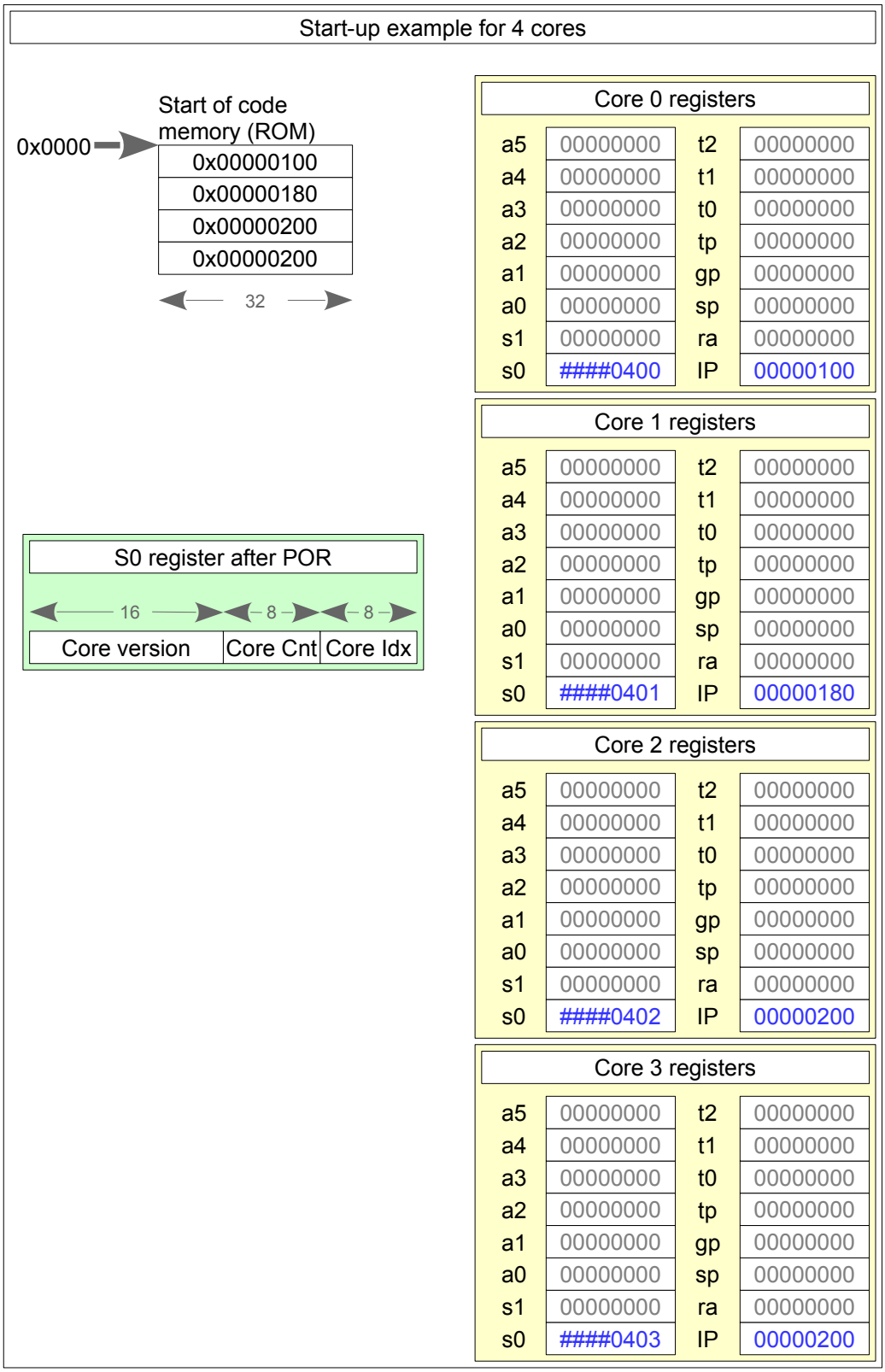
CPU startup

After power-on reset processor reads a table at the beginning of code memory. Each element of the table must contain the IP address of each processor core.



