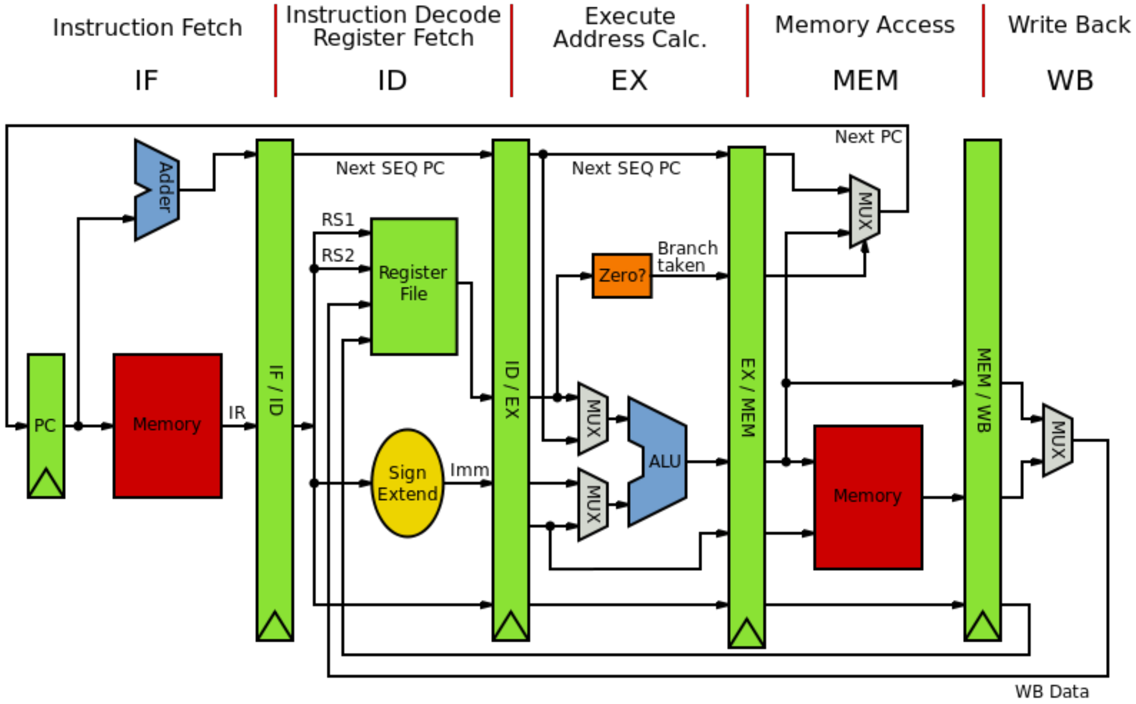
CSE113: Parallel Programming

April 8, 2021

- **Topic**: instruction-level parallelism (ILP)
  - dependency graphs/chains
  - loop unrolling
  - reductions
- and
  - Programming C++ threads



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

Announcements

- Last disruption for the quarter!
- Videos can now be downloaded
  - Do not distribute!
- Homework will be posted by Thursday midnight
  - It is mostly about ILP, so pay attention!
- My office hours are on Friday, 3 - 5 PM

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?

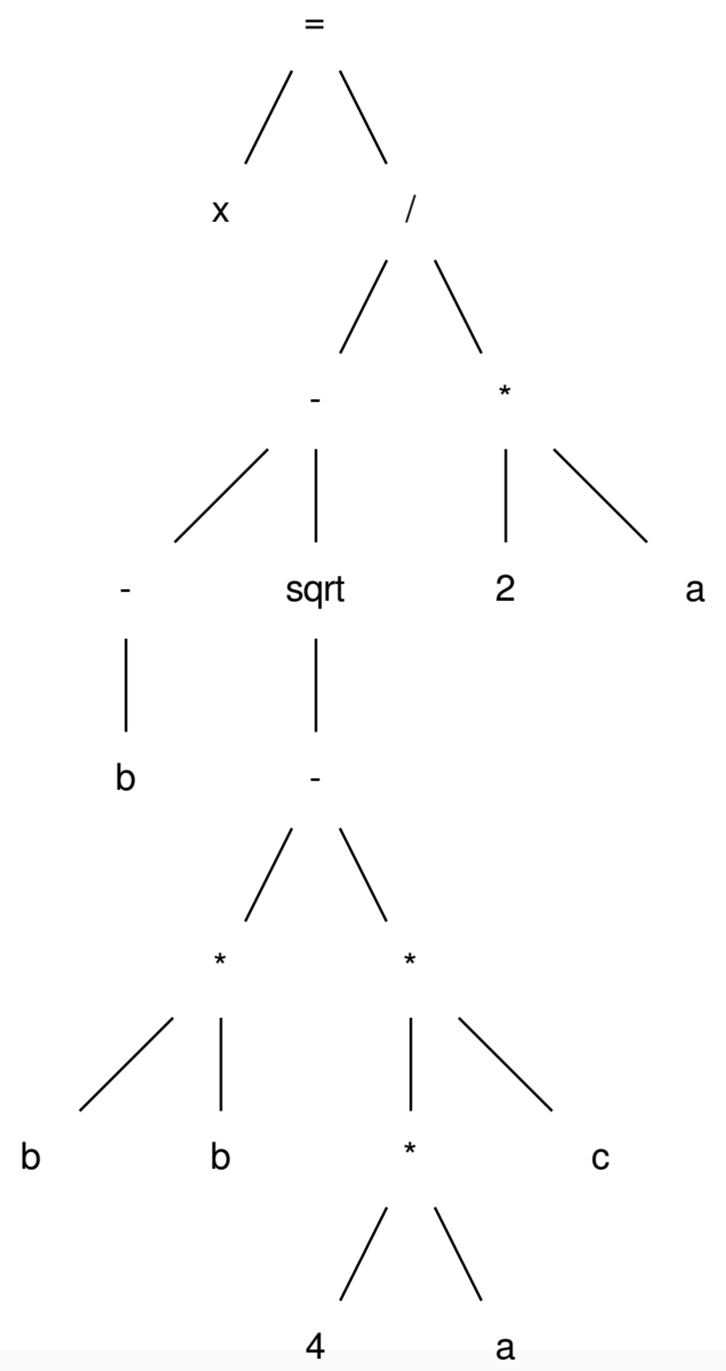
Quadratic formula

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

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

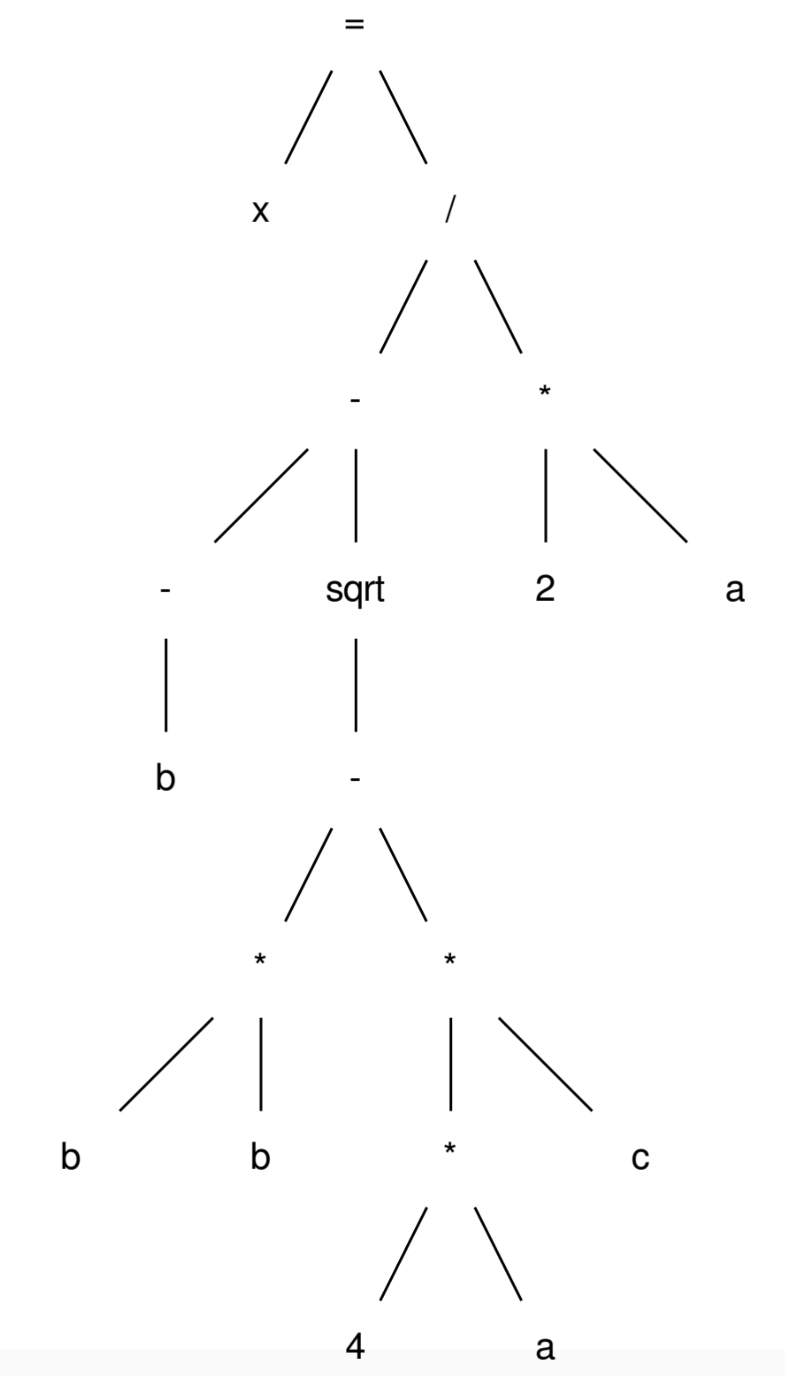
```
x = (-b - sqrt(b*b - 4 * a * c)) / (2*a)
```

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



Simplify this code:

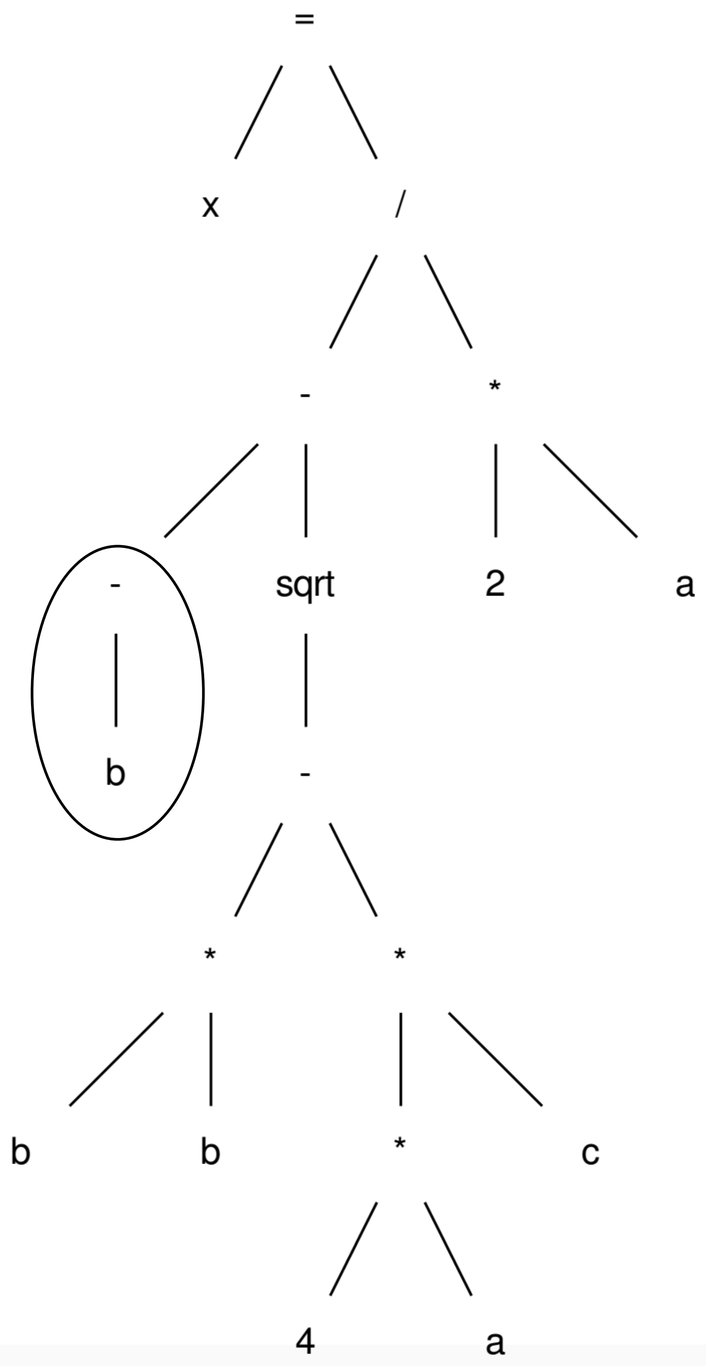
post-order traversal, using temporary variables



Simplify this code:

post-order traversal, using temporary variables

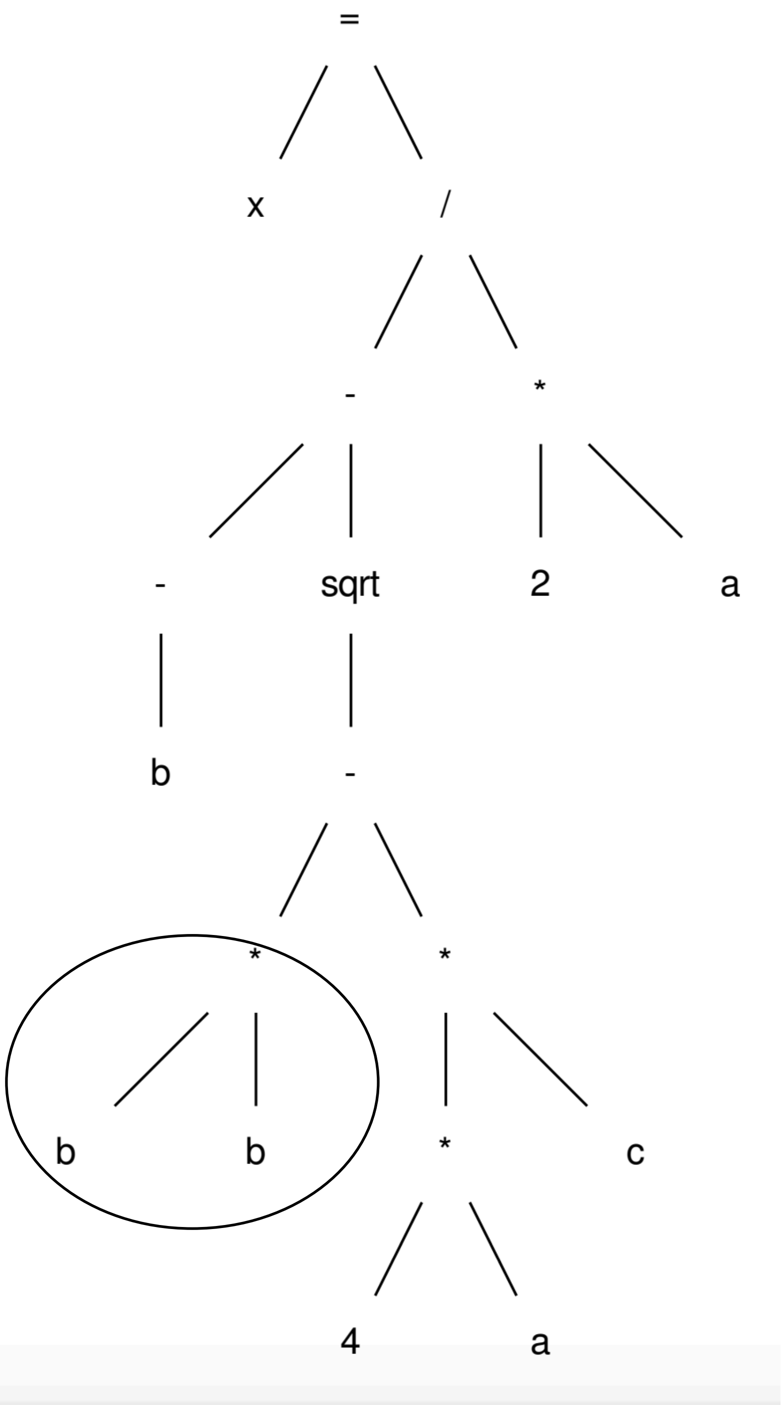
```
r0 = neg(b);
```



Simplify this code:

post-order traversal, using temporary variables

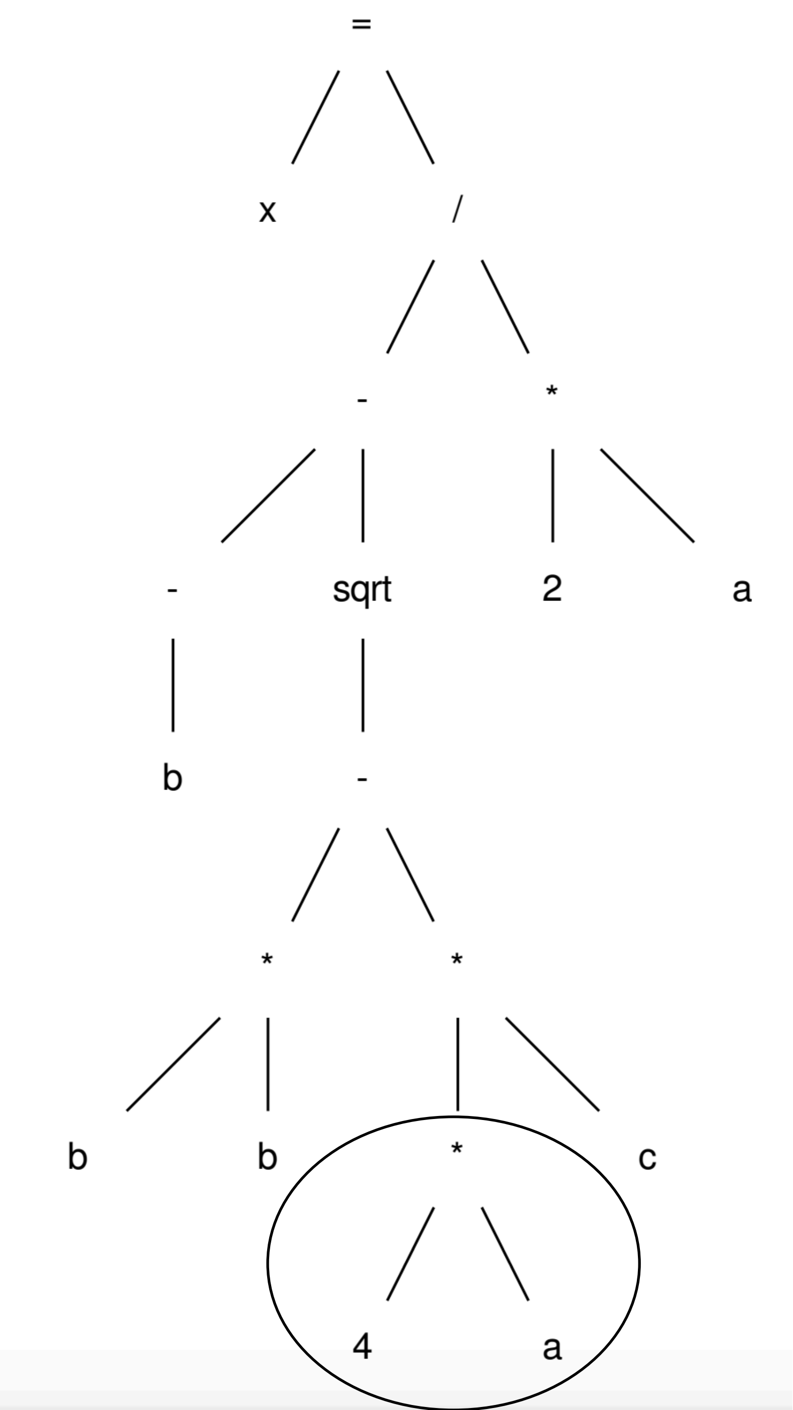
```
r0 = neg(b);
r1 = b * b;
```



Simplify this code:

post-order traversal, using temporary variables

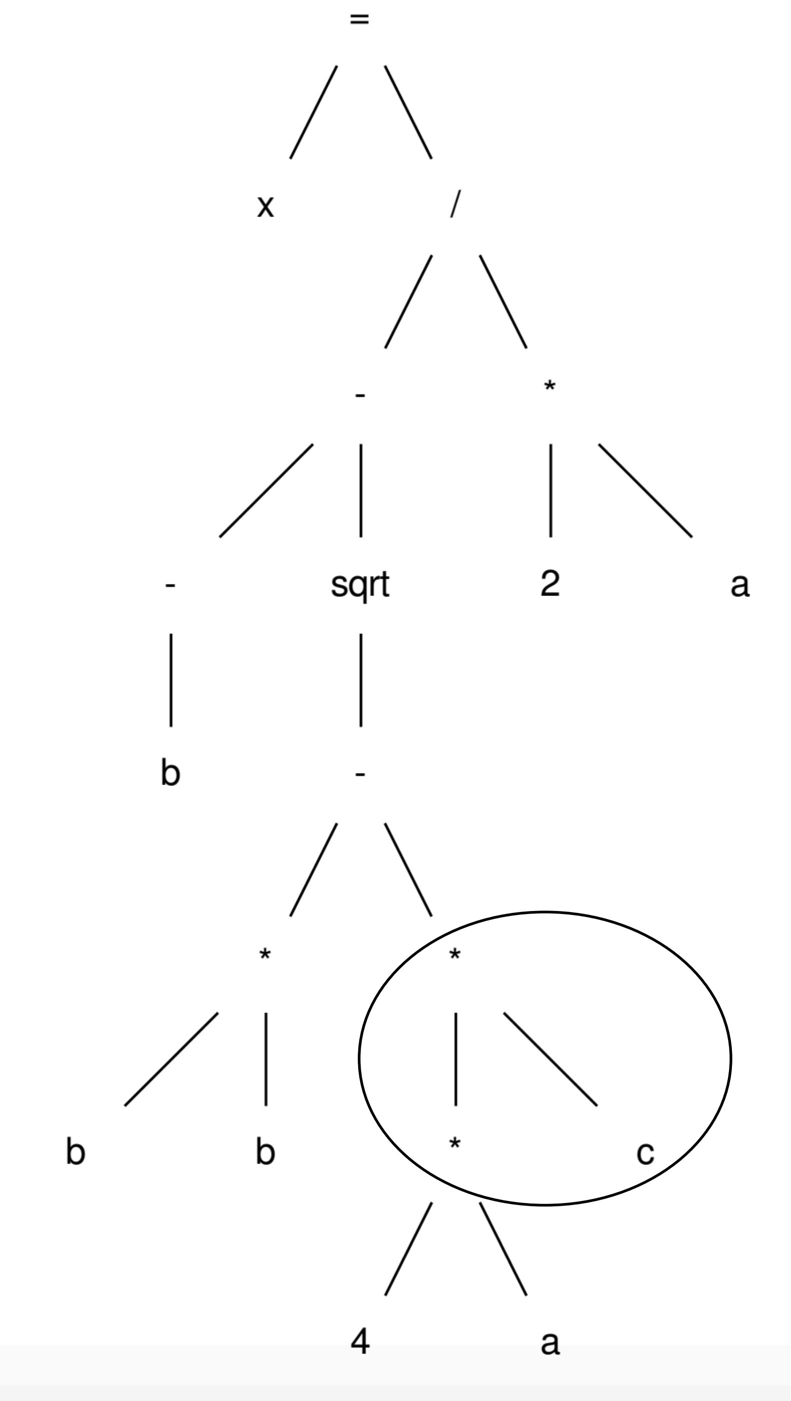
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
```



Simplify this code:

post-order traversal, using temporary variables

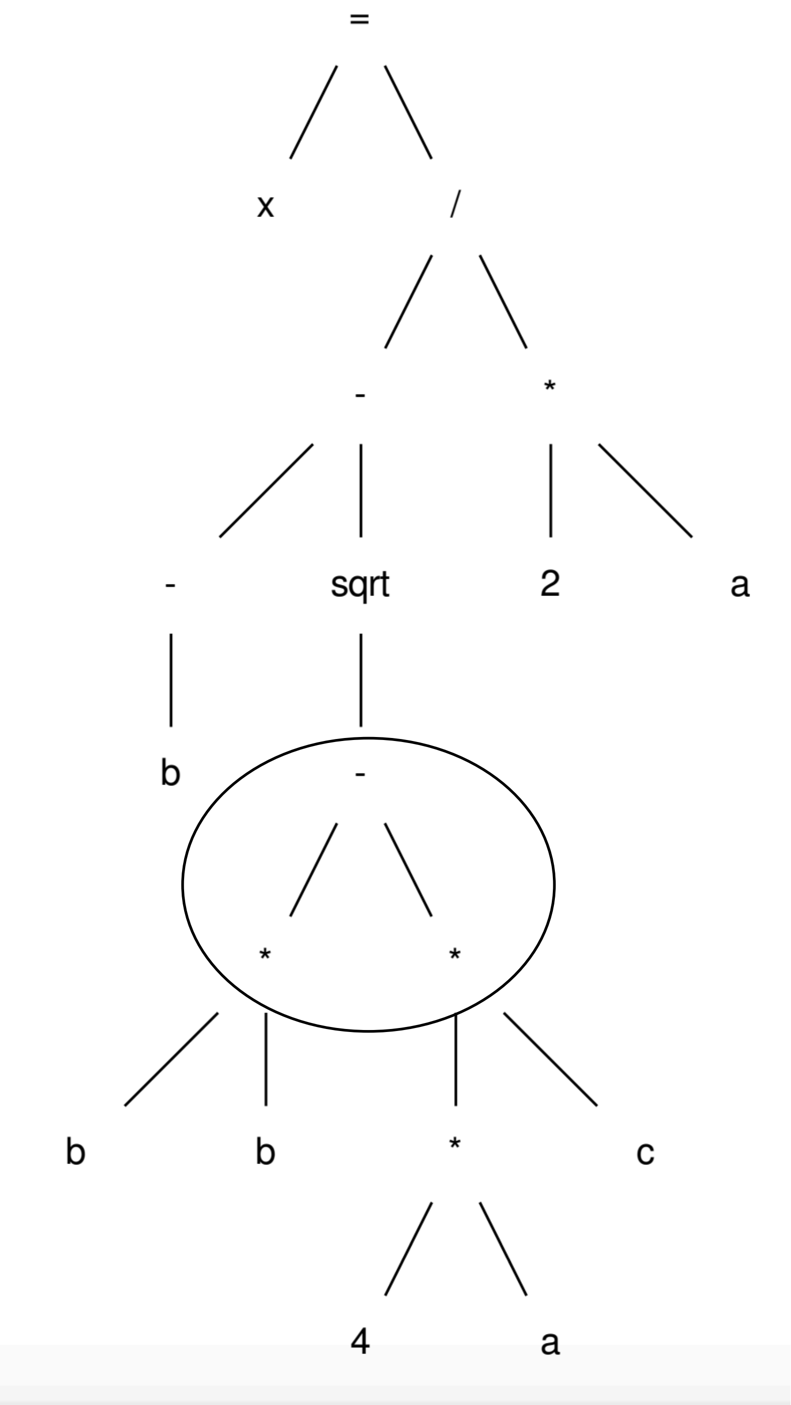
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
```



Simplify this code:

post-order traversal, using temporary variables

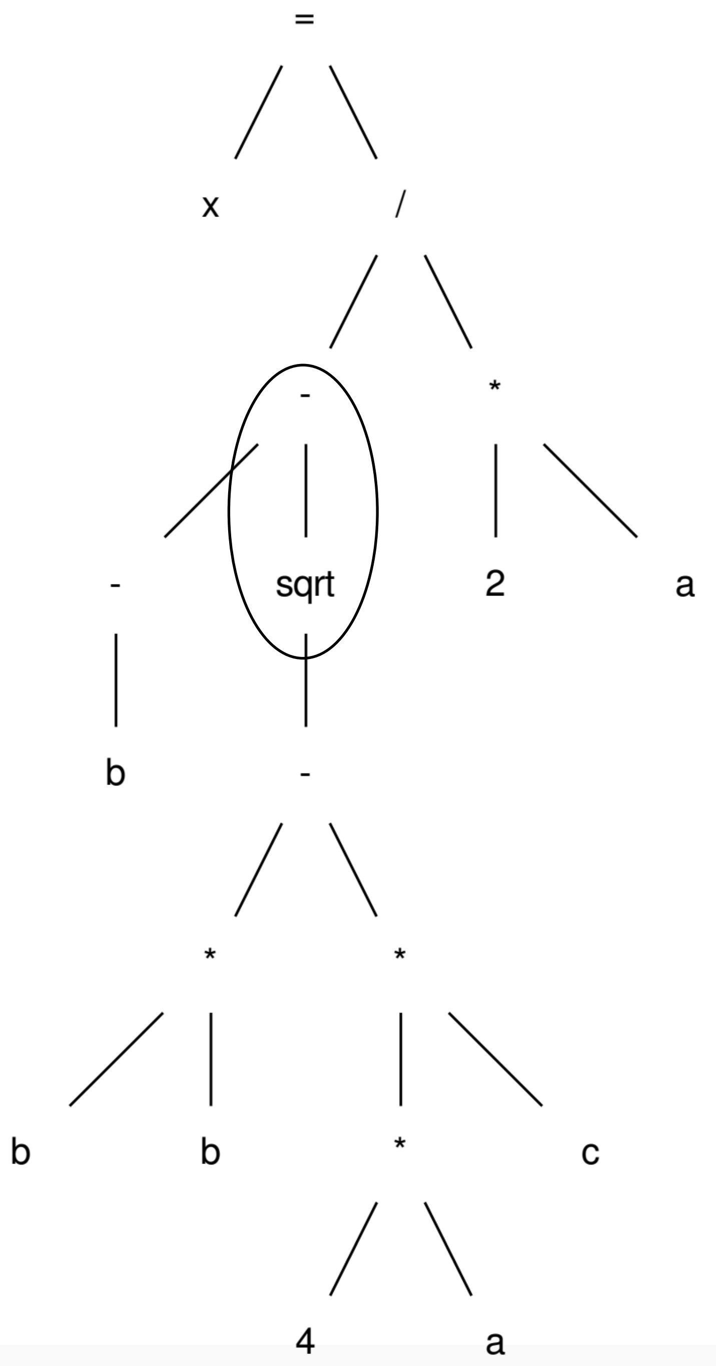
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
```



Simplify this code:

post-order traversal, using temporary variables

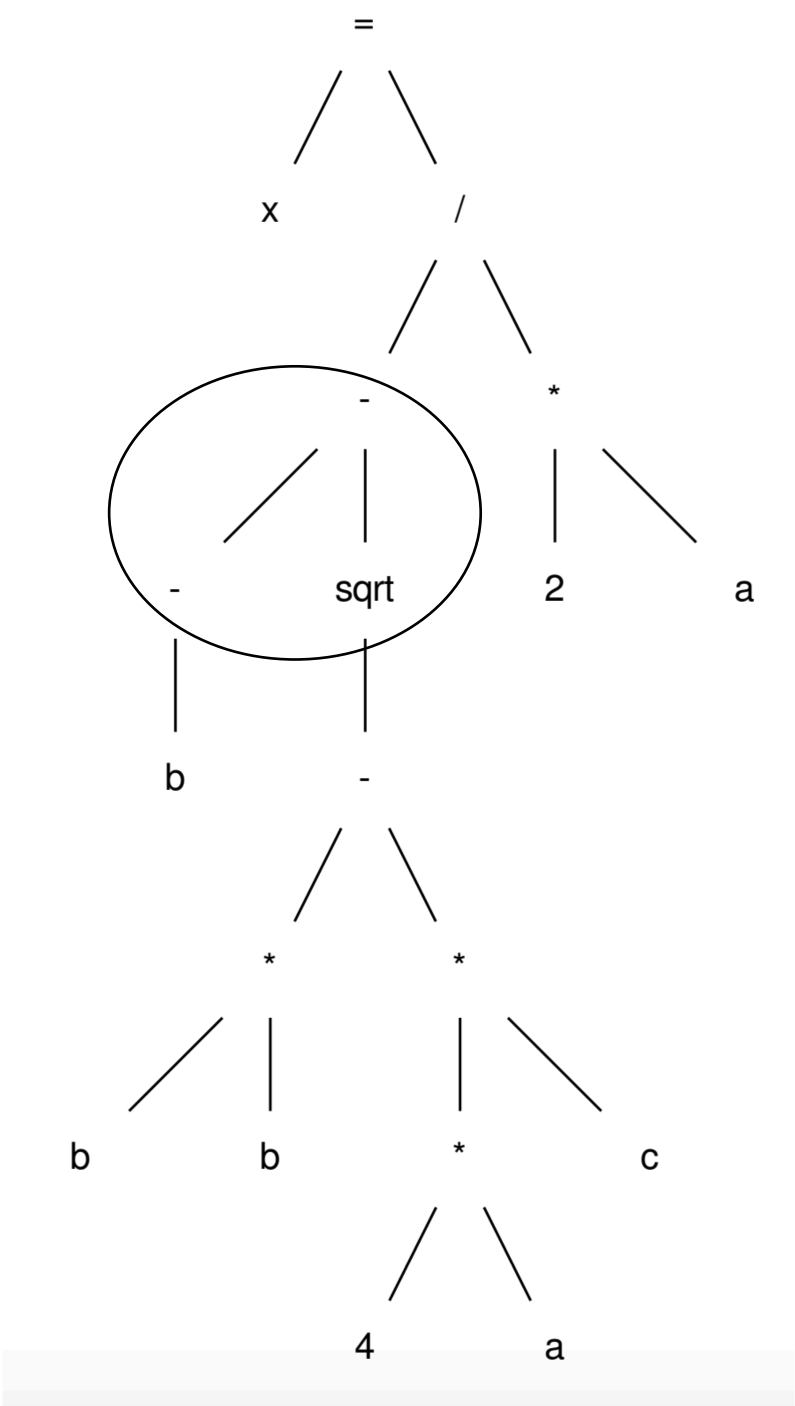
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
```



Simplify this code:

post-order traversal, using temporary variables

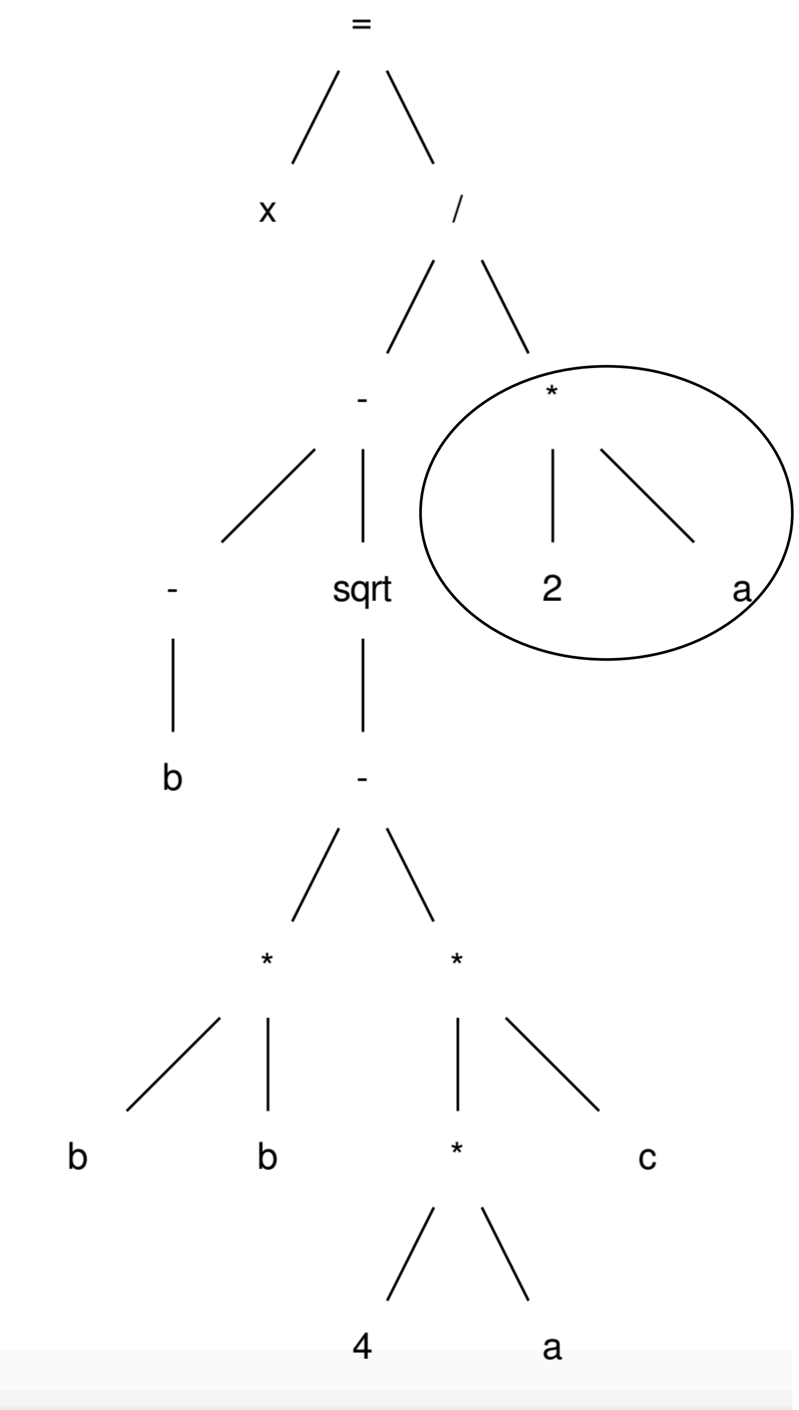
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
```



Simplify this code:

post-order traversal, using temporary variables

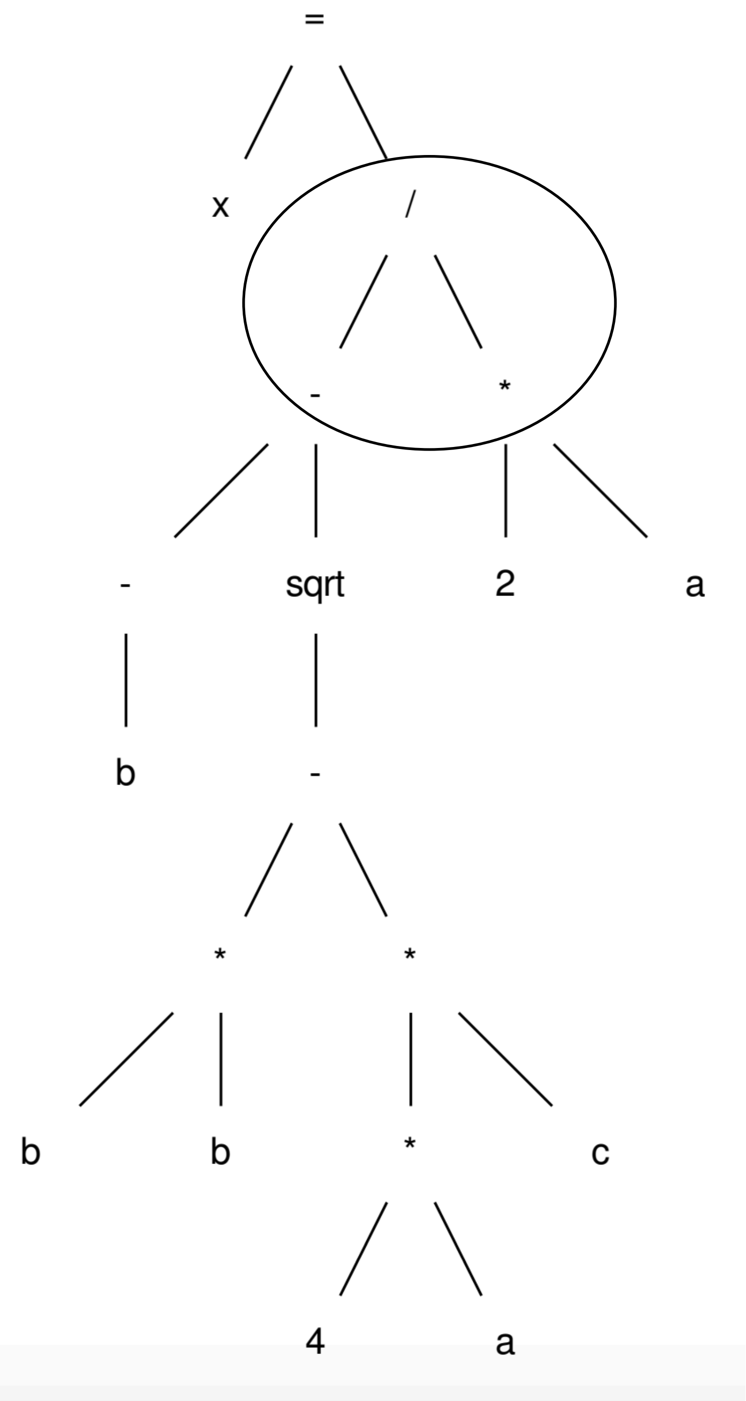
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
```



