ENCM 369 Winter 2018 Lab 6 for the Week of February 26

Steve Norman
Department of Electrical & Computer Engineering
University of Calgary

February 2018

Lab instructions and other documents for ENCM 369 can be found at
`http://people.ucalgary.ca/~norman/encm369winter2018/`

Administrative details

You may work in pairs on this assignment

You may complete this assignment individually or with *one* partner.

Students working in pairs must make sure both partners understand *all* of the exercises being handed in. The point is to help each other learn *all* of the lab material, not to allow each partner to learn only half of it! Please keep in mind that you will not be able to rely on a partner to do work for you on midterm #2 or the final exam.

Two students working together should hand in a *single assignment* with names and lab section numbers for both students on the cover page. Names should be complete and spelled correctly. If you as an individual are making the cover page, please get the information you need from your partner. For partners who are not both in the same lab section, please hand in the assignment to the collection box for the student whose last name comes first in alphabetical order.

Due Dates

The Due Date for this assignment is 3:30pm Friday, March 2.
The Late Due Date is 3:30pm Monday, March 5.

The penalty for handing in an assignment after the Due Date but before the Late Due Date is 3 marks. In other words, X/Y becomes (X–3)/Y if the assignment is late. There will be no credit for assignments turned in after the Late Due Date; they will be returned unmarked.

Marking scheme

A	4 marks
B	4 marks
E	4 marks
F	4 marks
TOTAL	16 marks

How to package and hand in your assignments

Please see the Lab 1 instructions.

Exercise A: 16-bit two's complement and MIPS instructions

Part I

Here is a MIPS assembly language instruction to allocate 40 words on the stack:

```
addiu   $sp, $sp, -160
```

Answer the following questions:

1. What is the machine code for the instruction? Show how you obtained your answer, and express your answer in base two.
2. Suppose that before the instruction is run, `$sp = 0x7fff_e8c0`. A 32-bit adder will compute the new value for `$sp`. What are the two 32-bit inputs to the adder, and what is the 32-bit output of the adder? Show how you obtained your answers, and express all your answers in base two.

What to Hand In for Part I

Hand in typed or neatly handwritten answers to questions 1 and 2.

Part II

In Lab 5 Exercise B you saw that store and load instruction mnemonics like `sw` and `sb` could be used to make pseudoinstructions with labels as operands, such as

```
sw      $t0, foo
```

and

```
sb      $t8, str99
```

where `foo` and `str99` were labels for data items in `.data` sections.

Now let's consider a similar pseudoinstruction for a load operation:

```
lw      $s0, bar
```

Suppose that the address of `bar` is `0x1002_b000`.

It would be tempting to conclude that the pseudoinstruction could be turned into the following two real instructions, in a way similar to what was done in Lab 5 Exercise B:

```
lui     $at, 0x1002
lw      $s0, [some offset]($at)
```

1. *Explain why it is impossible to load from address `0x1002_b000` using the above two instructions, regardless of the value of* $\boxed{\text{some offset}}$. (Hint: Remember that the offset in a load or store instruction is a *signed* 16-bit constant.)
2. *Show how the pseudoinstruction can be implemented using an `lui` instruction followed by a real `lw` instruction.* Express your answer in assembly language using a *base ten* offset in the `lw` instruction. (Hint: Use a negative offset.)

What to Hand In for Part II

Hand in typed or neatly handwritten answers to questions 1 and 2.

Exercise B: Integer addition examples

Read This First

This is a pencil and paper exercise. Don't use a computer or a calculator.

In this exercise, you get to add bit patterns as if you were an 8-bit processor, millions of times slower than electronic hardware.

What to Do

For each of the following pairs of hexadecimal patterns, write 8-bit binary patterns, and find the sum that would be generated by adding the patterns with an 8-bit adder.

