5118007-02 Computer Architecture

Ch. 2 Instructions: Language of the Computer

2 Apr 2024

Shin Hong

Bit-wise Operations

- Bit-wise operations are to operate on a part of word (e.g., 8 bits, 1 bit)
  - extract a third byte in a word
  - read a specific flag in a byte, and turn it on and off

- Bit-wise logical operators

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

Shift

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

- Move all the bits in a word to the left (to most-significant bit) or to the right (least-significant bit)
  - `sll` <$d>, <$s>, <n> # $d = $s << n
  - `srl` <$d>, <$s>, <n> # $s = $s >> n
  - Ex.

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001_{two} = 9_{ten}$$

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 0000_{two} = 144_{ten}$$

Bit-wise AND

- Bit-by-bit operation that produces 1 if and only if both bits of the operands are 1
  - `and` <$d> <$r1> <$r2>
  - used for reading a specific bit with a bitmask
- Ex.
  - `$t2 :` 0000 0000 0000 0000 0000 **1101 1100** 0000
  - `$t1 :` 0000 0000 0000 0000 00**11 1100** 0000 0000
  - `$t1 AND $t2:` 0000 0000 0000 0000 0000 **1100** 0000 0000

Bit-wise OR

- Bit-by-bit operation that produces 1 if and only if either bits of the operands is 1
  - `or` <$d> <$r1> <$r2>
  - used for writing specific bits with a bitmask
- Ex.
  - `$t2 :` 0000 0000 0000 0000 0000 **1101 1100** 0000
  - `$t1 :` 0000 0000 0000 0000 00**11 1100** 0000 0000
  - `$t1 OR $t2:` 0000 0000 0000 0000 00**11 1101 1100** 0000

Character Representation

- American Standard Code for Information Interchange (ASCII) is widely used as character encoding
  - a character takes 7-bits
- A string is represented as an array of characters ended with null

Dec	Hex	Name	Char	Ctrl-char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	Null	NUL	CTRL-@	32	20	Space	64	40	@	96	60	`
1	1	Start of heading	SOH	CTRL-A	33	21	!	65	41	A	97	61	a
2	2	Start of text	STX	CTRL-B	34	22	"	66	42	B	98	62	b
3	3	End of text	ETX	CTRL-C	35	23	#	67	43	C	99	63	c
4	4	End of xmit	EOT	CTRL-D	36	24	$	68	44	D	100	64	d
5	5	Enquiry	ENQ	CTRL-E	37	25	%	69	45	E	101	65	e
6	6	Acknowledge	ACK	CTRL-F	38	26	&	70	46	F	102	66	f
7	7	Bell	BEL	CTRL-G	39	27	'	71	47	G	103	67	g
8	8	Backspace	BS	CTRL-H	40	28	(	72	48	H	104	68	h
9	9	Horizontal tab	HT	CTRL-I	41	29	)	73	49	I	105	69	i
10	0A	Line feed	LF	CTRL-J	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	VT	CTRL-K	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	FF	CTRL-L	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage feed	CR	CTRL-M	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	SO	CTRL-N	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	SI	CTRL-O	47	2F	/	79	4F	O	111	6F	o
16	10	Data line escape	DLE	CTRL-P	48	30	0	80	50	P	112	70	p
17	11	Device control 1	DC1	CTRL-Q	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	DC2	CTRL-R	50	32	2	82	52	R	114	72	r
19	13	Device control 3	DC3	CTRL-S	51	33	3	83	53	S	115	73	s
20	14	Device control 4	DC4	CTRL-T	52	34	4	84	54	T	116	74	t
21	15	Neg acknowledge	NAK	CTRL-U	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	SYN	CTRL-V	54	36	6	86	56	V	118	76	v
23	17	End of xmit block	ETB	CTRL-W	55	37	7	87	57	W	119	77	w
24	18	Cancel	CAN	CTRL-X	56	38	8	88	58	X	120	78	x
25	19	End of medium	EM	CTRL-Y	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	SUB	CTRL-Z	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	ESC	CTRL-[	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	FS	CTRL-\	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	GS	CTRL-]	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	RS	CTRL-^	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	US	CTRL-_	63	3F	?	95	5F	_	127	7F	DEL