Simplify this code:

post-order traversal, using temporary variables

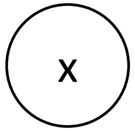
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
```



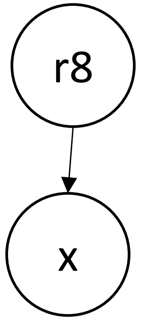
Now we build a “data dependency graph” (DDG)

```
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;
```

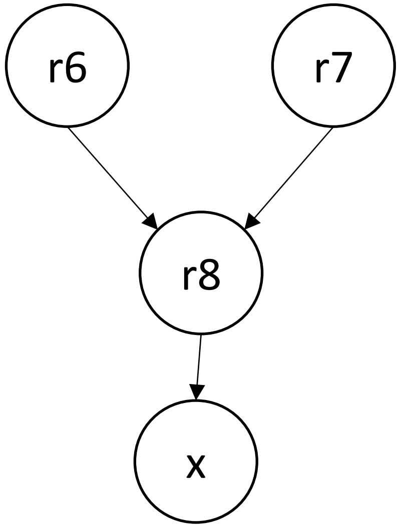
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



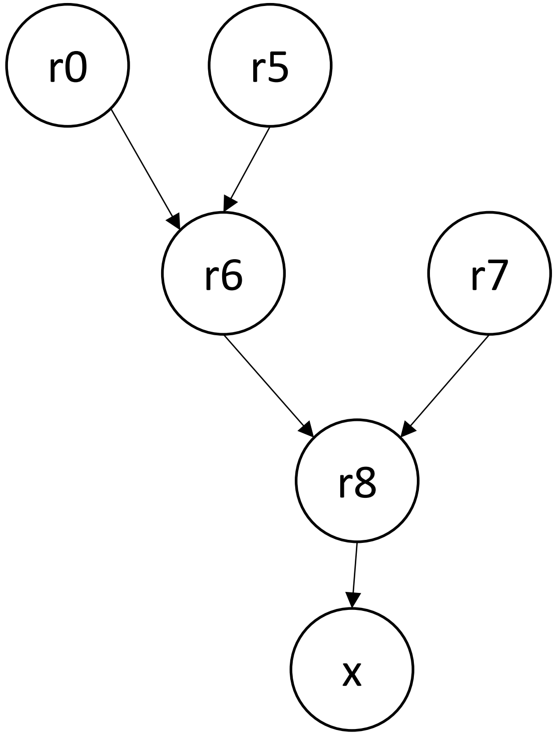
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



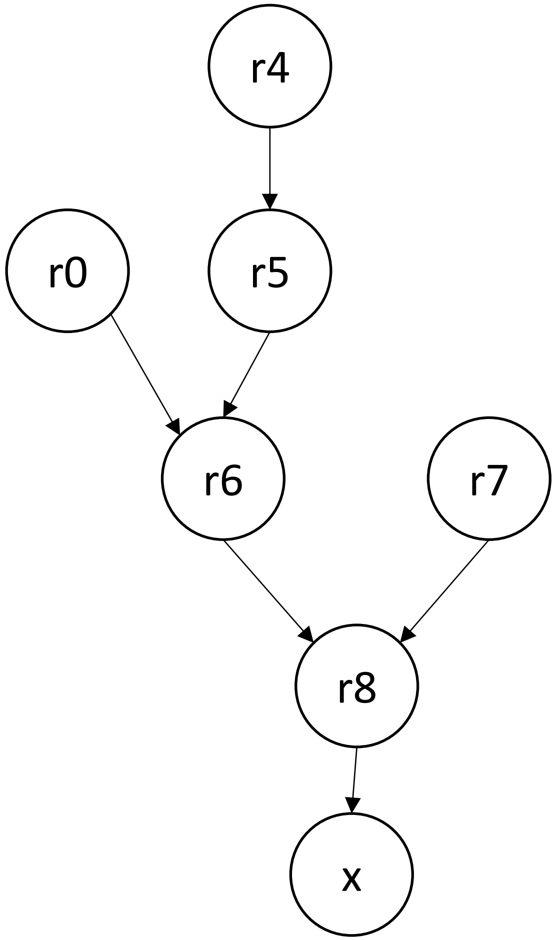
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



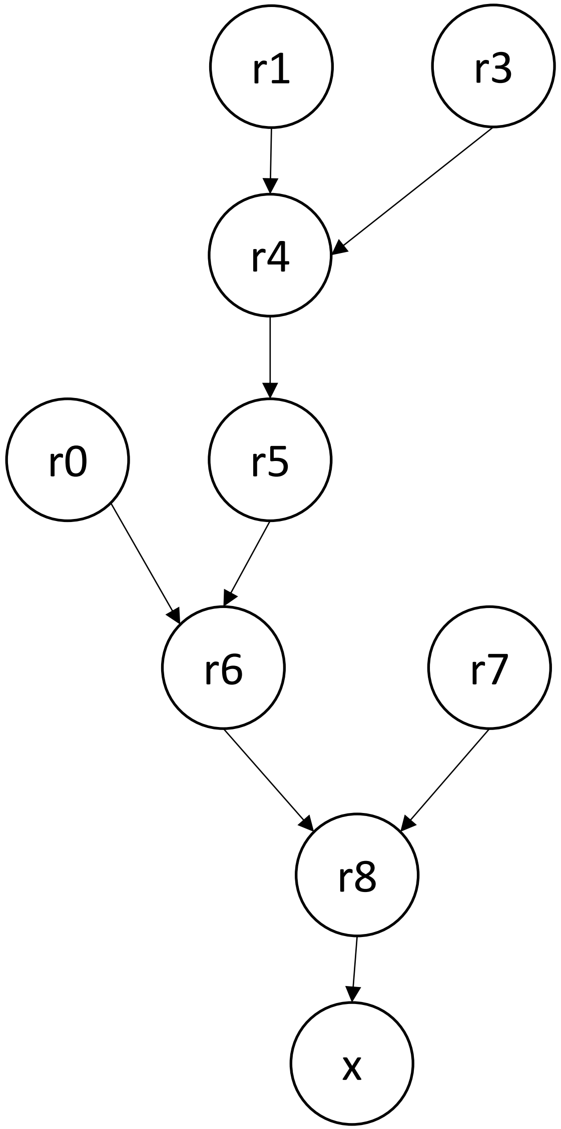
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



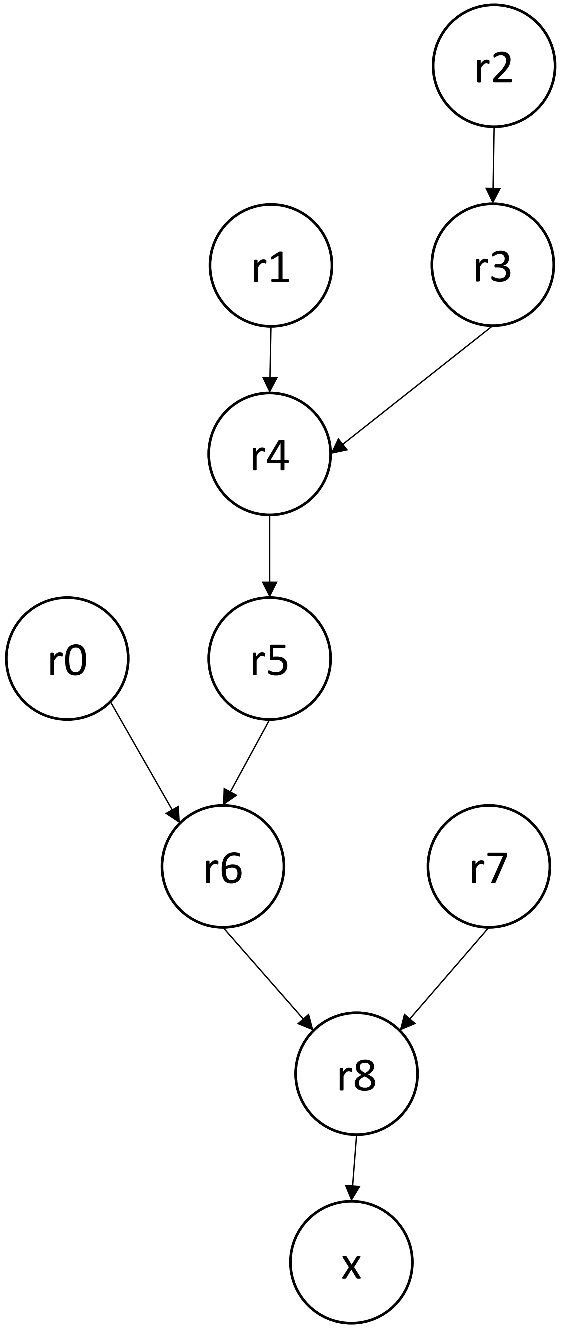
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



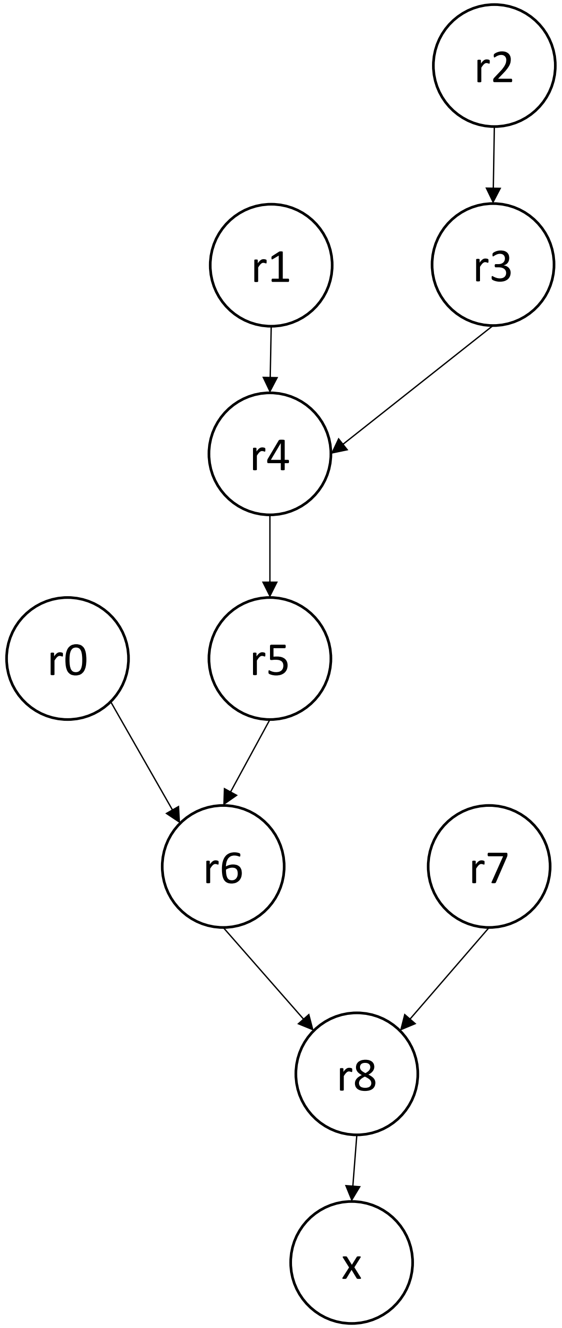
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



```
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;
```



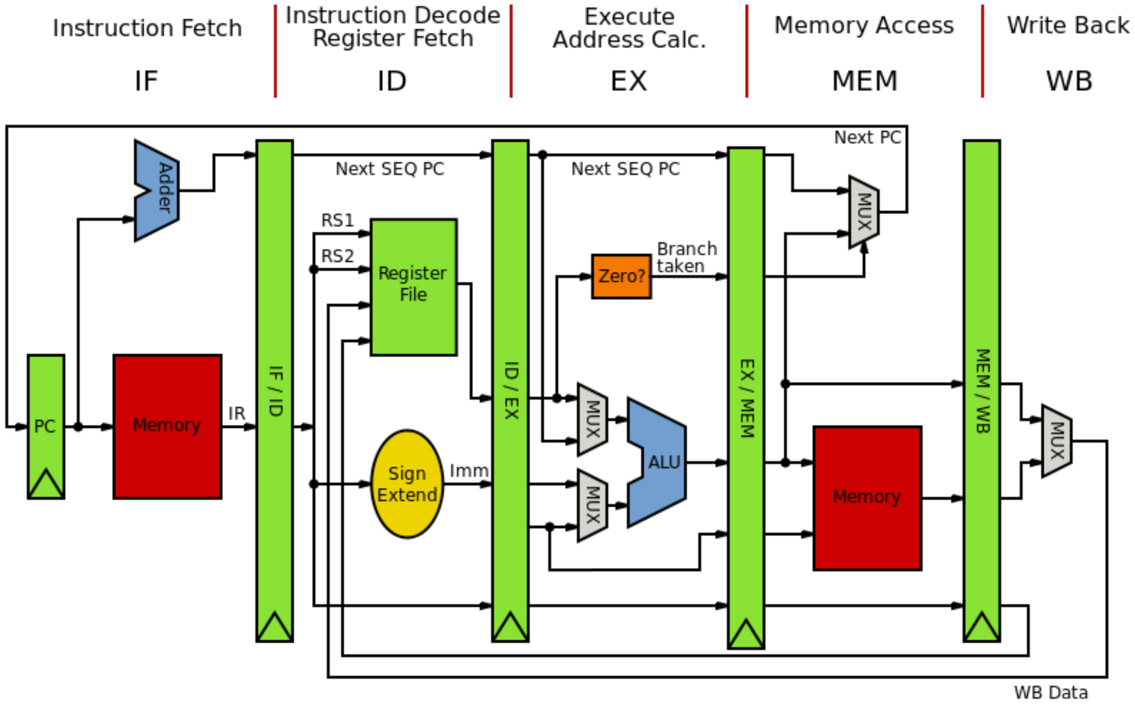
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



How can hardware execute ILP?

- Pipeline parallelism

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

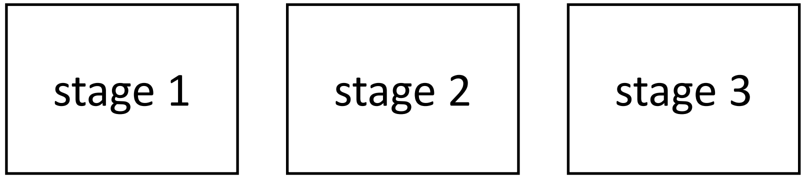


MIPS pipeline image from:
https://commons.wikimedia.org/wiki/Pipeline_(computer_hardware)

Pipeline

- Pipeline parallelism

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

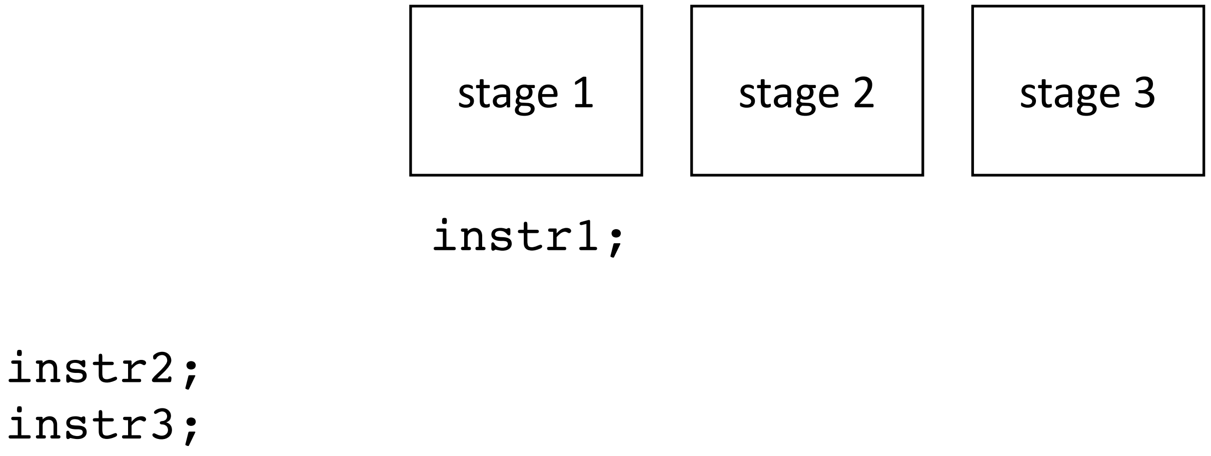


```
instr1;
instr2;
instr3;
```

Pipeline

- Pipeline parallelism

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



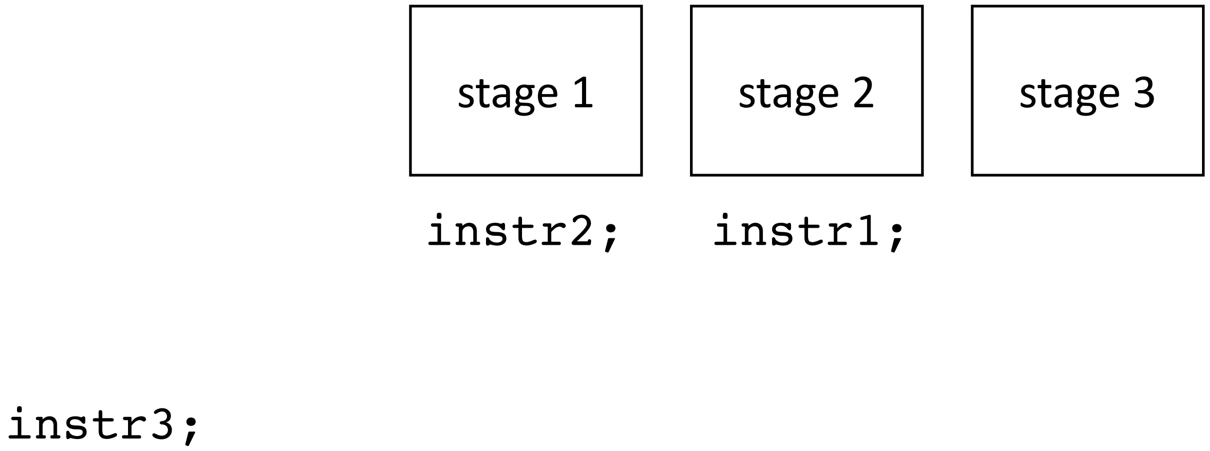
```
instr1;
```

