Telecommunications Exam Solutions

- Two hour Exam
- Answer 2 of 3 Questions
- 1 hour per question

2017 Exam:

Q1 - Memory & Pipelining

a) Explain the terms RAM, ROM, Flash, Dynamic RAM, Static RAM. What kinds of memory does the LPC2138 have?

- RAM: Random Access Memory – A type of memory that can be accessed randomly, that is any byte of memory can be accessed without touching the preceding bytes. RAM is used by the CPU to store memory that needs to be used/accessed quickly.

- ROM: Read Only Memory – A type of memory that can once it is written it can only be read from. Unlike main memory RAM, ROM contains information even when the PC is off. Computers tend to have a small amount of ROM which mainly stores critical programs such as boot sequences.

- Flash: Flash Memory is an electronic non-volatile computer storage medium that can be electrically erased and re-programmed. Flash memory retains the data even in the absence of power supply.

- Dynaic RAM: Dynamic RAM is a type of random-access memory that stores each bit of data within a separate capacitor within an integrated circuit. The capacitor can either be charged/discharged representing a 1 or a 0. It is called Dynamic as the capacitors need to be refreshed every few milliseconds to account for charge leaks.

- Static RAM: Static RAM is a type of random-access memory that retains data bits in its memory as long as power to the system is being supplied. Unlike DRAM, SRAM does not have to be continuously refreshed.

- LPC2138 Memory: The LPC2138 board contains Flash Memory, Static RAM, Non-Volatile On-Chip Memory.

b) What does Memory Mapped Input/Output mean?

Memory Mapped Input/Output is when a device, such as the LPC2138 contains peripheral devices which can be used to handle user input or output to/from external devices.

These devices are mapped to a respective area in memory on the board that is then written to/ read from in order to access these specific peripheral devices. Reading a specific bit sequence at a given address might give you information regarding the current state of a peripheral device e.g an LED. Whilst writing to a specific address in memory might change the state of a peripheral device such as an LED or a buzzer.

c) Give an account of the purpose, organisation and operation of a typical cache. Explain how memory locations might be mapped to it and explain how it flushes entries.

Main memory is very slow by comparison with instruction execution. It is extremely expensive and time consuming to read/write to/from memory and this has direct impacts with the performance time of a system.

As a result Cache sits logically between main memory and the CPU. It serves the purpose of implementing a middle ground between the CPU and main memory and allows for extremely fast memory accesses that can be used in a program.

There can be different levels of cache:

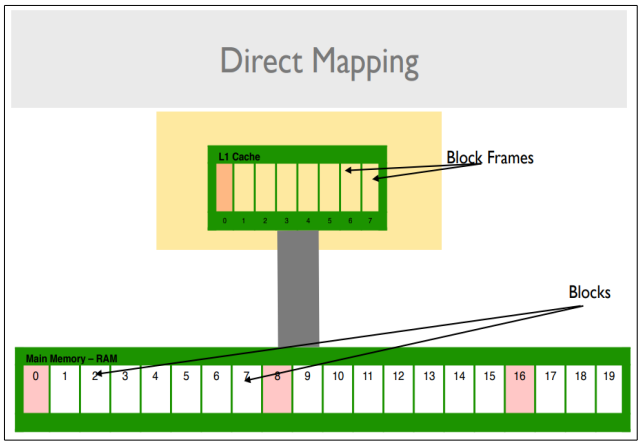
1. L1 – On Chip, Closest to the CPU, Very Fast, Very Small
2. L2 – On Chip, between CPU and Bus
3. L3 – Off Chip, between chip and main memory

Typically the L1 cache is divided into an Instruction Cache (I-Cache) and Data Cache (D-Cache)

Caches are un-named and un-numbered memory stores managed by hardware in response to run time conditions. Caches usually but not always contain duplicates of the contents of memory locations.

