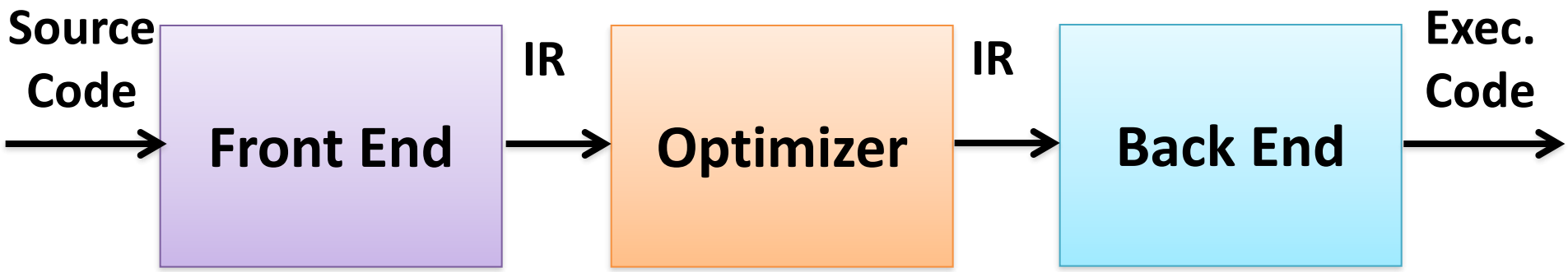
Introduction to Compilers, Part 3

ITP 435
Week 11, Lecture 1

Three Stage "Optimizing" Compiler

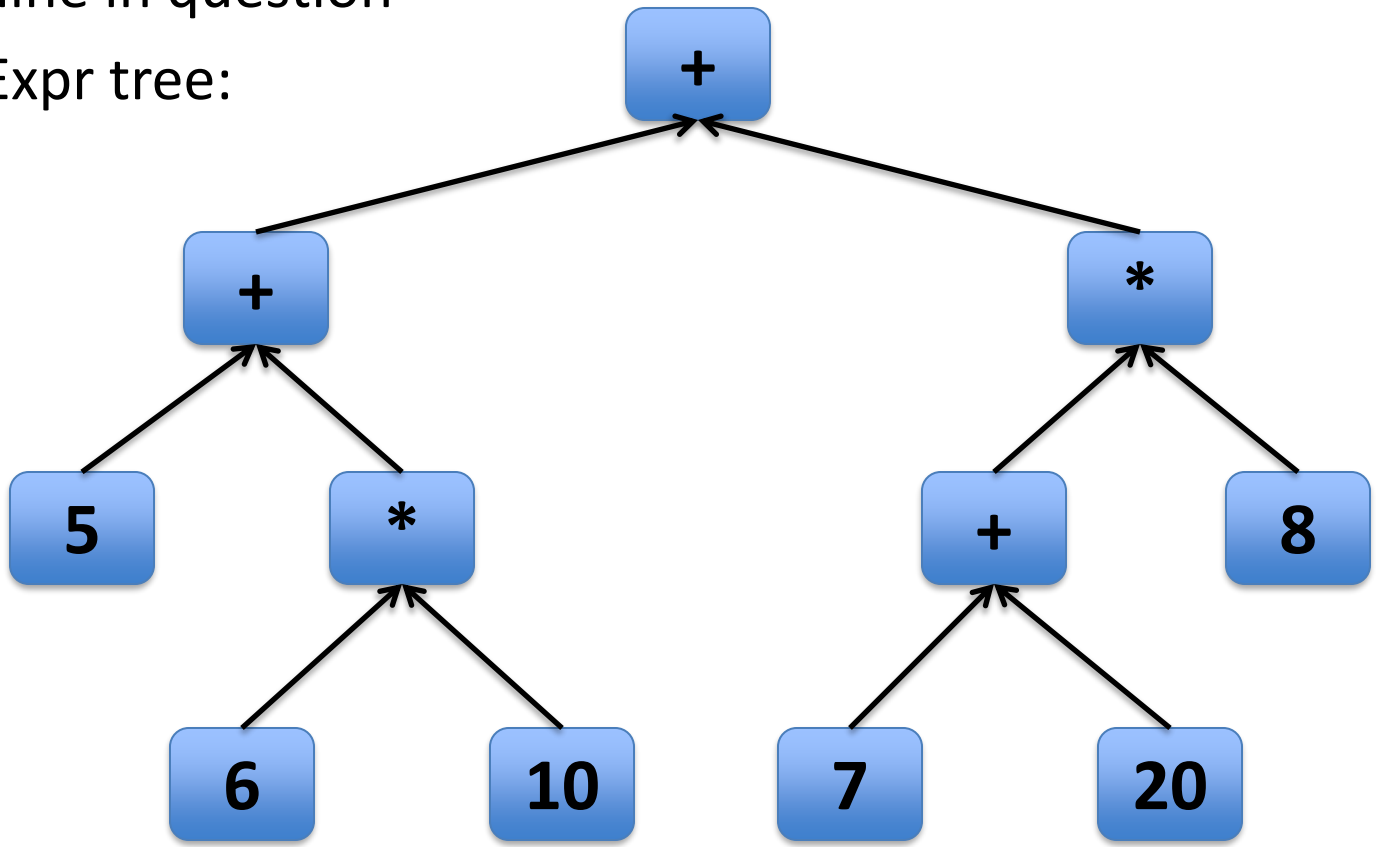


Back End


Instruction Selection
Register Allocation
Instruction Scheduling

Code Generation

- Given an IR, we must generate assembly that will run on the target machine in question
- Our Expr tree:



Our ITP-11 Architecture

- It's very RISC-like (kinda MIPSy)

- Many ops have a destination and two sources:

add r1,r2,r3

- (Commonly called three address code)

Adding Code Gen to the AST Node (Simple Case)

```
struct CodeContext
{
    std::vector<std::string> ops;
};

class Node
{
public:
    void codeGen(CodeContext& c) = 0;
};
```

Recommended CodeContext starting point

```
struct CodeContext
{
    // Store the op name separate from operands
    std::vector<std::pair<std::string,
                    std::vector<strings>>> ops;
};

class Node
{
public:
    void codeGen(CodeContext& c) = 0;
};
```

