Parallel Architectures

Chris Rossbach

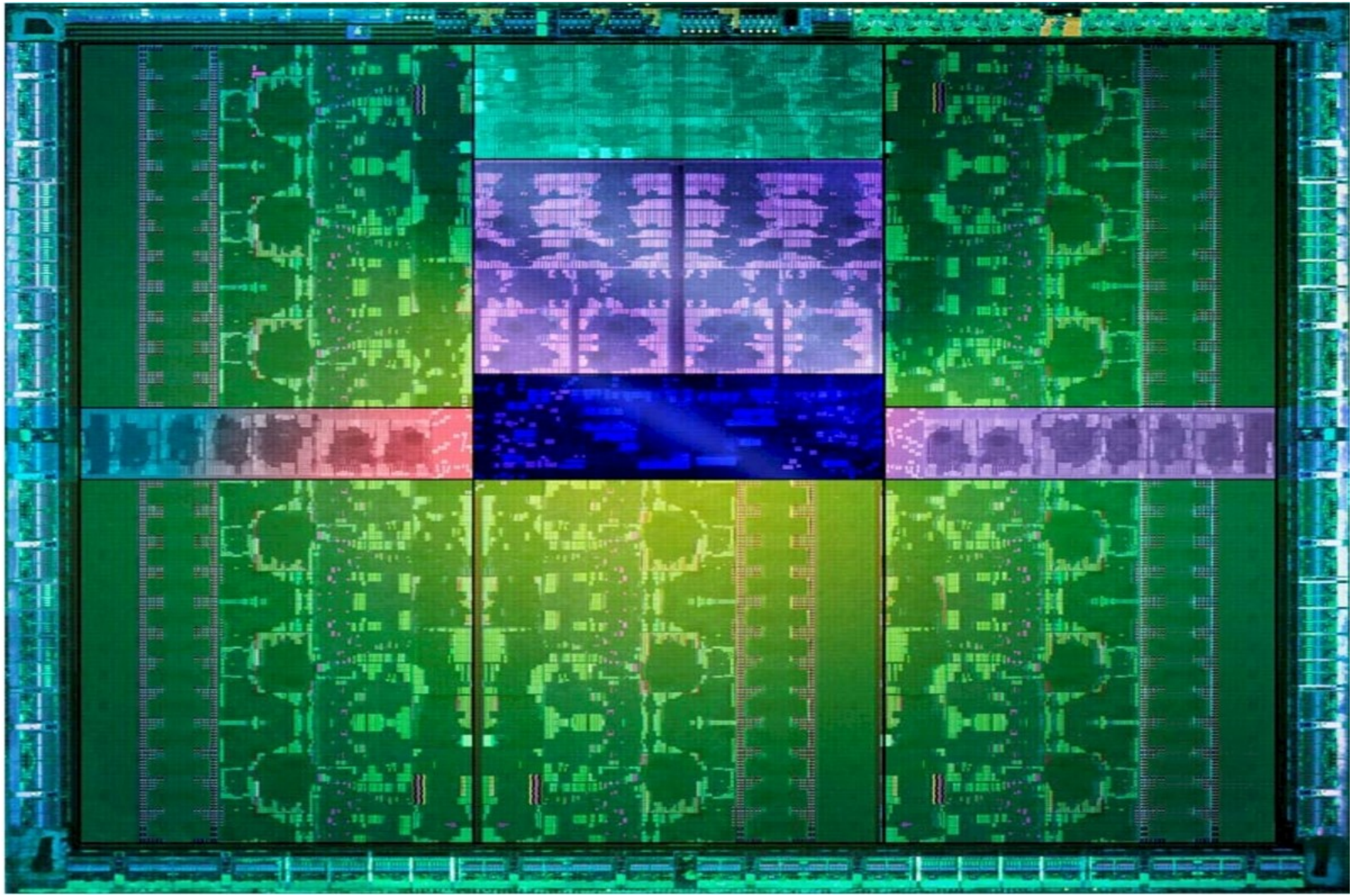
Outline for Today

- Questions?
- Administrivia

- Agenda
  - Parallel Architectures (GPU background)

Faux Quiz questions

- What is hardware multi-threading; what problem does it solve?
- What is the difference between a vector processor and a scalar?
- Implement a parallel scan or reduction
- How are GPU workloads different from GPGPU workloads?
- How does SIMD differ from SIMT?
- List and describe some pros and cons of vector/SIMD architectures.
- GPUs historically have elided cache coherence.Why? What impact does it have on the the programmer?
- List some ways that GPUs use concurrency but not necessarily parallelism.



A modern GPU: Volta V100

A modern GPU: Volta V100

- 80 SMs
  - Streaming Multiprocessor

A modern GPU: Volta V100

Also:
CU or *ACE*

- 80 SMs
  - Streaming Multiprocessor

A modern GPU: Volta V100

- 80 SMs
  - Streaming Multiprocessor

A modern GPU

- 80 SMs
  - Streaming Multiprocessor
  - 64 cores/SM
  - 5210 threads!
  - 15.7 TFLOPS

A modern GPU

- 80 SMs
  - Streaming Multiprocessor
  - 64 cores/SM
  - 5210 threads!
  - 15.7 TFLOPS

Roughly: all of pfxsum 1,000s X/sec

A modern GPU

- 80 SMs
  - Streaming Multiprocessor
  - 64 cores/SM
  - 5210 threads!
  - 15.7 TFLOPS

A modern GPU

- 80 SMs
  - Streaming Multiprocessor
  - 64 cores/SM
  - 5210 threads!
  - 15.7 TFLOPS
- 640 Tensor cores

A modern GPU

- 80 SMs
  - Streaming Multiprocessor
  - 64 cores/SM
  - 5210 threads!
  - 15.7 TFLOPS
- 640 Tensor cores
- HBM2 memory
  - 4096-bit bus
  - No cache coherence!

A modern GPU

- 80 SMs
  - Streaming Multiprocessor
  - 64 cores/SM
  - 5210 threads!
  - 15.7 TFLOPS
- 640 Tensor cores
- HBM2 memory
  - 4096-bit bus
  - No cache coherence!
- 16 GB memory
  - PCIe-attached

A modern GPU

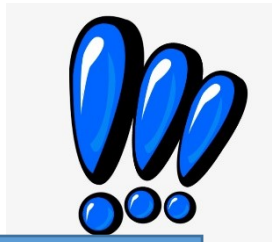
- 80 SMs
  - Streaming Multiprocessor
  - 64 cores/SM
  - 5210 threads!
  - 15.7 TFLOPS
- 640 Tensor cores
- HBM2 memory
  - 4096-bit bus
  - No cache coherence!
- 16 GB memory
  - PCIe-attached

A modern GPU

- 80 SMs
  - Streaming Multiprocessor
  - 64 cores/SM
  - 5210 threads!
  - 15.7 TFLOPS
- 640 Tensor cores
- HBM2 memory
  - 4096-bit bus
  - No cache coherence!
- 16 GB memory
  - PCIe-attached

A modern GPU

- 80 SMs
  - Streaming Multiprocessor
  - 64 cores/SM
  - 5210 threads!
  - 15.7 TFLOPS
- 640 Tensor cores
- HBM2 memory
  - 4096-bit bus
  - No cache coherence!
- 16 GB memory
  - PCIe-attached



How do you program a machine like this? pthread_create()?

GPUs: Outline

- Background from many areas
  - Architecture
    - Vector processors
    - Hardware multi-threading
  - Graphics
    - Graphics pipeline
    - Graphics programming models
  - Algorithms
    - parallel architectures → parallel algorithms
- Programming GPUs
  - CUDA
  - Basics: getting something working
  - Advanced: making it perform

Architecture Review: Pipelines

Processor algorithm:

```
main() {
  while(true)
    do_next_instruction();
}
```

Architecture Review: Pipelines

Processor algorithm:

```
main() {
  while(true)
    do_next_instruction();
}
```

```
do_next_instruction() {
  instruction = fetch();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

Architecture Review: Pipelines

Processor algorithm:

```
main() {
  v
}

do_n
  inst
  ops
  exe
  access_memory(ops, regs);
  write_back(regs);
}
```

```
main() {
  pthread_create(do_instructions);
  pthread_create(do_decode);
  pthread_create(do_execute);
  …
  pthread_join(…);
  …
}
```

Architecture Review: Pipelines

Processor algorithm:

```
main() {
  v
  }

do_n
  inst
  ops
  exe
  access_memory(ops, regs);
  write_back(regs);
}
```

```
main() {
  pthread_create(do_instructions);
  pthread_create(do_decode);
  pthread_create(do_execute);
  …
  pthread_join(…);
  …
}
```

```
do_instructions() {
  while(true) {
    instruction = fetch();
    enqueue(DECODE, instruction);
}}
```

```
do_decode() {
  while(true) {
    instruction = dequeue();
    ops, regs = decode(instruction);
    enqueue(EX, instruction);
}}
```

```
do_execute() {
  while(true) {
    instruction = dequeue();
    execute_calc_addrs(ops, regs);
    enqueue(MEM, instruction);
}}
```

….

Architecture Review: Pipelines

Processor algorithm:

```
main() {
  while(true) {
    do_next_instruction();
}
```

Architecture Review: Pipelines

Processor algorithm:

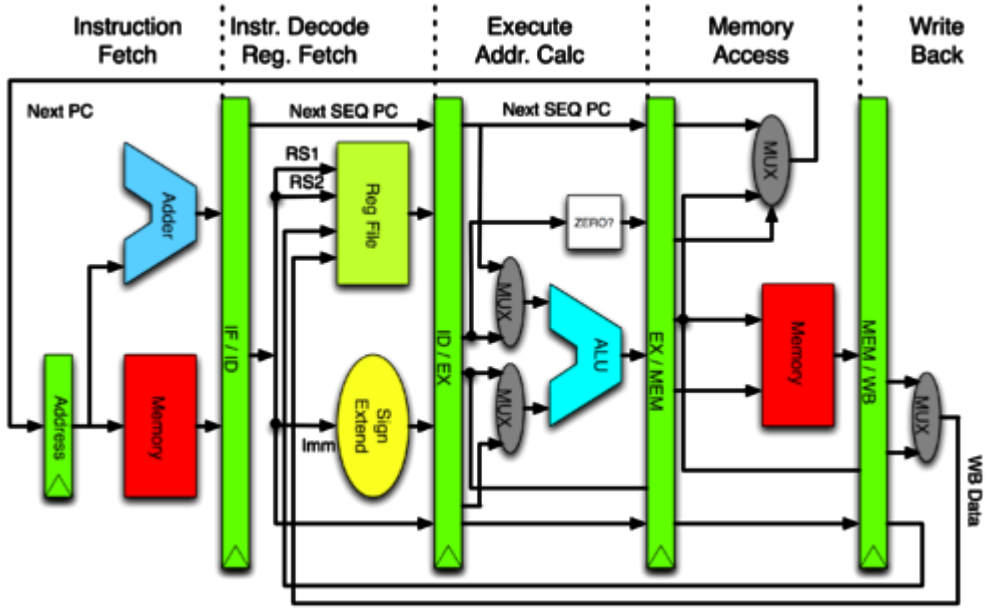
```
main() {
  while(true) {
    do_next_instruction();
}
```

```
do_next_instruction() {
  instruction = fetch();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

Architecture Review: Pipelines

Processor algorithm:

```
main() {
  while(true) {
    do_next_instruction();
  }
```

