

CSE211: Compiler Design

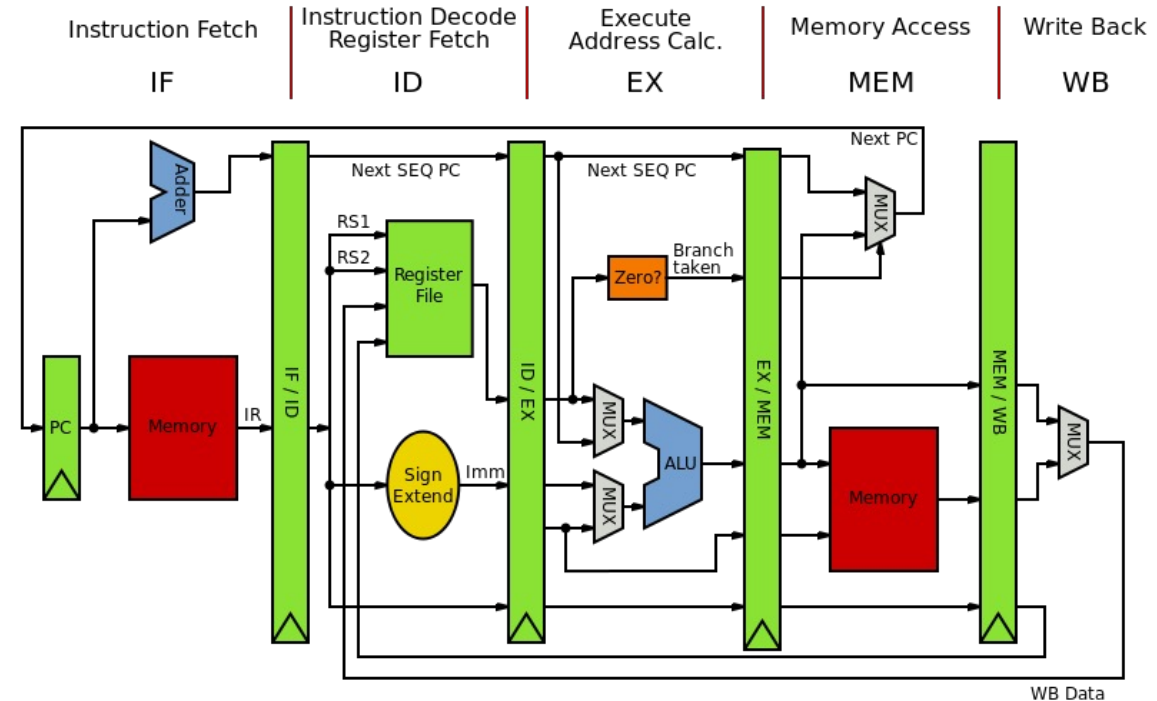
Nov. 13, 2023

- **Topic: More ILP!**

- loop unrolling
- reductions

- **Discussion questions (review):**

- What is instruction level parallelism?
- How can modern processors exploit ILP?



MIPS pipeline image from:

[https://commons.wikimedia.org/wiki/Pipeline_\(computer_hardware\)](https://commons.wikimedia.org/wiki/Pipeline_(computer_hardware))

Announcements

- Homework 2 is out
 - Due on *Wednesday*
 - I have office hours today
 - Rithik has office hours tomorrow
- Start thinking about 2nd paper
- Start thinking about final project
 - Deadline is to get final project APPROVED, not start brainstorming
- Homework 3 is assigned on Wednesday

Announcements

- Grading:
 - HW 1 grades and midterm are completely finished
 - Come see us during office hours if you have questions
 - Please make some time to see Rithik (for HW 1 or midterm) or me (midterm) if you lost any points that you don't think you should have.
 - We released the tests so you can see what you passed/failed
- Paper report *nearly* finished
- 2 week deadline to discuss grades

Review ILP

Instruction-level Parallelism (ILP)

- Parallelism from a single stream of instructions.
 - Output of program must match exactly a sequential execution!
- Widely applicable:
 - most mainstream programming languages are sequential
 - most deployed hardware has components to execute ILP
- Can benefit from a combination of hardware and software scheduling
- While it can be done by hand, it's less tedious to implement in a compiler

Finding dependencies in the compiler

- What type of instructions can be done in parallel?

two instructions can be executed in parallel if they are independent

```
x = z + w;  
a = b + c;
```

Two instructions are independent if the operand registers are disjoint from the result registers

instructions that are not independent cannot be executed in parallel

```
x = z + w;  
a = b + x;
```

Finding dependencies in the compiler

- What type of instructions can be done in parallel?

two instructions can be executed in parallel if they are independent

```
x = z + w;  
a = b + c;
```

Two instructions are independent if the operand registers are disjoint from the result registers

instructions that are not independent cannot be executed in parallel

```
x = z + w;  
a = b + x;
```

Many times, dependencies can be easily tracked in the compiler:

Easier with:
+ within a basic block
+ using SSA form

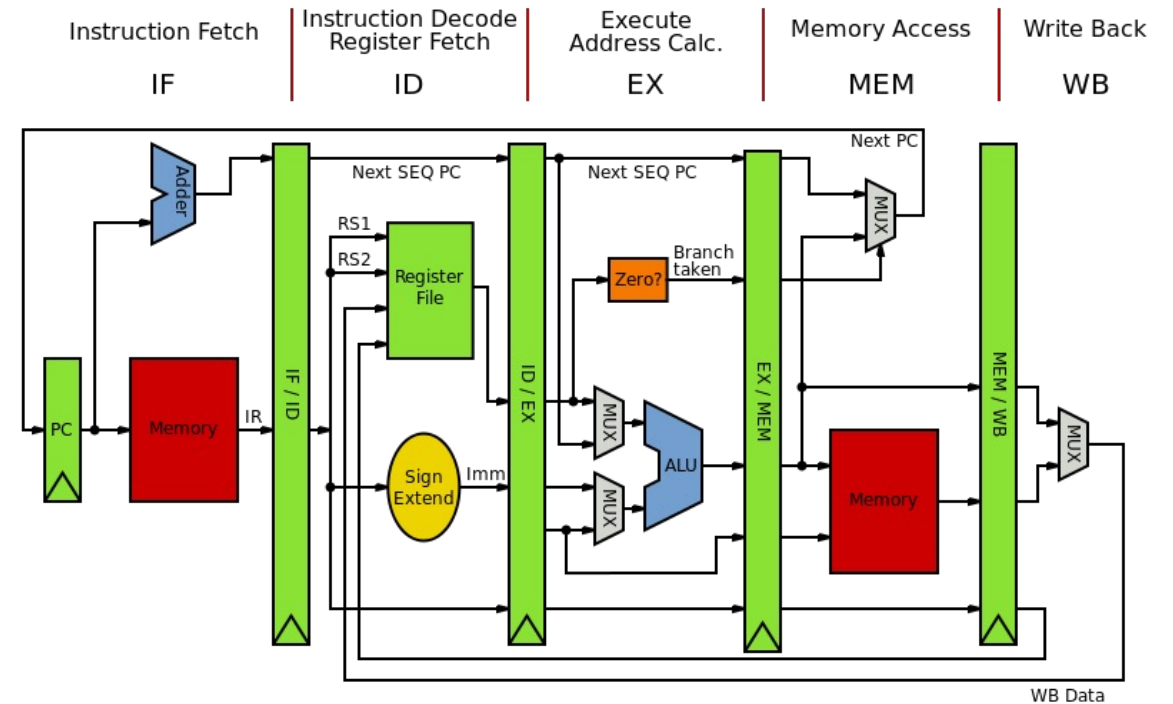
Harder with:
- memory locations

Different types of dependencies

- Data Dependence
- Control Dependence
- Memory Dependence

How can hardware execute ILP?

- Pipeline parallelism
- Abstract mental model for compiler:
 - N-stage pipeline
 - N instructions can be in-flight
 - Dependencies stall pipeline

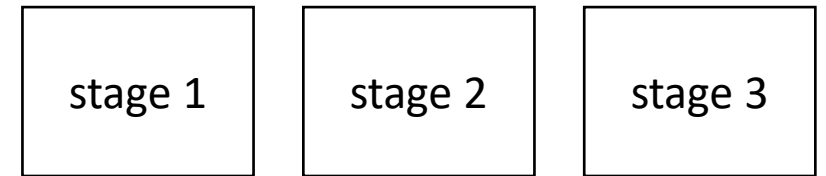


MIPS pipeline image from:
[https://commons.wikimedia.org/wiki/Pipeline_\(computer_hardware\)](https://commons.wikimedia.org/wiki/Pipeline_(computer_hardware))

How can hardware execute ILP?

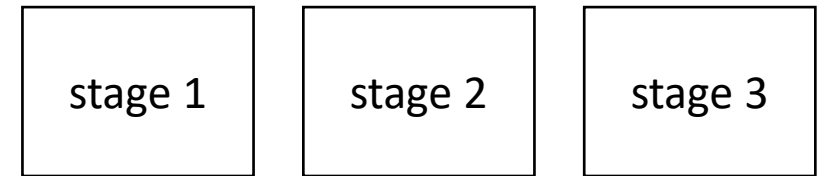
- Pipeline parallelism
- Abstract mental model for compiler:
 - N-stage pipeline
 - N instructions can be in-flight
 - Dependencies stall pipeline

```
instr1;  
instr2;  
instr3;
```



How can hardware execute ILP?

- Pipeline parallelism
- Abstract mental model for compiler:
 - N-stage pipeline
 - N instructions can be in-flight
 - Dependencies stall pipeline



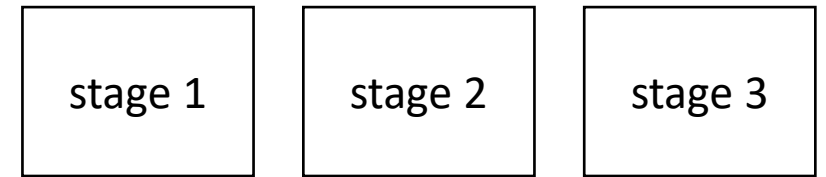
`instr1;`

`instr2;`

`instr3;`

How can hardware execute ILP?

- Pipeline parallelism
- Abstract mental model for compiler:
 - N-stage pipeline
 - N instructions can be in-flight
 - Dependencies stall pipeline



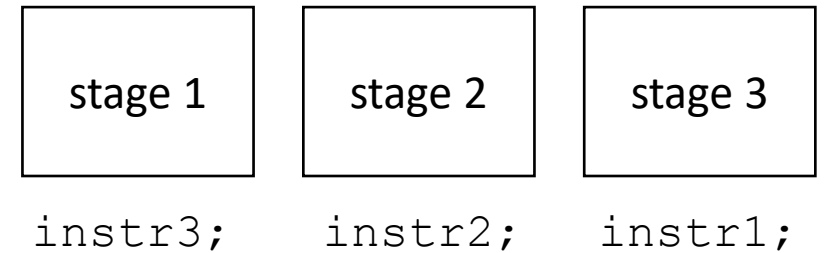
`instr2;`

`instr1;`

`instr3;`

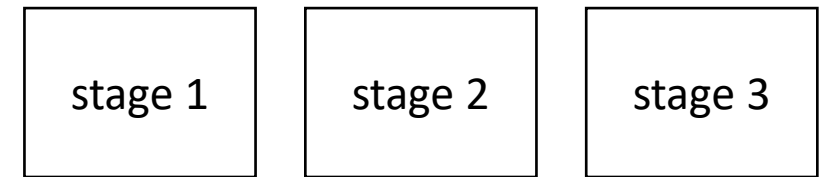
How can hardware execute ILP?

- Pipeline parallelism
- Abstract mental model for compiler:
 - N-stage pipeline
 - N instructions can be in-flight
 - Dependencies stall pipeline



How can hardware execute ILP?

- Pipeline parallelism
- Abstract mental model for compiler:
 - N-stage pipeline
 - N instructions can be in-flight
 - Dependencies stall pipeline



6 cycles for 3 independent instructions

Converges to 1 instruction per cycle

How can hardware execute ILP?

- Executing multiple instructions at once:
- Superscalar architecture:
 - Several sequential operations are issued in parallel
 - hardware detects dependencies

```
instr0;  
instr1;  
instr2;
```

issue-width is maximum number of instructions that can be issued in parallel

if instr0 and instr1 are independent, they will be issued in parallel

What does this look like in the real world?

- Intel Haswell (2013):
 - Issue width of 4
 - 14-19 stage pipeline
 - OoO execution
- Intel Nehalem (2008)
 - 20-24 stage pipeline
 - Issue width of 2-4
 - OoO execution
- ARM
 - V7 has 3 stage pipeline; Cortex V8 has 13
 - Cortex V8 has issue width of 2
 - OoO execution
- RISC-V
 - Ariane and Rocket are In-Order
 - 3-6 stage pipelines
 - some super scaler implementations (BOOM)

Other examples?

What does this mean for compiler writers?

