MIPS

CS61C Spring 2017

Contents

1 What is MIPS

In order to actually run a program, a program must be converted into an executable or binary that a computer can execute. To accomplish this, every computer uses a specific instruction set architecture (ISA). An ISA consists of two main parts: 1) a constrained set of assembly language instructions, that the program will be compiled into; 2) a CPU that understands that assembly language and can execute the compiled instructions. By limiting the possible instructions to a set of specific instructions, it can be guaranteed that the CPU will be able to execute each of those instructions and thus execute the entire program.

MIPS is the ISA that we use in this class, and runs on a 32 bit architecture. Nowadays, most machines generally use x86 and run on a 32 bit or 64 bit architecture, but x86 is a more complex ISA, so we teach the more intuitive MIPS instead. Just like in C programs, MIPS programs are read sequentially from one line to the next. Additionally, **labels** can be used to label a certain line in the MIPS program for both readability and easy access for a program to know where a specific line is.

1.1 Registers

While executing a program, the CPU will use registers to hold intermediate values while executing a program. These values are generally memory addresses or integers. Physically, registers are memory units smaller than DRAM (they can hold only 32 bits of data), but much faster to use than DRAM. The MIPS architecture has 32 of these registers, each with a designated purpose. Each register has a unique number, from 0 to 31, as well as a more common name that describes the register's purpose (e.g. register 29 is known as $sp, the stack pointer register). Registers are indicated by a $. Some of the most common registers include:

- *$t registers*—the temporary registers, these are used to hold intermediate or temporary values between computations.
- *$s registers*—the saved registers, these are used to hold intermediate values that are guaranteed to be preserved across function calls (explained more in Calling Convention).
- *$a registers*—the argument registers, before calling a function, argument values are placed in these registers.
- *$v registers*—the return value registers, before a function returns, the return value is placed in $v0 ($v1 is used for returning other system call metadata).
- *$sp register*—the stack pointer register, keeps track of the address of the current limit of the stack (the memory addresses greater than this register's value are in use, while those less than are free to use).
- *$ra register*—the return address register, keeps track of the instruction address to return to after the program finishes a function call.

1.2 Instructions

Instructions are specific actions that programs can take. MIPS instructions have the following format: `instruction arg0 arg1 arg2`. Depending on the specific instruction, an instruction may take one, two, or three arguments (such as `jal`, `lw`, or `add`, respectively) but a specific instruction will always take the same number of arguments (e.g. `add` always takes three arguments). The MIPS Green Sheet documents every MIPS instruction, telling you what that instruction does as well as other information that will be useful later. Since MIPS runs on a 32 bit architecture, each instruction along with its arguments can be represented with 32 bits and each type of instruction has a specific format of its arguments and those 32 bits. There are three types of MIPS instructions, each with a specific format:

- **R-type**: `rtype $rd $rs $rt` these instructions take three registers as arguments, with the result of the instruction being stored in $rd, or the destination register.
- **I-type**: `itype $rt $rs immediate` these instructions take two registers and an immediate as arguments. An immediate is any 16 bit integer (signed or unsigned depending on the instruction), and the result of the instruction is stored in $rt, or the target register.
- **J-type**: `jtype LABEL` these instructions take a label as a single argument, and will generally jump the program to the given label in order to continue program execution at the instruction at that label.

Also, note that *instruction* may be used to refer to a specific MIPS instruction (e.g. `jal, lw, add`) as well as an entire program line with arguments (e.g. `add $t0 $t1 $t2`).

2 MIPS Instructions & Programs

When writing MIPS programs, there are many useful types of instructions as well as common conventions to make programs clear and organized. Figures 1 and 2 are an example program (in C and in MIPS) that sets the value of each node in a linked list to the number of nodes after that node (e.g. a four node linked list would then have values $3 \rightarrow 2 \rightarrow 1 \rightarrow 0$). We will reference parts of it throughout this section, as we learn more about instructions and MIPS programs. Note: 0 is often used as the numerical value for `NULL`.

```c
struct node {
  int val;
  struct node* next;
};

void initLL(int len) {
  if (len == 0)
    return NULL;
  struct node* head = getNode();
  struct node* cur = head;
  while (len > 0) {
    cur->next = getNode();
    cur = cur->next;
    len -= 1;
   }
  return head;
}

void valToLen(struct node* nd) {
  if (nd == NULL)
    return 0;
  else {
    nd->val = valToLen(nd->next);
    return 1 + nd->val;
   }
}

int main(int argc, char* argv[]) {
  struct node* ll = initLL(4);
  valToLen(ll);
}
```

