

Announcements

Upcoming schedule ...

Start thinking about final project; form teams of 2 people (1-3 ok)

Final 2 labs will be devoted to working on your project (work on proposal as team; check-in)

Late days apply through Assign7, not final project

Single and double buffering review

font_get_char demonstration

Single and Double Buffering Review

font_get_char
demonstration

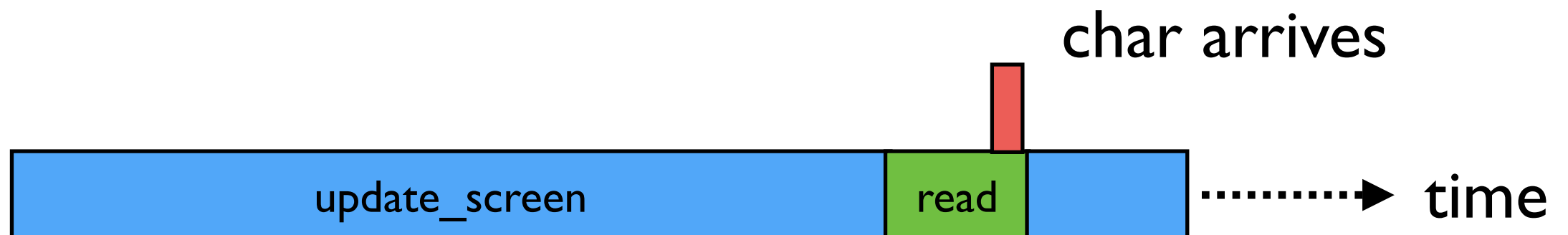
Interrupts



Blocking I/O

```
while (1) {  
    read_char_from_keyboard();  
    update_screen();  
}
```

Read fn loops until char is received - blocking



Blocking I/O

```
while (1) {  
    read_char_from_keyboard();  
    update_screen();  
}
```

How long does it take to send a scan code?

- 11kHz, 11 bits/scan code

How long does it take to update the screen?

What could go wrong?

code/glkeyboard

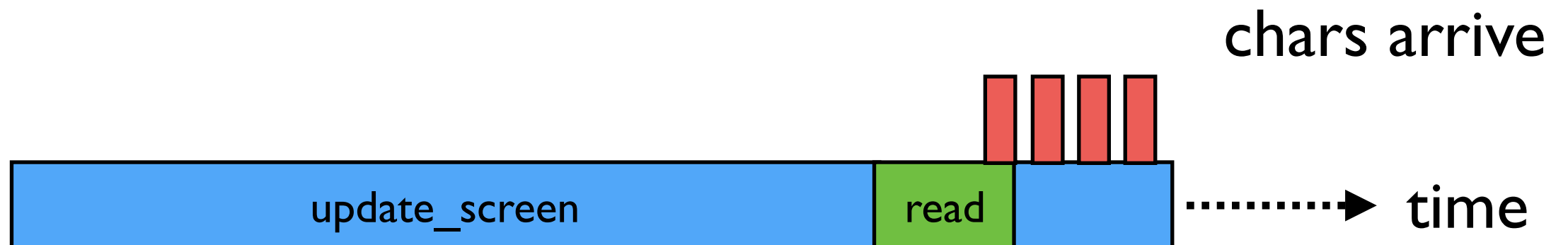
Blocking I/O

```
while (1) {  
    read_char_from_keyboard();  
    update_screen();  
}
```



Blocking I/O

```
while (1) {  
    read_char_from_keyboard();  
    update_screen();  
}
```



The Problem

Need long-running computations (graphics, computations, applications, etc.).

Need to respond to external events quickly.

