Name Solution

Computer Architecture
EE 4720
Final Examination

10 May 2007, 7:30–9:30 CDT

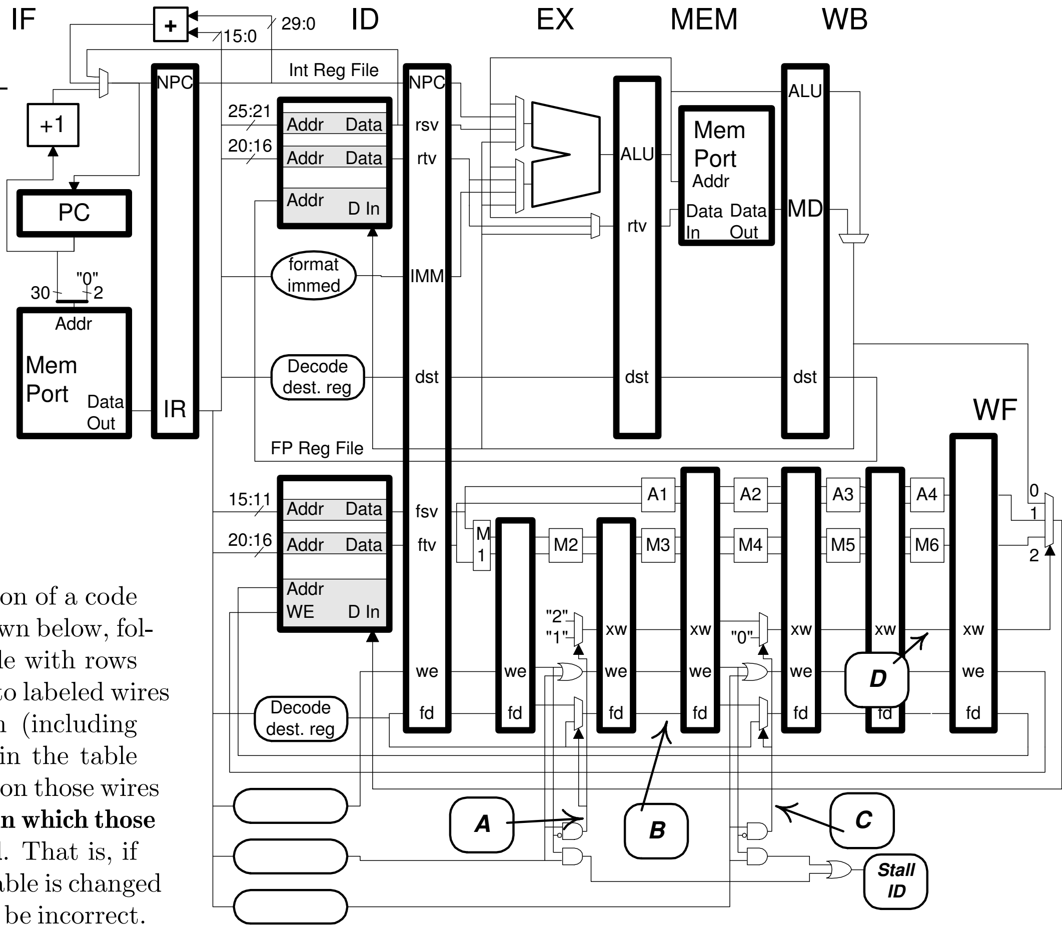
Problem 1		(20 pts)
Problem 2		(20 pts)
Problem 3		(20 pts)
Problem 4		(20 pts)
Problem 5		(20 pts)
Exam Total		(100 pts)

This is a genuine EE 4720 exam. This is a genuine EE 4720 exam. This is a genuine EE 4720 exam. This is a genuine EE 4720 exam. This is a genuine EE 4720 exam. This is a genuine EE 4720 exam. This is a genuine EE 4720 exam. This is a genuine EE 4720 exam.

Alias

Good Luck!

Problem 1: (20 pts) The statically scheduled MIPS implementation illustrated to the right is taken from the class notes. To avoid making things too easy some descriptions were removed from logic blocks in the lower-left corner.



