하드웨어도 소프트웨어처럼 짜야한다

소프트웨어재난연구센터
2023 여름 워크샵 (2023/07/07)

강지훈 (KAIST 전산학부)

KAIST

KAIST 전산학부 동시성 및 병렬성 연구실

- **현황**
  - 기간: 2019/02 – 현재
  - 수업: CS220 프로그래밍의 이해, CS420 컴파일러, CS431 동시성
  - 인원: 1(조교수과정) + 2(박사과정) + 7(석사과정) + ?(학사과정)
  - 웹사이트: https://cp.kaist.ac.kr/
- **연구**
  - 대상: 시스템 (OS, archi, …) 내 동시성 및 병렬성
  - 도구: 어려운걸 누구나 이해할 수 있도록 추상화 (PL)

병렬성: 자원이 많은 것

- **순차 프로그래밍: (병렬) 자원이 하나인 "것처럼"**
  - 비효율적 (2005년: Dennard scaling 끝)
- **병렬 프로그래밍: 병렬 자원을 명확하게 인식 (예: 멀티 스레드)**

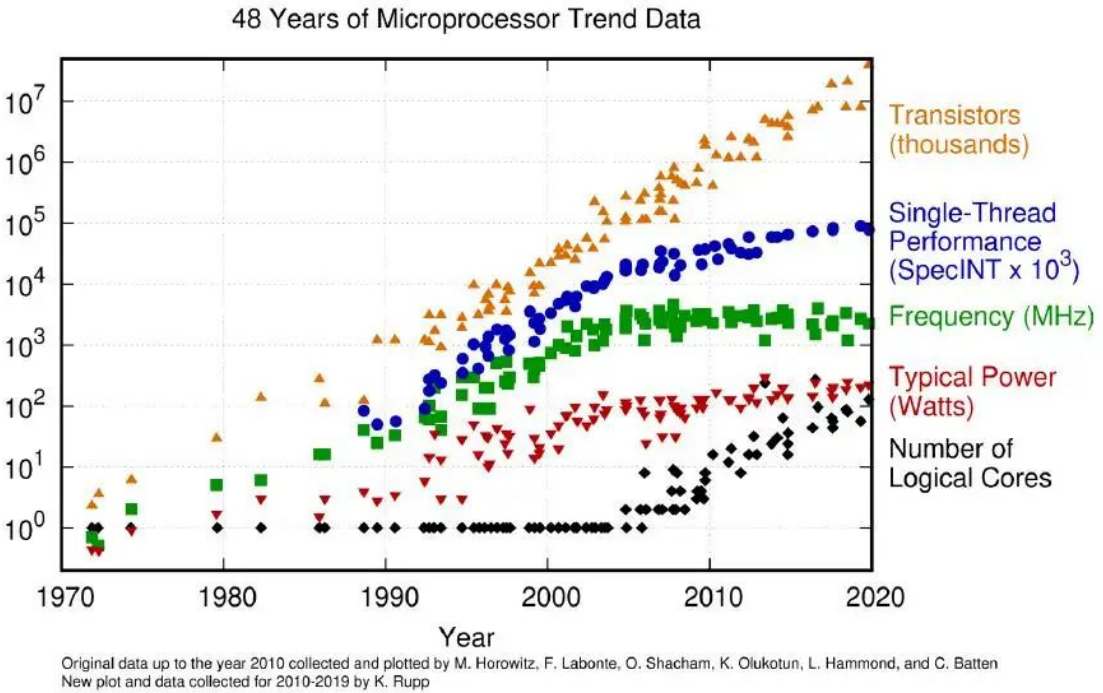


Figure 3. 48 Years of Microprocessor Trend Data. Source: K. Rupp.

동시성: 병렬 자원을 동시에

- **동시성: 여러 자원(멀티 스레드)이 같은 자원에(메모리) 동시 접근**

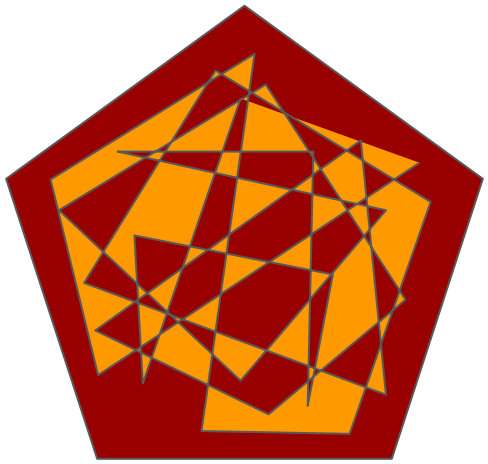
- **어려움 1: 모듈성 저하 (다른 자원의 불의의 습격)**
  - 예 1: data race, use-after-free, instruction reordering, …
  - 예 2: deadlock, livelock, …

- **어려움 2: 복잡한 동기화**
  - 이유: 공격적인 최적화 (불의의 습격**에도 불구**하고 올바른)

우리의 목표: 병렬 프로그래밍을 쉽게

- **추상화(PL): 복잡한 세부사항을 감추는 간단한 언어/라이브러리**
  - 목표: 순차 프로그래밍과 비슷한 난이도로 병렬 프로그래밍

- **병렬성 추상화 디자인**
  - 병렬성의 어려움인 동시성을 정확하게 포착
  - 프로그래머가 동시성 모르고도 병렬성 누릴 수 있게



- **병렬성 추상화 검증**
  - 목표: 수학적으로 엄밀하게 증명 (Coq)
  - 경험: 엄밀하게 증명 안하면 대체로 틀림… (POPL 2017, PLDI 2017, …)

병렬 프로그램 디자인 및 검증 프로젝트

- **영속성 메모리** (PLDI 2023, PLDI 2021)
- **하드웨어** (ASPLOS 2023)
- **메모리 재활용** (OOPSLA 2023, SPAA 2023, PLDI 2020)
- **약한 메모리** (PLDI 2022, PLDI 2019)
- **컴파일러 검증** (POPL 2022, POPL 2020 × 2)

