M A S S A C H U S E T T S I N S T I T U T E O F T E C H N O L O G Y
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.375 Complex Digital Systems
Spring 2006 - Quiz - March 24, 2006
80 Minutes

NAME: _____ SCORE: _____

Please write your name on every page of the quiz.

Not all questions are of equal difficulty, so look over the entire quiz and budget your time carefully.

Please carefully state any assumptions you make.

Enter your answers in the spaces provided below. If you need extra room for an answer or for scratch work, you may use the back of each page but please *clearly indicate where your answer is located.*

A list of useful equations is printed at the end of this quiz. You can detach this sheet for reference and do not have to hand this in. *We will not grade anything written on the equation sheet.*

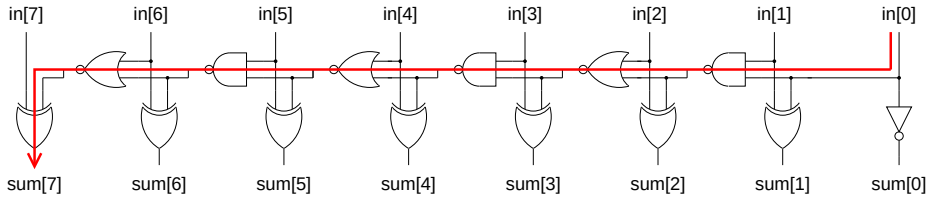
You will also receive a separate handout containing a copy of the relevant Bluespec lecture slides and code. *We will not grade anything written on the Bluespec slides.*

You must not discuss the quiz's contents with other students who have not yet taken the quiz. If, prior to taking it, you are inadvertently exposed to material in a quiz — by whatever means — you must immediately inform the instructor or a TA.

	Points	**Score**
Problem 1	25	
Problem 2	25	
Problem 3	25	
Problem 4	25	

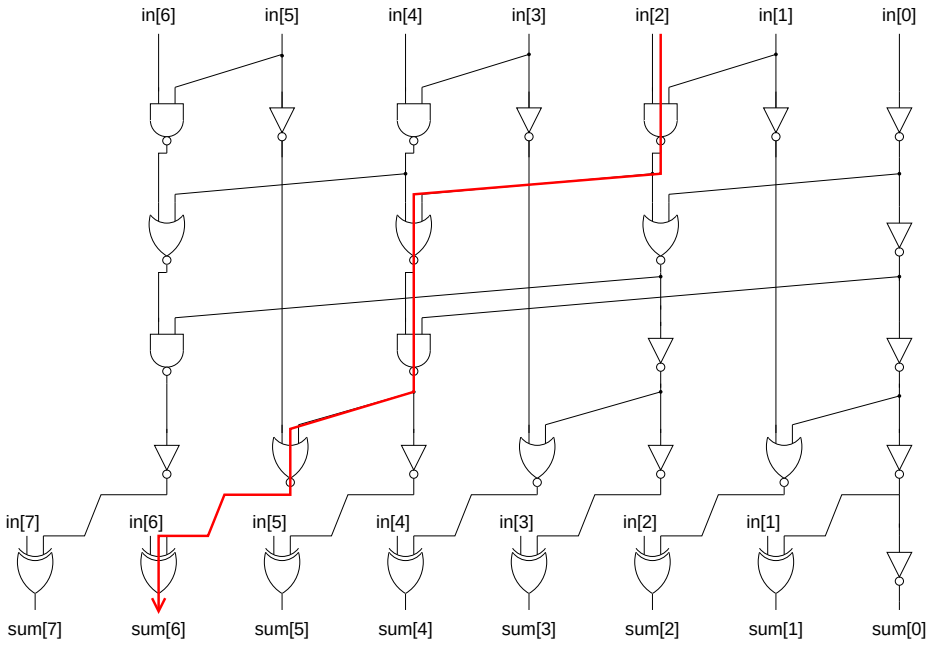
Problem 1 : Logical Effort for Incrementer Carry Chain (25 total points)

The following diagram illustrates two different incrementer architectures. For all parts of this question you should assume that the delay unit (τ) for this process is 10 ps and that the parasitic delay of a minimum-sized inverter (P_{inv}) is 1.



Ripple-Carry Architecture

There was an error in the ripple-carry gate topology given in the quiz. The last two NOR/NAND gates were accidently swaped. The correct topology is shown above. The error would not have affected your answers to Part 1.A or Part 1.B. It would have changed your answer to Part 1.C but full credit was awarded if you calculated the input caps using the incorrect gate topology.



