CMPEN 331 Exam 1 Review

Will Bochnowicz

February 23, 2023

Contents

1 Computer Basics

1.1 Seven Great Ideas

There are seven great ideas in computer design that we base our understanding and processor development methodology around.

1. Abstraction:

   We don't always want to work with the lowest level of the computer. We like to *abstract* away some of the peculiarities and concerns of the lower-level hardware to enhance understandability. How else could soydevs code in dynamically typed, garbage collected Esperanto?

2. Make the Common Case Fast:

   Amdahl's Law – The amount of performance increase that a single improvement has on overall performance depends on how much the affected component is used proportional to the rest of the system. Essentially, the best way to make a system faster is to make its most-used components/functions faster, as that is what's used most of the time.

3. Performance via Parallelism:

   If our computer can do multiple things at once, then it can do more things overall! This is the idea of parallel processors inside a CPU. This extremely powerful concept requires that code be rewritten in a parallel manner, which can be troublesome.

4. Performance via Pipelining:

   By designing better processes, we can make computer operations run faster and more efficiently. This applies to both software and hardware. For example, we can design hardware for floating-point addition and uses more adders to make the process of addition faster.

5. Performance via Prediction:

   Modern processors perform magic when considering conditionals, starting the process of which branch it is most likely to take before the result of the conditional has actually been evaluated. If we can prepare for a situation before it even arises, our system will be faster.

   This idea hinges on the predictions not being too computationally severe, so that an incorrect prediction doesn't have too severe consequences.

6. Hierarchy of Memories:

   By having a multi-leveled memory system (registers, cache, main memory, secondary memory), we can optimize for speed vs space based on what our needs are.

7. Dependability via Redundancy:

   Building fail-safes into hardware and software can help prevent costly downtime.

Some other sources include Moore's Law as a great idea in computer architecture – the prediction that the number of transistors in an IC will double every 18-24 months. However, this is not a design principle as much as an observation. What this translates to for practicality's sake is that designers have to try to predict where the market will be at in several years when designing the early stages of a product in order to be competitive with the market's advancement during the development time of the processor.

2 Memory

2.1 Memory Layout

Memory is where a computer stores information. To be accessible, each memory location must have an address. The memory space is 2^k locations wide, which is called the address space. For example, a computer with 32 address bits can have at most 2^{32}bits $= 4G$ of memory. Our computers are byte addressable, where each byte location (1, 2, 3, ...) refers to a group of 8 bits in memory that are treated as a single unit. Because our computers have 32 bit words, word locations are at 0, 4, 8, ... We like to align words to make them easier to access – each word starts at a byte that is a multiple of the number of bytes in a word.

MIPS is Big-Endian: This means that words are arranged such that lower addresses are the more significant bytes of a word. For example, 0xDEADBEEF, in a big-endian system, would be stored as DE AD BE EF in a single word, while a Little-Endian system could store this word as EF BE AD DE.

There are two memory operations – read and write.

A program's memory is laid out as a stack – lower addresses (around 0) are reserved, then there are `Text`, `Static Data`, `Dynamic Data` (Heap), and `Stack` areas in memory. The Stack is located at the highest memory address (`0xffffffff`) and grows down, while the heap is located lower than the stack and grows up. These two regions can collide, causing memory errors, is either is too large.

3 MIPS Instructions

Assembly is the language of the computer. These are basic operations that are easily converted (by the assembler) into machine code: the 1s and 0s that computers actually use to compute numbers. It is one of a number of Instruction Set Architectures, formal definitions of what a processor can and can't do, which have slightly varying Instruction Sets per processor model. For example, ARMv7-m is an Instruction Set Architecture, upon which the Cortex-M processors have conforming Instruction Sets despite having slightly different implementations.

