Name Solution

Computer Architecture

EE 4720

Final Examination

14 May 2003, 15:00–17:00 CDT

Problem 1		(20 pts)
Problem 2		(15 pts)
Problem 3		(15 pts)
Problem 4		(20 pts)
Problem 5		(30 pts)
Exam Total		(100 pts)

Alias The Next Spam

Good Luck!

Problem 1: The execution of a MIPS code fragment on a dynamically scheduled machine is shown in the tables and in the labels on the diagram, both on the next page. The tables show the contents of the ID Register Map, Commit Register Map, and the Physical Register File at each cycle. The diagram shows the values on certain wires at certain cycles. For example, $\boxed{4{:}65}$ means that at cycle 4 the labeled wire holds value 65.

The following are functional unit segment labels: Load/store, L1 L2; floating-point add, A1 A2 A3 A4; floating-point multiply, M1 M2 M3 M4 M5 M6; integer, EX. The register maps handle both integer and floating-point registers.

(*a*) Write a program consistent with these tables and labels.(12 pts)

- [x] Show a pipeline execution diagram, be sure to show where each instruction commits.
- [x] Choose consistent instructions.
- [x] Choose consistent registers. If a register number cannot be determined, use a question mark.

(*b*) Complete the tables on the next page as follows:(8 pts)

- [x] Show where registers are added to, "]", and removed from, "[", the free list.
- [x] Show the values on the line marked `X` in the illustration.

Solution is on the next page. Here is how the problem is solved:

Entries in the ID map are used to determine the destination registers (of the instruction in ID at that cycle) and when a physical register is removed from the free list.

Entries in the commit map are used to determine when an instruction commits. When an instruction commits the incumbent register (the same architected destination register used by an earlier instruction) is put back in the free list. The incumbent register is to the left of the committing register in the commit map table. (For example, at cycle 11 the instruction writing physical register 93 commits, the incumbent is 50 [and both are `f12`]). The incumbents are shown in the "X" row of the table, these physical registers are put back in the free list.

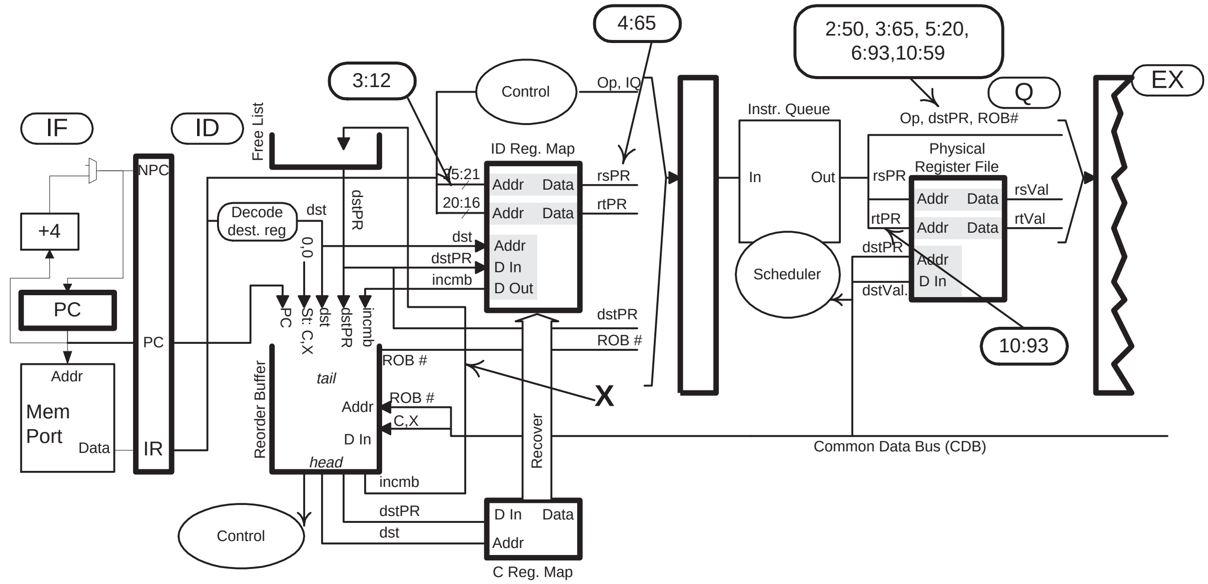
The big label in the diagram ("2:50, 3:65, etc.") shows when instructions are removed from the instruction queue to start execution. Qs are put in the pipeline execution diagram for this.

