ASYNCHRONOUS RESEARCH CENTER

Portland State University

Subject: **Seventh Class Handout - Data-Controlled Branch & Demand Merge**
Date: **November 10, 2010**
From: **Ivan Sutherland**
ARC#: **2010-is54**

References:

ARC# 2010-is43: Class 1 – Ring Oscillators, Ivan Sutherland, 26 September 2010
ARC# 2010-is44: Class 2 – GasP Rings, Ivan Sutherland, 1 October 2010
ARC# 2010-is45: Class 3 – Linear GasP, Ivan Sutherland, 8 October 2010
ARC# 2010-is49: Class 4 – Proper Stopper, Ivan Sutherland, 15 October 2010
ARC# 2010-is52: Class 5 – Broad Branch and Broad Merge, Ivan Sutherland, 22 October 2010
ARC# 2010-is53: Class 6 – Round Robin FIFO, Ivan Sutherland, 25 October 2010
C.H.vanBerkel and C.E.Molnar, Beware the Three-Way Arbiter, IEEE Journal Of Solid-State Circuits, Vol. 34, No. 6, June 1999

ABSTRACT

A data-controlled branch module can steer data to one output or another according to the data content. A tree network of data-controlled branch modules can steer data from a common source to many destinations. To combine data elements from several sources requires a demand merge module with two inputs and one output. Because data may arrive at any time, the demand merge module must contain a mutual exclusion element. Together data-controlled branch and demand merge modules make possible networks of a wide variety of topologies.

BACKGROUND

The last two lessons have involved branching pipelines. The broad branch and merge modules can send the same data into two or more parallel paths and accept data from the separate branches. The alternate branch and merge, or their generalization to more than two outputs, can send successive data elements down separate paths and gather up the data in their original sequence. The data-controlled branch and demand merge modules complete our family of branch and merge modules.

DATA CONTROLLED BRANCH

A data-controlled branch module chooses which successor or successors should get each incoming data element. It not only captures the incoming data, but also uses one or more bits of the incoming data to select which outgoing channels it will inform about arrival of the data. It can signal data presence to all, to some, or to none of its successors. In fact, the actual data wires carry the same data value to all potential recipients, but the branch module fills the state wires of only the chosen recipients. An exploration of the data-controlled branch must include at least the data wires used to control the branch. This simulation example therefore includes some data wires and their latches as well as the GasP control modules involved.

Recall the protocol of the state wires between GasP modules. When a module fires it does three things: it copies the input data, it drains its predecessor state wire, and it fills its successor state wire. A data-controlled branch module has several successor state wires from which to choose; it fills none, one, or more of them according to data bits contained in the incoming data value. This is the first module we have seen for which a data value matters. The other modules we have considered are completely agnostic to the data values they control.

How do we know that the data bits for controlling the branch arrive in time to select the appropriate successor? Do we need extra data latches in the branch module to capture separately data destined for different successors? Do we need multiple internal `fire` signals like those in the alternating branch?

The data-controlled branch module is just like the broad branch module except that each successor driver must be conditional. The library called `gaspL` offers two variations of the conditional successor driver called `sucConDri10` and `sucConDri20`. As their names suggest, they differ in the strength of their P-type successor drive transistor.

Recall that the regular successor driver, `sucDri10`, has a small inverter between the `fire` signal and the P-type successor drive transistor. Instead, the conditional successor driver, `sucConDri10`, has a NAND gate between the `fire` signal and the P-type successor drive transistor. The conditional successor driver fills its successor state wire only if the second NAND input is HI during the `fire` pulse. In `sucConDri10` this second NAND input is called `cond`. Obviously the data-controlled branch connects the `cond` terminal to a data bit.

Which set of latches holds the data bit to use as the `cond` signal? A common mistake is to think that the latches driven by the data-controlled branch module hold the routing information. The conditioning signal must be valid during the `fire` pulse. Thus we must use the *incoming* data to choose a destination. The incoming data must be valid during the `fire` pulse in order for the latches of the branch stage to copy their values. The outgoing data bits captured in the branch stage's own latches are

unsuitable because the latch outputs change during the `fire` pulse. The outgoing data bits hold valid data only *after* the stage's `fire` pulse renders their latches momentarily transparent. Thus the outgoing data bits become valid too late to control a successor state wire. *Please avoid thinking that we can capture data and **then** use it to steer an output.* Instead, we must use the *incoming* data to steer the output.

DATA CONTROLLED BRANCH MODULE

I thought that the library called `gaspL` contained a data-controlled branch module. Indeed it should. However, the version of 20 September 2010, designated 20sep10, is deficient in this regard. I have therefore included a data-controlled branch called `gaspConBranch10` in the homework library for this week. It will appear in later versions of the standard GasP library.

Examination of `gaspConBranch10` reveals that it is a two-way branch. Two conditional successor drivers, `sucConDri10`, each drive a separate successor wire. Moreover, It uses two bits of input data to control whether or not to fill each of its two successor state wires. The four combinations of the two control bits allow for alerting neither, either, or both successors to the presence of data. Notice that this form generalizes to any number of successor wires.

