Exercises for
EIT090 Computer Architecture
HT1 2009
DRAFT

Anders Ardö
Department of Electrical and Information technology
Lund University

August 27, 2009

Contents

1 Exercises week 2

1.1 Performance

Exercise 1.1 Describe the following items concerning computer architecture:

a) "make the common case fast"

b) "locality of reference"

c) The "SPEC" benchmark series

d) Amdahl's Law

Exercise 1.2 Hennessy/Patterson, Computer Architecture, 4th ed., exercise 1.5

Exercise 1.3 Amdahl observed: "a fairly obvious conclusion which can be drawn at this point is that the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude". This was then formulated as Amdahl's Rule. This rule can be applied in various areas.

1. A common transformation required in graphics processors is square root. Implementations of floating-point (FP) square root vary significantly in performance. Suppose FP square root (FPSQR) is responsible for 20 % of the execution time of a critical graphics benchmark. One proposal is to enhance the FPSQR hardware and speed this up by a factor of 10. The alternative is just to make all FP instructions run faster; FP instructions are responsible for half of the execution time. By how much do the FP instructions have to be accelerated to achieve the same performance as achieved by inserting the specialized hardware?

2. Suppose that we want to enhance the processor used for Web serving. The new processor is 10 times faster on computation in the Web serving application than the original processor. Assuming that the original processor is busy with computation 40 % of the time and is waiting for I/O 60 % of the time, what is the overall speedup gained by incorporating the enhancement?

Exercise 1.4 Use Amdahl's law to illustrate why it is important to keep a computer system balanced in terms of relative performance between for example I/O speed and raw CPU speed.

Exercise 1.5 Three enhancements with the following speed-ups are proposed for a new architecture:

- Enhancement 1: Speed-up = 30;
- Enhancement 2: Speed-up = 20;
- Enhancement 3: Speed-up = 15.

Only one enhancement is usable at a time.

1. How can Amdahl's Law be formulated to handle multiple enhancements?

2. If enhancements 1 and 2 are usable for 25% of the time, what fraction of the time must enhancement 3 be used to achieve an overall speed-up of 10?

Assume the enhancements can be used 25%, 35% and 10% of the time for enhancements 1, 2, and 3 respectively.

3. For what fraction of the reduced execution time is no enhancement is in use?

Assume, for some benchmark, the possible fraction of use is 15% for each of the enhancements 1 and 2 and 70% for enhancement 3. We want to maximize performance.

4. If only one enhancement can be implemented, which should it be? If two enhancements can be implemented, which should be chosen?

Exercise 1.6 Hennessy/Patterson, Computer Architecture, 4th ed., exercise 1.12

Exercise 1.7 Hennessy/Patterson, Computer Architecture, 4th ed., exercise 1.13

Exercise 1.8 Hennessy/Patterson, Computer Architecture, 4th ed., exercise 1.14

2 Exercises week 3

2.1 ISA

Exercise 2.1 What is a load-store architecture and what are the advantages/disadvantages of such an architecture compared to other GPR architectures?

Exercise 2.2 What are the advantages/disadvantages of fixed-length and variable-length instruction encodings?

Exercise 2.3 Assume an instruction set that uses a fixed 16-bit instruction length. Operand specifiers are 6 bits in length. There are 5 two operand instructions and 33 zero operand instructions. What is the maximum number of one-operand instructions that can be supported?

Exercise 2.4 Consider this high-level language code sequence of three statements:
A = B + C;
B = A + C;
D = A - B;

Use the technique of copy propagation (see figure B.20) to transform the code sequence to the point where no operand is a computed value. Note the instances in which the transformation has reduced the computational work of a statement an those cases where the work has increased.

What does this suggest about the technical challenge faced in trying to satisfy the desire for optimizing compilers?

Exercise 2.5 A given processor has 32 registers, uses 16-bit immediates and has 142 instructions in its ISA. In a given program,

- 20 % of the instructions take 1 input register and have 1 output register.,
- 30 % have 2 input registers and 1 output register,
- 25 % have 1 input register, 1 output register and take an immediate input as well,
- and the remaining 25 % have one immediate input and 1 output register.

1. For each of the 4 types of instructions , how many bits are required? Assume that the ISA requires that all instructions be a multiple of 8 bits in length.

2. How much less memory does the program take up if variable-length instruction set encoding is used as opposed to fixed-length encoding?

Exercise 2.6 Compute the effective CPI for MIPS using figure B.27. Suppose we have made the following measurements of average CPI for instructions:

Instruction	**Average CPI**
All ALU instructions	1.0
Load/Store	1.4
Conditional branches	
taken	2.0
not taken	1.5
Jumps	1.2

Assume that 60 % of the conditional branches are taken and that all instructions in the 'other' category in Fig B.27 are ALU instructions. Average the instructions frequencies of gap and gcc to obtain the instruction mix.

Exercise 2.7 Your task is to compare the memory efficiency of the following instruction set architectures:

- Accumulator -– All operations occur between a single register and a memory location. There are two accumulators of which one is selected by the opcode;
- 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

a. Invent your own assembly language mnemonics (fig B.2 provides a useful sample to generalize), and for each architecture write the best equivalent assembly language code for this high level language code:
   A = B + C;
   B = A + C;
   D = A - B;

b. Label each instance in your assembly codes for part (a) 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.

c. 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 of your choice answer the following questions:

   1. How many instruction bytes are fetched?
   2. How many bytes of data are transferred from/to memory?
   3. Which architecture is the most efficient as measures in code size?
   4. Which architecture is most efficient as measured by total memory traffic (code + data)?

3 Exercises week 4

3.1 Pipelining I

Exercise 3.1 Consider following assembly-language program:

1:	MOV	R3,	R7	
2:	LD	R8,	(R3)	
3:	ADD	R3,	R3,	4
4:	LOAD	R9,	(R3)	
5:	BNE	R8,	R9,	L3

a) This program includes WAW, RAW, and WAR dependencies. Show these?

b) What is the difference between a dependency and hazard?

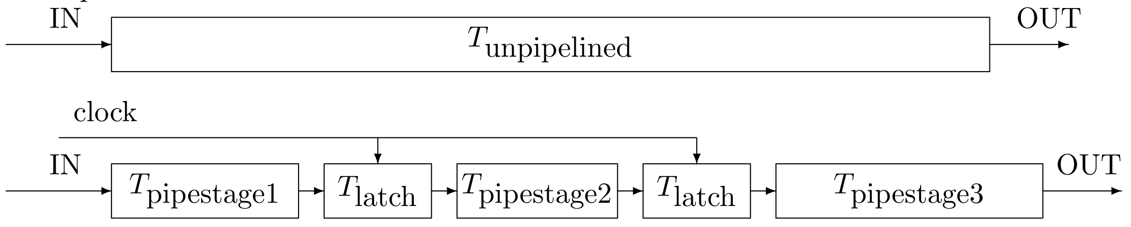
c) What is the difference between a name dependency and a true dependency?

e) Which of WAW, RAW, WAR are true dependencies and which are name dependencies?

Exercise 3.2 a) Define an expression (Speedup=) for pipeline speedup as a function of:

- $T_{\text{unpipelined}}$: execution time for the non-pipelined unit.
- $max(T_{\text{pipestage}})$: the execution time of the slowest pipe-stage.
- T_{latch}: overhead time (setup time) for the storage element between pipe-stages.

Example:



b) Define an expression for pipeline speedup as a function of (use only these):

- noofstages: number of pipe stages (assume equal pipe stage execution time).
- branch_freq (bf): relative frequency of branches in the program.
- branch_penalty (bp): number of clock cycles lost due to a branch.

c) Give an example of a piece of assembly code that contains WAW, RAW and WAR hazards and identify them. (Use for example the assembly instruction
`ADDD Rx,Ry,Rz`
which stores (Ry+Rz) in Rx)

Exercise 3.3 Use the following code fragment:

```
Loop: LD     F0,0(R2)
      LD     F4,0(R3)
      MULD   F0,F0,F4
      ADDD   F2,F0,F2
      DADDUI R2,R2,#8
      DADDUI R3,R3,#8
      DSUBU  R5,R4,R2
      BNEZ   R5,Loop
```

Assume that the initial value of R4 is R2+792.

For this exercise assume the standard five-stage integer pipeline and the MIPS FP pipeline as describe in section A.5. If structural hazards are due to write-back contention, assume the earliest instruction gets priority and other instructions are stalled.

a) Show the timing of this instruction sequence for the MIPS FP 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. Assume that the branch is handled by flushing the pipeline. If all memory references hit the cache, how many cycles does this loop take to execute?

b) Show the timing of this instruction sequence for the MIPS FP pipeline with normal forwarding and bypassing hardware. 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?

Exercise 3.4 Suppose the branch frequencies (as percentage of all instructions) are as follows:

Conditional branches	15 %
Jumps and calls	1 %
Conditional branches	60 % are taken

We are examining a four-deep pipeline where the branch is resolved at the end of the second cycle for unconditional branches and at the end of the third cycle for conditional branches. Assuming that only the first pipe stage can always be done independent of whether the branch goes and ignoring other pipeline stalls, how much faster would the machine be without any branch hazards?

