Solution

Performance Measurement and Amdahl's Law

Problem 1.

The table below shows average instruction frequency for a particular benchmark when it is compiled for processor Super-Zeta.

Table 1: Super-Zeta's instruction frequency for benchmark

instruction	benchmark
load	8
store	44
add	22
sub	2
miscellanious ALU	10
cond branch	12
jump	1.0
call	0.5
return	0.5

} 34% ALU

Suppose we have made the following measurements about the CPI for instructions.

Table 2: Super-Zeta's CPI's for different instructions

instruction	clock cycles	CPI*Freq	% time	
all ALU instructions	1.0	0.34	16.8	
loads	5	0.4	19.8	
stores	2	0.88	43.6	
cond branches	3	0.36	17.8	
uncond branches	2	0.04	2.0	

CPI = 2.02

(1a) Calculate the average CPI for the benchmark benchmark. Use a blank column in Table 2 to put your results. See column labelled "CPI*Freq"

CPI = 2.02 A Given entry = clock cycles * frequency

(1b) Calculate the percentage of the benchmark execution time that is spent in each category shown in Table 2. Use a blank column in Table 2 to put your results. See the column "% time."

a given entry = $\frac{CPI * Freq}{2.02}$

Solution

(1c) Two teams have been assembled for design recommendations for the next generation of Super-Zeta. Team A suggests that new ALU functional units be added so that each ALU instruction can be finished in 0.25 clock cycles. Team B opposes this idea, and suggests an enhancement to speed up branch execution by a factor of 3. Which design enhancement promises greater speed-up?

$$\text{Speed up (ALU)} = \frac{1}{(1 - 0.168) + \frac{0.168}{4}}$$

$$= 1.144$$

$$\text{speed-up (branch)} = \frac{1}{(1 - 0.198) + \frac{0.198}{3}}$$

$$= 1.152$$

Branch enhancement gives a larger speed up.

Solution

Instruction Set Design

Problem 2.
What is the difference between caller-save and callee-save procedure invocation strategies? What does MIPS employ? Why?

Caller-save The caller function saves all the registers before invoking a procedure, and then restores them when the procedure returns.

Callee-save The callee function saves all the registers as soon as it is invoked. It then restores them before returning.

MIPS uses a hybrid approach where some registers are callee saved and some are caller saved. The goal is to minimize the number of registers saved and restored.

Problem 3.
What is the function of MIPS registers `$a0`, `$a1`, `$a2`, and `$a3`? What about `$v0` and `$v1`. Write a short piece of code that illustrates the use of the above registers and register `$ra`.

The $ax registers are used to pass arguments to a called procedure.

The $vy registers contain the values returned by a procedure.

```
        add  $a0, $t0, zero
        add  $a1, $t1, zero
        jalr sum

sum:    add  $v0, $a0, $a1
        jr   $ra
```

Solutions

Problem 4. What is a register-register architecture? How does a register-register architecture help the performance of a pipelined processor? Give one argument for and one against having a large register set.

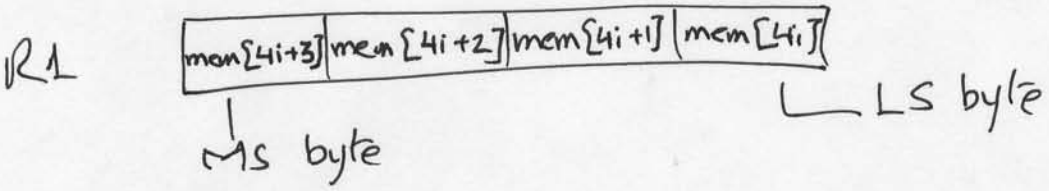
1) In a reg-reg architecture, operands of all ALU operations are in registers. Only load/store access memory.

2) Code for a reg-reg architecture allows re-ordering, which can help avoid some hazards in a pipelined processor.

3) FOR: Can keep more variables in register file.
AGAINST: Expensive.

Problem 5. A little-endian processor wants to load into register r1 a word that starts at address $4i$. Show the register contents in terms of the contents of memory locations $4i$, $4i+1$, $4i+2$, and $4i+3$.

The first byte from memory gets loaded into the littleend.



Problem 6. What do you understand by the *requirement of object alignment*?

Multibyte objects should be stored starting at addresses that are a multiple of their size.

Solutions

Pipelining

Examine the code sequence below.