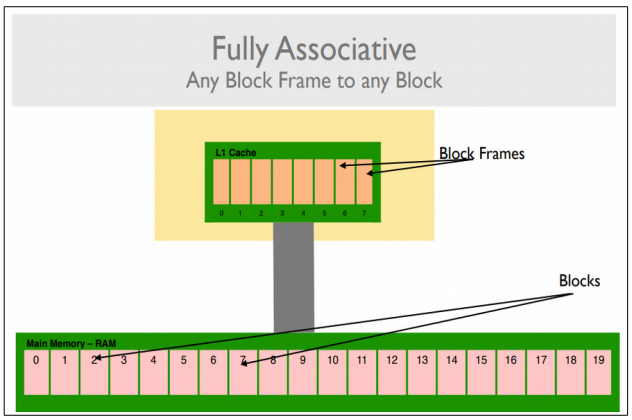
A cache is organised as a (very small) number of block frames each capable of holding a chunk of contiguous main memory locations in a structure called a 'cache line' or a 'cache block'. Due to problems building tag RAM there are different kinds of cache management

1. Direct Mapping



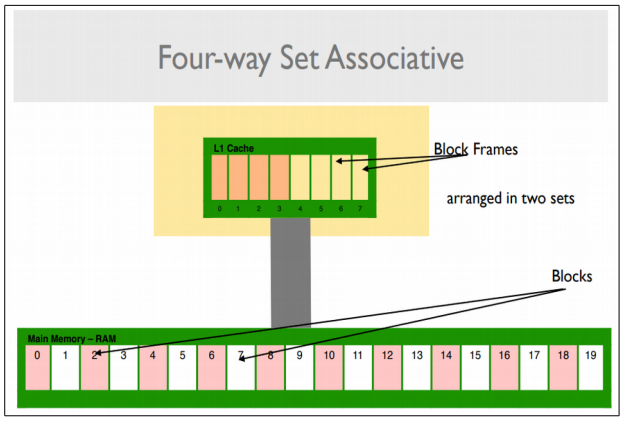
- Each block frame in cache can only cache a subset of memory blocks.
- Faster.
- Restrictive

2. Fully Associative Mapping



- Any RAM block can be mapped into any block frame in cache.
- Simple to understand.
- Problem is that to find a reference the entire tag RAM must be searched.

3. Four-Way Set Associative Mapping



- Blocks and block frames are divided into sets.
- Full associativity is possible within the sets.

Caches are invariably smaller than main memories therefore there is pressure for space within them. As programs execute over time caches will fill up and new references will have to be accomodated. This requires some form of Cache Replacement / Eviction Policy. Ideally this replacement policy should have minimum effect on the overall performance.

A Cache Miss is when a reference is made that the cache can't satisfy. There are three main reasons for a cache miss occuring:

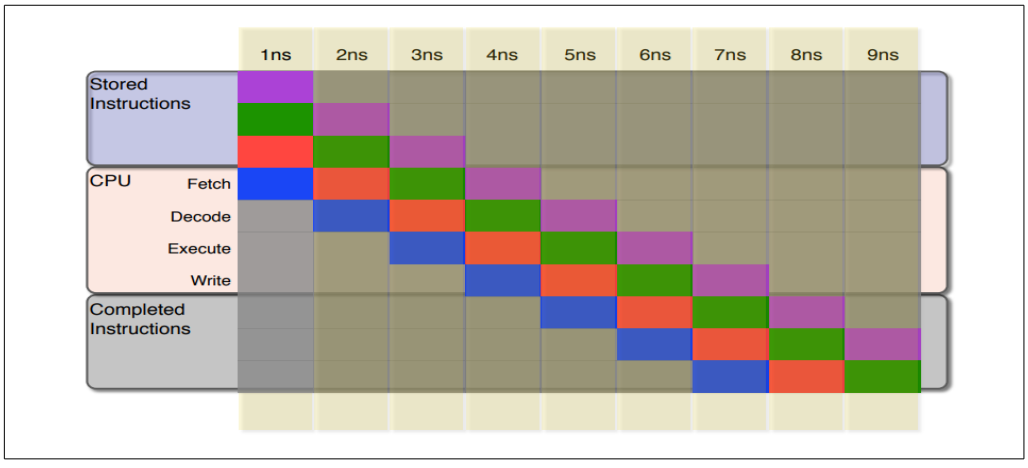
1. Compulsory Miss: The reference was never stored in cache.
2. Collision Miss: A reference was stored in cache but was evicted because of replacement.
3. Capacity Miss: A reference was stored in cache but was evicted because of memory.

d) Explain the operation of a typical three-stage pipelined processor such as the ARM processor you have been using in the laboratories.

A three staged processor such as the LG2138 board has a three-stage operational cycle which takes the form of:

1. Fetch: Instruction is fetched from memory.
2. Decode: Instruction is decoded and turned into machine code, ready to be executed.
3. Execute: Instruction is executed by the processor.