```
instr2;
instr3;
```

Pipeline

- Pipeline parallelism

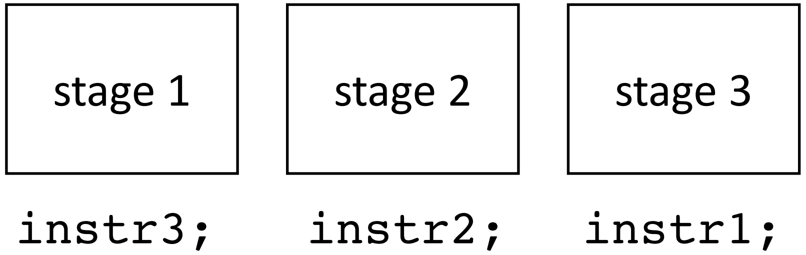
- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline



Pipeline

- Pipeline parallelism

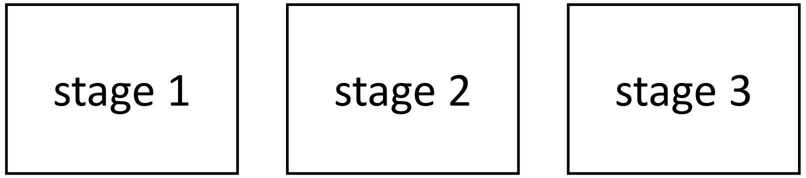
- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline



Pipeline

- Pipeline parallelism

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



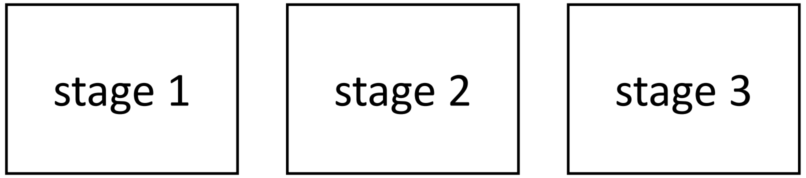
6 cycles for 3 independent instructions

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



stage 1

stage 2

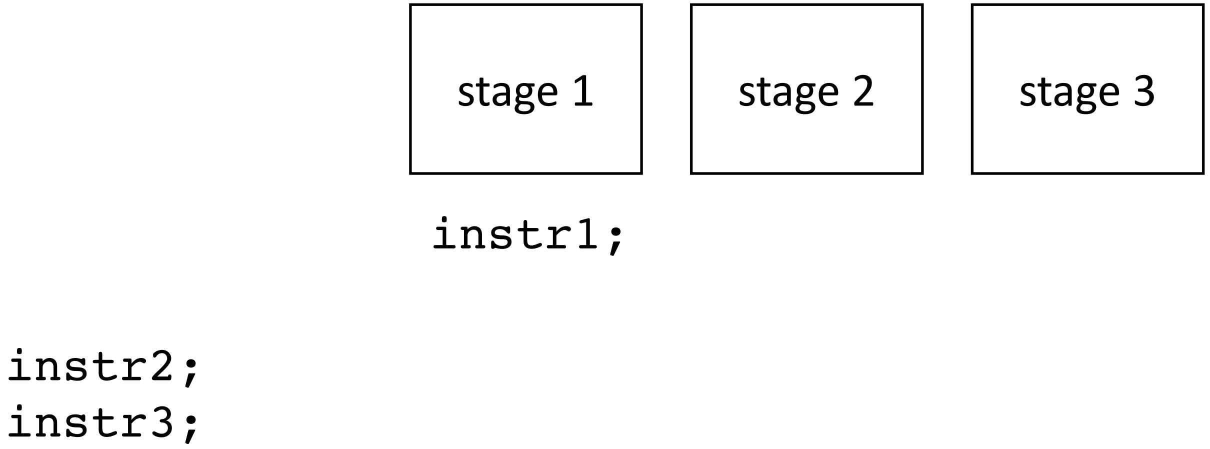
stage 3

```
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

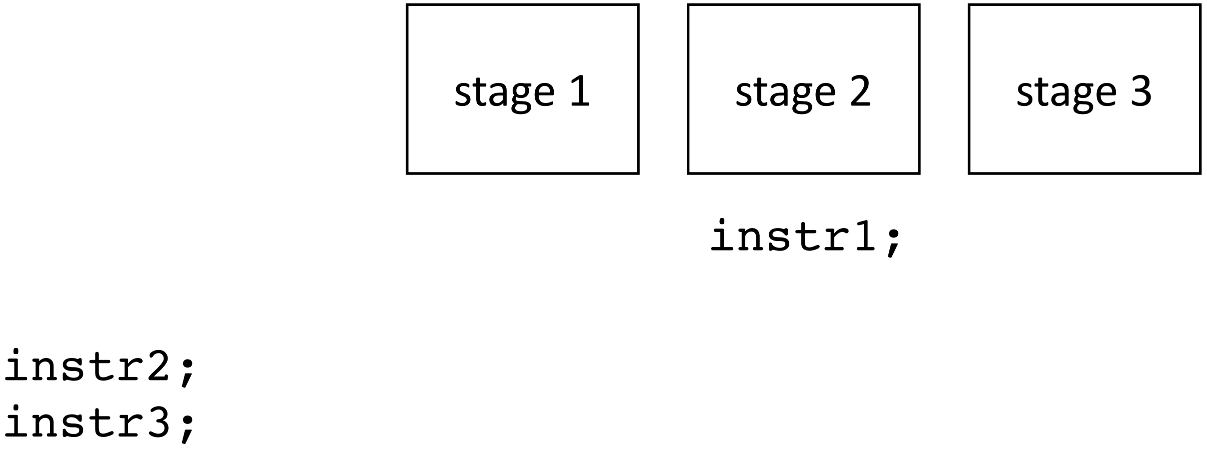


What if the instructions depend on each other?

Pipeline

- Pipeline parallelism

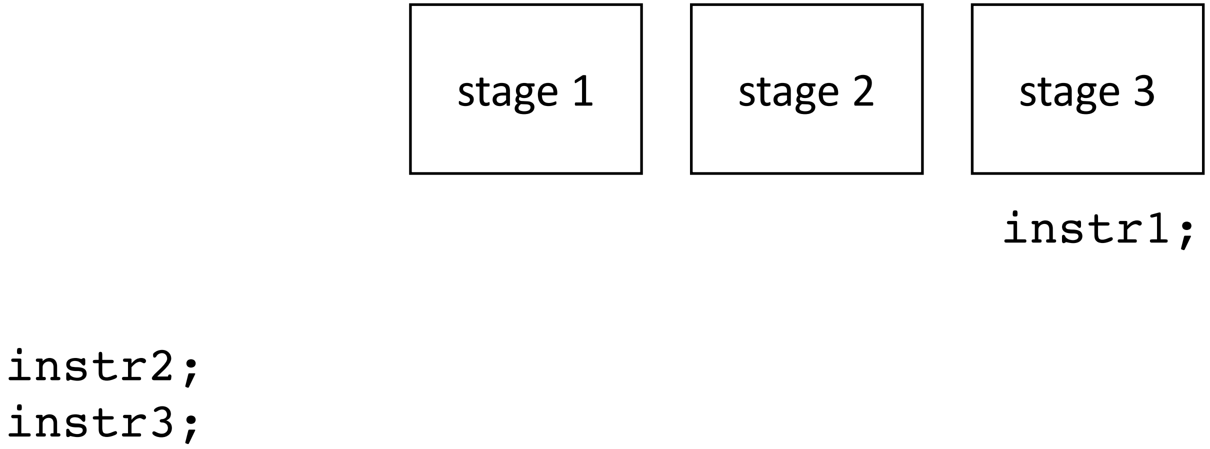
- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline



What if the instructions depend on each other?

Pipeline

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

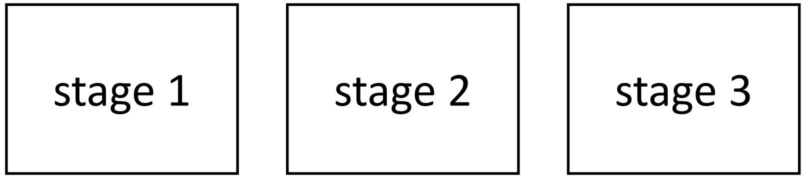


What if the instructions depend on each other?

Pipeline

- Pipeline parallelism

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



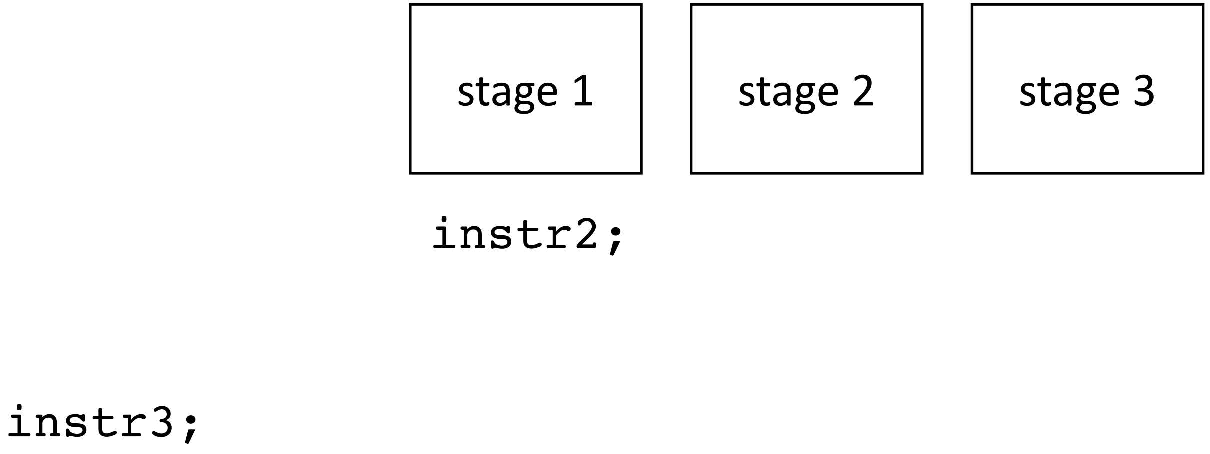
```
instr2;
instr3;
```

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



stage 1

stage 2

stage 3

`instr2;`

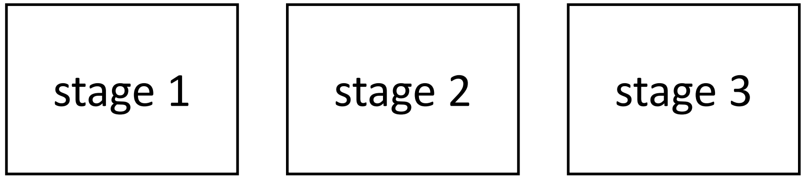
`instr3;`

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;`

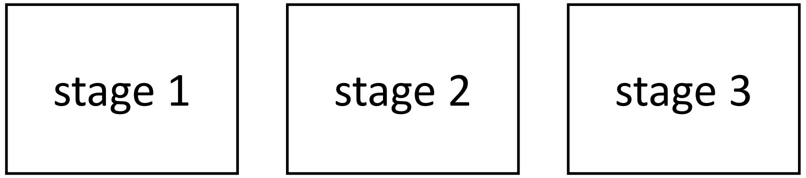
`instr3;`

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?

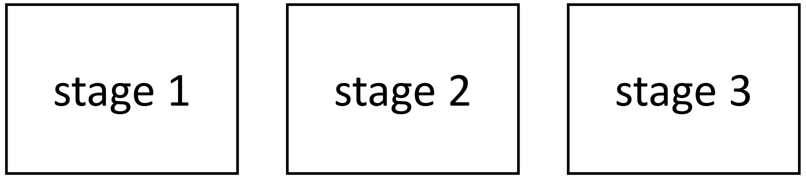
9 cycles for 3 instructions

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



stage 1

stage 2

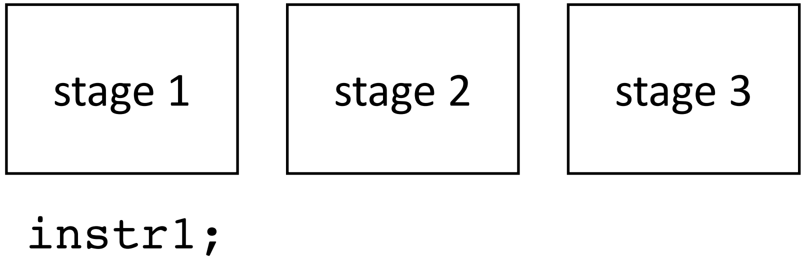
stage 3

```
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

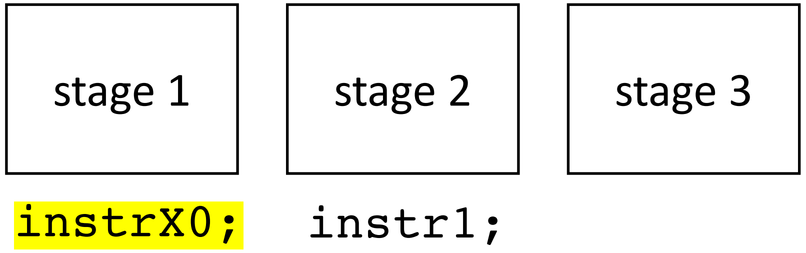


```
instrX0;
instrX1;
instr2;
instrX2;
instrX3;
instr3;
```

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

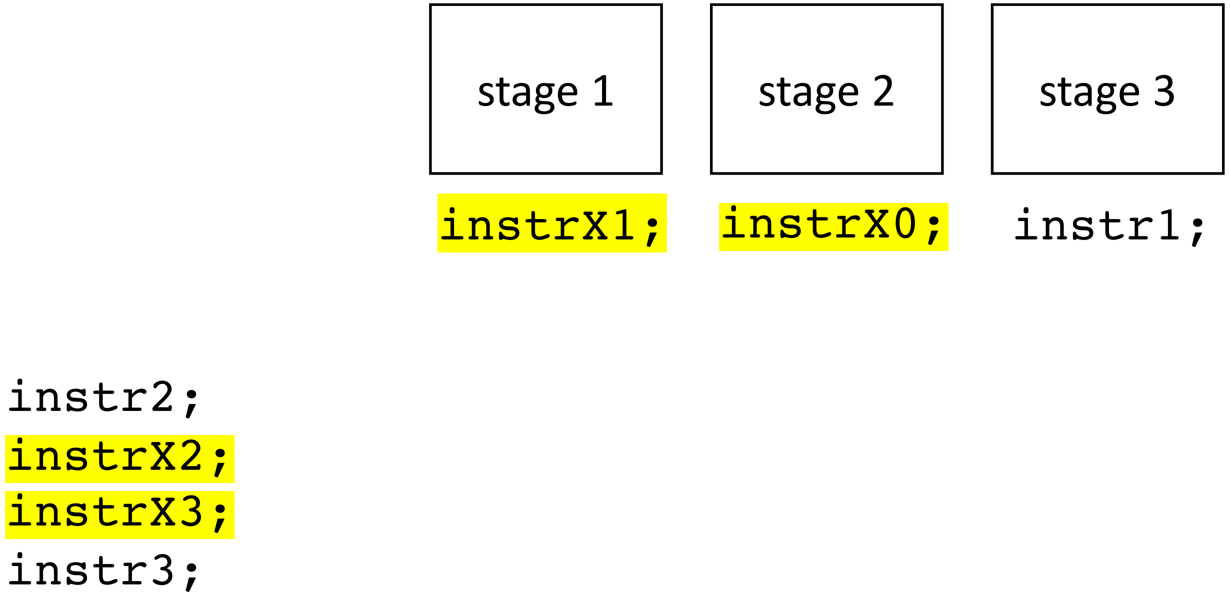


```
instrX1;
instr2;
instrX2;
instrX3;
instr3;
```

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



```
instrX1;   instrX0;   instr1;
```

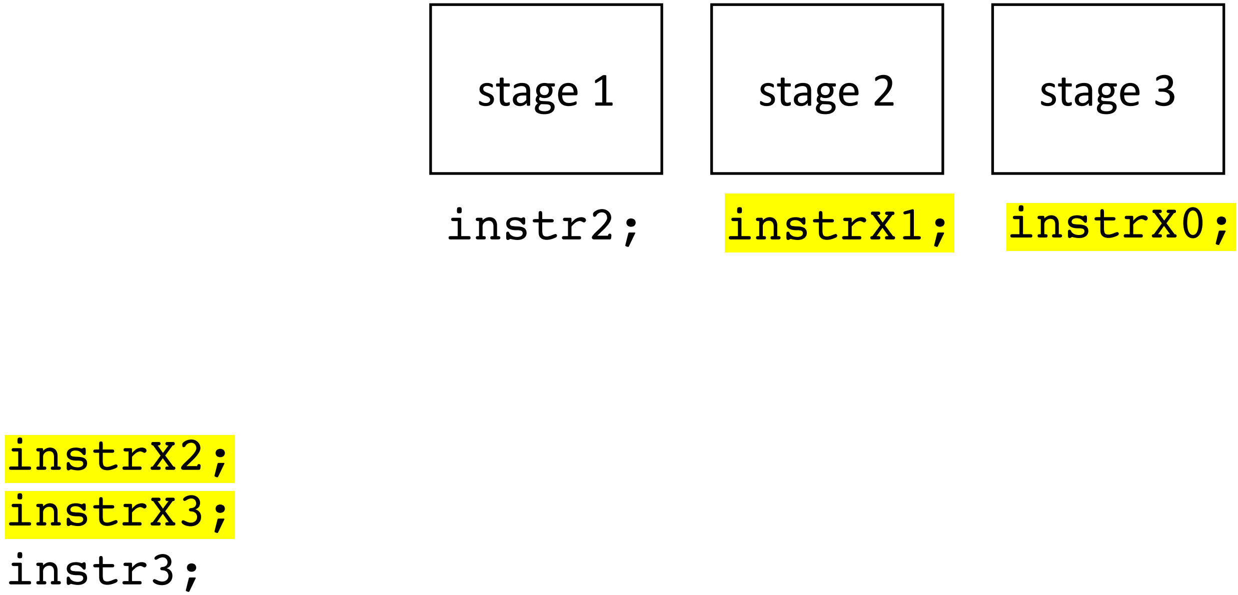
```
instr2;
instrX2;
instrX3;
instr3;
```

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



stage 1

stage 2

stage 3

instr2;

instrX1;

instrX0;

instrX2;

instrX3;

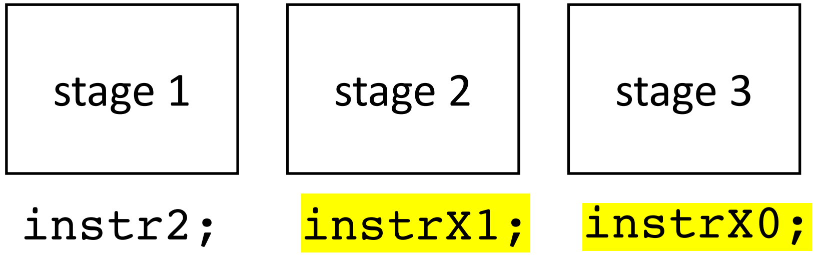
instr3;

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



stage 1

stage 2

stage 3

```
instr2;    instrX1;    instrX0;
```

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

and so on...

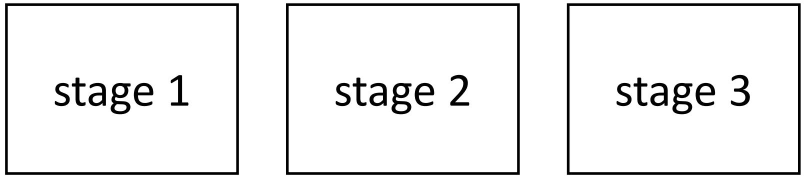
We converge to 1 cycle per instruction again!

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

Pipeline

- Pipeline parallelism

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



stage 1

stage 2

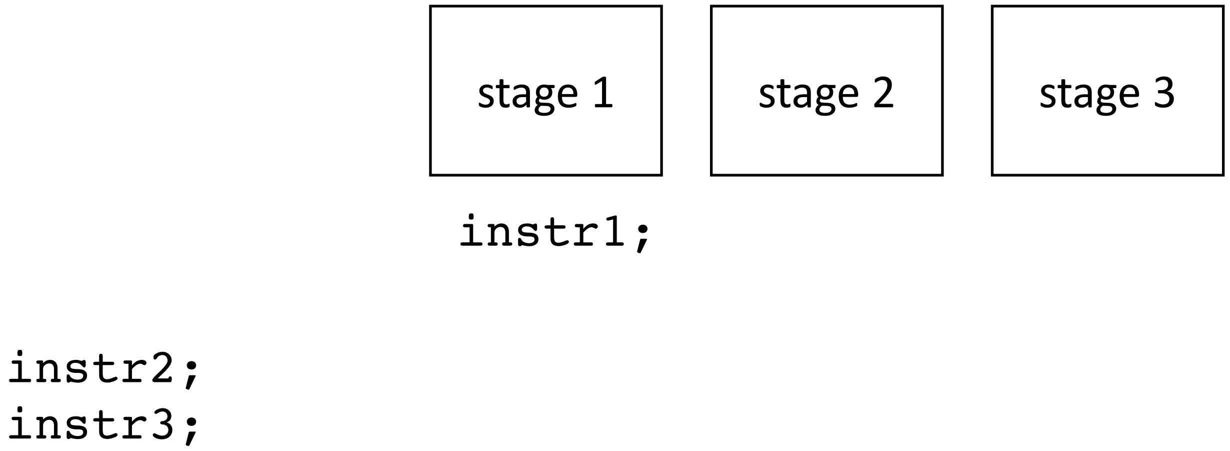
stage 3

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

Say instr2; and instr3; have a control dependence on instr1;

Pipeline

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



