Lecture 2:

Instruction-Level Parallelism

15-418 Parallel Computer Architecture and Programming

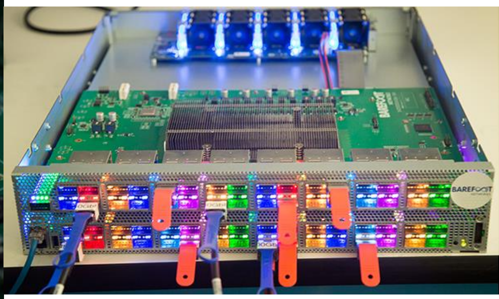
CMU 15-418/15-618, Spring 2020

Many kinds of processors

CPU

GPU

FPGA



Etc.

Why so many? What differentiates these processors?

Why so many kinds of processors?

Each processor is designed for different kinds of programs

TODAY

- CPUs
  - "Sequential" code – i.e., single / few threads
- GPUs
  - Programs with lots of independent work → "Embarrassingly parallel"
- Many others: Deep neural networks, Digital signal processing, Etc.

Parallelism pervades architecture

- Speeding up programs is all about parallelism
  - 1) Find independent work
  - 2) Execute it in parallel
  - 3) Profit

- Key questions:
  - Where is the parallelism?
  - Whose job is it to find parallelism?

Where is the parallelism?

Different processors take radically different approaches

- CPUs: Instruction-level parallelism
  - Implicit
  - Fine-grain

- GPUs: Thread- & data-level parallelism
  - Explicit
  - Coarse-grain

Whose job to find parallelism?

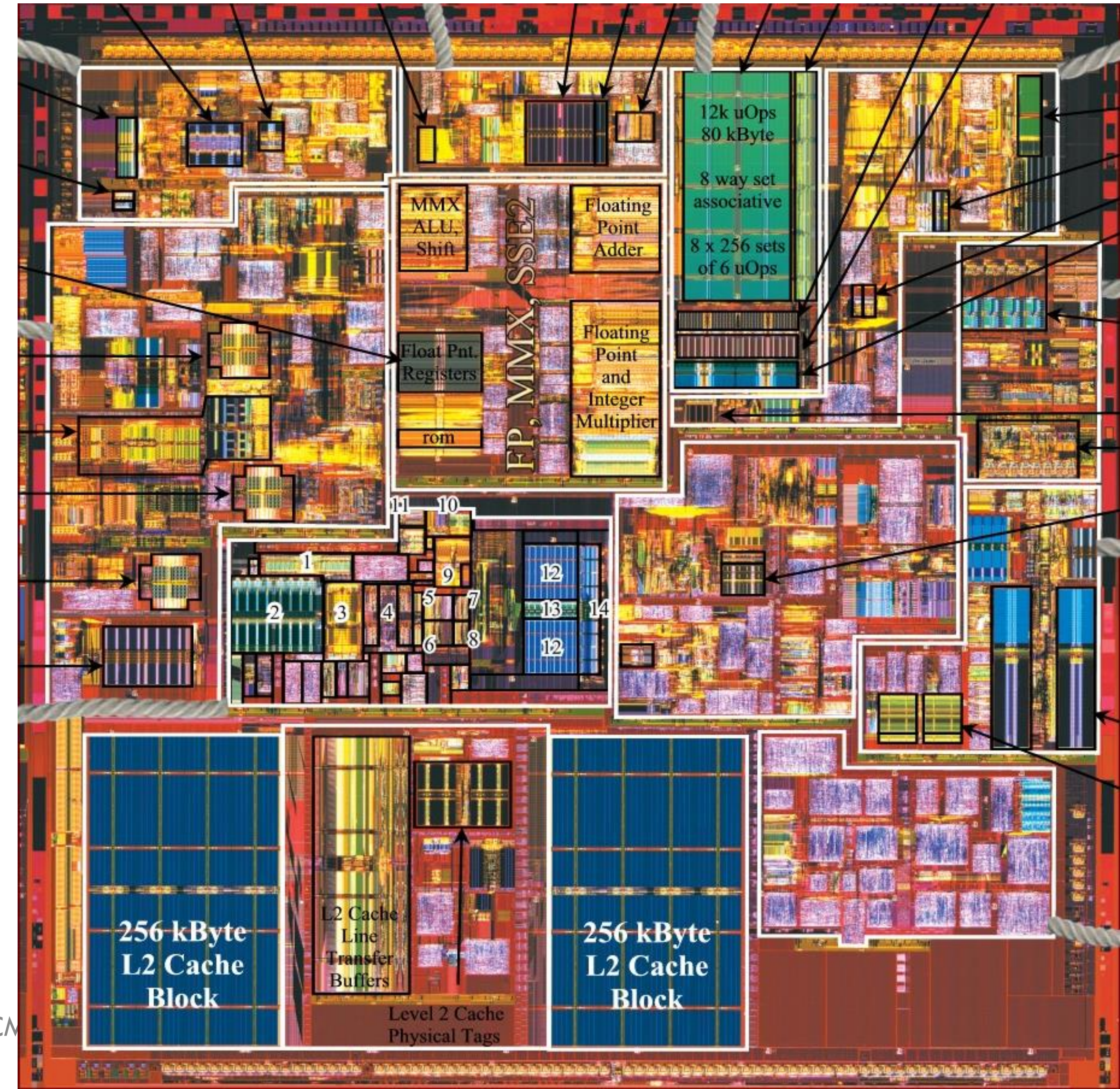
Different processors take radically different approaches

- CPUs: Hardware dynamically schedules instructions
  - Expensive, complex hardware ➔ Few cores (tens)
  - (Relatively) Easy to write fast software

- GPUs: Software makes parallelism explicit
  - Simple, cheap hardware ➔ Many cores (thousands)
  - (Often) Hard to write fast software

Visualizing these differences

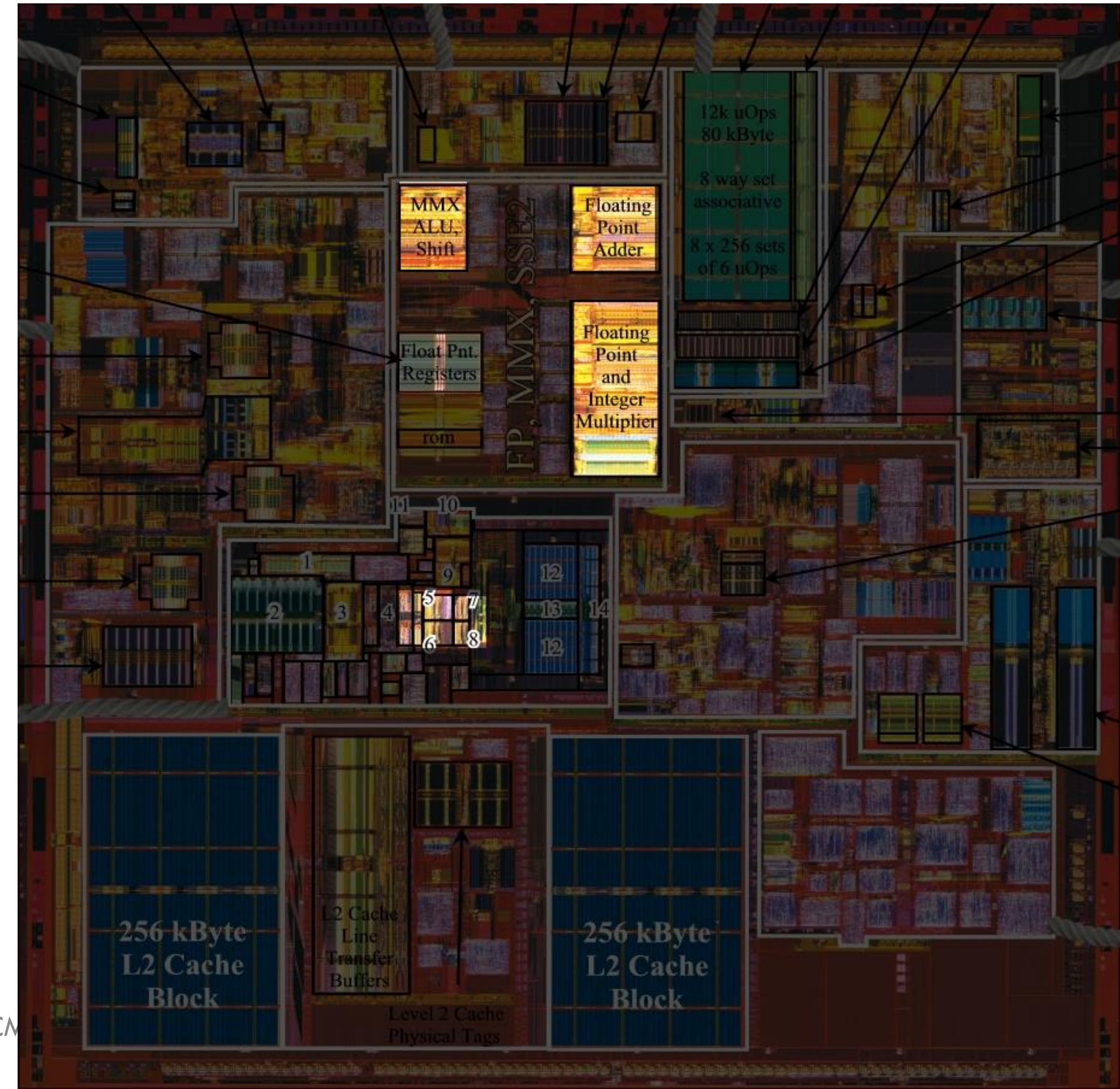
- Pentium 4 "Northwood" (2002)



Visualizing these differences

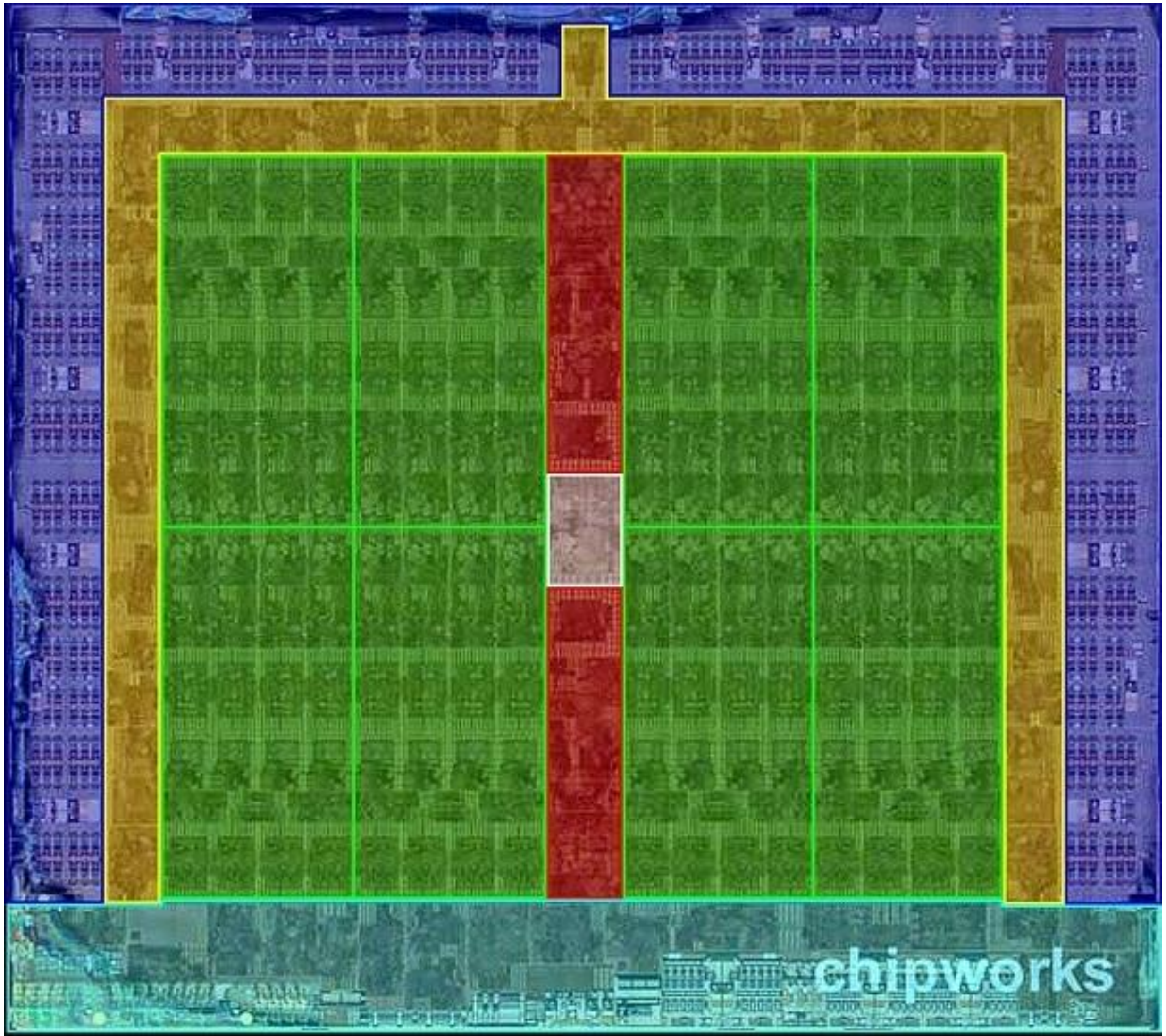
- Pentium 4 “Northwood” (2002)
- Highlighted areas actually execute instructions

  ➔ <u>Most area spent on scheduling</u>
  (not on executing the program)



Visualizing these differences

- AMD Fiji (2015)

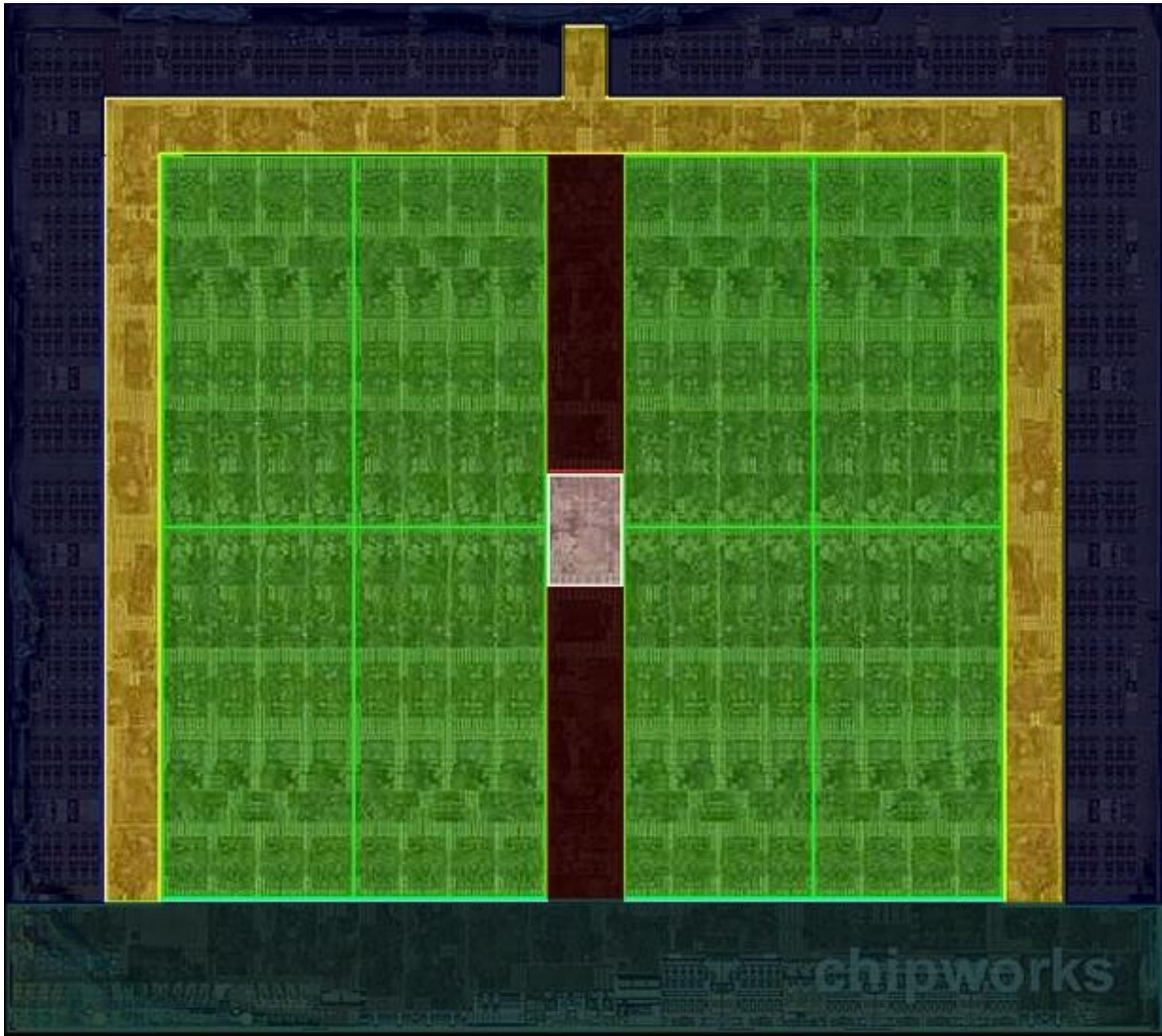


Visualizing these differences

- AMD Fiji (2015)

- Highlighted areas actually execute instructions

  ➔ Most area spent executing the program
  - (Rest is mostly I/O & memory, not scheduling)



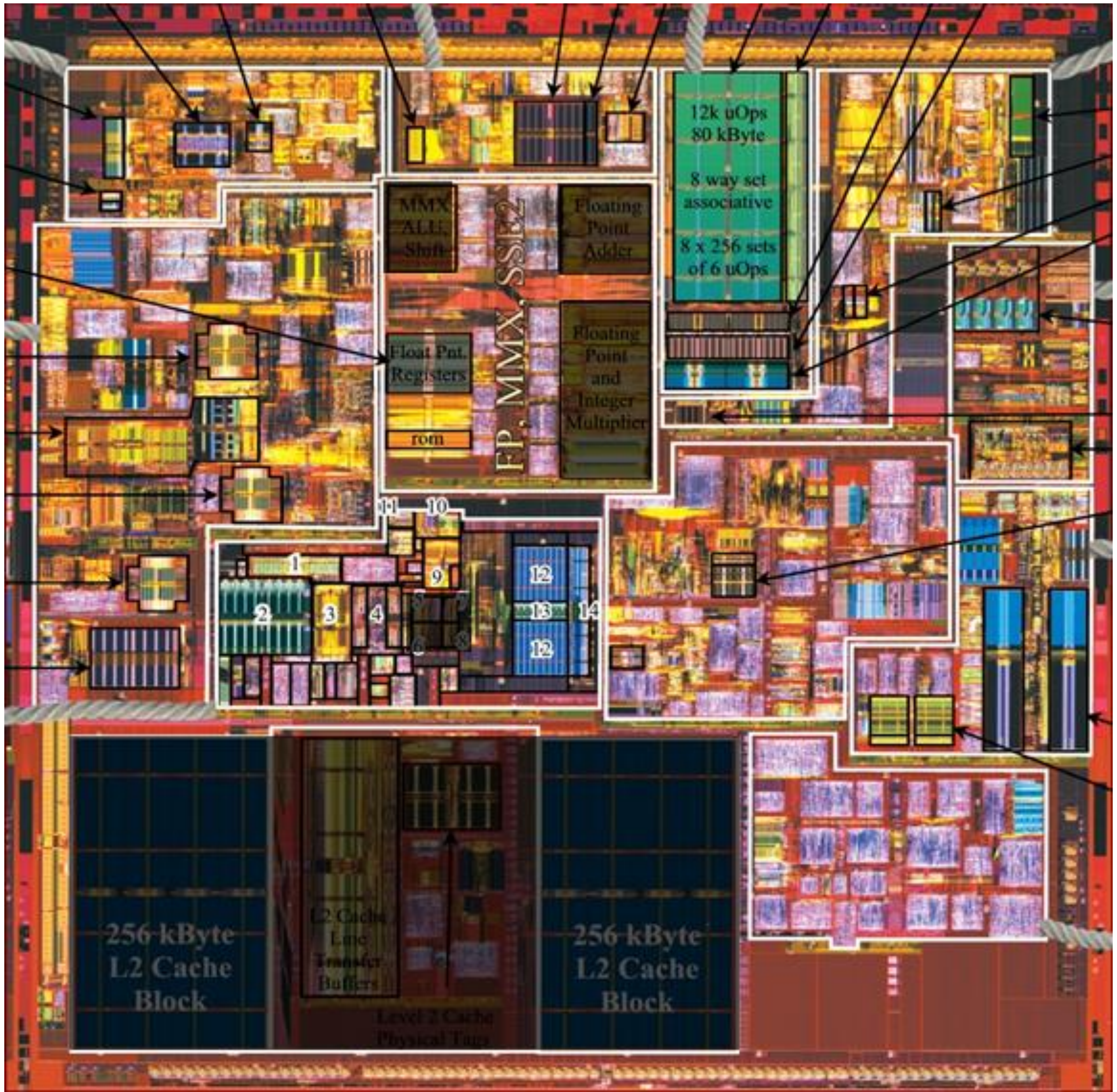
Today you will learn…

How CPUs exploit ILP to speed up sequential code

- Key ideas:
  - *Pipelining & Superscalar:* Work on multiple instructions at once
  - *Out-of-order execution:* Dynamically schedule instructions whenever they are "ready"
  - *Speculation*: Guess what the program will do next to discover more independent work, "rolling back" incorrect guesses

- CPUs must do all of this while preserving the illusion that instructions execute in-order, one-at-a-time

In other words… Today is about:



Buckle up!

…But please ask questions!

Example: Polynomial evaluation

```
int poly(int *coef,
         int terms, int x) {
  int power = 1;
  int value = 0;
  for (int j = 0; j < terms; j++) {
    value += coef[j] * power;
    power *= x;
  }
  return value;
}
```

Example: Polynomial evaluation

```
r0: value
r1: &coef[terms]
r2: x
r3: &coef[0]
r4: power
r5: coef[j]
```

- Compiling on ARM

```
int poly(int *coef,
         int terms, int x) {
  int power = 1;
  int value = 0;
  for (int j = 0; j < terms; j++) {
    value += coef[j] * power;
    power *= x;
  }
  return value;
}
```

```
poly:
    cmp     r1, #0
    ble     .L4
    push    {r4, r5}
    mov     r3, r0
    add     r1, r0, r1, lsl #2
    movs    r4, #1
    movs    r0, #0
.L3:
    ldr     r5, [r3], #4
    cmp     r1, r3
    mla     r0, r4, r5, r0
    mul     r4, r2, r4
    bne     .L3
    pop     {r4, r5}
    bx      lr
.L4:
    movs    r0, #0
    bx      lr
```

Example: Polynomial evaluation

- Compiling on ARM

```
r0: value
r1: &coef[terms]
r2: x
r3: &coef[j]
r4: power
r5: coef[j]
```

```
int poly(int *coef,
         int terms, int x) {
  int power = 1;
  int value = 0;
  for (int j = 0; j < terms; j++) {
    value += coef[j] * power;
    power *= x;
  }
  return value;
}
```

Preamble

```
poly:
  cmp     r1, #0
  ble     .L4
  push    {r4, r5}
  mov     r3, r0
  add     r1, r0, r1, lsl #2
  movs    r4, #1
  movs    r0, #0
```

Iteration

```
.L3:
  ldr     r5, [r3], #4
  cmp     r1, r3
  mla     r0, r4, r5, r0
  mul     r4, r2, r4
  bne     .L3
```

Fini

```
  pop     {r4, r5}
  bx      lr
.L4:
  movs    r0, #0
  bx      lr
```

Example: Polynomial evaluation

```
r0: value
r1: &coef[terms]
r2: x
r3: &coef[j]
r4: power
r5: coef[j]
```

- Compiling on ARM

```
for (int j = 0; j < terms; j++) {
  value += coef[j] * power;
  power *= x;
}
```

```
.L3:
  ldr     r5, [r3], #4      // r5 <- coef[j]; j++   (two operations)
  cmp     r1, r3            // compare: j < terms?
  mla     r0, r4, r5, r0    // value += r5 * power  (mul + add)
  mul     r4, r2, r4        // power *= x
  bne     .L3               // repeat?
```

Example: Polynomial evaluation

- Executing `poly(A, 3, x)`

```
cmp     r1, #0
ble     .L4
push    {r4, r5}
mov     r3, r0
add     r1, r0, r1, lsl #2
movs    r4, #1
movs    r0, #0
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
...
```

Example: Polynomial evaluation

- Executing `poly(A, 3, x)`

```
cmp     r1, #0                 Preamble
ble     .L4
push    {r4, r5}
mov     r3, r0
add     r1, r0, r1, lsl #2
movs    r4, #1
movs    r0, #0
ldr     r5, [r3], #4           J=0 iteration
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
...
```

Example: Polynomial evaluation

- Executing `poly(A, 3, x)`

Preamble

```
cmp     r1, #0
ble     .L4
push    {r4, r5}
mov     r3, r0
add     r1, r0, r1, lsl #2
movs    r4, #1
movs    r0, #0
```

J=0 iteration

```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
...
```

```
...
```

