NTNU – Trondheim
Norwegian University of
Science and Technology

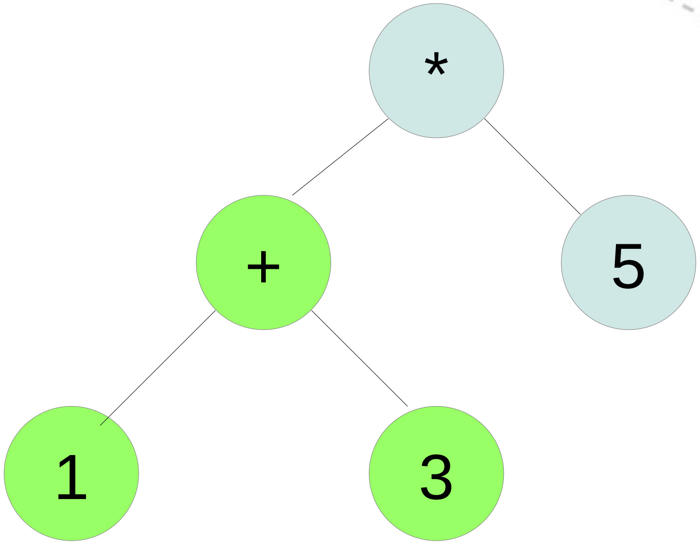
Simple CPU design and the run-time stack

Where we left off

- We have translated expressions, statements, conditions and loops into TAC
- We stopped at function parameters, call and return
- I’d like to dwell on those for a bit, because their implementation attaches to CPU design specifics

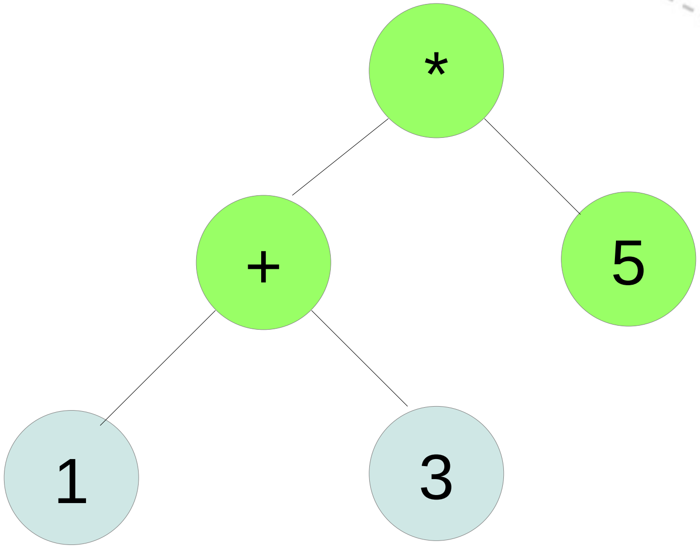
From tree to TAC

t1 = 1
t2 = 3
t3 = t1+t2



From tree to TAC

t1 = 1
t2 = 3
t3 = t1+t2
t4 = 5
t5 = t3*t4



A very simple CPU

- Suppose we have a machine with
  - A register to track its position in the program (**P**rogram **C**ounter)
  - Three slots for numbers (A, B, C)
  - Some memory
  - Operations to load, store, and combine values in registers