Figure 1: C Program (Linked List)

```
main:           addiu $a0 $0 4
                jal initLL
                addu $s0 $v0 $0
                move $a0 $s0
                jal valToLen
initLL:         bne $a0 $0 prolog1
                move $v0 $0
                j done
prolog1:        addiu $sp $sp -12
                sw $s0 0($sp)
                sw $s1 4($sp)
                sw $s2 8($sp)
                move $s0 $a0 # s0 = len
                jal getNode
                move $s1 $v0 # s1 = head
                move $s2 $s0 # s2 = cur
loop:           jal getNode
                sw $v0 4($s2)
                lw $s2 4($s2)
                addiu $s0 $s0 -1
                bne $s0 $0 loop
                move $v0 $s1
epilog1:        lw $s2 8($sp)
                lw $s1 4($sp)
                lw $s0 0($sp)
                addiu $sp $sp 12
done:           jr $ra
valToLen:       move $t0 $a0
if_:            bne $t0 $0 prolog2
                move $v0 $0
                j end
prolog2:        addiu $sp $sp -4
                sw $s0 0($sp)
else_:          move $s0 $a0
                lw $a0 4($s0)
recurse:        jal valToLen
                sw $v0 0($s0)
                addiu $v0 $v0 1
epilog2:        lw $s0 0($sp)
                addiu $sp $sp 4
end:            jr $ra
```

Figure 2: MIPS Program (Linked List)

2.1 Arithmetic Instructions & Sign Extension

There are specific R-type instructions to carry out each of the basic arithmetic and bitwise operations on two registers, such as addition, subtraction, OR, AND, and XOR. There are also I-type instruction versions of many of these operations that instead take a register and an immediate.

As shown on the green sheet, all of these instructions will sign extend the given immediate, except for `ori` and `andi`, which will zero extend the immediate instead. Since I-type instructions can only take a 16 bit immediate, but register values are 32 bits, the immediate must be extended to 32 bits. That is, bits 0 to 15 of the given 16-bit immediate are bits 0 to 15 of the extended 32-bit immediate, and we must then fill in bits 16 to 31 of the extended immediate. Sign extension means that bits 16 to 31 of the extended immediate will be equal to the most significant bit of the immediate (bit 15), that is the sign bit, while zero extension means bits 16 to 31 are instead always set to 0.

When multiplying two 32 bit integers, it is possible for the product to be greater than 32 bits (but not more than 64 bits). To handle this, the `mult` instruction will actually split the resulting product between two registers, placing the upper 32 bits (bits 32 to 63) of the product into the special `$HI` register and the lower 32 bits (0 to 31) into the special `$LO` register. Similarly, division produces both a quotient and a remainder. To handle this, the `div` instruction will store the quotient into the `$LO` register and the remainder into the `$HI` register.

2.2 Memory Instructions

As we saw in the last note, programs interact with memory frequently, so MIPS also provides specific instructions to interact with memory. The load instructions load data from memory, while the store instructions store data in memory. Depending on the instruction, the amount of data loaded/stored can be 1 byte (load/store byte), 2 bytes (halfword), or 4 bytes (word). The load and store instructions have the following format: `inst $rt offset($rs)`, where the value in `$rs` is a memory address. Load instructions then load the value in memory at address, `(address in $rs) + offset bytes`, into `$rt`, while store instructions will store the value in `$rt` into the memory at address, `(address in $rs) + offset bytes`. You can think of loading/storing as using the derference operator from C, with the value in `$rs` being the pointer you want to dereference.

