Stanford University 10 November 2005

Computer Architecture Comprehensive Exam Solutions

Exam Instructions
Answer each of the questions included in the exam. Write all of your answers directly on the examination paper, including any work that you wish to be considered for partial credit. The examination is open-book, and you may make use of the text, handouts, your own course notes, and a calculator. You may use a computer of any kind but no network.

On equations: Wherever possible, make sure to include the equation, the equation rewritten with the numerical values, and the final solution. Partial credit will be weighted appropriately for each component of the problem, and providing more information improves the likelihood that partial credit can be awarded.

On writing code: Unless otherwise stated, you are free to use any of the assembly instructions listed in the Appendix at the back of the book, including pseudoinstructions. You do not need to optimize your MIPS code unless specifically instructed to do so.

On time: You will have one hour to complete this exam. Budget your time and try to leave some time at the end to go over your work. The point weightings correspond roughly to the time each problem is expected to take.

THE STANFORD UNIVERSITY HONOR CODE

The Honor Code is an undertaking of the students, individually and collectively:

(1) that they will not give or receive aid in examinations; that they will not give or receive unpermitted aid in class work, in the preparation of reports, or in any other work that is to be used by the instructor as the basis of grading;
(2) that they will do their share and take an active part in seeing to it that others as well as themselves uphold the spirit and letter of the Honor Code.

I acknowledge and accept the Honor Code.

Magic Number ____

		Score	Grader
1. Short Answer	(15)	______	______
2. ISA	(15)	______	______
3. Pipelining	(15)	______	______
4. Cache	(15)	______	______

Total (60) ______

Problem 1: Short Answer (15 points)
Please provide short, concise answers.

(a) [3 points] Can a direct mapped cache sometimes have a higher hit rate than a fully associative cache with an LRU replacement policy (on the same reference pattern and with the same cache size)? If so, give an example. If not, explain why not?

Answer:

Imagine a 4 word cache with an access pattern of 0, 1, 2, 3, 4, 0, 1, 2, 3, 4. The directed mapped cache will have a 30% hit rate while the LRU fully associative cache will have a 0% hit rate.

(b) [3 points] Give two ways virtual memory address translation is useful even if the total size of virtual memory (summed over all programs) is guaranteed to be smaller than physical memory.

Answer:

Some examples are:

1. *isolation: protect processes fro each others' memory*
2. *relocation: allow any program to run anywhere in physical memory*

(c) [3 points] How does a data cache take advantage of spatial locality?

Answer:

When a word is loaded from main memory, adjacent words are loaded into the cache line. Spatial locality says that these adjacent bytes are likely to be used. A common example iterating through elements in an array.

(d) [6 points] What is the advantage of using a virtually-indexed physically-tagged L1 cache (as opposed to physically-indexed and physically-tagged)? What constraints does this design put on the size of the L1 cache

Answer:

[3 points] The cache index can be derived from the virtual address without translation. This allows the cache line to be looked up in parallel with the TLB access that will provide the physical cache tag. This helps keep the cache hit time low in the common case where of a TLB hit

[3 points] For this to work, the cache index cannot be affected by the virtual to physical address translation. This would happen if the any of the bits from the cache index came from the virtual page number of the virtual address instead of the page offset portion of the address..

Problem 2: Instruction Set Architecture (15 points)

At TGIF, you hear a computer architecture graduate student propose that MIPS would have been better if only it had allowed arithmetic and logical instructions to have a register-memory addressing mode. This new mode would allow the replacement of sequences like:

```
lw   $1,  0($n)
add  $2,  $2,  $1
```

with:

```
add  $2,  0($n)
```

The student waves his hands (one holding bread and one holding cheese) saying this can be accomplished with only a 5% increase in the clock cycle and no increase in CPI.

You are fascinated by the possibilities of this breakthrough. However, before you drop out to create a startup, you decide to use SPECint2000 to evaluate the performance because it contains your favorite three applications (gcc, gzip, and of course perl). Since you love computer architecture so much, you have an EE282 book that has the instruction set mix for SPECint2000 with you (table at right).

Instruction	Average
load	26%
store	10%
add	19%
sub	3%
mul	0%
compare	5%
load imm	2%
cond branch	12%
cond move	1%
jump	1%
call	1%
return	1%
shift	2%
and	4%
or	9%
xor	3%
other logical	0%

(a) [8 points] What percentage of loads must be eliminated for the machine with the new instruction to have at least the same performance?

