# Digital System Design

by

Dr. Lesley Shannon

Email: lshannon@ensc.sfu.ca

Course Website: http://www.ensc.sfu.ca/~lshannon/courses/ensc350

***Simon Fraser University***

Slide Set: 13
Date: March 16, 2009

# Slide Set Overview

- Floating point
- Asynchronous Circuits

# Floating Point Numbers

- Single Precision Floating point
  - Sign Bit
    - 1 bit
  - Exponent
    - 8 bits
    - Exponent bias 127
  - Mantissa
    - 23 bits

# The cost of Floating Point Numbers

- Resources for complex 32-bit operations
  - Add
    - Fixed Point: 116 Flipflops, 106 LUTs
    - Floating Point: 1182 Flipflops, 1160 LUTs
  - Subtract
    - Fixed Point: 116 Flipflops, 106 LUTs
    - Floating Point: 1182 Flipflops, 1160 LUTs
  - Multiply
    - Fixed Point: 170 Flipflops, 154 LUTs
    - Floating Point: 1732 Flipflops, 1530 LUTs

# The cost of Floating Point Numbers

- Floating point is not only bigger, it’s slower
- Anywhere from 2 to 4 times slower
- The point is that Floating point comes at a cost
  - So make sure you need it
    - The dynamic range should be a ***necessity***
    - This will typically only apply to high performance/scientific computing
- FYI Double precision is:
  - 1 sign bit
  - 11 exponent bits
  - 52 mantissa bits

# Intro to Asynchronous Circuits

Most real digital systems are purely synchronous, that is, they operate on a single clock. These are easy to design, but have some disadvantages. It is also possible to design a circuit that does not depend on a clock. In this slide set, we will talk about such circuits. Even though you may not see a huge system that is entirely asynchronous any time in the near future, you will likely come across small asynchronous circuits, so it is important that you have seen them before.

Problems with Synchronous Design:

1. In Synchronous design, clock period is dictated by the longest path

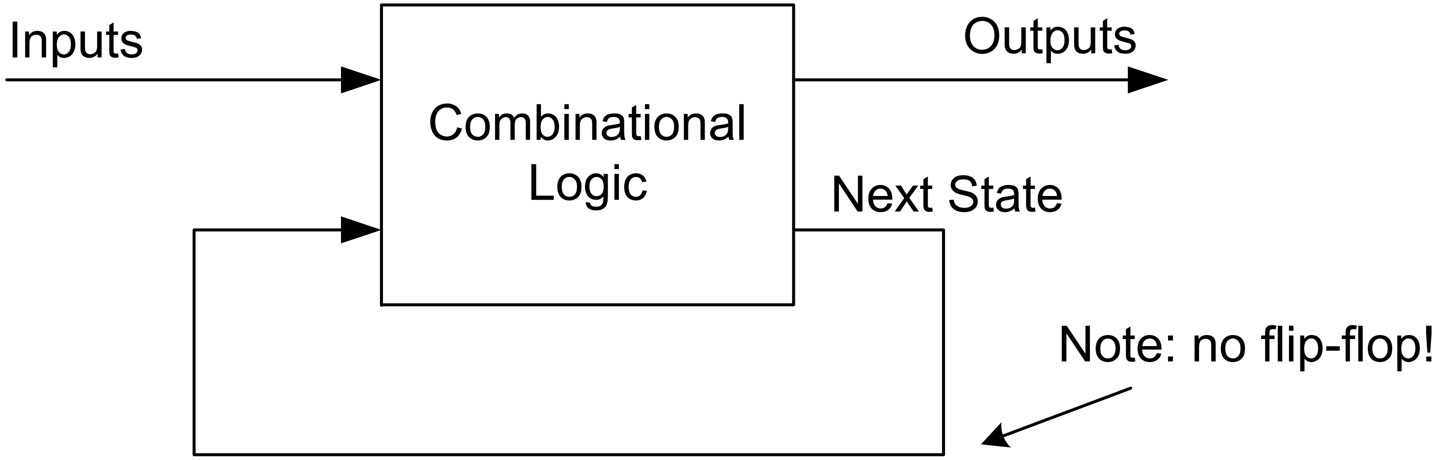
   “in the MIPS R10000, there was a single long path in the processor’s instruction fetch hardware.  This long path was limiting the achievable clock frequency, but the engineers couldn’t find it!  They finally found it and shortened it for the MIPS R12000”.

2. Since everything happens on the clock edge, instantaneous power is a problem

3. Excessive noise at the frequency of the clock

4. In large chips, distributing the clock is difficult

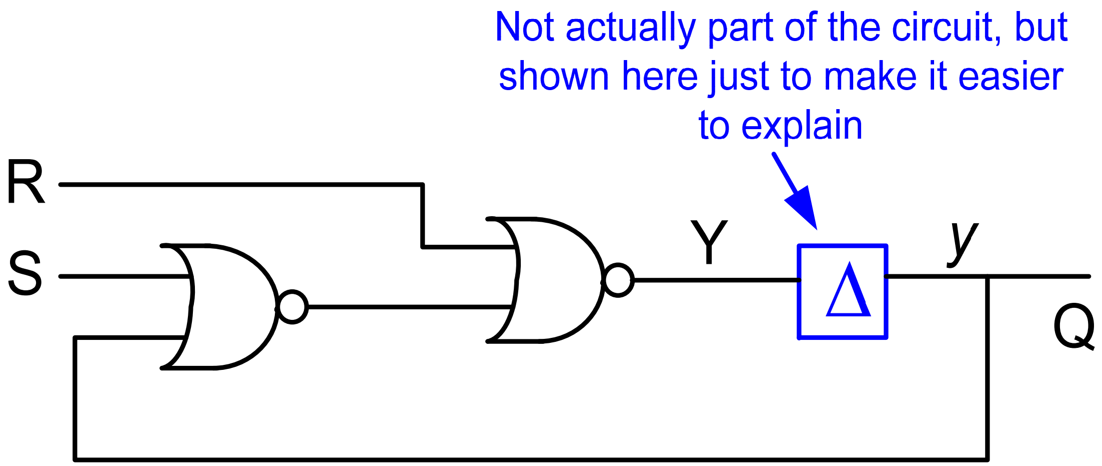
# Asynchronous State Machines



Note: no flip-flop!

Exactly the same as a synchronous state machine except there are no flip-flops. The value of the next_state wires indicate the state

# Simple Example



Start with Y=y=S=R=0.  Then,

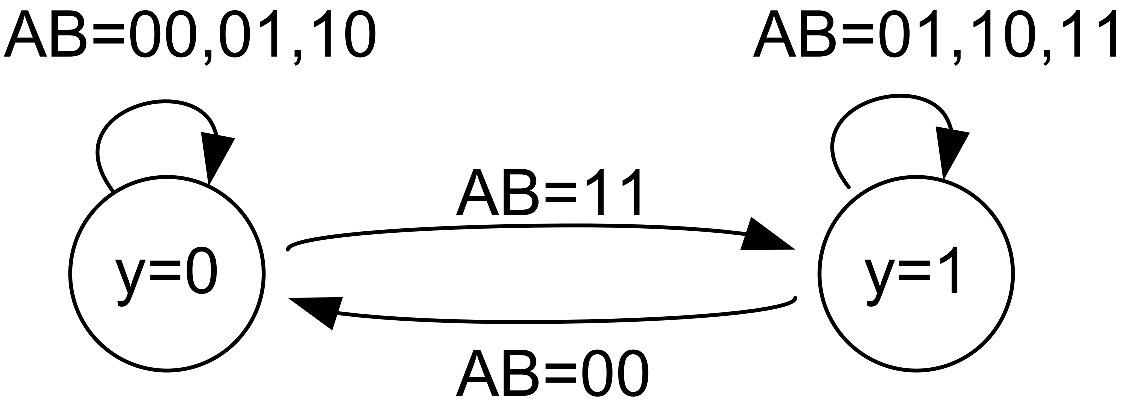
- What happens if R changes to 1?
- What happens if S now changes to 1?
- What happens if R then changes to 0?

# Designing Asynchronous State Machines

Starting from a State Diagram, you design an asynchronous state machine exactly as you did a Synchronous State Machine.

