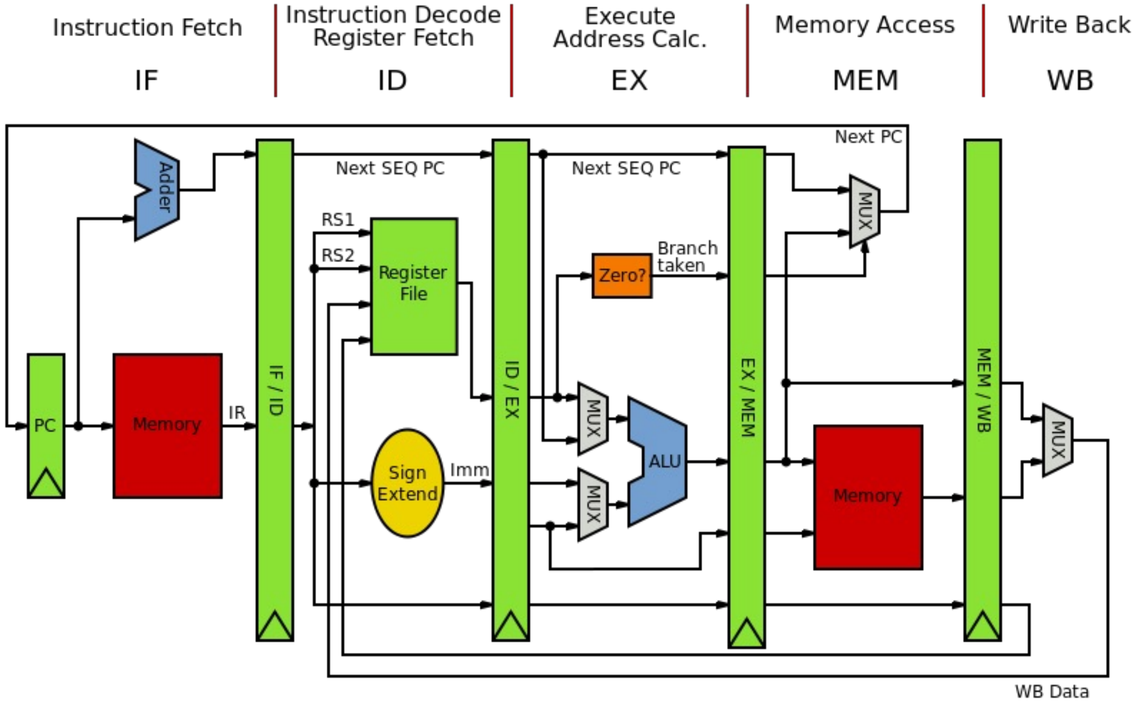
CSE211: Compiler Design

Oct. 27, 2022

- **Topic**: instruction-level parallelism (ILP)
  - dependency graphs/chains
  - loop unrolling
  - reductions

- **Discussion questions**:
  - What is instruction level parallelism?
  - How can modern processors exploit ILP?



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

Announcements

- Remote lecture today
  - Feeling better but testing positive ☹
  - Will be back next week. That will complete the 2 weeks of isolation
- Homework 2 is out
  - due Nov. 2
  - if you don’t have a partner let me know ASAP
  - please get started if you haven’t already
  - great discussion on Piazza about local value numbering
  - ask if you have any more questions!

Announcements

- Midterm
    - released tomorrow by 8 AM
    - due the next Friday (Nov. 4) by midnight
    - Rules:
        - Open book, open internet, open notes.
        - Do not discuss the test with any other student while it is out.
        - Do not google (or otherwise search) for exact questions. It is fine to search for concepts.
        - Do not post questions to others on the internet (e.g. through discord or reddit)
        - Any question should be asked as a private post on Piazza. If it's a clarification that needs to be made to the whole class, I will do it in a public Piazza thread

Announcements

- Midterm
    - Designed to take about 2 hours (not including studying)
    - Students report taking longer because they study while taking the test.
    - Students also report taking longer because they double check their answers and make the test nicely formatted.
    - Please look over the test as soon as it is released so that you roughly know how long it will take you.

    - LATE MIDTERMS WILL NOT BE ACCEPTED

Announcements

- Mark your attendance for today after you watch the recording (or if you are attending live)
  - Please try to keep on top of this.
  - We have more attendance put in, please let us know within 1 week if there are any issues

Review SSA optimizations

ϕ instructions

- Example: how to convert this code into SSA?

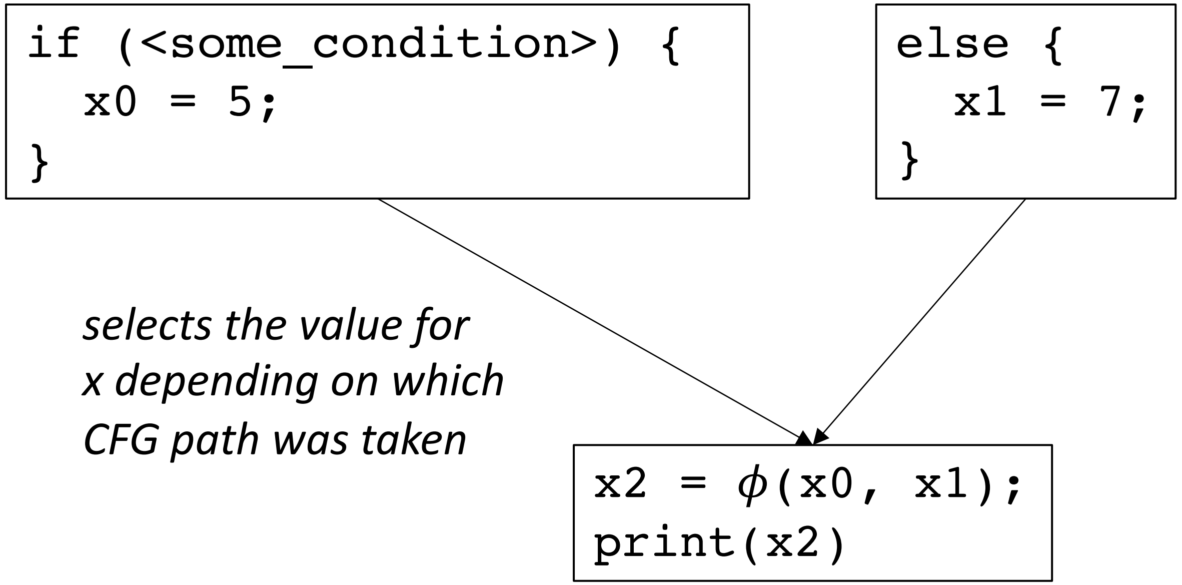
```
int x;

if (<some_condition>) {
  x = 5;
}

else {
  x = 7;
}

print(x)
```

ϕ instructions

- Example: how to convert this code into SSA?

```
int x;

if (<some_condition>) {
  x0 = 5;
}

else {
  x1 = 7;
}

x2 = ϕ(x0, x1);
print(x2)
```

number the variables



```
if (<some_condition>) {
  x0 = 5;
}
```

```
else {
  x1 = 7;
}
```

```
x2 = ϕ(x0, x1);
print(x2)
```

selects the value for x depending on which CFG path was taken

Converting to SSA

- Really Crude Approach
    - Every variable in every basic block has a phi node

- Maximal SSA
    - Every variable in every JOIN node in the cfg has a phi node

- Semi-pruned SSA
    - Computes dominance frontier
    - Variables assigned in block B need a phi node in the dominance frontier of B

Constant Propagation using SSA

```
x = 42;
y = x + 5;
```

single block can be optimized using local value numbering

```
y = 47;
```

What about across basic blocks?

```
x = 42;
z = 5;
if (<some condition> {
   y = 5;
}
else {
   y = z;
}
w = y;
```

A simple lattice

- A set of symbols: $\{c_1, c_2, c_3 ...\}$
- Special symbols:
  - Top : T
  - Bottom : ⊥
- Meet operator: ∧

Lattices are an abstract algebra construct, with a few properties:

$\bot \wedge x = \bot$
$T \wedge x = x$
Where x is any symbol

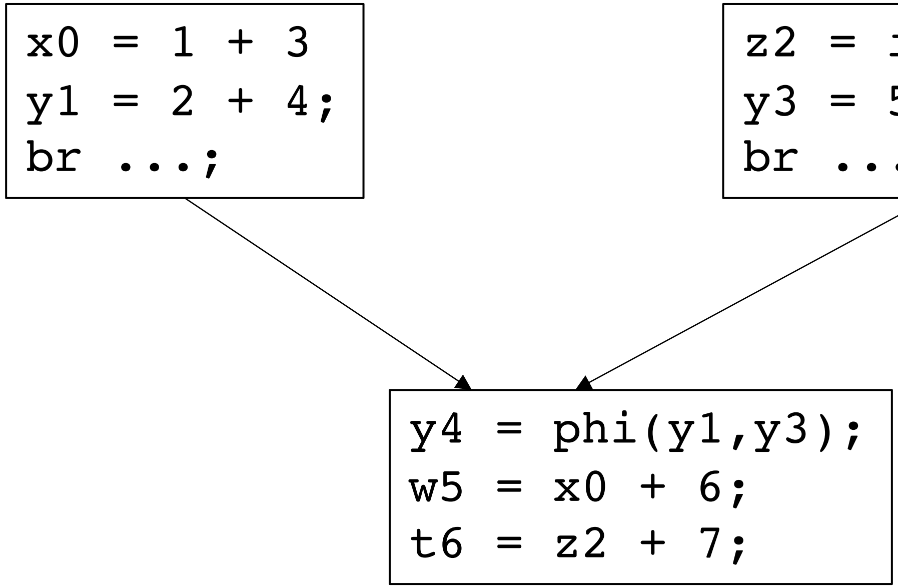
For constant propagation:

take the symbols to be integers

Simple meet operations for integers:

```
if $c_i$ != $c_j$ :
   $c_i \wedge c_j = \bot$

else:
$c_i \wedge c_j = c_j$
```

Example:



```
x0 = 1 + 3
y1 = 2 + 4;
br ...;
```

```
z2 = input();
y3 = 5 + 1;
br ...;
```

```
y4 = phi(y1,y3);
w5 = x0 + 6;
t6 = z2 + 7;
```

```
Value {
  x0 : 4
  y1 : 6
  z2 : B
  y3 : 6
  y4 : T
  w5 : 10
  t6 : T
}
```

```
Uses {
  x0 : [w5]
  y1 : [y4]
  z2 : [y3, t6]
  y3 : [y4]
  y4 : []
  w5 : []
  t6 : []
}
```

Worklist: `[x0,z2,y1,y3]`

```
B0: i0 = ...;

B1: a0 = φ(...);
    b1 = φ(...);
    c2 = φ(...);
    d3 = φ(...);
    a5 = ...;
    c6 = ...;
    br ... B2, B5;

B2: b7 = ...;
    c8 = ...;
    d9 = ...;

B3: a10 = φ(...);
    b11 = φ(...);
    c12 = φ(...);
    d13 = φ(...);
    y14 = ...;
    z15 = ...;
    i16 = ...;
    br ... B1, B4;
```

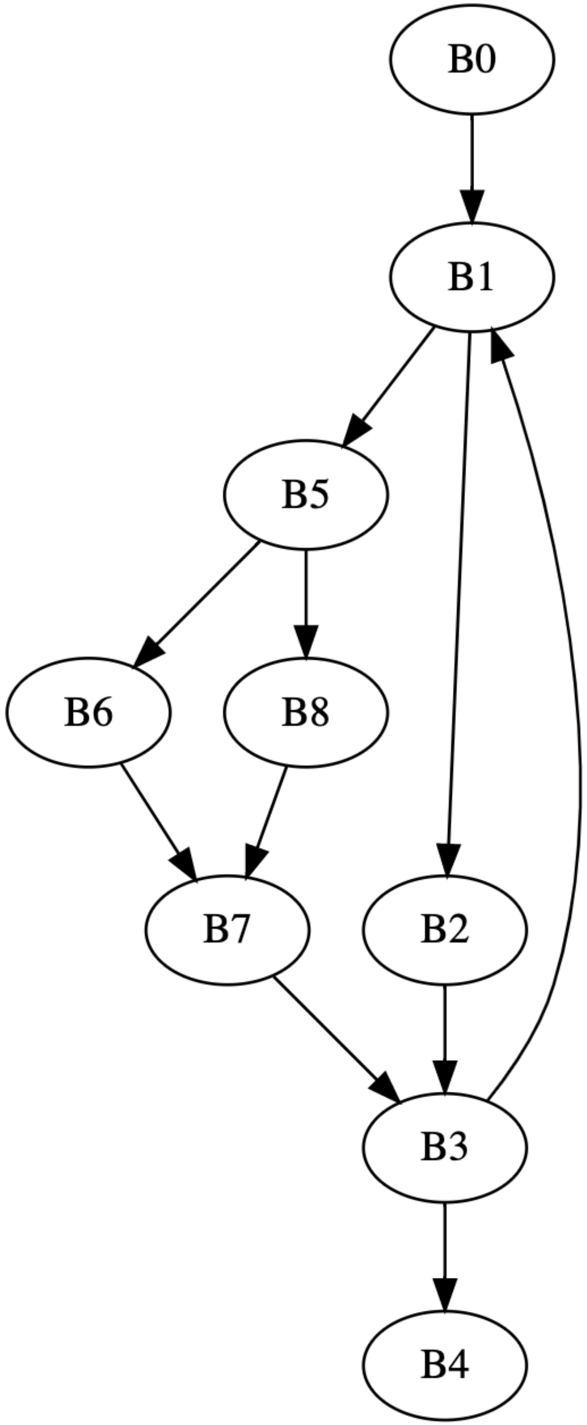
```
B4: return

B5: a17 = ...;
    d18 = ...;
    br ... B6, B8;

B6: d19 = ...;

B7: d20 = φ(...);
    c21 = φ(...);
    b22 = ...;

B8: c23 = ...;
    br B7;
```

Converting out of SSA

Two approaches:

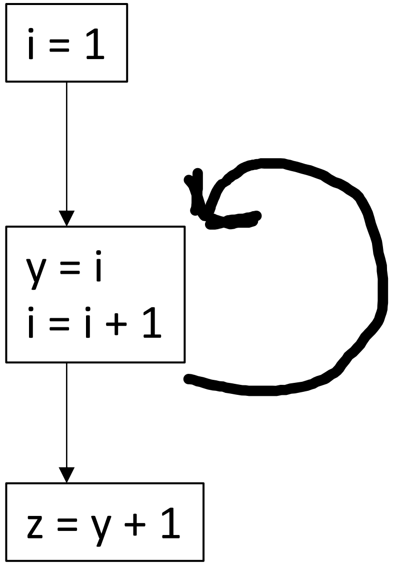
1. path tracking and conditionals
2. early assignment

Example using i in B1



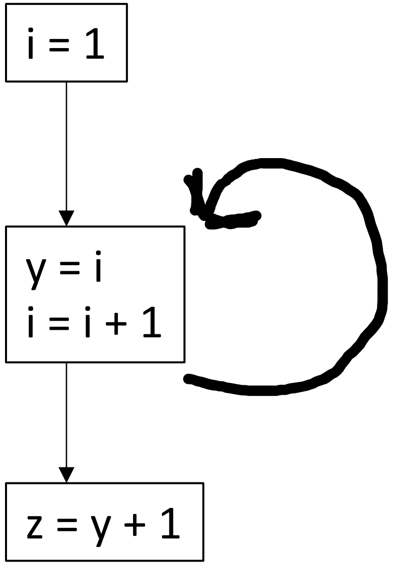
Lost copy issue

An issue with early assignment algorithm

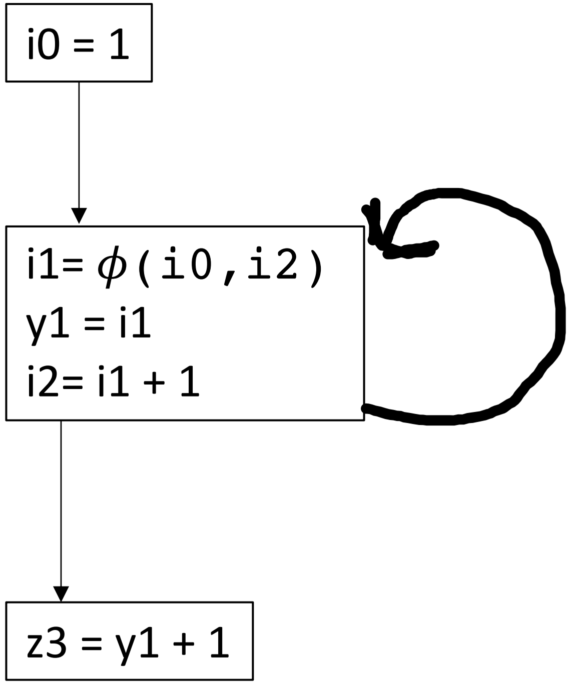


Example from https://www.clear.rice.edu/comp512/Lectures/13SSA-2.pdf

Lost copy issue

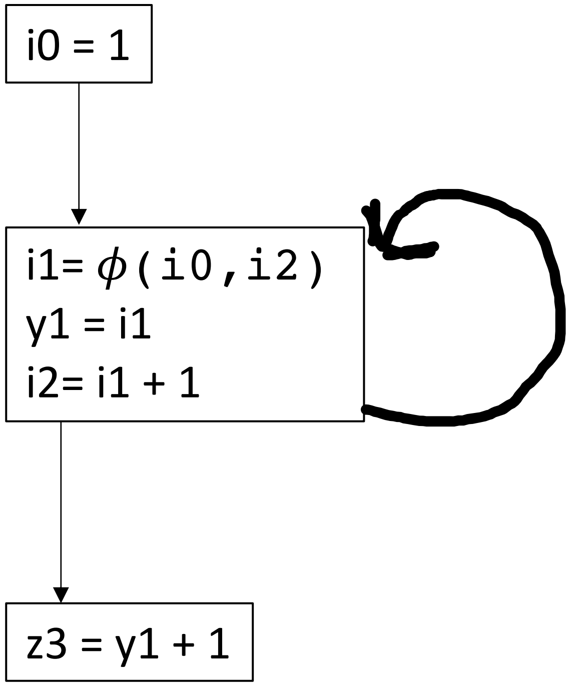


convert to SSA



Example from https://www.clear.rice.edu/comp512/Lectures/13SSA-2.pdf

Lost copy issue

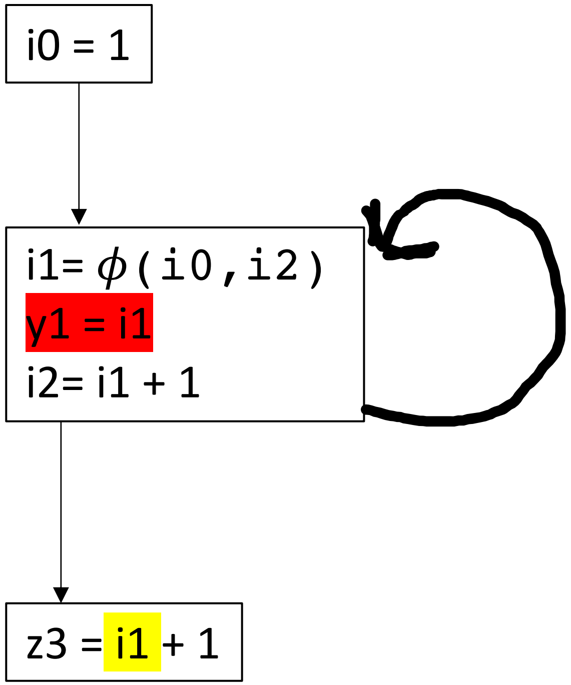


i0 = 1

i1= ϕ(i0,i2)
y1 = i1
i2= i1 + 1

z3 = y1 + 1

copy propagation



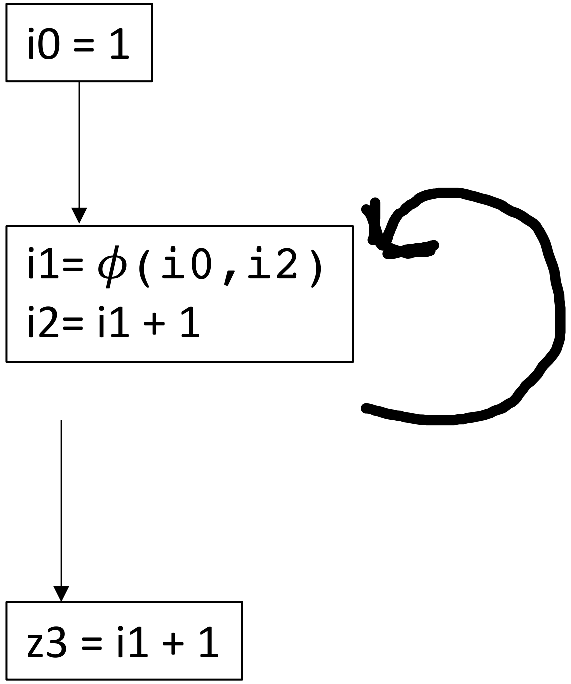
i0 = 1

i1= ϕ(i0,i2)
y1 = i1
i2= i1 + 1

z3 = i1 + 1

Example from https://www.clear.rice.edu/comp512/Lectures/13SSA-2.pdf

Lost copy issue



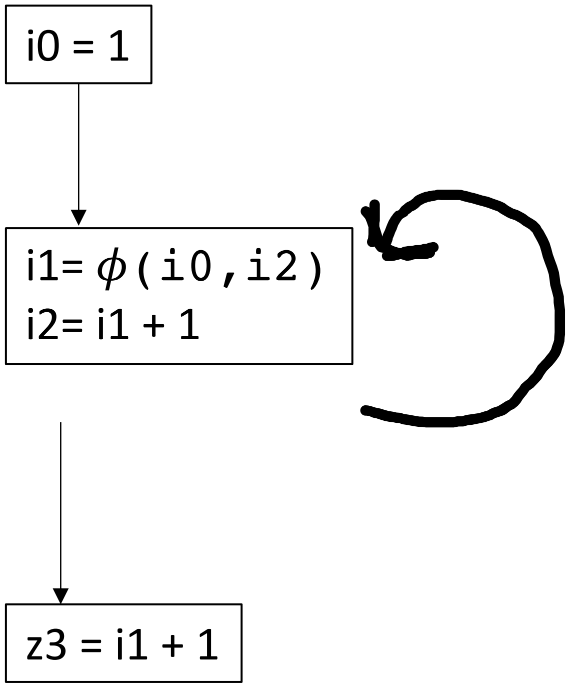
i0 = 1

i1= ϕ(i0,i2)
i2= i1 + 1

z3 = i1 + 1

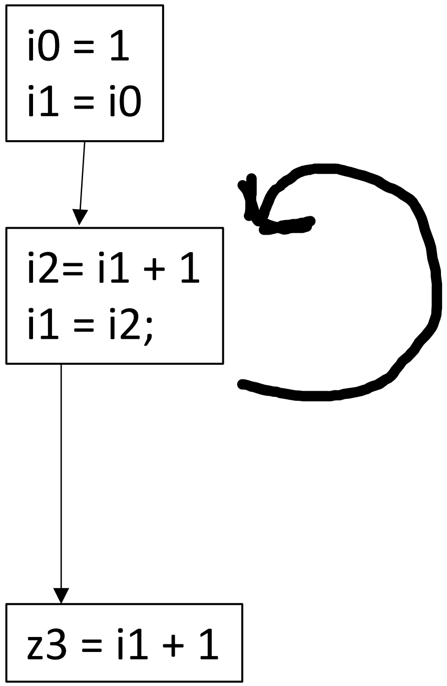
Example from https://www.clear.rice.edu/comp512/Lectures/13SSA-2.pdf