Suppose, instead, we can afford only one control bit in our data. If that bit is one, fill successor state wire A; if it is zero, fill successor state wire B. Clearly an inverter between the two `cond` inputs to the conditional state wire drivers would make sure only one of them acts when the `fire` pulse happens. This form is common and will work if the data arrive soon enough.

If unsure about how soon incoming data will arrive, use the alternative circuit called `sucConDri20inv`. This circuit uses the conditional data to control a P-type transistor in series with the one that fills the successor state wire. Thus only a LO input will drive the successor. Using one `sucConDri10` and one `sucConDri20inv` together permits proper operation in spite of late-arriving input data.

All of the branch modules we've considered can act only when all their successor state wires are LO meaning empty. Thus they are all subject to delay on both outputs if either output is unable to accept data promptly. A module that can deliver data to one output even if the other is blocked requires another stage of storage. It must be able to store the data for the blocked channel while delivering different data to the output that's not blocked. That's entirely equivalent to following the branch described here with a pair of plain GasP modules, one for each output.

DEMAND MERGE

Like other merge modules, the demand merge has two predecessor state wires and a single successor state wire. It drives a latch-with-multiplexed-input that can accept data from either source. The demand merge decides which input to accept and

renders that part of its two-input latches transparent. It must accept inputs from either predecessor, but how can it choose which to serve next?

Input signals may arrive at any time; indeed two inputs may arrive arbitrarily close together in time. A demand merge must accept inputs on a first-come-first-served basis. It must be able to deal with inputs arbitrarily close together in time. Of course, that requires a mutual exclusion element to decide cleanly who came first. As we've seen before, a mutual exclusion element may take extra time to decide which to serve next if two inputs arrive at nearly the same time.

The library called `gaspL` contains a demand merge module that is called `gaspDemandMerge`. To decide which input to serve next, the demand merge module uses the same mutual exclusion element we saw in the proper stopper. It uses two separate internal GasP modules to accept data from its two inputs. Ordinarily the mututal exclusion element causes almost no extra delay. Occasionally, however, difficult decisions by the mututal exclusion element may causes extra delay.

It would be nice to generalize the demand merge to more than two inputs. However, I don't know how to make a mutual exclusion element with more than two inputs. One might think of using a tri-flop, a device with three rather than two stable states, as the basis for three-way mutual exclusion. However, there's controversy about this topic in the literature as suggested by the paper cited above by vanBerkel and C.E.Molnar. I lack confidence in the meta-stability properties of the tri-flop. For use as a mutual exclusion element, such a circuit must have at least three different meta-stable ridge lines, each between two of its stable states.

Moreover, two-way demand merge elements suffice. If one must merge three streams of data, one can merge two first and then merge in the third. Freeway systems do quite nicely with two-way traffic merges; so can data networks. Trees of two-way demand merge can accommodate any number of inputs.

Examination of `gaspDemandMerge` reveals that each side waits for three things. Obviously each waits for the mutual exclusion element to decide in its favor. Each side also waits for their common successor to be empty. Because each side fills their common successor state wire, that might be enough to prevent further action until the successor stage accepts the data. However, we have found it advisable also to wait for the third condition, namely that the other side has finished firing.

Waiting for the end of the other `fire` pulse strengthens the circuit for a reason that may not be obvious. Suppose that the successor state wire is heavily loaded and therefore slow to respond. Suppose that the predecessor state wires are lightly loaded and thus fast to respond. As soon as one side's `fire` pulse starts, it drains its predecessor state wire. The drained predecessor state wire clears the mutual exclusion element.

That whole process can happen in as little as two gate delays. It only takes the same two gate delays for the `fire` pulse to fill the successor state wire. A slow successor and fast predecessor might cause an improper fire pulse because of inadequate timing margin. For safety we must be sure that the loosing side is inhibited throughout this brief period. Using the winning `fire` signal to inhibit the loosing side provides extra margin. That's the third input to the AND function.

SIMULATION ASSIGNMENT

This week's assignment involves a data-controlled branch, stage 3, followed by a demand merge. The branch sends data either to the upper pipeline or to the lower pipeline or to both or to neither. The two central pipelines differ in length: the lower one has six stages (6:11) and the upper one has two stages (4:5). The demand merge module, stage 12, combines the two pipeline outputs into a single output stream.

Each GasP stage drives three data latches. These three latches appear as a single symbol in the schematic. The indexed name above the latch symbol causes Electric to duplicate the latch. For example, `lat[1:3]` and `lat[22:24]` are two different groups of three latches each. Notice that the demand merge stage produces two `fire` signals, `fire[12A]` and `fire[12B]`, and the three latches, `lat[16:18]`, use `fire[12A]` and `fire[12B]` to control their two inputs.

