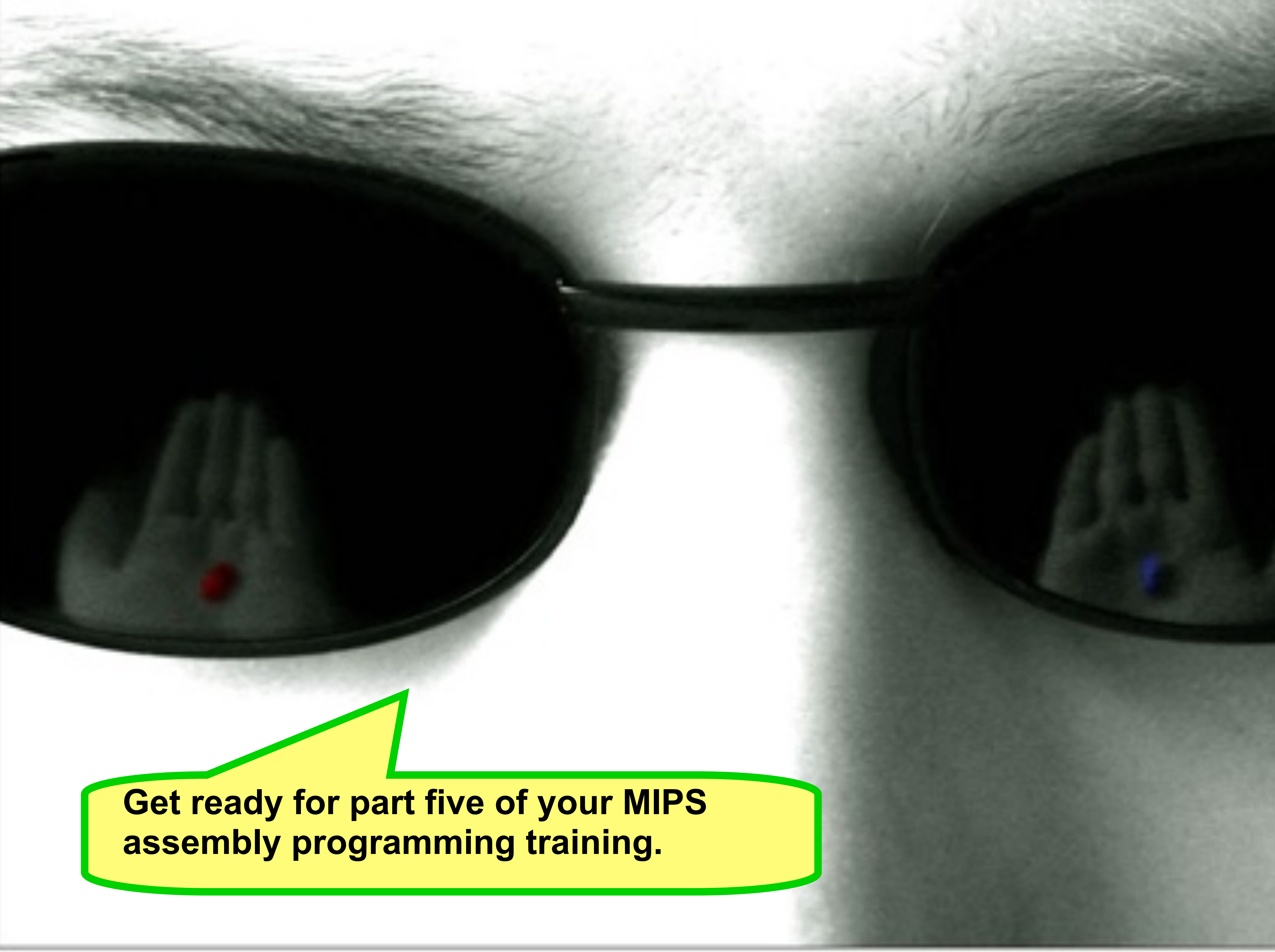
MIPS and SPIM tutorial

Part Five

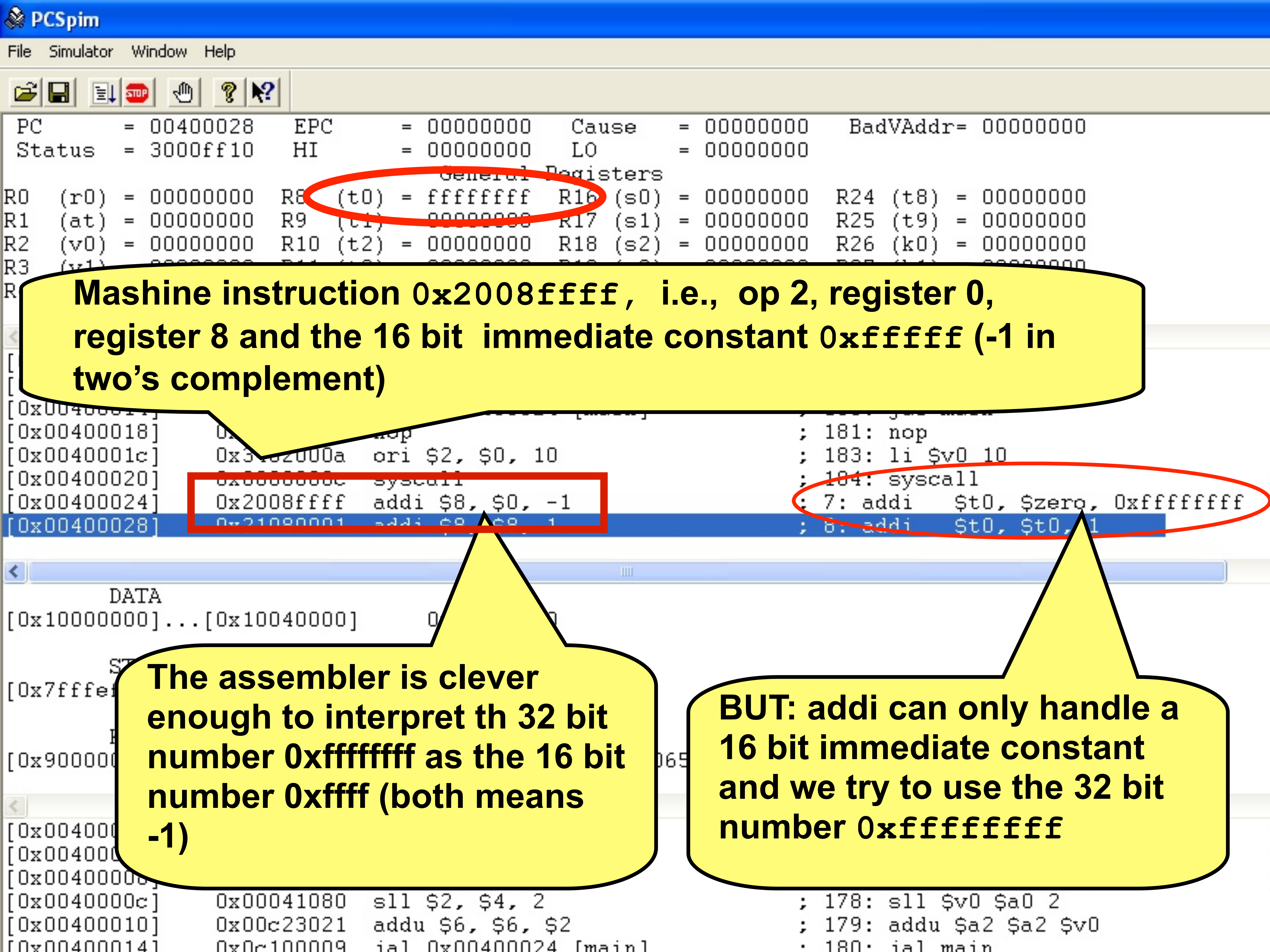
Exception and Interrups ► Polled and Interrupt driven I/O ► DMA ► Introduction to Operating Systems

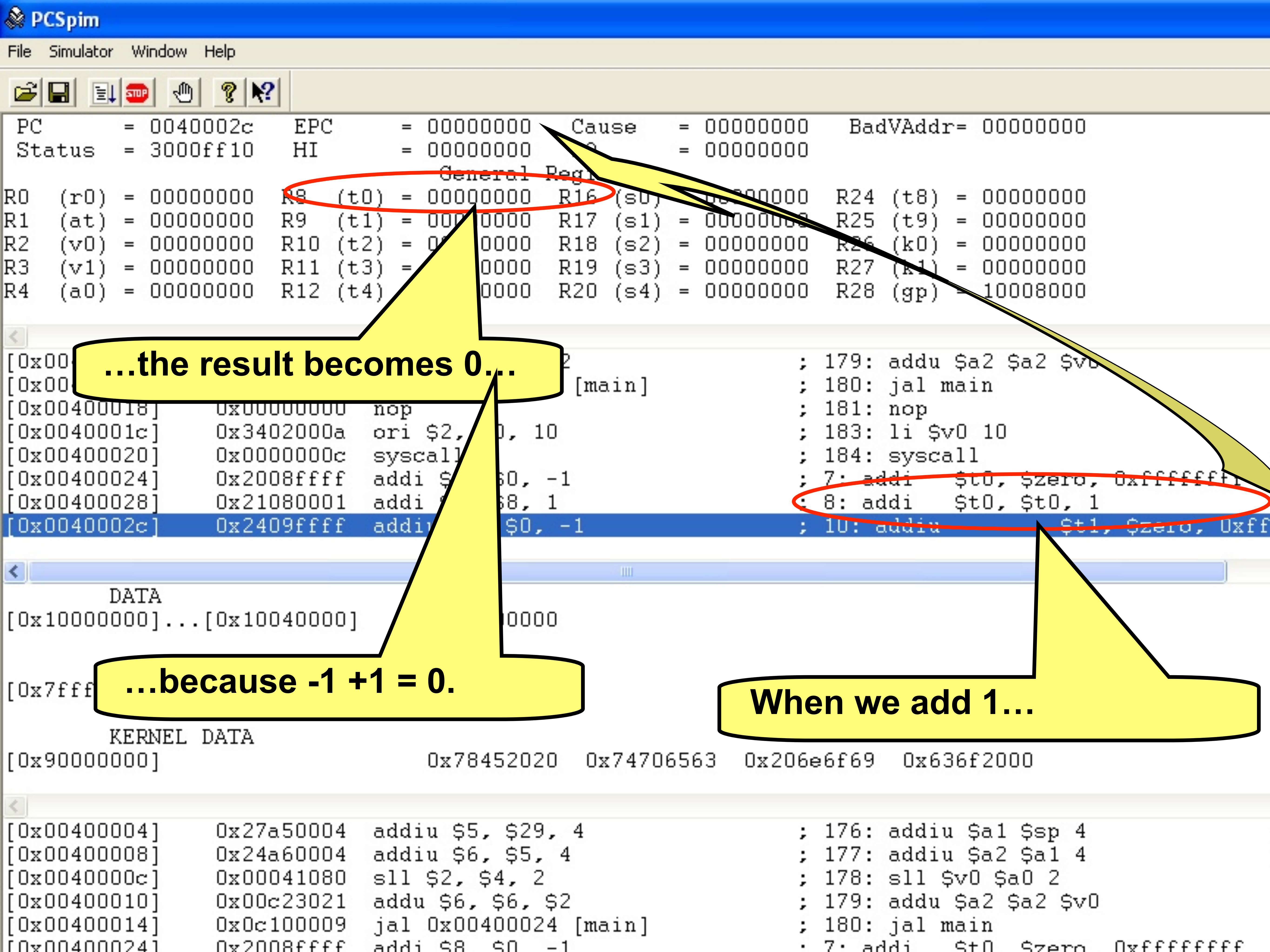
November 2008

karl.marklund@it.uu.se

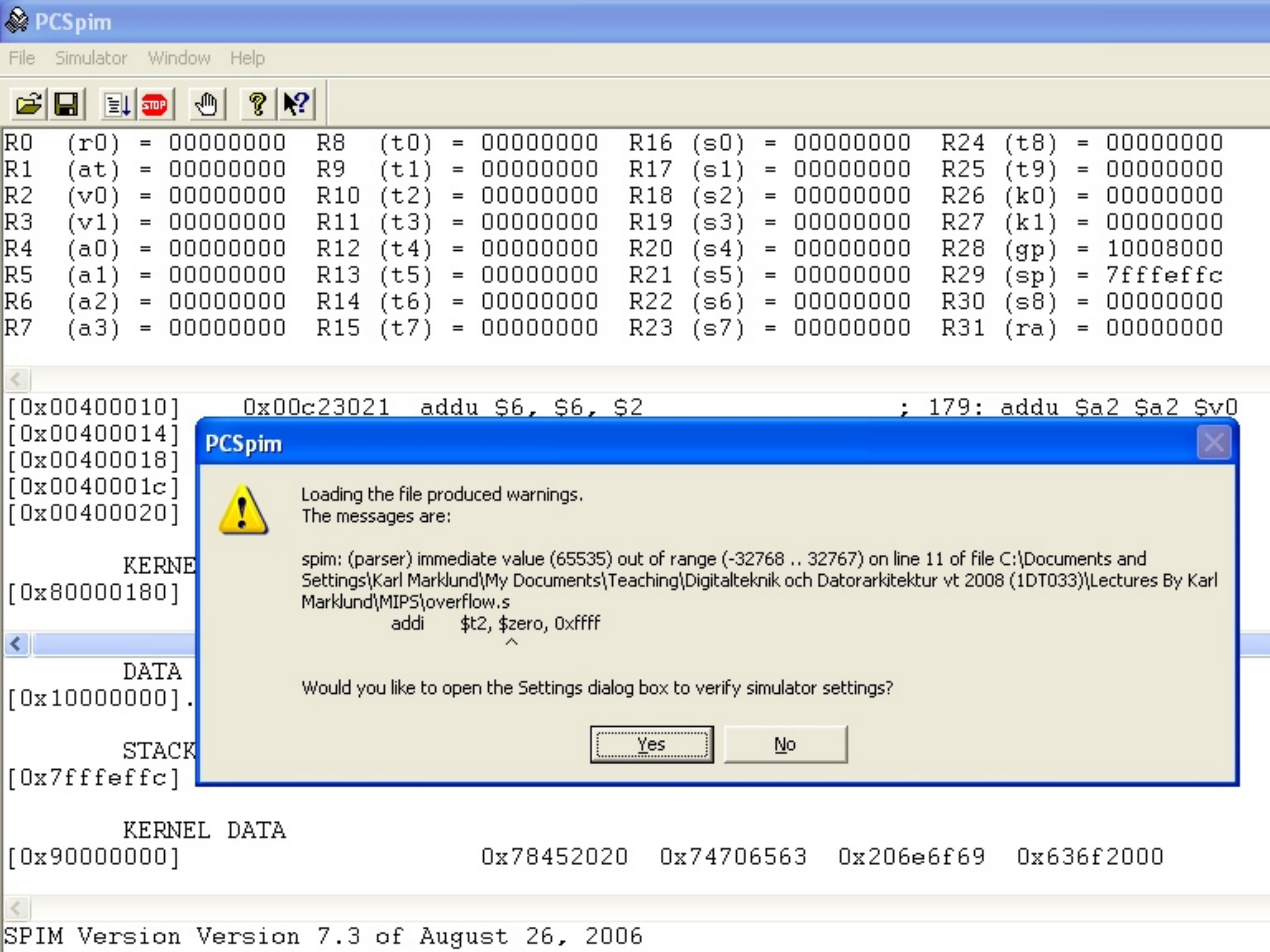

Get ready for part five of your MIPS assembly programming training.

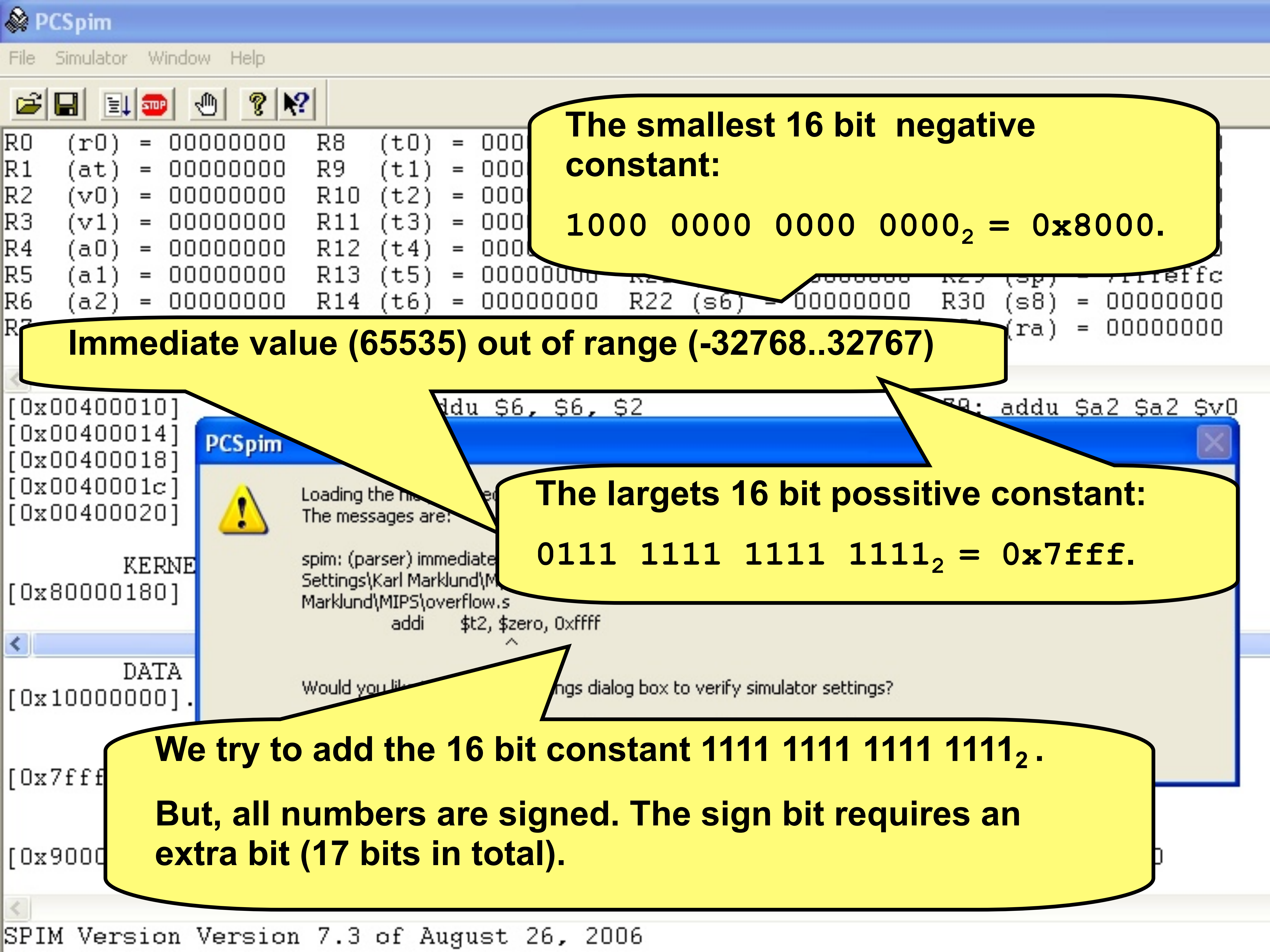

Time to create a really large number... Lets see how SPIM copes whith this.
addi $t0, $zero, 0xffffffff
addi $t0, $t0, 1

