Programmable Microfluidics

Bill Thies

Computer Science and Artificial Intelligence Laboratory

Massachusetts Institute of Technology

Stanford University – October 3, 2007

Acknowledgements

Prof. Saman Amarasinghe

Nada Amin

MIT Computer Science and Artificial Intelligence Laboratory

Prof. Todd Thorsen

J.P. Urbanski

David Craig

MIT Hatsopoulos Microfluids Laboratory

Prof. Jeremy Gunawardena

Natalie Andrew

Harvard Medical School

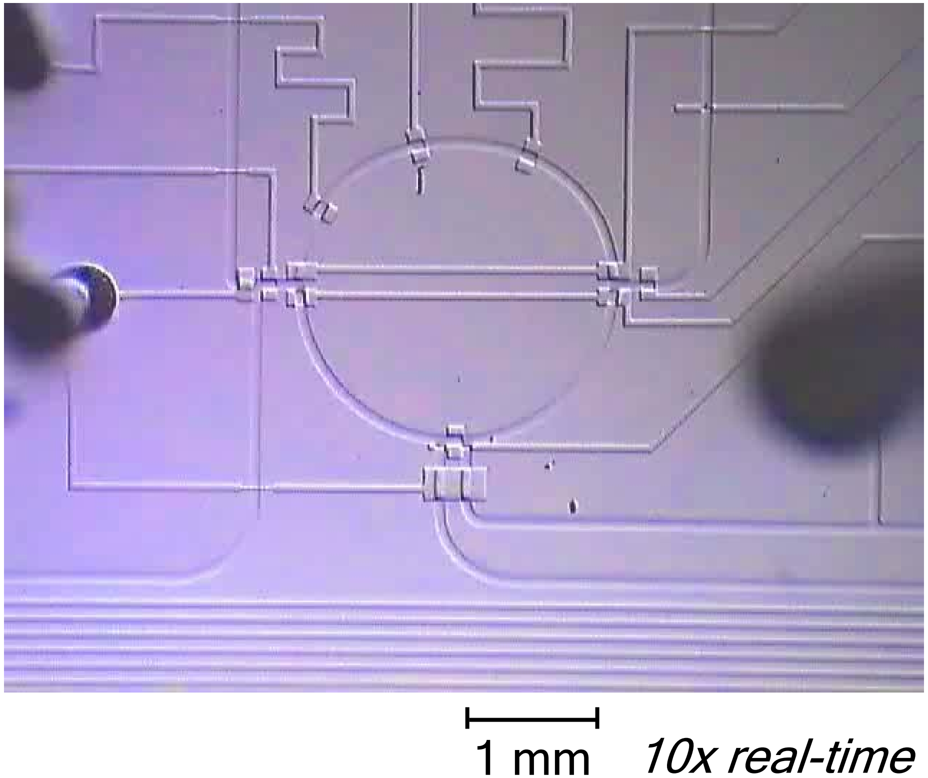
Prof. Mark Johnson

David Potter

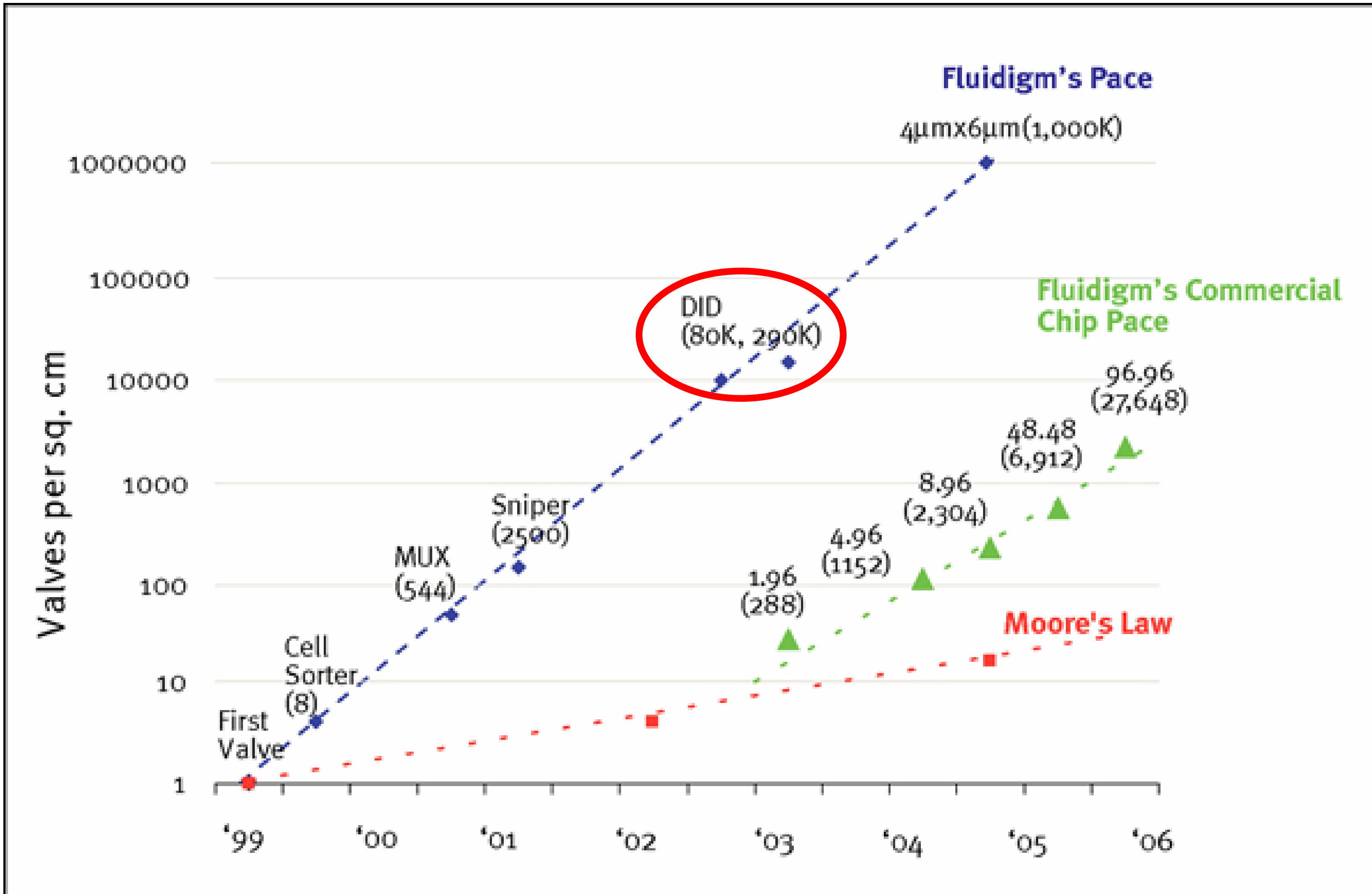
Colorado Center for Reproductive Medicine

Microfluidic Chips

- **Idea: a whole biology lab on a single chip**
  - Input/output
  - **Sensors**: pH, glucose, temperature, etc.
  - **Actuators**: mixing, PCR, electrophoresis, cell lysis, etc.
- **Benefits:**
  - Small sample volumes
  - High throughput
  - Geometrical manipulation
- **Applications:**
  - Biochemistry
  - Cell biology
  - Biological computing

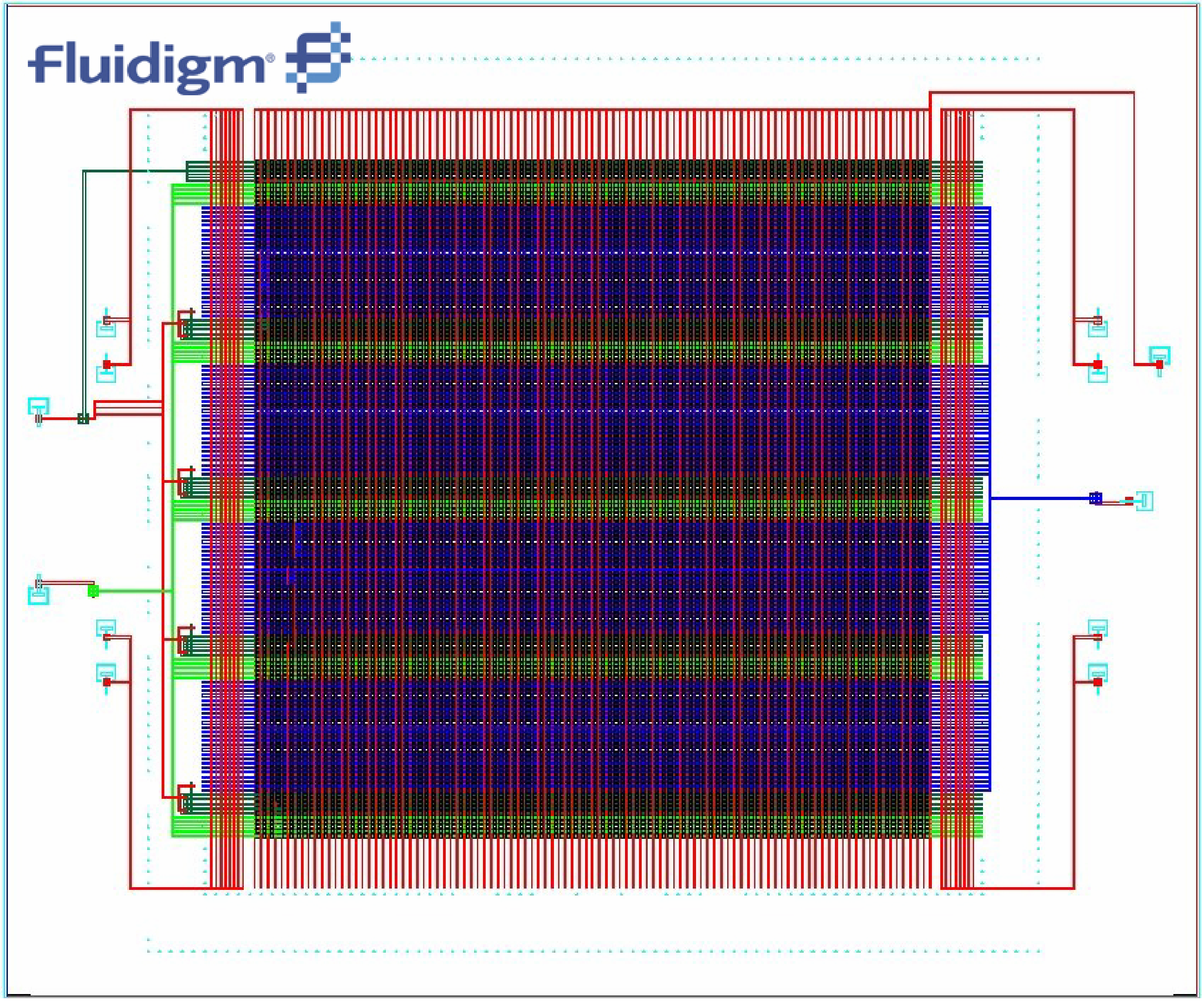


Moore's Law of Microfluidics: Valve Density Doubles Every 4 Months



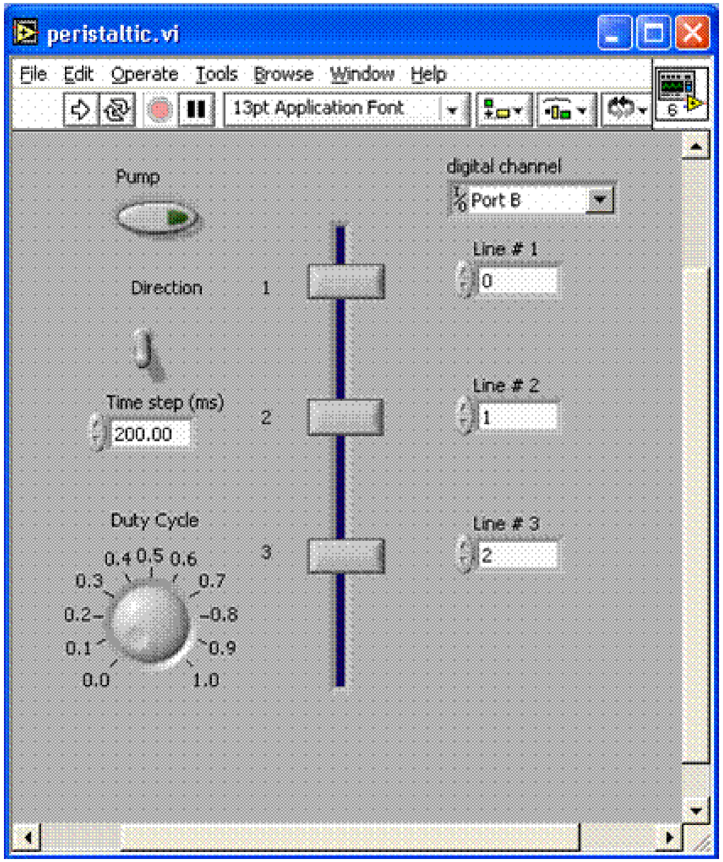
Source: Fluidigm Corporation (http://www.fluidigm.com/images/mlaw_lg.jpg)

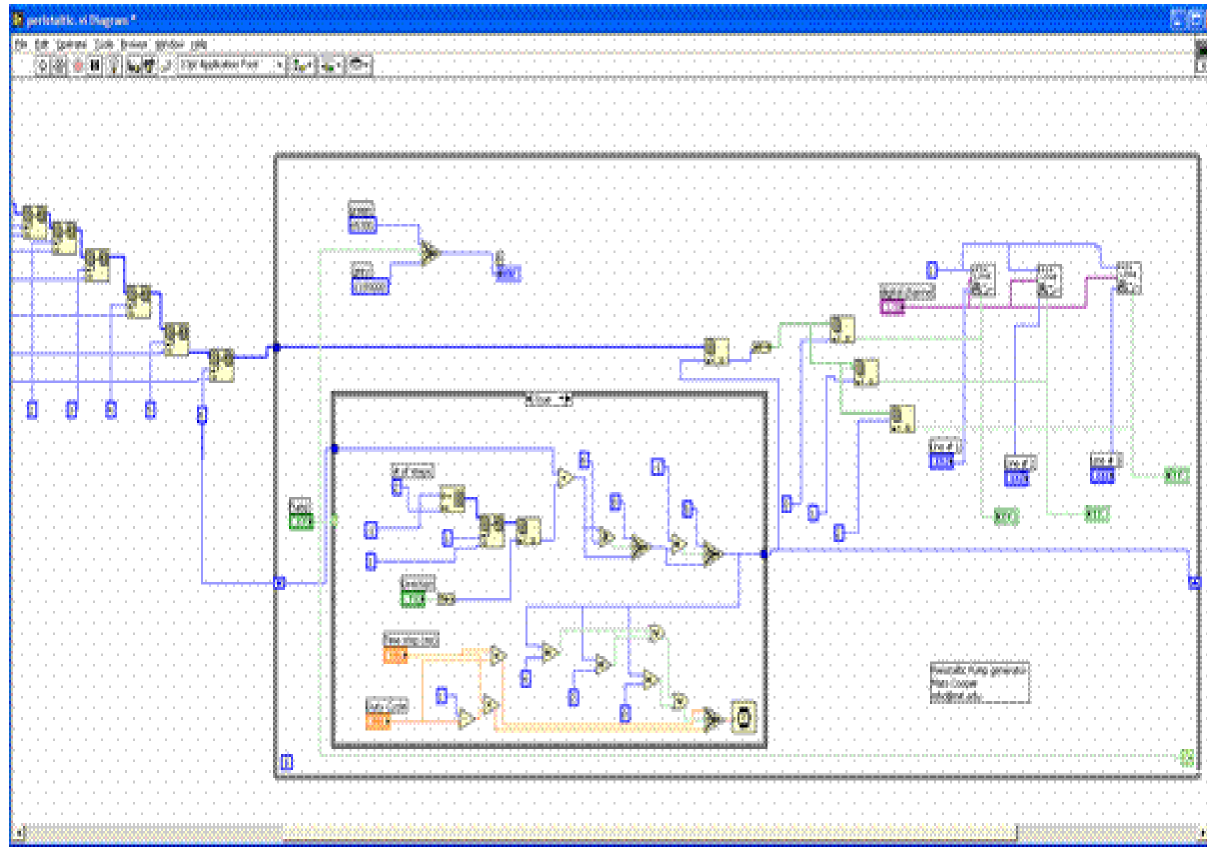
Moore's Law of Microfluidics: Valve Density Doubles Every 4 Months



Source: Fluidigm Corporation (http://www.fluidigm.com/didIFC.htm)

Current Practice:
Expose Gate-Level Details to Users





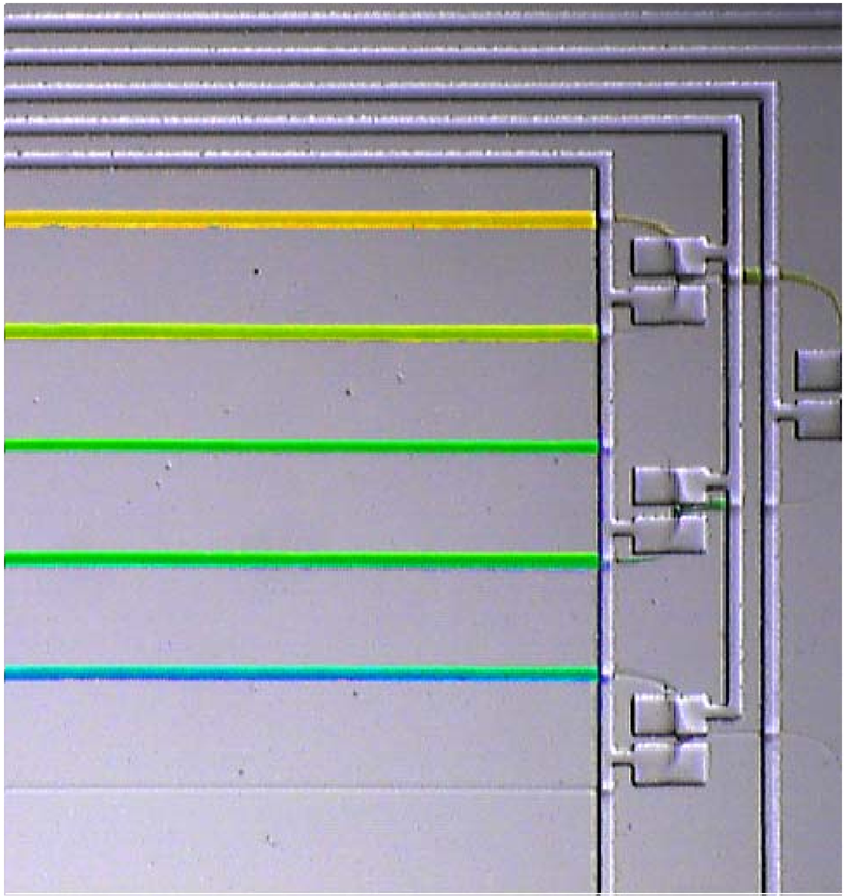
- **Manually map experiment to the valves of the device**
  - Using Labview or custom C interface
  - Given a new device, start over and do mapping again

