Name Solution

Computer Architecture
EE 4720
Final Examination

10 May 2013, 12:30–14:30 CDT

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

Alias Click Here

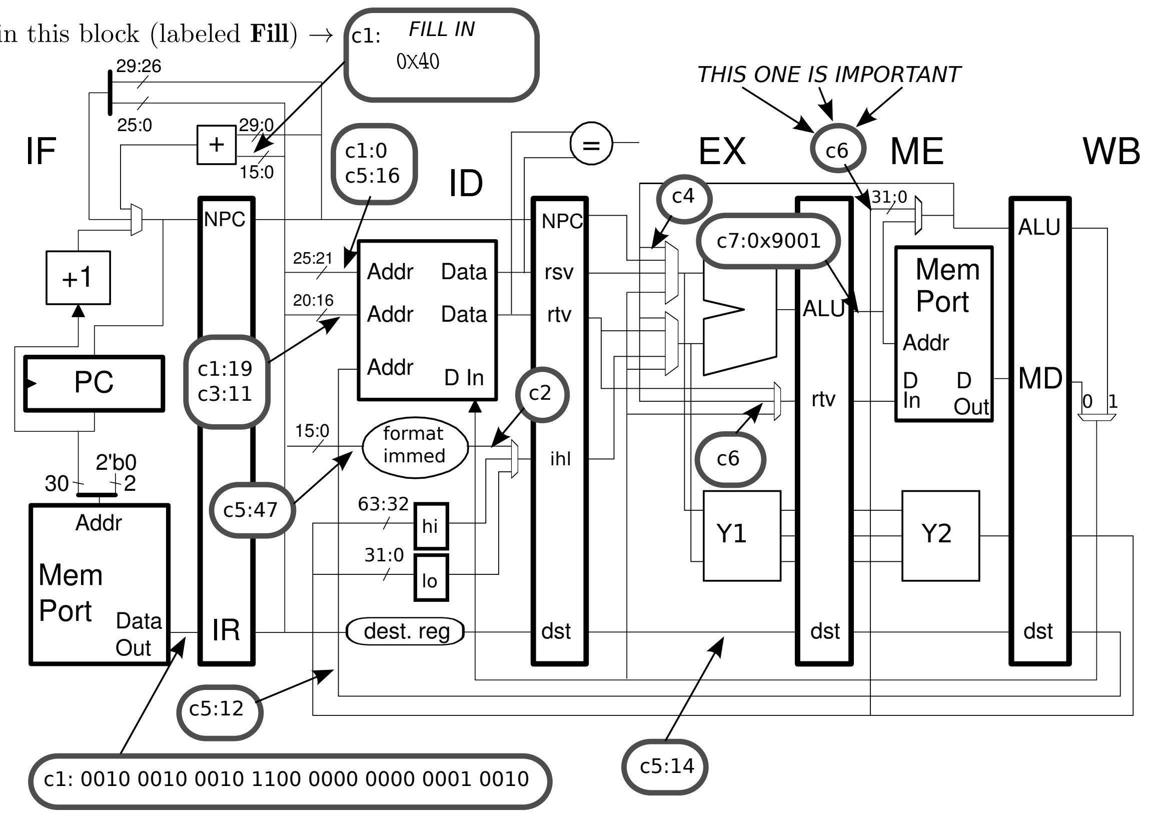
Good Luck!

Problem 1: (20 pts) The MIPS implementation below includes a two-stage integer multiply unit (`Y1` and `Y2`) for the `mult` instructions, similar to the MIPS implementation covered in Homework 4. Some wires are labeled with cycle numbers and values that will then be present. For example, $\boxed{\text{c5:14}}$ indicates that at cycle 5 the pointed-at wire will hold a 14. Other wires just have cycle numbers, indicating that they are used in that cycle. *Note that instruction addresses are provided.*

[✓] Write a program consistent with these labels.

[✓] All register numbers and immediate values can be determined.

[✓] Fill in this block (labeled **Fill**) →



```
# Cycle                     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
 0x1000: beq r0, r19 TARG   IF ID EX ME WB
 0x1004: addi r12, r17, 18     IF ID EX ME WB
 0x1104: mult r12, r11            IF ID EX ME WB
 0x1108: mflo r14                    IF ID EX ME WB
 0x110c: sb   r14, 47(r16)              IF ID EX ME WB
# Cycle                     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
```

Easy Source Registers: In ID the $\boxed{\text{c1:0}}$ tells us that in cycle 1 the instruction in ID (the one with address `0x1000`) uses `r0` as an `rs` source register. Similarly $\boxed{\text{c5:16}}$ tells us that the instruction at `0x110c` uses `r16` as a source. The $\boxed{\text{c1:19}}$ and $\boxed{\text{c3:11}}$ provide the `rt` registers for the instructions at `0x1000` and `0x1104`.

