NTNU – Trondheim
Norwegian University of
Science and Technology

Function calls and the run-time stack

Beyond jump and return

- We’ve looked at how jumps to saved addresses create the control flow of procedure calls
- Functions also require a local environment to be arranged
- Abandoning our hypothetical mini-CPU, we can examine how x86-s do it

The basic x86 approach

- Arguments need to go on the stack
  – The calling function handles putting them there, and taking them away again
- Return address must go on the stack
  – The calling function handles it, because it knows where to resume execution
- Local variables need to go on the stack
  – The called function knows how much space they will need, and allocates it
- Stack is both local namespace and temporary results
  – Stack pointer deals with intermediate results
  – Frame pointer locates the start of the local namespace
- Return value must go somewhere
  – A designated register plays this part

Activation record of our factorial function

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

Next call’s local var. “result”
My frame ptr.
Return address
Argument: value of “result-1”
(Intermediate data)
Local var: “result”
Caller’s frame ptr.
Return address
Argument: “n”

Callee places these, when called

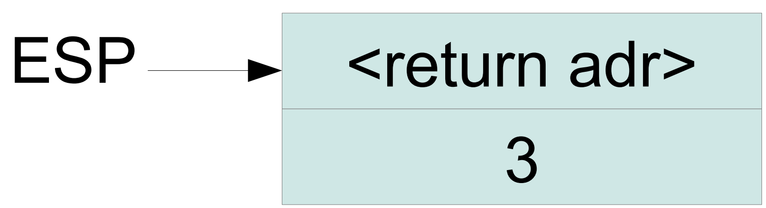
Generated function body places these

Caller places these, prior to call

Calling factorial(3)

push 3

call factorial



(EBP is somewhere below)

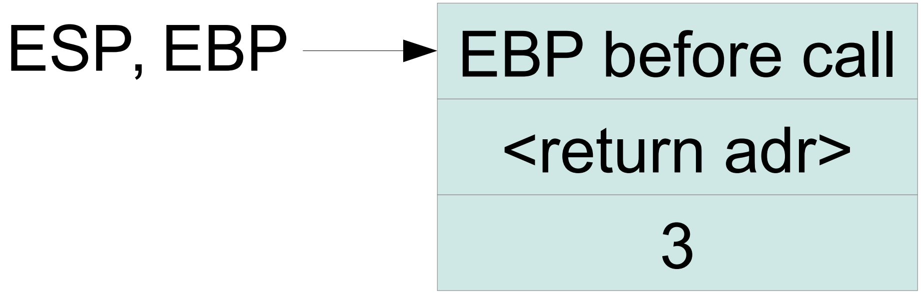
factorial(3) receives

push 3

call factorial

push EBP

move ESP into EBP



factorial() makes local space

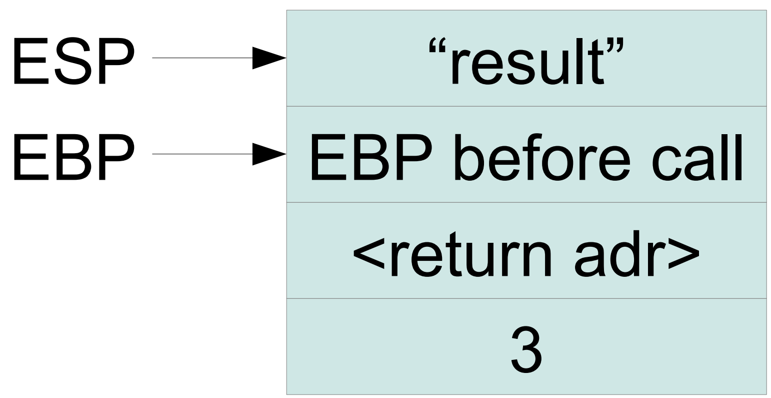
push 3

call factorial

push EBP

move ESP into EBP

sub 4, ESP



Assign argument n to “result”

push 3

call factorial

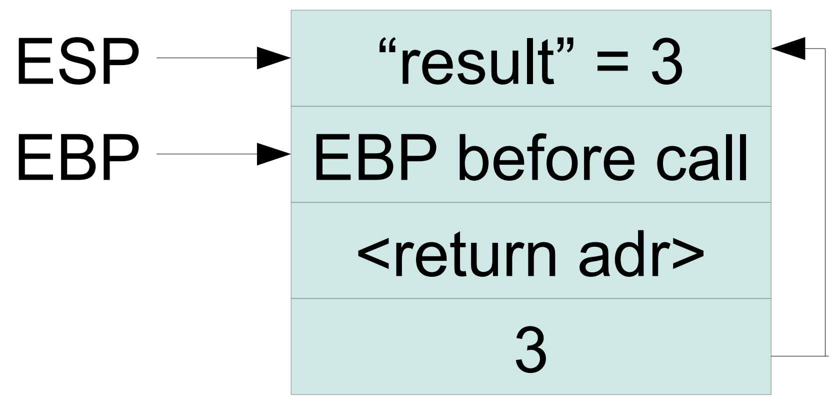
push EBP

move ESP into EBP

sub 4, ESP

move 12(EBP), EAX

move EAX, -4(EBP)



