The LNM Institute of Information Technology, Jaipur

Computer Organization and Architecture

TKGate Lab Project: ISA Design and CPU implementation

Part 1: ISA Design

The first part of this project requires you to design your own *instruction set architecture (ISA)*: a specification for format of machine code words and their meaning. You must define the size of each machine code instruction, how its bits are divided, what the values for each of those bits imply, and how the flow from one instruction to the next should progress.

Your ISA should be designed within the following constraints:

1. Word length: The machine word should be one byte (8 bits). That means that each register should hold one byte, and that each memory address should be one byte. Consequently, your main memory should $2^8 = 256$ *bytes* long.

2. Instruction length: A machine code instruction may consist of as many words as you like. I strongly recommend that it be a fixed number.

3. Registers: Your ISA must specify how many addressable registers are available for an instruction to use.
    - You may even choose no have no registers, where your instructions use only main memory addresses.
    - Alternatively, you may assume a register file with not exceeding 8 registers.
4. Capabilities: Your ISA must be capable of the following operations:

    - *Data movement:* You must provide instructions sufficient to set constants and move data.
        - A program must be able to load constants into either registers or main memory locations (or both);
        - It must also be able to copy values between registers and/or main memory locations.
        - Note that you may choose to provide a minimal set of instructions, and require the programmer to perform multiple steps for some kinds of data movement.
        - For example, to move from one main memory location to another, you may require the programmer to copy a value from the source main memory location into a register, and then copy it from the register to the destination main memory location. Such requirements save you the work of also creating an instruction that is capable of directly copying data from one main memory location to another.

- *Logical Operations:* Your ISA must allow for bitwise AND and OR operations on two input values, and NOT on a single input value. It may also provide other logic operations, such as XOR, NAND, NOR, etc.
- *Arithmetic Operations:* There must be instructions to perform *addition, subtraction (*using two-complement to represent negative integers*), multiplication* or *division* on two input values.
- *Unconditional branching:* There must be at least one type of *jump* instruction.
    - You may choose to use labels (i.e., immediate constants) as the jump targets. These constants may be expressed either as literal main memory locations or as offsets from the PC (depending on the size of your instructions).
    - The jump target could also be register based, where the branch target is taken from a register or main memory location.
    - Note that you do not need a procedure *call* instruction.
- *Conditional branching:* You must provide the ability to compare two values for equality or inequality, and to branch only if that condition is true. For example, you may choose to provide only comparisons to zero (*equal to zero*, *less than zero* or *negative*, *greater than zero* or *positive*). Any comparison between two values is possible when combined with simple arithmetic operations.

Part 2: Designing a CPU to implement your ISA

Once you have specified ISA, you have to build a CPU to carry out instructions. In particular, you must build a full datapath and control. Your design must include a register file (if your ISA assumes any addressable registers), a main memory, a PC, and an ALU. It should be possible to load the main memory with instructions of the format specified by your ISA, and then to have your CPU carry out the program specified by those instructions.

Design decisions

Instruction length: Multiple-word instructions can make the programming task easier, but that choice implies that your CPU will require multiple clock cycles to carry out each instruction. For each word of the n-word machine code instruction, at least $n-1$ cycles will be required to load the first $n-1$ words of the current instruction into temporary registers. At the earliest, on the n^{th} cycle the processor will have all of the bits of the instruction available to decode and execute. Thus, your control unit's inputs are no longer just the opcode, but also the cycle number for the processing of this instruction.

Unaddressable registers: Single-word instructions can be loaded from main memory and then immediately decoded and executed, avoiding the need (at least in terms of fetching the instruction) for multiple clock cycles per instruction. However, you may have noticed that a single word may have insufficient space to specify all of the registers, addresses, or immediate values that you need. For

example, if you want to add the values from two registers and store the resulting sum in a third register, there simply are not enough bits in one word for an opcode and three register numbers.

What to do? Create instructions that perform simpler tasks and that store those results in an intermediate register called an *accumulator*. So, for example, consider that to add the values in two source registers and then store the sum in a third register, you split the task across multiple instructions. One possibility is that you specify an add on the two source registers, but you don't specify a destination register. The result of the addition is placed into the accumulator so that the next instruction can copy the contents of the accumulator into the destination register.

Main memory access: Each main memory address is a one-word value that specifies a particular byte in the main memory. However, depending on how you have structured your ISA and imagined your CPU datapath and control, you may have to think carefully about how main memory is used.

For example, consider an instruction that copies a value from main memory and into a register. To perform this task, the instruction itself must be loaded from main memory, and then the value requested must be read from main memory as well. Since the main memory must be read twice for a single instruction, a multi-cycle control is necessary.

Another approach is to divide main memory into two components. The first component can be used to store only instructions, and the machine code of any program would have to be placed here. The second component can contain only data, and be accessed only by *load* and *store* instructions. This approach is less realistic than a real CPU using a typically unified main memory, but it is a reasonable approach to creating a working CPU.

Part 3: Testing your CPU

In order to determine whether your datapath and control work properly, you'll need to test it. Initially, you should hard-wire certain input patterns to your datapath to be sure that smaller components are responding properly. Once that's done, you need to write small programs to test the CPU's function overall. You should begin with extremely simple programs (e.g., add two numbers). Then move onto something slightly more complex (e.g., a simple loop).