- We should have an abstract and parametrized performance model for instruction scheduling (the order of instructions)
- Try not to place dependent instructions in sequence
- *Above all, instructions must respect sequential semantics!*

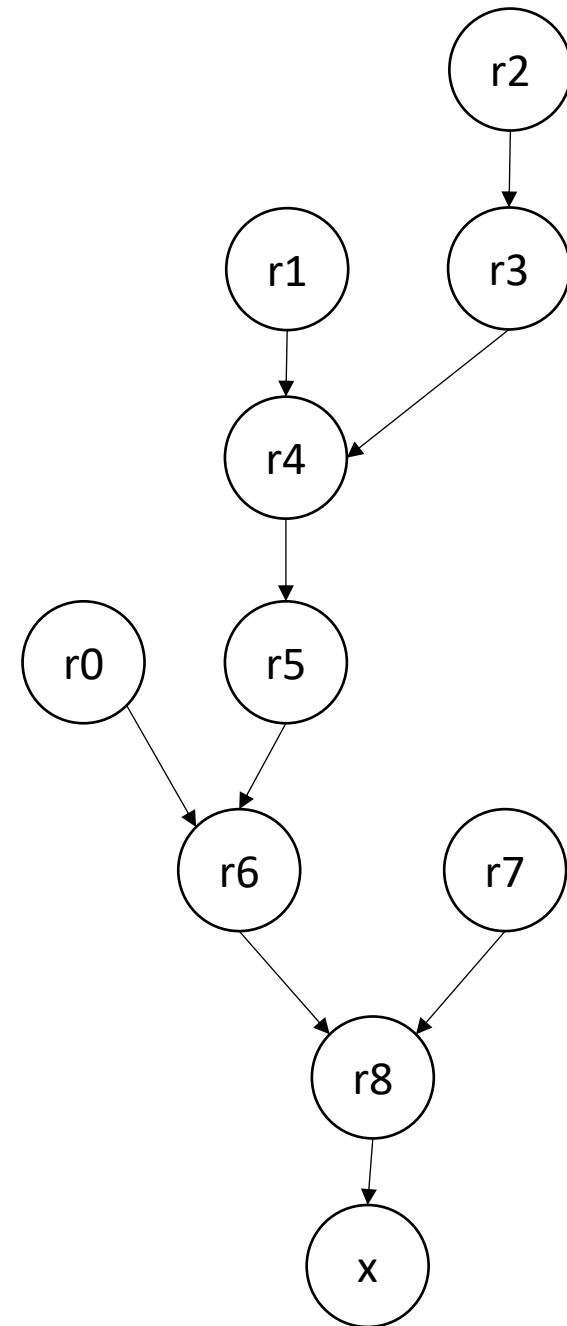
Four compiler techniques for better ILP

- Priority topological ordering
- Anticipatable expressions
- Independent for loops
- Reduction for loops

Priority Topological Ordering of DDGs for Superscalar

```
r0 = neg (b) ;  
r1 = b * b ;  
r2 = 4 * a ;  
r3 = r2 * c ;  
r4 = r1 - r3 ;  
r5 = sqrt (r4) ;  
r6 = r0 - r5 ;  
r7 = 2 * a ;  
r8 = r6 / r7 ;  
x = r8 ;
```

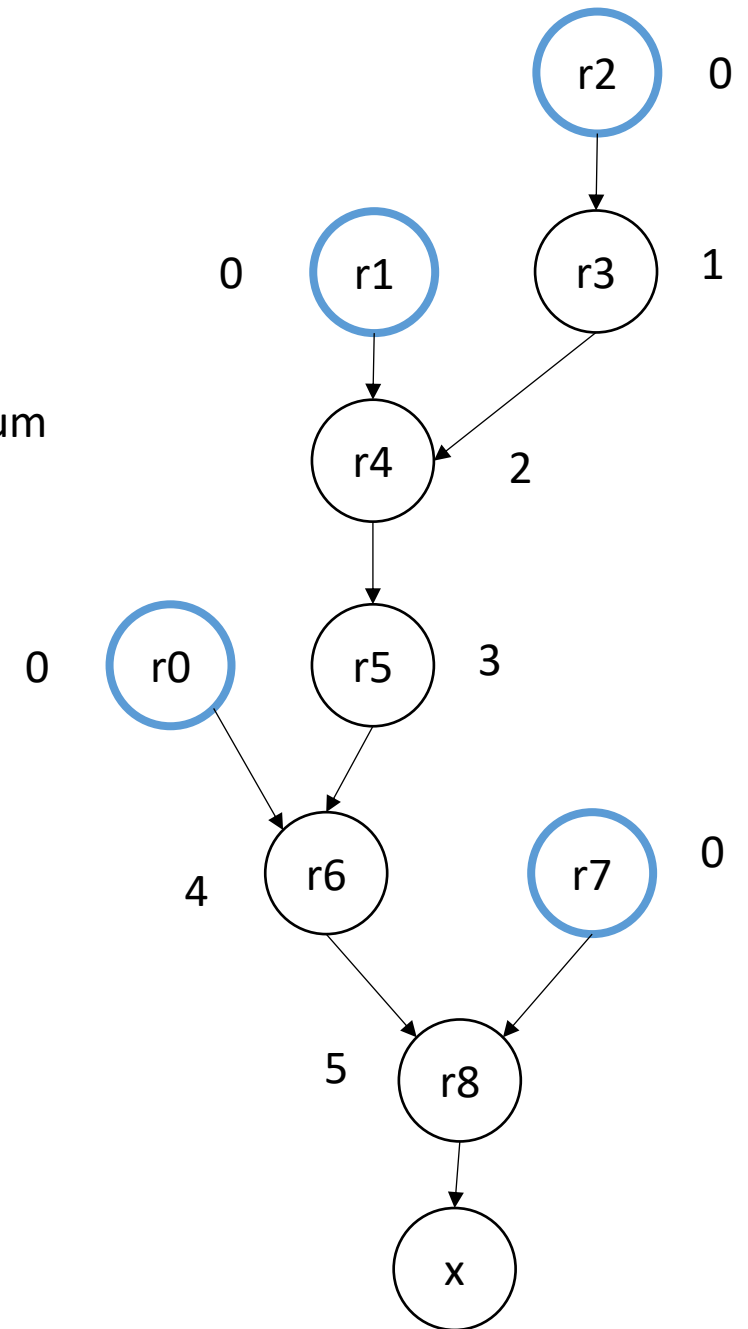
First, consider optimizing for superscalar



Priority Topological Ordering of DDGs for Superscalar

```
r0 = neg (b) ;  
r1 = b * b ;  
r2 = 4 * a ;  
r3 = r2 * c ;  
r4 = r1 - r3 ;  
r5 = sqrt (r4) ;  
r6 = r0 - r5 ;  
r7 = 2 * a ;  
r8 = r6 / r7 ;  
x = r8 ;
```

Label nodes with the maximum distance to a source

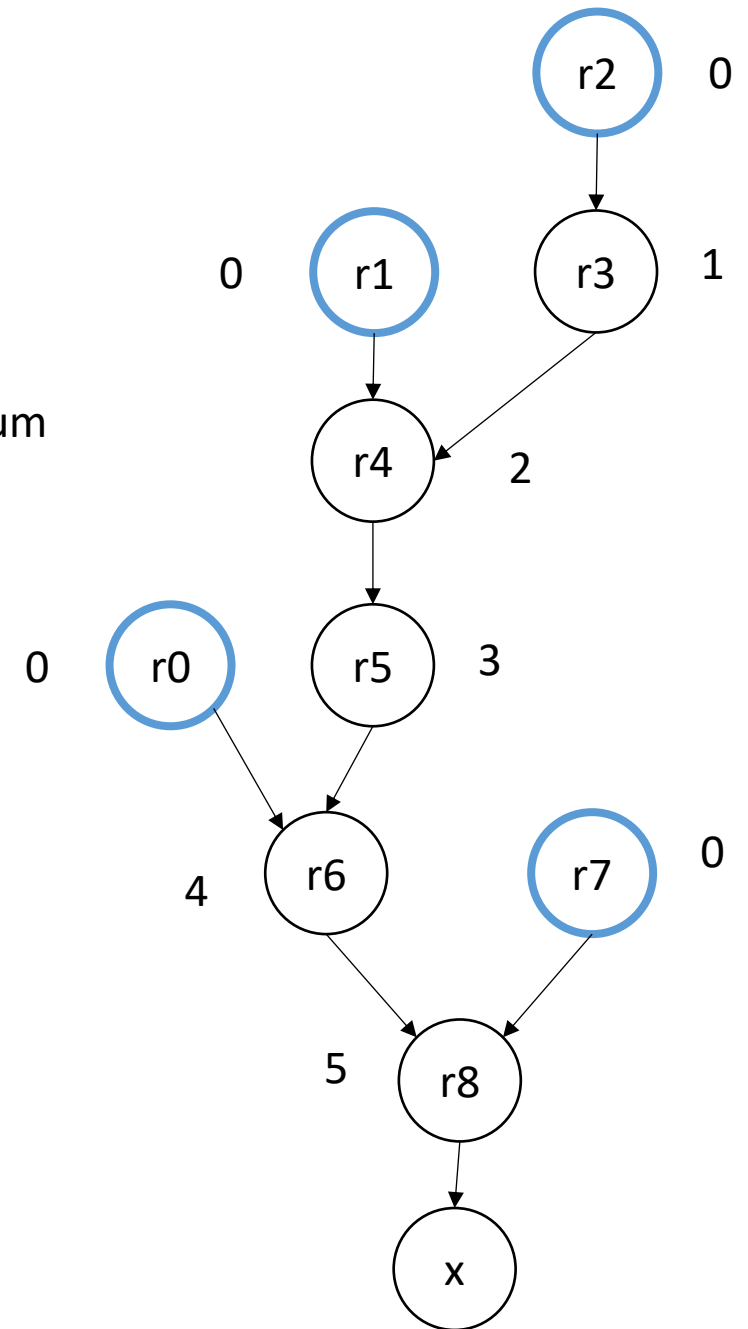


Priority Topological Ordering of DDGs for Superscalar

```
r0 = neg (b) ;  
r1 = b * b ;  
r2 = 4 * a ;  
r3 = r2 * c ;  
r4 = r1 - r3 ;  
r5 = sqrt (r4) ;  
r6 = r0 - r5 ;  
r7 = 2 * a ;  
r8 = r6 / r7 ;  
x = r8 ;
```

Label nodes with the maximum distance to a source

Break ties in topological order using this number

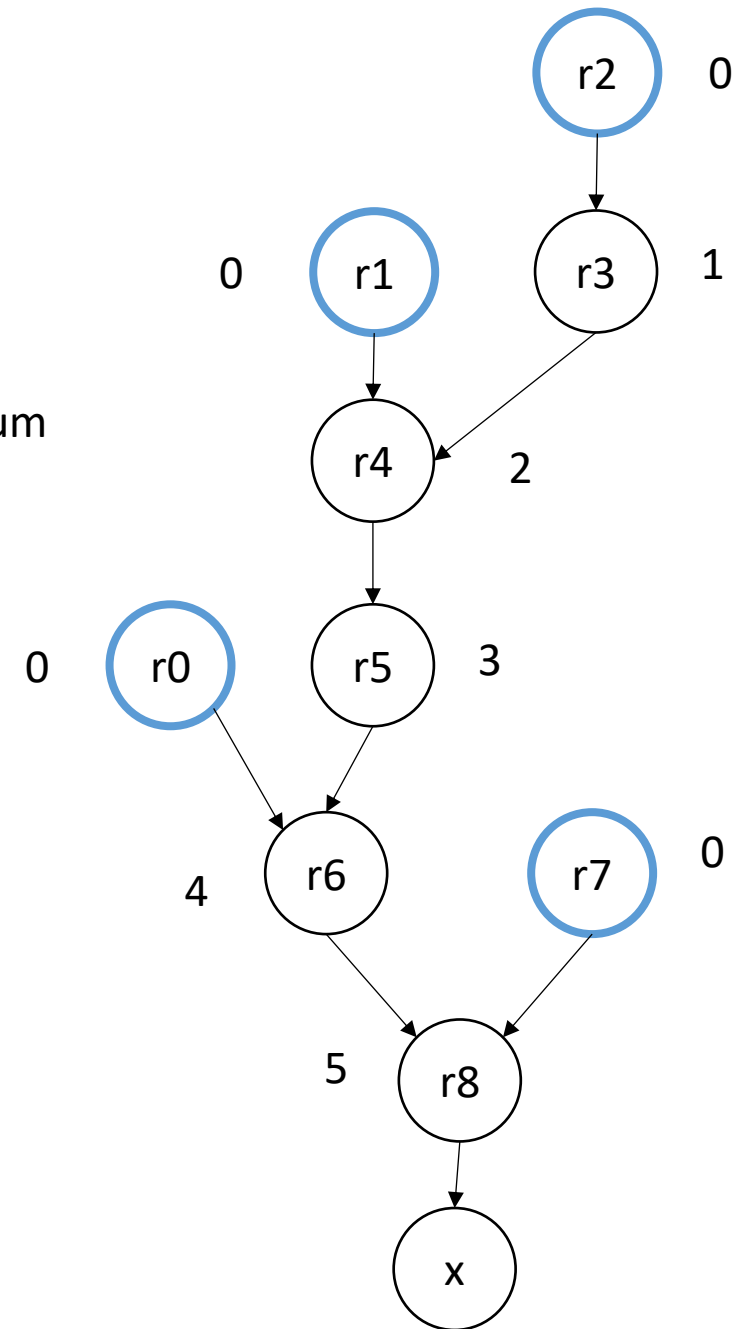


Priority Topological Ordering of DDGs for Superscalar

```
r0 = neg (b) ;  
r1 = b * b ;  
r2 = 4 * a ;  
r3 = r2 * c ;  
r4 = r1 - r3 ;  
r5 = sqrt (r4) ;  
r6 = r0 - r5 ;  
r7 = 2 * a ;  
r8 = r6 / r7 ;  
x = r8 ;
```

Label nodes with the maximum distance to a source

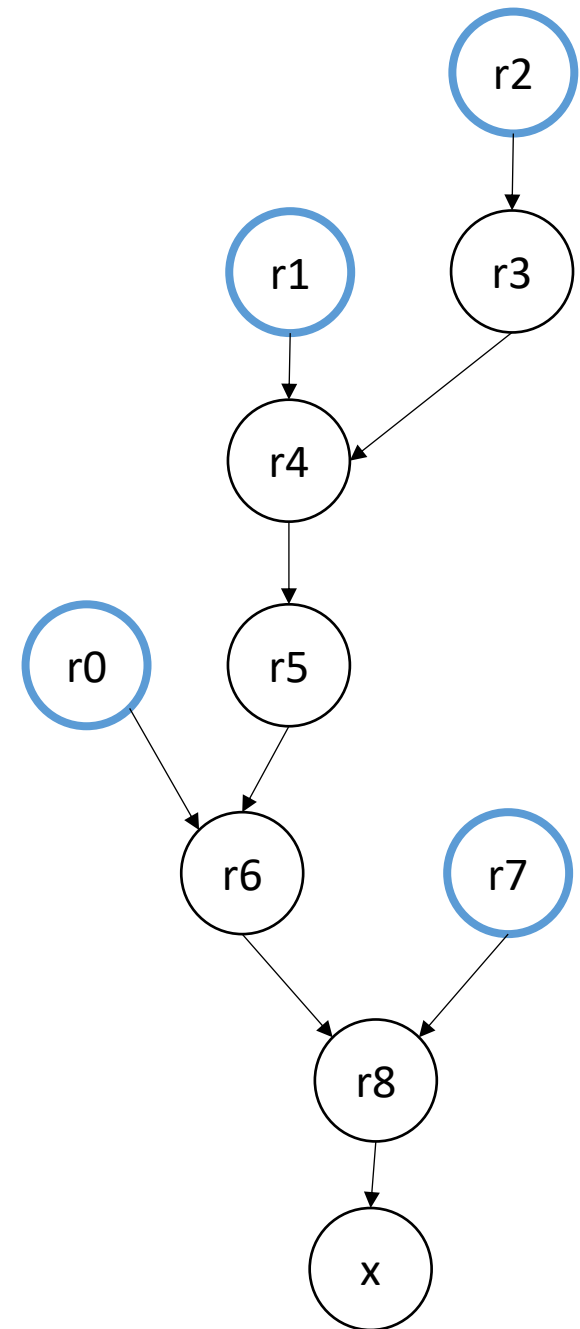
Break ties in topological order using this number



Priority Topological Ordering of DDGs for Pipelining

```
r0 = neg (b) ;  
r1 = b * b ;  
r2 = 4 * a ;  
r7 = 2 * a ;  
r3 = r2 * c ;  
r4 = r1 - r3 ;  
r5 = sqrt (r4) ;  
r6 = r0 - r5 ;  
r8 = r6 / r7 ;  
x = r8 ;
```

superscalar should move independent instructions as high as possible. What about pipelining?