(*a*) The execution of a code fragment is shown below, followed by a table with rows corresponding to labeled wires in the diagram (including Stall ID). Fill in the table showing values on those wires **only for cycles in which those values are used**. That is, if a value in the table is changed execution must be incorrect.

☑ Complete the table, omitting unused values.

```
# Cycle             0  1  2  3  4  5  6  7  8  9  10
 mul.d f2, f12, f18 IF ID M1 M2 M3 M4 M5 M6 WF
 add.d f8, f10, f16    IF ID A1 A2 A3 A4 WF
 sub.d f6, f20, f14       IF ID -> A1 A2 A3 A4 WF
 lwc1 f4, 0(r1)              IF -> ID ----> EX ME WF
# Cycle             0  1  2  3  4  5  6  7  8  9  10
 A                     1  0  1                       <- Solution
 B                        8  2  6                    <- Solution
 C                           0  0  0  1              <- Solution
 D                                 1  2  1  0        <- Solution
 Stall ID:          0  0  0  1  0  1  1  0  0  0  0  <- Solution
# Cycle             0  1  2  3  4  5  6  7  8  9  10
```

Solution Discussion

A: This control signal is used to insert `add` and other instructions that use the floating-point adder. When the signal is `0` `fd` passes through unchanged and `xw` is set to take the multiplier output. An instruction is inserted when the control signal is `1`: the `fd` value from `ID` stage will enter and the `xw` control signal will be set to take the adder output.

In the code above the `add.d` and `sub.d` need the FP adder. The `1` in cycle 2 is generated for the `add.d`, it arrives at `A1` in cycle 3. Also in cycle 3 a `0` is generated for the `sub.d` instruction, it is refused entry because it would have arrived at `WF` in the same cycle as `mul.d`. In cycle 4 the signal is `1` and so the `sub.d` enters `A1` in cycle 5. If there is no instruction being inserted and there is no multiply passing through it does not matter what the value is and so those entries are blank (cycles 0, 1, and $5 \cdots 10$).

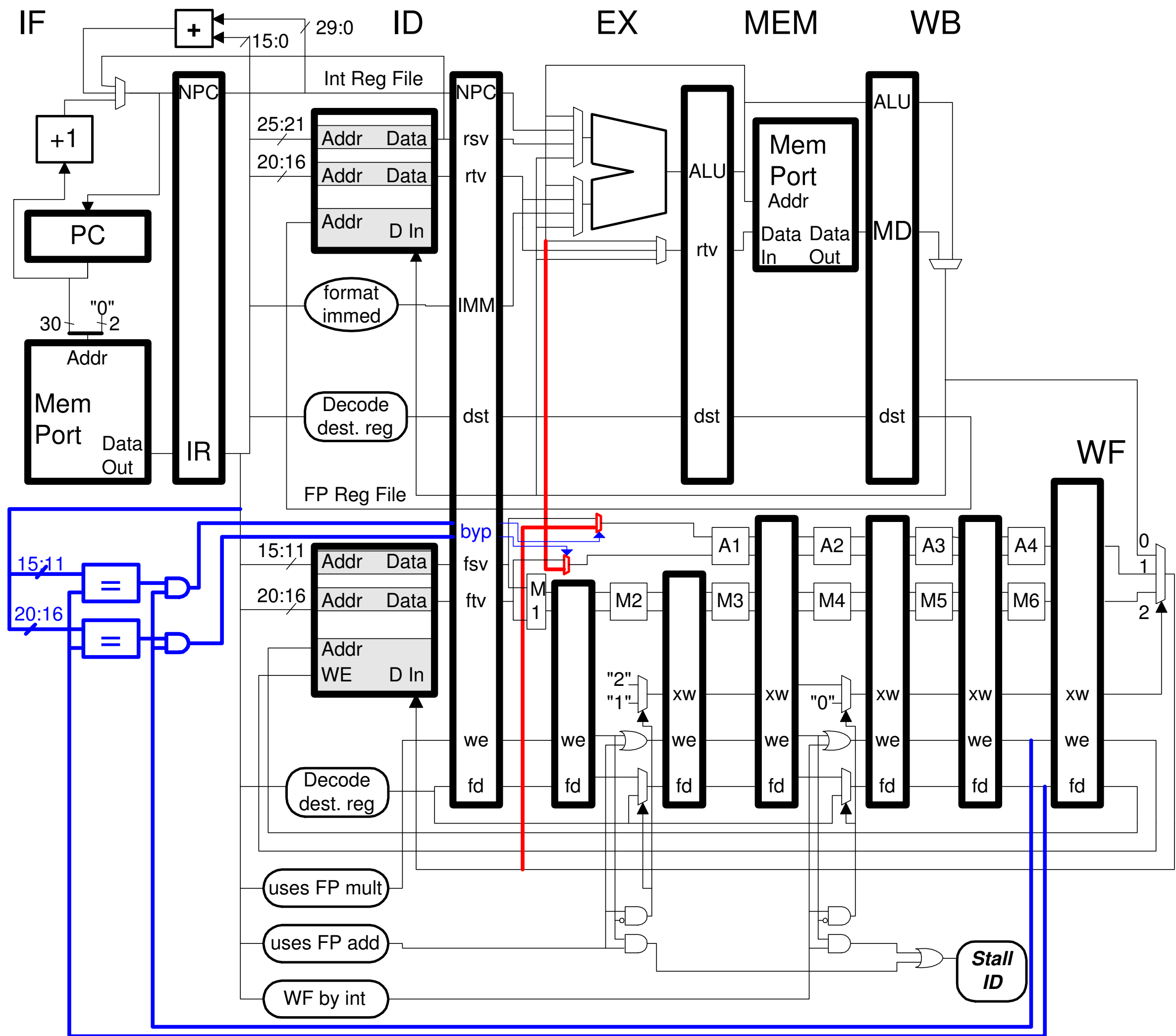
B: This signal provides the floating-point register to write for the instruction in stage `A1/M3`. Note that add-unit and multiply-unit instructions pass through this stage but instructions that use the integer pipeline to get a floating-point register value do not pass through this stage. That's why `f4` is not present here (it is inserted in the next stage).

