EE316 Embedded Systems & Applications

Major Test

Marks: 60 (Open Book) Time: 3 hrs

---

1. A synchronous parallel bus *input transfer* between a master and slave works by
   - Setting address buses in the positive half clock cycle starting at the rising edge,
   - Setting data buses in the following negative half clock cycle starting at the following edge, and
   - Reset of both buses at the rising edge that ends the negative half clock cycle

   Going to details, the bus clock is known to have 50% duty cycle:
   - The master side needs 1.5ns to set the address bus and command line,
   - The slave side needs 3ns to decode the address.
   - The slave side sets needs a maximum of 5ns to set up data on the bus.
   - The master side needs 1ns to set up and latch the input registers.

   If propagation delay on any of the wired connections is 4ns, what is the maximum possible bus clock frequency that the system may work with ?

   ...(10)

2. ARM11xxxx processors are SIMD (Single-Instruction-Multiple-Data) extensions of the ARM6 specifically intended for DSP applications. The family is typically provided block set associative (BSA) cache of configurable size.

   Consider a core that maps a 1GB memory space (say, the user space) on to a 4kB BSA cache with 4 blocks per set and 64 bytes per block.

   a. Calculate the number of bits in Tag, Set, and Word fields of memory address.
   b. Assume that the cache is initially empty. Suppose that the processor fetches 1088 words of four bytes each from successive word locations starting at location 0. It then repeats this fetch sequence nine more times. If the cache is 10 times faster than the memory, estimate the improvement factor resulting from the use of the cache.

      Assume that the LRU algorithm is used for block replacement.

   ...(5+10)

3. A subroutine `_ADVEC` is to be written in TMS320C31 assembly language that adds two floating point vectors **A** and **B** of length *N* each, saving the result in vector **C**. The vectors begin at locations `_A`, `_B`, and `_C`, respectively, and the value of *N* is stored in a location `_N` – all in the `.bss` segment. Write a code for this subroutine, if it is to return when the summation is over.

   ...(15)

4. Write the Intel 80x source code for a subroutine that solves the discrete time state equation of arbitrary order *N* and with input vector of order *M* (the integers *N* and *M* will be fed to the subroutine through memory locations `_N` and `_M`). The initial state vector passed into the function as a string starting at `_x` in the data segment , should be replaced by the final state vector when the function returns. The input at any call of the subroutine is stored in the data segment starting `_u`.

   Finally, remember that the function must be general purpose, that is the same function should be able to accept different **A** and **B** matrices and compute the concerned **x** before it returns. (You may however assume that none of the vectors or matrices will require `far` access, that is, a change of segment.)

   ...(20)

EE316 Embedded Systems & Applications

Solutions to Major Test

1. The minimum time required during the positive half clock cycle is 1.5 + 4 + 3 = 8.5ns.
The minimum time required by the negative half clock cycle is 5 + 4 + 1 = 10ns.

   Half the time period of the bus clock therefore needs to be at least 10ns, which implies a 20ns clock period. The maximum bus clock frequency can therefore be 50MHz.

   (10 marks)

2.

   a. A block has 64 bytes; hence the Word field is 6 bits long. With 4 x 64 = 256 bytes in a set, there are 4K/256 = 16 sets, requiring a Set field of 4 bits. This leaves 32 – 4 – 6 = 22 bits for the Tag field.

   b. The 1088 words constitute 68 blocks, occupying blocks 0 to 67 in the memory.

      The cache has space for 64 blocks. Hence, after blocks 0, 1, 2, . . . , 63 have been read from the memory into the cache on the first pass, the cache is full. The next four blocks, numbered 64 to 67, map to sets 0, 1, 2, and 3. Each of them will replace the least recently used cache block in its set, which is block 0.

      During the second pass, memory block 0 has to be reloaded into set 0 of the cache, since it has been overwritten by block 64. It will be placed in the least recently used block of set 0 at that point, which is block 1. Next, memory blocks 1, 2, and 3 will replace block 1 of sets 1, 2 and 3 in the cache, respectively. Memory blocks 4 to 15 will be found in the cache. Memory blocks 16 to 19, which were in block location 1 of sets 0 to 3, have now been overwritten, and will be reloaded in block location 2 of these sets.