J=1 iteration

```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
```

J=2 iteration

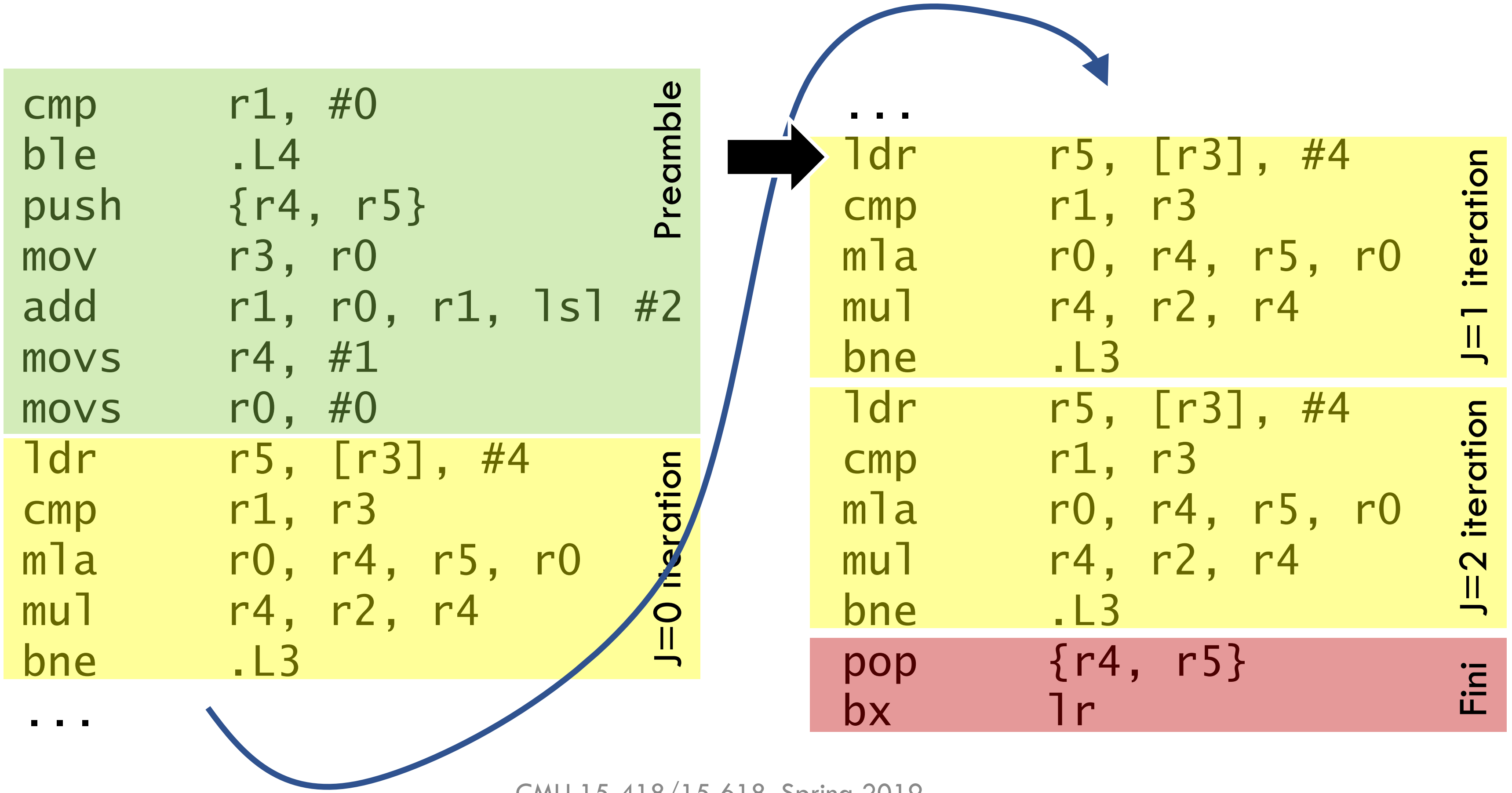
```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
```

Fini

```
pop     {r4, r5}
bx      lr
```

Example: Polynomial evaluation

- Executing `poly(A, 3, x)`



```
cmp     r1, #0
ble     .L4
push    {r4, r5}
mov     r3, r0
add     r1, r0, r1, lsl #2
movs    r4, #1
movs    r0, #0
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
...
```

```
...
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
pop     {r4, r5}
bx      lr
```

The software-hardware boundary

- The *instruction set architecture (ISA)* is a functional contract between hardware and software
  - It says **what** each instruction does, but not **how**
  - Example: Ordered sequence of x86 instructions

- A processor's *microarchitecture* is how the ISA is implemented

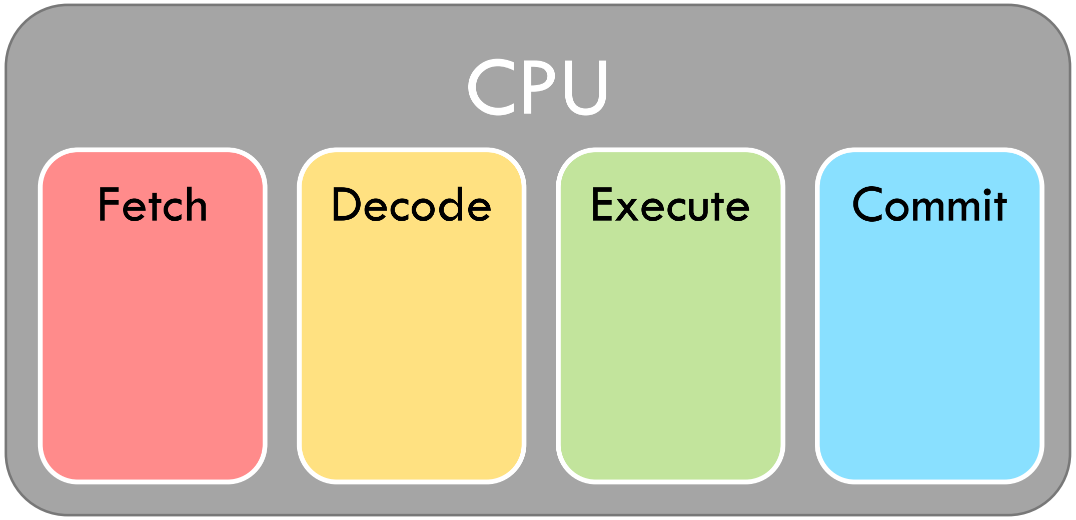
Arch : μArch :: Interface : Implementation

Simple CPU model

- Execute instructions in program order
- Divide instruction execution into stages, e.g.:
  - 1. Fetch – get the next instruction from memory
  - 2. Decode – figure out what to do & read inputs
  - 3. Execute – perform the necessary operations
  - 4. Commit – write the results back to registers / memory
  - (Real processors have many more stages)

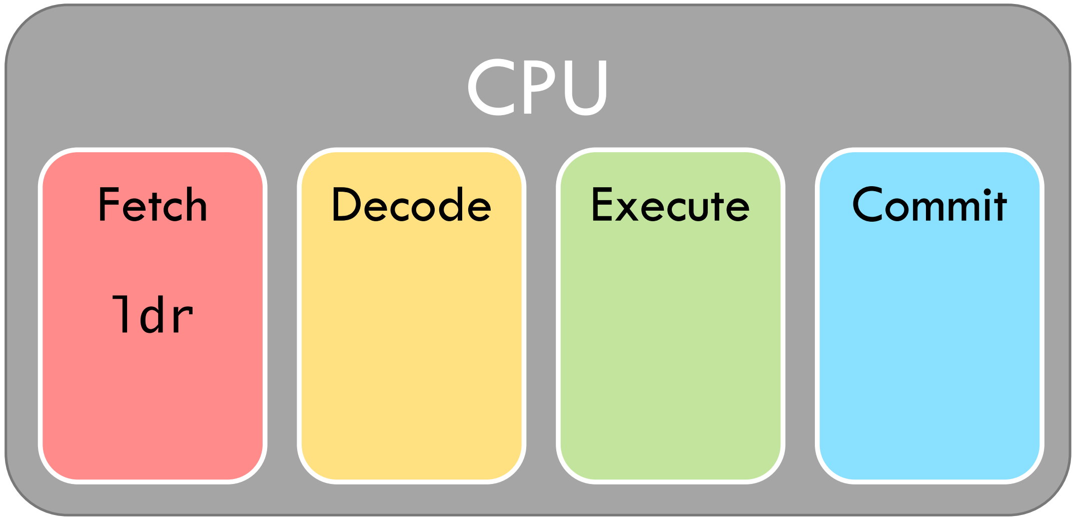
Evaluating polynomial on the simple CPU model

```
ldr      r5, [r3], #4
cmp      r1, r3
mla      r0, r4, r5, r0
mul      r4, r2, r4
bne      .L3

ldr      r5, [r3], #4
cmp      r1, r3
mla      r0, r4, r5, r0
mul      r4, r2, r4
bne      .L3

...
```



Evaluating polynomial on the simple CPU model

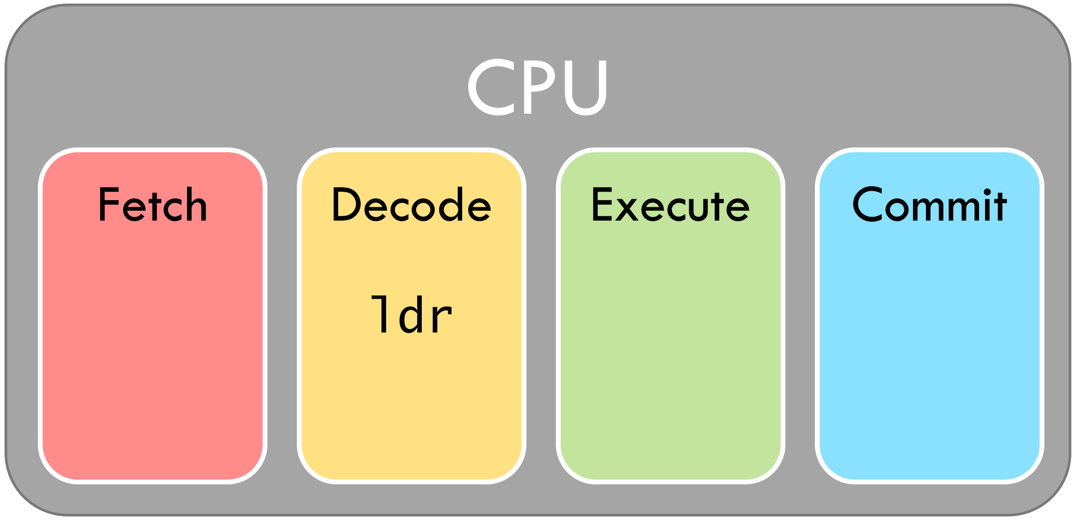
```
ldr      r5, [r3], #4
cmp      r1, r3
mla      r0, r4, r5, r0
mul      r4, r2, r4
bne      .L3

ldr      r5, [r3], #4
cmp      r1, r3
mla      r0, r4, r5, r0
mul      r4, r2, r4
bne      .L3

...
```



1. Read "ldr r5, [r3] #4" from memory

Evaluating polynomial on the simple CPU model

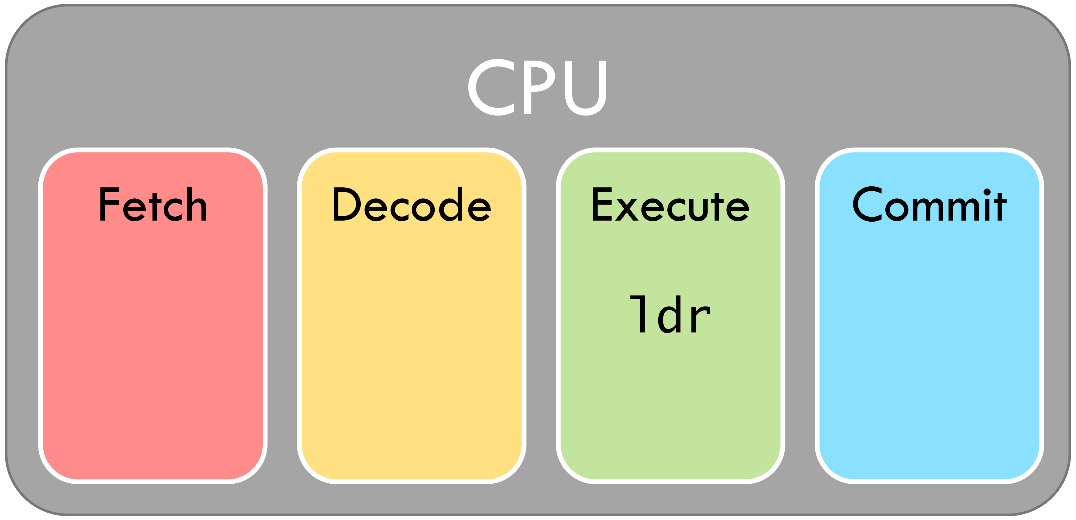
```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

...
```



2. Decode "ldr r5, [r3] #4" and read input regs

Evaluating polynomial on the simple CPU model

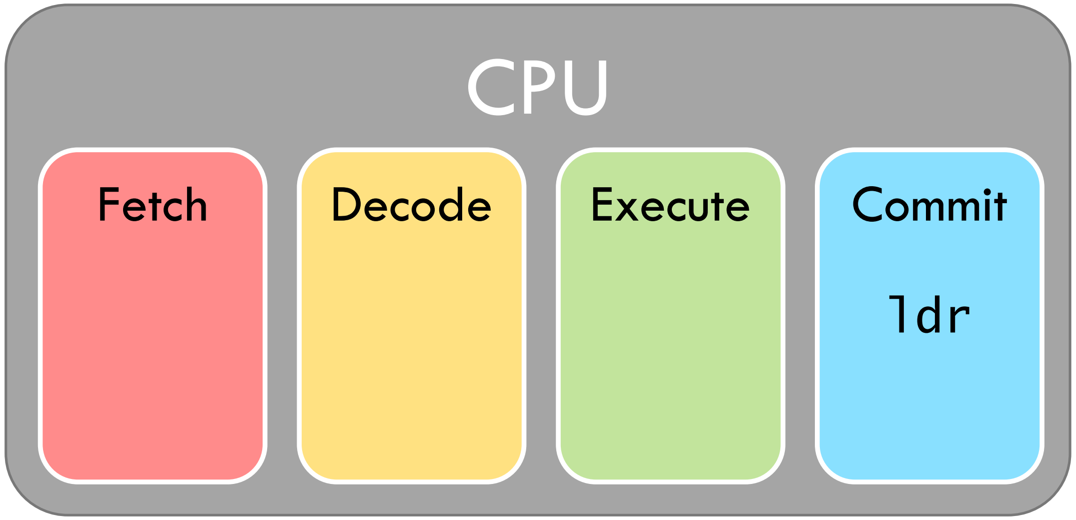
```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

...
```



3. Load memory at r3 and compute r3 + 4

Evaluating polynomial on the simple CPU model

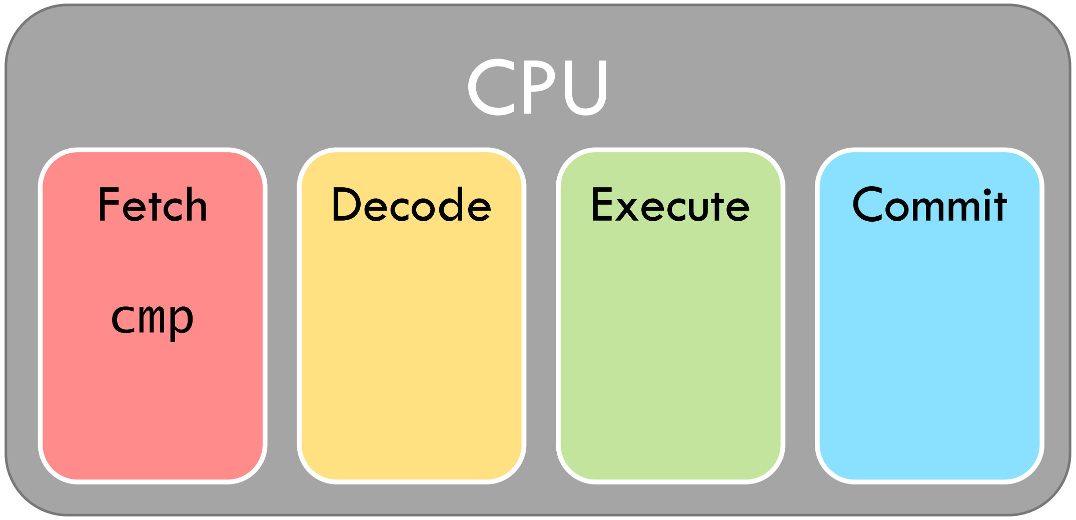
```
ldr      r5, [r3], #4
cmp      r1, r3
mla      r0, r4, r5, r0
mul      r4, r2, r4
bne      .L3

ldr      r5, [r3], #4
cmp      r1, r3
mla      r0, r4, r5, r0
mul      r4, r2, r4
bne      .L3

...
```



4. Write values into regs r5 and r3

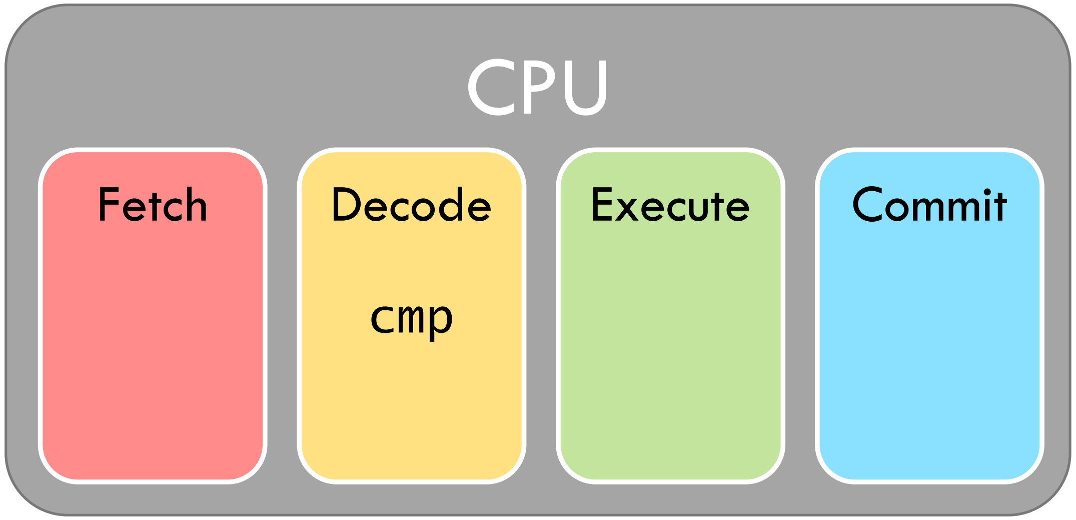
Evaluating polynomial on the simple CPU model

```
   ldr     r5, [r3], #4
➡  cmp     r1, r3
   mla     r0, r4, r5, r0
   mul     r4, r2, r4
   bne     .L3

   ldr     r5, [r3], #4
   cmp     r1, r3
   mla     r0, r4, r5, r0
   mul     r4, r2, r4
   bne     .L3

   ...
```



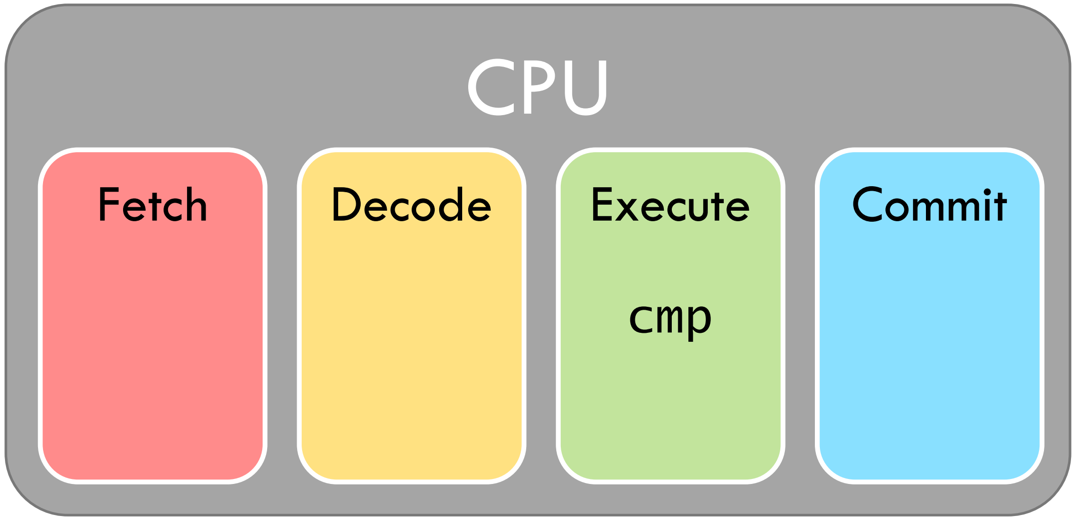
Evaluating polynomial on the simple CPU model

```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

...
```



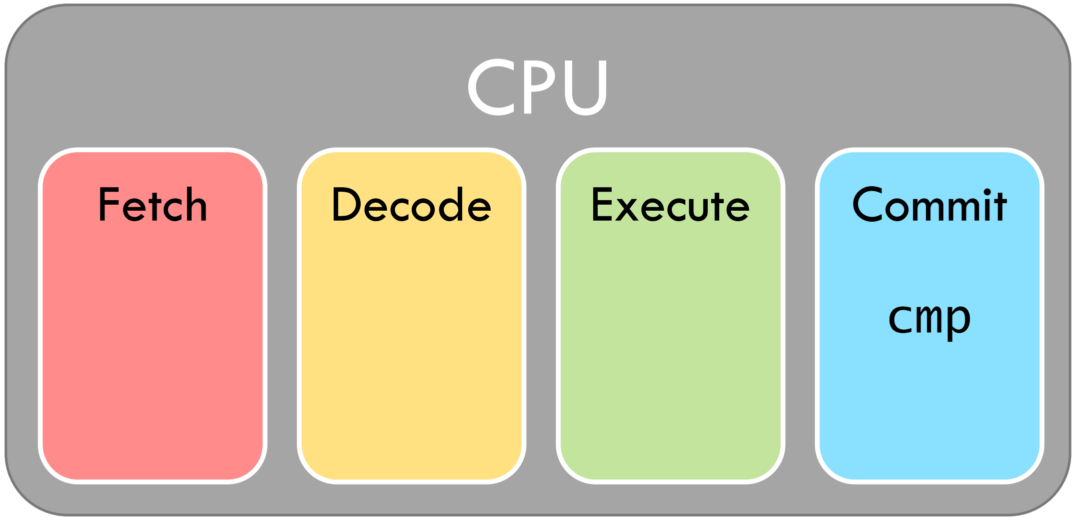
Evaluating polynomial on the simple CPU model

```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

...
```



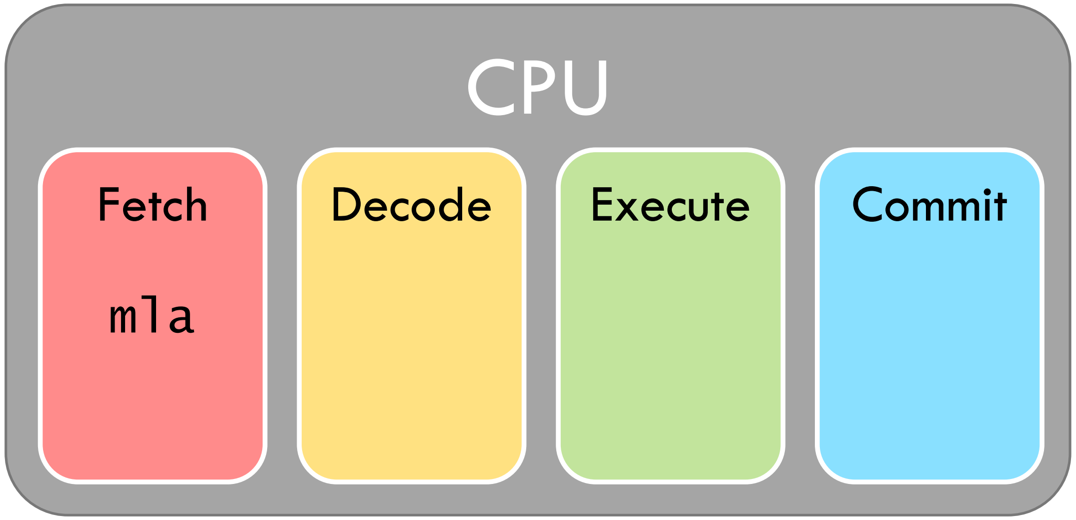
Evaluating polynomial on the simple CPU model

```
ldr      r5, [r3], #4
cmp      r1, r3
mla      r0, r4, r5, r0
mul      r4, r2, r4
bne      .L3

ldr      r5, [r3], #4
cmp      r1, r3
mla      r0, r4, r5, r0
mul      r4, r2, r4
bne      .L3

...
```



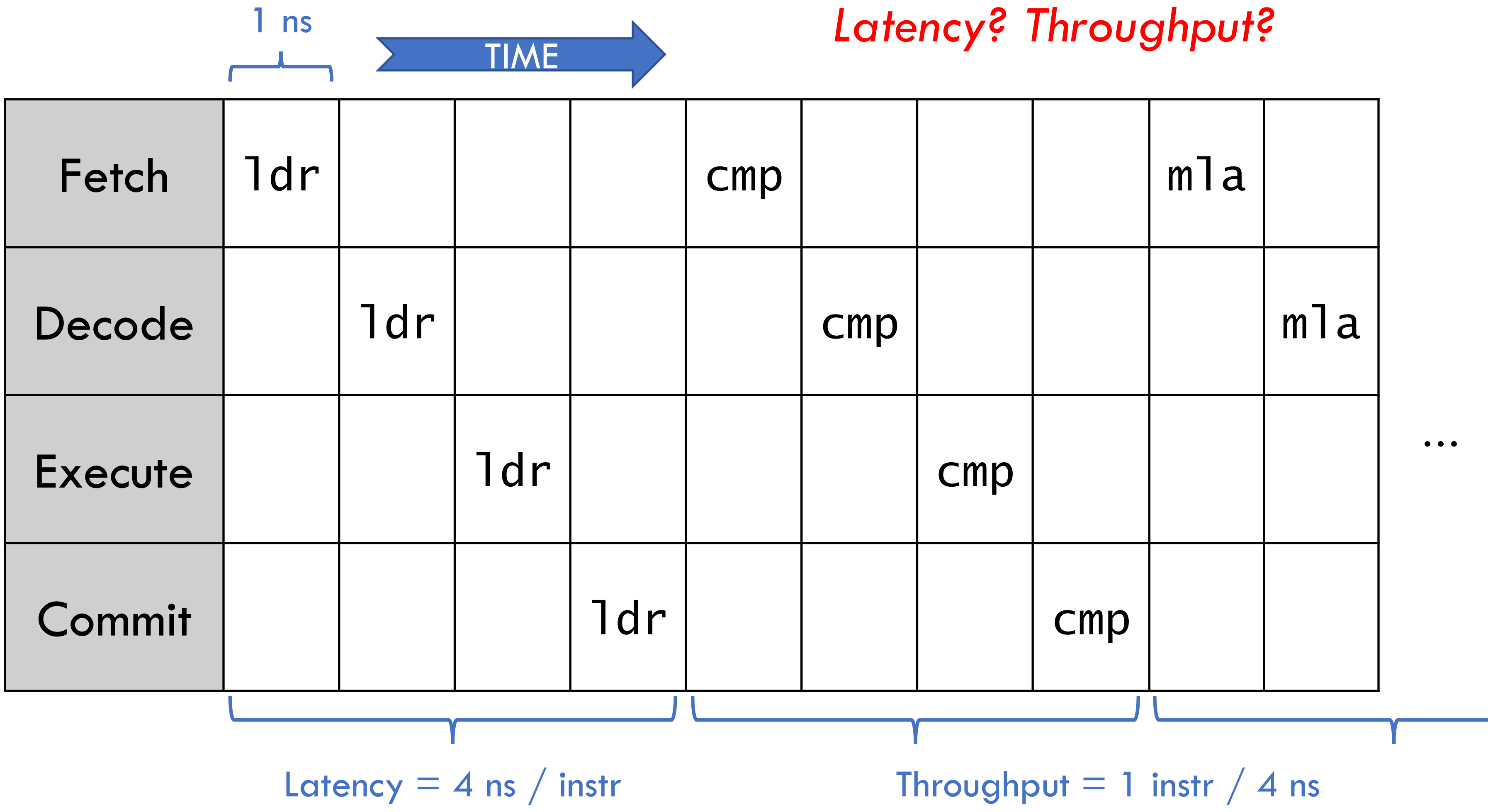
Evaluating polynomial on the simple CPU model

```
      ldr     r5, [r3], #4
      cmp     r1, r3
➡     mla     r0, r4, r5, r0
      mul     r4, r2, r4
      bne     .L3

      ldr     r5, [r3], #4
      cmp     r1, r3
      mla     r0, r4, r5, r0
      mul     r4, r2, r4
      bne     .L3

      ...
```