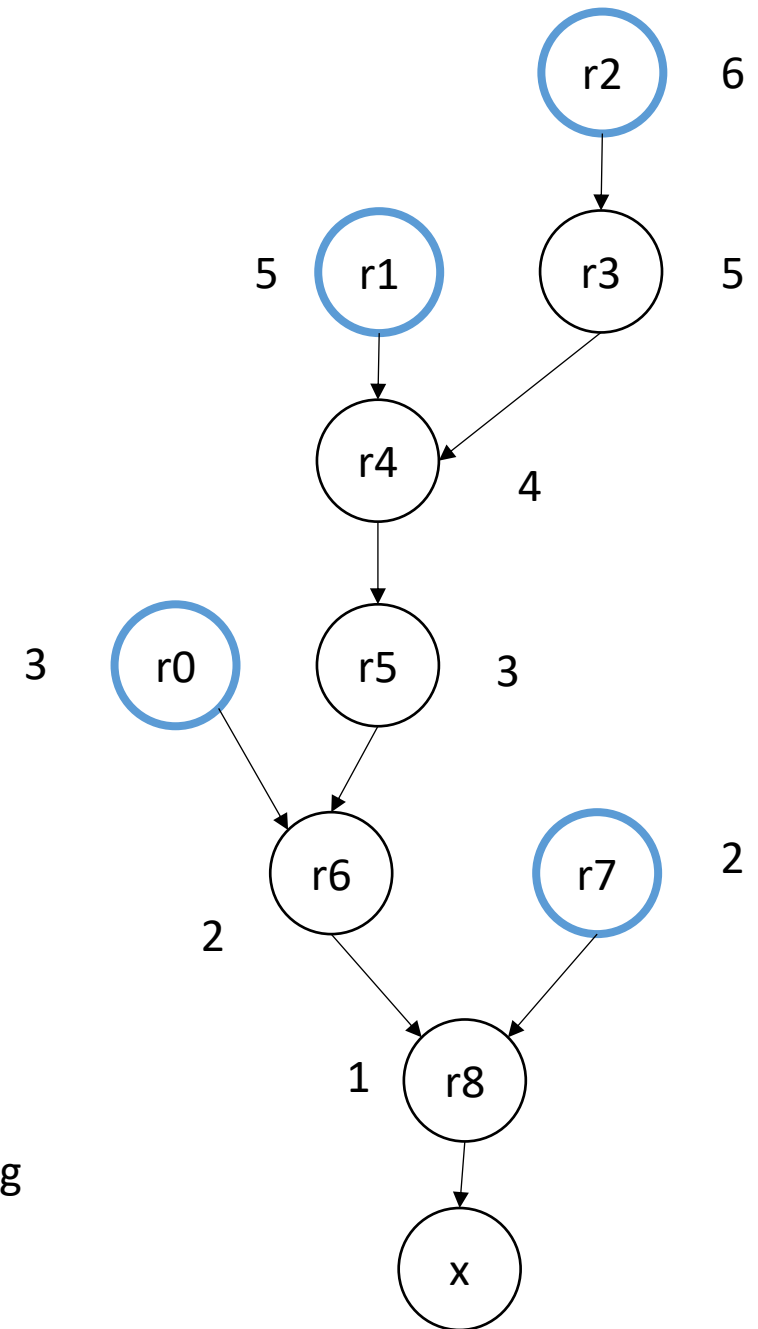


Priority Topological Ordering of DDGs for Pipelining

```
r0 = neg (b) ;  
r1 = b * b ;  
r2 = 4 * a ;  
r3 = r2 * c ;  
r4 = r1 - r3 ;  
r5 = sqrt (r4) ;  
r6 = r0 - r5 ;  
r7 = 2 * a ;  
r8 = r6 / r7 ;  
x = r8 ;
```

superscalar should move independent instructions as high as possible. What about pipelining?

label each node with a distance from the root. Schedule each node according to the level

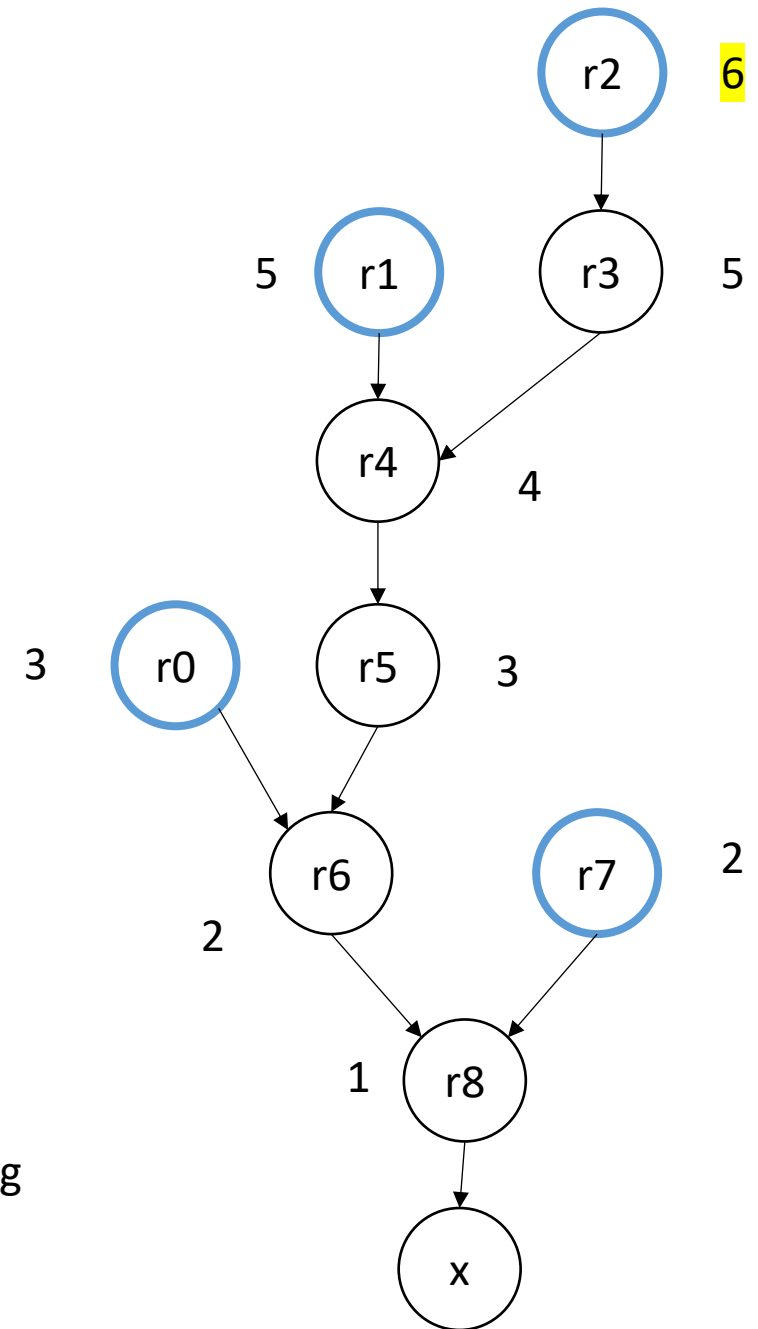


Priority Topological Ordering of DDGs for Pipelining

```
r0 = neg (b) ;  
r1 = b * b ;  
r2 = 4 * a ;  
r3 = r2 * c ;  
r4 = r1 - r3 ;  
r5 = sqrt (r4) ;  
r6 = r0 - r5 ;  
r7 = 2 * a ;  
r8 = r6 / r7 ;  
x = r8 ;
```

superscalar should move independent instructions as high as possible. What about pipelining?

label each node with a distance from the root. Schedule each node according to the level



Priority Topological Ordering of DDGs for Pipelining

```
r2 = 4 * a;
```

```
r0 = neg(b);
```

```
r1 = b * b;
```

```
r3 = r2 * c;
```

```
r4 = r1 - r3;
```

```
r5 = sqrt(r4);
```

```
r6 = r0 - r5;
```

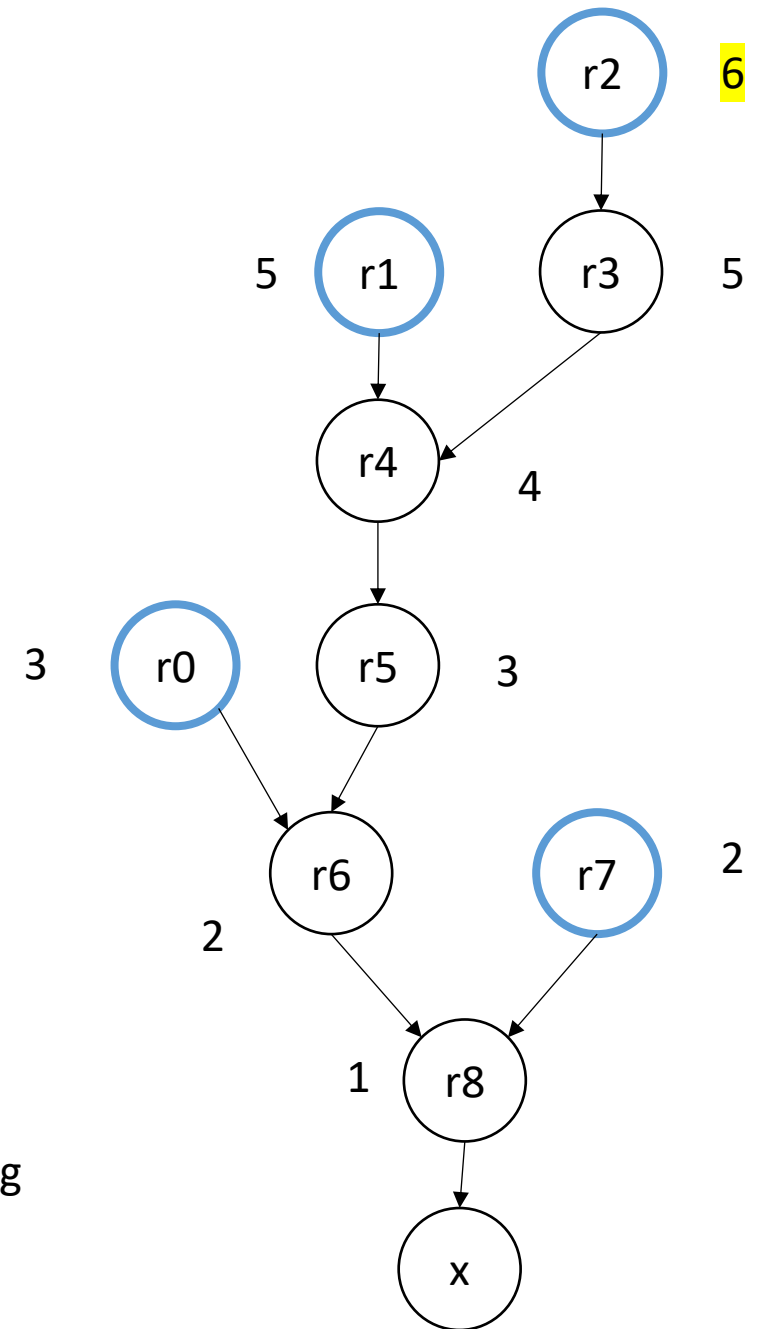
```
r7 = 2 * a;
```

```
r8 = r6 / r7;
```

```
x = r8;
```

superscalar should move independent instructions as high as possible. What about pipelining?

label each node with a distance from the root. Schedule each node according to the level

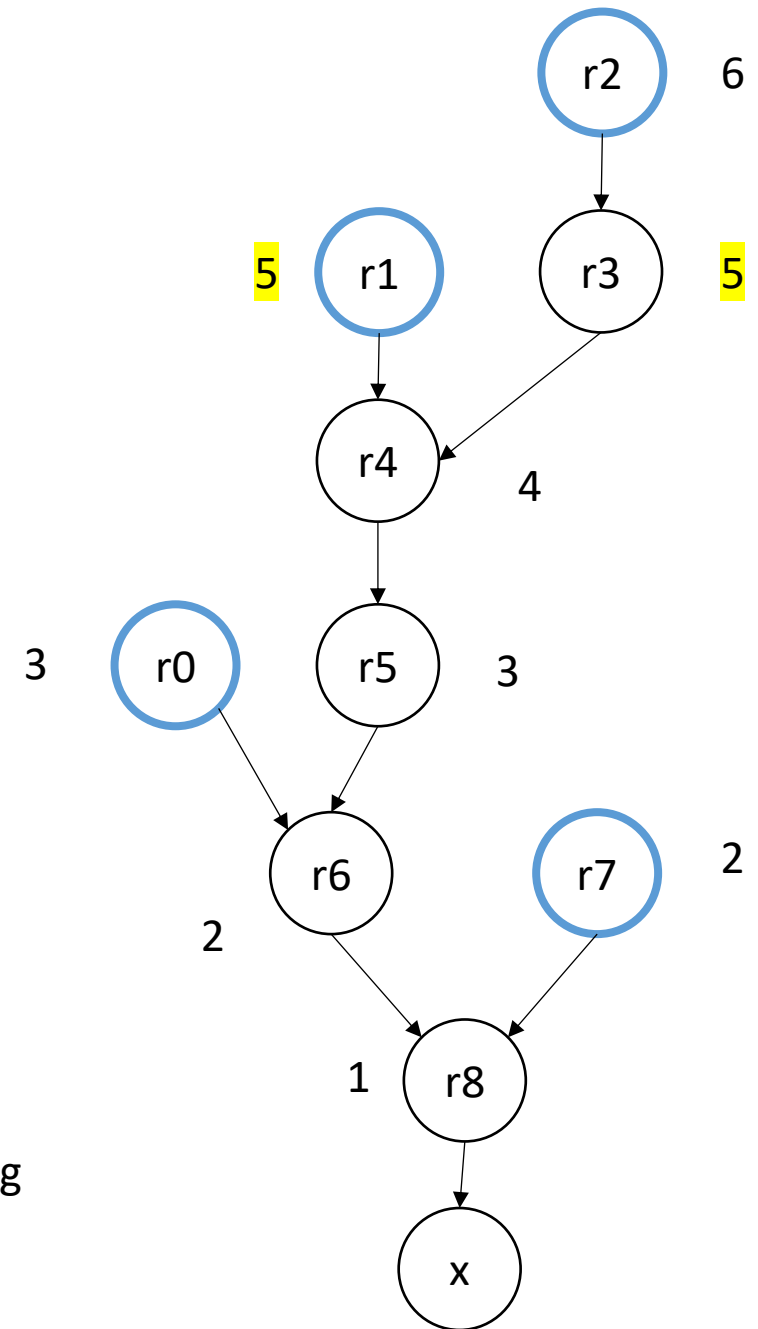


Priority Topological Ordering of DDGs for Pipelining

```
r2 = 4 * a;  
r0 = neg(b);  
r1 = b * b;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;  
r8 = r6 / r7;  
x = r8;
```

