Name Solution

Computer Architecture
EE 4720
Final Examination

8 May 2006, 10:00–12:00 CDT

Problem 1		(25 pts)
Problem 2		(25 pts)
Problem 3		(20 pts)
Problem 4		(30 pts)
Exam Total		(100 pts)

Alias DVD-HD or Blu-ray?

Good Luck!

Problem 1: The execution of code on a dynamically scheduled scalar MIPS implementation using Method 3 (the only one covered in class) is shown below. Beneath the diagram are signal labels, for example, `ID:dst`. (25 pts)

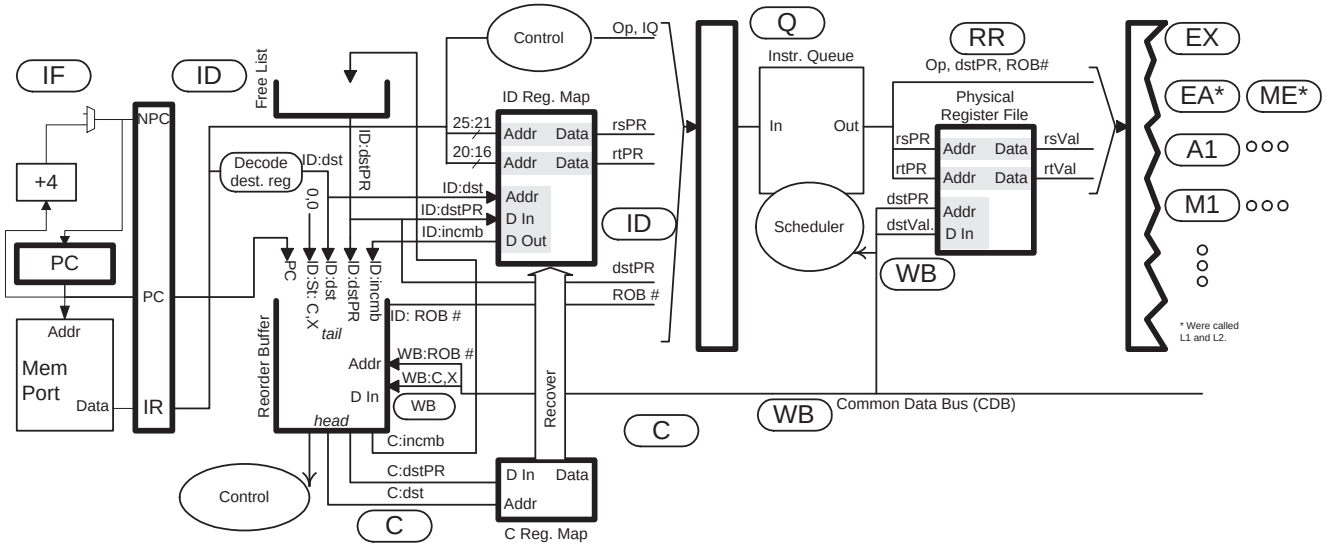
(*a*) On the next page show the values of the signals at the cycles they are used.

- Some values are shown, they are needed to determine other values.
- All values can be determined.
- Ignore the distinction between integer and FP registers.
- The free list contents at cycle zero is shown on the lower right-hand side of the figure on the next page.

Solution appears on next page in blue. The values of `ID:rsPR` in cycles 1 and 3 (appearing in the problem, not just the solution) tell us that register `f2` is initially mapped to physical register 56, `f4` is initially mapped to physical register 63, `r5` is initially mapped to physical register 30, and `r6` is initially mapped to physical register 54.