In the example above the system has 4 processing cores. Core 0 starts execution from address 0x00000100, core 1 starts execution from address 0x00000180, cores 2 and 3 start execution from address 0x00000200 eventually executing the same code. 32-bit register **s0** will contain information about

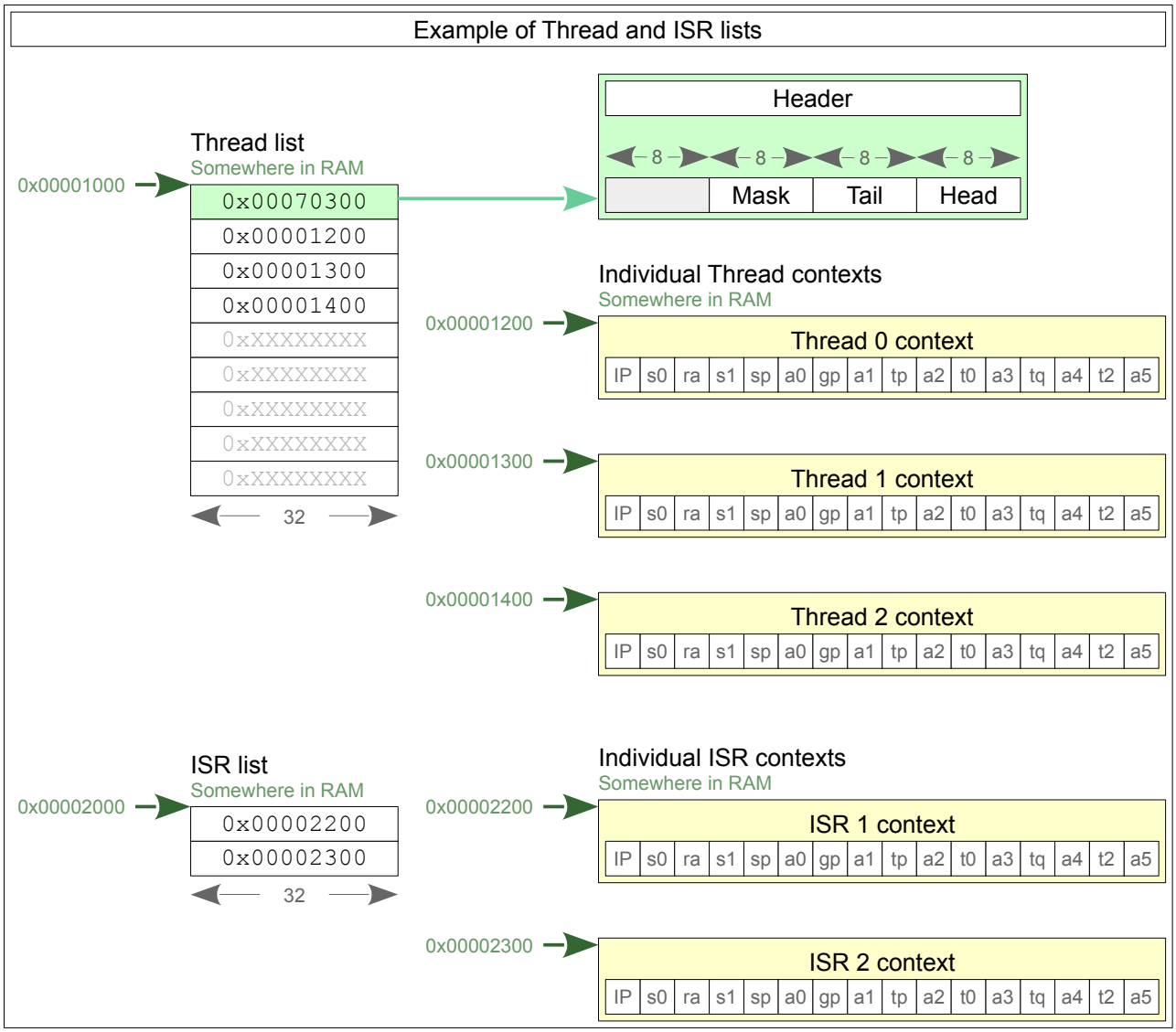
core/processor version, total number of cores and an individual core index. All other registers will contain zeroes.