The green cables between latches each carry three data wires. I have used double indexing to name the data wires so that the name "`data`" applies to all of them. The cable called `data[1:3][1]` is the first three-wire cable; the cable called `data[1:3][2]` is the second three-wire cable, and so on. I have omitted names from some of the less important cables.

Notice that `data[1:3][3]`, are the three data output wires from stage 3, the data-controlled branch. These three wires connect to the latch inputs in both stage 4 and stage 6. Their state wires, sw[3A] and sw[3B] are distinct. Thus the data controlled merge actually sends data to the inputs of the latches of both its successors. However, it may or not fill the two separate state wires.

Bits 1 and 2 of the three-bit data word represent address bits that control data flow in this simple network. Bit 1 of the three-bit data word controls whether or not data should enter the short upper pipeline. Bit 2 of the three-bit data word controls whether or not data should enter the long lower pipeline. If both bits are TRUE, data enter both pipelines. If neither bit is TRUE, no data pass beyond stage 3. Bit 3 of the three-bit word bit represents the "payload" bits carried through this network. Bit 3 is TRUE for some data elements and FALSE for others.

I wish there was a simple source of data synchronized with `fire[1]`, the source module's `fire` pulse, but there isn't. Instead, three timers at the left of the schematic change data inputs called `in[1:3]` from time to time to provide interesting

patterns of activity. It may happen that such timed inputs would drive some data latches into meta-stability, but that is unlikely. Moreover, data outputs from stage 1 are timed by the `fire[1]` pulse, and so stage 2 receives sensible data inputs.

My simulation repeats its pattern after 25 nsec, so you should simulate at least that much time. I have set the parameters of the pulse generators for `in[1:2]` to make an interesting pattern of activity. After sending nothing out, data go first to the longer pipeline, then to both, and finally to only the upper pipeline. Data collide at the demand merge, causing a backup in the pipelines. You can see all that in your simulation, and there are some questions for you to answer about it.

I've included a picture of my output so you can see the kind of pattern to expect. My output will differ in detail from yours, because I changed the timers after making my output picture. Answer the questions from your own simulation output.

ANSWER SHEET – due by beginning of class on **16 November 2010** **7**

Name _____

Turned in on Date _____

Answer from simulation:

My simulation uses _____ technology – e.g. 180 MOSIS

AT THE INPUT – stage 2 or 3

Immediately after master clear, `data[1:3][2]` are all <HI> <LO> . Why?
Hint: look inside the latches.

100 psec before the first pulse on `fire[3]`, `data[1:3][2]` are <HI> <LO> . Why?

The flow rate at `fire[2]` is _____ GDI/s (maximum) and _____ GDI/s (average).

WHERE DO DATA VALUES GO? - stage 3

The first _____ data values to reach stage 3 go nowhere.

The next _____ data values to reach stage 3 go only to stage 6.

The next _____ data values to reach stage 3 go to both stages 4 and 6.

The next _____ data values to reach stage 3 go only to stage 4.

AT THE DEMAND MERGE – stage 12

The first data to reach the demand merge come from <stage 5> <stage 11>

The first _____ data elements out of stage 12 came from the <upper> <lower> pipeline.

Pulses on `fire[12A,12B]` alternate when _____

At 12, the last _____ data elements in a group came from the <upper> <lower> pipe.

