

SWEST20 セッションS5c

「高位合成／FPGA活用 技術の最前線」

講演 1 : FPGAプログラマブルな電子楽器sigboost

青木 海 (sigboost株式会社)

講演 2 : Rubyコードをハードウェアへ

-Mulveryで打ち砕くハードウェアとソフトウェアの壁-

照屋 大地 (東京農工大学)

FPGAは組込みにも大事だけどハードウェア設計が大変なんだよなあ、高位合成なら楽できそうだけどけっきょく頑張らないと性能出ないらしいし、 , , と抱いている方、その考えはもう古いですよ！

本セッションでは、高位合成技術の提案でIPA未踏事業に採択され、その開発成果によってスーパークリエイターにも認定された気鋭の若手技術者・研究者が、高位合成とFPGA活用技術の最前線を語ります！講演者が開発されている、ビジュアル言語ベースでFPGAプログラマブルな電子楽器「sigboost」と、RubyのReactive Programmingに基づく「Mulvery」についてご紹介します。

高位合成とFPGA活用によるシステム設計の新時代に飛び込みましょう！

FPGAプログラマブルな電子楽器sigboost

FPGAを音声信号処理に活用したプログラマブルな電子楽器「sigboost」と、プログラマブル部分をサポートする高位合成処理系「sigboostHLS」についてご紹介します。

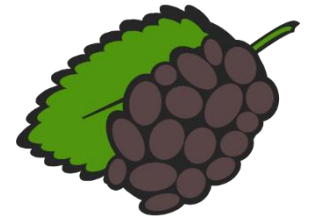


電子楽器はユーザーの入力に応じてリアルタイムな音声信号処理を行うため、スループットを確保したりレイテンシをできる限り短くすることが重要です。通常、これらは楽器メーカーのエンジニアリングやチューニングによって担保される部分ですが、プログラマブルな電子楽器ではアーティストがロジックを記述するため、これらの担保は難しくなります。

sigboostプロジェクトはFPGAとビジュアルプログラミング言語からの高位合成によって、簡潔な処理記述と処理性能を同時に実現しようというチャレンジです。また、同様のアプリケーション要件は楽器以外の分野にも存在します。それらへのアプローチについてもご紹介します。

SWEST20 2018 @下呂温泉 31, Sep., 2018

高位合成/FPGA活用技術の最前線
Rubyコードをハードウェアへ
-Mulveryで打ち砕くハードウェアと
ソフトウェアの壁-



東京農工大学 工学府 情報工学専攻
2017年度 未踏事業 スーパークリエイター
照屋大地

自己紹介

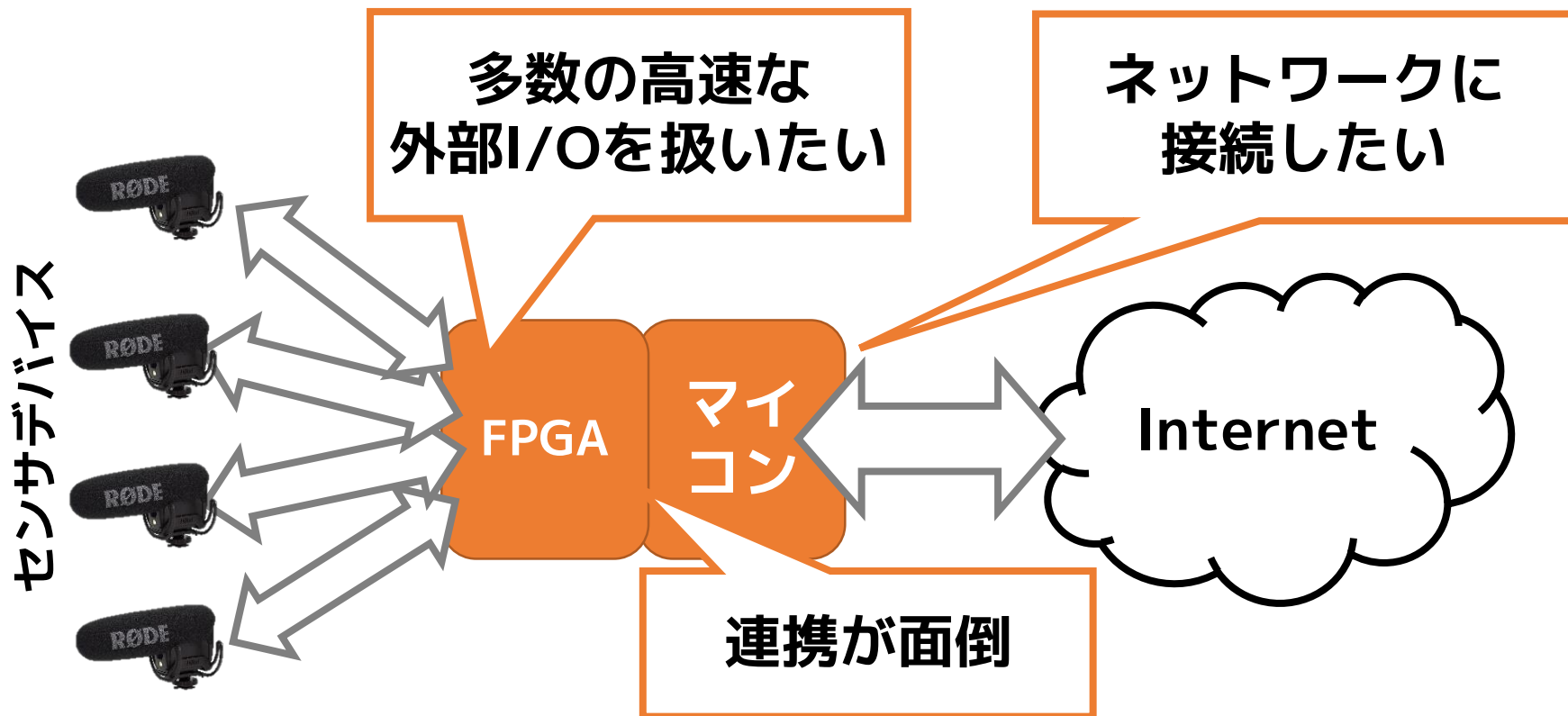
照屋 大地

東京農工大学 工学府
博士後期課程
電子情報工学専攻
知能・情報工学専修

2017年度未踏事業
スーパークリエイータ



きっかけ：IoTデバイス開発の課題



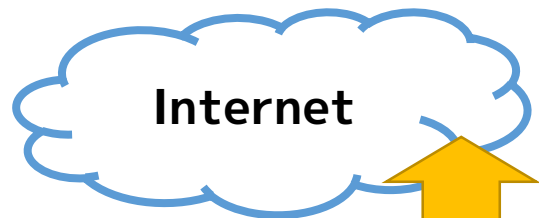
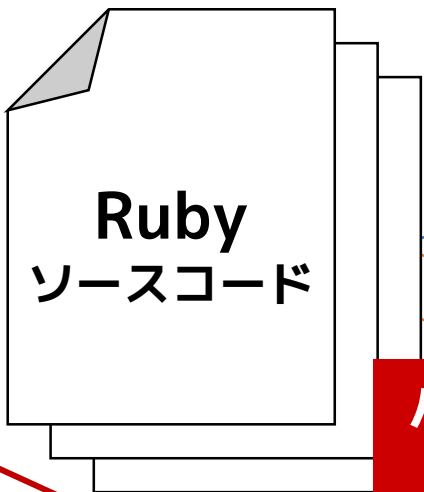
“ぜんぶRubyだけでサクサク書きたい”
Mulveryプロジェクト

Mulveryフレームワークとは？

開発段階

開発するシステム

Mulvery
フレームワーク



CPU

ソフトウェア

FPGA

ハードウェア

回路設計の
知識不要

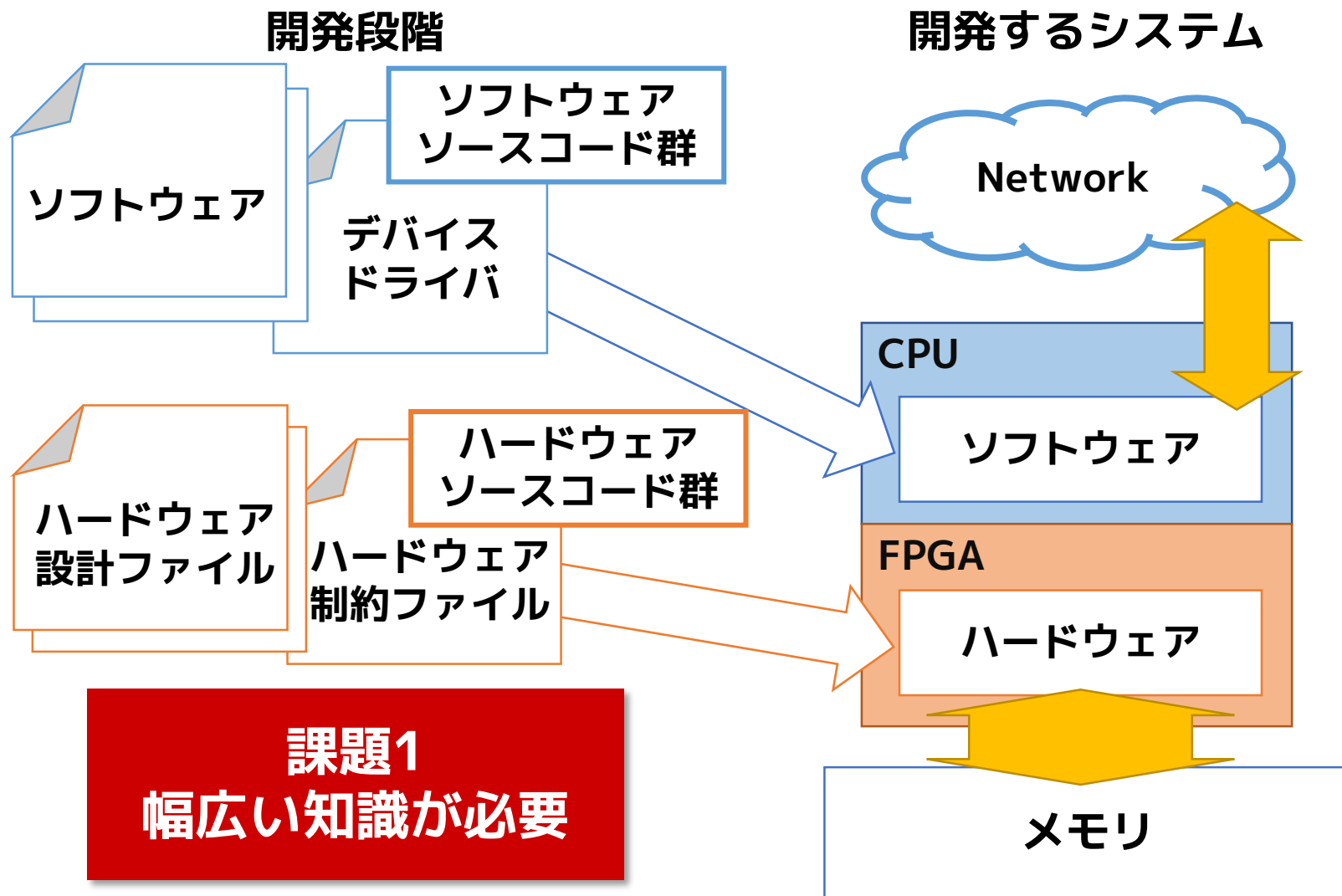
ハード化対象を
自動で抽出

Simple!