PART	*a*	*b*	PART	*a*	*b*
I	`0x90`	`0xc0`	III	`0x45`	`0x48`
II	`0x45`	`0x38`	IV	`0xf2`	`0xf3`

For each part, do the following after calculating the sum:

- First interpret the numbers as two's-complement signed values. Decide if the addition resulted in *signed* overflow.
- Then interpret the numbers as unsigned values. Decide if the addition resulted in *unsigned* overflow.

Show the work you did to generate the addition results and explain in each case how you decided whether overflow occurred.

Exercise C: Scary facts about real-world C integer types

Read This First

A key fact, emphasized in ENCM 369 lectures: Two's complement is by far the most common system for representing signed integers in computers. (This fact is not scary, but the next two are.)

Another key fact, also emphasized in ENCM 369 lectures: Typical implementations of C and C++ will allow integer arithmetic to generate results that do not make sense according to normal, everyday mathematics; these implementations will not do anything special to alert programmers or users that results probably do not make sense. For example, it is quite easy to get a C program to add two `int` variables with positive values and generate a sum that is apparently negative.

Here is a third key fact: C and C++ programming language standards *do not exactly specify* the numbers of bits that should be used for numerical types!

Back in the early days of C, there were three signed integer types: `short int`, `int`, and `long int`. (There were also various unsigned types, and `char`, which was usually an 8-bit type, but which was signed on some systems and unsigned on others.) It may seem weird, but for a given platform, typically one of the following was true:

- `short int` and `int` were the same size, while `long int` was wider than `int`;
- `short int` was narrower than `int`, while `long int` and `int` were the same size.

Figure 1: Sizes for signed integer types on various platforms.

Platform	`short int`	`int`	`long int`	`long long int`
Typical Unix minicomputer or workstation, starting in early 1980's	16 bits	32 bits	32 bits	n/a
PC with MS-DOS and typical C compiler, starting in early 1980's	16 bits	16 bits	32 bits	n/a
Linux system with x86 architecture	16 bits	32 bits	32 bits	64 bits
Linux system with x86-64 architecture	16 bits	32 bits	64 bits	64 bits
Current and recent Mac OS X (64-bit)	16 bits	32 bits	64 bits	64 bits
Current and recent Microsoft Windows (both 32- and 64-bit), with Microsoft compilers	16 bits	32 bits	32 bits	64 bits
Cygwin64 running on top of Microsoft Windows	16 bits	32 bits	64 bits	64 bits

Later a type called `long long int` became available; this type was usually 64 bits wide. Figure 1 gives some size information for some C types on some important platforms.

Now consider software development in the early 1980's. It would not have been uncommon for a programmer to write C code for an expensive Unix workstation, get an executable built and tested, then later copy the C source code to a cheaper PC running MS-DOS and try to build an executable for the PC. If values of `int`s could be expected to go beyond the range of a 16-bit type, the PC version of the program could fail in ways that were never seen on Unix, and that were either mysterious or just annoying, depending on how much the programmer knew about how `int` values were represented on the two different platforms.

In 2018, a similar problem can occur if programmers move code from one platform to another and are not careful to check on the width of the `long int` type—the width is 64 bits for many current platforms, but only 32 bits for others.

(By the way, the designers of the Java programming language had experience with the problems of a given C or C++ type having different sizes on different platforms. For Java there are strict rules such as, "an `int` must be represented using 32-bit two's-complement," and "a `long` must be represented using 64-bit two's-complement.")

What to Do

Copy the directory `encm369w18lab06/exC`

Do not yet make an executable and run it. Instead, carefully read the code in `long_ints.c` and develop predictions for the program output for the following cases: first, a platform on which `long int` is a 32-bit type; second, a platform on which `long int` is a 64-bit type. Here are a few facts which may be helpful:

$2^{30} = 1,073,741,824$
$2^{30} + 2^{28} = 1,342,177,280$

$2^{31} + 2^{28} = 2,415,919,104$
$2^{31} - 2^{28} = 1,879,048,192$

You can test your prediction for 64-bit `long int` by building and running executable on Cygwin64.

