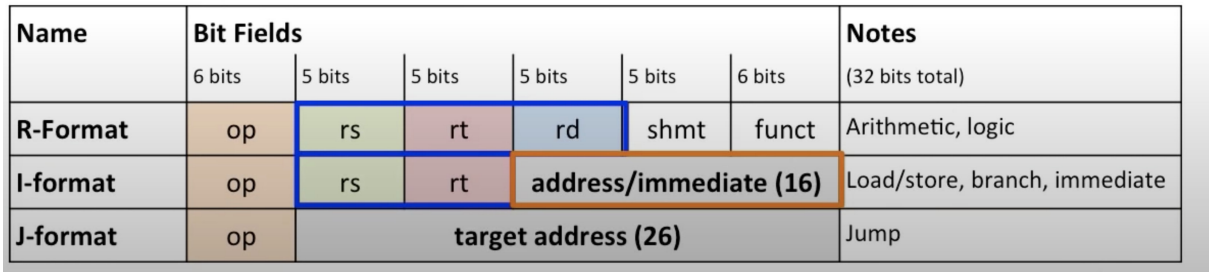
By Xinran Zhu

Last modified on 25/11/2020

MIPS Instruction Set

MIPS has 3 instruction formats:

- R: Operation 3 registers no immediate
- I: Operation 2 registers 16-bit immediate
- J: jump 0 registers 26-bit immediate



Name	Bit Fields						Notes
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	(32 bits total)
R-Format	op	rs	rt	rd	shmt	funct	Arithmetic, logic
I-format	op	rs	rt	address/immediate (16)			Load/store, branch, immediate
J-format	op	target address (26)					Jump

Integer Arithmetic Instructions

assembly	meaning	bit pattern
`add` r_d, r_s, r_t	$r_d = r_s + r_t$	000000ssssstttttddddd00000100000
`sub` r_d, r_s, r_t	$r_d = r_s$ - r_t	000000ssssstttttddddd00000100010
`mul` r_d, r_s, r_t	$r_d = r_s$ * r_t	011100ssssstttttddddd00000000010
`rem` r_d, r_s, r_t	$r_d = r_s$ % r_t	pseudo-instruction
`div` r_d, r_s, r_t	$r_d = r_s$ / r_t	pseudo-instruction
`addi` r_t, r_s, I	$r_t = r_s + \mathrm{I}$	001000ssssstttttIIIIIIIIIIIIIIII

Bit Manipulation Instructions

assembly	meaning	bit pattern
and r_d, r_s, r_t	$r_d = r_s$ & r_t	000000ssssstttttddddd00000100100
or r_d, r_s, r_t	$r_d = r_s$ | r_t	000000ssssstttttddddd00000100101
xor r_d, r_s, r_t	$r_d = r_s$ ^ r_t	000000ssssstttttddddd00000100110
nor r_d, r_s, r_t	$r_d = \sim (r_s \mid r_t)$	000000ssssstttttddddd00000100111
andi r_t, r_s, I	$r_t = r_s$ & I	001100ssssstttttIIIIIIIIIIIIIIII
ori r_t, r_s, I	$r_t = r_s$ | I	001101ssssstttttIIIIIIIIIIIIIIII
xori r_t, r_s, I	$r_t = r_s$ ^ I	001110ssssstttttIIIIIIIIIIIIIIII
not r_d, r_s	$r_d = \sim r_s$	pseudo-instruction

Jump Instruction

assem.	meaning	bit pattern
`j` *label*	pc = pc & 0xF0000000 | (X<<2)	000010XXXXXXXXXXXXXXXXXXXXXXXXXX
`jal` *label*	r_{31} = pc + 4; pc = pc & 0xF0000000 | (X<<2)	000011XXXXXXXXXXXXXXXXXXXXXXXXXX
`jr` r_s	pc = r_s	000000sssss00000000000000001000
`jalr` r_s	r_{31} = pc + 4; pc = r_s	000000sssss00000000000000001001