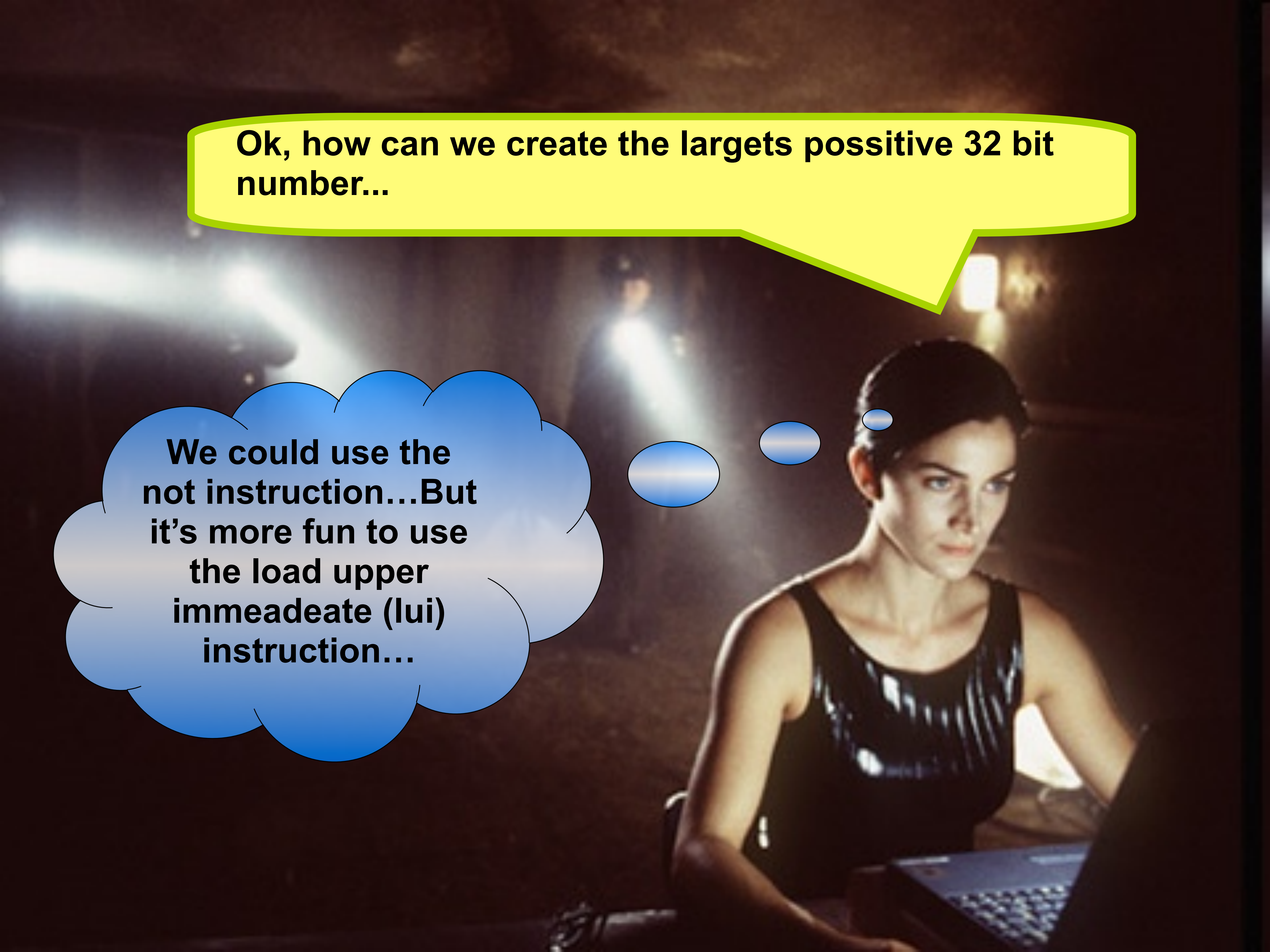

PCSpim
File Simulator Window Help
PC = 00400028 EPC = 00000000 Cause = 00000000 BadVAddr= 00000000
Status = 3000ff10 HI = 00000000 LO = 00000000
R0 (r0) = 00000000 R8 (t0) = ffffffff R16 (s0) = 00000000 R24 (t8) = 00000000
R1 (at) = 00000000 R9 R17 (s1) = 00000000 R25 (t9) = 00000000
R2 (v0) = 00000000 R10 (t2) = 00000000 R18 (s2) = 00000000 R26 (k0) = 00000000
Mashine instruction 0x2008ffff, i.e., op 2, register 0, register 8 and the 16 bit immediate constant 0xffff (-1 in two's complement)
[0x00400018] ; 181: nop
[0x0040001c] ori $2, $0, 10 ; 183: li $v0 10
[0x00400020] ; 184: syscall
[0x00400024] 0x2008ffff addi $8, $0, -1 ; 7: addi $t0, $zero, 0xffffffff
[0x00400028] ; 8: addi $t0, $t0, 1
DATA
[0x10000000]...[0x10040000]
The assembler is clever enough to interpret th 32 bit number 0xffffffff as the 16 bit number 0xffff (both means -1)
BUT: addi can only handle a 16 bit immediate constant and we try to use the 32 bit number 0xffffffff
[0x0040000c] 0x00041080 sll $2, $4, 2 ; 178: sll $v0 $a0 2
[0x00400010] 0x00c23021 addu $6, $6, $2 ; 179: addu $a2 $a2 $v0


PCSpim
File Simulator Window Help
PC = 0040002c EPC = 00000000 Cause = 00000000 BadVAddr= 00000000
Status = 3000ff10 HI = 00000000 = 00000000
R0 (r0) = 00000000
R1 (at) = 00000000 R9 (t1)
R2 (v0) = 00000000 R10 (t2)
R3 (v1) = 00000000 R11 (t3)
R4 (a0) = 00000000 R12 (t4)
R17 (s1) R24 (t8) = 00000000
R18 (s2) = 00000000 R25 (t9) = 00000000
R19 (s3) = 00000000 R27 = 00000000
R20 (s4) = 00000000 R28 (gp) = 10008000
[0x0040001c] 0x3402000a ori $2, ; 183: li $v0 10
[0x00400020] 0x0000000c syscall ; 184: syscall
[0x00400024] 0x2008ffff addi ; 7: addi $t0, $zero, 0xffffffff
[0x00400028] 0x21080001 addi ; 8: addi $t0, $t0, 1
[0x0040002c] 0x2409ffff addiu $0, -1 ; 10: addiu $t1, $zero, 0xff
DATA
[0x10000000]...[0x10040000]
KERNEL DATA
[0x90000000] 0x78452020 0x74706563 0x206e6f69 0x636f2000
[0x00400004] 0x27a50004 addiu $5, $29, 4 ; 176: addiu $a1 $sp 4
[0x00400008] 0x24a60004 addiu $6, $5, 4 ; 177: addiu $a2 $a1 4
[0x0040000c] 0x00041080 sll $2, $4, 2 ; 178: sll $v0 $a0 2
[0x00400010] 0x00c23021 addu $6, $6, $2 ; 179: addu $a2 $a2 $v0
[0x00400014] 0x0c100009 jal 0x00400024 [main] ; 180: jal main
When we add 1…
…the result becomes 0…
…because -1 +1 = 0.


Hmm... Lets make another experiment.
addi $t2, $zero, 0xffff


PCSpim
File Simulator Window Help
R0 (r0) = 00000000 R8 (t0) = 00000000 R16 (s0) = 00000000 R24 (t8) = 00000000
R1 (at) = 00000000 R9 (t1) = 00000000 R17 (s1) = 00000000 R25 (t9) = 00000000
R2 (v0) = 00000000 R10 (t2) = 00000000 R18 (s2) = 00000000 R26 (k0) = 00000000
R3 (v1) = 00000000 R11 (t3) = 00000000 R19 (s3) = 00000000 R27 (k1) = 00000000
R4 (a0) = 00000000 R12 (t4) = 00000000 R20 (s4) = 00000000 R28 (gp) = 10008000
R5 (a1) = 00000000 R13 (t5) = 00000000 R21 (s5) = 00000000 R29 (sp) = 7fffeffc
R6 (a2) = 00000000 R14 (t6) = 00000000 R22 (s6) = 00000000 R30 (s8) = 00000000
R7 (a3) = 00000000 R15 (t7) = 00000000 R23 (s7) = 00000000 R31 (ra) = 00000000
[0x00400010] 0x00c23021 addu $6, $6, $2 ; 179: addu $a2 $a2 $v0
[0x00400014]
[0x00400018]
[0x0040001c]
[0x00400020]
KERNE
[0x80000180]
DATA
[0x10000000].
STACK
[0x7fffeffc]
KERNEL DATA
[0x90000000] 0x78452020 0x74706563 0x206e6f69 0x636f2000
SPIM Version Version 7.3 of August 26, 2006
PCSpim
Loading the file produced warnings.
The messages are:
spim: (parser) immediate value (65535) out of range (-32768 .. 32767) on line 11 of file C:\Documents and Settings\Karl Marklund\My Documents\Teaching\Digitalteknik och Datorarkitektur vt 2008 (1DT033)\Lectures By Karl Marklund\MIPS\overflow.s
addi $t2, $zero, 0xffff
^
Would you like to open the Settings dialog box to verify simulator settings?
Yes
No


PCSpim
File Simulator Window Help
R0 (r0) = 00000000 R8 (t0) = 000
R1 (at) = 00000000 R9 (t1) = 000
R2 (v0) = 00000000 R10 (t2) = 000
R3 (v1) = 00000000 R11 (t3) = 000
R4 (a0) = 00000000 R12 (t4) = 000
R5 (a1) = 00000000 R13 (t5) = 00000000
R6 (a2) = 00000000 R14 (t6) = 00000000 R22 (s6) = 00000000 R30 (s8) = 00000000
(ra) = 00000000
The smallest 16 bit negative constant:
1000 0000 0000 0000₂ = 0x8000.
Immediate value (65535) out of range (-32768..32767)
[0x00400010]
[0x00400014]
[0x00400018]
[0x0040001c]
[0x00400020]
addu $6, $6, $2
addu $a2 $a2 $v0
PCSpim
Loading the
The messages are:
spim: (parser) immediate
Settings\Karl Marklund\M
Marklund\MIPS\overflow.s
addi $t2, $zero, 0xffff
Would you li
ngs dialog box to verify simulator settings?
The largets 16 bit possitive constant:
0111 1111 1111 1111₂ = 0x7fff.
KERNE
[0x80000180]
DATA
[0x10000000]
[0x7fff
[0x9000
We try to add the 16 bit constant 1111 1111 1111 1111₂ .
But, all numbers are signed. The sign bit requires an extra bit (17 bits in total).
SPIM Version Version 7.3 of August 26, 2006


Ok, how can we create the largets possitive 32 bit number...
We could use the not instruction…But it’s more fun to use the load upper immeadeate (lui) instruction…

Load Upper Immediate: *lui*

```
lui    $t2, 0x7fff

addi   $t3, $t2, 0x7fff
```

"upper" half

$t2 ← 0x7fff0000

$t3 ← 0x7fff7fff

"lower" half

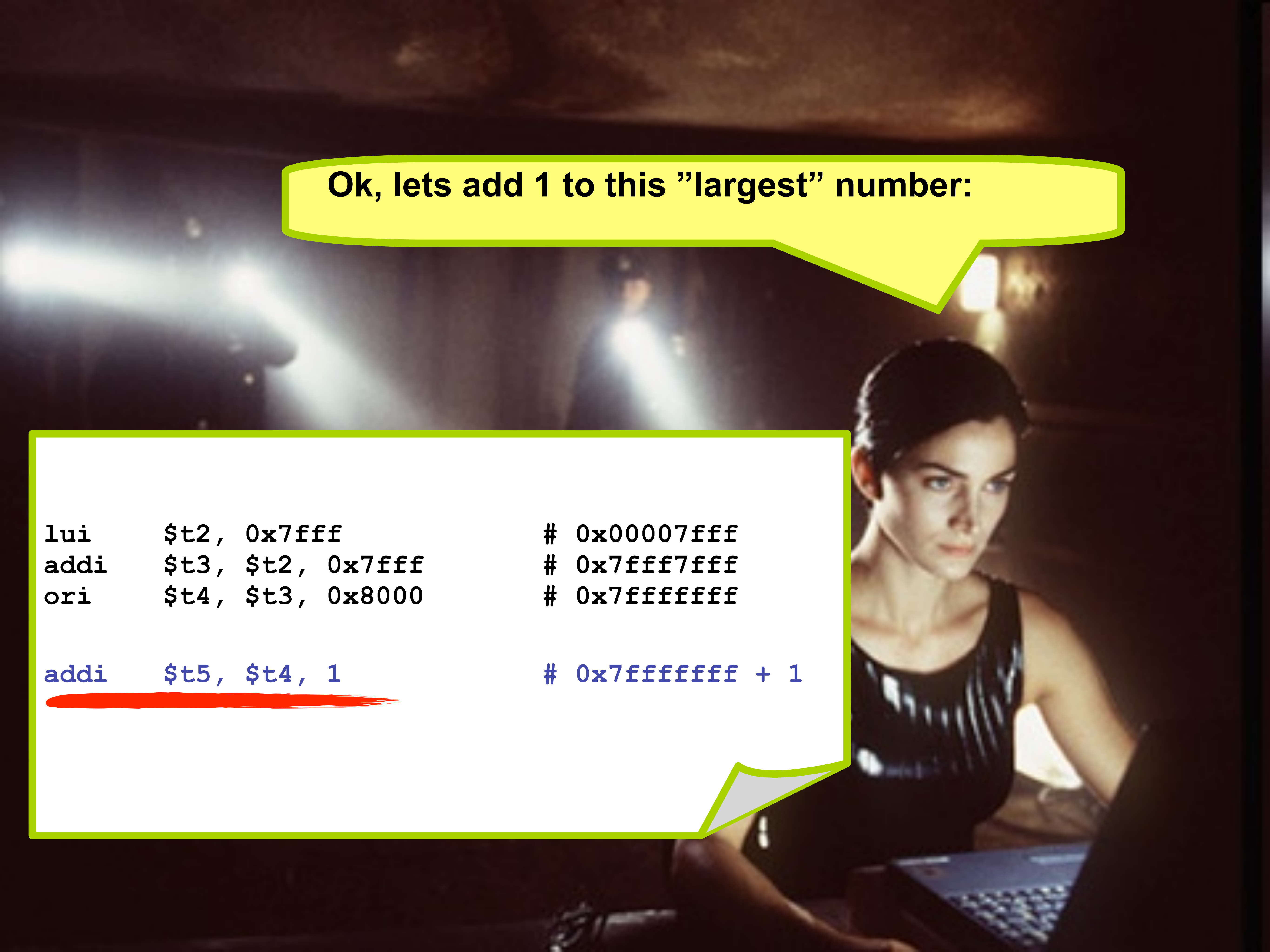
```
   0111 1111 1111 1111 0111 1111 1111 1111₂ (0x7fff7fff)

OR 0000 0000 0000 0000 1000 0000 0000 0000₂ (0x00008000)
-----------------------------------------------
   0111 1111 1111 1111 1111 1111 1111 1111  (0x7fffffff)
```