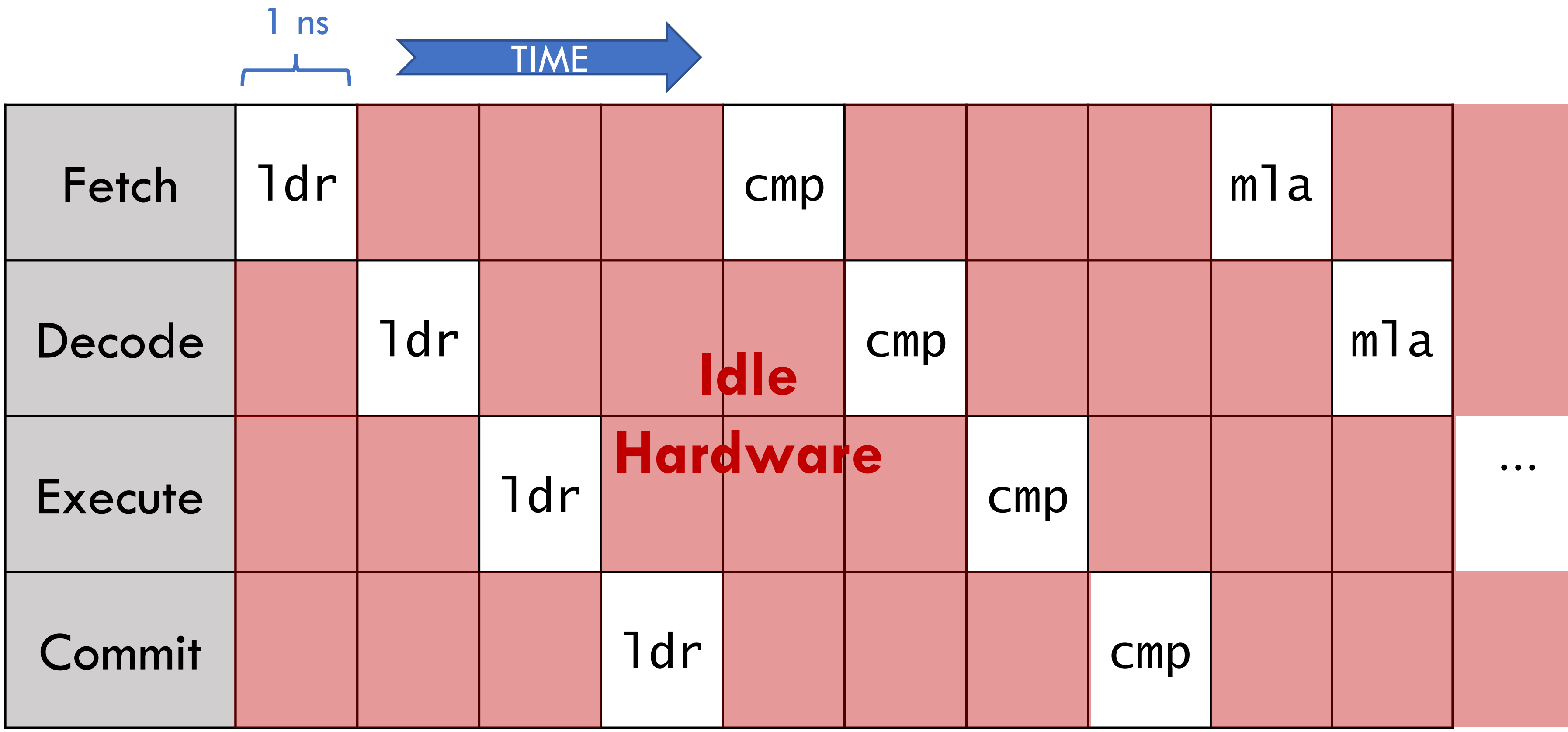
Evaluating polynomial on the simple CPU model

How fast is this processor?
Latency? Throughput?



Fetch	`ldr`				`cmp`				`mla`	
Decode		`ldr`				`cmp`				`mla`
Execute			`ldr`				`cmp`			
Commit				`ldr`				`cmp`		

Simple CPU is very wasteful

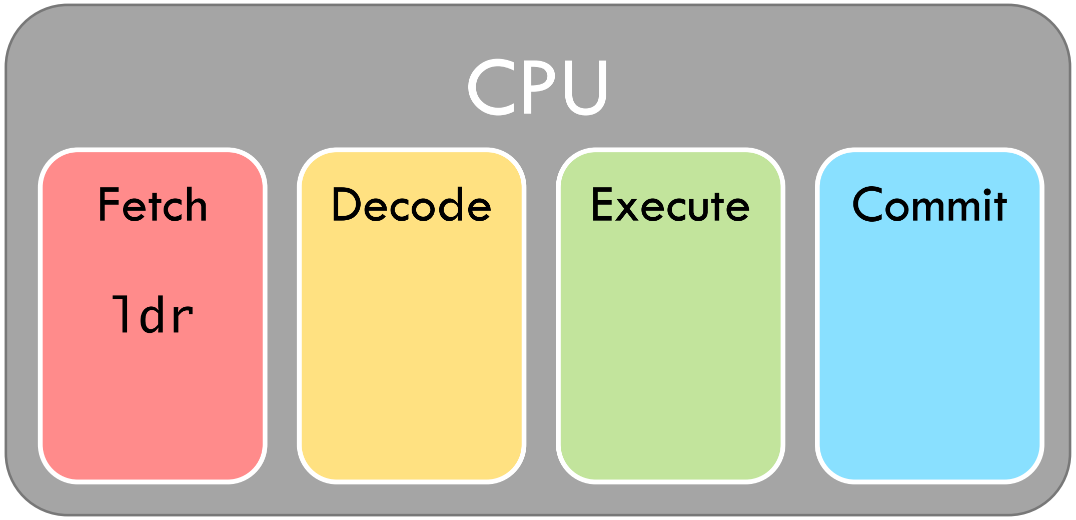


Pipelining

Pipelining keeps CPU busy through instruction-level parallelism

- Idea: Start on the next instr'n immediately

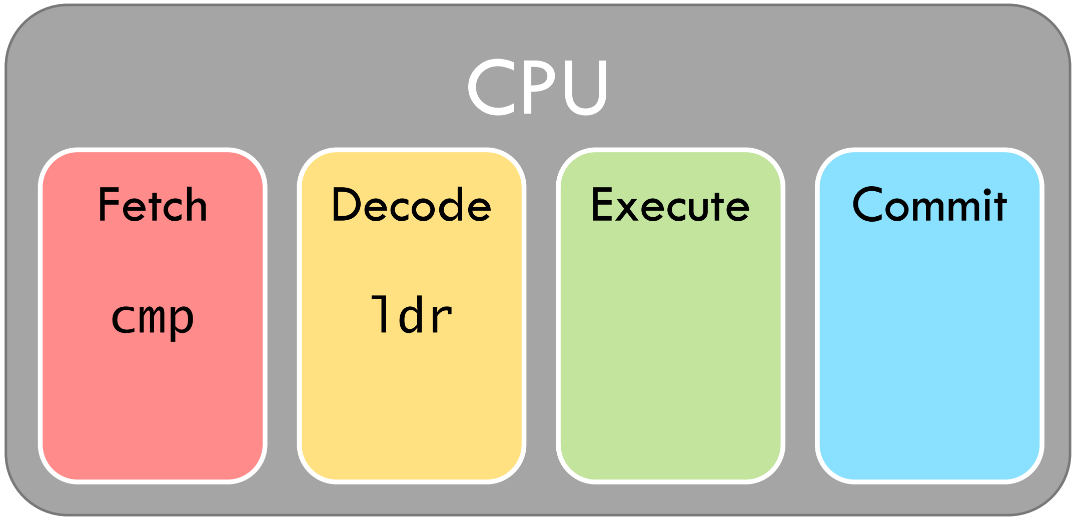
```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

...
```



Pipelining keeps CPU busy through instruction-level parallelism

- Idea: Start on the next instr'n immediately

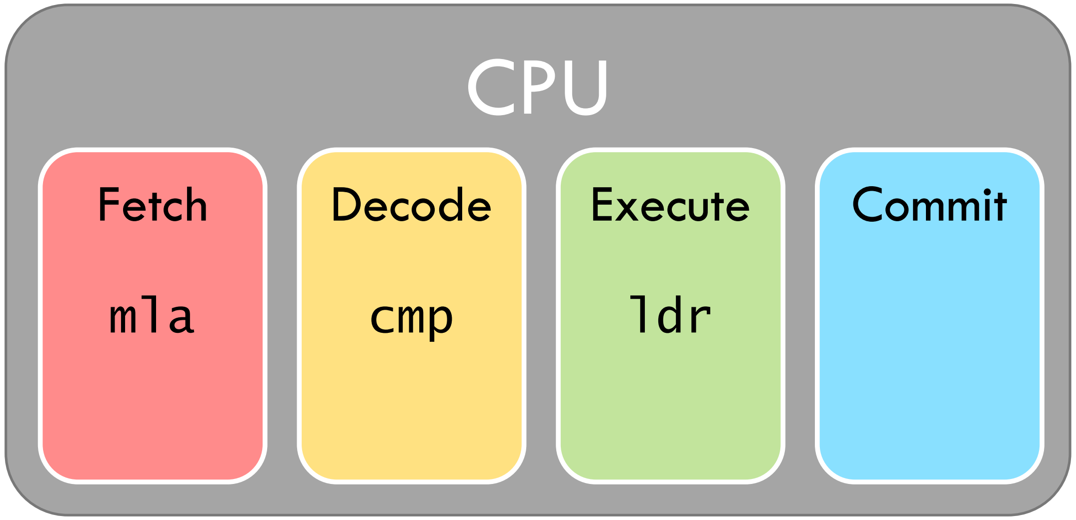
```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

...
```



Pipelining keeps CPU busy through instruction-level parallelism

- Idea: Start on the next instr'n immediately

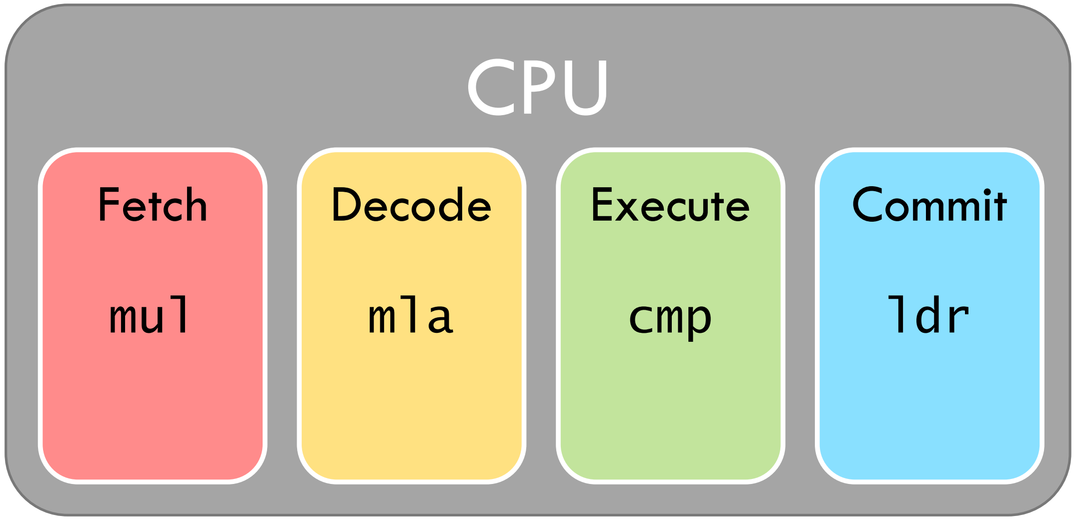
```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

...
```



Pipelining keeps CPU busy through instruction-level parallelism

- Idea: Start on the next instr'n immediately

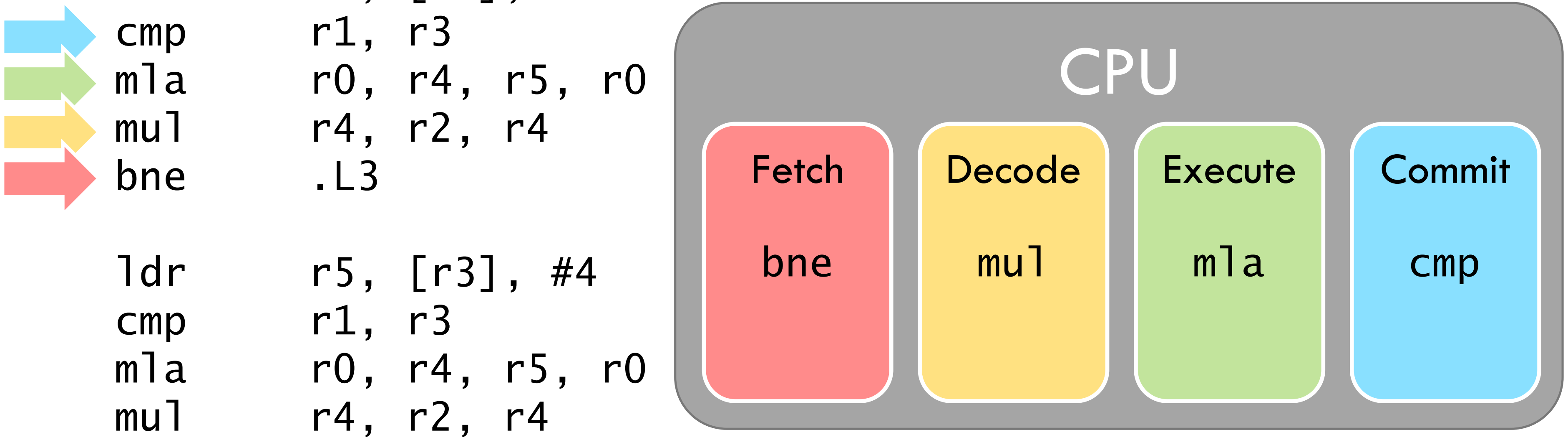
```
ldr      r5, [r3], #4
cmp      r1, r3
mla      r0, r4, r5, r0
mul      r4, r2, r4
bne      .L3

ldr      r5, [r3], #4
cmp      r1, r3
mla      r0, r4, r5, r0
mul      r4, r2, r4
bne      .L3

...
```



Pipelining keeps CPU busy through instruction-level parallelism

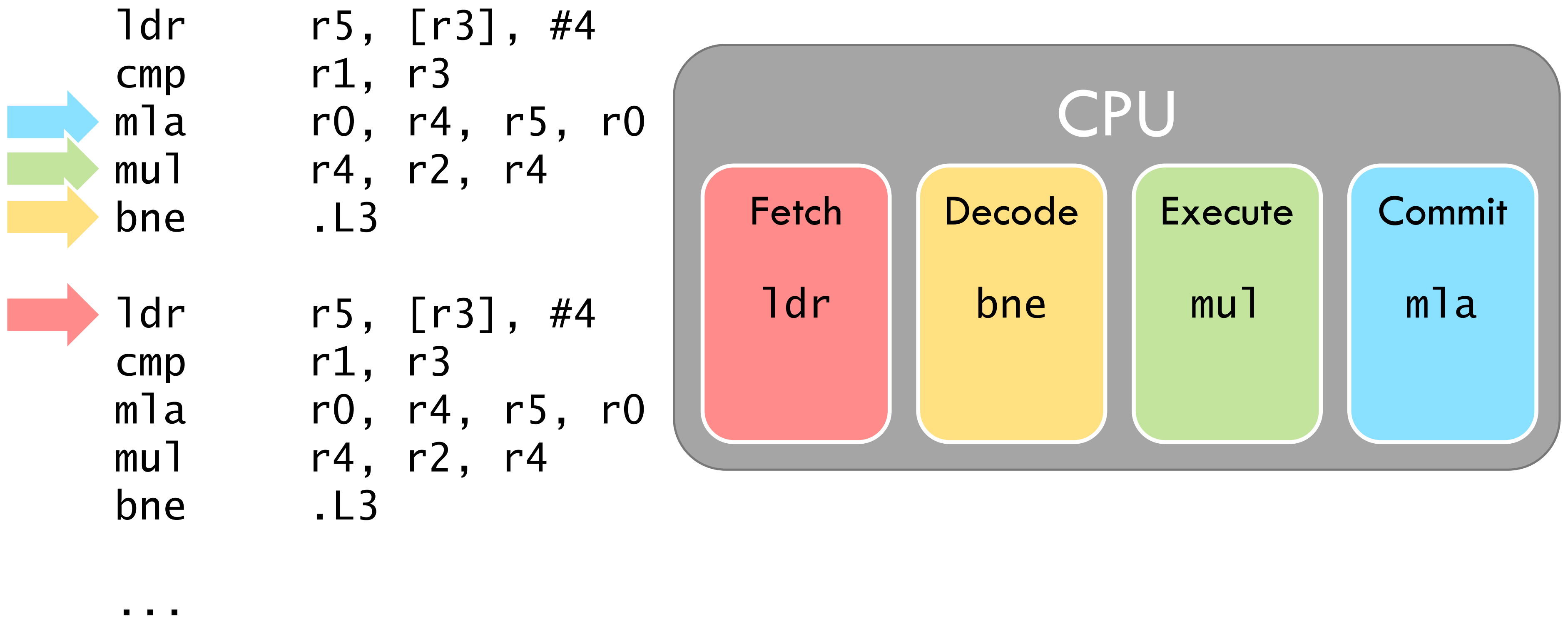
- Idea: Start on the next instr'n immediately

```
    ldr     r5, [r3], #4
    cmp     r1, r3
    mla     r0, r4, r5, r0
    mul     r4, r2, r4
    bne     .L3

    ldr     r5, [r3], #4
    cmp     r1, r3
    mla     r0, r4, r5, r0
    mul     r4, r2, r4
    bne     .L3

    ...
```



Pipelining keeps CPU busy through instruction-level parallelism

- Idea: Start on the next instr'n immediately



```
ldr      r5, [r3], #4
cmp      r1, r3
mla      r0, r4, r5, r0
mul      r4, r2, r4
bne      .L3

ldr      r5, [r3], #4
cmp      r1, r3
mla      r0, r4, r5, r0
mul      r4, r2, r4
bne      .L3

...
```

Evaluating polynomial on the pipelined CPU

How fast is this processor?
Latency? Throughput?

1 ns

TIME

Fetch	`ldr`	`cmp`	`mla`	`mul`	`bne`	`ldr`	`cmp`	`mla`	`mul`	`bne`
Decode	🟥	`ldr`	`cmp`	`mla`	`mul`	`bne`	`ldr`	`cmp`	`mla`	`mul`
Execute	🟥	🟥	`ldr`	`cmp`	`mla`	`mul`	`bne`	`ldr`	`cmp`	`mla`
Commit	🟥	🟥	🟥	`ldr`	`cmp`	`mla`	`mul`	`bne`	`ldr`	`cmp`

…

Latency = 4 ns / instr

Throughput = 1 instr / ns

4X speedup!

Speedup achieved through **pipeline parallelism**

TIME →

Processor works on 4 instructions at a time

Fetch	ldr	cmp	mla	mul	bne	ldr	cmp	mla	mul	bne
Decode	🟥	ldr	cmp	mla	mul	bne	ldr	cmp	mla	mul
Execute	🟥	🟥	ldr	cmp	mla	mul	bne	ldr	cmp	mla
Commit	🟥	🟥	🟥	ldr	cmp	mla	mul	bne	ldr	cmp

...

Limitations of pipelining

- Parallelism requires **independent** work
- Q: Are instructions independent?
- A: No! Many possible *hazards* limit parallelism…

Data hazards

```
ldr ra, [rb], #4 // ra ← Memory[rb]; rb ← rb + 4
cmp rc, rd       // rc ← rd + re
```

Q: When can the CPU pipeline the `cmp` behind `ldr`?

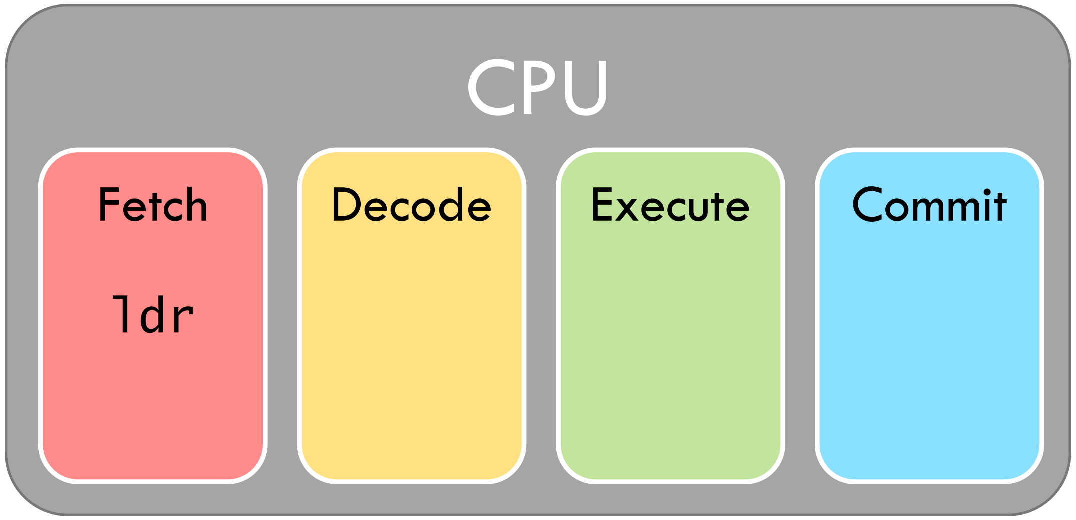
Fetch	`ldr`	`cmp`	…	…	…	…
Decode		`ldr`	`cmp`	…	…	…
Execute			`ldr`	`cmp`	…	…
Commit				`ldr`	`cmp`	…

- A: When they use **different registers**
  - Specifically, when `cmp` does not read any data written by `ldr`
  - E.g., `rb != rd`

Dealing with data hazards: Stalling the pipeline

- Cannot pipeline `cmp` (`ldr` writes `r3`)

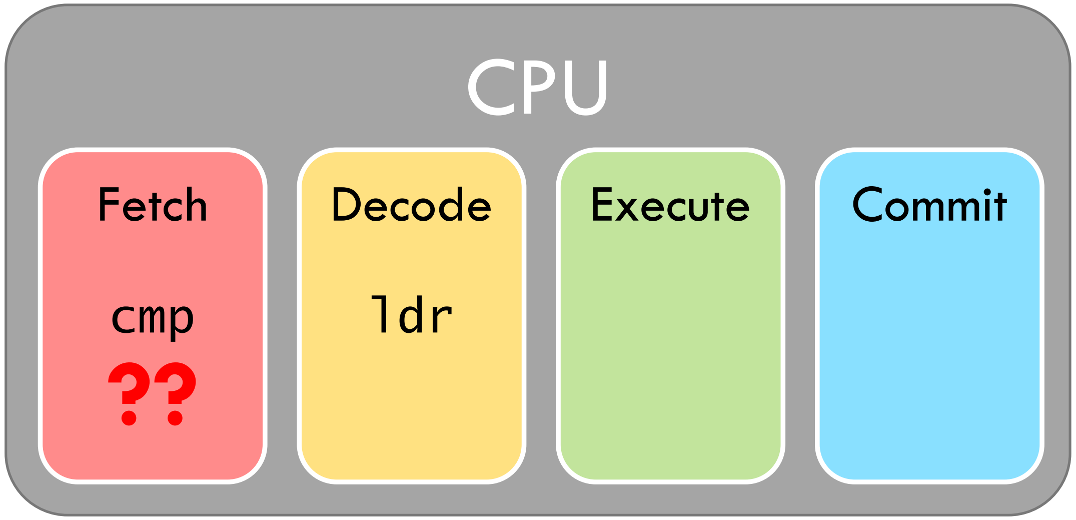
```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

...
```



Dealing with data hazards: Stalling the pipeline

- Cannot pipeline `cmp` (`ldr` writes `r3`)

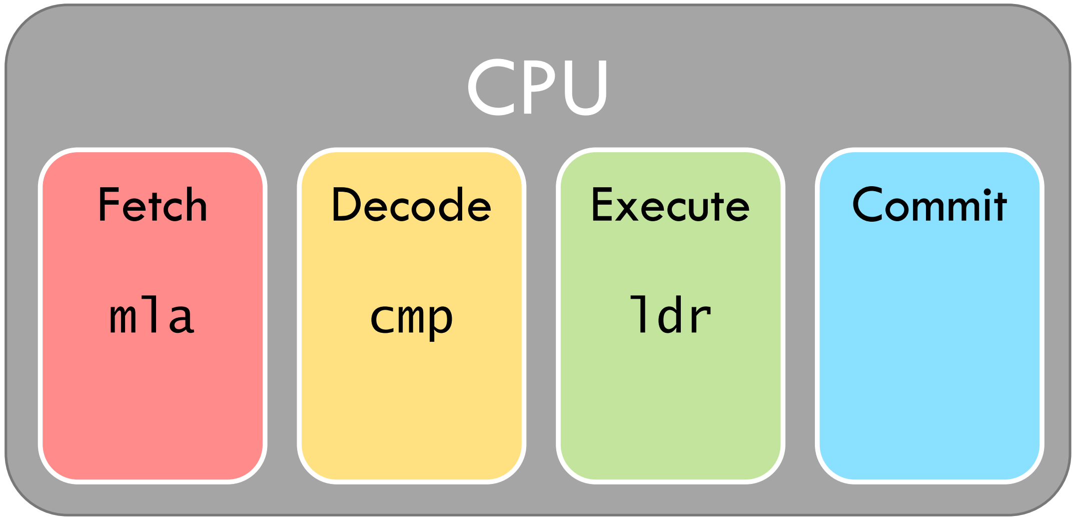
```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

...
```



Dealing with data hazards: Stalling the pipeline

- Cannot pipeline `cmp` (`ldr` writes `r3`)