Branch Instruction

assembler	meaning	bit pattern
`b` *label*	pc += I<<2	pseudo-instruction
`beq` r_s,r_t,*label*	if (r_s == r_t) pc += I<<2	000100ssssstttttIIIIIIIIIIIIIIII
`bne` r_s,r_t,*label*	if (r_s != r_t) pc += I<<2	000101ssssstttttIIIIIIIIIIIIIIII
`ble` r_s,r_t,*label*	if (r_s <= r_t) pc += I<<2	pseudo-instruction
`bgt` r_s,r_t,*label*	if (r_s > r_t) pc += I<<2	pseudo-instruction
`blt` r_s,r_t*label*	if (r_s < r_t) pc += I<<2	pseudo-instruction
`bge` r_s,r_t*label*	if (r_s >= r_t) pc += I<<2	pseudo-instruction
`blez` r_s,*label*	if (r_s <= 0) pc += I<<2	000110sssss00000IIIIIIIIIIIIIIII
`bgtz` r_s,*label*	if (r_s > 0) pc += I<<2	000111sssss00000IIIIIIIIIIIIIIII
`bltz` r_s,*label*	if (r_s < 0) pc += I<<2	000001sssss00000IIIIIIIIIIIIIIII
`bgez` r_s,*label*	if (r_s >= 0) pc += I<<2	000001sssss00001IIIIIIIIIIIIIIII

Syscall

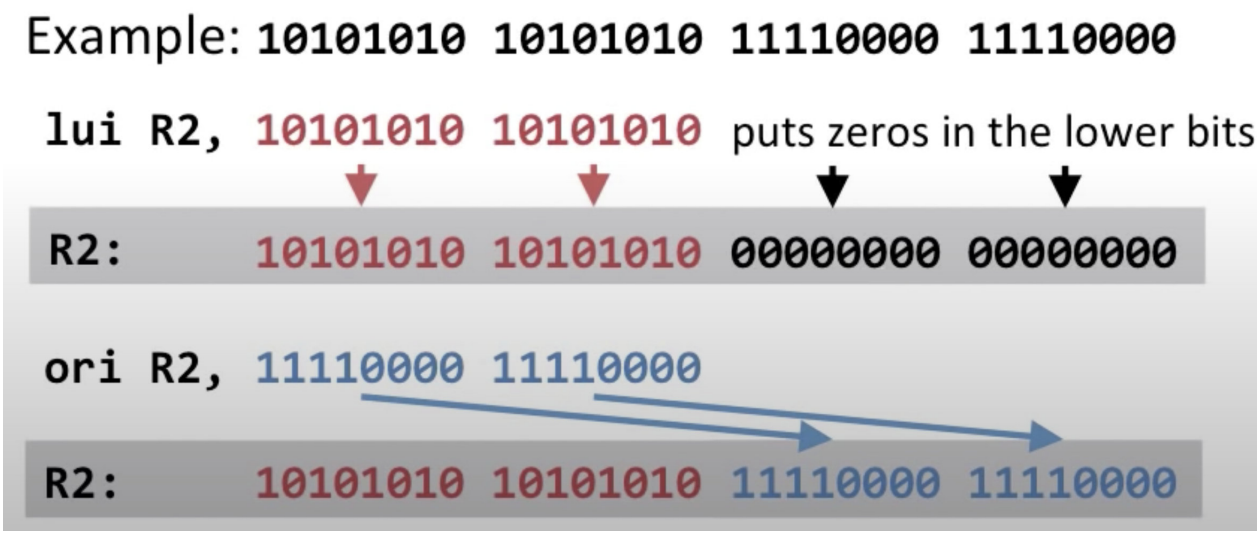
Service	$v0	Arguments	Returns
`printf("%d")`	1	int in $a0	
`printf("%s")`	4	string in $a0	
`scanf("%d")`	5	none	int in $v0
`fgets`	8	buffer address in $a0 length in $a1	
`exit(0)`	10	status in $a0	
`printf("%c")`	11	char in ‘$a0	
`scanf("%c")`	12	none	char in $v0

Sign Extensions

```
# $v1 = $a1 + $s1
add $v1, $a1, $s1
```

```
# $t0 = $t1 + 5 (signed 16 bits: -2^(16-1) to 2^(16-1) -1)
addi $t0, $t1, 5

# then 5 is sign extended to fit in the 32 bit register
```

```
0000 0000 0000 0101
to
0000 0000 0000 0000 0000 0000 0000 0101

# another example
1111 1111 0010 1100
to
1111 1111 1111 1111 1111 1111 0010 1100

# To store more than 16 bits, we use lui and ori together
```




```
# [pseudo] $t0 = $t1 % $t2
rem $t0, $t1, $t2

# real instruction
```

