Contents Lecture 8

- Cache memories

Basic Rule

- There is a trade-off between speed and size of memories.
- A memory component is either fast and small or big and slow (or a compromise somewhere in between).
- Programmers of course want fast and big memories.
- It should take one clock cycle to fetch from memory and it should be several gigabytes big.
- Unfortunately, reading from memory can take one hundred clock cycles.
- So, we need to solve a problem!

Two Observations

- When solving a computer engineering research problem we should always first try to make observations about how the system behaves and then exploit that knowledge.
- A first observation about memory usage is that programs usually access memory somewhat regularly, which is called **locality**.
- **Temporal locality** means that if a program has accessed a particular location A in memory, it is likely it will access A again soon.
- **Spatial locality** means that if a program has accessed a word at address A in memory, it is likely it will soon access the word at address $A + 1$.

Examples of Temporal Locality

- The instructions in a loop are accessed next iteration as well.
- The loop index variable is usually accessed frequently.
- The stack space is often accessed again when a new function is called (since that space is reused).
- An object is typically accessed for a while and then the program is done with it.

Examples of Spatial Locality

- The instructions are accessed one after the other — until there is a branch.
- Elements of an array are often accessed one after the other.
- Often several variables in an object is accessed and if these are put close together (by the programmer) then there will be spatial locality.

Exploiting Temporal Locality

- What is needed is a small memory on the same chip as the processor.
- If we describe our **hardware** in C, then extending our machine.c could look like:

```
typedef struct {
        int                 reg[32];
        int                 pc;
        struct {
                bool        valid;
                int         data;
                int         address;
        }                   cache_array[8];
} cpu_t;
```

Data and Address

- We have now a cache which can store eight popular words.
- The cache array contains eight pairs of **data** and **address**.
- There is also a boolean called **valid** which tells us whether the data and address are valid for that row.
- Recall that on paper we can write question marks but the machine can only write ones and zeros and cannot see whether they make sense so a separate bit must tell whether they mean anything useful.
- Suppose the compiler has decided that a global variable X should be put at the address 293, or 0x125, or 0001 0010 0101.

Using our Cache

- When the program (or CPU) wants to read variable X, it should check whether any of the eight rows has **valid** = **true** and **address** = **293**
- If the CPU found one such row (or, let's call it **line**), then the CPU can take the data from that line and avoid waiting for the slow memory! Great!
- We must call this event something: **a cache hit**
- It can save us 100 clock cycles.

Load Instruction

- ***In hardware all iterations are executed concurrently!!***
- The openmp directive is here to make you alert on that this is ***not*** a sequential loop.

```
case LD:    found = false;
            address = source1 + constant;
            #pragma omp parallel for
            for (i = 0; i < 8; i++) {
                    if (cache_array[i].valid &&
                            cache_array[i].address == address) {
                            data = cache_array[i].data;
                            found = true;
                            break;
                    }
            }
```

Cache Replacement

- We need to select one line.
- If there is one line with **valid** = **false** then we select that one.
- Otherwise, for now, we take a random line (row).
- If the row we selected had valid data, we need to copy the old data contents to memory (otherwise it's lost).
- Then we read our data from memory.
- Then we write our data into the selected row, set the address to our address and set valid to true.

Load Continued

```
if (!found) {
    i = select_row();

    if (cache_array[i].valid) // save old data to memory
        memory[cache_array[i].address] = cache_array[i].data;

    // read our data from memory
    data = memory[address];

    // save our data in the cache
    cache_array[i].data = data;
    cache_array[i].address = address;
    cache_array[i].valid = true;
}
```

Similar for a Store

```
case ST:
    found = false;
    address = source2 + constant;
    data = source1;
    #pragma omp parallel for
    for (i = 0; i < 8; i++) {
        if (cache_array[i].valid &&
            cache_array[i].address == address) {
            cache_array[i].data = data;
            found = true;
            break;
        }
    }
    if (found)
        break;
```

Store Continued

```
i = select_row();
if (cache_array[i].valid)
    memory[cache_array[i].address] = cache_array[i].data;
cache_array[i].data = data;
cache_array[i].address = address;
cache_array[i].valid = true;
```

- Next time we want to read or write that variable it is likely that it will be found in the cache.

The Loop — isn't it slow?