The largets possitive 32 bit number

```
ori   $t4, $t3, 0x8000
```

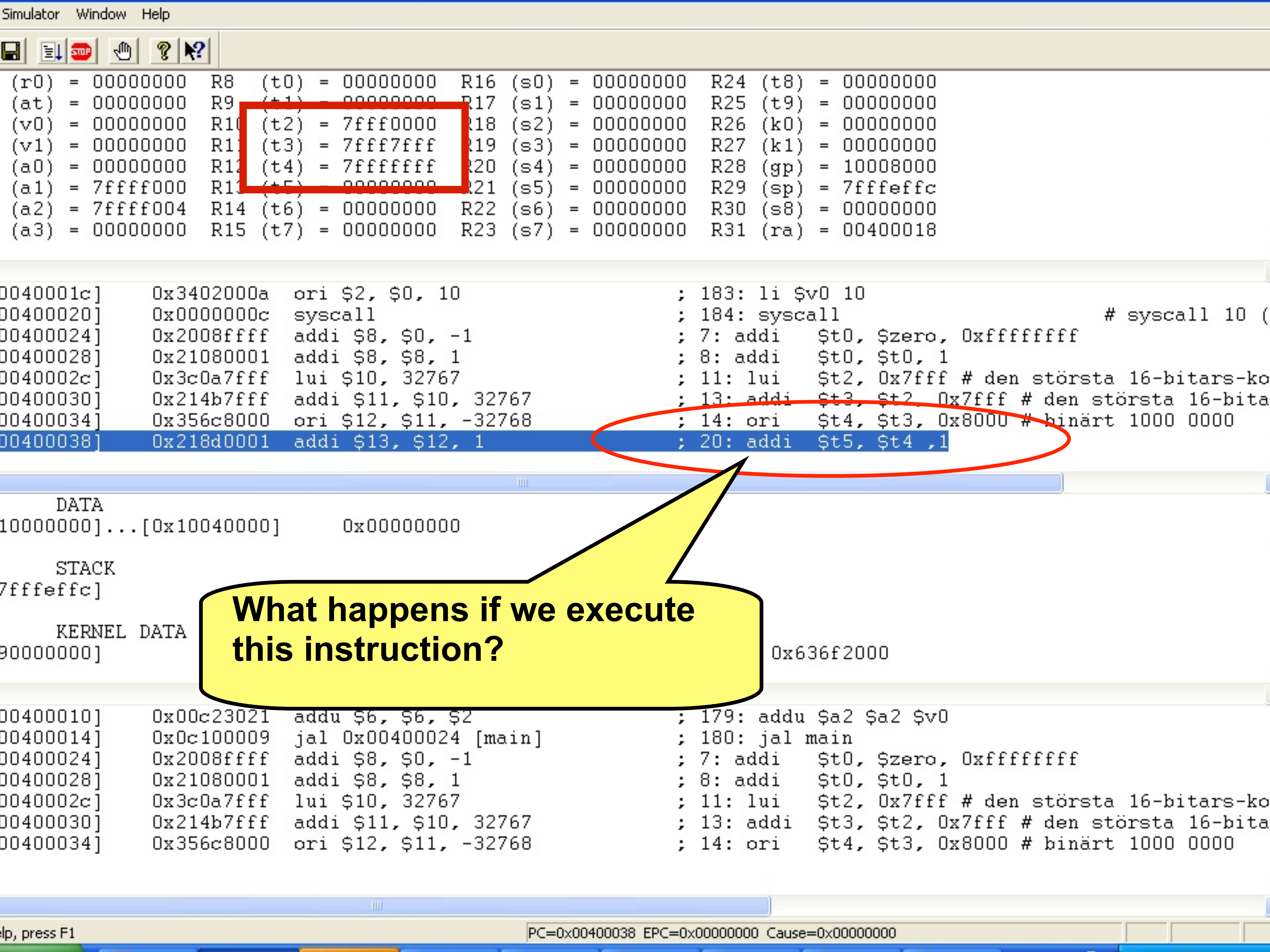
$t4 ← 0x7fffffff = $2.147.483.647_{10}$



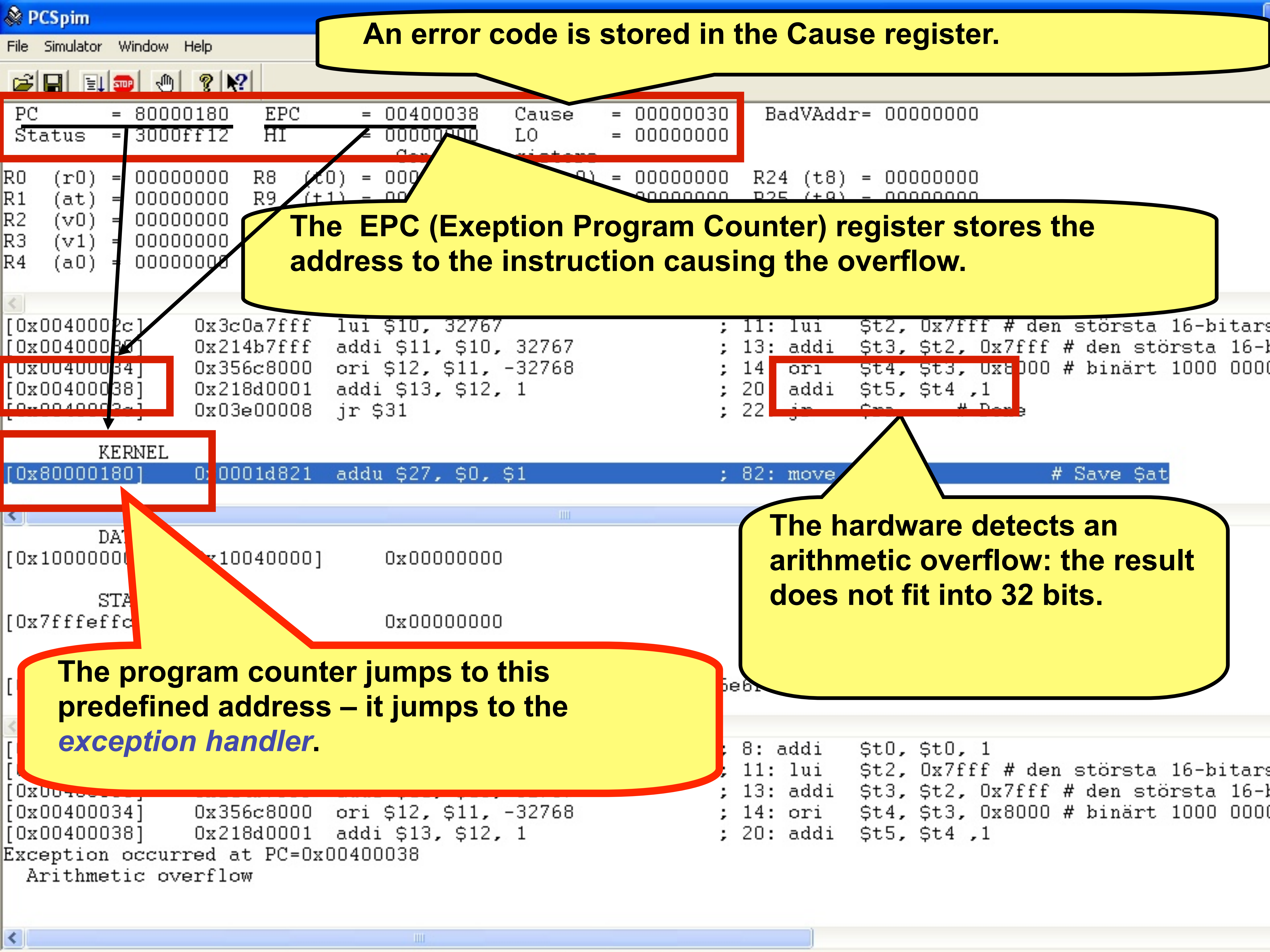
Ok, lets add 1 to this ”largest” number:

```
lui     $t2, 0x7fff                # 0x00007fff
addi    $t3, $t2, 0x7fff           # 0x7fff7fff
ori     $t4, $t3, 0x8000           # 0x7fffffff

addi    $t5, $t4, 1                # 0x7fffffff + 1
```



Simulator Window Help
(r0) = 00000000 R8 (t0) = 00000000 R16 (s0) = 00000000 R24 (t8) = 00000000
(at) = 00000000 R9 (t1) = 00000000 R17 (s1) = 00000000 R25 (t9) = 00000000
(v0) = 00000000 R10 (t2) = 7fff0000 R18 (s2) = 00000000 R26 (k0) = 00000000
(v1) = 00000000 R11 (t3) = 7fff7fff R19 (s3) = 00000000 R27 (k1) = 00000000
(a0) = 00000000 R12 (t4) = 7fffffff R20 (s4) = 00000000 R28 (gp) = 10008000
(a1) = 7ffff000 R13 (t5) = 00000000 R21 (s5) = 00000000 R29 (sp) = 7fffeffc
(a2) = 7ffff004 R14 (t6) = 00000000 R22 (s6) = 00000000 R30 (s8) = 00000000
(a3) = 00000000 R15 (t7) = 00000000 R23 (s7) = 00000000 R31 (ra) = 00400018
0040001c] 0x3402000a ori $2, $0, 10 ; 183: li $v0 10
00400020] 0x0000000c syscall ; 184: syscall # syscall 10 (
00400024] 0x2008ffff addi $8, $0, -1 ; 7: addi $t0, $zero, 0xffffffff
00400028] 0x21080001 addi $8, $8, 1 ; 8: addi $t0, $t0, 1
0040002c] 0x3c0a7fff lui $10, 32767 ; 11: lui $t2, 0x7fff # den största 16-bitars-ko
00400030] 0x214b7fff addi $11, $10, 32767 ; 13: addi $t3, $t2, 0x7fff # den största 16-bita
00400034] 0x356c8000 ori $12, $11, -32768 ; 14: ori $t4, $t3, 0x8000 # binärt 1000 0000
00400038] 0x218d0001 addi $13, $12, 1 ; 20: addi $t5, $t4 ,1
DATA
10000000]...[0x10040000] 0x00000000
STACK
7fffeffc]
KERNEL DATA
90000000] 0x636f2000
What happens if we execute this instruction?
00400010] 0x00c23021 addu $6, $6, $2 ; 179: addu $a2 $a2 $v0
00400014] 0x0c100009 jal 0x00400024 [main] ; 180: jal main
00400024] 0x2008ffff addi $8, $0, -1 ; 7: addi $t0, $zero, 0xffffffff
00400028] 0x21080001 addi $8, $8, 1 ; 8: addi $t0, $t0, 1
0040002c] 0x3c0a7fff lui $10, 32767 ; 11: lui $t2, 0x7fff # den största 16-bitars-ko
00400030] 0x214b7fff addi $11, $10, 32767 ; 13: addi $t3, $t2, 0x7fff # den största 16-bita
00400034] 0x356c8000 ori $12, $11, -32768 ; 14: ori $t4, $t3, 0x8000 # binärt 1000 0000
lp, press F1
PC=0x00400038 EPC=0x00000000 Cause=0x00000000

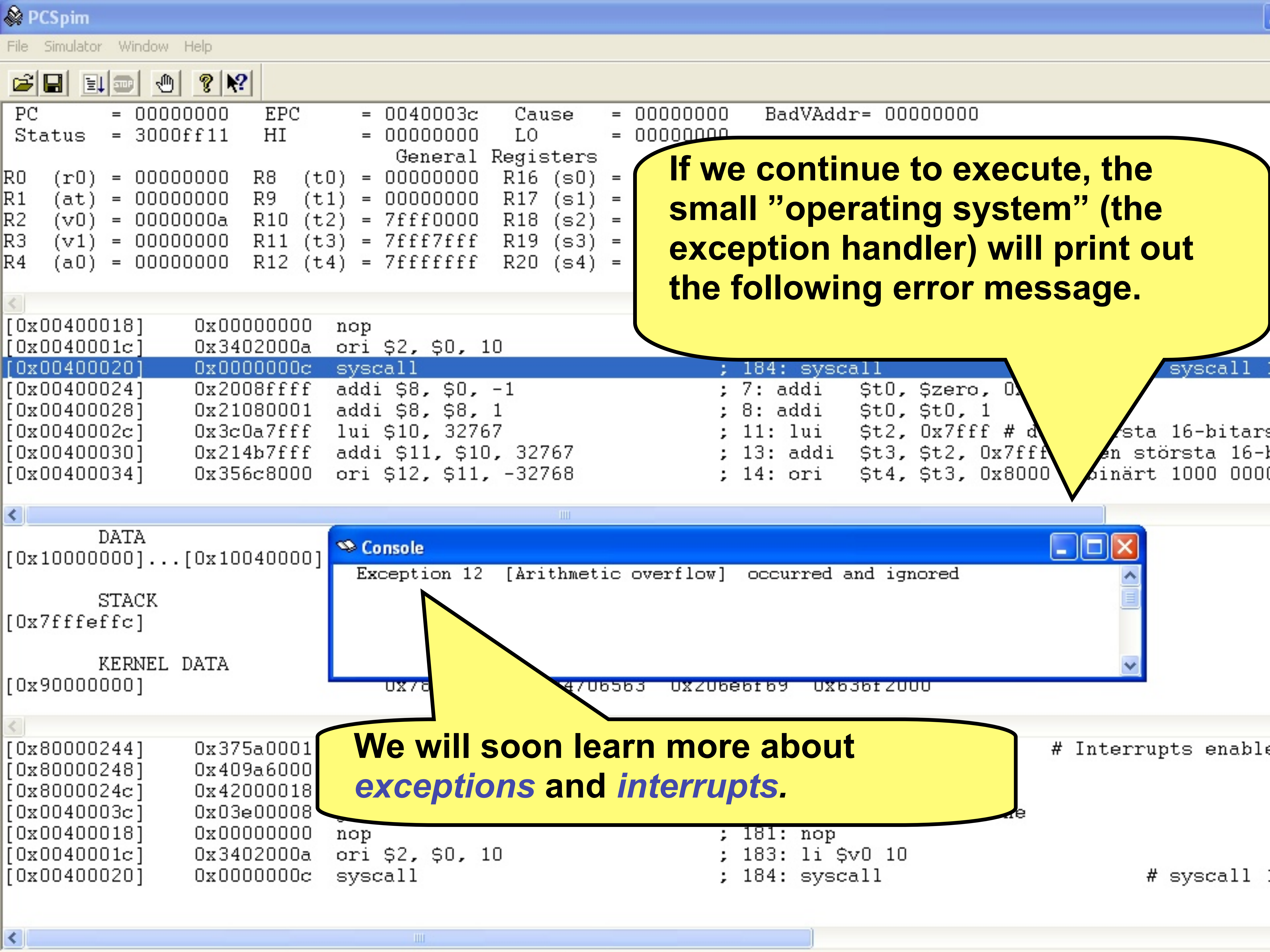


An error code is stored in the Cause register.

The EPC (Exeption Program Counter) register stores the address to the instruction causing the overflow.

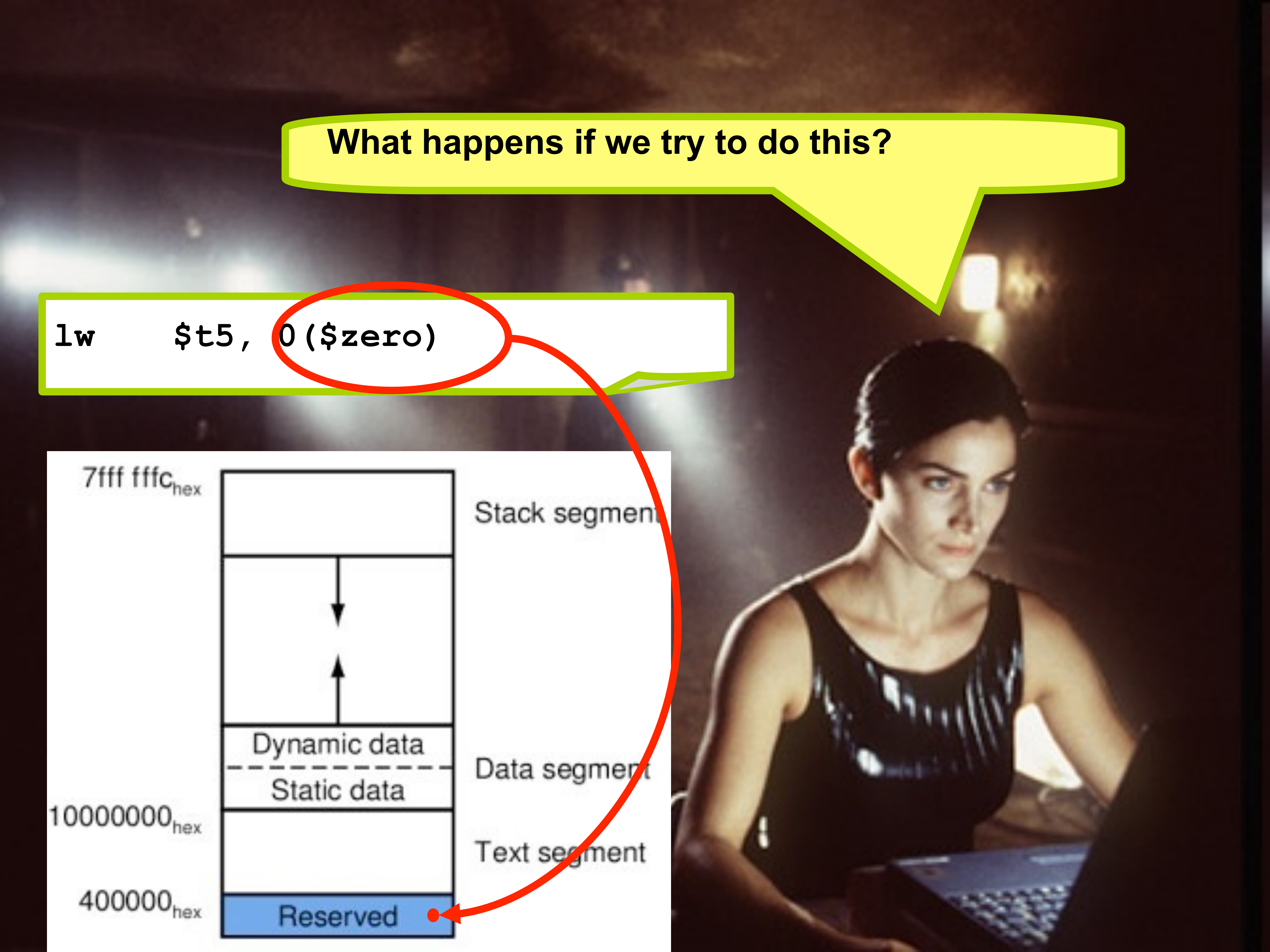
The hardware detects an arithmetic overflow: the result does not fit into 32 bits.

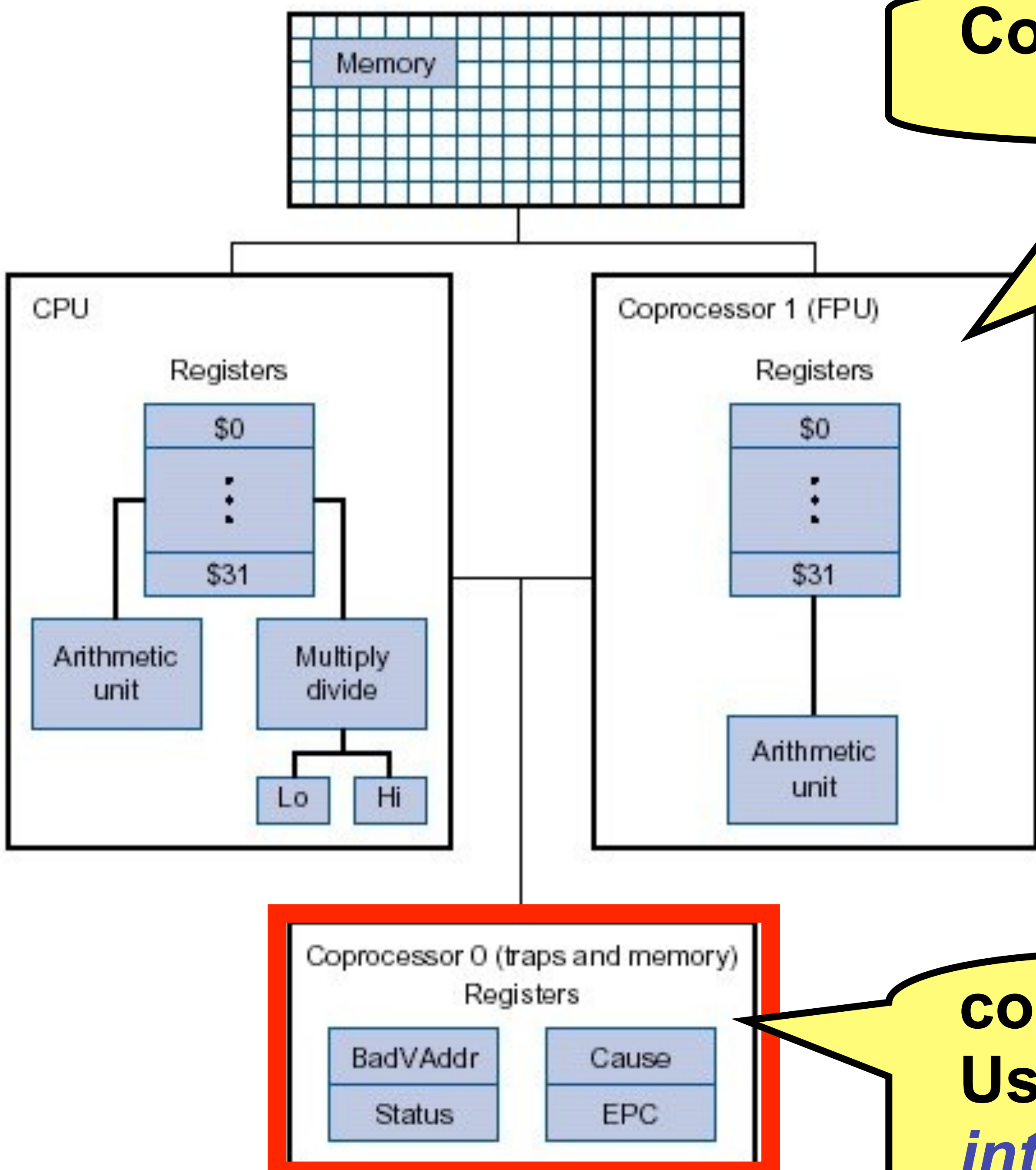
The program counter jumps to this predefined address – it jumps to the *exception handler*.



