NORTHWEST INSTITUTE *for* ADVANCED COMPUTING

AMATH 483/583
High Performance Scientific Computing

Lecture 6:
High Performance in Hierarchical Memory

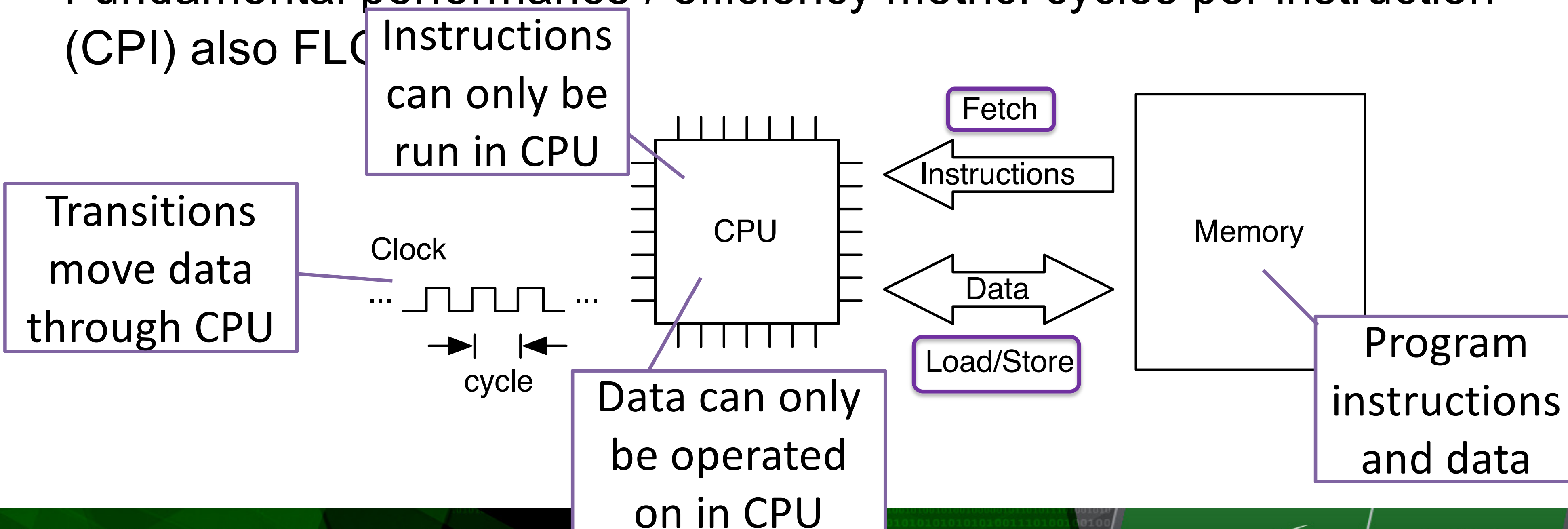
Andrew Lumsdaine
Northwest Institute for Advanced Computing
Pacific Northwest National Laboratory
University of Washington
Seattle, WA

Overview

- "PDP-11" machine model
- Pipelining, pipeline stalls
- Hierarchical memory
- Timing and benchmarking
- Compiler optimizations
- Tiling
- Blocking

Microprocessors

- Basic operation: read and execute program instructions stored in memory
- Fundamental performance / efficiency metric: cycles per instruction (CPI) also FL

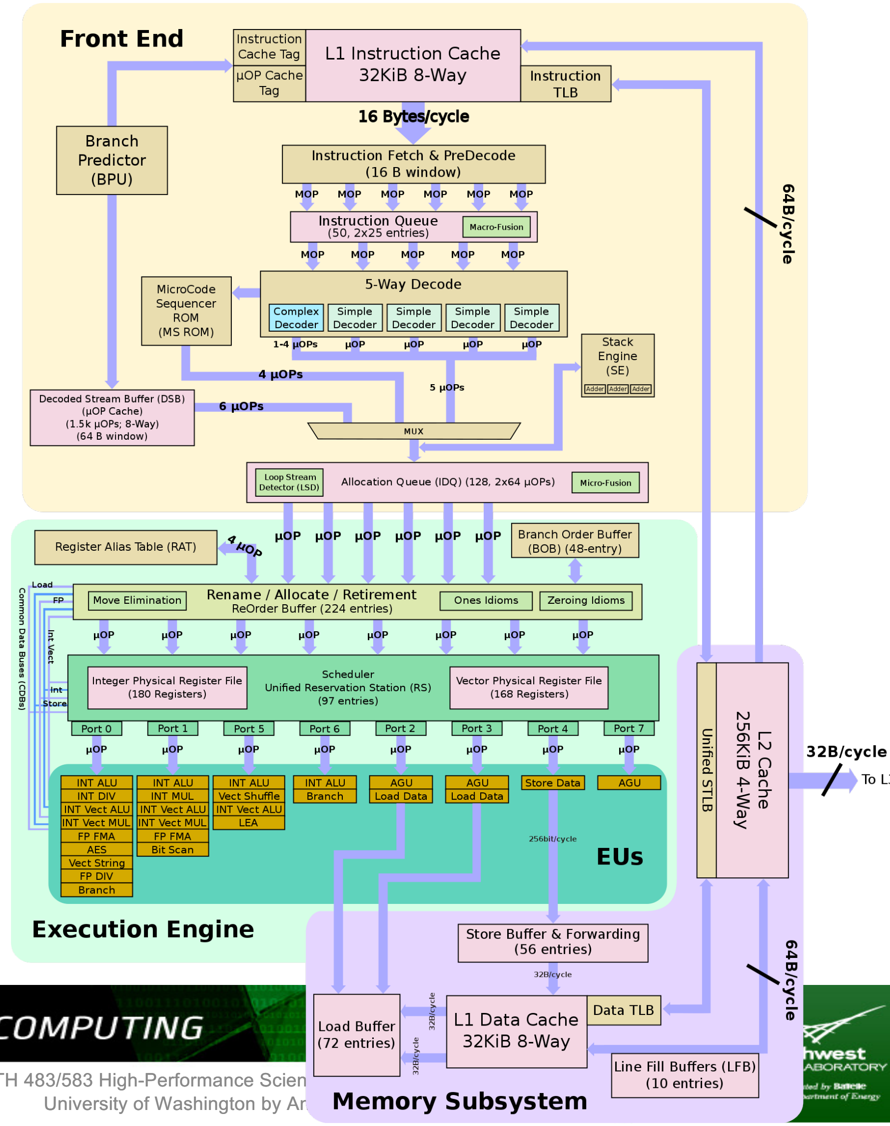


Performance-Oriented Architecture Features

- Execution Pipeline
  - Stages of functionality to process issued instructions
  - Hazards are conflicts with continued execution
  - Forwarding supports closely associated operations exhibiting precedence constraints
- Out of Order Execution
  - Uses reservation stations
  - Hides some core latencies and provide fine grain asynchronous operation supporting concurrency
- Branch Prediction
  - Permits computation to proceed at a conditional branch point prior to resolving predicate value
  - Overlaps follow-on computation with predicate resolution
  - Requires roll-back or equivalent to correct false guesses
  - Sometimes follows both paths, and several deep

Skylake

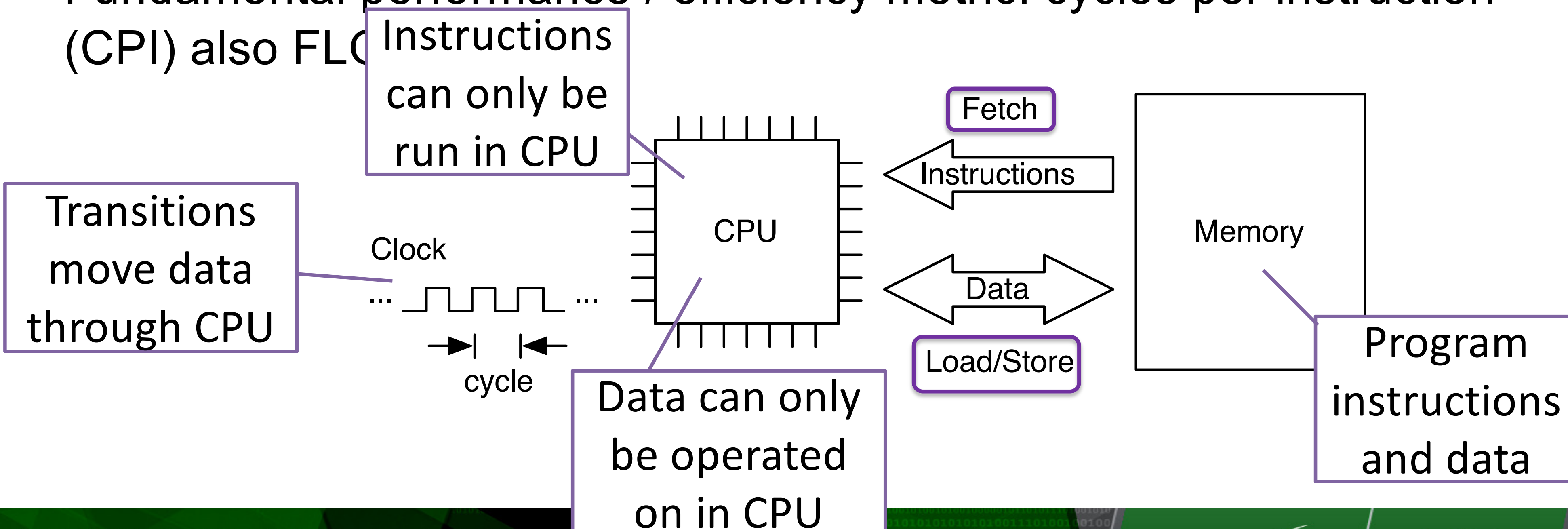
L1D$
Execution Units
Mem Management & Execution
L2$
OoO Scheduling Retirement
Decode & MSROM
L1I$



Front End
Instruction Cache Tag
µOP Cache Tag
L1 Instruction Cache 32KiB 8-Way
Instruction TLB
16 Bytes/cycle
Branch Predictor (BPU)
Instruction Fetch & PreDecode (16 B window)
MOP
Instruction Queue (50, 2x25 entries)
Macro-Fusion
MicroCode Sequencer ROM (MS ROM)
5-Way Decode
Complex Decoder
Simple Decoder
1-4 µOPs
µOP
4 µOPs
5 µOPs
Stack Engine (SE)
Decoded Stream Buffer (DSB) (µOP Cache) (1.5k µOPs; 8-Way) (64 B window)
6 µOPs
MUX
Loop Stream Detector (LSD)
Allocation Queue (IDQ) (128, 2x64 µOPs)
Micro-Fusion
64B/cycle
Register Alias Table (RAT)
4 µOP
Branch Order Buffer (BOB) (48-entry)
Rename / Allocate / Retirement
ReOrder Buffer (224 entries)
Move Elimination
Ones Idioms
Zeroing Idioms
Common Data Buses (CDBs)
Load
FP
Int Vect
Int
Store
Integer Physical Register File (180 Registers)
Scheduler
Unified Reservation Station (RS) (97 entries)
Vector Physical Register File (168 Registers)
Port 0
Port 1
Port 5
Port 6
Port 2
Port 3
Port 4
Port 7
INT ALU
INT DIV
INT Vect ALU
INT Vect MUL
FP FMA
AES
Vect String
FP DIV
Branch
INT MUL
Bit Scan
Vect Shuffle
LEA
AGU
Load Data
Store Data
256bit/cycle
EUs
Execution Engine
Unified STLB
L2 Cache 256KiB 4-Way
32B/cycle
To L3
Store Buffer & Forwarding (56 entries)
32B/cycle
Load Buffer (72 entries)
L1 Data Cache 32KiB 8-Way
Data TLB
Line Fill Buffers (LFB) (10 entries)
Memory Subsystem

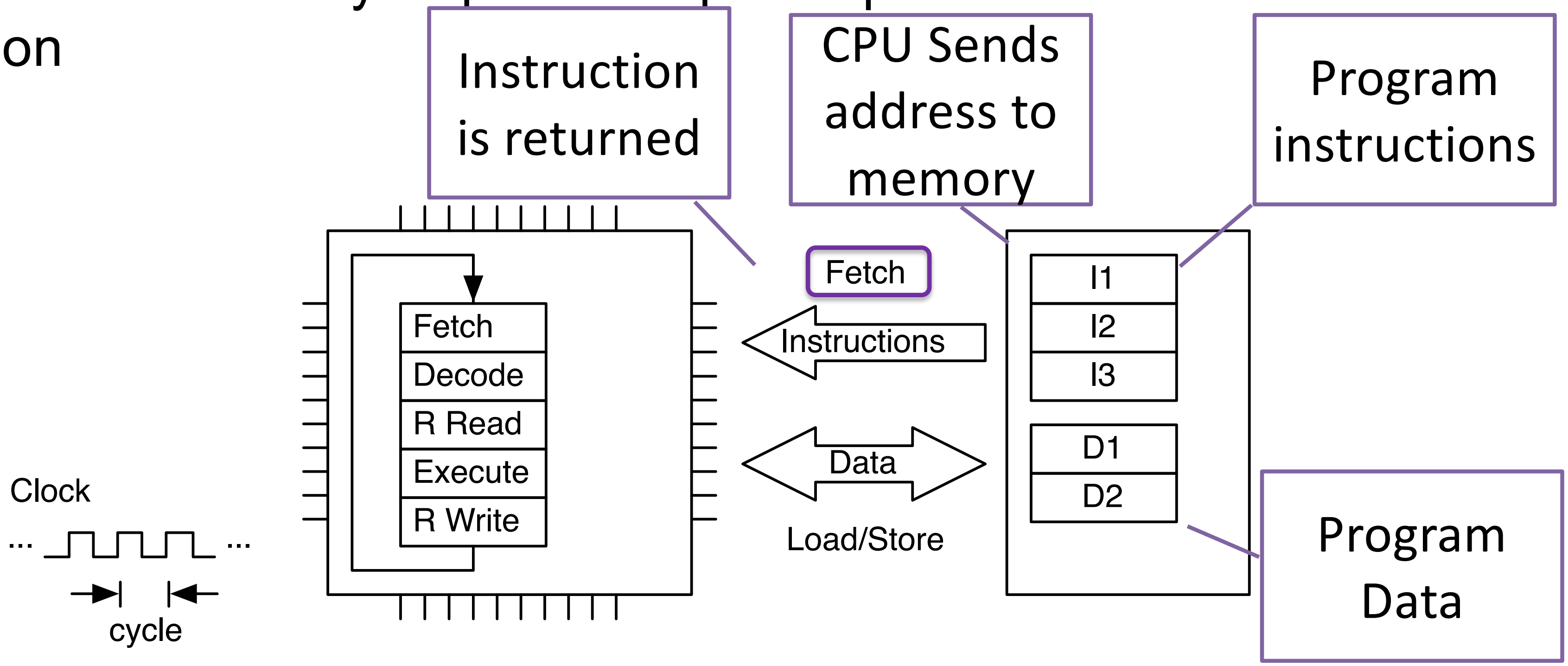
Microprocessors

- Basic operation: read and execute program instructions stored in memory
- Fundamental performance / efficiency metric: cycles per instruction (CPI) also FL



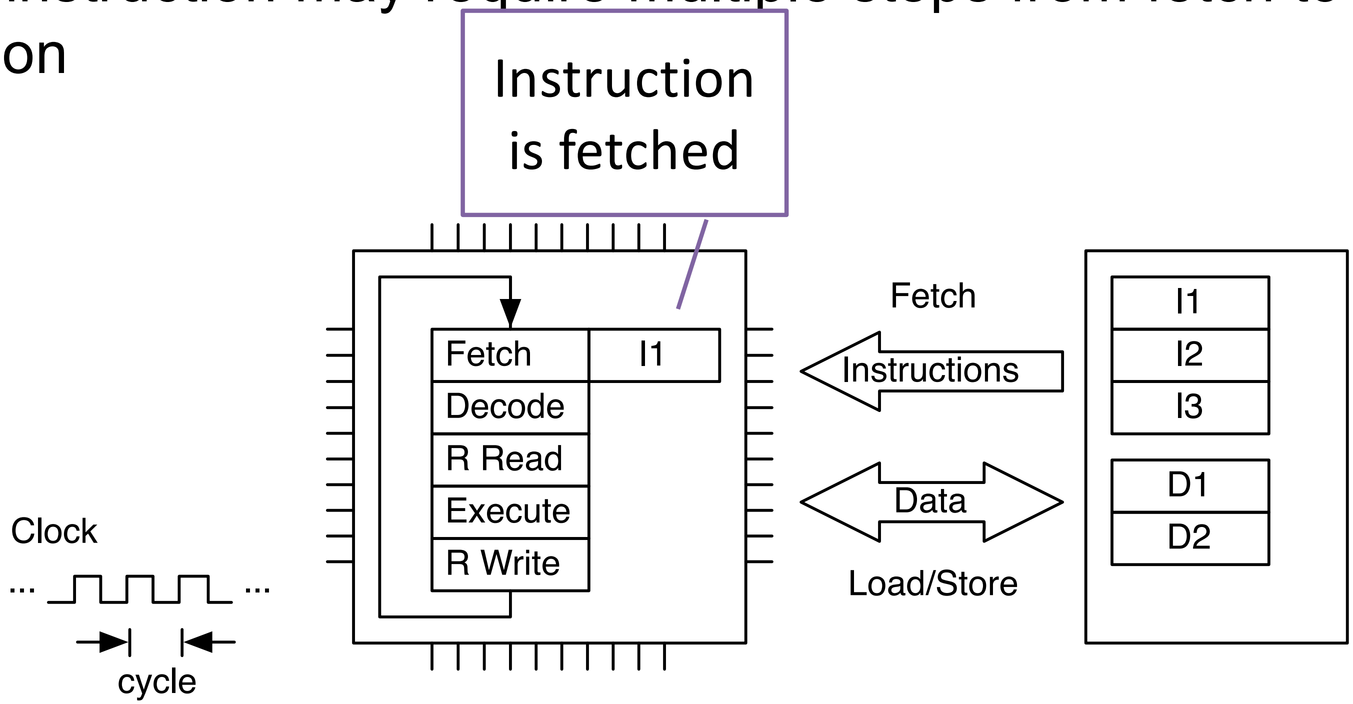
Processor Core Instruction Handling

- A single instruction may require multiple steps from fetch to completion



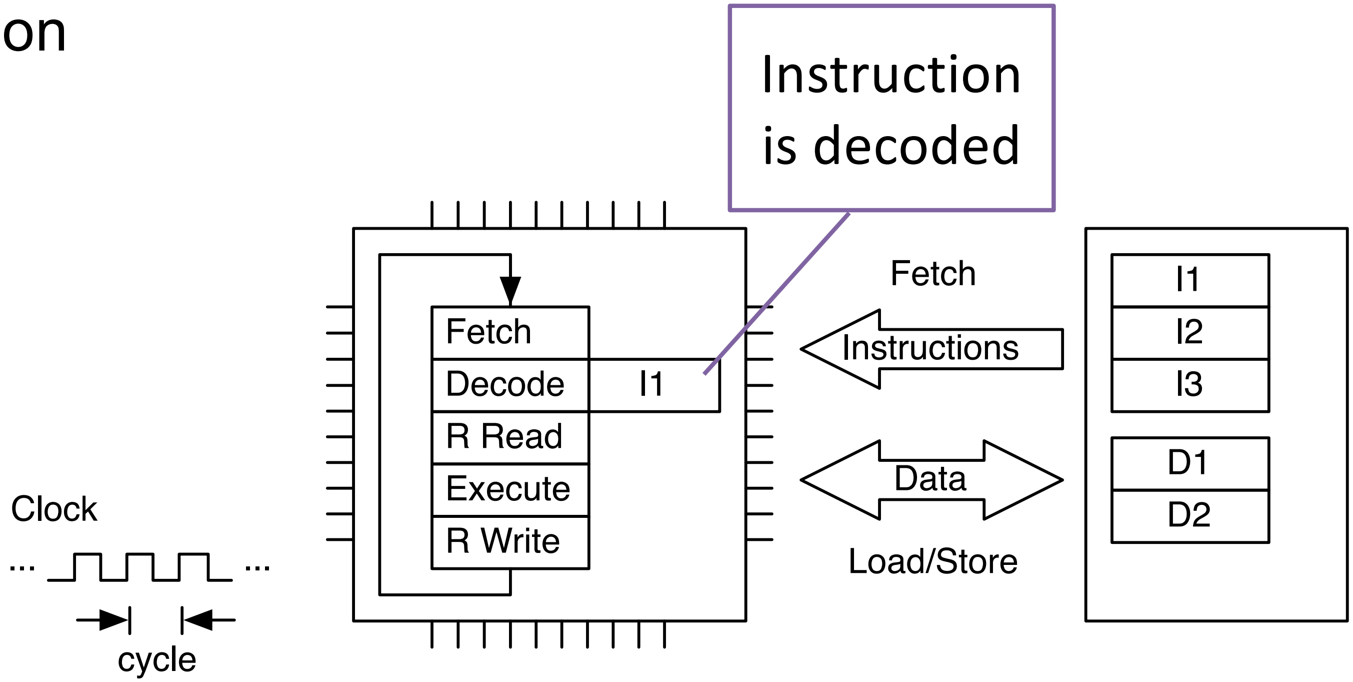
Processor Core Instruction Handling

- A single instruction may require multiple steps from fetch to completion



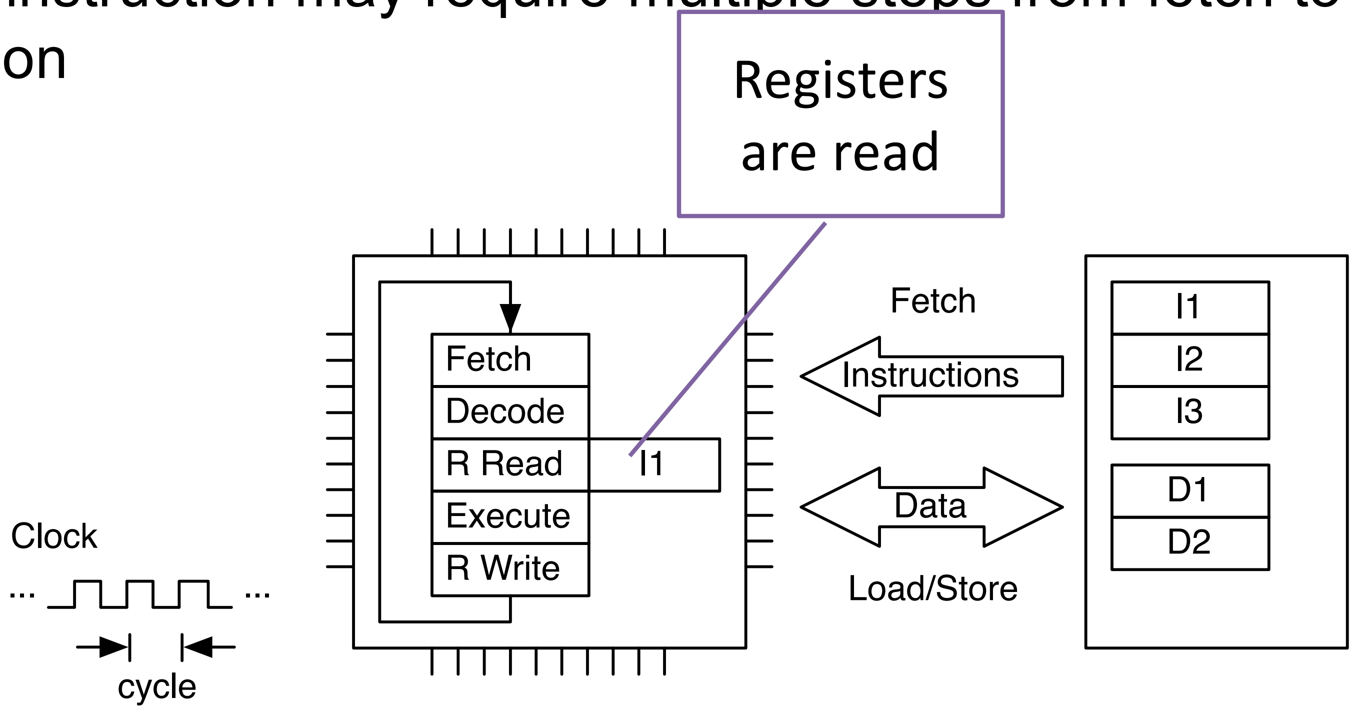
Processor Core Instruction Handling

- A single instruction may require multiple steps from fetch to completion



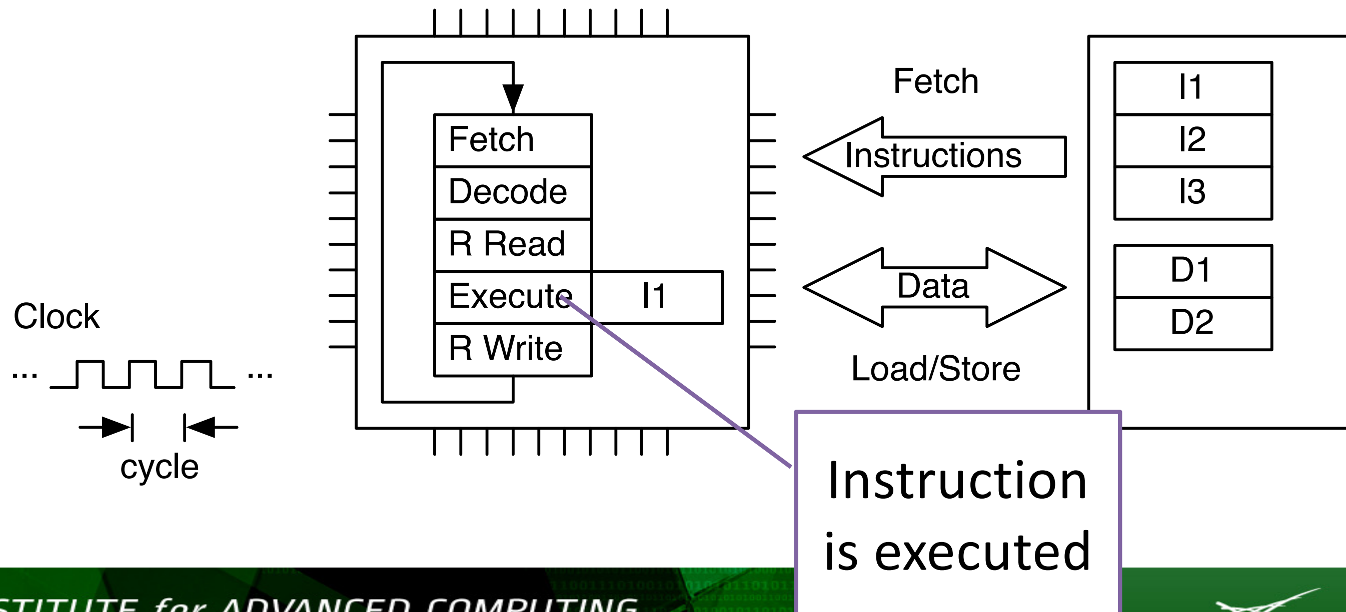
Processor Core Instruction Handling

- A single instruction may require multiple steps from fetch to completion



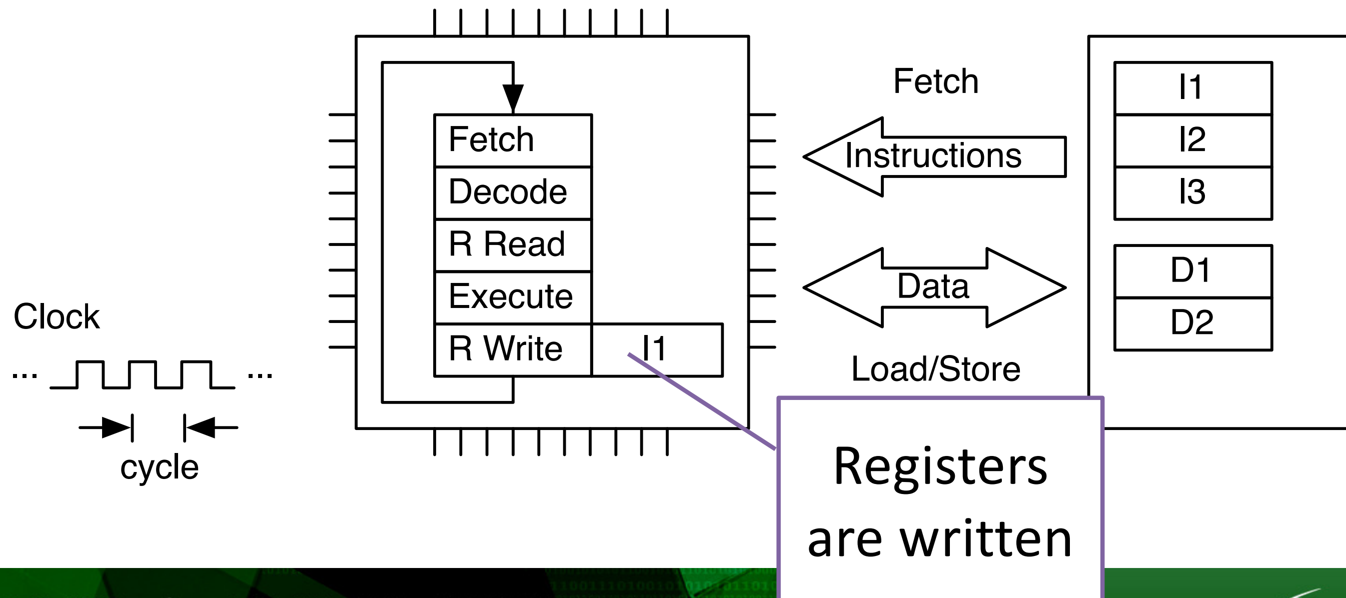
Processor Core Instruction Handling

- A single instruction may require multiple steps from fetch to completion



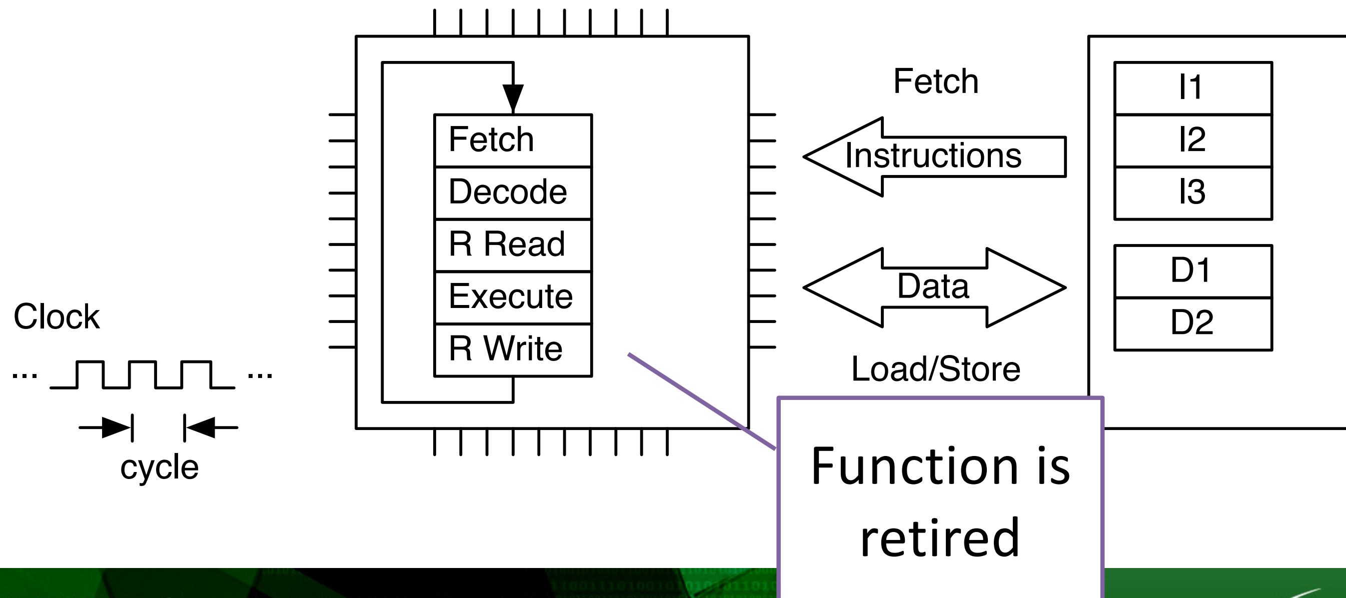
Processor Core Instruction Handling

- A single instruction may require multiple steps from fetch to completion



Processor Core Instruction Handling

- A single instruction may require multiple steps from fetch to completion



Processor Core Instruction Handling

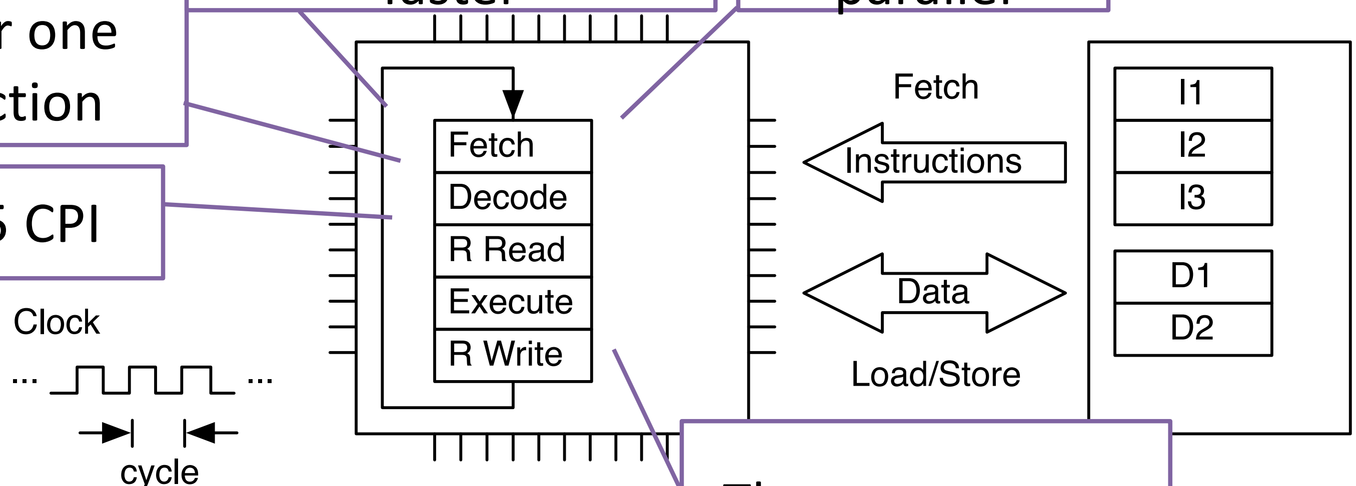
- A single instruction may require multiple steps from fetch to completion