Calculate result-1 for next call, push it as argument

push 3

call factorial

push EBP

move ESP into EBP

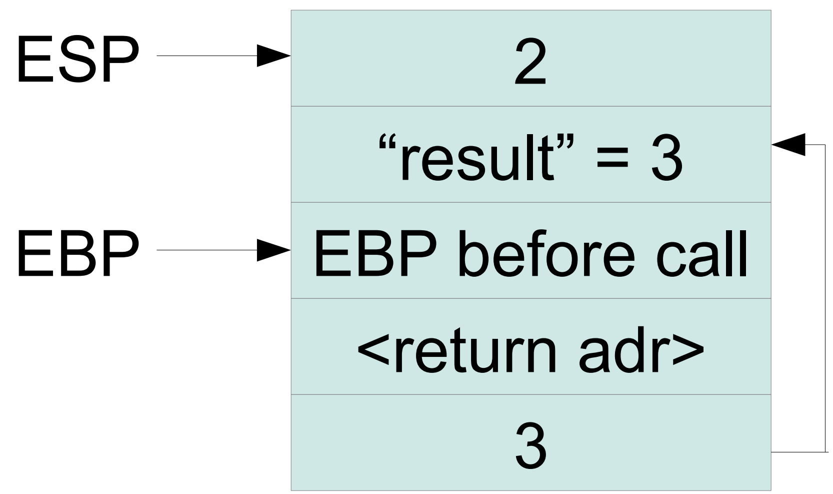
sub 4, ESP

move 8(EBP), EAX

move EAX, -4(EBP)

(...find out that 3-1 = 2…)

push 2



Make the next call, thus pushing return adr.

push 3

call factorial

push EBP

move ESP into EBP

sub 4, ESP

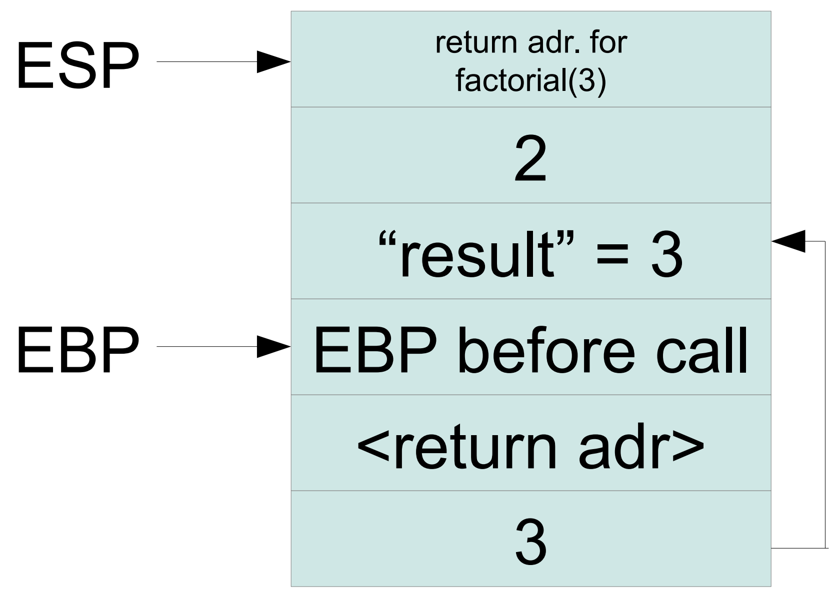
move 8(EBP), EAX

move EAX, -4(EBP)

(...find out that 3-1 = 2…)

push 2

call factorial



NTNU – Trondheim
Norwegian University of Science and Technology

...and the whole circus repeats...