Our Approach:
"Write Once, Run Anywhere"

- **Example: Gradient generation**

```
Fluid yellow = input (0);
Fluid blue = input(1);
for (int i=0; i<=4; i++) {
    mix(yellow, 1-i/4, blue, i/4);
}
```

- **Hidden from programmer:**
  - Location of fluids
  - Details of mixing, I/O
  - Logic of valve control
  - Timing of chip operations



450 Valve Operations

Our Approach:
"Write Once, Run Anywhere"

- **Example: Gradient generation**

```
Fluid yellow = input (0);
Fluid blue = input(1);
for (int i=0; i<=4; i++) {
    mix(yellow, 1-i/4, blue, i/4);
}
```

```
setValve(0, HIGH);   setValve(1, HIGH);
setValve(2, LOW);    setValve(3, HIGH);
setValve(4, LOW);    setValve(5, LOW);
setValve(6, HIGH);   setValve(7, LOW);
setValve(8, LOW);    setValve(9, HIGH);
setValve(10, LOW);   setValve(11, HIGH);
setValve(12, LOW);   setValve(13, HIGH);
setValve(14, LOW);   setValve(15, HIGH);
setValve(16, LOW);   setValve(17, LOW);
setValve(18, LOW);   setValve(19, LOW);
wait(2000);
setValve(14, HIGH); setValve(2, LOW);
wait(1000);
setValve(4, HIGH);   setValve(12, LOW);
setValve(16, HIGH); setValve(18, HIGH);
setValve(19, LOW);
wait(2000);
```

450 Valve Operations

- **Hidden from programmer:**
  - Location of fluids
  - Details of mixing, I/O
  - Logic of valve control
  - Timing of chip operations

Our Approach:
“Write Once, Run Anywhere”

- **Example: Gradient generation**

```
Fluid yellow = input (0);
Fluid blue = input(1);
for (int i=0; i<=4; i++) {
    mix(yellow, 1-i/4, blue, i/4);
}
```

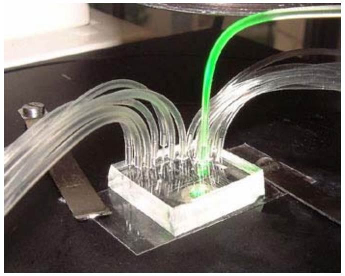
```
wait(2000);
setValve(14, HIGH); setValve(2, LOW);
wait(1000);
setValve(4, HIGH);   setValve(12, LOW);
setValve(16, HIGH); setValve(18, HIGH);
setValve(19, LOW);
wait(2000);
setValve(0, LOW);    setValve(1, LOW);
setValve(2, LOW);    setValve(3, HIGH);
setValve(4, LOW);    setValve(5, HIGH);
setValve(6, HIGH);   setValve(7, LOW);
setValve(8, LOW);    setValve(9, HIGH);
setValve(10, HIGH); setValve(11, LOW);
setValve(12, LOW);  setValve(13, LOW);
setValve(14, LOW);  setValve(15, HIGH);
setValve(16, HIGH); setValve(17, LOW);
setValve(18, HIGH); setValve(19, LOW);
```

- **Hidden from programmer:**
  - Location of fluids
  - Details of mixing, I/O
  - Logic of valve control
  - Timing of chip operations

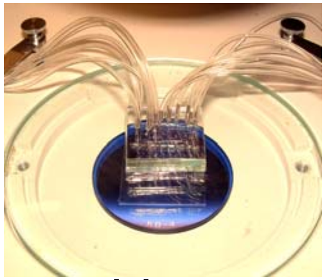
Fluidic Abstraction Layers

Protocol Description Language
- readable code with high-level mixing ops

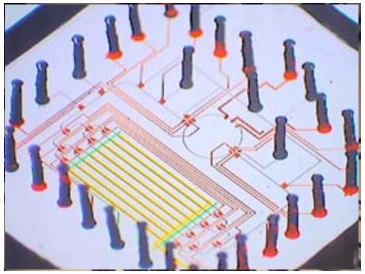
Fluidic Instruction Set Architecture (ISA)
- primitives for I/O, storage, transport, mixing



chip 1



chip 2



chip 3

Fluidic Hardware Primitives
- valves, multiplexers, mixers, latches

Silicon Analog

C

x86

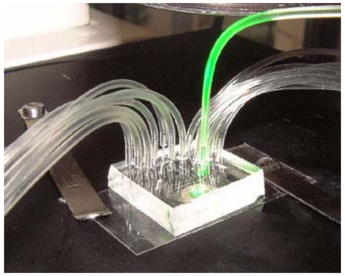
Pentium III, Pentium IV

transistors, registers, …

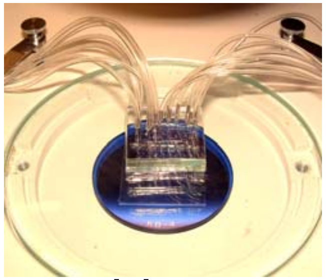
Fluidic Abstraction Layers

Protocol Description Language
- readable code with high-level mixing ops

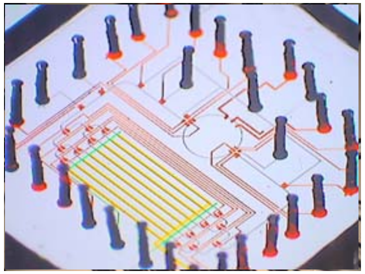
Fluidic Instruction Set Architecture (ISA)
- primitives for I/O, storage, transport, mixing



chip 1



chip 2



chip 3

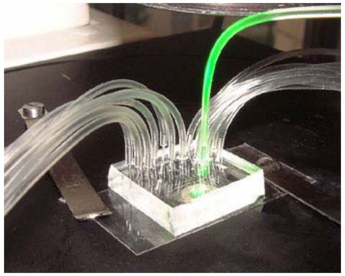
Fluidic Hardware Primitives
- valves, multiplexers, mixers, latches

- **Benefits:**
  - Division of labor
  - Portability
  - Scalability
  - Expressivity

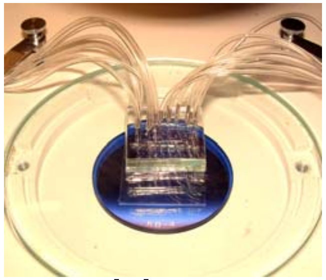
Fluidic Abstraction Layers

Protocol Description Language
- readable code with high-level mixing ops

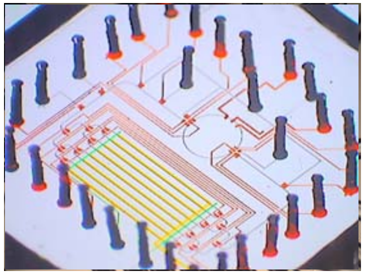
Fluidic Instruction Set Architecture (ISA)
- primitives for I/O, storage, transport, mixing



chip 1



chip 2

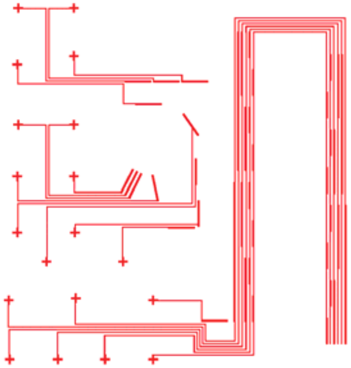


chip 3

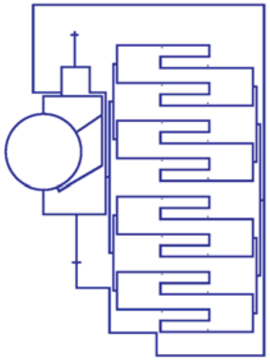
Fluidic Hardware Primitives
- valves, multiplexers, mixers, latches

- **Benefits:**
  - Division of labor
  - Portability
  - Scalability
  - Expressivity

Primitive 1: A Valve (Quake et al.)

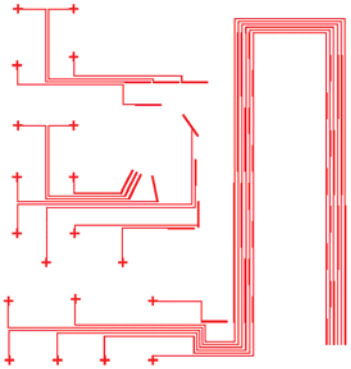


Control
Layer



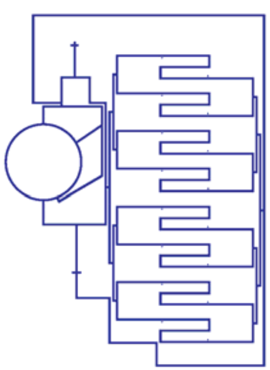
Flow
Layer

Primitive 1: A Valve (Quake et al.)



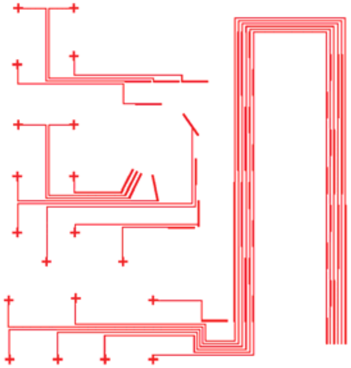
Control Layer

0. Start with mask of channels



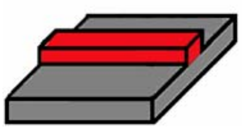
Flow Layer

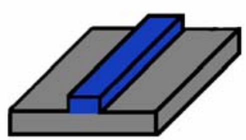
Primitive 1: A Valve (Quake et al.)

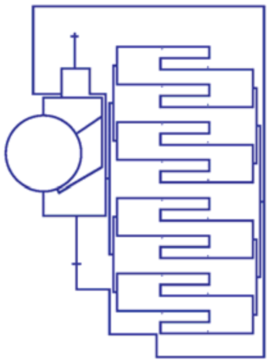


Control Layer

1. Deposit pattern on silicon wafer

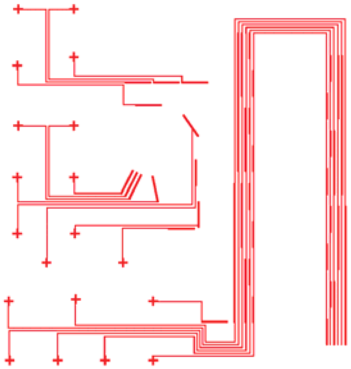






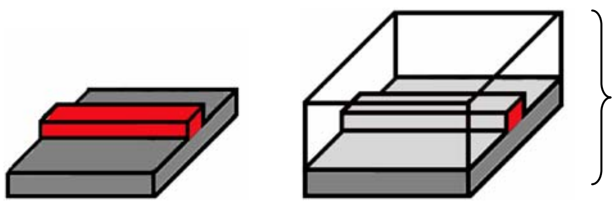
Flow Layer

Primitive 1: A Valve (Quake et al.)

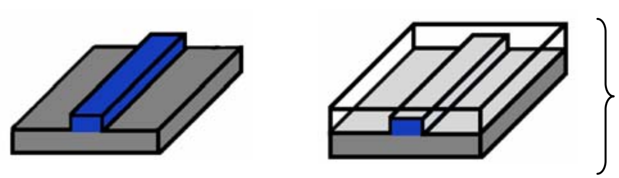


Control
Layer

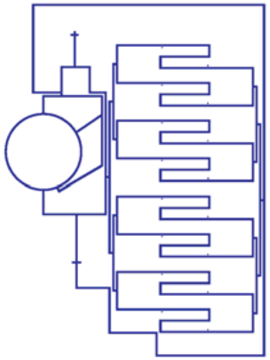
2. Pour PDMS over mold
  - polydimexylsiloxane: “soft lithography”



Thick layer (poured)

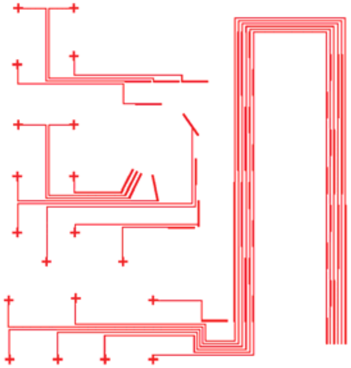


Thin layer (spin-coated)



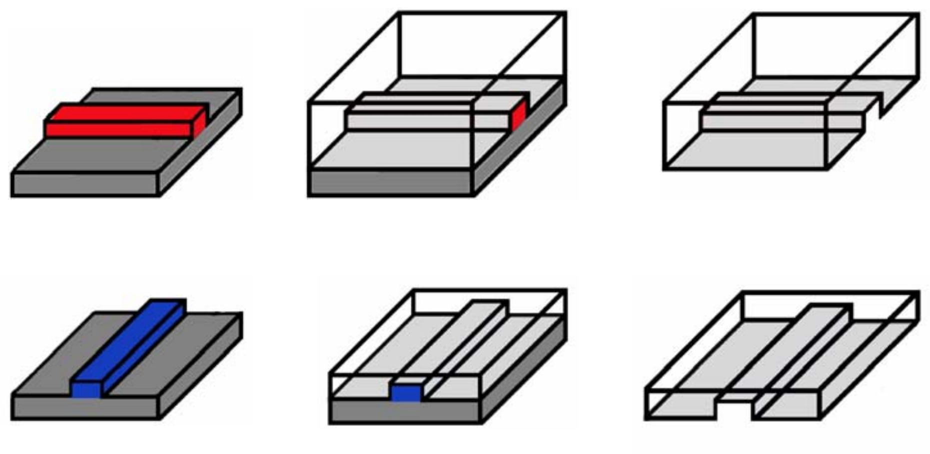
Flow
Layer

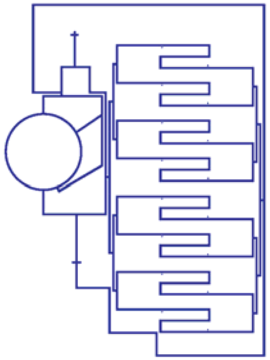
Primitive 1: A Valve (Quake et al.)



Control Layer

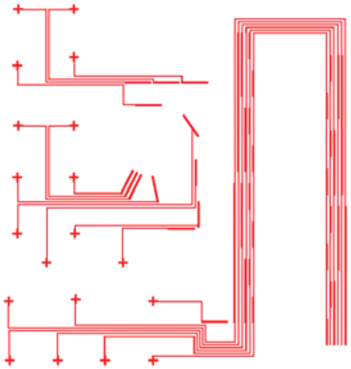
3. Bake at 80° C (primary cure), then release PDMS from mold





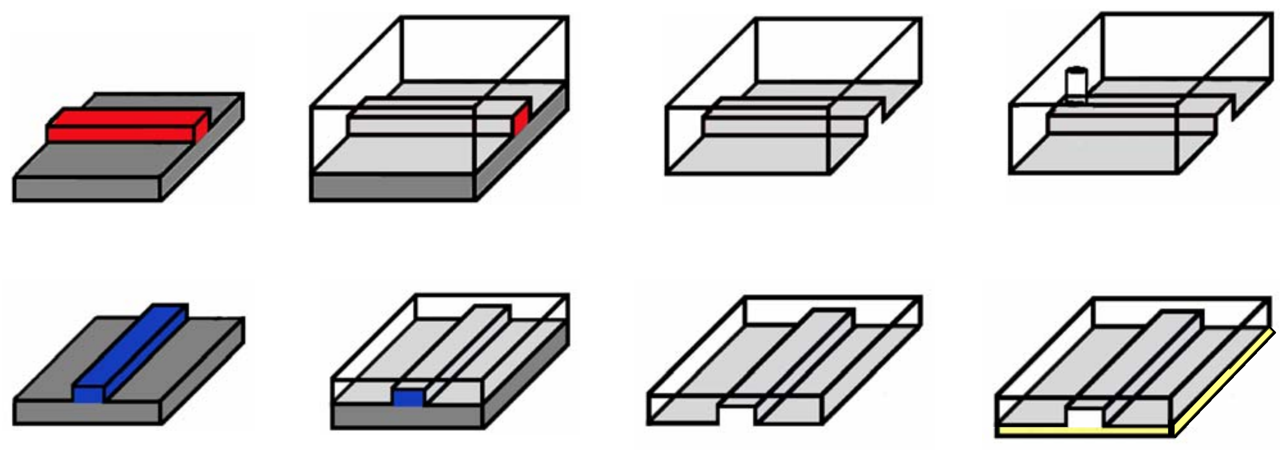
Flow Layer

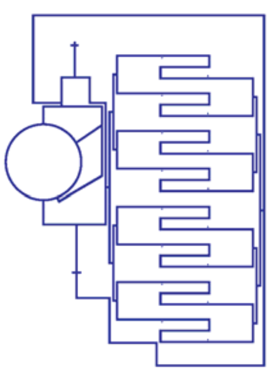
Primitive 1: A Valve (Quake et al.)



Control Layer

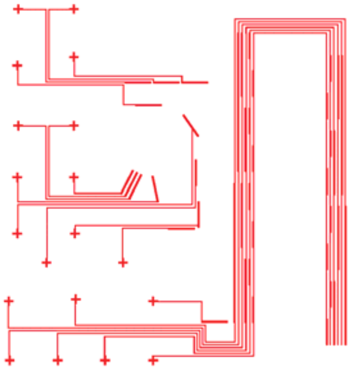
4a. Punch hole in control channel
4b. Attach flow layer to glass slide





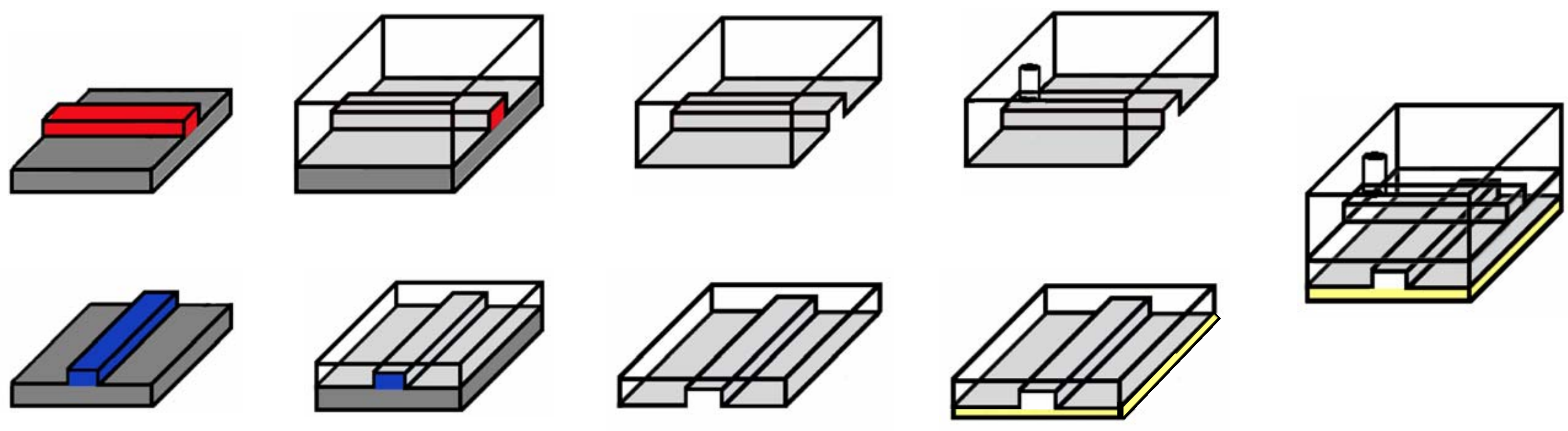
Flow Layer

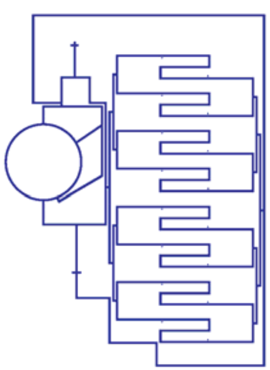
Primitive 1: A Valve (Quake et al.)