Easy Destination Registers: In EX the $\boxed{\text{c5:14}}$ tells us that the destination of the instruction at `0x1108` is `r14`. In WB (but appearing at the bottom of ID) the $\boxed{\text{c5:12}}$ tells us the instruction at `0x1004` has a destination of `r12`.

Easy Immediate: In ID the $\boxed{\text{c5:47}}$ tells at that the instruction at `0x110c` uses an immediate and that it is 47.

Easy Dependencies: The $\boxed{\text{c4}}$ in **EX** tells us that the first source of the instruction in **EX**, `0x1104`, uses the result of the instruction in **ME**, `0x1004`. Therefore, the first source of `0x1104` must be `r12`. Similarly the $\boxed{\text{c6}}$ in **EX** tells us that the `rt` source of `0x110c` is produced by `0x1108`; it also tells us that `0x110c` is a store instruction, the store value must come from `r14`.

Less-Common Dependency: The $\boxed{\text{c6}}$ in **ME** tells us that `0x1108` uses a result from an instruction in **WB**, `0x1004` (this bypass path can only bypass values that are to be written to the `lo` register). The `lo` register is the destination of `0x1104`, which must be a `mult` because no other instruction uses the **Y** units, and the source of `0x1108`, which must be a `mflo` because that is the only instruction that uses the `lo` register as a source. Note that the `lo` register is not shown in assembly language.

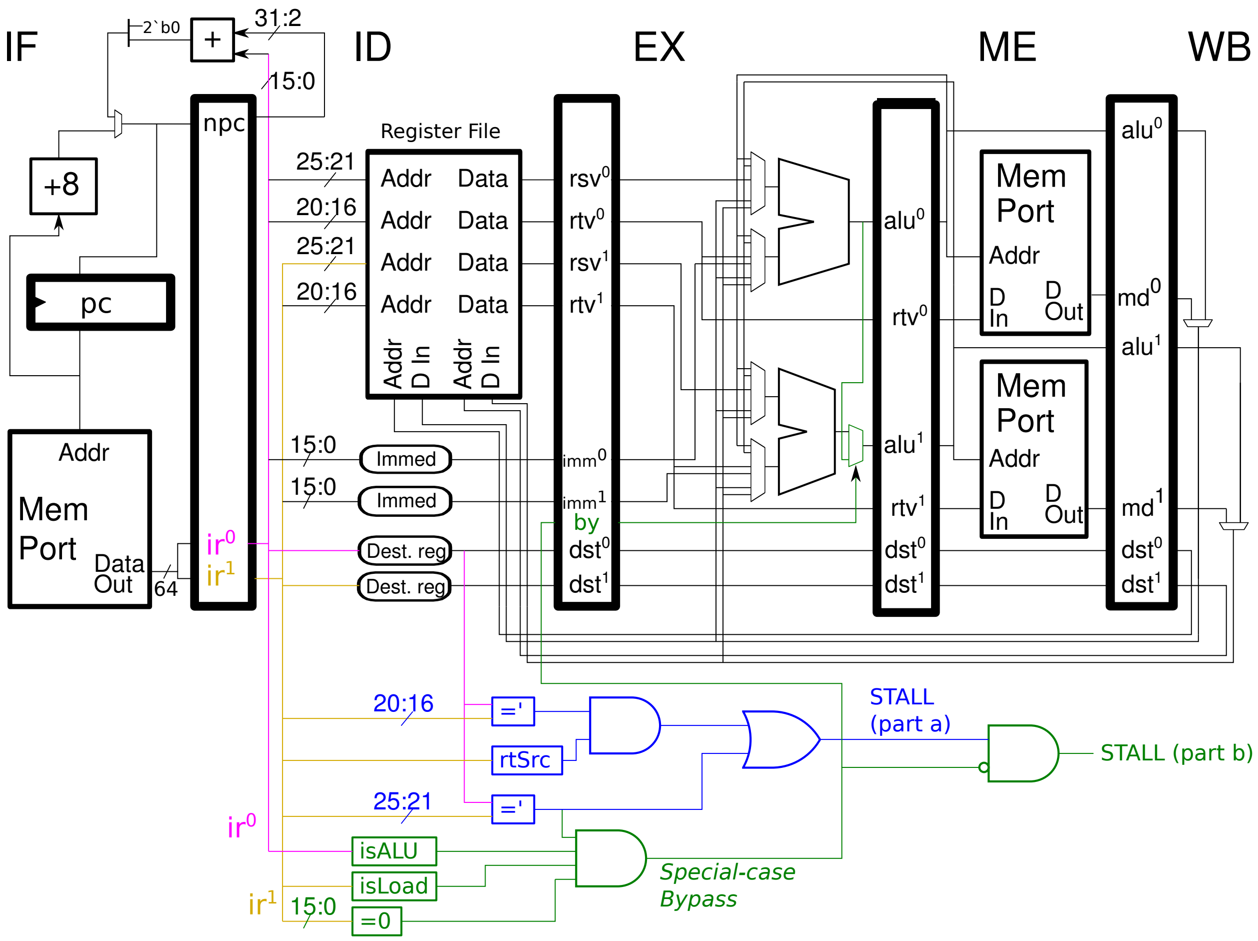
Parsing: The $\boxed{\text{c2}}$ in `ID` tells us that `0x1004` is a type I instruction. We have the encoded form of the instruction in `IF`: $\boxed{\text{c1: 0010 0010 0010 1100 0000 0000 0000 0001 0010}}$. From this we can parse out the `rs` and `rt` registers (`r17` and `r12`), as well as the immediate (18). We can also parse the opcode, 8, but one did not have to know that the opcode was for an addi instruction.