push 2

call factorial

push EBP

move ESP into EBP

sub 4, ESP

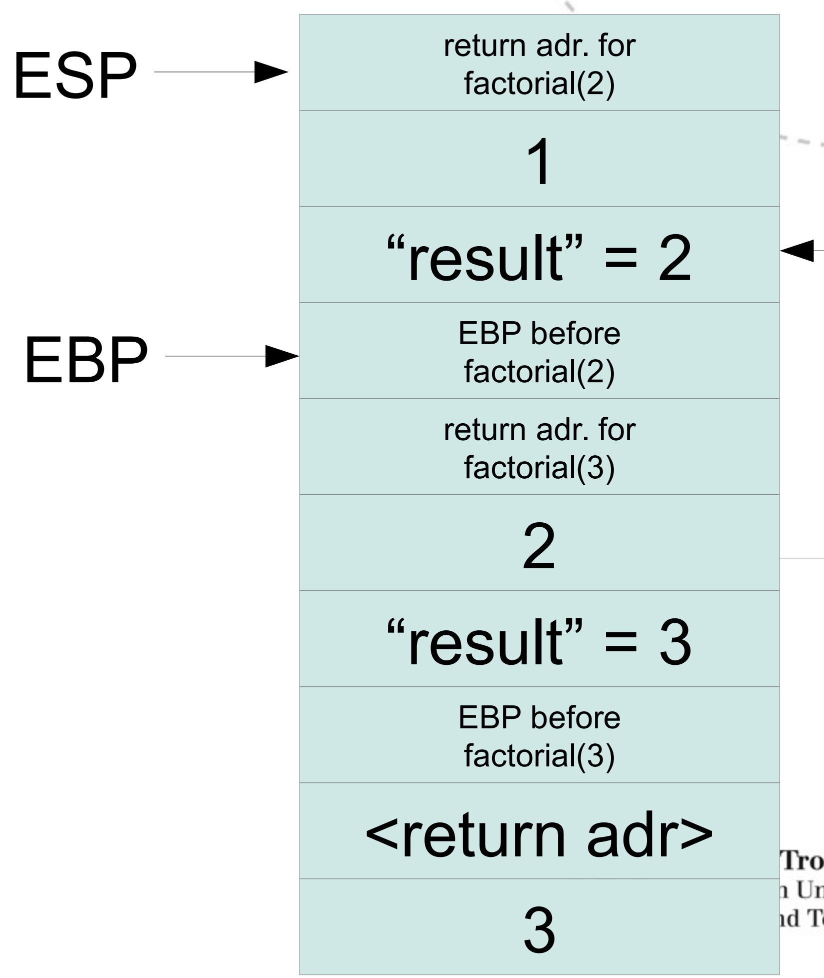
move 8(EBP), EAX

move EAX, -4(EBP)

(...find out that 2-1 = 1…)

push 1

call factorial



...until return.

Unwind factorial(1):

push 1

call factorial

push EBP

move ESP into EBP

sub 4, ESP

move 8(EBP), EAX

move EAX, -4(EBP)

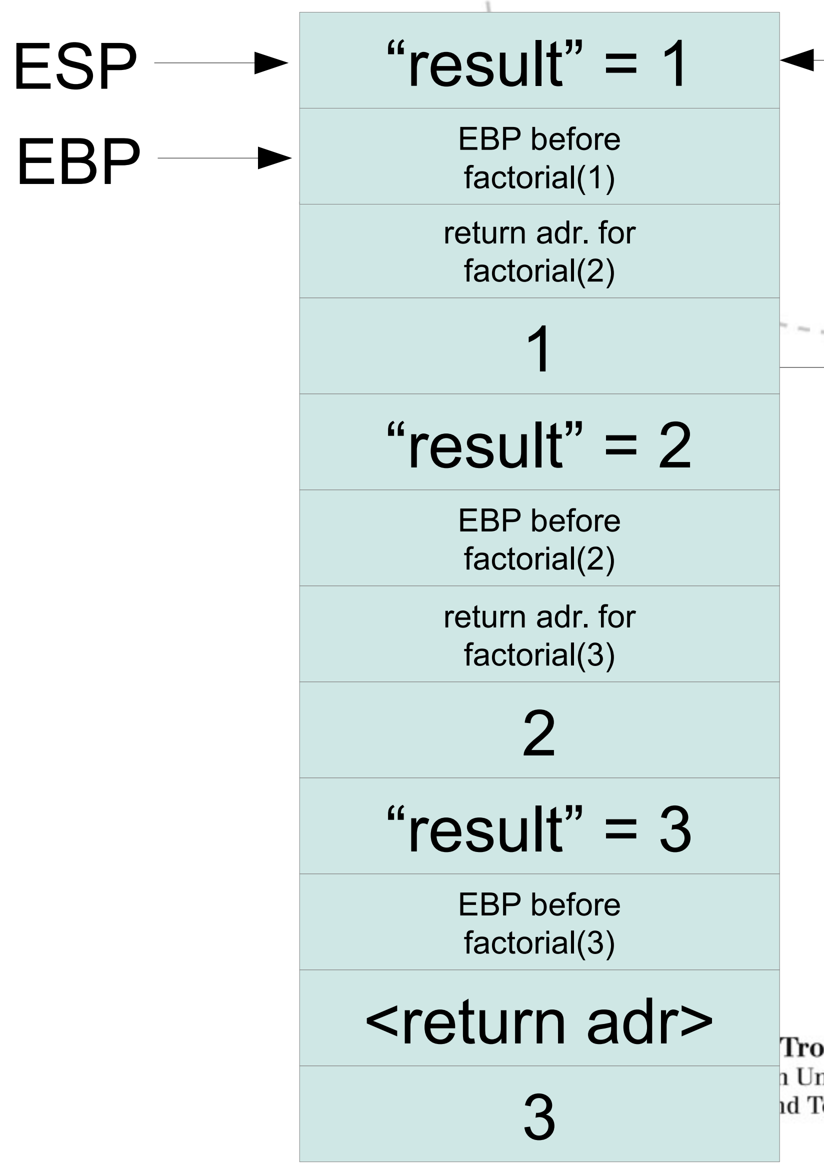
(...find out that 1 > 1 is false…)

