EECS 151/251A Homework 7

Instructor: Prof. John Wawrzynek, TAs: Christopher Yarp, Arya Reais-Parsi

Due Monday, Apr 8th, 2019

Problem 1: Fast Unsigned Integer Comparison [15 pts]

Consider the design of a combinational logic circuit for comparing unsigned integers. The circuit accepts two n-bit unsigned integers, A and B, and generates an output, eq, that is equal to 1 iff A and B are equal, and another output, lt, that is equal to 1 iff A is less than B. We would like the circuit to have relatively small delay for large n, so you need take an approach that will lead to delay scaling with log(n). One solution would be to use a fast subtractor circuit, but that is not the correct answer here. *Hint: Divide the input words in half and think about defining the function recursively.*

Sketch out enough of your circuit so that we understand your approach and detailed circuits.

Solution:

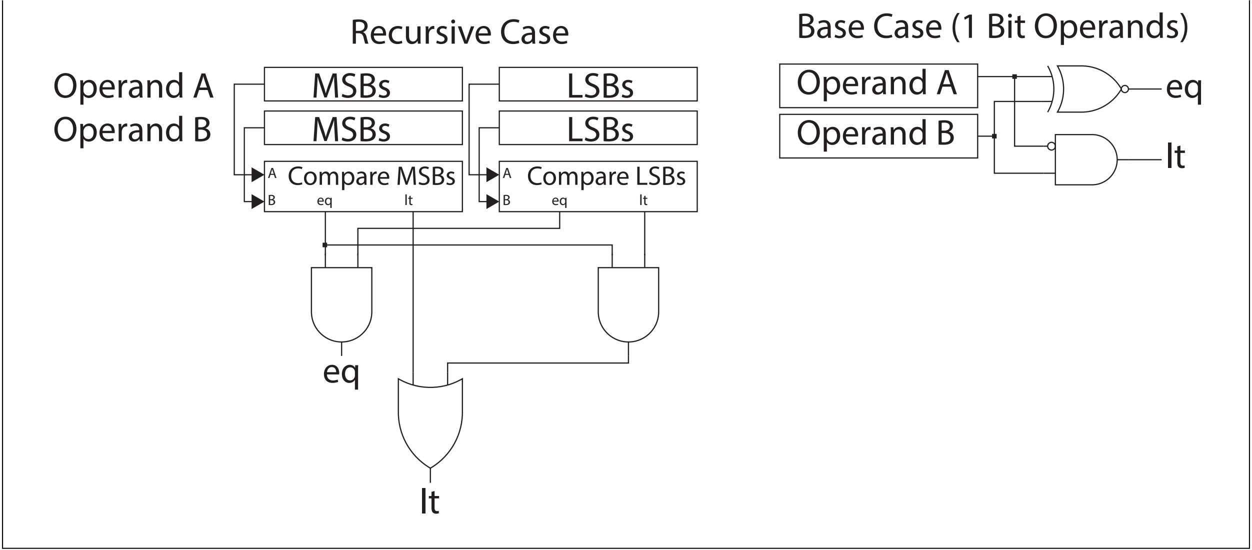
Unsigned numbers have a property that allows us to split the comparison operation into 2 parts. We can the numbers we are comparing into their MSBs and LSBs. Let's say operand A is split into $\{A_{msb}, A_{lsb}\}$ and operand B is split into $\{B_{msb}, B_{lsb}\}$ with A_{lsb} having the same number of bits as B_{lsb}.

If $A_{msb} < B_{msb}$ then we know that $A < B$. The LSBs cannot change this conclusion because the LSBs combined contribute less that the least significant bit of the A_{msb} and B_{msb} . For $A_{msb} < B_{msb}$, they must differ by at least the least significant bit of A_{msb} and B_{msb}. Likewise, $A_{msb} > B_{msb}$ then we know that $A > B$.

It is only in the case where $A_{msb} = B_{msb}$ that we need to look at the LSBs. In this case, if $A_{lsb} < B_{lsb}$ then we know that $A < B$. If $A_{lsb} = B_{lsb}$ then we know $A = B$.

Therefore, we can construct the unsigned comparison recursively by splitting the operands and using 2 comparison units, each of which is operating on a smaller word width. To have $O(logn)$ propagation delay, the operands should be split into equal parts in each stage.

The base case is the comparison of 2 single bit operands. In this case, the equivalence check is an XNOR gate. The LT check is $A'B$.



Problem 2: Extending RISC-V [8 pts]