Adding Code Gen to the AST Node, Cont'd

```
void NAdd::codeGen( CodeContext& c )
{
   mLhs->codeGen(c);
   mRhs->codeGen(c);

   // This is specifically from calc example
   // you won't have this function
   AddOp(c, std::string("add"));
}
```

Register Allocation

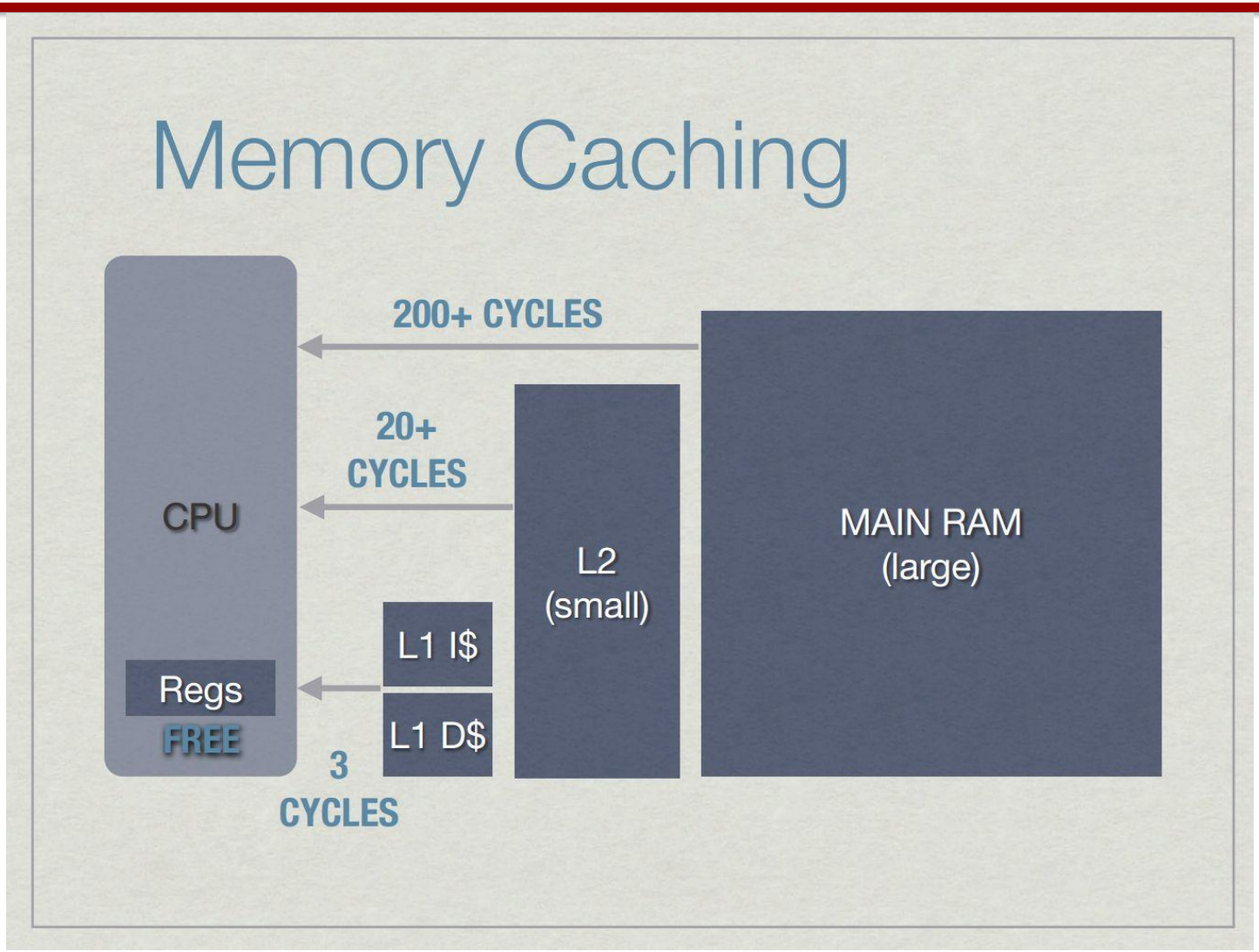
Problem: Register Allocation

- We have a limited number of registers
- We want data to be in registers as much as possible, because it's way more efficient
- Given the following expression:

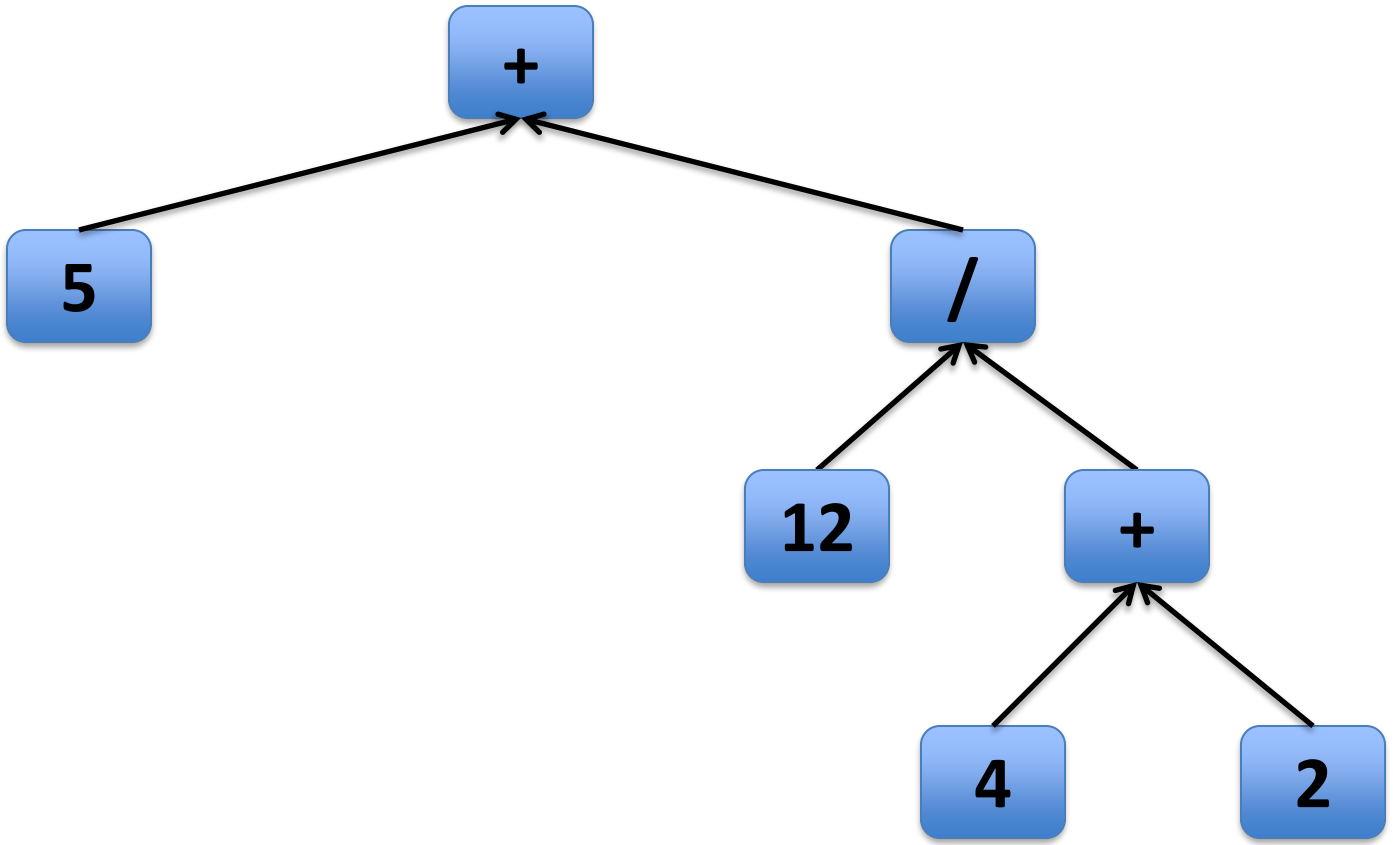
5+12/(4+2)