For 32-bits, you can write a small Mars program that uses `addu` to add the integers 1342177280 and 1073741824, then print the sum.

What to Hand In

There is nothing to hand in. If it's not clear to you why the two programs produce the output they do, you can check an explanation that will be posted on the course "lab solutions" web page on or before February 27.

Exercise D: 64-bit addition with 32-bit registers

Read This First

The material in Exercise C may raise a number of questions. One of them might be: How is a 64-bit integer type possible on a machine where the all the general-purpose registers (GPRs) are 32 bits wide?

A partial answer is: If 64-bit integer variables are to be in GPRs, each of those variables will need two GPRs; an operation such as addition cannot be done with a single instruction, so will need a sequence of instructions instead.

Consider this situation on MIPS: `i`, `j`, and `k` are all 64-bit signed integer variables in a C program. Suppose registers are allocated this way:

`i`: bits 31–0 in `$s0`, bits 63–32 in `$s1`
`j`: bits 31–0 in `$s2`, bits 63–32 in `$s3`
`k`: bits 31–0 in `$s4`, bits 63–32 in `$s5`

How could the C statement `k = i + j;` be implemented in assembly language?

Let's pretend that two untrue things are actually true. First, let's suppose that the MIPS `addu` instruction places the carry out of the most significant bit of the adder hardware into a special location within the processor called the *carry bit*. Second, let's suppose there is an instruction called `addc` ("add-with-carry") that is exactly like `addu`, except that it takes the 0 or 1 in the carry bit as a carry into the least significant bit of the adder hardware. With those two pretended conditions in place, the 64-bit addition is simple:

```
addu $s4, $s0, $s2  # get bits 31-0 of result
addc $s5, $s1, $s3  # get bits 63-32 of result with
                    # appropriate carry from bit 31 to 32
```

So why did I bother to explain how to solve the problem using fictional properties of the MIPS architecture? I did it because the carry bit and add-with-carry instructions are available on a wide range of important architectures, including x86 and ARM, and, I would guess, most others, so this kind of solution is useful to know about.

A similar solution is usually available to allow 32-bit integer addition on architectures with 16-bit GPRs.

What to Do

Consider the problem of implementing the 64-bit addition of the Read This First section with real MIPS instructions.

1. Find a solution that is a sequence of five instructions—`addu`, another `addu`, `addiu`, `beq`, `sltu`, *not* in that exact order. Hint: Think about the condition to detect unsigned overflow in addition—that happens if and only if the carry out of the most significant bit is 1.
2. Find another solution that is only four instructions in length, without any branches. Hint: Think about what possible bit patterns `sltu` could put into its destination register.

Note that if you want to test your work using MARS, you can, but first try to convince yourself of the correctness of your work without actually writing a MARS program.

What to Hand In

There is nothing to hand in. Solutions will be posted on the course "lab solutions" web page no later than March 2.

Exercise E: Integer subtraction examples

Read This First

An 8-bit adder can compute $a - b$ by inverting the bits of b and supplying a value of 1 as the carry in to the least significant bit. For example, if a is `0000_1001` (9 in base ten) and b is `0000_0100` (4 in base ten), here's how the subtraction works:

```
carry in:             11110111
----------------------------
bits of a:            00001001
inverted bits of b:   11111011
----------------------------
result:               00000101
```

Note that the correct result (5 in base ten) is produced.

What to Do

Like Exercise B, this is a pencil and paper exercise. Don't use a computer or a calculator.

For each of the following pairs of hexadecimal patterns, write 8-bit binary patterns, and find the 8-bit subtraction result $a - b$. As in Exercise B, identify cases where signed overflow has occurred and cases where unsigned overflow has occurred.