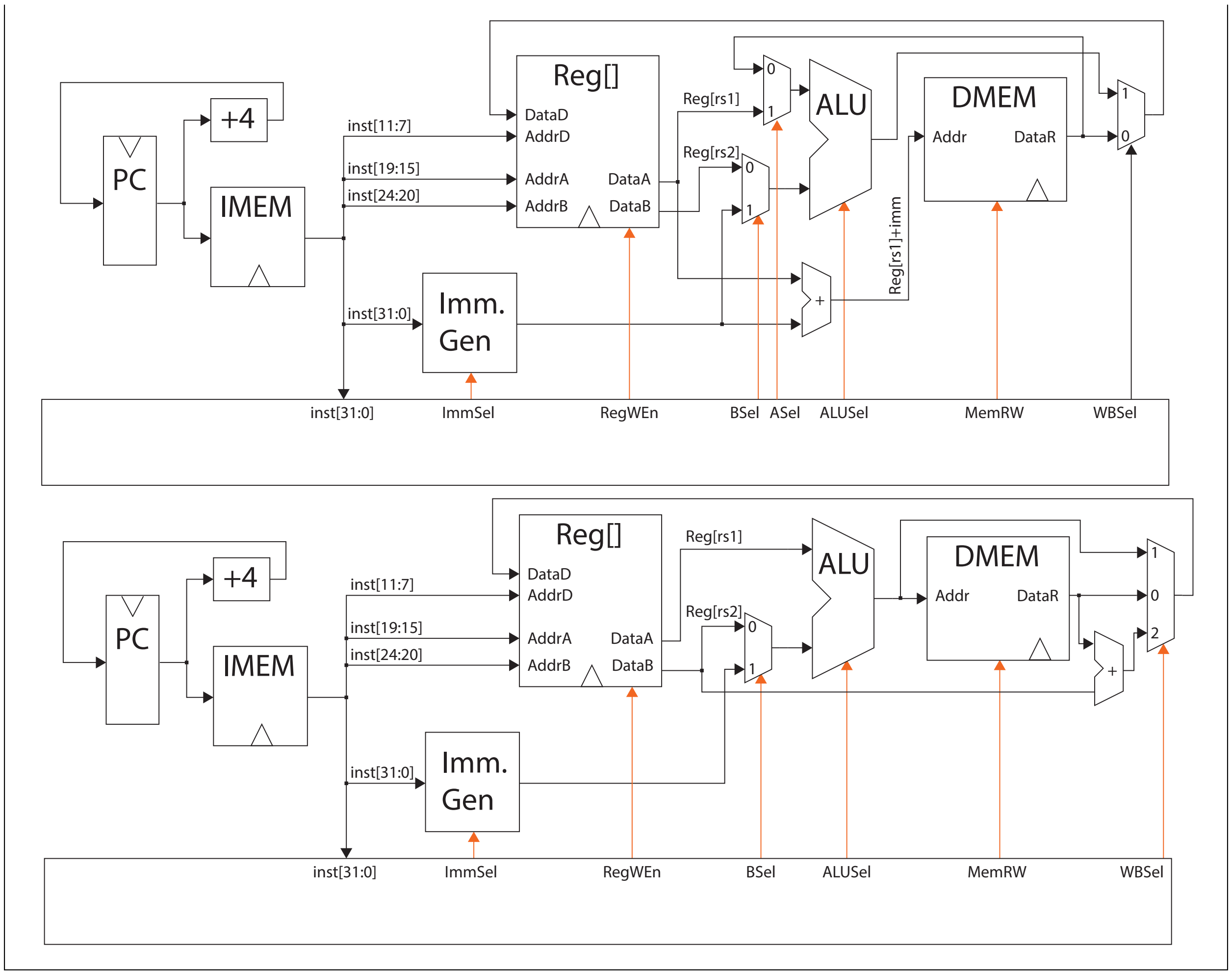
Based on the RISC-V datapath presented in lecture that included up to the `lw` instruction, draw a modified single cycle datapath that could execute a new add instruction named `addm` which takes one of its operands from memory and has the following behavior:

```
Reg[rd] <- Reg[rs2] + DMEM[ Reg[rs1] + offset ]
```

You don't need to describe how the instruction would be encoded in the instruction word.

Solution:

There are several options for adding this instruction. One method involves inserting a new adder which calculates memory addresses based on rs1 and the immediate. The value from memory is then fed back to the ALU to be added to rs2. Alternatively, the ALU can be used to compute the memory address. A separate adder is then inserted which takes the memory value and adds it to rs2.



Problem 3: 3 Stage RISC-V Pipeline Branching [8 pts]

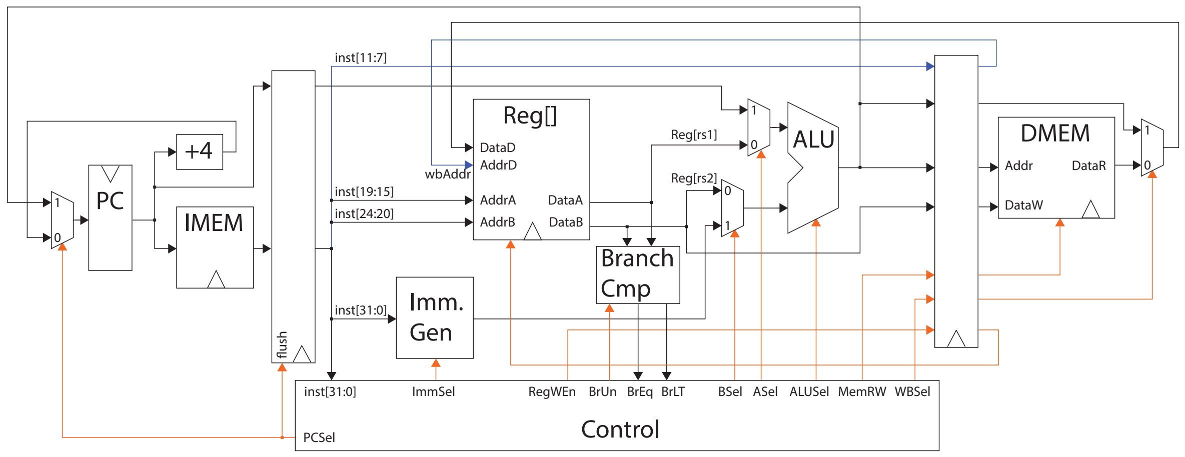
Based on the RISC-V single cycle datapath in the lecture slides, draw a simplified version of the datapath that includes just those components necessary for the branch instructions.