What is the most efficient way to calculate this in x86 assembly?

Register Allocation – What's the Big Deal?



AST for 5+12/(4+2)

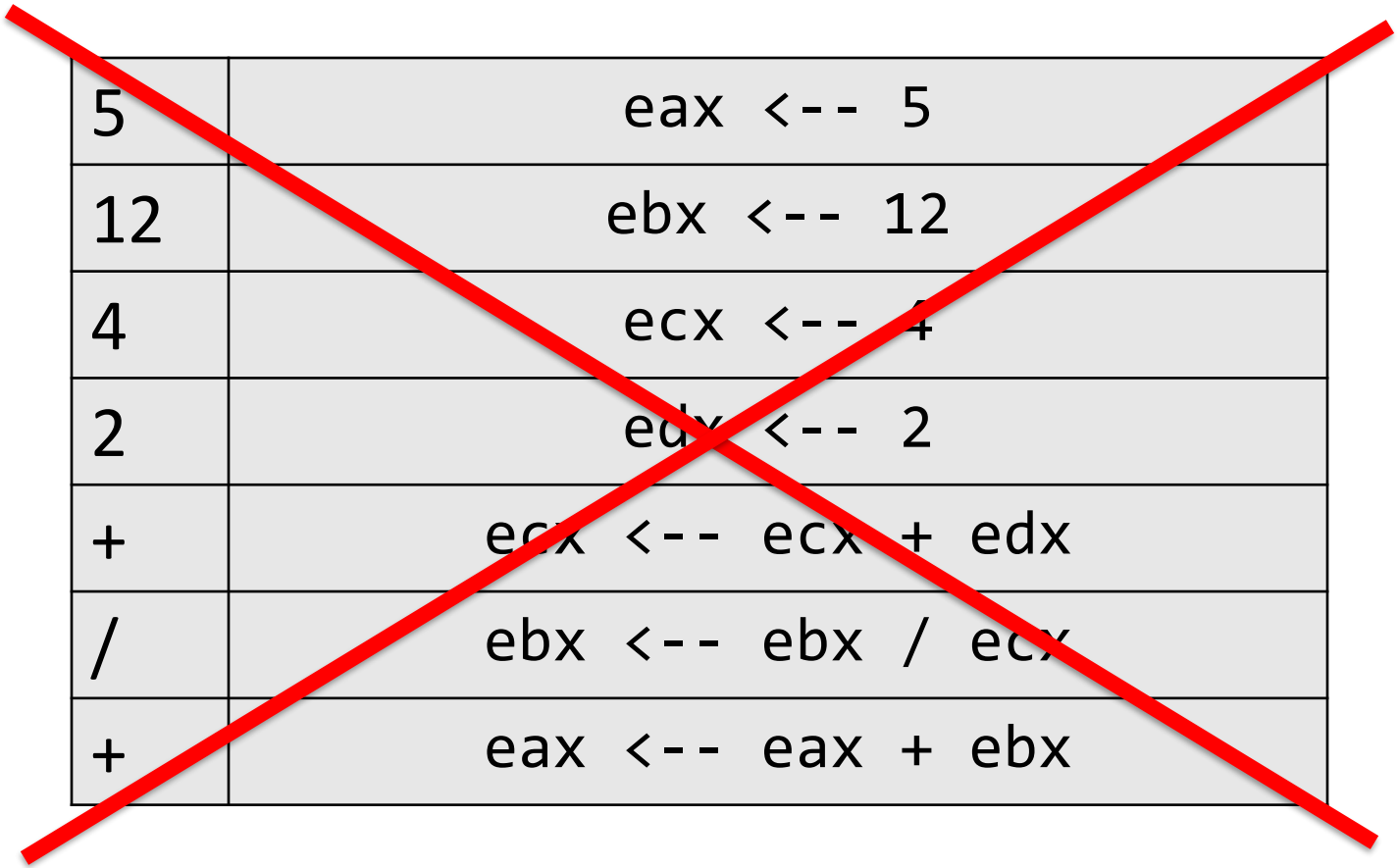


Post-Order Traversal: 5,12,4,2,+,/,+

Possible Solution?

- Always load an integer in a register
- Perform each operation as it comes up in the post-order traversal on the registers assigned to the lhs/rhs
- ??
- Profit!

In an ideal world…



5	eax <-- 5
12	ebx <-- 12
4	ecx <-- 4
2	edx <-- 2
+	ecx <-- ecx + edx
/	ebx <-- ebx / ecx
+	eax <-- eax + ebx

Problem: 4 Registers May Not Be Enough

- An expression that is a very imbalanced AST will need more than 4 registers…

- For instance:
- 5+12/(4+2***(5+3)**)

- This needs a minimum of **5** registers if we use that previous scheme.