If we continue to execute, the small "operating system" (the exception handler) will print out the following error message.

We will soon learn more about *exceptions* and *interrupts*.


What happens if we try to do this?
lw $t5, 0($zero)
7fff fffc hex
Stack segment
Dynamic data
Static data
Data segment
10000000 hex
Text segment
400000 hex
Reserved

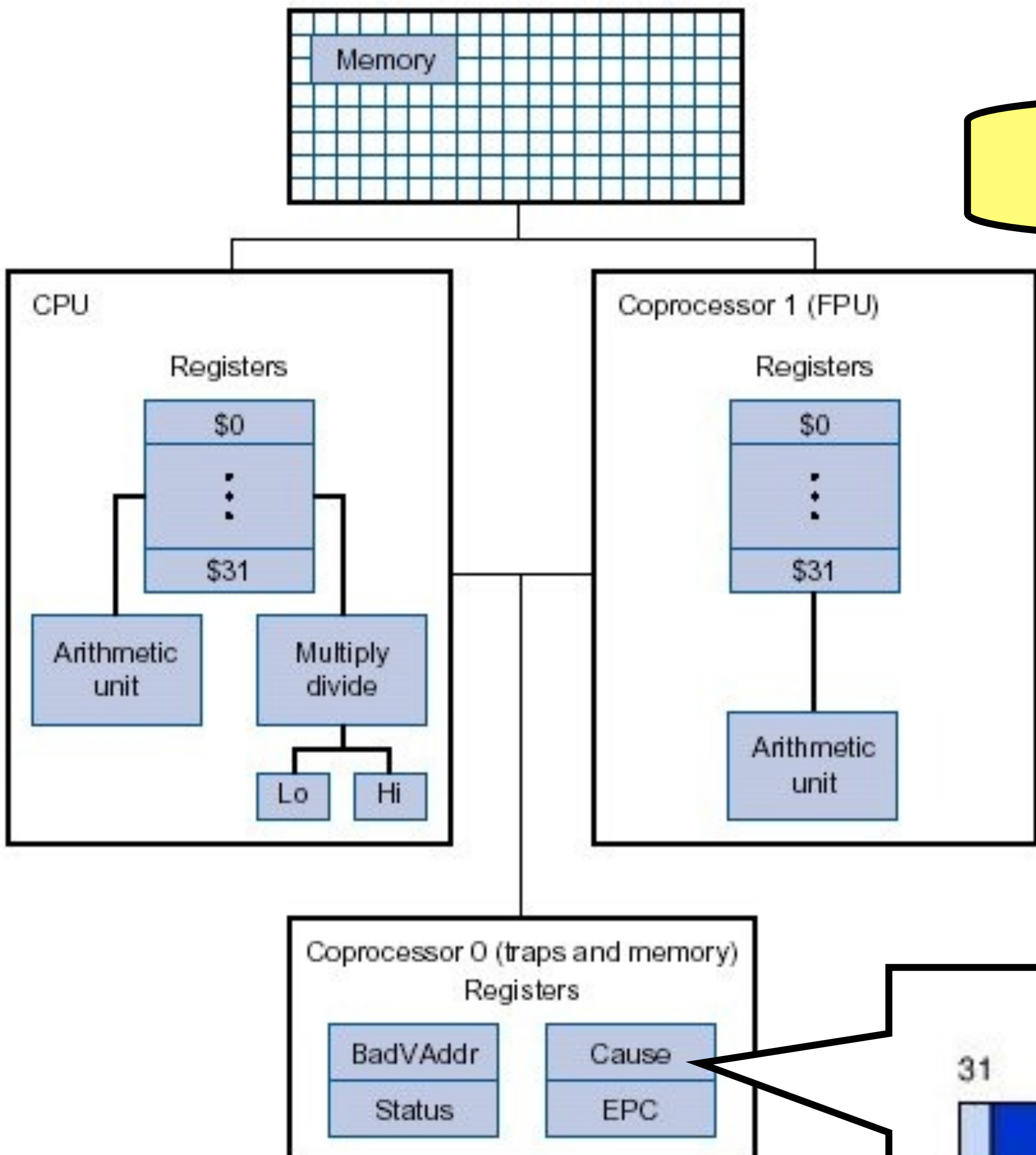


Coprocessor 1 – floating point unit.

coprocessor 0 – system monitoring. Used to manage *exceptions* and *interrupts*.

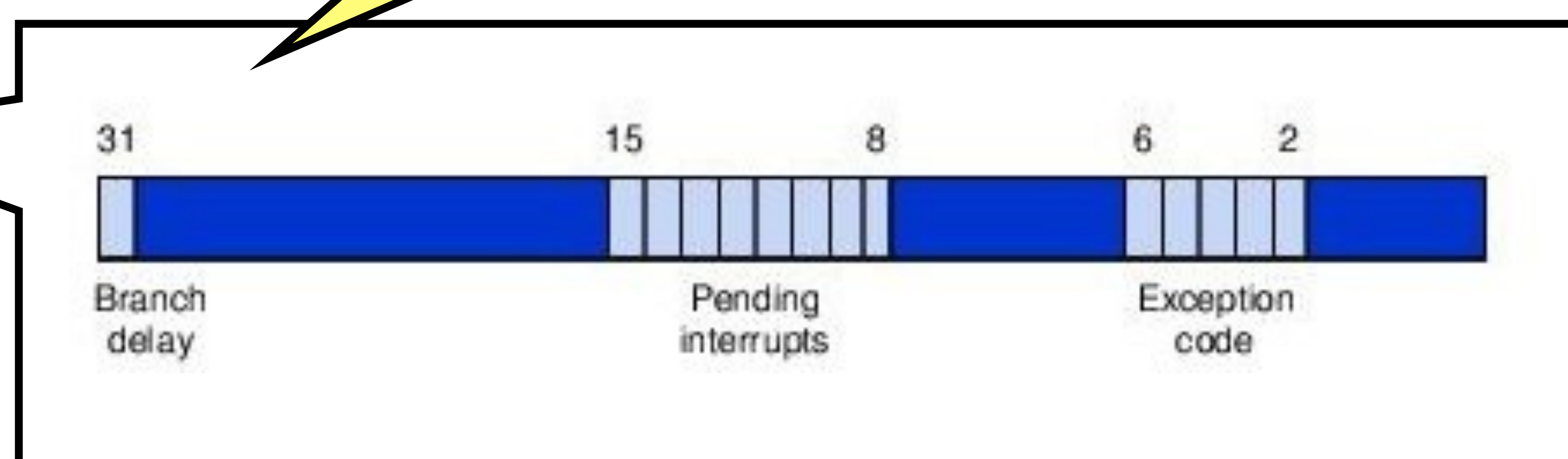
```
mfc0 $k0 $13          # Cause register
srl  $a0 $k0 2        # Extract ExcCode Field
andi $a0 $a0 0x1f
```

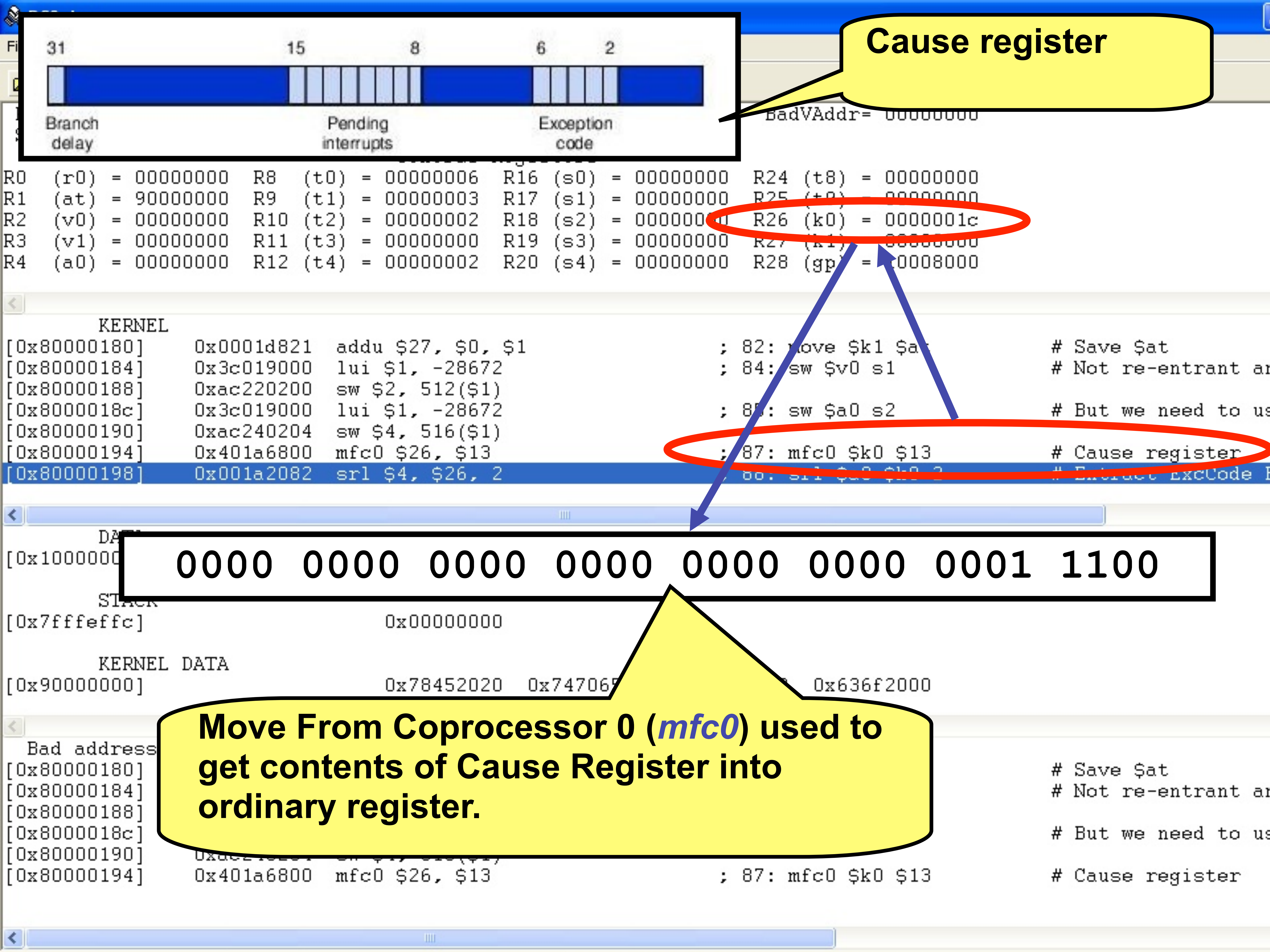
Move From Coprocessor 0: mfc0



Cause register:

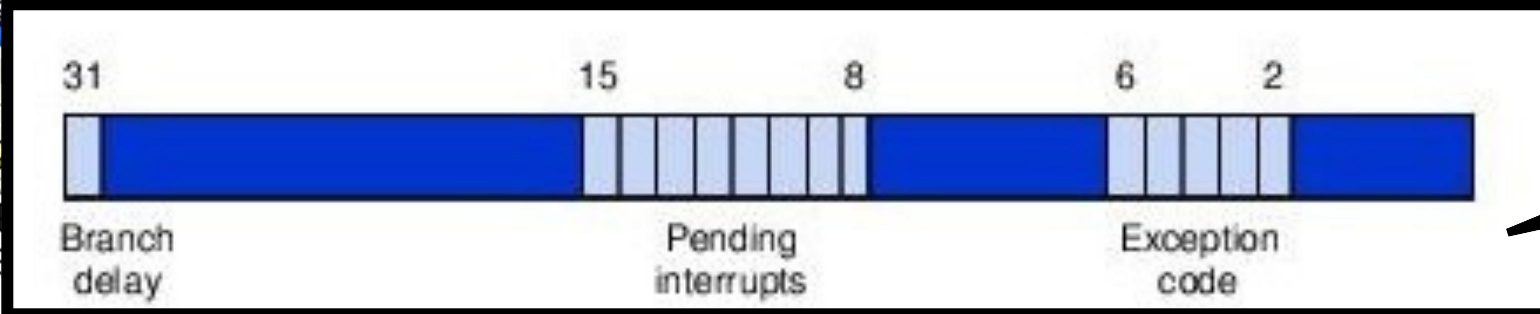
Register nr 13 in Coprocessor 0.





Cause register
31
15
8
6
2
Branch delay
Pending interrupts
Exception code
BadVAddr= 00000000
R0 (r0) = 00000000 R8 (t0) = 00000006 R16 (s0) = 00000000 R24 (t8) = 00000000
R1 (at) = 90000000 R9 (t1) = 00000003 R17 (s1) = 00000000
R2 (v0) = 00000000 R10 (t2) = 00000002 R18 (s2) = 00000000 R26 (k0) = 0000001c
R3 (v1) = 00000000 R11 (t3) = 00000000 R19 (s3) = 00000000
R4 (a0) = 00000000 R12 (t4) = 00000002 R20 (s4) = 00000000 R28 (gp) = 10008000
KERNEL
[0x80000180] 0x0001d821 addu $27, $0, $1 ; 82: move $k1 $at # Save $at
[0x80000184] 0x3c019000 lui $1, -28672 ; 84: sw $v0 s1 # Not re-entrant an
[0x80000188] 0xac220200 sw $2, 512($1)
[0x8000018c] 0x3c019000 lui $1, -28672 ; 85: sw $a0 s2 # But we need to us
[0x80000190] 0xac240204 sw $4, 516($1)
[0x80000194] 0x401a6800 mfc0 $26, $13 ; 87: mfc0 $k0 $13 # Cause register
[0x80000198] 0x001a2082 srl $4, $26, 2
0000 0000 0000 0000 0000 0000 0001 1100
[0x7fffeffc] 0x00000000
KERNEL DATA
[0x90000000] 0x78452020 0x636f2000
Bad address
[0x80000180] # Save $at
[0x80000184] # Not re-entrant an
[0x80000188]
[0x8000018c] # But we need to us
[0x80000190]
[0x80000194] 0x401a6800 mfc0 $26, $13 ; 87: mfc0 $k0 $13 # Cause register
Move From Coprocessor 0 (mfc0) used to get contents of Cause Register into ordinary register.

Cause register



BadVAddr= 00000000

```
R0  (r0) = 00000000  R8  (t0) = 00000006  R16 (s0) = 00000000  R24 (t8) = 00000000
R1  (at) = 90000000  R9  (t1) = 00000003  R17 (s1) = 00000000  R25 (t9) = 00000000
R2  (v0) = 00000000  R10 (t2) = 00000002  R18 (s2) = 00000000  R26 (k0) = 0000001c
R3  (v1) = 00000000  R11 (t3) = 00000000  R19 (s3) = 00000000  R27 (k1) = 00000000
R4  (a0) = 00000007  R12 (t4) = 00000002  R20 (s4) = 00000000  R28 (gp) = 10008000
```

```
[0x80000180]  0x0001d821  addu $27, $0, $1      ; 82: move $k1 $at      # Save $at
[0x80000184]  0x3c019000  lui $1, -28672        ; 84: sw $v0 s1         # Not re-entrant ar
[0x80000188]  0xac220200  sw $2, 512($1)
[0x8000018c]  0x3c019000  lui $1, -28672        ; 85: sw $a0 s2         # But we need to us
[0x80000190]  0xac240204  sw $4, 516($1)
[0x80000194]  0x401a6800  mfc0 $26, $13         ; 87: mfc0 $k0 $13      # Cause register
[0x80000198]  0x001a2082  srl $4, $26, 2        ; 88: srl $a0 $k0 2     # Extract ExcCode F
[0x8000019c]  0x3084001f  andi $4, $4, 31       ; 89: andi $a0 $a0 0x1f
```

DATA
[0x1000000

0000 0000 0000 0000 0000 0000 0001 1100

STACK
[0x7fffeffc] 0x00000000

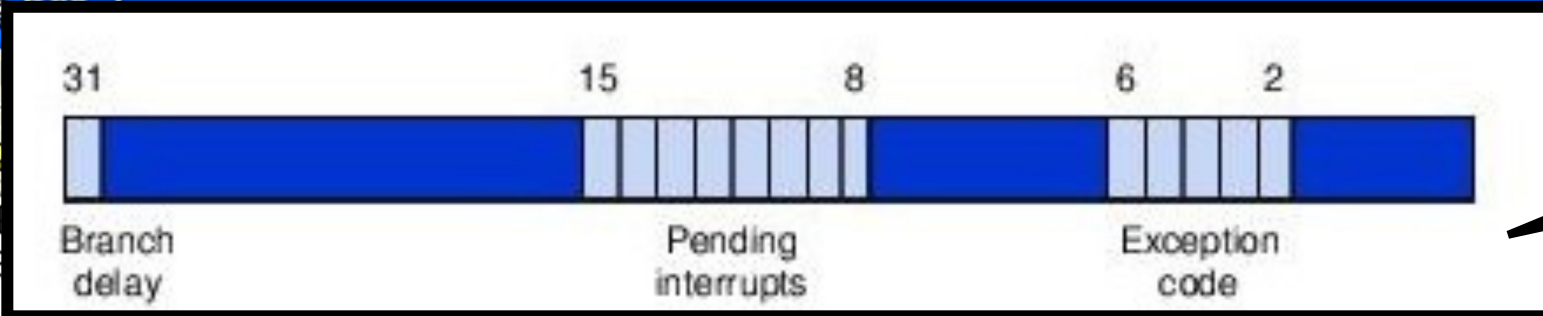
KI
[0x9000000

0000 0000 0000 0000 0000 0000 0000 0111

```
[0x80000180]  0x0001d821  addu $27, $0, $1      ; 82: move $k1 $at      # Save $at
[0x80000184]  0x3c019000  lui $1, -28672        ; 84: sw $v0 s1         # Not re-entrant ar
[0x80000188]  0xac220200  sw $2,
[0x8000018c]  0x3c019000  lui $1                ; 85: sw $a0 s2         # But we need to us
[0x80000
[0x80000                                                                # Cause register
[0x80000198]  0x001a2082  srl $4, $26, 2        ; 88: srl $a0 $k0 2     # Extract ExcCode F
```