PC 0	A 0
B 0	C 0

NTNU – Trondheim
Norwegian University of
Science and Technology

From TAC to operations

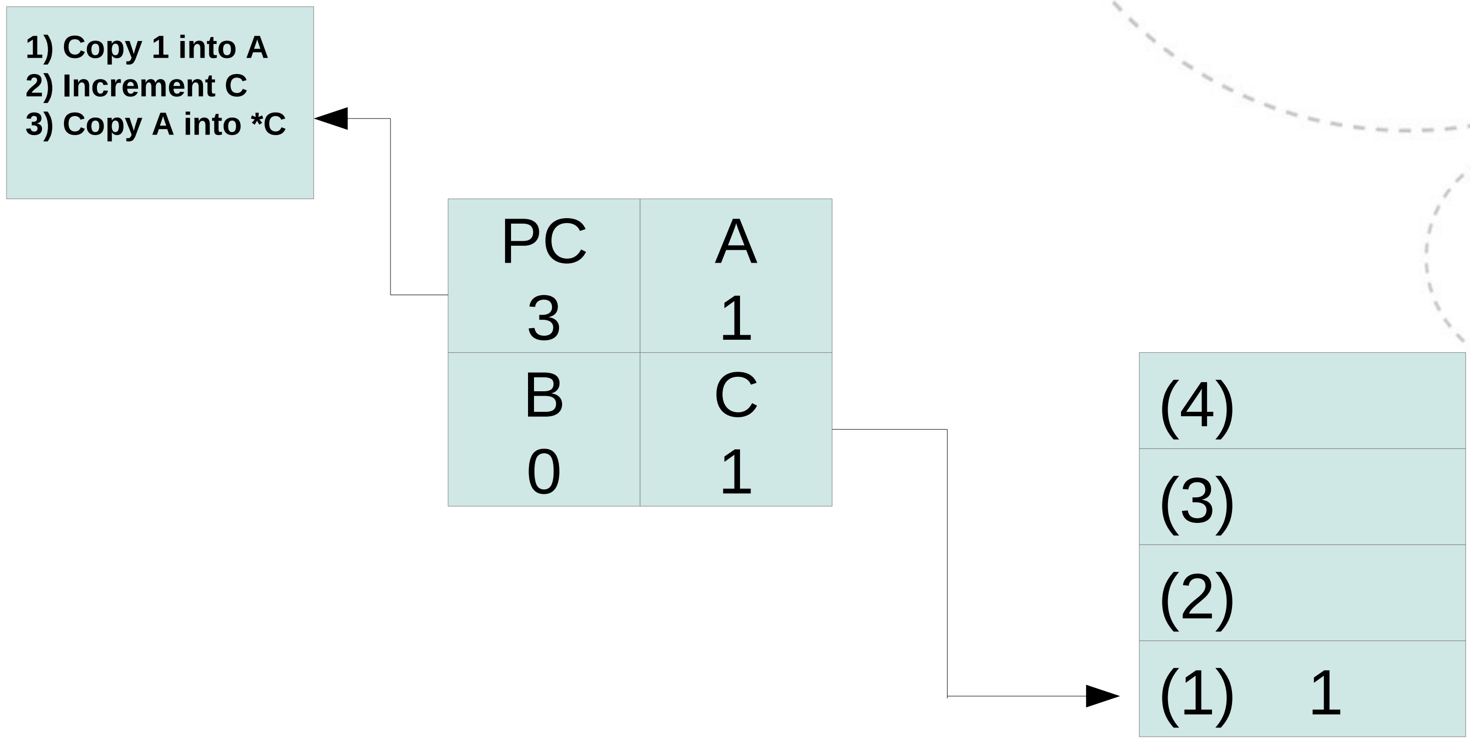
1) t1 = 1
2) t2 = 3
3) t3 = t1+t2
4) t4 = 5
5) t5 = t3*t4

PC	A
1	0
B	C
0	0

(4)
(3)
(2)
(1)