The main difference is that you don’t have flip-flops to hold your current state (use wires instead)

This the state machine for a Muller C Element (this is important when we talk about Asynchronous Datapaths):



In the previous example, we can see that the machine is:

Stable when one of the following is true:
    we are in state S1 and AB=00, 01, or 10
    we are in state S2 and AB=01, 10, or 11

Unstable when one of the following is true:
    we are in state S1 and AB=11
    we are in state S2 and AB=00

Why is this?  When we are in state S1, and we get a 11, we immediately go to S2.  Once we reach S2, we are stable again.

When we are in state S2, and we get a 00, we immediately go to S1. Once we reach S1, we are stable again.

This is a fundamental difference between a synchronous and asynchronous state machine:

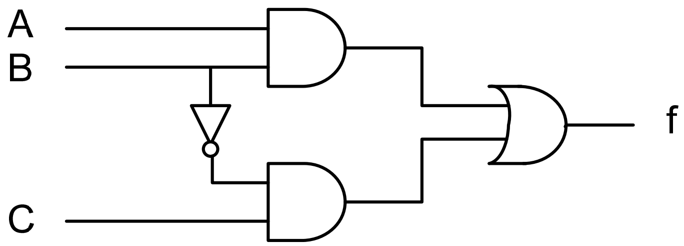
In a synchronous machine, when the “next state” is not the same as the “current state”, we wait until the next rising clock edge to actually change state.

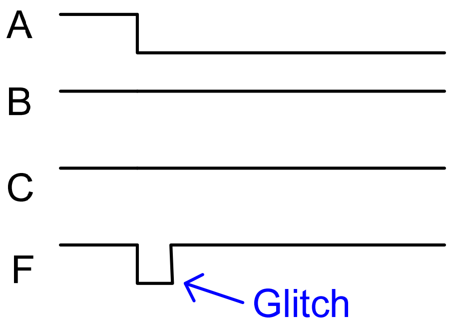
In an asychronous machine, when the “next state” is not the same as the “current state”, we immediately make a transition to the next state.

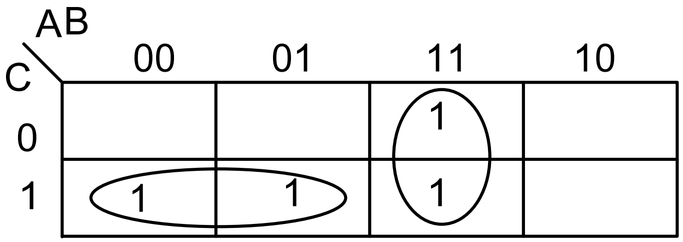
- The next state may also not be stable, in that case, we would calculate a new next state, and immediately make a transition there, and so on….
- To be useful, we have to eventually reach a stable state.

Two more differences between an asynchronous and synchronous machine:

1. Need to watch out for “hazards” (glitches)

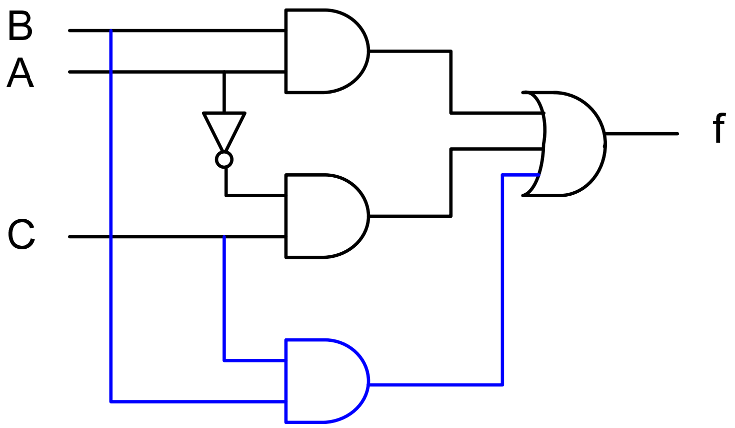


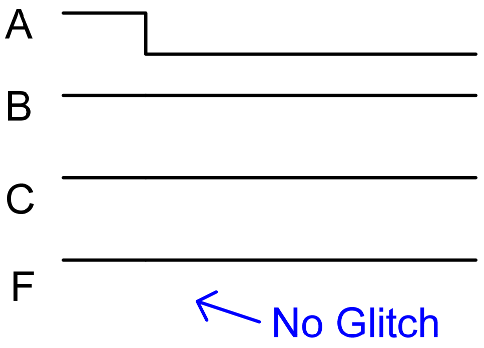




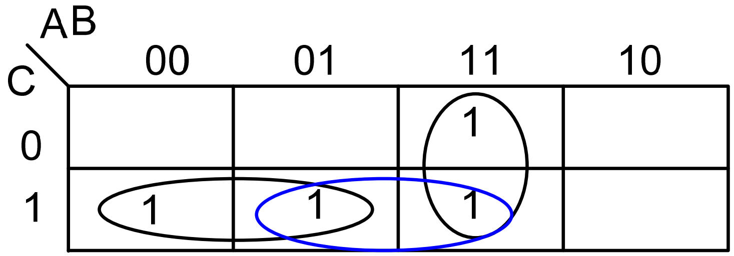
| C \ AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | | | 1 | |
| 1 | 1 | 1 | 1 | |

# If we add an extra “cover”, this eliminates the glitch





No Glitch



| C \ AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | | | 1 | |
| 1 | 1 | 1 | 1 | |

Glitches were never a problem for synchronous circuits; as long as they stabalized by the next rising clock edge, we were fine. Here, if the output goes low even for a short time, this may cause us to go into an unexpected state.

Moral: Make sure your next state logic and output logic is glitch-free!

2. In a synchronous machine, a transition from any state to any other state is allowed. But in an asynchronous state machine, a state transition from Sa to Sb is only allowed if Sa and Sb are “adjacent” (the state encodings differ in exactly one bit).

So, we can transition from

000 -> 010

But we can not make a direct transition from:

000 -> 011

Why not? The following example will make it clear.

This is what would happen if we chose the “bad” state assignment:

| Present State *y2 y1* | Next State | | Output |
|---|---|---|---|
| | w=0 | w=1 | |
| 00 | 00 | 01 | 0 |
| 01 | 10 | 01 | 1 |
| 10 | 10 | 11 | 1 |
| 11 | 00 | 11 | 0 |

Bad Assignment

| Present State *y2 y1* | Next State | | Output |
|---|---|---|---|
| | w=0 | w=1 | |
| 00 | 00 | 01 | 0 |
| 01 | 11 | 01 | 1 |
| 11 | 11 | 10 | 1 |
| 10 | 00 | 10 | 0 |

Good Assignment

Problem occurs when current state = D (bottom row) and w changes to 0

In that case, *y2y1* is supposed to change from 11 to 00

But, it is unlikely that *y2* and *y1* will change at exactly the same time

This is what would happen if we chose the “bad” state assignment:

| Present State *y2 y1* | Next State w=0 | Next State w=1 | Output |
|---|---|---|---|
| 00 | 00 | 01 | 0 |
| 01 | 10 | 01 | 1 |
| 10 | 10 | 11 | 1 |
| 11 | 00 | 11 | 0 |

Bad Assignment

| Present State *y2 y1* | Next State w=0 | Next State w=1 | Output |
|---|---|---|---|
| 00 | 00 | 01 | 0 |
| 01 | 11 | 01 | 1 |
| 11 | 11 | 10 | 1 |
| 10 | 00 | 10 | 0 |

Good Assignment

Suppose *y1* changes first:

For a short time, we would enter state *y2y1*=10

But since w=0, we would stay there

Suppose *y2* changes first:

For a short time we would enter state *y2y1*=01

But since w=0, we would go to 10 and stay there

“Race Condition”

Rule: All state transitions must involve only one change in variable

In other words, the Hamming Distance between present state and next state must be 1

