ENCM 369 Winter 2018 Lab 9 for the Week of March 19

Steve Norman
Department of Electrical & Computer Engineering
University of Calgary

March 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 Date

The Due Date for this assignment is 3:30pm Friday, March 23.
The Late Due Date is 3:30pm Monday, March 26.

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	8 marks
B	5 marks
C	5 marks
D	2 marks
TOTAL	20 marks

How to package and hand in your assignments

Please see the Lab 1 instructions.

Exercise A: Sequential logic timing in a pipelined processor

So you run and you run to catch up with the sun but it's sinking
Racing around to come up behind you again ...

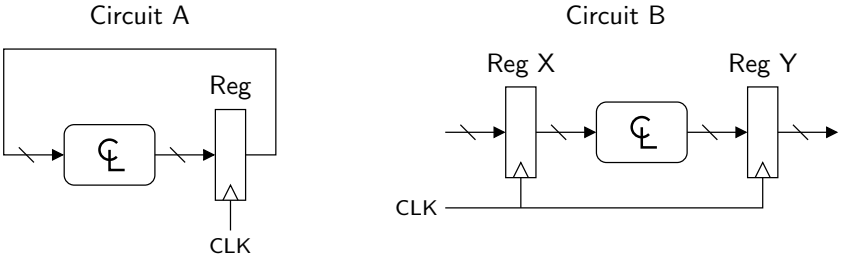
—lyrics from the excellent (but not totally encouraging) Pink Floyd song "Time", which, although certainly not written with synchronous logic design in mind, definitely has something to say about it.

Read This First

This exercise continues the timing analysis work started in Lab 8 Exercises A and B.

Circuit A in Figure 1 is a generalized circuit that describes both a simple counter like the one of Lab 6 Exercise G, Part I and Lab 8 Exercise A, Part II, and also a larger circuit such as the PC update logic for the single-cycle processor shown in Figure 7.11 in the course textbook.

Figure 1: In these two circuits, the symbol ℄ is used to indicate some unspecified combinational logic circuit, which could be a simple gate like an inverter, or could be a cascade of large, complex combinational elements such as adders, ROMs, multiplexers, an ALU, and so on. Reg, Reg X, and Reg Y are all parallel registers made up of one or more positive-edge-triggered D flip-flops with common clock inputs. The circuits appear to have different structures, but timing analysis is the same for both circuits.



Circuit B in Figure 1 is a generalization that describes some important circuits we are studying in this course:

- Consider the path from PC to destination GPR for an `LW` instruction in the single-cycle computer of textbook Figure 7.11. The PC is Reg X and the destination GPR is Reg Y. (This is somewhat confusing because it involves conceptually putting the read logic of the Register File within the ℄ block, while the destination GPR, also part of the Register File, is not part of the ℄ block. But it is a valid way of thinking about timing issues for an `LW` instruction.)
- In a pipelined processor, the ℄ block is combinational logic in between two pipeline registers.

Let's consider the problem of determining the minimum safe clock period for Circuits A and B. We'll assume that all the flip-flops in both circuits have identical specifications.

- When are the input signals for the ℄ block guaranteed to be ready? That happens with a delay of t_{pcq} (flip-flop clock-to-Q propagation delay) after a positive clock edge.

- When are the output signals of the ⧏ block guaranteed to be ready? That will be no more than t_{pd} (propagation delay) after the input signals are ready. Of course, determining t_{pd} for the ⧏ block may require finding the critical path through that block.
- When do the output signals of the ⧏ block need to be ready for correct update to Reg or Reg Y? That must be at least t_{setup} (flip-flop setup time) in advance of the next positive clock edge.

