The Beta is an example of a reduced-instruction-set computer (RISC) architecture.

“Reduced” refers to the fact that in the Beta ISA, most instructions only access the internal registers for their operands and destination.

Memory values are loaded and stored using separate memory-access instructions, which implement only a simple address calculation.

These reductions lead to smaller, higher-performance hardware implementations and simpler compilers on the software side.

The ARM and MIPS ISAs are other examples of RISC architectures.

Intel’s x86 ISA is more complex.

There is a limited amount of storage inside of the CPU — using the language of sequential logic, we’ll refer to this as the CPU state.

There’s a 32-bit program counter (PC for short) that holds the address of the current instruction in main memory.

And there are thirty-two registers, numbered 0 through 31.

Each register holds a 32-bit value.

We’ll use use 5-bit fields in the instruction to specify the number of the register to be used an operand or destination.

As shorthand, we’ll refer to a register using the prefix “R” followed by its number, e.g., “R0” refers to the register selected by the 5-bit field 0b00000.

Register 31 (R31) is special — its value always reads as 0 and writes to R31 have no affect on its value.

The number of bits in each register and hence the number of bits supported by ALU operations is a fundamental parameter of the ISA.

The Beta is a 32-bit architecture.

Many modern computers are 64-bit architectures, meaning they have 64-bit registers and a 64-bit datapath.

Main memory is an array of 32-bit words.

Each word contains four 8-bit bytes.

The bytes are numbered 0 through 3, with byte 0 corresponding to the low-order 7 bits of the 32-bit value, and so on.

The Beta ISA only supports word accesses, either loading or storing full 32-bit words.

Most “real” computers also support accesses to bytes and half-words.

Even though the Beta only accesses full words, following a convention used by many ISAs it uses byte addresses.

Since there are 4 bytes in each word, consecutive words in memory have addresses that differ by 4.

So the first word in memory has address 0, the second word address 4, and so on.

You can see the addresses to left of each memory location in the diagram shown here.

Note that we’ll usually use hexadecimal notation when specifying addresses and other binary values — the “0x” prefix indicates when a number is in hex.

When drawing a memory diagram, we’ll follow the convention that addresses increase as you read from top to bottom.

The Beta ISA supports 32-bit byte addressing, so an address fits exactly into one 32-bit register or memory location.

The maximum memory size is 2^32 bytes or 2^30 words — that’s 4 gigabytes (4 GB) or one billion words of main memory.

Some Beta implementations might actually have a smaller main memory, i.e., one with fewer than 1 billion locations.

Why have separate registers and main memory?

Well, modern programs and datasets are very large, so we’ll want to have a large main memory to hold everything.

But large memories are slow and usually only support access to one location at a time, so they don’t make good storage for use in each instruction which needs to access several operands and store a result.

If we used only one large storage array, then an instruction would need to have three 32-bit addresses to specify

the two source operands and destination — each instruction encoding would be huge!

And the required memory accesses would have to be one-after-the-other, really slowing down instruction execution.

On the other hand, if we use registers to hold the operands and serve as the destination, we can design the register hardware for parallel access and make it very fast.

To keep the speed up we won't be able to have very many registers — a classic size-vs-speed performance tradeoff we see in digital systems all the time.

In the end, the tradeoff leading to the best performance is to have a small number of very fast registers used by most instructions and a large but slow main memory.

So that's what the BETA ISA does.

In general, all program data will reside in main memory.

Each variable used by the program "lives" in a specific main memory location and so has a specific memory address.

For example, in the diagram below, the value of variable "x" is stored in memory location 0x1008, and the value of "y" is stored in memory location 0x100C, and so on.

To perform a computation, e.g., to compute x*37 and store the result in y, we would have to first load the value of x into a register, say, R0.

Then we would have the datapath multiply the value in R0 by 37, storing the result back into R0.

Here we've assumed that the constant 37 is somehow available to the datapath and doesn't itself need to be loaded from memory.

Finally, we would write the updated value in R0 back into memory at the location for y.

Whew! A lot of steps…

Of course, we could avoid all the loading and storing if we chose to keep the values for x and y in registers.

Since there are only 32 registers, we can't do this for all of our variables, but maybe we could arrange to load x and y into registers,

do all the required computations involving x and y by referring to those registers, and then, when we're done, store changes to x and y back into memory for later use.

Optimizing performance by keeping often-used values in registers is a favorite trick of programmers and compiler writers.

So the basic program template is some loads to bring values into the registers, followed by computation, followed by any necessary stores.

ISAs that use this template are usually referred to as "load-store architectures".