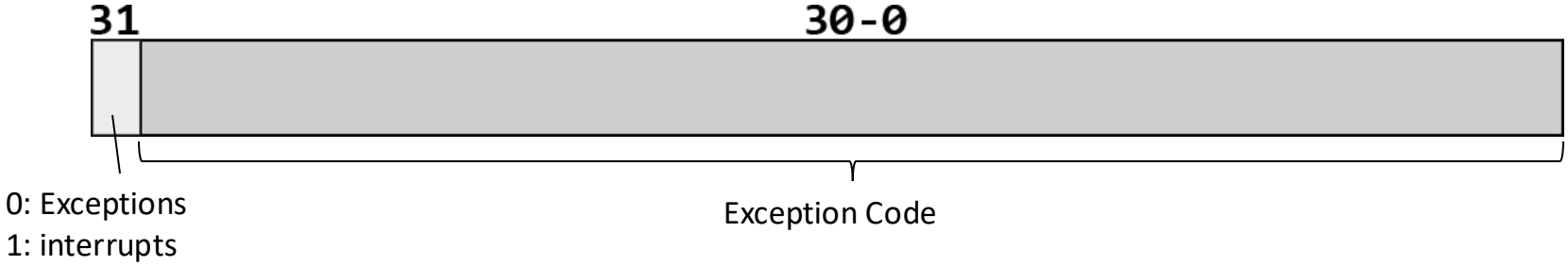
Topic V18

Exceptions in RISC-V

Reading: (Section 4.10)

Cause Register in RISC-V

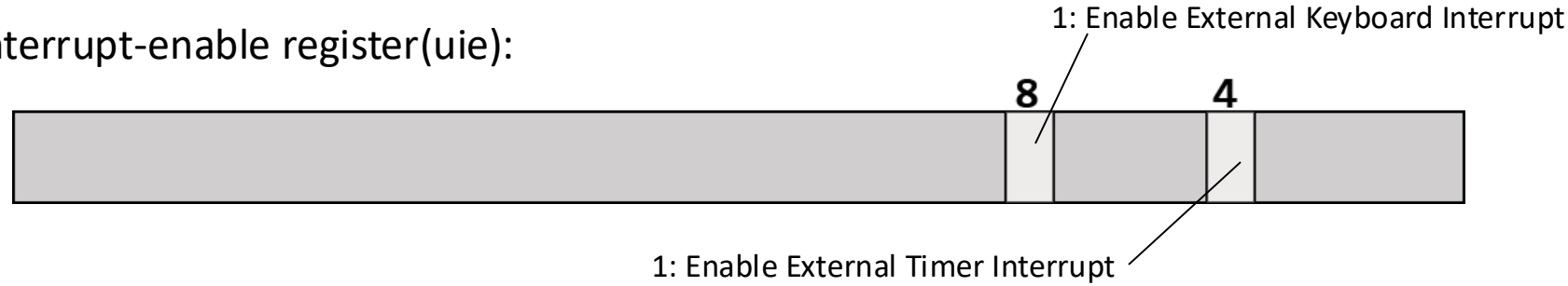
In RISC-V the *User Cause (ucause) Register* encodes the reason for an exception that was raised



Control and Status Registers in RISC-V

Status Register (ustatus):

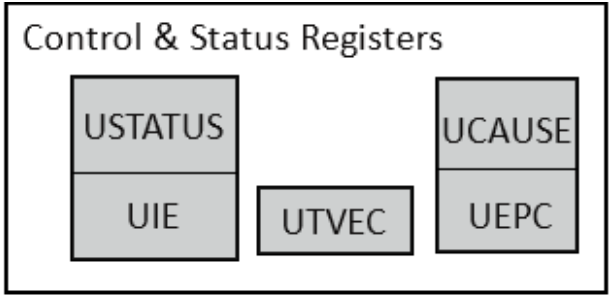
Interrupt-enable register(uie):



The CSR instructions

CSR = control and status register

RISC-V defines 4096 CSRs.
Each CSR instruction has 12 bits to specify a CSR



Atomic Read-Modify-Write Operations with CSRs

Atomic Read/Write CSR:

Specifies the CSR where to perform operation

CSR0 is the ustatus register

```
csrrw t0,  0,  t1
```

```
# t0 ← CSR0 zero extended
# CSR0 ← t1
```

For 64-bit RISCV

t0	0x00000035
t1	0x00000042

CSR0	0x01020010

Atomic Read/Write CSR:

Specifies CSR where to perform operation

CSR0 is the ustatus register

```
csrrw zero,  0,  t1        # zero ← CSR0 zero extended
                           # CSR0 ← t1
```

zero	0x00000000
CSR0	0x01020010
t1	0x00000042

If destination register is register zero, only CSR0 value changes.