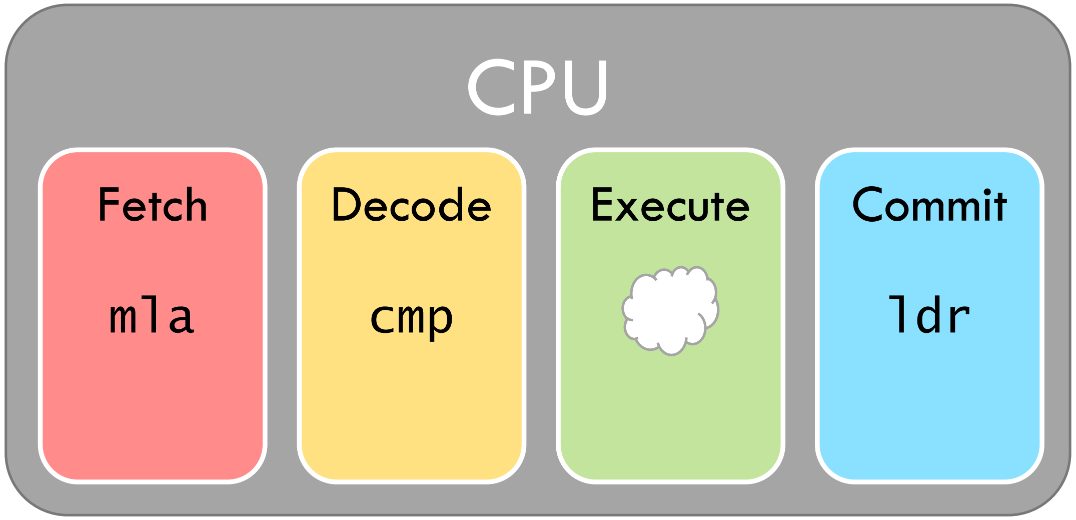
```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

...
```



Dealing with data hazards: Stalling the pipeline

- Cannot pipeline `cmp` (`ldr` writes `r3`)

```
ldr      r5, [r3], #4
cmp      r1, r3
mla      r0, r4, r5, r0
mul      r4, r2, r4
bne      .L3

ldr      r5, [r3], #4
cmp      r1, r3
mla      r0, r4, r5, r0
mul      r4, r2, r4
bne      .L3

...
```

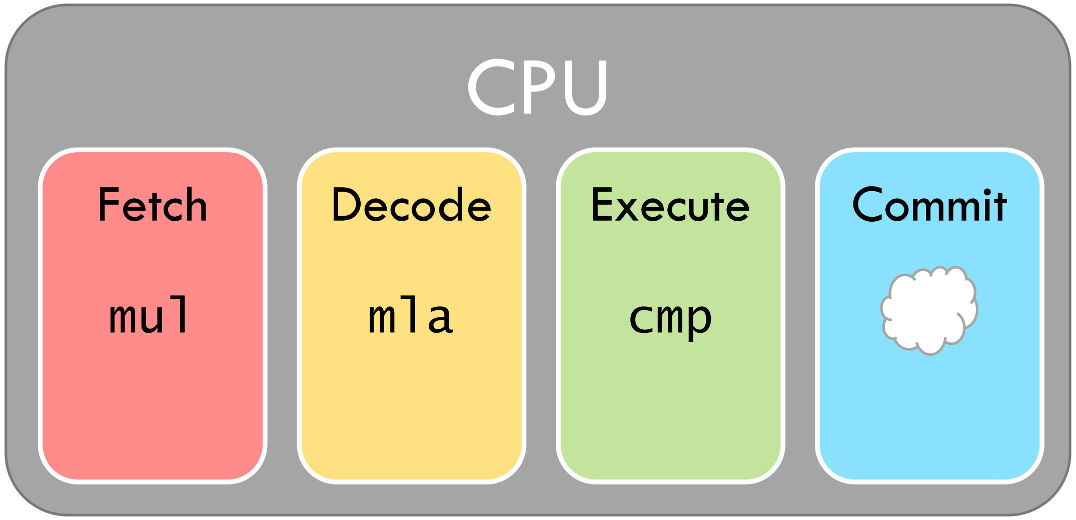


Inject a "bubble" (NOP) into the pipeline

Dealing with data hazards: Stalling the pipeline

- Cannot pipeline `cmp` (`ldr` writes `r3`)

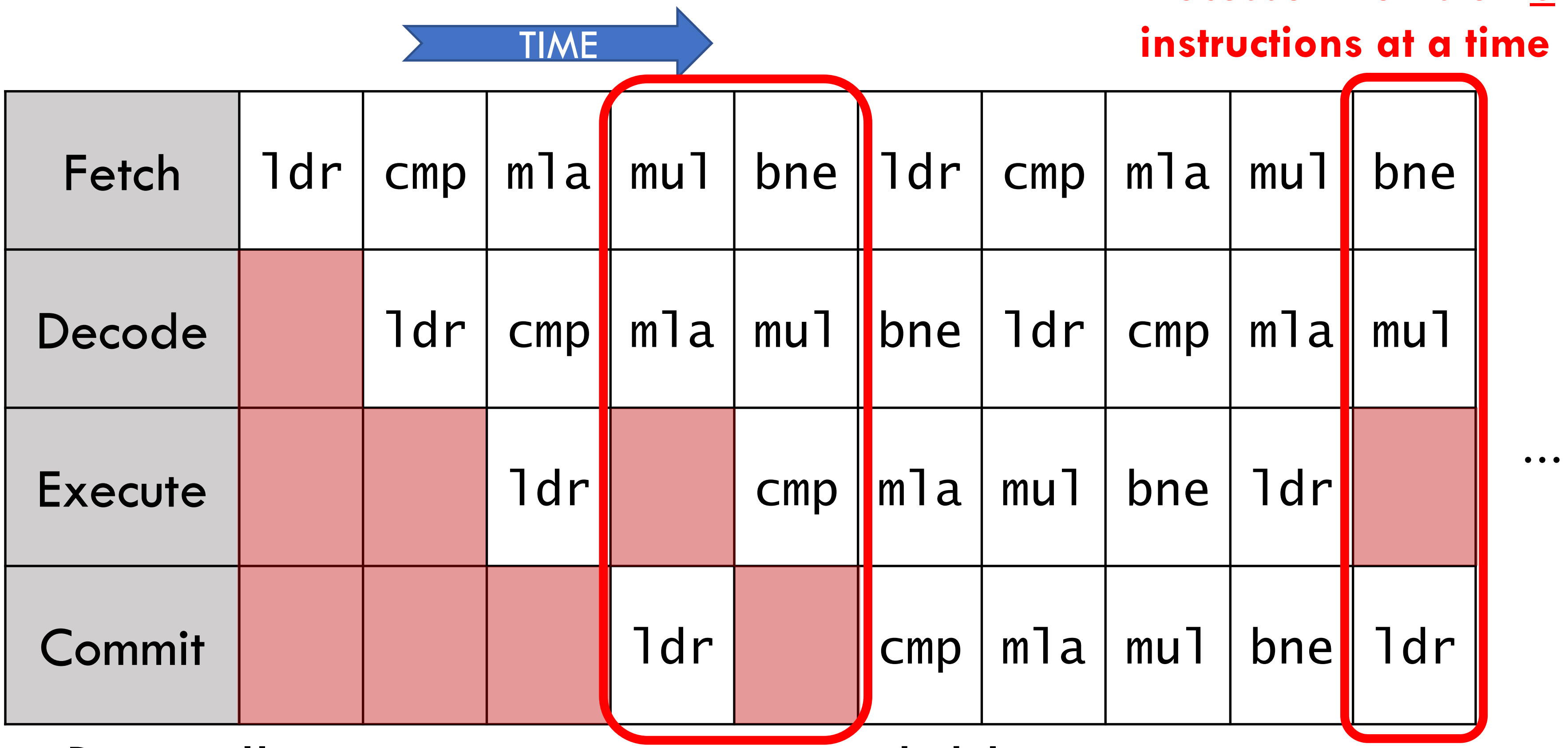
```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

...
```



`cmp` proceeds once `ldr` has committed

Stalling degrades performance

Processor works on <u>3</u> instructions at a time



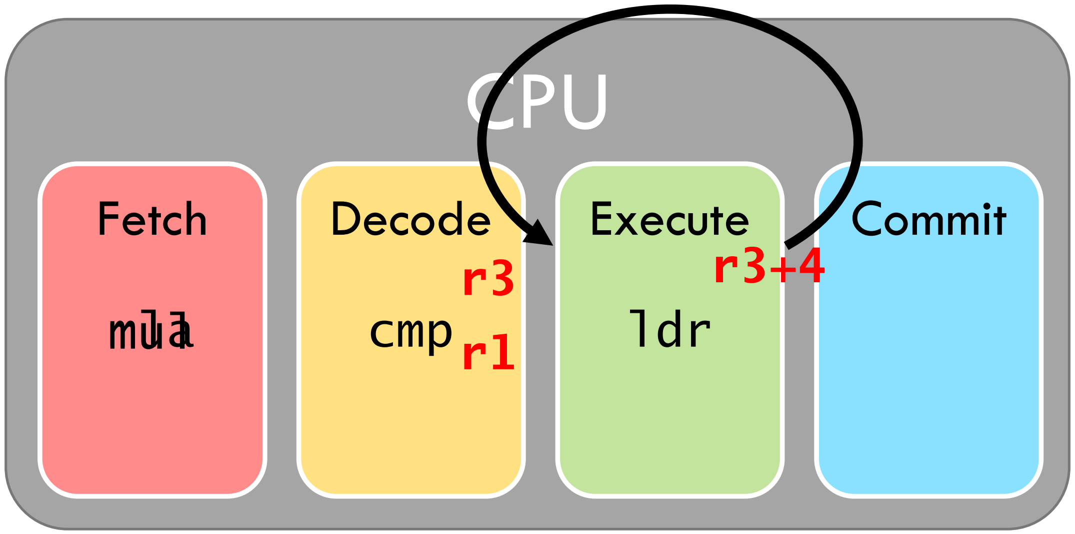
Fetch	ldr	cmp	mla	mul	bne	ldr	cmp	mla	mul	bne
Decode	🟥	ldr	cmp	mla	mul	bne	ldr	cmp	mla	mul
Execute	🟥	🟥	ldr	🟥	cmp	mla	mul	bne	ldr	🟥
Commit	🟥	🟥	🟥	ldr	🟥	cmp	mla	mul	bne	ldr

...

- But stalling is sometimes unavoidable
  - E.g., long-latency instructions (divide, cache miss)

Dealing with data hazards: Forwarding data

- Wait a second… data is available after Execute!



- Forwarding eliminates many (not all) pipeline stalls

Speedup achieved through **pipeline parallelism**

Processor works on 4 instructions at a time ☺

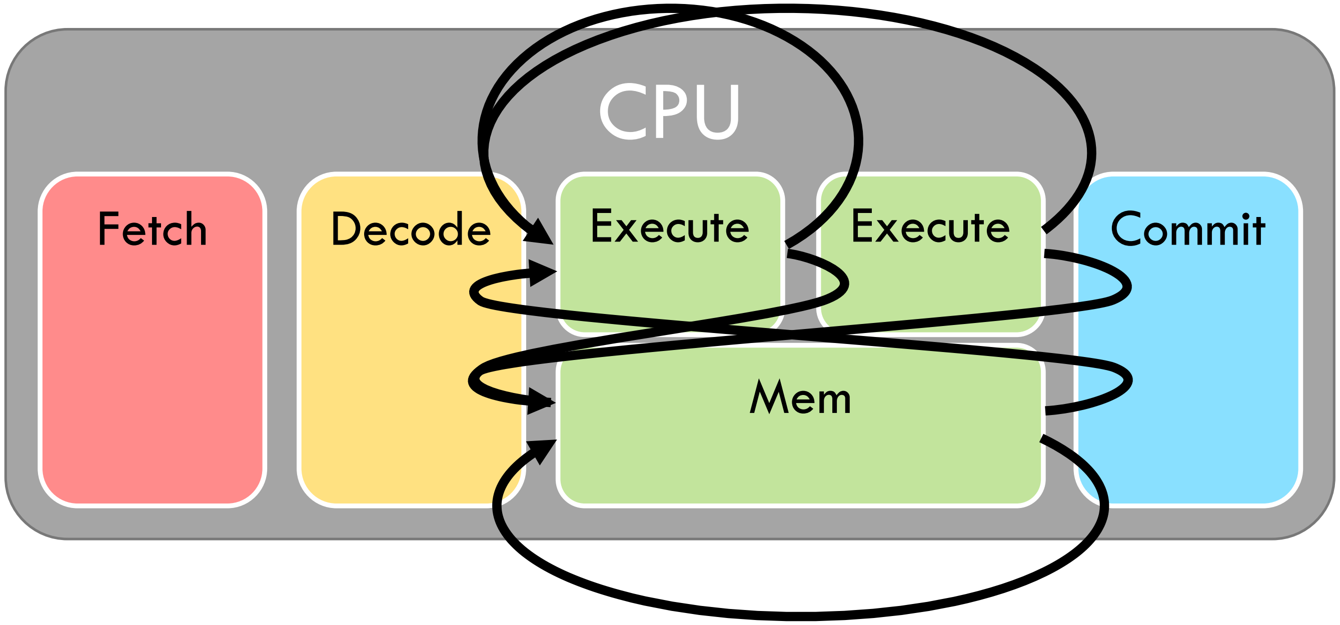
TIME →

Fetch	`ldr`	`cmp`	`mla`	`mul`	`bne`	`ldr`	`cmp`	`mla`	`mul`	`bne`
Decode		`ldr`	`cmp`	`mla`	`mul`	`bne`	`ldr`	`cmp`	`mla`	`mul`
Execute			`ldr`	`cmp`	`mla`	`mul`	`bne`	`ldr`	`cmp`	`mla`
Commit				`ldr`	`cmp`	`mla`	`mul`	`bne`	`ldr`	`cmp`

...

Pipelining is not free!

- Q: How well does forwarding scale?
- A: Not well… many forwarding paths in deep & complex pipelines



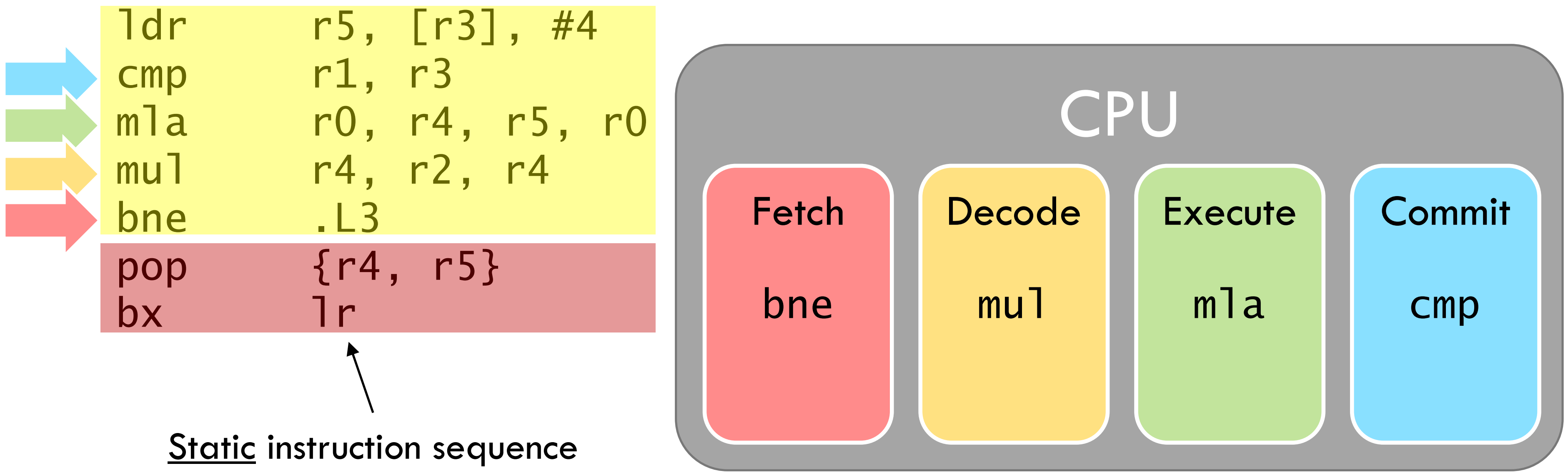
Control hazards + Speculation

- Programs must appear to execute *in program order*
  ➔ All instructions depend on earlier ones

- Most instructions implicitly continue at the next…
- But **branches** redirect execution to new location

Dealing with control hazards: Flushing the pipeline

- What if we always fetch the next instruction?

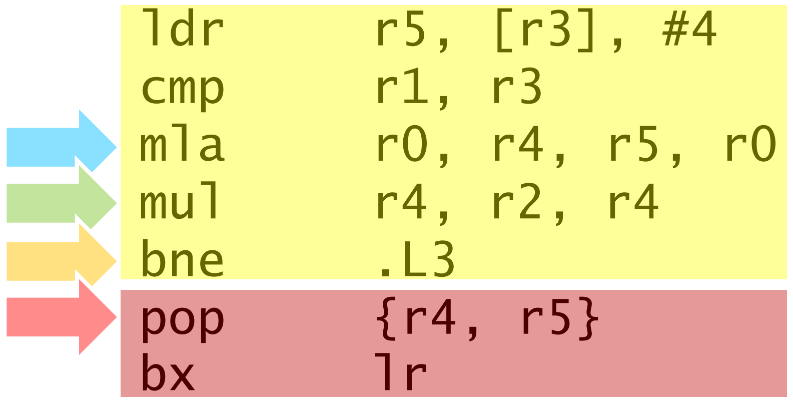


```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
pop     {r4, r5}
bx      lr
```

<u>Static</u> instruction sequence
(i.e., program layout in memory)

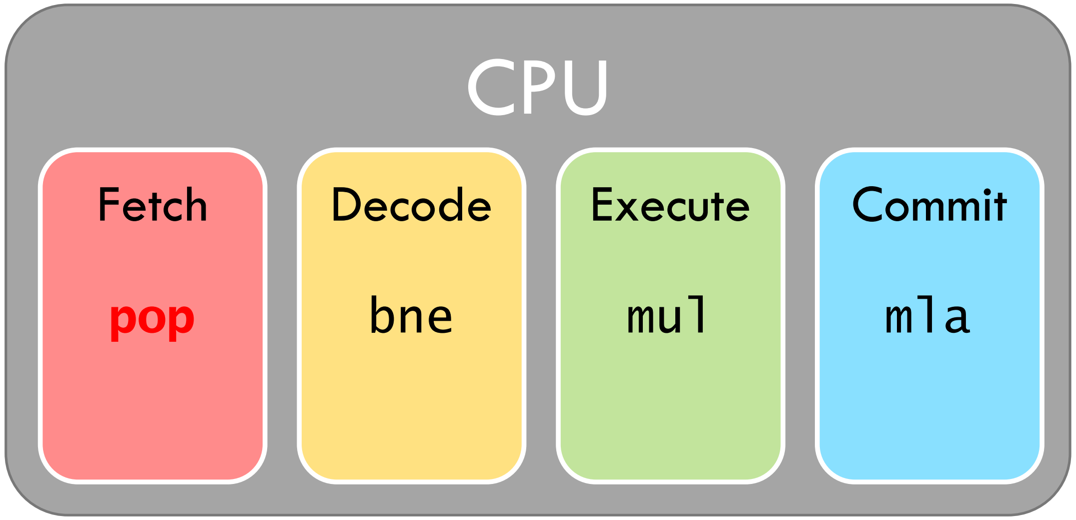
Dealing with control hazards: Flushing the pipeline

- What if we always fetch the next instruction?



```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
pop     {r4, r5}
bx      lr
```

Static instruction sequence
(i.e., program layout in memory)

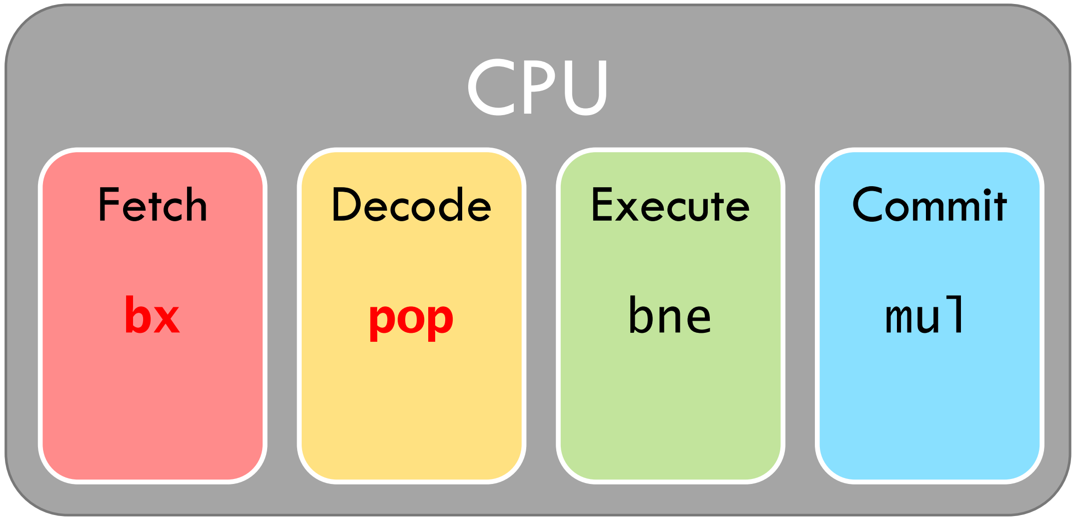


Dealing with control hazards: Flushing the pipeline

- What if we always fetch the next instruction?

```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
pop     {r4, r5}
bx      lr
```

Static instruction sequence (i.e., program layout in memory)



Whoops! We fetched the wrong instructions! (Loop not finished)

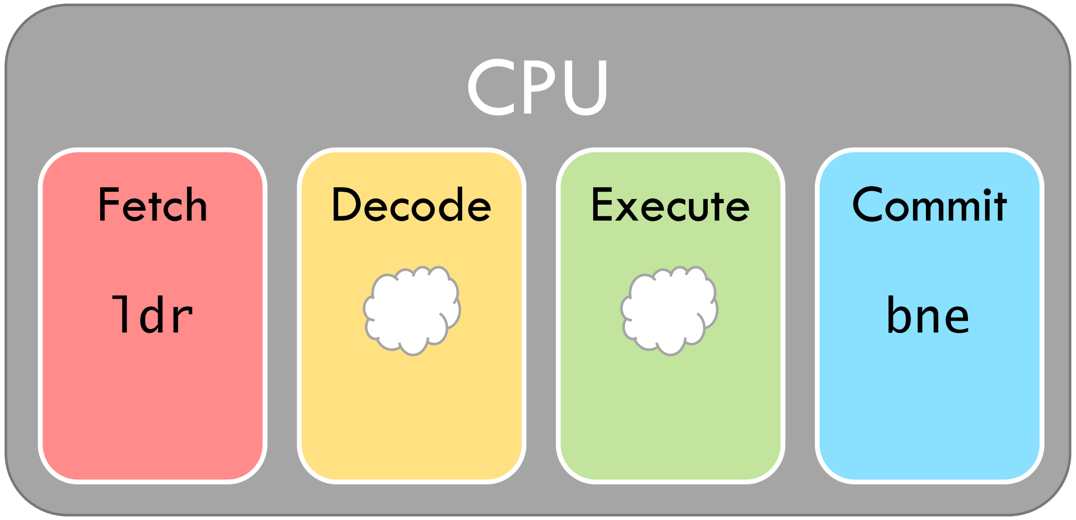
Dealing with control hazards: Flushing the pipeline

(Next loop iteration)

- What if we always fetch the next instruction?

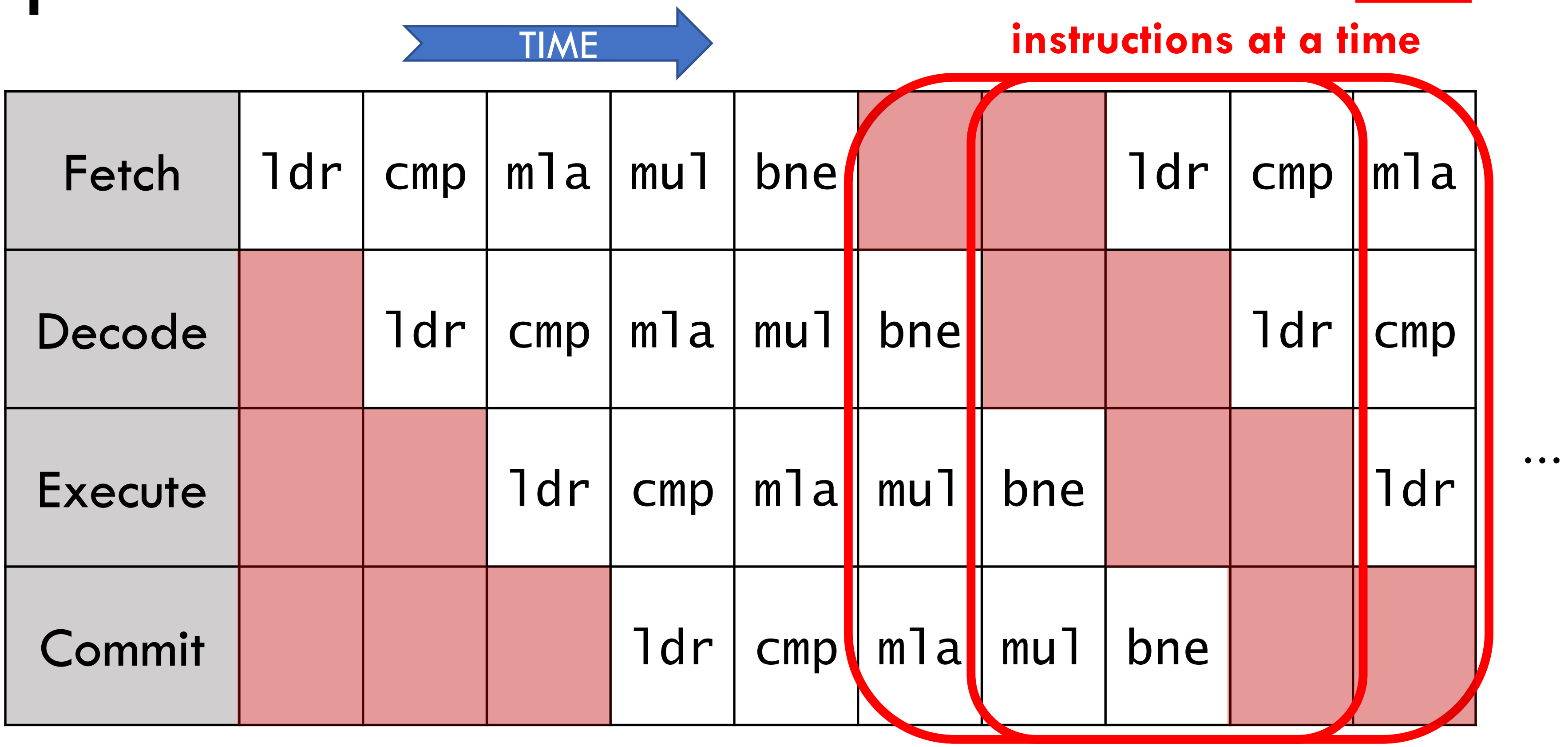
```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
pop     {r4, r5}
bx      lr
```

<u>Static</u> instruction sequence
(i.e., program layout in memory)



Whoops! We fetched the wrong instructions! (Loop not finished)

Pipeline flushes destroy performance



Fetch	ldr	cmp	mla	mul	bne			ldr	cmp	mla
Decode		ldr	cmp	mla	mul	bne			ldr	cmp
Execute			ldr	cmp	mla	mul	bne			ldr
Commit				ldr	cmp	mla	mul	bne		