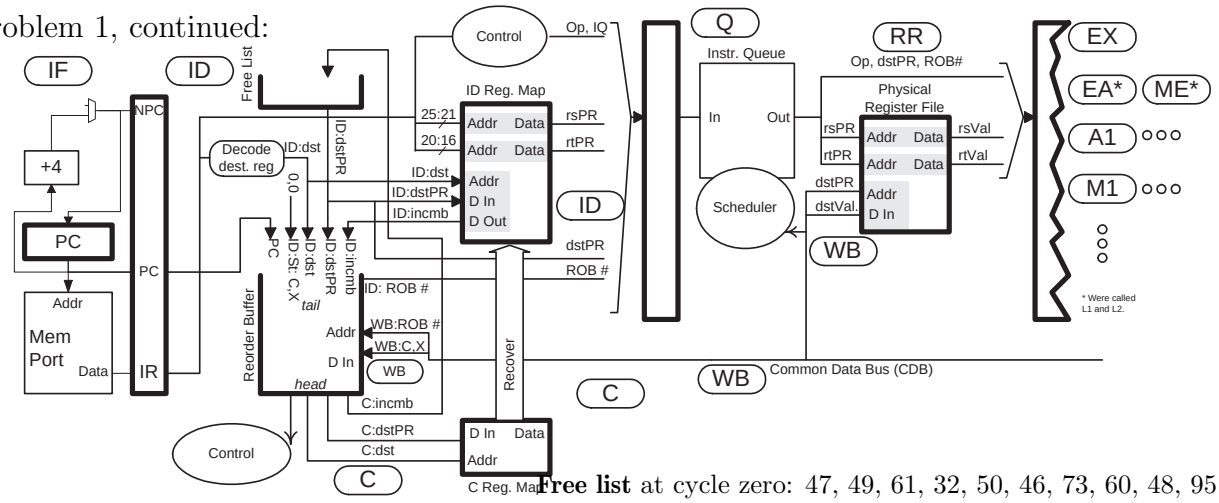
Because they are not written in the code fragment the mappings for `f4` and `r6` don't change. Registers `f2` and `r5` get new mappings, the physical registers are taken from the free list.

The `ID:dst` row shows an "r" or "f" in addition to the register number. In a real implementation there might be separate maps for integer and floating point registers. If not, a bit might be used to indicate whether the register is floating point.



Tables for answer on next page.

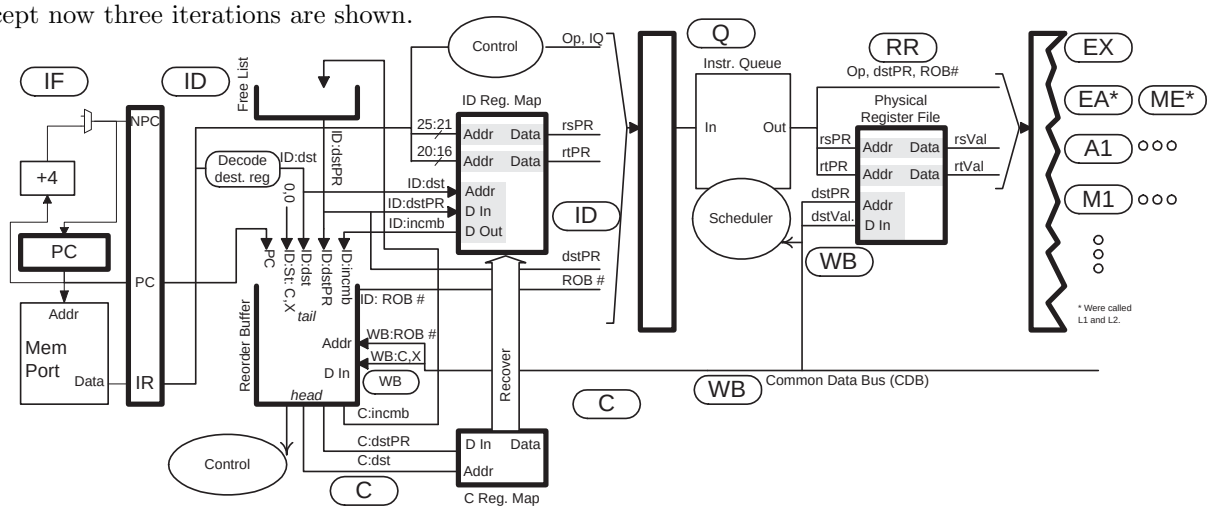
Problem 1, continued:



Free list at cycle zero: 47, 49, 61, 32, 50, 46, 73, 60, 48, 95

```
LOOP:              # First Iteration
 add.s  f2, f2, f4  IF ID Q  RR A1 A2 A3 A4 WB C
 bgtz r5  LOOP         IF ID Q  RR B  WB          C
 sub r5, r5, r6           IF ID Q  RR EX WB          C
# Cycle:            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
LOOP:              # Second Iteration
 add.s  f2, f2, f4           IF ID Q     RR A1 A2 A3 A4 WB C
 bgtz r5  LOOP                  IF ID Q     RR B  WB          C
 sub r5, r5, r6                    IF ID Q     RR EX WB          C
# Cycle:            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15

ID:rsPR               56 30 30 47 49 49

ID:rtPR               63    54 63    54

ID:dst                f2    r5 f2    r5

# Cycle:            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15

ID:dstPR              47    49 61    32

ID:incmb              56    30 47    49

RR:rsPR                     56 30 30    47 49 49

# Cycle:            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15

RR:rtPR                     63    54    63    54

WB:dstPR                                  49 47       32 61

# Cycle:            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15

C:incmb                                      56    30    47    49

C:dstPR                                      47    49    61    32

C:dst                                        f2    r5    f2    r5

# Cycle:            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
```