Ties are broken with the node that has the least parents

label each node with a distance from the root.
Schedule each node according to the level

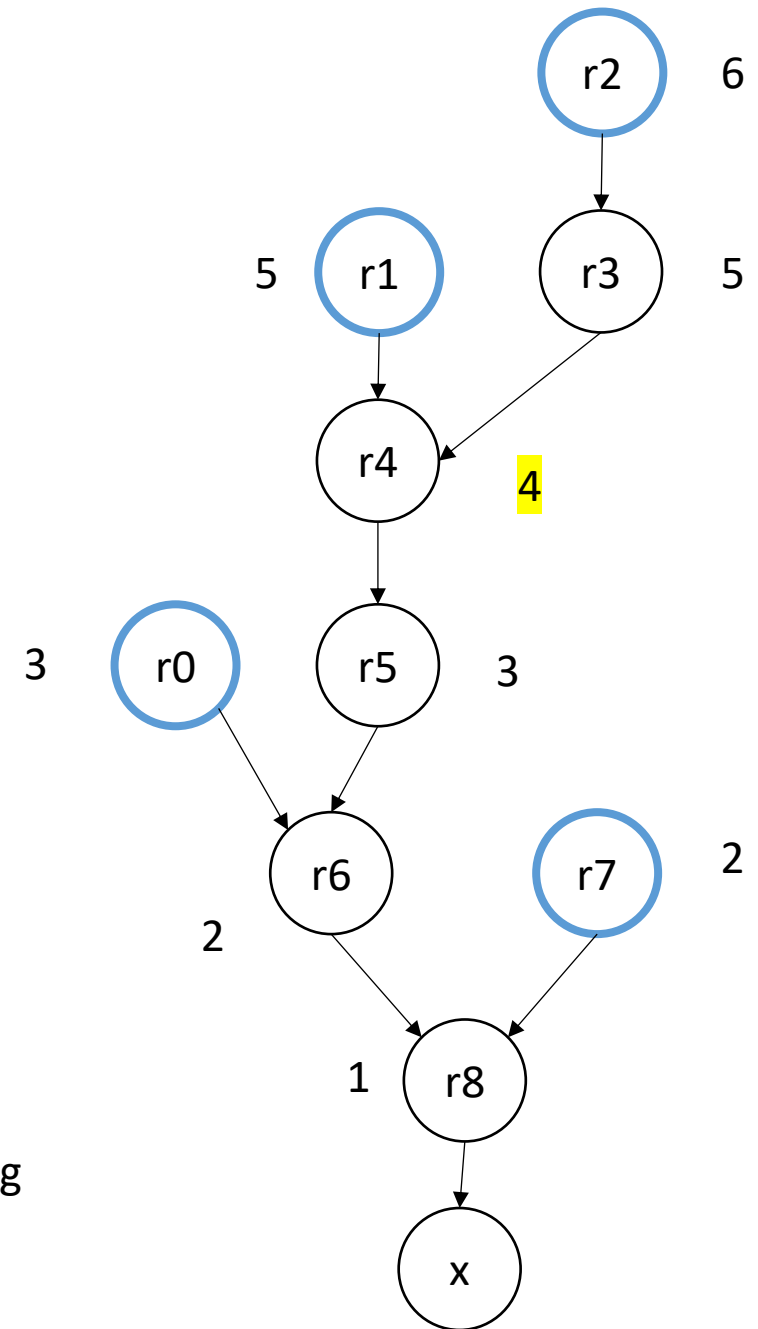


Priority Topological Ordering of DDGs for Pipelining

```
r2 = 4 * a;  
r1 = b * b;  
r3 = r2 * c;  
r0 = neg(b);  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;  
r8 = r6 / r7;  
x = r8;
```

Ties are broken with the node that has the least parents

label each node with a distance from the root.
Schedule each node according to the level



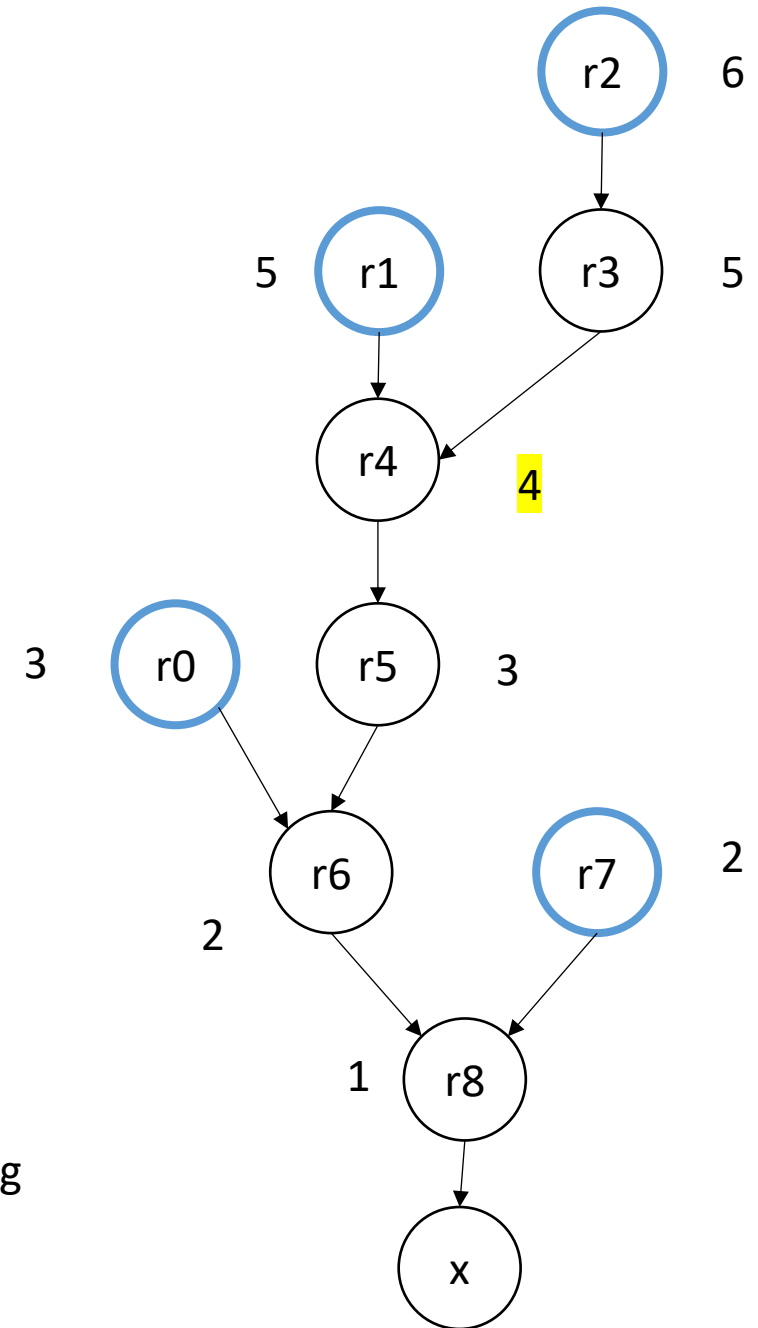
Priority Topological Ordering of DDGs for Pipelining

final

```
r2 = 4 * a;  
r1 = b * b;  
r3 = r2 * c;  
r4 = r1 - r3;  
r0 = neg(b);  
r5 = sqrt(r4);  
r7 = 2 * a;  
r6 = r0 - r5;  
r8 = r6 / r7;  
x = r8;
```

Ties are broken with the node that has the least parents

label each node with a distance from the root.
Schedule each node according to the level



In practice

- real machines are both pipelined and super scalar
- general algorithm for optimal schedules is expensive
- compilers use heuristics:
 - breaking ties in priority ordering
 - abstract performance models

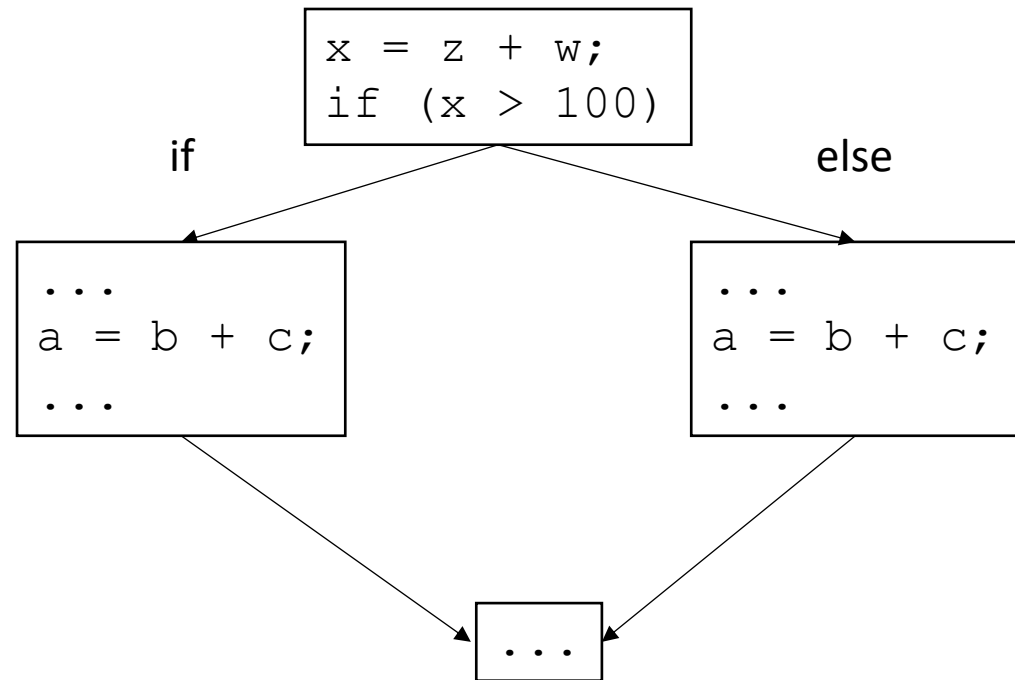
Anticipable Expressions

$$AntOut(n) = \bigcap_{s \text{ in succ}} UExpr(s) \cup (AntOut(s) \cap \overline{ExprKill(s)})$$

An expression e is “anticipable” at a basic block b_x if for all paths that leave b_x , e is evaluated

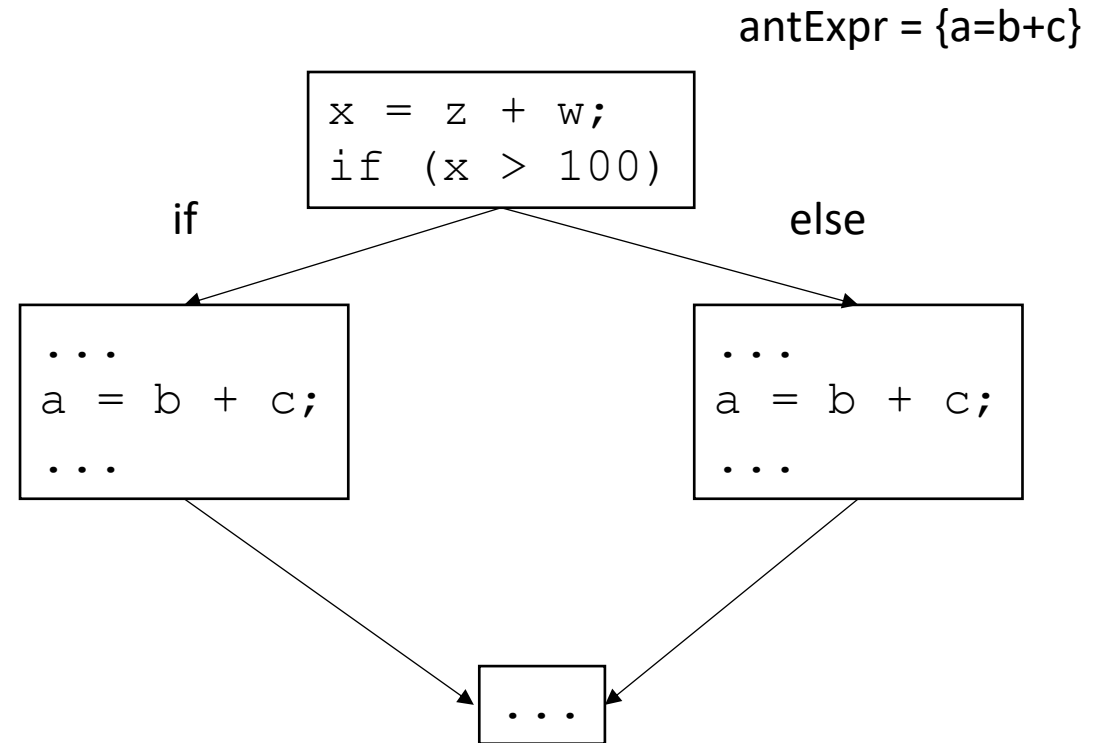
Anticipable Expressions

```
x = z + w;  
if (x > 100) {  
    ...  
    a = b + c;  
    ...  
}  
else {  
    ...  
    a = b + c;  
    ...  
}
```



