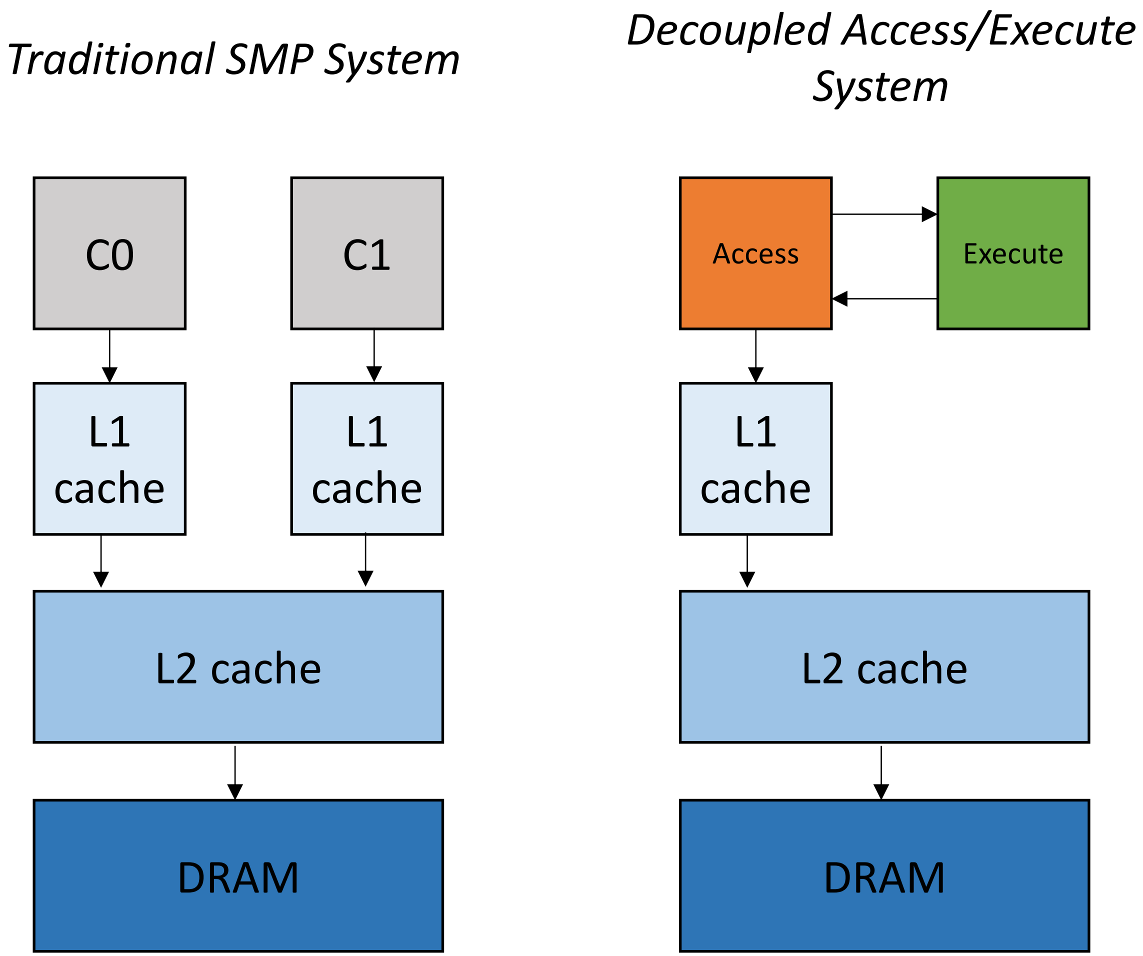
CSE211: Compiler Design

Nov. 15, 2021

- **Topic**: Decoupled Access Execute (DAE)

- **Discussion questions**:
  - What does it mean for an application to be memory bound?
  - What are some techniques for dealing with memory bottlenecks



Announcements

- Homework 2 and midterm are graded
  - Let me know if there are issues or if you have questions (Office hours on Thursday)
- Homework 3 is due on Wednesday
  - Feel free to share results on slack, but not code
  - Homework 4 is planned for release on Wednesday
- Finishing up parallelization, next we will start on DSLs
- Guest lecture for Nov. 22
  - Aviral Goel will talk about laziness in R

Announcements

- **Paper assignment**:
  - Add your paper, topic and name to sheet, please try to do by Wednesday
- **Project**:
  - Add your name and project title to sheet
  - You have until the 29th to switch to the final
  - Option for blog post (potentially)
- *I will email links to the sheets later tonight*

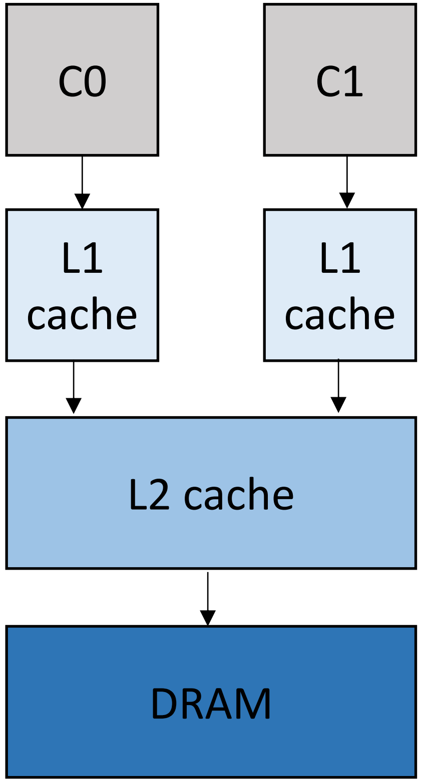
CSE211: Compiler Design

Nov. 15, 2021

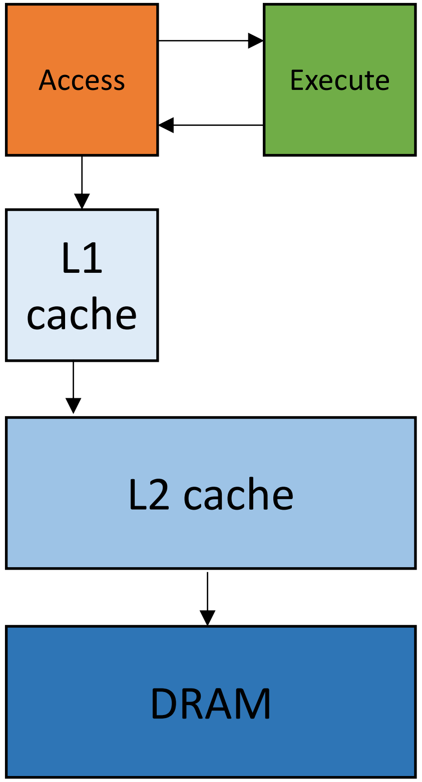
- **Topic**: Decoupled Access Execute (DAE)

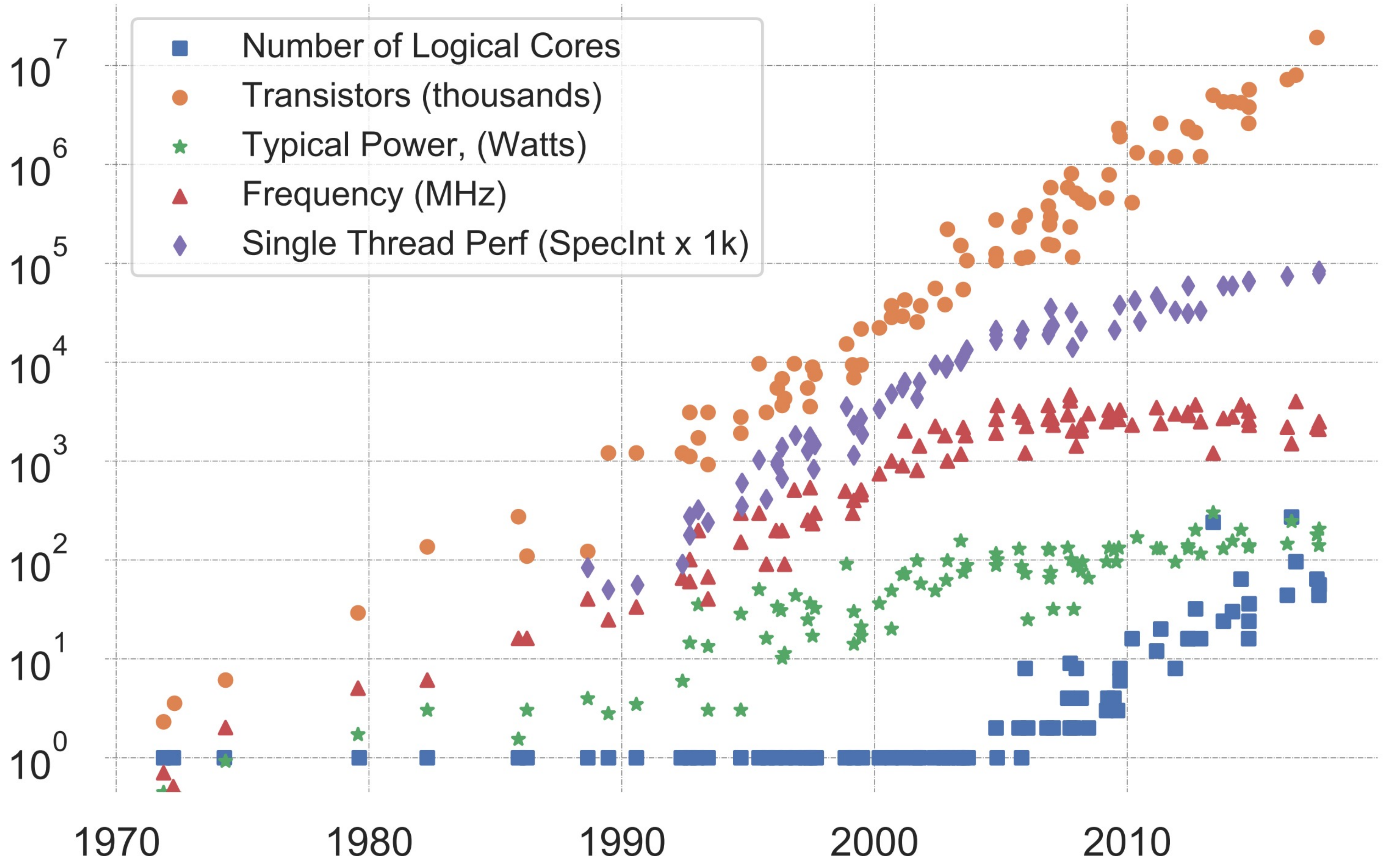
- **Discussion questions**:
  - What does it mean for an application to be memory bound?
  - What are some techniques for dealing with memory bottlenecks

Traditional SMP System



Decoupled Access/Execute System





K. Rupp, “40 Years of Mircroprocessor Trend Data,” https://www. karlrupp.net/2015/06/40-years-of-microprocessor-trend-data, 2015.

Specialization discussion

- CPUs:
  - Aim to be good at general tasks
  - poor area and energy utilization

Benchmarking TPU, GPU, and CPU Platforms for Deep Learning, arxiv 2019

Specialization discussion

- CPUs:
  - Aim to be good at general tasks
  - poor area and energy utilization

How many floating point operations per second (FLOPS) on matrix multiplication

2 TFLOPS

Benchmarking TPU, GPU, and CPU Platforms for Deep Learning, arxiv 2019

Specialization discussion

How many floating point operations per second (FLOPS) on matrix multiplication

2 TFLOPS

- CPUs:
  - Aim to be good at general tasks
  - poor area and energy utilization
- GPUs:
  - Good at regular, uniform parallelism
  - Bad at irregular parallelism and programs with control dependencies

Benchmarking TPU, GPU, and CPU Platforms for Deep Learning, arxiv 2019

Specialization discussion

How many floating point operations per second (FLOPS) on matrix multiplication

- CPUs:
  - Aim to be good at general tasks
  - poor area and energy utilization

2 TFLOPS

- GPUs:
  - Good at regular, uniform parallelism
  - Bad at irregular parallelism and programs with control dependencies

125 TFLOPS
(62x faster than CPU)

Benchmarking TPU, GPU, and CPU Platforms for Deep Learning, arxiv 2019

Specialization discussion

How many floating point operations per second (FLOPS) on matrix nultiplication

- CPUs:
  - Aim to be good at general tasks
  - poor area and energy utilization

2 TFLOPS

- GPUs:
  - Good at regular, uniform parallelism
  - Bad at irregular parallelism and programs with control dependencies

125 TFLOPS
(62x faster than CPU)

- TPUs:
  - Good at matrix multiplication
  - Not good at much else (12 instructions)

Benchmarking TPU, GPU, and CPU Platforms for Deep Learning, arxiv 2019

Specialization discussion

How many floating point operations per second (FLOPS) on matrix nultiplication

- CPUs:
  - Aim to be good at general tasks
  - poor area and energy utilization

2 TFLOPS

- GPUs:
  - Good at regular, uniform parallelism
  - Bad at irregular parallelism and programs with control dependencies

125 TFLOPS
(62x faster than CPU)

- TPUs:
  - Good at matrix multiplication
  - Not good at much else (12 instructions)

180 TFLOPS
(much faster than CPU,
1.4x faster than GPU)

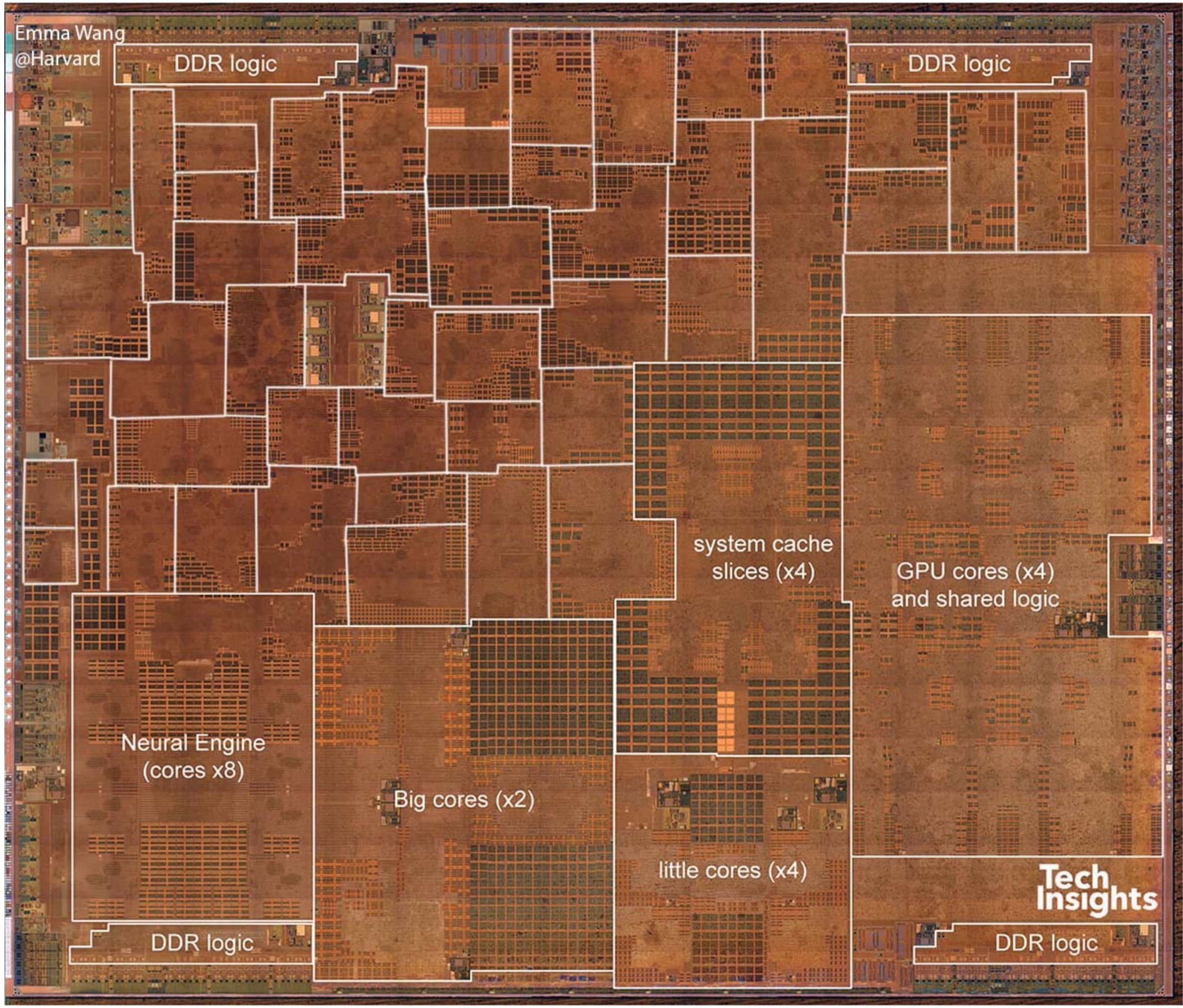
Benchmarking TPU, GPU, and CPU Platforms for Deep Learning, arxiv 2019

Specialization in modern SoCs

- From David Brooks lab at Harvard:

  http://vlsiarch.eecs.harvard.edu/research/accelerators/die-photo-analysis/

- CPUs, GPUs, Neural Engine, IP blocks (cryptography, DSP, etc.)



How do programs take advantage of specialization?

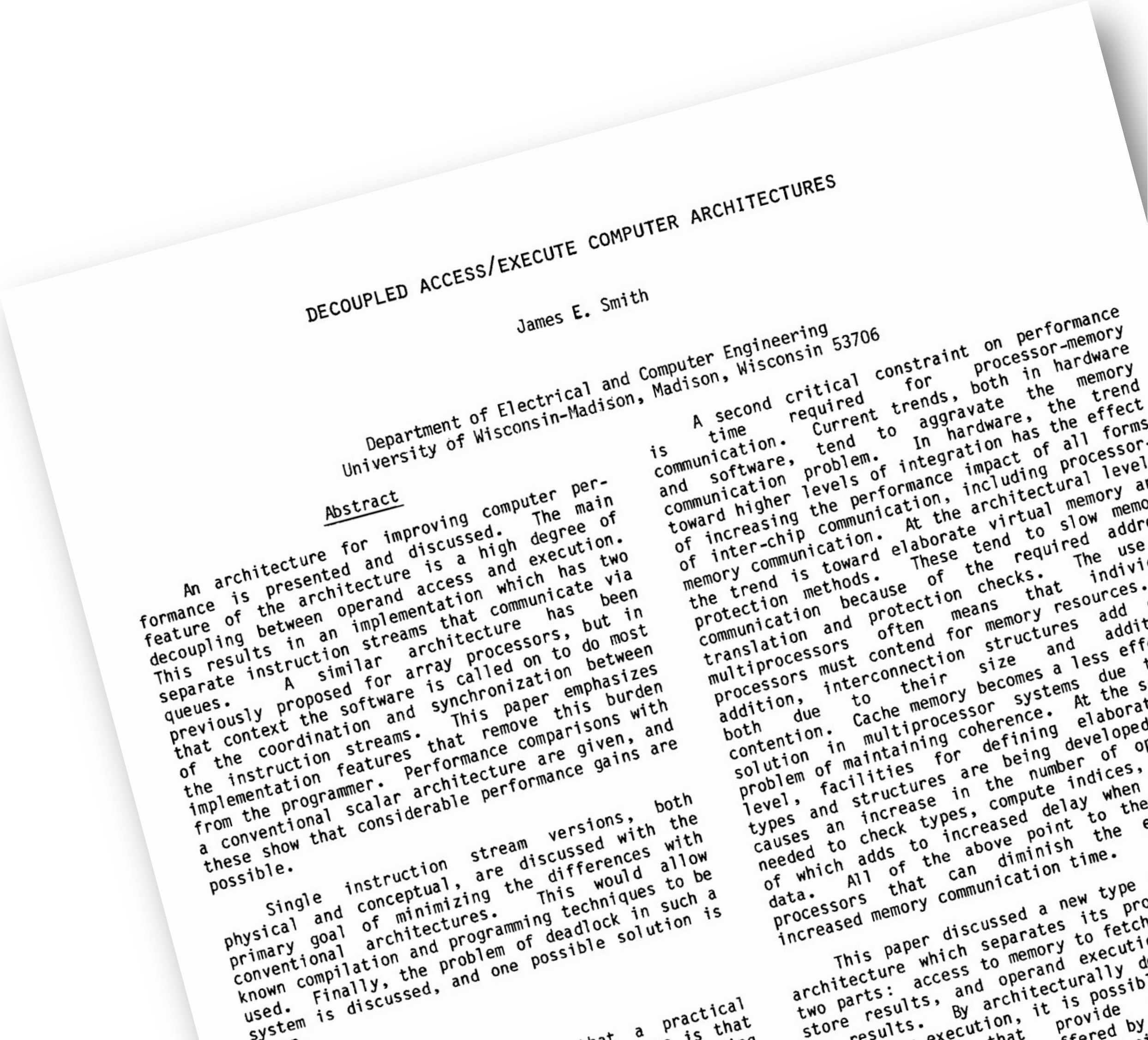
- **Programmer-centric:**
  - Programmers write code using a specific API
  - e.g. Tensorflow targets CPU, GPU, TPU
- **Hardware-centric:**
  - Hardware transparently optimizes programs
  - Pipelining, super scalar, caches, etc. (what our traditional systems already do)
- **Compiler-centric:**
  - Compiler performs non-trivial transformations to target specialized hardware

Specialization is not new

- First GPU in 1951 (MIT flight simulator)
- Architecture academic work proposes many new designs
  - Evaluated on detailed simulators; rarely taped out
- Had a hard time breaking into the mainstream:
  - benefits had to outweigh eventual returns from Dennard's Scaling and Moore's Law
- But now...
  - Hennessy and Patterson's 2017 Turing award lecture: The New Golden Age of Computer Architecture

Decoupled Access/Execute (DAE)

- 1982: James E. Smith
  - Lives in Montana now and gives interesting keynotes at architecture conferences
    "Reverse-Engineering the Brain: A Computer Architecture Grand Challenge" ISCA 2018



DECOUPLED ACCESS/EXECUTE COMPUTER ARCHITECTURES

James E. Smith

Department of Electrical and Computer Engineering
University of Wisconsin-Madison, Madison, Wisconsin 53706

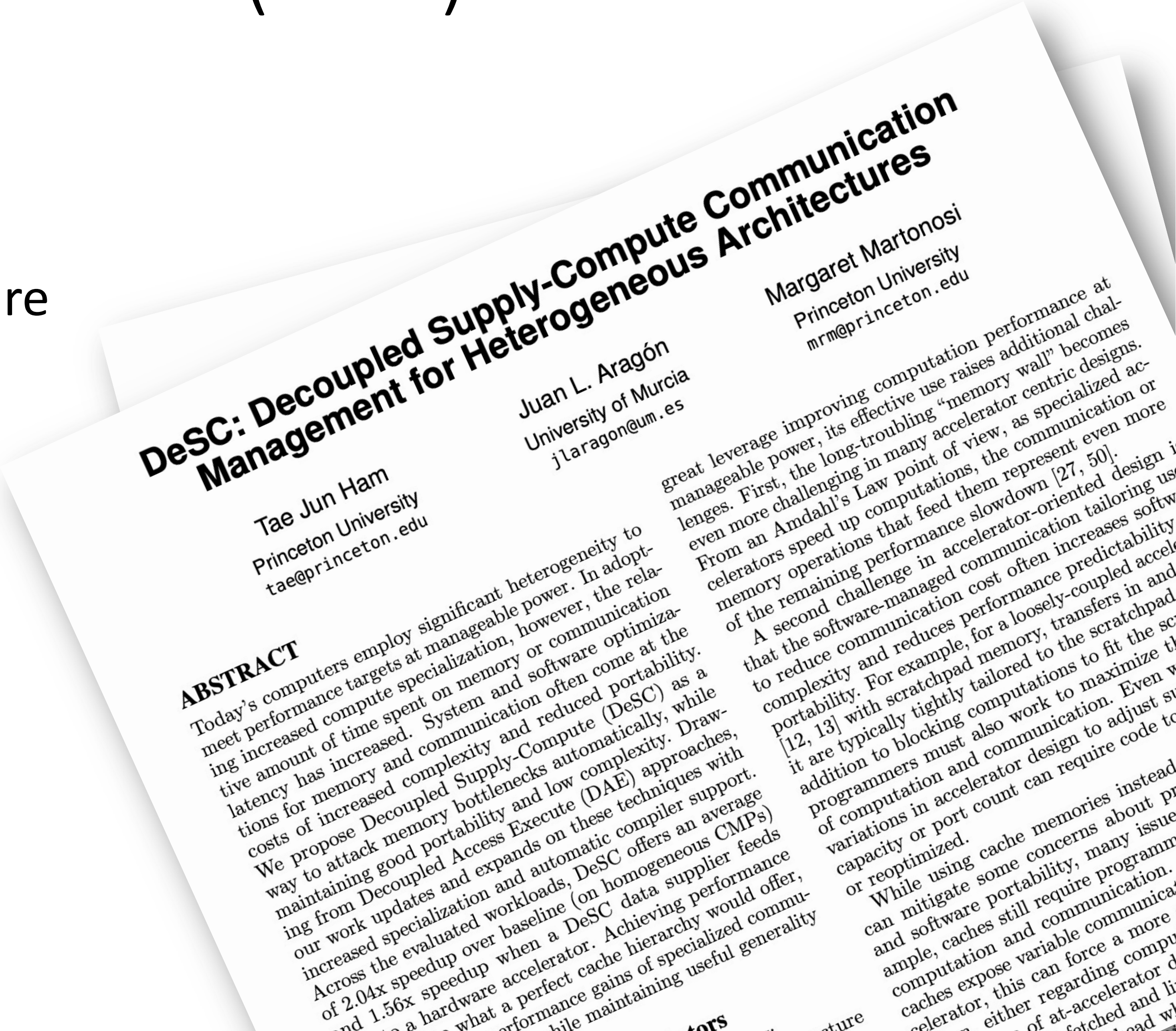
Abstract

An architecture for improving computer performance is presented and discussed. The main feature of the architecture is a high degree of decoupling between operand access and execution. This results in an implementation which has two separate instruction streams that communicate via queues. A similar architecture has been previously proposed for array processors, but in that context the software is called on to do most of the coordination and synchronization between the instruction streams. This paper emphasizes implementation features that remove this burden from the programmer. Performance comparisons with a conventional scalar architecture are given, and these show that considerable performance gains are possible.

Single instruction stream versions, both physical and conceptual, are discussed with the primary goal of minimizing the differences with conventional architectures. This would allow known compilation and programming techniques to be used. Finally, the problem of deadlock in such a system is discussed, and one possible solution is

Decoupled Access/Execute (DAE)

- 1982: James E. Smith
  - Lives in Montana now and gives interesting keynotes at architecture conferences
- 2015: DeSC by Ham et al.
  - More optimizations and practicalities



DeSC: Decoupled Supply-Compute Communication Management for Heterogeneous Architectures

Tae Jun Ham
Princeton University
tae@princeton.edu

Juan L. Aragón
University of Murcia
jlaragon@um.es

Margaret Martonosi
Princeton University
mrm@princeton.edu

ABSTRACT

Today's computers employ significant heterogeneity to meet performance targets at manageable power. In adopting increased compute specialization, however, the relative amount of time spent on memory or communication latency has increased. System and software optimizations for memory and communication often come at the costs of increased complexity and reduced portability. We propose Decoupled Supply-Compute (DeSC) as a way to attack memory bottlenecks automatically, while maintaining good portability and low complexity. Drawing from Decoupled Access Execute (DAE) approaches, our work updates and expands on these techniques with increased specialization and automatic compiler support. Across the evaluated workloads, DeSC offers an average of 2.04x speedup over baseline (on homogeneous CMPs) and 1.56x speedup when a DeSC data supplier feeds

great leverage improving computation performance at manageable power, its effective use raises additional challenges. First, the long-troubling "memory wall" becomes even more challenging in many accelerator centric designs. From an Amdahl's Law point of view, as specialized accelerators speed up computations, the communication or memory operations that feed them represent even more of the remaining performance slowdown [27, 50].

DAE - motivation

simple example program

```
for (int i = 0; i < SIZE; i++) {
   a[i] = b[i] * 3.14;
}
```

DAE - motivation

simple example program

```
for (int i = 0; i < SIZE; i++) {
   a[i] = b[i] * 3.14;
}
```

pseudo 3-address code

```
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   float r1 = r0 * 3.14;
   store(a + i, r1);
}
```

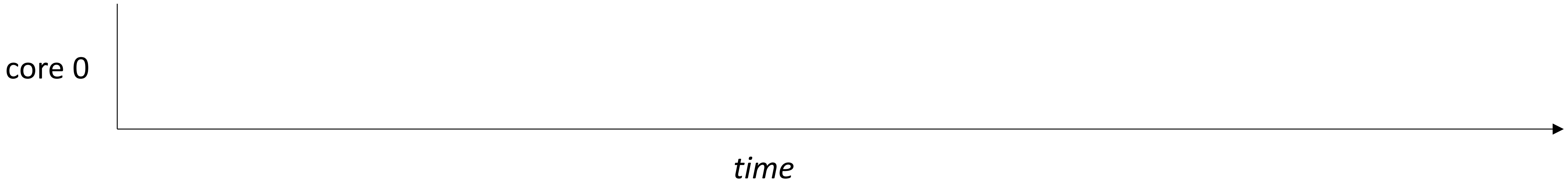
DAE - motivation

simple example program

```
for (int i = 0; i < SIZE; i++) {
   a[i] = b[i] * 3.14;
}
```

pseudo 3-address code

```
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   float r1 = r0 * 3.14;
   store(a + i, r1);
}
```



DAE - motivation

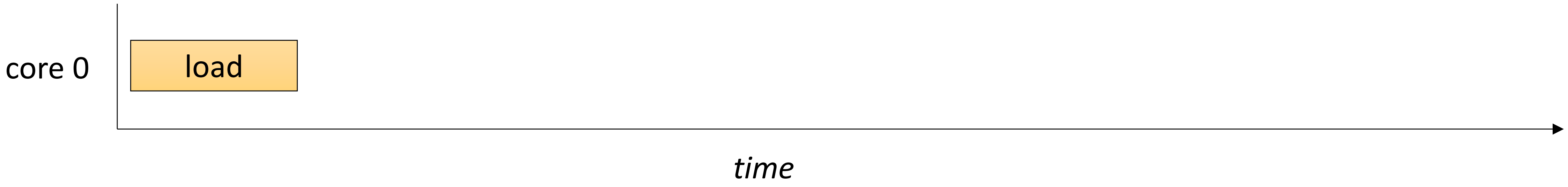
simple example program

```
for (int i = 0; i < SIZE; i++) {
   a[i] = b[i] * 3.14;
}
```

→

pseudo 3-address code

```
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   float r1 = r0 * 3.14;
   store(a + i, r1);
}
```



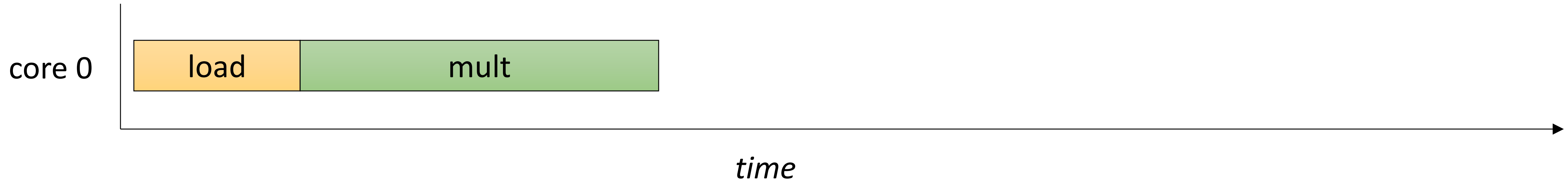
DAE - motivation

simple example program

```
for (int i = 0; i < SIZE; i++) {
   a[i] = b[i] * 3.14;
}
```

pseudo 3-address code

```
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   float r1 = r0 * 3.14;
   store(a + i, r1);
}
```



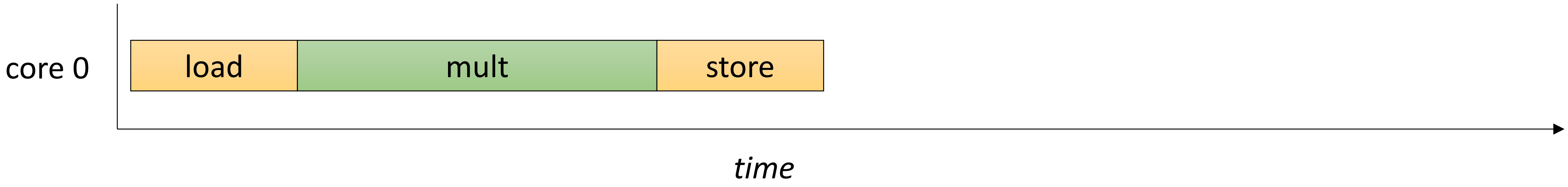
DAE - motivation

simple example program

```
for (int i = 0; i < SIZE; i++) {
    a[i] = b[i] * 3.14;
}
```

pseudo 3-address code

```
for (int i = 0; i < SIZE; i++) {
    float r0 = load(b + i);
    float r1 = r0 * 3.14;
    store(a + i, r1);
}
```



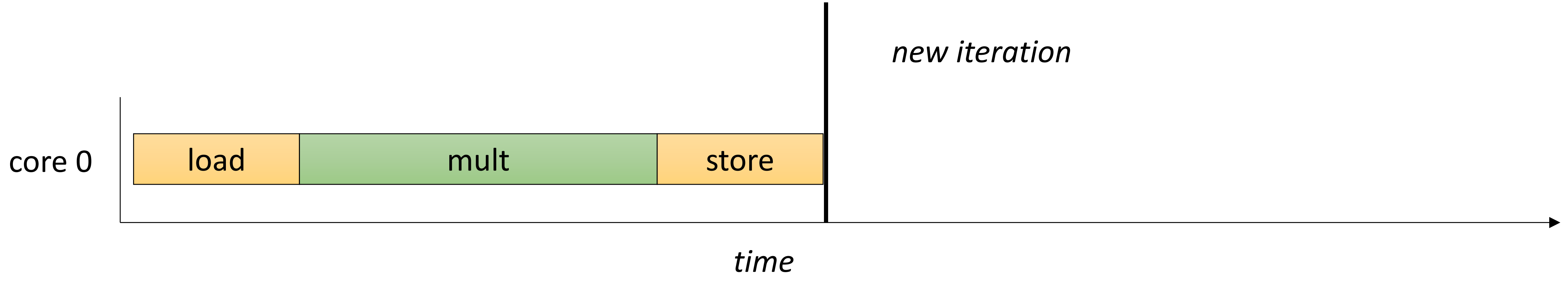
DAE - motivation

simple example program

```
for (int i = 0; i < SIZE; i++) {
   a[i] = b[i] * 3.14;
}
```

pseudo 3-address code

```
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   float r1 = r0 * 3.14;
   store(a + i, r1);
}
```



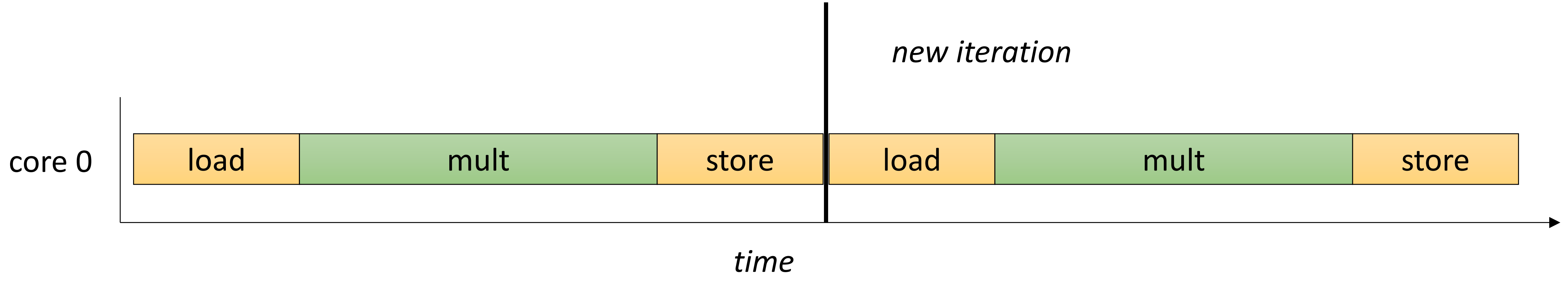
DAE - motivation

simple example program

```
for (int i = 0; i < SIZE; i++) {
   a[i] = b[i] * 3.14;
}
```

pseudo 3-address code

```
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   float r1 = r0 * 3.14;
   store(a + i, r1);
}
```