Control Layer

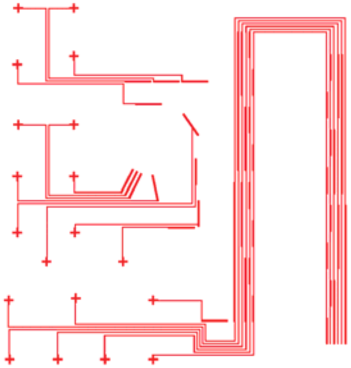
5. Align flow layer over control layer





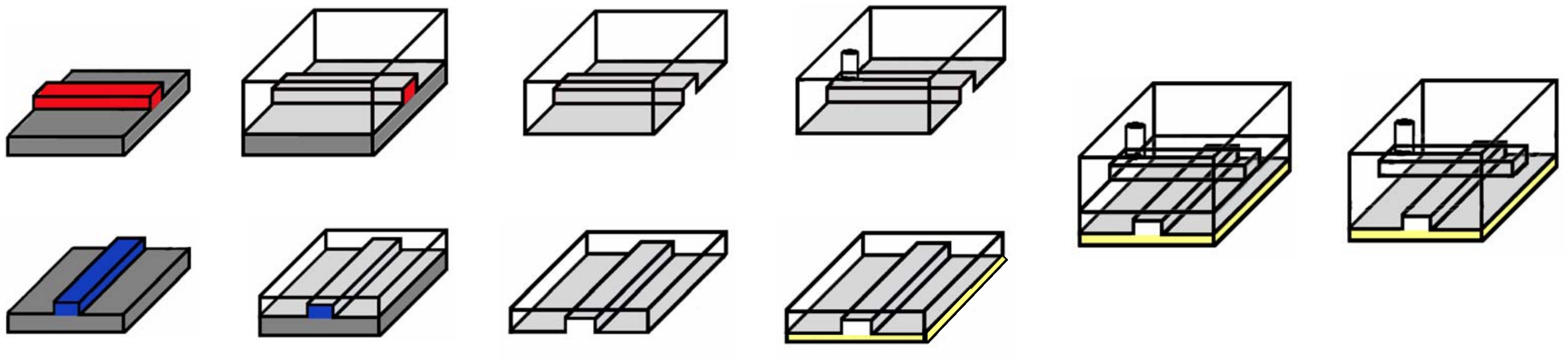
Flow Layer

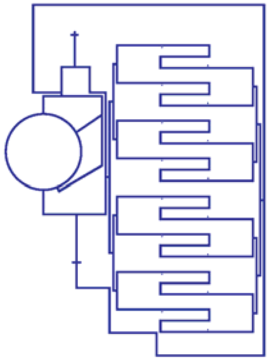
Primitive 1: A Valve (Quake et al.)



Control
Layer

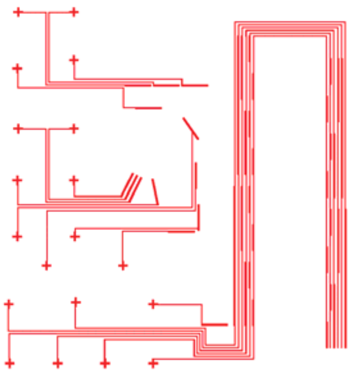
6. Bake at 80° C (secondary cure)





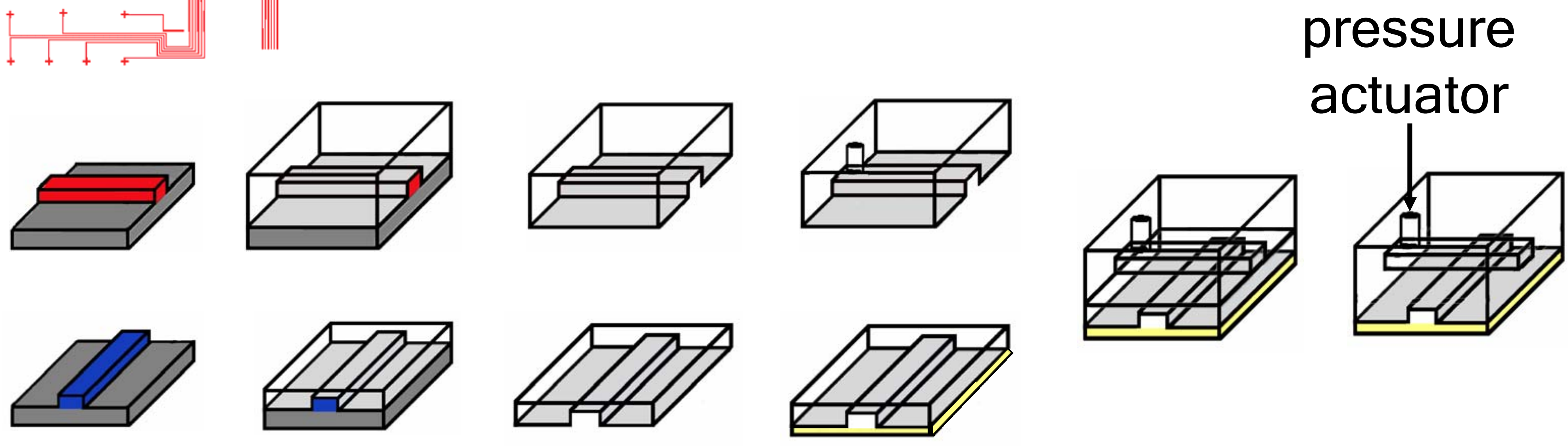
Flow
Layer

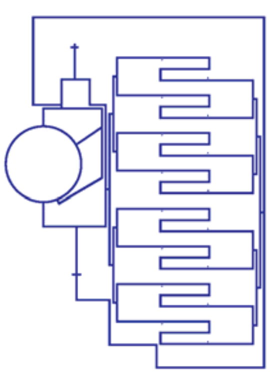
Primitive 1: A Valve (Quake et al.)



Control Layer

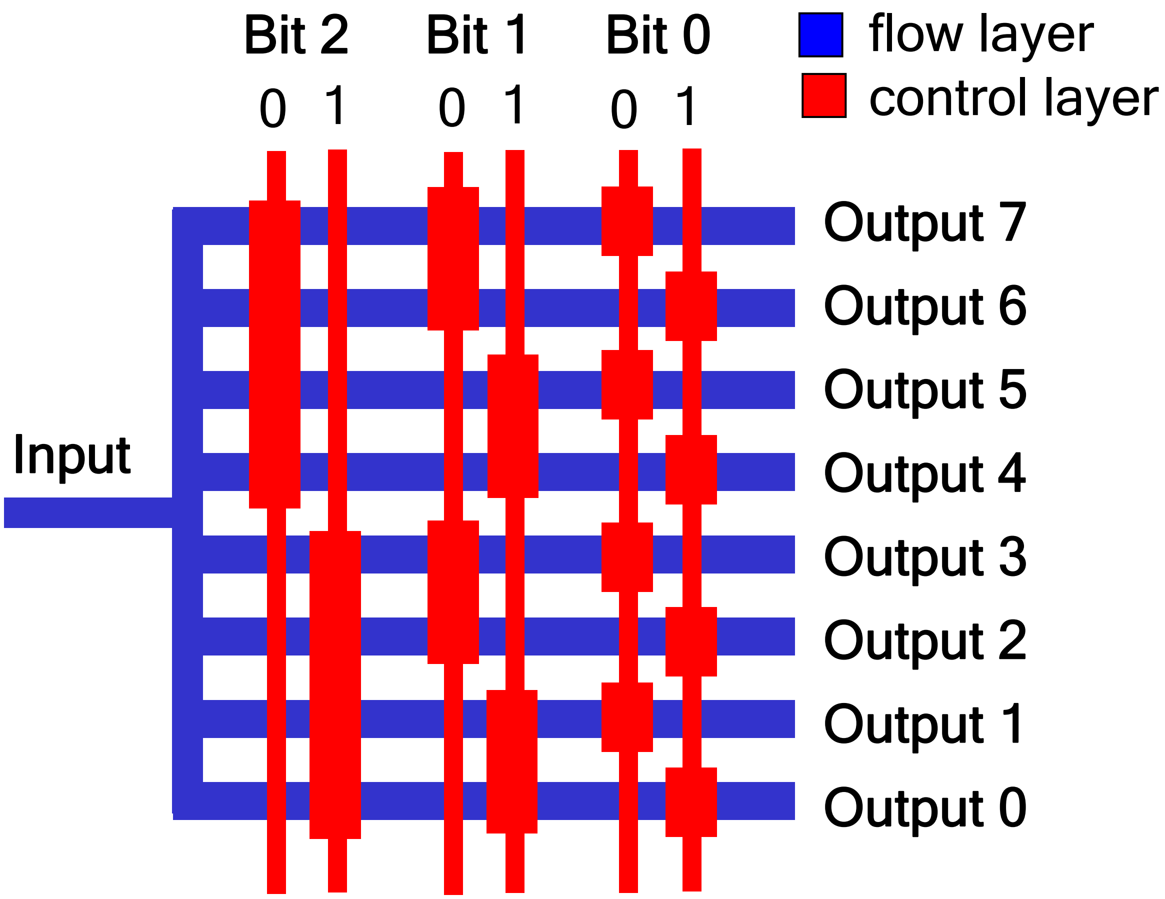
7. When pressure is high, control channel pinches flow channel to form a valve



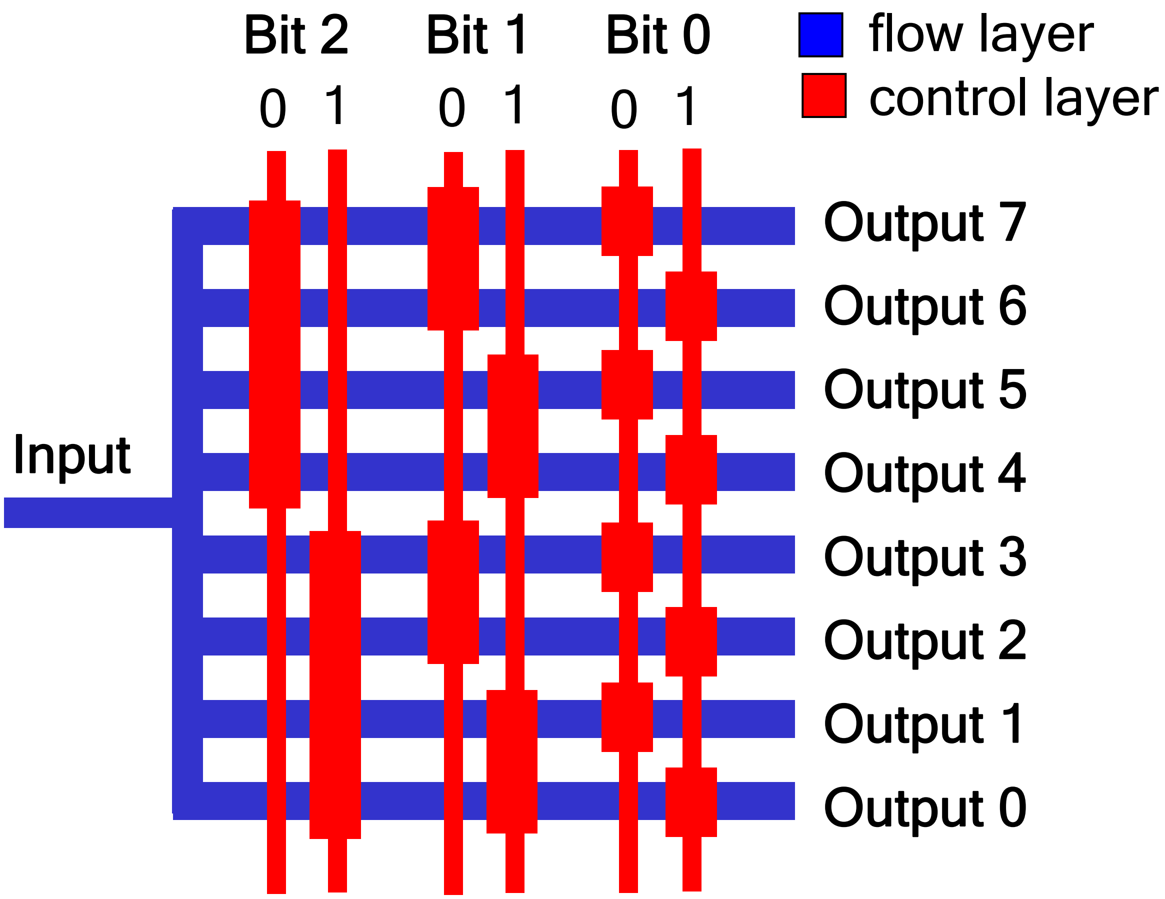


Flow Layer

Primitive 2: A Multiplexer (Thorsen et al.)

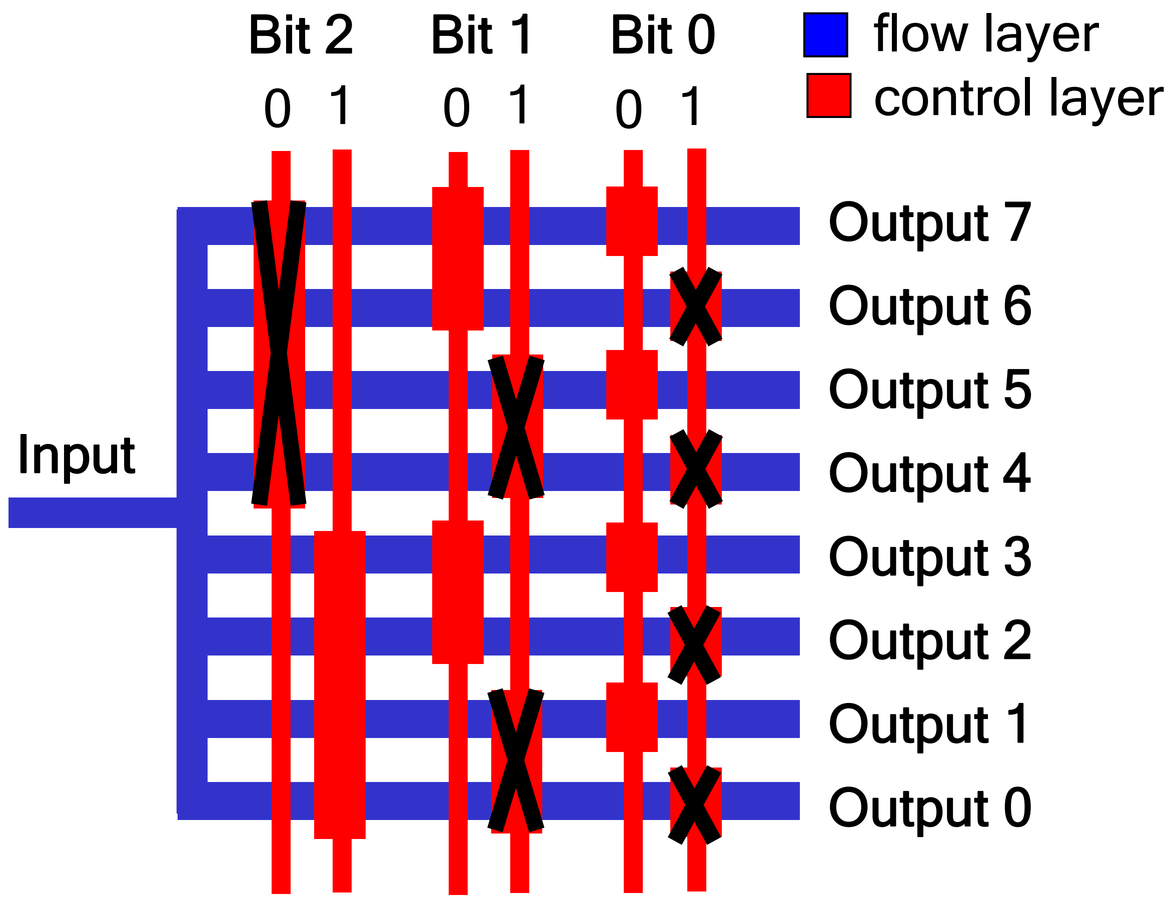


Primitive 2: A Multiplexer (Thorsen et al.)