Lost copy issue



```
i0 = 1

i1= ϕ(i0,i2)
i2= i1 + 1

z3 = i1 + 1
```

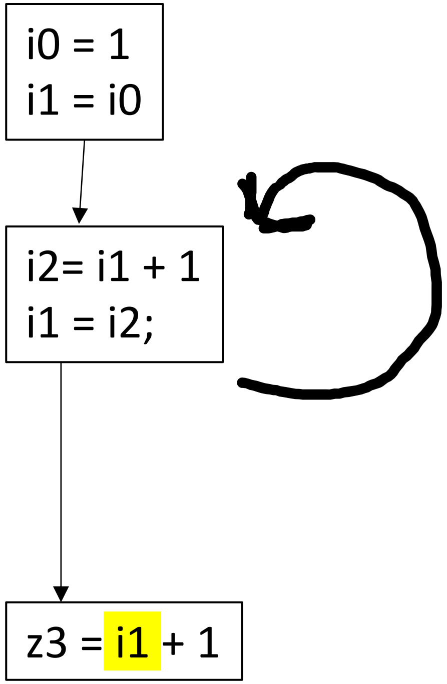
early
assignment



```
i0 = 1
i1 = i0

i2= i1 + 1
i1 = i2;

z3 = i1 + 1
```

Example from https://www.clear.rice.edu/comp512/Lectures/13SSA-2.pdf

Lost copy issue

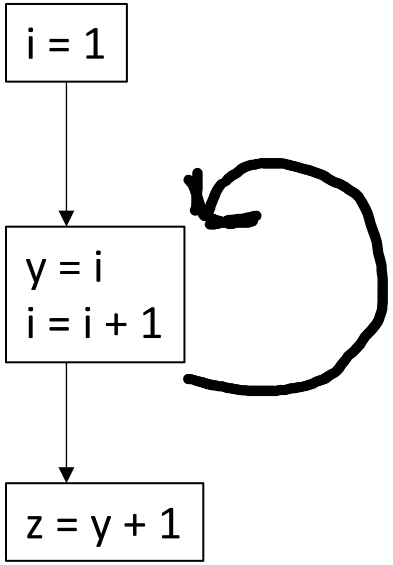


i0 = 1
i1 = i0

i2= i1 + 1
i1 = i2;

z3 = i1 + 1

are these 2 the same?



i = 1

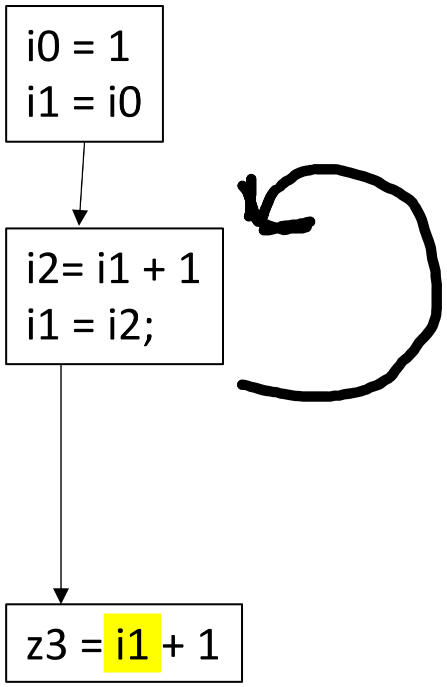
y = i
i = i + 1

z = y + 1

Example from https://www.clear.rice.edu/comp512/Lectures/13SSA-2.pdf

Lost copy issue

Known as the lost-copy problem
there are algorithms for handling this (see book)

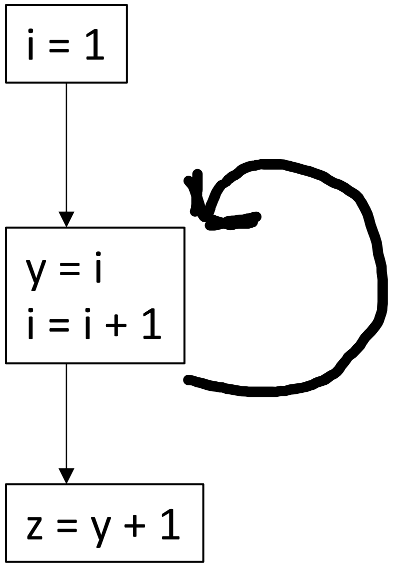


```
i0 = 1
i1 = i0
```

```
i2= i1 + 1
i1 = i2;
```

```
z3 = i1 + 1
```

are
these
2 the
same?



```
i = 1
```

```
y = i
i = i + 1
```

```
z = y + 1
```

Similar problem called the Swap problem

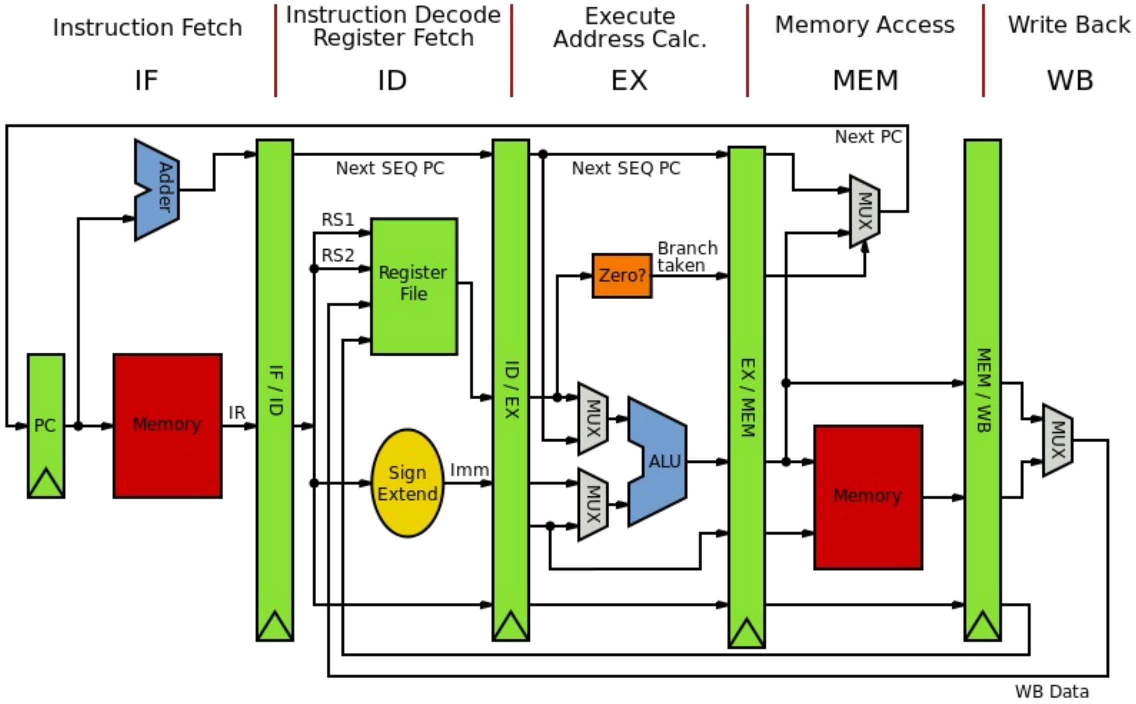
Example from https://www.clear.rice.edu/comp512/Lectures/13SSA-2.pdf

CSE211: Compiler Design

Oct. 27, 2022

- **Topic**: instruction-level parallelism (ILP)
  - dependency graphs/chains
  - loop unrolling
  - reductions

- **Discussion questions**:
  - What is instruction level parallelism?
  - How can modern processors exploit ILP?



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

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, its better to implement in a compiler

Finding dependencies in the compiler

- What type of instructions can be done in parallel?

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

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

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

Data dependency graphs (DDG)

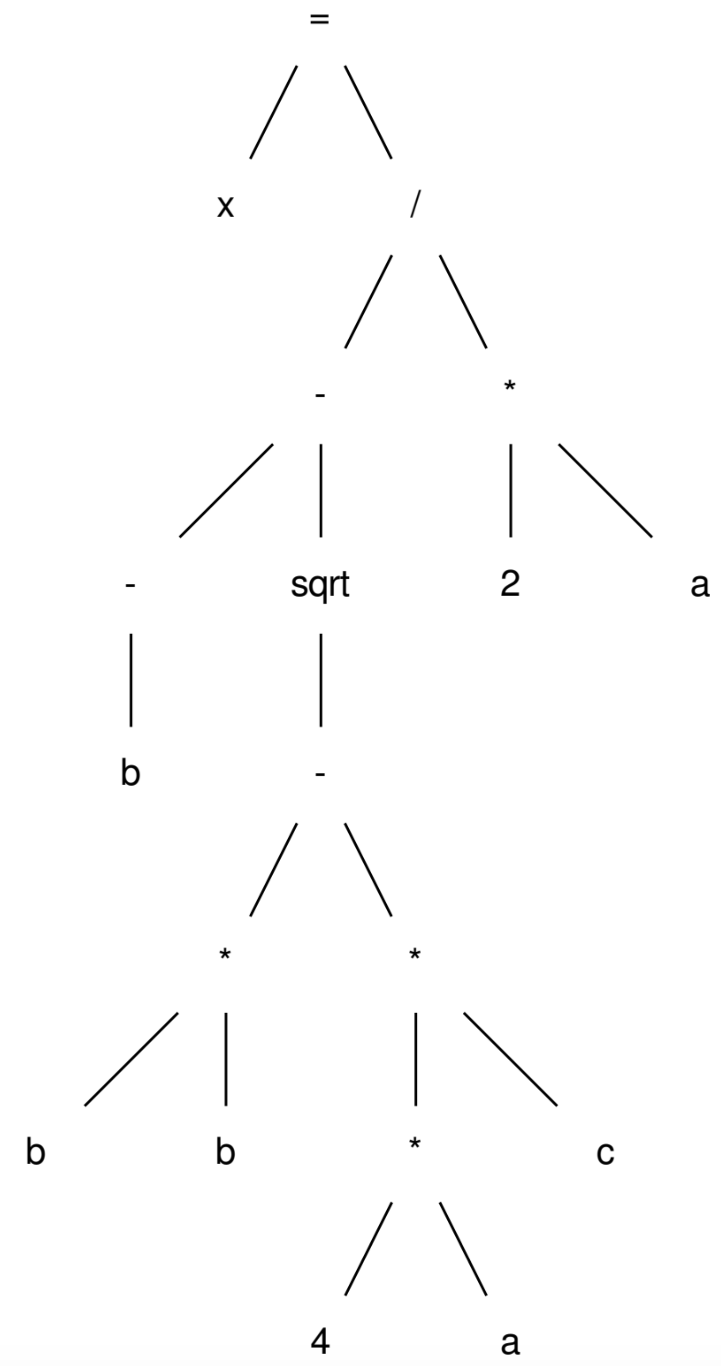
Quadratic formula

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

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

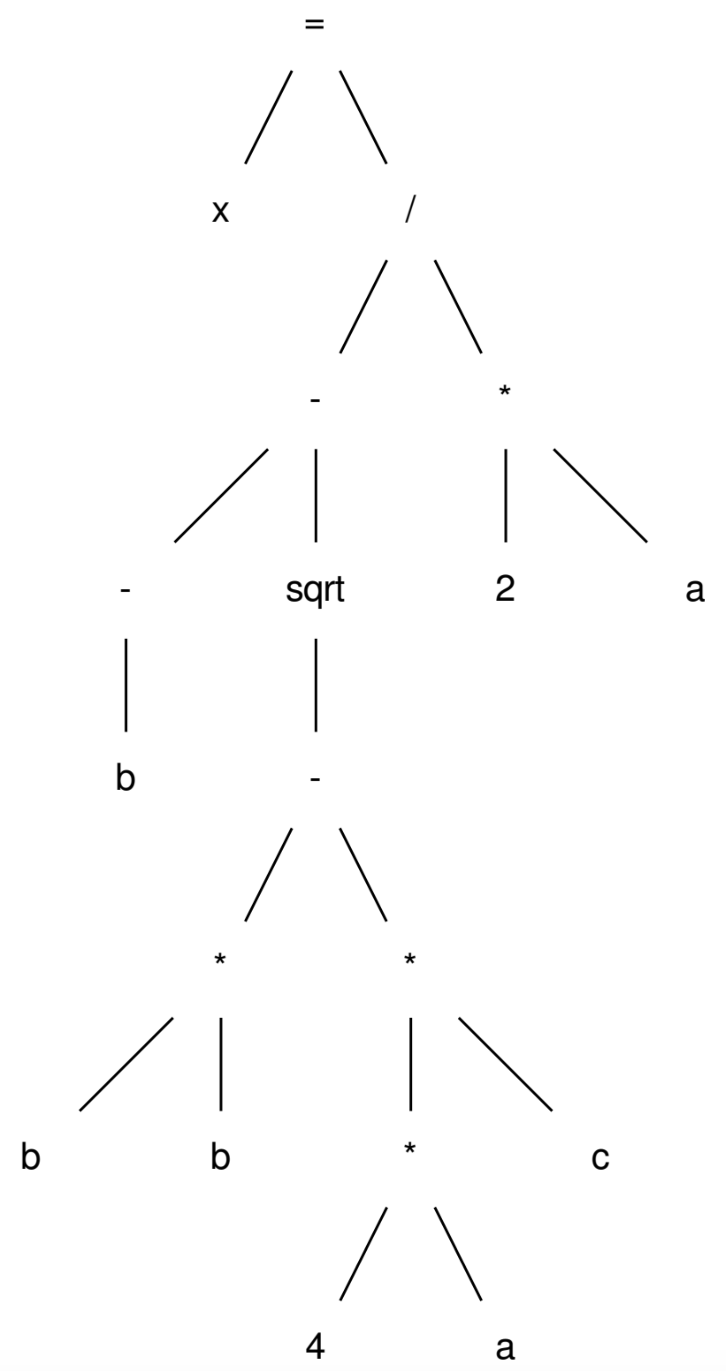
```
x = (-b - sqrt(b*b - 4 * a * c)) / (2*a)
```