```
instr1;
```

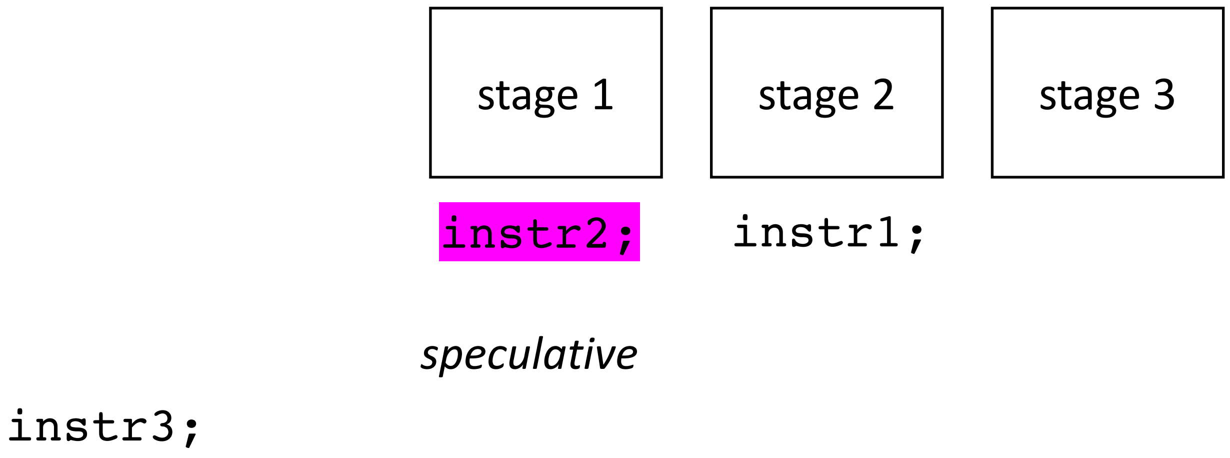
```
instr2;
instr3;
```

Say instr2; and instr3; have a control dependence on instr1;

Pipeline

- Pipeline parallelism

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



stage 1

stage 2

stage 3

instr2;

instr1;

speculative

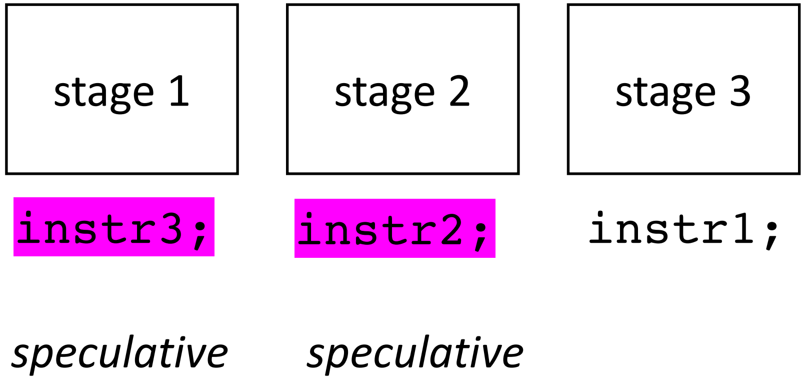
instr3;

Say instr2; and instr3;
have a control
dependence on instr1;

Pipeline

- Pipeline parallelism

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

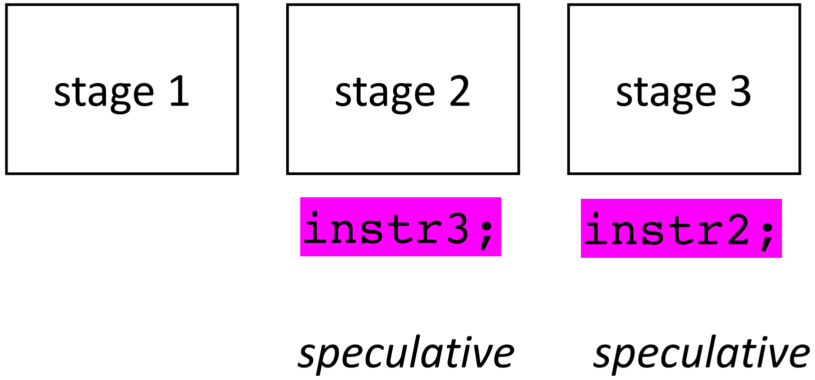


Say instr2; and instr3; have a control dependence on instr1;

Pipeline

- Pipeline parallelism

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



stage 1

stage 2

stage 3

`instr3;`

`instr2;`

speculative

speculative

Say instr2; and instr3;
have a control
dependence on instr1;

before we commit
the speculative instructions,
we check if the control
dependence was satisfied.

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

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

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

How can hardware execute ILP?

- Executing multiple instructions at once:

- Superscalar architecture:
  - Several sequential operations are issued in parallel
  - hardware detects dependencies

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

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

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!

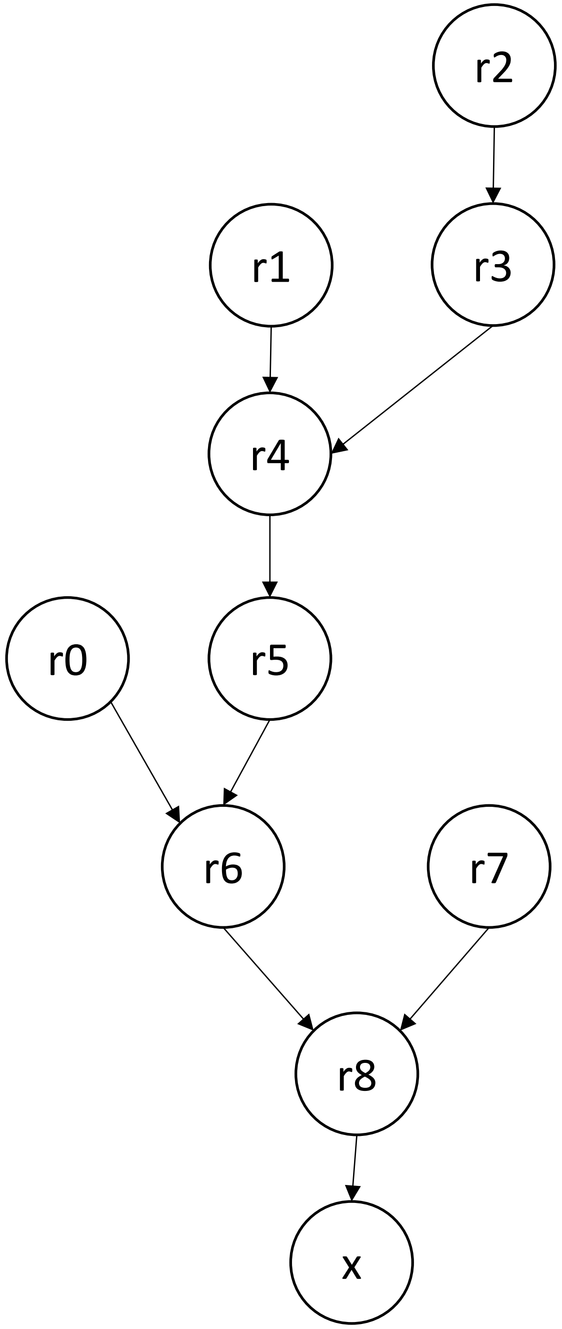
Three techniques to optimize for ILP

- 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;
```

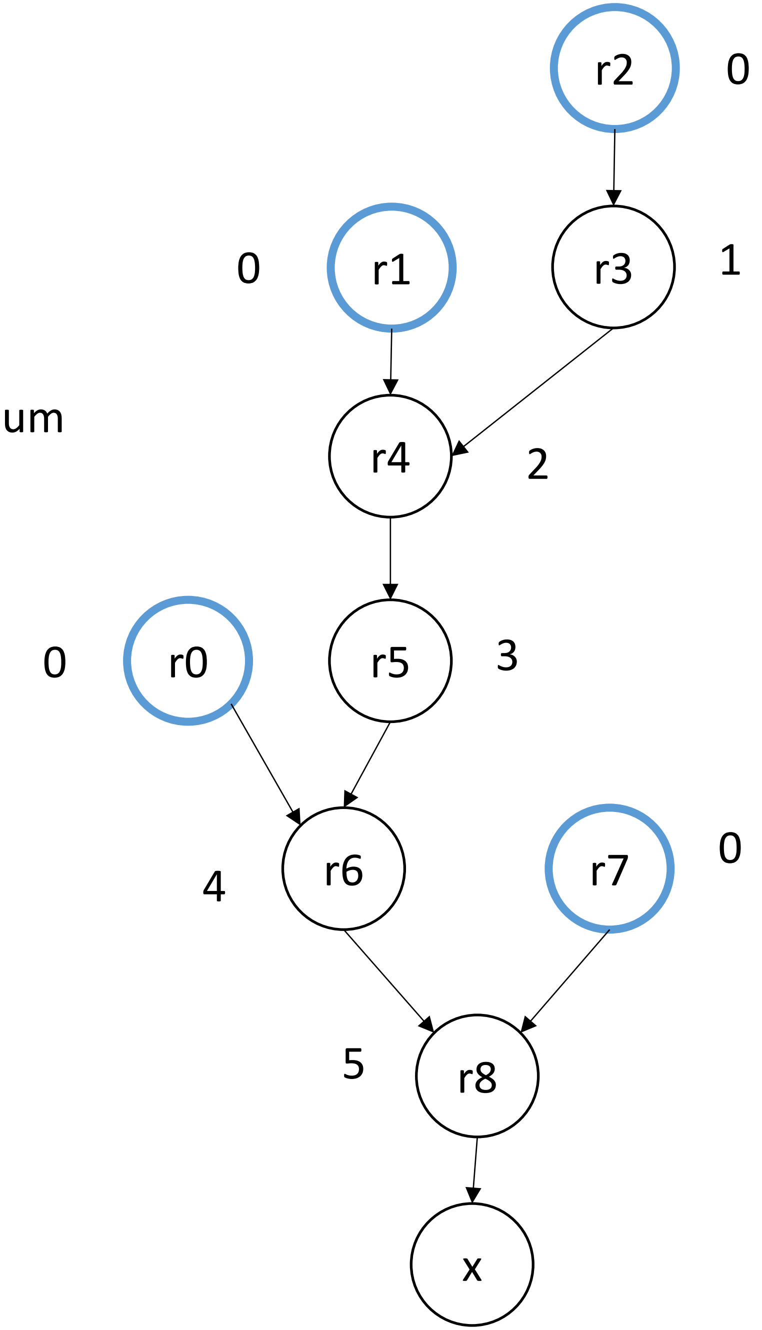
First, consider optimizing for superscalar



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

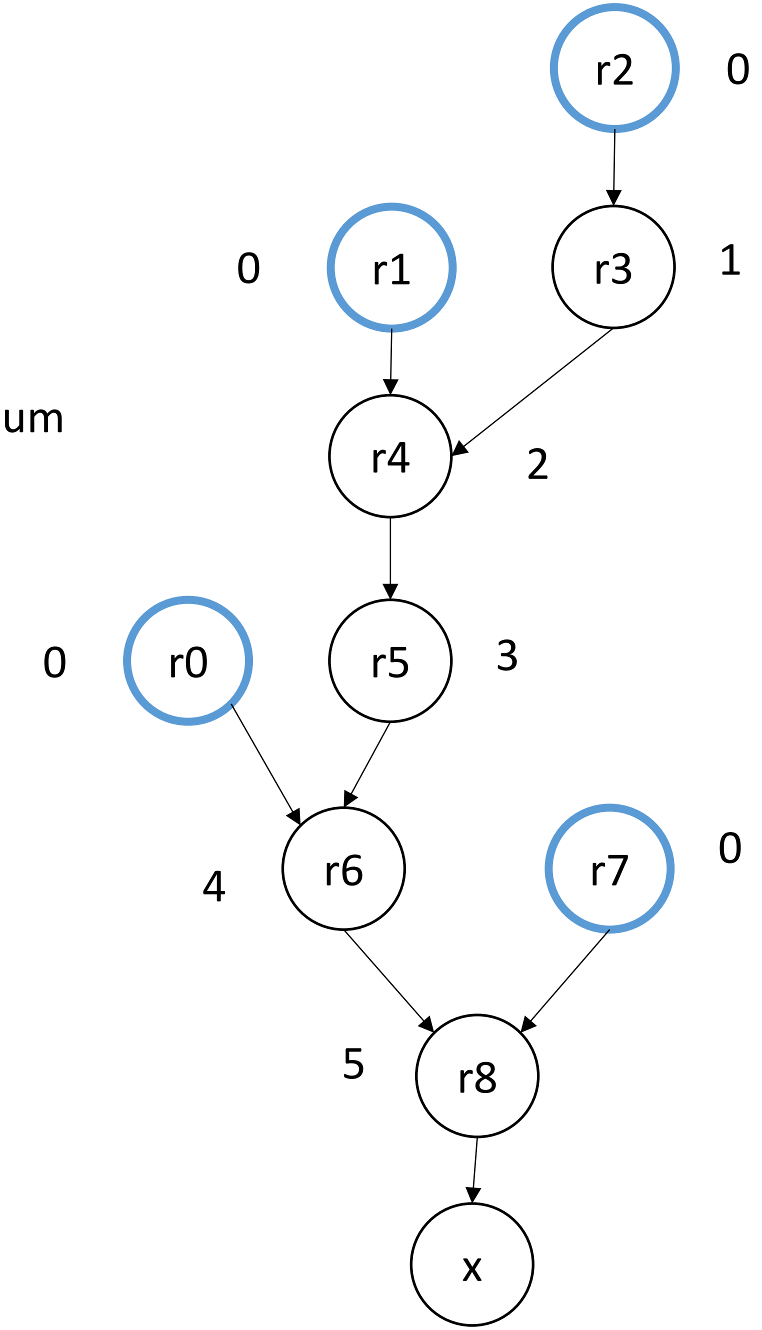


Priority Topological Ordering of DDGs for Superscalar

```
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

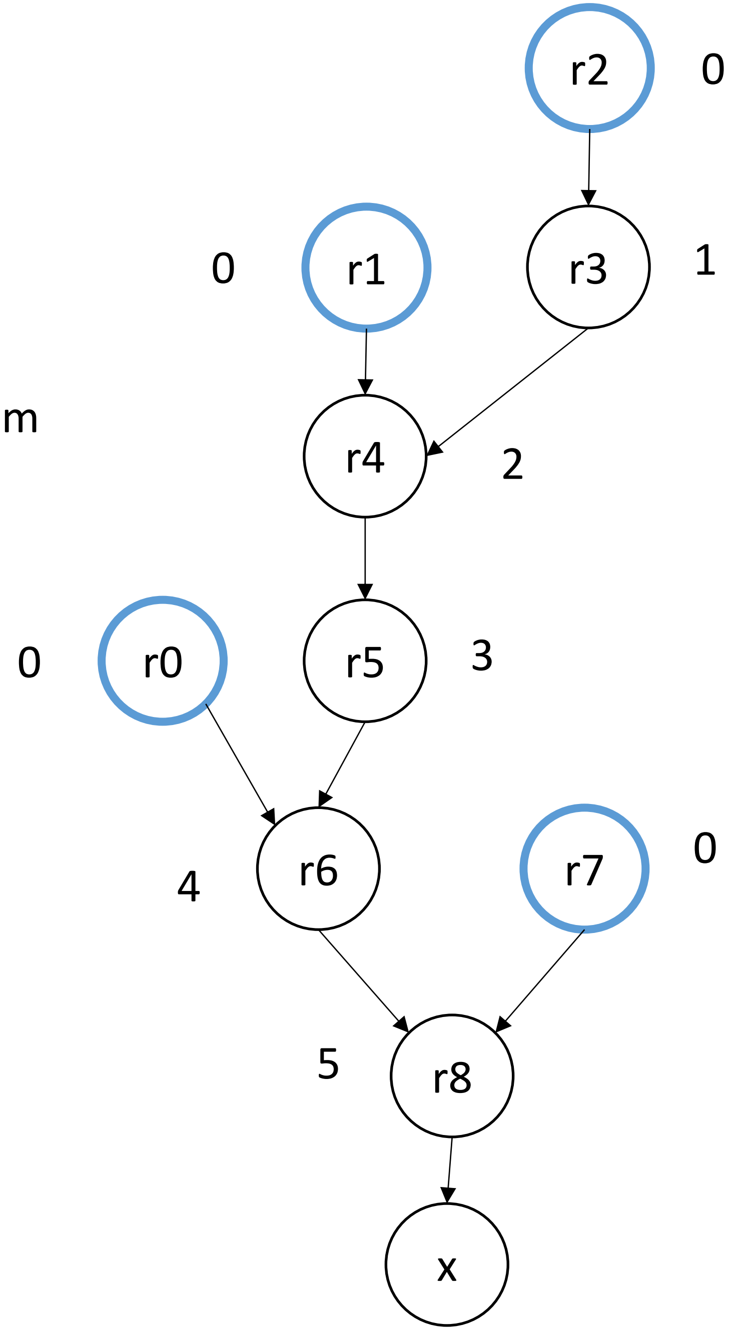


Priority Topological Ordering of DDGs for Superscalar

```
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

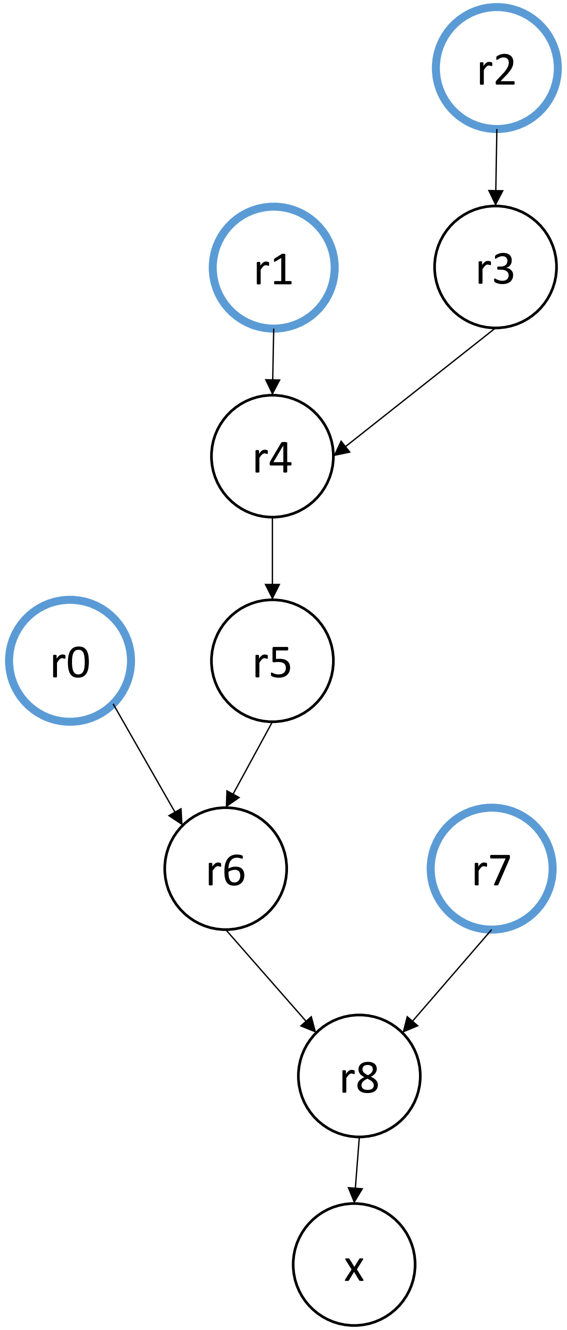
Break ties in topological order using this number



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?