C: This control signal is used to insert instructions that get their value from the integer pipeline. In the example, that's `lwc1`, another such instruction is `mtc1`, seen in part b. The `lwc1` is inserted in cycle 7, which is its last cycle in ID. The signal must be `0` when other instructions pass through, that happens in cycles 4, 5, and 6. At other cycles it doesn't matter what the value is.

D: This is the control signal for the `WF`-stage multiplexor. A value of `0` selects the value from the integer pipeline, `1` selects the floating-point add unit, and a `2` selects the multiply unit. The value has no effect if there isn't an instruction in WF in the next cycle, for that reason values are blank in cycles 0 to 5 and cycle 10.

Stall ID: This value is 1 if there is a stall. Note that a stall starts in the cycle before the beginning of an arrow (cycles 3 and 6) and the stall ends in the cycle before the arrowhead. Unlike the other signals this one must be shown every cycle to achieve the given execution.

Grading Note: For Stall ID very few showed 0's for every cycle without a stall.



(*b*) Show the execution of the code below on the implementation assuming that all needed bypass are present. Don't forget to check for dependencies. (Instruction `mtc1` moves a value from an integer register to a floating-point register.)

☑ Pipeline diagram.

```
# Cycle                 0  1  2  3  4  5  6  7  8  9  10 11
 mtc1 f2, r7            IF ID EX ME WF                        <- Solution
 add.s f3, f4, f2          IF ID A1 A2 A3 A4 WF               <- Solution
 add.s f6, f3, f8             IF ID -------> A1 A2 A3 A4 WF   <- Solution
# Cycle                 0  1  2  3  4  5  6  7  8  9  10 11
```

Diagram shown above. Note that it's possible to bypass `f2` from `mtc1` to the `add.s` without a stall.

Grading Note: Most did not realize a `mtc1`-to-`add.s` bypass was possible and so showed a stall in cycle 3.

(*c*) Add the bypass paths needed by the code above and show the control logic for the added paths. **Do not add unneeded bypass paths.** *Hint: Control logic should consist of two one-bit signals.*

☑ Bypass paths for code above.