Anticipable Expressions

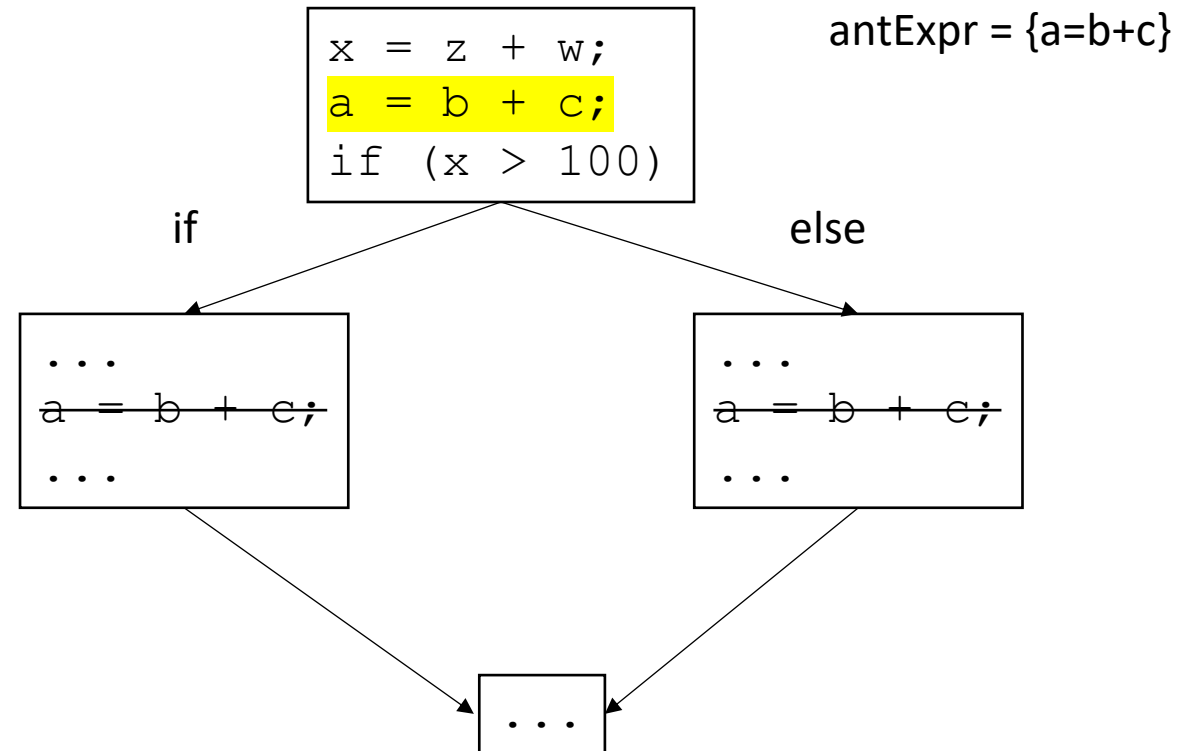
```
x = z + w;  
if (x > 100) {  
    ...  
    a = b + c;  
    ...  
}  
else {  
    ...  
    a = b + c;  
    ...  
}
```



Anticipable Expressions

also called "Upward code motion"

```
x = z + w;  
a = b + c;  
if (x > 100) {  
    ...  
    a = b + c;  
    ...  
}  
else {  
    ...  
    a = b + c;  
    ...  
}
```



Using Loop Unrolling to Exploit ILP

- for loops with independent chains of computation

```
for (int i = 0; i < SIZE; i++) {  
    SEQ(i);  
}
```

where: `SEQ(i) = instr1;`

`instr2;`

`...`

`a[i] = instrN;`

and let `instr(N)` depends on `instr(N-1)`

loops only write to memory
addressed by the loop variable

Using Loop Unrolling to Exploit ILP

- Simple loop unrolling:

```
for (int i = 0; i < SIZE; i+=2) {  
    SEQ(i);  
    SEQ(i+1);  
}
```

*If you know that SIZE
is divisible by 2*

Saves one addition and one comparison per loop, but doesn't help with ILP

Using Loop Unrolling to Exploit ILP

- Simple loop unrolling:

```
for (int i = 0; i < SIZE; i+=2) {  
    SEQ(i);  
    SEQ(i+1);  
}
```

Let **green highlights** indicate instructions from iteration i .

Let **blue highlights** indicate instructions from iteration $i + 1$.

Using Loop Unrolling to Exploit ILP

- Simple loop unrolling:

```
for (int i = 0; i < SIZE; i+=2) {  
    SEQ(i);  
    SEQ(i+1);  
}
```

Let $SEQ(i, j)$ be the j th instruction of $SEQ(i)$.

Let each instruction chain have N instructions

Using Loop Unrolling to Exploit ILP

- Simple loop unrolling:

```
for (int i = 0; i < SIZE; i+=2) {  
    SEQ(i, 1);  
    SEQ(i, 2);  
    ...  
    SEQ(i, N); // end iteration for i  
    SEQ(i+1, 1);  
    SEQ(i+1, 2);  
    ...  
    SEQ(i+1, N); // end iteration for i + 1  
}
```

Let $SEQ(i, j)$ be the j th instruction of $SEQ(i)$.

Let each instruction chain have N instructions

Using Loop Unrolling to Exploit ILP

- Simple loop unrolling:

```
for (int i = 0; i < SIZE; i+=2) {  
    SEQ(i, 1);  
    SEQ(i+1, 1);  
    SEQ(i, 2);  
    SEQ(i+1, 2);  
    ...  
    SEQ(i, N);  
    SEQ(i+1, N);  
}
```

They can be interleaved

Using Loop Unrolling to Exploit ILP

- Simple loop unrolling:

```
for (int i = 0; i < SIZE; i+=2) {  
    SEQ(i, 1);  
    SEQ(i+1, 1);  
    SEQ(i, 2);  
    SEQ(i+1, 2);  
    ...  
    SEQ(i, N);  
    SEQ(i+1, N);  
}
```

They can be interleaved

two instructions can be pipelined, or executed on a superscalar processor

Loop Unrolling for Reduction Loops

- Prior approach examined loops with independent iterations and chains of dependent computations
- Now we will look at reduction loops:
 - Entire computation is dependent
 - Typically short bodies (addition, multiplication, max, min)

1	2	3	4	5	6
---	---	---	---	---	---

addition: 21

max: 6

min: 1

Loop Unrolling for Reduction Loops

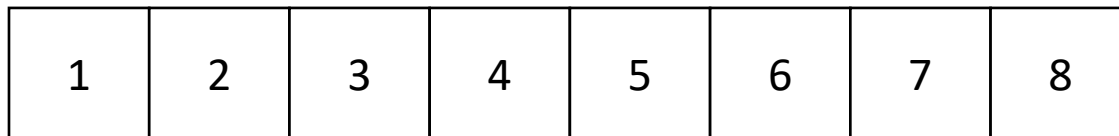
- Simple implementation:

```
for (int i = 1; i < SIZE; i++) {  
    a[0] = REDUCE(a[0], a[i]);  
}
```

If the reduction operator is associative, we can do better!

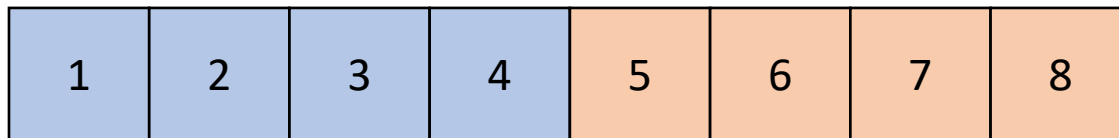
Loop Unrolling for Reduction Loops

- chunk array in equal sized partitions and do local reductions
- Consider size 2:



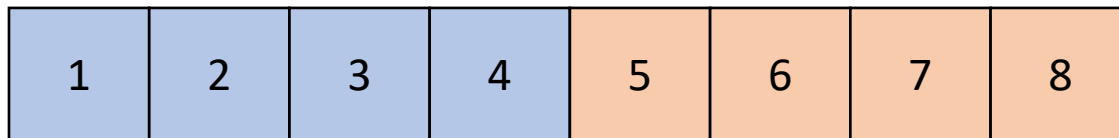
Loop Unrolling for Reduction Loops

- chunk array in equal sized partitions and do local reductions
- Consider size 2:



Loop Unrolling for Reduction Loops

- chunk array in equal sized partitions and do local reductions
- Consider size 2:



Do addition reduction in base memory location

Loop Unrolling for Reduction Loops

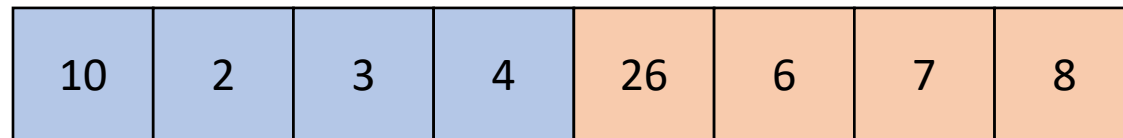
- chunk array in equal sized partitions and do local reductions
- Consider size 2:



Do addition reduction in base memory location

Loop Unrolling for Reduction Loops

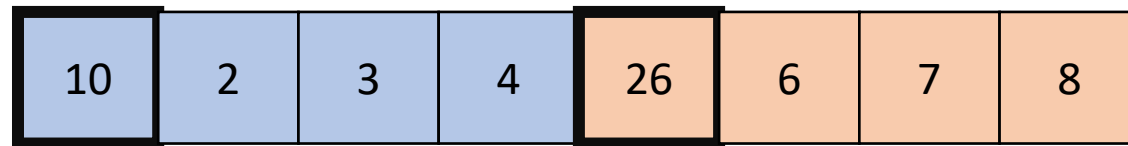
- chunk array in equal sized partitions and do local reductions
- Consider size 2:



Add together base locations

Loop Unrolling for Reduction Loops

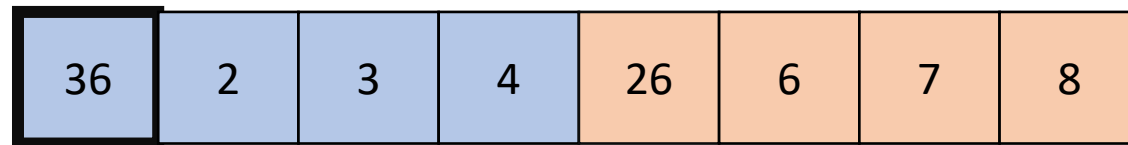
- chunk array in equal sized partitions and do local reductions
- Consider size 2:



Add together base locations

Loop Unrolling for Reduction Loops

- chunk array in equal sized partitions and do local reductions
- Consider size 2:



Add together base locations

Loop Unrolling for Reduction Loops

- Simple implementation:

```
for (int i = 1; i < SIZE/2; i++) {  
    a[0] = REDUCE(a[0], a[i]);  
    a[SIZE/2] = REDUCE(a[SIZE/2], a[(SIZE/2)+i]);  
}
```

```
a[0] = REDUCE(a[0], a[SIZE/2])
```

Loop Unrolling for Reduction Loops

- Simple implementation:

```
for (int i = 1; i < SIZE/2; i++) {  
    a[0] = REDUCE(a[0], a[i]);  
    a[SIZE/2] = REDUCE(a[SIZE/2], a[(SIZE/2)+i]);  
}
```

```
a[0] = REDUCE(a[0], a[SIZE/2])
```

Loop Unrolling for Reduction Loops

- Simple implementation:

```
for (int i = 1; i < SIZE/2; i++) {  
    a[0] = REDUCE(a[0], a[i]);  
    a[SIZE/2] = REDUCE(a[SIZE/2], a[(SIZE/2)+i]);  
}
```

```
a[0] = REDUCE(a[0], a[SIZE/2])
```

*independent
instructions
can be done
in parallel!*

Watch out!

- Our abstraction: separate dependent instructions as far as possible
- Pros:
 - Simple
- Cons:
 - Can lead to register spilling, causing expensive loads

consider `instr1` and `instr2` have a data dependence, and `instrX`'s are independent

```
instr1;
```

```
instrX0;
```

```
instrX1;
```

```
...
```

```
instr2;
```

independent instructions. If they overwrite the register storing `instr1`'s result, then it will have to be stored to memory and retrieved before `instr2`

Watch out!

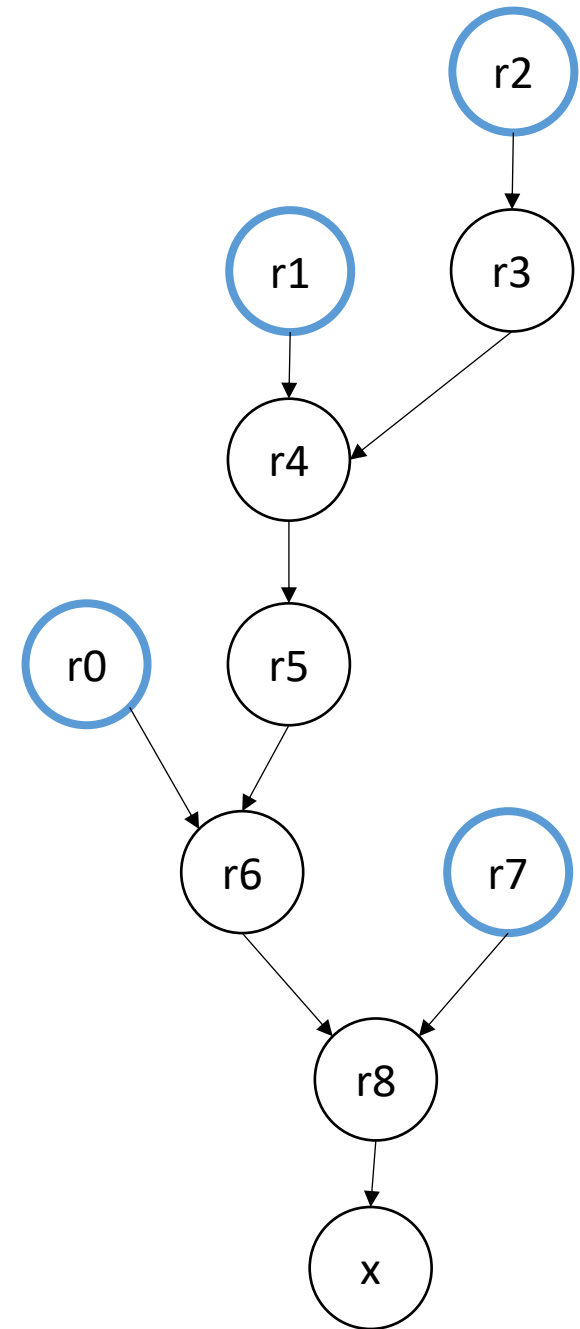
- Our abstraction: separate dependent instructions as far as possible
- Pros:
 - Simple
- Cons:
 - Can lead to register spilling, causing expensive loads

Solutions include using a **resource model** to guide the topological ordering. Highly architecture dependent. Algorithms become more expensive

Typically doesn't show up in basic block analysis. In loop unrolling, it will influence the number of unrolls you do.