좀 더 자세히

KAIST 전산학부 동시성 및 병렬성 연구실 사람들

Jeehoon Kang
Principal Investigator

Jaemin Choi
Master Student

Jung In Rhee
Master Student

Sunho Park
Undergraduate Student

Kyeongmin Cho
PhD Student

Seungmin Jeon
Master Student

Haechan An
Master Student

Jaeyong Sung
Undergraduate Student

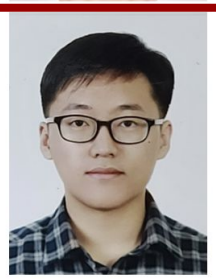
Jaehwang Jung
PhD Student

Janggun Lee
Master Student

Taewoo Kim
Undergraduate Student

Jaewoo Kim
Undergraduate Student

Shuangshuang Zhao
PhD Student



Minseong Jang
Master Student

Woojin Lee
Undergraduate Student

Jeonghyeon Kim
Undergraduate Student

KAIST

하드웨어도
소프트웨어처럼
짜야한다

하드웨어 가속기 등장

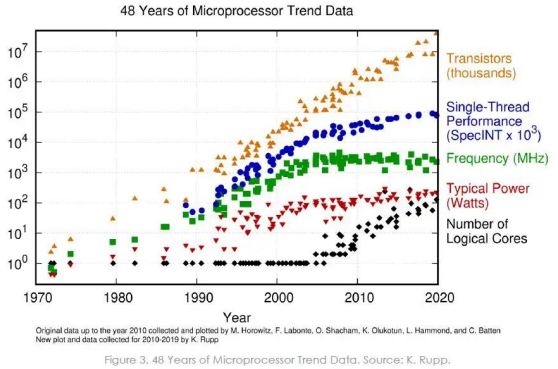
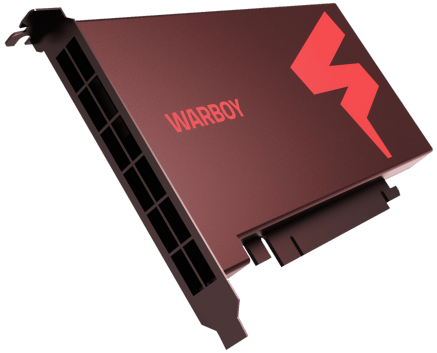
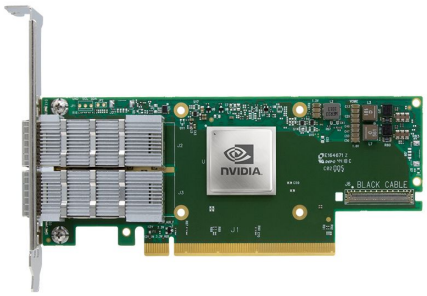


Figure 3. 48 Years of Microprocessor Trend Data. Source: K. Rupp.

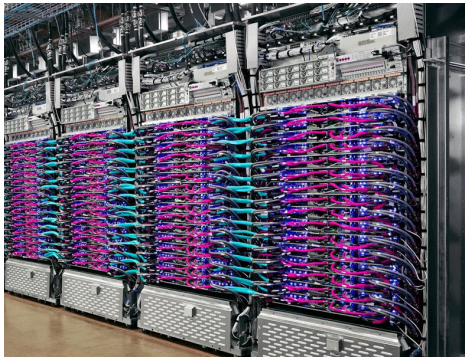
- **무어의 법칙, 데나드 스케일링 시대의 끝**
  - 성능 향상을 위한 **응용 마춤 컴퓨팅 (GPU, FPGA, ASIC)**
  - 다양한 application 개발을 위한 **높은 생산성** 필요



FuriosaAI
(NPU)



NVIDIA
(DPU)


Google
(TPU)

하드웨어 설계 방법론의 변화

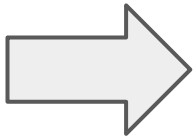
이게 뭐꼬?

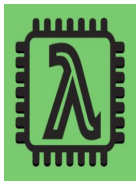
현대 HDL

저수준 HDL

Verilog

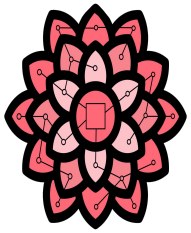
VHDL





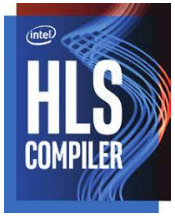
Clash

SpinalHDL



Dahlia

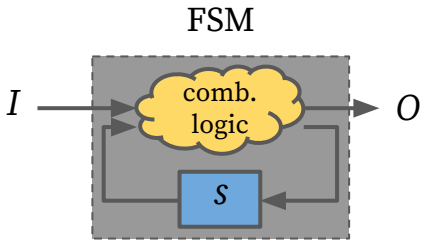
고수준 합성



Catapult

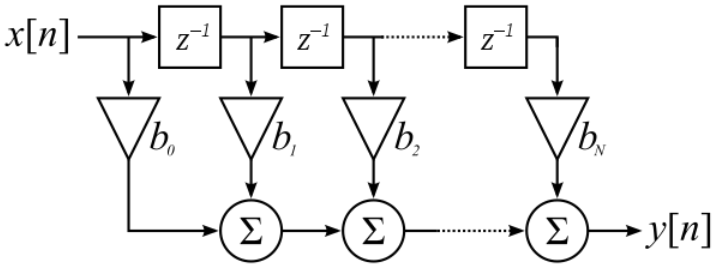
KAIST

기존 하드웨어 설계 방법론



- **하드웨어 기술 언어 (HDL)**
  - 하드웨어 가속기의 내부 로직은 **유한 상태 기계 (FSM)**
  - FSM을 표현하기 위한 언어가 HDL