センサ
(アクチュエータ)



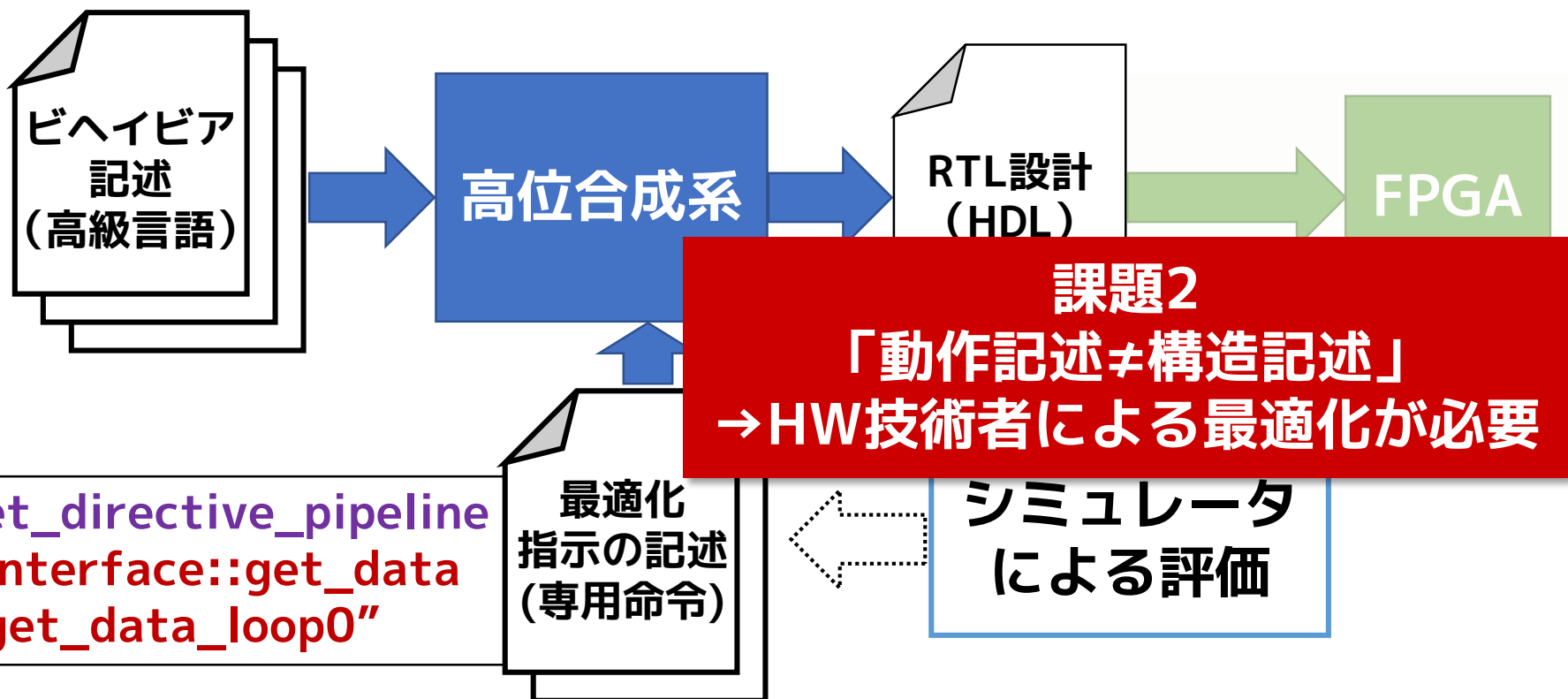
これまでの再構成可能計算システム



高位合成とその複雑さ

- 高位合成(High Level Synthesis, HLS)

CやJava等の**高級言語**を, ハードウェア記述言語(HDL)による**レジスタ転送レベル(RTL)設計に変換**する技術



目指すところ

開発段階

開発するシステム

Mulvery
フレームワーク

mRuby
ソースコード

CPU

ソフトウェア

FPGA

ハードウェア

Internet

回路の知識
不要

FPGAとの
つなぎ込み不要

Simple!

センサ
（アクチュエータ）

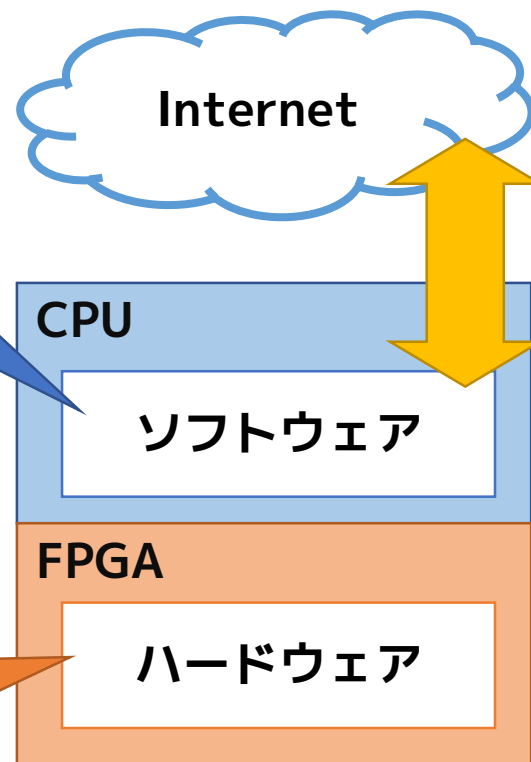


データ処理のアクセラレータにも

開発するシステム

ネットワーク接続,
データのパース・整形,
条件分岐の多い処理, ...

ストリームデータの
アグリゲーション, 操作,
並列処理, ...

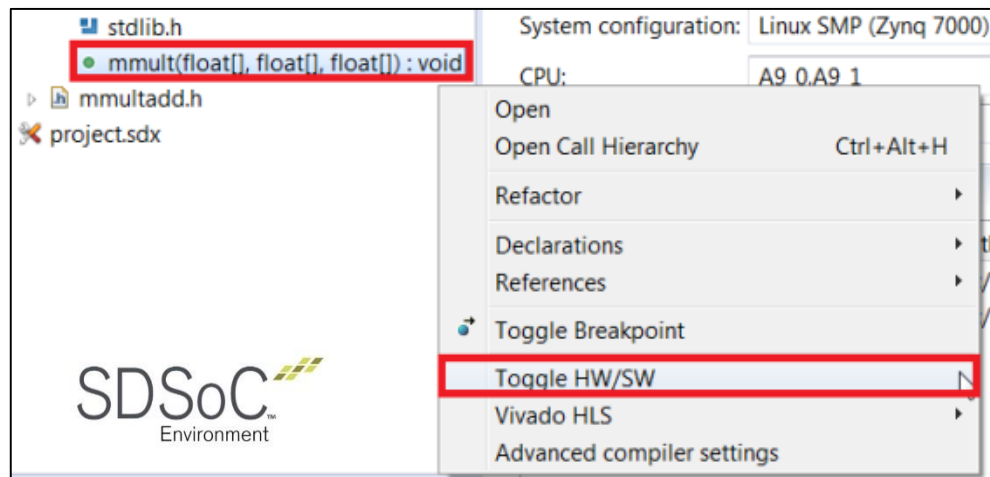


HW-SW協調設計環境

SDSoC[™]
Environment

Intel FPGA SDK
for OpenCL

- C/C++コードの一部をハードウェア化
- オフロード対象を**明示的に指定**する



高速化のための
試行錯誤が必要
⇒HW技術者向け

軽量言語によるメタプログラミング

CHISEL
(Scala)

CLaSH
(Haskell)

- 軽量言語をドメイン固有言語(DSL)として用いる
- **生成されるHWを設計しておく必要がある**
(ScalaもHaskellも強い静的型付けでHW合成しやすそう)

```
class Max2 extends Module {  
  val io = new Bundle {  
    val x = UInt(INPUT, 8)  
    val y = UInt(INPUT, 8)  
    val z = UInt(OUTPUT, 8)  
    io.z = Mux(io.x > io.y, io.x, io.y)  
  }  
}
```

ハードウェア記述言語の
メタプログラミング

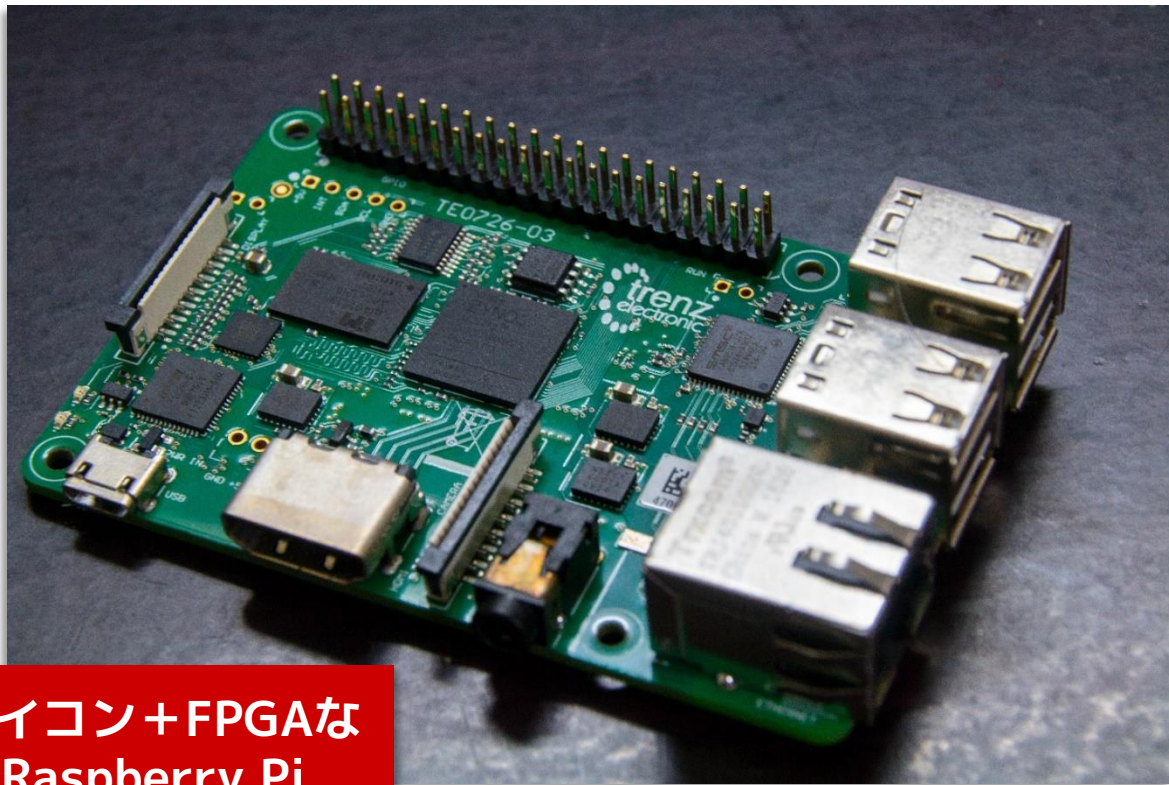
⇒HW技術者向け

マイコン+FPGAな環境？

ラズパイみたいなものもある！



2018/08/31



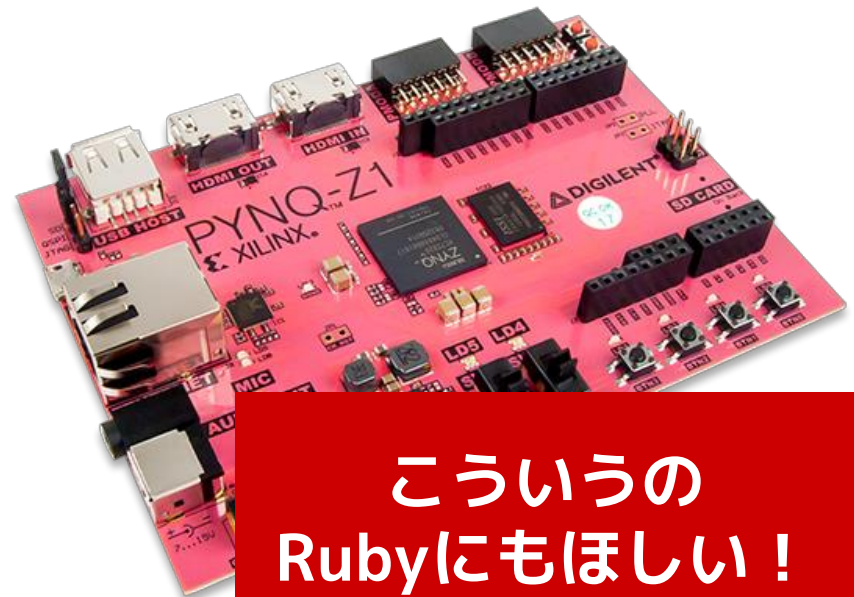
マイコン+FPGAな
Raspberry Pi
互換ボード

Trenz Electronics社
“Zynqberry”