Translation C --> Assembly

The main function

```c
int main(void) {
    // code
    return 0;
}
```

```
main:
    # code
    li $v0, 0       # return 0
    jr $ra
```

printf

```c
int i = 5;
printf("%d\n", i);
```

```mips
# int i = 5;
li $s0, 5       # save i into $s0

# printf("%d\n", i);
# step1: printf("%d", i);
li $v0, 1       # printing an int
move $a0, $s0
syscall

# step2: printf("%c", '\n');
li $v0, 11      # printing a char
li $a0, '\n'
syscall
```

scanf

syscall code for reading integer: 5

```c
int num;
scanf("%d", &num);
```

```mips
li $v0, 5
syscall
```

if...else...

- In C
  - if condition true --> Do thing
  - if conditio false --> Do branch
- In MIPS
  - If condition is true --> branch
  - if condition is false --> do thing

```c
if (a < 10) {
    // code1
} else {
    // code2
}
// ...
```

```
# save a in $s0

    bge $s0, 10, else   # if a >= 10
    # code 1
    b next
else:
    # code2
next:
    # ...
```

Tricks of Branching

Comparing value

```
# assume i in $t0
    bge $t0, 0, else1   # if (i < 0) {
    # code1             #   // code1
goto end1               # }
else1:                  # else {
    # code2             #   // code 2
end1:                   # }
```

```
# assume i in $t0
    bgt $t0, 0, else1   # if (i <= 0) {
    # code1             #   // code1
goto end1               # }
else1:                  # else {
    # code2             #   // code 2
end1:                   # }
```

```
# assume i in $t0
    ble $t0, 0, else1   # if (i > 0) {
    # code1             #   // code1
goto end1               # }
else1:                  # else {
    # code2             #   // code 2
end1:                   # }
```

```
# assume i in $t0
    blt $t0, 0, else1   # if (i >= 0) {
    # code1             #   // code1
goto end1               # }
else1:                  # else {
    # code2             #   // code 2
end1:                   # }
```

Divisibility

```
rem r1, r2, r3       r1 = r2 % r3
```

```
# assume i in $t0
    rem $t1, t0, 7
    bne $t1, 0, else1   # if (i % 7 == 0) {
    # code1             #   // code1
goto end1               # }
else1:                  # else {
    # code2             #   // code 2
end1:                   # }
```

||

```
i < 0 || n >= 42
```

```
# assume i in $t0
    blt $t0, 0, else1       # if (i < 0 || n >= 42) {
    bge $t1, 42, else1
    # code1                 #   // code1
    j end1                  # }
else1:                      # else {
    # code2                 #   // code2
end1:                       # }
```

&&

```
!(i < 0 && n >= 42) == (i > 0 || n < 42)
```

```
# assume i in $t0
    bge $t0, 0, else1       # if (i < 0 && n >= 42) {
    blt $t1, 42, else1
    # code1                 #   // code1
    j end1                  # }
else1:                      # else {
    # code2                 #   // code2
end1:                       # }
```

Memory

Traverse an 1D array

```c
// read 10 numbers into an array then print the 10 numbers
#include <stdio.h>

int numbers[10] = { 0 };

int main(void) {
    int i;

    i = 0;
    while (i < 10) {
        printf("Enter a number: ");
        scanf("%d", &numbers[i]);
        i++;
    }
    i = 0;
    while (i < 10) {
        printf("%d\n", numbers[i]);
        i++;
    }
    return 0;
}
```

```
# i in t4
# temporary data in t0 - t3

.text
main:

    li      $t4, 0
loop1:
    bge     $t4, 10, end1