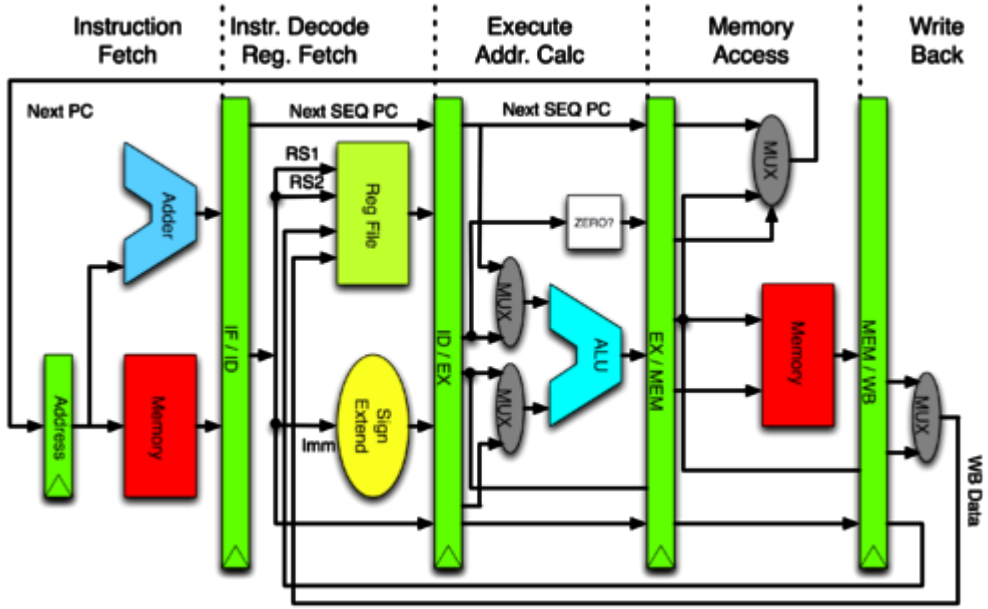



```
do_next_instruction() {
  instruction = fetch();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

Architecture Review: Pipelines

Processor algorithm:

```
main() {
  while(true) {
    do_next_instruction();
  }
```



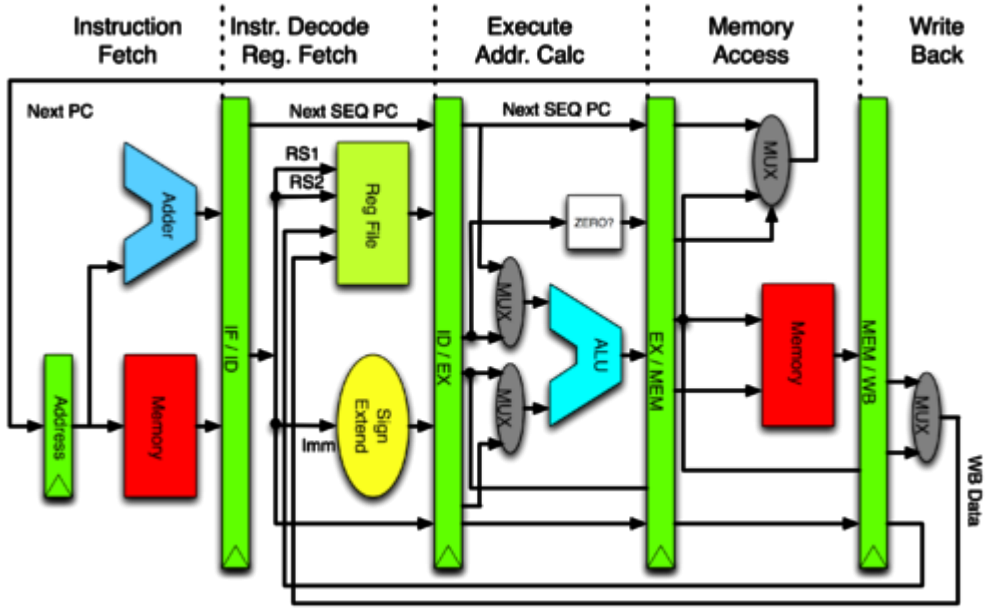

```
do_next_instruction() {
  instruction = fetch();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

Instr No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Architecture Review: Pipelines

Processor algorithm:

```
main() {
  while(true) {
    do_next_instruction();
  }
```




```
do_next_instruction() {
  instruction = fetch();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

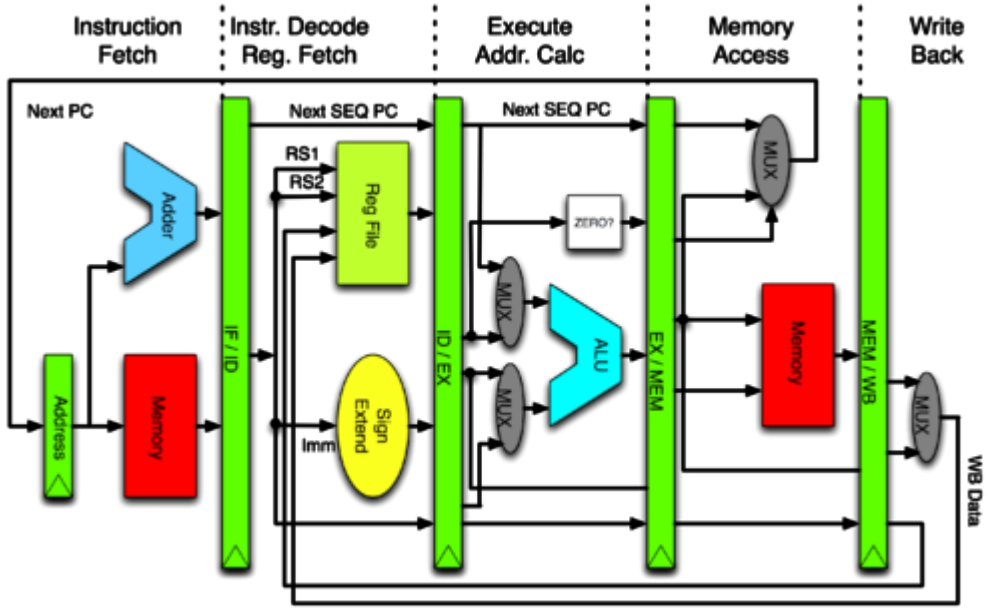
Instr No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

What is the name of this kind of parallelism?

Architecture Review: Pipelines

Processor algorithm:

```
main() {
  while(true) {
    do_next_instruction();
  }
```




```
do_next_instruction() {
  instruction = fetch();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

Instr No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM

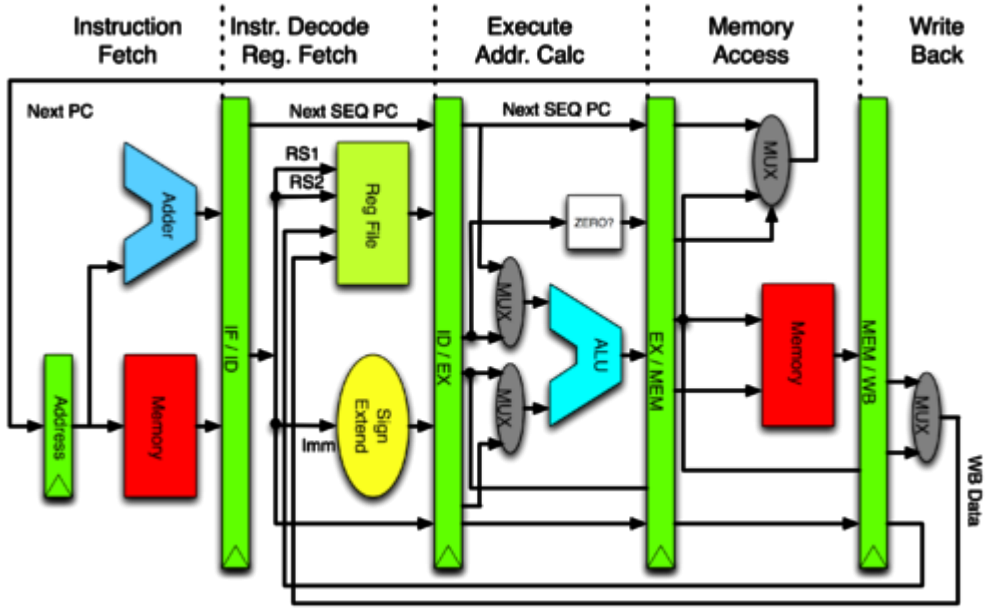
Works well if pipeline is kept full
What kinds of things cause "bubbles"/stalls?

What is the name of this kind of parallelism?

Architecture Review: Pipelines

Processor algorithm:

```
main() {
  while(true) {
    do_next_instruction();
  }
```



