



# COMP 520 - Compilers

## Lecture 19 – Register Minimalization and Dynamic Runtime Entities

# PA4 Annoucements: IDIV

- Won't be asked to do a divide operation that results in an exception
- Note: it is enough to zero RDX for the autograder
- It is better to implement an instruction called CQO

# PA4: Little Endian

- You can write 8, 4, and 2 bytes in proper little endian encoding using:

Bytes	Method
8	<code>x64.writeLong( _b, imm64 );</code>
4	<code>x64.writeInt( _b, imm32 );</code>
2	<code>x64.writeShort( _b, imm16 );</code>
1	<code>_b.write( imm8 );</code>

# PA4: Ending your program

- Implement `sys_exit( 0 )` like other syscalls
- Reminder: parameters are in registers just like the other system calls
- ALSO: `sys_write` shouldn't count the null terminator as a part of the string (so if outputting one byte, your size is 1, not 2 by including the null terminator)

# WA4 & WA5

- WA4: Helps visualize memory layouts in your x64 compiler. Fairly short, and should help with PA4.
- WA5: Minimizing registers in RISC. Covered Today!

# Course Evaluations

- Please complete your course evaluations sooner rather than later.
- Feedback will be helpful.



# Minimizing Register Usage in RISC

# Why RISC?

- Can make simplifying assumptions:
  - Cannot operate on memory locations directly

```
add [80000520+rcx*4], rdx
```



# Why RISC? (2)

- Can make simplifying assumptions:
  - Cannot operate on memory locations directly
  - Must spend one instruction to load from a memory address into a register
  - Must spend one instruction to store memory from a register into a memory location

```
lw $t0, @8000000520
```

```
sw $t0, @8000000520
```

# Why RISC? (3)

- Can make simplifying assumptions:
  - Cannot operate on memory locations directly
  - Must spend one instruction to load from a memory address into a register
  - Must spend one instruction to store memory from a register into a memory location
- Meaningful operations are done on registers/immediates

```
add $t0, $t0, $t1
```

# Comparison

## x64

```
add [80000520+rcx*4], rdx
```

## MIPS (reduced)

```
lis    $t0, 8000  
addi   $t0, 0520      ; t0 := 80000520  
sll    $t1, $t1, 2     ; t1 := t1 * 4  
add    $t0, $t0, $t1   ; t0 := t0 + t1  
lw     $t3, $t0        ; t3 := [t0]  
add    $t3, $t3, $t4   ; t3 := t3 + t4  
sw     $t3, $t0        ; [t0] := t3
```

# Why RISC? (4)

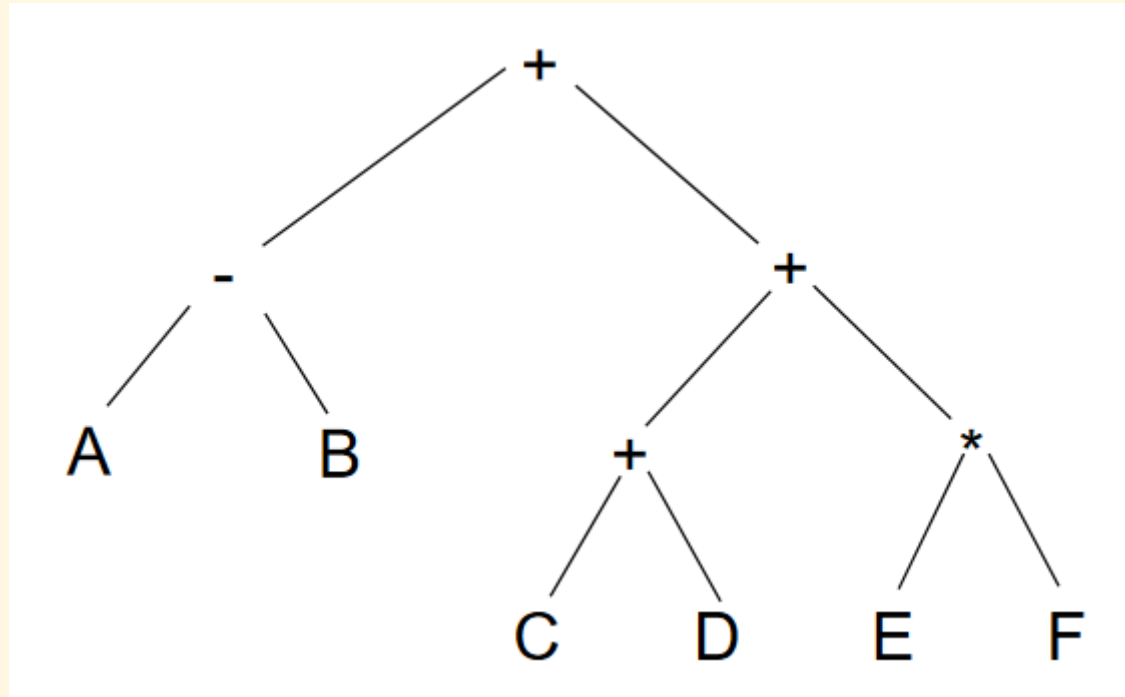
- RISC generalizes well
- Can apply RISC optimizations first, then apply additional optimizations specific to your ISA

# Goal

- Solve:  $(A - B) + ((C + D) + (E * F))$
- Goals:
  - Minimize the number of registers needed
  - Get the correct result

