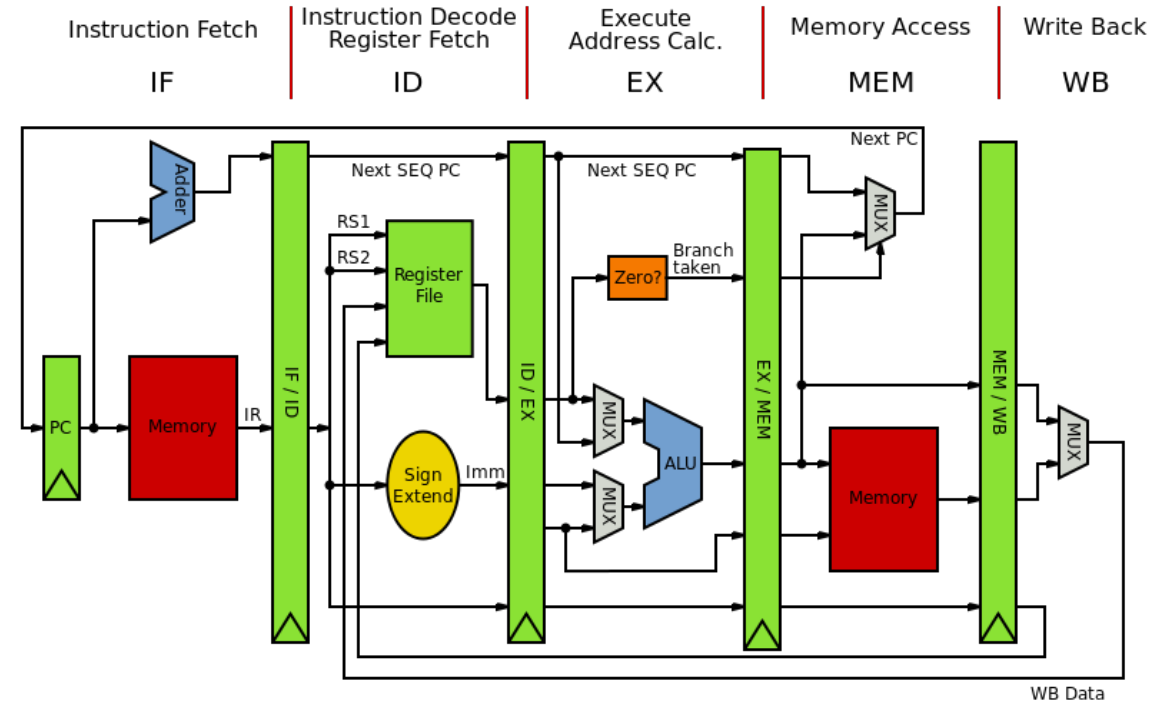


CSE113: Parallel Programming

Jan. 10, 2022

- **Topics:**

- False sharing
- Instruction level parallelism (ILP)
- Loop unrolling



MIPS pipeline image from:

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

Announcements

- Office hours and tutors are available this week!
 - Announcements on Canvas and Piazza with zoom links for tutors
 - Reese will post his before his office hours
 - I post mine around noon on Thursday (along with a sign up sheet)
- Homework 1 is released
 - You can get started setting up the docker
 - After today you can do part 1
 - After Wednesday you can do part 2
 - After Friday you can do part 3

Let us know about any typos in the homework!

Today's Quiz

- We'll continue having them given after lecture for now
- Make sure to do it!

Previous quiz

Some answers

The following statement in a language like C or Java would be compiled to how many instructions in low-level code?

```
z = x + x + x + x;
```

0		0%	✓
1	3 respondents	6%	
2	3 respondents	6%	
4	44 respondents	88%	

Some answers

How many levels of caches does a typical x86 system have?

1		0 %	✓
2	7 respondents	14 %	
3	37 respondents	74 %	
4	6 respondents	12 %	

Some answers

Write a few reasons why it may be difficult to reason about program performance when using a high-level language like Python

Some answers

Using your best guess, how much faster do you think a program written in C/Java is than a program written in Python? Give a few reasons explaining your guess. Feel free to run a simple experiment and see what happens!

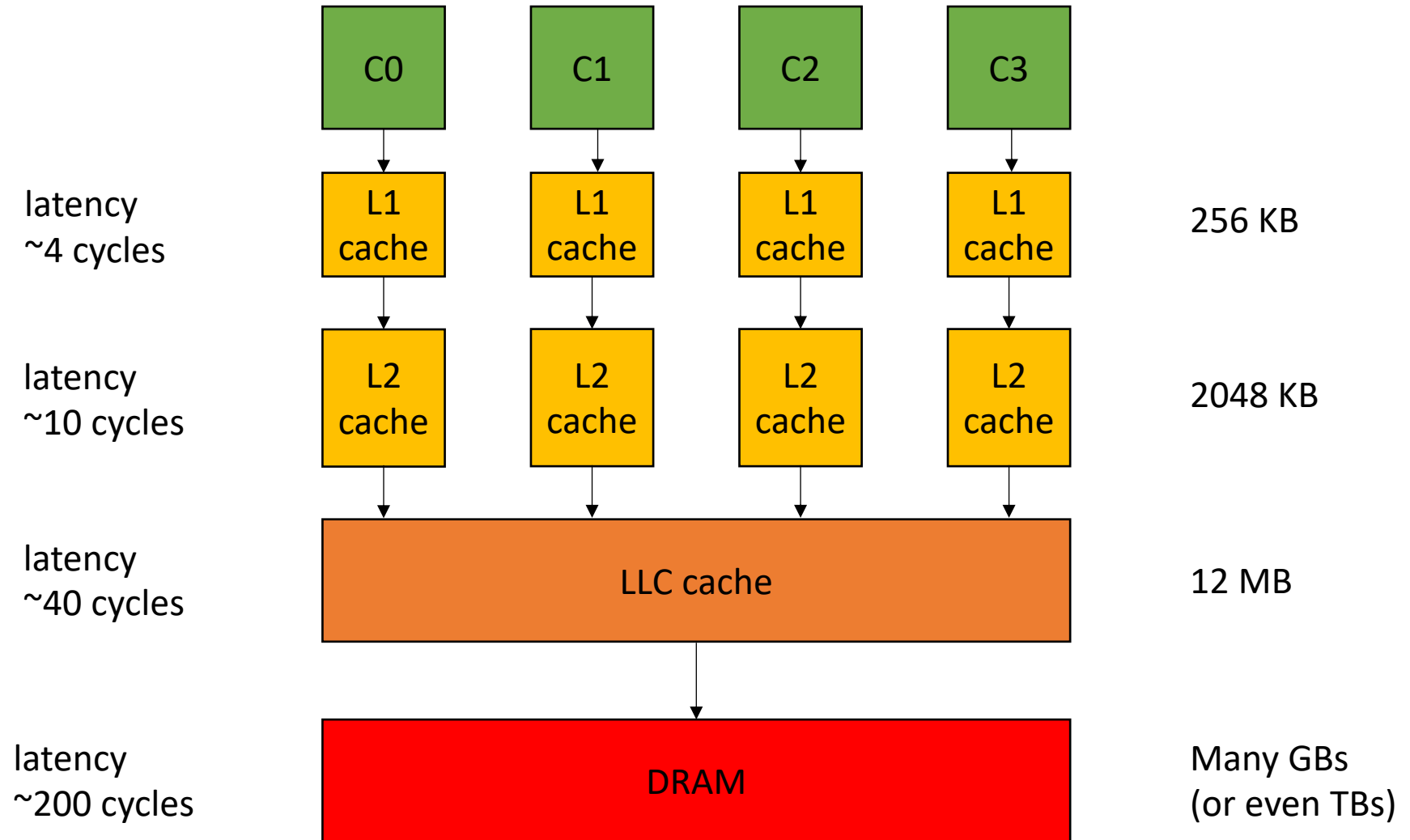
Thanks!

- Thanks for all the interesting answers on quizzes!
- For those of you who have not done the quizzes, please do so, submitting them counts towards your grade!

Review

- Caches

Caches

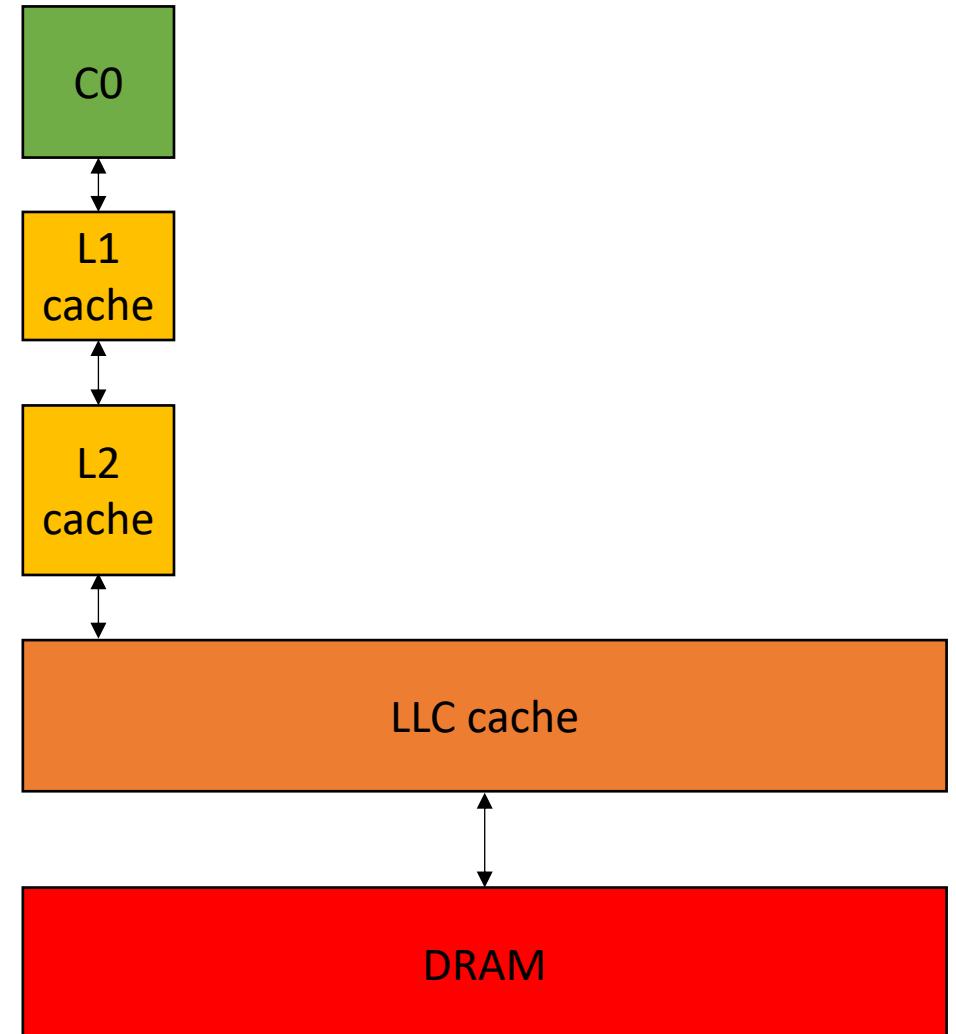


Cache lines

```
int increment(int *a) {  
    a[0]++;  
}
```

```
%5 = load i32, i32* %4  
%6 = add nsw i32 %5, 1  
store i32 %6, i32* %4
```