PythonでFPGAを便利に使うヤツ



- ハードウェアをAPI的にPythonから呼び出せる
- SDK+ハードウェア
- **ハードは専門家が作る**
- DeepLearningやリアルタイム画像処理で遊べる



こういうの
Rubyにもほしい！

ReactiveXと ハード合成のアイデア

Mulveryのアプローチ

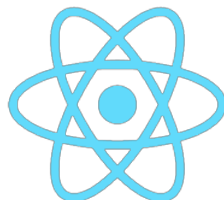
Reactive Programmingを用いたプログラムからの合成

Reactive Programmin (RP)

「データストリーム」をオブジェクトとして扱う
プログラミングパラダイム



Vue.js



React

Web開発



ReactiveX

Web開発
Android開発

今回は
Rxを使ったコードを
速くする

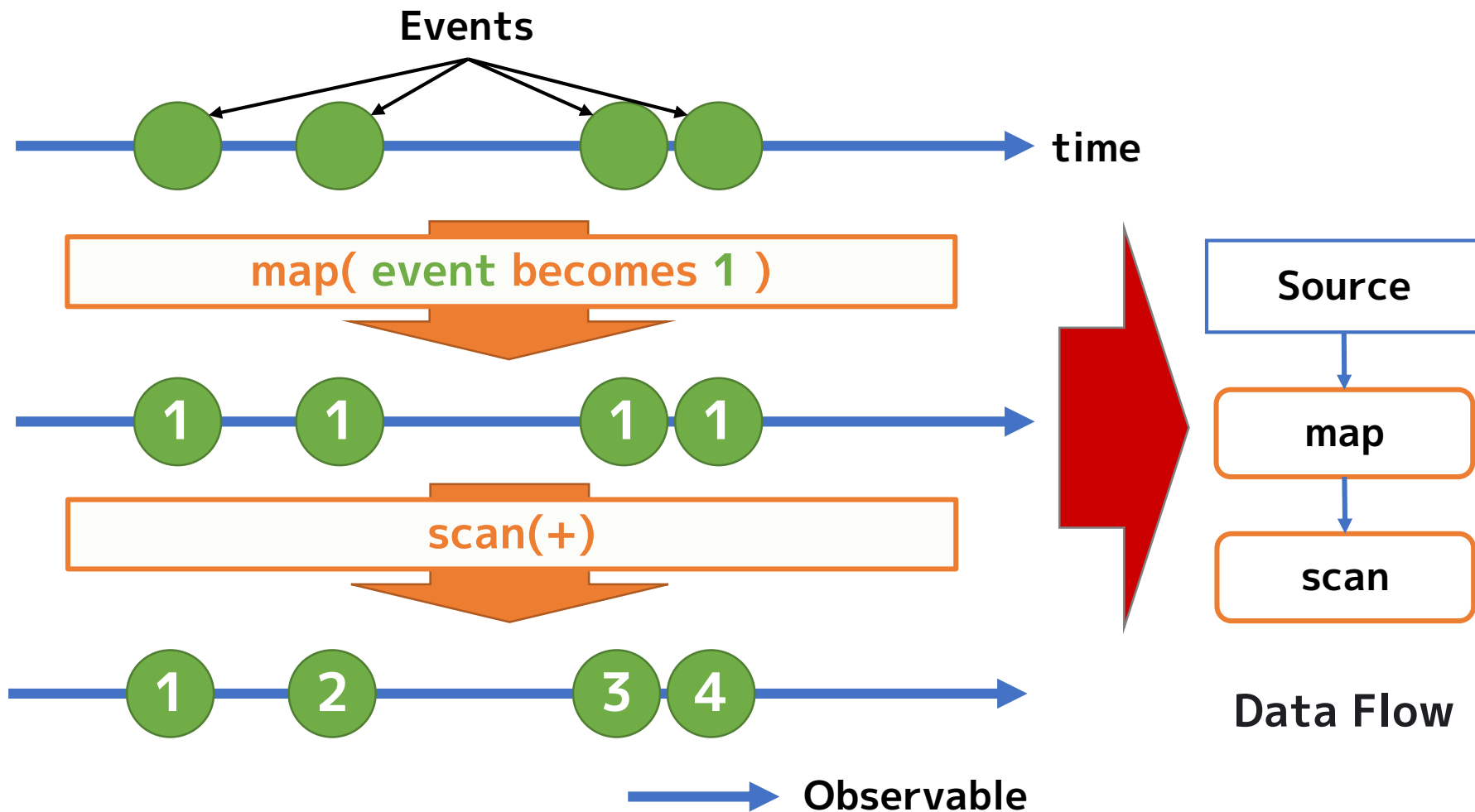
ReactiveXとは

次の3つの要素を組み合わせて
データフローを記述してプログラムを作る

- **Observer Pattern** … 非同期の実現
- **LINQ** ……………… 操作クエリの実装
- **Scheduler** ……………… スケジューリング

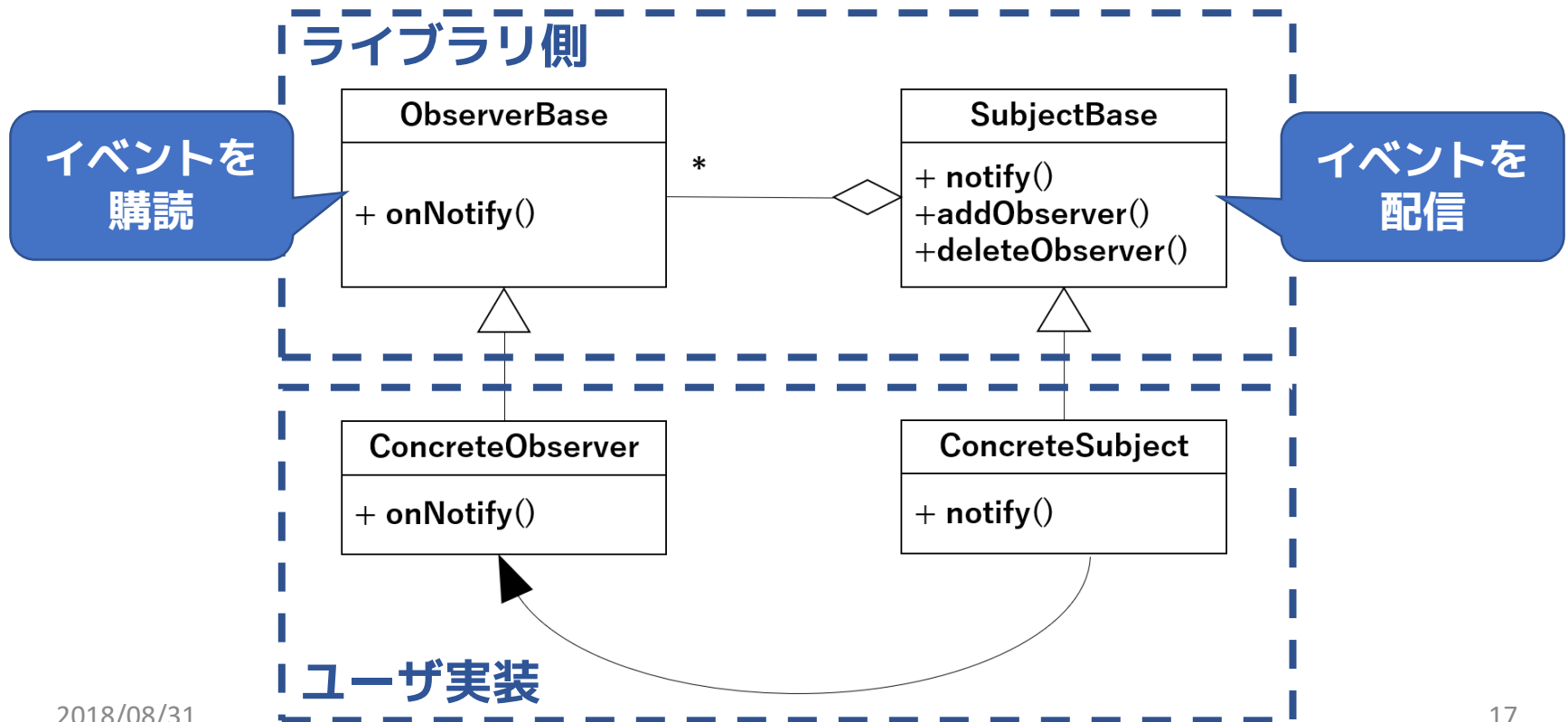
Reactive Programmingの例

- データの到着回数を数える例：



Observer Pattern

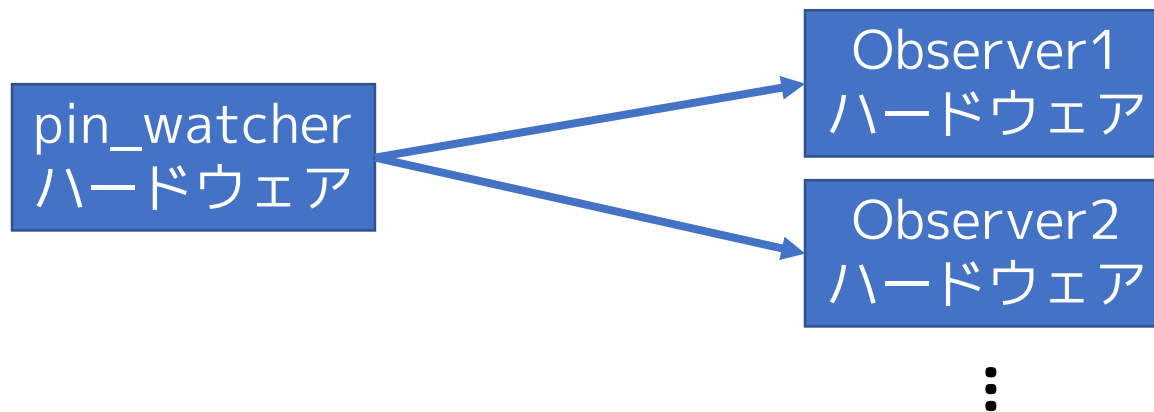
- Publish/Subscribeとも
- ObserverがSubjectを観察する
- RxではSubject=Observable