Priority Topological Ordering of DDGs

```
r7 = 2 * a;  
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r8 = r6 / r7;  
x = r8;
```



Discussion

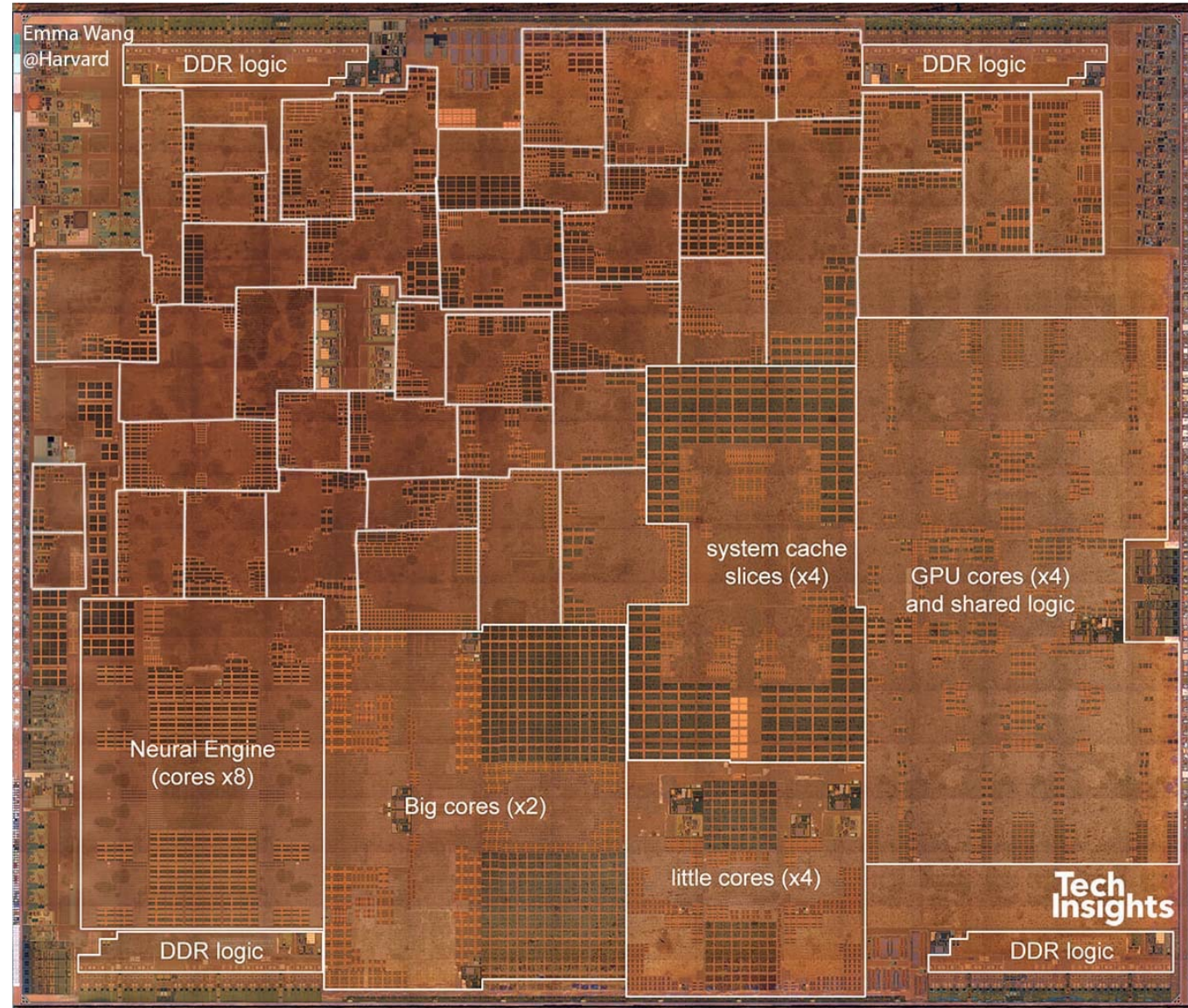
- Where is parallelism most commonly found?
 - Non-numeric applications are thought to have lots of dependencies:
 - I/O (file, network, user),
 - OS, event-driven
 - *[source needed]*
 - numeric applications have less dependencies:
 - media processing (image, video, sound)
 - machine-learning
- More and more, numeric applications are moving to accelerators

Modern SoC

- From David Brooks lab at Harvard:

<http://vlsiarch.eecs.harvard.edu/research/accelerators/di-e-photo-analysis/>

- Compilers will need to be able to map software efficiently to a range of different accelerators



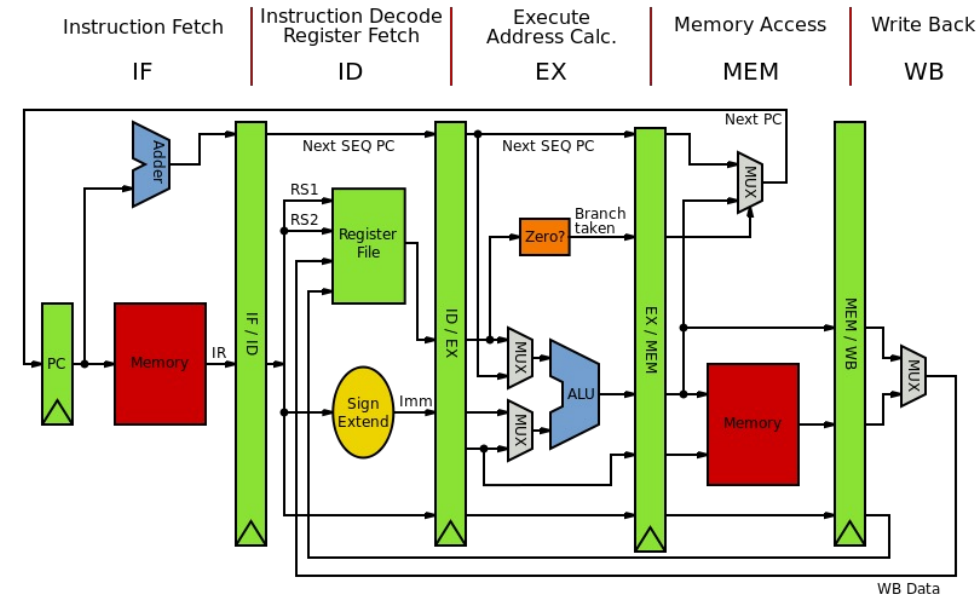
Current tensions

- Simple cores with accelerators/GPUs?
 - Less need for pipelines, OoO, and superscalar
 - Hard to port legacy code
- Complicated cores
 - area/power hungry
 - great for legacy code
- Where do compilers fit in?

Moving on to Symmetric Multiprocessing (SMP)

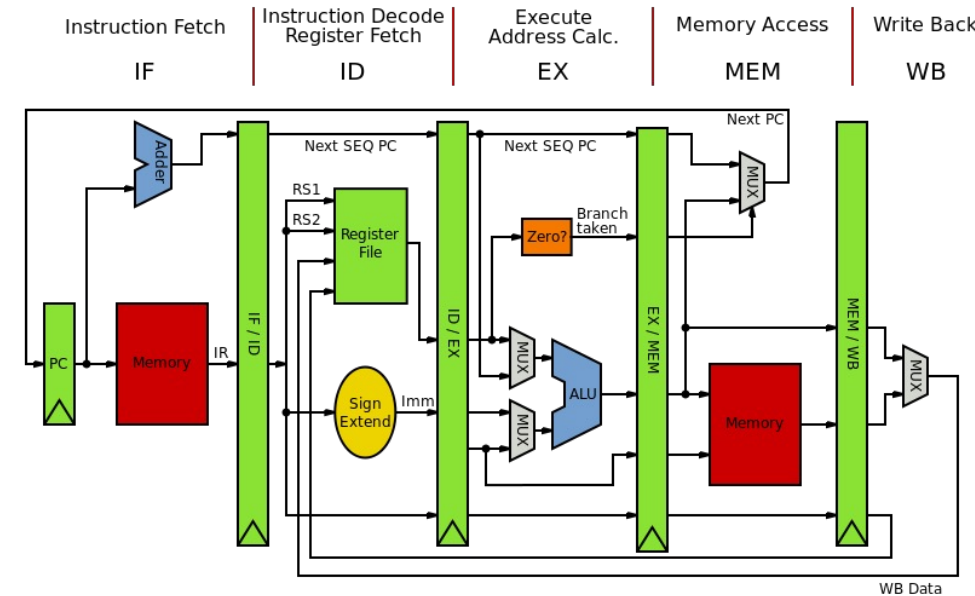
Limits of ILP?

- Pipelines?
 - Only so much meaningful work to do per-stage.
 - Stage timing imbalance
 - Staging overhead
- Superscalar width?
 - Hardware checking becomes prohibitive:



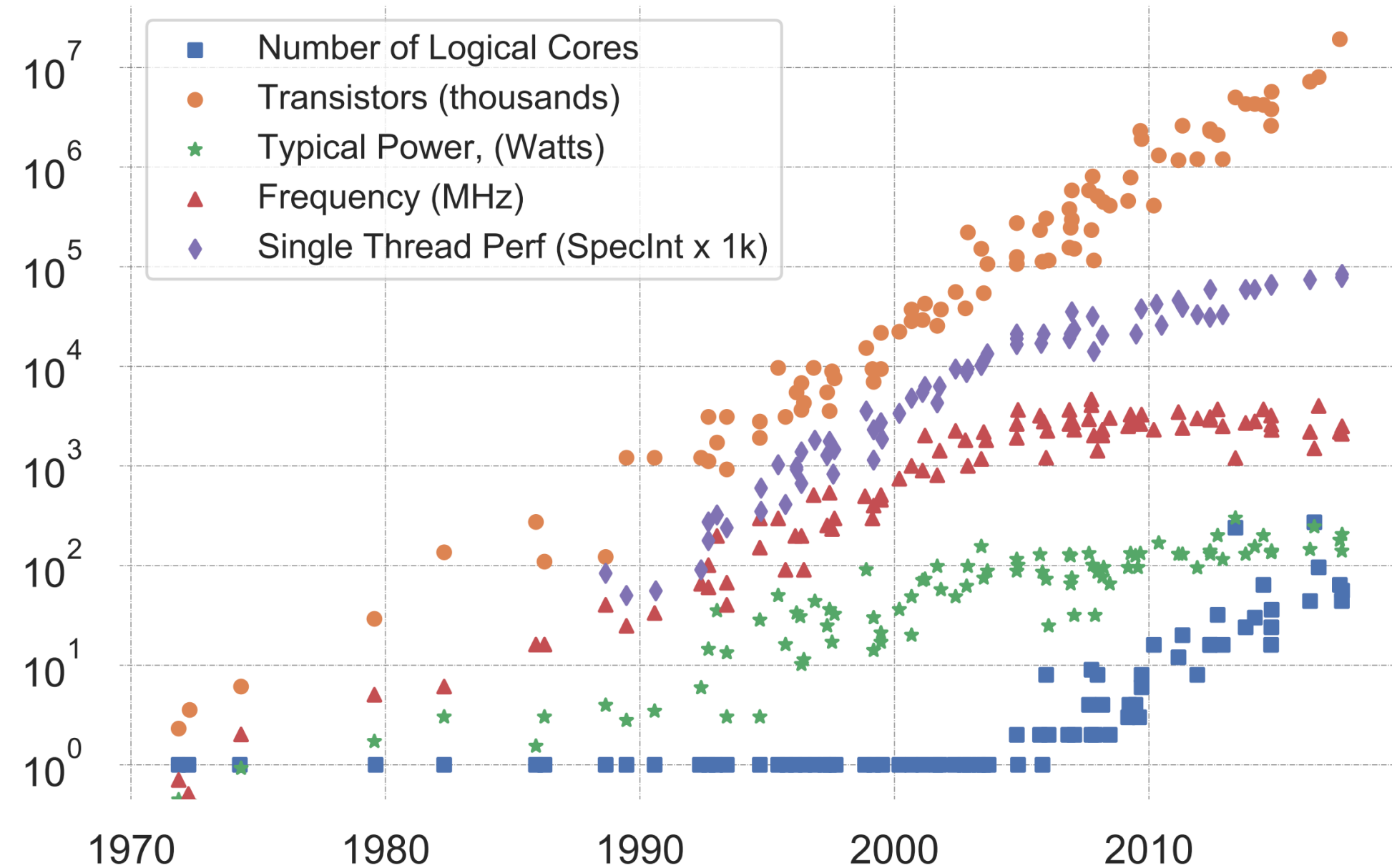
Limits of ILP

- Pipelines?
 - Only so much meaningful work to do per-stage.
 - Stage timing imbalance
 - Staging overhead
- Superscalar width?
 - Hardware checking becomes prohibitive:



Collectively the [power consumption](#), complexity and gate delay costs limit the achievable superscalar speedup to roughly eight simultaneously dispatched instructions.

https://en.wikipedia.org/wiki/Superscalar_processor#Limitations



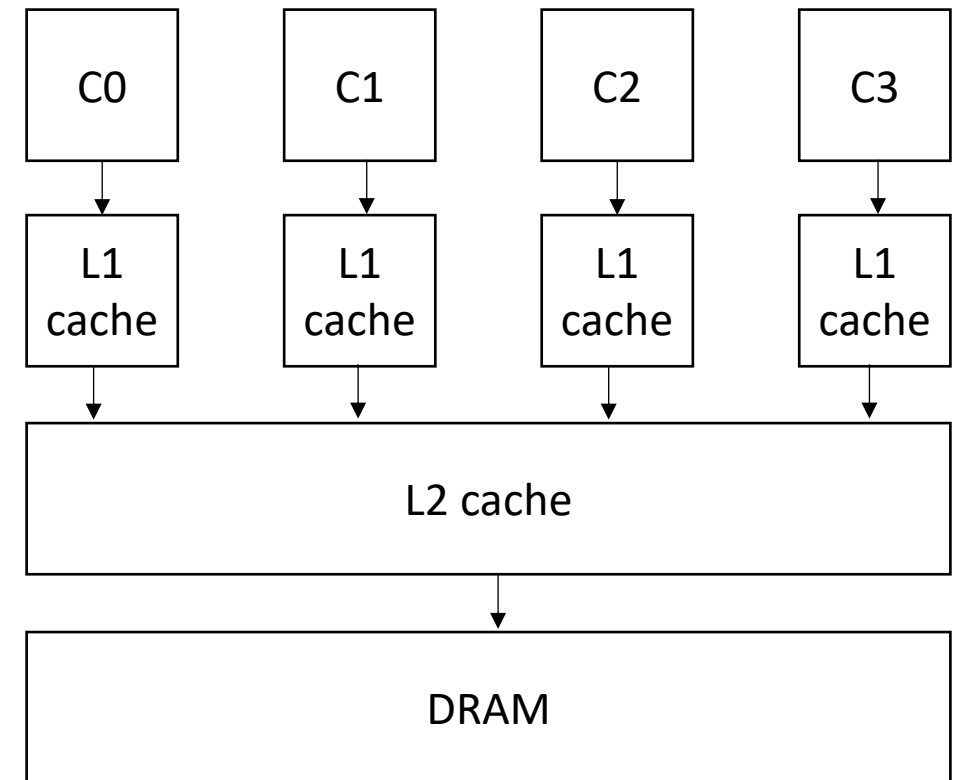
K. Rupp, "40 Years of Microprocessor Trend Data," <https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data>, 2015.