Assume a[0] is not in the cache



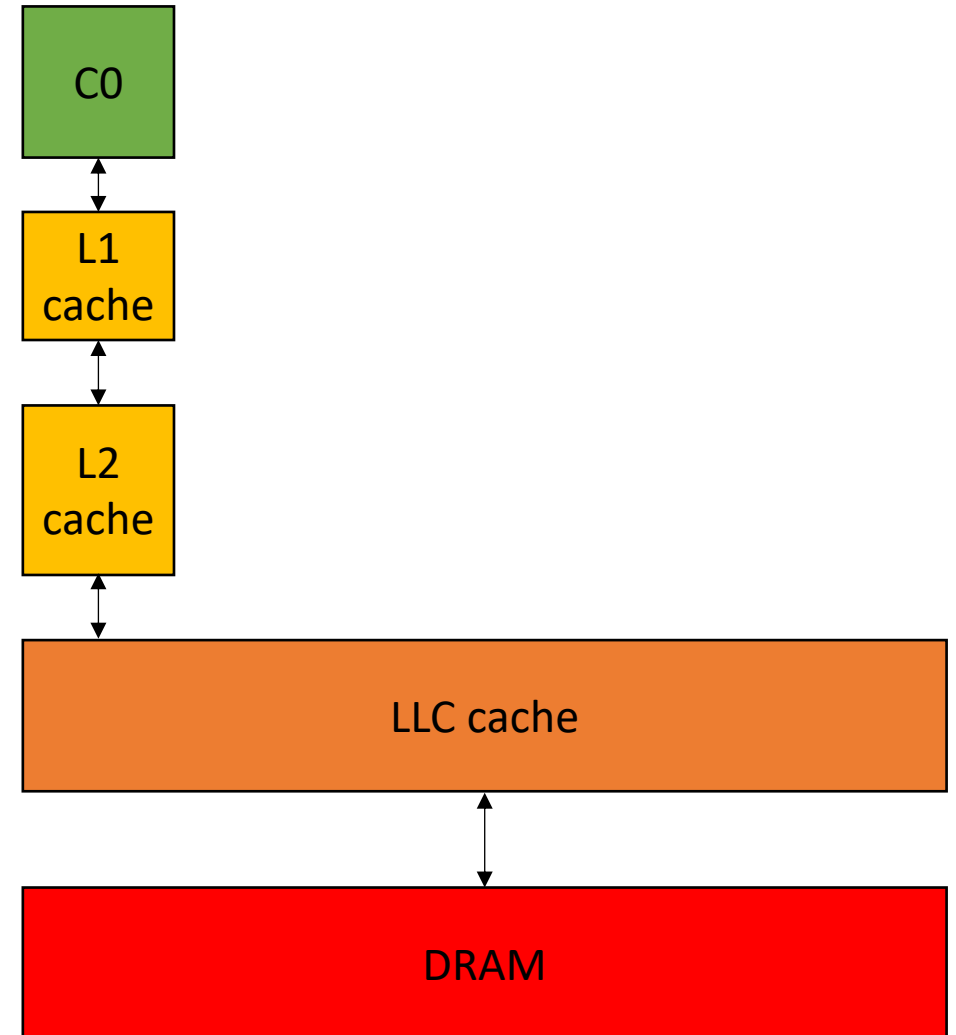
Cache lines

```
int increment(int *a) {  
    a[0]++;  
}
```

```
%5 = load i32, i32* %4  
%6 = add nsw i32 %5, 1  
store i32 %6, i32* %4
```

$a[0] - a[15]$

Assume $a[0]$ is not in the cache



Cache organization

Cache organization

Direct mapped: every memory location can go exactly one place in the cache.

cache block location = $(\text{address}/64) \% (\text{cache size})$

Cache

value	0	7	2			
address	0x00	0x1C0	0x80			

Example: Read address 0x180

Memory

value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440

Cache organization

N-way Associative: every memory location can go N places in the cache.

cache block location $(\text{address}/64) \% (\text{cache size} / N)$

More places to store collisions.

Cache

value	0	7	2	<i>set 1</i>
address	0x00	0x1C0	0x80	

value				<i>set 2</i>
address				

Read 0x180

example 2-way associative

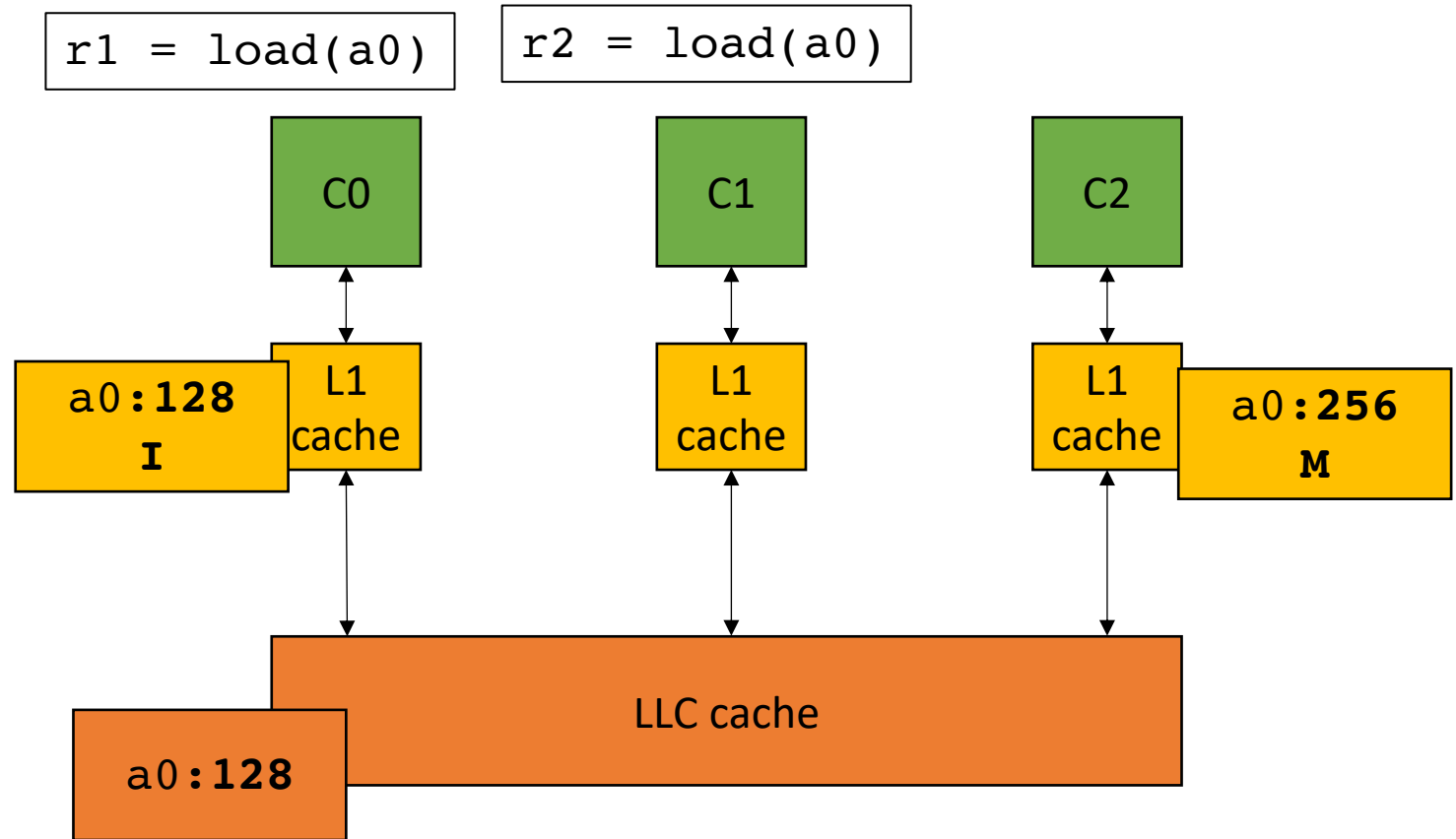
Memory

value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440

Cache coherence

- MESI protocol
- Cache line can be in 1 of 4 states:
 - **Modified** - the cache contains a modified value and it must be written back to the lower level cache
 - **Exclusive** - only 1 cache has a copy of the value
 - **Shared** - more than 1 cache contains the value, they must all agree on the value
 - **Invalid** - the data is stale and a new value must be fetched from a lower level cache

Cache coherence



On to the lecture!

Lecture Schedule

- False Sharing
- Instruction Level parallelism
- Loop unrolling

Lecture Schedule

- **False Sharing**
- Instruction Level parallelism
- Loop unrolling

Example

- A function that increments a memory location ITERATION times

```
void repeat_increment(volatile int *a) {  
    for (int i = 0; i < ITERATIONS; i++) {  
        int tmp = *a;  
        tmp += 1;  
        *a = tmp;  
    }  
}
```

Example

- A function that increments a memory location ITERATION times

guarantees that memory accesses are not optimized!

```
void repeat_increment(volatile int *a) {  
    for (int i = 0; i < ITERATIONS; i++) {  
        int tmp = *a;  
        tmp += 1;  
        *a = tmp;  
    }  
}
```

Example

- A function that increments a memory location ITERATION times
- Do this for 8 elements:
 - Allocate a contiguous array

Example

- A function that increments a memory location ITERATION times
- Do this for 8 elements:
 - Allocate a contiguous array
- Loop through the 8 elements and increment each one:

```
for (int i = 0; i < NUM_ELEMENTS; i++) {  
    repeat_increment(a+i);  
}
```

Example

- We can also do each array element in parallel!

```
for (int i = 0; i < NUM_ELEMENTS; i++) {  
    repeat_increment(a+i);  
}
```

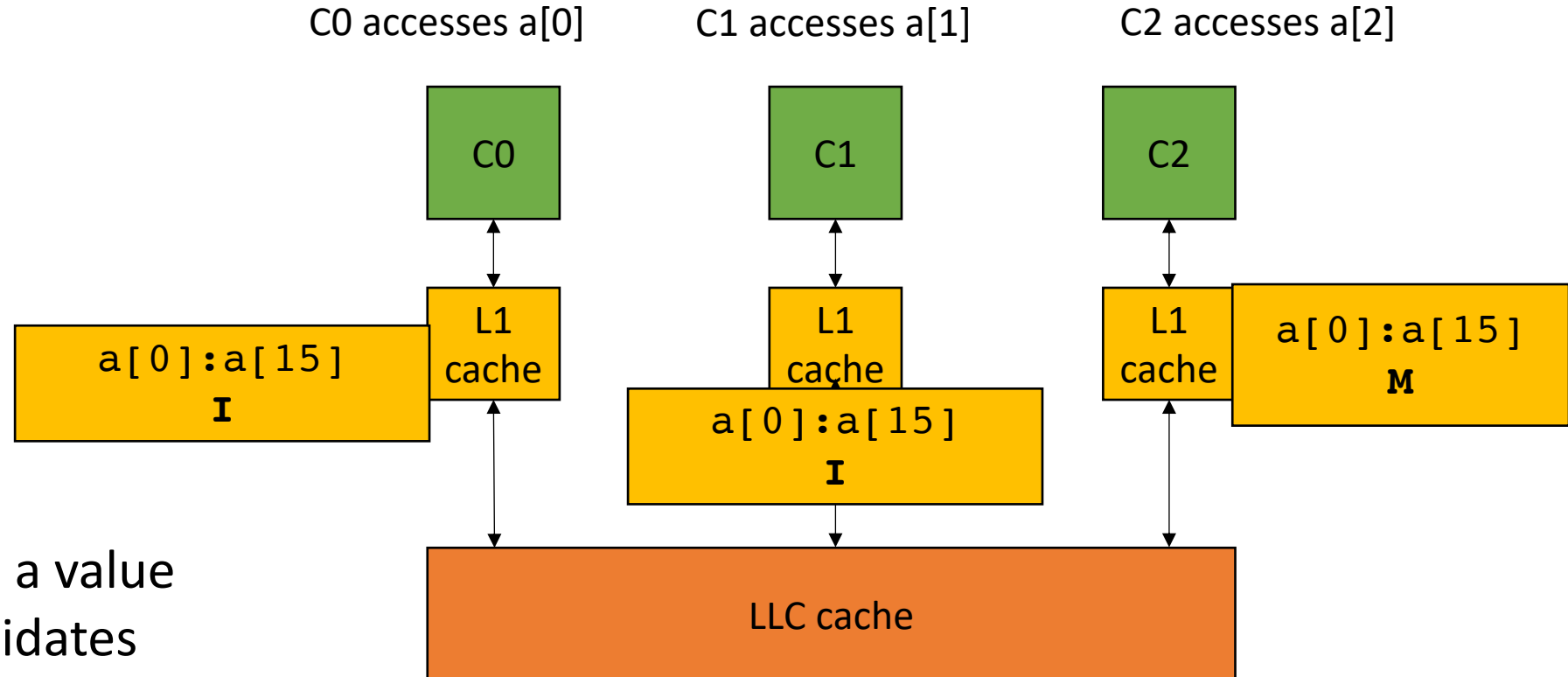
```
for (int i = 0; i < NUM_ELEMENTS; i++) {  
    thread(repeat_increment, a+i);  
}
```

*Don't worry, we will go over C++ thread
in more detail on Thursday*

Example

- Run example

What's going on?



when one core modifies a value in the cache line, it invalidates everyone else's cache line.

This is called ***False Sharing***

Fix?