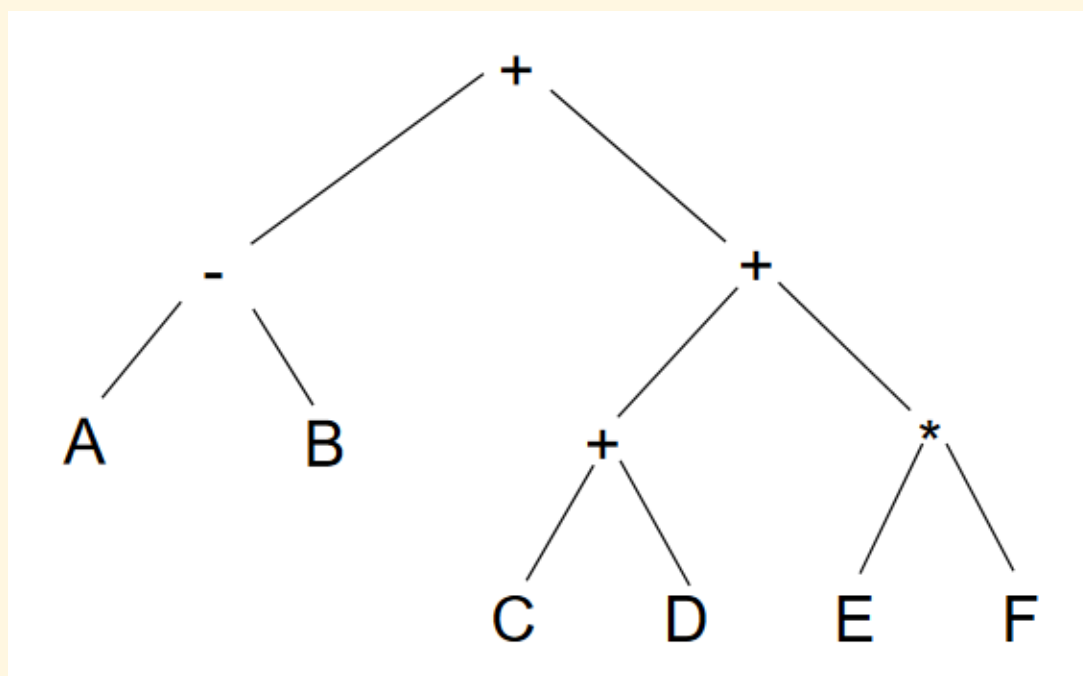
# Step 1: Create an AST

$(A - B) + ((C + D) + (E * F))$



# Step 2: Create Tuple Code (1)

$(A - B) + ((C + D) + (E * F))$



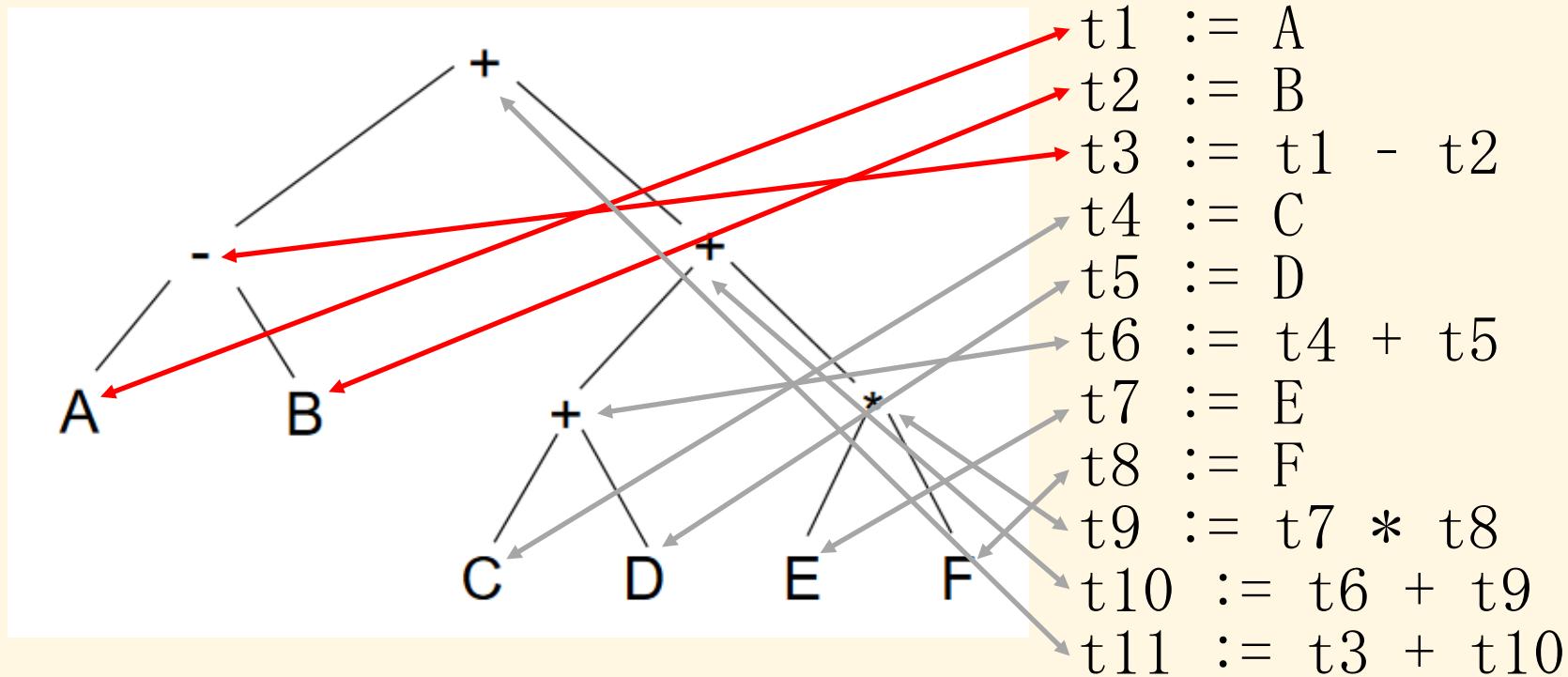
Idea: greedily assign all instructions a register  
Assume infinite registers.

```
t1 := A
t2 := B
t3 := t1 - t2
t4 := C
t5 := D
t6 := t4 + t5
t7 := E
t8 := F
t9 := t7 * t8
t10 := t6 + t9
t11 := t3 + t10
```

# Step 2: Create Tuple Code (2)

$(A - B) + ((C + D) + (E * F))$

Idea: greedily assign all instructions a register  
Assume infinite registers.

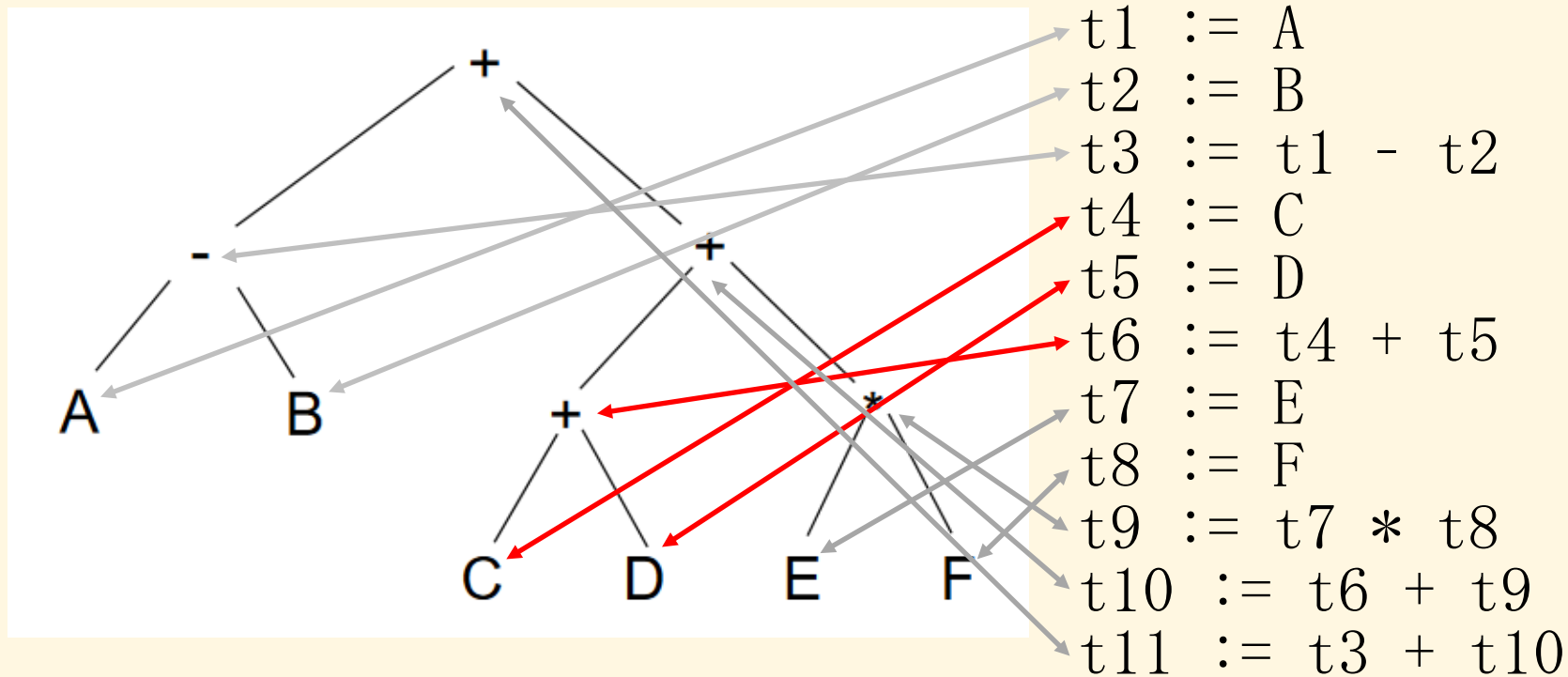




# Step 2: Create Tuple Code (3)

$(A - B) + ((C + D) + (E * F))$

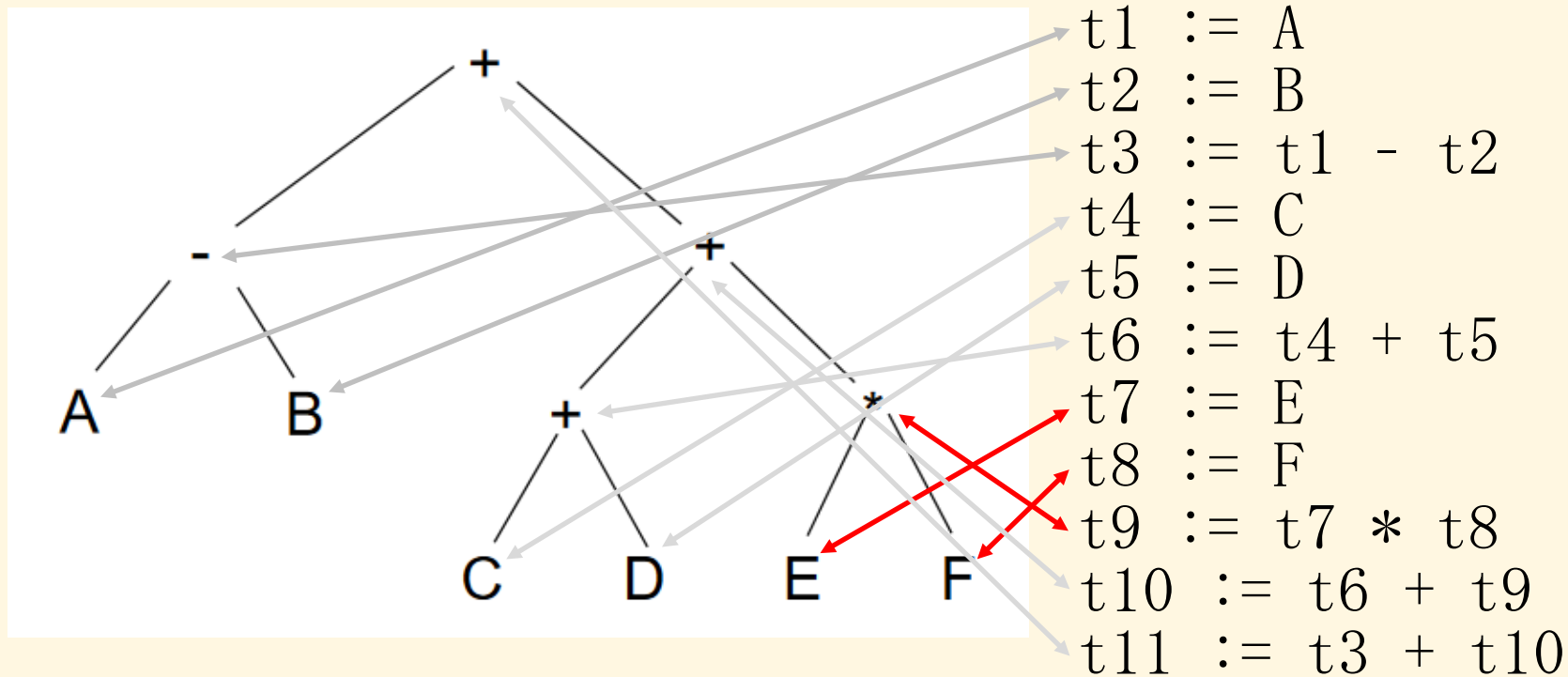
Idea: greedily assign all instructions a register  
Assume infinite registers.