Cause Register Shifted right 2 bits

5 Bit Exception Code



Cause register

```
R0  (r0) = 00000000  R8  (t0) = 00000006  R16 (s0) = 00000000  R24 (t8) = 00000000
R1  (at) = 90000000  R9  (t1) = 00000003  R17 (s1) = 00000000  R25 (t9) = 00000000
R2  (v0) = 00000000  R10 (t2) = 00000002  R18 (s2) = 00000000  R26 (k0) = 0000001c
R3  (v1) = 00000000  R11 (t3) = 00000000  R19 (s3) = 00000000  R27 (k1) = 00000000
R4  (a0) = 00000007  R12 (t4) = 00000002  R20 (s4) = 00000000  R28 (gp) = 10008000
```

```
[0x80000184]  0x3c019000  lui $1, 28672      ; 84: sw $v0 s1        # Not re-entrant an
[0x80000188]  0xac220200  sw $2, 512($1)
                                             ; 85: sw $a0 s2        # But we need to us
                                             ; 87: mfc0 $k0 $13     # Cause register
                                             ; 88: srl $a0 $k0 2    # Extract ExcCode F
                                  $4, 31     ; 89: andi $a0 $a0 0x1f
[0x800001a0]  0x34020004  ori $2, $0, 4      ; 93: li $v0 4         # syscall 4 (print_
```

Bit Mask **used to mask out only the desired bits**

```
      0000 0000 0000 0000 0000 0000 0000 0111

      xxxx xxxx xxxx xxxx xxxx xxxx xxx0 0111
AND   0000 0000 0000 0000 0000 0000 0001 1111
      ---------------------------------------
      0000 0000 0000 0000 0000 0000 0000 0111
```

Exception Code == 7_{10}

At the beginning of the the default exception handler for spim (*exceptions.s*).

```
        .kdata
__m1_:  .asciiz "  Exception "
__m2_:  .asciiz " occurred and ignored\n"

__e0_:  .asciiz "  [Interrupt] "
__e1_:  .asciiz    "  [TLB]"
__e2_:  .asciiz    "  [TLB]"
__e3_:  .asciiz    "  [TLB]"
__e4_:  .asciiz    "  [Address error in inst/data fetch] "
__e5_:  .asciiz    "  [Address error in store] "
__e6_:  .asciiz    "  [Bad instruction address] "
__e7_:  .asciiz    "  [Bad data address] "
__e8_:  .asciiz    "  [Error in syscall] "
__e9_:  .asciiz    "  [Breakpoint] "
__e10_: .asciiz    "  [Reserved instruction] "
__e11_: .asciiz    ""
__e12_: .asciiz    "  [Arithmetic overflow] "
.
.
.
__e30_: .asciiz    "  [Cache]"
__e31_: .asciiz    ""
__excp: .word __e0_, __e1_, __e2_, __e3_, __e4_, __e5_, __e6_, __e7_, __e8_, __e9_
        .word __e10_, __e11_, __e12_, __e13_, __e14_, __e15_, __e16_, __e17_, __e18_,
        .word __e19_, __e20_, __e21_, __e22_, __e23_, __e24_, __e25_, __e26_, __e27_,
        .word __e28_, __e29_, __e30_, __e31_
```

A number of strings

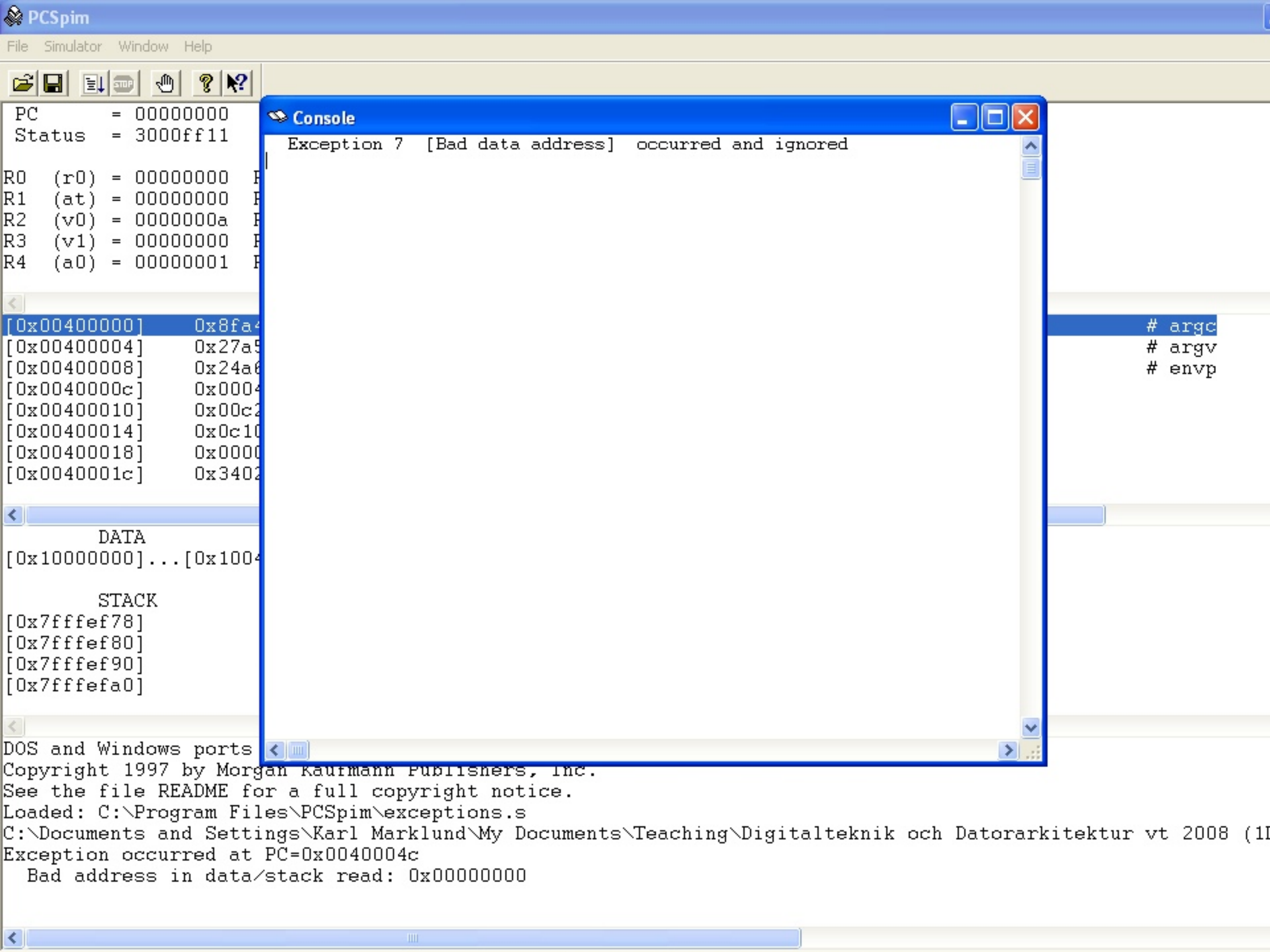
An array of strings

At the beginning of the the default exception handler for spim (*exceptions.s*).

```
        .kdata
__m1_:  .asciiz "  Exception "
__m2_:  .asciiz " occurred and ignored\n"

__e0_:  .asciiz "  [Interrupt] "
__e1_:  .asciiz    "  [TLB]"
__e2_:  .asciiz    "  [TLB]"
__e3_:  .asciiz    "  [TLB]"
__e4_:  .asciiz    "  [Address error in inst/data fetch] "
__e5_:  .asciiz    "  [Address error in store] "
__e6_:  .asciiz    "  [Bad instruction address] "
__e7_:  .asciiz    "  [Bad data address] "
__e8_:  .asciiz    "  [Error in syscall] "
__e9_:  .asciiz    "  [Breakpoint] "
__e10_: .asciiz    "  [Reserved instruction] "
__e11_: .asciiz    ""
__e12_: .asciiz    "  [Arithmetic overflow] "
.
.
.
__e30_: .asciiz    "  [Cache]"
__e31_: .asciiz    ""
__excp: .word __e0_, __e1_, __e2_, __e3_, __e4_, __e5_, __e6_, __e7_, __e8_, __e9_
        .word __e10_, __e11_, __e12_, __e13_, __e14_, __e15_, __e16_, __e17_, __e18_,
        .word __e19_, __e20_, __e21_, __e22_, __e23_, __e24_, __e25_, __e26_, __e27_,
        .word __e28_, __e29_, __e30_, __e31_
```

Error message for exception nr 7


PCSpim
File Simulator Window Help
PC = 00000000
Status = 3000ff11
R0 (r0) = 00000000
R1 (at) = 00000000
R2 (v0) = 0000000a
R3 (v1) = 00000000
R4 (a0) = 00000001
[0x00400000] 0x8fa4
[0x00400004] 0x27a5
[0x00400008] 0x24a6
[0x0040000c] 0x0004
[0x00400010] 0x00c2
[0x00400014] 0x0c10
[0x00400018] 0x0000
[0x0040001c] 0x3402
argc
argv
envp
DATA
[0x10000000]...[0x1004
STACK
[0x7fffef78]
[0x7fffef80]
[0x7fffef90]
[0x7fffefa0]
Console
Exception 7 [Bad data address] occurred and ignored
DOS and Windows ports
Copyright 1997 by Morgan Kaufmann Publishers, Inc.
See the file README for a full copyright notice.
Loaded: C:\Program Files\PCSpim\exceptions.s
C:\Documents and Settings\Karl Marklund\My Documents\Teaching\Digitalteknik och Datorarkitektur vt 2008 (1D
Exception occurred at PC=0x0040004c
Bad address in data/stack read: 0x00000000

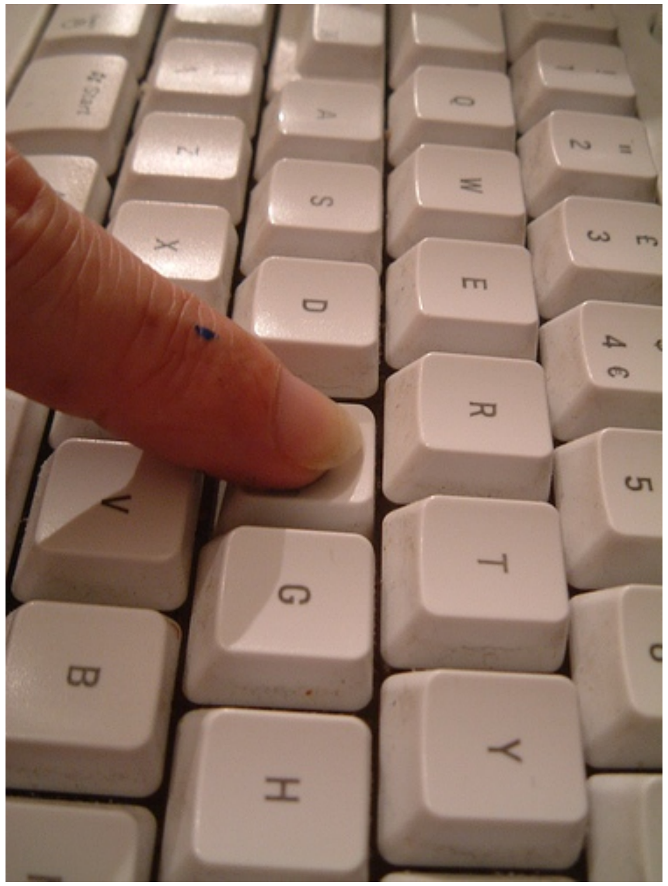
Overflow and bad data adress are examples of *internal* errors in a program

For the same input, these internal errors will occurs at the same place every time.

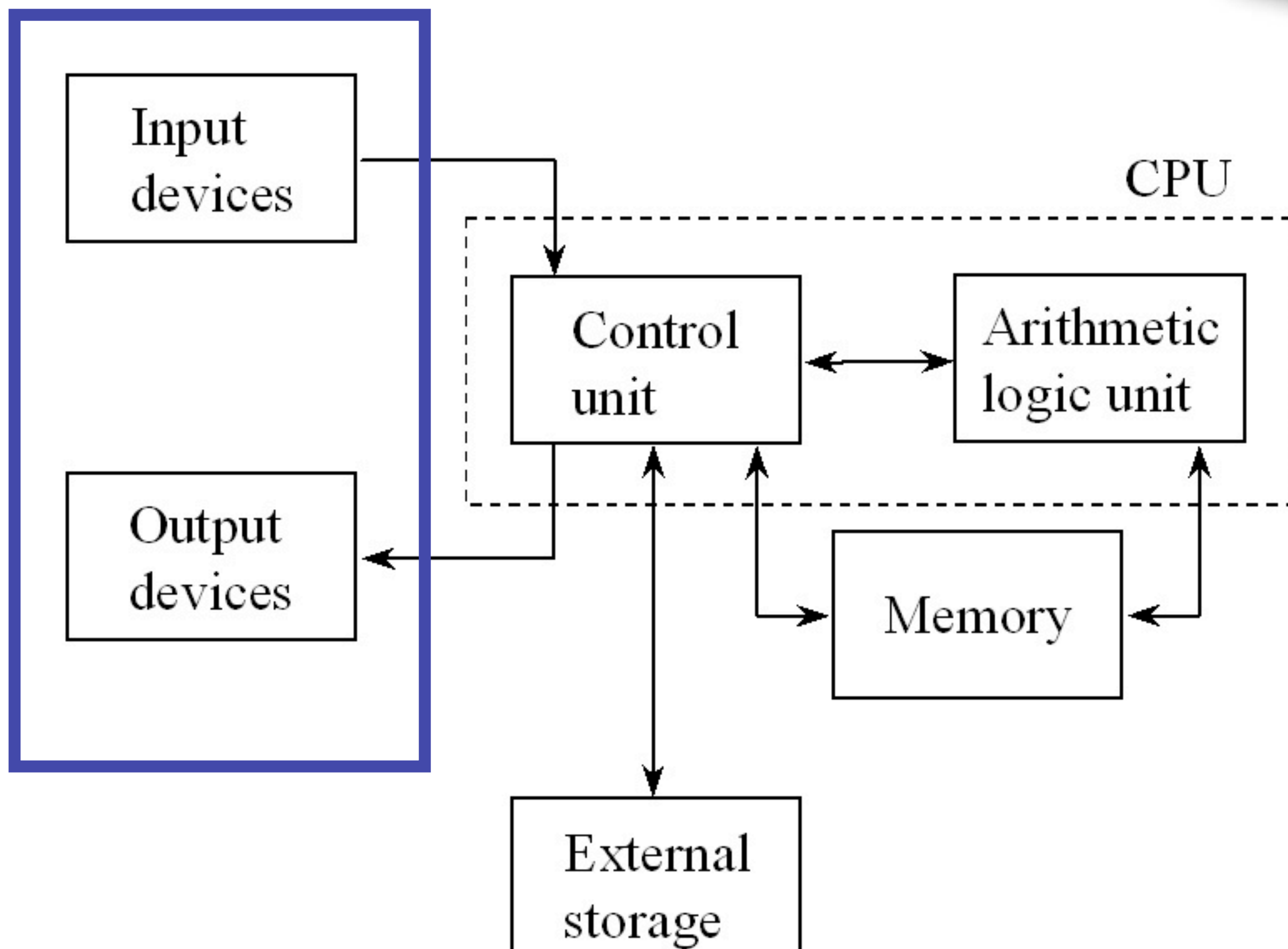
***Exceptions* are *internal* and *synchronous*.**

