CSE220 – Architectures & MIPS

MIPS – Assembly Language

- Stands for “Microprocessor without Interlocked Pipeline Stages”
- It is an **Instruction Set Architecture** used primarily in embedded systems
  - PlayStation 1 & 2, Nintend64, & PSP
  - digital cameras, DVD players, routers, wireless phones, HDTVs, etc.
- Frequently used for Educational purposes due to its simplicity and that it is still widely used

- To familiarize ourselves with what MIPS looks like, consider a basic java hello world program
  
  ```java
  public class HelloWorld {
    public static void main(String[] args) {
      // Prints "Hello, World" to the terminal window.
      System.out.println("Hello, World");
    }
  }
  ```

- In MIPS, the same program would look like:

  ```assembly
  # text segment #
  .text             # tells assembler program code starts here
  .globl main       # defines label for execution start

  main:             # label for where execution starts
    la $a0,str    # put address of string into argument register
    li $v0,4      #
    syscall      # system call to print a string to terminal
    li $v0, 10    # Load exit syscall value
    syscall      # Exit

  # data segment #
  .data            # tells assembler data segment begins here
  str: .asciiz "hello world\n" # declaration of a NULL terminated string
  ```

- This format is **assembly language**. It uses **mnemonics** to represent the instructions. It is easier to read than its binary equivalent.
- This is the format in which you will programming in this semester.

- Each line of assembly code is translated into their binary representations by the **assembler**. The assembler converts the labels, instructions, register, etc into the appropriate binary. The binary encoded is called the **machine language**.

- As we will see throughout the semester, the hardware only understands machine language, the 0’s and 1’s.

- **Side Note:** x86 is used in most modern products, however we do not study x86 architectures as they are much more complicated in nature than MIPS. If you learn the concepts/foundation and architectural principles in MIPS, you can similarly apply and extend the ideas when trying to understand x86 architectures.

- **Fun fact:** MIPS was invented by John Hennessy, the former president of Stanford University and alumnus of Stony Brook’s MS and PhD programs in Computer Science

**MIPS Programming 101**
• Take a look at the MIPS code for Hello world in more detail

• **Comments** – Any characters on a line which appear after a ‘#’ symbol are comments.

  ```
  # text segment #
  .text # tells assembler program code starts here
  .globl main # defines label for execution start

  main: # label for where execution starts
    la $a0,str # put address of string into argument register
    li $v0,4 #
    syscall # system call to print a string to terminal
    li $v0, 10 # load exit syscall value
    syscall # exit
  
  # data segment #
  .data # tells assembler data segment begins here
  str: .asciiz "hello world\n" # declaration of a NULL terminated string
  ```

• **Labels** – Lines starting with `text_name:` specify mnemonic names to refer to particular memory addresses
  
  o Labels are used to reference a specific instruction, for the start of a method, to global variables, or to the start of a block of code
  
  o Labels can contain characters, digits, dot, or underscore characters and are followed by a colon.

  They cannot begin with a number.

  o **EX:** main: and str:

• **Assembler Directives** – Lines that start with ‘.’ are assembler directives

  o `.text`: everything below this is source code

  o `.globl label`: makes the label available externally. (In MARS we can set the program to begin execution from this label. More about the MARS simulator in recitation.)

  o `.data`: specifies the area where global variables are to be declared

  o `.asciiz`: for declaring a NULL terminated string

  Other directives:

  o `.ascii`: an ASCII string, not terminated by NULL

  o `.word`: allocates space for one or more 32-bit words

  o `.byte`: allocates space for one or more bytes

• **Instruction Names** – Each instruction in the instruction set has a mnemonic abbreviation for the programmer to use.

  o **EX:** `la` (load address), `li` (load immediate)

• **Registers** – Registers are local storage within the CPU (for fast access) where information is stored

  o `$` before name

  o Example: `$zero` or `$0` can be used to refer to “register zero”

  o Registers are used for specific purposes:

    - **$0** always holds the constant value 0.

    - The **saved registers**, `$s0-$s7`, are used to hold variables

    - The **temporary registers**, `$t0-$t9`, are used to hold intermediate values during a larger computation

  o MIPS has thirty-two 32-bit registers

    - Faster than main memory, but much smaller in amount of total space
- Reading data from a small set of registers is faster than from a larger one (simpler circuitry)
  - MIPS is a 32-bit architecture because it operates on 32-bit data. Each register holds 32-bits.