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



post-order traversal, using temporary variables

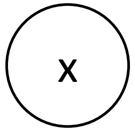
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
```



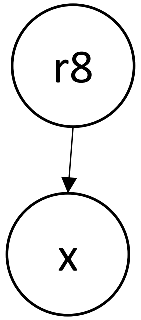
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```

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

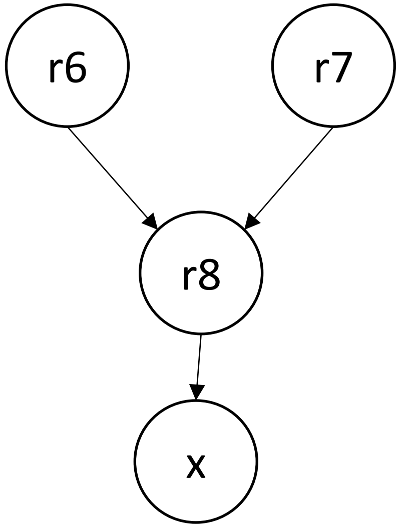
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



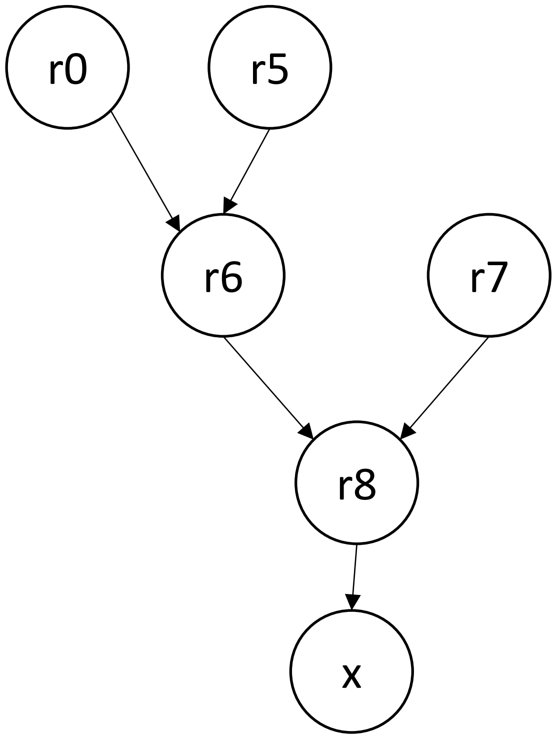
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



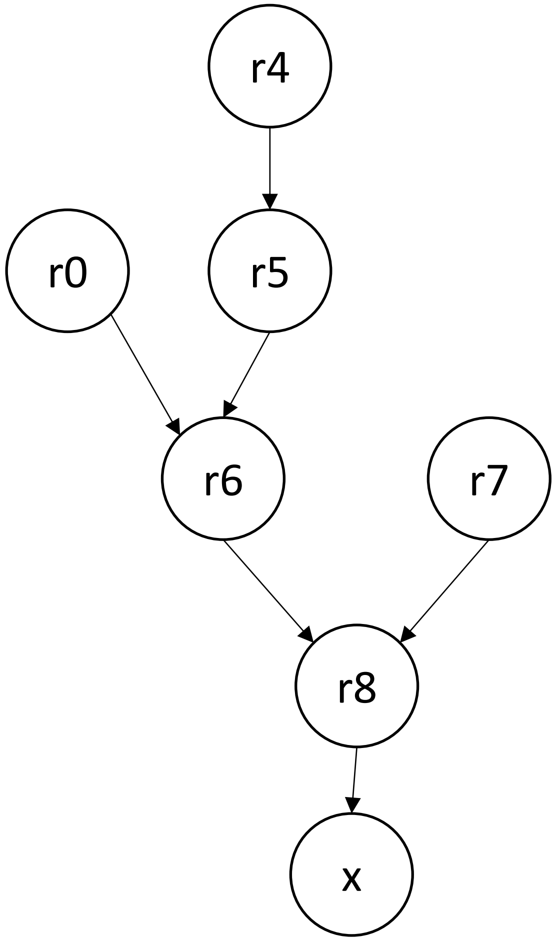
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



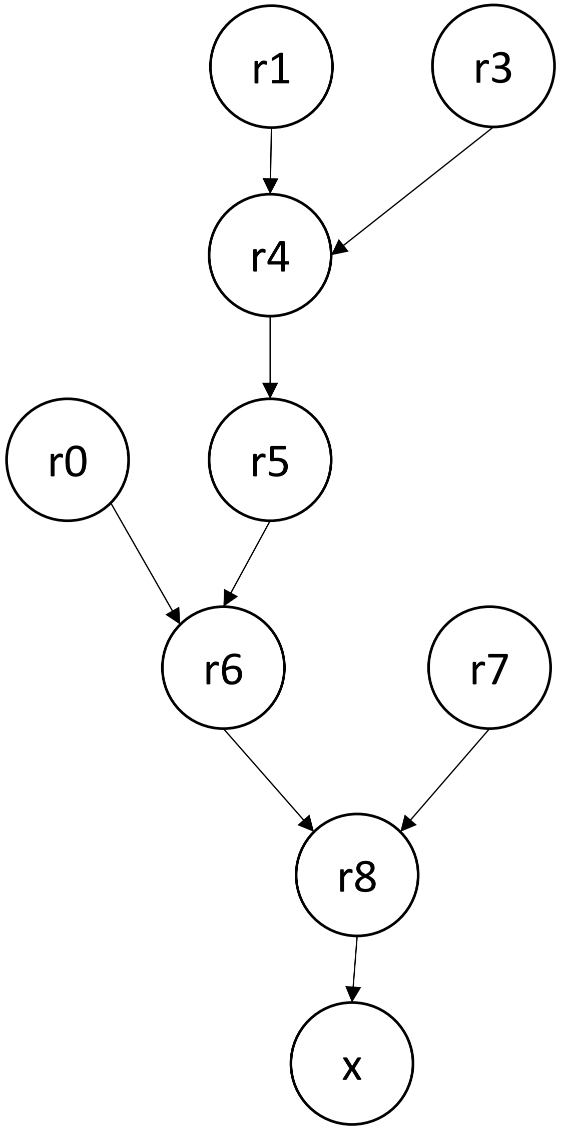
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



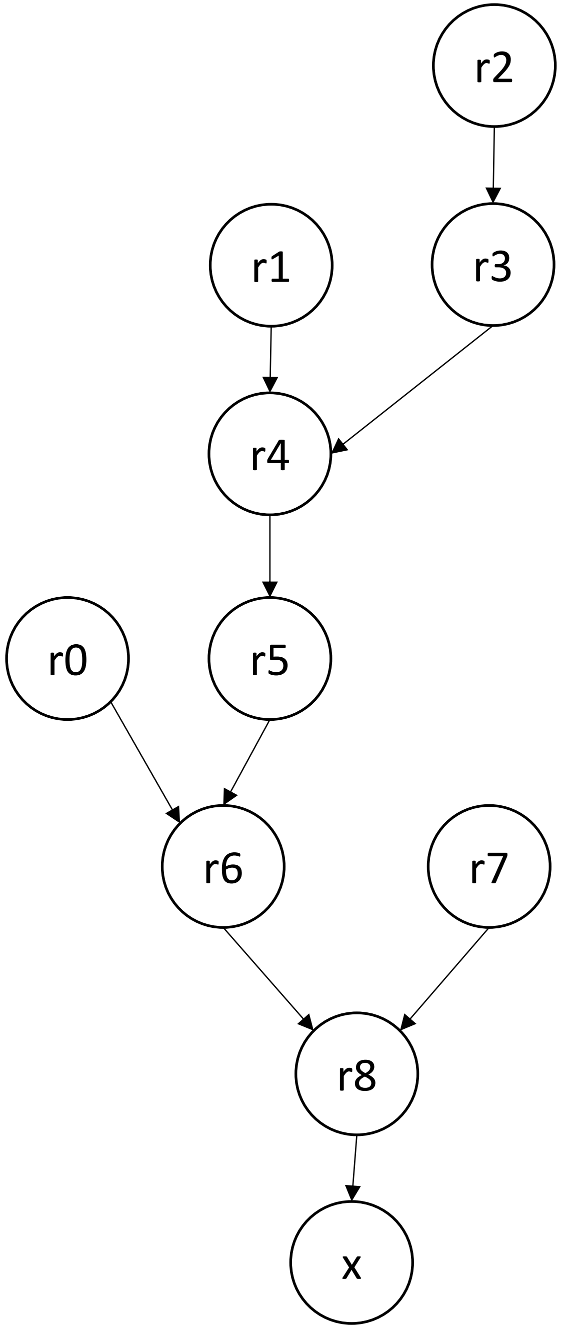
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



```
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;
```



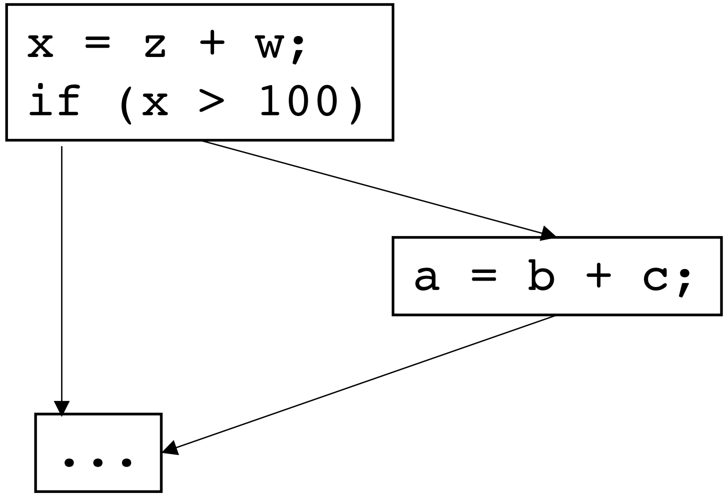
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



Control dependencies

```
x = z + w;
if (x > 100)
   a = b + c;
```

Instructions in different CFG nodes have control-dependencies



Memory dependencies

True dependence:
Read-after-write

```
a[i] = z + w;
x = a[i]
```

Memory dependencies

True dependence:
Read-after-write

```
a[i] = z + w;
x = a[i]
```

Output dependence:
Write-after-write

```
a[i] = z + w;
a[i] = a + b;
```

Memory dependencies

True dependence:
Read-after-write

```
a[i] = z + w;
x = a[i]
```

Output dependence:
Write-after-write

```
a[i] = z + w;
a[i] = a + b;
```

anti-dependence:
Write-after-read

```
x = a[i]
a[i] = z + w;
```

Memory dependencies

True dependence:
Read-after-write

```
a[i] = z + w;
x = a[i]
```

Output dependence:
Write-after-write

```
a[i] = z + w;
a[i] = a + b;
```

Dependencies can be removed

```
reg_a_i = z + w;
a[i] = a + b;
```

anti-dependence:
Write-after-read

```
x = a[i]
a[i] = z + w;
```

Dependencies can be delayed

```
x = a[i]
reg_a_i = z + w;
...
a[i] = reg_a_i;
```

Memory dependencies

True dependence:
Read-after-write

```
a[i] = z + w;
x = a[i]
```

Output dependence:
Write-after-write

```
a[i] = z + w;
a[i] = a + b;
```

Dependencies can be removed

```
reg_a_i = z + w;
a[i] = a + b;
```

Can we just remove this line?

anti-dependence:
Write-after-read

```
x = a[i]
a[i] = z + w;
```

Dependencies can be delayed

```
x = a[i]
reg_a_i = z + w;
...
a[i] = reg_a_i;
```

Memory dependencies

All of this depends on accurate pointer analysis!

True dependence:
Read-after-write

```
a[i] = z + w;
x = a[i]
```

Output dependence:
Write-after-write

```
a[i] = z + w;
a[i] = a + b;
```

Dependencies can be removed

```
reg_a_i = z + w;
a[i] = a + b;
```

anti-dependence:
Write-after-read

```
x = a[i]
a[i] = z + w;
```

Dependencies can be delayed

```
x = a[i]
reg_a_i = z + w;
...
a[i] = reg_a_i;
```

Memory dependencies

All of this depends on accurate pointer analysis!

True dependence:
Read-after-write

```
a[i] = z + w;
x = a[i]
```

Output dependence:
Write-after-write

```
a[i] = z + w;
a[j] = a + b;
```

Dependencies can be removed

```
reg_a_i = z + w;
a[i] = a + b;
```

anti-dependence:
Write-after-read

```
x = a[i]
a[i] = z + w;
```

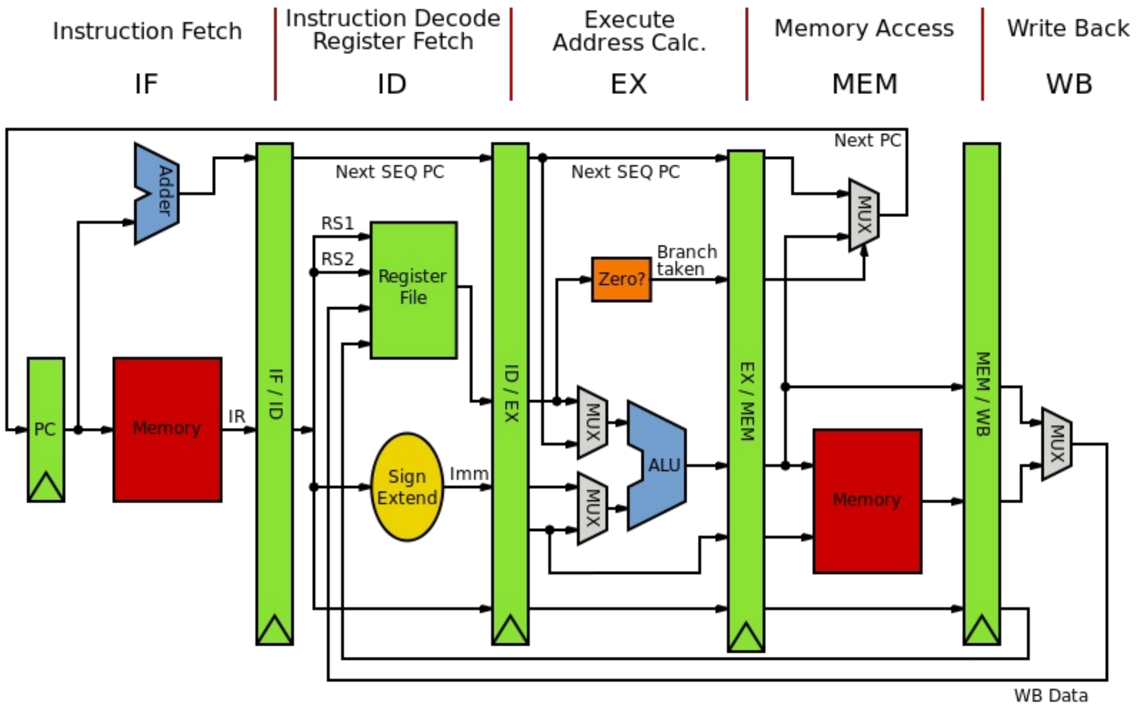
Dependencies can be delayed

```
x = a[i]
reg_a_i = z + w;
...
a[i] = reg_a_i;
```

How can hardware execute ILP?

- Pipeline parallelism

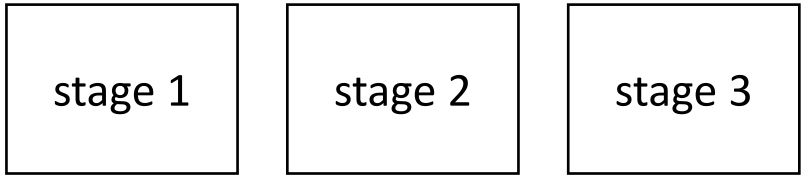
- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline



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

How can hardware execute ILP?

- Pipeline parallelism



stage 1

stage 2

stage 3

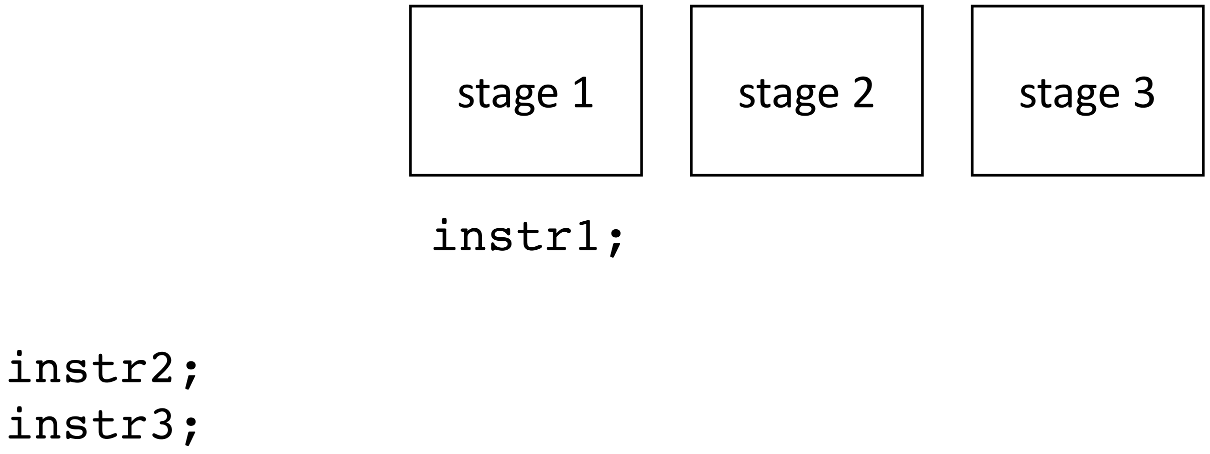
- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline

```
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



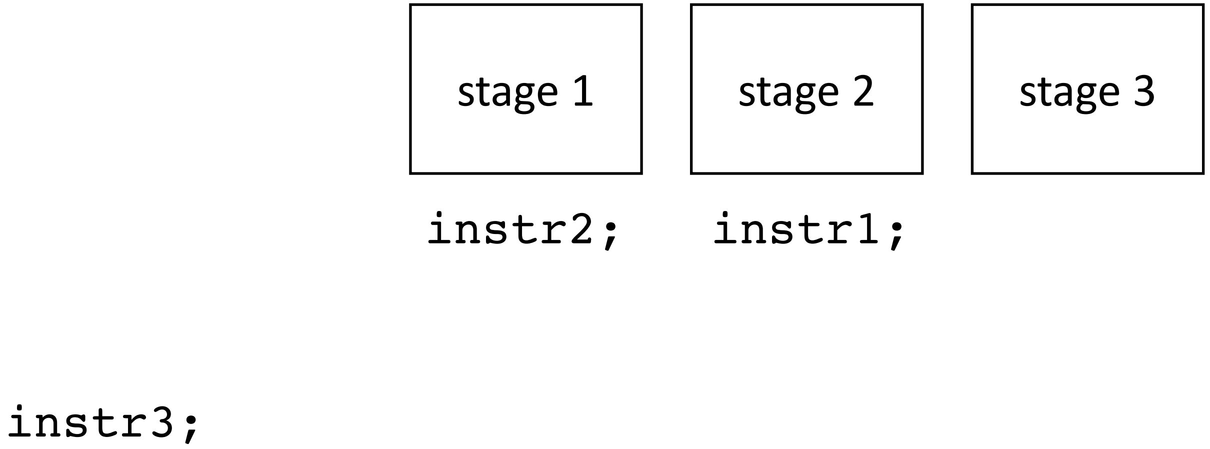
```
instr1;
```

```
instr2;
instr3;
```

How can hardware execute ILP?

- Pipeline parallelism

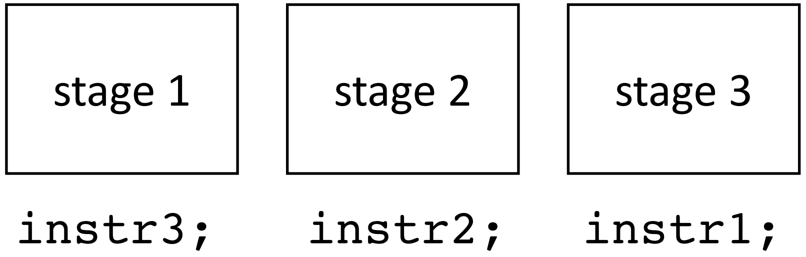
- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline



How can hardware execute ILP?

- Pipeline parallelism

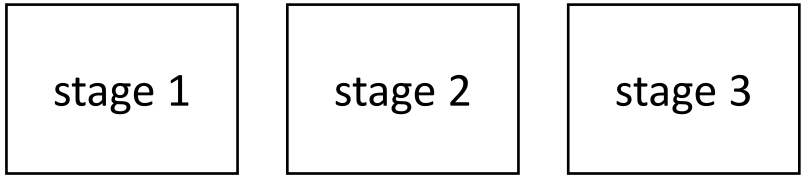
- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline



How can hardware execute ILP?

- Pipeline parallelism

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



stage 1

stage 2

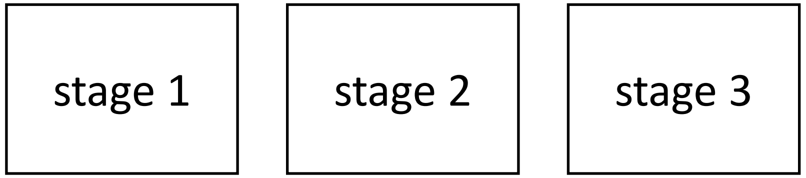
stage 3

6 cycles for 3 independent instructions

Converges to 1 instruction per cycle

How can hardware execute ILP?

- Pipeline parallelism



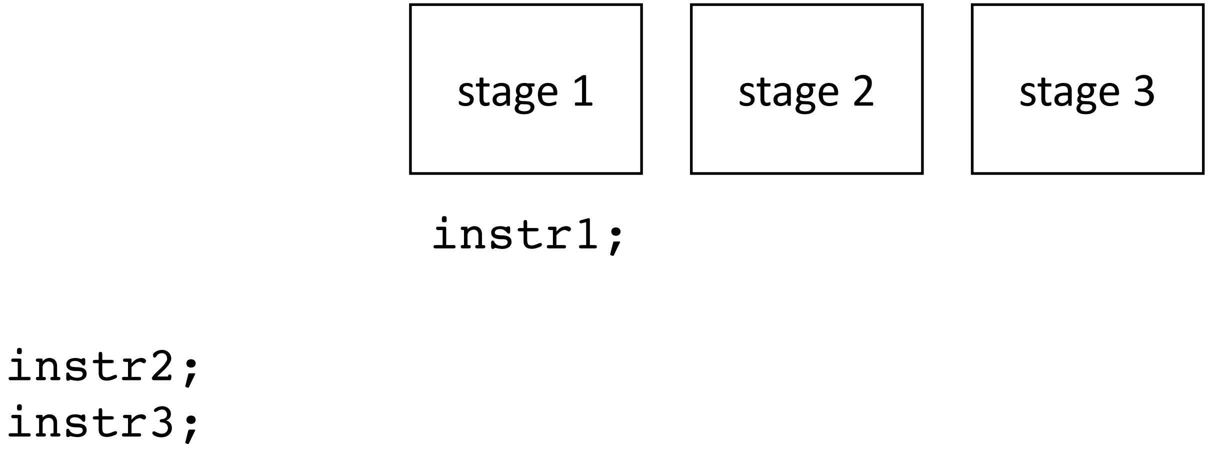
- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline

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

What if the instructions depend on each other?

How can hardware execute ILP?

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

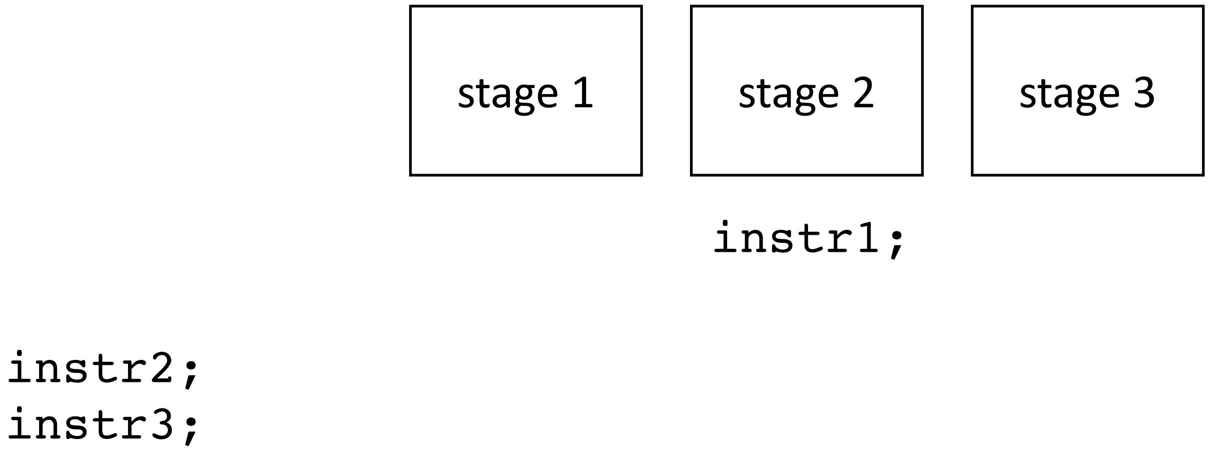


What if the instructions depend on each other?

