Revision question

- In the following inline assembly statement, what is the purpose of "=r" in the constraint list?

```
int result;
__asm__ ("ADD %0, %1, %2" : "=r" (result) : "r" (a), "r" (b));
```

  Output operator; the result %0 will be stored in variable result.

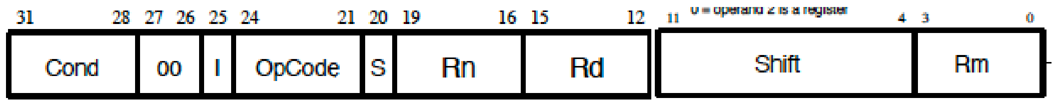
- What will the value of x be after executing the following C program?

```
int x = 10;
int y;
__asm__ ("LDR %0, [%1]" : "=r" (y) : "r" (&x));
```

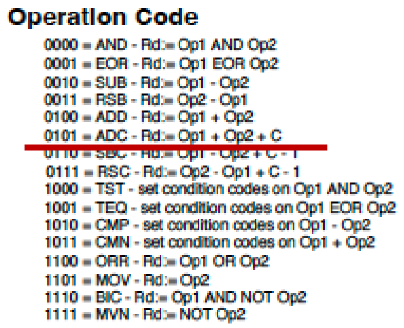
  *y = *(&x) //pass the addr. of x in asm and use ldr to read value*

Revision Question

- Which instruction is the following? adc r0, r1, r3



```
1110 0000 1010 0001 0000 0000 0000 0011
```



Operation Code

0000 = AND - Rd:= Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2
0011 = RSB - Rd:= Op2 - Op1
0100 = ADD - Rd:= Op1 + Op2
0101 = ADC - Rd:= Op1 + Op2 + C
~~0110 = SBC - Rd:= Op1 - Op2 + C - 1~~
0111 = RSC - Rd:= Op2 - Op1 + C - 1
1000 = TST - set condition codes on Op1 AND Op2
1001 = TEQ - set condition codes on Op1 EOR Op2
1010 = CMP - set condition codes on Op1 - Op2
1011 = CMN - set condition codes on Op1 + Op2
1100 = ORR - Rd:= Op1 OR Op2
1101 = MOV - Rd:= Op2
1110 = BIC - Rd:= Op1 AND NOT Op2
1111 = MVN - Rd:= NOT Op2

- Convert the following instruction to binary format:

add r3, r4

8 bits	1 bit	4 bits	3 bits
01000100	0	0100	011

0x 4 4 2 3 (hex)

CPU Exceptions

Overview

- Program flow is controlled by the PC.
- Branching (B) can change the program flow.
- Functions allows code modularity and reuse.

- What happens when my code is buggy?
  - Reading an invalid memory address.
- How can a peripheral device tell the CPU that something happened?
  - New data on Bluetooth device.

Exceptions

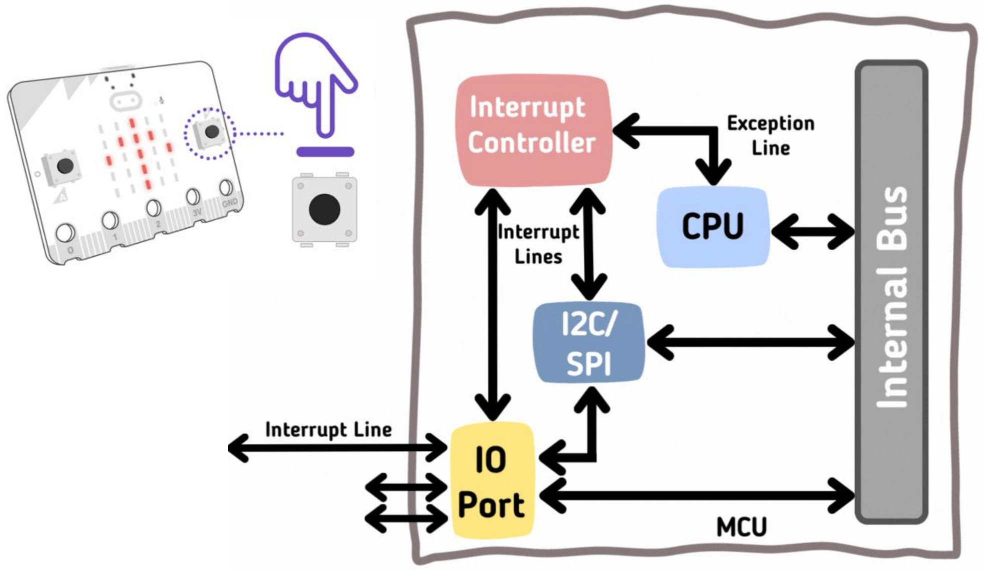
- Program execution is defined by the programmer.
  - Nonetheless, events occur external to the program logic.
- An exception is like an unscheduled function call that branches to a new address.
  - Allows a CPU to notify your program to implement code that manages external events.
- Exceptions may be caused by hardware or software.
  - User presses a key on a keyboard (hardware exception, input/output (I/O) device).
  - Encounter undefined instructions.
  - System call (SVC); a program invokes a service OS, running at a higher privilege level.