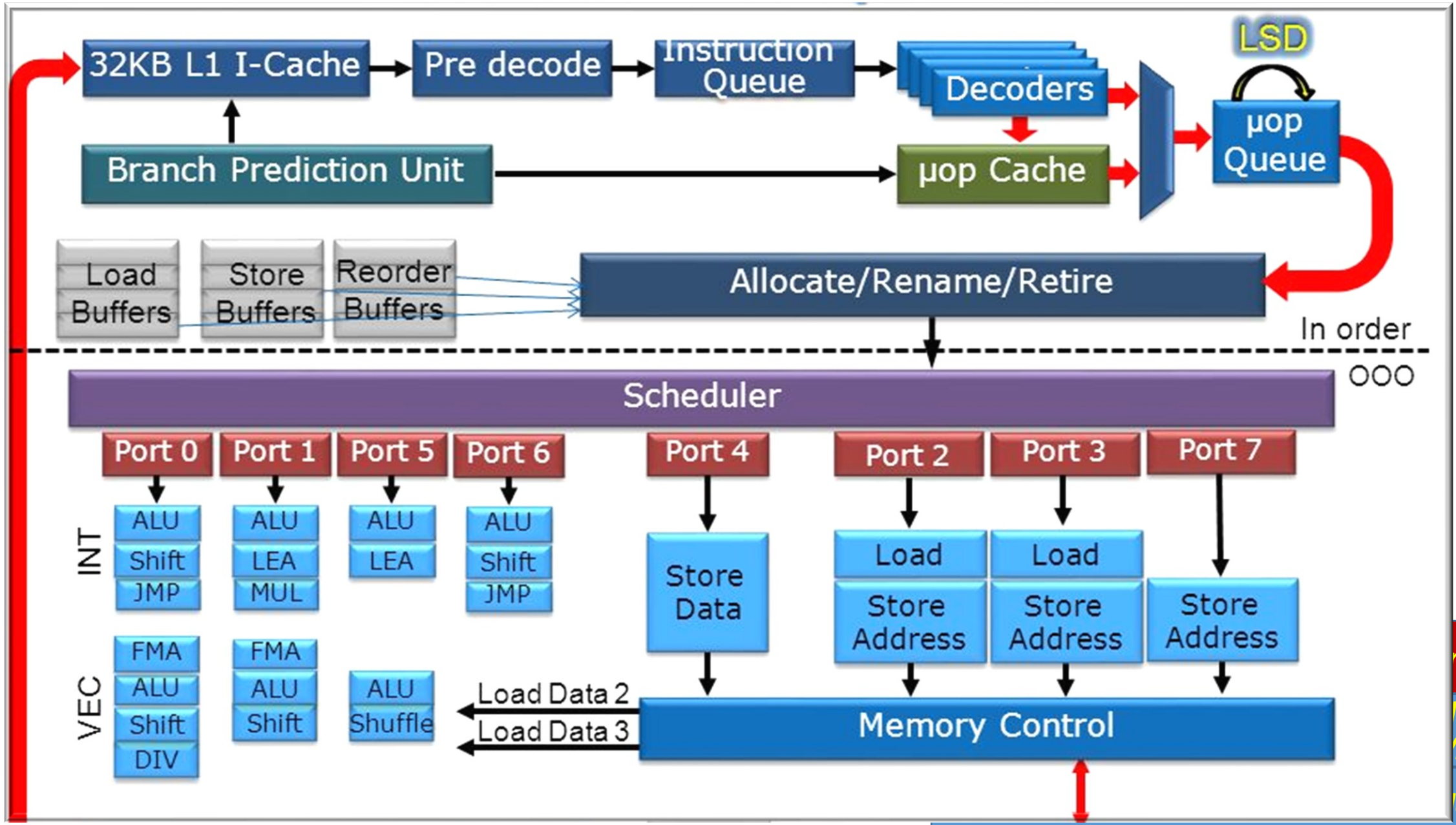

```
do_next_instruction() {
  instruction = fetch();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

Instr No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	

How can we get *more* parallelism?

Works well if pipeline is kept full
What kinds of things cause "bubbles"/stalls?

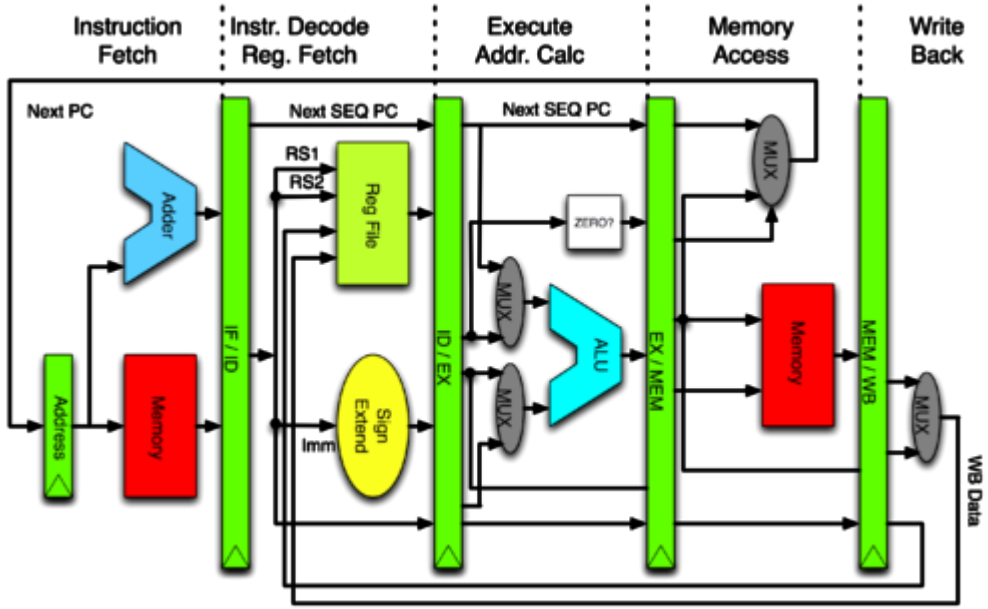
What is the name of this kind of parallelism?


32KB L1 I-Cache
Pre decode
Instruction Queue
Decoders
LSD
µop Queue
Branch Prediction Unit
µop Cache
Load Buffers
Store Buffers
Reorder Buffers
Allocate/Rename/Retire
In order
OOO
Scheduler
Port 0
Port 1
Port 5
Port 6
Port 4
Port 2
Port 3
Port 7
INT
ALU
Shift
JMP
ALU
LEA
MUL
ALU
LEA
ALU
Shift
JMP
Store Data
Load
Store Address
Load
Store Address
Store Address
VEC
FMA
ALU
Shift
DIV
FMA
ALU
Shift
ALU
Shuffle
Load Data 2
Load Data 3
Memory Control

Architecture Review: Pipelines

Processor algorithm:

```
main() {
  while(true) {
    do_next_instruction();
  }
```




```
do_next_instruction() {
  instruction = fetch();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

Instr No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	

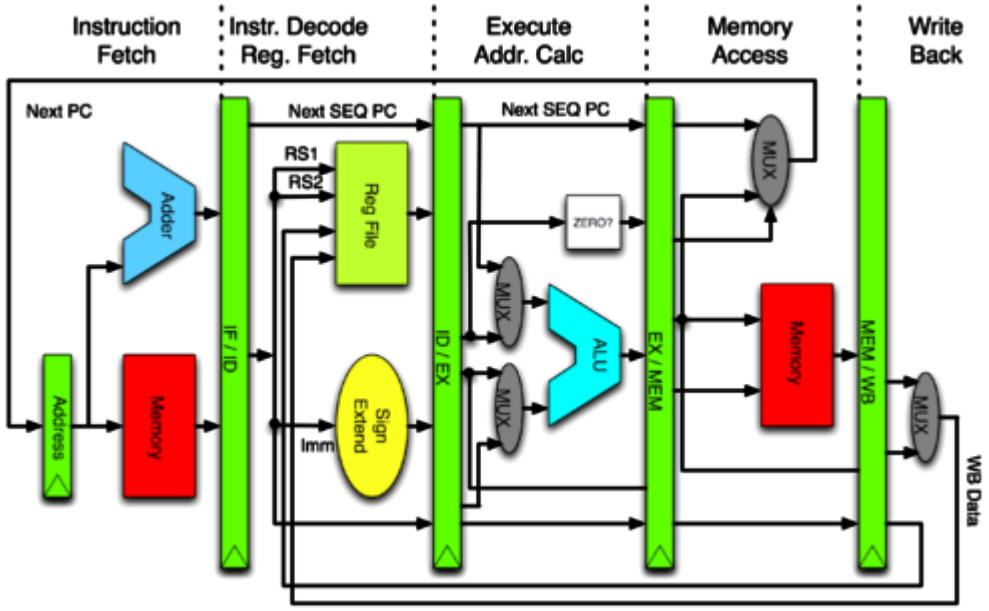
How can we get *more* parallelism?

Works well if pipeline is kept full
What kinds of things cause "bubbles"/stalls?

What is the name of this kind of parallelism?