Observer PatternとHW合成アイデア

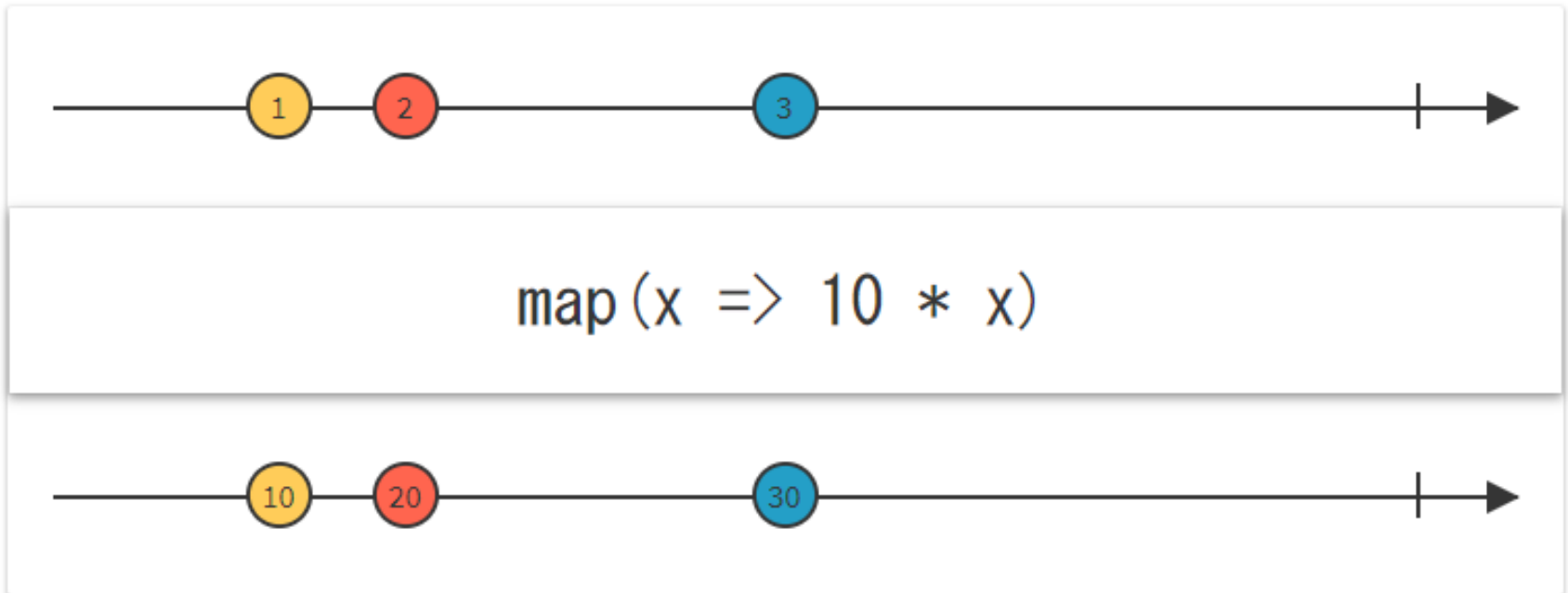
```
class Pin < SubjectBase
  <pin_watcherの定義>
  def notify(event)
    @observers.each do |o|
      o.onNotify(event)
    end
  end
end
```

```
Pin pin_0
pin_0.addObserver do |e|
  <Observer1の定義>
end
pin_0.addObserver do |e|
  <Observer2の定義>
end
```



LINQ

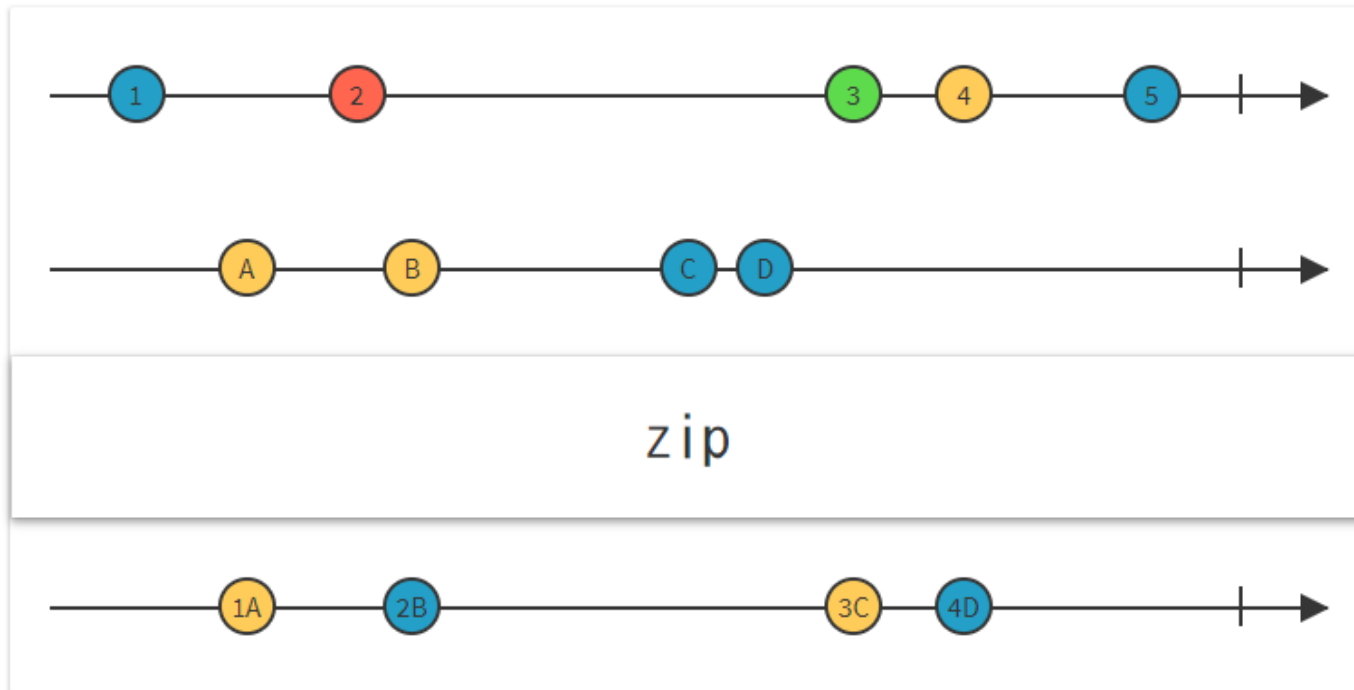
- Language-**IN**tegrated **Q**uery
- データ列（DBとか）を操作するための標準クエリ



ラムダ抽象の内容を各イベントに適用

LINQ

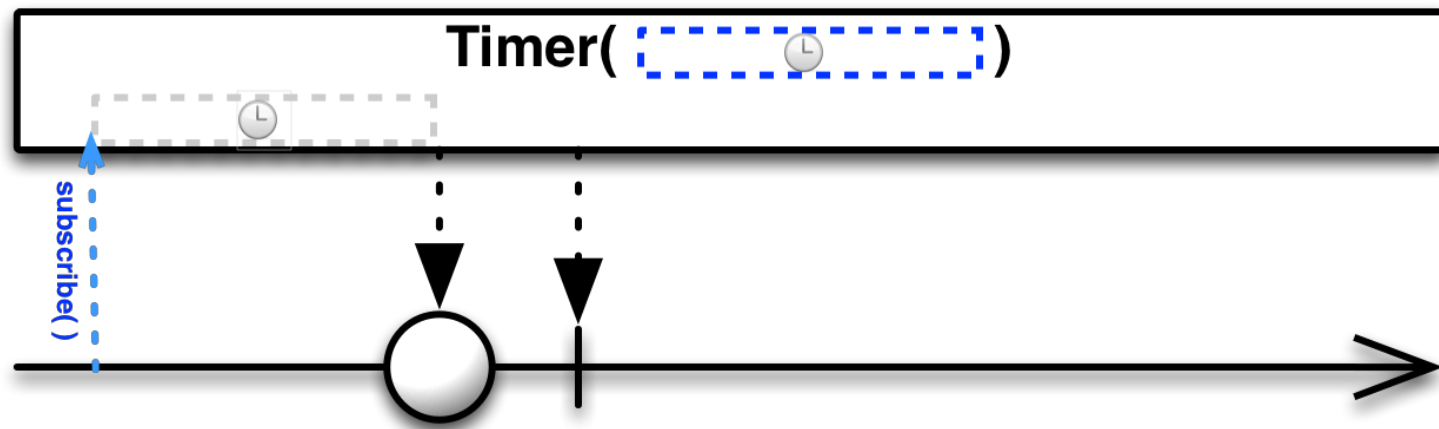
- Language-**IN**tegrated **Q**uery
- データ列（DBとか）を操作するための標準クエリ



2つのデータフローを待ち合わせる

LINQ

- Language-**IN**tegrated **Q**uery
- データ列（DBとか）を操作するための標準クエリ



一定時間後にイベントを発生させる

LINQ on Reactive Extensions



ReactiveX Introduction Docs

- **Repeat** — create an Observable that emits
- **Start** — create an Observable that emits t
- **Timer** — create an Observable that emits a

Transforming Observables

Operators that transform items that are emitted b

- **Buffer** — periodically gather items from an Observable and emit the items one at a time
- **FlatMap** — transform the items emitted by an Observable into a single Observable
- **GroupBy** — divide an Observable into a set of Observables, organized by key
- **Map** — transform the items emitted by an Observable
- **Scan** — apply a function to each item emitted by an Observable, returning a value
- **Window** — periodically subdivide items from an Observable into windows rather than emitting the items one at a time

Filtering Observables

Operators that selectively emit items from a source Observable.

- **Debounce** — only emit an item from an Observable if a particular timespan has passed without it emitting another item
- **Distinct** — suppress duplicate items emitted by an Observable
- **ElementAt** — emit only item *n* emitted by an Observable
- **Filter** — emit only those items from an Observable that pass a predicate test
- **First** — emit only the first item, or the first item that meets a condition, from an Observable
- **IgnoreElements** — do not emit any items from an Observable but mirror its termination notification
- **Last** — emit only the last item emitted by an Observable
- **Sample** — emit the most recent item emitted by an Observable within periodic time intervals

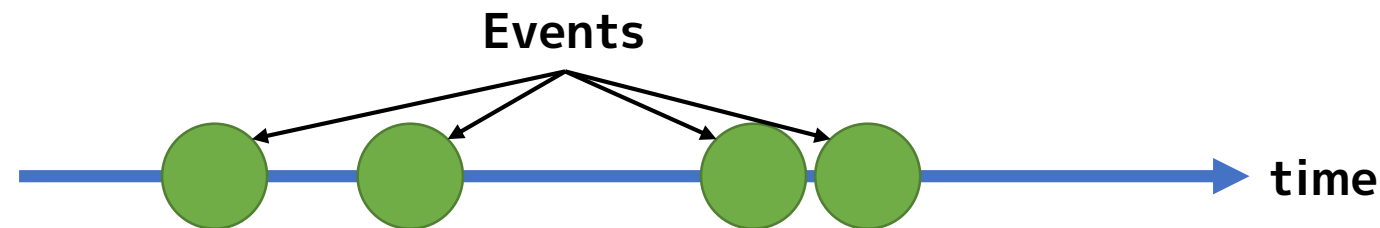
Transforming Observables

Operators that transform items that are emitted by an Observable

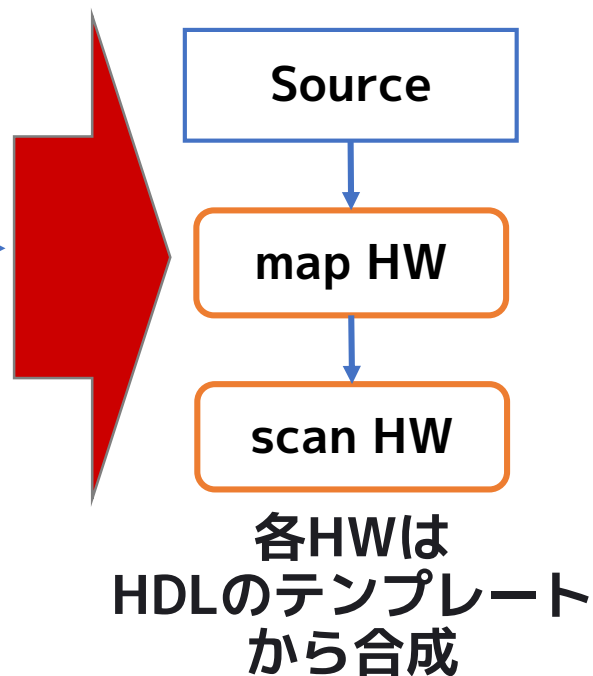
- **Buffer** — periodically gather items from an Observable and emit the items one at a time
- **FlatMap** — transform the items emitted by an Observable into a single Observable
- **GroupBy** — divide an Observable into a set of Observables, organized by key
- **Map** — transform the items emitted by an Observable
- **Scan** — apply a function to each item emitted by an Observable, returning a value
- **Window** — periodically subdivide items from an Observable into windows rather than emitting the items one at a time