How long did it take for one instruction

5 CPI

But we can't make one instruction 5X faster

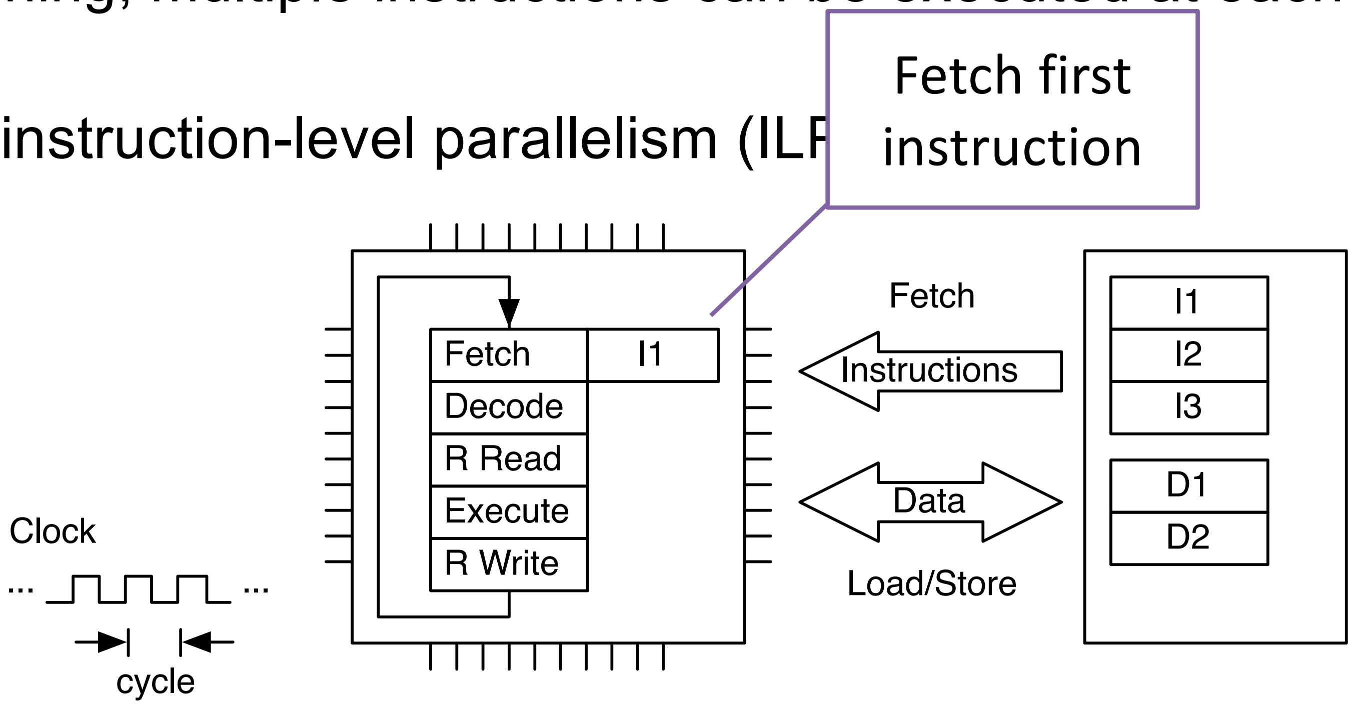
Let's use them in parallel



These stages can process in parallel

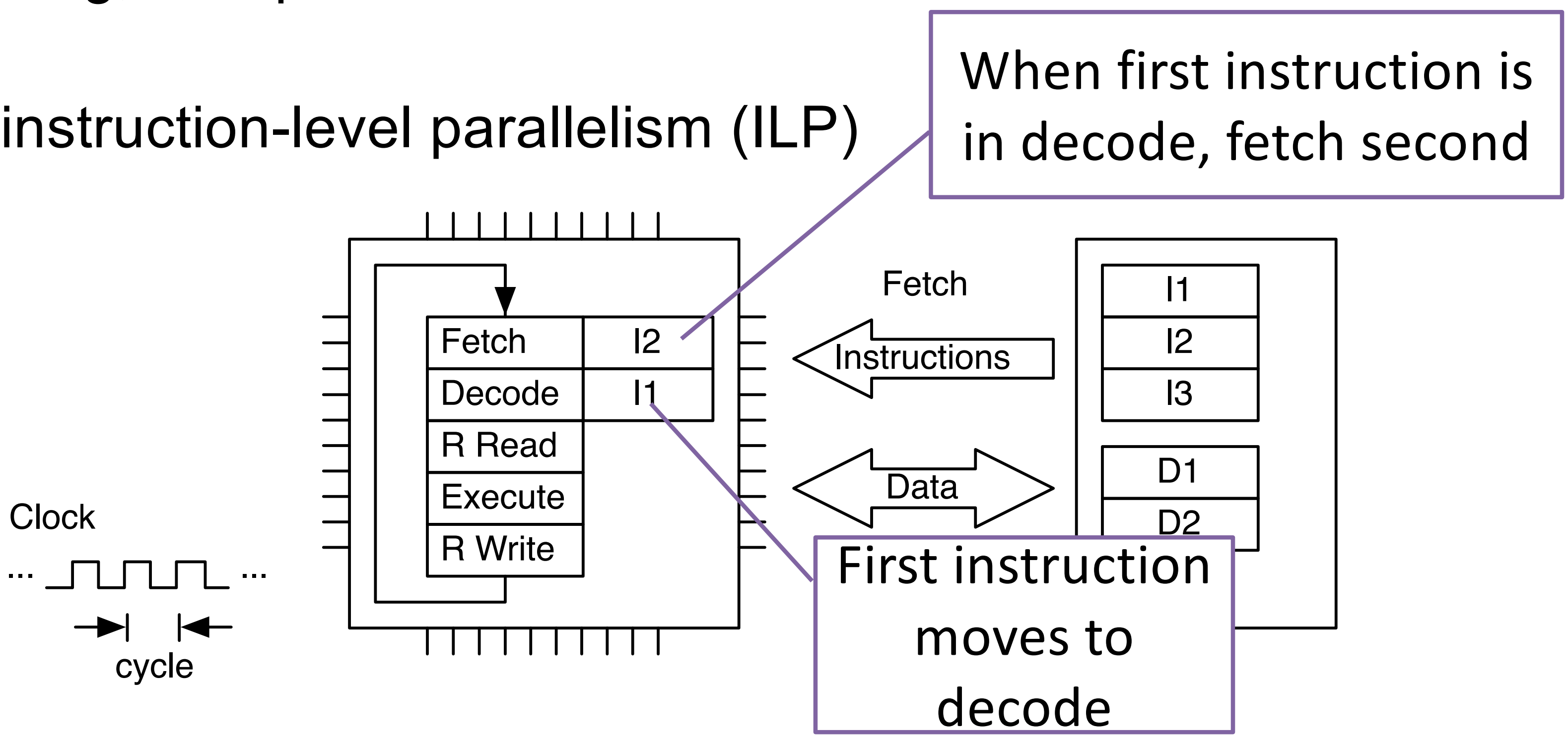
Processor Core Instruction Handling

- By pipelining, multiple instructions can be executed at each clock cycle
- Form of instruction-level parallelism (ILP)



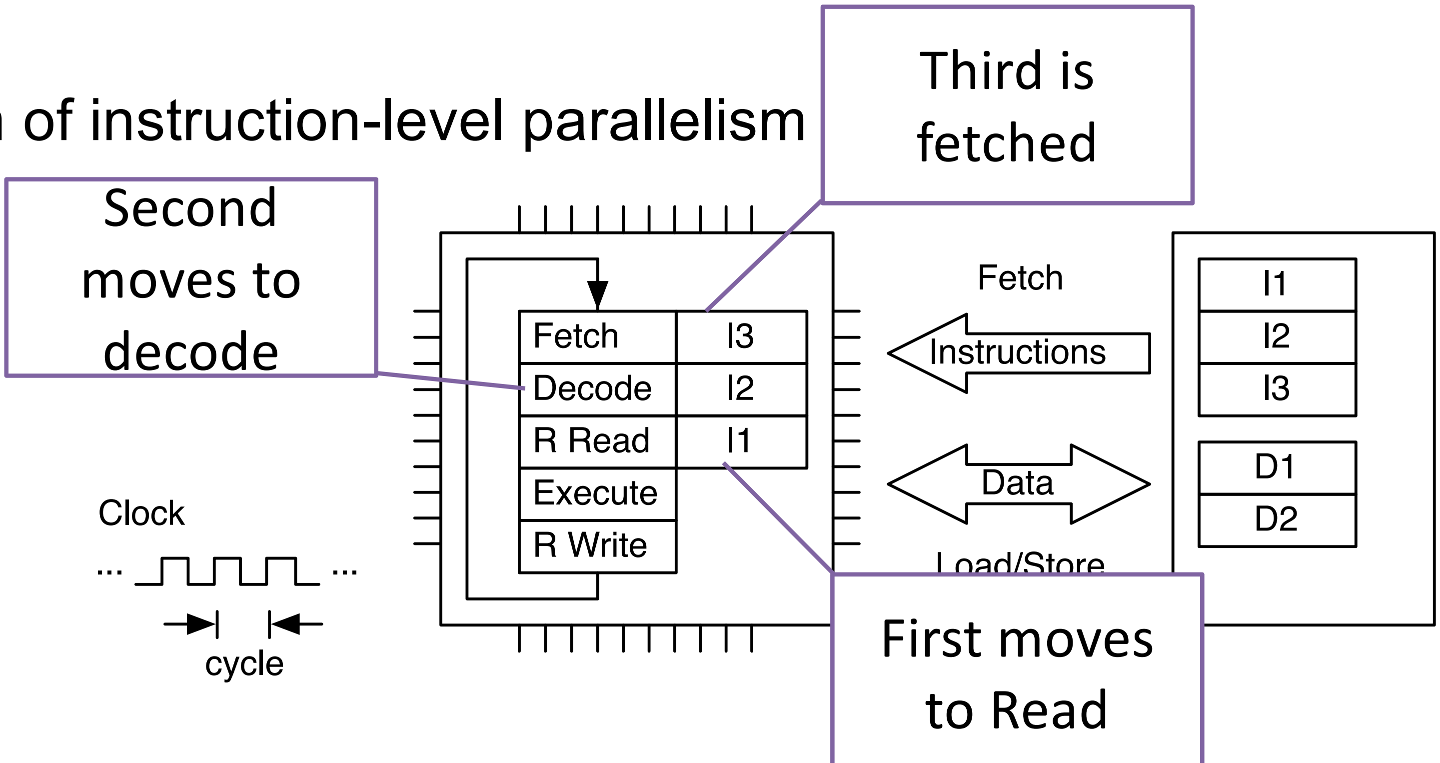
Processor Core Instruction Handling

- By pipelining, multiple instructions can be executed at each clock cycle
- Form of instruction-level parallelism (ILP)



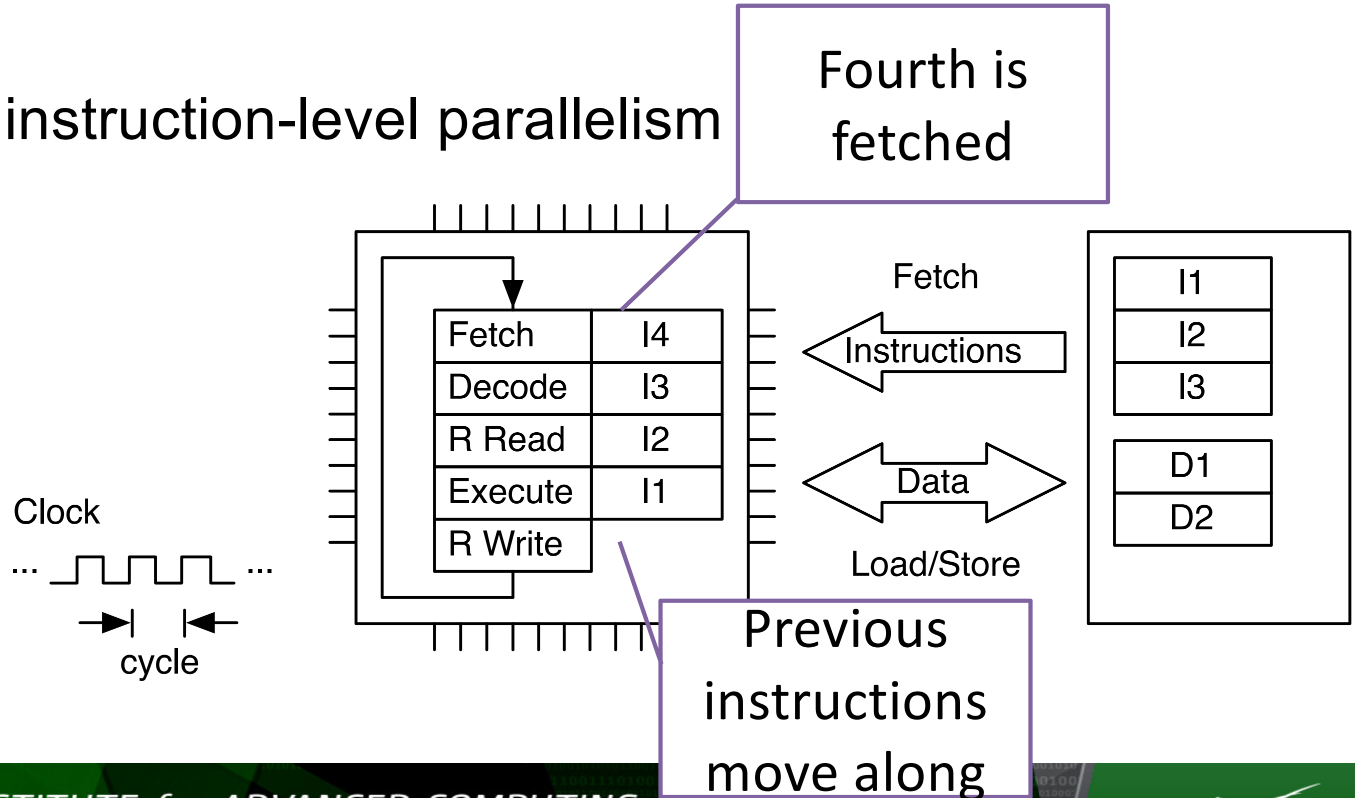
Processor Core Instruction Handling

- By pipelining, multiple instructions can be executed at each clock cycle
- Form of instruction-level parallelism



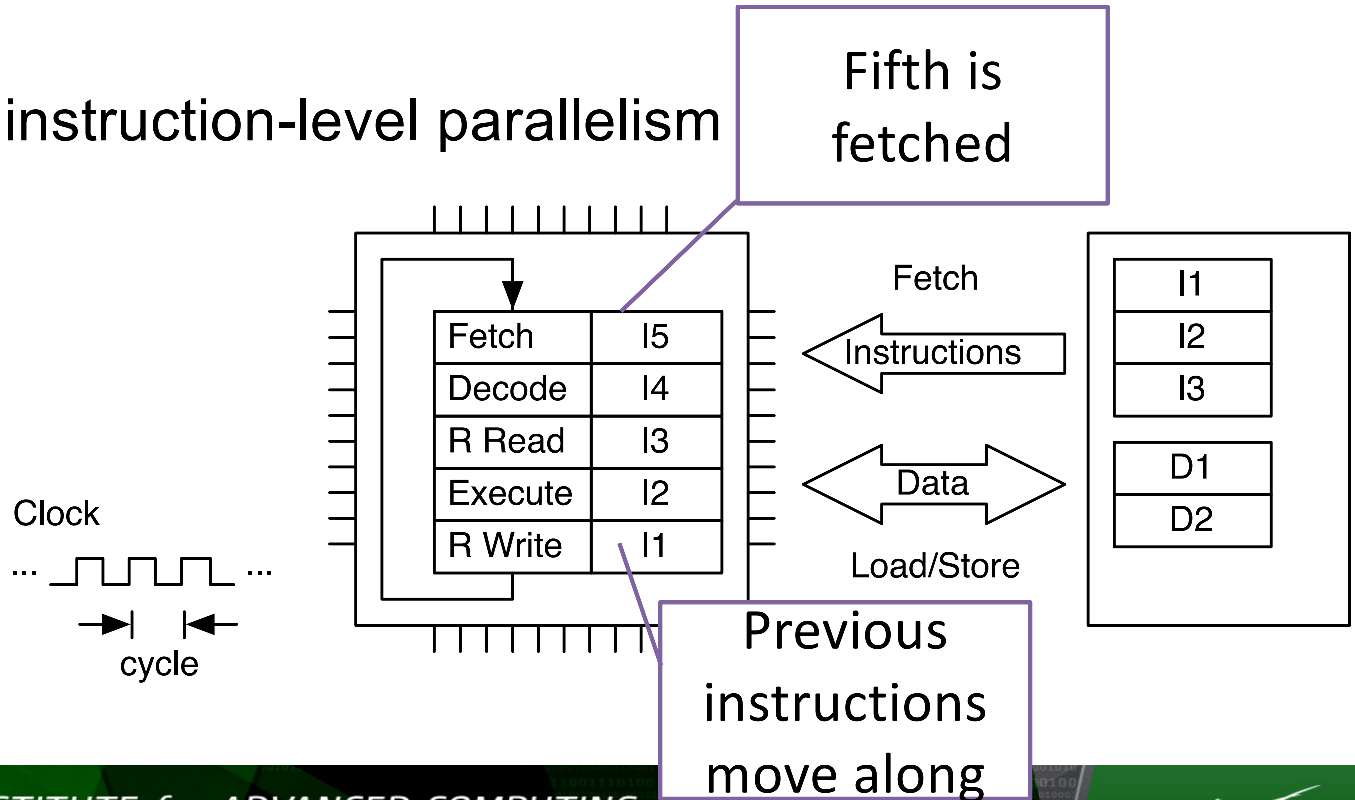
Processor Core Instruction Handling

- By pipelining, multiple instructions can be executed at each clock cycle
- Form of instruction-level parallelism



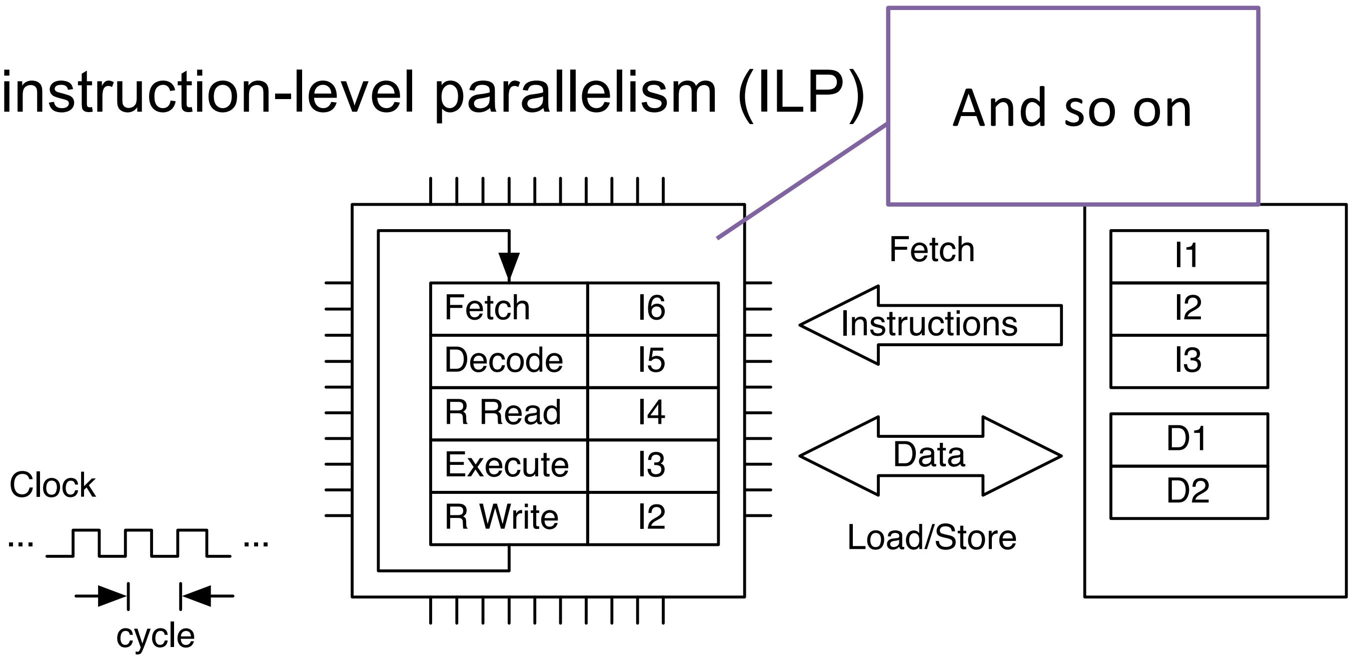
Processor Core Instruction Handling

- By pipelining, multiple instructions can be executed at each clock cycle
- Form of instruction-level parallelism



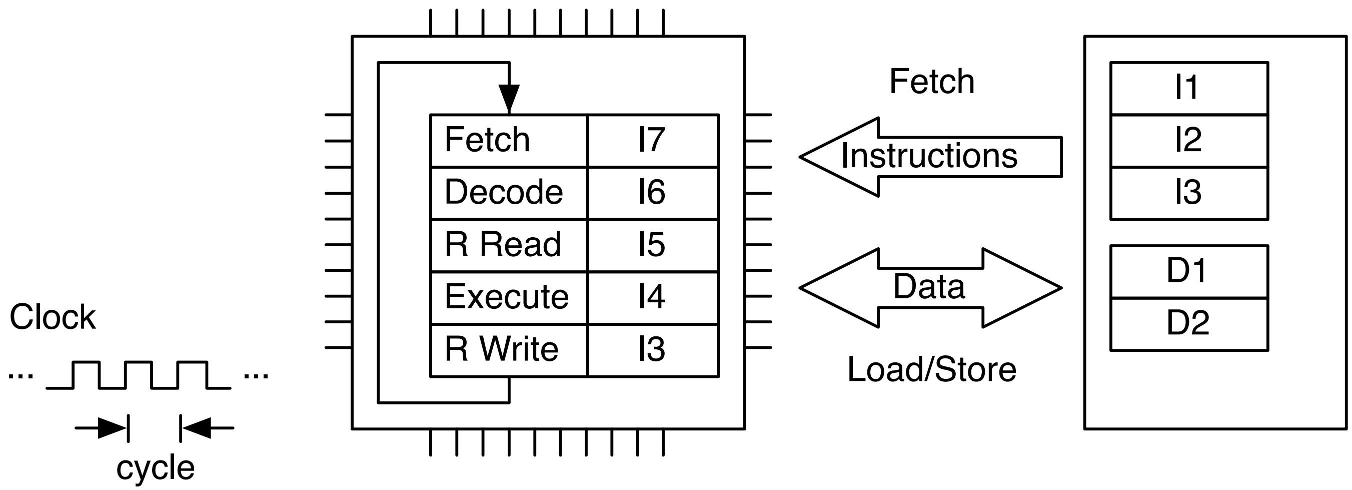
Processor Core Instruction Handling

- By pipelining, multiple instructions can be executed at each clock cycle
- Form of instruction-level parallelism (ILP)



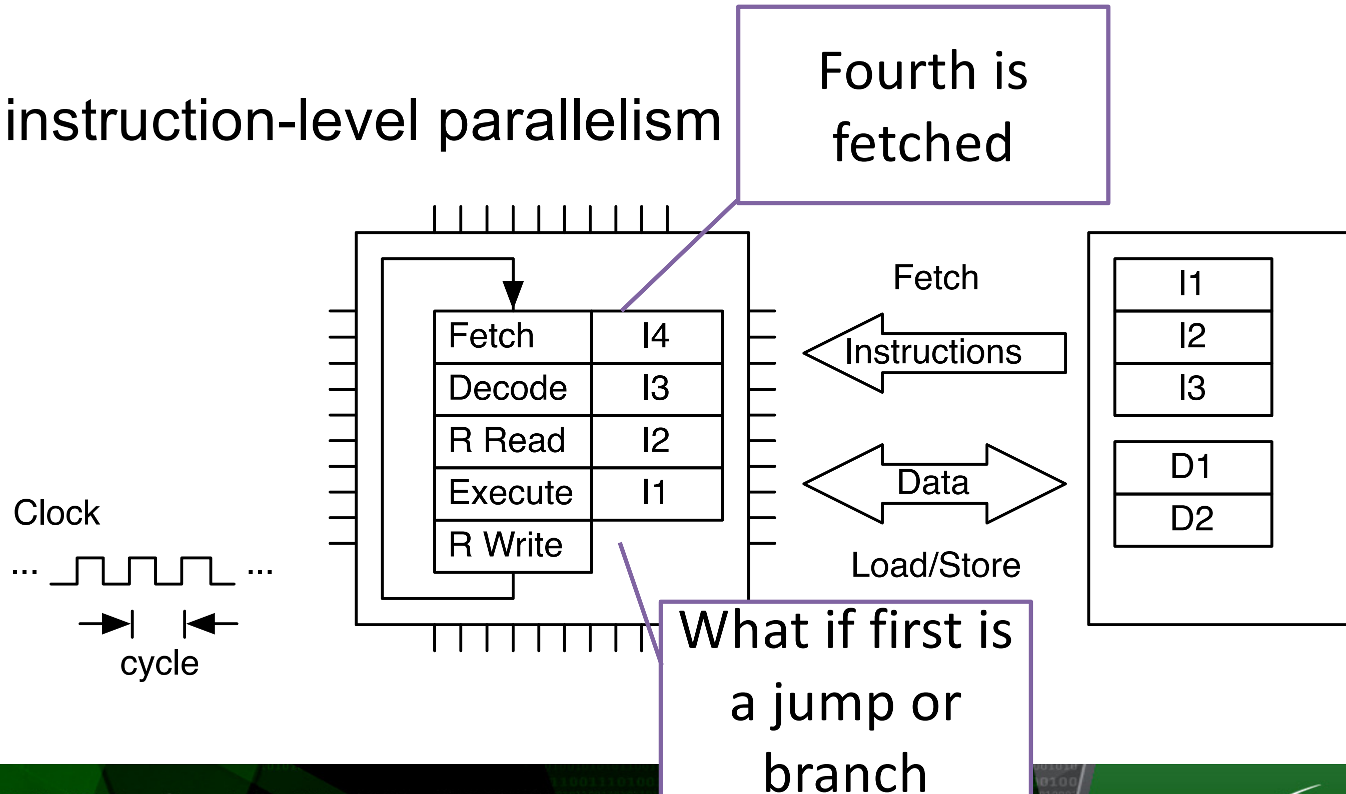
Processor Core Instruction Handling

- By pipelining, multiple instructions can be executed at each clock cycle
- Form of instruction-level parallelism (ILP)



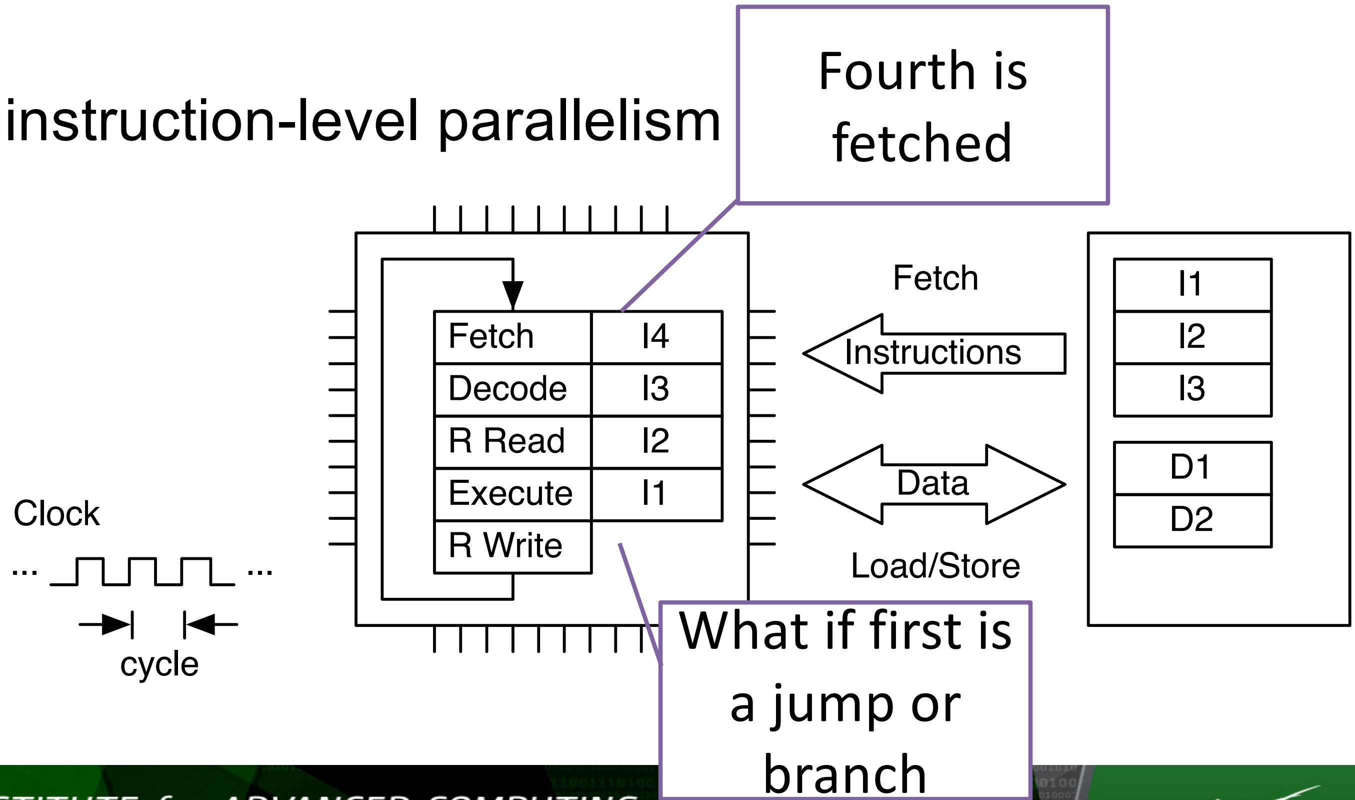
Pipeline Stall

- By pipelining, multiple instructions can be executed at each clock cycle
- Form of instruction-level parallelism



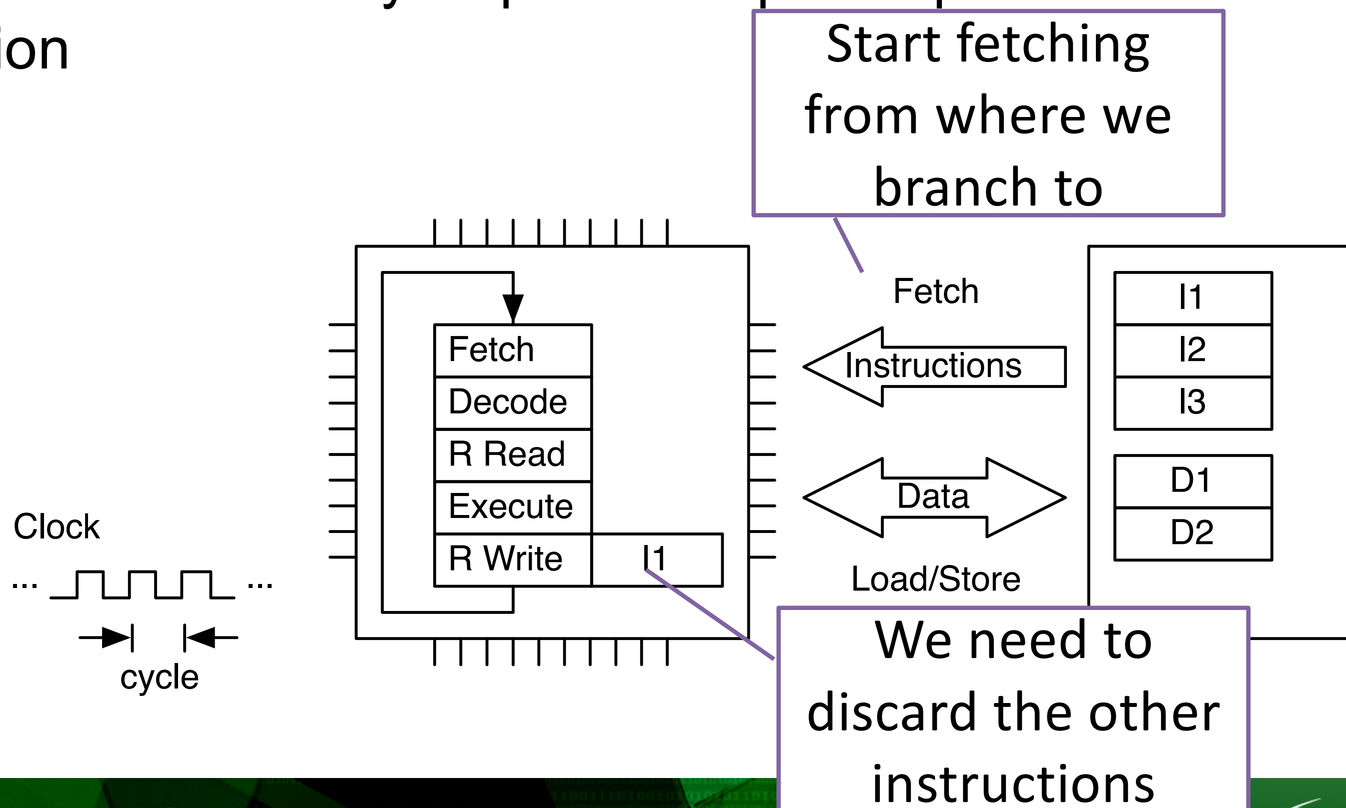
Pipeline Stall

- By pipelining, multiple instructions can be executed at each clock cycle
- Form of instruction-level parallelism