Exception Types

- An **Exception** is any unexpected change in flow control
- **Synchronous Exception**:
  - Caused by an instruction in the running program
  - Arithmetic exceptions, invalid memory addresses in load/store, trap instructions, …
- **Asynchronous Exception**:
  - Caused by an I/O device requesting the processor
  - Also known as *Hardware Interrupt*

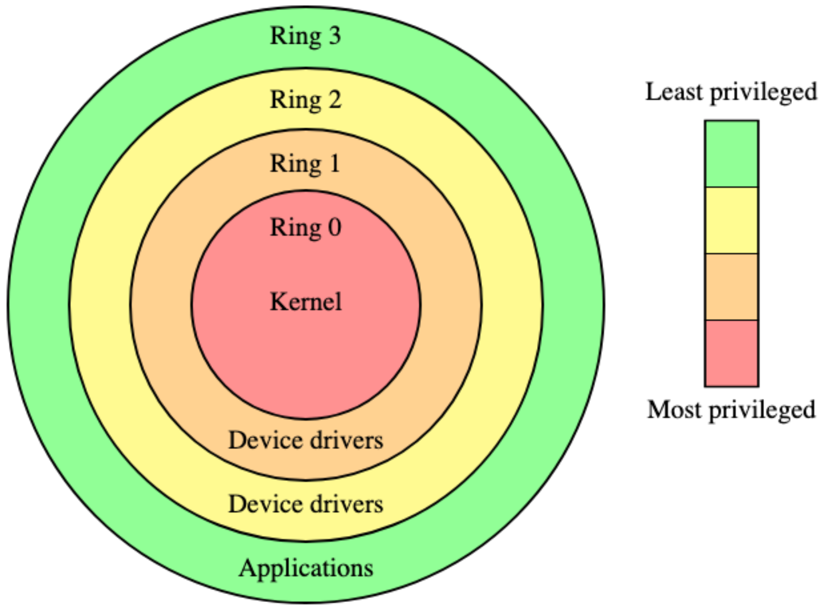
Example: Hardware Interrupts

- When you press the micro:bit button, an interrupt is raised to the CPU.
- The OS interrupts the running code to service the interrupt.
  - The Interrupt handler processes the interrupt
  - The OS returns control to the running code, when done.



CPU Privilege Levels

- Computers do not allow full access to resources to any code.
  - Buggy code can corrupt systems.
  - Virtualisation (device, memory).
  - Security.
- CPUs offer can limit resource access to different processes.
  - x86: normal process run in Ring3, OS in ring 0.



