Why it can be useful
The executable SW resides ROM and Flash in embedded systems. The bigger the SW is, more ROM/Flash it occupies. This influences a lot the device area thus the final price of the device. Indirectly, it influences the power consumption.

Some theory
Processors can be classified by architecture and functions they perform (see following table). Of course, there can be multiple ways to classify or group them and the table below is just a highly simplified variant of classification.

CPU classes		
Class	Tasks in which the CPU is good	Opcode length
Vector	Video processing, AI, complex DSPs, other	>32bit
Universal (Risc-V)	All	32bit (with 16-bit extension, up to 64bit)
Universal (Intel, Mirabelle [this is ours])	All	16bit (up to 48bit with data)
Accumulator	Communication, embedded	8 bit (up to 24bit with data)
Stack	Math, simple DSP	<8 bit

Only “universal”, “accumulator” and “stack” class will be used here for simplicity and to describe the idea.

As one can see from the table, there is a big difference in opcode length.

Usually register based CPUs use registers to perform operations. In the instruction opcode the source and destination registers are encoded and the choice of registers is not restricted.

Example

If we want to compute 3+2=5, we can load arguments 3 and 2 to the registers and then perform the operation.

mov ar,3
mov br,2
add cr,ar,br ← means cr = ar + br (or 5 = 3+2)

In this example **ar** and **br** are source registers and **cr** is a destination register.

16-bit instruction opcode can encode all arithmetic, memory operations and manipulate 8 to 16 general purpose registers with no restriction on register choice.

The opcode can be reduced to 8 bit, but the number of operations and register choice will be reduced. One of the possible ways is the “accumulator” type CPU where the destination register is always the same for all commands: the “accumulator” register. The flexibility will be reduced and only ~4 general purpose registers can be used.

Another way is a “stack” architecture where all operations are performed at the top of a so-called stack. There is no register choice at all: only the top of the stack can be manipulated.

Example

If we want to compute (3+2)*4=20 on a “stack” architecture it will look like following. We fill the stack in the inverse order.

push 4
push 2
push 3

add ← The operation is performed on “3” and “2” which will be removed. “5” will be pushed
mul ← The top of the stack is “5” and “4”, mul will result “20”

It turns out that stack architecture is very good for computation, because mathematical formulas can easily be optimized for them. But they work badly with hardware peripherals and in communications.

Accumulator architectures work very well with hardware, because using only “accumulator” register is just enough. They are very good in communications also, but very bad for mathematics and other stuff.

CPUs in embedded systems work with some peripherals, computations (simple DSP) and communication. Usually “universal” CPU class is used because it covers all tasks relatively well though will use bigger code size or be less optimal (slower).

Our proposal

We made a CPU which is multi-tasking, multi-core and multi-ISA [ISA = instruction set architecture]. It can decode and execute code written for different kind of processors (at this moment only 2).

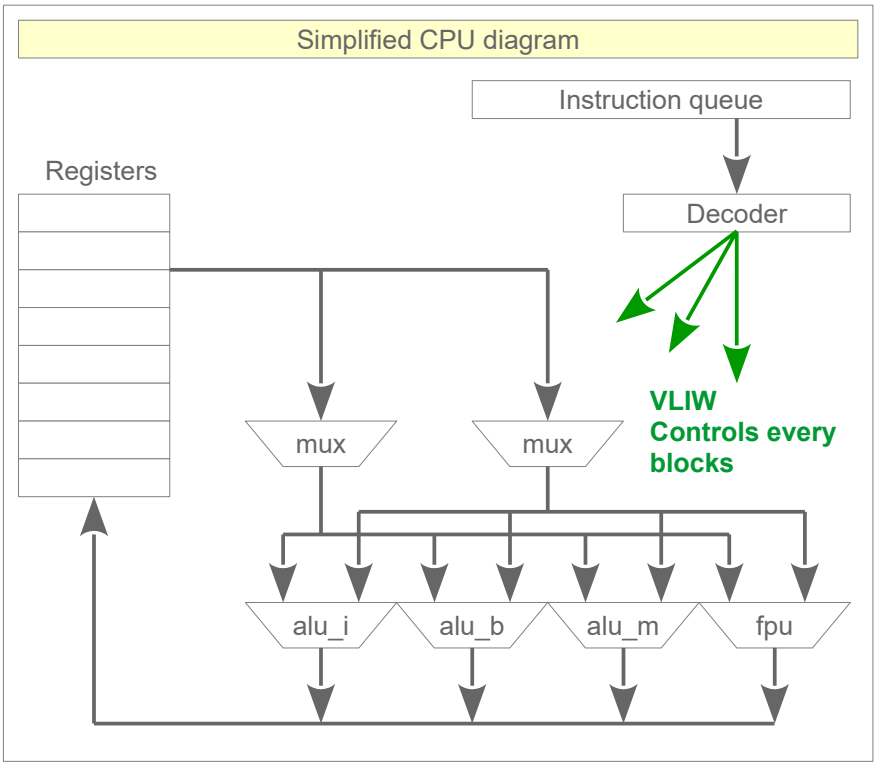
We propose to mix the architectures by mapping “accumulator” and “stack” types on a more generic “universal” model. This universal model is able to execute any piece of code.