The Store Instruction: As mentioned earlier the $\boxed{\text{c6}}$ in **EX** reveals that `0x110c` is some kind of store instruction. The $\boxed{\text{c7:0x9001}}$ tells us that it cannot be a `sw` because the address of a `sw` must be a multiple of 4, similarly it can't be a `sh`. The only common store remaining is `sb` (store byte).

The `mult` and `mflo` Instructions: See the less-common dependencies paragraph above.

The Branch: The pattern of instruction addresses tells us that the instruction at `0x1000` must be some kind of delayed control transfer (branch, jump, call). The $\boxed{\text{c1: FILL IN}}$ in `ID` tells us it must be a branch. Let i denote the immediate field value (which is what is needed for the fill-in box. The branch target is computed by adding $4\times$ the immediate to the delay slot address, that is: `0x1104 = 0x1004 + 4i`. From that we get $i = 40_{16}$ (or `0x40`).

Problem 2: (15 pts) Illustrated below is a 2-way superscalar MIPS implementation. Design the hardware described below. You can use the following logic blocks (with appropriate inputs) in your solution: The output of logic block $\boxed{\text{isALU}}$ is 1 if the instruction's result is computed by the integer ALU. The output of logic block $\boxed{\text{rtSrc}}$ is 1 if the instruction uses the `rt` register as a source. The output of logic block $\boxed{\text{isLoad}}$ is 1 if the instruction is a load.



(*a*) Design logic to generate a signal named `STALL`, which should be `1` when there is a true (also called data or flow) dependence between the two instructions in `ID`.

☑ Control logic to detect true dependence in `ID` and assign `STALL`.

Solution appears above in blue. The stall signal for this part is the output of the OR gate. To help understand what's going on the IR (instruction register) for slot 0 is shown in purple and the IR for slot 1 is shown in gold. The logic compares the destination of the instruction in ID slot 0 with the two sources of the instruction in slot 1. If either matches, the stall signal is generated (which before part b would be the output of the OR gate). The $\boxed{\text{rtSrc}}$ logic is needed because the `rt` field might hold a destination register number, and we would not want to stall for that.

(*b*) The code fragment below should generate a stall in our two-way superscalar implementation when the two instructions are in the same fetch group. However this particular arithmetic/load pair is a special case in which the stall is not necessary when the right bypass path(s) and control logic are provided. *Hint: Something about the* `lw` *makes it a special case.*

```
0x1000: add r1, r2, r3
0x1004: lw r4, 0(r1)
```

☑ Add the bypass path(s) needed so that the code executes without a stall.