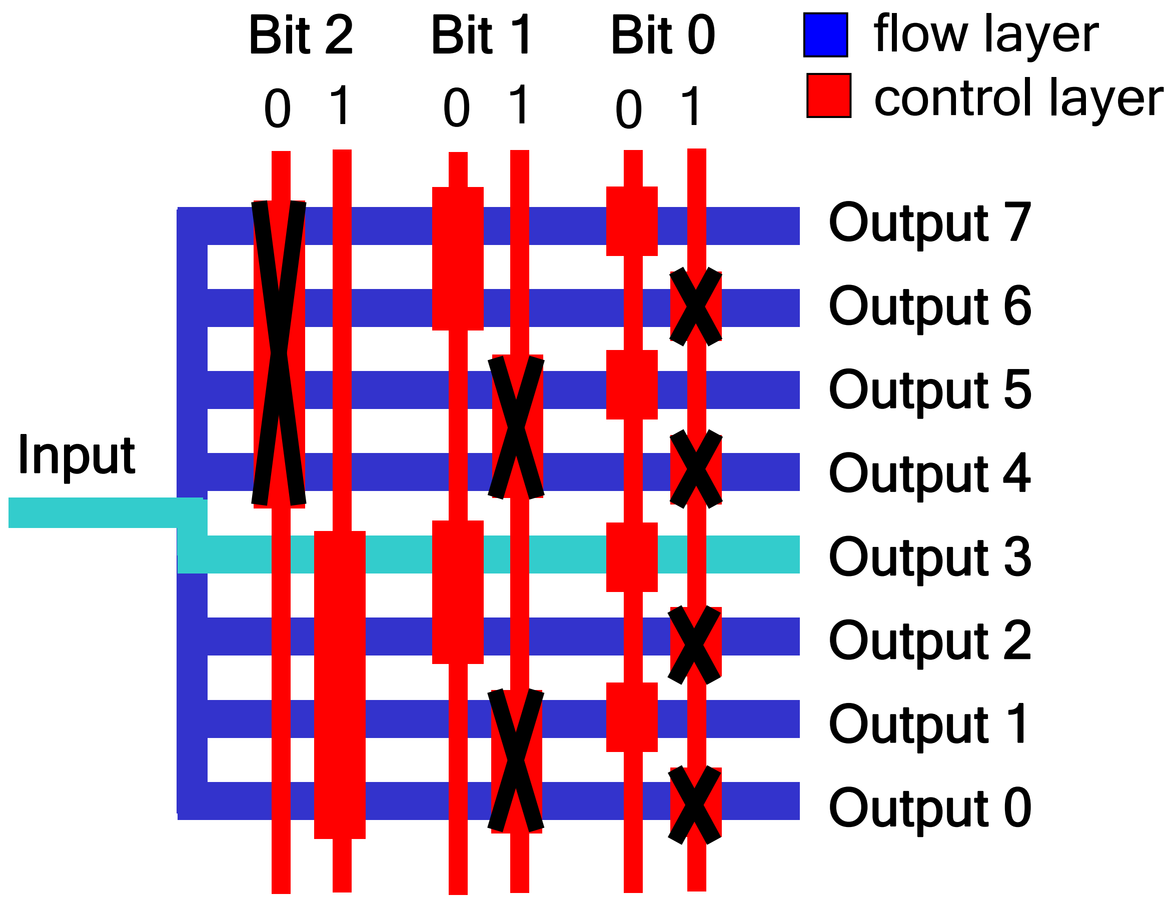
Example: select 3 = 011

Primitive 2: A Multiplexer (Thorsen et al.)



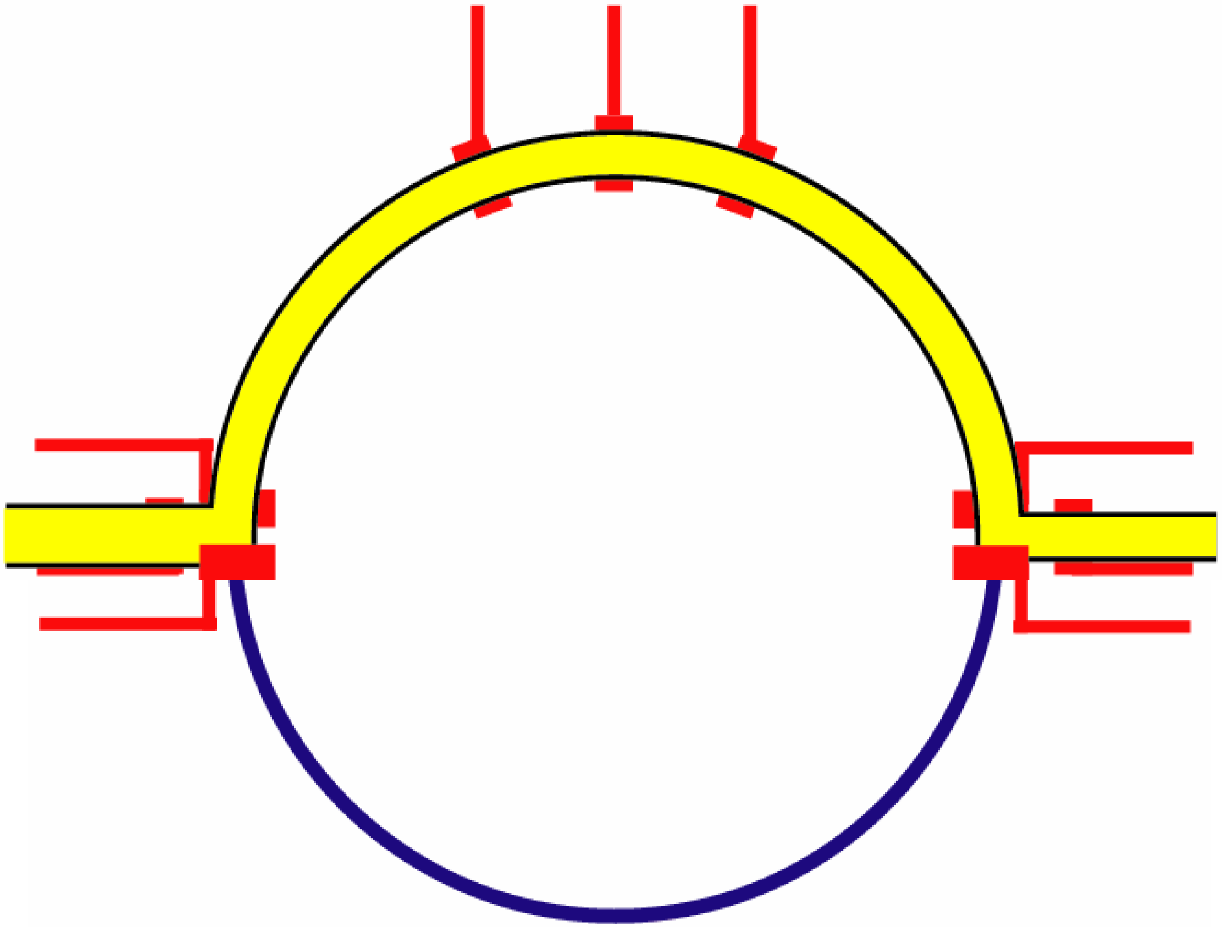
Example: select 3 = 011

Primitive 2: A Multiplexer (Thorsen et al.)



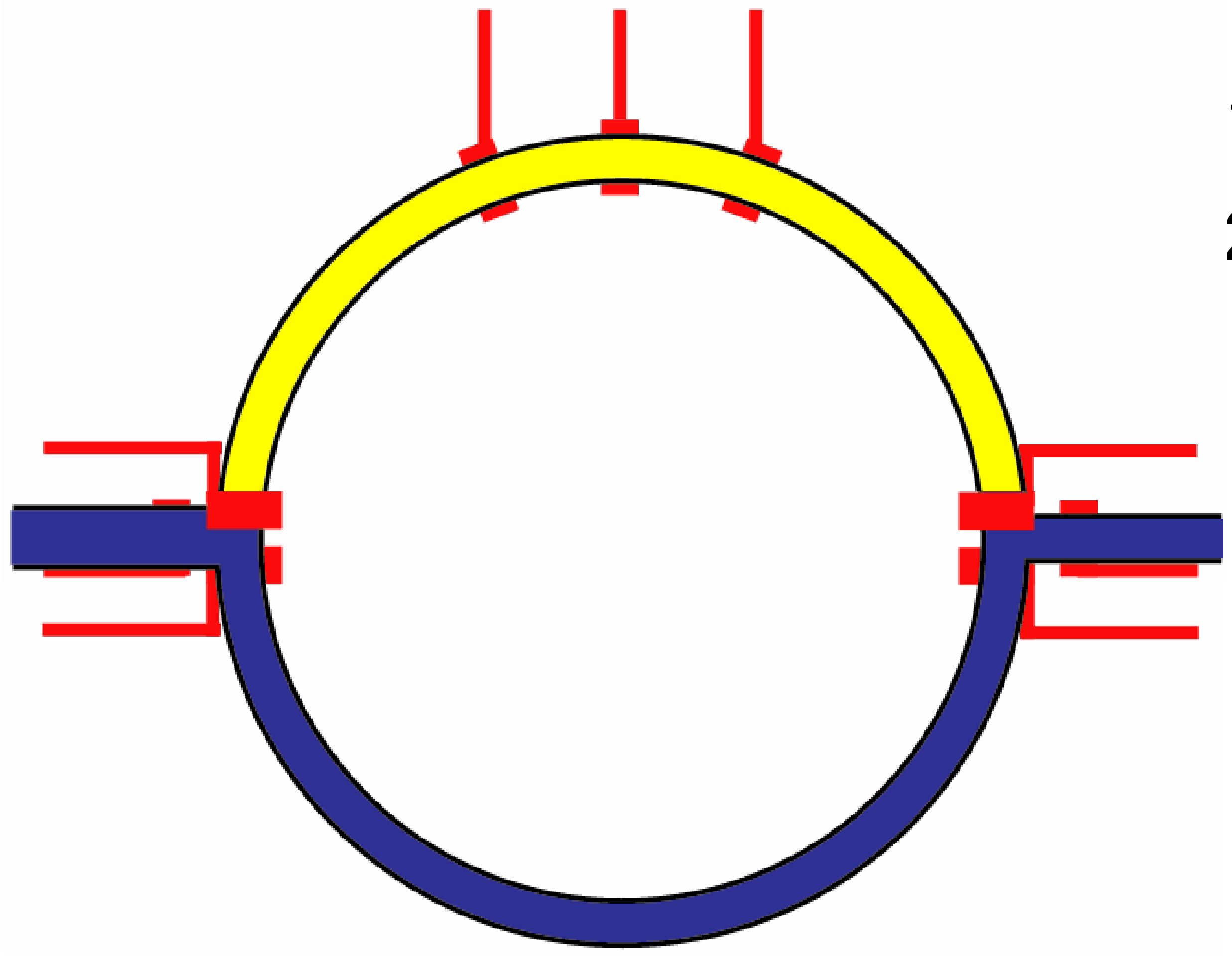
Example: select 3 = 011

Primitive 3: A Mixer (Quake et al.)



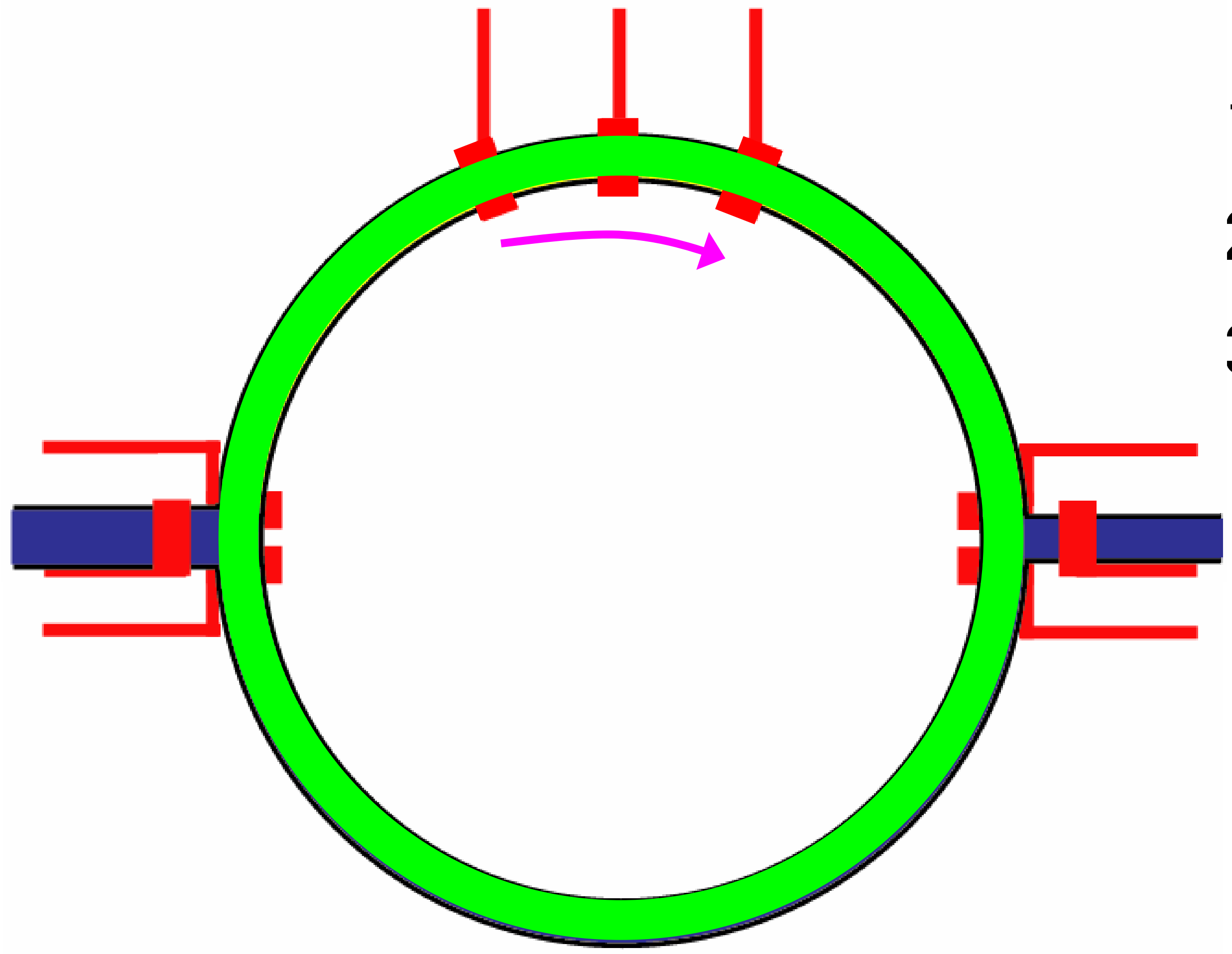
1. Load sample on top

Primitive 3: A Mixer (Quake et al.)



1. Load sample on top
2. Load sample on bottom

Primitive 3: A Mixer (Quake et al.)

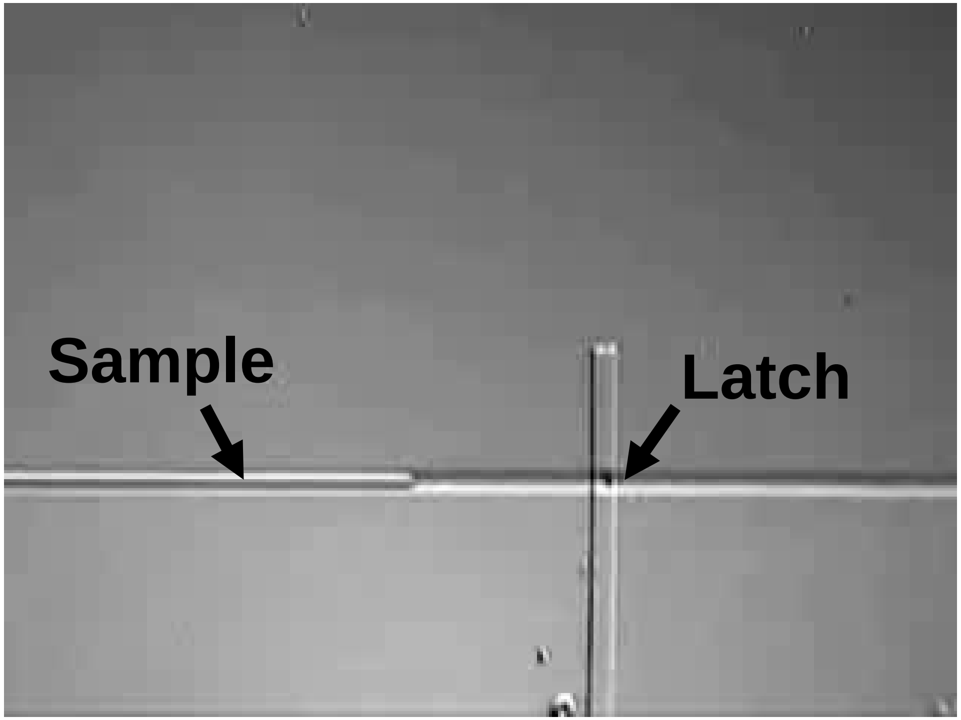


1. Load sample on top
2. Load sample on bottom
3. Peristaltic pumping

Rotary Mixing

Primitive 4: A Latch (Our contribution)

- **Purpose: align sample with specific location on device**
  - Examples: end of storage cell, end of mixer, middle of sensor

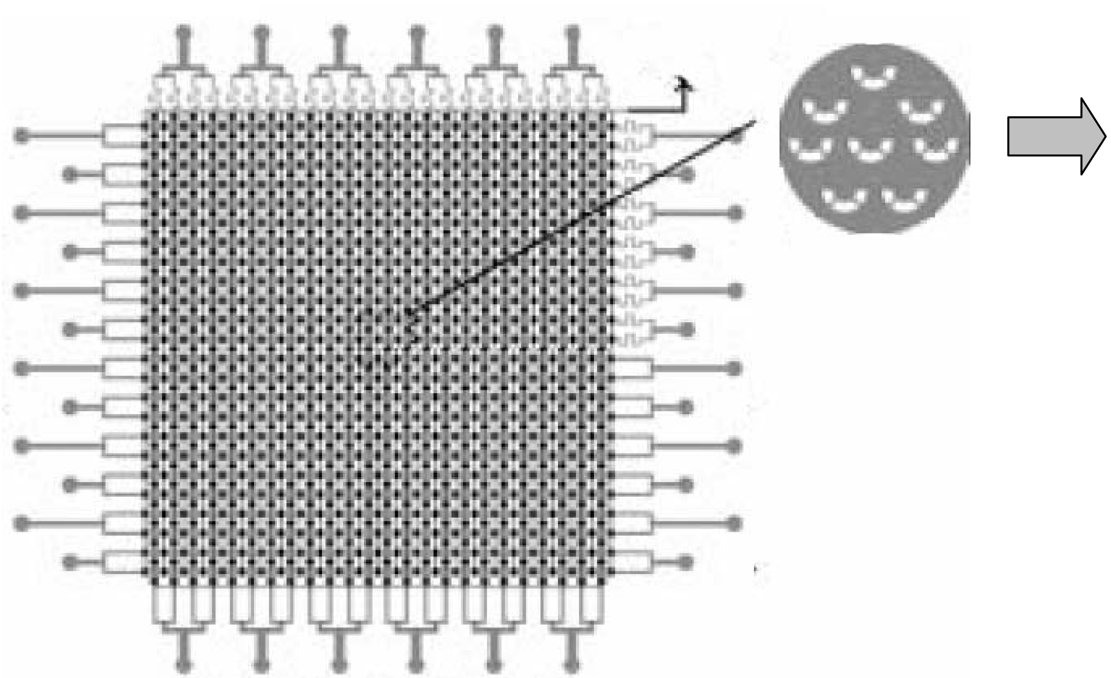


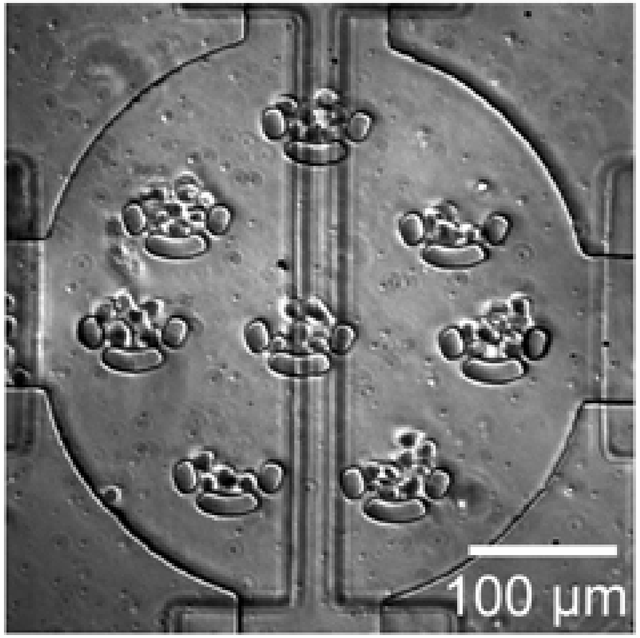
- **Latches are implemented as a partially closed valve**
  - Background flow passes freely
  - Aqueous samples are caught