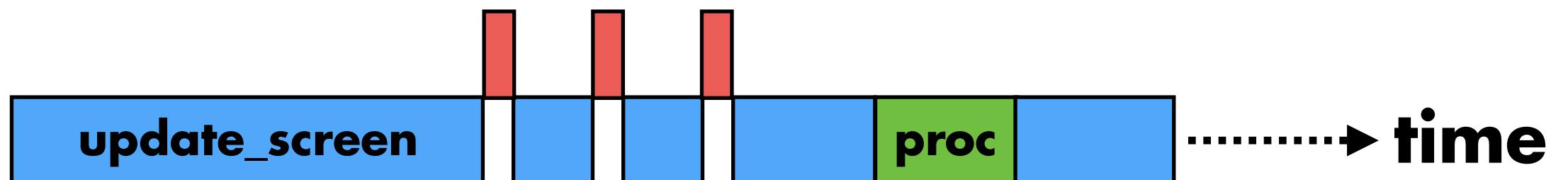
How could we change this code?

```
while (1) {  
    read_char_from_keyboard( );  
    update_screen( );  
}
```

Interrupts

```
when a scan code arrives {  
    add_scan_code_to_buffer();  
}
```

```
while (1) {  
    // Doesn't block  
    while (read_chars_from_buffer()) {}  
    update_screen();  
}
```



Interrupts to the Rescue

Cause processor to pause what it's doing and immediately execute interrupt code, returning to original code when done.

- **External events (reset, timer, GPIO)**
- **Internal events (bad memory access, software trigger)**

Critical for responsive systems.

Using interrupts exercises everything you've learned so far.

- **Architecture, assembly, linking, memory, C, peripherals**

They'll complete your interactive graphics console.

code/blink

blink.c

Timer causes an interrupt once per second

while() loop in main() is interrupted

Interrupt handler increments counter

while() checks for counter change

Toggles ACT_LED and prints counter

Why is counter declared volatile?

How is interrupt_handler called?

Interrupt Vectors

8 Kinds of Interrupts

`_vectors:`

```
    ldr pc, _reset_asm
    ldr pc, _undefined_instruction_asm
    ldr pc, _software_interrupt_asm
    ldr pc, _prefetch_abort_asm
    ldr pc, _data_abort_asm
    ldr pc, _reset_asm
    ldr pc, _interrupt_asm
    ldr pc, _fast_asm
```

<code>_reset_asm:</code>	<code>.word impossible_asm</code>
<code>_undefined_instruction_asm:</code>	<code>.word impossible_asm</code>
<code>_software_interrupt_asm:</code>	<code>.word impossible_asm</code>
<code>_prefetch_abort_asm:</code>	<code>.word impossible_asm</code>
<code>_data_abort_asm:</code>	<code>.word impossible_asm</code>
<code>_interrupt_asm:</code>	<code>.word interrupt_asm</code>
<code>_fast_asm:</code>	<code>.word impossible_asm</code>

`_vectors_end`

Interrupt Vectors

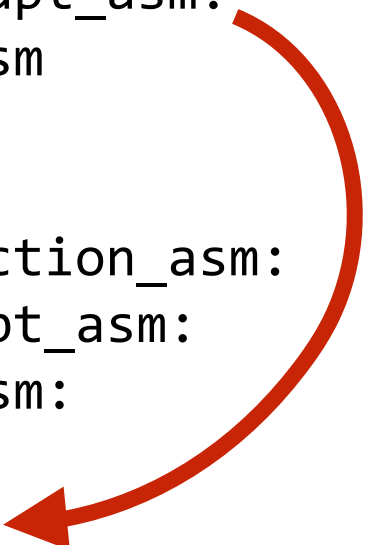
```
_vectors:
    ldr pc, _reset_asm
    ldr pc, _undefined_instruction_asm
    ldr pc, _software_interrupt_asm
    ldr pc, _prefetch_abort_asm
    ldr pc, _data_abort_asm
    ldr pc, _reserved_asm
    ldr pc, _interrupt_asm:
    ldr pc, _fast_asm

_reset_asm:
_undefined_instruction_asm:
_software_interrupt_asm:
_prefetch_abort_asm:
_data_abort_asm:
_interrupt_asm:
_fast_asm:

.word impossible_asm
.word impossible_asm
.word impossible_asm
.word impossible_asm
.word impossible_asm
.word interrupt_asm
.word impossible_asm

_vectors_end
```

branch to interrupt_asm



How is interrupt_asm is called?

