# Implementing Microprocessors from Simplified Descriptions

Nikhil A. Patil, Derek Chiou

The University of Texas at Austin

Asia South Pacific Design Automation Conference
January 2013

# Simplify description of processor hardware

- Processors are becoming more and more complex
- Most complexity is added *on purpose*
- Better performance and power efficiency

Make processor hardware description simpler
without reducing the ability to specify inherent complexity

# What does *simplify* mean?

simplify: to make simple; reduce to basic essentials

# What does *simplify* mean?

simplify: to make simple; reduce to basic essentials

fig/breadboard-messy

fig/breadboard-clean

# What does *simplify* mean?

simplify: to make simple; reduce to basic essentials

structure of `rec` is entangled in the description

```
wire [96:0] rec;
wire [15:0] val;

assign val = rec[96] ? rec[95:80]
                     : rec[79:64];
```

```
typedef struct packed {
    bit align;
    bit [15:0] x;
    bit [15:0] y;
    bit [63:0] theta;
} rec_t;

rec_t rec;
wire [15:0] val;

assign val = rec.align ? rec.x
                       : rec.y;
```

# What does *simplify* mean?

simplify: to make simple; reduce to basic essentials

- Compiler generates packing
- Simpler to read/debug
- Easy to add new fields
- Avoid bugs accessing fields

```
typedef struct packed {
    bit align;
    bit [15:0] x;
    bit [15:0] y;
    bit [63:0] theta;
} rec_t;
```

```
wire [96:0] rec;
wire [15:0] val;

assign val = rec[96] ? rec[95:80]
                     : rec[79:64];
```

```
rec_t rec;
wire [15:0] val;

assign val = rec.align ? rec.x
                       : rec.y;
```

# Problem: disentangling functionality

Processor *design* clearly separates correctness and performance

- Instruction set: correctness
- Microarchitecture: power, performance

But processor *implementation* intimately entangles them

Make processor hardware description simpler
without reducing the ability to specify inherent complexity
by disentangling functionality from the description

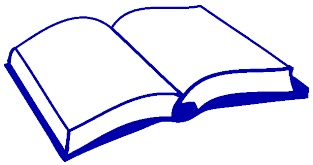
# Impact of disentangling functionality

- Compiler generates ISA-dependent logic:
  - Microcode table (control store)
  - Structure & encoding of control words
  - Logic controlled by microcode control bits
- Description becomes simpler to write, read and modify
- Avoid instruction implementation bugs
- Easy to add new instructions

# Toolflow

## Bluespec

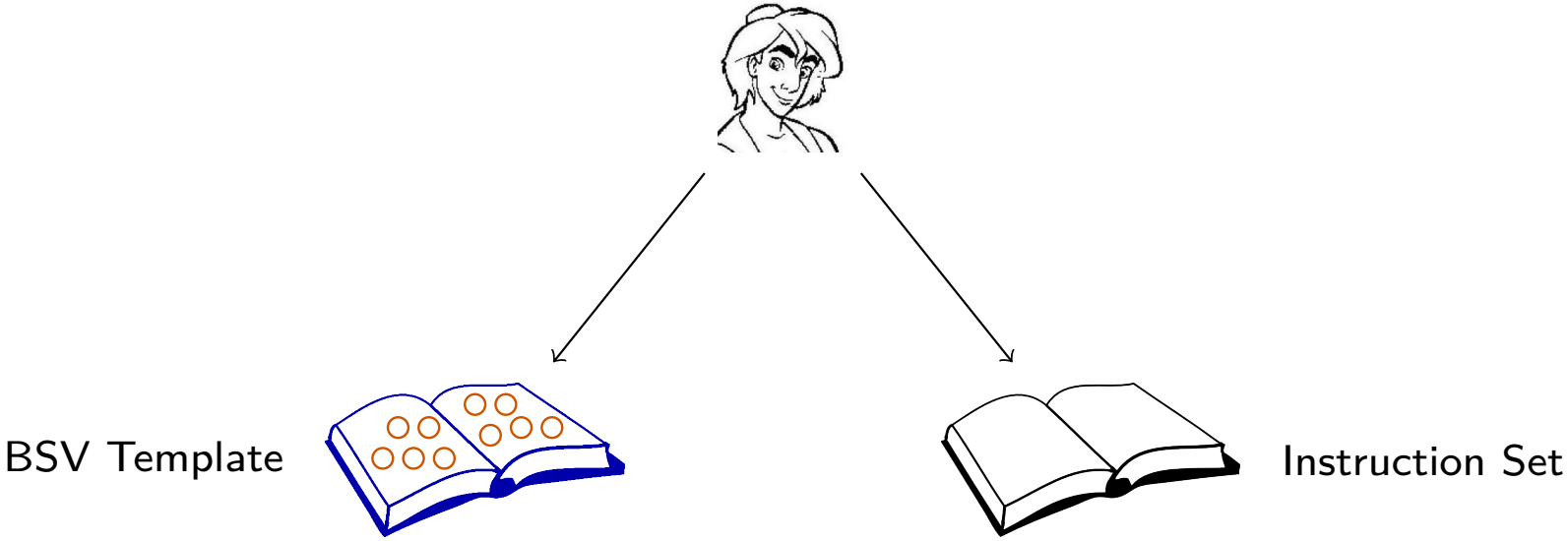
Bluespec
SystemVerilog
(BSV)