Now, draw another version of the datapath with modifications necessary to permit the 3-stage pipelining scheme with the "predict 'not taken' " strategy described in lecture. You don't need to show the details of how to kill an instruction. However, remember that in this approach only one instruction is killed when a branch is taken. For this part assume that the instruction memory has *asynchronous read*.

Solution:

Depending on your implementation, you may decide to merge the ID stage of the 5 stage pipeline into either the IF or X stage. Either will work. In the case of merging ID with the IF stage, it is important to pipeline the relevant portions of the instruction to be used in the X stage for control.

In the example shown below, the next PC is assumed to be PC+4. This predicts that a branch will not be taken. The branch comparison happens in the X stage and the branch address is muxed into the PC. At this point, one incorrect instruction has been fetched and is sitting at the input to the IF/X pipeline register. This instruction must be converted into a NOP. before it is allowed to progress through the pipeline.

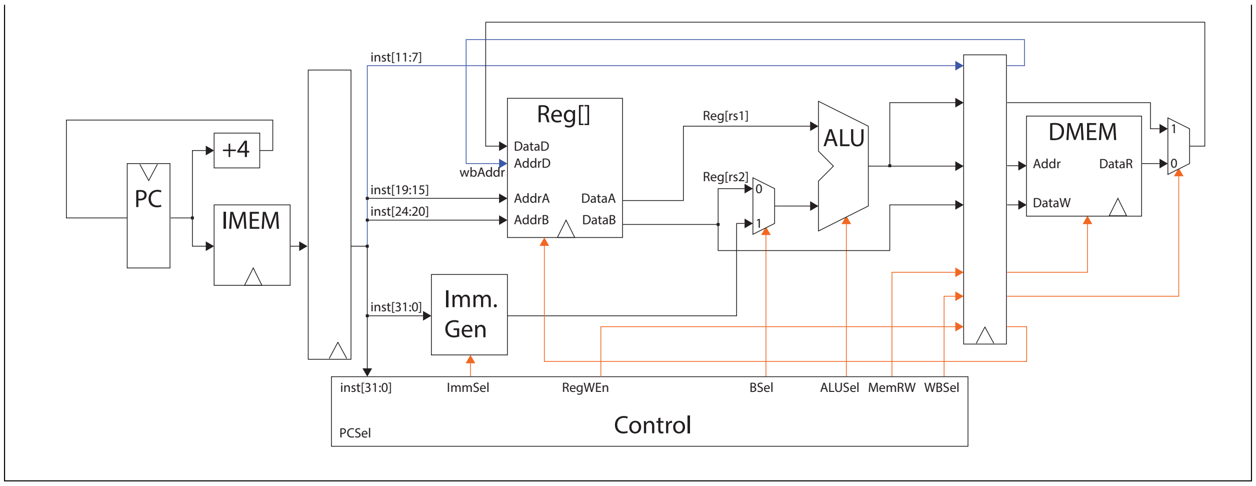


Problem 4: 3 Stage RISC-V Pipeline [8 pts]

Using the RISC-V datapath presented in class that includes the `lw` instruction, draw a new version of the datapath that is modified to permit the 3-stage pipelining scheme described in lecture. Remember that the load delay should be only 1 cycle. You do not need to show the details of instruction stall in the case of an instruction dependent on the load. Assume that both the IMEM and DMEM have asynchronous read.

Solution:

A pipeline stage can be inserted between the ALU and the DMEM. As compared to the 5 stage pipeline, the implementation shown below merges the WB stage into the M stage. The address is calculated in the X stage and is passed to the M stage on the following positive clock edge. The memory is then read and the value is fed back to the register file along with the pipelined register write address and register write enable lines. The value is registered on the next positive clock edge, resulting in a 1 cycle latency for the load operation



Problem 5: 3 Stage RISC-V Pipeline with Synchronous Memory [6 pts]