Of course each core can execute its dedicated thread (task, hart). But this way can be very inefficient: if the thread waits for an event or some data it can release the core and thus give processing time for another thread. Also, the SW system may contain more threads than cores. The HW provides the mechanism to switch from one thread to another called the “context switch”. It means that the core can save the entire register file from the current thread to memory and load another register file of the text thread. The register file is called a “thread context”.

The context switch can be initiated both by the thread and by the interrupt.

The context switch is automatic and performed by the hardware. The following explanation and examples are provided for clarification purpose.

There is a special hardware which performs the context switch. At the beginning of execution this hardware needs to be initialized.



In the example above there are 3 threads and 2 ISRs. All elements can be located anywhere in RAM. Address is highlighted in green.

Thread table has a header, ISR table has no header. In Thread table the “mask” field indicates how many elements can be located in total. 0x07 = 8 elements, 0x0F = 16 elements and so on. It also contains the positions of Head and Tail. Software can add elements to the table and remove them. There cannot be more than 8 elements in total for the given example.

When Hardware performs a context switch, it stores the current thread at the Tail of the table and picks a new thread from the head of the table.

The SW can choose whether to perform a context switch or terminate the thread. In the second case the current context will not be saved and the core will only pick the next thread.

There is a special command “end thread” (In Mirabelle ISA there is a special opcode, but can be implemented as a bit in SFR).

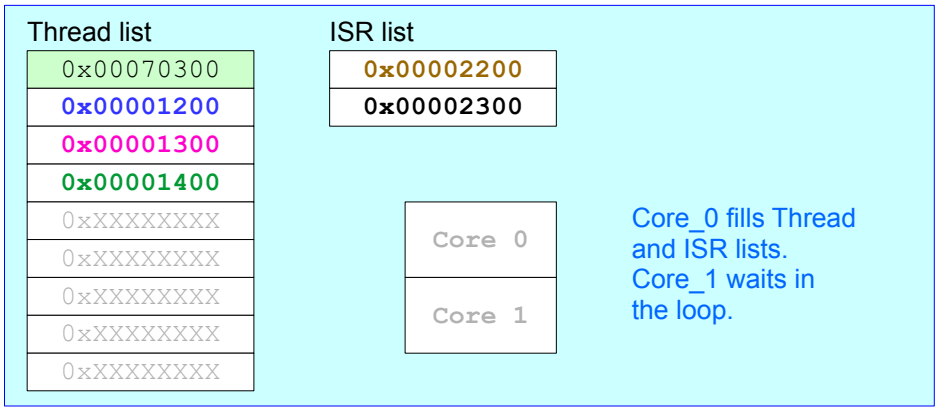
A generic conditional or unconditional JMP command can be used as “switch thread”.

Example

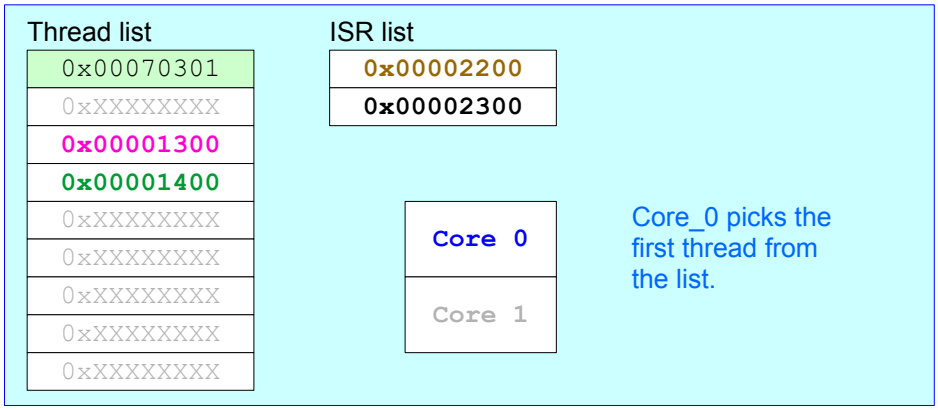
Imagine the HW containing 2 processing cores. For illustration purpose the thread context addresses will be highlighted by colours.

