iteration the hit ratio will also be 0.5, so the overall hit ratio is, of course, $\boxed{0.5}$.

```
double sum = 0.0, *a = 0x2000000;  // sizeof(double) = 8 characters
int i, j, ILIMIT = 0x1000000;

for(j=0; j<2; j++)
  for(i=0; i<ILIMIT; i++)
    sum += a[ i ];
```

The code in the problems below is to be run on a cache of unknown configuration, though it will be one of the types discussed in class (direct mapped or set-associative). In all cases the cache is empty when the program starts. Routine `get_miss_count()` *returns the number of cache misses at the time of the call (and does not cause any misses).*

(*c*) Complete the code so that `line_size` is assigned the correct line size based on the number of misses encountered and the nature of the code. Assume that all misses are due to access to `a`.

```
int ILIMIT = 0x10000;
char *a;
int miss_count_before = get_miss_count();
for(i=0; i<ILIMIT; i++) sum += a[i];
int miss_count_during = get_miss_count() - miss_count_before;

int line_size =  ILIMIT / miss_count_during;          // SOLUTION
```

There will be one miss for each line used to cache the array. The total size of the array accessed is `ILIMIT` characters so the line size is that divided by the number of misses.

(*d*) Complete the code so that `associativity` is aptly assigned. Assume that the cache is smaller than 256 MiB (2^{28})(but the exact size is not known), that the associativity is no larger than 64, and that there is a 64-bit address space.

```
int ISHIFT =  28;                                    // SOLUTION
int JSHIFT =  28;                                    // SOLUTION
char *a;  // Pointer to a really large array.
int miss_count_before = get_miss_count();
for(i=0; i<64; i++) sum += a[ i << ISHIFT ];
for(j= 63; j>=0; j--           )                     // SOLUTION
   sum += a[ j << JSHIFT ];

int miss_count_during = get_miss_count() - miss_count_before;
int associativity =  128 - miss_count_during;        // SOLUTION
```

Use the `i` loop to load 64 lines with different tags and the same index by setting `ISHIFT` to 28 (since the cache size is no more than 2^{28} bytes). The `i` loop will miss 64 times, once for each iteration. Have the `j` loop check for the lines, starting with the most recent one (`j=63`) which should be a hit. Then check `j=62`, if the cache is direct mapped it will be a miss, otherwise a hit, check `j=61`, if the cache is less than 3-way it will be a miss, etc.

Problem 5: Answer each question below.

(*a*) Suppose in a five-stage statically scheduled MIPS implementation (like the one in class) an instruction could raise an exception in the WB stage. Explain why that would make precise exceptions for that instruction impossible. Use a code example to explain what should happen for a precise exception and why its impossible (or very difficult) if the exception is raised in WB. (5 pts)

If the instruction **after** the faulting instruction were a store it could not be stopped from writing to memory, and so the exception could not be precise. (In a precise exception the handler should start at a point in the program just before the faulting instruction.) In the example below the `jalr` instruction uses an illegal destination register (it can't be the same as the source). If that illegal register were detected in WB rather than ID, then it would be too late to stop the `sw`. When the handler started it would see the effect of the `sw`, which means the `jmpl`'s exception was not precise.

```
jalr r1, r1   IF ID EX ME WB
sw r3, 0(r4)     IF ID EX ME WB
```

(*b*) Arrange the ISA families below in order by code (program) size, the one for which programs are smallest should be first. Starting at the second ISA in the arranged list, explain why code size is larger than the ISA above. (That is, provide three reasons why code size is larger.) (5 pts)

ISA families (in alphabetical order): CISC RISC Stack VLIW

[✓] ISA families in code size order.

[✓] Reasons for size differences.

Smallest: Stack. Unlike CISC and the other ISAs, Stack instructions refer to the stack and so register operands are not used in arithmetic instructions, making them small. In those cases where operands are not in the right place in the stack rearrangement instructions are needed, but even so stack programs are smaller than programs in other ISAs.

CISC. Unlike RISC instructions, CISC instructions are variable size and so they are no larger than they have to be. All RISC instructions are the same size, so some have unused space.

RISC. Unlike VLIW, a RISC program consists only of RISC instructions and other than four-byte (usually) alignment there is little restriction on where instructions can be placed. In VLIW they must be arranged into bundles with restrictions on placement, for example, it might not be possible to put a floating-point instruction in the first slot of a bundle. Because of these restrictions slots may occasionally be filled with nops. The bundles also include template information. In Itanium three RISC-like instructions take 128 bits, a RISC ISA could fit four instructions in the same space. Branches must be to the beginning of a bundle, further wasting space.

Largest programs: VLIW.

(*c*) Deciding on a line size for a cache is a tough decision. (5 pts)

☑ Describe the behavior of programs that run better on caches with smaller line sizes.

The programs should have low spatial locality, meaning if they access an address they probably won't be accessing something nearby anytime soon. Small line sizes help because less space is wasted bringing in data near something which is accessed.

☑ Describe the behavior of programs that run better on caches with larger line sizes.

The programs should have high spatial locality. For example, they might access data sequentially.

(*d*) Answer the following question on cost.(5 pts)

☑ Name two parts of an n-way superscalar processor that cost about n times as much as a comparable scalar processor.

The fetch hardware. The integer ALUs. In each case no more than n of them are needed.

☑ Name two parts of an n-way superscalar processor that cost about n^2 times as much as a comparable scalar processor.

The control logic for detecting dependencies because the source of each of n instructions being decoded must be compared with the destinations of other instructions in the same stage, and n instructions in each of the next few stages. That's proportional to n^2 comparisons.

The bypass connections. It should be possible to bypass results to any of the n instructions in EX; each ALU input will have a multiplexor connecting to n pipeline latches in each of the next two stages (for the five stage design used in class).