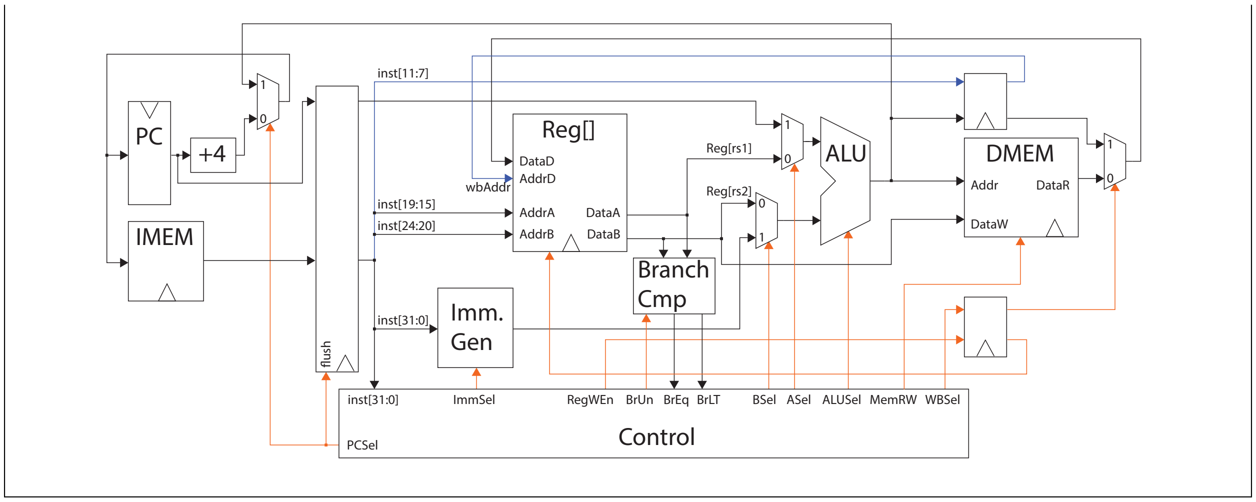
Now consider how your answers to the previous two problems would change if the memories we used for IMEM and DMEM had *synchronous read* instead of *asynchronous read*.

Synchronous read is defined as follows. *On the rising edge of the clock, the read address is registered on the read port and the stored data is then sent to the output.* We assume the read address hold time is short (similar to registers), and the access time is long compared to the clock period. (That is the reason we dedicate an entire pipeline stage to instruction fetch and to data memory access.)

Solution:

Synchronous memories register the read/write address on the positive edge of a clock cycle and then proceed to perform the read or write action as requested. Their impact on timing is very similar to that of a register with actions only taking place after a positive clock edge. One method to deal with this increased latency is to relocate the memories to be in line with a pipeline register. This is the technique used below with DMEM which was moved in line with the X/M register.

The instruction memory is a little trickier. We could move it to be in line with the IF/X pipeline register. However, this would result in a long critical path which would include the instruction memory, the register file (read), and the ALU. Because of this, we will keep the IMEM completely in the IF stage. Proceeding with an unmodified IF stage would unintentionally add another pipeline stage to the design (between the PC and the IMEM). This would require 2 instructions to be killed for each branch mispredict or jump. However, IMEM can be placed in line with the PC with the target address (PC+4 or branch/jump address) fed into the IMEM address port. With this configuration, PC will output the address of the instruction at the output of IMEM.



Problem 6: 3 Stage RISC-V Pipeline Bypassing [5 pts]

We have a 3-stage pipelined RISC-V datapath that passes the 32-bit instruction through the pipeline. The instruction register separating the I and X stages is $IReg_{IX}$ and between the X and M stages is $IReg_{XM}$. As shown in page 63 of the lecture notes, the ALU has a bypass mux which allows ALU data hazards to execute without any delay. Describe the logic needed in the controller for correct operation in the case of back to back r-type instruction execution. You do not need to show detailed circuit diagrams, but describe precisely what operation(s) need to be done.

Solution:

Let's say that we have two R-type instructions, A and B, B immediately following A. We'll look at the case where bypassing is sometimes required: when an R-type instruction (A) is in the M stage and a R-type instruction (B) is in the X stage. Bypassing (or forwarding) a result from the $IReg_{XM}$ register back to one (or both) of the ALU input ports is only required if $rs0$ or $rs1$ of B are equal to rd of A. This is because the result of A has not yet been written back into the register file and will not be available during the X stage of B.

If $rs1_B = rd_A$, then the value from $IReg_{XM}$ must be bypassed to the $rs1$ input of the ALU by setting the corresponding mux select line. If $rs1_B \neq rd_A$, then the mux corresponding to the $rs1$ input of the ALU is set to allow the value read from the register file (or the PC of B) to be passed to the ALU.

Likewise, if $rs2_B = rd_A$, then the value from $IReg_{XM}$ must be bypassed to the $rs2$ input of the ALU by setting the corresponding mux select line. If $rs2_B \neq rd_A$, then the mux corresponding to the $rs2$ input of the ALU is set to allow the value read from the register file (or the immediate) to be passed to the ALU.

Problem 7: Line Drawing Accelerator [20 pts]

Draw a block diagram for the design of a datapath and controller for an implementation of the line-drawing accelerator presented in class. For this problem, you may assume that $x_0 < x_1, y_0 < y_1$, and slope is $\leq 45°$.

Try to minimize the number of cycles per output result.