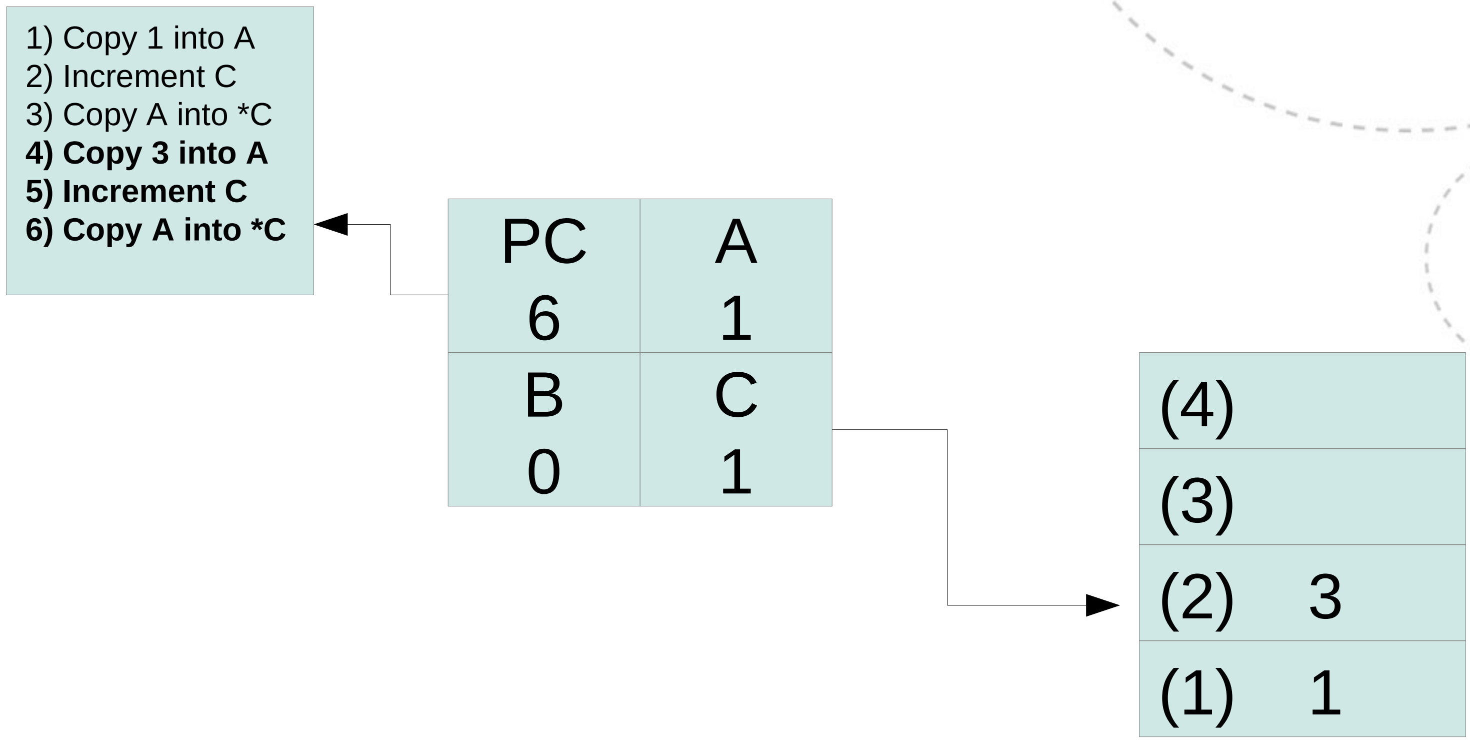
First step on a simple machine

t1 = 1
t2 = 3
t3 = t1+t2
t4 = 5
t5 = t3*t4



Another step much like it

t1 = 1
t2 = 3
t3 = t1+t2
t4 = 5
t5 = t3*t4



Evaluation of an intermediate result

t1 = 1
t2 = 3
t3 = t1+t2
t4 = 5
t5 = t3*t4

1) Copy 1 into A
2) Increment C
3) Copy A into *C
4) Copy 3 into A
5) Increment C
6) Copy A into *C
7) Copy *C into A
8) Decrement C
9) Copy *C into B
10) Decrement C