Atomic Read/Write Immediate CSR:

```
csrrwi  t0,  0,  0x01 # t0 <- CSR zero extended
                      # CSR <- zero-extended immed.
```

t0	0x00000042
CSR0	0x00000001

Atomic Read and Clear Bits in CSR:

```
csrrc  t0,  0,  t1          # t0 ← CSR zero extended
                            # CSR ← CSR & /t1
```

t0	0x00000042
t1	0xFFFF0000

CSR0	0x00000010

A 1 bit in t1 cause the corresponding bit in CSR0 to be cleared.

Atomic Read and Clear Bits immediate in CSR:

High bits in immed. cause the corresponding bit in CSR0 to be cleared.

```
csrrci   t0,  0,  0x01 # t0 <- CSR zero extended
                       # CSR <- CSR & /immed.
```

t0	0x00000042

CSR0	0x01020010

If immediate is zero, simply copy CSR value into destination register.

Atomic Read and Set Bits in CSR

```
csrrs t0,  0,  t1    # t0 <- CSR zero-extend
                     # CSR <- CSR OR t1
```

t0	0x00000042
t1	0xFFFF0000

CSR0	0xFFFF0011

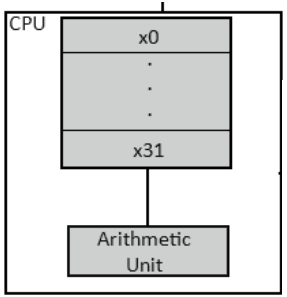
High bits in `t1` cause the corresponding bit in `CSR0` to be set.

Atomic Read and Set Bits Immediate in CSR

```
csrrsi   t0,  0,  0x04   # t0 ← CSR zero-extend
                         # CSR ← CSR OR immed
```

t0	0x00000042
CSR0	0x01020015

RISC-V Coprocessors



Control & Status register available in RARS

Register name	Register number	Usage

Control & Status register available in RARS

Register name	Register number	Usage
ustatus	0x00	interrupt mask and enable bits
uie	0x04	Enable user interrupts such as Timer and Keyboard (external)
utvec	0x05	The base address of the trap handler is stored in this register
uscratch	0x40	Temporary register to be used freely in the user trap handler
uepc	0x41	address of instruction that caused exception
ucause	0x42	exception type
utval	0x43	memory address of an offending memory reference
uip	0x44	User Interrupt pending

Privilege Levels

Privilege Levels

more access
to machine

Level	Encoding	Name	Abbreviation
0	00	User/Application	U

All instructions
are allowed

Each level has its own set of CSR's, for example:

- Mstatus
- Sstatus
- Ustatus

https://riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf

Protection by Privilege Levels

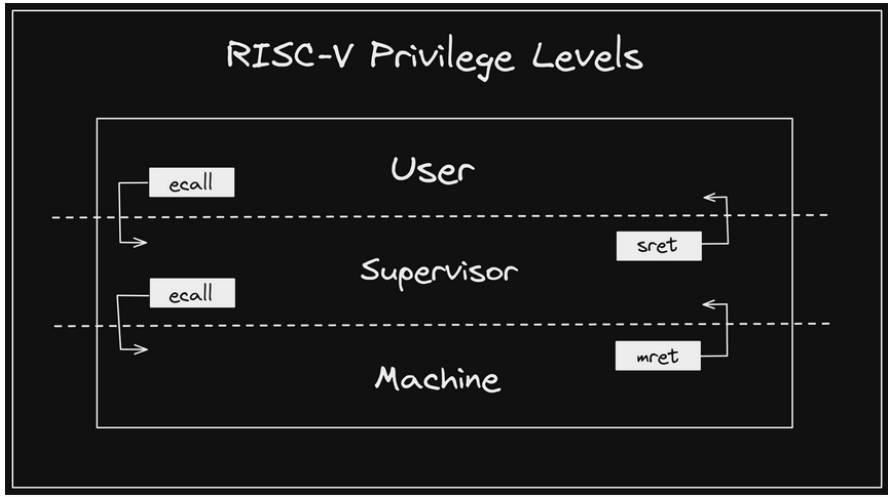
- A program running in U cannot access any M or S CSR registers.
  - If a U program could change the base address of the trap handler for the machine level (mtvec), it could change an exception handle to do nasty things.
  - The operating system ensures that no user process accesses illegal memory.