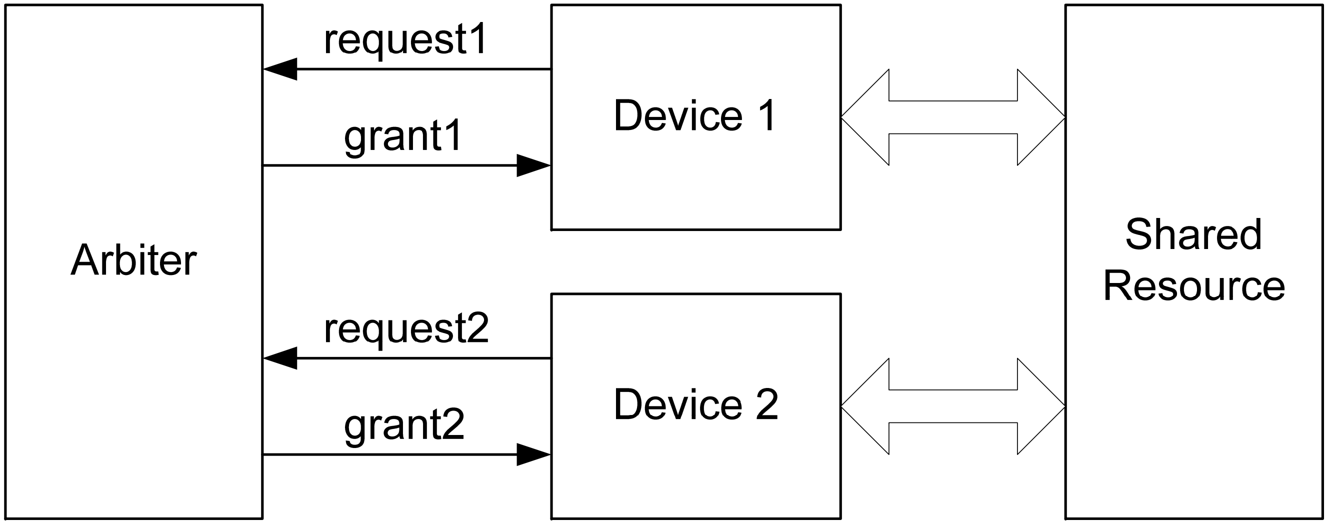
If you can’t find an appropriate state assignment, you have to add extra states!

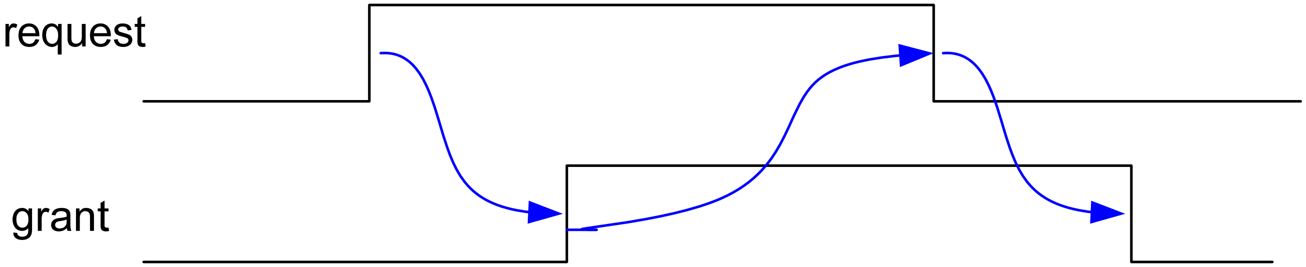
Implicit in the previous example was the assumption that both inputs don’t change at exactly the same time.

Another way of saying this is that inputs operate in *fundamental mode*

- This makes it easy to analyze and design asynch. state machines and in practice is always true.
- We will assume all inputs operate in fundamental mode

# Another design example we will do on the board: Design an arbiter!





# Asynchronous Logic in VHDL

Two alternatives:

```
process(r1, r2)
  variable y2,y1 : std_logic;
begin
   y2 := ( (not r1) and r2 and (not y1)) or (y2 and r2);
   y1 := r1 and not y2;
   g2 <= y2;
   g1 <= y1;
end process;
```

Or (not in a process):

```
y2 <= ( (not r1) and r2 and (not y1)) or (y2 and r2);
y1 <= r1 and not y2;
g2 <= y2;
g1 <= y1;
```

Are these synthesizable?

Depends on your tools. For the latest version of Quartus II (which we are using), the answer is YES!

- But, you get a warning:

  Warning: Timing Analysis is analyzing one or more combinational loops as latches

Other tools: some would be able to synthesize this and some wouldn’t. To be the most portable, use explicit equations rather than a process.

# Datapaths

In a synchronous system, signals must arrive before “the next rising clock edge”. It is up to the user to make sure the clock speed is slow enough that all signals “make it” in time.

In an asynchronous datapath, we don’t have a clock.

- Each element performs an operation and sends 2 things to its neighbour:
  - The result of the operation
  - An indication that it is done

Two options to do this: ready/ack and transition signaling

Option 1:

When data is produced, *ready* is toggled. When receiver sees *ready*, it accepts the data and toggles to *ack* to indicate that it has accepted data

Data could be a bus, and you only need a single *ready* wire and a single *ack* wire

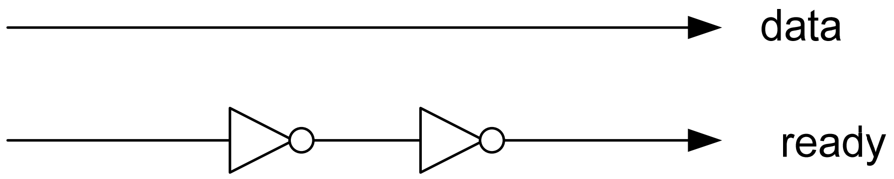
**Problem with this option:** What if the *ready* wire is faster than the data wire(s)?

- data will be read before it becomes valid

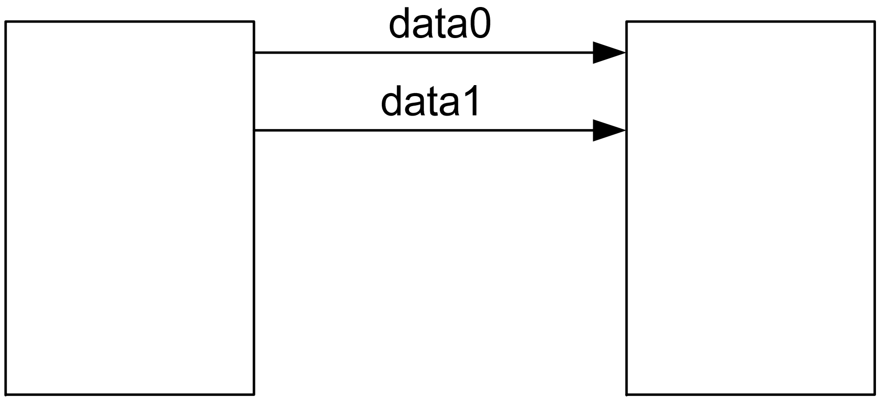
**Solutions:**

a) route *data, ready, ack* parallel to each other on the chip

b) intentionally add delay to the *ready* line



Option 2: Transition Signaling:



When transmitter wants to send a 0, it toggles data0
When transmitter wants to send a 1, it toggles data1
Receiver listens for a change in either data0 or data1

This works independently of wiring delays!

