CS 241 — Lecture 3

Bartosz Antczak *Instructor: Kevin Lanctot* *January 10, 2017*

Review of last lecture

Computers only understand binary (strings of ones and zeros). Humans program computers using high-level languages (such as Java, C++). To convert from high-level language to binary, we use an assembly language. This course will cover the MIPS assembly language. MIPS contains a set of instructions and registers that work with data. High-level code gets converted to these instructions. We've covered some of these instructions such as `add` or `jr`.

3.0.1 What's in a Computer?

- **RAM (Random Access Memory)**: stores data when the computer is on, it's essentially a massive array
- **Processor** manipulates data; it consists of two parts
  - *control unit*: controls the flow of data throughout the processor, RAM, and registers
  - *data path*: processes the data. The major components in it include:
    - Program Counter (PC): holds the address of the current instruction
    - Instruction Register (IR): holds the instruction that is being executed
    - Arithmetic Logic Unit (ALU): performs arithmetic calculations
    - General Purpose Registers: temporary storage within the processor (i.e., $4)

3.1 Conditional Execution in MIPS

In C++, we have

```
if ... else if ... else
while () { ... }
for () { ... }
```

But in MIPS, we have a different set of instructions for conditional statements and loops

- `beq`: branch if equal. The instruction is structured as `beq $s, $t, i`, which compares the contents of registers $s and $t. If they're equal, we skip `i` instructions. `i` can be positive or negative
- `bne`: branch if not equal. Similar to `beq`, except this instruction skips `i` instructions if the contents of the two registers are not equal
- `slt`: set if less than. Structured as `slt $d, $s, $t`, which compares register $s and $t, and if $s $<$ $t, then we set $d to 1; otherwise we set $d to 0
- `sltu`: similar to `slt`, except it compares the values in both registers as *unsigned*. Unsigned is another way of saying "natural numbers" (in CS 241, natural numbers start at 0. This means that unsigned numbers are only positive), whereas signed means "integers" (which means it includes negative numbers)

3.1.1 How the PC changes when we branch

Whenever we branch (using either `bne` or `beq`) the PC skips i instructions, and since each instruction is 4 bytes long, we add $i \times 4$ to the current PC. In addition to that, we increment the PC by 4 which happens each time an instruction gets executed. Thus, whenever we branch, the PC changes by $\underline{\text{PC} = \text{PC} + 4 + (i \times 4)}$

3.2 Memory Access in MIPS

The maximum size of memory in a 32-bit architecture is 2^{32} bytes = 4 GB. It's acceptable to think of RAM as one big array, which stores data in a distinct location in memory called an address. To access memory, we can access any of the 2^{32} bytes directly; however, it's more convenient to access any of the 2^{30} words directly, and that's how MIPS accesses addresses. This is why each address in MIPS is divisible by 4. We access this memory using two instructions:

- `lw`: load word. Structured as `lw $t, i($s)`, which loads a word from memory address ($s + i)* into $t
- `sw`: store word. Structured as `sw $t, i($s)`, which stores a word from $t into memory address ($s + i)*

* = address must be divisible by 4

3.3 More Arithmetic Operations in MIPS

We can also multiply and divide values in MIPS. These operations use special registers *hi* and *lo*:

- `mult $s, $t`: multiplies the contents of registers $s and $t. Since our answer can be larger than 32 bits, we place the most significant 32 bits in *hi* and the least significant 32 bits in *lo*
- `div $s, $t`: divides $s by $t. Places the result in *lo* and the remainder in *hi*

These instructions also work with unsigned integers; `multu` and `divu` (these instructions take in the same arguments as their respective "signed" counterparts).

Accessing hi and lo

To access the hi register, use instruction `mfhi $d` to copy the contents into register $d.
To access the lo register, use instruction `mflo $d` to copy the contents into register $d.