Primitive 5: Cell Trap

- **Several methods for confining cells in microfluidic chips**
  - U-shaped weirs
  - C-shaped rings / microseives
  - Holographic optical traps
  - Dialectrophoresis

- **In our chips: U-Shaped Microseives in PDMS Chambers**

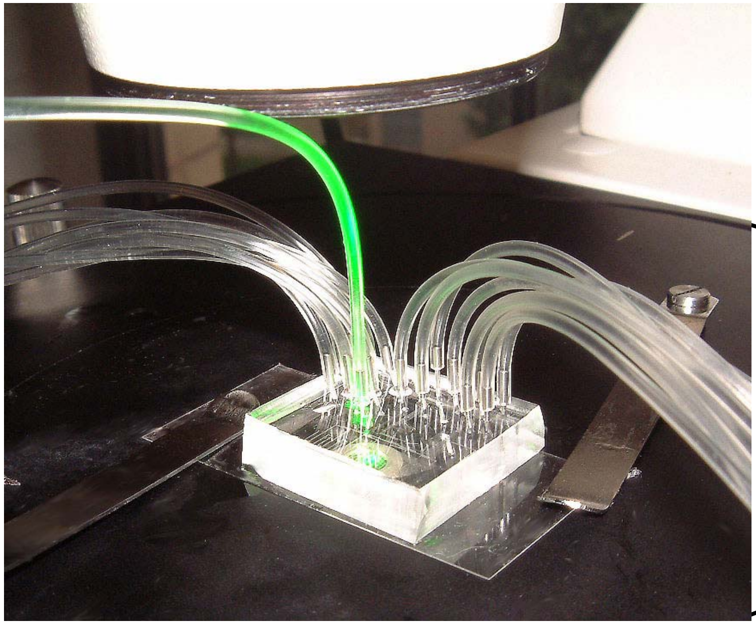




Source: *Wang, Kim, Marquez, and Thorsen, Lab on a Chip 2007*

Primitive 6: Imaging and Detection

- **As PDMS chips are translucent, contents can be imaged directly**
  - Fluorescence, color, opacity, etc.

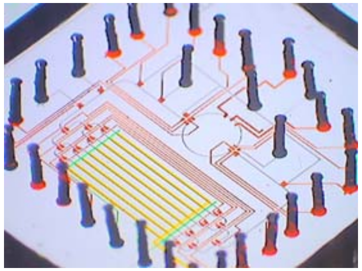


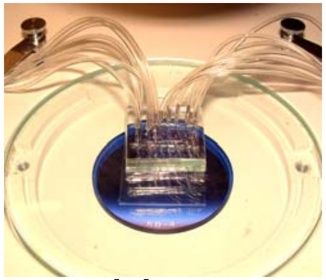
- **Feedback can be used to drive the experiment**

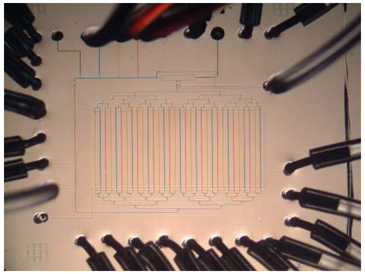
Fluidic Abstraction Layers

Protocol Description Language
- readable code with high-level mixing ops

Fluidic Instruction Set Architecture (ISA)
- primitives for I/O, storage, transport, mixing

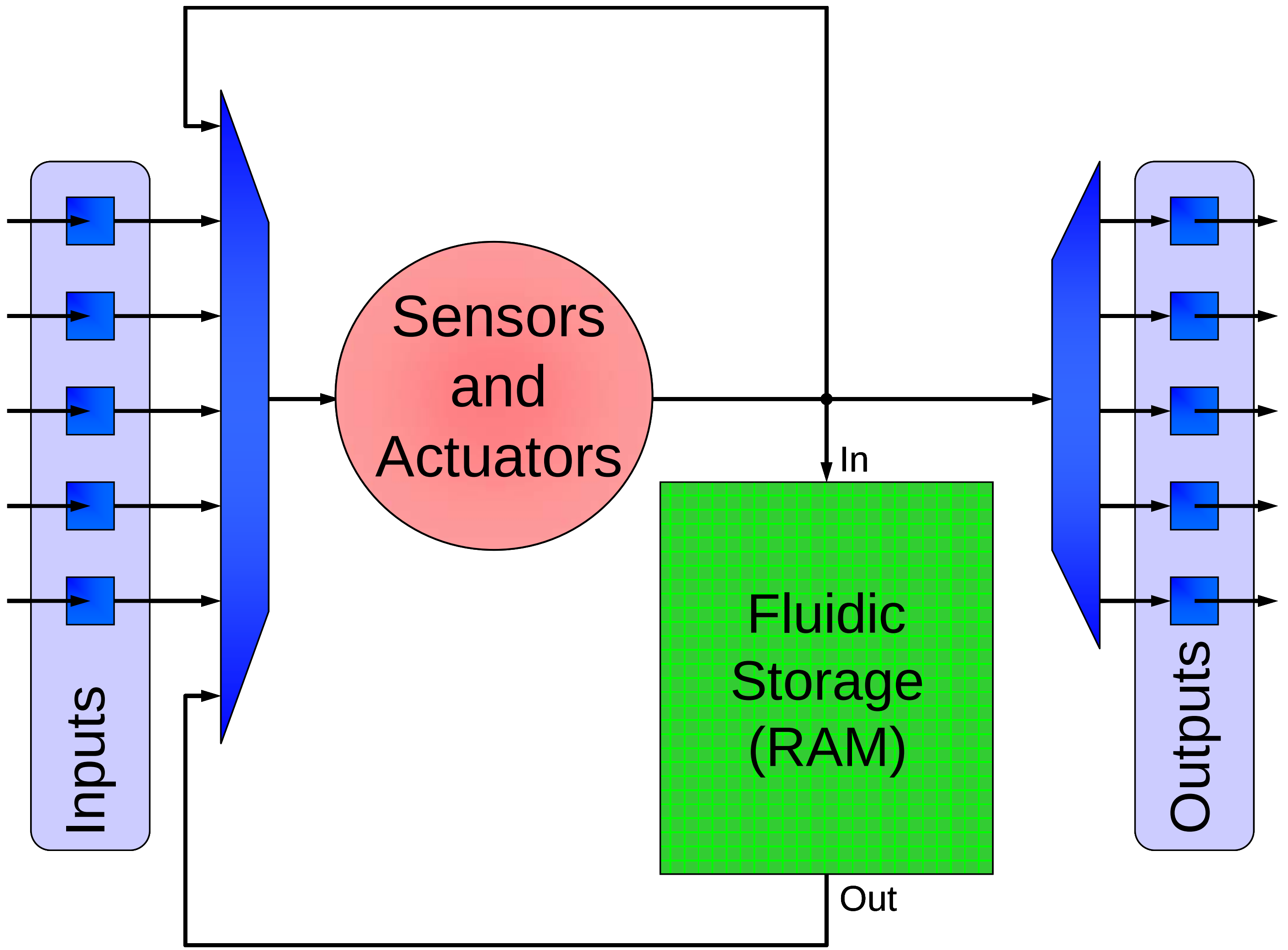

chip 1


chip 2


chip 3

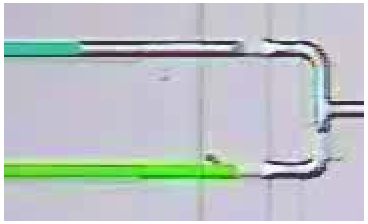
Fluidic Hardware Primitives
- valves, multiplexers, mixers, latches

Toward "General Purpose" Microfluidic Chips

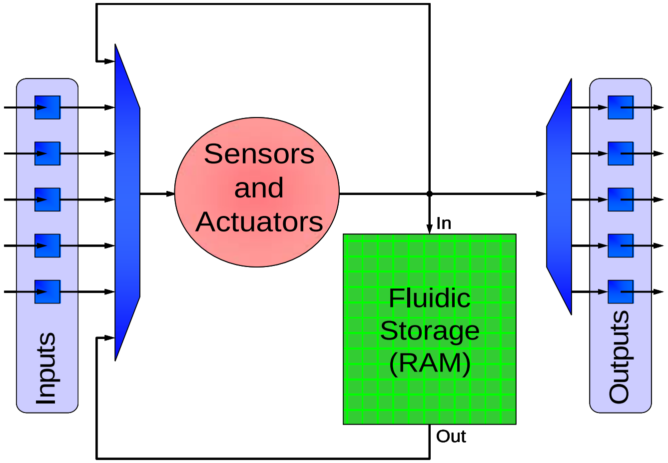


Abstraction 1: Digital Architecture

- **Recent techniques can control independent samples**
  - Droplet-based samples [Fair et al.]
  - Continuous-flow samples [Our contribution]
  - Microfluidic latches [Our contribution]



- **In abstract machine, all samples have unit volume**
  - Input/output a sample
  - Store a sample
  - Operate on a sample



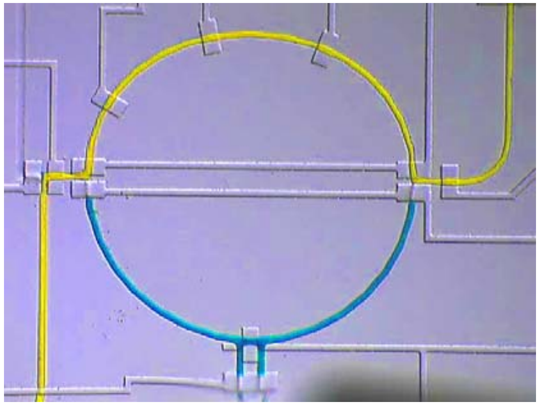
- **Challenge for a digital architecture: fluid loss**
  - No chip is perfect – will lose some volume over time
  - Causes: imprecise valves, adhesion to channels, evaporation, ...
  - How to maintain digital abstraction?

Maintaining a Digital Abstraction

Electronics		**Microfluidics**
Soft error Handling?	**High-Level Language**	Expose loss in language - User deals with it
? ? ?		
Randomized Gates [Palem]	**Instruction Set Architecture (ISA)**	Expose loss in ISA - Compiler deals with it
? ? ?		
Replenish charge (GAIN)	**Hardware**	Replenish fluids? - Maybe (e.g., with water) - But may affect chemistry
Loss of charge		Loss of fluids

Abstraction 2: Mix Instruction

- **Microfluidic chips have various mixing technologies**
  - Electrokinetic mixing [Levitan et al.]
  - Droplet mixing [Fair et al.]
  - Rotary mixing [Quake et al.]



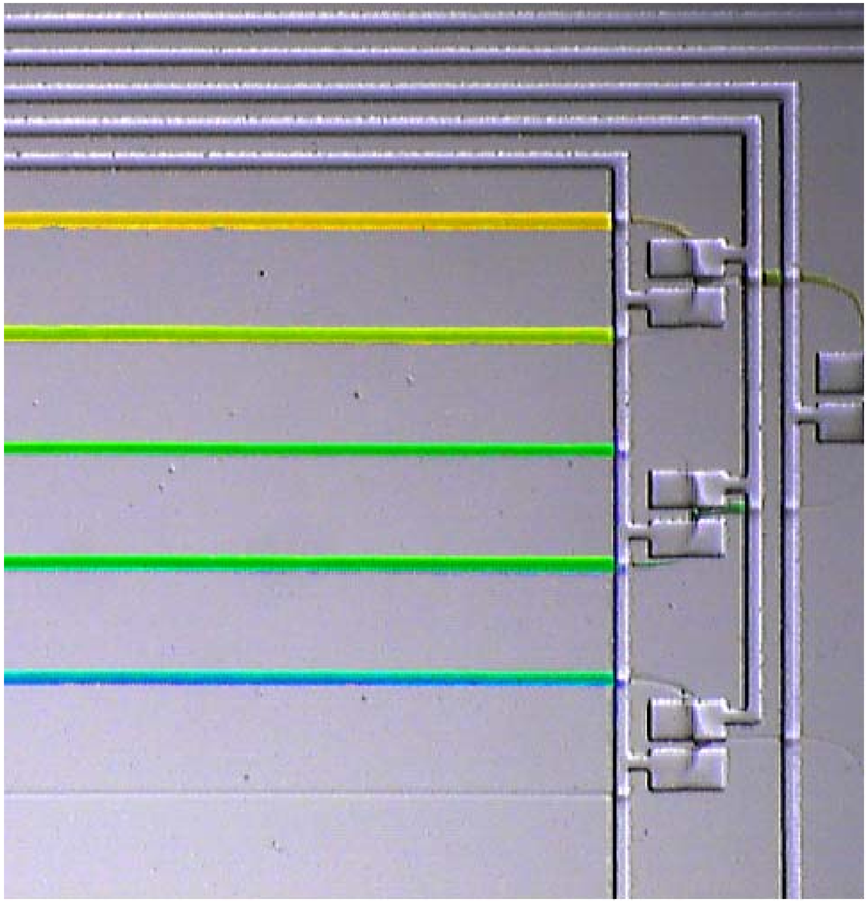
- **Common attributes:**
  - Ability to mix two samples in equal proportions, store result
- **Fluidic ISA: mix** (int src_1, int src_2, int dst)
  - Ex: mix(1, 2, 3)

Storage Cells	
1	
2	
3	
4	

Mixer

  - To allow for lossy transport, only 1 unit of mixture retained

Gradient Generation in Fluidic ISA



Gradient Generation in Fluidic ISA

```
wait(2000);
setValve(14, HIGH); setValve(2, LOW);
wait(1000);
setValve(4, HIGH);   setValve(12, LOW);
setValve(16, HIGH); setValve(18, HIGH);
setValve(19, LOW);
wait(2000);
setValve(0, LOW);    setValve(1, LOW);
setValve(2, LOW);    setValve(3, HIGH);
setValve(4, LOW);    setValve(5, HIGH);
setValve(6, HIGH);   setValve(7, LOW);
setValve(8, LOW);    setValve(9, HIGH);
setValve(10, HIGH); setValve(11, LOW);
setValve(12, LOW);  setValve(13, LOW);
setValve(14, LOW);  setValve(15, HIGH);
setValve(16, HIGH); setValve(17, LOW);
setValve(18, HIGH); setValve(19, LOW);
```

abstraction

```
input(0, 0);
input(1, 1);
input(0, 2);
mix(1, 2, 3);
input(0, 2);
mix(2, 3, 1);
input(1, 3);
input(0, 4);
mix(3, 4, 2);
input(1, 3);
input(0, 4);
mix(3, 4, 5);
input(1, 4);
mix(4, 5, 3);
mix(0, 4);
```