Problem 2: Multiply/Divide in x86

- Addition/subtraction can be done with arbitrary registers.
- This is essentially eax += ebx:

```
add eax, ebx
```

- **Multiplication/division cannot be done with arbitrary registers.**
- Multiplication is between eax and the register you supply

- Multiplication results in a 64-bit number. The 32 least significant bits are stored in eax, the 32 most significant bits are stored in edx

- Eg. this is essentially edx:eax = eax * ebx:

```
imul ebx
```

Solution for Calc 3 Example…

- Integers are pushed onto the CPU stack
- When it's time to do an operation, values are popped off the stack, the result is computed and pushed back onto the stack
- This is essentially the way calculation was done in the first calc example, but now with assembly magics!

Code Generated by Calc 3

- Here is the x86 assembly generated for 5+12/(4+2)

```
push 5
push 12
push 4
push 2
pop ebx
pop eax
add eax,ebx
push eax
pop ebx
pop eax
cdq
idiv ebx
push eax
pop ebx
pop eax
add eax,ebx
push eax
```

Calc 3 Closer Look

- Let's take a look at calc 3 now…

5+12/(4+2) Assembly, Second Look

- The assembly calc 3 generated is actually pretty terrible:

```
push 5
push 12
push 4
push 2
pop ebx
pop eax
add eax,ebx
push eax
pop ebx
pop eax
cdq
idiv ebx
push eax
pop ebx
pop eax
add eax,ebx
push eax
```

Hand-Written x86

```
mov ebx,4
mov ecx,2
add ebx,ecx
mov eax,12
cdq
idiv ebx
mov ebx,5
add eax,ebx
```

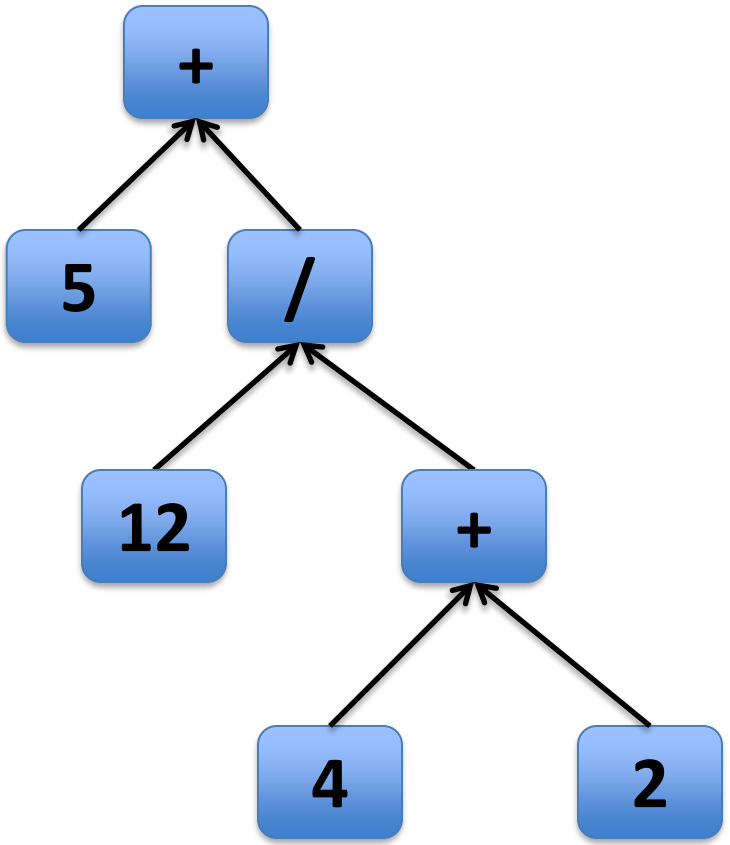
Virtual Registers

Virtual Registers

- Don’t assign the actual registers during the CodeGen pass, instead use ***virtual*** registers
- Assume we have an infinite number of virtual registers
- This makes it easier to code gen without worrying about register limitations

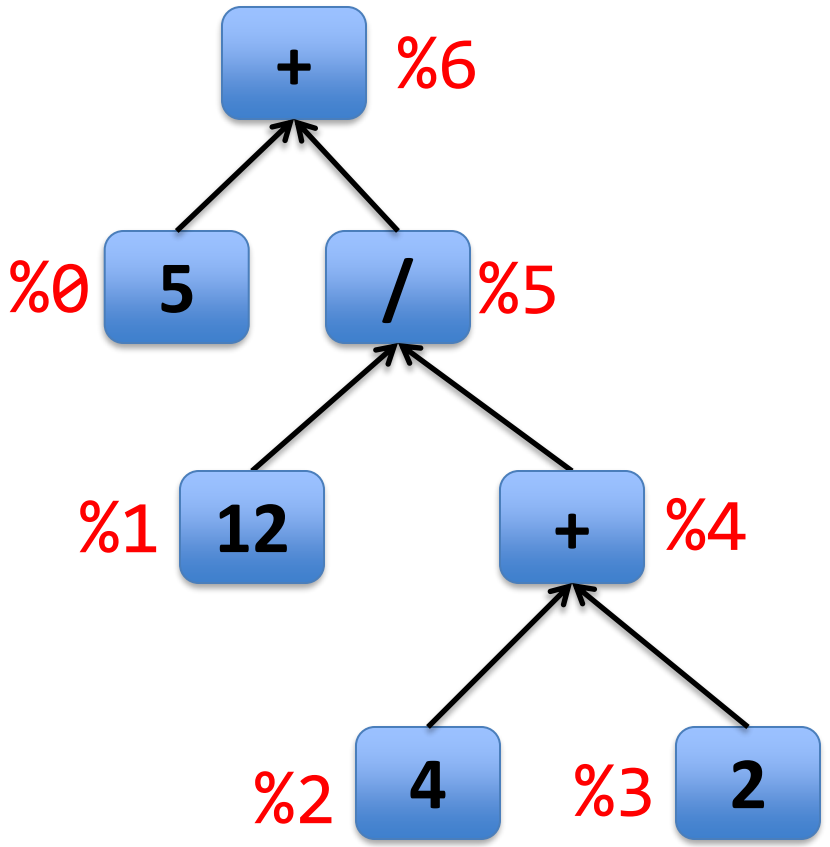
AST for 5+12/(4+2)