Lines 34-37 of Figure 2 demonstrate using `lw` and `sw`, and Figure 3 shows the corresponding data in memory. In this example, node B is the next node after node A, and `$s0` contains the address of node A (`0x8FBAAAE0`). `lw $a0 4($s0)` then loads the pointer to node B into `$a0`, and `sw $v0 0($s0)` stores the return value of the recursion, 3, into node A's `val` member. Specifically, `lw $a0 4($s0)` goes to address `0x8FBAAAE0 + 4 = 0x8FBAAAE4` and loads the value of the next four bytes of memory at that address into `$a0`; `sw $v0 0($s0)` then goes to address `0x8FBAAAE0 + 0 = 0x8FBAAAE0` and sets the value of the next four bytes of memory to 3.

	···				`0x8FBAAAF0`
B	8F	BA	AA	F0	struct node* next;
	00	00	00	02	int val;
A	8F	BA	AA	E8	struct node* next;
	00	00	00	03	int val;
	···				`0x8FBAAADC`

Figure 3: Corresponding memory contents to fig. 2, lines 34-37.

2.3 The Program Counter

MIPS programs keep track of the current line that is being executed using a special register, called the **program counter**, or **PC**, which simply holds the address of the current line. This is an address in memory in the code segment of memory. When a program executes, the CPU knows which line to execute next by checking the value of the PC, going to that address in memory, and then reading the line stored

```
while (len > 0) {
  cur->next = getNode();
  cur = cur->next;
  len -= 1;
}

if (nd == NULL)
  return 0;
else {
  nd->val = valToLen(nd->next);
  return 1 + nd->val;
}
```

Figure 4: if/else & while loop in C.

```
loop:           jal getNode
                sw $v0 4($s2)
                lw $s2 4($s2)
                addiu $s0 $s0 -1
                bne $s0 $0 loop
                move $v0 $s1
                ...
if_:            bne $t0 $0 prolog2
                move $v0 $0
                j end
                ...
else_:          move $s0 $a0
                lw $a0 4($s0)
recurse:        jal valToLen
                addiu $t0 $v0 1
                sw $t0 0($s0)
                move $v0 $t0
```

Figure 5: corresponding code in MIPS.

there. The PC is incremented by 4 bytes (i.e. 32 bits) to the next instruction, after each line, unless an instruction is executed that changes the PC to a different address.

2.4 Control Flow Instructions

The branch instructions can be used to implement the coding constructs that are based on conditionals, such as if statements and while/for loops. Both `branch equal (beq)` and `branch not equal (bne)` have the following format: `branch $rs $rt LABEL`. For `beq`, the branch will be taken (i.e. the Program Counter will be set to the address of the label and the instruction at the label will be executed next) if `$rs` is equal to `$rt`, otherwise the branch is not taken (i.e. simply the next sequential line is executed). For `bne`, the branch is taken if the two registers are not equal to each other.

The branch instructions can implement an if-else statement by branching to the "true/false" block of code, and otherwise executing the "other" block of code. Similarly, the branch instructions can implement while/for loops by branching to the beginning of the loop if the condition is still true. Figures 4 and 5 show an example of both from the linked list program.

In addition to the branch instructions, the set less than instructions (`slt, slti, sltu, sltiu`) are also useful for comparisons for conditionals. These instructions will compare a register to an immediate or another register. If the register is less then the immediate/other register, the target/destination register will be set to 1, otherwise it will be set to 0.

2.5 Signed vs. Unsigned

Many instructions have both a signed and unsigned version. For example, `addi` and `addiu`. Unfortunately, the meaning of signed/unsigned depends on the specific instruction and is often not what you'd think.

For the arithmetic instructions, both registers are considered as signed numbers, and unsigned indicates that signed overflow should not stop execution while signed indicates that signed overflow should stop execution.

For the load instructions, since loading halfwords and bytes of data into 32 bit registers requires the halfword/byte to be expanded to 32 bits, signed indicates that the loaded halfword/byte will be sign extended, while unsigned indicates that the halfword/byte will be zero extended.

For the set less than instructions, signed indicates to consider the registers as signed integers while unsigned indicates to consider the registers as unsigned integers.

2.6 Jump Instructions and Function Calls

Like branch instructions, jump instructions can be used to redirect the program from executing the next sequential line to another line identified by a label. Jump, or `j LABEL`, does just that by setting the Program Counter to the address of the given label. This jumping behavior is useful for executing function calls by jumping to the program line where the function body begins.