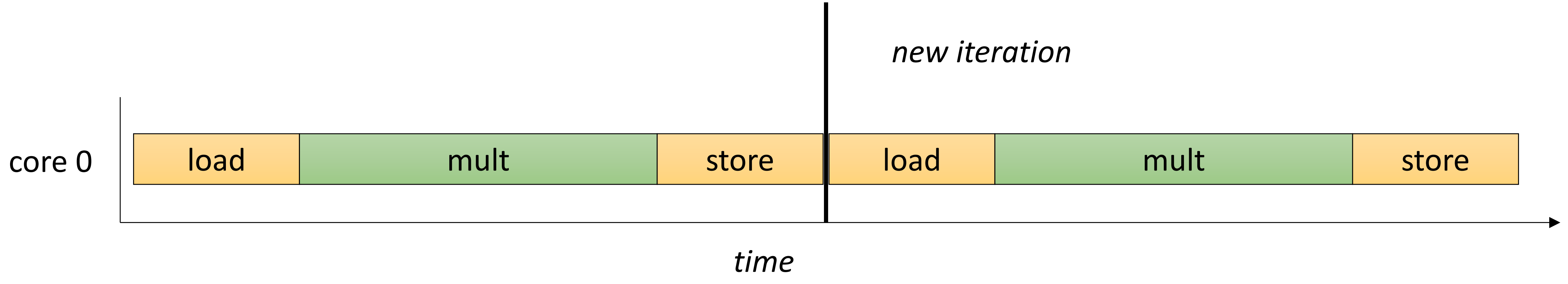
DAE - motivation

simple example program

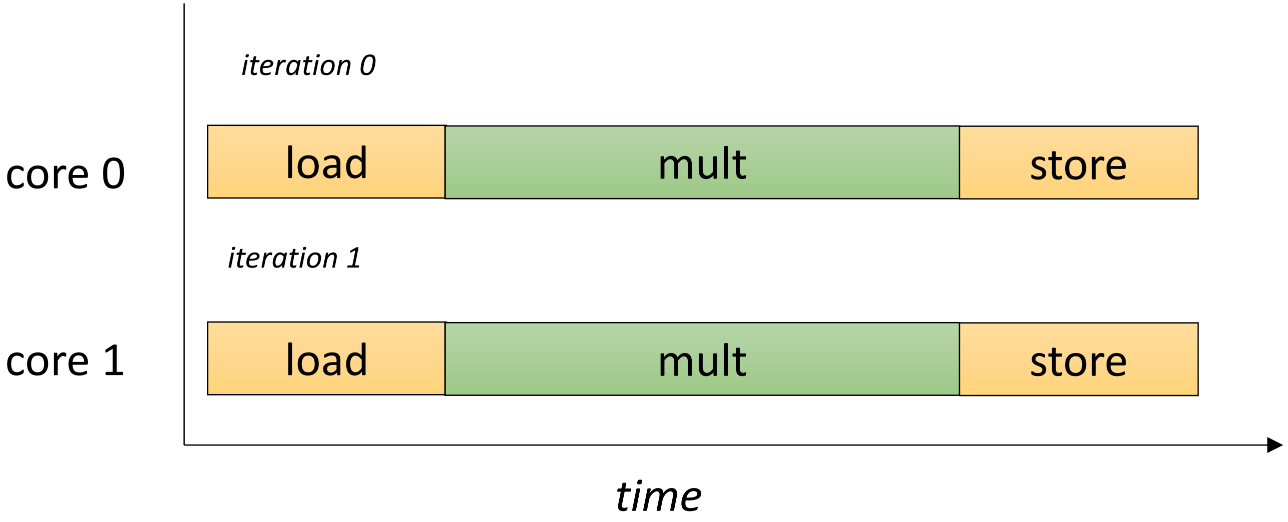
```
for (int i = 0; i < SIZE; i++) {
   a[i] = b[i] * 3.14;
}
```

pseudo 3-address code

```
#pragma parallel
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   float r1 = r0 * 3.14;
   store(a + i, r1);
}
```



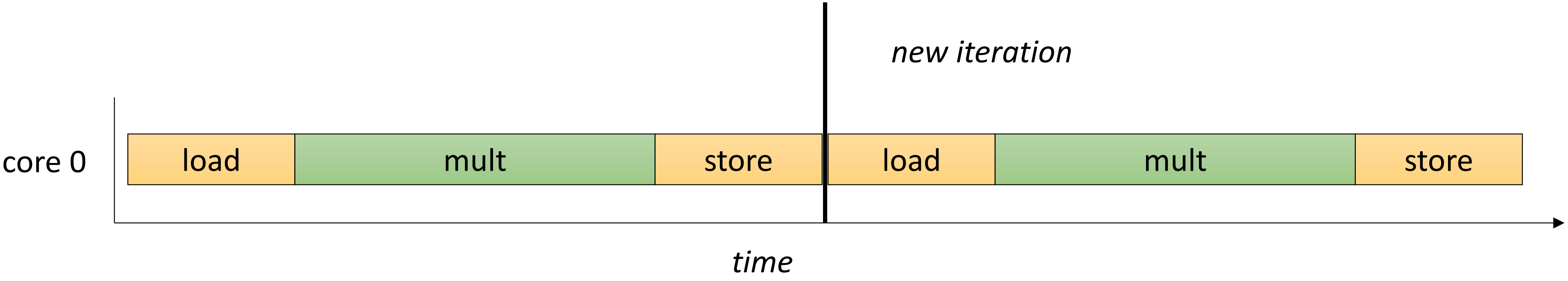
DAE - motivation



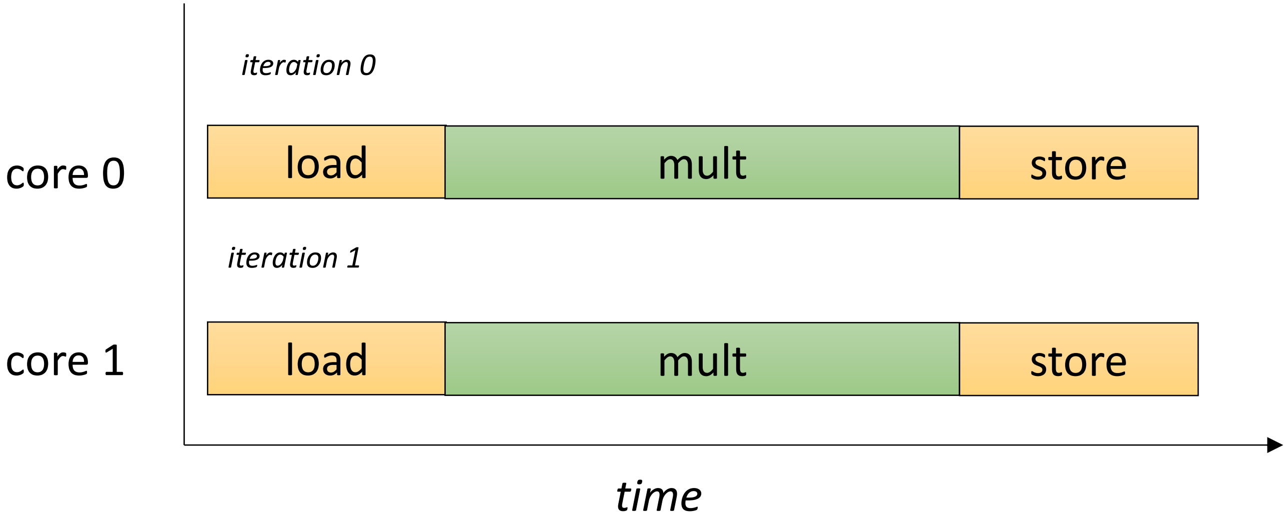
homogeneous SMP parallelism

pseudo 3-address code

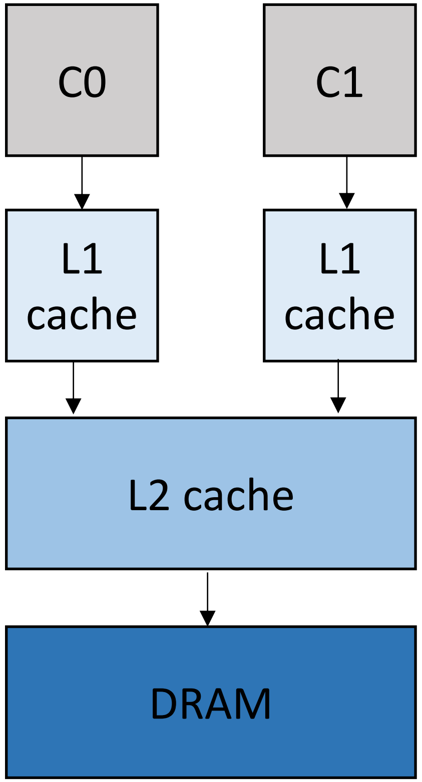
```
#pragma parallel
for (int i = 0; i < SIZE; i++) {
    float r0 = load(b + i);
    float r1 = r0 * 3.14;
    store(a + i, r1);
}
```

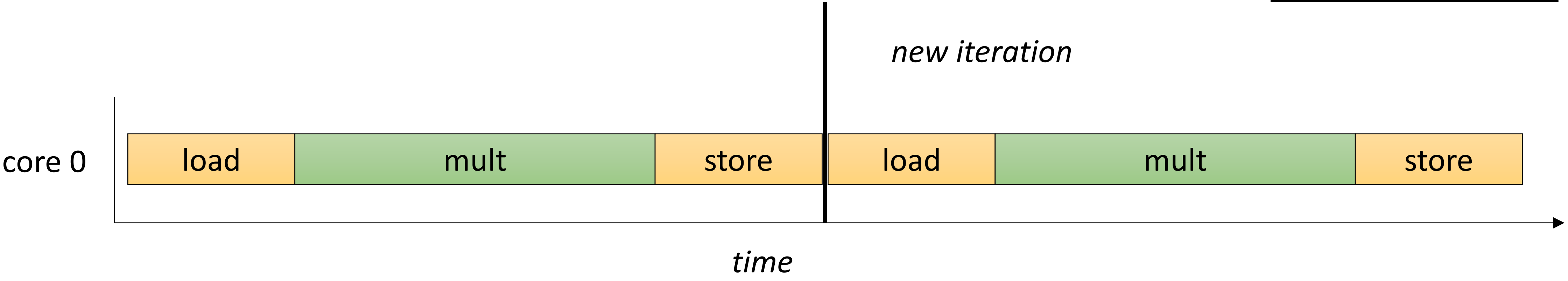


DAE - motivation

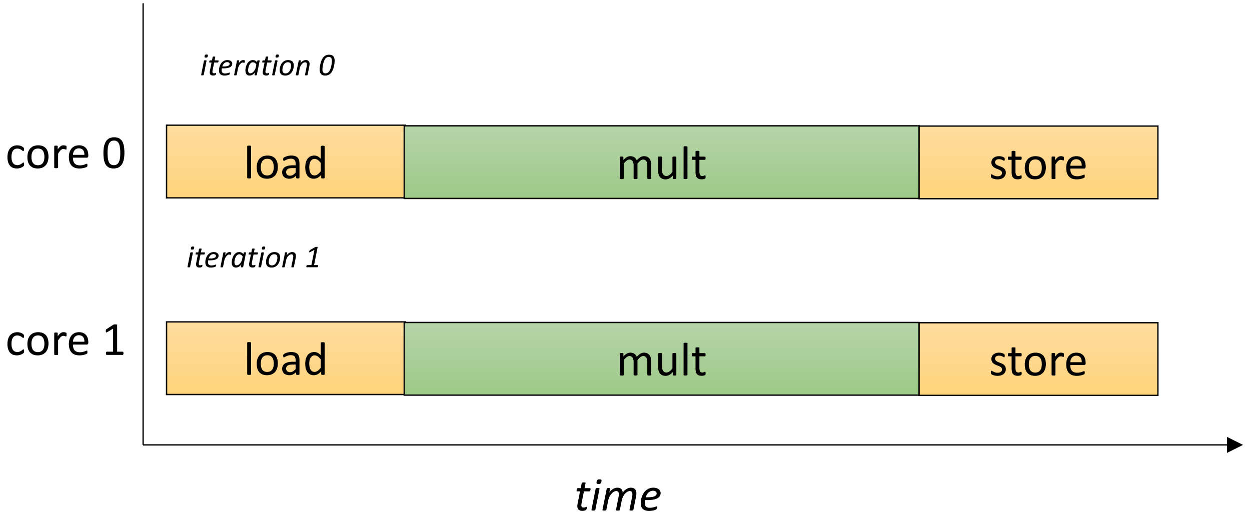


homogeneous SMP parallelism





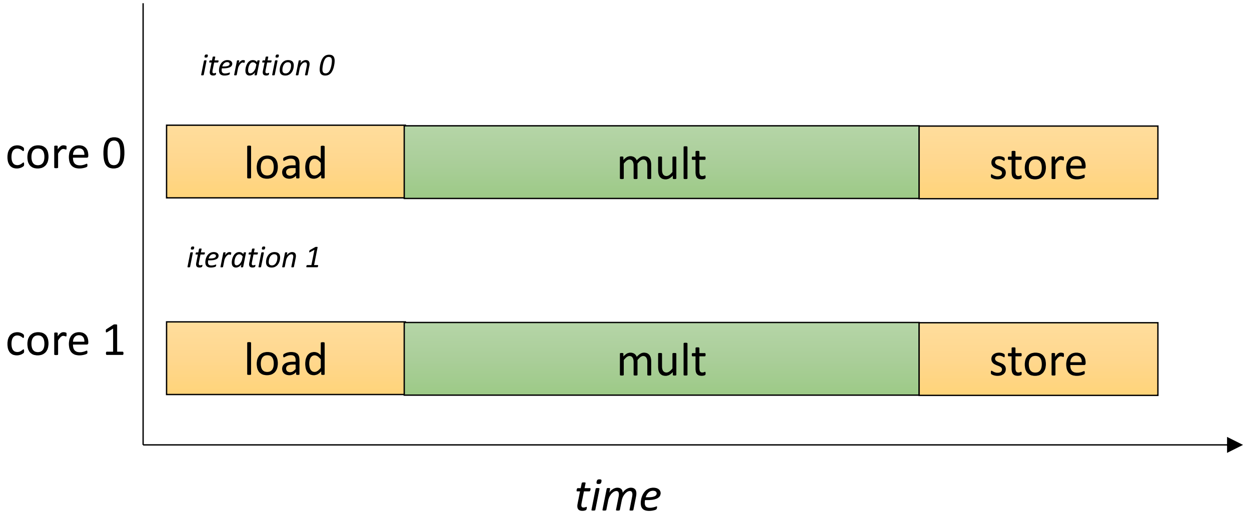
DAE - illustration



homogeneous SMP parallelism

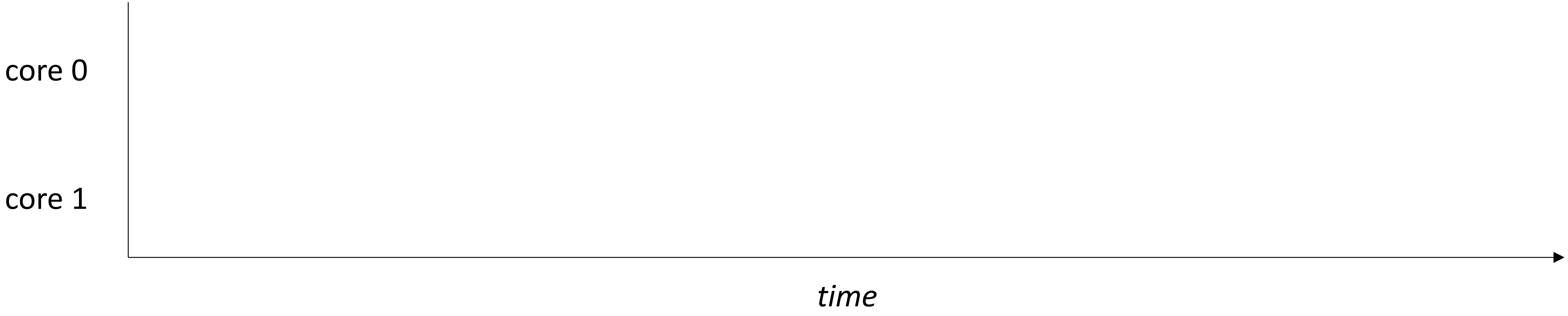
DAE: split into heterogeneous parallelism: one core does memory and one does computation

DAE - illustration

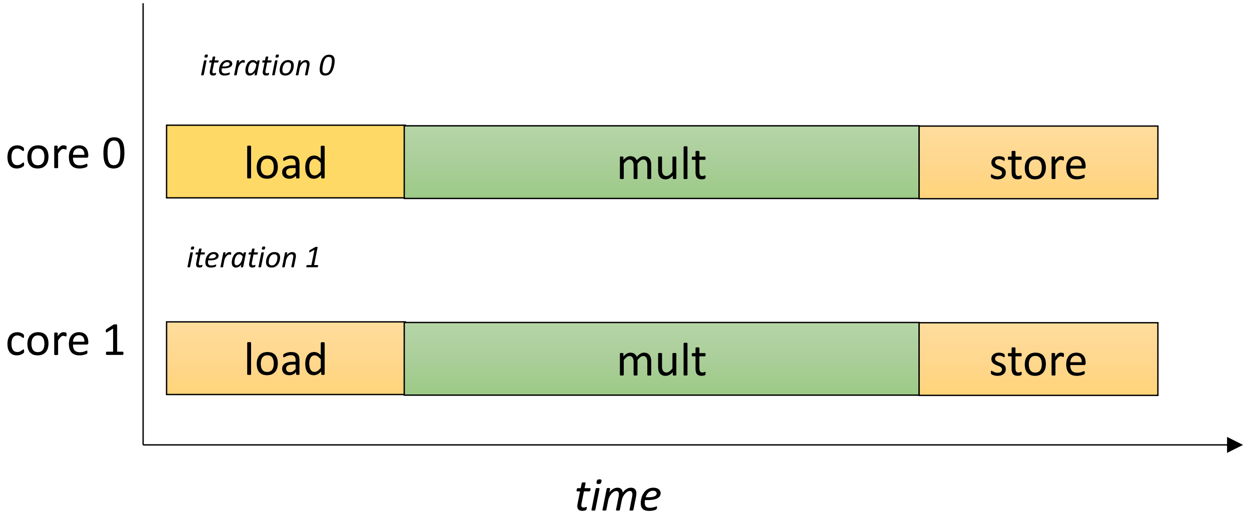


homogeneous SMP parallelism

DAE: split into heterogeneous parallelism: one core does memory and one does computation

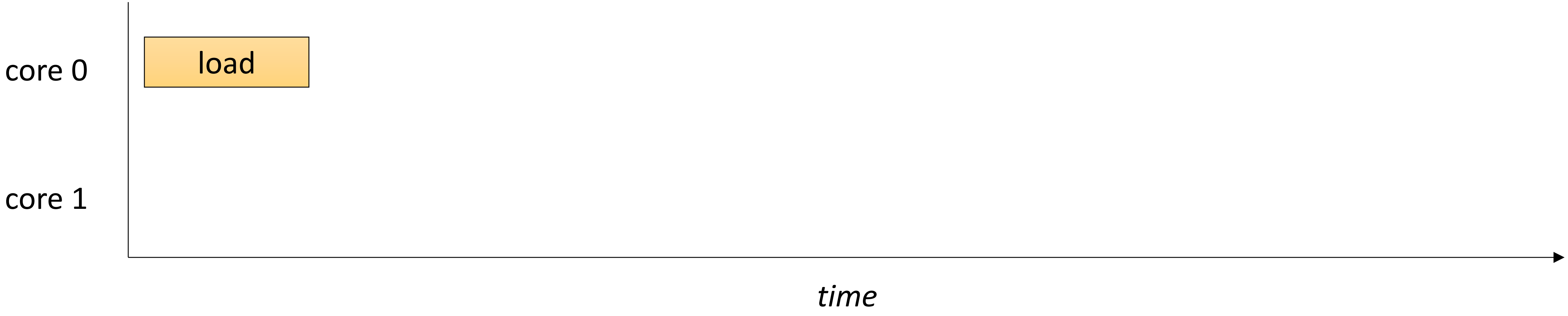


DAE - illustration

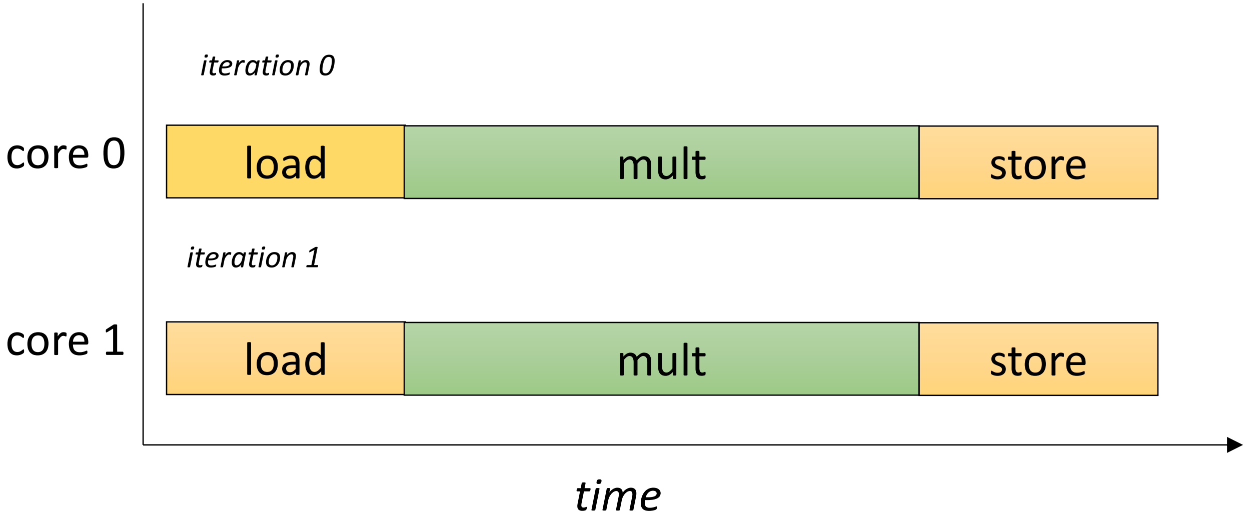


homogeneous SMP parallelism

DAE: split into heterogeneous parallelism: one core does memory and one does computation

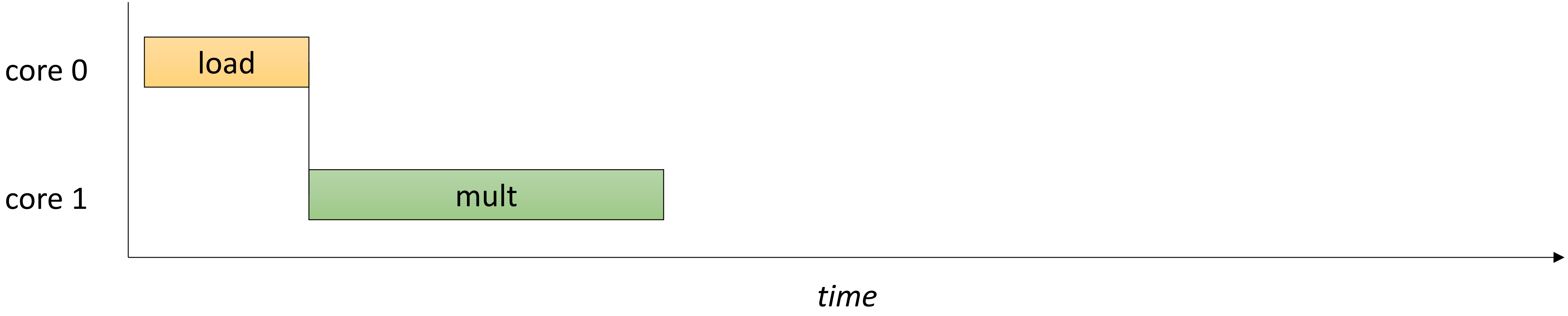


DAE - illustration

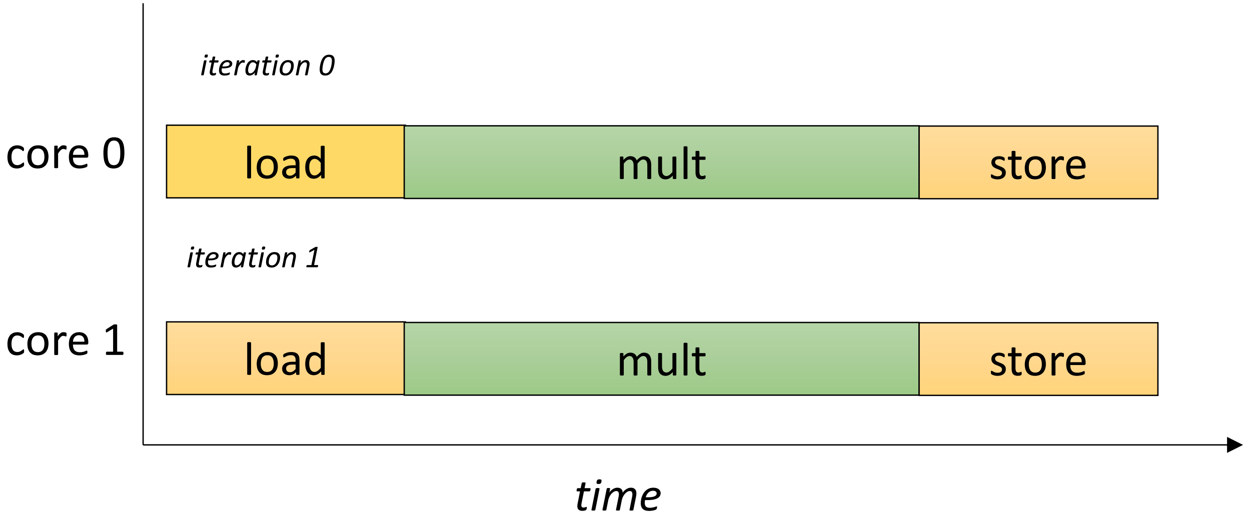


homogeneous SMP parallelism

DAE: split into heterogeneous parallelism: one core does memory and one does computation

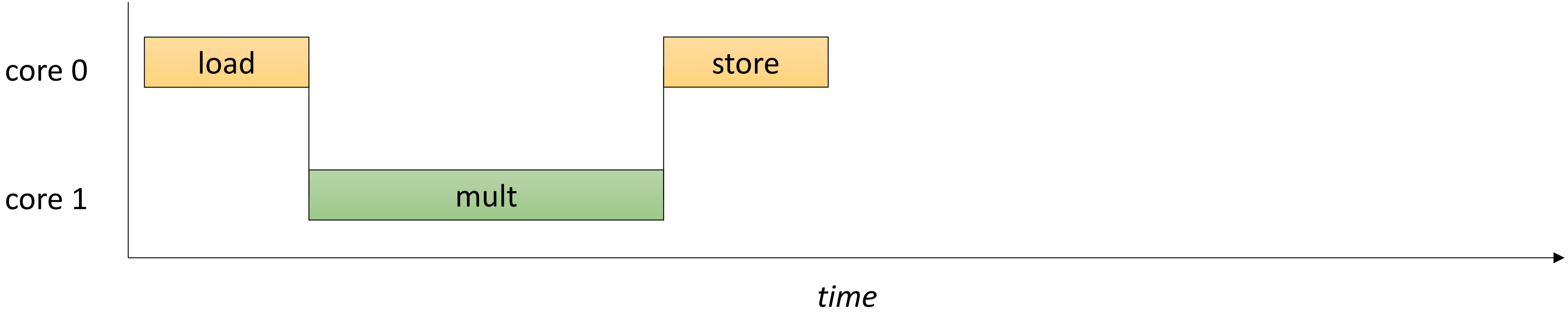


DAE - illustration

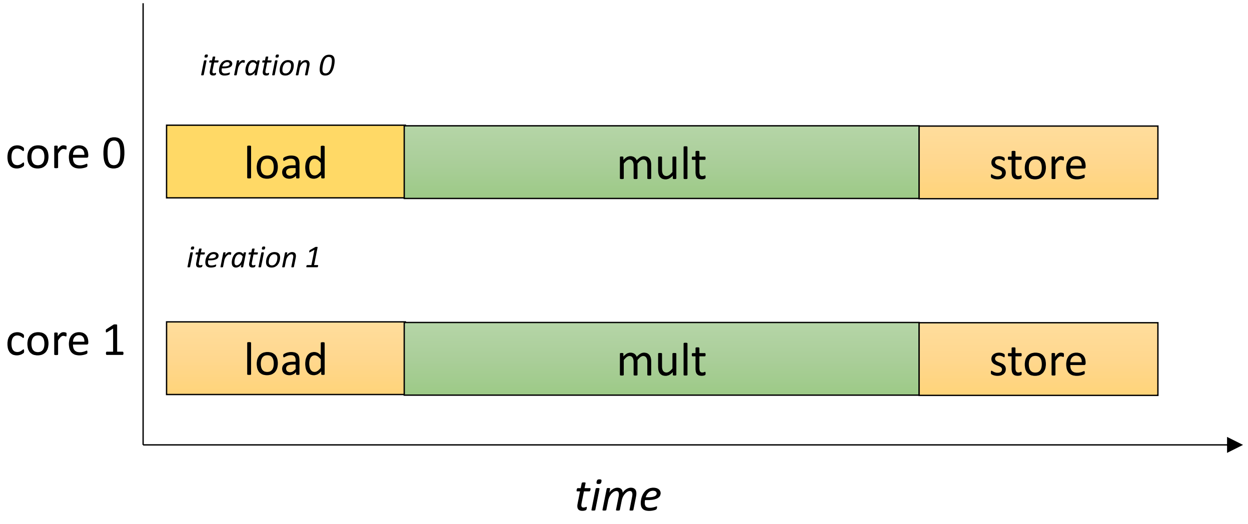


homogeneous SMP parallelism

DAE: split into heterogeneous parallelism: one core does memory and one does computation



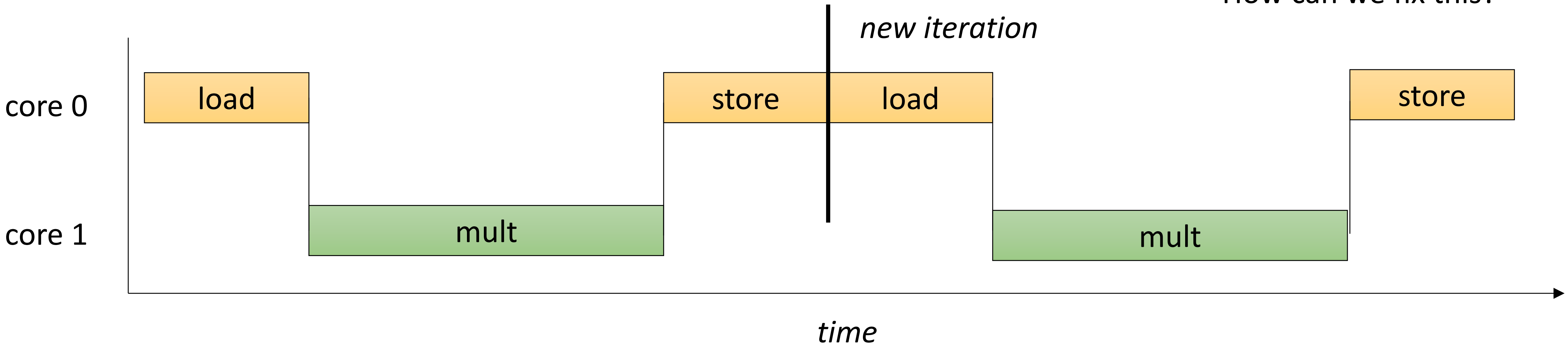
DAE - illustration



homogeneous SMP parallelism

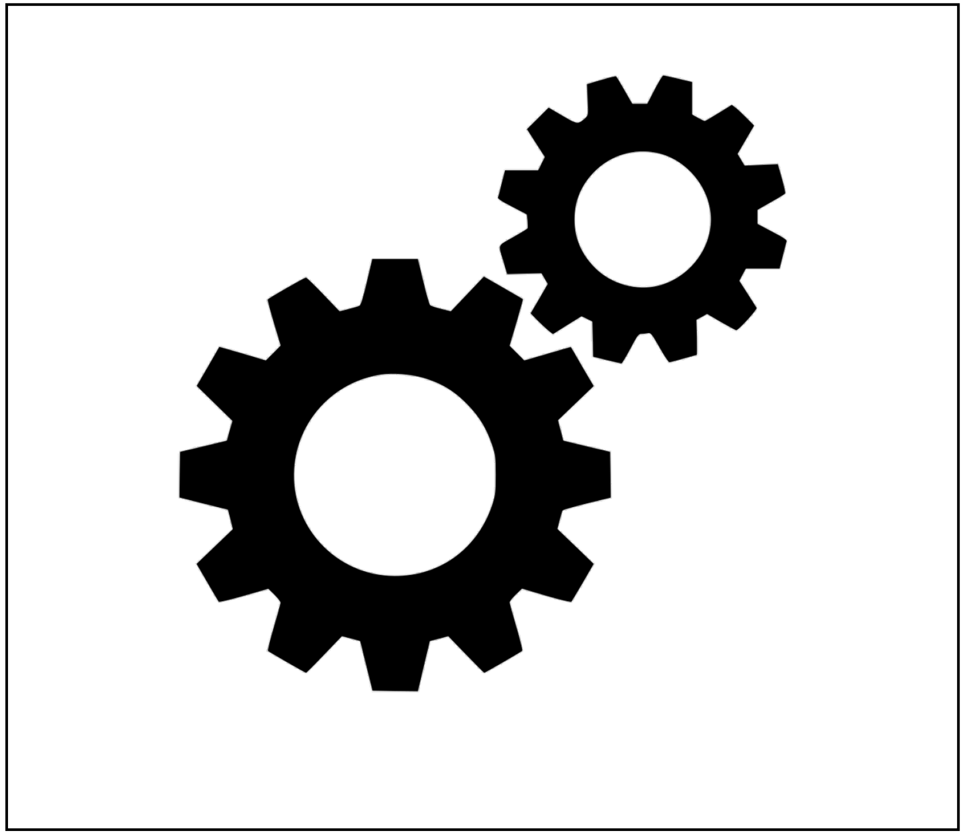
DAE: split into heterogeneous parallelism: one core does memory and one does computation

This is sequentialized ☹
How can we fix this?