I/O enhet

Processor

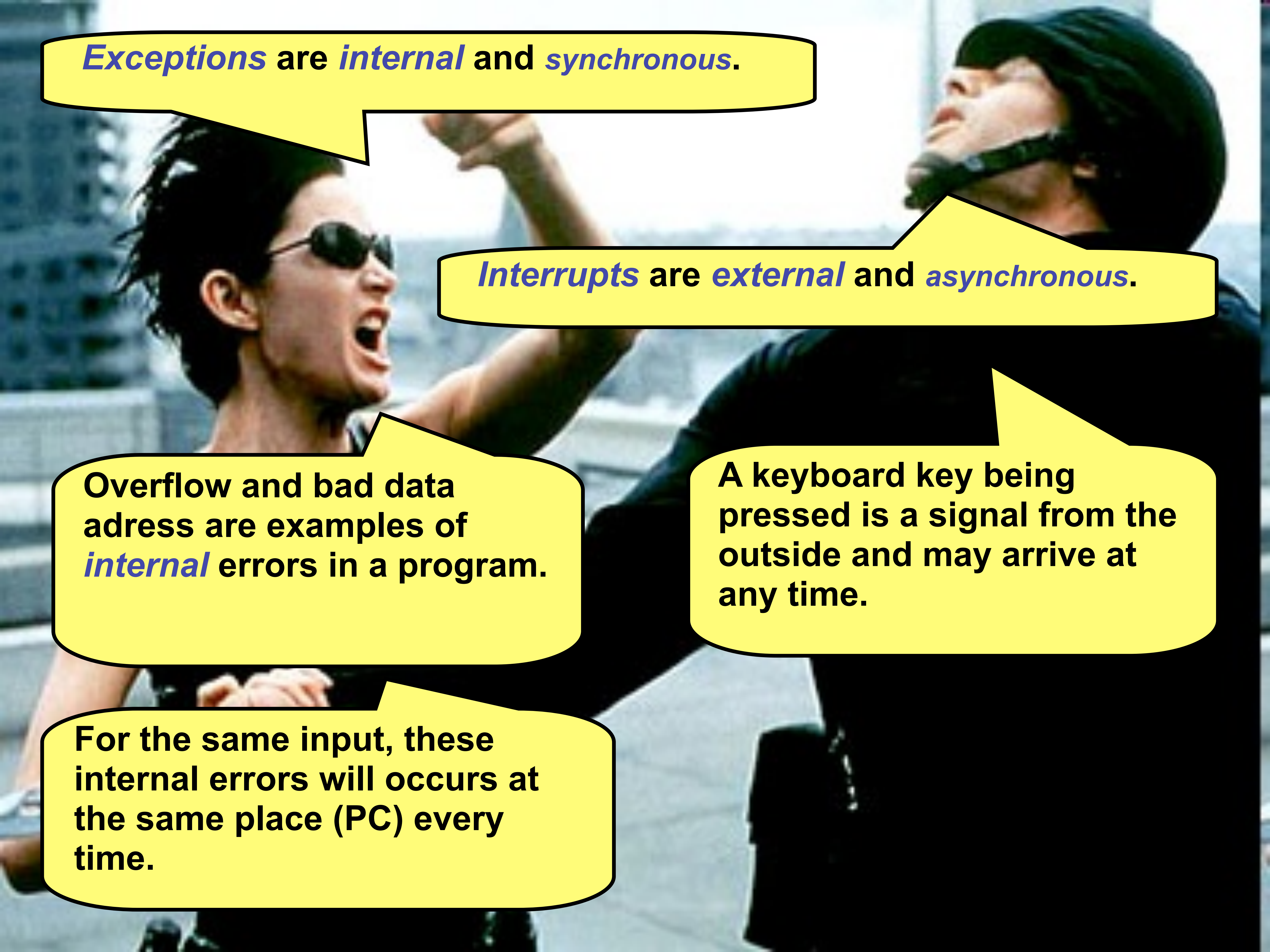


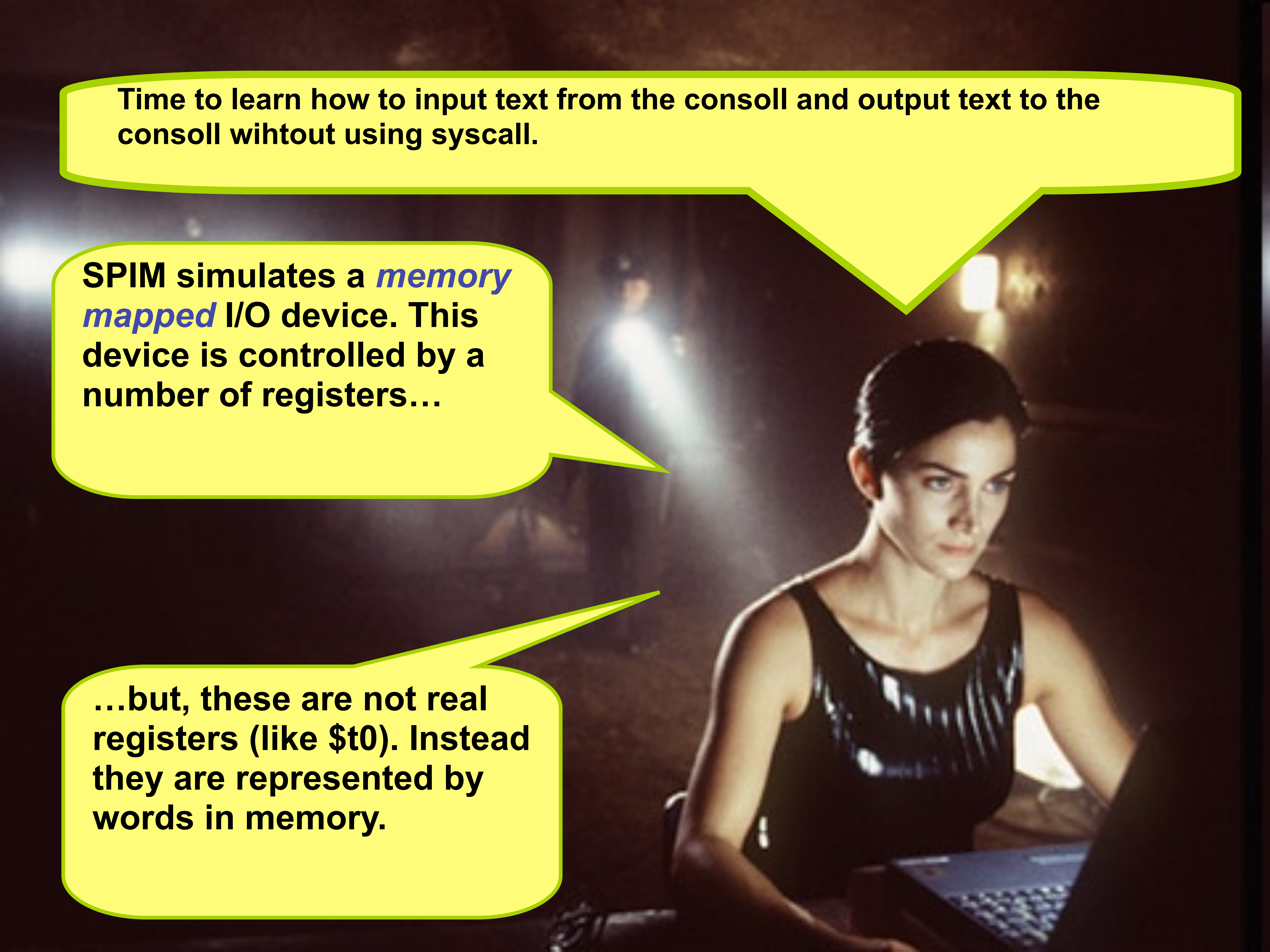
Communication?



Programs we write...

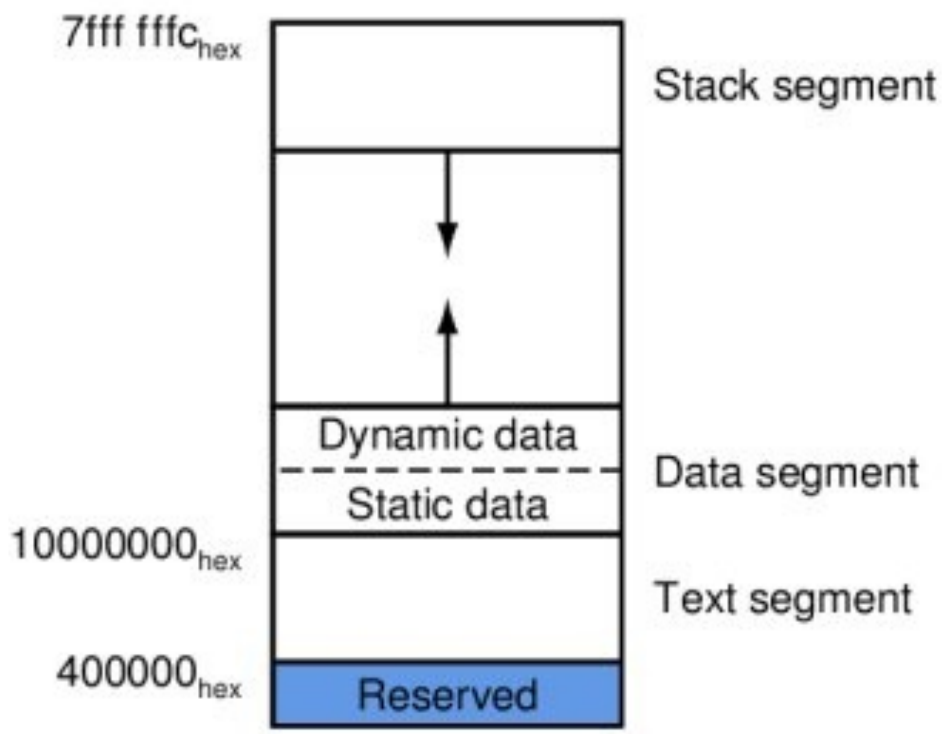

A keyboard key being pressed is a signal from the outside and may arrive at any time.
Reading and writing to a and from a hard disk is very similar in nature.
Interrupts are *external* and *asynchronous.*


Exceptions are *internal* and *synchronous*.
Interrupts are *external* and *asynchronous*.
Overflow and bad data adress are examples of *internal* errors in a program.
A keyboard key being pressed is a signal from the outside and may arrive at any time.
For the same input, these internal errors will occurs at the same place (PC) every time.


Time to learn how to input text from the consoll and output text to the consoll wihtout using syscall.
SPIM simulates a memory mapped I/O device. This device is controlled by a number of registers…
…but, these are not real registers (like $t0). Instead they are represented by words in memory.

Memoy Mapped registers for output of a charactes

Output – *Transmitting* characters



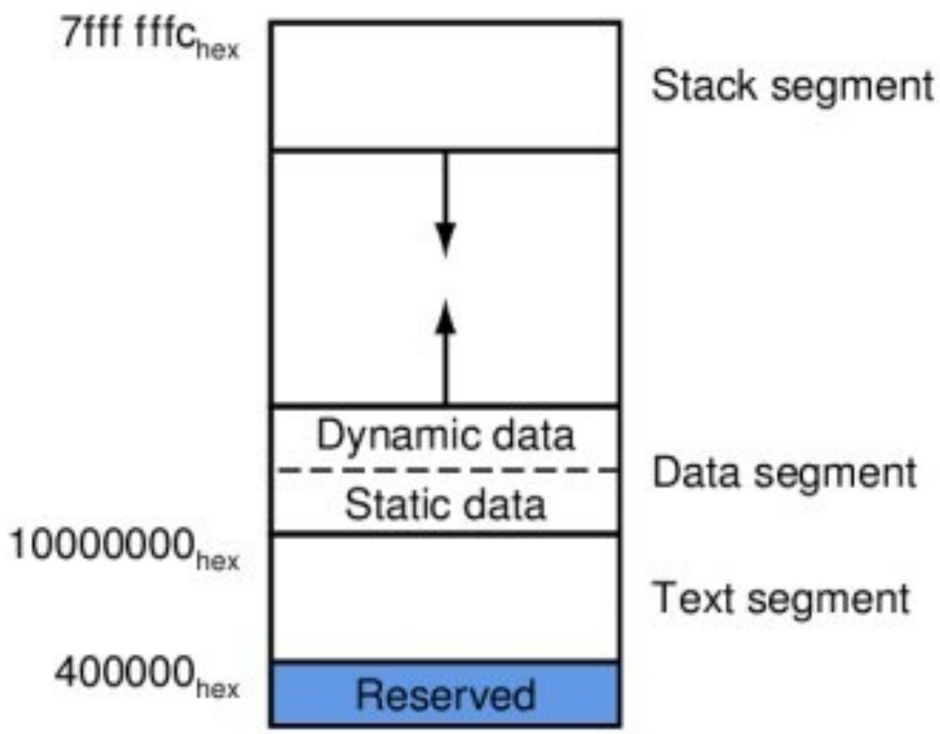
Ready-Bit: automatically set to 1 if the device is ready to transmit a new character, that is, if the character stored in transmitter data has been consued.

Address in memory for the memory mapped register Transmitter control.

Address in memory for the memory mapped register Transmitter data.

ASCII value of character to output.

Minnes-mappade register för inmatning av tecken.



Input – *Receiving* characters

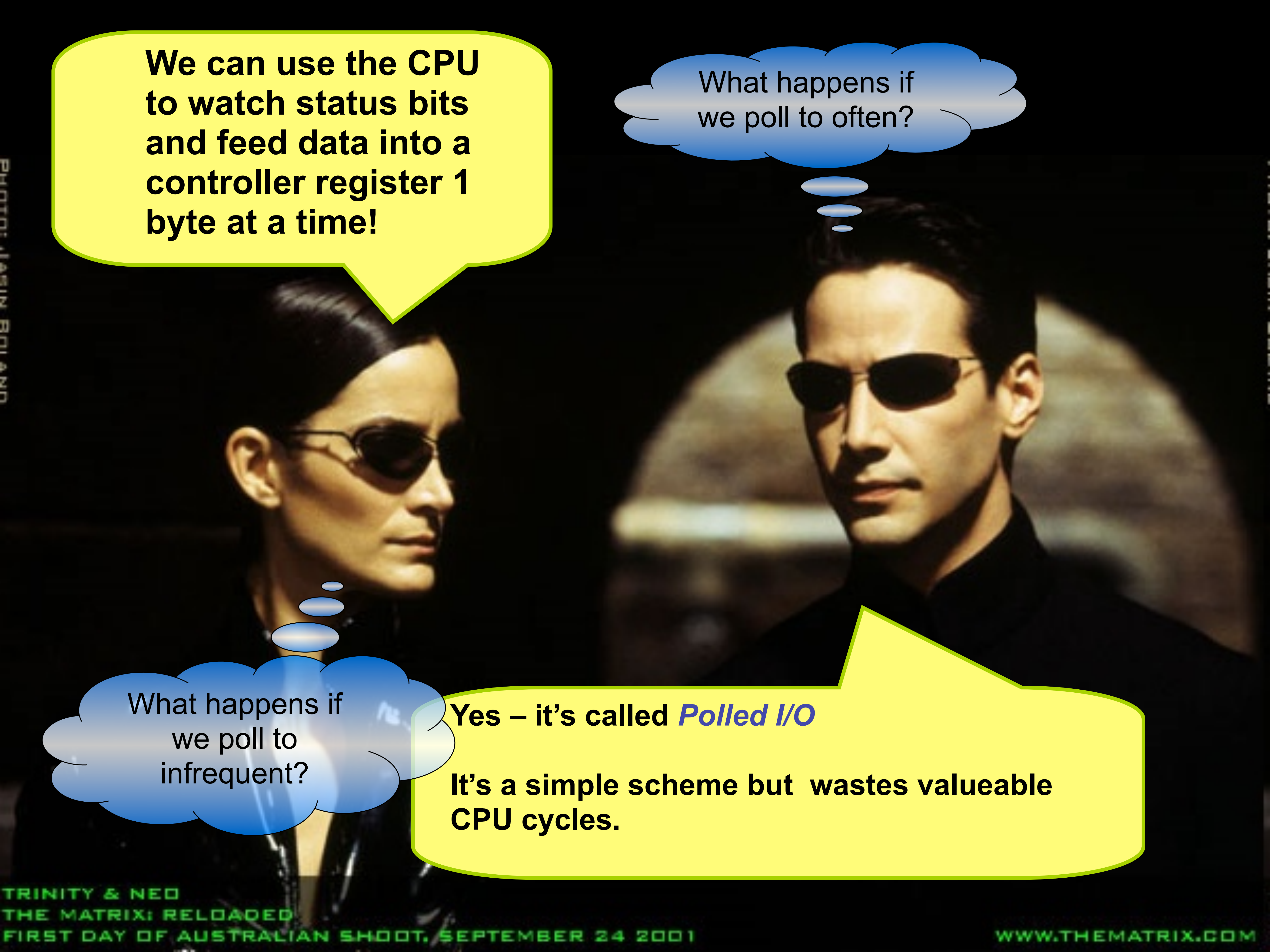
Ready-Bit: automatically set to 1 when a new character arrives. Automatically set to 0 when the charecters is consumed (loaded) from Receiver data.

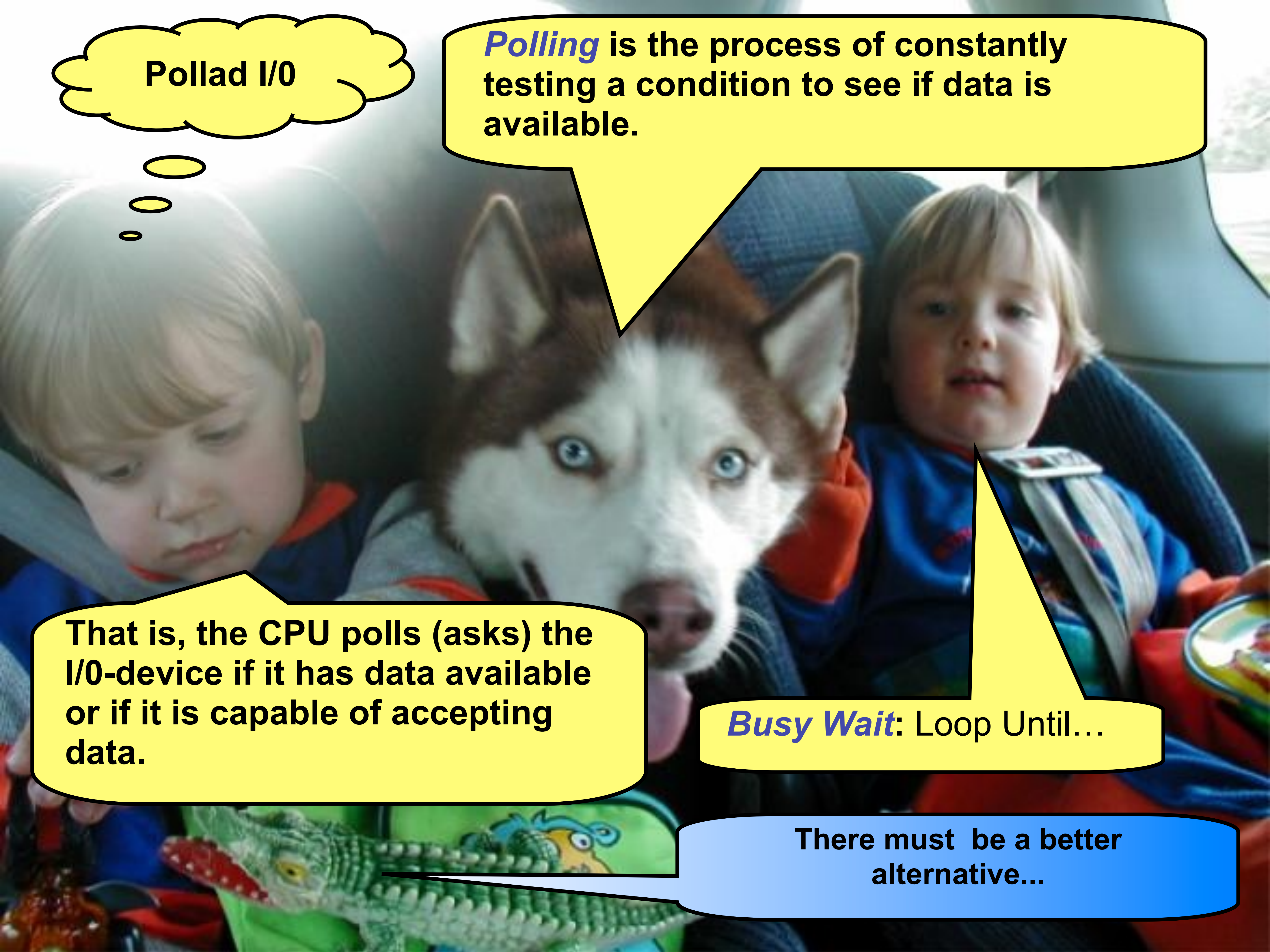
Address in memory for the memory mapped register Receiver control.

Address in memory for the memory mapped register Receiver data.

ASCII value of the received character.

