ISA designers receive many requests for what are affectionately known as "features" - additional instructions that, in theory, will make the ISA better in some way.

Dealing with such requests is the moment to apply our quantitive approach in order to be able to judge the tradeoffs between cost and benefits.

Our first "feature request" is to allow small constants as the second operand in ALU instructions.

So if we replaced the 5-bit "rb" field, we would have room in the instruction to include a 16-bit constant as bits [15:0] of the instruction.

The argument in favor of this request is that small constants appear frequently in many programs and it would make programs shorter if we didn't have use load operations to read constant values from main memory.

The argument against the request is that we would need additional control and datapath logic to implement the feature, increasing the hardware cost and probably decreasing the performance.

So our strategy is to modify our benchmark programs to use the ISA augmented with this feature and measure the impact on a simulated execution.

Looking at the results, we find that there is compelling evidence that small constants are indeed very common as the second operands to many operations.

Note that we're not so much interested in simply looking at the code.

Instead we want to look at what instructions actually get executed while running the benchmark programs.

This will take into account that instructions executed during each iteration of a loop might get executed 1000's of times even though they only appear in the program once.

Looking at the results, we see that over half of the arithmetic instructions have a small constant as their second operand.

Comparisons involve small constants 80% of the time.

This probably reflects the fact that during execution comparisons are used in determining whether we've reached the end of a loop.

And small constants are often found in address calculations done by load and store operations.

Operations involving constant operands are clearly a common case, one well worth optimizing.

Adding support for small constant operands to the ISA resulted in programs that were measurably smaller and faster.

So: feature request approved!

Here we see the second of the two Beta instruction formats.

It's a modification of the first format where we've replaced the 5-bit "rb" field with a 16-bit field holding a constant in two's complement format.

This will allow us to represent constant operands in the range of 0x8000 (decimal -32768) to 0x7FFF (decimal 32767).

Here's an example of the add-constant (ADDC) instruction which adds the contents of R1 and the constant -3, writing the result into R3.

We can see that the second operand in the symbolic representation is now a constant (or, more generally, an expression that can evaluated to get a constant value).

One technical detail needs discussion: the instruction contains a 16-bit constant, but the datapath requires a 32-bit operand.

How does the datapath hardware go about converting from, say, the 16-bit representation of -3 to the 32-bit representation of -3?

Comparing the 16-bit and 32-bit representations for various constants, we see that if the 16-bit two's-complement constant is negative (i.e., its high-order bit is 1), the high sixteen bits of the equivalent 32-bit constant are all 1's.

And if the 16-bit constant is non-negative (i.e., its high-order bit is 0), the high sixteen bits of the 32-bit constant are all 0's.

Thus the operation the hardware needs to perform is "sign extension" where the sign-bit of the 16-bit constant is replicated sixteen times to form the high half of the 32-bit constant.

The low half of the 32-bit constant is simply the 16-bit constant from the instruction.

No additional logic gates will be needed to implement sign extension - we can do it all with wiring.

Here are the fourteen ALU instructions in their "with constant" form, showing the same instruction mnemonics but

with a "C" suffix indicate the second operand is a constant.

Since these are additional instructions, these have different opcodes than the original ALU instructions.

Finally, note that if we need a constant operand whose representation does NOT fit into 16 bits, then we have to store the constant as a 32-bit value in a main memory location and load it into a register for use just like we would any variable value.

To give some sense for the additional datapath hardware that will be needed, let's update our implementation sketch to add support for constants as the second ALU operand.

We don't have to add much hardware: just a multiplexer which selects either the "rb" register value or the sign-extended constant from the 16-bit field in the instruction.

The BSEL control signal that controls the multiplexer is 1 for the ALU-with-constant instructions and 0 for the regular ALU instructions.

We'll put the hardware implementation details aside for now and revisit them in a few lectures.