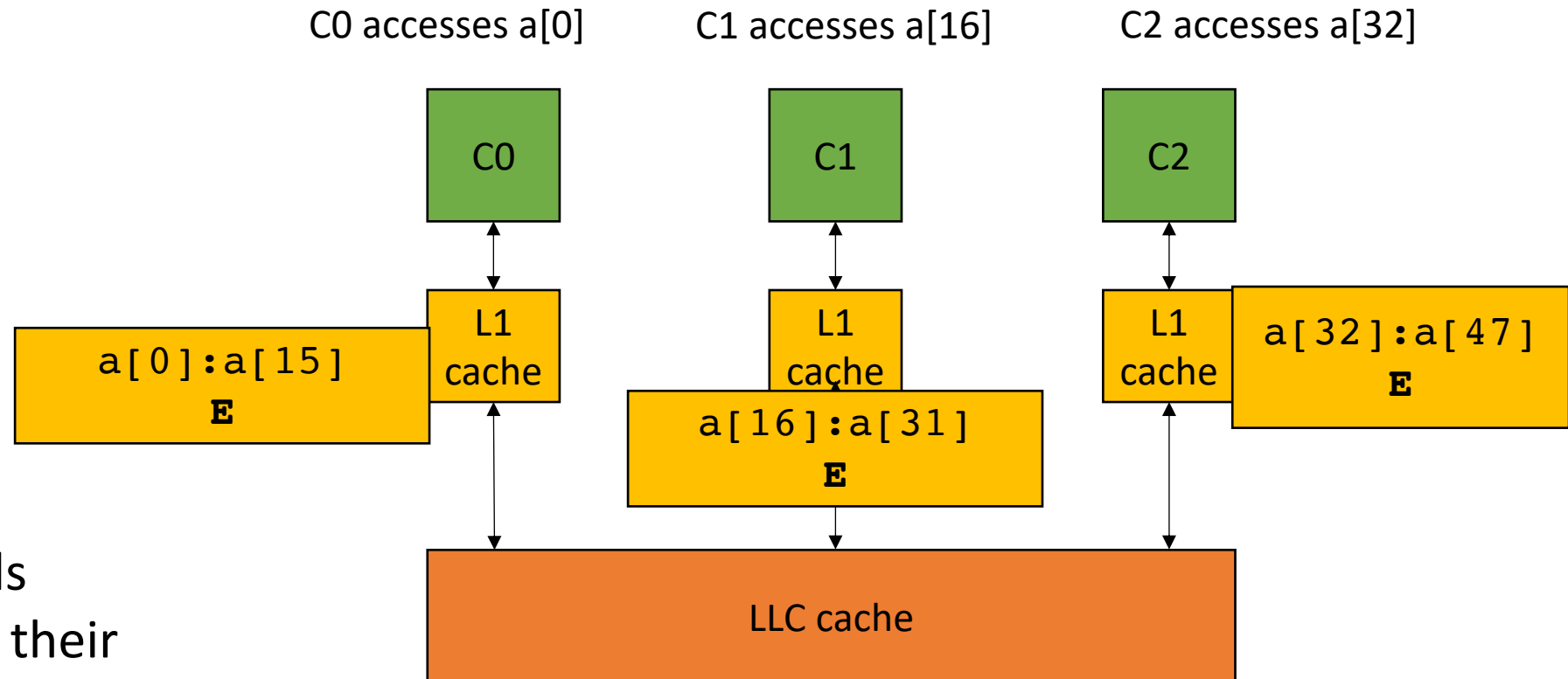
Fix?

- **Padding:** give each element its own cache line:
 - Recall cache line is size 16 ints, so we will use 16x more memory

```
int a[NUM_ELEMENTS * 16];
```

```
for (int i = 0; i < NUM_ELEMENTS; i++) {  
    thread(repeat_increment, a+(i*16));  
}
```

What's going on?



With padding, all threads have exclusive access to their lines! No need to trigger invalidations or write-back each operation

Lecture Schedule

- False Sharing
- **Instruction Level parallelism**
- Loop unrolling

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
- Done by a combination of programmer, compiler, and hardware

Instruction-level Parallelism (ILP)

- What type of instructions can be done in parallel?

Instruction-level Parallelism (ILP)

- What type of instructions can be done in parallel?

two instructions can be executed in parallel if they are independent

Instruction-level Parallelism (ILP)

- 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

(assume all letter variables are registers)

Instruction-level Parallelism (ILP)

- 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

(assume all letter variables are registers)

instructions that are not independent cannot be executed in parallel

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

Instruction-level Parallelism (ILP)

- 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

(assume all letter variables are 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:

How about a more complicated program?

