

MIPS Virtual Machine

MIPS Virtual Machine es una Máquina Virtual que emula el procesador [MIPS](#). Se trata de un procesador que tiene un juego de instrucciones RISC (Reduced Instruction Set Computing), en el que todas las instrucciones máquina tienen un tamaño de palabra fijo.

Esta máquina virtual trabaja con el juego de instrucciones del MIPS II, es un procesador de 32 bits cuyo tamaño de palabra es de 4 bytes.

Este proyecto consta de dos programas: un ensamblador que traduce el código fuente de ensamblador MIPS a código máquina para el MIPS II y el intérprete o máquina virtual que se encarga de ir procesando instrucción a instrucción el ejecutable resultante del ensamblado.

¿Qué utilidad tiene?

Uno de los propósitos de este proyecto es el educacional. A la hora de enseñar Arquitectura de Sistemas en muchas Universidades emplean como procesador un MIPS. Programar utilizando un lenguaje ensamblador sencillo, como es el caso, ayuda a comprender como trabaja un procesador internamente.

Uno de los objetivos que tenía en mente era el de poder ejecutar los programas ensamblados para MIPS en todos los Sistemas Operativos. Un programa se podría ejecutar nativamente en un procesador MIPS y a través de la máquina virtual, en un procesador Intel en Windows o en Mac OS X, por ejemplo.

1. La Máquina Virtual

El programa se llama mips y hay que ejecutarlo desde una terminal. Tan sólo necesita un parámetro para funcionar, que es el archivo que va a ejecutar.

El algoritmo que utiliza la máquina virtual es bastante simple. Abre el archivo, comprueba que se trata de archivo ejecutable en formato [Elf](#), carga en memoria la sección ejecutable y la sección de datos, y finalmente interpreta instrucción a instrucción.

Como cada instrucción ocupa 4 bytes, el tamaño de la sección ejecutable (.text) debe ser múltiplo de 4.

La sección de datos no es obligatoria, puedes hacer un programa sin esta sección.

Los procesadores MIPS tienen una serie de registros, que es donde almacenas los resultados de las operaciones.

Registros			
Nombre	Número	Uso	Preservado en llamada
\$zero	\$0	constante entera 0	sí
\$at	\$1	temporal del ensamblador	no
\$v0-\$v1	\$2-\$3	Valores de retorno de funciones y evaluación de expresiones	no
\$a0-\$a3	\$4-\$7	Argumentos de funciones	no
\$t0-\$t7	\$8-\$15	Temporales	no
\$s0-\$s7	\$16-\$23	Temporales salvados	sí
\$t8-\$t9	\$24-\$25	Temporales	no
\$k0-\$k1	\$26-\$27	Reservados para el núcleo del SO	no
\$gp	\$28	puntero global	sí
\$sp	\$29	puntero de pila	sí
\$fp	\$30	puntero de "frame"	sí
\$ra	\$31	dirección de retorno	no

Además de estos registros, existe uno especial que se llama PC (Program Counter), se trata del contador de programa que determina la siguiente instrucción a ejecutar. El valor del PC se incrementa de 4 en 4, y los saltos pueden modificar su valor, pero siempre debe ser múltiplo de 4.

El *corazón* de la máquina virtual está en la función `execute()`.

```
/*
 * Función execute.
 * Ejecuta el código del programa ya cargado en memoria interpretando cada instrucción. El programa
 * termina cuando se llama explícitamente la syscall número 10 o el PC (Program Counter) llega al
 * final del programa.
 */
void execute()
{
    uint32_t opcode = 0;

    while (cpu.PC < cpu.program_size && cpu.syscallTermination == 0)
    {
        opcode = cpu.byteCode[cpu.PC >> 2]; //Dividimos entre 4
        #ifdef DEBUG
            printf("PC: %.8x Opcode: %.8x\n", cpu.PC, opcode);
        #endif

        cpu.PC += 4;

        interpretarInstruccion(opcode);
    }

    visualizarCPUInfo(cpu);
}
```

El programa entra en un bucle que termina cuando se ejecuta una llamada del sistema específica o el PC sobrepasa el tamaño del programa en memoria (quiere decir que llegó al final).

En cada iteración se carga en **opcode** (Operation Code) la instrucción a ejecutar, se incrementa el contador del programa en 4 y se ejecuta la instrucción.

El juego de instrucciones del procesador MIPS posee 3 tipos de instrucciones dependiendo de su formato, que son:

Tipo	Formato (bits)					
-31-	-0-					
R	codop (6)	rs (5)	rt (5)	rd (5)	shamt (5)	codfunc (6)
I	codop (6)	rs (5)	rt (5)	inmediato/desplazamiento (16)		
J	codop (6)	dirección (26)				

La máquina virtual puede averiguar fácilmente que tipo de instrucción va a ejecutar, mirando los 6 bits más significativos de la instrucción. Las instrucciones Tipo-R tienen los 6 bits más significativos a 0, pero se puede saber que tipo de instrucción es mediante los 6 bits menos significativos con el campo *codfunc*.