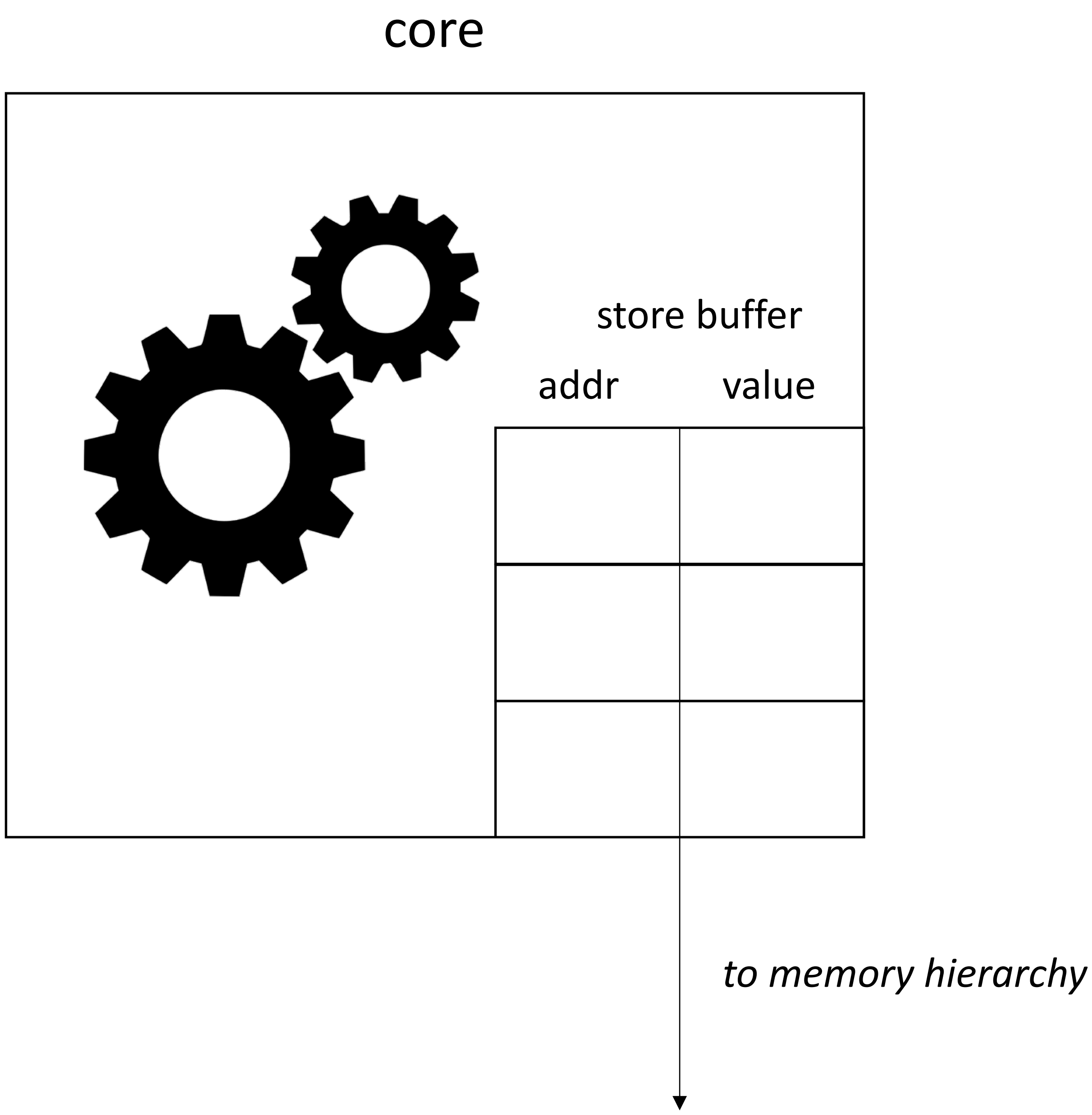


Store Buffers

core

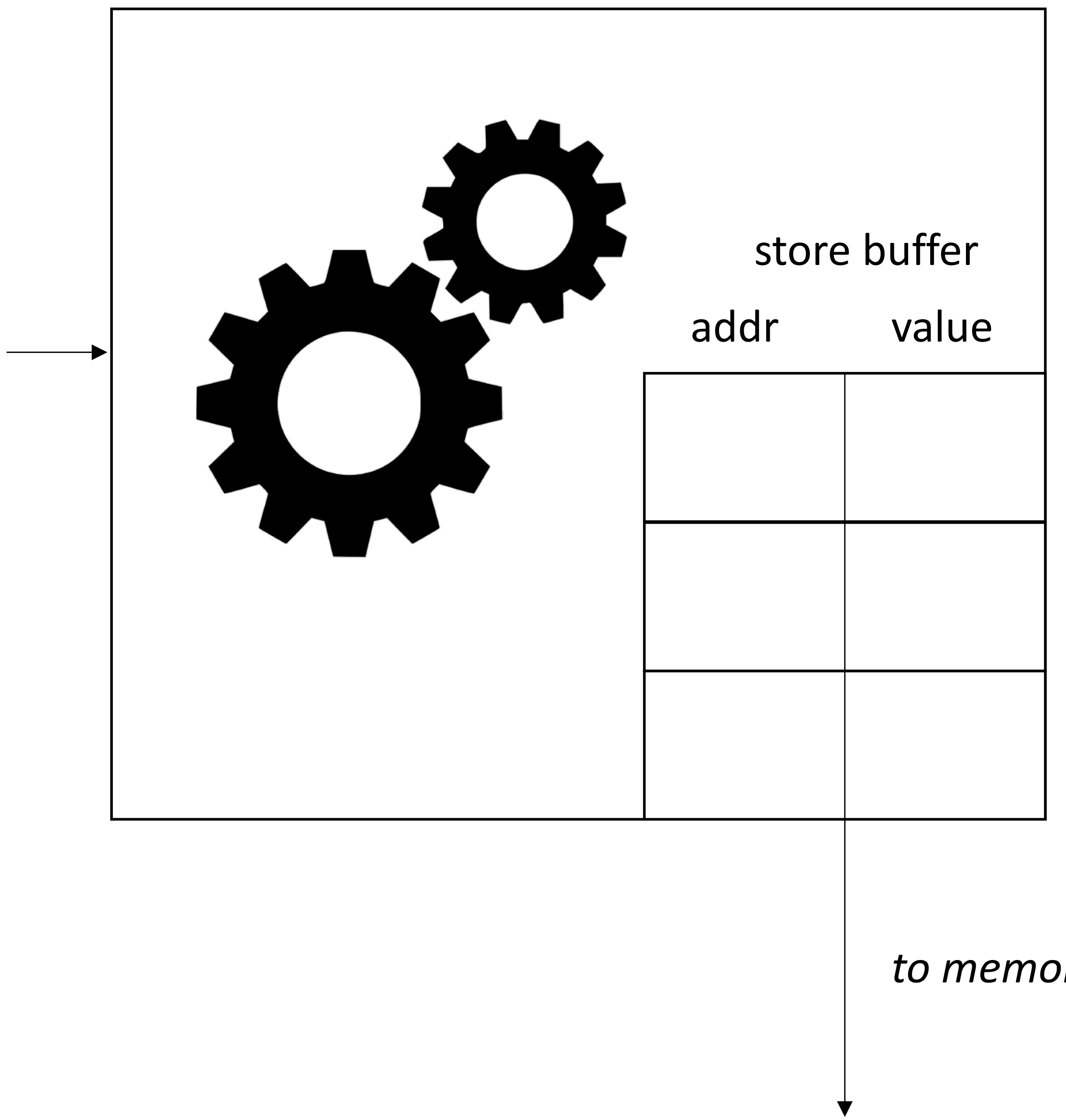


Store Buffers

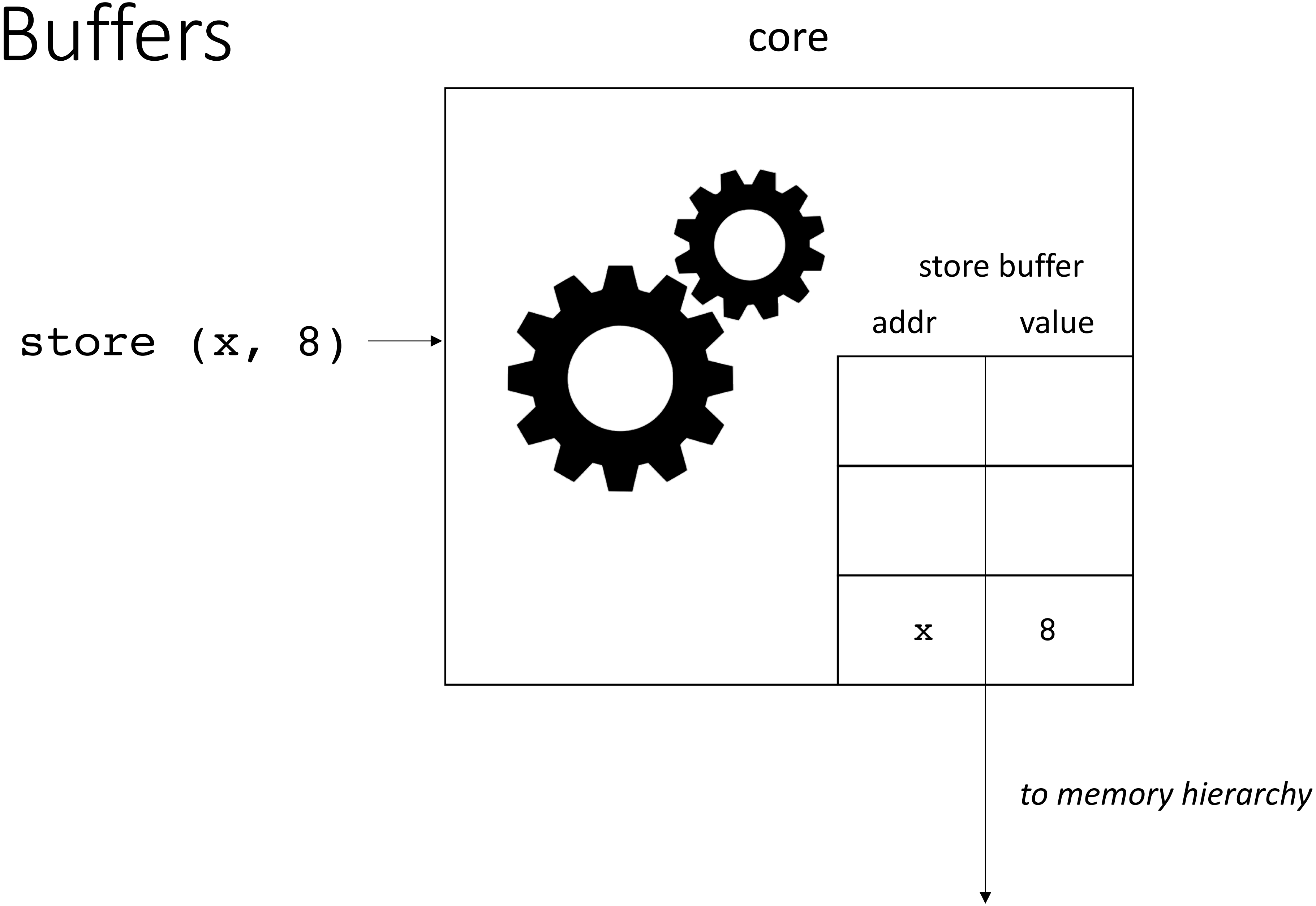


Store Buffers

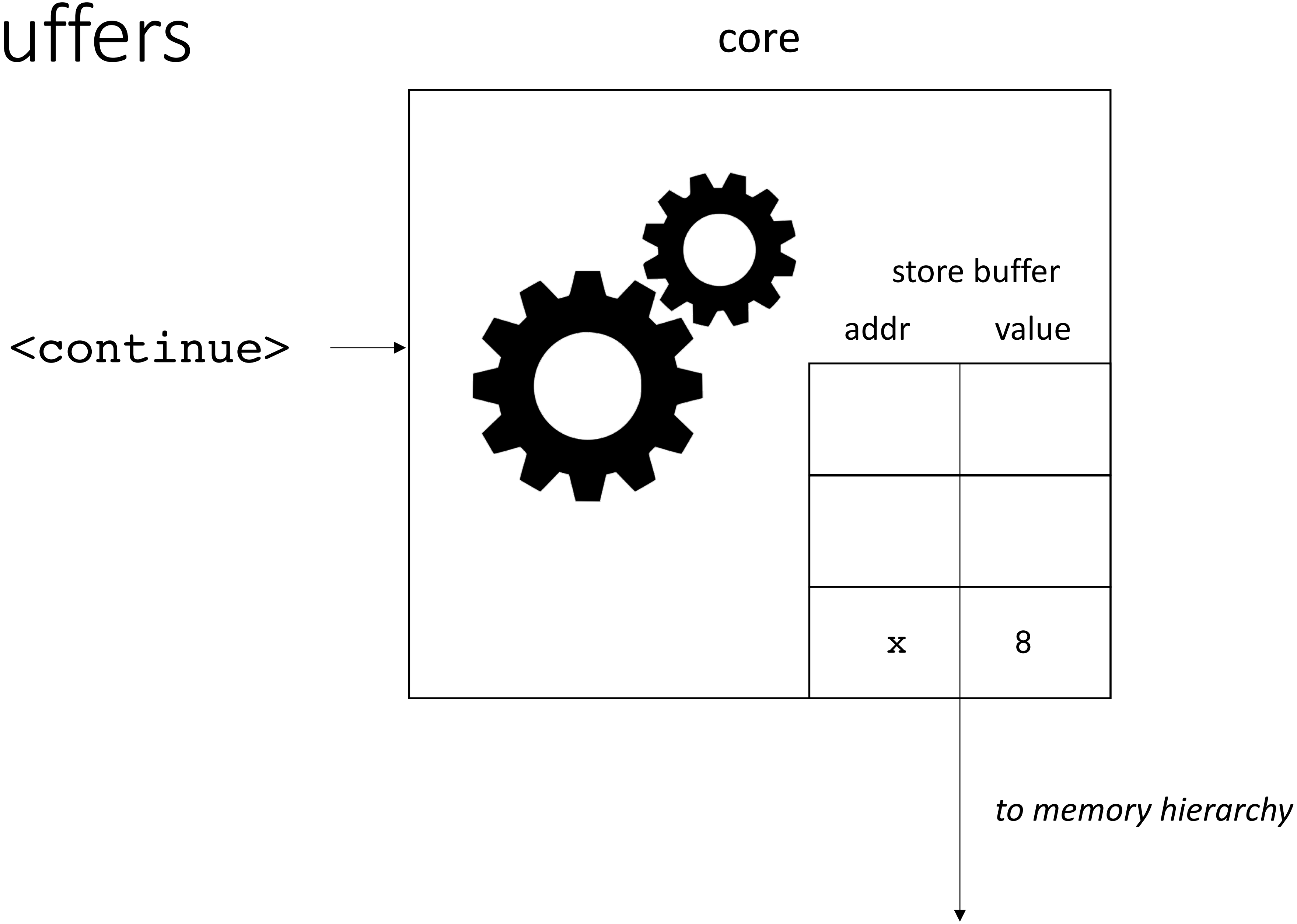
`store (x, 8)`



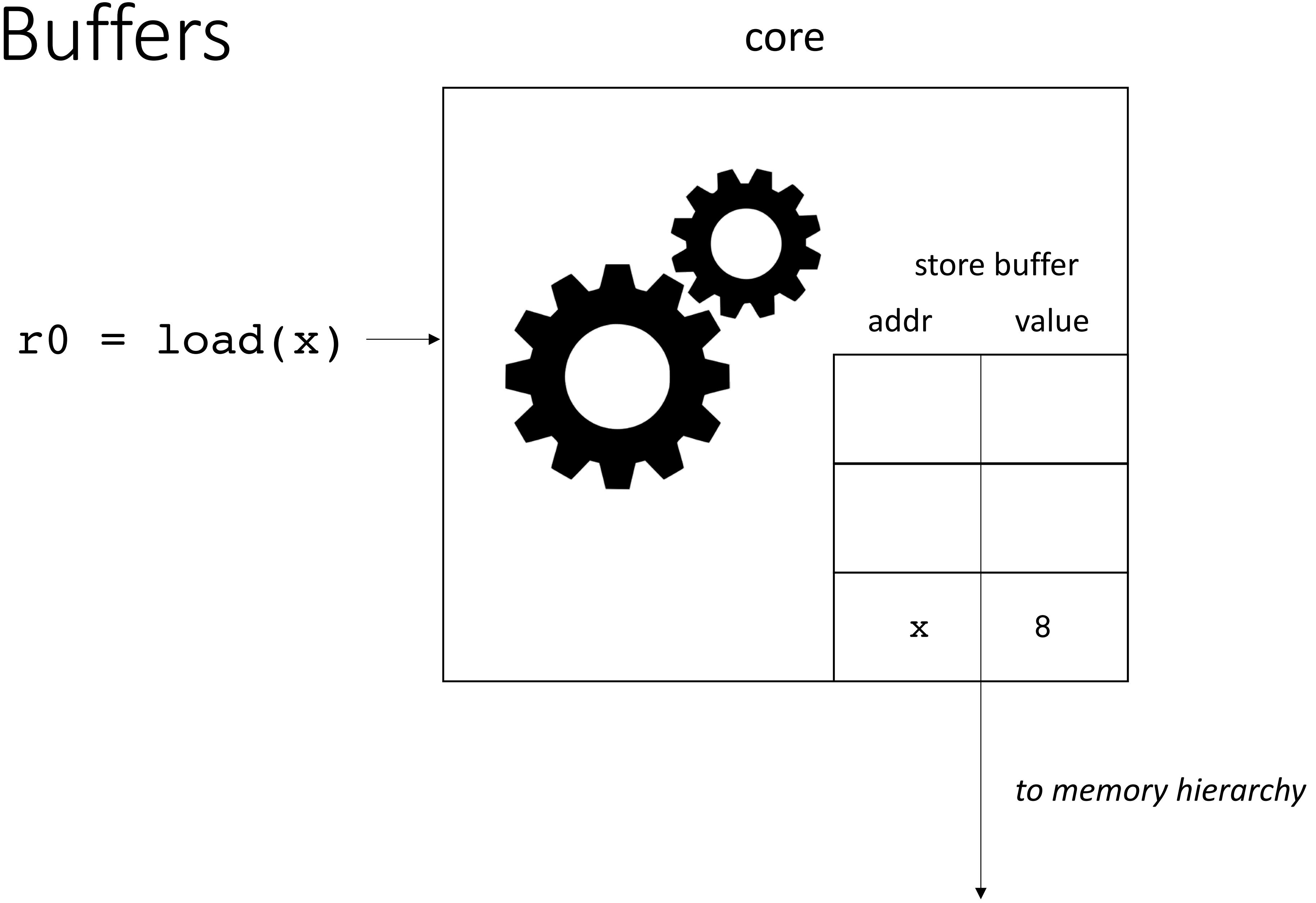
Store Buffers



Store Buffers



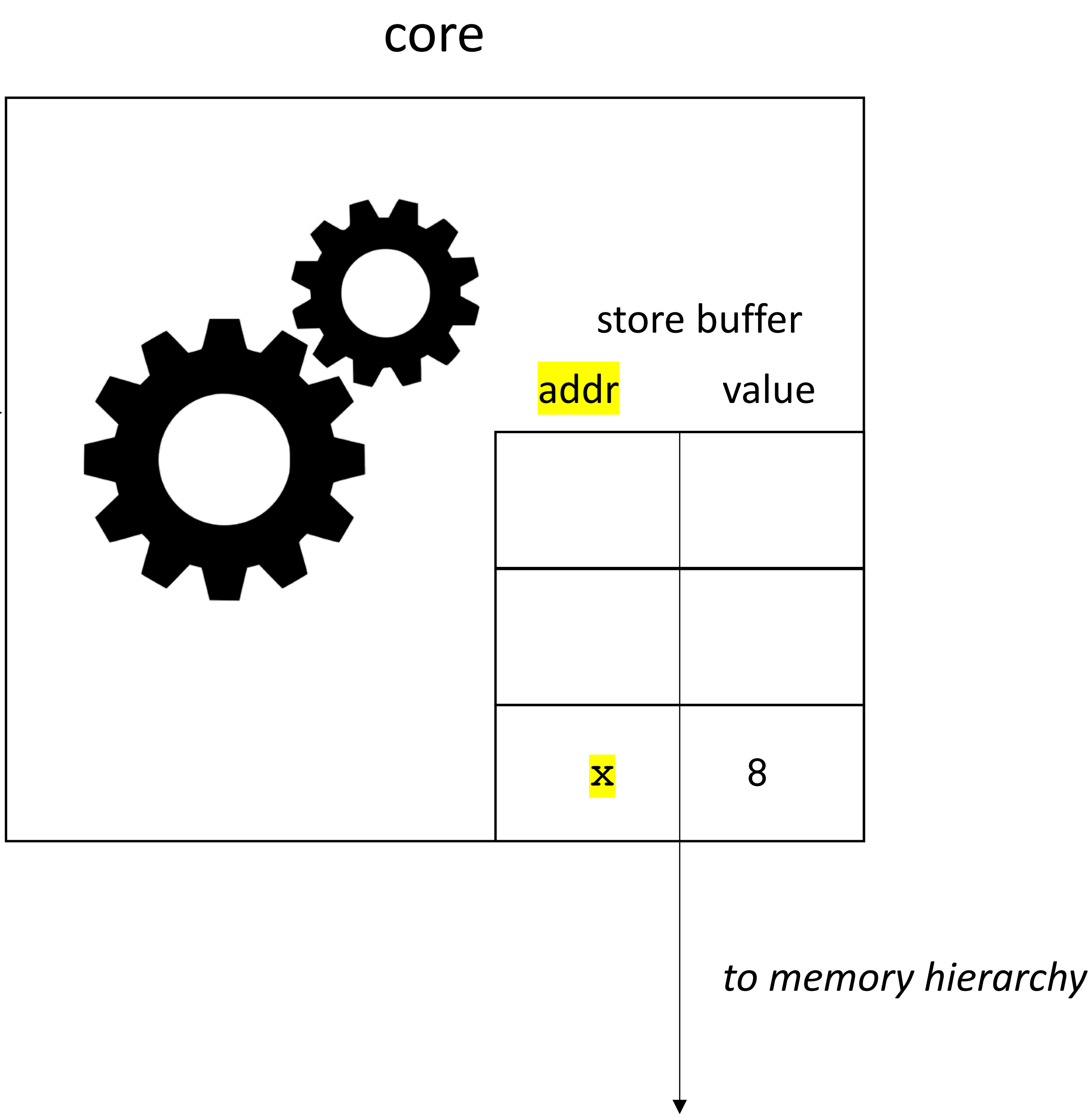
Store Buffers



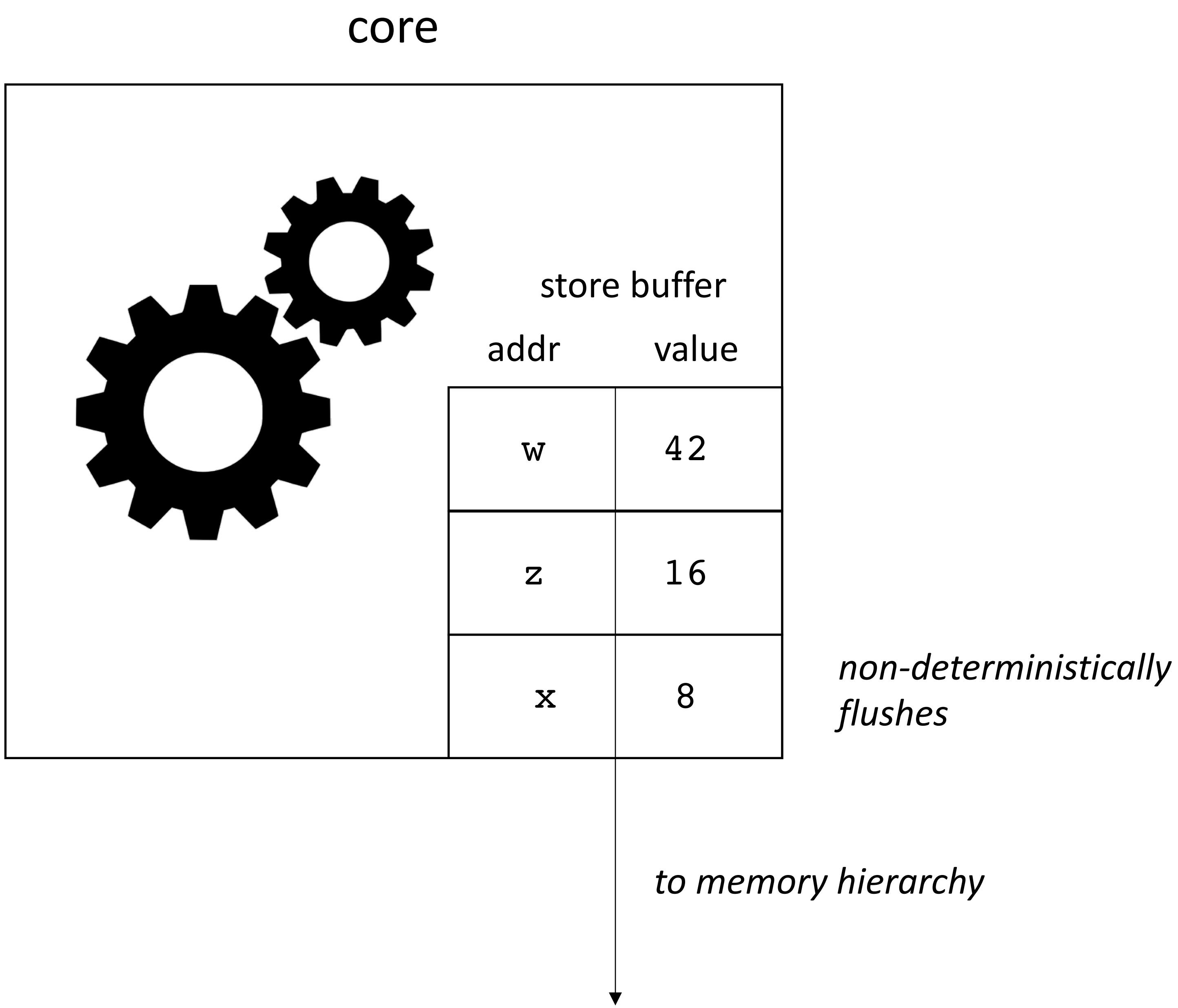
Store Buffers

`r0 = load(x)`

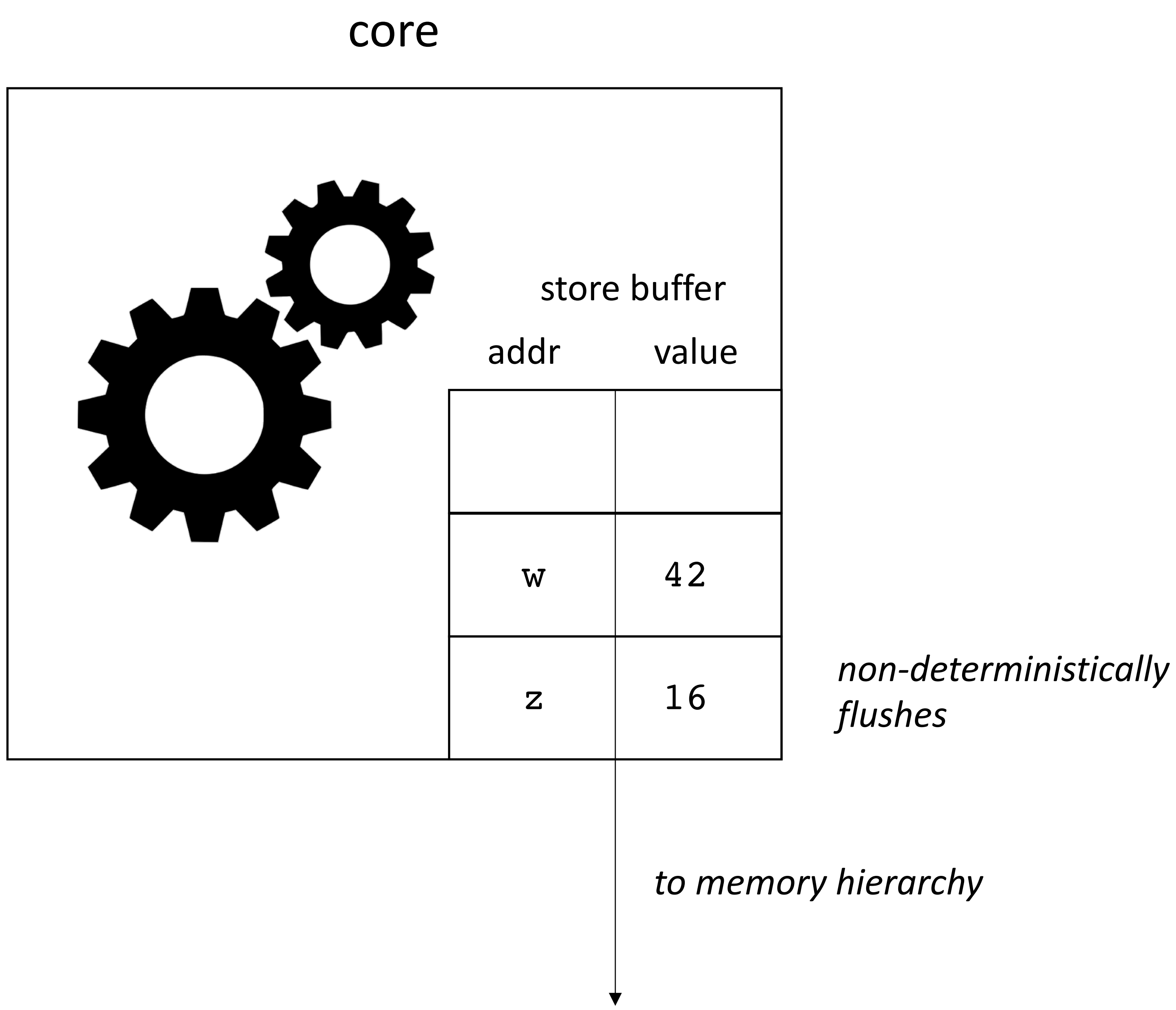
check store buffer
before going to memory



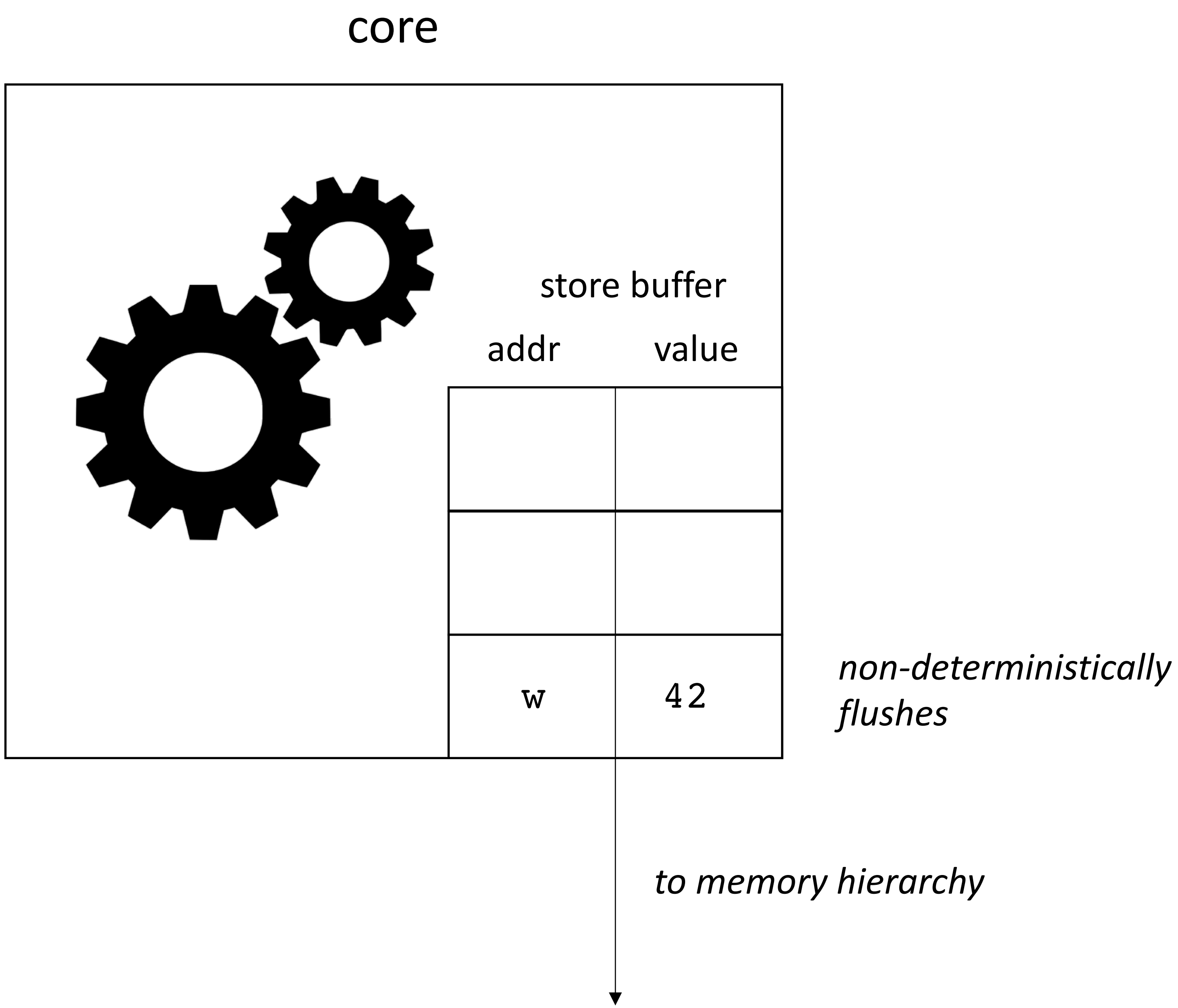
Store Buffers



Store Buffers

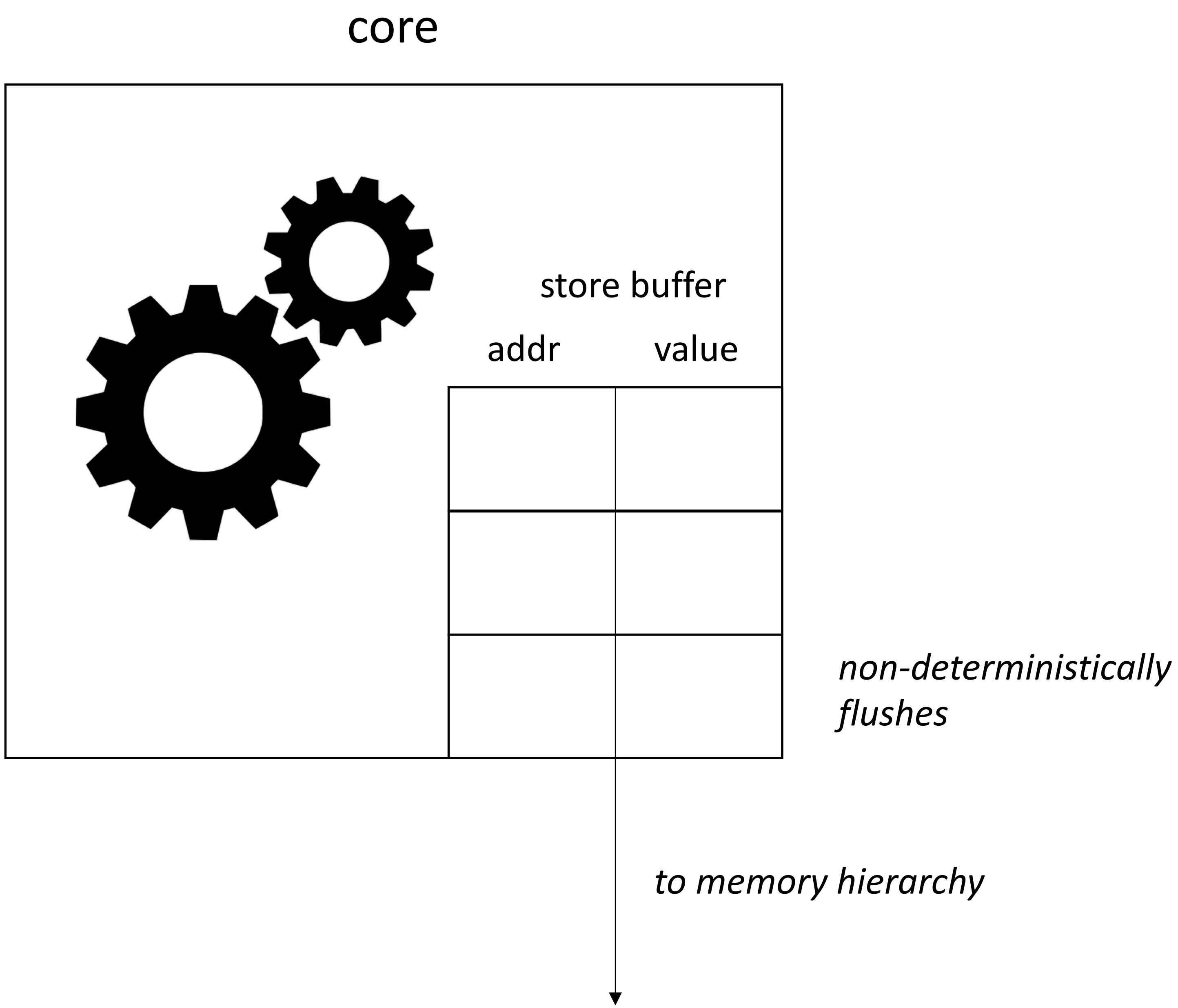


Store Buffers

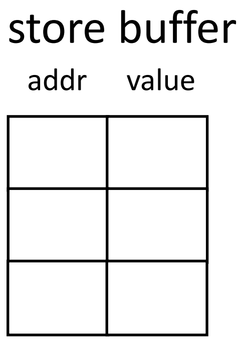


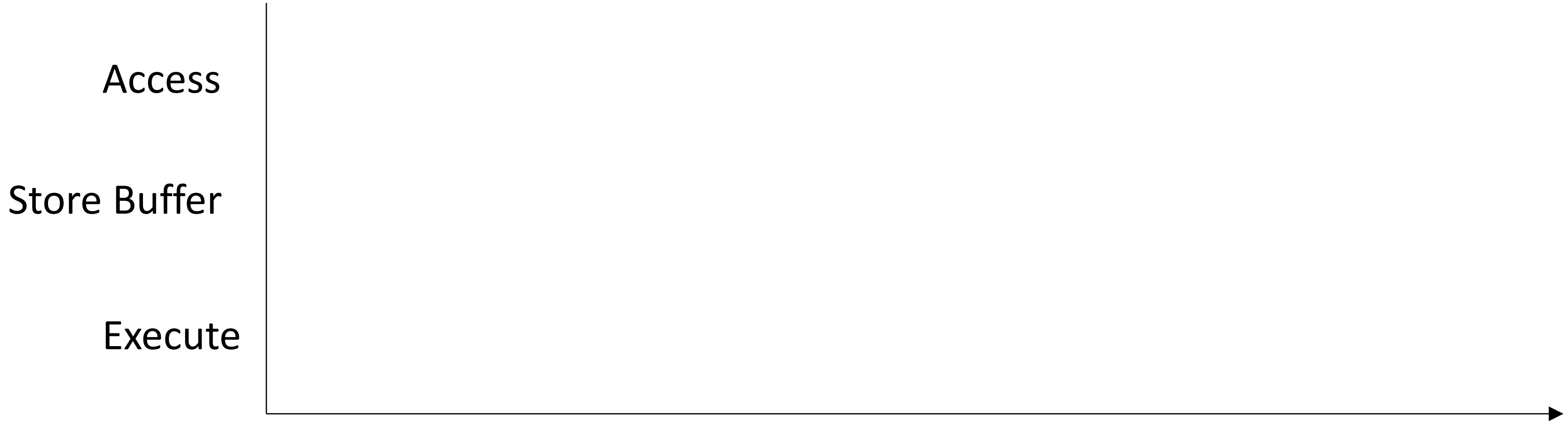
Store Buffers

key insight:
Store buffers allow asynchronous stores!