Bluespec
synthesis
toolflow

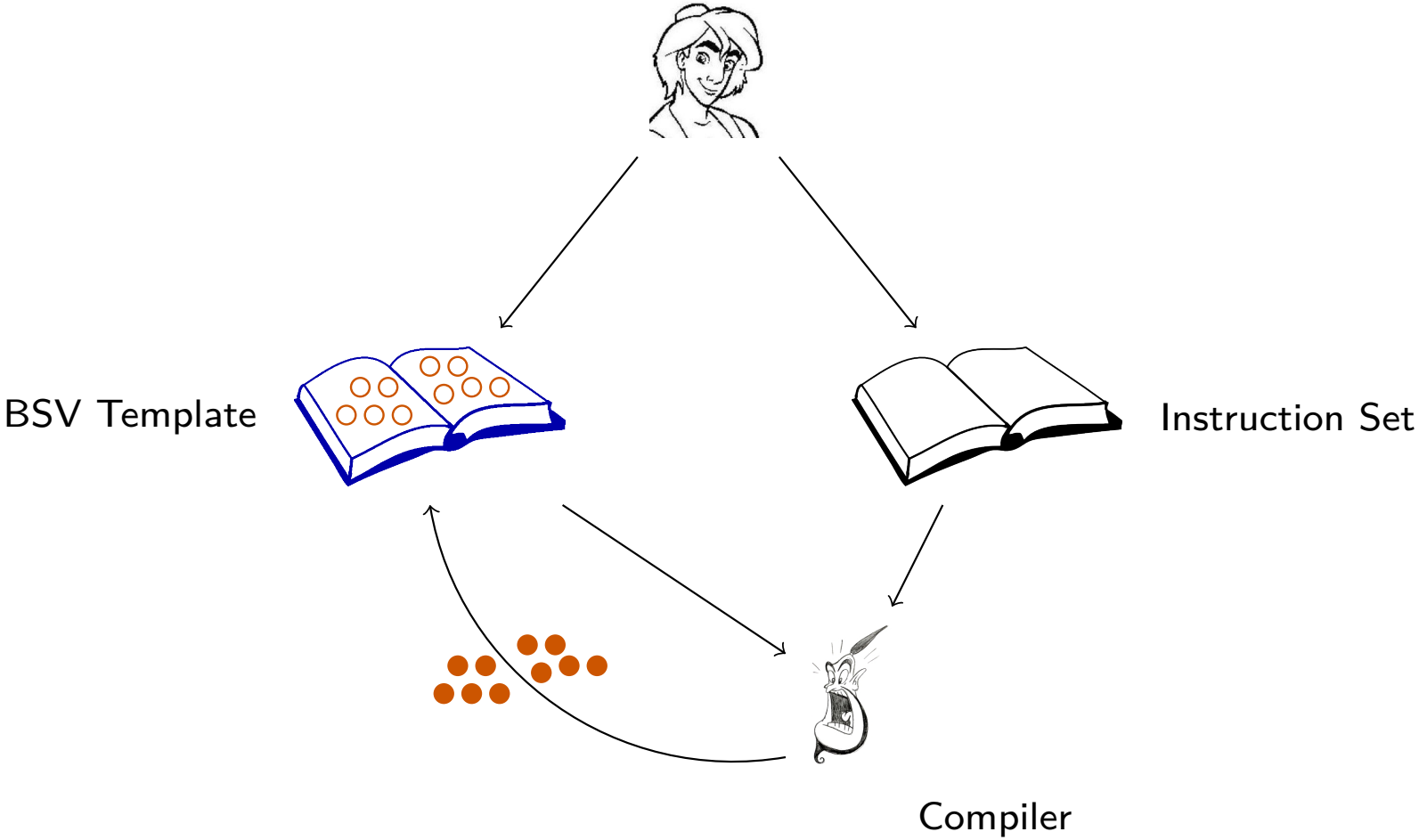
# Toolflow

## Two descriptions



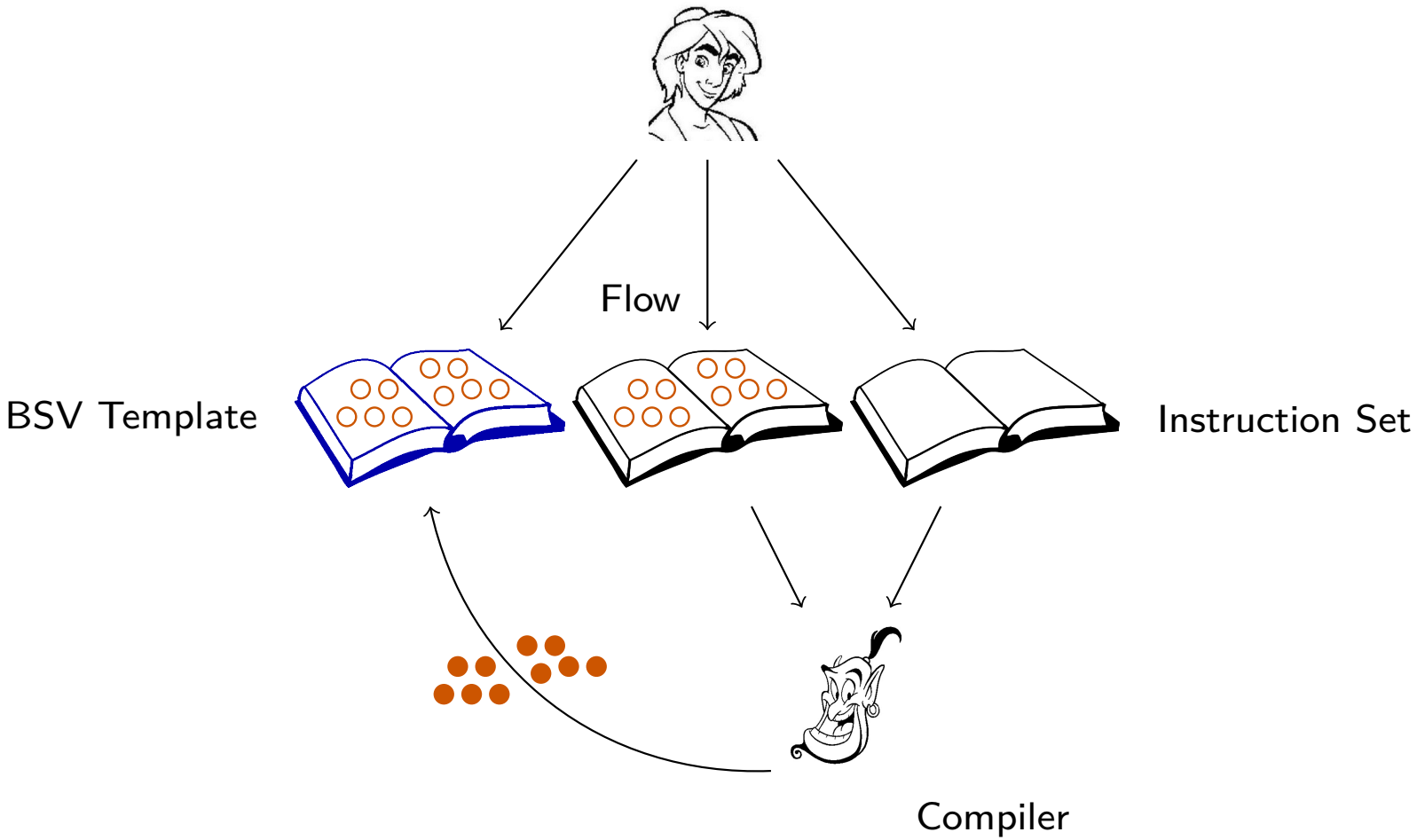
# Toolflow

## Holes



# Toolflow

## Flow



# Architectural State

μL language

- sequential
- static types
- type inference
- bit-width inference
- extensible structs

```
RegID ← defenum [EAX, ECX, EDX, ...]

State ← defstruct
          [ (Bit 32                  , PC)
          , (Array RegID (Bit 32)    , RF)
          , (Array Addr (Bit 8)      , MEM)
          , (CCode                   , CC)
          , (Bool                    , HLT)
          ]
```

# Instruction Set

μL language

- sequential
- static types
- type inference
- bit-width inference
- extensible structs

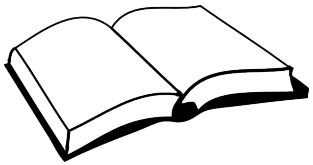
```
def NOP(inst):
        pc ← select (PC)
        update (PC, pc + 1)

def JMP(inst):
        pc ← select (PC)
        update (PC, pc + signExt(inst.IMM))

def POP(inst):
        pc ← select (PC)
        update (PC, pc + 2)
        sp ← read (RF, ESP)
        x ← read4 (MEM, sp)
        write (RF, ESP, sp + 4)
        write (RF, inst.DEST, x)
```

# Instruction Set

It is easy to describe "functionality" without reference to "timing"



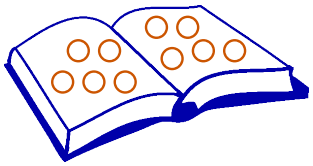
Instruction Set

Limitation: we cannot directly specify

- memory model
- non-deterministic instructions

# Microarchitectural Template: holes

It is difficult to describe microarchitecture without reference to ISA



BSV Template

- Details of the ISA manifest "all over the place"
- User can use holes to skip such details by using holes in BSV
- Compiler tries to fill in holes using ISA information

