INSTRUCTION/DATA ENCODING

TEAM MEMBERS:

- MOTTAM SHIVA TEJA
- BHUPATI RAHUL
- MEGAVATH PREM KUMAR
- KUNSOTH PUNNAMCHAND NAIK
- J N SAI TEJA
- KAMMARI SRI RAM CHARI
- RAPAKA VINEETH KUMAR

Abstract:

We have introduced a variety of useful instructions and addressing modes. These instructions specify the actions that must be performed by the processor circuitry to carry out the desired tasks. We have often referred to them as machine instructions. Actually, the form in which we have presented the instructions is indicative of the form used in assembly languages, except that we tried to avoid using acronyms for the various operations, which are awkward to memorize and are likely to be specific to a particular commercial processor. To be executed in a processor, an instruction must be encoded in a compact binary pattern. Such encoded instructions are properly referred to as machine instructions. The instructions that use symbolic names and acronyms are called assemblylanguage instructions, which are converted into the machine instructions using the assembler program.

We have seen instructions that perform operations such as add, subtract, move, shift, rotate, and branch. These instructions may use operands of different sizes, such as 32-bit and 8-bit numbers or 8-bit ASCII-encoded characters. The type of operation that is to be performed and the type of operands used may be specified using an encoded binary pattern referred to as the OP code for the given instruction. Suppose that 8 bits are allocated for this purpose, giving 256 possibilities for specifying different instructions. This leaves 24 bits to specify the rest of the required information.

DESCRIPTION

Assembly language is the symbolic representation of a computer's binary encoding—machine language. Assembly language is more readable than machine language because it uses symbols instead of bits. The symbols in assembly language name commonly occurring bit

patterns, such as opcodes and register specifiers, so people can read and remember them. In addition, assembly language permits programmers to use labels to identify and name particular memory words that hold instructions or data.

Introduction

- *Remember that in a stored program computer, instructions are stored in memory (just like data)*
- *Each instruction is fetched (according to the address specified in the PC), decoded, and exectuted by the CPU*
- *The ISA defines the format of an instruction (syntax) and its meaning (semantics)*
- *An ISA will define a number of different instruction formats.*
- *Each format has different fields*
- *The OPCODE field says what the instruction does (e.g. ADD)*
- *The OPERAND field(s) say where to find inputs and outputs of the instruction.*

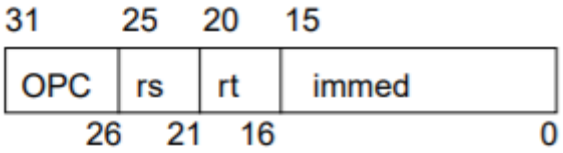
MIPS Encoding

- *The nice thing about MIPS (and other RISC machines) is that it has very few instruction formats (basically just 3)*
- *All instructions are the same size (32 bits = 1 word)*
- *The formats are consistent with each other (i.e. the OPCODE field is always in the same place, etc.)*
- *The three formats:*
  - *I-type (immediate)*
  - *R-type (register)*
  - *J-type (jump)*

I-type (immediate) Format

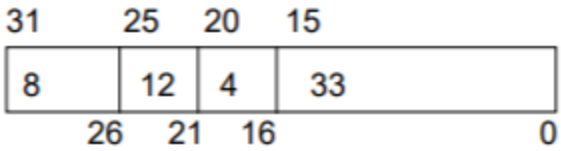
- *An immediate instruction has the form:XXXI rt, rs, immed*

- *Recall that we have 32 registers, so we need ??? bits each to specify the rt and rs registers*
  - *We allow 6 bits for the opcode (this implies a maximum of ??? opcodes, but there are actually more, see later)*
- *This leaves 16 bits for the immediate field.*



I-type Example

- *Example: ADDI $ a0, $ 12, 33 # a0 <- r12 + 33*
- *The ADDI opcode is 8, register a0 is register # 4.*



Load-Store Formats

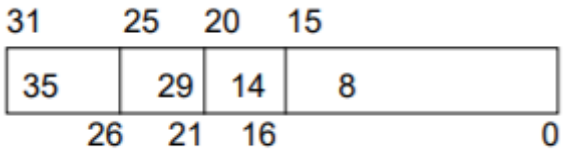
- *A memory address is 32 bits, so it cannot be directly encoded in an instruction.*

- *Recall the use of a base register + offset (16-bits) in the load-store instructions.*

  - *Thus, we need an OPCODE, a destination/source register (destination for load, source for store), a base register, and an offset.*

  - *This sounds very similar to the I-type format... example: LW $ 14, 8($ sp) # r14 is loaded from stack+8*