```

```
    li      $v0, 4
    la      $a0, prompt
    syscall

    li      $v0, 5
    syscall

    la      $t1, num
    mul     $t2, $t4, 4
    add     $t2, $t2, $t1

    sw      $v0, 0($t2)

    addi    $t4, $t4, 1
    j       loop1
end1:

    li      $t4, 0
loop2:
    bge     $t4, 10, end2

    la      $t1, num
    mul     $t2, $t4, 4
    add     $t2, $t2, $t1

    lw      $a0, 0($t2)

    li      $v0, 1
    syscall

    li      $a0, '\n'
    li      $v0, 11
    syscall

    addi    $t4, $t4, 1
    j       loop2
end2:

    li      $v0, 0
    jr      $ra

.data
    .align 2
num:
    .space 40
    .align 2
prompt:
    .asciiz "Enter a number: "
```

Traverse a 2D array

```c
// print a 2d array

#include <stdio.h>

int numbers[3][5] = {{3,9,27,81,243},{4,16,64,256,1024},
{5,25,125,625,3125}};

int main(void) {
    int i = 0;
    while (i < 3) {
        int j = 0;
        while (j < 5) {
            printf("%d", numbers[i][j]);
            printf("%c", ' ');
            j++;
        }
        printf("%c", '\n');
        i++;
    }
    return 0;
}
```

```asm
# i in $t4
# j in $t5
.text
main:
    li      $t4, 0
loop1:
    bge     $t4, 3, end1
    li      $t5, 0
loop2:
    bge     $t5, 5, end2

    la      $t0, array
    mul     $t1, $t4, 20
    mul     $t2, $t5, 4
    add     $t0, $t0, $t1
    add     $t0, $t0, $t2

    lw      $a0, 0($t0)
    li      $v0, 1
    syscall
```

```
    li      $a0, ' '
    li      $v0, 11
    syscall

    addi    $t5, $t5, 1
    j       loop2
end2:

    li      $a0, '\n'
    li      $v0, 11
    syscall

    addi    $t4, $t4, 1
    j       loop1
end1:

    li      $v0, 0
    jr      $ra


.data
array:
    .word 3, 9, 27, 81, 243, 4, 16, 64, 256, 1024, 5, 25, 125, 625, 3125
```

Struct

```c
// access fields of a simple struct

#include <stdio.h>
#include <stdint.h>

struct details {
    uint16_t  postcode;
    char      first_name[7];
    uint32_t  zid;
};

struct details student = {2052, "Alice", 5123456};

int main(void) {
    printf("%d", student.zid);
    putchar(' ');
    printf("%s", student.first_name);
    putchar(' ');
    printf("%d", student.postcode);
```

```
    putchar('\n');
    return 0;
}
```

Arrays in memory

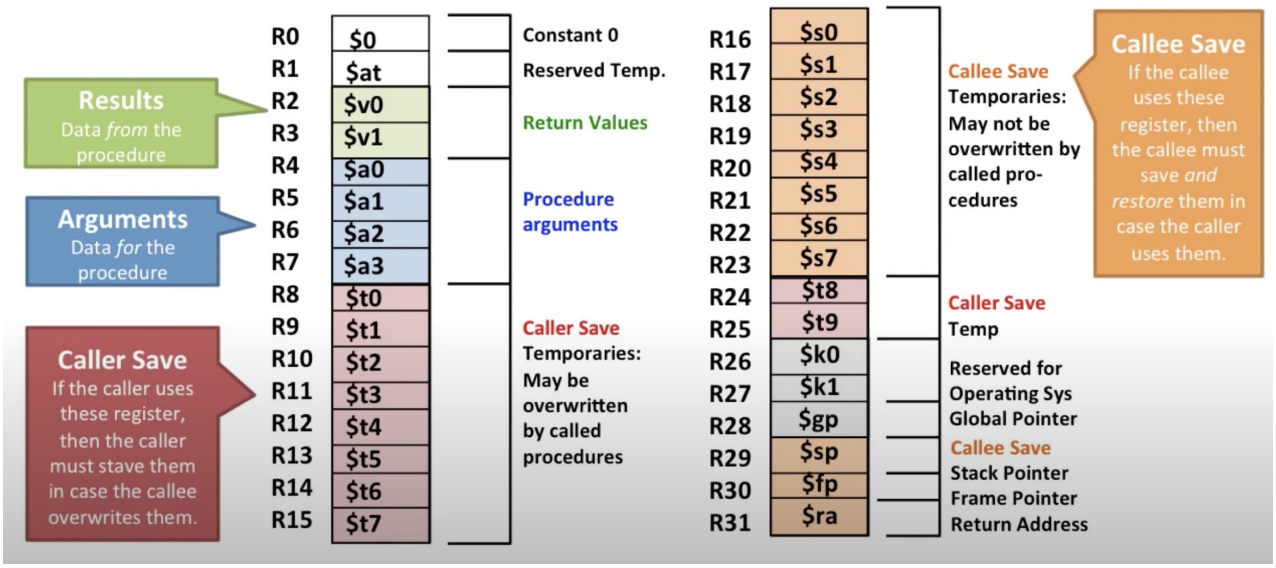
- Create an array in stack without using `.data`
- This will more be like an uninitialised local variable

```c
int main(void) {
    int squares[10];
    int i = 0;
    while (i < 10) {
        squares[i] = i * i;
        i++;
    }
    i = 0;
    while (i < 10) {
        printf("%d", squares[i]);
        printf("%c",'\n');
        i++;
    }
    return 0;
}
```

```mips
# i in $s0
main:
    addi    $sp, $sp, -40   # 10 int

    li      $s0, 0
loop1:
    bge     $s0, 10, end1

    mul     $t0, $s0, 4
    add     $t0, $t0, $sp

    mul     $t1, $s0, $s0
    sw      $t1, 0($t0)

    addi    $s0, $s0, 1
    j       loop1
end1:

    li      $s0, 0
```

```
20  loop2:
21      bge     $s0, 10, end2
22
23      mul     $t0, $s0, 4
24      add     $t0, $t0, $sp
25
26      lw      $a0, 0($t0)
27      li      $v0, 1
28      syscall
29
30      li      $a0, '\n'
31      li      $v0, 11
32      syscall
33
34      addi    $s0, $s0, 1
35      j       loop2
36  end2:
37      addi    $sp, $sp, 40
38      li      $v0, 0
39      jr      $ra
```