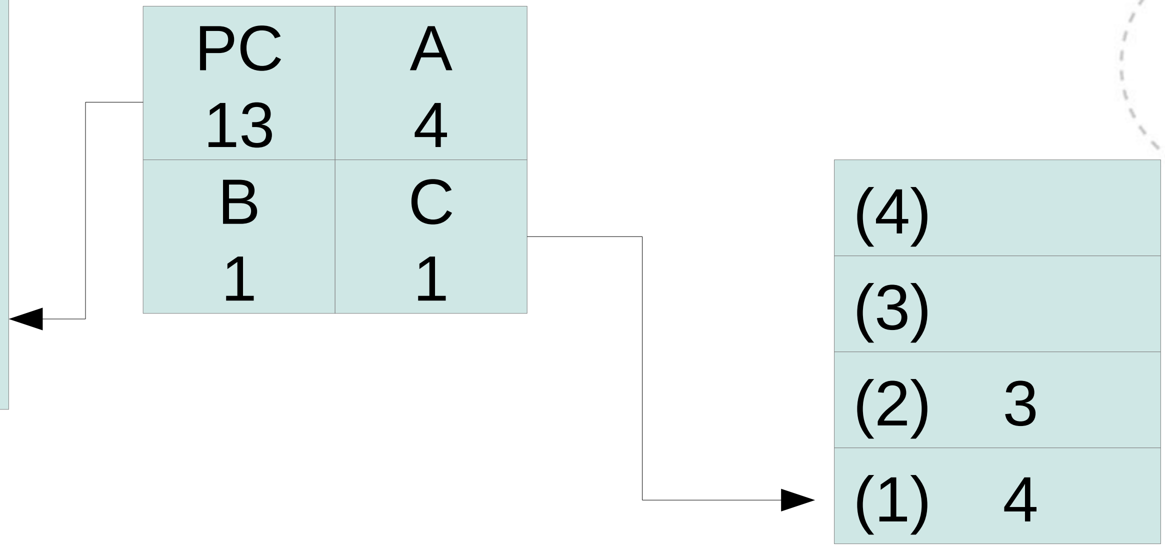
PC	A
10	3
B	**C**
1	0

(4)	
(3)	
(2)	3
(1)	1

Evaluation of an intermediate result

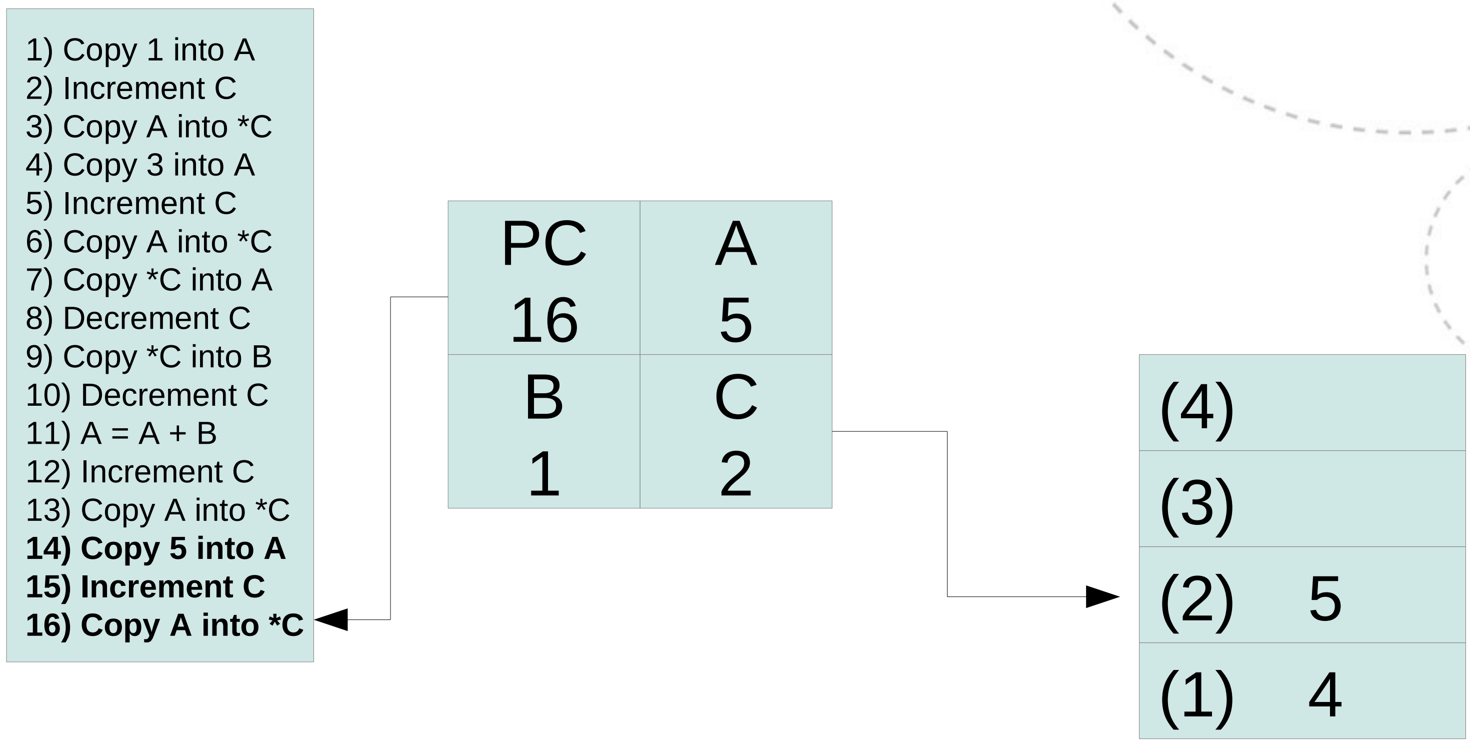
t1 = 1
t2 = 3
t3 = t1+t2
t4 = 5
t5 = t3*t4