LINQとHW合成アイデア

- データの到着回数を数える例（再掲）



Observable



Scheduler

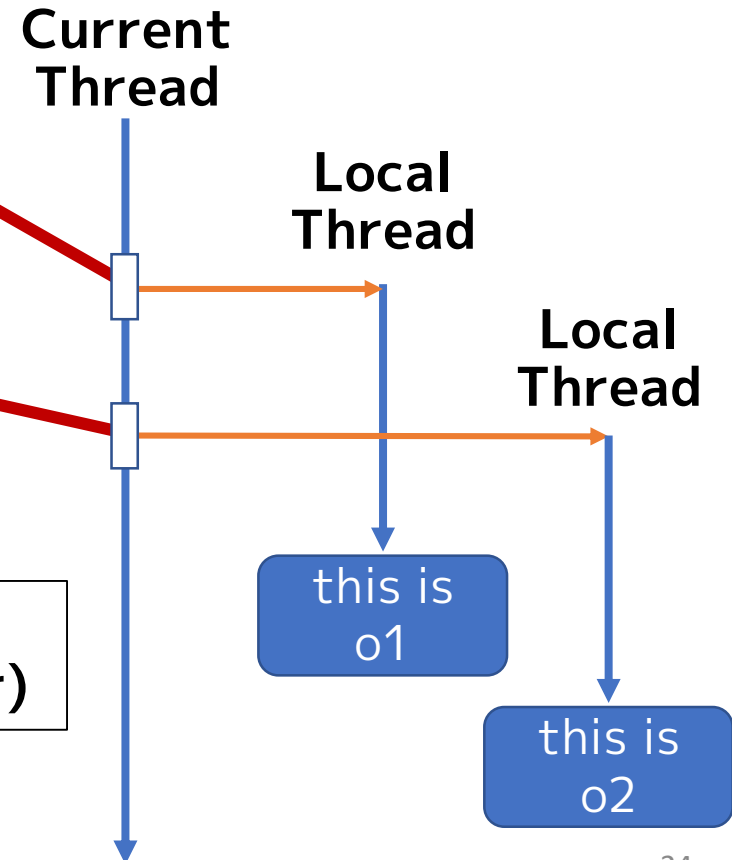
- 動作スレッドを制御するための仕組み
- たとえばTimerだと . . .

```
o1 = Observable.timer(1) do  
  p "this is o1"  
end
```

```
o2 = Observable.timer(1) do  
  p "this is o2"  
end
```

```
timer(len,  
      scheduler=DefaultScheduler)
```

使い方に応じてschedulerを変える



Scheduler

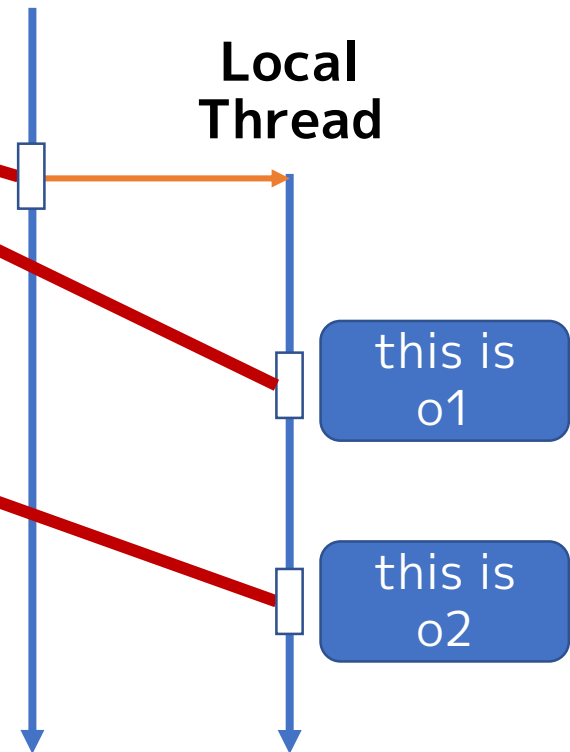
- 同ースレッドで時間計測させる例

SchedulerBaseを使って
任意の戦略が実装可能

```
m_th = LocalScheduler.new  
  
o1 = Observable.timer(1, m_th) do  
  p "this is o1"  
end  
  
o2 = Observable.timer(1, m_th) do  
  p "this is o2"  
end
```