Whilst the system is in the process of decoding the current instruction, the pipeline is simultaneously fetching the next instruction to be decoded. Similarly, when the system is executing an instruction it is also decoding the next instruction and fetching the instruction to be decoded after that one.



d) Explain how branch or jump instructions can disrupt the operation of a processor's pipeline. What can be done to minimize this problem?

If a branch or a jump instruction is executed, the instructions that were fetched sequentially (were further back in the pipeline) must then be discarded as they are no longer part of the current instruction cycle.

The pipeline must be flushed and refilled. The deeper the pipeline the more chance of a branch instruction being in the pipeline and also the more instructions that must be discarded. This is known as a Control Hazard.

To minimize control hazards sometimes the status of a system and the previous registers before the branch or jump had occurred are preserved, be it on a designed data structure or on the system stack. This can be seen in subroutines when the previous registers are preserved onto the system stack in the event that they might be used again.

Q2 - Interfaces & Subroutines

a) What does an interface do? What two or three general categories of information flow between an interface and a program?

An interface is a method/program that allows the system to communicate to and from peripheral devices such as an LED or a buzzer. An interface allows users to make full use of the peripheral device whilst also being able to draw information from such device.

The general categories of information flow between an interface and a program are:

1. Program → Interface: Setting bits, setting timer, program using the peripheral device.
2. Interface → Program: Interrupts, Exceptions, Buttons

b) What would be the characteristics of a well-behaved subroutine?

A well-behaved subroutine is one that could perform its functionality whilst not interfering (unintentionally) with any other systems and/or their information.

Such a subroutine would be one that, when called, preserves the registers of the system before the subroutine itself had been called. It would also preserve the flags and status (modes) of the system before the subroutine had been called, this is done by backing up the CPSR. Finally, it would also reserve the value of the program counter before the subroutine as to allow the flow of execution to return to its original location.

When the subroutine had finished it would then restore the registers, CPSR and PC to their original values whilst also ensuring that the stack pointer is now pointing to the previous place within the system stack.

c) Write a subroutine to read the state of four switches each of which is connected to one bit of an eight-bit parallel interface whose address is equated to the label KEYS. The switches are connected to bits 0, 1, 2, and 3 respectively. The subroutine should return the bit number of the switch being pressed in R0. If more than one switch is pressed, the subroutine should return the value -1 in R0. Normally, when a switch is not pressed the value of its corresponding bit is 1 and when it is pressed the value is 0.

```
1  //Button Handler Subroutine
2
3  //Inputs :
4  //Outputs: R0 – Bit Number of switch being pressed if any (-1 if none or more than one).
5
6  buttonHandler
7
8    STMFD SP!,{R1-R4,LR}    //Preserve registers on the stack
9
10   LDR R0, =KEYS           //R0 = Interface Address
11   LDRB R1, [R0]           //R1 = Interface Bytes (Switches connected to bit 0,1,2,3)
12
13   LDR R2, =0              //count
14   LDR R3, =0              //switchOnCount
15   LDR R4, =0              //switchOnID
16
17 floop
18   CMP R2, #3
19   BGE exitLoop
20   LDR R0, = 0x01          //Switch Mask
21   AND R0, R0, R1          //Extract Bit 0 (Switch X)
22
23   CMP R0, #1              //If switch.isOff()
24   BEQ switchOn
25   ADD R3, R3, #1          //switchOnCount++
26   MOV R4, R2              //switchOnID = count
27
28   ADD R2, R2, #1          //count++
29   LSR R1, R1, #1          //Interface Bytes >>
30
31 switchOn
32   ADD R2, R2, #1          //count++
33   LSR R1, R1, #1          //Interface Bytes >>
34
35 exitLoop
36   CMP R3, #1              //if switchOnCount != 1
37   BNE negativeReturn
38   MOV R0, R4              //R0 = switchOnID
39   B end
40
41 negativeReturn
42   LDR R0, =0xFFFFFFFF     //R0 = -1
43 end
44   LDMFD SP!,{R1-R4,LR}
```

d) If the switch suffers from "bounce", explain how you would modify the subroutine to deal with it?

A switch suffering from bounce means that it temporarily flickers between closed and open (milliseconds) whilst the internal parts of the system come into contact to implement the switch being closed. This means that there could be a time when you poll the switch and it appears to be off however it actually isn't. To account for this you could implement a timer and poll the switch before and after the timer to ensure the correct result.