Let's see by example the processing step-by-step.

After POR core 0 will create the Thread and ISR tables, and core 1 will wait in the loop.

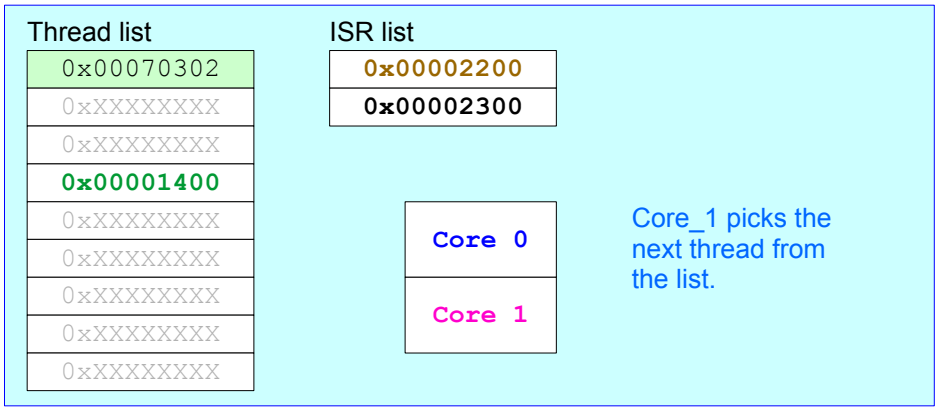


When Core_0 finishes with tables, it executes the “end thread” command and picks “blue” thread from the list.



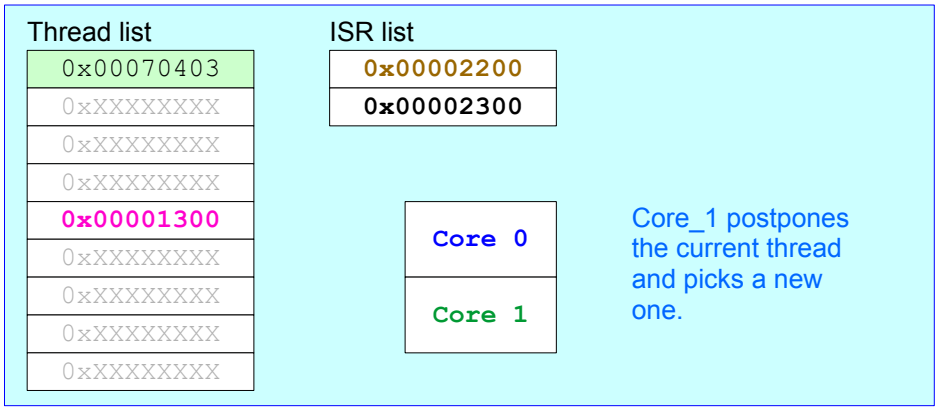
The Thread Head field (highlighted by green) in the Thread table header has changed from 00 to 01 (i.e. the value has changed from 00070300 to 00070301).

Now Core_1 exits the waiting loop and executes the “end thread” command and picks “pink” thread from the list.



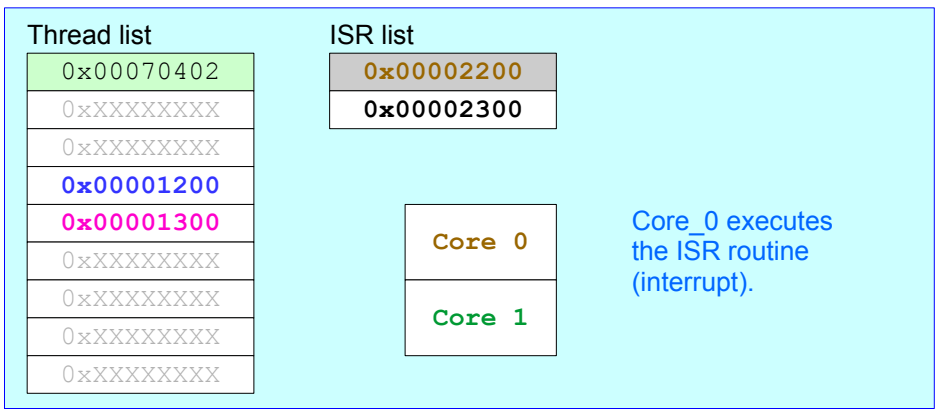
Head position has changed again, now from 01 to 02 (see value which is now 00070302).