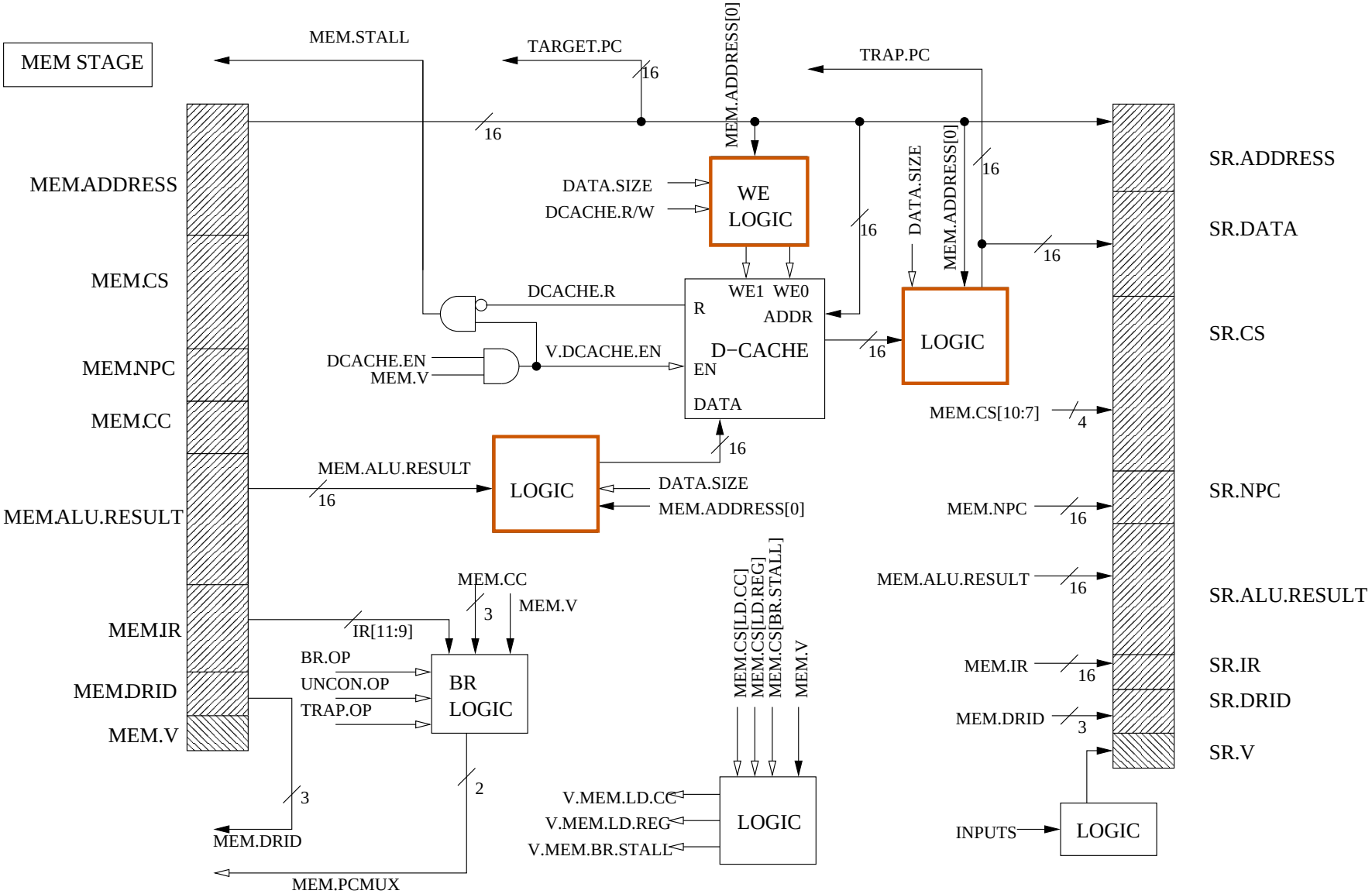
▷ A. Solar-Lezama et al, PLDI 2005
*Programming by Sketching for Bitstreaming Programs*