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;
```

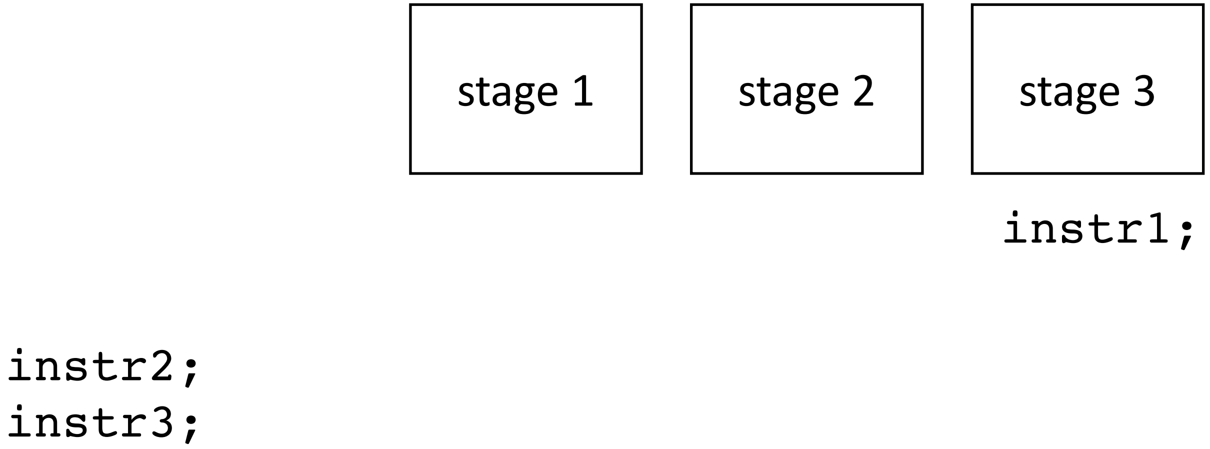
```
instr2;
instr3;
```

What if the instructions depend on each other?

How can hardware execute ILP?

- 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

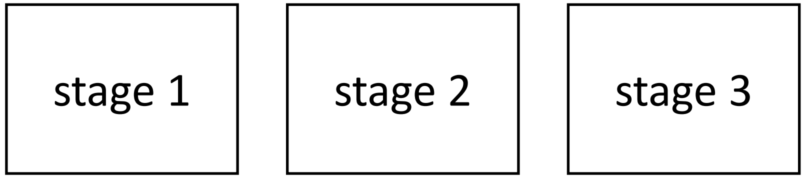
`instr1;`

```
instr2;
instr3;
```

What if the instructions depend on each other?

How can hardware execute ILP?

- Pipeline parallelism



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

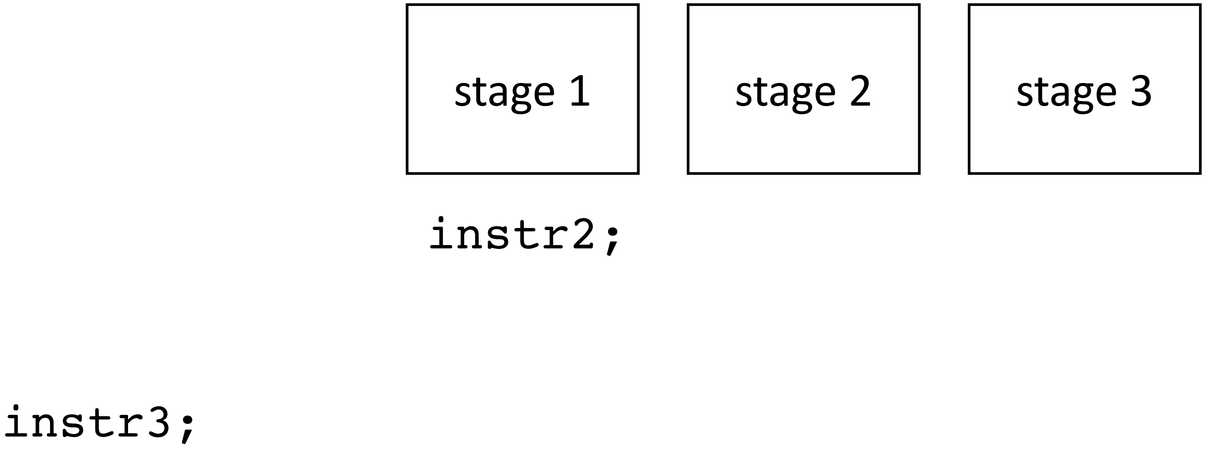
```
instr2;
instr3;
```

What if the instructions depend on each other?

How can hardware execute ILP?

- 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

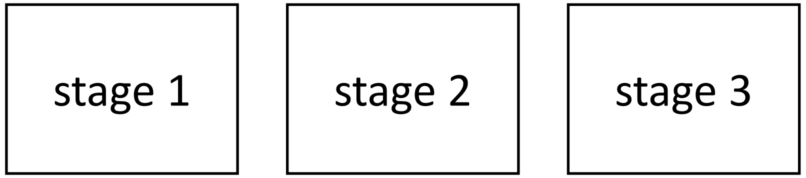
`instr2;`

`instr3;`

What if the instructions depend on each other?

How can hardware execute ILP?

- 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

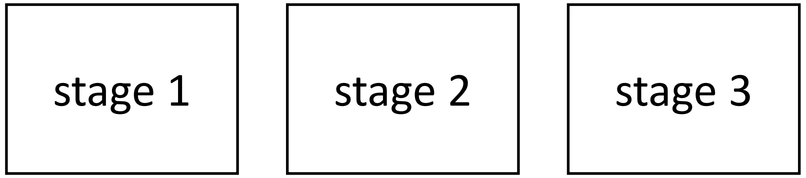
What if the instructions depend on each other?

9 cycles for 3 instructions

converges to 3 cycles per instruction

How can hardware execute ILP?

- Pipeline parallelism



stage 1

stage 2

stage 3

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

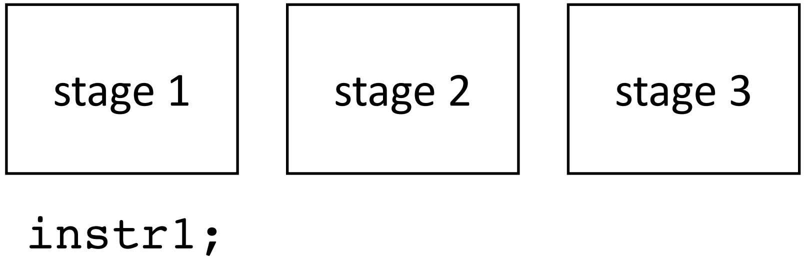
```
instr1;
instrX0;
instrX1;
instr2;
instrX2;
instrX3;
instr3;
```

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

How can hardware execute ILP?

- Pipeline parallelism

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



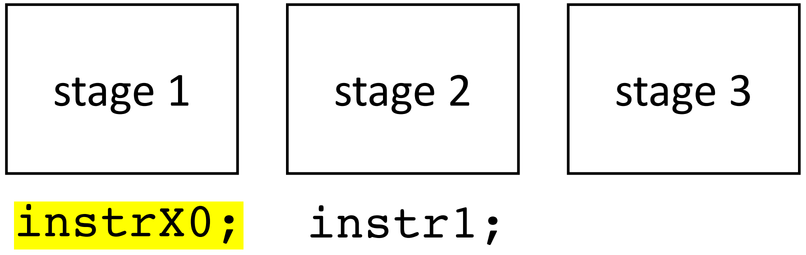
```
instrX0;
instrX1;
instr2;
instrX2;
instrX3;
instr3;
```

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

How can hardware execute ILP?

- 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

instrX0;

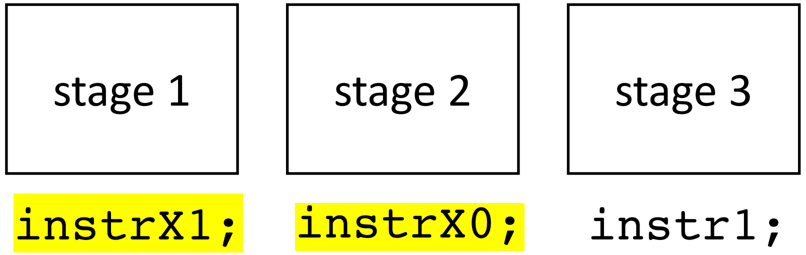
instr1;

```
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!

How can hardware execute ILP?

- Pipeline parallelism



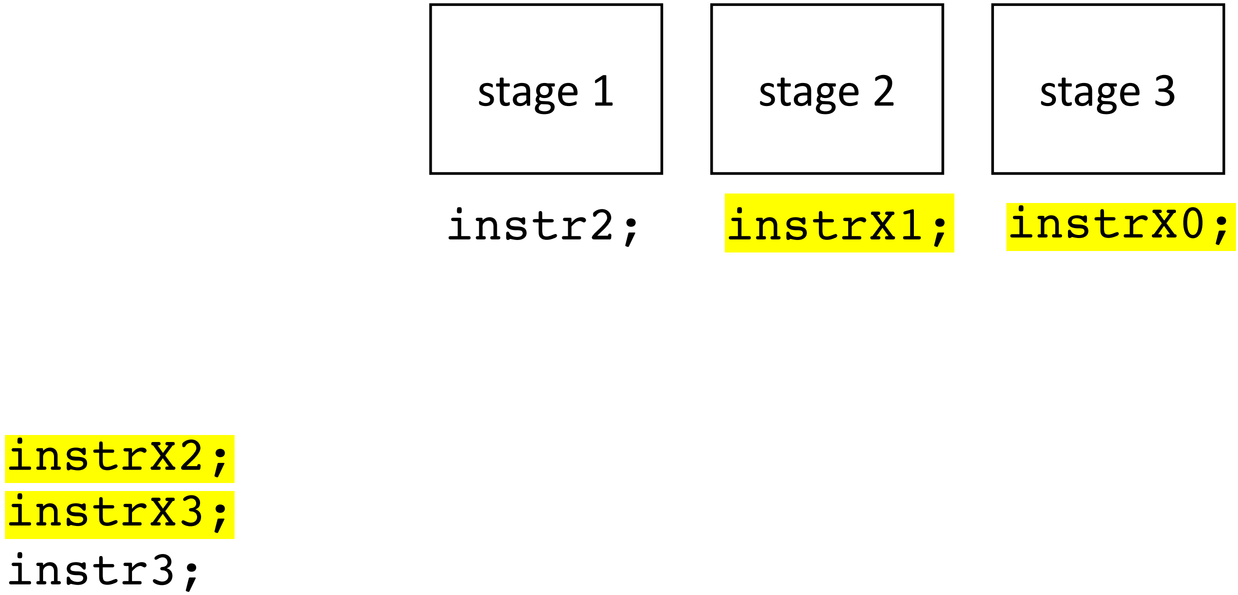
- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline

```
instr2;
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!

How can hardware execute ILP?

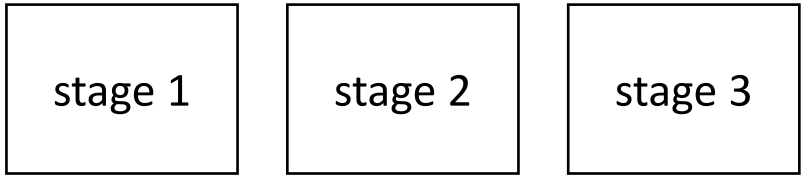
- Pipeline parallelism
- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline



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?

- Pipeline parallelism



stage 1

stage 2

stage 3

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

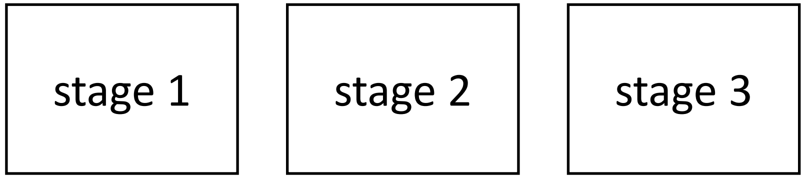
```
instr1;
instr2;
instr3;
```

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

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;
```

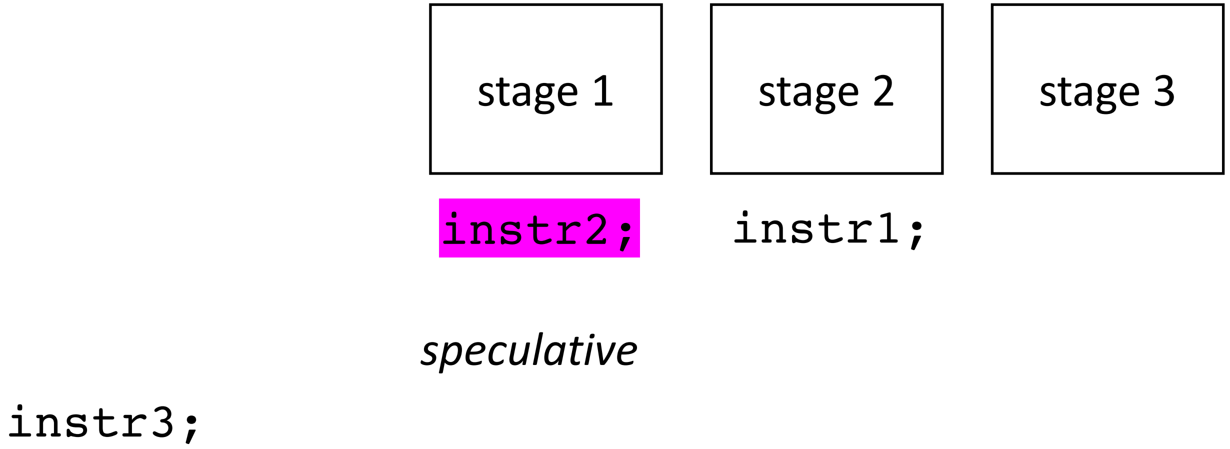
```
instr2;
instr3;
```

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

How can hardware execute ILP?

- 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

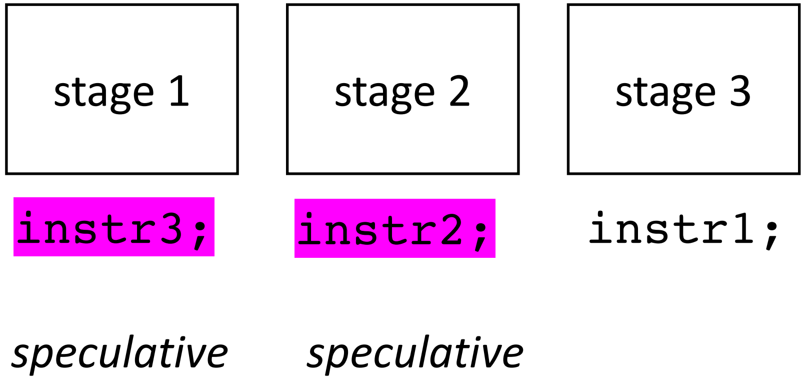
`instr3;`

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

How can hardware execute ILP?

- 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;` `instr1;`

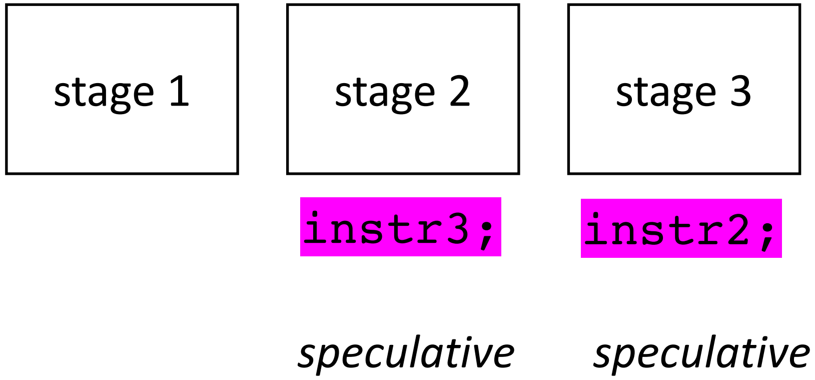
speculative *speculative*

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

How can hardware execute ILP?

- 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;

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)