Requesting Service

`ecall`:

Requests service from a more privileged mode.

Requests may percolate to higher levels.



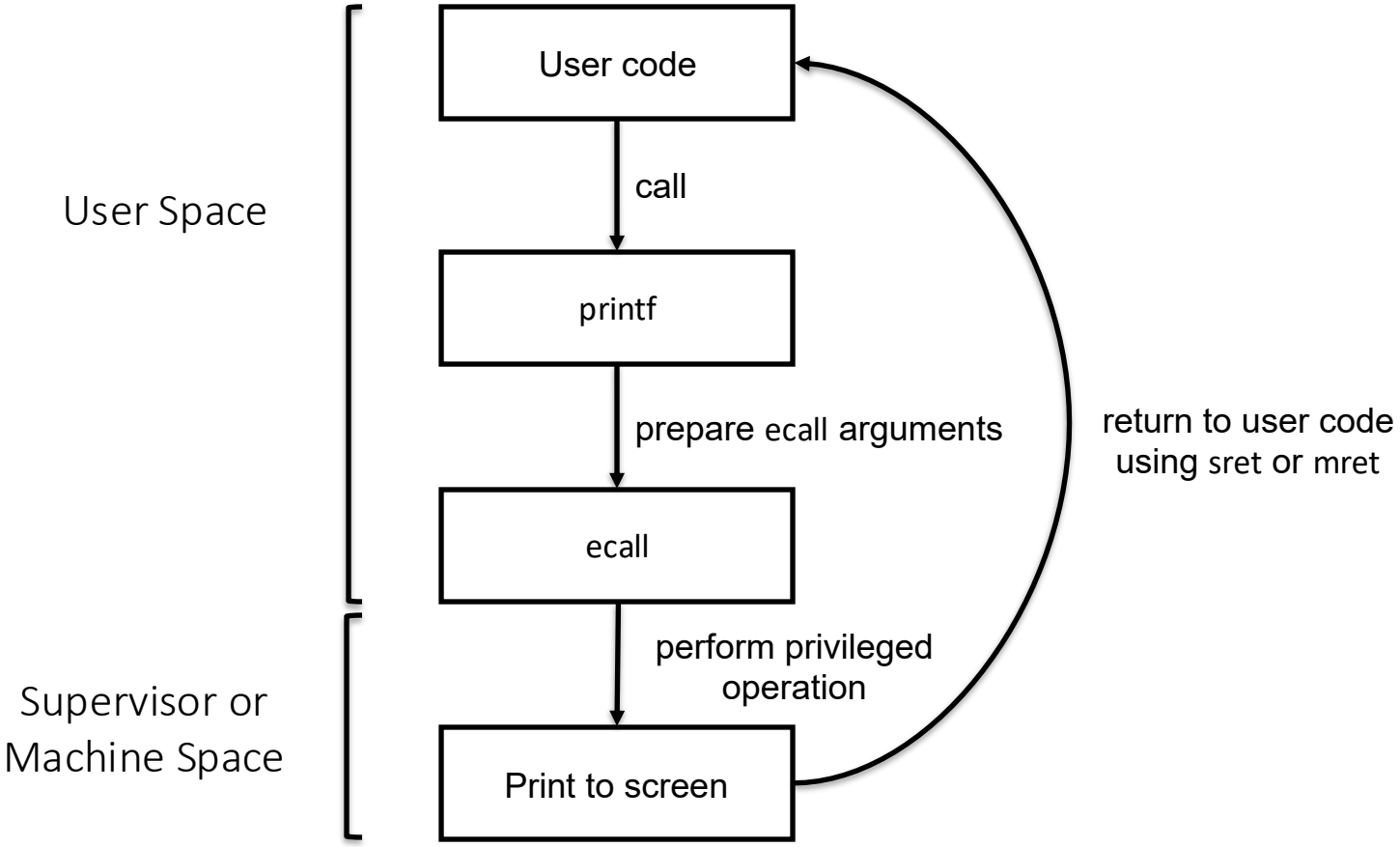
https://danielmangum.com/posts/risc-v-bytes-privilege-levels/