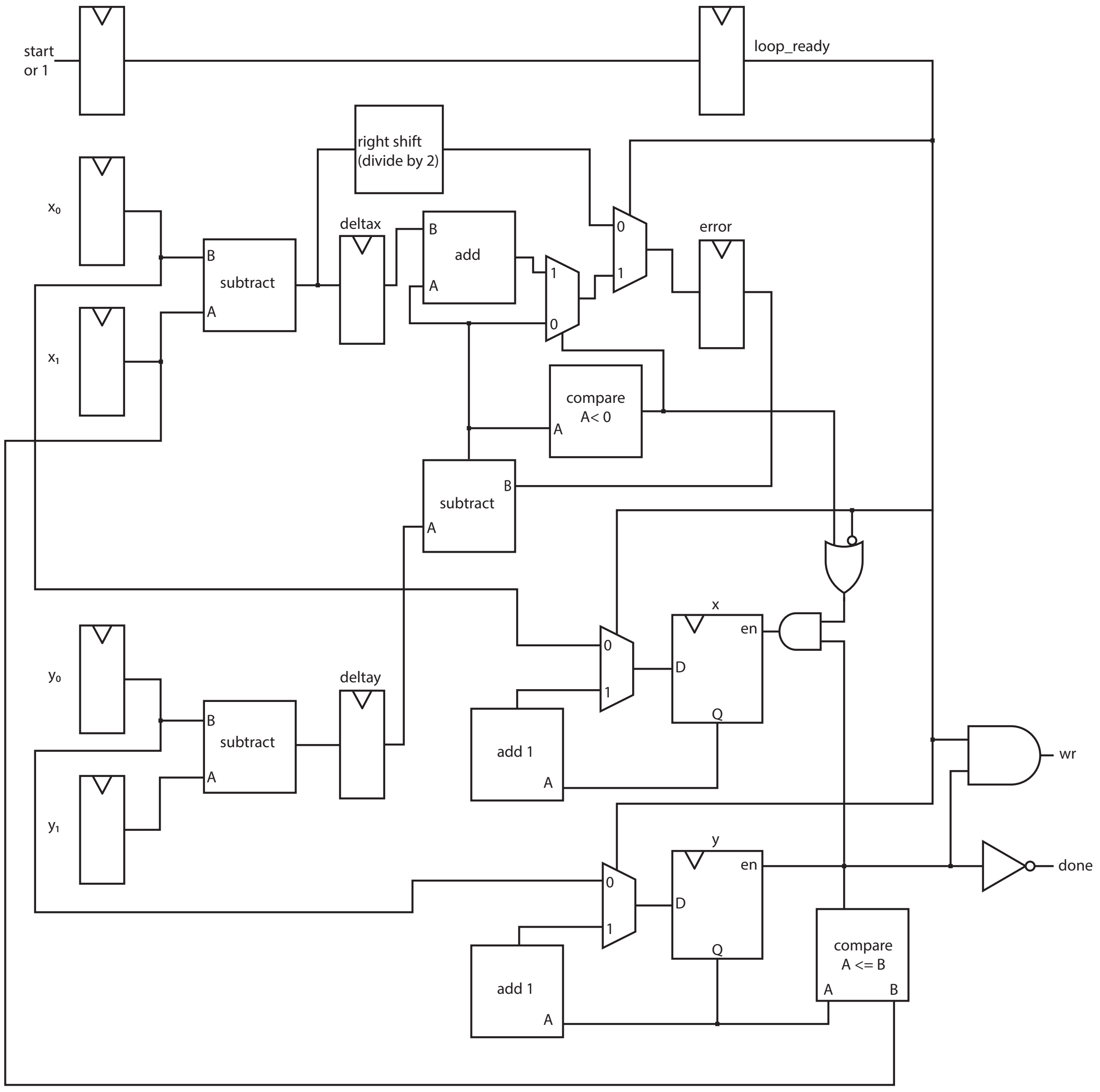
You don't need to show your design to the gate level and may use components at the level of arithmetic blocks, comparator blocks, muxes, registers, etc. Make sure you show the details of how the looping is controlled.

Assume that the x_0, x_1, y_0 and y_1 values are available in registers. The outputs from the accelerator should be a signal, wr, that indicates when it is time to write a pixel, and x, y which define the pixel location to write.

Solution:

One solution is shown below. After an initial set-up cycle, this solution will produce a result every cycle until complete.

Note however that this solution is not optimal; the critical path is quite long. A more optimal solution would introduce additional registers and use pipelining to reduce the clock period.



The accelerator has a set-up stage and a loop stage. Control between the two is managed using a `loop_ready` signal, which is 0 in the first cycle and 1 in cycles following the first: a `start` signal - or just a plain 1 if we assume that the accelerator is already "started" - is shifted into the `loop_ready` register by the time the set-up completed. All registers are set to 0 on reset. We make use of the enable pins on the `x` and `y` registers when their value would be unchanged in the algorithm.

On the first cycle, `deltax`, `deltay` and `error` are computed and stored in registers.

On every cycle of the loop;

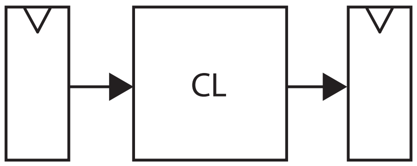
- as long as the value in `x` is less than or equal to `x1`, `wr` is high to signal a write at the current `x` and `y` values;

- `x` is incremented;
- `y` is incremented only when `error` less `deltay` is negative;
- `error` takes the value of `error - deltay` or `error - deltay + deltax`, depending on whether the former is negative.

The loop ends when `x` is greater than `x1`, at which point both the `x` and `y` registers are disabled. Note that as drawn, the `error` register must also be disabled to stop needless computation beyond completion. An elegant way to do this is to reset the `loop_ready` registers, which would prepare the circuit for the next invocation with new parameters.

Problem 8: Energy Efficiency Improvements [10 pts]

You are given the circuit shown below with $\tau_{CL} = 16ns$, and $\tau_{setup} = \tau_{clk-Q} = 1ns$.



On average, at some V_{dd} the energy for one data item passed through the combinational logic block is 1 Joule. The registers each consume 0.1 Joule on average for each new data word stored.