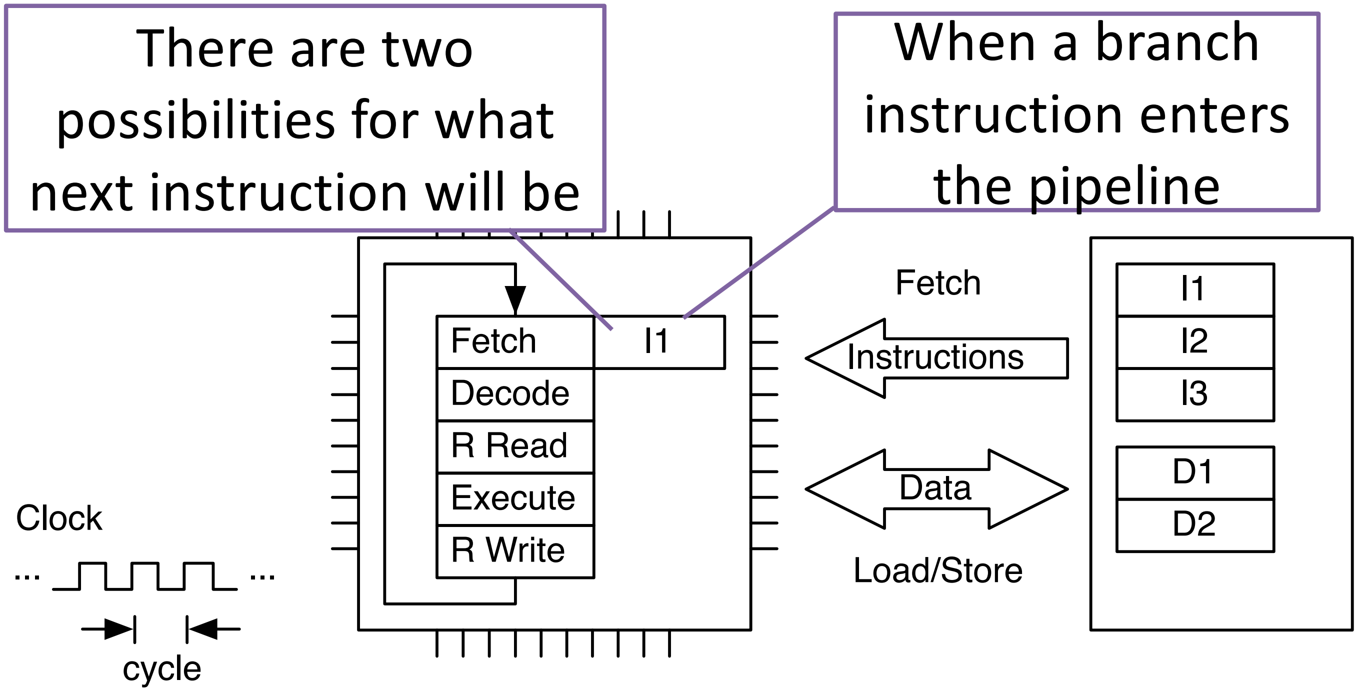
Pipeline Stall

- A single instruction may require multiple steps from fetch to completion



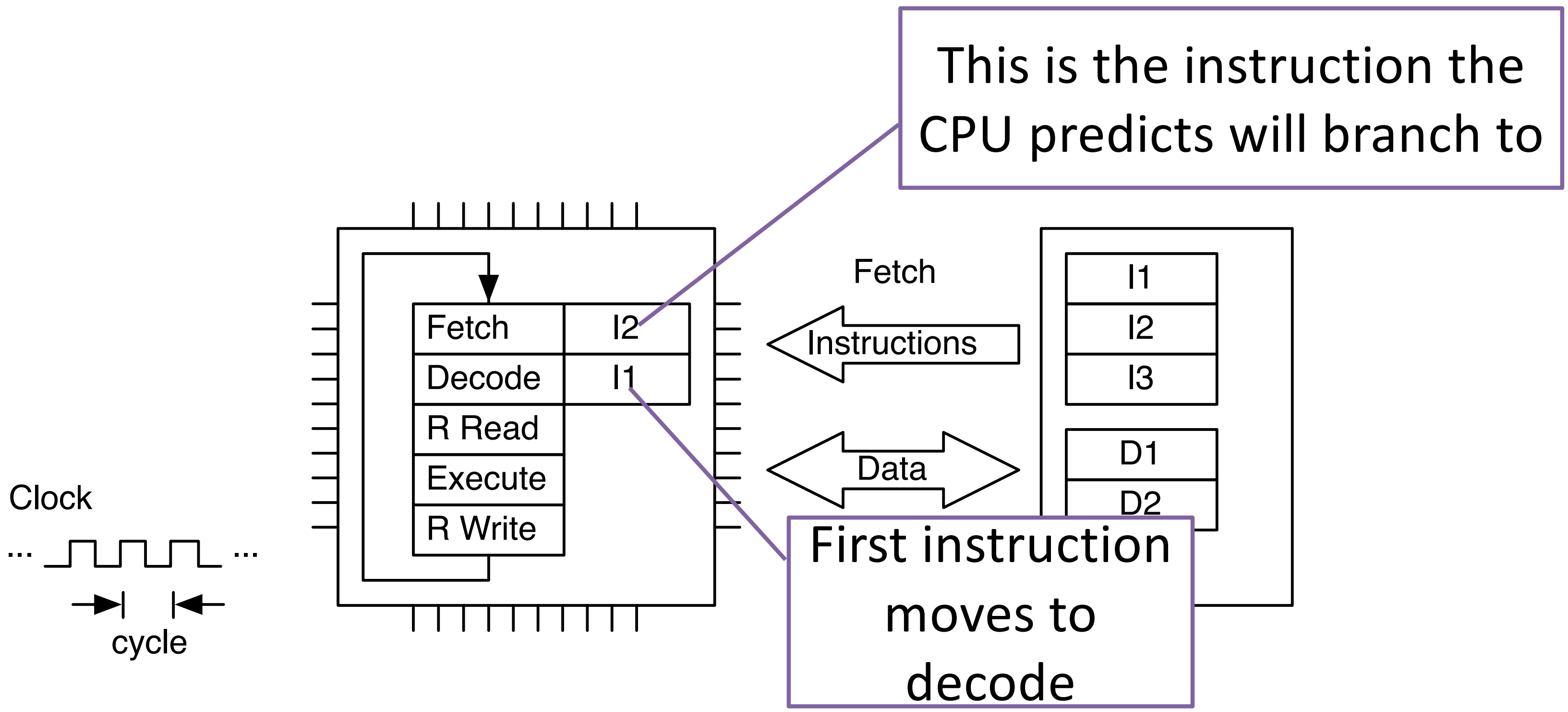
Branch Prediction

- Load the instructions we think will be branched to



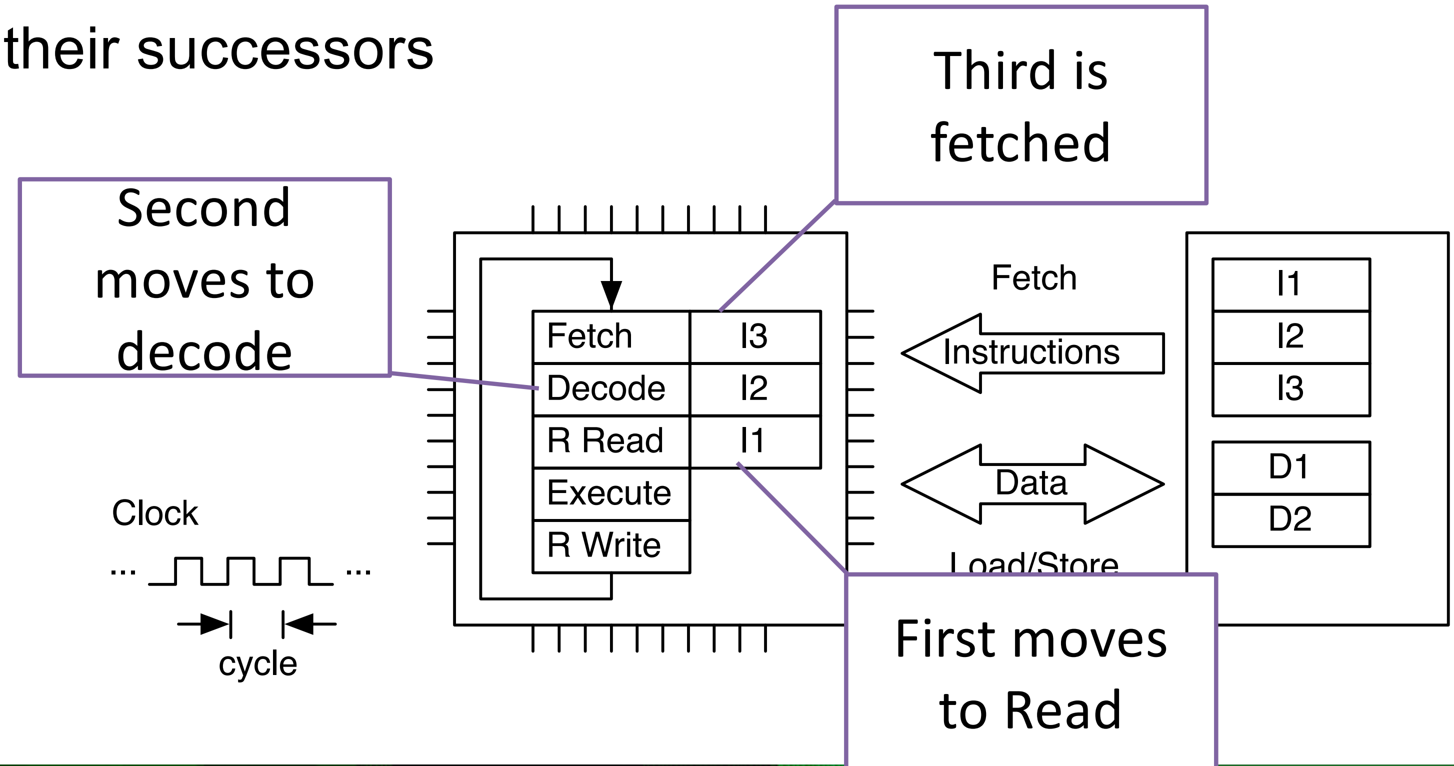
Branch Prediction

- Load the instructions we think will be branched to



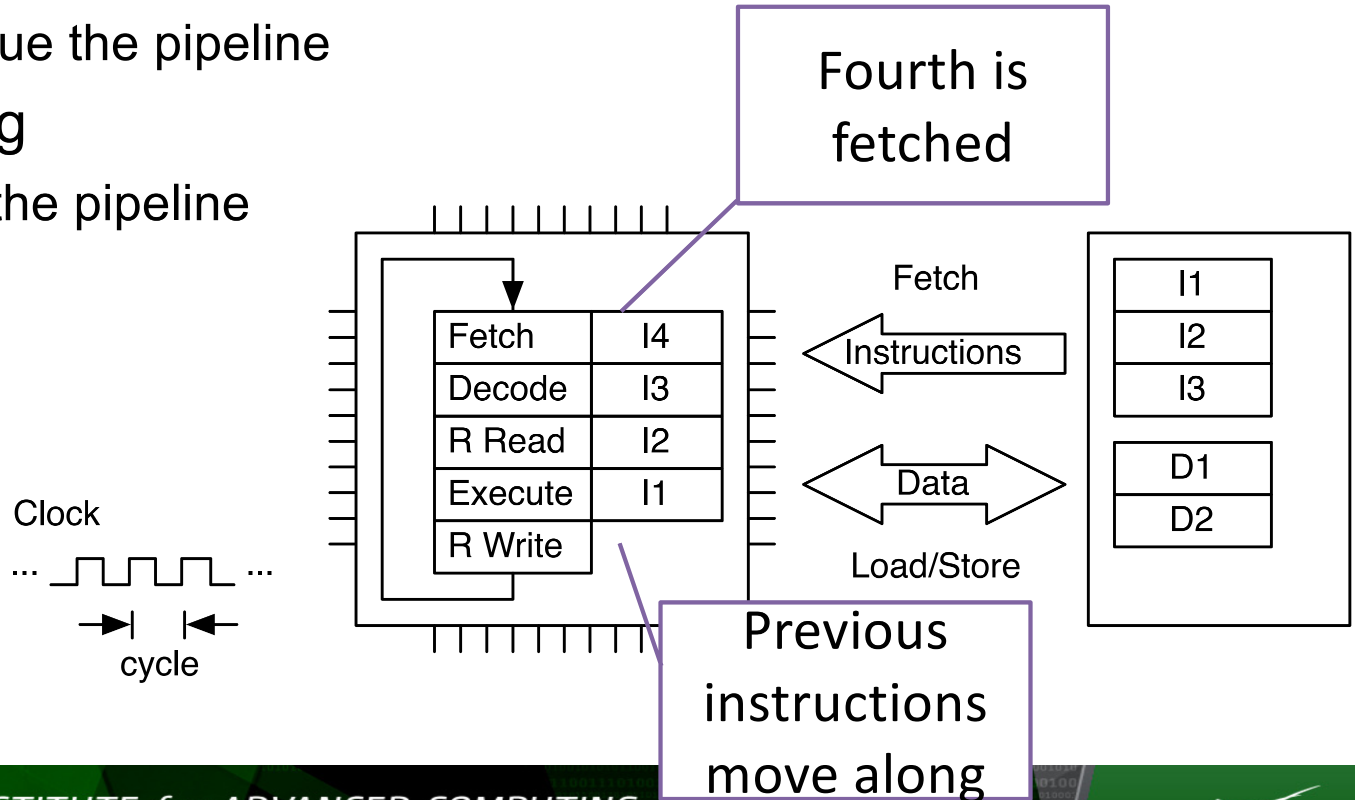
Branch Prediction

- Load the instructions we think will be branched to
- And their successors



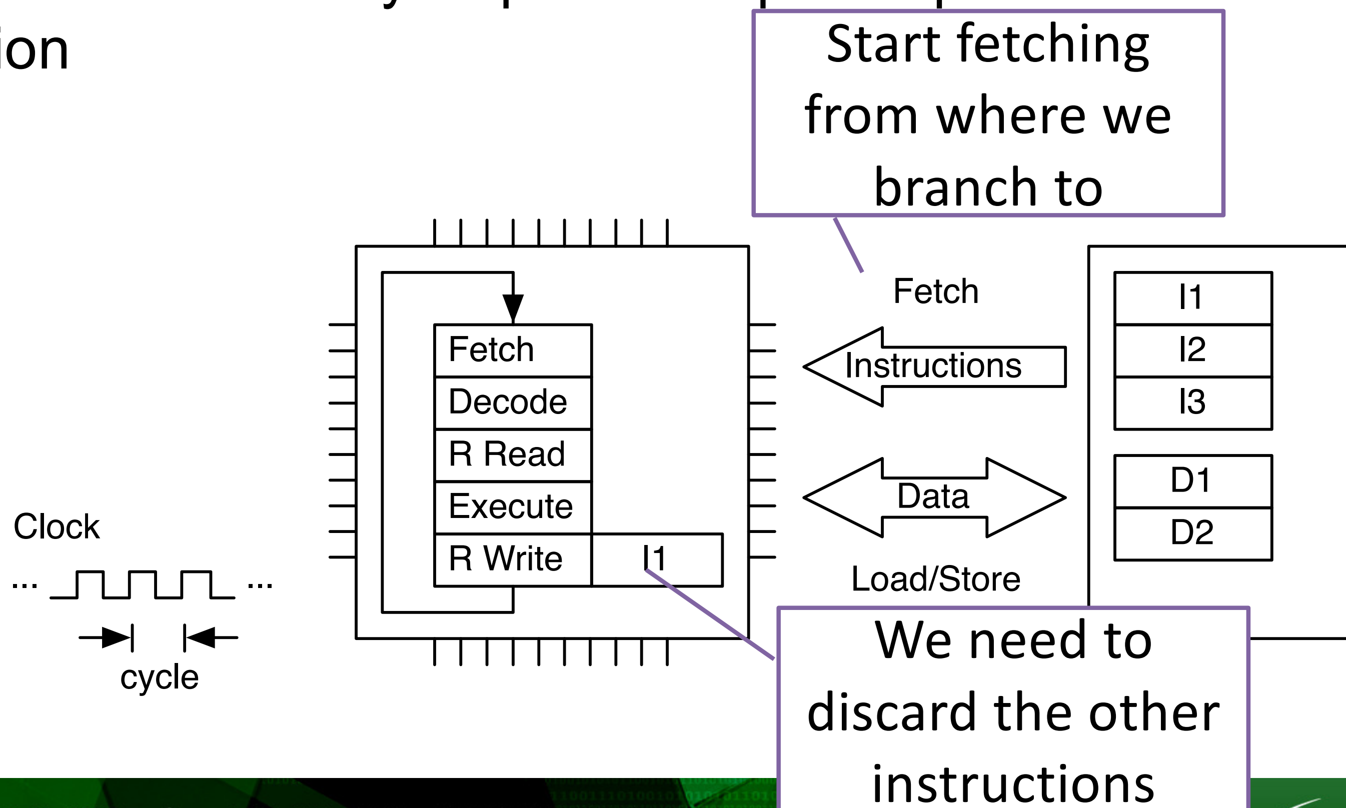
Instruction Pipelining

- When instruction is executed we were either right
  - Continue the pipeline
- Or wrong
  - Flush the pipeline



Pipeline Stall from Mis-Predict

- A single instruction may require multiple steps from fetch to completion

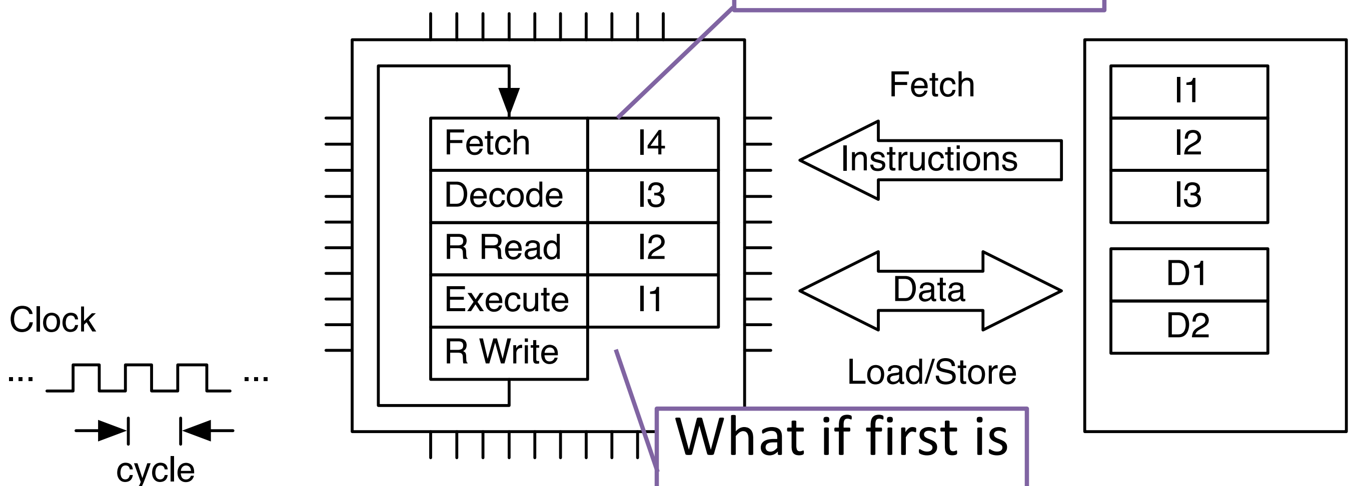


Performance-Oriented Architecture Features

- Execution Pipeline
  - Stages of functionality to process issued instructions
  - Hazards are conflicts with continued execution
  - Forwarding supports closely associated operations exhibiting precedence constraints
- Out of Order Execution
  - Uses reservation stations
  - Hides some core latencies and provide fine grain asynchronous operation supporting concurrency
- Branch Prediction
  - Permits computation to proceed at a conditional branch point prior to resolving predicate value
  - Overlaps follow-on computation with predicate resolution
  - Requires roll-back or equivalent to correct false guesses
  - Sometimes follows both paths, and several deep

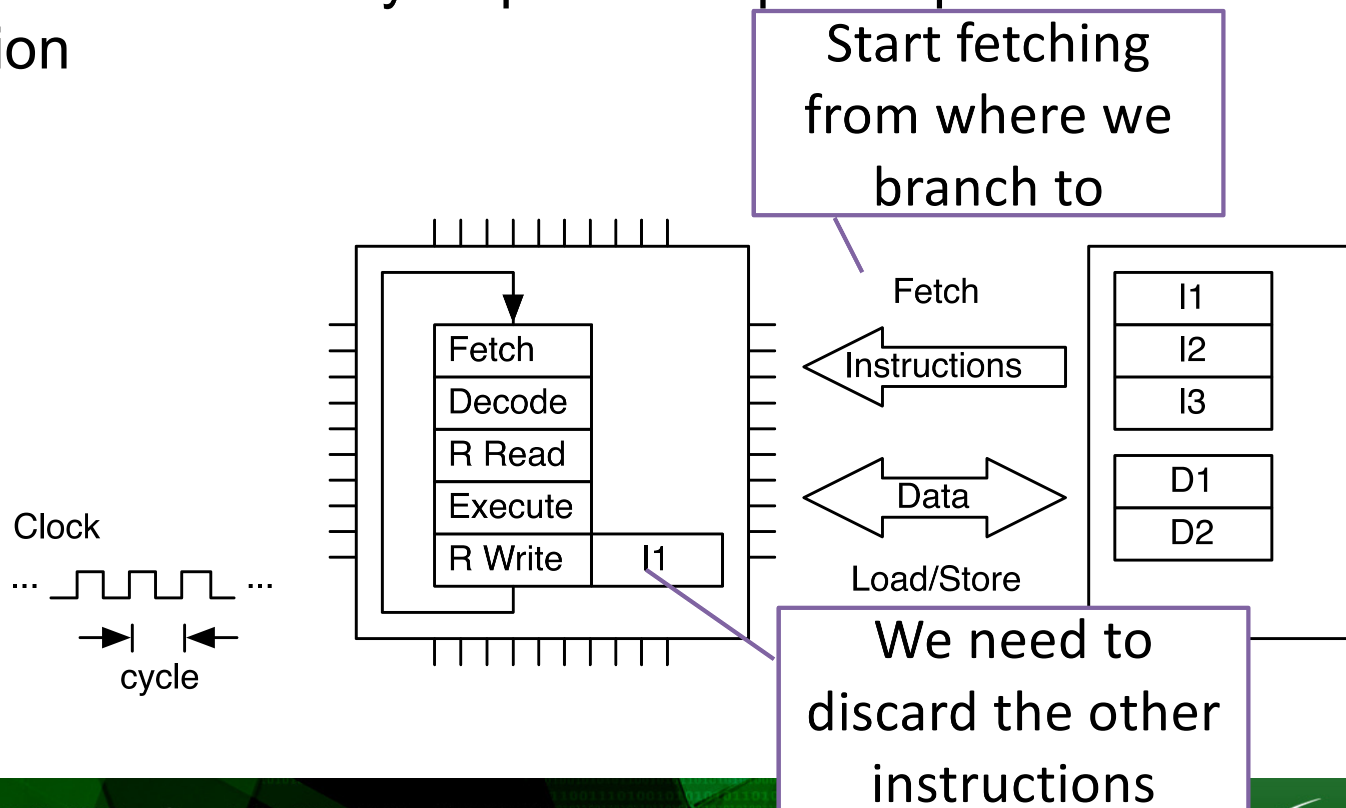
Pipeline Stall

- By pipelining, multiple instructions can be executed at each clock cycle
- Form of instruction-level parallelism



Pipeline Stall

- A single instruction may require multiple steps from fetch to completion



Compiling functions

```cpp
#include <iostream>
#include <cmath>

double sqrt583(double z) {
  double x = 1.0;

  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-z) / (2.0*x) ;
    x += dx;
    if (abs(dx) < 1.e-9) break;
  }

  return x;
}


int main () {

  std::cout << sqrt583(2.0) << std::endl;


  return 0;
}
```

```
$ c++ main.cpp
$ ./a.out
1.4142
```

Compile main.cpp

```
$ c++ main.cpp
```

Translate it into a language the cpu can run

```
$ ./a.out
```

The executable (program that the cpu can run)

But what is this really?

Compiled language

```cpp
#include <iostream>
#include <cmath>

double sqrt583(double z) {
  double x = 1.0;

  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-z) / (2.0*x) ;
    x += dx;
    if (abs(dx) < 1.e-9) break;
  }

  return x;
}


int main () {

  std::cout << sqrt583(2.0) << std::endl;


  return 0;
}
```

```
$ c++ main.cpp
```

main

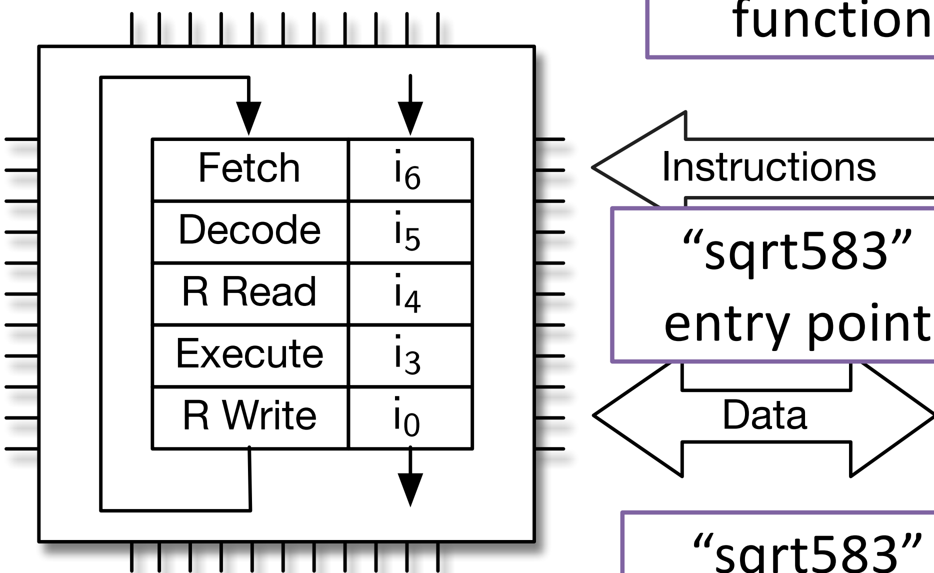
```
subq      $64, %rsp
movsd     LCPI1_0(%rip), %xmm0
movl      $0, -36(%rbp)
movsd     %xmm0, -48(%rbp)
movsd     -48(%rbp), %xmm0
callq     __Z7sqrt583d
movq      %rax, -24(%rbp)
movq      %rdi, -32(%rbp)
movq      -24(%rbp), %rdi
subq      *-32(%rbp)
. . .
```

sqrt583

```
movsd     LCPI0_0(%rip), %xmm1
movsd     %xmm0, -16(%rbp)
movsd     %xmm1, -24(%rbp)
movq      $0, -32(%rbp)
cmpq      $32, -32(%rbp)
jae       LBB0_6
movsd     LCPI0_1(%rip), %xmm0
movsd     LCPI0_3(%rip), %xmm1
movabsq   $-9223372036854, %rax
movsd     -24(%rbp), %xmm2
. . .
```

Fetch Decode Execute

CPU instructions are stored in memory



"main" entry point

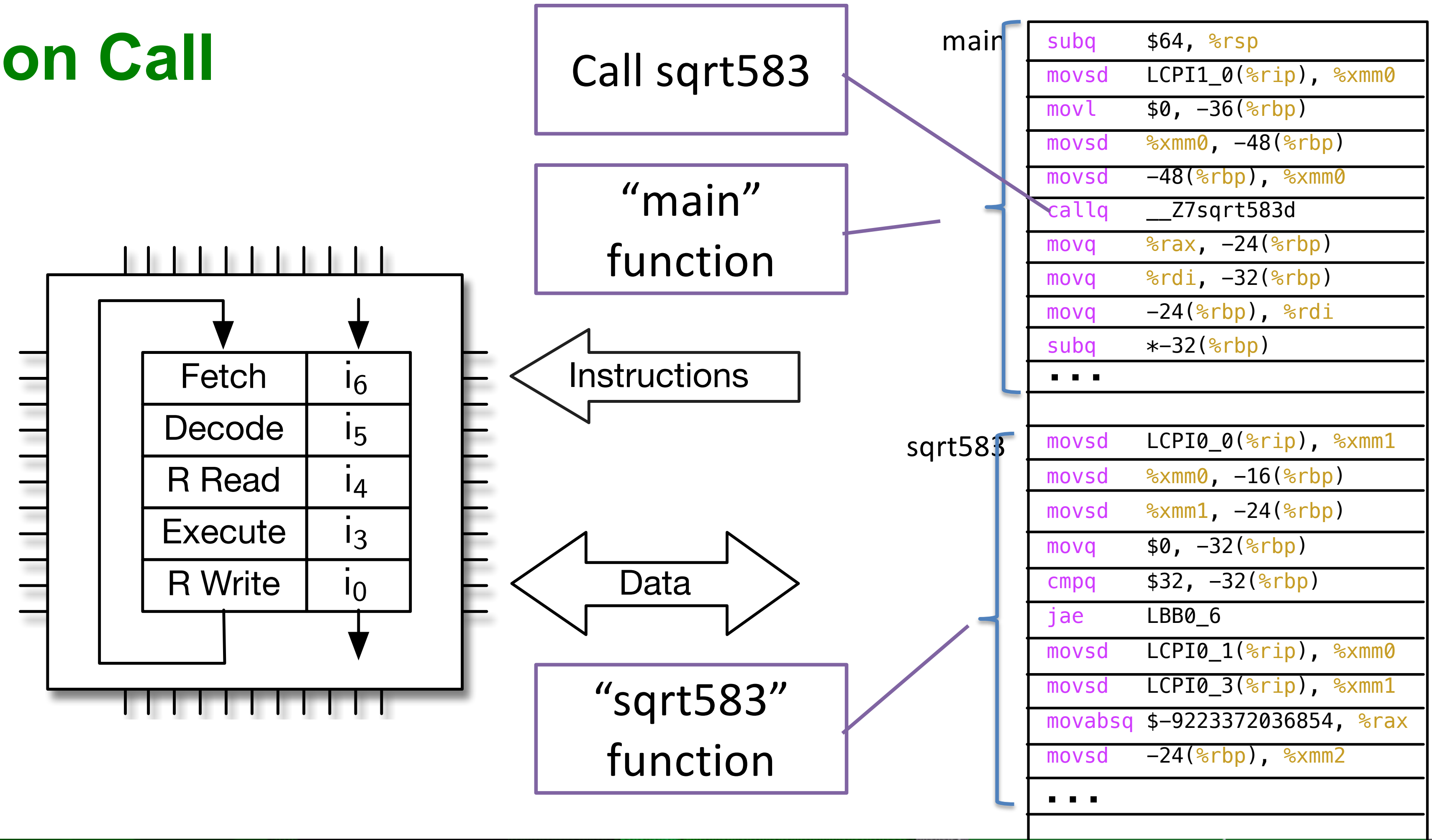
"main" function

"sqrt583" entry point

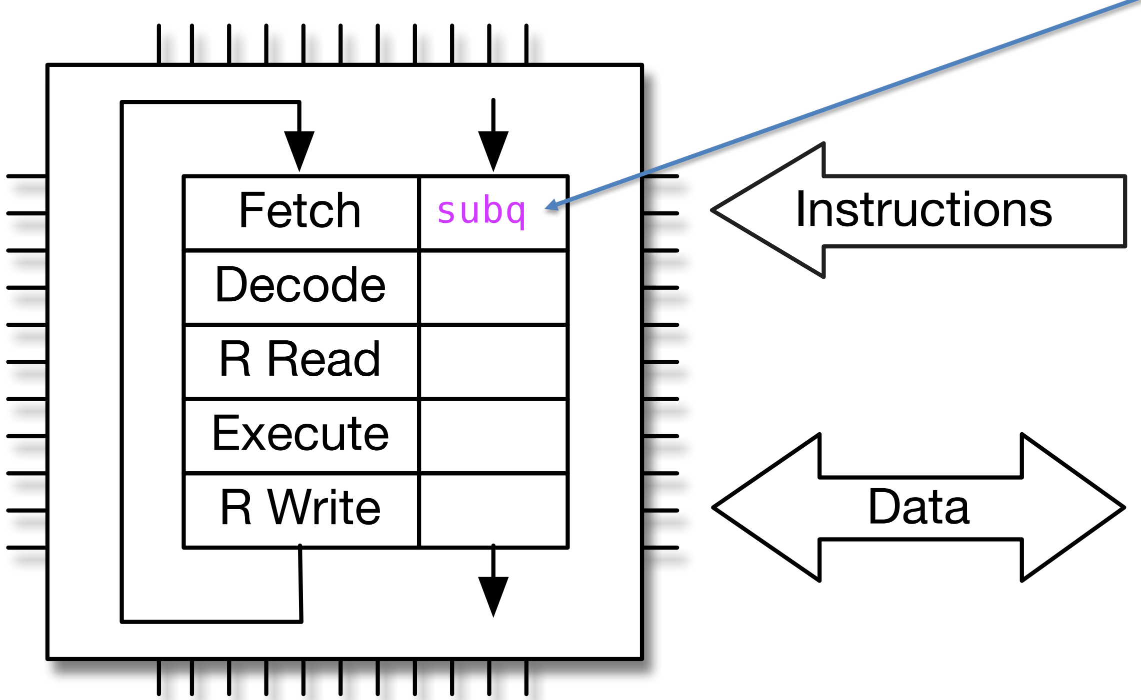
"sqrt583" function

```
main:
subq      $64, %rsp
movsd     LCPI1_0(%rip), %xmm0
movl      $0, -36(%rbp)
movsd     %xmm0, -48(%rbp)
movsd     -48(%rbp), %xmm0
callq     __Z7sqrt583d
movq      %rax, -24(%rbp)
movq      %rdi, -32(%rbp)
movq      -24(%rbp), %rdi
subq      *-32(%rbp)
...

sqrt583:
movsd     LCPI0_0(%rip), %xmm1
movsd     %xmm0, -16(%rbp)
movsd     %xmm1, -24(%rbp)
movq      $0, -32(%rbp)
cmpq      $32, -32(%rbp)
jae       LBB0_6
movsd     LCPI0_1(%rip), %xmm0
movsd     LCPI0_3(%rip), %xmm1
movabsq   $-9223372036854, %rax
movsd     -24(%rbp), %xmm2
...
```

Function Call



Function Call



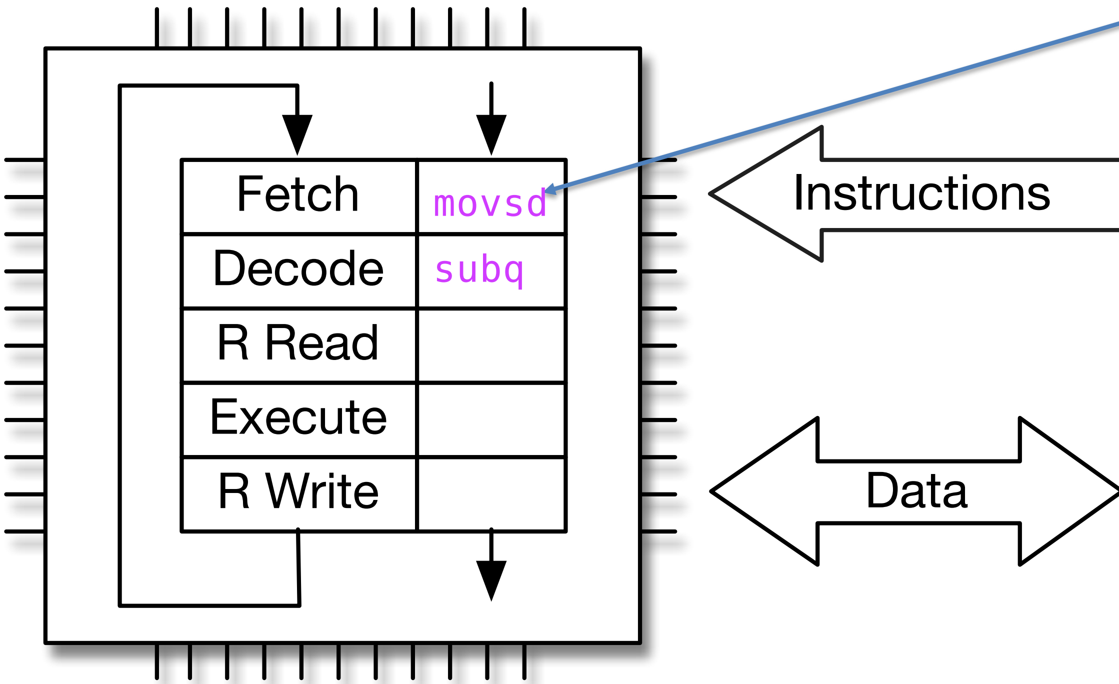
main

```
subq      $64, %rsp
movsd     LCPI1_0(%rip), %xmm0
movl      $0, -36(%rbp)
movsd     %xmm0, -48(%rbp)
movsd     -48(%rbp), %xmm0
callq     __Z7sqrt583d
movq      %rax, -24(%rbp)
movq      %rdi, -32(%rbp)
movq      -24(%rbp), %rdi
subq      *-32(%rbp)
. . .
```

sqrt583

```
movsd     LCPI0_0(%rip), %xmm1
movsd     %xmm0, -16(%rbp)
movsd     %xmm1, -24(%rbp)
movq      $0, -32(%rbp)
cmpq      $32, -32(%rbp)
jae       LBB0_6
movsd     LCPI0_1(%rip), %xmm0
movsd     LCPI0_3(%rip), %xmm1
movabsq   $-9223372036854, %rax
movsd     -24(%rbp), %xmm2
. . .
```