# Step 2: Create Tuple Code (4)

$(A - B) + ((C + D) + (E * F))$

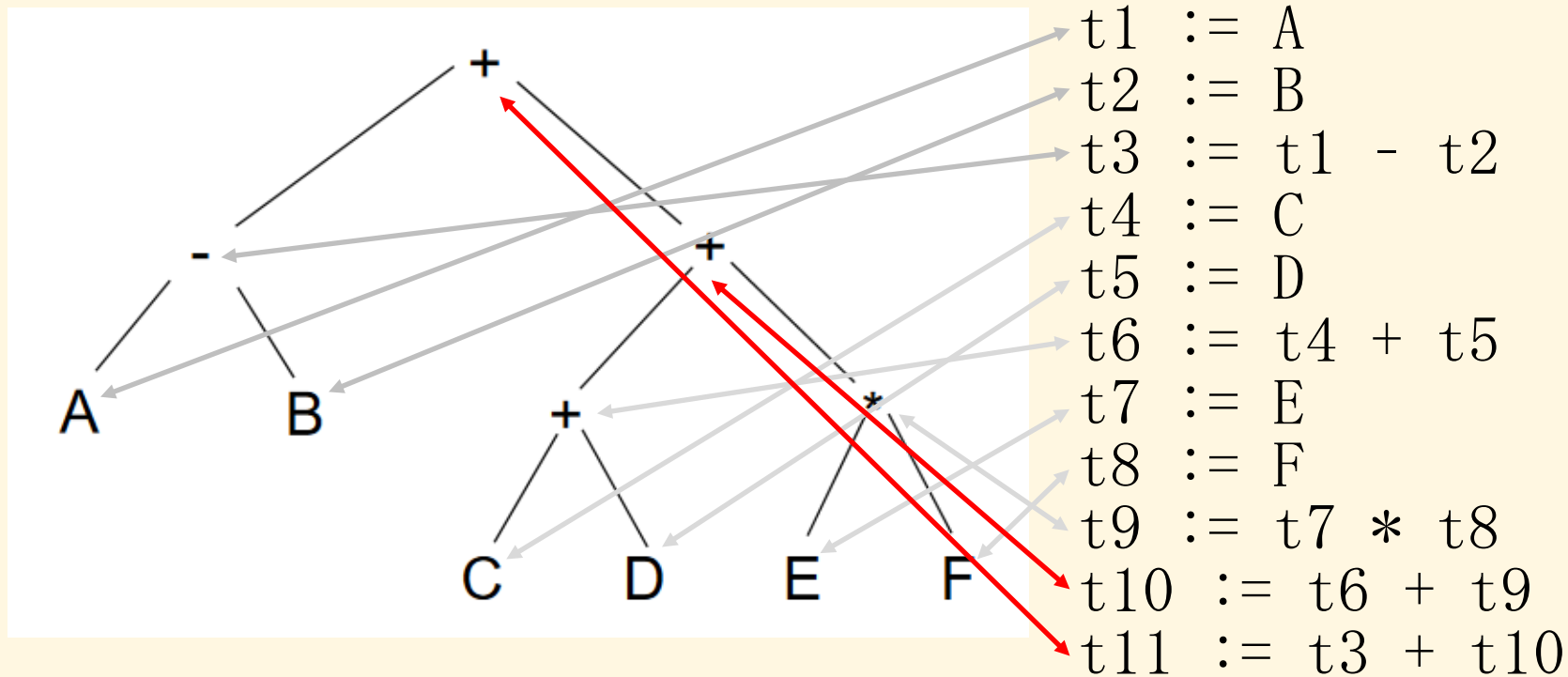
Idea: greedily assign all instructions a register  
Assume infinite registers.



# Step 2: Create Tuple Code (5)

$(A - B) + ((C + D) + (E * F))$

Idea: greedily assign all instructions a register  
Assume infinite registers.



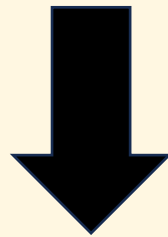
# Can use Data/Expression Liveness

$$(A - B) + ((C + D) + (E * F))$$

- This example is not as interesting.
- We have already seen data liveness.

# Let's look at a more interesting example

$$(A - B) + ((C + D) + (E * F))$$

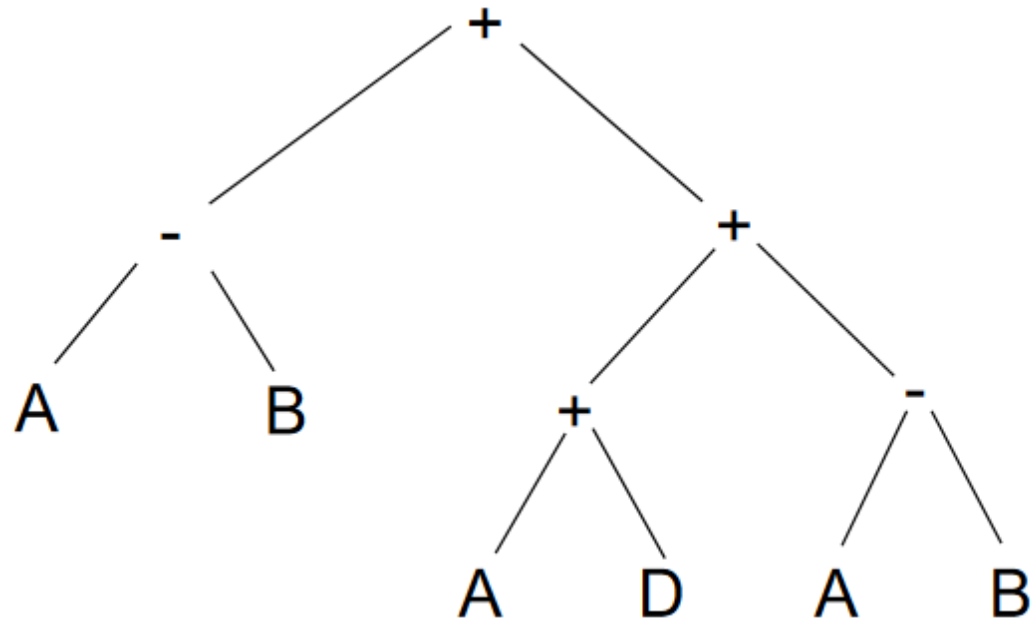


$$(A - B) + ((A + D) + (A - B))$$

Reused variables and expressions.

# Step 2: Generate Tuple Code

$(A - B) + ((A + D) + (A - B))$



$t1 := A$

$t2 := B$

$t3 := t1 - t2$

$t4 := A$

$t5 := D$

$t6 := t4 + t5$

$t7 := A$

$t8 := B$

$t9 := t7 - t8$

$t10 := t6 + t9$

$t11 := t3 + t10$

# Repeat 3: Replace Redundant Loads

```
t1 := A
t2 := B
t3 := t1 - t2
t4 := A
t5 := D
t6 := t4 + t5
t7 := A
t8 := B
t9 := t7 - t8
t10 := t6 + t9
t11 := t3 + t10
```

```
t1 := A
t2 := B
t3 := t1 - t2
t4 := t1
t5 := D
t6 := t4 + t5
t7 := t1
t8 := t2
t9 := t7 - t8
t10 := t6 + t9
t11 := t3 + t10
```

# Repeat 3: Replace Aliases

```
t1 := A
t2 := B
t3 := t1 - t2
t4 := t1
t5 := D
t6 := t4 + t5
t7 := t1
t8 := t2
t9 := t7 - t8
t10 := t6 + t9
t11 := t3 + t10
```

```
t1 := A
t2 := B
t3 := t1 - t2
t4 := t1
t5 := D
t6 := t1 + t5
t7 := t1
t8 := t2
t9 := t1 - t2
t10 := t6 + t9
t11 := t3 + t10
```



