# Digital Hardware
# Problems 3

Mike Spivey, Hilary Term 2004

**1** [H&P ex. 3.1, modified] Add comments to the following MIPS code, and describe in one sentence what it computes. Assume the input, a positive integer $n$, is in register `$2`, and the output appears in register `$3`.

```
begin:
   addi $4, $0, 0
   addi $5, $0, 1
loop:
   slt $6, $2, $5
   bne $6, $0, finish
   add $4, $4, $5
   addi $5, $5, 2
   br loop
finish:
   add $3, $4, 0
```

**2** (i) Write a program in Oberon or C that, given two numbers $a \geq 0$ and $b > 0$, computes $q$ and $r$ such that

$$a = q \times b + r \quad \text{and} \quad 0 \leq r < b.$$

Use the simplest algorithm you can.

(ii) Rewrite your program in MIPS assembly language, keeping all variables in registers.

**3** (i) Write a program in Oberon or C that, given an array $a[0\,..\,n)$ of non-negative 32-bit integers, finds the maximum value in $a$.

(ii) Rewrite your program in MIPS assembly language, keeping all variables except the array $a$ in registers, and assuming that the address of $a$ fits in 16 bits.

**4** [H&P exx. 5.1–2, modified] A common fault in chip manufacture is that signals become stuck at logic 0 or logic 1. For each of the following signals in the processor design that is shown in Figure 5.19 of the book, describe the effect if the signal suffers either of these faults. Which instructions would still work correctly?

(i) *RegDst*,

(ii) *ALUSrc*,

(iii) *MemToReg*,

(iv) *Zero*.

**5** [H&P ex. 3.10, modified] The MIPS has a restricted set of instructions that must be combined to carry out common tasks. MIPS assemblers make the programmer's task easier by providing a richer set of macro-instructions, and replacing them by short sequences of real instructions as the program is translated.

For this purpose, programmers agree not to keep any value in register $1, so that the assembler can use it to hold a temporary value. The processor provides an instruction

lui $*rt*, *imm*

that loads the 16-bit quantity *imm* into the *upper* half of register $*rt*, setting the lower half to zero. Thus

```
lui $2, 0x1234
```

set `$2` to the value `0x12340000`.

Using this instruction, and others such as `slt`, show how to implement the following macro-instructions. Here *big* refers to a specific constant that needs 32 bits, and *small* refers to a specific constant that fits in 16 bits.

| *Instruction* | *Effect* |
|---|---|
| move $5, $3 | $\$5 := \$3$ |
| clear $5 | $\$5 := 0$ |
| li $5, *small* | $\$5 := small$ |
| li $5, *big* | $\$5 := big$ |
| lw $5, *big*($3) | $\$5 := mem_4[\$3 + big]$ |
| addi $5, $3, *big* | $\$5 := \$3 + big$ |
| beq $5, *small*, L | branch if $\$5 = small$ |
| beq $5, *big*, L | branch if $\$5 = big$ |
| ble $5, $3, L | branch if $\$5 \leq \$3$ |
| bgt $5, $3, L | branch if $\$5 > \$3$ |
| bge $5, $3, L | branch if $\$5 \geq \$3$ |

**6** The MIPS has two instructions `slt` and `sltu` that set a result register to either 1 or 0 depending in a comparison of their two input registers $x$ and $y$. The `slt` instruction interprets $x$ and $y$ as twos-complement signed numbers and sets its result to 1 if $x < y$. The `sltu` instruction is similar, but interprets $x$ and $y$ as unsigned numbers.

(i) Give an example to show that `slt` and `sltu` must be implemented differently.

(ii) Give an example to show that `slt` cannot be implemented by performing a 32-bit subtraction and taking the sign bit of the result.

(iii) Show a circuit that does give the correct result for `slt`.

(iv) Design additional logic that also computes the correct result for `sltu`.