OUTPUT VS INPUT – stage 2 and 14

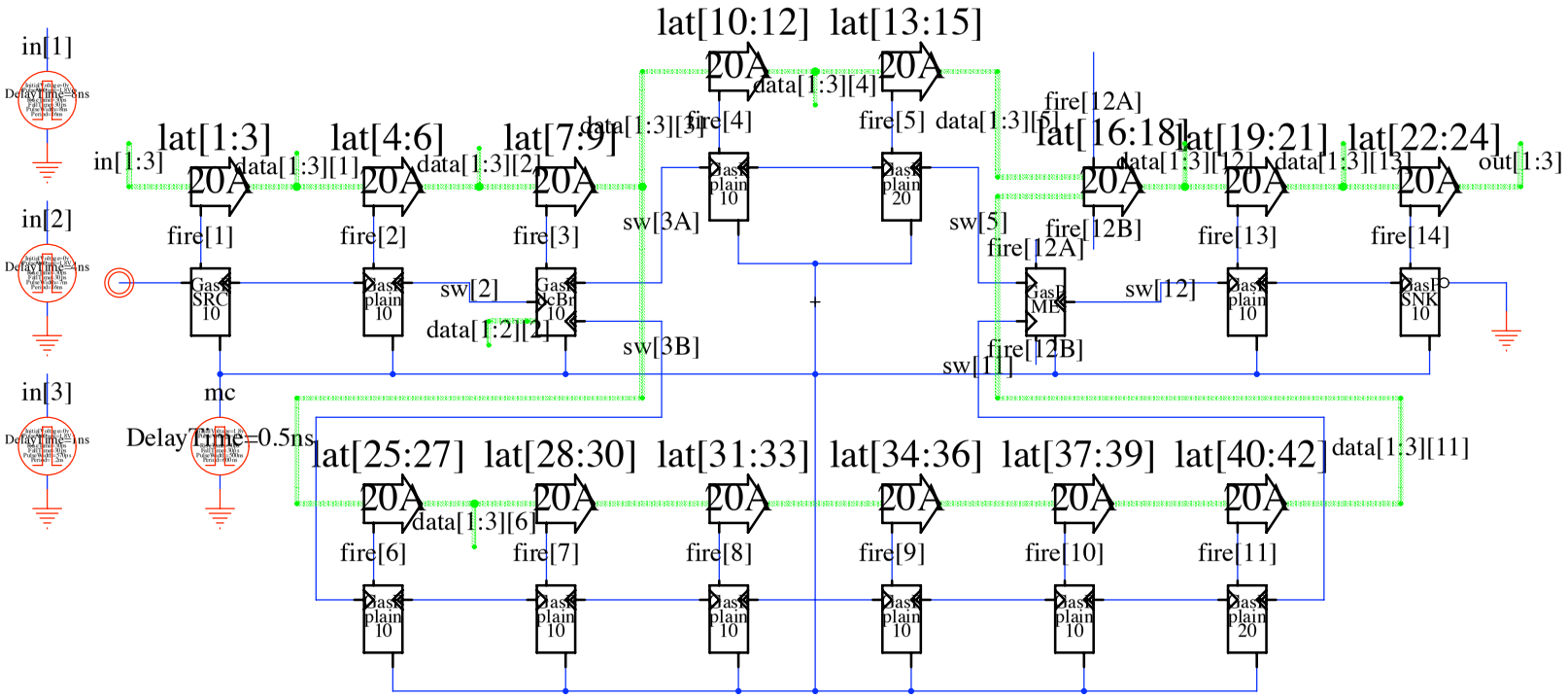
The flow rate at `fire[14]` is _____ GDI/s (maximum) and _____ GDI/s (average).

Why do these values differ from the values at stage 2?

What causes the irregular pattern of pulses at stage 14? At at stage 2?