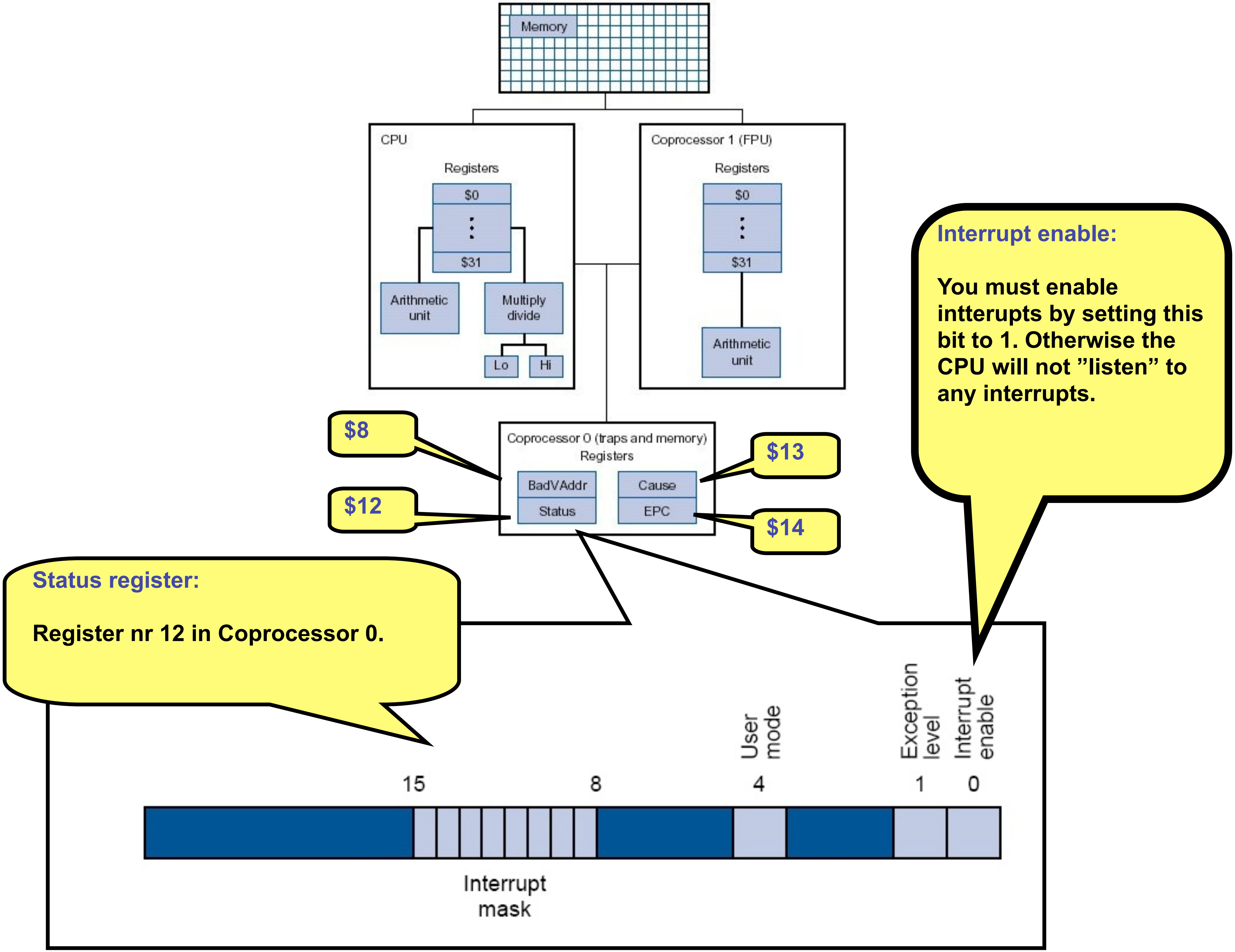

Are we there yet?
Are we there yet?
Are we there yet?
Are we there yet?
Are we there yet?
Are we there yet?
Are we there yet?


We can use the CPU to watch status bits and feed data into a controller register 1 byte at a time!
What happens if we poll to often?
What happens if we poll to infrequent?
Yes – it's called *Polled I/O*
It's a simple scheme but wastes valueable CPU cycles.
TRINITY & NEO
THE MATRIX: RELOADED
FIRST DAY OF AUSTRALIAN SHOOT, SEPTEMBER 24 2001
WWW.THEMATRIX.COM

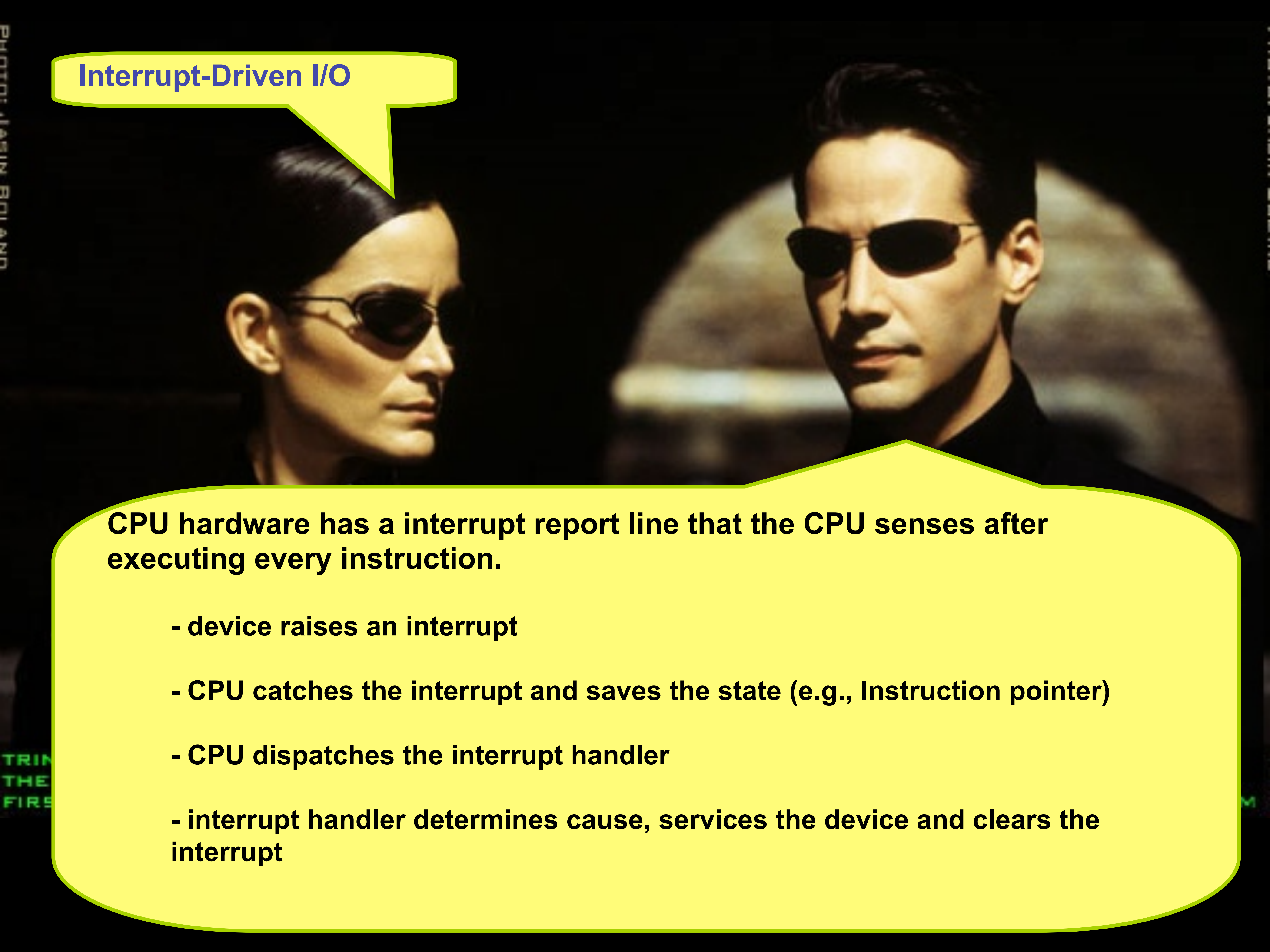

Pollad I/0
Polling is the process of constantly testing a condition to see if data is available.
That is, the CPU polls (asks) the I/0-device if it has data available or if it is capable of accepting data.
Busy Wait: Loop Until…
There must be a better alternative...

Interrup-enable: If you want the I/O device to generate an interrupt when the device is ready to output a new character – set this bit to 1.

Interrup-enable: If you want the I/O device to generate an interrup when a new character has arrived – set this bit to 1.


Memory
CPU
Registers
$0
$31
Arithmetic unit
Multiply divide
Lo
Hi
Coprocessor 1 (FPU)
Registers
$0
$31
Arithmetic unit
Coprocessor 0 (traps and memory)
Registers
BadVAddr
Cause
Status
EPC
$8
$12
$13
$14
Interrupt enable:
You must enable intterupts by setting this bit to 1. Otherwise the CPU will not ”listen” to any interrupts.
Status register:
Register nr 12 in Coprocessor 0.
User mode
Exception level
Interrupt enable
15
8
4
1
0
Interrupt mask

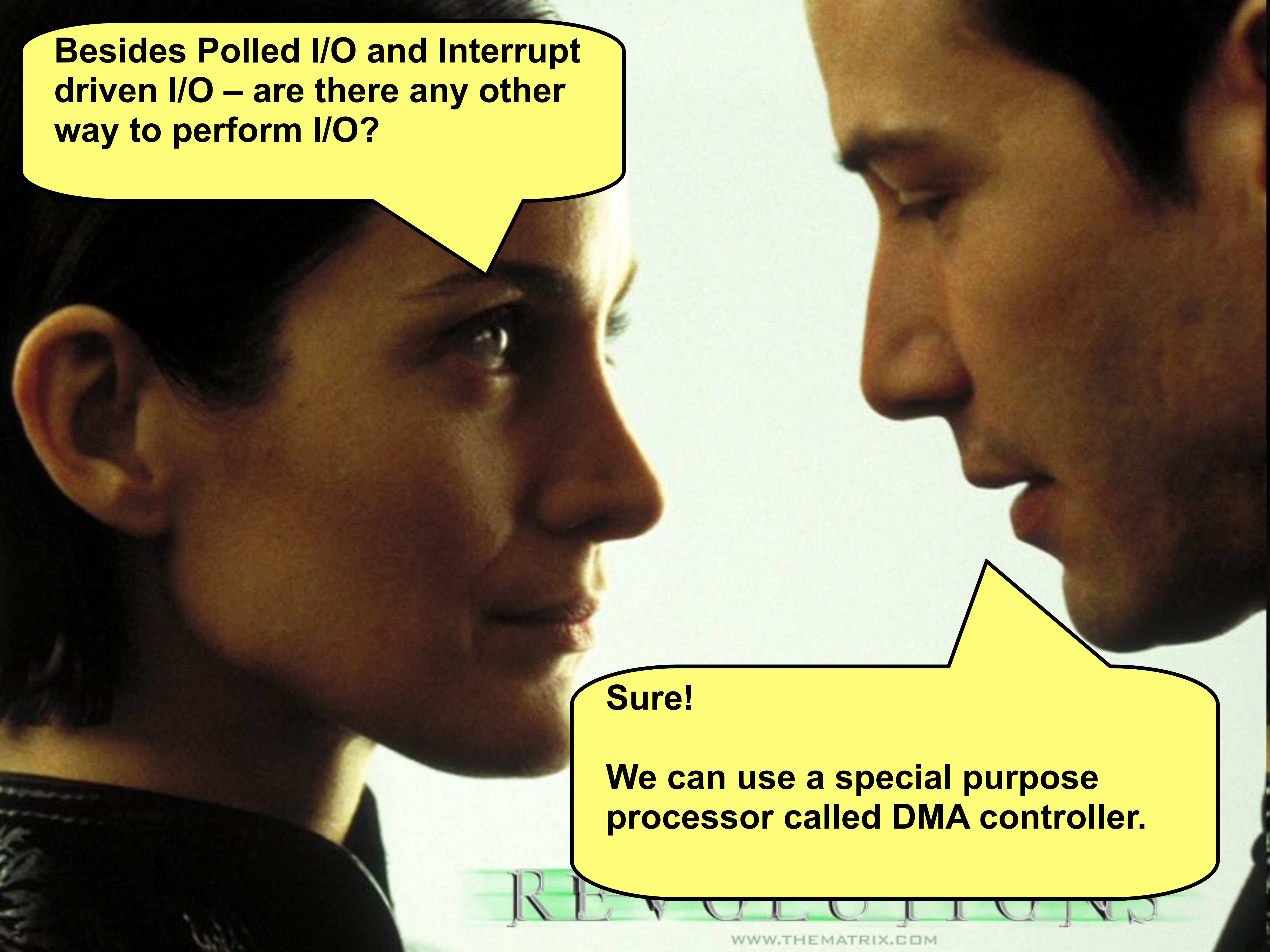

We have arrived at our destination!



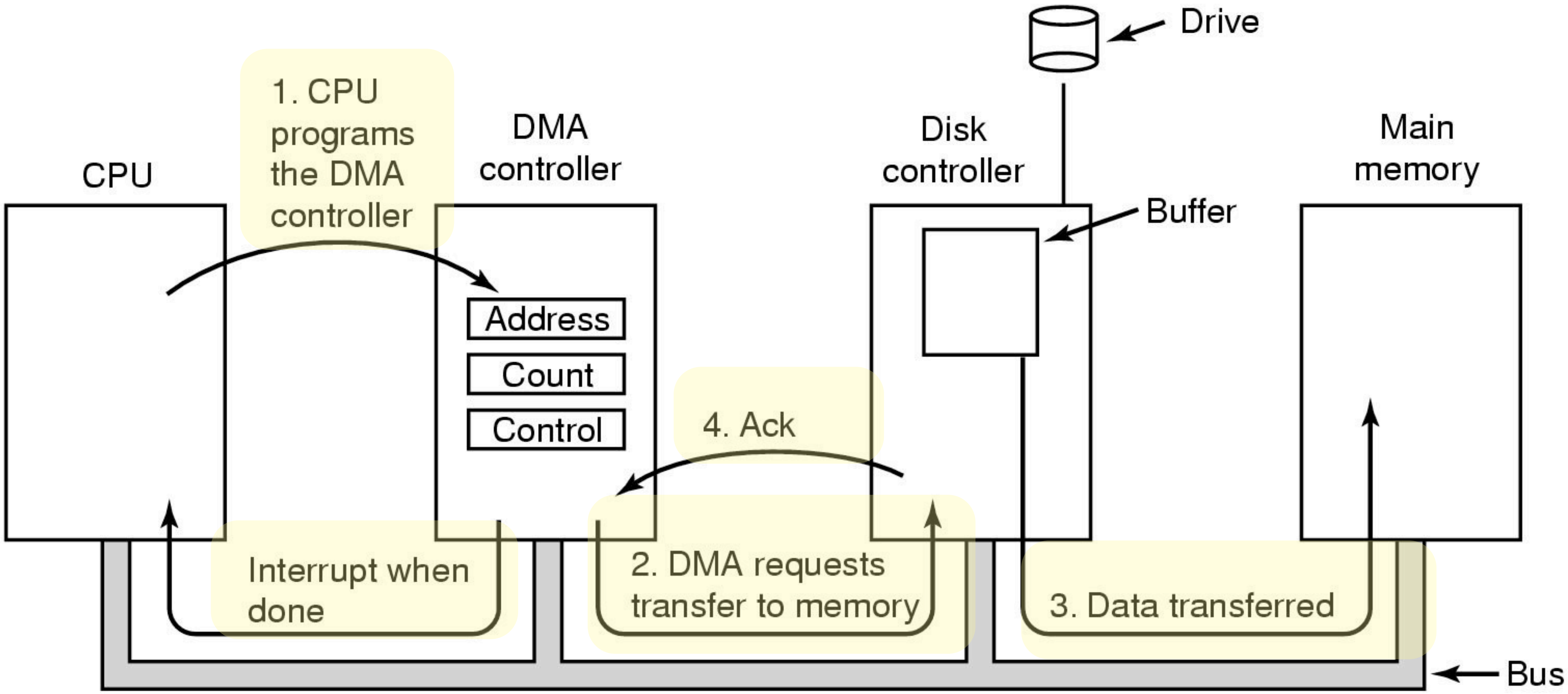
Interrupt-Driven I/O

CPU hardware has a interrupt report line that the CPU senses after executing every instruction.

- **device raises an interrupt**
- **CPU catches the interrupt and saves the state (e.g., Instruction pointer)**
- **CPU dispatches the interrupt handler**
- **interrupt handler determines cause, services the device and clears the interrupt**


Besides Polled I/O and Interrupt driven I/O – are there any other way to perform I/O?
Sure!
We can use a special purpose processor called DMA controller.
WWW.THEMATRIX.COM

Direct Memory Access (DMA)



With DMA, the CPU would initiate the transfer, do other operations while the transfer is in progress, and receive an interrupt from the DMA controller once the operation has been done.

DMA controller feeds the characters from disk one at the time, without CPU being bothered. DMA is actually the programmed IO, only with DMA controller doing the work.

DMA Issues

- *Handshaking* between DMA controller and the device controller
- *Cycle stealing*
  - DMA controller takes away CPU cycles when it uses CPU memory bus, hence blocks the CPU from accessing the memory
- In general DMA controller improves the total system performance