MIPS is a family of Instruction Set Architectures. They are RISC based, which means that they are relatively restricted in size and can only do operations on the contents of registers. The contents of memory must be moved into a register before it can be acted upon (load/store architecture). There are a number of ways that MIPS instructions, which are actual commands to the processor, are represented in binary. For the sake of this course, we only consider 32-bit MIPS architectures, which means that instructions are only 32 bits long.

Sidenote: Other instruction set paradigms, such as Complex Instruction Set Computers (CISC) exist. CISC can have multi-word instructions and allow operands directly from memory.

A register is a small storage medium that is located directly in the CPU. It is the fastest form of memory, but has extremely limited size due to its location within the CPU. MIPS has 32 general purpose registers, as well as some others (LO, HI, PC, floating point registers) that serve specific purposes.

Some MIPS instructions are so-called "pseudoinstructions", as they are not actual instructions in the MIPS ISA. Rather, they are shorthand for ease of programming and are expanded into actual instructions by the assembler. For example, the `blt $s1, $s2, Label` instruction actually expands to two instructions: `slt $at, $s1, $s2` and `bne $at, $zero, Label`, where `$at` is a

register that is reserved for the assembler to do operations of this sort (side-note: `$at` is one of the 32 general-purpose registers, despite being reserved by the assembler).

3.1 R-Type

R-Type instructions represent instructions that act on three registers. These registers are labeled as the Source, Target, and Destination

Examples of R-Type instructions: `add`, `sub`, `mul`

op	rs	rt	rd	shamt	func
6	5	5	5	5	6

op: "Operation" – The basic operation of the instruction, also called "opcode". 6 bits (just happened to be leftover from dividing 32 bits between 4 5-bit fields and 2 other fields).

rs: "Resister Source" – the first operand of the instruction. 5 bits, as there are 32 general registers.

rt: "Register Target" – the second operand of the instruction. This can occasionally serve as the destination in other types of operations. 5 bits, as there are 32 general registers.

rd: "Register Destination" – the third operand of the instruction. This is where the result of the operation is stored. 5 bits, as there are 32 general registers.

shamt: "Shift Amount" – The amount that the target register is shifted, used for sll and srl operations (where the source register is unused). This field is zero for most arithmetic operations. 5 bits, in order to access all 32 bits in a word.

func: "Function Code" – Specifies the variant of the Operation that is used. For most R-Type, this contains the entire operation ("op" is zero for add, sub, mult, div, etc). 6 bits.

3.1.1 Problems

Q :Write the binary representation of the `add` command.

Note that `add`'s `op` and `func` fields are 000000 and 100000, respectively. (1)

Also assume that for $a = b + c$, a is in register 1, b is in register 2, and c is in register 3.

3.2 I-Type

I-Type instructions are instructions that work on immediates, which are constants in memory. These instructions have extra space to store the binary representation of a number directly in their instructions, rather than in a source, target, or destination register.

Instructions of I-Type instructions are all Load and Store instructions (lw/sw: `op base rt offset`), `addi` and `subi`, and Branch instructions (beq/bne: `op rs rt offset`, where offset is where you jump relative to PC)

op	rs	rt	immediate
6	5	5	16

op: "Operation" – The basic operation of the instruction, also called "opcode". 6 bits.

rs: "Resister Source" – The first operand of the instruction. 5 bits, as there are 32 general registers.

rt: "Register Target" – The second operand of the instruction. This can occasionally serve as the destination in other types of operations. 5 bits, as there are 32 general registers.

immediate: "Immediate" – The constant immediate value that is being used in the operation. 16 bits can store up to the value 65,536.

3.2.1 Problems

Q :Write the MIPS assembly instruction for the following I-Type binary instruction:

00010101000001001101010101010101010 (2)

You can look up a table of MIPS commands in binary and the addresses of registers.