PART	a	b	PART	a	b
I	`0x70`	`0xe9`	III	`0xc5`	`0xb2`
II	`0x93`	`0x14`	IV	`0x41`	`0xfe`

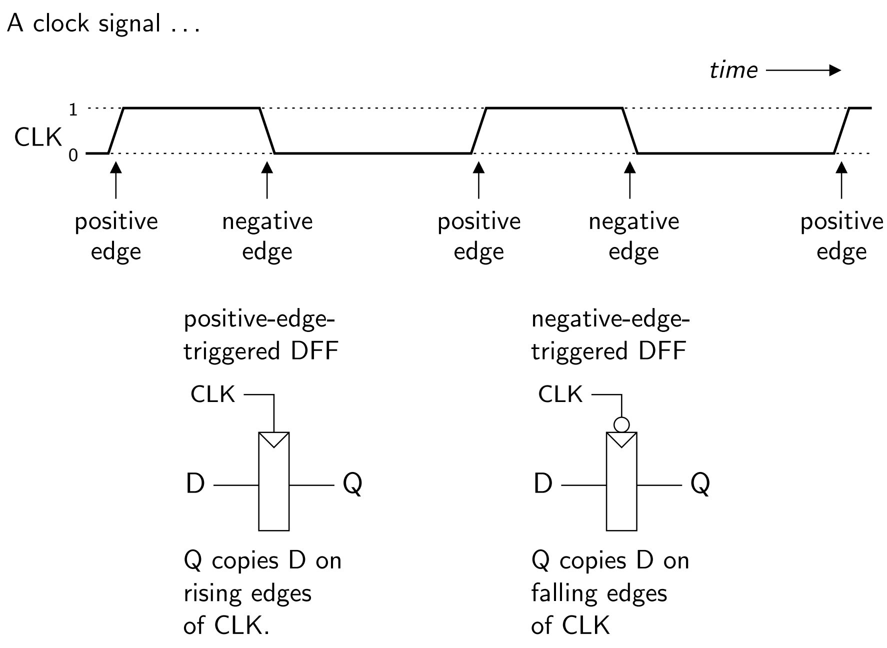
Show the work you did to generate the subtraction results and explain in each case how you decided whether signed overflow and/or unsigned overflow has occurred.

Exercise F: Review of D Flip-Flops

Read This First

D flip-flops (DFFs) were covered in ENEL 353. They are very important building blocks for the processor designs that we will study in lectures starting on February 16.

Figure 2: Summary of D flip-flop behaviour. *Nothing about digital logic design is more important than this!* Getting this wrong in any way should be considered to be just as serious a mistake as writing an incorrect truth table for an inverter, an OR gate, or an AND gate.


A clock signal . . .
time
CLK
1
0
positive edge
negative edge
positive edge
negative edge
positive edge
positive-edge-triggered DFF
CLK
D
Q
Q copies D on rising edges of CLK.
negative-edge-triggered DFF
CLK
D
Q
Q copies D on falling edges of CLK

Very shortly *after* an active clock edge, Q takes on the value D had just *before* that clock edge; Q then *holds* that value until the next active clock edge.

The critical thing to understand about a DFF is that its state Q can change *only* in response to an *active edge* of its *clock* input.

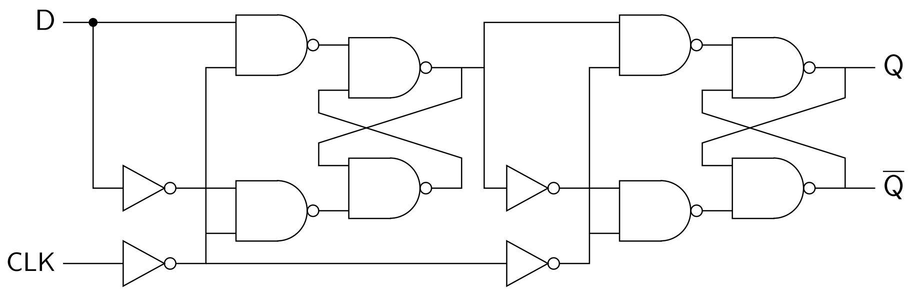
- For positive-edge-triggered DFFs (the kind we saw a lot in ENEL 353) the active edge of the clock is the *rising edge*—a low-to-high transition;
- for negative-edge-triggered DFFs (which we did *not* see much in ENEL 353), the active edge of the clock is the *falling edge*—a high-to-low transition.