Function Call



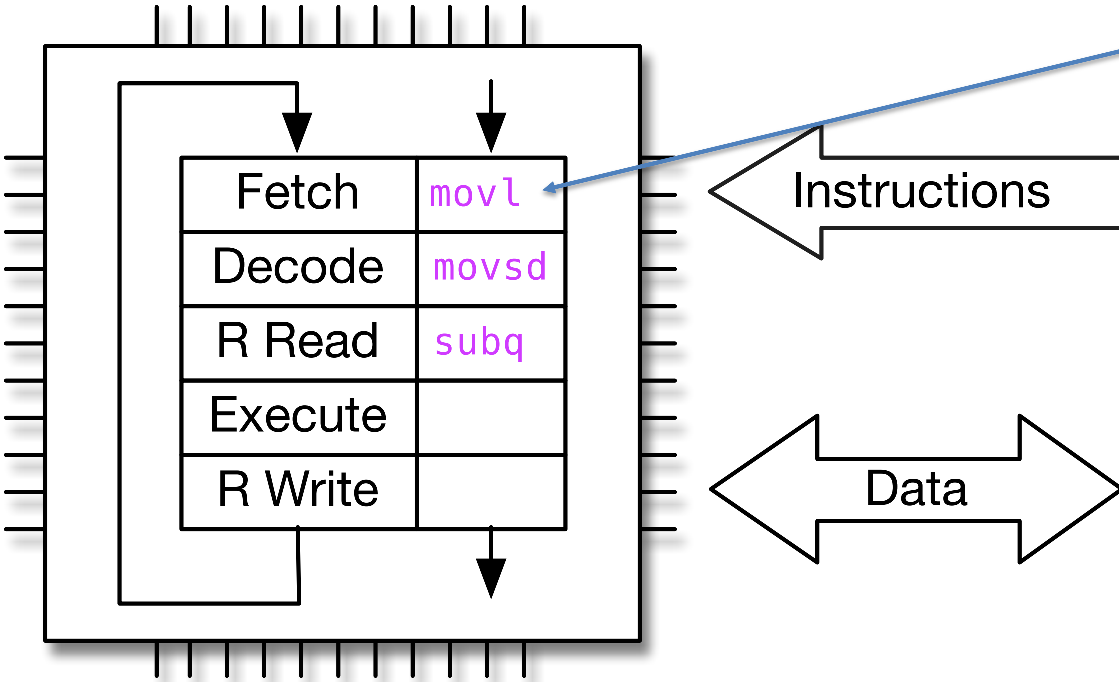
main

```
subq      $64, %rsp
movsd     LCPI1_0(%rip), %xmm0
movl      $0, -36(%rbp)
movsd     %xmm0, -48(%rbp)
movsd     -48(%rbp), %xmm0
callq     __Z7sqrt583d
movq      %rax, -24(%rbp)
movq      %rdi, -32(%rbp)
movq      -24(%rbp), %rdi
subq      *-32(%rbp)
. . .
```

sqrt583

```
movsd     LCPI0_0(%rip), %xmm1
movsd     %xmm0, -16(%rbp)
movsd     %xmm1, -24(%rbp)
movq      $0, -32(%rbp)
cmpq      $32, -32(%rbp)
jae       LBB0_6
movsd     LCPI0_1(%rip), %xmm0
movsd     LCPI0_3(%rip), %xmm1
movabsq   $-9223372036854, %rax
movsd     -24(%rbp), %xmm2
. . .
```

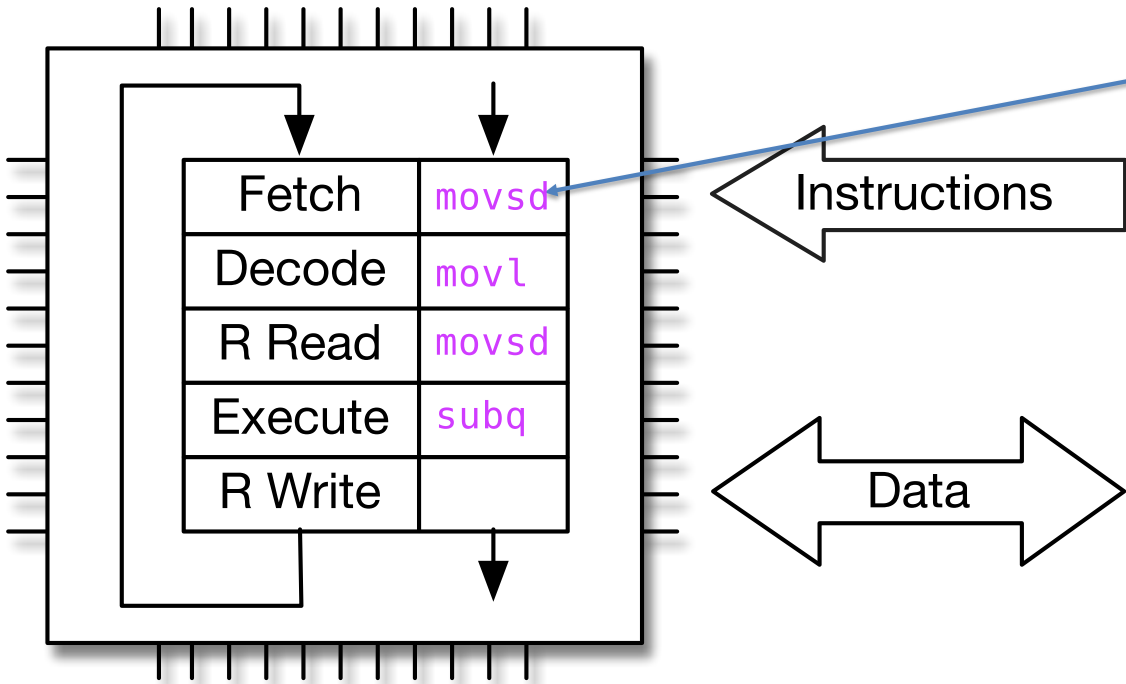
Function Call



main
```
subq      $64, %rsp
movsd     LCPI1_0(%rip), %xmm0
movl      $0, -36(%rbp)
movsd     %xmm0, -48(%rbp)
movsd     -48(%rbp), %xmm0
callq     __Z7sqrt583d
movq      %rax, -24(%rbp)
movq      %rdi, -32(%rbp)
movq      -24(%rbp), %rdi
subq      *-32(%rbp)
...
```

sqrt583
```
movsd     LCPI0_0(%rip), %xmm1
movsd     %xmm0, -16(%rbp)
movsd     %xmm1, -24(%rbp)
movq      $0, -32(%rbp)
cmpq      $32, -32(%rbp)
jae       LBB0_6
movsd     LCPI0_1(%rip), %xmm0
movsd     LCPI0_3(%rip), %xmm1
movabsq   $-9223372036854, %rax
movsd     -24(%rbp), %xmm2
...
```

Function Call



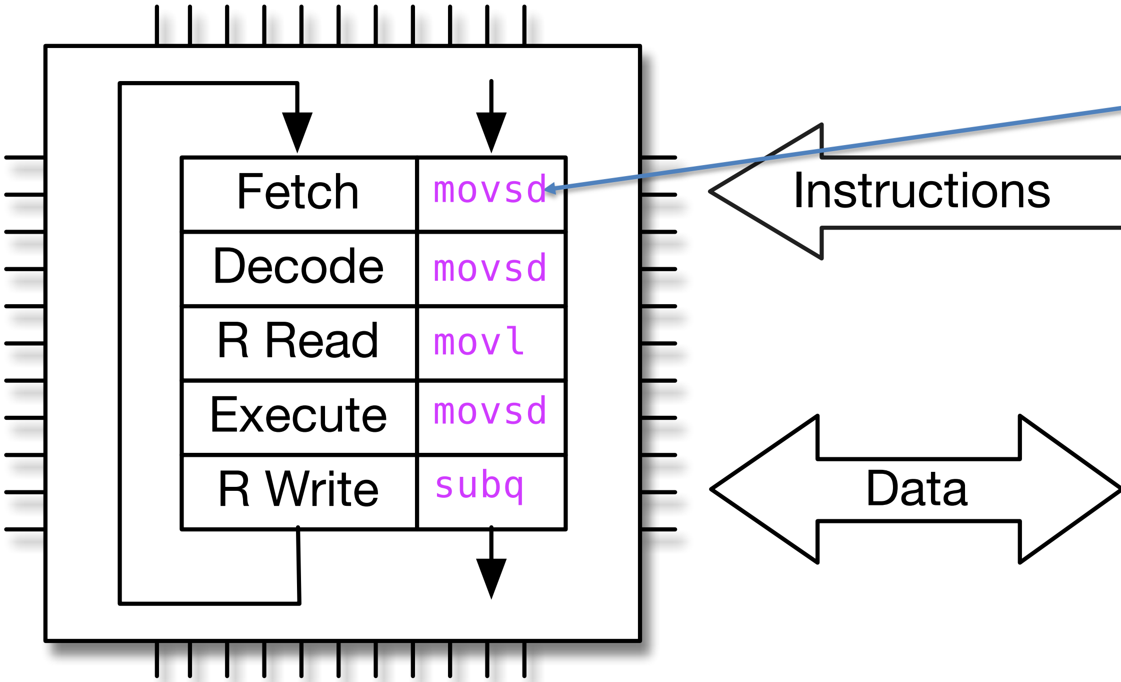
main

```
subq     $64, %rsp
movsd    LCPI1_0(%rip), %xmm0
movl     $0, -36(%rbp)
movsd    %xmm0, -48(%rbp)
movsd    -48(%rbp), %xmm0
callq    __Z7sqrt583d
movq     %rax, -24(%rbp)
movq     %rdi, -32(%rbp)
movq     -24(%rbp), %rdi
subq     *-32(%rbp)
...
```

sqrt583

```
movsd    LCPI0_0(%rip), %xmm1
movsd    %xmm0, -16(%rbp)
movsd    %xmm1, -24(%rbp)
movq     $0, -32(%rbp)
cmpq     $32, -32(%rbp)
jae      LBB0_6
movsd    LCPI0_1(%rip), %xmm0
movsd    LCPI0_3(%rip), %xmm1
movabsq  $-9223372036854, %rax
movsd    -24(%rbp), %xmm2
...
```

Function Call



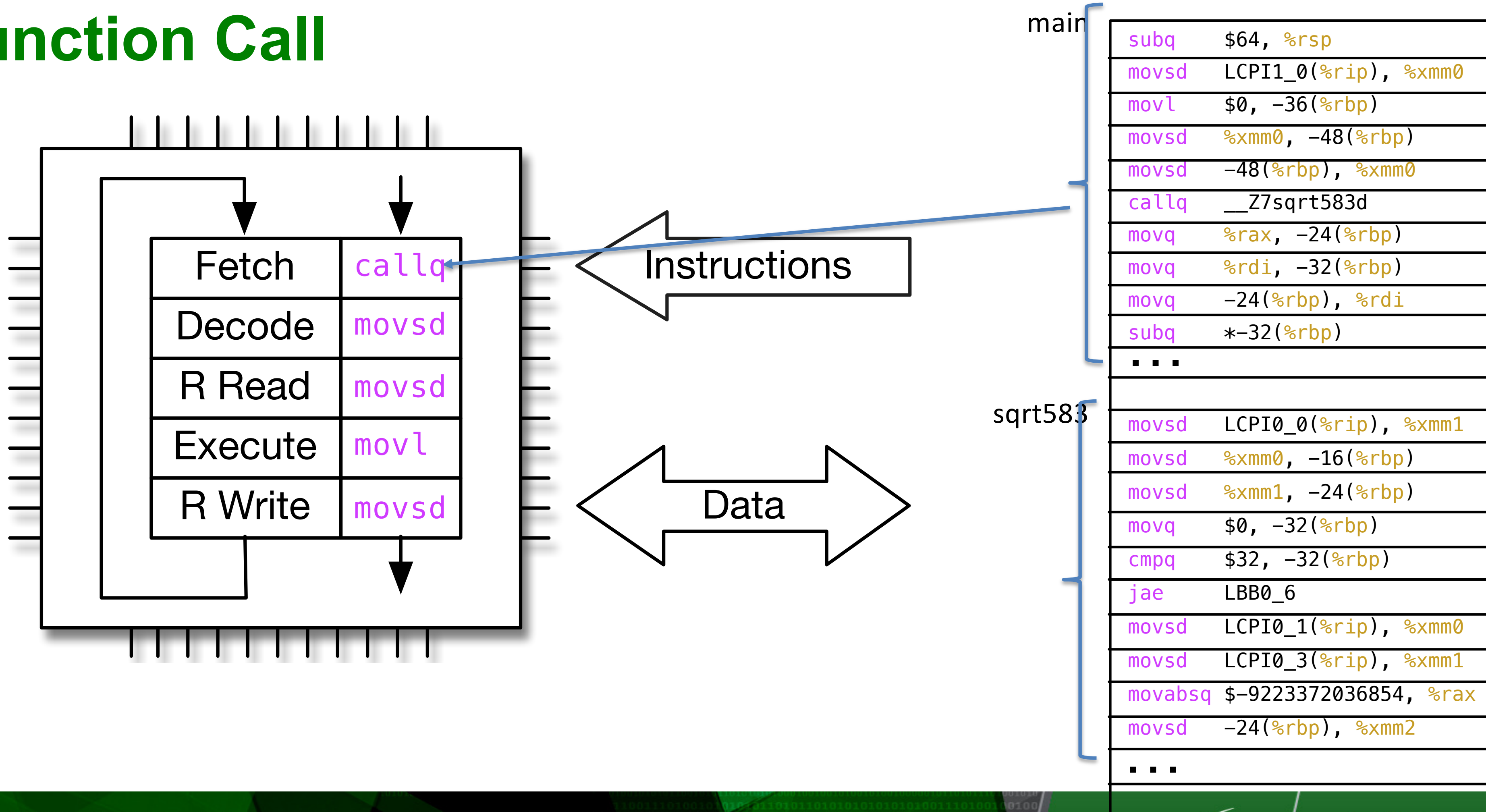
main

```
subq      $64, %rsp
movsd     LCPI1_0(%rip), %xmm0
movl      $0, -36(%rbp)
movsd     %xmm0, -48(%rbp)
movsd     -48(%rbp), %xmm0
callq     __Z7sqrt583d
movq      %rax, -24(%rbp)
movq      %rdi, -32(%rbp)
movq      -24(%rbp), %rdi
subq      *-32(%rbp)
...
```

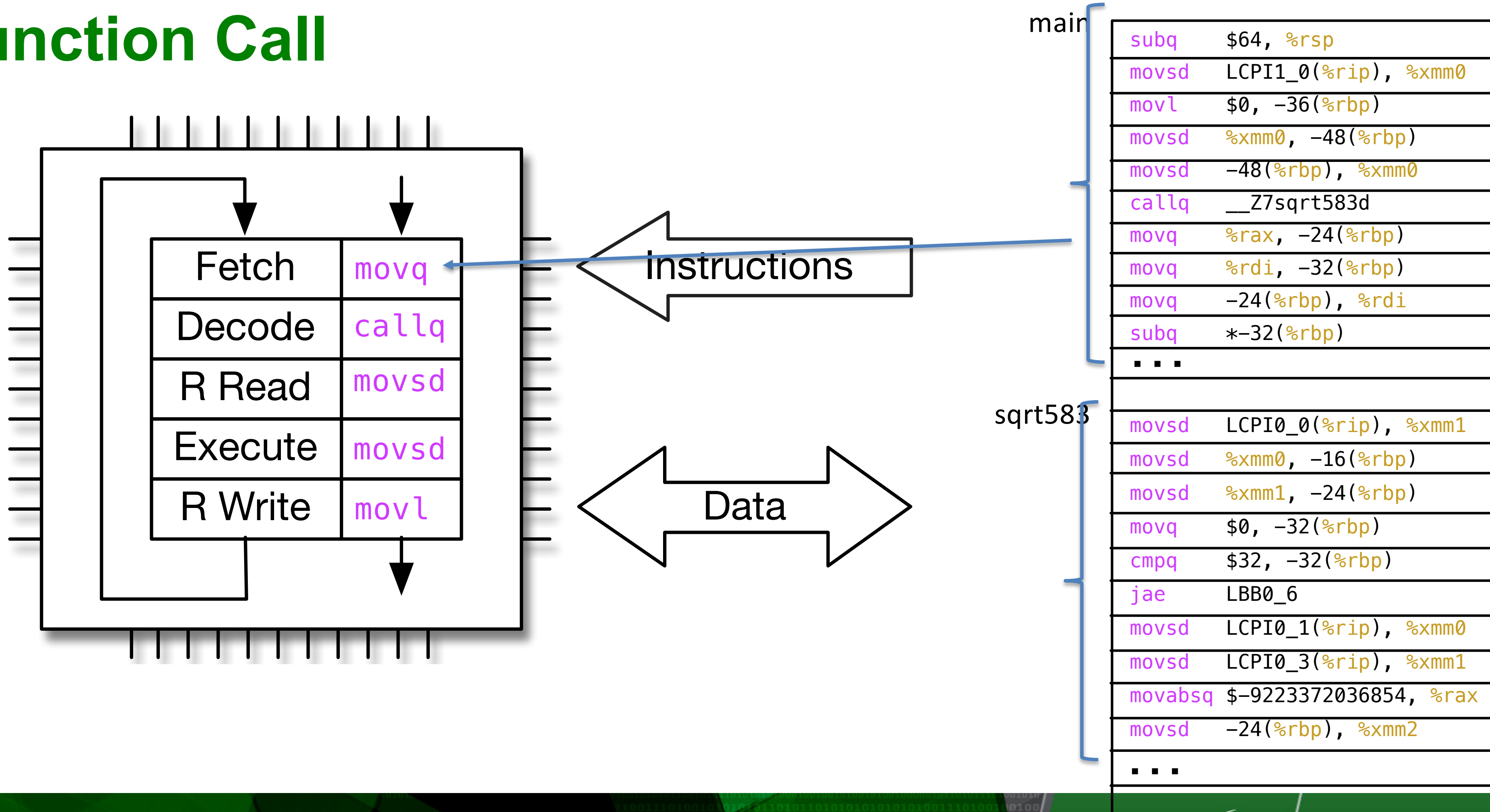
sqrt583

```
movsd     LCPI0_0(%rip), %xmm1
movsd     %xmm0, -16(%rbp)
movsd     %xmm1, -24(%rbp)
movq      $0, -32(%rbp)
cmpq      $32, -32(%rbp)
jae       LBB0_6
movsd     LCPI0_1(%rip), %xmm0
movsd     LCPI0_3(%rip), %xmm1
movabsq   $-9223372036854, %rax
movsd     -24(%rbp), %xmm2
...
```

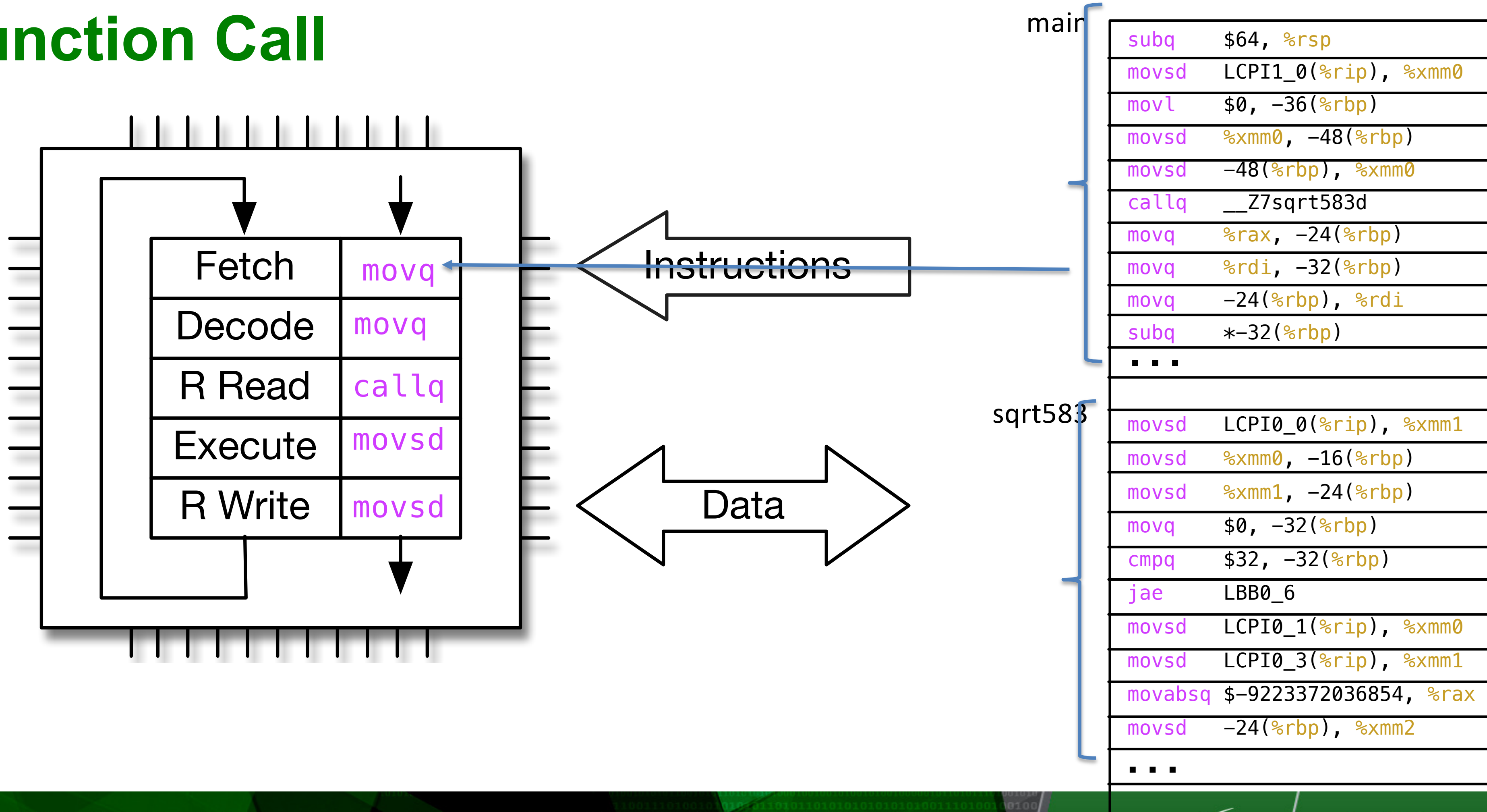
Function Call



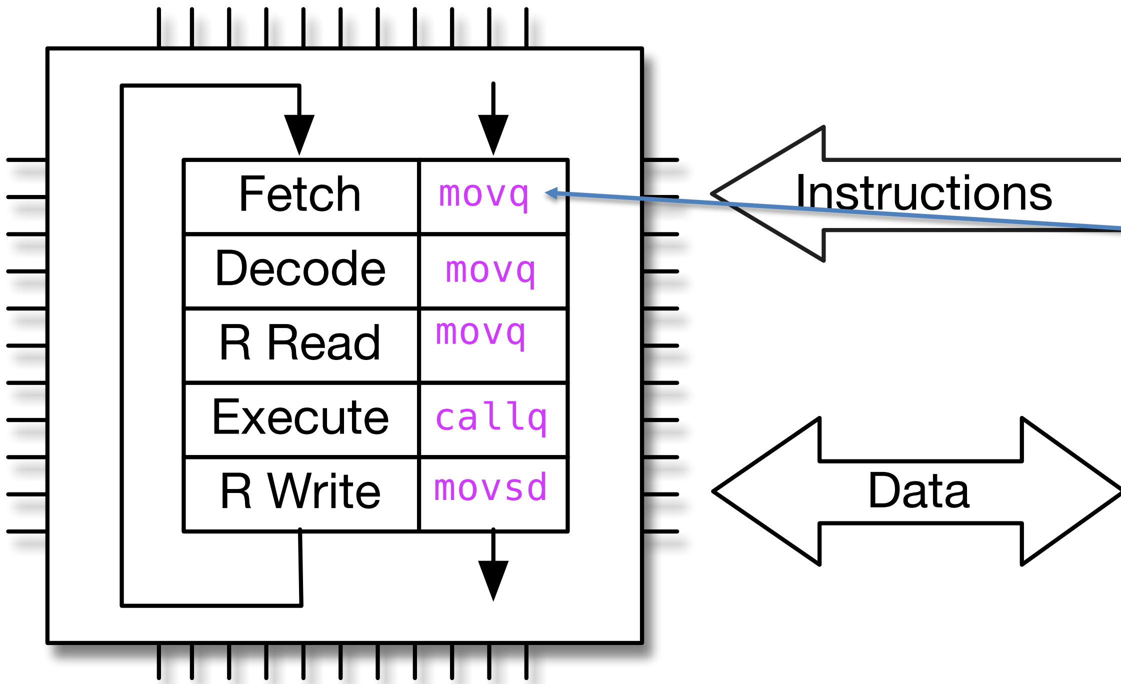
Function Call



Function Call



Function Call



main

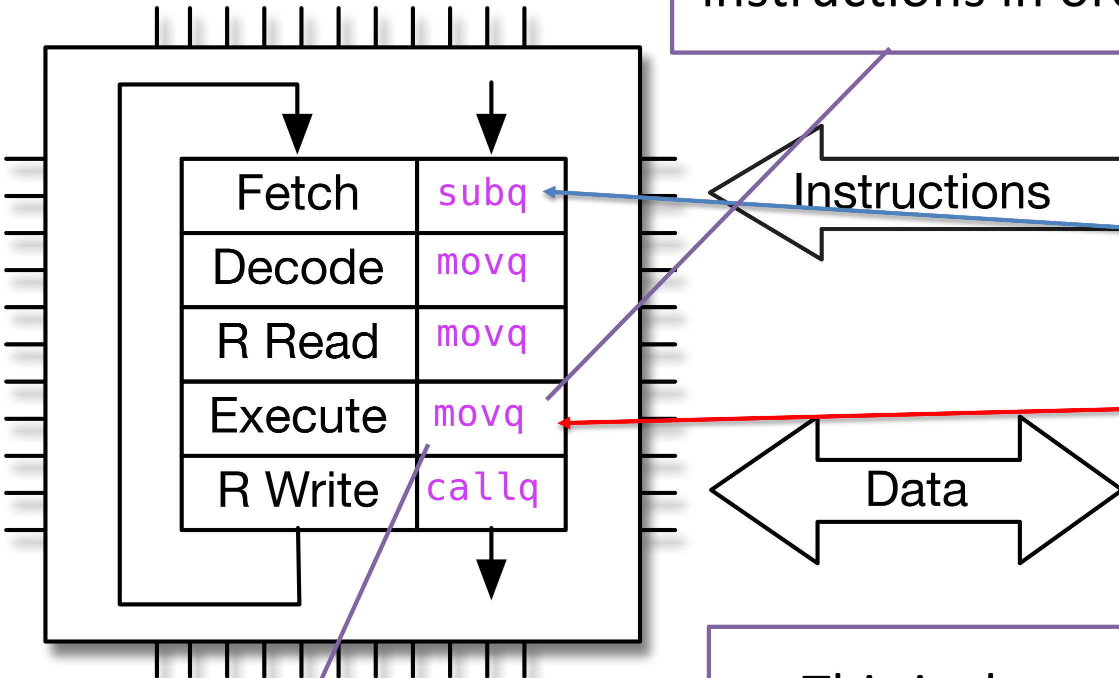
```
subq     $64, %rsp
movsd    LCPI1_0(%rip), %xmm0
movl     $0, -36(%rbp)
movsd    %xmm0, -48(%rbp)
movsd    -48(%rbp), %xmm0
callq    __Z7sqrt583d
movq     %rax, -24(%rbp)
movq     %rdi, -32(%rbp)
movq     -24(%rbp), %rdi
subq     *-32(%rbp)
...
```

sqrt583

```
movsd    LCPI0_0(%rip), %xmm1
movsd    %xmm0, -16(%rbp)
movsd    %xmm1, -24(%rbp)
movq     $0, -32(%rbp)
cmpq     $32, -32(%rbp)
jae      LBB0_6
movsd    LCPI0_1(%rip), %xmm0
movsd    LCPI0_3(%rip), %xmm1
movabsq  $-9223372036854, %rax
movsd    -24(%rbp), %xmm2
...
```

Function Call

But we just fetched instructions in order



main

```
subq      $64, %rsp
movsd     LCPI1_0(%rip), %xmm0
movl      $0, -36(%rbp)
movsd     %xmm0, -48(%rbp)
movsd     -48(%rbp), %xmm0
callq     __Z7sqrt583d
movq      %rax, -24(%rbp)
movq      %rdi, -32(%rbp)
movq      -24(%rbp), %rdi
subq      *-32(%rbp)
...
```

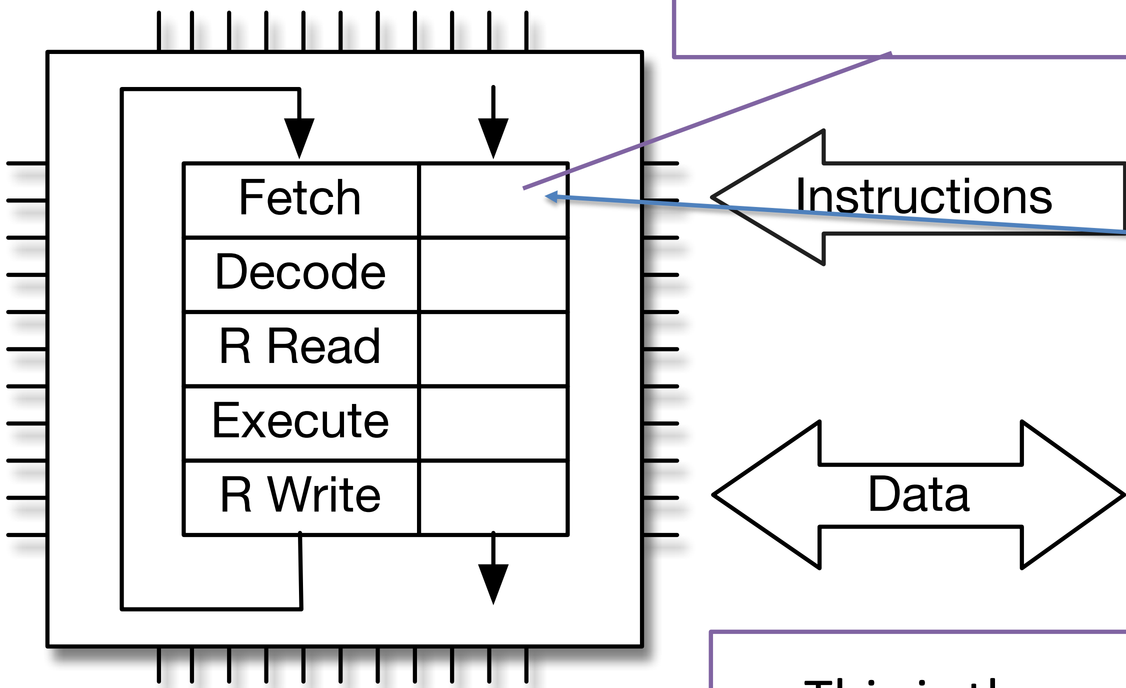
sqrt583

```
movsd     LCPI0_0(%rip), %xmm1
movsd     %xmm0, -16(%rbp)
movsd     %xmm1, -24(%rbp)
movq      $0, -32(%rbp)
cmpq      $32, -32(%rbp)
jae       LBB0_6
movsd     LCPI0_1(%rip), %xmm0
movsd     LCPI0_3(%rip), %xmm1
movabsq   $-9223372036854, %rax
movsd     -24(%rbp), %xmm2
...
```

This is the next instruction after callq

These are all wrong

Function Call



Flush the pipeline

This is the next instruction after callq

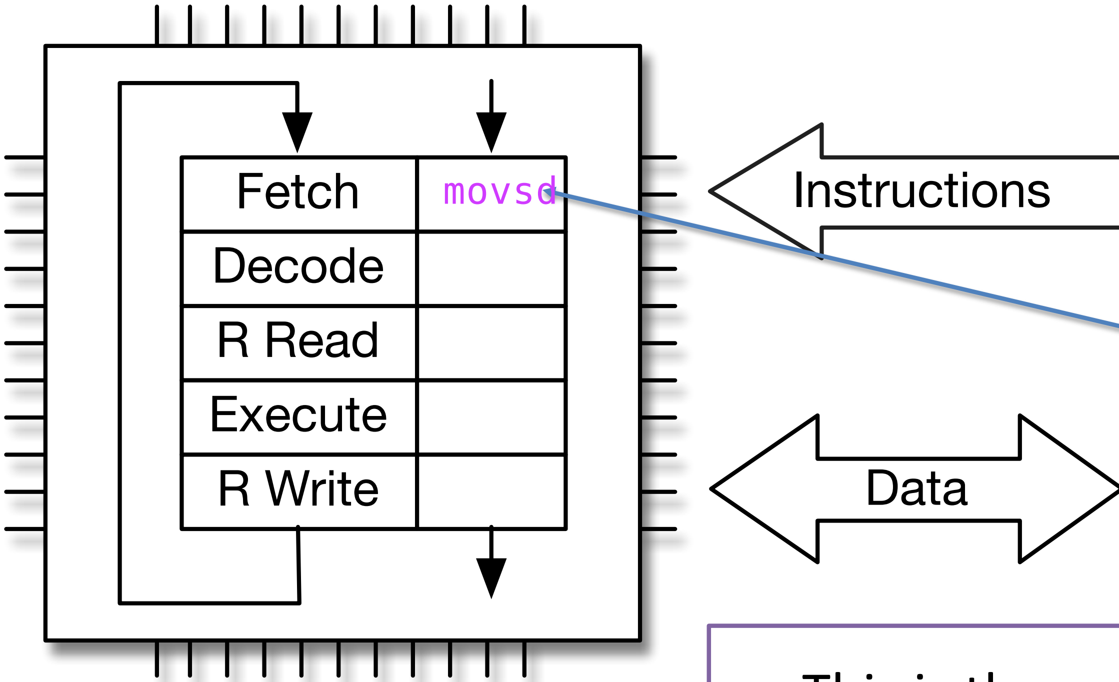
main

```
subq      $64, %rsp
movsd     LCPI1_0(%rip), %xmm0
movl      $0, -36(%rbp)
movsd     %xmm0, -48(%rbp)
movsd     -48(%rbp), %xmm0
callq     __Z7sqrt583d
movq      %rax, -24(%rbp)
movq      %rdi, -32(%rbp)
movq      -24(%rbp), %rdi
subq      *-32(%rbp)
...
```

sqrt583

```
movsd     LCPI0_0(%rip), %xmm1
movsd     %xmm0, -16(%rbp)
movsd     %xmm1, -24(%rbp)
movq      $0, -32(%rbp)
cmpq      $32, -32(%rbp)
jae       LBB0_6
movsd     LCPI0_1(%rip), %xmm0
movsd     LCPI0_3(%rip), %xmm1
movabsq   $-9223372036854, %rax
movsd     -24(%rbp), %xmm2
...
```

