ECED 3403 – Computer Architecture

Quiz 2

Solutions

30 June 2021

Here are a set of suggested solutions to Quiz 2.

1. *(1) Can a stack frame be accessed using absolute addressing? Explain your answer.*

An **absolute** address is one that remains fixed for the duration of a program's execution. Global variables are typically absolute. The memory locations used by a program's instructions can be thought of as absolute because they do not change as the program runs.

A stack frame is created on a process's stack whenever a subroutine (function) is called. The stack frame is accessed by a register often referred to as BP or the base pointer. Each time the function is called, its base pointer can change, depending on the depth of the stack (that is, the number of function calls and the size of their stack frames).

Since the stack frame's address can change over time, we cannot use absolute addressing to access its contents (i.e., arguments and automatics). We can (and do) use **relative** addressing to access the stack frame.

In relative addressing, we have a base address and from the base address, we can access various memory locations using an offset. For example, we can access the contents of the stack frame using the BP as the base register and offsets into the stack frame, with positive offsets for the arguments and negative offsets for the automatics.

Relative addressing is used in other data structures (see question 4) and code structures (see question 5).

2. *(1.5) CISC (Complex Instruction Set Computer) architectures have a myriad of instructions, some of which are effectively duplicates, such as* `INC R1` *and its equivalent* `ADD #1,R1`*. On the other hand, RISC (Reduced Instruction Set Computer) architectures, like XM3, with a limited number of instructions overcome this problem by using a preprocessor to convert instructions like* `INC R1` *into* `ADD #1,R1`*. RISC architectures reduce ISA size and complexity. How could the following instructions found in most ISAs be made from XM3's existing instructions:*

   a) *(0.5)* `NOP` *– No operation.*

The NOP instruction causes the machine to do "nothing" for one instruction cycle. In other words, there is not to be a state change, other than the PC being incremented to the next instruction. The NOP instruction can be made quite easily by moving a register to itself:

```
MOV R1,R1
```

The instruction does not change the value of R1, so the machine state is not changed, although the PC is incremented as expected. The use of electricity and the change in entropy are not considered.

   b) *(0.5)* `CLR Rx` *– Clear (zero) a register (one of R0 through R7).*

Clearing a register or memory location involves writing a zero to the structure. XM3 does not have a CLR instruction, so we must "build it" from its existing instructions.

As we saw in assignment 1, assigning a value to a register uses one of the MOVx instructions (MOVL, MOVLZ, MOVLS, and MOVH). For example, the MOVLS (move low and set upper byte) instruction moves the supplied byte into the LSByte of the register and then sets the bits in the MSByte:

```
MOVLS  #AA,R3 ; R3.LSB = #AA and R3.MSB = #FF
```

The MOVLZ is like the MOVLS instruction in that the supplied byte is stored in the LSB, but the MSByte is cleared (i.e., assigned #00). We can use the MOVLZ instruction to achieve the same effect of the CLR Rx instruction:

```
MOVLZ  #0,Rx  ; Rx.LSB = #0 and Rx.MSB = #00 (Rx is R0 through R7)
```

c) (0.5) `SHL Rx` *– Shift a register (one of R0 through R7) left by one bit.*

XM3 has several instructions to shift a register to the right, but none to shift a register to the left.

When a register is shifted to the left, the bits move by 1 to the left, bit the MSBit usually being copied into the carry bit and a zero being fed into the LSBit. This is equivalent to doubling the value of the register or multiplying it by two (why?).

To create the equivalent of the SHL we need an instruction that can double the value of a register. One way of doing this is to add the register to itself:

```
ADD Rx,Rx  ; Where Rx is a register R0 through R7
```

Using the `ADD` instruction this way means the `SHL` instruction can be emulated without having to design special circuitry to handle an additional instruction.

The above instructions, and others, can be found in section 7 of the XM3 Instruction Set Architecture book available on the course website. (SHL is shown as SLA.)

3. *(1) A one-address ISA has four index registers. Instructions using an index register represent the register using two bits, with the MSBit stored in bit 2 and the LSBit stored in bit 5. If you were to design a disassembler for this ISA, how would you find which register is being accessed in an instruction? (That is, the register number, 0 through 3.)*

In assignment 1 we extracted bits from an instruction. In the case of the register bits, they were in contiguous bits, making them easy to extract. For example, the destination register was always the instruction ANDed with 0x07 and the source register was the instruction shifted right by 3 and then ANDed with 0x07.

In this question, the instructions aren't so nicely laid out, so to get the register number we need to access two bits in separate locations and then merge them together.

The machine has four index registers meaning we can use two bits to represent them (00, 01, 10, and 11). Since XM3 has eight registers, they can be represented in three bits (000, R0; 001, R1; through 111, R7).

An instruction using the registers is formatted as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	-	-	-	LSBit	-	-	MSBit	-	-

The LSBit (bit 5) must be stored in bit 0 and the MSBit (bit 2) must be stored in bit 1 to obtain the register number. We can do this as follows:

```
LSBit = (instruction >> 5) & 0x01;
MSBit = (instruction >> 1) & 0x02;
regno = MSBit | LSBit;
```

Or simply:

```
regno = (instruction >> 1) & 0x02 | (instruction >> 5) & 0x01;
```

When finished, `regno` has the value 0 through 3, indicating index register 0 through 3.