# Repeat 3: Replace Redundant Expressions

t1 := A

t2 := B

t3 := t1 - t2

t5 := D

t6 := t1 + t5

t9 := t1 - t2

t10 := t6 + t9

t11 := t3 + t10

t1 := A

t2 := B

t3 := t1 - t2

t5 := D

t6 := t1 + t5

t9 := t3

t10 := t6 + t9

t11 := t3 + t10

# Repeat 3: Replace Aliases

t1 := A  
t2 := B  
t3 := t1 - t2

t5 := D  
t6 := t1 + t5

t9 := t3  
t10 := t6 + t9  
t11 := t3 + t10

t1 := A  
t2 := B  
t3 := t1 - t2

t5 := D  
t6 := t1 + t5

~~t9 := t3~~  
t10 := t6 + t3  
t11 := t3 + t10

# Step 4: Reorder Numberings

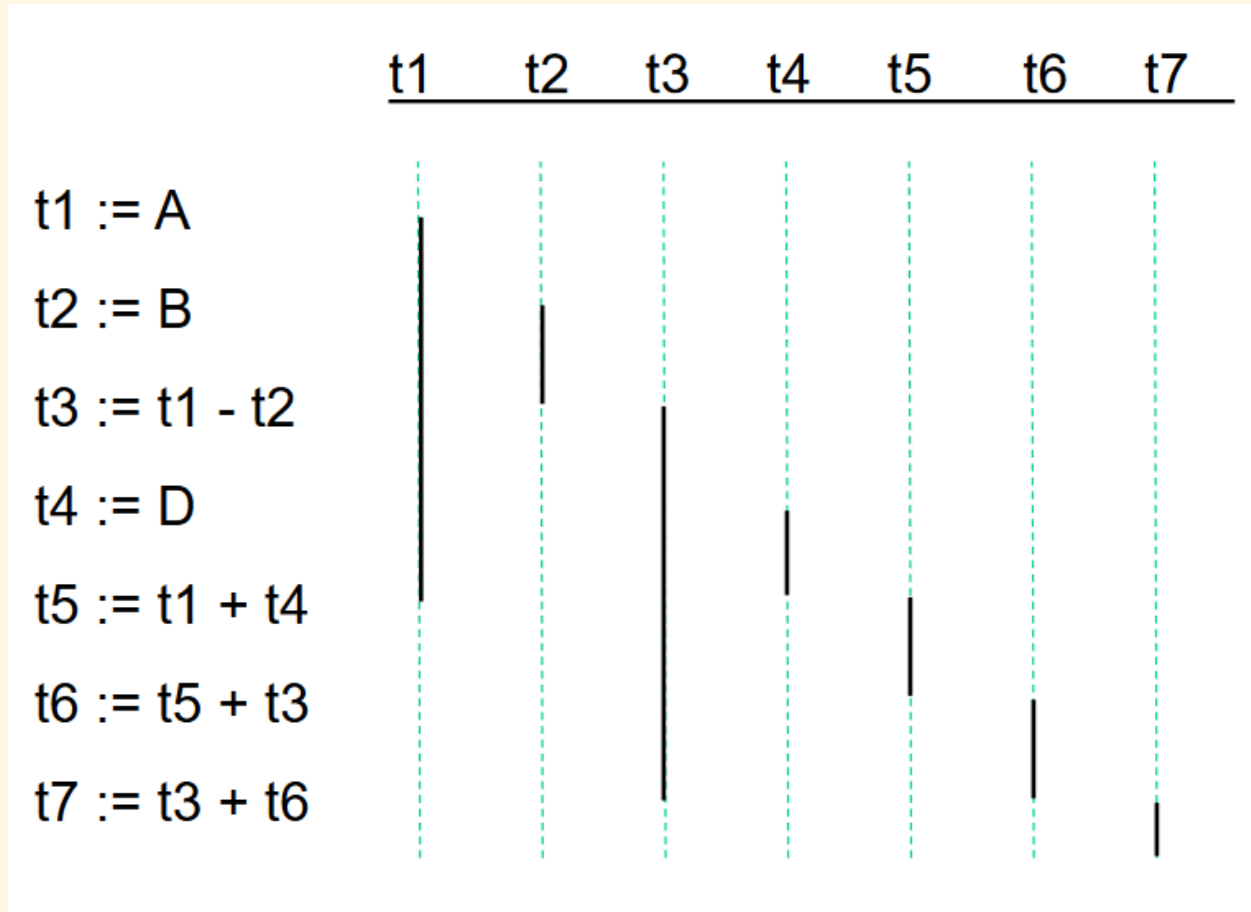
t1 := A  
t2 := B  
t3 := t1 - t2

t5 := D  
t6 := t1 + t5

t10 := t6 + t3  
t11 := t3 + t10

t1 := A  
t2 := B  
t3 := t1 - t2  
t4 := D  
t5 := t1 + t4  
t6 := t5 + t3  
t7 := t3 + t6

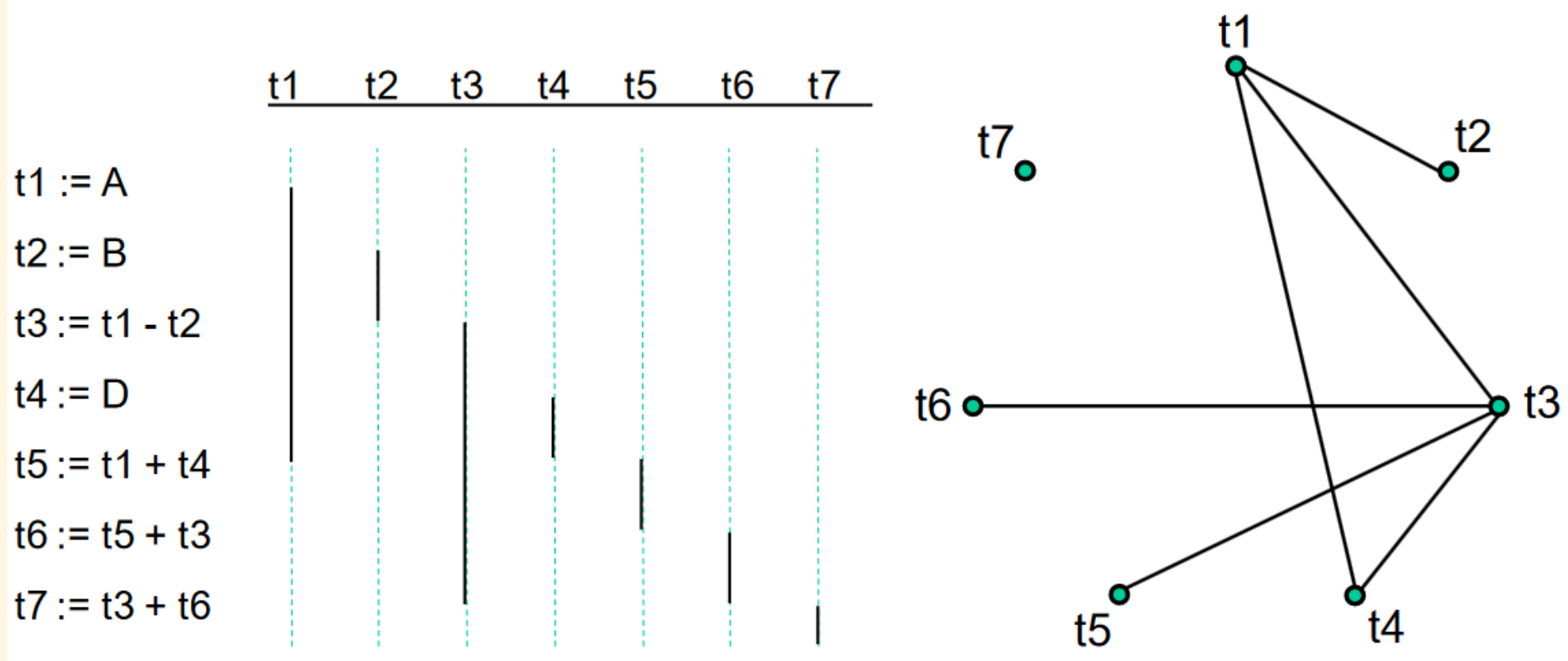
# Step 5: Apply Lifetime Analysis



# Prior to these slides...

- Idea of lifetime analysis was to “eyeball it”
- We need something more formal
- We will construct an interference graph
  - Give a node for each temporary register
  - Edge  $(u,v)$  if life range of  $u$  overlaps with live range of  $v$

# Step 6: Apply Interference Graph



$$G = (E, V)$$

$$\forall u, v : u, v \in V :: ((u, v) \in E) \leftrightarrow (\exists l :: \text{alive}(l, u) \wedge \text{alive}(l, v))$$

# k-Coloring

- $k$  is the number of registers available
- NP-complete problem for most architectures
- What does this mean for easily minimizing register usage?

# k-Coloring - Greedy

- Greedy Heuristic (may fail)
  - Pick a node where indegree  $< k$  when possible
  - Repeatedly remove node and all associated edges and add to an ordered list
  - When graph is empty, color nodes in reverse order
- If it fails:
  - Need to add in a temporary  $t$  and insert fetch and store operations for  $t$  when needed
  - Repeat lifetime analysis, interference analysis
  - Might still fail, and more repeating needed



# Kempe Algorithm (1879)

- Given  $G=(V,E)$
- Color nodes in  $V$  using  $k$  colors such that
  - $\forall (u, v) \in E :: \text{color}(u) \neq \text{color}(v)$

```
while  $G \neq \{\}$  do
  choose some minimum degree node  $t$  in  $V$ 
   $S.\text{push}(t)$ 
   $G.\text{remove}(t)$  // remove  $t$  and edges incident on  $t$  from  $G$ 
end
```

} simplify

```
while  $S \neq \{\}$  do
   $t = S.\text{pop}()$ 
   $G.\text{insert}(t)$  // add  $t$  and edges in  $G$  incident on  $t$ 
  color  $t$ , if possible
end
```