What is the difference between interrupt_asm and _interrupt_asm?

Interrupt Vectors

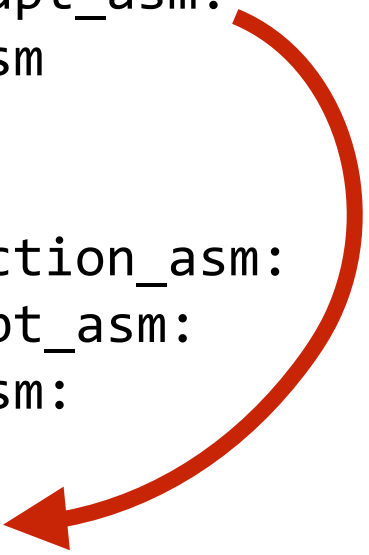
```
_vectors:
    ldr pc, _reset_asm
    ldr pc, _undefined_instruction_asm
    ldr pc, _software_interrupt_asm
    ldr pc, _prefetch_abort_asm
    ldr pc, _data_abort_asm
    ldr pc, _reset_asm
    ldr pc, _interrupt_asm:
    ldr pc, _fast_asm

_reset_asm:
_undefined_instruction_asm:
_software_interrupt_asm:
_prefetch_abort_asm:
_data_abort_asm:
_interrupt_asm:
_fast_asm:

.word impossible_asm
.word impossible_asm
.word impossible_asm
.word impossible_asm
.word impossible_asm
.word interrupt_asm
.word impossible_asm

_vectors_end
```

branch to interrupt_asm



How does the system know what to call when an interrupt occurs?

cstart.c

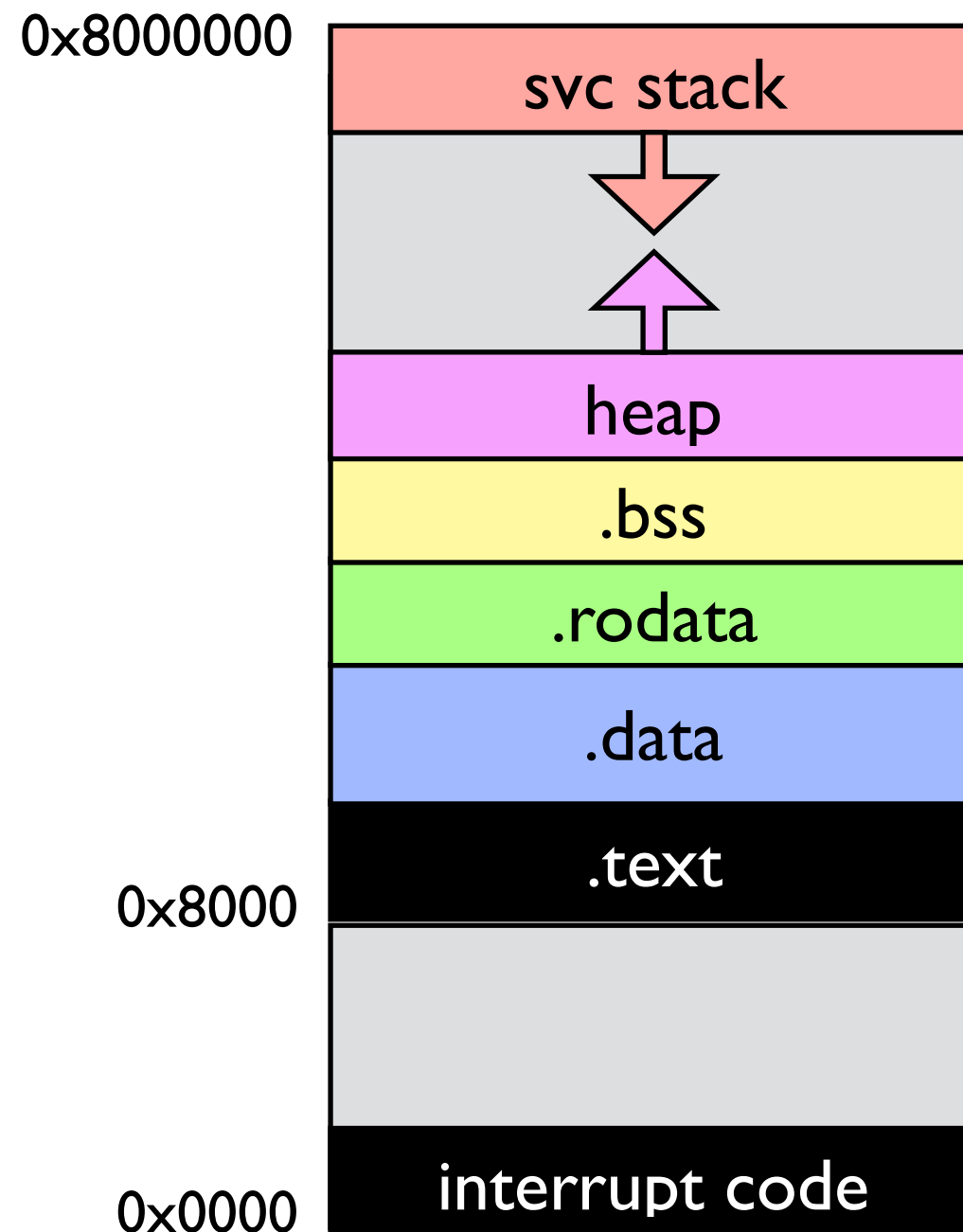
```
static int * const RPI_INTERRUPT_VECTOR_BASE = 0x0;  
  
int* vectorsdst = RPI_INTERRUPT_VECTOR_BASE;  
int* vectors = &_amp;vectors;  
int* vectors_end = &_amp;vectors_end;  
while (vectors < vectors_end) {  
    *vectorsdst++ = *vectors++;  
}
```