- During the CodeGen post-order traversal, we assign virtual registers as the nodes are generated



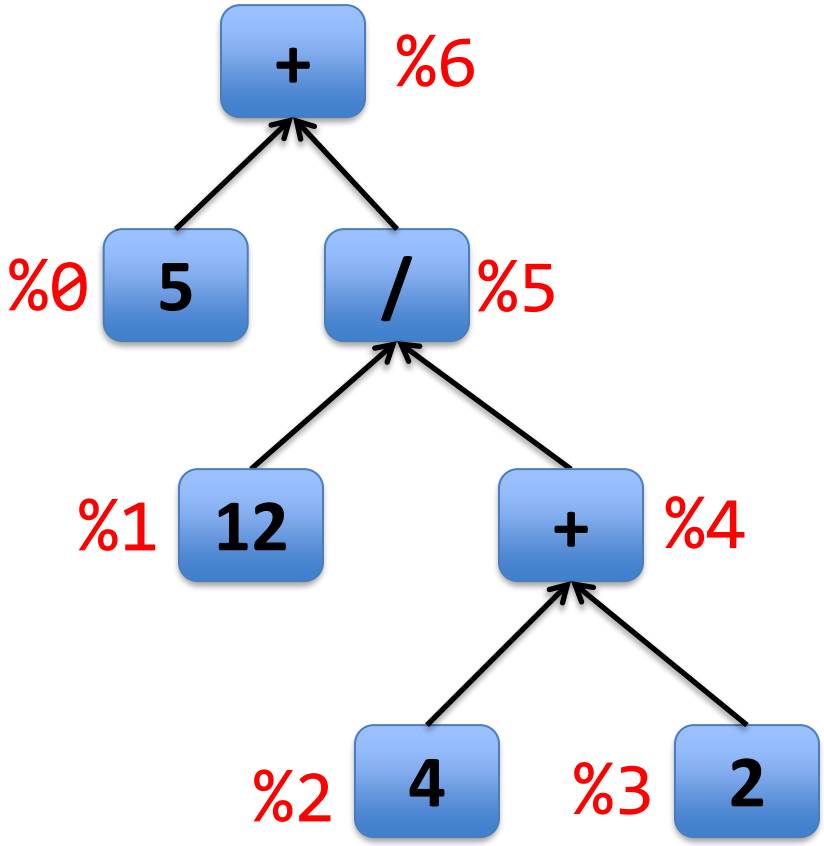
AST for 5+12/(4+2)

- During the CodeGen post-order traversal, we assign virtual registers as the nodes are generated
- Denoted by % in front of the number (just to distinguish it from a real register)



In ITP-11 Code…

```
movi %0,5
movi %1,12
movi %2,4
movi %3,2
add %4,%2,%3
div %5,%1,%4
add %6,%0,%5
```



Register Allocation Process

- When we do register allocation (in a bit), we will then need to convert virtual registers into actual registers on the target machine

- For ITP-11, we want to only allocate r1 through r7, since the other registers are used for special cases

```
movi %0,5
movi %1,12
movi %2,4
movi %3,2
add %4,%2,%3
div %5,%1,%4
add %6,%0,%5
```

CodeGen for Arrays

The Problem

- Suppose I have the following statements:

```
abc = 20;
a = 2
myarr[a] = abc * 5;
```

- ***How do I calculate the correct stack offset for this array index?***

Array Memory Address Calculation

- Given:
  - base = Base address where the array starts at (this is a known constant in our case)
  - subscript = subscript index you want (this could be a variable)
  - size = sizeof each element of type (in PA6 this is 1)

```
address = base + subscript * size
```

OR in PA6…

```
address = base + subscript
```

What's the subscript?

```
abc = 20;
a = 2
myarr[a] = abc * 5;
```

1. Code gen the expression inside of the [ ]. This will give you a subscript in a temp virtual register

2. movi the base address into another temp virtual register

3. Generate an add ***instruction*** between 1 and 2 to get the actual stack offset you want!

CodeGen for If and While

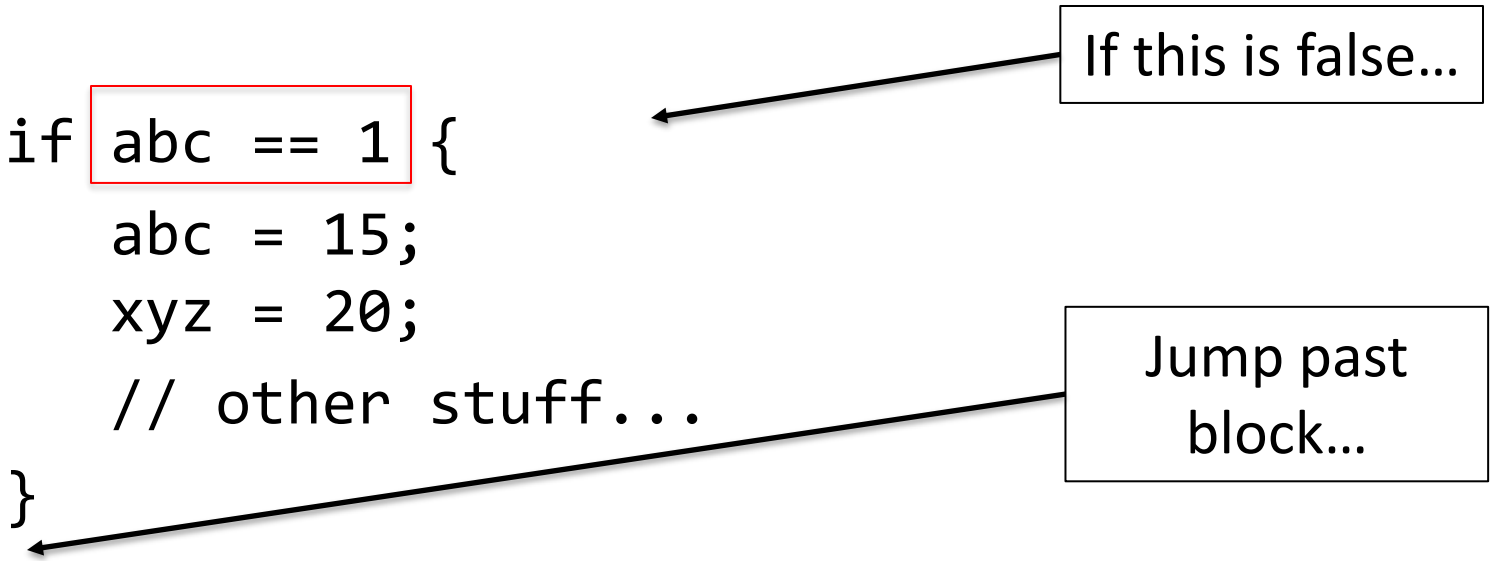
The Problem

- Jumps need to have the instruction number specified (cause they set pc = location)

```
if abc == 1 {
    abc = 15;
    xyz = 20;
    // other stuff...
}
```