Context Switching in the Linux Kernel

- Synchronous (syscall == ecall)
  - Before handling the issue, the OS must:
    - save all the s registers
    - save the ra register
- Asynchronous (exceptions)
  - Before handling the issue, the OS must:
    - save <u>all</u> the registers

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/riscv/kernel/entry.S?h=v5.7-rc5

`printf` execution



Trap Levels

- Vertical Trap
  - Increase privilege level
- Horizontal Trap
  - Privilege level does not change
  - Can be implemented as vertical traps that return control to a horizontal trap handler in the less-privileged mode.

Exception Codes

Interrupt	Exception Code	Description

Exception Codes

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	*Reserved for future standard use*
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	*Reserved for future standard use*
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	*Reserved for future standard use*
1	11	Machine external interrupt
1	12–15	*Reserved for future standard use*
1	$\geq$16	*Reserved for platform use*

What happens when an exception occurs?

Processor jumps to code starting at base address of the trap handler stored in the utvec register (5).

- This is the first instruction of the exception handler.

Exception handler:

- examines the cause of the exception
- jumps to the OS handler for that exception

OS either:

- terminates the process that caused exception;
- or takes care of exception condition and restarts the process;

Writing an exception Handler

Can the handler save registers to the stack?

No!

Example where using the SP could go wrong

In both cases the sp is corrupted and cannot be used by the handler

```
li      sp, 0
lw      t0, 0(sp)
```

This instruction causes
a segmentation fault

```
li      sp, 0x80008001
lw      t0, 0(sp)
```

This instruction causes an
Unaligned access exception

How to save registers without using the user stack?

Create a stack in the kernel area!

A re-entrant exception handler

J. Nelson Amaral

University of Alberta

Create a kernel stack in the kernel memory region

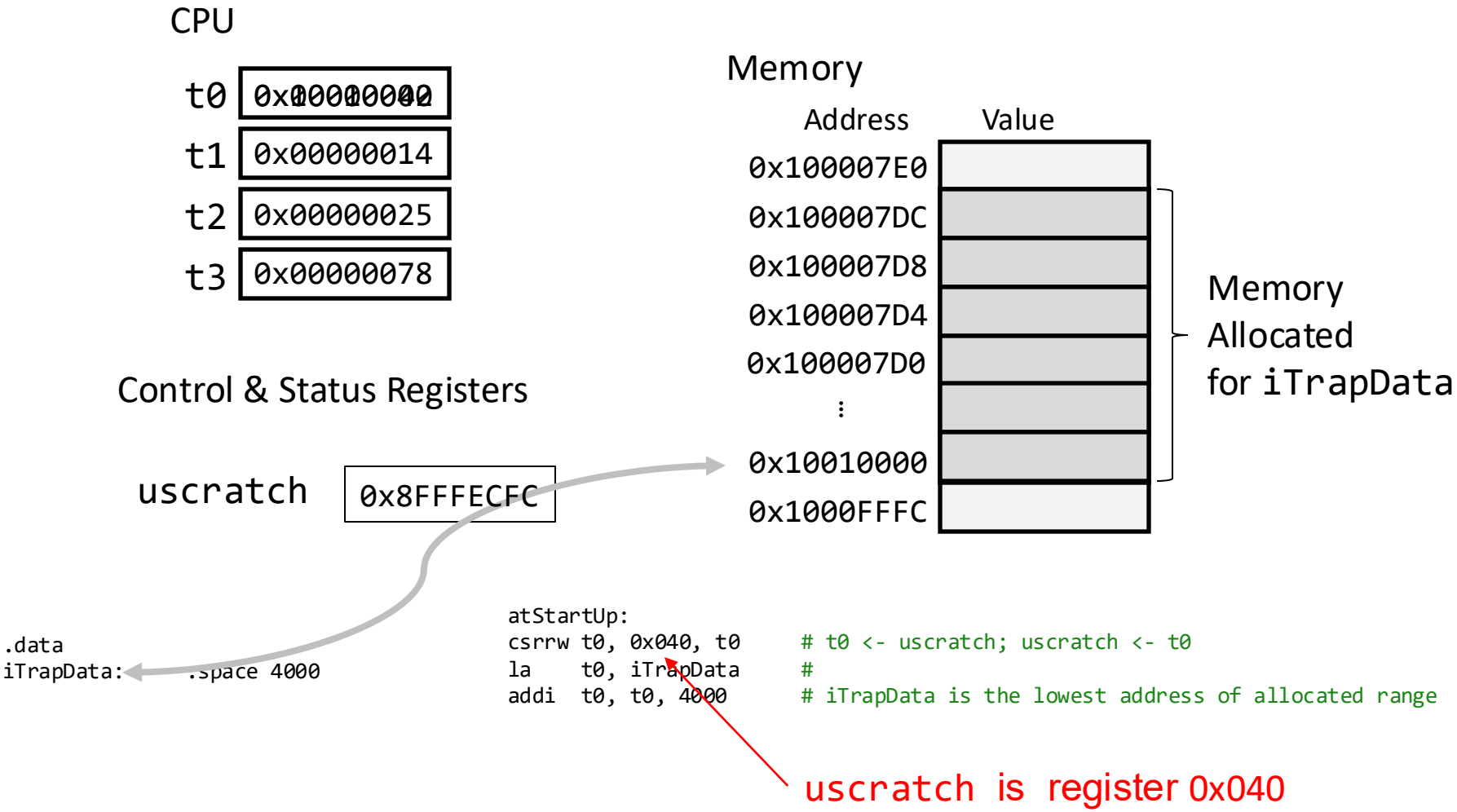
```
.data
iTrapData:      .space 4000
```