Bypass paths shown in red. Note that the bypass from `mtc1` to `add.s` is taken from the `ME` stage by extending the `ME`-to-`EX` bypass connection.

Also, it would probably make more sense to connect the outputs of added bypass multiplexors to both the add and multiply units. The reason for not doing it in the diagram was just space. Those solving the problem might have avoided doing so since unneeded bypass paths were not to be added. Points would not have been deducted for that since those `ME`-to-`M1` paths are free.

☑ Control logic for added bypass paths.

Control logic shown in blue. The control signals are computed in `ID`, pass through the `ID/A1` latch and are used in `A1`. Computing the control signals in `A1` is not a good idea because extra pipeline latch space would be needed for `fs` and `ft` (ten bits versus two) and worse the floating point add would have to wait for the comparison units.

Problem 2: (20 pts) Illustrated is the execution of some code on our dynamically scheduled MIPS implementation along with the contents of the ID register map, the commit register map, and the physical register file. The implementation itself is shown on the next page.

```
 # 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
 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
 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
  9                      [                                  2.1                      ]
 12          2.0                                                 ]
 18          4.0                                                                ]
 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
```

(*a*) Answer the following

☑ Which physical register was allocated for `f4` in `add.d`?

Physical register 51.

☑ If one used the ID map to determine the value of `f2` in cycle 11, what value would one obtain? *Hint: It's a two-step process.*

Value of 2.3 (from physical register 99).

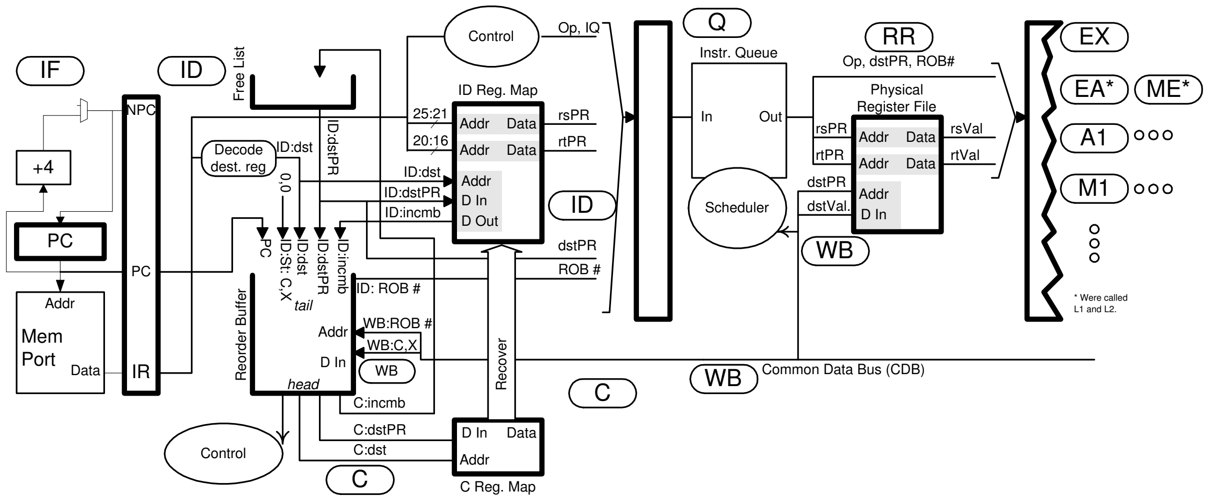
☑ If one used the commit map to determine the value of `f2` in cycle 11, what value would one obtain? *Hint: It's a two-step process.*

Value of 2.1 (from physical register 9).

☑ In cycle 11 where is the value of `f2` from `ldc1` located?

In the physical register file, in physical register 71.

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



Problem 2, continued:

```
 # 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
 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
 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
  9                      [                                  2.1                      ]
 12          2.0                                                 ]
 18          4.0                                                                ]
 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
```

(*b*) Suppose that in cycle 11 the contents of physical register number 71 was somehow changed, perhaps due to a one-time problem. If the code executes as shown then the program runs correctly. But if a hardware interrupt happens (is taken) at the wrong time execution would be incorrect because of this change. Explain why, illustrate timing details on the diagram.