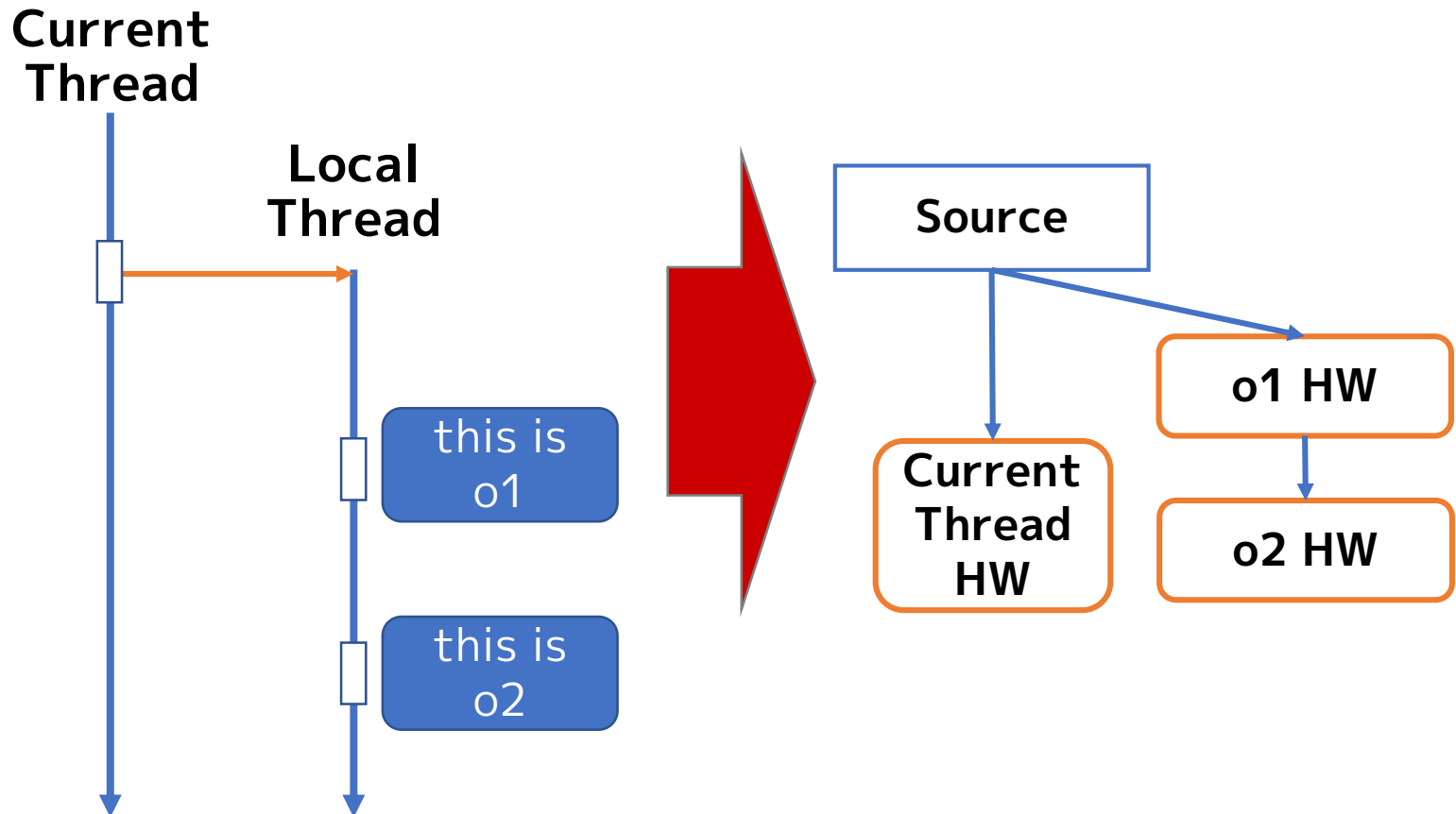
Current
Thread

Local
Thread



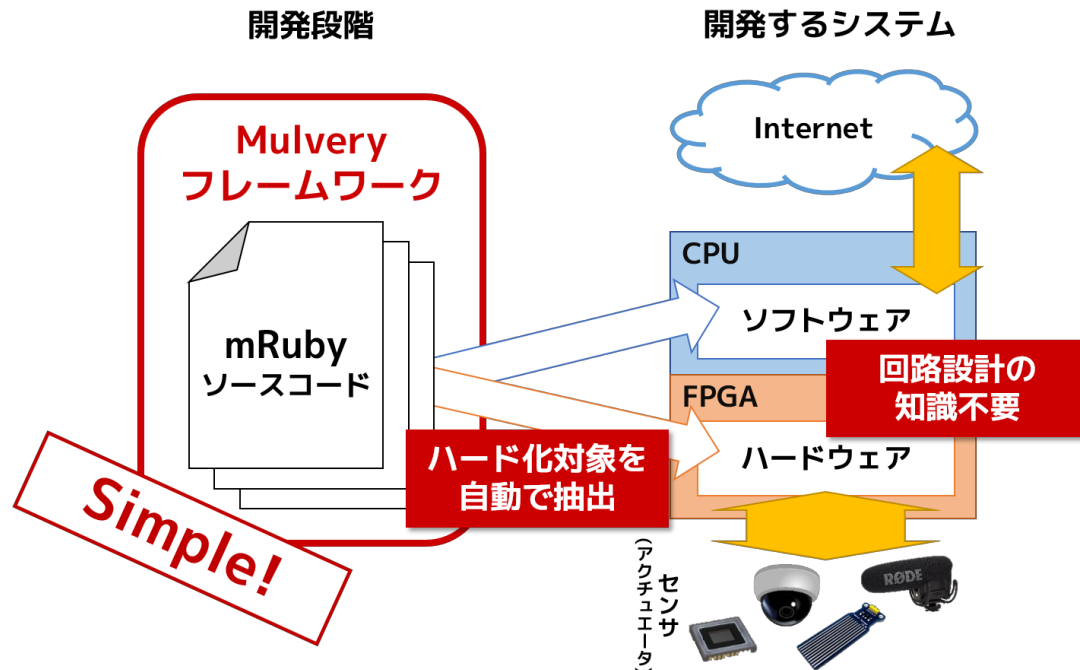
SchedulerとHW合成アイデア

- 動的スケジューリングを制限すれば
そのままデータフローグラフとして扱えそう



これまでの開発

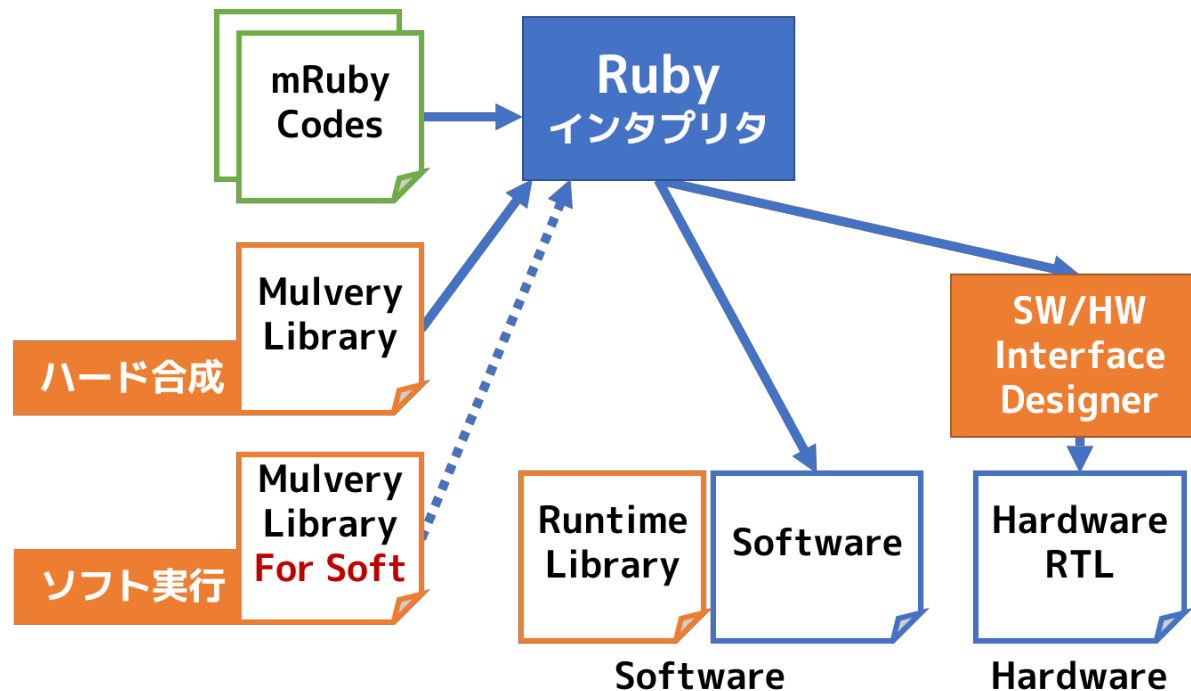
実装しなければならないもの



1. HW抽出機構
2. HW合成機構
3. ラムダ抽象の合成
4. Schedulerの合成
5. ソフト/ハード連携

V2 : 動的解析による実装

- 多段階計算を使った動的解析の実装への舵切り
- フレームワークとして提供するので通常のRubyで合成も実行も全部できる



ハード/ソフトの自動分割

Rx記述⇒ハード それ以外⇒ソフト

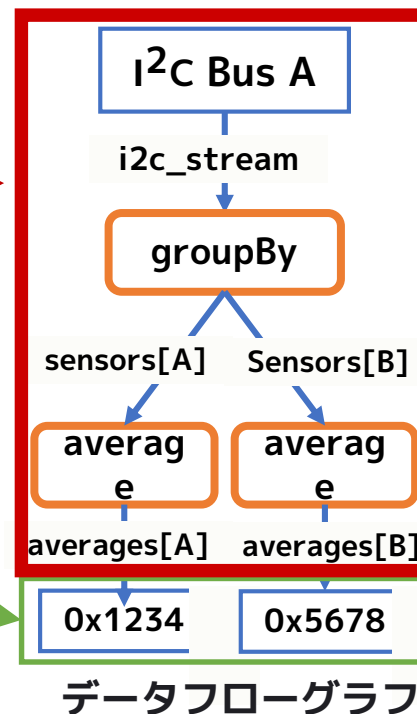
ハードウェア部

```
def initialize()
  i2c_stream = Stream.new(I2C_BUS_A)
  sensors = i2c_stream.group_by(2)
  { |d| d.sensor_id }
  for sensor in sensors do
    @averages[sensor] = sensor.average(5)
  end
end
```

ソフトウェア部

```
def main()
  for average in @averages do
    p average[sensor].get_latest()
  end
  wait_ms(1000)
end
```

Rxで書きにくい部分は
多く場合ソフト実行が速い



ハード化

メモリに
マップ

ハードウェア合成の例

```
events = Stream.new(MIO::p0)
count = events
  .map(){ |event| 1 }
  .scan(){ |acc, x| acc + x }
```

Streamオブジェクトが
ハードウェアを生成する

S_0.v

```
module S_0(p0);
  ...
  Map_0 map_0(
    .din(w_0_0),
    .v_din(v_w_0_0),
    .dout(w_0_1),
    .v_dout(v_w_0_1));
```

```
  Scan_0 scan_0(
    ...
  endmodule
```

Map_0.v

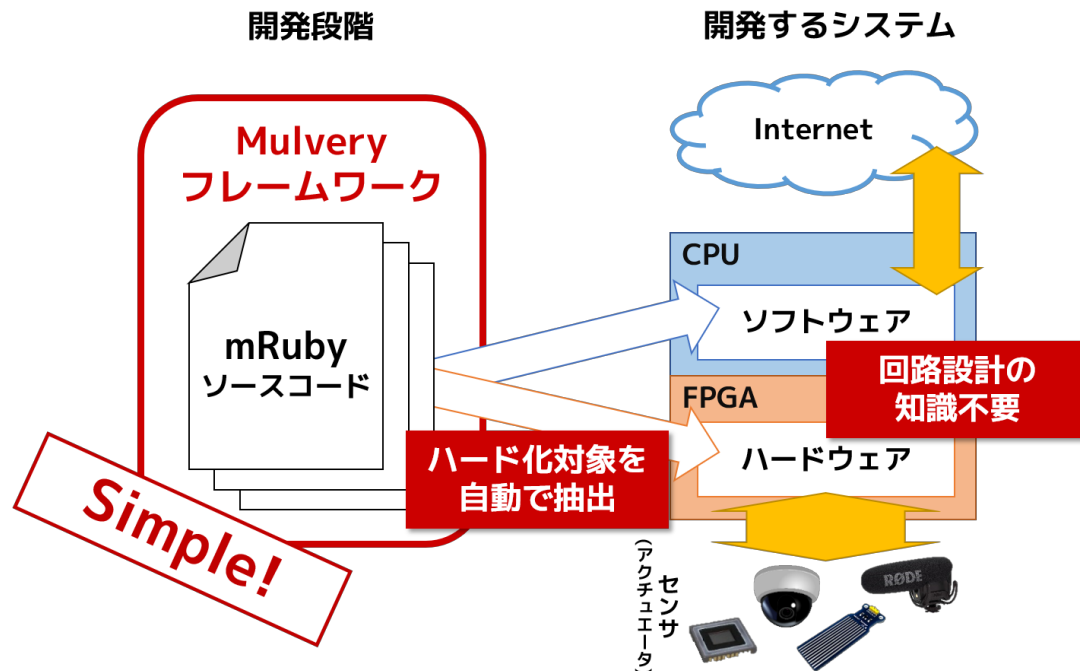
```
module Map_0(...);
  ...
  dout <= 8'd1;
  v_din <= 1'b1;
  ...
endmodule
```

テンプレート
から生成

Scan_0.v

```
module Scan_0(...);
  ...
endmodule
```

実装しなければならないもの





- ✓ 1. HW抽出機構
- ✓ 2. HW合成機構
- 3. ラムダ抽象の合成
- 4. Schedulerの合成
- 5. ソフト/ハード連携

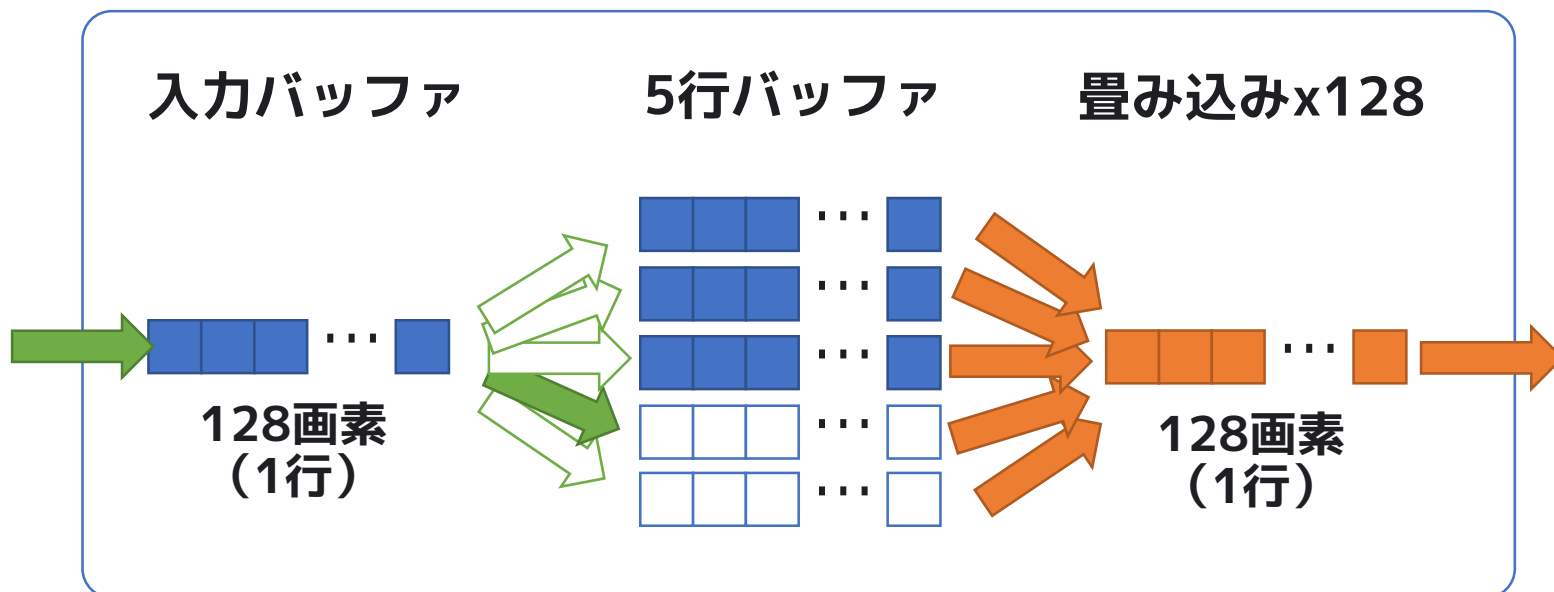
ラムダ抽象の合成について

- LINQクエリの多くは高階関数
 - ラムダ抽象の合成はどうしよう . . .
- ⇒ **現状は用途に合わせた
多段階計算で誤魔化している**

サンプル：畳み込み演算

- 128x128の画像に対するラプラシアンフィルタ適用


$$* \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & -24 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix} =$$




コーディング例

```
buffer = input.sliding_buffer(5)
```

```
line = buffer.map(){ |data|
```

```
  mats = Array.new()
```

```
  for i in (0...SIZE) do
```

```
    mats.push(Matrix[data[0]
```

```
      data[1]
```

```
      data[2][i, 5], ¥
```

```
  end
```

```
  result_row = Array.new()
```

```
  mats.map() do |mat|
```

```
    result_row.push(mat.conv(kernel))
```

```
  end
```

```
}
```

```
result = line.buffer(128)
```