Your application for this circuit requires results be computed at a rate of 50MHz (one result per cycle). Also, for this application, the latency from input to output is not important.

Assume the combinational logic block can be split evenly (in terms of both delay and energy) into multiple blocks.

Devise a scheme that would improve the switching energy efficiency while meeting the application requirements. Compare the switching energy per result of the original circuit and your new one.

Assume that a $1/n$ reduction in clock frequency can accommodate a $1/n$ reduction in V_{dd}.

Solution:

The given solution is running slightly faster than is required by the critical path at the specified voltage level (2 ns positive slack for 50 MHz clock). Increasing the clock period by a factor 1/(18/20) would allow Vdd to be decreased by a factor 18/20.

Since switching power is $P_{\text{SW}} = \frac{1}{2}\alpha C V_{dd}^2 F$, the power ratio between version, when F and V_{dd}

are scaled by k is:

$$\frac{P_{\text{SW new}}}{P_{\text{SW orig}}} = \frac{V_{\text{dd new}}^2 F_{\text{new}}}{V_{\text{dd orig}}^2 F_{\text{orig}}} = \frac{(kV_{\text{dd orig}})^2 kF_{\text{orig}}}{V_{\text{dd orig}}^2 F_{\text{orig}}} = k^3$$

However, changing the clock frequency by a factor of k changes the amount of time it takes for the operation to complete by a factor of 1/k. Since Energy = Power*Time, the net scaling in energy/operation in the logic is k^2.

For the scaling above, the energy per operation becomes $(1J + 0.1(2)J)(18/20)^2 = 0.972J$ compared to $1.2J$ in the original.

A more substantial energy efficiency improvement can be gained by pipelining the combinational logic. When pipelining, the critical path is reduced which allows the clock rate to be increased without changing V_{dd}. Since this application does not require a clock rate increase, the clock frequency and V_{dd} can both be scaled down from the new clock frequency made possible by pipe-lining. The total amount of time an operation take to get through the pipeline changes to become the number of pipeline stages * the delay through each pipeline stage.

This changes the energy/op scaling factor because power is scaled by k^3 but time/op is now scaled by n/k. This leads to energy/op scaling by nk^2.

Let's divide the combination logic into 2 equal parts (from the perspective of delay) and insert a pipeline register between them. We incur the additional energy use of the pipeline register but can now reduce the combinational logic delay to 8 ns. This leads to a critical path of 10 ns. This allows clock frequency and V_{dd} to be scaled by a factor of 1/2 to reach the target clock frequency of 50 MHz (20 ns period). The total energy per operation of this pipelined version is $(1J + 0.1(3)J)(2)(1/2)^2 = 0.65J$.

This process can be tried again by splitting the combinational logic into n segments with n-1 additional regsiters.

Assuming the combinational logic is evenly split, the critical path when pipelined by an integer factor n is $16/n + 2$ ns. The energy/operation is $(1 + 0.1(1 + n))((16/n + 2)/20)^2 n$

To find when the energy/op is minimum, we can take the first derivative of the above expression and get

$$0.027 - 0.704n^{-2} - 0.02n$$

The only real zero is at $n \approx 4.431$.

The second derivative is:

$$0.002 + 1.408n^{-3}$$

This is positive for $n \approx 4.431$ identifying $n \approx 4.431$ as a local minimum.

Since n must be an integer, we will look at $n = 4$ and $n = 5$

For $n = 4$, the critical path delay is $16/4 + 2 = 4/3$ ns. The energy/operation is $(1 + 0.1(1 + 4))((16/4 + 2)/20)^2(4) \approx 0.54J$.

For $n = 5$, the critical path delay is $16/5 + 2 = 26/5$ ns. The energy/operation is $(1 + 0.1(1 + 5))((16/5 + 2)/20)^2(5) \approx 0.5408J$.

Therefore, the minimum energy/operation is achieved for this circuit when pipelined by a factor of 4 yielding an energy/op of $0.54J$.

Problem 9: Race to Halt [4 pts]

An effective scheme for improving energy efficiency when static power consumption is a significant component of total power consumption is a technique call "race to halt". The basic idea is to run the hardware at maximum speed to quickly compute the necessary set of computations, then turn off the power, thus preventing leakage.

Suppose you have a CPU that runs your application with an average power consumption of 8 Watts, where 50% of the power is dynamic and 50% is static. Your application requires an average of 100 Million operations per second but the CPU is capable of 400M ops/sec. Assume that no other program also running on the CPU.

You would like to determine the most effective way to run your application to preserve the battery life. You have the ability to control the supply voltage (V_{dd}), the clock frequency (f), and if needed can put the CPU into a sleep mode where static power is essentially zero. You consider two schemes.

1. Reduce supply voltage and clock frequency [2 pts].
2. Keep V_{dd} and f unchanged, but use the sleep mode [2 pts].

Which approach will be better at conserving your battery charge? Show your work and justify your answer.

Solution:

1. When varying frequency f and suppy voltage V_{dd}, we assume that the static power usage remains constant. This is more or less not a lie.