Entries in the physical register file table show when instructions write back. The time between Q and WB is spent in an execution unit, the amount of time (say four cycles) determines the type of unit and the type of instruction.

The three small labels in the diagram provide information about source registers, two are in ID, one is in Q.

The completed tables are on the next page.

Problem 1, continued: See previous page for instructions.




```
# Cycle               0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
X (Solution)                                 7  3        50 10       93
# Cycle               0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
# Solution
add.s f12, f??, f??   IF ID Q  A1 A2 A3 A4 WC
or   r5, r?, r?          IF ID Q  EX WB       C
add.s f12, f12, f??         IF ID       Q  A1 A2 A3 A4 WC
LwC1 f10, 0(r5)                IF ID Q  L1 L2 WB          C
add.s f12, f??, f12               IF ID             Q  A1 A2 A3 A4 WC

# ID Map       Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
f12    7                 50    93    59
r5     3                    65
f10   10                          20
# Commit Map   Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
f12    7                                   50          93          59
r5     3                                      65
f10   10                                                  20

# Phys. Reg. File  Cy 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
# Solution
3              0                              ]
7              0.                          ]
10             0.                                         ]
20                                [           0.
50                       [                 0.          ]
59                                   [                             0.
65                          [        100
93                             [                       0.          ]
# Cycle               0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
```

Problem 2: The diagram below shows the branch outcome patterns for a branch. (15 pts)

```
BIGLOOP:

B1: 0x1000 beq $t1, $t2, SKIP1    N  N  N  T  T  N  N  N  T  T  N  N  N  T  T
...
B2: 0x1020 beq $v0, $v1, SKIP2
...
    0x2010 j BIGLOOP
```

How accurately would branch B1 be predicted by a bimodal (one-level) branch predictor with a 2^{14}-entry branch history table?

For the version using a 2-bit counter the accuracy is $\frac{2}{5} = 40\%$. See the diagram below.

```
Counter: 0  0  0  0  1  2  1  0  0  1  2  1  0  0  1
B1:        N  N  N  T  T  N  N  N  T  T  N  N  N  T  T
Predict:  N  N  N  N  N  T  N  N  N  N  T  N  N  N  N
Correct:   Y  Y  Y  N  N  N  Y  Y  N  N  N  Y  Y  N

Repeating pattern has two correct, three wrong predictions.
```

How accurately would branch B1 be predicted by a local history predictor with a 10-bit local history and a 2^{14}-entry branch history table?

The repeating pattern can easily be handled by a 10-bit local history so the accuracy is 100%.

What is the minimum local history size needed to predict B1 with 100% accuracy (after warmup). No partial credit without an explanation.

Three outcomes. Just 3. See the table of patterns (NNN, etc.) and predictions.

Pat	Pred
NNN	T
NNT	T
NTT	N
TTN	N
TNN	N

Find a pattern for branch B2 that will reduce the accuracy of the local predictor on branch B1. The branch history table (not to be confused with the pattern history table) remains 2^{14} entries and the history length remains 10 bits.

The pattern below has and extra "T" at the end. So the pattern history table entry for NNNTTNNNTT would be used by both branches, B1 would decrement the entry and B2 would increment it.

```
B2:  N  N  N  T  T  N  N  N  T  T  T   N  N  N  T  T  N  N  N  T  T  T
```

Problem 3: Answer the following load/store unit questions.

(a) Why don't store instructions write to the cache until they commit? (5 pts)

There's a chance they will be squashed, if they wrote to the cache there would be no way (ordinarily) to recover the data that was overwritten.