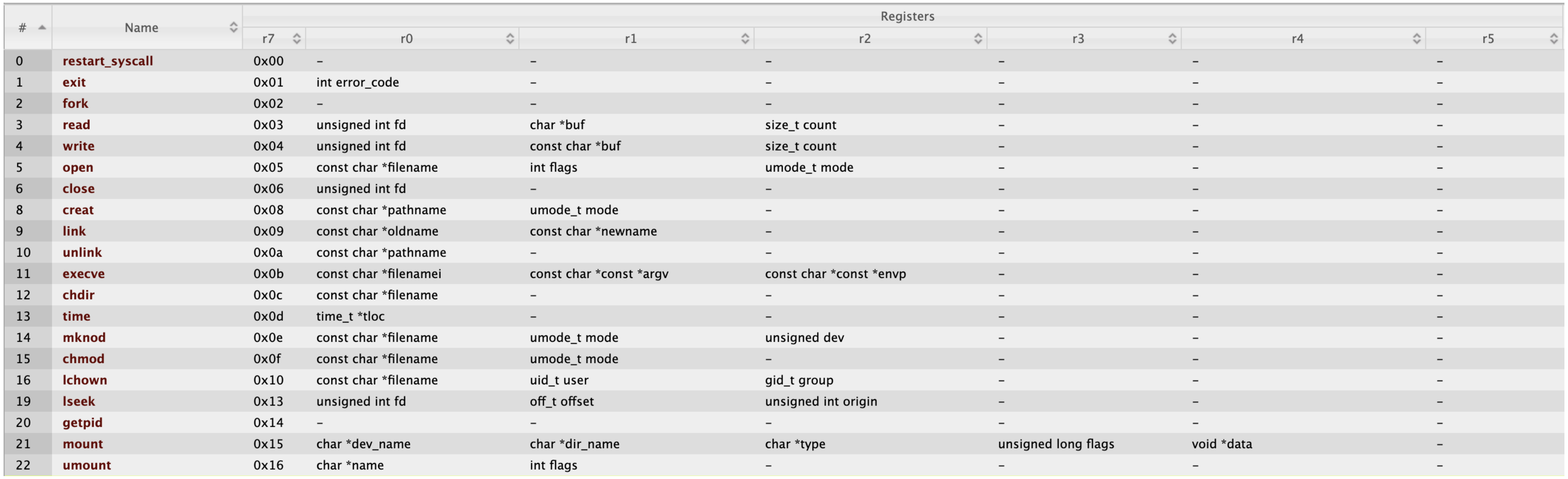
Example: Supervisor Instruction

- Every CPU architecture provides a way for user code to request a service from the OS (e.g., print to standard IO)
  - SVC in ARM
  - syscall in Intel x86
- In the micro:bit, your code runs "bare metal" – **there is no OS**
- In real world, only way for user code to access system resources is by invoking functions in the OS (OS is **"trustworthy"** service provider)
  - These functions are called **system calls**

Transitioning between Privilege Levels

- User code operates at a low privilege level.
- Supervisor instruction (`svc`) is used to transition between levels.
- CPU reads the arguments from “certain” registers and executes the specific flavour of the `svc` instruction.
- SVC is a software exception.
- A tables specifies **what actions to take** next based on the service user code wants from the OS.

Sample Supervisor Call in Linux

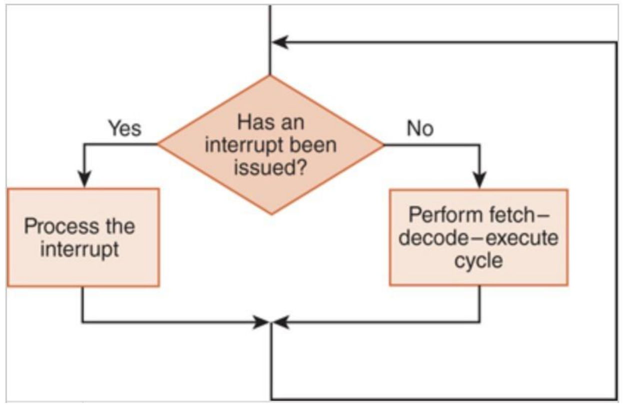


#	Name	Registers						
		r7	r0	r1	r2	r3	r4	r5
0	**restart_syscall**	0x00	–	–	–	–	–	–
1	**exit**	0x01	int error_code	–	–	–	–	–
2	**fork**	0x02	–	–	–	–	–	–
3	**read**	0x03	unsigned int fd	char *buf	size_t count	–	–	–
4	**write**	0x04	unsigned int fd	const char *buf	size_t count	–	–	–
5	**open**	0x05	const char *filename	int flags	umode_t mode	–	–	–
6	**close**	0x06	unsigned int fd	–	–	–	–	–
8	**creat**	0x08	const char *pathname	umode_t mode	–	–	–	–
9	**link**	0x09	const char *oldname	const char *newname	–	–	–	–
10	**unlink**	0x0a	const char *pathname	–	–	–	–	–
11	**execve**	0x0b	const char *filenamei	const char *const *argv	const char *const *envp	–	–	–
12	**chdir**	0x0c	const char *filename	–	–	–	–	–
13	**time**	0x0d	time_t *tloc	–	–	–	–	–
14	**mknod**	0x0e	const char *filename	umode_t mode	unsigned dev	–	–	–
15	**chmod**	0x0f	const char *filename	umode_t mode	–	–	–	–
16	**lchown**	0x10	const char *filename	uid_t user	gid_t group	–	–	–
19	**lseek**	0x13	unsigned int fd	off_t offset	unsigned int origin	–	–	–
20	**getpid**	0x14	–	–	–	–	–	–
21	**mount**	0x15	char *dev_name	char *dir_name	char *type	unsigned long flags	void *data	–
22	**umount**	0x16	char *name	int flags	–	–	–	–