<table>
<thead>
<tr>
<th>Name</th>
<th>Register #</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>the constant value 0</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>assembler temporary</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>function return values</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>function arguments</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>temporaries</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved variables</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>more temporaries</td>
</tr>
<tr>
<td>$k0-$k1</td>
<td>26-27</td>
<td>OS temporaries</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>function return address</td>
</tr>
</tbody>
</table>

- When programming in MIPS you must follow certain register conventions (rules) in order for your assembly code to work properly in all scenarios. We will discuss these in more detail later.

- Syscall – a request made by the program to the OS to perform a service (read input, print output, quit the program)
  - Each syscall number has a purpose. The Code is loaded into $v0 to indicate the operation to be performed.

<table>
<thead>
<tr>
<th>Service</th>
<th>System Call Code</th>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_int</td>
<td>1</td>
<td>$a0=integer</td>
<td></td>
</tr>
<tr>
<td>print_float</td>
<td>2</td>
<td>$f12=float</td>
<td></td>
</tr>
<tr>
<td>print_double</td>
<td>3</td>
<td>$f12=double</td>
<td></td>
</tr>
<tr>
<td>print_string</td>
<td>4</td>
<td>$a0=string</td>
<td></td>
</tr>
<tr>
<td>read_int</td>
<td>5</td>
<td></td>
<td>integer (in $v0)</td>
</tr>
<tr>
<td>read_float</td>
<td>6</td>
<td></td>
<td>float (in $f0)</td>
</tr>
<tr>
<td>read_double</td>
<td>7</td>
<td></td>
<td>double (in $f0)</td>
</tr>
<tr>
<td>read_string</td>
<td>8</td>
<td>$a0=buffer, $a1=length</td>
<td></td>
</tr>
<tr>
<td>sbrk</td>
<td>9</td>
<td>$a0=amount</td>
<td></td>
</tr>
<tr>
<td>exit</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- These are the original MIPS system calls
The MARS simulator has a few custom ones you will learn about later. These system calls are not available in the “vanilla” version of MARS publicly available on the web.

- There’s a lot more to the MIPS instruction set still to cover, but we (almost) know enough now to write some simple programs that do computations
- Every statement is divided into fields (Parts in square brackets are optional):
  - [Label:] operation [operands] [#comment]

**Program Structure**

- Programs are written in a plain text file with data declarations, program code (name of file should end in suffix .asm for MARS)
- EX of empty program:

```assembly
.text
.globl main
main:
    # put your code/instructions here
.data
    # put your global variables here
```

- Code (green region)
  - placed in section of text identified with assembler directive .text
  - contains program code (instructions)
  - starting point for code e.g. execution given label main:
  - ending point of main code should use exit system call (see below under System Calls)
- Data Declarations (blue region)
  - placed in section of program identified with assembler directive .data
  - declares variable names used in program; storage allocated in main memory (RAM)

**Memory (RAM)**

- All instructions and data are stored in memory (RAM).
- The format of MIPS memory is shown below. Each region of memory holds different types of information.
The instructions of the program from the .text section are stored in the Text Segment.

The static variables (data) of the program declared in the .data section are stored in the Data Segment.

The Stack Segment is a region of memory which your program can use to temporarily hold data. The stack grows downward. We will learn more about the stack later in the semester.

The Dynamic Data is for heap allocations (in Java, `new`). This space is allocated upwards into the same space as the stack. The stack and dynamic data must share the same total space.

Most computer architectures cannot read individual bits from memory.

Rather, the architecture’s instruction set can process only entire *words* or individual *bytes*.

A *word* is the unit of data used natively by a CPU. In MIPS a word is 32-bits.

Each word is divided into smaller segments called *bytes*. A byte holds 8 bits. There are 4 bytes per word.

Each row of memory holds 32 bits of data (MIPS is a 32-bit architecture).

If the smallest unit of data we can read from memory is a word, we say that the memory is **word-addressable**.

The MIPS architecture, in contrast, is **byte-addressable**. This means each byte has its own memory address.