For the two problems below consider the four instructions (repeated) which are in the reorder buffer of a dynamically scheduled 1-way system. On a cache miss data will arrive in four cycles or more. The effective address for the second load is `0x1000`. (10 pts)

(*b*) Describe a scenario in which the data for address `0x1000` is not cached but the second load does not wait for its data more than a few cycles.

- [x] Show a pipeline execution diagram.
- [x] Explain the involvement of the load/store queue and why the second load does not wait more than a cycle or two.

The effective address of the second load (`0x1000`) is the same as that of the store and so the second load gets its data from the store value (there is no need to check the cache and it doesn't matter that the data is not there). For this to happen the store address must not be delayed (see the next part) which means the first load must hit the cache.

Involvement of the load/store queue: At cycle 7 the LSQ processes the second load. It scans "upward" (toward the head where the older instructions are) looking at each store instruction. The first entry it finds is the store instruction, since the addresses match and the store data is ready the load data will be taken from the store data.

```
# Cycle                      0  1  2 3  4  5  6  7  8
First:  lw $t1, 0($t2)       IF ID Q L1 L2 WB

        addi $t0, $0, 4720      IF ID Q EX WB

        sw 0($t1), $t0             IF ID Q L1 L2 WB

Second: lw $t3, 0($t4)                IF ID Q L1 L2 WB
```

(*c*) Describe a scenario in which the data for address `0x1000` is cached but the second load waits for its data at least four cycles.

- [x] Show a pipeline execution diagram.
- [x] Explain the involvement of the load/store queue and why the second load waits.

In this scenario the first load misses the cache and so the store cannot determine its effective address until cycle 10. Since the store address is unknown the second load cannot safely check the cache (because the store address might match the second load address). At cycle 11 the load can finally be processed. The timing below is correct for a store address of `0x1000` or some other address, either way the load waits until cycle 11 at which time it gets its data.

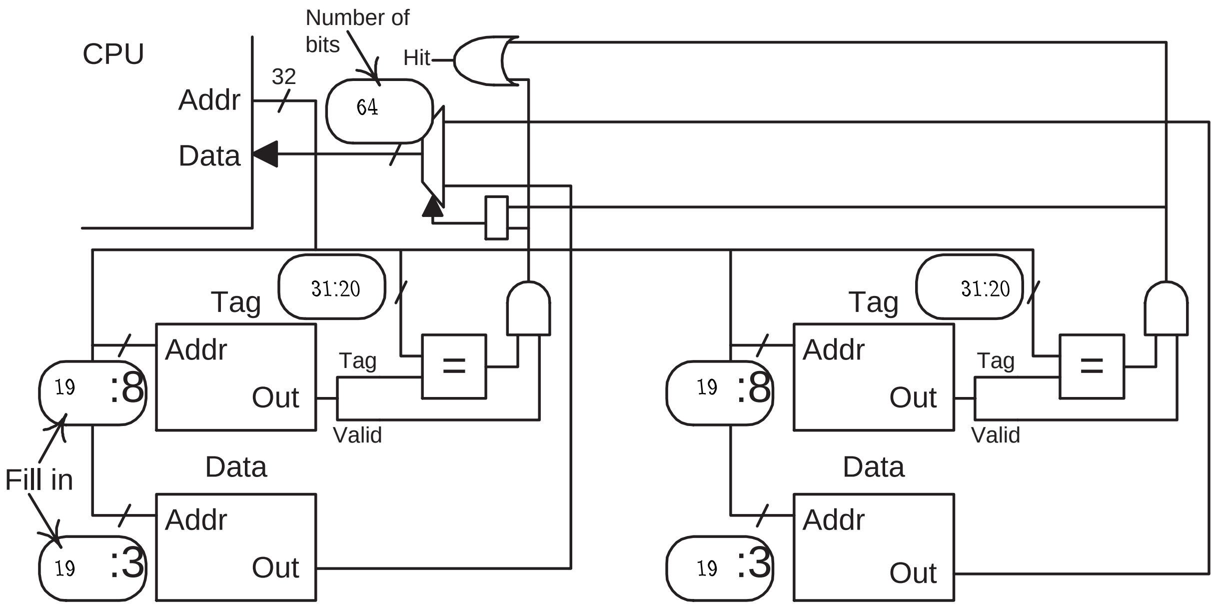
Involvement of load/store queue: The load store queue processes the second load at each cycle from 7 to 11. At cycles 7 to 10 it finds the store with an unresolved address and so the load cannot be completed. Finally at cycle 11 the store address is available. If it matches the second load address the store data is bypassed to the load, otherwise the cache is checked.

```
# Cycle                      0  1  2 3  4  5  6  7  8  9  10 11 12
First:  lw $t1, 0($t2)       IF ID Q L1 L2             L2 WB

        addi $t0, $0, 4720      IF ID Q EX WB

        sw 0($t1), $t0             IF ID Q                L1 L2 WB

Second: lw $t3, 0($t4)                IF ID Q L1             L2 WB
```

