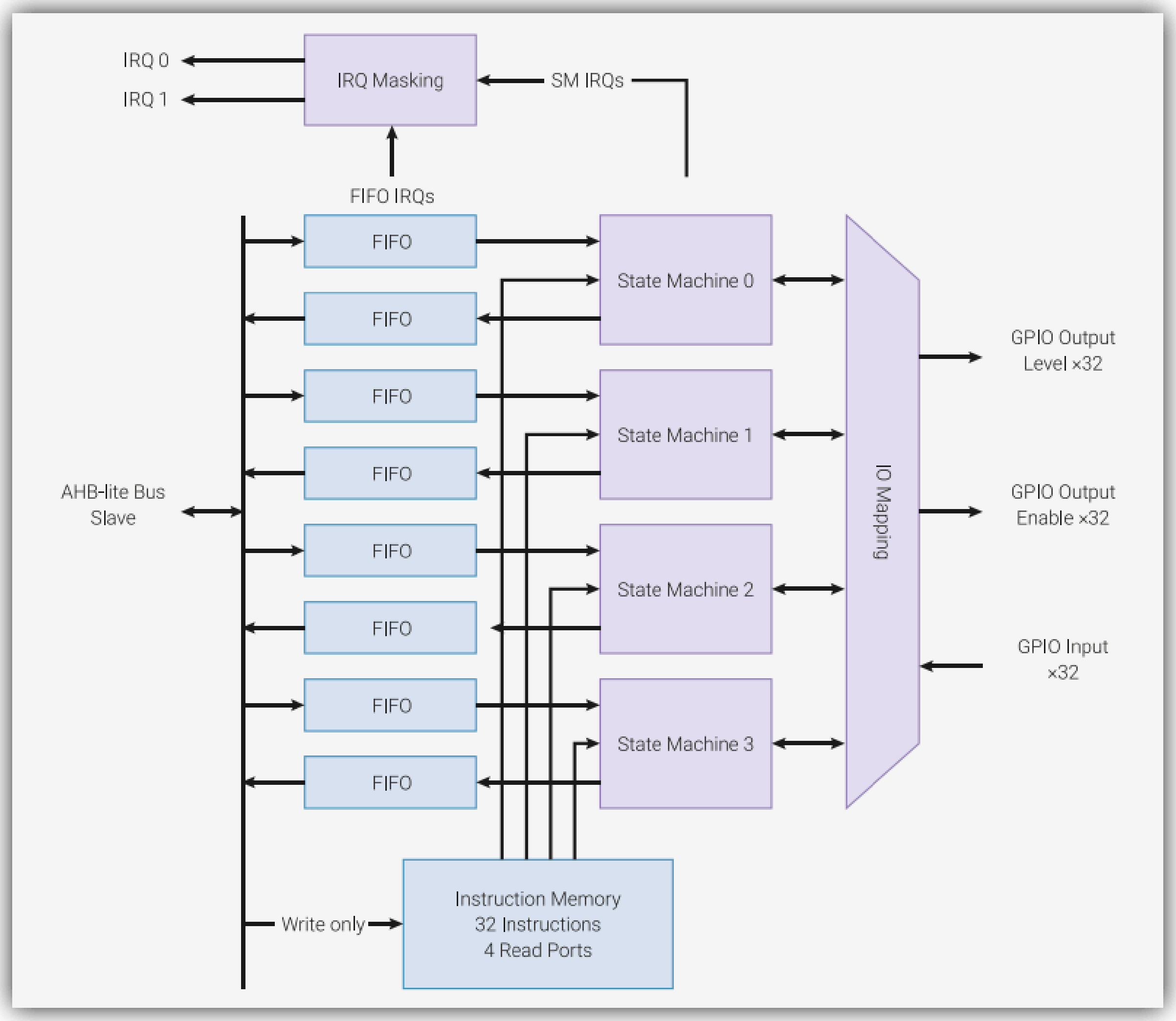
Lab-4. EE3401, Jan-Apr2024

In this lab we will work out the examples a particular processor called PIO (programmable IO) on Raspbeery Pi Pico board, which consists of RP2040 processor. This processor has a ARM Cortex M0 dual core and also contains a particularly nice hardware block called programmable IO (PIO) block. It can be programmed by its own instruction set which provides a good contrast to the ARM ISA we have worked till now.