The ISA will be set individually for each thread and will keep the same within the thread. It seems that this way is good enough and simple in implementation.

It could be possible though to compile different pieces of code using different targets and let the compiler decide which architecture is more optimal. Then it could be possible to switch the architecture upon “call” or “ret” instruction. Nevertheless in our implementation we haven’t gone so far in order to keep things simple.

HW implementation

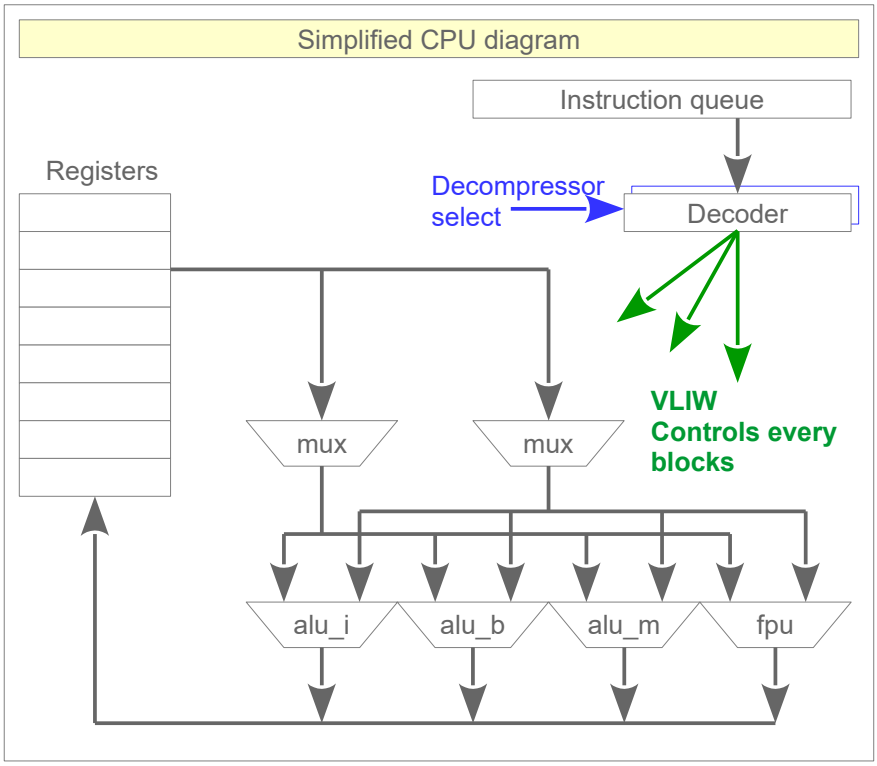
In Mirabelle CPU instruction opcode is decoded (decompressed) first into a so called VLIW (very long instruction word). This VLIW controls directly every single bit of processor HW like multiplexers, ALUs, memory control signals etc. In our current implementation VLIW length is ~130bit, while instruction opcode can vary from 8 to 32bit.



