Becoming Independent

Debuggers introduction

Assembly language programming
By xorpd

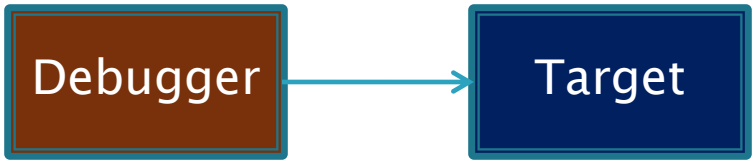
xorpd.net

Objectives

- We will learn about:
  - What is a debugger?
  - Why use a debugger?
  - Common mechanisms used by debuggers.

Debugger

- A program that is used to test and examine other programs, dynamically.



- Two programs:
  - Debugger
  - Debuggee / “target” program
- The debugger controls the target program.
  - Pause and continue execution.
  - Read or change internal state. (Memory or registers)
- The debugger usually gets help from:
  - The Operation system
  - The Processor

Debugger (Cont.)

- Low level debuggers are different from high level languages debuggers.
- High level debuggers inspect higher level constructs.
  - Language dependent. (Python, Ruby, Lisp etc.)
- Low level debuggers deal with assembly instructions and raw memory.
- We are going to talk about low level assembly debuggers.

Why use a debugger?

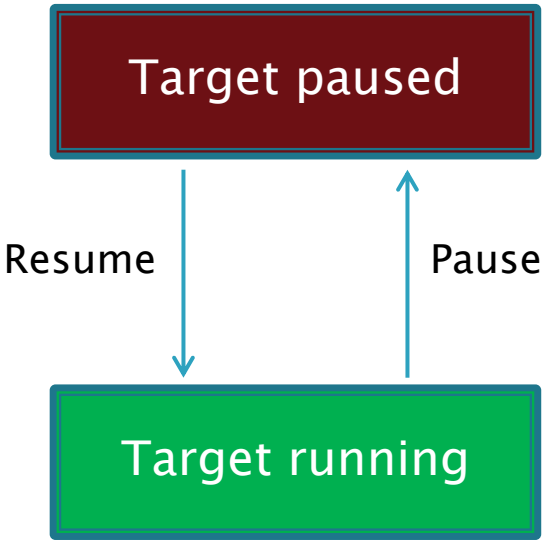
- Understand how a program works.
  - Dynamic analysis gives much information.

- Find and understand bugs in your code.