3.3 J-Type

J-Type instructions are used exclusively for jump instructions. They feature the most space of any instruction for immediates, which are used to represent the address that is being jumped to. While only 26 bits, these bits can be expanded to the full 32 bit word size by concatenating the highest 4 bits of PC to the target address (somewhat limiting the range of this jump), and concatenating 00 to the end of the instruction, as instructions are always aligned on 32-bit words.

op	target address
6	26

op: "Operation" – The basic operation of the instruction, also called "opcode". 6 bits.

target address: "Target Address" – An immediate that represents the instruction number that is to be jumped to.

3.3.1 Problems

3.4 Logical Operators

The logical operators are a class of instructions that compare the binary values within registers to each other.

Command Type	Examples
Shift Left	`sll $t0, $t1, 4` shifts the bits of t1 left by 4 and stores in t0
Shift Right	`srl $t0, $t1, 4` shifts the bits of t1 right by 4 and stores in t0
Bitwise AND	`and $t0, $t1, $t2` is `t0 = t1 & t2`; `andi $t0, $t1, 27` is `t0 = t1 & 27`
Bitwise OR	`or $t0, $t1, $t2` is `t0 = t1 | t2`; `ori $t0, $t1, 27` is `t0 = t1 | 27`
Bitwise NOR	`nor $t0, $t1, $t2` is `t0 =` $\sim$`(t1 | t2)`
Bitwise NOT	`nor $t0, $t1, $zero` is `t0 =` $\sim$`(t1 | 0) = t1`

4 Addressing Modes

There are multiple ways that memory is addressed for various operations in MIPS assembly.

4.1 Register Addressing

Register Addressing is a form of direct addressing – the value inside a register is used as a memory address. The `jr` and `jalr` use this form of addressing.

Algebraic operations, such as `add`, `sub`, etc, all use this method of addressing.

4.2 Immediate Addressing

An immediate value that is inserted directly into the instruction is used as a memory address. Limited to 16 bits, as that is the size of the Immediate field in I-Type instructions. The `addi`

and related commands use immediate addressing, as the value of the immediate is baked into the command itself.

4.3 Base Addressing

Base addressing is where you reference a location in memory based on an offset from a memory location that is stored in a register. Examples of this include the `lw` command. Once again, due to being an I-Type instruction, the offset values are limited to 16 bits.

4.4 PC-Relative Addressing

PC-Relative Addressing is used in the `beq` and `bne` instructions. In this mode, the memory location of data or an instruction is specified using an offset from the current PC value. Once again, this type of addressing is used for I-Type instructions where the offset is stored as an immediate constant.

One advantage of this method is that it works regardless of where the code is located in memory. While some other addressing modes can produce code that point directly to a specific point in memory, that form of addressing depends on those instructions being located in that specific memory location. PC-Relative Addressing creates machine code that acts independently of what specific location in memory the instructions are stored – the offsets are always the same between instructions because word size is known (can be implemented at compile time), but exact locations in memory are only known at link time.

4.5 Jump Addressing / Pseudo-direct Addressing

Using J-Type instructions, this type of addressing is only used for `j` and `jal` commands. Expansion to a full address is done by concatenating the highest 4 bits of PC to the 26 bits of the target address (somewhat limiting the range of this jump), then concatenating 00 to the end of the instruction, as instructions are always aligned on 32-bit words. (Copied from J-Type explanation above).

The limitations of this addressing type come from using the upper 4 bits of PC – this limits the jump to the current 256MB of code, which is only $\frac{1}{16}$ of the total 4GB address space. To solve this issue, use the Register Addressing `jr` and `jalr` commands. Because these use full 32-bit register values, the entire address space can be jumped to with these commands.

As stated in section 4.4, the addresses used for jump addressing are computed during the linking phase of machine code generation, which can be slower than other addressing methods.

5 Solutions

1: The binary for this operation is 00000000010000110000100000100000.
OP = 000000
RS = 00010
RT = 00011
RD = 00001
SHAMT = 00000
FUNC = 100000

2: This translates to the instruction `bne $t0 $t1 0xaaaa`.
BNE = 000101

$t0 = 01000
$t1 = 01001
$0xaaaa = 1010101010101010$

Index