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

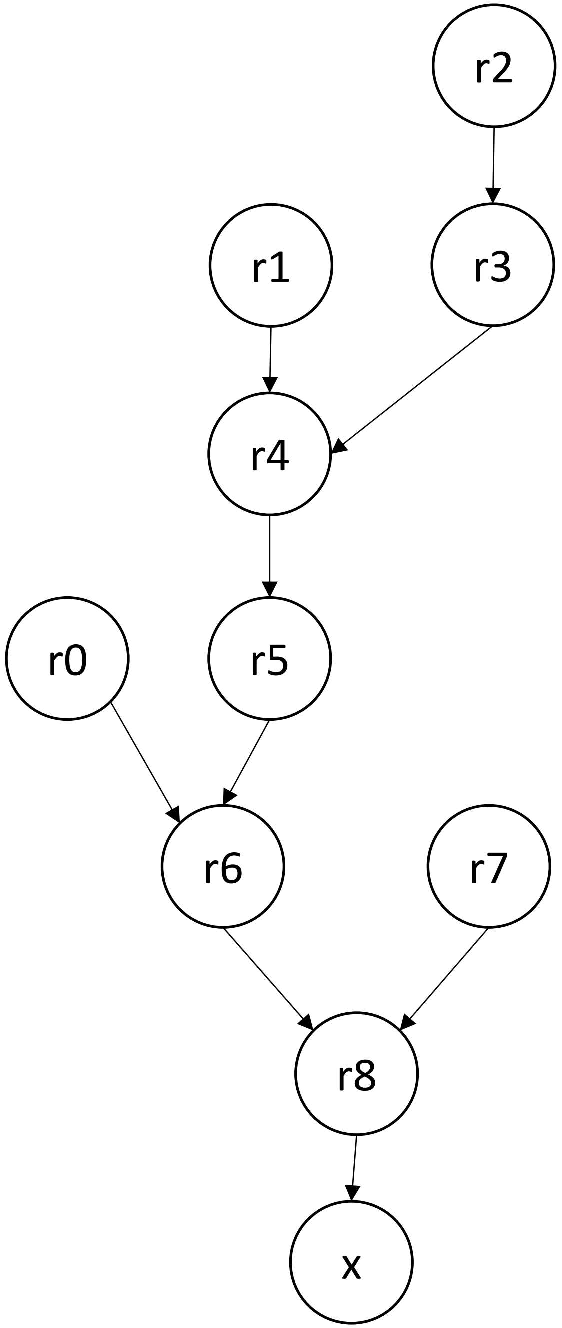
Four compiler techniques for better ILP

- **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;
```

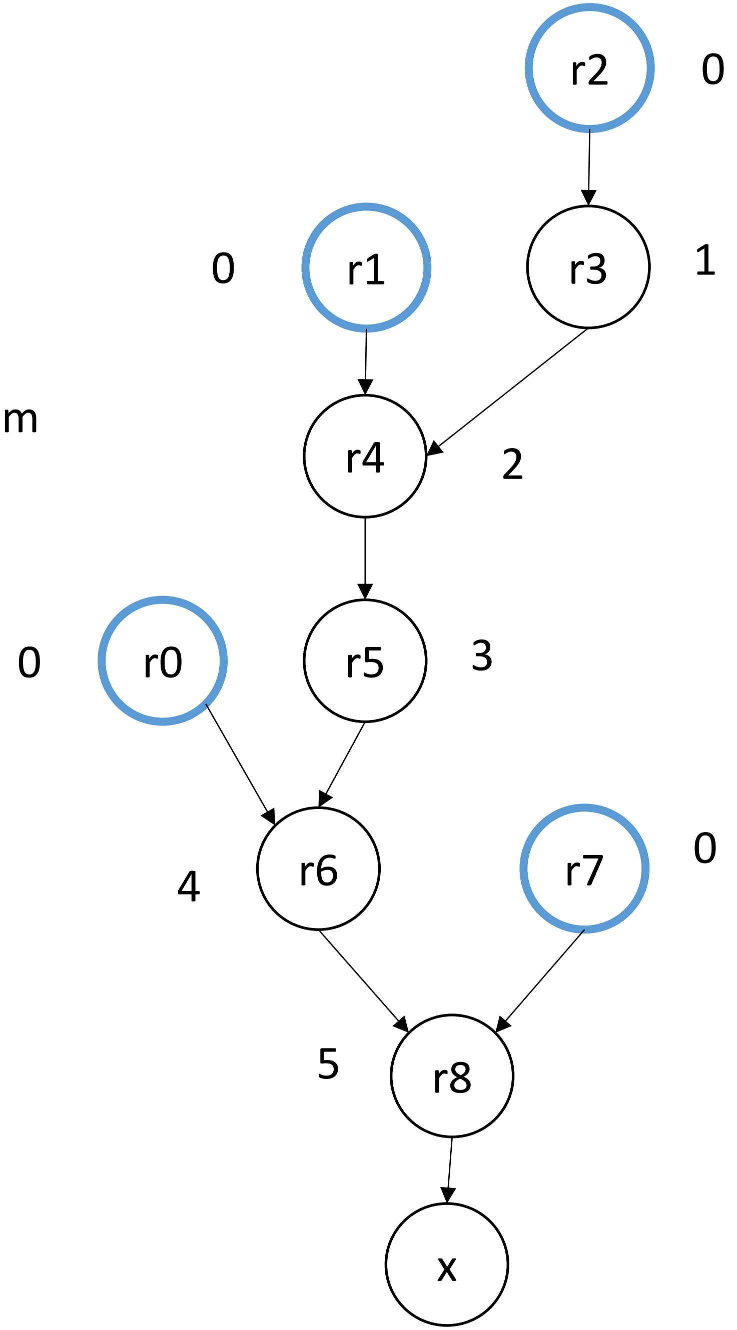
First, consider optimizing for superscalar



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

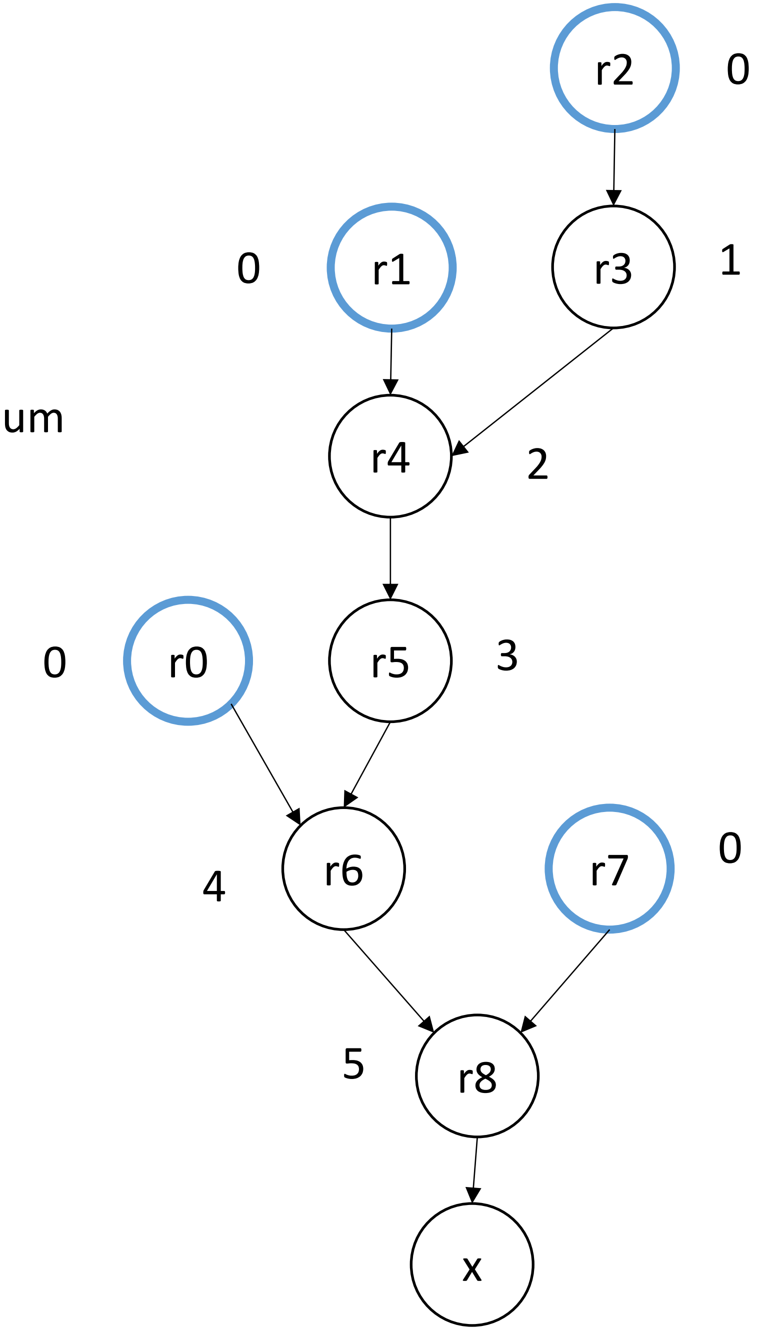


Priority Topological Ordering of DDGs for Superscalar

```
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

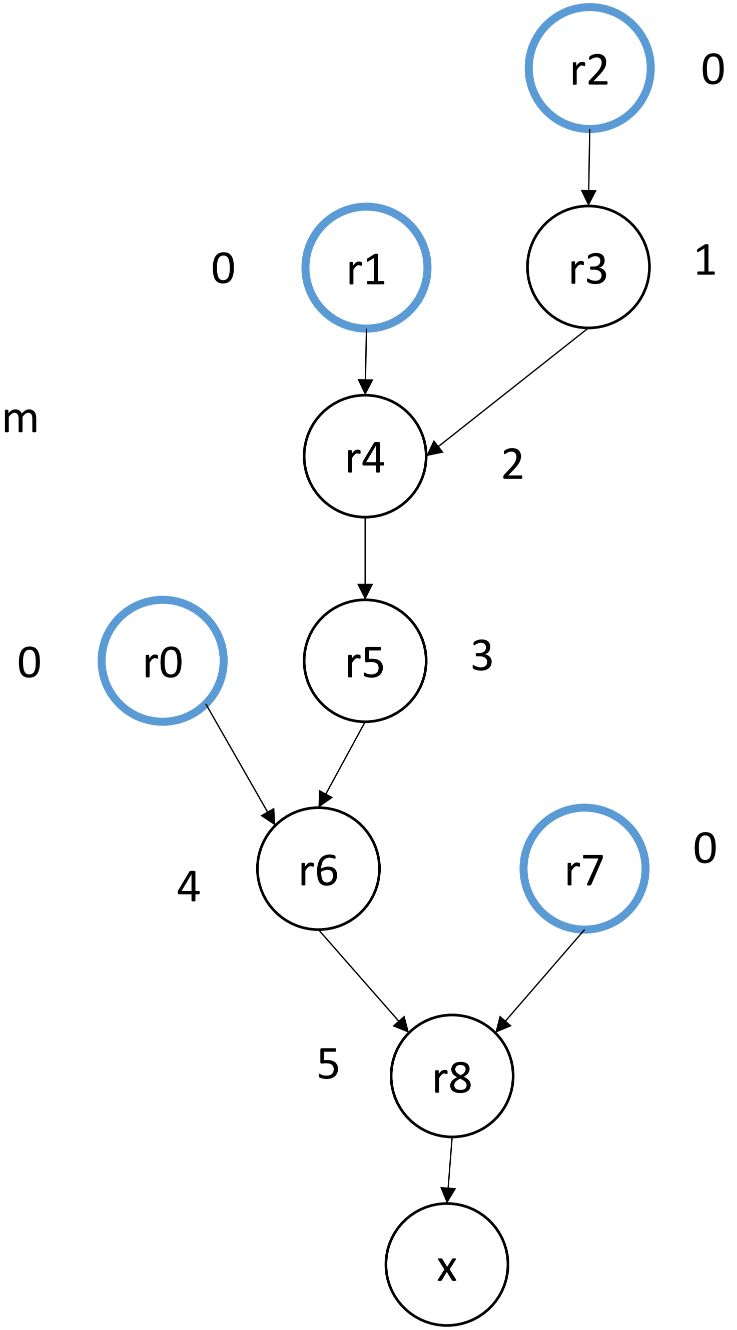


Priority Topological Ordering of DDGs for Superscalar

```
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

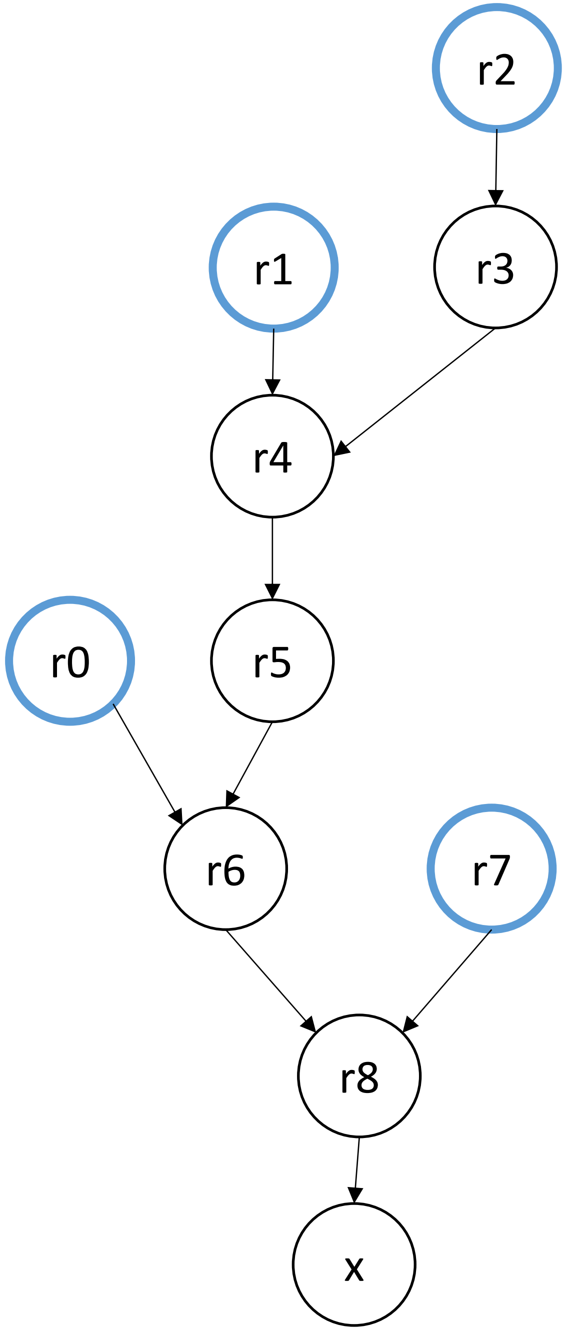
Break ties in topological order using this number



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?

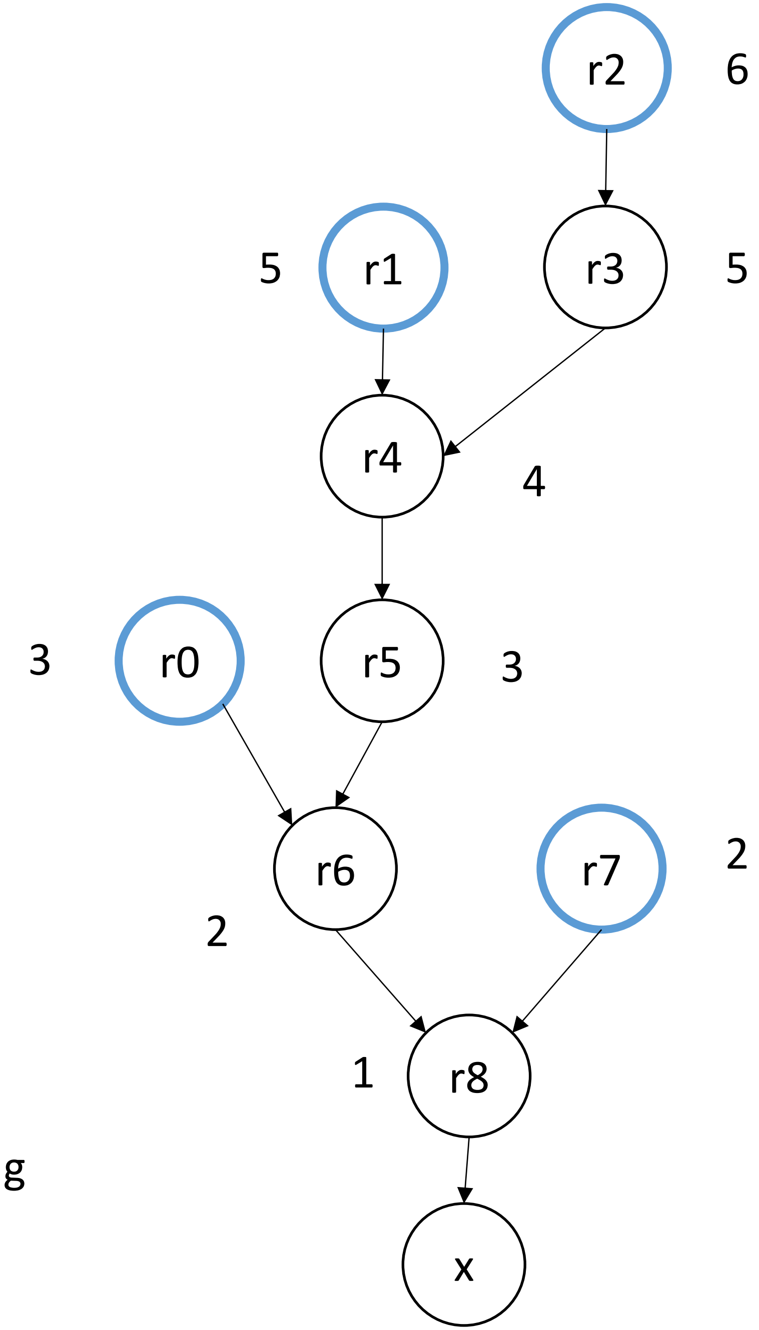


Priority Topological Ordering of DDGs for Pipelining

```
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

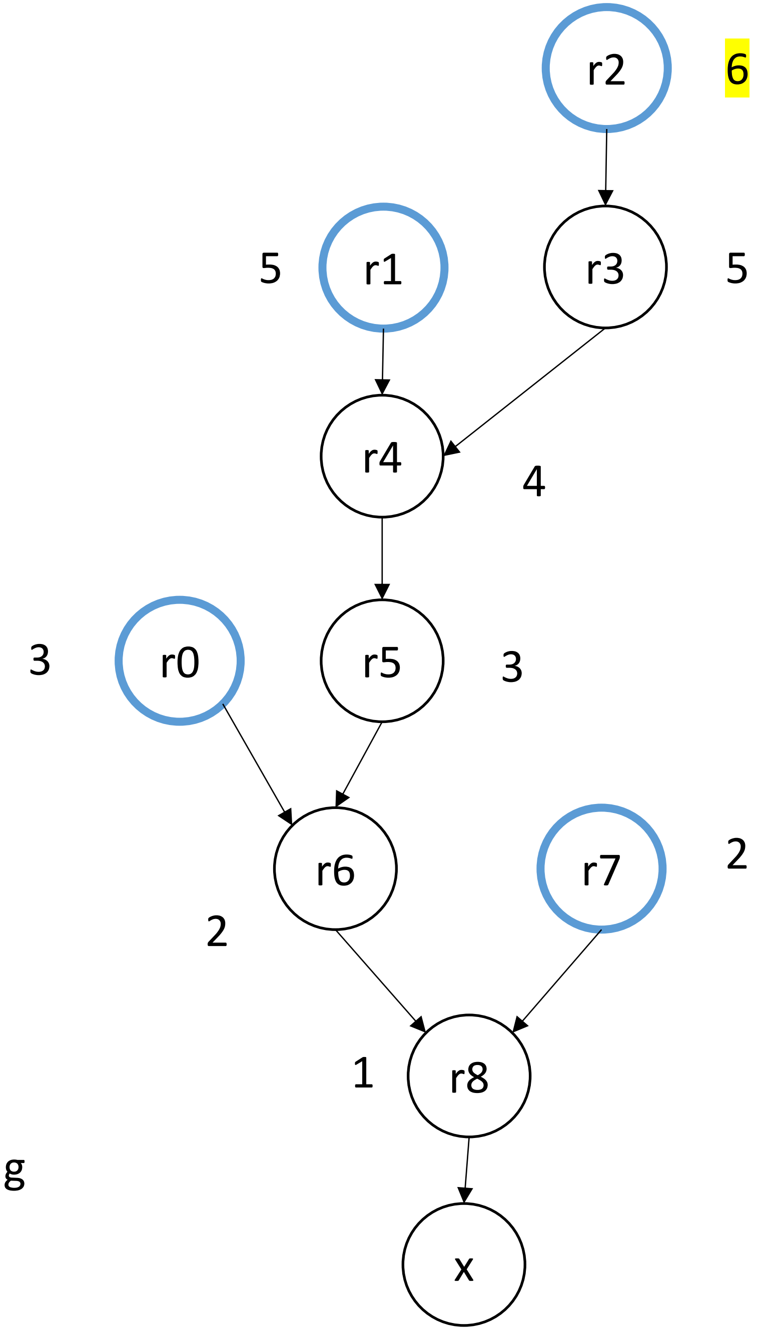


Priority Topological Ordering of DDGs for Pipelining

```
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 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

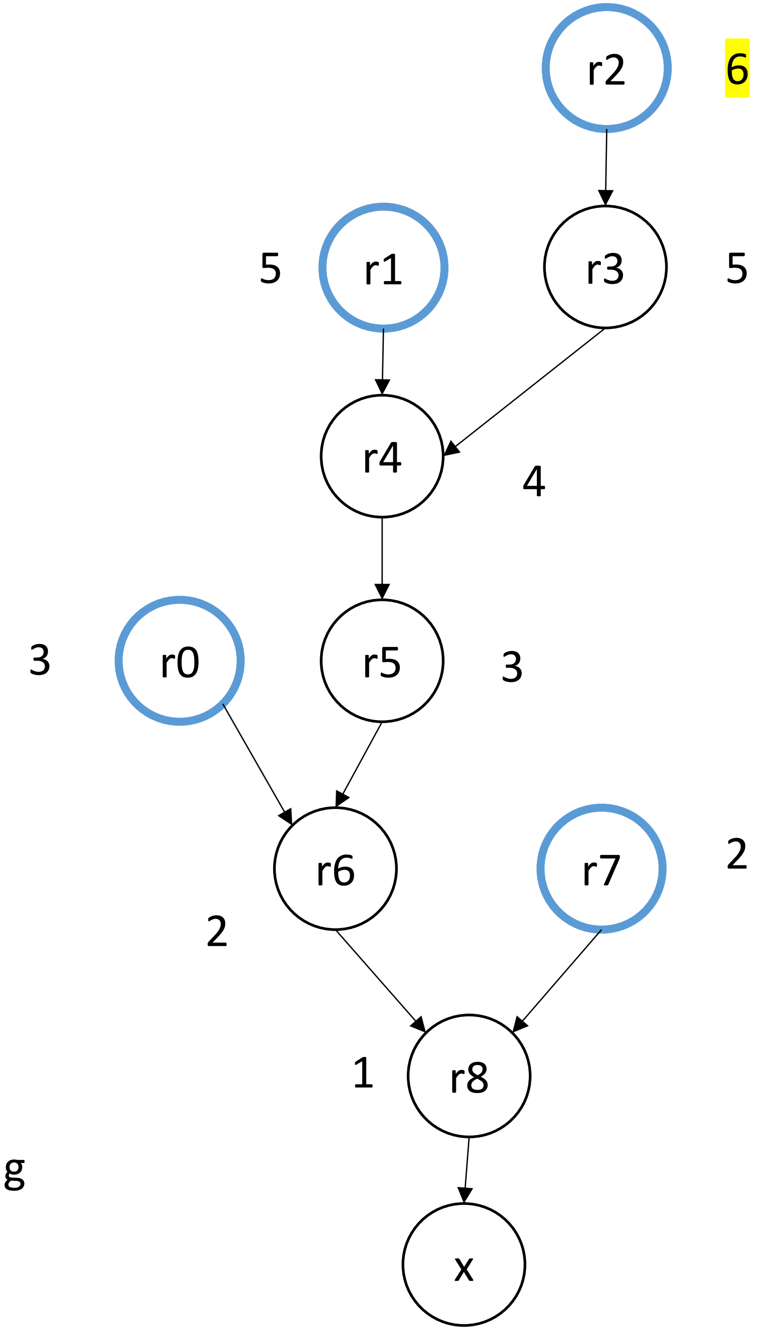


Priority Topological Ordering of DDGs for Pipelining