While Core_0 is still busy with it's thread, Core_1 waits data from UART and data is not yet there. So, Core_1 can postpone the current thread and execute the next one. It performs the “context switch”.



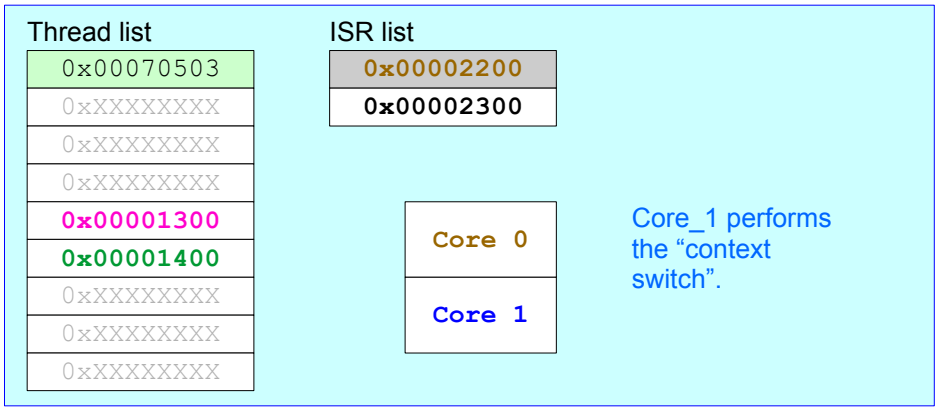
Now both Head and Tail position has changed (new value is 00070403).

Interrupt 0 arrives. Let's assume that interrupts are disabled by Core_1 and enabled by Core_0, so ISR will be executed by Core_0.



The thread interrupted by the IRQ is inserted to the Head (and not Tail) position. The Head index is decremented (new value is 00070402).

Let's imagine that while Core_0 is still executing the ISR, the “green” thread executed by Core_1 can be postponed. Core_1 will perform the “context switch”.

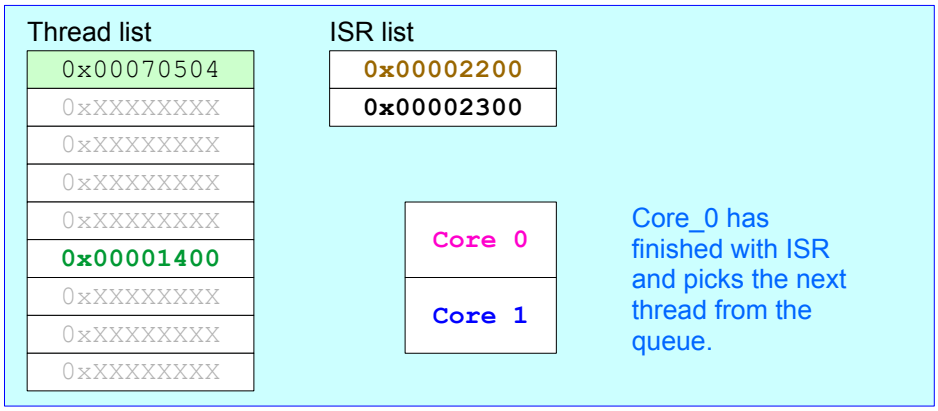


Core_1 picks the “blue” thread from the Head, interrupted some time ago by the IRQ.

This “blue” thread was executed by Core_0 before it was interrupted. It turns out that now it is Core_1 which continues.

Pay attention that when thread intentionally has decided to switch then it is written to the Tail position of the queue and thus has the longest time to wait a free core to pick it. But in this example the thread was interrupted by the incoming event, so it was not the thread itself who decided to switch. And in this case it is written to the Head position and has the highest priority and the shortest time to wait for the free core to be picked.

Now imagine that Core_0 has finished with ISR.



Note that the order of threads in the list can easily change.