Jump for Branching and Looping

- A jump instruction is to determine which instruction will be executed in the next cycle

- Conditional jump (branching)
  - `beq <$r1>, <$r2>, L1    # if (r1 == r2) goto L1 ;`
  - `bne <$r1>, <$r2>, L1    # if (r1 != r2) goto L1 ;`

- Unconditional jump
  - `j L1                    # goto L1 ;`

Example

```
if (e == f) {
    a = b + c ;
}
else {
    a = b - c ;
}
```

```
bne $s3, $s4, Else
add $s0, $s1, $s2
j Exit
Else:
sub $s0, $s1, $s2
Exit:
```

Example

```
int a[100] ;
(…)
while (a[i] == k) {
  i += 1 ;
}
```

```
# $s3:i, $s5:k, $s6:a
Loop:
sll $t1, $s3, 2
add $t1, $t1, $s6
lw $t0, 0($t1)
bne $t0, $s5, Exit
addi $s3, $s3, 1
j Loop
Exit:
```

Test for Inequality

- Set-on-less-than instruction
  - `slt <$t0>, <$s3>, <$s4>`
    - `$t0` will be 1 if `$s3` < `$s4`; 0, otherwise.
  - `slti <$t0>, <$s3>, <n>`
  - `stlu <$t0>, <$s3>, <$s4>`
    - `test $s3` < `$s4` as unsigned values
    - ex. `$s3 : 0000 0000 0000 0000 0000 1101 1100 0000`

      `$s4 : 1111 1111 1111 1111 1111 1111 1111 1111`

Procedure Call (1/2)

- A procedure (or function) is a sequence of instructions that provides a specific functionality

- Jump-and-link instruction
  - `jal <L1>`
    - jump to L1, the starting location of a target procedure while automatically storing the location of the next instruction (i.e., PC+4) at `$ra`
  - `jr $ra`
    - come back to the origin

Procedure Call (2/2)

1. put parameters in a place where the procedure accesses
   - `$a0-$a3` : four arguments
2. jump to the beginning of the procedure
3. acquire the storage resources needed for the procedure
4. perform the desired task
5. put the result value when the caller accesses
   - `$v0-$v1`: return values
6. return the control back to the origin

Stack

- Allocate memory for a procedure execution using a special region called *stack*
  - stack pointer (`$sp`): a register indicating the top
  - push : to place given data to the stack
  - pop: to remove stored data from the stack
- The stack grows downward
  - initially, `$sp` is assigned with the greatest address
  - decrease `$sp` by the number of bytes that a procedure uses
  - refer each local variable with `$sp` as base

Example: strcpy

```
void
strcpy (char * x, char * y) {
  int i ;
  i = 0 ;
  while ((x[i] = y[i]) != '\0') {
    i += 1 ;
  }
}
```

```
strcpy:
  addi $sp, $sp, -4
  sw $s0, 0($sp)
  add $s0,$zero,$zero
L1:
  add $t1,$s0,$a1
  lbu $t2,0($t1)
  add $t3,$s0,$a0
  sb $t2,0($t3)
  beq $t2,$zero,L2
  addi $s0,$s0,1
  j L1:
L2:
  lw $s0,0($sp)
  addi $sp,$sp,4
  jr $ra
```

Nested Procedure Call

- What if a called function invokes another function?
  - conflicts in register

- Push all registers that must be preserved onto the stack
  - a caller pushes argument registers and temporary registers needed after the call
  - a callee pushes the return address register, and any saved registers that will be used by the callee
  - At the return, the registers are restored from memory

