MITOCW | MIT6_004S17_10-02-01_300k

In the previous lecture we developed the instruction set architecture for the Beta, the computer system we'll be building throughout this part of the course.

The Beta incorporates two types of storage or memory.

In the CPU datapath there are 32 general-purpose registers, which can be read to supply source operands for the ALU or written with the ALU result.

In the CPU's control logic there is a special-purpose register called the program counter, which contains the address of the memory location holding the next instruction to be executed.

The datapath and control logic are connected to a large main memory with a maximum capacity of 2^32 bytes, organized as 2^30 32-bit words.

This memory holds both data and instructions.

Beta instructions are 32-bit values comprised of various fields.

The 6-bit OPCODE field specifies the operation to be performed.

The 5-bit Ra, Rb, and Rc fields contain register numbers, specifying one of the 32 general-purpose registers.

There are two instruction formats: one specifying an opcode and three registers, the other specifying an opcode, two registers, and a 16-bit signed constant.

There three classes of instructions.

The ALU instructions perform an arithmetic or logic operation on two operands, producing a result that is stored in the destination register.

The operands are either two values from the general-purpose registers, or one register value and a constant.

The yellow highlighting indicates instructions that use the second instruction format.

The Load/Store instructions access main memory, either loading a value from main memory into a register, or storing a register value to main memory.

And, finally, there are branches and jumps whose execution may change the program counter and hence the address of the next instruction to be executed.

To program the Beta we'll need to load main memory with binary-encoded instructions.

Figuring out each encoding is clearly the job for a computer, so we'll create a simple programming language that will let us specify the opcode and operands for each instruction.

So instead of writing the binary at the top of slide, we'll write assembly language statements to specify instructions in symbolic form.

Of course we still have think about which registers to use for which values and write sequences of instructions for more complex operations.

By using a high-level language we can move up one more level abstraction and describe the computation we want in terms of variables and mathematical operations rather than registers and ALU functions.

In this lecture we'll describe the assembly language we'll use for programming the Beta.

And in the next lecture we'll figure out how to translate high-level languages, such as C, into assembly language.

The layer cake of abstractions gets taller yet: we could write an interpreter for say, Python, in C and then write our application programs in Python.

Nowadays, programmers often choose the programming language that's most suitable for expressing their computations, then, after perhaps many layers of translation, come up with a sequence of instructions that the Beta can actually execute.

Okay, back to assembly language, which we'll use to shield ourselves from the bit-level representations of instructions and from having to know the exact location of variables and instructions in memory.

A program called the "assembler" reads a text file containing the assembly language program and produces an array of 32-bit words that can be used to initialize main memory.

We'll learn the UASM assembly language, which is built into BSim, our simulator for the Beta ISA.

UASM is really just a fancy calculator!

It reads arithmetic expressions and evaluates them to produce 8-bit values, which it then adds sequentially to the array of bytes which will eventually be loaded into the Beta's memory.

UASM supports several useful language features that make it easier to write assembly language programs.

Symbols and labels let us give names to particular values and addresses.

And macros let us create shorthand notations for sequences of expressions that, when evaluated, will generate the binary representations for instructions and data.

Here's an example UASM source file.

Typically we write one UASM statement on each line and can use spaces, tabs, and newlines to make the source as readable as possible.

We've added some color coding to help in our explanation.

Comments (shown in green) allow us to add text annotations to the program.

Good comments will help remind you how your program works.

You really don't want to have figure out from scratch what a section of code does each time you need to modify or debug it!

There are two ways to add comments to the code.

"//" starts a comment, which then occupies the rest of the source line.

Any characters after "//" are ignored by the assembler, which will start processing statements again at the start of the next line in the source file.

You can also enclose comment text using the delimiters "/*" and "*/" and the assembler will ignore everything in-between.

Using this second type of comment, you can "comment-out" many lines of code by placing "/*" at the start and, many lines later, end the comment section with a "*/".

Symbols (shown in red) are symbolic names for constant values.

Symbols make the code easier to understand, e.g., we can use N as the name for an initial value for some computation, in this case the value 12.

Subsequent statements can refer to this value using the symbol N instead of entering the value 12 directly.

When reading the program, we'll know that N means this particular initial value.

So if later we want to change the initial value, we only have to change the definition of the symbol N rather than find all the 12's in our program and change them.

In fact some of the other appearances of 12 might not refer to this initial value and so to be sure we only changed the ones that did, we'd have to read and understand the whole program to make sure we only edited the right 12's.

You can imagine how error-prone that might be!

So using symbols is a practice you want to follow!

Note that all the register names are shown in red.

We'll define the symbols R0 through R31 to have the values 0 through 31.

Then we'll use those symbols to help us understand which instruction operands are intended to be registers, e.g., by writing R1, and which operands are numeric values, e.g., by writing the number 1.

We could just use numbers everywhere, but the code would be much harder to read and understand.

Labels (shown in yellow) are symbols whose value are the address of a particular location in the program.

Here, the label "loop" will be our name for the location of the MUL instruction in memory.

In the BNE at the end of the code, we use the label "loop" to specify the MUL instruction as the branch target.

So if R1 is non-zero, we want to branch back to the MUL instruction and start another iteration.

We'll use indentation for most UASM statements to make it easy to spot the labels defined by the program.

Indentation isn't required, it's just another habit assembly language programmers use to keep their programs readable.

We use macro invocations (shown in blue) when we want to write Beta instructions.

When the assembler encounters a macro, it "expands" the macro, replacing it with a string of text provided by in the macro's definition.

During expansion, the provided arguments are textually inserted into the expanded text at locations specified in the macro definition.

Think of a macro as shorthand for a longer text string we could have typed in.

We'll show how all this works in the next video segment.