- Penalty <u>increases</u> with deeper pipelines

Dealing with control hazards: *Speculation!*

- Processors do not wait for branches to execute
- Instead, they speculate (i.e., guess) where to go next + start fetching
- Modern processors use very sophisticated mechanisms
  - E.g., speculate in Fetch stage—before processor even knows instrn is a branch!
  - >95% prediction accuracy
  - Still, branch mis-speculation is major problem

Pipelining Summary

- Pipelining is a simple, effective way to improve throughput
  - N-stage pipeline gives up to $N \times$ speedup

- Pipelining has limits
  - Hard to keep pipeline busy because of hazards
  - Forwarding is expensive in deep pipelines
  - Pipeline flushes are expensive in deep pipelines

➔ Pipelining is ubiquitous, but tops out at $N \approx 15$

Software Takeaways

- Processors with a simple "in-order" pipeline are very sensitive to running "good code"
  - Compiler should target a specific model of CPU
  - Low-level assembly hacking

- …But very few CPUs are in-order these days
  - E.g., embedded, ultra-low-power applications

- Instead, ≈all modern CPUs are "out-of-order"
  - Even in classic "low-power domains" (like mobile)

Out-of-Order Execution

Increasing parallelism via dataflow

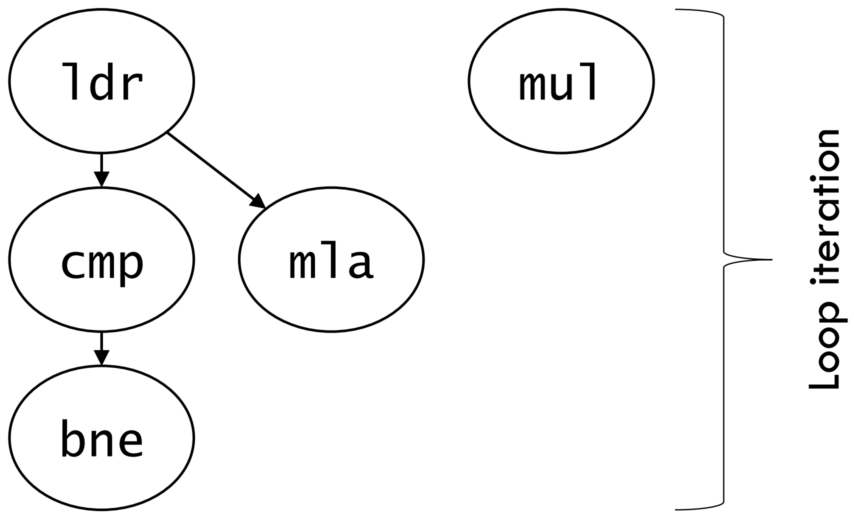
- Parallelism limited by many *false dependencies*, particularly *sequential program order*

- **<u>Dataflow</u>** tracks how instructions actually depend on each other
  - *True dependence*: read-after-write

Dataflow increases parallelism by eliminating unnecessary dependences

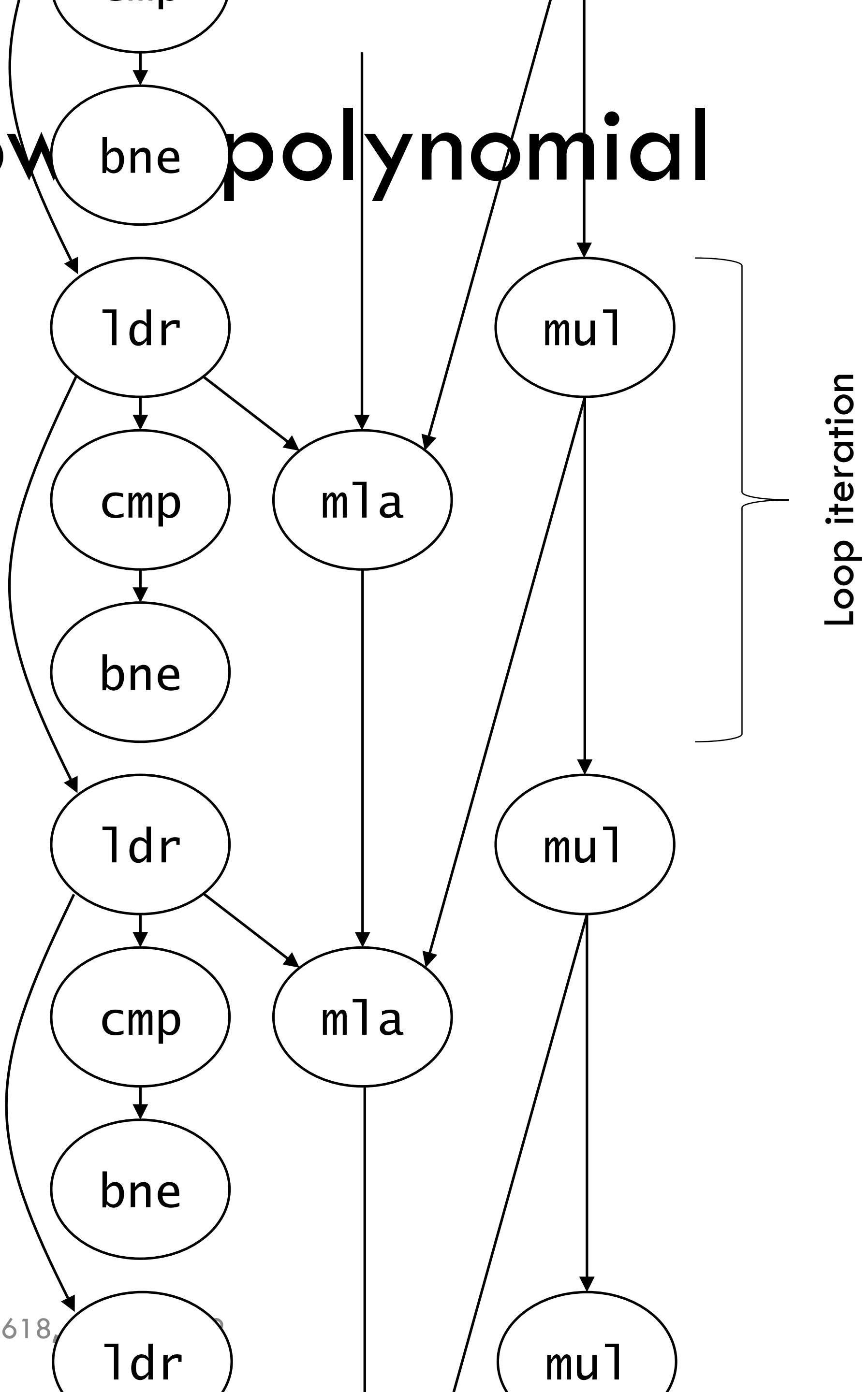
Example: Dataflow in polynomial evaluation

```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3

...
```



Example: Dataflow polynomial evaluation

```
ldr      r5, [r3], #4
cmp      r1, r3
mla      r0, r4, r5, r0
mul      r4, r2, r4
bne      .L3

ldr      r5, [r3], #4
cmp      r1, r3
mla      r0, r4, r5, r0
mul      r4, r2, r4
bne      .L3

...
```

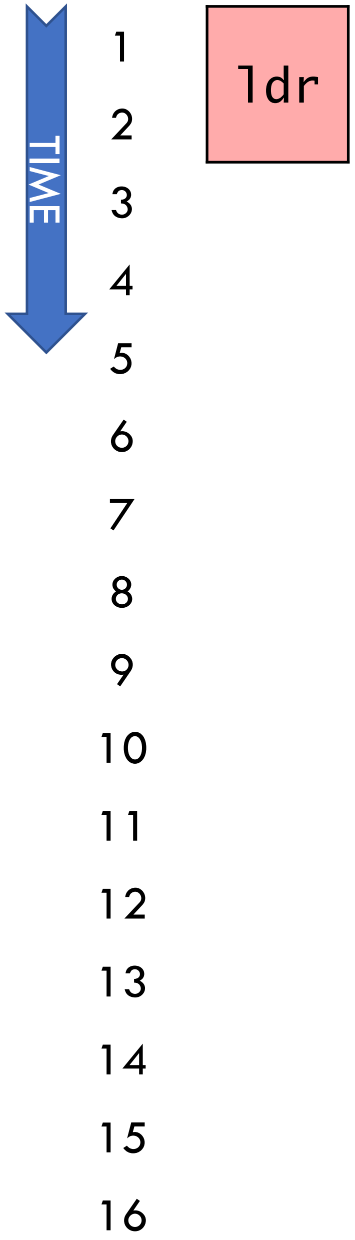


Example: Dataflow polynomial execution

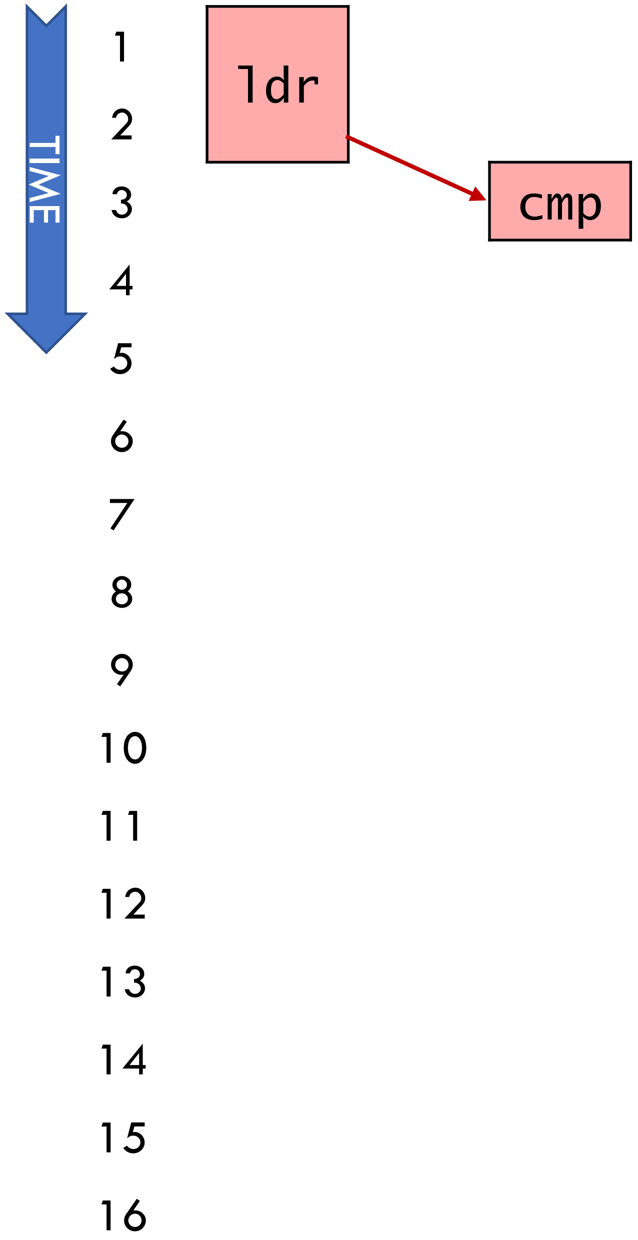
- Execution <u>only</u>, with perfect scheduling & unlimited execution units
  - `ldr`, `mul` execute in 2 cycles
  - `cmp`, `bne` execute in 1 cycle
  - `mla` executes in 3 cycles

- Q: Does dataflow speedup execution? By how much?

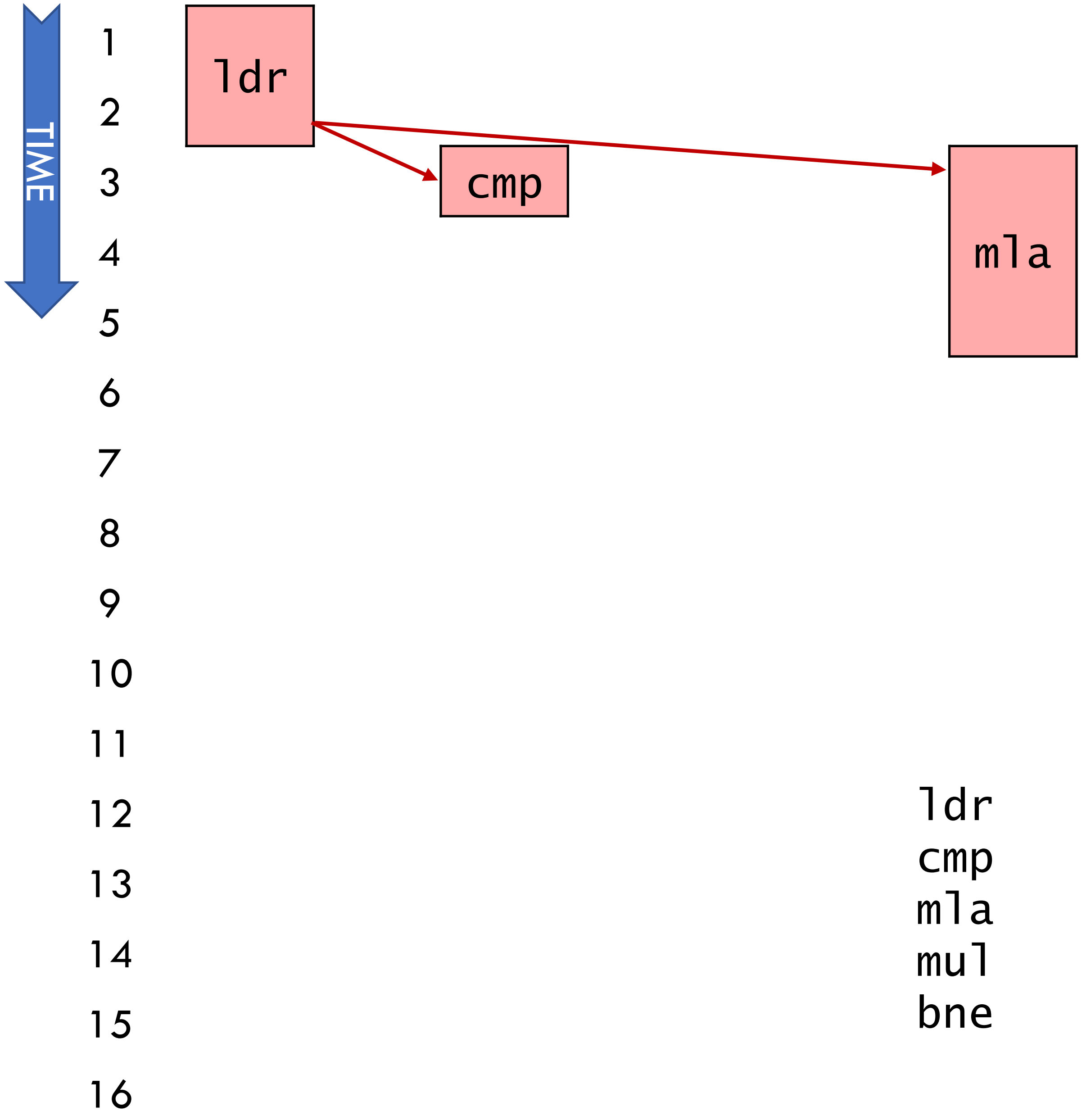
- Q: What is the performance bottleneck?



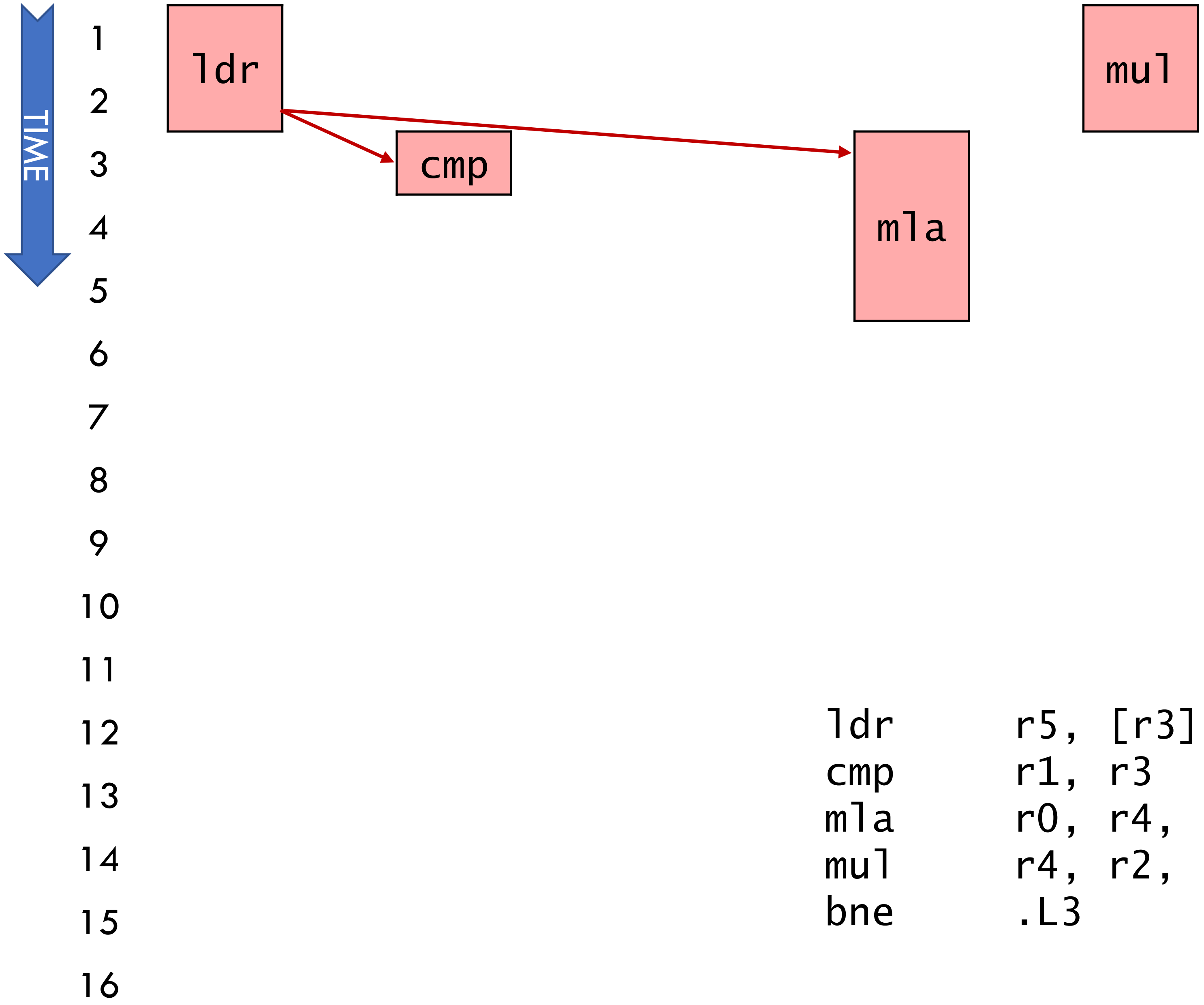
```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
```



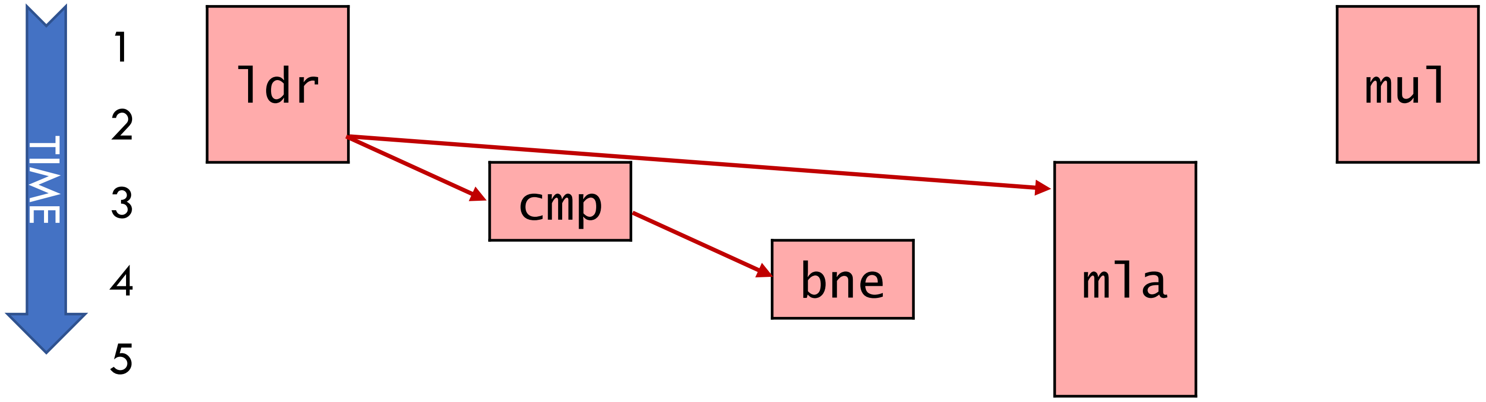
```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
```



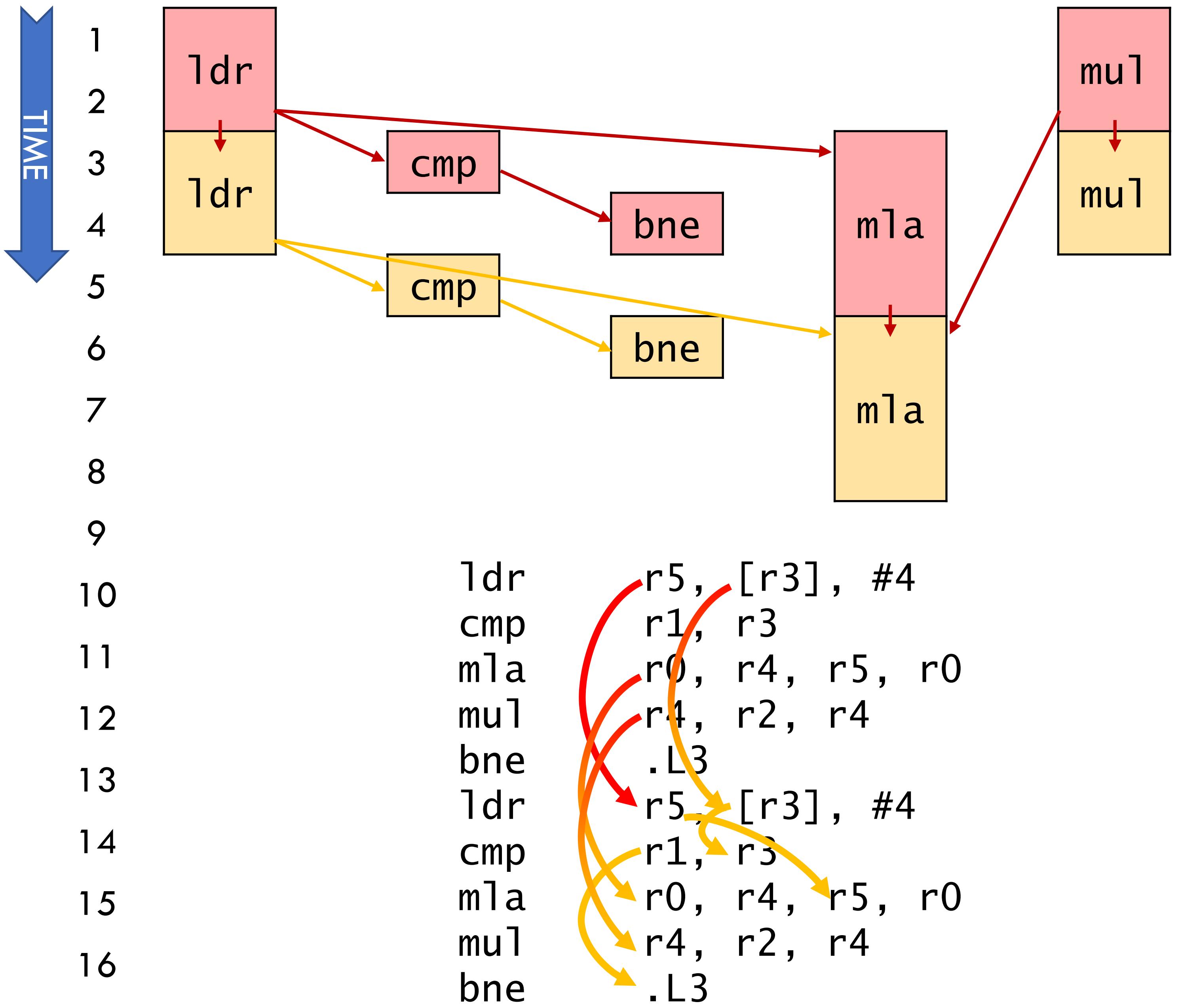
```
ldr      r5, [r3], #4
cmp      r1, r3
mla      r0, r4, r5, r0
mul      r4, r2, r4
bne      .L3
```



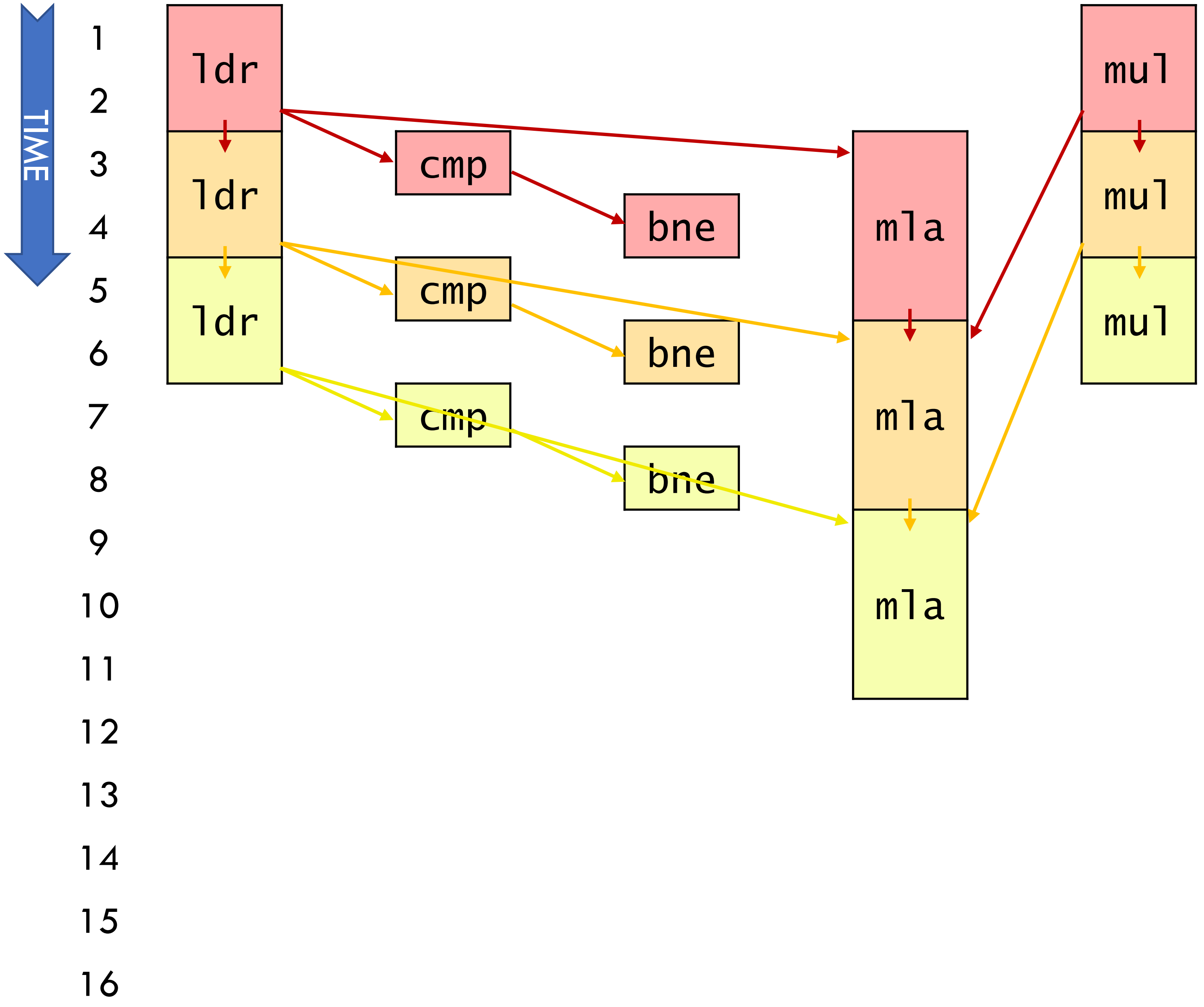
```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
```