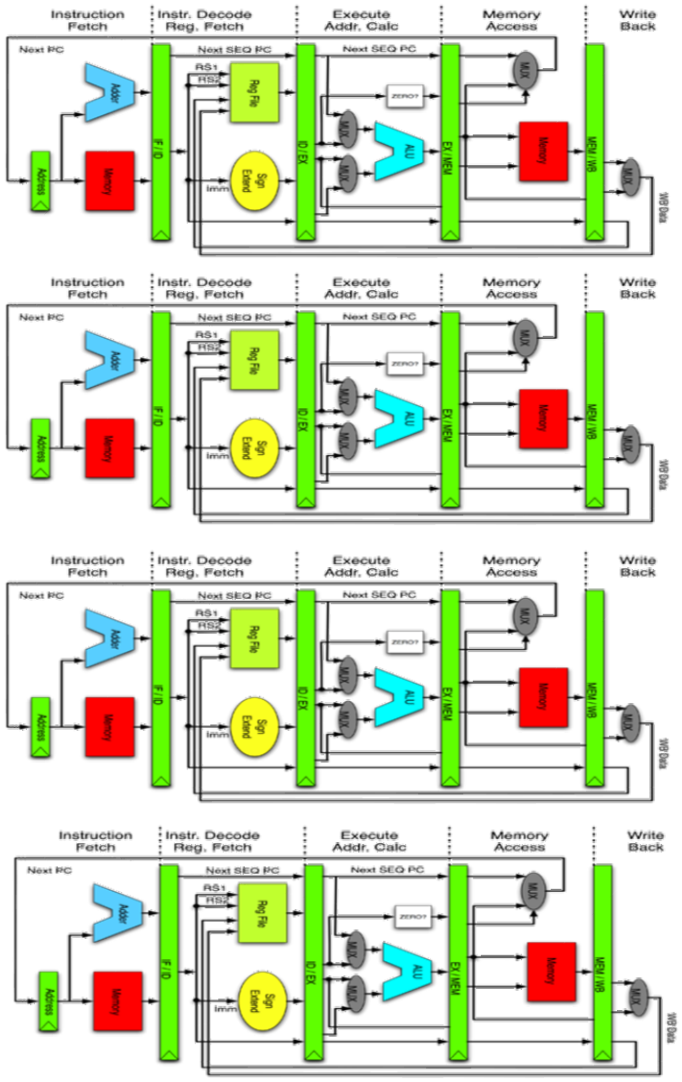
Multi-core/SMPs

Multi-core/SMPs

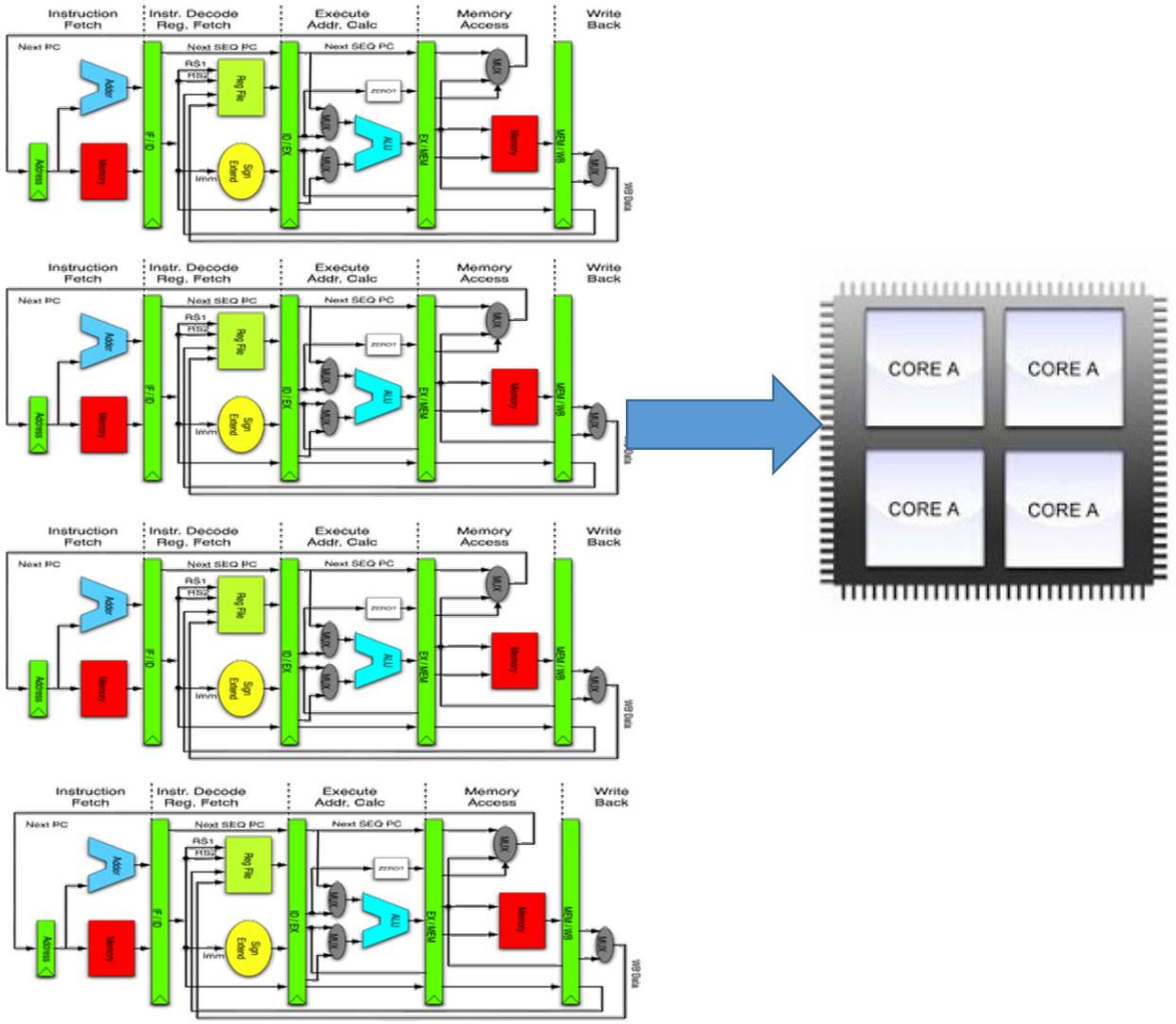


Multi-core/SMPs

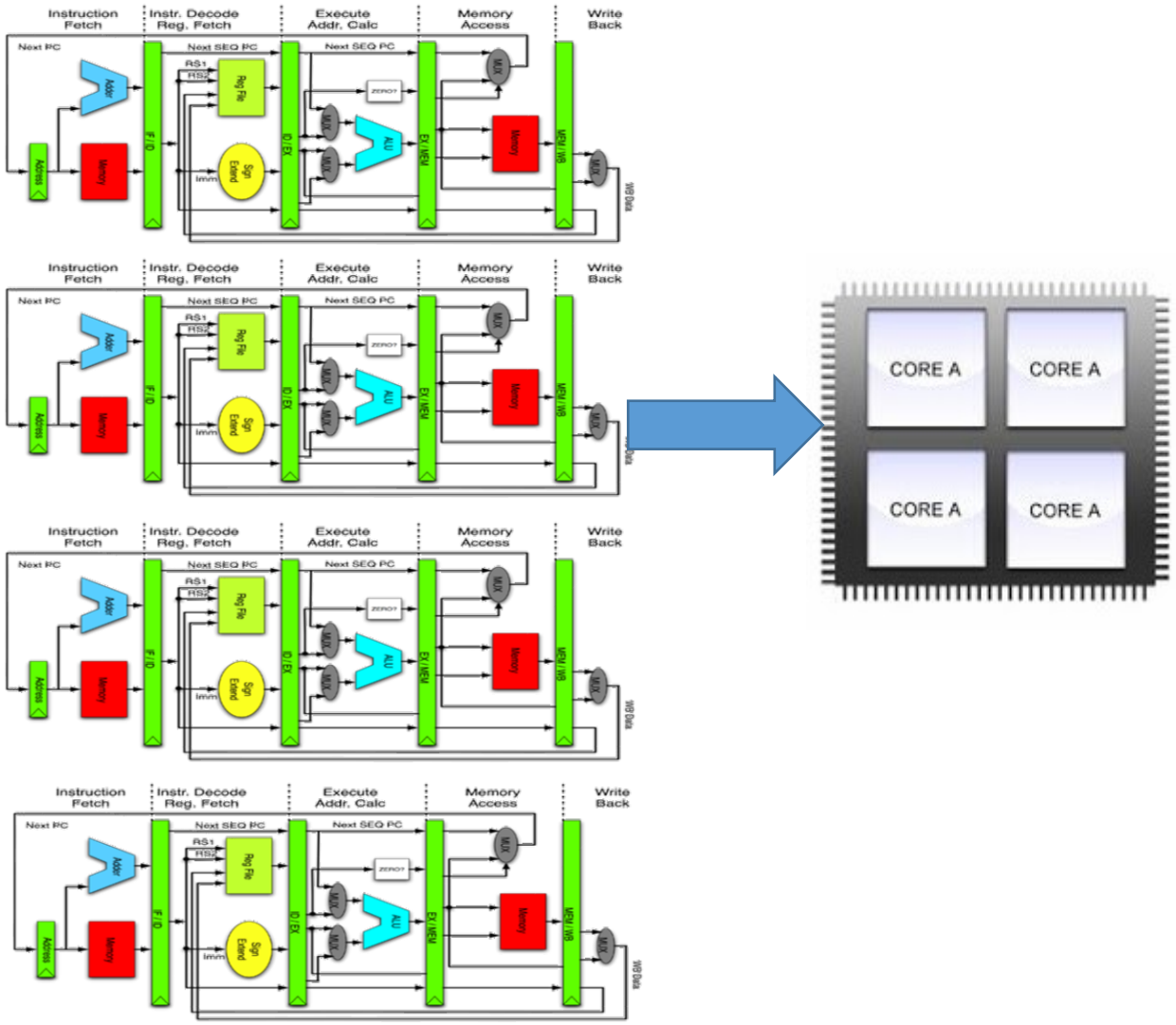
Multi-core/SMPs



Multi-core/SMPs



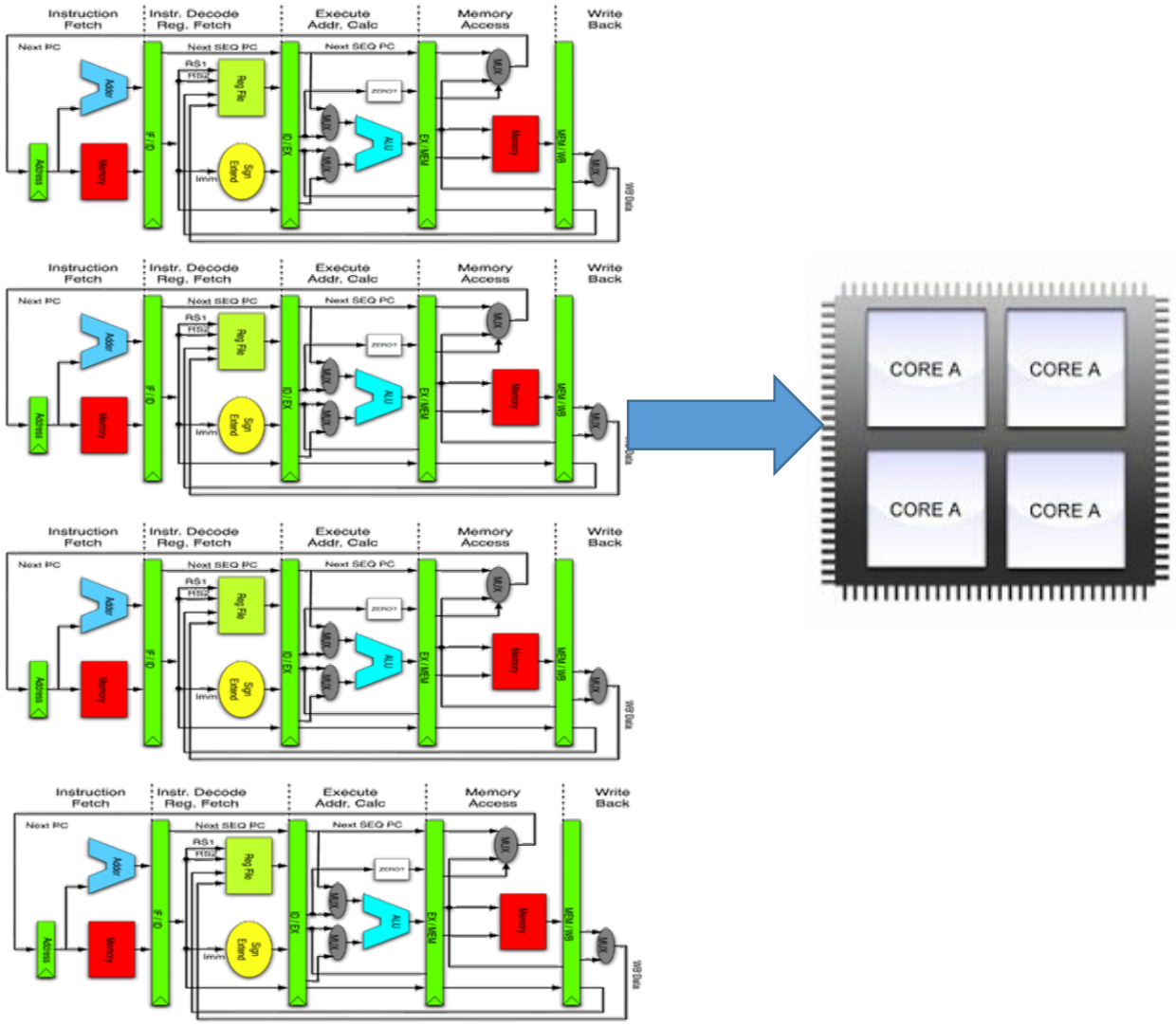
Multi-core/SMPs




```
main() {
  for(i=0; i<CORES; i++) {
    pthread_create(
       do_instructions());
  }
}
do_instructions() {
  while(true) {
    instruction = fetch();
    ops, regs = decode(instruction);
    execute_calc_addrs(ops, regs);
    access_memory(ops, regs);
    write_back(regs);
}}
```