The schematic diagram of this block is this:



It consists of 4 state machines (pink blocks in the middle), which control GPIO pins (pink trapezoid on the right) on the RP2040 processor. It is not connected to the memory directly, but a bunch of FIFO queues, which can get data from the memory (using something called direct memory access).

“For each simple processor (called a state machine), there are two First-In-First-Out (FIFO) structures: one for data coming in and one for data going out. These are used to link the state machine to everything else. Essentially, they are just a type of memory where the data is read out in the same order it’s written in – for this reason, they’re often called queues because they work in the same way as a queue of people. The first piece of data in (or first person in the queue) is the first to get out the other end. When you need to send data via a PIO state machine, you push it to the FIFO, and when your state machine is ready for the next chunk of data, it pulls it. This way, your PIO state machine and program running on the main core don’t have to be perfectly in-sync. The FIFOs are only four words (of 32 bits) long, but you can link these with direct memory access (DMA) to send larger amounts of data to the PIO state machine without needing to constantly write it from your main program. The FIFOs link to the PIO state machine via the input shift register and the output shift register. There are also two scratch registers called X and Y. These are for storing temporary data. The processor cores are simple state machines that have nine instructions:

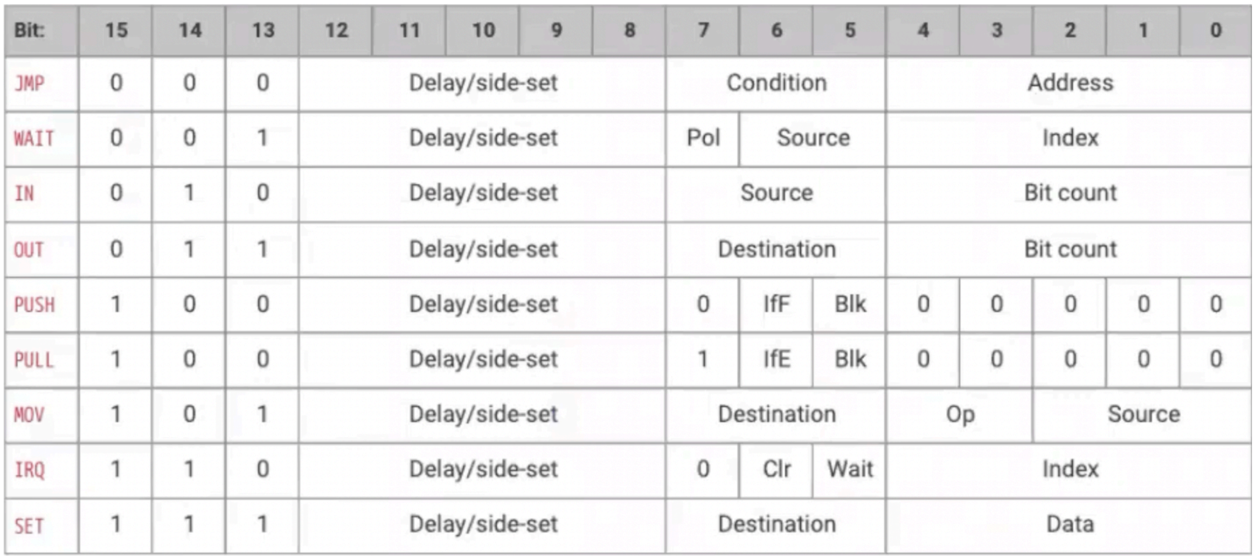
- **IN** shifts 1–32 bits at a time into the input shift register from somewhere (such as a set of pins or a scratch register)
- **OUT** shifts 1–32 bits from the output shift register to somewhere
- **PUSH** sends data to the RX FIFO
- **PULL** gets data from the TX FIFO
- **MOV** moves data from a source to destination
- **IRQ** sets or clears the interrupt flag
- **SET** writes data to destination
- **WAIT** pauses until a particular action happens
- **JMP** moves to a different point in the code

Within these there are options to change the behaviour and locations of the data. Take a look at the data sheet for a full overview.