https://syscalls.w3challs.com/?arch=arm_thumb

Managing Exceptions

- An exception will create an execution interrupt for your CPU.
- Process looks like a special function.
  - Function is called by the CPU, not by your program.
- Interrupt handling:
  - Stop normal cycle (fetch-decode-execute).
  - Save current CPU state, i.e. registers.
  - Run user code to manage interrupt.
  - Change code execution to interrupt handling code.

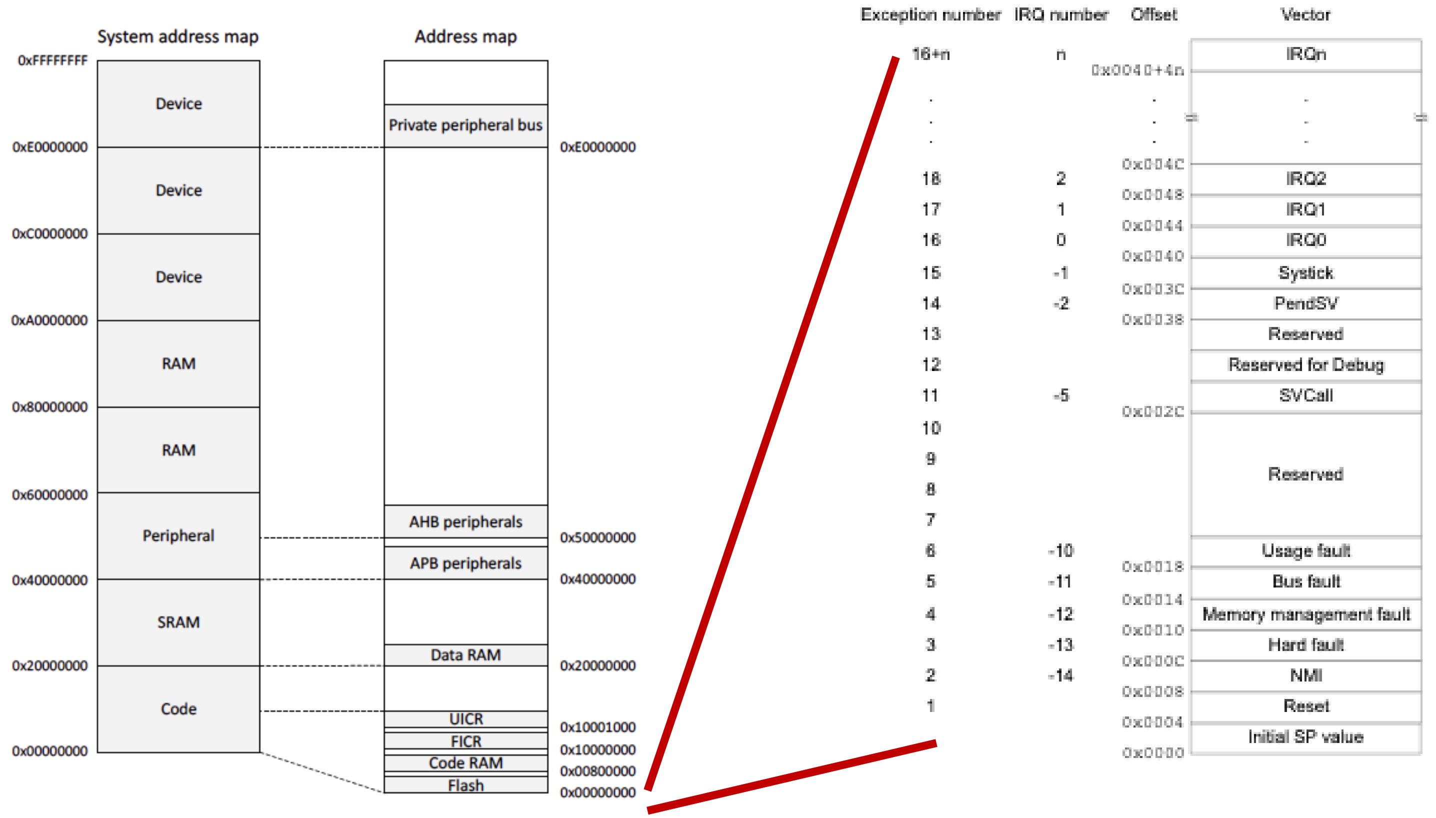

Has an interrupt been issued?
Yes
No
Process the interrupt
Perform fetch–decode–execute cycle

Exception Handling

- **Exception handler:** Exceptions use a vector table to determine where to jump to the exception handler.
  - Stores an address to execute next when exception/interrupt is raised.
  - Remember function labels.
- The table is placed in low-memory
- Instructions to handle an interrupt is at `0x00000018`
- On power-up, the CPU goes to address `0x00000000`

- The exception vector contains a **branch instruction to an exception handler**, code that handles the exception and then **returns to user code**

Vector Table

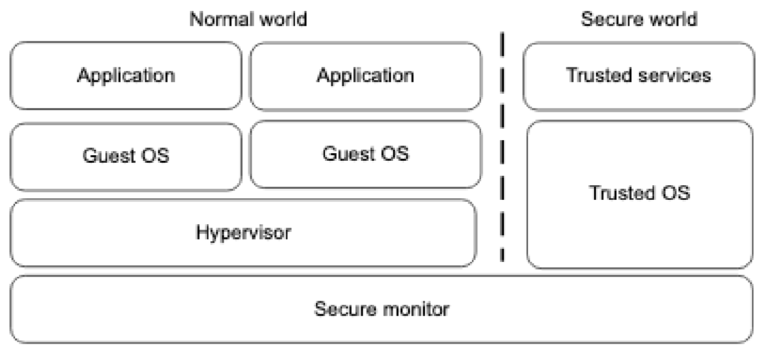


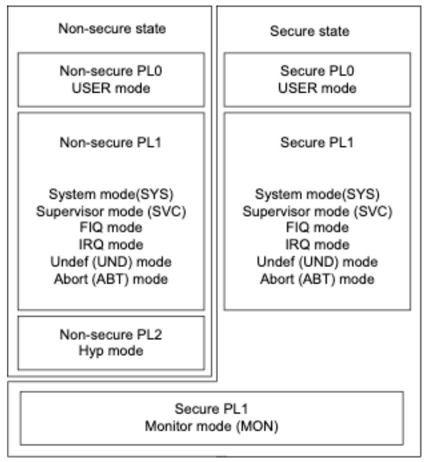
Internal Interrupts

IRQ	Priority	Source	Comment
-1	0	Systick	*24-bit clock*
-2	0	PendSV	*SVCall handler*
-5	0	SVCall	*SVC instruction*
-13	-1	Hard Fault	*Hardware failures*
-14	-2	NMI	Interrupts
	-3	Reset	Power or Reset

ARM Execution Modes

- ARM processors can operate in one of several execution modes with different privilege levels
- The mode is specified in the bottom bits of the CPSR
- User mode operates at privilege level PL0 and other modes operate at PL1, which can access all system resources
- Execution mode helps the CPU with proper book-keeping
  - Which registers to save (more on this later)
  - Where to return after an interrupt?





Banked Registers

- Exception handler is like a normal function call:
  - Need to know where to return
  - *Need to preserve registers*
- Stack is in memory and memory is slow.
- FIQ is a fast, low-latency interrupt handling mechanism in ARM.
- FIQ uses banked registers, i.e. extra (shadow) registers, used to copy values from actual registers on an interrupt, i.e. faster interrupt handling.