- *The LW opcode is 35 (0x23)*



R-type (register) format

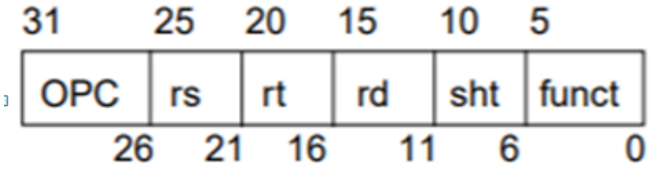
- *General form: XXX rd, rt, rs*

  - *Arithmetic-logical and comparison instructions require the encoding of 3 registers, the rest can be used to specify the OPCODE.*

  - *To keep the format as regular as possible, the OPCODE has a primary "opcode" and a "function" field.*

- *We also need 5 bits for the shift-amount, in case of SHIFT instructions.*

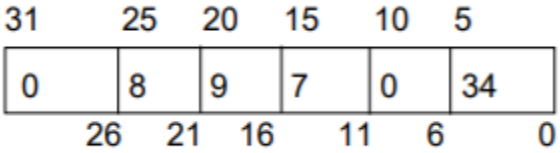
    - *The 16 bits used for the immediate field in the I-type instruction are split into 5 bits for rd, 5 bits for shift-amount, and 6 bits for function (the other fields are the same).*



R-type Example

SUB $ 7, $ 8, $ 9 # r7 <- r8 - r9

- *The opcode for all R-type instructions is zero, the function code for SUB is 34, the shift amount is zero.*

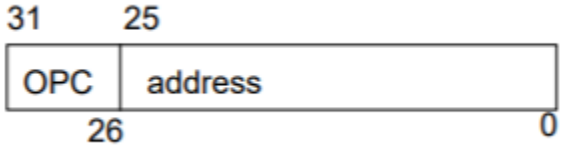


J-type (Jump) Format

- *For a jump, we only need to specify the opcode, and we can use the other bits for an address*

- *We only have 26 bits for the address, but MIPS addresses are 32 bits long...*

    - *Because the address must reference an instruction, which is a word address, we can shift the address left by 2 bits (giving us 28 bits). We get the other 4 bits by combining with the 4 high-order bits of the PC.*



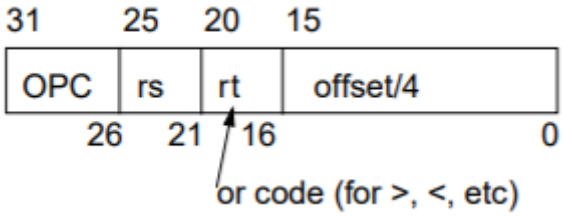
Branch Addressing

- *There are 2 kinds of branches:*

1. EQ/NEQ family (compares 2 regs for (in)equality), example: BEQ $ 14, $ 8, 1000

2. Compare-to-zero family (compares 1 reg to zero), example: BGEZ $ 14, 1000

- *Both "families" require OPCODE, rs register, and offset*
  - *(1.) requires an additional register (rt)*
  - *(2.) requires some encoding for (>=, <=,)*

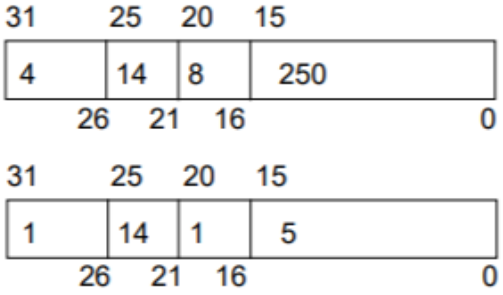


Branch example

BEQ $ 14, $ 8, 1000 # PC := PC+1000 if r14==r8

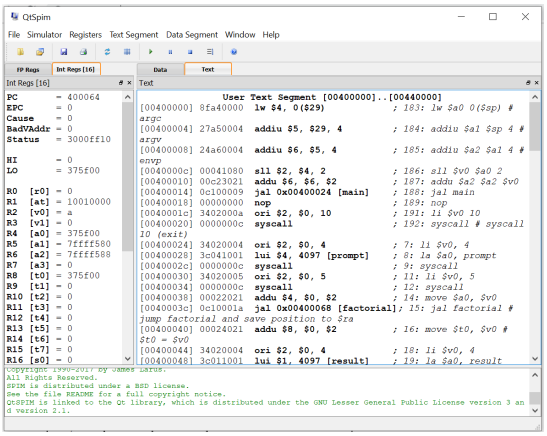
BGEZ $ 14, 20 # PC := PC+20 if r14 >= 0

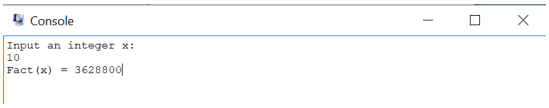
- *The opcode for BEQ is 4; for BGEZ is 1, the code for >= is 1*