☑ Reason for incorrect execution.

Physical register 71 holds the value of **f2** used as a source by **sub.d**. If the value were changed in cycle 11 that would not affect **sub.d** since it already read (actually bypassed) the value in cycle 7 (if it got it from the register file it would be in cycle 6). If by chance the handler for the hardware interrupt starts right after **ldc1** then execution would have to resume at **sub.d** (and since it's a hardware interrupt there's no reason to think any of the instructions above caused the problem). When the handler returns **sub.d** will again execute (it didn't commit the first time) but this time it will read the changed 71 from the physical register file.

The key feature is the handler starting between **ldc1** and **sub.d**.

Grading Notes: Many solutions described the handler as starting, say, in cycle 12. Instead, the solution should have indicated the last instruction to commit before the handler starts.

☑ Timing details on diagram.

See the answer above.

Problem 3: (20 pts) The MIPS code below runs on a system using a bimodal branch predictor of the indicated sizes. Branch outcomes are shown for each branch, the outcome patterns will continue to repeat.

```
BIGLOOP:
B1: 0x1000 beq r1, r2    T  T  T  T  T  N  T  N  N  T  T  T  T  T  T  N  T  N  N  T ...
... nonbranch insn.
...
B2: 0x1100 bne r3, r4     N  N  N  N  N  N  N  N  N  N  N  N  N  N  N  N  N  N  N  N  ...
nop
j BIGLOOP
nop
```

(*a*) What is the accuracy after warmup of a bimodal branch predictor with a 2^{14}-entry BHT on branch B1?

☑ 2^{14}-entry BHT bimodal accuracy on B1.

Accuracy is 60%.

(*b*) What is the accuracy after warmup of a bimodal branch predictor with a 2^4-entry BHT on branch B1?

☑ 2^4-entry BHT bimodal accuracy on B1.

With a 2^4-entry BHT branches B1 and B2 will share the same entry. Since B2 always executes after an execution of B1 it will be as though the counter were unchanged when B1 is taken and decremented by 2 when B1 is not taken. The counter will soon reach zero and only the not-taken outcomes will be correctly predicted and so the accuracy is 30%.

(*c*) What is the smallest BHT size for which one can obtain the same accuracy on branch B1 as a 2^{14} entry table? Explain.

☑ Smallest size for 2^{14} entry accuracy.

Smallest size: 2^7 entries.

☑ Reason.

That is the smallest size at which B1 and B2 use separate entries. It's the sharing of entries that results in the lower accuracy in part b.

(*d*) Normally the BHT in a MIPS implementation is indexed starting at bit position 2 (omitting the 2 least-significant bits) of the branch PC (address). For the following questions think about answers to the preceding parts but answer the question for ordinary programs, not the code sample above.

☑ Why might starting at position 3 or 4 be better?

Better means reducing the number of collisions (sharing of entries, as in part b). Taking, say, the 14 PC bits from 16:3 instead of 15:2 would separate branches that differed at bit 16 but whose lower bits are identical. That's good. That might be offset by collisions from branches which differ only in bit position 2; those branches would be adjacent and since that's unlikely there would probably be a benefit by starting at 3.