Current Program Status Register (CPSR)

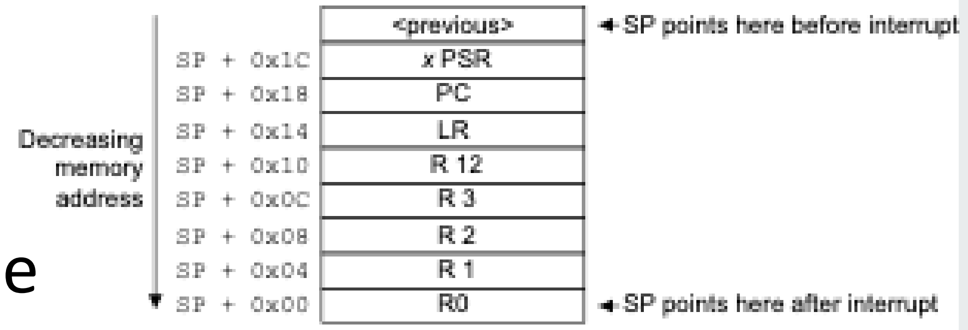
- CPSR is an ARM register that records the state of the program.
  - Arithmetic instructions will affect its value every time.
- **I bit - ”IRQ flag”:** disable IRQ interrupts.
- **F bit -”FIQ flag”:** disable FIQ (Fast IRQ) interrupts.
- **T bit - ”Thumb flag”:** Disable thumb mode.
- **Mode:** Current execution mode.