Where do we put the interrupt vectors?

Why do we need to copy them?

Where are `_vectors` and `_vectors_end` defined?

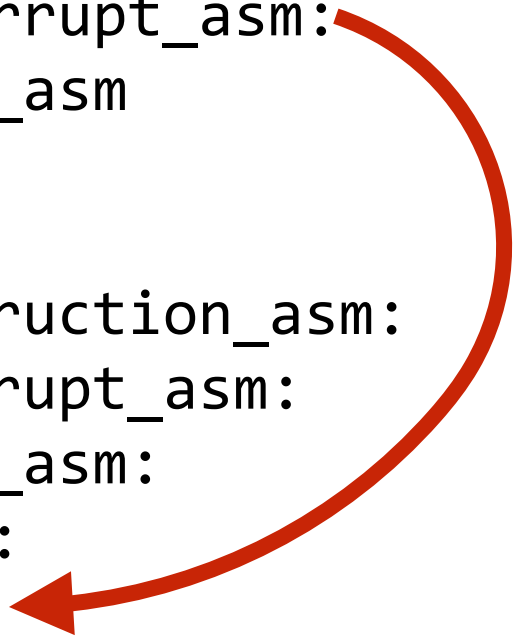
CPU Address Space



```
_vectors:  
    ldr pc, _reset_asm  
    ldr pc, _undefined_instruction_asm  
    ldr pc, _software_interrupt_asm  
    ldr pc, _prefetch_abort_asm  
    ldr pc, _data_abort_asm  
    ldr pc, _reset_asm  
    ldr pc, _interrupt_asm:  
    ldr pc, _fast_asm
```

branch to interrupt_asm

```
_reset_asm: .word impossible_asm  
_undefined_instruction_asm: .word impossible_asm  
_software_interrupt_asm: .word impossible_asm  
_prefetch_abort_asm: .word impossible_asm  
_data_abort_asm: .word impossible_asm  
_interrupt_asm: .word interrupt_asm  
_fast_asm: .word impossible_asm
```



```
_vectors_end
```

Why does this code work if it is copied to address 0?


