Assignment 2 - Simulation Assignment Report

Anagh Singh

January 24, 2016

Question 1 [CACTI]

CACTI: Answer the following questions for a 32nm, 4 bank, 2-way Set Associative, 64B line size, 16KB cache:

(a) How many total sets per bank are there?

(b) What is the default Vdd value?

(c) What are the components of the access time parameter in Cacti? What was the value in your case?

(d) Amongst the Data array and the Tag array, which consumed the most dynamic and leakage power? What are the values?

(e) How large (in mm^2) is the 16KB cache?

(f) Draw the architecture of a 4 bank, 2-way Set Associative, 64B line size, 16KB cache

Solution:

The CACTI simulator was proposed by researchers at the University of Utah. Most of the following answers have been based on the original paper published by the researchers at ISCA'07 outlining their simulator.

(a) For a given total cache size, we partition the cache into 2^N cache banks (N varies from 1 to 12) and for each N, we organize the banks in a grid with 2^M rows (M varies from 0 to N). Here, N=2, since there are 4 banks. Thus the grid is of size 2x2. Therefore the number of sets are 32.

(b) The default Vdd value can be found in the ***technology.c*** file. The default value is set to 0.6V as can be seen in the following screenshot.

```
//32 nm LOP
vdd[2] = 0.6;
Lphy[2] = 0.016;
Lelec[2] = 0.01172;//Lelec is the electrical gate-length.
t_ox[2] = 0.8e-3;//micron
v_th[2] = 0.0521;//V
c_ox[2] = 1.69e-14;//F/micron2
mobility_eff[2] =   751.71 * (1e-2 * 1e6 * 1e-2 * 1e6); //micron2 / Vs
Vdsat[2] = Lelec[2] * 0.42e+6 / (1e-2 * 1e6); //V/micron
c_g_ideal[2] = 2.7e-16;//F/micron
c_fringe[2] = 0.06e-15;
c_junc[2] = 1.0e-15;//F/micron2
I_on_n[2] = 843.4e-6;//A/micron
I_on_p[2] = I_on_n[2] / 2;
Rnchannelon[2] = vdd[2] / I_on_n[2];//ohm-micron
Rpchannelon[2] = vdd[2] / I_on_p[2];
I_off_n[2][0] = 8.41e-6;
I_off_n[2][10] = 9.52e-5;
I_off_n[2][20] = 9.52e-5;//MASTAR does not generate numbers for > 320 deg, so simply fixing
//leakage currents for temp > 320 equal to the 320 deg value.
I_off_n[2][30] = 9.52e-5;
I_off_n[2][40] = 9.52e-5;
I_off_n[2][50] = 9.52e-5;
I_off_n[2][60] = 9.52e-5;
I_off_n[2][70] = 9.52e-5;
I_off_n[2][80] = 9.52e-5;
I_off_n[2][90] = 9.52e-5;
I_off_n[2][100] = 9.52e-5;
for(i = 0; i <= 100; i += 10){
    I_off_p[2][i] = I_off_n[2][i];
}
```

Figure 1: Default Vdd value

(c) The components of the access time parameter in Cacti are Bank Access time, Avg. Network Delay, and Contention Cycles. The value in this case was 189 cycles.

```
Optimal number of banks - 4
Grid organization rows x columns - 2 x 2
Average access latency to a random bank
        (Bank Access time + Avg. Network Delay + Contention Cycles)- 189 cycles
Average dynamic energy/access (nJ) - 2.35767
Network frequency - 5 GHz
Cache dimension (mm x mm) - 16.3561 x 6.84994
```

Figure 2: Access time components

The bank access time can further be divided in the following manner, with the specific values being:

```
Time Components:

  Data side (with Output driver) (ns): 1.73033
        H-tree input delay (ns): 0.750742
        Decoder + wordline delay (ns): 0.0993786
        Bitline delay (ns): 0.0557184
        Sense Amplifier delay (ns): 0.03
        H-tree output delay (ns): 0.794494

  Tag side (with Output driver) (ns): 0.829167
        H-tree input delay (ns): 0.343505
        Decoder + wordline delay (ns): 0.0748184
        Bitline delay (ns): 0.0165114
        Sense Amplifier delay (ns): 0.03
        Comparator delay (ns): 0.0104131
        H-tree output delay (ns): 0.353919
```

Figure 3: Bank time components

(d) For the Data Array, the dynamic power consumed was 0.299086 nJ, while leakage power consumed was 9200.46 mW. For the Tag Array, the dynamic power consumed was 0.00443067 nJ, while leakage power consumed was 482.168 mW.

```
Power Components:

 Data array: Total dynamic read energy/access  (nJ): 0.299086
       Total leakage read/write power all banks at maximum frequency (mW): 9200.46
       Total energy in H-tree (that includes both address and data transfer) (nJ): 0.268346
       Decoder (nJ): 0.000418295
       Wordline (nJ): 0.000845348
       Bitline mux & associated drivers (nJ): 0.000385224
       Sense amp mux & associated drivers (nJ): 0.000466887
       Bitlines (nJ): 0.024646
       Sense amplifier energy (nJ): 0.00055296
       Sub-array output driver (nJ): 0.00296301

 Tag array:  Total dynamic read energy/access (nJ): 0.00443067
       Total leakage read/write power all banks at maximum frequency (mW): 482.168
       Total energy in H-tree  (nJ): 0.00234163
       Decoder (nJ): 7.90318e-05
       Wordline (nJ): 0.000190553
       Bitline mux & associated drivers (nJ): 9.31938e-05
       Sense amp mux & associated drivers (nJ): 5.42713e-05
       Bitlines (nJ): 0.00121808
       Sense amplifier energy (nJ): 0.00020736
       Sub-array output driver (nJ): 0.000138371
```