Function Call

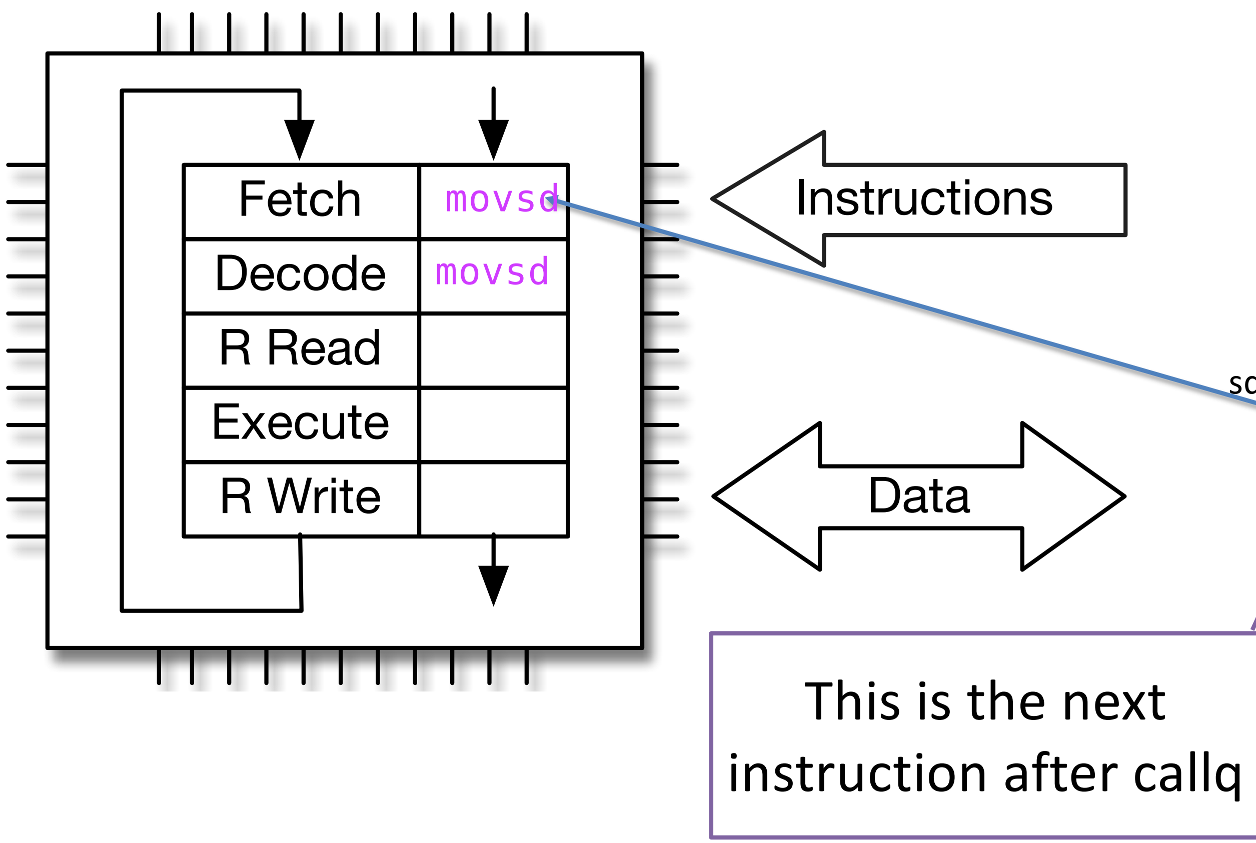


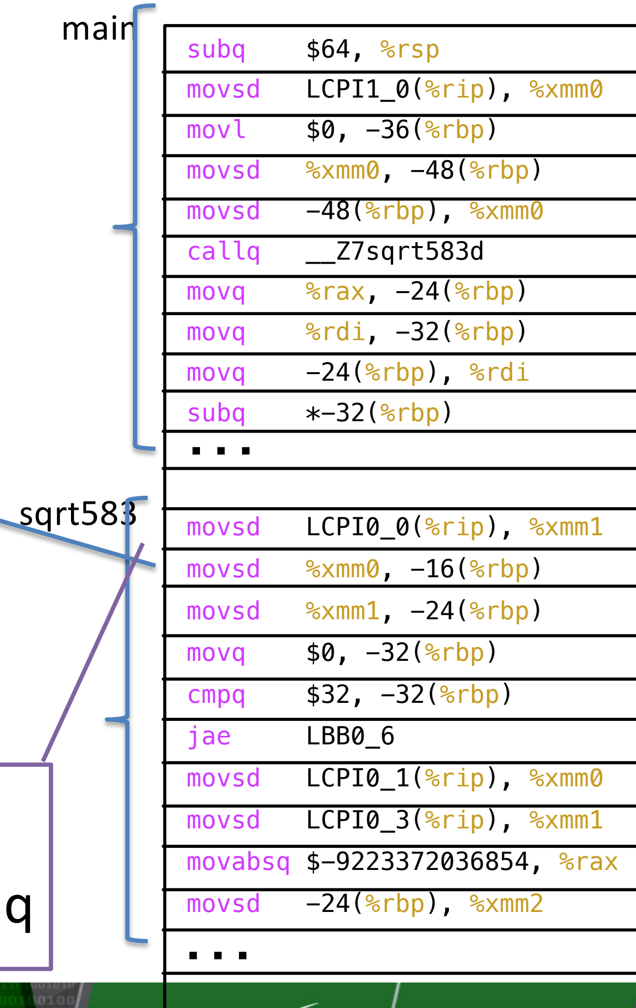
main
```
subq      $64, %rsp
movsd     LCPI1_0(%rip), %xmm0
movl      $0, -36(%rbp)
movsd     %xmm0, -48(%rbp)
movsd     -48(%rbp), %xmm0
callq     __Z7sqrt583d
movq      %rax, -24(%rbp)
movq      %rdi, -32(%rbp)
movq      -24(%rbp), %rdi
subq      *-32(%rbp)
. . .
```

sqrt583
```
movsd     LCPI0_0(%rip), %xmm1
movsd     %xmm0, -16(%rbp)
movsd     %xmm1, -24(%rbp)
movq      $0, -32(%rbp)
cmpq      $32, -32(%rbp)
jae       LBB0_6
movsd     LCPI0_1(%rip), %xmm0
movsd     LCPI0_3(%rip), %xmm1
movabsq   $-9223372036854, %rax
movsd     -24(%rbp), %xmm2
. . .
```

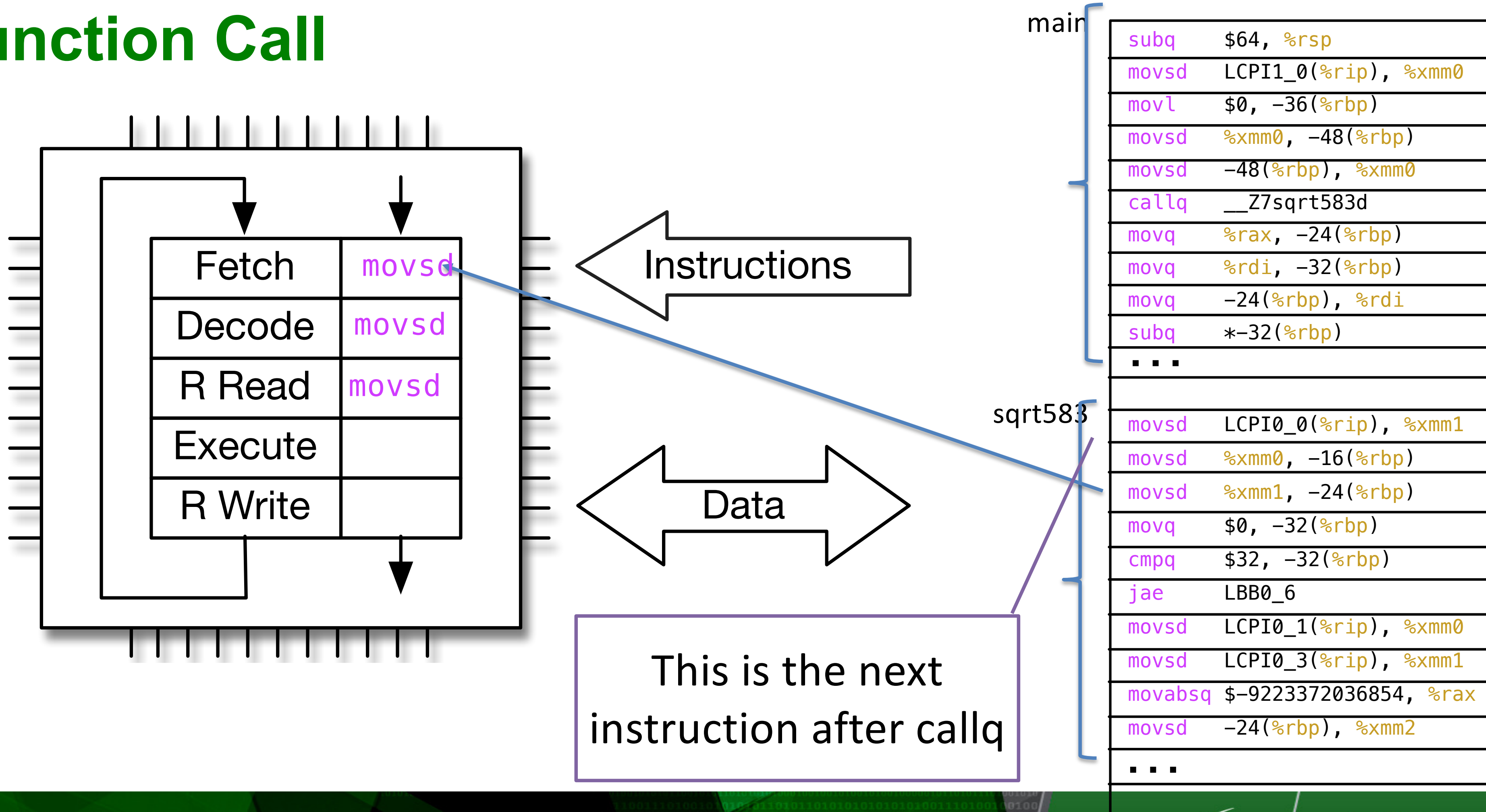
This is the next instruction after callq

Function Call

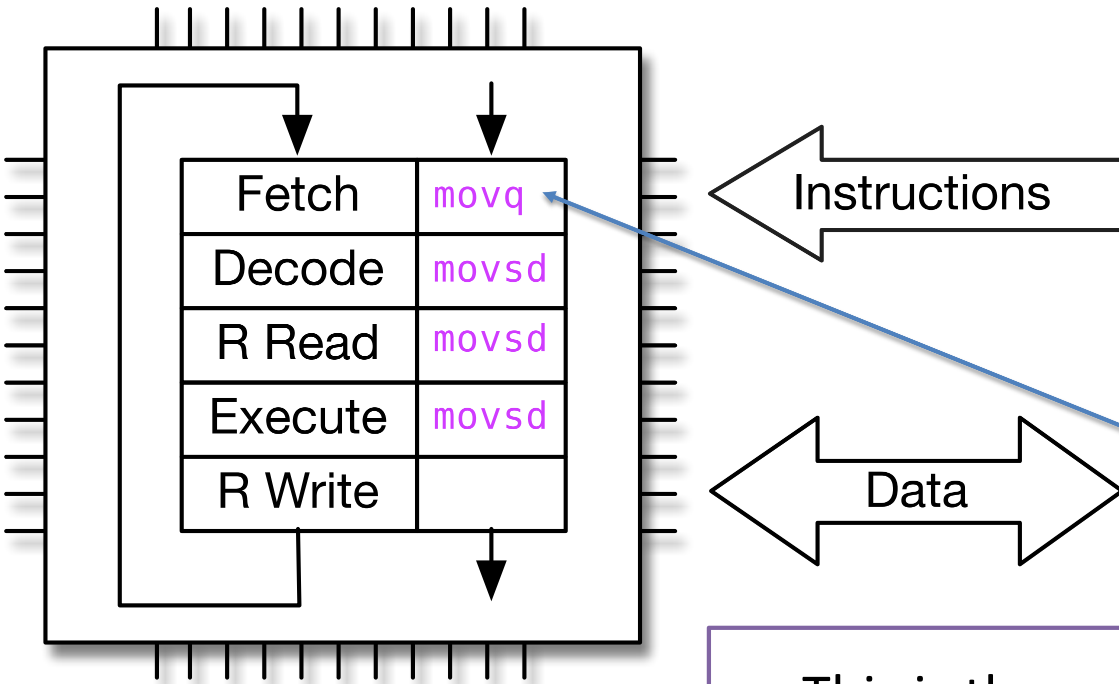




Function Call



Function Call



main

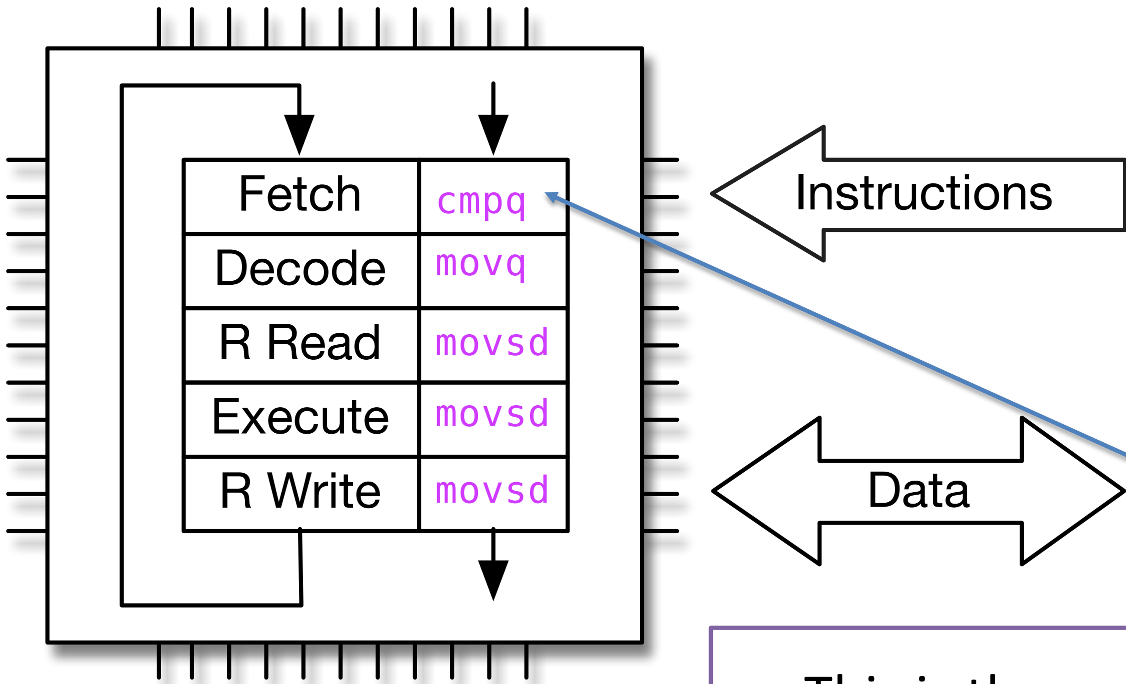
```
subq      $64, %rsp
movsd     LCPI1_0(%rip), %xmm0
movl      $0, -36(%rbp)
movsd     %xmm0, -48(%rbp)
movsd     -48(%rbp), %xmm0
callq     __Z7sqrt583d
movq      %rax, -24(%rbp)
movq      %rdi, -32(%rbp)
movq      -24(%rbp), %rdi
subq      *-32(%rbp)
. . .
```

sqrt583

```
movsd     LCPI0_0(%rip), %xmm1
movsd     %xmm0, -16(%rbp)
movsd     %xmm1, -24(%rbp)
movq      $0, -32(%rbp)
cmpq      $32, -32(%rbp)
jae       LBB0_6
movsd     LCPI0_1(%rip), %xmm0
movsd     LCPI0_3(%rip), %xmm1
movabsq   $-9223372036854, %rax
movsd     -24(%rbp), %xmm2
. . .
```

This is the next instruction after callq

Function Call



main

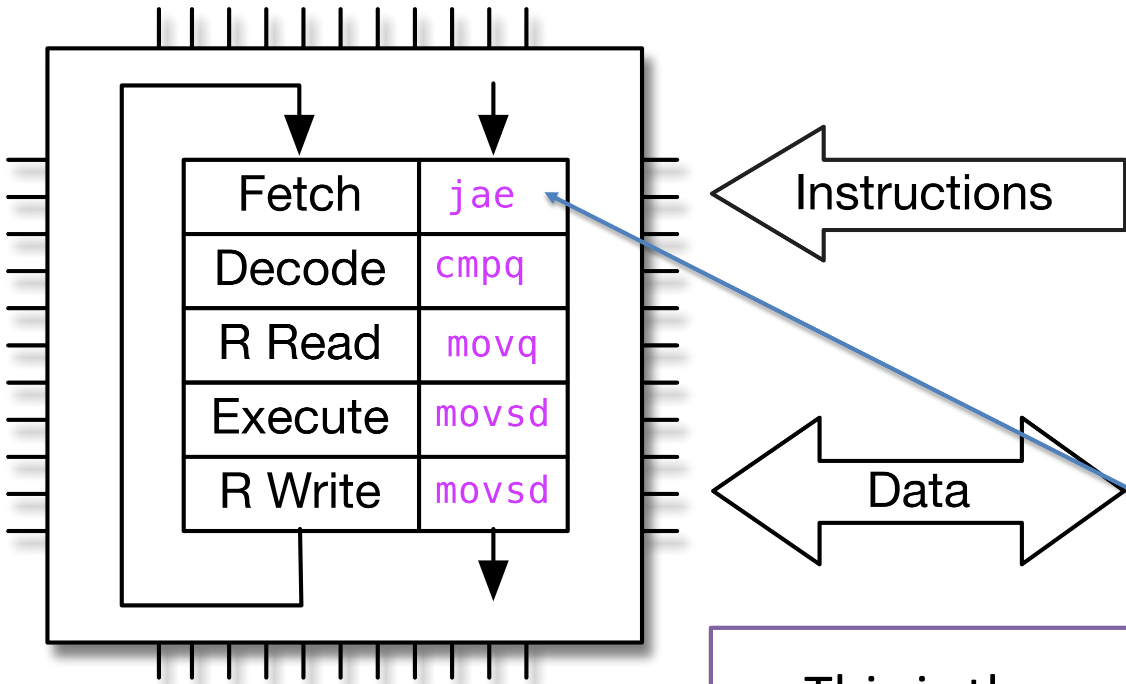
```
subq      $64, %rsp
movsd     LCPI1_0(%rip), %xmm0
movl      $0, -36(%rbp)
movsd     %xmm0, -48(%rbp)
movsd     -48(%rbp), %xmm0
callq     __Z7sqrt583d
movq      %rax, -24(%rbp)
movq      %rdi, -32(%rbp)
movq      -24(%rbp), %rdi
subq      *-32(%rbp)
...
```

sqrt583

```
movsd     LCPI0_0(%rip), %xmm1
movsd     %xmm0, -16(%rbp)
movsd     %xmm1, -24(%rbp)
movq      $0, -32(%rbp)
cmpq      $32, -32(%rbp)
jae       LBB0_6
movsd     LCPI0_1(%rip), %xmm0
movsd     LCPI0_3(%rip), %xmm1
movabsq   $-9223372036854, %rax
movsd     -24(%rbp), %xmm2
...
```

This is the next instruction after callq

Function Call



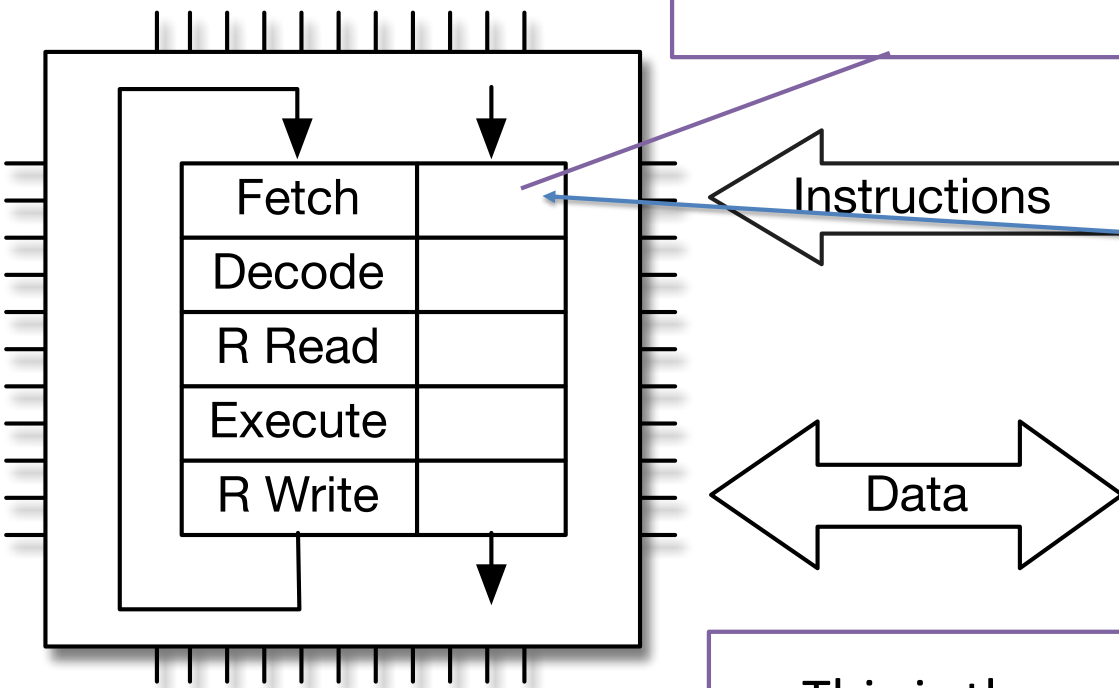
This is the next instruction after callq

main
```
subq      $64, %rsp
movsd     LCPI1_0(%rip), %xmm0
movl      $0, -36(%rbp)
movsd     %xmm0, -48(%rbp)
movsd     -48(%rbp), %xmm0
callq     __Z7sqrt583d
movq      %rax, -24(%rbp)
movq      %rdi, -32(%rbp)
movq      -24(%rbp), %rdi
subq      *-32(%rbp)
. . .
```

sqrt583
```
movsd     LCPI0_0(%rip), %xmm1
movsd     %xmm0, -16(%rbp)
movsd     %xmm1, -24(%rbp)
movq      $0, -32(%rbp)
cmpq      $32, -32(%rbp)
jae       LBB0_6
movsd     LCPI0_1(%rip), %xmm0
movsd     LCPI0_3(%rip), %xmm1
movabsq   $-9223372036854, %rax
movsd     -24(%rbp), %xmm2
. . .
```

Pipeline flush: Bad

Flush the pipeline



This is the next instruction after callq

main

```
subq      $64, %rsp
movsd     LCPI1_0(%rip), %xmm0
movl      $0, -36(%rbp)
movsd     %xmm0, -48(%rbp)
movsd     -48(%rbp), %xmm0
callq     __Z7sqrt583d
movq      %rax, -24(%rbp)
movq      %rdi, -32(%rbp)
movq      -24(%rbp), %rdi
subq      *-32(%rbp)
...
```

sqrt583

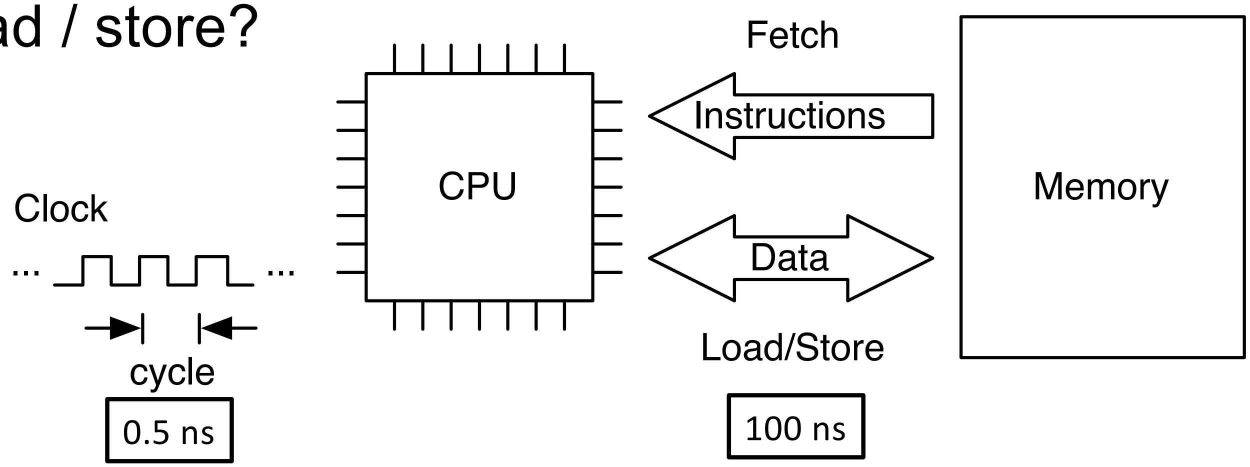
```
movsd     LCPI0_0(%rip), %xmm1
movsd     %xmm0, -16(%rbp)
movsd     %xmm1, -24(%rbp)
movq      $0, -32(%rbp)
cmpq      $32, -32(%rbp)
jae       LBB0_6
movsd     LCPI0_1(%rip), %xmm0
movsd     LCPI0_3(%rip), %xmm1
movabsq   $-9223372036854, %rax
movsd     -24(%rbp), %xmm2
...
```

Memory Access

- What are typical costs for accessing memory?
- What is typical clock cycle time?
- How many clock cycles to fetch an instruction? 200
- How many clock cycles to execute load / store instruction? 400
- CPI for load / store?

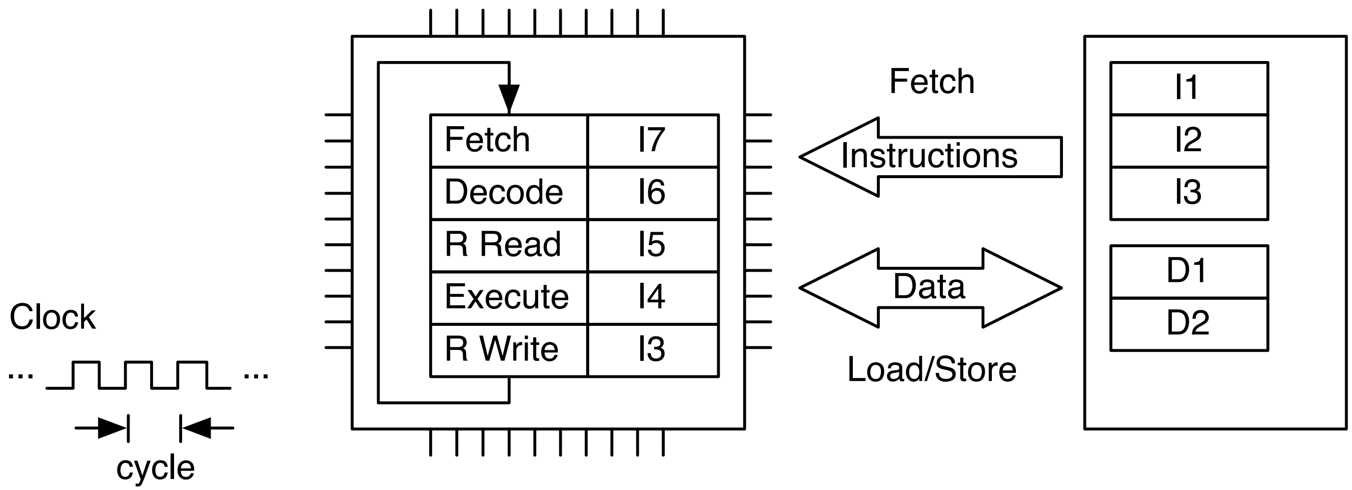
600

The next one may be cheaper



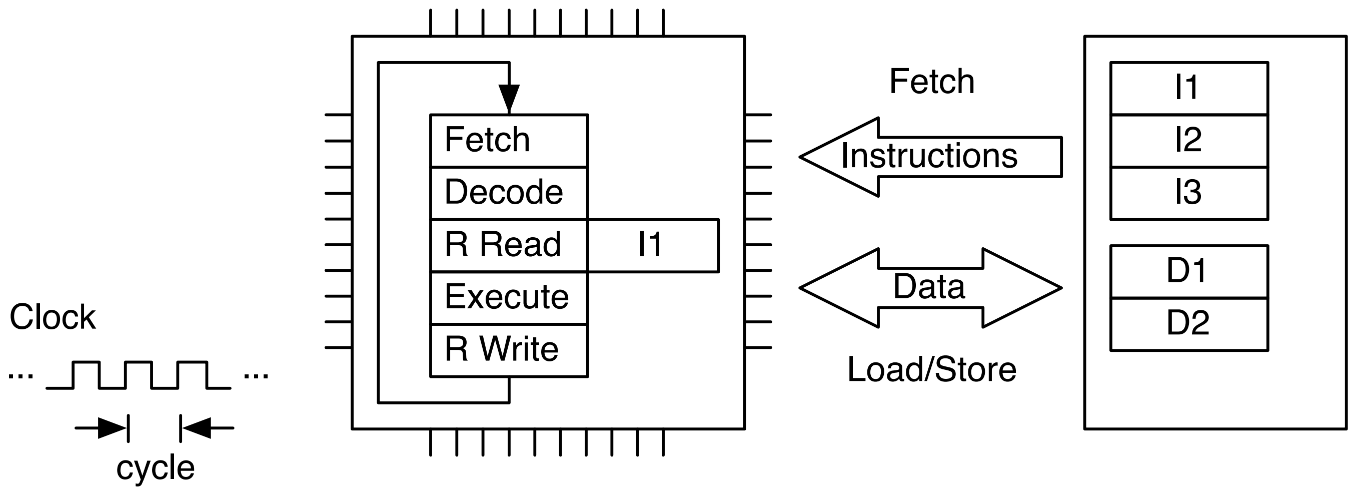
Memory Access Costs

- Access to main memory has huge impact on performance



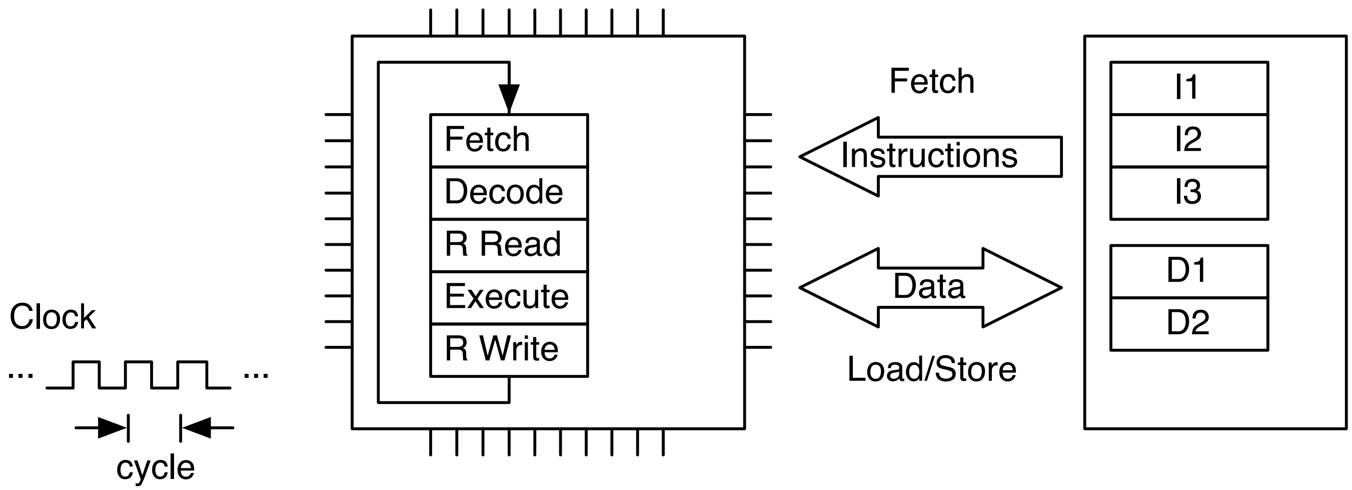
Memory Access Costs

- Access to main memory has huge impact on performance
- ***Latency***: How long does the first access to data take
- ***Bandwidth***: How much data can we continuously fetch