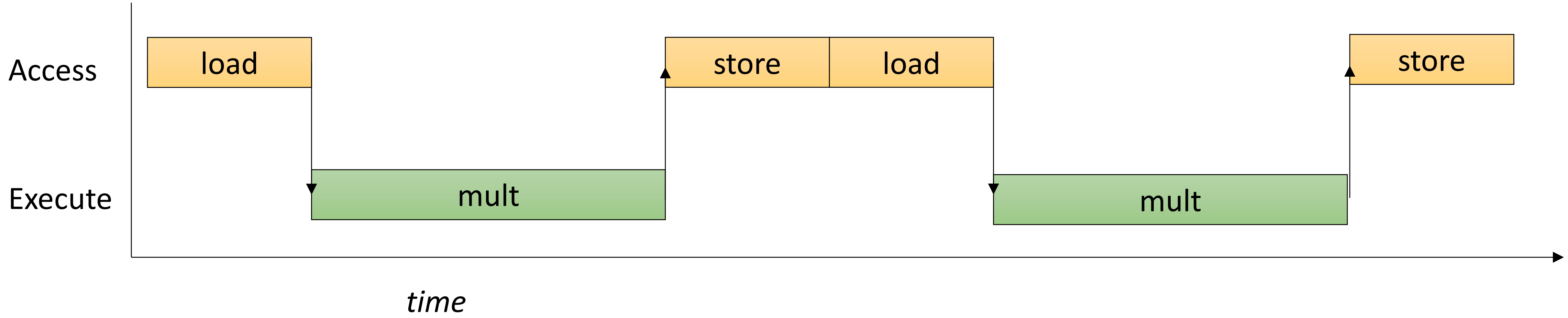


DAE Parallelism

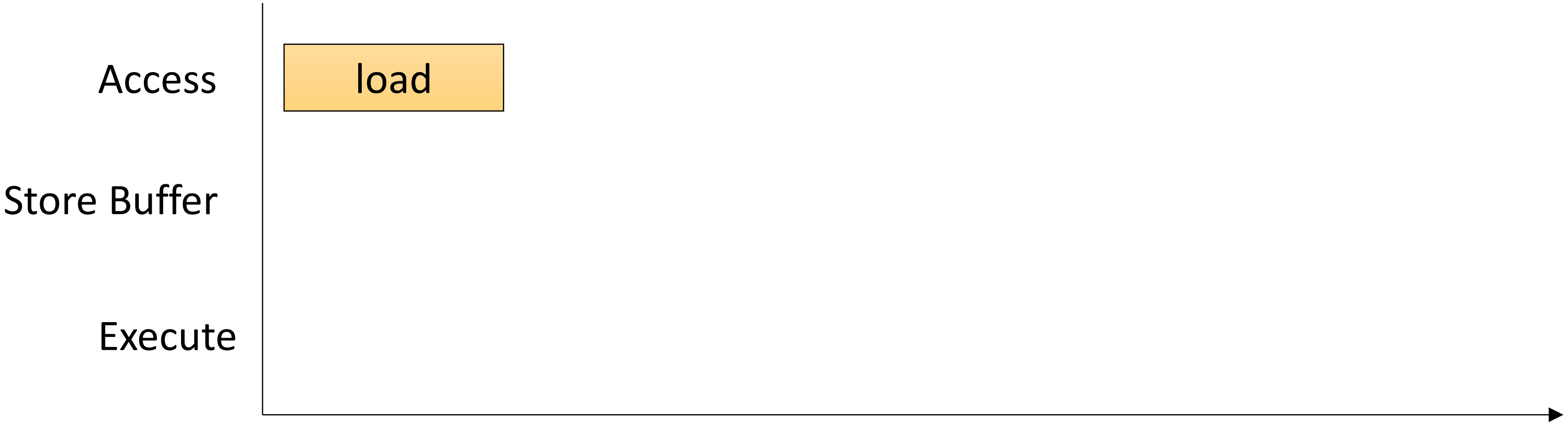





```
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   float r1 = r0 * 3.14;
   store(a + i, r1);
}
```



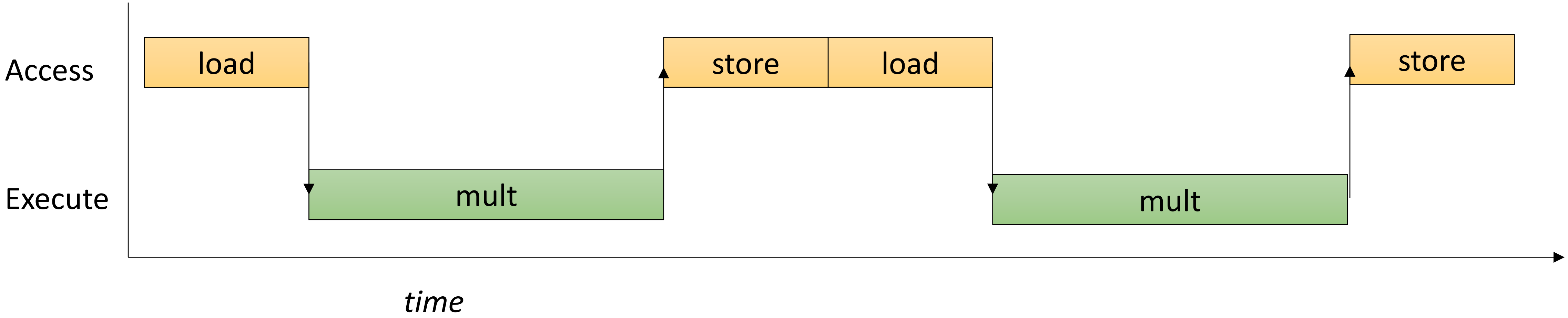
DAE Parallelism



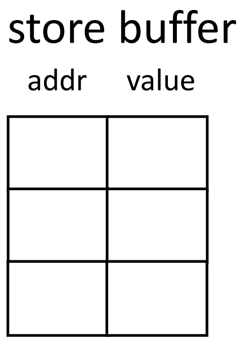
store buffer

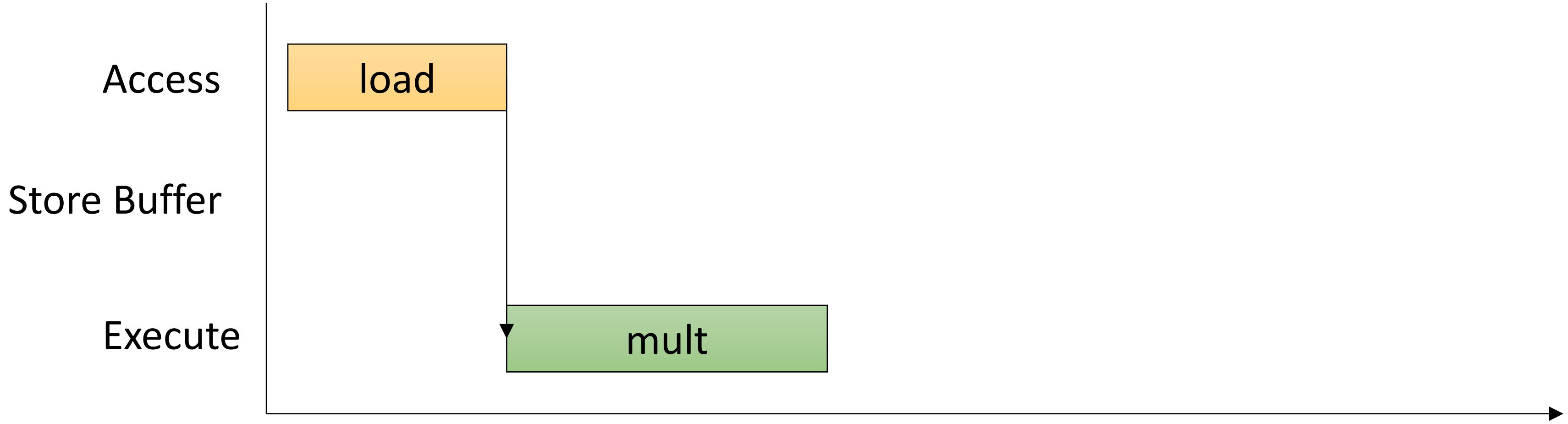
addr	value

```
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   float r1 = r0 * 3.14;
   store(a + i, r1);
}
```

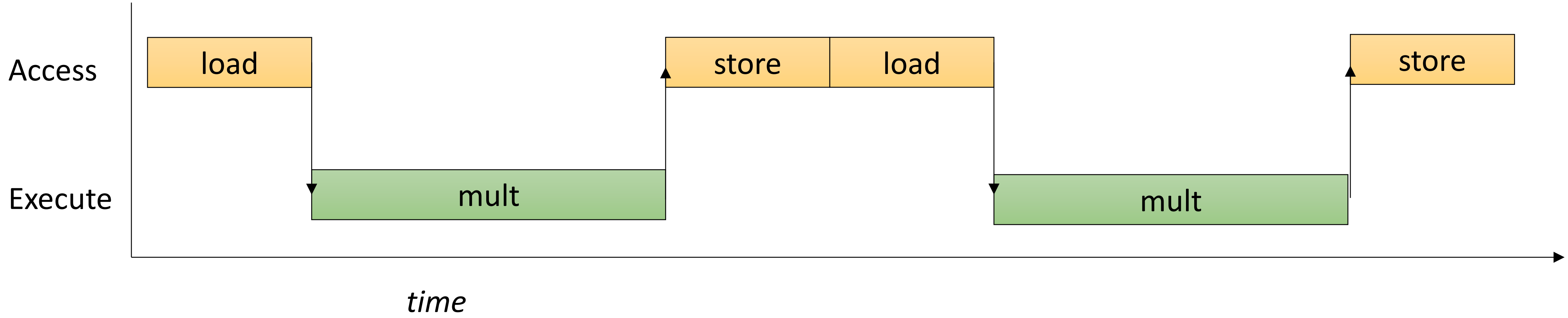


DAE Parallelism

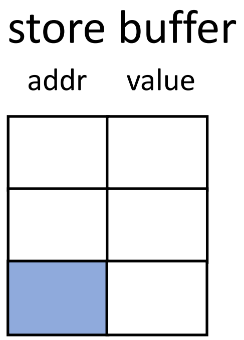


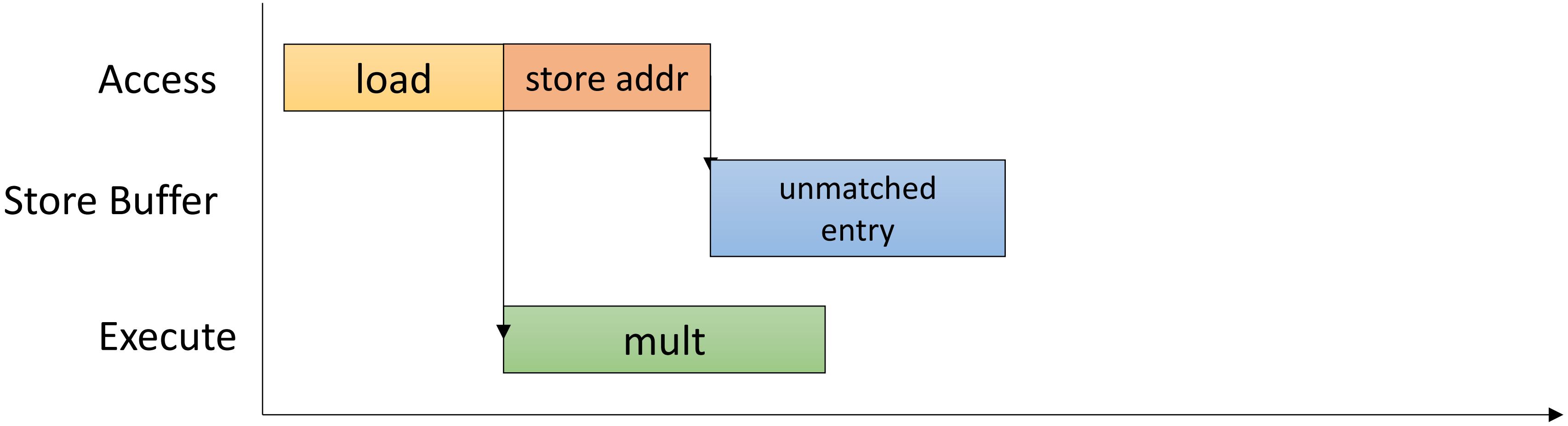



```
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   float r1 = r0 * 3.14;
   store(a + i, r1);
}
```

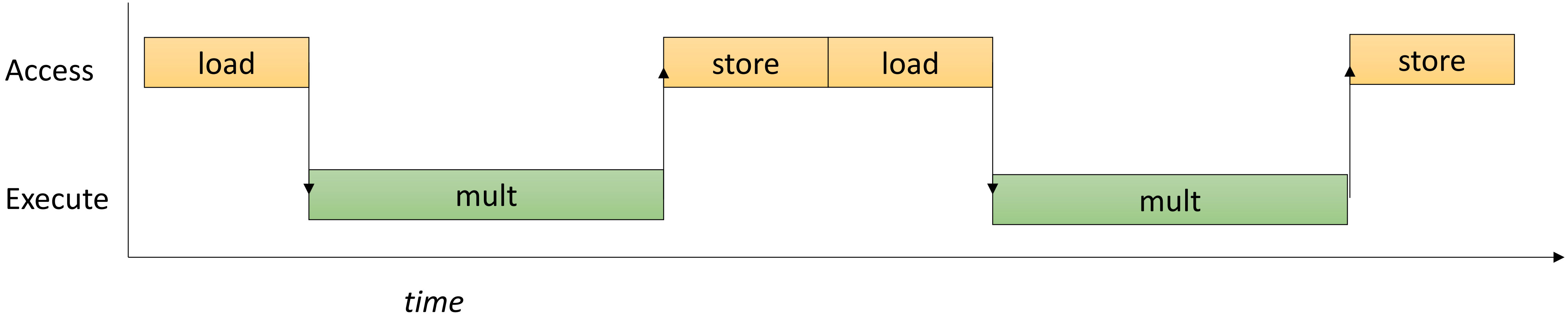


DAE Parallelism

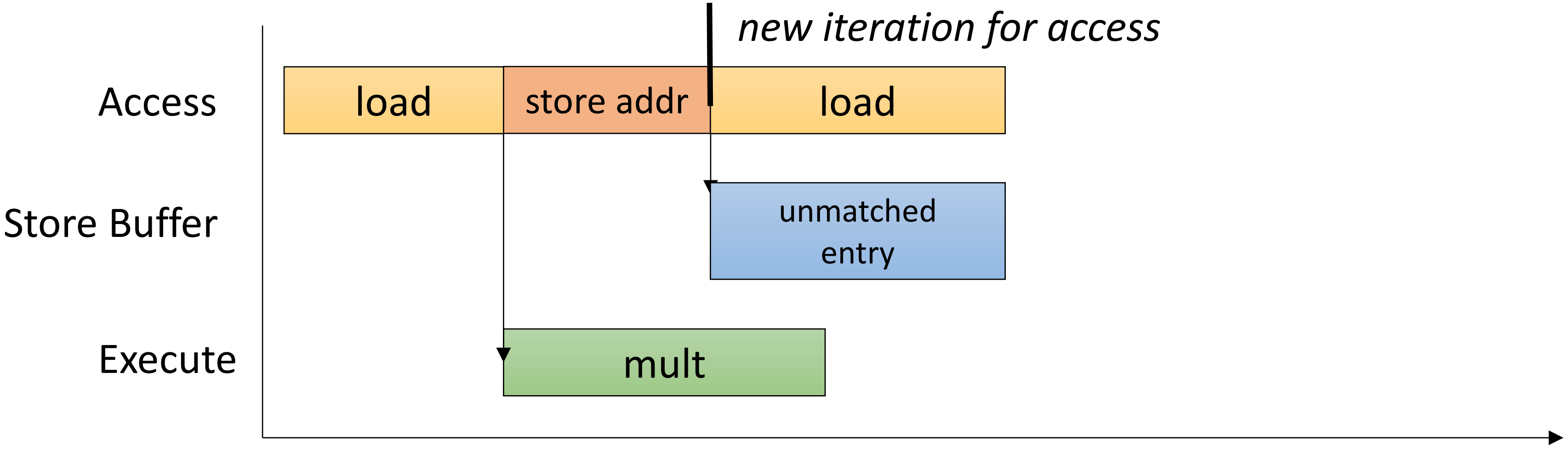


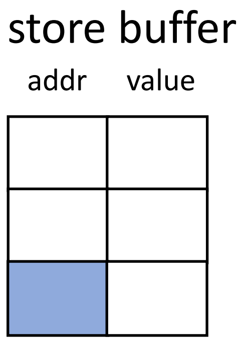



```
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   float r1 = r0 * 3.14;
   store(a + i, r1);
}
```

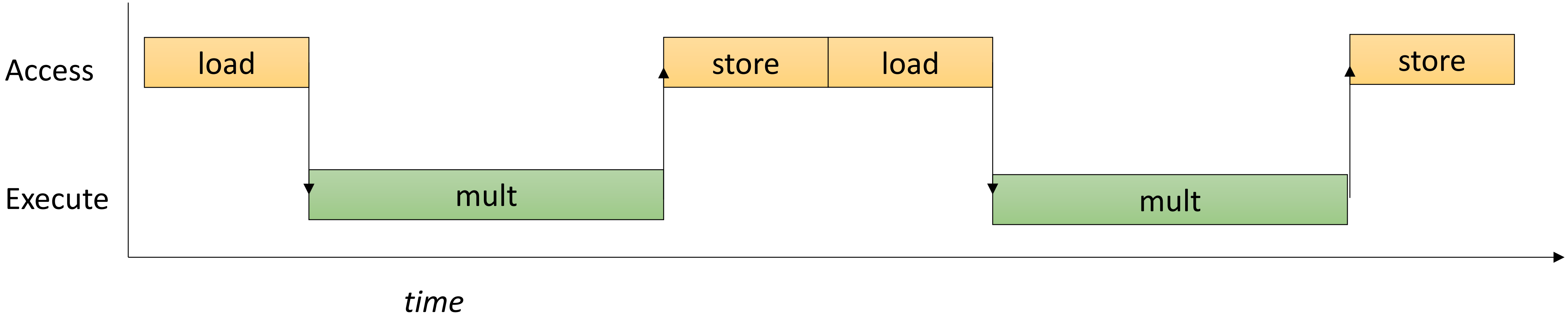


DAE Parallelism

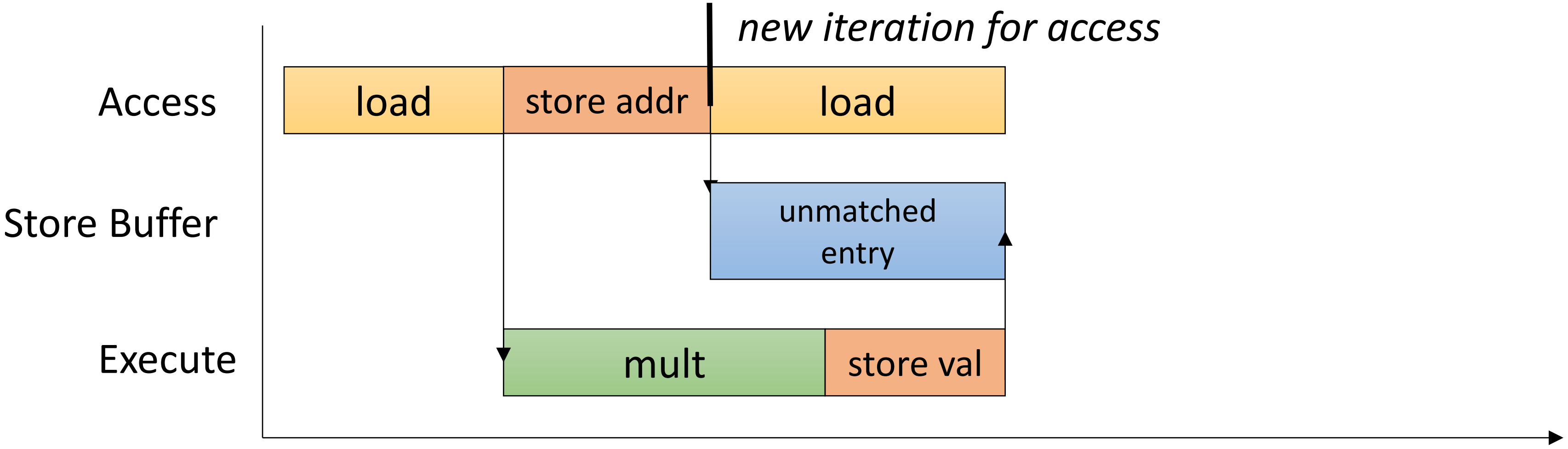


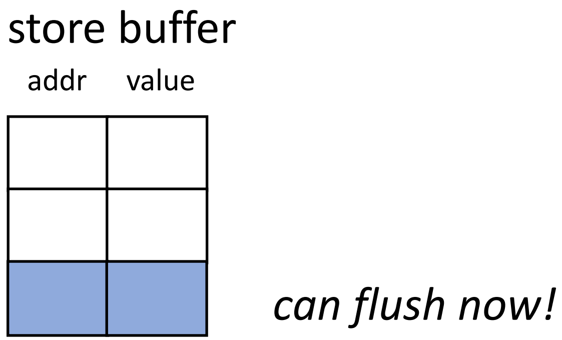



```
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   float r1 = r0 * 3.14;
   store(a + i, r1);
}
```

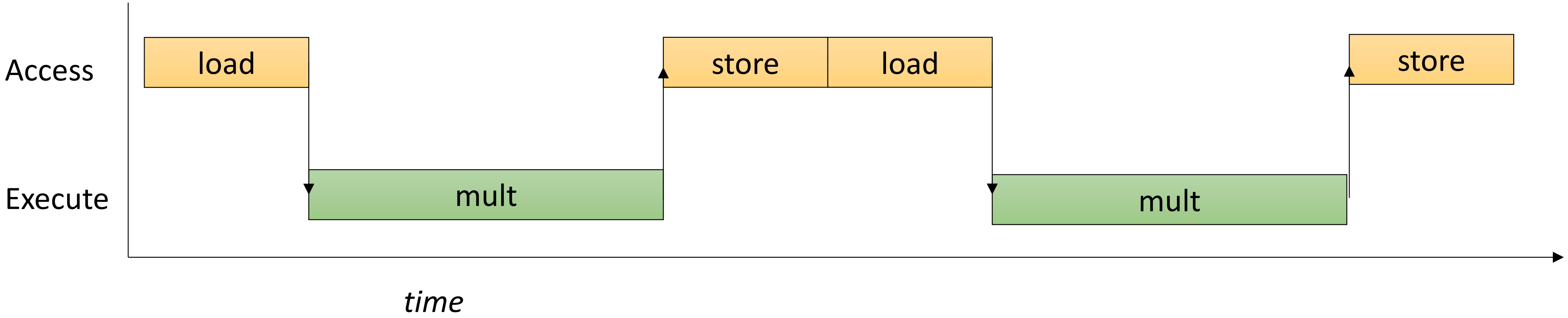


DAE Parallelism

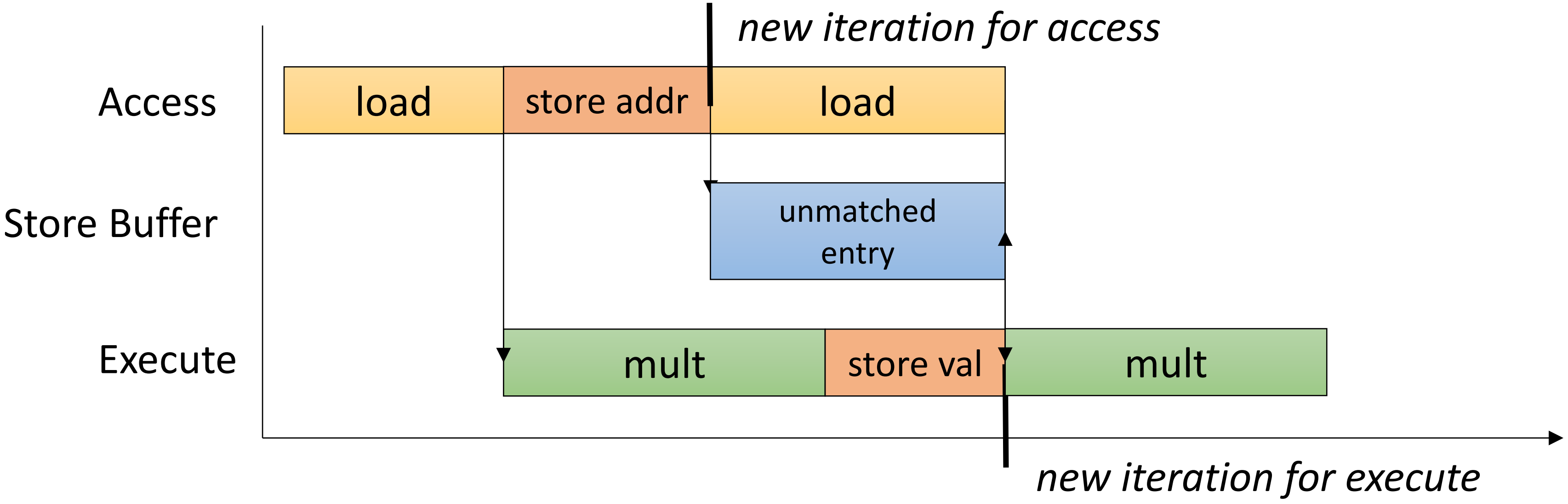


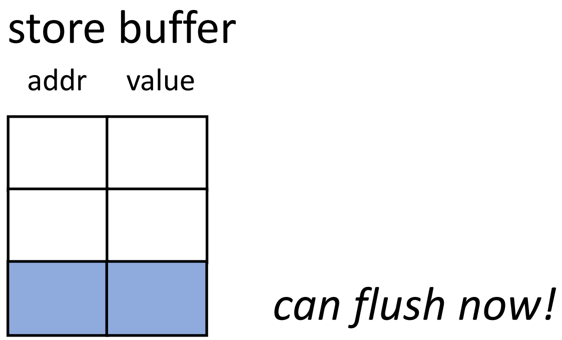



```
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   float r1 = r0 * 3.14;
   store(a + i, r1);
}
```

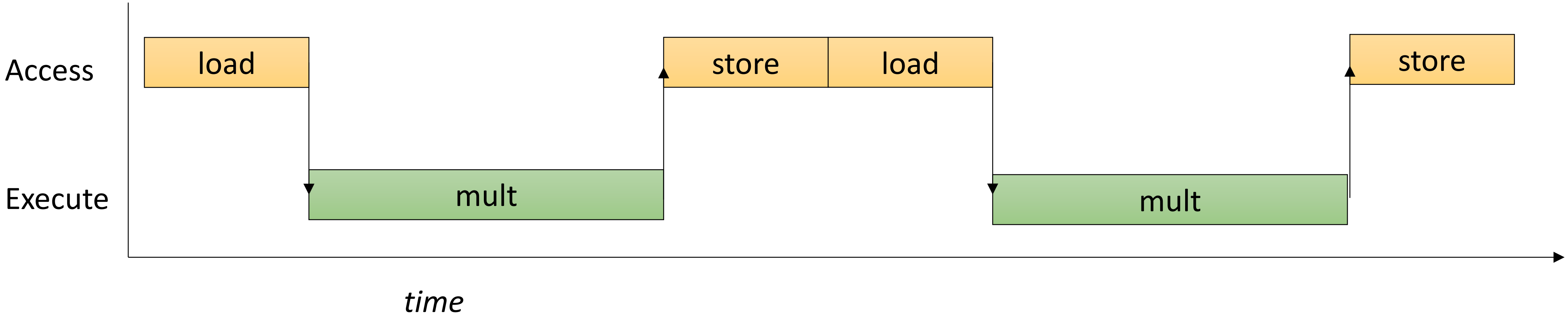


DAE Parallelism

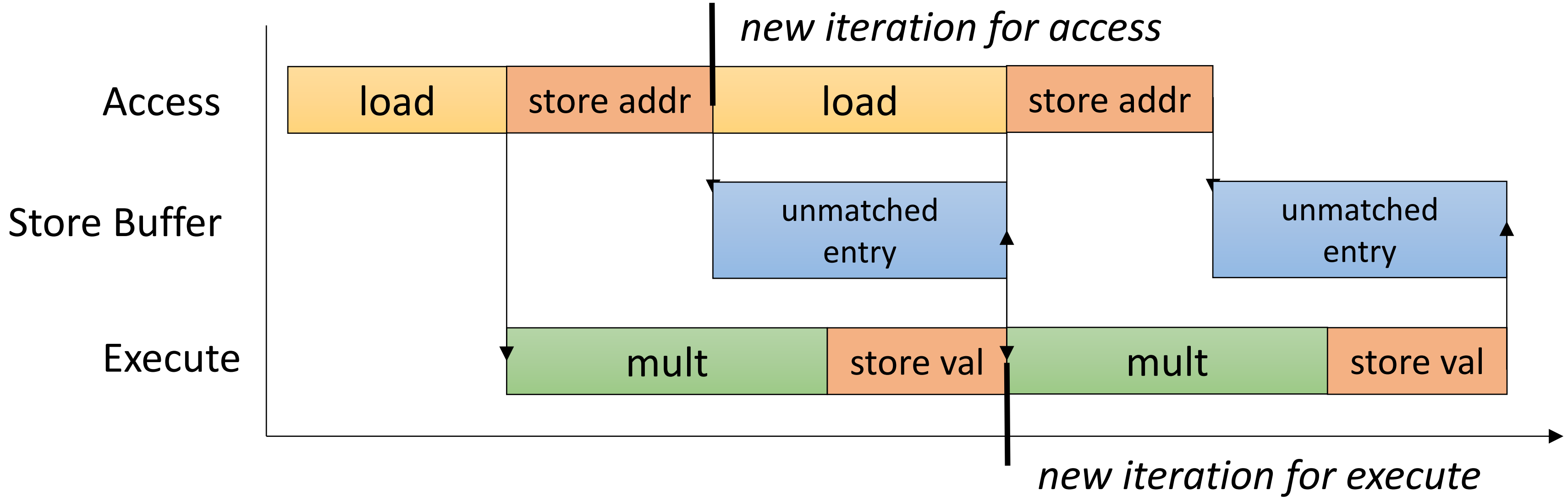


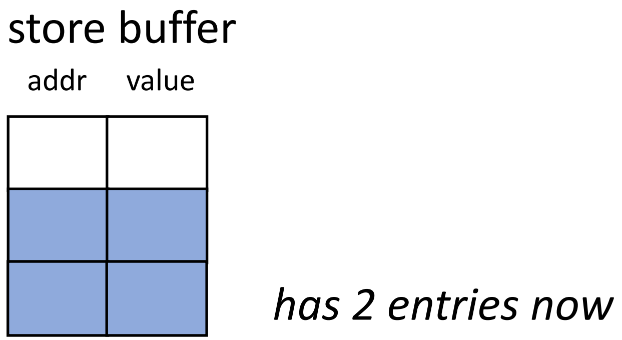



```
for (int i = 0; i < SIZE; i++) {
    float r0 = load(b + i);
    float r1 = r0 * 3.14;
    store(a + i, r1);
}
```

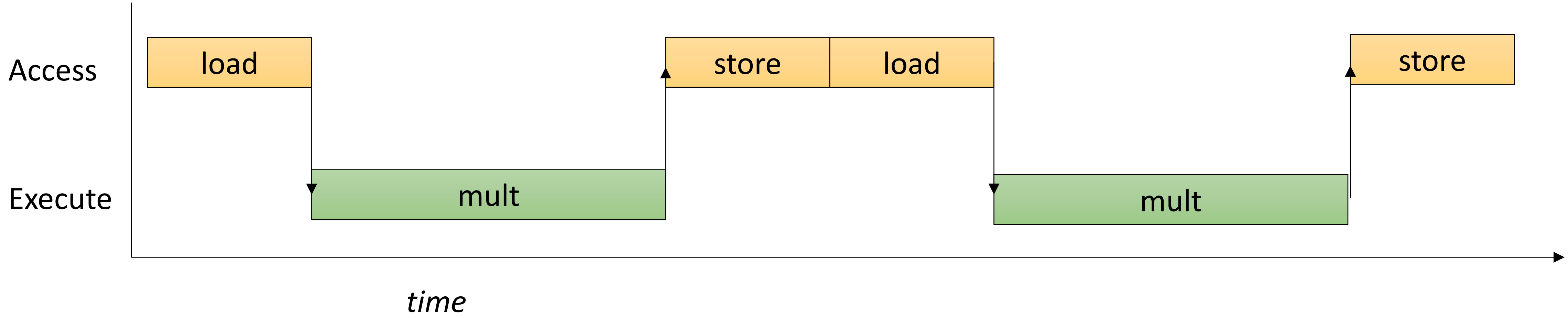


DAE Parallelism

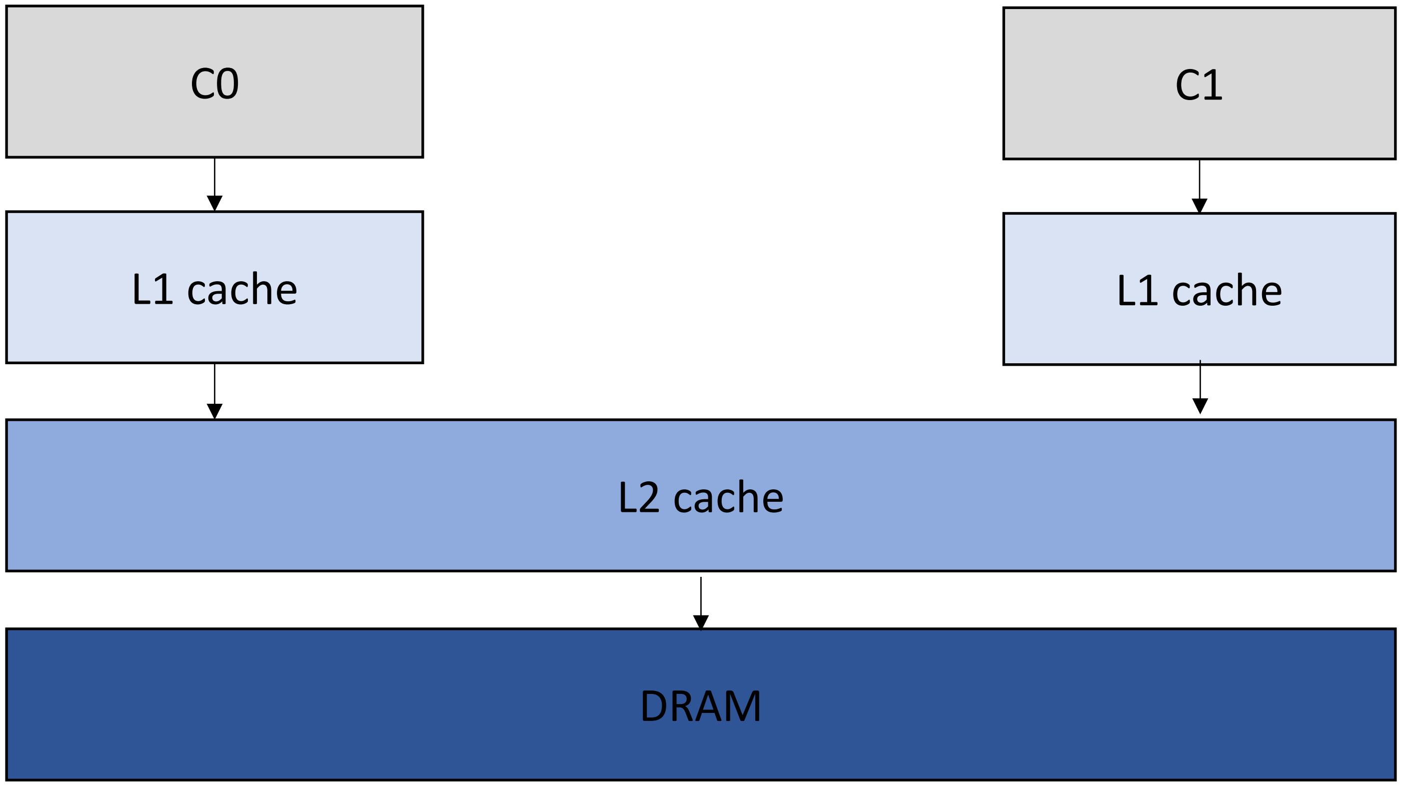




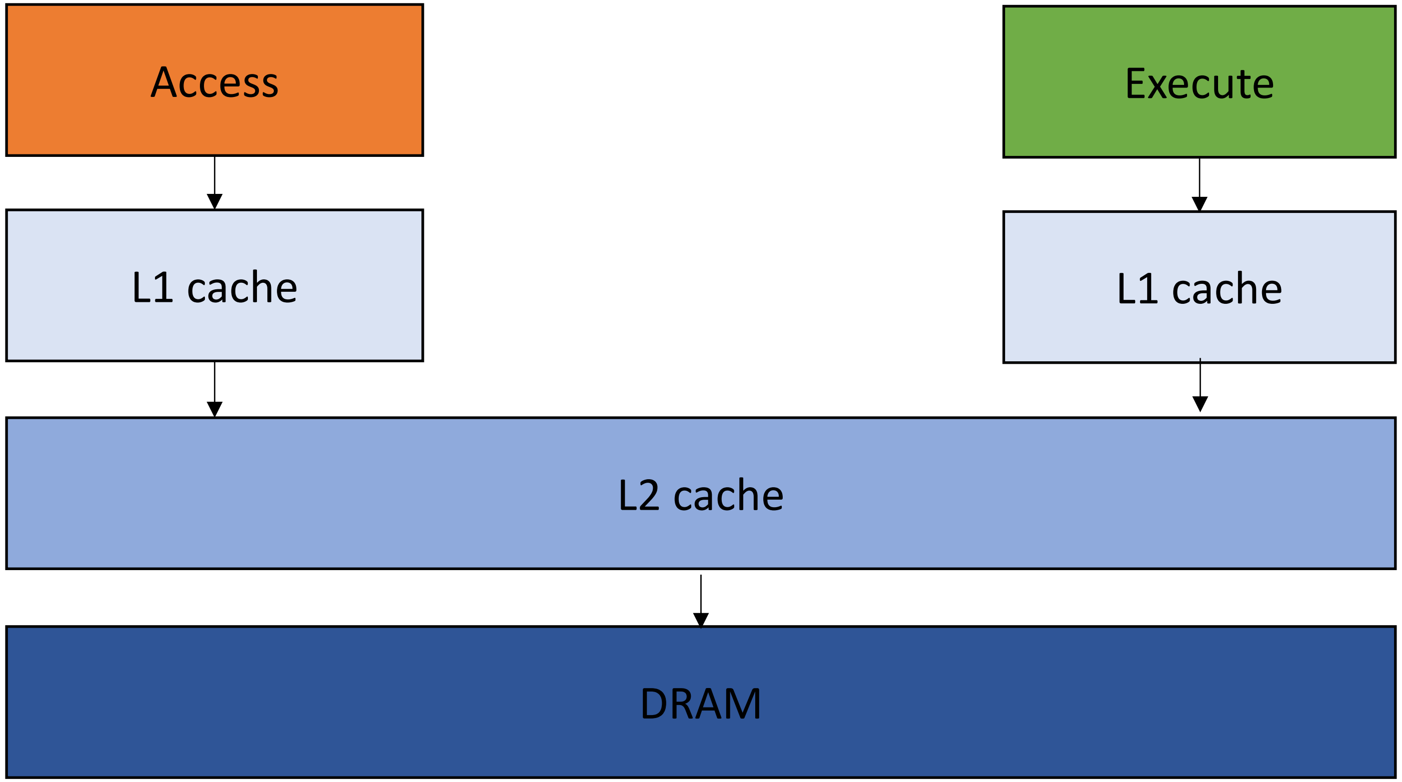

```
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   float r1 = r0 * 3.14;
   store(a + i, r1);
}
```