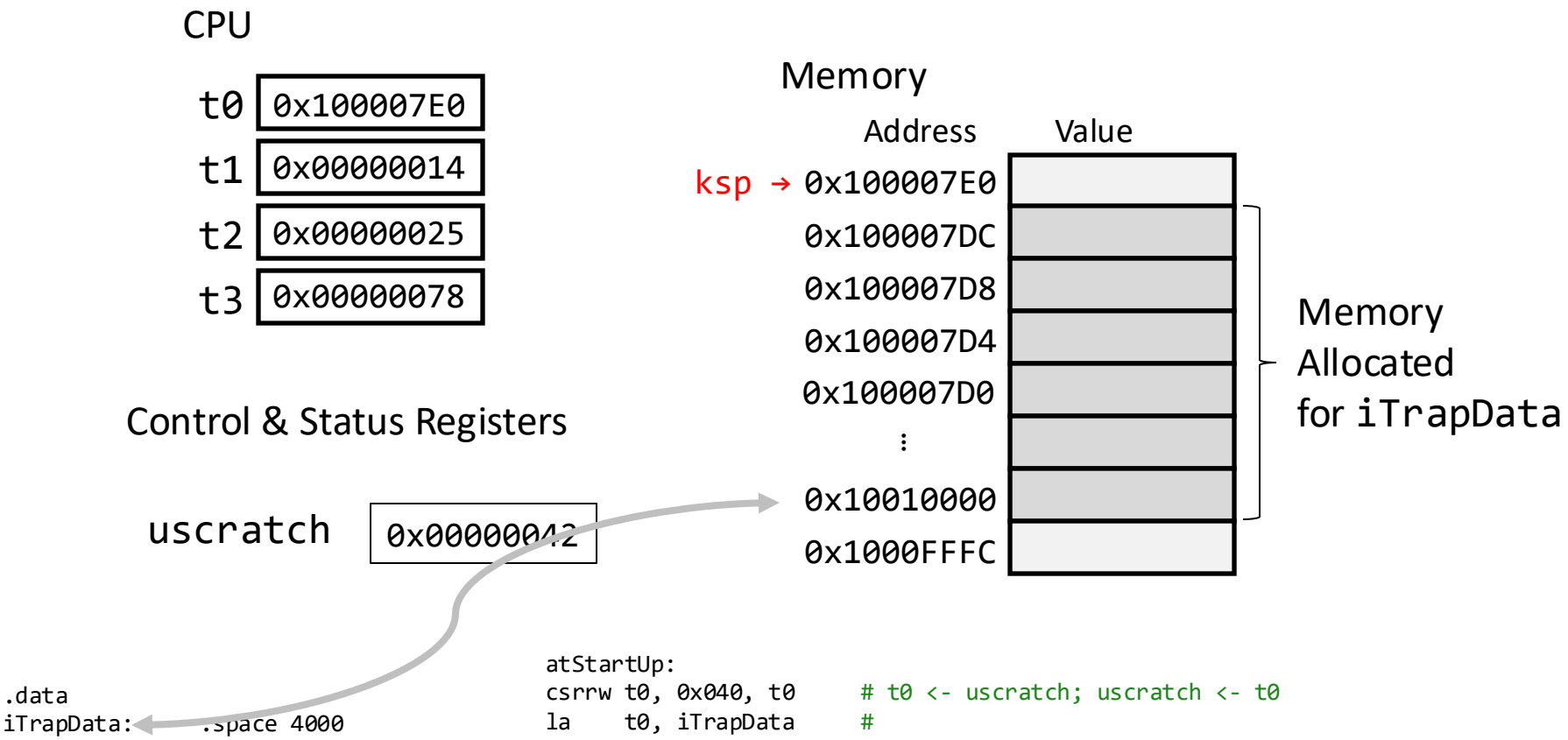
Let `ksp` be the address used for the kernel stack pointer.