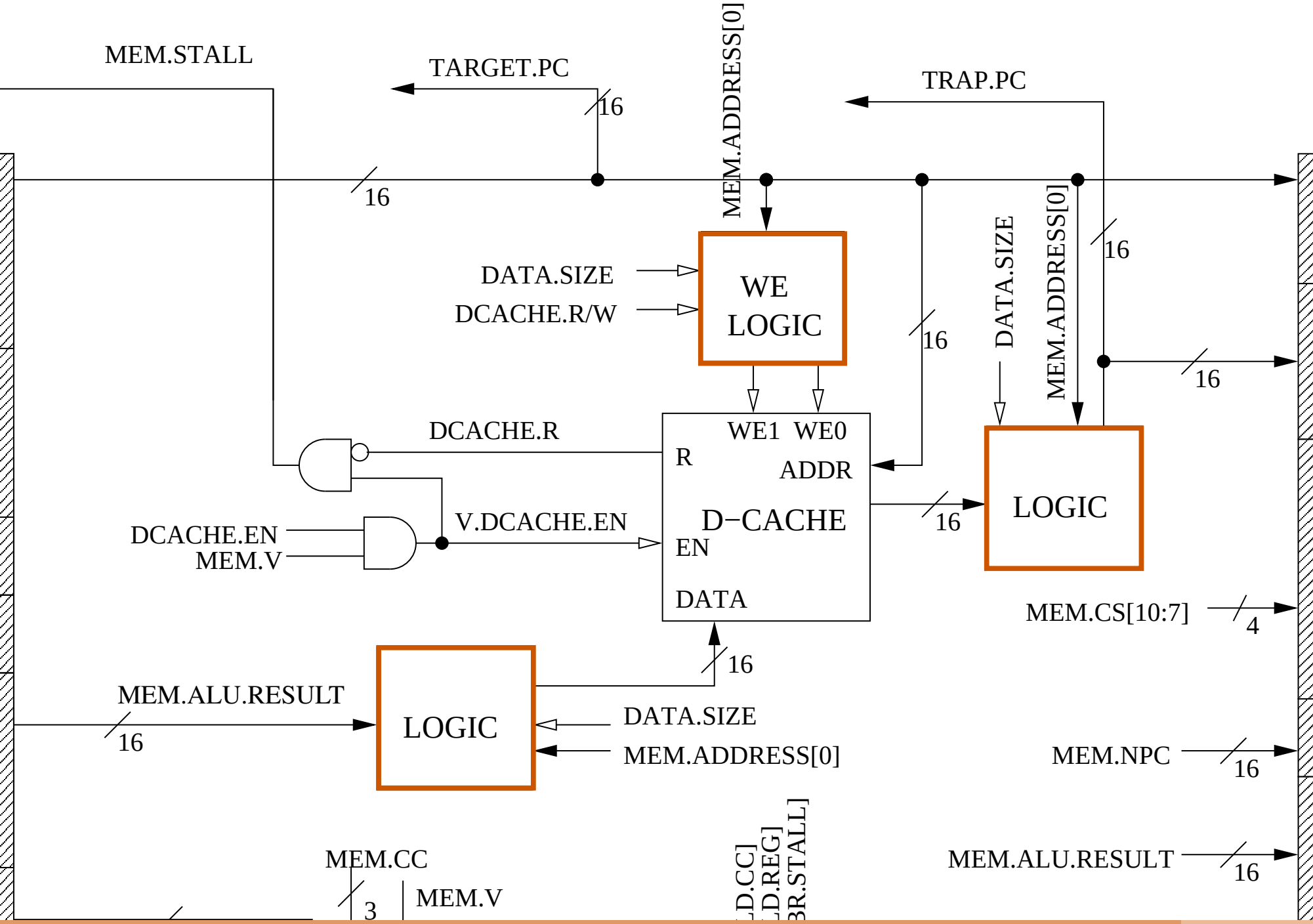
# Holes: used in teaching

University of Texas at Austin

- EE 460N: Undergraduate Computer Architecture
- Simulate a 5-stage pipeline for LC3b ISA
- Given a detailed document describing microarchitecture
- Students fill in empty LOGIC boxes

# LC3b pipeline: memory stage




MEM.STALL
TARGET.PC
16
MEM.ADDRESS[0]
TRAP.PC
16
DATA.SIZE
DCACHE.R/W
WE
LOGIC
16
DATA.SIZE
MEM.ADDRESS[0]
16
16
DCACHE.R
WE1 WE0
R
ADDR
D−CACHE
EN
DATA
16
LOGIC
DCACHE.EN
MEM.V
V.DCACHE.EN
MEM.CS[10:7]
4
16
MEM.ALU.RESULT
16
LOGIC
DATA.SIZE
MEM.ADDRESS[0]
MEM.NPC
16
MEM.CC
3
MEM.V
MEM.ALU.RESULT
16

# How big are the holes?

carefree user

make the entire processor a single hole

insert pipeline registers leaving each stage as a hole

refine the fetch stage to use an instruction-cache

...

describe detailed hardware with several small holes all over

power user → many, small, combinational holes

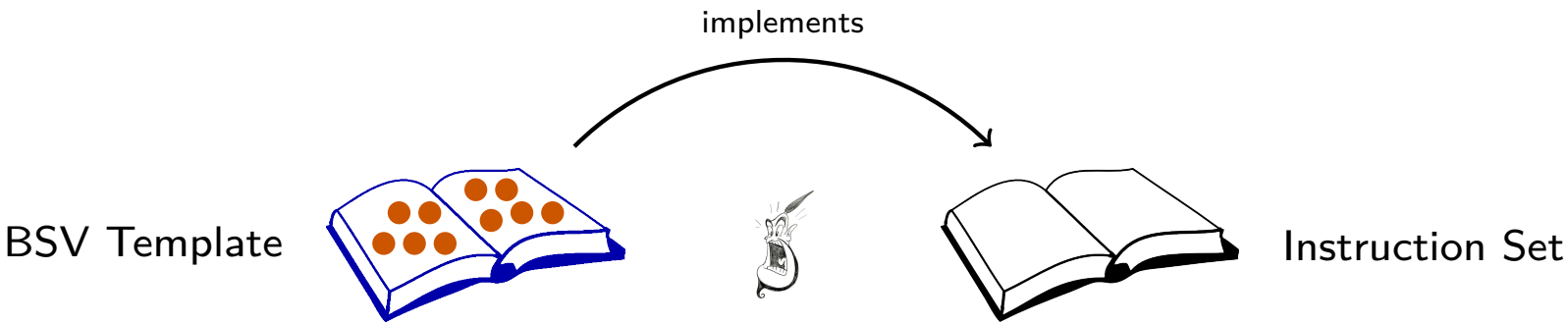
# Holes in BSV

- Manually specified as a pure function: $\#hole(\cdots)$
- Explicit inputs, but not necessarily minimal
- Automatically defined using ISA information
- Synthesized to combinational logic

## Example

```
dcache.request(#ldst_p(uop),                    ▷ load/store
               phyAddr(#addr(uop)),             ▷     address
                 #st_data(uop));                ▷ store-data
```

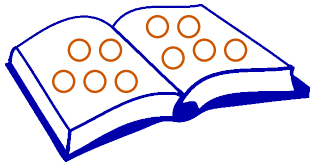
# Compiler correctness



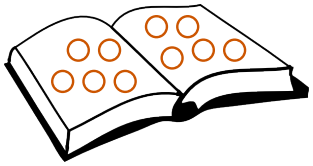

Filled-in BSV is a valid microarchitecture for this ISA

- Makes hole synthesis at least as "hard" as verification
- Too strong a notion for correctness
- How can we weaken this?