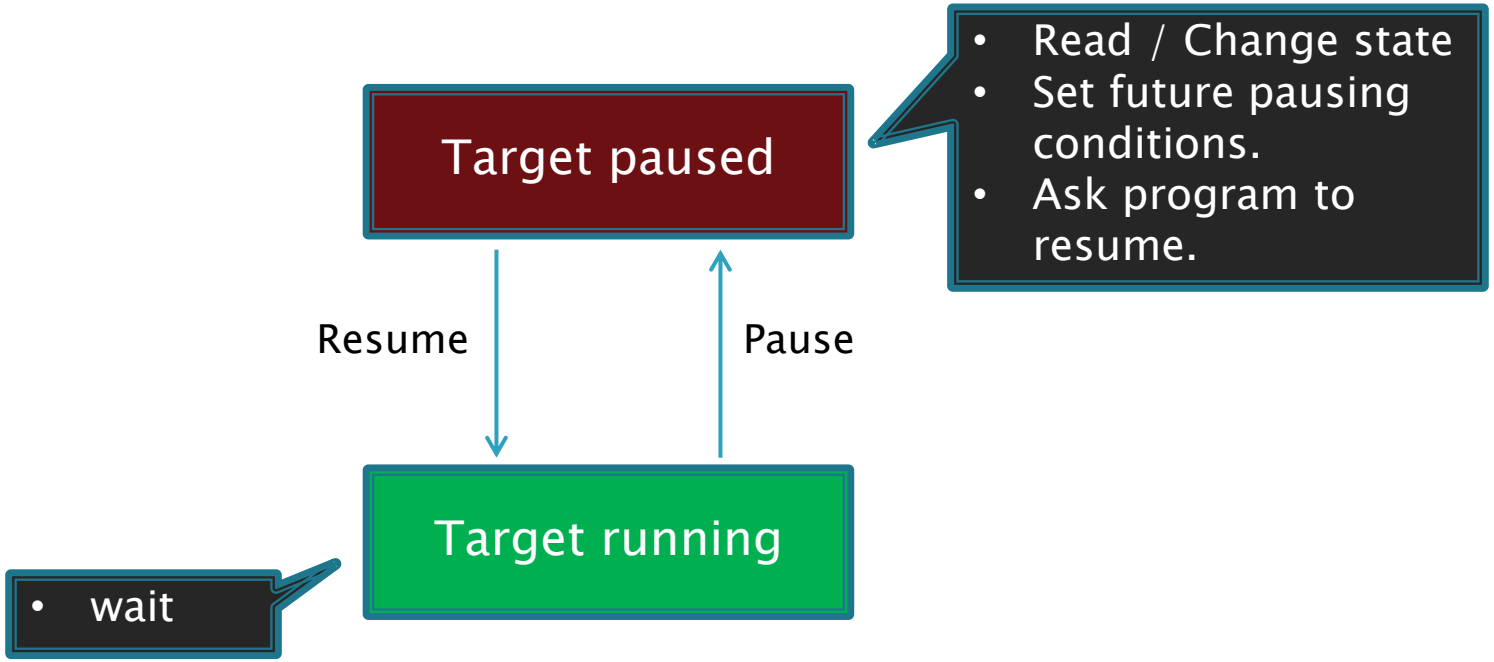
Generic operation

- The target is always in one of two states:
  - Paused or running.
- The target is usually launched in a paused state by the debugger.
- During a paused state, the debugger can:
  - Read the internal state of the target.
  - Change the internal state of the target.
  - Set future conditions for pausing the target program.
  - Ask the target program to resume.
- While the target is running, the debugger waits.

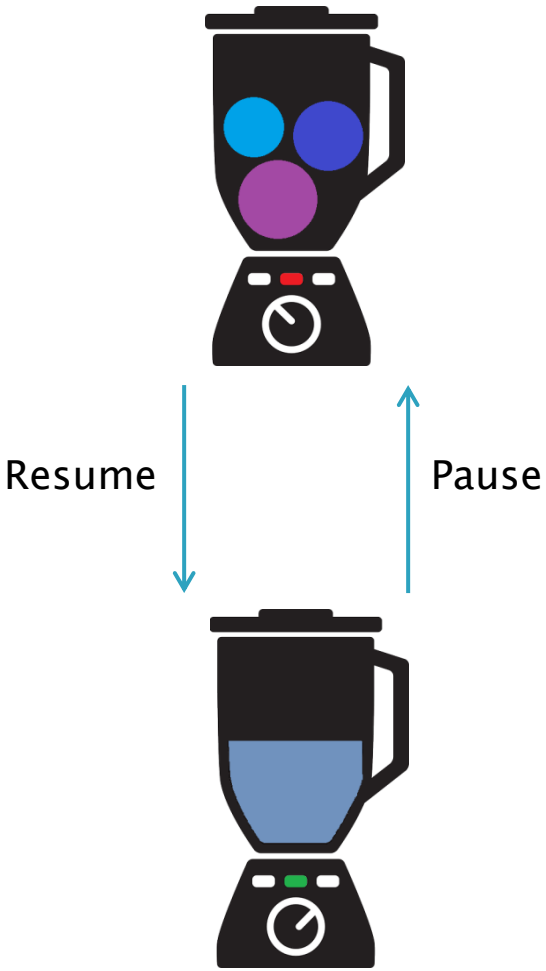
Generic operation (Cont.)