- **예시: FIR Filter**

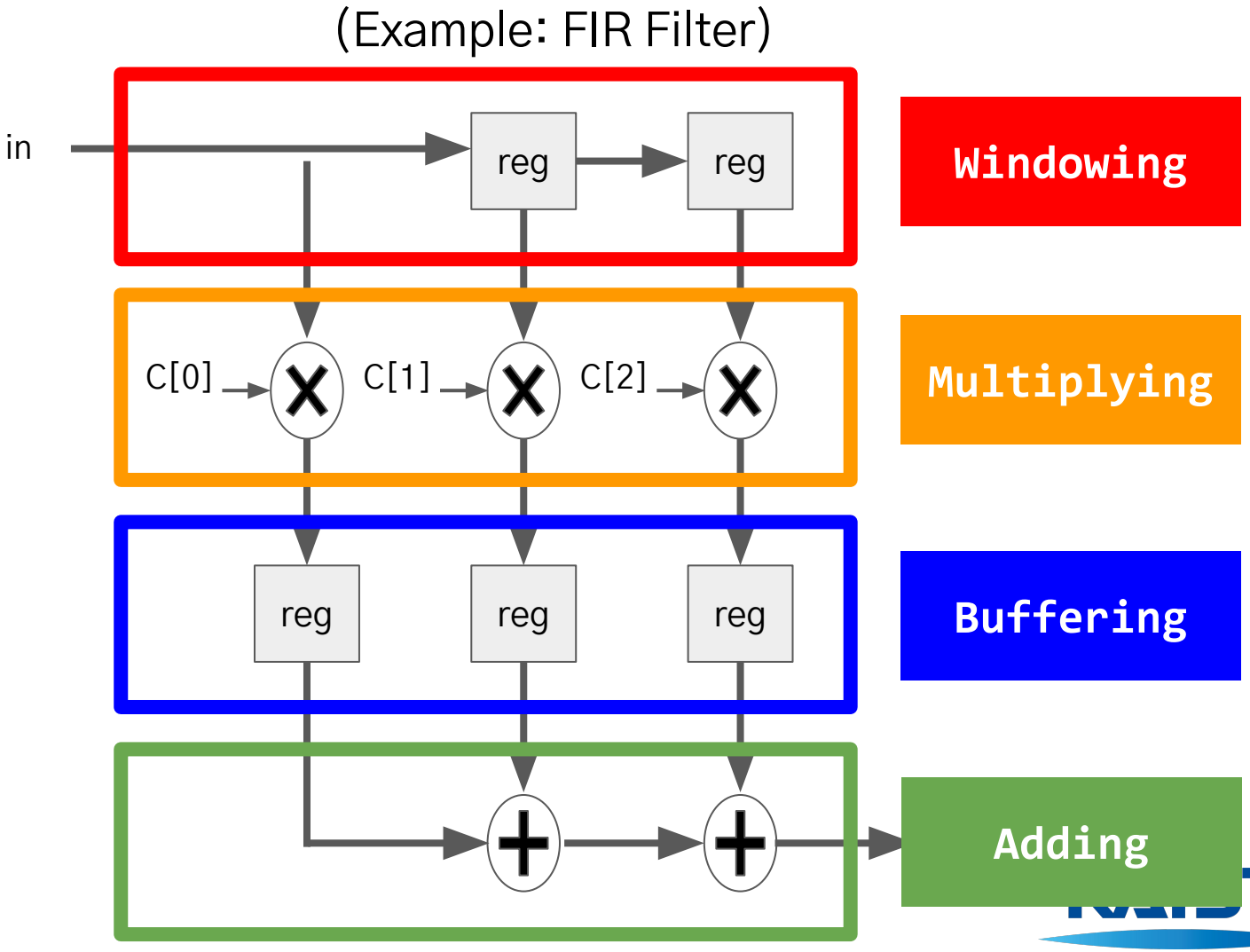


회로의 모양을 기술하는 방식
- 각 **cycle**의 행동을 **통짜로** 기술
- 병렬성을 **register/wire 수준에서** 기술

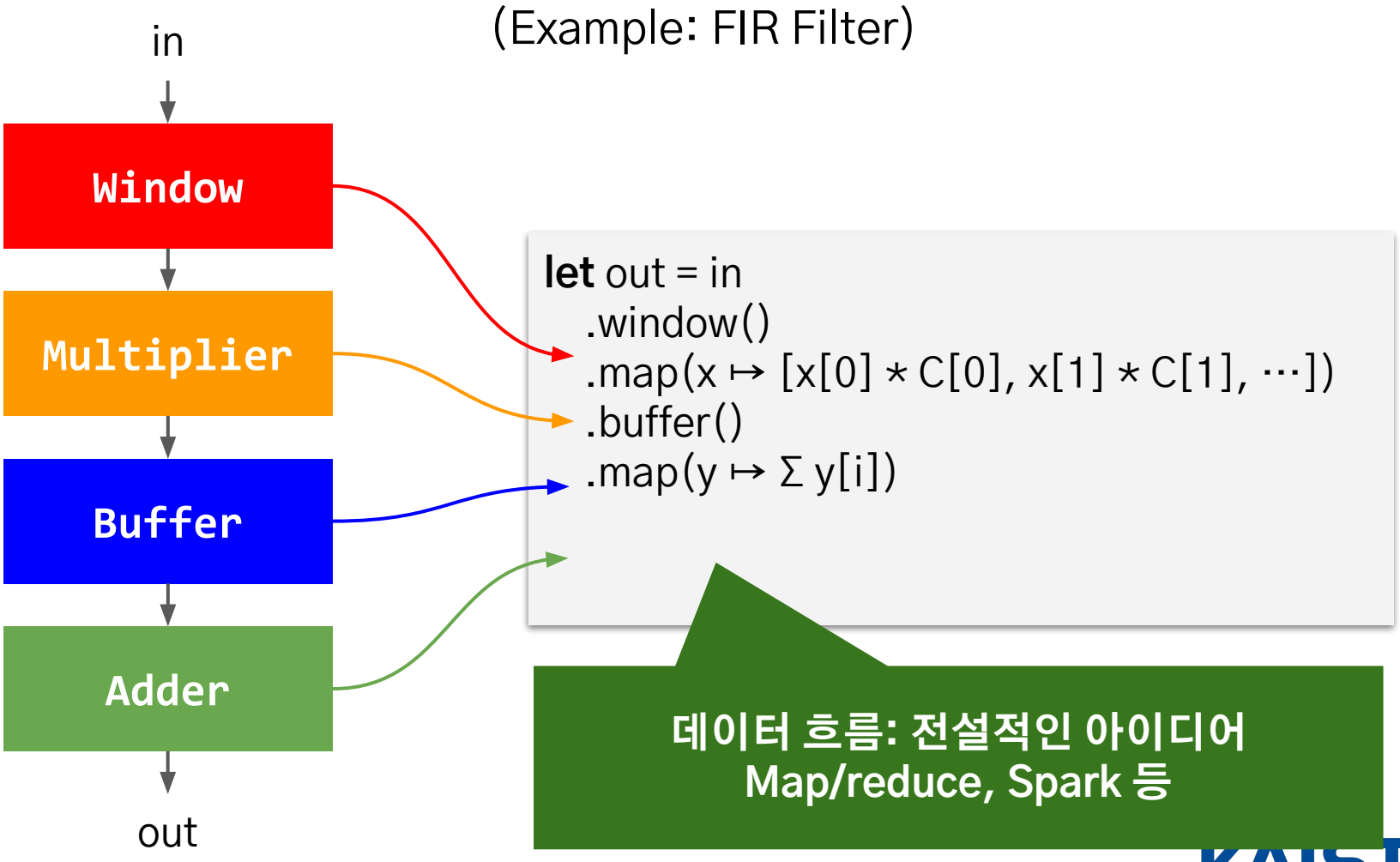
```
always_ff @(posedge clk) begin
  if (reset) begin
    for(int i=0; i<41; i++) begin
      reg_x[i] <= {16'b0}; reg_y[i] <= {32'b0};
    end
    acc<=0;
  end
  else
    for (i=40; i>0; i=i-1) begin
      reg_x[i] <= reg_x[i-1];
    end
    reg_x[0] <= din;

   acc <= 0;
    for(int i=0; i<41; i++) begin
      acc <= acc + reg_x[i]*coeff[i];
    end
  end
```

관찰: 잘 쪼개짐


(Example: FIR Filter)
in
reg
reg
C[0]
C[1]
C[2]
reg
reg
reg
Windowing
Multiplying
Buffering
Adding

관찰: 쪼갠 것들끼리 데이터 흐름



데이터 흐름 스타일 하드웨어 설계 (1980년대부터)

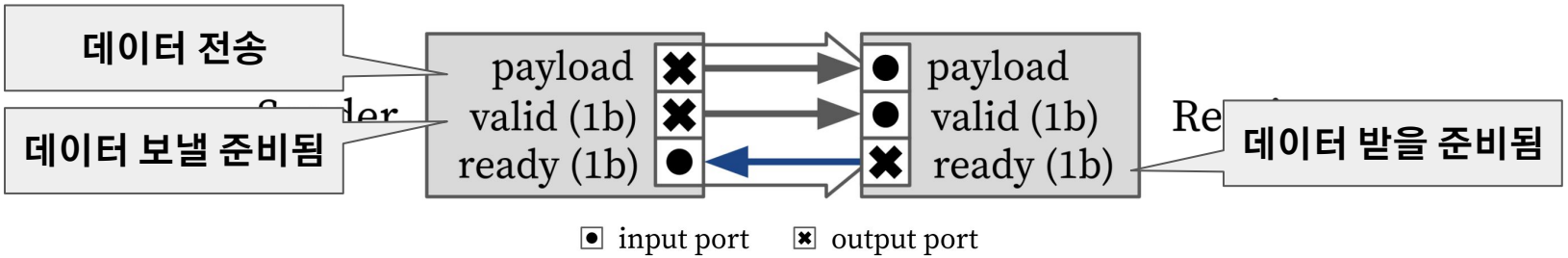
```
let out = in
    .window()
    .map(|x| x.zip(coeff).map(mul))
    .buffer()
    .map(|x| x.sum());
```