Problem 4: The diagram below is for a 2-MiB (2^{21} bytes) cache on a system with 8-bit characters.

(a) Answer the following, formulæ are fine as long as they consist of grade-time constants. (7 pts)

Fill in the blanks in the diagram.



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

Address:

Tag	Index	Offset	
31 … 20	19 … 8	7 … 3	2 … 0

Associativity: 2

Memory Needed to Implement (Indicate Unit!):

It's the cache capacity plus $2 \times 2^{20-8}(32 - 20 + 1)$ bits.

Show the bit categorization for a four-way set-associative cache with the same capacity and line size.

Address:

Tag	Index	Offset	
31 … 19	18 … 8	7 … 3	2 … 0

Problem 4, continued:

(b) The code below runs on the cache from the previous part. When the code below starts running the cache is empty. Consider only accesses to the array. (7 pts)

```
char *a = 0x1000000;  // sizeof(char) = 1 character
int sum, i, j;
int ILIMIT = 1024;

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

☑ What is the hit ratio for the program above?

The element size is one character but the code accesses every other element. The line size is $2^8 = 256$ characters and so $\frac{256}{2} = 128$ will access the same line. The first access will miss, the rest will hit. So for the first j iteration the hit ratio is $127/128$. The total amount of memory accessed is 1024 characters, but that covers 2048 characters of memory because of the skipping. That is much smaller than the 2 MiB capacity so on the second j iteration every access will hit.

The overall hit ratio is $\boxed{\frac{1}{2}\left(\frac{127}{128}+1\right) = \frac{255}{256}.}$

☑ What is the minimum value of ILIMIT needed to fill the cache?

The cache capacity is 2^{21} characters each iteration covers two characters (one read, one skipped) so to fill the cache $\boxed{\texttt{ILIMIT}=2^{20}}$.

Problem 4, continued:

(*c*) The code below runs on the same cache as parts above. Initially the cache is empty; consider only accesses to the arrays. (6 pts)

☑ Choose values for `KLIMIT`, `MLIMIT`, `KSHIFT`, and `MSHIFT` so that array `matrix` is completely removed from the cache with the **minimum** number of accesses (to `b`). That is, each `j` iteration must begin with `matrix` reloaded. Don't forget that the cache is set associative.

The first step is to determine how much space `matrix` takes. Since each element is a 4-character integer and 1024 of them are being accessed it covers 4096 characters or $4096/256 = 2^{12-8} = 16$ lines. The addresses range from `0x1000000` to `0x1000ffc`. Those addresses have indices from `0` to `0xf`. (The index bits start at the third hexadecimal digit.)

To remove `matrix` from the cache accesses to `b` must use each of those indices twice (because the cache is two-way set associative) and with a different tag each time. To generate each index with the minimum number of accesses set `MLIMIT = 0xf;` and `MSHIFT = 8;`, this shifts the `m` into the index part of the address. To generate each tag twice set `KLIMIT = 2;` and `KSHIFT = 20;`, this shifts `k` into the tag part of the address. The solution would also work with `k` and `m` swapped and with a larger `KSHIFT`.

Array `matrix` can be evicted with a small number of accesses because its index bits are the same as `b`, zeros. If index bits 12 or higher were different, say `b == 0x1001000`, it would take many more accesses (using the code below) to evict `matrix`.

Note: The exam was given on the opening day for *The Matrix Reloaded*.