Problem 1, continued: The diagram below is for the same code on the same system as the previous part, except now three iterations are shown.




```
LOOP:            #  First Iteration
 add.s  f2, f2, f4  IF ID Q  RR A1 A2 A3 A4 WB C
 bgtz r5  LOOP         IF ID Q  RR B  WB          C
 sub r5, r5, r6           IF ID Q  RR EX WB          C
# Cycle:            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
LOOP:            #  Second Iteration
 add.s  f2, f2, f4           IF ID Q     RR A1 A2 A3 A4 WB C
 bgtz r5  LOOP                  IF ID Q     RR B  WB          C
 sub r5, r5, r6                    IF ID Q     RR EX WB          C
# Cycle:            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
LOOP:            #  Third Iteration
 add.s  f2, f2, f4                    IF ID Q        RR A1 A2 A3 A4 WB C
 bgtz r5  LOOP                           IF ID Q  RR B  WB                C
 sub r5, r5, r6                             IF ID Q     RR EX WB             C
```

(*b*) What is the CPI for a large number of iterations? *Hint: It's not 1.*

☑ CPI?

The quick, common sense way to solve this problem is to note that each `add.s` will start right after the completion of the `add.s` in the previous iteration and the `add.s` will commit in the cycle after `WB` (since the `sub` and `bgtz` finish much faster), and so execution will proceed at a rate of four cycles per iteration or a CPI of $\frac{4}{3}$.

The regular way to determine CPI is to find a repeating pattern. Normally, we look for two cycles in which the first instruction of the loop is in `IF` and in which the state of the system is identical. That doesn't happen in the three iterations appearing above, and it won't happen for a while because at each iteration there are more instructions in flight.

An alternative way to find CPI is to look at the state of the system when the first instruction is in commit. In each case `add.s` in the following iteration is in `A2`. Assuming nothing interferes with the `add.s` in the next iteration starting execution on time, the processor will sustain a commit rate of four cycles per iteration. (Note that with the regular technique there is no need to make such assumptions.)

Therefore $\boxed{\text{CPI} = \frac{17-13}{3} = \frac{4}{3}}$.

Problem 1, continued:

(*c*) Suppose the ROB can hold 256 entries and there are an unlimited number of physical registers. Assuming a very large number of iterations, at about what cycle will fetch stall?

☑ Fetch will stall at cycle ≈

Instructions are being fetched at 1 instruction per cycle and committed at a rate of $\frac{3}{4}$ IPC. Let $n(c)$ denote the number of instructions in the reorder buffer at cycle c, then $n(c) = c - \frac{3}{4}(c-9)$ for $c \geq 9$. (Note that no instructions are committed until cycle 9.) Solving for c yields $c = 4n(c) - 27$, substituting $n(c) \rightarrow 256$ gives $\boxed{c = 997 \text{ cycles}}$.

(*d*) Would the code above run faster on a four-way superscalar dynamically scheduled system with the same clock frequency and pipeline depth? Explain.

☑ Circle One: Faster $\boxed{\text{Not Faster}}$.

☑ Because

Execution above is limited by the execution of the `add.s`. A superscalar system would fetch instructions faster but since the next `add.s` starts as soon as the previous one finishes, that won't help.

Problem 2: The code fragments below run on three systems which are identical except for the branch predictor: One uses a **bimodal** predictor with a 2^{14}-entry BHT, one uses a **local predictor** with an 8-outcome local history and a 2^{14}-entry BHT, and one uses a **global predictor** with an 8-outcome global history. (25 pts)

(*a*) Provide the information requested below.