- 회로의 병렬성/동시성을 "조립기"로 길들임 (cf. map/reduce, Spark)
- 통짜가 아닌 조립식
- 자원이 아닌 개념 추상화

- FPCA (1981-1995): functional programming + computer architecture
  - 후에 ICFP로 변경 (CS Top)

데이터 흐름이 만난 도전: 지연시간 무관 전송 규약

- **목표:** 모듈간 안정적으로 데이터 주고받기
- **예제:** Valid/Ready 프로토콜



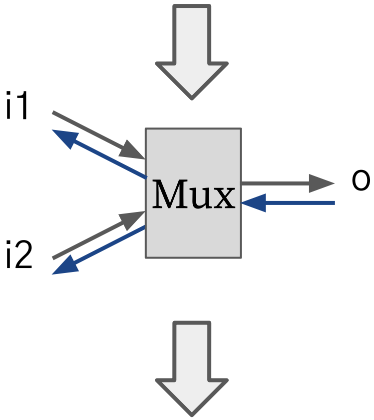
out 쪽에서 받을 준비가 안됐다

```
let out = in
    .window()
    .map(|x| x.zip(coeff).map(mul).sum());
```

window/map에서 어떻게 알지?

해결책: ShakeFlow (ASPLOS 2023)

```
let o = (i1, i2).mux();
```



Map/reduce 스타일 **흐름 조합기**

기존

```
let o.fwd = (i1.fwd, i2.fwd).mux();
let i1.ready = o.ready;
let i2.ready = o.ready & !i1.fwd.valid;
```

지연시간 무관 전송 규약 지원

(전송 규약 조합기)

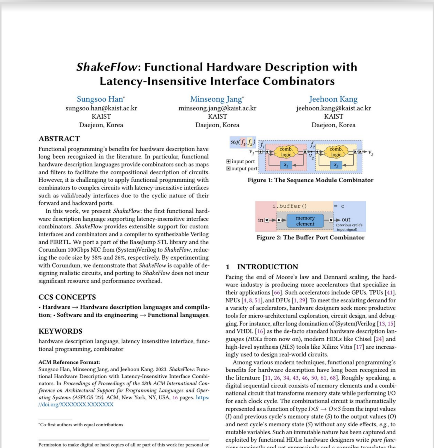
우리

```
let o = (i1, i2).mux();
```

Verilog FIRRTL

Verilog, FIRRTL로 컴파일

생산성 증가, 성능 하락 ✗



ShakeFlow: Functional Hardware Description with Latency-Insensitive Interface Combinators

Sungsoo Han* Minseong Jang* Jeehoon Kang
KAIST KAIST KAIST
Daejeon, Korea Daejeon, Korea Daejeon, Korea

ABSTRACT

1 INTRODUCTION

CCS CONCEPTS

KEYWORDS

전송 규약 조합기: 큰그림

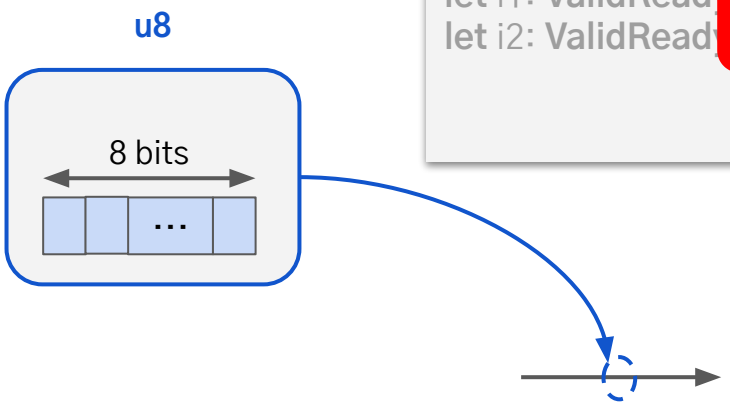
```
// Example: transforming payload (XOR 42)
let i1: ValidReady<u8> = ⋯;
let i2: ValidReady<u8> = i1.map(v ↦ v ^ 42);
```

KAIST

전송 규약 조합기: 큰그림

(1) 마춤 신호 타입

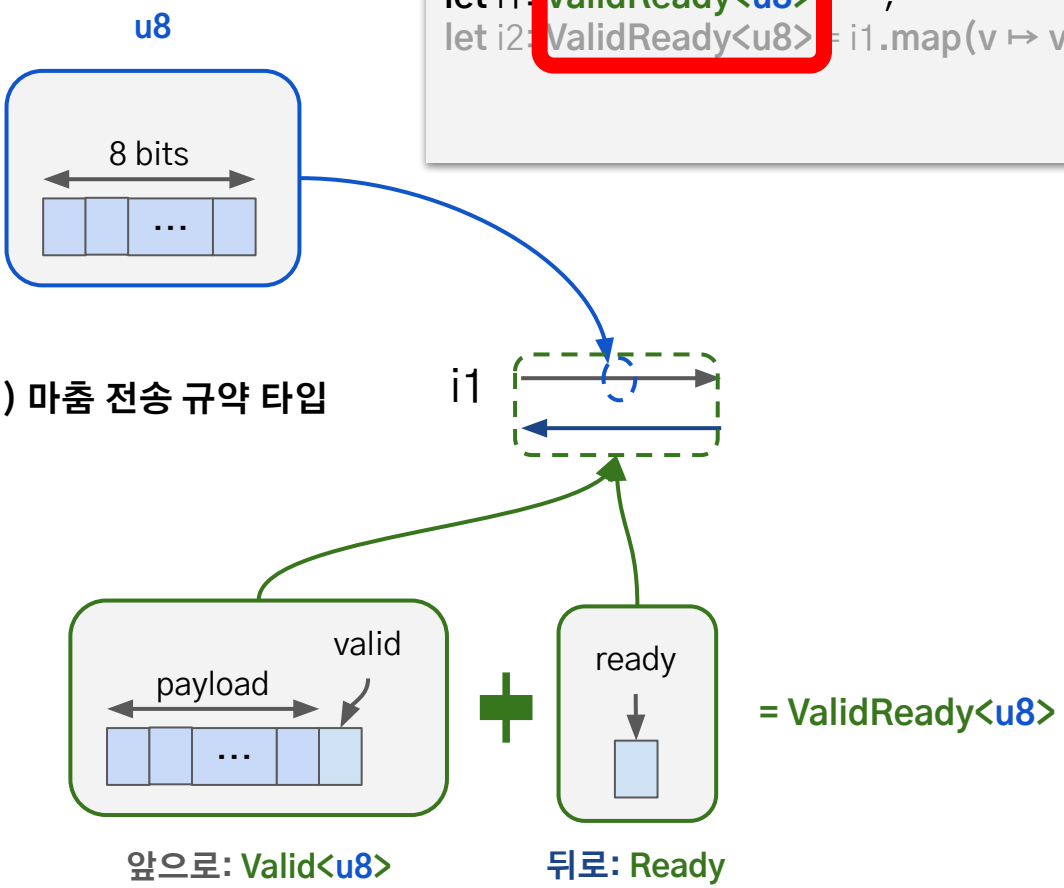
```
// Example: transforming payload (XOR 42)
let i1: ValidReady<u8> = ⋯;
let i2: ValidReady<u8> = i1.map(v ↦ v ^ 42);
```