Exercise 3.5 A reduced hardware implementation of the classic five stage RISC pipeline might use the EX stage hardware to perform a branch instruction comparison and then not actually deliver the branch target PC to the IF stage until the clock cycle in which the branch instruction reaches the MEM stage. Control hazard stalls can be reduced by resolving branch instructions in ID, but improving performance in one respect may reduce performance in other circumstances. How does determining branch outcome in the ID stage have the potential to increase data hazard stall cycles?

Exercise 3.6 a) Show with an example using two assembler instructions only that exceptions may be generated in a pipeline in another order than the execution order. Assume a 5 stage pipeline with stages IF, ID, EXE, MEM, and WB.

b) What is meant with "precise exceptions" and why is it important?

4 Exercises week 5

4.1 Pipelining II

Exercise 4.1 Consider an unpipelined processor. Assume that it has 1-ns clock cycle and that it uses 4 cycles for ALU operations and 5 cycles for branches and 4 cycles for memory operations. Assume that the relative frequencies of these operations are 50 %, 35 % and 15 % respectively. Suppose that due to clock skew and set up, pipelining the processor adds 0.15 ns of overhead to the clock. Ignoring any latency impact, how much speed up in the instruction execution rate will we gain from a pipeline?

Exercise 4.2 Explain what control hazard is and how to avoid it.

Exercise 4.3

- Identify all data dependencies in the following code, assuming that we are using the 5-stage MIPS pipelined datapath. Which dependencies can be resolved via forwarding?

```
ADD R2,R5,R4
ADD R4,R2,R5
SW  R5,100(R2)
ADD R3,R2,R4
```

- Consider executing the following code on the 5-stage pipelined datapath: Which registers are read during the fifth clock cycle, and which registers are written at the end of the fifth clock cycle? Consider only the registers in the register file (i.e., R1, R2, R3, etc.)

```
ADD R1,R2,R3
ADD R4,R5,R6
ADD R7,R8,R9
ADD R10,R11,R12
ADD R13,R14,R15
```

Exercise 4.4 Briefly give two ways in which loop unrolling can increase performance and one in which it can decrease performance.

Exercise 4.5 Hennessy/Patterson, Computer Architecture, 4th ed., exercise 2.2

Exercise 4.6 Hennessy/Patterson, Computer Architecture, 4th ed., exercise 2.7

5 Exercises week 6

5.1 Pipelining III

Exercise 5.1 Consider the following assembly program:

```
0       ADD       r3,r31,r2
1       LW        r6,0(r3)
2       ANDI      r7,r5,#3
3       ADD       r1,r6,r0
4       SRL       r7,r0,#8
5       OR        r2,r4,r7
6       SUB       r5,r3,r4
7       ADD       r15,r1,r10
8       LW        r6,0(r5)
9       SUB       r2,r1,r6
10      ANDI      r3,r7,#15
```

Assume the use of a four-stage pipeline: fetch (IF), decode/issue (DI), execute (EX) and write back (WB). Assume that all pipeline stages take one clock cycle except for the execute stage. For simple integer arithmetic and logical instructions, the execute stage takes one cycle, but for a load from memory, five cycles are needed in the execute stage. Suppose we have a simple scalar pipeline but allow some sort of out-of-order execution that results in the following table for the first seven instructions:

	Instruction	**IF**	**DI**	**EX**	**WB**
0	ADD r3,r31,r2	0	1	2	3
1	LW r6,0(r3)	1	2	4[a]	9
2	ANDI r7,r5,#3	2	3	5[b]	6
3	ADD r1,r6,r0	3	4	10	11
4	SRL r7,r0,#8	4	5	6	7
5	OR r2,r4,r7	5	6	8	10
6	SUB r5,r3,r4	6	7	9	12[c]

A number in the table indicates the clock cycle a certain instruction starts at a pipeline stage. There are a lot of implementation details that can be deduced from the execution table above.

a) Explain why the first lw-instruction (instruction 1) cannot start in the execute stage until clock cycle 4.

b) Explain why the first and-instruction (instruction 2) cannot start the execution stage until clock cycle 5.

c) Explain why the first sub-instruction (instruction 6) cannot start the write back stage until clock cycle 12.

d) Complete the table for the remaining instructions.

e) Suppose instruction 2 was changed to: andi r6,r5,#3. What implications would that have on the design of the pipeline? How would the table look like?

Exercise 5.2 Hennessy/Patterson, Computer Architecture, 4th ed., exercise 2.11

Exercise 5.3 Schedule the following code using Tomasulo's algorithm assuming the hardware has three Load-units with a two-cycle execution latency, three Add/Sub units with 2 cycles execution latency, and two Mult/Div units where Mult has an execution latency of 10 cycles and Div 40 cycles. Assume the first instruction (LD F6,34(R2)) is issued in cycle 1.

```
LD    F6,34(R2)
LD    F2,34(R3)
MULTD F0,F2,F4
SUBD  F8,F6,F2
DIVD  F10,F0,F6
ADDD  F6,F8,F2
```

1. In which clock cycle (numbered 0,1,2,...) does the second LD instruction complete?
2. In which clock cycle does the MULTD instruction complete?
3. In whic clock cycle does the ADDD instruction complete?

Exercise 5.4 Assume a processor with a standard five-stage pipeline (IF, ID, EX, MEM, WB) and a branch prediction unit (a branch history table) in the ID-stage. Branch resolution is performed in the EX-stage. There are four cases for conditional branches:

- The branch is not taken and correctly predicted as not taken (NT/PNT)
- The branch is not taken and predicted as taken (NT/PT)
- The branch is taken and predicted as not taken (T/PNT)
- The branch is taken and correctly predicted as taken (T/PT) Suppose that the branch penalties with this design are:
- NT/PNT: 0 cycles
- T/PT: 1 cycle
- NT/PT, T/PNT: 2 cycles

a) Describe how the CPU Performance Equation (see assignment 1 a) can be modified to take the performance of the branch prediction unit into account. Define the information you need to know to assess the performance.

b) Use the answer in the assignment above to calculate the average CPI for the processor assuming a base CPI of 1.2. Assume 20% conditional branches (disregard from other branches) and that 65% of these are taken on average. Assume further that the branch prediction unit mispredicts 12% of the conditional branches.

c) In order to increase the clock frequency from 500 MHz to 600 Mhz, a designer splits the IF-stage into two stages, IF1 and IF2. This makes it easier for the instruction cache to deliver instructions in time. This also affects the branch penalties for the branch prediction unit as follows:

   - NT/PNT: 0 cycles
   - T/PT: 2 cycles

– NT/PT, T/PNT: 3 cycles

How much faster is this new processor than the previous that runs on 500 MHz?

d) Propose a solution to reduce the average branch penalty even further.

Exercise 5.5 A processor with dynamic scheduling and issue bound operand fetch has 3 execution units – one LOAD/STORE unit, one ADD/SUB unit and one MUL/DIV unit. It has a reservation station with 1 slot per execution unit and a single register file. Starting with the following instruction sequence in the instruction fetch buffer and empty reservation stations, for each instruction find the cycle in which it will be issued and the cycle in which it will write result.

```
LOAD R6, 34(R12)
LOAD R2, 45(R13)
MUL R0, R2, R4
SUB R8, R2, R6
DIV R10, R0, R6
ADD R6, R8, R2
```

Assume out of order issue and out of order execution. Execute cycles taken by different instructions are:
LOAD/STORE: 2
ADD/SUB: 1
MUL: 2
DIV: 4

Exercise 5.6 This assignment focuses on processors with dynamic scheduling, speculative execution using a reorder buffer, and dynamic branch prediction.

```
   ANDI  R3, R3, 0    # R3 = 0
LOOP:
   LD    R4, 0(R1)
   DMUL  R5, R4, R4
   DADD  R5, R3, R5
   SD    R5, 0(R1)    # new[i] = old[i-1] + old[i]*old[i]
   DADDI R3, R4, 0
   DADDI R1, R1, 8
   BNE   R1, R2, LOOP
```

1. Explain how dynamic branch prediction works when using a 2-bit prediction scheme. Use the program above to demonstrate how the prediction works. Also, describe how the prediction works for a program containing two or more branches?