The above points can be summarized graphically in timing diagrams, as shown in Figure 2. For safe operation of either Circuit A or Circuit B,

$$t_{\text{pcq}} + t_{\text{pd}} \leq T_{\text{C}} - t_{\text{setup}},$$

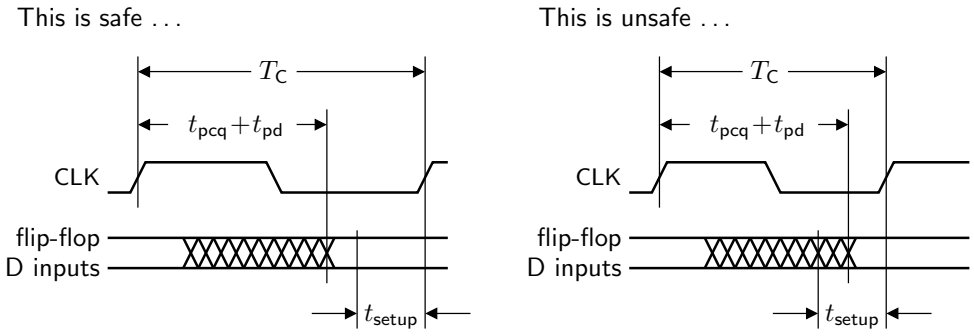
where T_{C} is the clock period. If the circuit design is fixed but you get to choose the clock period, that means

$$T_{\text{C}} \geq t_{\text{pcq}} + t_{\text{pd}} + t_{\text{setup}}.$$

On the other hand, if you are trying to design the ⧏ block to be compatible with given flip-flops and a specified T_{C}, you must ensure that

$$t_{\text{pd}} \leq T_{\text{C}} - t_{\text{pcq}} - t_{\text{setup}}.$$

Figure 2: For reliable updates to flip-flops at the end of a clock cycle, the clock period T_{C} must satisfy $t_{\text{pcq}} + t_{\text{pd}} \leq T_{\text{C}} - t_{\text{setup}}$.



What to Do, Part I

We're going to look at timing constraints for the pipelined processor of Figure 7.47 in the course textbook. We'll assume the timing parameters given in the tables in Figure 3. We'll also assume, as explained in lectures, that the PC, pipeline registers, and Data Memory are updated in response to *positive* clock edges, but the Register File is updated in response to *negative* clock edges.

In this part of the exercise, we'll look at the Fetch stage. We'll concern ourselves only with the update to the F/D pipeline register, because the update to the PC depends partly on work done in the Memory stage, which we'll look at later.

1. Using whatever data you need from Figure 3, determine the shortest safe clock period that will allow the Fetch stage of the Figure 7.47 computer to work correctly.
2. Suppose the desired clock frequency is 3.333 GHz, and suppose it is not possible to reduce t_{pcq} or t_{setup} for the PC or the pipeline registers. One of the components in the Fetch stage will have to have its t_{pd} reduced. Which one is it, and what is the maximum allowable t_{pd} for that component?

Figure 3: Timing parameters for Exercise A. Data for the Register File is omitted because Register File timing is complicated by use of negative-edge triggering for GPR updates in the pipelined designs. *Be aware that this exercise uses different timing parameters and a slightly different timing model from what is presented in Section 7.5.5 of the textbook!*

component	t_{pd}
Instruction Memory	265 ps
Control Unit	145 ps
Register File	**n/a**
Sign-extend	25 ps
Multiplexer	36 ps
ALU	160 ps
Data Memory	275 ps
Adder	90 ps
Shift left 2	0 ps
AND gate	15 ps

component	t_{pcq}	t_{setup}
PC	19 ps	28 ps
pipeline registers	19 ps	28 ps
Register File	**n/a**	**n/a**
Data Memory	**n/a**	42 ps

What to Do, Part II

Let's move on to the Execute stage of the Figure 7.47 computer. (We've skipped the Decode stage, but we'll take care of that later.)

1. Using whatever data you need from Figure 3, determine the critical path through the Execute stage. What is the overall t_{pd} for the Execute stage?

2. Explain why it is not necessary to modify any part of the Execute stage to allow a clock frequency of 3.333 GHz.

What to Do, Part III

Now let's consider the Memory stage of the Figure 7.47 computer. Note that this stage updates the M/W pipeline register, and also plays a role in updating the PC.

1. What is the longest possible delay from a positive clock edge to the point in time when the input to the PC is stable? What does that say about the minimum clock period for the computer? Note that answering these questions involves looking at the AND gate in the Memory stage and also the adder and the multiplexer in the Fetch stage.