---

**Memory addresses** (byte addresses)

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7FFFFFC</td>
<td></td>
</tr>
<tr>
<td>0x10000000</td>
<td></td>
</tr>
<tr>
<td>0x04000000</td>
<td></td>
</tr>
</tbody>
</table>

In MARS, static data starts at 0x10010000 & dynamic data starts at 0x10040000.
Little Endian vs. Big Endian

- Each 32-bit word has 4 bytes. How should we number the bytes within a word?
  - **Little-endian**: byte numbers start at the little (least significant) end – right side
  - **Big-endian**: byte numbers start at the big (most significant) end – left side
  - LSB = least significant byte; MSB = most significant byte
  - Word address is the same in either case

<table>
<thead>
<tr>
<th>Big-Endian</th>
<th>Little-Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte Address</td>
<td>Word Address</td>
</tr>
<tr>
<td>C D E F C</td>
<td>F E D C</td>
</tr>
<tr>
<td>8 9 A B 8</td>
<td>B A 9 8</td>
</tr>
<tr>
<td>4 5 6 7 4</td>
<td>7 6 5 4</td>
</tr>
<tr>
<td>0 1 2 3 0</td>
<td>3 2 1 0</td>
</tr>
</tbody>
</table>

- Why do we care?
  - There is no advantage to either of these two formats. Historically some machines used big endian and other used little endian memory formats.
  - When communicating between machines of the same endianness, there are no problems. However when you transfer data between machines of different endianness, there can be issues.
  - Data transferred byte by byte will be stored in reverse when transferred. However data transferred in larger sizes at once, like words at once, will be correctly transferred.

- **EX:** Given the integer value 260 (a 4-byte quantity) at address 100.
  00000000 00000000 00000001 00000100
  When reading back bytes from locations, 100, 101, 102, 103:
  - little-endian: You get 4, 1, 0, 0.
  - big-endian: You get 0, 0, 1, 4.

- Modernly almost all architectures adapt Little Endian.

MIPS Instructions

- MIPS has different categories of instructions:
  - **Load and Store instructions** – Access to RAM (memory)
  - **Arithmetic Instructions** – Perform mathematic operations
  - **Logical Instructions** – Perform operations on the bits of data
  - **Jump Instructions** – Changes to the flow of program execution
  - **Branch Instructions** – Conditional change to the flow of program execution

- **Instructions** which are specified in the instruction set and directly implemented by the architecture.
- Each instructions is specified using one of the 3 MIPS instruction formats: R-Type, I-type, or J-type.
  - **R-Type** – register operands (register-type instruction)
  - **I-Type** – immediate operand (immediate-type instruction)
  - **J-Type** – for jumping (jump-type instruction)

- As we discuss each category of instructions, we will define the instruction format used.

- **Basic MIPS Instructions**
  - Java code: `a = b + c;`  
    MIPS assembly code: `add a, b, c`
    - `add`: mnemonic indicates operation to perform
    - `b, c`: source operands (on which the operation is to be performed)
    - `a`: destination operand (to which the result is written)
    - In MIPS, `a, b` and `c` are registers. **EX:** `add $v0, $t0, $t1`
Java code: \( a = b - c; \)  
MIPS assembly code: \texttt{sub a, b, c}

- Notice that the MIPS instruction format is identical except for the mnemonic name
- The consistent instruction format and regularity is one of the principles of RISC. This makes the instructions easier to encode and the hardware itself simpler to build.

Java code: \( a = b + c - d; \)  
MIPS assembly code: \texttt{add t, b, c # t = b + c}  
\texttt{sub a, t, d # a = t - d}

- Complex statements are broken down into multiple lines of assembly code.
- Complex instructions are less likely to be used frequently. When implementing hardware, we want it to be fast, so we make the simple instructions as fast as possible. The hope is the combination of multiple simple fast instructions will be faster than the hardware for complex instructions. (This is the RISC principle. It does not hold true for all programs)

**Load and Store instructions**

- The only instructions which can access memory are the load and store instructions. They transfer data between the registers and memory.
  - **Load**: to read a word of data from memory  
  \texttt{lw $s0, 16($t1)}
  - **Store**: to write a word of data to memory  
  \texttt{sw $s5, 0($s1)}

- These instructions are I--type instructions.
  - 3 operands:
    - \( rs, rt \): register operands
    - \( imm \): 16-bit 2’s complement immediate (aka offset)
  - Other fields:
    - \( op \): the opcode

