Department of Information Technology
Lund University

Computer Architecture, EIT090 exam 15-12-2004

I. Problem 1 (15 points)

Briefly (1-2 sentences) describe the following items/concepts concerning computer architecture:

1. *[40, 257]* Memory bandwidth;
   Memory bandwidth is defined as the maximum rate at which information can be read from and written to the memory.
2. *[278, 323]* VLIW;
   Multiple-issue processors come in two flavours: superscalar and very long instruction word. Such VLIWs issue a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet.
3. *[242, 292]* Score-boarding;
   Score-boarding is a technique for allowing instructions to execute out of order when there are sufficient resources and no data dependencies. When a next instruction is stalled, other instructions can be issued and executed if they do not depend on any active or stalled instruction.
4. *[637/640, 534]* Shared-memory architecture;
   SMA is a class of MIMD machines. In centralized SMA a small amount of processors are connected to a single memory (also called uniform memory access); in distributed SMA many processors have a shared address space to physically separate memories (also called non-uniform memory access, as access times are dependent on the actually required path from processor to memory).
5. *[29-32, 17]* Amdahl's Law;
   "The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used." or
   speed-up = task performance using enhancement / original task performance.
6. *[443, 432]* Virtual memory miss;
   A Virtual Memory is a 2-tier addressing scheme, where a contiguous address space is partitioned into virtual blocks that are individually mapped onto the larger physical world, possible loosing contiguity. When the requested information is not present in the first memory layer, a miss will be generated as usual.
7. *[657, 467]* Bus snooping;
   Every cache that has a copy of the data from a block of physical memory also has a copy of the sharing status of the block, and no centralized state is kept. The caches are usually on a shared-memory bus, and all cache controllers monitor or snoop on the bus to determine whether or not they have a copy of a block that is requested on the bus.
8. *[655-658, 466]* Cache coherence;
   Cache coherence becomes a problem when multiple processors can have different values for the same memory location.

9. *[195, 287]* Out-of-order completion;
   This occurs when an instruction issued early may complete after an instruction issued later and gives rise to imprecise exceptions or additional matters to bring instructions into order.

10. *[A-36, A-30]* Inexact exception.
    This is one of the five exceptions stated in the IEEE FP number representation standard. The inexact exception occurs when either the result of an operation mut be rounded or when it overflows. It happens exceptionally often.

II. Problem 2 (20 points)

[Q5.14, Q5.12] Smith and Goodmann found that for a given small size, a direct-mapped instruction cache consistently outperformed a fully associative instruction cache using LRU replacement.

1. *[390, 423]* Shortly describe the three C's model;
   The three C's model sorts the causes for cache misses into three categories:
   - Compulsory – The very first access can never be in cache and is therefore bound to generate a miss;
   - Capacity – If the cache cannot contain all the blocks needed for a program, capacity misses may occur;
   - Conflict – If the block placement strategy is set-associative or direct-mapped, conflict misses will occur because a block can be discarded and later retrieved if too many blocks map to its set.

2. *[5.6, 5.5]* Explain how this would be possible (*HINT*: you cannot explain this with the three C's model because it "ignores" replacement policy);
   A small direct-mapped cache can potentially outperform a fully associative cache if the system is executing a loop that does not fit entirely in cache. To see how this happens, consider a loop that accesses three unique addresses: A, B and C and then repeats the sequence by looping back to A. To reference stream for such a program would look like this: ABCABCABC…, where each letter corresponds to an address in the reference stream. To simplify the discussion, we assume that our caches have only two blocks of storage space with addresses A and C mapping into the first block and address B mapping into the second lock. For a fully associative version of the 2-block cache, our reference team always misses the cache if the replacement policy is LRU. Such behaviour does not occur in a direct-mapped cache using LRU replacement because each address maps into the specific location of the cache. In the direct-mapped version, we only miss on an access to A or C but we always hit on an access to B (ignoring initially compulsory misses).

3. *[393, 426]* Explain where replacement policy fits into the three C's model, and explain why this means that misses caused by a replacement policy are "ignored" _ or more precisely cannot in general be definitively classified _ by the three C's model;
   The three C's give insight into the cause of misses, but this simple model has its limits; it gives you insight into average behaviour but may not explain an individual miss. For example, changing cache size changes conflict misses as well as capacity misses, since a larger cache spreads out references to more blocks. Thus, a miss might move from a capacity miss to a conflict miss as cache size changes. Note that the three C's also ignore replacement policy, since it is difficult to model and since,

in general, it is less significant. In specific circumstances the replacement policy can actually lead to anomalous behaviour, such as poorer miss rates for larger associativity, which is contradictory to the three C's model.