Parallel-Prefix Architecture

Part 1.A : Critical paths for the adder architectures (5 points)

Draw a line through the critical path for both the ripple-carry and the parallel-prefix architectures. When determining the critical path you can assume that XOR gates are slower than NAND/NOR gates which are slower than inverters.

Part 1.B : Optimal delay of the adder architectures (10 points)

Use logical effort to calculate the optimal delay of the critical path for both architectures in picoseconds. *You should ignore all gates which are not on the critical path!* Do not using branching effort. Ignore the fact that some gates have a fanout greater than one. The desired input capacitance of the isolated carry chain is 6 fF (since we are ignoring gates which are not on the critical path this is the input capacitance for a single gate). The load capacitance of every sum output is 60 fF. Show all your work.

Ripple-Carry Architecture

$$\begin{aligned} F &= GH = (4/3)^3(5/3)^3(4) \times (60/6) = 43.89 \times 10 = 438.9 \\ P &= (3 \times 2p_{inv}) + (3 \times 2p_{inv}) + 4p_{inv} = 16 \\ f_{opt} &= F^{1/N} = 438.9^{1/7} = 2.385 \\ D_{opt} &= Nf_{opt} + P = 7 \times 2.385 + 16 = 32.69 \\ D_{abs} &= D_{opt}\tau = 326.9\text{ps} \end{aligned}$$

Parallel-Prefix Architecture

$$\begin{aligned} F &= GH = (4/3)^2(5/3)^2(4) \times (60/6) = 19.75 \times 10 = 197.5 \\ P &= (2 \times 2p_{inv}) + (2 \times 2p_{inv}) + 4p_{inv} = 12 \\ f_{opt} &= F^{1/N} = 197.5^{1/5} = 2.878 \\ D_{opt} &= Nf_{opt} + P = 5 \times 2.878 + 12 = 26.39 \\ D_{abs} &= D_{opt}\tau = 263.9\text{ps} \end{aligned}$$

Part 1.C : Gate sizing for the adder architectures (10 points)

Identify the optimum gate sizes for each gate in the critical path for both architectures. The gate sizes should be in femtofarads of input capacitance.

Ripple-Carry (correct topology)

Gate	C_{in} ***(fF)***
NAND2	*6.0*
NOR2	*10.7*
NAND2	*15.4*
NOR2	*27.5*
NAND2	*39.3*
NOR2	*70.3*
XOR2	*100.6*
Cout	*60*

Ripple-Carry (incorrect topology)

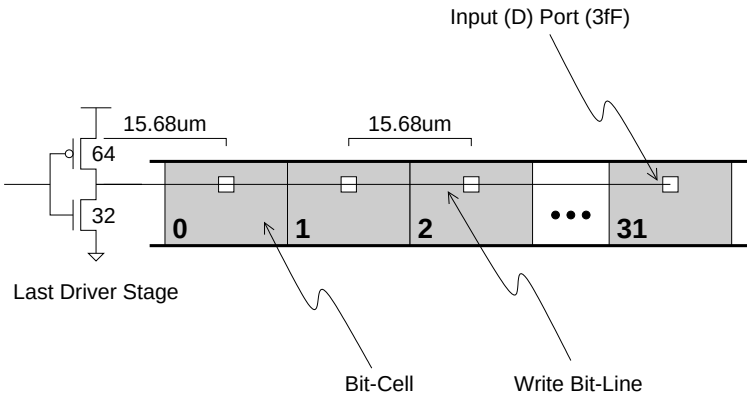
Gate	C_{in} ***(fF)***
NAND2	*6.0*
NOR2	*10.7*
NAND2	*15.4*
NOR2	*27.5*
NOR2	*39.3*
NAND2	*56.3*
XOR2	*100.6*
Cout	*60*

Parallel-Prefix

Gate	C_{in} ***(fF)***
NAND2	*6.0*
NOR2	*13.0*
NAND2	*22.4*
NOR2	*48.3*
XOR2	*83.4*
Cout	*60*

Problem 2 : RC Modeling of Register File Write Bitline (25 total points)

In this problem we will be revisiting the register file write bitline you analyzed in Lab 2. Remember that the write bitline must drive the D input port of 32 flip-flops. The combined gate capacitance of these flip-flops can be a significant load on the write bitline. The load on the write bitline is further increased by wire capacitance, since flip-flops are usually large and thus often spread apart. The following figure illustrates the write bitline including a reasonable final stage of the bitline driver. For this problem we will only consider this final stage even though the real driver might include many stages. As you determined in the lab assignment, each bitcell is 15.68 μm wide and the input capacitance of the bitcell's D port is 3 fF. The following figure illustrates the register file write bitline. The bitline is routed on Metal 2. You can ignore any via resistance or capacitance. Remember that the driver PMOS/NMOS sizes are in units of minimum NMOS transistor width (0.36 μm). For example, the NMOS for the last stage of the bitline driver is 0.36 μm $\times$ 32 = 11.52 μm.