- All accuracies are after warmup.
- For the warmup time show the approximate number of times the predicted branch needs to be executed before the predictor reaches its warmup accuracy. Assume the 2-bit counters start out at 0 or 3, whichever is worse.

```
BIG:addi r3, r0, 3
LP: bne r3, r0 LP  # Iterates four times.
    addi r3, r3, -1

    lw r1,0(r2)
C2: beq r1, 0 SK   # T N T T N  T N T T N  T N T T N  T N T T N  ...
    nop
    nop
SK: j BIG
    addi r1, r1, 4
```

☑ Bimodal: accuracy on C2: 60%

☑ Bimodal: warmup time on C2: 8 executions

☑ Local: accuracy of C2: 100%

☑ Local: warmup time on C2: $\approx 8 + 2 \times 5 = 18$ executions

☑ Local: smallest history size needed for 100% accuracy on C2: 5 (If LP ignored would be 4 outcomes.)

☑ Global: accuracy on C2: 80%

☑ Global: warmup time on C2: $\approx 1 + 6 + 2 = 9$ executions

☑ Global: smallest history size needed for 100% accuracy on C2: $4 \times 5 = 20$

See next page for discussion of solution.

Problem 2, continued:

The code below is repeated from the previous page with stuff added to show how to compute bimodal prediction accuracy.

```
BIG:addi r3, r0, 3
LP: bne r3, r0 LP  # Iterates four times.
    addi r3, r3, -1

    lw r1,0(r2)
# Comments show solution work for bimodal accuracy.
# Execution Num    # 0  1  2  3  4   5  6  7  8  9   10 11 12 13 14
C2: beq r1, 0 SK   # T  N  T  T  N   T  N  T  T  N   T  N  T  T  N ...
# Counter:          0  1  0  1  2  1   2  1  2  3  2   3  2  3  3  2  ...
# Prediction:       N  N  N  N  T  N   T  N  T  T  T   T  T  T  T  T
# Correct            x     x  x  x   x  x  x     x      x        x
    nop
    nop
SK: j BIG
    addi r1, r1, 4

# Global history, C2 outcomes in upper case, LP outcomes in lowercase.
# T tttn N tttn T tttn T tttn N tttn  T tttn N tttn T tttn T tttn N tttn  ...
```

General: Branch C2 has an outcome pattern of length 5: TNTTN.

A branch is warmed up when the 2-bit counters it uses reach the correct value. Bimodal is easiest since it uses just one counter. If the branch is highly biased it will take two executions to warm up a counter. Branch C2 is biased, but not highly so it will take more. More on bimodal further below. For the other predictors you need to compute how many counters they use (by finding the number of local or global history patterns used to predict them). See the discussion of those predictors, below.

Bimodal: The code above shows the 2-bit counter used by the bimodal predictor to predict branch C2. A repeating pattern is established after execution 9 (see the numbers in the diagram), in which the counter value is 2, it is also 2 after 14, and so the pattern will repeat. With 3 out of 5 correct the accuracy is 60%.

Bimodal just uses one counter. Since C2 is not highly biased it will take more than 2 executions to warm it up, from the diagram above it can be seen that it takes about 8.

Local: The 5-outcome pattern easily fits in the 8-outcome history so prediction accuracy is 100%. After 8 executions (the length of the local history) there will be 5 different local histories for C2: `TNTTNTNT`, `NTTNTNTT`, `TTNTNTTN`, `TNTNTTNT`, `NTNTTNTN`. At worst, each will take two executions to warm up, so the warmup time is $8 + 5 \times 2 = 18$ executions.

If ignoring the LP branch the smallest history size would be 4. Two would not be enough because `NT` can be followed by an `N` or `T`; three is not enough because `TNT` can be followed by an `N` or `T`. If the LP branch is considered then 4 is not enough because patterns `TTNT` and `TNTT` occur with both branches, LP and C2, and the subsequent outcomes differ. Five outcomes are sufficient.

Global: The first step in solving this is determining the global histories seen when making the prediction. Between each outcome of C2 there are four executions of LP (SK doesn't count because it's a jump, not a branch). The global history is shown in the bottom of the code above, the history used when predicting C2 is the eight outcomes just before an upper-case letter, for example, **`ttn N tttn`** and **`ttn T tttn`**. In fact, these are the only two distinct global histories see when predicting C2. Pattern **`ttn N tttn`** is seen twice, in both cases before a taken branch, so its predictions will always be correct. Pattern **`ttn T tttn`** is seen three times, once before a `T` outcome and twice before an `N` outcome, and so after warmup it will predict `N` and be correct $\frac{2}{3}$ of the time. The overall prediction accuracy is $\frac{1}{5}\left(2 \times 1 + 3 \times \frac{2}{3}\right) = \frac{4}{5}$.

