

Question Q1–1: (Solution, p 2) Distinguish between a load-store instruction set (such as x86) and a memory-register instruction set (such as MIPS). What are their relative performance advantages in today’s technology?

Question Q1–2: (Solution, p 2) Write a MIPS assembly language program that, given n as a parameter, computes the lower 32 bits of $n!$ using recursion.

Question Q1–3: (Solution, p 2) Explain the purpose underlying the register \$at, which assembly programs should not utilize.

Question Q1–4: (Solution, p 2) The MIPS calling convention includes some callee-save registers, such as \$s0 and \$s1, and some caller-save registers, such as \$t0 and \$t1. Explain why the inclusion of both types of registers in the calling convention improves program performance.

Question Q1–5: (Solution, p 3) Explain why modern processors do not generally use the simple single-cycle implementation, even though this implementation provides the fastest possible completion of each instruction and requires less hardware than other techniques.

Question Q1–6: (Solution, p 3) Explain what *control hazards* are in the context of pipelining, and describe two techniques to reduce their impact.

Question Q1–7: (Solution, p 3) In a dynamic multiple-issue architecture, the processor engage in register renaming to reduce the number of apparent conflicts between instructions. Rewrite the MIPS code below to reflect how it would look after renaming; in renaming, use the register values \$f0 to \$f15. The \$zero register should not be renamed.

```
addi    $t0, $zero, 0
addi    $t1, $zero, 4
add     $t0, $t0, $t1
addi    $t1, $t1, -1
add     $t0, $t0, $t1
addi    $t1, $t1, -1
add     $t0, $t0, $t1
addi    $t1, $t1, -1
add     $t0, $t0, $t1
```

Solution Q1–1: (Question, p 1) A load-store instruction set has many instructions that combine a memory access with another operation, such as an add or branch. By contrast, the only instructions in a memory-register instruction set that access memory are the load and store instructions; other operations can operate only on registers and immediates.

Load-store instructions sets are advantageous in situations where memory speed is fast relative to the processor. But with today’s technology, memory is much slower, so restricting memory access to specialized instructions allows the other operations to complete more quickly.

Solution Q1–2: (Question, p 1)

```
fact:      addi   $sp, $sp, -8
           sw    $a0, 0($sp)
           sw    $ra, 4($sp)
           addi  $v0, $zero, 1
           beq   $a0, $zero, done
           addi  $a0, $a0, -1
           jal   fact
           lw    $a0, 0($sp)
           mul   $v0, $a0
           mfhi  $v0
done:      lw    $ra, 4($sp)
           addi  $sp, $sp, 8
           jr    $ra
```

Solution Q1–3: (Question, p 1) Most assemblers have *pseudo-operations*, which are instructions that appear to be a native instruction, but which really translate to a different instruction — or several different instructions. The register \$at is reserved for temporary data needed for such instructions. For example, assemblers might include a **bgt** pseudo-operation for branching if one number is greater than another.

```
bgt    $t1, $t0, is_greater
```

There is no single instruction that does this, and the only reasonable instruction involves computing a temporary value using the **slt** instruction

```
slt    $at, $t0, $t1
bne    $at, $zero, is_greater
```

Solution Q1–4: (Question, p 1) Callee-save registers allow a subroutine to use a register to remember a value across other subroutine calls, without having to push and pop the stack surrounding each subroutine call. (The callee-save register would be only pushed and popped once — at the beginning and end of the subroutine.) Caller-save registers allow a subroutine to employ registers for short-term computations without an intervening subroutine call, without having to worry about restoring the register’s previous value. Since both situations show up in typical subroutines, having both categories of registers allows a compiler to minimize the overall usage of the stack.

Solution Q1–5: (Question, p 1) While it completes instructions quickly individually, it is only working on one instruction at a time, and so the overall throughput is less; a pipelined architecture increases latency but increases throughput, and throughput is what really governs program performance.

The advantage of requiring less transistors is a real advantage, but a very minor one: With today's technology, the number of transistors that can cheaply be placed on a chip far exceeds the number of transistors required for a single-cycle implementation.

Solution Q1–6: (Question, p 1) Control hazards are situations that lead to losing maximum pipeline usage due to not knowing the precise future sequence of instructions, usually due to a branch instruction (but also possible due to an indirect jump). Their impact is reduced by any of the following.

- *Speculative execution* has the processor guess whether a branch will be taken (or not taken), so that it can immediately begin executing succeeding instructions accordingly. If the guess is wrong, the succeeding instructions can promptly be canceled; but if the guess is correct, the pipeline will be fully utilized throughout the branch.
- Sophisticated *branch prediction* will improve the performance of speculative execution further, by ensuring that the guesses are more often correct.
- Making the decision of whether to branch earlier in the pipeline will reduce the overall penalty for an incorrect prediction.
- A *delay slot*, where the instruction after each branch will be executed regardless of the branch decision, allows at least one instruction after each branch to not be wasted.

Solution Q1–7: (Question, p 1)

```
addi  $f0, $zero, 0
addi  $f1, $zero, 4
add   $f2, $f0, $f1
addi  $f3, $f1, -1
add   $f4, $f2, $f3
addi  $f5, $f3, -1
add   $f6, $f4, $f5
addi  $f7, $f5, -1
add   $f8, $f6, $f7
```