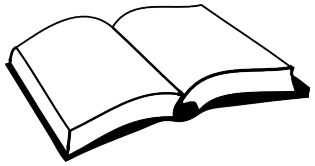
# Flow: an intermediate spec



BSV Template



Flow



Instruction Set

Flow describes the functional execution of a single instruction through the microarchitecture template

# Flow: example

μL language

- sequential
- static types
- type inference
- bit-width inference
- extensible structs

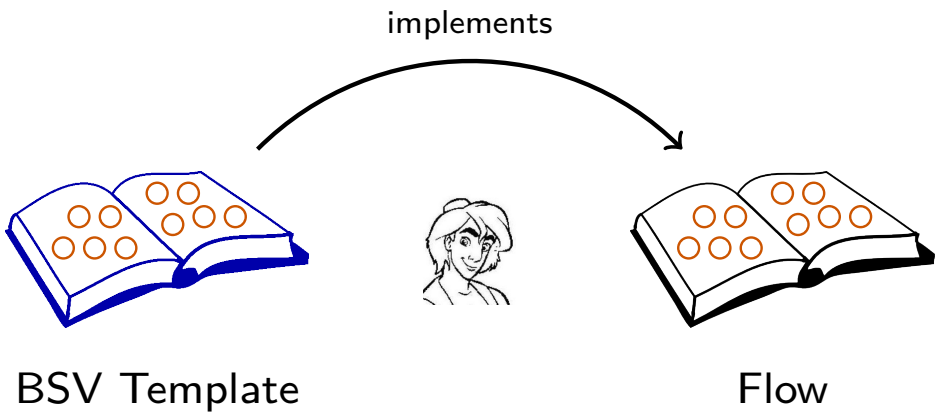
```
while (true):
    pc ← select (PC)
    inst_0 ← fetch (pc)
    update (PC, pc + #ilen(inst_0))

    x ← read (RF, #src(inst_0))
    inst_1 ← inst_0 +: (DATA, x)

    z ← #add_x(inst_1) + #add_y(inst_1)
    inst_2 ← inst_1 +: (RESULT, z)

    pupdate (PC, #jmp(inst_2), #target(inst_2))
    pwrite (RF, #wr_en(inst_2), #dest(inst_0), z)
```