} color

# Kempe – Simplify Step

Stack of nodes  $S$  initially empty

Undirected graph  $G = (V, E)$


**while**  $G \neq \{\}$  **do**

    choose some minimum degree

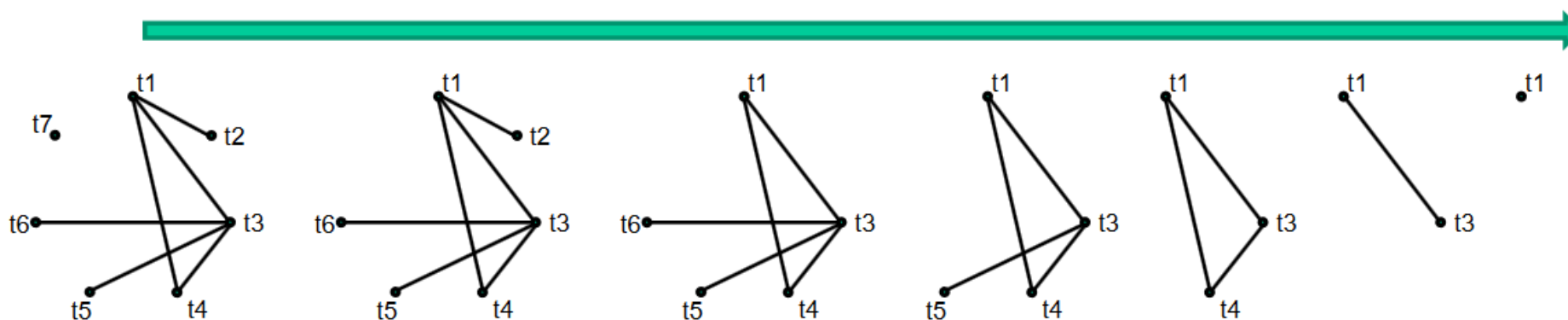
    node  $t$  in  $V$

$S.push(t)$

$G.remove(t)$

$S =$    $t1$   
 $t3$   
 $t4$   
 $t5$   
 $t6$   
 $t2$   
 $t7$

$G =$



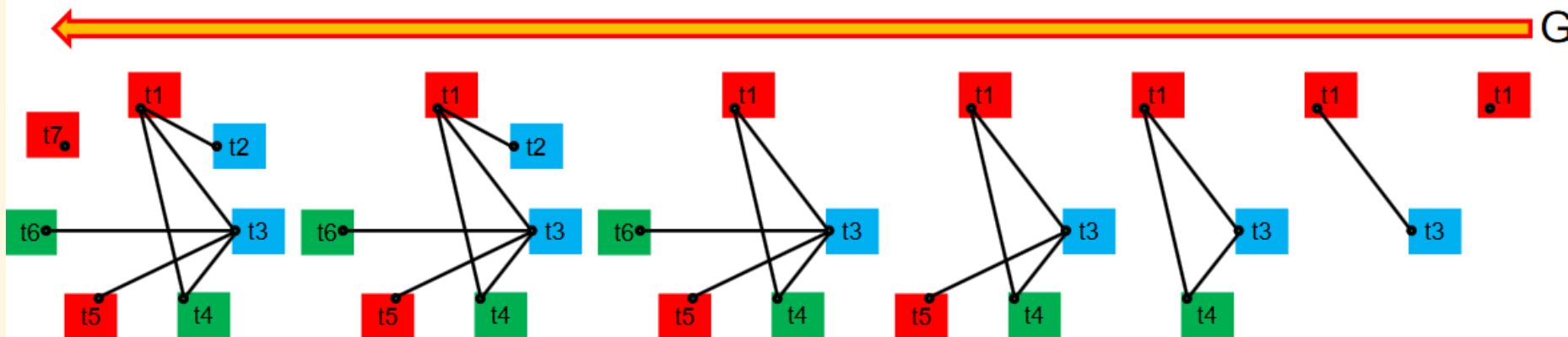
# Kempe – Color Step

Stack of nodes  $S$  to color in order  
Undirected graph  $G = (V, E)$  initially empty

```
while  $S \neq \{\}$  do
   $t = s.pop()$ 
   $G.insert(t)$  // add  $t$  and edges in  $G$  incident on  $t$ 
  color  $t$ , if possible
end
```

$k = 3$  colors: ■ ■ ■

$S =$  ↓  $t1$   
 $t3$   
 $t4$   
 $t5$   
 $t6$   
 $t2$   
 $t7$



# Kempe – Rewrite Step

t1 := A

t2 := B

t3 := t1 - t2

t4 := D

t5 := t1 + t4

t6 := t5 + t3

t7 := t3 + t6



r1 := A

r2 := B

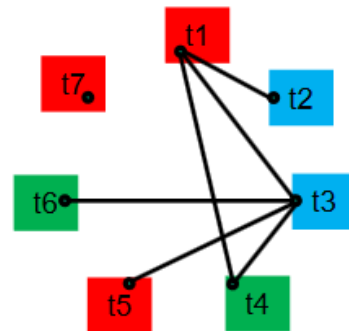
r2 := r1 - r2

r3 := D

r1 := r1 + r3

r3 := r1 + r2

r1 := r2 + r3



r1

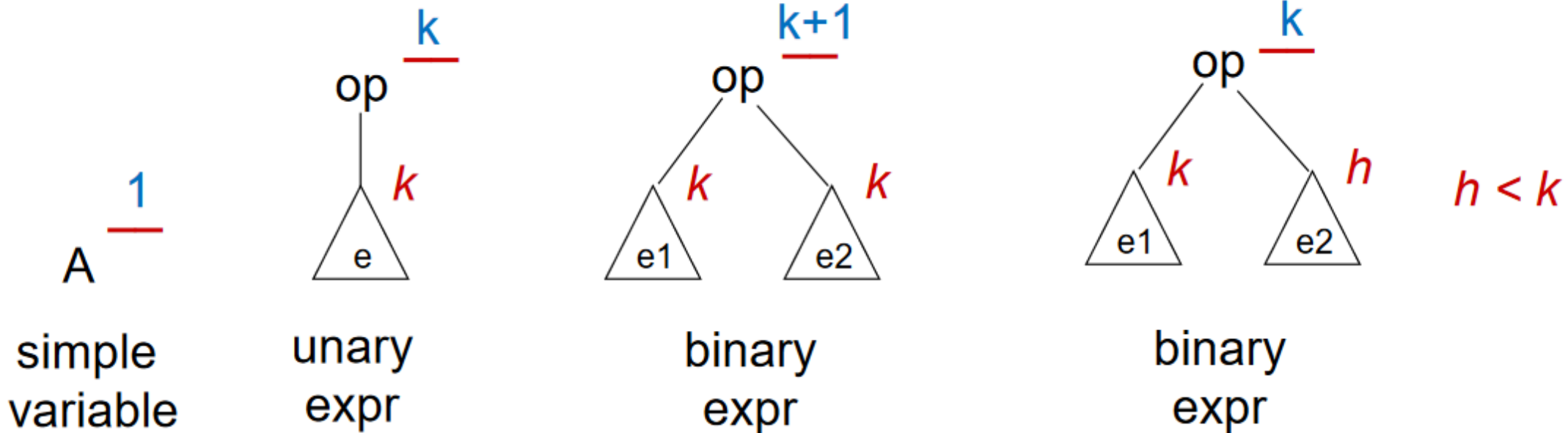
r2

r3

# Process Overview

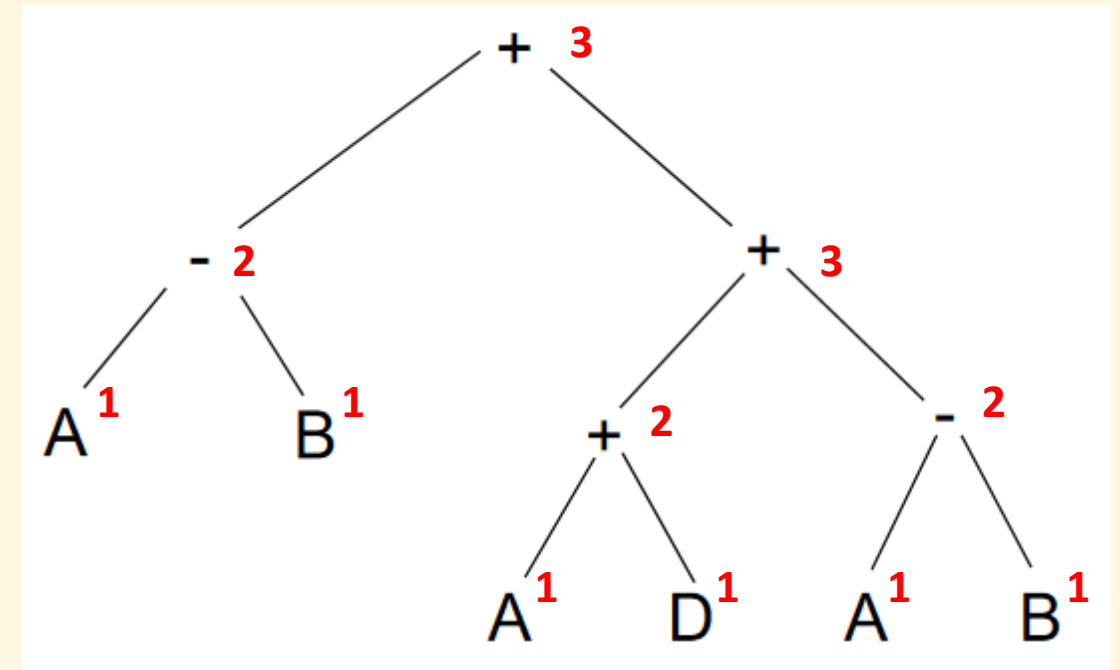
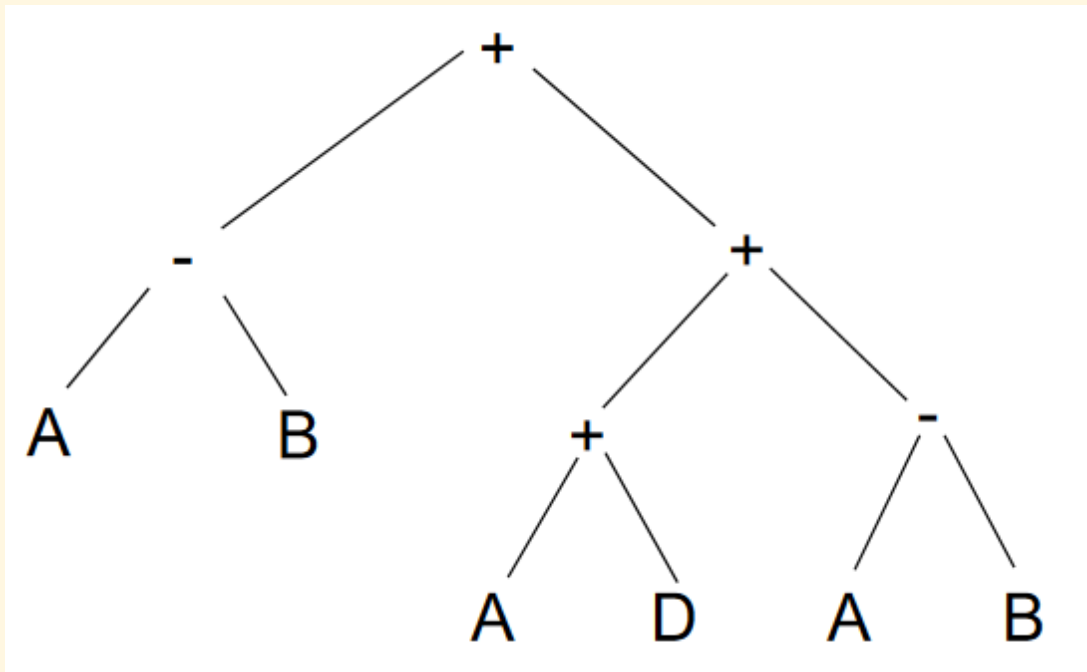
1. AST -> Tuple Code Generation
2. Tuple Code -> Lifetime Analysis
3. Interference Graph
  - If failure, then generate new temporaries (“spill code”), and start from step 2 again with load/stores placed
4. Graph -> New Tuple Code with Registers
5. Generate bytecode