- When we generate the condition, we don't know the target of the jump until we generate the block inside the if

The Problem



- Each statement may be several instructions long. We don't know how many until we generate the block

Solution

- Suppose we so far have 10 other instructions prior to the if

#	Instruction
9	. . .

- Next instruction will be at index 10, then

- (We'll just assume there's no prior virtual registers used)

Solution

- First, generate the condition, which:
- First generates the left hand side

```
if abc == 1 {
    abc = 15;
    xyz = 20;
}
```

#	Instruction
9	...
10	loadi %0,{abc}

- {abc} is the constant stack index we're using for abc, which we'd know cause we saved it in the CodeContext

Solution

- Next, generate the expression for the right hand side of the condition

```
if abc == 1 {
    abc = 15;
    xyz = 20;
}
```

#	Instruction
9	...
10	loadi %0,{abc}
11	movi %1,1

Solution

- Next, generate the comparison

```
if abc == 1 {
    abc = 15;
    xyz = 20;
}
```

#	Instruction
9	...
10	loadi %0,{abc}
11	movi %1,1
12	cmpeq %0,%1

Solution

- Now generate the “jump if not true”, with a placeholder address

```
if abc == 1 {
    abc = 15;
    xyz = 20;
}
```

#	Instruction
9	...
10	loadi %0,{abc}
11	movi %1,1
12	cmpeq %0,%1
13	movi %2,??
14	jnt %2

- Since we don’t know the address for the end of the if block yet, we leave the number for the movi blank (denoted by ??)

Solution

- Now generate the block of statements (which would each individually CodeGen…)

```
if abc == 1 {
    abc = 15;
    xyz = 20;
}
```

#	Instruction
9	...
10	loadi %0,{abc}
11	movi %1,1
12	cmpeq %0,%1
13	movi %2,??
14	jnt %2
15	movi %3,15
16	storei {abc},%3
17	movi %4,20
18	storei {xyz},%4

Solution

- Once we're done with the block, we know that the "if" block ends at instruction 18…
- So we should fix the movi to be one past the end, or 19!

#	Instruction
9	...
10	loadi %0,{abc}
11	movi %1,1
12	cmpeq %0,%1
13	movi %2,19
14	jnt %2
15	movi %3,15
16	storei {abc},%3
17	movi %4,20
18	storei {xyz},%4

If with else

- Same idea, just a little more gnarly:
  1. Generate if condition
  2. Generate jnt w/ placeholder address (to else block)
  3. Generate if block code
  4. Generate unconditional jmp w/ placeholder address (to past else block)
  5. Fix up address in #2
  6. Generate else block code
  7. Fix up address in #4

```
if abc < 37 {
    x = 1;
} else {
    x = 0;
}
```

While

- Similar as well:
  1. Generate while condition
  2. Generate jnt w/ placeholder address (to past while loop)
  3. Generate while block code
  4. Generate unconditional jmp back to first instruction of #1
  5. Fix up address in #2

```
while x < count {
    myarr[x] = x * 5;
    ++x;
}
```

Linear Scan Register Allocator

Register Allocation Process

- We need to convert virtual registers into actual registers on the target machine
- For ITP-11, we want to only allocate r1 through r7, since the other registers are used for special cases

```
movi %0,5
movi %1,12
movi %2,4
movi %3,2
add %4,%2,%3
div %5,%1,%4
add %6,%0,%5
```

Methods for Register Allocation

- Register allocation is an NP-hard problem
- There are several methods to allocate registers, including graph coloring
- We're going to use a fairly simple approach called ***linear scan***

Linear Scan: Code w/ Virtual Registers

- Suppose our code gen yielded the following sequence of instructions using virtual registers

#	Instruction
0	`push 0`
1	`movi %0,5`
2	`movi %1,12`
3	`movi %2,4`
4	`movi %3,2`
5	`add %4,%2,%3`
6	`div %5,%1,%4`
7	`add %6,%0,%5`
8	`storei 0, %6`

Linear Scan: Generate Intervals

- First, we need to generate the intervals for each virtual register, which tells us the first and last instruction a VR is used at

#	Instruction
0	`push 0`
1	`movi %0,5`
2	`movi %1,12`
3	`movi %2,4`
4	`movi %3,2`
5	`add %4,%2,%3`
6	`div %5,%1,%4`
7	`add %6,%0,%5`
8	`storei 0, %6`

VR	First	Last
%0	1	7
%1	2	6
%2	3	5
%3	4	5
%4	5	6
%5	6	7
%6	7	8

Linear Scan: The Scan

- Before we start
  - Virtual registers have no real register assignments
  - All real registers are available

VR	First	Last
%0	1	7
%1	2	6
%2	3	5
%3	4	5
%4	5	6
%5	6	7
%6	7	8

VR	Real
%0	
%1	
%2	
%3	
%4	
%5	
%6	

R	Available?
r1	true
r2	true
r3	true
r4	true
r5	true
r6	true
r7	true

Linear Scan: The Scan

- Next we loop from 0 to # of instructions, and at each instruction:
  - Decide if any VR just expired (we're at last for that VR) and if so, whichever real register it was assigned to is now free
  - Decide if any VR now is active (we're at first for that VR) and if so, assign first available real register