2. Determine the minimum clock period needed to allow safe operation of the Memory stage.

3. As in Part I, suppose it is not possible to reduce t_{pcq} or t_{setup} for the PC or the pipeline registers. What improvement must be made in the Memory stage to allow a clock frequency of 3.333 GHz.

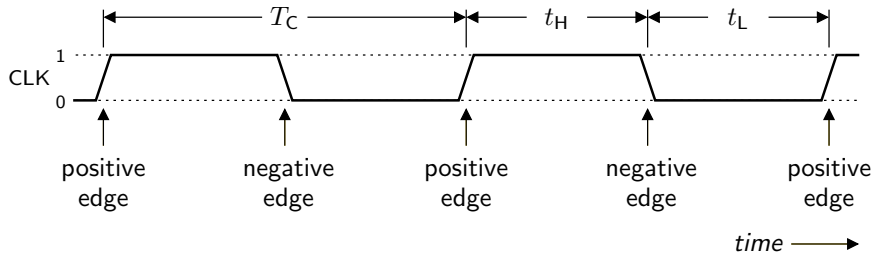
What to Do, Part IV

Thinking about timing issues for the Decode and Writeback stages is somewhat complicated by the fact that a GPR update within the Register File happens in response to a *negative* clock edge.

To keep things relatively simple we'll assume that if the clock period is T_{C}, negative clock edges occur $0.5T_{\text{C}}$ after positive clock edges, as shown in Figure 4.

Suppose that a Register File with the following properties has been designed for us to use.

Figure 4: A clock signal. It's possible to design electronics with $t_H \neq t_L$ but to keep things relatively simple in this exercise we'll assume that, as you see below, $t_H = t_L = 0.5T_C$.



- t_{setup} for writes to the Register File is 31 ps in advance of a *negative* clock edge.
- The RD1 and RD2 outputs are guaranteed to be ready no later than 117 ps after a negative clock edge.

  To see why it makes sense to measure this delay from the negative edge of the clock, consider this example instruction sequence, which presents the toughest kind of challenge for timing within the Register File:

```
add   $t0, $t1, $t2
lw    $t3, 0($t4)
sw    $t5, 0($t6)
sub   $t7, $t7, $t0
```

  The `$t0` value read in the Decode step of `sub` is written earlier in the same clock cycle by the Writeback step of `add`. The combinational logic that is supposed to copy the `$t0` value to RD2 has to wait for the updates to the flip-flops belonging to `$t0`; those updates happen in response to a negative clock edge.

The questions for you to answer are:

1. What is the minimum clock period T_C that will allow reliable operation of the Writeback stage? (For Writeback to work, the A3 and WD3 inputs of the Register File must meet a t_{setup} constraint in advance of a *negative* clock edge.)
2. What is the minimum clock period T_C that will allow reliable operation of the Decode stage? (For Decode to work, all inputs to the D/E register must meet a t_{setup} constraint in advance of a *positive* clock edge.)
3. Using your answers from the above two questions, is there anything about the designs of the Writeback and Decode stages that would prevent use of a 3.333 GHz clock?

What to Do, Part V

Let's draw some conclusions regarding the system as a whole, using answers from Parts I to IV.

1. Given the timing parameters of Figure 3 and the timing model for the Register File given in Part IV, what is the minimum clock period to allow reliable operation of *all five* pipeline stages?

2. Suppose that 3.333 GHz is the desired clock frequency, and suppose also that it is not possible to change any t_{pcq} or t_{setup} values. Which components in the system need to be modified to work faster, and what would the new timing parameters have to be for those components?

What to hand in

Hand in well-organized calcuations and clear answers to questions for Parts I to V.

Exercise B: `beq` in textbook Figure 7.47

Read This First

Note that the processor of textbook Figure 7.47 makes an attempt to handle `beq` instructions, but *does not* handle them correctly. In this exercise, you are asked to determine what exactly the processor does with a `beq`, *not* to guess based on what you think *should* happen.