Let `uscratch` register be used to store `ksp` at all times.

Exceptions can only be enabled when the value in `uscratch` is consistent with the state of the kernel stack in memory.

Initializing the `uscratch` register at startup time.


CPU
t0
0x10010040
t1
0x00000014
t2
0x00000025
t3
0x00000078
Memory
Address
Value
0x100007E0
0x100007DC
0x100007D8
0x100007D4
0x100007D0
0x10010000
0x1000FFFC
Memory Allocated for iTrapData
Control & Status Registers
uscratch
0x8FFFECFC
.data
iTrapData: .space 4000
atStartUp:
csrrw t0, 0x040, t0 # t0 <- uscratch; uscratch <- t0
la t0, iTrapData #
addi t0, t0, 4000 # iTrapData is the lowest address of allocated range
uscratch is register 0x040


CPU
t0 0x100007E0
t1 0x00000014
t2 0x00000025
t3 0x00000078
Control & Status Registers
uscratch 0x00000042
Memory
Address
Value
ksp → 0x100007E0
0x100007DC
0x100007D8
0x100007D4
0x100007D0
⋮
0x10010000
0x1000FFFC
Memory Allocated for iTrapData
.data
iTrapData: .space 4000
atStartUp:
csrrw t0, 0x040, t0 # t0 <- uscratch; uscratch <- t0
la t0, iTrapData #
addi t0, t0, 4000 #
csrrw t0, 0x040, t0 # t0 <- t0 original value
uscratch <- initial ksp
<other initialization tasks>

CPU

t0	0x00000042
t1	0x00000014
t2	0x00000025
t3	0x00000078

Control & Status Registers

uscratch	0x100007E0

Memory

	Address	Value
ksp →	0x100007E0	
	0x100007DC	
	0x100007D8	
	0x100007D4	
	0x100007D0	
	⋮	
	0x10010000	
	0x1000FFFC	

Memory Allocated for `iTrapData`

```
.data
iTrapData:      .space 4000
```

Now that uscratch is initialized to ksp, the handler can be invoked by interrupts.

Updating the `uscratch` upon entering the exception handler
Before enabling exceptions.