A hand analysis shows that the counter for **`ttn T tttn`** warms up in 6 executions (assuming a worst-case start value of 3), being highly biased, the counter for **`ttn N tttn`** warms up in 2 executions. The patterns above will not be seen until after the first iteration, so the total warmup time is $1 + 6 + 2 = 9$ executions.

For 100% accuracy the global history must hold four outcomes of C2 (see the discussion for the local predictor). It takes 5 global outcomes to capture one C2 outcome, so for four C2 outcomes the global history length must be 20 outcomes.

Problem 2, continued:

(*b*) The code below is similar to the code on the previous page except the four-iteration loop has been replaced by two random branches. Each random branch will be taken with probability .5 and the outcome is independent of everything, including the other random branch.

```
BIG:lb r3, 0(r4)
    beq r3, r0  S2  # Random.
    lb r3, 1(r4)
S2: beq r3, r0  S3  # Random.
    addi r4, r4, 2
S3: nop

    lw r1,0(r2)
C2: beq r1, 0  SK   # T N T T N  T N T T N  T N T T N  T N T T N  ...
    nop
    nop
SK: j BIG
    addi r1, r1, 4
```

```
 # Solution - Global History.
 # Upper case is C2, r is first random branch, s is second random branch;
 # r and s can be either taken or not-taken.
 # T r s N r s T r s T r s N r s  r s T r s N r s T r s T r s N r s ..
```

☑ Global: accuracy on C2: 80%

☑ Explain using GHR values.

☑ Global: warmup time on C2: $\approx 3 + 2 \times 3 \times 2^6$ executions

☑ Explain using GHR values.

☑ Global: smallest history size needed for 100% accuracy on C2: $4 \times 3 = 12$

☑ Explain using GHR values.

This loop brings good news and bad news. The good news is that an eight-outcome global history will hold two C2 outcomes. The bad news is that it doesn't help, and warmup time is worse.

Accuracy: The global history patterns when predicting C2 are: **rsTrsTrs**, **rsNrsTrs**, and **rsTrsNrs**, where each **r** and **s** can be either taken or not taken. A pattern of the form **rsTrsTrs** occurs once and is always followed by an **N** so it predicts at 100%. Form **rsNrsTrs** occurs twice and is followed by an **N** or **T**, its accuracy will be discussed further below. Form **rsTrsNrs** occurs twice and is always followed by a **T**, so it predicts at 100%.

As the alert reader guessed, **rsTrsNrs** is being called a "form" because it represents $2^6 = 64$ patterns, one for each distinct outcome of the random branches. For forms **rsTrsNrs** and **rsTrsTrs** the only impact of this diversity is increased warmup time. Each pattern still takes at worst 2 outcomes to warm up, but each form represents 64 patterns.

For pattern **rsNrsTrs** there is also an impact on accuracy. If **r** and **s** were always taken, then the counter for **rsNrsTrs** would see outcomes **T N T N T ...**, and so the prediction accuracy would be 50% if the initial value of the counter were 0, 2, or 3; if the counter were 1 the accuracy would be 0%. Since **r** and **s** are random each counter sees a random subset of **T N T N T N**, since that subset is balanced between **T**'s and **N**'s, it is equally likely that the counters predict taken or not taken. (Actually, when

the next C2 outcome is `T` there is a tiny bias towards predicting `N`, and vise versa.) So the prediction is random, uniform (between `T` and `N`), so the prediction accuracy for form `rsNrsTrs` will be 50%. The overall prediction accuracy is then still 80%.

As stated above, the worst-case warmup time for the highly biased patterns is 2 outcomes each. Assume the same is true for `rsNrsTrs`, though it will actually be larger. If the random branches uncharacteristically did not repeat a pattern, the total warmup time for all three forms would be $3 + 3 \times 2^6 \times 2$, but that's unlikely. Since branches are random there is no precise warmup time, instead the solution will be to find the number of executions at which the probability that the 2-bit counter used for a prediction is warmed up is .05. Let x denote the number of times form `rsNrsTrs` appears, and suppose without loss of generality that we are predicting `ttNtttTtt` (the $x + 1$ time the form appears). The probability that `ttNtttTtt` is seen a particular time is $\frac{1}{64}$, the probability it's not seen a particular time is $1 - \frac{1}{64}$ and the probability it's not been seen in the past x executions is $\left(1 - \frac{1}{64}\right)^x$. The probability that it's been seen exactly once is $\left(1 - \frac{1}{64}\right)^{x-1} \frac{1}{64}x$. The probability that it's been seen less than 2 times is the sum, $\left(1 - \frac{1}{64}\right)^x + \left(1 - \frac{1}{64}\right)^{x-1} \frac{1}{64}x$. So to find the warmup time for this pattern solve $\left(1 - \frac{1}{64}\right)^x + \left(1 - \frac{1}{64}\right)^{x-1} \frac{1}{64}x = 0.05$ for x. Using a symbolic math program (Mathematica) that yields $x = 301.7$. So the total warmup time is 3×301.7. *Grading note: no one got it quite that close.*