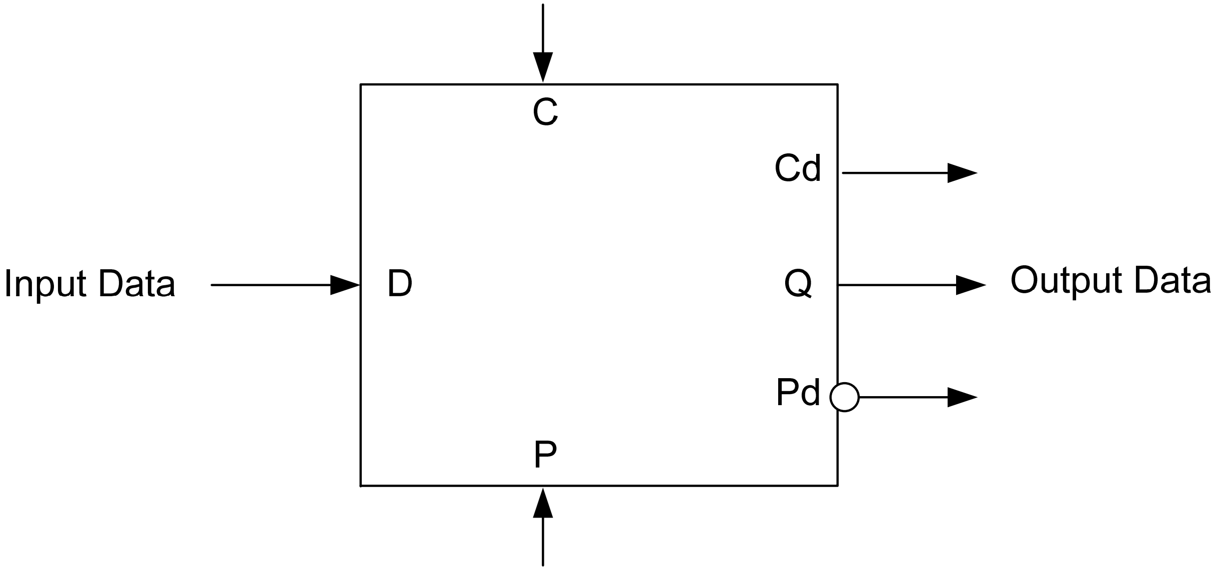
Problem: need 2 wires for each bit to be transmitted

# Micro-pipelines

A structured way to implement asynchronous datapaths
(uses request and ack lines)

This has been used in large processors, and probably is the most straight-forward way to achieve asynchronous datapaths.

Define a new “clockless register”



Transition on P (1 to 0 *or* 0 to 1) puts register in “propagate” mode

- in propagate mode, value in D is propagated to Q

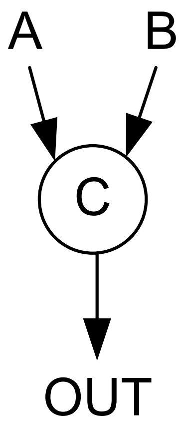
Transition on C (1 to 0 *or* 0 to 1) puts register in “hold” mode

- in hold mode, Q holds its value

Cd is a delayed version of C

Pd is a delayed version of $\overline{P}$

A Muller C-Element:

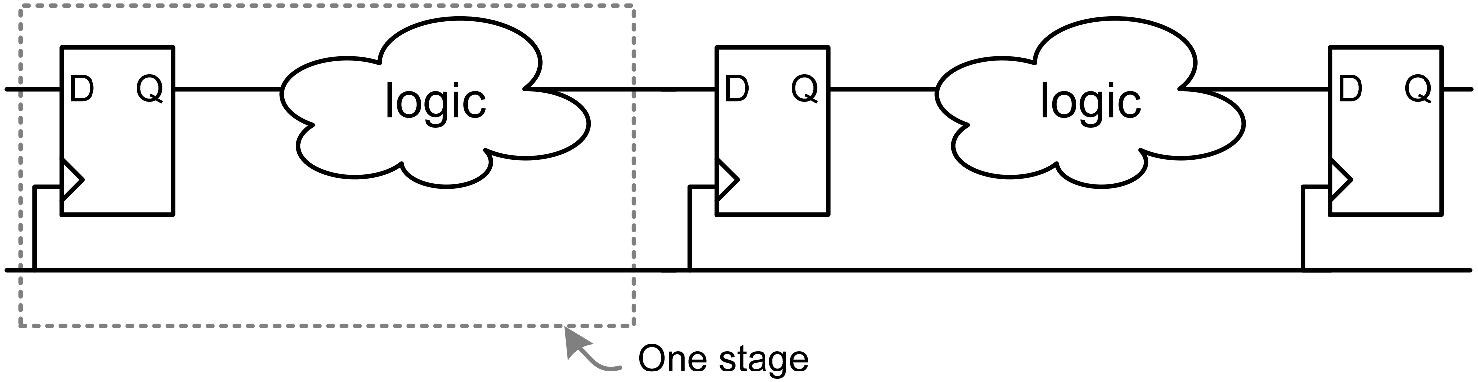


OUT is 0 if all inputs are 0
1 if all inputs are 1
unchanged otherwise

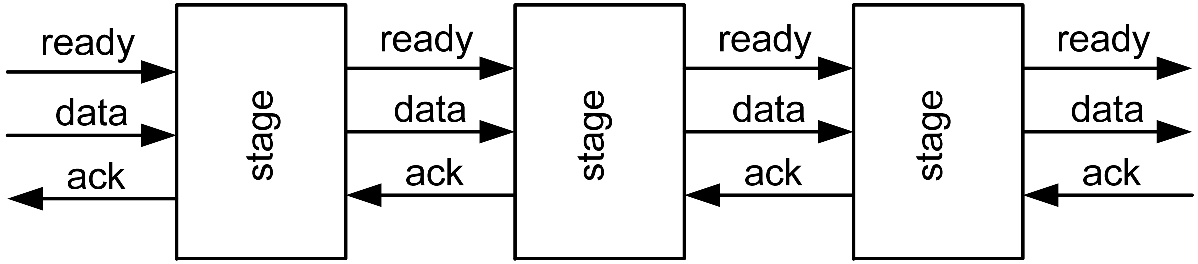
A transition on the output occurs after a transition on all inputs
(we showed this earlier in this slide set)

We will put these together to form a datapath.

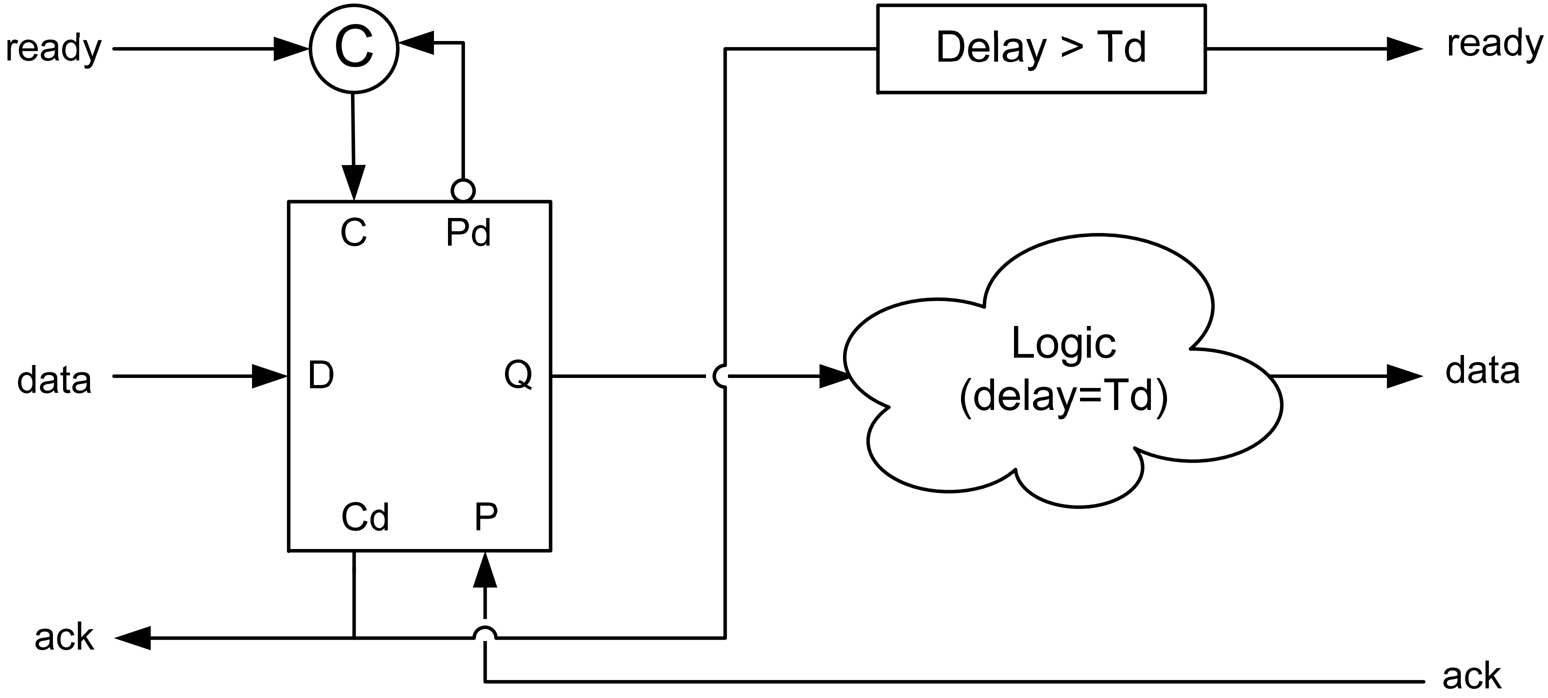
But first, just a reminder of a synchronous datapath:



Asynchronous version:



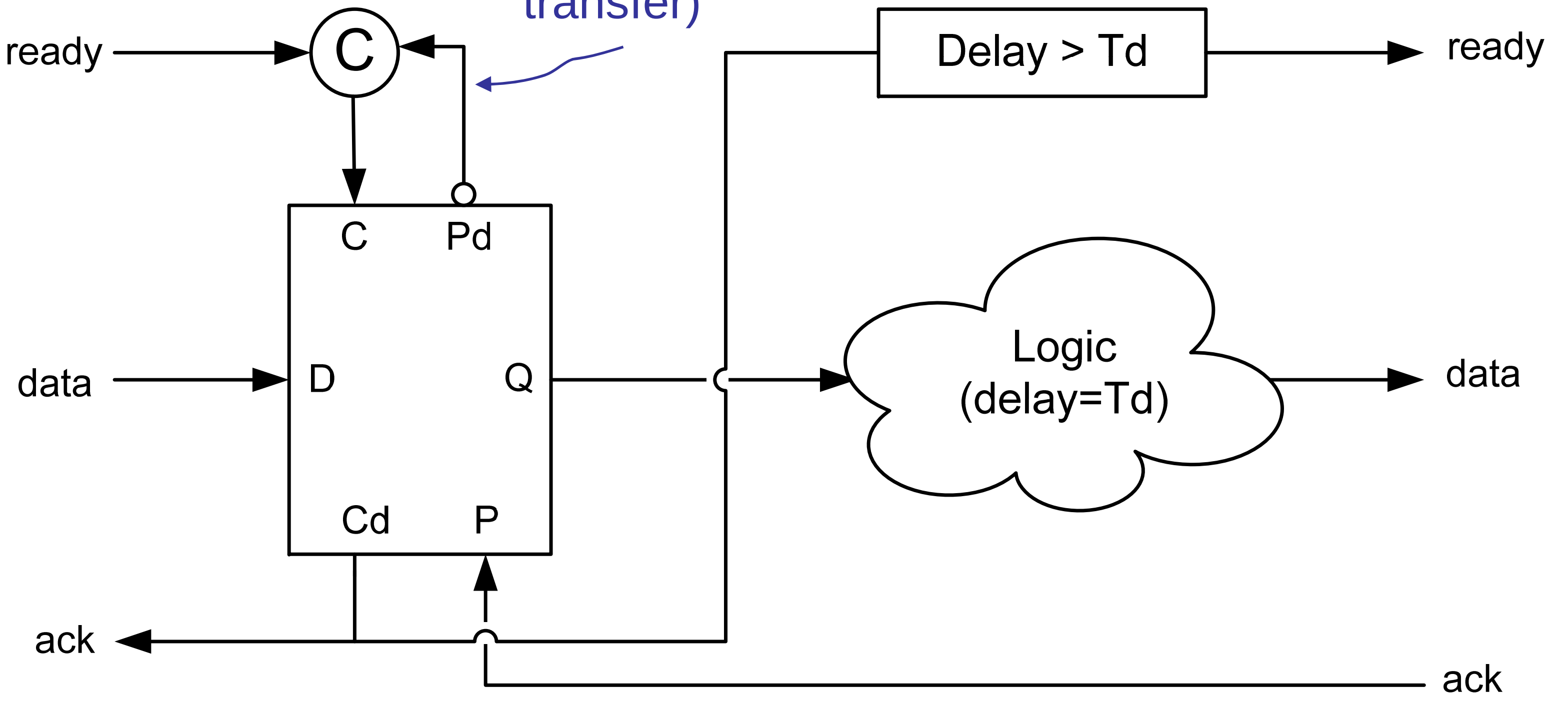
One Stage looks like this:



Operation of Micro-pipeline Stage:

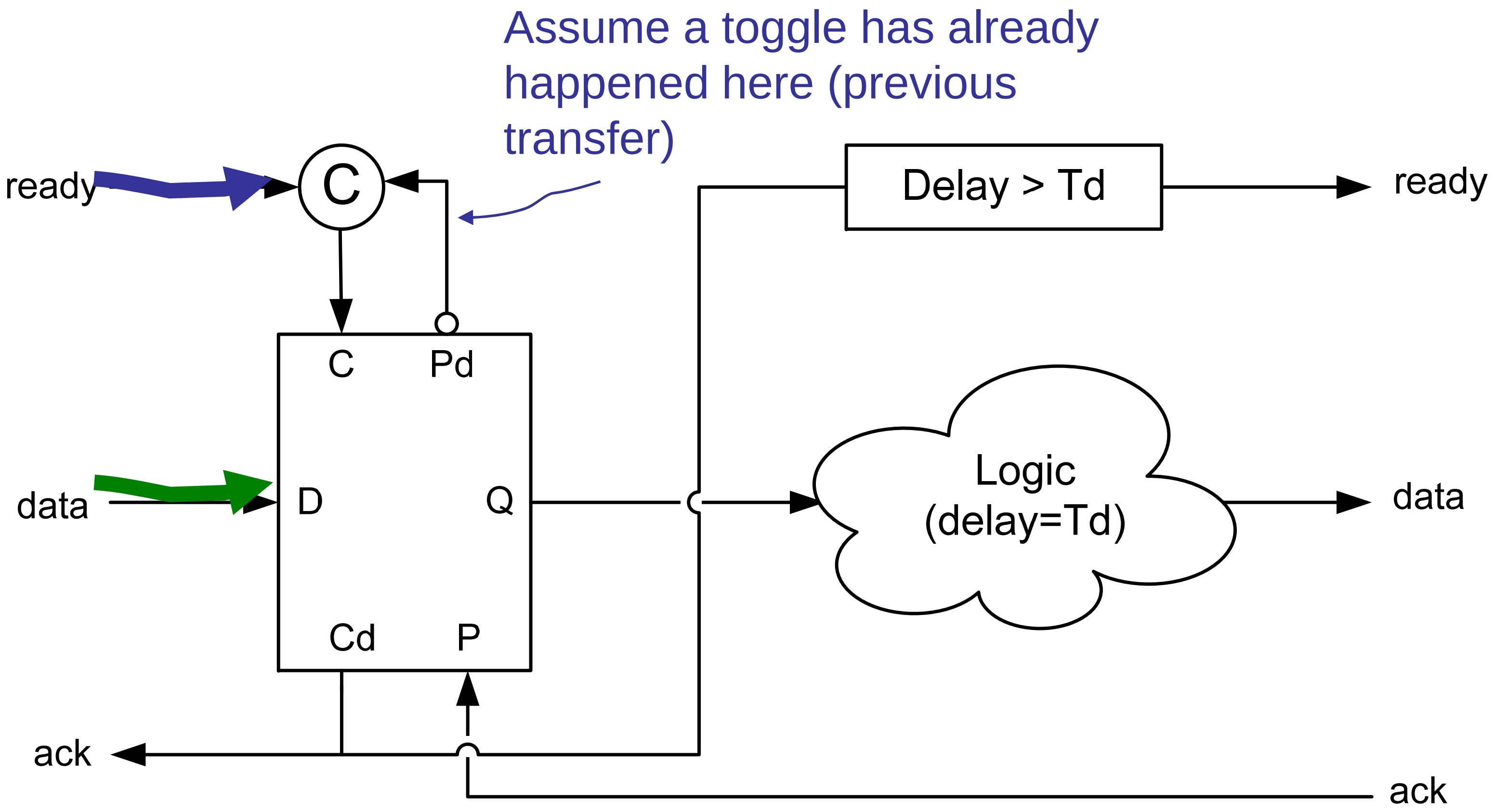
1. Previous stage drives *data* and toggles *ready*. Assume an event (toggle) has already occurred on Pd (at the end of the previous transfer)
2. Since an event has occurred on both C-element inputs, the C-element output toggles. This causes latch to hold (“latch-in”) the current input data.
3. A short time later, Cd toggles, which returns an event on *ack* to the previous stage.
4. Td later, the output *data* is valid.
5. Some time later, the output *ready* signal (to the next stage) toggles
6. The next stage latches and uses the data. When it reaches Step 3, our *ack* is toggled, causing our register to go back into propagate mode.