In addition, there are a couple of features to make life a little easier:

- Side-setting is where you can set one or more pins at the same time that another instruction runs
- Delays can be added to any instruction (the number of clock cycle delays in square brackets)”

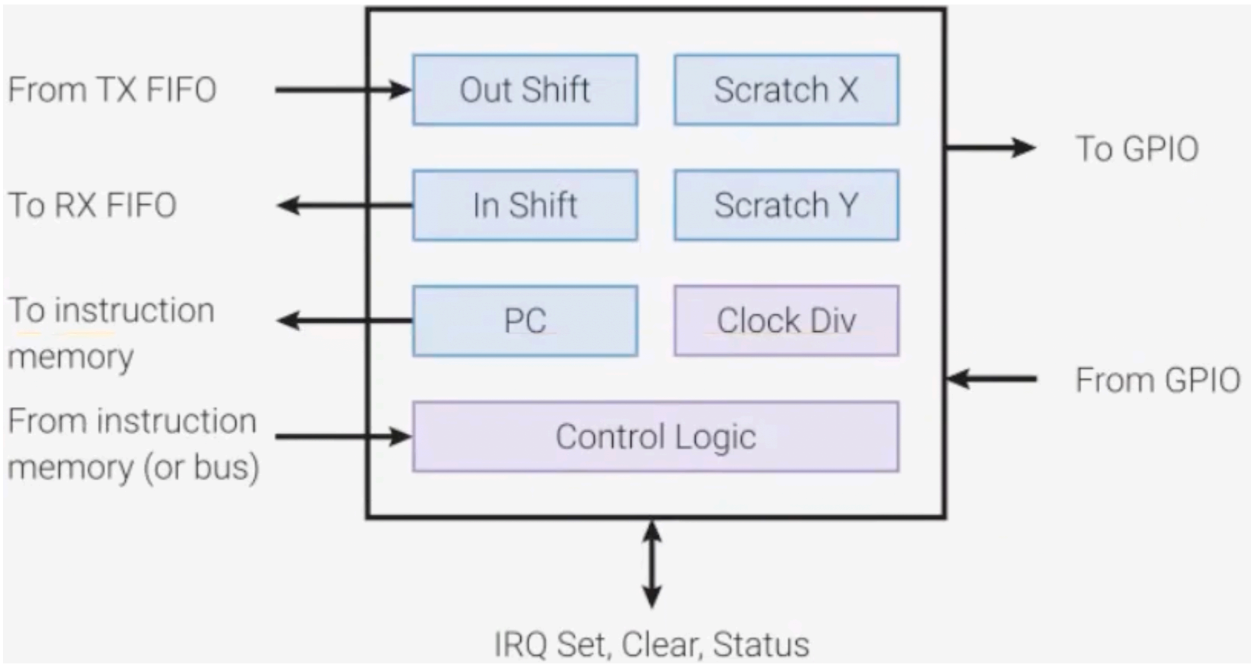
The opcodes look like:



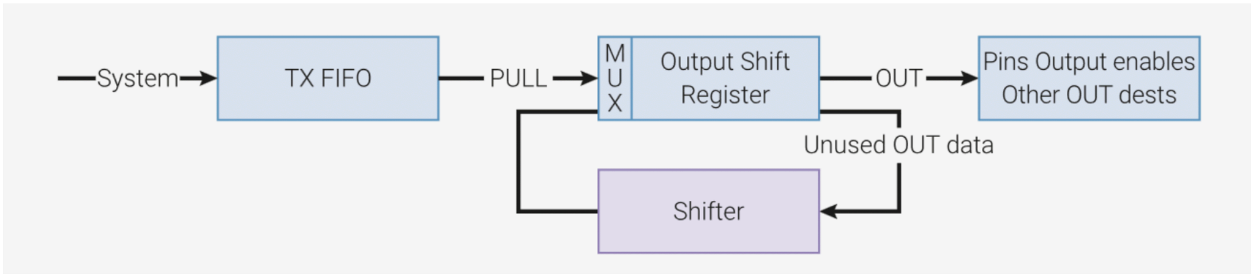
Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JMP	0	0	0	Delay/side-set					Condition			Address				
WAIT	0	0	1	Delay/side-set					Pol	Source		Index				
IN	0	1	0	Delay/side-set					Source			Bit count				
OUT	0	1	1	Delay/side-set					Destination			Bit count				
PUSH	1	0	0	Delay/side-set					0	IfF	Blk	0	0	0	0	0
PULL	1	0	0	Delay/side-set					1	IfE	Blk	0	0	0	0	0
MOV	1	0	1	Delay/side-set					Destination			Op		Source		
IRQ	1	1	0	Delay/side-set					0	Clr	Wait	Index				
SET	1	1	1	Delay/side-set					Destination			Data				

https://hackspace.raspberrypi.com/articles/what-is-programmable-i-o-on-raspberry-pi-pico