Trends

- Frequency scaling: **Dennard's scaling**
 - Mostly agreed that this is over
- Number of transistors: **Moore's law**
 - On its last legs?
 - Intel delayed 7nm chips (out now?). Apple has a 5nm. Roadmaps go to 3nm, or 1.8nm
- *Chips are not increasing in raw frequency, and space is becoming more valuable*

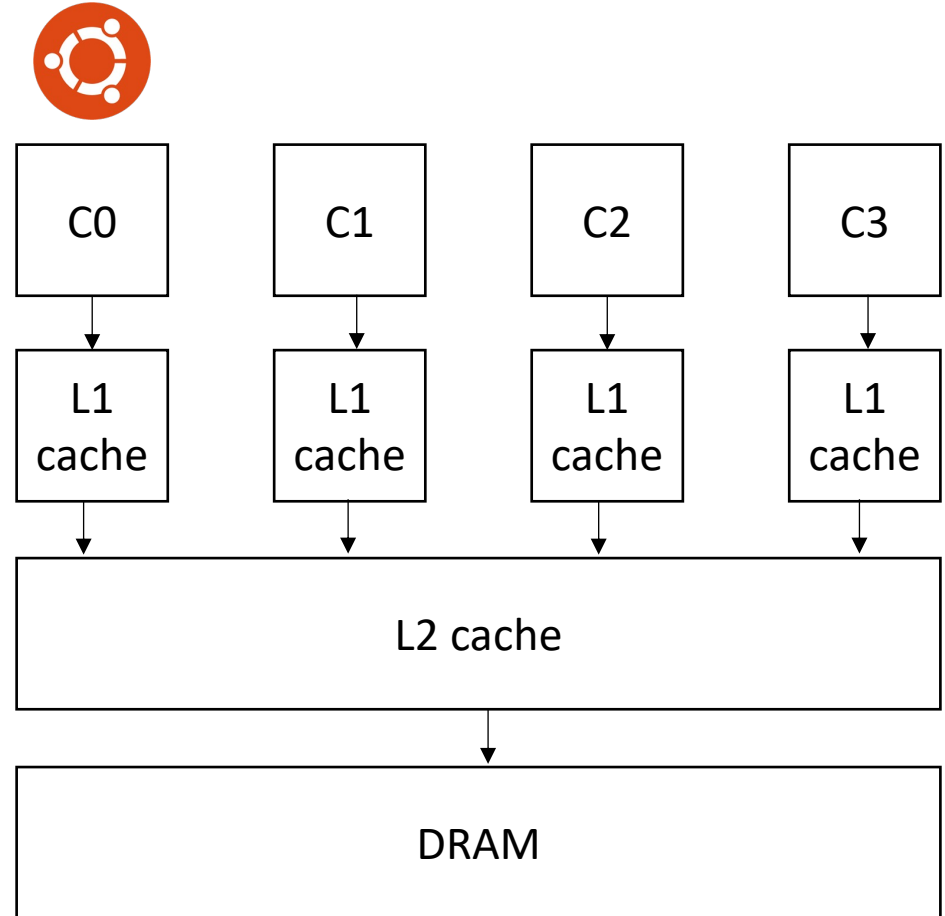
Symmetric Multiprocessing (SMP)

- Collection of “identical” cores
 - Shared memory (access to all system resources)
 - Managed by a single OS
- Pros:
 - Simple(r) HW design
 - Great for multitasking machines



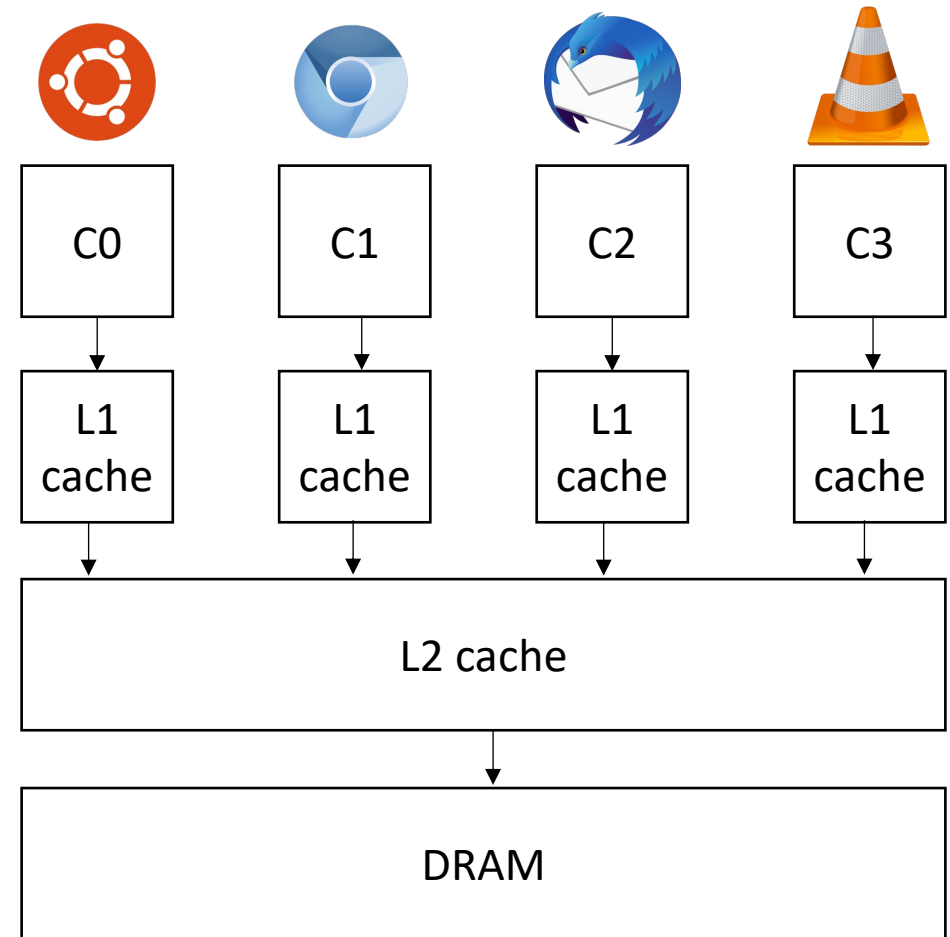
Symmetric Multiprocessing (SMP)

- Collection of “identical” cores
 - Shared memory (access to all system resources)
 - Managed by a single OS
- Pros:
 - Simple(r) HW design
 - Great for multitasking machines



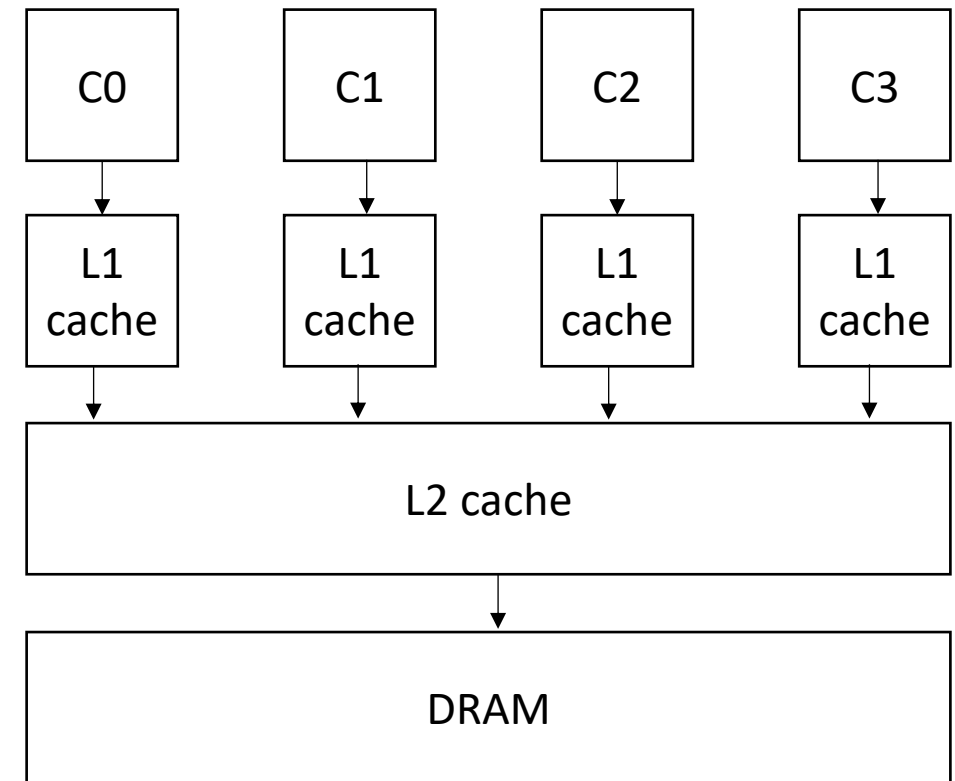
Symmetric Multiprocessing (SMP)

- Collection of “identical” cores
 - Shared memory (access to all system resources)
 - Managed by a single OS
- Pros:
 - Simple(r) HW design
 - Great for multitasking machines



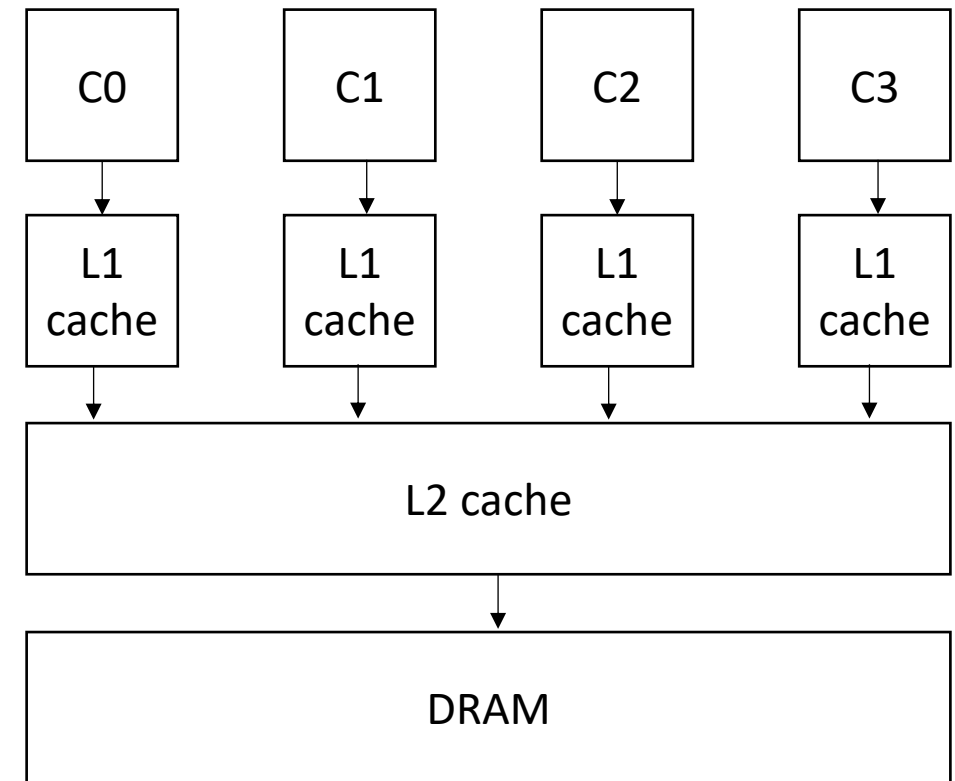
Symmetric Multiprocessing (SMP)

- Collection of “identical” cores
 - Shared memory (access to all system resources)
 - Managed by a single OS
- Pros:
 - Simple(r) HW design
 - Great for multitasking machines
 - Can provide (close to) linear speedups for parallel applications



Symmetric Multiprocessing (SMP)

- Collection of “identical” cores
 - Shared memory (access to all system resources)
 - Managed by a single OS
- Pros:
 - Simple(r) HW design
 - Great for multitasking machines
 - Can provide (close to) linear speedups for parallel applications
- Cons: difficult to program!



