Exam in EDAF15 Algorithm Implementation

May 26, 2012, 8-13

Inga hjälpmedel!

Examinator: Jonas Skeppstedt, tel 0767 888 124

30 out of 60p are needed to pass the exam.

1. (4p) In a pipelined RISC processor, an individual instruction does not execute in fewer clock cycles. How is it then possible that a complete program takes fewer clock cycles with pipelining than without pipelining? If we have N pipeline stages, will all programs become N times faster? Why or why not?

   Answer *See page 148 in the book. No, they will not. Mispredicted branches (or taken but not predicted at all), cache misses, loads directly followed by a use of the loaded value, and other situations cause pipeline stalls.*

2. (6p) In a superscalar processor speculative execution is very important. What is it, why can it make execution faster, and how is chaos prevented (for instance, that a register or memory location is modified only if it should be)?

   Answer *See pages 149-150 in the book.*

3. (10p) What is the purpose of having sets in a cache —

   Answer *The purpose is to reduce the risk of two or more addresses (in use at the same time) being mapped to the same row in the cache, which would result in the data being overwritten many times with the effect of many cache misses.*

   (2p) why are caches not "simplified" by having only one set?

   Answer *Then we would need too many comparators (which must operate in parallel in order not to be too slow) which would not be useful. Chip area can be used more cleverly to other things.*

   Suppose a cache is divided into sets where each set contains four cache blocks. (1p) What is such a cache called?

   Answer *4-way set associative cache.*

   (2p) With the same cache, can the data at a memory address A be put in any set? Why or why not?

Answer *No, there is a function which maps the address to a specific set.*

(2p) In any of the four cache blocks? Why or why not?

Answer *Yes. That's why we have four comparators.*

(2p) What is meant by cache levels and

Answer *Two or three levels of caches are used. Level 1 is the smallest and fastest. The data in level i is a subset of that in level $i + 1$.*

(2p) what could the reason be for having separate first level caches for data and instructions?

Answer *The reason is that both instruction fetch and data fetch should be performed at the same time, and these two caches only very rarely have any data in common (the instruction cache does not check what is put into the data cache — the programmer must use explicit instructions to remove stale copies from the instruction cache).*

4. (10p) Implement the following functions for a stack of void pointers using the type `stack_t`, which you should make complete (see below). Your stack struct should contain an array to store the data in, which you can allocate e.g. using `malloc` or `calloc` When there is no more space left in the stack and `push` is called, you should somehow increase the memory area for the array. You can do this either using `realloc` (preferable) or `malloc` (not too good -1 point). Give two disadvantages with using `malloc` in this situation.

   Recall the declaration of `realloc`:

   ```
   void* realloc(void* ptr, size_t size);
   ```

   and that it tries to set the size of the allocated memory pointed to by `ptr` to `size`. If that fails it tries to allocate new memory using `malloc`, copying the old data to the newly allocated memory, and then `free`ing what `ptr` points to and finally returning the allocated pointer. If that also fails, it returns a null pointer.

   If `top` or `pop` are called with empty stacks, it's not your problem and you should not waste execution time checking it.

   ```
   typedef struct stack_t  stack_t;

   struct stack_t {
           /* ... */
   };

   /* create an empty stack. */
   ```

```
stack_t*        new_stack(void);

/* deallocate the stack and set what the parameter
 * points to to NULL. */
void            free_stack(stack_t** stack);

/* push data onto the stack. */
void            push(stack_t* stack, void* data);

/* remove and return the top of the stack. */
void*           pop(stack_t* stack);

/* return the top of the stack. */
void*           top(stack_t* stack);
```

Answer *See book on page 99-101. Two disadvantages of using* `malloc` *instead of* `realloc`*: you can run out of memory with* `malloc` *in a situation in which* `realloc` *would find memory, and* `realloc` *has a chance to be faster.*

5. (10p) Suppose you wish to understand why one implementation of an algorithm is faster than another when executed on a particular machine. How can the following tools be useful?

   - `cachegrind`
   - `gcov`
   - `gprof`
   - OProfile
   - A pipeline level simulator such as IBM's clock cycle true simulators for different implementations of the Power processor?

   Answer *See Chapter 6 in the book.*

6. (10p) Write a function to check that a void pointer `ptr` has a value which is a multiple of `size_t a` which is a power of two. If this is the case, you should return one and otherwise return zero.

   You will probably find the unsigned integer type `uintptr_t` useful. It can hold any address. You are not allowed to use the `%` operator.

   Explain why it works — illustrate with an example.

```
#include <stdint.h>

int aligned(void* ptr, size_t a)
{
        uintptr_t       b;

        b = (uintptr_t)ptr;

        return (b & (a - 1)) == 0;
}
```

Answer *We know that a is a power of two. Assume $a = 2^k$. As a binary number a is represented by a one followed by k zeroes. Subtracting one from a results in the number consisting of k ones. If the bitwise and of ptr and $a - 1$ is zero, then all the k least significant bits of ptr are zeroes, i.e. it is a multiple of a. Only integer numbers are allowed as operands to the bitwise and operator so therefore we cast the pointer to the unsigned integer type* `uintptr_t`.

For an example, in a similar situation, see page 81 in the book.

7. (10p) Write a function `count` which counts the number of ones in an `unsigned long long` (ie. the number of bits with value one in the binary number) as efficiently as you can. Your function should be more efficient than iterating through the number as many iterations as the number of bits in an `unsigned long long`. Estimate roughly how many clock cycles your function takes. State any assumptions you make.

   For instance, `count(7) = 3`

   Obviously, there are numerous different correct answers.

   Answer *The following solution solves the problem by counting the number of bits (using a table) in each byte of the number. The table can be generated by the following program:*

```
#include <limits.h>
#include <stdio.h>

int count(unsigned int a)
{
        int     i;
        int     ones;

        ones = 0;

        for (i = 0; i < CHAR_BIT; ++i) {
                if ((a & 1) == 1)
                        ones += 1;
                a >>= 1;
        }

        return ones;
}

int main(void)
{
        unsigned int    c;

        printf("static int table[1<<CHAR_BIT] = {\n");

        for (c = 0; c < (1<<CHAR_BIT); ++c)
                printf("\t%d,\n", count(c));

        printf("};\n");
}
```

Having executed that program we get the table so we can write the `count` *function as follows:*

```
#include <limits.h>

static int table[1<<CHAR_BIT] = {
        0,
        1,
        1,
        2,
        1,
        2,
        2,
        3,

/* parts of the table skipped to save space ... */

        7,
        6,
        7,
        7,
        8,
};

int count(unsigned long long a)
{
        int             i;
        unsigned char*  b;
        int             ones;

        b = (unsigned char*)&a;
        ones = 0;

        for (i = 0; i < sizeof a; ++i)
                ones += table[b[i]];

        return ones;
}
```