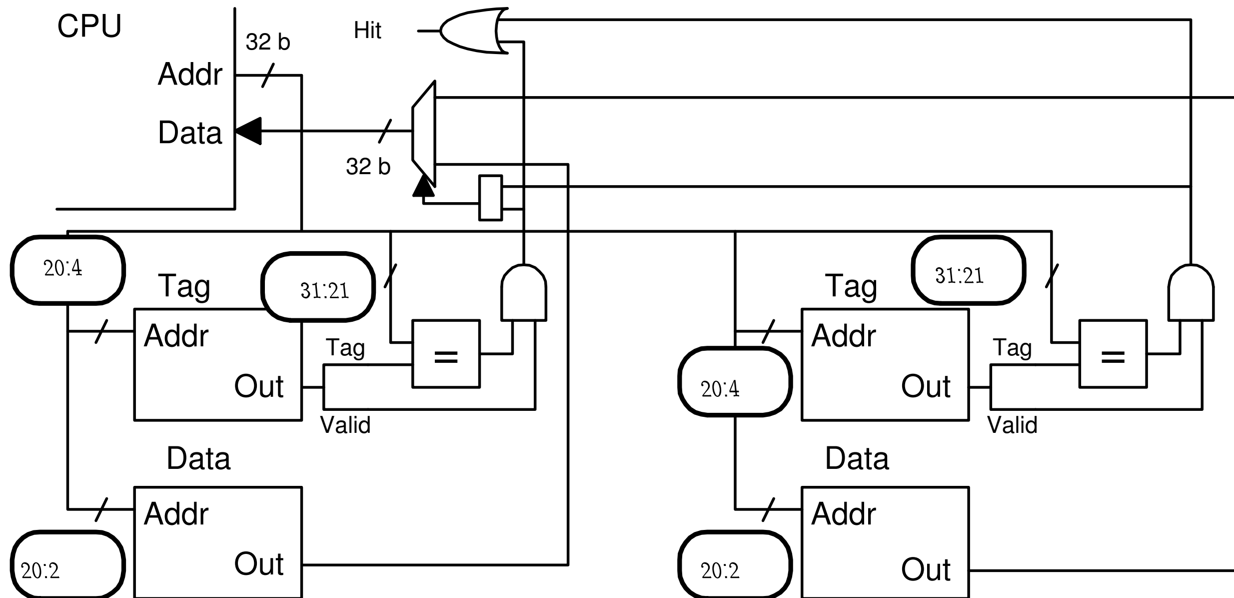
☑ Why might starting at position 10 or 11 be worse?

About one out of six instructions is a branch, so starting at 10 would certainly result in collisions by nearby branches.

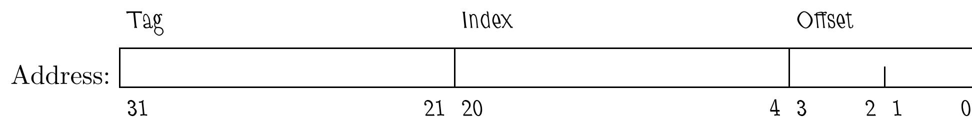
Problem 4: (20 pts) The diagram below is for a 4-MiB (2^{22}-character) set-associative cache with a line size of 16 characters on a system with the usual 8-bit characters.

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

☑ Fill in the blanks in the diagram.



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



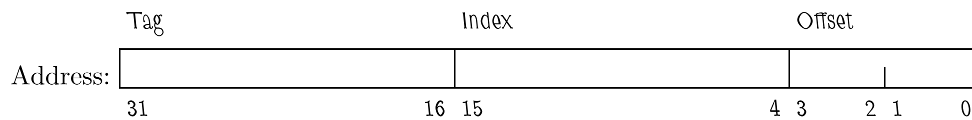
☑ Associativity:

The cache is 2-way set associative.

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

It's the cache capacity plus $2 \times 2^{21-4}\,(32 - 21 + 1)$ bits.

☑ Show the bit categorization for a 64-way set-associative cache with the same capacity and line size.



Problem 4, continued:

(*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? Explain

```
double sum = 0.0;
short *a = 0x2000000;     // sizeof(short) = 2 characters.
int i;
int ILIMIT = 1 << 10;     // = 2^10

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

The line size of 16 characters is given, the size of an array element is two characters. The miss on the first iteration will bring in 16 characters, a line. The second iteration will access data on this line. The line will be "used up" at $i = \frac{16}{2} = 8$, and so for each miss there are 7 hits. The hit ratio is $\frac{7}{8}$..

(*c*) The code below runs on a **direct mapped cache** with the same line size and capacity as the cache from the first part. Initially the cache is empty; consider only accesses to the arrays. Choose `b`, `ILIMIT`, and `ISTRIDE` so that the cache is completely filled in the minimum number of iterations (minimum `ILIMIT`). (Every access should be a miss.)

☑ `b`, `ILIMIT`, and `ISTRIDE`

☑ Briefly explain each choice.