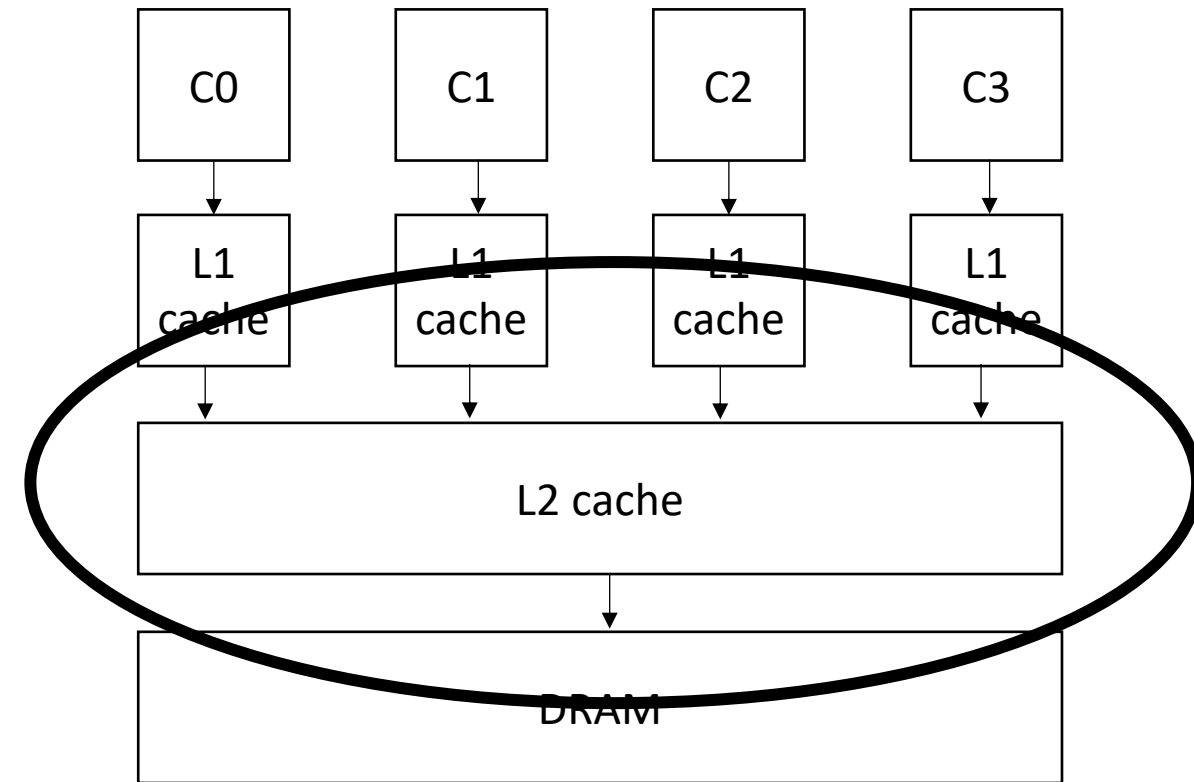
SMP systems are widespread

- Laptops
 - My laptop has 8 cores
 - Most have at least 2
 - New Macbook: 16 core
- Workstations:
 - 2 - 64 cores
 - ARM racks: 128
- Phones:
 - iPhone: 2 big cores, 4 small cores
 - Samsung: 1 + 3 + 4

*<https://www.crn.com/news/components-peripherals/ampere-s-new-128-core-altra-cpu-targets-intel-amd-in-the-cloud>

SMP systems are widespread

- Laptops
 - My laptop has 8 cores
 - Most have at least 2
 - New Macbook: 10 core
- Workstations:
 - 2 - 64 cores
 - ARM racks: 128
- Phones:
 - iPhone: 2 big cores, 4 small cores
 - Samsung: 1 + 3 + 4



*<https://www.crn.com/news/components-peripherals/ampere-s-new-128-core-altra-cpu-targets-intel-amd-in-the-cloud>

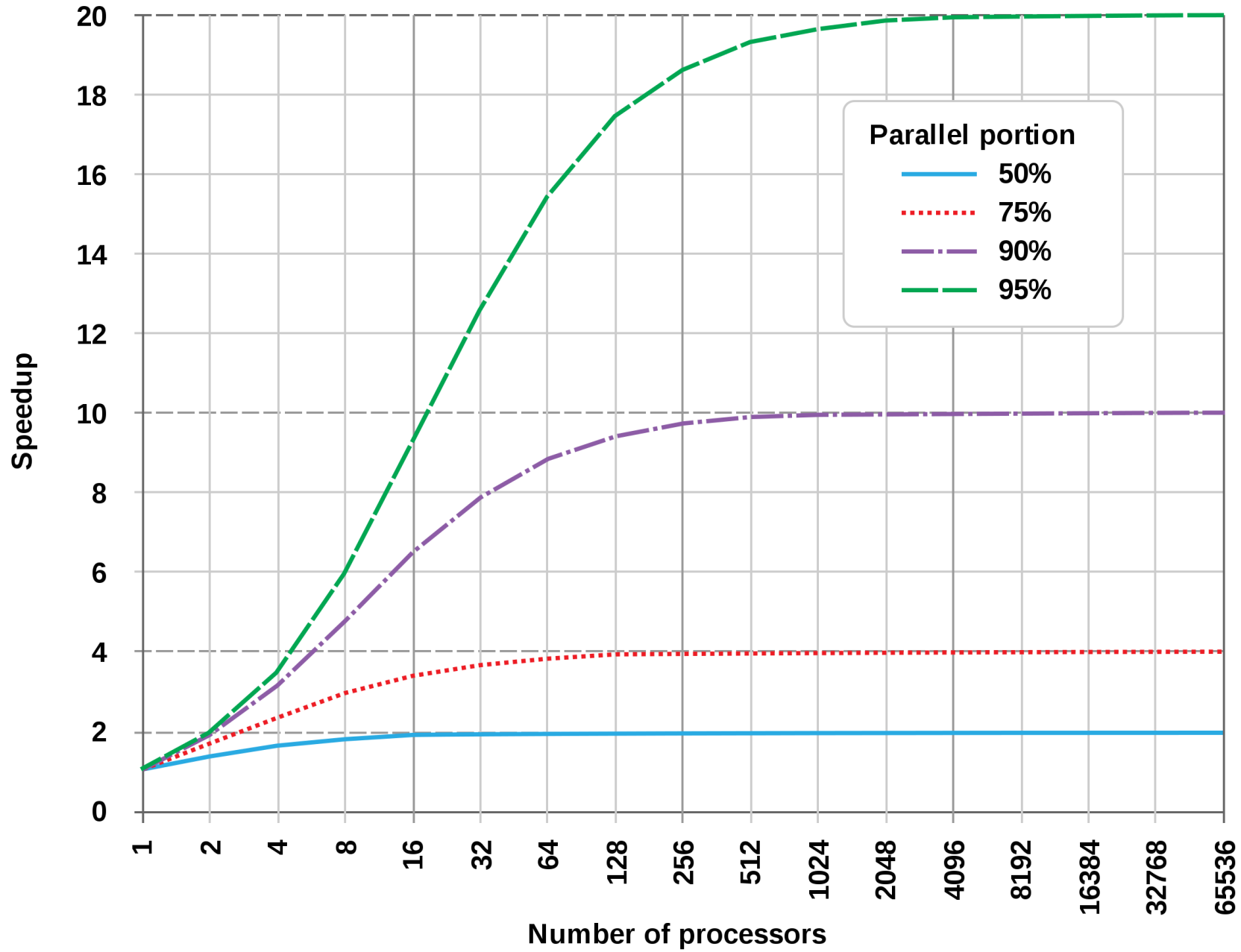
Potential for Parallel Speedup

- Amdahl's law

- $Speedup(c) = \frac{1}{(1-p) + \frac{p}{c}}$

- Where c is the number of cores and p is the percentage of the program execution time that would be improved by parallelism
- Assumes linear speedups

Amdahl's Law



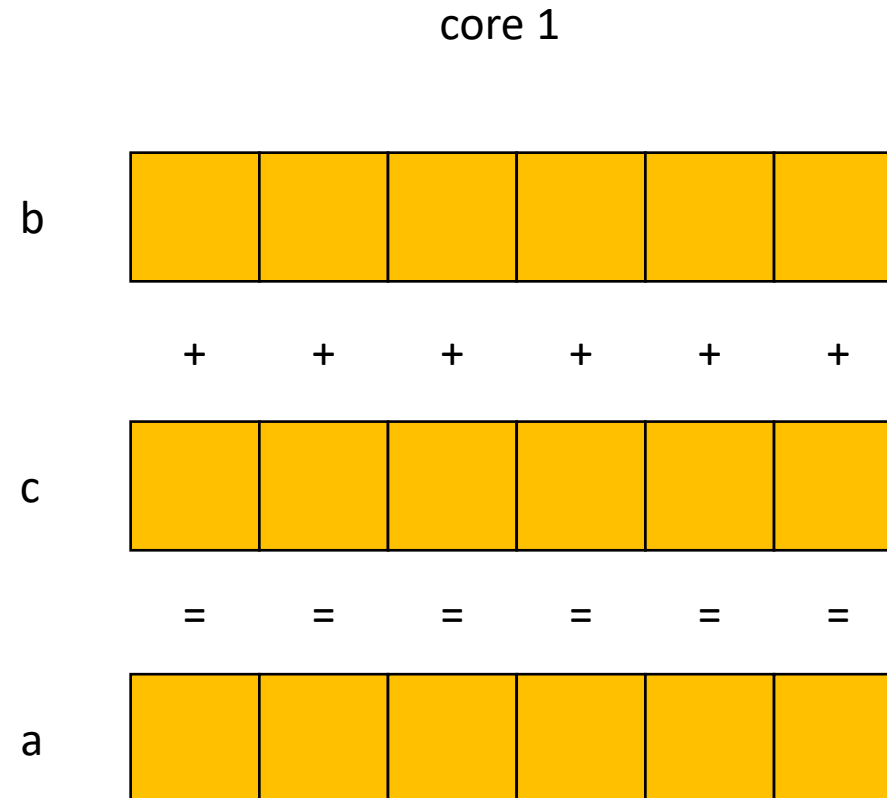
from wikipedia

Can compilers help?

- Much like ILP: convert sequential streams of computation in to SMP parallel code.
- Much harder constraints
 - Correctness
 - Performance
- For loops are a good target for compiler analysis

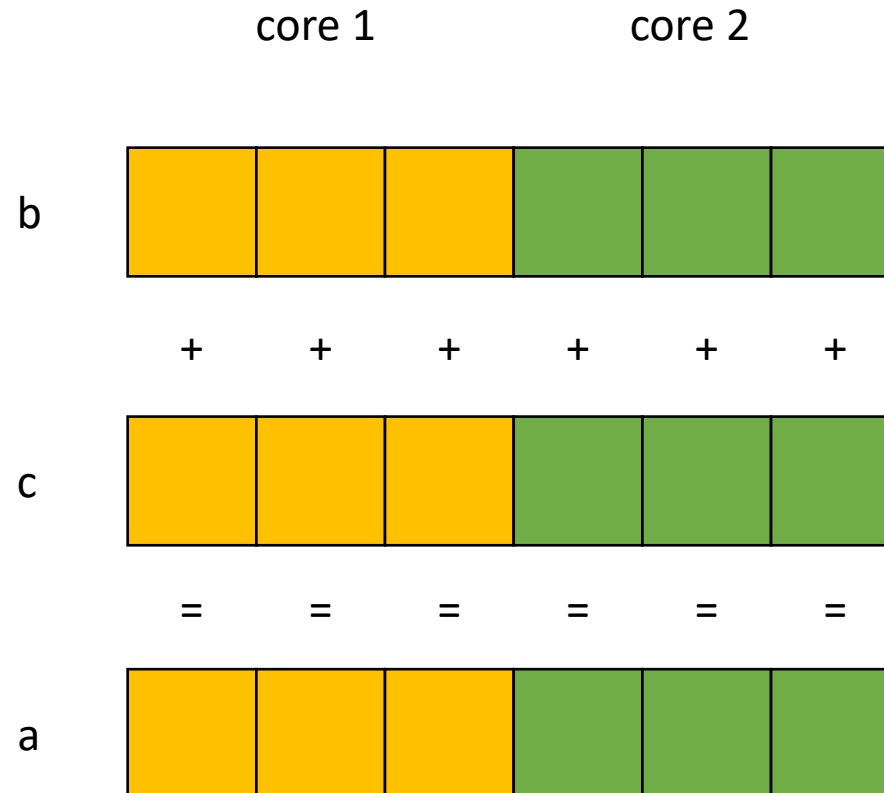
For loops are great candidates for SMP parallelism

```
for (int i = 0; i < 6; i++) {  
    a[i] = b[i] + c[i]  
}
```



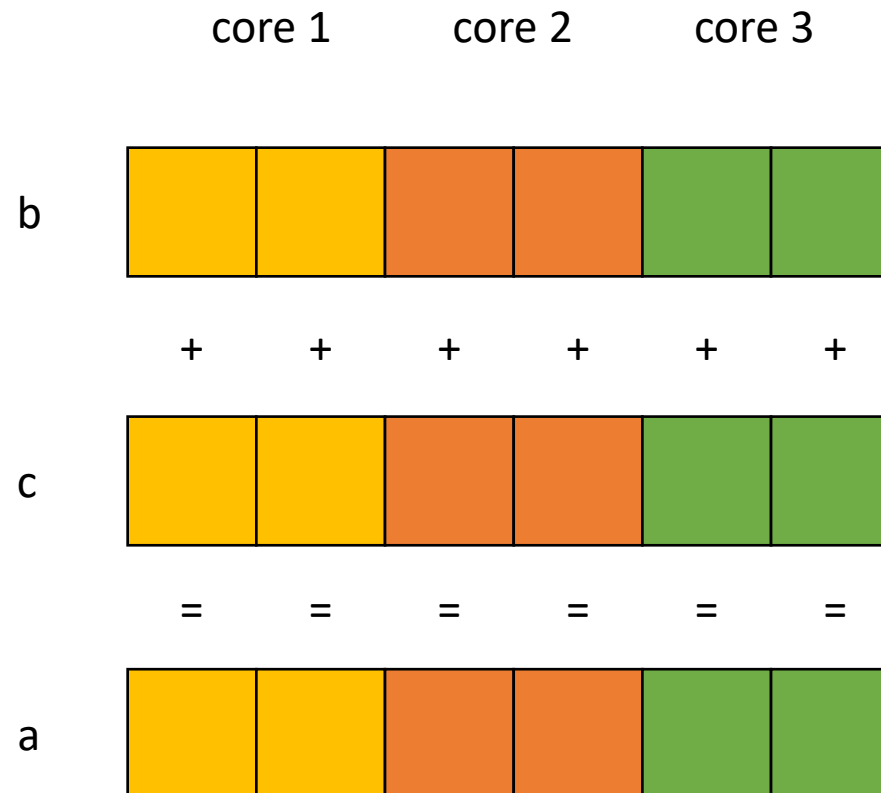
For loops are great candidates for SMP parallelism

```
for (int i = 0; i < 6; i++) {  
    a[i] = b[i] + c[i]  
}
```



For loops are great candidates for SMP parallelism

```
for (int i = 0; i < 6; i++) {  
    a[i] = b[i] + c[i]  
}
```



See you on Next time!

- DOALL For loops