31	30	29	28		7	6	5	4 0
N	Z	C	V	…	I	F	T	Mode

pattern	Mode
10000	User
10010	IRQ
10011	Supevisor
11111	System

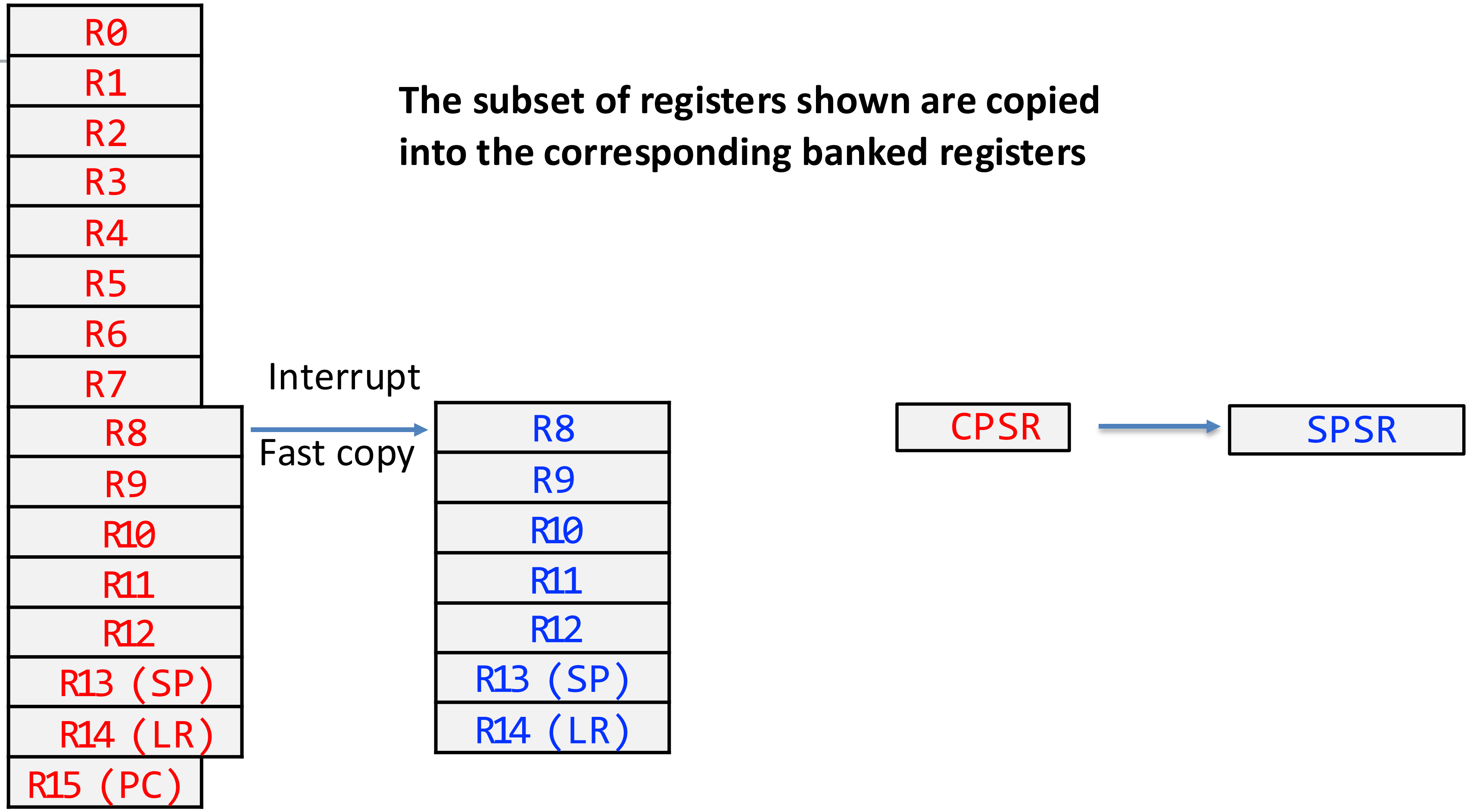
Exception Handling

1. Store the cpsr into banked spsr.
2. Set execution mode and privilege level based on exception type.
3. Disable interrupts using thr interrupt mask bits in cpsr.
4. Store the return address into banked `lr`
5. Pushes other registers onto stack
6. Branch to the exception function based on the address from the vector table
7. Execute exception handler
8. Copy `lr` → `pc` (`movs pc, lr`), banked `spsr` → `cpsr`
9. Pop the registers back off the stack



Example: Fast Interrupt Ex. Mode (FIQ)

The subset of registers shown are copied into the corresponding banked registers



Example: BusFault Handler