VR	First	Last
%0	1	7
%1	2	6
%2	3	5
%3	4	5
%4	5	6
%5	6	7
%6	7	8

Linear Scan: The Scan

- Instruction 0
  - Nothing is expired
  - Nothing is active yet

VR	First	Last
%0	1	7
%1	2	6
%2	3	5
%3	4	5
%4	5	6
%5	6	7
%6	7	8

VR	Real
%0	
%1	
%2	
%3	
%4	
%5	
%6	

R	Available?
r1	true
r2	true
r3	true
r4	true
r5	true
r6	true
r7	true

Linear Scan: The Scan

- Instruction 1
  - Nothing is expired

VR	First	Last
%0	1	7
%1	2	6
%2	3	5
%3	4	5
%4	5	6
%5	6	7
%6	7	8

VR	Real
%0	
%1	
%2	
%3	
%4	
%5	
%6	

R	Available?
r1	true
r2	true
r3	true
r4	true
r5	true
r6	true
r7	true

Linear Scan: The Scan

- Instruction 1
  - Nothing is expired
  - %0 is now active, so assign first available (r1)

VR	First	Last
%0	1	7
%1	2	6
%2	3	5
%3	4	5
%4	5	6
%5	6	7
%6	7	8

VR	Real
%0	r1
%1	
%2	
%3	
%4	
%5	
%6	

R	Available?
r1	false
r2	true
r3	true
r4	true
r5	true
r6	true
r7	true

Linear Scan: The Scan

- Instruction 2
  - Nothing is expired

VR	First	Last
%0	1	7
%1	2	6
%2	3	5
%3	4	5
%4	5	6
%5	6	7
%6	7	8

VR	Real
%0	r1
%1	
%2	
%3	
%4	
%5	
%6	

R	Available?
r1	false
r2	true
r3	true
r4	true
r5	true
r6	true
r7	true

Linear Scan: The Scan

- Instruction 2
  - Nothing is expired
  - %1 is now active, so assign first available (r2)

VR	First	Last
%0	1	7
%1	2	6
%2	3	5
%3	4	5
%4	5	6
%5	6	7
%6	7	8

VR	Real
%0	r1
%1	r2
%2	
%3	
%4	
%5	
%6	

R	Available?
r1	false
r2	false
r3	true
r4	true
r5	true
r6	true
r7	true

Linear Scan: The Scan

- Instruction 3
  - Nothing is expired
  - %2 is now active, so assign first available (r3)

VR	First	Last
%0	1	7
%1	2	6
%2	3	5
%3	4	5
%4	5	6
%5	6	7
%6	7	8

VR	Real
%0	r1
%1	r2
%2	r3
%3	
%4	
%5	
%6	

R	Available?
r1	false
r2	false
r3	false
r4	true
r5	true
r6	true
r7	true

Linear Scan: The Scan

- Instruction 4
  - Nothing is expired
  - %3 is now active, so assign first available (r4)

VR	First	Last
%0	1	7
%1	2	6
%2	3	5
%3	4	5
%4	5	6
%5	6	7
%6	7	8

VR	Real
%0	r1
%1	r2
%2	r3
%3	r4
%4	
%5	
%6	

R	Available?
r1	false
r2	false
r3	false
r4	false
r5	true
r6	true
r7	true

Linear Scan: The Scan

- Instruction 5
  - Both %2 and %3 expired, so set their real registers to available

VR	First	Last
%0	1	7
%1	2	6
%2	3	5
%3	4	5
%4	5	6
%5	6	7
%6	7	8

VR	Real
%0	r1
%1	r2
%2	r3
%3	r4
%4	
%5	
%6	

R	Available?
r1	false
r2	false
r3	true
r4	true
r5	true
r6	true
r7	true

Linear Scan: The Scan

- Instruction 5
  - Both %2 and %3 expired, so set their real registers to available
  - %4 is now active, so assign first available (r3)

VR	First	Last
%0	1	7
%1	2	6
%2	3	5
%3	4	5
%4	5	6
%5	6	7
%6	7	8

VR	Real
%0	r1
%1	r2
%2	r3
%3	r4
%4	r3
%5	
%6	

R	Available?
r1	false
r2	false
r3	false
r4	true
r5	true
r6	true
r7	true

Linear Scan: The Scan

- Instruction 6
  - %1 and %4 are now expired, so their real registers are available

VR	First	Last
%0	1	7
%1	2	6
%2	3	5
%3	4	5
%4	5	6
%5	6	7
%6	7	8

VR	Real
%0	r1
%1	r2
%2	r3
%3	r4
%4	r3
%5	
%6	

R	Available?
r1	false
r2	true
r3	true
r4	true
r5	true
r6	true
r7	true

Linear Scan: The Scan

- Instruction 6
  - %1 and %4 are now expired, so their real registers are available
  - %5 is now active, so assign first available (r2)

VR	First	Last
%0	1	7
%1	2	6
%2	3	5
%3	4	5
%4	5	6
%5	6	7
%6	7	8

VR	Real
%0	r1
%1	r2
%2	r3
%3	r4
%4	r3
%5	r2
%6	

R	Available?
r1	false
r2	false
r3	true
r4	true
r5	true
r6	true
r7	true

Linear Scan: The Scan

- Instruction 7
  - %0 and %5 are now expired, so r1/r2 are available

VR	First	Last
%0	1	7
%1	2	6
%2	3	5
%3	4	5
%4	5	6
%5	6	7
%6	7	8

VR	Real
%0	r1
%1	r2
%2	r3
%3	r4
%4	r3
%5	r2
%6	

R	Available?
r1	true
r2	true
r3	true
r4	true
r5	true
r6	true
r7	true

Linear Scan: The Scan

- Instruction 7
  - %0 and %5 are now expired, so r1/r2 are available
  - %6 is now active, so assign first available (r1)

VR	First	Last
%0	1	7
%1	2	6
%2	3	5
%3	4	5
%4	5	6
%5	6	7
%6	7	8

VR	Real
%0	r1
%1	r2
%2	r3
%3	r4
%4	r3
%5	r2
%6	r1

R	Available?
r1	false
r2	true
r3	true
r4	true
r5	true
r6	true
r7	true

Linear Scan: The Scan

- Instruction 8
  - %6 is now expired, so r1 is available

VR	First	Last
%0	1	7
%1	2	6
%2	3	5
%3	4	5
%4	5	6
%5	6	7
%6	7	8

VR	Real
%0	r1
%1	r2
%2	r3
%3	r4
%4	r3
%5	r2
%6	r1

R	Available?
r1	true
r2	true
r3	true
r4	true
r5	true
r6	true
r7	true

Linear Scan: The Scan

- Instruction 8
  - %6 is now expired, so r1 is available
  - Nothing new is active