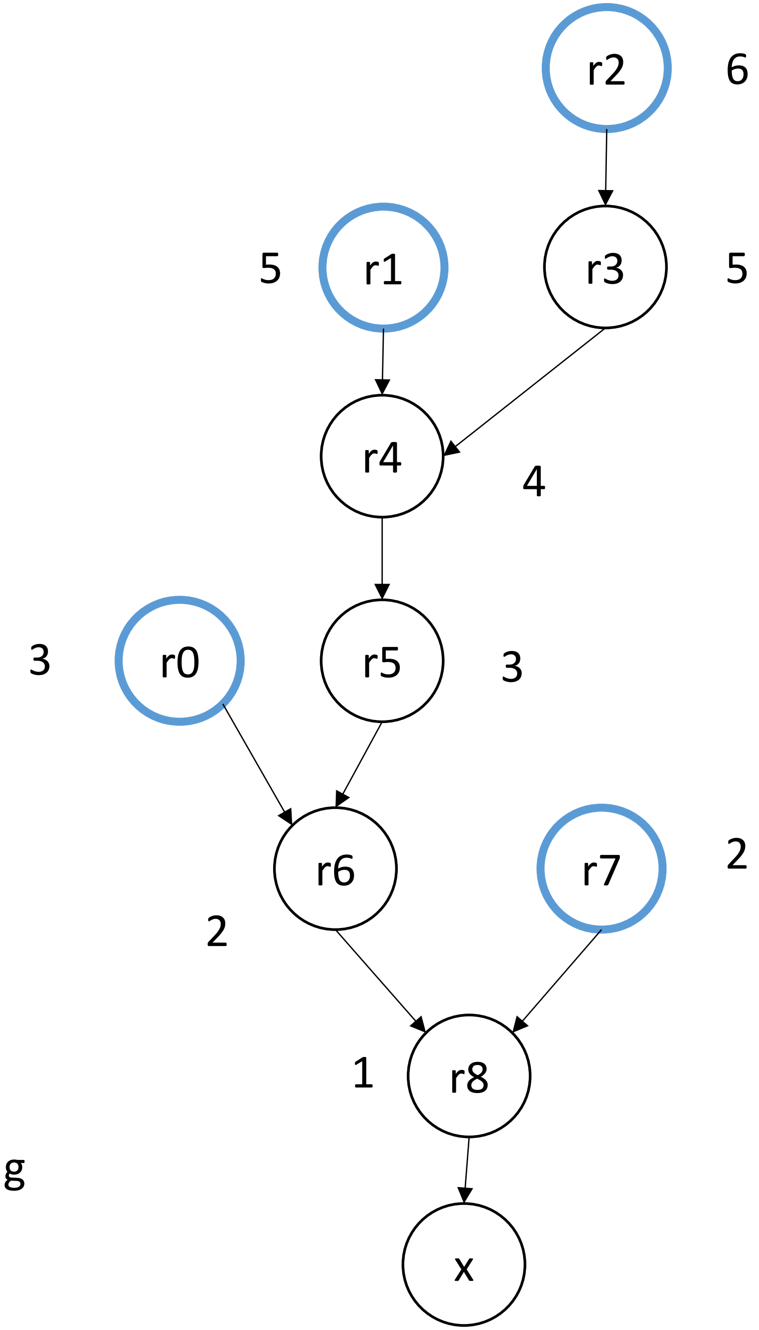


Priority Topological Ordering of DDGs for Pipelining

```
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 intendent instructions as high as possible. What about pipelining?

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

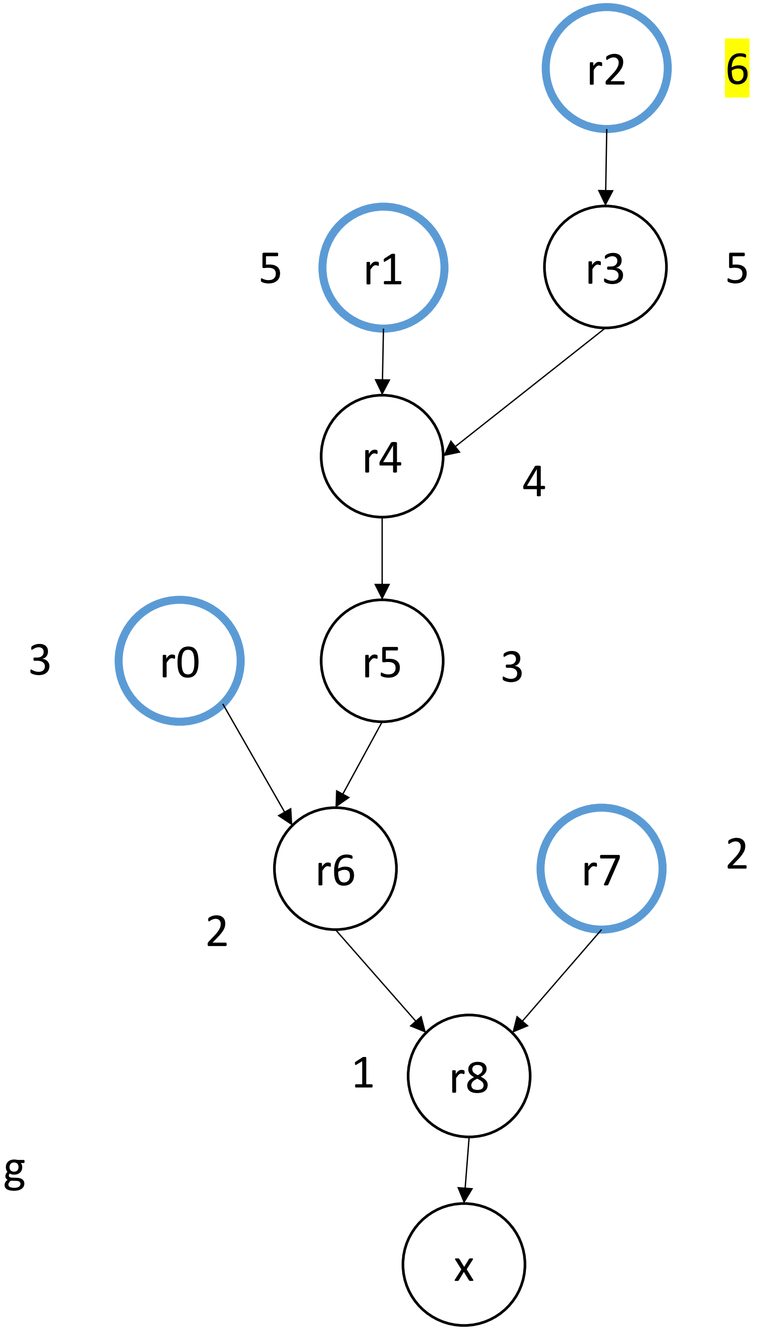


Priority Topological Ordering of DDGs for Pipelining

```
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 intendent instructions as high as possible. What about pipelining?

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

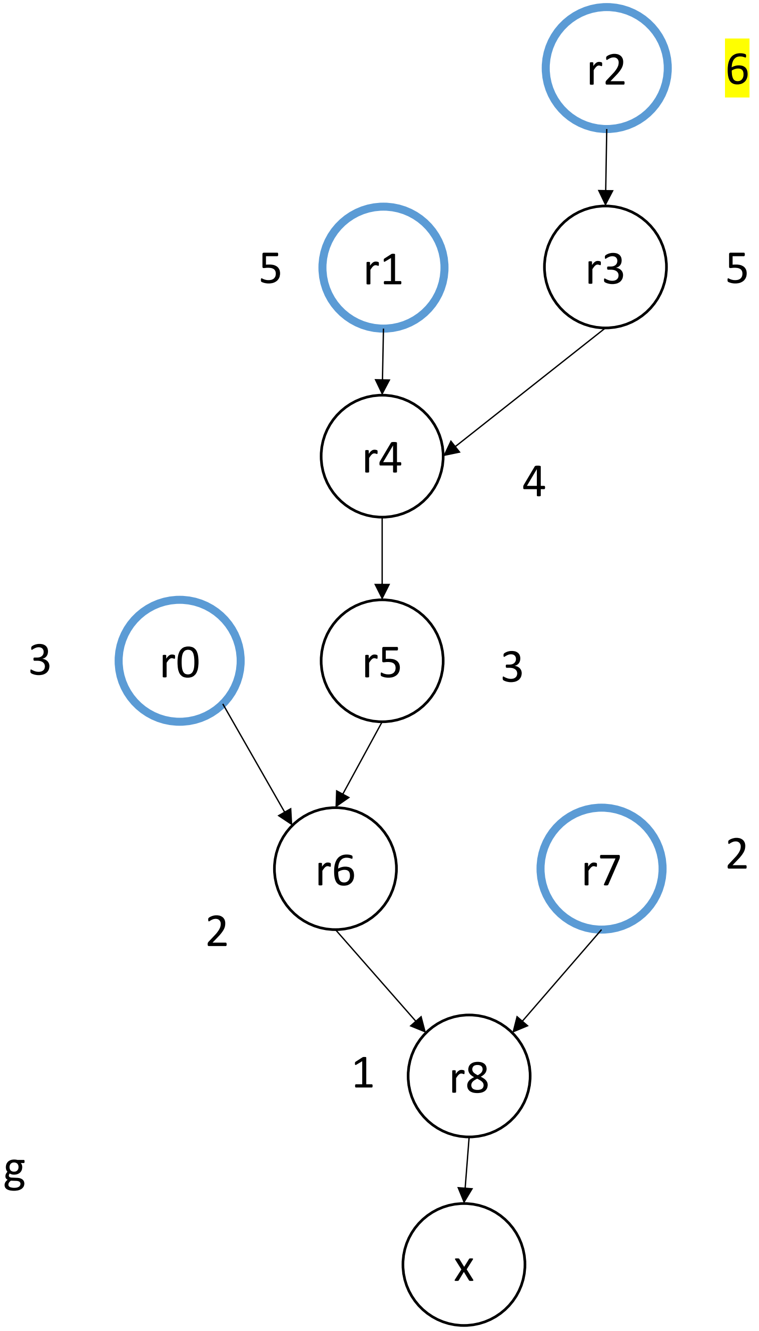


Priority Topological Ordering of DDGs for Pipelining

```
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 intendent instructions as high as possible. What about pipelining?

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

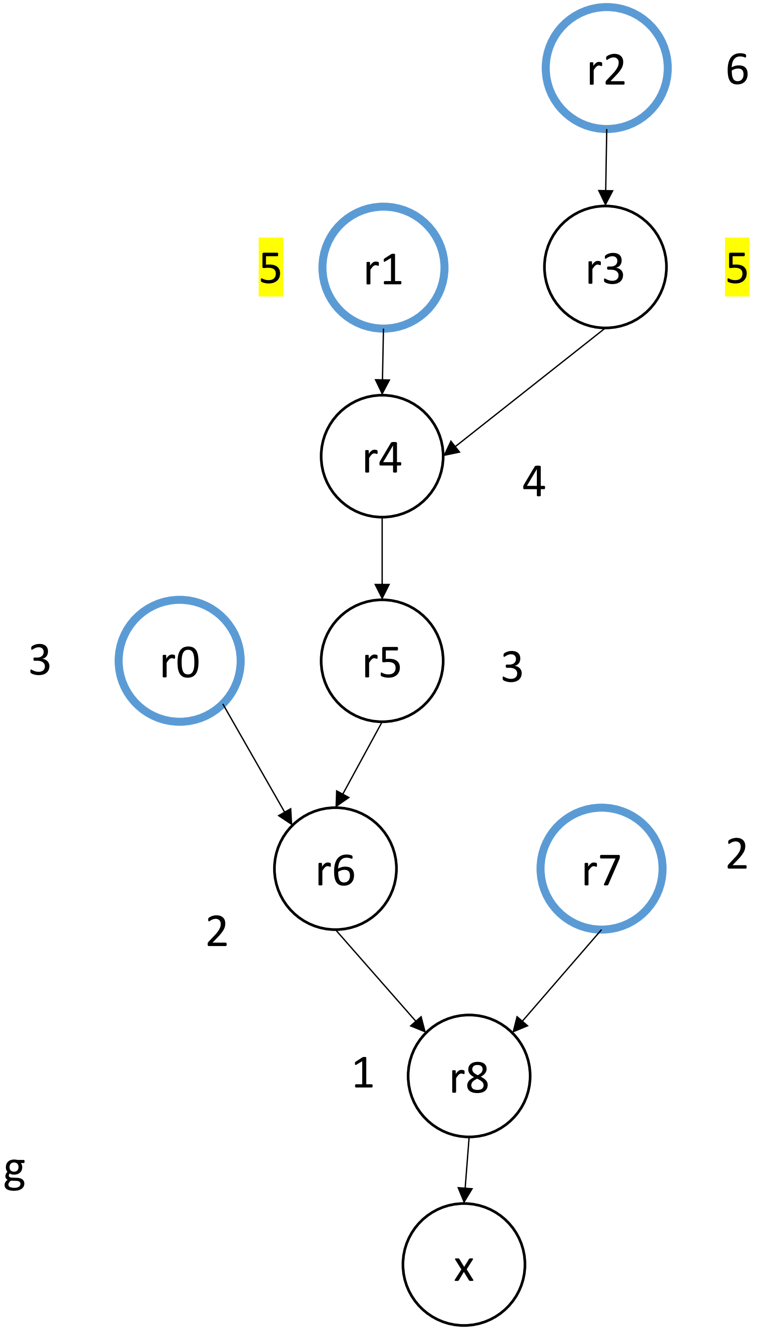


Priority Topological Ordering of DDGs for Pipelining

```
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

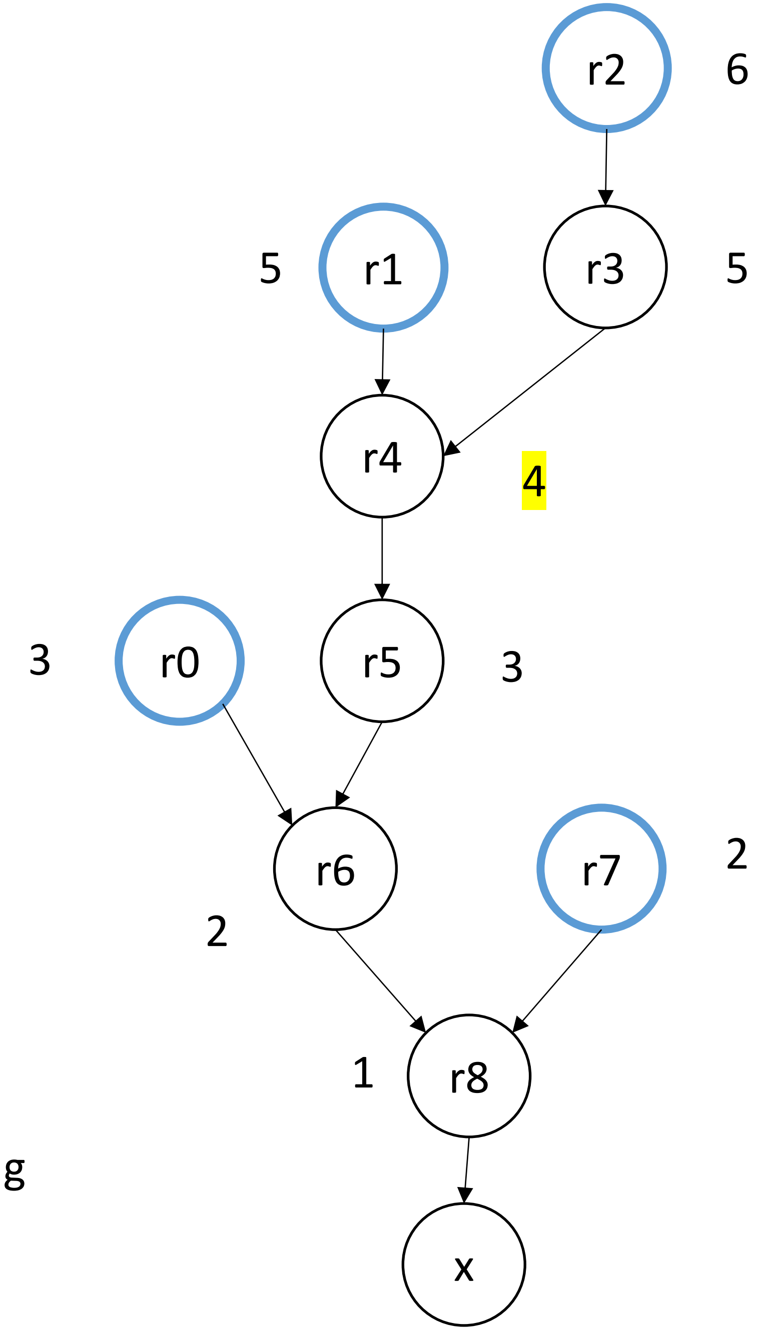


Priority Topological Ordering of DDGs for Pipelining

```
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



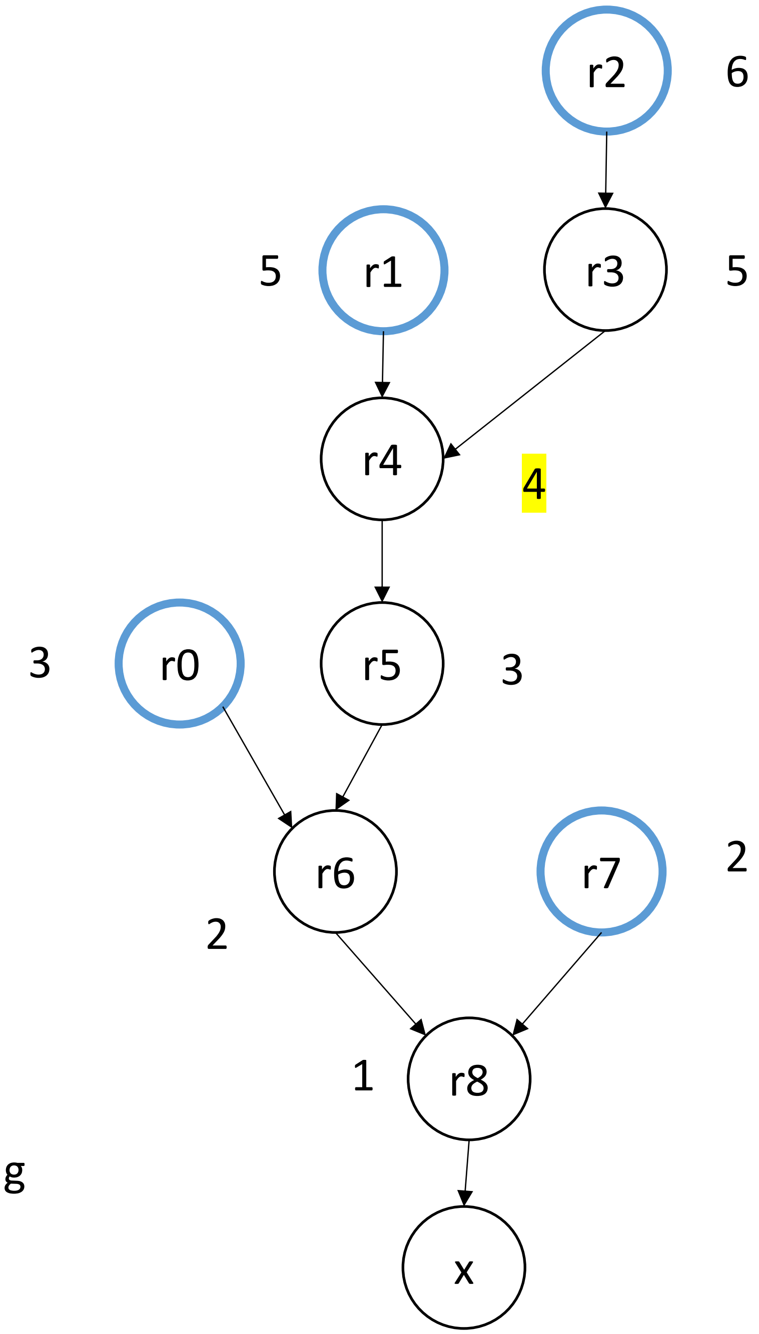
Priority Topological Ordering of DDGs for Pipelining

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

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 `SEQ(i,j)` be the jth 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);
}
```

Let `SEQ(i,j)` be the jth 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+1,1);
    ...
    SEQ(i,N);
    SEQ(i+1, N);
}
```

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

Let `SEQ(i,j)` be the jth 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+1,1);
    ...
    SEQ(i,N);
    SEQ(i+1, N);
}
```

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!