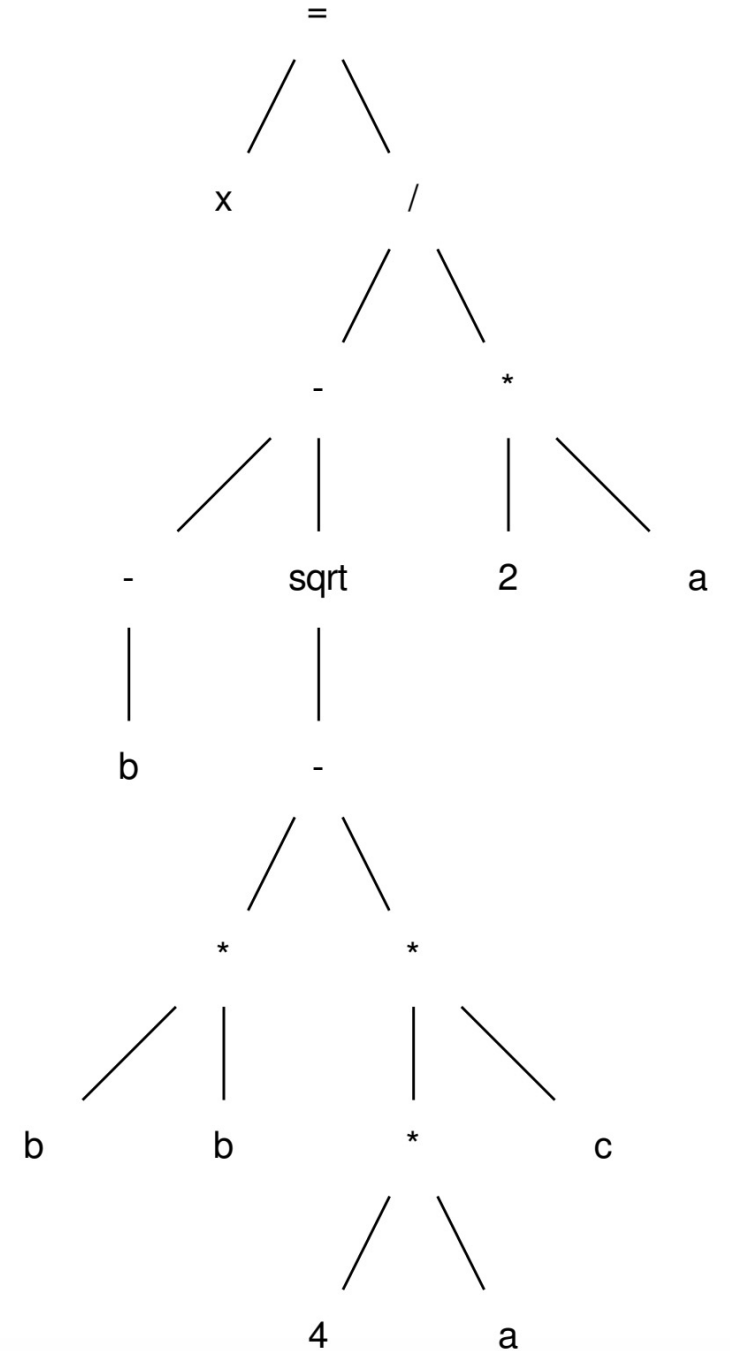
Quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
x = (-b - sqrt(b*b - 4 * a * c)) / (2*a)
```

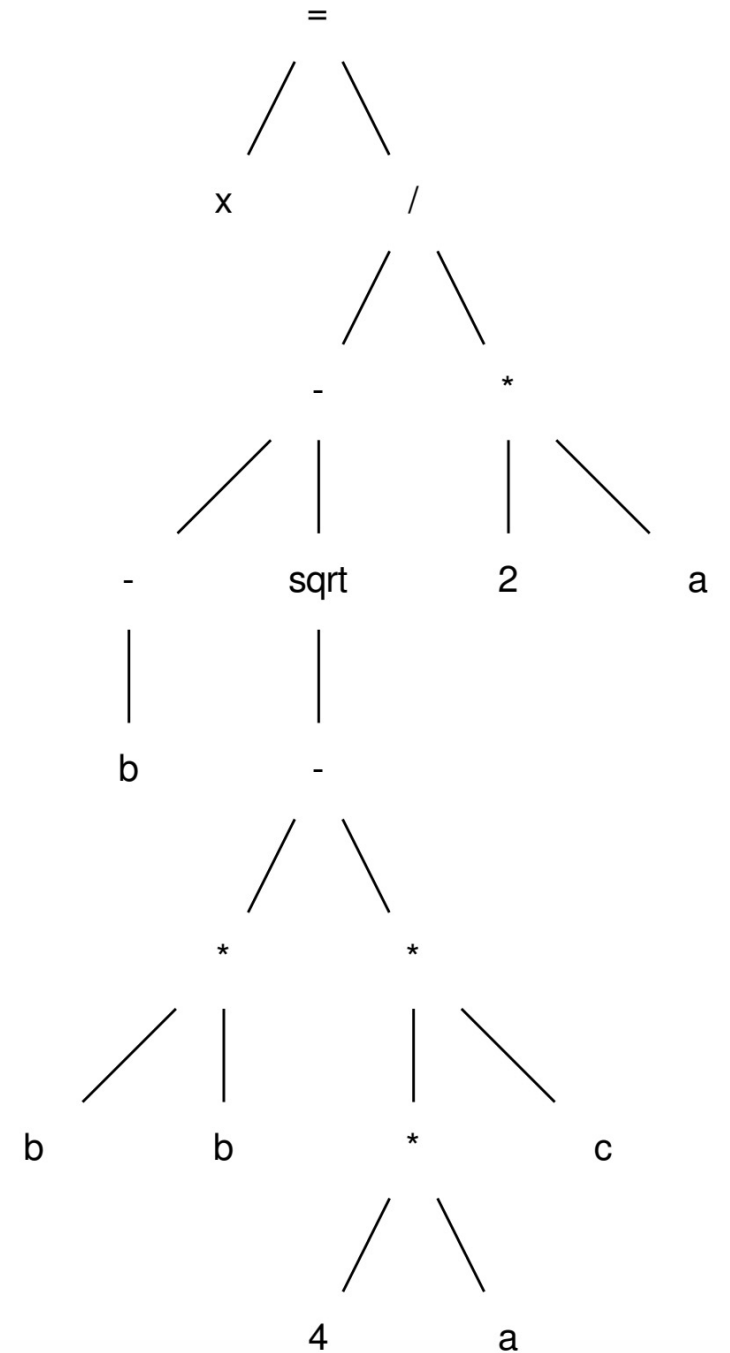
$$x = (-b - \text{sqrt}(b*b - 4 * a * c)) / (2*a)$$

A compiler will turn this into an *abstract syntax tree* (AST)



post-order traversal, using temporary variables

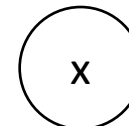
```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;  
r8 = r6 / r7;
```



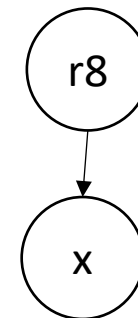
```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;  
r8 = r6 / r7;  
x = r8;
```

Now we build a “data dependency graph” (DDG)

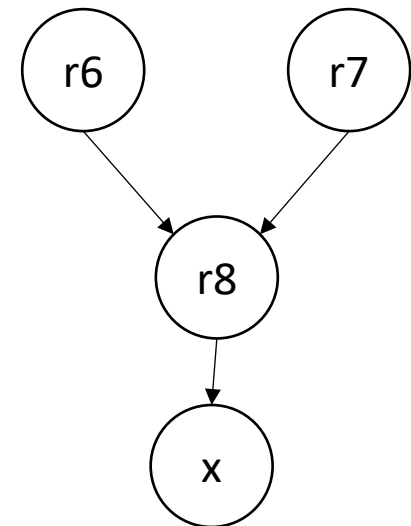
```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;  
r8 = r6 / r7;  
x  = r8;
```



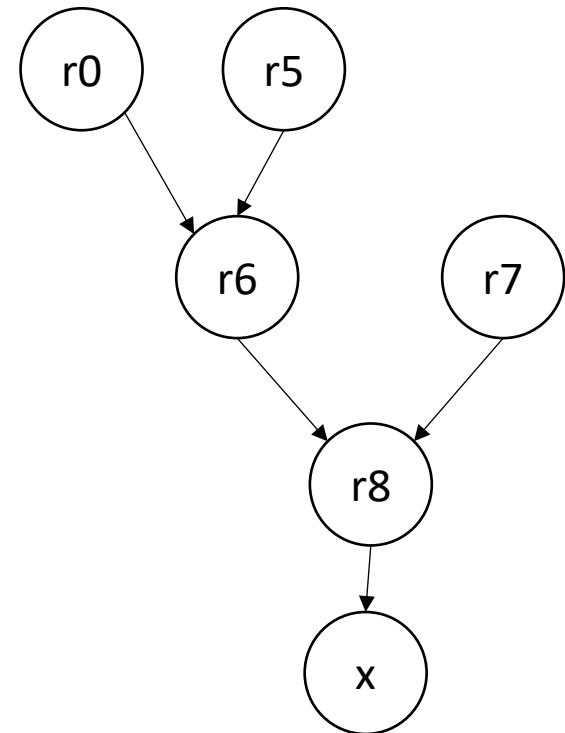
```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;  
r8 = r6 / r7;  
x  = r8;
```



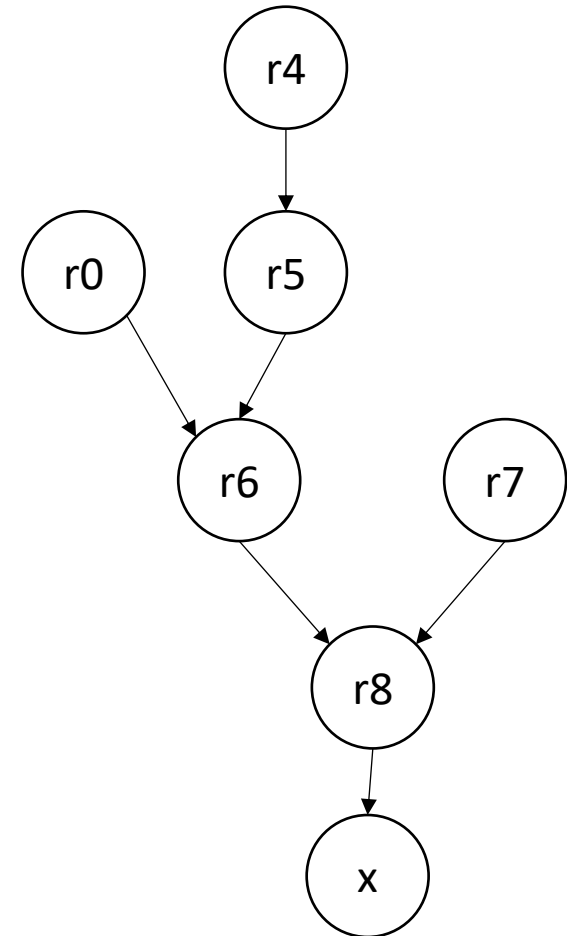
```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;  
r8 = r6 / r7;  
x  = r8;
```



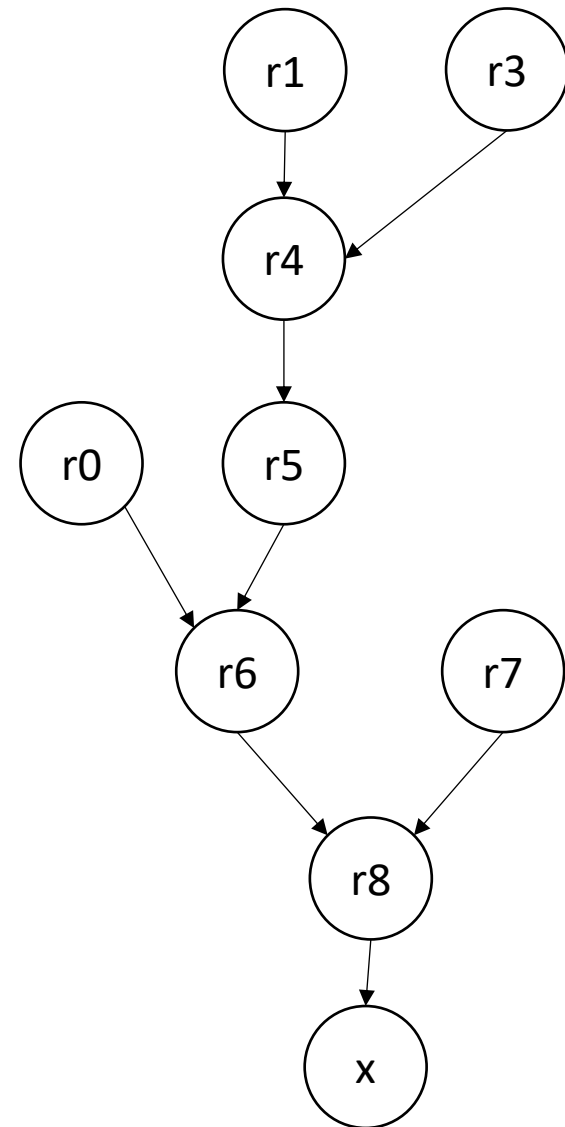
```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;  
r8 = r6 / r7;  
x  = r8;
```



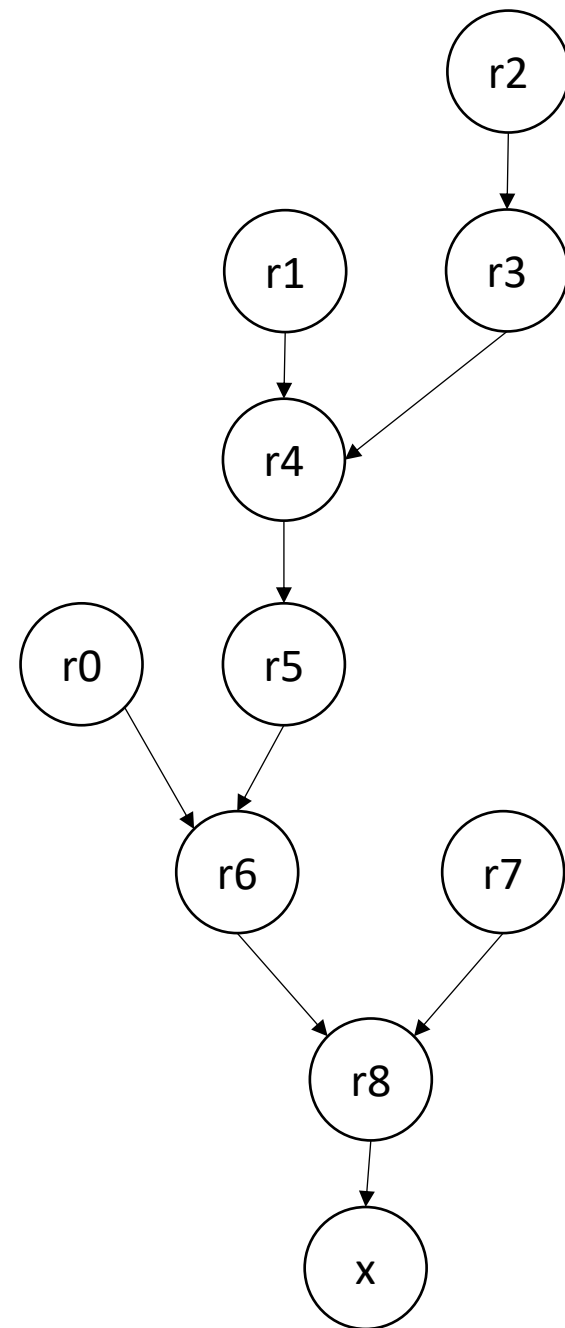
```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;  
r8 = r6 / r7;  
x  = r8;
```



```
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;
```

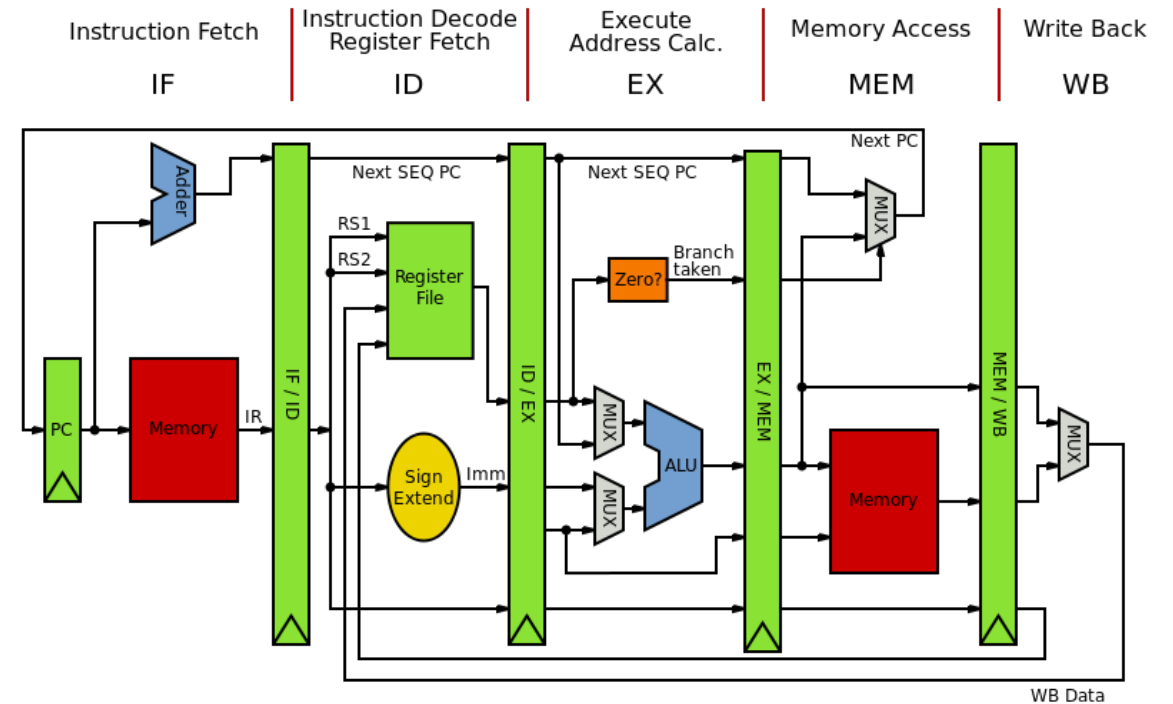



