COSC 2440 – Computer Organization and Architecture – Fall 2019 - Kevin B Long

# 

# Homework #2

Due 11:59pm, Monday, 24 February, 2020

Multiple submissions accepted, last one graded.

100 points total.

Name: Xena Toumajian

PeopleSoft ID: 1662518

**1.9** (30 pts) We are running on a system with three classes of instructions: arithmetic instructions, load/store instructions, and branch instructions with. Our program executes the number of instructions shown per category in the first row below. Given the IC, the CPI, and processor speed, we can calculate an execution time.

What happens if we add more processors? Does having *p* processors mean our program will run in 1/*p*th the time? Not quite. We have to do extra work to coordinate things to make sure we don’t step on each other. To account for this, we will multiply the number of processors by 70%. So instead of dividing up IC across *p* processors, it’s as if we only had (*p*\*.7). Use this for math and load/store commands but not for branches – that number is constant.

1. (26 pts) <COD §1.7> Find the total execution time of this program for each scenario shown, and show the relative speedup relative to the single processor result. Use the following table for your answers. The original number of instructions on which you will base subsequent rows is given.

The speed of the processor is: 6.00 GHz. Don’t show more than 3 digits of significance anywhere.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **#p** | **# arith instr** | **CPI Arith** | **# L/S instr** | **CPI L/S** | **# branch instr** | **CPI Branch** | **cycles** | **exec time (sec)** | **speedup** |
| 1 | 5.00E9 | 1 | 2.00E9 | 9 | 1.00E9 | 8 | 5E9+ 2E9\*9+1E9\*8  =3.10E10 | 3.1e10/6e9  =5.17 | 1.00 |
| 2 | /.7 | 1 | /.7 | 9 |  | 8 | 3.29E10/2+1e9\*8  =2.44e10 | 4.08 | 5.17/4.08  =1.27 |
| 4 |  | 1 |  | 9 |  | 8 | =1.62E10 | 2.70 | 1.91 |
| 8 |  | 1 |  | 9 |  | 8 | =1.21E10 | 2.02 | 2.56 |
| 1000 |  | 1 |  | 9 |  | 8 | =8.03E9 | 1.34 | 3.86 |

1. (4 pts) [10] <COD §§1.6, 1.8> To what value would the load/store CPI for 1 processor have to change to match the execution time of the 4-processor system? In other words, if you want the 1-processor system to run your program in the same execution time as the 4-processor system, figure out what the Load/Store CPI needs to change to. In Excel, you can use a Goal Seek function to check your answer, but I want to see your equation.

Here’s something that might help. Let’s call math instructions category 1, load/store 2 and branch 3. The execution time can be calculated with:

If you replace the execution time for the single-processor system with that for the 4-processor row and solve for , you’ll have your answer.

T1=2.17+ CPI2\*.33=2.7

CPI2=(2.7-2.17)/.33=1.59

What is it? 1.59

**1.10** (24 pts) <§1.5> We have begun fabrication of a 300cm wafer, designed to produce big System-on-Chip dies, 20mm x 20mm.

1. What’s the size of the area of each wafer: 70685.83 mm2
2. How many whole and fractional SoC dies can you fit? 176.71457 dies

Round down to the nearest multiple of 10 to remove the partial dies at the circular edge:

170 dies

1. (5 pts) If you have a defect rate of 1 defect per cm2, how many defects would you on average find across the surface of a wafer?

7068

1. How many perfect 20 mm x 20 mm dies are you likely to produce per wafer?

7068/70685=.099

170\*.09= ~16

170-16=

154

1. If you could reduce the number of defects per wafer to say, 50, how many of your dies would you expect to be free of defects?

50/ 170 = .29 29%.

120

1. If yield is the % of good dies versus total dies you produce, what’s our yield?

Good 71% bad 29%

According to Adam Traidman from [Chip Estimate](http://www.chipestimate.com), the average die sizes being produced look like this these days:

**Smallest die size:**   
0.683 mm × 0.683 mm at the 90 nm technology node  
1.533 mm × 1.533 mm at the 65 nm technology node

**Average die size**   
7.020 mm × 7.020 mm at the 90 nm technology node  
2.130 mm × 2.130 mm at the 65 nm technology node

**Biggest die size:**   
20.253 mm × 20.253 mm at the 65 nm technology node

1. So our 20mm x 20mm die is a big one. What if we switched to using our 300mm wafers to make small dies, say 1mm x 1mm. Rounding down to the nearest 1000, how many dies would we be able to fabricate on our wafers?

70,000

1. If we build small dies, with that same 50 defects spread across your wafer, what would our yield be now?

99.93%

1. If you want an 80% yield for your large dies (20mm x 20mm), you need to reduce the defect rate. How many of your dies from (b) can be defective to keep you at 80%?

35

1. If that’s the number of defects across your wafer, about how many per cm2 is that?

.005 per 1 cm^2

1. If you think you’ll be able to sell 4M of these dies, how many wafers will you have to produce? Round up to the next 100.

4M/170 = 23600

1. Your product manager is told by finance that you have to generate $800M to pay for the factory retooling your share of R&D, and the desired profit, how much will you sell each die for?

$200

**1.11v2** (10 pts) From <https://www.spec.org>, search their CPU 2017 results for submissions from Supermicro.

Download the results as a csv file and open with your favorite spreadsheet program.