Functions



- Caller saves all the `t` registers
  - The caller cannot assume that the `t` registers still have their initial values after calling a function
  - This is because the callee are free to overwrite the `t` registers
  - Therefore, caller need to save the `t` registers to stack if they need to **reuse** them after the call
- Caller saves its own `$ra`

- Callee saves all the `s` registers
  - Callee is free to use any `t` registers
  - Callee needs to make sure all the `s` registers are restored before it returns

General tips

- Save a value of a register to stack if:
  - You need to **reuse** this value after it has the potential to be modified by other functions.

Pass arguments to and receive returned value from another function

```
main:
    addi    $sp, $sp, -4        #   save the $ra
    sw      $ra, 0($sp)

    li      $a0, 1              #   a0 = 1
    li      $a1, 2              #   a1 = 2

    jal     function            #   call the function
    move    $s0, $v0            #   save the returned value to $s0

    lw      $ra, 0($sp)
    addi    $sp, $sp, 4

    li      $v0, 0
    jr      $ra

function:
    add     $v0, $a0, $a1       #    return a0 + a1;
    jr      $ra
```

```
// simple example of returning a value from a function

#include <stdio.h>

int answer(void);

int main(void) {
    int a = answer();
    printf("%d\n", a);
    return 0;
```

```
}

int answer(void) {
    return 42;
}
```

```
main:                           # int main(void) {

    # save the ra
    addi    $sp, $sp, -4
    sw      $ra, 0($sp)

    li      $a0, 1              #   a0 = 1
    li      $a1, 2              #   a1 = 2

    jal     answer              #   int a = sum();

    move    $a0, $v0
    li      $v0, 1
    syscall

    li      $a0, '\n'
    li      $v0, 11
    syscall

    lw      $ra, 0($sp)
    addi    $sp, $sp, 4

    li      $v0, 0              #   return 0;
    jr      $ra
                                # }

sum:                            # int sum(void) {
    add     $v0, $a0, $a1       #    return a0 + a1;
    jr      $ra
                                # }
```

Multiple layers of functions

```
// example of function calls

#include <stdio.h>

int sum_product(int a, int b);
int product(int x, int y);