   Say that we've halved the supply voltage, $V'_{dd} = \frac{V_{dd}}{2}$, and halved the frequency, $f' = \frac{f}{2}$. Our new dynamic power usage is

$$\begin{aligned} P'_{dyn} &= \frac{1}{2}\alpha C V'^2_{dd} f' \\ &= \frac{1}{2}\alpha C \frac{V^2_{dd}}{4}\frac{f}{2} \\ &= \frac{1}{8}P_{dyn} \end{aligned}$$

   So our dynamic power usage is $\frac{1}{8}$ what it was. Since we know that our average 8W power draw was comprised of half dynamic power usage, we know that our dynamic power usage becomes $P'_{dyn} = 4/8 = 0.5$ watts. The total power draw is now $P'_{tot} = P_{static} + P'_{dyn} = 4 + 0.5 = 4.5W$.

However, to determine what will conserve the most battery, we have to determine the energy usage for our application. Energy is the product of power and time; since we have halved the frequency we can expect that the computation will take twice as long. Now,

$$\begin{aligned} E' &= P'_{tot}t' \\ &= 4.5(2t) \\ &= 9t \end{aligned}$$

Our original circuit took $E = 8t$, leaving t as an unknown parameter. Thus, modifying V_{dd} and f in this manner has increased the total energy usage of the circuit.

2. Since our CPU has the capacity, suppose we do $4\times$ as many operations per second and so sleep three-quarters of the time. Our activity factor α has quadrupled so the new dynamic power usage is:

$$\begin{aligned} P'_{dyn} &= \frac{1}{2}(4\alpha)CV_{dd}^2 f \\ &= 4P_{dyn} \\ &= 16\text{W} \end{aligned}$$

Since we can sleep the circuit $\frac{3}{4}$ of the time, the static power consumption *and* dynamic power consumption become 0 when the circuit is not running. Out new total energy usage is:

$$\begin{aligned} E' &= (P'_{dyn} + P_{static})\frac{t}{4} \\ &= 5t \end{aligned}$$

This scheme greatly reduces the overall energy usage of our application.

Problem 10: Memory [9 pts]

(a) Suppose you want to design a 1-Byte wide memory block with a capacity of 2K Bytes of storage (remember 1K = 1024). We would like to have the core of the block square (equal number of rows and columns). How many total address bits are needed for this memory? How many address bits are used by the row-decoder? How many address bits are used by the column-decoder?

(b) Now you want to design the row decoder using the predecoder technique presented in lecture. Try two difference approaches. The first approach can use only gates with no more than 2-inputs. The second scheme can use some gates with 4-inputs.

Map out each scheme and describe the design of each of these decoders.

Solution:

1. Our memory needs to have $2048 = 2 \times 1024 = 2 \times 2^{10} = 2^11$ bytes of memory. Each byte is 2^3 bits. Thus in total our memory must have $2^{11} \times 2^3 = 2^{14}$ bits of memory.

   To arrange as a square, note that the square root $\sqrt{2^{14}} = 2^7$. Thus we need 2^7 columns and 2^7 rows.

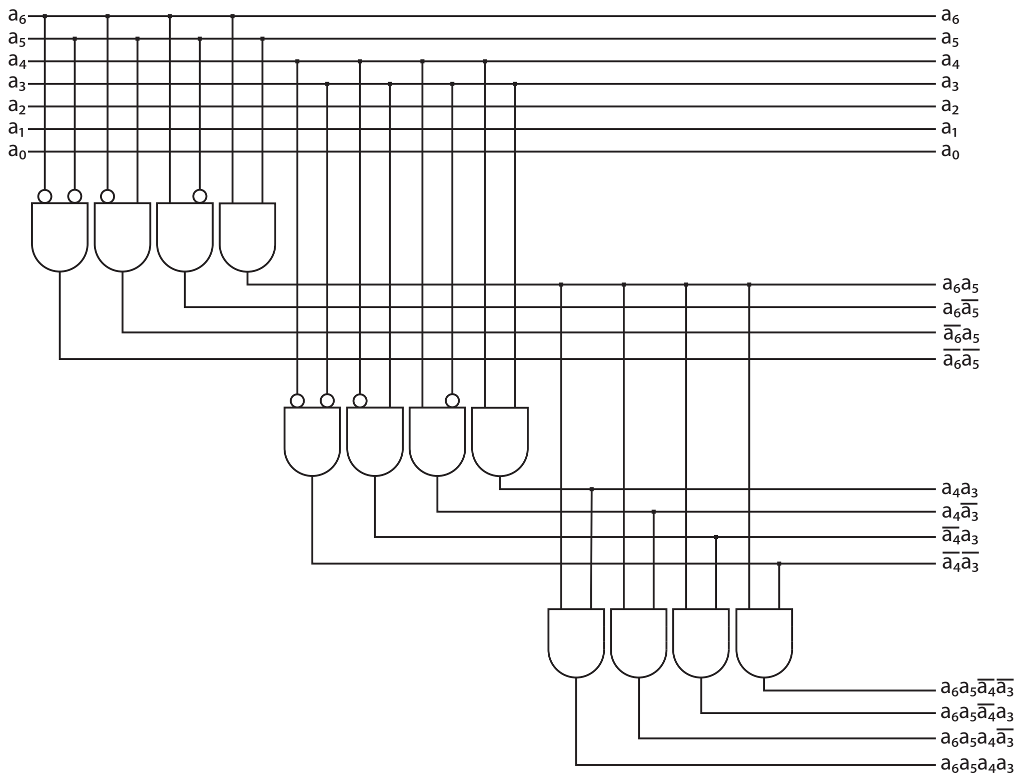
   Since we address bytes, there are only $\log_2 2^{11} = 11$ address bits. Rows use $\log_2 2^7 = 7$ address bits, and this is how many the row decoder uses. Columns only need the remaining 4 bits, since the memory is arranged in groups of bytes (which are 8 bits each).