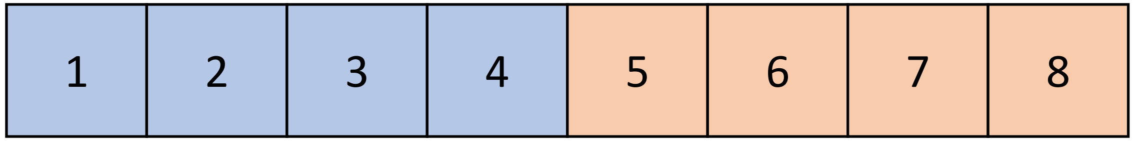
Loop Unrolling for Reduction Loops

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

1	2	3	4	5	6	7	8

Loop Unrolling for Reduction Loops

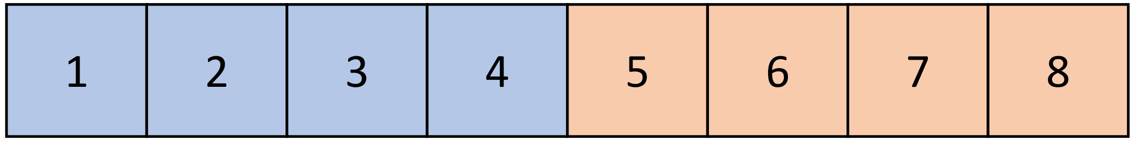
- chunk array in equal sized partitions and do local reductions
- Consider size 2:



1	2	3	4	5	6	7	8

Loop Unrolling for Reduction Loops

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

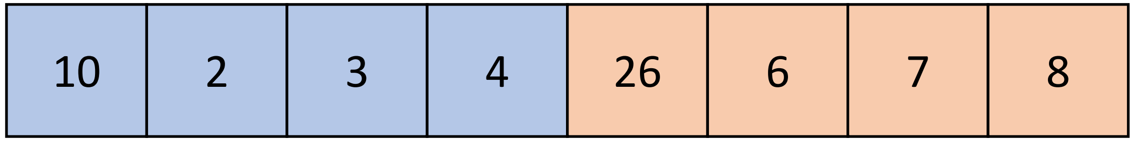


1	2	3	4	5	6	7	8

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:

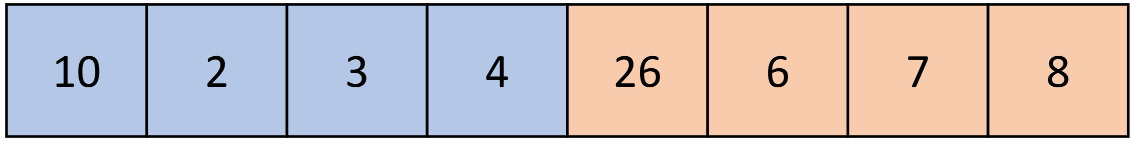


10	2	3	4	26	6	7	8

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:

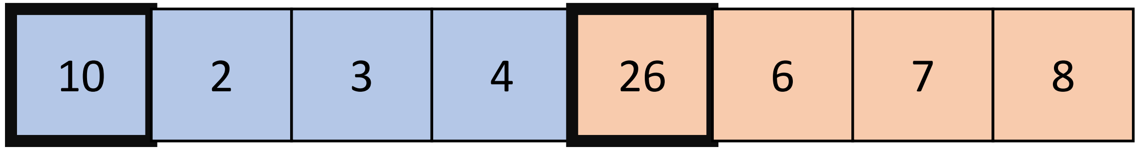


10	2	3	4	26	6	7	8

Add together base locations

Loop Unrolling for Reduction Loops

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

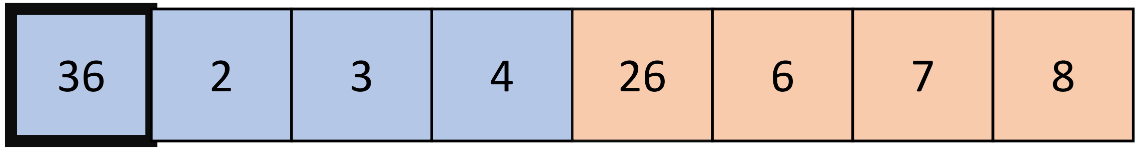


10	2	3	4	26	6	7	8

Add together base locations

Loop Unrolling for Reduction Loops

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



36	2	3	4	26	6	7	8

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;   | independent instructions. If they overwrite the register storing instr1’s result, then it will have to
instrX1;   | be stored to memory and retrieved before instr2
...
instr2;
```

Watch out!

- Our abstraction: separate dependent instructions as far as possible

- Pros:
  - Simple

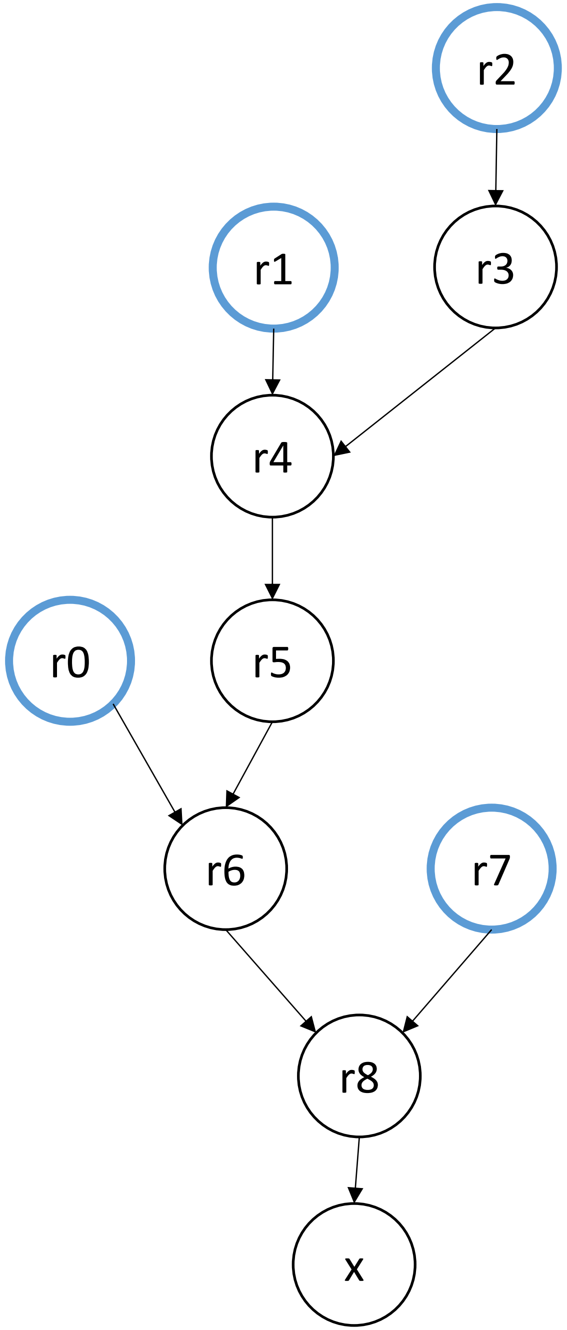
- Cons:
  - Can lead to register spilling, causing expensive loads

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

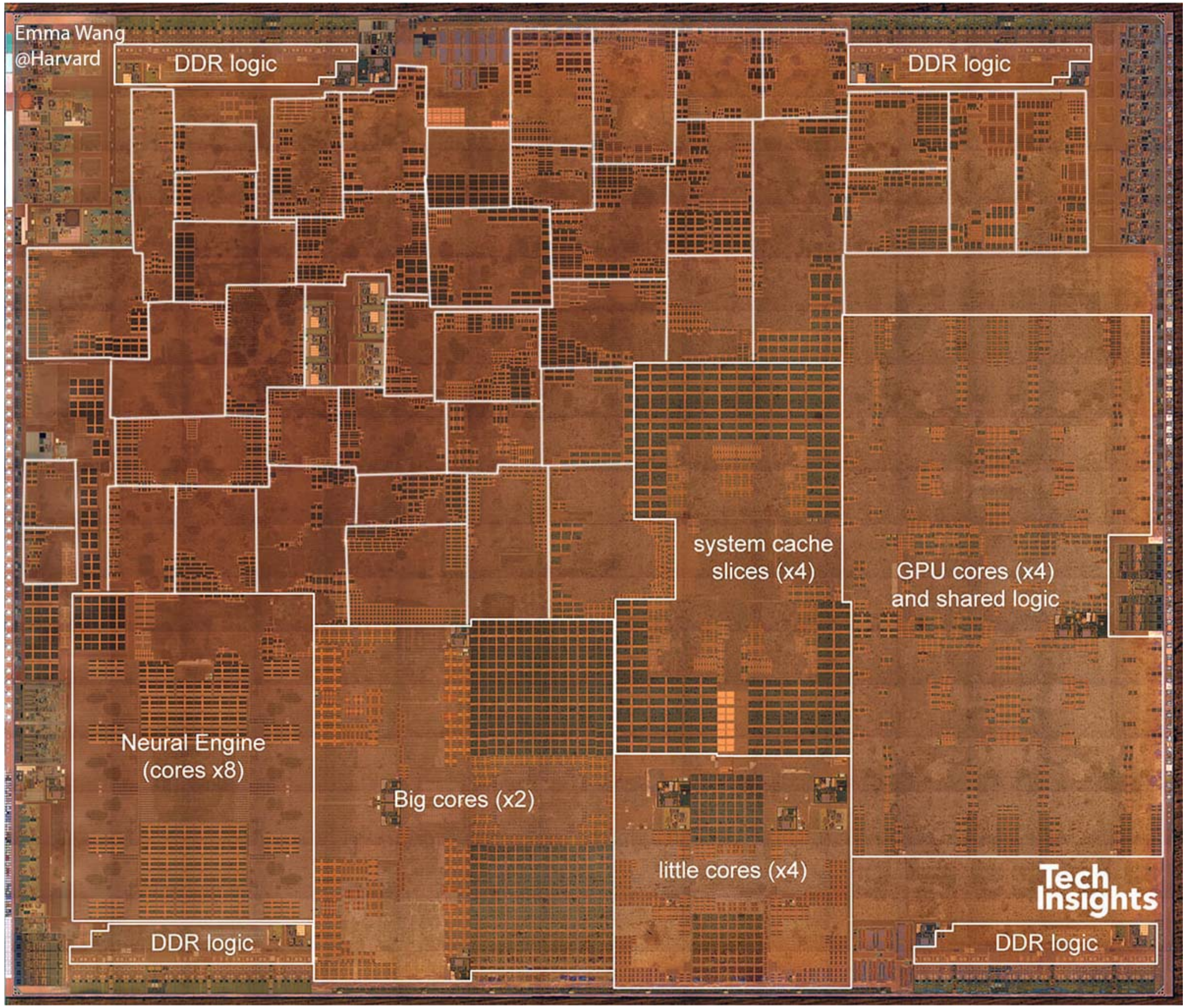
- Where is parallelism most commonly found?
    - Non-numeric applications are thought to have very little. lots of:
        - I/O (file, network, user),
        - events,
        - *source needed*
    - numeric applications have more:
        - media processing (image, video, sound)
        - machine-learning (esp. inference)
- More and more, numeric applications are moving to accelerators

Modern SoC

- From David Brooks lab at Harvard:

  http://vlsiarch.eecs.harvard.edu/research/accelerators/die-photo-analysis/

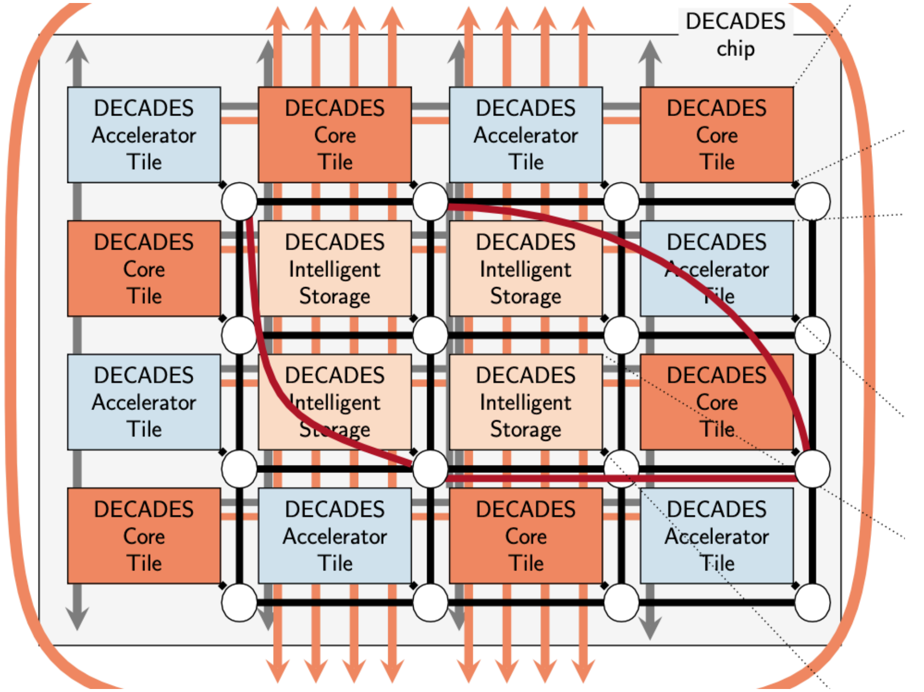
- Compilers will need to be able to map software efficiently to a range of different accelerators



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



Academic prototype chip that I worked on at Princeton!

C++ Threads

- Introduction
  - Learn as needed throughout class

- Multi-threading officially introduced in C++11
  - only widely available after ~2014
  - official specification
  - cross-platform

- Before C++ threads
  - pthreads

C++ Threads

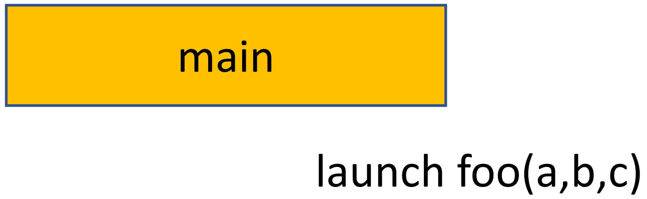
- Introduction
  - Learn as needed throughout class

- Multi-threading officially introduced in C++11
  - only widely available after ~2014
  - official specification
  - cross-platform

- Before C++ threads
  - pthreads
  - volatile

C++ Threads

- Main idea:
  - run functions concurrently

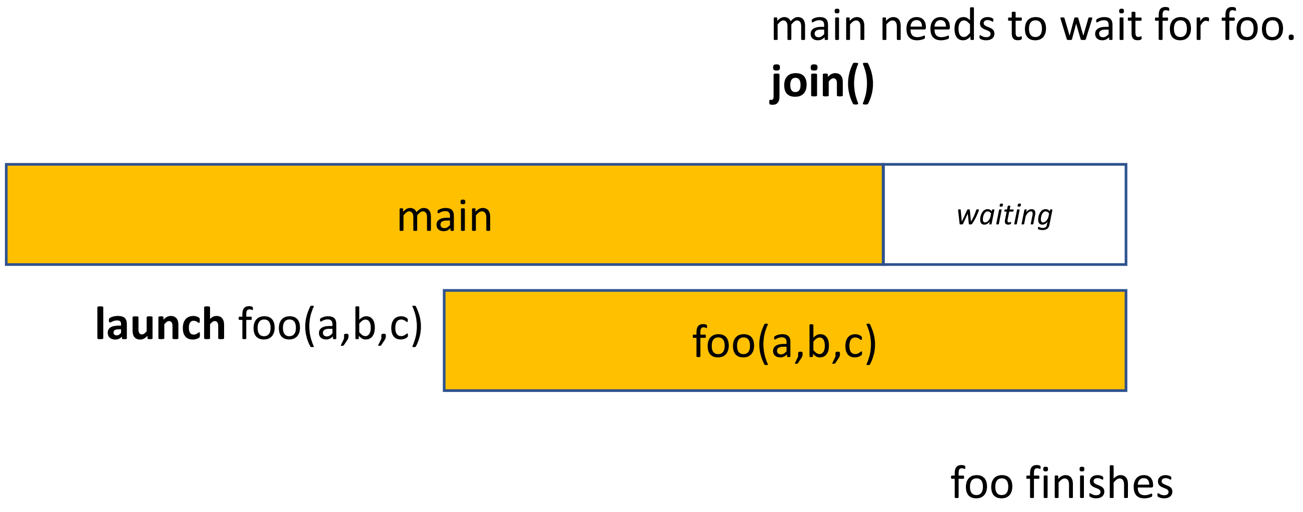


main

launch foo(a,b,c)

C++ Threads

- Main idea:
  - run functions concurrently

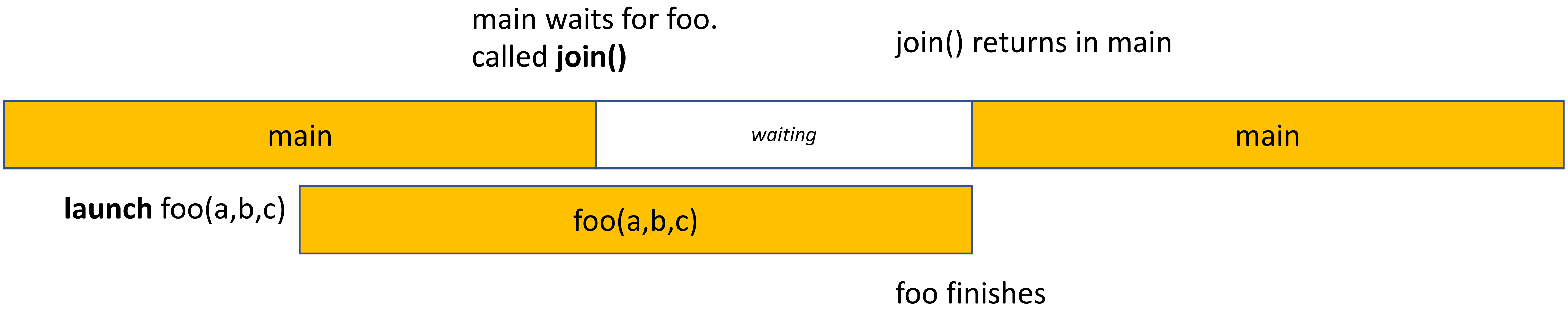


main needs to wait for foo.
join()

foo finishes

C++ Threads

- Main idea:
  - run functions concurrently



main waits for foo.
called **join()**

join() returns in main

main

waiting

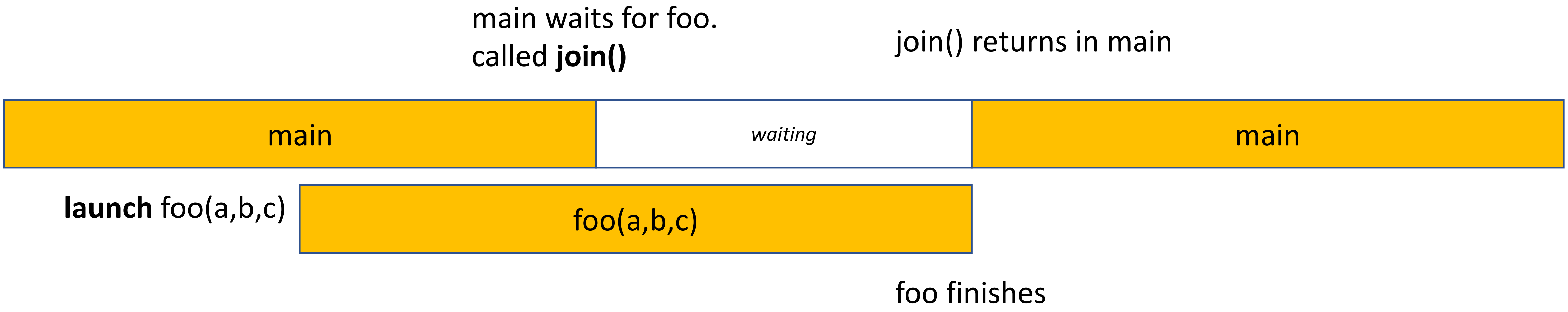
main

launch foo(a,b,c)

foo(a,b,c)

foo finishes