- These instruction use indirect addressing to refer to the location in memory to read/write the data
  - Address calculation: add base address (\( \$t1 \)) to the offset (16)  
  - A register first needs to have the base address that we want to add to the offset  
  - effective address = \( \$t1 + 16 \)
  - Result: \( \$s0 \) holds the value at address \( \$t1 + 16 \)

- **EX**: Suppose we want to read a word of data at memory address 8 into \( \$s3 \)
  - address = \( \$0 + 8 \) = 8
  - What we want is for \( \$s3 \) to hold  
  - 0x01EE2842  
  - Assembly code: \texttt{lw $s3, 8($0)}  
  # read memory from word 2 into $s3
  \begin{array}{c|c|c}
  \text{Word} & \text{Address} & \text{Data} \\
  \hline
  0000000C & 4 & 0 \\
  \hline
  00000008 & 0 & 1 \\
  \hline
  00000004 & 2 & E \\
  \hline
  00000000 & 3 & E \\
  \hline
  \end{array}

- **EX**: Suppose we want to write (store) the value in register \( \$s5 \) into memory address stored in \( \$s1 \).
  - Assume \( \$s1 \) holds the value 4.  
  - address = \( \$s1 + 0 \) = 4  
  - Assembly code: \texttt{sw $s5, 0($s1)}  
  # write the value in $s5 into memory address 4 (word 1)
  # overwrites the data in word 1 (0xF2F1AC07)
o The address of a memory word must be a multiple of 4. A word, is a row in memory. They can only be accessed as a full row.

o **EX:**
  - the address of memory word #2 is $2 \times 4 = 8$
  - the address of memory word #10 is $10 \times 4 = 40 \text{ (0x28)}$
  - Do not forget this: MIPS is byte-addressed, not word-addressed!
  - To read/write a word from/to memory, your lw/sw instruction must provide an effective address that is **word-aligned**!

o There are load and store instructions for smaller amounts of data
  - Load byte: lb
    - When loading a byte, what do we do with the other 24 bits in the 32-bit register?
      - lb sign-extends to fill the upper 24 bits
    - MIPS instruction that does not sign extend when loading bytes, load byte unsigned: lbu
  - Store byte: sb
  - Load half-word: lh
  - Store half-word: sh

• **Arithmetic Instructions**
  - These instructions only perform operations on registers.
    - Addition: add $s0, $t0, $t1 \quad \# \; s0 = t0 + t1$
    - Subtraction: sub $s1, $s0, $s2 \quad \# \; s1 = s0 + s2$
  - The basic arithmetic instructions are R-type instructions.
  - R-Type Instructions
    - 3 register operands:
      - rs, rt: source registers
      - rd: destination register
    - Other fields:
      - op: the **operation code** or opcode (0 for R-type)
      - funct: the **function**; with opcode, tells CPU what operation to perform
      - shamt: the **shift amount** for shift instructions; otherwise it’s 0
    - Other instructions such as multiplication and division exist. They also use R-type format, even though they only specify the rt and rd registers.
  - Multiplication: mult $s0, s1$
    - 32-bit $\times$ 32-bit multiplication produces a 64-bit result in {hi, lo}
    - Result goes into special registers lo and hi
    - Need special instructions to load the values from lo and hi into standard registers
      - mflo $s2$
      - mfhi $s3$
  - Division: div $s0, s1 \quad \# \; s0 / s1$
    - Quotient in lo
    - Remainder in hi

• **Logical Instructions**
  - For performing logical bitwise operations: and-ing, or-ing, xor, nor, bitwise shifting, etc.
  - and $s3, s1, s2 \quad \# \; s3 = s1 \text{ AND } s2$
    - bitwise ands the numbers in 2 registers and places the result in a 3rd register
    - **Masking** out (excluding) all but the least significant byte of a value:
$0xF234012E \text{ AND } 0x000000FF = 0x0000002E$


1111 0010 0011 1000 0000 0001 0010 1110
0000 0000 0000 0000 0000 0000 0010 1110
0 0 0 0 0 0 2 E (HEX)

- or $4$, $1$, $2$  \# $4 = $1 OR $2$
  - bitwise ors the numbers in 2 registers and places the result in a 3rd register
  - useful for combining bit fields:
    
    
    