1) Copy 1 into A
2) Increment C
3) Copy A into *C
4) Copy 3 into A
5) Increment C
6) Copy A into *C
7) Copy *C into A
8) Decrement C
9) Copy *C into B
10) Decrement C
11) A = A + B
12) Increment C
13) Copy A into *C



More of the same

t1 = 1
t2 = 3
t3 = t1+t2
t4 = 5
t5 = t3*t4



The final result

t1 = 1
t2 = 3
t3 = t1+t2
t4 = 5
t5 = t3*t4

1) Copy 1 into A
2) Increment C
3) Copy A into *C
4) Copy 3 into A
5) Increment C
6) Copy A into *C
7) Copy *C into A
8) Decrement C
9) Copy *C into B
10) Decrement C
11) A = A + B
12) Increment C
13) Copy A into *C
14) Copy 5 into A
15) Increment C
16) Copy A into *C
17) Copy *C into A
18) Decrement C
19) Copy *C into B
20) Decrement C
21) A = A * B
22) Increment C
23) Copy A into *C

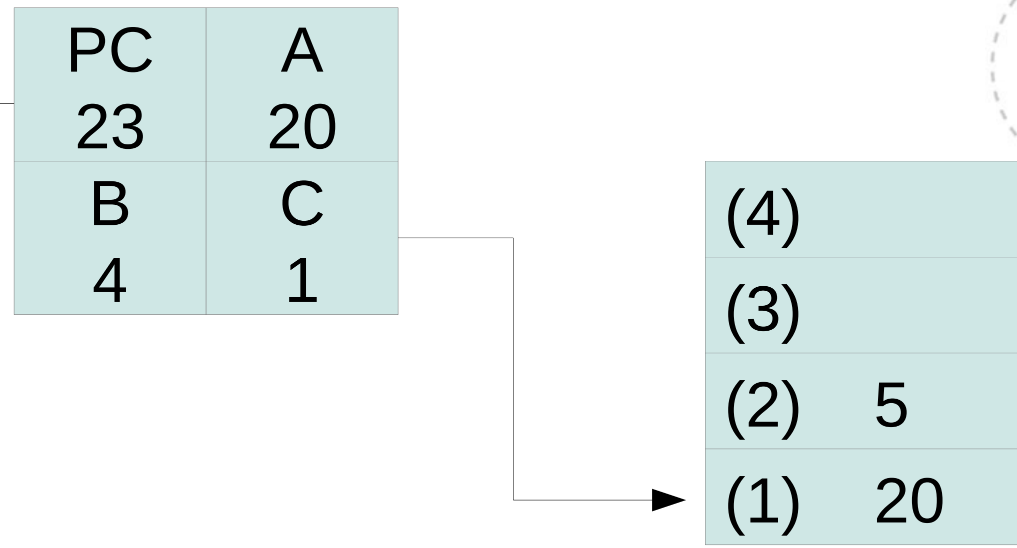
PC	A
20	5
B	C
4	0

(4)	
(3)	
(2)	5
(1)	4

The final result

t1 = 1
t2 = 3
t3 = t1+t2
t4 = 5
t5 = t3*t4

1) Copy 1 into A
2) Increment C
3) Copy A into *C
4) Copy 3 into A
5) Increment C
6) Copy A into *C
7) Copy *C into A
8) Decrement C
9) Copy *C into B
10) Decrement C
11) A = A + B
12) Increment C
13) Copy A into *C
14) Copy 5 into A
15) Increment C
16) Copy A into *C
17) Copy *C into A
18) Decrement C
19) Copy *C into B
20) Decrement C
21) A = A * B
22) Increment C
23) Copy A into *C



Many of those operations were repetitive

- Sequences like

    Set A to (value)

    Increment C

    Put value of a in memory at adr. C

  appear whenever (value) needs to be stored away
- Sequences like

    Set A to memory value at adr. C

    Decrement C

  appear when we need them again

Register C isn't special

- The pattern we used to lay out the operations here could just as well have used A or B to track memory locations, and the other two for operations
- The one we choose behaves like a pointer to the top of a stack, because we manipulate it that way

Stack operation support

