CSE352 Autumn 2013 Homework #7

Instructor: Mark Oskin
TAs: Vincent Lee, Mark Wyse

Due In Class 12/6/2013
Version 1.3

Please write your name and student ID at the top right corner of each page, and staple or paperclip your work together. We are NOT responsible for losing papers that were not stapled or paperclipped together.

Complete the following questions. Please write legibly and try to draw clean diagrams. Spaghetti wiring in circuit diagrams is difficult to grade. We will not grade work that is too heavily encrypted for us to read (i.e. we can't read it, we can't grade it). Please consider typesetting your work if you think that it may not be legible to the grader. You are encouraged to collaborate with your peers but you must turn in your own work. Justice will be enforced if you are caught cheating.

Select and complete two problems that you find most interesting of the first three. Each problem has several parts. Problems 4 and 5 are optional.

Problem 1 *Branch Prediction*

Suppose we use a TAKEN/NOT TAKEN scheme for our branch predictor. In this implementation, we keep a lookup table that takes the lower 16 bits of the branch instruction address and looks to see if the branch was TAKEN or NOT TAKEN the last time this entry in the lookup table was accessed. For instance, if an instruction at address 0x80001000 was TAKEN, the next time an instruction with address ending in 0x1000 is seen, the predictor will predict TAKEN. If the prediction is correct, no update occurs to the branch predictor's lookup table.

If the prediction is incorrect, the lookup table will be updated with the correct direction of the branch. For example, if the branch predictor predicted TAKEN for an address ending in 0x1004 but the actual outcome is NOT TAKEN, then the entry in the lookup table for 0x1004 will be updated to NOT TAKEN so that the next time an address ending in 0x1004 is seen, the predictor will predict NOT TAKEN.

Assume that all entries in the lookup table are initialized to NOT TAKEN.

(a) For the following sequence of branch addresses, and branch outcomes, fill out the corresponding branch predictions. The first few are done for you:

Address	Prediction	Outcome
0x1000004	NOT TAKEN	TAKEN
0x2000004	TAKEN	NOT TAKEN
0x2000032	NOT TAKEN	TAKEN
0x200003C		TAKEN
0x1000004		TAKEN
0x2000004		NOT TAKEN
0x2000032		NOT TAKEN
0x2000018		TAKEN
0x1000016		NOT TAKEN

(b) For the following programs, explain whether you expect this branch predictor to perform well or to perform poorly. Justify your answer.

(i)
```
int main() {
    int LOL = 0;
    for (int i = 0; i < 1000; i++)
        LOL++;
    }
}
```

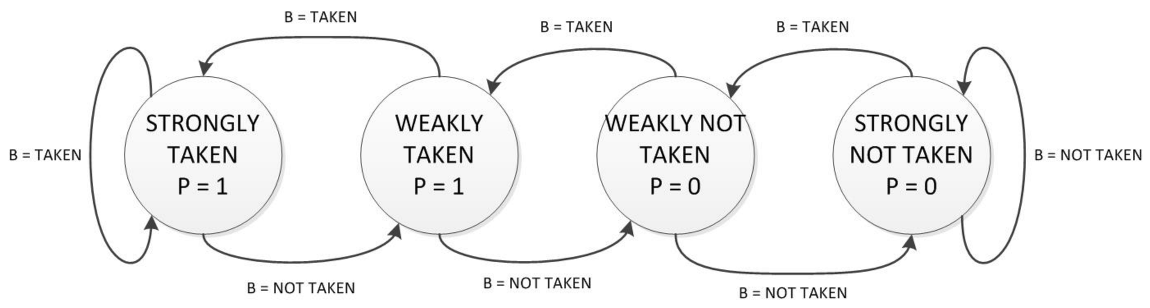
(ii)
```
int meow(int mix) {
    if (mix % 2 == 1)
        return mix + 1;
    else
        return mix - 1;
}
int main() {
    int temp = 0;
    temp = meow(temp);
    temp = meow(temp);
    temp = meow(temp);
    temp = meow(temp);
}
```

(iii)
```
int meow(int mix) {
    if (mix % 2 == 1)
        return mix + 1;
    else
        return mix - 1;
}
int main() {
     int j = 0;
    do {
        meow(j);
        j++;
    } while (j < 10)
}
```

(c) For what sequence of branch patterns and outcomes does this branch predictor fail spectacularly (> 90% mispredict rate)?

(d) Suppose we augment our branch predictor and use a STRONGLY TAKEN, WEAKLY TAKEN, WEAKLY NOT TAKEN, STRONGLY NOT TAKEN scheme. In this scheme, we initialize the lookup table to WEAKLY NOT TAKEN. If a lookup value is either STRONGLY TAKEN or WEAKLY TAKEN, then the branch predictor predicts TAKEN, otherwise the predictor predicts NOT TAKEN.