¿Cómo diferenciar las instrucciones y que operación ejecutar de forma elegante?

Este problema lo resolví utilizando un array global de una estructura que entre sus campos contiene: nombre de la instrucción, código de la operación, tipo de instrucción (R, I o J), código de función (0x00 si es de Tipo I o J) y un puntero a la operación que va a ejecutar esa instrucción.

El resultado en el código fuente es este:

```
/* Lista de códigos de operación, separados por su nombre,
   código de operación, tipo de instrucción, código de función y
   el puntero a una función en C que realizará la operación deseada. */
opcode_t listaInstrucciones[] = {
    /* Instrucciones Tipo-R */
    {"add",      0x00, 'R', 0x20, add}, //add $d, $s, $t
    {"addu",     0x00, 'R', 0x21, addu}, //addu $d, $s, $t
    {"and",      0x00, 'R', 0x24, and}, //and $d, $s, $t
    {"div",      0x00, 'R', 0x1A, div}, //div $s, $t
    {"divu",     0x00, 'R', 0x1B, divu}, //divu $s, $t
    {"jalr",     0x00, 'R', 0x09, jalr}, //jalr $d, $s
    {"jr",       0x00, 'R', 0x08, jr}, //jr $s
    {"mfhi",     0x00, 'R', 0x10, mfhi}, //mfhi $d
    {"mflo",     0x00, 'R', 0x12, mflo}, //mflo $d
    {"mthi",     0x00, 'R', 0x11, mthi}, //mthi $s
    {"mtlo",     0x00, 'R', 0x13, mtlo}, //mtlo $s
    {"mult",     0x00, 'R', 0x18, mult}, //mult $s, $t
    {"multu",    0x00, 'R', 0x19, multu}, //multu $s, $t
    {"nor",      0x00, 'R', 0x27, nor}, //nor $d, $s, $t
    {"or",       0x00, 'R', 0x25, or}, //or $d, $s, $t
    {"sll",      0x00, 'R', 0x00, sll}, //sll $d, $t, h
    {"sllv",     0x00, 'R', 0x04, sllv}, //sllv $d, $t, $s
    {"slt",      0x00, 'R', 0x2A, slt}, //slt $d, $s, $t
```

Todos los punteros a funciones reciben los mismos parámetros que son estos:

- Un puntero a un entero de 32 bits llamado rs.
- Un puntero a un entero de 32 bits llamado rt.
- Un puntero a un entero de 32 bits llamado rd.
- Un byte que contiene el *shamt* (para trabajar con desplazamiento de bits).
- Un entero de 16 bits que contiene un desplazamiento.
- Un número entero positivo de 32 bits que contiene una dirección de memoria.

Cómo se puede observar, en cada operación paso por parámetro datos que cubren todos los campos de cada formato de instrucción, aunque no se utilicen todos.

Los 3 punteros, agilizan el proceso de modificación de los datos que contienen los registros de la CPU. Creando otro array de estructuras identifico el número del registro con su puntero adecuado.

```
/* Lista de registros con su nombre, su número de registro(son 32)
   y el puntero al registro de la CPU al que hacen referencia. */
registro_t listaRegistros[REG_COUNT] = {
    {"$zero", 0, &cpu.registros.zero},
    {"$at", 1, &cpu.registros.at},
    {"$v0", 2, &cpu.registros.v0},
    {"$v1", 3, &cpu.registros.v1},
    {"$a0", 4, &cpu.registros.a0},
    {"$a1", 5, &cpu.registros.a1},
    {"$a2", 6, &cpu.registros.a2},
    {"$a3", 7, &cpu.registros.a3},
    {"$t0", 8, &cpu.registros.t0},
    {"$t1", 9, &cpu.registros.t1},
    {"$t2", 10, &cpu.registros.t2},
    {"$t3", 11, &cpu.registros.t3},
    {"$t4", 12, &cpu.registros.t4},
    {"$t5", 13, &cpu.registros.t5},
    {"$t6", 14, &cpu.registros.t6},
    {"$t7", 15, &cpu.registros.t7},
    {"$s0", 16, &cpu.registros.s0},
    {"$s1", 17, &cpu.registros.s1},
    {"$s2", 18, &cpu.registros.s2},
    {"$s3", 19, &cpu.registros.s3},
    {"$s4", 20, &cpu.registros.s4},
    {"$s5", 21, &cpu.registros.s5},
    {"$s6", 22, &cpu.registros.s6},
    {"$s7", 23, &cpu.registros.s7},
    {"$t8", 24, &cpu.registros.t8},
    {"$t9", 25, &cpu.registros.t9},
    {"$k0", 26, &cpu.registros.k0},
    {"$k1", 27, &cpu.registros.k1},
    {"$gp", 28, &cpu.registros.gp},
    {"$sp", 29, &cpu.registros.sp},
    {"$fp", 30, &cpu.registros.fp},
    {"$ra", 31, &cpu.registros.ra}
};
```