The following table lists various parameters for a 0.18 μm technology which you may find useful when solving this problem. Remember that there is a list of equations at the end of this quiz.

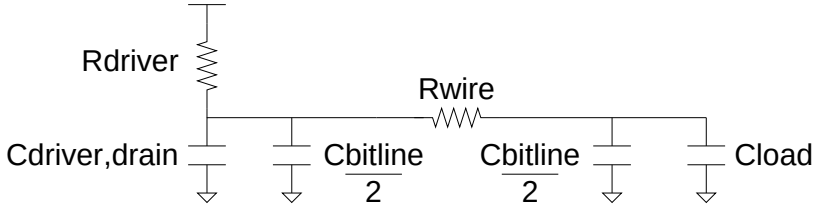
Transistor Process Parameters	**Value**
Desired ratio of PMOS/NMOS widths	2
PMOS gate capacitance per μm of transistor width	1.5 fF/μm
NMOS gate capacitance per μm of transistor width	1.5 fF/μm
PMOS drain capacitance per μm of transistor width	0.3 fF/μm
NMOS drain capacitance per μm of transistor width	0.3 fF/μm
PMOS effective on resistance	6.6 k$\Omega\mu$m
NMOS effective on resistance	3.3 k$\Omega\mu$m

Parameters for Metal 2 Wire	**Value**
Wire resistance per unit length	0.4 Ω/μm
Wire capacitance per unit length	0.2 fF/μm

Part 2.A : Delay calculation with end-of-line driver (10 points)

Draw a simple RC model for the register file write bitline. Only include the final stage of the driver. Use a lumped π wire model. Use the RC model to determine the delay of the write bitline. Express your answer in RC time constants. This part is very similar to the question asked in Lab 2.

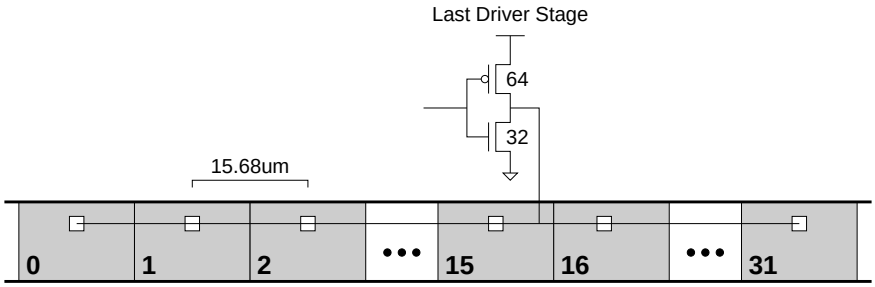
Students received 5 points for correctly drawing the RC model and 5 points for correctly performing the delay calculation. Students were not penalized if they reported their delay as a "unit-less" delay instead of reporting it in picoseconds.



$$
\begin{aligned}
R_{driver} &= 3.3\text{k}\Omega\mu\text{m}/11.52\mu\text{m} = 0.286\text{k}\Omega \\
C_{driver,drain} &= 96 \times 0.36\mu\text{m} \times 0.3\text{fF}/\mu\text{m} = 10.4\text{fF} \\
L_{wire} &= 32 \times 15.68\mu\text{m} = 501.75\mu\text{m} \\
R_{wire} &= 501.75\mu\text{m} \times 0.4\Omega/\mu\text{m} = 0.2\text{k}\Omega \\
C_{bitline,wire} &= 501.75\mu\text{m} \times 0.2\text{fF}/\mu\text{m} = 100.4\text{fF} \\
C_{bitline,gate} &= 31 \times 3\text{fF} = 93\text{fF} \\
C_{bitline} &= C_{bitline,wire} + C_{bitline,gate} = 193.4\text{fF} \\
C_{load} &= 3\text{fF} \\
\text{Time Constant} &= R_{driver} \times (C_{driver,drain} + C_{bitline}/2) \\
&+ (R_{driver} + R_{wire}) \times (C_{bitline}/2 + C_{load}) \\
&= 30.6\text{ps} + 48.6\text{ps} = 79\text{ps}
\end{aligned}
$$

Part 2.B : Delay calculation with middle-line driver (15 points)

