ASIP Laboratory - Session 1

Paul Georg Wagner Majd Mansour

May 10, 2017

Exercise 1: Basic Assembly Instructions

a) *What is the reason for this high number of cycles? Which instruction causes that behaviour and why is it doing so?*

When the given code is executed, dlxsim shows a total of 41 executed cycles, even though there are only seven instructions. From those 41 cycles, 34 cycles are needed for the `mult` instruction alone, and seven cycles for the rest of the instructions. So the reason for the high number of cycles is the multiplication in line 8. A multiplication is an instruction that is more complicated than additions, subtractions or simple logical operations like `and` as well as `or`. Hence a multiplication takes longer than one cycle to execute. This causes a pipeline stall while executing the program, since all subsequent instructions must wait for the multiplication to finish. Furthermore, there are no pipeline stalls due to data dependencies in the code.

b) *What is the purpose of the "trap #0" instruction?*

The trap instruction interrupts dlxsim and hands the control back to the user (i.e. the shell). It can be used to stop the execution of the program at any point and then the registers or memory contents can be manually evaluated.

Exercise 2: Memory Access

a) *Explain the goal of the instruction combination lhi / ori. What is generally (so: not only for this specific example) the register value after lhi, what is it after the additional ori?*

The combination of the `lhi` and `ori` instruction is used to load the address of the first data value (i.e. the address of the label `_data`) into register r1. The `lhi` instruction first loads the highest 16 bits of the address to the highest 16 bits of register r1. So after this instruction has been executed, the register r1 will contain the most significant bits of the address of label `_data`, with the lower 16 bits still zeroed out. Then the `ori` instruction also adds the lower 16 bits of the address to the lower 16 bits of register r1. In order to only get the lower 16 bits of the address, the `ori` instruction operates on a masked value, i.e. `_data & 0xFFFF`. After the `ori` instruction is completed, the register r1 will contain the full 32 bits of the address of the label `_data`. It is necessary to load the address in two steps with 16 bits each, because there is no instruction that loads 32 bit immediate values directly into registers.

b) *Why is it in general not possible to omit the lhi instruction, although it would be possible in this special example?*

The purpose of the `lhi` instruction is to load the most significant 16 bits of the address to the register. Since this is also a part of the address to be loaded, it cannot be omitted in general. However, in this case the specific address to load is only 0x24, which is sufficiently small for the most significant 16 bits of the 32 bit long address to be zero. This is why in this special case the `lhi` instruction does not change the already zeroed register, and could therefore be omitted.

Exercise 3: Branches

a) *Which high-level control structure (e.g. "call subroutine"...) is implemented in this example code?*

The code contains an if-else structure.

```
if(r1 > r2)
  r4 = r1;
else
  r4 = r2;
```

b) *What is computed with this example? r4 = function (r1, r2);*

The code implements a routine that returns the maximum of two values in register r4.

```
r4 = max(r1,r2);
```

c) *Look at the NOP instructions and explain why they are placed there.*

In the provided code there is one NOP instruction after each branch and jump. This is necessary because whenever a branch is taken, the program flow changes and the instructions that reside immediately below the branch usually should not be executed. However, due to the pipeline architecture, some instructions below the branch will already be loaded to the pipeline. Those instructions become obsolete by branching and, if executed, could interfere with the correct execution of the program. In order to avoid problems with the pipeline a NOP is inserted after each branch. When the branch instruction reaches the EXE phase of the pipeline, the program counter is updated to the new address, and the subsequent instruction is loaded into the IF phase. During this, the additional NOP fills the ID phase and avoids that any other instruction is erroneously decoded. Without the NOP, whatever instruction resides immediately after the branch instruction would be executed as well, even though the program flow already should have been changed.

d) *Which meaningful instruction (i.e. an instruction that helps executing the code; not just any dummy instruction) can replace the first NOP instruction without changing the final result and why does it not change the final result?*

In this specific case the first NOP instruction can be replaced with an `add r4, r0, r1` instruction. If the branch is then taken (i.e. r1 > r2), the added instruction is decoded and executed before the actual add instruction after the jump in line 12. This gives us the result in register r4 one cycle earlier.

Furthermore, in this case the first NOP can also be omitted altogether. If the branch is then not taken (i.e. r2 $\geq$ r1), we save one cycle. If the branch is taken, the subsequent `add r4, r0, r2` instruction erroneously enters the ID phase and is executed. However, after the jump the wrong result in register r4 is overwritten by the `add r4, r0, r1` instruction, so the final result of r4 will still be correct.