5行ぶん待つ

5行毎に処理（畳み込み）

128行出力待つ

input

sliding_
buffer

map

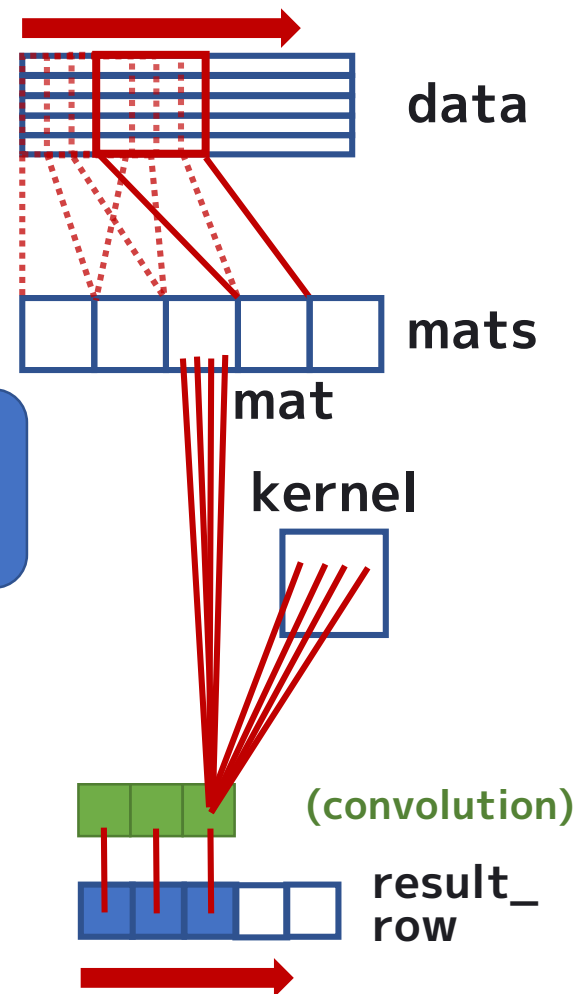
buffer

result

ラムダ抽象をメタプロ的に扱う

```
buffer = input.sliding_buffer(5)
line = buffer.map(){ |data|
  mats = Array.new()
  for i in (0...SIZE) {
    mats.push(Matrix[data[0][i, 5], ¥
                      data[1][i, 5], ¥
                      data[2][i, 5], ¥
                      data[3][i, 5], ¥
                      data[4][i, 5], ¥])
  }
  result_row = Array.new()
  mats.map() { |mat|
    result_row.push(mat.conv(kernel))
  }
}
result = line.buffer(128)
```

Array, Matrixは
オーバーロード
されている



IF文のハードウェア化

```
line = buffer.map(){ |data|  
  check(data == 0) {  
    do_something_1()  
  }  
  .elsewhen(data == 1) {  
    do_something_2()  
  }  
  .otherwise {  
    do_something_3()  
  }  
  .endcheck  
}
```

check

CheckContext#elsewhen

CheckContext#otherwise

CheckContext#endcheck

ハード合成時：すべて評価
ソフト実行時：IFとおなじ

実装しなければならないもの

開発段階

開発するシステム

Mulvery
フレームワーク

mRuby
ソースコード

ハード化対象を
自動で抽出

Simple!



CPU

ソフトウェア

FPGA

ハードウェア

回路設計の
知識不要

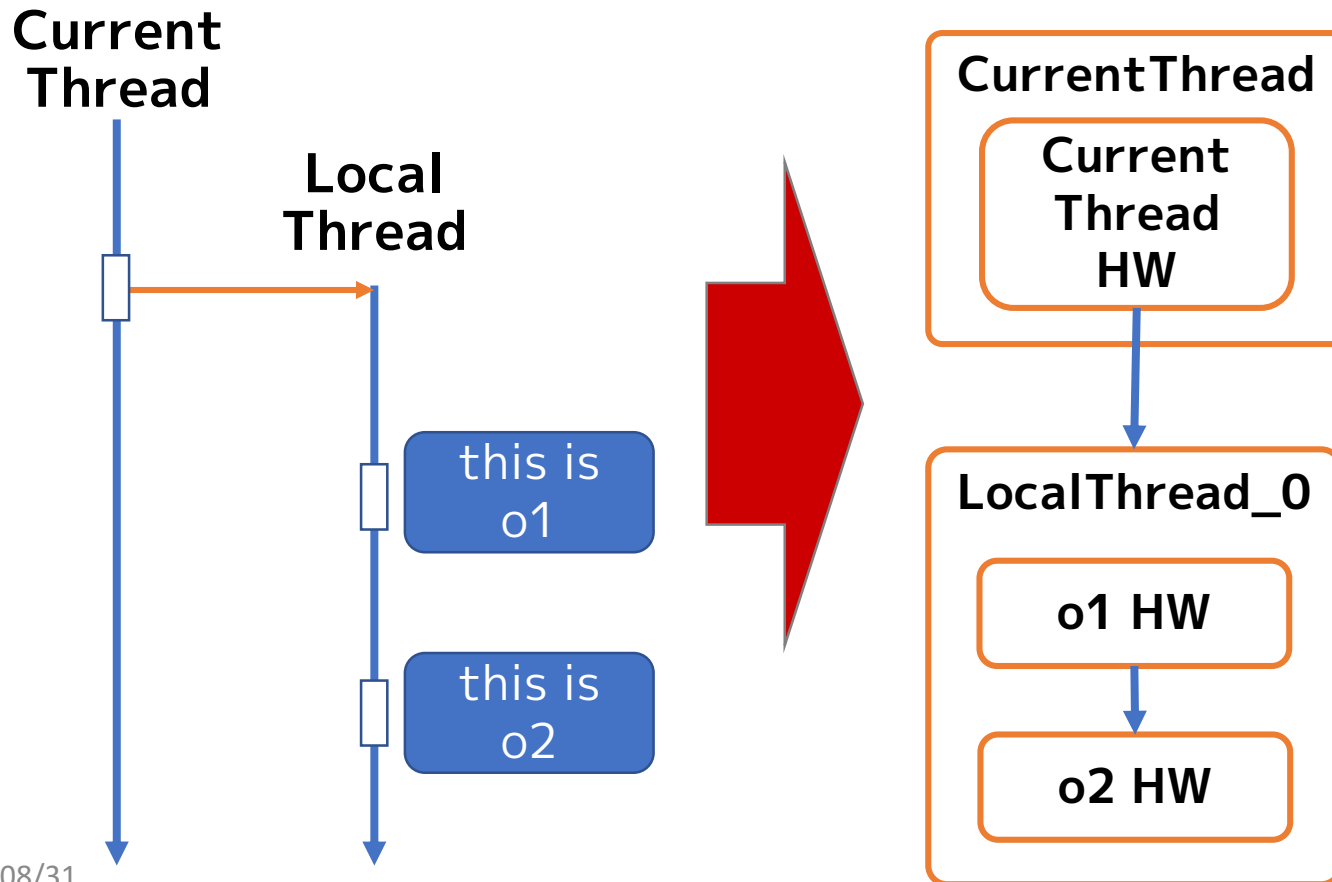
アクセサ
リ



- ✓ 1. HW抽出機構
- ✓ 2. HW合成機構
- ✓ 3. ラムダ抽象の合成
- 4. Schedulerの合成
- 5. ソフト/ハード連携

Schedulerの合成

- Thread毎にステートマシンを持つ
- Current Threadはかならずできる



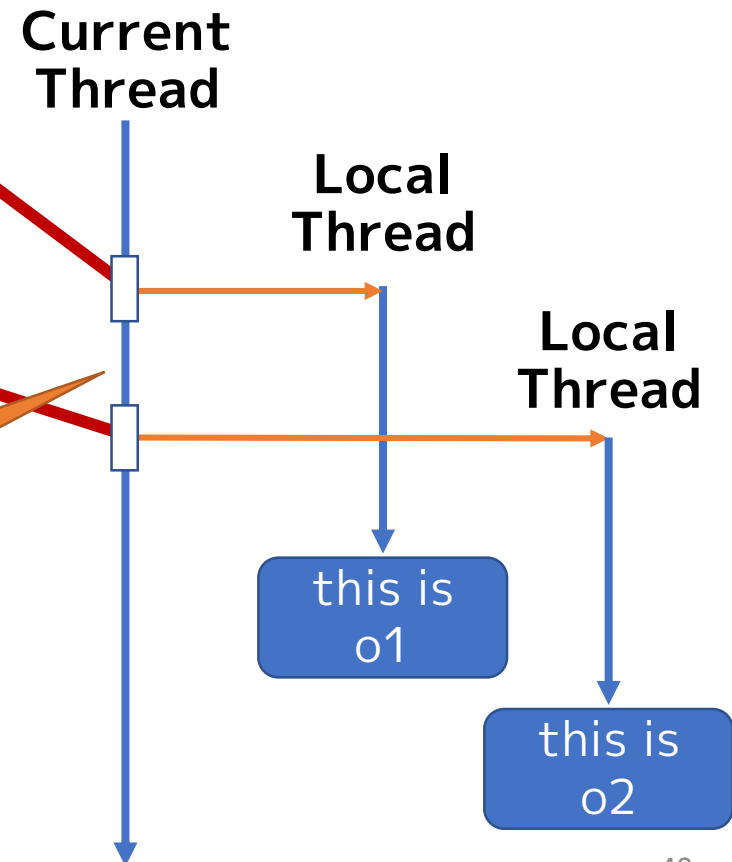
Schedulerの合成の問題

- 複数のThreadを同時にキックしたい場合
(未解決)

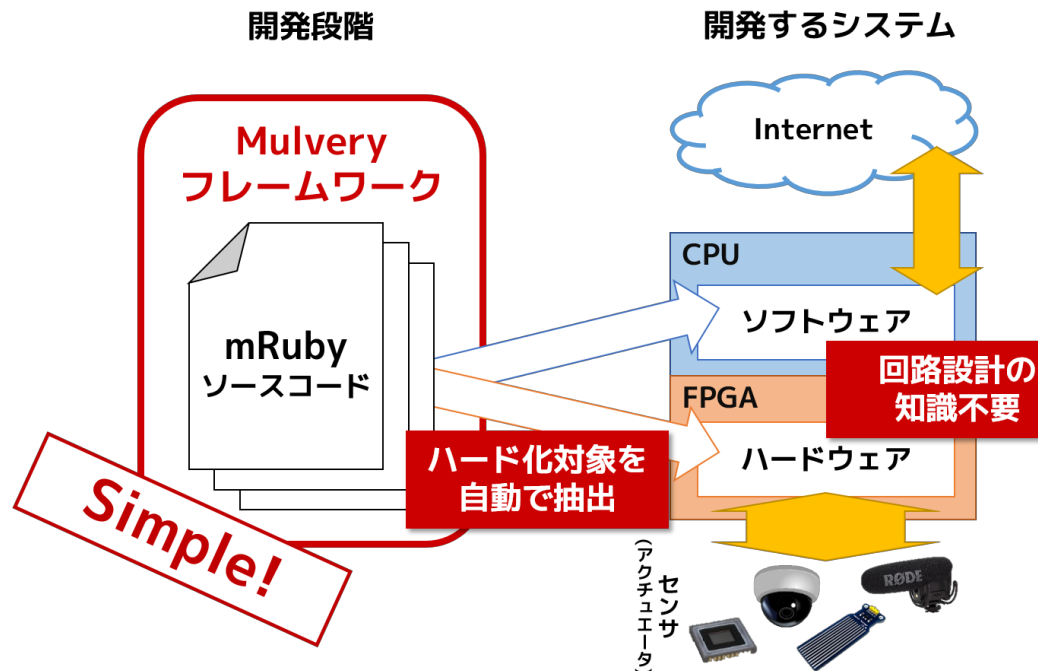
```
o1 = Observable.timer(1) do  
  p "this is o1"  
end
```

```
o2 = Observable.timer(1) do  
  p "this is o2"  
end
```