There are four classes of results – integer speed, integer rate, floating point speed and rate. In the CSV they’re called CINT2017, CINT2017rate, CFP2017, and CFP2017rate,

1. (8 pts) Of all of the vendor’s systems that reported a CFP2017 speed result, what was the highest ***peak*** result reported, and by which system (copy and paste the whole system name)?

Value: 189

System: A+ Server 2023US-TR4 (H11DSU-iN , AMD EPYC 7742)

1. (2 pts) Find these specific results back in the spec web site. How many programs in the benchmark suite did the vendor run to achieve this result?

10

**4.** (16 pts) Add each pair of numbers below, and indicate if the result shows overflow and/or carryout.

|  |  |  |  |
| --- | --- | --- | --- |
| Decimal | Binary |  |  |
| -121 | 10000111 |  |  |
| 82 | 01010010 | Overflow? | Carryout? |
| -39 | 11011001 | NO | NO |

|  |  |  |  |
| --- | --- | --- | --- |
| Decimal | Binary |  |  |
| 90 | 01011010 |  |  |
| -36 | 11011100 | Overflow? | Carryout? |
| 54 | 100110110 | NO | YES |

|  |  |  |  |
| --- | --- | --- | --- |
| Decimal | Binary |  |  |
| -34 | 11011110 |  |  |
| -109 | 10010011 | Overflow? | Carryout? |
| -143 | 101110001 | YES | YES |

|  |  |  |  |
| --- | --- | --- | --- |
| Decimal | Binary |  |  |
| 106 | 01101010 |  |  |
| 61 | 00111101 | Overflow? | Carryout? |
| 167 | 10100111 | YES | NO |

5. Based on the tutorial in lecture 9, complete the following table, showing how to multiply 13 x 22. Use 13 as the Multiplier. I’ve complete the first row of the table for you.

Assume all registers are 6 bits, as in the lecture.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Multiplier in Dec | Multiplicand in Dec |  |
|  |  | 13 | 22 |  |
| **Step** | **Action** | **Multiplier** | **Multiplicand** | **Product** |
| **0** | Load values | 001 101 | 000 000 010 110 | 000 000 000 000 |
|
| **1.1** | ☐LSB=0->do nothing, or | 001 101 | 000 000 010 110 | 000 000 010 110 |
| ☐LSB->add Md to Prod |
| **1.2** | Shift Multiplicand Left |  | 000 000 101 100 |  |
|
| **1.3** | Shift Multiplier Right | 000 110 |  |  |
|
| **2.1** | ☐LSB=0->do nothing, or | 000 110 | 000 000 101 100 | 000 000 010 110 |
| ☐LSB->add Md to Prod |
| **2.2** | Shift Multiplicand Left |  | 000 001 011 000 |  |
|
| **2.3** | Shift Multiplier Right | 000 011 |  |  |
|
| **3.1** | ☐LSB=0->do nothing, or | 000 011 | 000 001 011 000 | 000 001 101 110 |
| ☐LSB->add Md to Prod |
| **3.2** | Shift Multiplicand Left |  | 000 010 110 000 |  |
|
| **3.3** | Shift Multiplier Right | 000 001 |  | 000 010 110 000 |
|
| **4.1** | ☐LSB=0->do nothing, or | 000 001 | 000 010 110 000 | 000 100 011 110 |
| ☐LSB->add Md to Prod |
| **4.2** | Shift Multiplicand Left |  | 000 101 100 000 |  |
|
| **4.3** | Shift Multiplier Right | 000 000 |  |  |
|
| **5.1** | ☐LSB=0->do nothing, or | 000 000 | 000 101 100 000 | 000 100 011 110 |
| ☐LSB->add Md to Prod |
| **5.2** | Shift Multiplicand Left |  |  |  |
|
| **5.3** | Shift Multiplier Right |  |  |  |
|
| **6.1** | ⌧LSB=0->do nothing, or |  |  |  |
| ☐LSB->add Md to Prod |
| **6.2** | Shift Multiplicand Left |  |  |  |
|
| **6.3** | Shift Multiplier Right |  |  |  |
|

5. (20 pts) On slide 38 of Chapter 2, we saw the following MIPS assembly code:

Loop: sll $t1, $s3, 2

add $t1, $t1, $s6

lw $t0, 0($t1)

bne $t0, $s5, Exit

addi $s3, $s3, 1

j Loop

Exit:

Convert the first 3 instructions (sll, add, and lw) into HEX by converting them into 32-bit instructions and then to an 8-character hexadecimal string. The following site is helpful, but the syntax is pretty tricky: <https://www.eg.bucknell.edu/~csci320/mips_web/>

1. Instruction 1: sll $t1, $s3, 2

0000 0000 0001 0011 0100 1000 1000 0000

\_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_\_

Opcode rs rt rd shamt func

Hex: 0x00134880

1. Instruction 2: add $t1, $t1, $s6

0000 0001 0011 0110 0100 1000 0010 0000

\_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_\_

Opcode rs rt rd shamt func

Hex: 0x01364820

1. Instruction 3: lw $t0, 0($t1)

1000 91101 0010 1000 0000 0000 0000 0000

\_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_\_

Opcode rs rt rd shamt func

Hex: 0x8D280000

1. What about the command **0x14b40021**? Convert it to binary and then MIPS:

0001 0100 1011 0100 0000 0000 0010 0001

\_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_\_

Opcode rs rt rd shamt func

MIPS instruction: BNE $a1 $s4 0x0021