What to Do

Consider this program fragment, running in the processor of Figure 7.47 with a clock period of 0.5 ns (as in Lab 8, Exercise C):

instruction address	*instruction*	*disassembly*
0x0040_0090	0x8e12_0000	`L1: lw $18, ($16)`
0x0040_0094	0x0211_8020	`add $16, $16, $17`
0x0040_0098	0x0000_0000	`nop`
0x0040_009c	0x0000_0000	`nop`
0x0040_00a0	0x0000_0000	`nop`
0x0040_00a4	0x1240_fffa	`beq $18, $0, L1`
0x0040_00a8	0x0319_4022	`sub $8, $24, $25`
0x0040_00ac	0x0319_482a	`slt $9, $24, $25`
0x0040_00b0	0x0319_5024	`and $10, $24, $25`
0x0040_00b4	0x0319_5825	`or $11, $24, $25`

Suppose that the Fetch stage for the `beq` instruction starts at $t = 45.0$ ns, and suppose that the `lw` instruction has put a value of zero into `$18`.

Answer the following questions. As in Lab 8, Exercise C, use hexadecimal notation for 32-bit numbers, and explain how you got your answers.

1. At $t = 45.5$ ns, what gets written into the PC?
   Shortly after $t = 45.5$ ns, what are the values of InstrD and PCPlus4D?

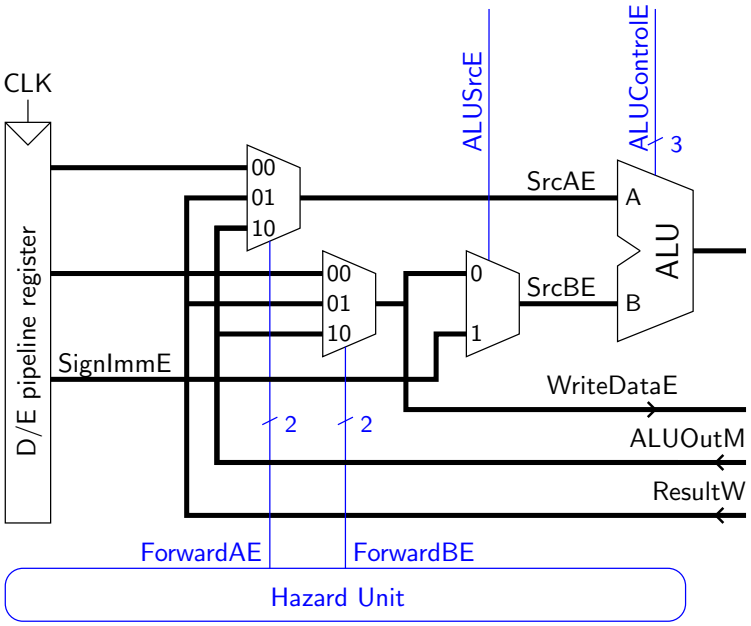
2. At $t = 46.0$ ns, what gets written into the PC?
   Shortly after $t = 46.0$ ns, what are the values of InstrD and PCPlus4E?

3. At $t = 46.5$ ns, what gets written into the PC?
   Shortly after $t = 46.5$ ns, what are the values of InstrD, PCBranchM, and ZeroM?

4. At $t = 47.0$ ns, what gets written into the PC?
   Shortly after $t = 47.0$ ns, what is the value of InstrD?

5. At $t = 47.5$ ns, what gets written into the PC?
   Shortly after $t = 47.5$ ns, what is the value of InstrD?

Figure 5: For Exercise C, diagram to clarify which are the "A" and "B" inputs of the ALU, and to highlight the multiplexers added for forwarding in the Execute stage in textbook Figure 7.50. See the textbook figure for full details, including the *inputs* to the Hazard Unit.



Exercise C: Forwarding details

Read This First

The point of this exercise is to help you understand the forwarding logic presented in Figure 7.50 in your textbook.

Figure 5 shows the outputs of the Hazard Unit, and the multiplexers that guide the correct source data into the ALU.

For this exercise, we'll assume that the Control Unit design has been extended to support the `addi` instruction. It's explained in Section 7.3.3 of the textbook how that can be done. (Section 7.3.3 refers to the single-cycle computer, but the same extension works for the pipelined computer of Figure 7.50.)