```
double sum = 0.0;
char *a = 0x2000000;  // sizeof(char) = 1 character.

char *b = 0x3000010;                                  // SOLUTION

int i;
int ILIMIT = 1 << 17;                                 // SOLUTION

int ISTRIDE = 1 << 5;                                 // SOLUTION


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

Since the cache is direct mapped generate each index exactly once. Accesses to `a` generate even indices and `b` odd indices.

Problem 5: Answer each question below.

(*a*) (5 pts) Consider trap instructions and instructions that raise exceptions.

☑ What are trap instructions typically used for?

Trap instructions allow code running in user mode to call operating system routines. Those routines can do things that user code is not allowed to do, such as accessing hardware. A trap instruction might be used for system calls, for example, to open or read from a file.

☑ Sometimes when an instruction in a program raises an exception the program ultimately is allowed to continue. Give an example of such an exception, and what the handler might do.

Load and store instructions frequently encounter routine problems that the operating system can fix. This might include moving a page of memory from the disk into memory. After the handler fixes such a problem it will restart the program at the load or store that raised the exception, with the expectation that the load or store would now complete normally.

(*b*) (5 pts) When a MIPS instruction raises an exception the type of exception is written to the cause register. SPARC V8 lacks an equivalent of a cause register, so what does it use as a substitute? Explain.

☑ SPARCs alternative to MIPS' cause register.

SPARC calls different exception handlers for each type of exception so there is no need to check a cause register. For example, suppose a FP instruction raises an exception due to an arithmetic error. On SPARC a handler just for that exception would be called (located at entry 8 in the trap table), in MIPS a generic handler would be called which would have to examine the cause register to discover, for example, that a FP exception was raised. It would then jump to the appropriate handler.

(*c*) (5 pts) An early critic might have said that the improvements realized by dynamically scheduled systems could be achieved on much less expensive statically scheduled systems by using better compilers. The particular compiler improvements would help statically scheduled systems but have no impact on dynamically scheduled ones. Consider two-way superscalar statically and dynamically scheduled systems for the examples needed below.

☑ Explain what the compilers would have to do and why.

A two-way statically scheduled superscalar system would likely suffer from stalls due to true dependencies, see the first code fragment below. A compiler could avoid those stalls by scheduling (re-arranging) instructions, see the second fragment below.

☑ Provide an example, showing code before and after optimization.

```
# Solution

# Before optimization, execution on statically scheduled 2-way superscalar.
#    Execution is same as a less-expensive scalar system.

 add r1, r2, r3   IF ID EX ME WB
 sub r4, r1, r5   IF ID -> EX ME WB
 or  r6, r7, r8      IF -> ID EX ME WB
 and r9, r6, r10     IF -> ID -> EX ME WB


# Before optimization, execution on dynamically scheduled 2-way superscalar.
#    Though SUB and AND instructions wait other instructions aren't delayed.

 add r1, r2, r3   IF ID Q  RR EX WB C
 sub r4, r1, r5   IF ID Q     RR EX WB C
 or  r6, r7, r8      IF ID Q  RR EX WB C
 and r9, r6, r10     IF ID Q     RR EX WB C


# After optimization: Execution is ideal, same as a more-expensive DS system.

 add r1, r2, r3   IF ID EX ME WB
 or  r6, r7, r8   IF ID EX ME WB
 sub r4, r1, r5      IF ID EX ME WB
 and r9, r6, r10     IF ID EX ME WB
```

(*d*) (5 pts) What is it about loads that allow dynamically scheduled systems to outperform statically scheduled systems even with good compilers?

☑ Explain.

Unlike most other integer instructions, loads take two cycles to produce a result and so a compiler will need to find at least n instructions to place between a load and an instruction sourcing the loaded value. What's worse than that is that sometimes loads miss the cache. If the compiler misses the L1 cache but hits the L2 cache it might be ten cycles before a result is ready, so the compiler might be able to at least schedule away some stalls for a particular load. However, it can't do that for all loads.

In contrast a dynamically scheduled system will allow non-dependent instructions following a missing load to execute.