La variable **cpu** es una estructura que contiene los siguientes datos:

```
typedef struct registers{
    int32_t zero, at;
    int32_t v0, v1;
    int32_t a0, a1, a2, a3;
    int32_t t0, t1, t2, t3, t4, t5, t6, t7, t8, t9;
    int32_t s0, s1, s2, s3, s4, s5, s6, s7;
    int32_t k0, k1;
    int32_t gp, sp, fp, ra;

    int32_t L0, HI; //Registros no referenciables directamente
} registers_t;

typedef struct cpu{
    uint32_t * byteCode;

    uint32_t program_size;
    uint32_t PC;

    uint8_t * memory;
    uint32_t memory_size;

    uint8_t syscallTermination;

    registers_t registros;
}cpu_t;
```

De esta forma nos queda un programa modular y fácilmente mantenible, ya que para agregar más instrucciones tenemos que añadir una entrada en el array de instrucciones y crear la función a ejecutar. El módulo **alu.c** tiene esta estructura:

```
void div(int32_t * rs, int32_t * rt, int32_t * rd, uint8_t shamt, int16_t offset, uint32_t direction)
{
    cpu.registros.L0 = (*rs) / (*rt);
    cpu.registros.HI = (*rs) % (*rt);
}

void divu(int32_t * rs, int32_t * rt, int32_t * rd, uint8_t shamt, int16_t offset, uint32_t direction)
{
    cpu.registros.L0 = (*rs) / (*rt);
    cpu.registros.HI = (*rs) % (*rt);
}

void jalr(int32_t * rs, int32_t * rt, int32_t * rd, uint8_t shamt, int16_t offset, uint32_t direction)
{
    *rd = cpu.PC;
    cpu.PC = *rs;
}

void jr(int32_t * rs, int32_t * rt, int32_t * rd, uint8_t shamt, int16_t offset, uint32_t direction)
{
    cpu.PC = cpu.registros.ra;
}

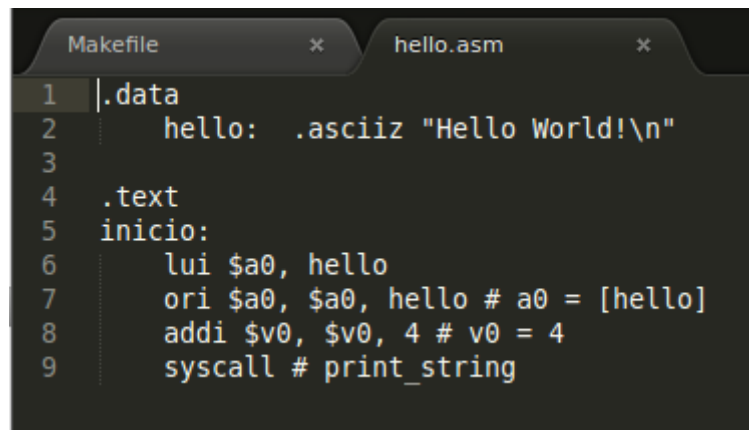
void mfhi(int32_t * rs, int32_t * rt, int32_t * rd, uint8_t shamt, int16_t offset, uint32_t direction)
{
    *rd = cpu.registros.HI;
}
```

2. El ensamblador

El programa se llama mas (Mips Assembler). Necesita 3 parámetros para funcionar: el archivo a ensamblar, un parámetro -o que indica que el siguiente parámetro va a ser el nombre del ejecutable una vez ensamblado y el nombre del ejecutable.

Los archivos de código fuente están divididos en dos partes: la sección .data y la sección .text.

La sección .data contiene variables que pueden ser arrays o datos inicializados y la sección .text contiene el código fuente de nuestro programa.



```
1 |.data
2 |    hello: .asciiz "Hello World!\n"
3 |
4 |.text
5 |inicio:
6 |    lui $a0, hello
7 |    ori $a0, $a0, hello # a0 = [hello]
8 |    addi $v0, $v0, 4 # v0 = 4
9 |    syscall # print_string
```

La sección .data es opcional y siempre tiene que ir encima de la sección .text.

El algoritmo que emplea el ensamblador para reconocer la sintaxis es bastante simple, aunque no óptimo. Abre el archivo de texto y procesa línea a línea su sintaxis, troceando las líneas eliminando los espacios, tabulaciones y saltos de línea.

Antes de trocear una línea de código fuente, la convierte a minúsculas y elimina los comentarios. Una vez tenemos la línea “limpia” y los trozos, el ensamblador utiliza un array de estructuras similar al del intérprete.

En esta estructura guarda el nombre de la instrucción, su código de operación, su tipo (R, I o J), su código de función y la cantidad de parámetros que tiene. Según el tipo de instrucción que sea, llama a una función o a otra para obtener su código máquina a partir de los parámetros. Todas estas funciones están almacenadas en el archivo mips.c

