MIPS Architecture

- Each MIPS instruction is a 32-bit (4-byte) quantity.
- MIPS is a load/store architecture. Most instructions require their operands and results to be in registers. There are two exceptions:
  1. Load instructions: copy data from memory into a register.
  2. Store instructions: copy data from a register into memory.

MIPS Architecture ...

- In general, RISC processors are load/store architectures, and CISC processors are not. CISC processors allow any or all of the operands and results of an instruction to be in memory.
- What implications does this have?
- RISC = Reduced Instructions Set Computer
- CISC = Complex Instructions Set Computer

Programming in Machine Code

- The topic on computer organization was a bit vague on the details. An instruction was represented as a sequence of four numbers: an operation, two operand registers, and a result register. The last topic showed us how to represent numbers. Now we'll learn how to package them together into instructions.
- Machine code are the numbers that represent computer instructions. We'll look at MIPS machine code in particular.
- The set of valid instructions a given computer is called its instruction set.
Some computers have non-orthogonal instruction sets; certain operations can only be done on certain registers, e.g. multiplication reads its operands from specific registers and writes the result to specific registers. Non-orthogonal instruction sets are typically used because of hardware constraints; they aren’t pleasant to program.

- Memory accesses must be properly aligned according to size. Words must be aligned on word boundaries (i.e. the address must be divisible by 4), half-words on half-word boundaries, etc. This includes instructions, which must be aligned on word boundaries.
- MIPS instructions are three-address, meaning an instruction can have up to two operand registers and a destination register. The same register may be used for any or all addresses. Note the confusing use of the term "address", which assumes the CPU is a CISC.

MIPS caveats:
1. Register 0 always contains value 0. You can’t change it. It’s a good place to get a 0 if you need one, and to store the result of an operation if you don’t want it.
2. Register 31 holds the return address for procedure calls (we’ll get to this later).
3. There are software conventions for using the other registers. This means that the hardware doesn’t care what you do with them, but other programmers do. We’ll also cover this later.

- The MIPS has 32 general-purpose registers, each containing 32 bits.
- General purpose registers can be used to hold data values or addresses, and you can perform any operation on them. There are also special-purpose MIPS registers that are used for special operations.
- The MIPS instruction set is relatively orthogonal, meaning you can mix-and-match general-purpose registers and operations; there are no restrictions.
• Register rs (r0) is added to the value in field imm16. imm16 contains a 16-bit immediate value – the value of the number in those 16 bits of the instruction is used as one of the operands. In this instruction imm16 is 0x1000.
• The instruction computes r1 = r0 + 0x1000, which is the same as r1 = 0x1000.

• Now do the same for the second instruction.
  000000|00000|00001|00001|00001|00000
  op  rs  rt  rd  shamt  func
  op = 0. The func field to determine the operation. func = 0 which is shift left logical. This instruction shifts the value in register rt (r1) by the amount in the field shamt (1), and stores the result in register rd (r1).
• Group the bits into fields according to the ”MIPS Instruction Encodings” document:
  001001|00000|00001|000100000000000
  op  rs  rt  imm16
  • The op is the operation to be performed. This is often called the opcode (operation code). In this case the opcode is 0x9, which corresponds to addition immediate without overflow. The ”without overflow” means that the values are treated as unsigned binary numbers.
  • The result is put in register rt (r1)
Assembly Code

- An assembler is a program that takes a textual representation of a program, and turns them into machine code. For example, we could write the above program as:

  \[
  \text{addiu r1, r0, 0x1000} \\
  \text{sll r1, r1, 1} \\
  \text{bne r0, r1, -1}
  \]

- This is much easier to understand than machine code (although still a bit cryptic), and can be converted into machine code by a simple assembler.

Sample MIPS machine code ...

- And the last instruction:

  \[
  000101|00000|00001|1111111111111111 \\
  \text{op rs rt offset}
  \]

- \( \text{op} = 5 \), which is branch on not equal. This instruction changes the PC (program counter) by the number of instructions stored in offset (0xFFFF = -1) if the value in rs (r0) does not equal the value in rt (r1). In other words, the PC moves back one instruction if r1 is not zero.

Readings and References

- Read Maccabe, section 3.1.4, pp. 80–85.

Sample MIPS machine code ...

- In pseudo-code:

  \[
  \text{r1} = 0x1000 \\
  \text{do} \\
  \text{\hspace{1em} r1 = r1 \ll 1} \\
  \text{\hspace{2em} \textbf{while} (r1 != 0)}
  \]

  where "\( \ll \)" means shift left logical, and "\(!=\)" means "does not equal".

- Programming in machine code can be done, but it's (obviously) not pleasant. We'd like to use words, not bits, to describe what's going on. The pseudo-code is much more understandable than the machine code.