Exercise 4: Loops

a) *What is computed with this example? r3 = function (r1, r2). Debug the application step-by-step with the capabilities of dlxsim.*

This code implements a multiplication.

```
r3 = r1 * r2;
```

b) *The approach to compute the function in the way this example is doing it has two specific names. Do you know those names? (One is founded by the main operations, while the other is founded historically. You either know the names or you don't. If you don't know both names, you may guess)*

The given code implements multiplication by adding and shifting. The algorithm iterates the multiplier and sums up a shifted value of the multiplicand for each "1" bit found in the multiplier. This approach to multiplication is also called peasant multiplication or Egyptian multiplication [1].

c) *In general, how often is the loop maximally executed? How the input data has to look like to get this maximal number of iterations?*

The number of loop iterations is simply the number of binary digits of the value in register r1, i.e. the multiplier. Since the register r1 is 32 bits in size, the maximum number of iterations is 32. In order for the maximum of 32 iterations to be executed, the most significant bit of register r1 has to be set. The number of executed additions in the algorithm is not directly linked to the number of iterations, but is instead the number of ones in the multiplier.

Exercise 5: A High Level Structure

a) *Which high-level control structure do you recognize? Explain the purpose of the instruction-block between "addi r3, r0, 2" and "jr r4".*

The given code implements a simple switch-case statement where register r3 chooses the jump target. If register r3 is `i`, the code will jump to the label `mark_i`. In order to do so, the `lhi`/`ori` instructions in line 6 and 7 first load the address of `_label1` into register r4. Then the `slli` instruction in line 8 multiplies the jump target number stored in r3 with 8, using three left shifts. The result is stored back in register r3. The jump target number has to be multiplied with 8 because the jump targets – beginning with line 14, 16, 18, ... – are 8 bytes apart (2 instructions, each having 4 bytes). Finally, the registers r3 and r4 are added together in order to get the final jump address (`add` instruction in line 9). Then the `jr r4` instruction in line 10 jumps to the determined address. Since in the provided example the register r3 is set to the value 2 in the beginning, the code jumps to the label `mark_2` and executes the `xor` instruction in line 35.

b) *Why do you have to shift by value 3? Explain it with a close view to the body of the control structure and pay attention to the addressing mode of the DLX processor.*

The `slli` instruction in line 8 shifts the value in register r3 three bits to the left. This shift equals a multiplication with value $2^3 = 8$. This multiplication is necessary because each instruction is 4 bytes in length and we have to add the length of two instructions for each case statement, since each case statement consists of one jump and one NOP instruction, beginning at `_label1`. So by multiplying the jump target value given in register r3 with 8 the necessary jump offset from `_label1` is determined. When this offset is added to the absolute address of `_label1`, the result is the absolute address of the jump target. This calculation is subsequently done in line 9.

c) *What are the general differences between branch and jump instructions in the DLX instruction set (also have a look at the different instruction formats to find a part of the answer)?*

[1] https://en.wikipedia.org/wiki/Ancient_Egyptian_multiplication

Branching (`beqz`, `bnez`) is conditional, defines PC-relative addressing and uses the I-Format of the instruction formats. Register jumping (`jr`, `jalr`) is unconditional, defines absolute addressing and also uses the I-Format. Jumping (`j`, `jal`) is unconditional, defines PC-relative addressing and uses the J-Format of the instruction formats.

Exercise 6: Writing Assembly Code

a) *Attach the assembly file to the mail.*

The task is solved in file 6_assembly.s.

```
; Input r1(address of A), r2(address of B),
; r3(number of elements), r4(address of C)

_main:

        ; Parameters
        lhi     r1, (_A >> 16)
        ori     r1, r1, (_A & 0xFFFF)
        lhi     r2, (_B >> 16)
        ori     r2, r2, (_B & 0xFFFF)
        addi    r3, r0, 10
        lhi     r4, (_C >> 16)
        ori     r4, r4, (_C & 0xFFFF)

        ; Loop
        lw      r6, 0(r4)           ; Load value C
_loop:  beqz    r3, _end            ; Step out of the loop
        nop
        lw      r7, 0(r2)           ; Load value B[i]
        addi    r7, r7, 5           ; B[i] + 5
        add     r7, r7, r6          ; B[i] + 5 + C
        sw      0(r1), r7           ; Store result in A[i]
        addi    r1, r1, 0x4         ; Go to next value in array
        addi    r2, r2, 0x4         ; Go to next value in array
        subi    r3, r3, 1           ; Decrement counter by 1
        j       _loop               ; Jump to loop
        nop

_end:   trap    #0


_A:     .data.32        0,0,0,0,0,0,0,0,0,0
_B:     .data.32        0,1,2,3,4,5,6,7,8,9
_C:     .data.32        42
```

b) *How many assembly instructions do you need altogether?*

The developed solution requires 7 instructions for setting up the parameters and another 12 instruction for the actual loop. This gives a total of 19 instructions to solve the task.

c) *How many clock cycles does your code need to execute?*

The developed code takes a total of 148 cycles to execute.

d) *How many memory accesses are performed altogether?*

The developed solution makes 10 memory accesses to the array A (store word), 10 memory accesses to the array B (load word) and another single memory access to the value C (load word). This gives a total of 21 memory accesses.