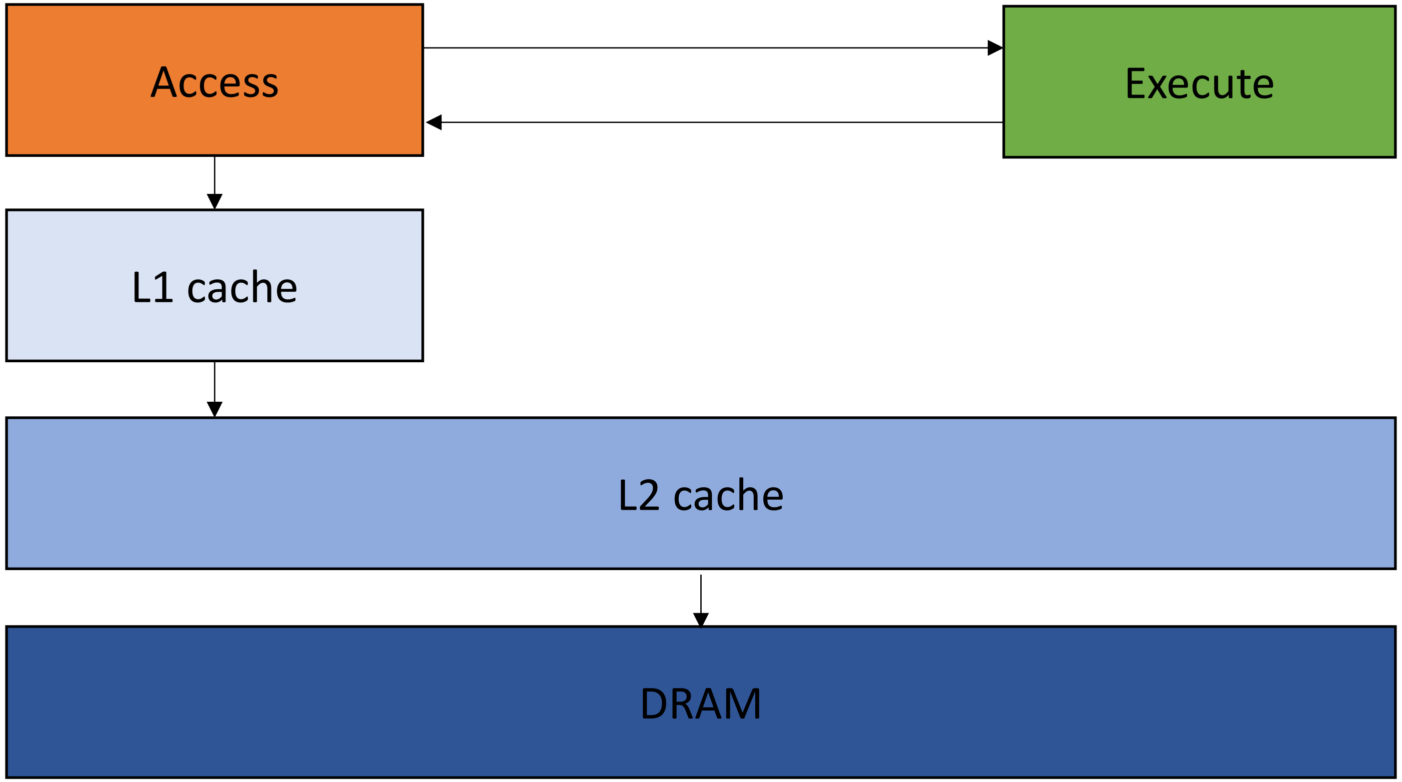
Specializing a DAE architecture



Specializing a DAE architecture



Specializing a DAE architecture

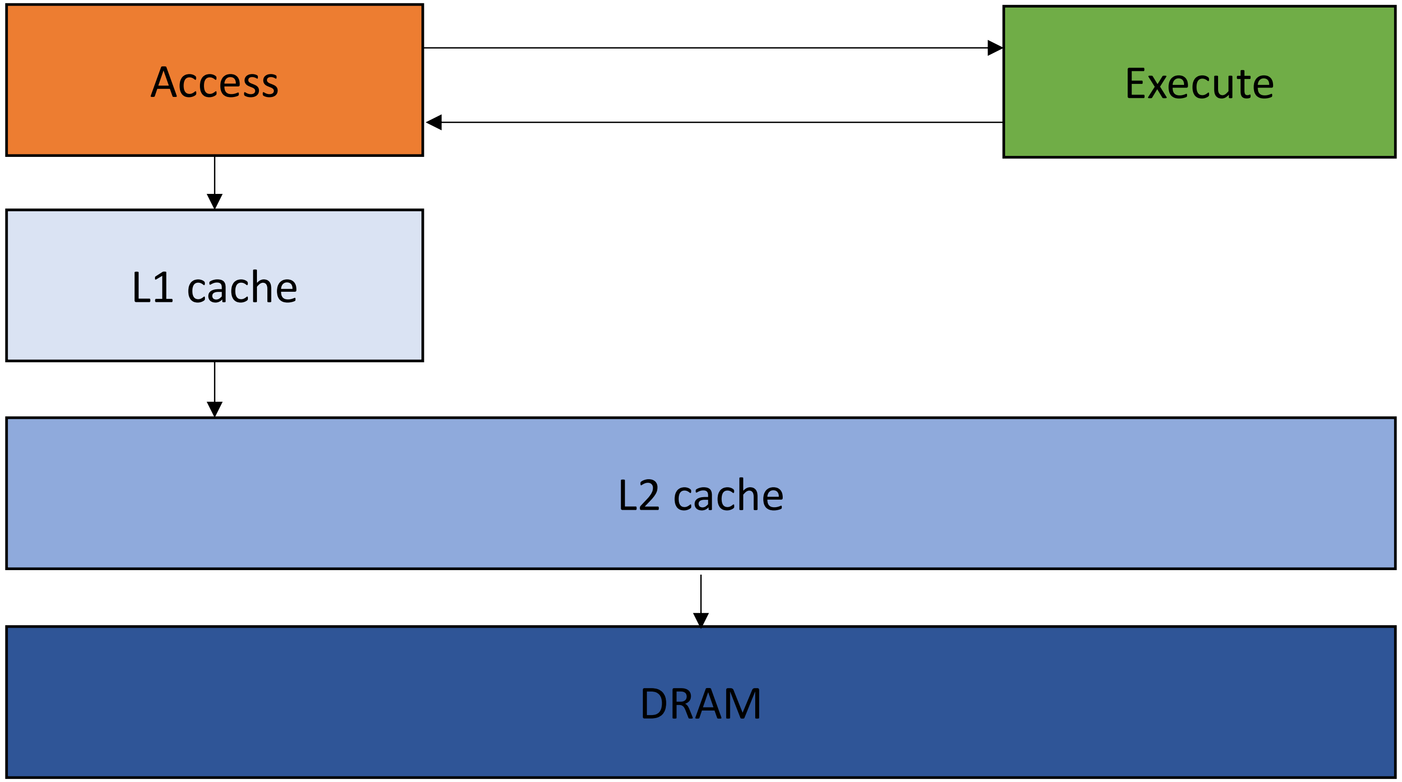


Less contention
on memory hierarchy

Specializing a DAE architecture

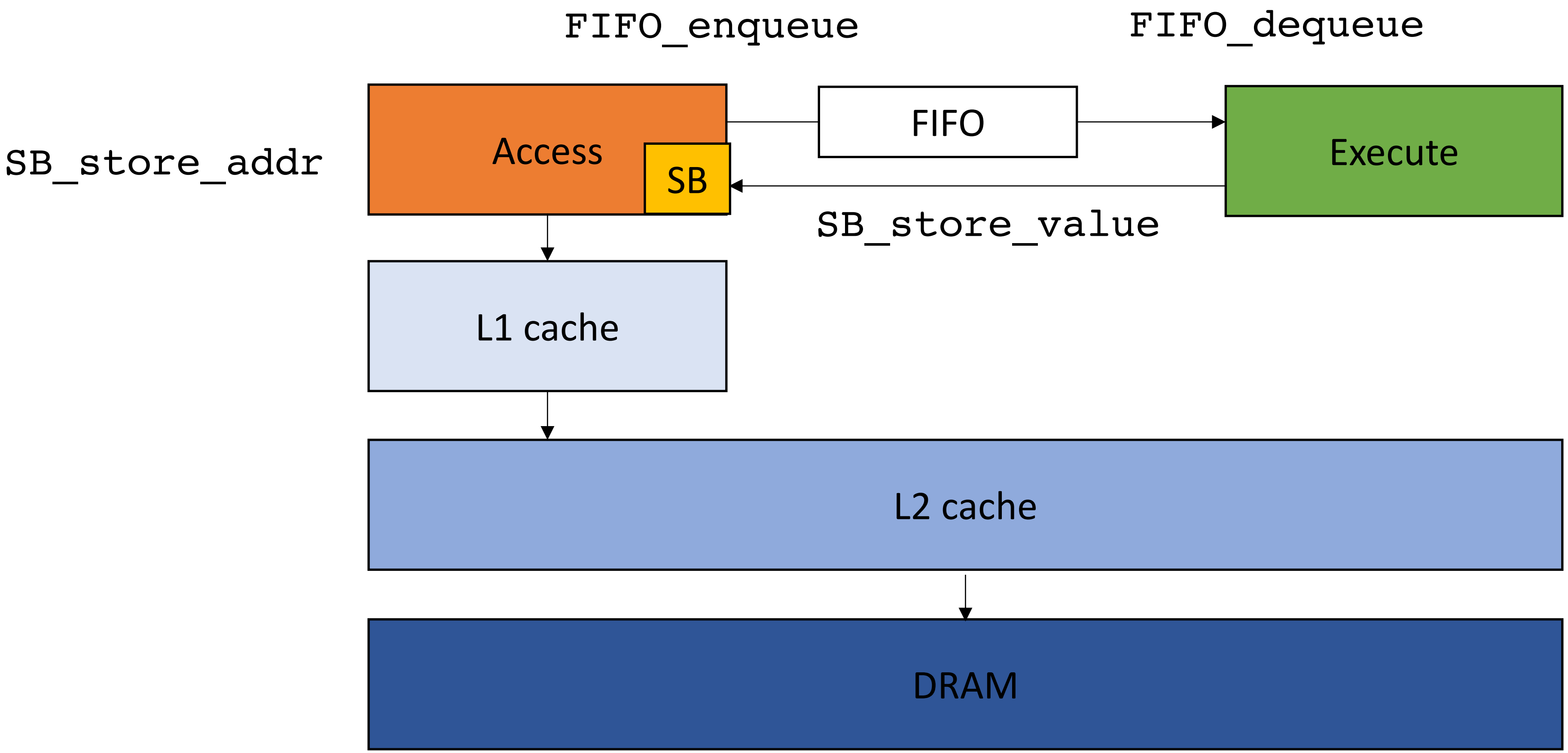
optimizations?
FP unit
Vector units

Less contention
on memory hierarchy



optimizations?
Load/Store unit
Storebuffer

DAE API


FIFO_enqueue
FIFO_dequeue
SB_store_addr
Access
SB
FIFO
Execute
SB_store_value
L1 cache
L2 cache
DRAM

Compiler

- Given a sequential program, how can we automatically target a DAE architecture?

Program slicing

- Mark Weiser in 1981.
  - presented as a formalization of debugging

PROGRAM SLICING*

Mark Weiser

Computer Science Department
University of Maryland
College Park, MD 20742

Abstract

Program slicing is a method used by experienced computer programmers for abstracting from programs. Starting from a subset of a program's behavior, slicing reduces that program to a minimal form which still produces that behavior. The reduced program, called a "slice", is an independent program guaranteed to faithfully represent the original program within the domain of the specified subset of behavior.

Finding a slice is in general unsolvable. A dataflow algorithm is presented for approximating slices when the behavior subset is specified as the values of a set of variables at a statement. Experimental evidence is presented that these slices are used by programmers during debugging. Experience with two automatic slicing tools is summarized. New measures of program complexity are suggested based on the organization of a program's slices.

KEYWORDS: debugging, program maintenance, software tools, program metrics, human factors, dataflow analysis

Introduction

program is more easily con-
tained when broken
t methods

behavior is of interest. For instance, during debugging a subset of behavior is being corrected, and in program modification or maintenance a subset of behavior is being improved or replaced. In these cases, a programmer starts from the program behavior and proceeds to find and modify the corresponding portions of program code. Code not having to do with behavior of interest is ignored. Gould and Dronkowski (1974) report programmers behaving this way during debugging, and a further confirming experiment is presented below.

A programmer maintaining a large, unfamiliar program would almost have to use this behavior-first approach to the code. Understanding an entire system to change only a small piece would take too much time. Since most program maintenance is done by persons other than the program designers, and since 67 percent of programming effort goes into maintenance (Zelkowitz, Shaw, and Gannon 1979), decomposing programs by behavior must be a common occurence.

Automatic slicing requires that behavior be specified in a certain form. If the behavior of interest can be expressed as the values of some sets of variables at some set of statements, then this specification is said to be a slicing criterion. Dataflow analysis (Hecht 1977) can find all the program code which might have influenced the specified behavior, and this code is called a slice of the program. A slice is itself an executable program, whose behavior must be identical to the specified subset of the original pro-

of some slicing crite-

Program slicing

Main idea:

Program slicing

Main idea:

- **Forward Slicing**: given statements S, remove all statements except for those that depend (control or data) on $s \in S$
  - Intuitively: get a minimal (heuristically) program where all actions depend on statements in S

Program slicing

Main idea:

- **Forward Slicing**: given statements S, remove all statements except for those that depend (control or data) on $s \in S$
  - Intuitively: get a minimal (heuristically) program where all actions depend on statements in S
- **Backward Slicing**: given statements S, remove all statements except for those that for which $s \in S$ depend on.
  - Intuitively: get a minimal (heuristically) program where all actions influence statements in S

Program slicing

Main idea:

- **Forward Slicing**: given statements S, remove all statements except for those that depend (control or data) on $s \in S$
  - Intuitively: get a minimal (heuristically) program where all actions depend on statements in S

This is the one we will focus on

- **Backward Slicing**: given statements S, remove all statements except for those that for which $s \in S$ depend on.
  - Intuitively: get a minimal (heuristically) program where all actions influence statements in S

Program slicing

```
1. r0 = a + b;
2. r1 = b + c;
3. r2 = r0 * r0;
4. r4 = r1 + r0;
5. r5 = r2 + r0;
6. r6 = 128;
7. assert(r5 == r6)
```

slicing criterion: [
`“7. assert(r5 == r6)”`
]

start with the statement and work backwards until there are no more dependencies

Program slicing

```
1. r0 = a + b;
2. r1 = b + c;
3. r2 = r0 * r0;
4. r4 = r1 + r0;
5. r5 = r2 + r0;
6. r6 = 128;
7. assert(r5 == r6)
```

slicing criterion: [
“7. assert(r5 == r6)”
]

start with the statement and work backwards until there are no more dependencies

Program slicing

```
1. r0 = a + b;
2. r1 = b + c;
3. r2 = r0 * r0;
4. r4 = r1 + r0;
5. r5 = r2 + r0;
6. r6 = 128;
7. assert(r5 == r6)
```

slicing criterion: [
“7. assert(r5 == r6)”
]

start with the statement and work backwards until there are no more dependencies

Program slicing

```
1. r0 = a + b;
2. r1 = b + c;
3. r2 = r0 * r0;
4. r4 = r1 + r0;
5. r5 = r2 + r0;
6. r6 = 128;
7. assert(r5 == r6)
```

slicing criterion: [
“7. assert(r5 == r6)”
]

start with the statement and work backwards until there are no more dependencies

Program slicing

```
1. r0 = a + b;
2. r1 = b + c;
3. r2 = r0 * r0;
4. r4 = r1 + r0;
5. r5 = r2 + r0;
6. r6 = 128;
7. assert(r5 == r6)
```

slicing criterion: [
“`7. assert(r5 == r6)`”
]

start with the statement and work backwards until there are no more dependencies

Program slicing - Control dependence

```
1.   r0 = a + b;
2.   r1 = b + c;
3.   r2 = r0 * r0;
4.   r4 = r1 + r0;
5.   bne r4, 64, END
6.   r5 = r2 + r0;
7.   r6 = 128;
8.   assert(r5 == r6)
9.END:
```

slicing criterion: [
“`8. assert(r5 == r6)`”
]

start with the statement and work backwards until there are no more dependencies

Program slicing - Control dependence

```
1.   r0 = a + b;
2.   r1 = b + c;
3.   r2 = r0 * r0;
4.   r4 = r1 + r0;
5.   bne r4, 64, END
```

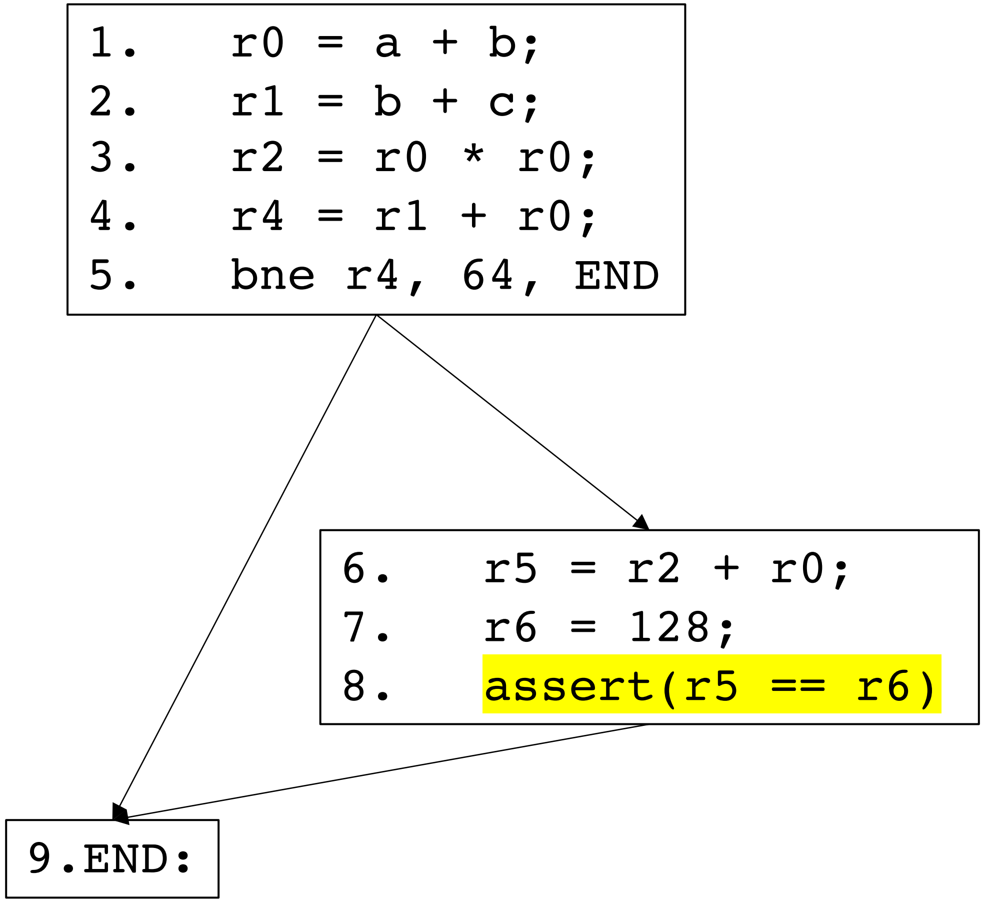
```
6.   r5 = r2 + r0;
7.   r6 = 128;
8.   assert(r5 == r6)
```

```
9.END:
```

slicing criterion: [
“`8. assert(r5 == r6)`”
]

start with the statement and work backwards until there are no more dependencies

Program slicing - Control dependence



slicing criterion: [
`“8. assert(r5 == r6)”`
]

start with the statement and work backwards until there are no more dependencies

Program slicing - Control dependence

```
1.    r0 = a + b;
2.    r1 = b + c;
3.    r2 = r0 * r0;
4.    r4 = r1 + r0;
5.    bne r4, 64, END
```

branch statement

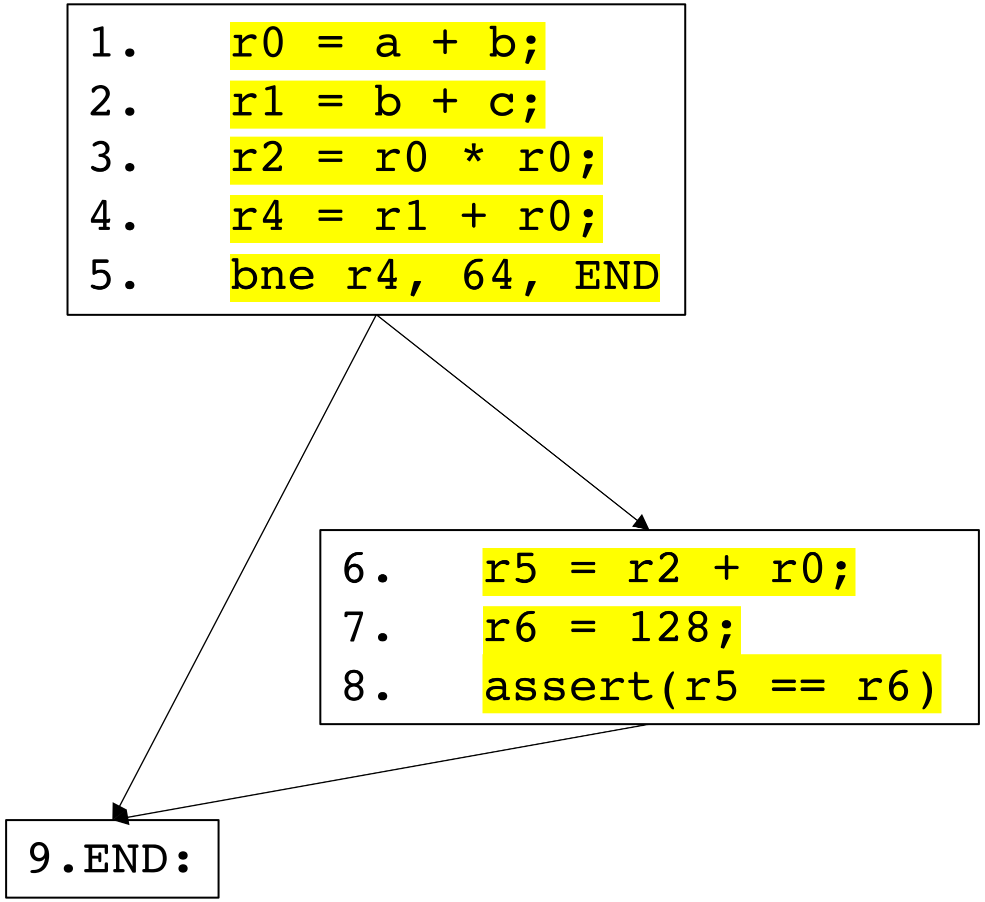
```
6.    r5 = r2 + r0;
7.    r6 = 128;
8.    assert(r5 == r6)
```

```
9.END:
```

slicing criterion: [
“8. assert(r5 == r6)”
]

start with the statement and work backwards until there are no more dependencies

Program slicing - Control dependence



slicing criterion: [
`“8. assert(r5 == r6)”`
]

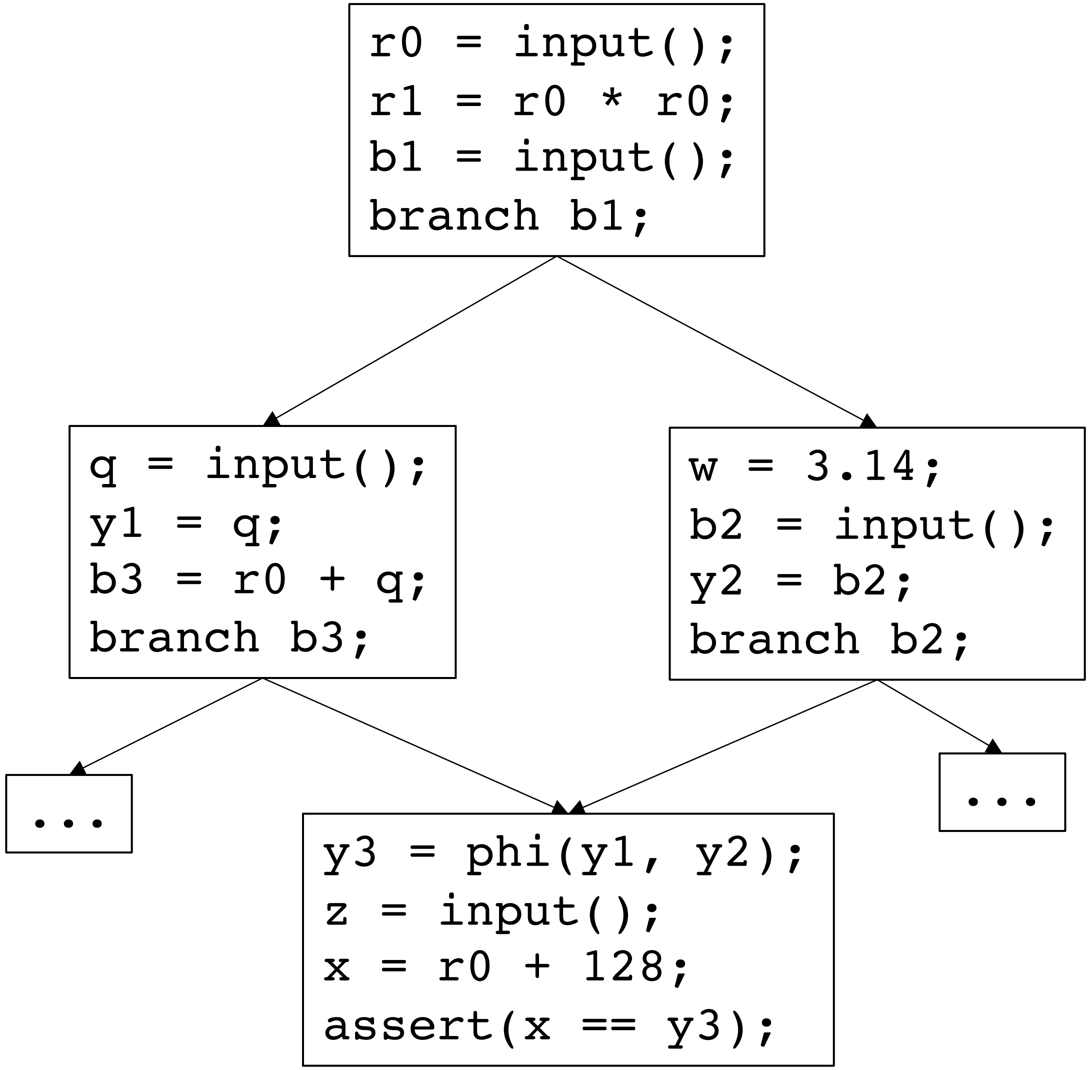
start with the statement and work backwards until there are no more dependencies

Backwards slicing algorithm

```
Worklist = S; // slicing criteria
while (!Worklist.empty()) {
  stmt = Worklist.pop();
  if (is_marked(stmt)) {
    continue;
  }
  mark(stmt);
  for a in stmt.args() {
    worklist.append(a);
  }
  for p in cfg[stmt].predecessors() {
    worklist.append(p.branch_stmt());
  }
}
```

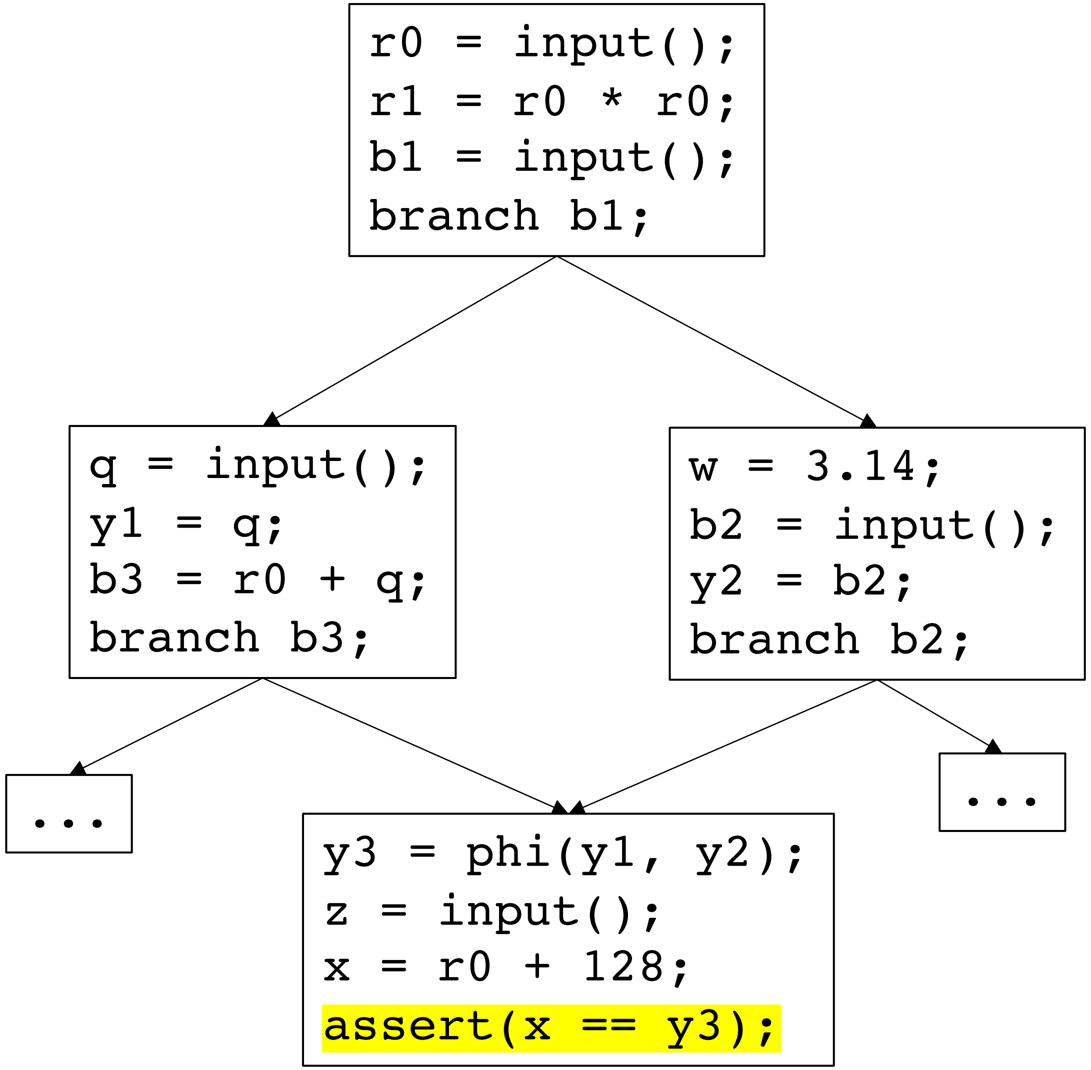
Backwards slicing algorithm

```
Worklist = S; // slicing criteria
while (!Worklist.empty()) {
  stmt = Worklist.pop();
  if (is_marked(stmt)) {
    continue;
  }
  mark(stmt);
  for a in stmt.args() {
    worklist.append(a);
  }
  for p in cfg[stmt].predecessors() {
    worklist.append(p.branch_stmt());
  }
}
```



Backwards slicing algorithm

```
Worklist = S; // slicing criteria
while (!Worklist.empty()) {
  stmt = Worklist.pop();
  if (is_marked(stmt)) {
    continue;
  }
  mark(stmt);
  for a in stmt.args() {
    worklist.append(a);
  }
  for p in cfg[stmt].predecessors() {
    worklist.append(p.branch_stmt());
  }
}
```



marked:

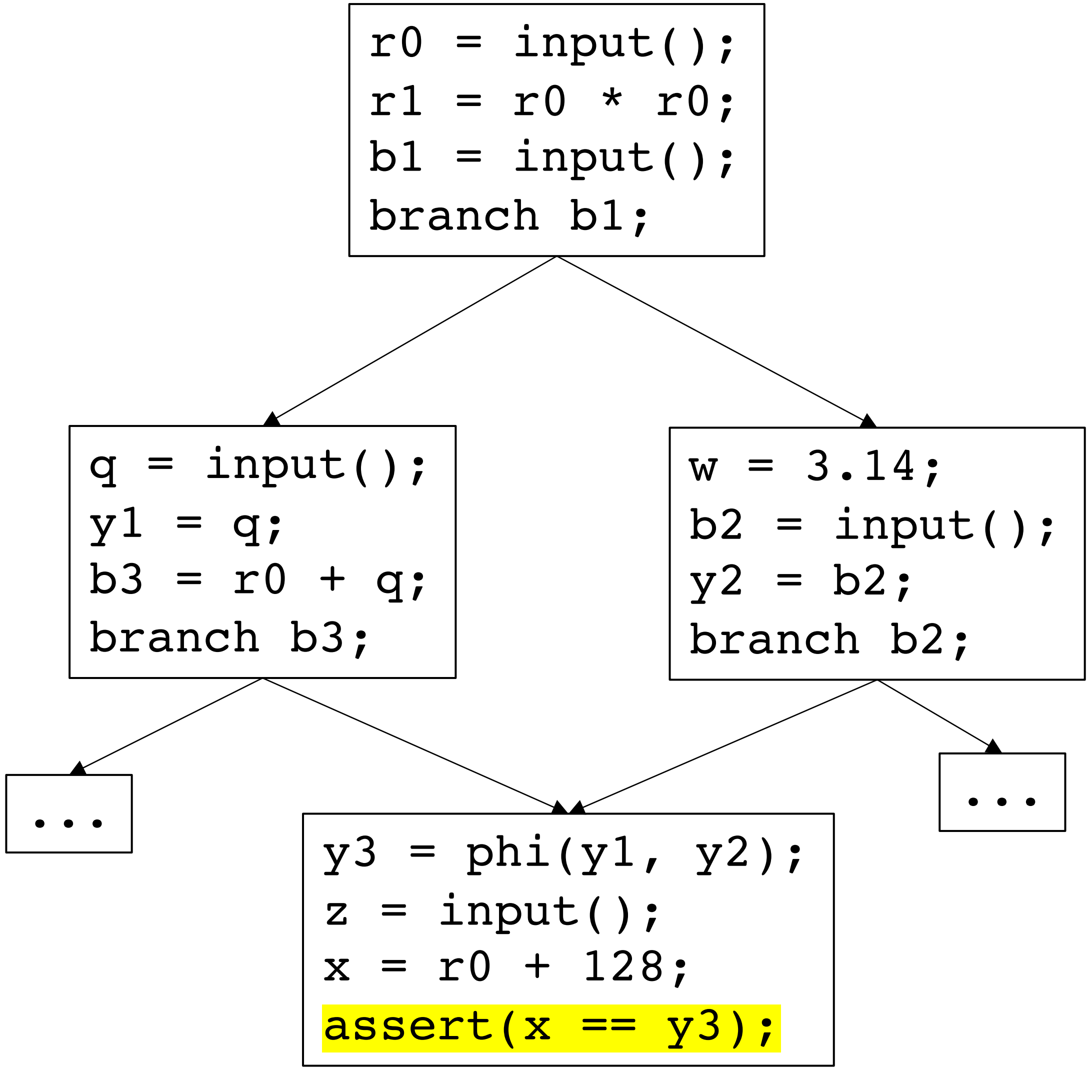
worklist:
assert()

Backwards slicing algorithm

```
Worklist = S; // slicing criteria
while (!Worklist.empty()) {
  stmt = Worklist.pop();
  if (is_marked(stmt)) {
    continue;
  }
  mark(stmt);
  for a in stmt.args() {
    worklist.append(a);
  }
  for p in cfg[stmt].predecessors() {
    worklist.append(p.branch_stmt());
  }
}
```

marked:
assert()