# Flow: divide problem into two

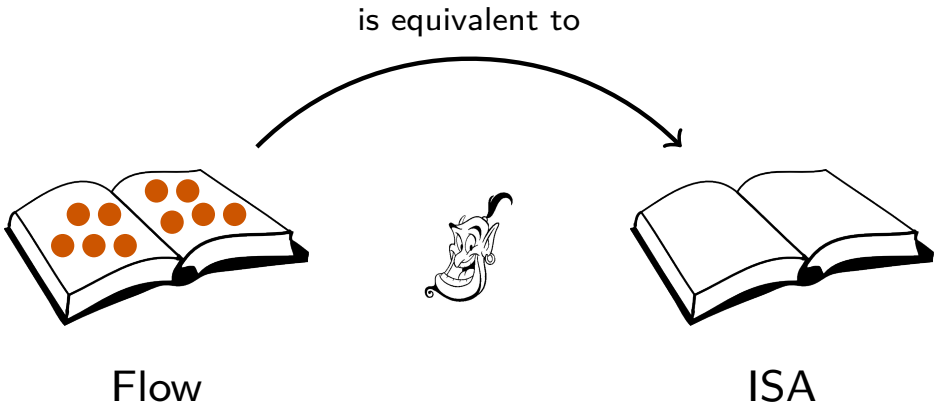



## User

BSV template must "implement" flow for every type-correct hole definition

## Compiler

Flow must be equivalent to ISA for the generated hole definition



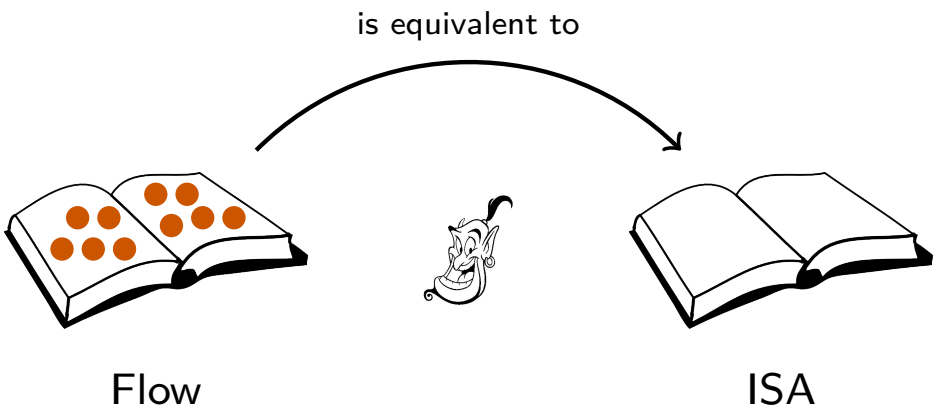
# Flow restricts holes

Holes are general enough to disentangle the instruction set

But, a hole cannot contain

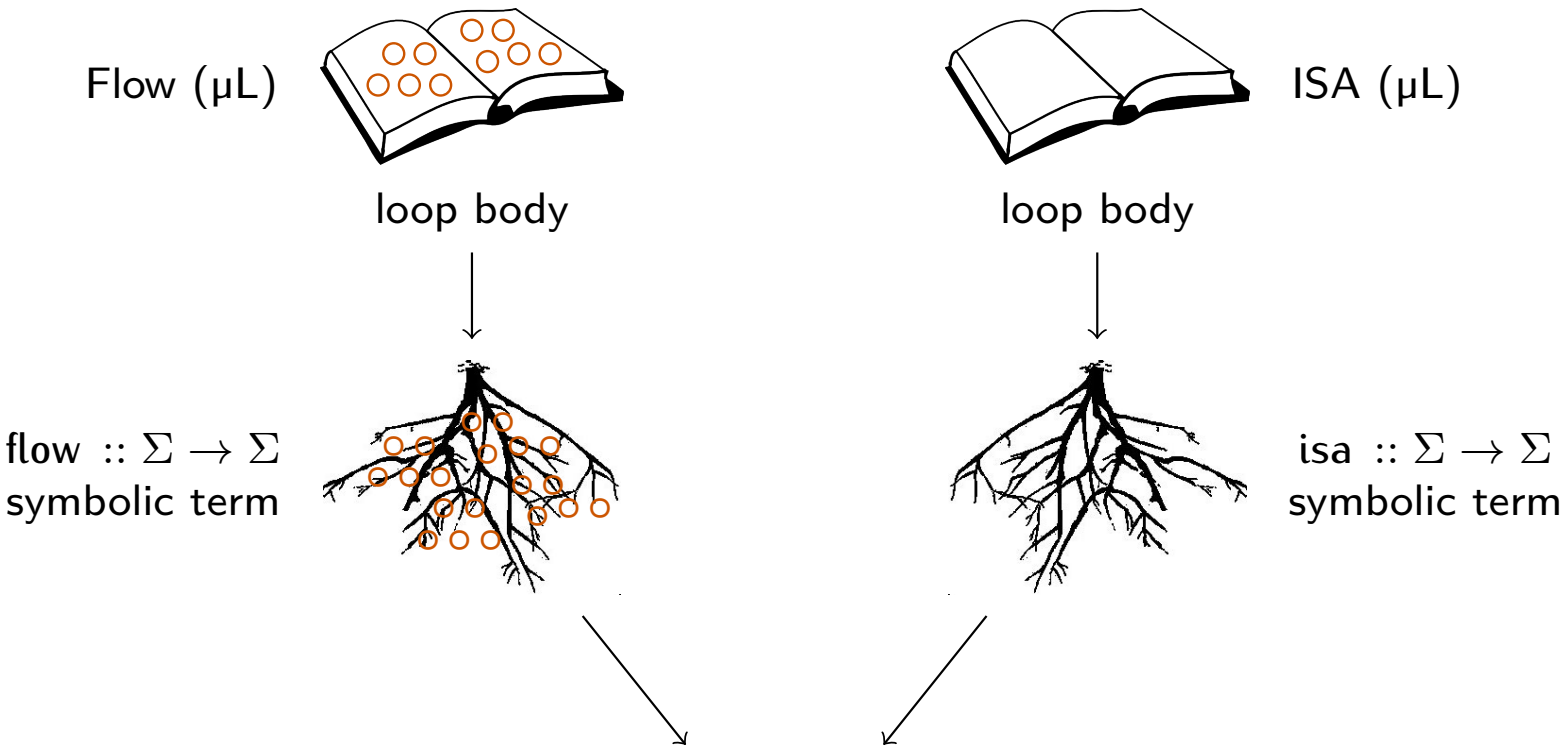
- intrinsically time-dependent logic
- pipeline control logic
- cache control logic
- inter-instruction dependency logic

# Compiler overview




- Generate hole definitions such that flow is equivalent to ISA
- Equivalence determined by a term rewriting system $\mathcal{R}$

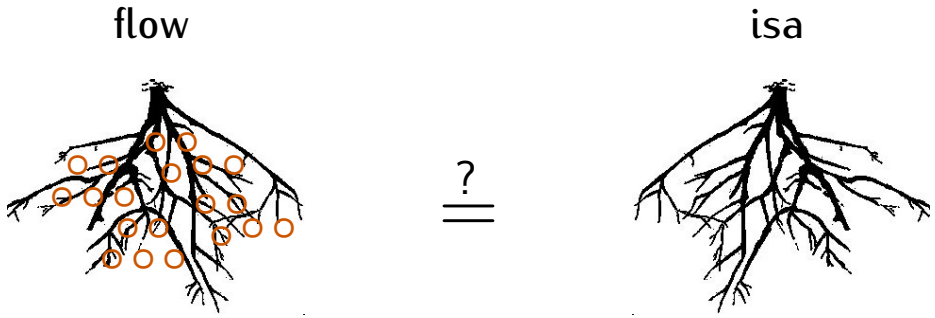
# Compiler problem formulation




$$\exists\theta.\ \forall\Sigma.\ \ \theta(\mathbf{flow}\,(\Sigma)) \stackrel{?}{=} \mathbf{isa}\,(\Sigma)$$

Second-order $\mathcal{R}$-matching

# $\mathcal{R}$-matching & Lazy Narrowing




- Problem formulated as second-order $\mathcal{R}$-matching
- Narrowing is a systematic way to solve such problems
- Lazy narrowing is a refinement that works outside-in
- Heuristic-guided systematic search, backtracking as necessary