VR	First	Last
%0	1	7
%1	2	6
%2	3	5
%3	4	5
%4	5	6
%5	6	7
%6	7	8

VR	Real
%0	r1
%1	r2
%2	r3
%3	r4
%4	r3
%5	r2
%6	r1

R	Available?
r1	true
r2	true
r3	true
r4	true
r5	true
r6	true
r7	true

Linear Scan: Result

- Given the code with virtual registers and the allocations, we simply replace the virtual registers with the real ones

#	Instruction
0	push 0
1	movi %0,5
2	movi %1,12
3	movi %2,4
4	movi %3,2
5	add %4,%2,%3
6	div %5,%1,%4
7	add %6,%0,%5
8	storei 0, %6

VR	Real
%0	r1
%1	r2
%2	r3
%3	r4
%4	r3
%5	r2
%6	r1

#	Instruction
0	push 0
1	movi r1,5
2	movi r2,12
3	movi r3,4
4	movi r4,2
5	add r3,r3,r4
6	div r2,r2,r3
7	add r1,r1,r2
8	storei 0, r1

Linear Scan: The Scan (Summary)

- Loop from 0 to # of instructions, and at each iteration we:
  - Decide if any VR just expired (we're at last for that VR) and if so, whichever real register it was assigned to is now free
  - Decide if any VR now is active (we're at first for that VR) and if so, assign first available real register

VR	First	Last
%0	1	7
%1	2	6
%2	3	5
%3	4	5
%4	5	6
%5	6	7
%6	7	8

VR	Real
%0	r1
%1	r2
%2	r3
%3	r4
%4	r3
%5	r2
%6	r1

Example from PA6: High-level Code

```
data {
    var x;
}
main {
    x = 10 + 5;
    x = x * 2;
    x = (x + 20) / 3;
}
```

In-class Activity

```
0: push r0
1: movi %0,10
2: movi %1,5
3: add %2,%0,%1
4: storei 0,%2
5: loadi %3,0
6: movi %4,2
7: mul %5,%3,%4
8: storei 0,%5
9: loadi %6,0
10: movi %7,20
11: add %8,%6,%7
12: movi %9,3
13: div %10,%8,%9
14: storei 0,%10
15: exit
```

What are the intervals of the virtual registers?

Example from PA6

```
push r0
movi %0,10
movi %1,5
add %2,%0,%1
storei 0,%2
loadi %3,0
movi %4,2
mul %5,%3,%4
storei 0,%5
loadi %6,0
movi %7,20
add %8,%6,%7
movi %9,3
div %10,%8,%9
storei 0,%10
exit
```

```
INTERVALS:
%0:1,3
%1:2,3
%2:3,4
%3:5,7
%4:6,7
%5:7,8
%6:9,11
%7:10,11
%8:11,13
%9:12,13
%10:13,14
```

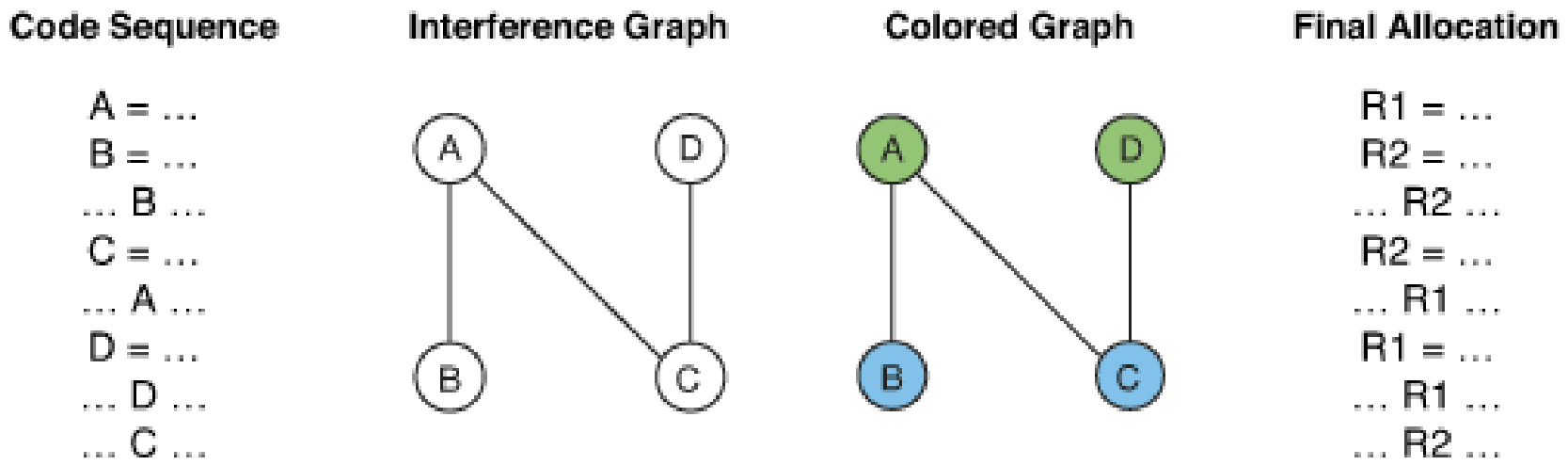
```
ALLOCATION:
%0:r1
%1:r2
%2:r1
%3:r1
%4:r2
%5:r1
%6:r1
%7:r2
%8:r1
%9:r2
%10:r1
```

```
push r0
movi r1,10
movi r2,5
add r1,r1,r2
storei 0,r1
loadi r1,0
movi r2,2
mul r1,r1,r2
storei 0,r1
loadi r1,0
movi r2,20
add r1,r1,r2
movi r2,3
div r1,r1,r2
storei 0,r1
exit
```

Spilling

- For a real program, we still might run out of registers
- Ultimately some stuff needs to be “spilled” into memory
- Don’t have to worry about this in our case

Graph Coloring Allocator



- We won't do this, but another popular method
- Convert intervals into interference graph and then colors graph
- Gives better allocations than linear scan, though more expensive to compute