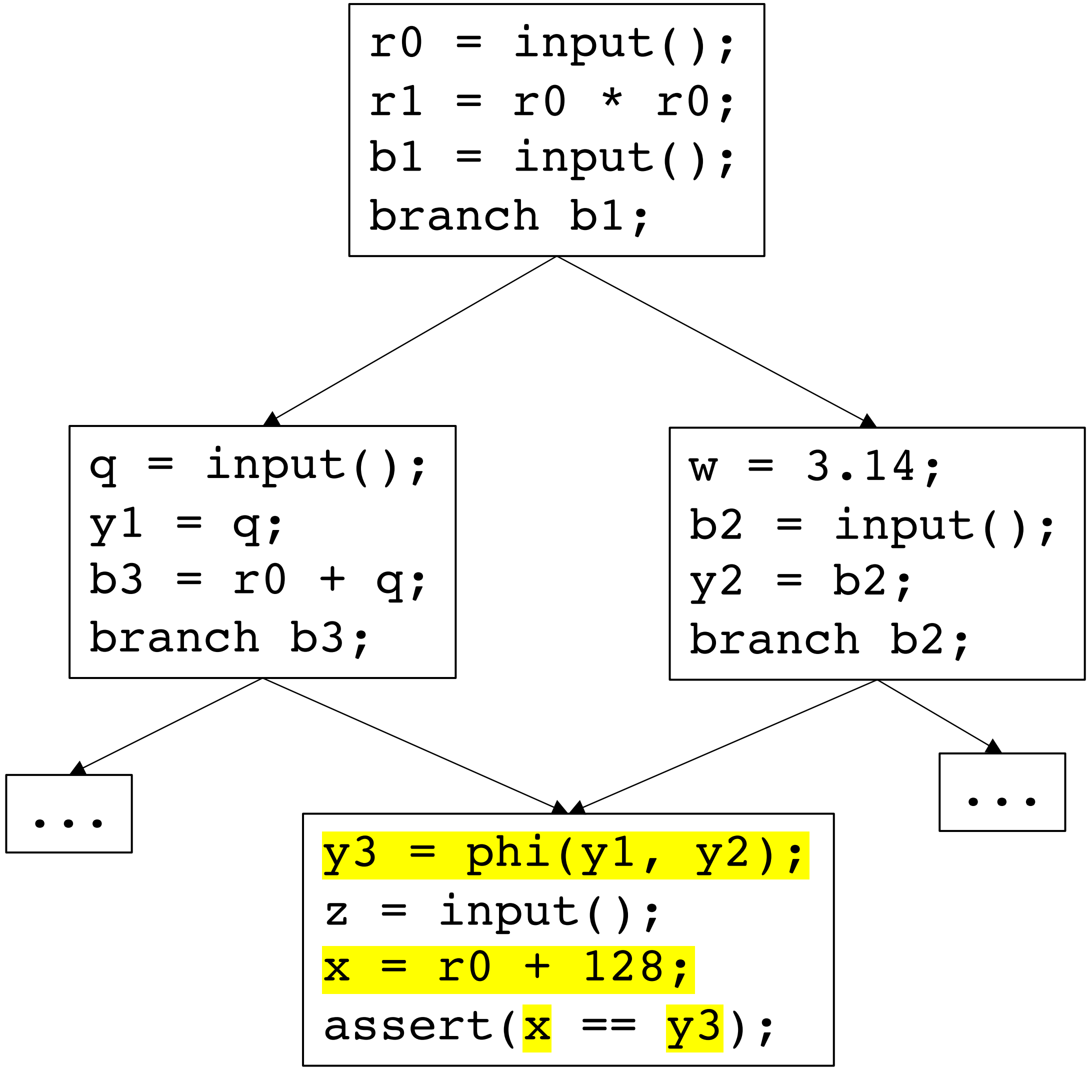
worklist:



Backwards slicing algorithm

```
Worklist = S; // slicing criteria
while (!Worklist.empty()) {
  stmt = Worklist.pop();
  if (is_marked(stmt)) {
    continue;
  }
  mark(stmt);
  for a in stmt.args() {
    worklist.append(a);
  }
  for p in cfg[stmt].predecessors() {
    worklist.append(p.branch_stmt());
  }
}
```

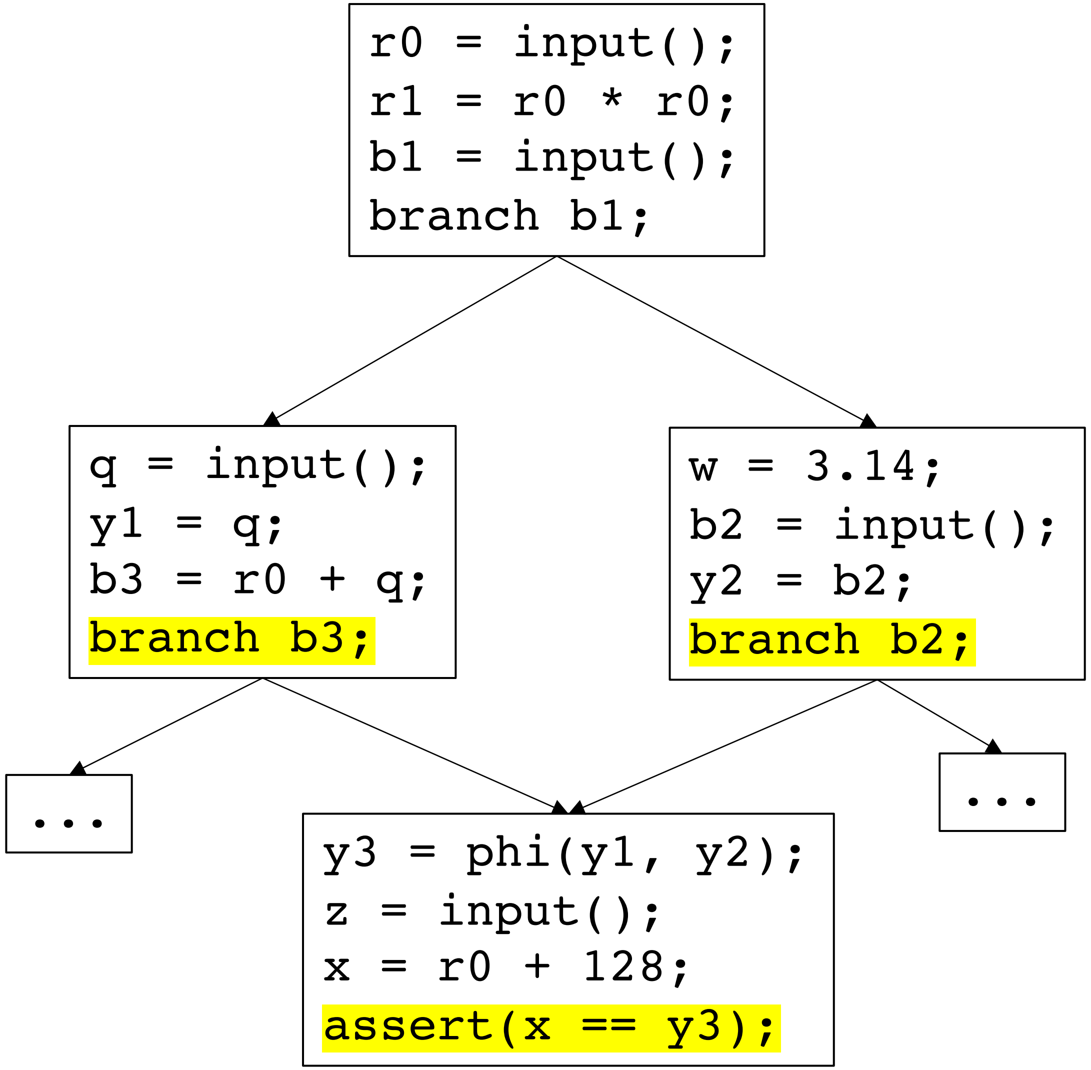
marked:	worklist:
assert()	x
	y3



Backwards slicing algorithm

```
Worklist = S; // slicing criteria
while (!Worklist.empty()) {
  stmt = Worklist.pop();
  if (is_marked(stmt)) {
    continue;
  }
  mark(stmt);
  for a in stmt.args() {
    worklist.append(a);
  }
  for p in cfg[stmt].predecessors() {
    worklist.append(p.branch_stmt());
  }
}
```

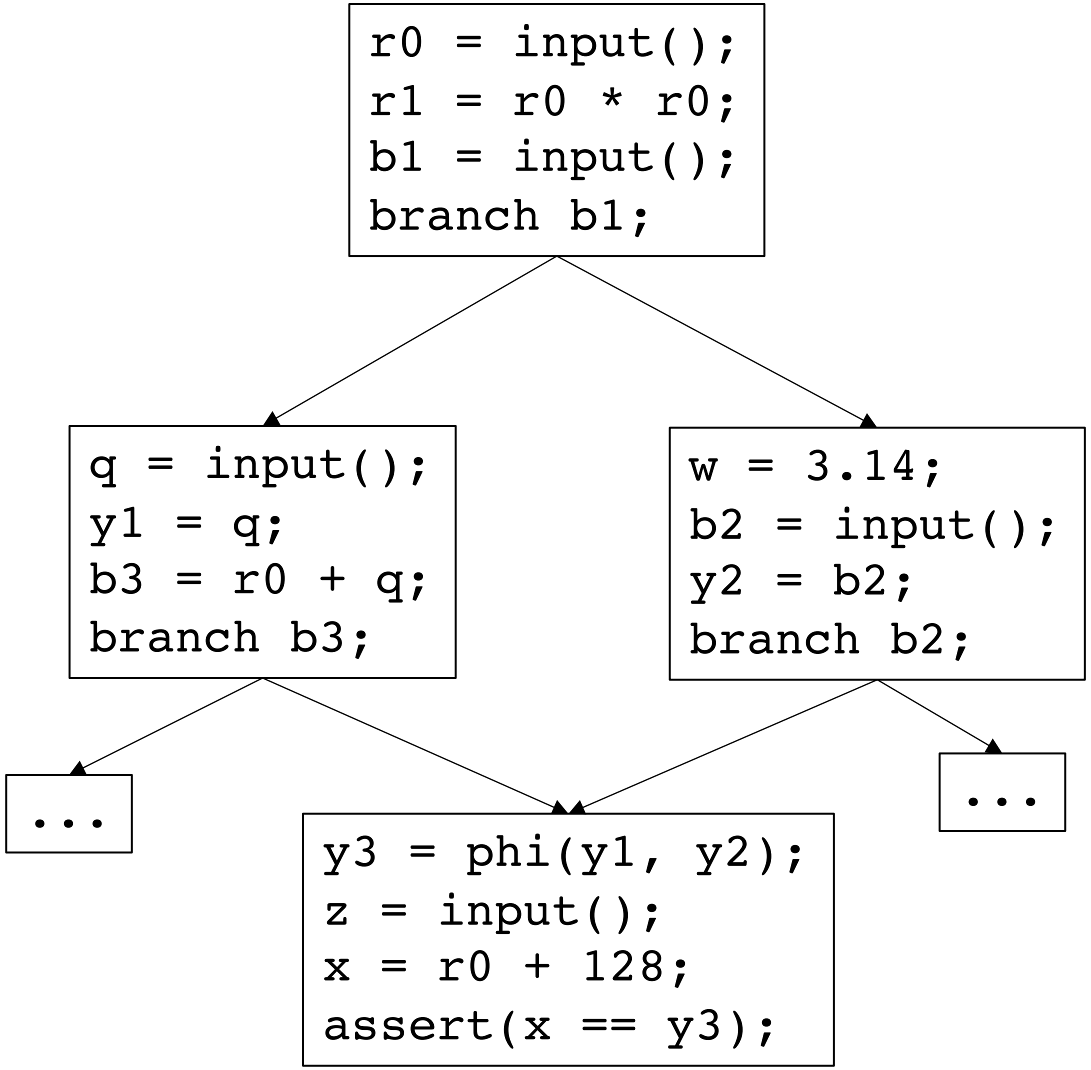
marked:	worklist:
assert()	x
	y3
	branch b3
	branch b2



Backwards slicing algorithm

```
Worklist = S; // slicing criteria
while (!Worklist.empty()) {
  stmt = Worklist.pop();
  if (is_marked(stmt)) {
    continue;
  }
  mark(stmt);
  for a in stmt.args() {
    worklist.append(a);
  }
  for p in cfg[stmt].predecessors() {
    worklist.append(p.branch_stmt());
  }
}
```

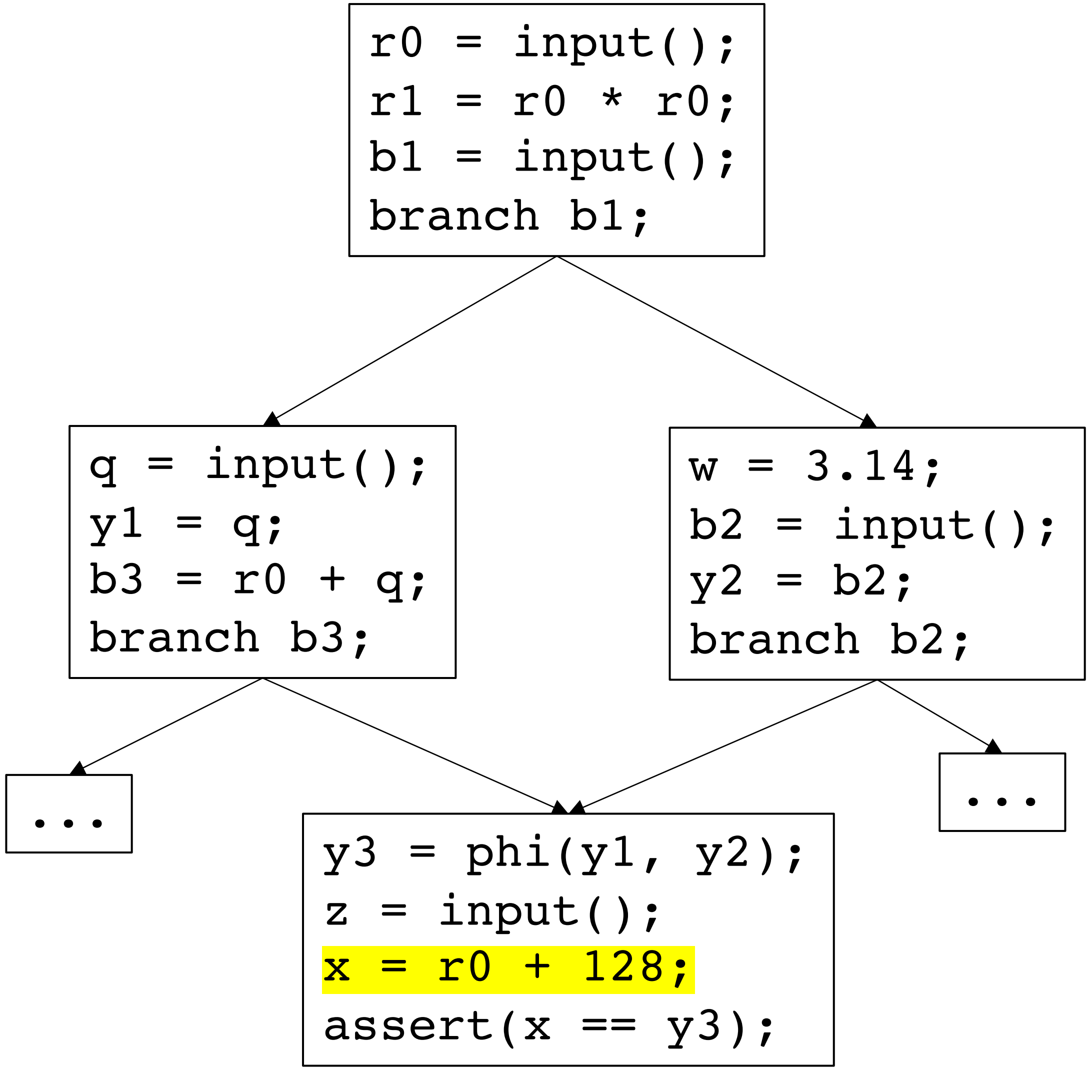
marked:	worklist:
assert()	x
	y3
	branch b3
	branch b2



Backwards slicing algorithm

```
Worklist = S; // slicing criteria
while (!Worklist.empty()) {
  stmt = Worklist.pop();
  if (is_marked(stmt)) {
    continue;
  }
  mark(stmt);
  for a in stmt.args() {
    worklist.append(a);
  }
  for p in cfg[stmt].predecessors() {
    worklist.append(p.branch_stmt());
  }
}
```

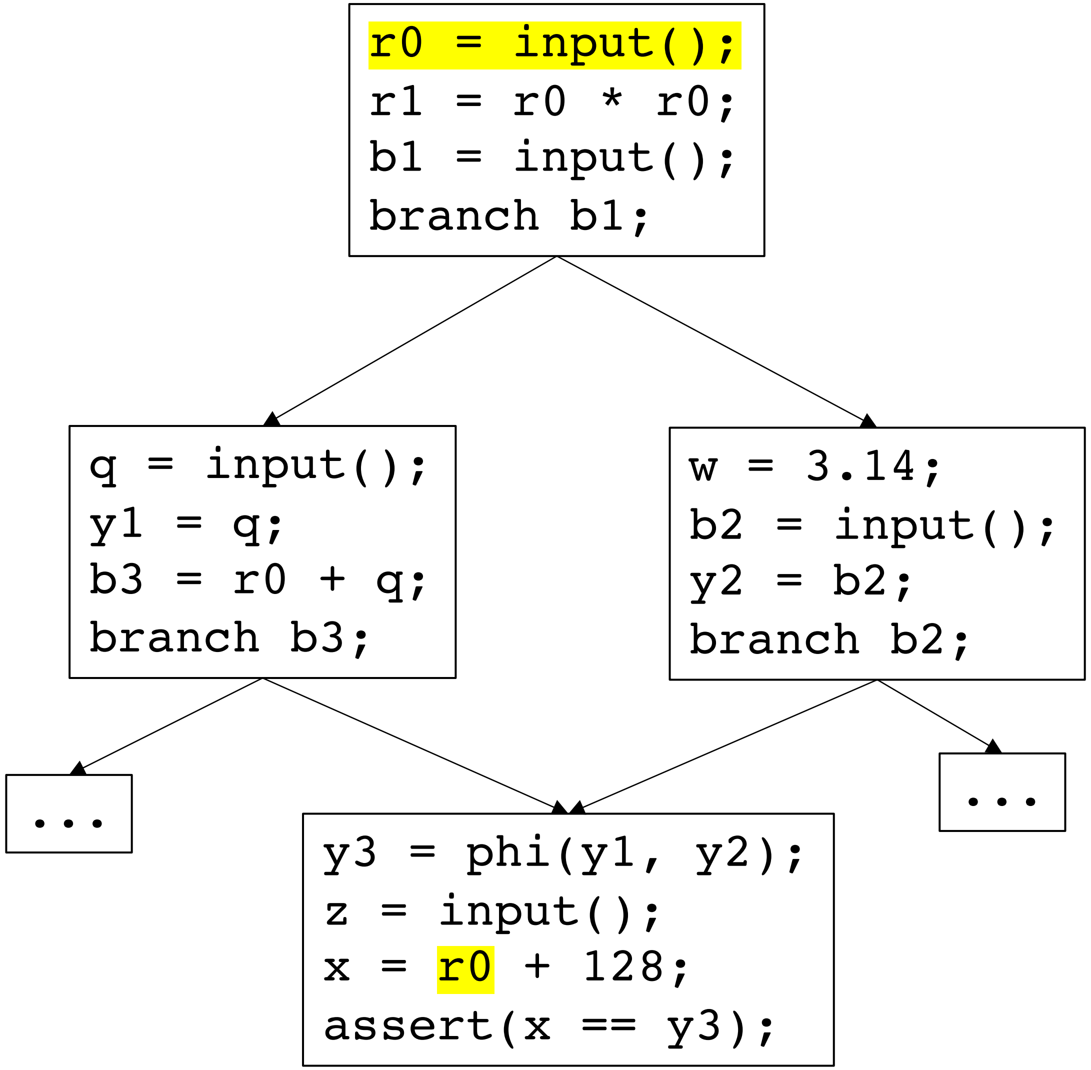
marked:	worklist:
assert()	y3
x	branch b3
	branch b2



Backwards slicing algorithm

```
Worklist = S; // slicing criteria
while (!Worklist.empty()) {
  stmt = Worklist.pop();
  if (is_marked(stmt)) {
    continue;
  }
  mark(stmt);
  for a in stmt.args() {
    worklist.append(a);
  }
  for p in cfg[stmt].predecessors() {
    worklist.append(p.branch_stmt());
  }
}
```

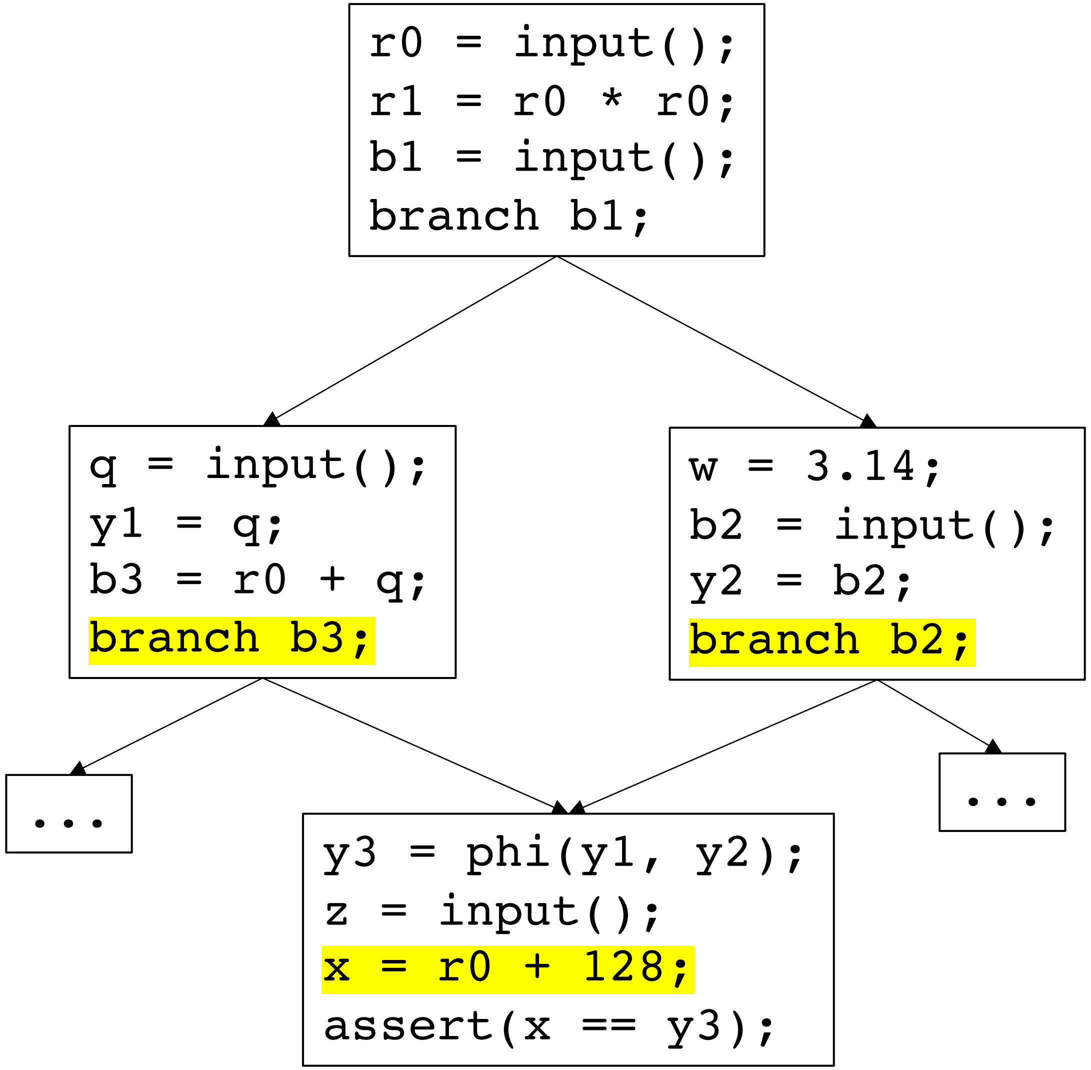
marked:	worklist:
assert()	y3
x	branch b3
	branch b2
	r0



Backwards slicing algorithm

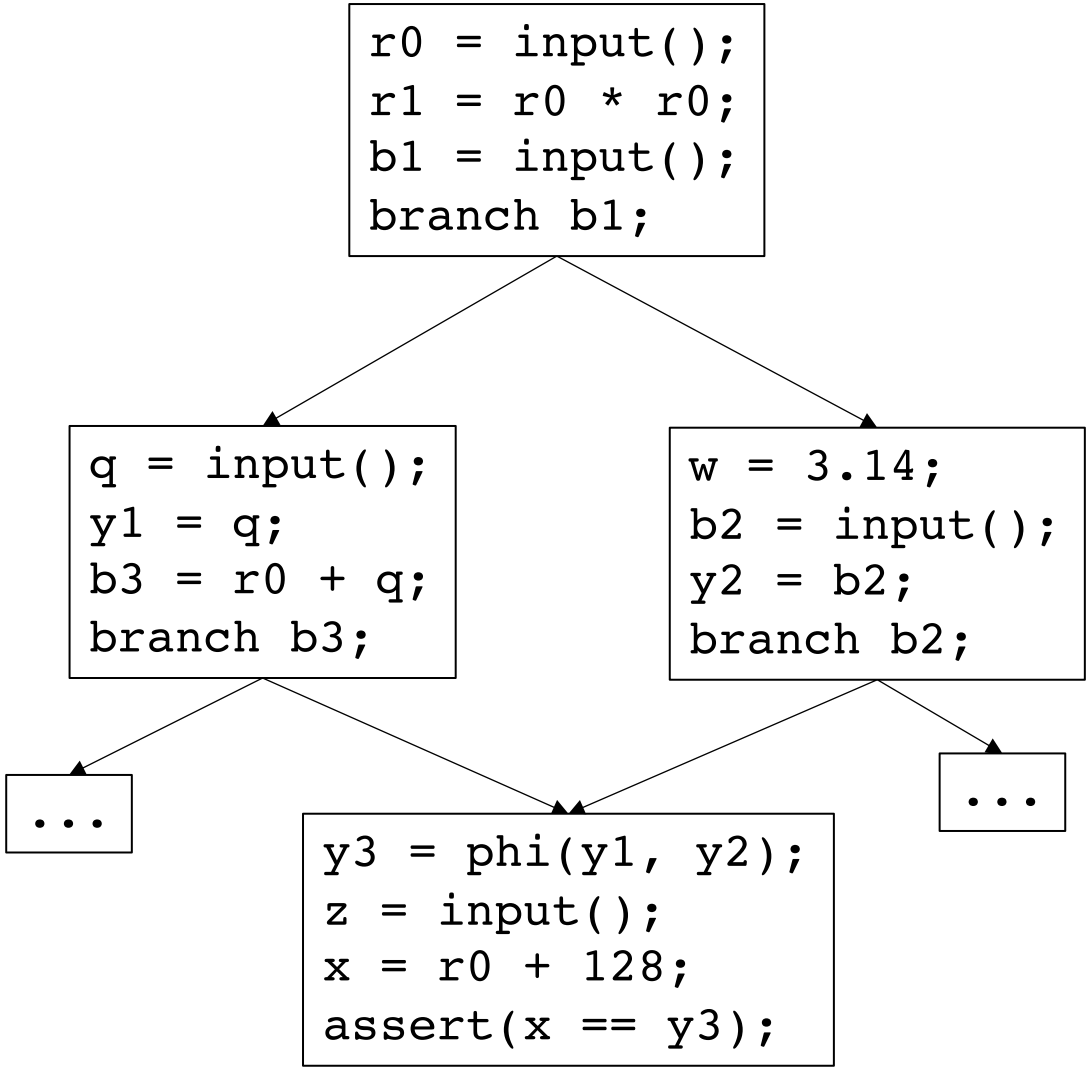
```
Worklist = S; // slicing criteria
while (!Worklist.empty()) {
  stmt = Worklist.pop();
  if (is_marked(stmt)) {
    continue;
  }
  mark(stmt);
  for a in stmt.args() {
    worklist.append(a);
  }
  for p in cfg[stmt].predecessors() {
    worklist.append(p.branch_stmt());
  }
}
```

marked:	worklist:
assert()	y3
x	branch b3
	branch b2
	r0



Backwards slicing algorithm

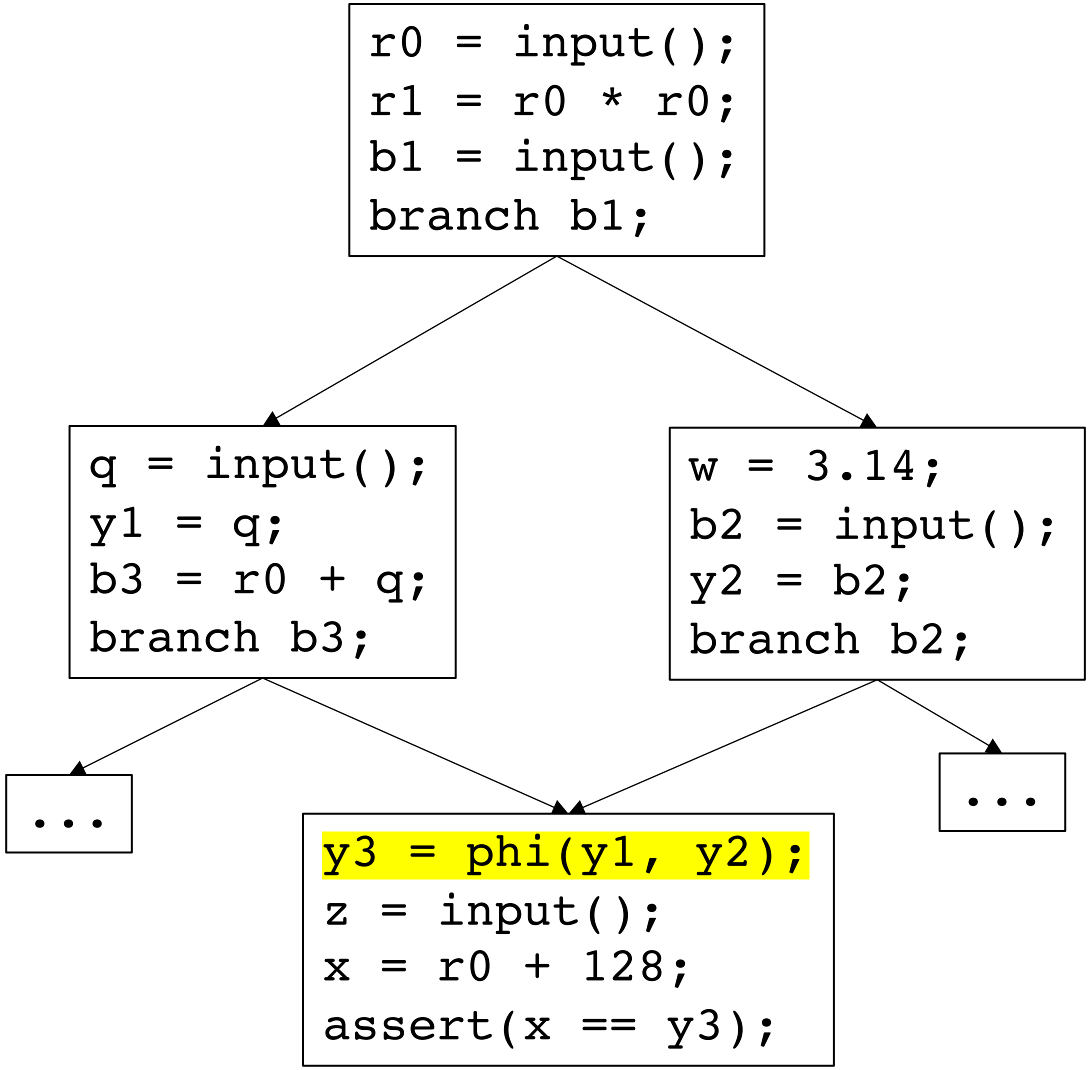
```
Worklist = S; // slicing criteria
while (!Worklist.empty()) {
  stmt = Worklist.pop();
  if (is_marked(stmt)) {
    continue;
  }
  mark(stmt);
  for a in stmt.args() {
    worklist.append(a);
  }
  for p in cfg[stmt].predecessors() {
    worklist.append(p.branch_stmt());
  }
}
```



marked:	worklist:
assert()	y3
x	branch b3
	branch b2
	r0

Backwards slicing algorithm

```
Worklist = S; // slicing criteria
while (!Worklist.empty()) {
  stmt = Worklist.pop();
  if (is_marked(stmt)) {
    continue;
  }
  mark(stmt);
  for a in stmt.args() {
    worklist.append(a);
  }
  for p in cfg[stmt].predecessors() {
    worklist.append(p.branch_stmt());
  }
}
```