# Register Number: Sethi-Ullman Labeling

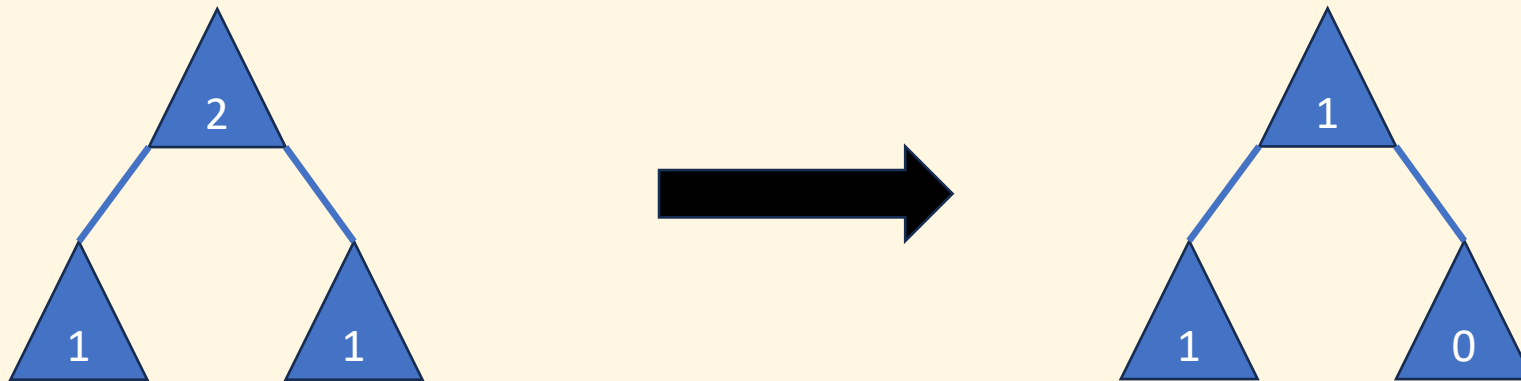


# Sethi-Ullman Example

$$(A - B) + ((A + D) + (A - B))$$



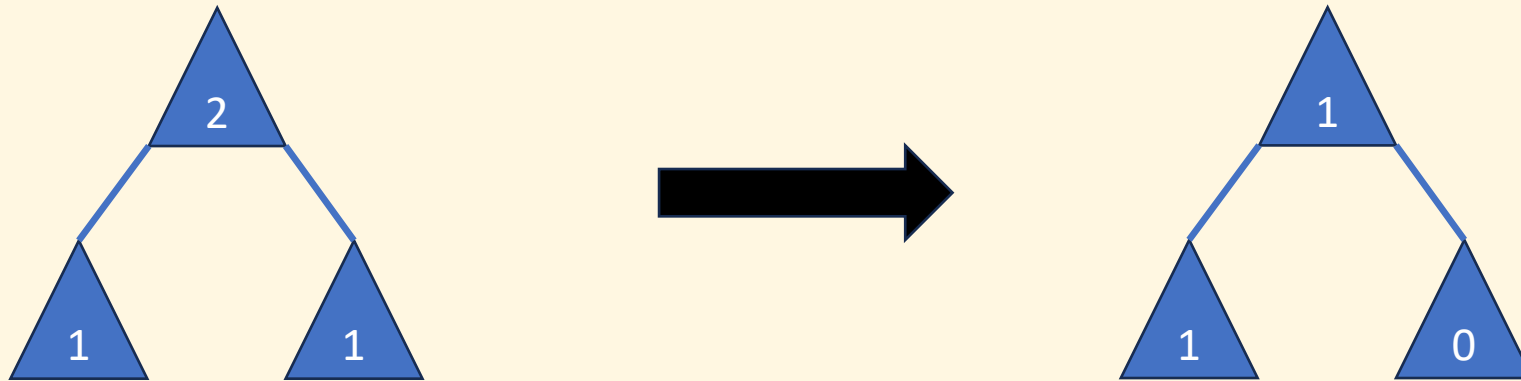
# CISC Sethi-Ullman



Why?



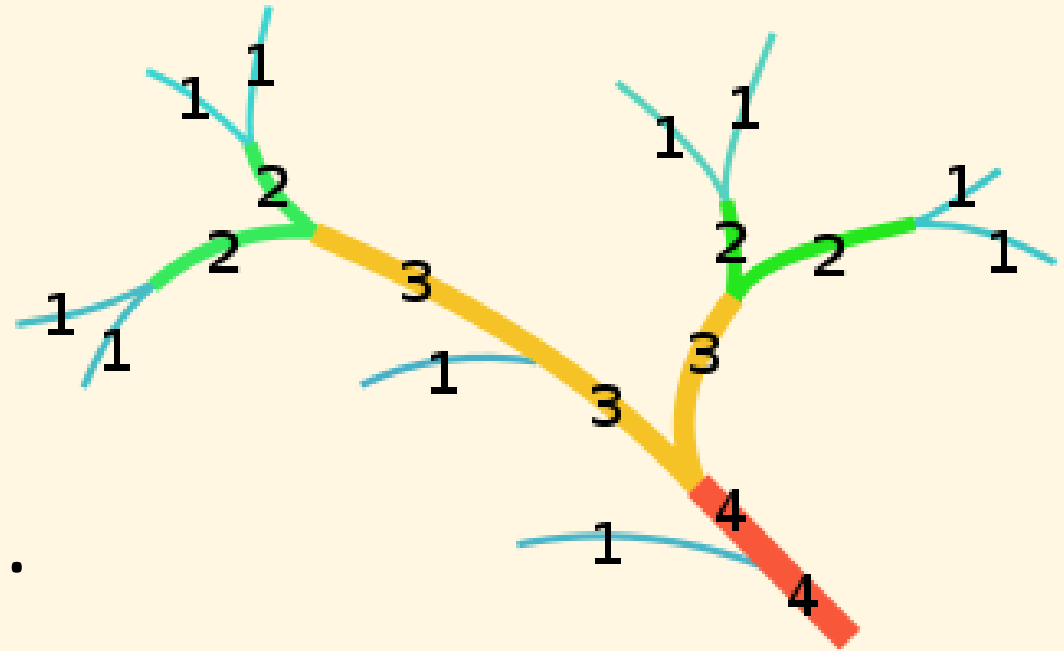
# CISC Sethi-Ullman



While it looks simple enough, difficult to tell if the choice is optimal or not (Either in the same expression or later, what if that variable is used when we used a 0-cost memory operand?)

# Strahler Number

- The number obtained by Sethi-Ullman is called the Strahler Number
- However, recall that Sethi-Ullman is a linear-time complexity algorithm.
- If K-coloring is NP-complete, then what does that say about Sethi-Ullman?