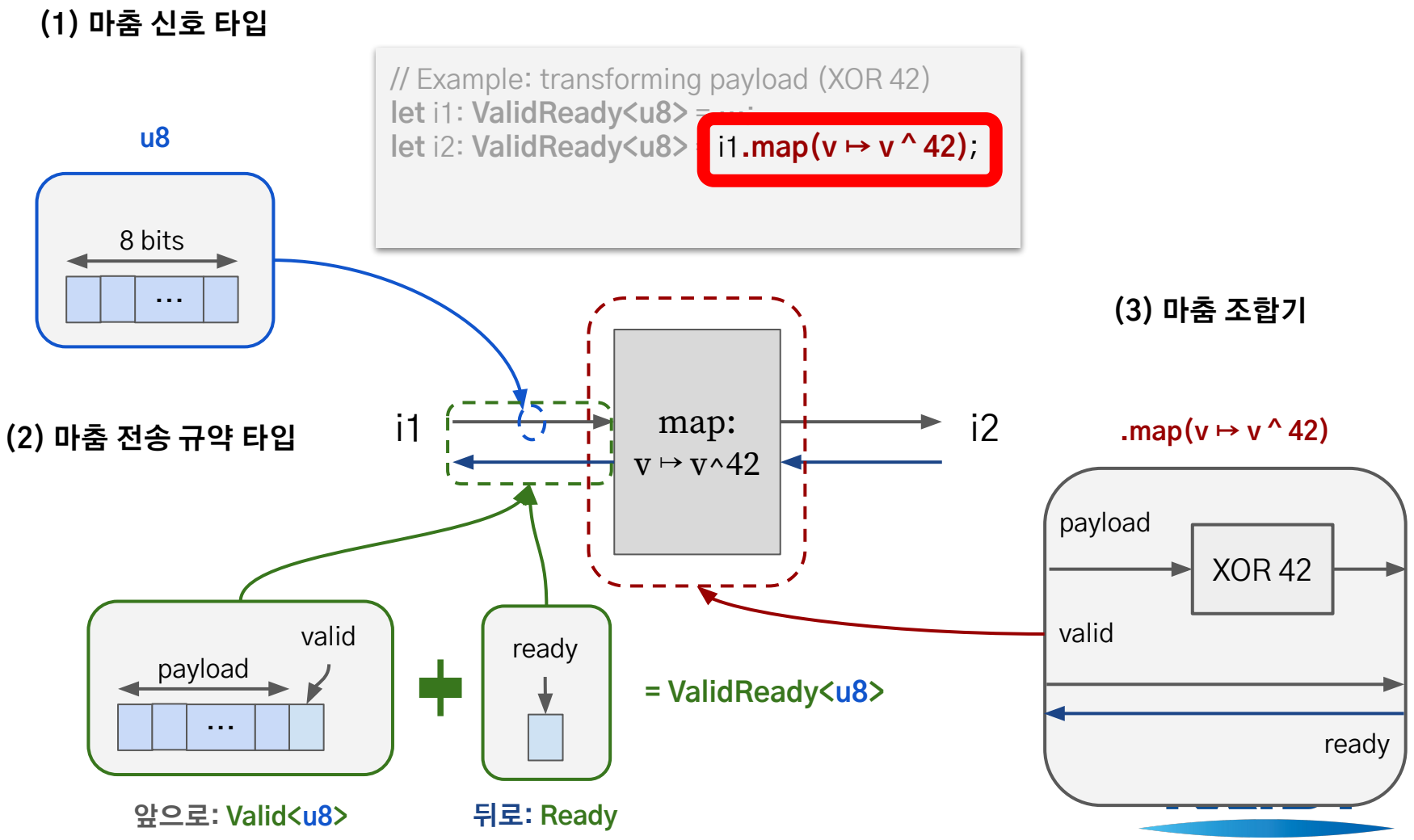
전송 규약 조합기: 큰그림

(1) 마춤 신호 타입

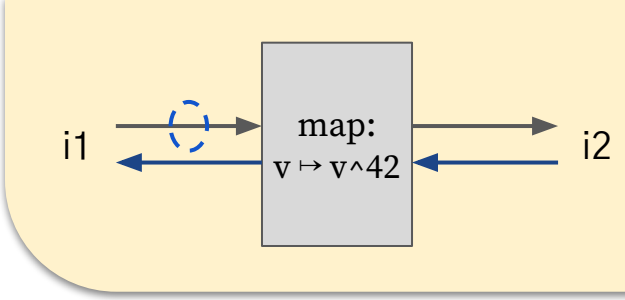
```
// Example: transforming payload (XOR 42)
let i1: ValidReady<u8> = ⋯;
let i2: ValidReady<u8> = i1.map(v ↦ v ^ 42);
```



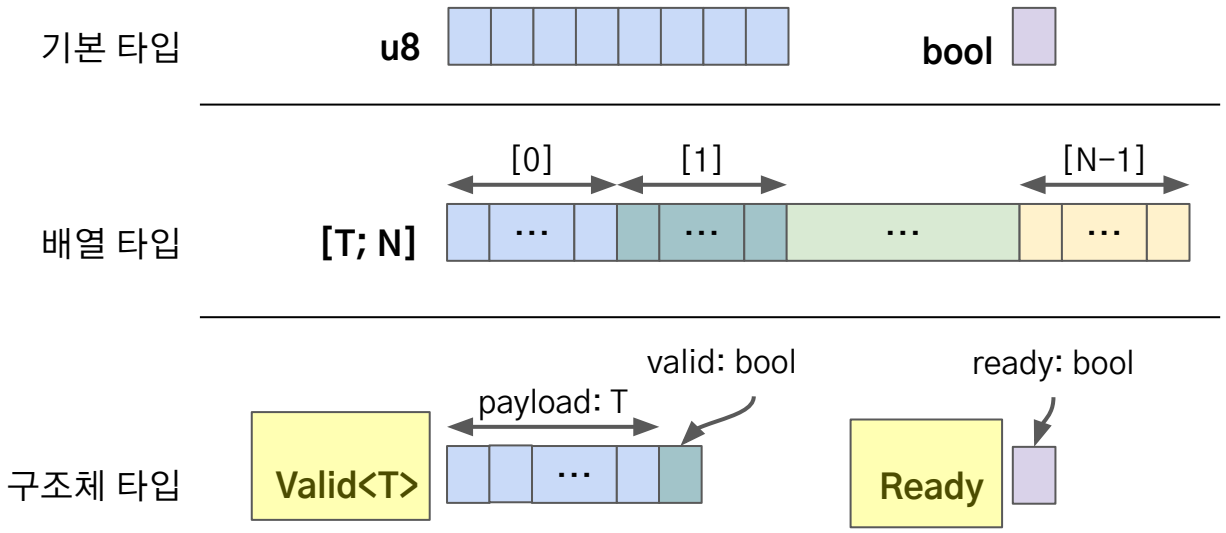
전송 규약 조합기: 큰그림


(1) 마춤 신호 타입
// Example: transforming payload (XOR 42)
let i1: ValidReady<u8> = ...;
let i2: ValidReady<u8> = i1.map(v ↦ v ^ 42);
u8
8 bits
...
(2) 마춤 전송 규약 타입
i1
map:
v ↦ v^42
i2
(3) 마춤 조합기
.map(v ↦ v ^ 42)
payload
XOR 42
valid
ready
valid
payload
...
ready
= ValidReady<u8>
앞으로: Valid<u8>
뒤로: Ready