違うstateで
キックしてしまう



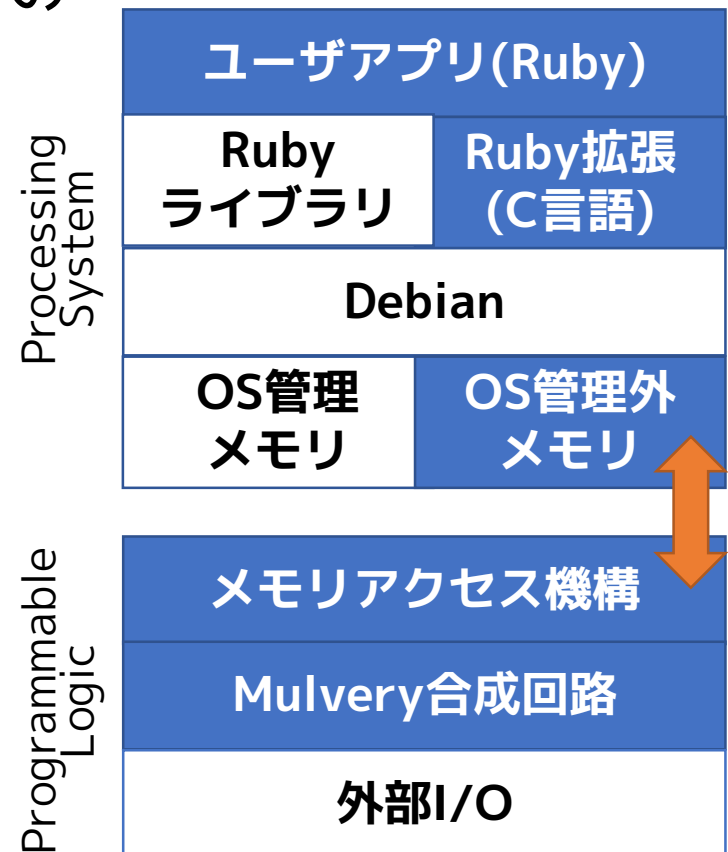
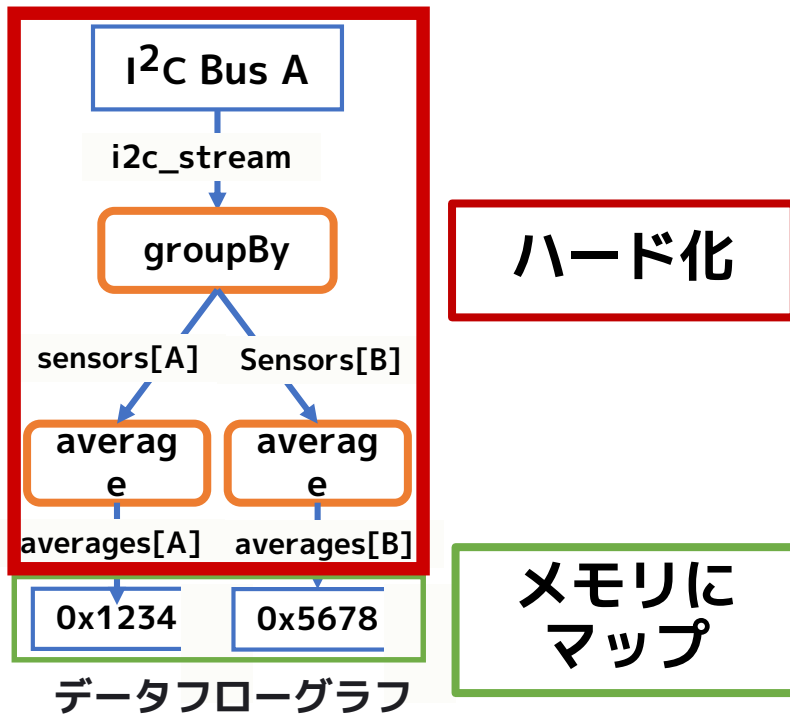
実装しなければならないもの



- ✓ 1. HW抽出機構
- ✓ 2. HW合成機構
- ✓ 3. ラムダ抽象の合成
- ✓ 4. Schedulerの合成
- 5. ソフト/ハード連携

ソフト/ハード自動連携

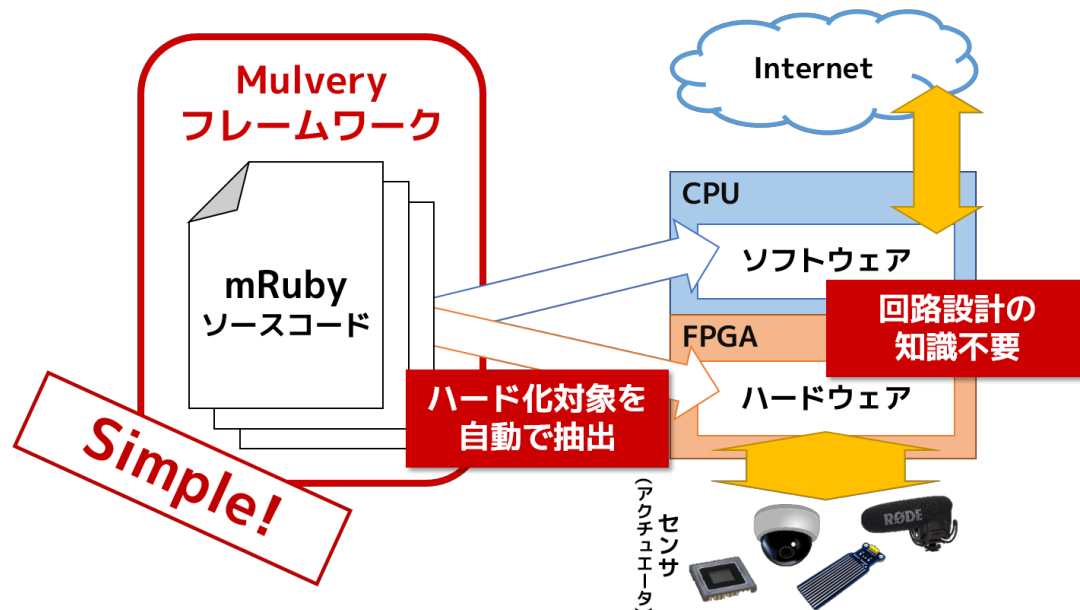
- 現状はかなり素朴な実装
- 共有メモリでデータ共有のみ



実装しなければならないもの

開発段階

開発するシステム



- ✓ 1. HW抽出機構
- ✓ 2. HW合成機構
- ✓ 3. ラムダ抽象の合成
- ✓ 4. Schedulerの合成
- ✓ 5. ソフト/ハード連携

Simple!

Mulveryの 楽しいところ

透過性の高い記述

```
def func(data)
```

```
  ...
```

```
end
```

```
stream_1 = Stream.new()
```

```
grouped_stream_1 = stream_1.group_by{|d| func(d)}
```

```
array_1 = [1, 2, 3, 4, 5, 6]
```

```
grouped_array_1 = array_1.group_by{|d| func(d)}
```

Stream :
FPGAで実行

非Stream :
CPUで実行

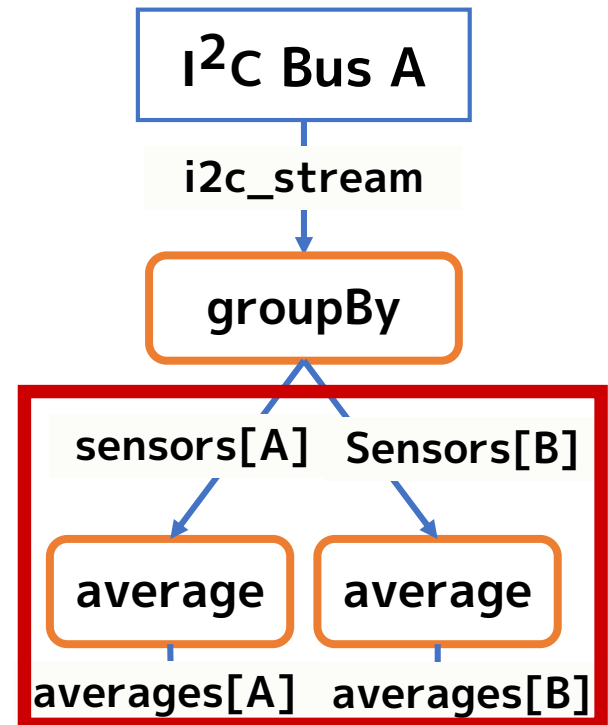
データに近い場所で処理が行われる
透過性の高い記述

透過性の高い記述の例 2

```
i2c_stream = Stream.new(I2C_BUS_A)
sensors = i2c_stream.group_by(2)
  { |d| d.sensor_id }
```

```
averages = sensors.map { |key, val|
  [key, sensor.average(5)]
}.to_h()
```

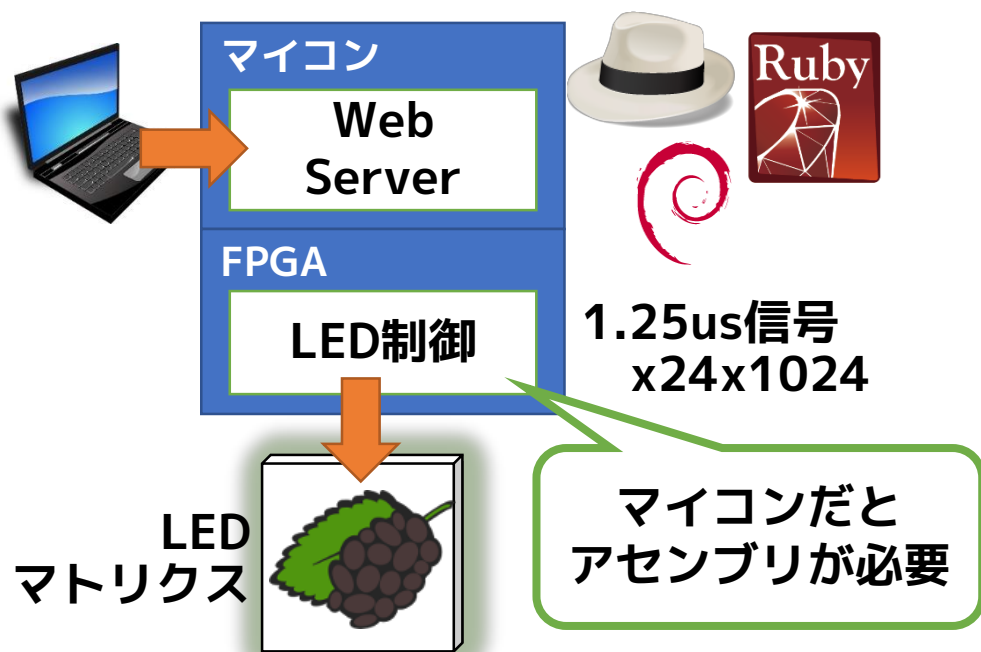
⇒ ハードにもソフトにもならないmap



同じメソッドが「ソフトウェア記述」
「ハードウェア記述」「generator文」の3役を果たす

デバイスコントローラとしての活用

1024個のフルカラーLED(NeoPixel)を制御してみる



Rubyの資産を使って
システム実装できる



余談

なぜRubyなのか

Rubyの方が開発が素早そうだったから

Python

```
events = Stream.from_pin(MIO::p0)
t = events.map(lambda event: 1)
count = t.scan(lambda acc, x: acc + x)
```

- ・ Method chainがしにくい
 - ・ ブロックが渡せない
- ⇒ **ちょっとイマイチ**

VS

Ruby

```
events = Stream.from_pin(MIO::p0)
count = events
  .map(){ |event| 1 }
  .scan(){ |acc, x| acc + x }
```

- ・ Method chainできる
 - ・ ラムダ抽象をブロックで
- ⇒ **いいかんじ!**

ML/DL向けに今後はPythonにも展開していきたい

自動テストの話

• RSpec+自作テスト on Travis CI