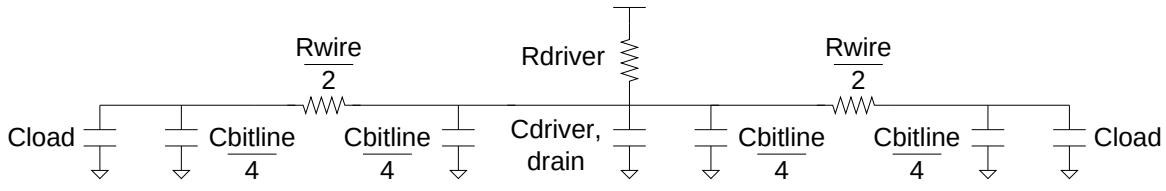
There is no reason we have to position the write bitline driver at one end of the bitline. In this part we will evaluate moving the driver to the middle of the bitline. The following figure illustrates the new design.



Draw a new RC model for the register file write bitline. Use the RC model to determine the delay of the write bitline. Express your answer in RC time constants. How does this new design compare to the baseline design evaluated in Part 2.A? Does this approach help mitigate wire resistance, wire capacitance, or both?

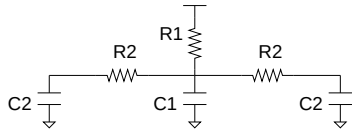
Students received 5 points for correctly drawing the RC model, 7 points for correctly performing the delay calculation, and 3 points for correctly answering whether this approach helps mitigate wire resistance, wire capacitance, or both. It was not terribly clear in the problem text, but the intent was for the students to calculate the time constant for driving the last bit of the bitline (as was the case in Lab 2 and in the previous part).

After studying this problem, the staff have determined that we provided the students with insufficient information to correctly solve this problem. We did not explain in lecture how to deal with "branching" when using the Penfield-Rubinstein approximation. All the students who attempted this question either significantly over-estimated or under-estimated the time constant. In this solution we first show how to correctly estimate the time constant, then we show the two (incorrect) approaches used by the students. Although incorrect, as long as the student attempted the delay calculation in a reasonable way we awarded the full 7 points.



This is a bit of an approximation since we are using the same $C_{bitline}$ as in the previous part, but it should still be a good approximation.

We can simplify this RC model as follows where $R_1 = R_{driver}$, $R_2 = R_{wire}/2$, $C_1 = C_{bitline}/2 + C_{driver,drain}$, *and* $C_2 = C_{bitline}/4 + C_{load}$.



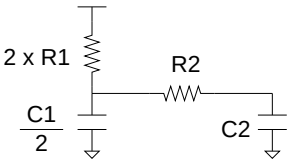
There are two approaches to estimate the time constant. The first uses the Penfield-Rubinstein approximation. In lecture, we only illustrated using the Penfield-Rubinstein approximation for RC chains, but in this example we must work with an RC tree. The classic paper by Rubinstein, Penfield, and Horowitz entitled "Signal Delay in RC Tree Networks" explains how to estimate the delay of these type of RC trees. This paper is available in the course locker (`/mit/6.375/doc/rubinstein-penfield.pdf`*). One of the primary results from the paper is that the time constant for a specific path from an input node to an output node i can be roughly approximated with the following equation.*

$$\text{Time Constant} = \sum_k R_{ki} C_k$$

where the summation is over all nodes in the tree and R_{ki} *is defined as the resistance of the portion of the path between the input and node i that is common with the path between the input and node k. The key points is that we* do *include capacitances which are off the path of interest but we* do not *include resistances which are off the path of interest. So using the Penfield-Rubinstein equation, the time constant for driving the final bitcell is approximately as follows.*

$$\begin{aligned} \text{Time Constant} &= R_1 C_1 + (R_1 + R_2) C_2 + R_1 C_2 \\ &= R_1 C_1 + (2R_1 + R_2) C_2 \end{aligned}$$

An alternative method exploits the fact that both branches of the RC tree are symmetric. Each branch gets exactly half of the drive current. Essentially it is as if we were driving each half of the bitline with a driver whose effective resistance is twice the original effective resistance. The following figure illustrates an equivalent RC model.