# Why not bother with x64?

- How many registers does it take to evaluate the following expression and push it on the stack?

$$3 + x$$

# Final bit of Optimization

- Recall our expression evaluation relied on the stack
- Results in code like this:  $(3+x)$

push 3

push [x]                      (a.k.a. push [rbp-8])

pop rax

pop rcx

add rax, rcx

push rax

# Final bit of Optimization

- Recall our expression evaluation relied on the stack
- Results in code like this: (3+x)

```
push 3  
push [...]  
pop rax  
pop rcx  
add rax, rcx  
push rax
```

**Condense consecutive push/pop**

```
mov rax, 3  
mov rcx, [...]  
add rax, rcx  
push rax
```

# Final bit of Optimization

- Recall our expression evaluation relied on the stack
- Results in code like this: (3+x)

```
push 3  
push [rbp-8]  
pop rax  
pop rcx  
add rax, rcx  
push rax
```

Condense consecutive push/pop

```
mov rax, 3  
mov rcx, ...  
add rax, rcx  
push rax
```

Does rcx's lifetime end here?

```
mov rax, 3  
add rax, [rbp-8]  
push rax
```

Do such operands by memory

# Final bit of Optimization

- Recall our expression evaluation relied on the stack
- Results in code like this: (3+x)

```
push 3  
push [rbp-8]  
pop rax  
pop rcx  
add rax, rcx  
push rax
```

Condense consecutive push/pop

```
mov rax, 3  
mov rcx, ...  
add rax, rcx  
push rax
```

Does rcx's lifetime end here?

```
mov rax, 3  
add rax, [rbp-8]  
push rax
```

**Condense Constants  
/ Rewrite**  
**push [rbp-8]**  
**add [rsp],3**

**Create this variable first,  
then apply constant operations**

# It is material you have seen before!

- By learning RISC optimization techniques, you can apply these towards CISC.
  - Requires just a few modifications to learned techniques!
  - (Condensing constants, creating stack temporaries, ...)
- Some optimizations however are hardware-specific...



# That's it for optimization

- If you are interested in the topic, see the [agner.org](https://agner.org) documentation on how to optimize architectures like x64
- Some really interesting stuff about how “very specific instructions, when side-by-side, can be combined into one instruction and reduce latency in x64”
- Example: **Instruction Fusion** of CMP, TEST, ADD, SUB, AND, OR, XOR, INC, DEC paired with a conditional jump is actually one instruction from the processor's perspective.
- Example: XOR rax,rax is actually a **zero time** instruction



# Using Libraries

An abrupt change of topics. Originally intended to be a different lecture, but we are running short on time remaining in the semester.

# For a change of pace: Windows

- Goal: learn what the file headers for windows are
- Goal: learn how to “import” code from another binary
- Why? Well, why reinvent the wheel?
- See PA4: it is very limited in memory allocation and output capabilities
- We want things like printf, file IO, etc.

# Portal Executable Format (haha)

- Windows uses an MZ header and a PE header
- The MZ header was used by DOS programs in the strict 16-bit era (LE was the 16/32-bit era)
- COM was another header type in the 16-bit era that is reserved for “extremely simple executables”

# MZ Header

- Named after Mark Zbikowski
- Still needed by Windows.. for some reason..
- It does one of the most meaningful things to the Windows loader: tell you where it ends

# MZ Header

- It does one of the most meaningful things to the Windows loader: tell you where it ends

Name	Offset + Size	Purpose
Signature	0x00 (2 bytes)	Must be 'MZ'
Header Size	0x08 (2 bytes)	Size of this header
PE Header Location	0x3C (4 bytes)	An addition to the original MZ header. Specifies where in the file is the PE header.

# Backwards Compatibility

## MZ Header

Largely ignored now.  
Helps load old 16-bit binaries.

## DOS Stub

“This program cannot be run  
on your version of Windows”  
(If you try to run it on DOS).

## PE Header

Modern headers for Windows.

# Sections, Segments, Linux, Windows

- Linux Reminder: segments are a chunk of memory with some properties (RWX, init from file, etc.) while sections can subdivide the segments for special purposes.

Segment: Load 0x1000 bytes from file, Readable, Writeable, Executable			
text: RX	data: RW	tls: RW	...



# Sections, Segments, Linux, Windows

- In windows, segments are referring to segmentation in 32-bit mode.
- Thus, when it comes to mapping memory locations, everything is defined by sections, and not segments.
- What do you think? Is it useful to define segments AND sections, or Windows approach for just sections?

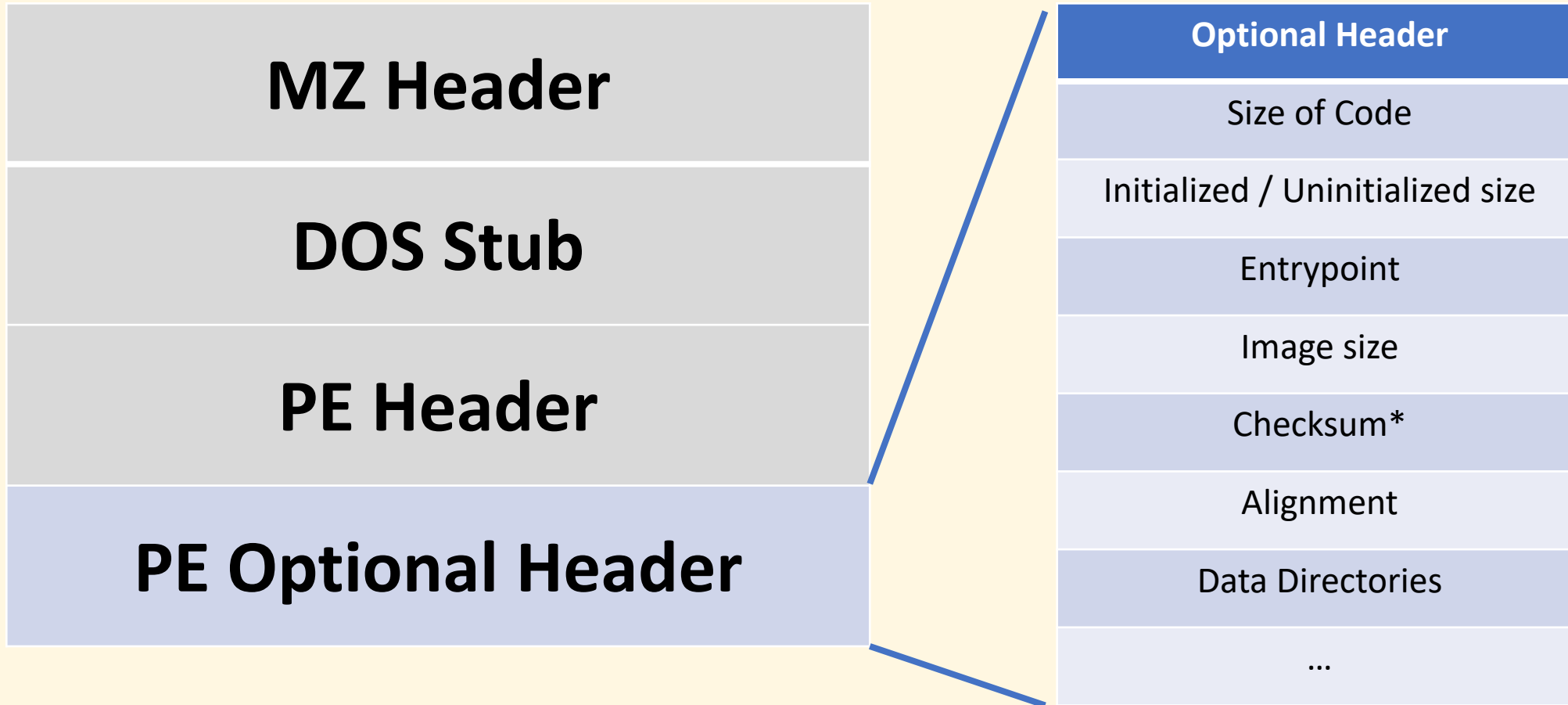
# PE Header Characteristics

- Only contains critical information
  - What machine is it for? i386?
  - How many sections are defined?
  - Is there an optional header?
- Optional Header?