- No, it doesn't exist!
- It only exists in the software model of the hardware.
- Recall: ***in hardware the loop is run in parallel.***
- In our case, there are eight so called **comparators** which compare the address requested with the address in its row and says **"here!"** if the addresses are equal and the valid bit is true.

A look at our Cache

- Our cache does not exploit spatial locality, yet.
- Instructions may also be put in the cache.
- Hit rate is the fraction of hits in the cache.
- Let us test it on the factorial program.

# rows	reads	read hits	writes	write hits	hit rate
8	77	21	11	0	23.9 %
16	77	36	11	0	40.9 %
32	77	50	11	0	56.8 %
64	77	50	11	0	56.8 %
128	77	50	11	0	56.8 %

What can have happened?

- All writes always miss — the factorial program writes to new places on the stack.
- The reads benefit from a larger cache since then what fac(5) and fac(4) saved on the stack will remain in the cache when they later read the saved parameter and return values.
- The 27 read misses are due to instruction which are fetched for the first time.

Some Comments

- Our cache is very small, only a half KB. Usually the cache closest to the processor, called the L1 cache is 32 KB or 64 KB.
- Having 128 comparators might be feasible but hundreds of them is too much. That is a problem which we must address.
- It is much better to read and write multiple words from/to memory rather than only one at a time. This we will also address.

Cache Associativity

- In our cache, a word can be put in any row in the cache.
- That means every row must be checked to see if the address matches.
- Our cache is called a **fully-associative cache**. These are expensive.
- If there is a function (in hardware) which maps an address to a particular row, then we only need one comparator, since there is only one row to look in. That is called a **direct-mapped cache**.
- In a direct-mapped cache we can use the least significant bits of the address as the function.
- Why should we not use the most significant bits as the function instead???

Direct Mapped Cache

- The purpose is to avoid so many comparators.
- In the C code the loop will disappear.
- If the number of rows in the cache is a power of two we can do as follows *instead of having a loop*:

```
i = address & (CACHE_ROWS - 1);
if (cache_array[i].valid
        && cache_array[i].address == address)
        data = cache_array[i].data;
// etc as in LD: above
```

N-way Associative Cache

- In direct mapped cache, we can be unlucky and having two frequently used items which are mapped to the same row in the cache which will result in many cache misses when they replace each other.
- A compromise is a so called **N-way associative cache**.
- Now we group the cache rows into sets, where each set has N rows.
- The number of sets, CACHE_SETS = CACHE_ROWS / N.
- An address is now mapped to a set and within its set, the address can be put in any of the N rows.
- N comparators are needed now, and typical values of N is 2, 4 or 8.

Accessing Data in an N-way Associative Cache

```
i = address & (CACHE_SETS - 1);
for (j = 0; j < N; j++) {
        if (cache_array[i][j].valid
                && cache_array[i][j].address == address) {
                data = cache_array[i][j].data;
                found = true;
                break;
        }
}
```

- In a fully associative cache, CACHE_SETS = 1.
- In a direct-mapped cache, N = 1.

Cache Block Size

- So far we have only transferred one word between the cache and the memory.
- It is more efficient if multiple words, say 8, 16, or 32 words are transferred at a time.
- power.cs.lth.se transfers 128 bytes at a time.
- Assume instead that our cache block size is 8 words.
- Then we can eg fetch eight instructions at a time.
- Since we store 8 consecutive words from memory in a cache row, we only need, of course, to know the address of the first word in the block.

Cache Block Number

- We can now view memory as an array, not of words, but of cache blocks.
- When the cache block size is eight words (or 2^3 words) we get the cache block number of a word by dividing the word number by eight.
- Alternatively we shift the word number, ie the address, to the right by three, ie, we throw the last three bits of the address away.
- The number of words in a cache block is called the BLOCK_SIZE.
- Actually the BLOCK_SIZE is the number of bytes but we simplify the presentation and only consider words.

Cache Block Number

```
block_num = address / BLOCK_SIZE;
i = block_num & (CACHE_SETS - 1);
for (j = 0; j < N; j++) {
    if (cache_array[i][j].valid
        && cache_array[i][j].block_num == block_num) {

        k = address & (BLOCK_SIZE - 1);
        data = cache_array[i][j].data[k];
        found = true;
        break;
    }
}
```

- The data in a row is now an array of BLOCK_SIZE words.