`ld r3, 20(r4)`	5	5
`add r4, r3, r16`	6+2	6+1
`ld r5, 20(r6)`	9	8
`add r6, r5, r6`	10+2	9+1
`sub r5, r6, r2`	13+2	11
`sub r2, r1, r3`	16	12
`and r12, r2, r5`	17+2	13
`or r13, r6, r12`	20+2	14
`add r14, r2, r2`	23	15
`sw r15, 100(r2)`	24	16
`ld r3, 0(r4)`	25	17
`ld r2, 0(r5)`	26	18
`add r1, r2, r7`	27+2	19+1
`sub r12, r3, r5`	30	21

Problem 7a. Assuming that each instruction takes 5 clock cycles to execute, determine the total number of clock cycles to execute this program for the *unpipelined case.*

14 instn × 5 cc/instn = 70 cc

7b. Repeat 7(a) for the pipelined case where there is no forwarding between the pipeline stages. However the register file can be read and written in the same cycle.

There is a 2-cycle stall for every RAW hazard.
The last instruction finishes in 30th cc.

Solution

7c. Repeat 7(b) for the case of forwarding-enabled pipeline.

This time all stalls for ALU-to-ALU RAW hazards disappear. However there is still a stall left for each RAW hazard caused by a load instruction.

7d. The compiler has been charged with the task of producing an optimized version of the above code. Write the best re-ordered version the compiler can produce. Determine the total number of clock cycles to execute this re-ordered code.

1) Change
```
add r4 ...        ld r5, ...
ld  r5 ...   ⟹    add r4, ...
```

AND

2a) change
```
ld r3, 0(r4)      ld r2, 0(r5)
ld r2, 0(r5)  ⟹   ld r3, 0(r4)
```

OR

2b) swap the last two instructions.

other possibilities may exist.

Solutions

Problem 8a. In the context of a MIPS pipeline, *precisely define* a RAW data hazard (also called just "data hazard" in the lecture notes)?

An instruction tries to read an operand which has not yet been written by a prior instruction.

8b. Give a piece of MIPS code that will pose a data hazard?

```
add $1, $2, $3
sub $6, $1, $7
```

RAW hazard exists on R1.

8c. How can a compiler avoid a data hazard?

It can either re-order the code to insert a safe instruction between the instructions that pose RAW hazard

OR

It can insert sufficient number of "nops" between the instructions that pose RAW hazard.

Solutions

8d. It is also possible to have the hardware detect and avoid a data hazard. How does the hardware detect a data hazard?

When inst X is decoded, a RAW hazard exists if any source operand of X is also a destination operand of (X−1) or (X−2).

8e. Once the hardware detects a data hazard, it may choose *pipeline forwarding* to avoid the hazard. Explain forwarding by an example.

Instead of reading a source operand from register file, it is read from a pipeline register of the prior instruction.

8f. Give an example of a RAW data hazard that cannot be avoided by forwarding. What is the usual solution?

```
ld  r1, 0(r2)
add r3, r1, r7
```

RAW hazard on r1. Pipeline forwarding will eliminate one stall but one stall will remain.

Usual solution

Usual solution is to re-order the instructions. If that is not possible, either insert a nop or stall the pipeline.

Solutions

Caches

Problem 9. You are given 16 kilobytes of SRAM to construct a cache that has 64 byte blocks. Assume that you are building a 4-way associative cache. Assume that the CPU address is 32 bits long.

9a. Determine the number of frames in the cache.

$$\#\text{ of frames} = \frac{\text{cache size}}{\text{cache frame size}} = \frac{16 \times 1024}{64}$$

$$= 256$$

9b. Determine the size (in bits) of the block offset. What is the purpose of block offset bits?

$$\text{offset size} = \log_2(64) = 6$$

Used to locate the byte within a block.

9c. Determine the size (in bits) of the index. What is the purpose of index bits?

$$\text{index size} = \log_2(\#\text{ of sets})$$

$$= \log_2\left(\frac{\#\text{ of frames}}{\text{associativity}}\right)$$

$$= \log_2\left(\frac{256}{4}\right) = \log_2(64)$$

$$= 6$$

Used to identify that set in a cache that might have the block.

Solutions

9d. Determine the size (in bits) of the tag. What is the purpose of tag bits?

tag size = total address size − index − offset

= 32 − 6 − 6 = 20 bits

tag of the CPU address is matched with all tags in a given set to see if the required block is in cache.

9e. Determine the total size (in bits) of the entire cache.

# of bits per frame = 64 × 8 + 20 + 1

= 533

total cache size = 533 × 256

= 136,448 bits

9f. Draw a clear fully labelled block diagram to show how a CPU address is used to determine if the addressed block is in this 4-way associative cache. Indicate what address bits are used for each task.