However, for most programs, if the program's current code block calls a function, then after the function call, the next line in the code block should be executed. Therefore, it is useful to save the return address before we jump into a function call, so that after the function call finishes, the program will know the address of the line to execute next.

Jump and link, or `jal LABEL`, performs this behavior by first storing the address of the next line to execute (i.e. the current value of the Program Counter plus 4 bytes) into the return address register, `$ra`, and then jumping to the given label. After the function call finishes, the program then simply needs to set the Program Counter to the value in `$ra`. Jump register, or `jr $rs`, is usually used to do this, as it sets the Program Counter equal to the value in `$rs`. Figure 6 shows a snippet of the linked list program that calls various functions.

Lastly, jump and link register, or `jalr $rs`, can be used to jump to the address in `$rs` while also setting the value of `$ra` to the Program Counter plus 4 bytes.

In addition to jumping to the function body, a program must also pass arguments and obtain the return value of the function when executing a function call. To pass arguments, argument values are stored in the `$a registers` before jumping to the function body. The function body will then store the return value of the function into `$v0` before returning (i.e. executing `jr $ra`). Figure 6 shows an example of this argument passing and return value storing.

```
main:           addiu $a0 $0 4  #set len arg of initLL to 4
                jal initLL      #call initLL
                addu $s0 $v0 $0 #store retval in $s0
                move $a0 $s0    #set nd arg of valToLen
                jal valToLen    #call valToLen
                ...
                lw $a0 4($s0)   #set nd arg of valToLen
recurse:        jal valToLen    #call valToLen
                addiu $t0 $v0 1
                sw $t0 0($s0)
                move $v0 $t0    #set return value
                ...
                jr $ra          # return to previous recurse call
```

Figure 6: Function calls.

2.7 Calling Convention

Programs are complex, and it is unlikely that the entire program could be executed using only up to 32 intermediate values at a time (since there are only 32 registers). Additionally, due to the specific purposes of different registers, function are likely to use the same registers in their function bodies; this creates problems when a function needs to preserve values in certain registers but also needs to call another function that may change the values of those registers.

For example, in our linked list program, a call to `valToLen` will need to remember the value of its `nd` arg (stored in `$a0`) to eventually set `nd->val`, but before that, the recursive call to `valToLen` will require the value of `$a0` to be changed to `nd->next`.

MIPS provides a calling convention in order to solve this problem and clearly coordinate which registers' contents will be preserved across function calls. During a function call, the line (and surrounding code section) that calls the function is the *caller*, while the function being called is the *callee*. The caller can trust that the values of the registers below will remain the same after the function call, and that if the callee needs to use those registers, the callee will save and restore those registers' values by the end of the function call. Every other register's value may not be preserved after the function call, and therefore the callee may freely change the values of these registers without needing to save and restore them later.

Saved	Not Saved
`$s registers`	`$t registers`
`$sp`	`$a registers`
`$fp`	`$v registers`
`$gp`	`$k registers`
	`$ra`
	`$at`

Figure 7: Calling Convention registers

To actually save and restore these register values, the values are stored onto the stack before the actual logic of the function and then loaded back into the matching registers after the function logic is complete (but before returning). These sections of MIPS functions are called the **prologue** and **epilogue**. In the prologue, to reserve space on the stack, the stack pointer is *decremented* by k bytes, where k is the number of registers to save multiplied by 4. The value of the first register to save is then stored on the stack at `0($sp)`, the next at `4($sp)`, and so on and so forth. The order that the registers are saved in does not matter, as long as they are restored in the same order in the epilogue. The epilogue then reverses the effects of the prologue, by loading the values from the stack back into the matching registers, and then *incrementing* the stack pointer back to its original value. Program B shows an example of a prologue and epilogue.

```
prolog1:        addiu $sp $sp -12  #decrement stack pointer
                sw $s0 0($sp)
                sw $s1 4($sp)
                sw $s2 8($sp)
                ...                    # do stuff
epilog1:        lw $s2 8($sp)
                lw $s1 4($sp)
                lw $s0 0($sp)
                addiu $sp $sp 8  # increment stack pointer
done:           jr $ra
```

Figure 8: A prologue and epilogue.