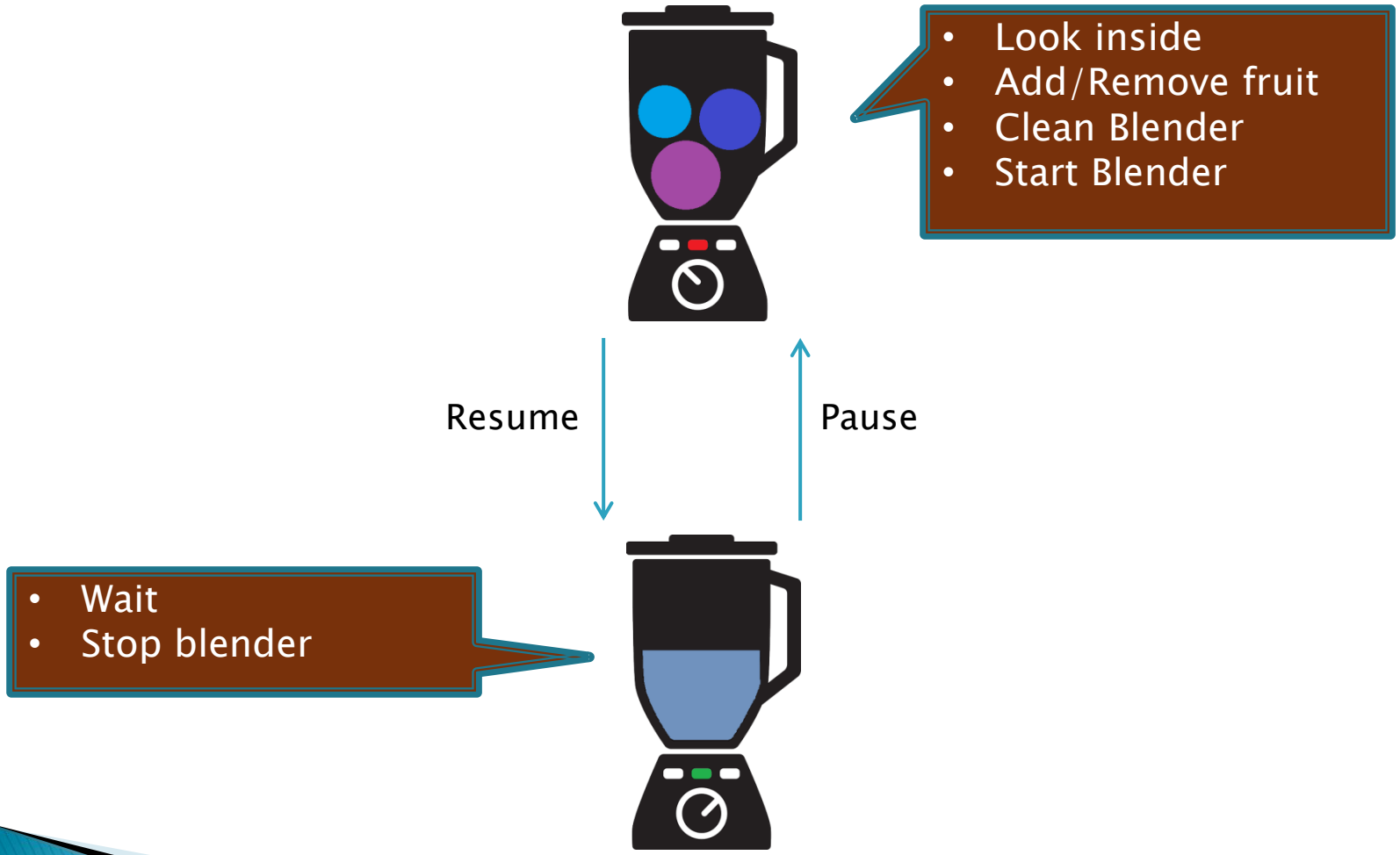
Generic operation (Cont.)



Blender idea



Blender idea



Target's pause time

- Most of the interesting things could be done while the target is paused:
  - Read / Change registers.
    - Change EIP
  - Read / Change memory.
  - Set future conditions for pausing.

Stepping

- Step: Resume for one instruction and then pause.
  - Execute instructions one by one.

- Most modern debuggers support “stepping”.

- Most debuggers will distinguish between two types of “steps”:
  - Step **into**: Step into functions.
  - Step **over**: Step over functions.

Stepping (Cont.)

- Stepping “into”

Target program

```
    mov     eax,2
    mov     ecx,3
    call    simple_func
    add     eax,ecx
    ...
```

```
simple_func:
    add     eax,ecx
    ret
```

Stepping (Cont.)

- Stepping “into”

Target program

```
→ mov     eax,2
  mov     ecx,3
  call    simple_func
  add     eax,ecx
  ...
```

```
simple_func:
    add     eax,ecx
    ret
```

Stepping (Cont.)

- Stepping “into”

Target program

```
    mov     eax,2
→   mov     ecx,3
    call    simple_func
    add     eax,ecx
    ...
```

```
simple_func:
    add     eax,ecx
    ret
```

Stepping (Cont.)

- Stepping “into”

Target program

```
    mov     eax,2
    mov     ecx,3
→   call    simple_func
    add     eax,ecx
    ...
```

```
simple_func:
    add     eax,ecx
    ret
```

Stepping (Cont.)

- Stepping “into”

Target program

```
    mov     eax,2
    mov     ecx,3
    call    simple_func
    add     eax,ecx
    ...
```

```
simple_func:
→   add     eax,ecx
    ret
```

Stepping (Cont.)