The bypass path appears at the output of the slot-1 ALU. The slot 0 ALU computes the add (for the example above) and the slot 1 ALU computes `r1 + 0`. For this special case, where the immediate is zero and we are using the destination of the slot-0 instruction, we can bypass.

☑ Add control logic to detect this special case and use it to suppress the stall signal from the first part.

The control logic appears in green. The logic checks for an ALU instruction in slot 0, a load with a zero immediate in slot 1, and a dependence between the slot-0 and -1 instructions. If all are true the special-case-bypass signal is 1, and that is used to control the new bypass multiplexor and to suppress the stall signal.

Note that the bypass signal must go through the pipeline latch, if it were connected directly to the green EX-stage mux it would be affecting the wrong instructions.

Problem 3: (20 pts) Code producing the branch patterns shown below is to run on three systems, each with a different branch predictor. All systems use a 2^{14} entry BHT. One system has a bimodal predictor, one system has a local predictor with a *12-outcome local history*, and one system has a global predictor with a *12-outcome global history.*

(*a*) Branch behavior is shown below. The `r` outcomes for branch `B2` are random and are modeled by a Bernoulli random variable with $p = .25$ (taken probability is .25). Answer each question below, the answers should be for predictors that have already warmed up.

```
B1:  T    N    T    T    T    N     T    N    T    T    T    N     ...
B2:   T    N    T    r    T    N     T    N    T    r    T    N    ...
B3:    T    T    T    T    T    T     T    T    T    T    T    T   ...
```

☑ What is the accuracy of the bimodal predictor on branch `B1`?

By straightforward analysis (see work below) the $\boxed{\text{accuracy is } \frac{4}{6} = \frac{2}{3}}$. Note that the accuracy is based on a repeating pattern (the part under the dashes). The pattern of outcomes repeat, and the counter value is the same, 2, at the beginning and the end of the repeating part.

```
SOLUTION WORK
Repeating pattern:             -------------------------------
Ctr:0   1     0    1    2    3     2    3    2    3    3    3  2
B1:  T    N    T    T    T    N     T    N    T    T    T    N     ...
Mispred:                                x                   x
```

☑ What is the approximate accuracy of the bimodal predictor on branch `B2`? ☑ Explain.

Short answer: The 2-bit counter value will be 2 or 3, and so all the taken branches will be correctly predicted, for a $\boxed{\text{prediction ratio of } \frac{3+0.25}{6}}$.

Explanation: Notice that there never will be more than one consecutive not-taken outcome and that sometimes there are three consecutive taken outcomes (when `r` is taken). For that reason the 2-bit counter cannot be lower than 2 (after reaching a 3 during warmup). With a value of 2 or 3 it will always predict taken, that is the correct prediction for the three `T` outcomes and is correct for `r` 25% of the time. The total accuracy is then $\frac{3+0.25}{6}$.

☑ What is the minimum local history size needed to predict `B1` with 100% accuracy? Ignore branch `B2` for this question.

The minimum history size is $\boxed{\text{4 outcomes}}$. To see why consider the six possible local histories (sorted) below. Suppose the local history size were just two outcomes, and consider the first two patterns below. If the local history were `NT` the next outcome might be `N` or `T`, and so the branch could not be predicted. Similarly three outcomes are insufficient, consider local history `TNT`. But four are sufficient, that's easy to tell in this case because every four-outcome history for this branch is different (look at the first four columns of the table below).

```
NTNTTTNT
NTTTNTNT
TNTNTTTN
TNTTTNTN
TTNTNTTT
TTTNTNTT
```

☑ What is the accuracy of the local predictor on branch `B2`, ignoring the effect of `B1`? ☑ Explain.

The local predictor will predict the non-`r` outcomes with 100% accuracy because the pattern repeats and there is enough local history. The `r` outcomes add to the number of patterns, but other than increasing warmup time they don't affect accuracy of the non-`r` outcomes. Since `r` is biased not-taken the counter for the `r` outcomes will usually be 0 or 1. Assuming it is always 0 or 1, the accuracy or the `r` outcomes would be 75%. The overall accuracy under this assumption is $\boxed{\frac{5+0.75}{6}}$. It's easy to find the actual probability of each counter value by solving a simple Markov chain. The probability of a counter value of i is $\frac{\omega-1}{\omega^4-1}\omega^i$ where $\omega = \frac{0.25}{1-0.25}$. The probability of a zero count is .675 and a one count is .225, so the probability of predicting not-taken is .9. The probability of a correct prediction for the `r` outcome is $0.9 \times 0.75 + 0.1 \times 0.25 = .7$, slightly lower than the .75 we estimated.