move -4(EBP), EAX

move EBP, ESP

pop EBP

ret



Result: EAX=2

Unwinding factorial(2)

add 4, ESP

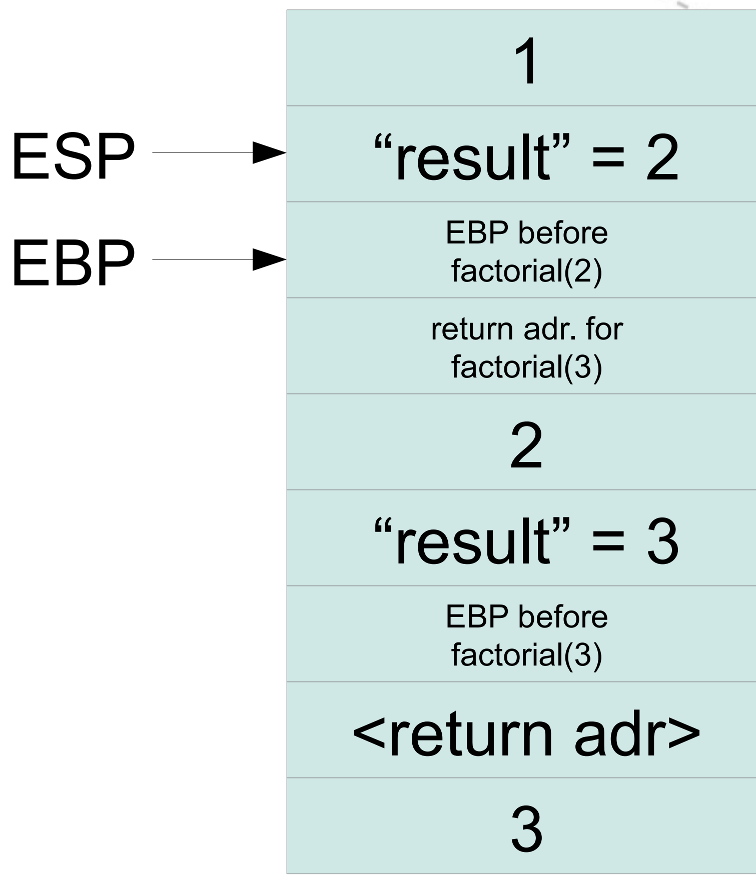
...multiply EAX into -4(EBP)…

move -4(EBP), EAX

move EBP, ESP

pop EBP

ret



Result: EAX=6

Unwinding factorial(3)