- ***Pros: Simple***
- ***Cons: programmer has to find the parallelism!***

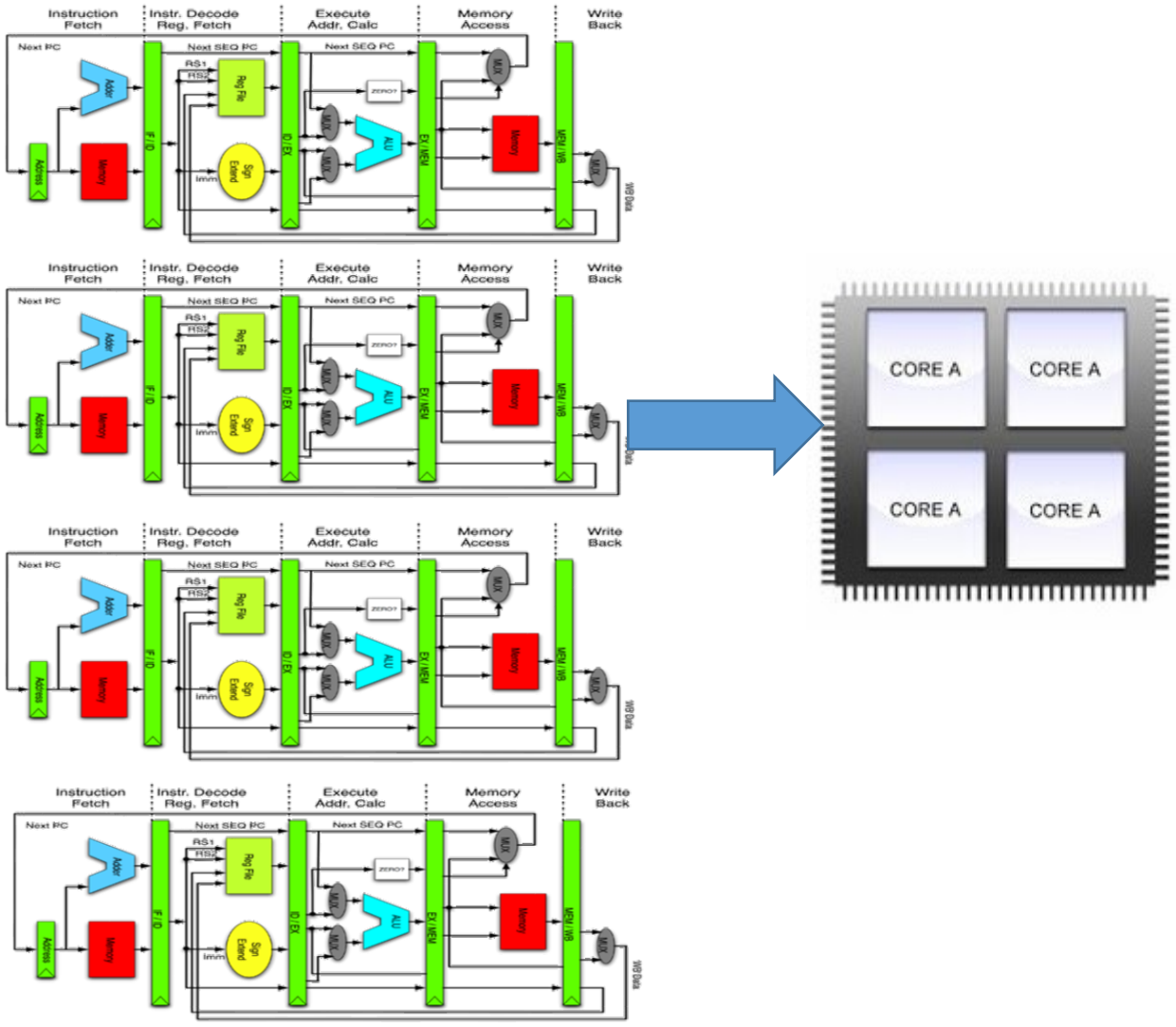
Multi-core/SMPs




```
main() {
  for(i=0; i<CORES; i++) {
    pthread_create(
       do_instructions());
  }
}
do_instructions() {
  while(true) {
    instruction = fetch();
    ops, regs = decode(instruction);
    execute_calc_addrs(ops, regs);
    access_memory(ops, regs);
    write_back(regs);
}}
```

- *Pros: Simple*
- *Cons: programmer has to find the parallelism!*

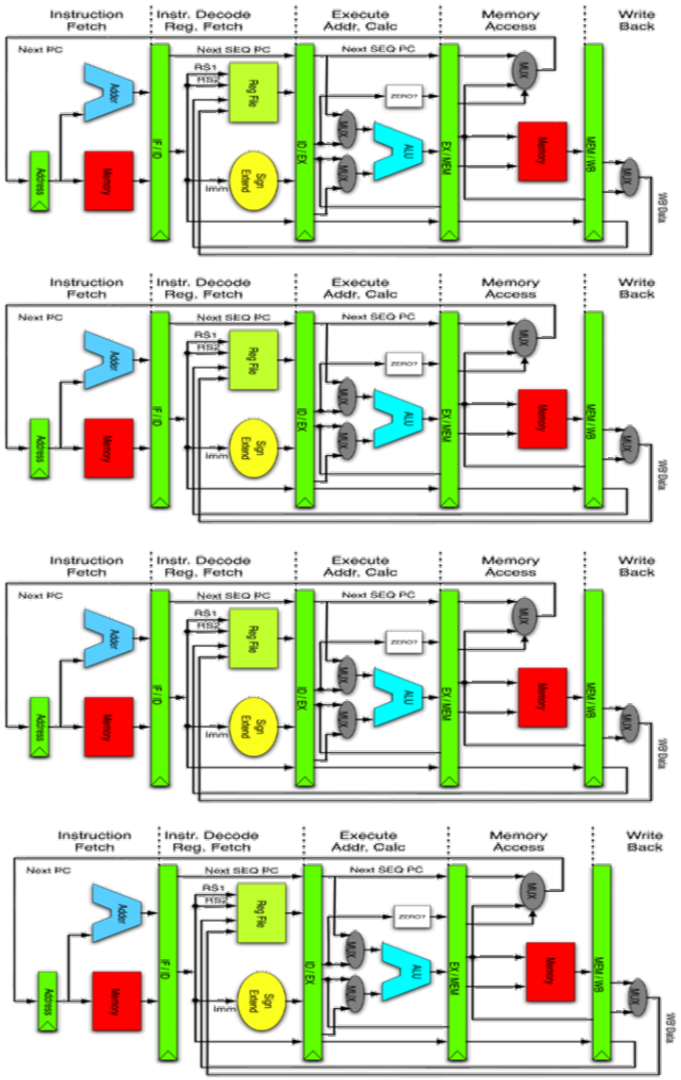
Multi-core/SMPs




```
main() {
  for(i=0; i<CORES; i++) {
    pthread_create(
      do_instructions());
  }
}
do_instructions() {
  while(true) {
    instruction = fetch();
    ops, regs = decode(instruction);
    execute_calc_addrs(ops, regs);
    access_memory(ops, regs);
    write_back(regs);
  }}
```

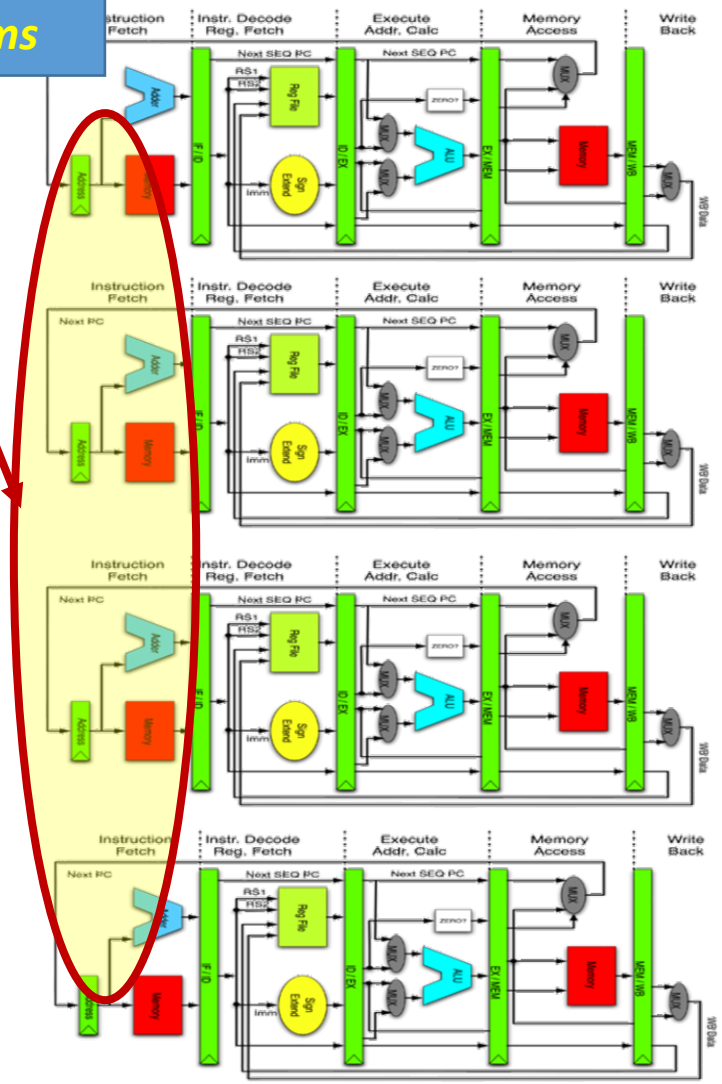
Other techniques extract parallelism here, try to let the machine find parallelism