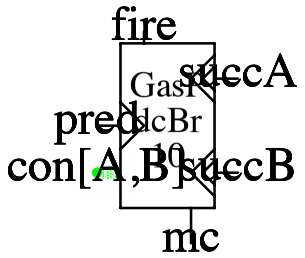
splitPath

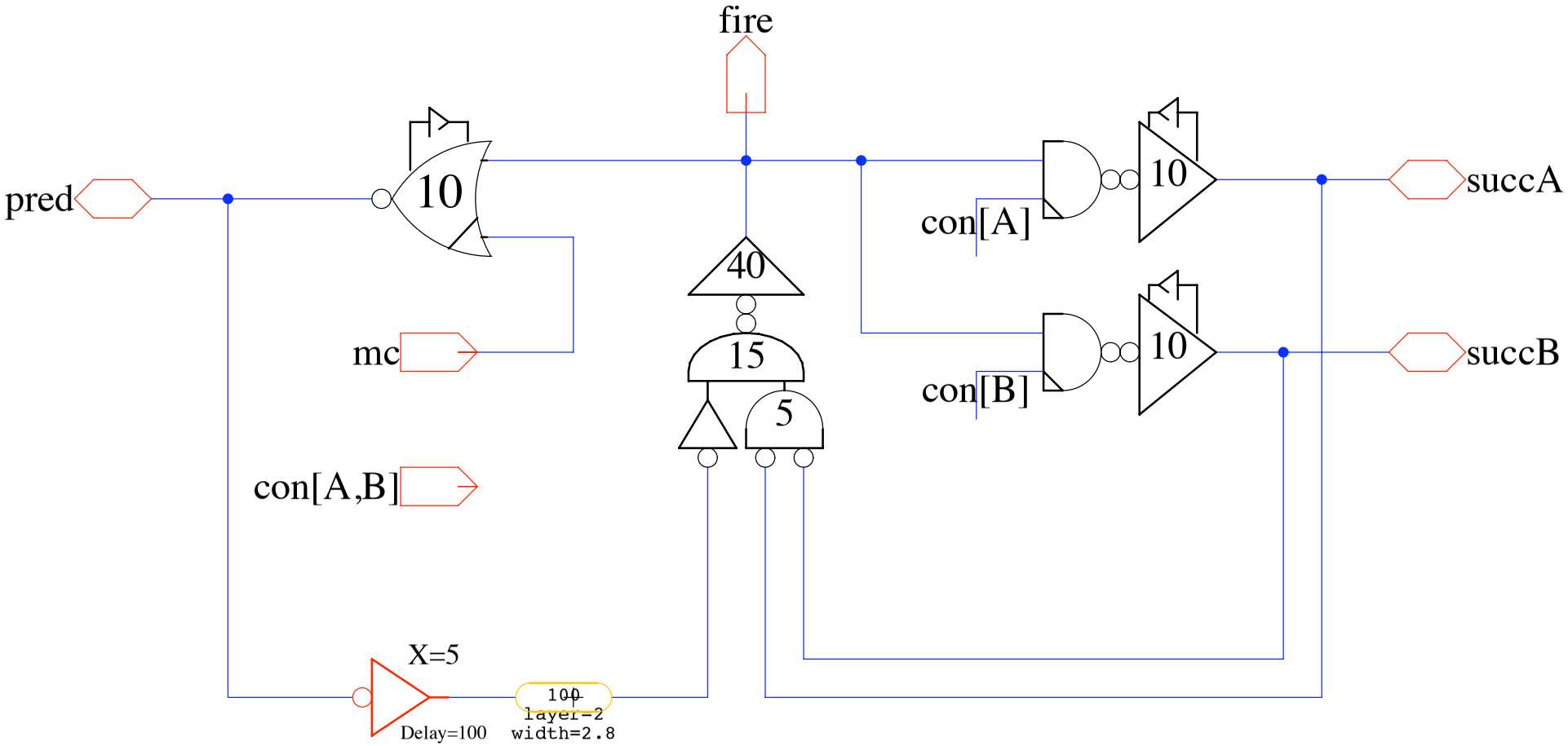
ies 9 November 2010



gaspDataBranch10

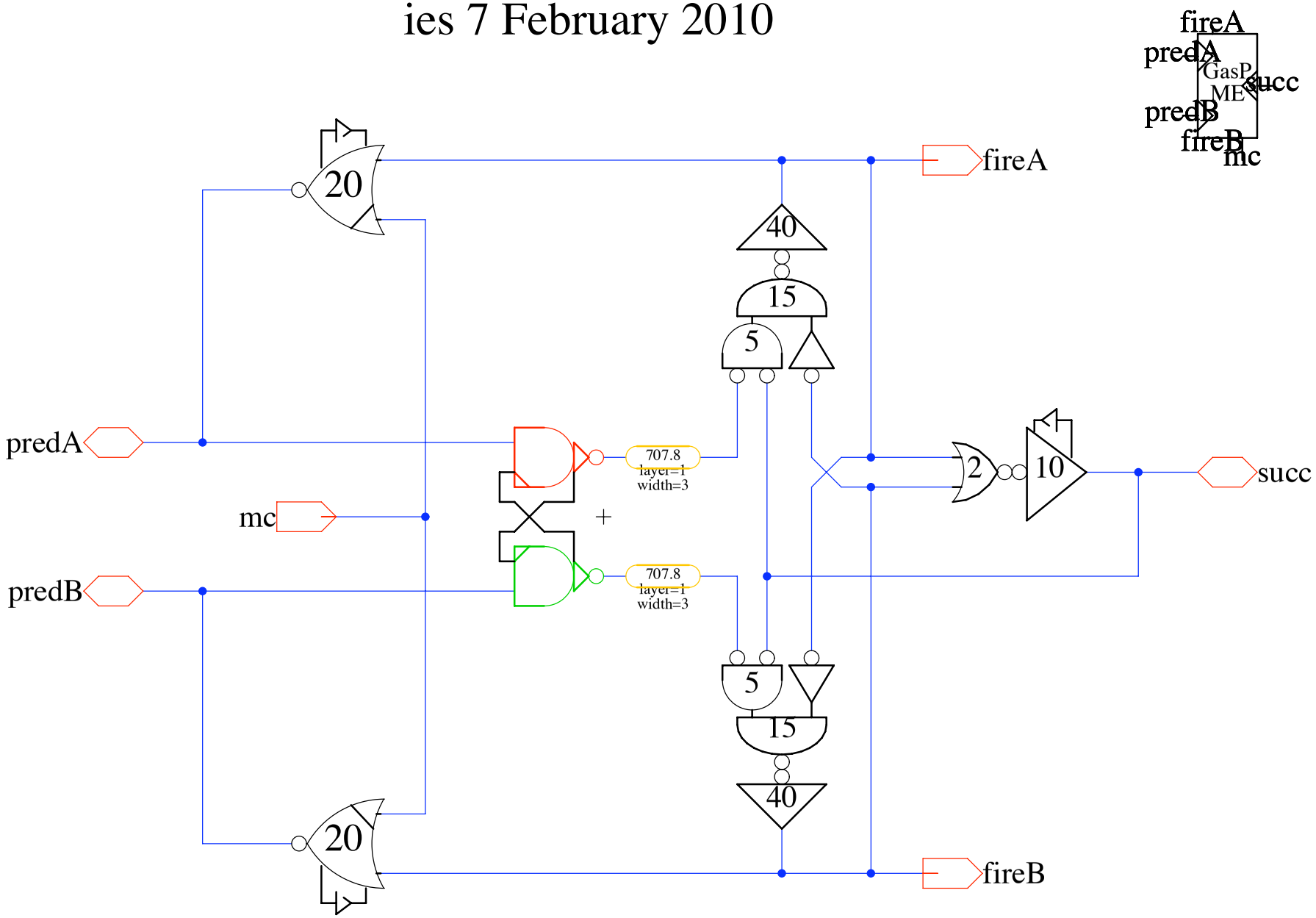
ies 1 November 2010





gaspDemandMerge10

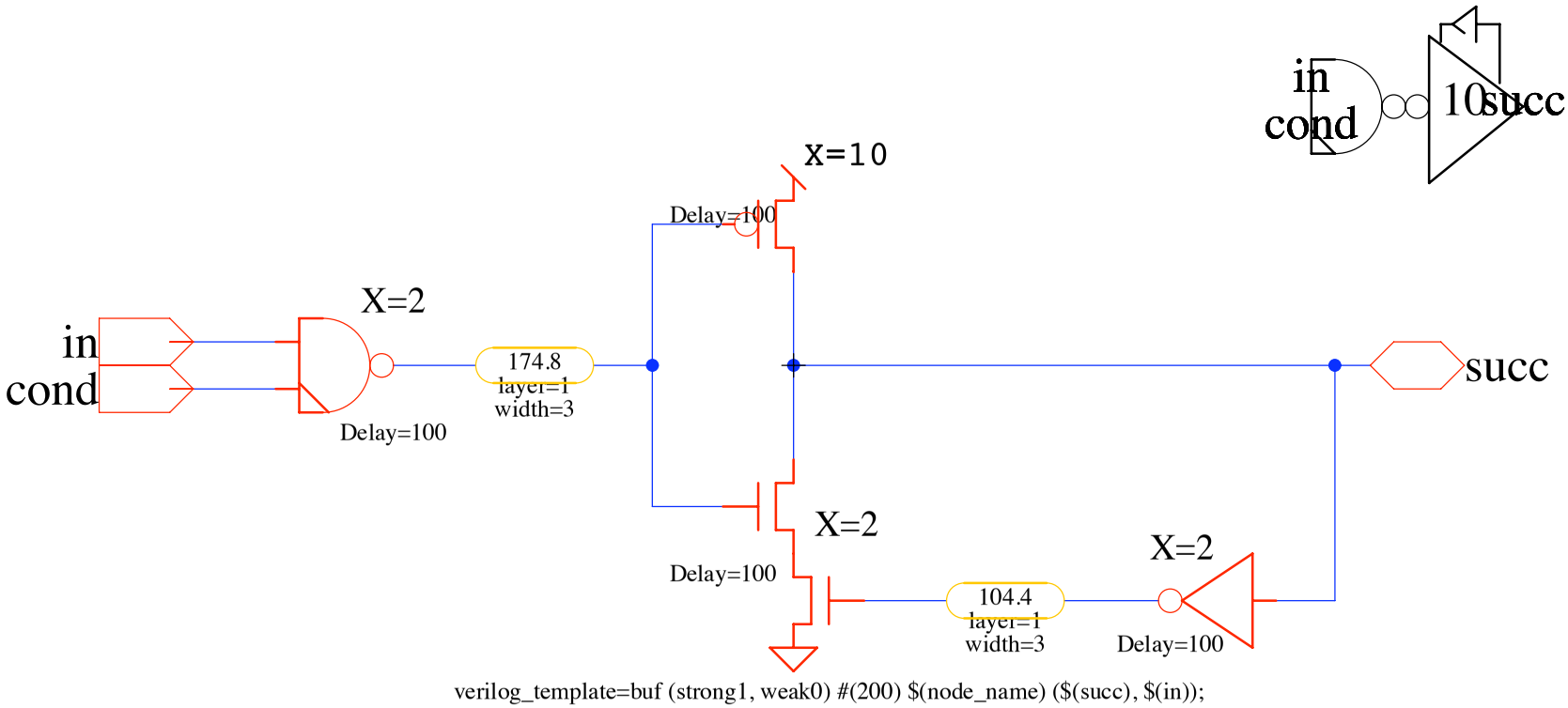
ies 7 February 2010



sucConDri10

HI is full conditional successor driver

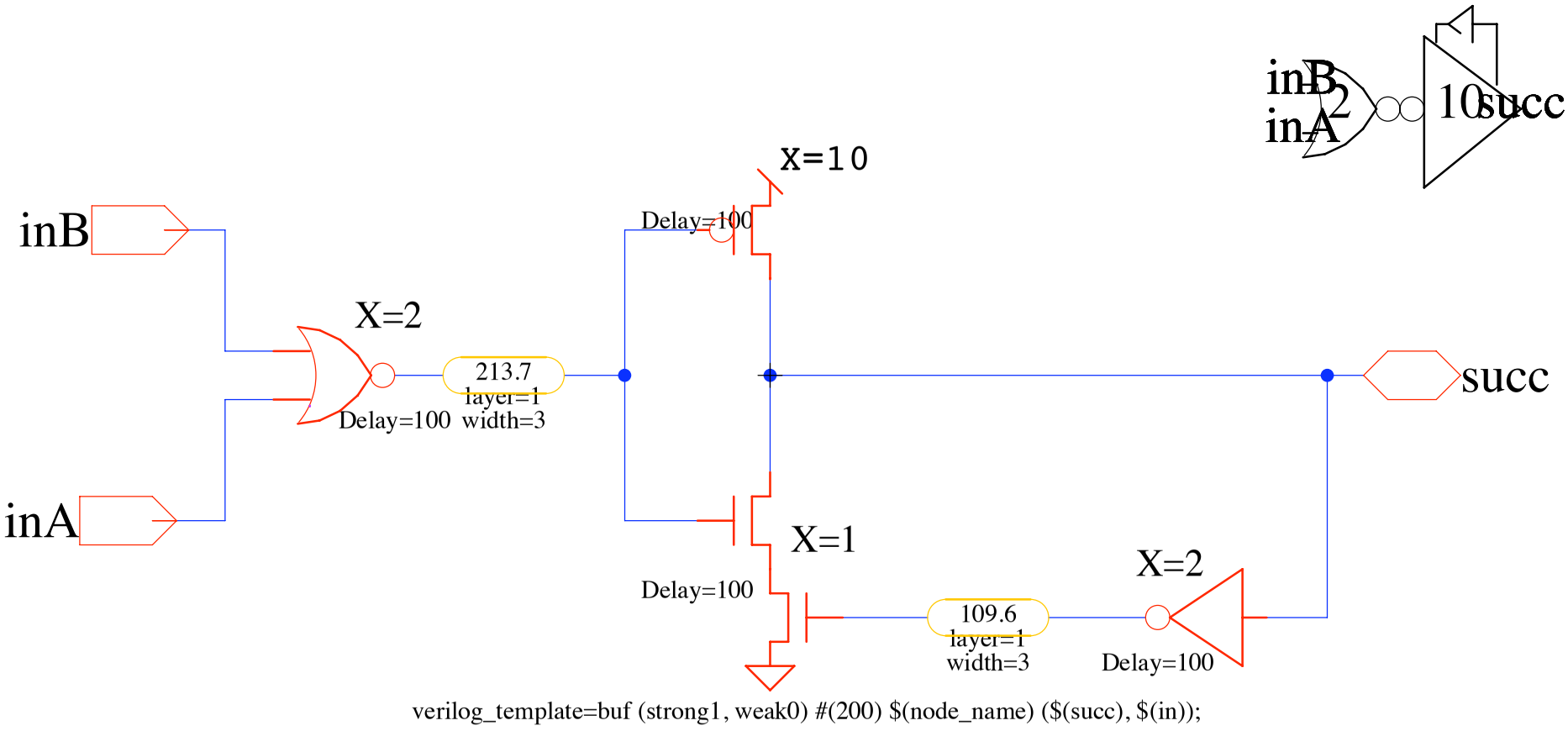
ies 10 August 2010

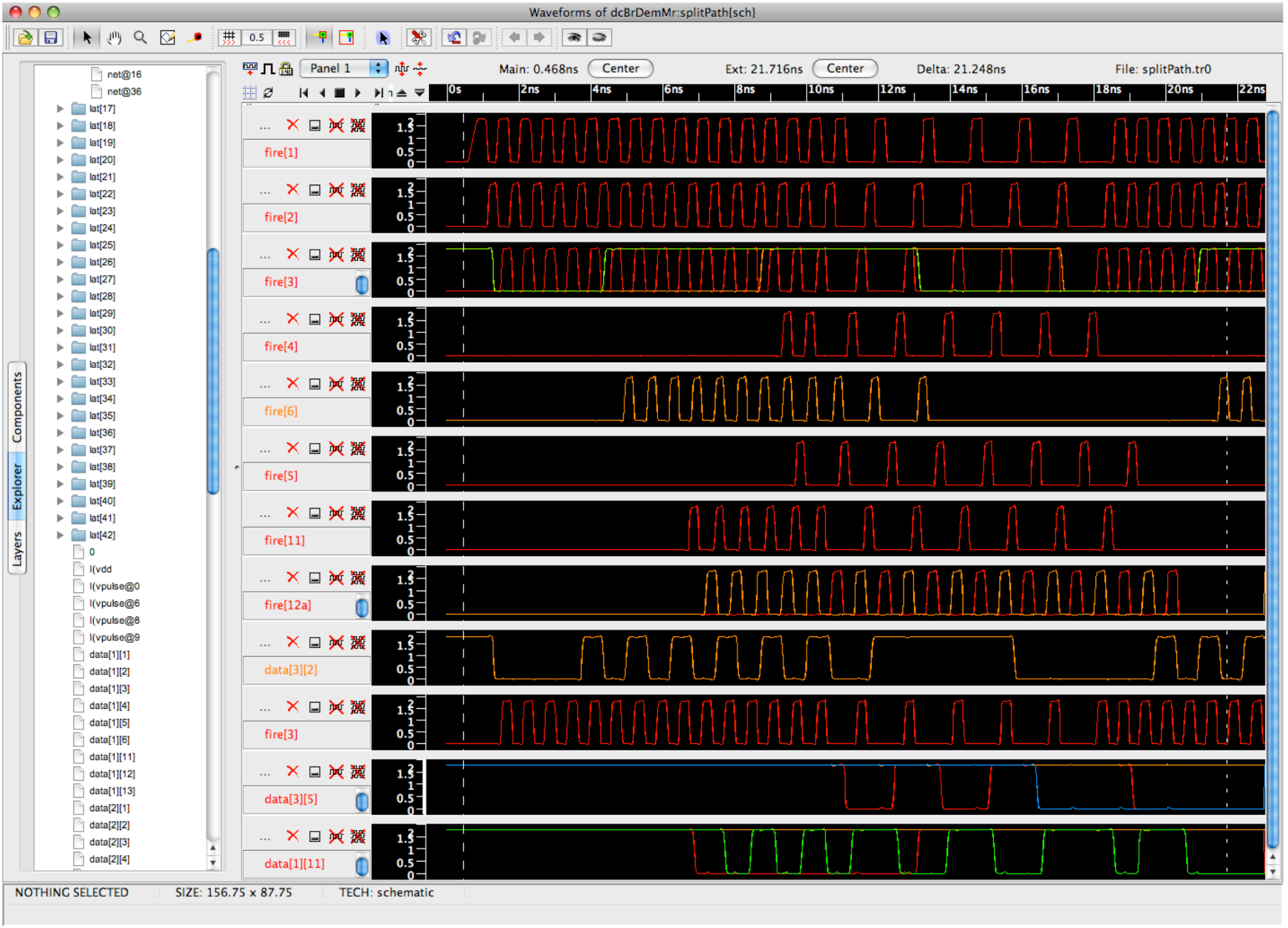


sucOrDri10

HI is full successor driver

ies 18 June 2010




Waveforms of dcBrDemMr:splitPath{sch}
Panel 1
Main: 0.468ns
Center
Ext: 21.716ns
Center
Delta: 21.248ns
File: splitPath.tr0
0s
2ns
4ns
6ns
8ns
10ns
12ns
14ns