add 4, ESP

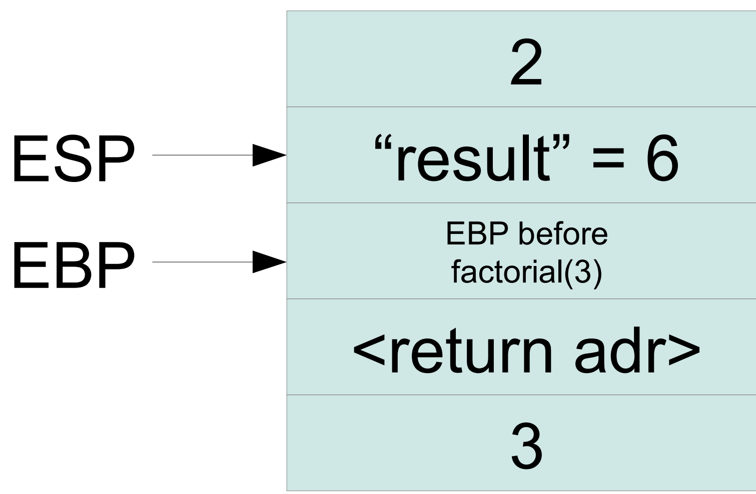
...multiply EAX into -4(EBP)…

move -4(EBP), EAX

move EBP, ESP

pop EBP

ret



Result: EAX=6

Returning to caller

```
add 4, ESP
```

...multiply EAX into -4(EBP)…

```
move -4(EBP), EAX
move EBP, ESP
pop EBP
ret
```

The answer is here

EBP off somewhere below

ESP

3

A handful of details

- All my addresses are in multiples of 4, on the assumption that “int” is 32 bits (4 bytes)
- x86 stack space grows from high to low addresses, because it starts from the end of the process image:

0	text	data	heap →		← stack	*2^64-1*

  - “push” subtracts from the stack pointer
  - “pop” adds to the stack pointer

NTNU – Trondheim
Norwegian University of
Science and Technology

A handful of white lies

- This was *almost* the sequence of operations you'll get out if you punch in "factorial.c" and run it through "cc -m32 -S factorial.c" to get the x86 assembly

  ...but not *quite*…
- The dimensioning of local space (movement of ESP at activation) isn't exactly flush with the number of local variables
- I skipped evaluation of conditionals and multiplication
  - We've covered them in TAC, and can do them up in assembly later
- Syntax deviates
  - You can't copy-paste what's written here and expect it to assemble

The focal point

- Function call in TAC looks like this

  ```
  param t1
  param t3
  param x
  call foo
  ```

  for a function foo(a,b,c)
- The ‘param’ notation has an immediate interpretation in IA-32 assembly, *i.e.* “push the parameter on stack”
- It has a slightly different one in x86_64 which we’ll look at later
- Together, they may clarify why a low-IR (abstract assembler) has use for the ‘param’ notation

NTNU – Trondheim
Norwegian University of Science and Technology

Secondary points

- We didn't talk a lot about indirect addressing, except for its use in arrays
  - i.e. expressions like t2 = 12(t1)
  - to mean "the value 12 addresses away from that in t1"
- The layout of an activation record makes an obvious use of it
  - Local variables are translated into stack positions, located by their offset from the frame pointer

Back to the overview

- Expressions translate into strings of operations, with temporaries for intermediate results
- Loops and conditionals translate into evaluation code for the condition, followed by fixed control flow patterns
- Function call and return translates into buffering up the arguments and jumping to the function
- Function bodies translate into a machine-related convention for where to find the arguments and where to put the local environment

The Keys to the Kingdom

- What hasn't been mentioned is that these translation patterns are not final definitions taken from the Great Standard of Program Constructions™
  - They are devices we invent to give source languages their meaning
  - If you implement another translation of switch statements, you redefine what every source program with a switch in will do
  - If you invent a new language construct, the translation pattern you assign to it will specify what it can be used for
- This is the biggest takeaway from compiler construction:

  The evaluation rules you learn for any language only appear because someone decided to implement them that way

The processor doesn't care, you can make different rules if you like.

Inefficiencies that appear

- Duplicate values

    t1 = x

    t2 = y

    t3 = t1 + t2

  might as well be

    t1 = x + y

  if the expression-translation recognizes the special case where its operands are terminals

Redundant temporaries

- Temporary vars. have limited lifespan:

  t1 = 1
  t2 = 2
  t3 = 1 + 2
  t4 = 6
  t5 = 7
  t6 = t4 + t5

  might as well re-use t1, t2

  t1 = 6
  t2 = 7
  t4 = t1 + t2

  when their work is done.
- Pro: less space
- Con: less *precise* analyses at optimization

We'll return to what this means

Jumps to unconditional jumps

If a then if b then c=d else e=f else g=h

```
becomes
ifFalse a goto L1
    ifFalse b goto L2
        c=d
    jump Lend2
    L2:
        e=f
    Lend2:
jump Lend1
L1:
    g = h
Lend1:
```

NTNU – Trondheim
Norwegian University of
Science and Technology

This may as well shortcut

If a then if b then c=d else e=f else g=h

```
ifFalse a goto L1
    ifFalse b goto L2
        c=d
    jump Lend1
    L2:
        e=f

jump Lend1
L1:
    g = h
Lend1:
```