ECE 6913 Final Exam

Computing Systems Architecture

Fall 2021 NYU ECE

Time: December 16th 2021, 11:00 AM – 1:30 PM.

Solutions

1. Compute the effective CPI for RISC-V using *Figure P1* below and *Table P1* below. Average the instruction frequencies of `perlbench` and `sjeng` (shown in red box in Fig P1) to obtain the instruction mix

Instruction	Clock cycles
All ALU operations	1.0
Loads	3.5
Stores	2.8
Branches	
Taken	4.0
Not taken	2.0
Jumps	2.4

Table PI: CPI for Instruction classes

Program	Loads	Stores	Branches	Jumps	ALU operations
astar	28%	6%	18%	2%	46%
bzip	20%	7%	11%	1%	54%
gcc	17%	23%	20%	4%	36%
gobmk	21%	12%	14%	2%	50%
h264ref	33%	14%	5%	2%	45%
hmmer	28%	9%	17%	0%	46%
libquantum	16%	6%	29%	0%	48%
mcf	35%	11%	24%	1%	29%
omnetpp	23%	15%	17%	7%	31%
perlbench	25%	14%	15%	7%	39%
sjeng	19%	7%	15%	3%	56%
xalancbmk	30%	8%	27%	3%	31%

Figure P1: RISC-V dynamic instruction mix for the SPECint2006 programs.

Instruction category	perlbench	sjeng	Average of perlbench and sjeng
ALU operations	39%	56%	47.5%
Loads	25%	19%	22%
Stores	14%	7%	10.5%
Branches	15%	15%	15%
Jumps	7%	3%	5%

Table S1: RISC-V dynamic instruction mix average for perlbench and sjeng programs

For `perlbench` and `sjeng`, the average instruction frequencies are shown in Table S1 above

The effective CPI for programs in Figure P1 is computed by combining instruction category frequencies with their corresponding average CPI measurements.

Effective CPI =

Σ Instruction category frequency x Clock cycles for category

= (0.475) x (1.0) + (0.22)x(3.5) + (0.105)x(2.8) + (0.15)[(0.6)x(4.0) + (1-0.6)x(2.0)] + (0.5)x(2.4)

= 3.189

(The above assumes **50% of Branches are taken and 50% are not taken** with the CPI for not taken branches = 2.0)

Σ Instruction category frequency x Clock cycles for category

= (0.475) x (1.0) + (0.22)x(3.5) + (0.105)x(2.8) + (0.15)[(1)x(4.0) + (0)x(2.0)] + (0.5)x(2.4)

=3.339

(The above assumes **100% of Branches are taken and 0% are not taken** with the CPI for not taken branches = 2.0)

2. Assume that we make an enhancement to a computer that improves some mode of execution by a factor of 10. Enhanced mode is used 50% of the time, measured as a percentage of the execution time when the enhanced mode is in use. Recall that Amdahl's law depends on the fraction of the original, unenhanced execution time that could make use of enhanced mode. Thus, we cannot directly use this 50% measurement to compute speedup with Amdahl's law

(i) What is the speedup we have obtained from fast mode?

old execution time = 0.5 new + 0.5 × 10 new = 5.5 new

(ii) What percentage of the original execution time has been converted to fast mode?

In the original code, the unenhanced part is equal in time to the enhanced part

sped up by 10, therefore:

(1 – x) = x/10

10 – 10x = x

10 = 11x

10/11 = x = 0.91

3. Your company has just bought a new Intel Core i5 dual core processor, and you have been tasked with optimizing your software for this processor. You will run two applications on this dual core, but the resource requirements are not equal. The first application requires 80% of the resources, and the other only 20% of the resources. Assume that when you parallelize a portion of the program, the speedup for that portion is 2

(i) Given that 40% of the first application is parallelizable, how much speedup would you achieve with that application if run in isolation?

`1/(0.6 + 0.4/2) = 1.25`

(ii) Given that 99% of the second application is parallelizable, how much speedup would this application observe if run in isolation?

`1/(0.01 + 0.99/2) = 1.98`

(iii) Given that 40% of the first application is parallelizable, how much overall system speedup would you observe if you parallelized it?

`1/(0.2 + 0.8×0.6 + 0.8×0.4/2) = 1/(.2 + .48 + .16) = 1.19`

4. A. A. This question considers the basic, RISC-V, 5-stage pipeline. (In this problem, you may assume that there is full forwarding.)