4. *[,429]* Are there any replacement policies for the fully associative cache that would outperform the direct-mapped cache? (Ignore the policy of "do what a direct-mapped cache would do");

In principle the miss rate will be reduced by higher associativity, though the effect quickly saturates with increasing associativity. Then again, the improvement comes at the cost of increased hit time.

5. *[379, 401]* What is a write-through cache? Is it faster/slower than a write-back cache with respect to the time it takes for writing.

A cache write is called write-through when information is passed both to the block in the cache and to the block in the lower-level memory; when information is only written to the block, it is called write-back. Write-back is the fastest of the two as it occurs at the speed of the cache memory, while multiple writes within a block require only one write to the lower-level memory.

III. Problem 3 (25 points)

[Q3.1, Q2.4] Use the following code fragment whereby the initial value of R3 is R2+396:

```
Loop:   LW      R1, 0(R2)
        ADDI    R1, R1, #1
        SW      0(R2), R1
        ADDI    R2, R2, #4
        SUB     R4, R3, R2
        BNEZ    R4, Loop
```

Throughout this exercise use the MIPS integer pipeline and assume all memory accesses are cache hits.

- Show the timing of this instruction sequence for the MIPS pipeline *without* any forwarding or bypassing hardware but assuming a register read and a write in the same clock cycle "forwards" through the register file. Use a pipeline-timing chart. Assume that the branch is handled by flushing the pipeline. If all memory references hit in the cache, how many cycles does this loop take to execute?

Instruction		*Clock Cycle*																					
		1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*
lw	*r1,0(r2)*	*F*	*D*	*X*	*M*	*W*																	
addi	*r1,r1,#1*		*F*	*s*	*s*	*D*	*X*	*M*	*W*														
sw	*o(r2),r1*					*F*	*S*	*s*	*D*	*X*	*M*	*W*											
addi	*r2,r2,#4*								*F*	*D*	*X*	*M*	*W*										
sub	*r4,r3,r2*									*F*	*s*	*s*	*D*	*X*	*M*	*W*							
bnz	*r4,loop*												*F*	*s*	*s*	*D*	*X*	*M*	*W*				
lw	*r1,o(r2)*															*F*	*s*	*s*	*F*	*D*	*X*	*M*	*W*

There are several cycles lost to stalls.

- *cycles 3-4: add1 stalls ID to wait for lw to write back r1.*
- *cycles 6-7: sw stalls ID to wait for addi to write back r1.*
- *cycles 10-11: sub stalls ID to wait for addi to write back r2.*
- *cycles 13-14: bnz stalls ID to wait for sub to write back r4.*

- *cycles 16-17: bnz computes the next PC in MEM implying that the lw cannot be fetched until after cycle 17 (note the fetch in cycle 15 is also wasted)*

In this figure we have assumed the version of MIPS, which resolves branches in MEM. The second iteration begins 17 clocks after the first iteration and the last iteration takes 18 cycles to complete. This implies that iteration i (where iterations are numbered from 0 to 98) begins on clock cycle 1+17xi. As the loop executes 99 times the loop executes in a total of (98x17)+18=1684 clocks.

- Show the timing of this instruction sequence in the MIPS pipeline with normal forwarding and bypassing hardware. Use a pipeline timing chart. Assume that the branch is handled by predicting it as not taken. If all memory references hit in the cache, how many cycles does this loop take to execute?

Instruction		*Clock Cycle*														
		1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
lw	*r1,0(r2)*	*F*	*D*	*X*	*M*	*W*										
addi	*r1,r1,#1*		*F*	*s*	*D*	*X*	*M*	*W*								
sw	*o(r2),r1*				*F*	*D*	*X*	*M*	*W*							
addi	*r2,r2,#4*					*F*	*D*	*X*	*M*	*W*						
sub	*r4,r3,r2*						*F*	*s*	*D*	*X*	*M*	*W*				
bnz	*r4,loop*								*F*	*s*	*D*	*X*	*M*	*W*		
lw	*r1,o(r2)*										*F*	*s*	*D*	*X*	*M*	*W*

The second iteration begins 10 clocks after the first iteration and the last iteration takes 13 cycles to complete. This implies that iteration i (where iterations are numbered from 0 to 98) begins on clock cycle 1+10xi. As the loop executes 99 times the loop executes in a total of (98x10)+13=993 clocks.