2. In what way does a reorder buffer help speculation? What is the key points when introducing support for speculation?

3. In a processor with dynamic scheduling according to Tomasulo's algorithm, hardware based speculation using a reorder buffer (ROB), and dynamic branch prediction, execution of each instruction follows a sequence of steps that is somewhat different depending on the type of instruction. Below are the steps carried out for an ALU instruction:

(a) Issue when reservation station and ROB entry is available
    - Read already available operands from registers and instruction
    - Send instruction to reservation station
    - Tag unavailable operands with ROB entry
    - Tag destination register with ROB entry
    - Write destination register to ROB entry
    - Mark ROB entry as busy

2. Execute after issue
    - Wait for operand values on CDB (if not already available)
    - Compute result

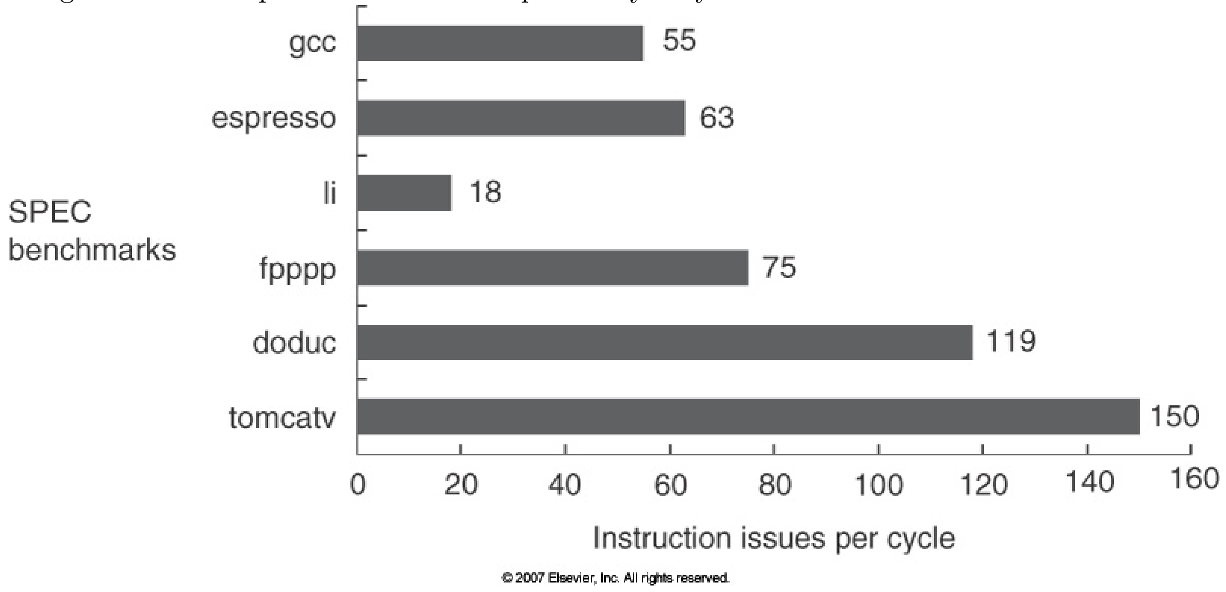
3. Write result when CDB and ROB available
    - Send result on CDB to reservation stations
    - Update ROB entry with result, and mark as ready
    - Free reservation station

4. Commit when at head of ROB and ready
    - Update destination register with result from ROB entry
    - Untag destination register
    - Free ROB entry

What are the corresponding steps for handling a store instruction?

4. Explain how RAW hazards are resolved in the basic Tomasulo's algorithm.

Exercise 5.7 As Figure 3.1, from the book Computer Architecture, A Quantitative Approah, below shows, the ILP available in many applications can be fairly high. Nevertheless, in practical processor implementations it can be hard to exploit this available ILP. List at least three factors that might limit the exploitable ILP and explain why they limit the ILP.



6 Exercises week 7

6.1 Memory systems, Cache I

Exercise 6.1 Describe the concept "memory hierarchy", and state why it is important. State the function of each part, normally used hardware components, and what problems they solve (if any).

Exercise 6.2 Suppose that CPI for a given architecture (with a perfect memory system, using 32 bit addresses) is 1,5. We are considering the following cache systems:

- A 16 KB direct mapped "unified" cache using "write-back". Miss-ratio = 2.9%. Does not affect the cycle length.
- A 16 KB 2-way set–associative "unified" cache using "write-back". Miss-ratio = 2.2%. Increases the cycle length with a factor 1.2
- A 32 KB direct mapped "unified" cache using "write-back". Miss-ratio = 2.0%. Increases the cycle length with a factor 1.25

Suppose a memory latency of 40 cycles, 4 bytes transferred per cycle and that 50% of the blocks are "dirty". There are 32 bytes per block and 20% of the instructions are "data transfer" instructions. A "write buffer" is not used.

a) Calculate effective CPI for the three cache systems.

b) Recalculate the CPI using a system with a TLB with a 0.2 % miss–rate and a 20 cycle penalty. The caches are physically addressed.

c) Which of the above cache–systems is best?

d) Draw a block diagram of the set–associative cache. Show different address fields and describe how they are used to determine hit/miss.

e) How does a TLB affect performance if the cache is virtually or physically addressed?

Exercise 6.3 What are the design trade-offs between a large register file and a large data cache?

Exercise 6.4 Suppose that it was possible to design a unified (common to instructions and data) first-level cache with two ports so that there would be no structural hazard in the the pipeline. What are the design trade-offs between using the unified cache compared to separate instruction and data cache? The total amount of cache memory should of course be the same in both cases.

Exercise 6.5 What is a write-through cache? Is it faster/slower than a write-back cache with respect to the time it takes for writing.

Exercise 6.6 Hennessy/Patterson, Computer Architecture, 4th ed., exercise 5.1

Exercise 6.7 Let's try to show how you can make unfair benchmarks. Here are two machines with the same processor and main memory but different cache organizations. Assume that both processors run at 2 GHz, have a CPI of 1, and have a cache (read) miss time of 100 ns. Further, assume that writing a 32-bit word in main memory requires 100 ns (for the write-through cache). The caches are unified – they contain both instructions and data -, and each cache has a total capacity of 64 kB, not including tags and status bits. The cache on system A is two-way set associative and has 32-byte blocks. It is write-through and does not allocate a block on a write miss. The cache on system B is direct mapped and has 32-byte blocks. It is write-back and allocates a block on a write miss.

- Describe a program that makes system A run as fast as possible relative to system B's speed. How much faster is this program on system A as compared to system B.
- Describe a program that makes system B run as fast as possible relative to system A's speed. How much faster is this program on system B as compared to system A.

Exercise 6.8 Assume having a two level memory hierarchy: a cache and a main memory, which are connected with a 32 bit wide bus. A hit in the cache can be executed within one clock cycle. At a cache miss an entire block must be replaced. This is done by sending the address (32 bits) to the memory which needs 4 clock cycles before it can send back a block by the bus. Every bus-transfer requires one clock cycle. The processor will need to wait until the entire block is in the cache. The following table shows the average miss ratio for different block sizes:

Block size (B, bytes)	**Miss-ratio (M), %**
4	4.5
16	2.4
32	1.6
64	1.0
128	0.64

a) Which block size results in the best average memory-access time?

b) If bus arbitration requires 2 clock cycles in average, which block size is the most optimal?

c) Which block size is the best one if the bus between the cache and the main memory is widen to 64 bits (both with and without the cost of the bus-arbitration of assignment a and b)?

Assume a cache with a size of 256 bytes and a block size of 16 bytes. The blocks in the cache are numbered from 0 and upwards. Specify the sizes of the tag, index and byte fields when using 16 bits addressing, and in which block (or blocks) in the cache the address $28E7_{16}$ is represented, for:

d) A direct mapped cache.

e) A 2-way set-associative cache.

Exercise 6.9 a) What is a memory hierarchy? Describe the problem a memory hierarchy solves and explain why it works so well.

b) A system has a 256 byte large cache. It is a set-associative cache with 16 sets. An address is 16 bits and the tag field in the address is 9 bits.

- Specify the number of blocks in the cache.
- Specify the size of the set and byte offset fields.
- Draw a sketch of how the cache is organized and describe what the different address fields are used for.
- In which set can a byte on the address 3333_{16} end up in, and which byte within the block is it?

c) Explain and discuss two different ways to handle writes in the cache.

7 Brief answers

7.1 Performance

1.1 a) Invest in resources that are used often!

b) A computer uses data (and instructions) that are often close in the address space (spacial locality), and also close in time (temporal locality).

c) SPEC = "Standard Performance Evaluation Corporation", a series of typical integer and floating point programs used to characterize the performance of a computer. Exists from several years: SPEC89, SPEC92, SPEC95, SPEC2000, ...

d) "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".

Suppose that enhancement E accelerates a fraction F of the task by a factor S, and the remainder of the task is unaffected, then $T_{exe}(withE) = T_{exe}(withoutE) * [(1-F) + F/S]$.

1.2 See 'Case Study Solutions' at http://www.elsevierdirect.com/companion.jsp?ISBN=9780123704900

1.3 1. For the specialized hardware we find as speedup

$$FPSQR : S = 1/[(1–0.2) + 0.2/10] = 1/0.82 = 1.22$$

To achieve the same performance we have to make the floating-point instructions about 60% faster:

$$FP : S = 1/[(1–0.5) + 0.5/1.6] = 1/0.8125 = 1.23$$

1.3 With a speedup of 10 for a fraction of 0.4, the overall speedup becomes

$$S = 1/[0.6 + 0.4/10] = 1/0.64 = 1.56$$

1.4 Amdahl's law: system speedup limited by the slowest component:

- Assume 10% I/O
- CPU speedup = 10 $\Longrightarrow$ System speedup = 5
- CPU speedup = 100 $\Longrightarrow$ System speedup = 10

I/O will more and more become a bottleneck!