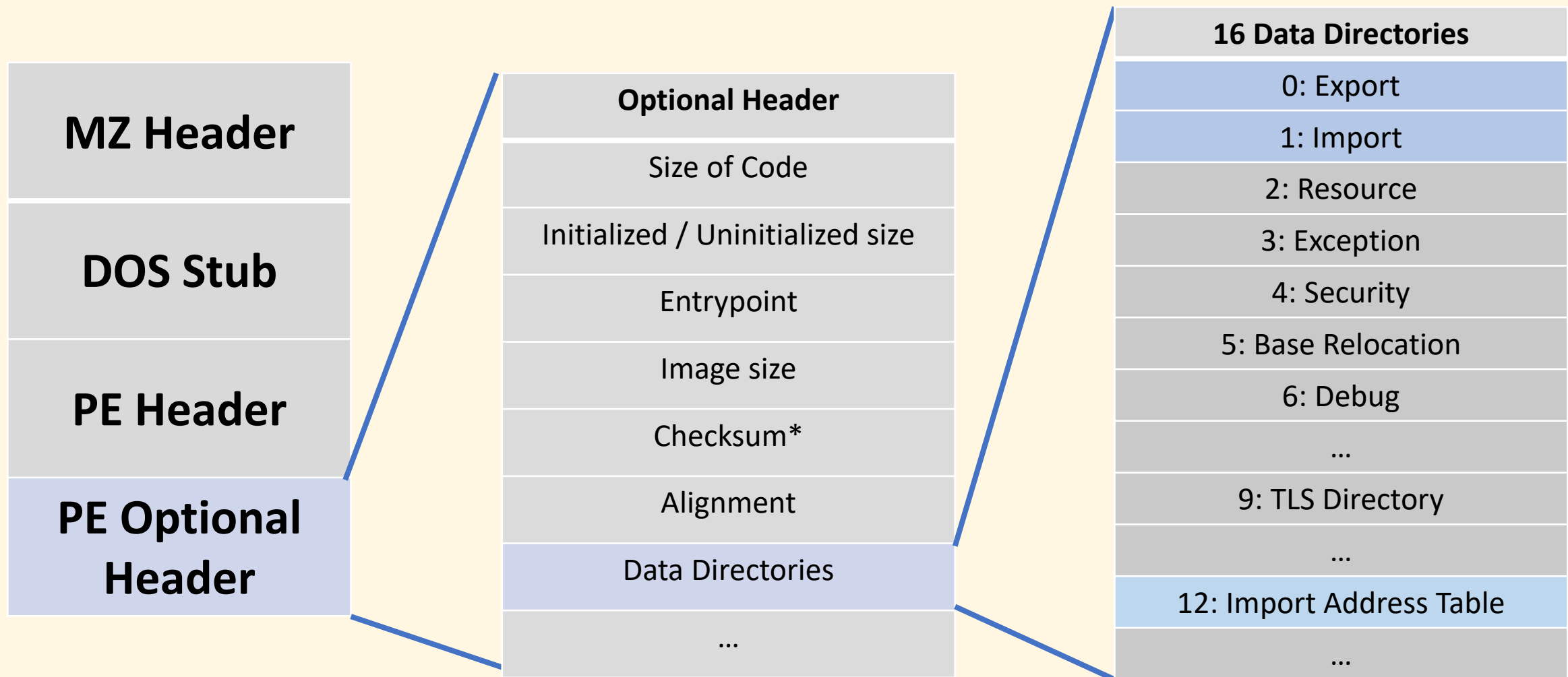
# PE Optional Header

- Not so optional usually
- Contains entrypoint, size of the code, data, alignment, **imports and exports**

# Finding out where are the imports



# Finding out where are the imports (2)

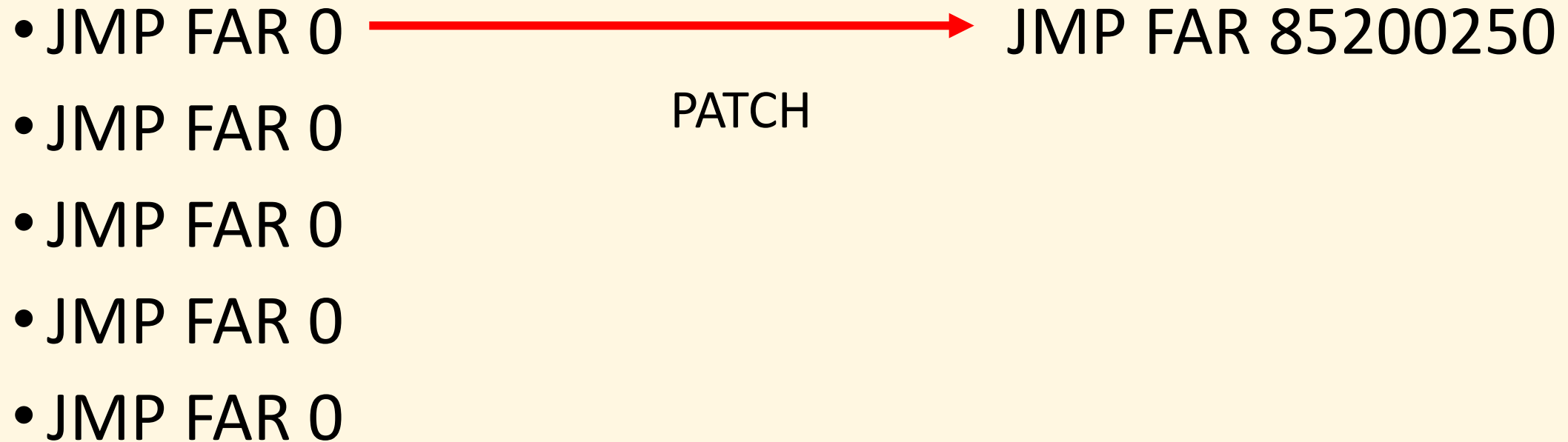


# Import Address Table (IAT)

- It looks exactly like an unpatched jump instruction
- JMP FAR 0
- JMP FAR 0
- JMP FAR 0
- JMP FAR 0
- JMP FAR 0

# Import Address Table (IAT)

**On runtime...**



# During Runtime

- “I need to call printf”
- “The first entry in the IAT is printf”
- CALL [IAT+0]



# During Runtime

- “I need to call printf”
- “The **first entry** in the IAT is printf”

- ... Code here ...

- **CALL [IAT+0]**

- ... Next Instruction ...

JMP FAR 55200250

**JMP FAR 85200250**

JMP FAR C5200250

Real Procedure

...

...

...

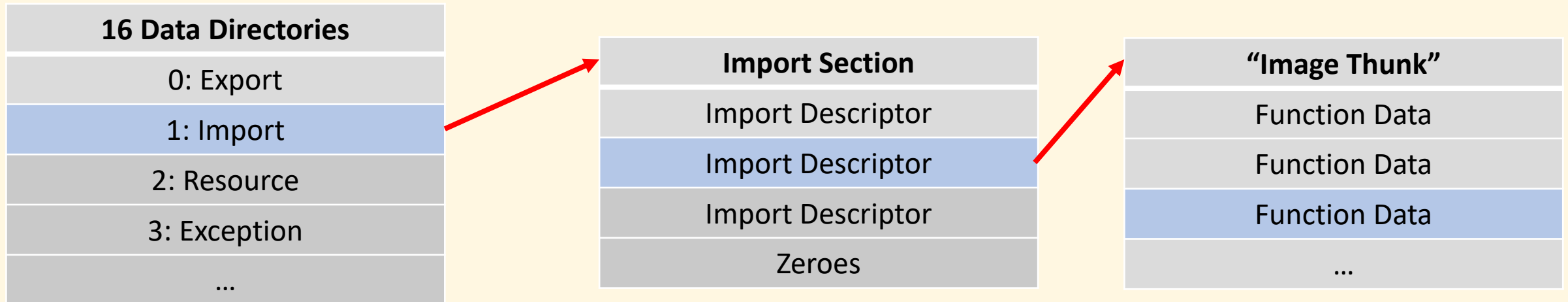
...

...

**ret**

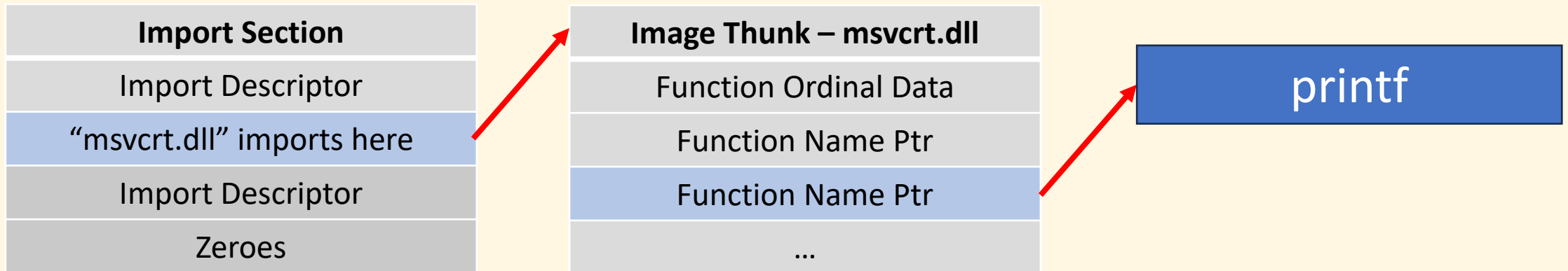
# How does the OS know where the real procedure is?

- This is done through the Import data directory



# Function Data

- Import by ordinal or name



# To bring it all together

- If “msvcrt.printf” is the 20<sup>th</sup> function import
- Then it corresponds to the 20<sup>th</sup> entry in the IAT
- Some simplifications are made,  
but that is the general idea.

# Imports don't have to be done in PE

- See MSDN:
  - LoadLibrary( "msvcrt.dll" )
  - GetProcAddress( hDll, "printf" )

# Side Discussion: What is a packer?

- Many things can be done on runtime. What if you compress your executable code?
- Then, specify a decompression algorithm as the entrypoint, decompress the data, then jump to it.
- Useful for hiding your code, makes it only analyzable at runtime.

# Anti-Tamper Software

- Removes all imports (does them manually)
- Encrypts all of the executable
- Decrypts only parts when needed
- “Phones home” randomly
- Makes various CRC checks



# Reminder: Course Evaluations



# Reminder: Course Evaluations

- If you found some lectures/PA checkpoints particularly effective, let us know
- If you found some lectures/PA checkpoints particularly difficult, let us know
- If you have something nice to say, or not, it is worth saying to improve this course for subsequent students

# Reminder: Course Evaluations

- I'm still learning to teach, but that isn't an excuse to not give course feedback
- All feedback is valuable
- Hope to see you next week and good luck on PA4!

End