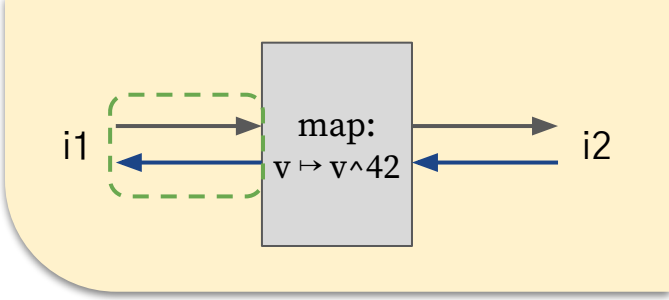
(1) 마춤 신호 타입



신호: 전깃줄을 타고 다니는 값 (비트 덩어리)

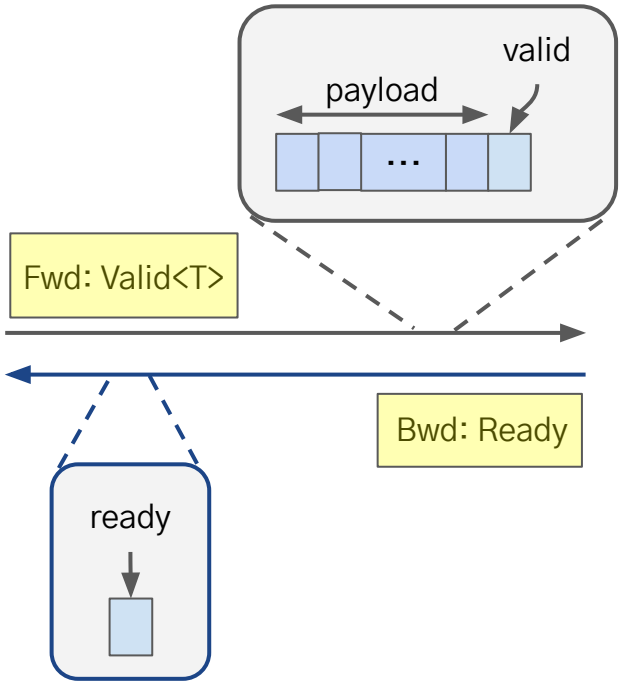


(2) 마춤 전송 규약 타입

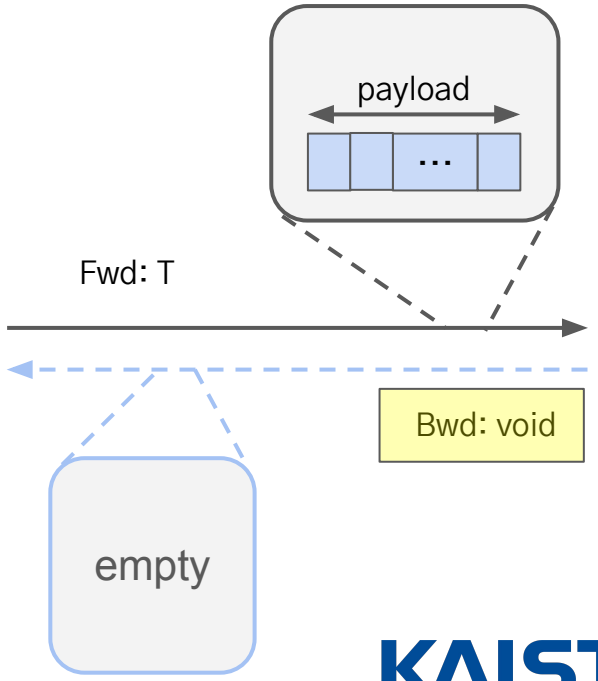


전송 규약: 앞으로 가는 전깃줄과 뒤로 가는 전깃줄

지연시간 **무관** 전송 규약 (`ValidReady<T>`)

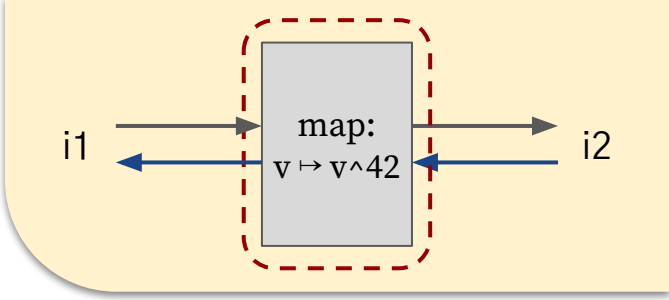


지연시간 **유관** 전송 규약 (`Port<T>`)

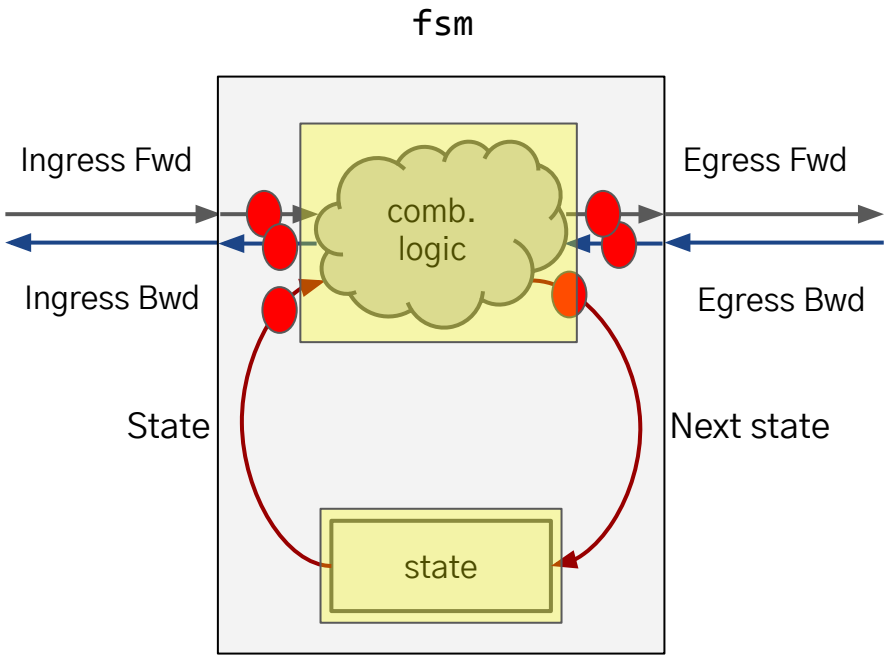


KAIST

(3) 마춤 조합기는 유한상태기계

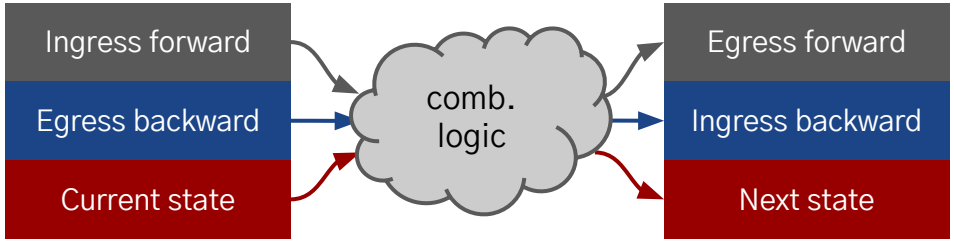


유한상태기계 (FSM): (입력 * 상태) -> (출력 * 상태)