The approximated time constant for this equivalent circuit is the same as for the Penfield-Rubinstein approximation.

$$\begin{aligned} \text{Time Constant} &= 2R_1 C_1/2 + (2R_1 + R_2) C_2 \\ &= R_1 C_1 + (2R_1 + R_2) C_2 \end{aligned}$$

For both of these solution we can rewrite our result to look more like the result we found in Part 2.A.

$$\begin{aligned}\text{Time Constant} &= R_1C_1 + (2R_1 + R_2)C_2 \\ &\approx R_{driver} \times (C_{driver,drain} + C_{bitline}/2) \\ &+ (R_{driver} + R_{wire}/4) \times (C_{bitline}/2 + C_{load}) \\ &= 30.1\text{ps} + 33.5\text{ps} = 63.6\text{ps}\end{aligned}$$

Driving the bitline from the middle helps mitigate the impact of wire resistance. We can also determine this intuitively by considering two scenarios. In the first scenario, the wire capacitance is very large relative to the wire resistance. In this scenario you should be able to qualitatively see that driving the bitline from the middle will not really help the delay. Since the wire resistance is small, we can just lump all of the wire capacitance together - it does not matter where we drive the load from. In the second scenario, the wire resistance is very large relative to the wire capacitance. Now you should be able to see that driving the bitline from the middle will definitely help reduce the delay since it helps reduce the resistive path through which the driver charges up the load capacitance. The intuition then is that distributed drivers help mitigate wire resistance and are not really useful for mitigating wire capacitance.

Let us now consider the two solutions most students provided and explain why they are incorrect. The first solution includes the time constant for the left-hand branch in the approximation.

$$\begin{aligned}\text{Time Constant} &= R_1C_1 + (R_1 + R_2)C_2 + (R_1 + R_2)C_2 \\ &= R_1C_1 + 2(R_1 + R_2)C_2 \\ &\approx R_{driver} \times (C_{driver,drain} + C_{bitline}/2) \\ &+ (R_{driver} + R_{wire}/2) \times (C_{bitline}/2 + C_{load}) \\ &= 30.1\text{ps} + 38.5\text{ps} = 68.6\text{ps}\end{aligned}$$

This is an overly pessimistic approximation. Imagine if the branches were not symmetric and the resistance of the left-hand branch was very large. This would result in a large time constant for driving the right-hand branch since we are factoring in how long it takes to charge up the left-hand branch. The second solution provided by students completely ignored the left-hand branch.

$$\begin{aligned}\text{Time Constant} &= R_1C_1 + (R_1 + R_2)C_2 \\ &\approx R_{driver} \times (C_{driver,drain} + C_{bitline}/2) \\ &+ (R_{driver} + R_{wire}/2) \times (C_{bitline}/4 + C_{load}) \\ &= 30.1\text{ps} + 19.8\text{ps} = 49.9\text{ps}\end{aligned}$$

This is an overly optimistic approximation. This approach ignores the fact that some of the current to charge up the right-hand branch is being diverted to charge up the left-hand branch. The Penfield-Rubinstein approximation helps properly account for the left-hand branch without being overly pessimistic or optimistic.

Problem 3 : Bluespec Synthesis (25 total points)

Consider the algorithm for binary multiplication presented in Lecture 7 (Introduction to Bluespec):

```
   1001     // d = 4'd9
  x 0101    // r  = 4'd5
 ---------
   1001     // d << 0 (since r[0] == 1)
  0000      // 0 << 1 (since r[1] == 0)
 1001       // d << 2 (since r[2] == 1)
0000        // 0 << 3 (since r[3] == 0)
---------
0101101     // product (sum of above) = 45
```

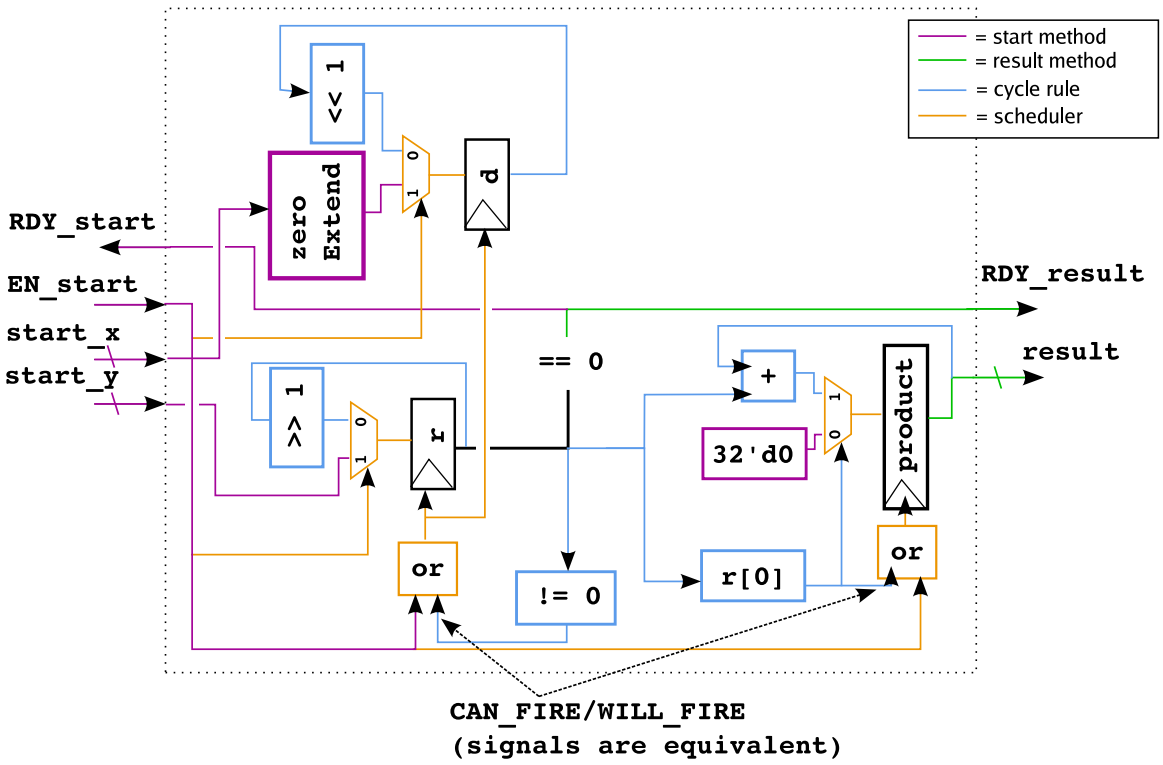
This algorithm is actually quite similar to the software multiplication algorithm you implemented for SMIPS in Lab 1. For this problem we will explore implementing this as a hardware module in Bluespec.