What to Do, Part I

There are four data hazards in the following instruction sequence:

```
sw      $0, 0($29)      # line  1
sw      $0, 4($29)      # line  2
sw      $0, 8($29)      # line  3
add     $25, $16, $17   # line  4
sub     $24, $18, $25   # line  5
addi    $23, $25, 1     # line  6
lw      $10, ($24)      # line  7
addi    $24, $24, 4     # line  8
sw      $10, ($23)      # line  9
```

The first hazard is the use of the first `add` result as a source in the `sub` instruction of line 5. Here is a detailed description of how this hazard is managed by the

forwarding unit:

> During the Execute stage of `sub`, the Hazard Unit detects that RsE (11001_{two} for `$25`) matches WriteRegM (also 11001_{two} for `$25`) and that RegWriteM=1. So it sets ForwardBE=10 so that ALUOutM (the first `add` result) is passed to the "B" input of the ALU.

Identify the other three data hazards. For each hazard, give details of how the Hazard Unit solves the hazard, using the above example as a model.

What to Do, Part II

There is an obvious data hazard in the following instruction sequence:

```
lw      $10, ($8)
sw      $10, ($9)
```

This kind of code is very common when data is being copied from one memory location to another memory location.

Consider the circuit of Figure 7.50 in the textbook. For the above instruction sequence, does the circuit

- handle the hazard properly using only forwarding;
- handle the hazard properly using a combination of stalling and forwarding;
- fail to handle the hazard properly, storing an out-of-date `$10` value;
- or fail in a different way, storing some other wrong value?

Give a brief explanation for your answer—just a few short sentences.

What to Hand In

Hand in your answers to Parts I and II.

Exercise D: Avoiding branch instructions

Read This First

One of the conclusions that can be reached from reading the material titled "Solving Control Hazards" starting on page 421 of the textbook is that branch instructions are potentially very expensive. If there is no branch prediction, or if branch prediction is present but incorrect, a branch instruction may cause a stall of several clock cycles.

As a result, modern compilers try to avoid generating branch instructions when possible.

Here is an example C code fragment from lectures on solutions to control hazards:

```
/* Count negative elements in an array of ints. */
do {
  if (*p < 0)
    count++;
  p++;
} while (p != past_end);
```

MIPS code I thought of for this fragment, aimed at the real MIPS instruction set with delayed branches, is

```
L1:   lw    $t0, ($a0)
      slt   $t1, $t0, $zero
      beq   $t1, $zero, L2
      addiu $a0, $a0, 4
      addiu $t9, $t9, 1
L2:   bne   $a0, $t8, L1
      nop
```

However, when I gave the C code to gcc 4.8.3 for MIPS, it cleverly came up with code like this:

```
L1:   lw    $t0, ($a0)
      addiu $a0, $a0, 4
      slt   $t1, $t0, $zero
L2:   bne   $a0, $t8, L1
      addu  $t9, $t9, $t1   # add slt result to count
```

Notice that the loop is reduced from seven instructions to five, and there is no possibility of a stall around the decision on `*p < 0`. Note also that the update to `count` is in the delay slot of `bne`, so the update is part of the loop, and that `p++` got moved to avoid a stall due to the data hazard of loading to `$t0` and then using `$t0` as source in `slt`.

Read This Second

Modern instruction sets offer various ways to do some things conditionally without using a branch instruction. For example, MIPS has "conditional move" instructions called `movz` and `movn`:

```
movz  $t0, $t1, $t2 # if ($t2 == 0) $t0 = $t1
movn  $s0, $a0, $a1 # if ($a1 != 0) $s0 = $a0
```

The operands can be any GPRs—the ones used above are just examples.

What to Do

Consider this C code:

```
do {
  if (*p > 0)
    sum += *p;
  p++;
} while (p != past_last);
```

Using ideas from "Read This First" and "Read This Second", show how this loop can be coded in the real MIPS instruction set without using a branch instruction for the `if-else` statement, and without any `nop` instructions. Assume that `p` and `past_last` are of type pointer-to-`int` in `$a0` and `$a1`, and that `sum` is an `int` in `$t0`.

This is a pencil-and-paper exercise—you do not need to make it work in MARS.

What to Hand In

Hand in your code.