1.5 Amdahl's Law can be generalized to handle multiple enhancements. If only one enhancement can be used at a time during program execution, then for enhancements 1,2,3,...,i

$$Speedup = \left[1 - \sum_i FE_i + \sum_i \frac{FE_i}{SE_i}\right]^{-1}$$

where FE_i is the fraction of time that enhancement i an be used and SE_i is the speedup of enhancement i. For a single enhancement the equation reduces to the familiar form of Amdahl's Law.

1.5 1. With three enhancements we have

$$Speedup = \left[1 - (FE_1 + FE_2 + FE_3) + \left(\frac{FE_1}{SE_1} + \frac{FE_2}{SE_2} + \frac{FE_3}{SE_3}\right)\right]^{-1}$$

1.5 2. Substituting in the known quantities gives

$$10 = \left[1 - (0.25 + 0.25 + FE_3) + \left(\frac{0.25}{30} + \frac{0.25}{20} + \frac{FE_3}{15}\right)\right]^{-1}$$

Solving yields $FE_3 = 0.45$. Thus, the third enhancement must be usable 45% of the time.

1.5 3. Let T_e and TNE_e denote execution time with enhancement and the time during enhanced execution in which no enhancements are in use, respectively. Let $T_{original}$ and $FNE_{original}$ stand for execution time without enhancements and the fraction of that time that cannot be enhanced. Finally, let FNE_e represent the fraction of the reduced (enhanced) execution time for which no enhancement is in use. By definition

$$FNE_e = TNE_e/T_e$$

. Because the time spent executing code that cannot be enhanced is the same whether enhancements are in use or not, and by Amdahl's Law, we have

$$\frac{TNE_e}{T_e} = \frac{FNE_{original} * T_{original}}{T_{original}/Speedup}$$

Canceling factors and substituting equivalent expressions for $FNE_{original}$ and Speedup yields

$$\frac{FNE_{original} * T_{original}}{T_{original}/Speedup} = \frac{1 - \sum_i FE_i}{1 - \sum_i FE_i + \sum_i \frac{FE_i}{SE_i}}$$

Substituting with known quantities,

$$FNE_e = \frac{1 - (0.25 + 0.35 + 0.10)}{1 - (0.25 + 0.35 + 0.10) + (\frac{0.25}{30} + \frac{0.35}{20} + \frac{0.10}{15})} = \frac{0.3}{0.3325} = 90\%$$

1.5 4. Let the speedup when implementing only enhancement i be $Speedup_i$, and let $Speedup_{ij}$ denote the speedup when employing enhancement i and j.

$$Speedup_1 = \left(1 - 0.15 + \frac{0.15}{30}\right)^{-1} = 1.17$$

$$Speedup_2 = \left(1 - 0.15 + \frac{0.15}{20}\right)^{-1} = 1.17$$

$$Speedup_3 = \left(1 - 0.70 + \frac{0.70}{15}\right)^{-1} = 2.88$$

Thus, if only one enhancement can be implemented, enhancement 3 offers much greater speedup.

$$Speedup_{12} = \left(1 - (0.15 + 0.15) + (\frac{0.15}{30} + \frac{0.15}{20})\right)^{-1} = 1.40$$

$$Speedup_{13} = \left(1 - (0.15 + 0.70) + (\frac{0.15}{30} + \frac{0.70}{15})\right)^{-1} = 4.96$$

$$Speedup_{23} = \left(1 - (0.15 + 0.70) + (\frac{0.15}{20} + \frac{0.70}{15})\right)^{-1} = 4.90$$

Thus, if only a pair of enhancements can be implemented, enhancements 1 and 3 offer the greatest speedup.

Selecting the fastest enhancement(s) may not yield the highest speedup. As Amdahl's Law states, an enhancement contributes to speedup only for the fraction of time it can be used.

1.6 See 'Case Study Solutions' at http://www.elsevierdirect.com/companion.jsp?ISBN=9780123704900

1.7 See 'Case Study Solutions' at http://www.elsevierdirect.com/companion.jsp?ISBN=9780123704900

1.8 See 'Case Study Solutions' at http://www.elsevierdirect.com/companion.jsp?ISBN=9780123704900

7.2 ISA

2.1 A load-store architecture is one in which only load and store instructions can access the memory system. In other GPR architectures, some or all of the other instructions may read their operands from or write their results to the memory system.

The primary advantage of non-load-store architectures is the reduced number of instructions required to implement a program and lower pressure on the register file.

The advantage of load-store architectures is that limiting the set of instructions that can access the memory system makes the micro-architecture simpler, which often allows implementation at a higher clock rate.

Depending on whether the clock rate increase or the decrease in number of instructions is more significant, either approach can result in greater performance.

2.2 Variable-length instruction encodings reduce the amount of memory that programs take up, since each instruction takes only as much space as it requires. Instructions in a fixed-length encoding all take up as much storage as the longest instruction in the ISA, meaning that there is some number of wasted bits in the encoding of instructions that take fewer operands, don't allow immediate constants, etc.

However, variable-length instruction sets require more complex instruction decode logic than fixed-length instructions sets, and they make it harder to calculate the address of the next instruction in memory. Therefore, processors with fixed-length instruction sets can often be implemented at higher clock rate than processors with variable-length instruction sets.

2.3 Use 2 bits for instruction type (4 different types), two-operand instructions need two of those (we need to encode 5 instructions and they use 12 bits for operand specifiers leaving us with $16-2-12 = 2$ bits for instruction code), one-operand 1, and zero-operand 1. One-operand instructions need a 6 bit operand specifier $\Rightarrow 16-2-6 = 8$ bits $\Rightarrow$ 256 one-operand instructions

2.4 Take the code sequence one line at a time.

1. A = B + C ;	The operands here are given, not computed by the code, so copy propagation will not transform this statement.
2. B = A + C ;	Here A is a computed value, so transform the code by substituting A = B + C to get
B = B + C + C ;	No operand is computed
3. D = A - B ;	Both operands are computed so substitute for both to get
D = (B + C) - (B + C + C) ;	Simplify algebraically to get
D = - C ;	This is a given, not computed, operand

Copy propagation has increased the work in statement 2 from one addition to two. It has changed the work in statement 3 from subtraction to negation, possibly a savings.

The above suggests that writing optimizing compilers means incorporating sophisticated trade-off analysis capability to control any optimizing steps, if the best results are to be achieved.

2.5 1. 142 instructions ⇒ **8** bits ($128 < 142 < 256$); 32 registers ⇒ **5** bits; **16** bit immediates

- 1 reg in, 1 reg out: $8 + 5 + 5 = 18$ bits ⇒ 24 bits

- 2 reg in, 1 reg out: $8 + 5 + 5 + 5 = 23$ bits ⇒ 24 bits

- 1 reg in, 1 reg out, 1 imm: $8 + 5 + 5 + 16 = 34$ bits ⇒ 40 bits

- 1 imm in, 1 reg out: $8 + 16 + 5 = 29$ bits ⇒ 32 bits

2. Since the largest instruction type requires 40-bit instructions, the fixed-length encoding will have 40 bits per instruction. Each instruction type in the variable encoding will use the number of bits from 1: $0.2 * 24 + 0.3 * 24 + 0.25 * 40 + 0.25 * 32 = 30$ bits on average, ie 25 % less space.

2.6 The first challenge of this exercise is to obtain the instruction mix. The instruction frequencies in Figure 2.32 must add to 100.0, although gap and gcc add to 100.2 and 99.5 %, respectively, because of rounding errors. Because each each total must in reality be 100.0, we should not attempt to scale the per instruction average frequencies by the shown totals of 100.2 and 99.5. However, in computing the average frequencies to one significant digit to the right of the decimal point, we should be careful to use an unbiased rounding scheme so that the total of the averaged frequencies is kept as close to 100 as possible. One such scheme is called round to even, which makes the least significant digit always even. For example, 0.15 rounds to 0.2, but 0.25 also rounds to 0.2. For a summation of terms, round to even will not accumulate an error as would, for example, rounding up where 0.15 rounds to 0.2 and 0.25 rounds to 0.3.

The gap and gcc the average instruction frequencies are shown below.

Instruction	**Average frequency gap, gcc**	**Category**
load	25.8	load/store
store	11.8	load/store
add	20.0	ALU
sub	2.0	ALU
mul	0.8	ALU
compare	4.4	ALU
load imm	3.6	ALU
cond branch	10.7	Cond branch
cond move	0.5	ALU
jump	0.8	jump
call	1.1	jump
return	1.1	jump
shift	2.4	ALU
and	4.4	ALU
or	8.2	ALU
xor	2.0	ALU
other logical	0.2	ALU

The exercise statement gives CPI information in terms of four major instruction categories, with two subcategories for conditional branches. To compute the average CPI we need to aggregate the instruction frequencies to match these categories. This is the second challenge, because it is easy to miscategorize instructions. The four main categories are ALU, load/store, conditional branch, and jumps. ALU instructions are any that take operands from the set of registers and return a result to that set of registers. Load/store instructions access memory. Conditional branch instructions must be able to set the program counter to a new value based on a condition. Jump-type instructions set the program counter to a new value no matter what.