The following module implements this algorithm using two shifters to form an iterative multiplier:

```
interface I_mult;
  method Action start( Bit#(16) x,Bit#(16) y );
  method Bit#(32) result();
endinterface

module mkMult ( I_mult );
  Reg#(Bit#(32)) product <- mkReg(0);
  Reg#(Bit#(32)) d       <- mkReg(0);
  Reg#(Bit#(16)) r       <- mkReg(0);

  rule cycle ( r != 0 );
    if ( r[0] == 1 )
      product <= product + d;
    d <= d << 1;
    r <= r >> 1;
  endrule

  method Action start( Bit#(16) x, Bit#(16) y ) if ( r == 0 );
    d <= zeroExtend(x);
    r <= y;
    product <= 0;
  endmethod

  method Bit#(32) result() if ( r == 0 );
    return product;
  endmethod
endmodule
```

Diagram the hardware that the Bluespec compiler should produce for this module, including interface ports. Clearly circle and label which parts correspond to the rule, the scheduler, the `start` and `result` methods. Label which wire or wires correspond to `CAN_FIRE_cycle` and `WILL_FIRE_cycle`, as well as all ports corresponding to method ready and enable signals.

Some students included the CLK and RST as inputs. This was not required, but of course, not penalized. The CLK and RST are implicit in the diagram solution. Other students decided to represent register enables as an extra level of muxing. This is also acceptable, and in many cases easier to reason about. The compiler found a small optimization on the product_enable signal that no student found, but of course this is unnecessary.



Problem 4 : Rule Scheduling in Bluespec (25 total points)

In this problem we will explore the behavior of the pipeline used in Lab 3 and presented in class. The reference code has been included in a separate handout.

In order to gain fine-grained control over the scheduling, it is often desirable to split large rules with case statements into multiple rules. Consider the `execute` rule. It only interacts with the `dataReqQ` on a memory operation, so one natural partitioning is to create an `execMem` rule which handles Load and Store operations.

Similarly the `execute` rule only interacts with the `pc` when the current instruction is a branch. Therefore one design choice might be to separate the handling of branch instructions into a separate rule. However this choice is actually too restrictive. In point of fact, the execute stage only sets `pc` on a *taken* branch. Consider the design where execute is split into four rules, `execALU`, `execMem`, `execBr_NotTaken`, and `execBr_Taken`.