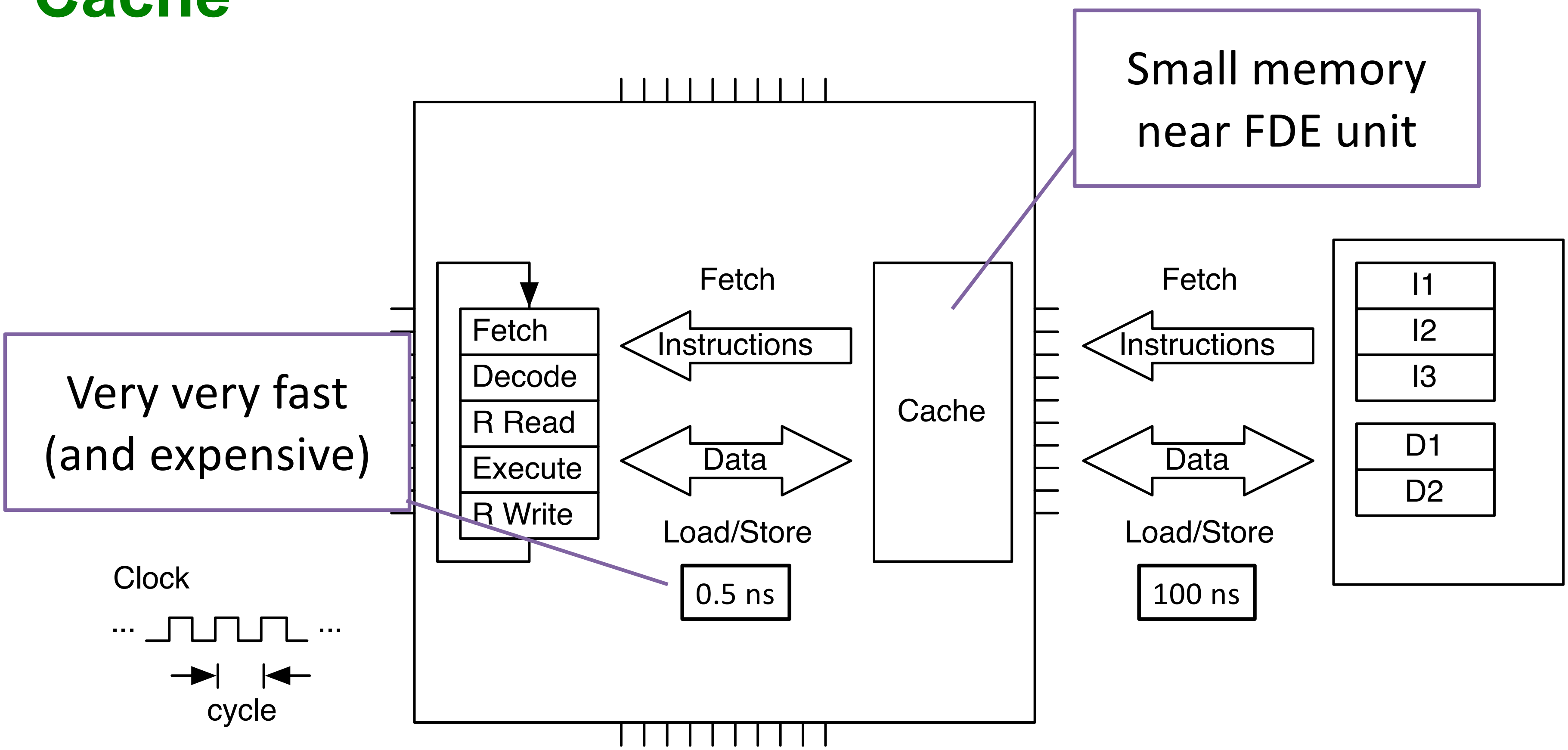


Memory Access Costs

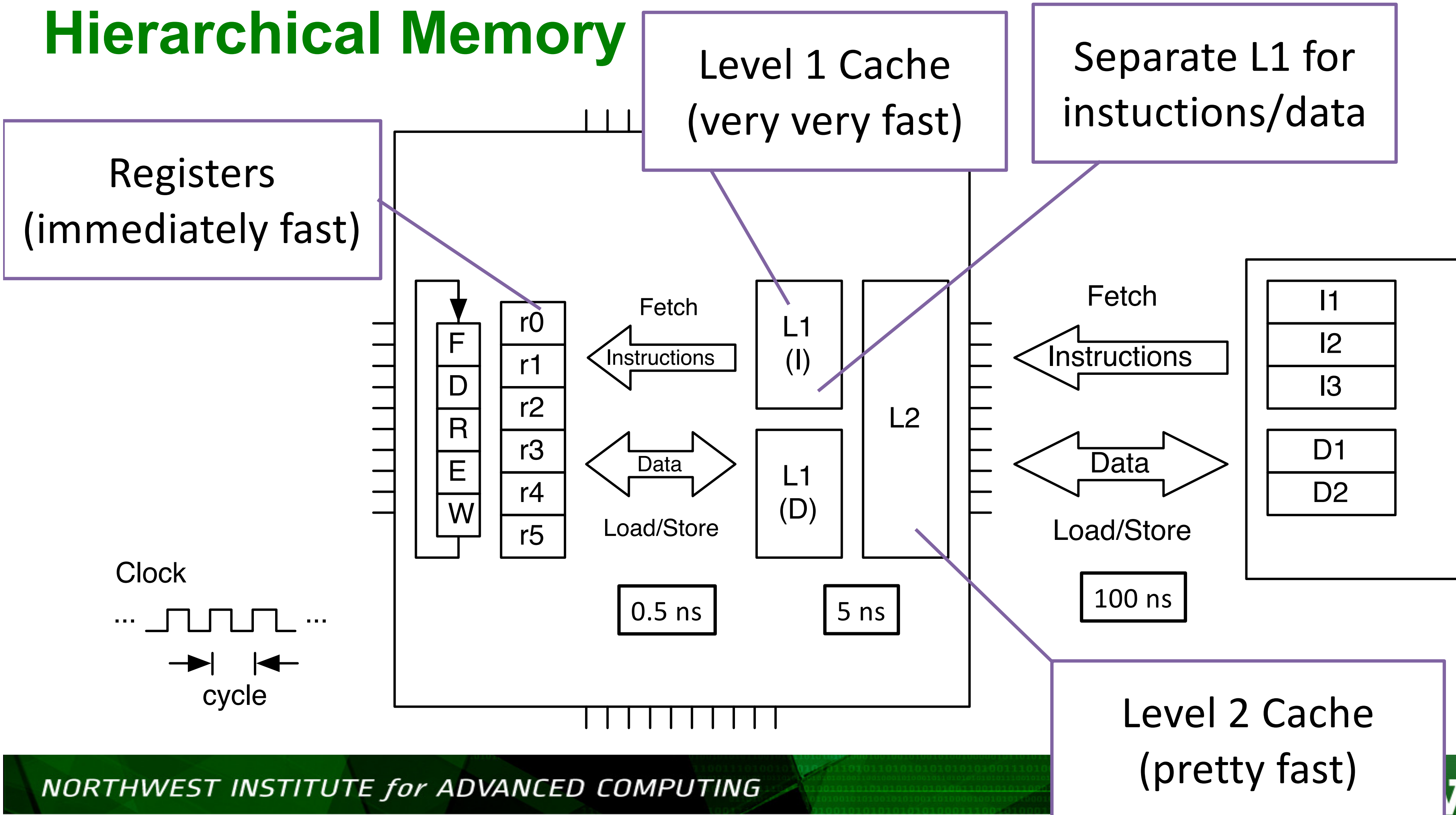
- Access to main memory has huge impact on performance (600X)
- Processor would be idle almost all the time



Cache


Small memory near FDE unit
Very very fast (and expensive)
Fetch
Decode
R Read
Execute
R Write
Fetch
Instructions
Data
Load/Store
0.5 ns
Cache
Fetch
Instructions
Data
Load/Store
100 ns
I1
I2
I3
D1
D2
Clock
cycle

Hierarchical Memory



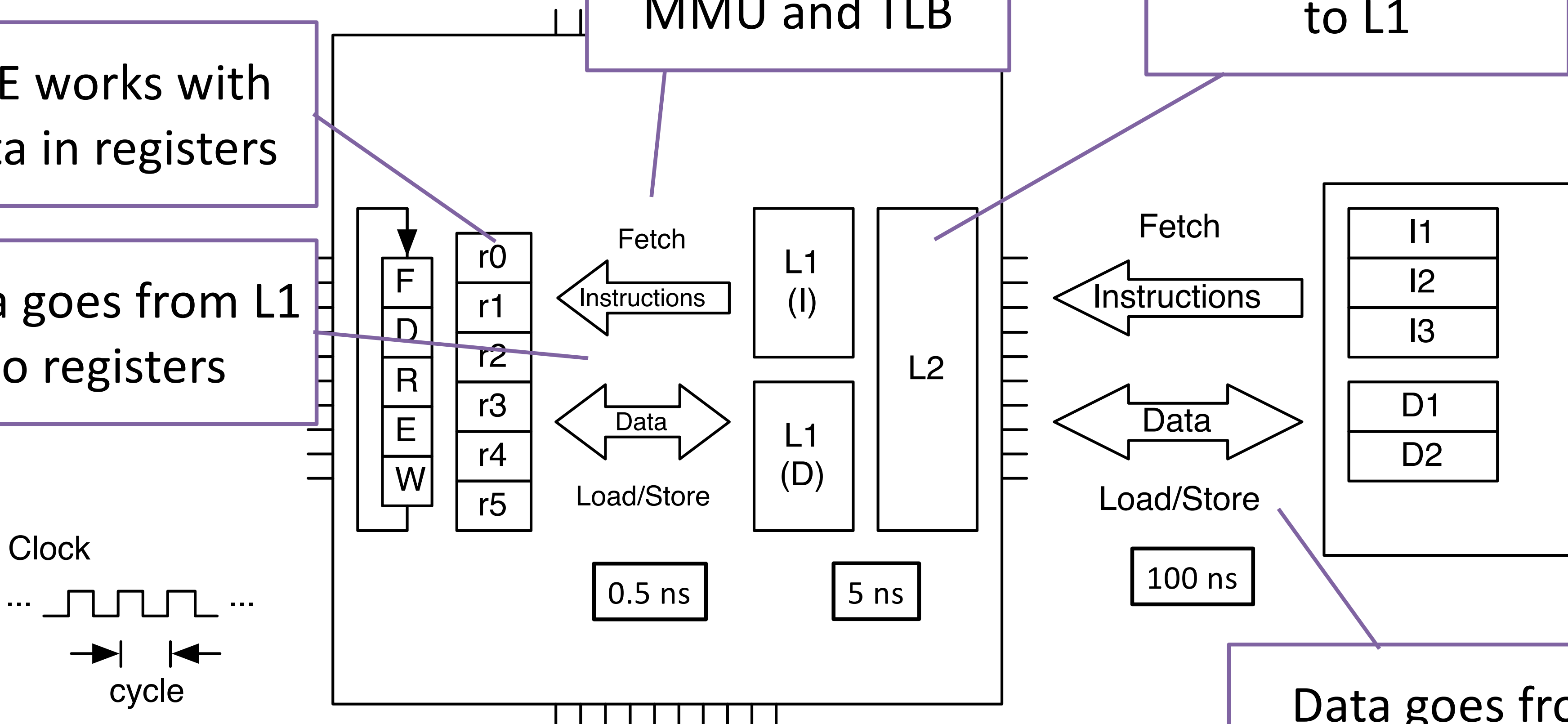
Hierarchical Memory

There is also an MMU and TLB

Data goes from L2 to L1

FDE works with data in registers

Data goes from L1 to registers



Data goes from main memory to L2

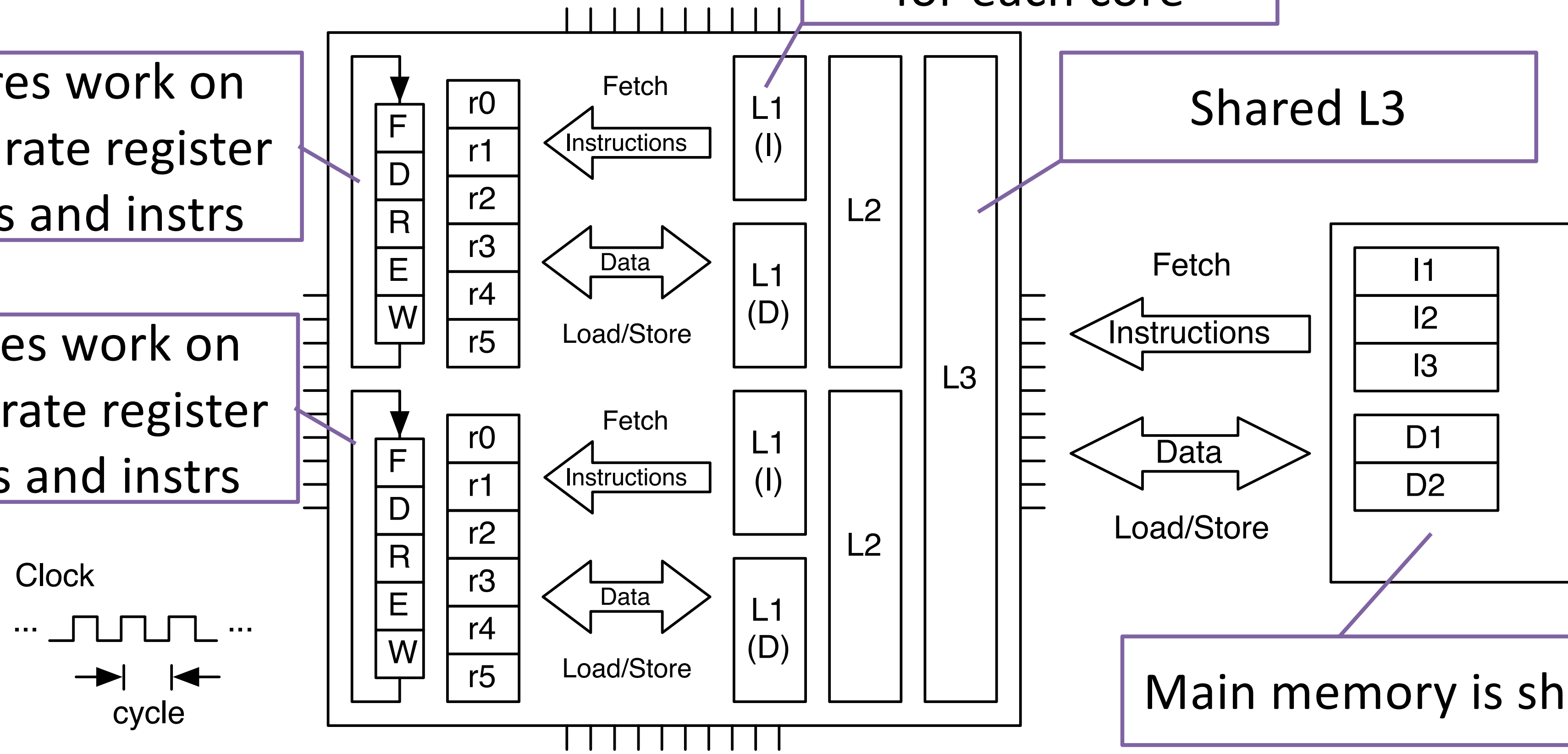
Cache and Multicore

Separate L1 and L2 for each core

Cores work on separate register sets and instrs

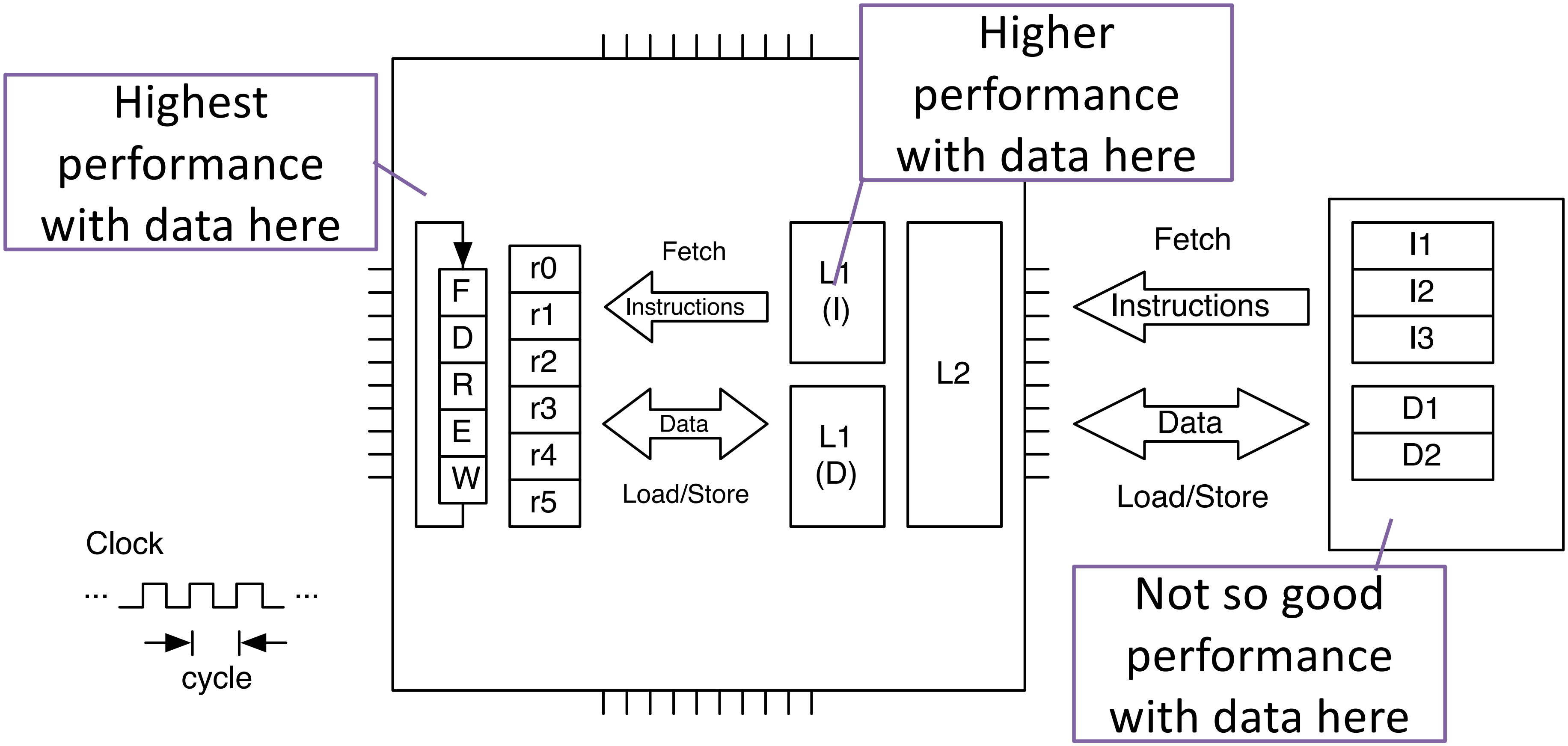
Cores work on separate register sets and instrs

Shared L3

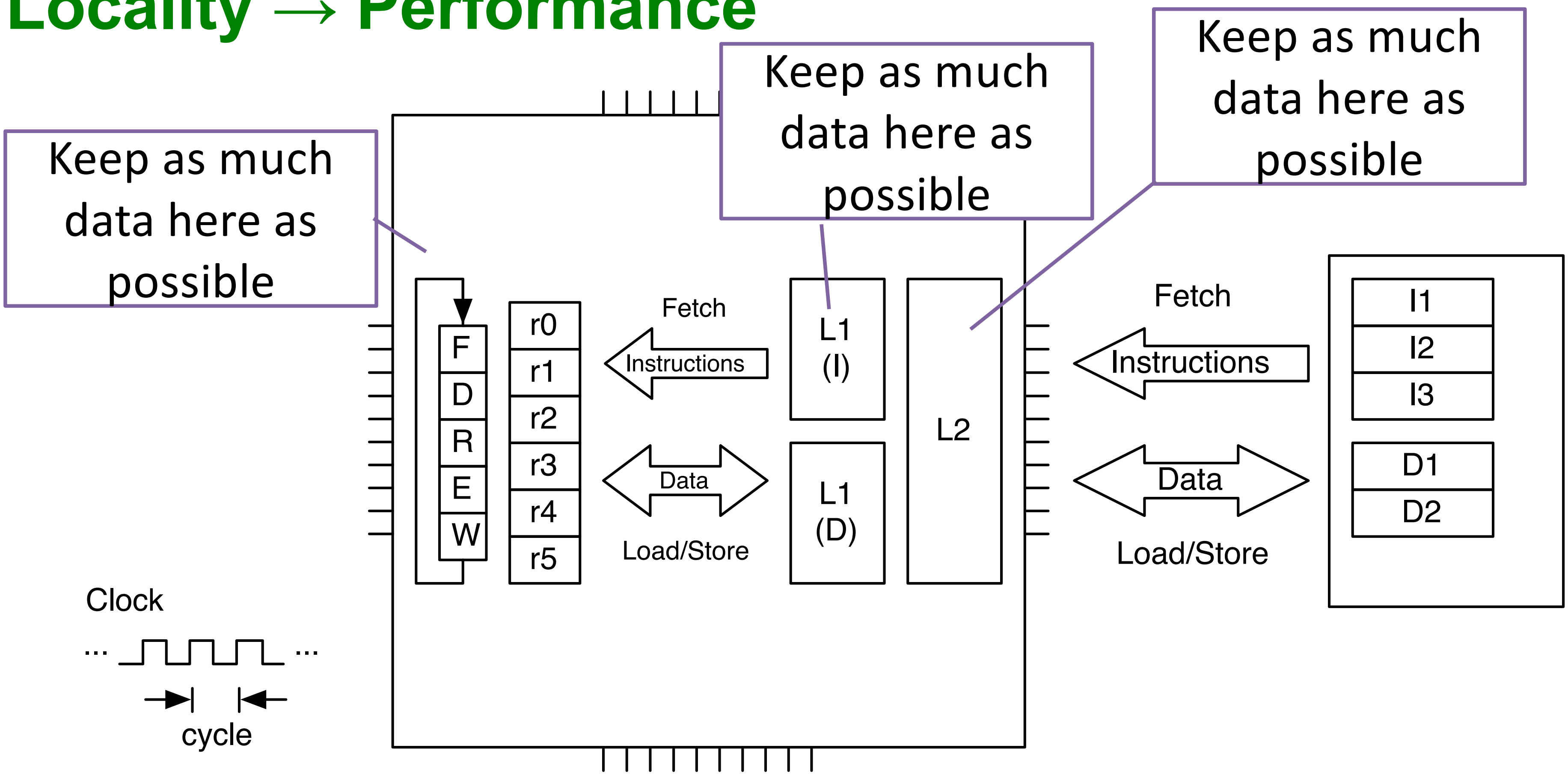


Main memory is shared

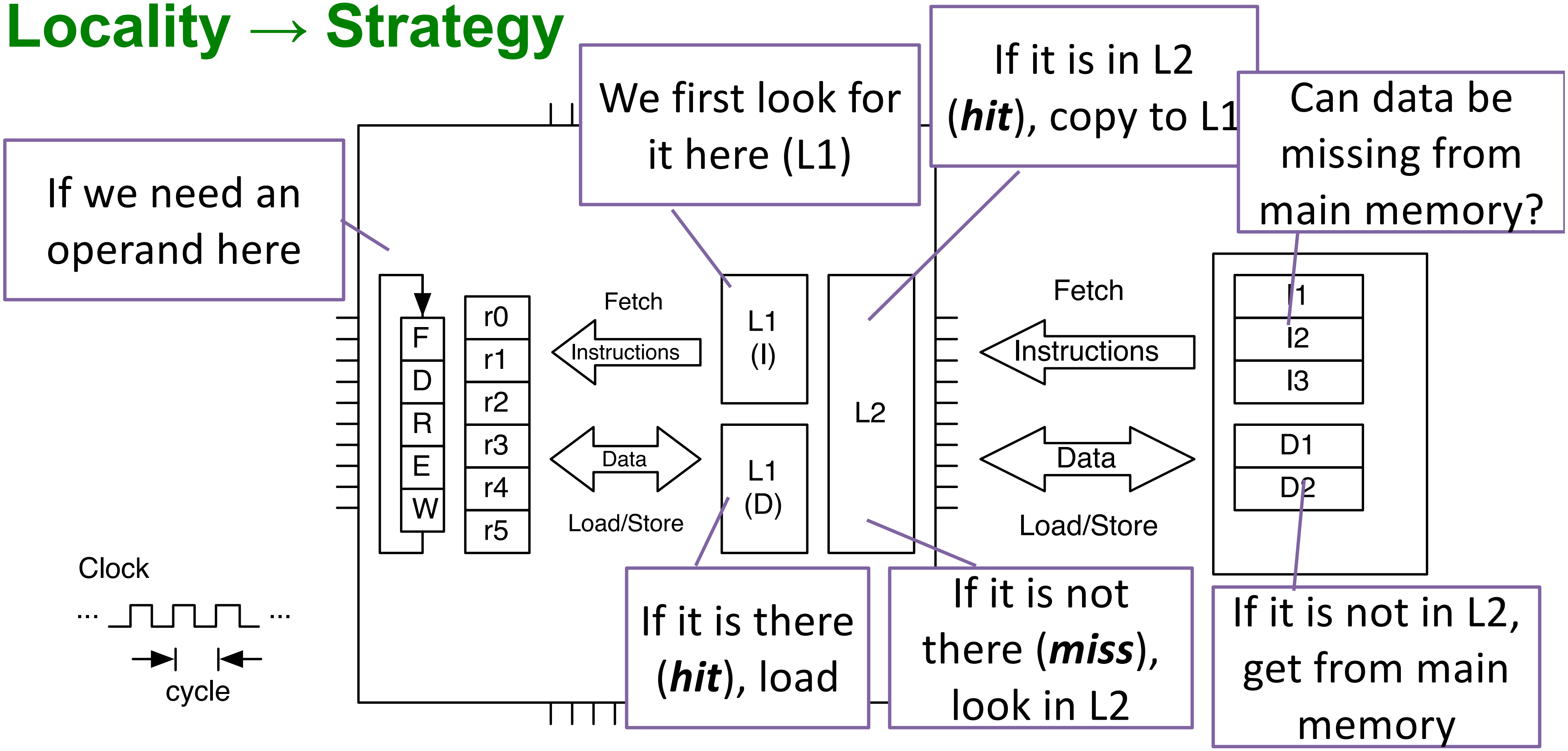
Performance



Locality → Performance



Locality → Strategy

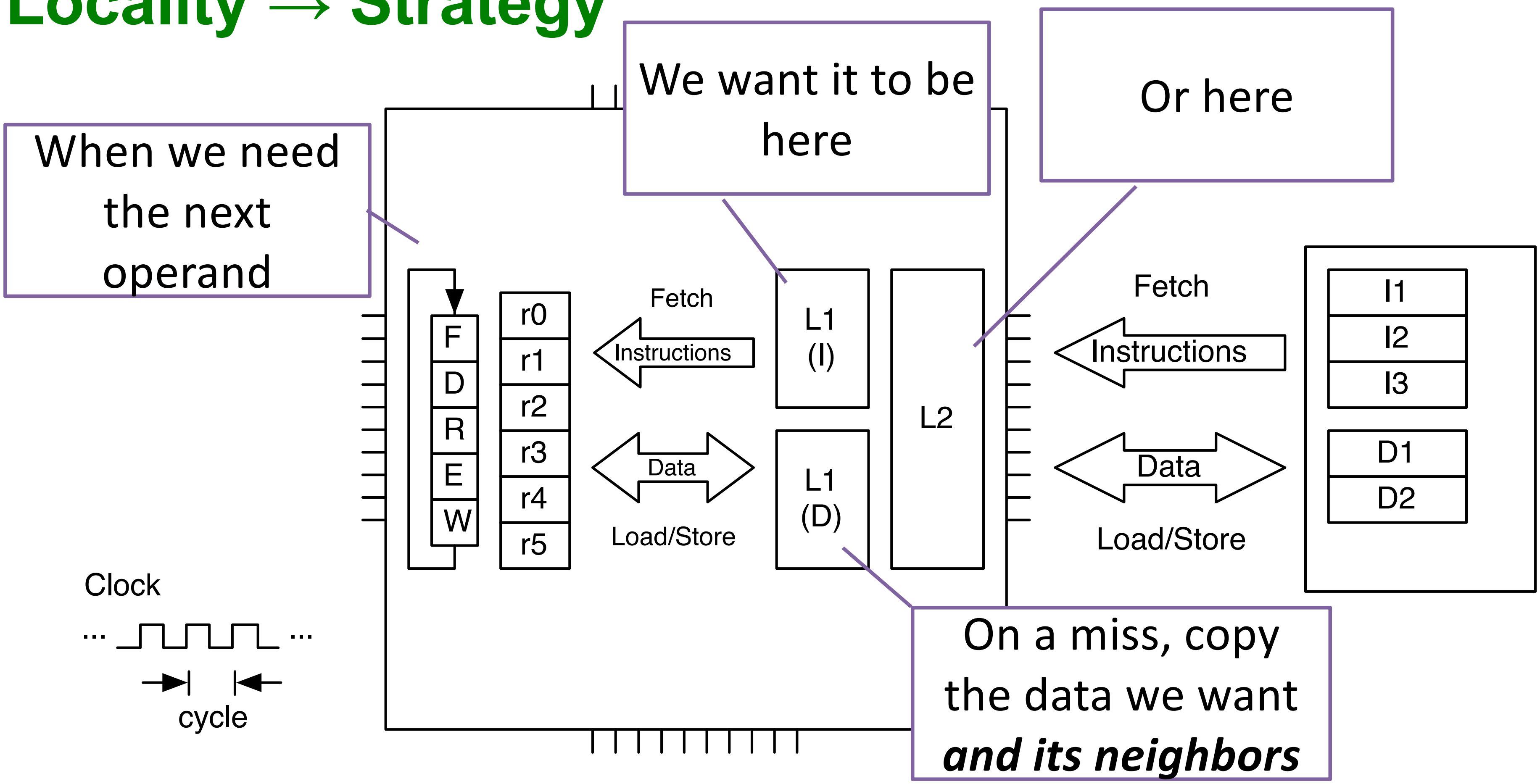


NORTHWEST INSTITUTE *for* ADVANCED COMPUTING

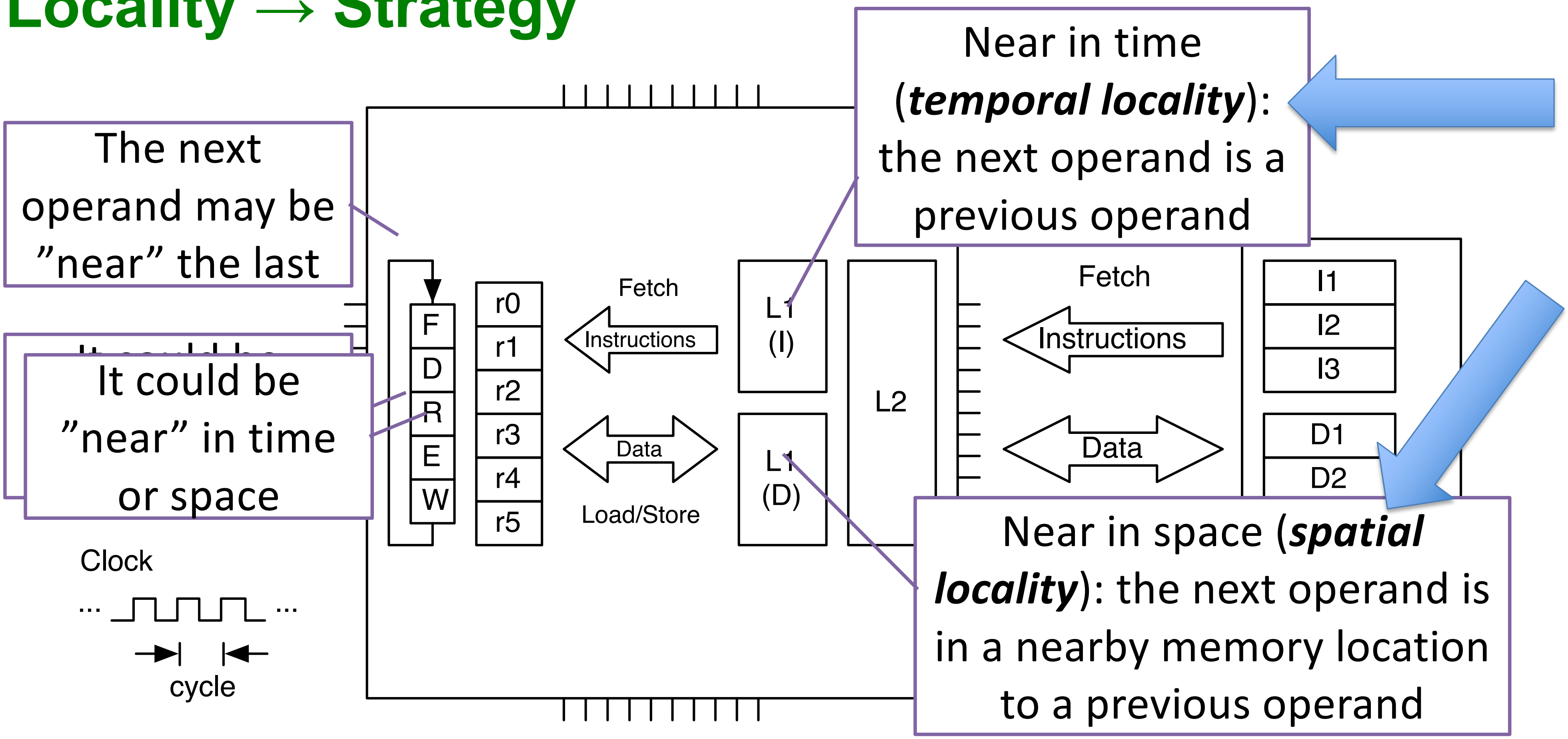
Pacific Northwest NATIONAL LABORATORY

UNIVERSITY of WASHINGTON

Locality → Strategy



Locality → Strategy



Locality → Performance

- Caches are much smaller than main memory. How do we decide what data to keep in cache to effect higher performance (more accesses)?
- **Temporal Locality**: if a program accesses a memory location, there is a much higher than random probability that the same location will be accessed again
  - Cache replacement policies attempt to keep cached elements in the cache for as long as possible
- **Spatial Locality**: if a program accesses a memory location, there is a much higher than random probability that nearby locations will also be accessed (soon)
  - Cache policies read contiguous chunks of data – a referenced element and its neighbors – not just single elements

Matrix Vector Product

- Recall for ANN $x^{i+1} = S(W^i x^i)$
- In general $y \leftarrow A \times x$

$$x^{i+1} \leftarrow W^i \times x^i$$

$$y_i = \sum_{i=0}^{N-1} A_{ij} x_j, \quad i = 0, \ldots, M$$

num_cols()

num_rows()

summation

Two nested loops

```
for (size_t i = 0; i < A.num_rows(); ++i) {
  for (size_t j = 0; j < A.num_cols(); ++j) {
    y(i) += A(i, j) * x(j);
  }
}
```

How many flops?

How many times is this done?

How much data?

Matrix-matrix product

$$C_{ij} = \sum_{k=0}^{K-1} A_{ik} B_{kj}$$

```
for (size_t i = 0; i < C.num_rows(); ++i) {
  for (size_t j = 0; j < C.num_cols(); ++j) {
    for (size_t k = 0; k < A.num_cols(); ++k) {
      C(i, j) += A(i, k) * B(k, j);
    }
  }
}
```

Three nested loops

How many flops?

How many times is this done?

How much data?

Timing and Benchmarking

- Humans have pathological need to see who is better at everything
- But ordering requires a single number corresponding to "goodness"
- Which is impossible of course
- So we take one task and turn that into the definition of goodness (cf IQ)
  - (What is IQ? It's the thing that the IQ test measures.)

My personal rant

- In HPC, we take performance on a particular computational task to rank the worlds computers with the 500 best scores on this task
  - Linear system solution – matrix matrix product at the core
  - Performance = FLOPS = (Total computations) / (Time to compute)
  - Linpack → $2N^3$/ (Time to compute)

Timing a Program

- The time program in Linux (Unix) will measure time resources a process uses

```
$ time ls -lR /usr > /dev/null

real   0m0.464s
user   0m0.080s
sys   0m0.380s
```

Elapsed Wall Clock time

Time Spent running user code

Time Spent running system code

This is what we'll be using

But finer grained control

C++ Timer

And this will be provided to you

```
class Timer {
private:
  typedef std::chrono::time_point<std::chrono::system_clock> time_t;

public:
  Timer() : startTime(), stopTime() {}

  time_t start()    { return (startTime = std::chrono::system_clock::now()); }
  time_t stop()     { return (stopTime  = std::chrono::system_clock::now()); }
  double elapsed()  { return
    std::chrono::duration_cast<std::chrono::milliseconds>(stopTime-startTime).count(); }

private:
  time_t startTime, stopTime;
};
```

All you need to worry about

Measuring Matrix Matrix Product

```
#include <iostream>
#include "Matrix.hpp"
#include "Timer.hpp"
using namespace std;

int main() {
  cout << "N\tElapsed" << endl;

  for (int N = 8; N < 1024; N *= 2) {
    Matrix A(N, N), B(N, N), C(N, N), D(N, N);

    Timer T; T.start();
    A = B*C;
    T.stop();

    cout << N << "\t" << T.elapsed() << endl;
  }

  return 0;
}
```

Declare Timer T

Start Timer T

Stop Timer T

And???