Answer:

NOTE: load imm (load immediate) is not a kind of lw instruction

Because the new design has a clock cycle time equal to 1.05 times the original clock cycle time, the new design must execute fewer instruction to achieve the same execution time. For the original design, we have:

$$CPU\ Time_{old} = CPI_{old} * CC_{old} * IC_{old}$$

For the new design, the equation is:

$$CPU\ Time_{new} = CPI_{new} * CC_{new} * IC_{new}$$
$$= CPI_{old} * (1.05*CC_{old}) * (IC_{old} - R)$$

where the CPI of the new design is the same as the original (as stated in the question), the new clock cycle is 5% longer, and the new design executes R fewer instructions than the original design. To find out how many loads must be removed to match the performance of the original design set the above two equations equal and solve for R:

$$CPI_{old} * CC_{old} * IC_{old} = CPI_{old} * (1.05*CC_{old}) * (IC_{old} - R)$$
$$R = 0.0476\ IC_{old}$$

Thus, the instruction count must decrease by 4.76% overall to achieve the same performance, and this 4.76% is comprised entirely of loads. The data shows that 26% of the gcc instruction mix is loads, so (4.76%/26%) = 18.3% of the loads must be replaced by the new register-memory instruction format for the performance of the old and new designs to be the same. If more than 18.3% of the loads can be replaced, then performance of the new design is better.

(b) [4 points] Give an example of a code sequence where the compiler could not perform this replacement even though it matches the general pattern?

Answer:

One example is when the $1 variable is reused in another instruction below. Another is when the initial sequence uses the same register as the load destination and the add destination instead of distinct registers.

(c) [3 points] Considering the usual 5 stage MIPS pipeline, why might this new instruction be problematic to implement with no change in CPI?

Answer:

The result from the load will be available after the 4^{th} stage (M) however it needs to be an input to the 3^{rd} stage (X).

Problem 3: Pipelining (15 points)

Consider the following code:

```
Loop:lw   $1,  0($2)
     addi $1,  $1,  1
     sw   $1,  0($2)
     addi $2,  $2,  4
     sub  $4,  $3,  $2
     bne  $4,  $0, Loop
```

Assume that the initial value or R3 is R2 + 396

This code snippet will be executed on a MIPS pipelined processor with a 5-stage pipeline. Branches are resolved in the decode stage and *do not* have delay slots. All memory accesses take 1 clock cycle.

In the following three parts, you will be filling out pipeline diagrams for the above code sequence. Please use acronyms F, D, X, M and W for the 5 pipeline stages. For all cases of forwarding, use arrows to connect the source and destination stages. Simulate at most 7 instructions, making one pass through the loop and performing the first instruction a second time.

(a) [5 points] Fill in the pipeline diagram below for the execution of the above code sequence *without* any forwarding or bypassing hardware but assuming a register read and a write in the same clock cycle "forwards" through the register file.

Answer:

	Cycle																
Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
lw $1,0($2)	*F*	*D*	*X*	*M*	*W*												
addi $1,$1,1		*F*	*D*	*D*	*D*	*X*	*M*	*W*									
sw $1,0($2)			*F*	*F*	*F*	*D*	*D*	*D*	*X*	*M*	*W*						
addi $2,$2,4						*F*	*D*	*D*	*D*	*X*	*M*	*W*					
sub $4,$3,$2							*F*	*D*	*D*	*D*	*D*	*D*	*X*	*M*	*W*		
bne $4,$0,…								*F*	*F*	*F*	*F*	*F*	*D*	*D*	*D*		
lw $1,0($2)																*F*	*D*

(b) [5 points] Fill in the pipeline diagram below for the execution of the above code sequence *with* traditional pipeline forwarding:

Answer:

	Cycle																
Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`lw $1,0($2)`	*F*	*D*	*X*	*M*	*W*												
`addi $1,$1,1`		*F*	*D*	*D*	*X*	*M*	*W*										
`sw $1,0($2)`			*F*	*F*	*D*	*X*	*M*	*W*									
`addi $2,$2,4`					*F*	*D*	*X*	*M*	*W*								
`sub $4,$3,$2`						*F*	*D*	*X*	*M*	*W*							
`bne $4,$0,…`							*F*	*D*	*D*								
`lw $1,0($2)`										*F*	*D*	*X*	*M*	*W*			

(c) [5 points] <u>Aggressively</u> rearrange the order of the instructions (data dependencies have to be preserved) so that the number of instructions/cycles needed to execute the code snippet is minimized. Fill in the following table with the rearranged instruction sequence assuming traditional pipeline forwarding like part (b):

Answer:

	Cycle																
Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`lw $1,0($2)`	*F*	*D*	*X*	*M*	*W*												
`addi $2,$2,4`		*F*	*D*	*X*	*M*	*W*											
`addi $1,$1,1`			*F*	*D*	*X*	*M*	*W*										
`sw $1,-4($2)`				*F*	*D*	*X*	*M*	*W*									
`sub $4,$3,$2`					*F*	*D*	*X*	*M*	*W*								
`bne $4,$0,…`						*F*	*D*										
`lw $1,0($2)`								*F*	*D*	*X*	*M*	*W*					

There was no way to aggressively rearrange the order without changing the offset of the sw instruction from 0 to -4 as shown in the above solution. Full credit was also given for saying it was impossible to reorder without changing the instructions.

Problem 4: Cache Performance (15 points)

The following problem concerns basic cache lookups.