- Assuming the MIPS pipeline with a single-cycle delayed branch and normal forwarding and bypassing hardware, schedule the instructions in the loop including the branch-delay slot. You may reorder instructions and modify the individual instructions operands, but do not undertake other loop transformations that change the number and opcode of the instructions in the loop. Show a pipeline timing chart and compute the number of cycles needed to execute the entire loop.

Instruction		*Clock Cycle*											
		1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
lw	*r1,0(r2)*	*F*	*D*	*X*	*M*	*W*							
addi	*r2,r2,#4*		*F*	*D*	*X*	*M*	*W*						
addi	*r1,r1,#1*			*F*	*D*	*X*	*M*	*W*					
sub	*r4,r3,r2*				*F*	*D*	*X*	*M*	*W*				
sw	*o(r2),r1*					*F*	*D*	*X*	*M*	*W*			
bnz	*r4,loop*						*F*	*D*	*X*	*M*	*W*		
lw	*r1,o(r2)*							*F*	*S*	*D*	*X*	*M*	*W*

The first half iteration takes 8 clock cycles. At the 6th clock cycle we start already with the first half of the second iteration while continuing with the remainder of the first (this overlap is shown in the figure). The third iteration begins 9 clocks after the second iteration and the last iteration takes 9 cycles to complete. This implies that iteration i (where iterations are numbered from 0 to 98) begins on clock cycle 8+9xi. As the loop executes 99 times the loop executes in a total of 8+(97x9)+9=890 clocks.

IV. Problem 4 (25 points)

[Q2.3, Q3.18] Your task is to compare the memory efficiency of four different styles of instruction set architectures. The architecture styles are:

- *Accumulator* – All operations occur between a single register and a memory location;
- *Memory-memory* _ All instruction addresses reference only memory locations;
- *Stack* _ All operations occur on top of the stack. Push and pop are the only instructions that access memory; all others remove their operands from the stack and replace them with the result. The implementation uses a hardwired stack for only the top two stack entries; which keeps the processor circuit very small and low cost. Additional stack positions are kept in memory locations, and accesses to these stack positions require memory references;
- *Load-store* – All operations occur in registers, and register-to register instructions have three register names per instruction.

To measure memory efficiency, make the following assumptions about all 4 instruction sets:

- All instructions are an integral number of bytes in length;
- The opcode is always 1 byte (8 bits);
- Memory accesses use direct, or absolute addressing;
- The variables A, B, C, and D are initially in memory.

Invent your own assembly language mnemonics and for each architecture write the best equivalent assembly language code for the high-level language code sequence:

A = B + C;
B = A + C;
D = A – B;

Stack	*Accumulator*	*Store/Load*	*Memory/memory*
Push B	*Load B*	*Load R2, B*	*Add M0,M1,M2*
Push C	*Add C*	*Load R3, C*	*Add M1,M0,M2*
Add	*Store A*	*Add R1,R2,R3*	*Sub M3,M0,M1*
Push Top	*Add C*	*Add R2,R1,R3*	
Push C	*Store B*	*Sub R4,R1,R2*	
Add	*Sub A*	*Store R4, D*	
Push Top	*Store D*		
Pop B			
Sub			
Pop D			

Label each instance in your assembly codes where a value is loaded from memory after having been loaded once. Also label each instance in your code where the result of one instruction is passed to another instruction as an operand, and further classify these events as involving storage within the processor or storage in memory.

Stack	*Accumulator*	*Store/Load*	*Memory/memory*
Push B	*Load B*	*Load R2, B*	*Add M0,M1,M2 /3*
Push C	*Add C*	*Load R3, C*	*Add M1,M0,M2 /3*
Add /2	*Store A /3*	*Add R1,R2,R3 /2*	*Sub M3,M0,M1*
Push Top	*Add C /1*	*Add R2,R1,R3 /2*	
Push C /1	*Store B /3*	*Sub R4,R1,R2 /2*	
Add /2	*Sub A /2*	*Store R4, D*	

Push Top	*Store D*		
Pop B			
Sub /2			
Pop D			

with Label /1 if a value is re-loaded from memory
Label /2 if the result from one instruction is consumed through local storage
Label /3 if the result from one instruction is consumed through local memory