Q3 – Interrupts & Modes

a) The ARM processor you've been studying has a number of different modes. Name four of them and explain what they are for.

- User: Unprivileged mode, used for execution of normal programs.

  Access to : R0 – R12, SP, LR, PC and CPSR.

- FIQ: Fast Interrupt Request mode this is used when responding to interrupts from outside of the system in a quick manner. It is used when handling high priority interrupt requests.

  Access to : R8 – R14, SPSR.

- IRQ: Interrupt Request mode this is used when responding to general-purpose interrupt requests.

  Access to : R13 – R14, SPSR.

- Supervisor: Privileged mode used when OS calls (Software Interrupts – SWI's) occur within a system.

  Access to : R13 – R14, SPSR.

- Undefined: Mode used to handle undefined instructions that occur within program execution.

  Access to : R13 – R14, SPSR.

- System: Privileged mode, *but* with access to all registers. It is a cross between User and Supervisor mode offering the privileges of Supervisor with the register banks of User.

  Access to : R0 – R12, SP, LR, PC, CPSR.

b) Explain what an interrupt is, what might cause it and what it is for.

An interrupt is a signal that occurs within a program causing the regular execution cycle to halt and to enter a new branch of execution known as an interrupt handler. Interrupts may be caused intentionally within a program, known as a Software Interrupt (SWI) or as a result of an external/internal interface within a system such as the LPC2138 being pressed such as a buzzer or switch.

They are used to allow for communication from external devices and for certain functionality within programs. The ARM7 has two interrupt requesting lines; IRQ and FIQ. As a result, when handling these interrupts the ARM7 uses a Vectored Interrupt Controller (VIC) to map each respective input to a corresponding interrupt handler subroutine.

c) Explain what is the difference between a vectored interrupt and a non-vectored interrupt. Why choose one over the other? How are vectored interrupts handled in the LPC2138?

When an interrupt occurs, beit a FIQ or an IRQ it is mapped to the Vectored Interrupt Controller (VIC) within the LPC2138. This is an internal device that contains a mapping table which maps each respective interrupt with a corresponding interrupt handler subroutine. It can be programmed specifically to handle certain interrupts and link them with a given subroutine.

Non-vectored interrupts are interrupts that occur and have no pre-defined/mapped subroutine to manage the handling of such an interrupt. These are not ideal within a system as they lead to unpredictable program execution and non-deterministic outputs. It is much more ideal to use vectored interrupts as they are somewhat deterministic as they are mapped to interrupt handlers specifically designed for each given case.

d) As you know, in the Software Interrupt (SWI) instruction, the least significant 24 bits of the instruction are uncommitted, and so can be used to signify anything. Write an SWI exception handler which treats the 24 least significant bits of the SWI instruction as three signed 8-bit numbers and which returns the sum of the three numbers as a 32-bit signed integer in R0. All other register values should be unaffected.

```
1
2     AREA TopLevelSwi, CODE, READONLY               // Name this block of code.
3
4     EXPORT    SWI_Handler
5
6   /* Software Interrupt Handler Subroutine */
7
8   /* SWI Instruction = [0000][0000][00000000 00000000 00000000]
9                         COND   X        SWI Number (24 bits)
10
11       This subroutine extracts SWI Number and treats it as
12       3 x 8 bit values respectively. Subroutine returns
13       the sum of the 3 values in R0
14
15    */
16
17  SWI_Handler
18    STMFD SP!,{R1-R12,LR}                          // Store registers.
19
20    LDR   R0,[LR,#-4]                              // Calculate address of SWI instruction and load it into r0.
21    BIC   R0,R0,#0xFF000000                        // Mask off top 8 bits of instruction to give SWI number.
22
23    AND R1, R0, #0x000000FF                        // Extract leftmost 8 bits
24
25    AND R2, R0, #0x0000FF00                        // Extract middle 8 bits
26    LSR R2, R2, #8                                 // middleBits >> 8
27
28    AND R3, R0, #0x00FF0000                        // Extract rightmost 8 bits
29    LSR R3, R3, #16                                // rightBits >> 16
30
31    ADD R0, R1, R2                                 // sum = leftBits + middleBits
32    ADD R0, R0, R3                                 // sum = sum + rightBits
33
34    LDMFD   SP!, {R1-R12,PC}^                      // Restore registers and return.
35
```