With the above category definitions, the frequency of ALU instructions is the sum of the frequencies of the add, sub, mul, compare, load imm (remember, this instruction does not access memory, instead the value to load is encoded in a field within the instruction itself), cond move (implemented as an OR instruction between a controlling register and the register with the data to move to the destination register), shift, and, or, xor, and other logical for a total of 48.5 %. The frequency of load/store instructions is the sum of the frequencies of load and store for a total of 37.6 %. The frequency of conditional branches is 10.7 %. Finally, the frequency of jumps is the sum of the frequencies of the jump-type instructions, namely jump, call, and return, for a total of 3.0 %.

$$Effective\_CPI = \sum_{categories} Instruction\_category\_frequency * Clock\_cycles\_for\_category$$

$$= 0.485 * 1.0 + 0.367 * 1.4 + 0.107 * (0.6 * 2.0 + (1 - 0.6) * 1.5) + 0.03 * 1.2 = 1.24$$

2.7 a. Assembly programs:

Stack	**Acc**	**Load-store**	**Mem-mem**
Push B	Load B	Load R2,B	Add A,B,C
Push C	Add C	Load R3,C	Add B,A,C
Add	Store A	Add R1,R2,R3	Sub C,A,B
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			

b. Labels (L)

- R: Value is reloaded from memory.
- P: Result from one instruction is consumed from storage within the processor
- M: Result from one instruction is consumed from memory storage

Stack	L	**Acc**	L	**Load-store**	L	**Mem-mem**	L
Push B		Load B		Load R2,B		Add A,B,C	
Push C		Add C		Load R3,C		Add B,A,C	M
Add	P	Store A	M	Add R1,R2,R3	P	Sub C,A,B	M
PushTop		Add C	R	Add R2,R1,R3	P		
Push C	R	Store B	M	Sub R4,R1,R2	P		
Add	P	Sub A	P	Store R4,D			
PushTop		Store D					
Pop B							
Sub	P						
Pop D							

c. Code sizes

Stack:	1 byte for opcode plus 2 bytes for memory addresses when needed
Accumulator:	1 byte for opcode plus 2 bytes for memory addresses
Load-store:	1 byte for opcode, half a byte for subject register, plus either 2 bytes for memory addresses or 1 byte for operand registers (round to whole bytes).
Mem-mem:	1 byte for opcode plus 6 bytes for memory addresses

	Stack	**Acc**	**Load-store**	**Mem-mem**
1. Code	$5*1+5*3$ 20	$7*3$ 21	$3*3+3*4$ 21	$3*7$ 21
2. Mem	$5*2$ 10	$7*2$ 14	$3*2$ 6	$9*2$ 18
3.	*			
4. Code+Mem	30	34	**27**	39

7.3 Pipelining I

3.1 a) WAW: 1,3;
RAW: 1,2; 1,3, 3,4; 2,5; 4,5
WAR: 2,3;

b) A hazard is created when there is a dependency between instructions and they are close enough that the overlap caused by pipelining (or reordering) would change the order of access to the dependent operand.

c) In a true dependency information is transmitted between instructions, while this is not the case for name dependencies.

d) Name dependencies: WAR, WAW
True dependencies: RAW

3.2 a) Speedup $= \frac{T_{\text{unpipelined}}}{max(T_{\text{pipestage}})+T_{\text{latch}}}$

b) Speedup $= \frac{noofstages}{1+bf*bp}$

c) 1: MOV R3, R7
2: LD R8,(R3)
3: ADDDI R3, R3, 4
4: LD R9, (R3)
5: BNE R8, R9, Loop

WAW: 1,3 RAW: 1,2; 1,3; 2,5; 3,4; 4,5 WAR: 2,3;

3.3 The pipeline of Sections A.4 and A.5 resolves branches in ID and has multiple execution function units. More than one instruction may be in execution at the same time, but the exercise statement says write-back contention is possible and is handled by processing one instruction at a time in that stage.

Figure A.30 lists the latencies for the functional units; however, it is important to note that these data are for functional unit results forwarded to the EX stage. In particular, despite a latency of 1 for data memory access, it is still the case that for a cache hit the MEM stage completes memory access in one clock cycle.

Finally, examining the code reveals that the loop iterates 99 times. With this and analysis of iteration timing on the pipeline, we can determine loop execution time.

a) Figure 1 shows the timing of instructions from the loop for the first version of pipeline hardware. There are several stall cycles shown in the timing diagram:

- Cycles 5–6: MUL.D stalls in ID to wait for F0 and F4 to be written back by the L.D instructions.
- Cycles 8–15: ADD.D stalls in ID to wait for MUL.D to write back F0.
- Cycle 19: DSUBU stalls in ID to wait for DADDUI to write back R2.
- Cycles 21–22: BNEZ stalls in ID to wait for DSUBU to write back R5. Because the register file can read and write in the same cycle, the BNEZ can read the DSUBU result and resolve in the same cycle in which DSUBU writes that result.

- Cycle 20: While not labeled a stall, because initially it does not appear to be a stall, the fetch made in this cycle will be discarded because this pipeline design handles the uncertainty of where to fetch after a branch by flushing the stages with instructions fetched after the branch. The pipeline begins processing after the branch with the correct fetch in cycle 23.

There are no structural hazard stalls due to write-back contention because processing instructions as soon as otherwise possible happens to use WB at most once in any clock cycle.

```
                                          Clock cycle
 Instruction     1 2 3 4 5 6 7 8 ... 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
LD     F0,0(R2)  F D E M W
LD     F4,0(R3)    F D E M W
MULD   F0,F0,F4      F D s s E E ...  E  M  W
ADDD   F2,F0,F2        F s s D s ...  s  s  s  E  E  E  E  M  W
DADDUI R2,R2,#8              F s ...  s  s  s  D  E  M  W
DADDUI R3,R3,#8                                F  D  E  M  W
DSUBU  R5,R4,R2                                   F  D  s  E  M  W
BNEZ   R5,Loop                                       F  s  D  s  r
LD     F0,0(R2)                                            F  s  s  F  D  E  M  W
```

Figure 1: Pipeline timing diagram for the pipeline without forwarding, branches that flush the pipeline, memory references that hit in cache, and FP latencies from Figure A.30. The abbreviations F, D, E, M, and W denote the fetch, decode, execute, memory access, and write-back stages, respectively. Pipeline stalls are indicated by s, branch resolution by r. One complete loop iteration plus the first instruction of the subsequent iteration is shown to make clear how the branch is handled. Because branch instructions complete (resolve) in the decode stage, use of the following stages by the BNEZ instruction is not depicted.

Figure 1 shows two instructions simultaneously in execution in clock cycles 17 and 18, but because different functional units are handling each instruction there is no structural hazard.

The first iteration ends with cycle 22, and the next iteration starts with cycle 23. Thus, each of the 99 loop iterations will take 22 cycles, so the total loop execution time is $99 * 22 = 2178$ clock cycles.

b) Figure 2 shows the timing of instructions from the loop for the second version of pipeline hardware.

   There are several stall cycles shown in the timing diagram:

   - Cycle 5: MUL.D stalls at ID waiting for L.D to forward F4 to EX from MEM. F0 reaches the register file by the first half of cycle 5 and thus is read by ID during this stall cycle.
   - Cycles 7–12: ADD.D stalls at ID waiting for MUL.D to produce and forward the new value for F0.

– Cycle 16: DSUBU is stalled at ID to avoid contention with ADD.D for the WB stage. Note the complexity of pipeline state analysis that the ID stage must perform to ensure correct pipeline operation.
– Cycle 18: BNEZ stalls in ID to wait for DSUBU to produce and forward the new value for R5. While forwarding may deliver the needed value earlier in the clock cycle than can reading from the register file, and so in principle the branch could resolve earlier in the cycle, the next PC value cannot be used until the IF stage is ready, which will be with cycle 19.
– Cycle 17: Initially this cycle does not appear to be a stall because branches are predicted not taken and this fetch is from the fall-through location. However, for all but the last loop iteration this branch is mispredicted. Thus, the fetch in cycle 17 must be redone at the branch target, as shown in cycle 19.

```
                                 Clock cycle
 Instruction        1 2 3 4 5 6 7 . . . 12 13 14 15 16 17 18 19 20 21 22 23
LD     F0,0(R2)     F D E M W
LD     F4,0(R3)       F D E M W
MULD   F0,F0,F4         F D s E E . . .  E  M  W
ADDD   F2,F0,F2           F s D s . . .  s  E  E  E  E  M  W
DADDUI R2,R2,#8               F s . . .  s  D  E  M  W
DADDUI R3,R3,#8                             F  D  E  M  W
DSUBU  R5,R4,R2                                F  D  s  E  M  W
BNEZ   R5,Loop                                    F  s  D  r
LD     F0,0(R2)                                         F  s  F  D  E  M  W
```

Figure 2: Pipeline timing diagram for the pipeline with forwarding, branches handled by predicted-not-taken, memory references that hit in cache, and FP latencies from Figure A.30. The notation used is the same as in Figure 1.