```cpp
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
  // some foo code
}

int main() {
  // some main code
  thread thread_handle (foo,1,2,3);
  // code here runs concurrently with foo
  thread_handle.join();
  return 0;
}
```



main waits for foo.
called **join()**

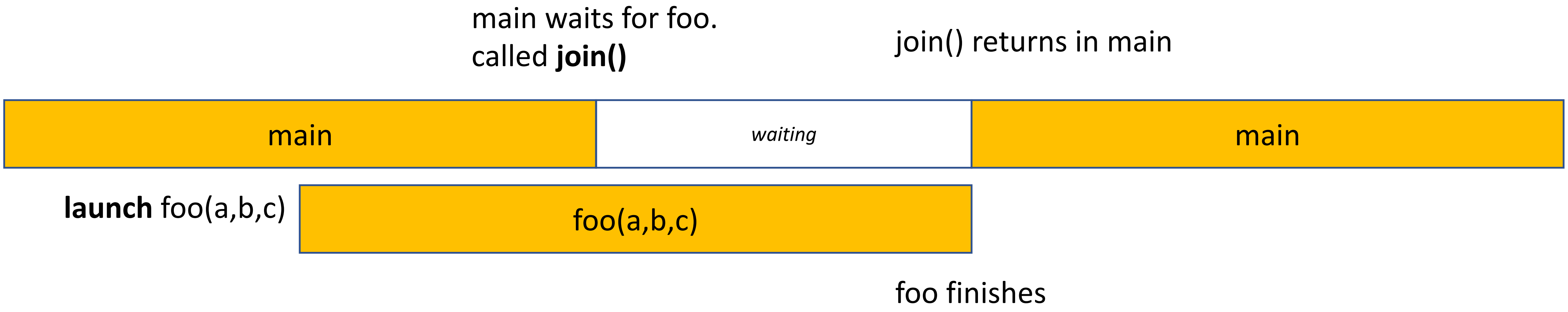
join() returns in main

launch foo(a,b,c)

foo finishes

```
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
  // some foo code
}

int main() {
  // some main code
  thread thread_handle (foo,1,2,3);
  // code here runs concurrently with foo
  thread_handle.join();
  return 0;
}
```

header and namespace



main waits for foo.
called **join()**

join() returns in main

launch foo(a,b,c)

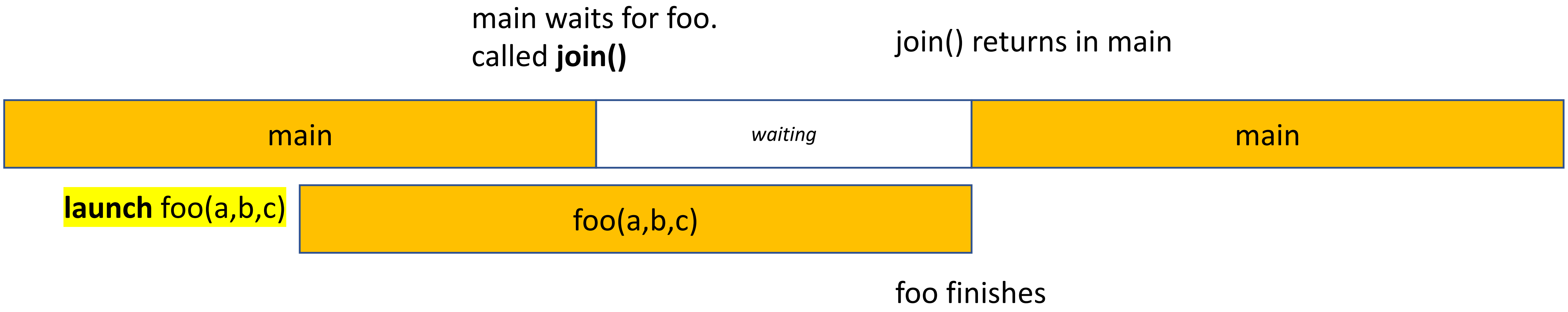
foo finishes

```
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
  // some foo code
}

int main() {
  // some main code
  thread thread_handle (foo,1,2,3);
  // code here runs concurrently with foo
  thread_handle.join();
  return 0;
}
```

Launches a concurrent thread that executes foo

Stores a handle in thread_handle (don’t lose the handle!)

constructor takes in the function, and all arguments



main waits for foo.
called **join()**

join() returns in main

main

waiting

main

launch foo(a,b,c)

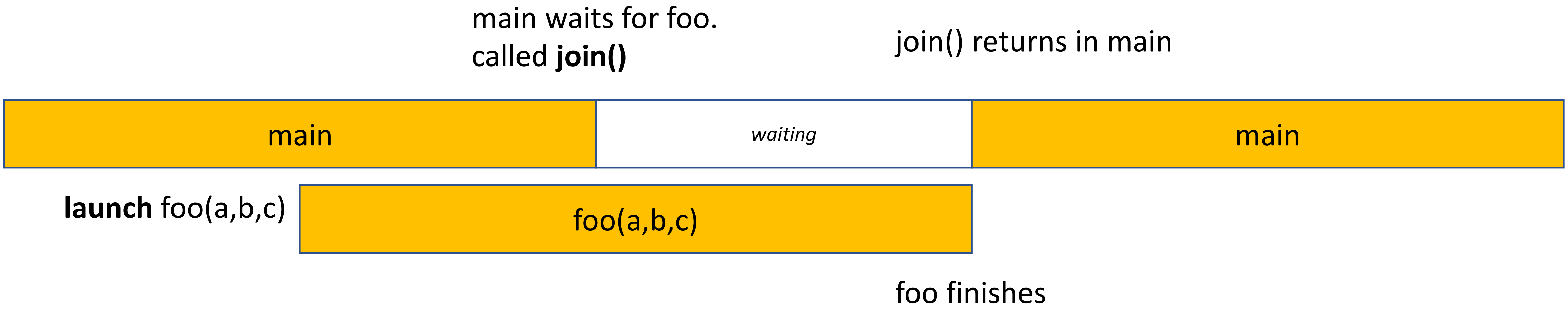
foo(a,b,c)

foo finishes

```
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
  // some foo code
}

int main() {
  // some main code
  thread thread_handle (foo,1,2,3);
  // code here runs concurrently with foo
  thread_handle.join();
  return 0;
}
```

Requires C++14

clang++ -std=c++14 main.cpp



main waits for foo.
called **join()**

join() returns in main

main

waiting

main

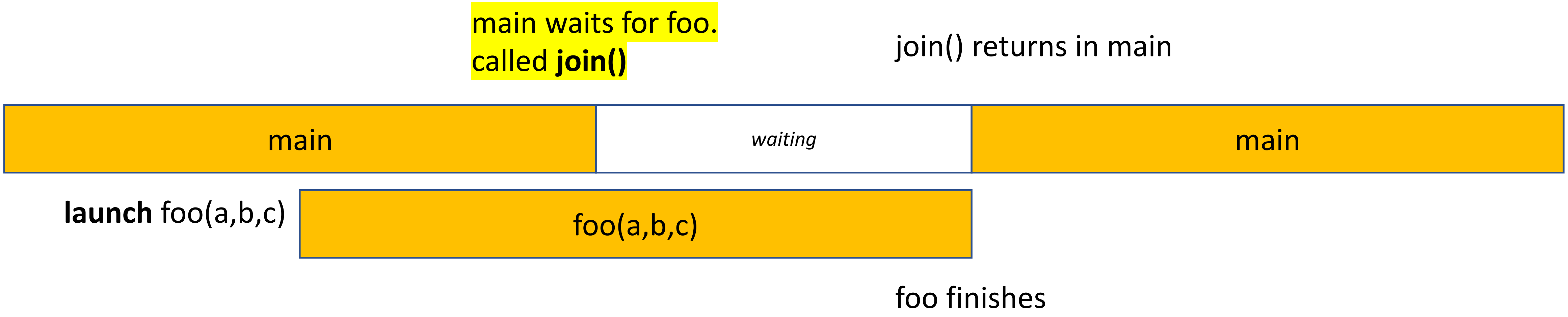
launch foo(a,b,c)

foo(a,b,c)

foo finishes

```
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
  // some foo code
}

int main() {
  // some main code
  thread thread_handle (foo,1,2,3);
  // code here runs concurrently with foo
  thread_handle.join();
  return 0;
}
```

calling join() on the thread handle will cause main to wait for the thread launched with thread_handle to finish.

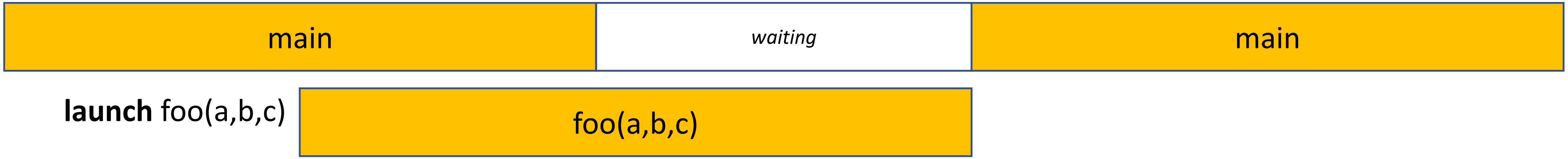


```
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
  // some foo code
}

int main() {
  // some main code
  thread thread_handle (foo,1,2,3);
  // code here runs concurrently with foo
  thread_handle.join();
  return 0;
}
```

After foo finishes,
main starts executing again

main waits for foo.
called **join()**

join() returns in main



main

waiting

main

launch foo(a,b,c)

foo(a,b,c)

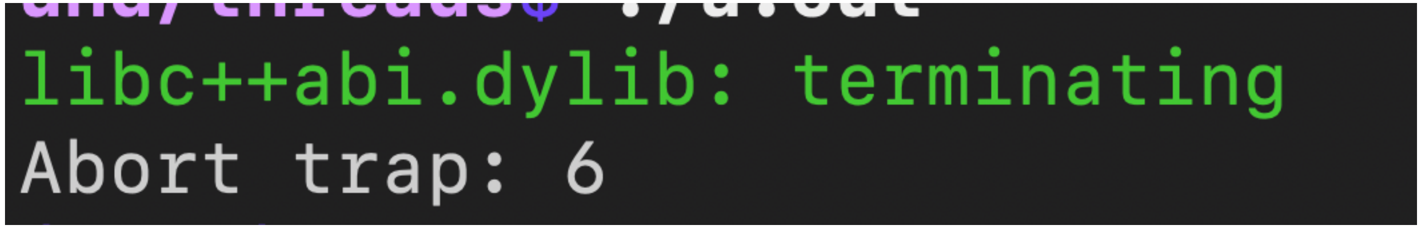
foo finishes

```
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
  // some foo code
}

int main() {
  // some main code
  thread thread_handle (foo,1,2,3);
  // code here runs concurrently with foo
  ~~thread_handle.join();~~
  return 0;
}
```

What happens if you don’t join your threads?

```
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
   // some foo code
}

int main() {
   // some main code
   thread thread_handle (foo,1,2,3);
   // code here runs concurrently with foo
   ~~thread_handle.join();~~
   return 0;
}
```

What happens if you don’t join your threads?



```
and/threads$ ./a.out
libc++abi.dylib: terminating
Abort trap: 6
```

JOIN YOUR THREADS!!!

```
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
  // some foo code
}

int main() {
  // some main code
  thread thread_handle (foo,1,2,3);
  // code here runs concurrently with foo
  thread_handle.join();
  return 0;
}
```

return value?

Doesn’t have to be void,
but it is ignored

how to get values back
from threads?

```
#include <thread>
#include <iostream>
using namespace std;

void foo(int a, int b, int &c) {
  // return a + b;
  c = a + b;
}

int main() {
  // some main code
  int ret = 0;
  thread thread_handle (foo,1,2, ref(ret));
  // code here runs concurrently with foo
  thread_handle.join();
  cout << ret << endl;
  return 0;
}
```

Options

pass by reference (C++)

```
#include <thread>
#include <iostream>
using namespace std;

void foo(int a, int b, int *c) {
  // return a + b;
  *c = a + b;
}

int main() {
  // some main code
  int ret = 0;
  thread thread_handle (foo,1,2, &ret);
  // code here runs concurrently with foo
  thread_handle.join();
  cout << ret << endl;
  return 0;
}
```

Options

pass by address (C++ or C)

```
#include <thread>
#include <iostream>
using namespace std;

int c;
void foo(int a, int b) {
  // return a + b;
  c = a + b;
}

int main() {
  // some main code
  ~~int ret = 0;~~
  thread thread_handle (foo,1,2);
  // code here runs concurrently with foo
  thread_handle.join();
  cout << c << endl;
  return 0;
}
```

Options

global variable
(don’t do this!)

```
#include <thread>
#include <iostream>
using namespace std;

void foo(int a, int b, int *c) {
  // return a + b;
  *c = a + b;
}

int main() {
  // some main code
  int ret = 0;
  thread thread_handle (foo,1,2, &ret);
  // code here runs concurrently with foo
  cout << ret << endl;
  thread_handle.join();
  return 0;
}
```

What if....

```
#include <thread>
#include <iostream>
using namespace std;

void foo(int a, int b, int *c) {
  // return a + b;
  *c = a + b;
}

int main() {
  // some main code
  int ret = 0;
  thread thread_handle (foo,1,2, &ret);
  // code here runs concurrently with foo
  cout << ret << endl;
  thread_handle.join();
  return 0;
}
```

What if....

Undefined behavior!
Cannot access the same values concurrently without protection!

Next module we will talk protection (locks)

```
#include <thread>
#include <iostream>
using namespace std;

void foo(int a, int b, int *c) {
  // return a + b;
  *c = a + b;
}

int main() {
  // some main code
  int ret = 0;
  thread thread_handle (foo,1,2, &ret);
  // code here runs concurrently with foo
  cout << ret << endl;
  thread_handle.join();
  return 0;
}
```

What if....

Undefined behavior! Cannot access the same values concurrently without protection!

Next module we will talk protection (locks)

SPMD programming model

- Same program, multiple data

- Main idea: many threads execute the same function, but they operate on different data.

- How do they get different data?
  - each thread can access their own thread id, a contiguous integer starting at 0 up to the number of threads

SPMD programming model

```
void increment_array(int *a, int a_size) {
   for (int i = 0; i < a_size; i++) {
       a[i]++;
   }
}
```

lets do this in parallel!

each thread increments different
elements in the array

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
   for (int i = 0; i < a_size; i++) {
       a[i]++;
   }
}
```

The function gets a thread id and the number of threads

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
   for (int i = 0; i < a_size; i++) {
       a[i]++;
   }
}
```

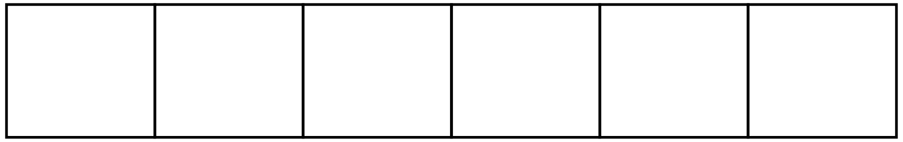
A few options on how to split up the work
lets do round robin

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
   for (int i = tid; i < a_size; i+=num_threads) {
       a[i]++;
   }
}
```

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
    for (int i = tid; i < a_size; i+=num_threads) {
        a[i]++;
    }
}
```



array a

Assume 2 threads

lets step through thread 0

i.e.

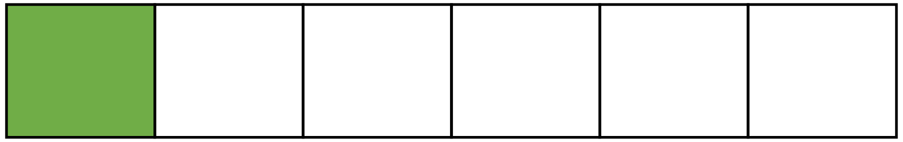
tid = 0

num_threads = 2

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
   for (int i = tid; i < a_size; i+=num_threads) {
       a[i]++;
   }
}
```

iteration 1 computes index 0



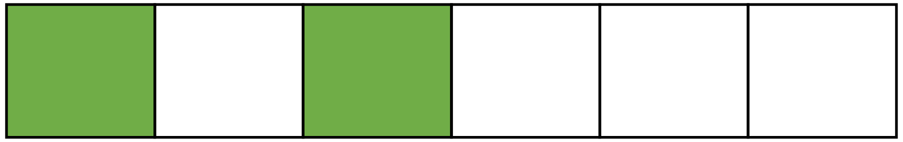
array a

Assume 2 threads
lets step through thread 0
i.e.
tid = 0
num_threads = 2

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
   for (int i = tid; i < a_size; i+=num_threads) {
       a[i]++;
   }
}
```

iteration 2 computes index 2



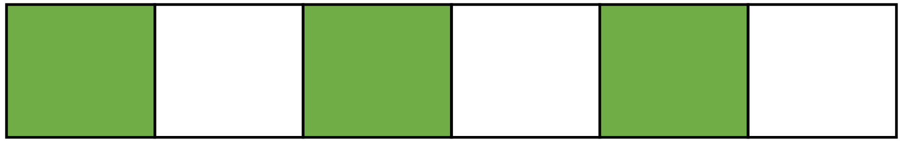
array a

Assume 2 threads
lets step through thread 0
i.e.
tid = 0
num_threads = 2

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
   for (int i = tid; i < a_size; i+=num_threads) {
       a[i]++;
   }
}
```

iteration 3 computes index 4



array a

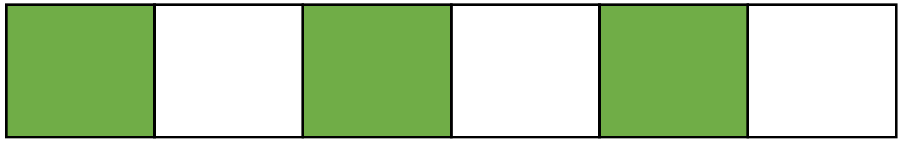
Assume 2 threads
lets step through thread 0
i.e.
tid = 0
num_threads = 2

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
   for (int i = tid; i < a_size; i+=num_threads) {
       a[i]++;
   }
}
```

switch to thread 1

Assume 2 threads
lets step through thread 1
i.e.
tid = 1
num_threads = 2



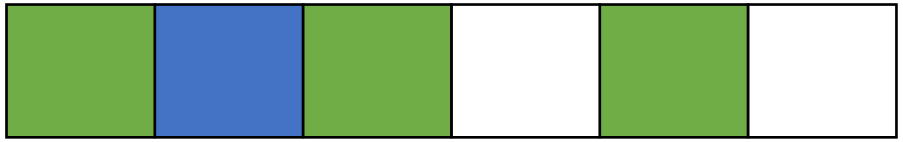
array a

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
   for (int i = tid; i < a_size; i+=num_threads) {
       a[i]++;
   }
}
```

switch to thread 1

iteration 1 computes index 1



array a

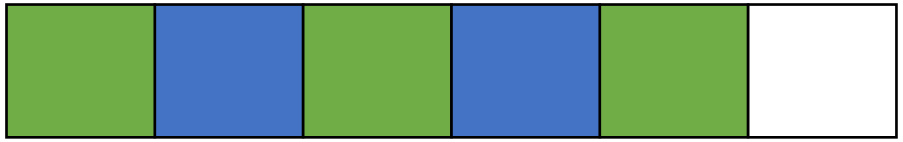
Assume 2 threads
lets step through thread 1
i.e.
tid = 1
num_threads = 2

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
   for (int i = tid; i < a_size; i+=num_threads) {
       a[i]++;
   }
}
```

switch to thread 1

iteration 2 computes index 3



array a

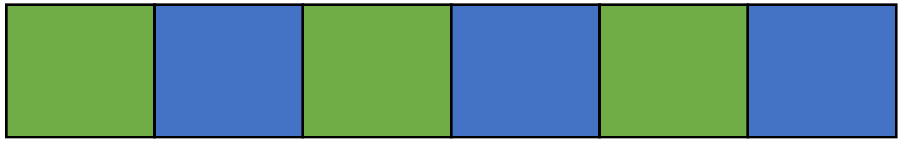
Assume 2 threads
lets step through thread 1
i.e.
tid = 1
num_threads = 2

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
   for (int i = tid; i < a_size; i+=num_threads) {
       a[i]++;
   }
}
```

switch to thread 1

iteration 3 computes index 5



array a

Assume 2 threads
lets step through thread 1
i.e.
tid = 1
num_threads = 2

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads);


#define THREADS 8
#define A_SIZE 1024
int main() {
  int *a = new int[A_SIZE];
  // initialize a
  thread thread_ar[THREADS];
  for (int i = 0; i < THREADS; i++) {
    thread_ar[i] = thread(increment_array, a, A_SIZE, i, THREADS);
  }
  for (int i = 0; i < THREADS; i++) {
    thread_ar[i].join();
  }
  delete[] a;
  return 0;
}
```

See you all on Tuesday!

- Finished up Module 1
  - Hardware and compiler overview
- Next module begins on Tuesday
  - Mutual Exclusion
- Homework posted by tomorrow midnight
- Office hours on Friday