```
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

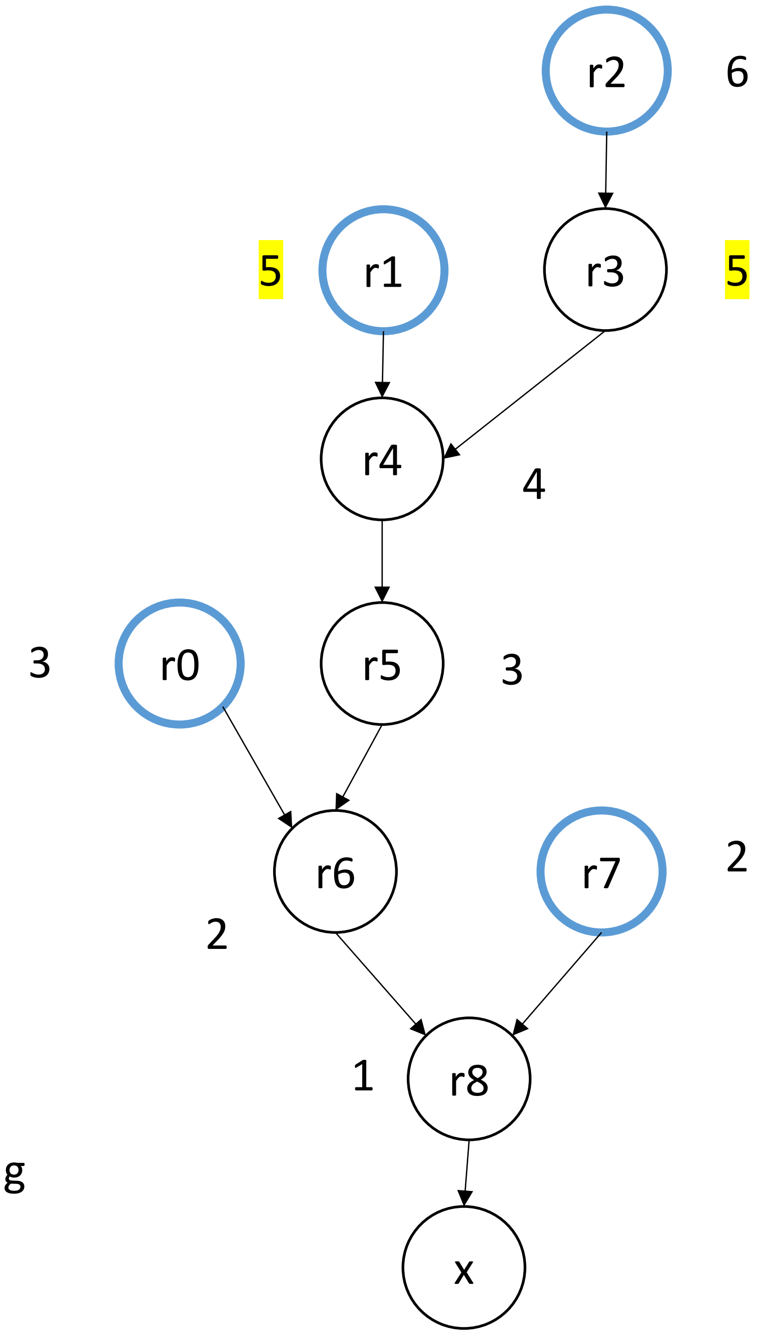


Priority Topological Ordering of DDGs for Pipelining

```
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

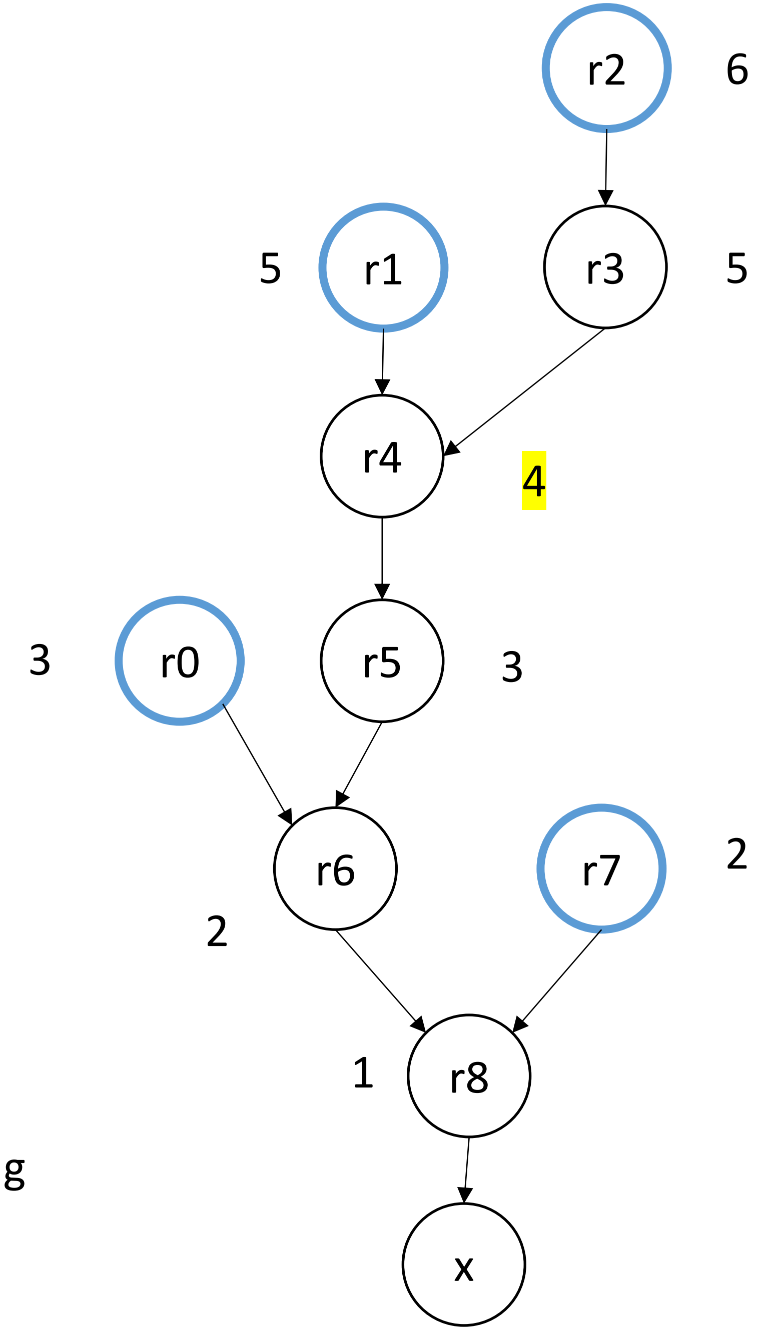


Priority Topological Ordering of DDGs for Pipelining

```
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



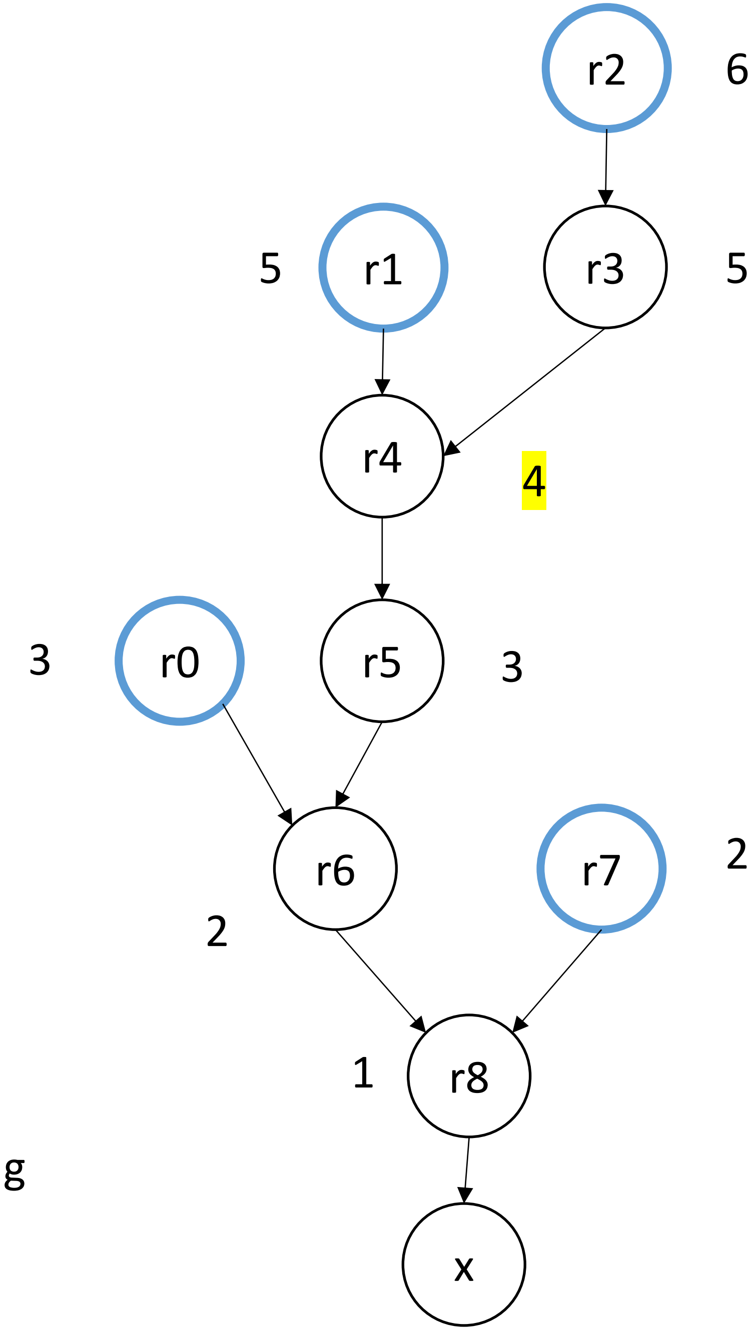
Priority Topological Ordering of DDGs for Pipelining

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

- 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

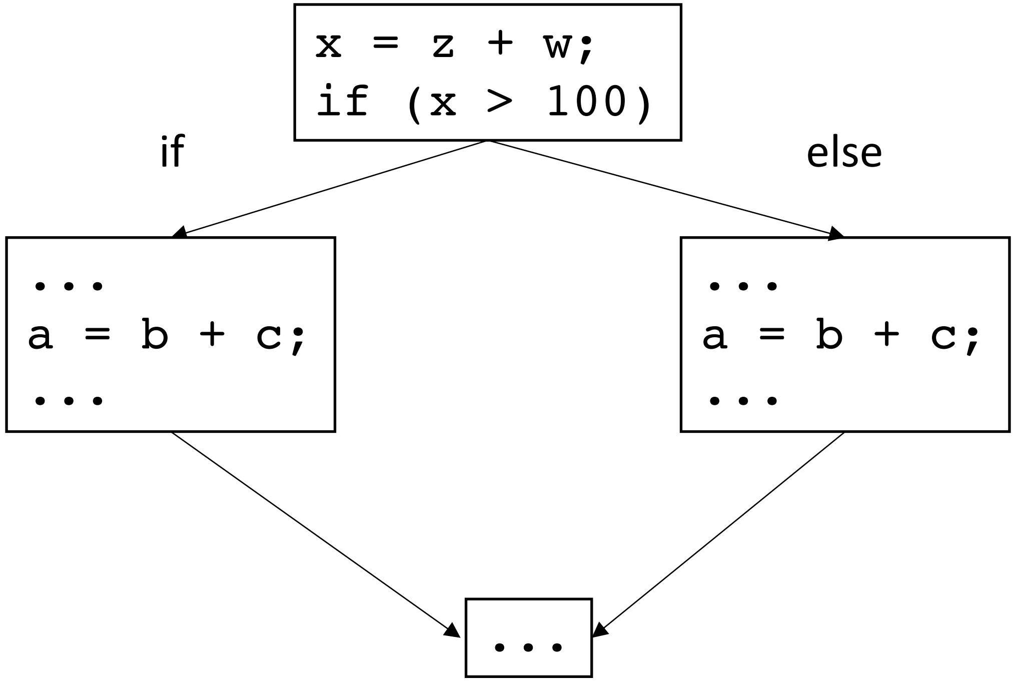
Anticipable Expressions

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

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

Anticipable Expressions

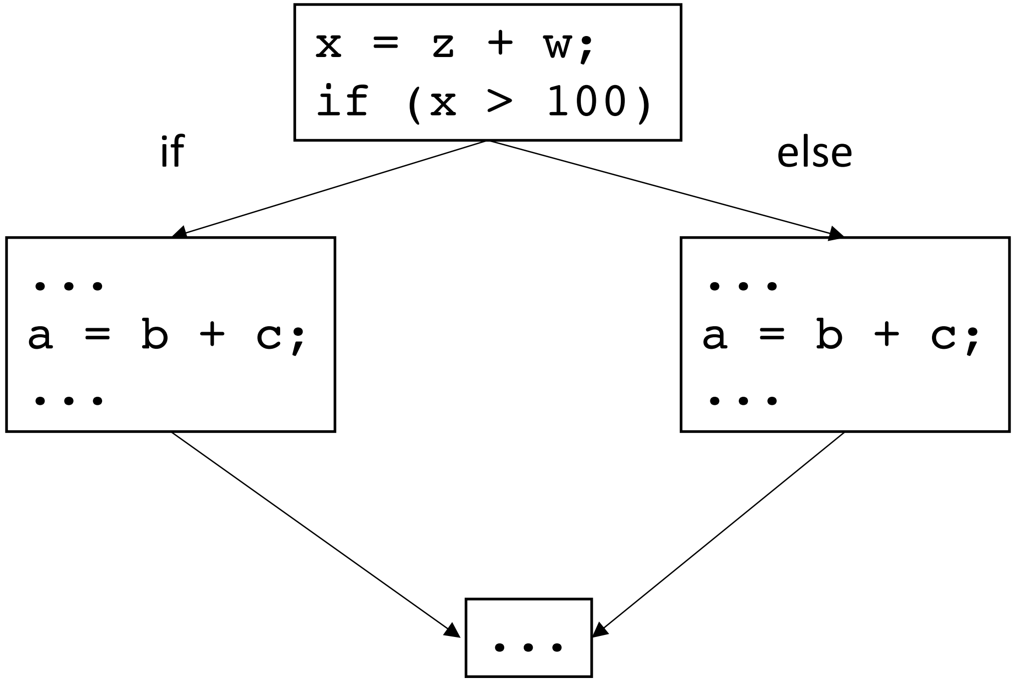
```
x = z + w;
if (x > 100) {
    ...
    a = b + c;
    ...
}
else {
    ...
    a = b + c;
    ...
}
```



Anticipable Expressions

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

antExpr = {a=b+c}



```
x = z + w;
if (x > 100)
```

```
...
a = b + c;
...
```

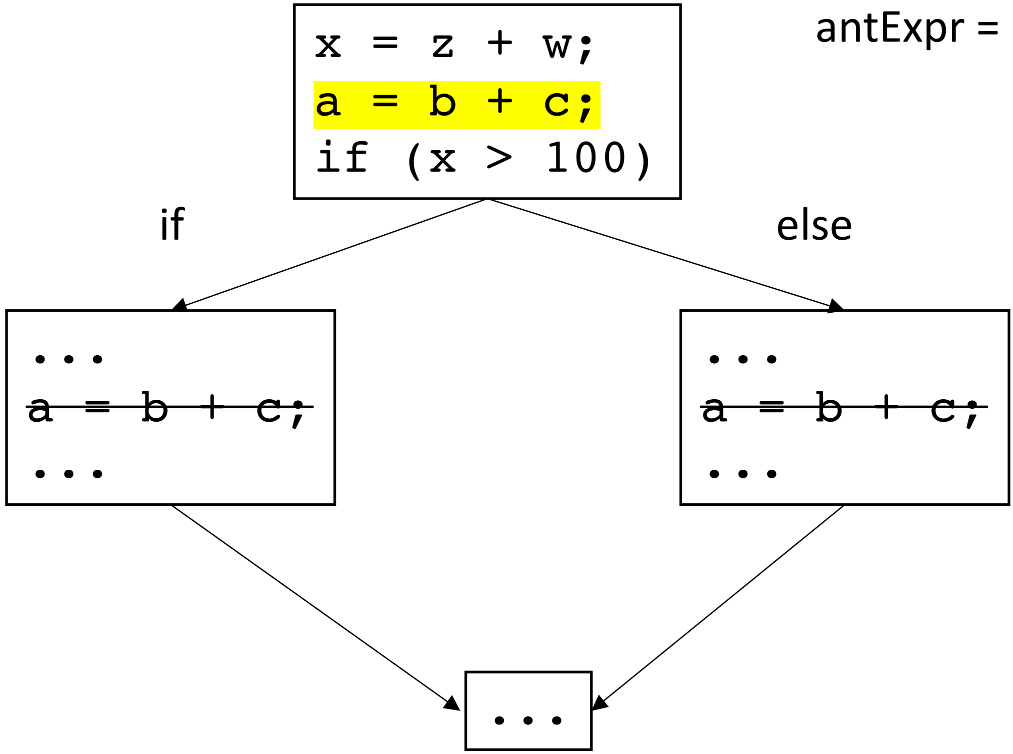
```
...
a = b + c;
...
```

```
...
```

Anticipable Expressions

also called ”Upward code motion”

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



antExpr = {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);
}
```

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 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,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 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);
}
```

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!

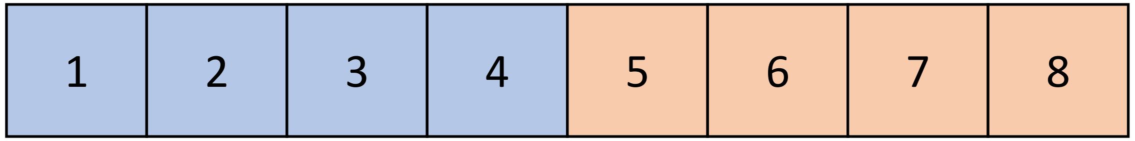
Loop Unrolling for Reduction Loops

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

1	2	3	4	5	6	7	8

Loop Unrolling for Reduction Loops

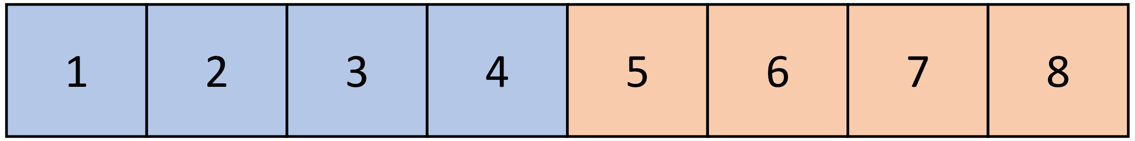
- chunk array in equal sized partitions and do local reductions
- Consider size 2:



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:

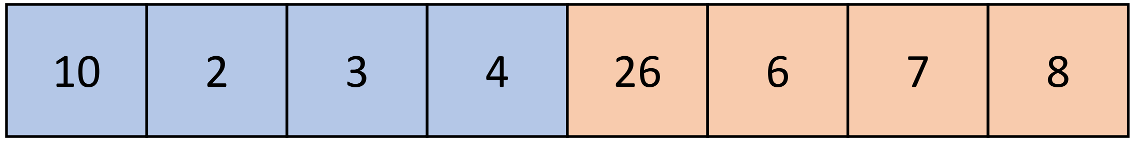


1	2	3	4	5	6	7	8

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

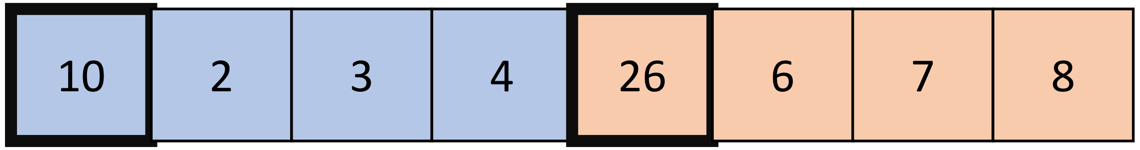
- chunk array in equal sized partitions and do local reductions
- Consider size 2:

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:

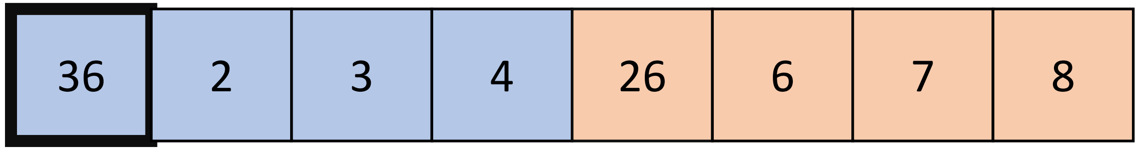


10	2	3	4	26	6	7	8

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;
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