Print Elapsed Time

Insufficient resolution

```
$ ./a.out
N         Elapsed
8         0
16        0
32        0
64        0
128       2
256       28
512       315
```

What All Are We Timing

```
Matrix operator*(const Matrix& A, const Matrix& B) {
  Matrix C(A.num_rows(), B.num_cols());
  zeroize(C);
  for (size_t i = 0; i < A.num_rows
    for (size_t j = 0; j < B.num_cols(); ++j ) {
      for (size_t k = 0; k < A.num_cols(); ++k ) {
        C(i, j) += A(i, k) * B(k, j);
      }
    }
  }
  return C;
}
```

Allocating a Matrix

Zeroing it out

The actual matrix product

Never allocate new memory in performance critical sections of code

Just For Benchmarking

```
Matrix operator*(const Matrix& A, const Matrix&B) {
  Matrix C(A.num_rows(), B.num_cols());
  zeroizeMatrix(C);
  multiply(A, B, C);
  return C;
}

void multiply(const Matrix& A, const Matrix&B, Matrix&C) {
  for (size_t i = 0; i < A.num_rows(); ++i) {
    for (size_t j = 0; j < B.num_cols(); ++j) {
      for (size_t k = 0; k < A.num_cols(); ++k) {
            C(i,j) += A(i,k) * B(k,j);
      }
    }
  }
}
```

C++ Core Guideline Violation

F.20: For "out" output values, prefer return values to output parameters

Benchmarking

```
double benchmark(int M, int N, int K, long numruns) {
  Matrix A(M, K), B(K, N), C(M, N);

  Timer T;
  T.start();
  for (int i = 0; i < numruns; ++i) {
    multiply(A, B, C);
  }
  T.stop();

  return T.elapsed();
}
```

Run the core loop many times to get sufficient resolution for small(er) sizes

Bonus Question (Advanced Topic)

```
double benchmark(int M, int N, int K, long n
  Matrix A(M, K), B(K, N), C(M, N);

  Timer T;
  T.start();
  for (int i = 0; i < numruns; ++i) {
    multiply(A, B, C);
  }
  T.stop();

  return T.elapsed();
}
```

If we have different multiply routines (and we will), how many of these do we write?

By how much do they differ?

How can we parameterize that?

Bonus Question (Advanced Topic)

```
double benchmark(int M, int N, int K, long nu
                        <something> f) {
  Matrix A(M, K), B(K, N), C(M, N);

  Timer T;
  T.start();
  for (int i = 0; i < numruns; ++i) {
    f(A, B, C);
  }
  T.stop();

  return T.elapsed();
}
```

We want to pass in something

That we call like a function

Double bonus: It just needs an operator()()

Let’s not get carried away

Functions as Data

```
#include <functional>

double benchmark(int M, int N, int K, long numruns,
  function<void (const Matrix&, const Matrix&, Matrix&)>f) {
  Matrix A(M, K), B(K, N), C(M, N);

  Timer T;
  T.start();
  for (int i = 0; i < numruns; ++i) {
    f(A, B, C);
  }
  T.stop();

  return T.elapsed();
}
```

Is a function

And takes two const Matrix& and a Matrix& for args

Parameter f

That returns void

Like multiply()

```
void multiply(const Matrix& A, const Matrix&B, Matrix&C);
```

Functions as Data (Advanced)

Functions returning void

And taking two const Matrix& and a Matrix& for args

```
void multiply(const Matrix& A, const Matrix &B, Matrix& C);
void multiply_2(const Matrix& A, const Matrix &B, Matrix& C);
void yet_another(const Matrix& A, const Matrix &B, Matrix& C);

//...

double t1 = benchmark(100, 100, 100, multiply);
double t2 = benchmark(100, 100, 100, multiply_2);
double t2 = benchmark(100, 100, 100, yet_another);
```

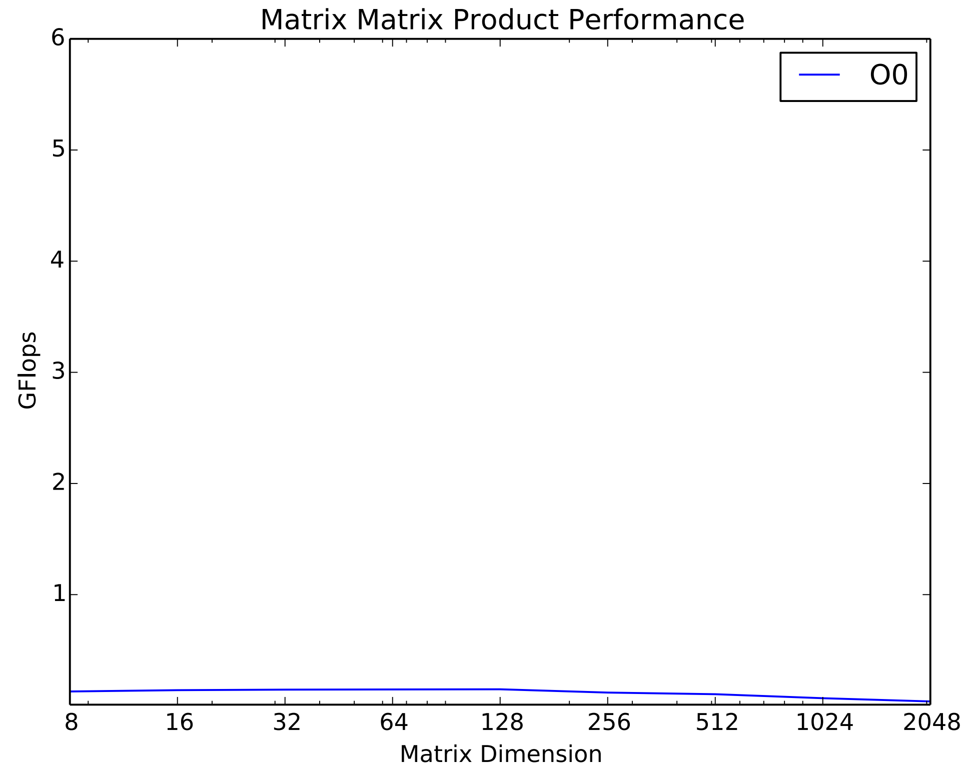
Pass them into another function

Let's Start Benchmarking

```
double benchmark(int M, int N, int K, long numruns) {
  Matrix A(M, K), B(K, N), C(M, N);

  Timer T;
  T.start();
  for (int i = 0; i < numruns; ++i) {
    multiply(A, B, C);
  }
  T.stop();

  return T.elapsed();
}
```

```
bench: bench.o Matrix.o
c++ -std=c++11 bench.o Matrix.o -o bench

bench.o: bench.cpp Matrix.hpp
c++ -std=c++11 -c bench.cpp -o bench.o

Matrix.o: Matrix.cpp Matrix.hpp
c++ -std=c++11 -c Matrix.cpp -o Matrix.o
```

Base Performance Results



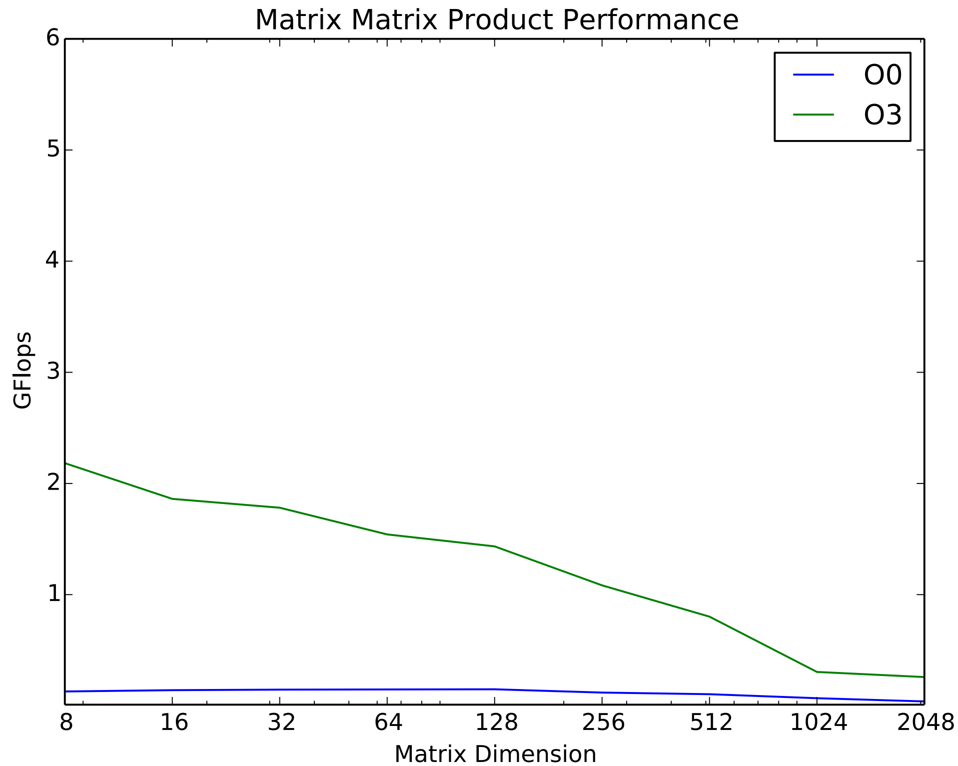
Let's Make One Small Change

```
double benchmark(int M, int N, int K, long numruns) {
  Matrix A(M, K), B(K, N), C(M, N);

  Timer T;
  T.start();
  for (int i = 0; i < numruns; ++i) {
    multiply(A, B, C);
  }
  T.stop();

  return T.elapsed();
}
```

Tell the compiler to use optimization level 3

```
bench: bench.o Matrix.o
c++ -O3 -std=c++11 bench.o Matrix.o -o bench

bench.o: bench.cpp Matrix.hpp
c++ -O3 -std=c++11 -c bench.cpp -o bench.o

Matrix.o: Matrix.cpp Matrix.hpp
c++ -O3 -std=c++11 -c Matrix.cpp -o Matrix.o
```

Base Performance Results



The Three Most Important Requirements for HPC

- Locality
- Locality
- Locality

Locality -> Performance

- Caches are much smaller than main memory. How do we decide what data to keep in cache to effect higher performance (more accesses)?
- **Temporal Locality:** if a program accesses a memory location, there is a much higher than random probability that the same location will be accessed again
  - Cache replacement policies attempt to keep cached elements in the cache for as long as possible
- **Spatial Locality:** if a program accesses a memory location, there is a much higher than random probability that nearby locations will also be accessed (soon)
  - Cache policies read contiguous chunks of data – a referenced element and its neighbors – not just single elements

Improving Locality

- Consider each step of inner loop

```
for (int i = 0; i < M; ++i)
  for (int j = 0; j < N; ++j)
    for (int k = 0; k < K; ++k)
      C(i,j) += A(i,k) * B(k,j);
```

What can be reused?

- Load `C(i,j)` into register
- Load `A(i,k)` into register
- Load `B(k,j)` into register
- Multiply
- Add
- Store `C(i,j)`

- Four memory operations and two floating point operations per iteration
- 1/3 flop per cycle (if each operation is one cycle)

Improving Locality

```
void multiply(const Matrix& A, const Matrix&B, Matrix&C) {
  for (size_t i = 0; i < A.num_rows(); ++i) {
    for (size_t j = 0; j < B.num_cols(); ++j) {
      for (size_ k = 0; k < A.num_cols(); ++k) {
        C(i,j) += A(i,k) * B(k,j);
      }
    }
  }
}
```

What can be reused?

- Load `C(i,j)` into register
- Load `A(i,k)` into register
- Load `B(k,j)` into register
- Multiply
- Add
- Store `C(i,j)`

- Four memory operations and two floating point operations per iteration
- 2/6 = 1/3 flop per cycle (if each operation is one cycle)

Hoisting

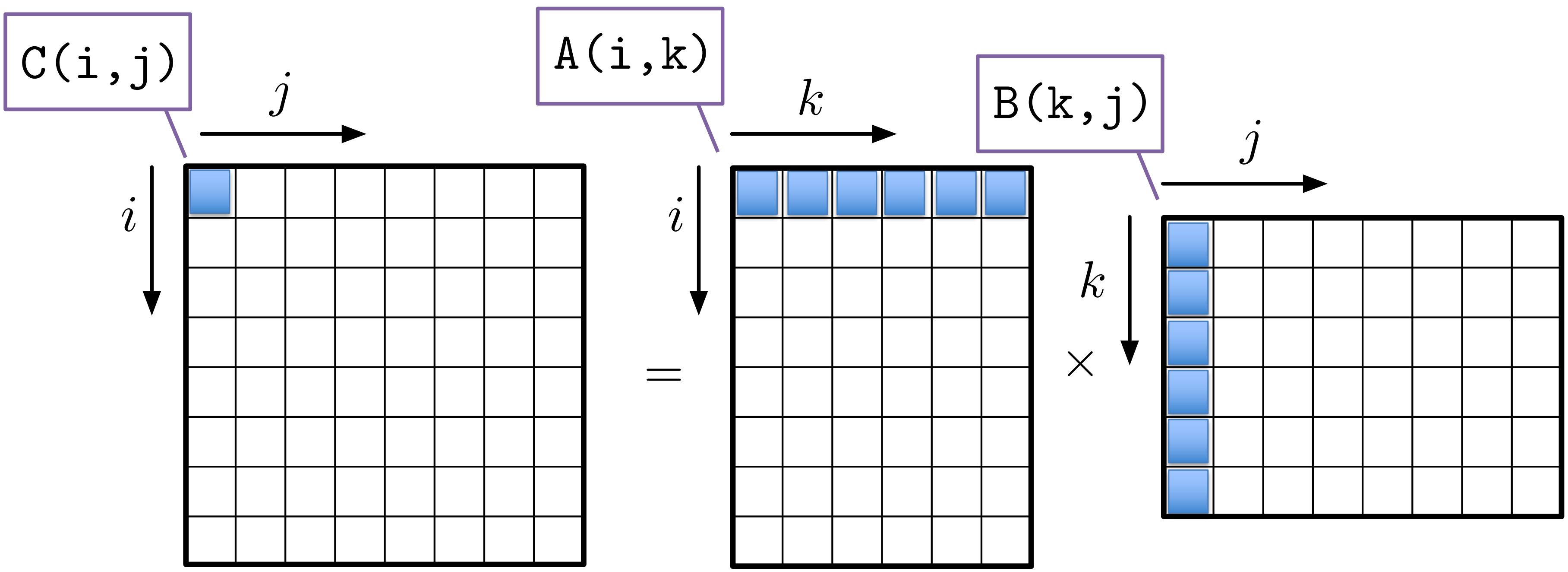
Hoist C(i,j)

```
void multiply(const Matrix& A, const Matrix&B, Matrix&C) {
  for (size_t i = 0; i < A.num_rows(); ++i) {
    for (size_t j = 0; j < B.num_cols(); ++j) {
      double t = C(i,j);
      for (size_t k = 0; k < A.num_cols(); ++k) {
        t += A(i,k) * B(k,j);
      }
      C(i,j) = t;
    }
  }
```

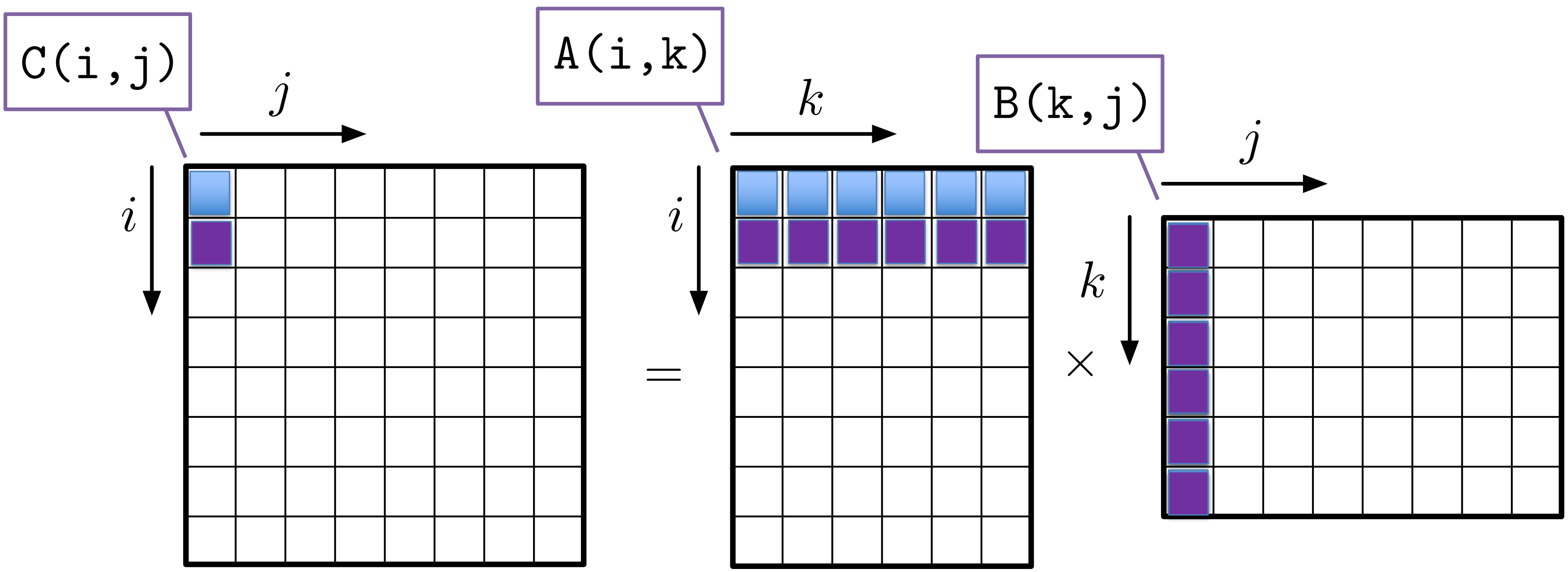
- Load `A(i,k)`
- Load `B(k,j)`
- Multiply
- Add

- Two memory operations and two floating point operations per iteration
- 2/4 = 1/2 flop per cycle (if each operation is one cycle)

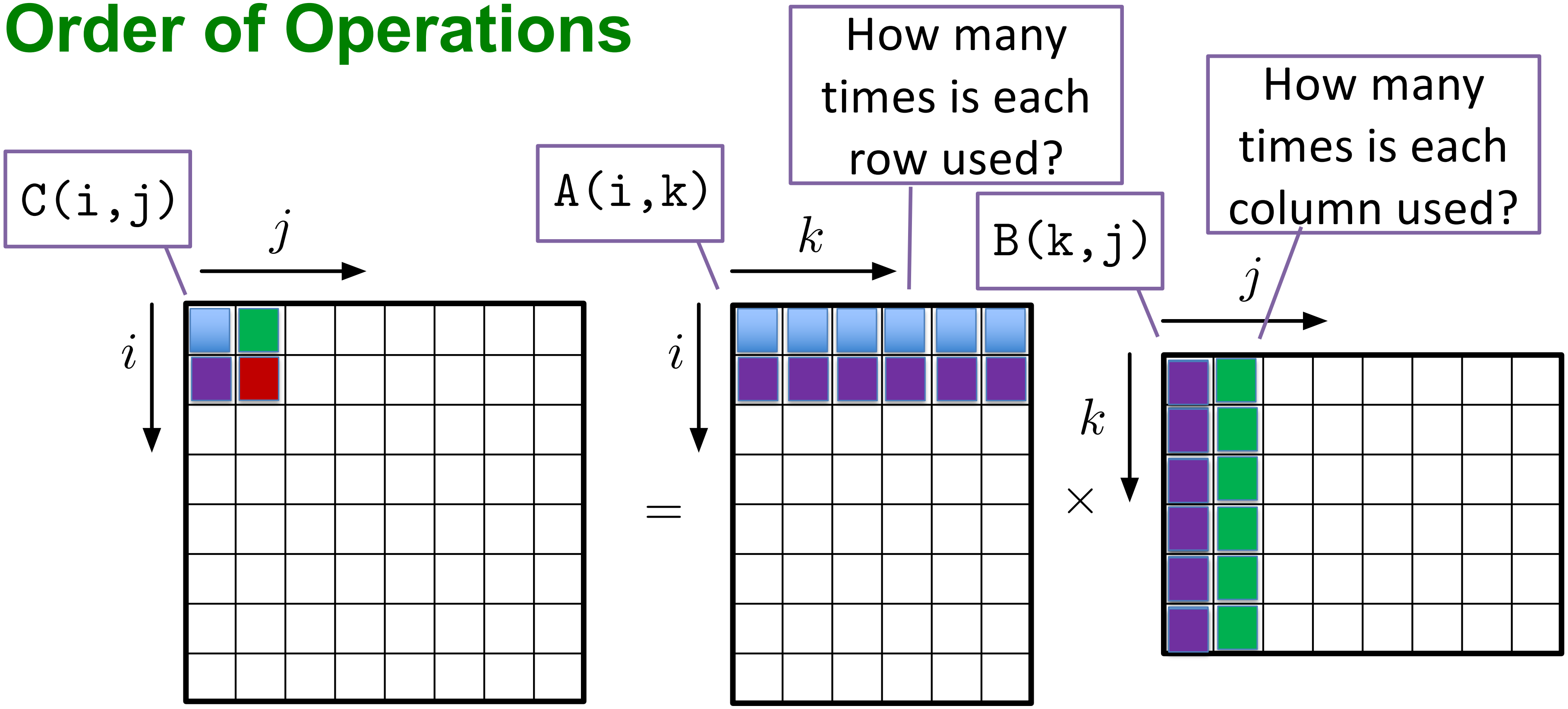
Order of Operations



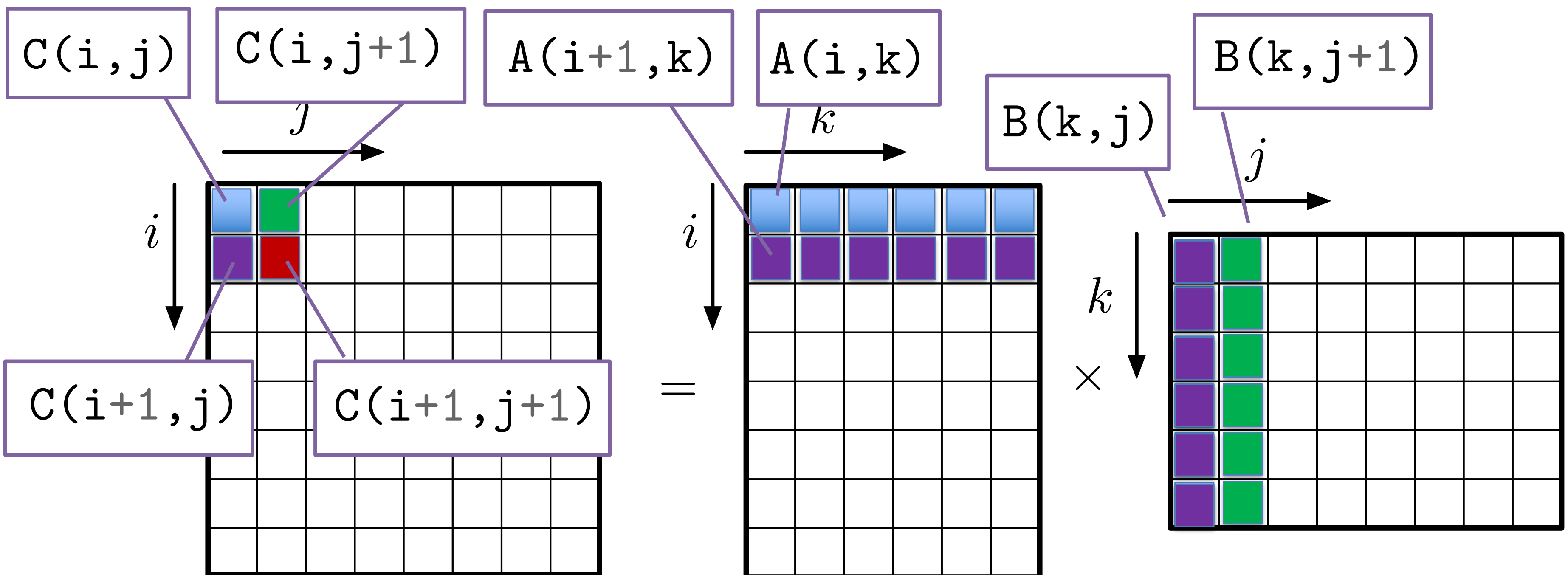
Order of Operations



Order of Operations



Reuse: How Many Times Are Data Reused?



Each is used twice

Improving Locality: Unroll and Ja

```
void tiledMultiply2x2(const Matrix& A, const Matrix&B
  for (size_t i = 0; i < A.num_rows(); i += 2) {
    for (size_t j = 0; j < B.num_cols(); j += 2) {
      for (size_t k = 0; k < A.num_cols(); ++k) {
            C(i  , j  ) += A(i  , k) * B(k, j  );
            C(i  , j+1) += A(i  , k) * B(k, j+1);
            C(i+1, j  ) += A(i+1, k) * B(k, j  );
            C(i+1, j+1) += A(i+1, k) * B(k, j+1);
      }
    }
  }
}
```

B(k,j) is used twice

B(k,j+1) is used twice

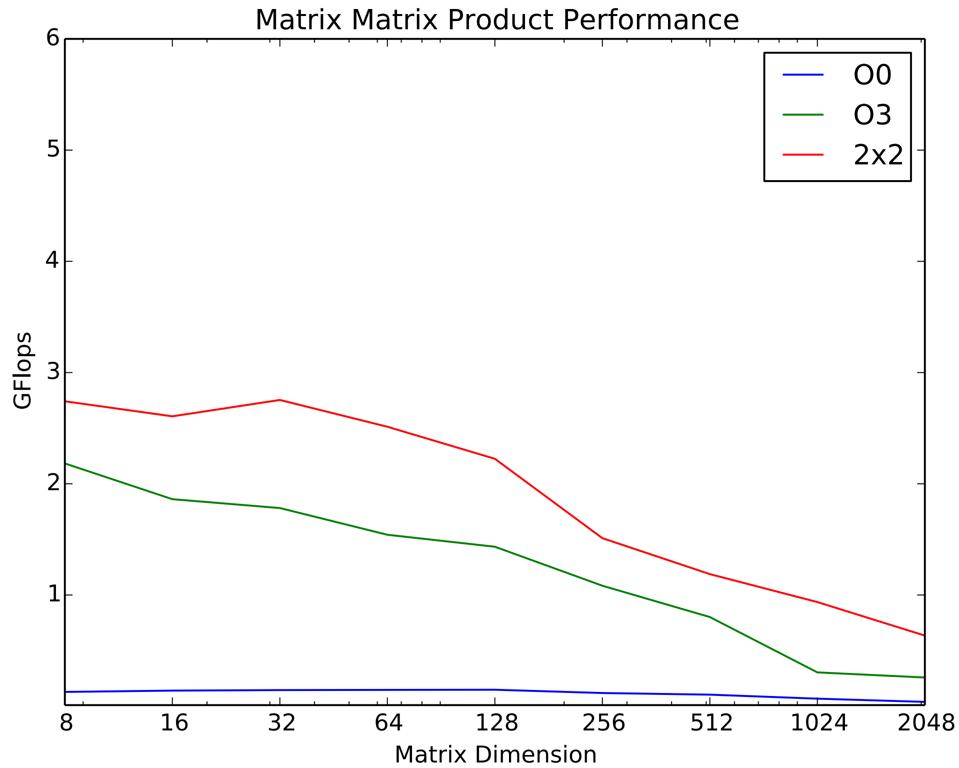
A(i,k) is used twice

A(i+1,k) is used twice

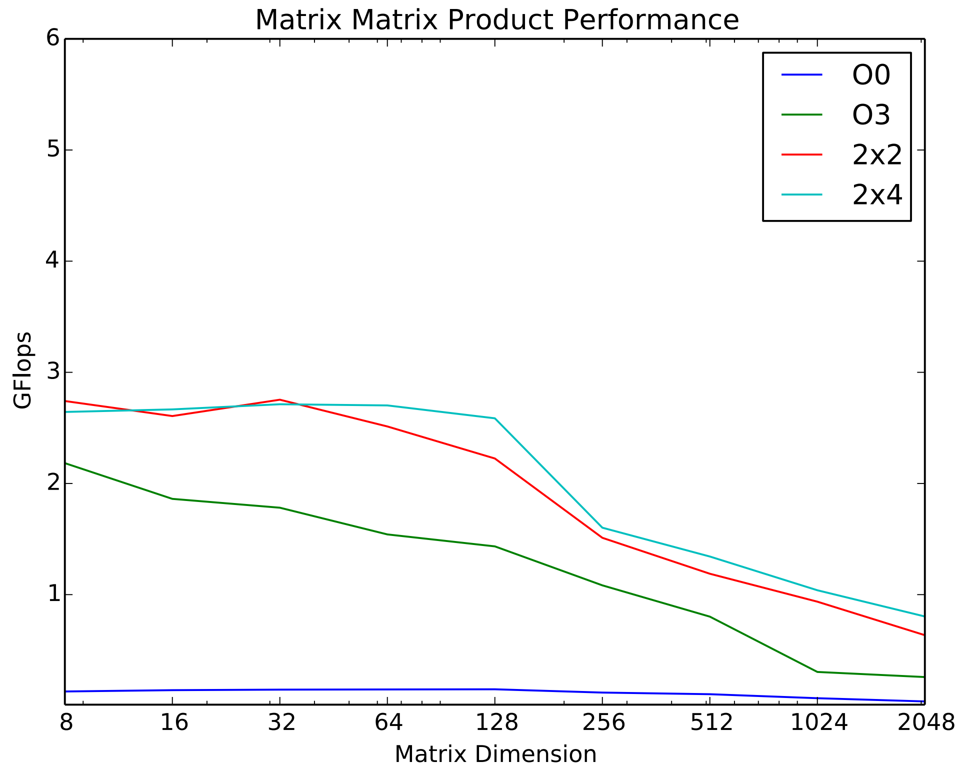
Can also hoist (independent of k)

- Four memory operations and eight floating point operations per iteration
- 8/12 = 2/3 flop per cycle (if each operation is one cycle) – 2X the base case

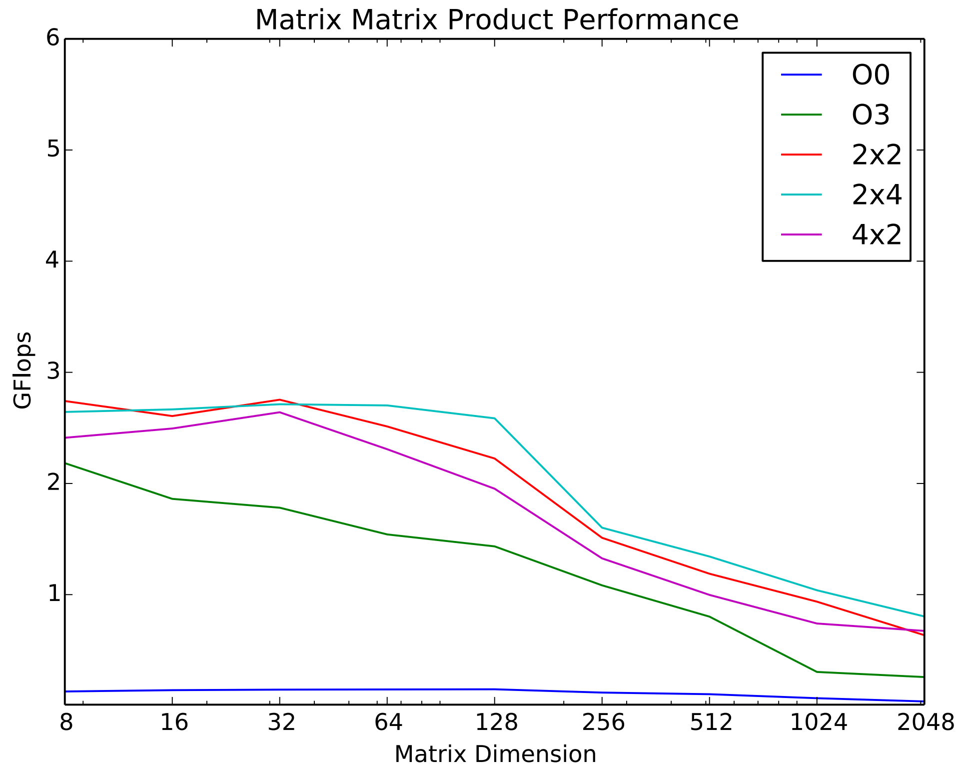
Example: Register Locality



2 by 4

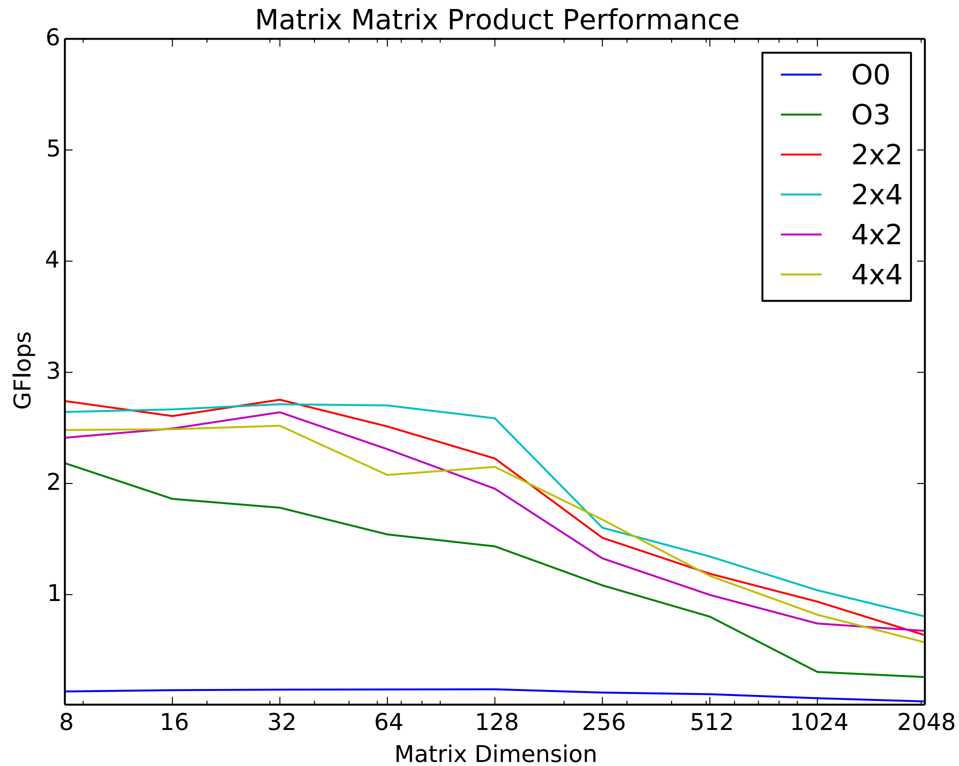

Matrix Matrix Product Performance
GFlops
Matrix Dimension
6
5
4
3
2
1
8
16
32
64
128
256
512
1024
2048
O0
O3
2x2
2x4