Figure 4: Power usage by Data and Tag Arrays

(e) The cache has an area of 112.030125584 mm^2 as can be seen from the following screenshot.

```
Optimal number of banks - 4
Grid organization rows x columns - 2 x 2
Average access latency to a random bank
       (Bank Access time + Avg. Network Delay + Contention Cycles)- 189 cycles
Average dynamic energy/access (nJ) - 2.35767
Network frequency - 5 GHz
Cache dimension (mm x mm) - 16.3561 x 6.84994
```

Figure 5: Area of Cache

(f) The architecture of a 4 bank, 2-way Set Associative, 64B line size, 16KB cache is:

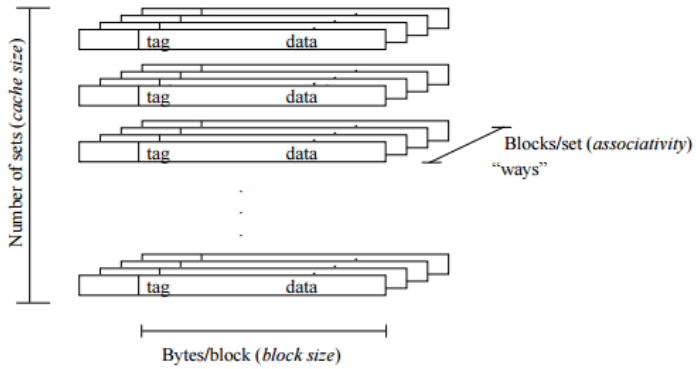

Figure 6: Architecture of Cache

The number of bytes/block is called the block size , which is 64B over here. The blocks/set is the associativity of the cache, which is 2 here, while the number of sets is 32.

Question 2 [ORION]

Estimate power and area of standard on-chip network routers. Present the area and power results of the following runs.

- The default values from SIM port.h file.
- A Router implemented on 45nm technology, running at 0.9V, built with HVT transistors.
- The default router with 6 input ports, 6 output ports, 128 bit flit width.

Solution:

The memory in a computer is divided up into distinct parts with the layers being as follows:

- Command Line Arguments and Variables
- Stack - Local variables
- Heap - Dynamic variables
- Data Segment - Global variables
- Text Segment - Static variables

The addresses of all these areas are in decreasing order since they are stored as described above. To check this, we print the addresses of all the variables which might be stored in those areas.

Question 3

Write a C program to calculate the factorial of an integer. Generate its assembly code for two different processors (e.g., x86 and MIPS). Explain how the function call mechanism works, including how the parameters are passed, how the stack/function frames are allocated, how the return address is retrieved, and how the stack and frame pointers are restored.

Solution:

The code to calculate the factorial of an integer is as follows:

The function call mechanism works as follows: The factorial function is called by name as shown in line 45. The value which has to be passed to the function is stored in RDI and copied into it by the register RAX. Post this, the function is called via name, and the arguments passed to it. Now whatever value was computed in the function call gets returned and stored again in the temporary register RAX. This is copied to the register RDX for storage, and the other arguments of the next instruction, which is the printf statement, are loaded into memory as shown in line 50. Upon proper completion, the function returns zero and is called in line 54. When the factorial function is called, new registers are declared, and values pushed onto them, for evaluation, as shown in line 67. The stack and frame pointers are kept track of in the Jump statements which are present in the code. After multiplication in line 82, the next instruction is to jump again, and check for the true condition. Upon checking, the factorial function is called again, in line 82. Upon completion of the loop, the leave and return instructions are issued which takes it to its last position on the stack, which is the print statement.

Question 4

Consider the Vector add program (Z[i] = X[i] + Y[i]) and compile it with and without optimization for your machine. See the differences in the assembly code generated with and without the optimizations. Study the loop unrolling gcc can do. Find the optimal unrolling factor for an array size of 16384.

Solution:

The code for vector addition can is as follows: Modern processors pipeline instructions. They predict which instruction is next and make optimisations based on assumptions of which order the instructions should be executed/ which branch they predicted.

A the end of a loop though, there are two possibilities- Either go back to the top, or continue on. The processor makes an educated guess on which is going to happen. If it gets it right, everything is good. If not, it has to flush the pipeline and stall for a bit while it prepares for taking the other branch. Unrolling a loop eliminates branches and the potential for those stalls, especially in cases where the odds are against a guess. Unrolling provides more benefits for shorter loops but ends up trashing performance if we are looping a large number of times. Usually, a smart compiler will take a decent guess about which loops to unroll but we can force it to a user defined value too. Loop unrolling does not work if the compiler can't predict the exact amount of iterations of the loop at compile time (or at least predict an upper bound, and then skip as many iterations as needed).

Also as can be seen in the attached screenshots, the **optimal unrolling factor for an array of size 16384 is 3**. Optimizations beyond level 3 have no further effect.