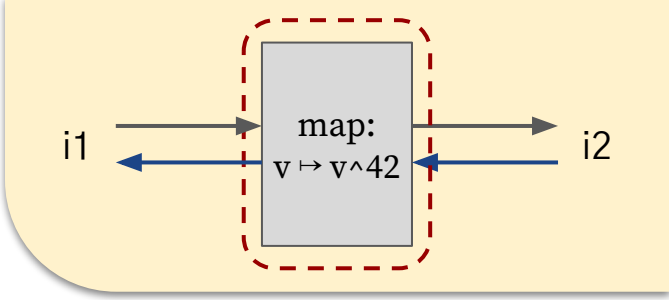

```
let ingress: I = ⋯;
let egress: E = ingress.fsm(init_state, comb_logic);
```

`comb_logic`: `fsm`의 조합 논리 함수 (pure)



KAIST

(3) 조합기 라이브러리 예제



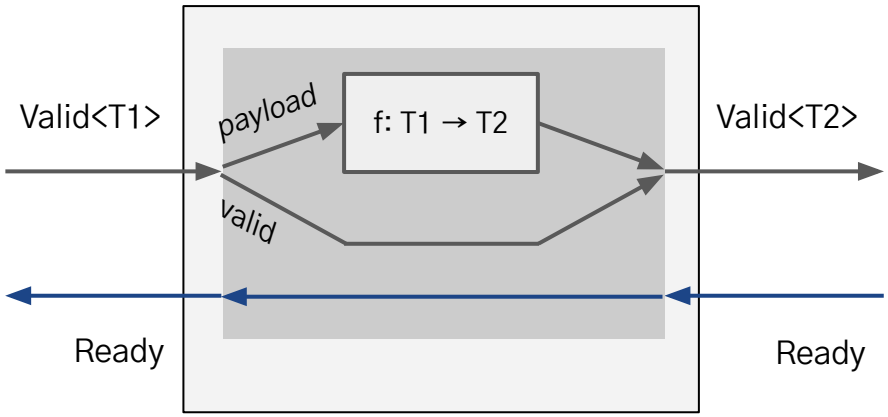
map: payload 신호를 주어진 함수로 변환

```
let ingress: ValidReady<T1> = ⋯;
let egress: ValidReady<T2> = ingress.map(v ↦ v ^ 42);
```

Implemented as

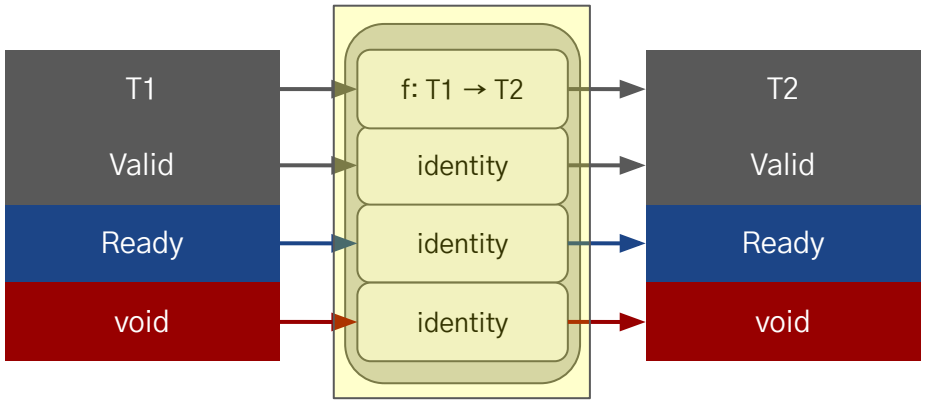
```
.fsm(void, map_comb(v ↦ v ^ 42));
```

map을 fsm으로 구현



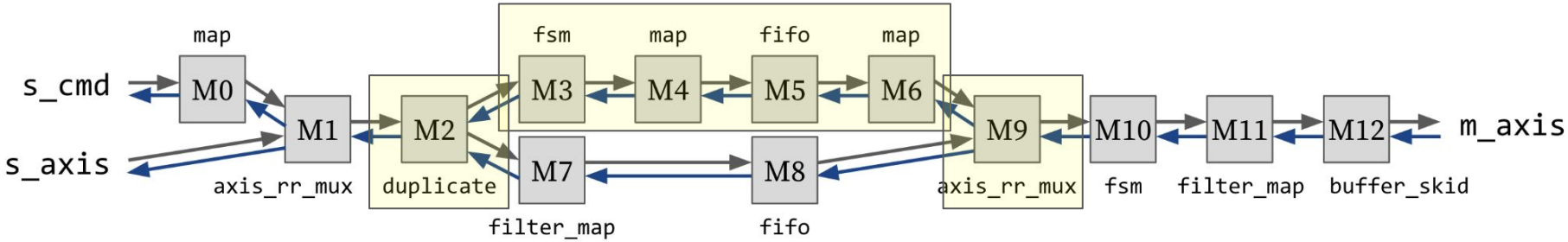
(상태는 비었음)

map_comb(f): map의 조합 논리 함수



KAIST

예제: tx_checksum (Corundum 100Gbps NIC)



Verilog

Monolithic,
requiring manual plumbing

```
        if (sum_last_reg[LEVELS-2]) begin
            if (sum_odd_reg[LEVELS-2]) begin
                csum_in_csum_reg[7:0] <= ~sum_acc_temp[15:8];
                csum_in_csum_reg[15:8] <= ~sum_acc_temp[7:0];
            end else begin
                csum_in_csum_reg[7:0] <= ~sum_acc_temp[7:0];
                csum_in_csum_reg[15:8] <= ~sum_acc_temp[15:8];
            end
            csum_in_offset_reg <= sum_offset_reg[LEVELS-2];
            csum_in_enable_reg <= sum_enable_reg[LEVELS-2];
            csum_in_valid_reg <= 1'b1;
            sum_acc_reg <= 0;
        end else begin
            sum_acc_reg <= sum_acc_temp;
        end
    end

    if (rst) begin
        csum_in_valid_reg <= 1'b0;
    end
end
  // checksum FIFO
  axis_fifo #(
      .DEPTH(CHECKSUM_FIFO_DEPTH),
  )
  csum_fifo (
```