Behaviour of DFFs is summarized in Figure 2.

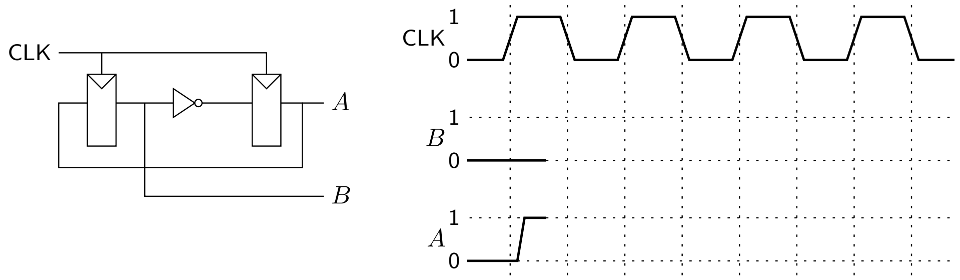
Knowing how a DFF responds as a "black box" to its clock and D inputs is essential in understanding sequential logic designs. On the other hand, knowing what is *inside* a DFF is not very important at all, unless your job is designing DFFs to meet specifications such as high speed, low chip area, and low power consumption.

Figure 3 shows a classic design for a DFF that was presented in ENEL 353. Don't worry if you don't understand how it works—that's not important in ENCM 369. (Note also that DFFs in modern CMOS integrated circuits don't use NAND gates at all—typically DFFs are built from inverters and devices called *transmission gates*, which we did not study in ENEL 353.)

Figure 3: Positive-edge-triggered D flip-flop built from two NAND-based D latches. **Knowing exactly how this circuit works will NOT help you at all in ENCM 369!** It is *infinitely more important* to understand what is meant by the text in Figure 2!



Example problem. (This was a problem on the December 2012 Final Exam in ENEL 353.) Complete the timing diagram:



Solution to example problem. Before drawing waveforms, a very small amount of analysis is required:

- The DFFs are positive-edge-triggered, so the dotted lines dropping from negative edges of the clock are irrelevant and should be ignored.
- For the FF that has A as its output, the D input is $\overline{B}$. (I am assuming you have not forgotten what an inverter does!) That means that just after a positive clock edge, A will take on the value $\overline{B}$ had just before that edge. In the notation of the textbook used for ENEL 353 and ENCM 369 this year—and for the previous few years— $A' = \overline{B}$.
- For the FF that has B as its output, A is wired directly to the D input. So $B' = A$.

This analysis leads to the solution in Figure 4.

What to Do

Print a copy of page 10 of this document.

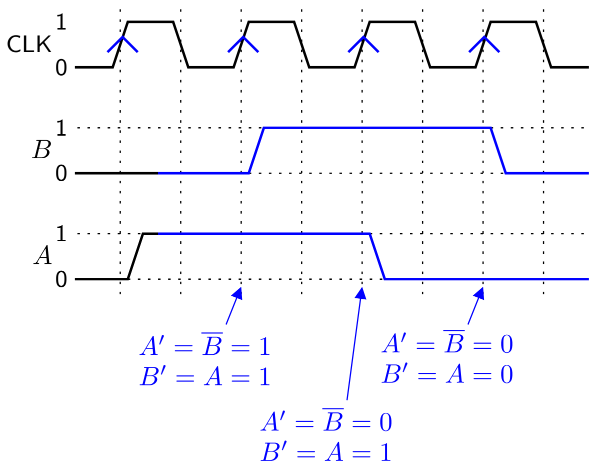
In the spaces beneath each of the two given circuits, write next-state equations for the circuit. In Part I, that means equations giving Q_1' and Q_0' in terms of Q_1 and Q_0. In Part II, that means equations giving Q_0', Q_1' and Q_2' in terms of Q_0, Q_1, Q_2 and the input A.