$\boxed{\checkmark}$ Describe a situation in which branch `B2` is mispredicted using the local predictor due to the effect of `B1`. The low bits of the address of `B1` and `B2` are different so there is no chance of a BHT collision. How frequently do such mispredictions occur? *Grading Note: The original exam and the Spring 2014 homework question did not include the statement about the low address bits.*

This happens when the `r` outcome is `t` two times in a row, and then is `n`. When we predict the `T` outcome (the third `r` outcome) the local history for `B2`, `tTNTNTtTNTNT`, matches a possible local history for `B1`. `B1` will have warmed up the corresponding PHT entry to 3, but for branch `B2` the predicted outcome should be not taken. The chance of this happening is $\frac{1}{4}\frac{1}{4}\frac{3}{4} = \frac{3}{64}$.

Thank you to EE 4720 student Payon Huskins for providing a correction to the solution above on 4 May 2015. (The version above has the correction applied.)

The original exam and the Spring 2014 homework assignment did not have the statement about the low address bits. The following explains why a BHT collision could not cause a misprediction, meaning that the only the answer above is correct, with or without the low-address-bits statement.

If bits 15:2 of `B1` and `B2` were identical the two branches would share a BHT entry and so their local histories would be intermingled (in the same way local histories are intermingled in the GHR). The positions of the N's in the local history would ensure that different PHT entries were used for the two branches, and so there is no chance of `B1` affecting `B2`. Even with the intermingling there would be enough local history to predict `B1` and `B2` with their respective maximum accuracies.

$\boxed{\checkmark}$ How many possible GHR values will be present when predicting branch `B3`? It's okay to show patterns that include `r`'s, but indicate how many of each pattern there are.

Consider the following global history when predicting `B3`: `TttTtrTttTnn`. The upper-case outcomes are `B3`, the lower-case outcomes are `B1` and `B2`. Because there is an `r` this pattern represents two possible global histories. Pattern `B1` can be in six possible rotations, for four of them the `r` is in the global history, so the total number of global histories when predicting `B3` is $\boxed{2 \times 4 + 2 = 10}$.

Problem 3, continued: Recall that the local predictor uses a 12-outcome local history and a 2^{14}-entry BHT.

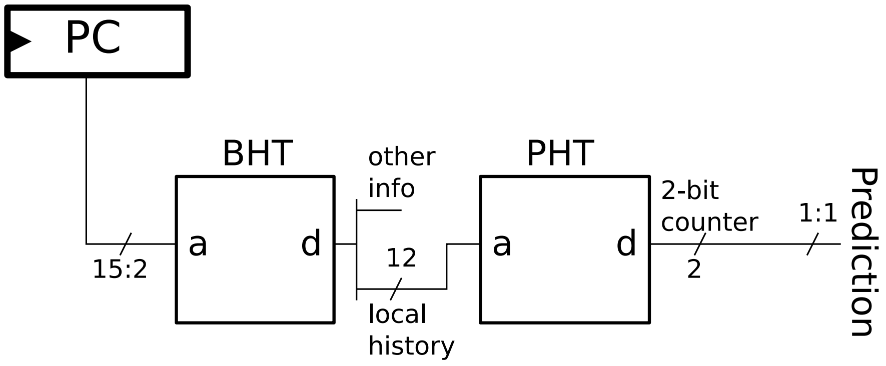
(*b*) Draw a diagram of the local predictor hardware used for making a prediction but omit the hardware for updating the predictor. The input to your hardware should be `PC`, and output should be a 1-bit quantity (`0` for not-taken, `1` for taken).

- Be sure to show the BHT and PHT.
- Assume that `PC` is the address of a branch. (That is, don't worry about detecting non-branch instructions.)
- Don't predict the target, just the direction (taken or not-taken).

☑ Local predictor hardware for predicting branch direction.

☑ Label wires with bit ranges (*e.g.* 31:26) or number of bits, as appropriate.

Solution appears below.



(*c*) How much memory is needed to implement the local predictor. Only include storage for predicting the branch direction, do not include storage for predicting the branch target.

☑ Storage needed to implement local predictor. ☑ Indicate unit.