4 by 2



Matrix Matrix Product Performance
GFlops
Matrix Dimension
6
5
4
3
2
1
8
16
32
64
128
256
512
1024
2048
O0
O3
2x2
2x4
4x2

4 by 4

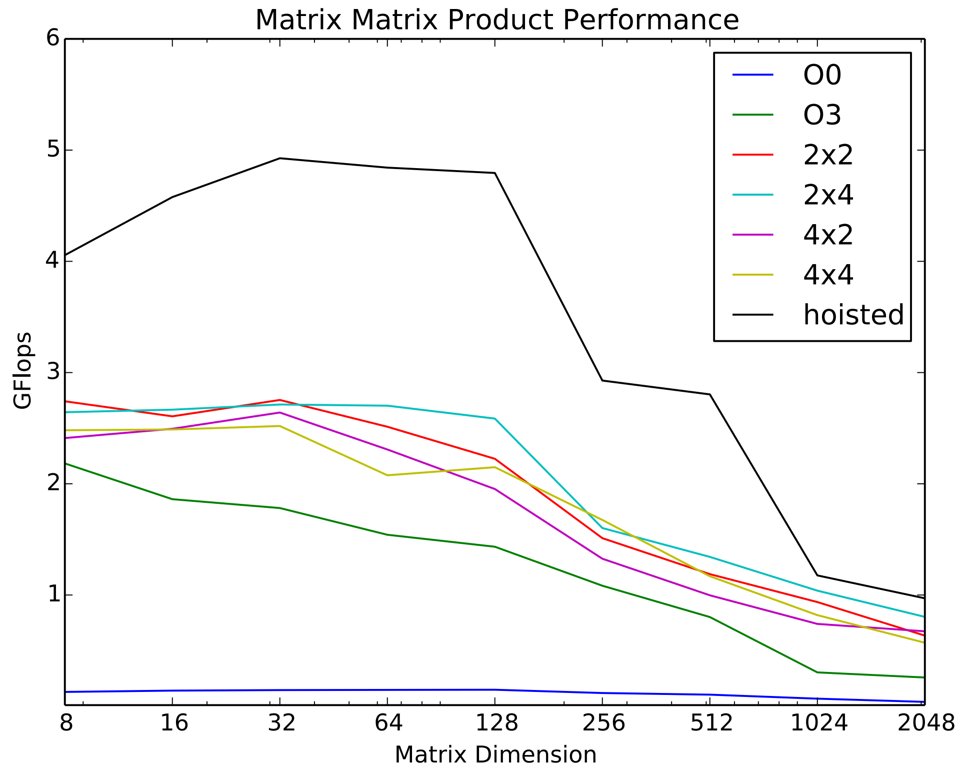

Matrix Matrix Product Performance
GFlops
Matrix Dimension
6
5
4
3
2
1
8
16
32
64
128
256
512
1024
2048
O0
O3
2x2
2x4
4x2
4x4

Tiling and Hoisting

```
void hoistedTiledMultiply2x2(const Matrix& A, const Matrix&B, Matrix&C) {
  for (size_t i = 0; i < A.num_rows(); i += 2) {
    for (size_t j = 0; j < B.num_cols(); j += 2) {
      double t00 = C(i,  j);     double t01 = C(i,  j+1);
      double t10 = C(i+1,j);     double t11 = C(i+1,j+1);
      for (size_t k = 0; k < A.num_cols(); ++k) {
            t00 += A(i  , k) * B(k, j  );
            t01 += A(i  , k) * B(k, j+1);
            t10 += A(i+1, k) * B(k, j  );
            t11 += A(i+1, k) * B(k, j+1);
      }
      C(i,  j) = t00;  C(i,  j+1) = t01;
      C(i+1,j) = t10;  C(i+1,j+1) = t11;
    }
  }
}
```

Hoist 2x2 tile

Tiling and Hoisting



Improving Locality: Cache

- Large matrix problems won't fit completely into cache
- Use blocked algorithm – work with blocks that will fit into cache

$$C_{IJ} = \sum_K A_{IK} B_{KJ}$$

$$\begin{bmatrix} C_{00} & C_{01} & C_{02} & C_{03} \\ C_{10} & C_{11} & C_{12} & C_{13} \\ C_{20} & C_{21} & C_{22} & C_{23} \\ C_{30} & C_{31} & C_{32} & C_{33} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix} \times \begin{bmatrix} B_{00} & B_{01} & B_{02} & B_{03} \\ B_{10} & B_{11} & B_{12} & B_{13} \\ B_{20} & B_{21} & B_{22} & B_{23} \\ B_{30} & B_{31} & B_{32} & B_{33} \end{bmatrix}$$

$$C_{21} = A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21} + A_{23}B_{31}$$

- Each product term fits completely into cache and runs at high-performance
- Cache misses amortized $O(N^3)$ work with $O(N^2)$ data

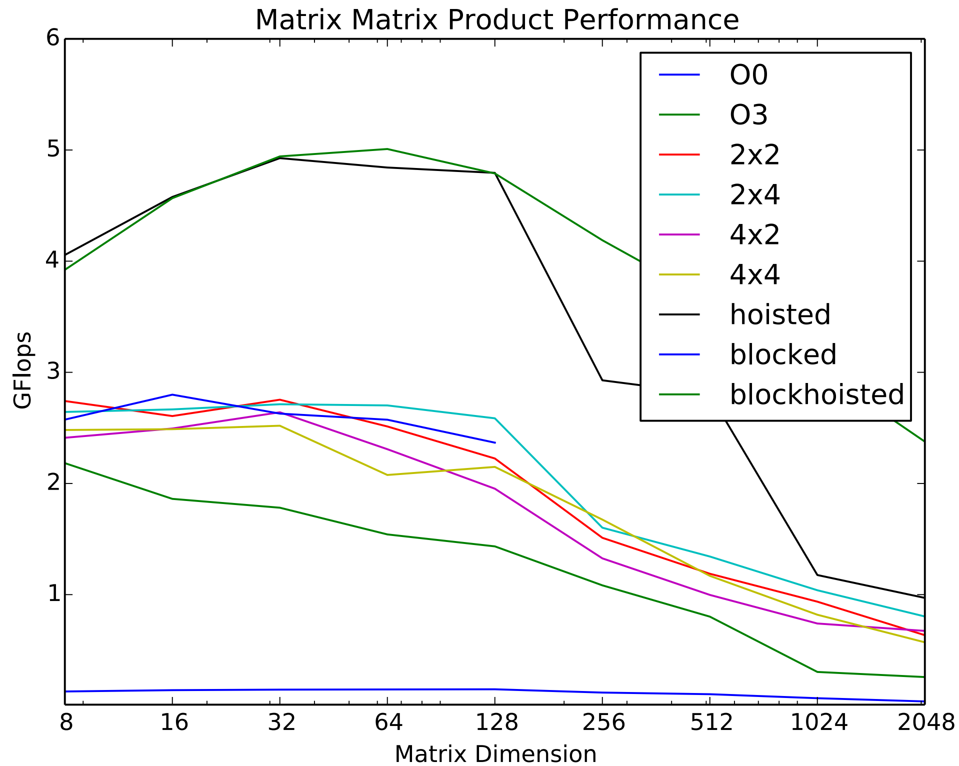
Blocking and Tiling

```
void blockedTiledMultiply2x2(const Matrix& A, const Matrix&B, Matrix&C) {
  const int blocksize = std::min(A.num_rows(), 32);

  for (size_t ii = 0; ii < A.num_rows(); ii += blocksize) {
    for (size_t jj = 0; jj < B.num_cols(); jj += blocksize) {
      for (size_t kk = 0; kk < A.num_cols(); kk += blocksize) {

        for (size_t i = ii; i < ii+blocksize; i += 2) {
          for (size_t j = jj; j < jj+blocksize; j += 2) {
            for (size_t k = kk; k < kk+blocksize; ++k) {
              C(i  , j  ) += A(i  , k) * B(k, j  );
              C(i  , j+1) += A(i  , k) * B(k, j+1);
              C(i+1, j  ) += A(i+1, k) * B(k, j  );
              C(i+1, j+1) += A(i+1, k) * B(k, j+1);
            }
          }
        }
      }
    }
  }
}
```

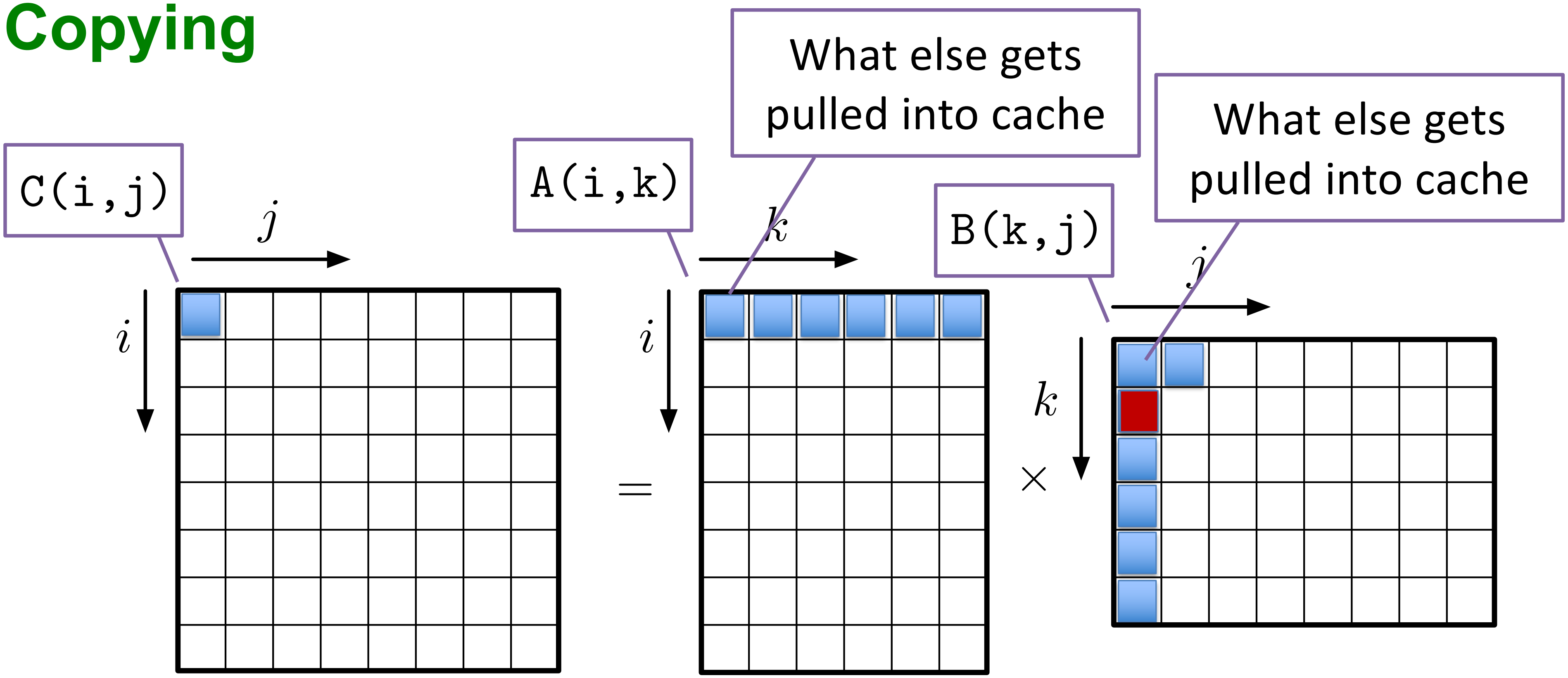
Outer loops work across blocks (for each block)

Inner loops work on blocks

Blocking and Tiling and Hoisting

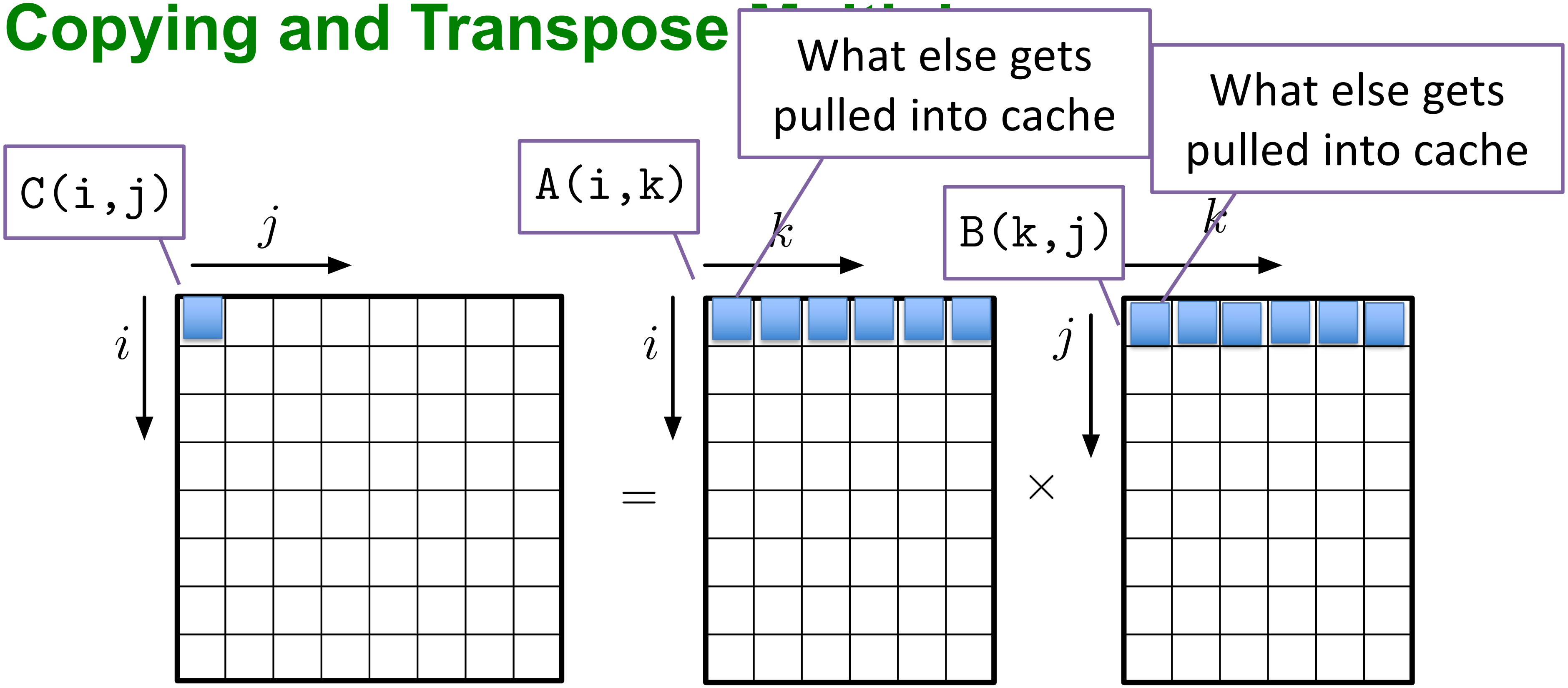


Copying

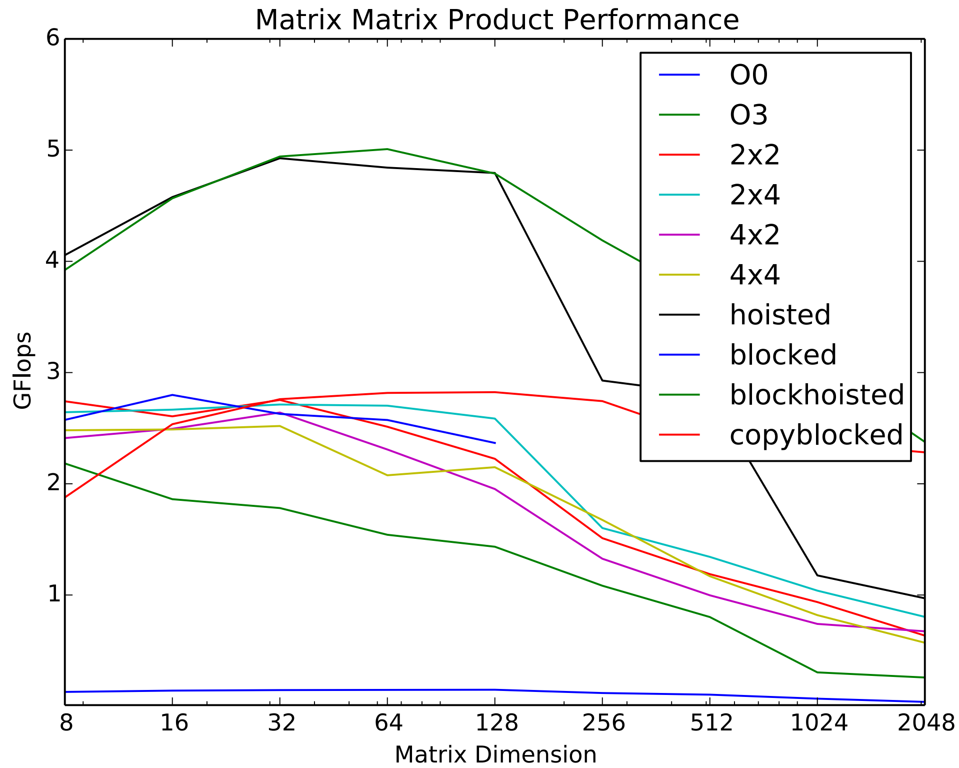


C(i,j)
A(i,k)
What else gets pulled into cache
B(k,j)
What else gets pulled into cache
j
i
k
i
j
k

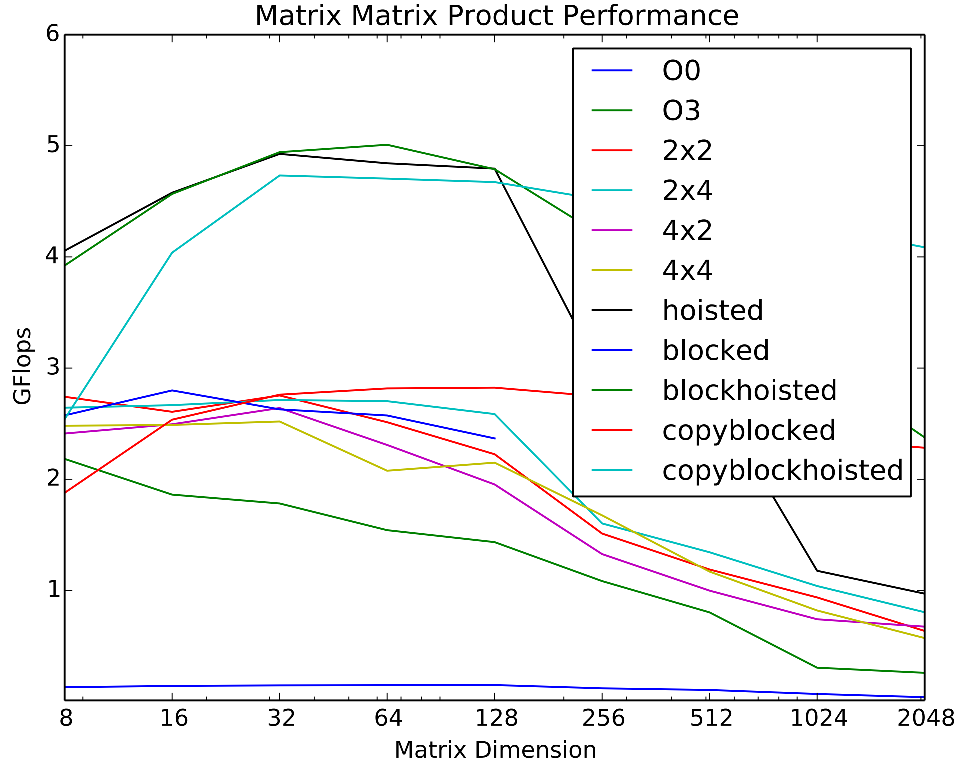
Copying and Transpose



Copying and Blocking and Tiling



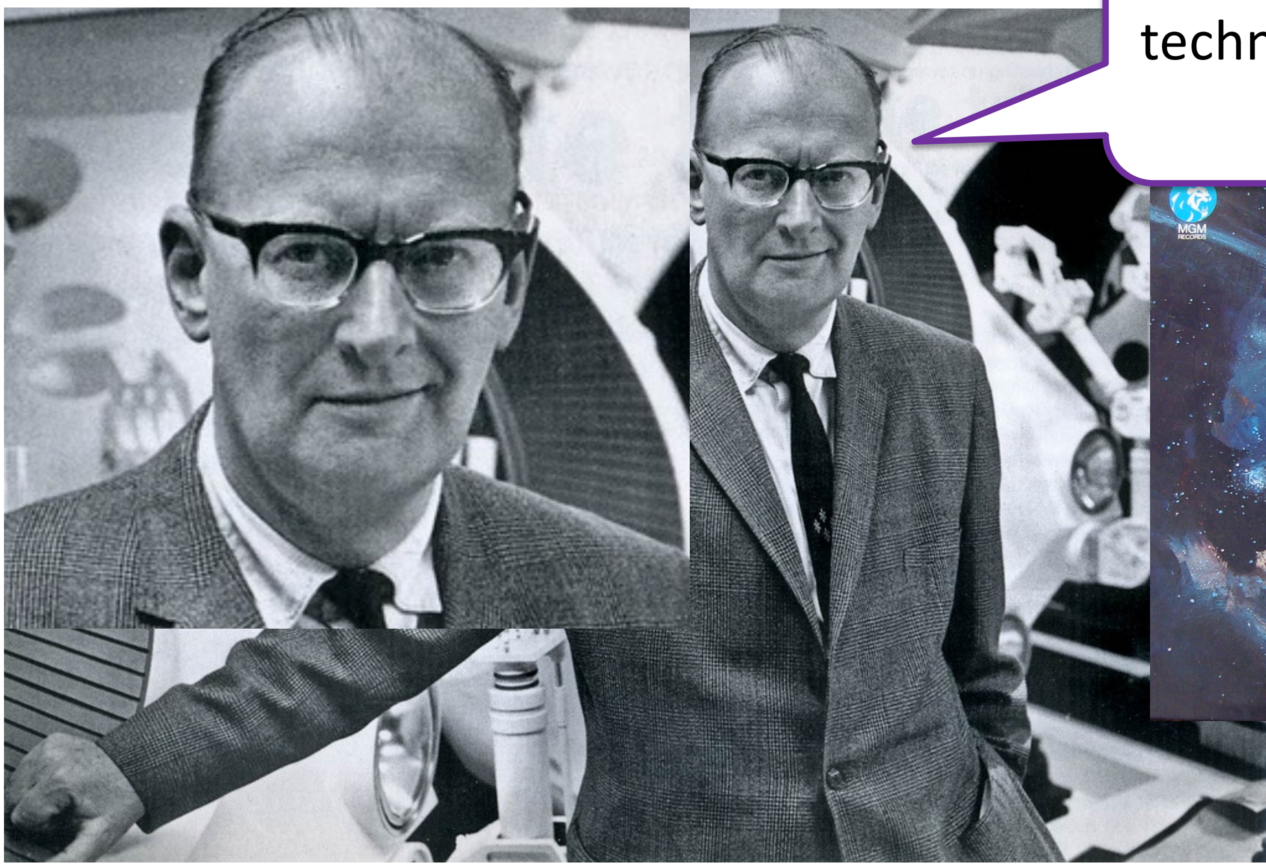
Blocking and Tiling and Hoisting and Copying



Recap

- Locality: Write software so hardware can leverage it
- Register locality (tiling / unroll and jam)
- Hoisting
- Blocking
- Copying / transpose multiply
- Always use –O3 for release (not for debug)

Name This Famous Person



Any sufficiently advanced technology is indistinguishable from magic

This Nearly Famous Person Says

Optimizing compilers are sufficiently advanced technology

And so are modern microprocessors

But especially optimizing compilers for modern microprocessors

Magic: the power of apparently influencing the course of events by using mysterious or supernatural forces

Tuning

- Starting with base code
- Various compiler optimizations help
- Tiling (which size)
- Blocking (what size)
- What size works best for Tiling and Blocking ***together?***

- What loop ordering? Matrix matrix product has six different orderings? What block ordering?

- What about when we add AVX, and threads, etc?

How do we find the optimal combination?

Magic: the power of apparently influencing the course of events by using mysterious or supernatural forces

The answer will be different for different CPUs

Finding the Sweet Spot

- Exhaustive parameter space search
  - Tiling, Blocking, Compiler flags, AVX inst, loop ordering
- Original project at UC Berkeley phiPAC (Bilmes et al)
- Further developed by Whaley and Dongarra → Automatically Tuned Linear Algebra Subprograms (ATLAS)
  - Recently honored with "test of time" award

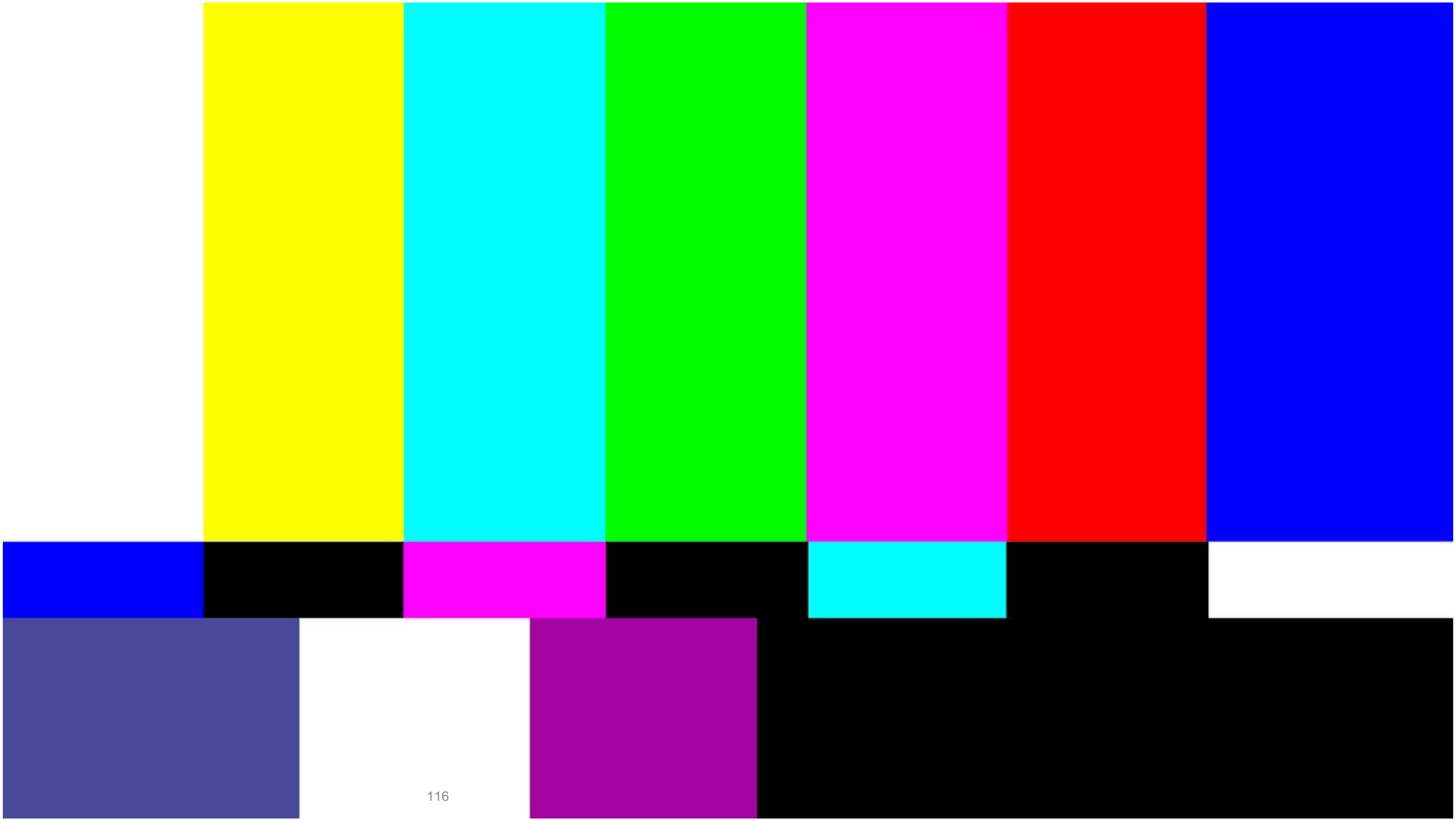
And wrote a program to generate different multiply functions

This started as a final course project

The competition was to write fastest matrix-matrix product

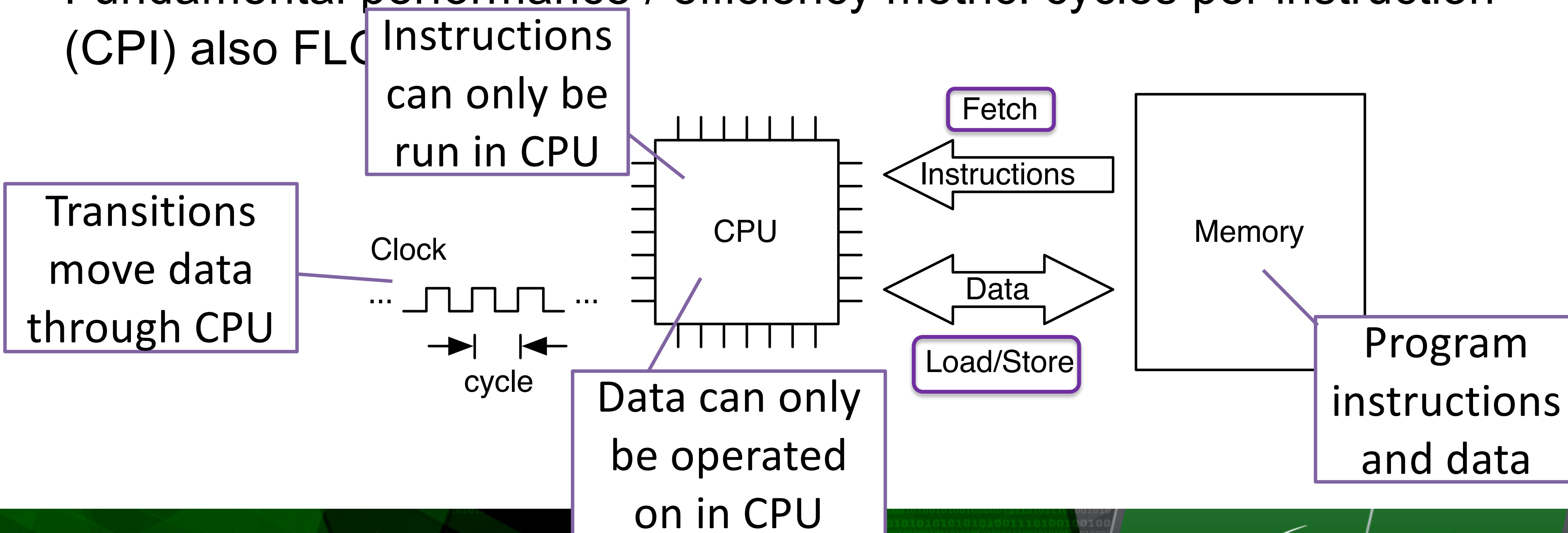
Students were the good kind of lazy

Thank you!



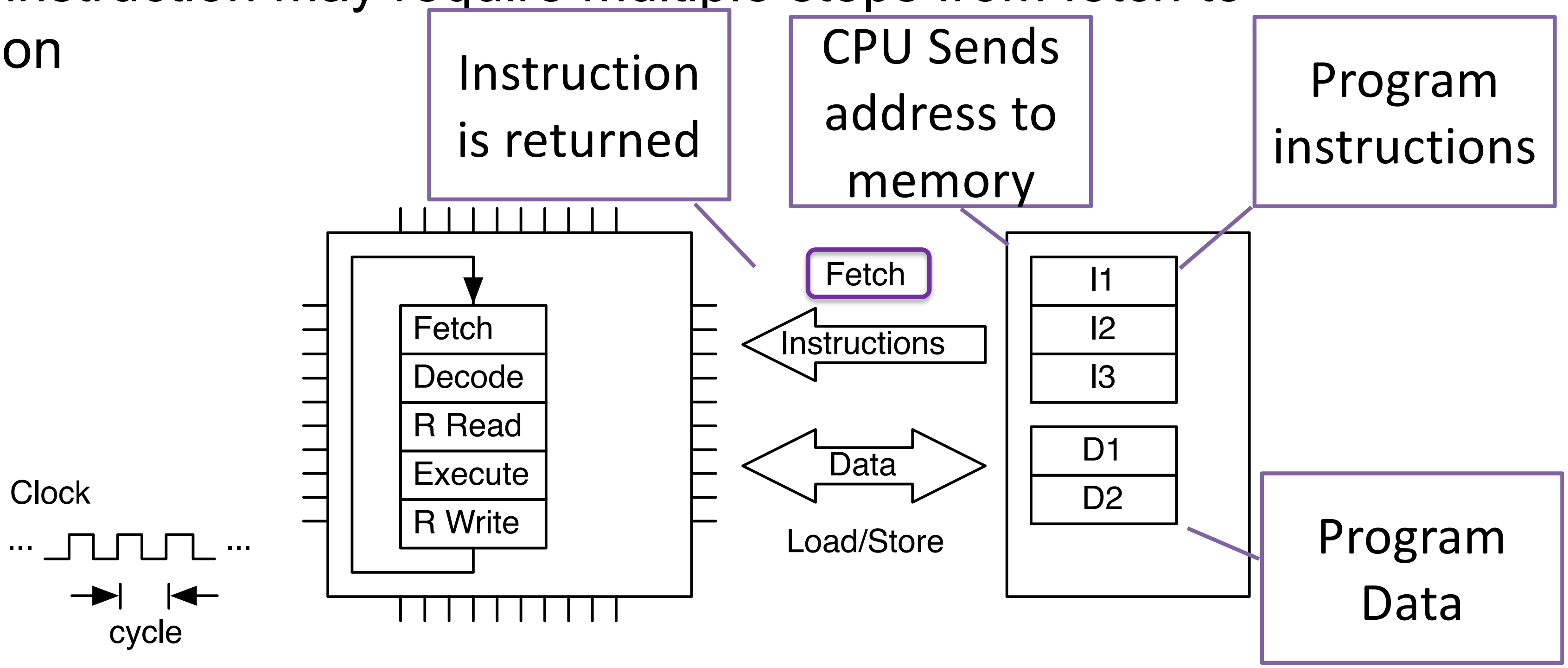
Microprocessors

- Basic operation: read and execute program instructions stored in memory
- Fundamental performance / efficiency metric: cycles per instruction (CPI) also FL



Processor Core Instruction Handling

- A single instruction may require multiple steps from fetch to completion



Processor Core Instruction Handling

- By pipelining, multiple instructions can be executed at each clock cycle
- Form of instruction-level parallelism