The following FSM describes how the predictor for a given lookup entry is updated:



The value P is the value of the prediction for this entry, while B is the outcome of a branch.

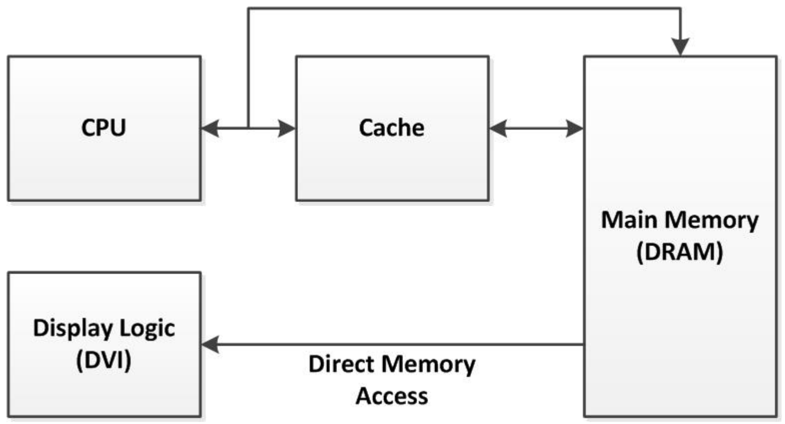
For the following sequence of branch addresses, fill out the following table for this new branch predictor:

Address	Old Predictor	New Predictor	Outcome
0x10000004	WEAKLY NOT TAKEN	WEAKLY TAKEN	TAKEN
0x10010004	WEAKLY TAKEN	STRONGLY TAKEN	TAKEN
0x10020008			NOT TAKEN
0x10120004			TAKEN
0x10000004			UNTAKEN
0x10000004			TAKEN
0x10000016			TAKEN
0x10010004			TAKEN
0x10000004			UNTAKEN
0x10300004			TAKEN

(e) Now write a simple C program that when run on the new branch predictor achieves at least a 10% prediction accuracy improvement over the original predictor in part A.

Problem 2 *Framebuffers and Simple Graphics Accelerators*

In this problem we will explore the concepts behind how images are created and read in hardware. A commonly used technique to handle graphics is to use allocate a frame buffer in memory and designate that region as the memory that the display hardware should read out of to get the display data via direct memory access. In this design, the display hardware operates in parallel with the CPU starting at the base address of the framebuffer to the end of the image and repeats. Assume throughout this problem that the main memory has a sufficient number of read and write ports to handle all the memory requests and services them appropriately.



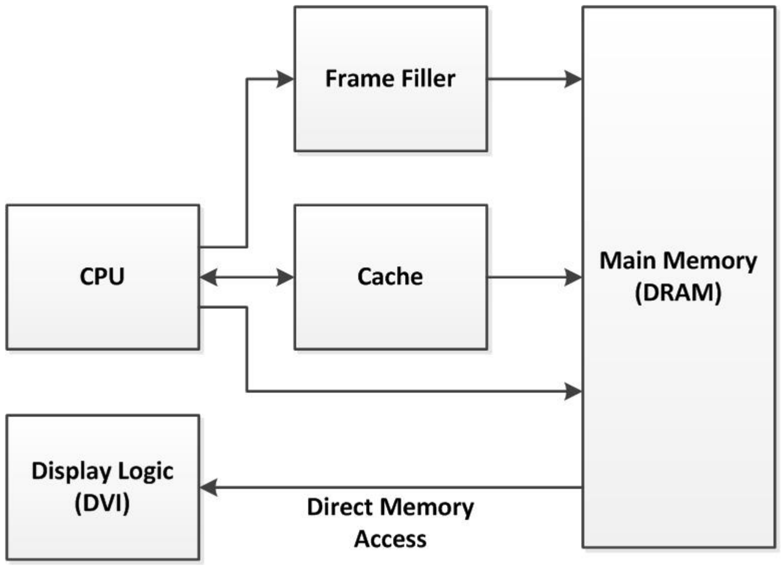
Simple Display Architecture

(a) Recall that an RGB image composes of 24 bit pixels, 8 bits per color channel. For simplicity, in this problem we will assume that an RGB image takes 32 bit so that they are word aligned and just zero pad the upper 8 bits such that the bit fields are {8'b0, red[7:0], green[7:0], blue[7:0]}. With this encoding, for an 800 x 600 RGB image, how many bytes are required to represent this image? Notice this is also the amount of memory required to store one frame of this image.