Direct Memory Access

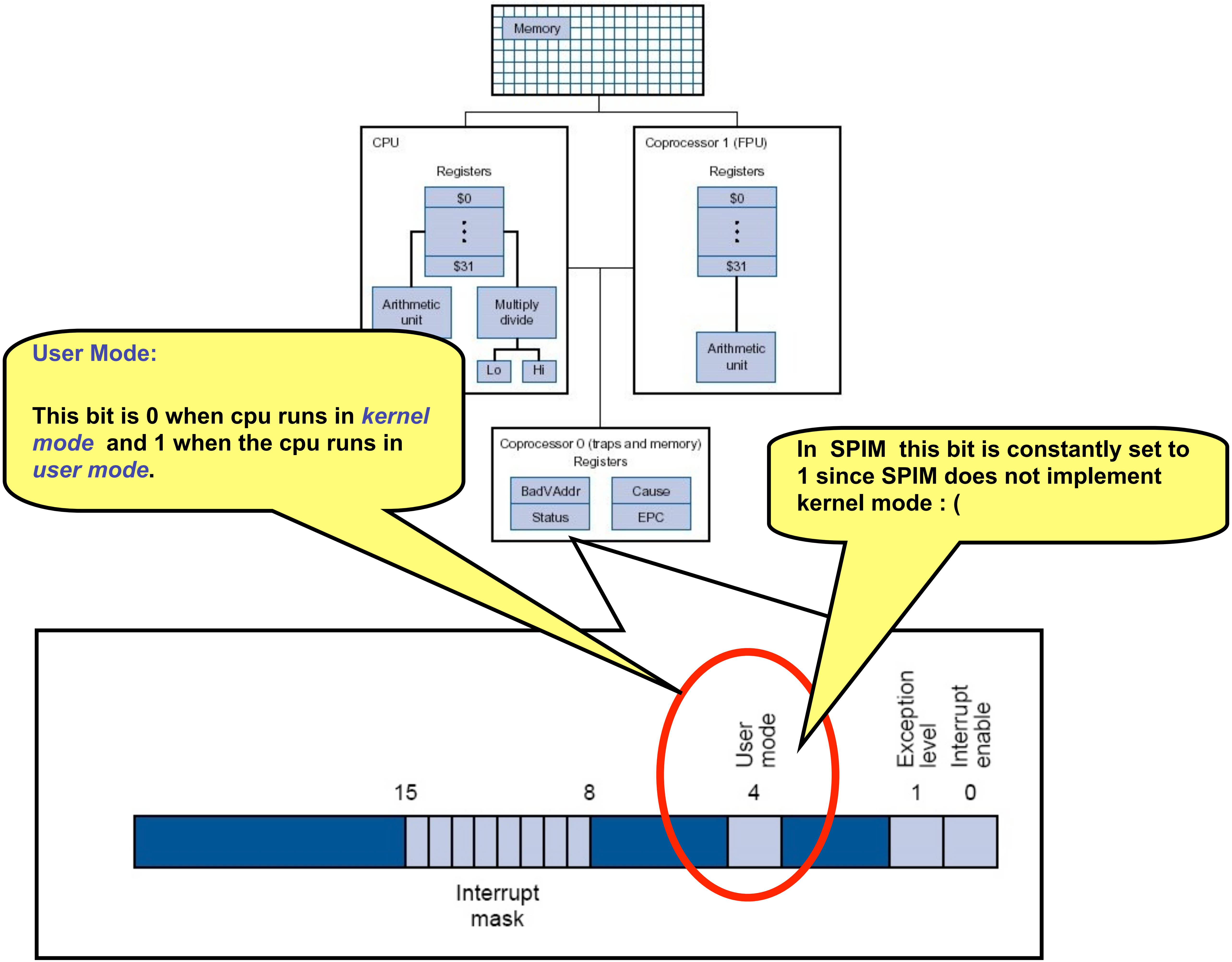
Want to put less load on the CPU

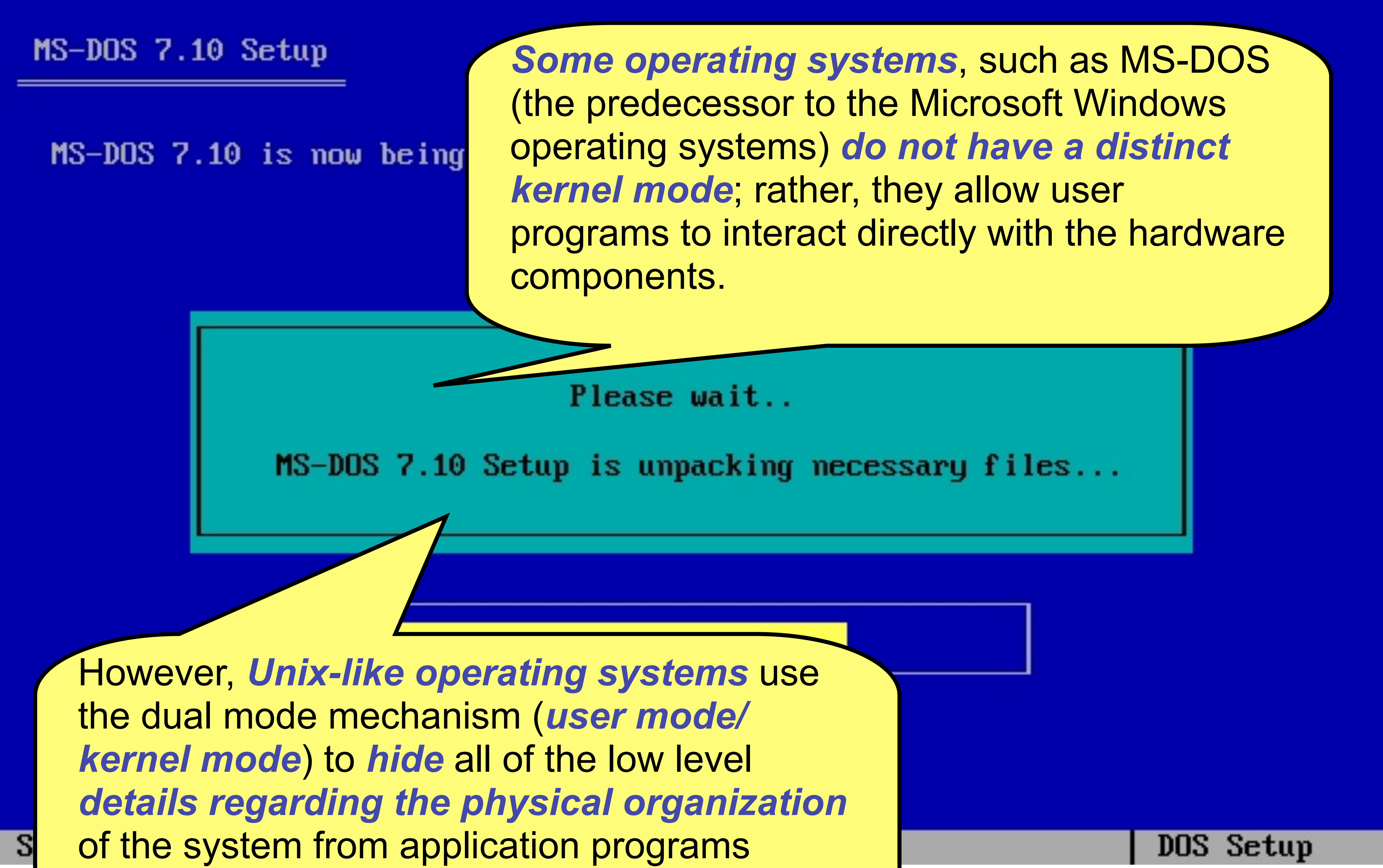
- For high-bandwidth devices (like disks) interrupt-driven I/O would consume a *lot* of processor cycles
- DMA – the I/O controller has the ability to transfer data directly to/from the memory without involving the processor
  - The processor initiates the DMA transfer by supplying the I/O device address, the operation to be performed, the memory address destination/source, the number of bytes to transfer
  - The I/O DMA controller manages the entire transfer (possibly thousand of bytes in length), arbitrating for the bus
  - When the DMA transfer is complete, the I/O controller interrupts the processor to let it know that the transfer is complete

Interrupt only when the transfer is done

There may be multiple DMA devices in one system

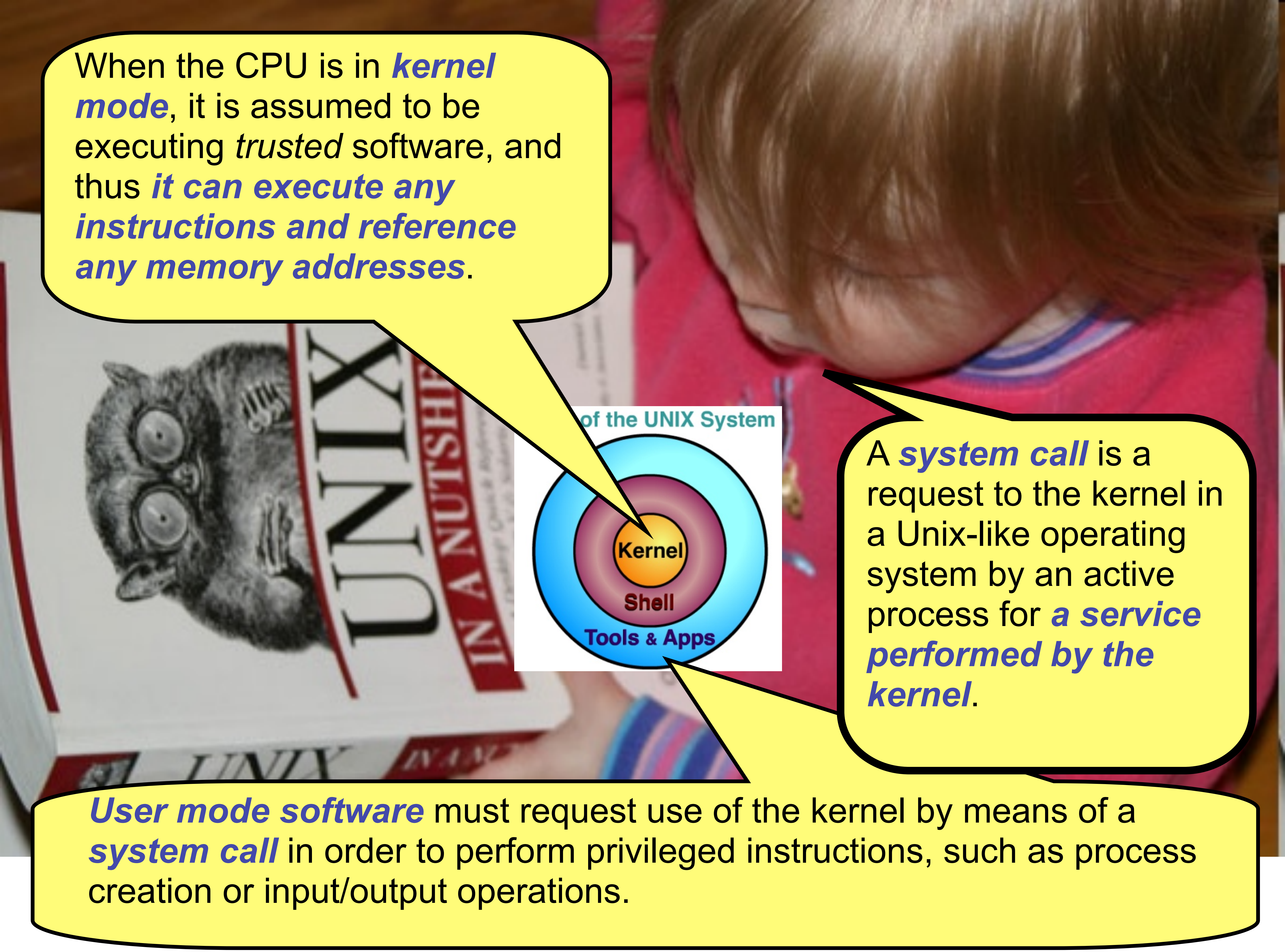
Processor and I/O controllers contend for bus cycles and for memory

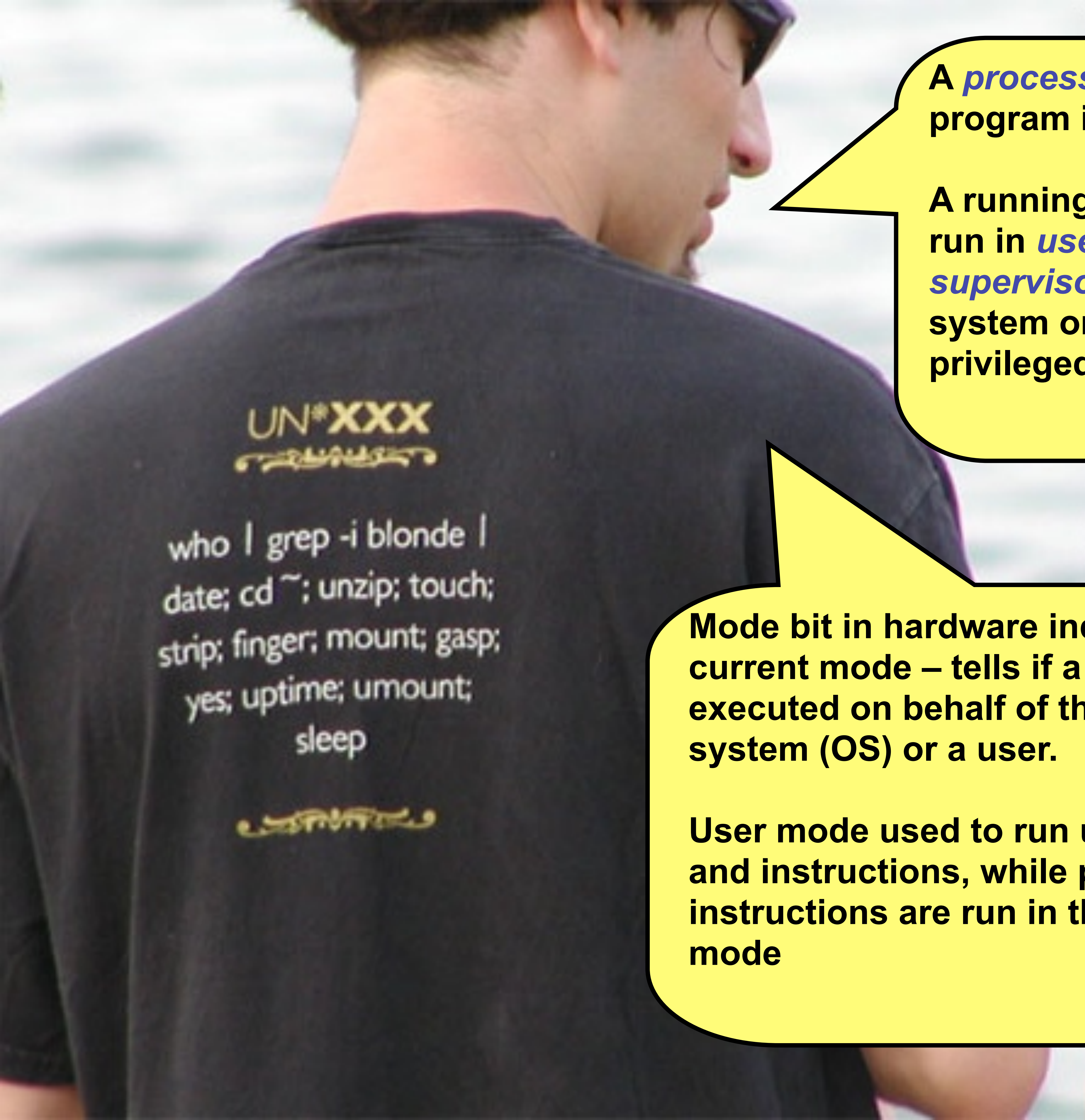

Memory
CPU
Registers
$0
$31
Arithmetic unit
Multiply divide
Lo
Hi
Coprocessor 1 (FPU)
Registers
$0
$31
Arithmetic unit
Coprocessor 0 (traps and memory)
Registers
BadVAddr
Cause
Status
EPC
User Mode:
This bit is 0 when cpu runs in *kernel mode* and 1 when the cpu runs in *user mode*.
In SPIM this bit is constantly set to 1 since SPIM does not implement kernel mode : (
15
8
4
1
0
User mode
Exception level
Interrupt enable
Interrupt mask



Some operating systems, such as MS-DOS (the predecessor to the Microsoft Windows operating systems) ***do not have a distinct kernel mode***; rather, they allow user programs to interact directly with the hardware components.

However, ***Unix-like operating systems*** use the dual mode mechanism (***user mode/ kernel mode***) to ***hide*** all of the low level ***details regarding the physical organization*** of the system from application programs launched by the user as a means of assuring system stability and security.


When the CPU is in ***kernel mode***, it is assumed to be executing *trusted* software, and thus ***it can execute any instructions and reference any memory addresses***.
UNIX
IN A NUTSHELL
of the UNIX System
Kernel
Shell
Tools & Apps
A ***system call*** is a request to the kernel in a Unix-like operating system by an active process for ***a service performed by the kernel***.
User mode software must request use of the kernel by means of a ***system call*** in order to perform privileged instructions, such as process creation or input/output operations.

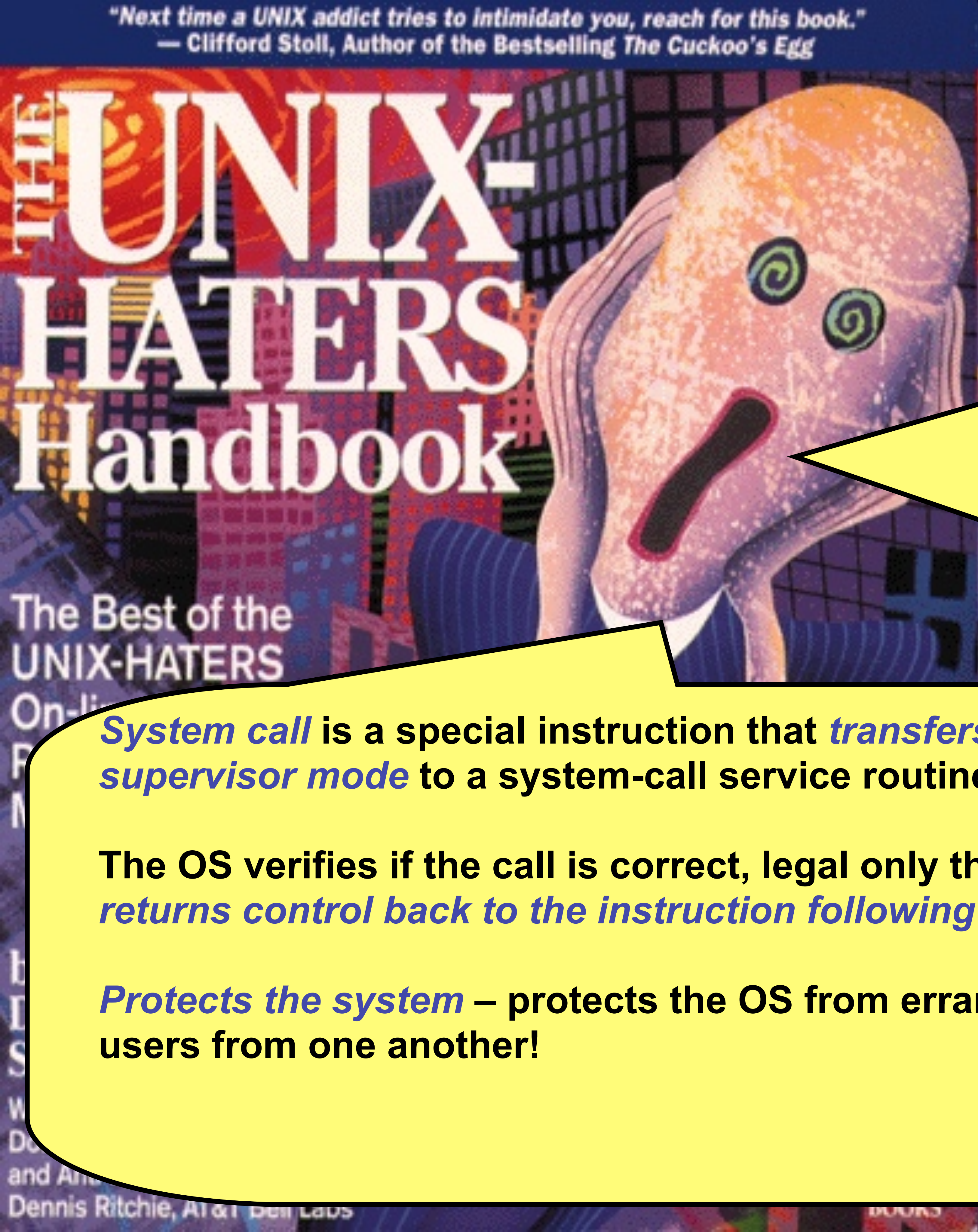


A *process* is an instance of a program in execution

A running process can be run in *user mode* or *supervisor* (also known as system or monitor or privileged) *mode*.

Mode bit in hardware indicates the current mode – tells if a task is being executed on behalf of the operating system (OS) or a user.

User mode used to run user programs and instructions, while privileged instructions are run in the supervisor mode



(Special) instructions that can cause harm are called *privileged instructions* e.g. open, create, delete, share files, using input/output devices.

Should be executed only by the OS in supervisor mode

System call is a special instruction that *transfers control from user mode to supervisor mode* to a system-call service routine that is part of the OS.

The OS verifies if the call is correct, legal only then executes the call and *returns control back to the instruction following the system call*.

Protects the system – protects the OS from errant/malicious users and errant users from one another!


You have now completed part five
of your MIPS assembly training.
A BEGINNinG HAS an END