Then complete the given timing diagrams. Be very careful to check which clock edges are active for the given DFFs.

What to Hand In

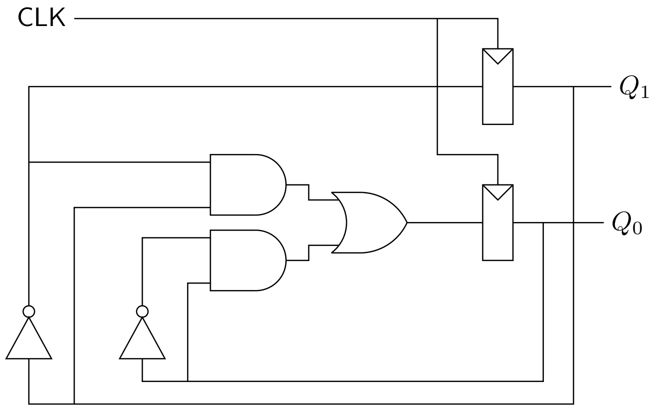
Hand in your completed worksheet.

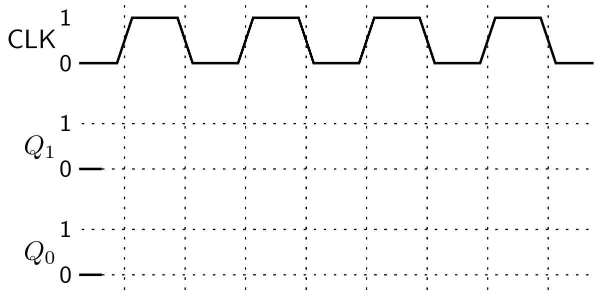
Figure 4: Solution to example DFF problem. Moving left to right, each time you encounter a positive clock edge, use values of signals just *before* the edge to find the values the FF outputs will take on *after* the edge.



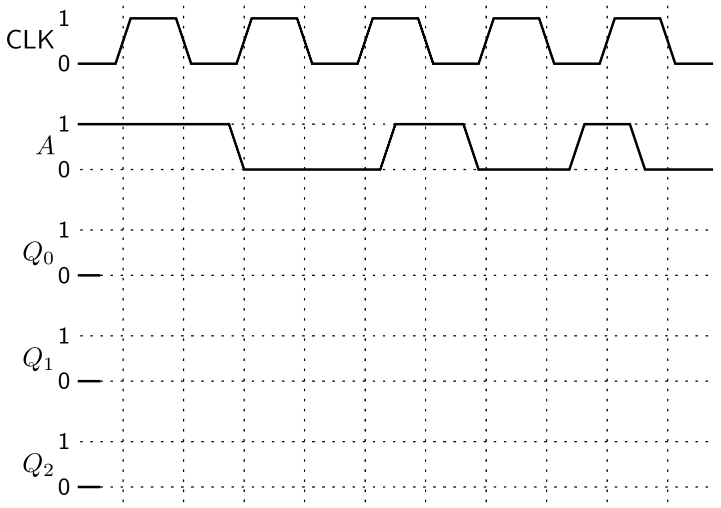
Worksheet for Exercise F

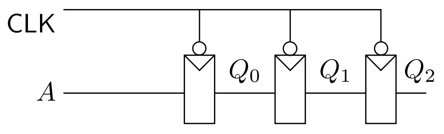
Part I





Part II


CLK
A
Q_0
Q_1
Q_2


CLK
A
Q_0
Q_1
Q_2