```
int  *matrix = 0x1000000;    // sizeof(int) = 4 characters
char *b      = 0x2000000;
int sum, dummy, i, j, k, m;


// Solution
int KLIMIT =  2;                                  int MLIMIT = 16;

int KSHIFT =  20;                                 int MSHIFT = 8;


for(j=0; j<3; j++)
 {
    for(i=0; i<1024; i++)
      sum += matrix[ i ];

    for(k=0; k<KLIMIT; k++)
      for(m=0; m<MLIMIT; m++)
        dummy += b[ ( k << KSHIFT ) + ( m << MSHIFT ) ];
 }
```

Problem 5: Answer each question below.

(a) Suppose the instructions below are being considered for the latest extension of the MIPS ISA. For each new instruction explain why it should be added or why it should not be added. Consider a *variety* of factors related to the ISA and implementation. (10 pts)

☑ A mask instruction. Based on analysis of benchmarks.

```
New Instruction
 mask $t1, $t2, 5

Equivalent Code Using Existing Instructions
 srl $t1, $t2, 5
 sll $t1, $t1, 5
```

Yes, it saves an instruction and looks easy implement. Saving one instruction may not sound like much but if that one instruction is 5% of the dynamic instruction count the minor addition would be very cost effective.

Grading Note: Some students pointed out that an `andi` instruction can perform the same function and so the `mask` instruction is not needed. That's only partly correct since the immediate value is limited to 16 bits whereas `mask` could mask any number of bits.

☑ An integer negate instruction. Based on analysis of existing code. With this extension `sub` does not have to be used for negation!!!

```
New Instruction
 neg $t1, $t2

Equivalent Code Using Existing Instructions
 sub $t1, $0, $t2
```

Since `sub` does the exact same thing there is no reason to add `neg`. Adding `neg` would waste an opcode and complicate decoding. A better alternative is to have the assembler recognize `neg` as a synthetic instruction and have it substitute `sub`.

☑ An indirect load. Useful based on most existing benchmarks.

```
New Instruction
 lwi $t1, 0($t2)

Equivalent Code Using Existing Instructions
 lw $t1, 0($t2)
 lw $t1, 0($t1)
```

This would require using the MEM stage twice, something no other instruction does and which does not follow RISC principles (which are intended to simplify implementation). It should not be added.

Continued from previous page.

☑ Added functionality for the `sllv` instruction. The existing `sllv` looks at the low 5 bits of the `rt` register, ignoring the other bits. It only shifts left. The improved instruction also shifts right if the `rt` register holds a negative value and left if it's positive.

```
 Improved Instruction
   sllv $t1, $t2, $t3   # Shifts right if $t3 negative.