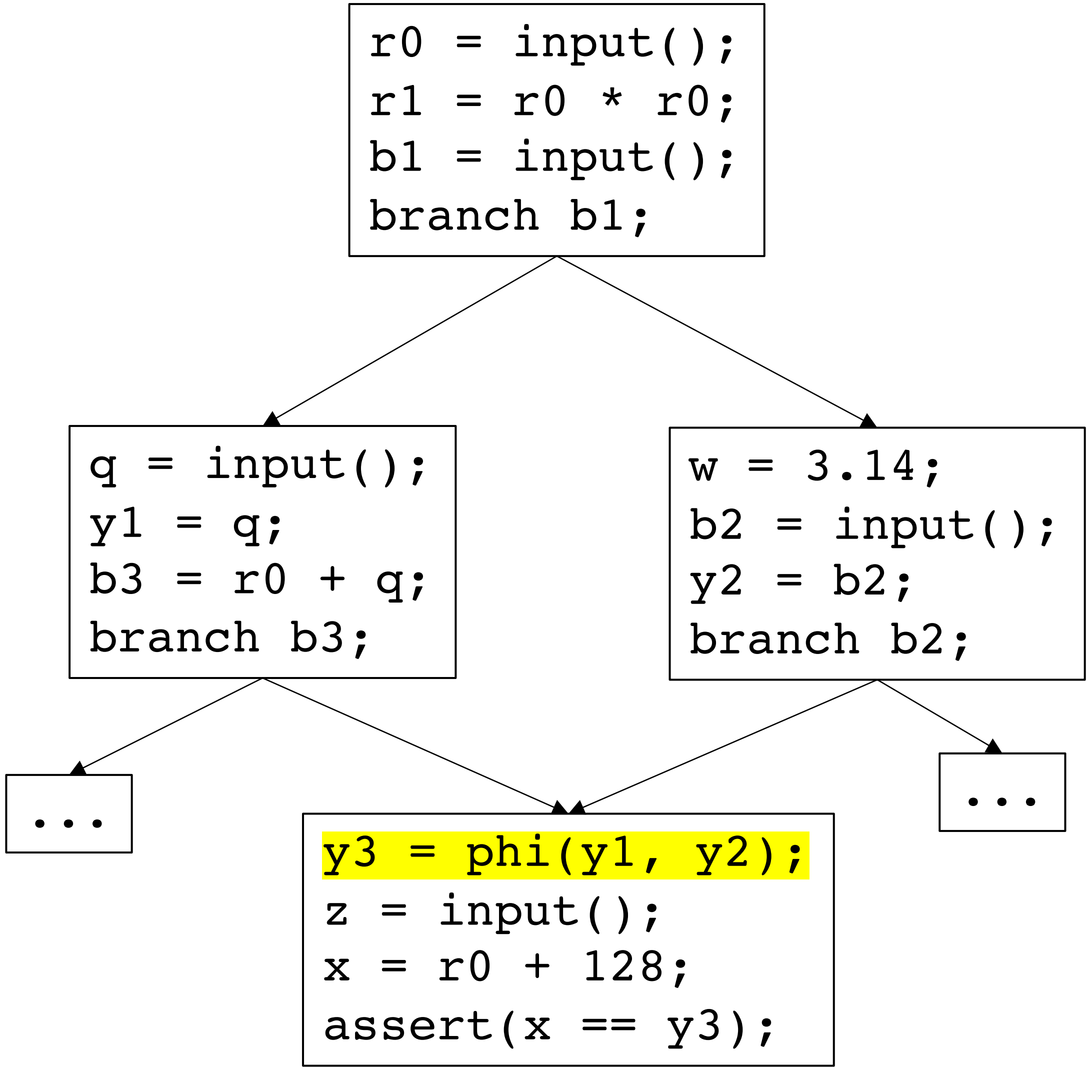


marked:	worklist:
assert()	y3
x	branch b3
	branch b2
	r0

Backwards slicing algorithm

```
Worklist = S; // slicing criteria
while (!Worklist.empty()) {
  stmt = Worklist.pop();
  if (is_marked(stmt)) {
    continue;
  }
  mark(stmt);
  for a in stmt.args() {
    worklist.append(a);
  }
  for p in cfg[stmt].predecessors() {
    worklist.append(p.branch_stmt());
  }
}
```

marked:	worklist:
assert()	branch b3
x	branch b2
y3	r0

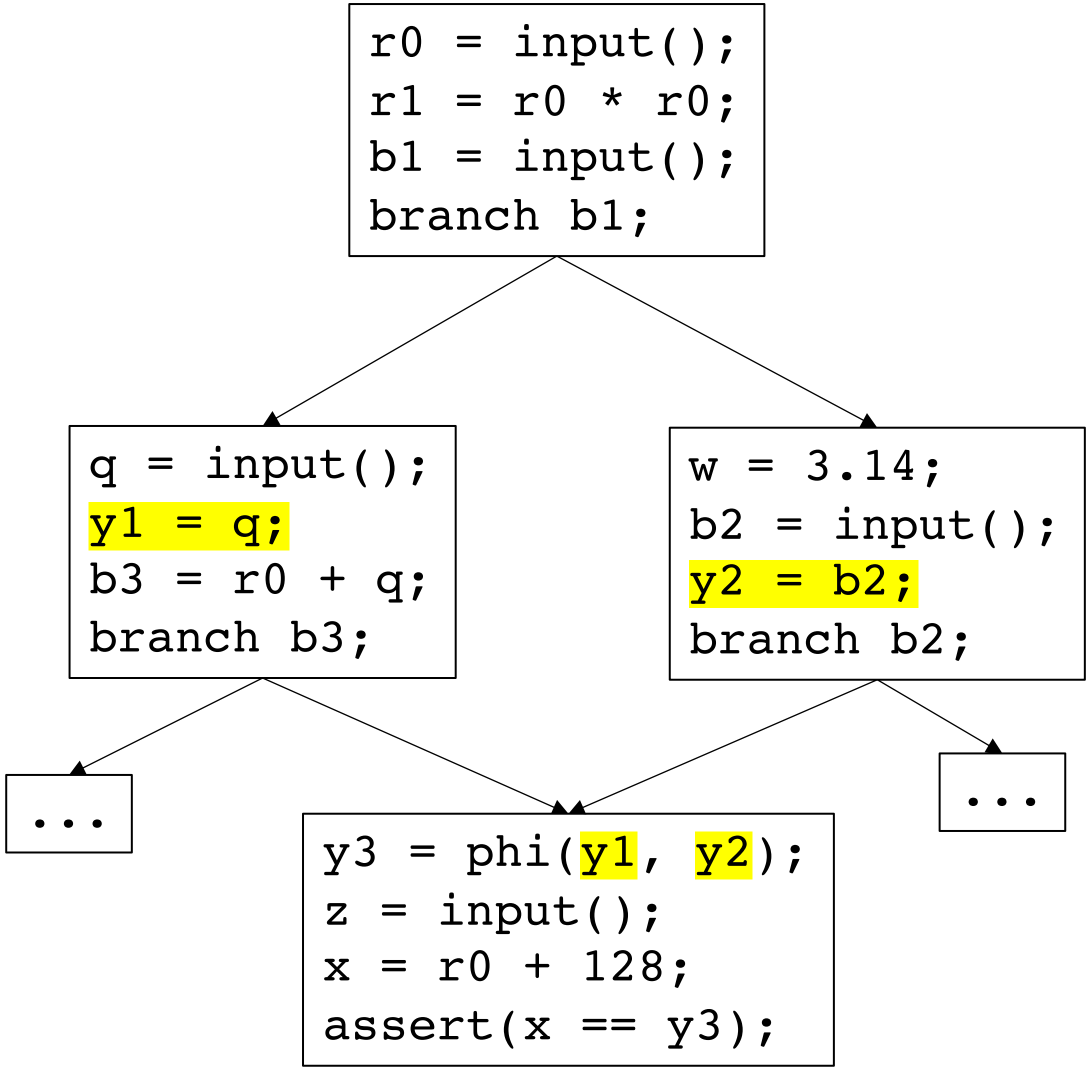


Backwards slicing algorithm

```
Worklist = S; // slicing criteria
while (!Worklist.empty()) {
  stmt = Worklist.pop();
  if (is_marked(stmt)) {
    continue;
  }
  mark(stmt);
  for a in stmt.args() {
    worklist.append(a);
  }
  for p in cfg[stmt].predecessors() {
    worklist.append(p.branch_stmt());
  }
}
```

marked:
assert()
x
y3

worklist:
branch b3
branch b2
r0
y1
y2

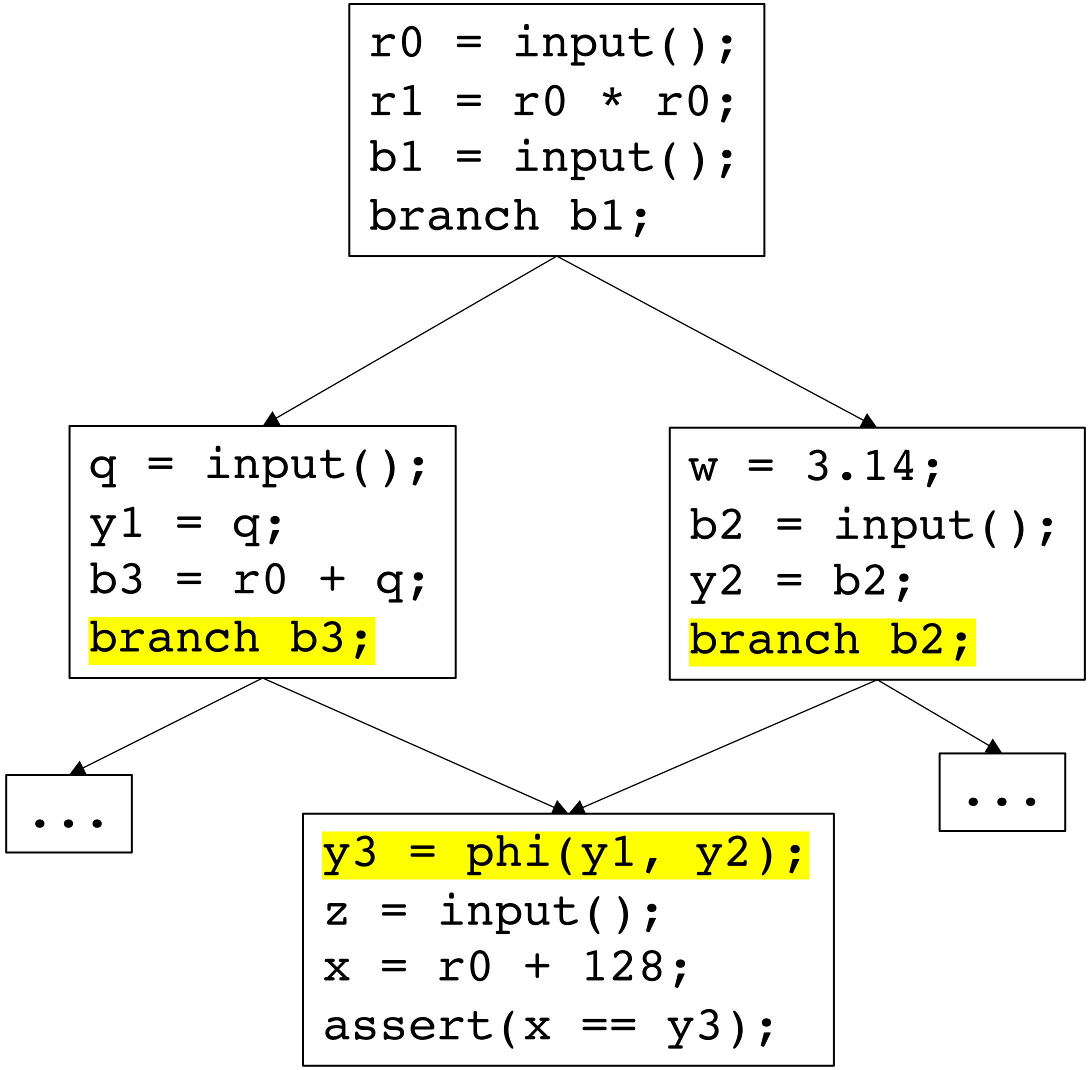


Backwards slicing algorithm

```
Worklist = S; // slicing criteria
while (!Worklist.empty()) {
  stmt = Worklist.pop();
  if (is_marked(stmt)) {
    continue;
  }
  mark(stmt);
  for a in stmt.args() {
    worklist.append(a);
  }
  for p in cfg[stmt].predecessors() {
    worklist.append(p.branch_stmt());
  }
}
```

marked:
assert()
x
y3

worklist:
branch b3
branch b2
r0
y1
y2

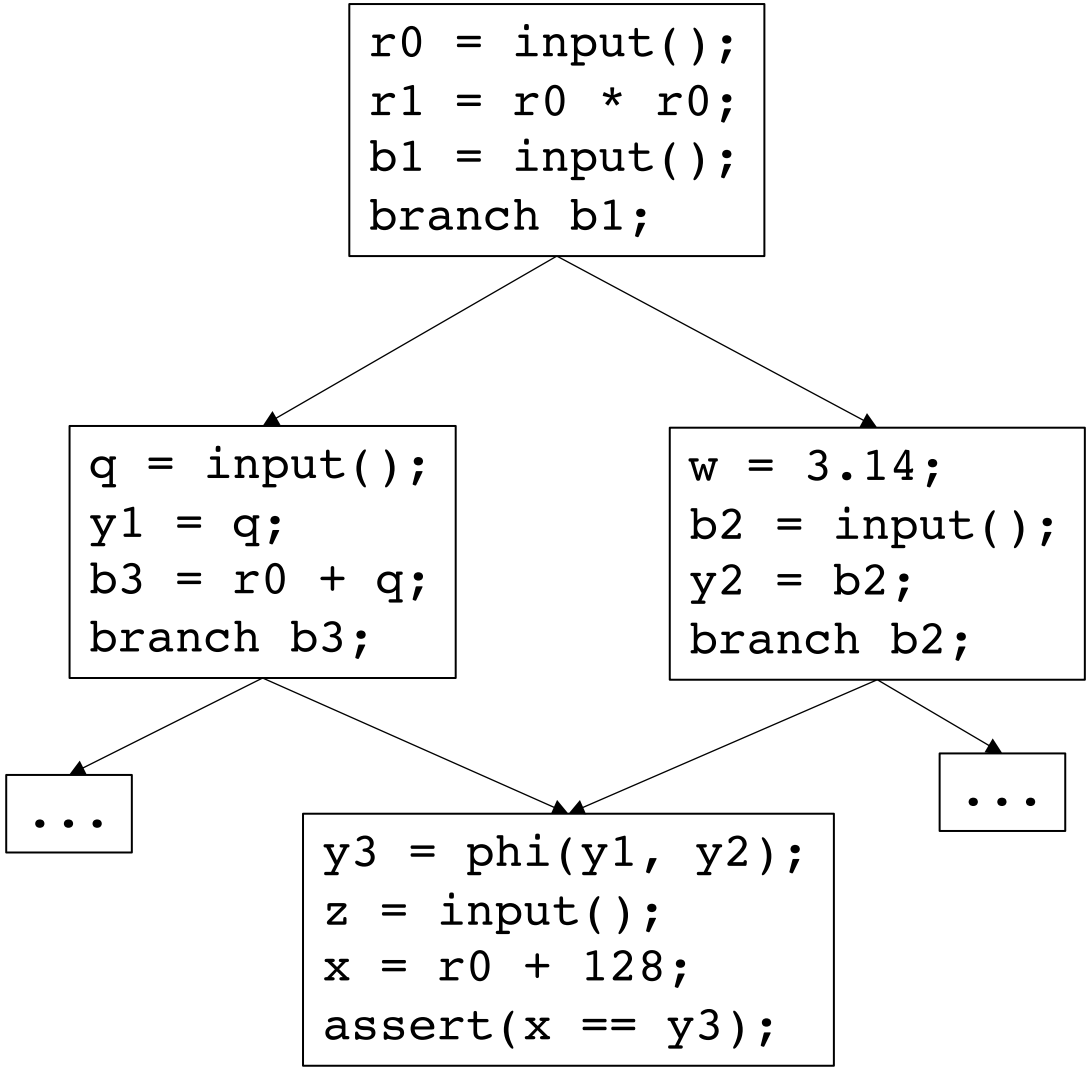


Backwards slicing algorithm

```
Worklist = S; // slicing criteria
while (!Worklist.empty()) {
  stmt = Worklist.pop();
  if (is_marked(stmt)) {
    continue;
  }
  mark(stmt);
  for a in stmt.args() {
    worklist.append(a);
  }
  for p in cfg[stmt].predecessors() {
    worklist.append(p.branch_stmt());
  }
}
```

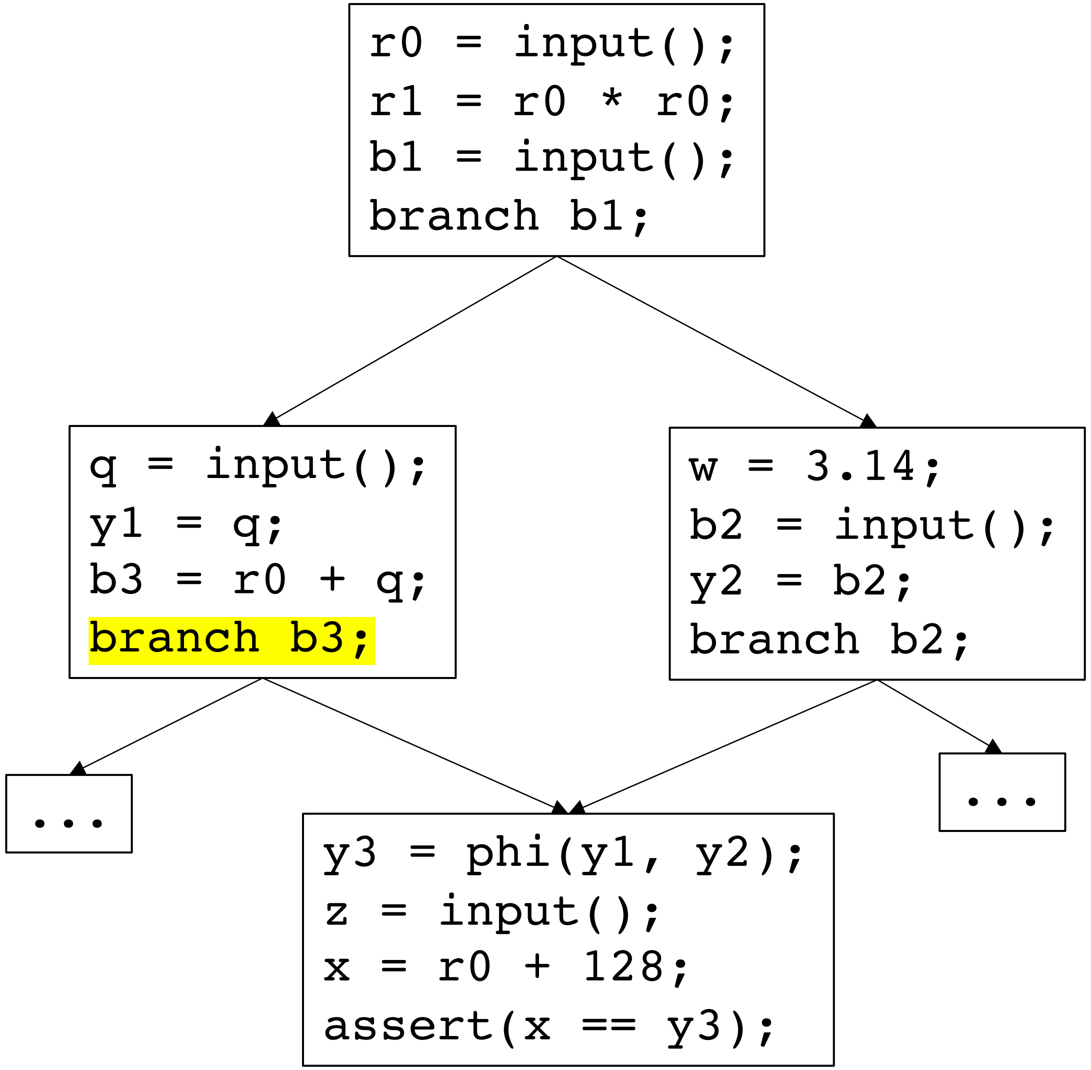
marked:
assert()
x
y3

worklist:
branch b3
branch b2
r0
y1
y2



Backwards slicing algorithm

```
Worklist = S; // slicing criteria
while (!Worklist.empty()) {
  stmt = Worklist.pop();
  if (is_marked(stmt)) {
    continue;
  }
  mark(stmt);
  for a in stmt.args() {
    worklist.append(a);
  }
  for p in cfg[stmt].predecessors() {
    worklist.append(p.branch_stmt());
  }
}
```



marked:
assert()
x
y3
branch b3

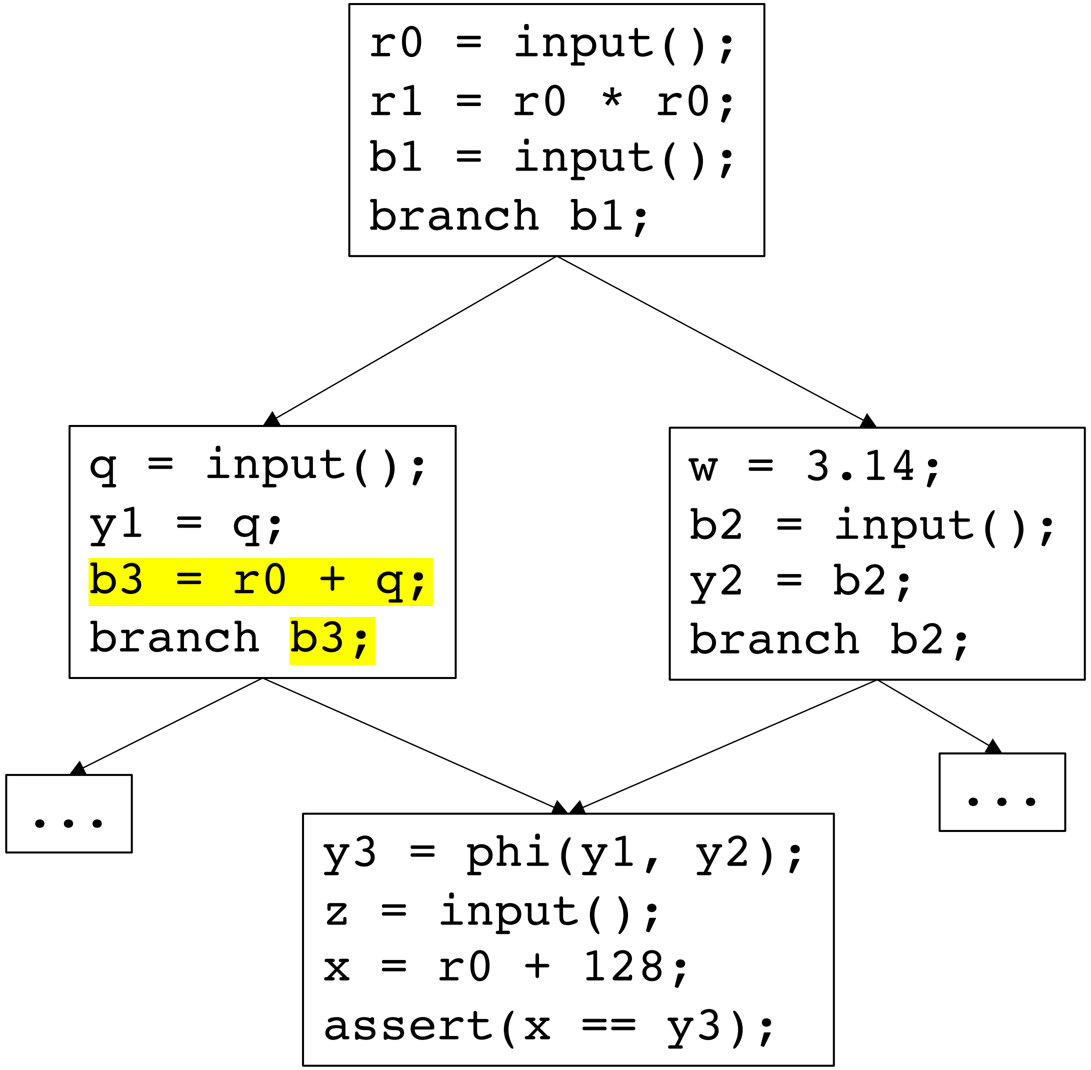
worklist:
branch b2
r0
y1
y2

Backwards slicing algorithm

```
Worklist = S; // slicing criteria
while (!Worklist.empty()) {
  stmt = Worklist.pop();
  if (is_marked(stmt)) {
    continue;
  }
  mark(stmt);
  for a in stmt.args() {
    worklist.append(a);
  }
  for p in cfg[stmt].predecessors() {
    worklist.append(p.branch_stmt());
  }
}
```

marked:
assert()
x
y3
branch b3

worklist:
branch b2
r0
y1
y2
b3

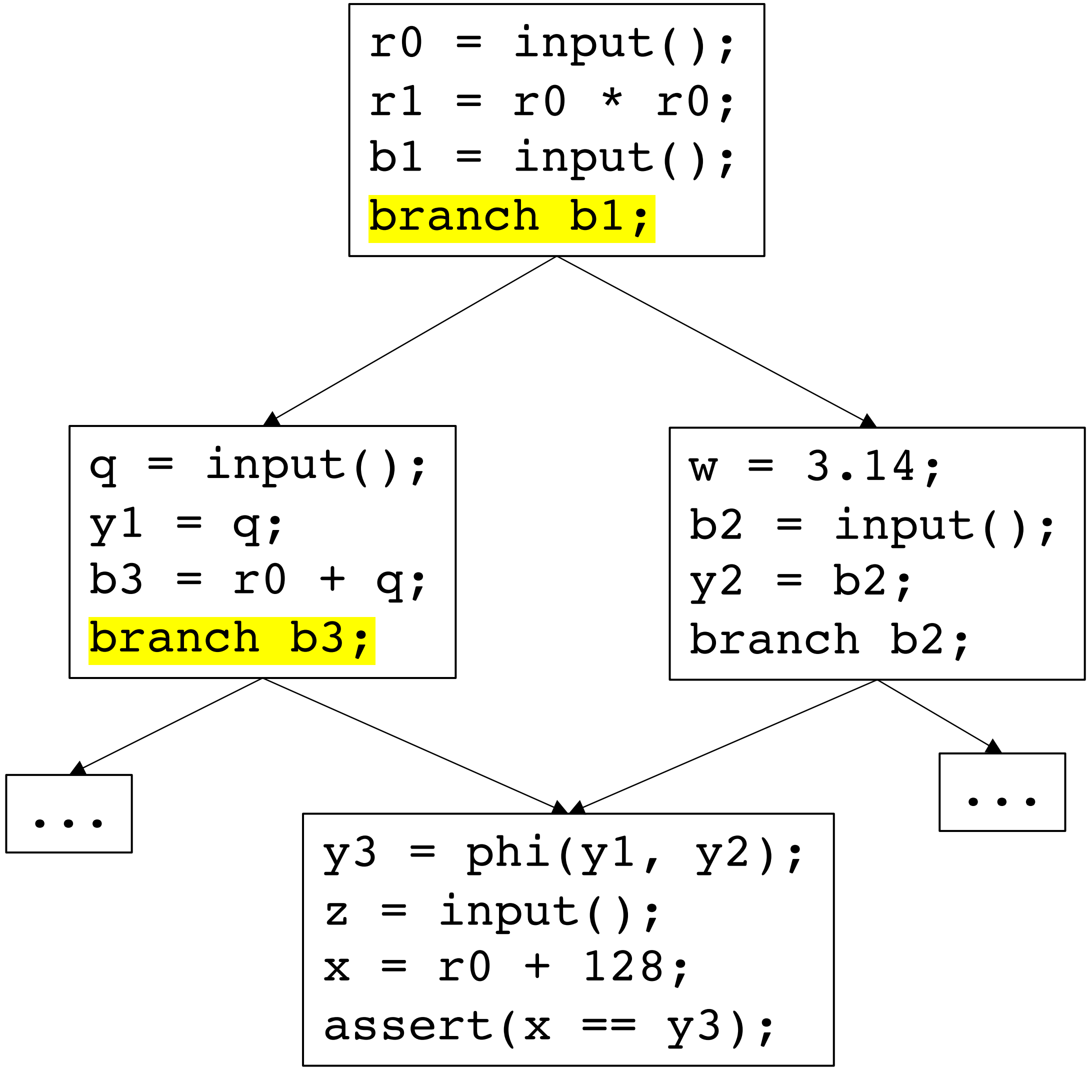


Backwards slicing algorithm

```
Worklist = S; // slicing criteria
while (!Worklist.empty()) {
  stmt = Worklist.pop();
  if (is_marked(stmt)) {
    continue;
  }
  mark(stmt);
  for a in stmt.args() {
    worklist.append(a);
  }
  for p in cfg[stmt].predecessors() {
    worklist.append(p.branch_stmt());
  }
}
```

worklist:
branch b2
r0
y1
y2
b3
branch b1

marked:
assert()
x
y3
branch b3

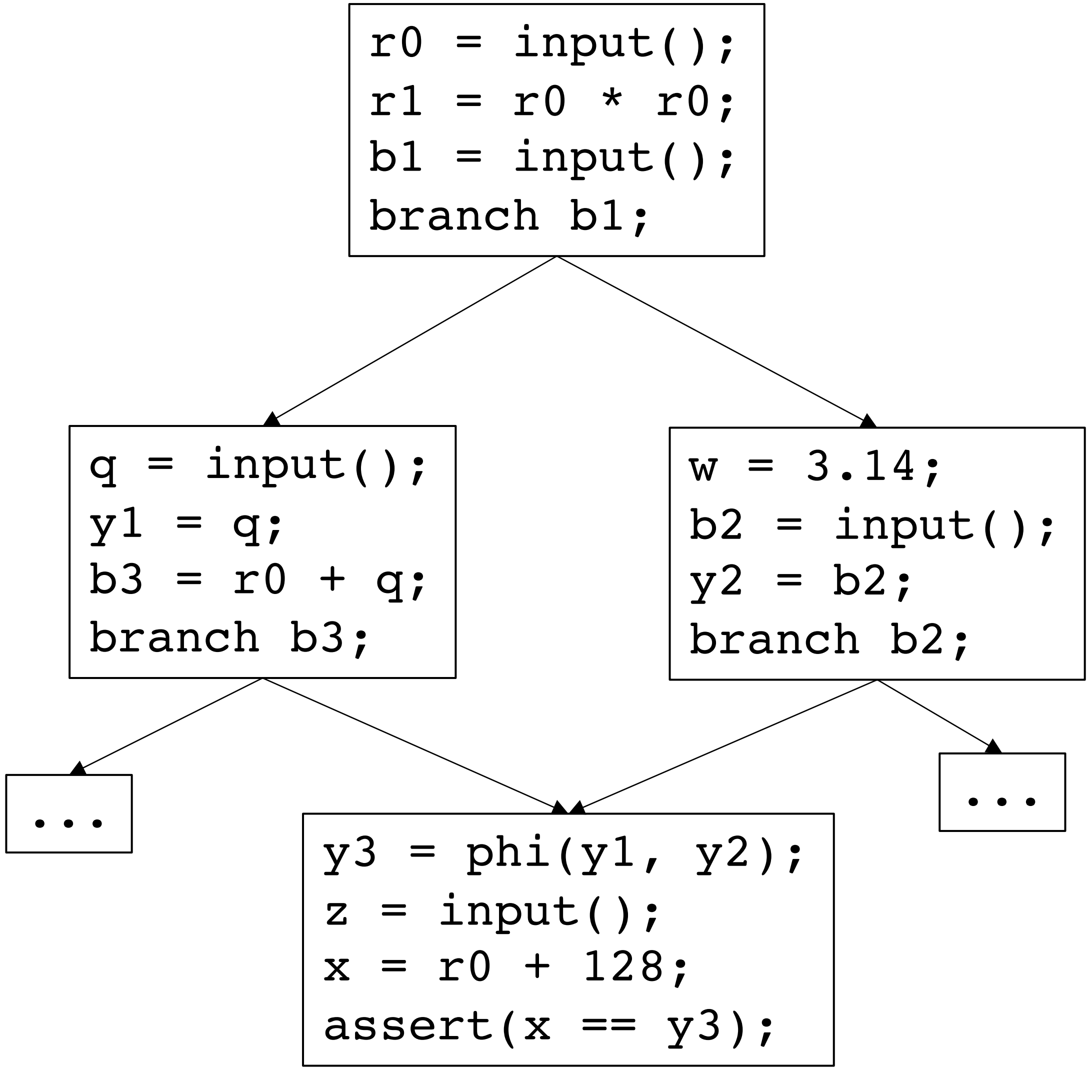


Backwards slicing algorithm

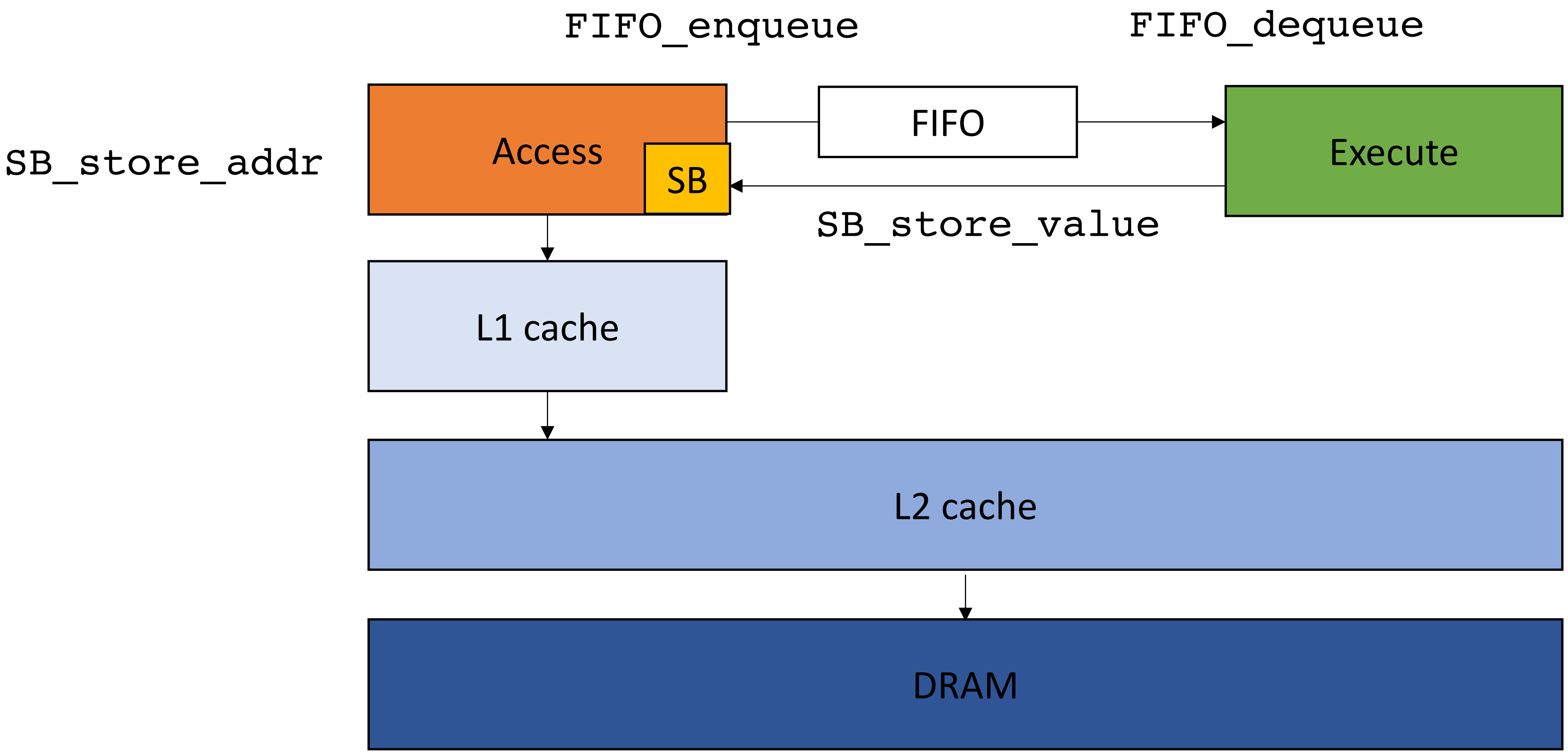
```
Worklist = S; // slicing criteria
while (!Worklist.empty()) {
  stmt = Worklist.pop();
  if (is_marked(stmt)) {
    continue;
  }
  mark(stmt);
  for a in stmt.args() {
    worklist.append(a);
  }
  for p in cfg[stmt].predecessors() {
    worklist.append(p.branch_stmt());
  }
}
```

marked:	worklist:
	branch b2
	r0
assert()	y1
x	y2
y3	b3
branch b3	branch b1

rest of example is an exercise



Back to DAE


FIFO_enqueue
FIFO_dequeue
SB_store_addr
Access
SB
FIFO
Execute
SB_store_value
L1 cache
L2 cache
DRAM

Compiler

Step 1: compile to SSA

```
for (int i = 0; i < SIZE; i++) {
   a[i] = b[i] * 3.14;
}
```

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   float r1 = r0 * 3.14;
   store(a + i, r1);
}
```

Compiler

Step 2: Create two copies, one for the access and one for the execute

Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   float r1 = r0 * 3.14;
   store(a + i, r1);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   float r1 = r0 * 3.14;
   store(a + i, r1);
}
```

Compiler

Step 3: Replace loads in Execute with FIFO reads, stores with SB_store_values

Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   float r1 = r0 * 3.14;
   store(a + i, r1);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   float r1 = r0 * 3.14;
   store(a + i, r1);
}
```

Compiler

Step 3: Replace loads in Execute with FIFO reads

Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   float r1 = r0 * 3.14;
   store(a + i, r1);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

Compiler

Step 4: Enqueue loaded values on the Access. Store addresses instead of values

Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   float r1 = r0 * 3.14;
   store(a + i, r1);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

Compiler

Step 4: Enqueue loaded values on the Access. Store addresses instead of values

Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   float r1 = r0 * 3.14;
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

Compiler

Step 5: Slice the Execute on all FIFO dequeue and SB store value calls

Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   float r1 = r0 * 3.14;
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

Compiler

Step 6: Slice the Access on all FIFO enqueue and SB store address calls

Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   float r1 = r0 * 3.14;
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

Compiler

Step 6: Slice the Access on all FIFO enqueue and SB store address calls

Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   float r1 = r0 * 3.14;
   SB_store_addr(a + i);
}
```