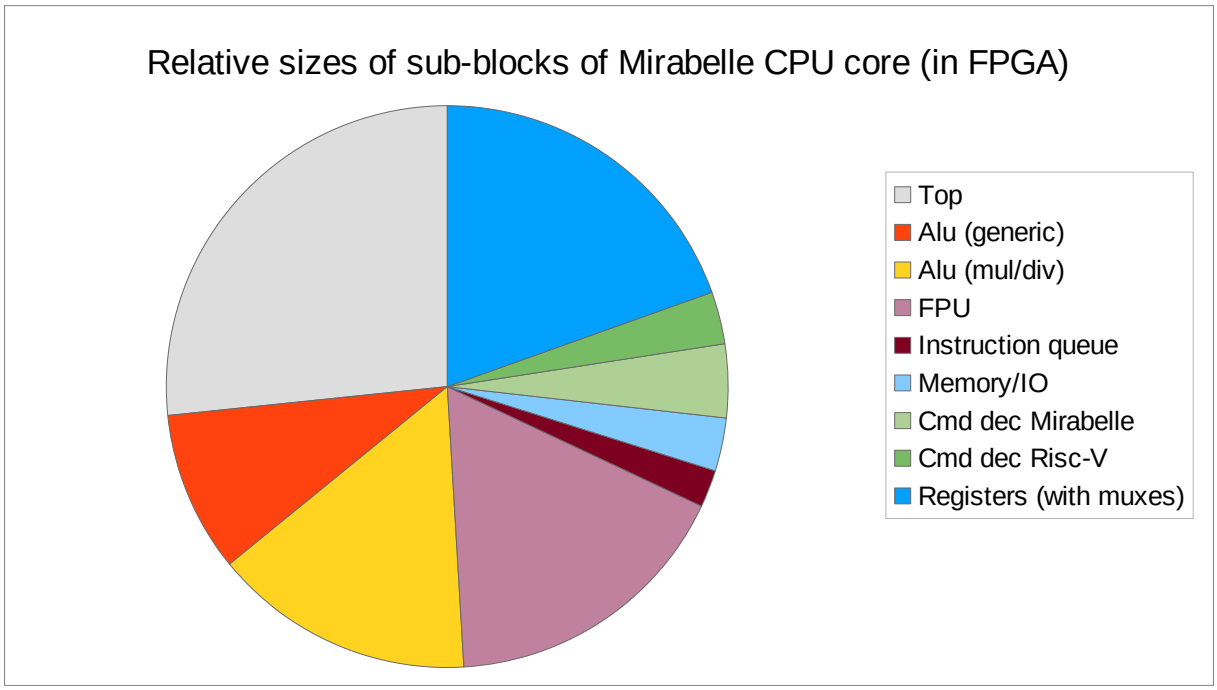
We propose to use several decoders (decompressors). Special bits will select which decoder to use. Those bits can be stored in the thread context and in our current implementation they are a part of IP register (IP = “Instruction Pointer”).

Remark
The idea comes from a bug. By mistake, instead of “Flags” register we mapped a higher part of IP to the multiplexer. We decided not to fix this bug but rather benefit from it.

Modified (simplified) CPU architecture looks like following.



Remark
The picture above is rather simplified. In practice there are several tricks which help using different decoders.



As we can see from the chart above, decoders (from ISA to VLIW) occupy ~3-4 % from the total core size.

Possible issues (technical)

It is supposed that the underlying HW architecture of base “universal” class CPU is flexible enough to fit other CPU architectures. It is relatively easy to host an “accumulator” CPU class but can be relatively difficult to host a “stack” class.

The additional decoder itself occupies some space also. But it is relatively small and occupies not more than 2-3% of a total CPU size. But the additional decoder does not consume power. We claim that by investing into the additional decoder we will harvest much more in Flash size saving.

Some architectures will be very difficult to map. For example if Base architecture supposes call stack growing up and mapped architecture to grow down, this will be very difficult to realize in practice. We want to limit our CPU to practical cases easy to implement.