- This is such a common thing to do that CPU designers embed support for it into the instruction set
- If we *make* register C special by designating it as the stack-pointer register, it can support instructions like

  push 5 (Move reg C “forward” & place 5 where it points)

  pop B (Put value from adr. in reg C into B & move C “backward”)

  and the program shortens to

```
push 1
push 3
pop A
pop B
A = A + B
push A
…
```

Stack machines

- Instruction support doesn't prevent the stack pointer register from containing whatever you like
  - All it tells us is that the value will change as a side effect of push and pop operations
- Popping values off stack doesn't delete them
  - They will just be overwritten when the stack pointer next comes by there
- The scheme is enough to handle arbitrarily complicated expressions
  - There can be as many temporary values on stack as needed, while we use registers for two at a time

It could be even simpler

- We could get away with
  - one “accumulator” register
  - an implicit stack pointer
  - operations that combine values from the top of the stack into the accumulator
- We could even drop explicit registers altogether, using
  - an implicit stack pointer
  - operations that combine the top two elements
- CPUs like this work, but they result in longer programs with more memory traffic
  - They’re kind of old-fashioned, yet simple to make

Unconditional jumps

- Jump instructions have a straightforward interpretation in our minimal CPU model
  - they are assignments to the PC register, like so:

1) Copy 1 into A
2) Increment C
3) Copy A into *C
4) **Jump 8**
5) Copy 3 into A
6) Increment C
7) Copy A into *C
8) Copy 2 into A

PC	A
4	1
B	C
0	1

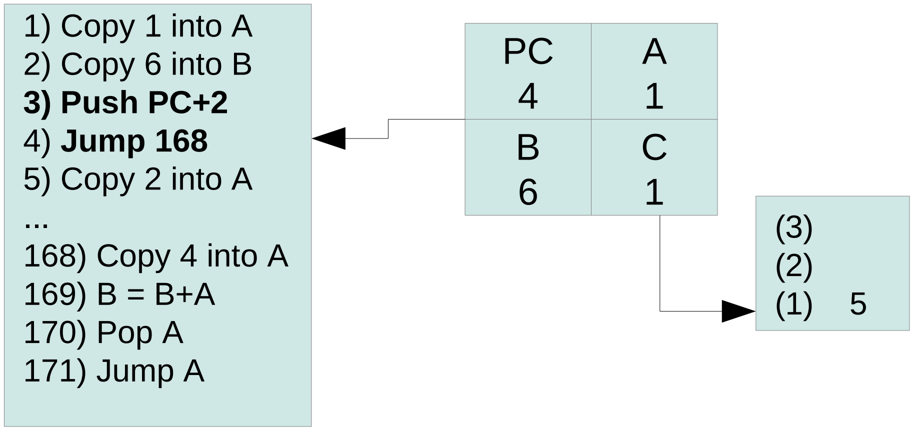
1) Copy 1 into A
2) Increment C
3) Copy A into *C
4) jump 8
5) Copy 3 into A
6) Increment C
7) Copy A into *C
8) **Copy 2 into A**