Execute

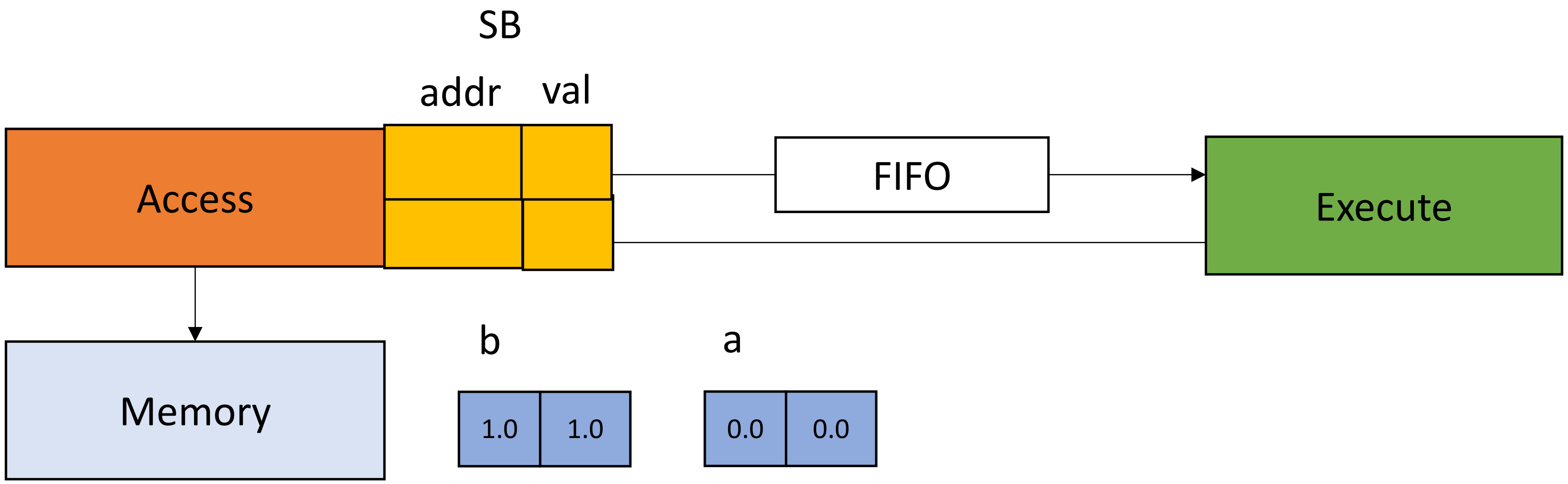
```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```



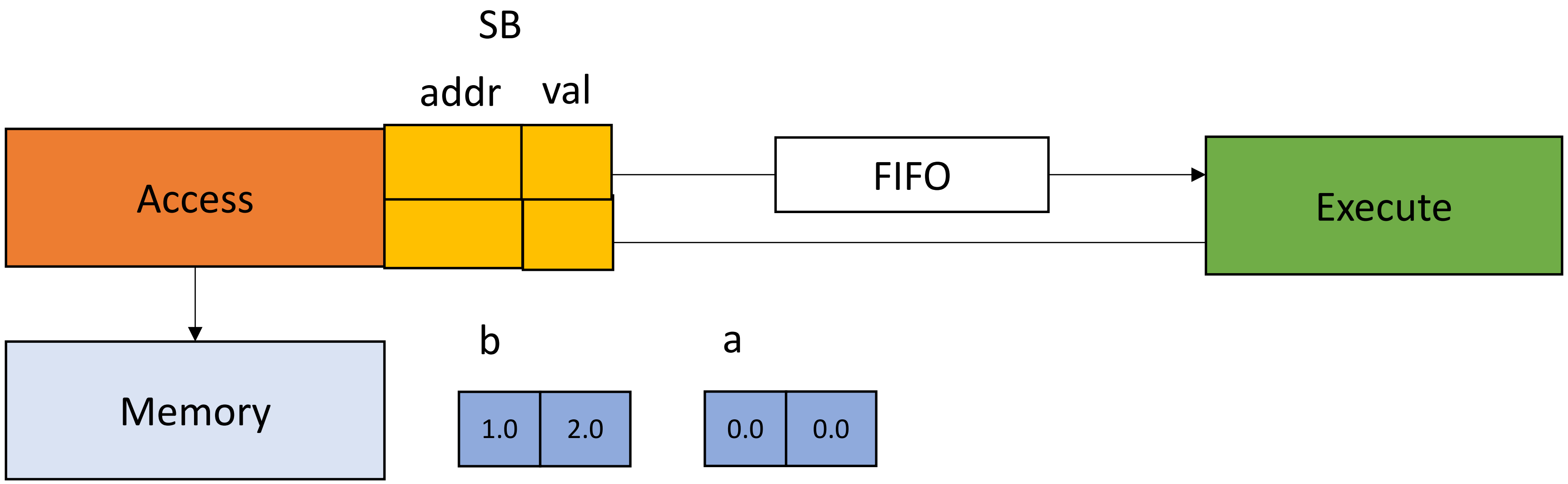
Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

blocks until queue has an item to dequeue



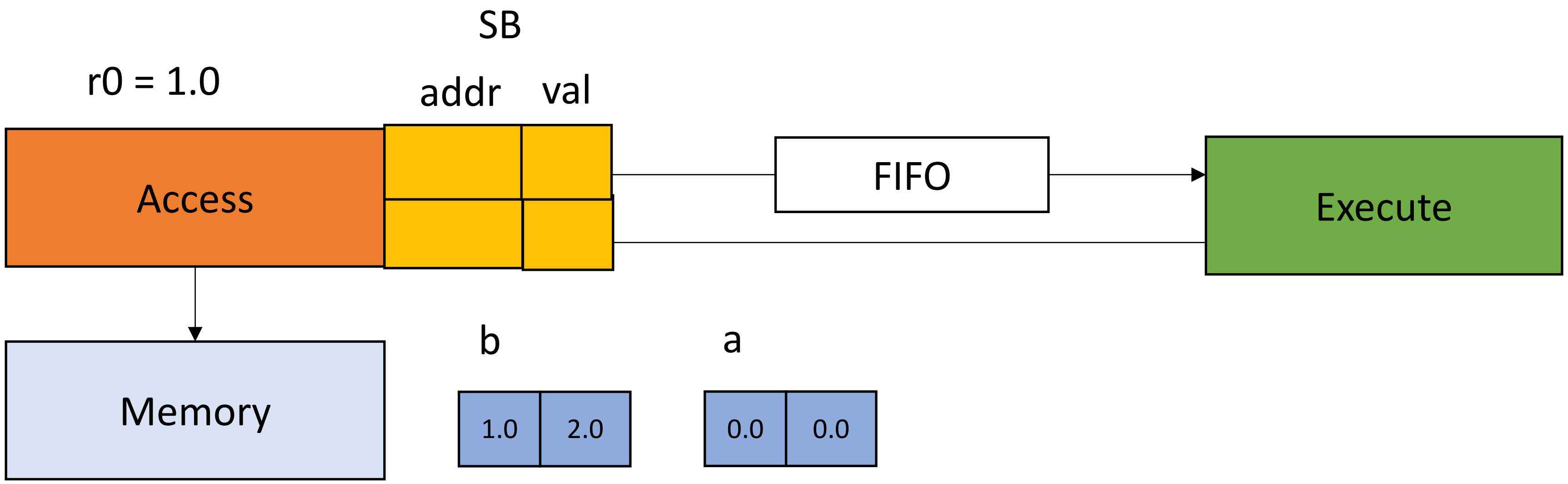
Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

blocks until queue
has an item to dequeue



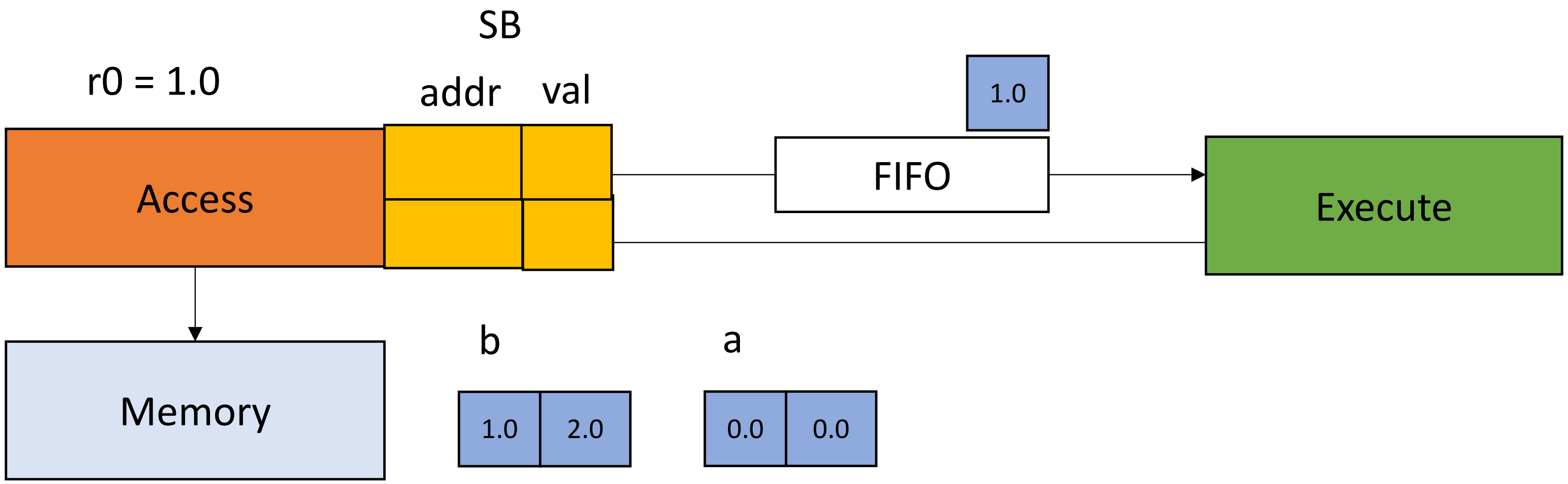
Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

blocks until queue has an item to dequeue



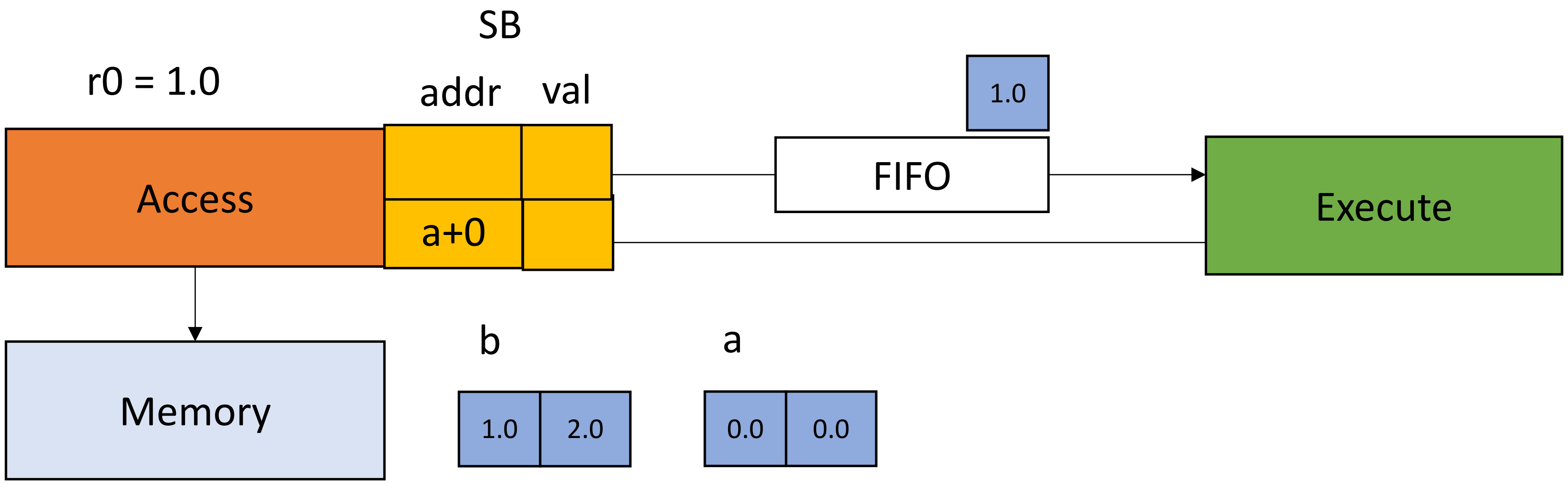
Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

blocks until queue has an item to dequeue



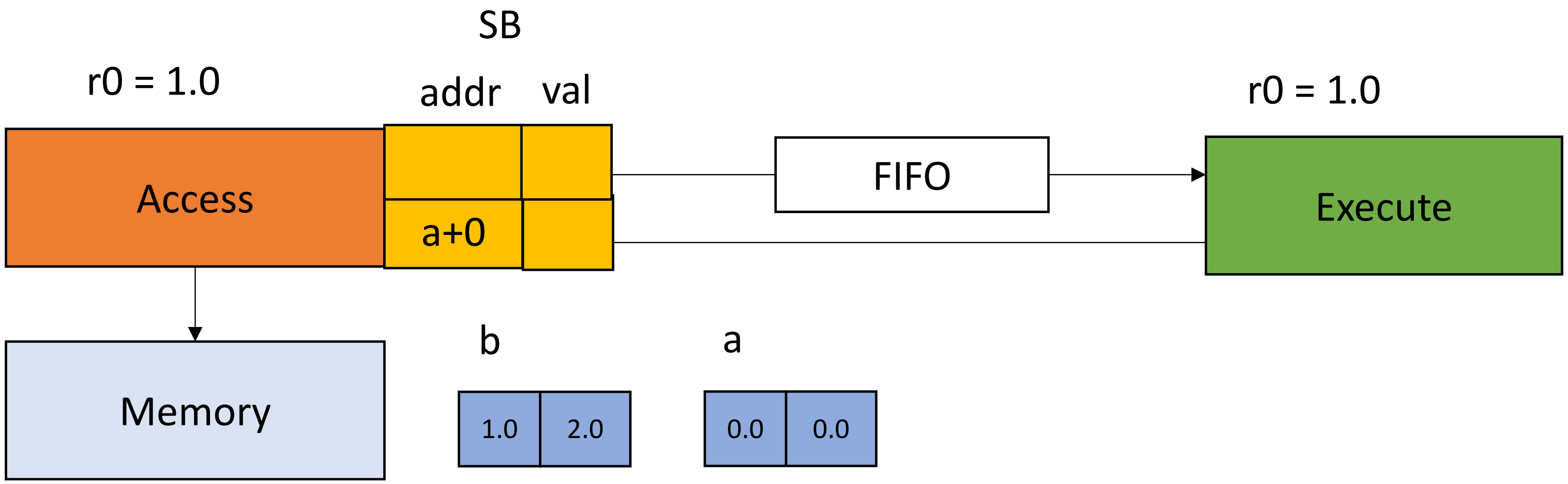
Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

blocks until queue has an item to dequeue



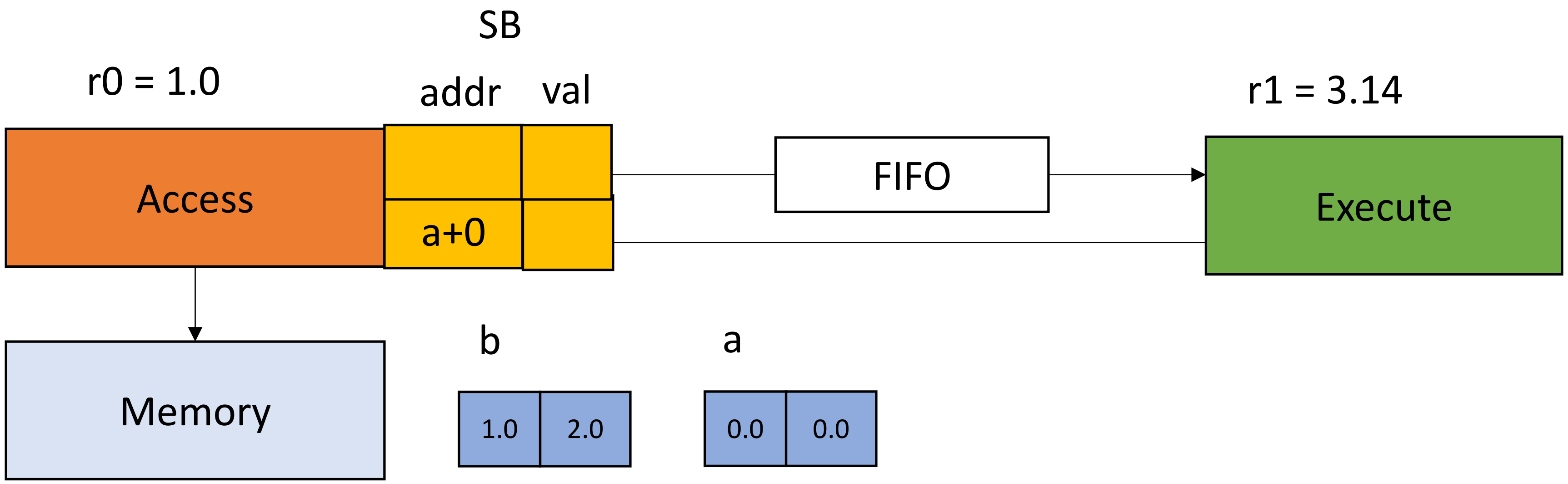
Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

blocks until queue has an item to dequeue



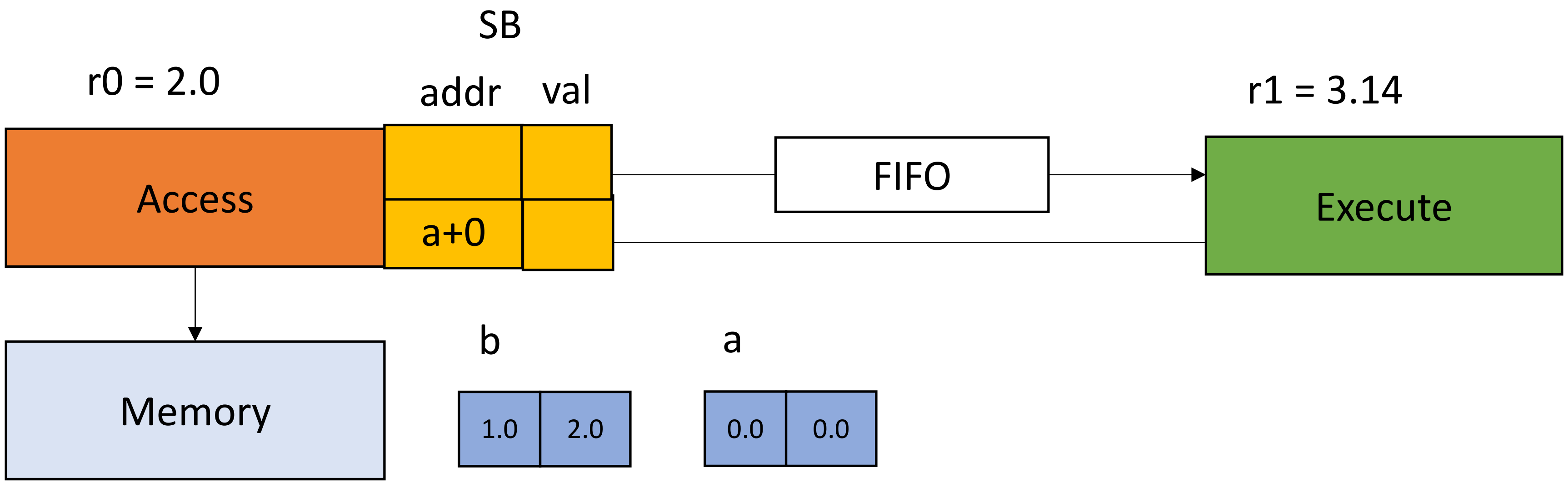
Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

blocks until queue
has an item to dequeue



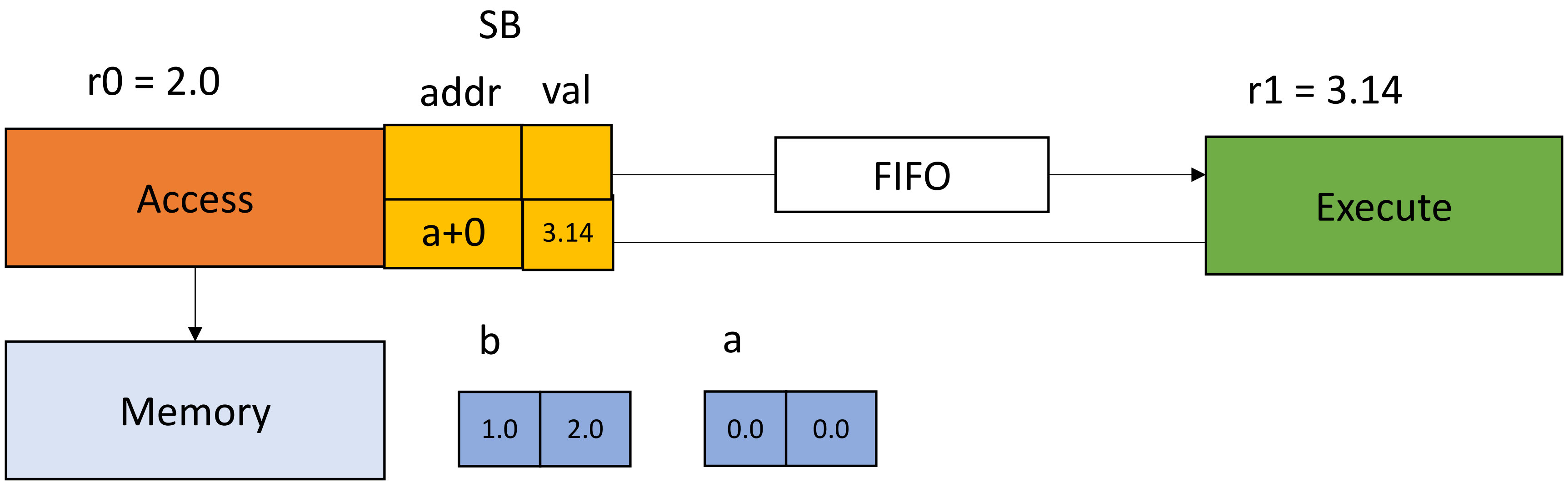
Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

blocks until queue has an item to dequeue



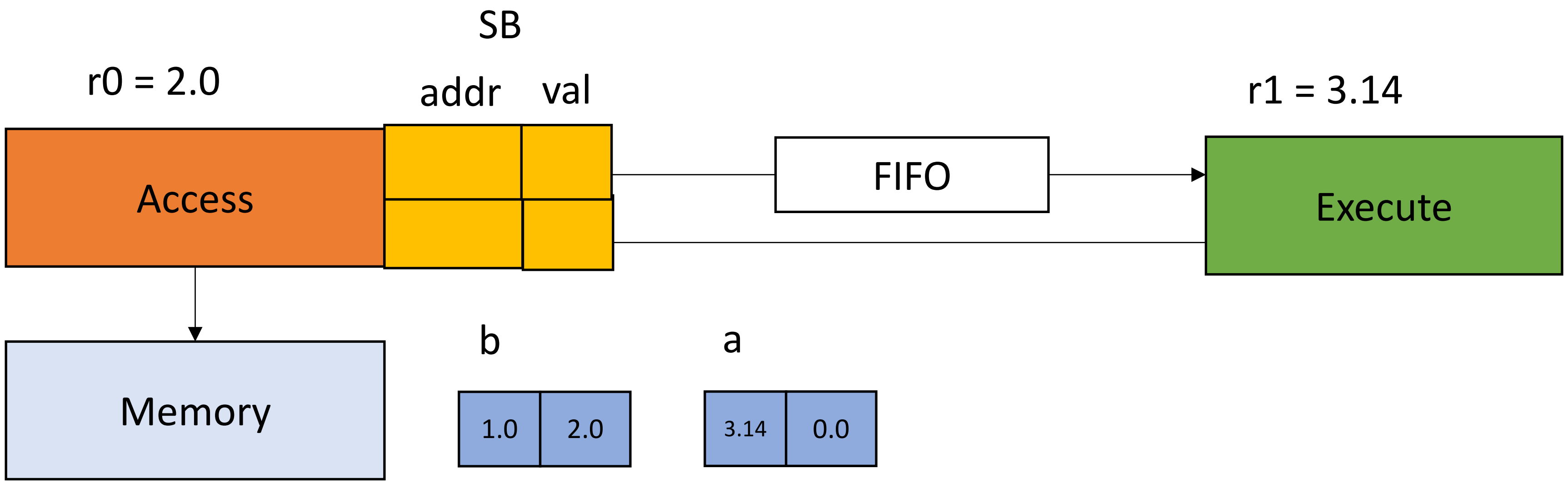
Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

blocks until queue has an item to dequeue



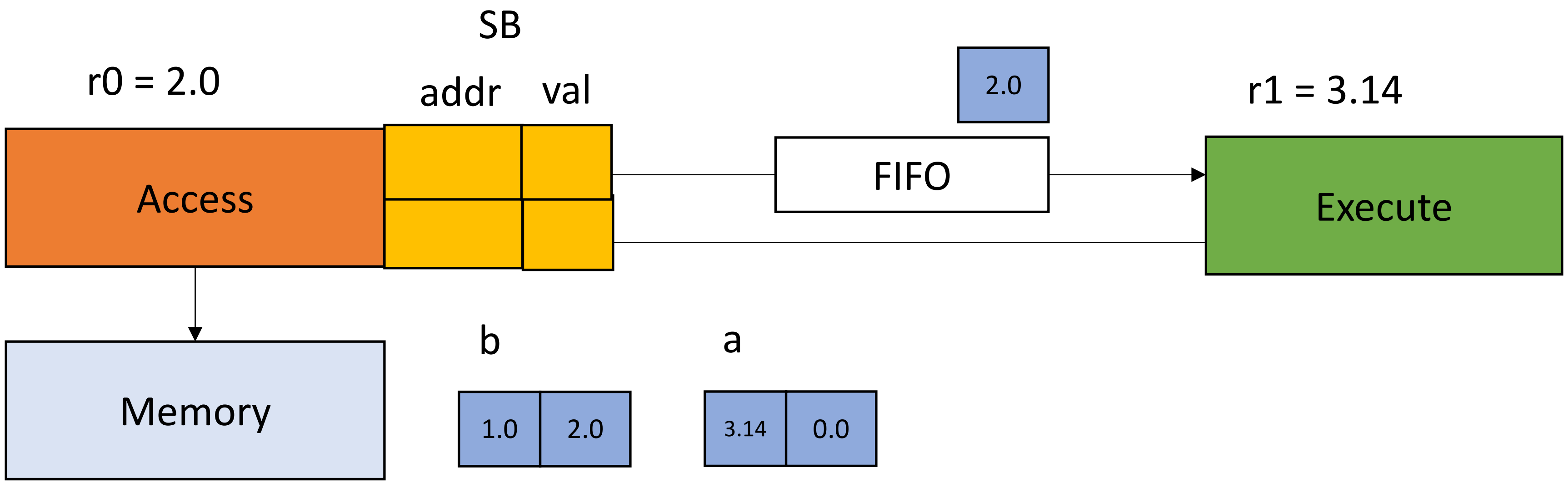
Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

blocks until queue has an item to dequeue



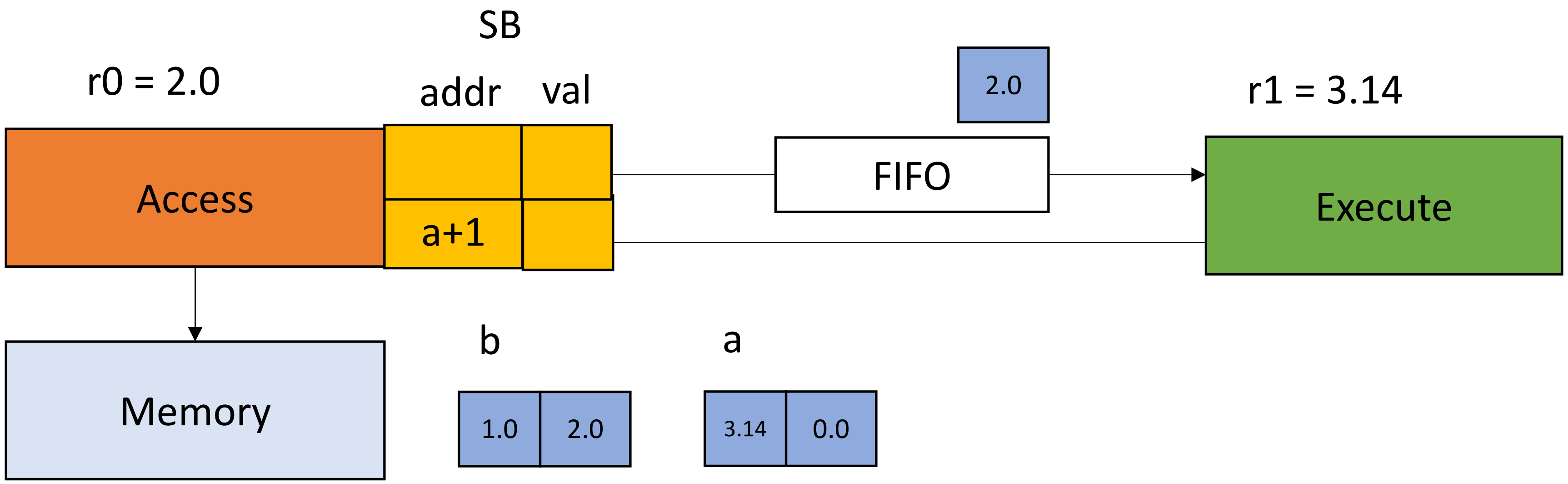
Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

blocks until queue has an item to dequeue



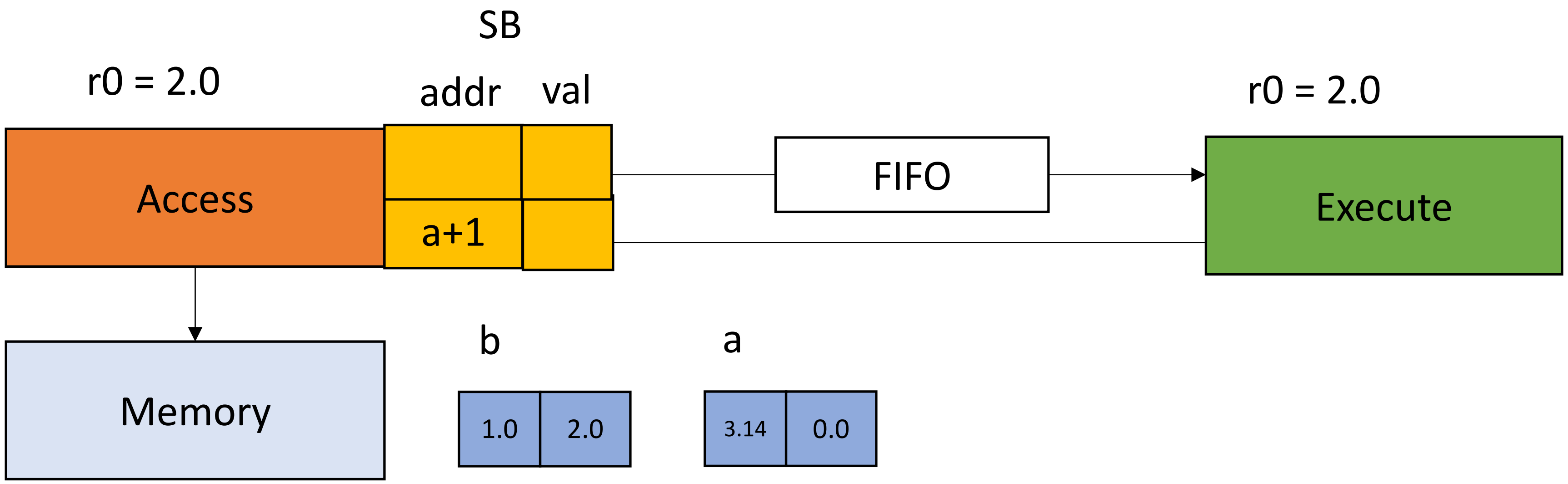
Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

blocks until queue has an item to dequeue



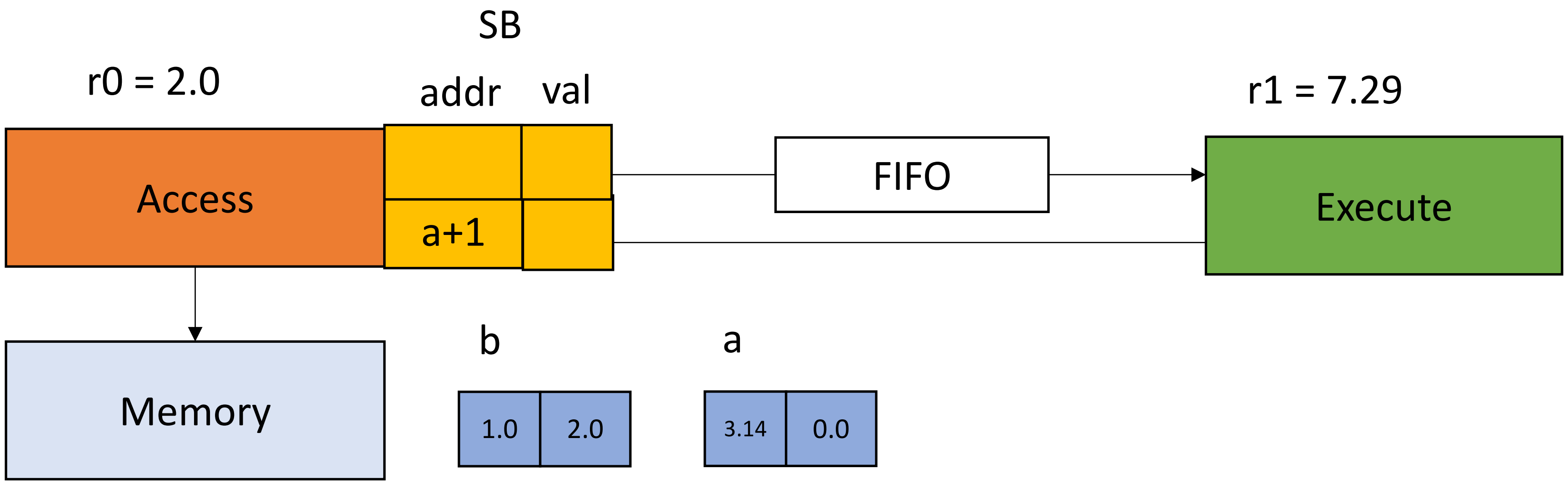
Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

blocks until queue has an item to dequeue



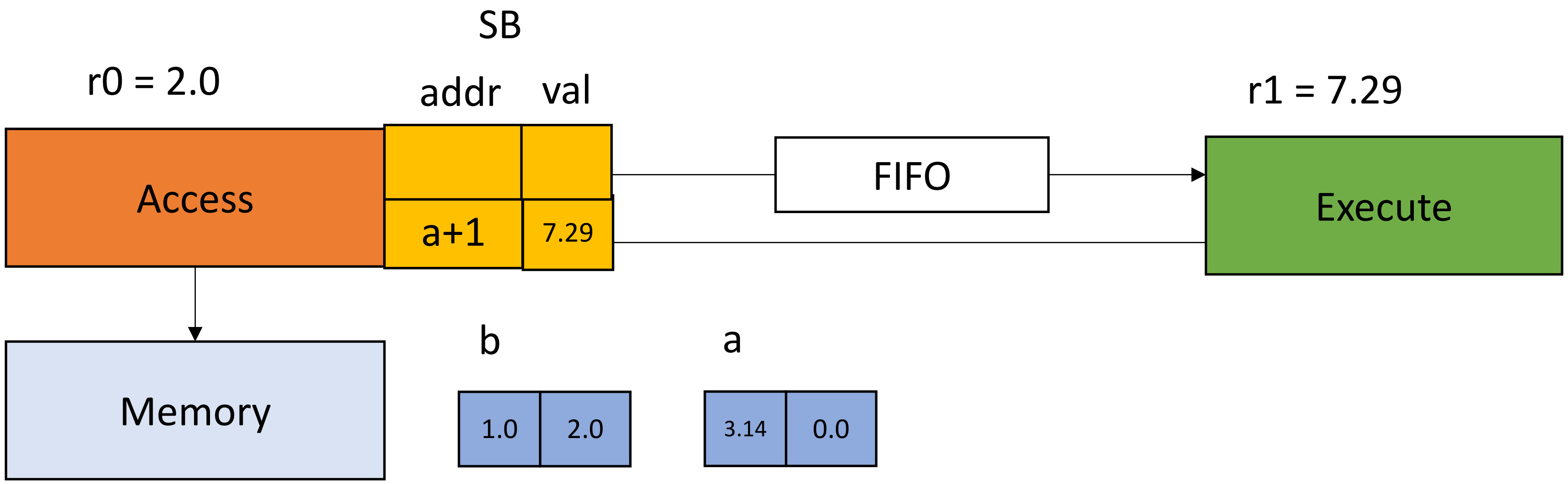
Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

blocks until queue has an item to dequeue



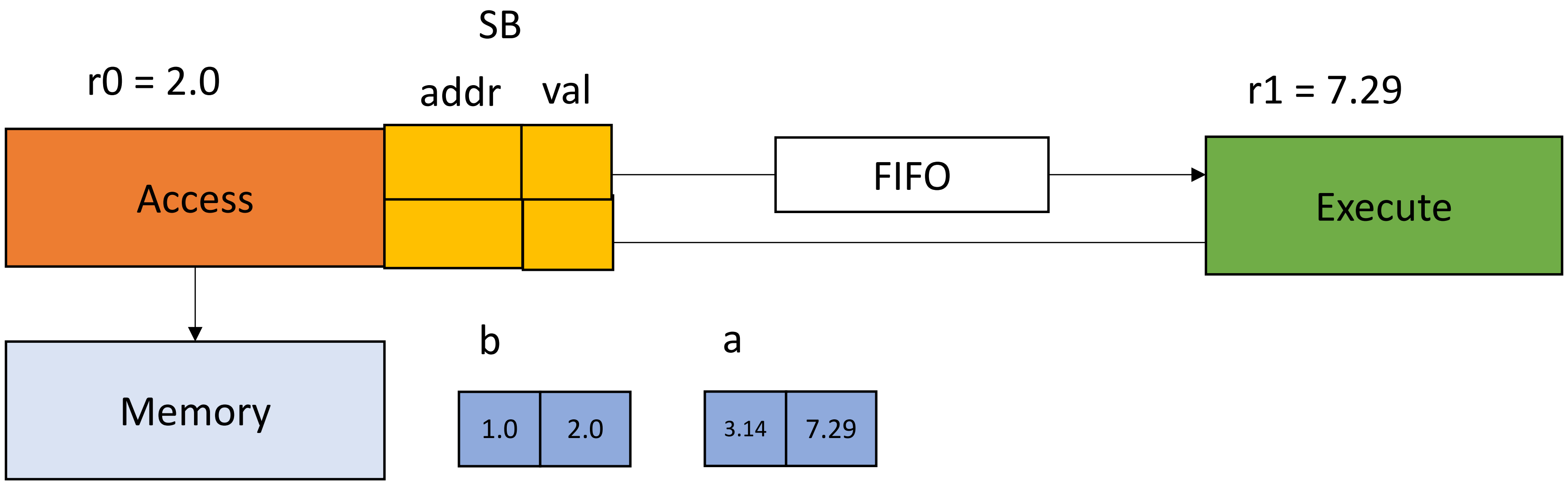
Access

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = load(b + i);
   FIFO_enqueue(r0);
   SB_store_addr(a + i);
}
```

Execute

```
// SSA pseudo code
for (int i = 0; i < SIZE; i++) {
   float r0 = FIFO_dequeue()
   float r1 = r0 * 3.14;
   SB_store_value(r1);
}
```

blocks until queue has an item to dequeue



Performance bounds

- A program p has execution time of $E(p)$. The time spent on compute (arithmetic) is $C(p)$. The time spent on memory latency is $M(p)$.
- For simple core models, we can approximate: $E(p) = C(p) + M(p)$
  - Why might this not be completely accurate for more complex cores?
- In DAE, the Execute time ideally is $C(p)$, and the Access ideally is $M(p)$.
- Optimistic estimates of DAE performance is
  - $\max(C(p), M(p))$
  - best case is when $C(p) \sim = M(p)$, we get $2x$ performance increase
- Pros/cons?

Other considerations

- Dependencies:
  - If Access depends on a value from Execute, performance can suffer
  - Also called LoD (loss of decoupling events)

- Coherence:
  - Access must read up-to-date values, even when waiting on Execute

- More optimizations:
  - Similar to asynchronous stores, some loads can be done asynchronously as well (if the value is not needed by the Access).

On Wednesday

- Start the module on DSLs by talking about Halide!