INDIAN INSTITUTE OF TECHNOLOGY BOMBAY
ELECTRICAL ENGINEERING DEPARTMENT

REPLACEMENT TEST-2

Saturday	**EE 309: Microprocessors**	Time: 1700-1800
Apr. 17, 2021	Spring Semester 2020-21	Marks: 10

Marks will be scaled depending on the tests being replaced.
Code should be clear and well commented.

Q–1 A fixed program microprocessor carries out the following operations repeatedly:
I) Fetch an instruction from ROM,
II) Decode it,
III) Fetch an operand from data memory and another from a register in parallel,
IV) execute the instruction and
V) Write the result to data memory.

Every read from ROM or read/write operation from/to data memory takes 3 clock cycles, while decoding, execution and on-chip register read/write takes a clock cycle each.

We want to assess the effect of placing an on-chip instruction cache. The probability of finding the next instruction in the cache is P and fetching it from the cache takes just one clock cycle.

Evaluate the expected speed up factor for the processor with cache as a function of P. Find the limiting values of speed up as P tends to zero and as P tends to 1.

Soln. 1) Without the instruction cache, the processor will take
3 (instr. fetch) + 1 (decode) + 3 (operand fetch) + 1 execute + 3 (write) = 11 cycles.
With the cache, instruction fetch will be done in 1 cycle with probability P and in 3 cycles with probability 1-P.
Thus average time for completing the instruction is: 3 (1-P) + 1 P + 8 = 11-2P cycles.
Therefore speed up is 11/(11-2P)
for P = 0 this is 1, as expected. For P = 1, it is 11/9. – **[2]**

Q–2 **a)** What is the advantage of using segmented addresses as is done by the 8086 processor? Describe the different models used by 8086 based programs with respect to the size of their code and data areas. How does the compiler/assembler use the information about the model while compiling a program?

Soln. 2-a) Most code and data accesses in a program are local. This means that the most significant bits of memory addresses change rarely. Segmented addresses essentially separate the most significant bits of the address from the less significant bits, storing these in different (and smaller) registers. Now the more significant bit registers are rarely modified and most of the time, we store and manipulate only the much smaller offset addresses.
In case of 8086, the segment registers (more significant bits) are shifted four positions to the left and added to the effective address to get the physical addresses sent to the memory. Both are 16 bits wide. Together these span a 20 bit address

space. A region of memory which is 64KB in size can be accessed without changing the segment register. This makes programs faster and smaller in size for storage.

Obviously, if code or data size is $<$ 64KB, we can initialize the corresponding segment registers once and for all and never bother about these during program execution. Accordingly, the following models are defined:

Small: Here data is $<$ 64 KB in size, as well as the code is $<$ 64 KB. Here all segment registers can be initialized once and for all.

Compact: Here Code is $<$ 64 KB in size, but data is larger. Here, the code segment registers need not be touched during the execution. Data segment registers will have to be modified during operation.

Medium: Here code is $>$ 64 KB in size, data is $<$ 64 KB. Here the data segment registers are kept constant, while code addresses need to specify both the segment and offset.

Large: Here both code and data are $>$ 64 KB in size. We need to use the full segment+offset specification for all memory accesses.

Tiny: This is a special case of small model where the entire code, data and stack segment together are $<$ 64 KB in size. Here, not only are all segment registers constant, but all of them have the same value.

Huge: This is a special case of large model. Here not only is the data segment $>$ 64 KB in size, but individual data items like arrays can be $>$ 64 KB. In this case, the segment register may have to be manipulated even while accessing elements of the same data array.

Compilers make use of the model declared while developing a program. For example in small and compact models, return addresses and direct jump addresses need not be included in the instructions and data structures. – **[2]**

b) 8086 uses a full descending stack. It takes more clock cycles to push a general purpose register than to pop it. What can be the reason for this imbalance, assuming that memory read and write times are identical?

Soln. 2-b) The stack pointer points to the last placed item on the stack in a full stack. Thus, when a push operation is carried out, the stack pointer has to be advanced first (decremented for a full descending stack) before data can be copied to this address. Hence the memory operation cannot begin until the update of the stack pointer is complete.

In case of a pop operation, data is fetched from the address in stack pointer which is adjusted (incremented in case of a full descending stack) *after* the memory access. So in this case, the memory access can be started right away and after the address has been issued on the memory bus, incrementing of stack pointer can take place in parallel with memory read.

For this reason, push is slower than pop for an 8086. – **[1]**

c) Write an 8086 assembly program using string instructions to find the length of a string terminated by a `'\0'` as used in C. You can assume that the maximum length of the string is 130 characters.

Soln. 2-c) We assume that ES:DI is pointing to the beginning of the string

```
StrLen:  SUB    AL, AL      ; Make AL = 0, looking for a NUL in string
         MOV    CX, 0xFFFF  ; Initialize CX to -1
         CLD                ; Increment string pointer as we repeat
         REPNE  SCASB       ; Repeat while string element is unequal to AL
         NOT    CX          ; To change sign of CX
         DEC    CX          ; Adjust for 2's complement
```

We have initialized CX to -1. It will be further decremented for every iteration of REP. If the string had zero length, CX would be decremented once to -2. If the string has n non-zero characters before the terminating NUL character, CX will contain -(n+2) at the end. To change its sign, we complement it and add 1 which will give n+2. Then to remove the +2, we need to subtract 2. Adding one and subtracting two is equivalent to a single decrement. This is done by NOT CX followed by DEC CX after REPNE terminates.

Could we not have initialized CX to 0, so that we don't need to decrement it at the end? Unfortunately, no!
If CX is initialized to zero, the REP loop will terminate right away, irrespective of match with AL. – **[3]**

Q–3 MIPS provides a minimal set of natively implemented instructions. The assembler can construct instructions and addressing modes not provided by the processor. A register ($at) is reserved for the use of the assembler for this purpose.

Show how the assembler constructs an instruction to load a 32 bit immediate value into a register using multiple native instructions.
Also, show how it can construct the based indexed addressing mode where the effective address is the sum of two registers and an immediate constant.

Soln. 3) Let the 32 bit immediate value by HL where H is the upper half-word and L is the lower half-word.
Then we can construct the pseudo instruction
li $t0, HL #(Load 32bit immediate constant HL into $t0) as:

```
lui $t0, H # Load upper half-word of $t0 with H, zero the lower half-word
ori $t0, L # Insert L into the lower half-word (no sign extension)
```

To construct the based indexed addressing mode, let us say we want to load a word into $t0 from the address which is the sum of $t1, $t2 and a 16bit signed offset X. This can be done with:

```
addu $at, $t1, $t2 # Add $t1 and $t2, store sum in $at
lw $t0, X($at) # Load a word into $t0 from $at + sign extended X
```

– **[2]**