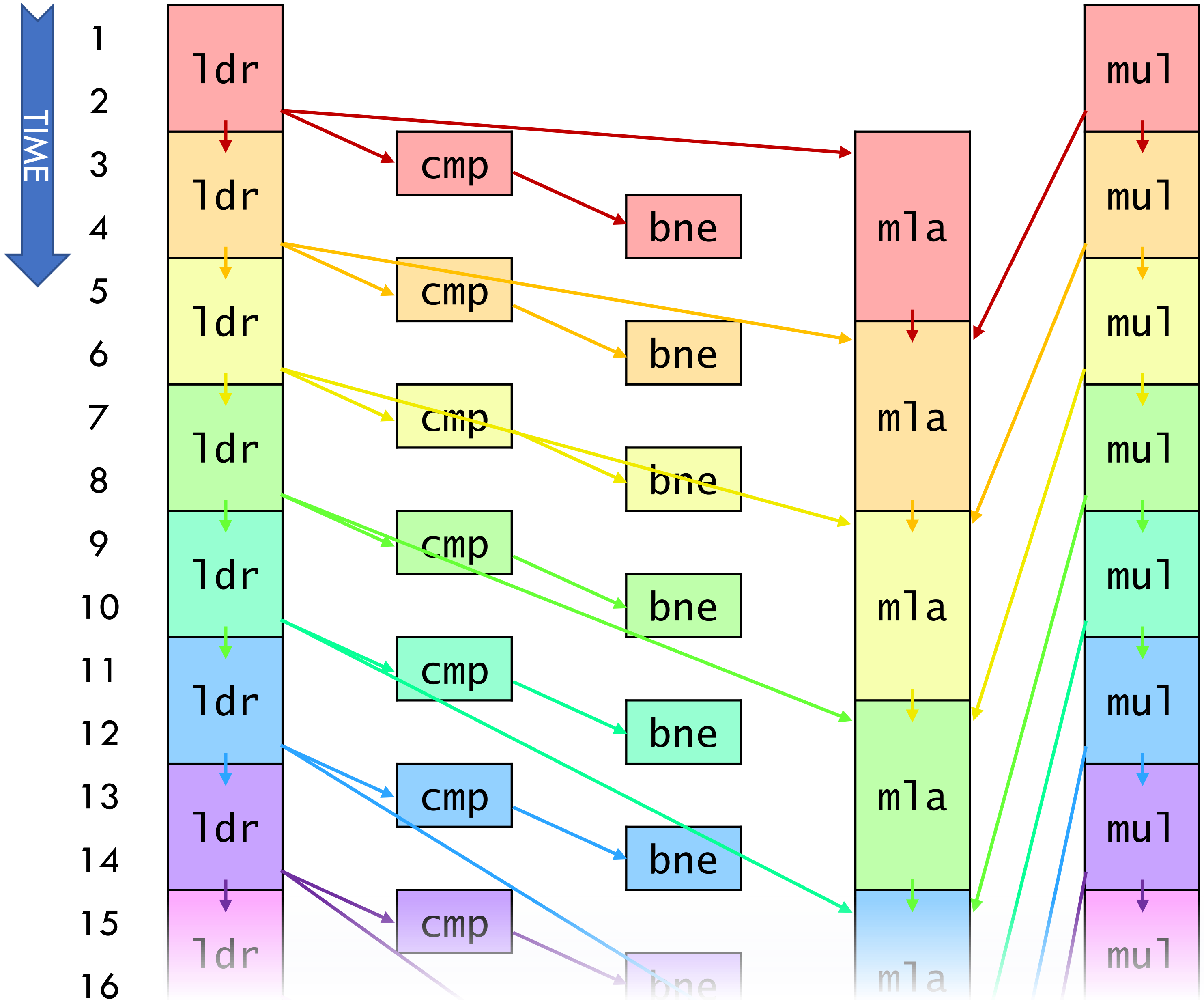


```
ldr      r5, [r3], #4
cmp      r1, r3
mla      r0, r4, r5, r0
mul      r4, r2, r4
bne      .L3
```



```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
```


TIME
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
ldr
ldr
ldr
cmp
cmp
cmp
bne
bne
bne
mla
mla
mla
mul
mul
mul


TIME
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
ldr
ldr
ldr
ldr
ldr
ldr
ldr
ldr
cmp
cmp
cmp
cmp
cmp
cmp
cmp
bne
bne
bne
bne
bne
bne
bne
mla
mla
mla
mla
mla
mul
mul
mul
mul
mul
mul
mul
mul

Example: Dataflow polynomial execution

- Q: Does dataflow speedup execution? By how much?
  - Yes! 3 cycles / loop iteration
  - Instructions per cycle (IPC) = 5/3 ≈ 1.67
    (vs. 1 for perfect pipelining)

- Q: What is the performance bottleneck?
  - `mla`: Each `mla` depends on previous `mla` & takes 3 cycles
  - ➔ This program is **latency-bound**

Latency Bound

- What is the "critical path" of the computation?
  - Longest path across iterations in dataflow graph
  - E.g., `mla` in last slide (but could be multiple ops)

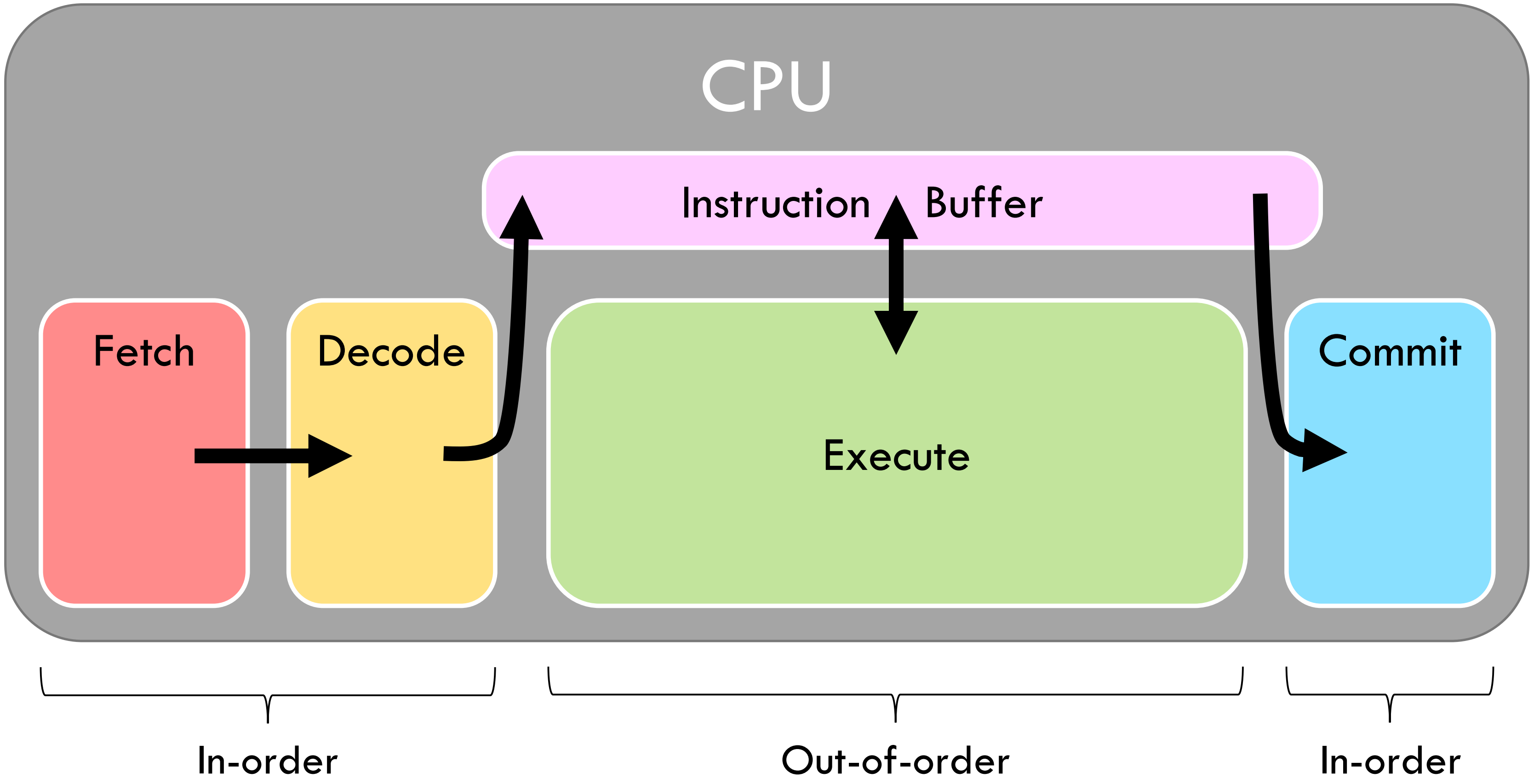
- Critical path limits maximum performance
- Real CPUs may not achieve latency bound, but useful mental model + tool for program analysis

Out-of-order (OoO) execution uses dataflow to increase parallelism

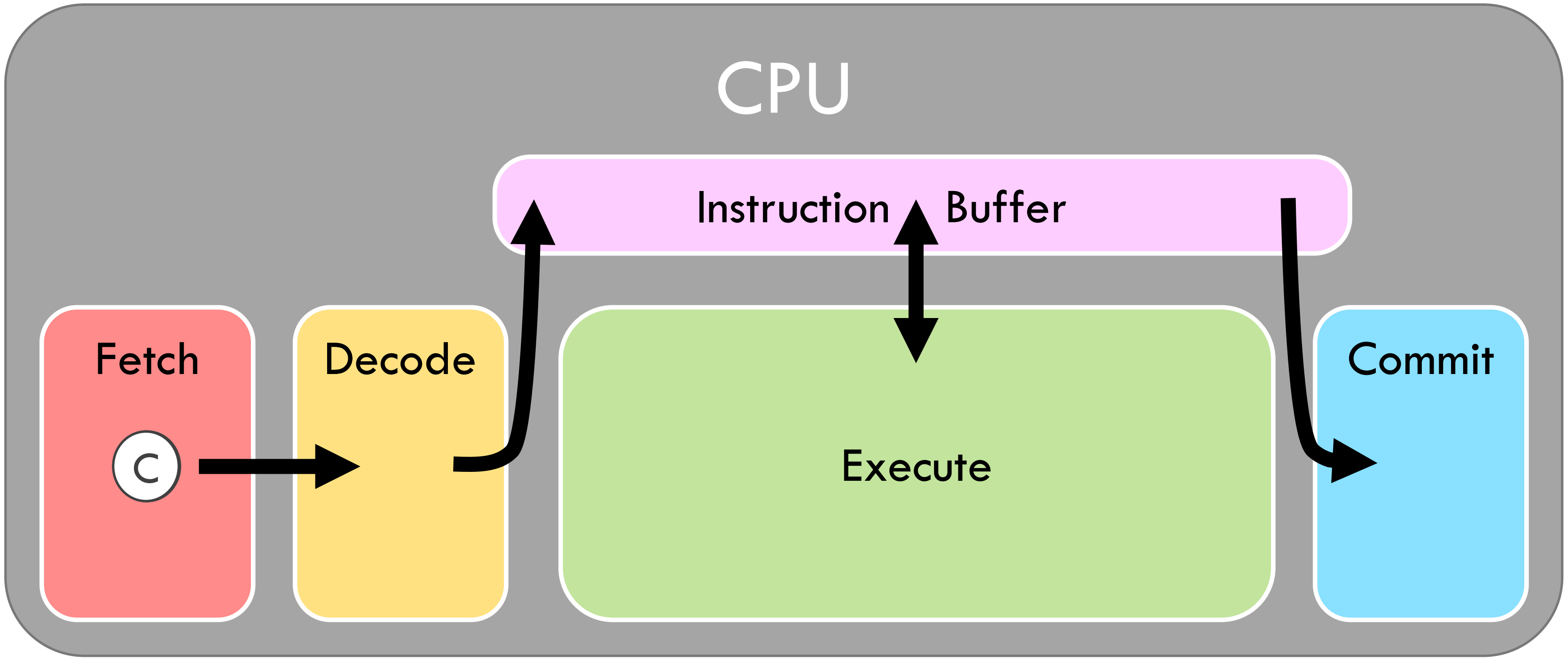
- Idea: Execute programs in dataflow order, but give the *illusion* of sequential execution

- This is a "restricted dataflow" model
  - *Restricted* to instructions near those currently committing
  - (Pure dataflow processors also exist that expose dataflow to software)

High-level OoO microarchitecture



OoO is **hidden** behind in-order frontend & commit

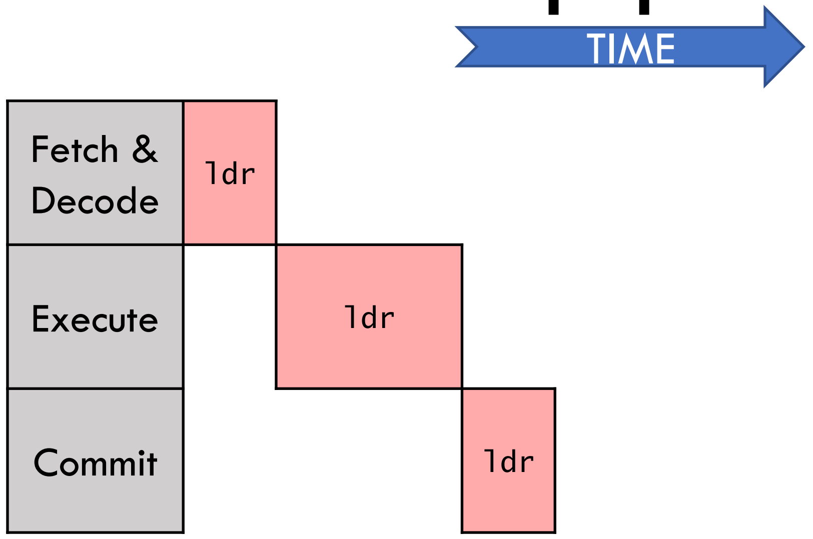


- Instructions only enter & leave instruction buffer in program order; all bets are off in between!

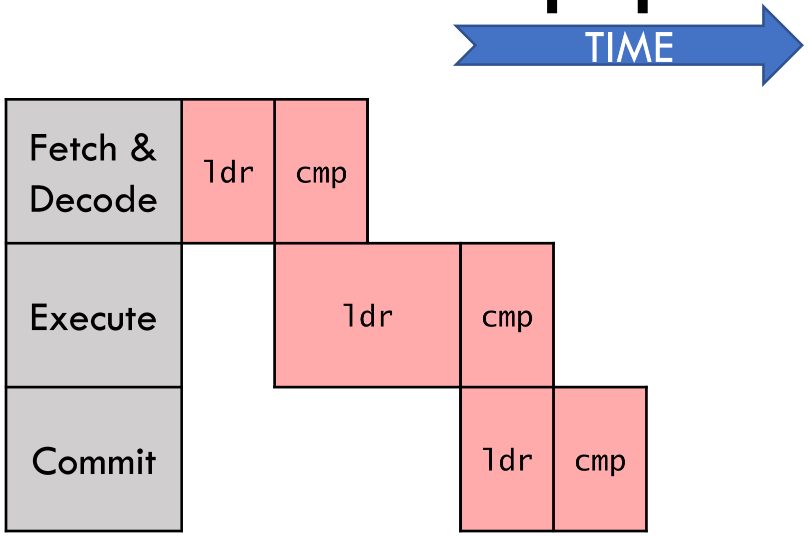
Example: OoO polynomial evaluation

- Q: Does OoO speedup execution? By how much?
- Q: What is the performance bottleneck?
- Assume perfect forwarding & branch prediction

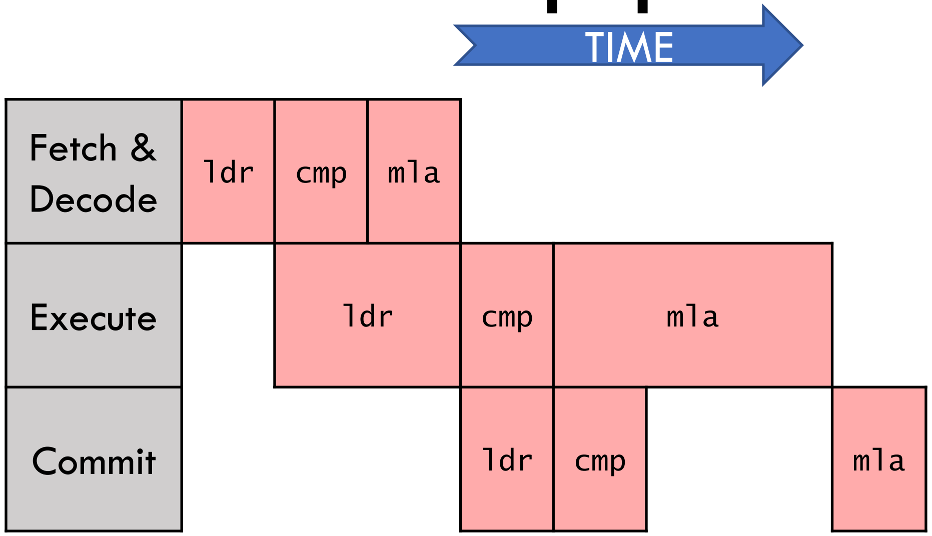
Example: OoO polynomial evaluation pipeline diagram



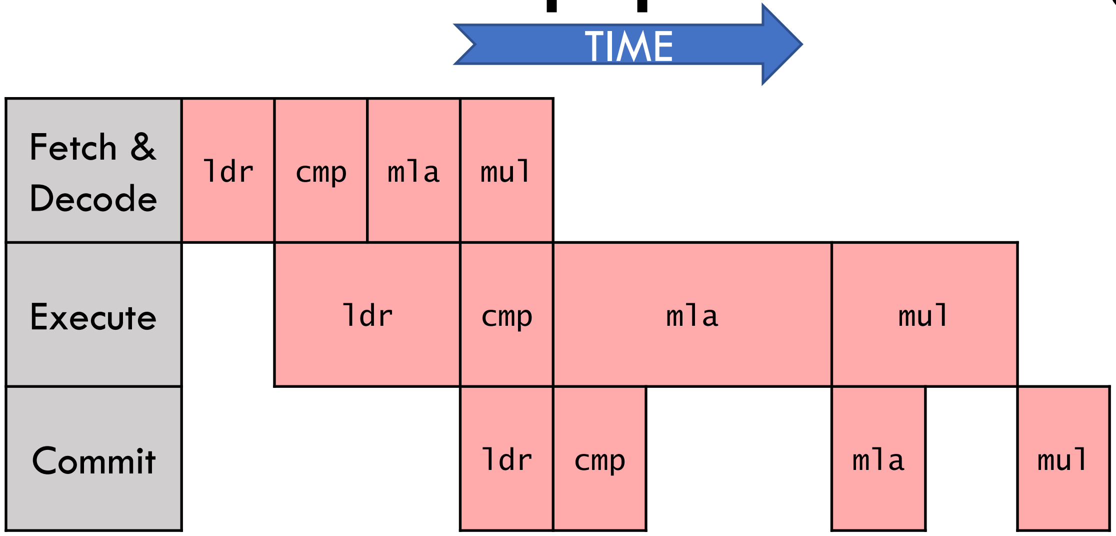
Example: OoO polynomial evaluation pipeline diagram



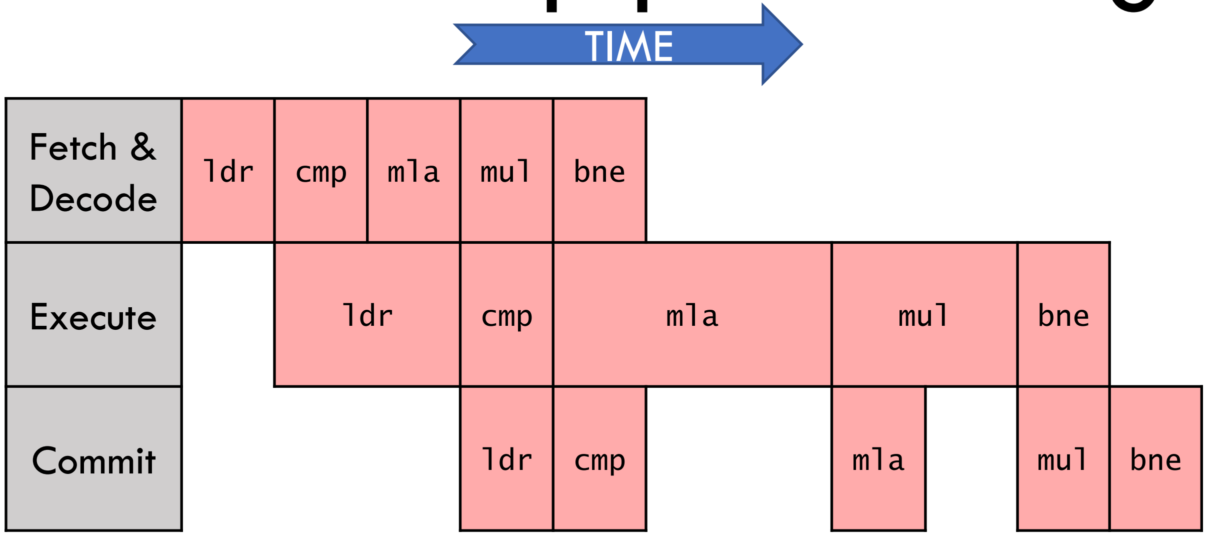
Example: OoO polynomial evaluation pipeline diagram



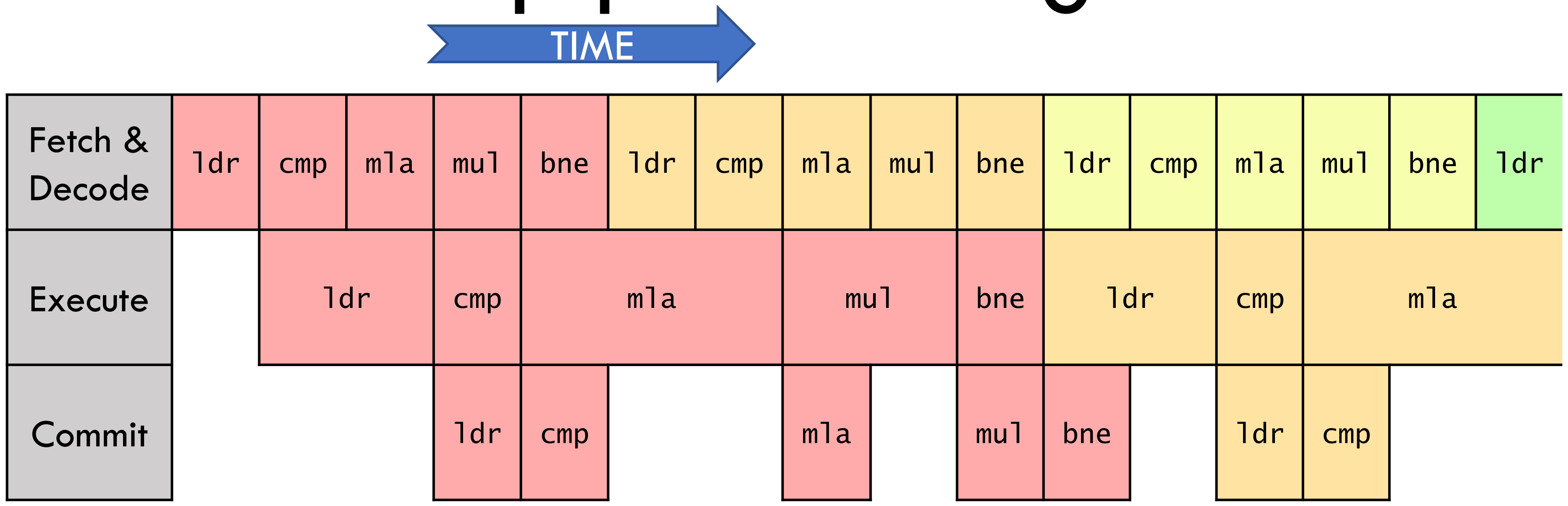
Example: OoO polynomial evaluation pipeline diagram



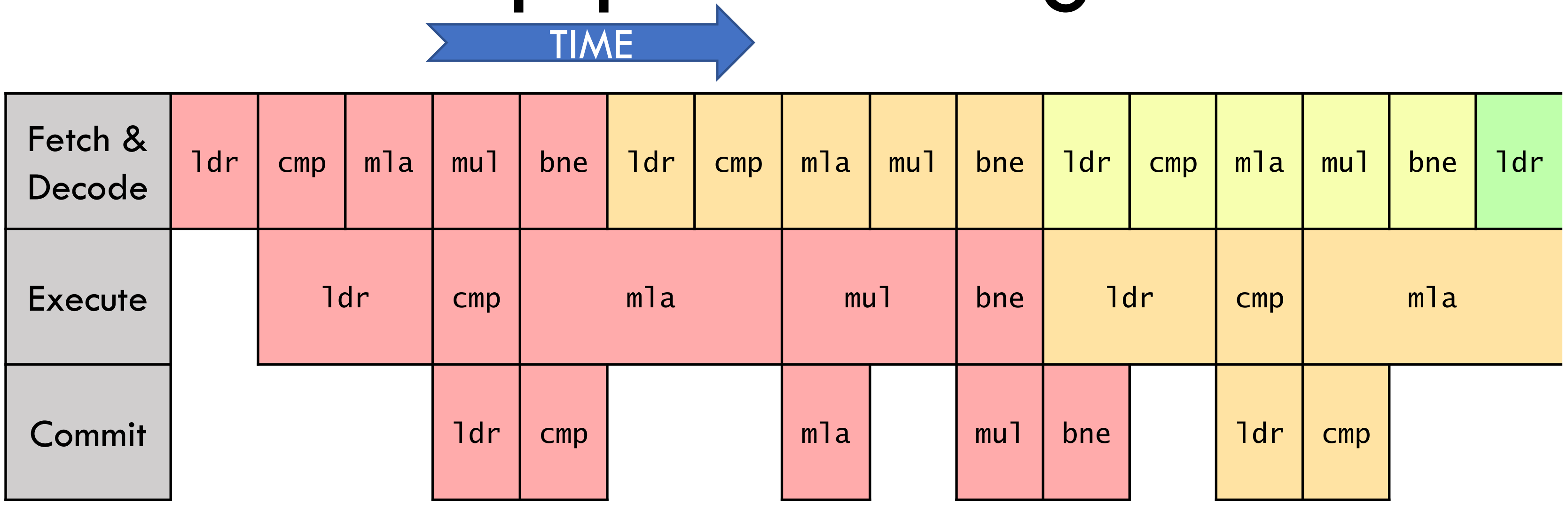
Example: OoO polynomial evaluation pipeline diagram