```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;  
r8 = r6 / r7;  
x  = r8;
```



How can hardware execute ILP?

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

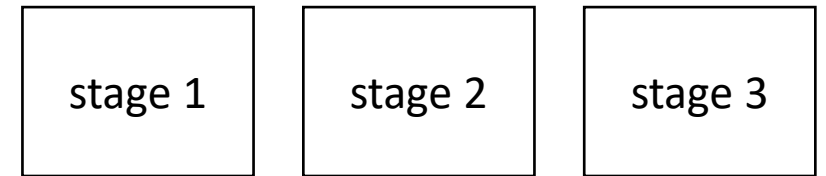


MIPS pipeline image from:
[https://commons.wikimedia.org/wiki/Pipeline_\(computer_hardware\)](https://commons.wikimedia.org/wiki/Pipeline_(computer_hardware))

Pipeline

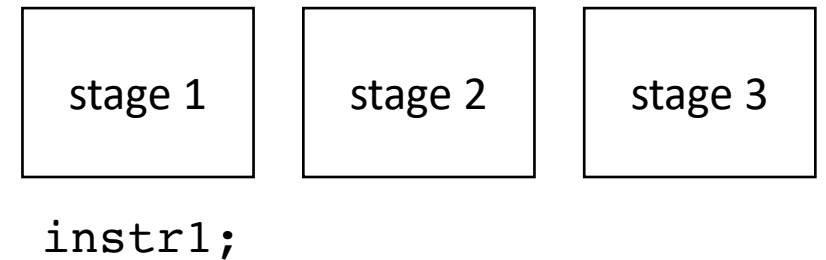
- Pipeline parallelism
- Abstract mental model for compiler:
 - N-stage pipeline
 - N instructions can be in-flight
 - Dependencies stall pipeline

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



Pipeline

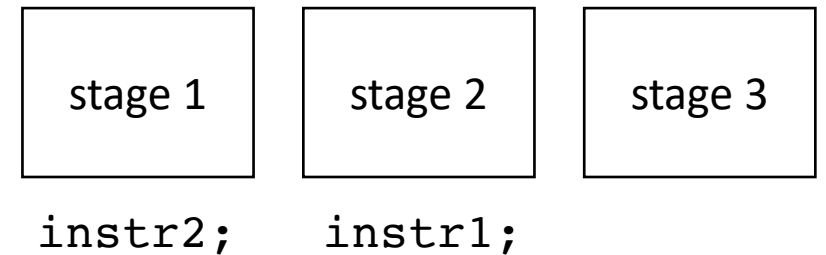
- Pipeline parallelism
- Abstract mental model for compiler:
 - N-stage pipeline
 - N instructions can be in-flight
 - Dependencies stall pipeline



`instr2;`
`instr3;`

Pipeline

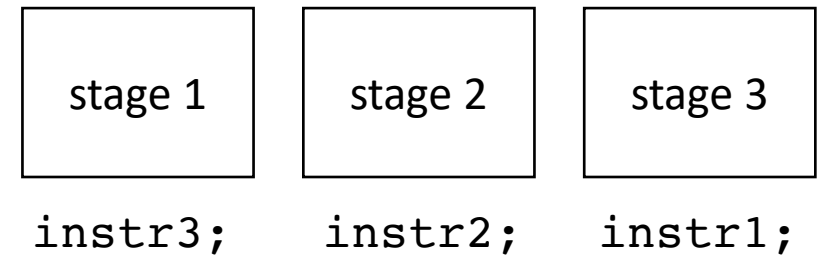
- Pipeline parallelism
- Abstract mental model for compiler:
 - N-stage pipeline
 - N instructions can be in-flight
 - Dependencies stall pipeline



`instr3;`

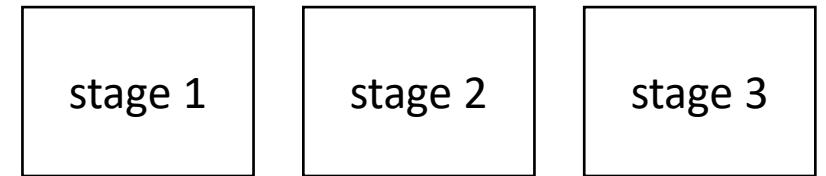
Pipeline

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



Pipeline

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

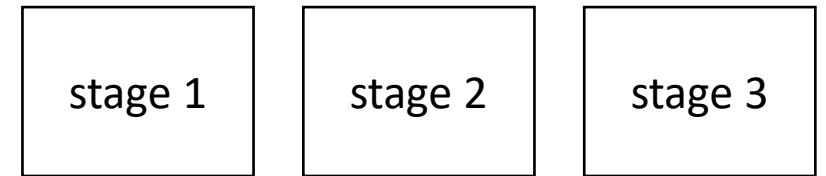


6 cycles for 3 independent instructions

Converges to 1 instruction per cycle

Pipeline

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

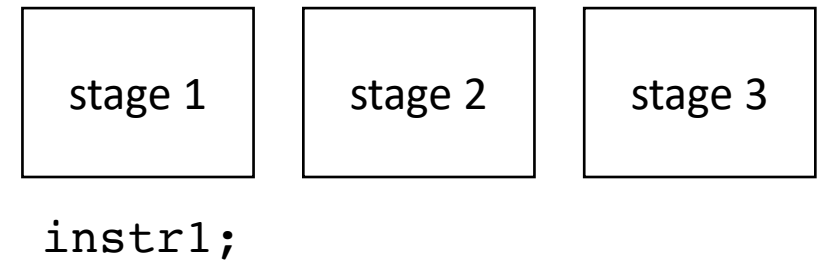


```
instr1;  
instr2;  
instr3;
```

What if the instructions depend on each other?

Pipeline

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

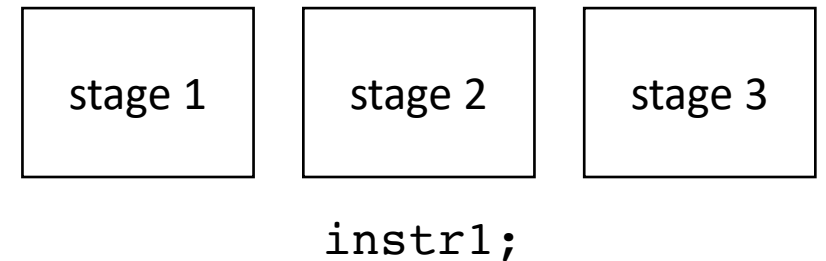


`instr2;`
`instr3;`

What if the instructions depend on each other?

Pipeline

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

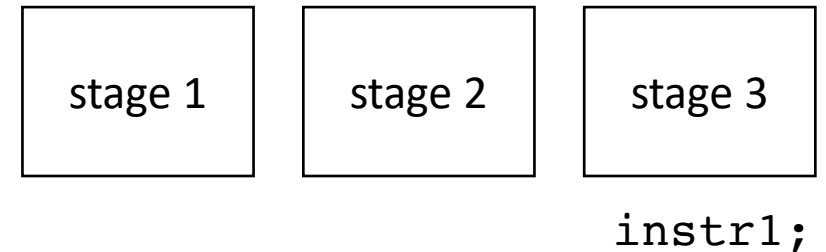


instr2;
instr3;

What if the instructions depend on each other?

Pipeline

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

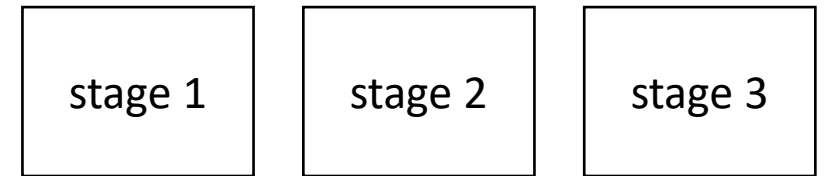


instr2;
instr3;

What if the instructions depend on each other?

Pipeline

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

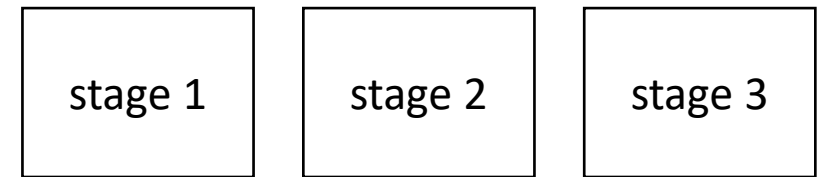


```
instr2;  
instr3;
```

*What if the
instructions depend on
each other?*

Pipeline

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



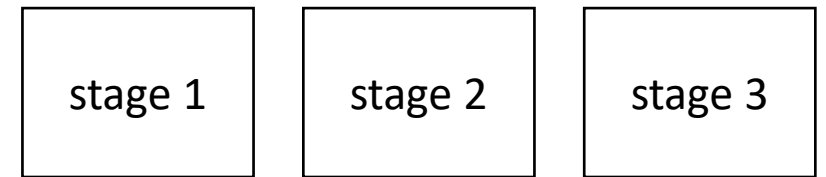
`instr2;`

`instr3;`

What if the instructions depend on each other?

Pipeline

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



`instr2;`

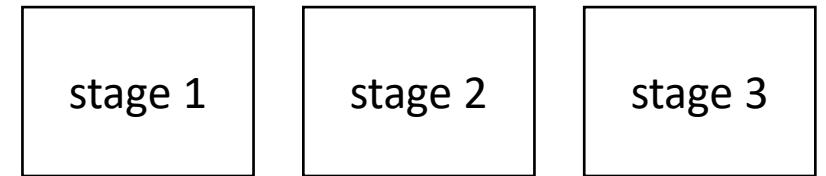
`instr3;`

and so on...

What if the instructions depend on each other?

Pipeline

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



What if the instructions depend on each other?

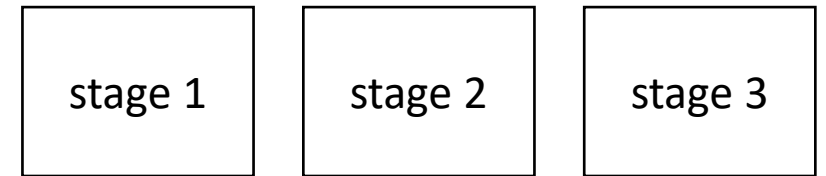
9 cycles for 3 instructions

converges to 3 cycles per instruction

Pipeline

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

```
instr1;  
instrX0;  
instrX1;  
instr2;  
instrX2;  
instrX3;  
instr3;
```

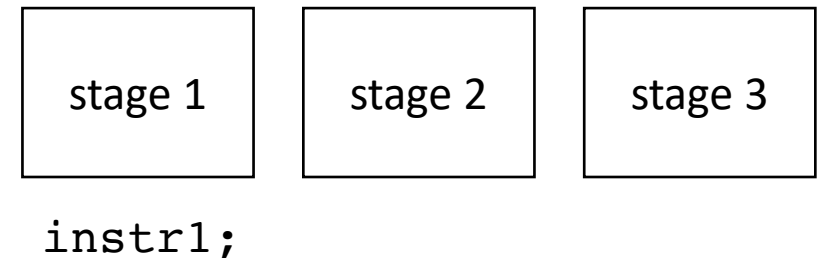


If there are non-dependent instructions from other places in the program that we can interleave then we can get back performance!

Pipeline

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

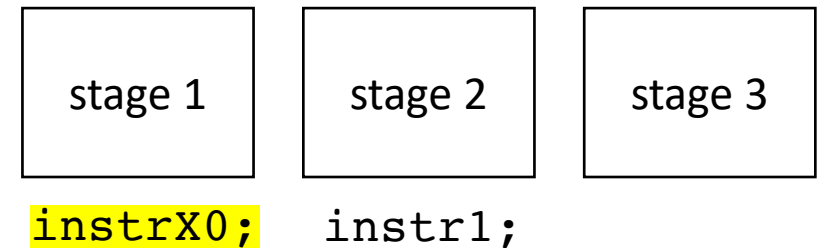
```
instrX0;  
instrX1;  
instr2;  
instrX2;  
instrX3;  
instr3;
```



If there are non-dependent instructions from other places in the program that we can interleave then we can get back performance!

Pipeline

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

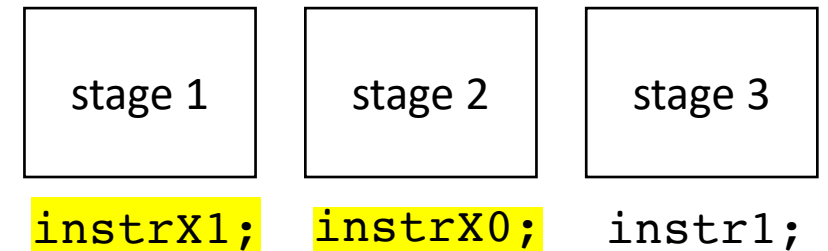


```
instrX1;  
instr2;  
instrX2;  
instrX3;  
instr3;
```

If there are non-dependent instructions from other places in the program that we can interleave then we can get back performance!

Pipeline

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

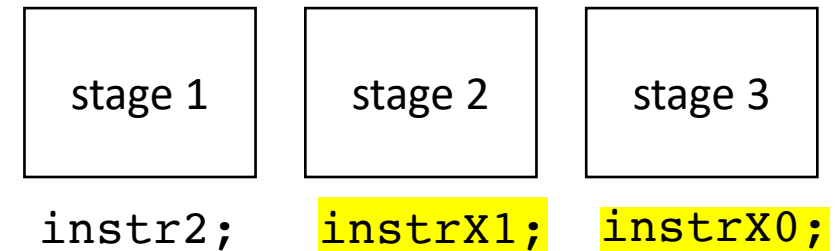