Again, there are instances of two instructions in the execute stage simultaneously, but using different functional units.

The first iteration ends in cycle 19 when DSUBU writes back R5. The second iteration begins with the fetch of L.D F0, 0(R2) in cycle 19. Thus, all iterations, except the last, take 18 cycles. The last iteration completes in a total of 19 cycles. However, if there were code following the instructions of the loop, they would start after only 16 cycles of the last iteration because the branch is predicted correctly for the last iteration.

The total loop execution time is $98 * 18 + 19 = 1783$ clock cycles.

3.4 This exercise asks, "How much faster would the machine be . . . ," which should make you immediately think speedup. In this case, we are interested in how the presence or absence of control hazards changes the pipeline speedup. Recall one of the expressions for the speedup from pipelining presented on page A-13

$$Pipeline\_speedup = \frac{1}{1 + Pipeline\_stalls} * Pipeline\_depth \qquad (1)$$

where the only contributions to Pipeline stalls arise from control hazards because the exercise is only focused on such hazards. To solve this exercise, we will compute the speedup due to pipelining both with and without control hazards and then compare these two numbers.

For the "ideal" case where there are no control hazards, and thus stalls, Equation 1 yields

$$Pipeline\_speedup_{ideal} = \frac{1}{1+0} * 4 = 4 \tag{2}$$

where, from the exercise statement the pipeline depth is 4 and the number of stalls is 0 as there are no control hazards.

For the "real" case where there are control hazards, the pipeline depth is still 4, but the number of stalls is no longer 0 as it was in Equation 2. To determine the value of Pipeline stalls, which includes the effects of control hazards, we need three pieces of information. First, we must establish the "types" of control flow instructions we can encounter in a program. From the exercise statement, there are three types of control flow instructions: taken conditional branches, not-taken conditional branches, and jumps and calls. Second, we must evaluate the number of stall cycles caused by each type of control flow instruction. And third, we must find the frequency at which each type of control flow instruction occurs in code. Such values are given in the exercise statement.

To determine the second piece of information, the number of stall cycles created by each of the three types of control flow instructions, we examine how the pipeline behaves under the appropriate conditions. For the purposes of discussion, we will assume the four stages of the pipeline are Instruction Fetch, Instruction Decode, Execute, and Write Back (abbreviated IF, ID, EX, and WB, respectively). A specific structure is not necessary to solve the exercise; this structure was chosen simply to ease the following discussion.

First, let us consider how the pipeline handles a jump or call. Figure 3 illustrates the behavior of the pipeline during the execution of a jump or call. Because the first pipe stage can always be done independently of whether the control flow instruction goes or not, in cycle 2 the pipeline fetches the instruction following the jump or call (note that this is all we can do – IF must update the PC, and the next sequential address is the only address known at this point; however, this behavior will prove to be beneficial for conditional branches as we will see shortly). By the end of cycle 2, the jump or call resolves (recall that the exercise specifies that calls and jumps resolve at the end of the second stage), and the pipeline realizes that the fetch it issued in cycle 2 was to the wrong address (remember, the fetch in cycle 2 retrieves the instruction immediately following the control flow instruction rather than the target instruction), so the pipeline reissues the fetch of instruction i + 1 in cycle 3. This causes a one-cycle stall in the pipeline since the fetches of instructions after i + 1 occur one cycle later than they ideally could have.

Figure 4 illustrates how the pipeline stalls for two cycles when it encounters a taken conditional branch. As was the case for unconditional branches, the fetch issued in cycle 2 fetches the instruction after the branch rather than the instruction at the target of the branch. Therefore, when the branch finally resolves in cycle 3 (recall that the exercise specifies that conditional branches resolve at the end of the third stage), the pipeline realizes it must reissue the fetch for instruction i + 1 in cycle 4, which creates the two-cycle penalty.

Figure 5 illustrates how the pipeline stalls for a single cycle when it encounters a not-taken conditional branch. For not-taken conditional branches, the fetch of instruction i + 1 issued in cycle 2 actually obtains the correct instruction. This occurs because the pipeline fetches the next sequential instruction from the program by default—which happens to be the instruction that follows a not-taken branch. Once the conditional branch resolves in cycle 3, the pipeline determines it does not need to reissue the fetch of instruction i + 1 and therefore can resume executing the instruction it fetched in cycle 2. Instruction i + 1 cannot leave the IF stage until after the branch resolves because the exercise specifies the pipeline is only capable of using the IF stage while a branch is being resolved.

			Clock cycle			
Instruction	1	2	3	4	5	6
Jump or call	IF	ID	EX	WB		
i+1		IF	IF	ID	EX	...
i+2			stall	IF	ID	...
i+3				stall	IF	...

Figure 3: Effects of a jump or call Instruction on the pipeline.

			Clock cycle			
Instruction	1	2	3	4	5	6
Taken branch	IF	ID	EX	WB		
i+1		IF	stall	IF	ID	...
i+2			stall	stall	IF	...
i+3				stall	stall	...

Figure 4: Effects of a taken conditional branch on the pipeline.

Combining all of our information on control flow instruction type, stall cycles, and frequency leads us to Figure 6.

			Clock cycle			
Instruction	1	2	3	4	5	6
Not-taken branch	IF	ID	EX	WB		
i+1		IF	stall	ID	EX	...
i+2			stall	IF	ID	...
i+3				stall	IF	...

Figure 5: Effects of a not-taken conditional branch on the pipeline.

Note that this figure accounts for the taken/not-taken nature of conditional branches. With this information we can compute the stall cycles caused by control flow instructions:

$$Pipeline\_stalls_{real} = (1 * 1\%) + (2 * 9\%) + (1 * 6\%) = 0.24 \tag{3}$$

where each term is the product of a frequency and a penalty. We can now plug the appropriate value for $Pipeline\_stalls_{real}$ into Equation 1 to arrive at the pipeline speedup in the "real" case:

$$Pipeline\_speedup_{real} = \frac{1}{1 + 0.24} * (4.0) = 3.23 \tag{4}$$

Finding the speedup of the ideal over the real pipelining speedups from Equations 2 and 4 leads us to the final answer: 4

$$Pipeline\_speedup_{without\_control\_hazards} = 4/3.23 = 1.24 \tag{5}$$

Thus, the presence of control hazards in the pipeline loses approximately 24% of the speedup you achieve without such hazards.

Control flow type	Frequency (per instruction)	Stalls (cycles)
Jumps and calls	1%	1
Conditional (taken)	15% * 60% = 9%	2
Conditional (not taken)	15% * 40% = 6%	1

Figure 6: A summary of the behavior of control flow instructions.

3.5 If a branch outcome is to be determined earlier, then the branch must be able to read its operand equally early. Branch direction is controlled by a register value that may either be loaded or computed. If the branch register value comparison is performed in the EX stage, then forwarding can deliver a computed value produced by the immediately preceding instruction if that instruction needs only one cycle in EX. There is no data hazard stall for this case. Forwarding can deliver a loaded value without a data hazard stall if the load can perform memory access in a single cycle and if at least one instruction separates it from the branch.

If now the branch compare is done in the ID stage, the two forwarding cases just discussed will each result in one data hazard stall cycle because the branch will need its operand one cycle before it exists in the pipeline. Often, instructions can be scheduled so that there are more instructions between the one producing the value and the dependent branch. With enough separation there is no data hazard stall.

So, resolving branches early reduces the control hazard stalls in a pipeline. However, without a sufficient combination of forwarding and scheduling, the savings in control hazard stalls will be offset by an increase in data hazard stalls.

3.6 a) One such situation occurs if we have e.g. the instruction LW (Load Word) in the MEM stage performing a data memory read operation exhibiting a "page fault" (because the referenced data is not present in PM), and any other instruction (say e.g. ADD) issued after LW exhibiting a "page fault" in the IF stage (because the referenced instruction is not present in PM). In this case LW will generate an exception after the second instruction (ADD) generates an exception.

b) If the pipeline can be stopped so that the the instructions just before the faulting instruction are completed and those after it can be restarted from scratch, the pipeline is said to have precise exceptions.

It is important in order to be able to correctly implement virtual memory and IEEE arithmetic trap handlers.

7.4 Pipelining II

4.1 The average instruction execution time on an unpipelined processor is

$$clockcycle * Avg.CPI = 1ns * ((0.5 * 4) + (0.35 * 5) + (0.15 * 4)) = 4.35ns$$

The avg. instruction execution time on pipelined processor is $= 1ns + 0.2ns = 1.2ns$ So speed up $= 4.35/1.2 = 3.3625$

4.2 Control hazard: Instruction fetch depends on some in-flight instruction being executed. For example, the target address of a branch or jump is not immediately available after the branch / jump exits from fetch stage.

- design a hazard unit to detect the hazard and stall the pipeline
- branch prediction
- using delay slot in the instruction scheduling

4.3
- The second instr. is dependent upon the first (because of R2) The third instr. is dependent upon the first (because of R2) The fourth instr. is dependent upon the first (because of R2) The fourth instr. is dependent upon the second (because of R4) There, All these dependencies can be solved by forwarding
- Registers R11 and R12 are read during the fifth clock cycle. Register R1 is written at the end of fifth clock cycle.