Position independent code

pc-relative addressing

```
% cd ../vector
% make main.list
% cat main.list
```

...

```
00008040 <_vectors>:
```



```
8040:    ldr pc, [pc, #24]    ; 8060 <_impossible_asm>
8044:    ldr pc, [pc, #20]    ; 8060 <_impossible_asm>
8048:    ldr pc, [pc, #16]    ; 8060 <_impossible_asm>
804c:    ldr pc, [pc, #12]    ; 8060 <_impossible_asm>
8050:    ldr pc, [pc, #8]     ; 8060 <_impossible_asm>
8054:    ldr pc, [pc, #4]     ; 8060 <_impossible_asm>
8058:    ldr pc, [pc, #4]     ; 8064 <_interrupt_asm>
805c:    ldr pc, [pc, #-4]    ; 8060 <_impossible_asm>
```

```
00008060 <_impossible_asm>:
```

```
8060:    .word    0x00008084
```

```
00008064 <_interrupt_asm>:
```

```
8064:    .word    0x00008068
```

```
. . .
```

```
00008084 <impossible_asm>:
```

```
. . .
```

```
00008068 <interrupt_asm>:
```

Interrupt Handler

Interrupt Handler

interrupt_asm:

mov sp, #0x8000

sub lr, lr, #4

push {r0-r12,lr}

mov r0, lr

bl interrupt_handler

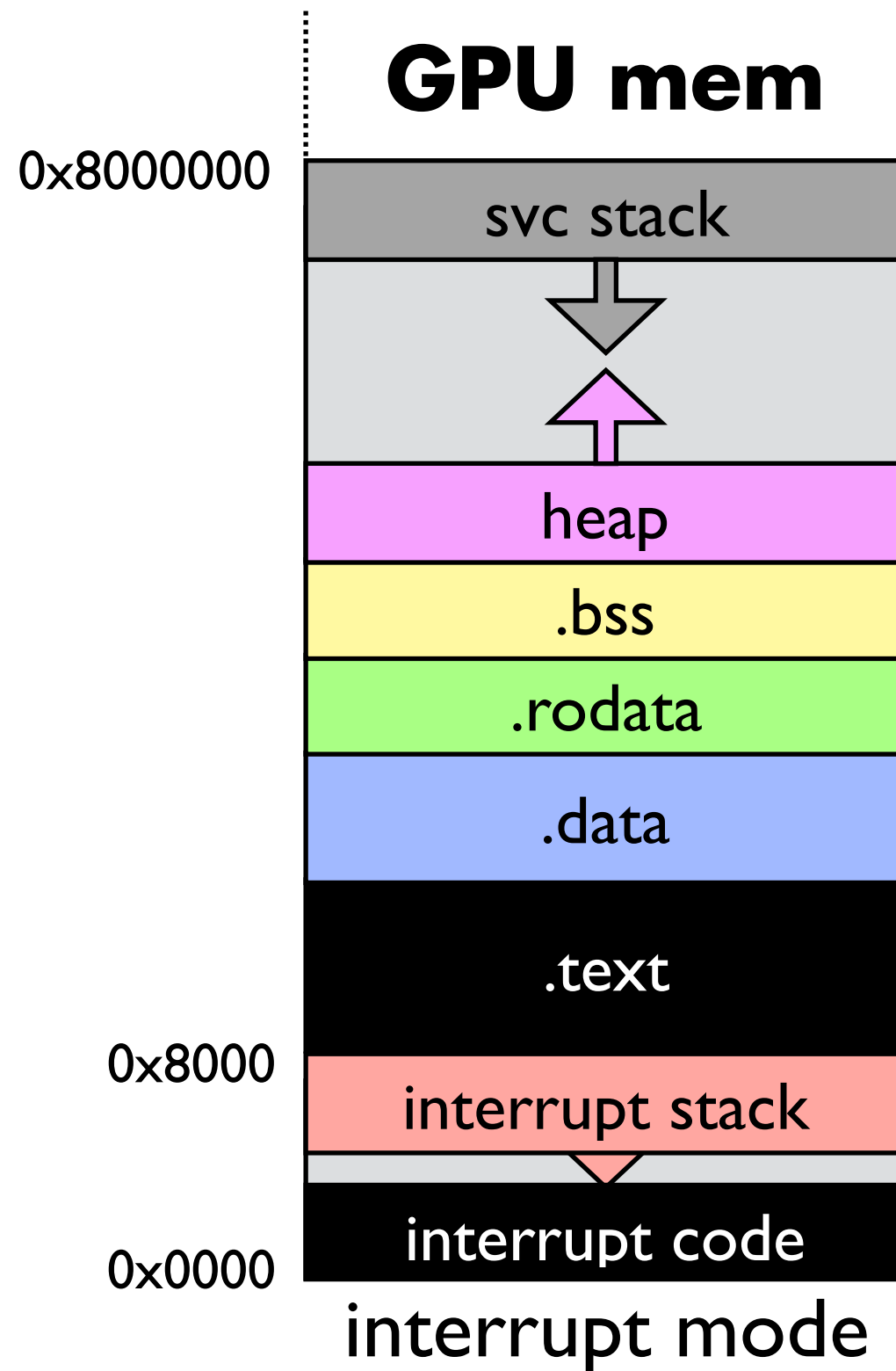
pop {r0-r12, lr}

movs pc, lr


```
interrupt_asm:  
    mov sp, #0x8000  
    sub lr, lr, #4  
  
    push {r0-r12,lr}  
  
    mov r0, lr  
    bl  interrupt_handler  
  
    pop {r0-r12, lr}  
    movs pc, lr
```

**Why do we save all of the registers?
Where do we save the registers?**

Interrupt Stack



Interrupt occurs right before instruction

Disassembly of section .text:

```
00008000 <_start>:
    8000:      e3a0d902      mov     sp, #32768      ; 0x8000
    8004:      eb000001      bl     8010 <_cstart>

00008008 <hang>:
    8008:      eb000039      bl     80f4 <led_on>
    800c:      eafffffe      b     800c <hang+0x4>

00008010 <_cstart>:
    8010:      e92d4800      push   {fp, lr} ← Interrupt!
```

**What is the pc when the interrupt occurs?
Where can we store that information?**