For reference, here is the code for the `execBr_NotTaken` and the `execBr_Taken` function.

```
function Bool isBranch( Instr i );
  // Returns True if i is a Branch
endfunction

function Bool branchTaken( Instr i );
  // If given a branch instruction, returns True if the branch is taken,
  // otherwise returns False.
  // Note that in some cases this involves reading the RegFile.
endfunction

rule execBr_NotTaken ( instRespQ.first() matches tagged LoadResp .ld
                       &&& ld.tag == epoch
                       &&& unpack(ld.data) matches .inst
                       &&& !stallfunc(inst)
                       &&& isBranch(inst)
                       &&& !branch_taken(inst) );

  pcQ.deq();
  instRespQ.deq();

endrule
```

```
rule execBr_Taken ( instRespQ.first() matches tagged LoadResp .ld
                    &&& ld.tag == epoch
                    &&& unpack(ld.data) matches .inst
                    &&& !stallfunc(inst)
                    &&& isBranch(inst)
                    &&& branch_taken(inst) );

  Addr next_pc;

  case (inst) matches
    tagged J     .it :
      next_pc = { pcQ.first()[31:28], it.target, 2'b0 };
    tagged JR    .it :
      next_pc = rf.rd1(it.rsrc);
    tagged JAL   .it :
     begin
      wbQ.enq( WB_ALU {dest: 31, data: pcQ.first()} );
      next_pc = { pcQ.first()[31:28], it.target, 2'b0 };
     end
    tagged JALR  .it :
     begin
      wbQ.enq( WB_ALU {dest: it.rdst, data: pcQ.first()} );
      next_pc = rf.rd1(it.rsrc);
     end
    //BLEZ, BGTZ, BTZ, BGEZ, BEQ, BNE
    default:
      next_pc = pcQ.first() + (sext(it.offset) << 2);
  endcase

  pc <= next_pc;
  epoch <= epoch + 1;
  pcQ.deq();
  instRespQ.deq();

endrule
```

After splitting this system the rules have the following resource usage. (Note that the FIFO clear methods are unused.)

pcGen	**discard**	**execALU**	**execMem**
pc.read	epoch.read	epoch.read	epoch.read
epoch.read	pcQ.deq	instRespQ.first	instRespQ.first
pc.write	instRespQ.deq	instRespQ.deq	instRespQ.deq
pcQ.enq		pcQ.deq	pcQ.first
instReqQ.enq		wbQ.enq	pcQ.deq
		wbQ.find1,2	wbQ.enq
		rf.rd1,2	wbQ.find1,2
			rf.rd1,2
			dataReqQ.enq

execBr_Taken	**execBr_NotTaken**	**writeback**	
epoch.read	epoch.read	wbQ.first	
instRespQ.first	instRespQ.first	wbQ.deq	
instRespQ.deq	instRespQ.deq	dataRespQ.first	
pcQ.first	pcQ.deq	dataRespq.deq	
pcQ.deq	wbQ.find1,2	rf.wr	
wbQ.enq			
wbQ.find1,2			
rf.rd1,2			
dataReqQ.enq			
epoch.write			
pc.write			

Part 4.A : Method scheduling 1 (6 points)

Suppose you want your system to have the following scheduling behavior when multiple rules execute in the same clock cycle:

```
pcGen < execBr_Taken < writeback
```

These rules interact through various modules such as the `pc` and `pcQ`. For each of these modules, give the method relationship necessary to meet the above scheduling behavior. For modules where the order is irrelevent or determined by factors outside of the processor write N/A. We've done `pc`, you do the rest.

```
pc: read < write          epoch:  read < write    rf: rd{0,1} < wr

pcQ: enq < first, deq

instReqQ: N/A                     instRespQ: N/A

wbQ: find{1,2}, enq < first, deq

dataReqQ: N/A                     dataRespQ: N/A
```

Part 4.B : Method scheduling 2 (7 points)

Perform the same reasoning, but for the following scheduling property:

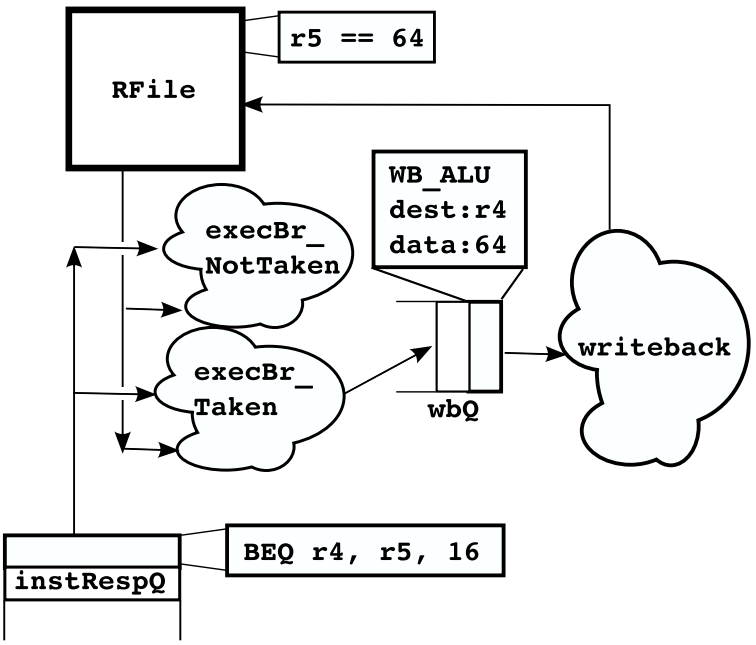
```
writeback < execBr_Taken < pcGen

pc: write < read        epoch: write < read     rf: wr < rd{1,2}

pcQ: first, deq < enq

instReqQ: N/A                     instRespQ: N/A

wbQ: first, deq < find{1,2}, enq

dataReqQ: N/A                     dataRespQ: N/A
```