Each FSM can be considered as having this model:



“The Output Shift Register (OSR) holds and shifts output data, between the TX FIFO and the pins (or other destinations, such as the scratch registers).”



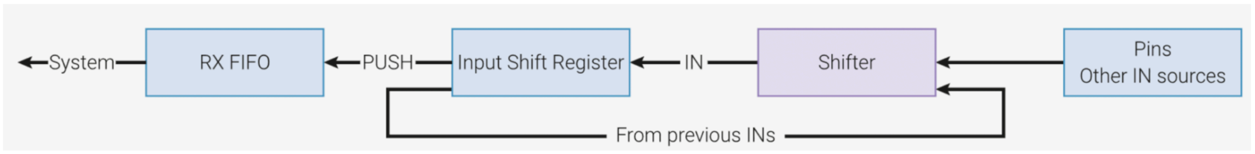
PULL instructions: remove a 32-bit word from the TX FIFO and place into the OSR.

• OUT instructions shift data from the OSR to other destinations, 1…32 bits at a time.

• The OSR fills with zeroes as data is shifted out

• The state machine will automatically refill the OSR from the FIFO on an OUT instruction, once some total shift count threshold is reached, if autopull is enabled

Input Shift Register:



- IN instructions shift 1…32 bits at a time into the register.

• PUSH instructions write the ISR contents to the RX FIFO.

• The ISR is cleared to all-zeroes when pushed.

• The state machine will automatically push the ISR on an IN instruction, once some shift threshold is reached, if autopush is enabled.

Each state machine has two 32-bit internal scratch registers, called X and Y.

They are used as:

• Source/destination for IN/OUT/SET/MOV

• Source for branch conditions

Assembler Instructions:

<instruction> (side <side_set_value>) ([<delay_value>])

<side_set_value> Is a value (see Section 3.3.2) to apply to the side_set pins at the start of the instruction. Note that the rules for a side-set value via side <side_set_value> are dependent on the .side_set (see [pioasm_side_set]) directive for the program. If no .side_set is specified then the side <side_set_value> is invalid, if an optional number of sideset pins is specified then side <side_set_value> may be present, and if a non-optional number of sideset pins is specified, then side <side_set_value> is required. The <side_set_value> must fit within the number of side-set bits specified in the .side_set directive.

<delay_value> Specifies the number of cycles to delay after the instruction completes. The delay_value is specified as a value (see Section 3.3.2), and in general is between 0 and 31 inclusive (a 5-bit value), however the number of bits is reduced when sideset is enabled via the .side_set (see [pioasm_side_set]) directive. If the <delay_value> is not present, then the instruction has no delay.

Example:

```
.program ws2812_led

public entry_point:
pull
set x, 23 ; Loop over 24 bits
bitloop:
set pins, 1 ; Drive pin high
out y, 1 [5] ; Shift 1 bit out, and write it to y
jmp !y skip ; Skip the extra delay if the bit was 0
nop [5]
skip:
set pins, 0 [5]
jmp x-- bitloop ; Jump if x nonzero, and decrement x
jmp entry_point
```

How do we experiment with this? Use an online emulator: https://wokwi.com/pi-pico.

This website lets you assemble hardware virtually and you can program them online. There are three example of the PIO programs (running a segment display, using a keypad with ISRs, and a tunes). We are going to look at the first two, run them and modify a bit.

It uses micropython (a smaller version of python) in which the PIO assembly is embedded. See https://docs.micropython.org/en/latest/library/rp2.StateMachine.html for micropython documentation.

1. Let's start with the seven segment display example. Your task is to modify it to display the counting in reverse.

2. For the other example, keypad interaction, modify the code to print “hello world” when a particular key sequence is pressed (4 digit long).