(b) Now suppose we allocate enough memory for one frame starting at address 0x00100000 in main memory which we will write pixel values to. This newly allocated region is called a framebuffer and 0x00100000 is the base address of the frame. One way we can create and image on the display is to run software on the CPU and write pixel values to the framebuffer for each pixel. Using volatile variables where appropriate, write a short C program that would fill the entire framebuffer with the color red (0x00FF0000).

(c) Suppose we are running on a 3 GHz processor; if our frame filling operation with red takes about 10 cycles to write each pixel, what percentage of every second is spent writing to the framebuffer if we try to achieve a 30 Hz framerate target? What if you were running a more complicated image and it required 100 instructions to compute each pixel value? What if we also increased the target framerate target to 60 Hz?

(d) The above solution is agonizingly slow and the computation ties up the CPU which could be doing other important things like running Starcraft II or loading your Facebook page. To make graphics rendering tractable we use accelerators or dedicated graphic processing units (GPUs) to offload the computation. Suppose we modify the architecture so that we have a frame filler accelerator as follows:



Display Architecture with Frame Filler Accelerator

Suppose we have the following memory mapped register locations that implement a ready valid interface with the accelerator:

Address	Value	Description
0xF0000000	Color	The color to fill the frame with (Write Only)
0xF0000004	Ready	The ready status of the frame filler. (Read Only)
0xF0000008	Valid	The valid status. Write 1 to this register to fire off the accelerator

Using volatile variables where appropriate, write C code that will fill the frame with red using this accelerator and interface. Notice that this solution drastically reduces the number of instructions run on the CPU to write the image.

Problem 3 *Cache Side Channel Attacks*

A hardware side channel attack is a method of compromising sensitive information about the target system such as what processes may be running or exposing RSA keys by exploiting aspects of the hardware architecture such as the cache or branch predictor. In this question we will explore the concept of a side channel attack targeting the cache, which is a central component to all commodity processors.

Assume for simplicity that our cache has only an L1 cache and is an 8 KB direct mapped, writeback, allocate, cache with 8 words per line, and 32 bits per word.

(a) Suppose the current state of each cache line is invalid (cold cache). How many memory accesses are necessary in order to fill the cache with all valid lines?

(b) For the following sequence of memory accesses, indicate whether the memory access was a hit or a miss. Assume the cache is initially cold.

Address	Hit?
0x00000000	
0x00000004	
0x00000020	
0x00000040	
0x00000080	
0x0000003C	

(c) Suppose a cache hit for our processor is one cycle, while a cache miss takes 50 cycles to refill the cache with the correct data. If the cache cycle time is 5 ns, what is the average memory access time (AMAT) for the memory access pattern in the previous part of the problem?

(d) Now suppose we have the following malicious code running on a thread of our machine. In terms of cache behavior, what does the following code do? Assume that the garbage array is allocated a contiguous piece of memory.

```
int64_t garbage[1024];
int64_t read;
while (true) {
   for (int i = 0; i < 1024; i++) {
      read = garbage[i];
   }
}
```

(e) Now suppose the malicious code has been modified to execute the following code. The function time() returns the processor time in picoseconds which you may assume is a 64 bit number. You may assume the amount of time to process this call is negligible.

```
int64_t garbage[1024];
int64_t read;
int64_t time;
while (true) {
   for (int i = 0; i < 1024; i+=4) {
      time = time();
      read = garbage[i];
      time = time - time();
```

```
        garbage[i] = time;
    }
    analyze(garbage);
}
```

Now what does the above code do? What is now stored in the array garbage?

(f) Now suppose there is another thread which we will refer to as the victim thread that is running on the same core and multiplexes time on the same processor. What does the malicious code now store in the garbage array? How can this information be used to analyze what is running on the victim thread? Your explanation should include a discussion of the cache behavior.

(g) Suppose our malicious thread recorded the following memory access pattern; what kind of program can you speculate is running on the victim thread? Notice that this data mining and speculation is possible as a consequence of how memory hierarchies (and therefore caches) are implemented in hardware.

```
0x00000400
0x00000200
0x00000300
0x00000380
0x00000340
0x00000360
0x00000360
0x00000360
```

Problem 4 *(Optional) Tournament Predictor*

The Tournament Predictor was a branch prediction module that was first used on the Alpha 21264 processor. At a high level, succinctly explain how this branch predictor shown in Figure 2 works. Don't just copy what they say in the paper; paraphrase the important aspects of the design.

A description and block diagram of the predictor can be found at:
http://www.cis.upenn.edu/ milom/cis501-Fall05/papers/Alpha21264.pdf

Problem 5 True/False/I Don't Care

[True/False/I Don't Care] This is the last question on the last problem set of this quarter.

This page left intentionally blank. WAT?