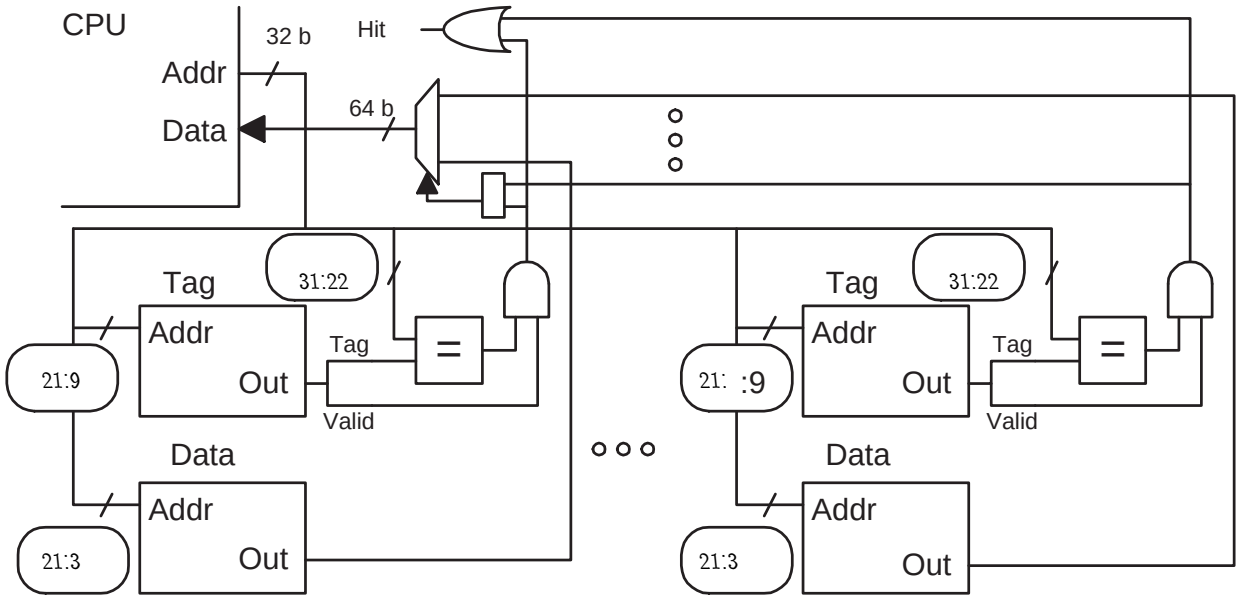
An acceptable answer to the problem would be a warmup time of $3 \times 2 \times 2^6$ executions, which is the boxed answer before the explanation.

The minimum history size has to be large enough to hold 4 C2 outcomes. It takes three global outcomes to hold one C2 outcome, so the minimum history size is 12 outcomes.

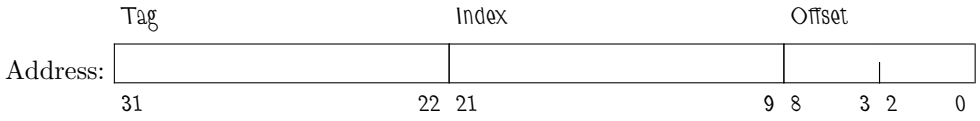
Problem 3: The diagram below is for a 64-MiB (2^{26}-character) 16-way set-associative cache on a system with the usual 8-bit characters. (20 pts)

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

☑ Fill in the blanks in the diagram.



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



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

It's the cache capacity plus $16 \times 2^{22-9}\,(32 - 22 + 1)$ bits.

☑ Line Size (Indicate Unit!!):

Line size is $2^9 = 512$ characters.

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

Problem 3, continued: For the problems on this page use the cache from the previous page.