Example: OoO polynomial evaluation pipeline diagram



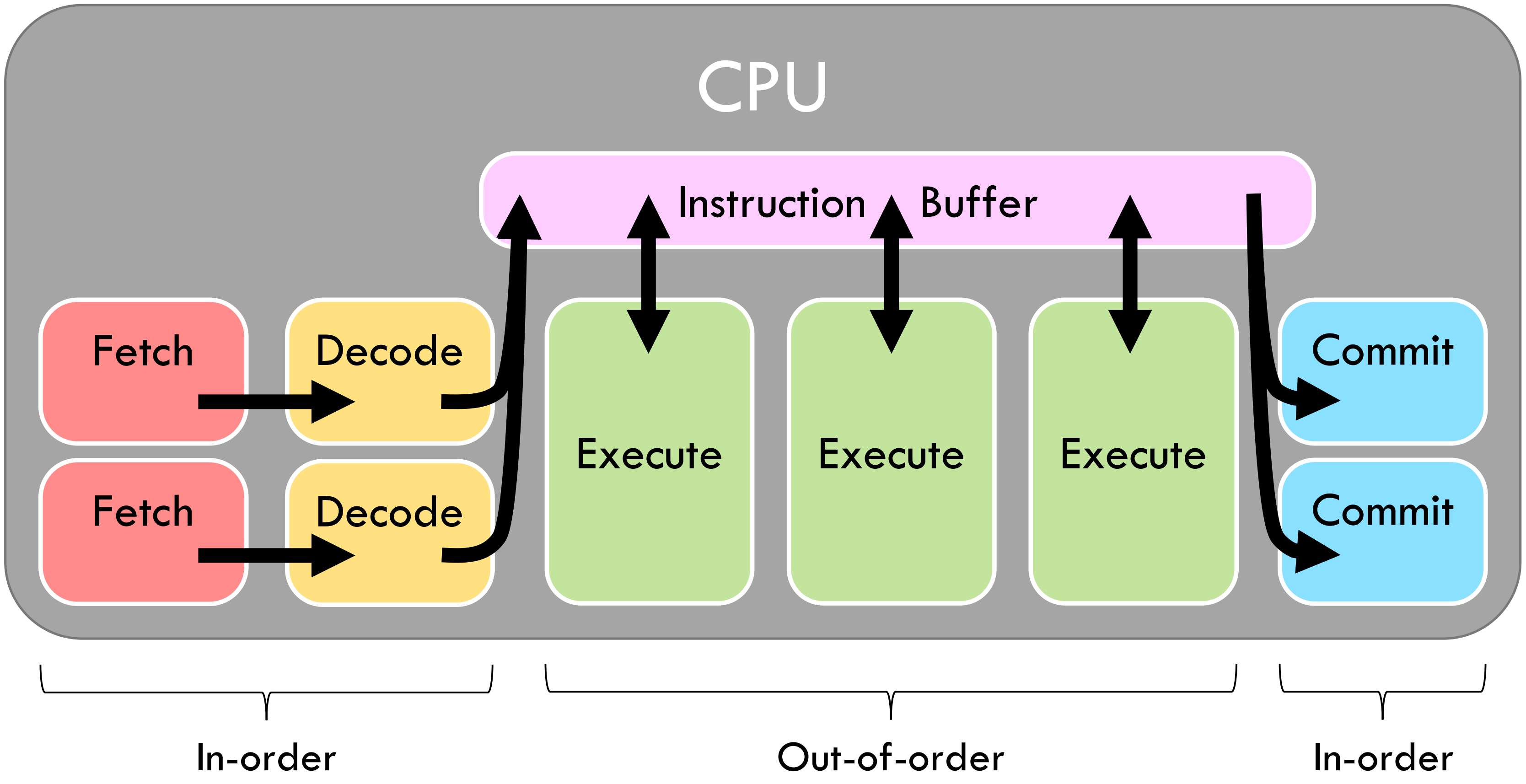
Example: OoO polynomial evaluation pipeline diagram


TIME
Fetch & Decode
ldr cmp mla mul bne ldr cmp mla mul bne ldr cmp mla mul bne ldr
Execute
ldr cmp mla mul bne ldr cmp mla
Commit
ldr cmp mla mul bne ldr cmp

- Wait a minute… this isn't OoO… or even faster than a simple pipeline!
- Q: What went wrong?
- A: We're **throughput-limited:** can only exec 1 instrn

High-level **Superscalar** OoO microarchitecture

- Must increase *pipeline width* to increase ILP > 1

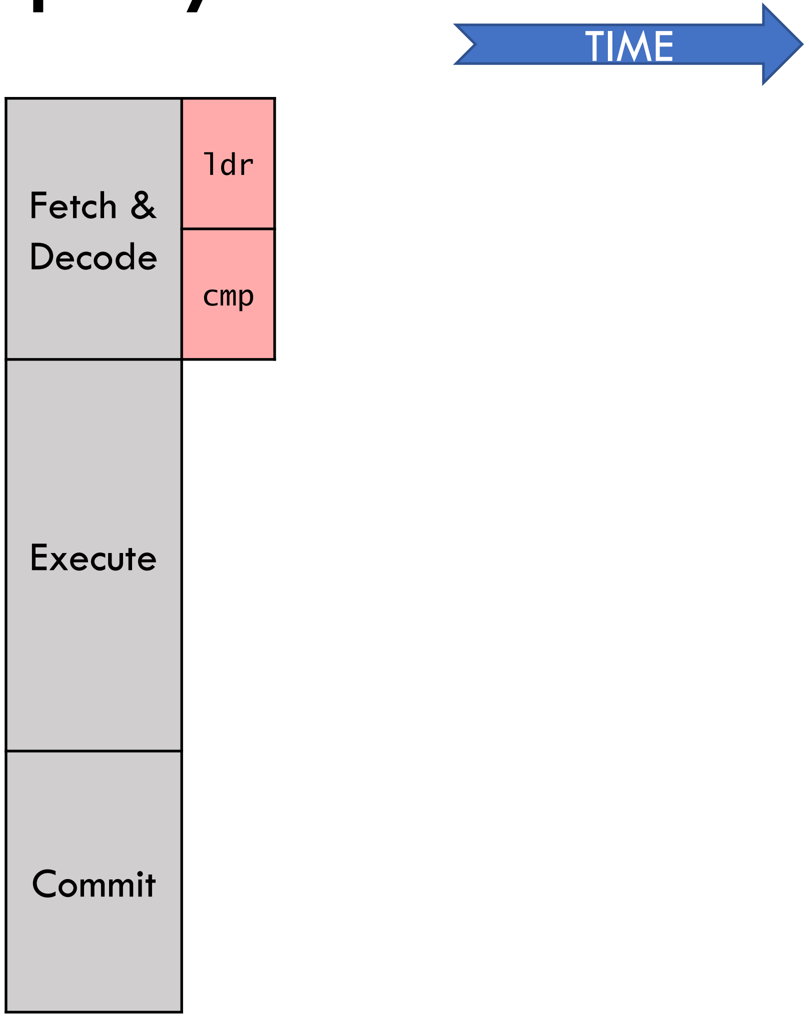


Focus on Execution, not Fetch & Commit

- Goal of OoO design is to only be limited by dataflow execution
- Fetch and commit are *over-provisioned* so that they (usually) do not limit performance
  ➔ Programmers can (usually) ignore fetch/commit

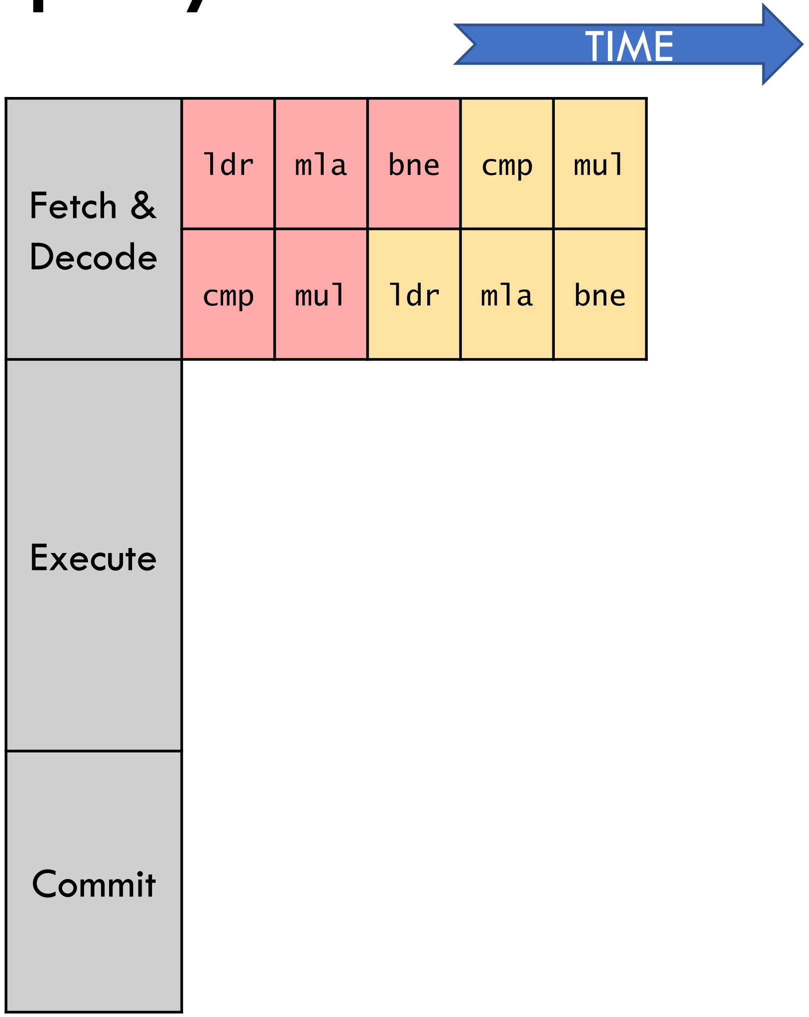
- **Big Caveat:** Programs with *inherently unpredictable* control flow will often be limited by fetch stalls (branch misprediction)
  - E.g., branching based on random data

Example: Superscalar OoO polynomial evaluation



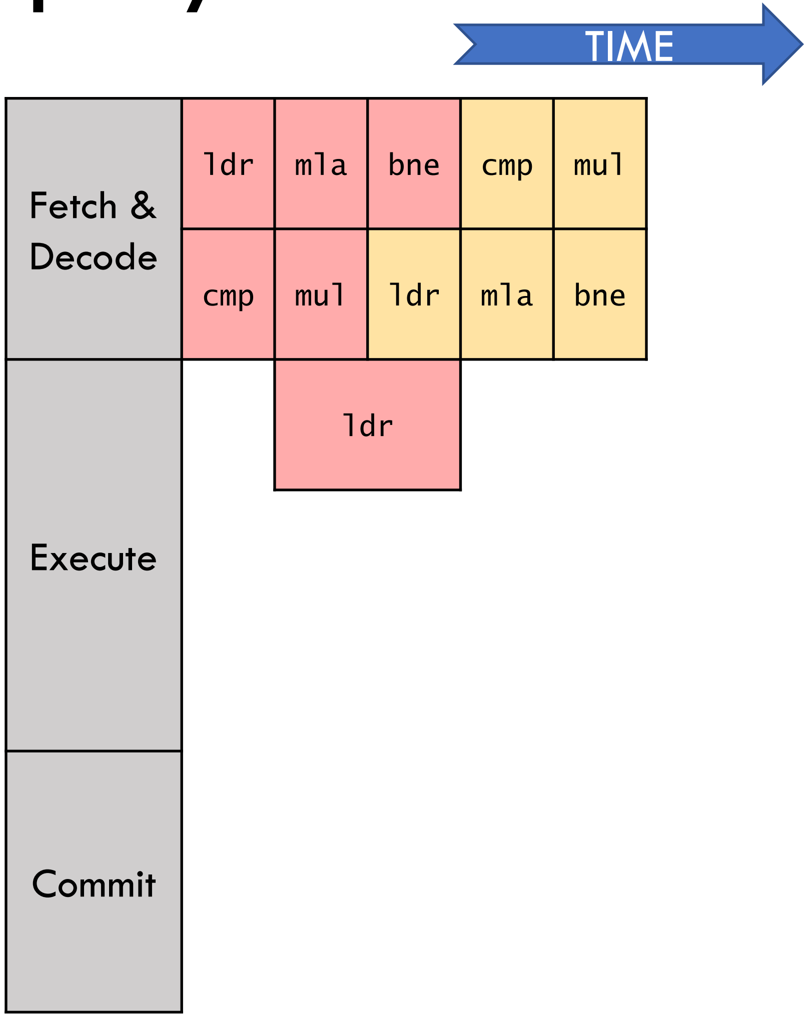
```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
```

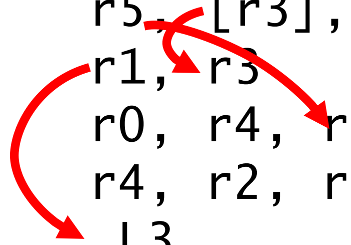
Example: Superscalar OoO polynomial evaluation



```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
```

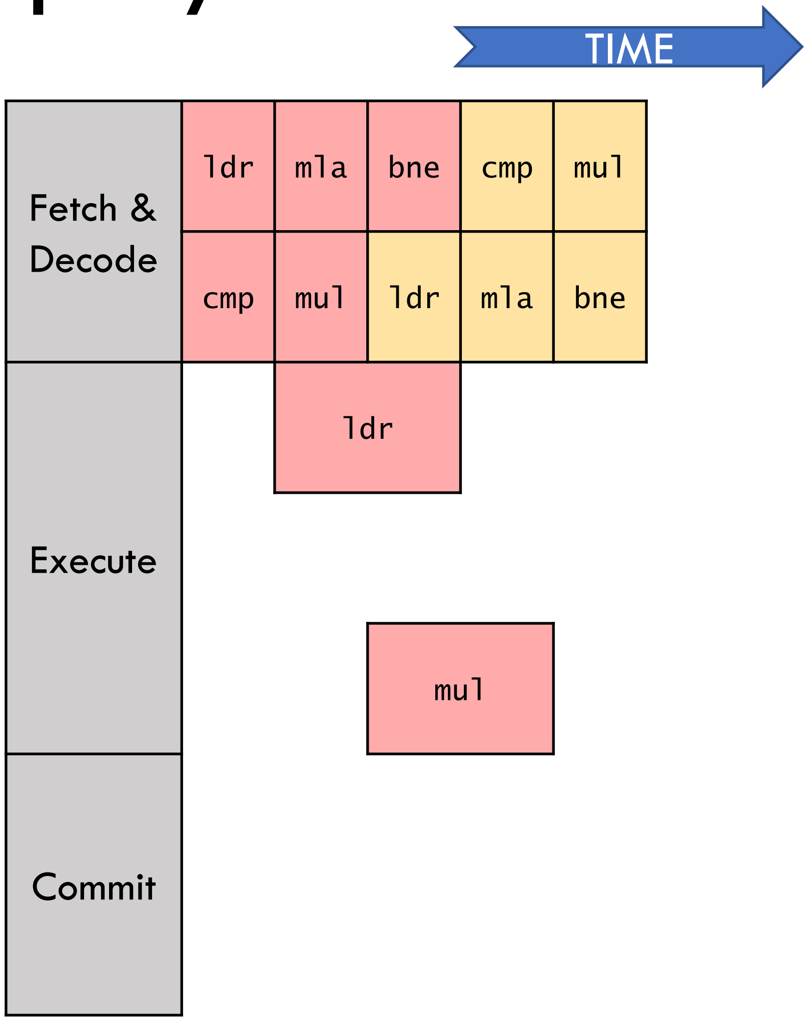
Example: Superscalar OoO polynomial evaluation

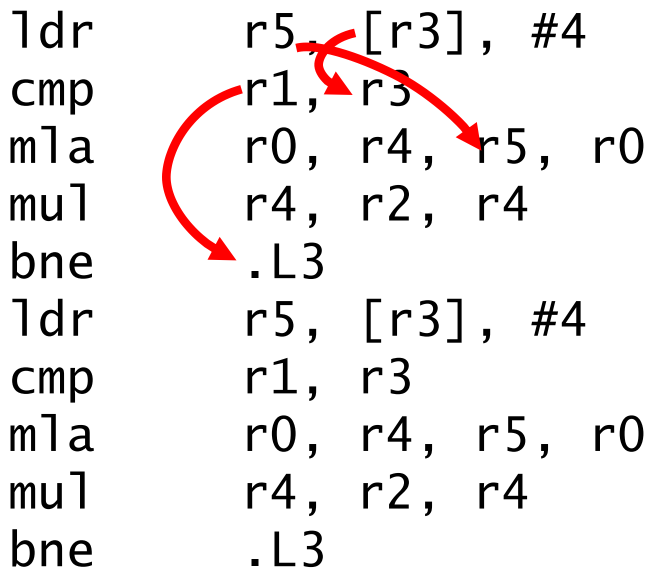




```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
```

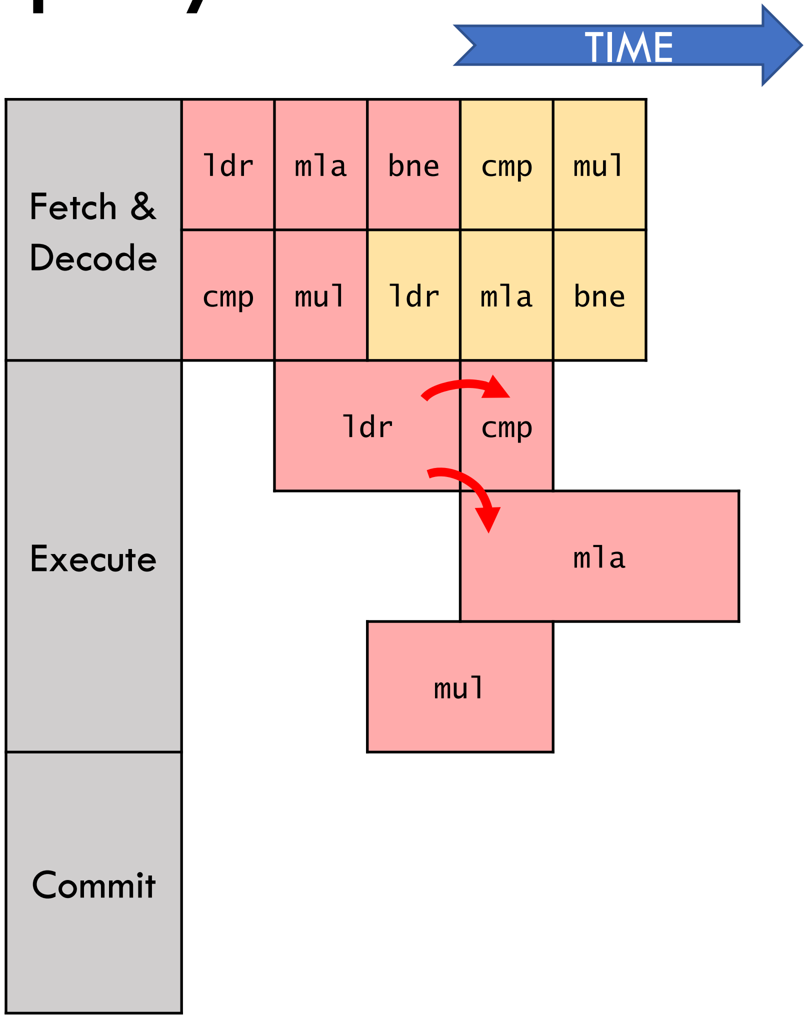
Example: Superscalar OoO polynomial evaluation





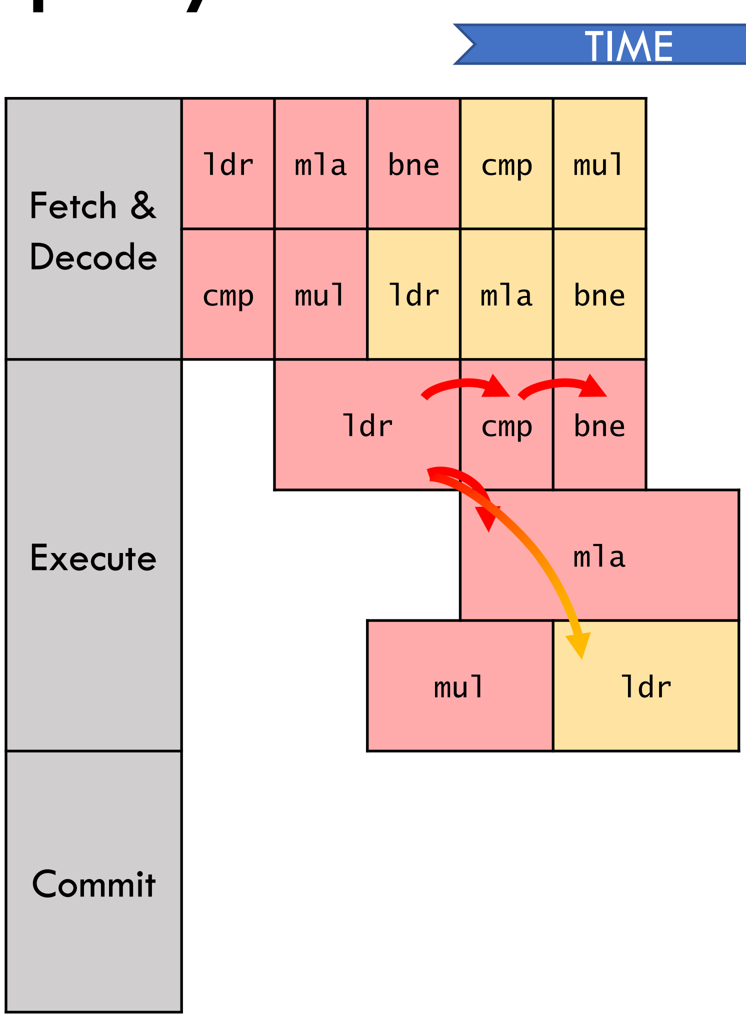
```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
```

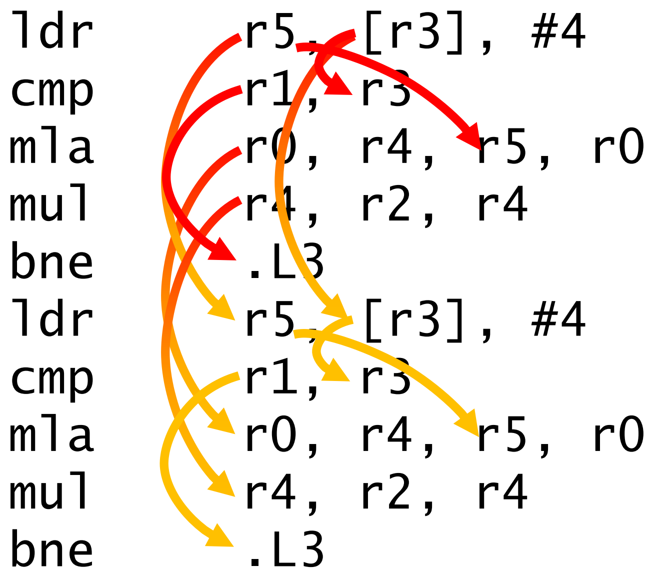
Example: Superscalar OoO polynomial evaluation



```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
```

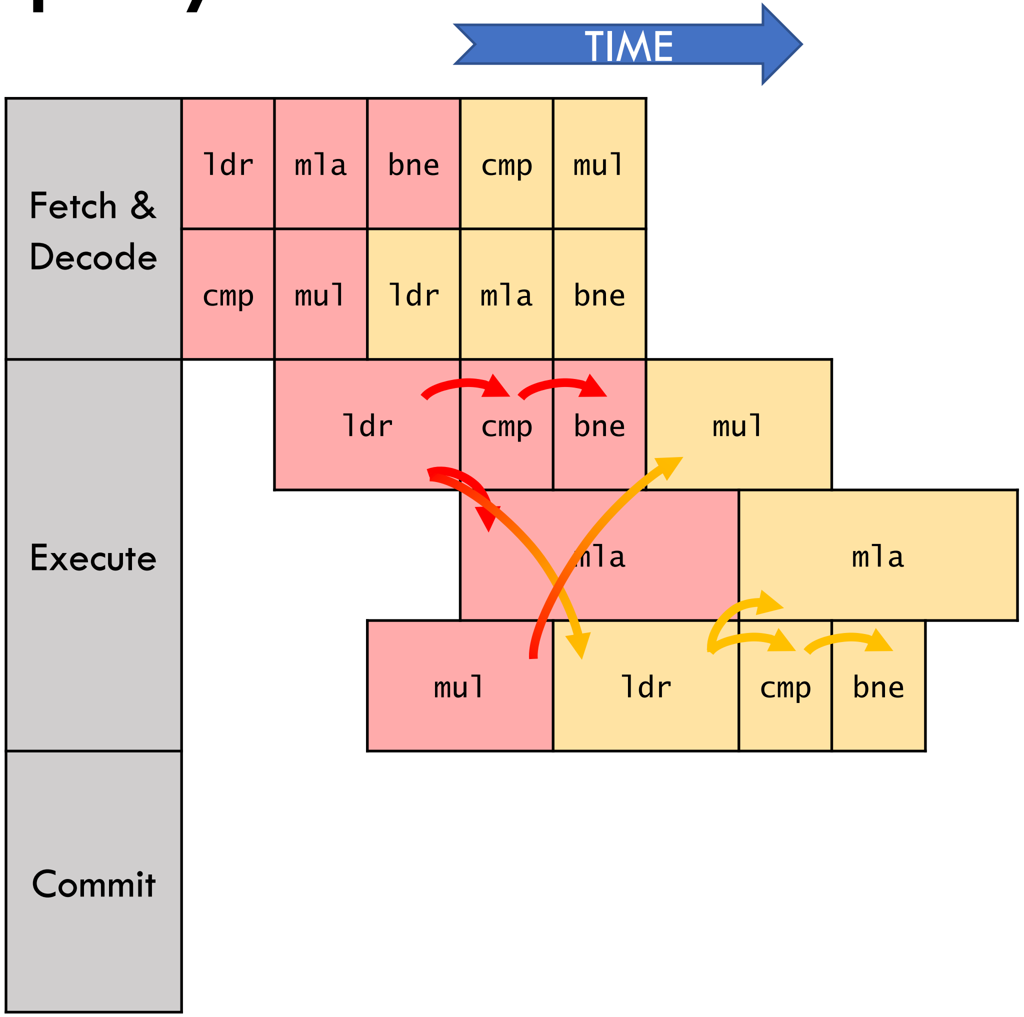
Example: Superscalar OoO polynomial evaluation