4. *(3) The following XM3 code fragment accesses a certain type of data structure:*

```
WHILE
    CMP   #0,R0
    CEX   NE,5,0
    LDR   R0,#0,R1
    LDR   R0,#2,R2
    BL    PRINT_R1_R2
    LDR   R0,#4,R0
    BRA   WHILE
;
ENDWHILE
```

From the information supplied in the code fragment, answer the following questions:

a) (0.5) What is the name of this type of data structure?

The data structure is a **linked list**, and the code is accessing a single **element** or **node** in the linked list.

b) (1.0) How do you know this?

Reading and understanding the code fragment is necessary to answer this question. An examination of the fragment code shows us the following:

```
WHILE
    CMP   #0,R0         ; Does R0 = 0 (NULL)?
    CEX   NE,5,0        ; No, execute the next 5 instructions
    LDR   R0,#0,R1      ; R1 <- mem[R0 + 0]
    LDR   R0,#2,R2      ; R2 <- mem[R0 + 2]
    BL    PRINT_R1_R2   ; Call print
    LDR   R0,#4,R0      ; R0 <- mem[R0 + 4]
    BRA   WHILE         ; Go to WHILE
;
ENDWHILE        ; R0 = NULL (done)
```

The code in this question uses **relative addressing** to access the data structure. We know this because the memory locations making up the structure are accessed using XM3's `LDR` (load relative) instruction.

As assignment 1 showed, `LDR` has operand three fields: the source register (containing the base address of the structure), the destination register (the register where the contents of the memory location read are to be stored), and the offset (into the structure, the address of which is the base address plus the offset). For example, if R0 is $1232, we find:

```
    LDR   R0,#0,R1      ; R1 <- mem[$1232 + 0]
    LDR   R0,#2,R2      ; R2 <- mem[$1232 + 2]
    LDR   R0,#4,R0      ; R0 <- mem[$1232 + 4]
```

The final set of clues that this is a linked list are the instructions:

```
  WHILE
    CMP   #0,R0         ; Does R0 = 0 (NULL)?
    CEX   NE,5,0        ; No, execute the next 5 instructions
    ...
    LDR   R0,#4,R0      ; R0 <- mem[R0 + 4]
    BRA   WHILE         ; Go to WHILE
```

The `LDR` instruction copies the contents of R0 + 4 into R0, indicating that the new value of R0 is stored in the structure. Since R0 must hold an address (why?), it suggests that this structure contains the address of the next structure.

Why the next structure? This is a loop and it access the same offsets each time, indicating that each structure shares the same fields (albeit they are associated with different base addresses).

The `CMP` compares #0 with R0, continuing only if R0 is not equal to #0. If it does equal zero, the five instructions after `CEX` are ignored and the loop is exited.

The XM3 code fragment is equivalent to the follow C code (is it?):

```
while (ptr != NULL)
{
  r1 = ptr -> field1;
  r2 = ptr -> field2;
  print_r1_r2(r1, r2);
  ptr = ptr -> next;
}
```

c) (1.5) How many bytes does the structure occupy and how many elements does it contain?

From the code fragment, we know that it contains three elements (fields):

0: Field 1	Value assigned to R1
2: Field 2	Value assigned to R2
4: Next	Address of next node or NULL

We also know that the loads (`LDR`) words (16-bit values), so we can assume that the three fields consist of three words. Since a word on XM3 is two-bytes, the structure must occupy six bytes.

The data structure and pointer could be defined in C as follows:

```
struct q4
{
short field1;
short field2;
struct q4 *next;
};
struct q4 *ptr;
```

5. (1.5) XM3 uses a 10-bit relative offset for branching instructions, for example:

```
SELF   BRA  SELF
```

What is the value of the 10-bit relative offset if the address of SELF is $2E3C? Explain your answer.

As assignment 1 showed, XM3 has two branching instructors, `BL` and `BRA`, both of which use **relative** rather than **absolute** addressing. In addition to the opcode, the instructions carry an encoded offset that must be decoded and then added to the PC, meaning that the destination or target address is relative to the PC.

The PC always points to the next instruction to be executed. After the instruction `BRA SELF` is fetched, the PC is incremented by 2 (why 2?) to the address of the next instruction:

```
nnnn       SELF   BRA  SELF
nnnn + 2   Next instruction
```

The instruction `BRA SELF` is to branch to `SELF` (address `nnnn`). The new value of the PC is determined as follows:

```
PC = PC + 2 + offset
```

In this case, we want PC to equal `nnnn`:

```
nnnn = nnnn + 2 + offset
```

rearranging:

```
offset = nnnn – (nnnn + 2)
```

gives:

```
offset = -2
```

This is the unencoded offset. The encoded offset is:

Offset = -2	`1111.1111.1111.1110`
Shifted right by 1	`1111.1111.1111.1111`
10-bit encoded offset	`11.1111.1111`

The 10-bit encoded offset is therefore `11.1111.1111` or `0x3FF`.

Does it matter that the address of `SELF` is `$2E3C`? Or `$1000` or `$80C4`? No, it does not because the address is always relative to the PC, regardless of the value of the PC.