Illustrated example of what is on the previous slide:

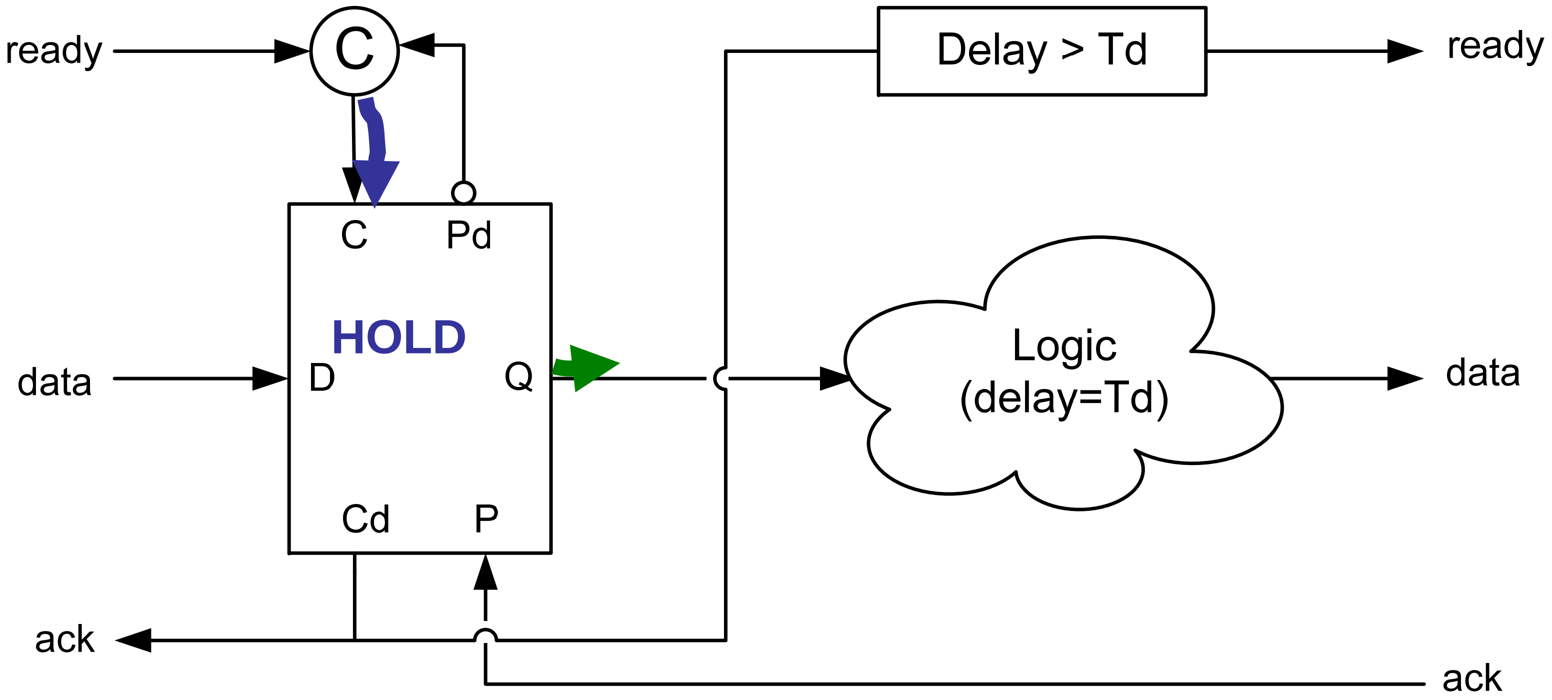


Assume a toggle has already happened here (previous transfer)

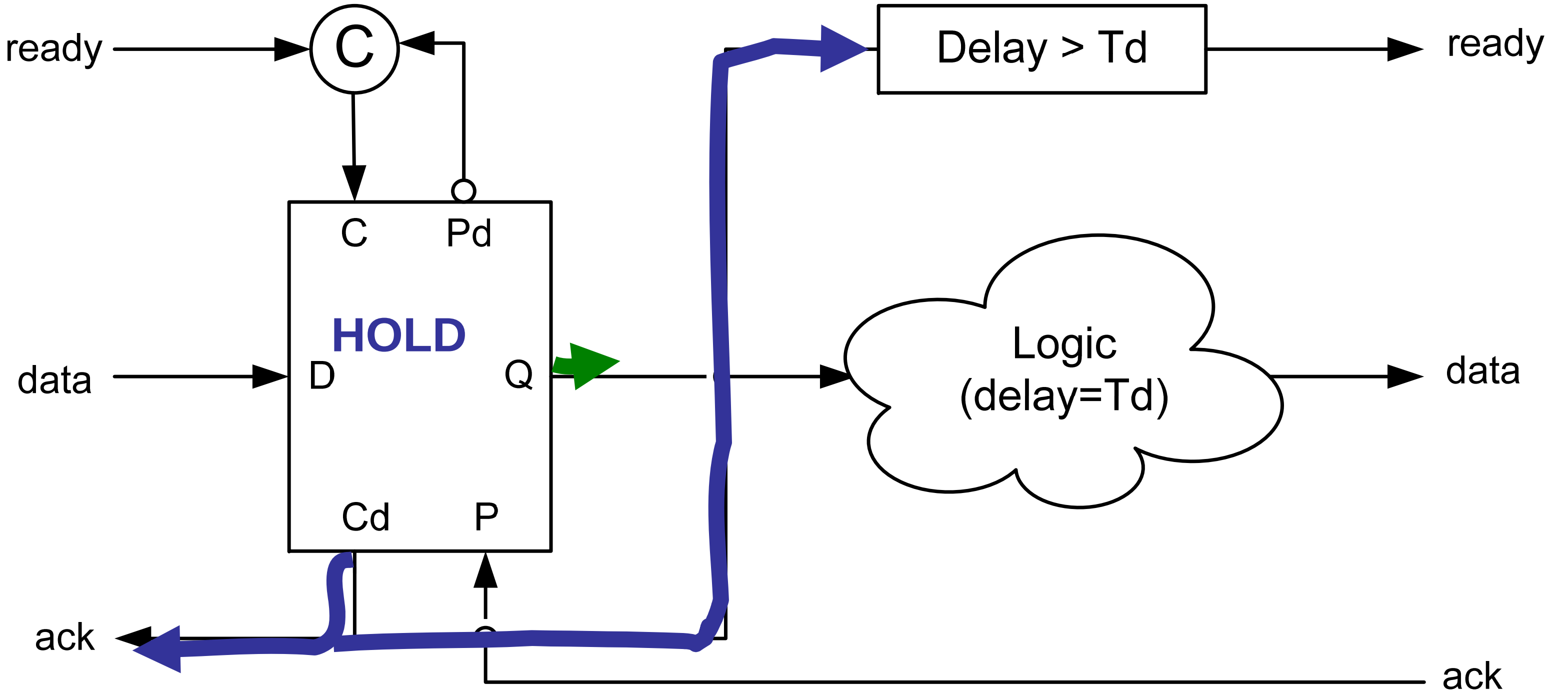
# Step 1: *data* and *ready* comes from previous stage