CPU

t0	0x00000042
t1	0x100007D4
t2	0x00000025
t3	0x00000078

Control & Status Registers

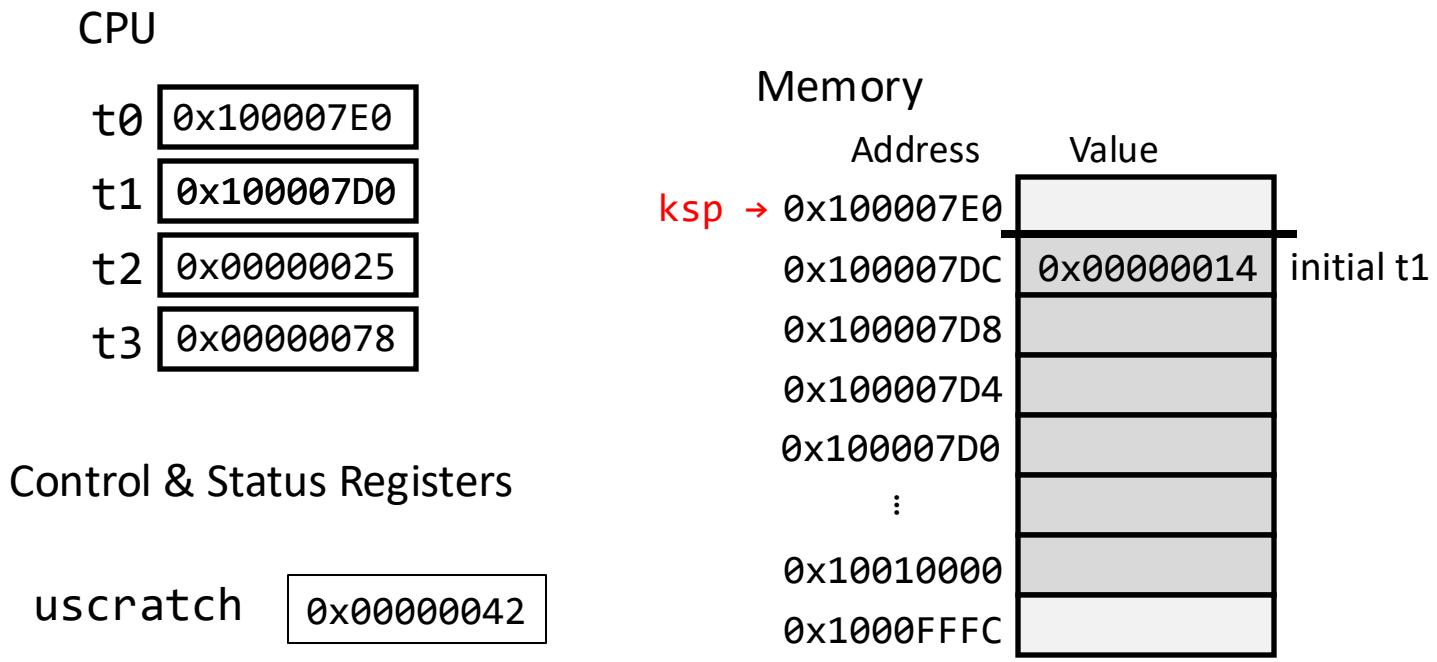
uscratch	0x100007E0

Memory

	Address	Value
ksp →	0x100007E0	
	0x100007DC	
	0x100007D8	
	0x100007D4	
	0x100007D0	
	⋮	
	0x10010000	
	0x1000FFFC	

```
.data
iTrapData:      .space 4000
```

```
handler:
csrrw t0, 0x040, t0      # t0 <- ksp; uscratch <- t0
sw    t1, -4(t0)         # Mem[ksp-4] <- t1
addi  t1, t0, -16        # t1 <- updated ksp
```




```
.data
iTrapData:      .space 4000
```

```
handler:
csrrw t0, 0x040, t0     # t0 <- ksp; uscratch <- t0
sw    t1, -4(t0)        # Mem[ksp-4] <- t1
addi  t1, t0, -16       # t1 <- updated ksp
csrrw t0, 0x040, t1     # t0 <- t0 original value;
                        # uscratch <- updated ksp
<enable exceptions because uscratch contains the correct ksp>
```

Saving registers that the handler will use.

CPU

t0	0x00000042
t1	0x100007D0
t2	0x00000025
t3	0x00000078

Control & Status Registers

uscratch	0x100007D0

Memory

	Address	Value	
	0x100007E0		
	0x100007DC	0x00000014	initial t1
	0x100007D8		
	0x100007D4		
ksp →	0x100007D0		
	⋮		
	0x10010000		
	0x1000FFFC		

```
.kdata
iTrapData:      .space 4000
```

```
sw      t0, 8(t1)           # save t0 into kstack
sw      t2, 4(t1)           # save t2 into kstack
sw      t3, 0(t1)           # save t3 into kstack
<handler can use t0, t2, t3>
```

Restoring registers before exiting the handler.