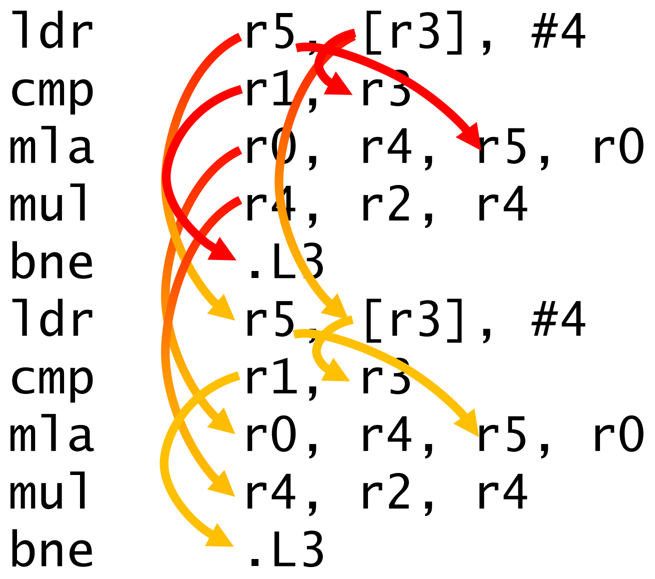




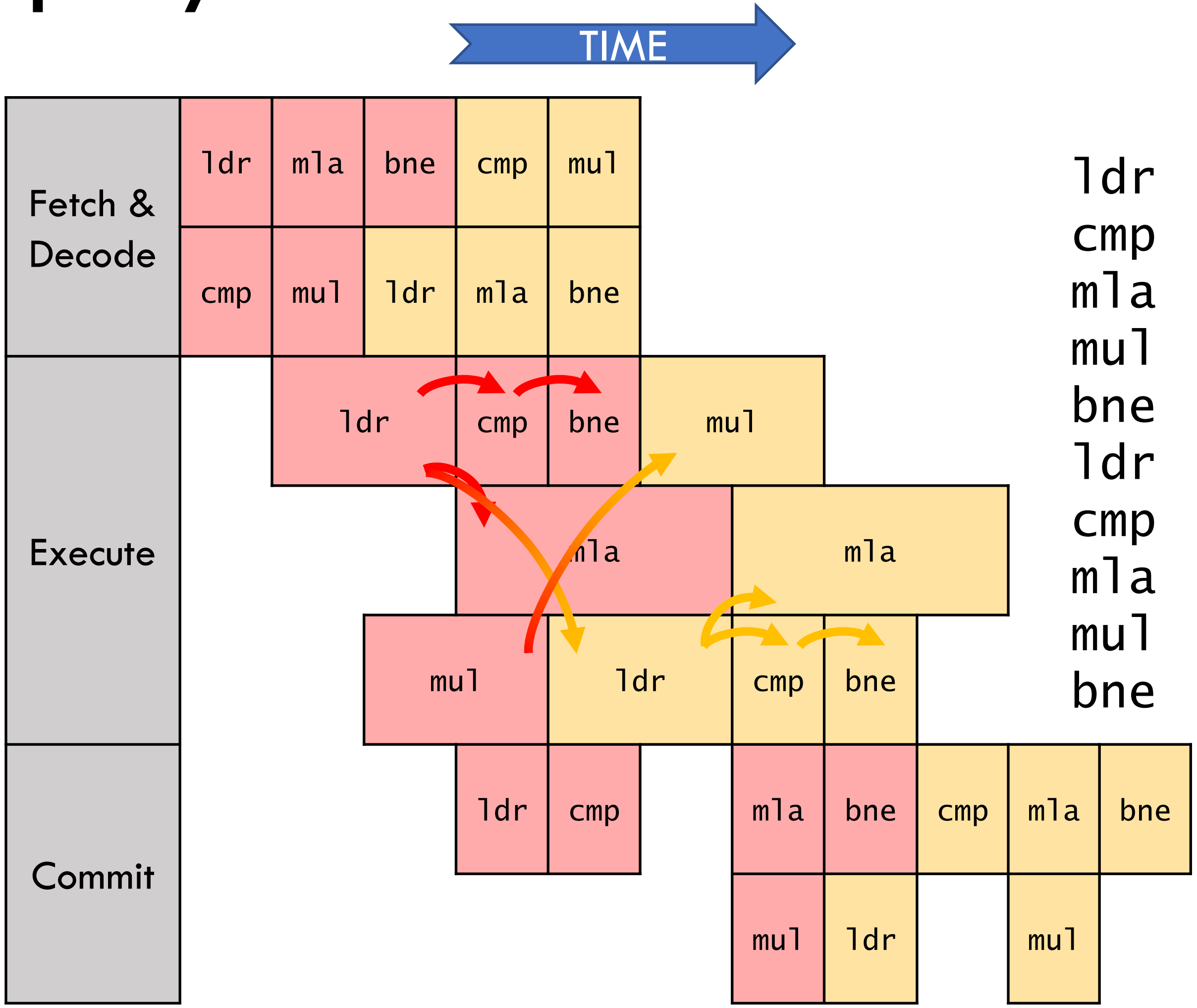
```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
```

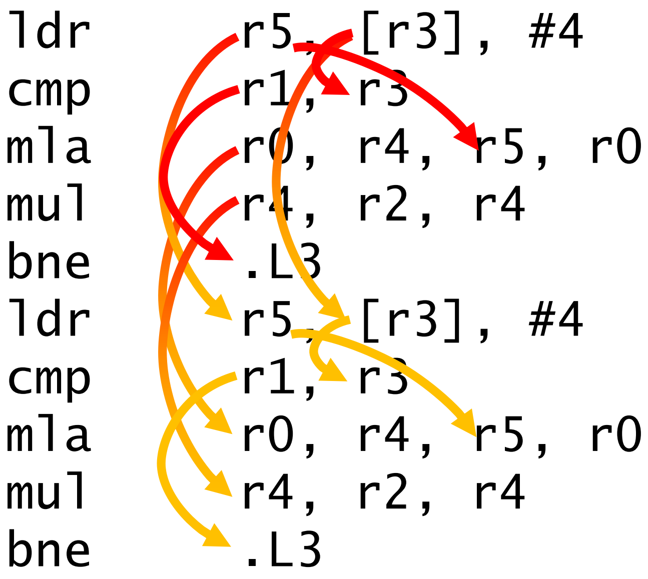
Example: Superscalar OoO polynomial evaluation




```
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
ldr     r5, [r3], #4
cmp     r1, r3
mla     r0, r4, r5, r0
mul     r4, r2, r4
bne     .L3
```

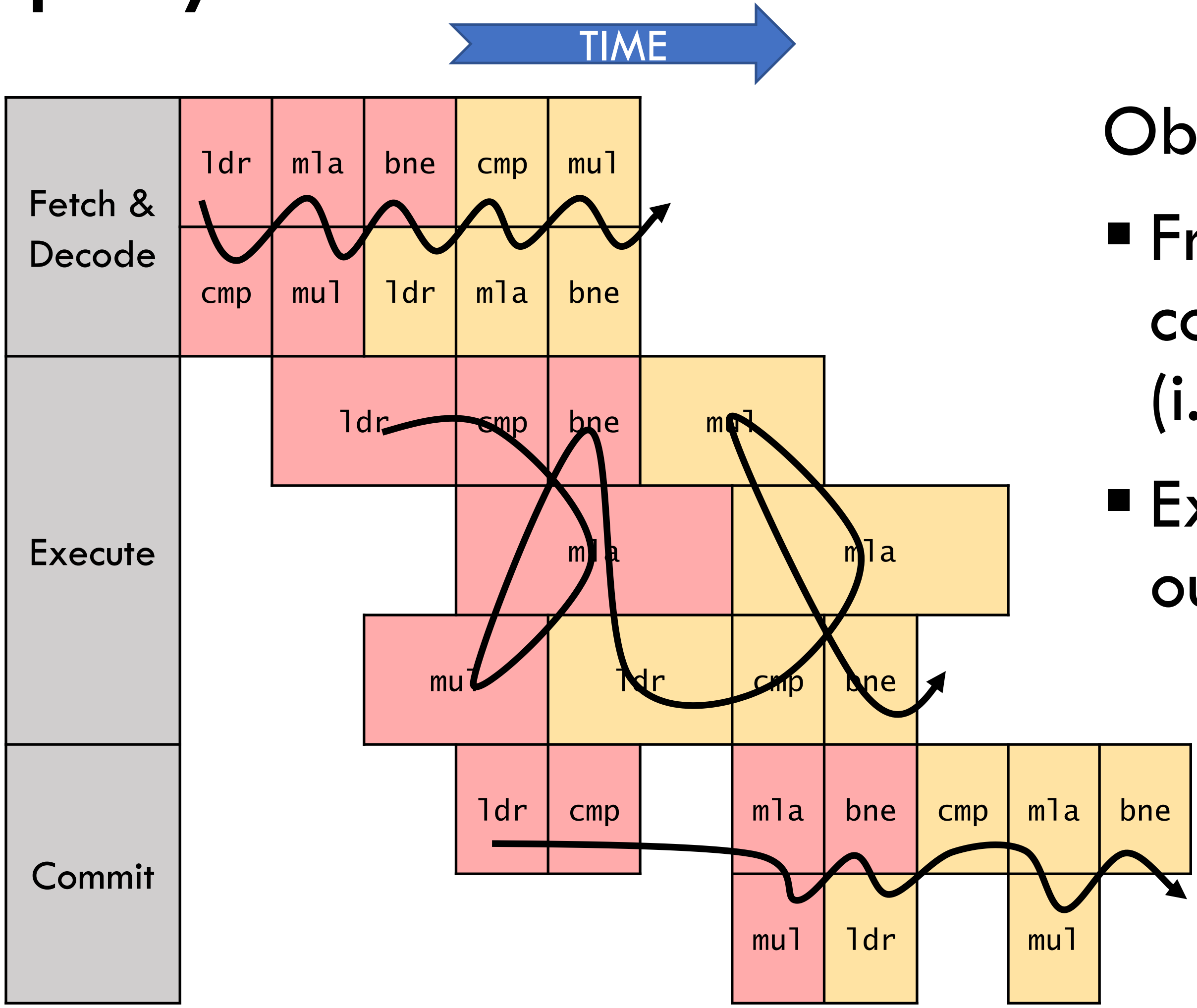
Example: Superscalar OoO polynomial evaluation





```
ldr   r5, [r3], #4
cmp   r1, r3
mla   r0, r4, r5, r0
mul   r4, r2, r4
bne   .L3
ldr   r5, [r3], #4
cmp   r1, r3
mla   r0, r4, r5, r0
mul   r4, r2, r4
bne   .L3
```

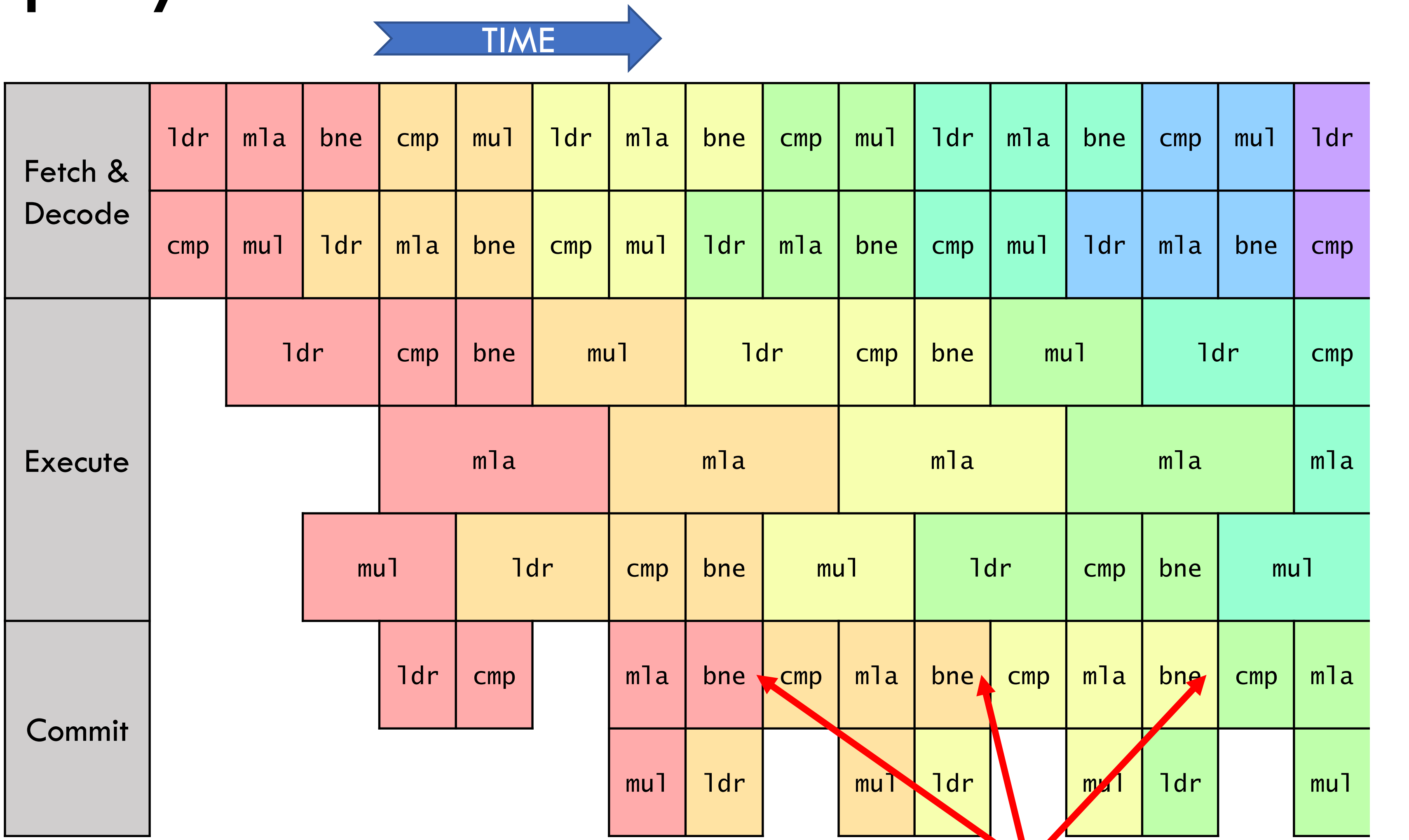
Example: Superscalar OoO polynomial evaluation



Observe:

- Front-end & commit in-order (i.e., left-to-right)
- Execute out-of-order

Example: Superscalar OoO polynomial evaluation


TIME
Fetch & Decode
Execute
Commit
ldr mla bne cmp mul ldr mla bne cmp mul ldr mla bne cmp mul ldr
cmp mul ldr mla bne cmp mul ldr mla bne cmp mul ldr mla bne cmp
ldr cmp bne mul ldr cmp bne mul ldr cmp
mla mla mla mla mla
mul ldr cmp bne mul ldr cmp bne mul
ldr cmp mla bne cmp mla bne cmp mla bne cmp mla
mul ldr mul ldr mul ldr mul

One loop iteration / 3 cycles!

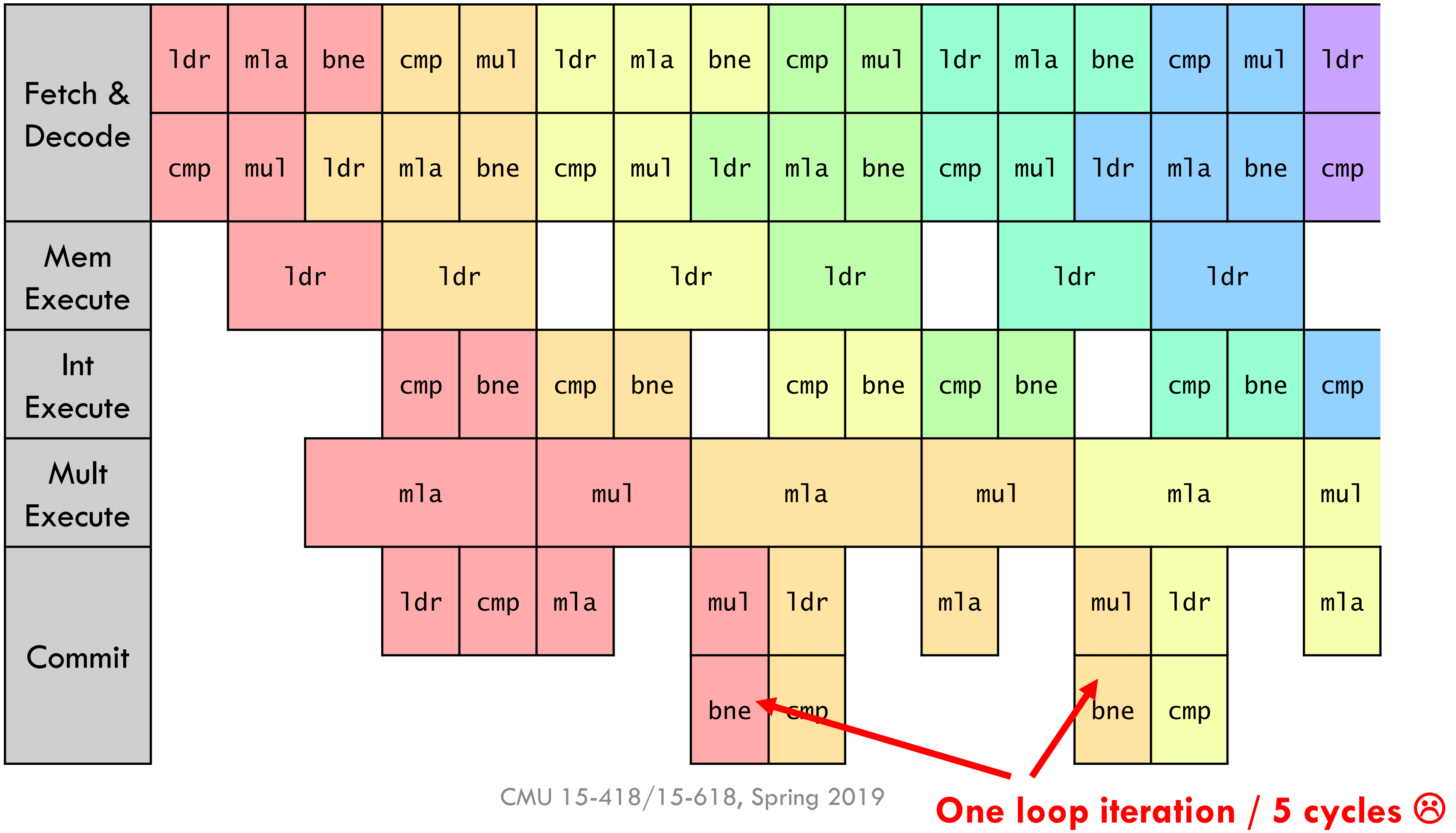
Structural hazards: Other throughput limitations

- Execution units are specialized
  - Floating-point (add/multiply)
  - Integer (add/multiply/compare)
  - Memory (load/store)

- Processor designers must choose <u>which</u> execution units to include and <u>how many</u>

- *Structural hazard:* Data is ready, but instr’n cannot issue because no hardware is available

Example: Structural hazards can severely limit performance



Fetch & Decode
ldr mla bne cmp mul ldr mla bne cmp mul ldr mla bne cmp mul ldr
cmp mul ldr mla bne cmp mul ldr mla bne cmp mul ldr mla bne cmp
Mem Execute
ldr ldr ldr ldr ldr ldr
Int Execute
cmp bne cmp bne cmp bne cmp bne cmp bne cmp
Mult Execute
mla mul mla mul mla mul
Commit
ldr cmp mla mul ldr mla mul ldr mla
bne cmp bne cmp
One loop iteration / 5 cycles ☹

Throughput Bound

- Ingredients:
  - Number of operations to perform (of each type)
  - Number & issue rate of “execution ports”/“functional units” (of each type)

- Throughput bound = ops / issue rate
  - E.g., (1 mla + 1 mul) / (2 + 3 cycles)

- Again, a real CPU might not exactly meet this bound

Software Takeaway

- OoO is much less sensitive to “good code”
  - Better performance portability
  - Of course, compiler still matters

- OoO makes performance analysis much simpler
  - **Throughput bound:** Availability of execution ports
  - **Latency bound:** “Critical path” latency
  - Slowest gives good approximation of program perf

Out-of-Order Execution: Under the Hood

OoO x86: Microcoding

- Each x86 instruction describes several operations
  - E.g., `add [esp+4], 5` means:
    1. Load `Mem[esp+4]`
    2. Add `5` to it
    3. Store result to `Mem[esp+4]`
- This is too much for (fast) hardware

- Instead, hardware decodes instr'ns into *micro-ops*
  ➔ Rest of pipeline uses micro-ops

Register Renaming

- "False dependences" can severely limit parallelism
  - Write-after-read
  - Write-after-write

- OoO processors eliminate false dependences by transparently *renaming registers*
  - CPU has many more "physical" than "architectural" registers
  - Each time register is written, it is allocated to a new physical register
  - Physical registers freed when instructions commit

Memory Disambiguation

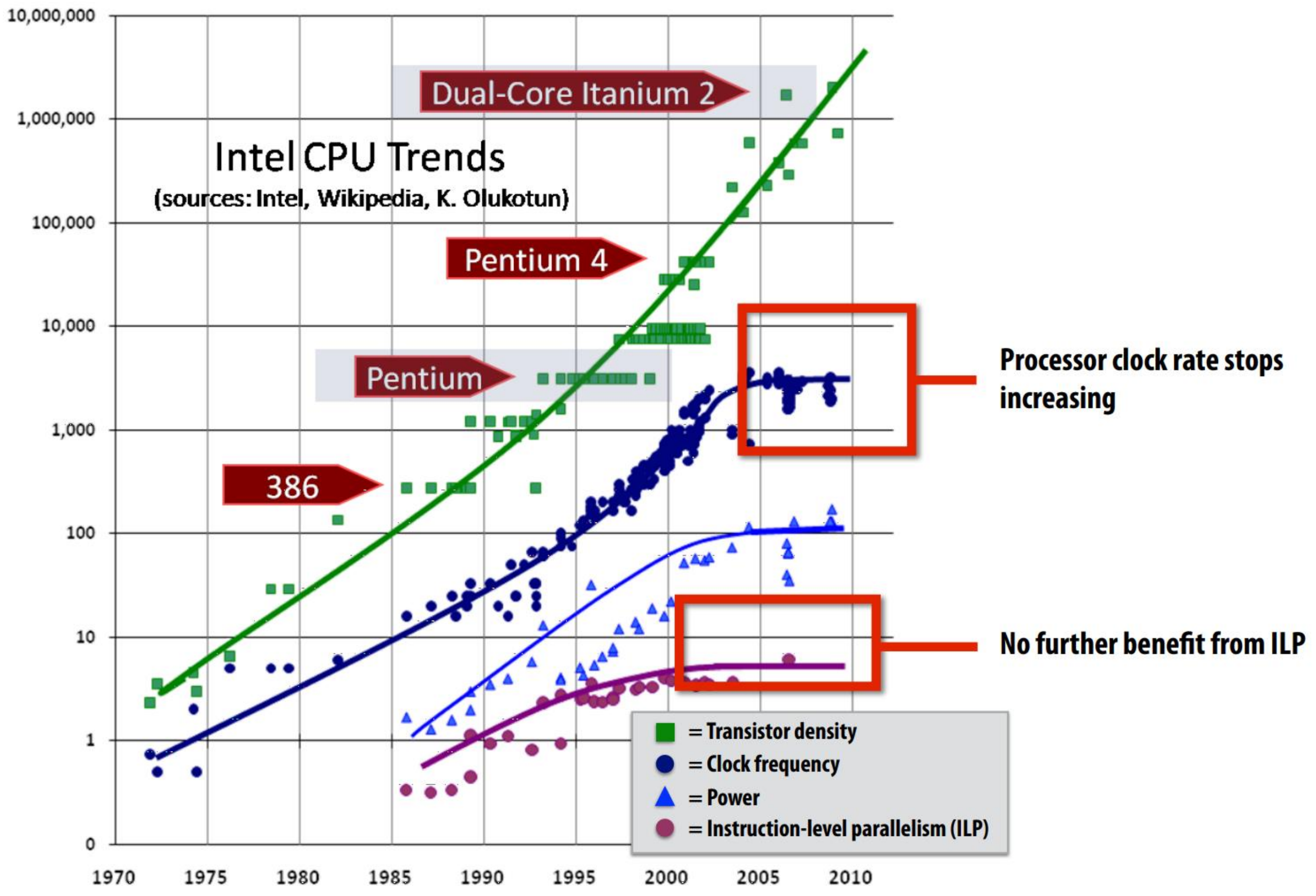
- CPU must respect store → load ordering
  - E.g., a later instruction reads a value from memory written by an earlier instruction
- But what if the OoO CPU executes the load first?
  - Must “rollback” + execute the load again (next slide)
- Corollary: OoO CPU must track the order of all loads & stores, and only write memory when a store commits

Rollback & Recovery

- OoO CPUs speculate constantly to improve performance
  - E.g., even guessing the results of a computation ("value prediction")
- Need mechanisms to "rollback" to an earlier point in execution when speculation goes awry
  - Complex: Need to recover old register names, flush pending memory operations, etc
- **Very** expensive: Up to hundreds of instrns of work lost!

Scaling Instruction-Level Parallelism

Recall from last time: ILP & pipelining tapped out... why?



Superscalar scheduling is complex & hard to scale

- Q: When is it safe to issue two instructions?
- A: When they are independent
  - Must compare all pairs of input and output registers

- Scalability: $O(W^2)$ comparisons where W is "issue width" of processor
  - Not great!

Limitations of ILP

- 4-wide superscalar × 20-stage pipeline = **80** instrns in flight
- High-performance OoO buffers *hundreds* of instructions

- **Programs have limited ILP**
  - Even with perfect scheduling, >8-wide issue doesn't help

- Pipelines can only go so deep
  - Branch misprediction penalty grows
  - Frequency (GHz) limited by power
- Dynamic scheduling overheads are significant
- Out-of-order scheduling is expensive

Limitations of ILP ➔ Multicore

- ILP works great! …But is complex + hard to scale
- From hardware perspective, multicore is much more efficient, but…
- **Parallel software is hard!**
  - Industry resisted multicore for as long as possible
  - When multicore finally happened, CPU μarch simplified ➔ more cores
  - Many program(mer)s still struggle to use multicore effectively