- Stepping “into”

Target program

```
    mov     eax,2
    mov     ecx,3
    call    simple_func
    add     eax,ecx
    ...
```

```
simple_func:
    add     eax,ecx
→   ret
```

Stepping (Cont.)

- Stepping “into”

Target program

```
    mov     eax,2
    mov     ecx,3
    call    simple_func
→   add     eax,ecx
    ...
```

```
simple_func:
    add     eax,ecx
    ret
```

Stepping (Cont.)

- Stepping “over”

Target program

```
    mov     eax,2
    mov     ecx,3
    call    simple_func
    add     eax,ecx
    ...
```

```
simple_func:
    add     eax,ecx
    ret
```

Stepping (Cont.)

- Stepping “over”

Target program

```
→ mov     eax,2
  mov     ecx,3
  call    simple_func
  add     eax,ecx
  ...
```

```
simple_func:
    add     eax,ecx
    ret
```

Stepping (Cont.)

- Stepping “over”

Target program

```
    mov     eax,2
→   mov     ecx,3
    call    simple_func
    add     eax,ecx
    ...
```

```
simple_func:
    add     eax,ecx
    ret
```

Stepping (Cont.)

- Stepping “over”

Target program

```
    mov     eax,2
    mov     ecx,3
→   call    simple_func
    add     eax,ecx
    ...
```

```
simple_func:
    add     eax,ecx
    ret
```

Stepping (Cont.)

- Stepping “over”

Target program

```
    mov     eax,2
    mov     ecx,3
    call    simple_func
→   add     eax,ecx
    ...
```

```
simple_func:
    add     eax,ecx
    ret
```

Software breakpoints

- “Pause whenever you get to this instruction”.
- Breakpoints are set during the target’s pause time.
- We set up breakpoints in a few interesting places, and let the target program run.
- The target program is paused whenever one of the breakpoints is reached.

Software breakpoints (Cont.)

- How does it work?

- INT 3
  - Trap to debugger.
  - Encoded as 0xcc

  - The debugger wakes up when this instruction is invoked.

Software breakpoints (Cont.)

- Example:

Target program

01 c0	add	eax,eax
05 05 00 00 00	add	eax,5h
29 d0	sub	eax,edx
40	inc	eax

Software breakpoints (Cont.)

- Example:

Target program

01 c0	add	eax,eax
05 05 00 00 00	add	eax,5h
29 d0	sub	eax,edx
40	inc	eax

Software breakpoints (Cont.)

- Example:

Target program

01 c0	add eax,eax
05 05 00 00 00	add eax,5h
cc	**int 3**
d0	; Leftovers
40	inc eax

Original opcode: 29 d0

Software breakpoints (Cont.)

- Example:

Target program

01 c0	add eax,eax
05 05 00 00 00	add eax,5h
cc	**int 3**
d0	; Leftovers
40	inc eax

Software breakpoints (Cont.)

- Example:

Target program

	01 c0	add eax,eax
→	05 05 00 00 00	add eax,5h
	cc	**int 3**
	d0	; Leftovers
	40	inc eax

Software breakpoints (Cont.)

- Example:

Target program

	01 c0	add eax,eax
	05 05 00 00 00	add eax,5h
→	**cc**	**int 3**
	d0	; Leftovers
	40	inc eax

Software breakpoints (Cont.)

- Example:

Target program

01 c0	add eax,eax
05 05 00 00 00	add eax,5h
→ **cc**	**int 3**
d0	; Leftovers
40	inc eax

- Target program is paused.
- Debugger is waken up.

Software breakpoints (Cont.)

- Example:

Target program

01 c0	add	eax,eax
05 05 00 00 00	add	eax,5h
→ 29 d0	sub	eax,edx
40	inc	eax

- Original instruction is reconstructed.
- Execution continues as usual.

Software breakpoints (Cont.)

- Example:

Target program

01 c0	add	eax,eax
05 05 00 00 00	add	eax,5h
29 d0	sub	eax,edx
→ 40	inc	eax

- Original instruction is reconstructed.
- Execution continues as usual.

Software breakpoints (Cont.)

- Modern debuggers do not expose you to the **int 3** replacement process.

- The debugger does the replacement and reconstruction for you.

Summary

- Debugger is a tool to help you understand code and solve problems in your programs.
- The Debugger controls the target program.
- The target program is always in one of two states:
  - Paused or Running.
- Stepping allows us to run the target program instruction by instruction.
- Software breakpoints wake up the debugger when a specific instruction is reached.