Superscalar processors



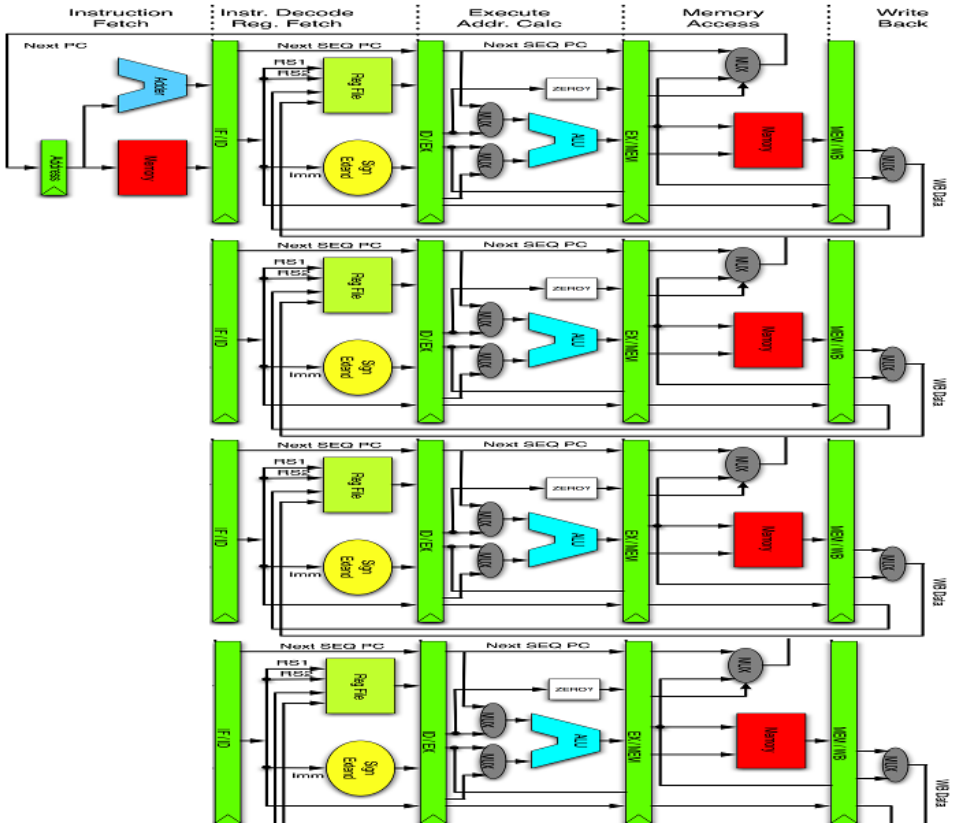
Superscalar processors

Remove extra instruction streams

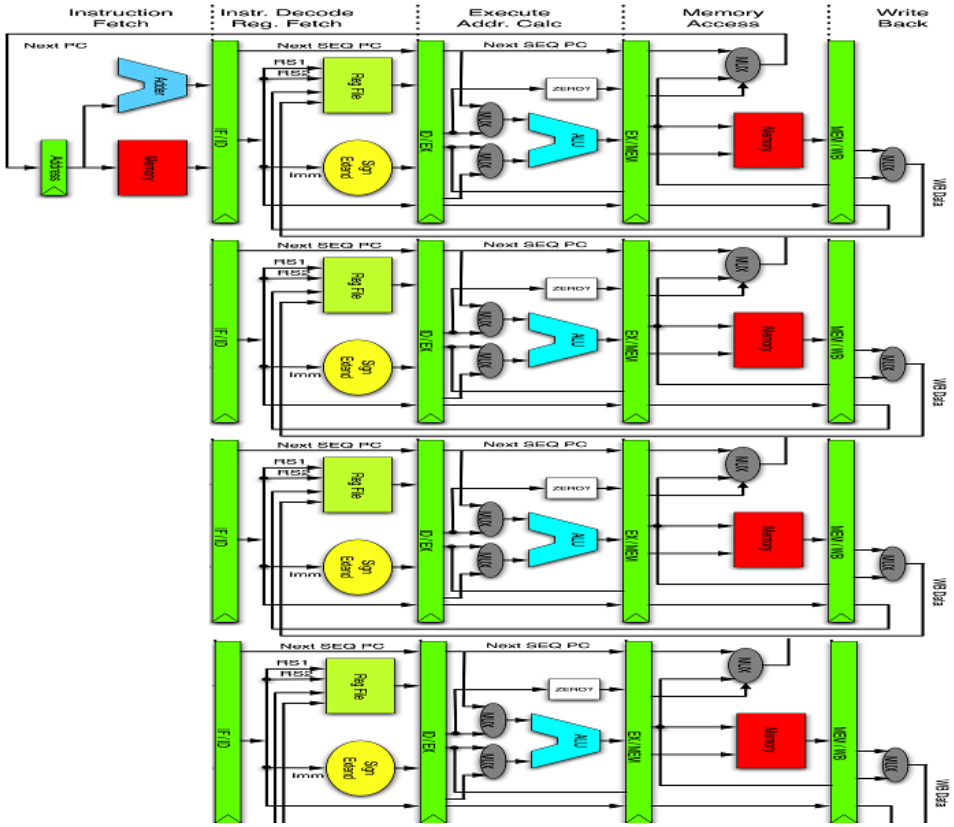


Superscalar processors

Superscalar processors



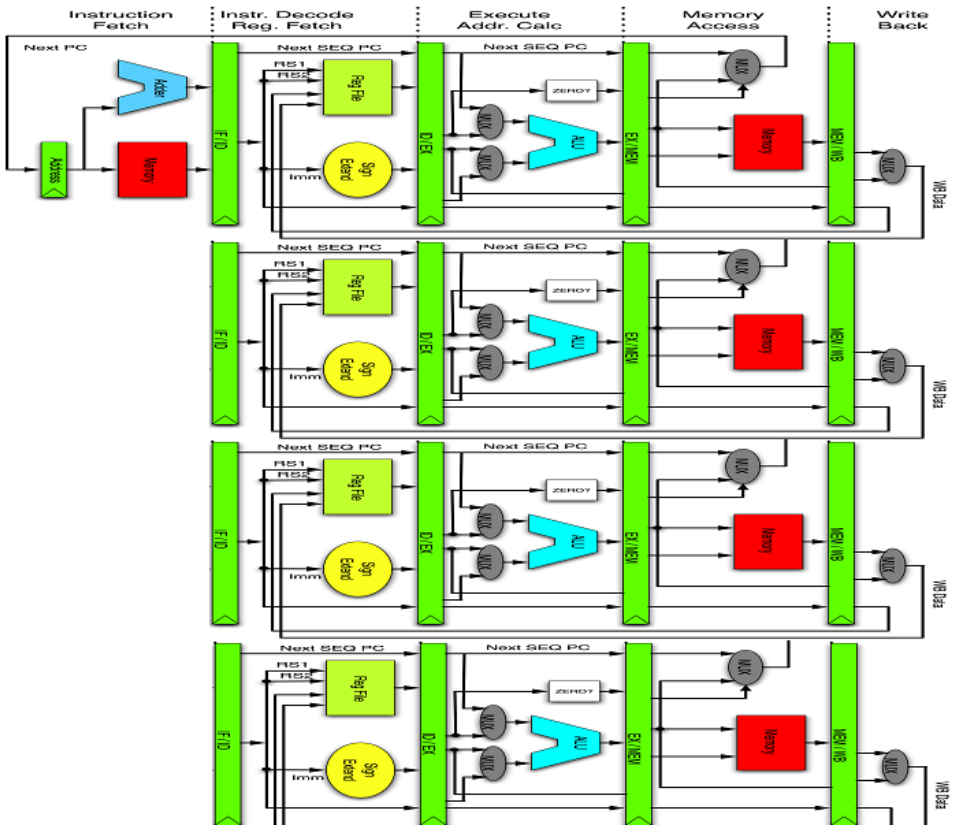
Superscalar processors




```
main() {
  for(i=0; i<CORES; i++)
    pthread_create(decode_exec);
    while(true) {
      instruction = fetch();
      enqueue(instruction);
    }
}

decode_exec() {
  instruction = dequeue();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

Superscalar processors

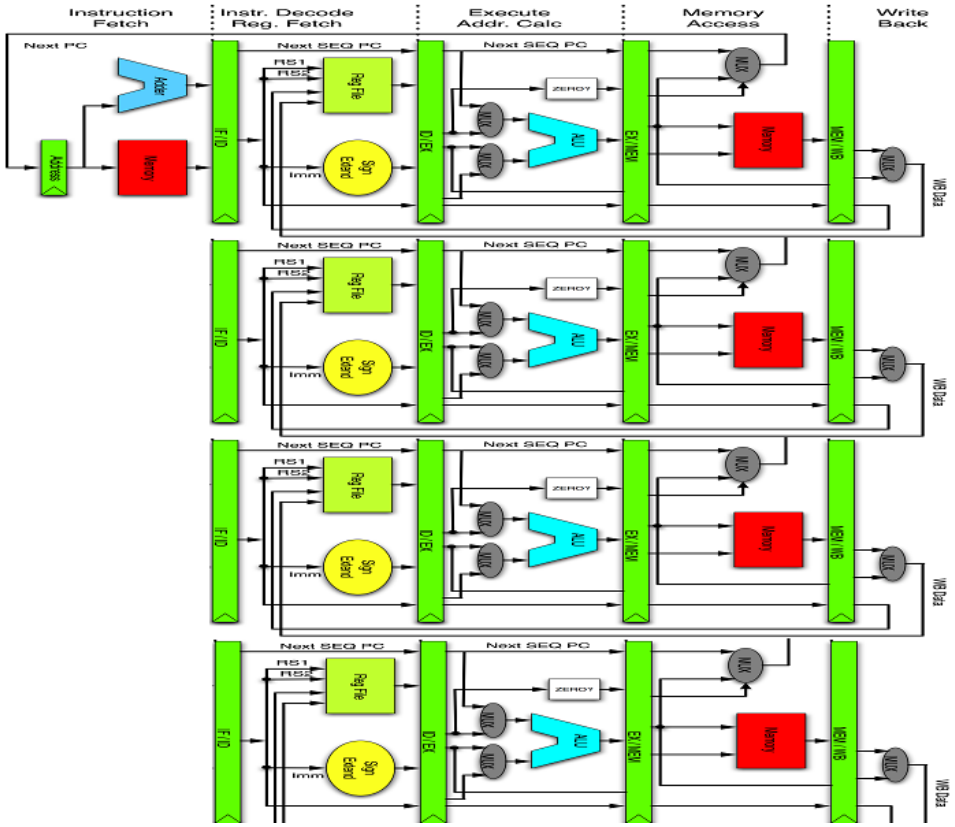



```
main() {
  for(i=0; i<CORES; i++)
    pthread_create(decode_exec);
    while(true) {
      instruction = fetch();
      enqueue(instruction);
    }
}

decode_exec() {
  instruction = dequeue();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

Doesn't look that different does it? Why do it?

Superscalar processors



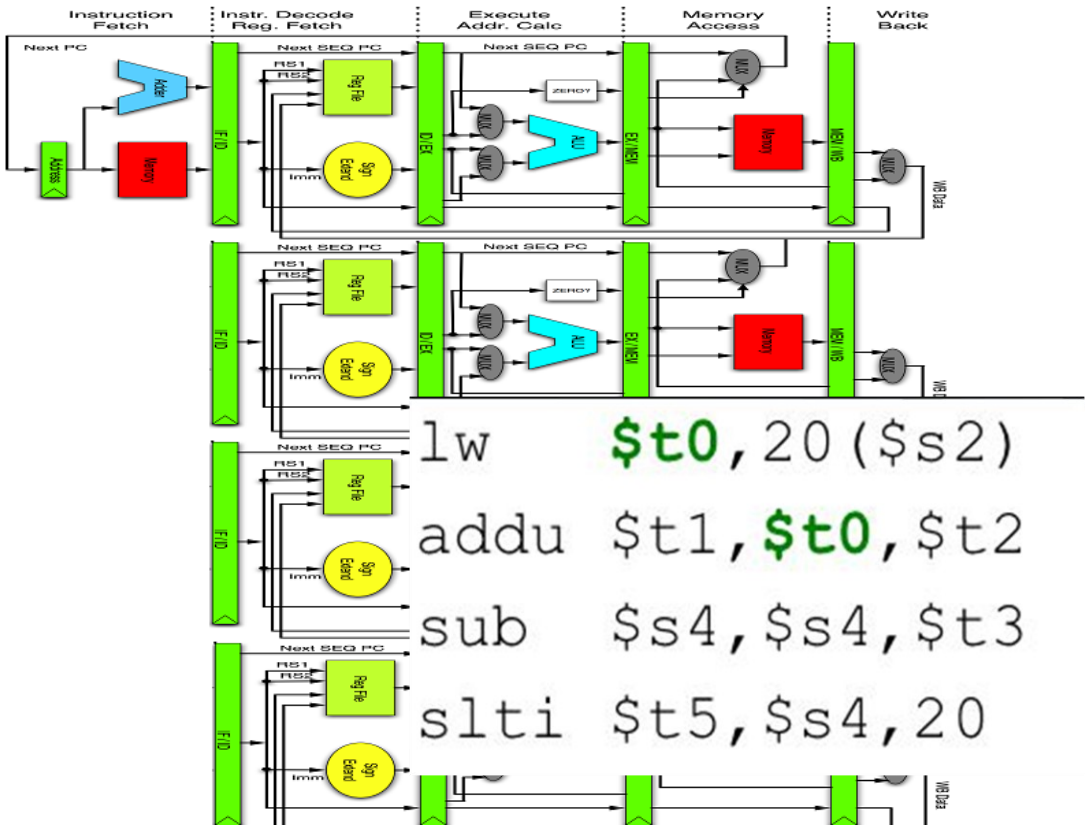

```
main() {
  for(i=0; i<CORES; i++)
    pthread_create(decode_exec);
    while(true) {
      instruction = fetch();
      enqueue(instruction);
    }
}

decode_exec() {
  instruction = dequeue();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

Doesn't look that different does it? Why do it?

Enables independent instruction parallelism.

Superscalar processors



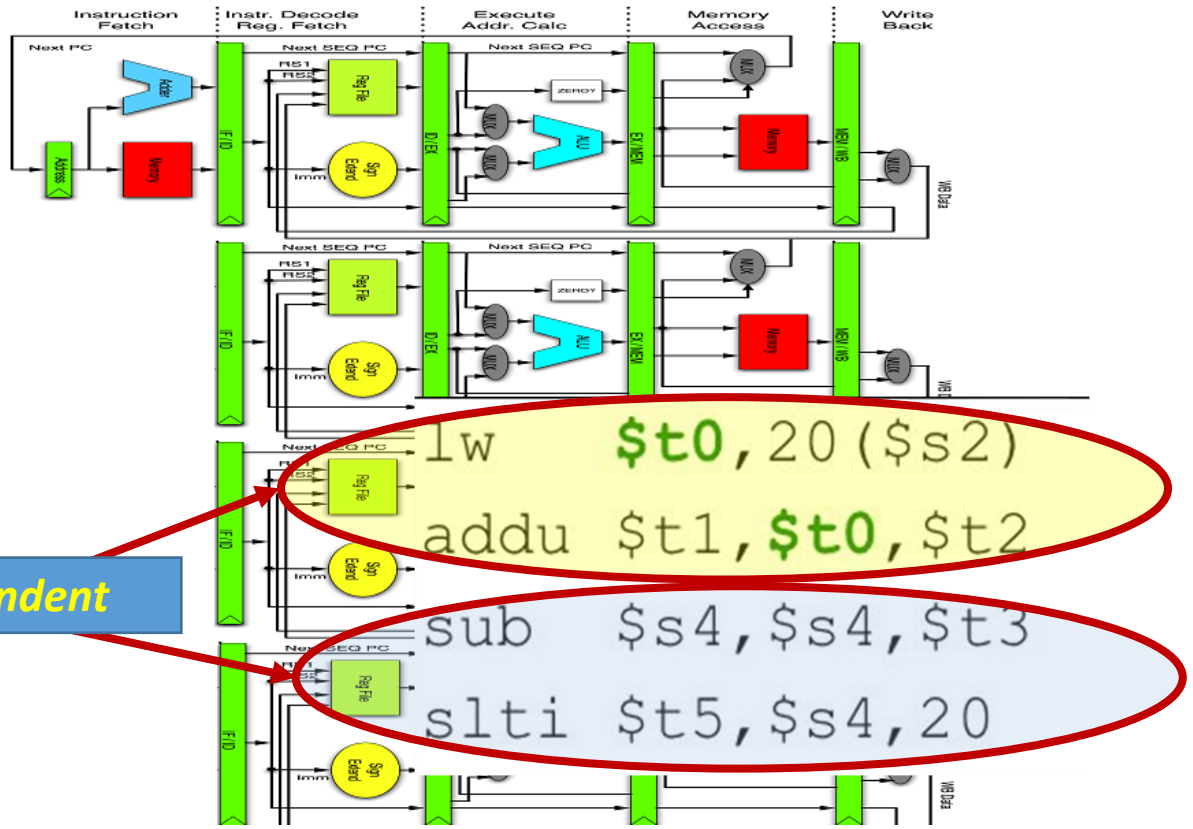