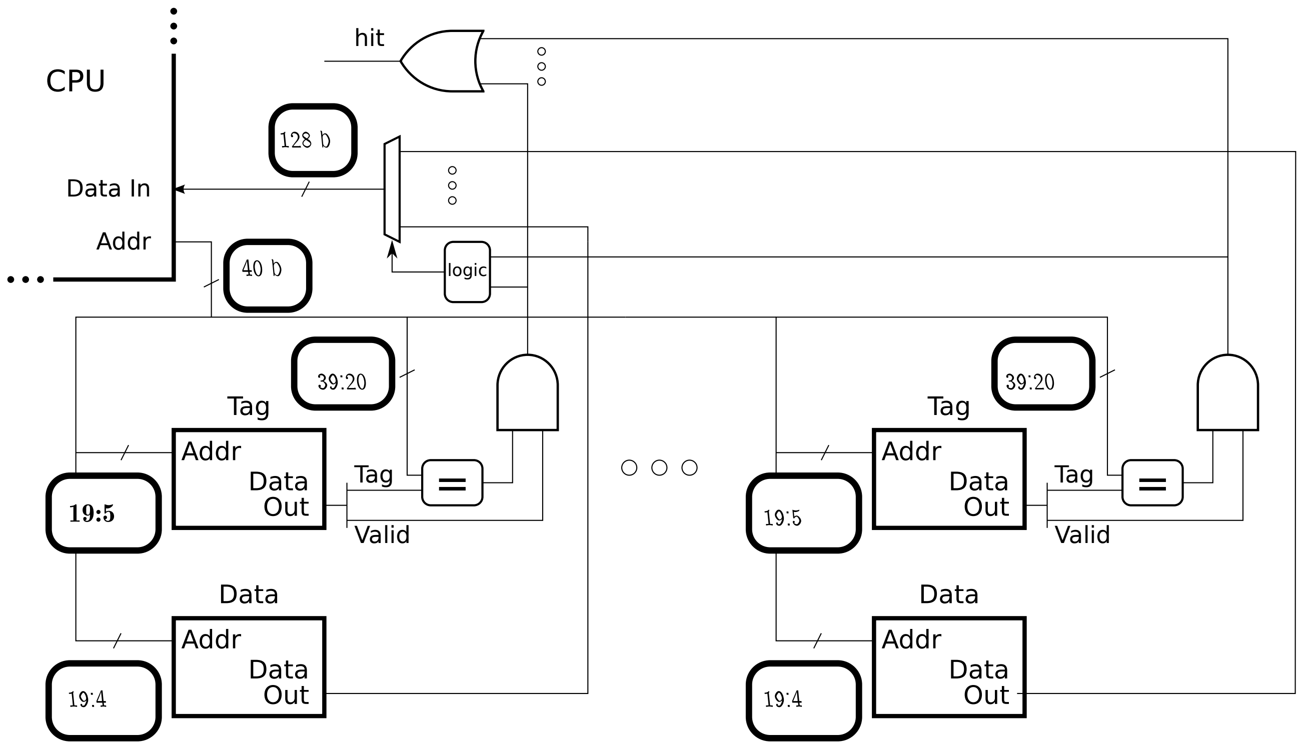
In real life [tm] a BHT entry for a local predictor might have a field for identifying the type of instruction (branch, jump, non-CTI, *etc.*), a tag, the CTI target, and of course the local history. As requested, we will only look at the local history. The size of the local history is 12 bits, and there are 2^{14} entries. The PHT has 2^{12} entries (one for each possible local history) and each entry is 2 bits.

The total storage is thus $\boxed{2^{14} \times 12 + 2^{12} \times 2 \text{ bits}}$.

Problem 4: (20 pts) The diagram below is for an 8-way set-associative cache. Hints about the cache are provided in the diagram and also in the address bit categorization just below the diagram.

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

[✓] Fill in the blanks in the diagram.



[✓] Complete the address bit categorization. Label the sections appropriately. (Index, Offset, Tag.)