Part 4.C : Dynamic Behavior (12 points)

Consider the following three variants of a processor:

- Behaves as if: `pcGen < execBr_NotTaken, execBr_Taken < writeback`
- Behaves as if: `writeback < execBr_NotTaken, execBr_Taken < pcGen`
- Behaves as if: `writeback < execBr_NotTaken < pcGen < execBr_Taken`

While running a program these processors reach the following state:



For each variant, answer the following. A) What rules (of those shown) will the scheduler choose to fire and why. B) What is the longest combinational path in the system (including the parts not shown)?

`pcGen < execBr_NotTaken, execBr_Taken < writeback`

A) The execBr rules will not be able to fire because of the stall signal, since wbQ.find < wbQ.deq. The writeback rule will fire, so the wbQ will be empty and the execBr_Taken rule will fire on the next cycle.

B) If you work with non-combinational memories, as we did in the lab, then the paths are as follows. pc->pcGen->instReqQ, instRespQ/RF->execs->dataReqQ, and dataRespQ->rf. If, on the other hand you assume the memories are combinational, then this will act like an unpipelined machine, with the longest path going through the whole machine, including the memories.

`writeback < execBr_NotTaken, execBr_Taken < pcGen`

A) The writeback rule will fire first, and dequeue the wbQ. Therefore there is no stall for the execBr rules, and the execBr_Taken rule will fire, since it will read r4 == r5 == 64.

B) The longest path in this system flows from the writeback rule, through the register file and through the execute rules. Then, since the execute rule updates the PC, the path continues through the pcGen rule and to the instReqQ and pcQ.

`writeback < execBr_NotTaken < pcGen < execBr_Taken`

A) This situation is the same as the previous, since we are not considering the pcGen rule (it is not in the diagram).

B) The longest path goes from the writeback rule, through the register file and through the execute rules. However, since the new PC is no longer bypassed to the pcGen rule, that part of the path is no longer present. In fact in this system it would not be surprising if the ALU were the longest path.

Equation Sheet

Equation or Symbol	Description
g	Gate logical effort
$h = C_{out}/C_{in}$	Gate electrical effort
$f = gh$	Gate effort
p	Gate parasitic delay
p_{inv}	Parasitic delay of minimum-sized inverter
τ	Delay unit
$d = f + p$	Delay in units of τ
$d_{abs} = d\tau$	Absolute delay in seconds
$G = \prod g_i$	Path logical effort
$H = C_{out}/C_{in}$	Path electrical effort
$F = GH$	Path effort
$D = \sum d_i = \sum g_i h_i + \sum p_i$	Path delay
$f_{opt} = F^{1/N}$	Optimal stage effort
$D_{opt} = N f_{opt} + P$	Optimal path delay
$C_{in,opt,i} = C_{out,i} \times g_i / f_{opt}$	Optimal input capacitance for stage i
Delay $= \sum_{i=0}^{n} \left(\sum_{j=0}^{j=i} R_j \right) C_i$	Penfield-Rubenstein wire-delay model
R_d	Effective driver resistance
R_w	Total wire resistance
C_w	Total wire capacitance
Delay $\propto R_d \times C_w/2 + (R_d + R_w) \times (C_w/2 + C_{load})$	Simple lumped π model

	Number of inputs					
Gate Type	**1**	**2**	**3**	**4**	**5**	**n**
Inverter Logical Effort	1					
NAND Logical Effort		4/3	5/3	6/3	7/3	$(n+2)/3$
NOR Logical Effort		5/3	7/3	9/3	11/3	$(2n+1)/3$
XOR/XNOR Logical Effort		4	12	32		
Inverter Parasitic Delay	p_{inv}					
NAND Parasitic Delay		$2p_{inv}$	$3p_{inv}$	$4p_{inv}$	$5p_{inv}$	np_{inv}
NOR Parasitic Delay		$2p_{inv}$	$3p_{inv}$	$4p_{inv}$	$5p_{inv}$	np_{inv}
XOR/XNOR Parasitic Delay		$4p_{inv}$	$4p_{inv}$	$4p_{inv}$	$4p_{inv}$	$4p_{inv}$