```
main() {
  for(i=0; i<CORES; i++)
    pthread_create(decode_exec);
    while(true) {
      instruction = fetch();
      enqueue(instruction);
    }
}

decode_exec() {
  instruction = dequeue();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

Doesn't look that different does it? Why do it?

Enables independent instruction parallelism.

Superscalar processors



independent

```
main() {
  for(i=0; i<CORES; i++)
    pthread_create(decode_exec);
    while(true) {
      instruction = fetch();
      enqueue(instruction);
    }
}

decode_exec() {
  instruction = dequeue();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

Doesn't look that different does it? Why do it?

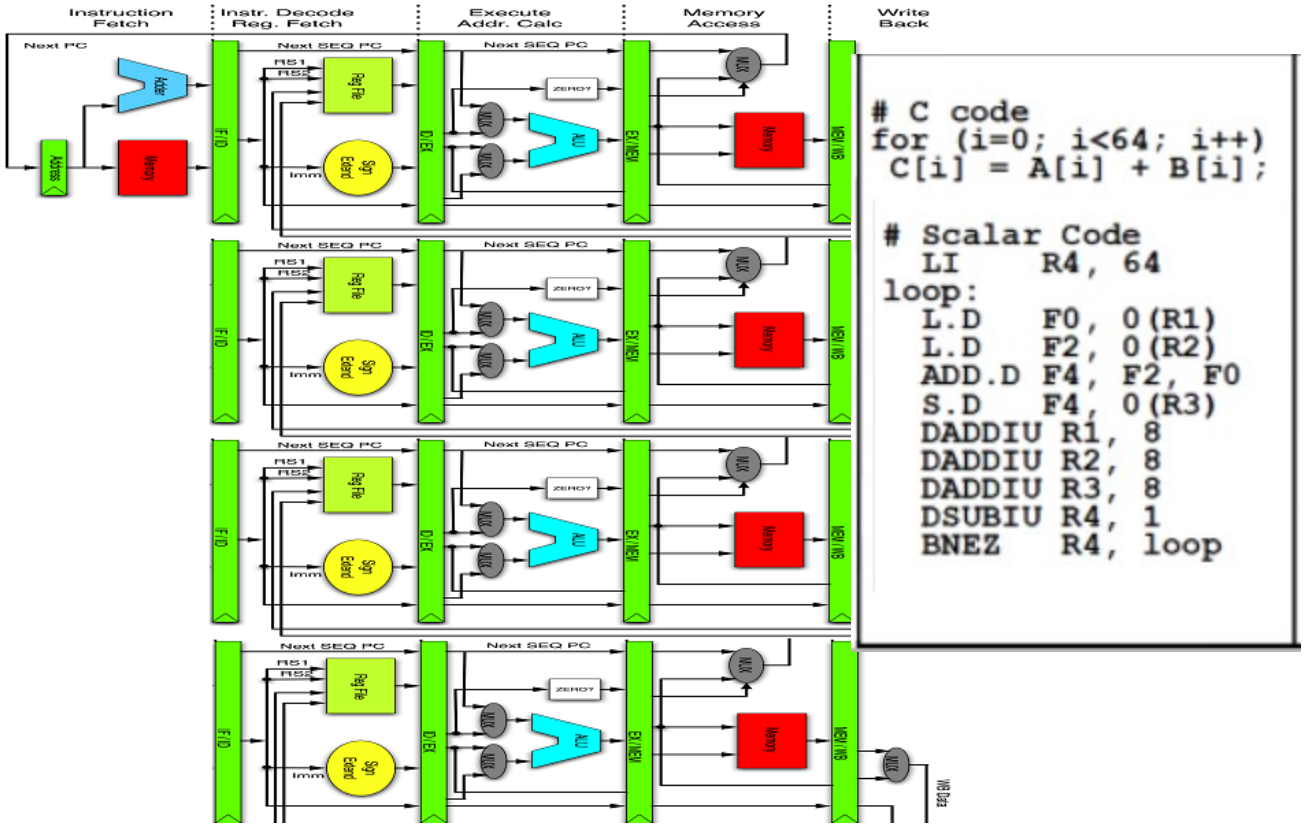
Enables independent instruction parallelism.

Vector/SIMD processors

```
# C code
for (i=0; i<64; i++)
 C[i] = A[i] + B[i];

# Scalar Code
   LI     R4, 64
loop:
   L.D    F0, 0(R1)
   L.D    F2, 0(R2)
   ADD.D  F4, F2, F0
   S.D    F4, 0(R3)
   DADDIU R1, 8
   DADDIU R2, 8
   DADDIU R3, 8
   DSUBIU R4, 1
   BNEZ   R4, loop
```

Vector/SIMD processors

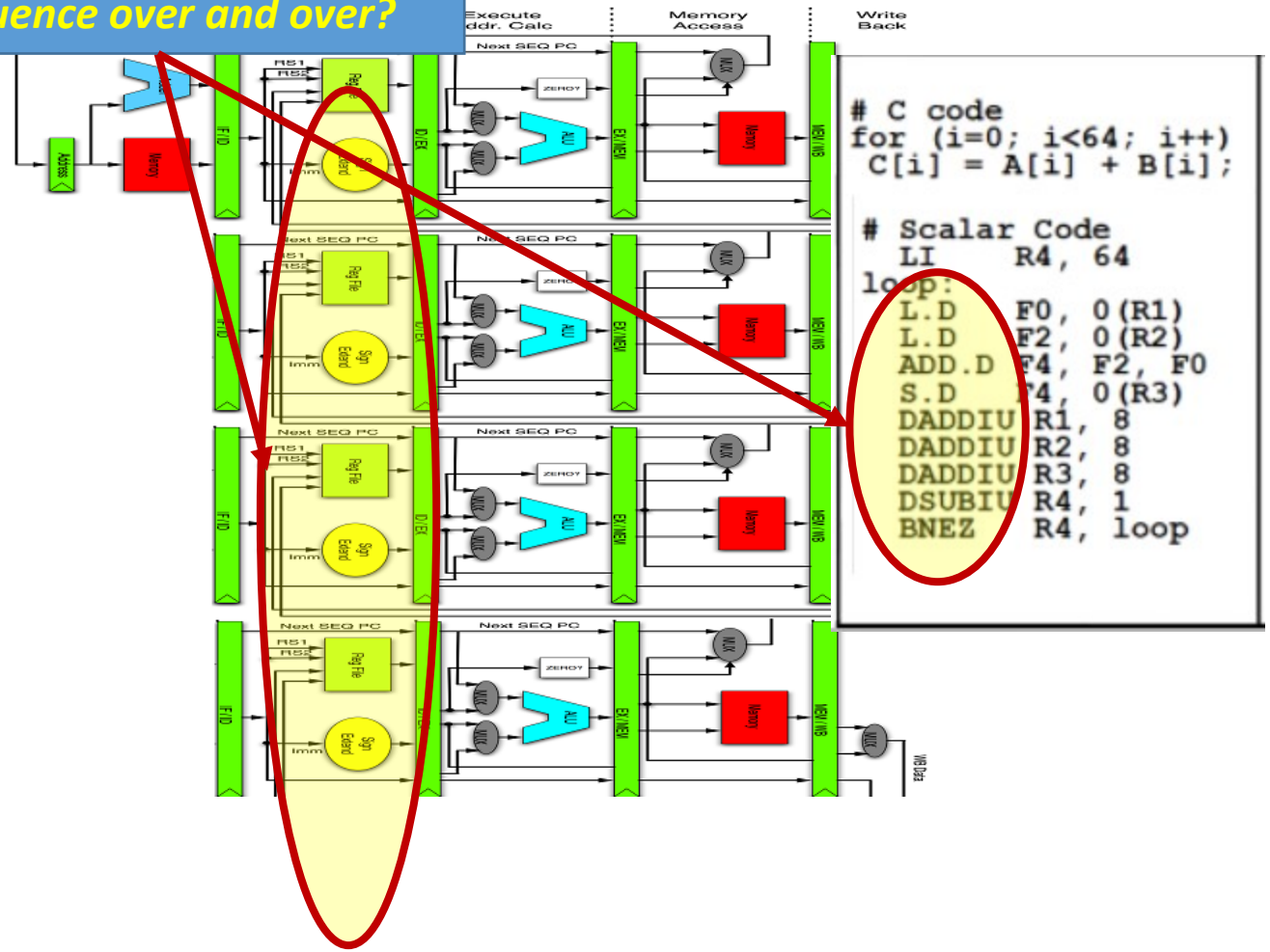



```
# C code
for (i=0; i<64; i++)
 C[i] = A[i] + B[i];