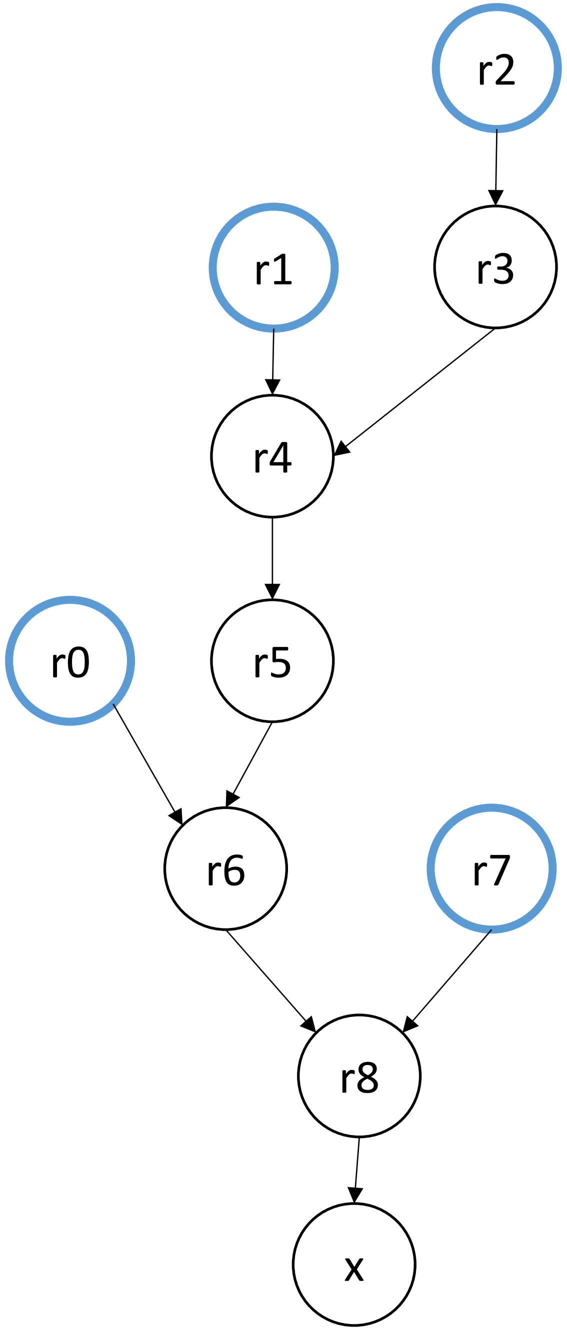
- Cons:
    - Can lead to register spilling, causing expensive loads

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

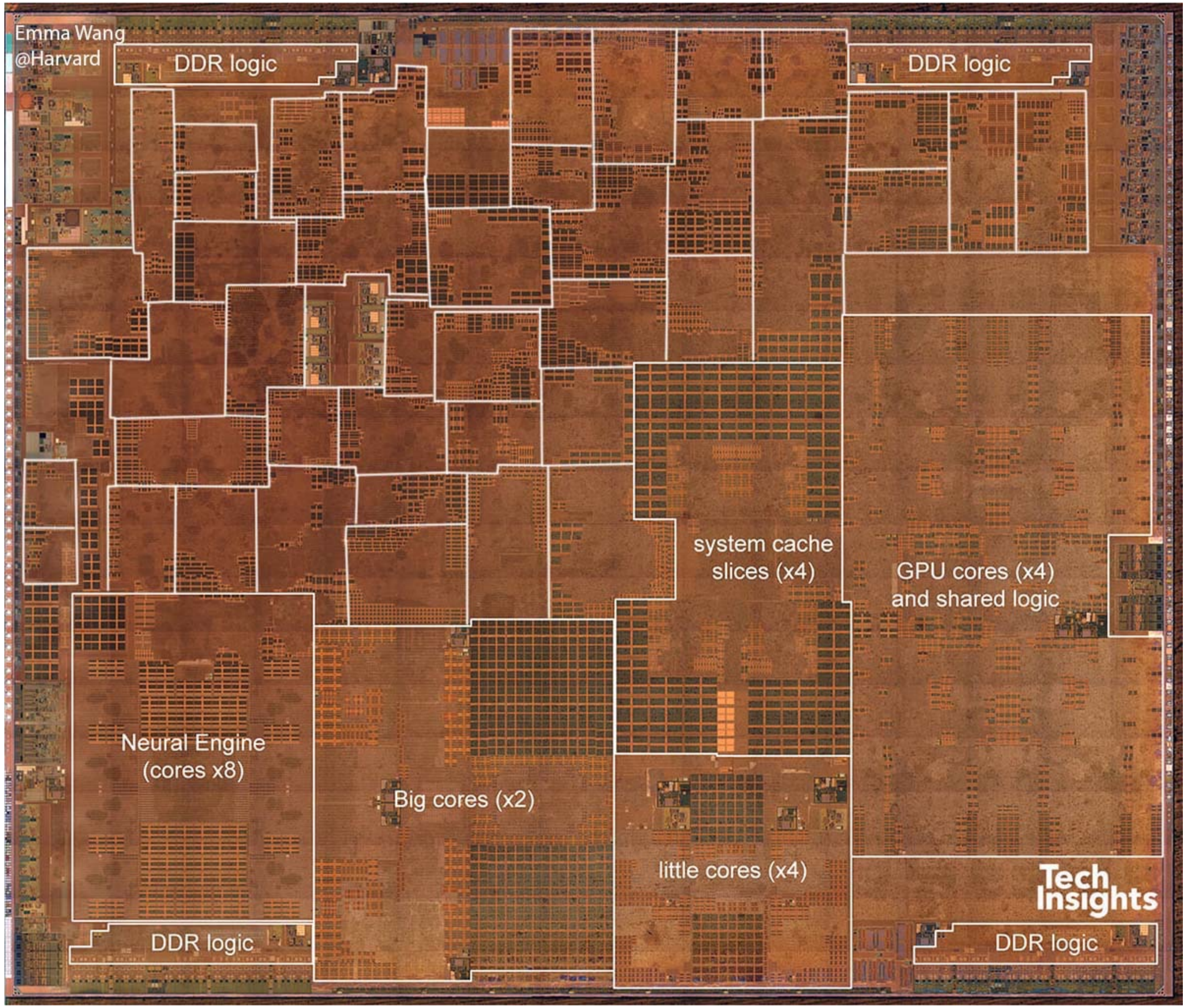
- Where is parallelism most commonly found?
    - Non-numeric applications are thought to have lots of dependencies:
        - I/O (file, network, user),
        - OS, event-driven
        - *[source needed]*
    - numeric applications have less dependencies:
        - media processing (image, video,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

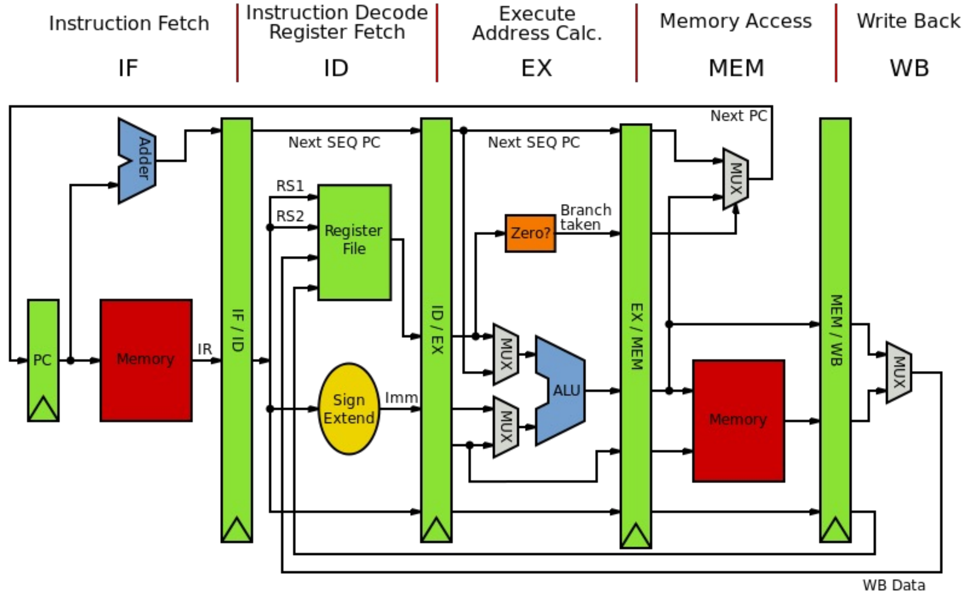
- Complicated cores
    - area/power hungry
    - great for legacy code

- Where do compilers fit in?

Symmetric Multiprocessing (SMP)

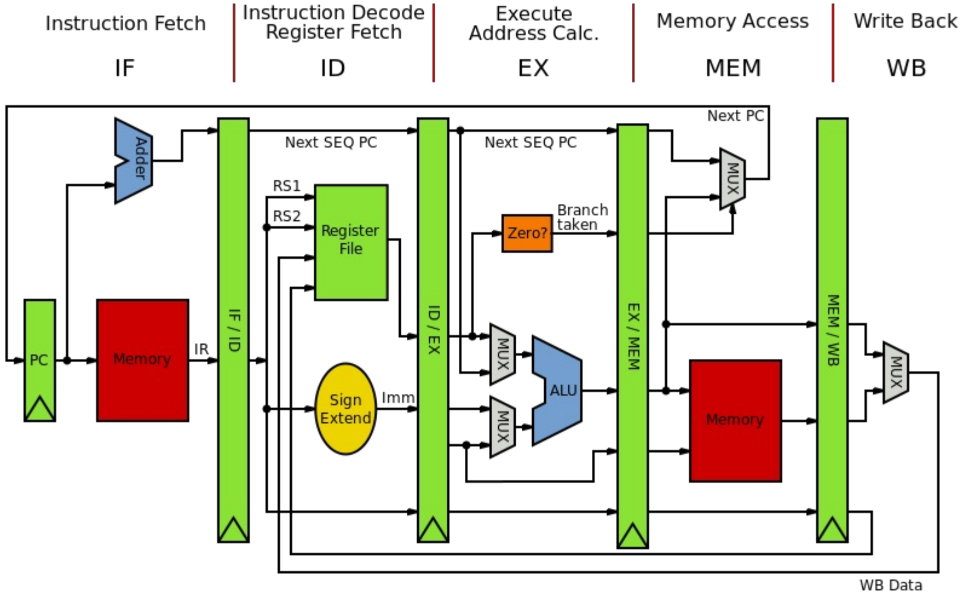
Limits of ILP?

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



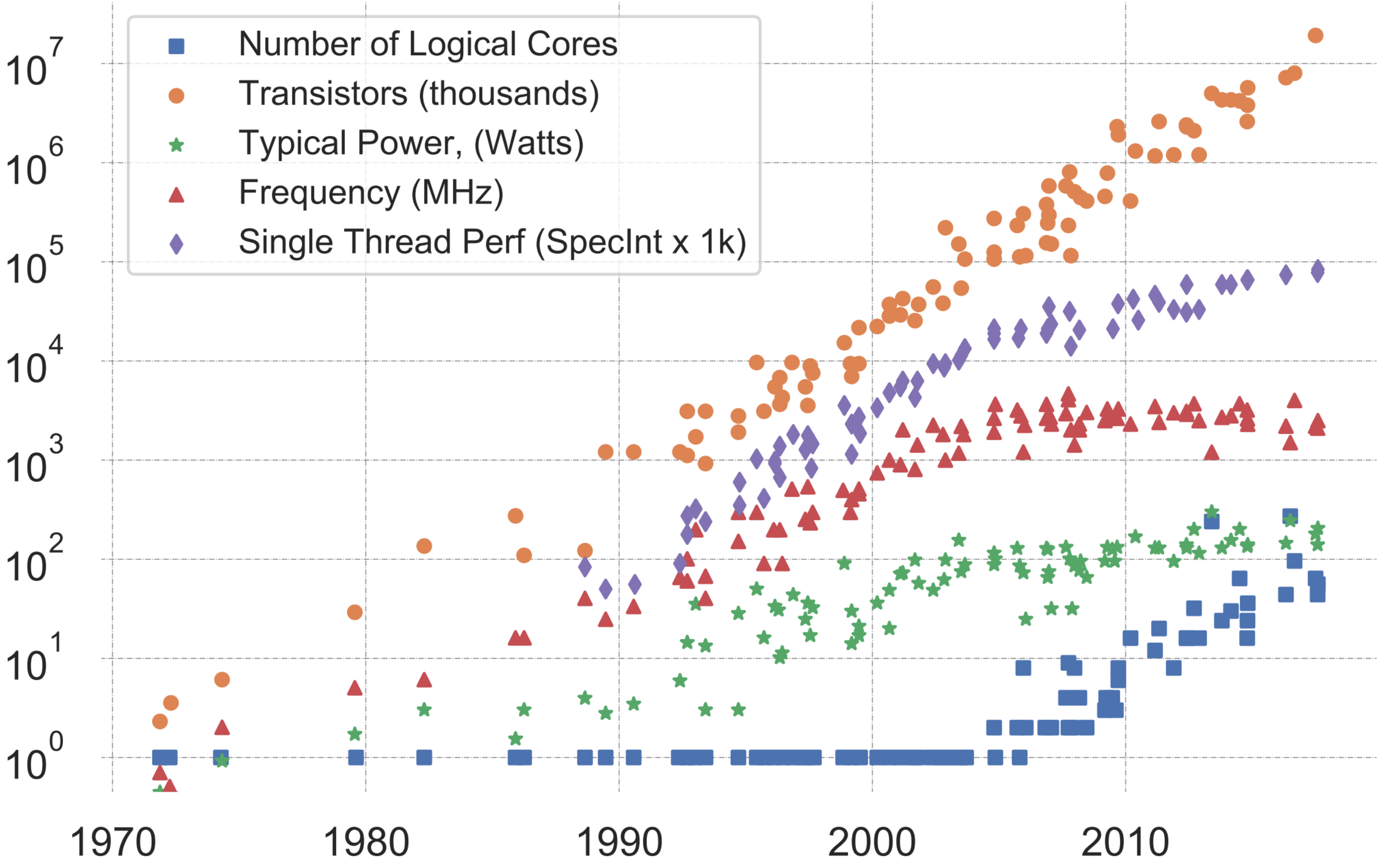
Limits of ILP

- 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 Mircroprocessor 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

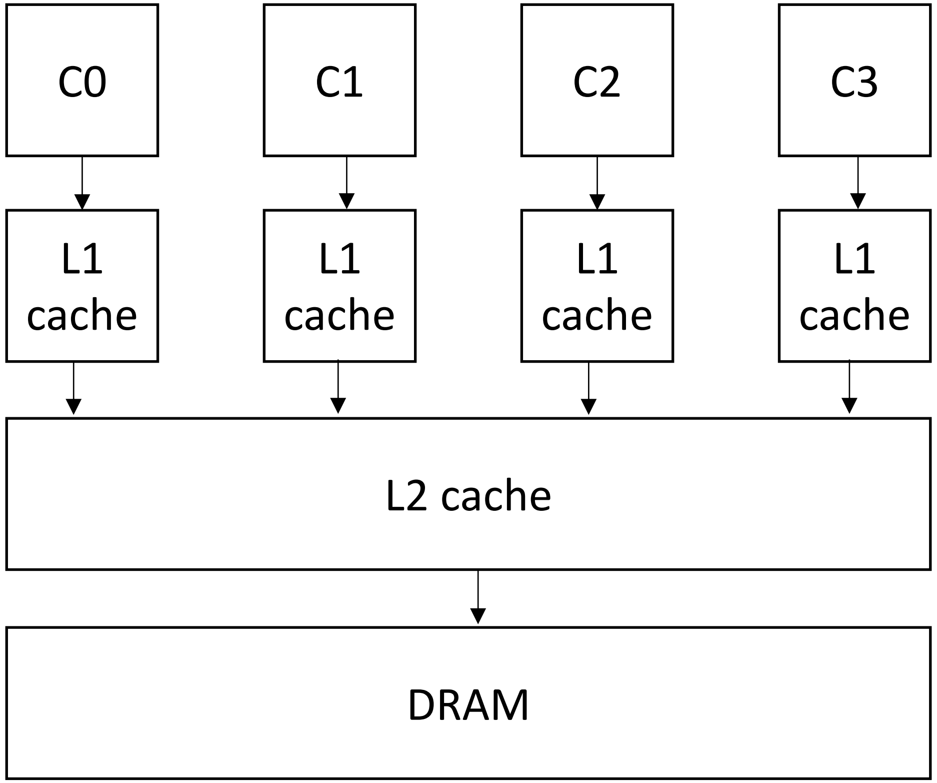
- Number of transistors: **Moore’s law**
  - On its last legs.
  - Intel delaying 7nm chips. Apple has a 5nm. Some roadmaps project up to 3nm

- *Chips are not increasing in raw frequency, and space is becoming more valuable*

Symmetric Multiprocessing (SMP)

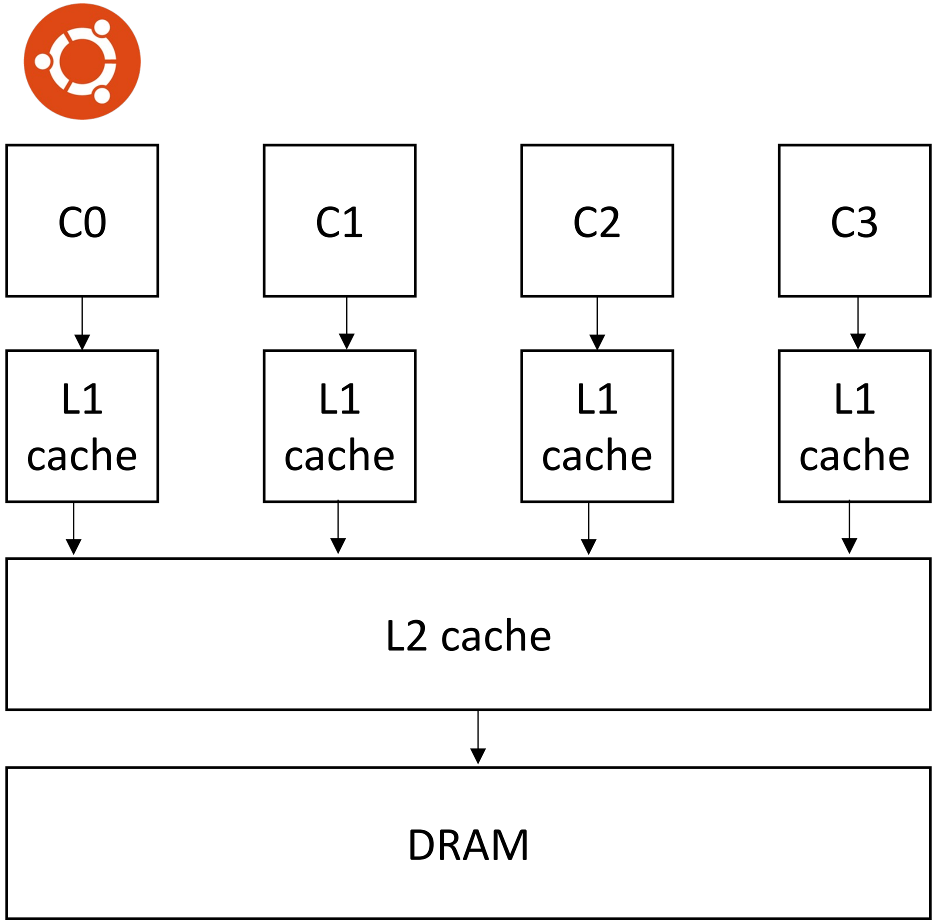
- Collection of “identical” cores
  - Shared memory (access to all system resources)
  - Managed by a single OS