```
instr2;  
instrX2;  
instrX3;  
instr3;
```

If there are non-dependent instructions from other places in the program that we can interleave then we can get back performance!

Pipeline

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

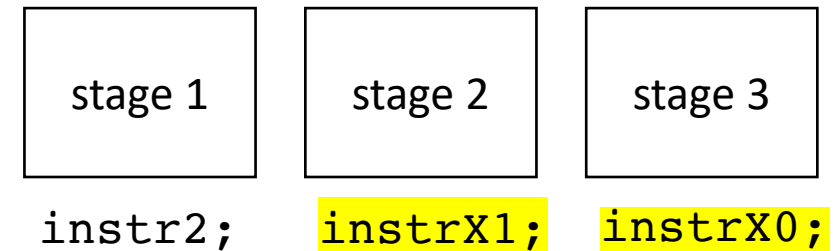


```
instrX2;  
instrX3;  
instr3;
```

If there are non-dependent instructions from other places in the program that we can interleave then we can get back performance!

Pipeline

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



instrX2;
instrX3;
instr3;

and so on...

We converge to 1 cycle per instruction again!

If there are non-dependent instructions from other places in the program that we can interleave then we can get back performance!

How can hardware execute ILP?

- Executing multiple instructions at once:
- Very Long Instruction Word (VLIW) architecture
 - Multiple instructions are combined into one by the compiler
- Superscalar architecture:
 - Several sequential operations are issued in parallel

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

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

It's even more complicated

- Out-of-order execution delays dependent instructions
 - Reorder buffers (RoB) track dependencies
 - Load-Store Queues (LSQ) hold outstanding memory requests

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)

What does this mean for us?

- We should have an abstract and parametrized performance model for instruction scheduling (the order of instructions)
- Try not to place dependent instructions in sequence
- Many times the compiler will help us here, but sometimes it cannot!

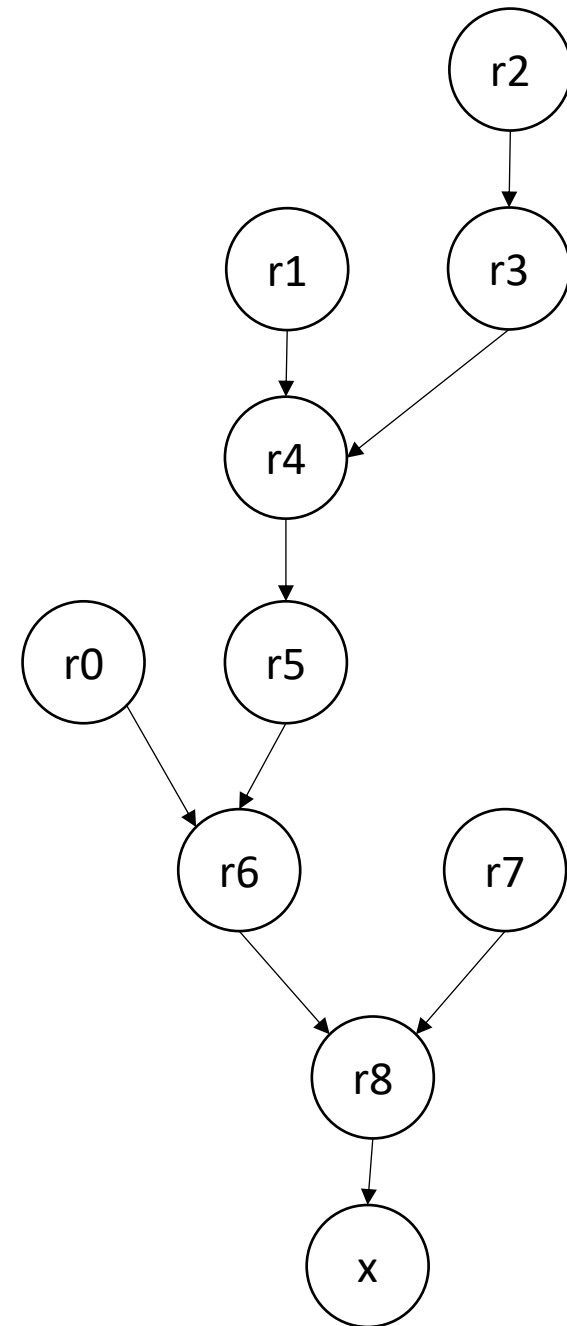
Three techniques to optimize for ILP

- Priority topological ordering
- 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;
```

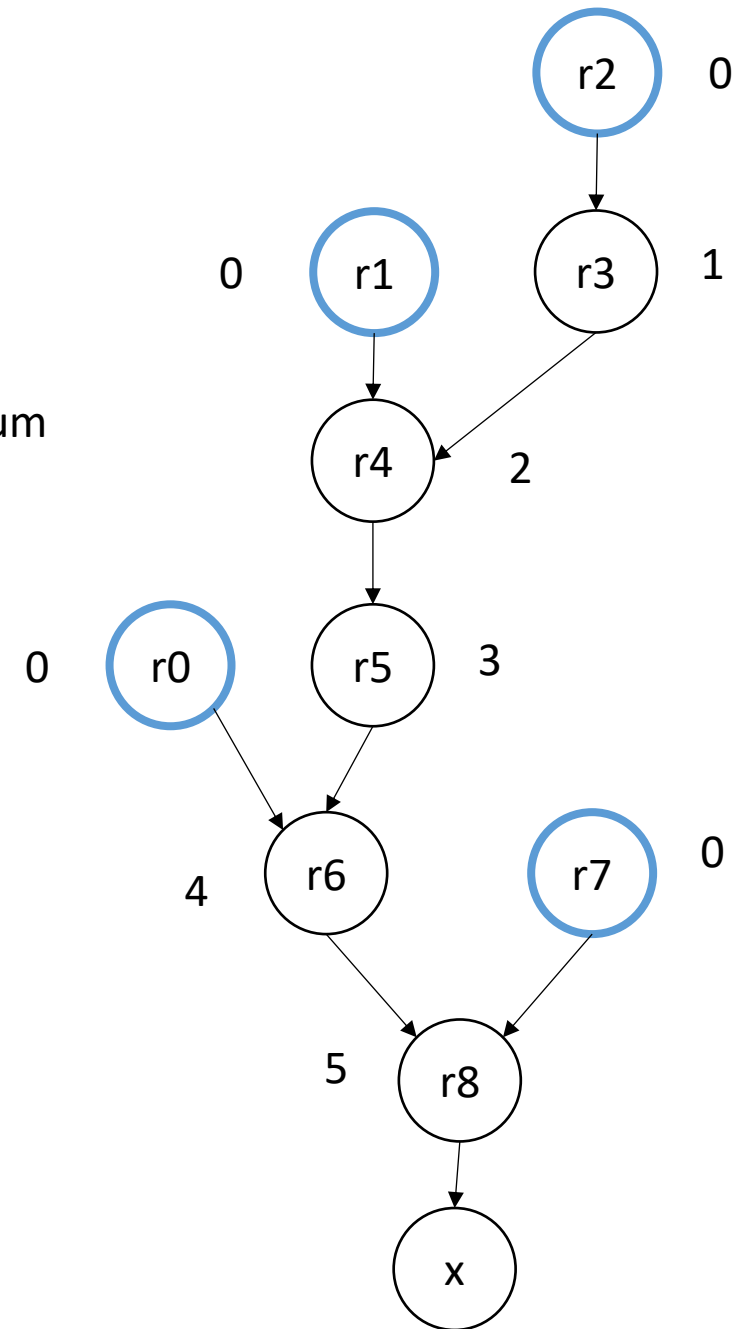
First, consider optimizing for superscalar



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

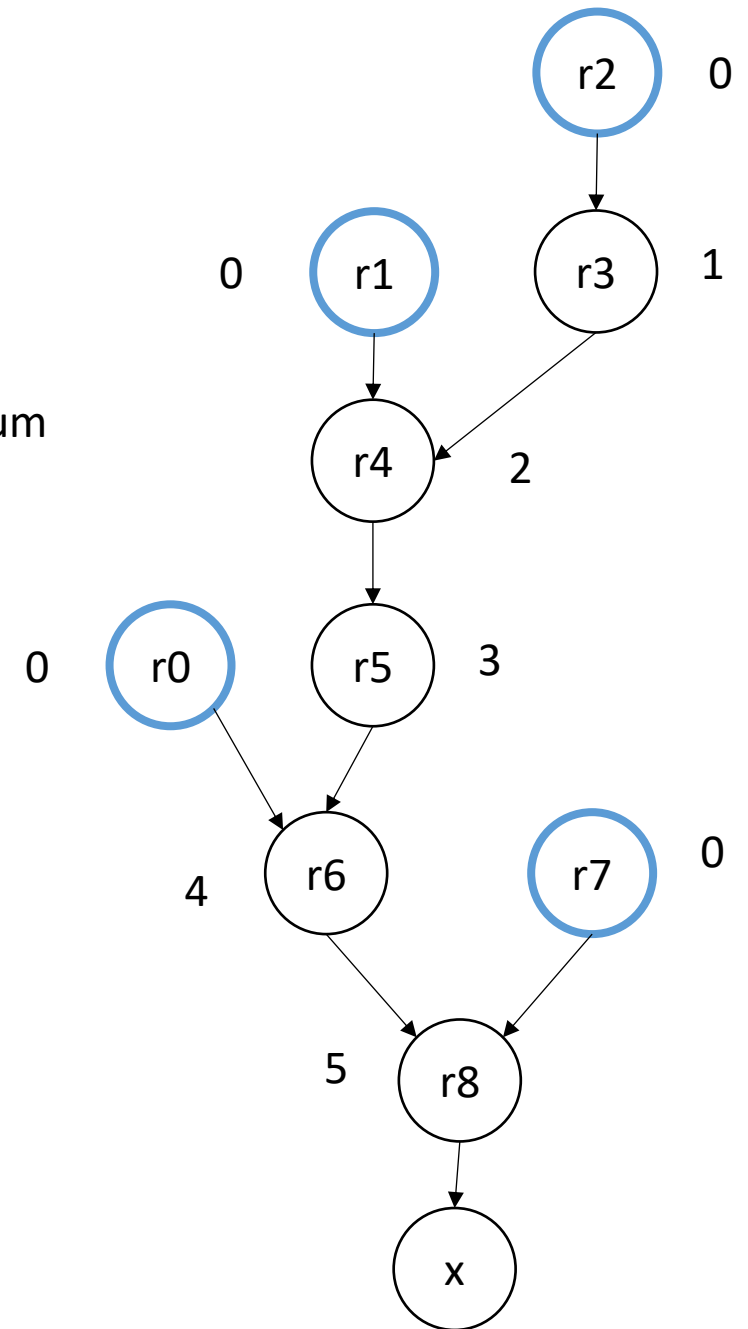


Priority Topological Ordering of DDGs for Superscalar

```
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

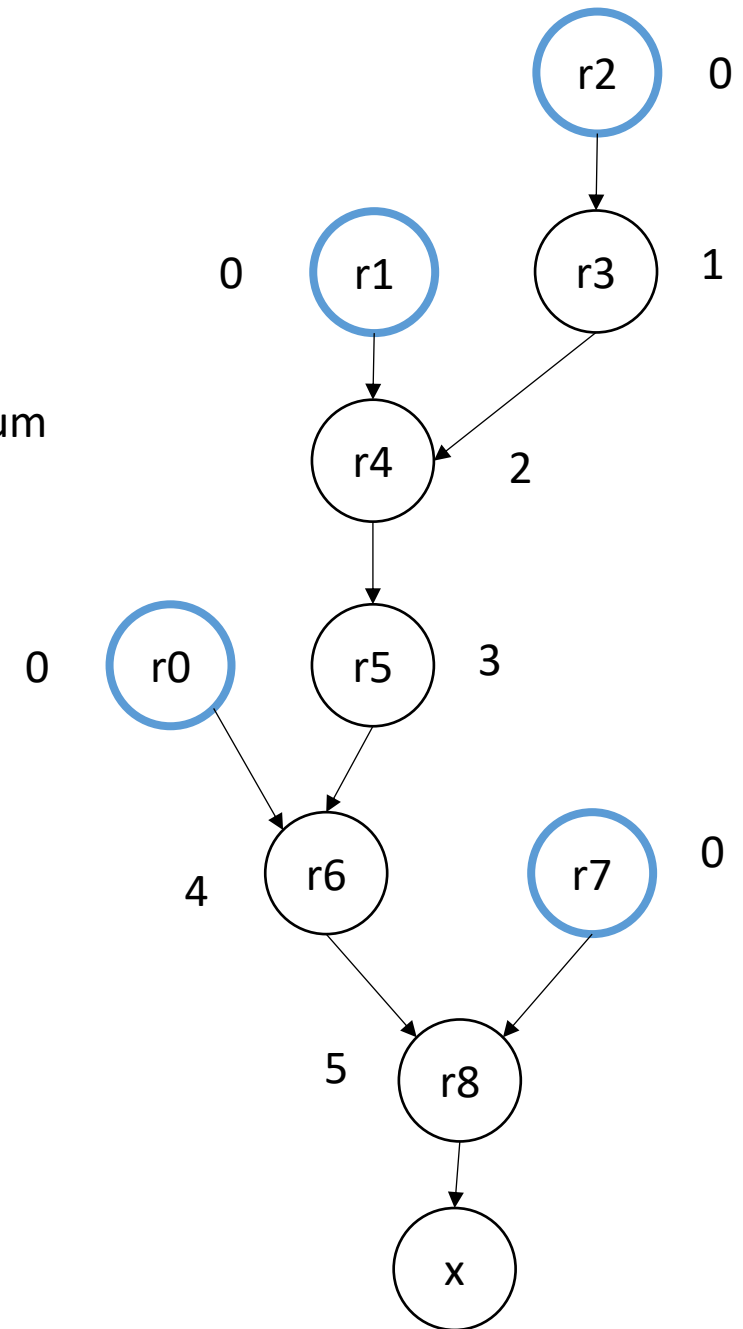


Priority Topological Ordering of DDGs for Superscalar