- A **BusFault** occurs during a memory access error in the **bus interface.** The **bus interface** transfers data between the CPU and peripherals, memory, or external devices.
  - Examples: Invalid memory address, misaligned memory access, faulty memory, corrupted stack pointer.
- On Cotrtex-M0, a BusFault raises escalates to a HardFault.

System Control Unit

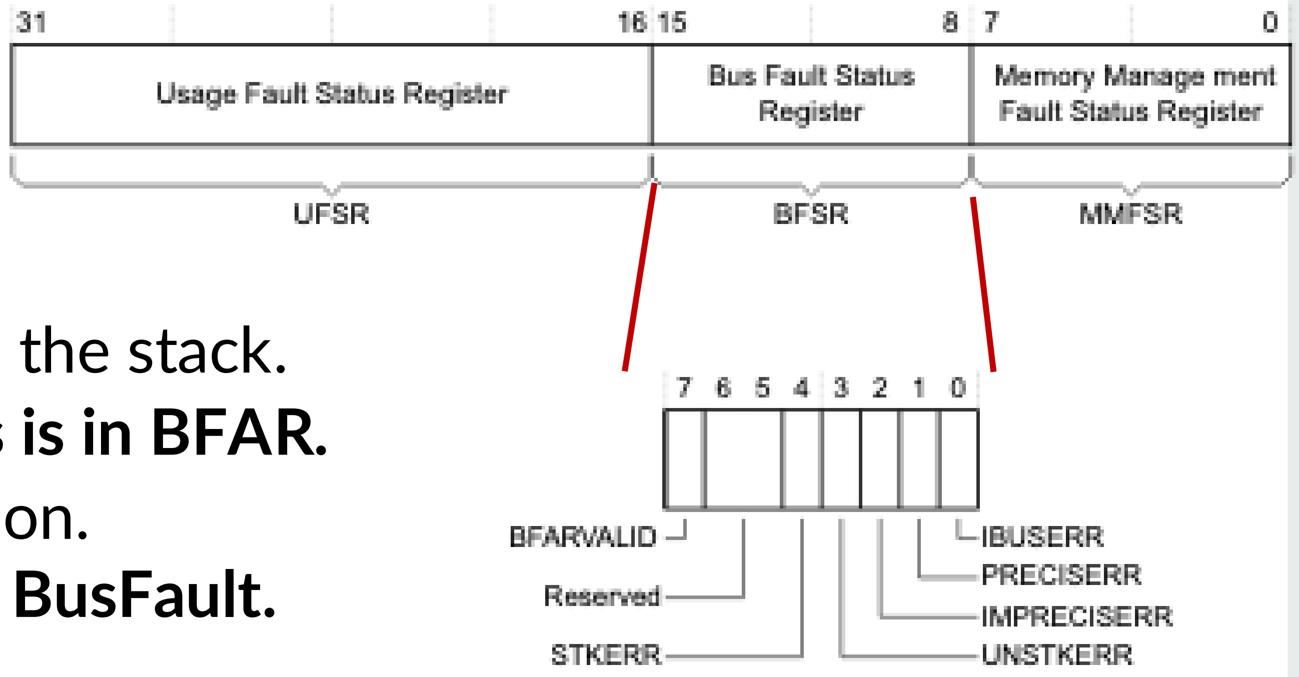
- The **System Control Unit (SCU)** in the ARM Cortex-M0 handles various **system-level operations,** like power management and reset control.
  - **SCR**: Controls **sleep modes** (Sleep, Deep Sleep).
  - **CCR:** Controls **stack alignment and unaligned memory access behaviour**.
  - **CFSR**: Provides details of the current HardFault.
  - **BFAR**: Stores the bus address that caused the error.