    0xF2340000 OR 0x000012BC = 0xF23412BC

    
    1111 0010 0011 1000 0001 0010 1011 1100
    F 2 3 5 1 2 B C (HEX)

- nor: useful for inverting bits $s0 \text{ NOR } s0 = \text{ NOT } s0$
- These instructions use R-type formats, as they all require 3 registers.
- Shift instructions exist for sll, srl, sra
  - sll: shift left logical sll $t0$, $t1$, 5 \# $t0 = t1 \ll 5$
    - Shifts bits to the left, filling least significant bits with zeroes
  - srl: shift right logical srl $t0$, $t1$, 5 \# $t0 = t1 \gg 5$
    - Zeroes shift into most significant bits
  - sra: shift right arithmetic sra $t0$, $t1$, 5 \# $t0 = t1 >> 5$
    - Sign bit shifts into most significant bits
- Rotate or Circular Shift
  - Bits are not lost when we rotate them (i.e., do a “circular shift”)
  - They wrap around and enter the register from the other end
  - rol: rotate left rol $s1$, $t2$, 4
    - EX: Rotate left bits of $t2$ by 4 positions and store it into $s1$
    - 1101 0010 0011 0100 0101 0110 0111 1000
      - 0100 0111 0100 0101 0110 0111 1000 1101
  - ror: rotate right ror $t2$, $t1$, 4
- Jump Instructions
  - These instructions change the flow of execution of the program. Instead of going to the next instruction in the program, a jump instruction tells the address of the next instruction to execute.
  - These instructions also provide support for conditional & iterative blocks of code, to call methods, and to return from methods
  - j label \# jump to the address of label
  - These instructions are J-type.
    - 26-bit address operand (addr)

- There are other unconditional jump instructions which use R-type instruction format
  - jr $ra$ \# jump to the address stored in register $ra$
- Branch Instructions
  - For if-type statements and controlling loops
  - beq $a0$, $s1$, Equal_Case \# if $a0 == s1$, jump to label Equal_Case
    - if two registers have the same data, jump to instructions at a provided memory address
bne $a0, $s1, Equal_Case  #if $a0 != $s1, jump to label Equal_Case
  # if two registers have different data, jump to the instructions at a provided memory address
These instructions use the I-type format, where the label is a reference to the number of instructions forward or backwards to branch to. This value is stored in the immediate field.

- **Pseudo instructions** are mnemonic or short-cut instructions which are recognized by the assembler and replaced with one or more actual instructions. Pseudo instructions are for the programmer’s convenience.
  - **EX:** to load a 32-bit integer into a register requires **lui** and **ori**
    - Instead we can use the **li** (**load immediate**) pseudoinstruction
    - li $v0, 4  # loads 4 into $v0
  - **EX:** assume that **str** is a label (i.e., a memory address). Load the address of **str** into a register.
    - Instead we can use the **la** (**load address**) pseudoinstruction
    - la $a0, str # loads addr of str into $a0
  - **EX:** copy the contents of one register to another.
    - Use the **move** pseudoinstruction
    - move $1, $2  # equivalent to add $1, $2, $0
  - **EX:** **mul** $t0, $s1, $v0  # $t0 = $s1 * $v0
    - This pseudo instruction is equivalent to
      - mult $s1, $v0
      - mflo $t0
    - A similar instruction exists for **div**
  - **EX:** **andi**, **ori**, **xori**
    - 16-bit immediate is zero-extended (not sign-extended)
    - **nori** not needed (can use **ori** and **nor**)
    - andi $t0, $t1, 0xFF
      0001 0010 0011 0100 0101 0110 0111 1000  $t1
      0000 0000 0000 0000 0000 0000 1111 1111  0xFF
      0000 0000 0000 0000 0000 0000 0111 1000  $t0

- For a full list of MIPS instructions and pseudo instructions refer to the help menu of MARS (the course simulator).
- **MIPS Programming Tips**
  - Initialize all your variables as needed (e.g., use li)
    - The MARS simulator fills the memory with zeroes, but this is merely a convenience and luxury
    - When we test your homework programs, we will fill the memory with garbage to make sure you initialize registers with values!
  - Use the MARS debugger to fix problems with your code
    - The Registers view (on right) is especially useful
  - Use $s0-$s7 for local variables, and $t0-$t9 for temporary values, such as intermediate results