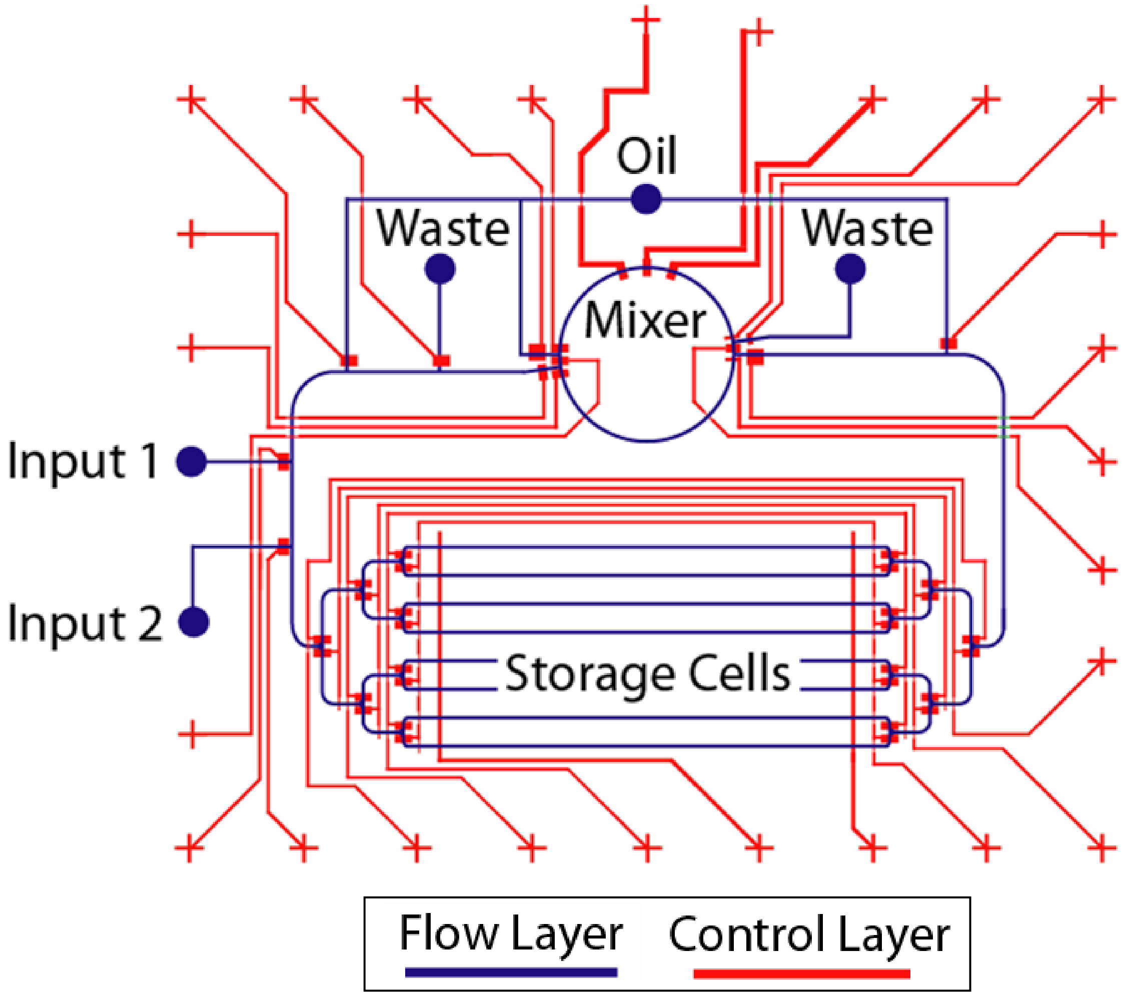
Direct Control

- 450 valve actuations
- only works on 1 chip

Fluidic ISA

- 15 instructions
- portable across chips

Implementation: Oil-Driven Chip



	Inputs	Storage Cells	Background Phase	Wash Phase	Mixing
Chip 1	2	8	Oil	—	Rotary

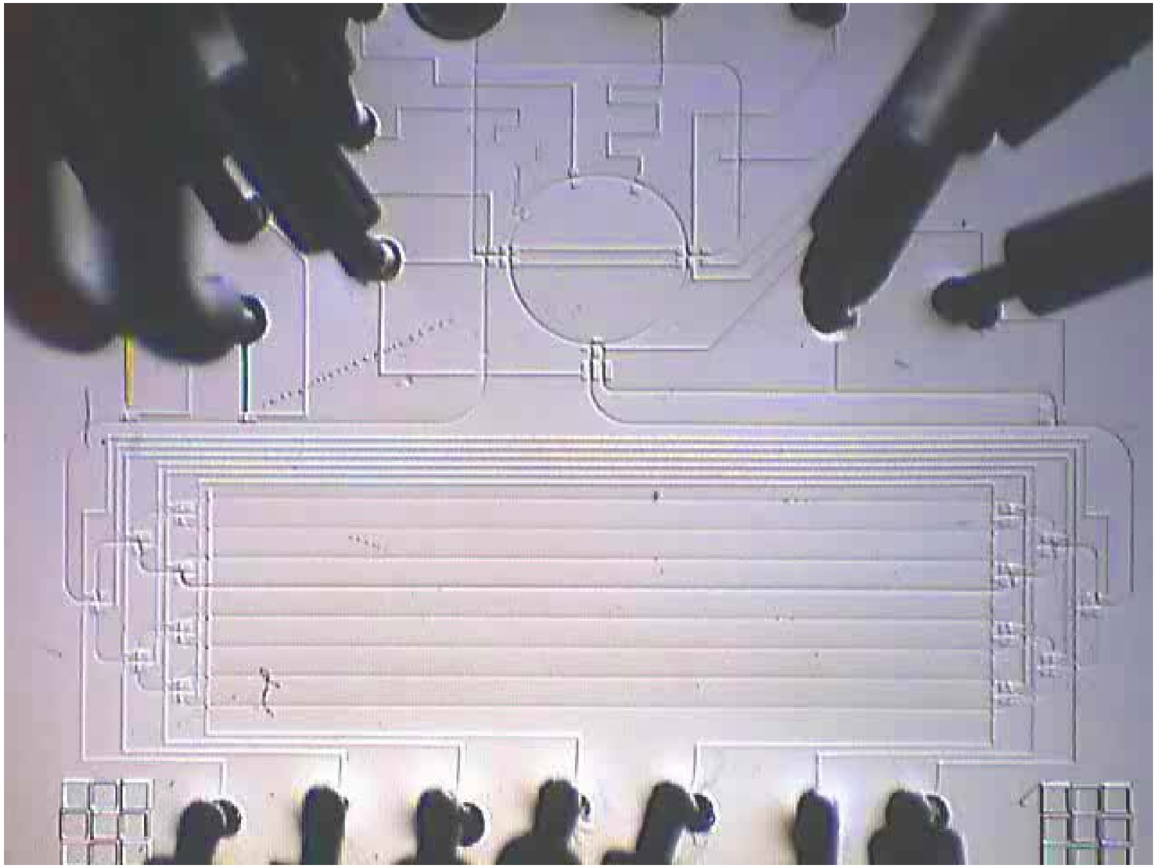
Implementation: Oil-Driven Chip

mix (S_1, S_2, D) {

1. Load S_1
2. Load S_2
3. Rotary mixing
4. Store into D

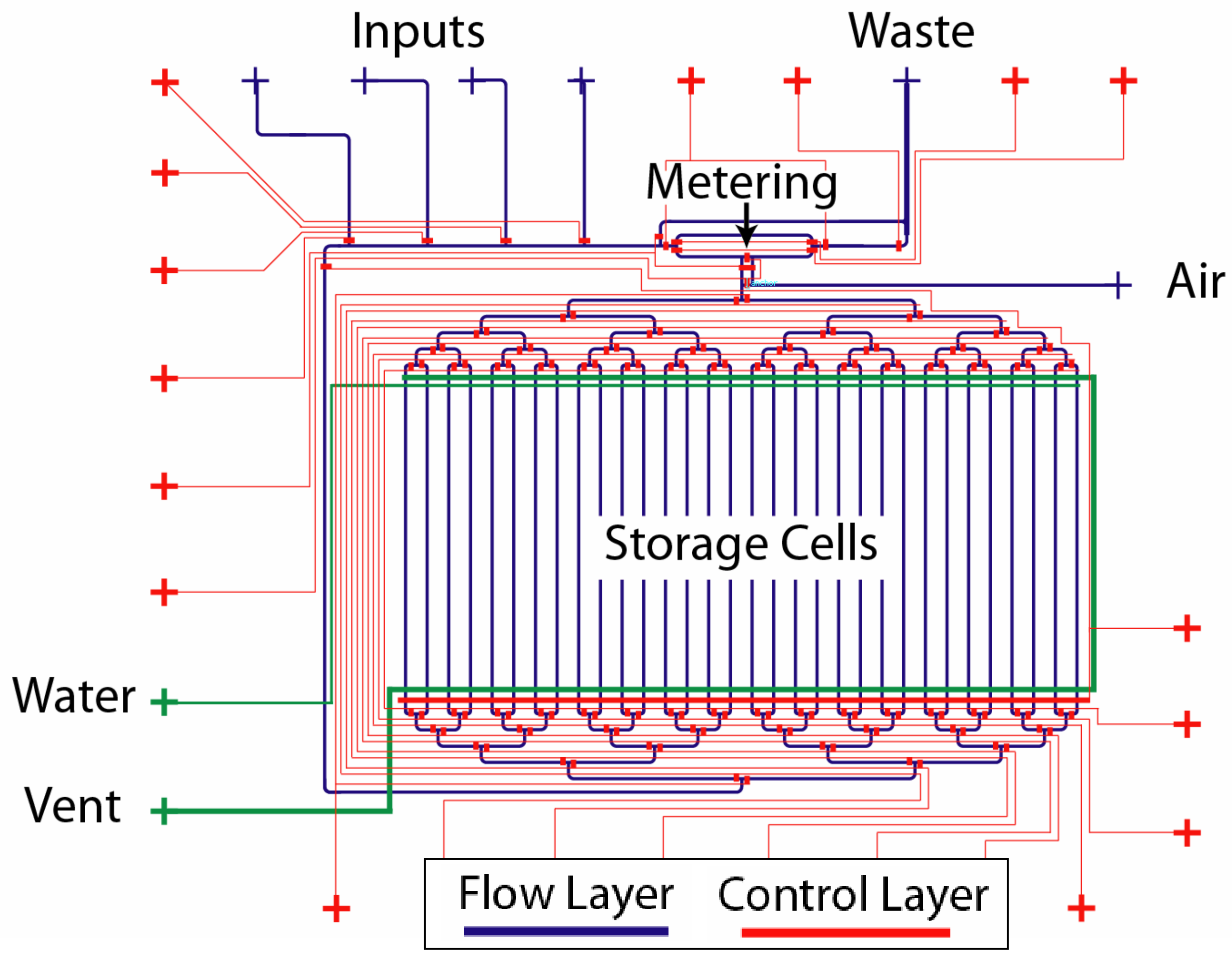
}

50x real-time



	Inputs	Storage Cells	Background Phase	Wash Phase	Mixing
Chip 1	2	8	Oil	—	Rotary

Implementation 2: Air-Driven Chip

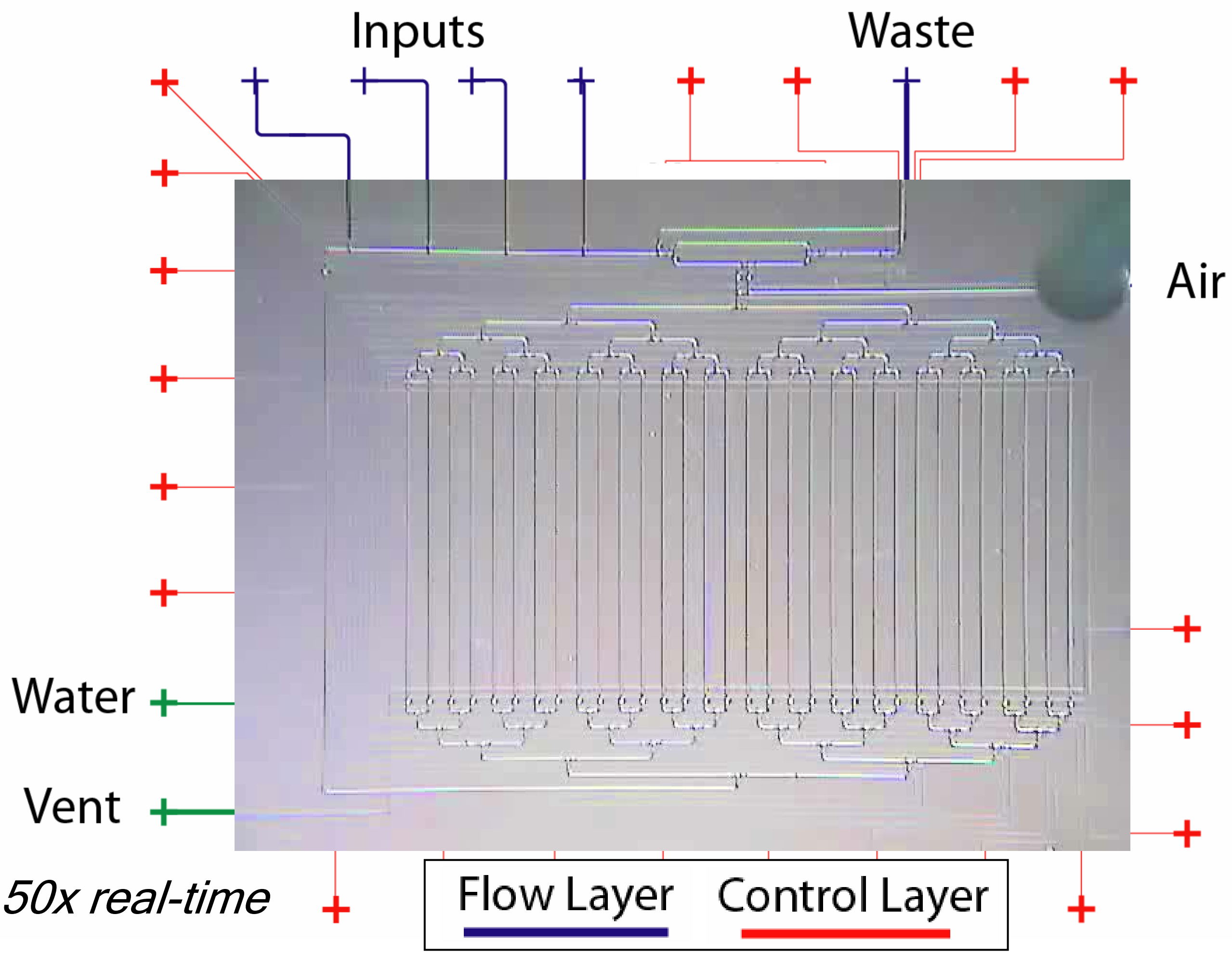


	Inputs	Storage Cells	Background Phase	Wash Phase	Mixing
Chip 1	2	8	Oil	—	Rotary
Chip 2	4	32	Air	Water	In channels

Implementation 2: Air-Driven Chip

mix (S_1, S_2, D) {
1. Load S_1
2. Load S_2
3. Mix / Store into D
4. Wash S_1
5. Wash S_2

}



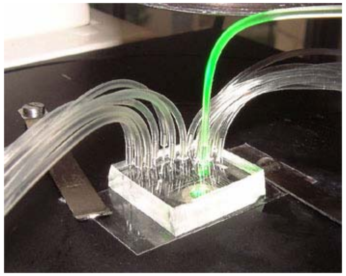
50x real-time

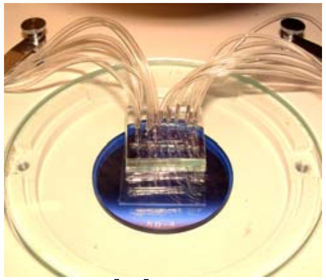
	Inputs	Storage Cells	Background Phase	Wash Phase	Mixing
Chip 1	2	8	Oil	—	Rotary
Chip 2	4	32	Air	Water	In channels

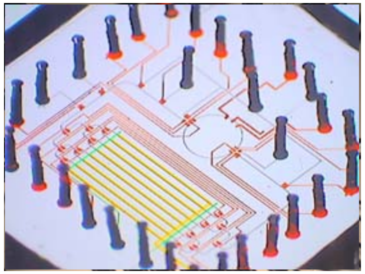
Fluidic Abstraction Layers

Protocol Description Language
- readable code with high-level mixing ops

Fluidic Instruction Set Architecture (ISA)
- primitives for I/O, storage, transport, mixing


chip 1


chip 2


chip 3

Fluidic Hardware Primitives
- valves, multiplexers, mixers, latches

Abstraction 1: Managing Fluid Storage

Fluidic ISA

```
input(0, 0);
input(1, 1);
input(0, 2);
mix(1, 2, 3);
input(0, 2);
mix(2, 3, 1);
input(1, 3);
input(0, 4);
mix(3, 4, 2);
input(1, 3);
input(0, 4);
mix(3, 4, 5);
input(1, 4);
mix(4, 5, 3);
mix(0, 4);
```

1. Storage Management

```
Fluid[] out = new Fluid[8];
Fluid yellow, blue, green;
out[0] = input(0);
yellow = input(0);
blue = input(1);
green = mix(yellow, blue);
yellow = input(0);
out[1] = mix(yellow, green);
yellow = input(0);
blue = input(1);
out[2] = mix(yellow, blue);
yellow = input(0);
blue = input(1);
green = mix(yellow, blue);
blue = input(1);
out[3] = mix(blue, green);
out[4] = input(1);
```

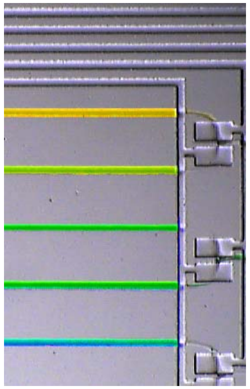
- **Programmer uses location-independent Fluid variables**
  - Runtime system assigns & tracks location of each Fluid
  - Comparable to automatic memory management (e.g., Java)

Abstraction 2: Fluid Re-Generation

```
Fluid[] out = new Fluid[8];
Fluid yellow, blue, green;
out[0] = input(0);
yellow = input(0);
blue = input(1);
green = mix(yellow, blue);
yellow = input(0);
out[1] = mix(yellow, green);
yellow = input(0);
blue = input(1);
out[2] = mix(yellow, blue);
yellow = input(0);
blue = input(1);
green = mix(yellow, blue);
blue = input(1);
out[3] = mix(blue, green);
out[4] = input(1);
```

```
Fluid[] out = new Fluid[8];
Fluid yellow = input(0);
Fluid blue = input(1);
Fluid green = mix(yellow, blue);

out[0] = yellow;
out[1] = mix(yellow, green);
out[2] = green;
out[3] = mix(blue, green);
out[4] = blue;
```

2. Fluid Re-Generation



- **Programmer may use a Fluid variable multiple times**
  - Each time, a physical Fluid is consumed on-chip
  - Runtime system re-generates Fluids from computation history

Custom Re-Generation

- **Some species cannot be regenerated by repeating history**
  - e.g., if selective mutagenesis has evolved unique sequence
- **Users can extend Fluid class, specify how to regenerate**
  - e.g., run PCR to amplify sequence of interest

```
class DNASample extends Fluid {

    // Return array of fluids that are equivalent to this fluid
    Fluid[] regenerate() {
        Fluid amplified = performPCR(this, cycles, primer1, primer2, …);
        Fluid[] diluted = dilute(amplified, Math.pow(2, cycles));
        return diluted;
    }

    // Return minimum quantity of this fluid needed to generate others
    int minQuantity() {
        return 1;
    }
}
```

Unique Fluids Prohibit Re-Generation

- **Some Fluids may be unique, with no way to amplify**
  - E.g., products of cell lysis
- **Users can express this constraint using a UniqueFluid:**

```
class UniqueFluid extends Fluid {
    Fluid[] regenerate() {
        throw new EmptyFluidException();
    }
}
```

```
UniqueFluid f = lysisProduct();
UniqueFluid[] diluted = dilute(f);
for (int i=0; i<diluted.length; i++) {
    analyze(diluted[i]);
}
```

- **Can compiler verify that unique fluids used only once?**
  - Unique (linear) types is a rich research area in prog. languages
    [Wadler] [Hogg] [Baker] [Minsky] [Boyland] [Fahndrich & DeLine]
  - But solutions often require annotations & do not handle arrays
  - Practical approach: verify in simple cases, warn about others

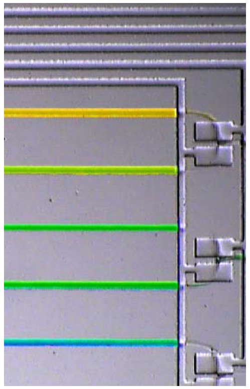
→Opportunity for programming language research

Abstraction 3: Arbitrary Mixing

```
Fluid[] out = new Fluid[8];
Fluid yellow = input(0);
Fluid blue = input(1);
Fluid green = mix(yellow, blue);

out[0] = yellow;
out[1] = mix(yellow, green);
out[2] = green;
out[3] = mix(blue, green);
out[4] = blue;
```

2. Fluid Re-Generation

```
Fluid[] out = new Fluid[8];
Fluid yellow = input (0);
Fluid blue = input (1);

out[0] = yellow;
out[1] = mix(yellow, 3/4, blue, 1/4);
out[2] = mix(yellow, 1/2, blue, 1/2);
out[3] = mix(yellow, 1/4, blue, 3/4);
out[4] = blue;
```