- The memory is byte addressable.
- Memory accesses are to **1-byte words** (not 4-byte words).
- Physical addresses are 13 bits wide.
- The cache is 4-way set associative, with a 4-byte block size and 32 total lines.

In the following tables, **all numbers are given in hexadecimal**. The *Index* column contains the set index for each set of 4 lines. The *Tag* columns contain the tag value for each line. The *V* column contains the valid bit for each line. The *Bytes 0–3* columns contain the data for each line, numbered left-to-right starting with byte 0 on the left.

The contents of the cache are as follows:

4-way Set Associative Cache																								
Index	Tag	V	Bytes 0–3				Tag	V	Bytes 0–3				Tag	V	Bytes 0–3				Tag	V	Bytes 0–3			
0	84	1	ED	32	0A	A2	9E	0	BF	80	1D	FC	10	0	EF	9	86	2A	E8	0	25	44	6F	1A
1	18	1	03	3E	CD	38	E4	0	16	7B	ED	5A	02	0	8E	4C	DF	18	E4	1	FB	B7	12	02
2	84	0	54	9E	1E	FA	84	1	DC	81	B2	14	48	0	B6	1F	7B	44	89	1	10	F5	B8	2E
3	92	0	2F	7E	3D	A8	9F	0	27	95	A4	74	57	1	07	11	FF	D8	93	1	C7	B7	AF	C2
4	84	1	32	21	1C	2C	FA	1	22	C2	DC	34	73	0	BA	DD	37	D8	28	1	E7	A2	39	BA
5	A7	1	A9	76	2B	EE	73	0	BC	91	D5	92	28	1	80	BA	9B	F6	6B	0	48	16	81	0A
6	8B	1	5D	4D	F7	DA	29	1	69	C2	8C	74	B5	1	A8	CE	7F	DA	BF	0	FA	93	EB	48
7	84	1	04	2A	32	6A	96	0	B1	86	56	0E	CC	0	96	30	47	F2	91	1	F8	1D	42	30

(a) [3 points] The box below shows the format of a physical address. Indicate (by labeling the diagram) the fields that would be used to determine the following:

O The block offset within the cache line
I The cache index
T The cache tag

12	11	10	9	8	7	6	5	4	3	2	1	0
T	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*I*	*I*	*I*	*O*	*O*

(b) [5 points] For the given physical address, indicate the cache entry accessed and the cache byte value returned in hex. Indicate whether a cache miss occurs. If there is a cache miss, enter “-” for “Cache Byte returned”.

Physical address: 0x0D74

Physical address format (one bit per box)

12	11	10	9	8	7	6	5	4	3	2	1	0
0	*1*	*1*	*0*	*1*	*0*	*1*	*1*	*1*	*0*	*1*	*0*	*0*

Physical memory reference:

Parameter	Value
Cache Offset (CO)	*0x00*
Cache Index (CI)	*0x05*
Cache Tag (CT)	*0x6B*
Cache Hit? (Y/N)	*N*
Cache Byte returned	*0x-*

Physical address: 0x0AEE

Physical address format (one bit per box)

12	11	10	9	8	7	6	5	4	3	2	1	0
0	*1*	*0*	*1*	*0*	*1*	*1*	*1*	*0*	*1*	*1*	*1*	*0*

Physical memory reference:

Parameter	Value
Cache Offset (CO)	*0x02*
Cache Index (CI)	*0x03*
Cache Tag (CT)	*0x57*
Cache Hit? (Y/N)	*Y*
Cache Byte returned	*0xFF*

(c) [4 points] For the given contents of the cache, list all of the hex physical memory addresses that will hit in Set 7. To save space, you should express contiguous addresses as a range. For example, you would write the four addresses 0x1314, 0x1315, 0x1316, 0x1317 as 0x1314-0x1317.

Answer: *0x109C – 0x109F, 0x123C – 0x123F*

The following templates are provided as scratch space:

12	11	10	9	8	7	6	5	4	3	2	1	0
1	*0*	*0*	*0*	*0*	*1*	*0*	*0*	*1*	*1*	*1*	-	-

12	11	10	9	8	7	6	5	4	3	2	1	0
1	*0*	*0*	*1*	*0*	*0*	*0*	*1*	*1*	*1*	*1*	-	-

12	11	10	9	8	7	6	5	4	3	2	1	0

(d) [3 points] For the given contents of the cache, what is the probability (expressed as a percentage) of a cache hit when the physical memory address ranges between 0x1080 - 0x109F. Assume that all addresses are equally likely to be referenced.

Probability = *50*%

The following templates are provided as scratch space:

12	11	10	9	8	7	6	5	4	3	2	1	0
1	*0*	*0*	*0*	*0*	*1*	*0*	*0*	*0*	*0*	*0*	*0*	*0*

12	11	10	9	8	7	6	5	4	3	2	1	0
1	*0*	*0*	*0*	*0*	*1*	*0*	*0*	*1*	*1*	*1*	*1*	*1*