```
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

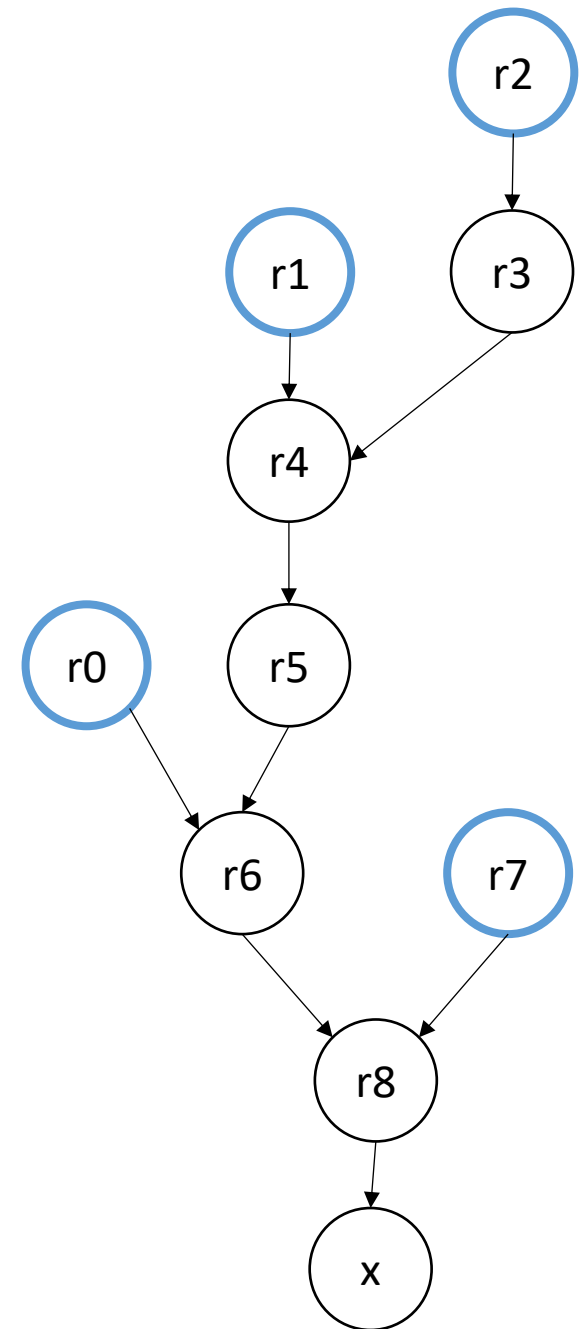
Break ties in topological order using this number



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?

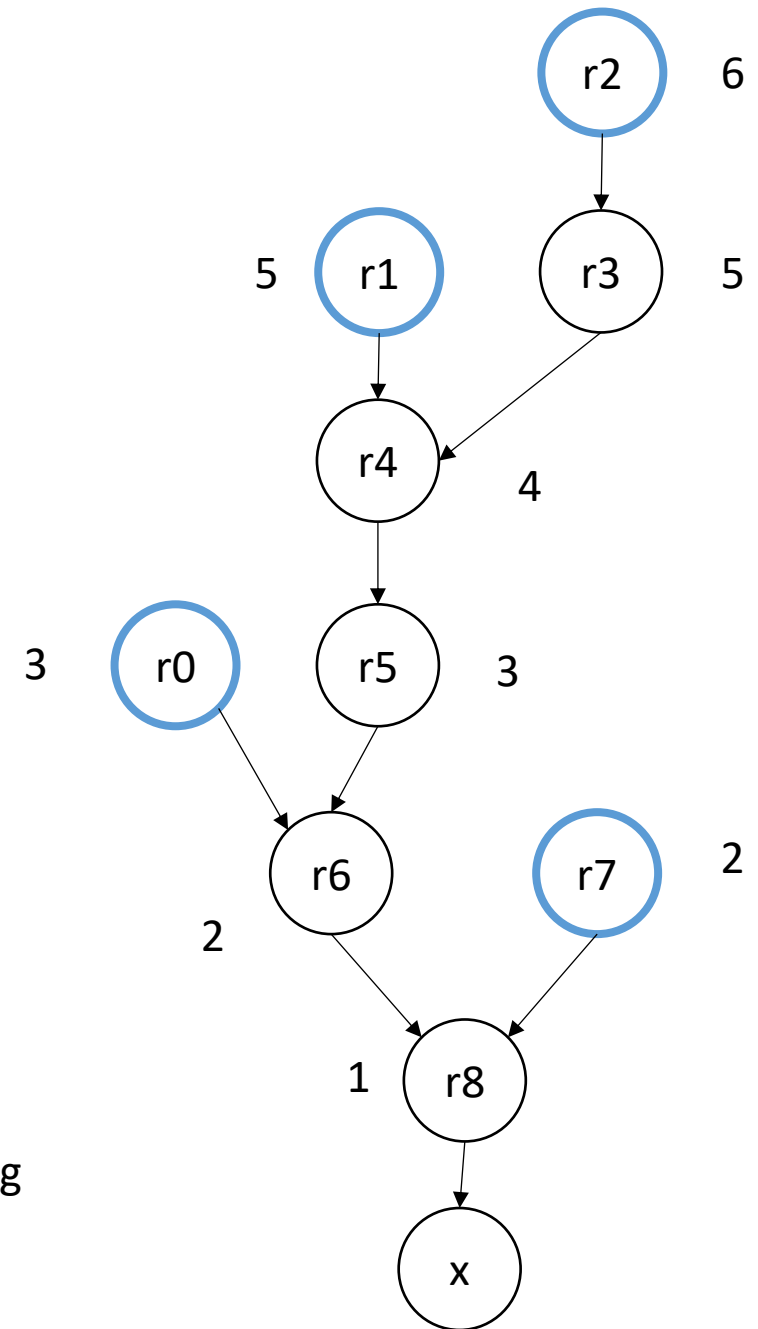


Priority Topological Ordering of DDGs for Pipelining

```
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

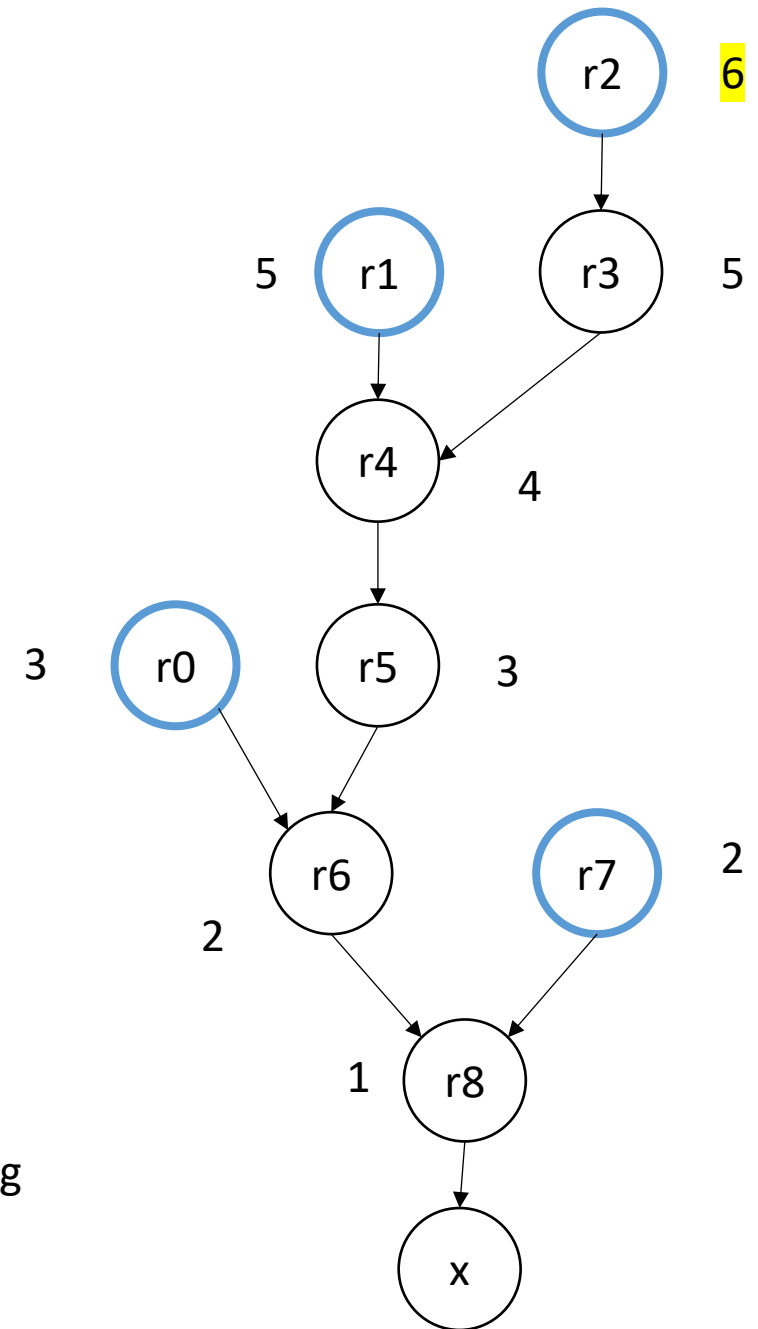


Priority Topological Ordering of DDGs for Pipelining

```
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;
```

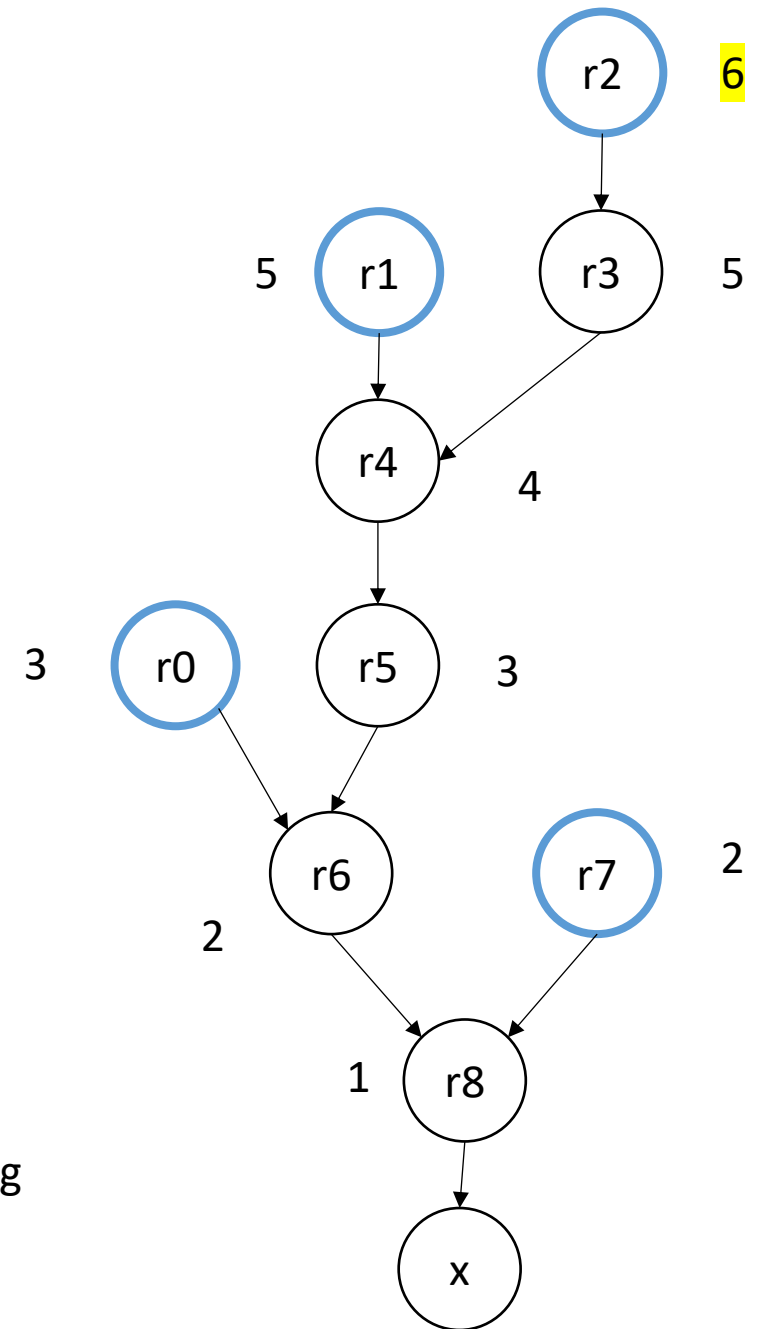
```
r7 = 2 * a;
```