EXAMPLES

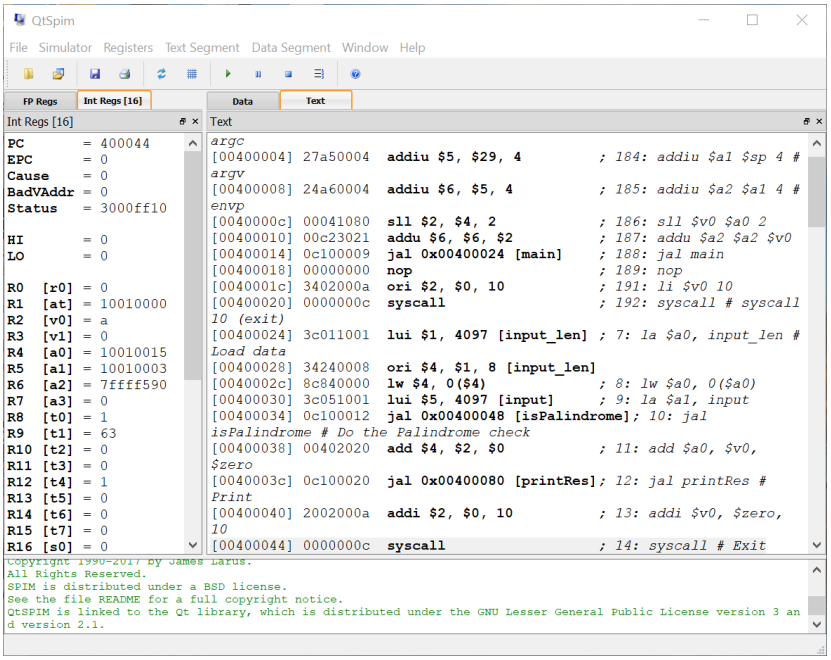
Assembly Program to find factorial of a number:





For the instruction ***lw $ 4, 0($ 29)*** *the machine code is* ***0x****8fa40000*

Assembly Program to check palindrome



For the instruction ***addiu $ 5, $ 29, 4*** *the machine code is* ***0x****27a50004*

REFERENCES

- *https://scholar.google.co.in/scholar?hl=en& as_ sdt=0% 2C5& q=assemblers+linkers& btnG=# d=gs_ qabs& u=% 23p% 3Dbow7UUi2IkAJ*
- *http://web.cs.iastate.edu/∼ cs321/pa1.html# lod*
- *https://medium.com/coderscorner/machine-code-dd8fcbe3153*

WHY THE TOPIC IS INTERESTING??

While googling for the topic, we actually found the topic interesting because of the vital uses and facets of assembly language. On further research on the topic in web we found that the codes get executed at a faster rate when written in assembly code than in higher level languages like C,JAVA etc as it is directly converted to the machine code or the object file which the computer understands. So the interest started from there which helped us to start learning the assembly language. Then an other question popped in our mind that is how is the assembly code converted to machine code by the assembler. This developed our interest towards data/instruction encoding.

This led us to further learn about MIPS encoding and all other stuff like number of bits that are required to actually implement any simple instruction , how the instructions are stored in the CPU registers etc. We also learnt about the different types of registers available in the processor and their specific functions like the v-type, a-type, t-type, s-type etc.

In the process of learning all these new stuff the interest on the topic automatically developed thus leading us to do this project.

RESULT AND CONCLUSION:

Programming in assembly language requires a programmer to trade off helpful features of high-level languages—such as data structures, type checking, and control constructs—for complete control over the instructions that a computer executes. External constraints on some applications, such as response time or program size, require a programmer to pay close attention to every instruction. However, the cost of this level of attention is assembly language programs that are longer, more time-consuming to write, and more difficult to maintain than high-level language programs.

Moreover, three trends are reducing the need to write programs in assembly language. The first trend is toward the improvement of compilers. Modern compilers produce code that is typically comparable to the best handwritten code—and is sometimes better. The second trend is the introduction of new processors that are not only faster, but in the case of processors that execute multiple instructions simultaneously, also more difficult to program by hand. In addition, the rapid evolution of the modern computer favors high-level language programs that are not tied to a single architecture. Finally, we witness a trend toward increasingly complex applications—characterized by complex graphic interfaces and many more features than their predecessors. Large applications are written by teams of programmers and require the modularity and semantic checking features provided by high-level languages.

rapid evolution of the modern computer favors high-level language programs that are not tied to a single architecture. Finally, we witness a trend toward increasingly complex applications—characterized by complex graphic interfaces and many more features than their predecessors. Large applications are written by teams of programmers and require the modularity and semantic checking features provided by high-level languages.