# Scalar Code
   LI     R4, 64
loop:
   L.D    F0, 0(R1)
   L.D    F2, 0(R2)
   ADD.D  F4, F2, F0
   S.D    F4, 0(R3)
   DADDIU R1, 8
   DADDIU R2, 8
   DADDIU R3, 8
   DSUBIU R4, 1
   BNEZ   R4, loop
```

Vector/SIMD processors

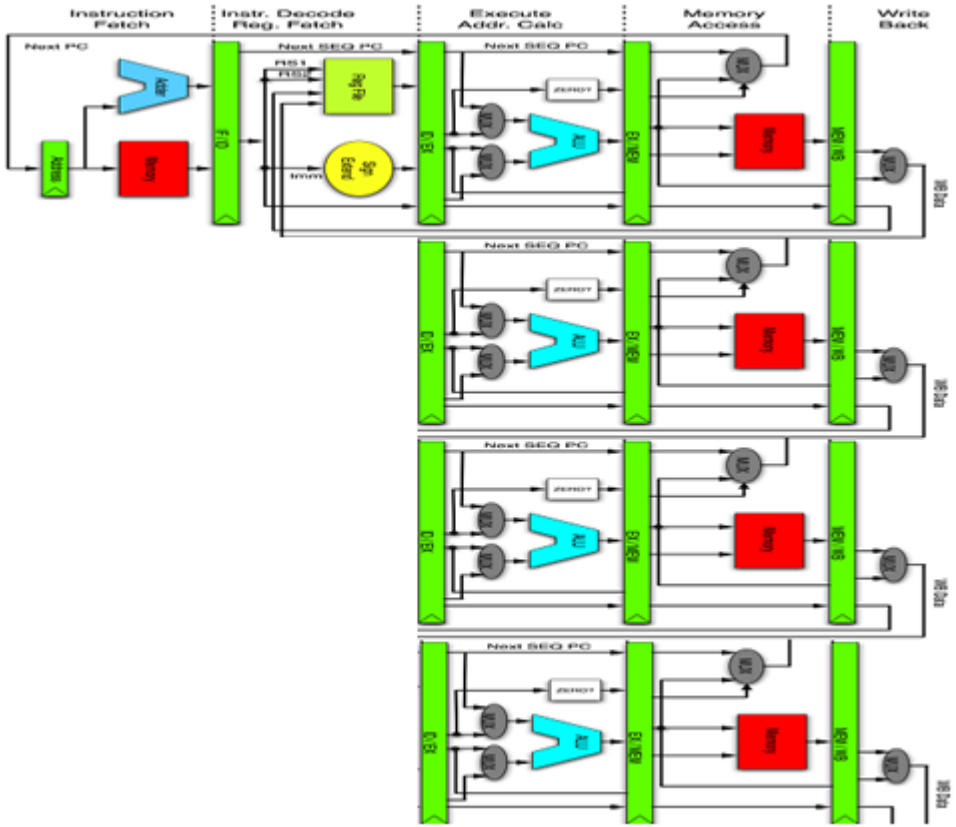
Why decode same instruction sequence over and over?



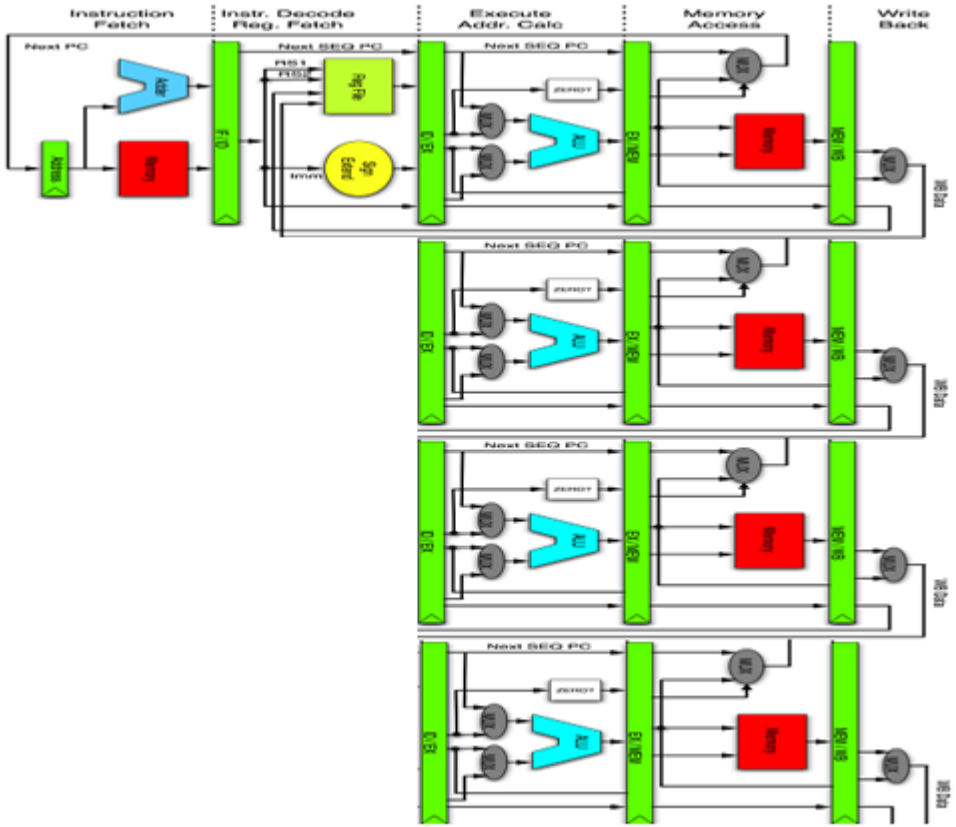

```
# C code
for (i=0; i<64; i++)
 C[i] = A[i] + B[i];

# Scalar Code
   LI     R4, 64
loop:
   L.D    F0, 0(R1)
   L.D    F2, 0(R2)
   ADD.D  F4, F2, F0
   S.D    F4, 0(R3)
   DADDIU R1, 8
   DADDIU R2, 8
   DADDIU R3, 8
   DSUBIU R4, 1
   BNEZ   R4, loop
```

Vector/SIMD processors



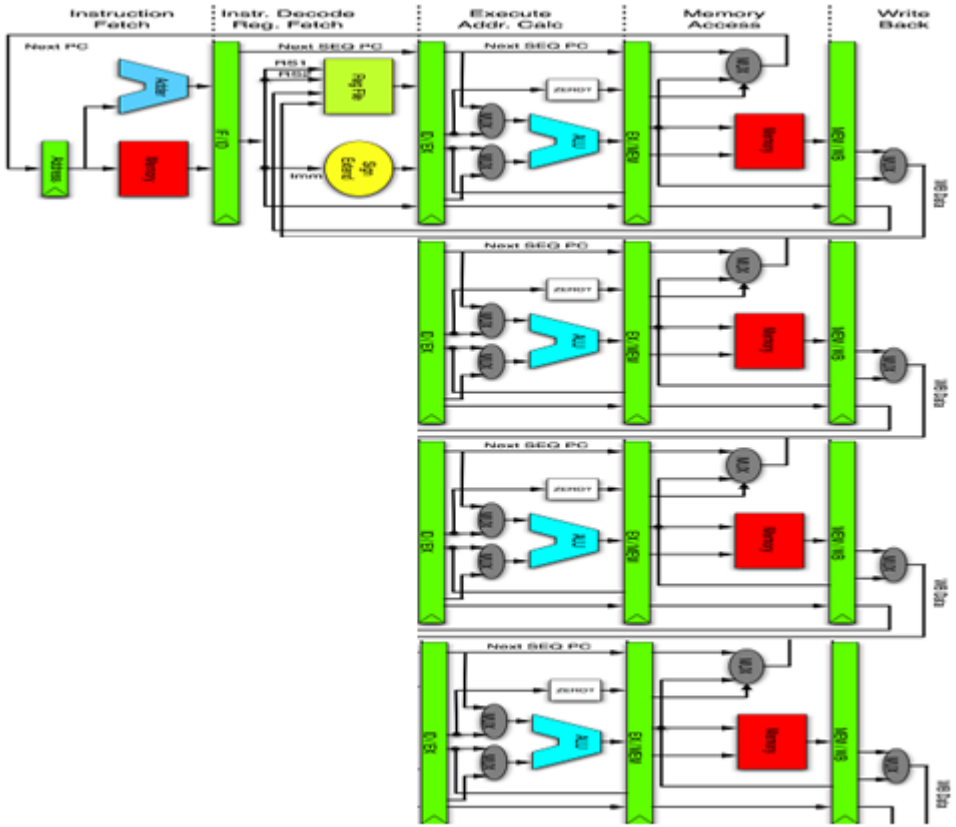
Vector/SIMD processors




```
main() {
  for(i=0; i<CORES; i++)
    pthread_create(exec);
    while(true) {
      ops, regs = fetch_decode();
      enqueue(ops, regs);
    }
}

exec() {
  ops, regs = dequeue();
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

Vector/SIMD processors

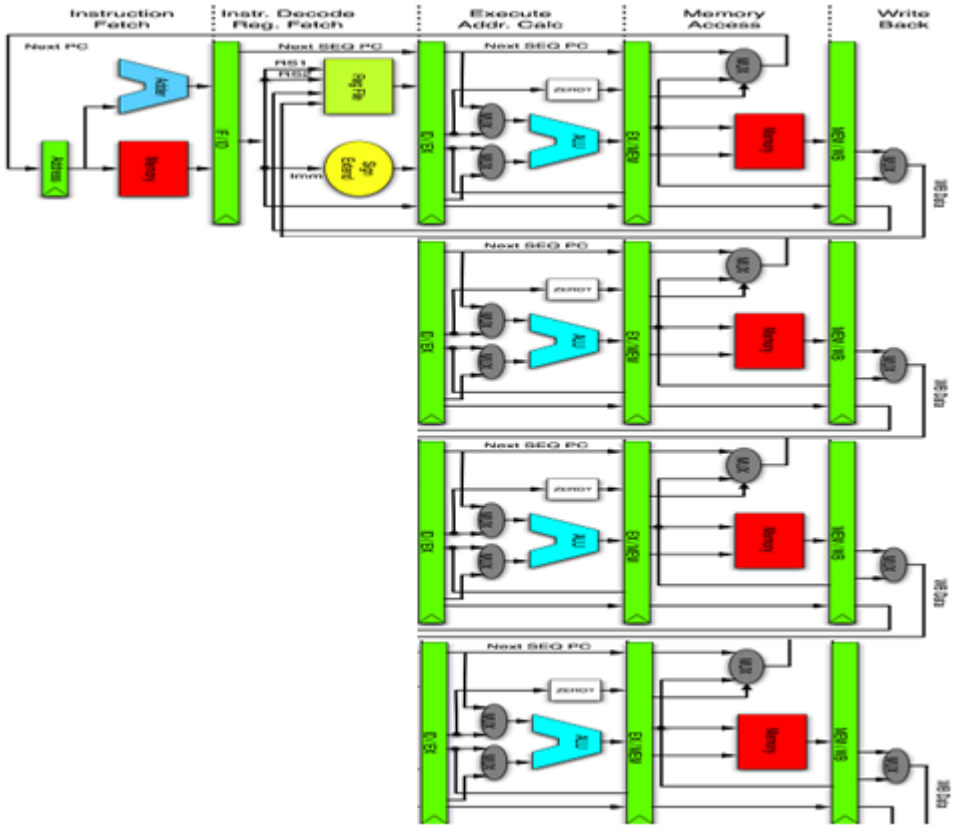



```
main() {
  for(i=0; i<CORES; i++)
    pthread_create(exec);
    while(true) {
      ops, regs = fetch_decode();
      enqueue(ops, regs);
    }
}

exec() {
  ops, regs = dequeue();
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

Single instruction stream, multiple computations

Vector/SIMD processors




```
main() {
  for(i=0; i<CORES; i++)
    pthread_create(exec);
    while(true) {
      ops, regs = fetch_decode();
      enqueue(ops, regs);
    }
}

exec() {
  ops, regs = dequeue();
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

Single instruction stream, multiple computations

But now all my instructions need multiple operands!