During the prologue, the callee need only save the register values of the registers that the callee will change during the function call. For example, even though all `$s registers` must be preserved, if the callee never changes the value of `$s0` then the callee does not need to save the value of `$s0` on the stack. This also means that if only say `$t registers` are used in a function body, then no register values need to be saved and no prologue and epilogue is necessary. However, when writing a function body, it doesn't hurt to save every (non `$t`) register that you will use to guarantee that the callee will not compromise the caller (for this reason, you might sometimes see prologues that do save the `$a registers`).

2.8 Pseudoinstructions

All of the above instructions are **True Assembly Language (TAL)**, meaning that the instructions are built into the MIPS ISA. However, other instructions exist that are not built into the MIPS ISA, but perform common behavior, such as moving a register's value into another register. These other instructions are called **pseudoinstructions**, which are shortcut instructions that the assembler will later expand into actual instructions. Along with the TAL instructions, these pseudoinstructions make up **Machine Assembly Language (MAL)**. All of the MIPS pseudoinstructions can be found under the pseudoinstructions set section of the MIPS green sheet.

Pseudoinstructions may simply substitute for one instruction (e.g. `move $rd $rs` is expanded to `addu $rd $rs $0`), or for multiple instructions. For an example of the latter, the `li $rd imm` instruction loads a

32 bit immediate into a register. Since the immediate of an I-Type instruction is only 16 bits, the compiler will expand `li $rd imm` into two instructions: 1) `lui $rd imm[31:16]`, which will load the upper 16 bits of the immediate into the register; 2) `ori $rd $rd imm[15:0]`, which will then ”load” the lower 16 bits of the immediate.

3 Converting MIPS to binary

We’ve learned how to convert C programs into a smaller set of MIPS instructions, and now we’ll see how to convert any MIPS instruction to its binary representation. By converting each instruction of a MIPS program into binary, we can then store the binary into memory at the code segment. For all instructions, the registers used are represented by their register number (listed on the Green Sheet). For example, `$t1` is register 17 and would be represented as 0b10001.

3.1 R-Type

R-Type instructions have the format below. They can be identified by their opcode field with value zero, and each has a unique funct field.

31 26	25 21	20 16	15 11	10 6	5 0
opcode	$rs	$rt	$rd	shamt	funct

- *opcode*—unused for R-type instructions, this field is always all zeroes for R-type instructions
- *$rs, $rt, $rd*—corresponding number of the register used. Since there are 32 different registers, five bits are necessary to represent all possible register numbers. If an instruction does not use one of these registers (`$rs, $rt, $rd`), then its corresponding field can be anything (usually all zeroes).
- *shamt*—used only for `sll`, `srl`, and `sra`, this field indicates the number of bits to shift. Since there are 32 bits in a register, five bits are necessary to represent all possible number of bits to shift by. If an instruction is not a shift instruction, this field can be anything (usually all zeroes).
- *funct*—identifies the specific R-type instruction, the Green Sheet documents the funct field for each R-type instruction.

3.2 I-Type

I-Type instructions have the format below. They can be identified by their nonzero opcode field.

31 26	25 21	20 16	15 0
opcode	$rs	$rt	immediate

- *opcode* identifies the specific I-type instruction, the Green Sheet documents the opcode field for each I-type instruction.
- *$rs, $rt* corresponding number of the register used. Since there are 32 different registers, five bits are necessary to represent all possible register numbers.
- *immediate* value of the immediate used in the instruction (this is why immediates have to be 16 bits!). For memory instructions, the offset corresponds to the value of this immediate field. For branch instructions, the value of the immediate field is the number of lines away from the PC that the given label is, or (*PC addr. − label addr.*)/4, and is calculated after compilation (explained in detail in Assembler).

3.3 J-Type

J-Type instructions have the format below. There are only two J-type instructions: `j` and `jal` with opcodes 0x02 and 0x03 respectively. `jr` and `jalr` are actually R-type instructions!

31 26	25 0
opcode	address

- *opcode* either 0x02 or 0x03
- *address* the 26 bit truncation of the address of the label, specifically bits 2 to 27 of the label address. The full address to jump to can then be computed by concatenating the four leftmost bits of the PC to these given 26 bits and multiplying by 4; or, as $PC[31:28] \| addr.value \| 00$, where $\|$ indicates concatenation and PC[31:28] is bits 28 to 31 of the current PC address.[1]

4 CALL: Translating C programs to Binary

Now that we know how to translate C programs to MIPS programs and then into binary, we can now understand the exact stages that actually perform these translations and how they do so. The following four stages are how the `gcc` command takes in your C file and outputs an executable!