Stack

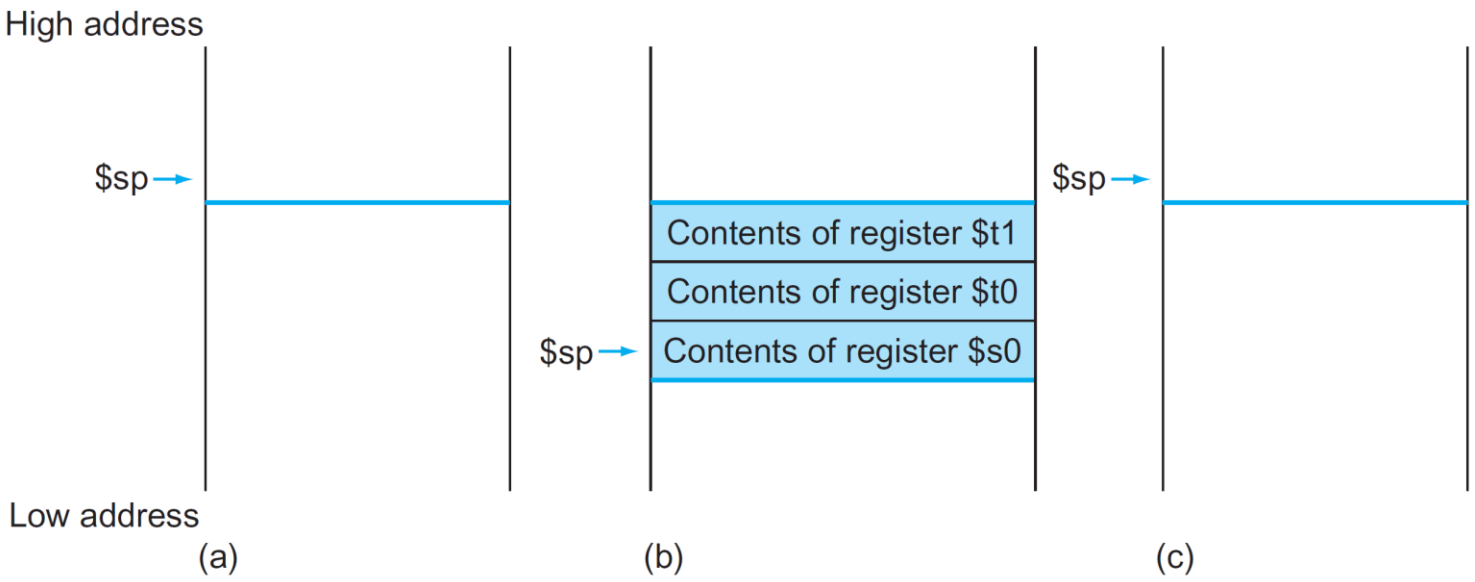


FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

Example

```
int fact (int n)
{
    if (n < 1) return (1);
          else return (n * fact(n - 1));
}
```

```
fact:
    addi  $sp, $sp, -8 # adjust stack for 2 items
    sw    $ra, 4($sp)  # save the return address
    sw    $a0, 0($sp)  # save the argument n

    slti  $t0,$a0,1     # test for n < 1
    beq   $t0,$zero,L1  # if n >= 1, go to L1

    addi  $v0,$zero,1 # return 1
    addi  $sp,$sp,8   # pop 2 items off stack
    jr    $ra         # return to caller

L1: addi $a0,$a0,-1  # n >= 1: argument gets (n - 1)
    jal fact         # call fact with (n -1)
    lw   $a0, 0($sp)  # return from jal: restore argument n
    lw   $ra, 4($sp)  # restore the return address
    addi $sp, $sp, 8  # adjust stack pointer to pop 2 items

    mul  $v0,$a0,$v0   # return n * fact (n - 1)
    jr   $ra           # return to the caller
```

Calling Convention

Preserved	Not preserved
Saved registers: $s0–$s7	Temporary registers: $t0–$t9
Stack pointer register: $sp	Argument registers: $a0–$a3
~~Return address register: $ra~~	Return value registers: $v0–$v1
Stack above the stack pointer	Stack below the stack pointer

- Preserved: what the callee should store (or do not change)
- Not preserved: what the caller should store