CPU

t0	0x00FFFFCC
t1	0x00000002
t2	0xDDDDDDDA
t3	0x12345678

Control & Status Registers

uscratch	0x100007D0

Memory

	Address	Value	
	0x100007E0		
	0x100007DC	0x00000014	initial t1
	0x100007D8	0x00000042	initial t0
	0x100007D4	0x00000025	initial t2
ksp →	0x100007D0	0x00000078	initial t3
	⋮		
	0x10010000		
	0x1000FFFC		

```
.kdata
iTrapData:      .space 4000
```

```
...
lw      t0, 8(t1)       # restore t3 from kstack
lw      t2, 4(t1)       # restore t2 from kstack
lw      t3, 0(t1)       # restore t3 from kstack
lw      t1, 12(t1)      # restore t1 from kstack
```

Disable exceptions before moving the `ksp` by updating the value in `sscratch`.

CPU

t0	0x10000740
t1	0x00000014
t2	0x00000025
t3	0x00000078

Control & Status Registers

uscratch	0x100007D0

Memory

	Address	Value
	0x100007E0	
	0x100007DC	0x00000014
	0x100007D8	0x00000042
	0x100007D4	0x00000025
ksp →	0x100007D0	0x00000078
	⋮	
	0x10010000	
	0x1000FFFC	

```
.data
iTrapData:      .space 4000
```

```
<disable exceptions while updating ksp in sscratch>
csrrw   t0, 0x040, t0   # t0 <- ksp; uscratch <- t0
addi    t0, t0, 16      # t0 <- ksp back to original position
```

CPU

t0	0x100007E0
t1	0x00000014
t2	0x00000025
t3	0x00000078

Control & Status Registers

uscratch	0x00000042

Memory

	Address	Value
	0x100007E0	
	0x100007DC	0x00000014
	0x100007D8	0x00000042
	0x100007D4	0x00000025
ksp →	0x100007D0	0x00000078
	⋮	
	0x10010000	
	0x1000FFFC	

```
.data
iTrapData:      .space 4000
```

```
<disable exceptions while updating ksp in sscratch>
csrrw   t0, 0x040, t0   # t0 <- ksp; uscratch <- t0
addi    t0, t0, 16      # t0 <- ksp back to original position
csrrw   t0, 0x040, t0   # t0 <- original t0; csrrw <- ksp
<enable exceptions and return>
```

```
.text
atStartUp:
csrrw t0, 0x040, t0     # t0 <- uscratch; uscratch <- t0
la    t0, kstack        #
addi  t0,t0, 4000       # kstack is the lowest address of allocated range
csrrw t0, 0x040, t0     # t0 <- t0 original value
                        # uscratch <- initial ksp
<other initialization tasks>



handler:
csrrw t0, 0x040, t0     # t0 <- ksp; uscratch <- t0
sw    t1, -4(t0)        # Mem[ksp-4] <- t1
addi  t1, t0, -16       # t1 <- updated ksp
csrrw t0, 0x040, t1     # t0 <- t0 original value;
                        # uscratch <- updated ksp
<enable exceptions because uscratch contains the correct ksp>
sw     t0, 8(t1)          # save t0 into kstack
sw     t2, 4(t1)          # save t2 into kstack
sw     t3, 0(t1)          # save t3 into kstack
<handler can use t0, t2, t3>
...
lw      t0, 8(t1)       # restore t3 from kstack
lw      t2, 4(t1)       # restore t2 from kstack
lw      t3, 0(t1)       # restore t3 from kstack
lw      t1, 12(t1)      # restore t1 from kstack
<disable exceptions while updating ksp in sscratch>
csrrw   t0, 0x040, t0   # t0 <- ksp; uscratch <- t0
addi    t0, t0, 16      # t0 <- ksp back to original position
csrrw   t0, 0x040, t0   # t0 <- original t0; uscratch <- ksp
<enable exceptions and return>
```

User-visible CSRs

Number	Name	Description
0x000	ustatus	User status register.
0x004	uie	User interrupt-enable register.
0x005	utvec	User trap handler base address.
0x040	uscratch	Scratch register for user trap handlers.
0x041	uepc	User exception program counter.
0x042	ucause	User trap cause.
0x043	utval	User bad address or instruction.
0x044	uip	User interrupt pending.