      As execution proceeds, all memory blocks that occupy the first four of the 16 cache sets are always overwritten before they can be used on a succeeding pass. Memory blocks 0, 16, 32, 48, and 64 continually displace each other as they compete for the 4 block positions in cache set 0. The same thing occurs in cache set 1 (memory blocks 1, 17, 33, 49, 65), cache set 2 (memory blocks 2, 18, 34, 50, 66), and cache set 3 (memory blocks 3, 19, 35, 51, 67). Memory blocks that occupy the last 12 sets (sets 4 through 15) are fetched once on the first pass and remain in the cache for the next 9 passes.

      In summary, on the first pass, all 68 blocks of the loop are fetched from the memory. On each of the 9 successive passes, 48 blocks are found in sets 4 through 15 of the cache, and the remaining 20 blocks must be fetched from the memory. Let τ be the access time of the cache. Therefore,

      Improvement factor = (Time without cache) / (Time with cache)
      $= (10 \times 68 \times 10\tau) / (1 \times 68 \times 11\tau + 9(20 \times 11\tau + 48\tau)$
      $= 2.15$

      (5+10 marks)

3.

```
_ADVEC:  PUSH  AR2              ;     Save registers to be used as
         PUSH  AR3              ;                        pointers.
         PUSH  AR4
         PUSH  AR5              ;     Save register for counter.

         PUSH  R0               ;     Save EPR required.
         PUSHF R0

         LDI   @_N, AR5         ;     Load vector size in counter.

         LDI   @_A, AR2         ;     Load pointers from .bss.
```

```
          LDI   @ B, AR3
          LDI   @_C, AR4

SUM:      ADDF3 *AR2, *AR3, R0    ;     Add vector elements.
          ADDI  2, AR2            ;     Increment pointers by 2
          ADDI  2, AR3
          STF   R0, *AR4++        ;     Save vector sum element
          STI   R0, *AR4++

          DBNZ  AR5, SUM          ;     Dec counter and loop if NZ.

          POPF  R0                ;     Restore EPR used.
          POP   R0

          POP   AR5               ;     Restore AR's
          POP   AR4
          POP   AR3
          POP   AR2

          RETS                    ;     Return.
```

(15 marks)

4.

```
_ST_EQN:
             PUSH     AX
             PUSH     DS              ; Save relevant registers
             PUSH     ES
             PUSH     BP
             PUSH     SI
             PUSH     DI

AxX:
             LEA      DS, _A          ; Set ptr to first row of A
             LEA      ES, _x          ; Set ptr to x
             LEA      BP, PPROD       ; Set ptr to first elem of PPROD.
             MOV      SI, 0
             MOV      DI, 0           ; Initialise index registers.
             MOV      [BP+DI], 0      ; Initialise first elem of PPROD.
AxX_LOOP:
             MOV      AL, [DS+SI]     ; A(i, j)
             IMUL     [ES+SI]         ; A(i, j) * x(j)
             ADD      [BP+DI], AX     ; x(i) += A(i, j) * x(j)
             INC      SI              ; j++
             CMP      SI, _N
             JNE      ROW_LOOP        ; j<N ?

             ADD      DS, _N          ; Set ptr to next row of A.
             INC      DI              ; Set ptr to next elem of PPROD.
             CMP      DI, _N          ; i<N ?
             JNE      AxX_LOOP

BxU:
             LEA      DS, _B          ; Set ptr to first row of B.
             LEA      ES, _u          ; Set ptr to u.
             LEA      BP, PPROD       ; Set ptr to first elem of PPROD.
             MOV      SI, 0
             MOV      DI, 0           ; Initialise index registers.
BxU_LOOP:
             MOV      AL, [DS+SI]     ; B(i, j)
             IMUL     [ES+SI]         ; B(i, j) * u(j)
             ADD      [BP+DI], AX     ; x() += B(i, j) * u(j)
             INC      SI              ; j++
             CMP      SI, _M
             JNE      BxU_LOOP        ; j<M ?

             ADD      DS, _M          ; Set ptr to next row of B.
             INC      DI              ; Set ptr to next elem of PPROD.
             CMP      DI, _N          ; i<N ?
```

```
            JNE     BxU LOOP

SAVE_x:
            LEA     ES, _x              ; Set ptr to x.
            LEA     BP, PPROD           ; Set ptr to PPROD.
            MOV     DI, 0
SAVE_LOOP:
            MOV     [ES+DI], [BP+DI]    ; PPROD(i) -> x(i)
            INC     DI
            CMP     DI, _N
            JNE     SAVE_LOOP           ; Update x.

RESTORE:
            POP     DI                  ; Restore relevant registers.
            POP     SI
            POP     BP
            POP     ES
            POP     DS
            POP     AX
            RET                         ; Return.
```

(20 marks)