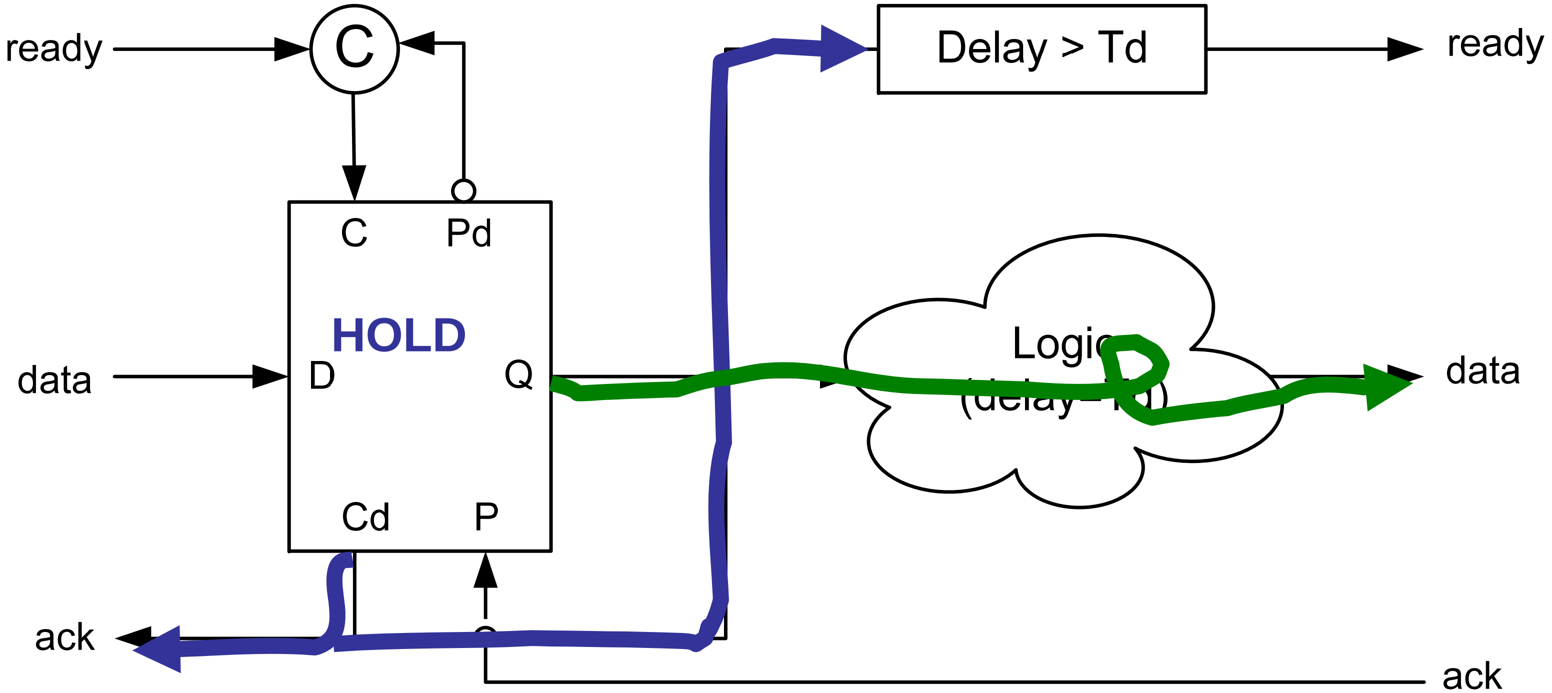
# Step 2: Output of C element toggles, latch goes into “hold” mode



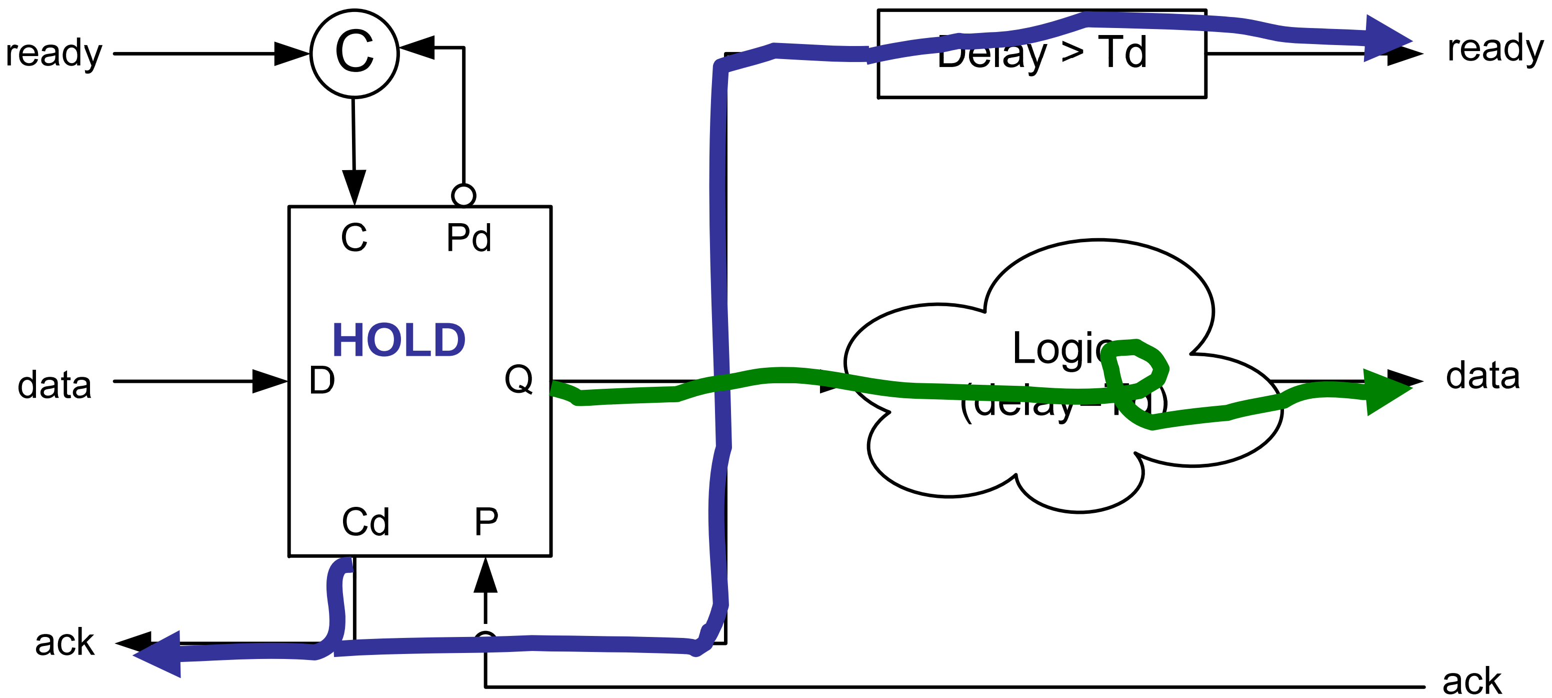
# Step 3: A short time later, Cd toggles, sending *ack* back to predecessor