	Tag	Index	Offset	
Address:	39 … 20	19 … 5	4 … 4	3 … 0

[✓] Cache Capacity (Indicate Unit!!):

The cache capacity can be determined from the following pieces of information: The lowest tag bit position is 20, which means that the size of the combined index and offset is 20 bits, and so each data store holds 2^{20} characters. Nothing was said about the character size and so we can assume that it is 8 bits, which we will call a byte. We were told that the cache is 8-way set associative, and so there are 8 data stores, and so the $\boxed{\text{cache capacity is } 8 \times 2^{20} = 8\,\text{MiB}.}$

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

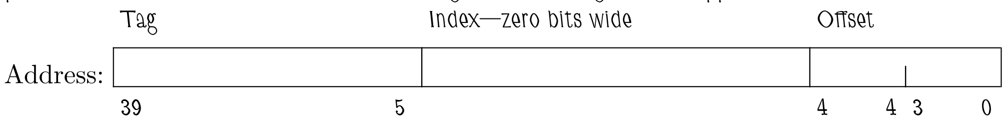
It's the cache capacity, 2^{23} bytes, plus $8 \times 2^{20-5}\,(40 - 20 + 1)$ bits.

[✓] Line Size (Indicate Unit!!):

Lower bit position of the address going into the tag store gives the line size, $2^5 = 32$ characters.

☑ Show the bit categorization for a **fully associative** cache with the same capacity and line size.

Because the cache is fully associative the number of index bits is zero. The line size doesn't change so we don't change the offset bit positions, instead we increase the size of the tag. The bit categorization appears below.



Problem 4, continued: The problems on this page are **not** necessarily based on the cache from the previous page. The code in the problems below run on a 4 MiB (2^{22} byte) 4-way set-associative cache with a line size of 256 bytes.

Each code fragment starts with the cache empty; consider only accesses to the arrays.

(*b*) Find the hit ratio executing the code below.

```
  Complex sum = {0,0};
  Complex *a = 0x2000000; // sizeof(Complex) == 16  Each element loaded with 1 load insn.
  int i;
  int ILIMIT = 1 << 11;     // = 2^11

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

☑ What is the hit ratio running the code above? Show formula and briefly justify.

The line size of $2^8 = 256$ bytes is given. The size of an array element, of type complex, is $16 = 2^4$ characters, and so there are $2^8/16 = 2^{8-4} = 2^4 = 16$ elements per line. The first access, at `i=0`, will miss but bring in a line with 2^4 elements, and so the next $2^4 - 1 = 15$ accesses will be to data on the line, hits. The access at `i=16` will miss and the process will repeat. Therefore the $\boxed{\text{hit ratio is } \frac{15}{16}}$.

(*c*) Find the smallest value for `ASIZE` that will minimize the hit ratio (make things as bad as they can get) of the code below.

```
struct Some_Struct {
  double val;         // sizeof(double) = 8
  double norm_val;
  double a[ASIZE];  };

  const int BSIZE = 1 << 10;
  Some_Struct *b;
  for ( int i = 0;  i < BSIZE;  i++ ) sum += b[i].val;
  for ( int i = 0;  i < BSIZE;  i++ ) b[i].norm_val = b[i].val / sum;
```

☑ Smallest value of `ASIZE` to minimize hit ratio:

This problem requires some thought.

When `ASIZE` is zero the element size is 16 and a 256-byte line can hold 16 elements. The first `for` loop will enjoy a $\frac{15}{16}$ hit ratio (see the previous part). The second loop, because the cache size is 4 MiB, will have a 100% hit ratio. As we increase `ASIZE` from 0 the hit ratio first drops due to the loss of spatial locality. The drop in hit ratio will temporarily level off when `sizeof(Some_Struct)` is equal to the line size. That occurs when $(2 + s)8 = 256$, where s is `ASIZE`, the value is 30. With this value the hit ratio in the first `for` loop is zero, but the hit ratio in the second `for` loop is still 100%. This answer, `ASIZE=30`, would only get partial credit because a smaller hit ratio is possible.

Increasing `ASIZE` to, say, 40, won't reduce the hit ratio from its `ASIZE=30` value, but there is a point where it will start dropping again. If `ASIZE` were $2^{17} - 2$ then the structure size would be $(2 + 2^{17} - 2)8 = 2^{20}$ bytes. For this cache two consecutive elements of the structure would have the same index but different tags. Since the cache is 4-way it could only hold no more than four elements, which means the second `for` loop would have a hit ratio of only 50% (it's not zero because a miss to the `val` member brings in the data for the `norm_val` member). This answer, `ASIZE=` $2^{17} - 2$ would only get partial credit because it's possible to get the same hit ratio with a smaller value of `ASIZE`.

With `ASIZE` set to $2^{17} - 2$ the cache can only hold four elements (the part with `val` and `norm_val`). Suppose the cache could only hold `BSIZE/2` elements (or for that matter, `BSIZE-1` elements). Then the hit ratio of the second `for` loop would be just 50%, the minimum value. The minimum `ASIZE` is for the case where eight elements have the same tag: if more than eight elements have the same tag (as in the previous paragraph) then `ASIZE` is larger than it has to be. If fewer than eight (four is the next smaller value) have the same tag then there will be no misses in the second `for` loop. If eight elements have the same tag then there must be `BSIZE/8=128` different indices. That means $128(s+2)8 = 2^{20}$. Solving yields $s = 2^{10} - 2 = 1022$, the full-credit value of `ASIZE`.

Problem 5: (25 pts) Answer each question below.

(*a*) Indicate whether each feature below is a feature of an ISA or the feature of an implementation.

☑ Number of stages in pipeline. *Circle One*: ISA or [✓ Implementation]

☑ Number of integer registers. *Circle One*: [✓ ISA] or Implementation

☑ Number of bits in an integer register. *Circle One*: [✓ ISA] or Implementation

☑ Whether an adjacent pair of instructions, such as `add r1,r2,r3; sub r4,r1,r2` will generate a stall. *Circle One*: ISA or [✓ Implementation]

(*b*) Describe the problem with each of the following MIPS code fragments.

☑ Problem with fragment below:

```
addi r1, r2, 0x123456
```

The immediate value is too large, it must be limited to 16 bits. Fixed code appears below:

```
# SOLUTION
lui r1, 0x12
ori r1, r1, 0x3456
add r1, r2, r1
```

☑ Problem with code execution on our FP pipeline.

```
add.s f1, f2, f3  IF ID A1 A2 A3 A4 WF
sub.s f4, f1, f5     IF ID A1 A2 A3 A4 WF
```

There is a dependency carried by `f1`, the `sub.s` should have stalled. The correct execution appears below.

```
# SOLUTION
add.s f1, f2, f3  IF ID A1 A2 A3 A4 WF
sub.s f4, f1, f5     IF ID -------> A1 A2 A3 A4 WF
```

☑ Problem with code execution on our 5-stage pipeline.

```
lw r1, 2(r3)      IF ID EX ME WB
beq r1, r4 TARG      IF ID EX ME WB
xor r6, r7, r8          IF ID EX ME WB
TARG:
add r9, r10, r11           IF ID EX ME WB
```

There is a dependence between the `lw` and `beq` carried by `r1`, the branch should stall.

(*c*) Consider the following two equal-cost design options:

A dual-core chip, each core is 4-way superscalar and dynamically scheduled.

A 16-core chip, each core is 2-way superscalar and statically scheduled.

Both have the same clock frequency.

☑ What is the peak execution rate of each chip, in IPC?

For the dual core the rate is $2 \times 4 = 8\,\mathrm{insn/cycle}$.

For the 16-core chip the rate is $16 \times 2 = 32\,\mathrm{insn/cycle}$.

☑ Describe a situation in which the dual-core chip is a better choice than the 16-core chip.

If we are running code with only one thread the dual core can execute it at up to $4\,\mathrm{insn/cycle}$, but the second can only reach a peak of $2\,\mathrm{insn/cycle}$.

☑ Describe a situation in which the 16-core chip is a better choice than the dual-core chip.

If a program had 16 threads, and there were at most minor inefficiencies due to parallelization, the 16-core chip would be the better choice. Or, you needed to run 16 different programs at the same time. Either way, the 16-core chip is better because of its higher peak.

(*d*) Consider a $5n$-stage implementation created by splitting each stage of a 5-stage implementation into n pieces.

☑ How important is it to have a higher clock frequency in the $5n$-stage design (compared to the 5-stage design)? ☑ Explain.

It's very important. Both the original and $5n$ stage pipeline have a peak execution rate of $1\,\mathrm{insn/cycle}$, so if the clock frequency isn't higher there is no gain in performance.

(*e*) When an instruction raises an exception execution will switch to a trap handler.

☑ How is the trap handler address determined for MIPS?

The MIPS ISA specifies a handler address for each category of interrupt.

☑ How is the trap handler address determined for SPARC?

SPARC has a special register called the Trap Base Register (TBR). The TBR has the address of the start of what's called a trap table. Hardware will generate a trap type, the value of the trap type is determined by the kind of exception. The trap type is combined with the value in the TBR to form the handler address.