3. Arbitrary Mixing

- **Allows mixing fluids in any proportion, not just 50/50**
  - Fluid **mix** (Fluid F_1, float p_1, Fluid f_2, float F_2)

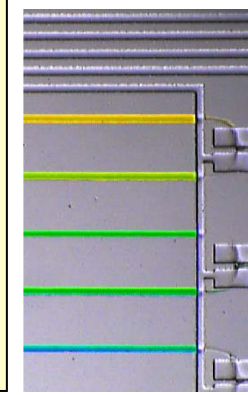
    → Returns Fluid that is p_1 parts $\mathbf{F_1}$ and $\mathbf{p_2}$ parts $\mathbf{F_2}$
  - Runtime system translates to 50/50 mixes in Fluidic ISA
  - Note: some mixtures only reachable within error tolerance ε

Abstraction 3: Arbitrary Mixing

```
Fluid[] out = new Fluid[8];
Fluid yellow = input (0);
Fluid blue = input (1);

out[0] = yellow;
out[1] = mix(yellow, 3/4, blue, 1/4);
out[2] = mix(yellow, 1/2, blue, 1/2);
out[3] = mix(yellow, 1/4, blue, 3/4);
out[4] = blue;
```

3. Arbitrary Mixing

```
Fluid[] out = new Fluid[8];
Fluid yellow = input (0);
Fluid blue = input (1);

for (int i=0; i<=4; i++) {
  out[i] = mix(yellow, 1-i/4, blue, i/4);
}
```

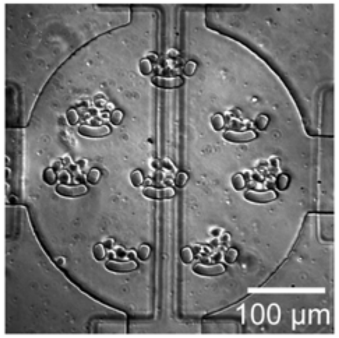


4. Parameterized Mixing

- **Allows mixing fluids in any proportion, not just 50/50**
  - Fluid **mix** (Fluid F_1, float p_1, Fluid f_2, float F_2)
    - → Returns Fluid that is p_1 parts $\mathbf{F_1}$ and $\mathbf{p_2}$ parts $\mathbf{F_2}$
  - Runtime system translates to 50/50 mixes in Fluidic ISA
  - Note: some mixtures only reachable within error tolerance ε

Abstraction 4: Cell Traps

- **Unlike fluids, cells adhere to a specific location on chip**
  - To interact with cells, need to move Fluids to their location



- **CellTrap abstraction establishes a fixed chamber on chip**
  - Fundamental capability: fill with a given fluid (incl. cell culture)

```
class CellTrap {

    // establish a new, empty location on chip
    CellTrap();

    // replace contents of cell trap with new fluid; return old contents
    UniqueFluid drainAndRefill(Fluid newContents);

    // regenerate contents of cell trap; return drained fluid as needed
    Fluid drainAndRegenerate();
}
```

Abstraction 4: Cell Traps

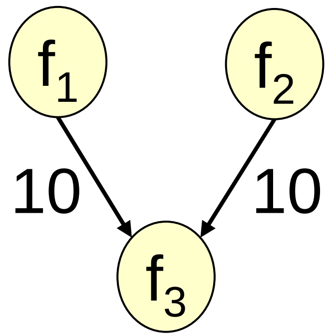
```
CellTrap celltrap = new CellTrap();                   // setup cell culture
for (int i=0; i<N; i++)
    celltrap.drainAndRefill(cellCulture);

celltrap.drainAndRefill(distilledWater);              // analyze cell metabolites
Fluid metabolites = drainAndRegenerate();
analyzeWithIndicators(metabolites);

celltrap.drainAndRefill(antibodyStain);               // stain cells for imaging
```

→ **Must schedule all uses of metabolites before staining**

- Otherwise, runtime error
- Like unique variables, difficult to verify safety in general case
- But thanks to language, compiler can give useful warnings

Abstraction 5: Timing Constraints

- **Precise timing is critical for many biology protocols**
  - Minimum delay: cell growth, enzyme digest, denaturing, etc.
  - Maximum delay: avoid precipitation, photobleaching, etc.
  - Exact delay: regular measurements, synchronized steps, etc.
- **Simple API for indicating timing constraints:**
  - fluid.**useBetween**(N, M)
  - celltrap.**useBetween**(N, M)
  - → Schedule next use of a Fluid (or drain of a CellTrap) between N and M seconds from time of the call
  - → Also becomes part of Fluid's regeneration history
- **Note: may *require* parallel execution**
  - Fluid f1 = mix(…); f1.useBetween(10, 10);
  - Fluid f2 = mix(…); f2.useBetween(10, 10);
  - Fluid f3 = mix(f1, f2);



Scheduling the Execution

- **Scheduling problem has two parts:**
  1. Given dependence graph, find a good schedule
  2. Extract dependence graph from the program

1. Finding a Schedule

Abstract scheduling problem:

– Given task graph G = (V, E) with [min, max] latency per edge

– Find shortest schedule ($V \mapsto Z$) respecting latency on each edge

→ Case 1: Unbounded parallelism

- Can express as system of linear difference constraints
- Solve optimally in polynomial time

→ Case 2: Limited parallelism

- Adds constraint: only k vertices can be scheduled at once
- Can be shown to be NP-hard (reduce from PARTITION)
- Rely on greedy heuristics for now

2. Extracting Dependence Graph

- **Static analysis difficult due to aliasing, etc.**
  - Requires extracting precise producer-consumer relationships
- **Opportunity:**
  Perform scheduling at runtime, using lazy evaluation
  - Microfluidic operations are slow → computer can run ahead
  - Build dependence graph of all operations up to decision point
- **Hazard: constraints that span decision points**
  - Dynamic analysis cannot look into upcoming control flow
  - We currently prohibit such constraints – leave as open problem

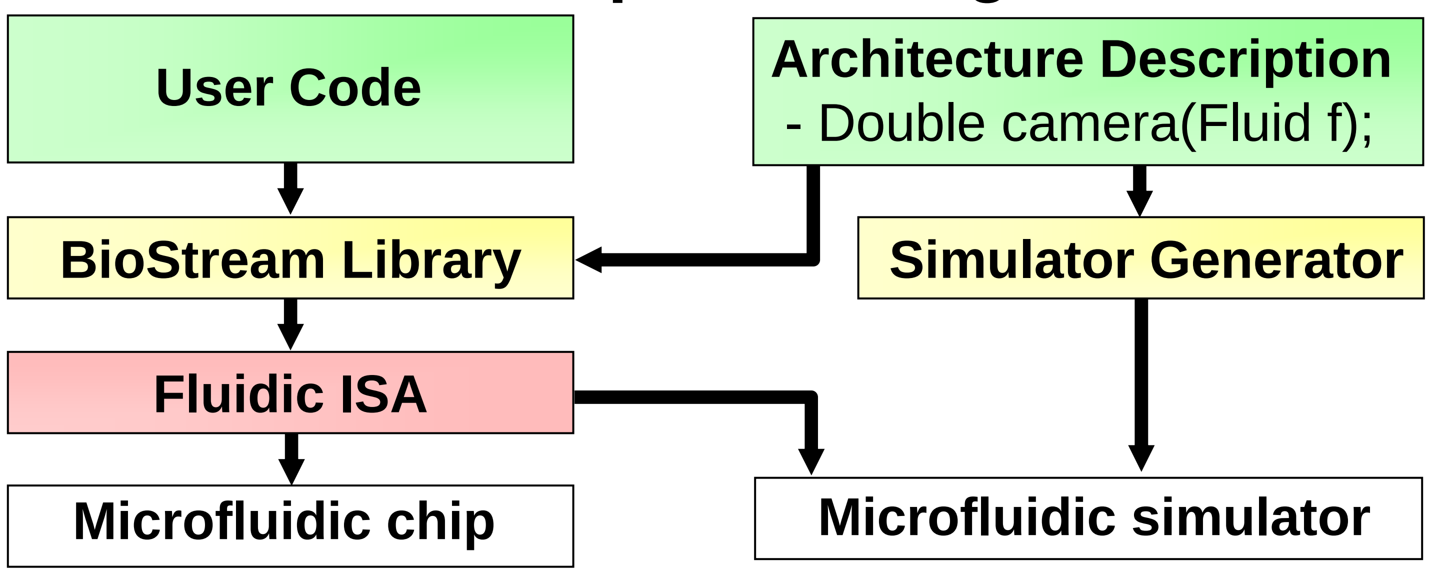
BioStream Protocol Language

- **Implements the abstractions**
  - Full support for storage management, fluid re-generation, arbitrary mixing
  - Partial support for cells, timing

```
Fluid yellow = input (0);
Fluid blue = input (1);
Fluid[] out = new Fluid[8];

for (int i=0; i<=4; i++)
  out[i] = mix(yellow, 1-i/4,
               blue, i/4);
```

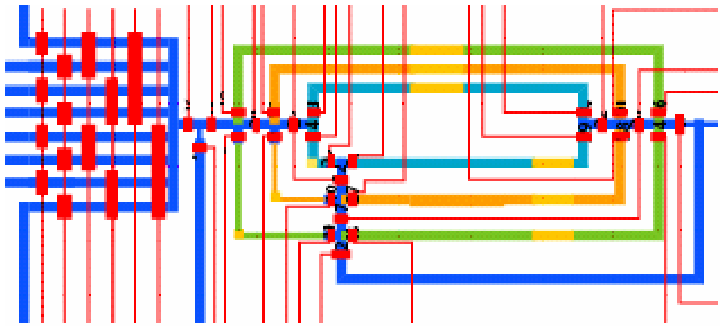
- **Implemented as a Java library**
  - Allows flexible integration with general-purpose Java code

- **Targets microfluidic chips or auto-generated simulator**

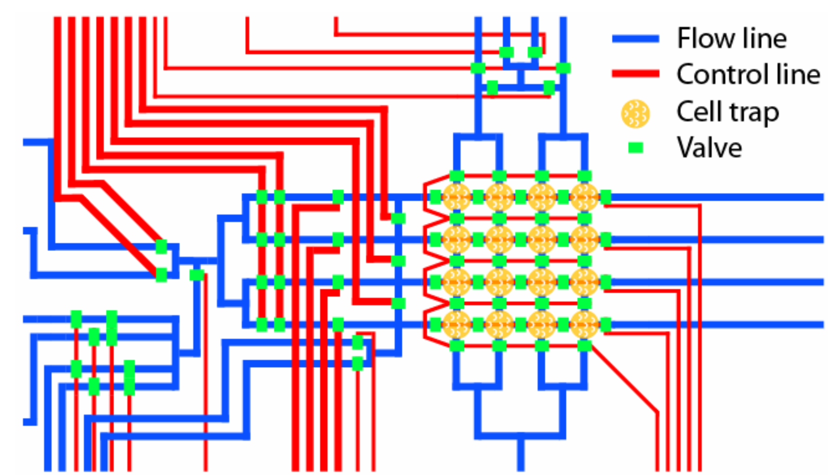


Applications in Progress

1. **What are the best indicators for oocyte viability?**
   - With Mark Johnson's and Todd Thorsen's groups
   - During in-vitro fertilization, monitor cell metabolites and select healthiest embryo for implantation



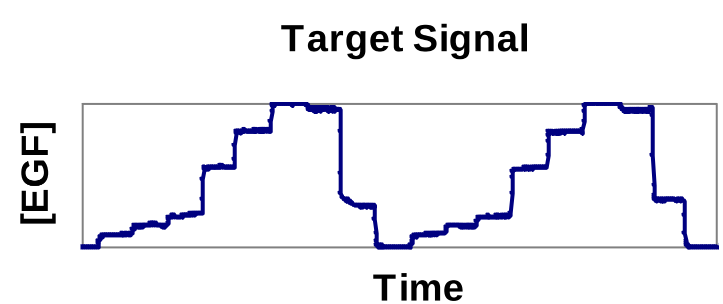
2. **How do mammalian signal transduction pathways respond to complex inputs?**
   - With Jeremy Gunawardena's and Todd Thorsen's groups
   - Isolate cells and stimulate with square wave, sine wave, etc.



Generating Complex Signals

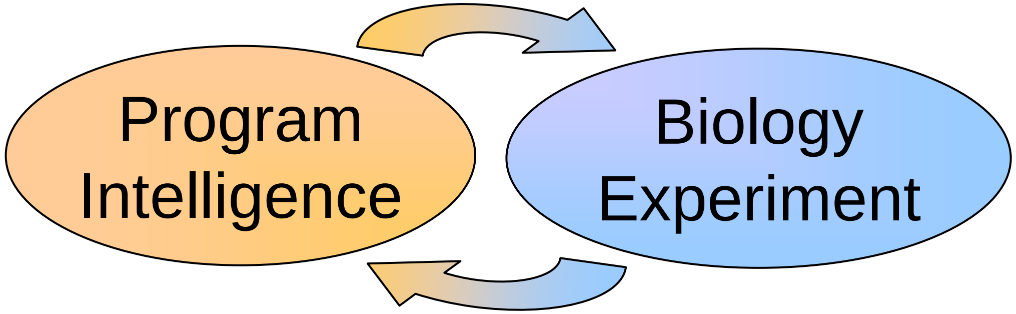
```
CellTrap cells = new CellTrap();
… // setup cell culture
while (true) {
    float target = targetSignal(getTime());
    Fluid f = mix(EGF, target,
                  WATER, 1-target);
    cells.drainAndFill(f);
    cells.useAfter(10*SEC);
}
```

Video courtesy David Craig



Additional Applications

- **Killer apps: react to feedback, redirect the experiment**
  - Recursive-descent search
  - Fixed-pH reaction
  - Directed evolution
  - Long, complex protocols



- **Application to biological computation**
  - Many emerging technologies:
    DNA computing, cellular signaling, biomolecular automata, …
  - But not yet able to assemble, sustain, and adapt themselves
  - Microfluidics provides a scaffold to explore underlying biology

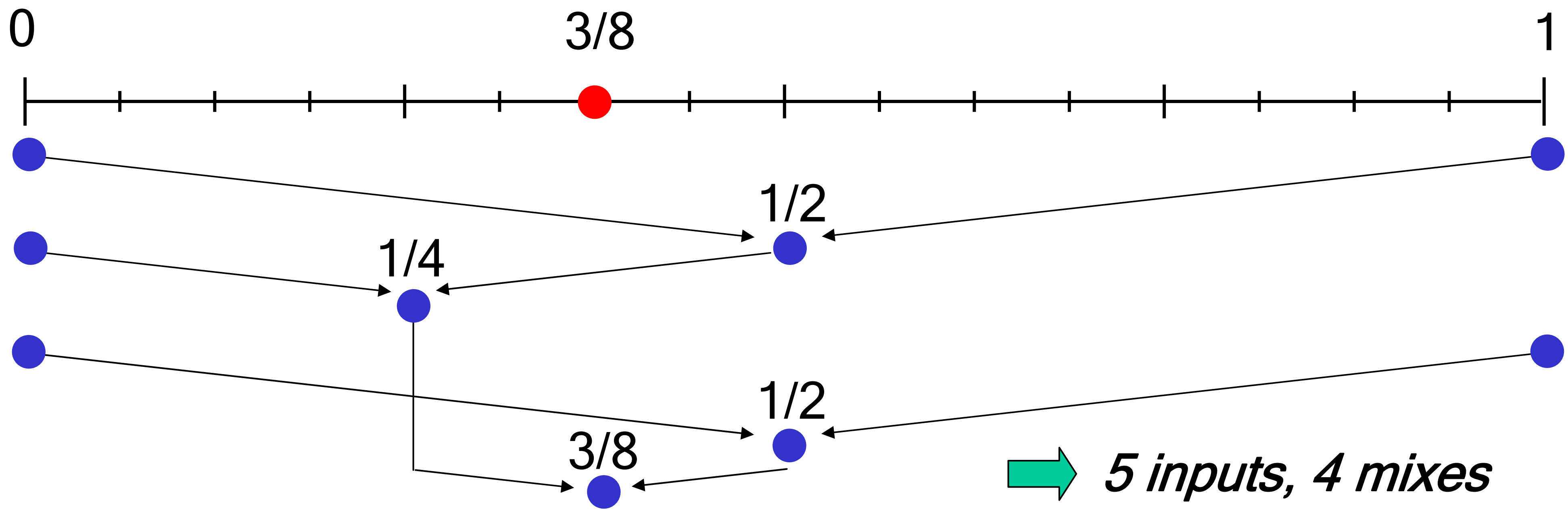
Compiler Optimizations

Algorithms for Efficient Mixing

- **Mixing is fundamental operation of microfluidics**
  - Prepare samples for analysis
  - Dilute concentrated substances
  - Control reagent volumes
  - ➡ Analogous to ALU operations on microprocessors

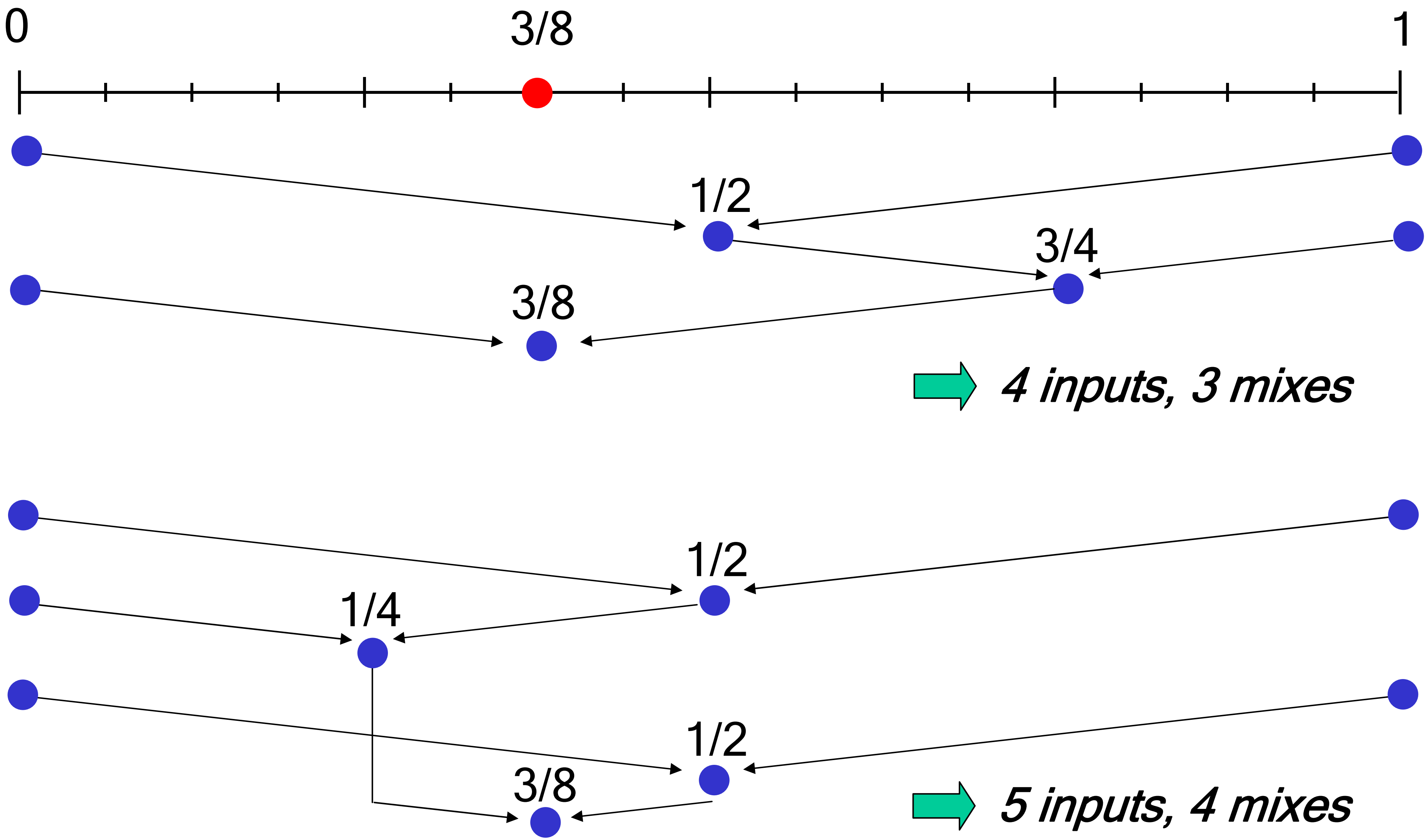
- **How to synthesize complex mixture using simple steps?**
  - Many systems support only 50/50 mixers
  - Should minimize number of mixes, reagent usage
  - Note: some mixtures only reachable within error tolerance ε
  - ➡ Interesting scheduling and optimization problem

Why Not Binary Search?