▷ Christian Prehofer, *Solving Higher-Order Equations*, 1995

# Practical concerns

- Solver can be very slow and timeout
- Need limits on hole size
- Equation solver errors are cryptic

# Evaluation target

Y86-inorder

- CMU, Introduction to Computer Systems
- 27 instructions (including variants)
- 5-stage inorder pipeline

# Y86-inorder: Compile times

| Compiler stage | Time |
|---|---|
| Typecheck | 0.2 s |
| Normalize | 1.5 s |
| Solver | 4.9 s |
| Microcode | 0.1 s |
| Total | 6.7 s |

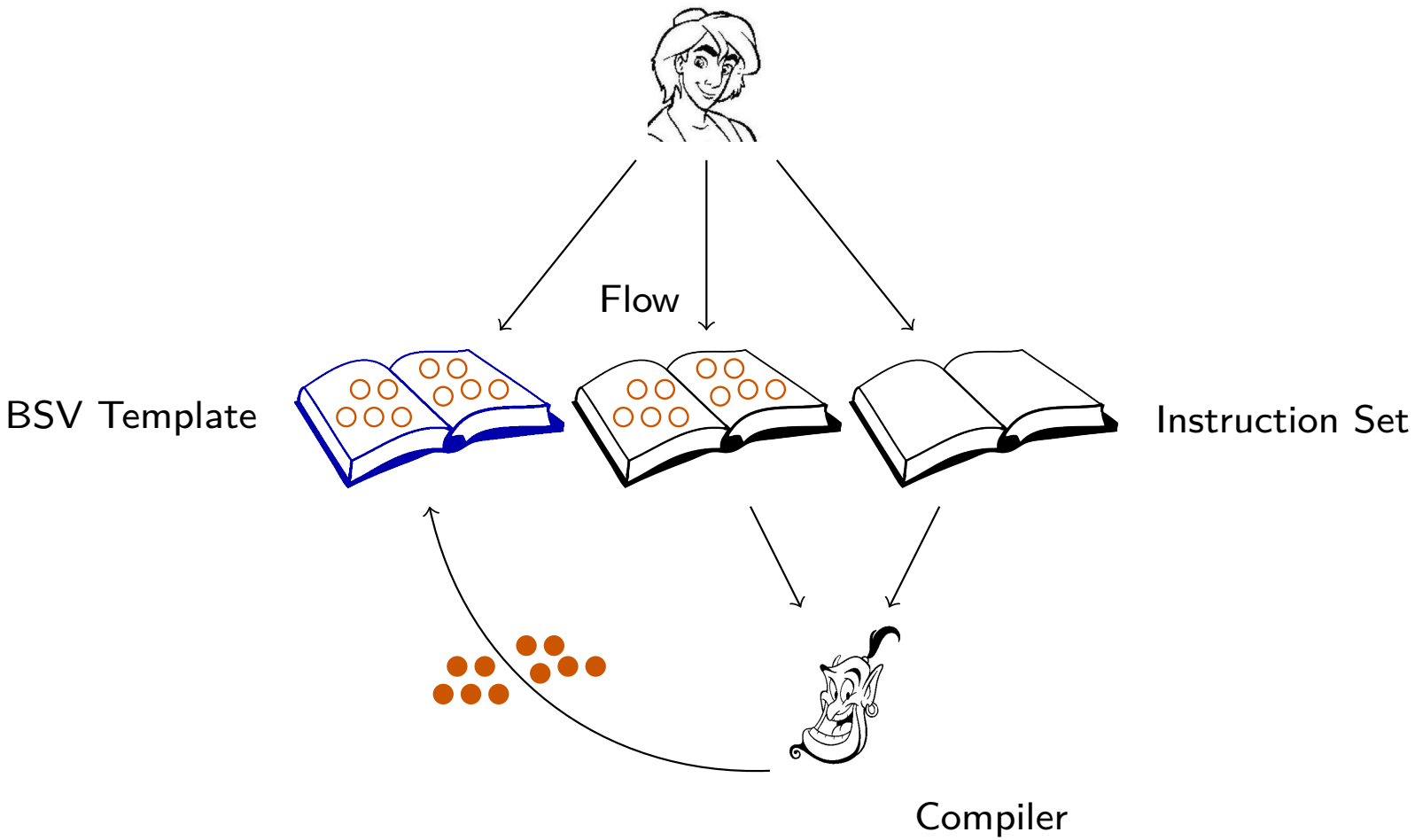
$\longrightarrow$

| Instruction | Time |
|---|---|
| RRMOV | 0.42 s |
| RMMOV | 0.33 s |
| MRMOV | 0.31 s |
| ADD | 0.32 s |
| JXX | 0.16 s |
| PUSH | 0.29 s |
| POP | 0.39 s |
| CALL | 0.24 s |
| RET | 0.43 s |
| LEAVE | 0.35 s |
| ... | |

# Y86-inorder: Lines of code

| | | | | |
|---|---|---|---|---|
| Vanilla BSV flow | Full processor | 3300 | lines | BSV |
| | Core pipeline | 450 | lines | BSV |
| Input to compiler | Core pipeline | 400 | lines | BSV |
| | ISA | 180 | lines | μL |
| | Flow | 63 | lines | μL |
| Output of compiler | Microcode | 20 | lines | BSV |
| | | 510 | bits | |
| | Functions | 300 | lines | BSV |
| | | 730 | gates | |

# Summary



Thank you!