```
interrupt_asm:
    mov sp, #0x8000
    sub lr, lr, #4

    push {r0-r12,lr}

    mov r0, lr
    bl  interrupt_handler

    pop {r0-r12, lr}
    movs pc, lr
```

Why do we subtract 4 from lr?

What is the value passed to interrupt_handler?

Processor Modes

User - unprivileged mode

IRQ - interrupt mode

FIQ - fast interrupt mode

Supervisor - privileged mode, entered on reset

Abort - memory access violation

Undefined - undefined instruction

System - privileged mode that shares user regs

Shared / Unshared Registers

General Registers and Program Counter Modes

User32	FIQ32	Supervisor32	Abort32	IRQ32	Undefined32
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fig	R8	R8	R8	R8
R9	R9_fig	R9	R9	R9	R9
R10	R10_fig	R10	R10	R10	R10
R11	R11_fig	R11	R11	R11	R11
R12	R12_fig	R12	R12	R12	R12
R13	R13_fig	R13_svc	R13_abt	R13_irq	R13_und
R14	R14_fig	R14_svc	R14_abt	R14_irq	R14_und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

Program Status Registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fig	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

Return from Interrupt

interrupt_asm:

mov sp, #0x8000

sub lr, lr, #4

push {r0-r12,lr}

mov r0, lr

bl interrupt_handler

pop {r0-r12, lr}

movs pc, lr

Restore the registers

movs causes a return from interrupt

**Can this code be written in
C?**

Summary

Interrupts allow external events to trigger code to run with very little delay: responsiveness despite long-running functions

- **They bring together everything you've learned so far**

Running code at arbitrary points is dangerous!

- **Copies of lr and sp, use separate stack**

Interrupt vectors are at 0x0-0x1c

- **Have to copy them there at boot time**
- **Generating safe assembly requires explicitly embedding addresses**

Next time: using and writing interrupts (the return of GPIO)