(i) Explain how pipelining can improve the performance of a given instruction mix

Pipelining provides "pseudo-parallelism" – instructions are still issued sequentially, but their overall execution (i.e., from fetch to write back) overlaps

- Performance is improved via higher throughput

- An instruction (ideally) finishes every short Clock Cycle

(ii) Show how these instructions will flow through the pipeline:

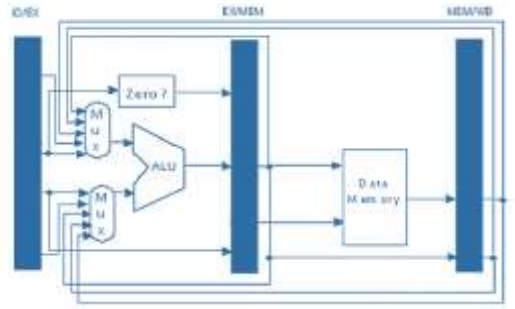
	1	2	3	4	5	6	7	8	9	10	11
lw x10, 0(x11)	F	D	E	M	W						
add x9, x11, x11		F	D	E	M	W					
sub x8, x10, x9			F	D	E	M	W				
lw x7, 0(x8)				F	D	E	M	W			
sw x7, 4(x8)					F	D	E	M	W		

(iii) Where might the sw instruction get its data from? Be very specific. (i.e. "from the lw instruction" is not a good answer!)

Data is available from the lw in the MEM / WB register

- There could be a feedback path from this register back to one of the inputs to data memory

- A control signal could then select the appropriate input to data memory to support this lw – sw input combination



B. This question considers the basic, RISC-V, 5-stage pipeline. (In this problem, you may assume that there is full forwarding.) Show how these instructions will flow through the pipeline below. For the instruction mix above, on what instruction results does the last add instruction depend on? predict that the `beq` instruction is not taken

	1	2	3	4	5	6	7	8	9	10	11	12
beq x1, x2, X	F	D	E									
lw x10, 0(x11)		F	D									
sub x14, x10, x10			F									
X: add x4, x1, x2				F	D	E	M	W				
lw x1, 0(x4)					F	D	E	M	W			
sub x1, x1, x1						F	D	D	E	M	W	
add x1, x1, x1							F	F	D	E	M	W

5. (i) A cache may be organized such that:

- In one case, there are more data elements per block and fewer blocks
- In another case, there are fewer elements per block but more blocks However, in both cases – i.e. larger blocks but fewer of them OR shorter blocks, but more of them – the cache's total capacity (amount of data storage) remains the same

What are the pros and cons of each organization? Support your answer with a short example assuming that the cache is direct mapped

If block size is larger:

- Con: There will be fewer blocks and hence a higher potential for conflict misses

- Pro: You may achieve better performance from spatial locality due to the larger block size

- Example: If you have a high degree of sequential data accesses, this makes more sense

If there are fewer elements per block and more blocks:

- Con: You may be more subject to compulsory misses due to the smaller block size

- Pro: You may see fewer conflict misses due to more unique mappings

- Example: If you have more random memory accesses, this makes more sense

(ii) Assume:

- A processor has a direct mapped cache
- Data words are 8 bits long (i.e., 1 byte)
- Data addresses are to the word
- A physical address is 20 bits long
- The tag is 11 bits
- Each block holds 16 bytes of data

How many blocks are in this cache?

index and offset

We can determine the number of bits of offset as the problem states that:

- Data is word addressable and words are 8 bits long

- Each block holds 16 bytes

As there are 8 bits / byte, each block holds 16 words, thus 4 bits of offset are needed.

This means that there are 5 bits left for the index. Thus, there are 25 or 32 blocks in the cache.

(iii) Consider a 16-way set-associative cache:

- Data words are 64 bits long

- Words are addressed to the half-word

- The cache holds 2 Mbytes of data

- Each block holds 16 data words

- Physical addresses are 64 bits long

How many bits of tag, index, and offset are needed to support references to this cache?

We can calculate the number of bits for the offset first:

- There are 16 data words per block which implies that at least 4 bits are needed

- Because data is addressable to the ½ word, an additional bit of offset is needed

- Thus, the offset is 5 bits

To calculate the index, we need to use the information given regarding the total capacity of the cache:

- 2 MB is equal to 2^{21} total bytes.

- We can use this information to determine the total number of blocks in the cache…

    - 2^{21} bytes x (1 block / 16 words) x (1 word / 64 bits) x (8 bits / 1 byte) = 2^{14} blocks

- Now, there are 16 (or 2^4) blocks / set

    - Therefore there are 2^{14} blocks x (1 set / 2^4 blocks) = 2^{10} or 1024 sets

- Thus, 10 bits of index are needed

Finally, the remaining bits form the tag:

- 64-5-10 = 49

- Thus, there are 49 bits of tag

To summarize: **Tag**: 49 bits; **Index**: 10 bits; **Offset**: 5 bits