4.4 Performance can be increased by:

- Fewer loop conditional evaluations.
- Fewer branches/jumps.
- Opportunities to reorder instructions across iterations.
- Opportunities to merge loads and stores across iterations. Performance can be decreased by:
- Increased pressure on the I-cache.
- Large branch/jump distances may require slower instructions

4.5 See 'Case Study Solutions' at http://www.elsevierdirect.com/companion.jsp?ISBN=9780123704900

4.6 See 'Case Study Solutions' at http://www.elsevierdirect.com/companion.jsp?ISBN=9780123704900

7.5 Pipelining III

5.1 a) Because there is no forwarding from the WB-stage and the correct value of source register r3 is therefore not available until clock cycle 4.

b) Only one instruction can be issued in each clock cycle and since instruction 1 has to wait, instruction 2 also must wait one clock cycle.

c) The write back stage is occupied in clock cycles 10 and 11 by instructions that have been issued earlier.

d) Here is the continuation of the table:

	Instruction	IF	DI	EX	WB	Comment
0	ADD r3,r31,r2	0	1	2	3	
1	LW r6,0(r3)	1	2	4	9	
2	ANDI r7,r5,#3	2	3	5	6	
3	ADD r1,r6,r0	3	4	10	11	
4	SRL r7,r0,#8	4	5	6	7	
5	OR r2,r4,r7	5	6	8	10	
6	SUB r5,r3,r4	6	7	9	12	
7	ADD r15,r1,r10	7	8	12	13	Wait for register r1
8	LW r6,0(r5)	8	9	13	18	Wait for register r5
9	SUB r2,r1,r6	9	10	19	20	Wait for register r6
10	ANDI r3,r7,#15	10	11	14	15	Can execute out-of-order

e) The main implication is that instruction 2 is not allowed to execute out-of-order in relation to instruction 1. There is a name-dependence between these instructions and if instruction 2 completes before instruction 1 there will be a WAW-hazard. Instruction 2 is stalled in the DI stage and the table must be modified:

	Instruction	IF	DI	EX	WB
0	ADD r3,r31,r2	0	1	2	3
1	LW r6,0(r3)	1	2	4	9
2	**ANDI r7,r5,#3**	2	3	**9**	**10**
3	ADD r1,r6,r0	3	4	10	11
4	SRL r7,r0,#8	4	5	6	7
5	OR r2,r4,r7	5	6	8	10
6	SUB r5,r3,r4	6	7	9	12

5.2 See 'Case Study Solutions' at http://www.elsevierdirect.com/companion.jsp?ISBN=9780123704900

5.3
1. 5
2. 16
3. 11

5.4 a) The instruction count and the clock cycle time are not affected. Therefore the only modification is in the CPI count which will have an addition that comes from branch instructions. The addition depends on the relative frequency of conditional branch instructions, f_{branch}, the fraction of these that are taken, b_{taken}, and the fraction of the branches that are miss-predicted, $b_{misspred}$.

$$\begin{aligned} CPI = CPI_{base} + f_{branch} * \Big(& b_{taken} * (b_{misspred} * 2 + (1 - b_{misspred}) * 1) \\ & +(1 - b_{taken}) * (b_{misspred} * 2 + (1 - b_{misspred}) * 0) \\ \Big) & \end{aligned}$$

b) $f_{branch} = 0.2$, $b_{taken} = 0.65$, $b_{misspred} = 0.12$.

$$\begin{aligned} CPI &= 1.2 + 0.2 * (0.65 * (0.12 * 2 + 0.88 * 1) + 0.35 * (0.12 * 2 + 0.88 * 0)) \\ &= 1.2 + 0.2 * (0.728 + 0.084) \\ &= 1.3624 \end{aligned}$$

c)

$$\begin{aligned}
CPI_{new} &= 1.2 + 0.2 * (0.65 * (0.12 * 3 + 0.88 * 2) + 0.35 * (0.12 * 3 + 0.88 * 0)) \\
&= 1.2 + 0.2 * (1.378 + 0.126) \\
&= 1.5008 \\
T_{exeold} &= IC * CPI_{old} * 1/500 \\
T_{exenew} &= IC * CPI_{new} * 1/600 \\
Speedup &= T_{exeold}/T_{exenew} \\
&= (CPI_{old} * 1/500)/(CPI_{new} * 1/600) \\
&= (1.3624/500)/(1.5008/600) \\
&= 1.089
\end{aligned}$$

The new processor with higher clock frequency is thus only 8.9% faster than the old even though the clock frequency has been increased by 20%. The sole reason for this is the branch hazards.

d) A branch target buffer can be used in the IF-stage. The branch penalty for correctly predicted branches will then be 0.

5.5 The following chart shows the execution of the given instruction sequence cycle by cycle. The stages of instruction execution:

- F Instruction fetch
- D Decode and issue
- E1 Execute in LOAD/STORE unit
- E2 Execute in ADD/SUB unit
- E3 Execute in MUL/DIV unit
- W Write back into register file and reservation stations

```
  Instruction       1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
LOAD R6, 34(R12) F  D E1 E1  W
LOAD R2, 45(R13) F  r  r  r  D E1 E1  W
MUL R0, R2, R4   F  D  s  s  s  s  s  s E3 E3  W
SUB R8, R2, R6   F  D  s  s  s  s  s  s E2  W
DIV R10, R0, R6  F  r  r  r  r  r  r  r  r  r  D  s E3 E3 E3 E3  W
ADD R6, R8, R2   F  r  r  r  r  r  r  r  r  D  s E2  W
```

Cycles in which an instruction is waiting for a reservation station are marked as 'r' and the cycles in which an instruction is waiting for one or more operands are marked as 's'. As seen in the time chart, the issue and write back cycles for various instructions are:

instruction	**issue cycle**	**write back cycle**
LOAD	1	4
LOAD	4	7
MUL	1	10
SUB	1	9
DIV	10	16
ADD	9	12

5.6 1. See book pages 82-83 and Figure 2.4. For our example program we would have one entry corresponding to the BNE instruction in the end of the program. The prediction would evolve according to this table if we assume we start in state 00:

Execution of the BNE instuction	**Prediction state before execution**	**Prediction**	**State after execution**
First	00	Not taken (wrong)	01 (it was taken)
Second	01	Not taken (wrong)	11 (it was taken)
Third	11	Taken (correct)	11 (it was taken)
4th, 5th, ...	11	Taken (correct)	11 (it was taken)
1000th	11	Taken (wrong)	10 (was not taken)

For a program with two branches, we would get a 2-bit state entry for each branch as long as they do not share the same index in the branch-prediction buffer. Branches with different index (entries) are not correlated with each other.

2. The key point for supporting speculation is to commit architectural visible changes in program order. It must be possible to cancel (flush) speculative results without any remaining visible changes in the registers or in the memory. Registers and memory should only be written in the commit stage. Before that, the reorder buffer holds the speculative (temporary) results.

3. The corresponding steps for a store instruction is:

   (a) Issue when reservation station and ROB entry is available
   - Read already available operands from registers and instruction
   - Tag unavailable operands with ROB entry
   - Mark ROB entry as busy

   (b) Execute after issue
   - Wait for operand values on CDB (if not already available)
   - Compute address and store it in ROB entry

   (c) Write result when CDB and ROB available
   - Update ROB entry with source register value, and mark as ready
   - Free reservation station

   (d) Commit when at head of ROB and ready
   - Write result (source register value) to memory at computed address
   - Free ROB entry

   The important points are: (1) The actual write is done at the commit stage, (2) To carry out the write, the address needs to be computed (execute stage), and, (3) the source operand (the value to write) is needed before commit is possible.

4. RAW hazards are avoided by delaying instruction execution until the source operands are available. Instructions wait in a reservation station until source operands are available. Earlier instructions that write dependent values send their results directly to the waiting reservation stations.

5.7 The advantage is a reduced hit time since the indexing in the cache does not need to wait for the virtual to physical address translation to finish. Disadvantages: (1) Page-level protection needs to be enforced even when no translation and lookup in the TLB is done, (2) The translation is process dependent. When processes are switched, the cache must maybe be flushed. (3) Alias problems might cause same data to appear in different cache locations leading to inconsistencies.

7.6 Memory systems, Cache I

6.1 In real life bigger memory is slower and faster memory is more expensive. We want to simultaneously increase the speed and decrease the cost. Speed is important because the widening performance gap between CPU and memory. Size is important since applications and data sets are growing bigger. Use several types of memory with varying speeds arranged in a hierarchy that is optimized with respect to the use of memory. Mapping functions provide address translations between levels.