```

```c
int main(void) {
    int z = sum_product(10, 12);
    printf("%d\n", z);
    return 0;
}

int sum_product(int a, int b) {
    int p = product(6, 7);
    return p + a + b;
}

int product(int x, int y) {
    return x * y;
}
```

```asm
main:
    # store ra of main
    addi    $sp, $sp, -4
    sw      $ra, 0($sp)

    li      $a0, 10
    li      $a1, 12

    jal     sum_product         # sum_product(10, 12);

    move    $a0, $v0            # printf("%d", z);
    li      $v0, 1
    syscall

    li      $a0, '\n'
    li      $v0, 11
    syscall

    lw      $ra, 0($sp)         # recover $ra from $stack
    addi    $sp, $sp, 4         # move stack pointer back up to what it was
when main called

    li      $v0, 0              # return 0 from function main
    jr      $ra


sum_product:
    # save ra and a0 a1
    addi    $sp, $sp, -12       # move stack pointer down to make room
    sw      $ra, 0($sp)
    sw      $a0, 4($sp)
```

```
    sw      $a1, 8($sp)

    li      $a0, 6
    li      $a1, 7
    jal     product
    move    $t0, $v0

    lw      $a1, 8($sp)         # restore a0 and a1
    lw      $a0, 4($sp)

    add     $v0, $t0, $a0       # calculate the return value
    add     $v0, $v0, $a1

    lw      $ra, 0($sp)
    addi    $sp, $sp, 12

    jr      $ra                 # return from sum_product

product:                        # product doesn't call other functions
    mul     $v0, $a0, $a1       # so it doesn't need to save any registers
    jr      $ra                 # return a0 * a1
```

Recursion

- Save the arguments to stack iff
  - You need to **reuse** them after they have the potential to be modified by other functions.
- Always save its `$ra`

```c
// recursive function which prints first 20 powers of two in reverse
#include <stdio.h>

void two(int i);

int main(void) {
    two(1);
}

void two(int i) {
    if (i < 1000000) {
        two(2 * i);
    }
    printf("%d\n", i);
}
```

```
main:
```

```
    addi    $sp, $sp, -4
    sw      $ra, 0($sp)

    li      $a0, 1
    jal     two

    lw      $ra, 0($sp)
    addi    $sp, $sp, 4


    li      $v0, 0
    jr      $ra

two:

    addi    $sp, $sp, -8
    sw      $ra, 0($sp)
    sw      $a0, 4($sp)

    bge     $a0, 1000000, else
    mul     $a0, $a0, 2
    jal     two

else:
    lw      $a0, 4($sp)
    lw      $ra, 0($sp)
    addi    $sp, $sp, 8

    li      $v0, 1                  # print the argument (a0 is reused)
    syscall

    li      $a0, '\n'
    li      $v0, 11
    syscall

    jr      $ra                     # return
```

Exercise1: my_strlen

NOTE: When dealing with `char`, use `lb` and `sb` instead of `lw` and `sw`!

Otherwise, these errors would occur:

```
Exception occurred at PC=0x0040006c
Exception occurred in loop.s at line: 31: lw      $t1, 0($t0)
Unaligned address in inst/data fetch: 0x10010001
```

```c
// calculate the  length of a string using a strlen like function

#include <stdio.h>

int my_strlen(char *s);

int main(void) {
    int i = my_strlen("Hello");
    printf("%d\n", i);
    return 0;
}

int my_strlen(char *s) {
    int length = 0;
    while (s[length] != 0) {
        length++;
    }
    return length;
}
```

```asm
main:
    addi    $sp, $sp, -4
    sw      $ra, 0($sp)

    la      $a0, string         # int i = my_strlen("Hello");
    jal     my_strlen

    move    $a0, $v0            # printf("%d\n", i);
    li      $v0, 1
    syscall

    li      $a0, '\n'
    li      $v0, 11
    syscall

    lw      $ra, 0($sp)
    addi    $sp, $sp, 4

    li      $v0, 0
    jr      $ra

my_strlen:
    # length in $t4
    move    $t0, $a0            # address of s in $t0
    li      $t4, 0              # int length = 0;
loop1:
    add     $t1, $t0, $t4       # calculate &s[length]
    lb      $t2, 0($t1)
```

```
    beq     $t2, 0, end1        # while (s[length] != 0) {

    addi    $t4, $t4, 1         #   length++;
    j       loop1
end1:                           # }

    move    $v0, $t4
    jr      $ra

.data
string:
    .asciiz "Hello"
```