PC	A
8	2
B	C
0	1

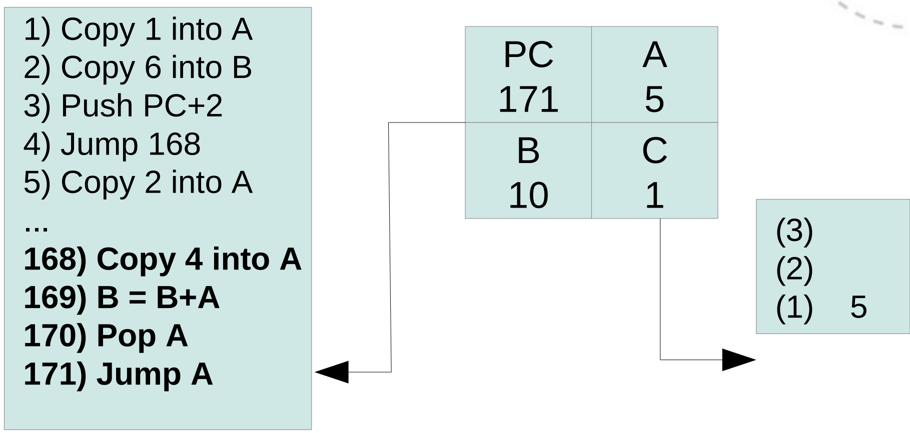
(Here, ops 5-7 will never be run)

Simple subroutines

- With memory indexing, we can store the value of PC
- This permits branching off to another part of the program, and coming back again



Jump away...



…do stuff, and return to where we were

Those can be operations too

- “Call” translates into
  - Push return address to remember
  - Jump to target
- “Return” translates into
  - Pop address to return to from stack
  - Jump there
- As with “push” and “pop”, call/return are just shorthands for sequences of operations we could also write out explicitly
  - Subroutines make code modular, sections of it can be re-used in several places
  - Subroutines don’t have local context, everything is just a global memory address
  - The GOSUB keyword in many (old) dialects of BASIC works this way

Function call and return

- Translating function calls into this low-level abstraction is a matter of using the stack for two purposes
  - Placing the return location in the program there
  - Placing the values local to the call there
- An *activation record* gives a policy on how to sort these things, so that they can be systematically manipulated and recovered at the appropriate time

IA-32 activation records

- The personal computers of yesteryear had a convention for how to structure stuff on the stack
- It's noticeably cleaner than its present successor, so it merits brief scrutiny
  - Contemporary 64-bit CPUs (Intel and relatives) will still run IA-32 code, they're backwards compatible
  - Contemporary compilers will still generate it, if you tell them to produce 32-bit x86 code (GCC does it with the flag -m32)
- We could have used it directly in the practical work, but it grows more contrived year
  - 16MHz 386/DX: performance monster of 1985
  - I believe in keeping up with progress, even when it's ugly

x86 in 60 seconds

- There's a stack pointer register called ESP
- There's a frame pointer register called EBP
- There are push and pop instructions that manipulate ESP as a side-effect
- There are 2-operand instructions which store the result in one of the operands (move, add, sub, …)
- There are another few registers
  - We can use EAX and EBX just like A and B from our mini-machine
- There are 'call' and 'ret' operations, as discussed

NTNU – Trondheim
Norwegian University of
Science and Technology

What's in a function's context?

- Let's take this one, in C:

```
int factorial ( int n ) {
  int result = n;
  if ( result > 1 )
      result *= factorial ( result – 1 );
  return result;
}
```

(This is an awful implementation, it's made more to illustrate stack frames than to compute factorials)

Key ingredients

Argument to receive

Local variable

```
int factorial ( int n ) {
  int result = n;
  if ( result > 1 )
     result *= factorial ( result – 1 );
  return result;
}
```

Argument to send

Return value

Next time, we will look at how x86 organized these parts, and connect it back to TAC representation