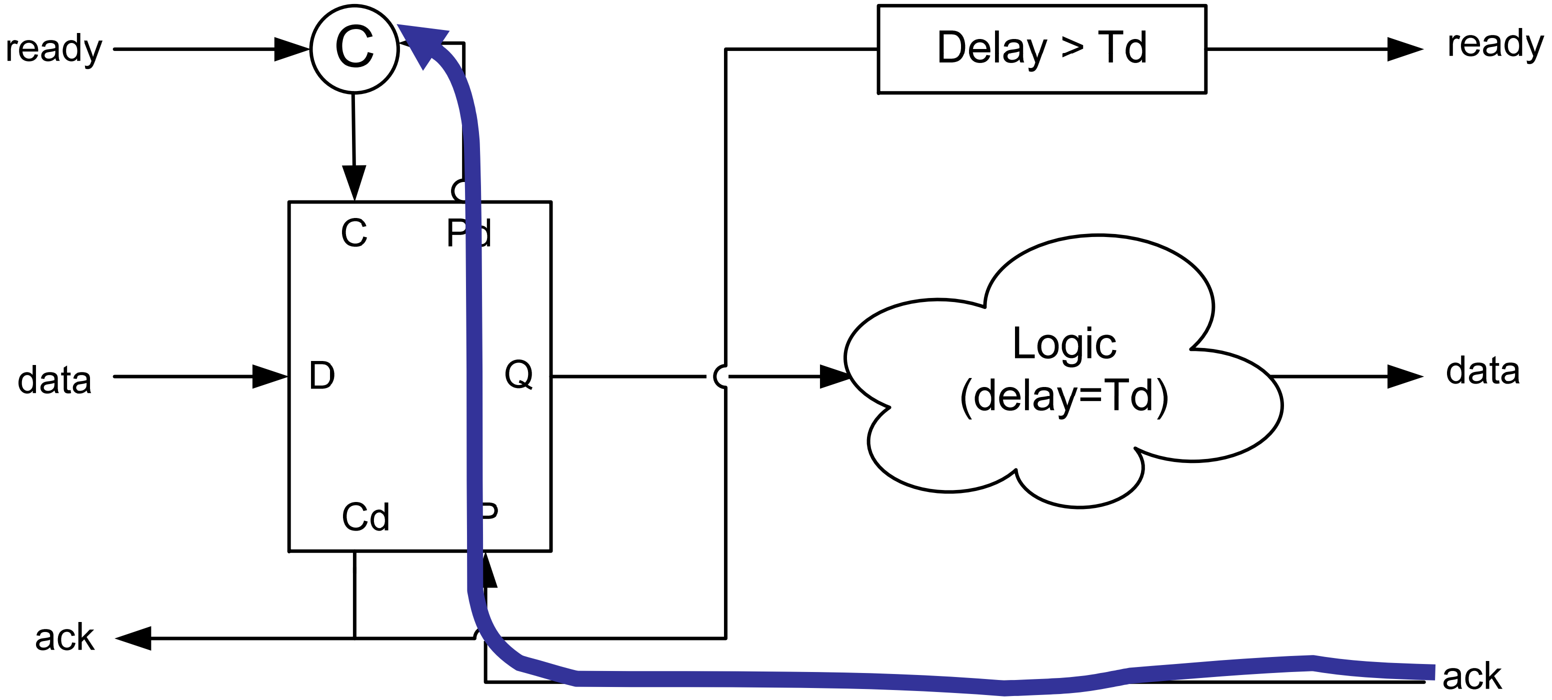
# Step 4: Td later, output *data* is valid



# Step 5: A little bit later, *ready* output toggles



Step 6:The next stage now sees ready go high, so it starts with Step 1, latching in data. Eventually, it reaches Step 3, sending us back a toggle on the ack signal



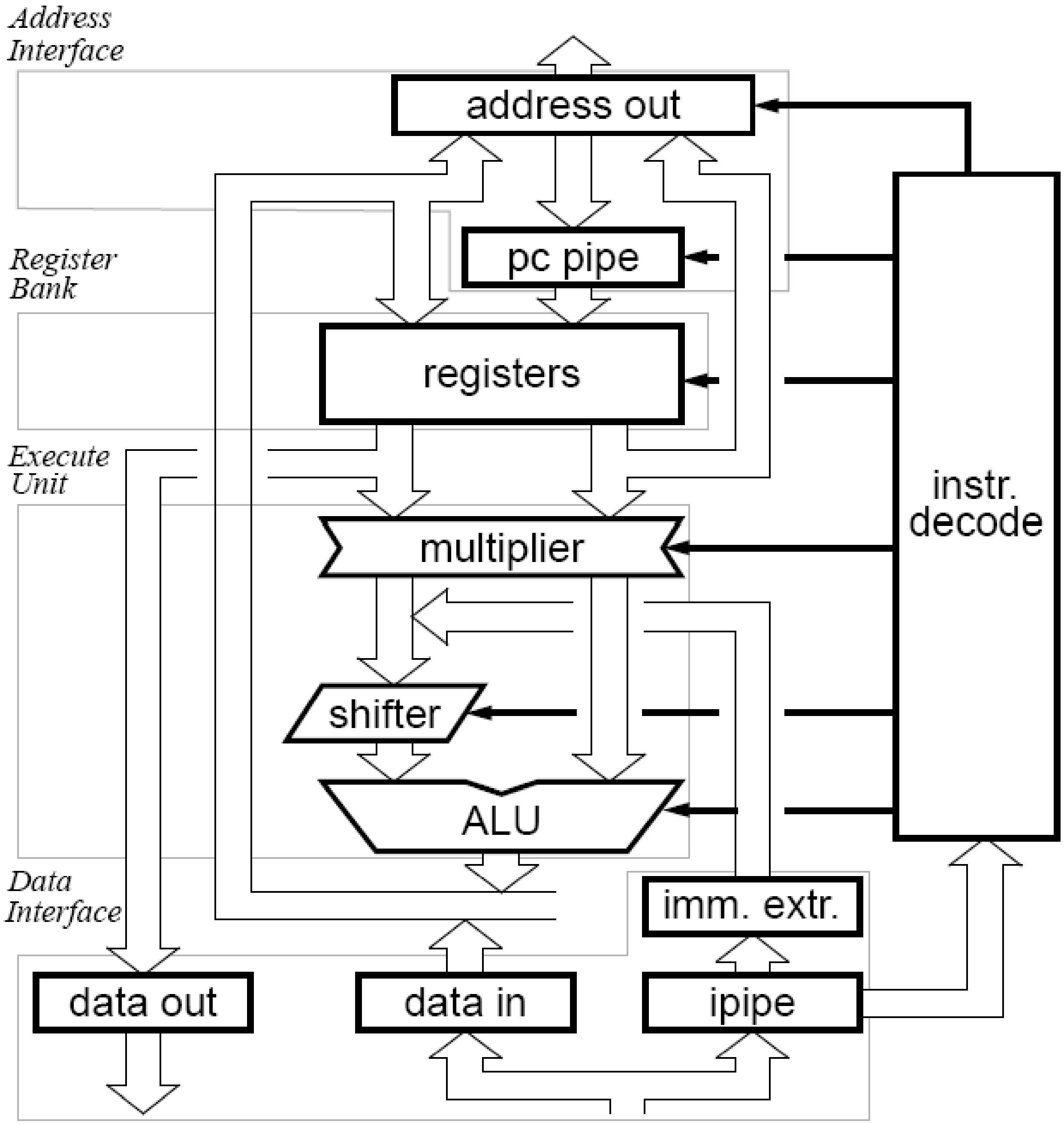
And we’re back where we started, ready for the next transfer

Steps 1 to 6 can be thought of as a “clock cycle”

But, notice that there is no global clock

- The speed is limited by the delay of the logic stage

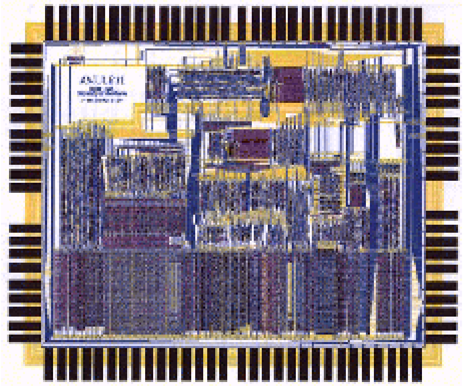
In the steady state, if the logic stages are all equal (rarely does this happen), we won’t run any faster than a synchronous system with a well-tuned clock



This is the Amulet Microprocessor

Built with Micropipeline techniques

58,000 Transistors



# Globally Asynchronous Locally Synchronous (GALS)

Each sub-block on a chip is designed as a synchronous circuit
- But, each operates on its own clock

When you connect these together, the subcircuits are asynchronous with respect to each other.

A bit tricky:
- Need to worry about interface between two synchronous blocks
    - Need some clever circuit design (what if the producer and consumer run at slightly different speeds?)
- Metastability is a concern: as far as any given subcircuit is concerned, the signals from other subcircuits could happen any time.

# Summary of this Slide Set

Asynchronous State Machines

- Like normal state machine design, but no clock and no registers
    - State is held in feedback wires
- A few things to be careful of:
    - Avoid glitches
    - Next states should be adjacent
- The level of support in modern CAD tools varies

Asynchronous Datapaths: Micropipelines

Globally Asynchronous Locally Synchronous (GALS)

# Questions

- What does GALS stand for?

- When is it used?

# Questions

- What is a Muller C element? What does the state machine look like?

- What would the circuit look like?

# Questions

- What is a Race condition

- What is a hazard?

# Questions

- How do micropipeline stages work?

- How does a “clockless register” work?

# Questions

- You should be able to draw the state machine and corresponding circuit for an asynchronous circuit (Think of slide 9 as an example).

- In an asynchronous datapath, what 2 things does a module have to send its neighbour?

# Questions

- Give two options for successful data transfers in asynchronous datapaths.