Registers: internal ultra-fast memory for CPU; static register

Cache: speed up memory access; SRAM

Main memory: DRAM

VM: make memory larger, disk; safe sharing between processes of physical memory, protection, relocation

(archival storage, backup on tape)

6.2 a) Block transfer time between cache and memory: 40+32/4 = 48 cycles.
Number of block transfers per instruction between cache and memory:
(1+0.2)*(1+0.5) * Miss ratio = 1.8 * Miss ratio
CPI 1 = 1.5 +1.8*48*0.029 = 4.01
CPI 2 =1.5 +1.8*48*0.022 = 3.40
CPI 3 =1.5 +1.8*48*0.020 = 3.22

b) We must do a TLB access for each cache access since the caches are physically addressed. We then in all three cases get an extra CPI offset of: 0.002*20*1.2 = 0.048.

c) Comparing execution times using CPU performance formula:
EXE 1 = 4.01 * 1 * CP * IC = 4.01 * CP * IC
EXE 1 = 3.40 * 1.2 * CP * IC = 4.08 * CP * IC
EXE 1 = 3.22 * 1.25 * CP * IC = 4.025 * CP * IC
Cache 1 is the best.

d) Similar to fig C.5, page C-13 in Hennessy & Patterson.

e) In a virtually addressed cache the TLB is only accessed at cache misses. In a physically addressed cache TLB is accessed for each cache access.

6.3 A large register file and a large data cache both serve the purpose of reducing memory traffic. From an implementation point of view, the same chip area can be used for either a large register file or a large data cache. With a larger register set, the instruction set must be changed so that it can address more register in the instructions. From a programming point of view, registers can be manipulated by program code, but the data cache is transparent to the user. In fact, the data cache is primarily involved in load/store operations. The addressing of a cache involves address translation and is more complicated than that of a register file.

6.4 With a unified cache, the fraction of the cache devoted to instructions and data respectively may change from one program to another. This can be a benefit but also a problem for, e.g., a small loop that touches a lot of data. In a unified cache, instructions will probably be replaced by data as the loop executes. With two separate caches, the entire loop can fit in cache and performance will probably be improved. The design trade-offs involve choosing the correct division between instruction and data caches for separate cache memories, studying the miss rate for a unified cache, choosing a correct address mapping for a unified cache and a replacement policy.

6.5 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.

6.6 See 'Case Study Solutions' at http://www.elsevierdirect.com/companion.jsp?ISBN=9780123704900

6.7 This exercise uses differences in cache organizations to illustrate how benchmarks can present a skewed perspective of system performance. Because the memory hierarchy heavily influences system performance, it is possible to develop code that runs poorly on a particular cache organization. This exercise should drive home not only an appreciation for the influence of cache organization on performance, but also an appreciation of how difficult it is for a single program to provide a reasonable summary of general system performance.

- Consider the MIPS code blurb in Figure 7. We make two assumptions in this code: First, the value of r0 is zero; second, locations f00 and bar both map onto the same set in both caches. For example, f00 and bar could be 0x00000000 and 0x80000000 (these addresses are in hexadecimal), respectively, since both addresses reside in the set zero of either cache. On Cache A, this code only causes two compulsory misses to load the two instructions into the cache. After that, all accesses generated by the code hit the cache. For Cache B, all the accesses miss the cache because a direct-mapped cache can only store one block in each set, yet the program has two active blocks that map to the same set. The cache will "thrash" because when it generates an access to f00, the block containing bar is resident in the cache, and when it generates an access to bar, the block containing f00 is resident in the cache. This is a good example of where a victim cache could eliminate the performance benefit of the associative cache. Keep in mind that in this example the information that Cache B misses on is always recently resident.

```
F00:      beqz    r0, bar  : branch iff r0 == 0
                  .
                  .
                  .
bar:      beqz    r0, f00  ; branch iff r0 == 0
```

Figure 7: MIPS code that performs better on cache A.

With all access hits, Cache A allow the processor to maintain CPI = 1. Cache B misses each access at cost of 100 ns, or 200 ns clock cycles. This Cache B allows its processor to achieve CPI = 200. Cache A offers a speed-up of 200 over Cache B.

- Consider the code blurb shown in Figure 8. We make two assumptions: first, locations baz and qux and the location pointed to by 0(r1) map to different sets within the caches and are all initially resident; second, r0 is zero (as it always is for MIPS) This code illustrates the main thrust of a program that makes a system with Cache B outperform a system with Cache A, that is, one, that repeatedly writes to a location that is resident in both caches. Each time the sw executes on Cache A, the data stored are written to memory because Cache A is write through. For Cache B, the sw always finds the appropriate block in the cache (we assume the data at location 0(r1) are resident in the cache) and updates only the cache block, as the cache is write back; the block is not written to main memory until it is replaced.

```
Baz:      sw       0(r1), r0 ; store r0 to memory
Qux:      beqz     r0, baz   ;branch iff r0 == 0
```

Figure 8: MIPS code that performs better on cache B.

In the steady state, Cache B hits on every write, and, so, maintain CPI = 1. Cache A writes to memory on each store, consuming an extra 100 ns each time. Cache B allows the processor to complete one iteration in 2 clocks. With Cache A the processor needs 202 clocks per iteration. Cache B offers a speedup of 1012 over Cache A.

6.8 Since a processor is clocked in discreet steps it's enough to count the extra cost you get on a miss. Here we have stated the miss cost as the number of clock cycles.

a) Miss cost $t_m(B) = M(B)/100 * (1 + 4 + B/4)$ which gives us
$t_m(32) = 0.016 * (5 + 32/4) = 0.2$ as the smallest value $\Rightarrow B_{opt} = 32$

b) Miss cost $t_m(B) = M(B)/100 * (2 + 1 + 4 + B/4)$ which gives us
$t_m(64) = 0.010 * (2 + 1 + 64/4) = 0.23$ as the smallest value $\Rightarrow B_{opt} = 64$

c) I: Miss cost $t_m(B) = M(B)/100 * (1 + 4 + \lceil B/8 \rceil)$ which gives us
$t_m(64) = 0.010 * (1 + 4 + 64/8) = 0.13$ as the smallest value $\Rightarrow B_{opt} = 64$
II: Miss cost $t_m(B) = M(B)/100 * (2 + 1 + 4 + \lceil B/8 \rceil)$ which gives us
$t_m(128) = 0.0064 * (2 + 1 + 4 + 128/8) = 0.1472$ as the smallest value $\Rightarrow B_{opt} = 128$

d) 16 byte in a block means that 4 bit in the byte offset field is needed. There are 256/16 = 16 blocks in the cache memory. This gives us that 4 bits are needed for the index field.

$$Block\_address = \lfloor address/block\_size \rfloor = \lfloor 28E7_{16}/10_{16} \rfloor = \lfloor 10471/16 \rfloor = 654$$

index = block address mod (number of blocks) = 654 mod 16 =14
Address $28E7_{16}$ is represented by block 14 in the cache memory.

e) The block address is the same. We now have 8 sets (2 blocks in every set) which means that 3 bits is needed for the index field. The address is represented in
$set = 654 mod 8 = 6$. This set includes block 2*6 and 2*6 + 1.

6.9 a) Modern computer systems needs large memories that at the same time isn't allowed to cost too much. Large memories are unfortunately also very expensive if they also shall be fast. The technique with memory hierarchy solves this problem by having a series of memories with small fast memories closest to the processor and gradually bigger and slower memories the further away from the processor you get. The level furthest away from the processor is big enough to have space for the total memory space you wish to address. The memory closest to the processor only have room for a fraction of the total addressable memory space. When the processor performs a memory reference (fetching of instructions or reading/ writing data) the first level in the memory hierarchy is checked to see if the wanted data is there. If is, then the memory referents is performed there. If it isn't there then the next level is checked and so forth. When you find what you are looking for in a memory hierarchy a block of data (or instructions) that contain this memory position will be moved to a higher level of the hierarchy. The size of the block you moves get smaller the higher up in the memory hierarchy you get.

The point of memory hierarchy is to have the data /instructions that the processor most often needs in the fast memory closest to the processor. You can afford fetching things that isn't used as often from the slower and bigger memory. The reason that it work so well is that computer programs often exhibits the principle of locality: Temporal locality: What has been used by the processor will with an high degree of likelihood be used again. This property leads to that it's enough that a higher lever of the memory hierarchy only contain a small part of the lower ones. Spatial locality: What is close (in the address space) to something the processor has accessed will with high degree of likelihood also be needed. By moving a block of data/instructions around what you need at a miss you don't have to fetch as many times.

b) The cache has 16 set which means that the index field in the address is four bits. The tag field is 9 bits and whats left for the byte offset is 16-9-4=3. This means that the block-size 23 = 8 byte and the number of blocks is 256/8 = 32. There are two blocks in each set which gives us that the cache is two set associative. Address 3333_{16} have the tag = 33, index = 6 and byte offset = 3. This give us that the data for this address will be placed in set 6 and byte 3.

c) Write-through and Copy-back: Write-through: All the writes are also sent to the main memory even if they do hit in the cache. This means that with a block replacement you don't have to update the main memory at the cost of slower writes or more complicated hardware. Copy-back: Writes are only performed locally in the cache. For every block you then will also need a dirty bit that tells if the block in the cache has been written to. On a block change the block also needs to be written back to the main memory if the dirty bit is true.