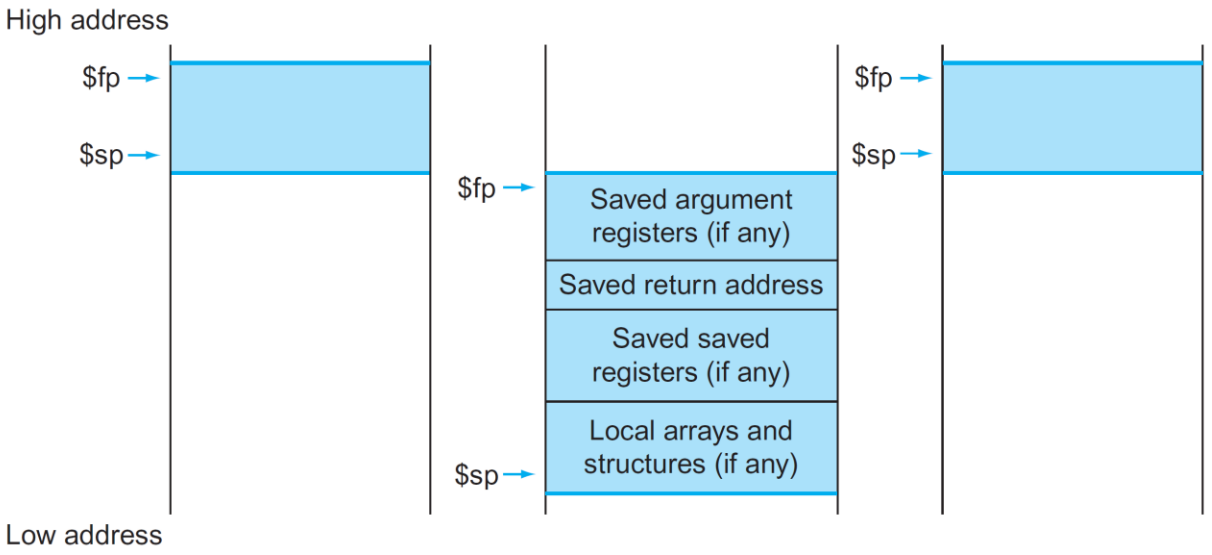
Example

```
int fact (int n)
{
    if (n < 1) return (1);
          else return (n * fact(n - 1));
}
```

```
fact:
    addi  $sp, $sp, -8 # adjust stack for 2 items
    sw    $ra, 4($sp)  # save the return address
    sw    $a0, 0($sp)  # save the argument n

    slti  $t0,$a0,1     # test for n < 1
    beq   $t0,$zero,L1  # if n >= 1, go to L1

    addi  $v0,$zero,1 # return 1
    addi  $sp,$sp,8   # pop 2 items off stack
    jr    $ra         # return to caller

L1: addi $a0,$a0,-1  # n >= 1: argument gets (n - 1)
    jal fact         # call fact with (n -1)
    lw   $a0, 0($sp)  # return from jal: restore argument n
    lw   $ra, 4($sp)  # restore the return address
    addi $sp, $sp, 8  # adjust stack pointer to pop 2 items

    mul  $v0,$a0,$v0   # return n * fact (n - 1)
    jr   $ra           # return to the caller
```

Allocating Local Variables

- Procedure frame (activation record)
  - the space between the first stack pointer and the new stack pointer in a procedure call
  - the frame pointer register (`$fp`) may be used to keep the first stack pointer value if the stack pointer changes during a procedure call