(*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;
char *a = 0x2000000;    // sizeof(char) = 1 character.
int i;
int ILIMIT = 1 << 27;  // = 2^27

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

The line size is $2^9 = 512$ characters and the size of an array element is one character. Therefore each miss will be followed by 511 hits. The hit ratio for the first loop will be $\frac{511}{512}$. Will `a[0]` still be cached when the second loop starts? The cache capacity is given as 2^{26} characters and the first loop accesses 2^{27} characters, so they can't all fit. The cache will replace the least-recently used line within a set, and so `a[0]` won't be cached when the second loop starts. In fact, nothing brought in by the first loop is used in the second loop so the overall $\boxed{\text{hit ratio is } \frac{511}{512}}$.

(*c*) The slightly different 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.

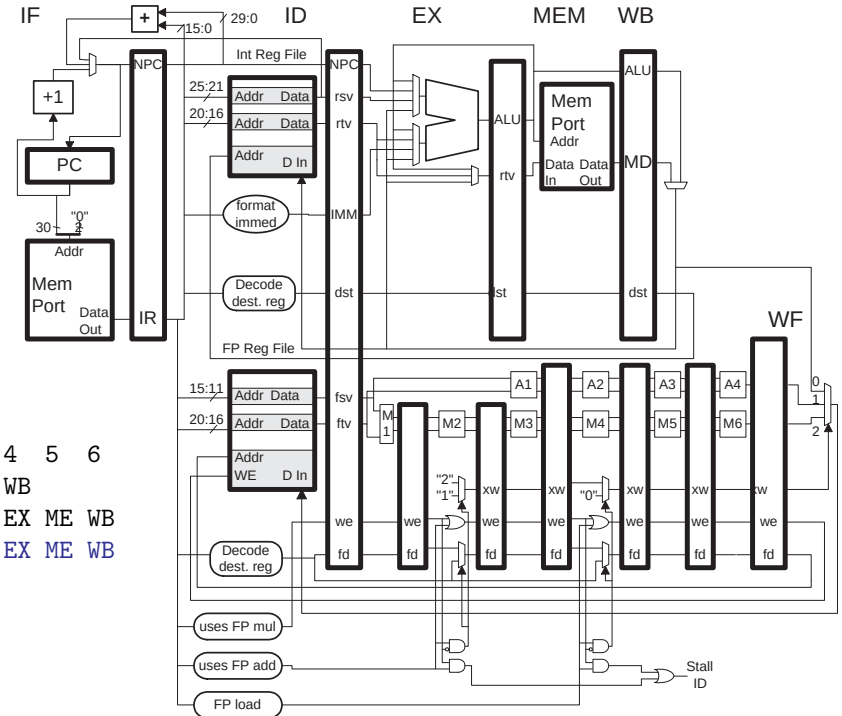
☑ Find the hit ratio. Very briefly explain.

```
double sum = 0.0;
char *a = 0x2000000;  // sizeof(char) = 1 character.
int i;
int ILIMIT = 1 << 27;

for(i=0;         i<ILIMIT; i++) sum += a[ i ];
for(i=ILIMIT-1; i>=0;     i--) sum += a[ i ];
```

The only difference here is that the second loop iterates backward. The last element accessed by the first loop is $\mathtt{a}[2^{27}-1]$, which is also the first element accessed by the second loop and so the second loop will start out with a hit (and 511 more while it consumes the line). Since the cache can hold half the array, half of the execution of the second loop will have a 100% hit ratio. The overall $\boxed{\text{hit ratio is } \frac{3}{4}\frac{511}{512} + \frac{1}{4}\frac{512}{512}}$.

Problem 4: Answer each question below.

(*a*) The execution of some code fragments on the illustrated implementation appear below. Explain why each execution is impossible and show a corrected pipeline execution diagram. (7 pts)




```
# Cycle          0  1  2  3  4  5  6
lw r2, 0(r4)     IF ID EX ME WB
add r1, r2, r3      IF -> ID EX ME WB
Fix:                IF ID -> EX ME WB
```

☑ Fix.

☑ Was impossible because:

The `add` did need to stall so the `r2` value could be bypassed from the `lw`. However there is no way the control logic could know this in cycle 2 because the `add` had not yet arrived.

```
# Cycle          0  1  2  3  4  5  6  7
lw r2, 0(r4)     IF ID EX ME WB
add r1, r2, r3      IF ID -> EX ME WB
xor r5, r6, r7         IF ID -> EX ME WB
Fix:                   IF -> ID EX ME WB
```

☑ Fix.