16ns
18ns
20ns
22ns
net@16
net@36
lat[17]
lat[18]
lat[19]
lat[20]
lat[21]
lat[22]
lat[23]
lat[24]
lat[25]
lat[26]
lat[27]
lat[28]
lat[29]
lat[30]
lat[31]
lat[32]
lat[33]
lat[34]
lat[35]
lat[36]
lat[37]
lat[38]
lat[39]
lat[40]
lat[41]
lat[42]
0
I(vdd
I(vpulse@0
I(vpulse@6
I(vpulse@8
I(vpulse@9
data[1][1]
data[1][2]
data[1][3]
data[1][4]
data[1][5]
data[1][6]
data[1][11]
data[1][12]
data[1][13]
data[2][1]
data[2][2]
data[2][3]
data[2][4]
Components
Explorer
Layers
fire[1]
fire[2]
fire[3]
fire[4]
fire[6]
fire[5]
fire[11]
fire[12a]
data[3][2]
fire[3]
data[3][5]
data[1][11]
1.5
1
0.5
0
NOTHING SELECTED
SIZE: 156.75 x 87.75
TECH: schematic

ANSWER SHEET – due by beginning of class on **16 November 2010** **7**

Name _____

Turned in on Date _____

Answer from simulation:

My simulation uses _____ technology – e.g. 180 MOSIS

AT THE INPUT – stage 2 or 3

Immediately after master clear, `data[1:3][2]` are all <HI> <LO> . Why?
Hint: look inside the latches.

100 psec before the first pulse on `fire[3]`, `data[1:3][2]` are <HI> <LO> . Why?

The flow rate at `fire[2]` is _____ GDI/s (maximum) and _____ GDI/s (average).

WHERE DO DATA VALUES GO? - stage 3

The first _____ data values to reach stage 3 go nowhere.

The next _____ data values to reach stage 3 go only to stage 6.

The next _____ data values to reach stage 3 go to both stages 4 and 6.

The next _____ data values to reach stage 3 go only to stage 4.

AT THE DEMAND MERGE – stage 12

The first data to reach the demand merge come from <stage 5> <stage 11>

The first _____ data elements out of stage 12 came from the <upper> <lower> pipeline.

Pulses on `fire[12A,12B]` alternate when _____

At 12, the last _____ data elements in a group came from the <upper> <lower> pipe.

OUTPUT VS INPUT – stage 2 and 14

The flow rate at `fire[14]` is _____ GDI/s (maximum) and _____ GDI/s (average).

Why do these values differ from the values at stage 2?

What causes the irregular pattern of pulses at stage 14? At at stage 2?