4.1 Compiler

- *Input:* C code (a .c file)
- *Output:* MAL (a .s file)

In this stage, the C program is compiled into MAL, or MIPS instructions that may include pseudoinstructions as well. In general, this stage will compile a high level language into assembly language that the assembler understands.

4.2 Assembler

- *Input:* MAL (a .s file)
- *Output:* Object code as TAL + information tables (a .o file)

In this stage, the intermediate file from the compiler is assembled into an object file. Pseudoinstructions are expanded, and the intermediate file is organized into specific sections, such as the text segment, the data segment, and relocation information. Two important information tables are also constructed that contain information on the labels that are present in the program and need to be resolved. The two most relevant tables are the **symbol table**, which tracks where labels can be found in the intermediate file, and the **relocation table**, which tracks the lines of the file that need labels to be resolved to numerical values. Figures 9 and 10 show examples of these two tables for our linked list program.

Specifically, the symbol table maps labels to the relative address of the line matching the label. The address is relative because it is relative to the beginning of the given intermediate file, and the absolute address of the line may change when all intermediate files are combined into a single file. The relocation table maps relative addresses to the label that is used in that line.

After constructing these two tables (in one pass over the intermediate file), the assembler can take a second pass over the intermediate file to resolve the labels in branch instructions to numerical values. As the assembler reads each line of the intermediate file, it can check if the current line's address is present in the relocation table. If the line is present, then the assembler can look for that label in the symbol table, find the address corresponding to the label, and calculate the immediate of the branch instruction as: $(current\ line\ address - label\ address)/4$.

[1]For example, if the current PC address was 0x3080004C and the value of the address field was 0x18FCFC0 (or 0b01,1000,1111,1110,1111,1110,0000) then the label address to jump to would be: $0011|01, 1000, 1111, 1110, 1111, 1110, 0000|00 = 0b0011, 0110, 0011, 1111, 1011, 1111, 1000, 0000$ or 0x363FBF80.

Label	**Address**
`main`	Line 1 = `0x00`
`initLL`	Line 6 = `0x05`
`prolog1`	Line 9 = `0x09`
`loop`	Line 14 = `0x0e`
`epilog1`	`0x14`
`done`	`0x17`
`valToLen`	`0x18`
`if_`	`0x19`
`prolog2`	`0x1c`
`else_`	`0x1e`
`recurse`	`0x20`
`epilog2`	`0x24`
`end`	`0x26`

Figure 9: Symbol Table

Address	**Label Used**
Line 1 = `0x00`	`initLL`
Line 5 = `0x04`	`valToLen`
Line 8 = `0x07`	`done`
Line 14 = `0x0c`	`getNode`
`0x12`	`loop`
`0x19`	`prolog2`
`0x20`	`valToLen`

Figure 10: Relocation Table

4.3 Linker

- *Input:* multiple object code files + information tables (a .o file)
- *Output:* a single executable file (a .out file)

In this stage, the multiple object files (corresponding to the multiple C files comprising the program) are linked into a single executable file. This is done by combining the object files' different text segments and different data segments into a single text segment and single data segment.

Additionally, now that all of these object files are now merged as a single file, the new address of a line is now its absolute address and the labels in jump instructions can be resolved to absolute addresses. Like for branch instructions, the linker can pass over the merged file to find jump instructions that need resolution and the labels they use with the relocation table, and then find the labels' relative addresses with the symbol table. Since the linker knows both the length of each text and data segment and the ordering of each segment, it can then calculate the absolute address of any label from its relative address (assuming that the first line of the text segment starts at 0x40000000). After calculating the absolute address, the actual address value for the instruction is computed as bits 2 to 27 of the absolute address.

4.4 Loader

- *Input:* executable file (a .out file)
- *Output:* a running program!

Finally, in this stage, the executable file is loaded into memory and the loader (the operating system) will read the executable file (i.e. the text segment) and execute the program.