2. Consider a row address denoted by $a_6a_5a_4a_3a_2a_1a_0$, where each a_i denotes the i-th address bit. As described in the lecture slides, predecoding is the process of grouping address bits and enumerating single signals for each of their possible combinations.

   Assuming we use only 2-input gates, we would group the address bits as

   $$(a_6a_5)(a_4a_3)(a_2a_1)a_0$$

   We would then generate signals for every combination of inputs in the first, second and third groups. As we combine the signals again, we have to use additional 2-input AND gates to combine every possible combination of signals from, for example, the first and second group:



But we have to repeat this for the other groups too, making it rather unweildy.

Assuming we use some 4-input gates, we can group the address bits as

$$(a_6a_5a_4a_3)(a_2a_1)a_0$$

This greatly simplifies our combinations. There will be $2^4 = 16$ gates for the first group and 4 for the second. Then, if we combine the three groups with 2-input AND gates, there will be $16 \times 4 = 48$ between the first group and second group, and another $2 \times 48 = 96$ between each of those and the 2 possible values of a_0.

What if we combined them with 4-input gates, tying one input HIGH and pretending it was 3-input gate? But then we would need $16 \times 4 \times 2 = 96$ gates to complete the connection. But is this such a good idea?

We could use 4-input AND gates for the second group too, grouping bits as $(a_6a_5a_4a_3)(a_2a_1a_0)$. This would require 16 and 8 gates for each group respectively, and 128 AND gates to connect them.

Which of these is best depends on the size and delay of our gates themselves, our budget for area, power, and so on.

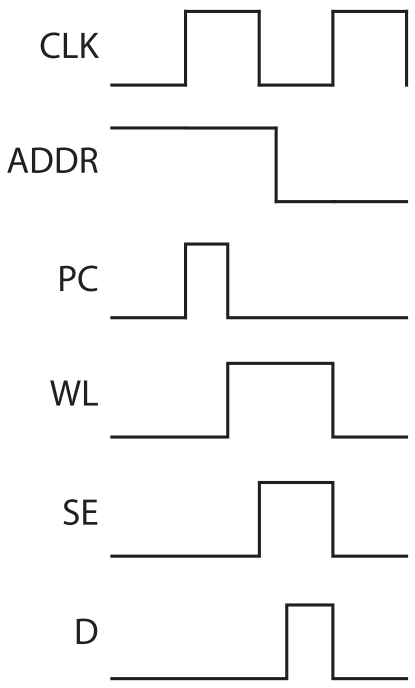
Problem 11: SRAM Waveform [5 pts]

Imagine you are designing the controller for an SRAM block. Draw the waveforms indicating the sequence of operations for a *READ*. The sequence begins when the address is registered on the positive-edge of the clock. Draw the approximate waveforms for all the signals.

1. CLK
2. ADDR (address)
3. WL (word line)
4. PC (precharge)
5. SE (sense amp enable)
6. D (data out)

Solution:

Not much detail is specified about the controller. What's important here is the relative timing of the various signals:



In order:

1. The read address is presented before and then registered (not shown) on the rising edge of the clock, starting the process.
2. Bit lines are precharged, so `PC` is driven high for a period of time.
3. Bit lines are turned off and the word line `WL` is driven high, connecting the value stored in the cell.

4. At the same time or a little after, the sense amps `SE` are enabled.
5. A short time after that, the value in the cell is propagated to the output, `D`.

Problem 12: SRAM vs. DRAM [4 pts]

For each of the following attributes, compare SRAM to DRAM and explain your reasoning.

(a) density (bits stored per unit area)

(b) access time

(c) noise resilience

(d) cost per bit

Solution:

1. DRAM is higher density than SRAM. SRAM takes 6 transistors, DRAM takes 1 transistor and a capacitive element, or 3 transistors in some designs.
2. DRAM is much slower than SRAM. Reading from DRAM requires several more steps and can take on the order of 100 cycles in modern systems.
3. SRAM is more resilient to noise than DRAM. Due to its densities, DRAM typically stores values as very small charges, relative to which any environmental noise is likely more significant than in an SRAM. SRAM stores values in a bistable cross-coupled inverter, which would take significant noise (more than half the high-low logic voltage level difference) to swing.
4. DRAM is much cheaper per bit (now) simply because its densities are much greater. Consider how much SRAM is typically found on a CPU (now on the order of 10 MB) compared to the amount of memory in a typical system (about 16 GB).

Problem 13: DRAM [4 pts]

1-transistor DRAM designs usually include a "row buffer"—a register on the periphery that is used to register an entire row. Explain how this register could be used and why it's a good idea.

Solution:

- It reduces power and increasing memory system speed. RAM accesses exhibit spacial locality to a high degree: it's likely that access to one word in a DRAM row is likely followed by another access to the same row. Buffering the row saves having to read the memory cells again, returning a value to the system faster and using less power.
- For writing: a row is opened (copied into the row buffer) and constituent bytes/words

are updated before the entire buffer is written back.