5 inputs, 4 mixes

Why Not Binary Search?

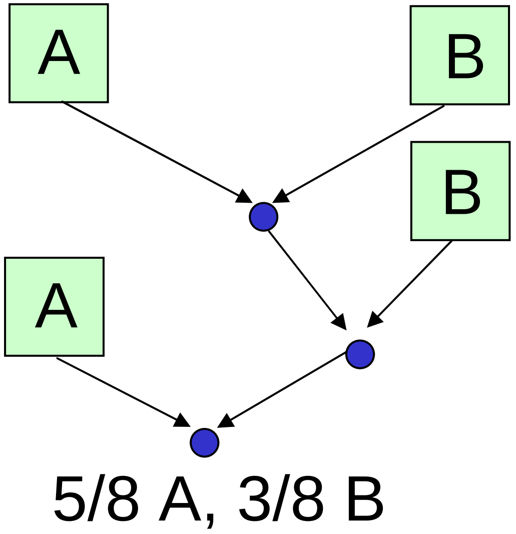


Min-Mix Algorithm

- **Simple algorithm yields minimal number of mixes**
  - For any number of reagents, to any reachable concentration
  - Also minimizes reagent usage on certain chips

Min-Mix Algorithm: Key Insights

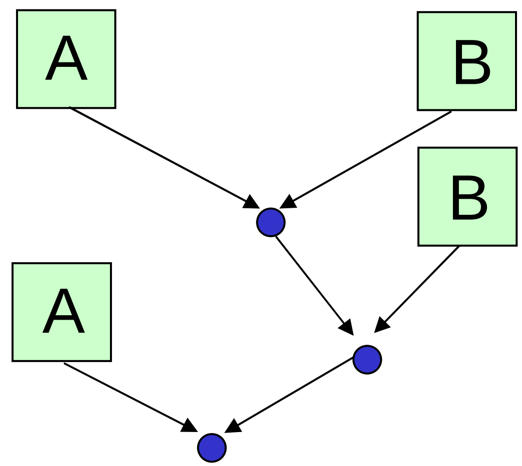
1. The mixing process can be represented by a tree.



Min-Mix Algorithm: Key Insights

1. The mixing process can be represented by a tree.

d	**2^{-d}**
3	1/8
2	1/4
1	1/2

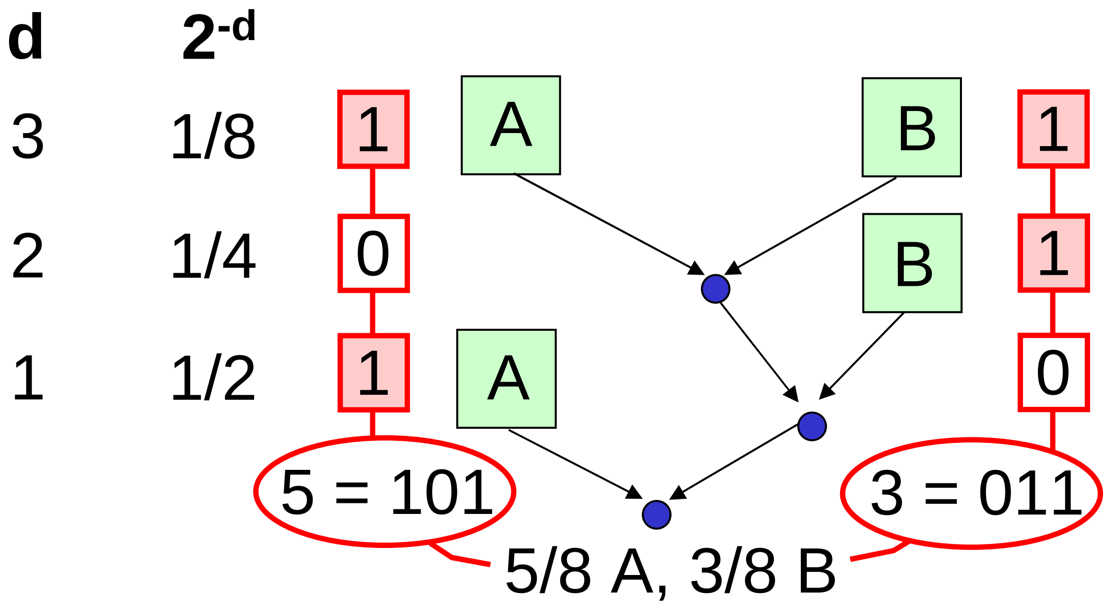


5/8 A, 3/8 B

2. The contribution of an input sample to the overall mixture is 2^{-d}, where d is the depth of the sample in the tree

Min-Mix Algorithm: Key Insights

1. The mixing process can be represented by a tree.



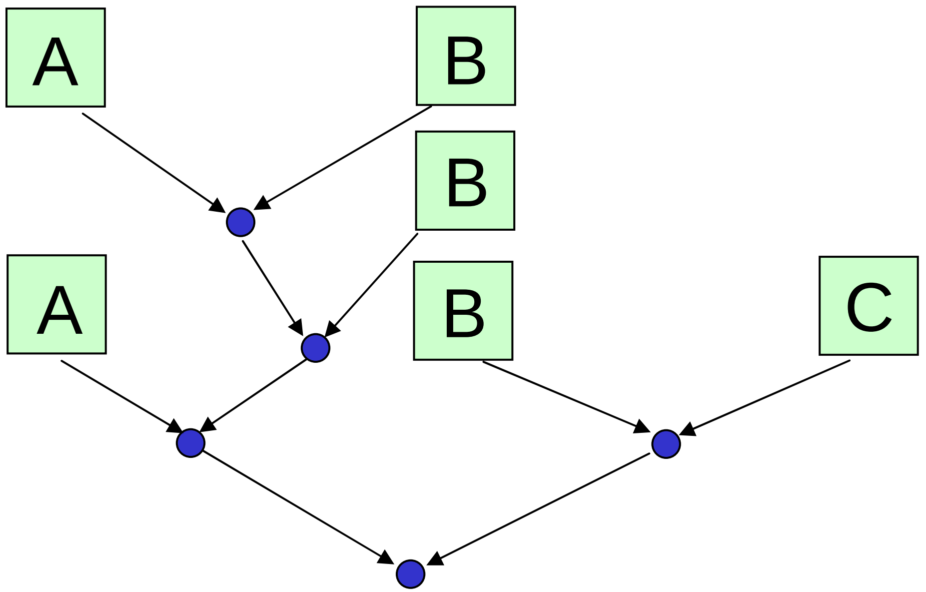
2. The contribution of an input sample to the overall mixture is 2^{-d}, where d is the depth of the sample in the tree

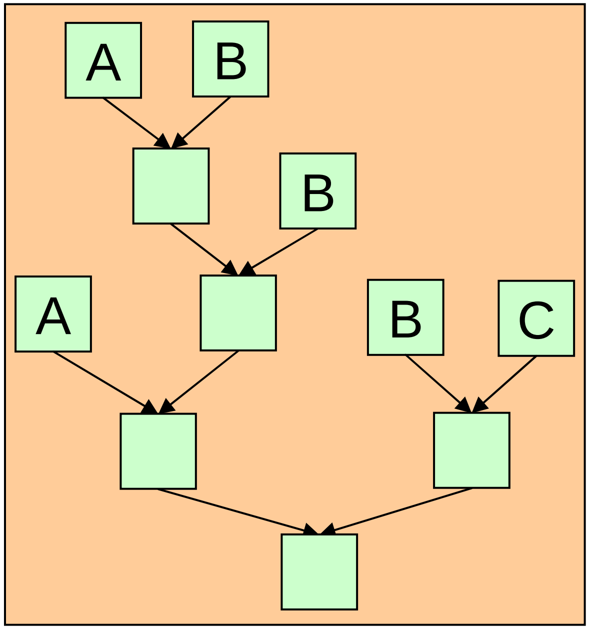
3. In the optimal mixing tree, a reagent appears at depths corresponding to the binary representation of its overall concentration.

Min-Mix Algorithm

- **Example: mix 5/16 A, 7/16 B, 4/16 C**

d	**2^{-d}**
4	1/16
3	1/8
2	1/4
1	1/2





A=5 **B**=7 **C**=4

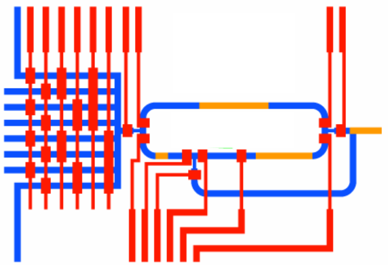
=0101 =0111 =0100

- **In paper: pseudocode, proof of correctness / optimality**

Work In Progress

CAD Tools for Microfluidic Chips

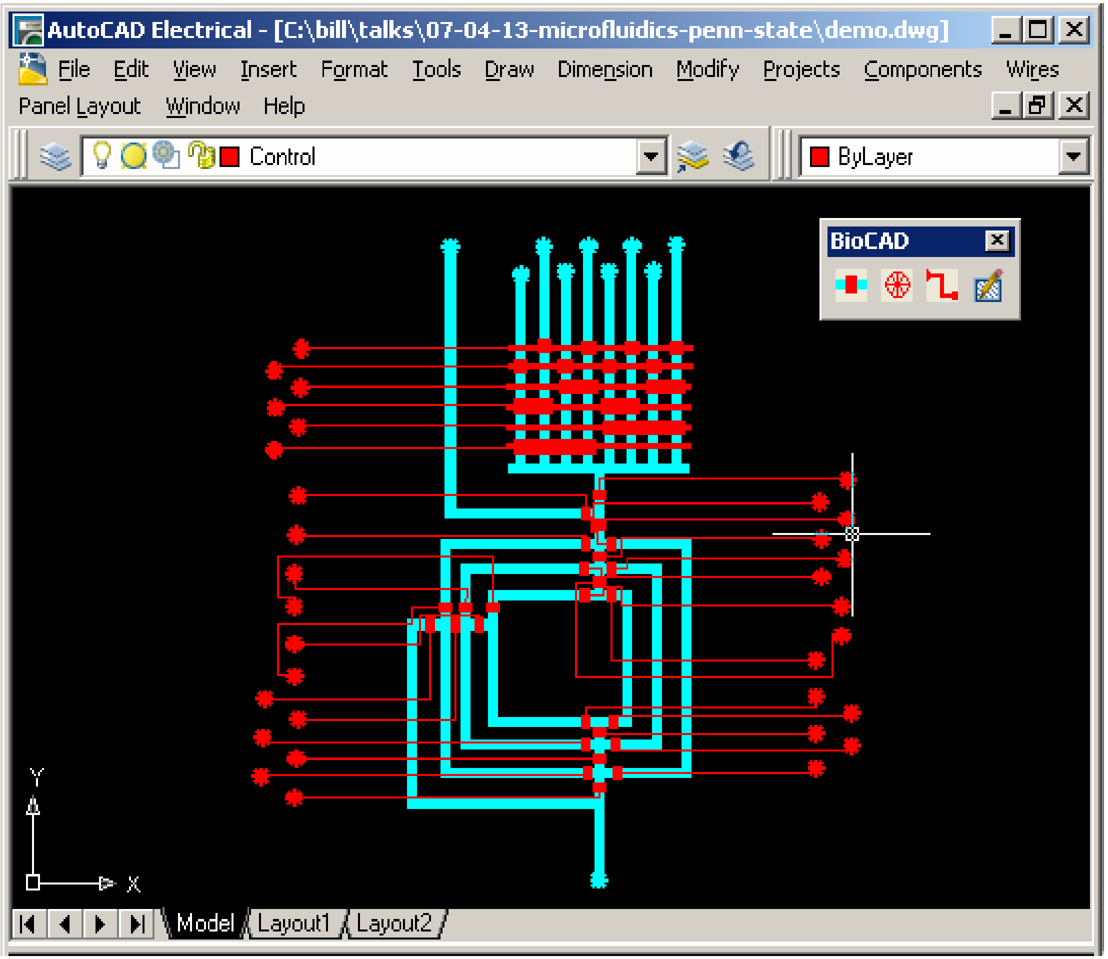
- **Microfluidic design tools are in their infancy**
  - Most groups use Adobe Illustrator or AutoCAD
  - Limited automation; every line drawn by hand



- **Due to fast fabrication, redesign is very frequent**
  - Student can do multiple design cycles per week

First Step: Automatic Routing

- **First target: automate the routing of control channels**
  - Connecting valves to pneumatic ports is very tedious
  - Simple constraints govern the channel placement
- **AutoCAD plugin automates this task**
  - Developed With Nada Amin

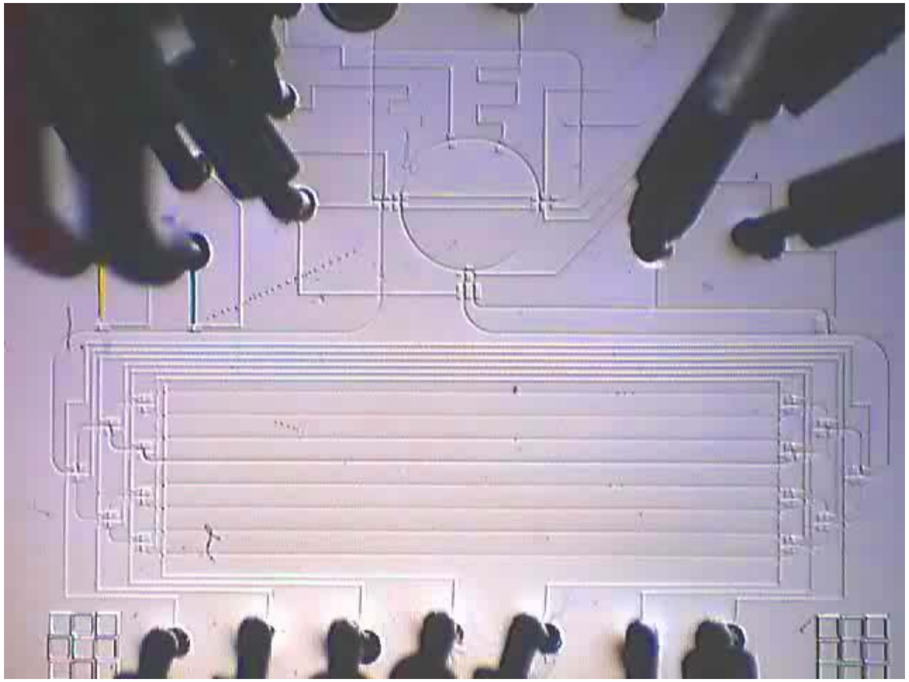


Related Work

- **Aquacore – builds on our work, ISA + architecture** [Amin et al.]
- **Automatic generation / scheduling of biology protocols**
  - Robot scientist: generates/tests genetic hypotheses [King et al.]
  - EDNAC computer for automatically solving 3-SAT [Johnson]
  - Compile SAT to microfluidic chips [Landweber et al.] [van Noort]
  - Mapping sequence graphs to grid-based chips [Su/Chakrabarty]
- **Custom microfluidic chips for biological computation**
  - DNA computing [Grover & Mathies] [van Noort et al.] [McCaskill] [Livstone, Weiss, & Landweber] [Gehani & Reif] [Farfel & Stefanovic]
  - Self-assembly [Somei, Kaneda, Fujii, & Murata] [Whitesides et al.]
- **General-purpose microfluidic chips**
  - Using electrowetting, with flexible mixing [Fair et al.]
  - Using dialectrophoresis, with retargettable GUI [Gascoyne et al.]
  - Using Braille displays as programmable actuators [Gu et al.]

Conclusions

- **Abstraction layers for programmable microfluidics**
  - General-purpose chips
  - Fluidic ISA
  - BioStream language
  - Mixing algorithms



- **Vision for microfluidics: everyone uses standard chip**
- **Vision for software: a defacto language for experimental science**
  - Download a colleague’s code, run it on your chip
  - Compose modules and libraries to enable complex experiments that are impossible to perform today

http://cag.csail.mit.edu/biostream

Extra Slides

How Can Computer Scientists Contribute?

- **Applying the ideas from our field to a new domain**
  - Sometimes requires deep adaptations (e.g., digital gain)
- **Our contributions:**
  - First soft-lithography digital architecture with sample alignment
  - First demonstration of portability: same code, multiple chips
  - New high-level programming abstractions for microfluidics
  - First O(lg n) mixing algorithm for lossy unit volumes (vs O(n))
- **Open problems:**
  - Adapt unique (linear) types to microfluidics
  - Sound scheduling under timing constraints
  - Dynamic optimization of *slow* co-processors (lazy vectorization?)
  - Mixing algorithms for different ISA’s (e.g., lossless mixing)
  - Generate a CAD layout from a problem description