 Equivalent Code Using Existing Instructions
   bltz $t3, SHIFTRIGHT
   nop
   sllv $t1, $t2, $t3
   j DONE
SHIFTRIGHT:
   sub $at, $0, $t3
   srlv $t1, $t2, $at
DONE:
```

If bidirectional shifts were used a bidirectional shift instruction should be added, but **NOT** by changing the behavior of an existing instruction. There might be programs in which `sllv` instructions execute with a negative value in `rt`. The new behavior would break those programs and the programmers could rightly point out that "ignored" is not the same thing as "assumed to be zero." (And even if they didn't have a good argument it doesn't matter because the customer is always right!) The behavior of `sllv` should not be changed, instead a new instruction could be added.

Grading Note: No one got this one right. Maybe next semester I'll bring in an actual pair of golden handcuffs when talking about ISA compatibility and IA-32. Hmmm, would the student tech fee cover the cost?

(*b*) Answer the following questions about exceptions. (5 pts)

☑ Why are precise exceptions necessary for instructions like `lw` but optional for instructions like `div`?

If an instruction raises a precise exception then it can be re-executed after the handler does whatever it is it has to do. If an instruction raises an exception that is not precise then the best the handler can do is restart the program several instructions after the faulting instruction, there is no way to re-execute it.

In normal use `lw` (and other memory access instructions) will raise exceptions. The exception handler tends to the memory system (updating the TLB or swapping pages) and then restarts `lw` and the program proceeds as if nothing happened. Without precise exceptions there would be no easy way to implement modern memory systems.

A divide would raise an exception because of a division by zero or some other problem. In general, there is nothing the handler can do to fix the problem since the divide instruction gave the best answer it could. With no reason to restart there is no need for precise exceptions. There still may be situations where they are useful, say substituting a large value on a division by zero, but that's not as critical as handling TLB misses.

☑ What can handlers for instructions that raise precise exceptions do that handlers for other instructions cannot? Explain how that capability is used for `lw`.

They can see the state of the program just before the faulting instruction executed. That enables the handler to restart the instruction, if appropriate. It is used to restart `lw` after performing a routine memory system task.

(*c*) Delayed branches are common in RISC ISAs. (5 pts)

☑ Explain why delayed branches were useful in early RISC processors, such as the 1-way statically scheduled MIPS implementation covered in class.

Early RISC processors (and some modern ones too) used short pipelines, say five stages (as used in class). The delay slot instruction is fetched in IF while the branch computes its direction and target in ID; in the next cycle the target or fall-through is fetched. Since the delay slot instruction is executed either way (normally) there is no branch penalty on a taken branch (bubble).

☑ Why are delayed branches of less benefit with more recent implementations?

In a two-way or higher superscalar processor at least one instruction past the delay slot is fetched by the time the branch is resolved. If the branch is mispredicted those instructions will have to be squashed. The number of squashed instructions is one less than without delayed branches, in deeply pipelined systems that might mean squashing 15 instead of 16, the difference is greater with dynamic scheduling. This small benefit is only realized on a misprediction (say 5% of the time). It is not worth the complexity of the control logic.

Comment: Once an instruction is in an ISA it cannot be removed. If a new ISA were designed for compact implementations (say, embedded on a chip with other logic and memory and used to control something like a cell phone) it could use a delayed branch. If an ISA were designed for high-speed implementations (for general-purpose use) then a delayed branch would be omitted.

(*d*) Why might a gshare branch predictor with a longer global history register (GHR) have a significantly longer warm up time? (5 pts)

With a longer GHR there are more possible GHR values for a particular branch. (In an extreme case the branch is in a loop with another branch having random outcomes.) There is a pattern history table (PHT) entry for each GHR value, each of these entries must be warmed up for the branch. The more possible GHR values there are for the branch the longer it takes the GHR to warm up for the branch.

Grading Note: Many answered that it takes longer for the GHR to fill up with branch outcomes. Suppose the GHR were increased from 10 to 16 bits. That change would affect six branches, out of say, 100 million.

A correct answer could start with "It takes longer for the GHR to full up..." but it would have to explain that this was important when entering a loop and that the problem was the irrelevant outcomes from the part of the GHR written before the loop was entered.

(*e*) Why might a functional unit with an initiation interval of 4 and latency of 3 be less costly (as in dollars) than a functional unit with an initiation interval of 1 and a latency of 3?(5 pts)

☐ Include an illustration with your answer.

Sorry, no illustration. If someone asks I'll put one in.

If a functional unit has an initiation interval of 1 then new instructions can enter every cycle and so there must be enough hardware to simultaneously execute 4 instructions. If the initiation interval were 4 then the same small piece of hardware could be used repeatedly for the same instruction, as in a multiplier or divider. Multipliers usually are pipelined (initiation interval 1) but dividers are not.