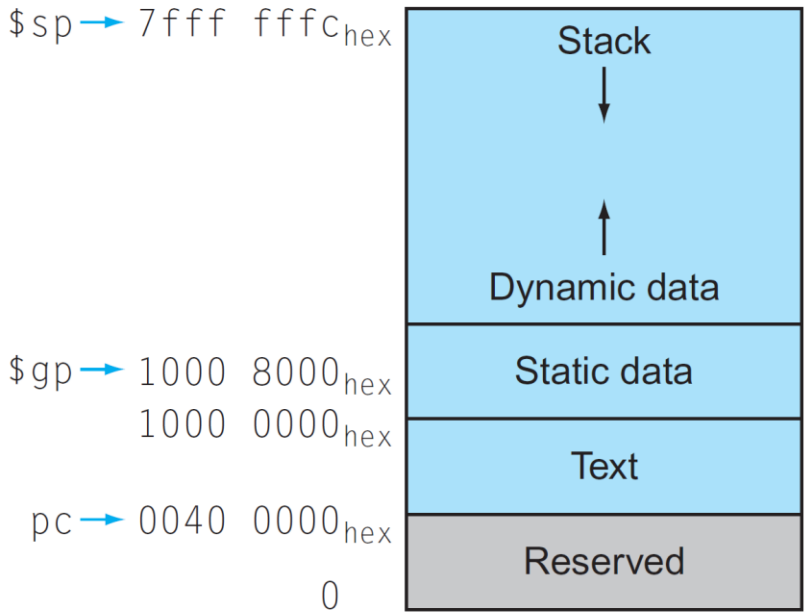
Example

```
int leaf_example
(int g, int h, int i, int j)
{
      int f;

      f = (g + h) - (i + j);
      return f;
}
```

```
leaf_example:

addi $sp, $sp, -12  # adjust stack to make room for 3 items
sw  $t1, 8($sp)     # save register $t1 for use afterwards
sw  $t0, 4($sp)     # save register $t0 for use afterwards
sw  $s0, 0($sp)     # save register $s0 for use afterwards

add $t0,$a0,$a1 # register  $t0 contains g + h
add $t1,$a2,$a3 # register  $t1 contains i + j
sub $s0,$t0,$t1 # f = $t0 - $t1, which is (g + h)-(i + j)

add $v0,$s0,$zero # returns f ($v0 = $s0 + 0)


lw $s0, 0($sp)  # restore register $s0 for caller
lw $t0, 4($sp)  # restore register $t0 for caller
lw $t1, 8($sp)  # restore register $t1 for caller
addi $sp,$sp,12 # adjust stack to delete 3 items

jr   $ra    # jump back to calling routine
```

Memory Layout

- text segment
- static data
- dynamic data (heap)
- stack



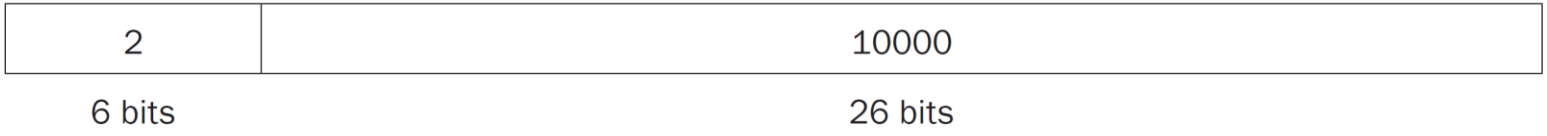
32-bit Immediate Operands

- Constants are commonly short, sometimes big
- MIPS allows 32-bit constants to span over two load-immediate instructions
- Examples

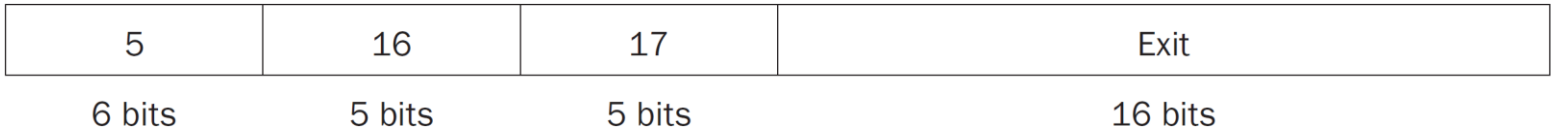
```
0000 0000 0011 1101 0000 1001 0000 0000

lui $s0, 61   # 61 decimal = 0000 0000 0011 1101 binary

ori $s0, $s0, 2304 # 2304 decimal = 0000 1001 0000 0000
```

Addressing in Jumps

- J-type (unconditional jump) uses 26-bits to represents an address of instruction



- Conditional jump uses only 16 bits for relative address
  - PC-relative address: word distance from the current PC



Example

```
Loop:sll $t1,$s3,2      # Temp reg $t1 = 4 * i
     add  $t1,$t1,$s6   # $t1 = address of save[i]
     lw   $t0,0($t1)    # Temp reg $t0 = save[i]
     bne  $t0,$s5, Exit # go to Exit if save[i] ≠ k
     addi $s3,$s3,1     # i = i + 1
     j    Loop          # go to Loop
Exit:
```

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024	...					