Assume the given code sequence is from a small, embedded computer application, such as a microwave oven controller, that uses 16-bit memory addresses and data operands. If a load-store architecture is used, assume that it has 16 general-purpose registers. For each architecture, answer the following questions:

1. How many instruction bytes are fetched?
   Stack requires 1 byte for the opcode for stack operations and additional 2 byte memory addresses, giving 5 times 1 plus 5 times 3 bytes is 20 bytes in total.
   Accumulator requires 1 byte for the opcode and 2 byte memory addresses, giving 7 times 3 bytes is 21 bytes in total.
   Store/Load requires 1 byte for the opcode, a half byte for the subject register and either 1 byte for the other registers or 2 bytes memory address, giving after rounding on integer bytes of the instruction length the total of 3 times 3 plus 3 times 4 is 21 bytes.
   Memory/memory requires 1 byte for the opcode and 6 bytes for the memory addresses, giving 3 times 7 bytes is 21 bytes in total.

2. How many bytes of data are transferred from/to memory?
   In Stack we read/write B once and C twice from memory and write the resulting D, totalling a transfer of 5 times 2 is 10 bytes.
   In Accumulator we find 7 memory accesses, totalling a transfer of 7 times 2 is 14 bytes.
   In Load/Store we find 3 memory accesses, totalling a transfer of 3 times 2 is 6 bytes.
   In Memory/Memory we find 3 instructions, each involving 3 memory references, totalling 9 times 2 is 18 bytes.

3. Which architecture is the most efficient as measures in code size?
   The stack architecture has the smallest code size, but the difference is marginal.

4. Which architecture is most efficient as measured by total memory traffic (code + data)?
   Stack transfers 20 instruction bytes and 10 data bytes, totalling 30 bytes.
   Accumulator transfers 20 instruction bytes and 14 data bytes, totalling 34 bytes.
   Load/Store transfers 21 instruction bytes and 6 data bytes, totalling 27 bytes.
   Memory/memory transfers 21 instruction bytes and 18 data bytes, totalling 39 bytes.
   This shows that the stack and load/store architectures are the best, the accumulator comes in on second place because its lacks for versatility, while the memory/memory architecture comes in last. Clearly the reached ranking is severely influenced by the applied matric.

V. Problem 5 (15 points)

[Q4.1/2, Q4.5/6] List all the dependencies (output, anti and true) in the following code fragment:

```
for (i=2; i<100; i=i+1) {
        a[i]=b[i]+a[i];          /* S1 */
        c[i-1]=d[i]+a[i];        /* S2 */
        a[i-1]=2 * b[i];         /* S3 */
        b[i+1]=2 * b[i];         /* S4 */
}
```

Indicate whether the true dependencies are loop-carried or not. Show why the loop is not parallel.

Loop-carried: data access in later iterations is dependent on data values produced in earlier iterations
Name: two instructions use the same register or memory location, but there is no flow of data between the instructions
Anti: one instruction is read, the other a write
Output: both are write
Data (true): consumption relation (in) direct between instructions

There are six dependencies in the C loop presented in the exercise:

1. *Anti-dependence from S1 to S1 on a.*
2. *True dependence from S1 to S2 on a.*
3. *Loop-carried true dependence from S4 to S1 on b.*
4. *Loop-carried true dependence from S4 to S3 on b.*
5. *Loop-carried true dependence from S4 to S4 on b.*
6. *Loop-carried output dependence from S3 to S1 on a.*

For a loop to be parallel, each iteration must be independent of all others, which is not the case in the code used for this exercise. Because dependencies 3, 4 and 5 are "true" dependences, they cannot be removed through renaming. In addition, as these dependencies are loop-carried, they imply that iterations of the loop are not independent. These factors together imply the loop cannot be made parallel as the loop is written.

Then here is another loop:

```
for (i=1; i<100; i=i+1) {
        a[i]=b[i]+c[i];          /* S5 */
        b[i]=a[i]+d[i];          /* S6 */
        a[i+1]= a[i]+e[i];       /* S7 */
};
```

List the dependencies and then rewrite the loop so that it is parallel.

There are four dependencies in the C loop presented in the exercise:
- True dependence from S5 to S6 on a (RAW).
- Loop-carried true dependence from S6 to S5 on b (WAR).
- Loop-carried output dependence from S7 to S5 on a.

```
for (i=1; i<99; i=i+1) {
        a[i]=b[i]+c[i];
        b[i]=a[i]+d[i];
};
a[100]= a[99]+e[99];
```