```
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

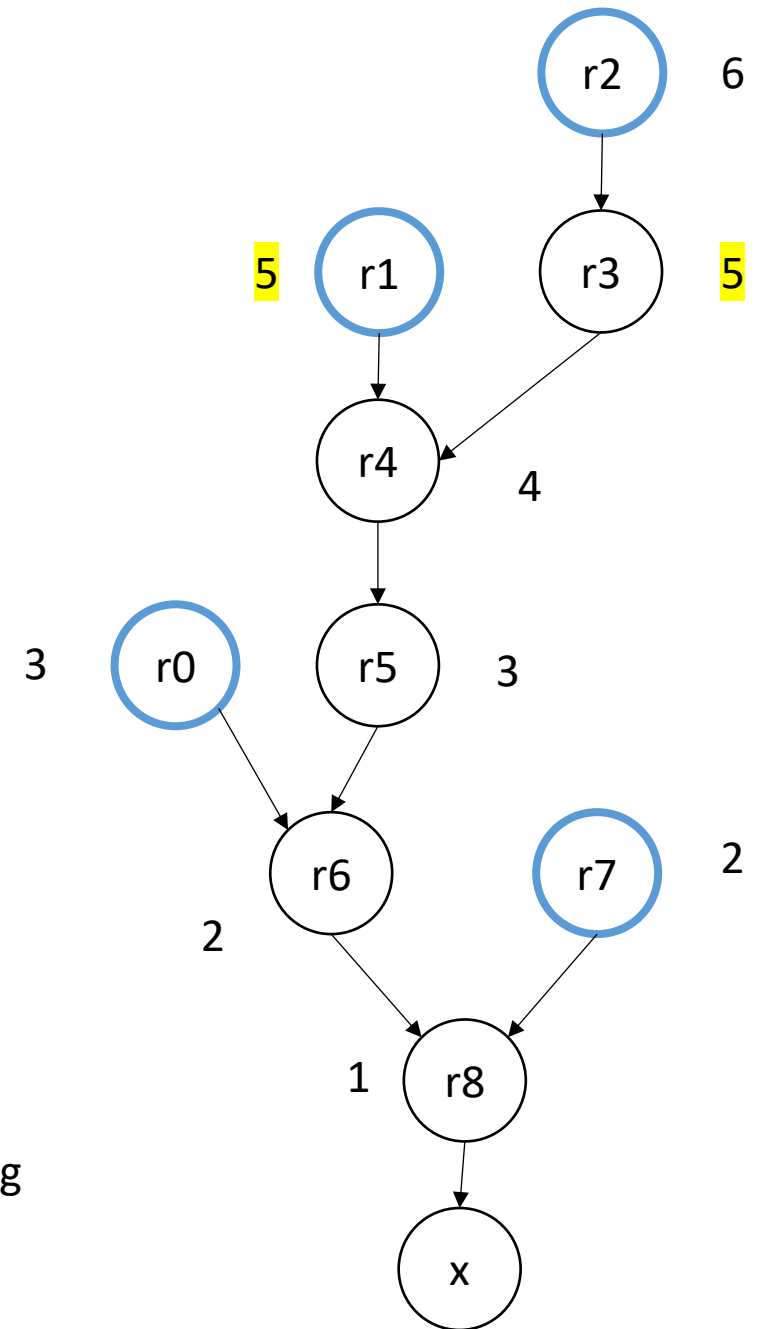


Priority Topological Ordering of DDGs for Pipelining

```
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

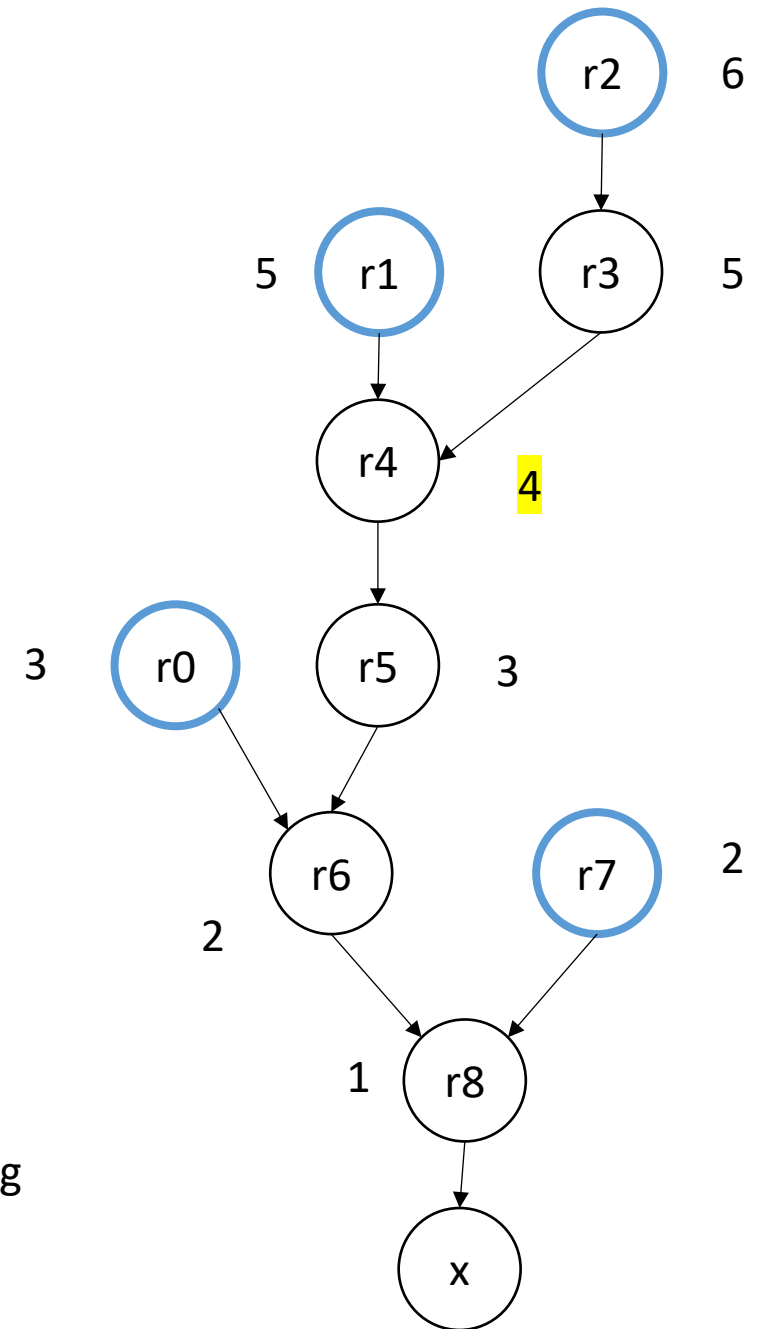


Priority Topological Ordering of DDGs for Pipelining

```
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



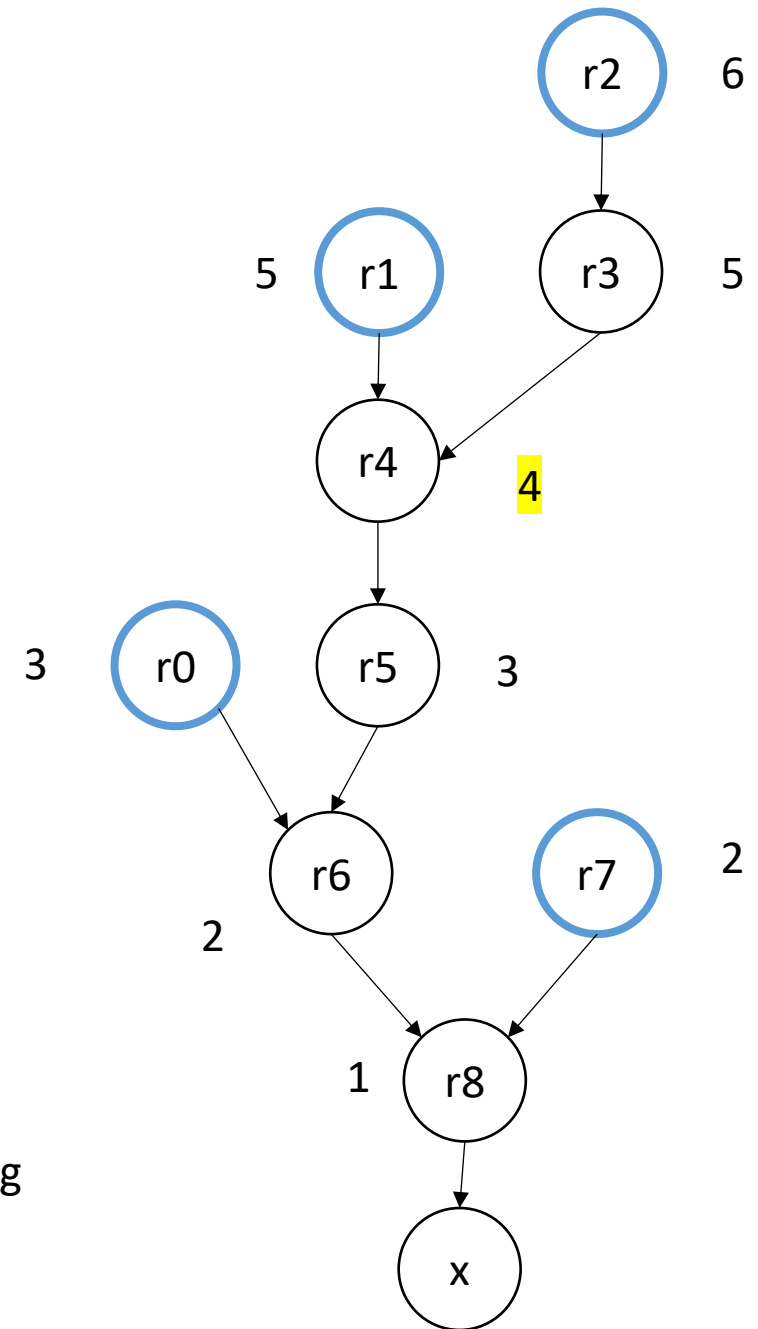
Priority Topological Ordering of DDGs for Pipelining

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

- A compiler will optimize for your architecture using a performance model
- Some approaches use a resource model that explicitly encode the issue-width and pipeline

Use-case

- Loop unrolling
- Reduction loops (we will do on Wednesday)

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);  
}
```

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

Using Loop Unrolling to Exploit ILP

- This is what you are doing in part 1 of homework 1
- You are playing the role of a compiler unrolling loops
- Your “compiler” is written in Python. You print out C++ code
- You the code is parameterized by dependency chain and by unroll factor

Thank you!

- Remember to do the quiz today!
- Get started on homework
 - Setting up docker
 - part 1
- We will discuss ILP for reduction loops and C++ parallelism in the next two lectures