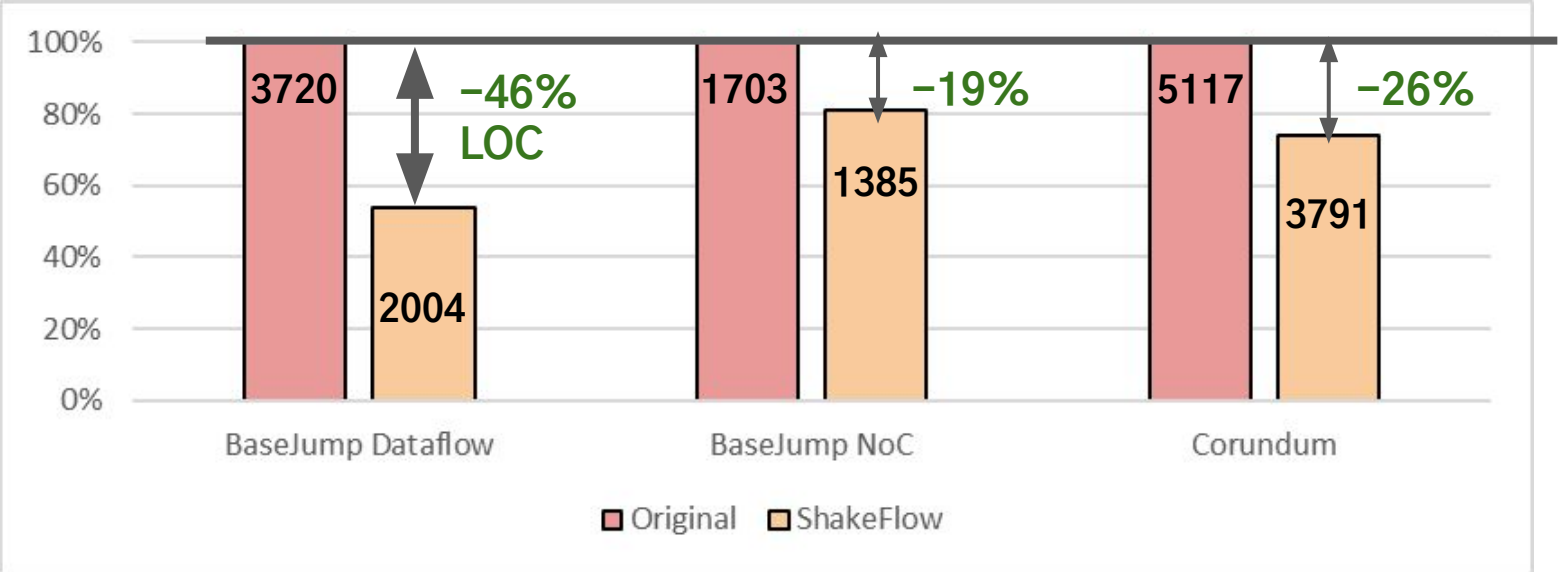
```
let (csum_in, data_in) = ….duplicate();          ← M2

let csum_out = csum_in
    .fsm(calc_csum_init, calc_csum)              ← M3
    .map(serialize)                              ← M4
    .fifo::<CHECKSUM_FIFO_DEPTH>()               ← M5
    .map(deserialize);                           ← M6

let data_out = data_in.…;                        ← M7-M8
(csum_out, data_out).axis_rr_mux();              ← M9
```

평가: 생산성 (LOC)

베이스라인 (작은게 좋음)



근데 솔직히 이렇게 평가하면 안됐다...

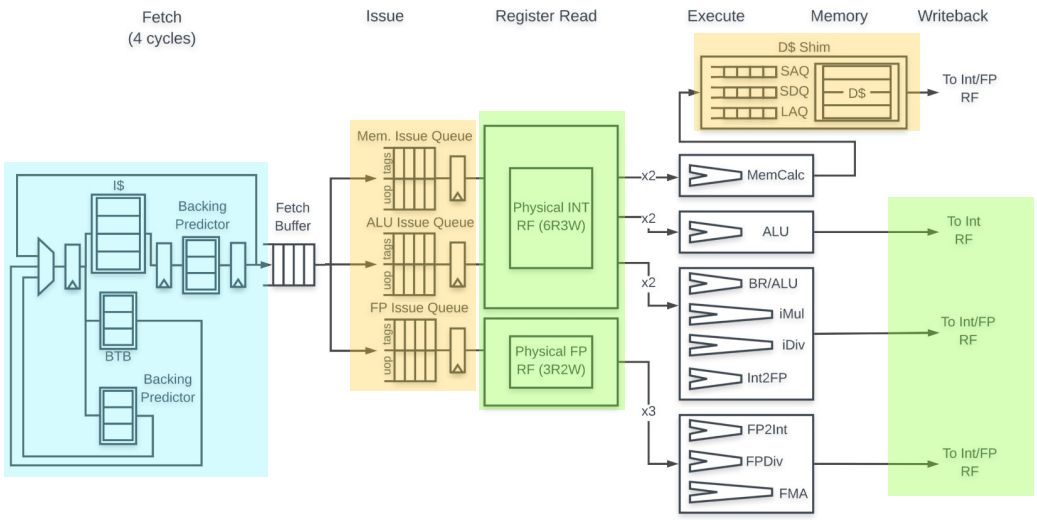
KAIST

하드웨어도
소프트웨어처럼
짜야한다

해… 해치웠나?!

진행중인 연구: CPU

- RISC-V BOOM CPU: 복잡한 제어 흐름의 끝판왕
  - **슈퍼스칼라**와 **비순차적 처리** 기능을 지원
  - **10단계**의 파이프라인 단계들로 구성
- ShakeFlow로 잘 표현 불가능



문제들

파이프라인 내 의존성 표현 불가

비효율적 지연시간 유관 인터페이스

비순차적 처리 표현 힘듦

아직 끝이 아니었다..

하드웨어도
소프트웨어처럼
짤 수 있다

몇년만 더 하면…

같은 테마로 연구실에서 하는 일

- **영속성 메모리** (PLDI 2023, PLDI 2021)
- **하드웨어** (ASPLOS 2023)
- **메모리 재활용** (OOPSLA 2023, SPAA 2023, PLDI 2020)
- **약한 메모리** (PLDI 2022, PLDI 2019)
- **컴파일러 검증** (POPL 2022, POPL 2020 × 2)

- **설계:** 양자 알고리즘 설계, 커널 최적화, AI 시스템 설계, 간헐적 컴퓨팅
- **검증:** 병렬 자료구조 검증, 시스템 검증, 검증 자동화

병렬성의 *끝없는* 도전

- **새로운 병렬 자원**
  - 계산: CPU (멀티코어), GPU, NPU (DNN 가속기), FPGA, …
  - 저장: 분산 메모리 시스템, 분산 스토리지 시스템, 영속성 메모리, CXL, …
  - 통신: 800Gbps, 프로그래밍 가능한 스위치, …
  - 하드웨어: 전깃줄/기억장치를 “소프트웨어 짜듯이” …
- **새로운 요구 조건에 따른 복잡한 동기화**
  - 저지연/고가용성/고성능 데이터센터 앱
  - 저지연/고연비 모바일 앱
  - 탄소인지형 컴퓨팅
- **새로운 대상**
  - 알고리즘 → 시스템

이론과 실제의 *끝없는* 만남

- **병렬성 추상화 디자인**
  - 병렬성의 어려움인 동시성을 정확하게 포착
  - 프로그래머가 동시성 모르고도 병렬성 누릴 수 있게
  - **프로그래머부터 자원까지 통합적 관점 필요**

- **병렬성 추상화 검증**
  - 목표: 수학적으로 엄밀하게 증명 (Coq)
  - 경험: **엄밀하게 증명 안하면 대체로 틀림**… (POPL 2017, PLDI 2017, …)

- **새로운 응용**
  - “숨쉬듯” 병렬 프로그래밍

감사합니다