- Pros:
  - Simple(r) HW design
  - Great for multitasking machines



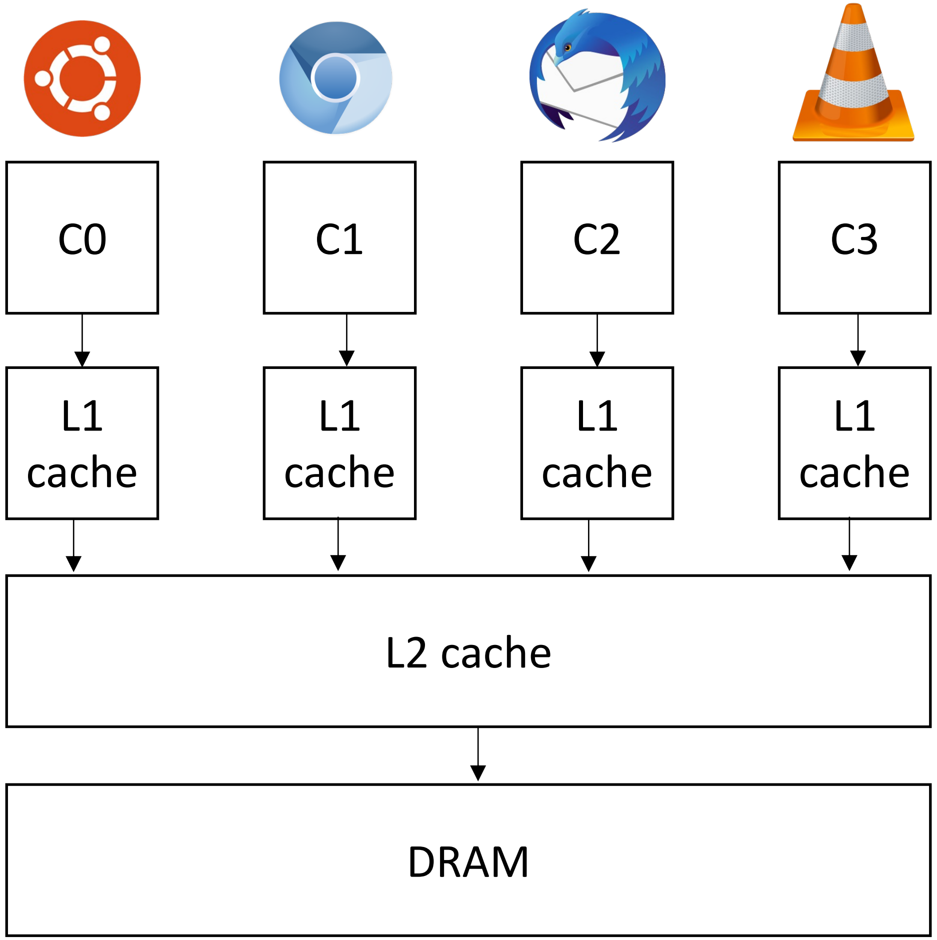
Symmetric Multiprocessing (SMP)

- 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

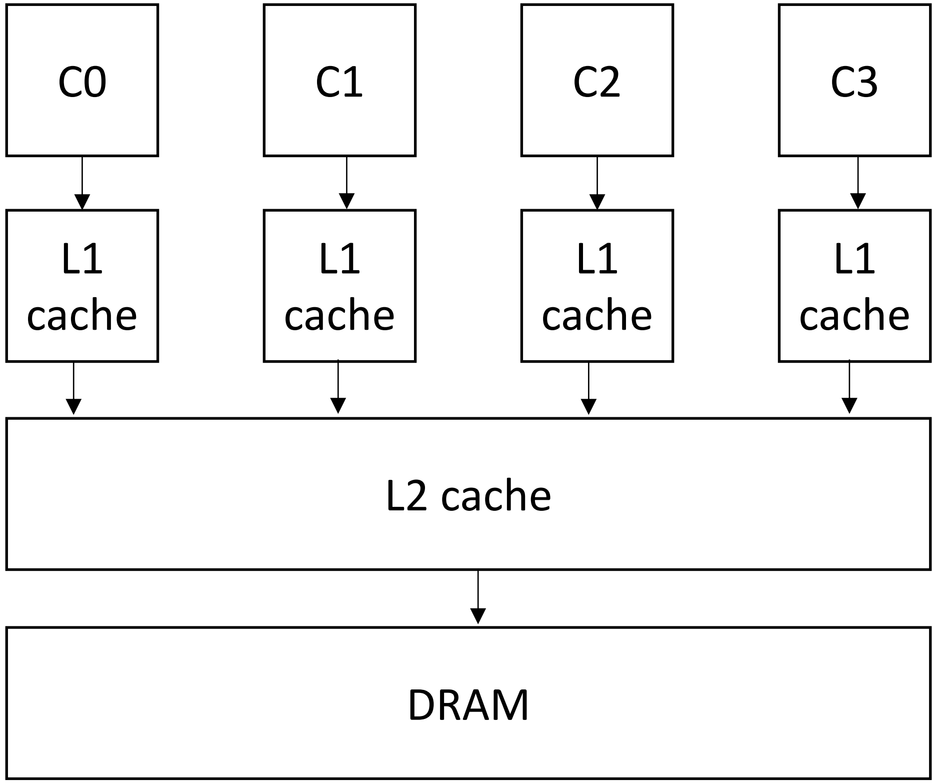


Symmetric Multiprocessing (SMP)

PyTorch

- 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

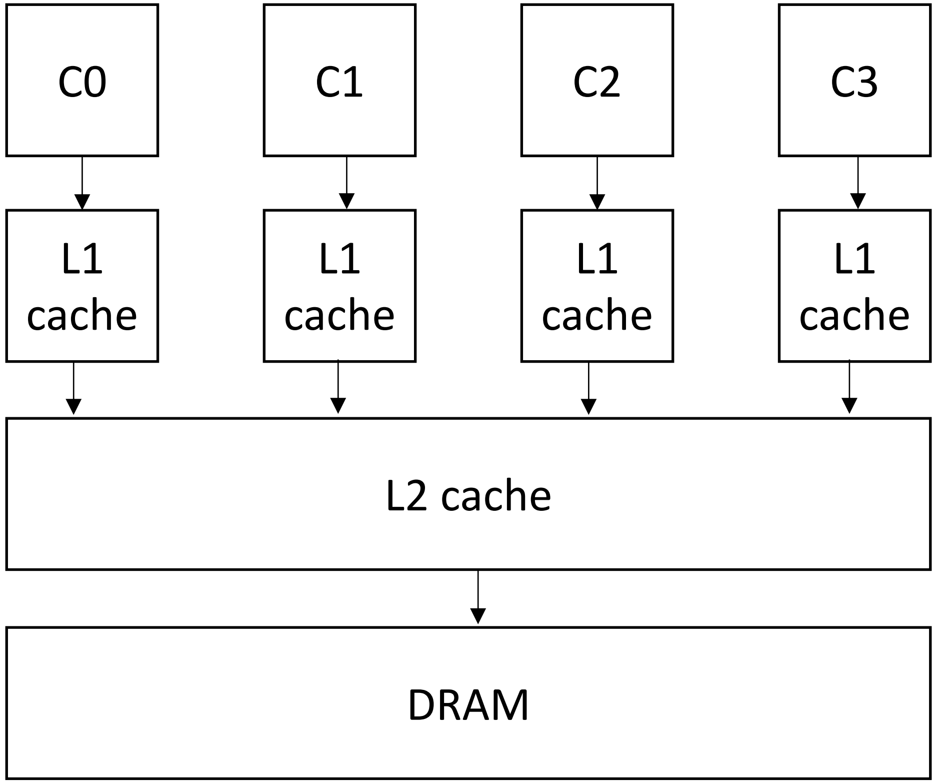


Symmetric Multiprocessing (SMP)

- 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!



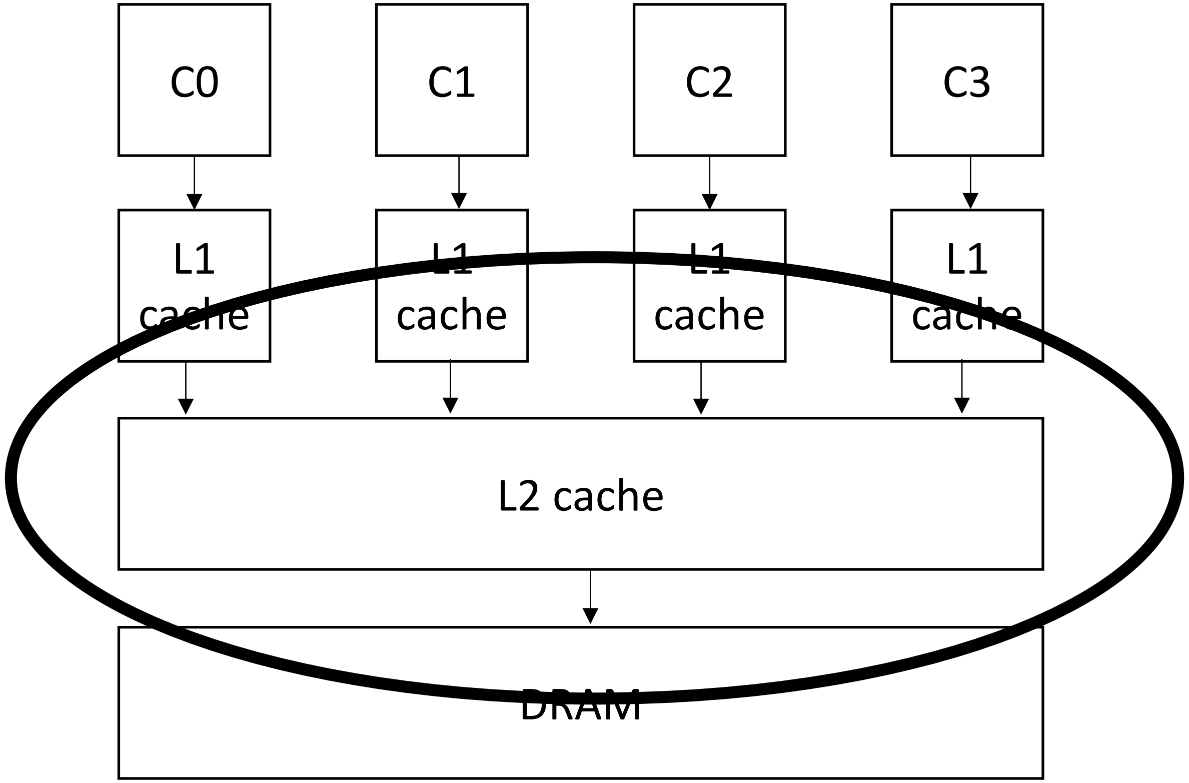
SMP systems are widespread

- Laptops
  - My laptop has 8 cores
  - Most have at least 2
  - New Macbook: 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

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

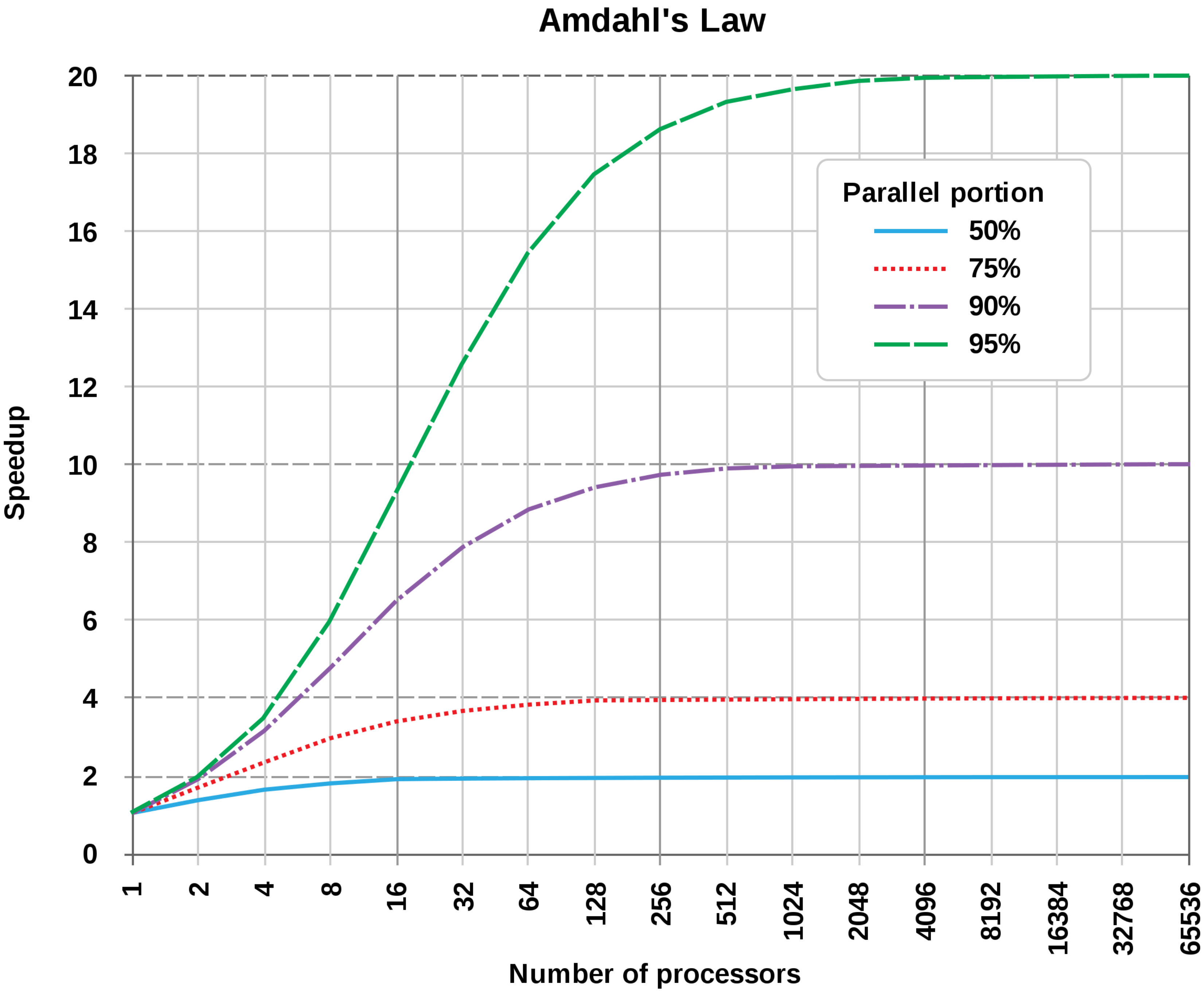
Potential for Parallel Speedup

- 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



from wikipedia

Can compilers help?

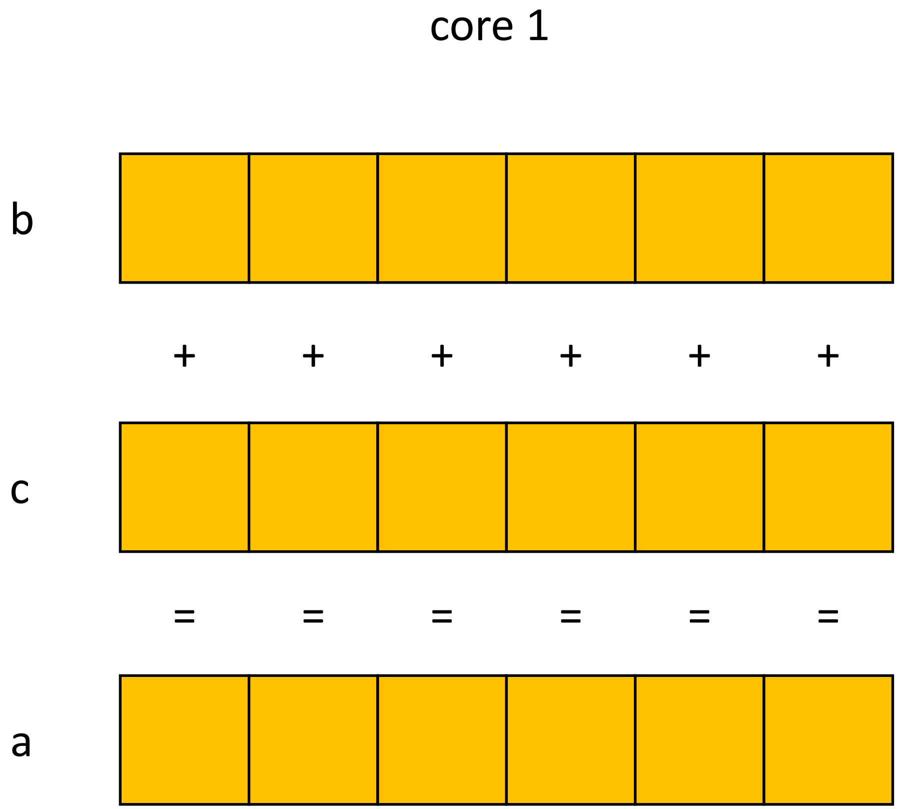
- Much like ILP: convert sequential streams of computation in to SMP parallel code.

- Much harder constraints
  - Correctness
  - Performance

- For loops are a good target for compiler analysis

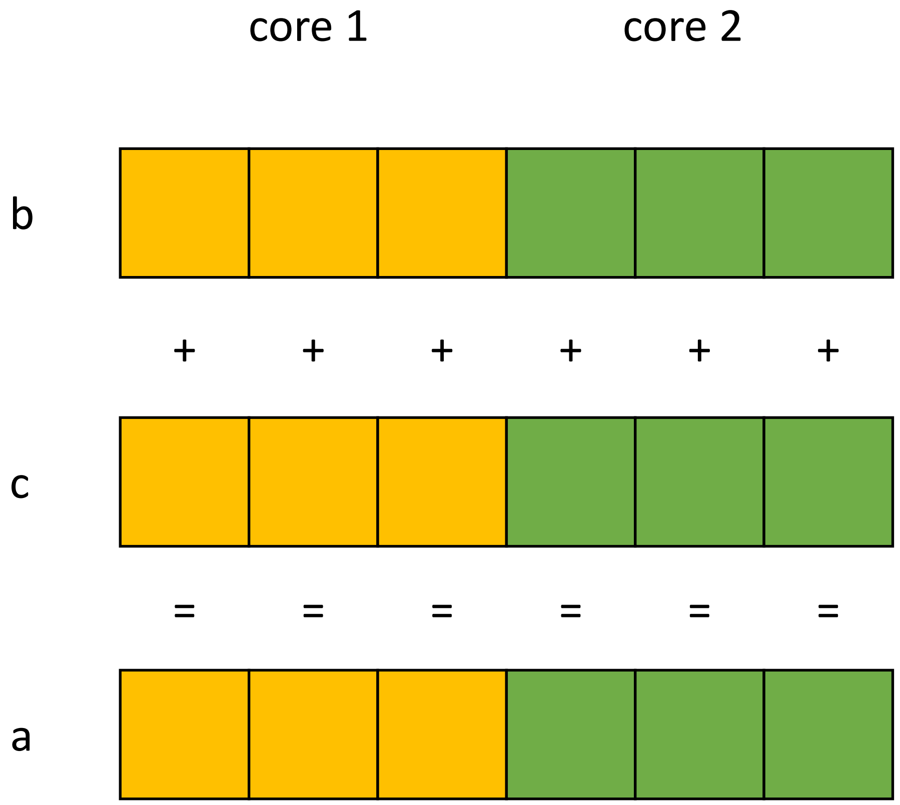
For loops are great candidates for SMP parallelism

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



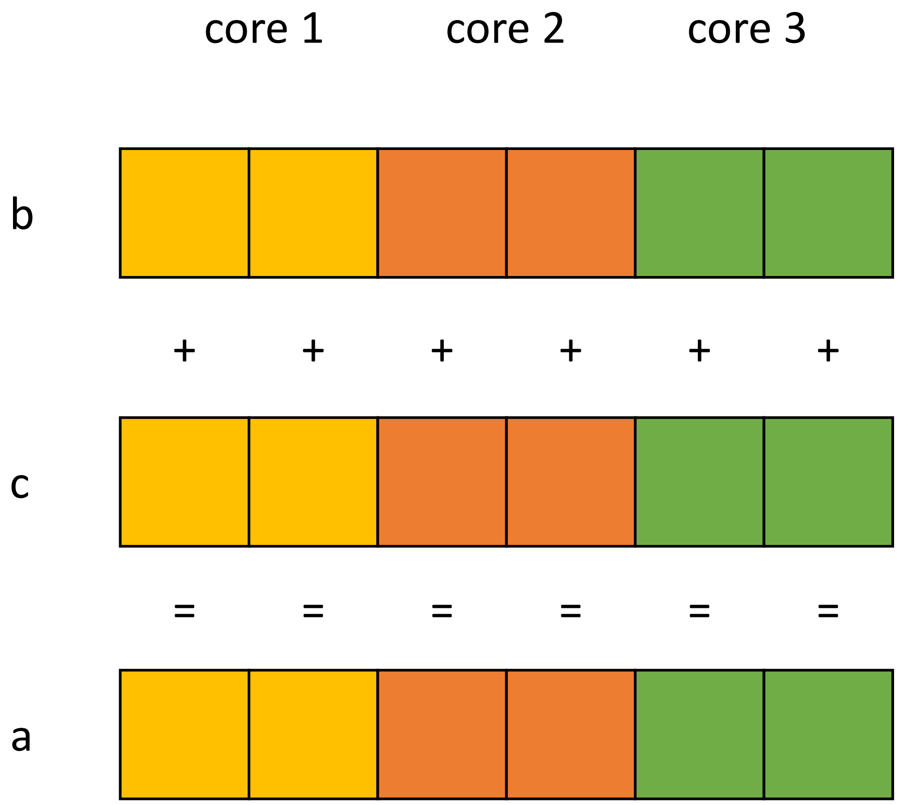
For loops are great candidates for SMP parallelism

```
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 Monday!

- DOALL For loops