Address	Name	Description
0xE000E008	ACTLR	Auxiliary Control Register
0xE000ED00	CPUID	CPUID Base Register
0xE000ED04	ICSR	Interrupt Control and State Register
0xE000ED10	SCR	System Control Register
0xE000ED14	CCR	Configuration and Control Register
0xE000ED28	CFSR	Configurable Fault Status Register
0xE000ED28	MMSR	MemManage Fault Status Register
0xE000ED29	**BFSR**	**BusFault Status Register**
0xE000ED2A	UFSR	UsageFault Status Register
0xE000ED2C	HFSR	HardFault Status Register
0xE000ED34	MMAR	MemManage Fault Address Register
0xE000ED38	**BFAR**	**BusFault Address Register**

https://developer.arm.com/documentation/dui0552/a/cortex-m3-peripherals/system-control-block

BusFault Status Register (BFSR)

- The **BFSR** is part of the **CFSR.**
- The BSSR contains register that Explain the nature of the error.
  - STKERR/UNSTKERR: Error with the stack.
  - **PRECISERR: Error Mem address is in BFAR.**
  - IBUSERR: Error reading instruction.
  - **BFARVALID: Records if this is a BusFault.**



https://developer.arm.com/documentation/dui0552/a/cortex-m3-peripherals/system-control-block/configurable-fault-status-register

Example: BusFault

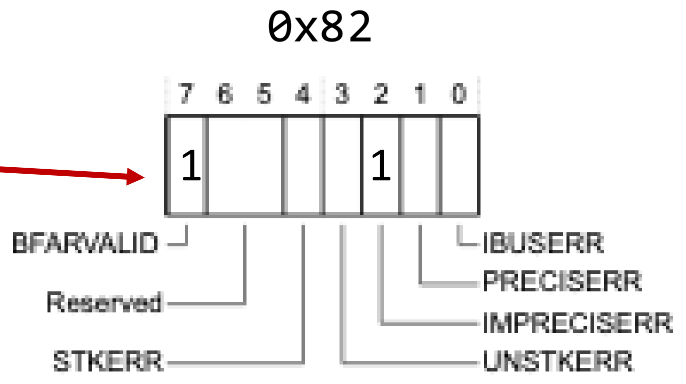
```
_start:
  @ Load an invalid memory address
  ldr r0, =0xFFFFFFF0

  @ Attempt to read from it (causes HardFault)
  ldr r1, [r0]
  b _start
.global HardFault_Handler
.type HardFault_Handler, %function HardFault_Handler:

  @ Get the fault status register - BFSR
  ldr r0, =0xE000ED28
  ldr r1, [r0]

  @ Get the memory address - BFAR
  ldr r0, =0xE000ED38
  ldr r1, [r0]

  bx lr
```



Conclusion

- CPU uses interrupts to integrate external events to the programming model.
  - Fault: memory, execution etc.
  - Interrupt: HW events, timers etc.
- The IRQ table allows users to program custom event handlers.
- HardFault interrupt examples.
- Next lecture: MMIO.