☑ Was impossible because:

In cycle 3 the `ID` stage holds two instructions, `add` and `xor`. It only has room for one.

```
add.d f0, f2, f4  IF ID A1 A2 A3 A4 ME WB
Fix:              IF ID A1 A2 A3 A4 WF
```

☑ Fix.

☑ Was impossible because:

Floating point instructions do not pass through the `ME` stage, and they use `WF` for writeback.

(*b*) For each feature below indicate whether it is usually a feature of the ISA or the implementation, and explain why it's not a feature of the other. (7 pts)

Feature: *The opcode can be found in bits 31:26.*

☑ [ISA] or Implementation?

☑ Why not the other?

The ISA should define everything you need to know to write a program (or to write a compiler). That would have to include instruction coding, including the position of the opcode field. If it were in the implementation then a compiler would have no idea where to place the opcode and different implementations could expect the opcode bits in different positions, so there would be no compatibility.

Feature: *The* `add` *in the code below will stall for one cycle.*

```
lw  r1, 0(r2)
add r3, r1, r4
```

☑ ISA or [Implementation]?

☑ Why not the other?

The ISA in principle could specify that the `add` should stall, but that would overly constrain implementations. If an implementation technique were developed that would allow the `add` execute without a stall it could not be used in the implementation of this ISA.

Feature: *Two consecutive delayed (as in MIPS) branches will yield unpredictable results.*

```
bneq r1, r2 DEST1
beq r3, r4 DEST2
addi r5, r6, r7
```

☑ [ISA] or Implementation?

☑ Why not the other?

The ISA specifies what programs do. Normally, an ISA would either say such pairs are illegal, in which case an implementation might raise an exception on such code, or it might say such pairs are fine (just execute one instruction at `DEST1` then continue at `DEST2`), in which case the implementation must handle them properly. Leaving it up to the implementation would break compatibility because each might handle such pairs differently.

Feature: *Integer addition overflow raises exception.*

☑ [ISA] or Implementation?

☑ Why not the other?

The ISA specifies the behavior of the program, including whether instructions raise exceptions. If it were up to the implementation then code would not be portable.

Note that the question is about whether overflow raises an exception, it is not about what the exception handler should do. The handler is part of the operating system, not the ISA or implementation.

(*c*) Consider the use of a packed-operand 8-bit add (of the type described in class) to speed up the code fragments below. For each fragment indicate whether it's certainly feasible, feasible with certain assumptions, or not feasible. (6 pts)

```
extern char *a, *b, *c;
for(i=0; i<1024; i++) a[i] = b[i] + c[i];
```

☑ Circle One: No. Yes! [Yes, assuming ...]

☑ Explain.

Yes assuming that saturating arithmetic is okay. That is, if the program expects $120 + 10 = -126$ (because of overflow), then saturating arithmetic would break the program. If, on the other hand the program works correctly with $120 + 10 = 127$ (because of saturation), then packed operand instructions should work fine.

```
extern int *a, *b, *c;
for(i=0; i<1024; i++) a[i] = b[i] + c[i];
```

☑ Circle One: No. Yes! [Yes, assuming ...]

☑ Explain.

Assuming that despite the use of integers, array elements are in the range $[-128, 127]$ and a sum is also in the range or that saturation is okay.

```
extern char *a, *b, *c;
for(i=1; i<1024; i++) a[i] = a[i] + a[i-1];
```

☑ Circle One: [No.] Yes! Yes, assuming ...

☑ Explain.

To compute `a[i]` element `a[i-1]` is needed so there is no easy way to do this in parallel with packed operand instructions.

(*d*) MIPS and other RISC ISAs typically have instructions using displacement addressing but lack register deferred addressing. (See the examples below.)

```
lw r1, 4(r2)   # Displacement addressing.
lw r1, (r2)    # Register deferred addressing.
```

Given that RISC and CISC ISAs have displacement addressing:

☑ (5 pts) Why is register deferred addressing helpful for CISC but not helpful for RISC?

Register deferred addressing lacks the displacement that is included in the displacement addressing already present in the two ISAs, they are otherwise identical. Because CISC instructions have variable length, an instruction with register deferred addressing can be shorter than one with displacement addressing, making programs smaller. In RISC all instructions are the same size so there is no advantage in omitting the displacement.

(*e*) Dead-code elimination (DCE) is a common compiler optimization.

☑ (5 pts) What is it? Illustrate using an example.

The removal of code that's either never executed or that produces values that are never used. See the example below.

```
a=1; // This line would be removed by DCE.
a=2;
```